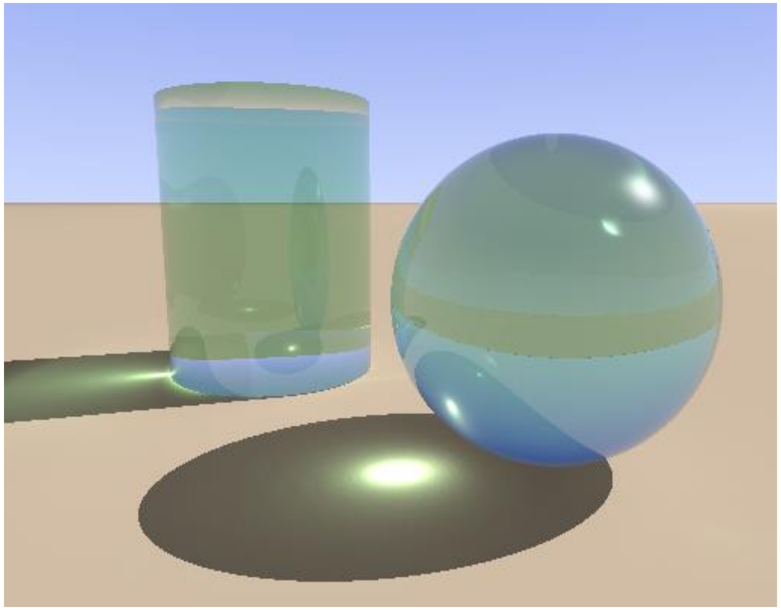
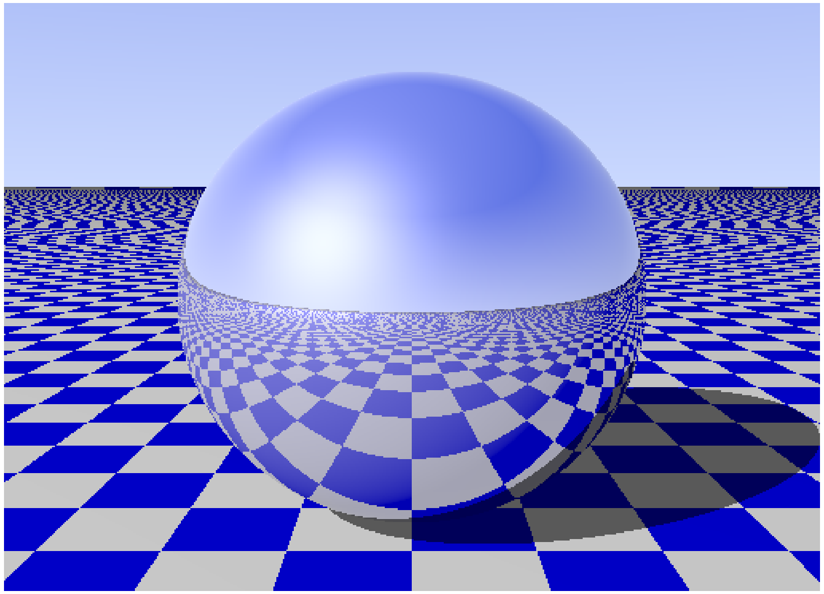
Ray Tracing

General introduction

- Ray tracing is a technique for producing photo-realistic images by tracing light paths through pixels in an image plane
- Produces high quality images
- Usually slower





Overview

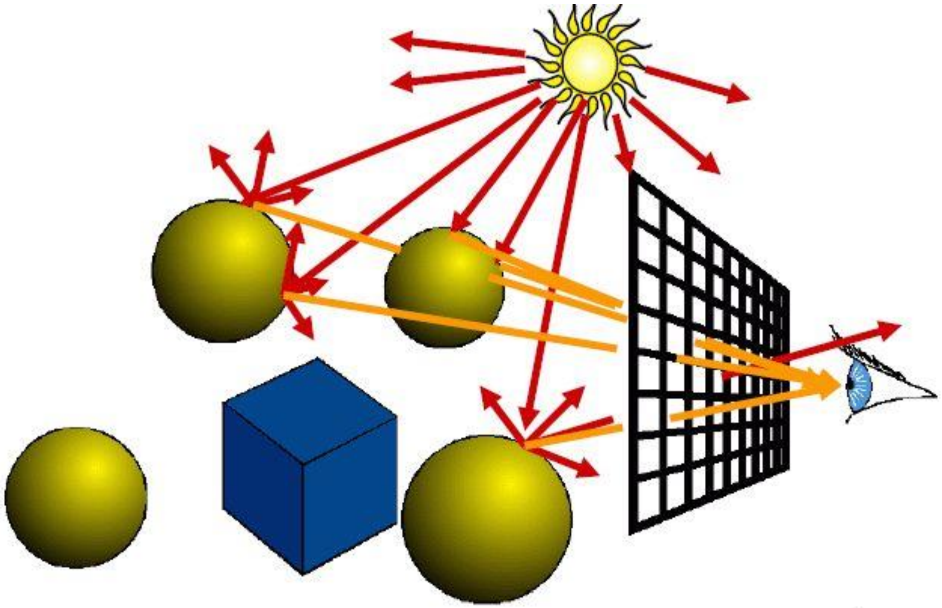
- Ray casting:
  - Ray from eye through pixel
  - Ray-scene intersection
  - Simple illumination model
- Ray tracing:
  - More complex illumination model
  - Recursive definition
  - After hitting an object in scene, ray can continue to travel:
    - By reflection on a shiny surface
    - By refraction in a transparent surface
    - Form a shadow if ray to light is intersecting an object

Ray tracing

Ray-tracing: compute an image of a scene by simulating rays of light in the real world.

Physically: rays of light are emitted from a light source and illuminate objects. The light reflects off of the objects or passes through transparent objects, until it hits our eyes or a camera lens.

Majority of rays never hit an observer, so it is computationally inefficient.



Backwards ray tracing

Ray-tracing programs start from the simulated camera and trace rays **backwards** out into the scene.
The user specifies: camera location, light sources, objects with their properties, and any atmospheric media such as fog, haze, or fire.

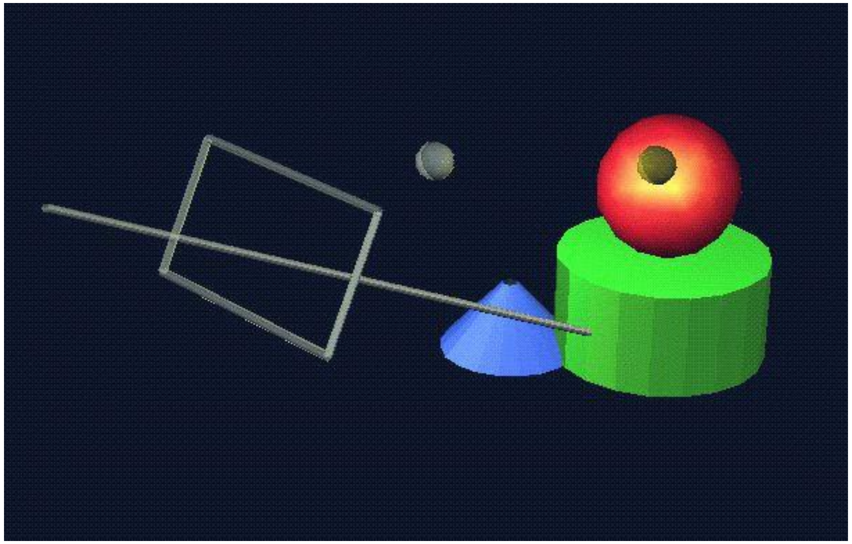
For every pixel in the final image one or more viewing rays are shot from the camera, into the scene to see if it intersects with any of the objects in the scene. These "viewing rays" originate from the viewer, represented by the camera, and pass through the viewing window (representing the final image).

Ray casting

Ray Casting (Pinhole camera model):

For every pixel:

- Determine the ray from eye to pixel.
- Compare ray with every object.
- Find nearest intersection.
- Compute color of pixel.



Ray equation

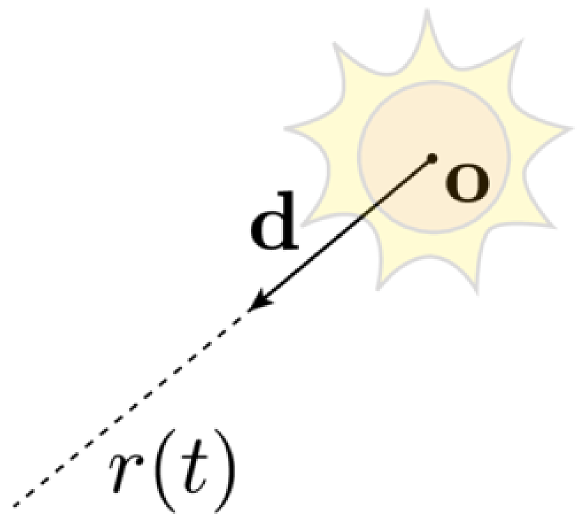
origin

unit direction

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

point along ray

"time"



Simplified ray tracer

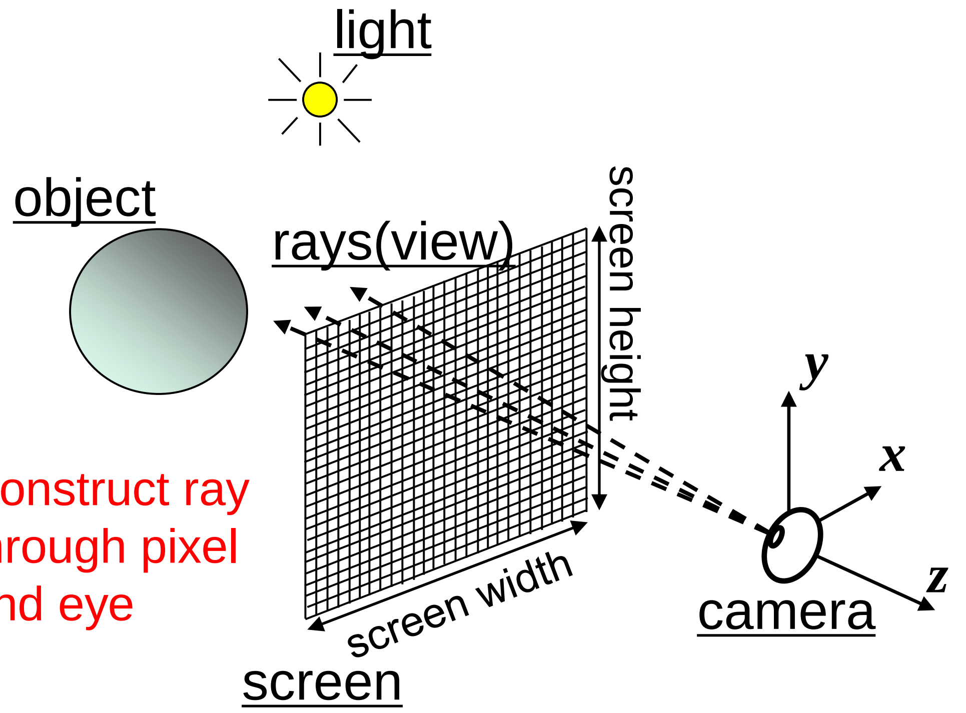
```
ColorRGB pixels[numPixWidth][numPixHeight];

Function Render() {
  for j=0 to numPixHeight - 1 do
    for i=0 to numPixWidth - 1 do {
      pixels[i][j] = GetColor(i, j);
    }
}

Camera camera;
LightList lightList;
ObjectList objectList;

Function GetColor(i, j) {
   ray.origin = camera.position
   ray.direction = [x(i), y(j), 0]
   ray.direction -= camera.position
   ray.direction.normalize()
   return RayTrace(ray, 0)
}
```

Construct ray through pixel and eye



Simplified ray tracer

```
//find closest intersection and calculate shading
Function RayTrace(ray, traceDepth) {
  intersectedObject = QueryScene(ray);
  if (intersectedObject == NIL)
    return backgroundColor;

  intersectedPosition = ray.origin + [distance to intersection]*ray.direction;
  return TotalShading(intersectedObject, ray, intersectedPosition);
}
```

```
//Search for the closest object
//which intersects with ray
Function QueryScene(ray) {
 ...introduced later
}
```

```
//calculate shading contribution from all lights
Function TotalShading(object, ray, intersect) {
  intensity = object.AmbientIntensity;
  foreach light in lightList do
    intensity += DiffuseShading(object, light, intersect) +
                 SpecularShading(object, light, intersect, ray);
  return intensity;
}
```

Using Phong (or Blinn-Phong) illumination model

Computation of ray direction

- Camera location is known
- Assume projection screen to be perpendicular to z axis, at the position z=0
- Final image of size W x H pixels
- Suppose projection screen dimension to be $[-1,1] \times [-1,1]$. Map pixels from $\{0, \ldots, W-1\} \times \{0, \ldots, H-1\}$ to it:

$$x(i) = 2 * \frac{i - (W-1)/2}{W-1}$$

$$y(j) = 2 * \frac{j - (H-1)/2}{H-1}$$

Computation of ray direction

If we have a larger scene, where (for example):

$$-s \leq x \leq s$$
$$-t \leq y \leq t$$

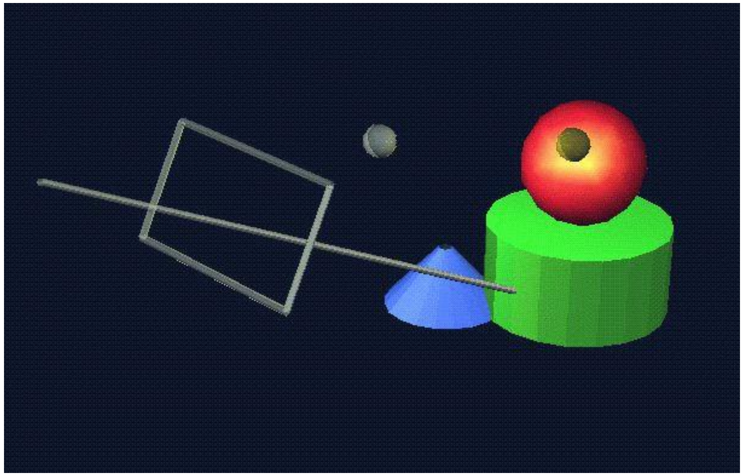
then we can simply rescale with:

x(i) $\leftarrow$ x(i) * s

y(j) $\leftarrow$ y(j) * t

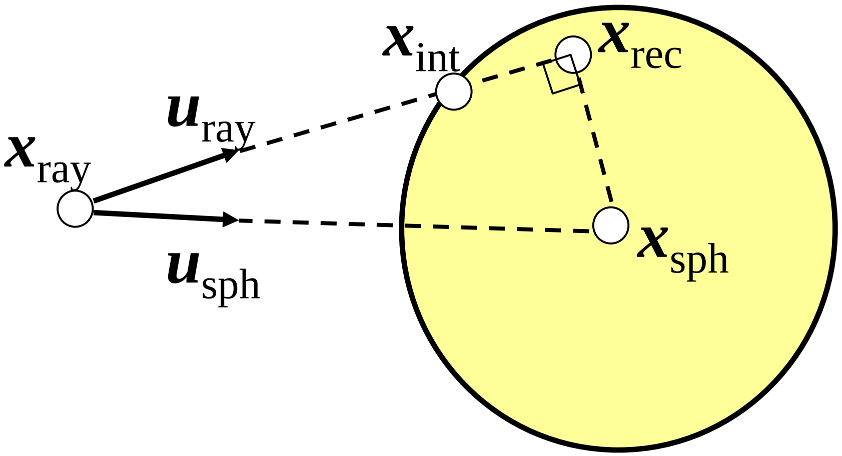
Simplified ray tracer

```
//Search for the closest object which
//intersects with ray
Function QueryScene(ray) {
  closestObj=NIL;
  distance=INFINITY;
  foreach object in sceneList do {
    if intersect( ray, object)
      if (object.distance<distance) {
        closestObj=object;
        distance=object.distance;
      }
  return closestObj;
}
```



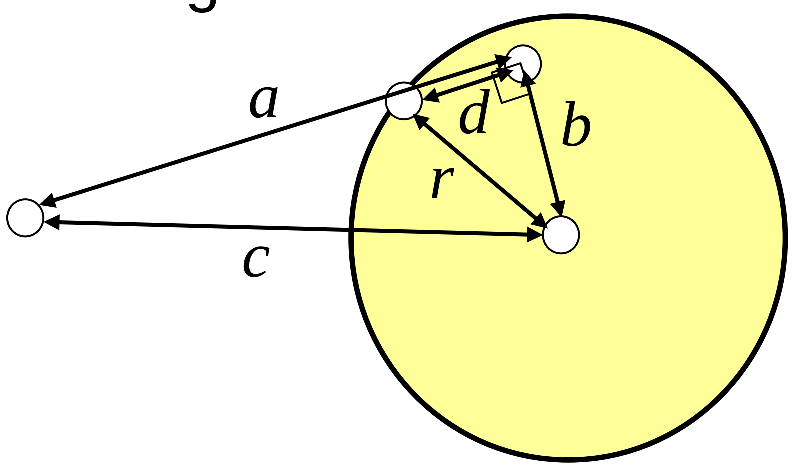
Ray-sphere intersection: Method 1

- Vectors/Points



$\boldsymbol{x}_{\text{ray}}$: starting position of ray
$\boldsymbol{x}_{\text{sph}}$: position of sphere
$\boldsymbol{x}_{\text{int}}$: position of intersection
$\boldsymbol{x}_{\text{rec}}$: position of third vertex
to form the right triangle
$(x_{ray}, x_{sph}, x_{rec})$
$\boldsymbol{u}_{\text{ray}}$: unit vector of ray direction
$\boldsymbol{u}_{\text{sph}}$: unit vector from ray position
to sphere position

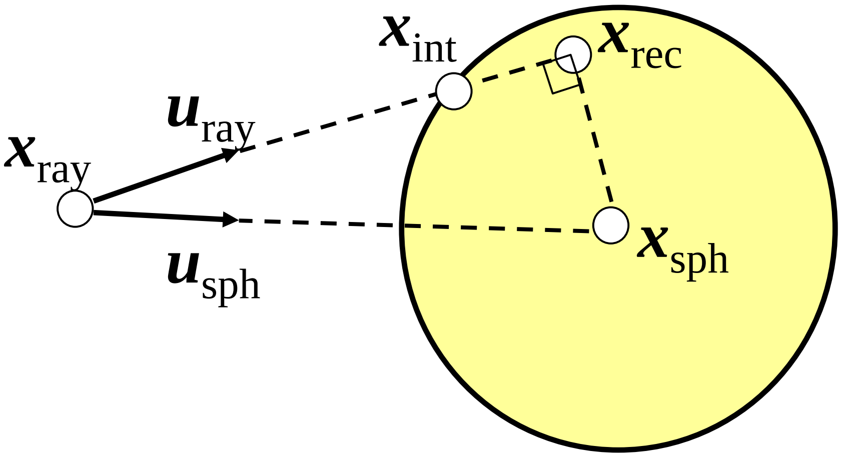
- Lengths



r : radius of sphere (known)
$c = |\boldsymbol{x}_{\text{ray}} - \boldsymbol{x}_{\text{sph}}|$ (known)
$a = |\boldsymbol{x}_{\text{ray}} - \boldsymbol{x}_{\text{rec}}|$ (unknown)
$b = |\boldsymbol{x}_{\text{sph}} - \boldsymbol{x}_{\text{rec}}|$ (unknown)
$d = |\boldsymbol{x}_{\text{int}} - \boldsymbol{x}_{\text{rec}}|$ (unknown)

Ray-sphere intersection: Method 1

- Vectors



From relations: $\mathbf{u}_{ray} \cdot \mathbf{u}_{sph} = \cos\theta$

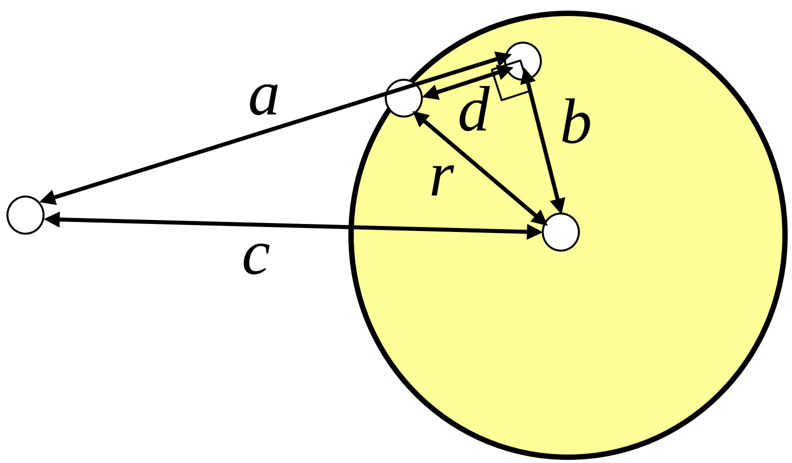
$$\cos\theta = a / c$$

$$a = c \times \mathbf{u}_{ray} \cdot \mathbf{u}_{sph} = \mathbf{u}_{ray} \cdot \left(x_{sph} - x_{ray}\right)$$

We have $b^2 = c^2 - a^2$

There is an intersection if $r^2 - b^2 \geq 0$

- Lengths



Position of intersection is

$$\boldsymbol{x}_{\text{int}} = \boldsymbol{x}_{\text{ray}} + (a - d)\boldsymbol{u}_{\text{ray}}$$

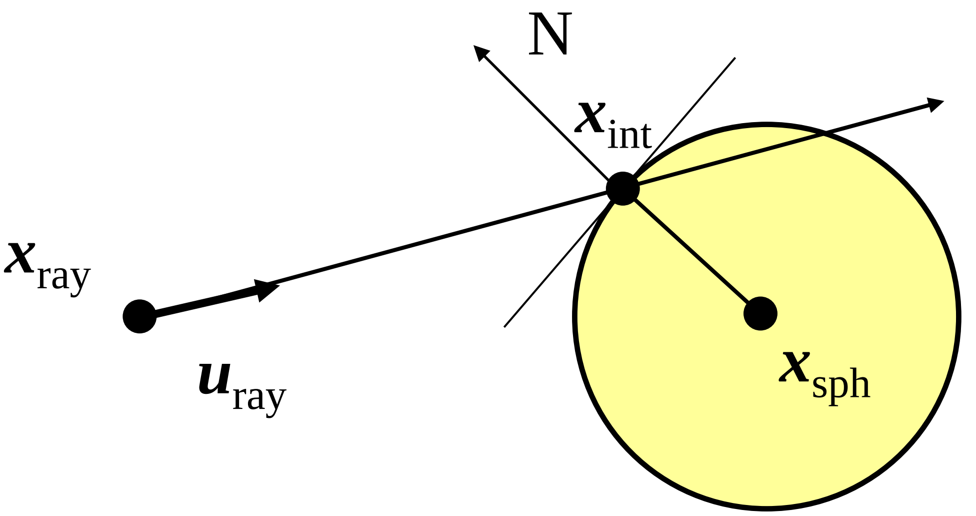
with $d = \pm\sqrt{r^2 - b^2}$

Positive smaller $(a - d)$ corresponds to distance to ray position to intersection.

Ray-sphere intersection: Method 1

For illumination, we need to compute the normal vector at the intersection point (p):

$$N = \frac{x_{int} - x_{sph}}{\left\| x_{int} - x_{sph} \right\|}$$



Ray-sphere intersection: Method 2

Example of the unit sphere

$$f(\boldsymbol{x}) = |\boldsymbol{x}|^2 - 1$$

Using the fact that **x** is on the ray **r**(t)

$$f(\boldsymbol{r}(t)) = |\boldsymbol{o} + t\boldsymbol{d}|^2 - 1$$

An intersection occurs when

$$|\boldsymbol{d}|^2 t^2 + 2(\boldsymbol{o} \cdot \boldsymbol{d})t + |\boldsymbol{o}|^2 - 1 = 0$$

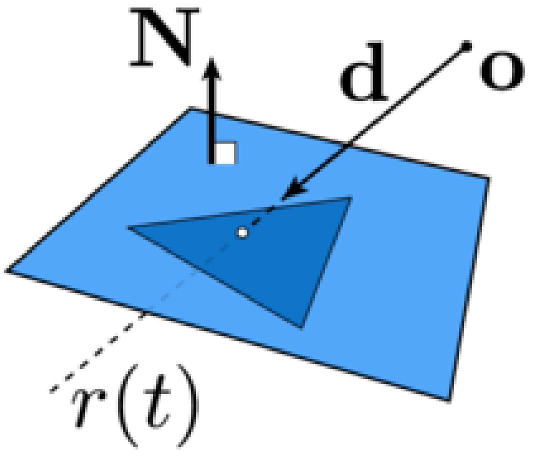
Solving for t (keeping the smallest root)

$$t = -\boldsymbol{o} \cdot \boldsymbol{d} - \sqrt{(\boldsymbol{o} \cdot \boldsymbol{d})^2 - |\boldsymbol{o}|^2 + 1}$$

Only if $t > 0$

Ray-plane intersection

Equation of plane: $\boldsymbol{N}^T \boldsymbol{x} = c$
N: Unit normal to the plane
c: offset



Using the fact that **x** is on the ray

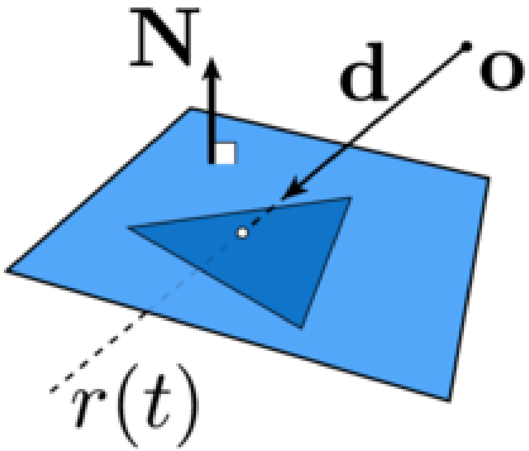
$$\boldsymbol{N}^T \boldsymbol{r}(t) = c$$
$$\boldsymbol{N}^T(\boldsymbol{o} + t\boldsymbol{d}) = c$$

Solving for t

$$t = \frac{c - \boldsymbol{N}^T \boldsymbol{o}}{\boldsymbol{N}^T \boldsymbol{d}}$$

Ray-triangle intersection

If the ray intersect a given plane, use barycentric coordinates to determine if the intersection point is within the triangle or not

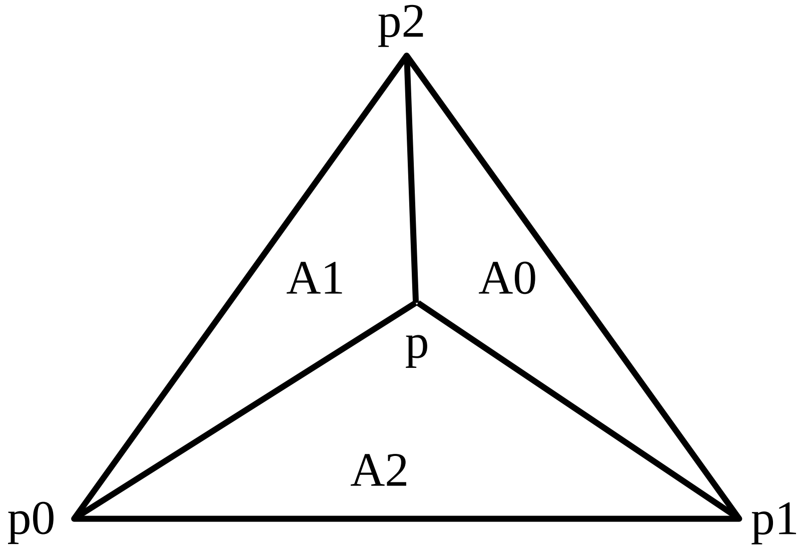


Point in triangle?

Let **N** be the normal to the triangle.

p is inside the triangle <p0,p1,p2>, if:
$((p_1 - p_0) \times (p - p_0)).N$
$((p_1 - p) \times (p_2 - p)).N$
$((p - p_0) \times (p_2 - p_0)).N$
all have the same sign

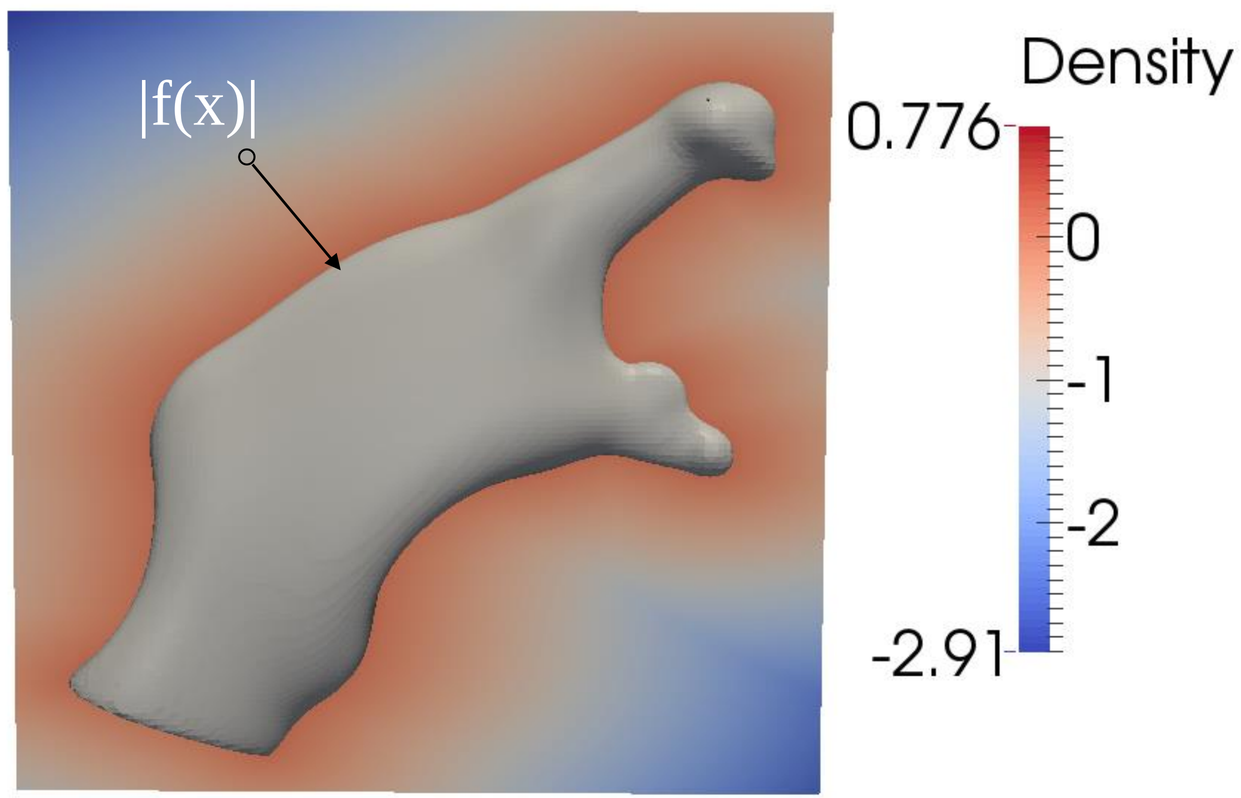
The normal to the triangle is given by:
$(p_1 - p_0) \times (p_2 - p_0)$
The unit normal is obtained by normalizing this vector.



Signed Distance Functions

SDF: Signed Distance Function

If f is an SDF, then f(**x**) gives the (signed) distance from **x** to the surface f=0



Lipschitz functions

Lipschitz function:

f is Lipschitz iff $\exists\ \lambda > 0, \forall\ \boldsymbol{x}, \boldsymbol{y} \in D$

$$|f(\boldsymbol{x}) - f(\boldsymbol{y})| \leq \lambda \|\boldsymbol{x} - \boldsymbol{y}\|$$

If f is Lipschitz with constant λ, then f/λ is a signed distance bound of its implicit surface.

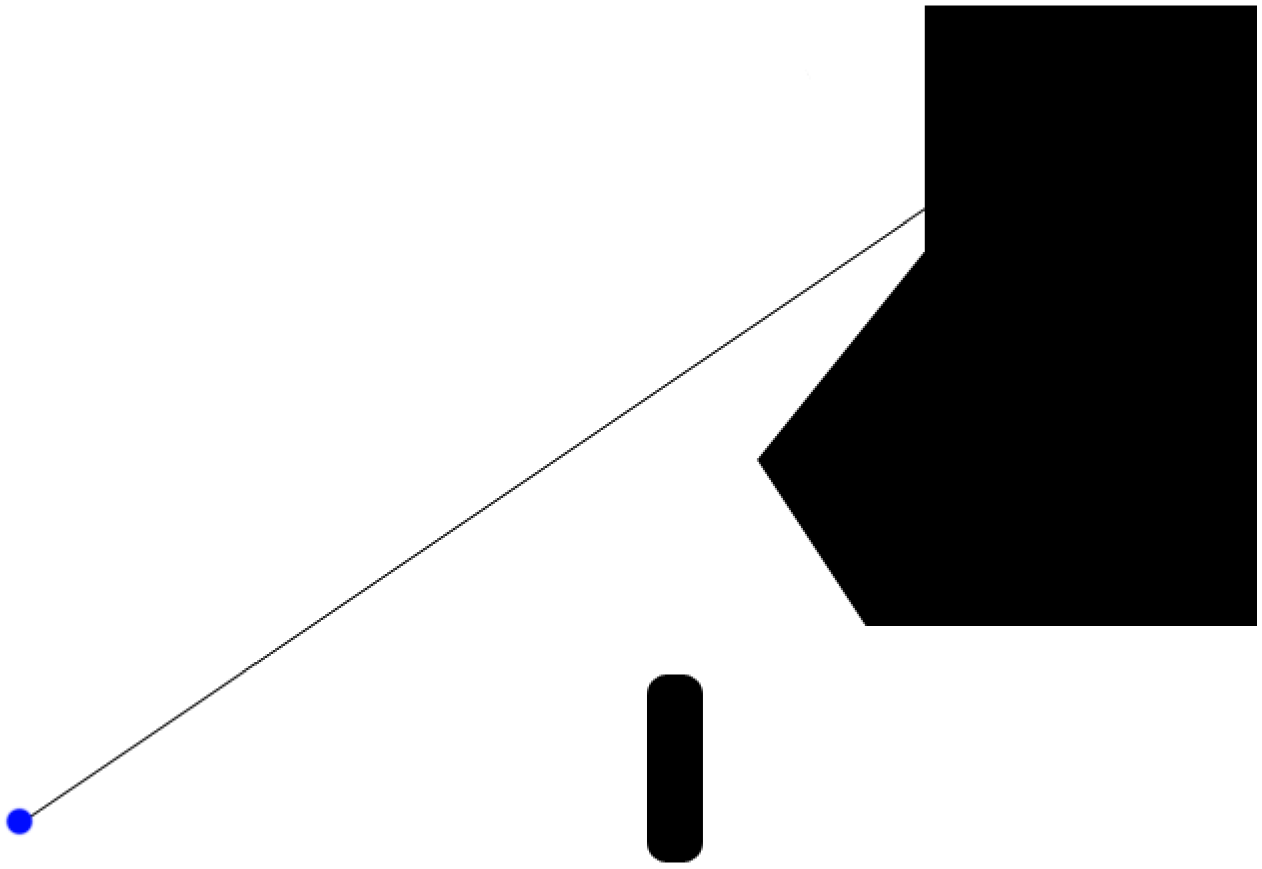
It is “easier” to find Lipschitz functions than to find analytical expressions of distance functions.

Intersection with SDF

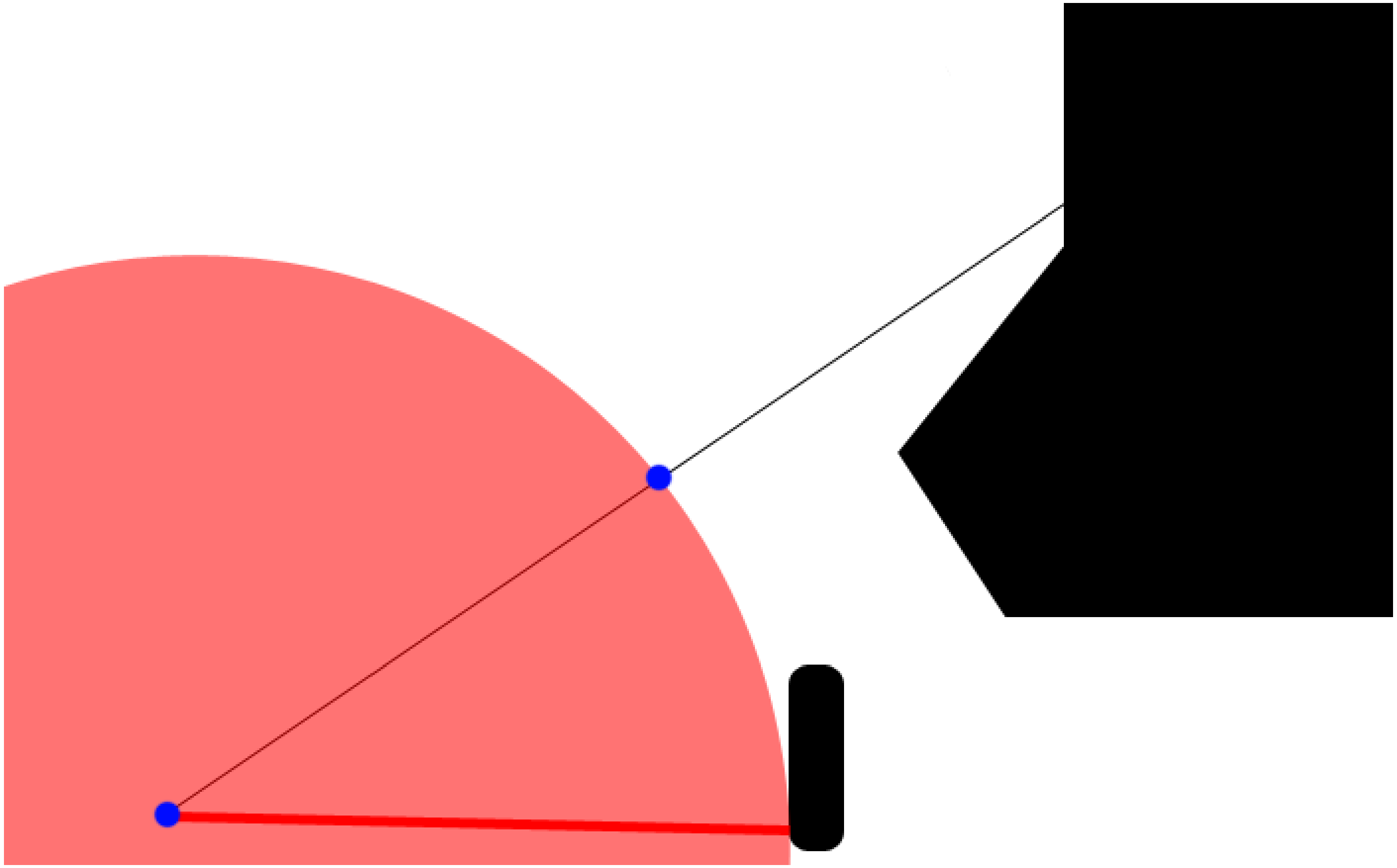
Ray-Casting SDF:

Initialize t = 0
While t < D:
    Let d = abs(f(r(t)))
    If $d < \varepsilon$, then return t // intersection
    t = t + d
Return NIL // no intersection

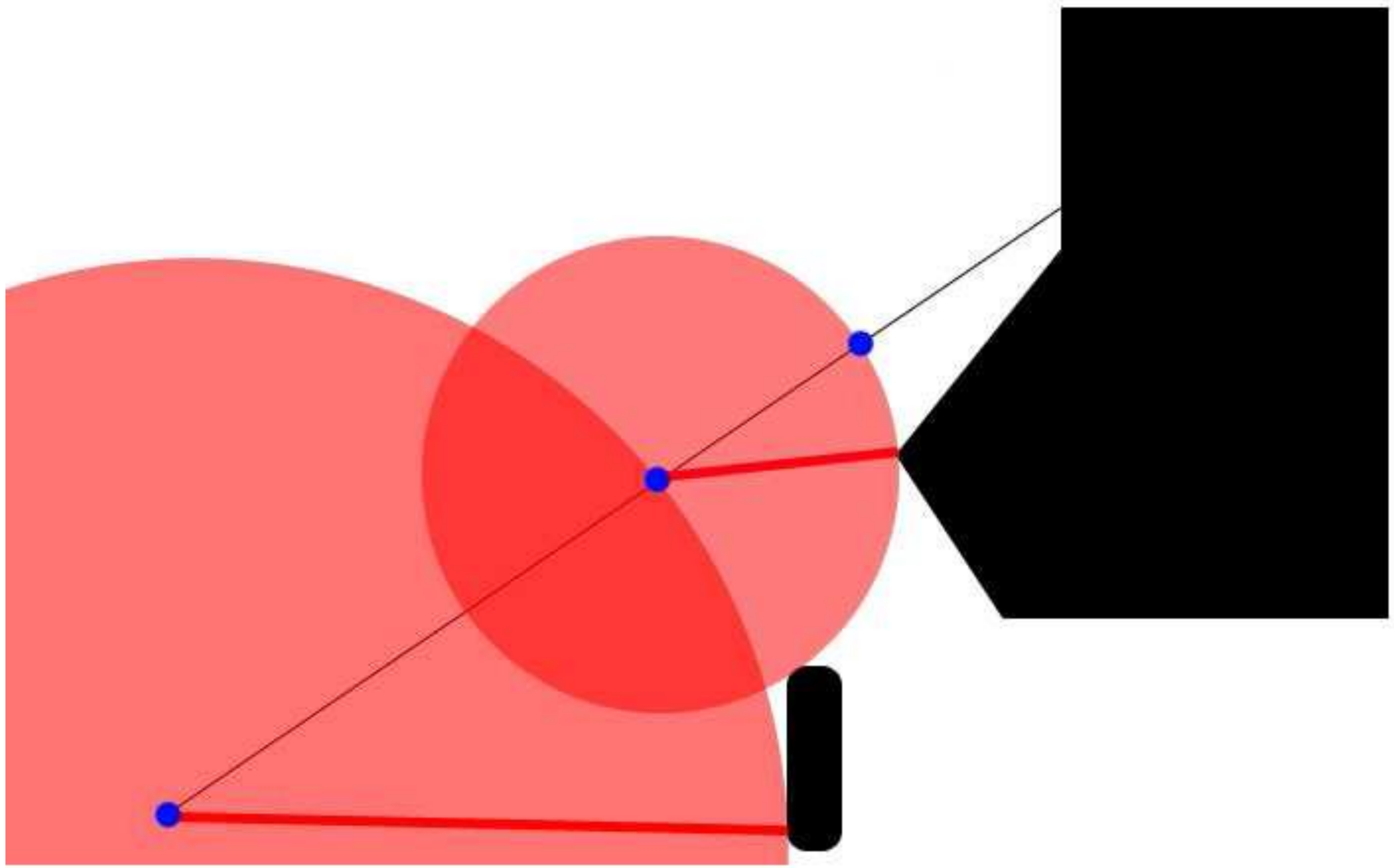
Intersection with SDF



Intersection with SDF



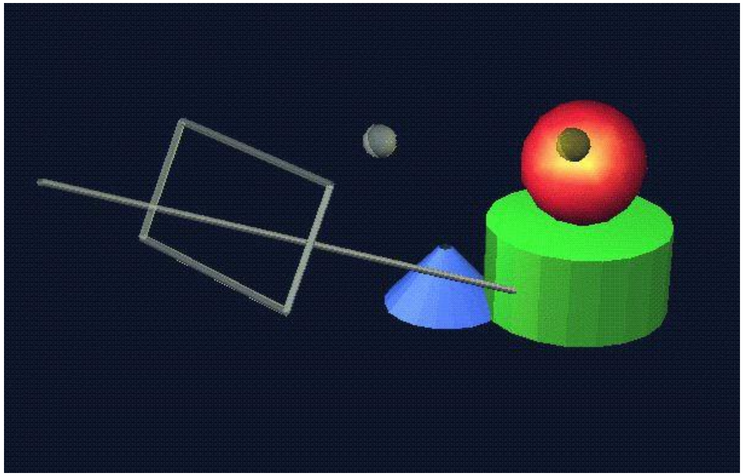
Intersection with SDF



Simplified ray tracer: Shading

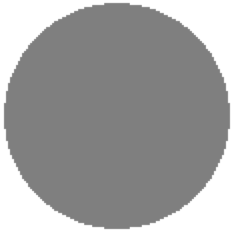
```
Function RayTrace(ray) {
  intersectedObject=QueryScene(ray);
  if (intersectedObject==NIL)
    return backgroundColour;
  return TotalShading(intersectedObject, ray);
}
```

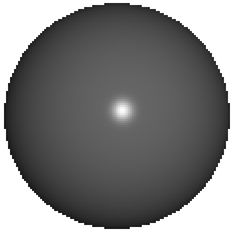
We have determined that the ray intersects the surface, now we need to determine the radiance at the intersection point.



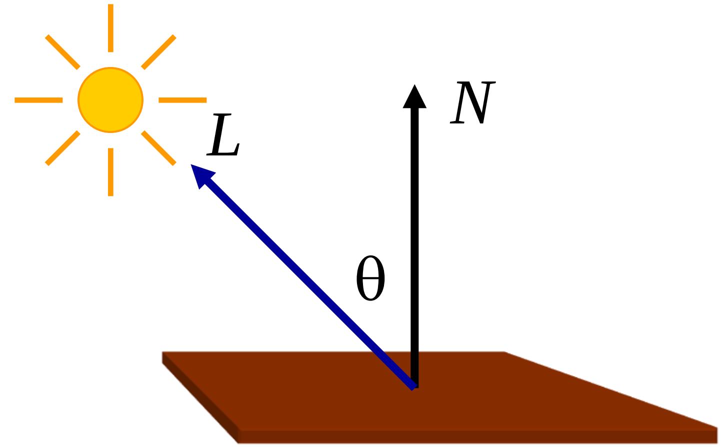
Illumination and shading

- Simple model:
  - For each intersection point, assign for the pixel the colour of the object intersected
  - Easy to compute
  - But not very realistic
  - Good for debugging
- Phong illumination and shading:
  - Apply Phong illumination model at each intersection point
  - Accounts for ambient, diffuse and specular components of light





Diffuse reflection



Ideal diffuse reflection
(Lambert reflection)

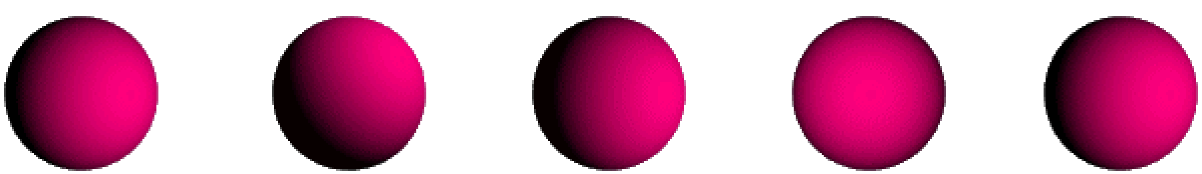
$$\boldsymbol{I}_d = \boldsymbol{k}_d \boldsymbol{I}\ \overline{\cos\theta} = \boldsymbol{k}_d \boldsymbol{I} \max\left(\vec{N}.\vec{L}, 0\right)$$

k_d - diffuse-reflection coefficient

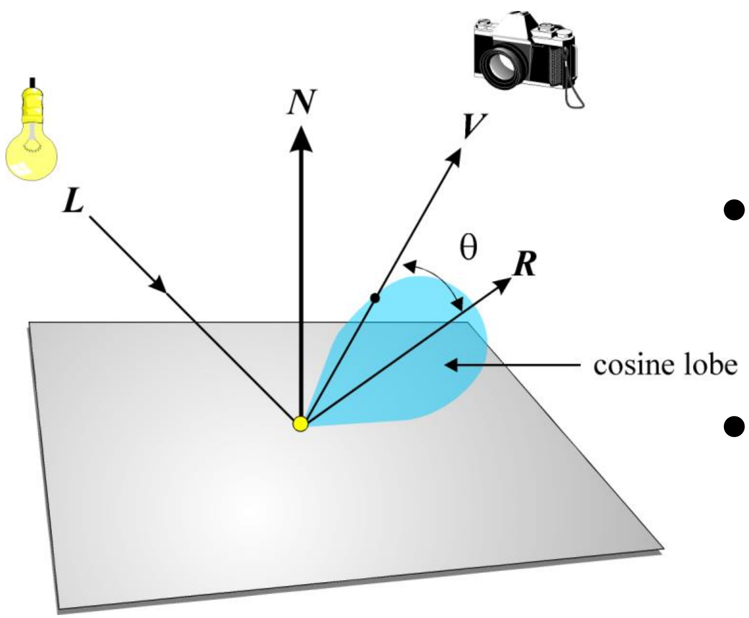
$\vec{N}$: Unit normal vector to the surface

$\vec{L}$: Unit vector from current point to the light source

A sphere seen at different lighting angles



Phong specular term



- Specular component expressed as:

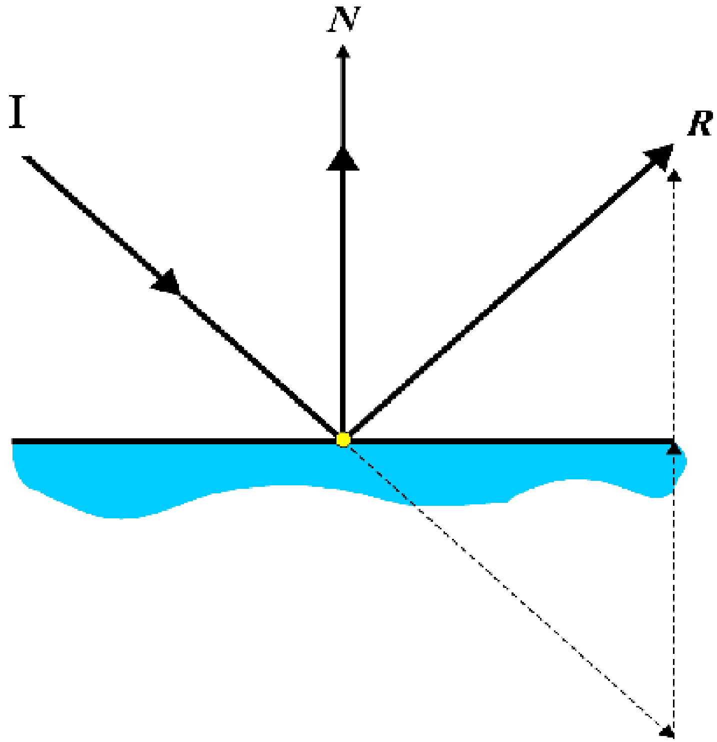
$$\boldsymbol{k_s I} \max\left(\vec{\boldsymbol{V}}.\vec{\boldsymbol{R}},\boldsymbol{0}\right)^m$$

- Where $\boldsymbol{k_s}$ is the specular color and $\boldsymbol{I}$ the light color
- $\vec{\boldsymbol{R}}$ is the reflected ray of light and $\vec{\boldsymbol{V}}$ is the vector from the intersection to the viewer

Computation of the reflected vector

I: Incident vector
R: Reflected vector (pure reflection)

If both I and N are unit, then so is R



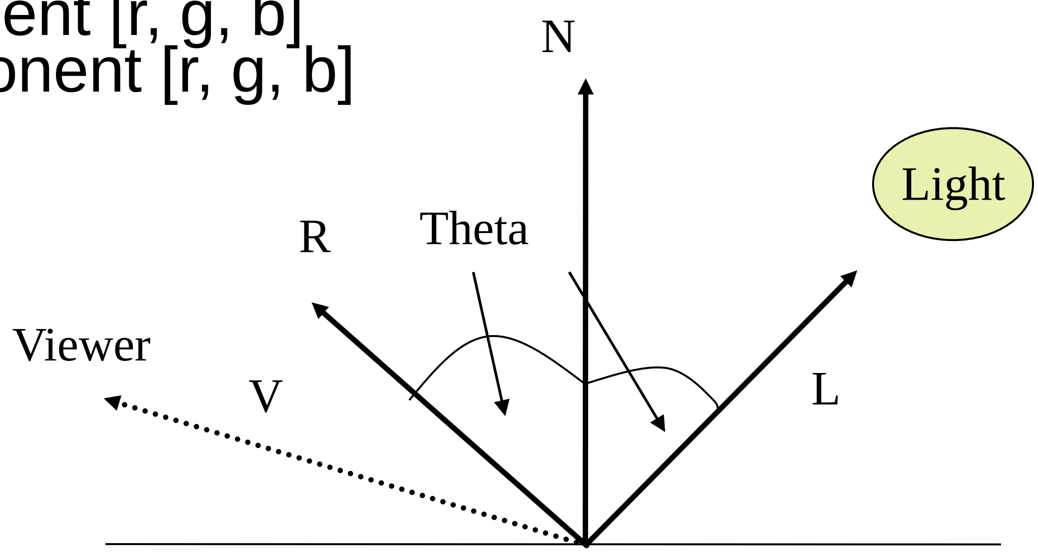
$$R = I + 2N(-I \cdot N)$$
$$= I - 2N(I \cdot N)$$

Total shading

- Adding ambient, diffuse and specular component:

$$I = K_a + \sum_i \left[K_d (L_i N) + K_s (R_i V)^n \right] I_{L_i}$$

Ka: material ambient component [r, g, b]
Kd: material diffuse component [r, g, b]
Ks: material specular component [r, g, b]
Il: light color [r, g, b]



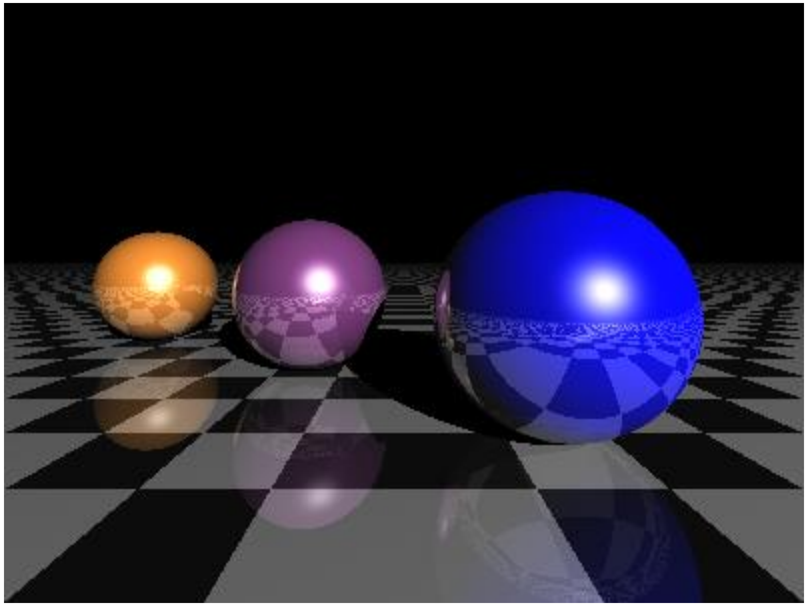
Ray tracing

- Ray tracing is an extension of the ray casting algorithm, where the ray continues to travel after intersection with an object
- Reflected and refracted rays are taking into account and contribute to the color at one pixel
- The function *render* is modified to ray cast these two new rays

Recursive ray tracing

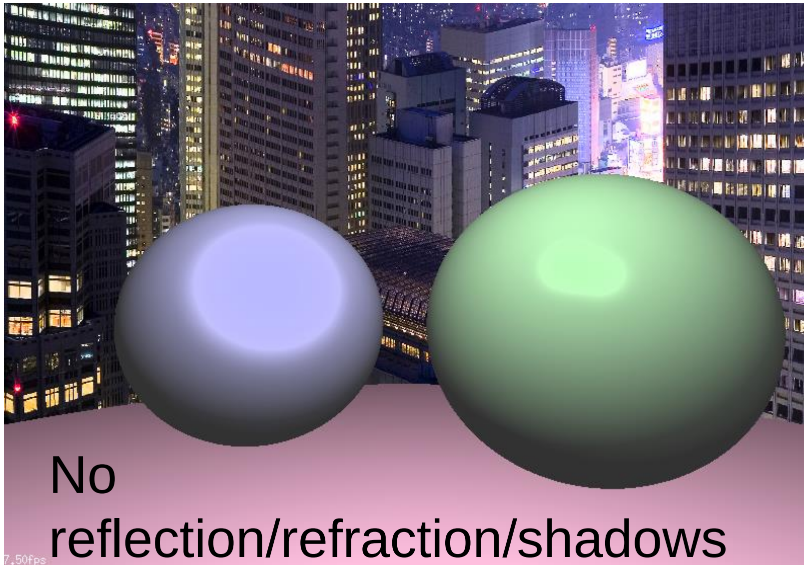
Recursive Ray Tracing:
Cast rays as before but at each ray−surface intersection trace new rays recursively:

- Shadow ray
- Reflected ray
- Refracted ray



Recursive ray tracer

```
Function RayTrace(ray, traceDepth) {
   if (++traceDepth > MAX_TRACE_DEPTH) return Color(0, 0, 0);
   object = QueryScene(ray);
   if (object == NIL) return backgroundColor;
   position = ray.origin + distanceToIntersection*ray.direction;
   intensity = TotalShading(object, ray, position);
  if (object.reflect > 0.0)
     intensity+= object.reflect*RayTrace(ReflectRay(object, position, ray), traceDepth);
  if (object.refract > 0.0)
     intensity+= object.refract*RayTrace(RefractRay(object, position, ray), traceDepth);
   return intensity;
}
```



Recursive ray tracer

```
Function RayTrace(ray, traceDepth) {
  if (++traceDepth > MAX_TRACE_DEPTH) return Color(0,0,0);
  object=QueryScene(ray);
  if (object==NIL) return backgroundColour;
  position = ray.origin + distanceToIntersection*ray.direction;
  intensity =TotalShading(object, ray, position);
  if (object.reflect>0.0)
    intensity += object.reflect*RayTrace(ReflectRay(object, position, ray), traceDepth);
  if (object.refract>0.0)
    intensity += object.refract*RayTrace(RefractRay(object, position, ray), traceDepth);
  return intensity;
}
```

Recursively call the ray tracing function with a new ray: the reflected ray

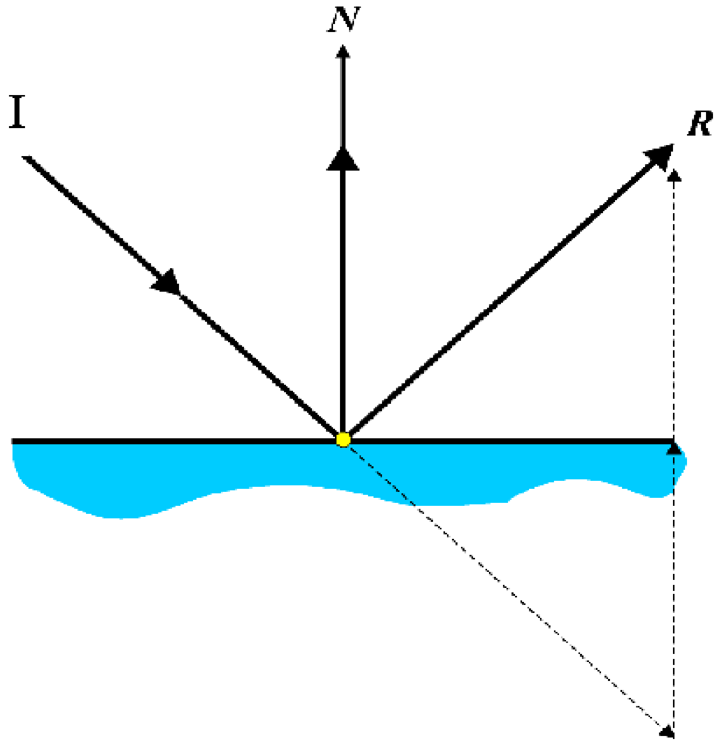
Compute a reflected ray starting at the intersection between the object and the incident ray

Computation of the reflected vector

I: Incident vector

R: Reflected vector (pure reflection)

If both I and N are unit, then so is R



$$R = I + 2N(-I \cdot N)$$
$$= I - 2N(I \cdot N)$$

Recursive ray tracer

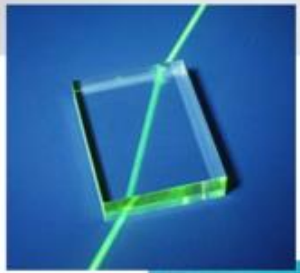
```
Function RayTrace(ray, traceDepth) {
  if (++traceDepth > MAX_TRACE_DEPTH) return Color(0,0,0);
  object=QueryScene(ray);
  if (object==NIL) return backgroundColour;
  position = ray.origin + distanceToIntersection*ray.direction;
  intensity =TotalShading(object, ray, position);
  if (object.reflect>0.0)
    intensity += object.reflect*RayTrace(ReflectRay(object, position, ray), traceDepth);
  if (object.refract>0.0)
    intensity += object.refract*RayTrace(RefractRay(object, position, ray), traceDepth);
  return intensity;
}
```

Recursively call the ray tracing function with a new ray: the reflected ray

Compute a refracted ray starting at the intersection between the object and the incident ray

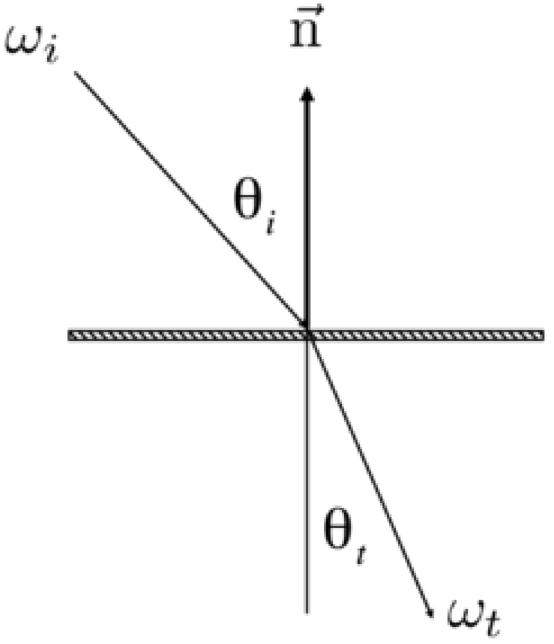
Transmission

In addition to reflecting off a surface, light can also be transmitted through a surface.



Snell's law

The transmitted angle depends on the index of refraction of the medium in which the ray propagates (η_i), and the index of refraction of the medium entered by the ray (η_t)

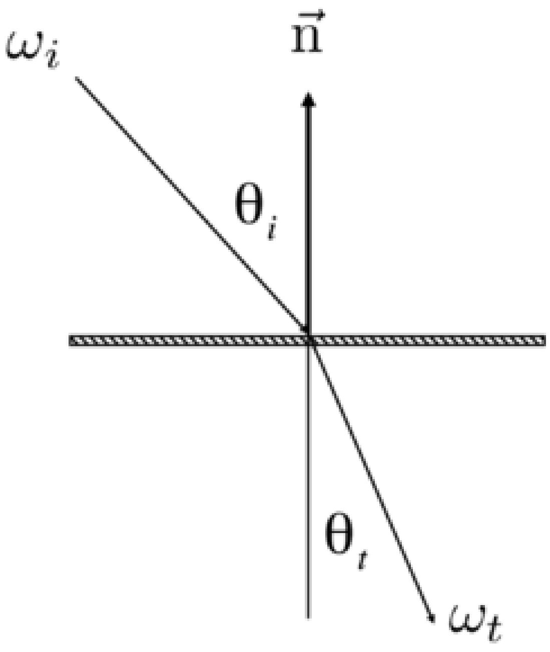


$$\eta_i \sin\theta_i = \eta_t \sin\theta_t$$

Examples:

- Vacuum: 1
- Air: ~1
- Water: 1.333
- Glass: 1.5-1.6

Refraction



$$\eta_i \sin\theta_i = \eta_t \sin\theta_t$$

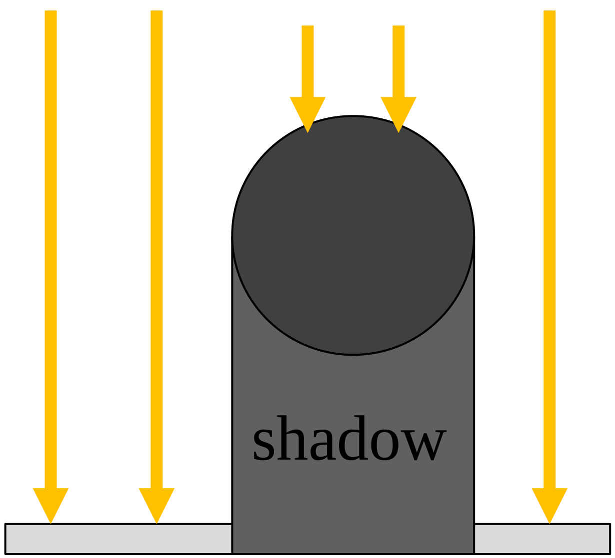
$$\eta_i \sin\theta_i = \eta_t \sin\theta_t$$

$$\begin{aligned}\cos\theta_t &= \sqrt{1-\sin^2\theta_t} \\ &= \sqrt{1-\left(\frac{\eta_i}{\eta_t}\right)^2 \sin^2\theta_i} \\ &= \sqrt{1-\left(\frac{\eta_i}{\eta_t}\right)^2 (1-\cos^2\ \theta_i)}\end{aligned}$$

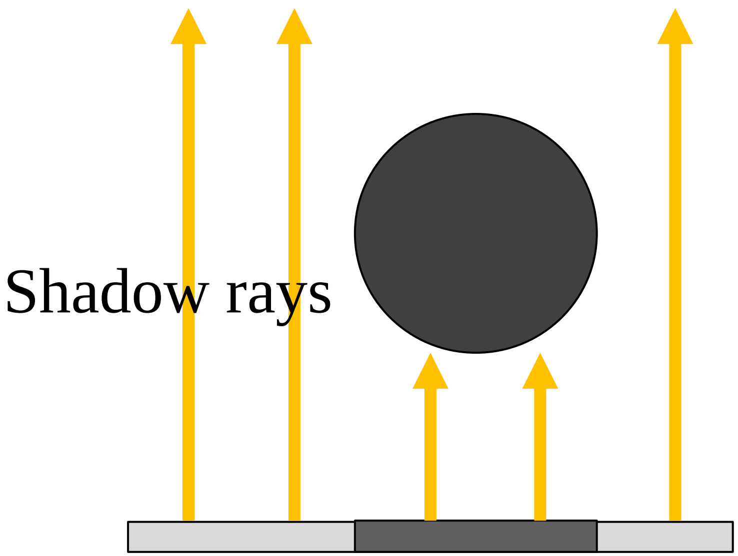
The transmitted vector is then:

$$\boldsymbol{\omega}_t = \frac{\eta_i}{\eta_t}\boldsymbol{\omega}_i + \left(\frac{\eta_i}{\eta_t}\cos\theta_i - \cos\theta_t\right)\boldsymbol{n}$$

Shadow



Real world



CG

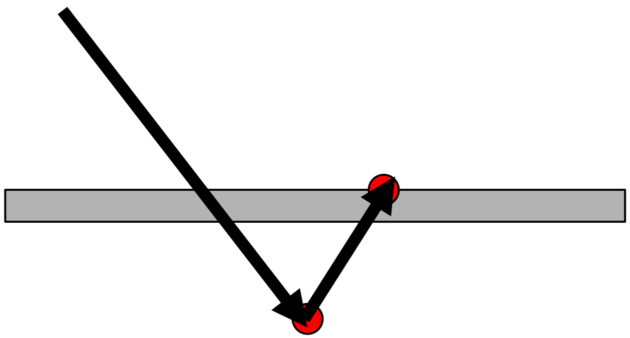
Shadow

```
Function TotalShading(object, ray, intersect) {
  intensity=object.AmbientIntensity;
  lightRay.origin=intersect;
  foreach light in LightList do {
    lightRay.direction=light.direction;
    if (QueryScene(lightRay)==NIL))
      intensity += DiffuseShading(object, light, intersect) +
                SpecularShading(object, light, intersect, ray);
  }
 return intensity;
}
```

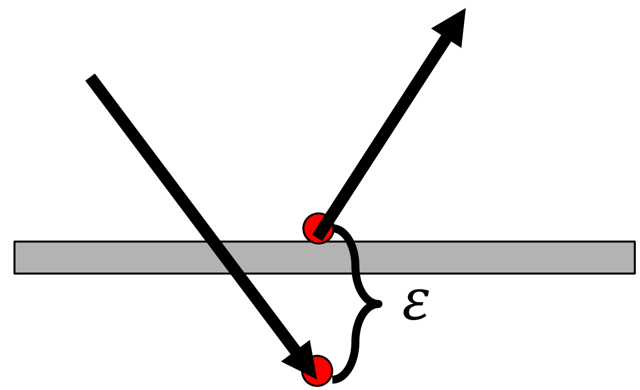
For shadows: check if the ray to the light source is intersecting another object in the scene

If not then compute direct illumination from light (e.g. by using Phong model)

Secondary rays and numerical precision



Re-intersection with same surface



Prevent re-intersection by adding a small displacement (ε) to the intersection