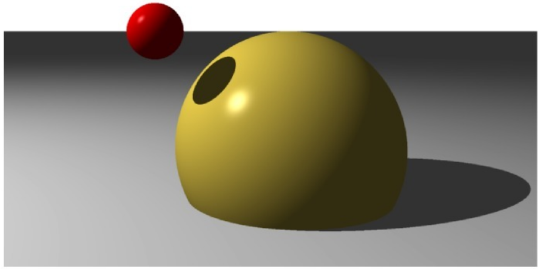
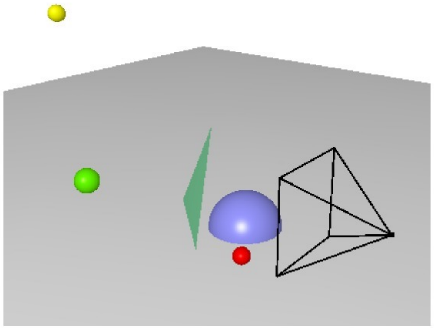
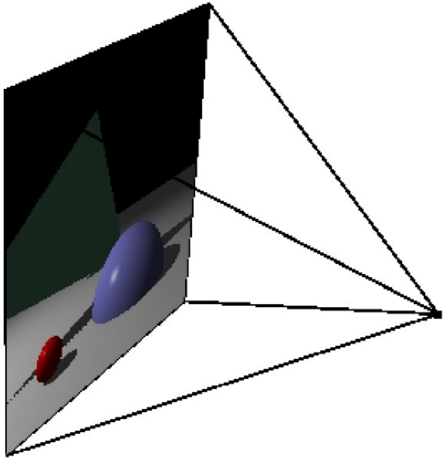


Ray tracing





Context

Rendering: **3D Model** → **Image**

Ex. 3D Sphere:

$$x^2 + y^2 + z^2 = 1 \xrightarrow{?}$$

Model *Image*

Context

Rendering: **3D Model** → **Image**

Ex. 3D Models

$$x^2 + y^2 + z^2 = 1 \xrightarrow{?}$$

Image

Multiples solutions (+/-)

- Projective rendering
- Reye
- Ray tracing

Overview

1/ General algorithm for ray tracing

2/ Simple scene application
=> *See objects*

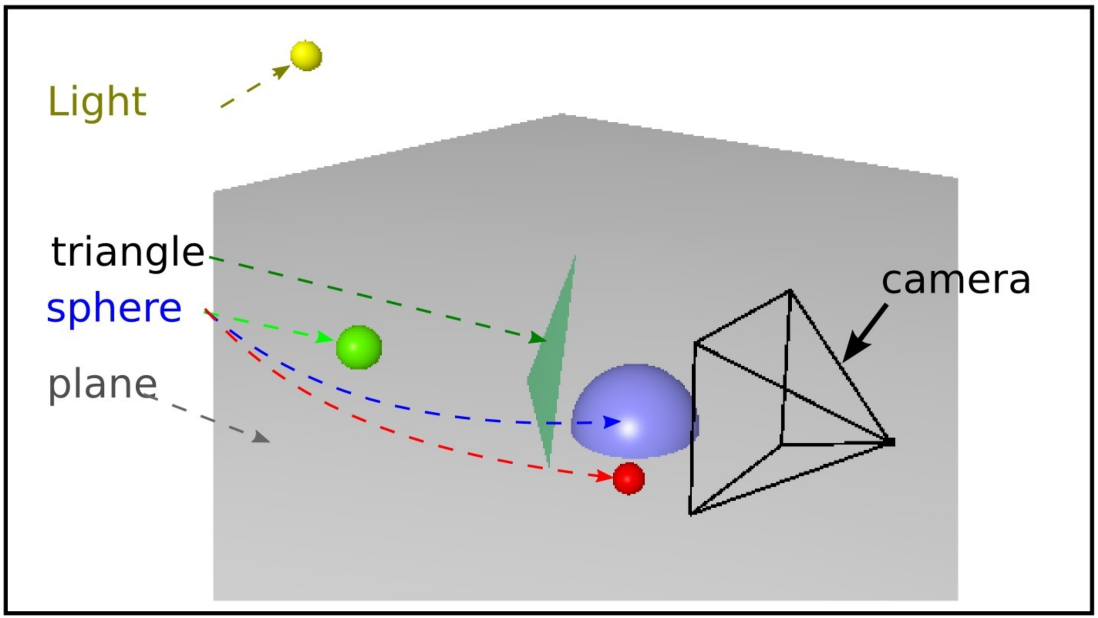
3/ Shading
=> *3D aspect*

4/ Physical model
=> *Toward photorealism*

1/ General algorithm for ray tracing

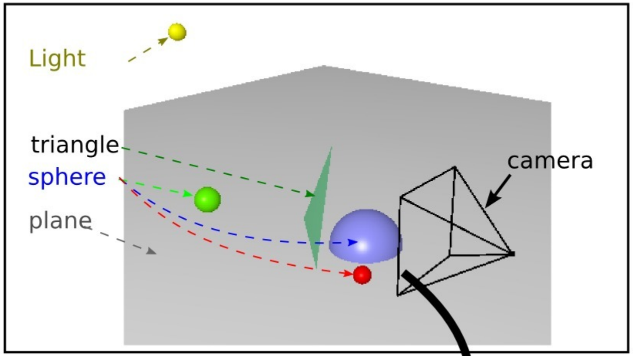
3D Scene

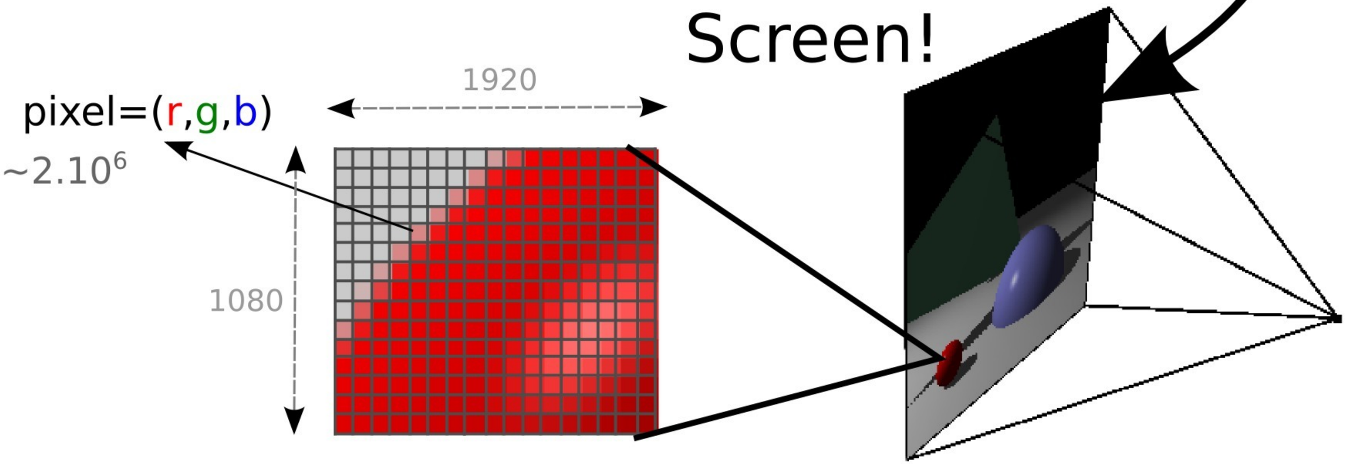
3D Scene=
- 3D Objet(s)
  - Geometry: equation
  - Color/Material
- Camera/screen
- Light source



3D Scene

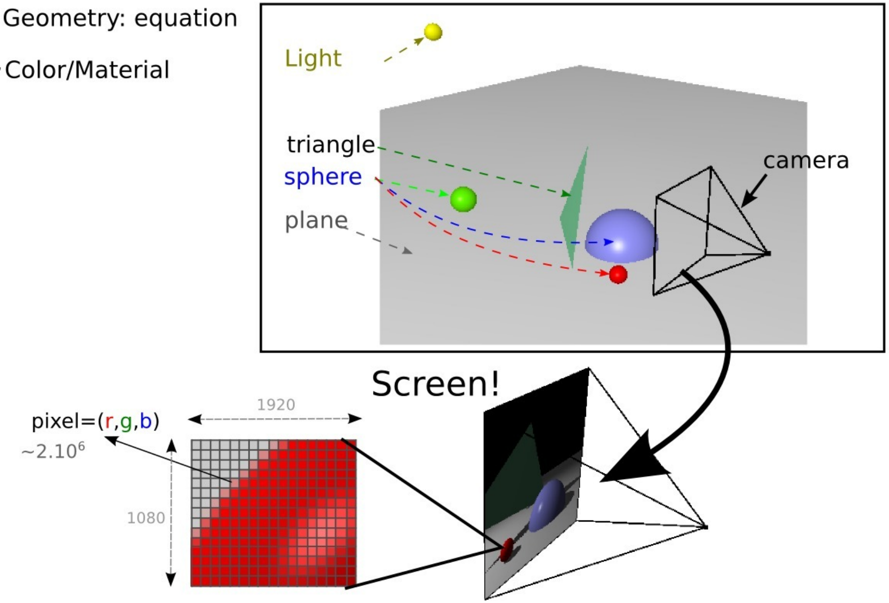
3D Scene=
- 3D Objet(s)
  - Geometry: equation
  - Color/Material
- Camera/screen
- Light source





3D Scene

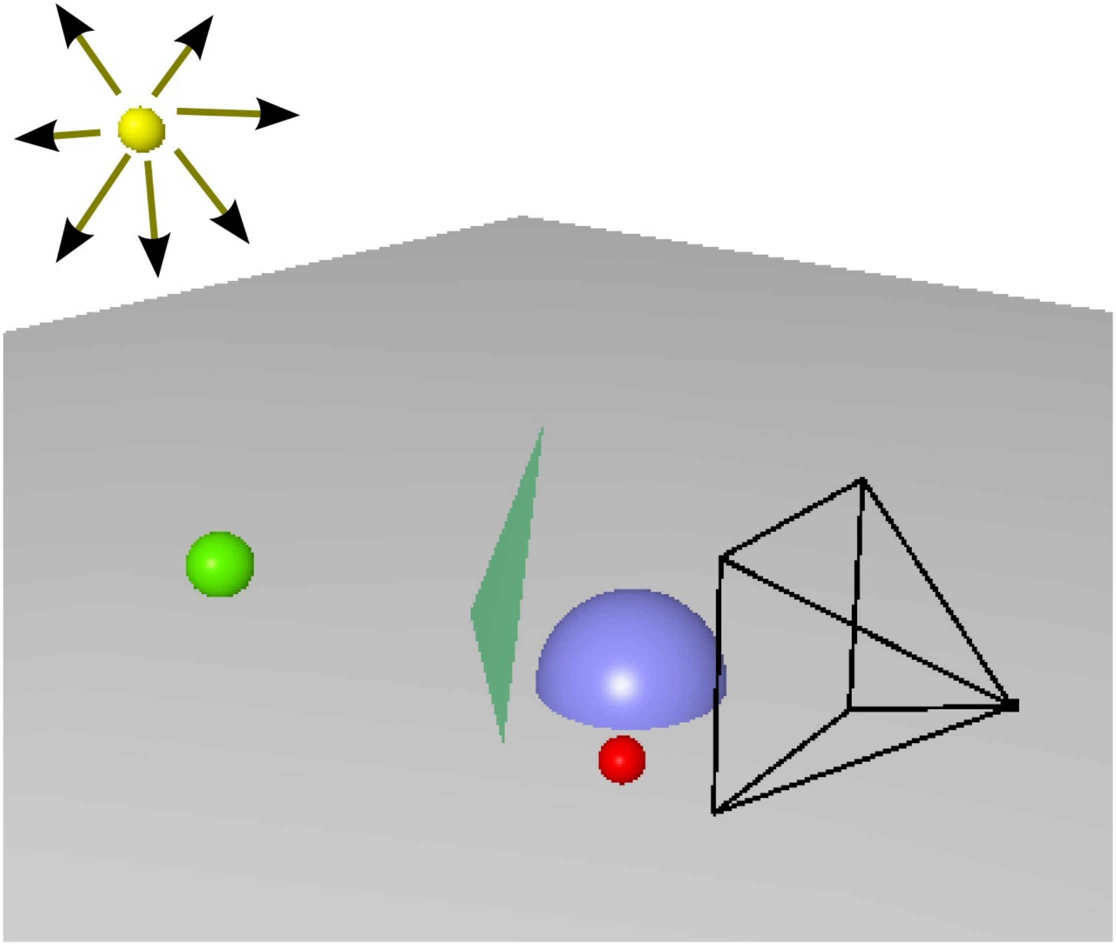
3D Scene=
- 3D Objet(s)
  - Geometry: equation
  - Color/Material
- Camera/screen
- Light source



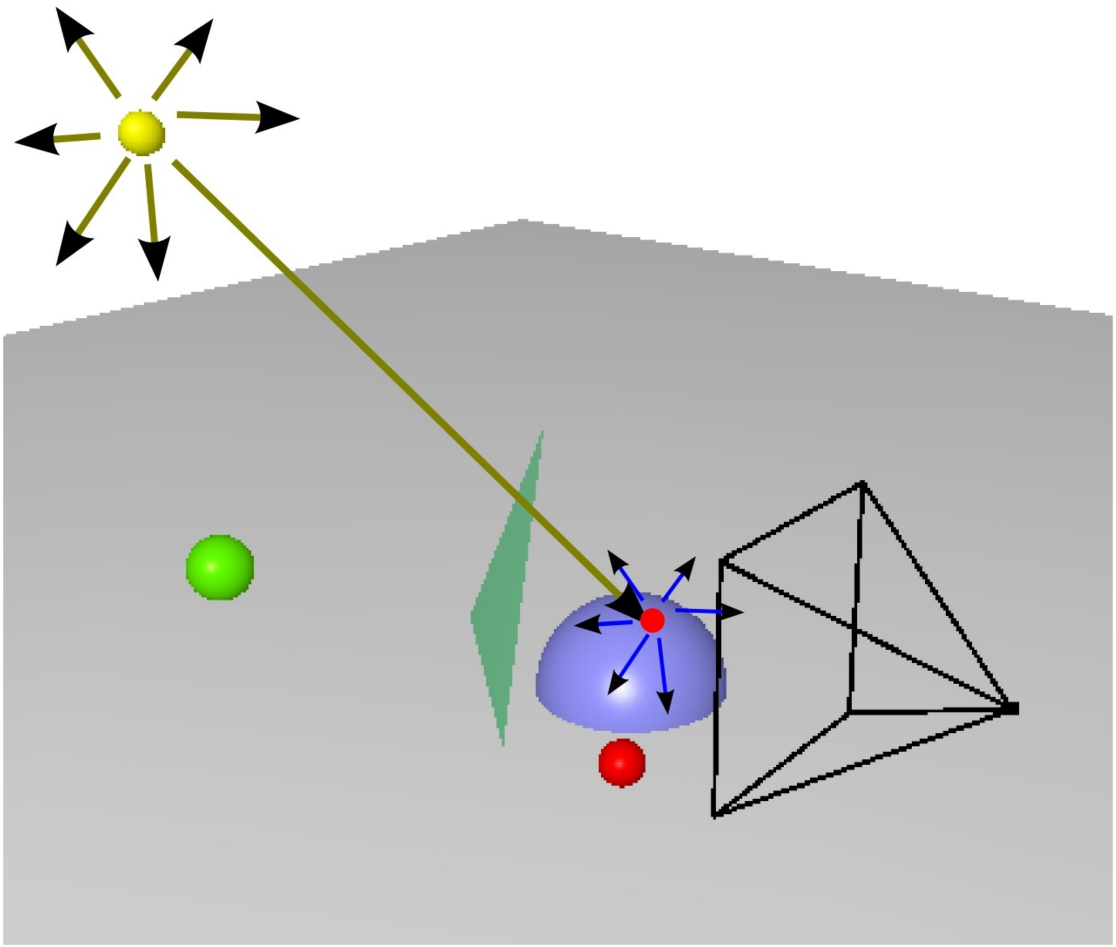
Question:

What color should we give to each pixel?

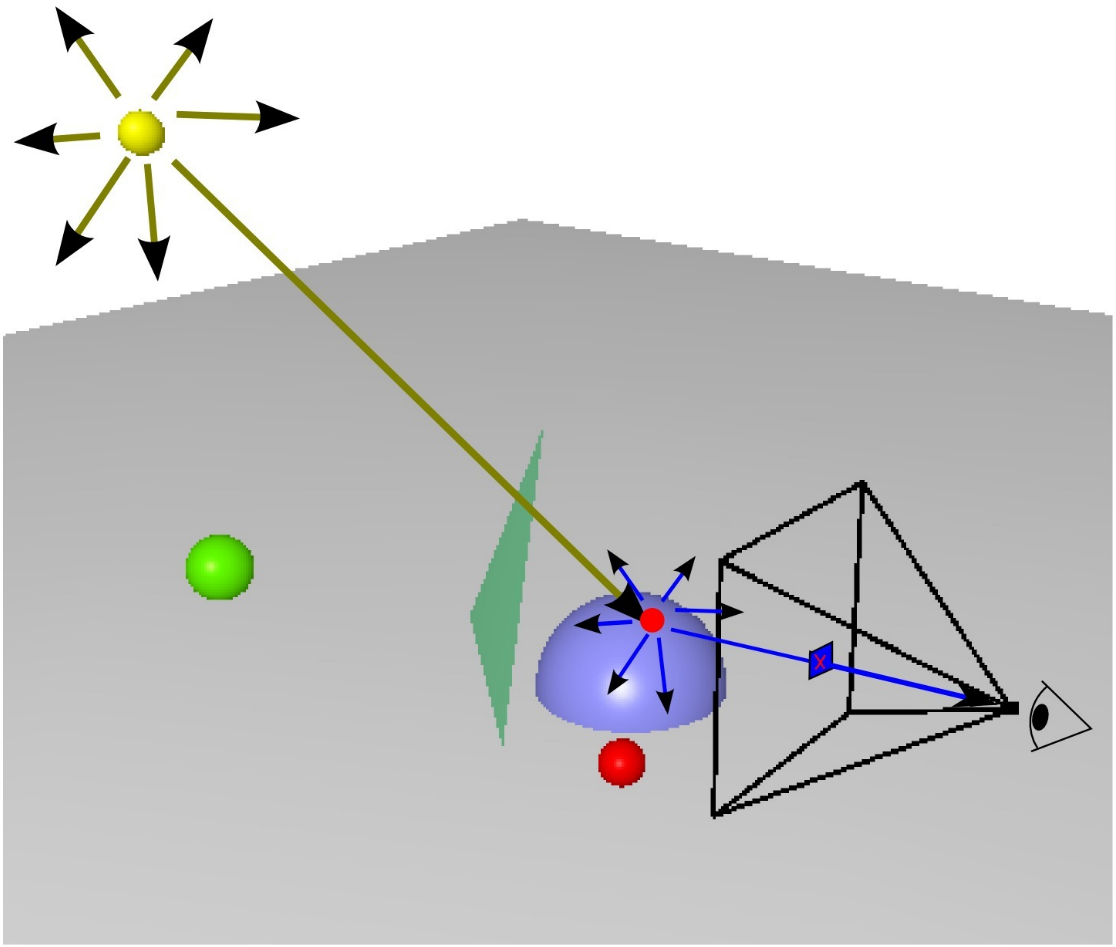
In the real world



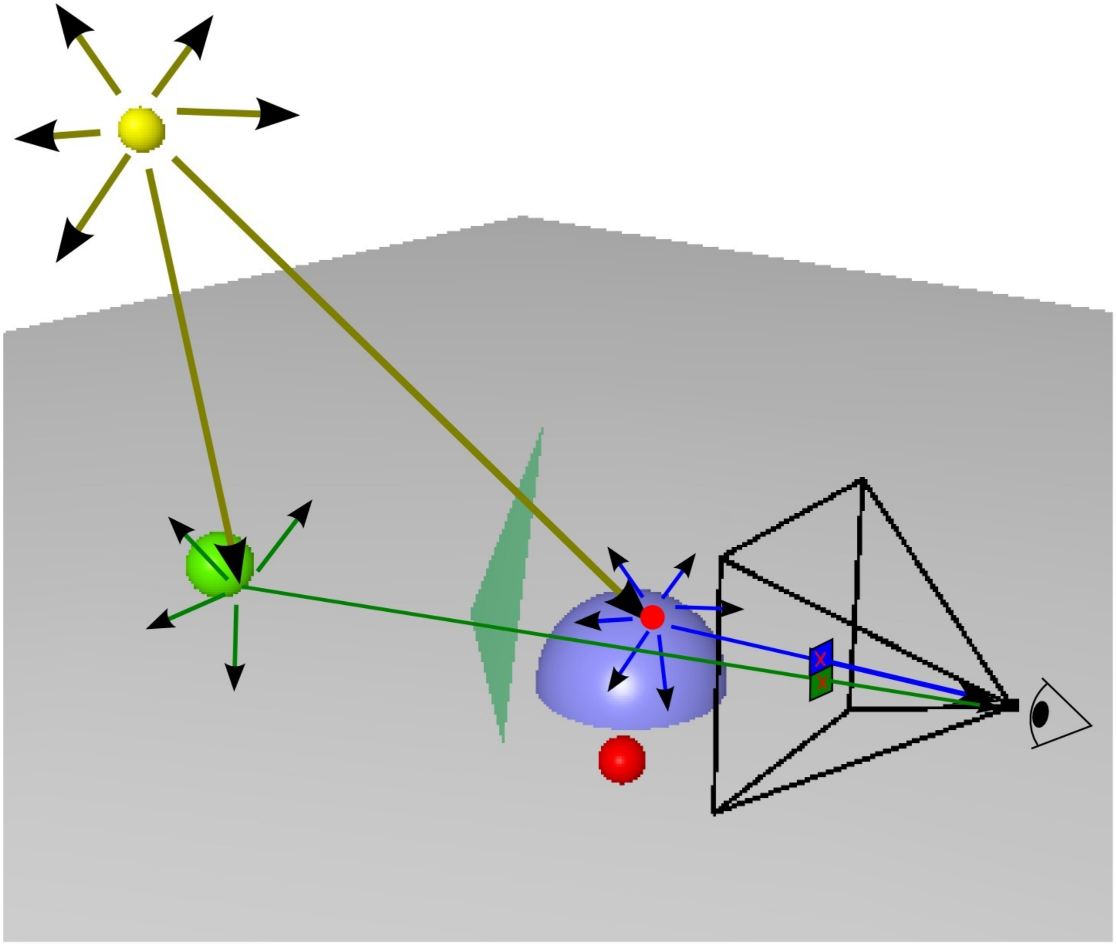
In the real world



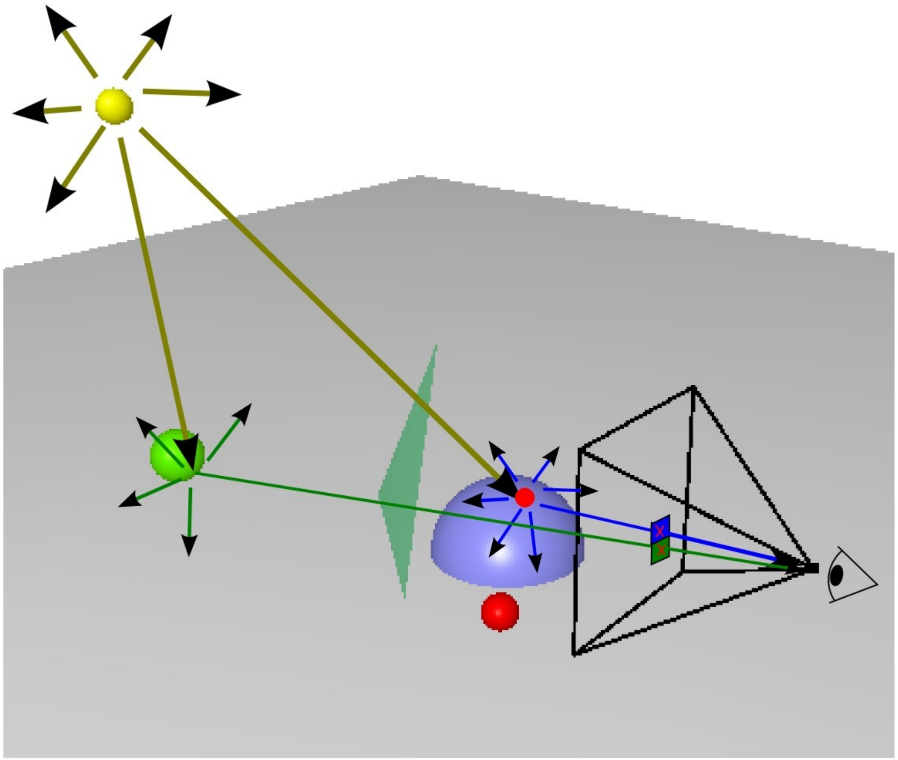
In the real world



In the real world



In the real world

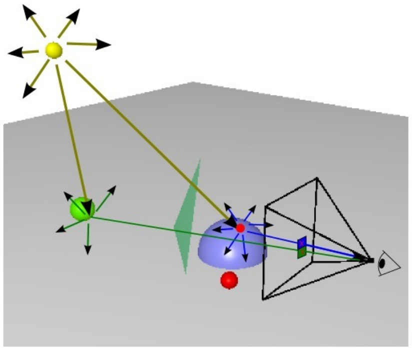


First algorithm

For all light sources
   For all direction d1
      Throw_ray(**d1**)

Throw_ray(**d**)
| If **d** intersect any object
| *Save* Color
| For all direction **d2**
| Throw_ray(**d2**)
|
| If d intersect screnn
| *Set* current Color to *pixel*

Our 1st model



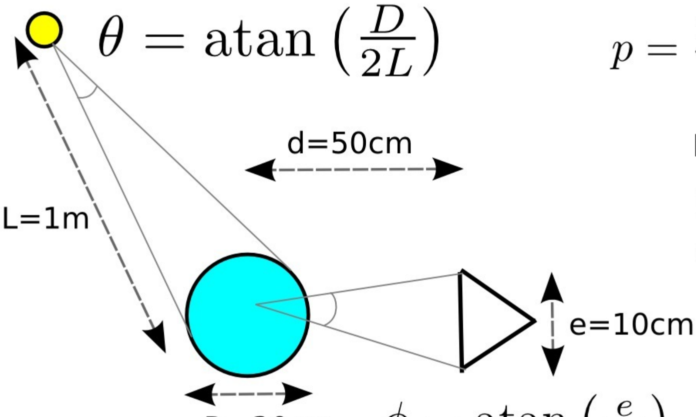
+ Simple
+ Physically accurate

- Algorithmic complexity

→ Inifinite number of reccursions

ex.

$$\theta = \mathrm{atan}\left(\frac{D}{2L}\right)$$



$$\phi = \mathrm{atan}\left(\frac{e}{2d}\right)$$

$$p = \frac{2\pi(1-\cos(\theta))}{4\pi} \times \frac{2\pi(1-\cos(\phi))}{4\pi}$$

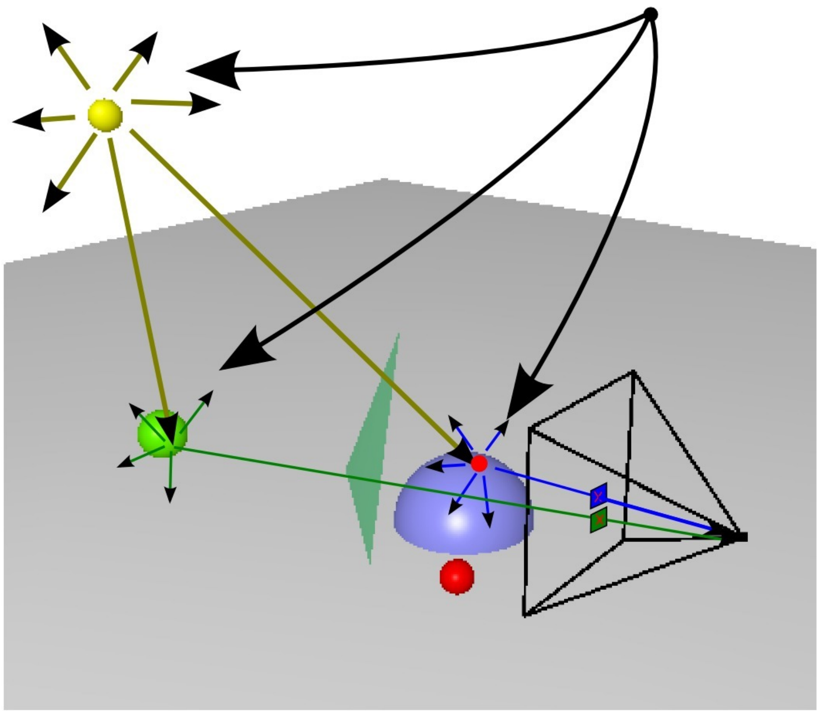
p touch object=0.0015%

p touch pixel=0.0000000007%

send 130 000 000 000 rays

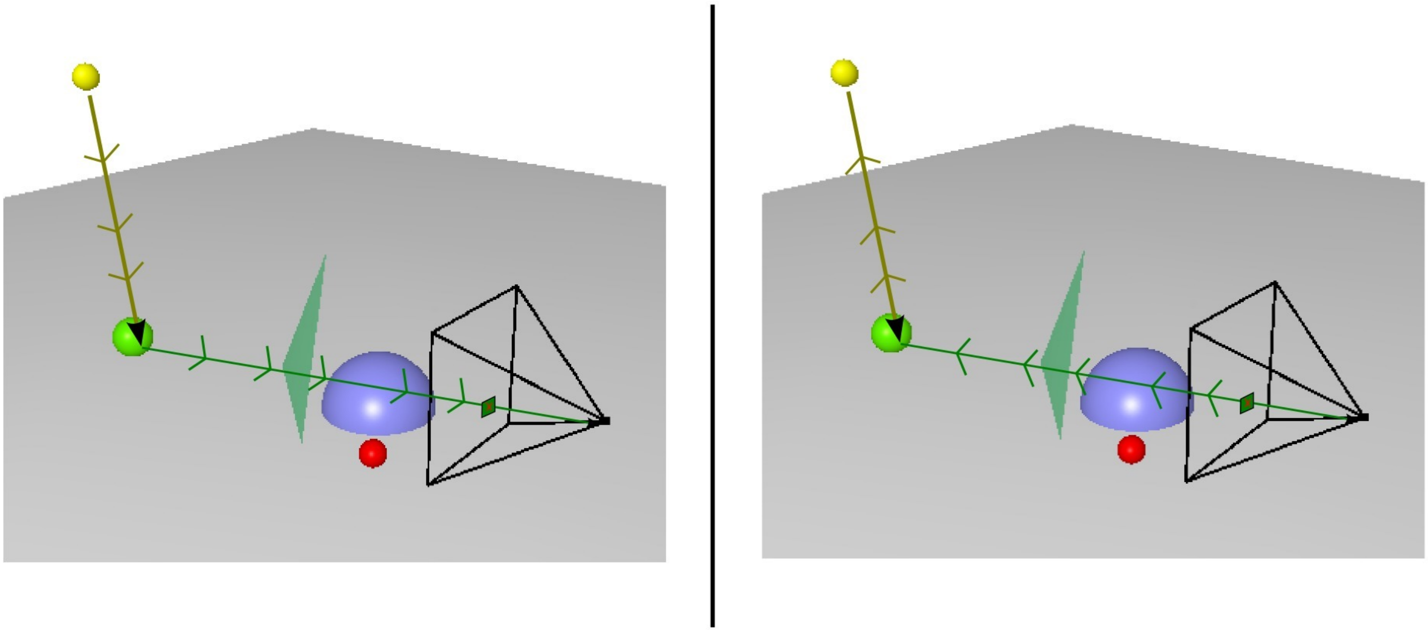
Accelerating the rendering

Problem : lot of useless rays



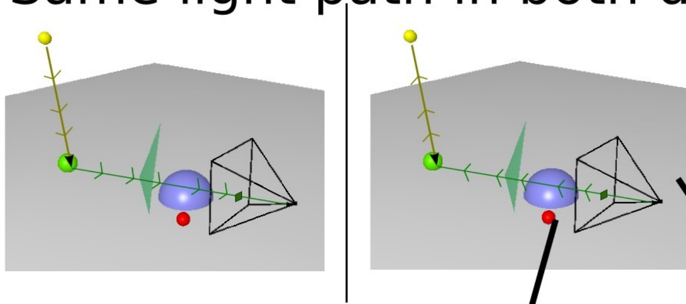
Accelerating the rendering

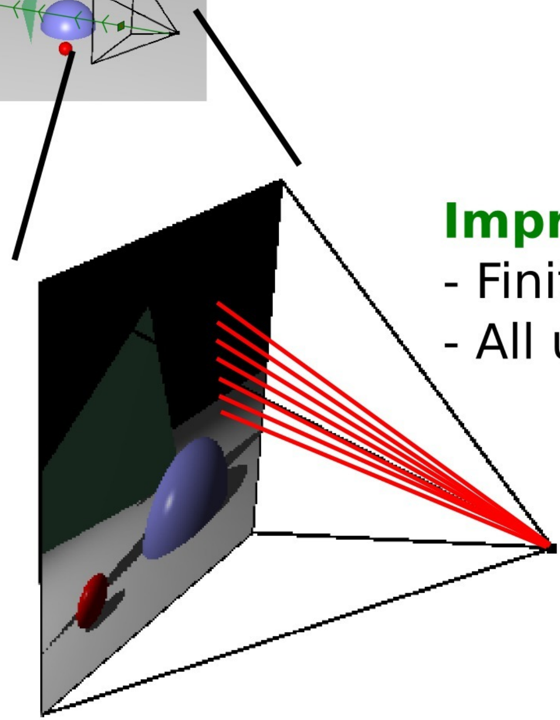
Fermat principle:
Same light path in both directions



Accelerating the rendering

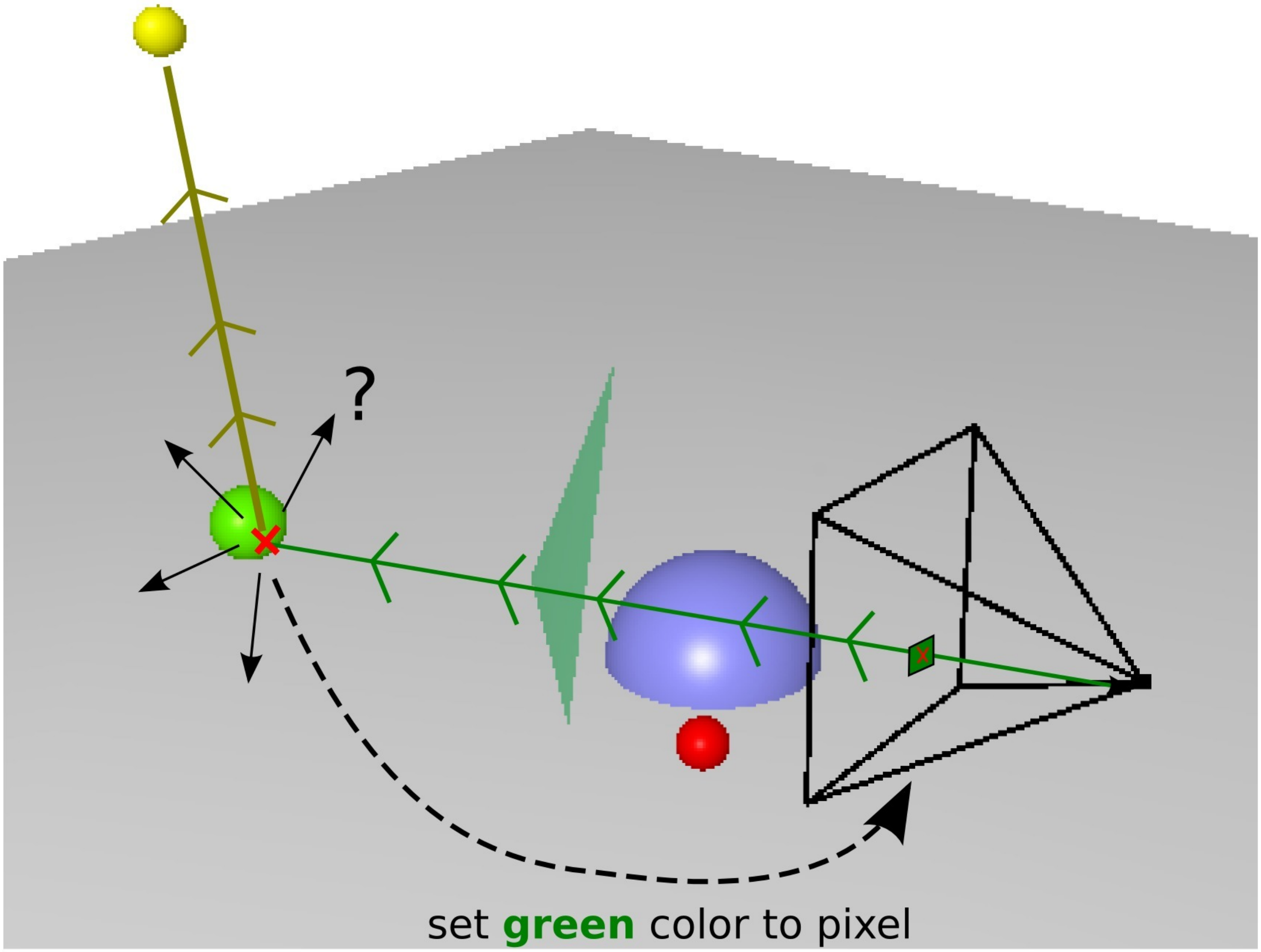
Fermat principle:
Same light path in both directions



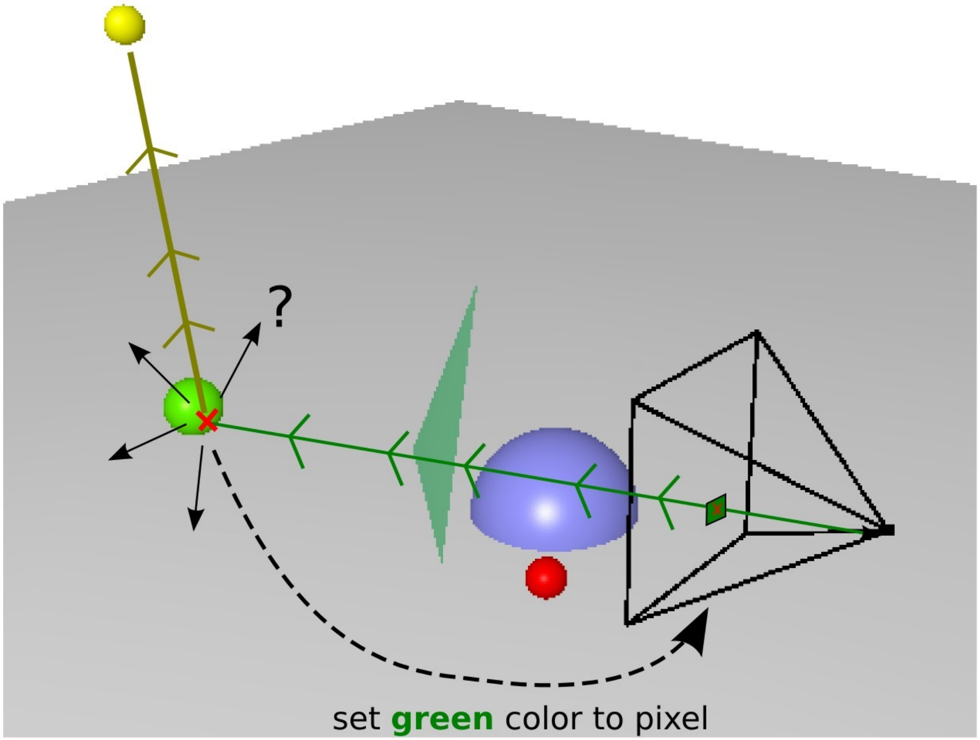


Improvement:
- Finite number of rays
- All usefull

Ray tracing algorithm



Ray tracing algorithm



1/ Fermat principle

+ Physically OK

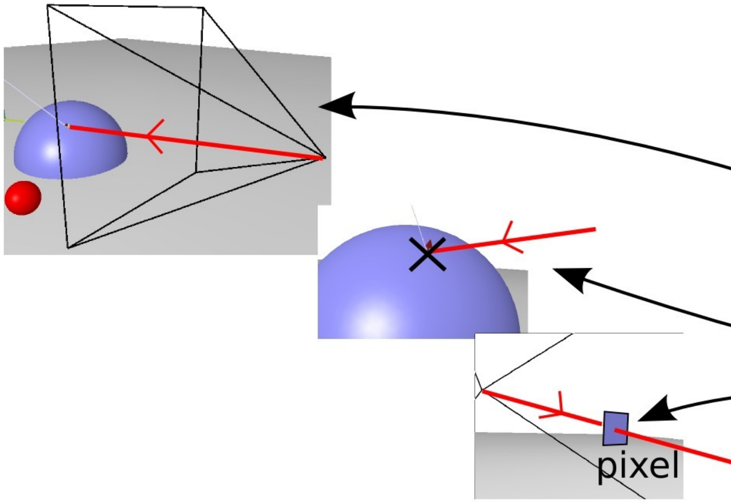
2/ No secondary diffusion

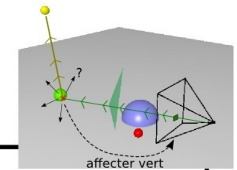
- Loss of physics

quantification:
~2 000 000 rays

0.1 ms/rays :
3 min/pic
VS 150 days/pic

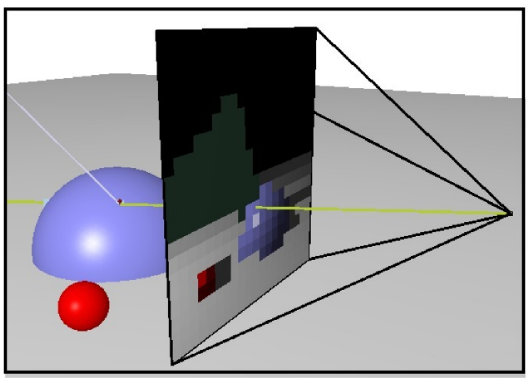
Ray tracing algorithm



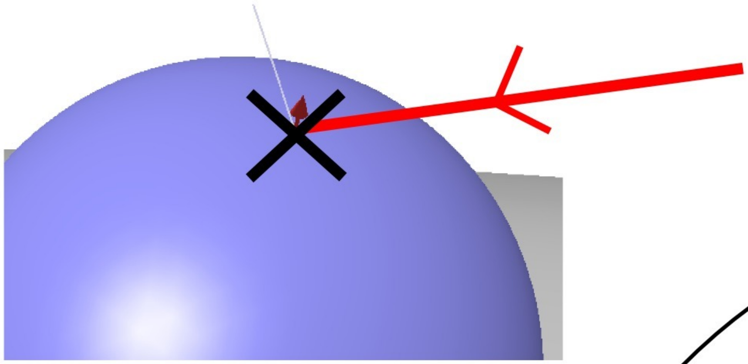


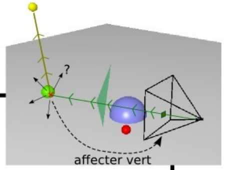
For k=1..N_pixels
  Throw_ray(**d**$_k$)

Throw_ray(**d**)
| If d intersect object
| *Set* object Color to Pixel



Ray-tracing summary





For k=1..N_pixels
   Throw_ray($\mathbf{d}_k$)

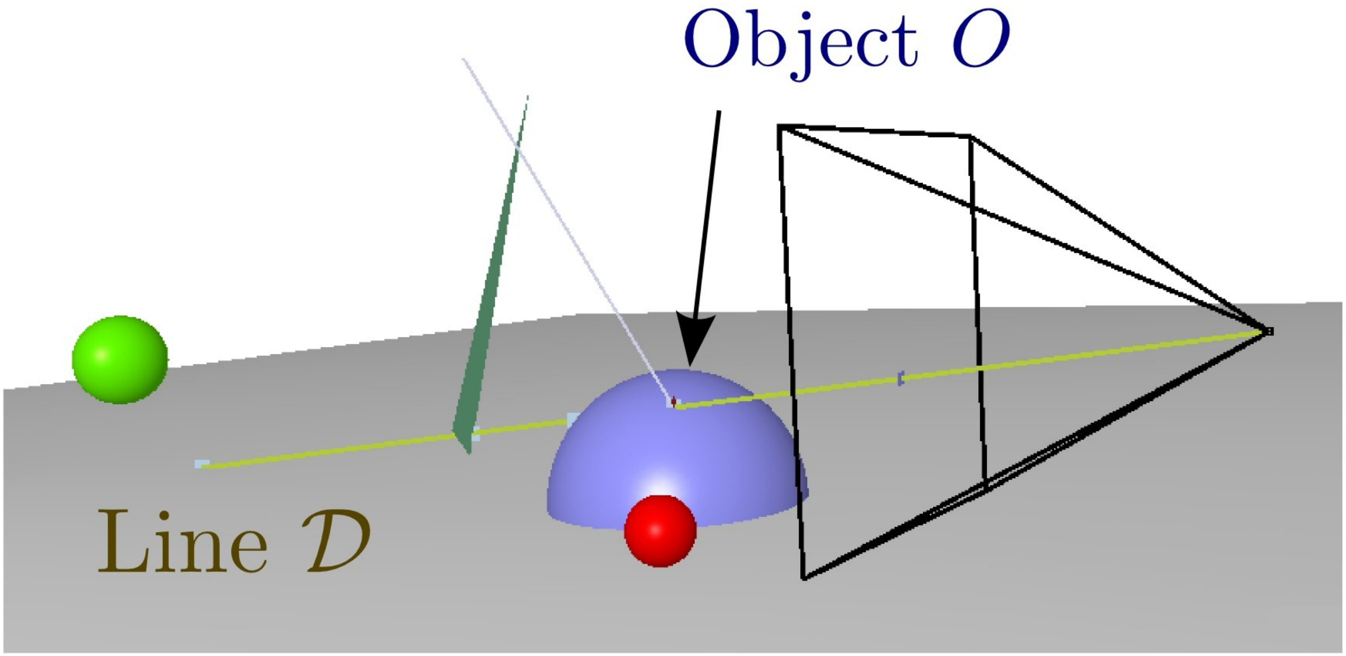
Throw_ray($\mathbf{d}$)
| If d intersect object
| *Set* object Color to Pixel

Computation complexity

Formalization

For all lines $\mathcal{D}$

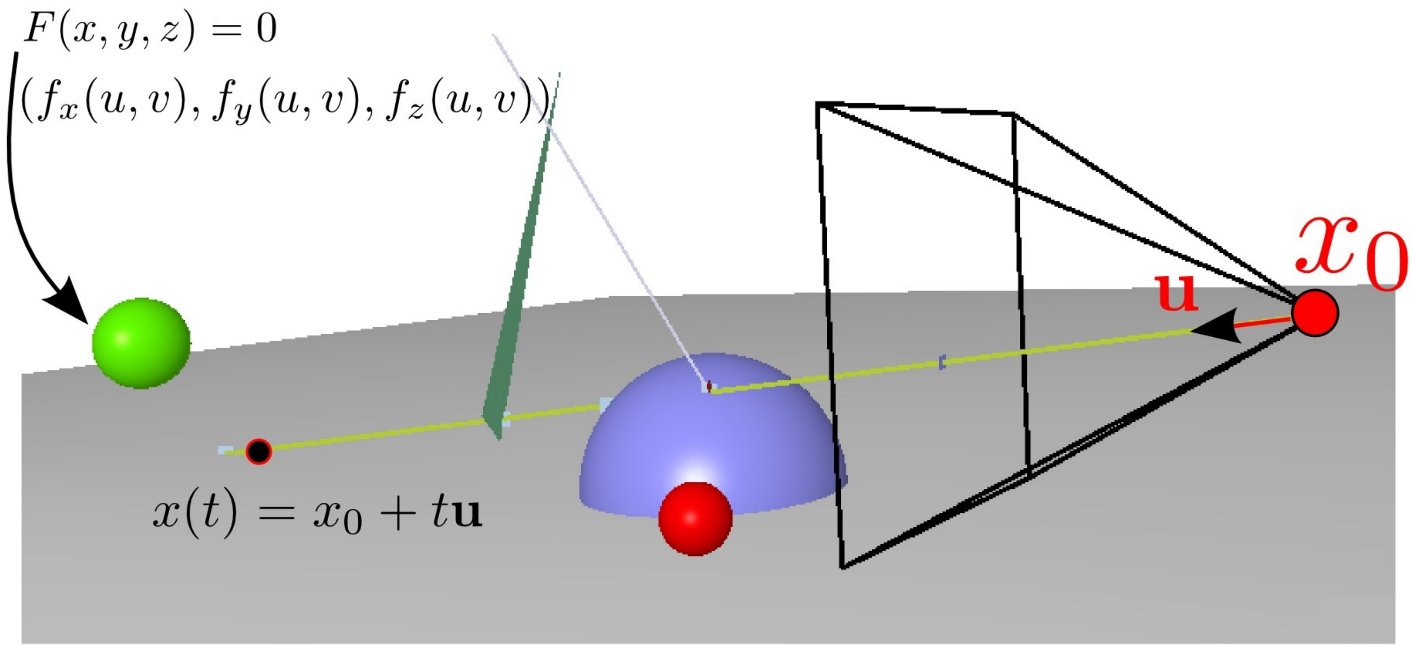
Compute $D \cap O$



Formalization

For all lines $\mathcal{D}$

Compute $D \cap O$



Formalization

1: We search

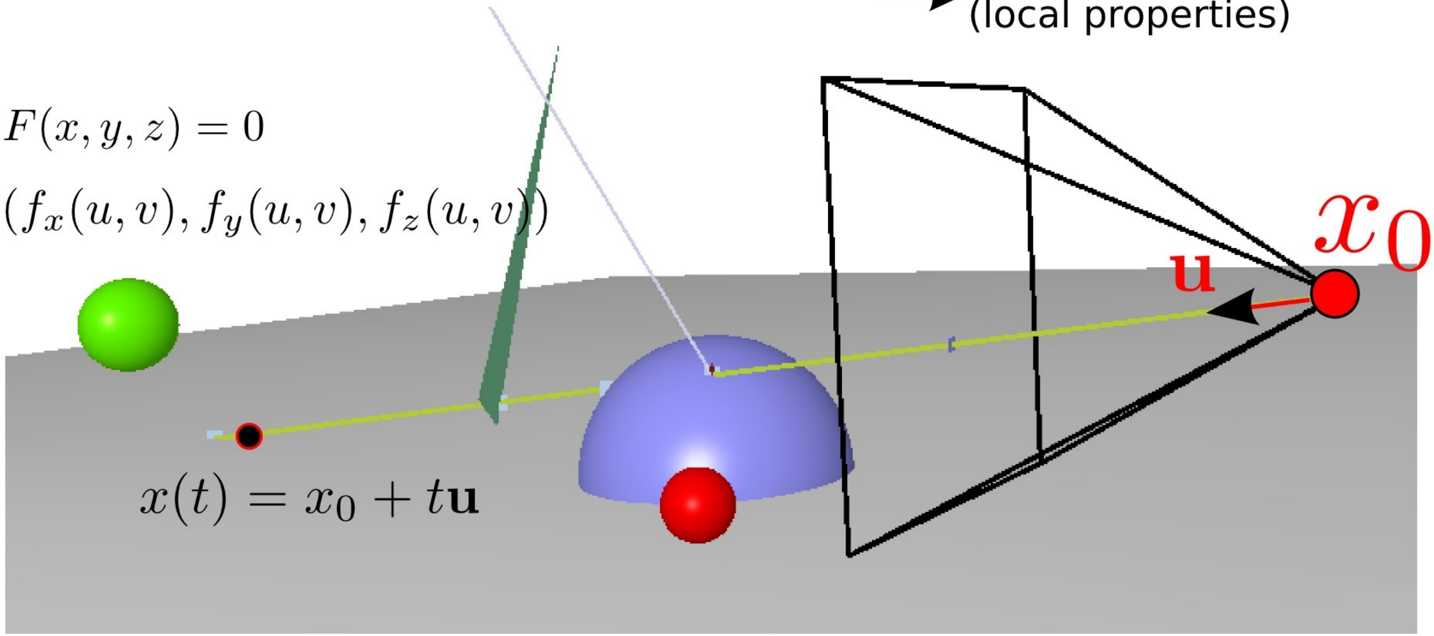
$$\begin{cases} F(x,y,z) = 0 \\ x(t) = x_0 + t\mathbf{u} \end{cases}$$

2: Solve for t

Deduce:
x(t) et **n**(t)

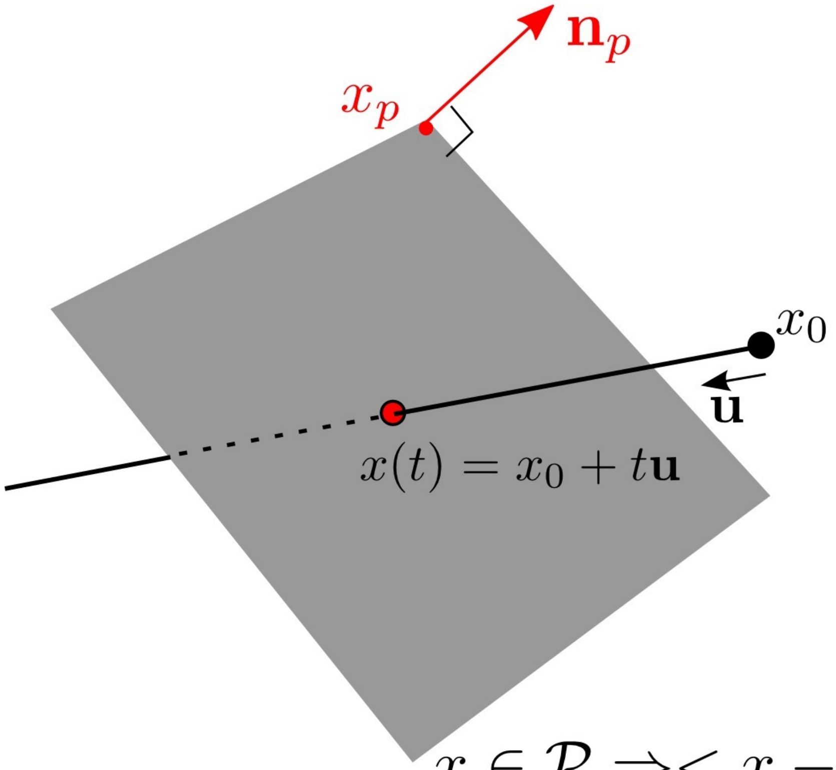
→ Keep only t>0

→ Shading
(local properties)



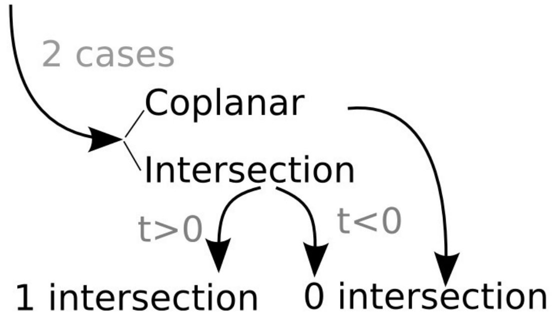
2/ Application for simple 3D scenes

Plane



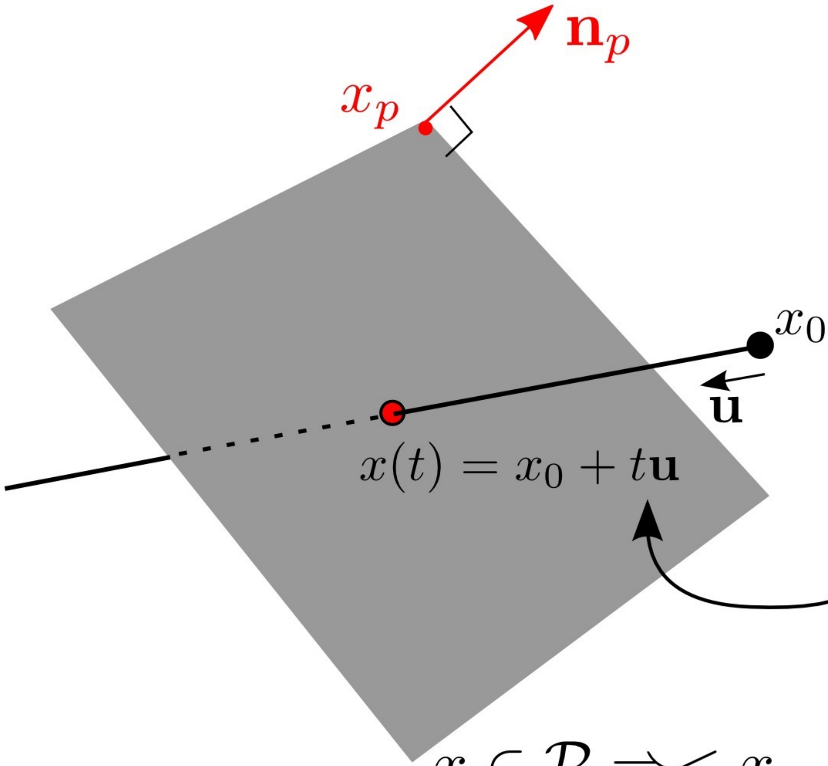
System to solve:

$$\begin{cases} x(t) = x_0 + t\mathbf{u} \\ < x(t) - x_p, \mathbf{n}_p >= 0 \end{cases}$$



$$x \in \mathcal{P} \Rightarrow < x - x_p, \mathbf{n}_p >= 0$$

Plane



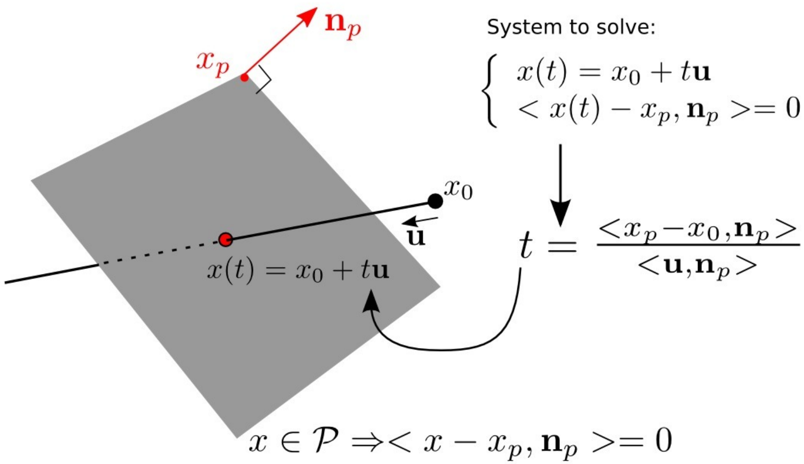
System to solve:

$$\begin{cases} x(t) = x_0 + t\mathbf{u} \\ < x(t) - x_p, \mathbf{n}_p >= 0 \end{cases}$$

$$t = \frac{<x_p - x_0, \mathbf{n}_p>}{<\mathbf{u}, \mathbf{n}_p>}$$

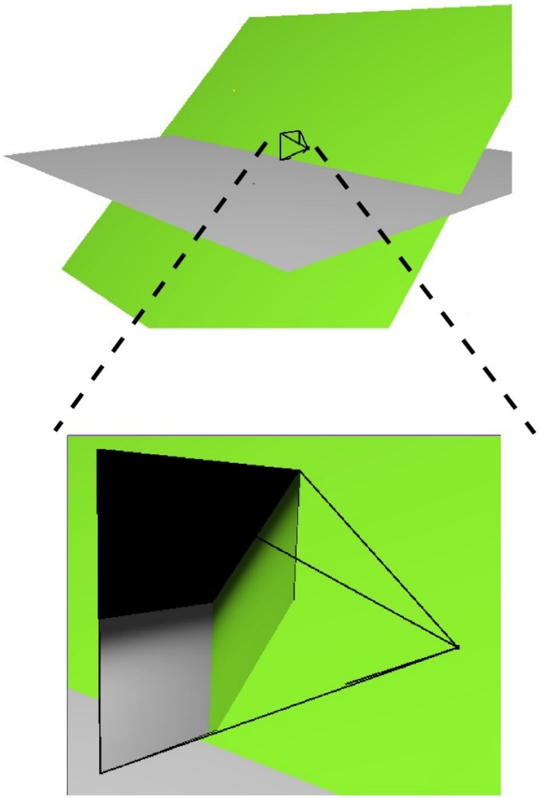
$$x \in \mathcal{P} \Rightarrow < x - x_p, \mathbf{n}_p >= 0$$

Plane

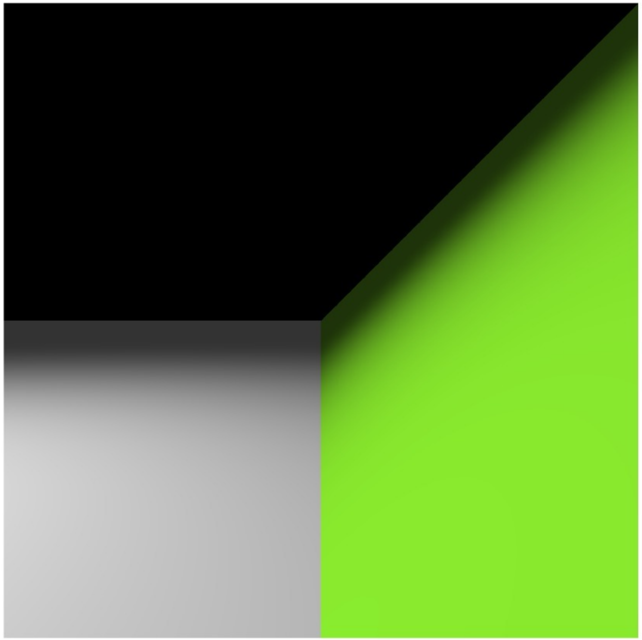



```
intersection_data intersect(v3 xp,v3 np,v3 xs,v3 u)
{
    double epsilon=1e-8;

    //<u,np>
    double proj=u.dot(np);

    //parrallel ray
    if(std::fabs(proj)<epsilon)
        return inter;//no intersection

    //t-intersection
    double t=(xp-xs).dot(n)/proj;

    inter.push_back(intersection_data( seg(t), n ,t ));

    return inter;
}
```

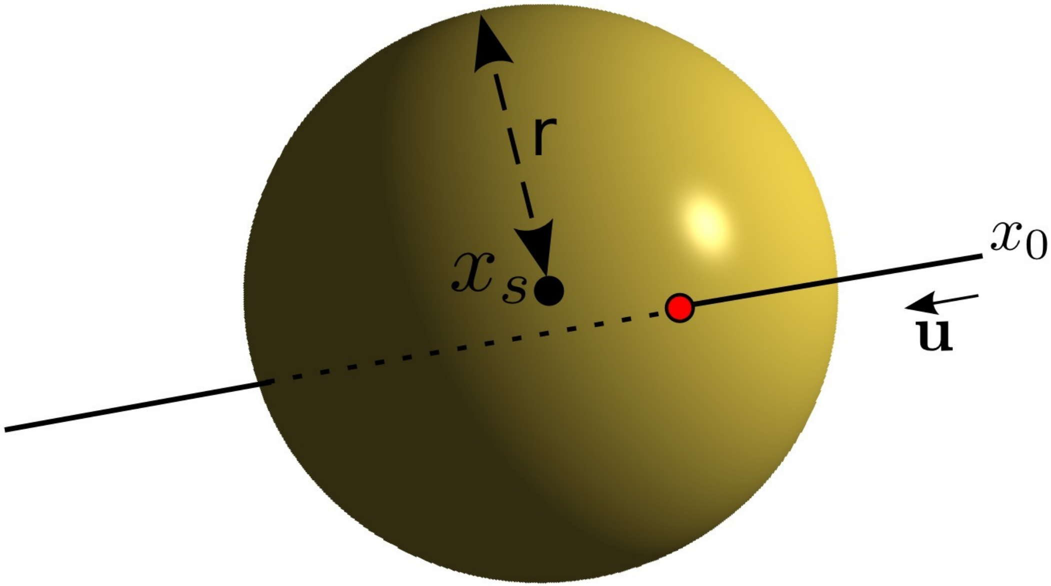
Plane



```
intersection_data intersect(v3 xp,v3 np,v3 xs,v3 u)
{
    double epsilon=1e-8;

    //<u,np>
    double proj=u.dot(np);

    //parrallel ray
    if(std::fabs(proj)<epsilon)
        return inter;//no intersection

    //t-intersection
    double t=(xp-xs).dot(n)/proj;

    inter.push_back(intersection_data( seg(t), n ,t ));

    return inter;
}
```

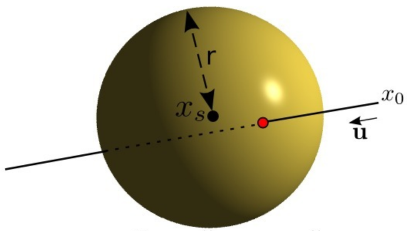


Sphere



$$x \in \mathcal{S} \Rightarrow \|x - x_s\| = r$$

Sphere



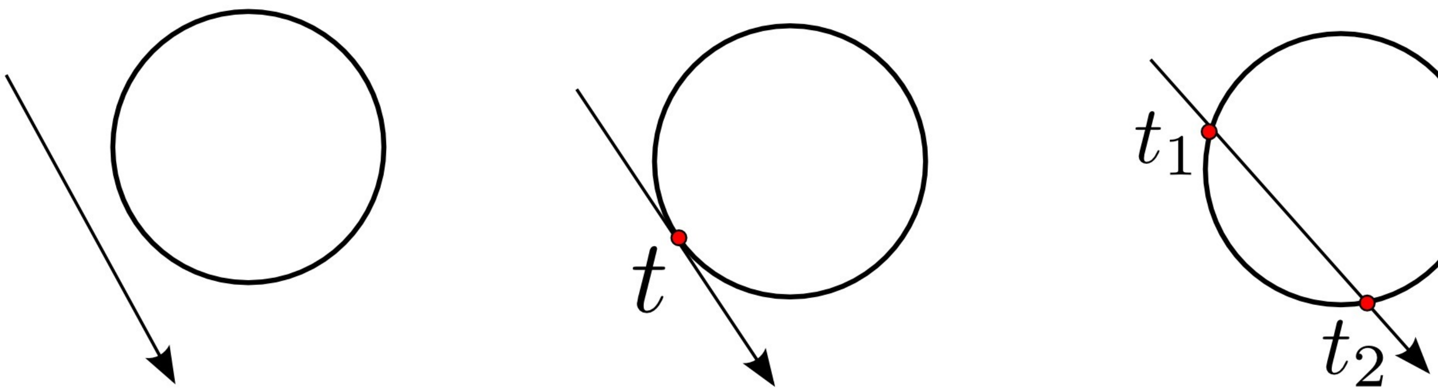
$x \in \mathcal{S} \Rightarrow \|x - x_s\| = r$

$$\begin{cases} x(t) = x_0 + t\mathbf{u} \\ \|x(t) - x_s\| = r \end{cases}$$

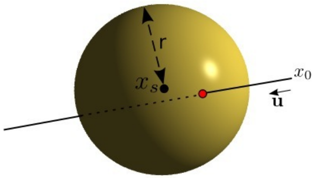
Case 1:

Case 2:

Case 3:



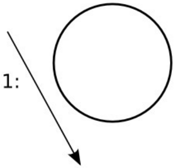
Sphere



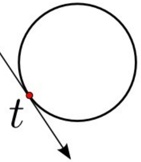
$$\begin{cases} x(t) = x_0 + t\mathbf{u} \\ \|x(t) - x_s\| = r \end{cases}$$

$$t^2 + 2t < x_0 - x_s, \mathbf{u} > + \left(\|x_0 - x_s\|^2 - r^2\right) = 0$$

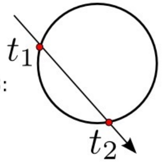
Case 1:



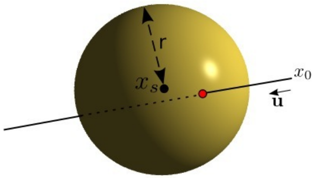
Case 2:



Case 3:



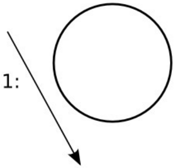
Sphere



$$\begin{cases} x(t) = x_0 + t\mathbf{u} \\ \|x(t) - x_s\| = r \end{cases}$$

$$t^2 + 2t < x_0 - x_s, \mathbf{u} > + \left(\|x_0 - x_s\|^2 - r^2\right) = 0$$

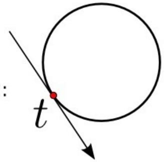
Case 1:



$$\Delta' =< x_0 - x_s, \mathbf{u} >^2 - \left(\|x_0 - x_s\|^2 - r^2\right)$$

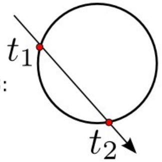
$$t_{1/2} = - < x_0 - x_s, \mathbf{u} > \pm\sqrt{\Delta}$$

Case 2:



$$\mathbf{n}(t) = \frac{x(t) - x_s}{\|x(t) - x_s\|}$$

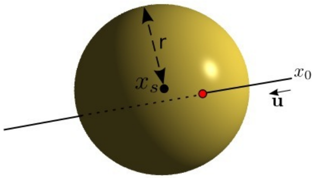
Case 3:



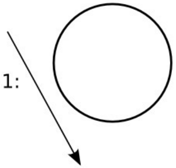
Sphere

$$\begin{cases} x(t) = x_0 + t\mathbf{u} \\ \|x(t) - x_s\| = r \end{cases}$$

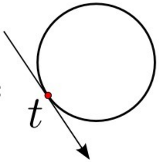
$$t_{1/2} = - < x_0 - x_s, \mathbf{u} > \pm\sqrt{\Delta}$$



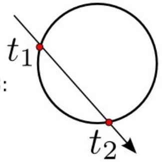
Case 1:



Case 2:

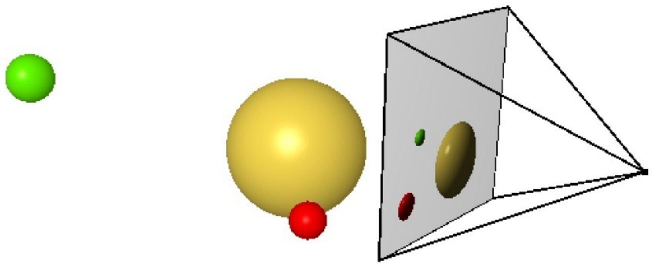


Case 3:



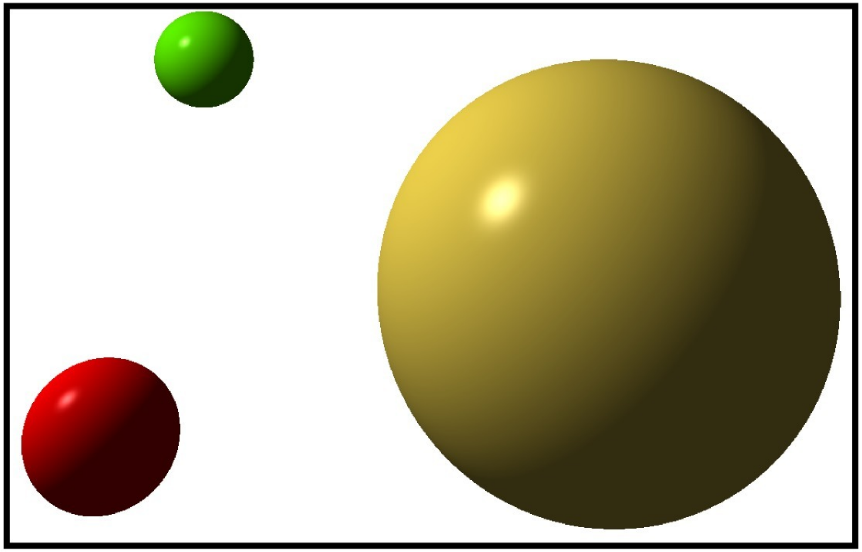

```
std::vector<intersection_data> sphere::intersect(const ray& seg) const
{
    std::vector<intersection_data> inter;

    v3 v=seg.x0()-x0;
    double a=seg.u().dot(seg.u());
    double b=2*v.dot(seg.u());
    double c=v.dot(v)-r*r;

    double delta=b*b-4*a*c;

    //no intersection
    if(delta<0)
        return inter;

    double epsilon=1e-8;

    if(std::fabs(delta)<epsilon) //1-intersection points
    {
        double t=-b/(2*a);
        v3 x_inter=seg(t);
        v3 n_inter=(x_inter-x0).normalized();
        inter.push_back(intersection_data(x_inter,n_inter,t));
    }
    else //2-intersection points (keep the first one)
    {
        double sqrt_delta=std::sqrt(delta);
        double t1=(-b+sqrt_delta)/(2*a);
        double t2=(-b-sqrt_delta)/(2*a);

        v3 x1_inter=seg(t1);
        v3 x2_inter=seg(t2);

        v3 n1_inter=(x1_inter-x0).normalized();
        v3 n2_inter=(x2_inter-x0).normalized();

        inter.push_back(intersection_data(x1_inter,n1_inter,t1));
        inter.push_back(intersection_data(x2_inter,n2_inter,t2));
    }

    return inter;
```

Sphere



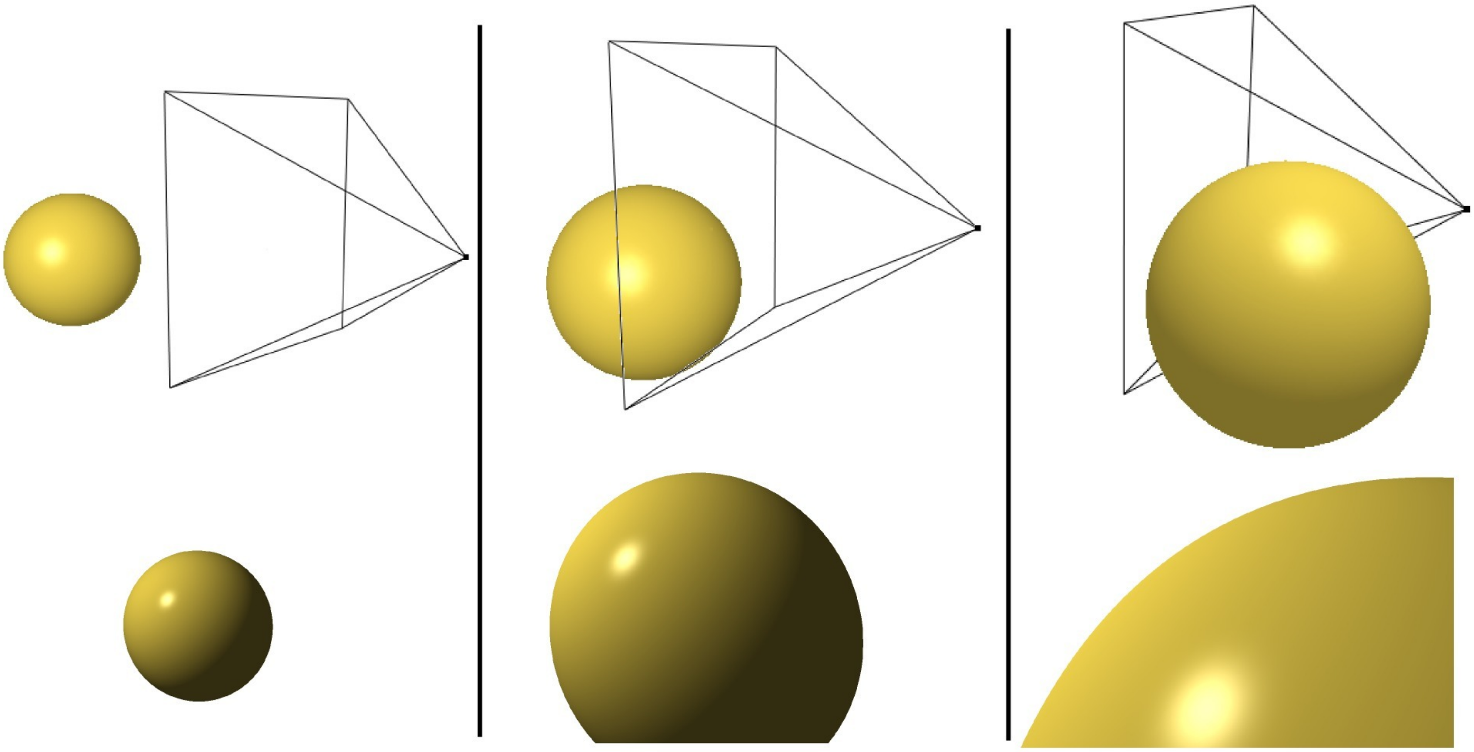
$$\Delta' = < x_0 - x_s, \mathbf{u} >^2 - \left(\|x_0 - x_s\|^2 - r^2\right)$$

$$t_{1/2} = - < x_0 - x_s, \mathbf{u} > \pm\sqrt{\Delta}$$



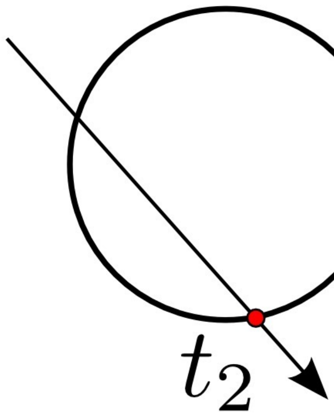
Sphere

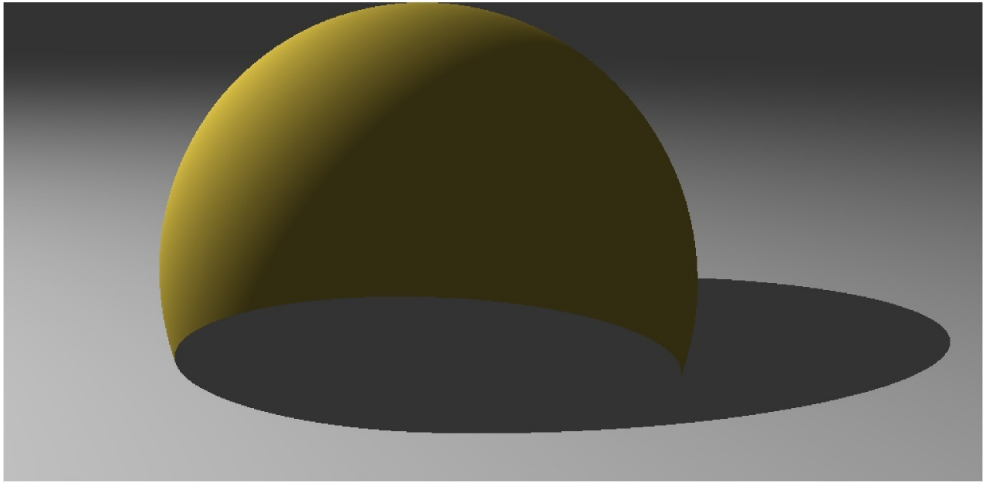
Real sphere model: no discretization



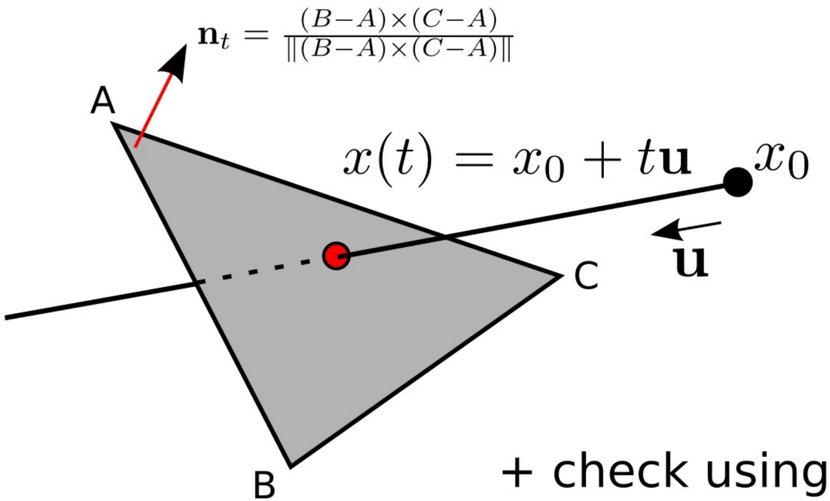
Sphere

Note: Using the wrong intersection





Triangle



Same as plane

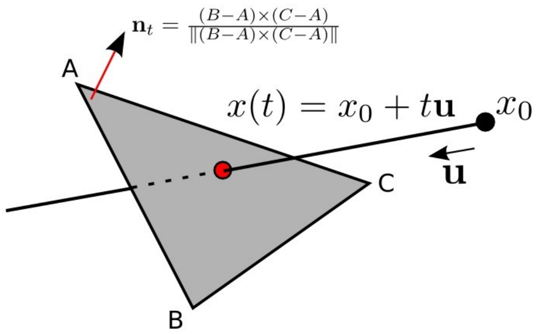
$$t = \frac{<A - x_0, \mathbf{n}_t>}{<\mathbf{u}, \mathbf{n}_t>}$$

+ check using barycentric coordinates

$$x = \alpha A + \beta B + \gamma C$$

$$x \in \mathcal{T} \Rightarrow \begin{pmatrix} \alpha + \beta + \gamma = 1 \\ 0 \leq \alpha \leq 1 \\ 0 \leq \beta \leq 1 \\ 0 \leq \gamma \leq 1 \end{pmatrix}$$

Triangle: barycentric coordinates



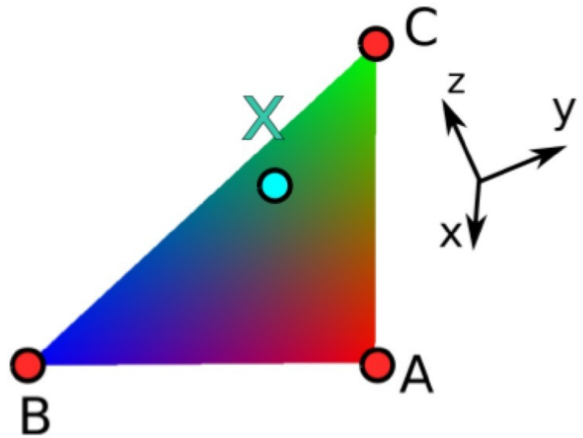
$$x = \alpha A + \beta B + \gamma C$$

$$\begin{cases} \alpha + \beta + \gamma = 1 \\ 0 \leq \alpha \leq 1 \\ 0 \leq \beta \leq 1 \\ 0 \leq \gamma \leq 1 \end{cases}$$

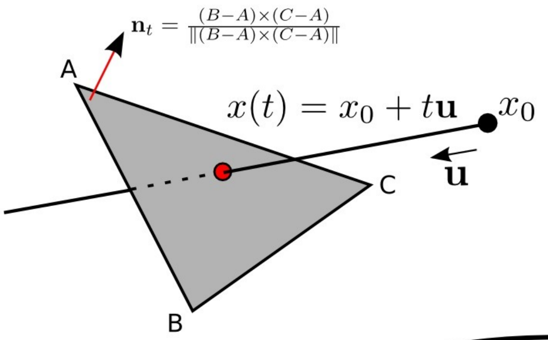
$$\begin{cases} A = \text{area}(\mathbf{x}_B - \mathbf{x}_A, \mathbf{x}_C - \mathbf{x}_A) \\ A_1 = \text{area}(\mathbf{x}_C - \mathbf{x}_B, \mathbf{x} - \mathbf{x}_B) \\ A_2 = \text{area}(\mathbf{x}_A - \mathbf{x}_C, \mathbf{x} - \mathbf{x}_C) \\ A_3 = \text{area}(\mathbf{x}_B - \mathbf{x}_A, \mathbf{x} - \mathbf{x}_A) \end{cases}$$

with $\text{area}(\mathbf{v}_0, \mathbf{v}_1) = 1/2\|\mathbf{v}_0 \times \mathbf{v}_1\|$

$$\Rightarrow \begin{cases} \alpha = A_1/A \\ \beta = A_2/A \\ \gamma = A_3/A \end{cases}$$



Triangle: barycentric coordinates

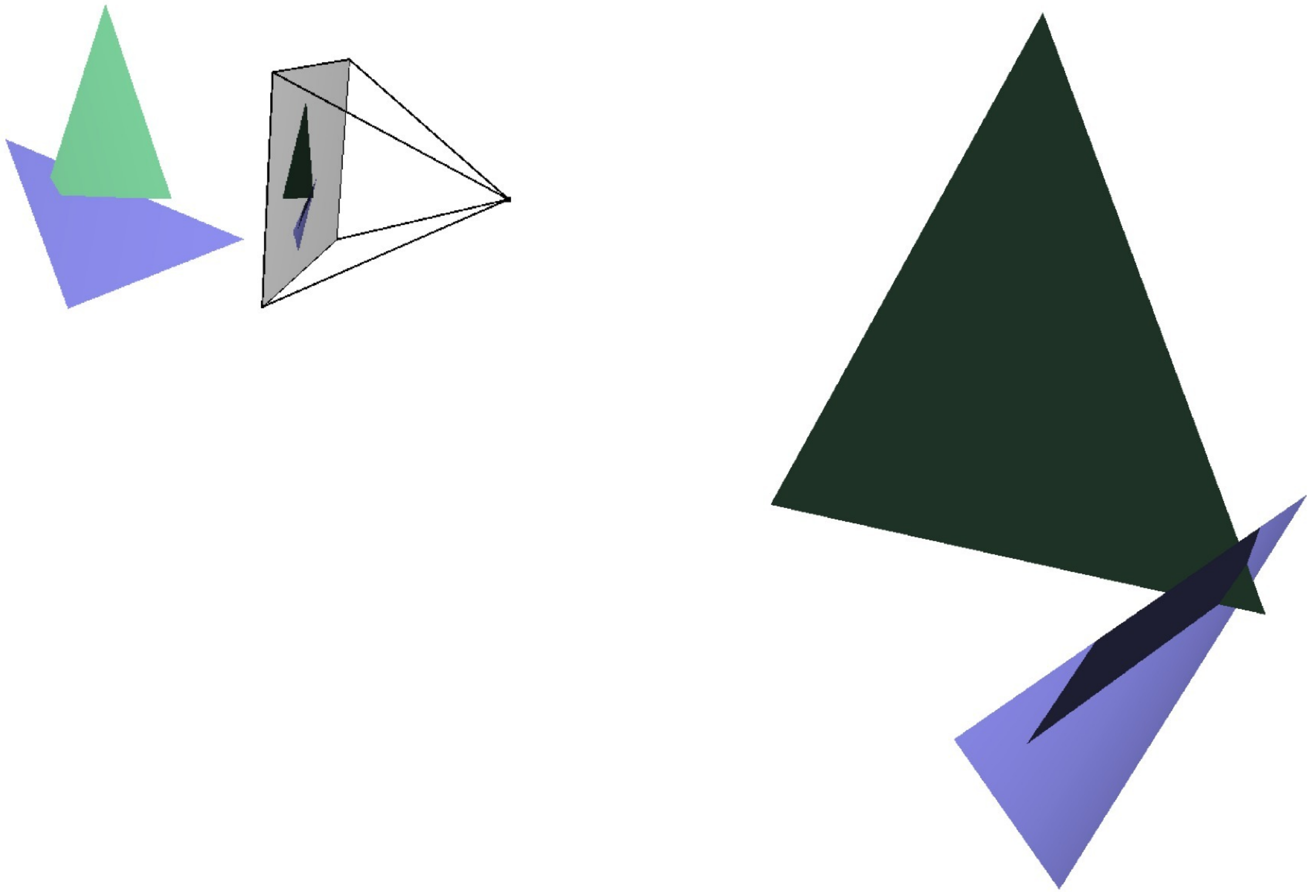


$$t = \frac{<A-x_0,\mathbf{n}_t>}{<\mathbf{u},\mathbf{n}_t>}$$

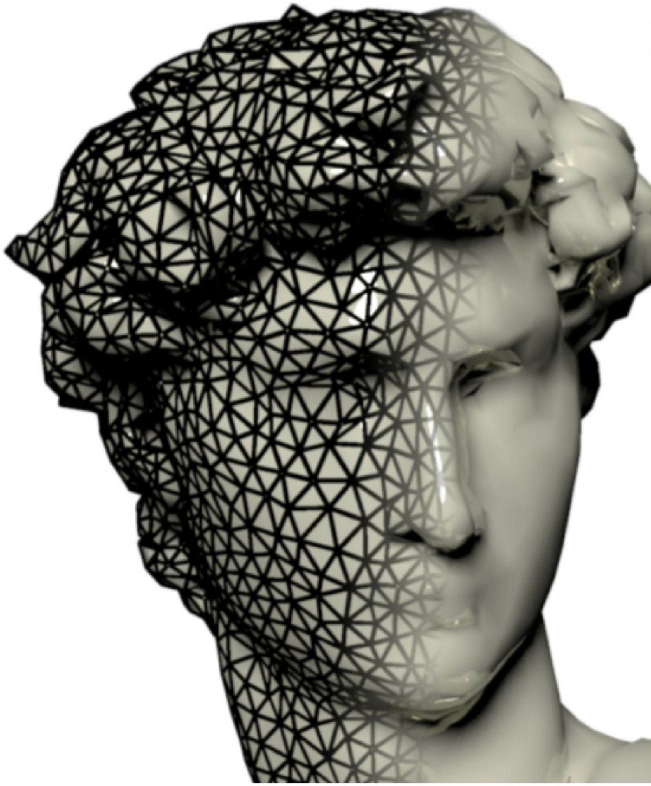
$$\begin{cases} \alpha+\beta+\gamma = 1 \\ 0 \leq \alpha \leq 1 \\ 0 \leq \beta \leq 1 \\ 0 \leq \gamma \leq 1 \end{cases}$$

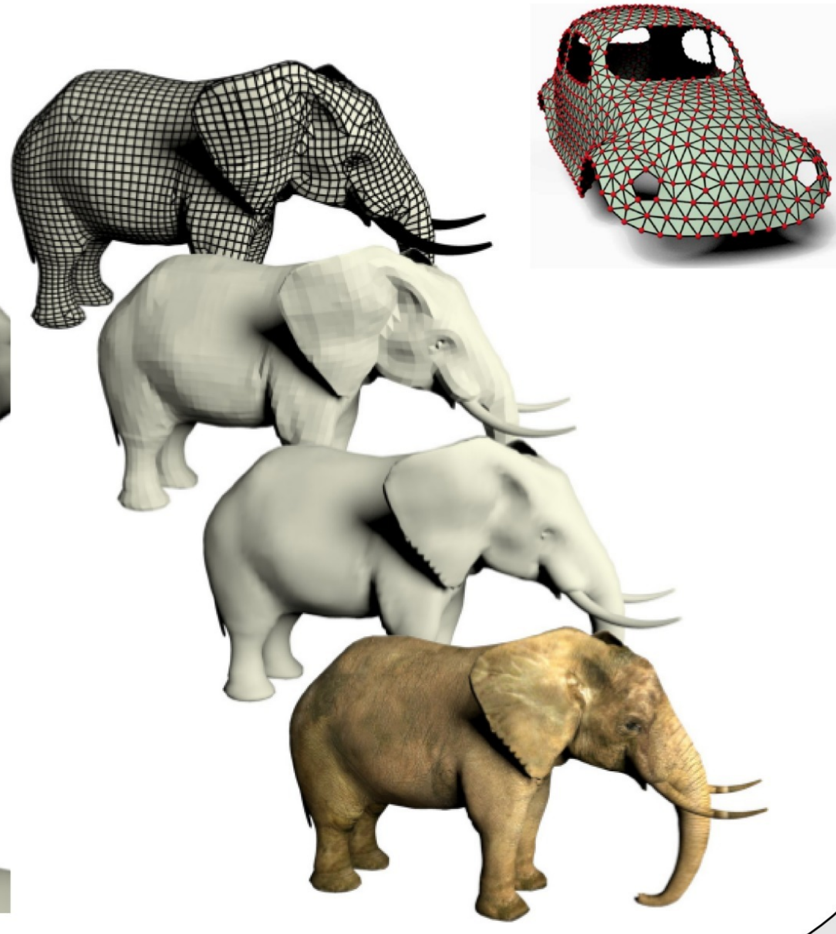
```
v3 x10=internal_x1-internal_x0;
v3 x20=internal_x2-internal_x0;

v3 u10=x10.normalized();
v3 u20=x20.normalized();

v3 n=u10.cross(u20).normalized();

const v3& u=seg.u();
double proj=u.dot(n);

double epsilon=1e-8;
if(std::fabs(proj)<epsilon)
    return inter;

double t=(internal_x0-seg.x0()).dot(n)/proj;
v3 xi=seg(t);

double area_0=(internal_x2-internal_x1).cross(xi-internal_x1).norm()/2.0;
double area_1=(internal_x0-internal_x2).cross(xi-internal_x2).norm()/2.0;
double area_2=(internal_x1-internal_x0).cross(xi-internal_x0).norm()/2.0;
double area  =x10.cross(x20).norm()/2.0;

double a=area_0/area;
double b=area_1/area;
double c=area_2/area;


if(a>=0 && b>=0 && c>=0 && a<=1 && b<=1 && c<=1)
    if(std::fabs(a+b+c-1.0)<epsilon)
        inter.push_back(intersection_data(xi,n,t));

return inter;
```

Triangle



Triangle => Mesh rendering

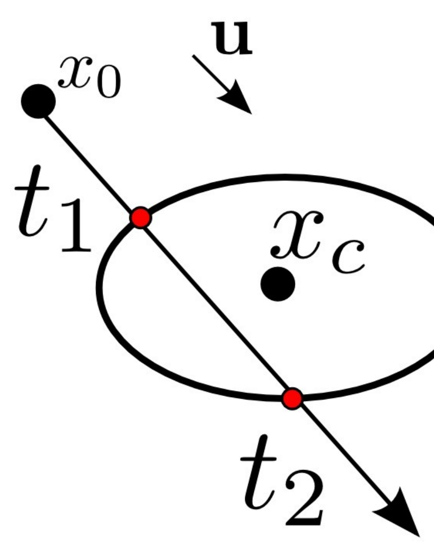




Ellipsoid

$$(x - x_c)^T \mathrm{D}(x - x_c) = 1$$

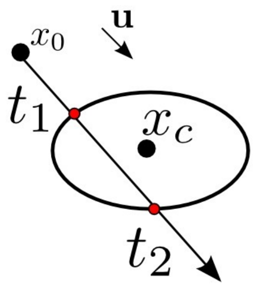
$$\mathrm{D} = \begin{pmatrix} A & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & C \end{pmatrix}$$



Ellipsoid

$$(x - x_c)^T \mathrm{D} (x - x_c) = 1$$

$$\mathrm{D} = \begin{pmatrix} A & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & C \end{pmatrix}$$



$$(\mathbf{u}^T \mathrm{D} \mathbf{u})\, t^2 + (\mathbf{y}^T \mathrm{D} \mathbf{u} + \mathbf{u}^T \mathrm{D} \mathbf{y})\, t + \mathbf{y}^T \mathrm{D} \mathbf{y} - 1 = 0$$

$$\mathbf{y} = x_0 - x_c$$

$$\Delta = (\mathbf{y}^T \mathrm{D} \mathbf{u} + \mathbf{u}^T \mathrm{D} \mathbf{y})^2 - 4(\mathbf{u}^T \mathrm{D} \mathbf{u})(\mathbf{y}^T \mathrm{D} \mathbf{y} - 1)$$

$$t_{1/2} = -\frac{\mathbf{y}^T \mathrm{D} \mathbf{u} + \mathbf{u}^T \mathrm{D} \mathbf{y} \pm \sqrt{\Delta}}{2\, \mathbf{u}^T \mathrm{D} \mathbf{u}}$$

Implicit surfaces

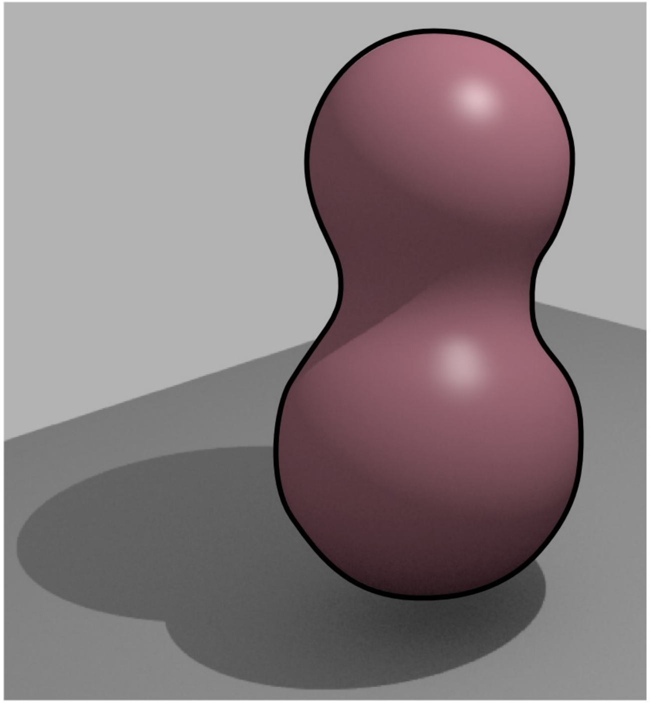
$$S = \{(x, y, z) \in \mathbb{R}^3 | F(x, y, z) = 0\}$$

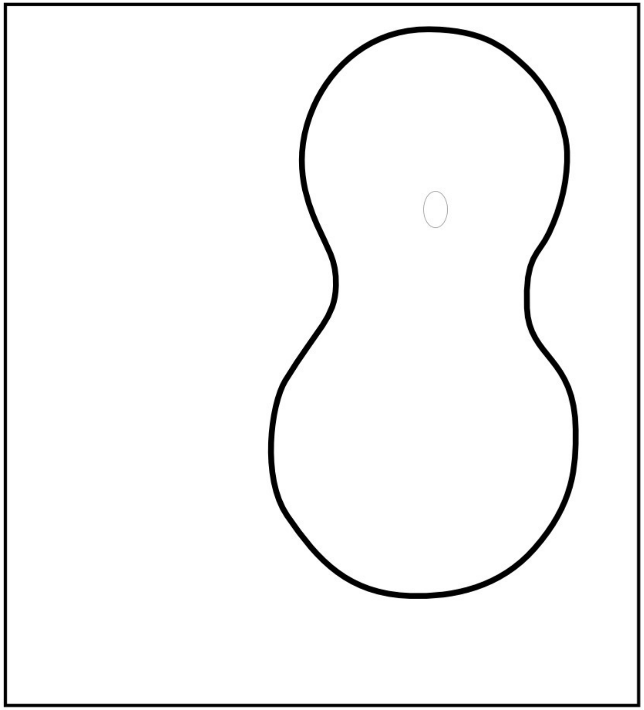
We generaly don't know the analytical solution of $S \cap \mathcal{D}$

We look for an approximation

Implicit surfaces

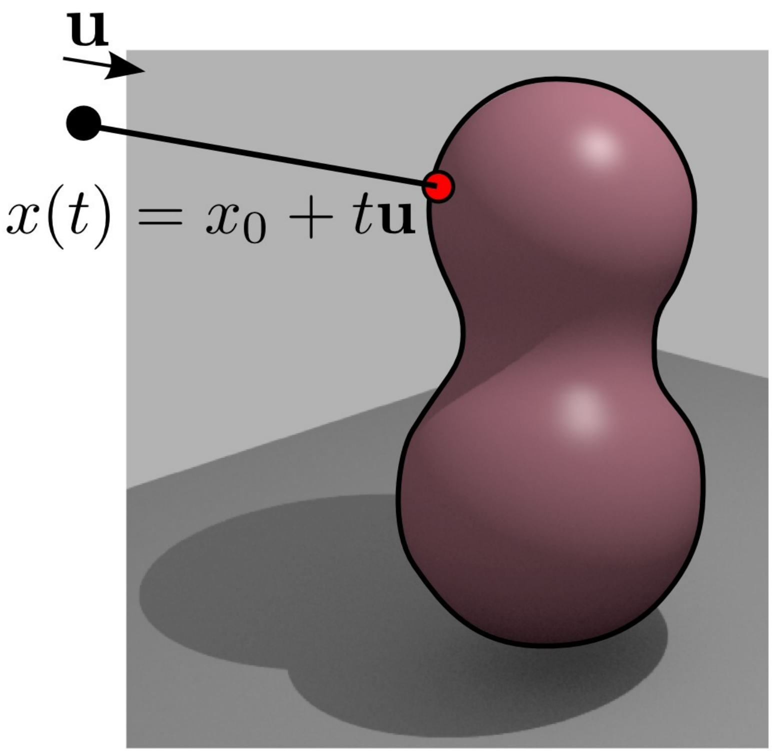
$$S = \{(x, y, z) \in \mathbb{R}^3 | F(x, y, z) = 0\}$$

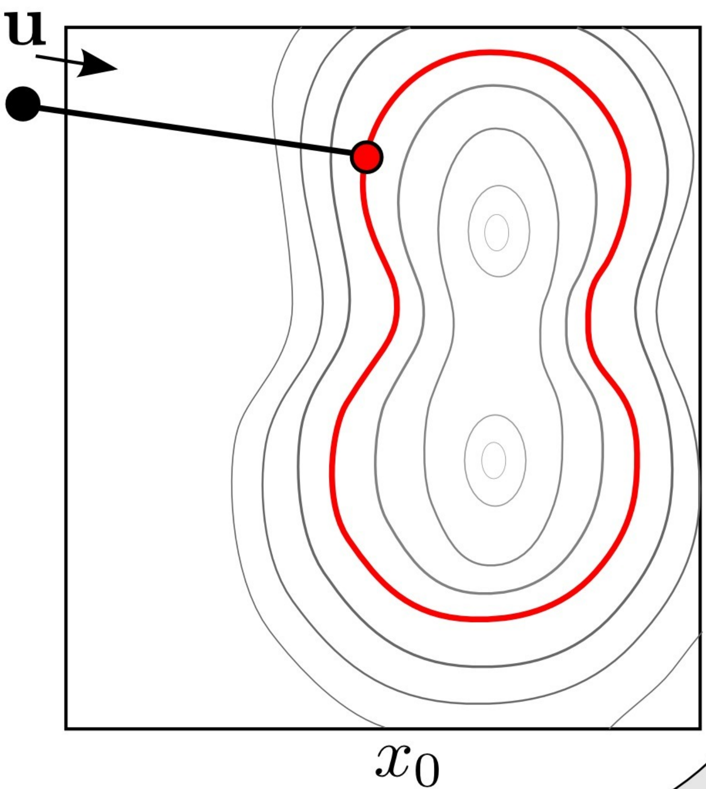




Implicit surfaces

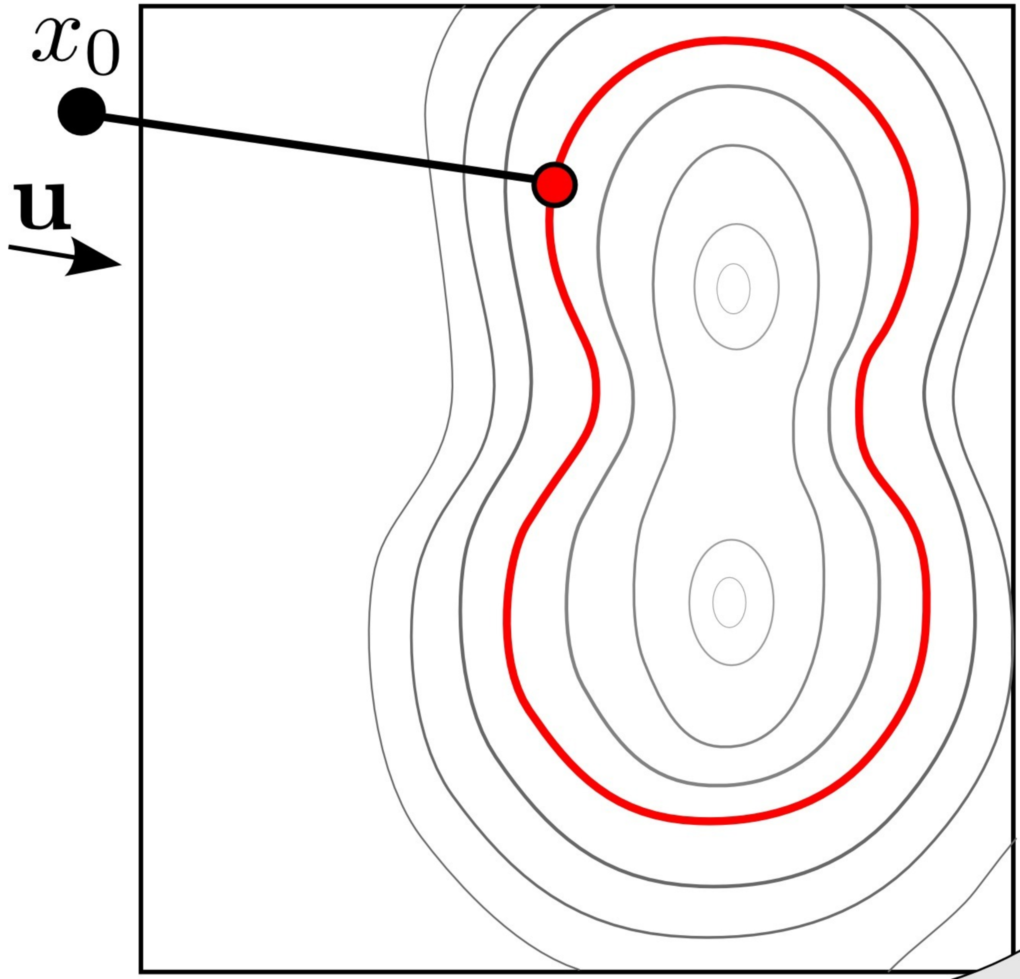
$$S = \{(x, y, z) \in \mathbb{R}^3 | F(x, y, z) = 0\}$$





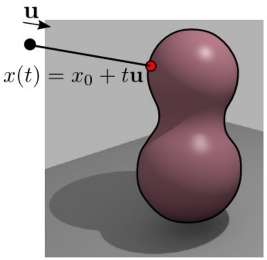
Implicit surfaces

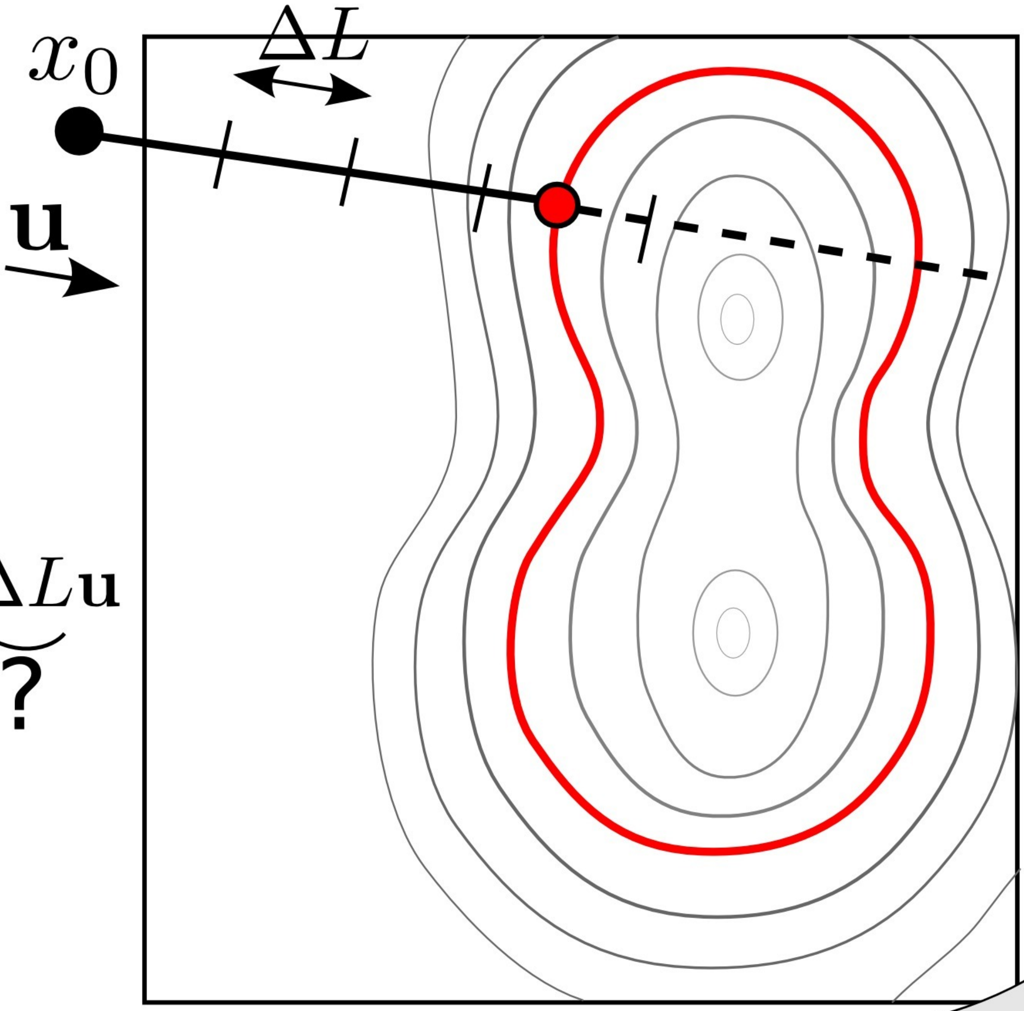
$$S = \{(x, y, z) \in \mathbb{R}^3 | F(x, y, z) = 0\}$$



Implicit surfaces

$$S = \{(x, y, z) \in \mathbb{R}^3 | F(x, y, z) = 0\}$$



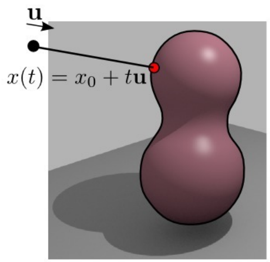


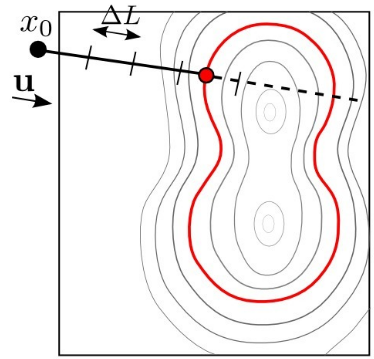
1st Solution:

$$x^{i+1} = x^i + \underbrace{\Delta L \mathbf{u}}_{?}$$

Implicit surfaces

$$S = \{(x, y, z) \in \mathbb{R}^3 | F(x, y, z) = 0\}$$



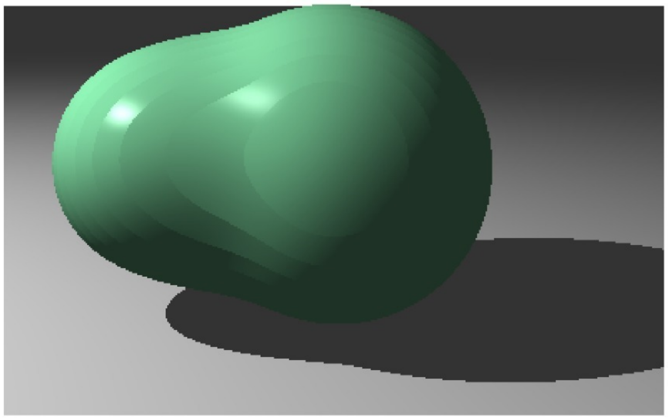


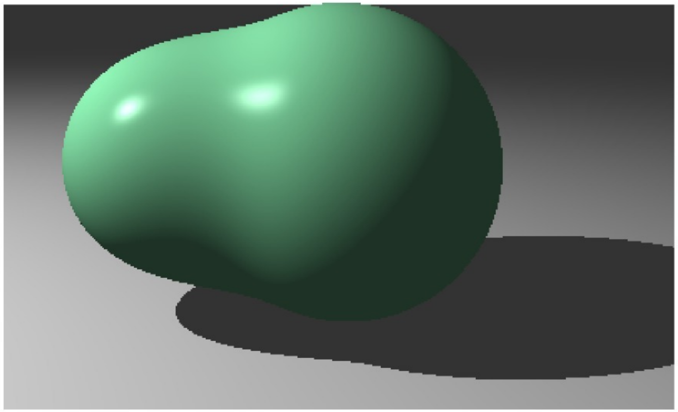
1st Solution:

$$x^{i+1} = x^i + \underbrace{\Delta L \mathbf{u}}$$

0.2

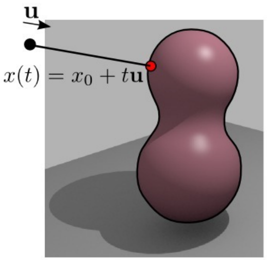
0.001

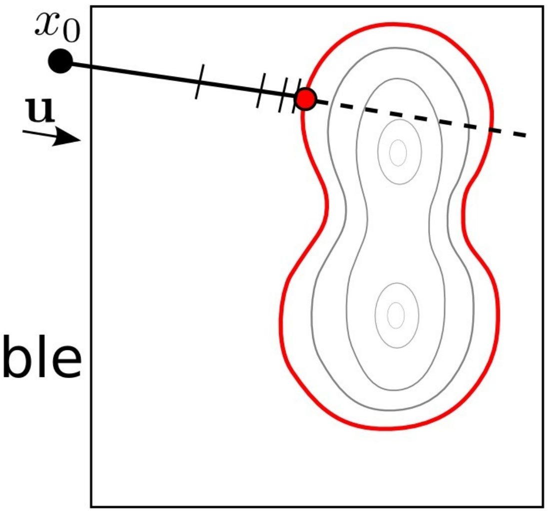




Implicit surfaces

$$S = \{(x,y,z) \in \mathbb{R}^3 | F(x,y,z) = 0\}$$





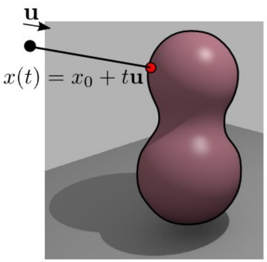
F: differentiable

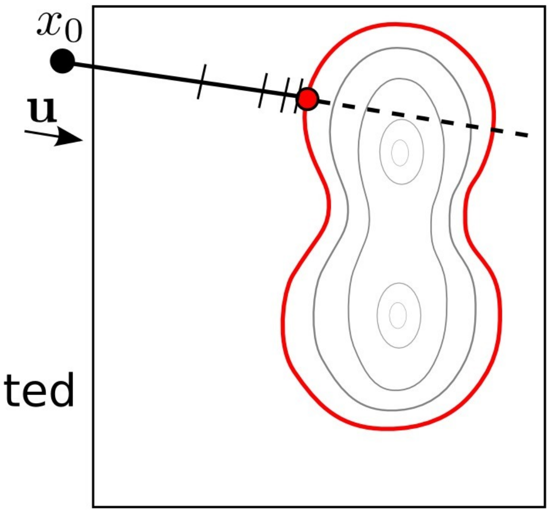
2nd Solution:

$$x^{i+1} = x^i + \frac{|F(x^i)|}{\|\nabla F\|_{\max}} \mathbf{u}$$

Implicit surfaces

$$S = \{(x,y,z) \in \mathbb{R}^3 | F(x,y,z) = 0\}$$



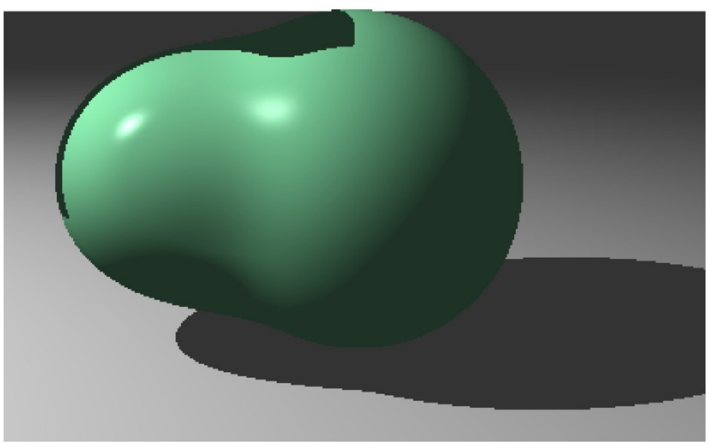


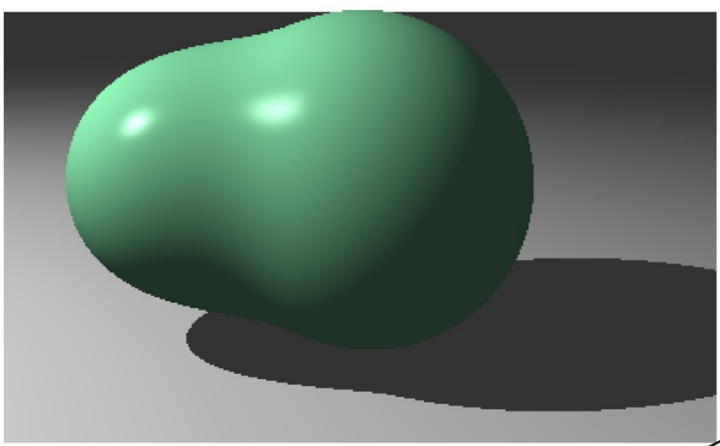
2nd Solution:

$$x^{i+1} = x^i + \frac{|F(x^i)|}{\|\nabla F\|_{\max}}\mathbf{u}$$

1

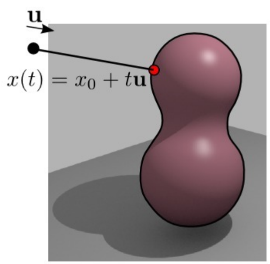
estimated 4



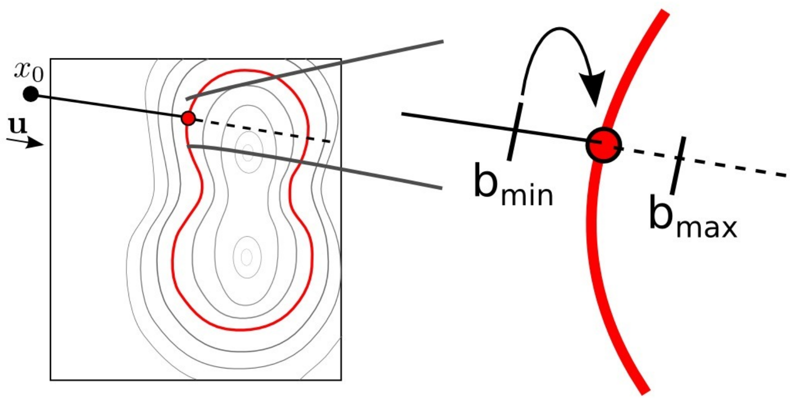


Implicit surfaces

$$S = \{(x, y, z) \in \mathbb{R}^3 | F(x, y, z) = 0\}$$



Acceleration of the convergence:



1: Binary search

2: Newton

at the end

$$x^{i+1} = x^i - \frac{F(x^i)}{<\nabla F(x^i), \mathbf{u}>}$$

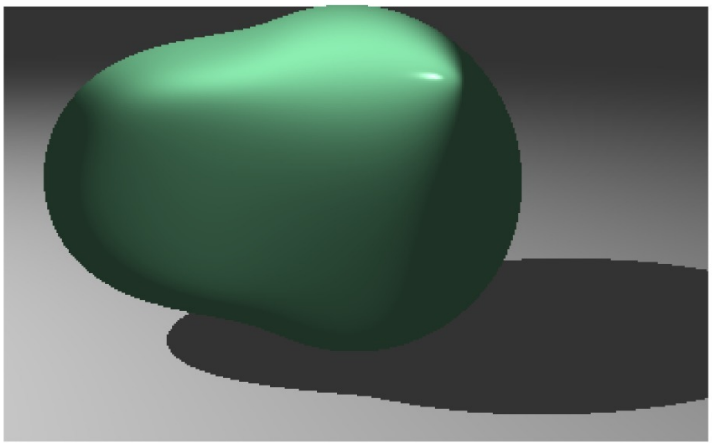
quadratic convergence

Implicit surfaces

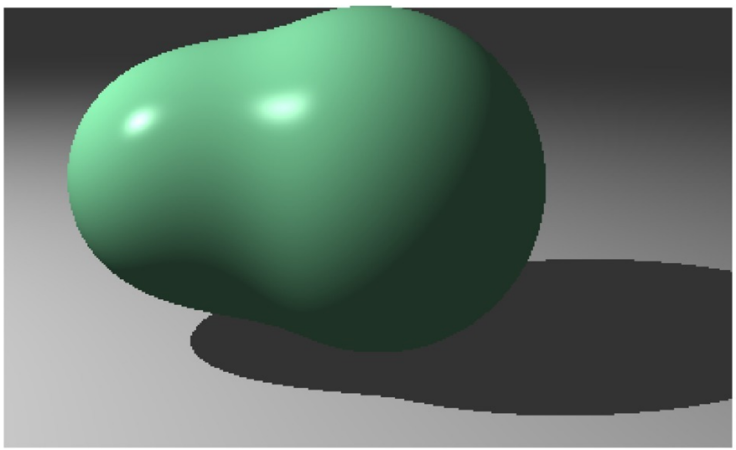
$$S = \{(x,y,z) \in \mathbb{R}^3 | F(x,y,z) = 0\}$$

The gradient is important:

$$\frac{F(x+h) - F(x)}{h}$$

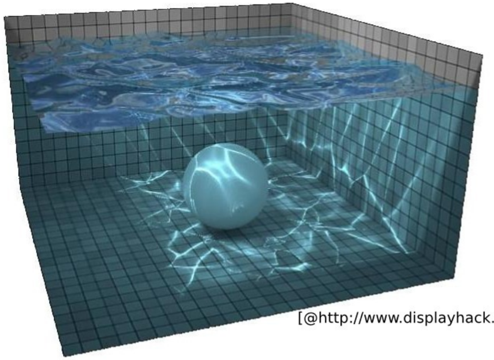


∇F analytical



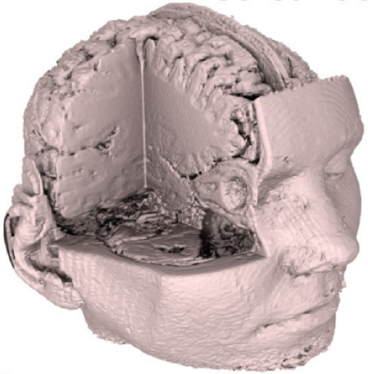
Implicit surfaces

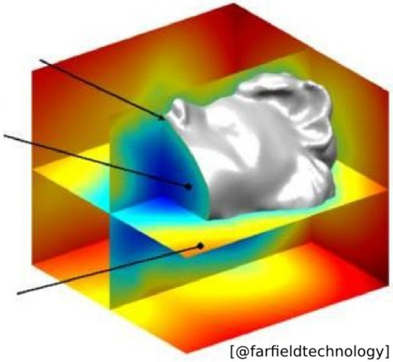
$$S = \{(x, y, z) \in \mathbb{R}^3 | F(x, y, z) = 0\}$$



[@http://www.displayhack.org/]

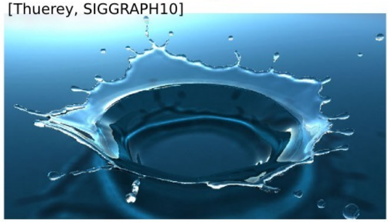
Medical data



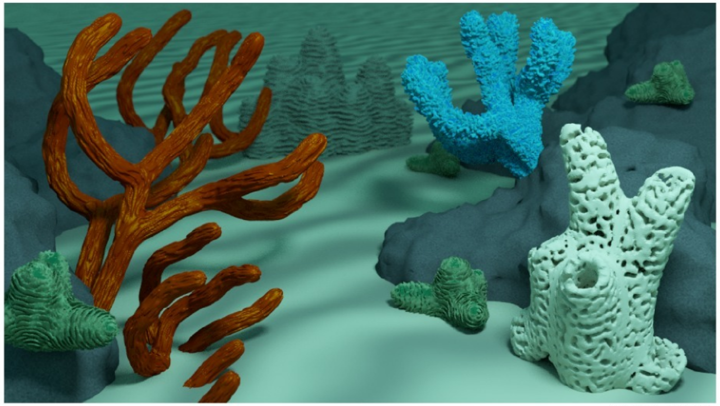


[@farfieldtechnology]

[Thuerey, SIGGRAPH10]



Fluid simulations



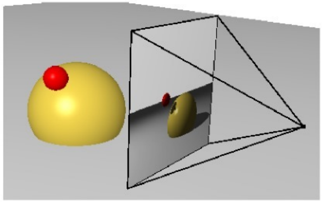
[Zanni, EG12]

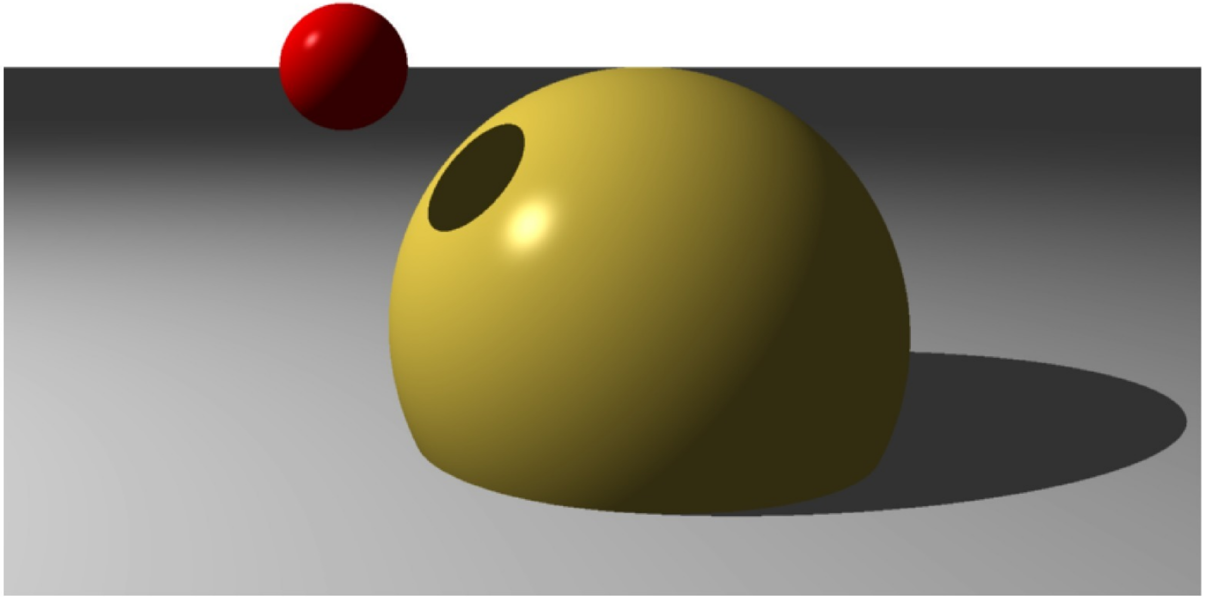
Modeling

Visual effects

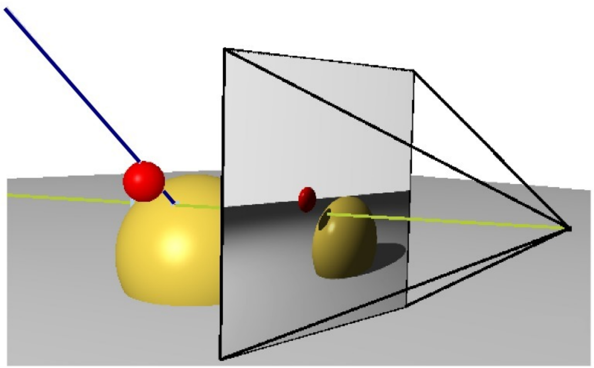
Shadows

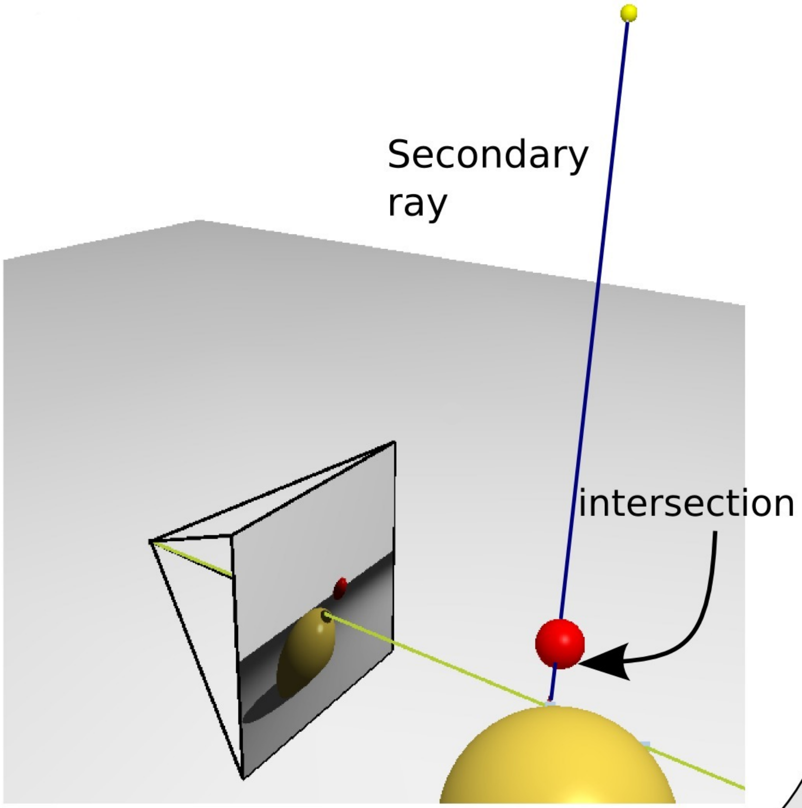
Obvious in ray-tracing



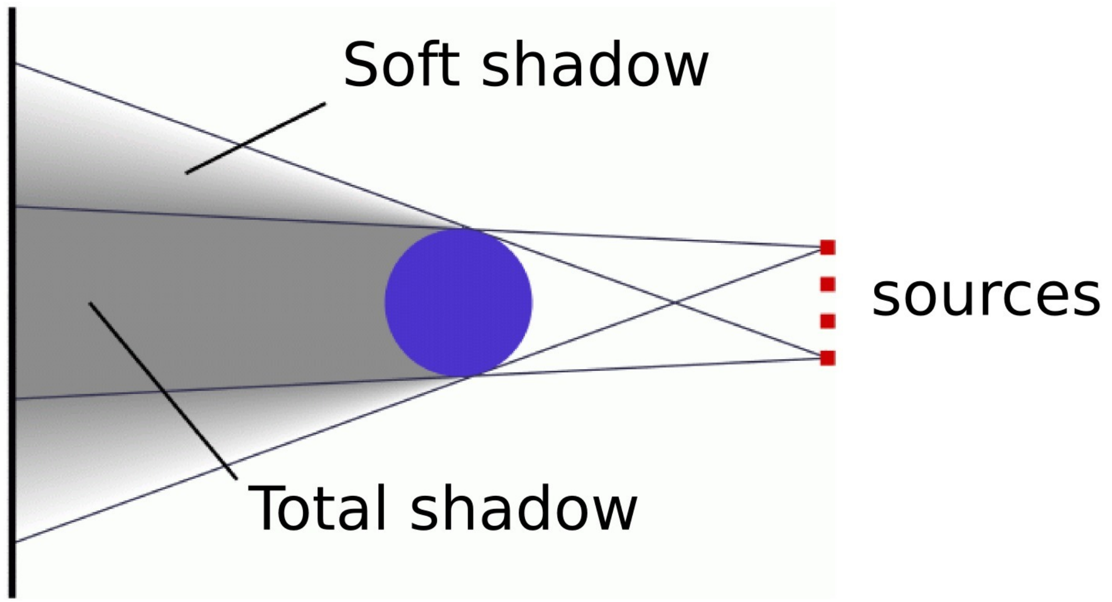


Shadows

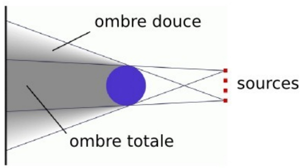


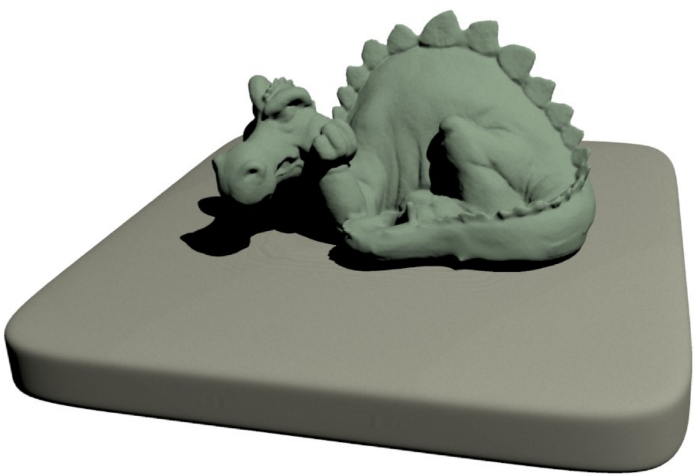


Soft shadows

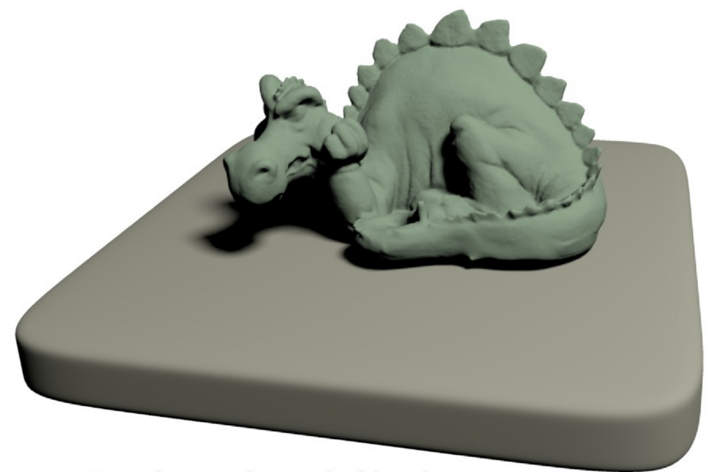


Soft shadows



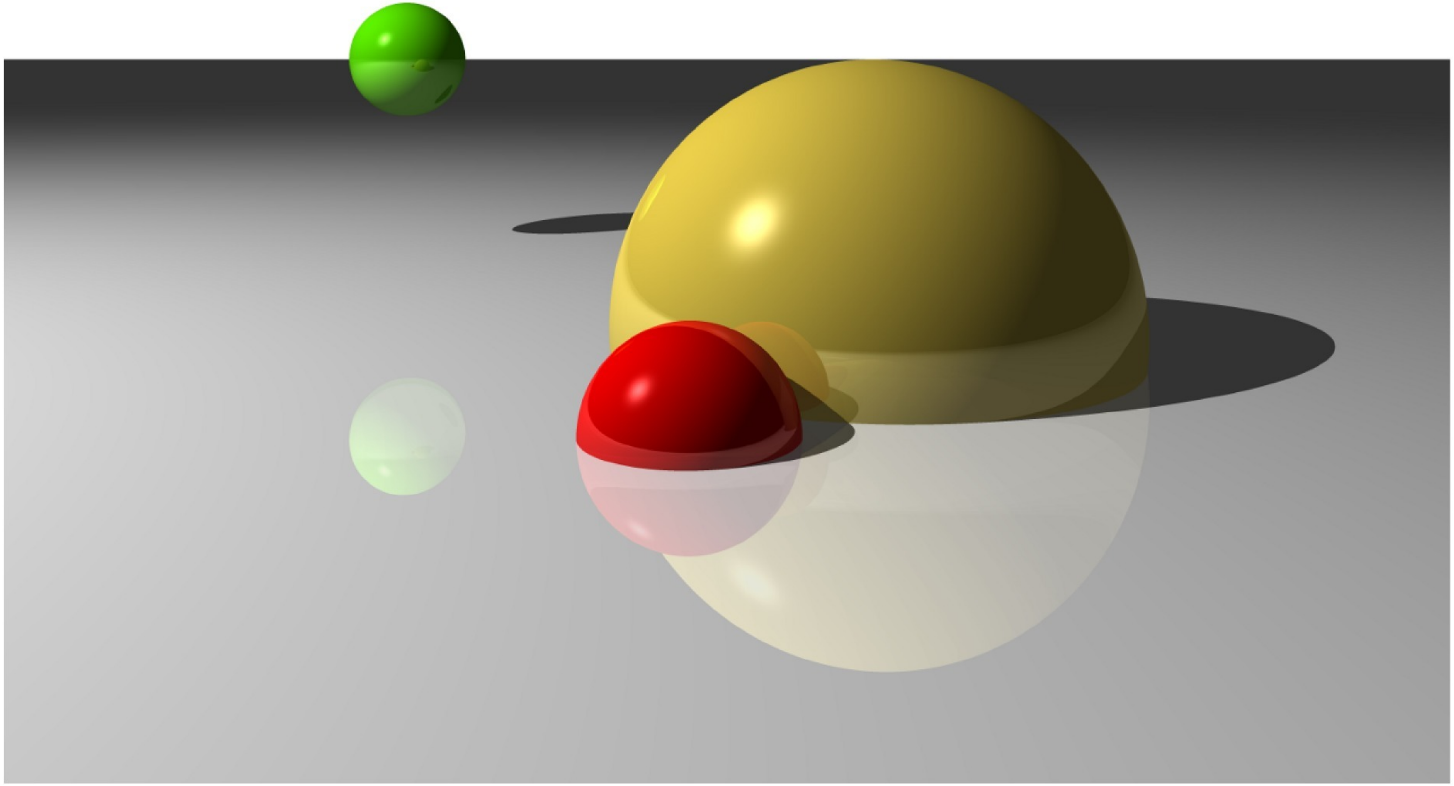


Point light source

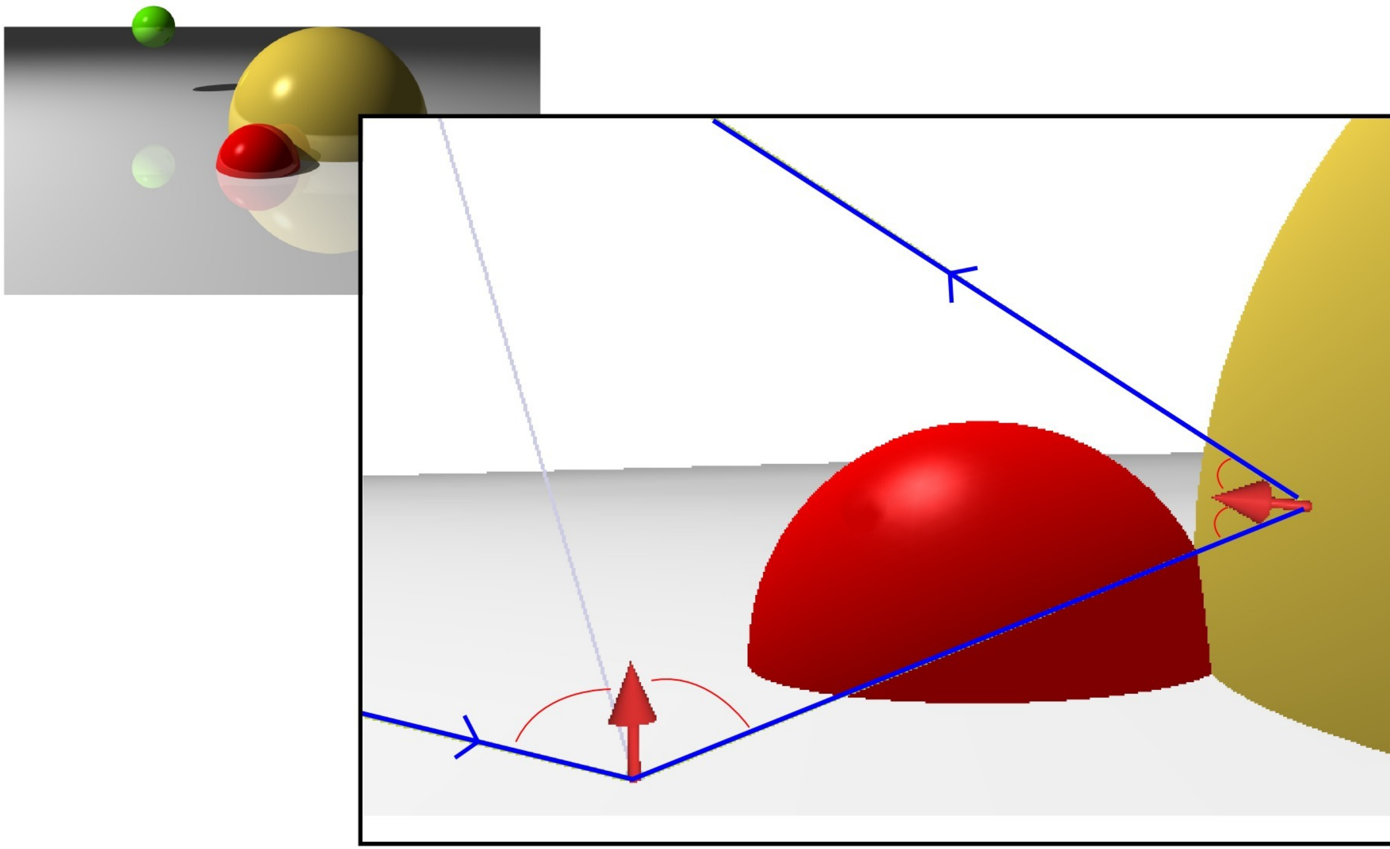


Spherical light source

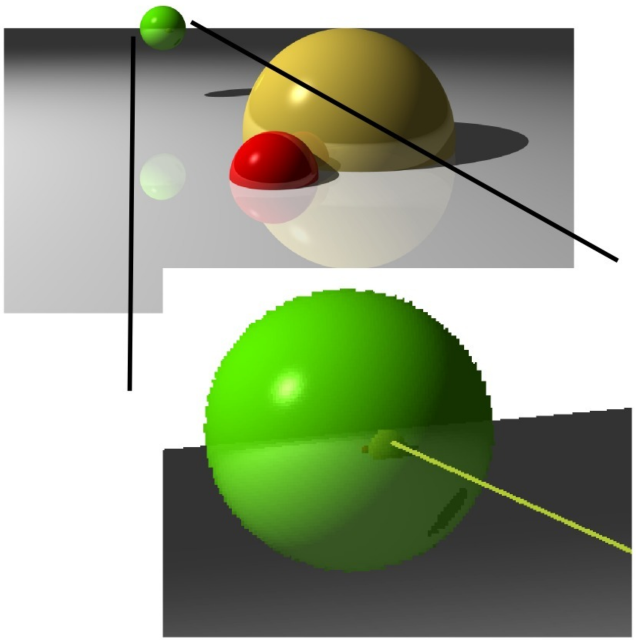
Reflections



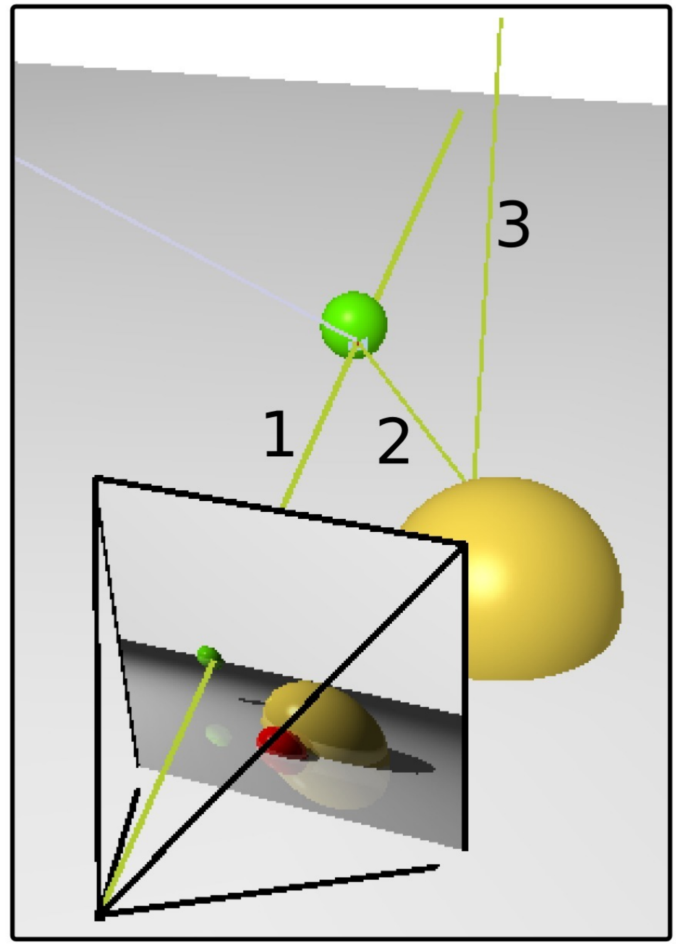
Reflections



Reflections

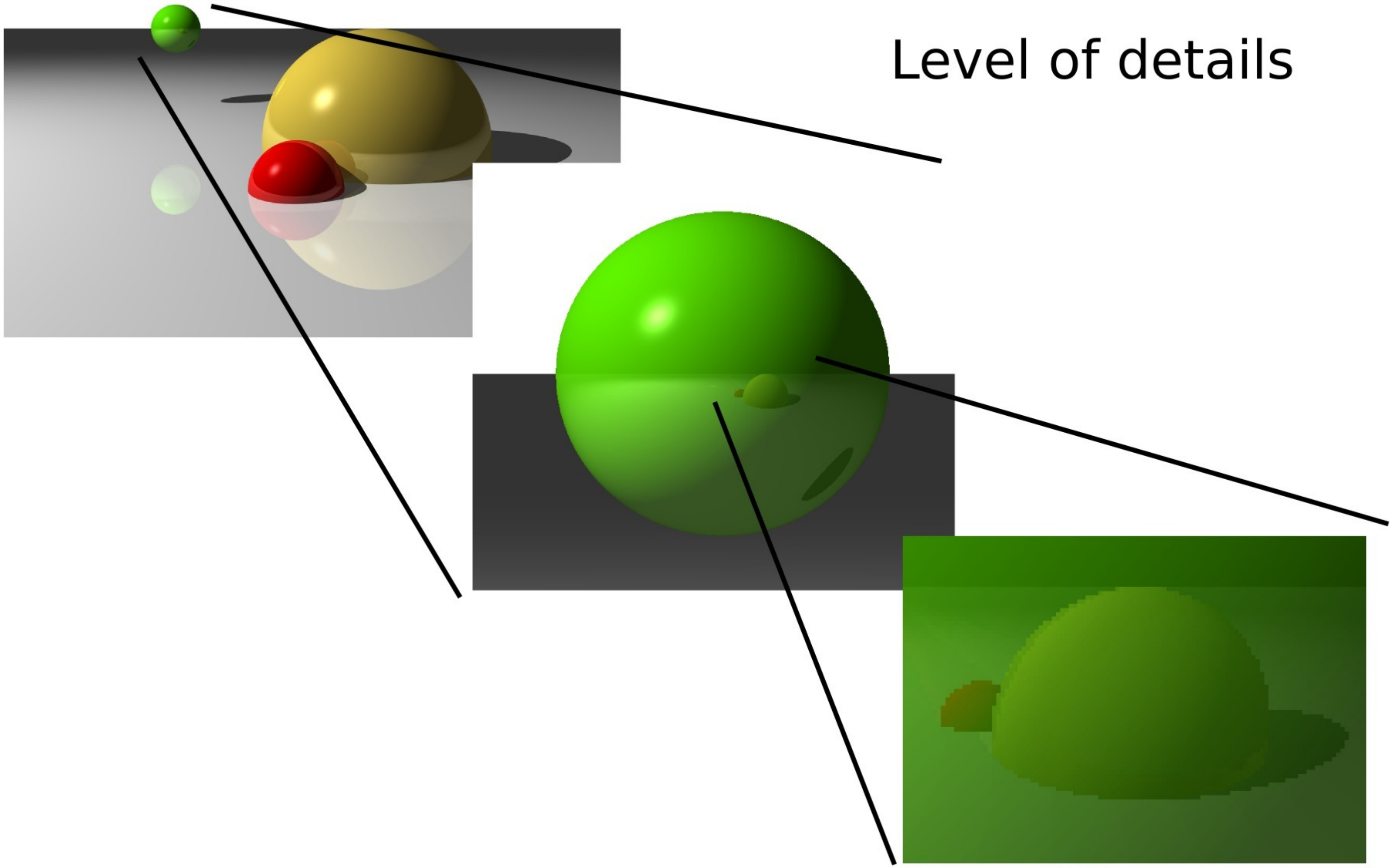


Several level of
reflections:

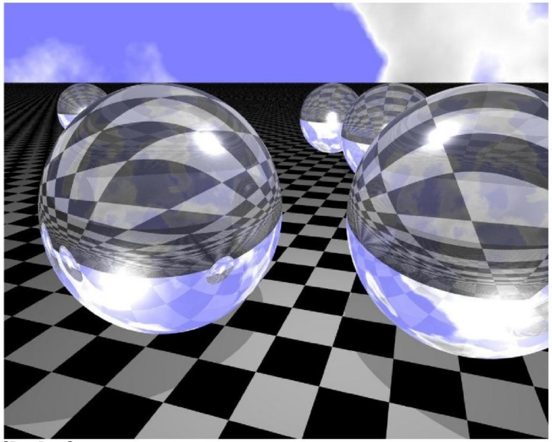


Reflections

Level of details

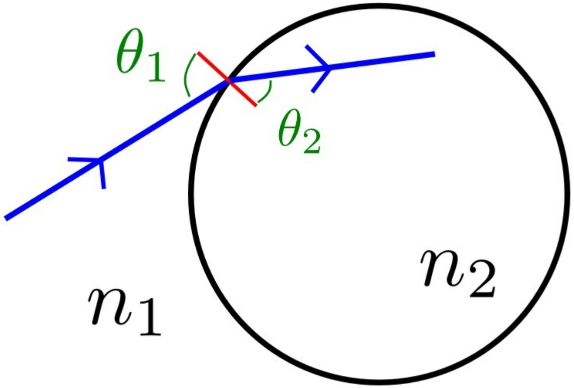


Refraction



[PovRay]

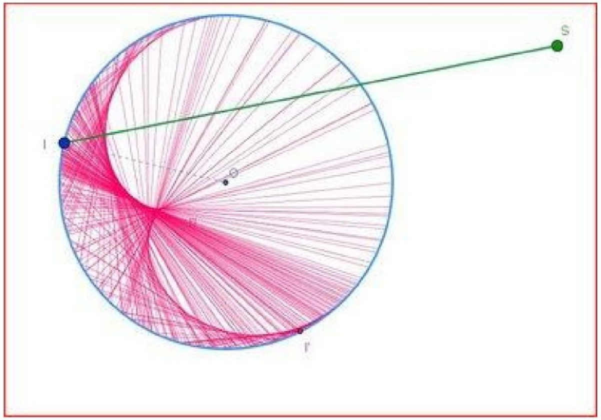
$$n_1 \sin(\theta_1) = n_2 \sin(\theta_2)$$



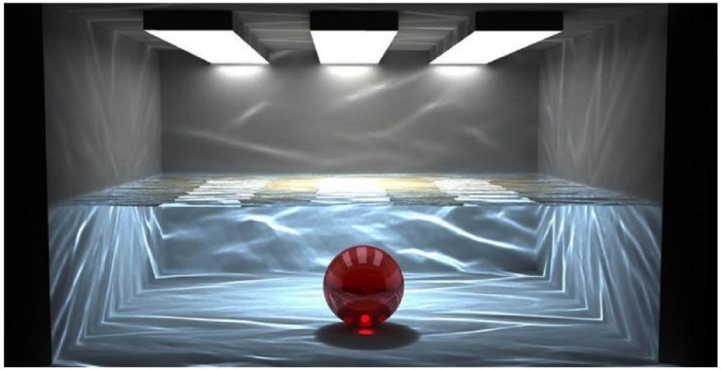
Caustics

Prefered path of the refracted/reflected rays

[http://abcmathsblog.blogspot.fr/]



Caustics

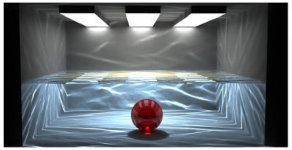


[CG Arena]

[deviantart]

Need a more advanced model of ray tracing

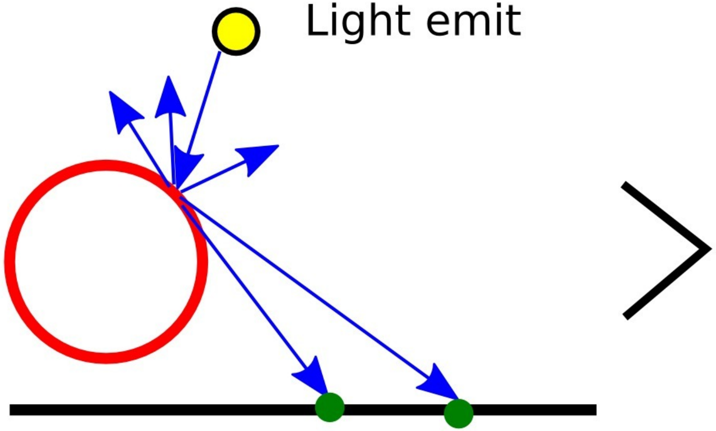
Caustics


[CG Arena]

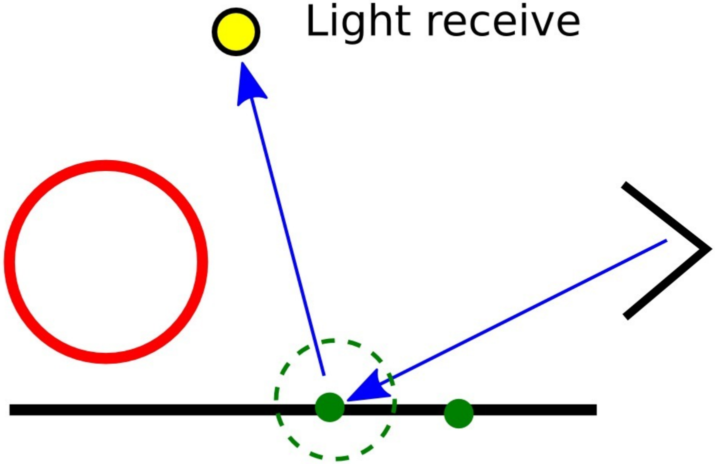

[deviantart]

Use of a photon map

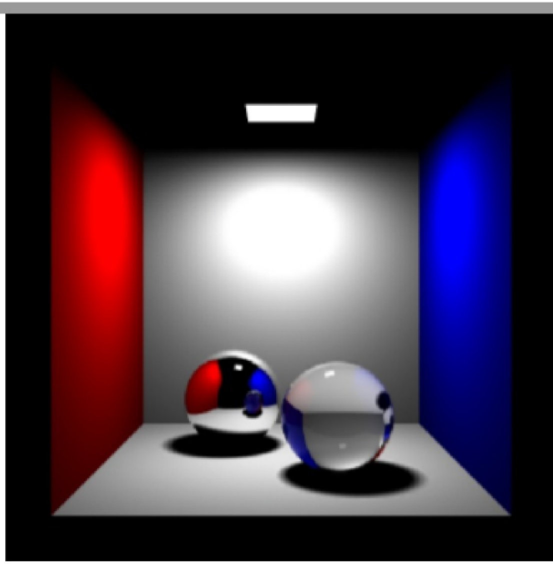
Etape 1:
Throwing *photons*

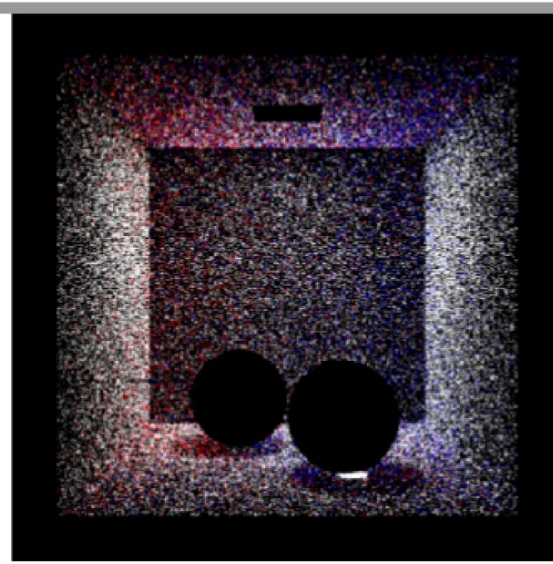


Etape 2:
Throwing rays

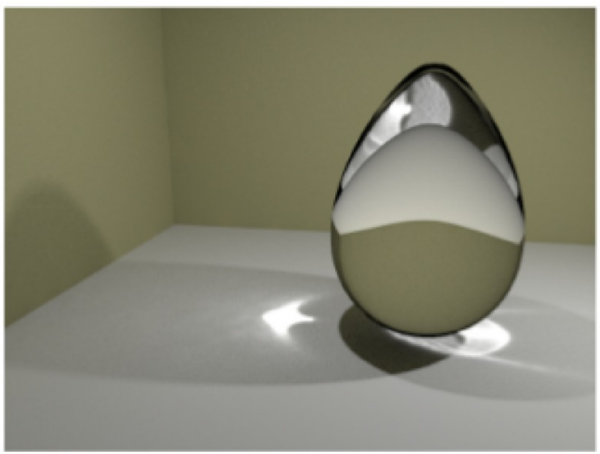


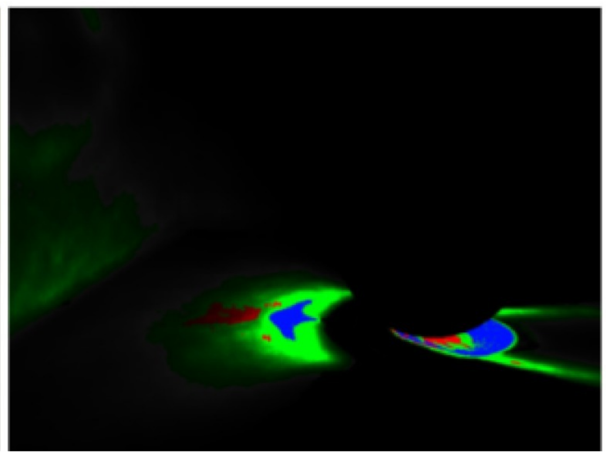
Caustics





Photon map





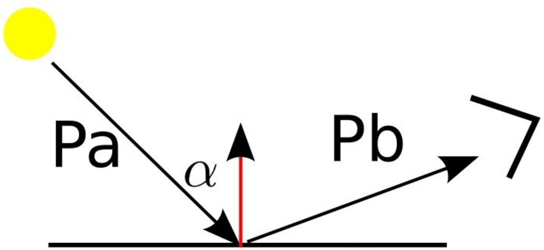
Caustics

[Jensen, EGWR96]

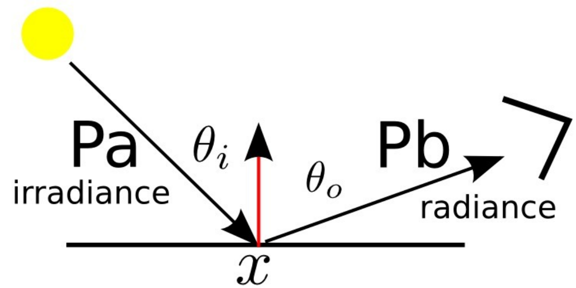
Physically based rendering

BRDF

Bidirectional
Reflectance
Distribution
Function



$$P_b = \cos(\alpha)\, P_a$$



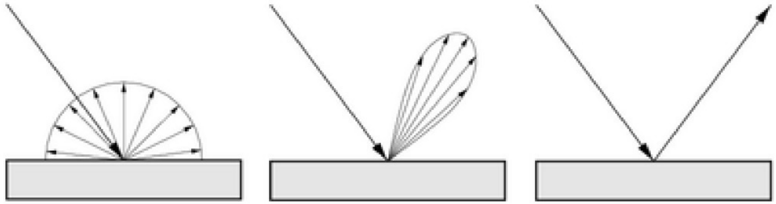
$$P_b = RBF(\theta_i, \theta_o, x)\cos(\theta_i)\, P_a$$

If depends of f => irridescence

[Wikipedia]

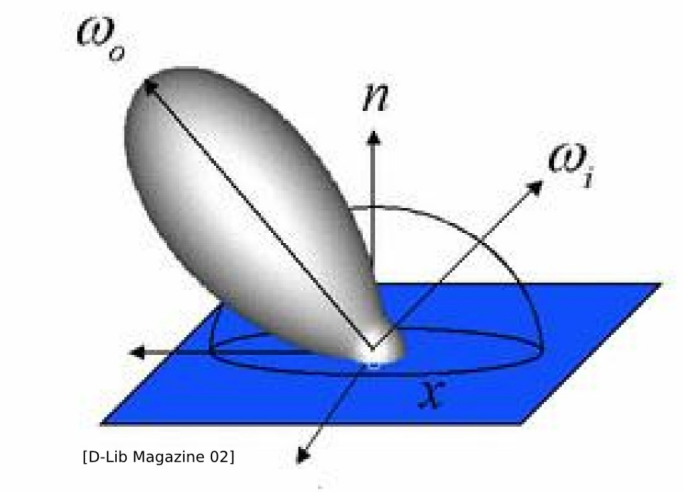
BRDF

Bidirectional **R**eflectance **D**istribution **F**unction



Perfectly diffuse

Perfectly specular



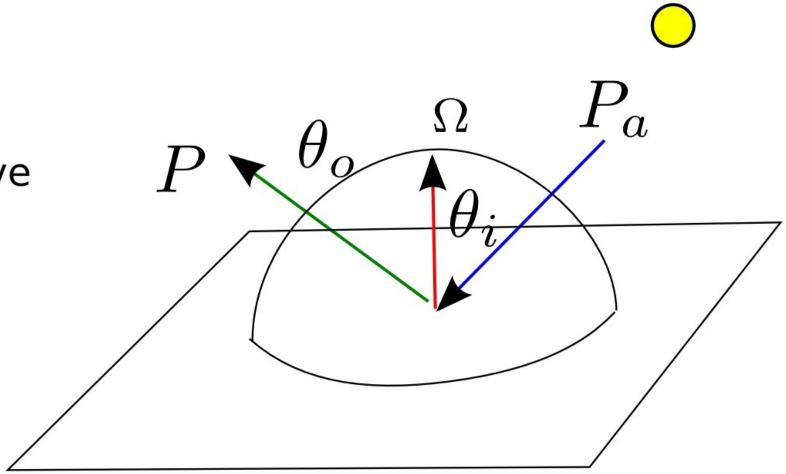
[D-Lib Magazine 02]

Rendering equation

$$P(x,\theta_o) = P_e(x,\theta_o) + \int_{\theta_i \in \Omega} f(x,\theta_i,\theta_o)\, P_a(x,\theta_i) \cos(\theta_i)\, \mathrm{d}\theta_i$$

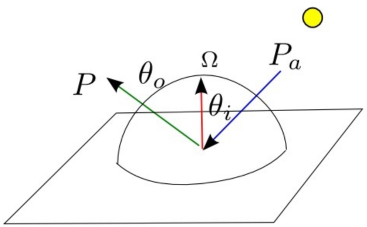
Problem: Pa depends on P

Integral equation: infinitely reccursive



Rendering equation

$$P(x,\theta_o) = P_e(x,\theta_o) + \int_{\theta_i \in \Omega} f(x,\theta_i,\theta_o)\, P_a(x,\theta_i)\cos(\theta_i)\,\mathrm{d}\theta_i$$



2 Approches:

1- Finite element discretization

Radiosity

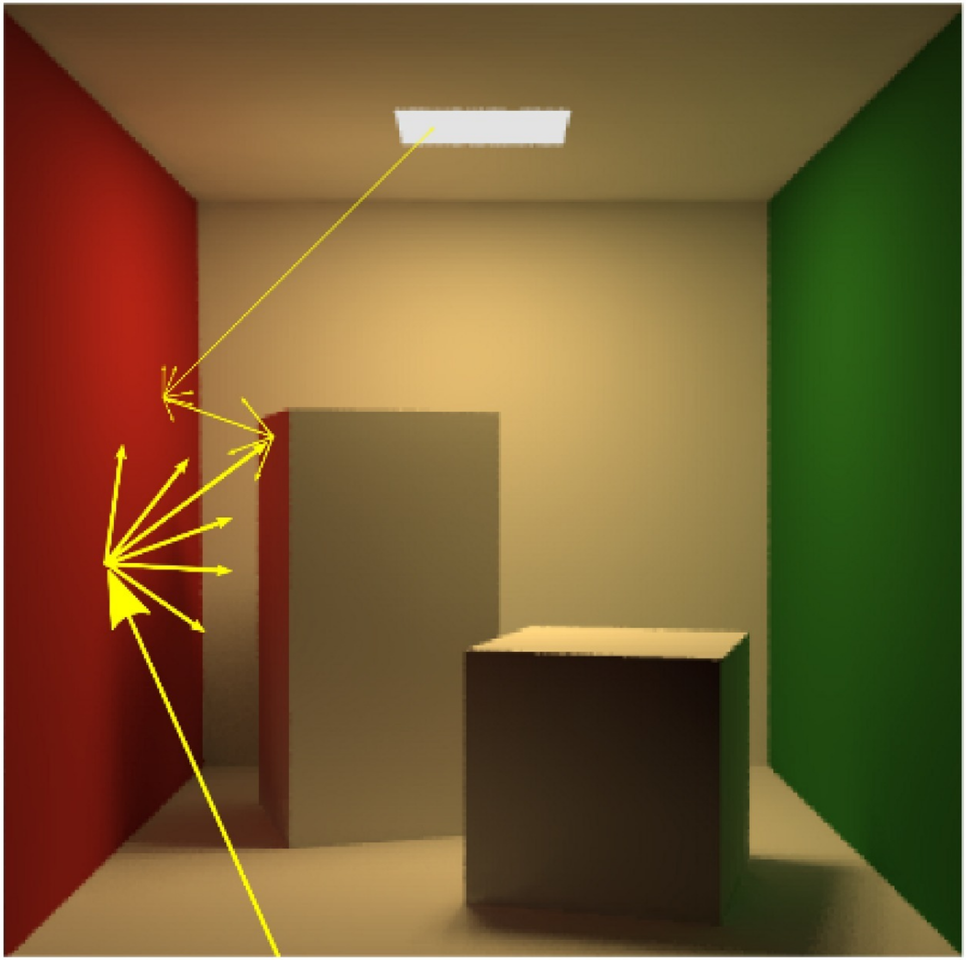
2- Monte-Carlo

Path Tracing

Metropolis Light Transport (MTL)

Path tracing

We randomly sample θ_i



Path tracing

Modeling secondary light sources:

Direct illumination

Path tracing