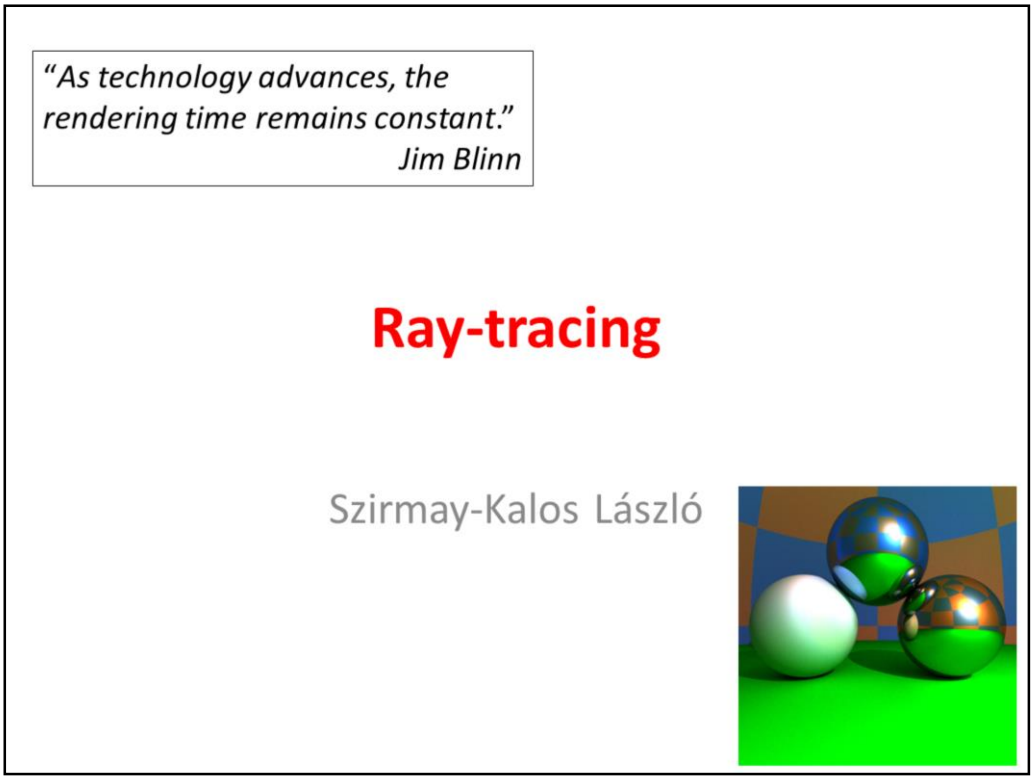
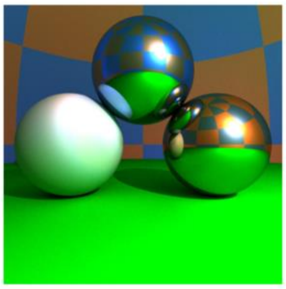


“As technology advances, the rendering time remains constant.”
Jim Blinn

# Ray-tracing

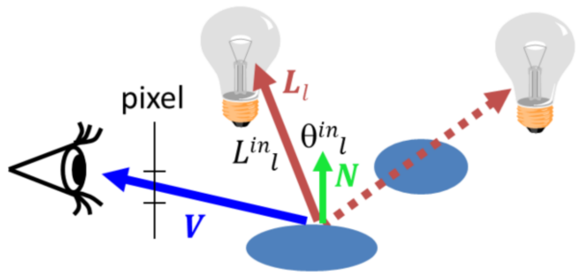
Szirmay-Kalos László



Ray-tracing is a fundamental algorithm to render 3D scenes in a realistic way. It is the basis of many photorealistic rendering methods and are used extensively in movie production rendering.

## Local illumination: rough surfaces, abstract light sources

For each abstract light source



$$L(\boldsymbol{V}) \approx \sum_l L^{in}{}_l \ * f_r(\boldsymbol{L}_l, \boldsymbol{N}, \boldsymbol{V}) \cdot \cos^{+}\theta^{in}{}_l$$

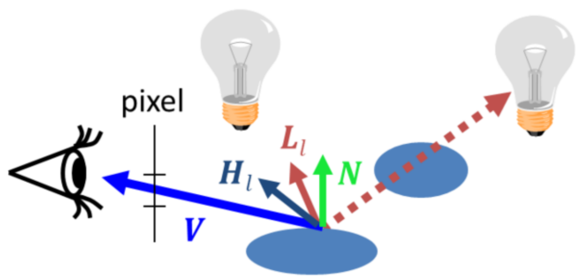
Illumination of an abstract light source
*Directional = constant; Point = decreases with the square of the distance*
*If occluded, zero*

In rendering, first the surface visible through a pixel should be determined by computing the first intersection of the surfaces and the half line starting at the eye position and going in the direction of the pixel. In local illumination rendering, having identified the surface visible in a pixel, we have to compute the reflected radiance due to the direct illumination of the light sources. We prefer abstract light sources for the sake of simplicity since 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. The visibility of light sources from the shaded point can be determined by sending a ray from here in the direction of the light source and checking whether or not this ray intersects any surface before reaching the light source.

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 the intensity decreases with the square of the distance.

## Local illumination: rough surfaces, abstract light sources

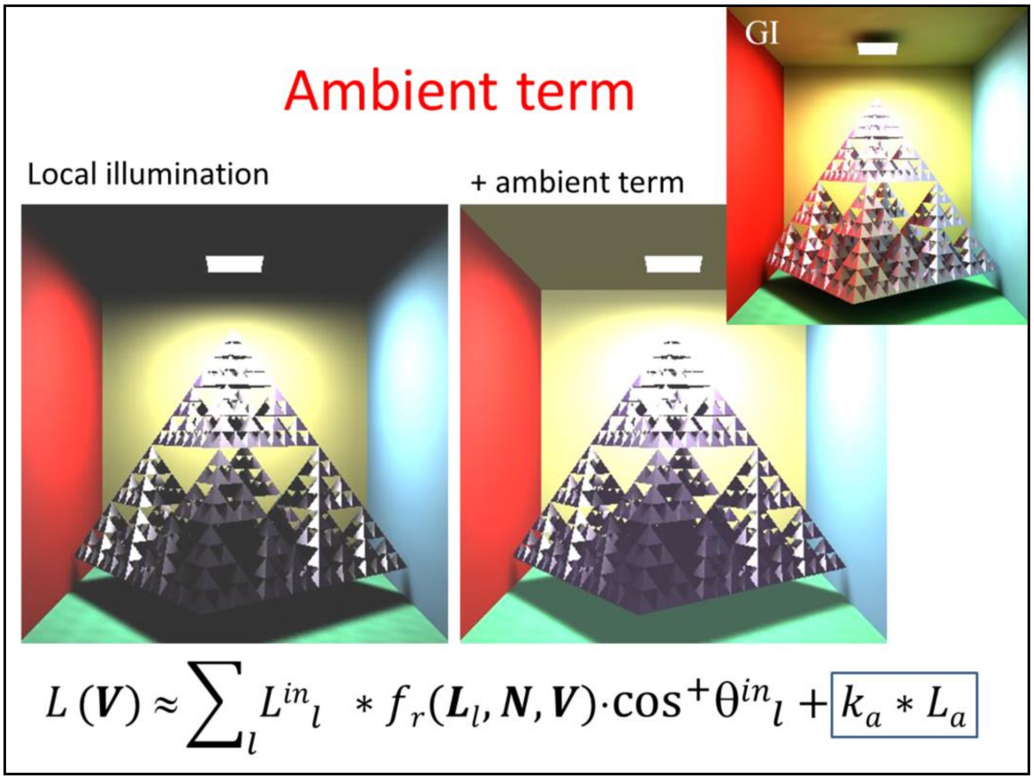
For each abstract light source



$$L\,(\boldsymbol{V}) \approx \sum_l L^{in}{}_l \; * \{k_d \cdot (\boldsymbol{L}_l \bullet \boldsymbol{N})^+ + k_s \cdot ((\boldsymbol{H}_l \bullet \boldsymbol{N})^+)^{shine}\}$$

Illumination of an abstract light source
*Directional = constant; Point = decreases with the square of the distance*
*If occluded, zero*

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. The plus sign in the superscript indicates that negative dot products are replaced by zero since this corresponds to the situation when the considered object occludes the given point from the light source.



## Ambient term

Local illumination

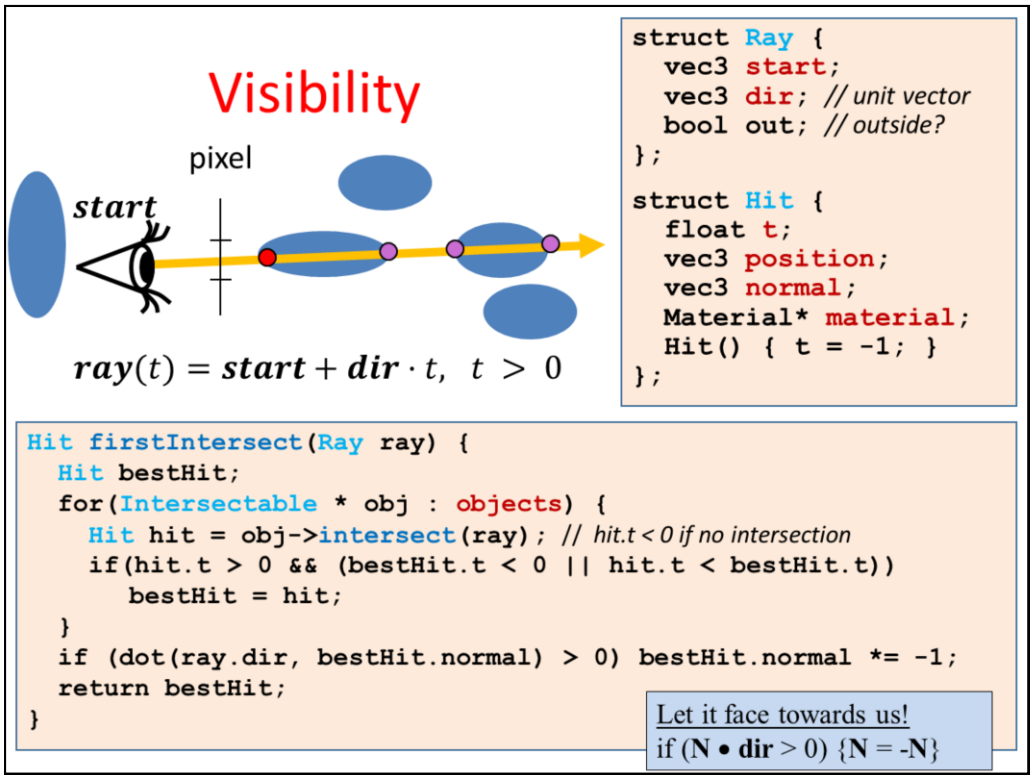
+ ambient term

GI

$$L\,(\boldsymbol{V}) \approx \sum_l L^{in}{}_l \;*\; f_r(\boldsymbol{L}_l, \boldsymbol{N}, \boldsymbol{V}) \cdot \cos^+ \theta^{in}{}_l + \boxed{k_a * L_a}$$

In the local illumination model, all surfaces that are not directly visible from the light sources are completely back. 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).



Visibility

$$\boldsymbol{ray}(t) = \boldsymbol{start} + \boldsymbol{dir} \cdot t, \quad t > 0$$

```
struct Ray {
   vec3 start;
   vec3 dir; // unit vector
   bool out; // outside?
};

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 if no intersection
      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;
}
```

Let it face towards us!
if (**N** • **dir** > 0) {**N** = -**N**}

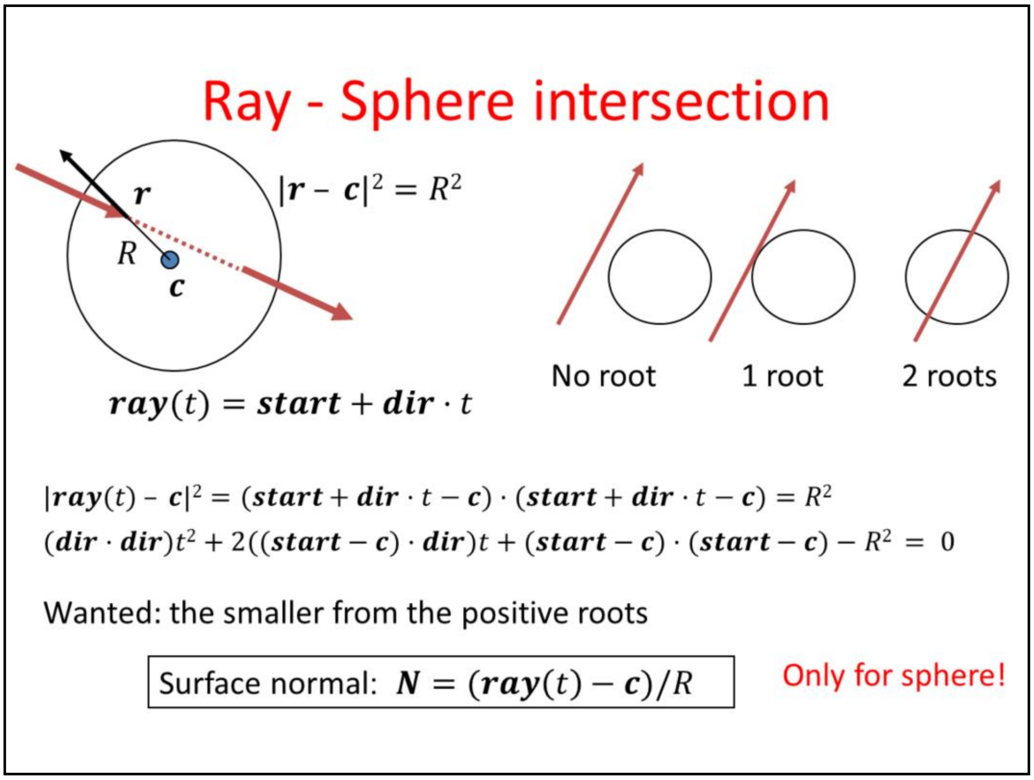
A fundamental operation of ray tracing is the identification of the surface point hit by a ray. A ray is defined by its start and direction. When transparent objects are also considered, it is useful to store whether the ray is inside some object or outside in the air since the index of refraction should be dealt with accordingly.

The ray may be a primary ray originating at the eye and passing through the pixel, or 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**, ray(t)=start + dir· t for some POSITIVE **ray parameter** t, and at the same time, it is also on the visible object, so point ray(t) also satisfies the equation of the surface. By emphasizing the constraint of positive ray parameters, we have a half line or restrict the visible objects to those that are in front of the eye. 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. This implementation assumes that the existence of objects' intersect function that returns a hit structure.

This hit structure contains the ray parameter of the intersection or a negative value when there is no intersection. In addition to the ray parameter t, the hit structure also stores the **position** of the intersection in the virtual world, and the **normal** and the material properties of the surface at the intersection point. So, intersection calculation should also determine the normal vector of the surface at the intersection point. Recall when we established formulas and implemented functions for the reflection or refraction direction computation, we assumed that the surface normal points towards the incoming ray, so the dot product of the ray direction and the surface normal is negative. To make sure that it always happens, it is worth checking this condition and flipping the surface normal if needed.

## Object = Intersectable

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



## Ray - Sphere intersection

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

$$\boldsymbol{ray}(t) = \boldsymbol{start} + \boldsymbol{dir} \cdot t$$

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

$$(\boldsymbol{dir} \cdot \boldsymbol{dir})t^2 + 2((\boldsymbol{start} - \boldsymbol{c}) \cdot \boldsymbol{dir})t + (\boldsymbol{start} - \boldsymbol{c}) \cdot (\boldsymbol{start} - \boldsymbol{c}) - R^2 = 0$$

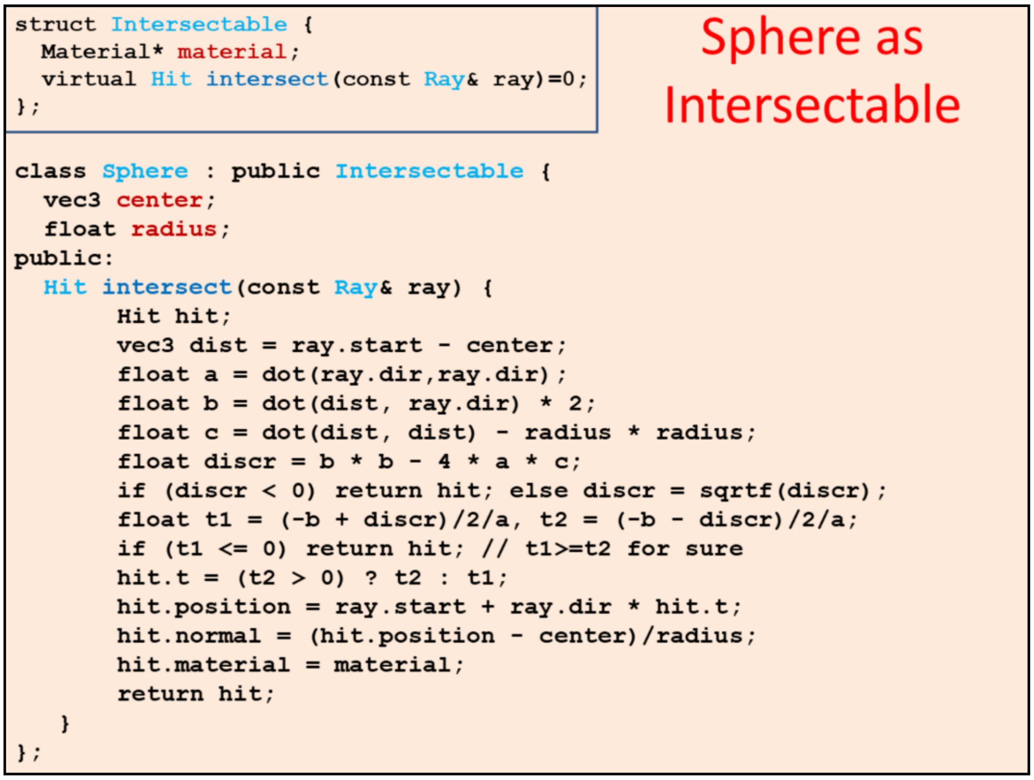
Wanted: the smaller from the positive roots

$$\boxed{\text{Surface normal: } \boldsymbol{N} = (\boldsymbol{ray}(t) - \boldsymbol{c})/R}$$

Only for sphere!

The implementation of function **intersect** depends on the actual type of the surface, since it means the inclusion of the ray equation into the particular 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 correspond 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 leaning it. From the existing roots, we need the smallest positive one.

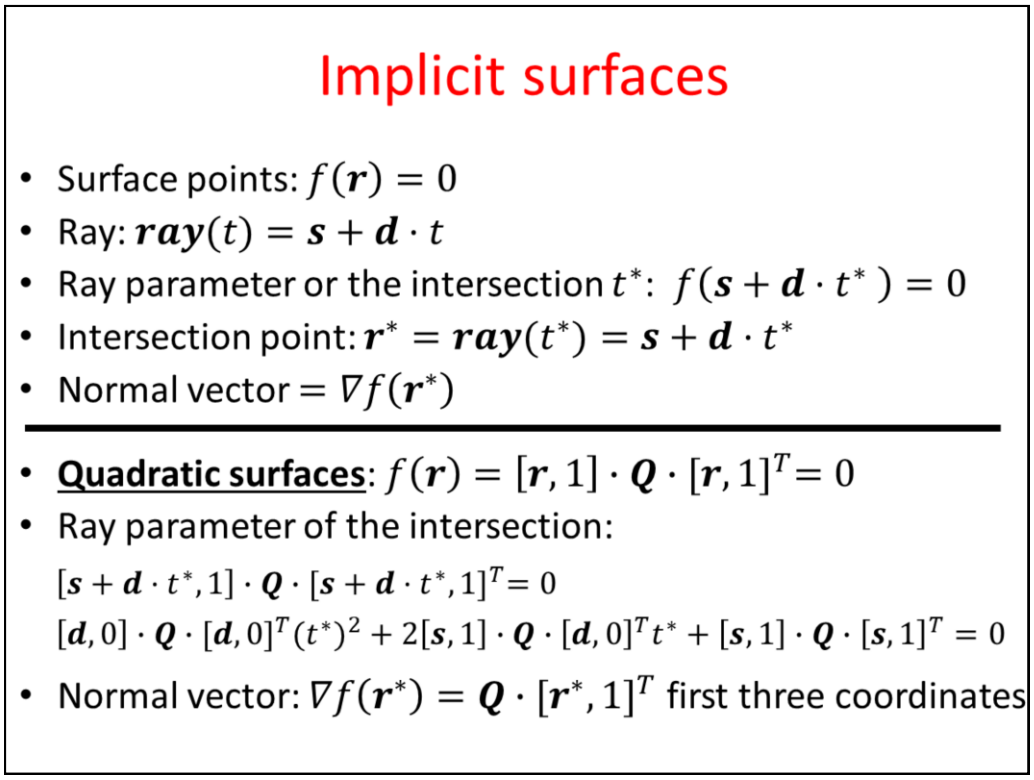
Recall that we also need the surface normal at the intersection point. For a sphere, the normal is parallel to the vector pointing from the center to the surface point. It can be normalized, i.e. turned to a unit vector, by dividing by its length, which equals to the radius of the sphere.



## 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;
   }
};
```

On the implementation level, the abstract definition of an object that can be rendered with ray tracing is provided by the Intersectable struct, which has material properties and an abstract function intersect that is supposed to compute the intersection of the ray and this surface. The intersect function can be given a real body if we know the equation of the surface. Therefore, specific types are inherited from the general abstract base class. Here, we present the Sphere inherited from Intersectable. The sphere has particular parameters like the center and the radius, and the intersection with a ray can be implemented.



## Implicit surfaces

- Surface points: $f(\boldsymbol{r}) = 0$
- Ray: $\boldsymbol{ray}(t) = \boldsymbol{s} + \boldsymbol{d} \cdot t$
- Ray parameter or the intersection $t^*$: $f(\boldsymbol{s} + \boldsymbol{d} \cdot t^*) = 0$
- Intersection point: $\boldsymbol{r}^* = \boldsymbol{ray}(t^*) = \boldsymbol{s} + \boldsymbol{d} \cdot t^*$
- Normal vector $= \nabla f(\boldsymbol{r}^*)$

---

- **Quadratic surfaces**: $f(\boldsymbol{r}) = [\boldsymbol{r}, 1] \cdot \boldsymbol{Q} \cdot [\boldsymbol{r}, 1]^T = 0$
- Ray parameter of the intersection:

$$[\boldsymbol{s} + \boldsymbol{d} \cdot t^*, 1] \cdot \boldsymbol{Q} \cdot [\boldsymbol{s} + \boldsymbol{d} \cdot t^*, 1]^T = 0$$

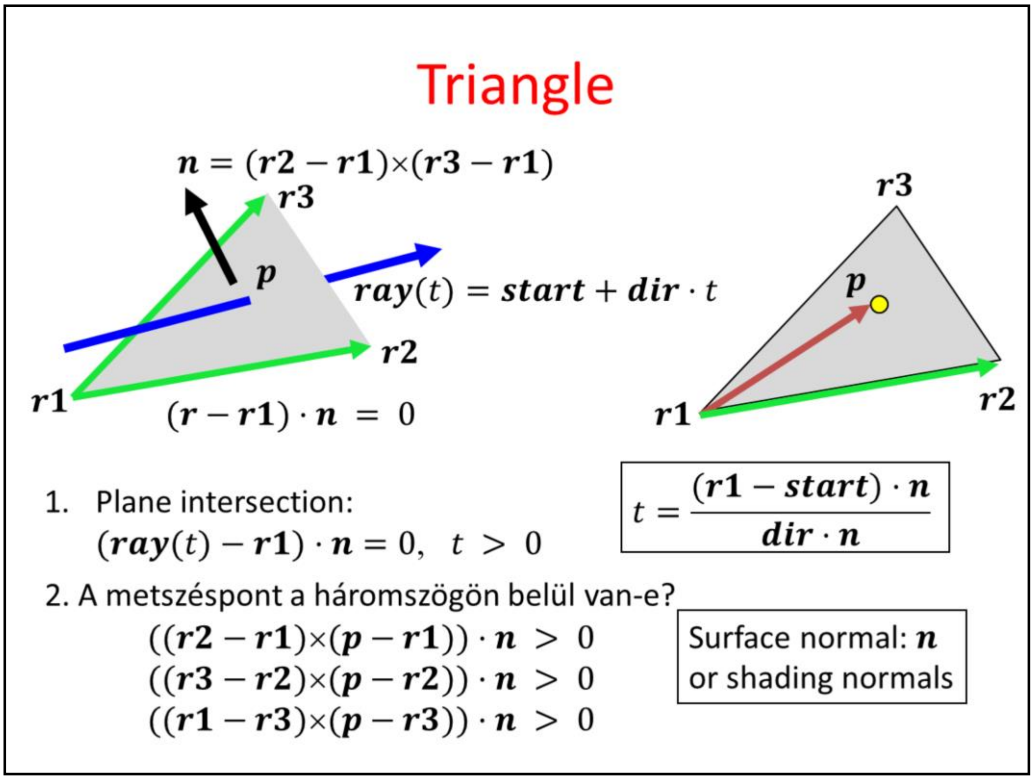
$$[\boldsymbol{d}, 0] \cdot \boldsymbol{Q} \cdot [\boldsymbol{d}, 0]^T (t^*)^2 + 2[\boldsymbol{s}, 1] \cdot \boldsymbol{Q} \cdot [\boldsymbol{d}, 0]^T t^* + [\boldsymbol{s}, 1] \cdot \boldsymbol{Q} \cdot [\boldsymbol{s}, 1]^T = 0$$

- Normal vector: $\nabla f(\boldsymbol{r}^*) = \boldsymbol{Q} \cdot [\boldsymbol{r}^*, 1]^T$ first three coordinates

The equation of the sphere is an example of a more general category of the implicit surfaces that are defined by an implicit equation of the x,y,z Cartesian coordinates or the position vectors 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 n\cdot(r – r0) = 0, which is the equation of the plane, where n=grad f.

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


r3
p
r2
r1
r3
p
r1
r2

## Triangle

$\boldsymbol{n} = (\boldsymbol{r2} - \boldsymbol{r1}) \times (\boldsymbol{r3} - \boldsymbol{r1})$

$\boldsymbol{ray}(t) = \boldsymbol{start} + \boldsymbol{dir} \cdot t$

$(\boldsymbol{r} - \boldsymbol{r1}) \cdot \boldsymbol{n} = 0$

1. Plane intersection:
   $(\boldsymbol{ray}(t) - \boldsymbol{r1}) \cdot \boldsymbol{n} = 0, \quad t > 0$

$$\boxed{t = \frac{(\boldsymbol{r1} - \boldsymbol{start}) \cdot \boldsymbol{n}}{\boldsymbol{dir} \cdot \boldsymbol{n}}}$$

2. A metszéspont a háromszögön belül van-e?

$$((\boldsymbol{r2} - \boldsymbol{r1}) \times (\boldsymbol{p} - \boldsymbol{r1})) \cdot \boldsymbol{n} > 0$$
$$((\boldsymbol{r3} - \boldsymbol{r2}) \times (\boldsymbol{p} - \boldsymbol{r2})) \cdot \boldsymbol{n} > 0$$
$$((\boldsymbol{r1} - \boldsymbol{r3}) \times (\boldsymbol{p} - \boldsymbol{r3})) \cdot \boldsymbol{n} > 0$$

Surface normal: $\boldsymbol{n}$
or 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 solution.

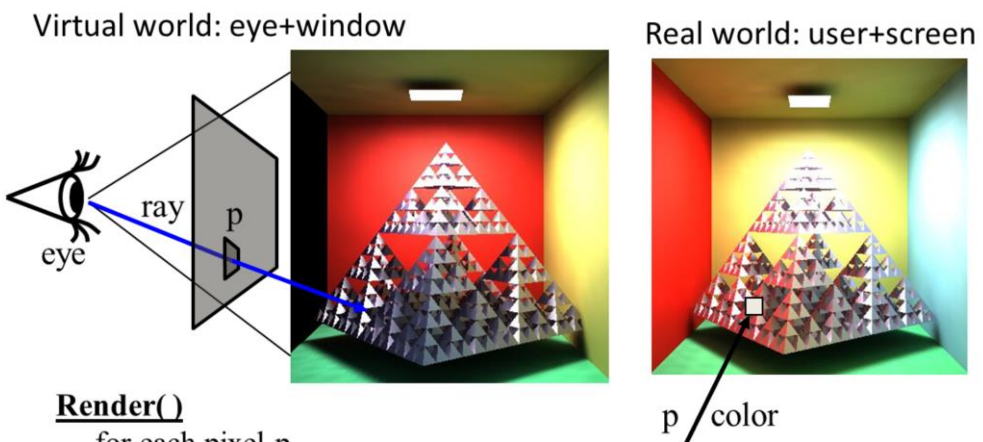
The algorithm consists of two steps, first the intersection with the plane of the triangle is found, then we determine whether or not the ray-plane intersection point is inside the triangle. Suppose that the triangle is given by its vertices r1, r2, r3. The equation of its plane is n\cdot(r-r0)=0 where n is the normal vector and r0 is a point of the plane. Position vector r0 can be any of the three vertices and normal vector n can be computed as the cross product of edge vectors r2-r1 and r3-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 p as the intersection with the plane.

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

Assuming that we look at the plane from above, (**r2** - **r1**) × (**p** - **r1)** will point towards us if p is on the left, and it will point down if p is on the right.

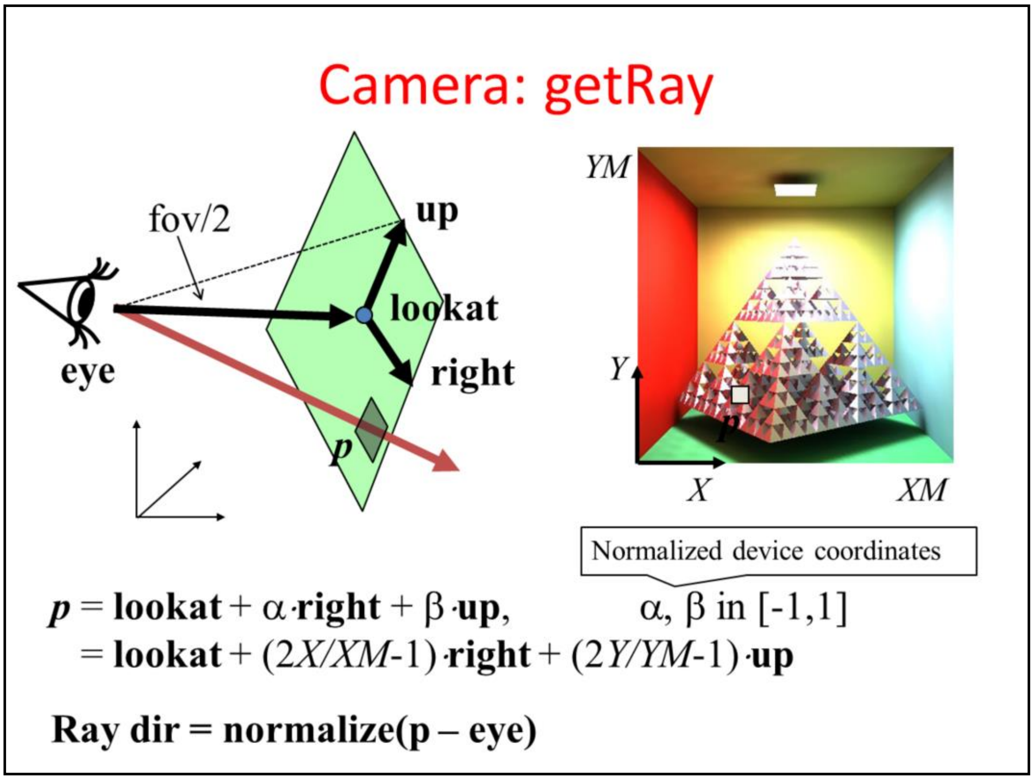
As **n** = (**r2** - **r1**) × (**r3** - **r1**) points towards us, we can check whether (**r2** - **r1**) × (**p** - **r1)** has the same direction by computing their dot product and checking if the result is positive (the dot product of two vectors point into the same direction is positive, the dot 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.

## Ray tracing: Render



```
Render( )
    for each pixel p
        Ray r = getRay( eye ⇨ pixel p )
        color = trace(ray)
        WritePixel(p, color)
    endfor
end
```

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.



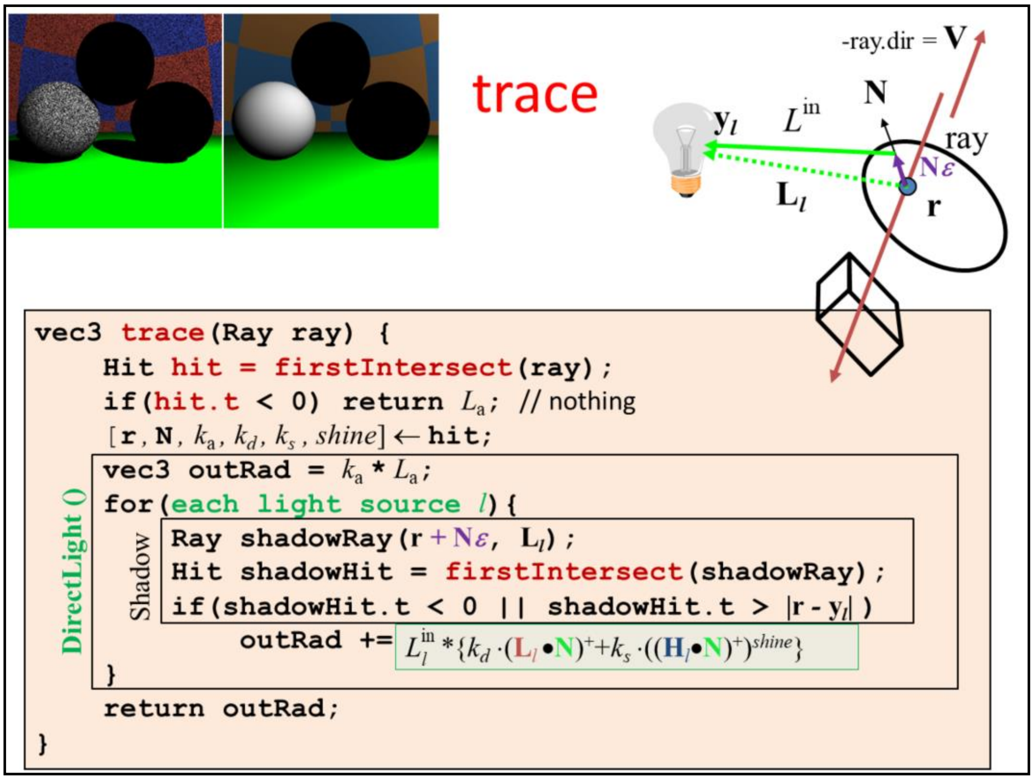
## Camera: getRay

$$\boldsymbol{p} = \mathbf{lookat} + \alpha\cdot\mathbf{right} + \beta\cdot\mathbf{up}, \qquad \alpha, \beta \text{ in } [-1,1]$$
$$= \mathbf{lookat} + (2X/XM-1)\cdot\mathbf{right} + (2Y/YM-1)\cdot\mathbf{up}$$

**Ray dir = normalize(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 position 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. Any point on the window can be obtained as going to the center called lookat and then applying the combination of right and up where the combination parameters alpha and beta are in [-1,1], which are the normalized screen space coordinates. Physical pixels are mapped to normalized screen coordinates using the XM, YM screen resolutions.

If the resolution of the target image is XM x YM, then the center of pixel (X,Y) in world coordinates is $\boldsymbol{p} = \mathbf{lookat} + (2X/XM-1)\cdot\mathbf{right} + (2Y/YM-1)\cdot\mathbf{up}.$



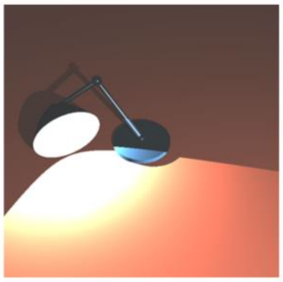
trace

```
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 + $N\varepsilon$, $L_l$);
        Hit shadowHit = firstIntersect(shadowRay);
        if(shadowHit.t < 0 || shadowHit.t > $|r - y_l|$ )
            outRad += $L_l^{in} * \{k_d \cdot (L_l \bullet N)^+ + k_s \cdot ((H_l \bullet N)^+)^{shine}\}$
    }
    return outRad;
}
```

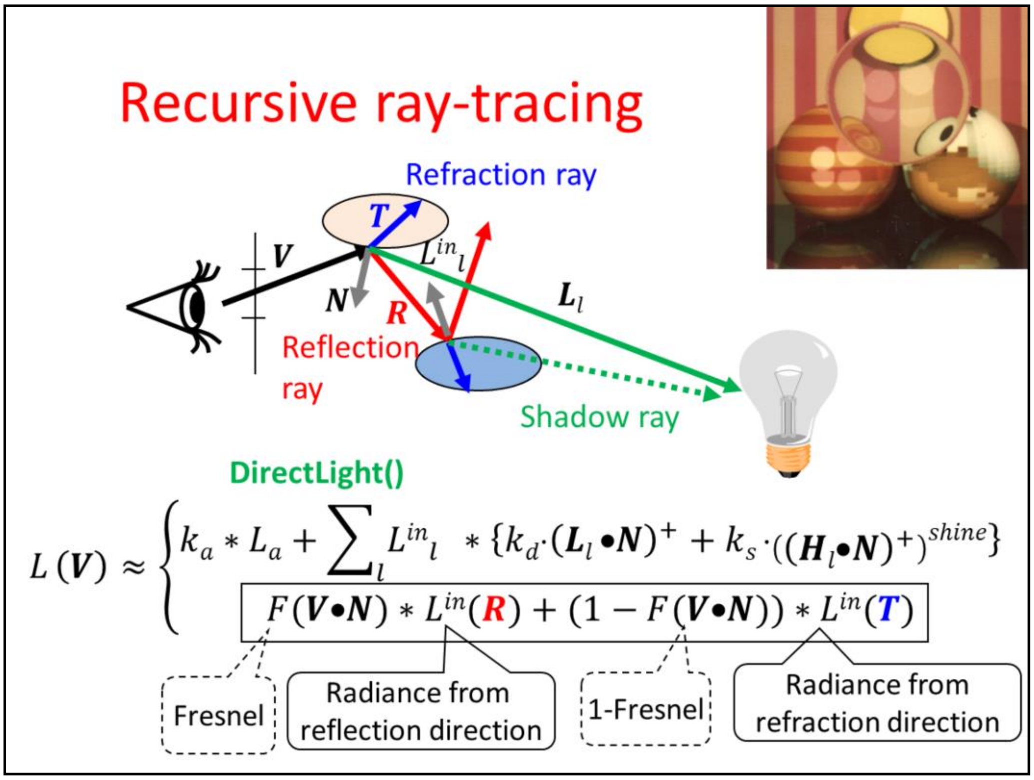
The **trace** function gets the ray that involves its origin and direction vectors. First we compute the intersection that is in front of the eye and is closest to the eye. The already implemented solution is **firstIntersect**. This function indicates with a negative value if there is no intersection. 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. Note that the start of the shadow ray is pushed a little in the direction of the surface normal N in order to avoid situations when numerical inaccuracies result in a intersection with the identified surface with not zero but some small positive ray parameter. This would produce ugly black dots on the surface as shown by the figure.

## Homework 2: Luxo Grandpa



Write a ray tracing program that renders a paraboloid lamp, planar table and arbitrary quadratic or polygonal objects on the table. A point light source is in the focal point of the paraboloid and casts shadows. In addition to the point source, the scene is illuminated by an ambient source. Objects can be diffuse-specular, or reflective or refractive.



## Recursive ray-tracing

DirectLight()

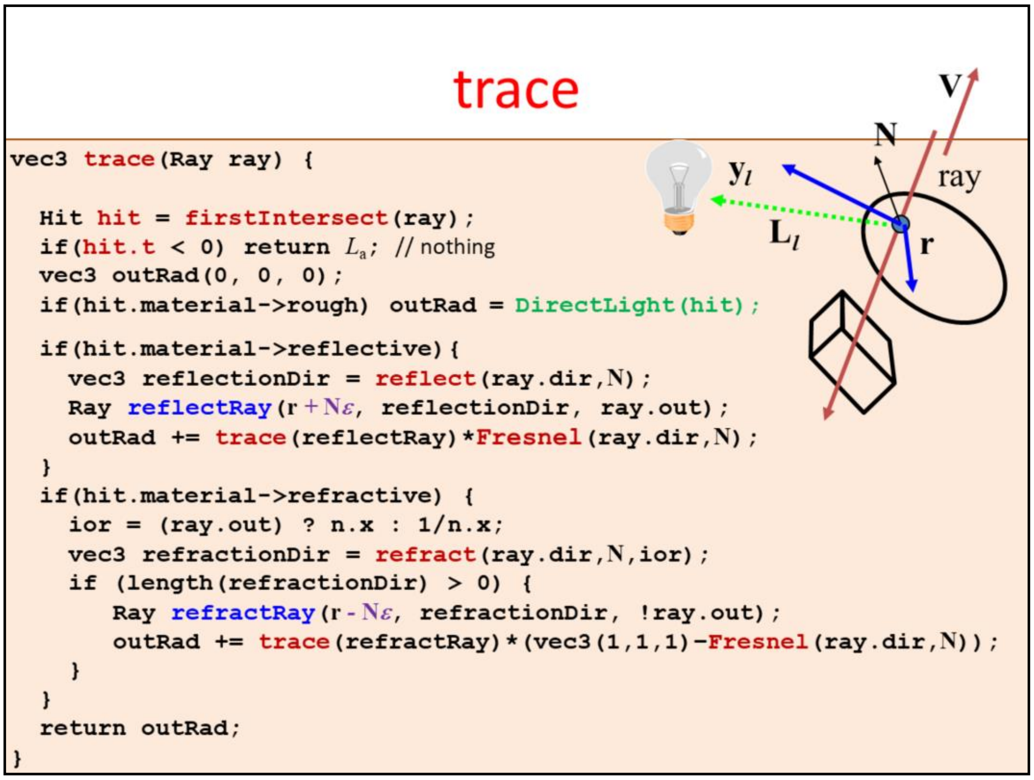
$$L(\boldsymbol{V}) \approx \begin{cases} k_a * L_a + \sum_l L^{in}_l * \{k_d \cdot (\boldsymbol{L}_l \bullet \boldsymbol{N})^+ + k_s \cdot ((\boldsymbol{H}_l \bullet \boldsymbol{N})^+)^{shine}\} \\ \boxed{F(\boldsymbol{V}\bullet\boldsymbol{N}) * L^{in}(\boldsymbol{R}) + (1 - F(\boldsymbol{V}\bullet\boldsymbol{N})) * L^{in}(\boldsymbol{T})} \end{cases}$$

Fresnel | Radiance from reflection direction | 1-Fresnel | Radiance from refraction direction

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 in case of rough surfaces. However, for optically smooth surfaces, the reflection of the radiance from the ideal reflection direction is computed, and for transparent objects, the refraction of the radiance coming from the ideal refraction direction is added. 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 on 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.


V
N
ray
$y_l$
$L_l$
r

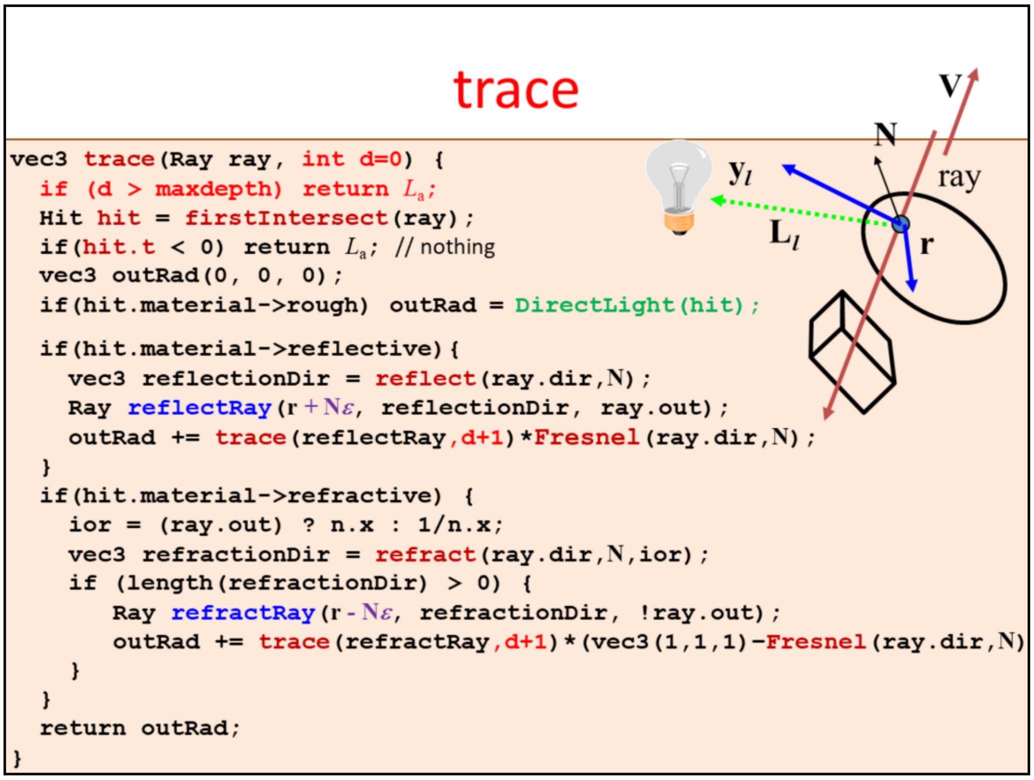
## trace

```
vec3 trace(Ray ray) {

  Hit hit = firstIntersect(ray);
  if(hit.t < 0) return L_a; // 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 a rough surface is seen, trace computes the contribution of the abstract light sources, for which we implemented the **DirectLight** function.

If the surface is smooth and is ideally reflective, then the reflection direction is computed with the already implemented **reflect** function 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.

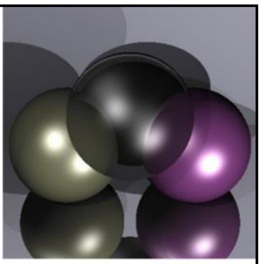


## trace

```
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;
}
```

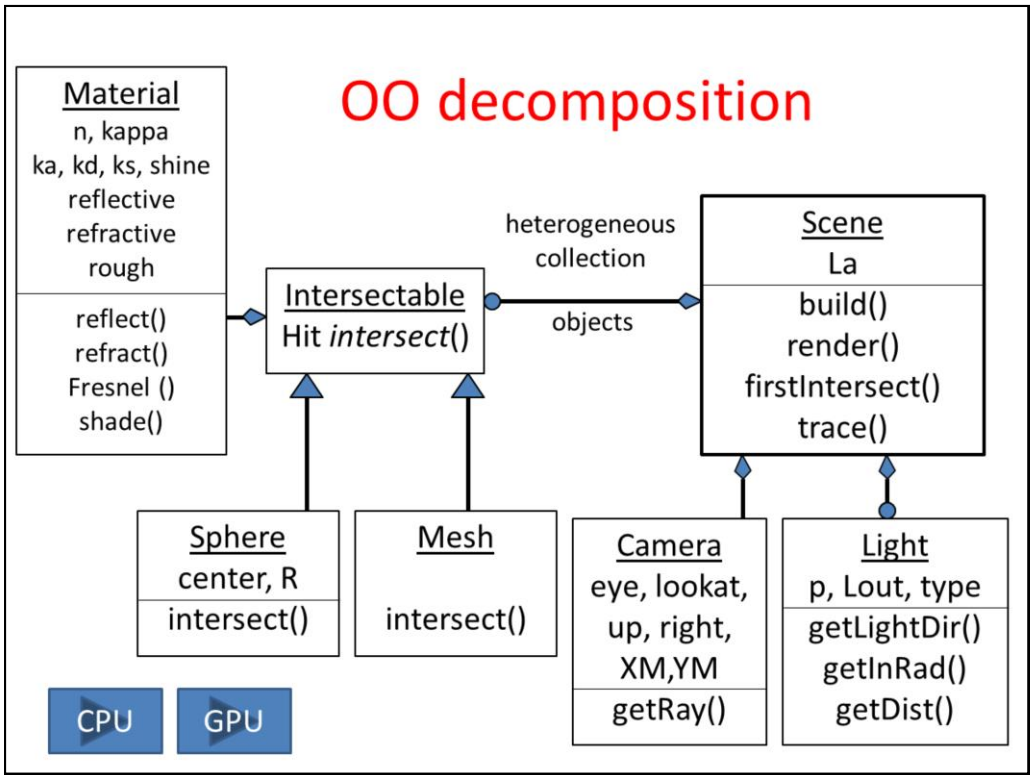
Recursion is a dangerous operation if we cannot make sure that it stops. Assume, for example, that this ellipsoid is made of glass. The ray is refracted into the glass and there can be infinite number of reflections on the internal surface. So our program will surely crash with a stack overflow error. We should limit the recursion depth for any price. This is possible with depth parameter d, which is incremented in each recursive call. If depth is greater than the limit, additional calculations are terminated.

## Paul Heckbert's business card



```
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,.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,*l;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("%.0f %.0f %.0f\n",U);}/*minray!*/
```

Ray tracing is an elegant algorithm, the implementation fits just a business card, but can render rough surfaces, reflective and refractive objects, and shadows.



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.

The virtual world is represented by the Scene class. A Scene can be built and rendered, which in turn requires the ability to trace a ray in the scene, which should find the first intersection with firstIntersect. Ambient light intensity La is a data member of the Scene. The Scene has embedded objects like the Camera and Light sources. The camera serves the rendering algorithm with its getRay member function. The Scene is also a collection of Intersectable objects. Intersectable is an abstract base class, so this is a heterogeneous collection. Any intersectable can be associated with a Material containing all possible material properties. Intersectable has a pure virtual function called intersect.

Specific geometric object types can be included in this virtual world by deriving their type from the general Intersectable and implementing the intersect method.