Ray Tracing

Zoran Popovic
CSE 457

Reading

Required:

- Marschner and Shirley, Ch. 4, Section 13.1-13.2 (online handout)
- Triangle intersection (online handout)

Further reading:

- Shirley errata on syllabus page, needed if you work from his book instead of the handout, which has already been corrected.
- T. Whitted. An improved illumination model for shaded display. Communications of the ACM 23(6), 343-349, 1980.
- A. Glassner. An Introduction to Ray Tracing. Academic Press, 1989.
- K. Turkowski, "Properties of Surface Normal Transformations," Graphics Gems, 1990, pp. 539-547.

Geometric optics

Modern theories of light treat it as both a wave and a particle.

We will take a combined and somewhat simpler view of light – the view of **geometric optics**.

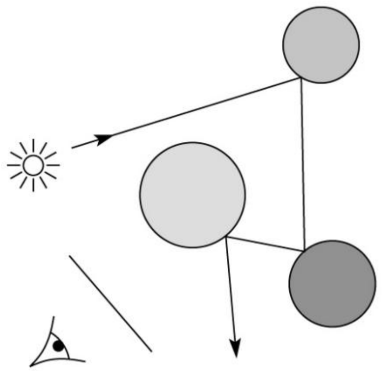
Here are the rules of geometric optics:

- Light is a flow of photons with wavelengths. We'll call these flows “light rays.”
- Light rays travel in straight lines in free space.
- Light rays do not interfere with each other as they cross.
- Light rays obey the laws of reflection and refraction.
- Light rays travel from the light sources to the eye, but the physics is invariant under path reversal (reciprocity).

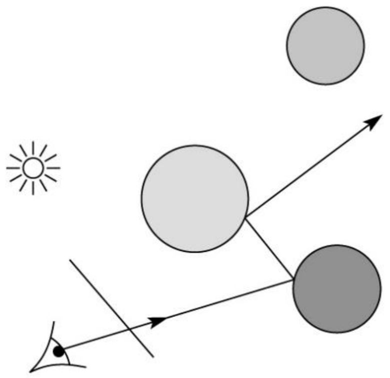
Eye vs. light ray tracing

Where does light begin?

At the light: light ray tracing (a.k.a., forward ray tracing or photon tracing)



At the eye: eye ray tracing (a.k.a., backward ray tracing)

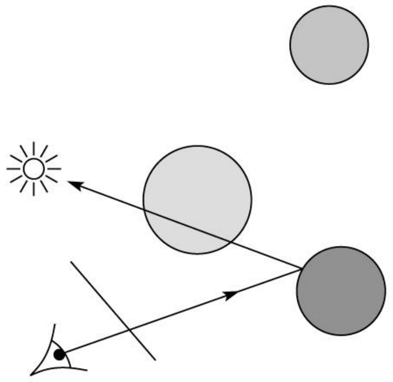


We will generally follow rays from the eye into the scene.

Precursors to ray tracing

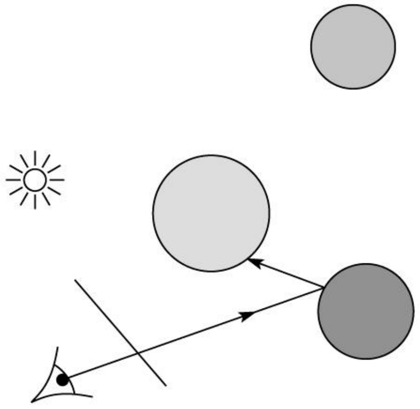
Local illumination

- Cast one eye ray, then shade according to light



Appel (1968)

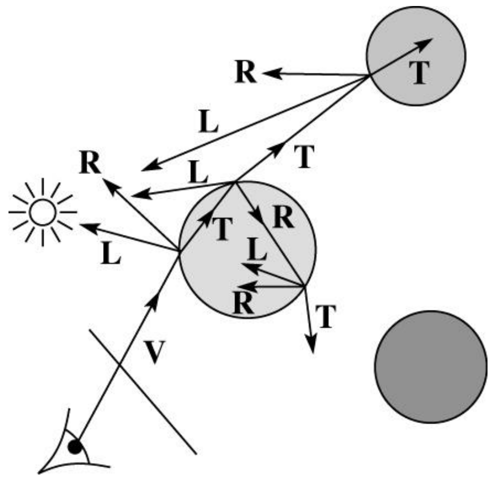
- Cast one eye ray + one ray to light



Whitted ray-tracing algorithm

In 1980, Turner Whitted introduced ray tracing to the graphics community.

- Combines eye ray tracing + rays to light
- Recursively traces rays



Algorithm:

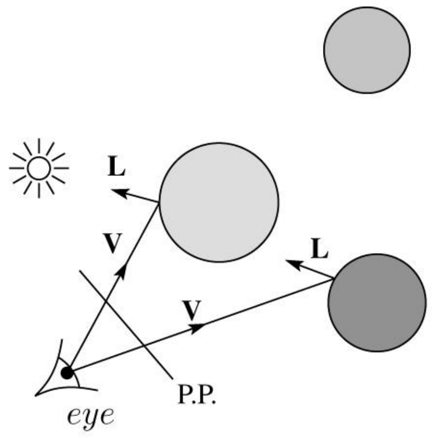
2. For each pixel, trace a **primary ray** in direction **V** to the first visible surface.

3. For each intersection, trace **secondary rays**:

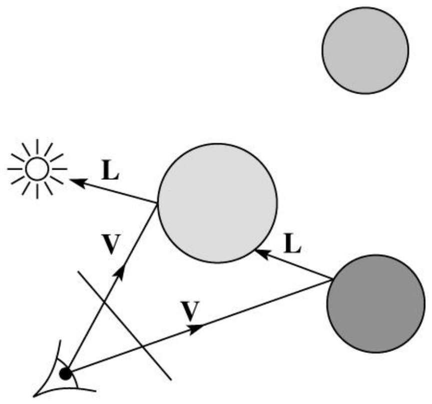
- **Shadow rays** in directions $\mathbf{L}_i$ to light sources
- **Reflected ray** in direction **R**.
- **Refracted ray** or **transmitted ray** in direction **T**.

Whitted algorithm (cont'd)

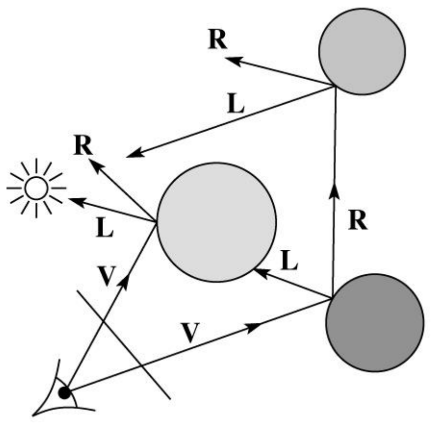
Let's look at this in stages:



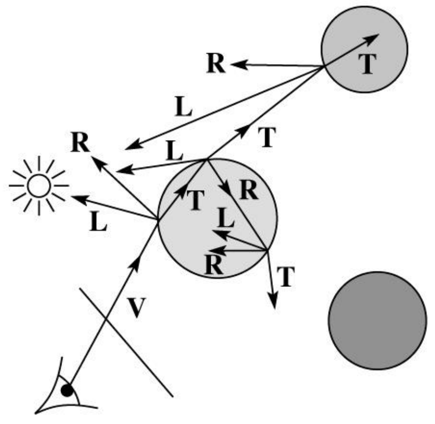
Primary rays



Shadow rays



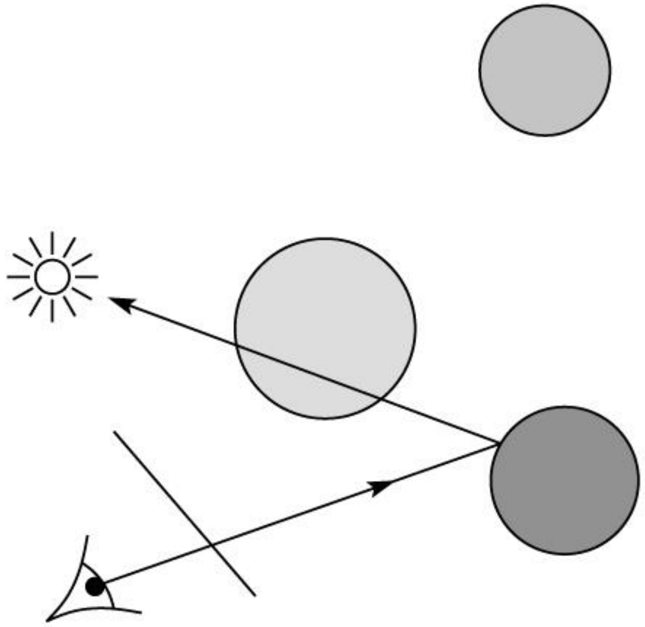
Reflection rays



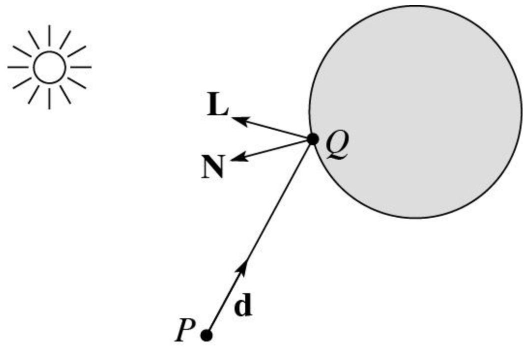
Refracted rays

Ray casting and local illumination

Now let's actually build the ray tracer in stages. We'll start with ray casting and local illumination:



Direct illumination



A ray is defined by an origin P and a unit direction $\mathbf{d}$ and is parameterized by $t > 0$:

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

Let $I(P, \mathbf{d})$ be the intensity seen along a ray. Then:

$$I(P, \mathbf{d}) = I_{\text{direct}}$$

where

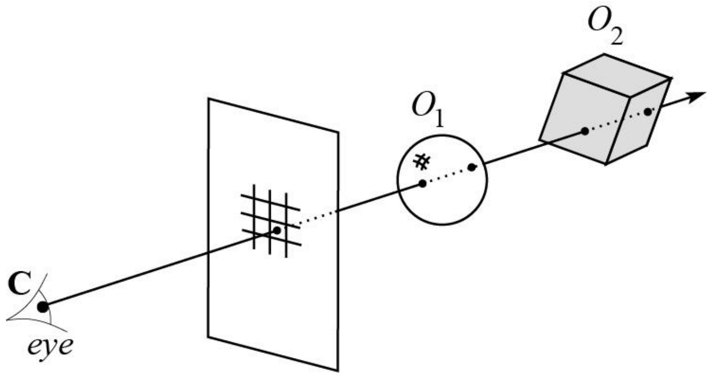
- I_{direct} is computed from the Blinn-Phong model

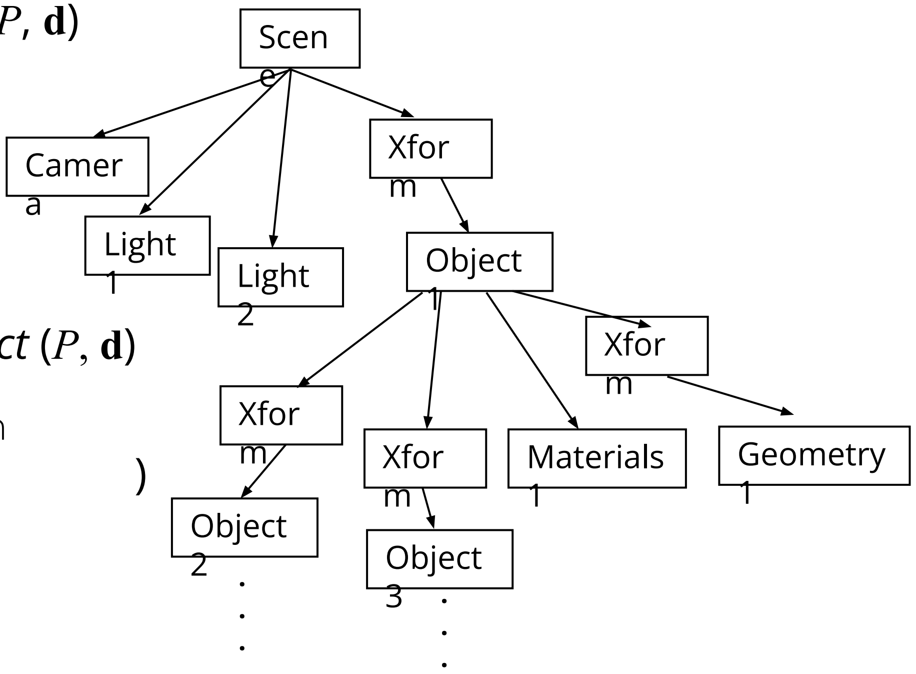
Ray-tracing pseudocode

We build a ray traced image by casting rays through each of the pixels.

```
function traceImage (scene):
        for each pixel $(i, j)$ in image
            $A$ = pixelToWorld $(i, j)$
            $P = \mathbf{C}$
            $\mathbf{d} = (A - P)/\| A - P \|$
            $I(\mathrm{i,j})$ = traceRay (scene, $P$, $\mathbf{d}$)
        end for
end function

function traceRay (scene, $P$, $\mathbf{d}$):
        ($t_\cap$, $\mathbf{N}$, mtrl) $\leftarrow$ scene.intersect ($P$, $\mathbf{d}$)
        $Q$ □ ray ($P$, $\mathbf{d}$) evaluated at $t_\cap$
        $I$ = shade (                              )
        return $I$
end function
```

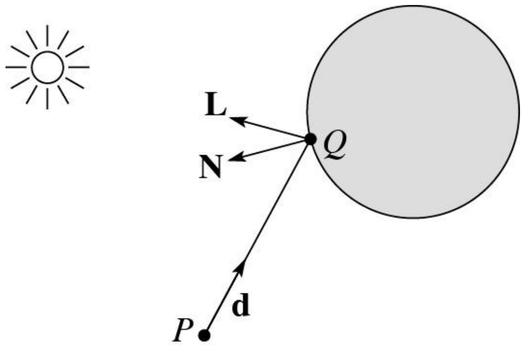




Shading pseudocode

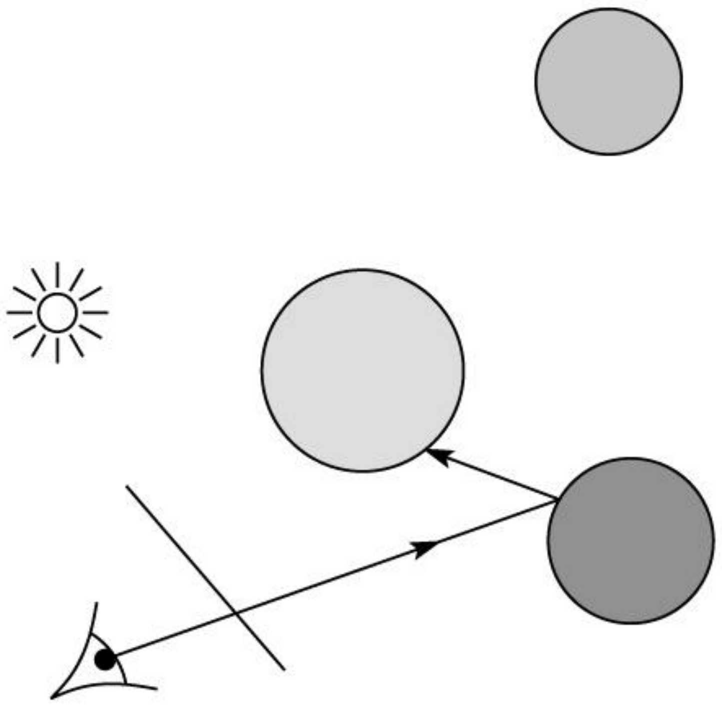
Next, we need to calculate the color returned by the *shade* function.

```
function shade (mtrl, scene, $Q$, $\mathbf{N}$, $\mathbf{d}$):
    $I \leftarrow$ mtrl.$k_e$
    for each light source Light do:
        atten = Light -> distanceAttenuation (          )
        $\mathbf{L}$ = Light -> getDirection (          )
        $I \leftarrow$ I + ambient + atten*(diffuse + specular)
    end for
    return $I$
end function
```



Ray casting with shadows

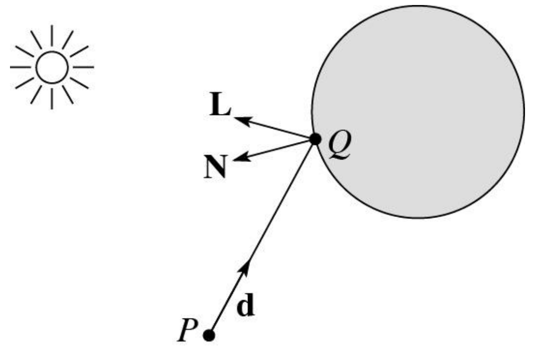
Now we'll add shadows by casting shadow rays:



Shading with shadows

To include shadows, we need to modify the shade function:

```
**function** *shade* (mtrl, scene, $Q$, $\mathbf{N}$, $\mathbf{d}$):
    I ← mtrl.$k_e$
    **for each** light source Light **do**:
        atten = Light -> *distanceAttenuation*($Q$ ) *
                Light -> *shadowAttenuation*( )
        $\mathbf{L}$ = Light -> getDirection ($Q$ )
        $I \leftarrow I$ + ambient + atten*(diffuse + specular)
    **end for**
    **return** $I$
**end function**
```

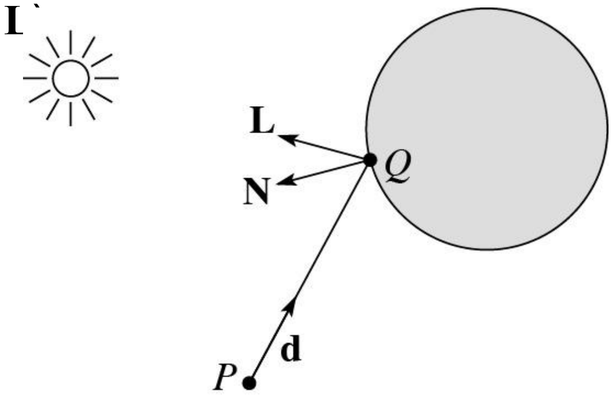


Shadow attenuation

Computing a shadow can be as simple as checking to see if a ray makes it to the light source.

For a point light source:

```
function PointLight::shadowAttenuation(scene, $Q$)
        $\mathbf{L}$ = getDirection($Q$)
        ($t_\cap$, $\mathbf{N}$, mtrl) $\leftarrow$ scene.intersect($Q$, $\mathbf{L}$
        Compute $t_{\text{light}}$
        if ($t_\cap < t_{\text{light}}$) then:
            atten = (0, 0, 0)
        else
            atten = (1, 1, 1)
        end if
        return atten
end function
```



Note: we will later handle color-filtered shadowing, so this function needs to return a *color* value.

Shading in "Trace"

The Trace project uses a version of the Blinn-Phong shading equation we derived in class, with two modifications:

- Distance attenuation is clamped to be at most 1:

$$A_j^{dist} = \min\left\{1, \frac{1}{a_j r_j^2 + b_j r_j + c_j}\right\}$$

- Shadow attenuation A^{shadow} is included and is RGB-valued.

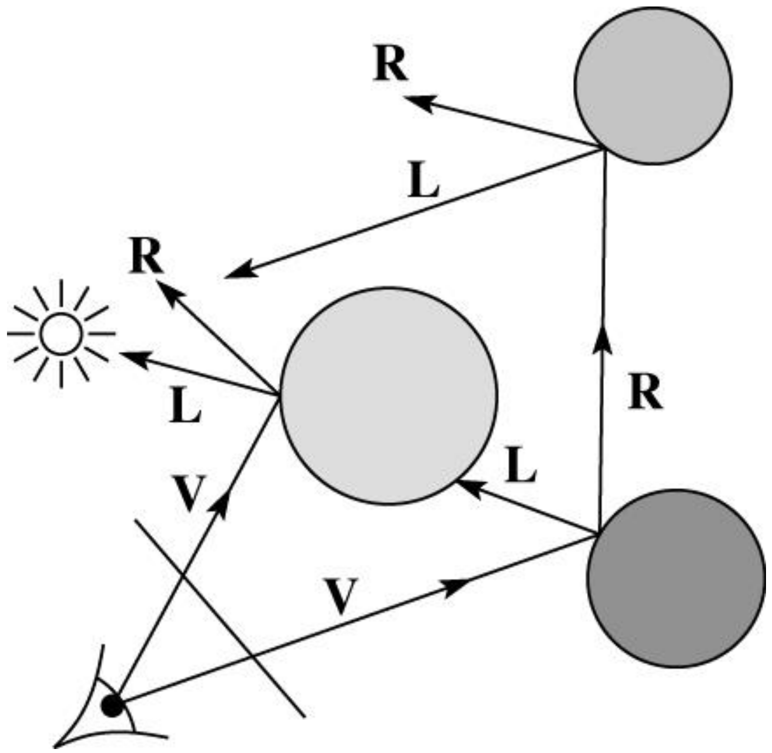
Here's what the shading equation should look like:

$$I = k_e + \sum_j k_d I_{La,j} + A_j^{shadow} A_j^{dist} I_{L,j} B_j \left[k_d \left(\mathbf{N} \cdot \mathbf{L}_j \right) + k_s \left(\mathbf{N} \cdot \mathbf{H}_j \right)_+^{n_s} \right]$$

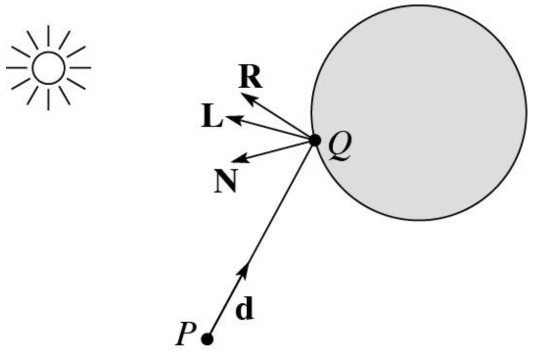
This is the shading equation to use in the Trace project!

Recursive ray tracing with reflection

Now we'll add reflection:



Shading with reflection



Let $I(P, \mathbf{d})$ be the intensity seen along a ray. Then:

$$I\,(P, \mathbf{d}) = I_{\text{direct}} + I_{\text{reflected}}$$

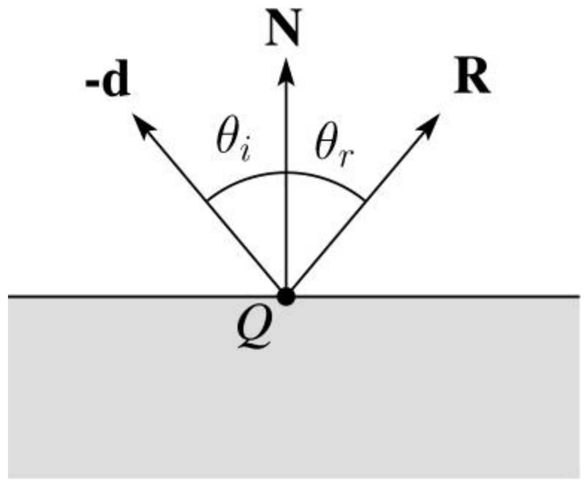
where

- I_{direct} is computed from the Blinn-Phong model, plus shadow attenuation
- $I_{\text{reflected}} = k_s\; I(Q, \mathbf{R})$

Remember that is a color value.

(Sometimes another variable, k_r, is used instead of k_s to allow for separate control of specular light

Reflection



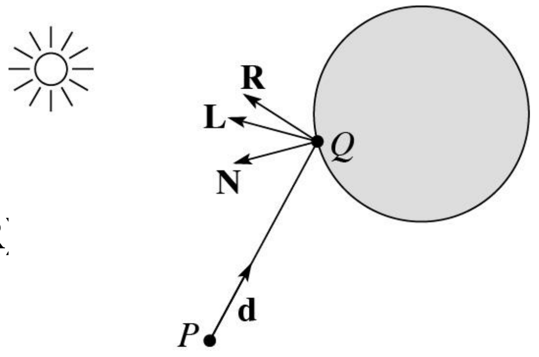
Law of reflection:

$$\theta_i = \theta_r$$

R is co-planar with **d** and **N**.

Ray-tracing pseudocode, revisited

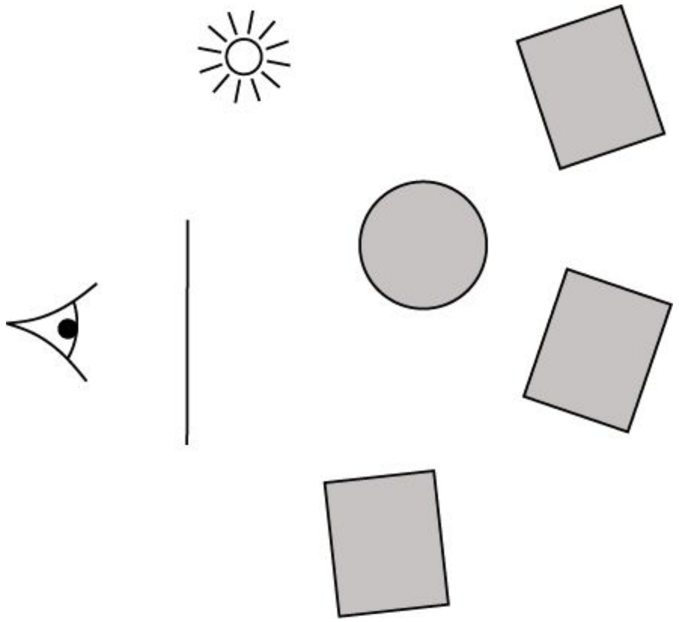
```
function traceRay (scene, $P$, $\mathbf{d}$):
        ($t_\cap$, $\mathbf{N}$, mtrl) $\leftarrow$ scene.intersect ($P$, $\mathbf{d}$)
        $Q$ □ ray ($P$, $\mathbf{d}$) evaluated at $t_\cap$
        $I$ = shade (scene, mtrl, $Q$, $\mathbf{N}$, $\mathbf{d}$)
        $\mathbf{R}$ = reflectDirection (                    )
        $I \leftarrow I$ + mtrl.$k_s$ * traceRay(scene, $Q$, $\mathbf{R}$
        return $I$
end function
```



Terminating recursion

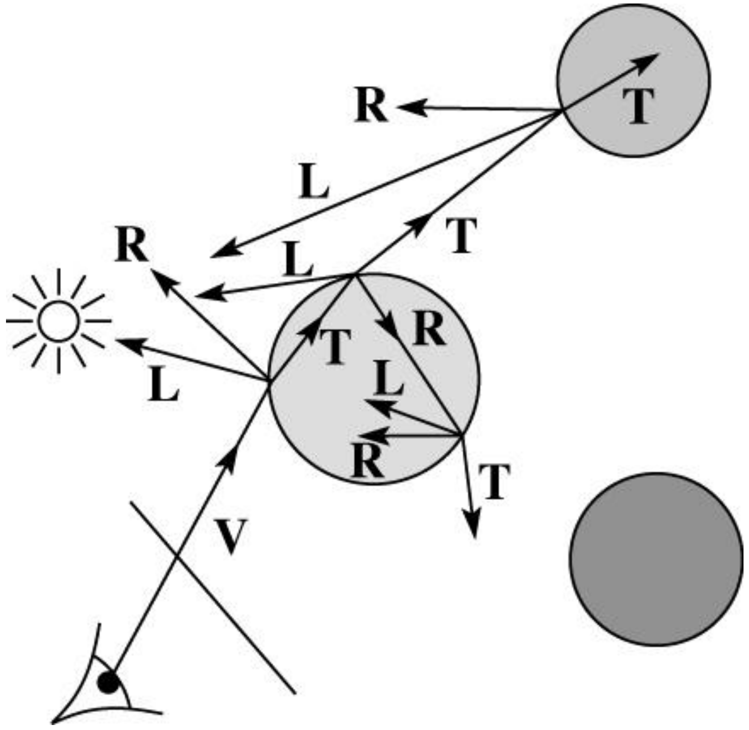
Q: How do you bottom out of recursive ray tracing?

Possibilities:

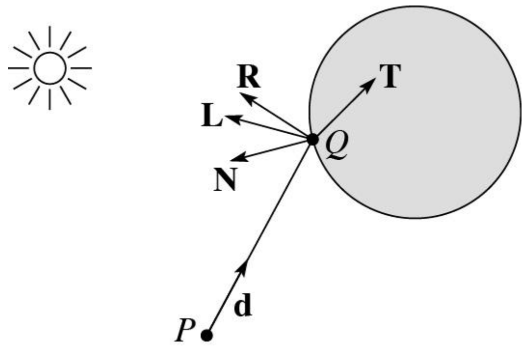


Whitted ray tracing

Finally, we'll add refraction, giving us the Whitted ray tracing model:



Shading with reflection and refraction



Let $I(P, \mathbf{d})$ be the intensity seen along a ray. Then:

$$I(P, \mathbf{d}) = I_{\text{direct}} + I_{\text{reflected}} + I_{\text{transmitted}}$$

where

- I_{direct} is computed from the Blinn-Phong model, plus shadow attenuation
- $I_{\text{reflected}} = k_s I(Q, \mathbf{R})$
- $I_{\text{transmitted}} = k_t I(Q, \mathbf{T})$

Typically, we set $k_t = 1 - k_s$ (or $(0,0,0)$, if opaque, where k_t is a color value).

[Generally, for ideal specular surfaces, k_s and k_t are determined by “Fresnel reflection,” which depends on angle of incidence and changes the polarization of the light. This is discussed in

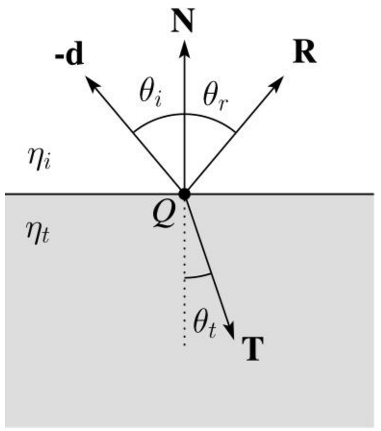
Refraction

Snell's law of refraction:

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

where η_i , η_t are **indices of refraction**.

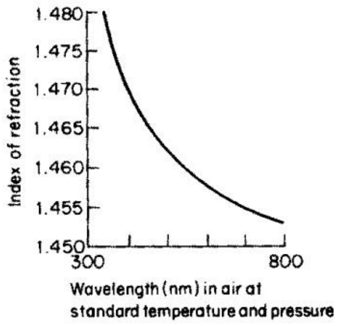
In all cases, $\mathbf{R}$ and $\mathbf{T}$ are co-planar with $\mathbf{d}$ and $\mathbf{N}$.



The index of refraction is material dependent.

It can also vary with wavelength, an effect called **dispersion** that explains the colorful light rainbows from prisms. (We will generally assume no dispersion.)

Medium	Index of refraction
Vaccum	1
Air	1.0003
Water	1.33
Fused quartz	1.46
Glass, crown	1.52
Glass, dense flint	1.66
Diamond	2.42



Index of refraction variation for fused quartz

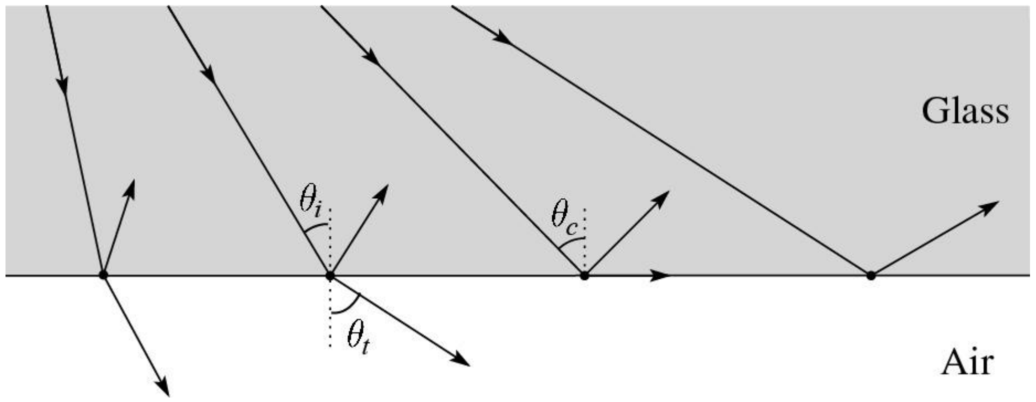
Total Internal Reflection

The equation for the angle of refraction can be computed from Snell's law:

What “bad thing” can happen when $\eta_i > \eta_t$?

When θ_t is exactly 90°, we say that θ_i has achieved the “critical angle” θ_c .

For $\theta_i > \theta_c$, *no rays are transmitted*, and only reflection occurs, a phenomenon known as “total internal reflection” or TIR.



Marschner's notation

Marschner uses different symbols. Here is the translation between them:

$$\mathbf{r} = \mathbf{R}$$

$$\mathbf{t} = \mathbf{T}$$

$$\phi = \theta_t$$

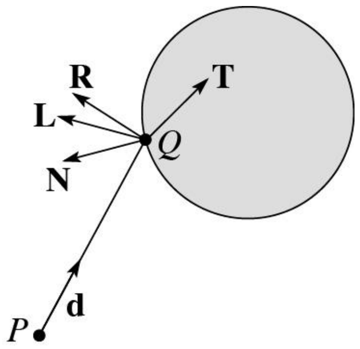
$$\theta = \theta_r = \theta_i$$

$$n = \eta_i$$

$$n_t = \eta_t$$

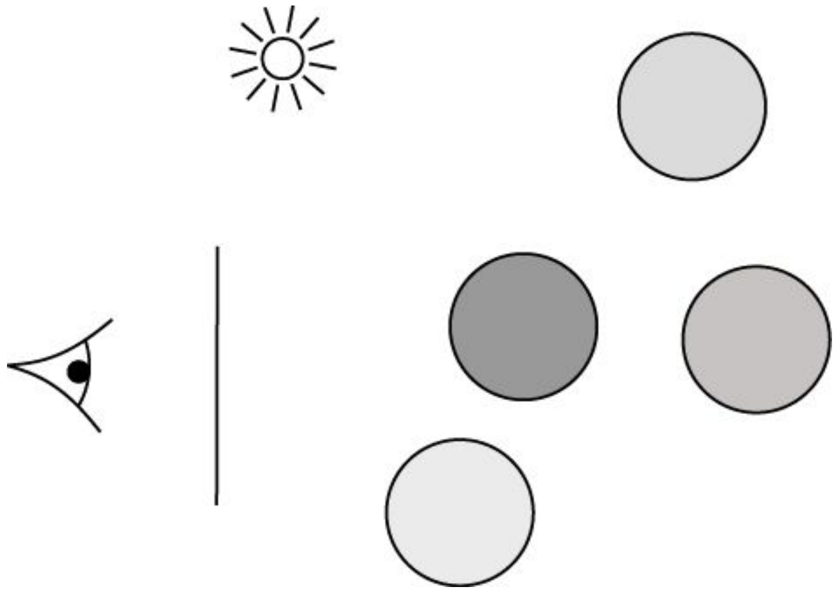
Ray-tracing pseudocode, revisited

```
function traceRay (scene, $P$, $\mathbf{d}$):
        ($t_\cap$, $\mathbf{N}$, mtrl) $\leftarrow$ scene.intersect ($P$, $\mathbf{d}$)
        $Q$ □ ray ($P$, $\mathbf{d}$) evaluated at $t_\cap$
        $I$ = shade (scene, mtrl, $Q$, $\mathbf{N}$, $\mathbf{d}$)
        $\mathbf{R}$ = reflectDirection ($\mathbf{N}$, $\mathbf{d}$)
        $I \leftarrow I$ + mtrl.$k_s$ * traceRay (scene, $Q$, R
        if ray is entering object then
            $\eta_i$ = index_of_air (=1.0003)
            $\eta_t$ = mtrl.index
        else
            $\eta_i$ = mtrl.index
            $\eta_t$ = index_of_air (=1.0003)
        if (notTIR (
)) then
            $\mathbf{T}$ = refractDirection (
)
            $I \leftarrow I$ + mtrl.$k_t$ * traceRay (scene, $Q$,
$\mathbf{T}$)
        end if
        return $I$
end function
```



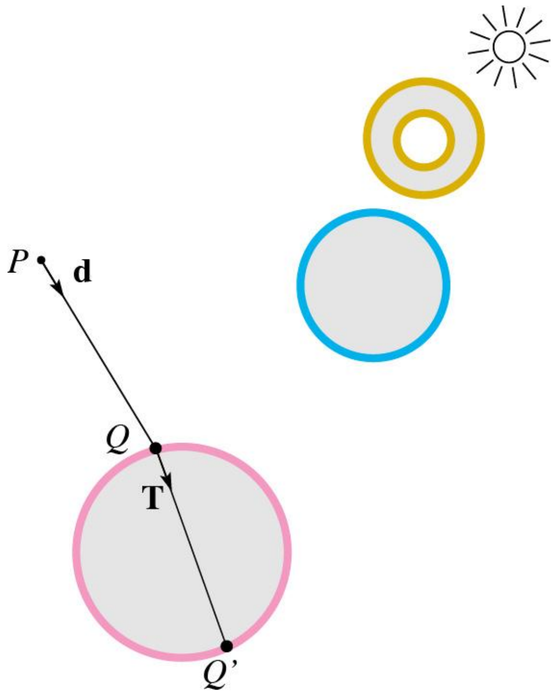
Terminating recursion, incl. refraction

Q: *Now* how do you bottom out of recursive ray tracing?



Shadow attenuation (cont'd)

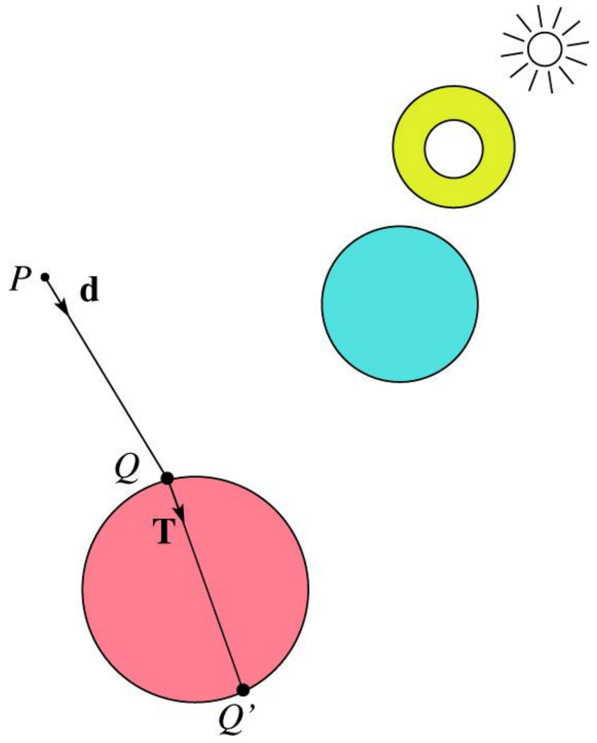
Q: What if there are transparent objects along a path to the light source?



We'll take the view that the color is really only at the surface, like a glass object with a colored transparency coating on it. In this case, we multiply in the transparency constant, k_t, every time an object is entered or exited, possibly more than once for the same object.

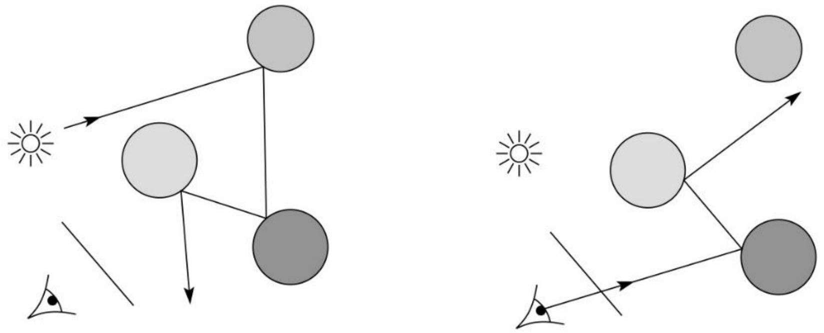
Shadow attenuation (cont'd)

Another model would be to treat the glass as solidly colored in the interior. Marschner's textbook describes a the resulting volumetric attenuation based on Beer's Law, which you can implement for extra credit.

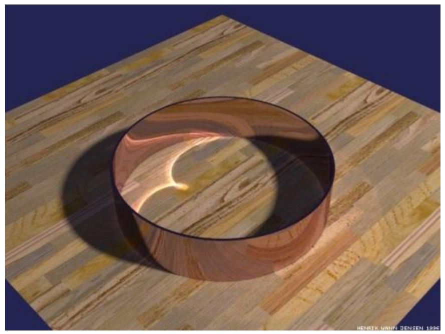


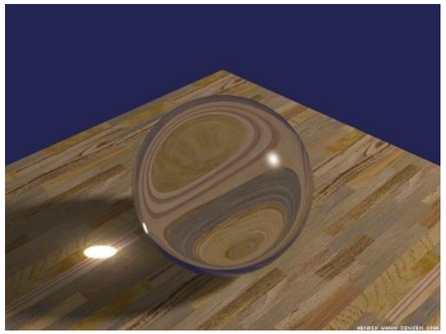
Photon mapping

Combine light ray tracing (photon tracing) and eye ray tracing:



...to get **photon mapping**.





Renderings by Henrik Wann Jensen:

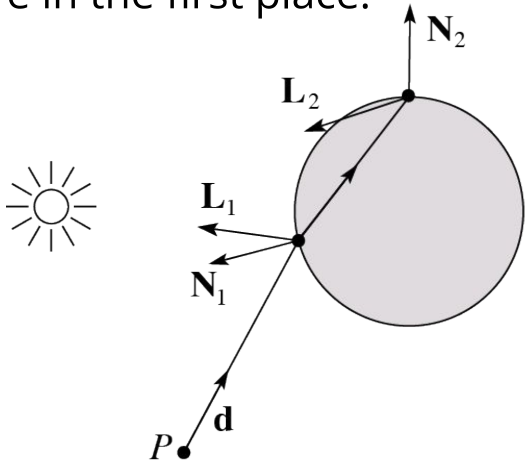
http://graphics.ucsd.edu/~henrik/images/caustics.html

Normals and shading, reflection, and refraction when inside

When a ray is inside an object and intersects the object's surface on the way out, the normal will be pointing ***away*** from the ray (i.e., the normal always points to the outside by default).

You must ***negate*** the normal before doing any of the **shading, reflection, and refraction** that follows.

Finally, when shading a point inside of an object, apply k_t to the ambient component, since that "ambient light" had to pass through the object to get there in the first place.

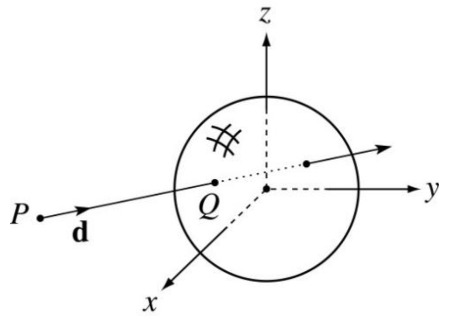


Geometric Aspects

Intersecting rays with spheres

Now we’ve done everything except figure out what that “scene.*intersect* (P, $\mathbf{d}$)” function does.

Mostly, it calls each object to find out the t-value at which the ray intersects the object. Let’s start with intersecting spheres…



Given:

- The coordinates of a point along a ray passing through P in the direction $\mathbf{d}$ are:

$$x = P_x + td_x$$
$$y = P_y + td_y$$
$$z = P_z + td_z$$

- A sphere S of radius r centered at the origin defined by the equation:

Intersecting rays with spheres

Solution by substitution:

$$x^2 + y^2 + z^2 - r^2 = 0$$

$$(P_x + td_x)^2 + (P_y + td_y)^2 + (P_z + td_z)^2 - r^2 = 0$$

$$at^2 + bt + c = 0$$

where

$$a = d_x^2 + d_y^2 + d_z^2$$

$$b = 2(P_x d_x + P_y d_y + P_z d_z)$$

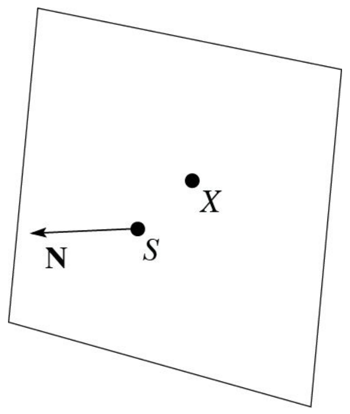
$$c = P_x^2 + P_y^2 + P_z^2 - r^2$$

Q: What are the solutions of the quadratic equation in t and what do they mean?

Q: What is the normal to the sphere at a point (x, y, z) on the sphere?

Note: the Trace project only requires you to handle a sphere of radius $r = 0.5$. This sphere may be arbitrarily transformed

Ray-plane intersection



Next, we will considering intersecting a ray with a plane.

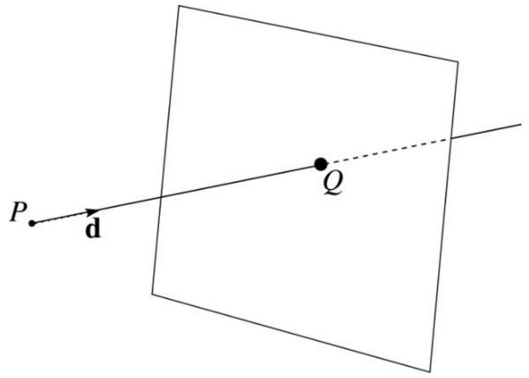
To do this, we first need to define the plane equation.

Given a point S on a plane with normal $\mathbf{N}$, how would we determine if another point X is on the plane?

(Hint: start by forming the vector X - S.)

This is the plane equation!

Ray-plane intersection (cont'd)

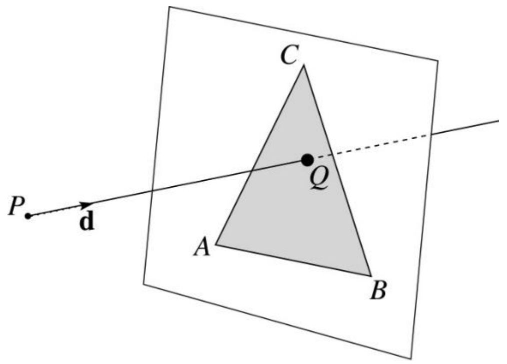


Now consider a ray intersecting a plane, where the plane equation is:

$$\mathbf{N} \cdot X = k$$

We can solve for the intersection parameter (and thus the point) by substituting X with the ray $P + t\,\mathbf{d}$:

Ray-triangle intersection



To intersect with a triangle, we first solve for the equation of its supporting plane.

How might we compute the (un-normalized) normal?

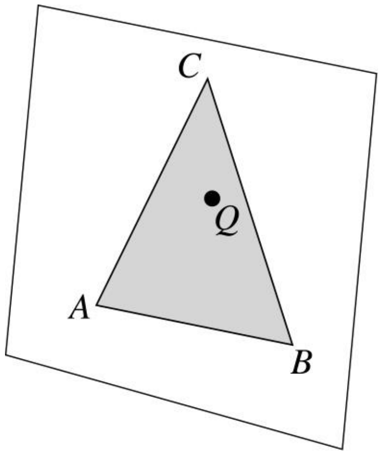
Given this normal, how would we compute k?

Using these coefficients, we can intersect the ray with the triangle to solve for Q.

Now, we need to decide if Q is inside or outside of the triangle…

3D inside-outside test

One way to do this "inside-outside test," is to see if Q lies on the left side of each edge as we move counterclockwise around the triangle.

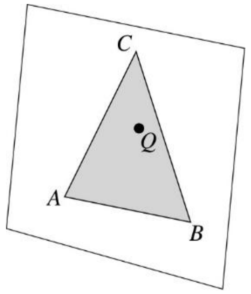


How might we use cross and products to do this?

Barycentric coordinates

As we'll see in a moment, it is often useful to represent Q as an **affine combination** of A, B, and C:

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



where:

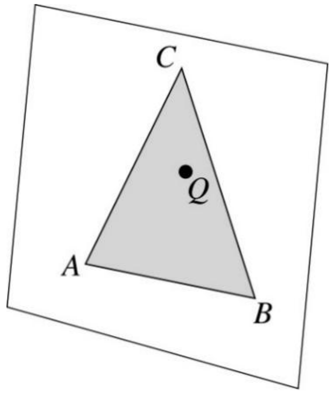
$$\alpha + \beta + \gamma = 1$$

We call α, β, and γ, the **barycentric coordinates** of Q with respect to A, B, and C.

Computing barycentric coordinates

Given a point Q that is inside of triangle ABC, we can solve for Q's barycentric coordinates in a simple way:

$$\alpha = \frac{\text{Area}(QBC)}{\text{Area}(ABC)} \quad \beta = \frac{\text{Area}(AQC)}{\text{Area}(ABC)} \quad \gamma = \frac{\text{Area}(ABQ)}{\text{Area}(ABC)}$$

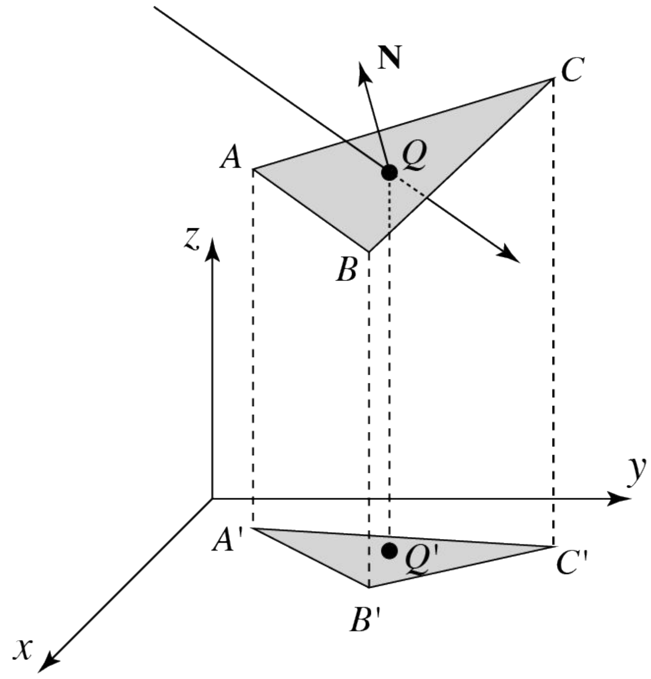


How can cross products help here?

In the end, these calculations can be performed in the 2D projection as well!

Improvement: project down to 2D first

Without loss of generality, we can make this determination after projecting down a dimension:



If *Q'* is inside of *A'B'C'*, then *Q* is inside of *ABC*.

Why is this projection desirable?

Which axis should you “project away”? (Hint: consider the triangle normal.)

Interpolating vertex properties

The barycentric coordinates can also be used to interpolate vertex properties such as:

- material properties
- texture coordinates
- normals

For example:

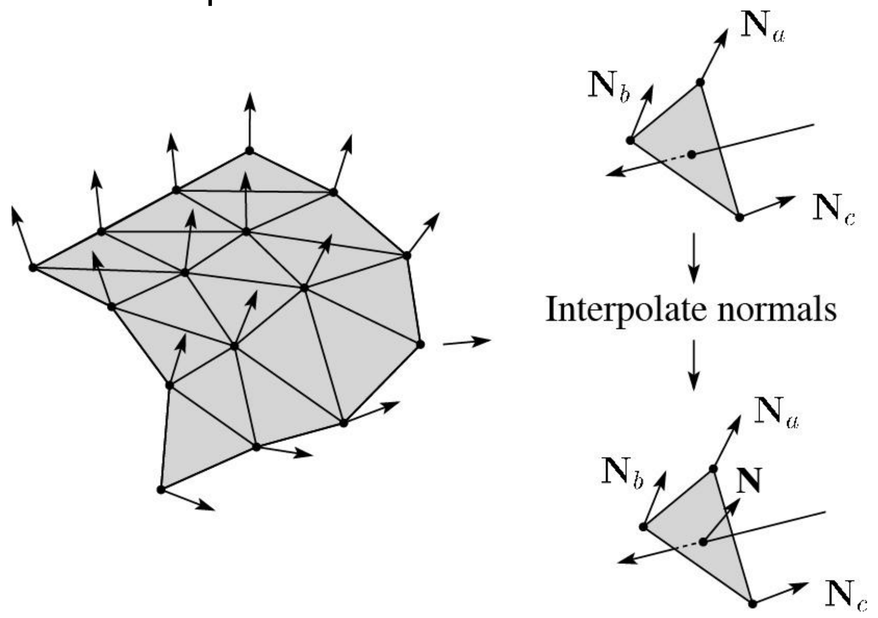
$$k_d(Q) = \alpha k_d(A) + \beta k_d(B) + \gamma k_d(C)$$

Phong interpolated normals

Recall the idea of interpolating normal from the shading lecture, now updated to allow reflection and refraction.

Here's how it works:

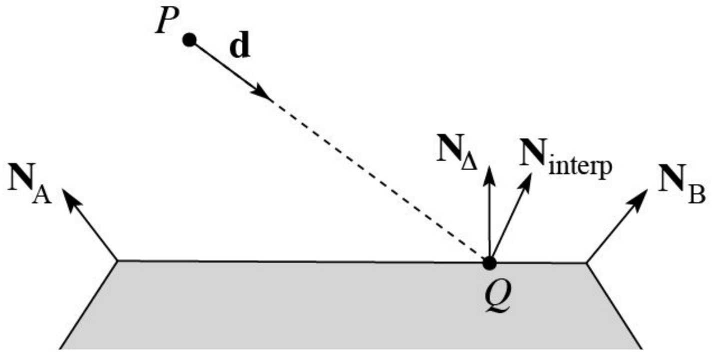
1. Compute normals at the vertices.
2. Interpolate normals and normalize.
3. Shade, reflect, and refract using the interpolated normals.



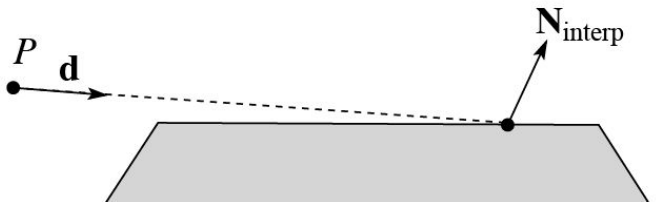
Q: How do we interpolate $\mathbf{N}_a$, $\mathbf{N}_b$, $\mathbf{N}_c$ to get $\mathbf{N}$?

Interpolated normal in a ray tracer

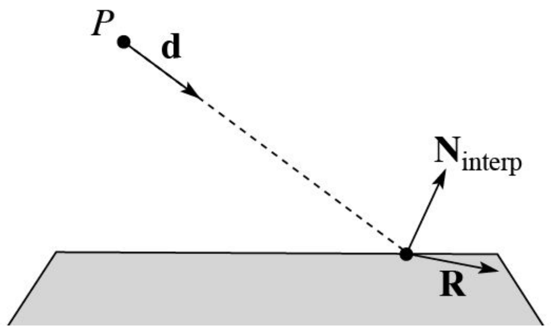
As before, we will use the interpolated normal for shading, but a problem can arise when using this normal for other ray tracing purposes. Consider:



We see that the interpolated normal $\mathbf{N}_{interp}$ is of course different from the true geometric (triangle) normal $\mathbf{N}_\Delta$. Here are a couple problems that can arise:



$\mathbf{N}_{interp} \bullet (-d) < 0$ -> exiting object (wrong!)



Reflected ray enters the object (wrong!)

A similar problem can arise for refraction, in which the refracted direction $\mathbf{T}$ is exiting the object, which is also wrong. There is no "right" answer for handling the discrepancy between normal and

Interpolated normal in a ray tracer, cont'd

We could play it "safe" and always use $\mathbf{N}_\Delta$, but then we will not get nice curved reflections and refractions even when these rays are valid. For Trace, do the following…

Determining when entering/exiting object:

- Use the geometric/true normal ($\mathbf{N}_\Delta$) when deciding whether you are entering/exiting an object.

Shading:

- Use $\mathbf{N}_{\text{interp}}$ for shading.

Reflection:

1. Start by using $\mathbf{N}_{\text{interp}}$ to compute reflection direction $\mathbf{R}$.
2. If $\mathbf{R}$ is (incorrectly) entering the object, then re-compute $\mathbf{R}$ using $\mathbf{N}_\Delta$.

Refraction:

3. Start by using $\mathbf{N}_{\text{interp}}$ to check for Total Internal Reflection (TIR).
4. If TIR, then do not cast a refracted ray.
5. Else, use $\mathbf{N}_{\text{interp}}$ to compute refraction direction $\mathbf{T}$.
6. If $\mathbf{T}$ is (incorrectly) exiting the object, then use $\mathbf{N}_\Delta$

Epsilons

Due to finite precision arithmetic, we do not always get the exact intersection at a surface.

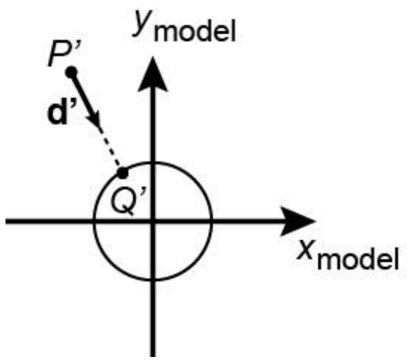
Q: What kinds of problems might this cause?

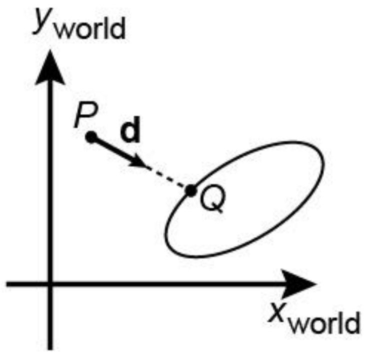
Q: How might we resolve this?

Intersecting with xformed geometry

In general, objects will be placed using transformations. What if the object being intersected were transformed by a matrix M?

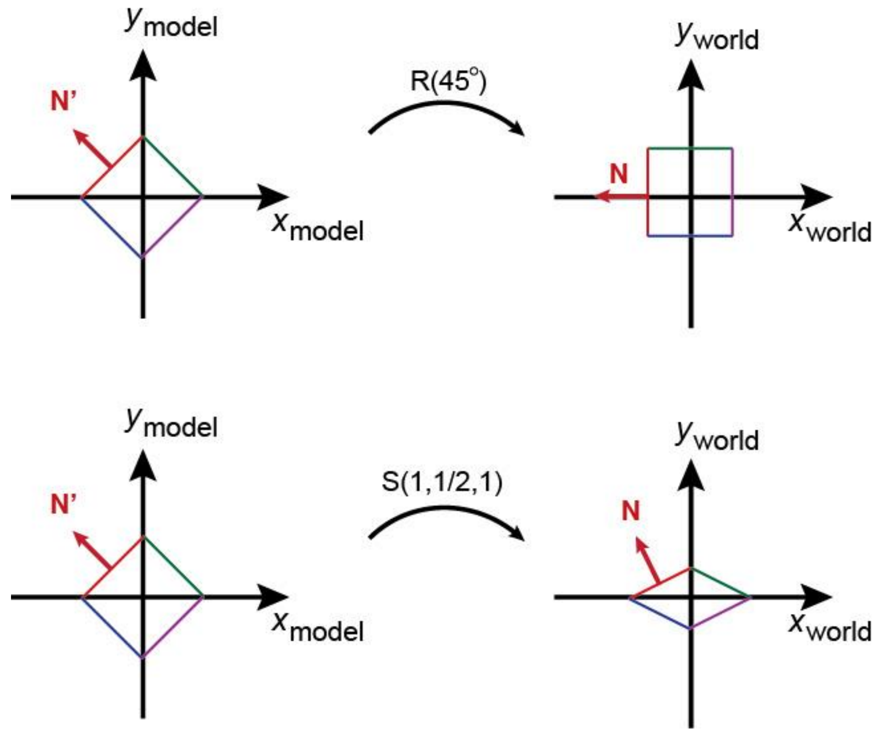
Apply M^{-1} to the ray first and intersect in object (local) coordinates! **Note**: *do not normalize* $\mathbf{d}'$!





Intersecting with xformed geometry

The intersected normal is in object (local) coordinates. How do we transform it to world coordinates?



Summary

What to take home from this lecture:

- The meanings of all the boldfaced terms.
- Enough to implement basic recursive ray tracing.
- How reflection and transmission directions are computed.
- How ray-object intersection tests are performed on spheres, planes, and triangles
- How barycentric coordinates within triangles are computed
- How ray epsilons are used.

Accelerated ray tracing

Reading

Required:

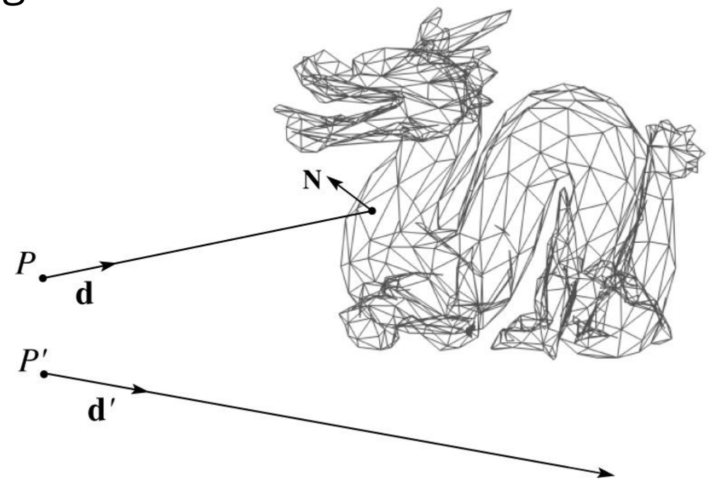
- Marschner and Shirley, Sections 12.3 (online handout)

Further reading:

- A. Glassner. An Introduction to Ray Tracing. Academic Press, 1989.

Faster ray-polyhedron intersection

Let's say you were intersecting a ray with a triangle mesh:



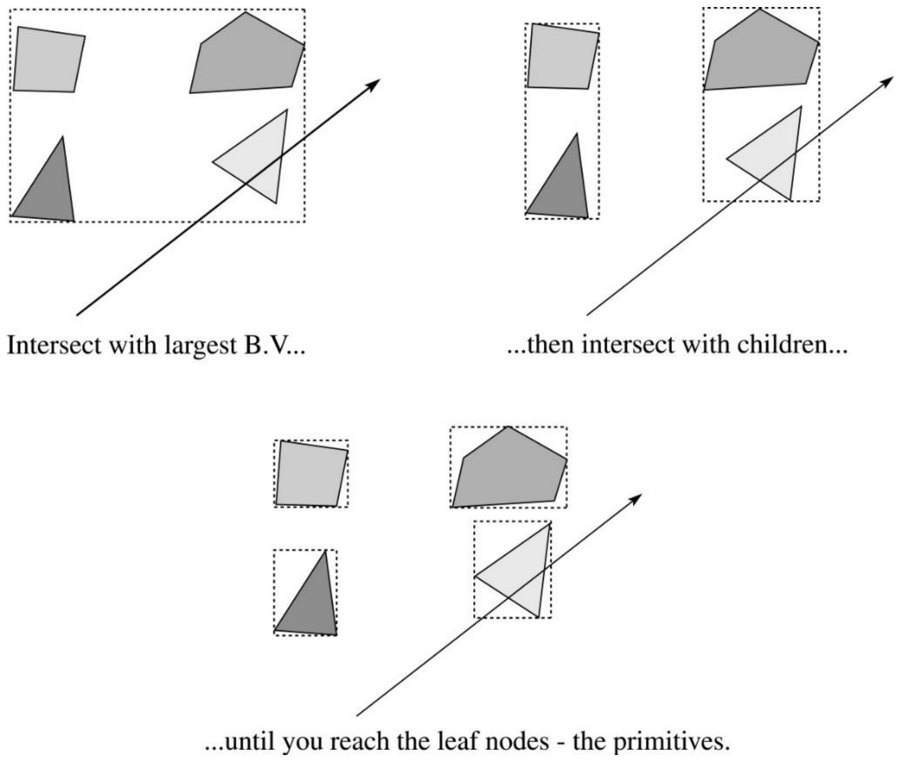
Straightforward method

- intersect the ray with each triangle
- return the intersection with the smallest *t*-value.

Q: How might you speed this up?

Bounding Volume Hierarchies (BVHs)

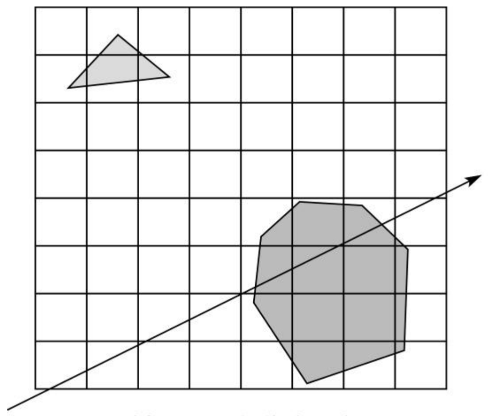
We can generalize the idea of bounding volume acceleration with **bounding volume hierarchies (BVHs)**.

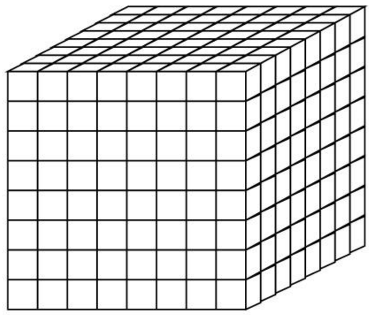

Intersect with largest B.V...
...then intersect with children...
...until you reach the leaf nodes - the primitives.

Key: build balanced trees with *tight bounding volumes*.

Uniform spatial subdivision

Another approach is **uniform spatial subdivision**.


Uniform subdivion in 2D


Uniform subdivion in 3D

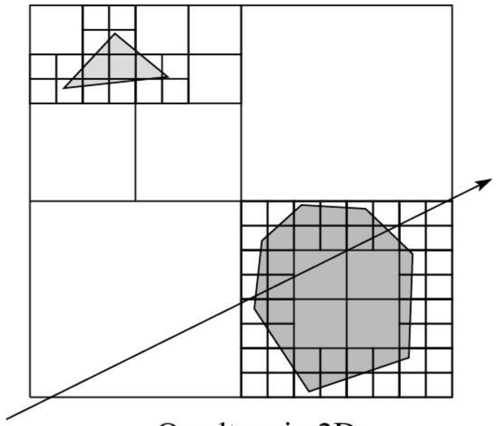
<u>Idea</u>:

- Partition space into cells (voxels)
- Associate each primitive with the cells it overlaps
- Trace ray through voxel array using fast incremental arithmetic to step from cell to cell

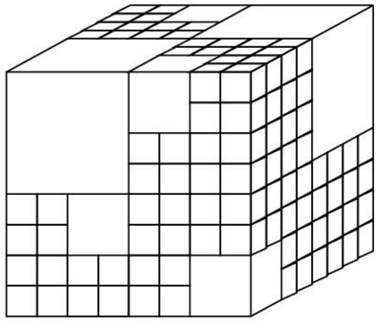
Q: Given a10^6 triangle football stadium with a 10^6 triangle teapot on one of the seats, would

Non-uniform spatial subdivision: octrees

Another approach is **non-uniform spatial subdivision**. One version of this is octrees:



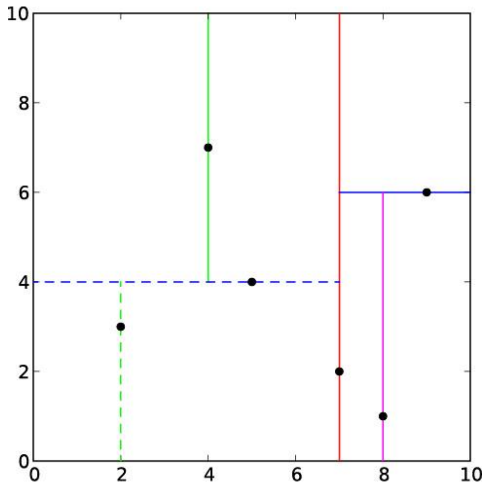
Quadtree in 2D



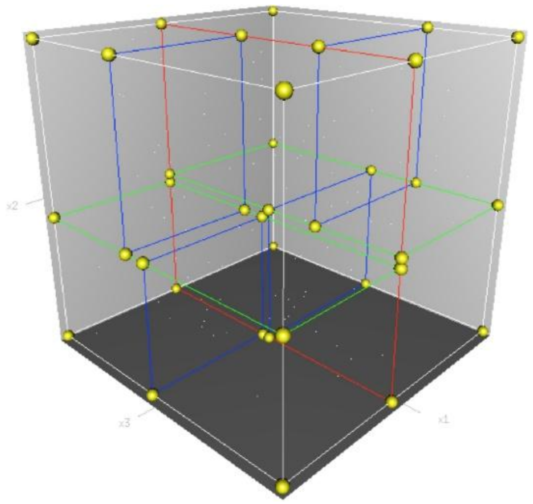
Octree in 3D

Non-uniform spatial subdivision: *k*-d trees

Another non-uniform subdivision is *k*-d (*k* –dimensional) trees:



k-d tree (*d* = 2)



k-d tree (*d* = 3)

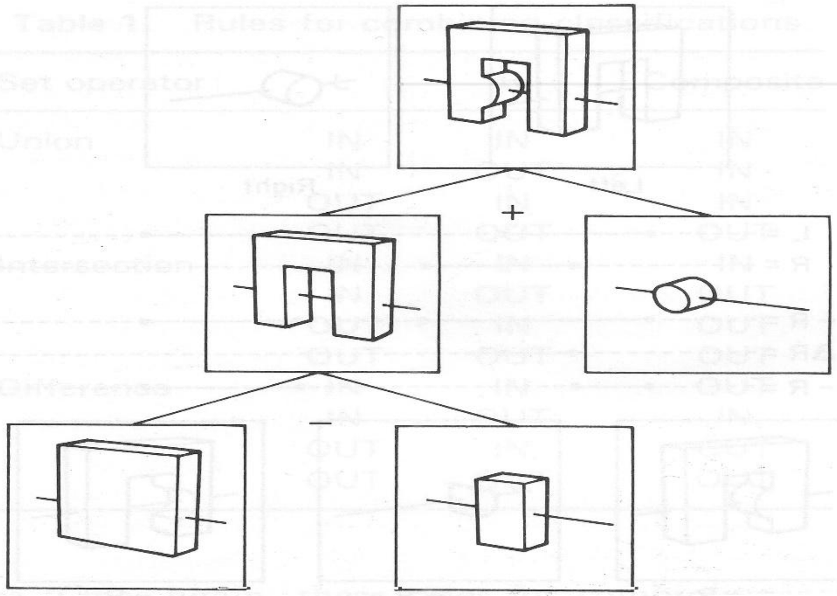
If the planes can be non-axis aligned, then you get BSP (binary space partitioning) trees.

Various combinations of these ray intersections techniques are also possible.

[Image credits: Wikipedia.]

CSG

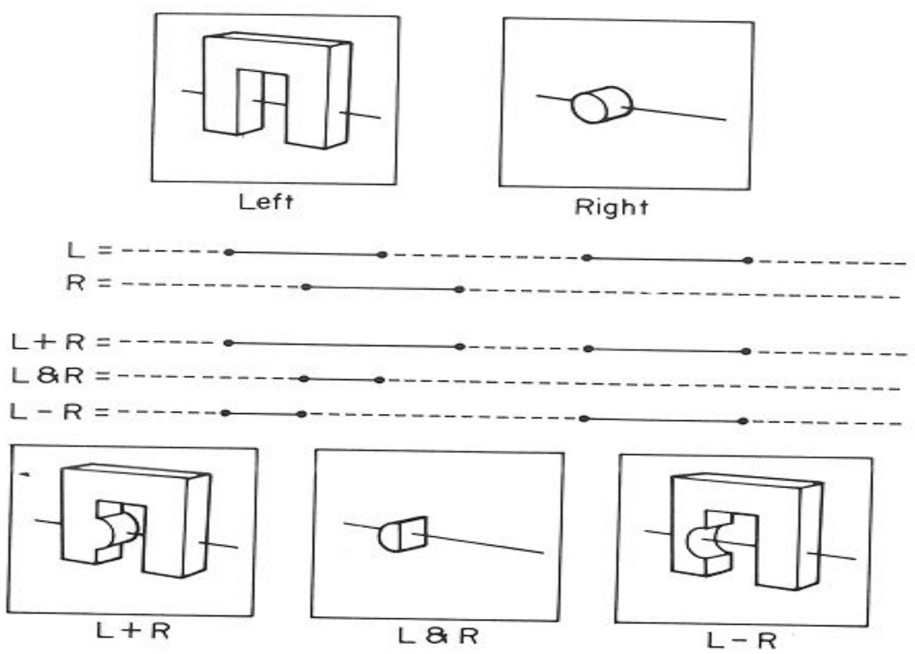
CSG (constructive solid geometry) is an incredibly powerful way to create complex scenes from simple primitives.



CSG Implementation

CSG intersections can be analyzed using “Roth diagrams”.

- Maintain description of *all intersections* of ray with primitive
- Functions to combine Roth diagrams under CSG operations

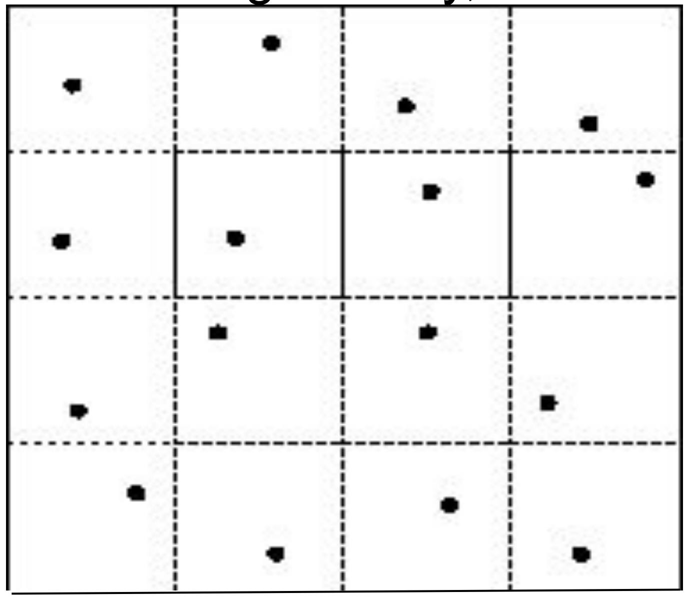


58

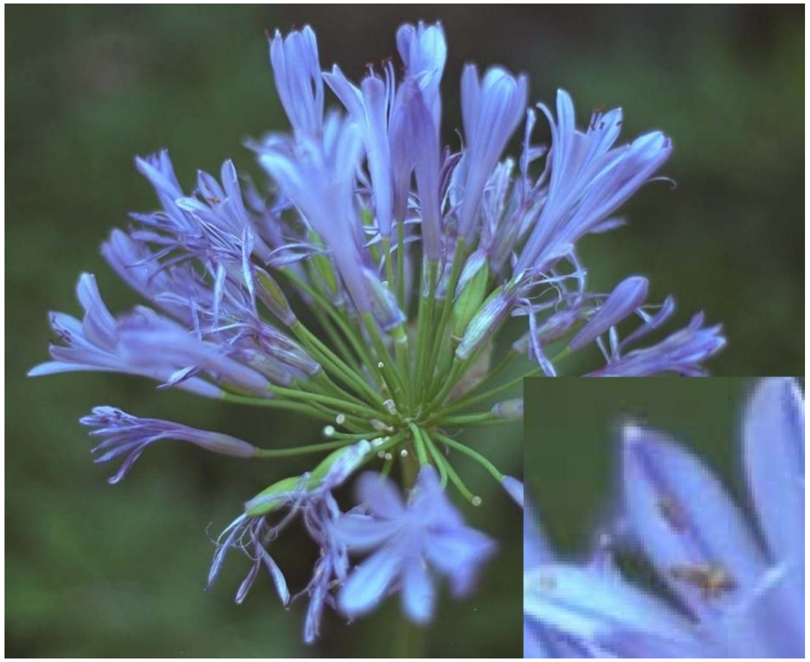
Distribution Ray Tracing

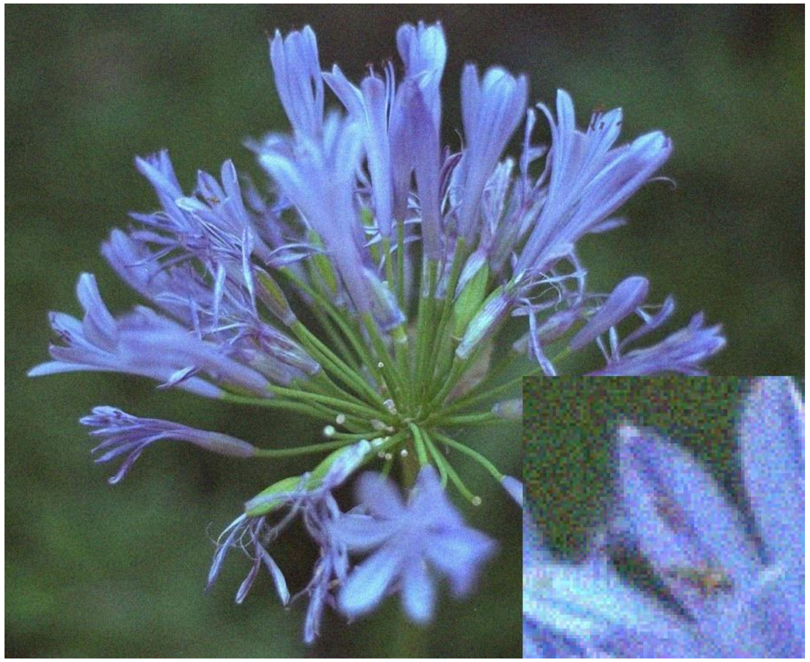
Usually known as “distributed ray tracing”, but it has nothing to do with distributed computing

General idea: instead of firing one ray, fire multiple rays in a jittered grid



Noise





Noise can be thought of as randomness added to the signal.

The eye is relatively insensitive to noise.

DRT pseudocode

traceImage() looks basically the same, except now each pixel records the average color of jittered sub-pixel rays.

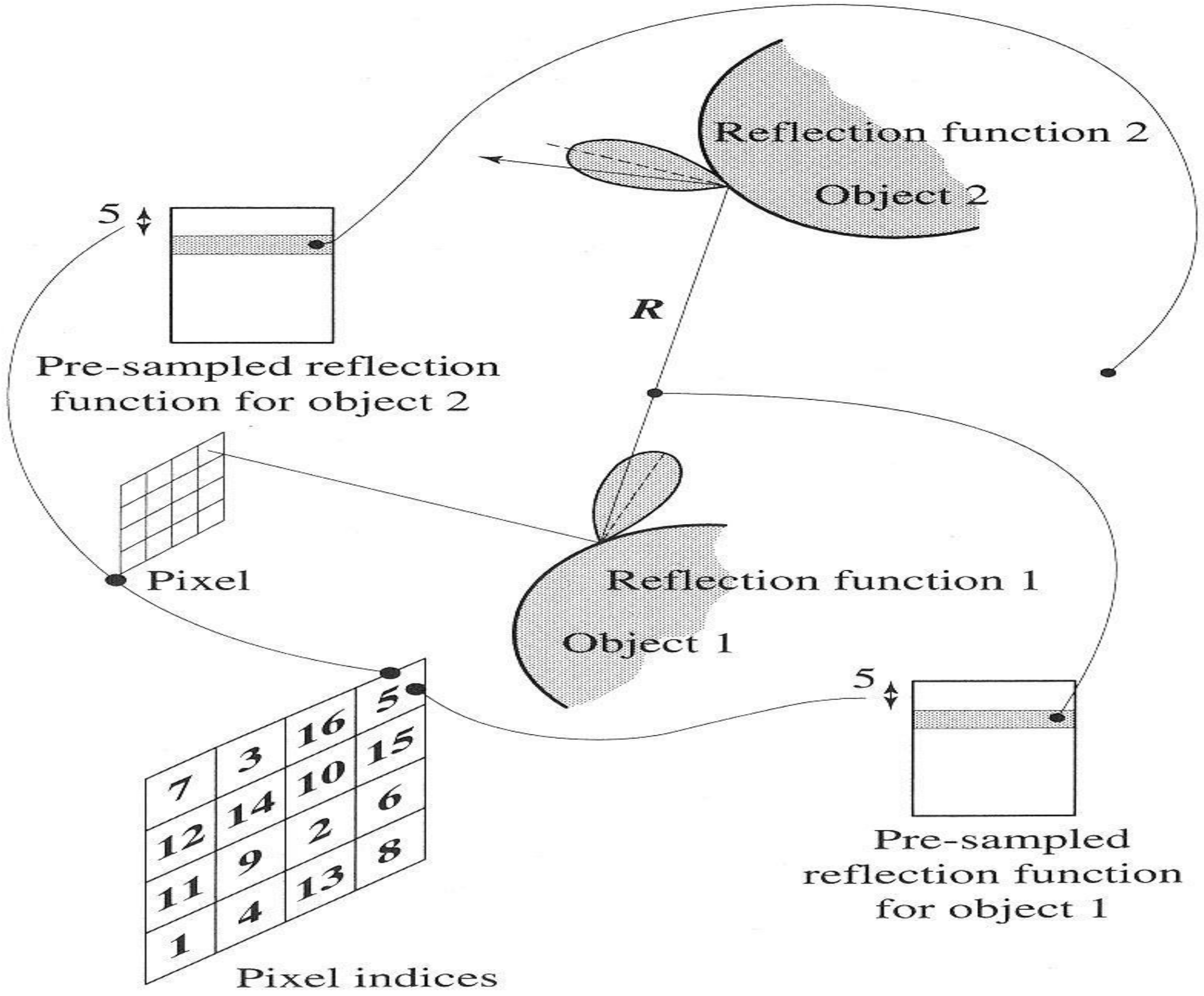
```
function traceImage (scene):
    for each pixel (i, j) in image do
        I(i, j) ← 0
        for each sub-pixel id in (i,j) do
            s ← pixelToWorld(jitter(i, j, id))
            p ← COP
            u ←(s - p).normalize()
            I(i, j) ← I(i, j) + traceRay(scene, p, u, id)
        end for
        I(i, j)  I(i, j)/numSubPixels
    end for
end function
```

DRT pseudocode (cont'd)

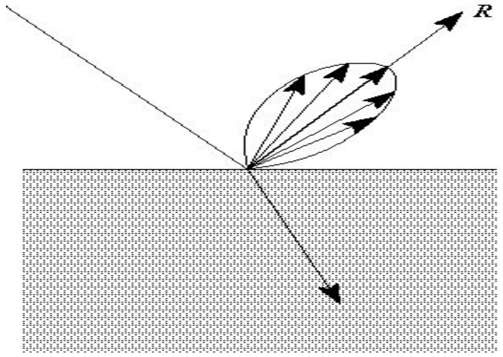
Now consider *traceRay*(), modified to handle (only) opaque glossy surfaces:

```
function traceRay(scene, p, u, id):
    (q, N, obj) ← intersect (scene, p, u)
    I ← shade(…)
    R ← jitteredReflectDirection(N, -u, id)
    I ← I + obj.$k_r$ * traceRay(scene, q, R, id)
    return I
end function
```

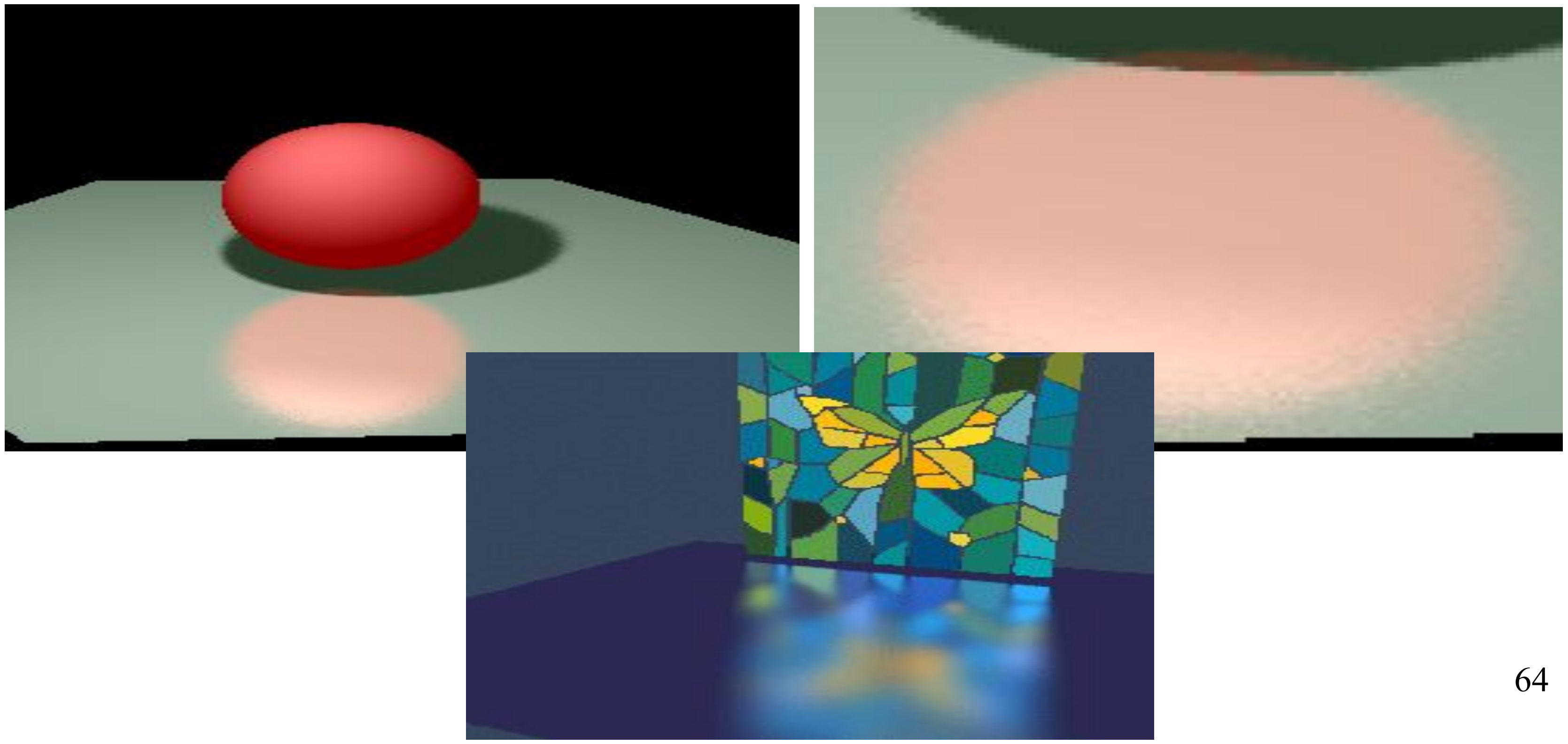
Pre-sampling glossy reflections



Distributing Reflections

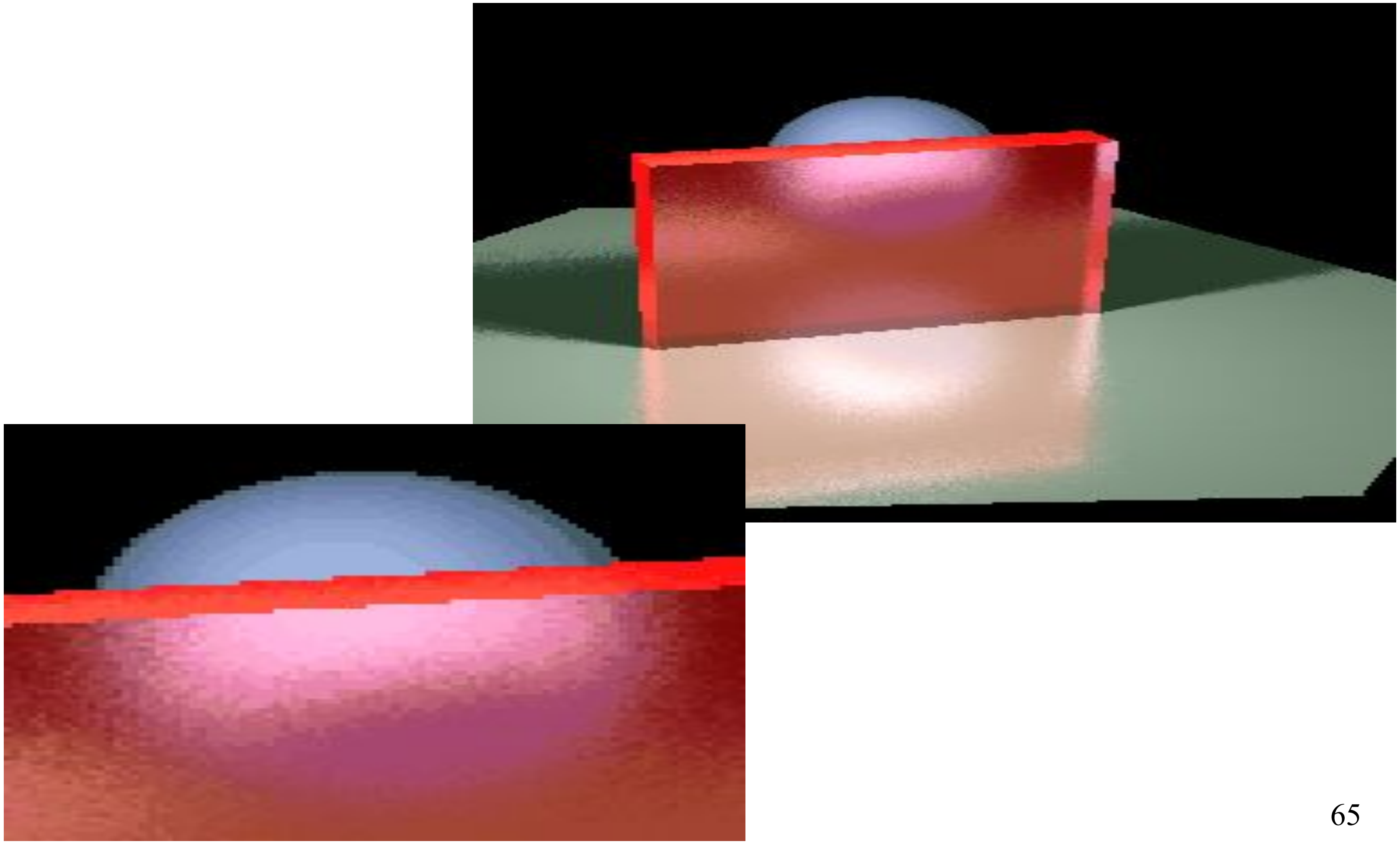


Distributing rays over reflection direction gives:



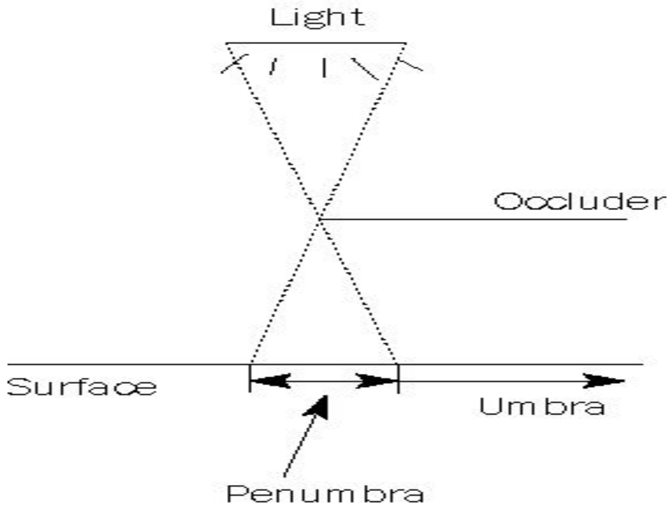
Distributing Refractions

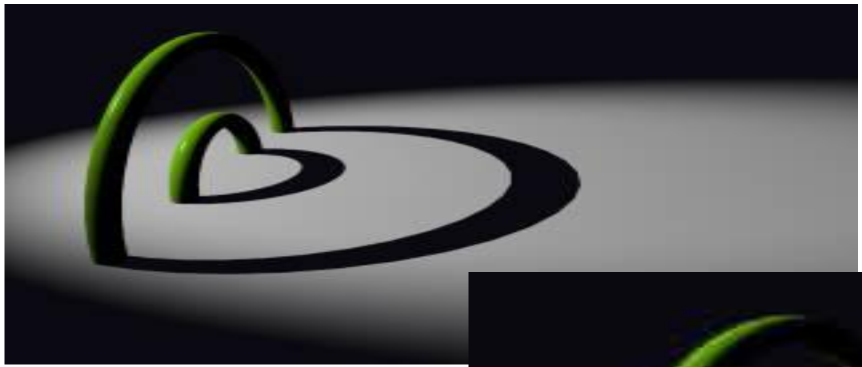
Distributing rays over transmission direction gives:

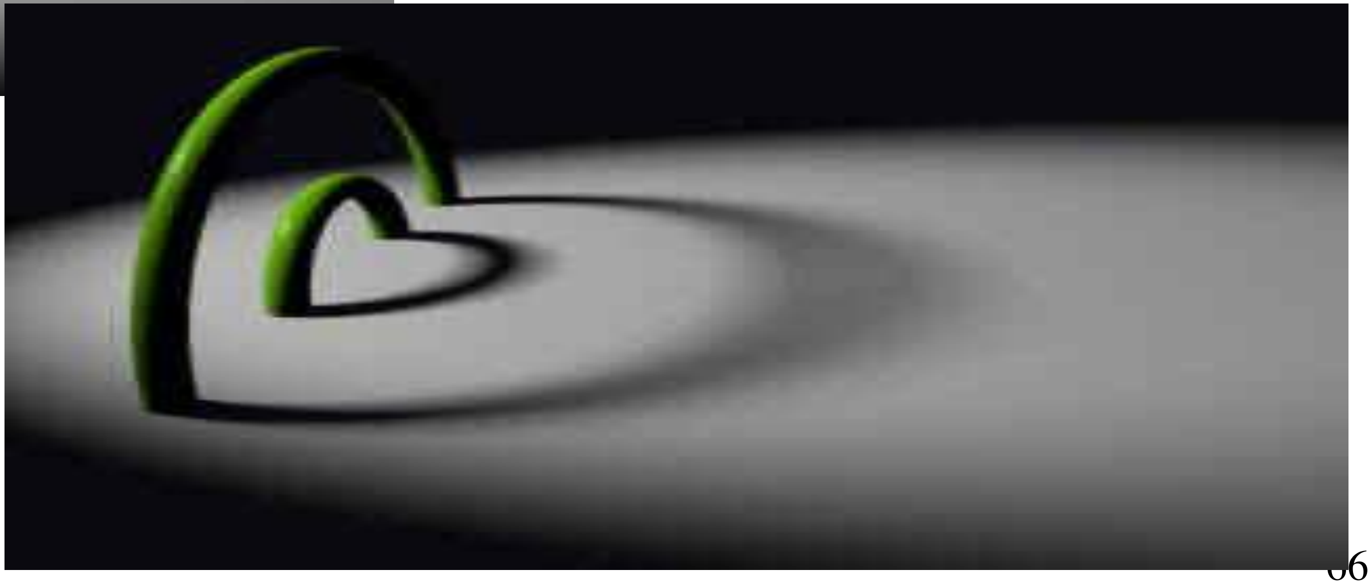


Distributing Over Light Area

Distributing over light area gives:

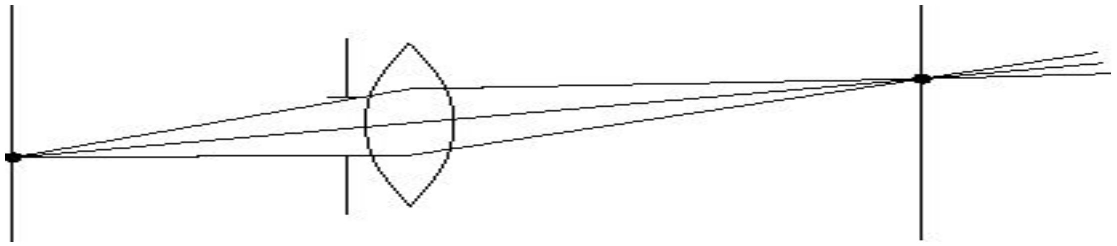






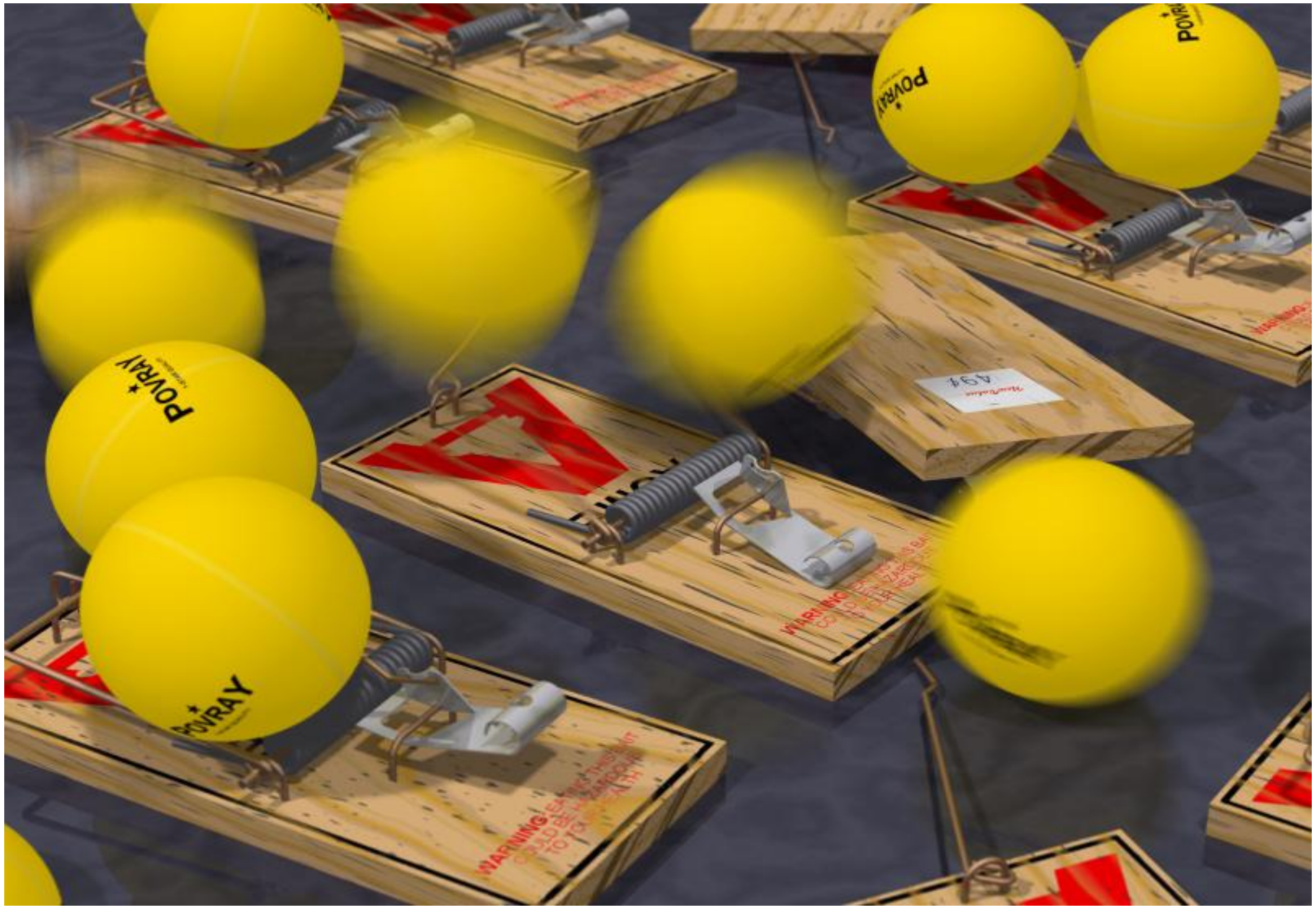
Distributing Over Aperature

We can fake distribution through a lens by choosing a point on a finite aperature and tracing through the "in-focus point".

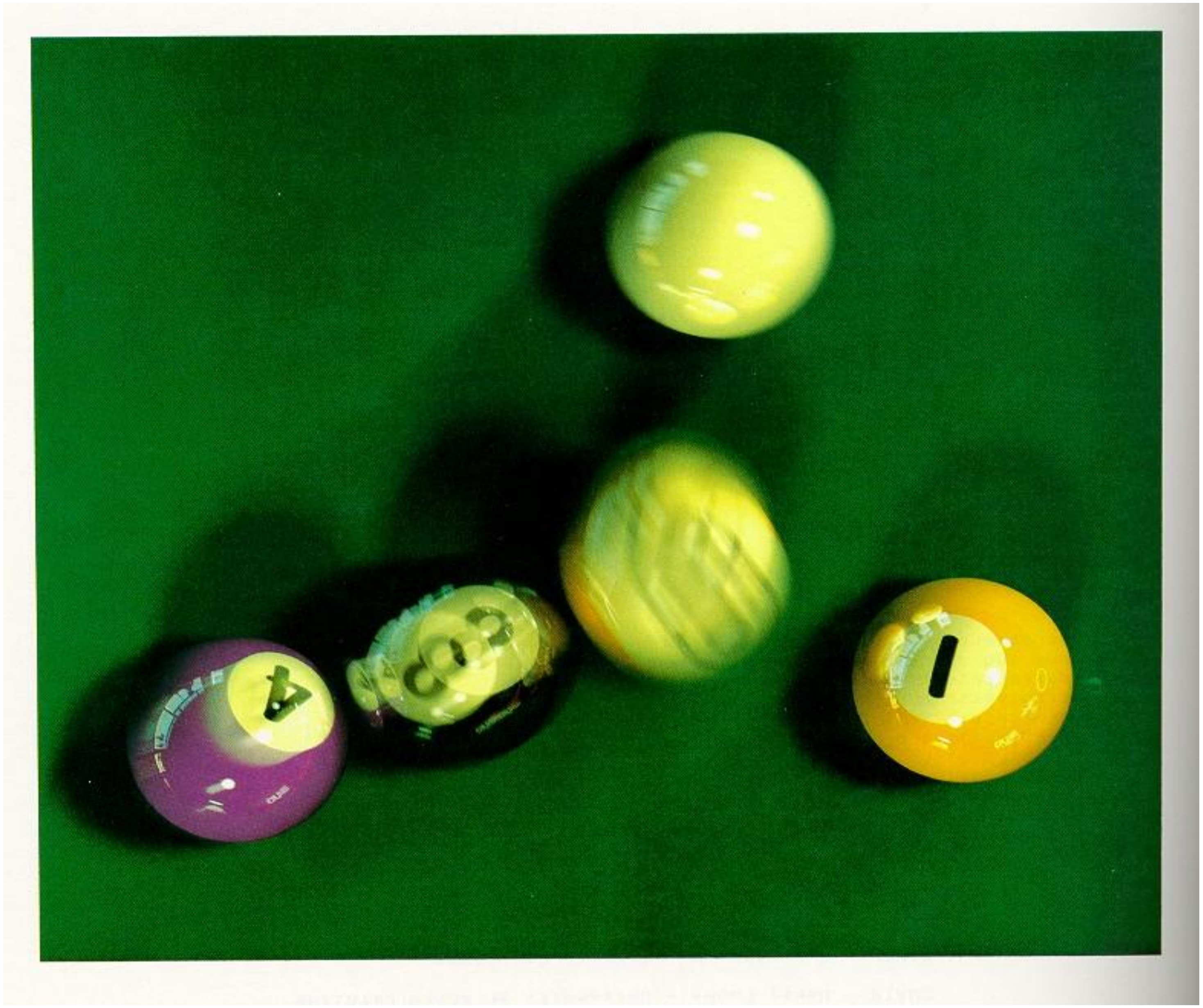


Distributing Over Time

We can endow models with velocity vectors and distribute rays over *time*. this gives:

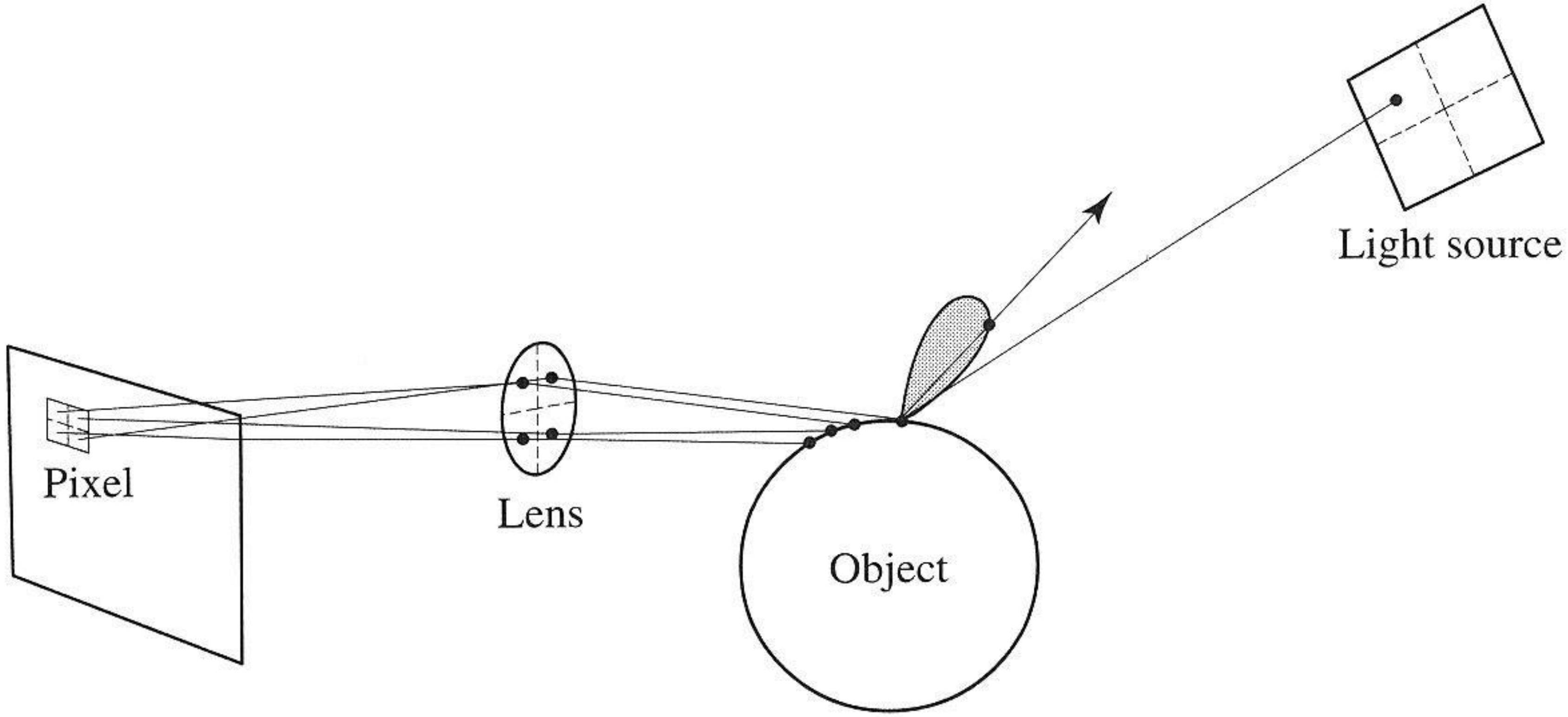


68



Chaining the ray id's

In general, you can trace rays through a scene and keep track of their id's to handle *all* of these effects:



Summary

What to take home from this lecture:

- An intuition for how ray tracers can be accelerated.