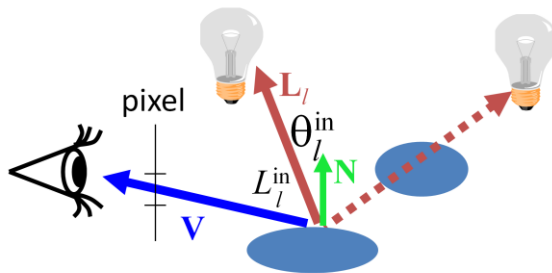


**Sugárkövetés:
ray-casting, ray-tracing, path-tracing**

Szirmay-Kalos László

Lokális illumináció: rücskös felületek, absztrakt fényforrások

Csak absztrakt
fényforrások
direkt megvilágítása



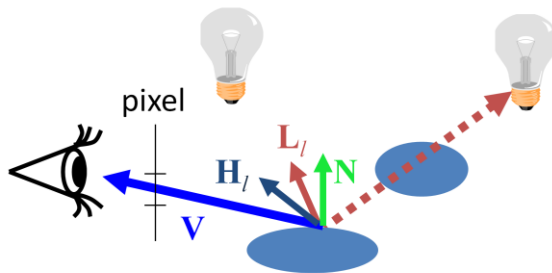
$$L(\mathbf{V}) \approx \sum_l L_l^{\text{in}} * f_r(\mathbf{L}_l, \mathbf{N}, \mathbf{V}) \cdot \cos^+ \theta_l^{\text{in}}$$

Absztrakt fényforrásokból származó megvilágítás.
(Irányforrás = konstans; Pontforrás = távolság négyzetével csökken
Ha takart, akkor zérus)

In local illumination rendering, having identified the surface visible in a pixel, we have to compute the reflected radiance due to only the few abstract light sources. An abstract light source may illuminate a point just from a single direction. The intensity provided by the light source at the point is multiplied by the BRDF and the geometry term (cosine of the angle between the surface normal and the illumination direction). The intensity provided by the light source is zero if the light source is not visible from the shaded point. For directional sources, the intensity and the direction are the same everywhere. For point sources, the direction is from the source to the shaded point and decreases with the square of the distance.

Lokális illumináció: rücskös felületek, absztrakt fényforrások

Csak absztrakt
fényforrások
direkt megvilágítása



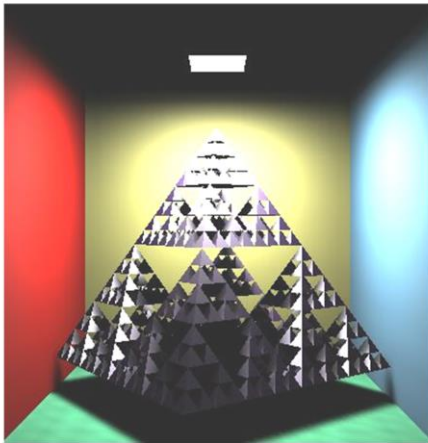
$$L(\mathbf{V}) \approx \sum_l L_l^{\text{in}} * \{k_d \cdot (\mathbf{L}_l \cdot \mathbf{N})^+ + k_s \cdot ((\mathbf{H}_l \cdot \mathbf{N})^+)^{\text{shine}}\}$$

Absztrakt fényforrásokból származó megvilágítás.
(Irányforrás = konstans; Pontforrás = távolság négyzetével csökken
Ha takart, akkor zérus)

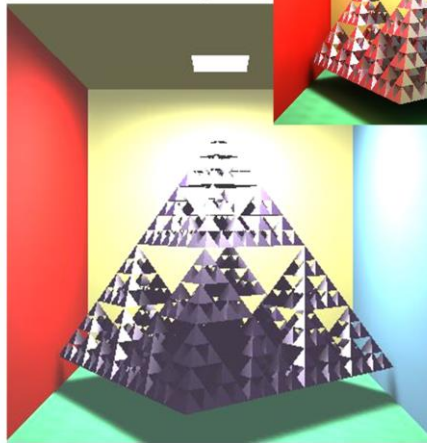
The BRDF times the geometry factor equals to the following expression for diffuse + Phong-Blinn type materials. Here we use different product symbols for different data types; * for spectra, · for multiplying with a scalar, and • for the dot product of two vectors.

Ambiens tag

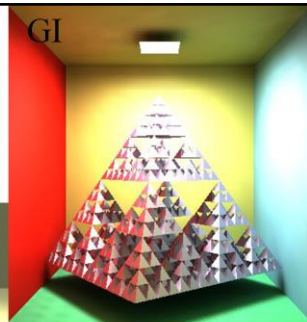
Lokális illumináció



+ ambiens tag



GI

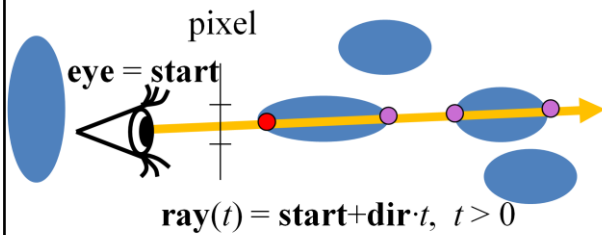


$$L(\mathbf{V}) \approx \sum_l L_l^{\text{in}} * f_r(\mathbf{L}_l, \mathbf{N}, \mathbf{V}) \cdot \cos^+\theta_l^{\text{in}} + \boxed{k_a * L_a}$$

In the local illumination model all surfaces that are not directly visible from the light sources are completely black. However, this is against everyday experience when some light indirectly illuminates even hidden regions as well. So, we add an ambient term to the reflected radiance, where the intensity is uniform everywhere and in all directions, and the ambient reflection k_a is a material property (if a physically accurate model is applied, $k_a = k_d * \pi$).

Note, however, that adding the ambient term is a very crude approximation of true indirect lighting, which can be obtained by global illumination algorithms (upper right image).

Láthatóság



```
struct Ray {
    vec3 start;
    vec3 dir; // egységvektor
    bool out; // kívül?
};

struct Hit {
    float t;
    vec3 position;
    vec3 normal;
    Material* material;
    Hit() { t = -1; }
};
```

```
Hit firstIntersect(Ray ray) {
    Hit bestHit;
    for(Intersectable * obj : objects) {
        Hit hit = obj->intersect(ray); // hit.t < 0 ha nincs metszés
        if(hit.t > 0 && (bestHit.t < 0 || hit.t < bestHit.t))
            bestHit = hit;
    }
    if (dot(ray.dir, bestHit.normal) > 0) bestHit.normal *= -1;
    return bestHit;
}
```

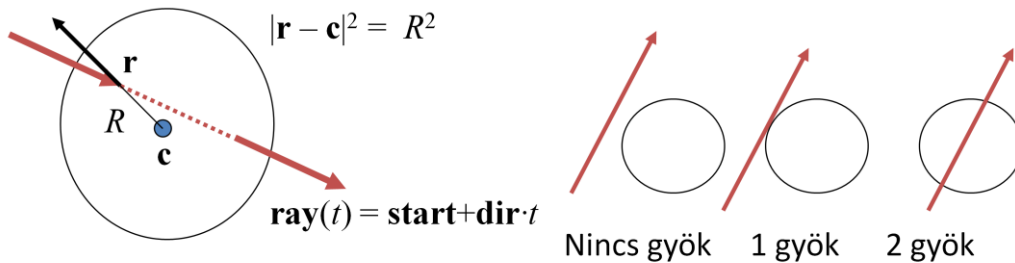
Nézzen felénk!

if ($\mathbf{N} \cdot \mathbf{dir} > 0$) { $\mathbf{N} = -\mathbf{N}$ }

A fundamental operation of ray tracing is the identification of the surface point hit by a ray. The ray may be a primary ray originating at the eye and passing through the pixel, it can be a shadow ray originating at the shaded point and going towards the light source, or even a secondary ray that also originates in the shaded point but goes into either the reflection or the refraction direction. The intersection with this ray is on the ray, thus it satisfies the ray equation, $\text{ray}(t) = \text{eye} + vt$ for some POSITIVE ray parameter t , and at the same time, it is also on the visible object, so point $\text{ray}(t)$ also satisfies the equation of the surface. A ray may intersect more than one surface, when we need to obtain the intersection of minimal positive ray parameter since this is the closest surface that occludes others.

Function FirstIntersect finds this point by trying to intersect every surface with function Intersect and always keeping the minimum, positive ray parameter t .

Metszéspont számítás gömbbel



$$|\text{ray}(t) - \mathbf{c}|^2 = (\text{start} + \mathbf{dir} \cdot t - \mathbf{c}) \bullet (\text{start} + \mathbf{dir} \cdot t - \mathbf{c}) = R^2$$

$$(\mathbf{dir} \bullet \mathbf{dir})t^2 + 2((\text{start} - \mathbf{c}) \bullet \mathbf{dir})t + (\text{start} - \mathbf{c}) \bullet (\text{start} - \mathbf{c}) - R^2 = 0$$

Wanted: a pozitív megoldások közül a kisebb

Felületi normális: $\mathbf{N} = (\text{ray}(t) - \mathbf{c})/R$

The implementation of function Intersect depends on the actual type of the surface, since it means the inclusion of the ray equation into the equation of the surface. The first example is the sphere. Substituting the ray equation into the equation of the sphere and taking advantage of the distributivity of the scalar product, we can establish a second order equation for unknown ray parameter t . A second order equation may have zero, one or two real roots (complex roots have no physical meaning here), which corresponds to the cases when the ray does not intersect the sphere, the ray is tangent to the sphere, and when the ray intersects the sphere in two points, entering then leaving it. From the roots, we need the smallest positive one.

Recall that we also need the surface normal at the intersection point. For a sphere, the normal is parallel to the vector pointing from the center to the surface point. It can be normalized, i.e. turned to a unit vector, by dividing by its length, which equals to the radius of the sphere.

Sphere as Intersectable

```
struct Intersectable {  
    Material* material;  
    virtual Hit intersect(const Ray& ray)=0;  
};
```

```
class Sphere : public Intersectable {  
    vec3 center;  
    float radius;  
public:  
    Hit intersect(const Ray& ray) {  
        Hit hit;  
        vec3 dist = ray.start - center;  
        float a = dot(ray.dir, ray.dir);  
        float b = dot(dist, ray.dir) * 2;  
        float c = dot(dist, dist) - radius * radius;  
        float discr = b * b - 4 * a * c;  
        if (discr < 0) return hit; else discr = sqrtf(discr);  
        float t1 = (-b + discr)/2/a, t2 = (-b - discr)/2/a;  
        if (t1 <= 0) return hit; // t1>=t2 for sure  
        hit.t = (t2 > 0) ? t2 : t1;  
        hit.position = ray.start + ray.dir * hit.t;  
        hit.normal = (hit.position - center)/radius;  
        hit.material = material;  
        return hit;  
    }  
};
```

Implicit felületek

- A felület pontjai: $f(x,y,z) = 0$ vagy $f(\mathbf{r}) = 0$
- A sugár pontjai: $\mathbf{ray}(t) = \mathbf{start} + \mathbf{dir} \cdot t$
- A metszés pont: $f(\mathbf{ray}(t)) = 0$,
 - 1 ismeretlenes, ált. nemlineáris egyenlet: t^*
 - $(x^*, y^*, z^*) = \mathbf{start} + \mathbf{dir} \cdot t^*$
- Normálvektor = $\text{grad } f \Big|_{x^*, y^*, z^*}$
 - $0 = f(x,y,z) = f(x^* + (x-x^*), y^* + (y-y^*), z^* + (z-z^*)) \approx$
 $f(x^*, y^*, z^*) + \frac{\partial f}{\partial x}(x-x^*) + \frac{\partial f}{\partial y}(y-y^*) + \frac{\partial f}{\partial z}(z-z^*)$

Az érintősík
egyenlete:

$$\left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) \bullet (x-x^*, y-y^*, z-z^*) = 0$$

$$\mathbf{n} \bullet (\mathbf{r} - \mathbf{r0}) = 0$$

The equation of the sphere is an example of a more general category, the implicit surfaces that are defined by an implicit equation of the x,y,z Cartesian coordinates of the place vectors \mathbf{r} of surface points. Substituting the ray equation into this equation, we obtain a single, usually non-linear equation for the single unknown, the ray parameter t . Having solved this equation, we can substitute the ray parameter t^* into the equation of the ray to find the intersection point.

The normal vector of the surface can be obtained by computing the gradient at the intersection point. To prove it, let us express the surface around the intersection point as a Taylor approximation. $f(x^*, y^*, z^*)$ becomes zero since the intersection point is also on the surface. What we get is a linear equation of form $\mathbf{n} \cdot (\mathbf{r} - \mathbf{r0}) = 0$, which is the equation of the plane, where $\mathbf{n} = \text{grad } f$.

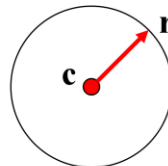
So, the gradient is the normal vector of the plane that approximates the surface locally in the intersection point.

Konkrét példa: gömb normálvektora

$$|\mathbf{r} - \mathbf{c}|^2 = R^2$$

$$|\mathbf{r} - \mathbf{c}|^2 - R^2 = 0$$

$$f(\mathbf{r}) = 0$$



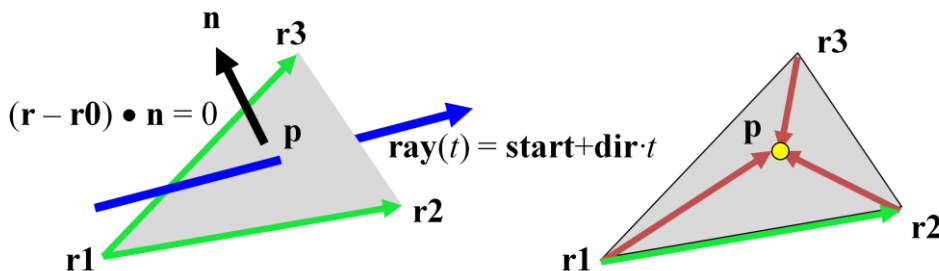
$$f(x,y,z) = (x-c_x)^2 + (y-c_y)^2 + (z-c_z)^2 - R^2 = 0$$

$$\frac{\partial f}{\partial x} = 2(x - c_x) + 0 + 0 - 0$$

$$\frac{\partial f}{\partial x} = 2(x - c_x) \quad \frac{\partial f}{\partial y} = 2(y - c_y) \quad \frac{\partial f}{\partial z} = 2(z - c_z)$$

$$\text{grad } f(x,y,z) = 2 (x-c_x, y-c_y, z-c_z)$$

Háromszög



1. Síkmetszés: $(\text{ray}(t) - \mathbf{r1}) \cdot \mathbf{n} = 0, \quad t > 0$
normál: $\mathbf{n} = (\mathbf{r2} - \mathbf{r1}) \times (\mathbf{r3} - \mathbf{r1})$

$$t = \frac{(\mathbf{r1} - \text{start}) \cdot \mathbf{n}}{\text{dir} \cdot \mathbf{n}}$$

2. A metszéspont a háromszögön belül van-e?

$$\begin{aligned} ((\mathbf{r2} - \mathbf{r1}) \times (\mathbf{p} - \mathbf{r1})) \cdot \mathbf{n} &> 0 \\ ((\mathbf{r3} - \mathbf{r2}) \times (\mathbf{p} - \mathbf{r2})) \cdot \mathbf{n} &> 0 \\ ((\mathbf{r1} - \mathbf{r3}) \times (\mathbf{p} - \mathbf{r3})) \cdot \mathbf{n} &> 0 \end{aligned}$$

Felületi normális: \mathbf{n}
vagy árnyaló normálok
(shading normals)

The triangle is the most important primitive because we often use it to approximate arbitrary surfaces. So effective ray-triangle intersection algorithms are still in the focus of research. Now, we present a very simple algorithm, which is far behind the leading methods in terms of efficiency.

The algorithm consists of two steps, first the intersection with the plane of the triangle is found, then we determine whether or not the ray-plane intersection point is inside the triangle. Suppose that the triangle is given by its vertices $\mathbf{r1}$, $\mathbf{r2}$, $\mathbf{r3}$. The equation of its plane is $\mathbf{n} \cdot (\mathbf{r} - \mathbf{r0}) = 0$ where \mathbf{n} is the normal vector and $\mathbf{r0}$ is a point of the plane. Place vector $\mathbf{r0}$ can be any of the three vertices and normal vector \mathbf{n} can be computed as the cross product of edge vectors $\mathbf{r2} - \mathbf{r1}$ and $\mathbf{r3} - \mathbf{r1}$. Substituting the ray equation into this linear equation, we get a linear equation for t , which can be solved. If t is negative, the intersection is behind the eye, so it must be ignored. The positive t is substituted back to the ray equation giving \mathbf{p} as the intersection with the plane.

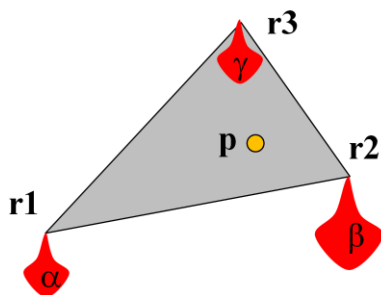
Now we should determine whether \mathbf{p} is inside the triangle. An edge line separates the plane into two half planes, a “good” or one (this is the left one if the edge vector points from $\mathbf{r1}$ to $\mathbf{r2}$) that contains the triangle and the third vertex and a “bad” one that contains nothing. Point \mathbf{p} must be on the good side, i.e. where the third vertex is. Points on the left and right with respect to edge $\mathbf{r1} - \mathbf{r2}$ can be separated using the properties of the cross product.

Assuming that we look at the plane from above, $(\mathbf{r2} - \mathbf{r1}) \times (\mathbf{p} - \mathbf{r1})$ will point towards us if \mathbf{p} is on the left, and it will point down if \mathbf{p} is on the right.

As $\mathbf{n} = (\mathbf{r2} - \mathbf{r1}) \times (\mathbf{r3} - \mathbf{r1})$ points towards us, we can check whether $(\mathbf{r2} - \mathbf{r1}) \times (\mathbf{p} - \mathbf{r1})$ has the same direction by computing their dot product and checking if the result is positive (the dot product of two vectors point into the same direction is positive, the dot product of two oppositely pointing vectors is negative). A single inequality states that the point is on the good side with respect to a

given edge vector. If this condition is met for all three edge vectors, the point is inside the triangle.

Háromszög metszés baricentrikus koordinátákban



$$\mathbf{p}(\alpha, \beta, \gamma) = \alpha \cdot \mathbf{r1} + \beta \cdot \mathbf{r2} + \gamma \cdot \mathbf{r3}$$

$$\alpha + \beta + \gamma = 1 \quad 1 \text{ skalár egyenlet}$$

$$\alpha, \beta, \gamma \geq 0$$

$$\mathbf{ray}(t) = \mathbf{p}(\alpha, \beta, \gamma)$$

$$\mathbf{eye} + \mathbf{v} \cdot t = \alpha \cdot \mathbf{r1} + \beta \cdot \mathbf{r2} + \gamma \cdot \mathbf{r3} \quad 3 \text{ skalár egyenlet}$$

4 ismeretlen: α, β, γ, t

A megoldás után ellenőrzés, hogy mind nem negatív-e

Another popular intersection algorithm is based on the recognition that a triangle is the convex combination of its vertices. Assume that we have unit mass sand, which is distributed into the three vertices putting weight α to $\mathbf{r1}$, β to $\mathbf{r2}$, and γ to $\mathbf{r3}$. The center of mass of this system is

$\mathbf{p}(\alpha, \beta, \gamma) = (\alpha \mathbf{r1} + \beta \mathbf{r2} + \gamma \mathbf{r3}) / (\alpha + \beta + \gamma) = \alpha \mathbf{r1} + \beta \mathbf{r2} + \gamma \mathbf{r3}$ as $\alpha + \beta + \gamma = 1$ since we distributed unit mass. Points \mathbf{p} defined this way are called the combination of points $\mathbf{r1}$, $\mathbf{r2}$, $\mathbf{r3}$. Clearly, such points are on the plane defined by the three points.

If we further assume that the weights are non-negative, the points must lie in the triangle of the three points. Such points are called convex combination.

Making the internal points expressed as convex combinations equal to the ray equation, we obtain a system of four scalar equations (three are the x,y,z components of the vector equation and the fourth is the $\alpha + \beta + \gamma = 1$ requirement), and we have four unknowns. Having solved it, we have to check whether all parameters are non-negative, i.e. the intersection is inside the triangle and not behind the eye.

Parametrikus felületek

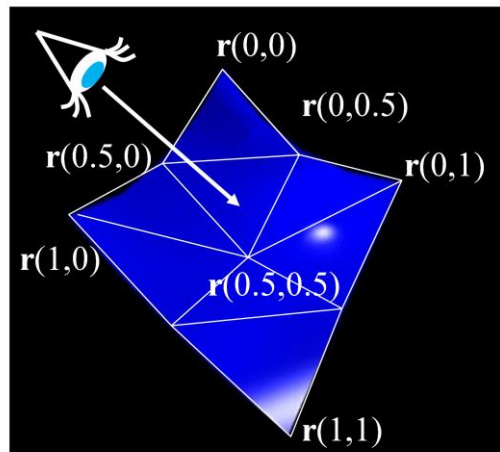
$$\mathbf{r}(u,v), \quad u,v \in [0,1]$$
$$\mathbf{ray}(t) = \mathbf{eye} + \mathbf{v} \cdot t, \quad t > 0$$

$$\mathbf{r}(u,v) = \mathbf{ray}(t)$$

Háromismeretlenes ált.
nem lineáris egyenletrendszer
megoldás: u^*, v^*, t^*

Teszt:

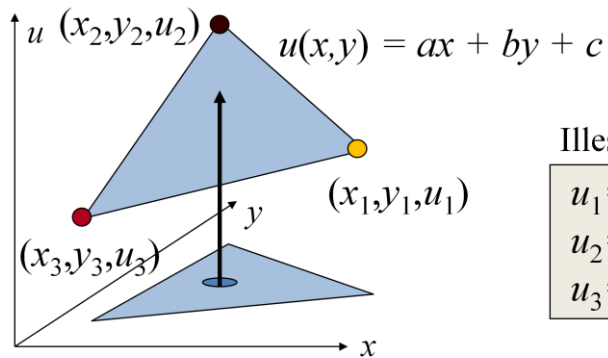
$$0 < u^*, v^* < 1,$$
$$t^* > 0$$



Rekurzív tesszelláció:
nem robusztus

Lineáris interpoláció

- 1. módszer



Illesztés

$$u_1 = ax_1 + by_1 + c$$

$$u_2 = ax_2 + by_2 + c$$

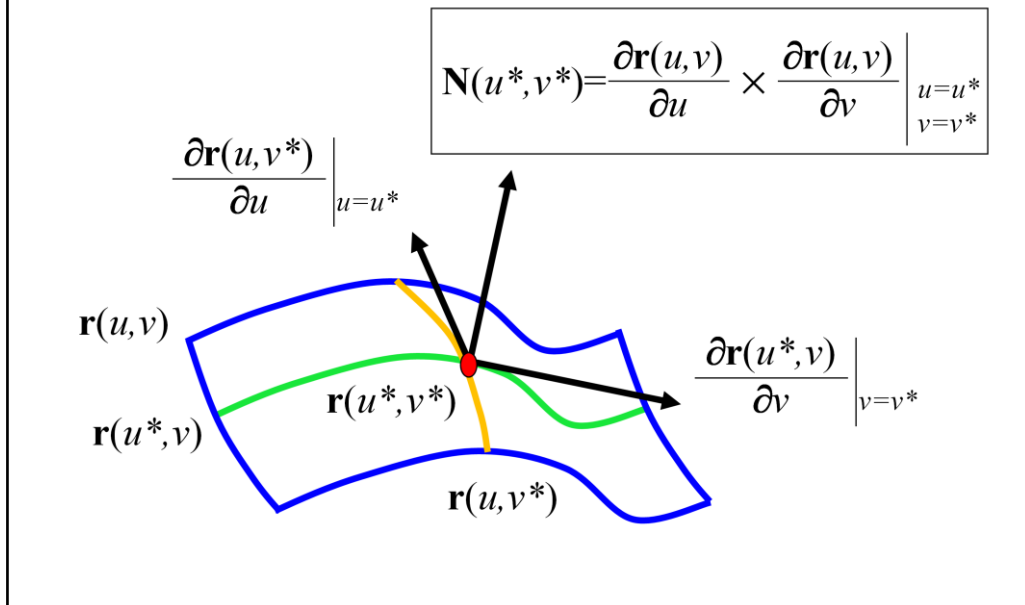
$$u_3 = ax_3 + by_3 + c$$

- 2. módszer

$$u(\alpha, \beta, \gamma) = \alpha \cdot u_1 + \beta \cdot u_2 + \gamma \cdot u_3$$

$$v(\alpha, \beta, \gamma) = \alpha \cdot v_1 + \beta \cdot v_2 + \gamma \cdot v_3$$

Parametrikus felületek normálvektora

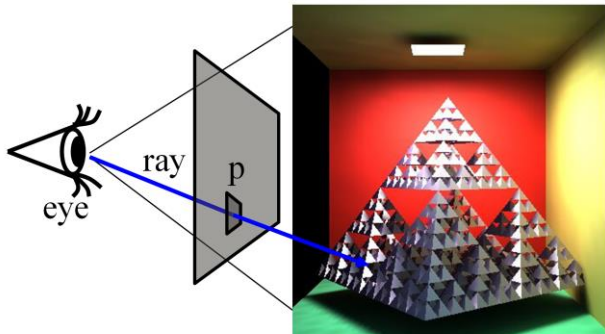


To get the normal vector of a parametric surface, we exploit isoparametric lines. Suppose that we need the normal at point associated with parameters u^* , v^* . Let us keep u^* fixed, but allow v to run over its domain. This $\mathbf{r}(u^*, v)$ is a one-variate parametric function, which is a curve. As it always satisfies the surface equation, this curve is on the surface and when $v=v^*$, this curve passes through the point of interest. We know that the derivative of a curve always tangent to the curve, so the derivative with respect to v at v^* will be the tangent of a curve at this point, and consequently will be in the tangent plane.

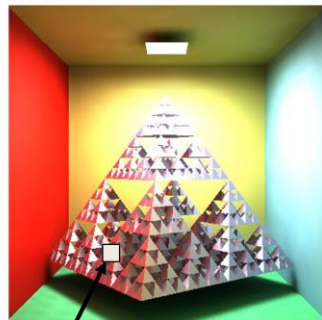
Similarly, the derivative with respect to u will always be in the tangent plane. The cross product results in a vector that is perpendicular to both operands, so it will be the normal of the tangent plane.

Sugárkövetés: Render

Virtuális világ: szem+ablak



Valós világ: néző+képernyő



Render()

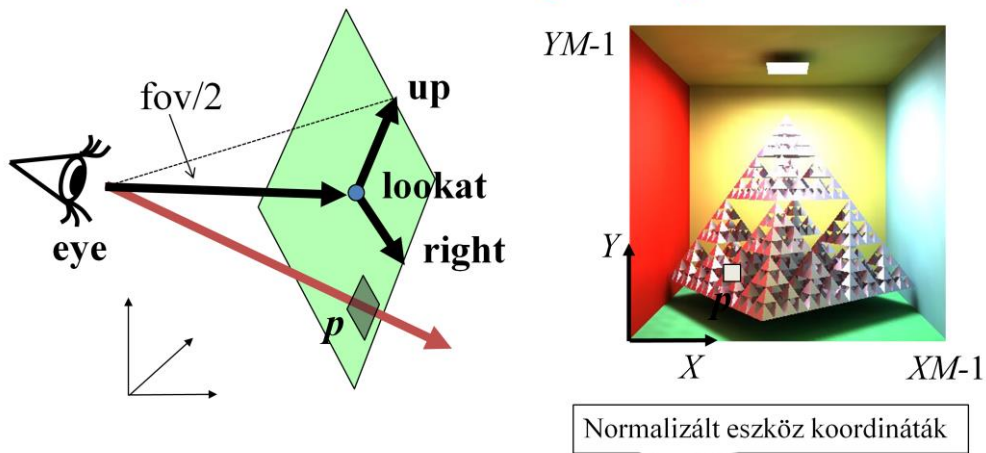
```

for each pixel p
    Ray r = getRay( eye  $\Rightarrow$  pixel p )
    color = trace(ray)
    WritePixel(p, color)
endfor
end
    
```

p color

On the top level, ray tracing rendering visits pixels one by one. For every pixel, the virtual camera has a point on its window (in real space we have the user and the screen; in virtual world one of the user's eye is the virtual eye and the display surface is a rectangle). The origin of primary rays is always the eye position. The direction of a ray is from the eye to the center of the pixel on the window rectangle, which is calculated by the GetRay function. With this ray, function Trace is called, which computes the radiance transferred back by this ray (i.e. the radiance of the point hit by this ray in the opposite of the ray direction). The radiance on the wavelengths of r,g,b is written into the current physical pixel.

Kamera: getRay



$$\begin{aligned}
 p &= \text{lookat} + \alpha \cdot \text{right} + \beta \cdot \text{up}, \\
 &= \text{lookat} + (2(X+0.5)/XM-1) \cdot \text{right} + (2(Y+0.5)/YM-1) \cdot \text{up}
 \end{aligned}$$

$\alpha, \beta \in [-1, 1]$

Ray: start = eye, dir = p – eye

To implement the GetRay function, the virtual camera should be defined in the virtual space. The user's location is specified by the place vector called eye. The display surface is represented by a 2D rectangle in the virtual world coordinates. The center of this rectangle is specified by the lookat point, and its orientation and size are defined by two vectors. Right points from the center of the window to the right edge, up from the center to the top edge. If the resolution of the target image is XM x YM, then the center of pixel (X,Y) in world coordinates is $p = \text{lookat} + (2X/XM-1) \cdot \text{right} + (2Y/YM-1) \cdot \text{up}$.

No Shadow
Shadow
 $\varepsilon = 0$: acne

```

vec3 trace(Ray ray) {
    Hit hit = firstIntersect(ray);
    if(hit.t < 0) return L_a; // nothing
    [r, N, k_a, k_d, k_s, shine] ← hit;
    vec3 outRad = k_a * L_a;
    for(each light source l) {
        Ray shadowRay(r + N*ε, L_l);
        Hit shadowHit = firstIntersect(shadowRay);
        if(shadowHit.t < 0 || shadowHit.t > |r - y_l|)
            outRad += L_l^in * {k_d * (L_l • N)^+ + k_s * ((H_l • N)^+)^shine};
    }
    return outRad;
}

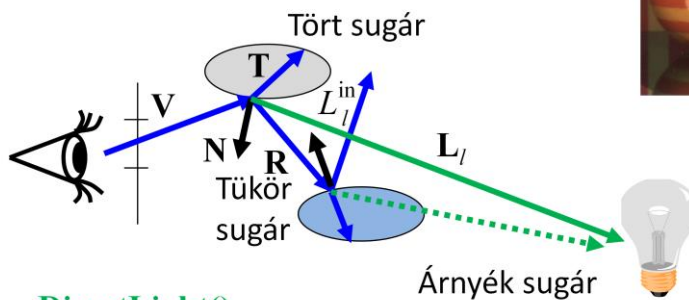
```

The trace function gets the ray that involves its origin and direction vectors. First we compute the intersection that is in front of the eye and is closest to the eye. The already implemented solution is firstIntersect. This function indicates with a negative value if there is no intersection. In this case, trace returns with the radiance of the ambient illumination.

If some surface is seen, trace computes the contribution of the abstract light sources. To check the visibility of a particular light source, a ray, called shadow ray, is sent from the shaded point towards the light source. If this ray intersects an object and this intersection is closer than the light source, the object occludes the light source so this point is in shadow.

If the surface is smooth and is ideally reflective, then the reflection direction is computed and the trace function is called recursively to compute the radiance of reflection direction. The same is done for the refraction direction if the surface is refractive.

Rekurzív sugárkövetés



DirectLight()

$$L(V) \approx \sum_l L_l^{\text{in}} * (k_d \cdot (L_l \cdot N)^+ + k_s \cdot ((H_l \cdot N)^+)^{\text{shine}}) + k_a * L_a$$

$$+ F(V \cdot N) * L^{\text{in}}(R) + (1 - F(V \cdot N)) * L^{\text{in}}(T)$$

Fresnel

Tükör irányból
érkező fény

1-Fresnel

Törési irányból
érkező fény

To simulate also smooth surfaces responsible for mirroring and light refraction, the local illumination model should be extended. When the surface visible from the eye is identified, we calculate the radiance as the contribution from abstract light sources but also add the reflection of the radiance from the ideal reflection direction and the refraction of the radiance coming from the ideal refraction direction. According physics, the scaling factors of the radiance values are the Fresnel and 1-Fresnel for reflection and refraction, respectively. However, we do not always insist of physical precision so may use other scaling factors that are set by an artist and not computed as the Fresnel function.

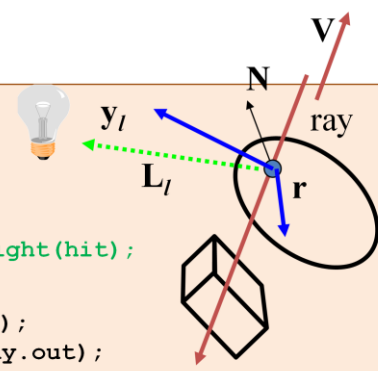
This equation expresses the radiance of a surface point in a given direction as the function of the direct light sources and the radiance coming from the ideal reflection and refraction directions. The question is how these extra terms can be computed.

Let us recognize, that the computation of the radiance delivered back by reflection and refraction rays is essentially the same computation what we are doing right now, just the ray origin and direction should be altered. So the solution of this problem is a recursive function.

trace

```
vec3 trace(Ray ray) {

    Hit hit = firstIntersect(ray);
    if(hit.t < 0) return La; // nothing
    vec3 outRad(0, 0, 0);
    if(hit.material->rough) outRad = DirectLight(hit);
    if(hit.material->reflective) {
        vec3 reflectionDir = reflect(ray.dir, N);
        Ray reflectRay(r+Nε, reflectionDir, ray.out);
        outRad += trace(reflectRay)*Fresnel(ray.dir, N);
    }
    if(hit.material->refractive) {
        ior = (ray.out) ? n.x : 1/n.x;
        vec3 refractionDir = refract(ray.dir, N, ior);
        if (length(refractionDir) > 0) {
            Ray refractRay(r-Nε, refractionDir, !ray.out);
            outRad += trace(refractRay)*(vec3(1,1,1)-Fresnel(ray.dir, N));
        }
    }
    return outRad;
}
```



The trace function gets the ray that involves its origin and direction vectors. First we compute the intersection that is in front of the eye and is closest to the eye. The already implemented solution is firstIntersect. This function indicates with a negative value if there is no intersection. In this case, trace returns with the radiance of the ambient illumination.

If some surface is seen, trace computes the contribution of the abstract light sources. To check the visibility of a particular light source, a ray, called shadow ray, is sent from the shaded point towards the light source. If this ray intersects an object and this intersection is closer than the light source, the object occludes the light source so this point is in shadow.

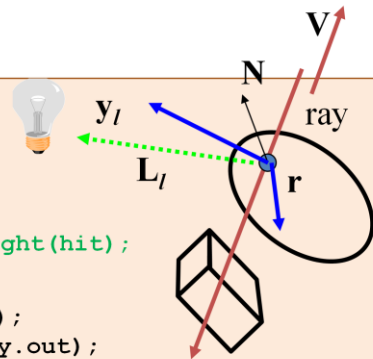
If the surface is smooth and is ideally reflective, then the reflection direction is computed and the trace function is called recursively to compute the radiance of reflection direction. The same is done for the refraction direction if the surface is refractive.

trace

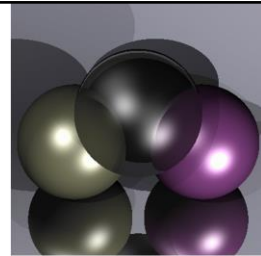
```

vec3 trace(Ray ray, int d=0) {
    if (d > maxdepth) return La;
    Hit hit = firstIntersect(ray);
    if(hit.t < 0) return La; // nothing
    vec3 outRad(0, 0, 0);
    if(hit.material->rough) outRad = DirectLight(hit);
    if(hit.material->reflective){
        vec3 reflectionDir = reflect(ray.dir,N);
        Ray reflectRay(r+Nε, reflectionDir, ray.out);
        outRad += trace(reflectRay,d+1)*Fresnel(ray.dir,N);
    }
    if(hit.material->refractive) {
        ior = (ray.out) ? n.x : 1/n.x;
        vec3 refractionDir = refract(ray.dir,N,ior);
        if (length(refractionDir) > 0) {
            Ray refractRay(r-Nε, refractionDir, !ray.out);
            outRad += trace(refractRay,d+1)*(vec3(1,1,1)-Fresnel(ray.dir,N))
        }
    }
    return outRad;
}

```

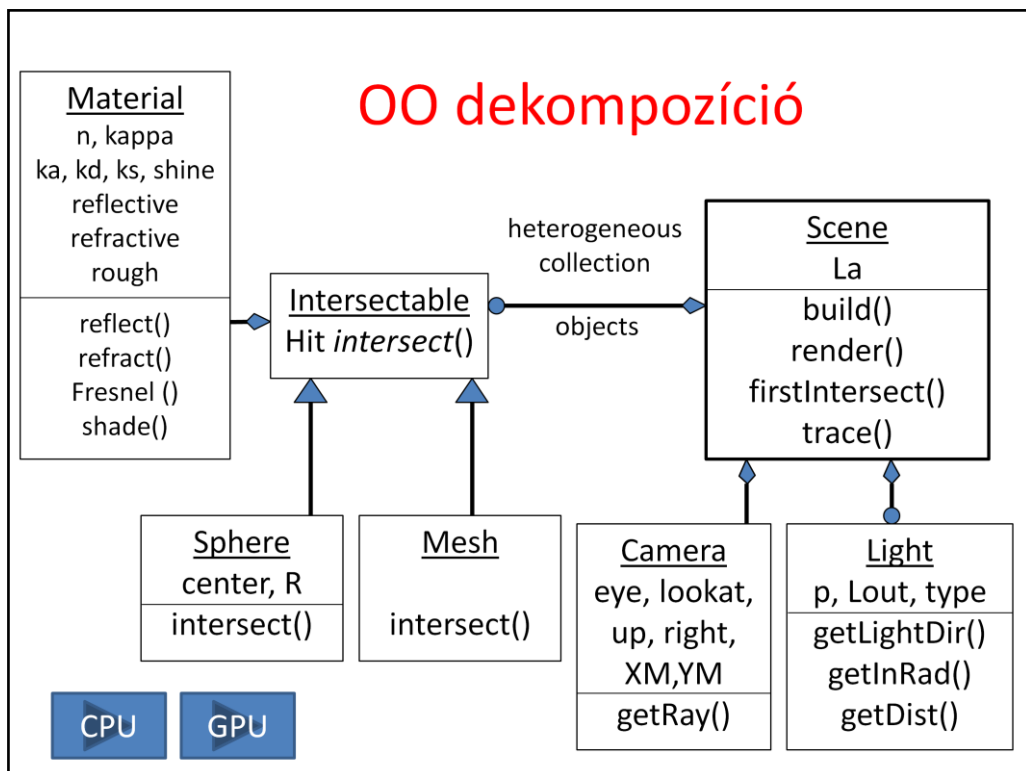


Heckbert Palika házija a névjegyén



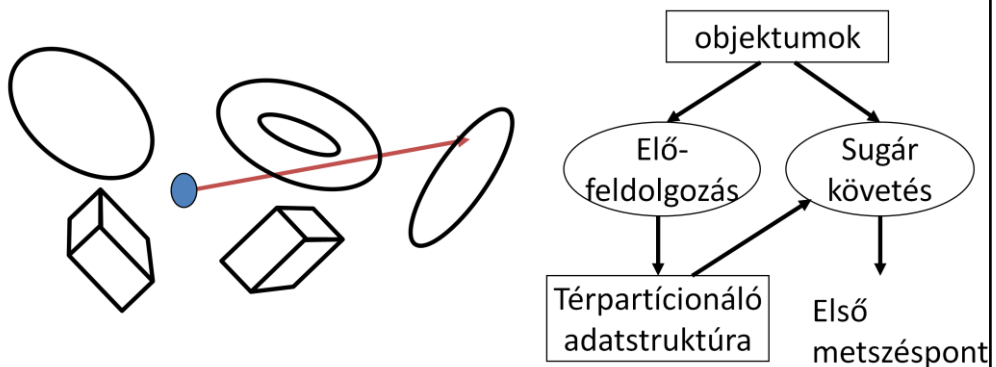
```
typedef struct { double x,y,z; } vec; vec U, black, amb = { .02, .02, .02 }; struct sphere { vec cen, color; double rad, kd, ks, kt, kl, ir; } *s,
*best, sph[] = { { 0., 6., 5.1, 1., 1., 9., .05, 2., 85, 0., 1.7, -1., 8., -5, 1., 5., 2, 1., 7., 3, 0., .05, 1.2, 1., 8., -5., 1., 8., 1., 3., 7, 0., 0., 1.2, 3., -6., 15., 1.,
.8, 1., 7., 0., 0., 0., 6, 1.5, -3., -3., 12., 8, 1., 1., 5., 0., 0., 0., 5, 1.5, }; yx; double u, b, tmin, sqrt(), tan(); double vdot(A, B) vec A, B;
{ return A.x*B.x+A.y*B.y+A.z*B.z; } vec vcomb(a, A, B) double a; vec A, B; { B.x+=a*A.x; B.y+=a*A.y; B.z+=a*A.z; return B; }
vec vunit(A) vec A; { return vcomb(1./sqrt( vdot(A, A)), A, black); } struct sphere *intersect(P, D) vec P, D; { best=0; tmin=1e30;
s= sph+5; while(s-->sph) b=vdot(D, U=vcomb(-1., P, s->cen)), u=b*b-vdot(U, U)+s->rad*s ->rad, u=u>0?sqrt(u):1e31, u=b-u>
1e-7?b-u:b+u, tmin=u>=1e-7&&u<tmin?best=s, u: tmin; return best; } vec trace(level, P, D) vec P, D; { double d, eta, e; vec N, color;
struct sphere *s; if(!level--) return black; if(s=intersect(P, D)); else return amb; color=amb; eta=s->ir; d= -vdot(D, N=vunit(vcomb
(-1., P=vcomb(tmin, D, P), s->cen ))); if(d<0) N=vcomb(-1., N, black), eta=1/eta, d= -d; l=sph+5; while(l-->sph) if(e=l ->kl*vdot(N,
U=vunit(vcomb(-1., P, l->cen))))>0&&intersect(P, U)==l) color=vcomb(e , l->color, color); U=s->color; color.x*=U.x; color.y*=
U.y; color.z*=U.z; e=1-eta* eta*(1-d*d); return vcomb(s->kt, e>0?trace(level, P, vcomb(eta, D, vcomb(eta*d-sqrt( e), N, black)))):
black, vcomb(s->ks, trace(level, P, vcomb(2*d, N, D)), vcomb(s->kd, color, vcomb(s->kl, U, black)))); }
```

```
main() { printf("%d %d\n", 32, 32); while(yx<32*32) U.x=yx%32-32/2, U.z=32/2-yx++/32, U.y=32/2/tan(25/114.5915590261),
U=vcomb(255., trace(3, black, vunit(U)), black), printf("%0.f %0.f %0.f\n", U); } /*minray!*/
```



Object oriented decomposition identifies the objects representing the problem. These objects are defined in an abstract way by specifying what operations can be executed on these elements.

Térpartícionáló módszerek

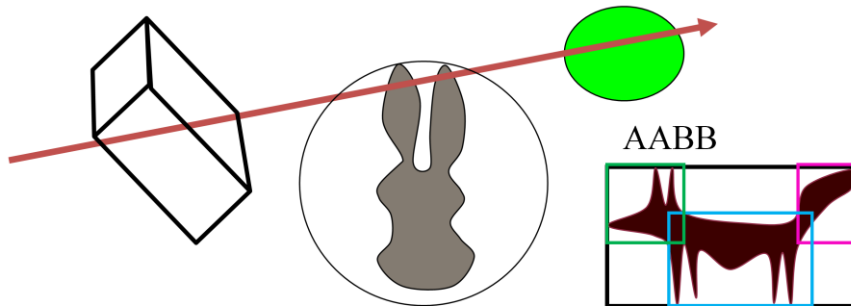


Adatstruktúra:

- Ha ismert a sugár, akkor a potenciális metszett objektumok számát csökkenti
- Ha a potenciálisak közül találunk egyet, akkor a többi nem lehet közelebb

Better complexity – at least in the average case - can be achieved with space partitioning schemes. Space partitioning schemes store information about the possibly intersectable objects for rays shot from arbitrary points and at directions. We have to pay some price for building them, but when they are available, we can ask them which objects are "in the direction of the ray" so intersection is possible, or in which order they follow each other with respect to the distance from the ray origin.

Befoglaló térfogat (Bounding Volume)

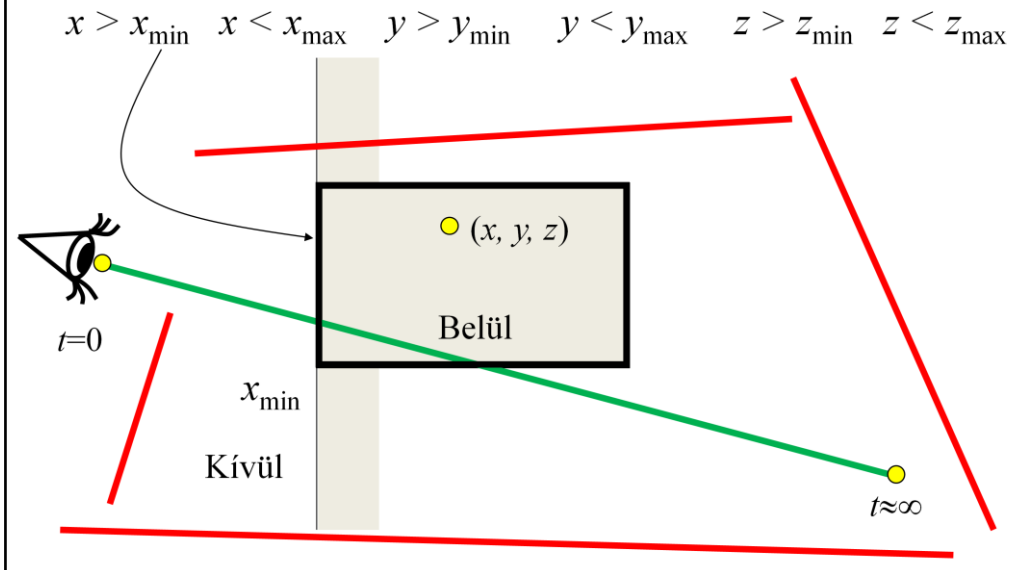


```
double IntersectBV( ray, object )    // < 0 ha nincs  
    IF ( Intersect( ray, bounding volume of object ) < 0 ) RETURN -1;  
    RETURN Intersect( ray, object );  
END
```

Complex objects may be enclosed in simple bounding volumes, such as spheres or AABBs (Axis Aligned Bounding Boxes). Then, before intersecting the complex object, we test the bounding volume. If the bounding volume is not intersected, neither the bounded object can be intersected.

This method speed up ray tracing by a linear factor, but does not change the inherent linear complexity with respect to the number of objects.

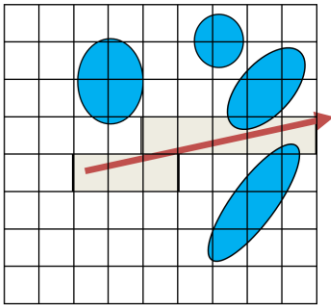
AABB metszés vágással



Reguláris térháló

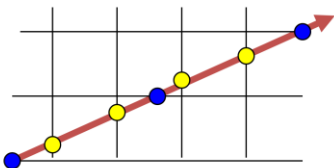
Előfeldolgozás:

Minden cellára a metszett objektumok
komplexitás: $O(n \cdot c) = O(n^2)$



Sugárkövetés:

```
FOR each cell of the line // line drawing
  Metszés a cellában lévőekkel
  IF van metszés RETURN
ENDFOR
```



átlagos eset komplexitás: $O(1)$

Regular grid encloses first the scene into an AABB, then this AABB is decomposed to cells of the same size obtaining a grid structure. During preprocessing, we check every object and every cell whether they overlap. To implement overlapping test, we usually check whether the cell AABB overlaps with the AABB of the object. In every cell, the ids of overlapping objects are stored, e.g. in a linked list or array.

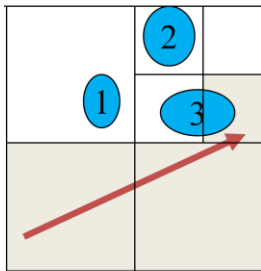
During ray tracing, when the ray gets available we identify the cell that contains this ray. First, objects overlapping with this cell are tested for intersection. If no intersection is found, we traverse only those cells that are intersected by the ray.

Note that regular grid helps excluding most of the objects that are surely not hit by the ray. If an object overlaps with no cell that is pierced by the ray, then this object is never tested for intersection. On the other hand, regular grid also provides an order of potentially intersectable objects in which they should be tested. When we found an intersection in a cell, cell traversal can be stopped. Even if there are intersections in farther cells, those intersections cannot be closer to the ray origin than the already identified intersection.

To implement cell traversal, we should identify cells that are intersected by

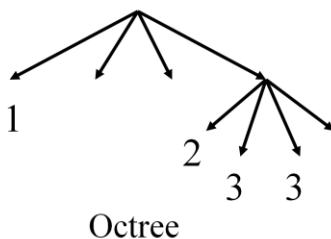
the ray. To find an efficient algorithm, we can exploit the incremental concept. Let us realize that the cell structure is defined by three sets of planes, a horizontal set, a vertical set and a depth set. The distance of intersections with the planes of a given set is constant. So we maintain three ray parameters, one for each plane set. From the three ray parameters, the minimum identifies the next ray cell intersection. Having stepped into the cell, the corresponding ray parameters is incremented by its constant increment.

Nyolcas (oktális) fa



Faépítés(cella):

```
IF a cellában kevés objektum van  
    cellában regisztráld az objektumokat  
ELSE  
    cellafelezés: c1, c2, ..., c8  
    Faépítés(c1); ... Faépítés(c8);  
ENDIF
```



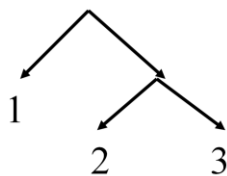
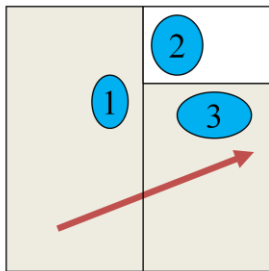
Sugárkövetés:

```
FOR összes sugár által metszett cellára  
    Metszés a cellában lévőekkel  
    IF van metszés RETURN  
ENDFOR
```

A drawback of regular grid is that we should make many cell steps even where the space is empty and there are no objects.

To speed up empty space traversal, we should use larger cells in these empty regions. Octree is based on this idea. It starts with a cell that contains the whole scene, and we check whether this cell has many overlapping objects. If it has, the cell is subdivided by three planes into 8 similar size cells, and we repeat this operation recursively. If the cell is empty or contains just a few objects, further subdivision is prevented.

Binary Space Partitioning fa (kd-fa)



kd-tree

Faépítés(cella):

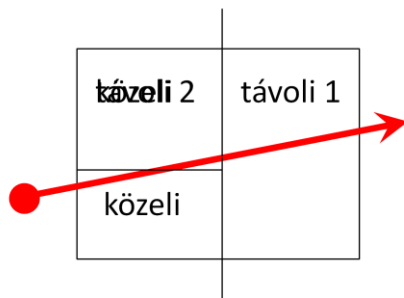
```
IF a cellában kevés objektum van  
    cellában regisztráld az objektumokat  
ELSE  
    sík keresés  
    cella felezés a síkkal: c1, c2  
    Faépítés(c1); Faépítés(c2);  
ENDIF
```

Sugárkövetés:

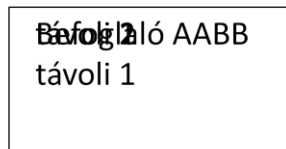
```
FOR each cell intersecting the line  
    Metszés a cellában lévőekkel  
    IF van metszés RETURN  
ENDFOR
```

Kd-tree (aka binary space partitioning tree) is even more adaptive than the octree. In a subdivision step it decomposes a cell into two cells, but selects the position of the subdivision carefully to guarantee that the probabilities of ray intersection in the two children will be similar.

kd-fa bejárása



```
while (stack nem üres) {  
  Pop(AABB)  
  while (nem levél) {  
    Ha van AABB-nek távoli  
    Push(távoli)  
    AABB = közeli  
  }  
  Regisztrált objektumok metszése  
  Ha van metszéspont return  
}
```



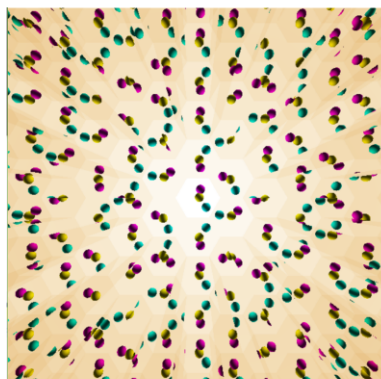
2. Házi feladat: Kaleidoszkóp

Készítsen virtuális kaleidoszkópot. A tükörrendszer szabályos sokszög, amelynek az oldalszáma 3-ról indul és az 'a' billentyűvel lehet inkrementálni. A tükör anyaga arany ('g') vagy ezüst ('s'). A kaleidoszkóp végén legalább három, különböző anyagtulajdonságú ellipszoid található, amelyet ambiens+diffúz+Phong-Blinn spekuláris modell szerint veri vissza a fényt. Az ellipszoidokat véletlen erők érik (Brown mozgás) amit követve mozognak.

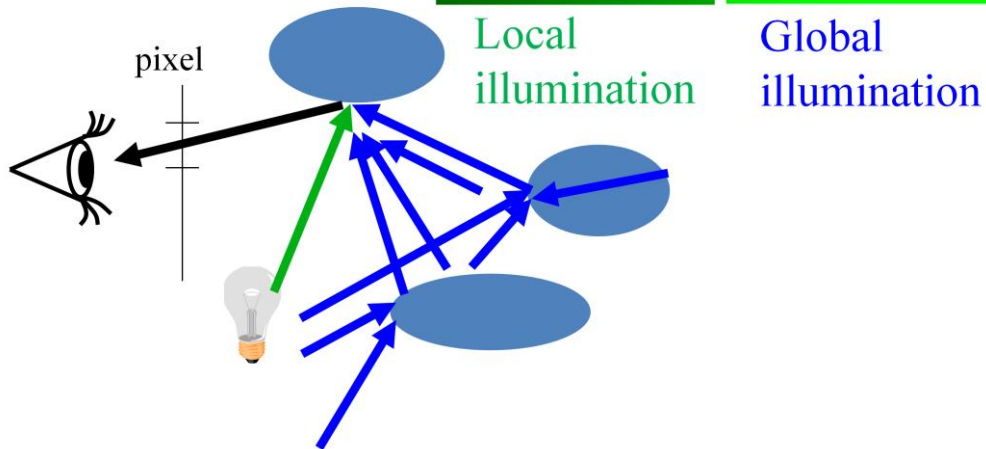
A sima anyagok törésmutatója és kioltási tényezője az R,G,B hullámhosszokon:

Arany (n/k): 0.17/3.1, 0.35/2.7, 1.5/1.9

Ezüst (n/k) 0.14/4.1, 0.16/2.3, 0.13/3.1



Képszintézis



There are different tradeoffs between accuracy of the light transport computation and the speed of the computation.

In the local illumination setting, when the radiance of a surface is calculated, we consider only the direct contribution of the light sources and ignore all indirect illumination.

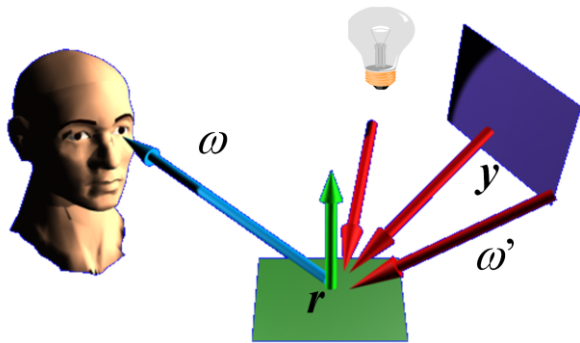
In recursive ray tracing, indirect illumination is computed only for smooth surfaces, in the ideal reflection and refraction directions.

In the global illumination model, indirect illumination is taken into account for rough surfaces as well. In engineering applications we need global illumination solutions since only these provide predictable results. However, in games and real time systems, local illumination or recursive ray tracing will also be acceptable.

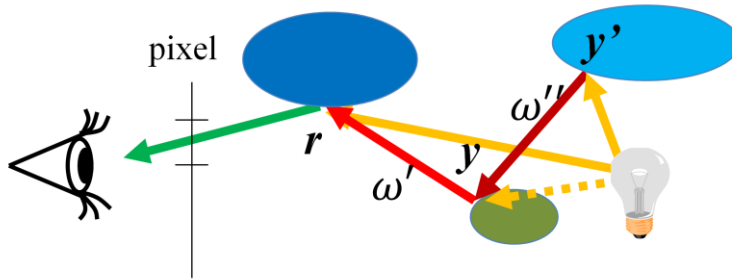
Rendering equation

OutRad = DirectLight + \sum InRad * Reflection

$$L(\mathbf{r}, \omega) = D(\mathbf{r}, \omega) + \int_{\Omega} L(\mathbf{y}, \omega') R(\omega, \omega') d\omega'$$



A megoldás integrálok sorozata

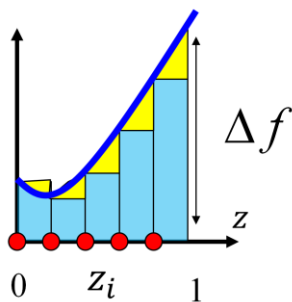


$$L(\mathbf{y}', \omega'') = D(\mathbf{y}', \omega'') + \int_{\Omega''} L(\mathbf{y}'', \omega''') R(\omega'', \omega''') d\omega'''$$

$$L(\mathbf{y}, \omega') = D(\mathbf{y}, \omega') + \int_{\Omega'} L(\mathbf{y}', \omega'') R(\omega', \omega'') d\omega''$$

$$L(\mathbf{r}, \omega) = D(\mathbf{r}, \omega) + \int_{\Omega} L(\mathbf{y}, \omega') R(\omega, \omega') d\omega'$$

Numerikus integrálás



$$\int_0^1 f(z) dz \approx \frac{1}{M} \sum_{i=1}^M f(z_i)$$

M minta

Átlagos
magasság

Alap

Háromszögek
száma

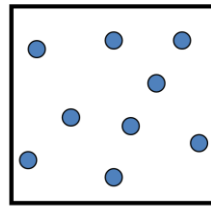
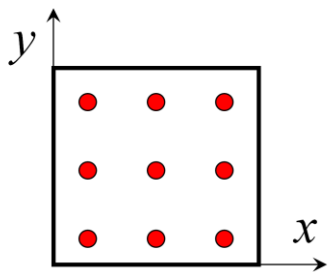
$$\text{Error} = \frac{\Delta f}{2M} \cdot \frac{1}{M} \cdot M = \frac{\Delta f}{2M} = O(M^{-1})$$

Let us consider the problem of dense samples used to estimate an integral. Suppose, for the sake of simplicity, that the integral is one-dimensional and the domain is the unit interval. The simplest integration rule place samples regularly in the domain, evaluates the integrand at these samples, and approximates the area below the function by the total area these bricks. This results in the following sum that approximates the integral.

The error of the integration is the total area of the triangle-like objects between the function curve and the bricks, which equals to the product of the average height of the triangles, the base, and the number of triangles.

We can conclude that the error is proportional to the total change, called variation of the function, and inversely proportional to the number of samples.

Magasabb dimenziók: Megátkozva



Véletlen minták

$n = \sqrt{M}$ minta egy dimenzióban

$$\text{Hiba 2-dim} = O(n^{-1}) = O(M^{-0.5})$$

$$\text{Hiba } D\text{-dim} = O(n^{-1}) = O(M^{-1/D})$$

Monte Carlo integrálás: Integrál = várható érték

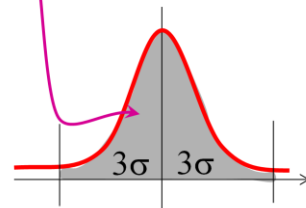
$$\int f(z)dz = \int \frac{f(z)}{p(z)} p(z)dz = \mathbf{E} \left[\frac{f(z)}{p(z)} \right] \approx \frac{1}{M} \sum_{i=1}^M \frac{f(z_i)}{p(z_i)}$$

A becslő egy valószínűségi változó:

$$\text{Variancia} = \sigma^2 = \mathbf{D}^2 \left[\frac{f(z)}{p(z)} \right] \frac{1}{M}$$

3 × szóráson belül
99.7% konfidenciával

$$\text{Error} < 3\sigma = \frac{3}{\sqrt{M}} \mathbf{D} \left[\frac{f(z)}{p(z)} \right]$$



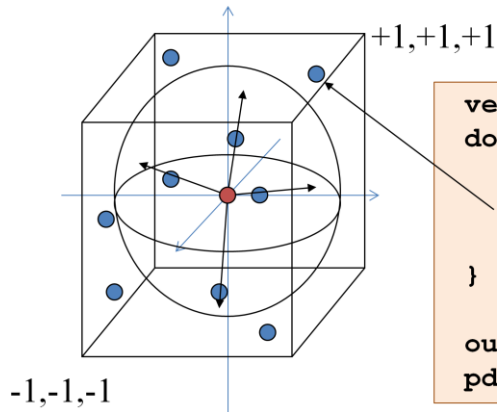
Monte-Carlo integration can also be understood in the following way. In order to evaluate an integral, let us divide and multiply the integrand by a probability density p . Obviously, it does not make any difference.

Looking at this formula, we can realize that this is the expected value of random variable f/p . According to the theorem of large numbers, expected values can be well approximated by averages. Thus taking M samples obtained with probability density p , this average will be a good estimate for the original integral.

The integration error will be proportional to the variation of f/p and inversely proportional to the square root of the number of samples.

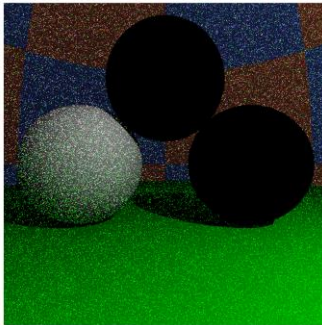
Írányok generálása egyenletes valószínűséggel

```
float random() { return (float)rand()/RAND_MAX; }
```

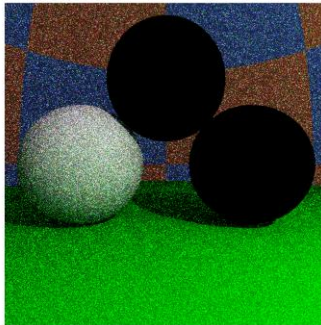


```
vec3 p;  
do {  
    p.x = 2*random()-1;  
    p.y = 2*random()-1;  
    p.z = 2*random()-1;  
} while(dot(p, p) > 1);  
  
outDir = normalize(p);  
pdf = 1.0/4.0/M_PI;
```

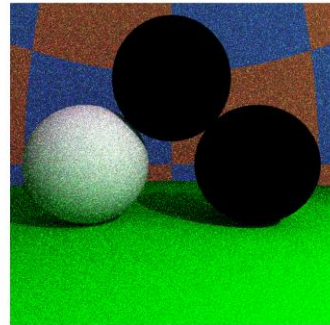
Uniform



1 sample/pixel



10 samples/pixel



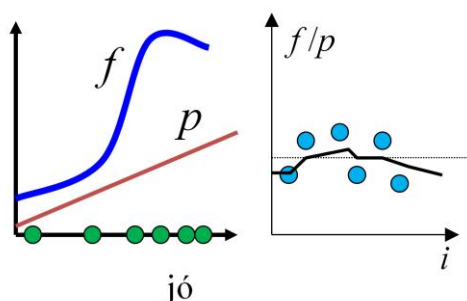
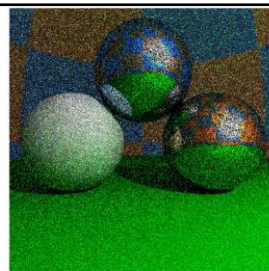
100 samples/pixel

Fontosság szerinti mintavétel

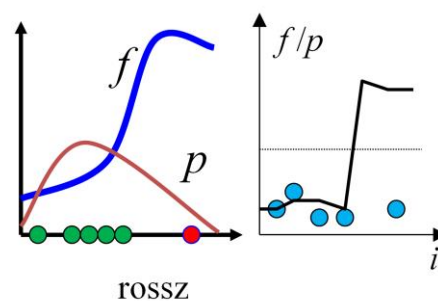
$$\text{Becsllő: } \frac{1}{M} \sum_{i=1}^M \frac{f(z_i)}{p(z_i)}$$

$f(z)/p(z)$ legyen lapos

Ahol f nagy, p is legyen nagy



hasonló f/p hozzájárulások



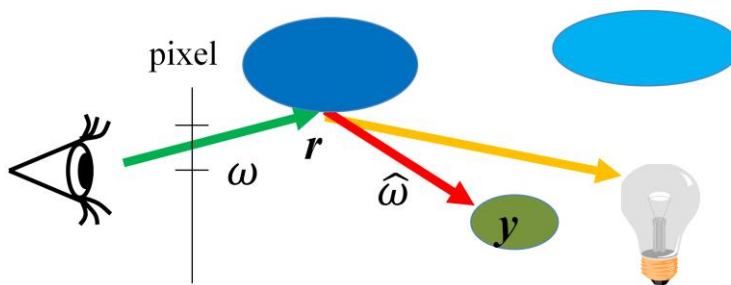
Ritka, nagy f/p hozzájárulások

In order to reduce the error, the variation of f/p should be small, that is where the integrand is large, the probability density should also be large. It means that the sampling with this probability density will place more samples where the integrand is large. This concept is called importance sampling.

This figure compares a good and a bad sampling densities. In the first case the f/p terms in the approximating average will be similar, thus the average remains nearly constant as we add new samples.

In the second case, the important region is sampled rarely, thus the approximating average contains many small values when a very large f/p value appears. The corresponding pixel color is dark and suddenly it becomes very bright and remains bright for a long time. This situation should be avoided.

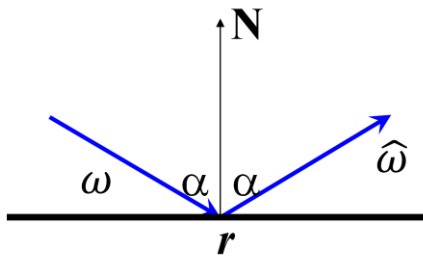
Véletlen bolyongás



$$\int_{\Omega} L(y, \omega') R(\omega, \omega') d\omega' \approx \frac{L(y, \hat{\omega}) R(\omega, \hat{\omega})}{p(\hat{\omega})}$$

- $p(\hat{\omega})$ –nak az integrandusra kell hasonlítani (fontosság)
- A $L(y, \omega')$ nem ismerjük, mert éppen számoljuk
- $p(\hat{\omega})$ legyen arányos $R(\omega, \hat{\omega})$

Tükör: Irány diszkrét eloszlású (Diract-delta)

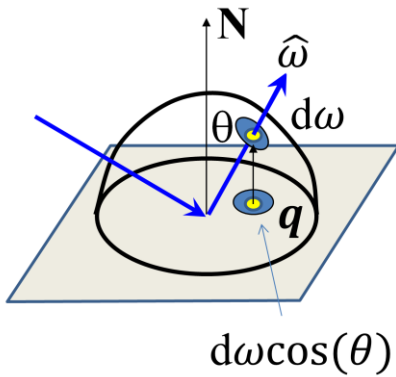


$$\cos \alpha = -(\omega \cdot \mathbf{N})$$

$$\hat{\omega} = \omega + 2\mathbf{N} \cos \alpha$$

```
float SampleMirror(vec3 N, vec3 inDir, vec3& outDir) {  
    outDir = inDir - N * dot(N, inDir) * 2.0f;  
    return 1; // pdf  
}
```

Diffúz: Irány cos eloszlással



$$p(\hat{\omega}) d\omega = \text{Prob}\{\hat{\omega} \text{ in } d\omega\}$$

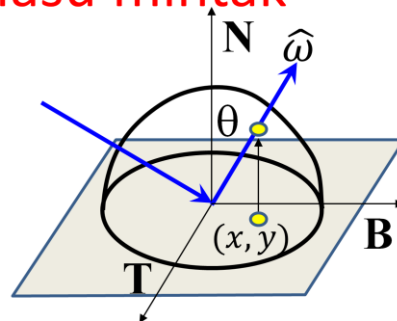
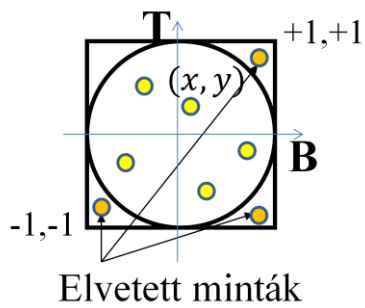
$$= \text{Prob}\{\mathbf{q} \text{ in } d\omega \cos(\theta)\}$$

$$= \frac{d\omega \cos(\theta)}{\text{kör területe}} \quad \text{Ha } \mathbf{q} \text{ egyenletes eloszlású}$$

$$= \frac{d\omega \cos(\theta)}{\pi}$$

$$p(\hat{\omega}) = \frac{\cos(\theta)}{\pi}$$

Cos eloszlású minták

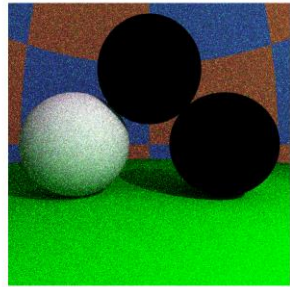
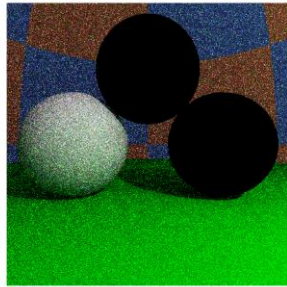
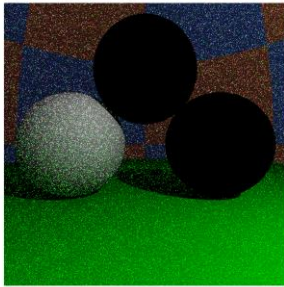


```
float SampleDiffuse(vec3 N, vec3 inDir, vec3& outDir) {
    vec3 T = normalize(cross(N, vec3(1,0,0)));
    vec3 B = cross(N, T);
    float x, y, z;
    do { x = 2*random()-1; y = 2*random()-1; }
        while(x*x + y*y > 1); // reject if not in circle

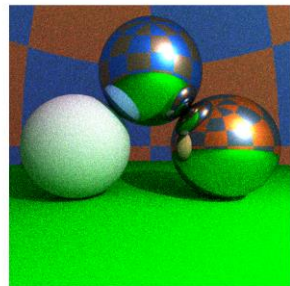
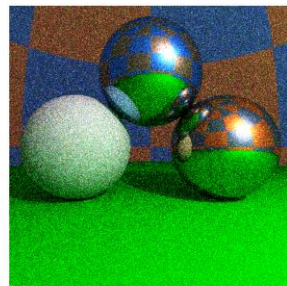
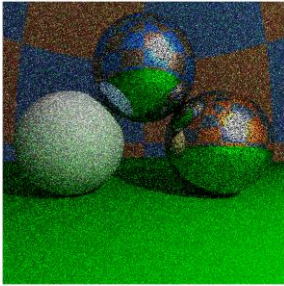
    z = sqrtf(1 - x*x - y*y); // project to hemisphere
    outDir = T * x + B * y + N * z;
    return z/M_PI; // pdf
}
```

Fontosság szerinti mintavétel

egyenletes



cos + diszkrét



1 sample/pixel

10 samples/pixel

100 samples/pixel

Megállás és visszaverődés típus: Orosz rulett

1. Monte Carlo integrál:

$$\int_{\Omega} L(\mathbf{y}, \omega') R(\omega, \omega') d\omega' = \mathbf{E} \left[\frac{L(\mathbf{y}, \hat{\omega}) k_d \cos^+(\theta)}{p_d(\hat{\omega})} \right] + \mathbf{E} \left[\frac{L(\mathbf{y}, \hat{\omega}) F \delta(\omega_m, \hat{\omega})}{p_m(\hat{\omega})} \right]$$

$R_d + R_m$
 $\mathbf{E}[L_d]$
 $\mathbf{E}[L_m]$

2. Számold $\mathbf{E}[L_d]$ -t s_d valószínűséggel, $\mathbf{E}[L_m]$ -t s_m valószínűséggel, egyébként 0
3. Kompenzálj s -sel osztással

Várható érték OK:

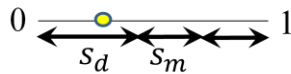
$$s_d \mathbf{E}[L_d/s_d] + s_m \mathbf{E}[L_m/s_m] + (1 - s_d - s_m)0 = \mathbf{E}[L_d] + \mathbf{E}[L_m]$$

A visszaverődés típusának kiválasztása

$$\frac{L_d}{s_d} = \frac{L(\mathbf{y}, \hat{\omega}) k_d \cos^+(\theta)}{p_d(\hat{\omega}) s_d} = L(\mathbf{y}, \hat{\omega}) \frac{k_d \cos^+(\theta)}{\frac{\cos(\theta)}{\pi} s_d} = L(\mathbf{y}, \hat{\omega}) \frac{k_d \pi}{s_d}$$

$$\frac{L_m}{s_m} = \frac{L(\mathbf{y}, \hat{\omega}) F \delta(\omega_m, \hat{\omega})}{p_m(\hat{\omega}) s_m} = L(\mathbf{y}, \hat{\omega}) \frac{F}{s_m}$$

- Tükörirány valószínűsége: $s_m = F$ luminance
- Diffúz irány valószínűsége: $s_d = k_d \pi$ luminance
- Megállás valószínűsége: $1 - s_m - s_d$



Path Tracer

```
vec3 trace(Ray ray, int depth = 0) {
    Hit hit = firstIntersect(ray);
    if (hit.t < 0 || depth >= maxdepth) return vec3(0,0,0);
    [r, N, kd, n, κ] ← hit;
    vec3 outRad = DirectLight(hit);
    rnd = random(); // Russian roulette
    sd = Luminance(kd π); sm = Luminance(Fresnel(ray.dir, N));
    if (rnd < sd) { // diffuse
        pdf = SampleDiffuse(N, ray.dir, outDir);
        inRad = trace(Ray(r + Nε, outDir), depth+1);
        outRad += inRad * kd * dot(N, outDir) / pdf / sd;
    } else if (rnd < sd + sm) { // mirror
        pdf = SampleMirror(N, ray.dir, outDir);
        inRad = trace(Ray(r + Nε, outDir), depth+1);
        outRad += inRad * Fresnel(ray.dir, N) / pdf / sm;
    }
    return outRad;
}
```