Ray Tracing

Reading

Foley *et al.*, 16.12

Optional:

- Glassner, An introduction to Ray Tracing, Academic Press, Chapter 1.
- T. Whitted. "An improved illumination model for shaded display". *Communications of the ACM* 23(6), 343-349, 1980.

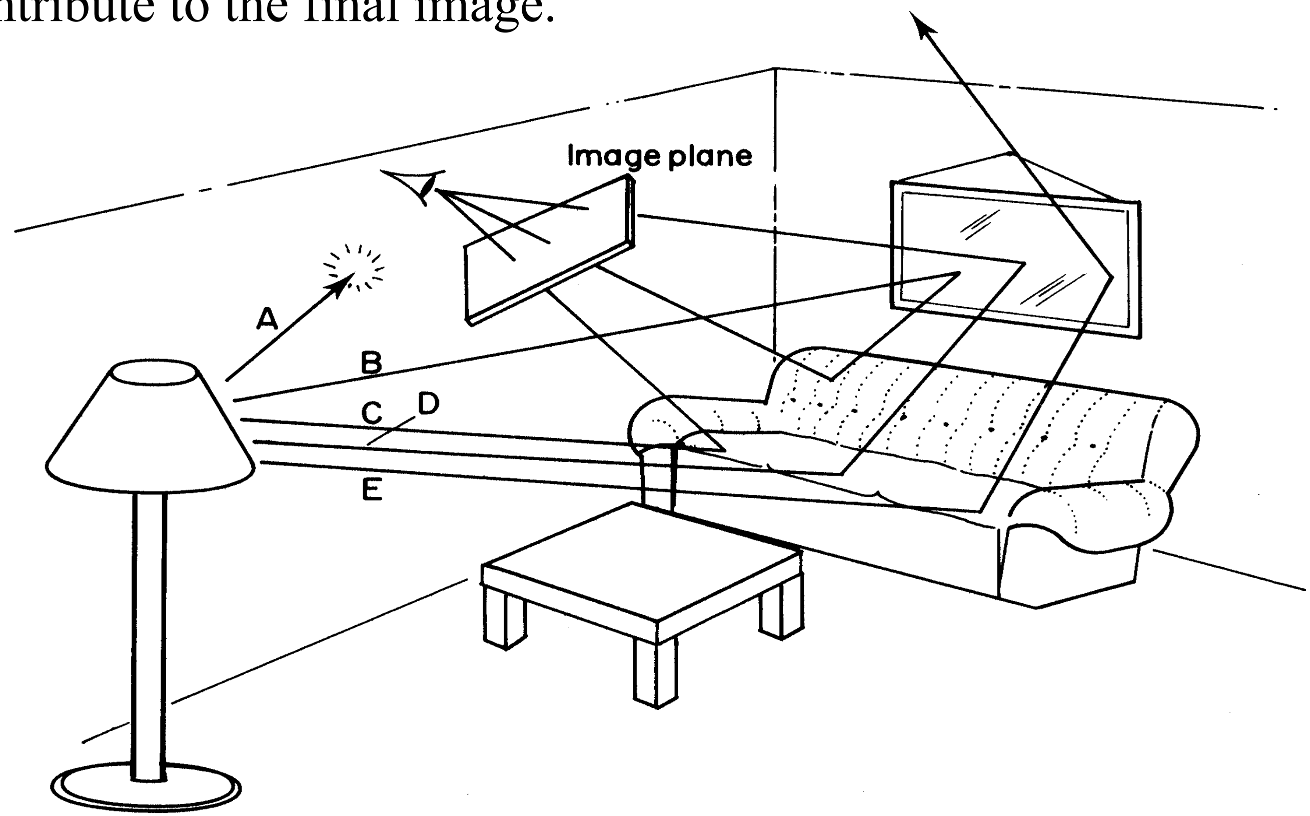
Geometric optics

We will take the view 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).

Forward Ray Tracing

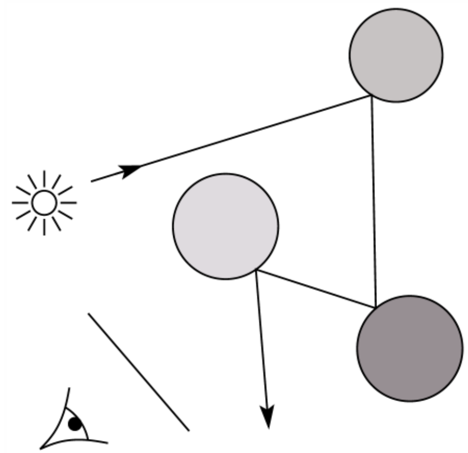
- Rays emanate from light sources and bounce around in the scene.
- Rays that pass through the projection plane and enter the eye contribute to the final image.



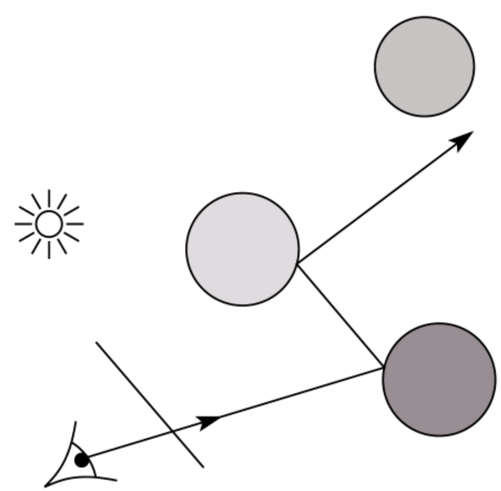
- What's wrong with this method?

Eye vs. Light

- Starting at the light (a.k.a. forward ray tracing, photon tracing)

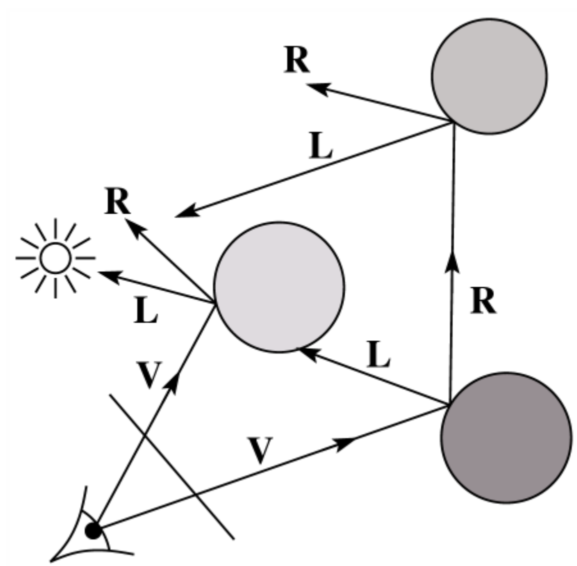


- Starting at the eye (a.k.a. backward ray tracing)



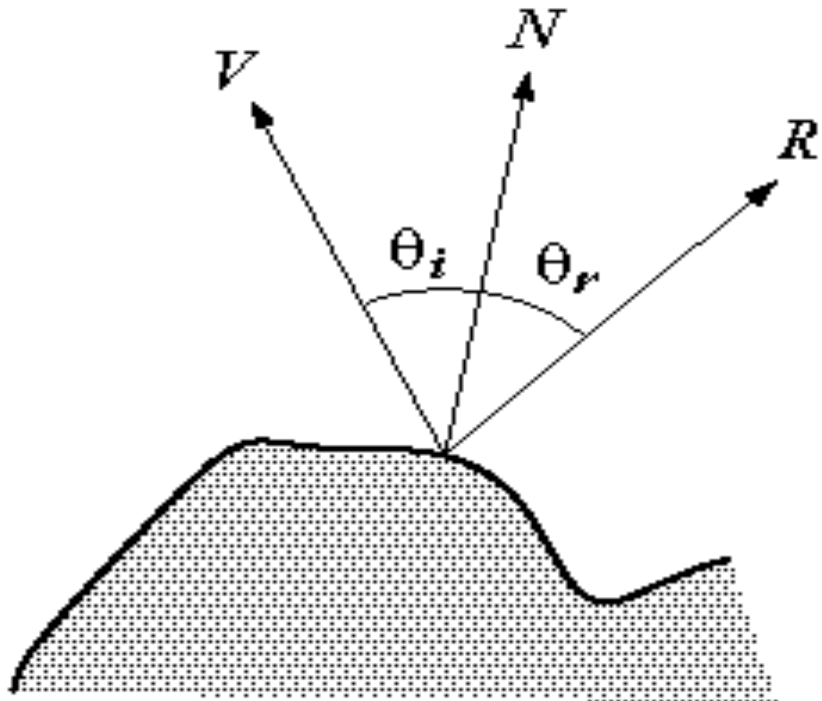
Whitted ray-tracing algorithm

1. For each pixel, trace a **primary ray** to the first visible surface
2. For each intersection trace **secondary rays**:
   - **Shadow rays** in directions L_i to light sources
   - **Reflected ray** in direction R
   - **Refracted ray** (**transmitted ray**) in direction T



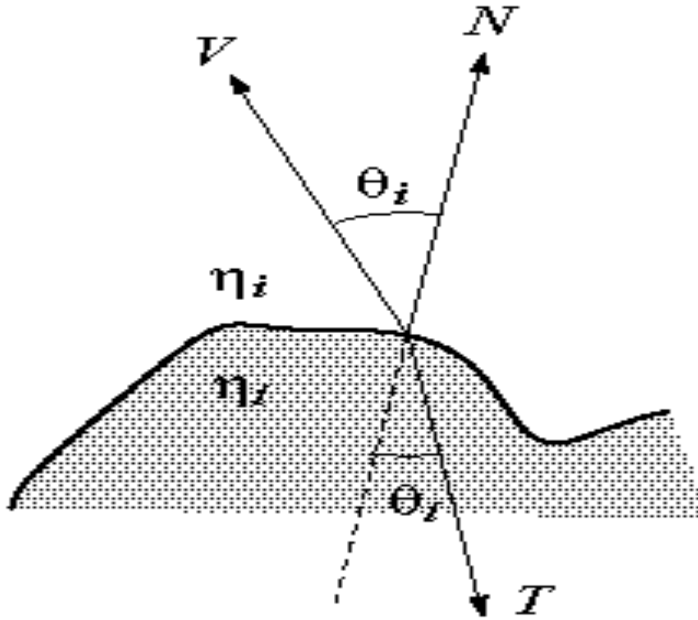
Reflection

- Reflected light from objects behaves like specular reflection from light sources
  - Reflectivity is just specular color
  - Reflected light comes from direction of perfect specular reflection



- Is this model reasonable?

Refraction

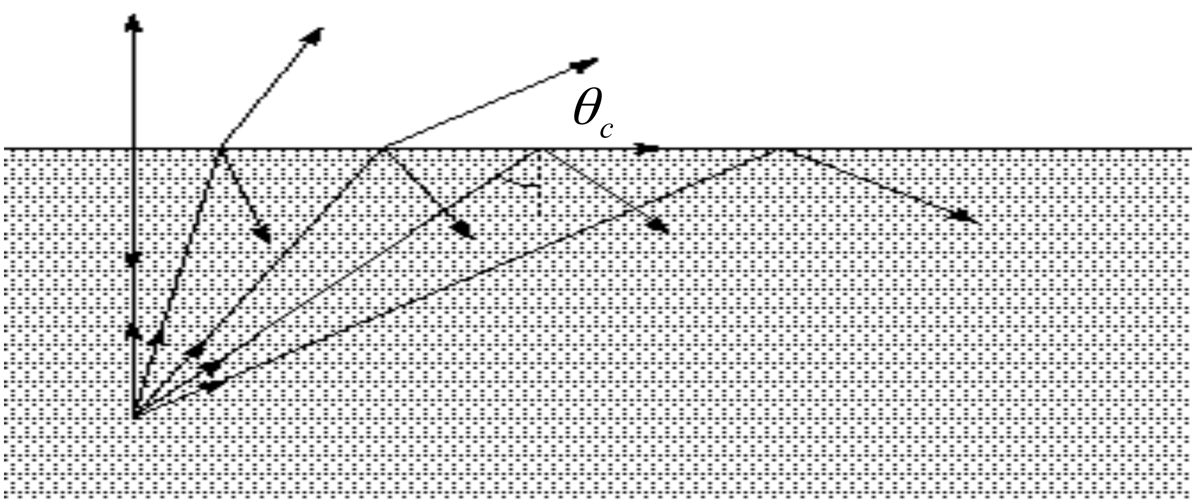


- Amount to transmit determined by transparency coefficient, which we store explicitly
- *T* comes from Snell's law

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

Total Internal Reflection

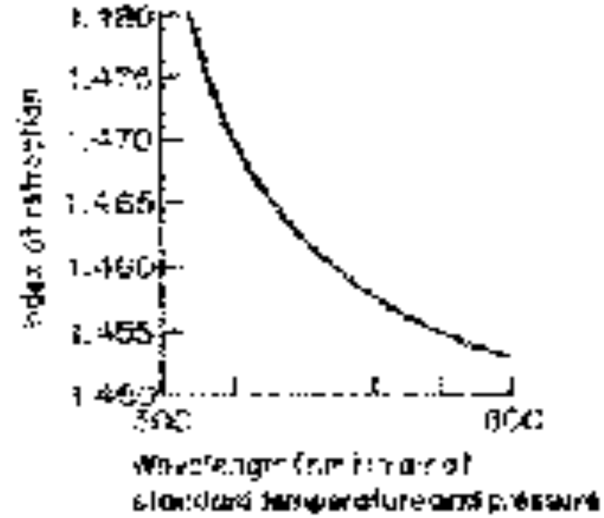
- When passing from a dense medium to a less dense medium, light is bent further away from the surface normal
- Eventually, it can bend right past the surface!
- The θ_i that causes θ_t to exceed 90 degrees is called the **critical angle (θ_c)**. For θ_i greater than the critical angle, no light is transmitted.
- A check for TIR falls out of the construction of T



Index of Refraction

- Real-world index of refraction is a complicated physical property of the material

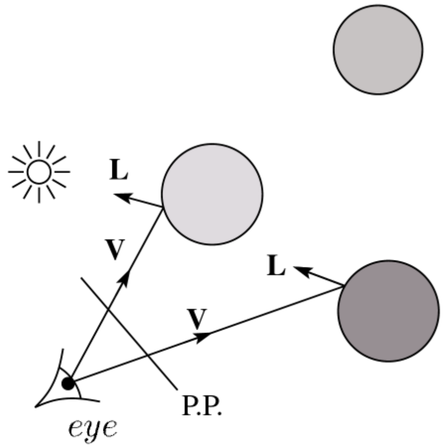
Medium	Index of refraction
Vaccum	1
Air	1.0003
Water	1.33
Fused quartz	1.46
Glass, crown	1.52
Glass, dense flint	1.66
Diamond	2.42



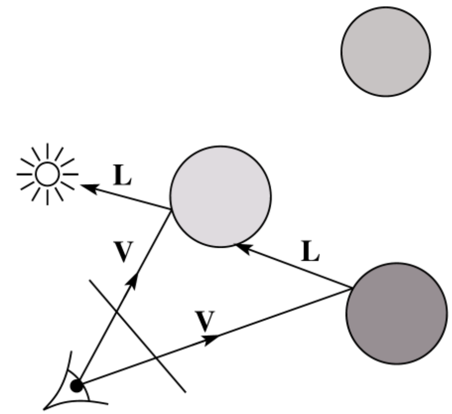
Index of refraction variation for fused quartz

- IOR also varies with wavelength, and even temperature!
- How can we account for wavelength dependence when ray tracing?

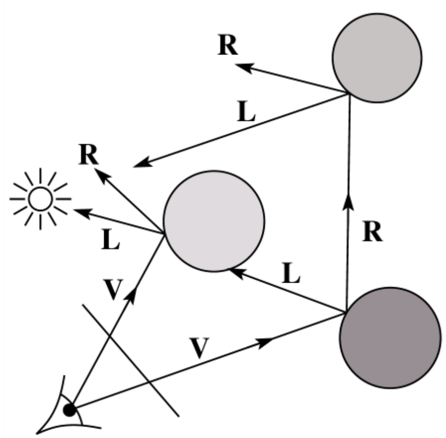
Stages of Whitted ray-tracing



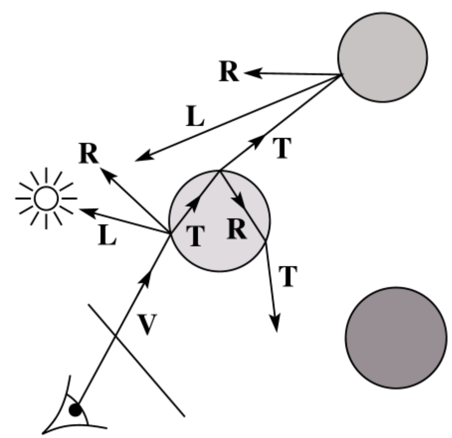
Primary rays



Shadow rays

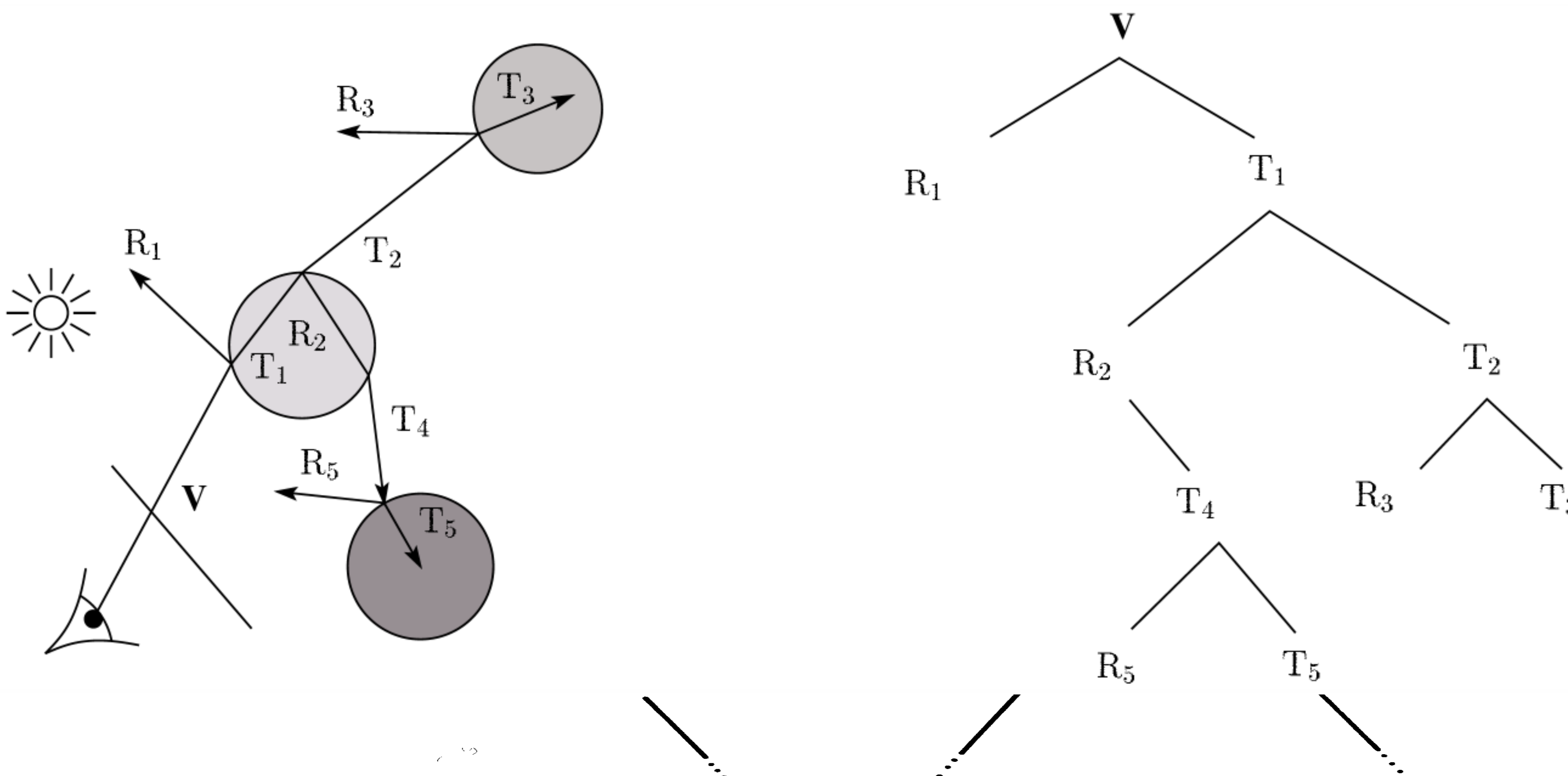


Reection rays



Refracted rays

The Ray Tree



Shading

If I(P_0, **u**) is the intensity seen from point P_0 along direction **u**

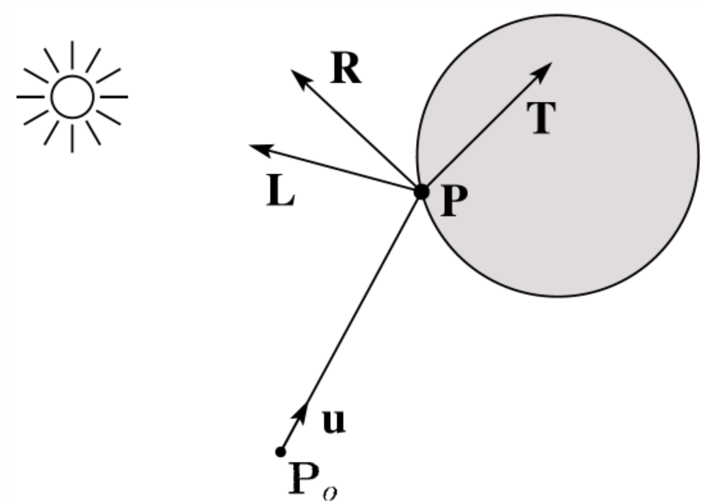
$$I(P_0, \mathbf{u}) = I_{direct} + I_{reflected} + I_{transmitted}$$

where

I_{direct} = Shade(**N**, **L**, **u**, **R**) (e.g. Phong shading model)

$$I_{reflected} = k_r I(P, \mathbf{R})$$

$$I_{transmitted} = k_t I(P, \mathbf{T})$$



Typically, we set $k_r = k_s$ and

$k_t = 1 - k_s$.

Parts of a Ray Tracer

- What major components make up the core of a ray tracer?
  - Outer loop sends primary rays into the scene
  - Trace arbitrary ray and compute its color contribution as it travels through the scene
  - Shading model

$$I = k_e + k_a I_a + \sum_i f(d_i) I_{li} \left[k_d (\mathbf{N} \cdot \mathbf{L}_i)_+ + k_s (\mathbf{V} \cdot \mathbf{R})_+^{n_s} \right]$$

Outer Loop

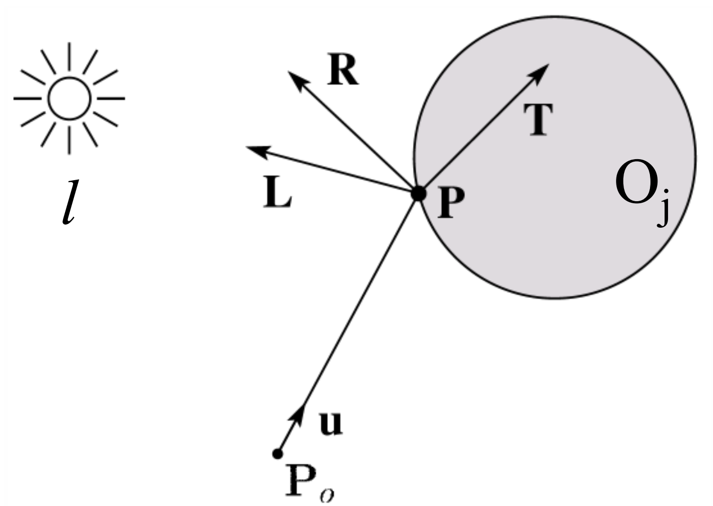
```
void traceImage (scene)
{
        for each pixel (i,j) in the image {

                p = pixelToWorld(i,j)
                c = COP
                u = (p - c)/||p – c||
                I(i,j) = traceRay (scene, c, u)

        }
}
```

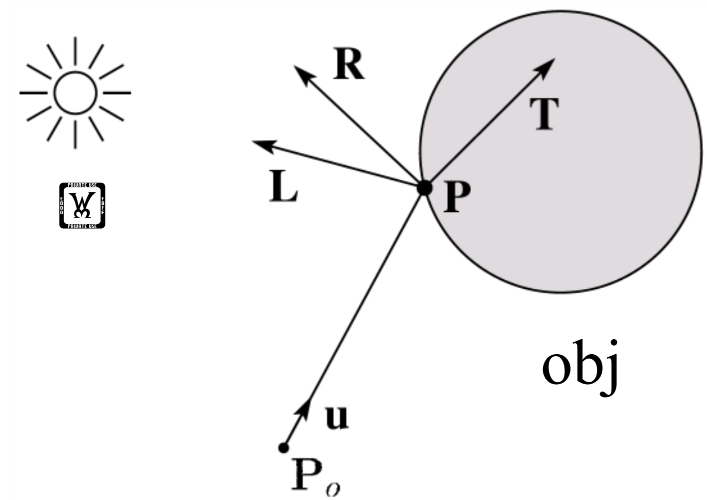
Trace Pseudocode

```
color traceRay(point P0, direction u )
{
        (P,Oi) = intersect( P0, u);
        I = 0
        for each light source l {
                (P', LightObj) = intersect(P, dir(P,l))
                if LightObj = l {
                        I = I + I(l)
                }
        }
        I = I + Obj.Kr * traceRay(P, R)
        I = I + Obj.Kt * traceRay(P, T)
        return I
}
```



TraceRay Pseudocode

```
function traceRay(scene, P0, u) {
    (t, P, N, obj)  ← scene.intersect (P0, u)
    I = shade( u, N, scene )
    R = reflectDirection( u, N )
    I ← I + obj.kr * traceRay(scene, P, R)
    if ray is entering object {
       (ni, nt) ← (index_of_air, obj.index)
    } else {
        (ni, nt) ← (obj.index, index_of_air)
    }
    if (notTIR ( u, N, ni, nt ) {
        T = refractDirection ( u, N, ni, nt )
        I ← I + obj.kt * traceRay(scene, P, T)
    }
    return I
}
```



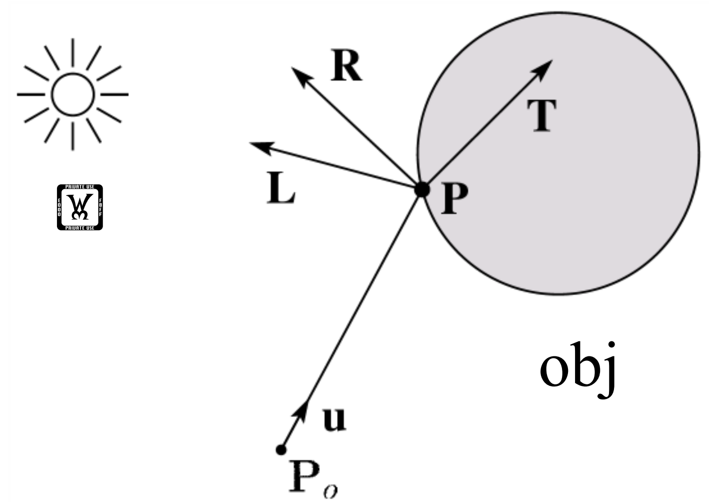
Raytracer Demo

Controlling Tree Depth

- Ideally, we'd spawn child rays at every object intersection forever, getting a "perfect" color for the primary ray.
- In practice, we need heuristics for bounding the depth of the tree (i.e., recursion depth)
- ?

Shading Pseudocode

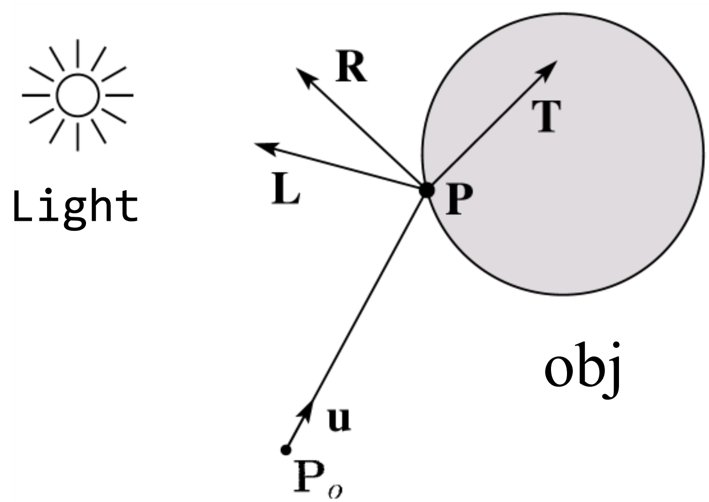
```
function shade(obj, scene, P, N, u) {
   I ← obj.k_e + obj. k_a * scene->I_a
   for each light source L_i {
      atten = distanceAttenuation(L_i, P) *
         shadowAttenuation(L_i, Scene, P)
      I ← I + atten*(diffuse term + spec term)
   }
   return I
}
```



Shadow attenuation pseudocode

Check to see if a ray makes it to the light source.

```
function shadowAttenuation(Li, scene, P) {
   d = (Li.position - P).normalize()
    (t, Pl, N, obj) ← scene.intersect(P, d)
   if Pl is before the light source {
      atten = 0
   } else {
      atten = 1
   }
   return atten
}
```

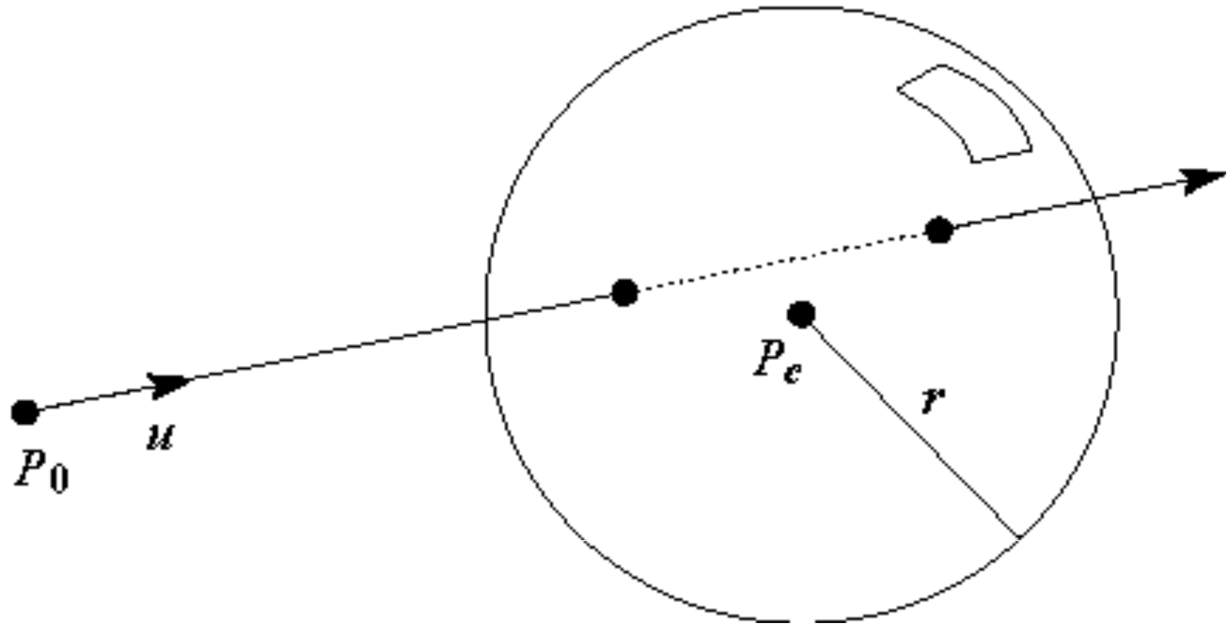


Q: What if there are transparent objects along a path to the light source?

Ray-Object Intersection

- Must define different intersection routine for each primitive
- The bottleneck of the ray tracer, so make it fast!
- Most general formulation: find all roots of a function of one variable
- In practice, many optimized intersection tests exist (see Glassner)

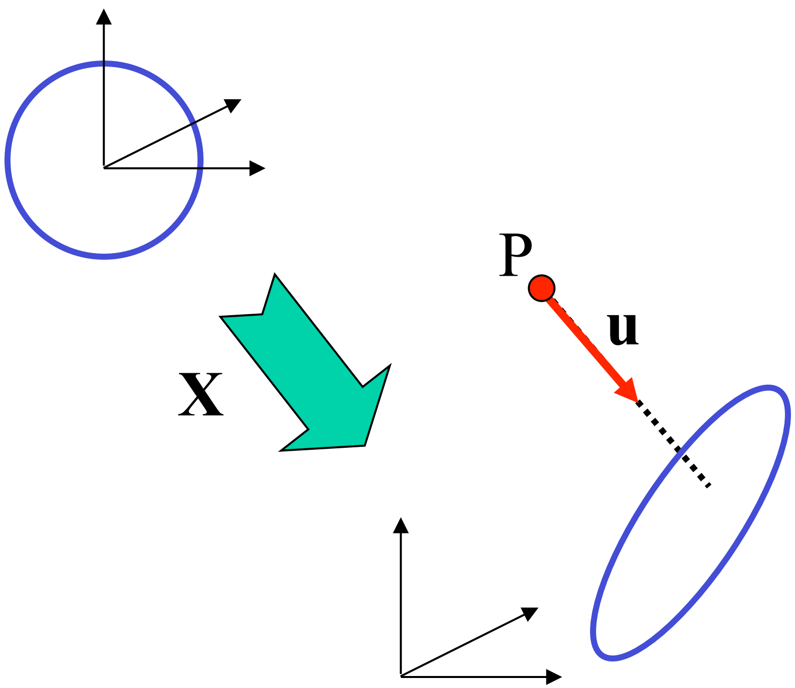
Ray-Sphere Intersection



- Given a sphere centered at P_c =[0,0,0] with radius r and a ray $P(t) = P_0 + t\boldsymbol{u}$, find the intersection(s) of $P(t)$ with the sphere.

Object hierarchies and ray intersection

How do we intersect with primitives transformed with affine transformations?



$$\mathbf{u}' = \begin{bmatrix} u_x \\ u_y \\ u_z \\ 0 \end{bmatrix} \mathbf{X}^{-1}$$

$$P' = \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} \mathbf{X}^{-1}$$

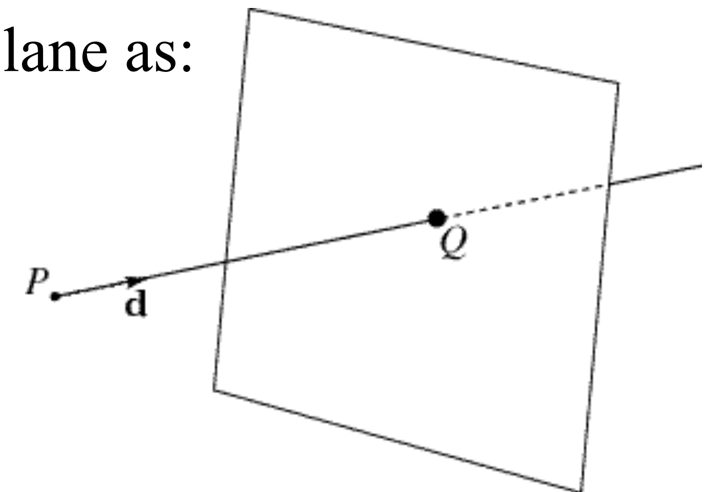
Numerical Error

- Floating-point roundoff can add up in a ray tracer, and create unwanted artifacts
  - Example: intersection point calculated to be ever-so-slightly *inside* the intersecting object. How does this affect child rays?
- Solutions:
  - Perturb child rays
  - Use global ray epsilon

Plane Intersection

- We can write the equation of a plane as:

$$ax + by + cz + d = 0$$

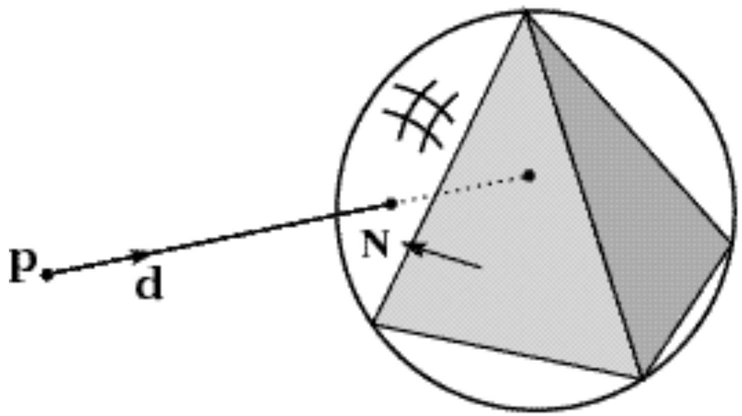


- The coefficients a, b, and c form a vector that is normal to the plane, $\mathbf{n} = [a\ b\ c]^T$. Thus, we can re-write the plane equation as:

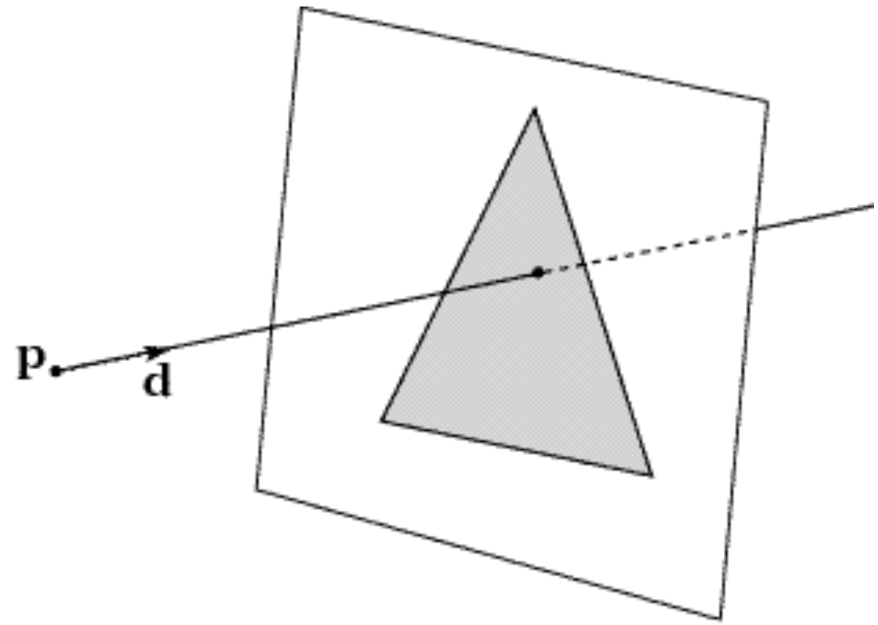
$$\mathbf{n} \cdot (\mathbf{P} + t\mathbf{u}) + d = 0$$

- We can solve for the intersection parameter (and thus the point):

Ray-Polymesh Intersection



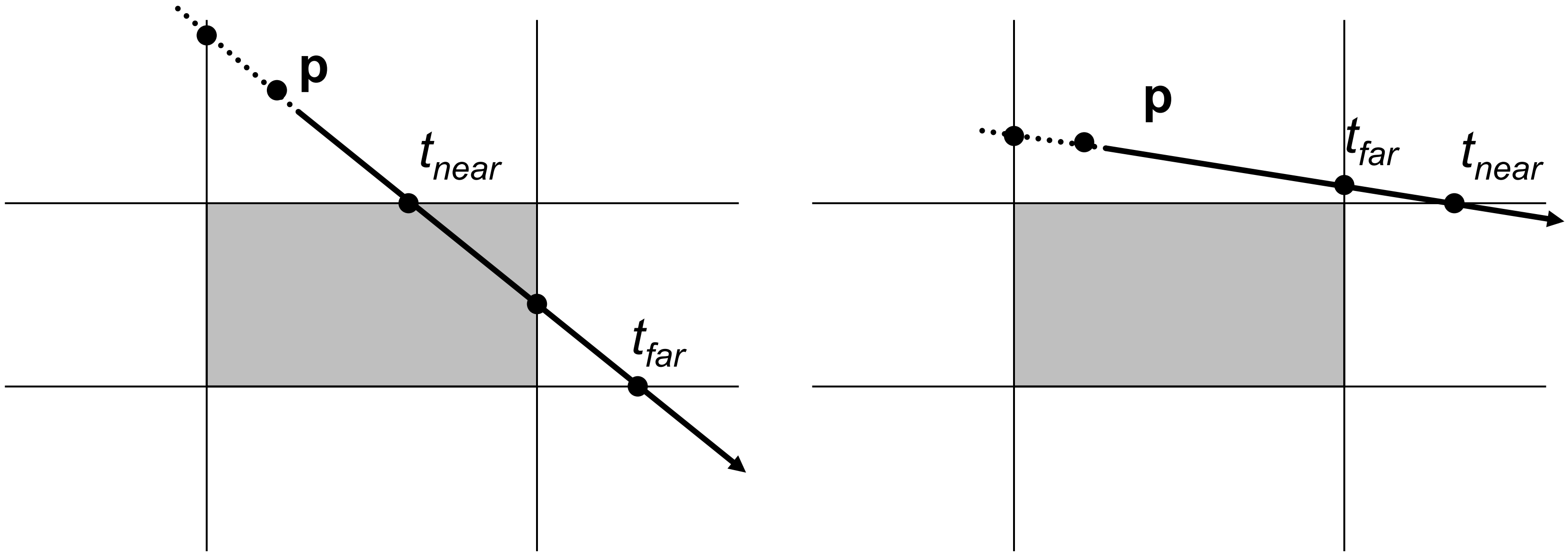
Polyhedron testing



Polygon testing

1. Use bounding sphere for fast failure
2. Test only front-facing polygons
3. Intersect ray with each polygon's supporting plane
4. Use a point-in-polygon test
5. Intersection point is smallest *t*

Axis-Aligned Cube Intersection



- for each pair of parallel planes, compute t intersection values for both
- Let t_{near} be the smaller, t_{far} be the larger
- let t_1 = largest t_{near}, t_2 = smallest t_{far}
- ray intersects cube if $t_1 <= t_2$
- intersection point given by t_1

Goodies

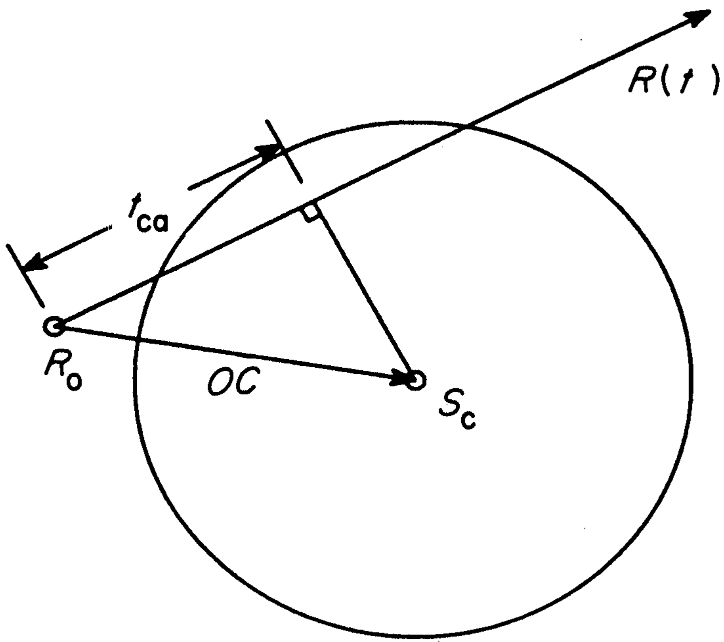
- There are some advanced ray tracing feature that self-respecting ray tracers shouldn't be caught without:
  - Acceleration techniques
  - Antialiasing
  - CSG
  - Distribution ray tracing

Acceleration Techniques

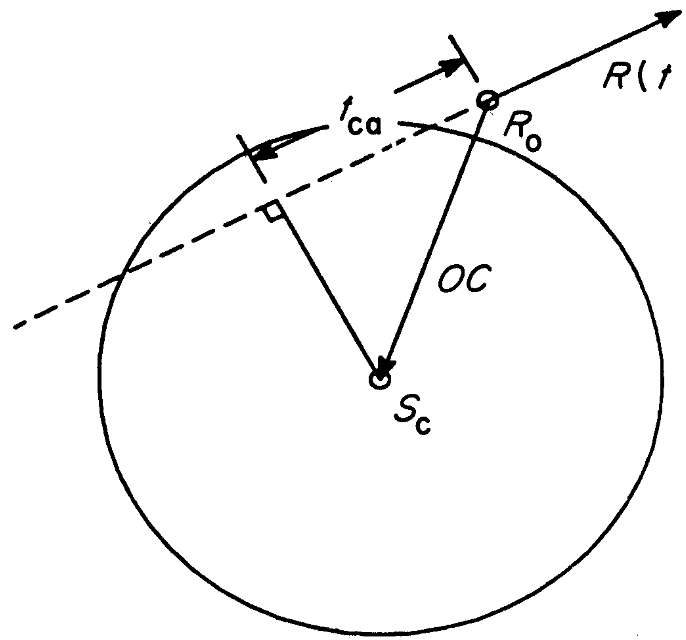
- Problem: ray-object intersection is very expensive
  - make intersection tests faster
  - do fewer tests

Fast Failure

- We can greatly speed up ray-object intersection by identifying cheap tests that guarantee failure
- Example: if origin of ray is outside sphere and ray points away from sphere, fail immediately.



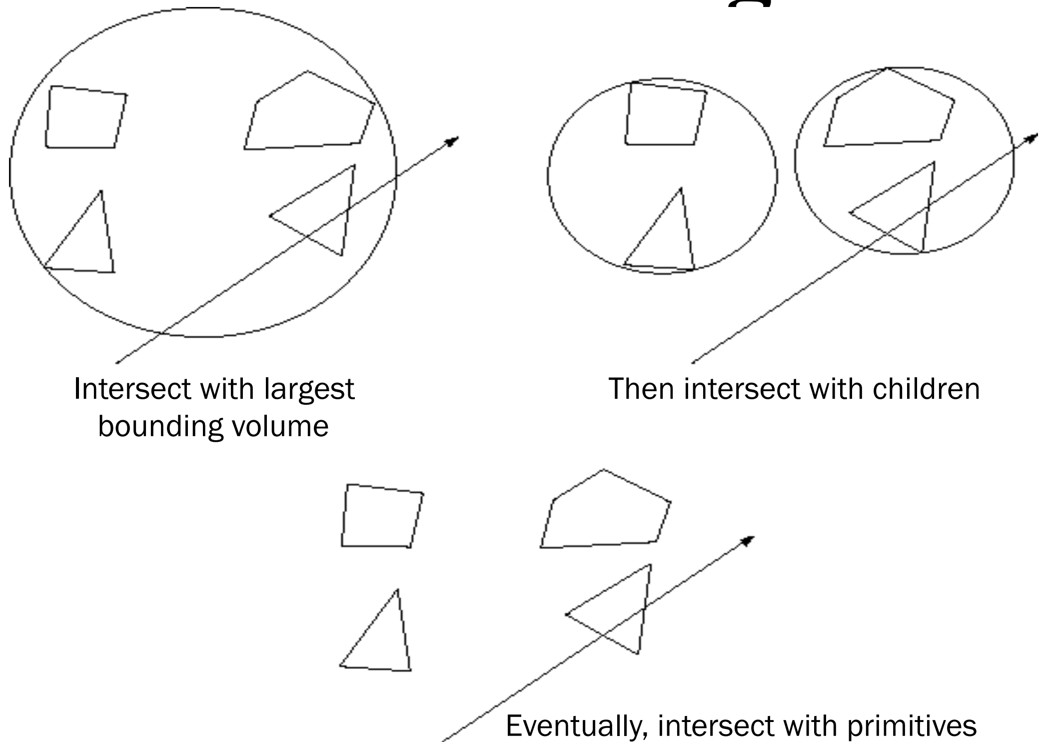
$t_{ca} > 0$, so the ray points toward the sphere



$t_{ca} < 0$, so the ray points away from the sphere

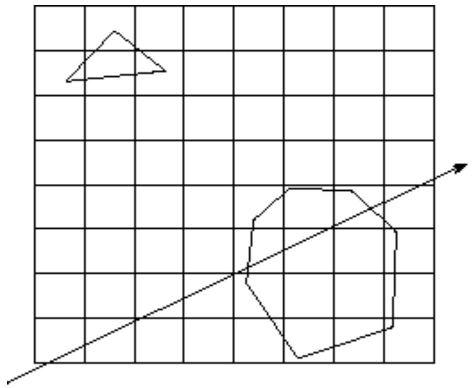
- Many other fast failure conditions are possible!

Hierarchical Bounding Volumes

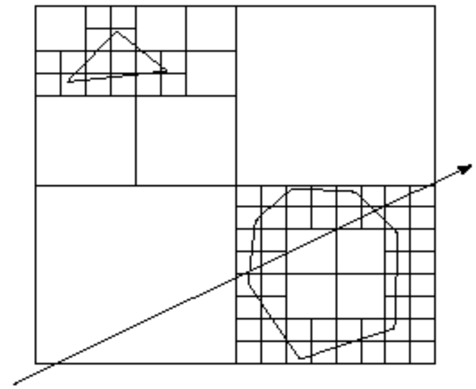


- Arrange scene into a tree
  - Interior nodes contain primitives with very simple intersection tests (e.g., spheres). Each node's volume contains all objects in subtree
  - Leaf nodes contain original geometry
- Like BSP trees, the potential benefits are big but the hierarchy is hard to build

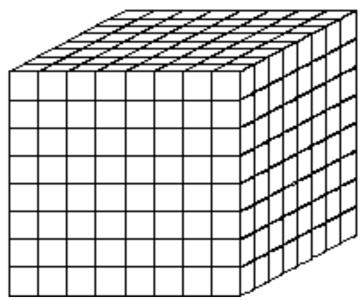
Spatial Subdivision



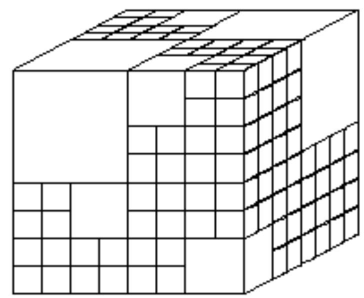
Uniform subdivision
in 2D



Quadtree



Uniform subdivision
in 3D

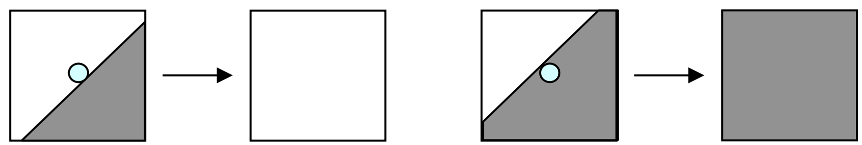


Octree

- Divide up space and record what objects are in each cell
- Trace ray through **voxel** array

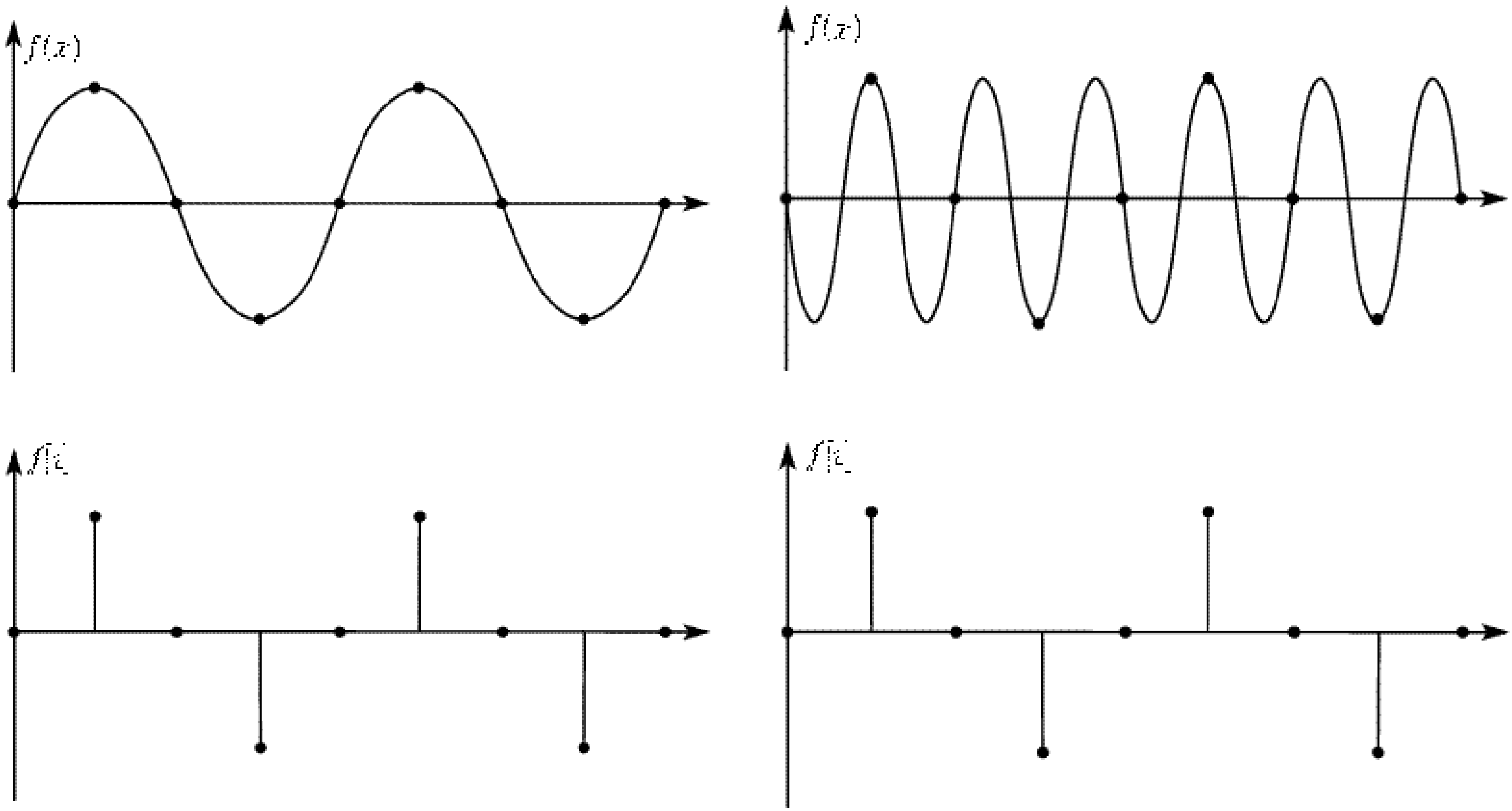
Antialiasing

- So far, we have traced one ray through each pixel in the final image. Is this an adequate description of the contents of the pixel?



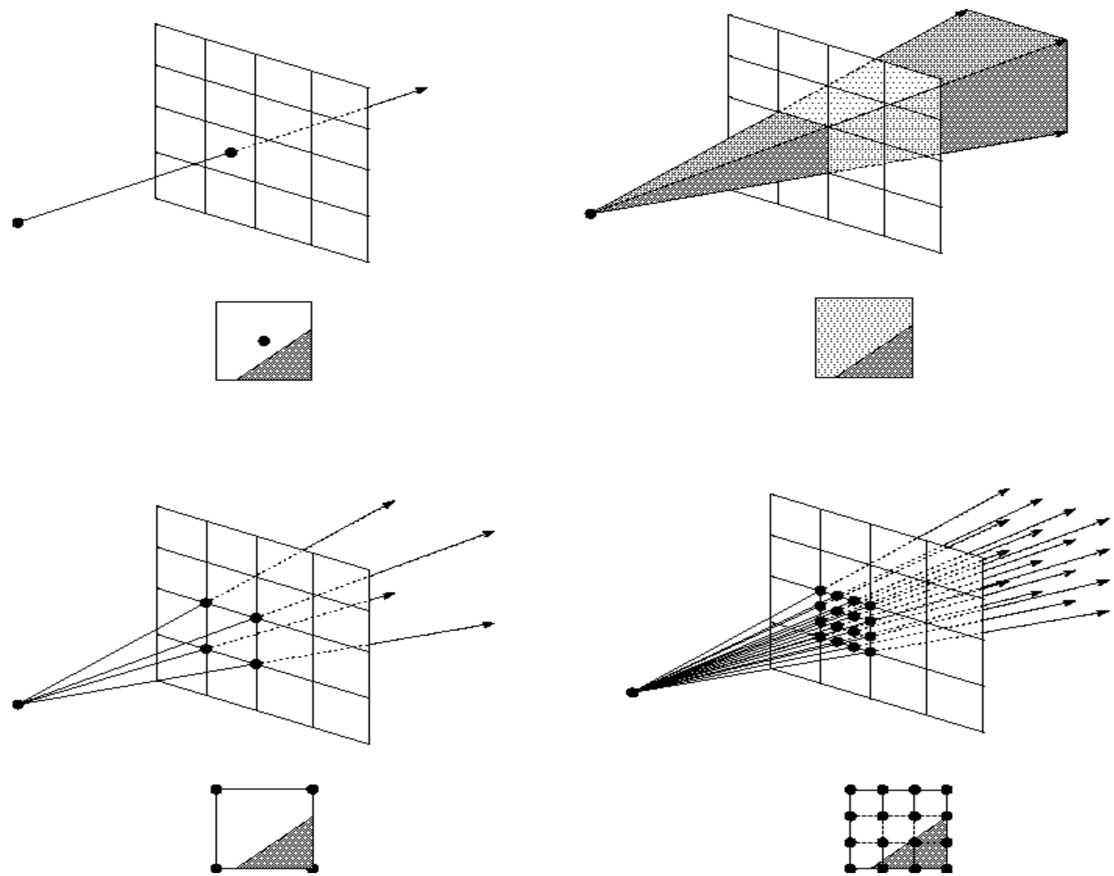
- This quantization through inadequate sampling is a form of **aliasing**. Aliasing is visible as "jaggies" in the ray-traced image.
- We really need to colour the pixel based on the *average*

Aliasing



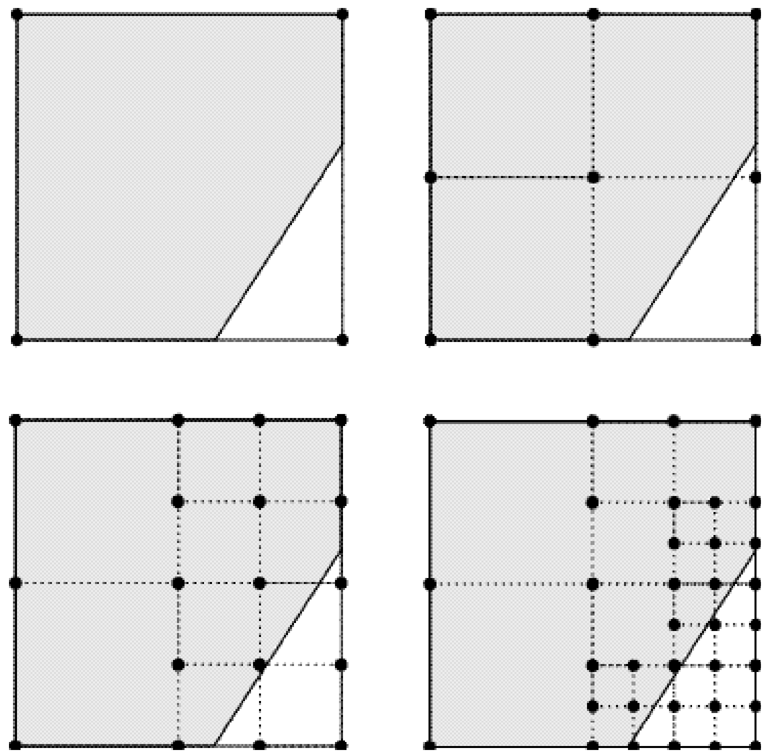
Supersampling

- We can approximate the average colour of a pixel's area by firing multiple rays and averaging the result.



Adaptive Sampling

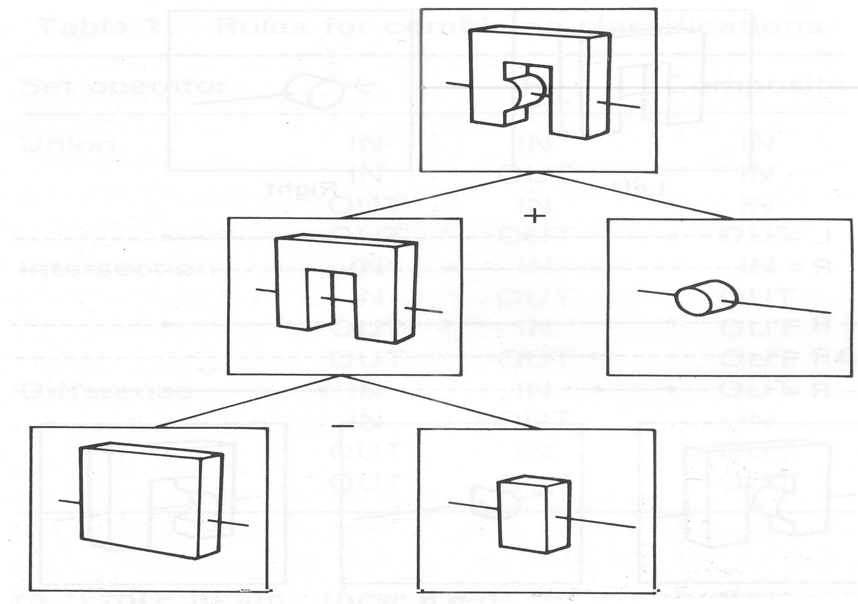
- Uniform supersampling can be wasteful if large parts of the pixel don’t change much.
- So we can subdivide regions of the pixel’s area only when the image changes in that area:



- How do we decide when to subdivide?

CSG

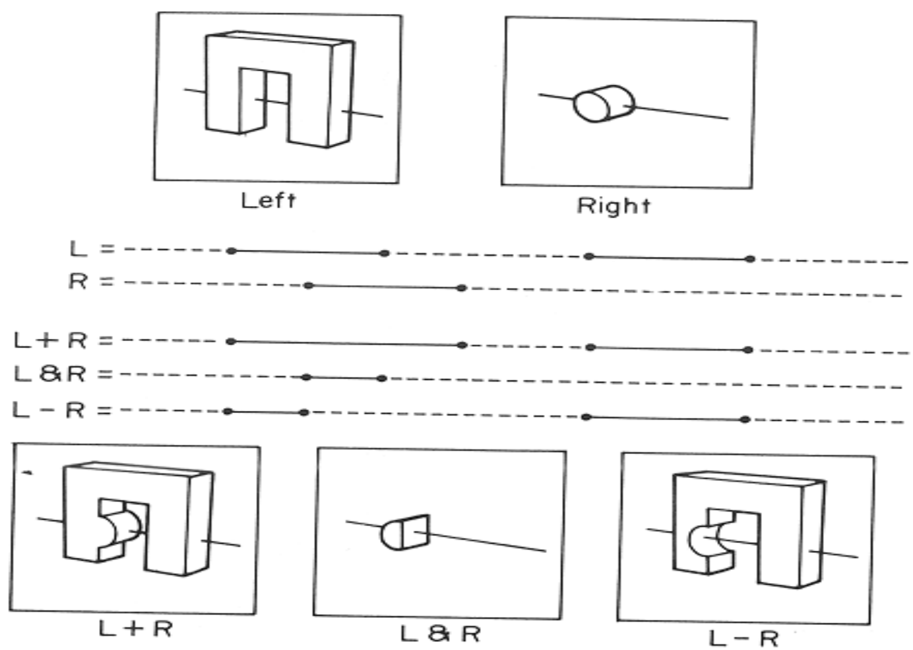
- CSG (constructive solid geometry) is an incredibly powerful way to create complex scenes from simple primitives.



- CSG is a modeling technique; basically, we only need to modify ray-object intersection.

CSG Implementation

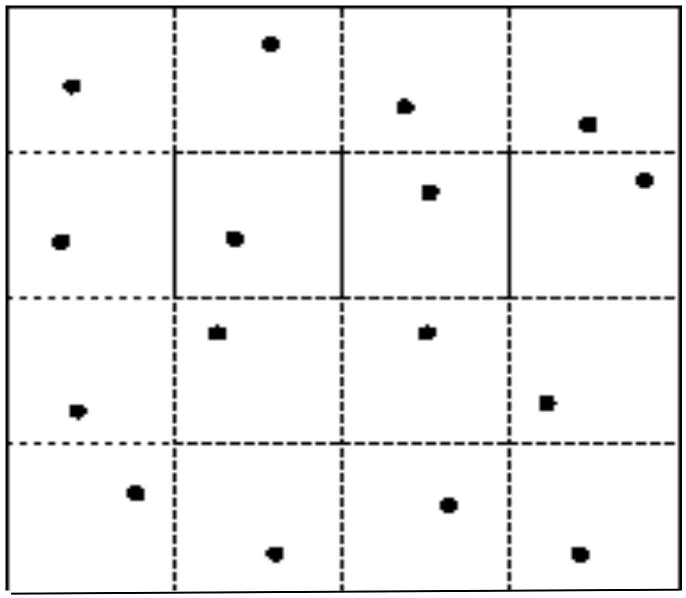
- CSG intersections can be analyzed using "Roth diagrams".
  - Maintain description of *all intersections* of ray with primitive
  - Functions to combine Roth diagrams under CSG operations



- An elegant and extremely slow system

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



- Distributing over different dimensions gives different effects
- Example: what if we distribute rays over pixel area?

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

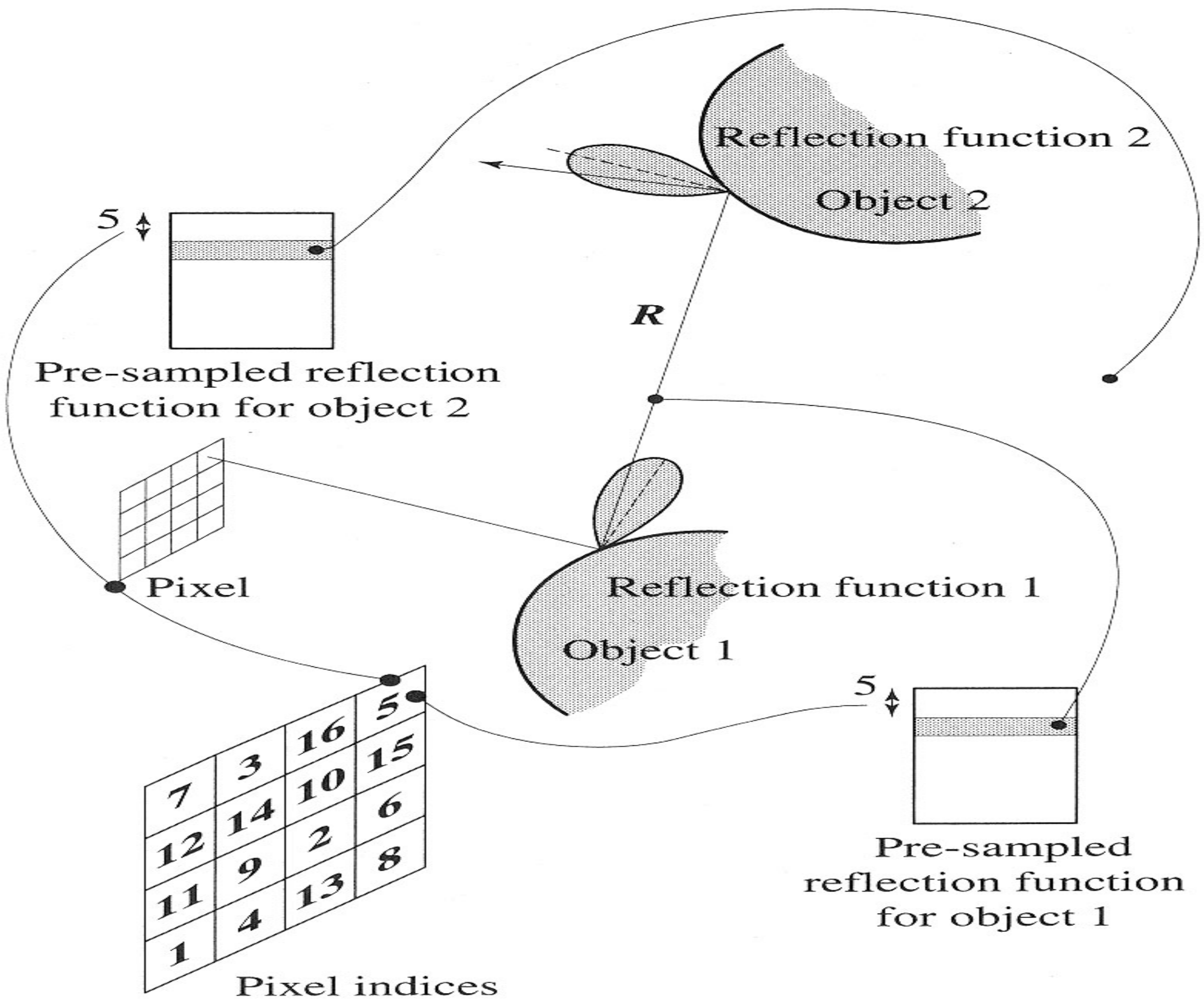
•A typical choice is numSubPixels = 4*4.

DRT pseudocode (cont'd)

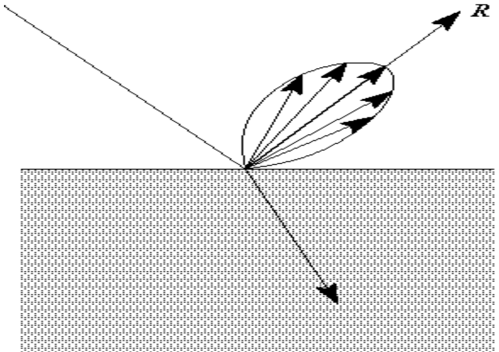
•Now consider *traceRay*(), modified to handle (only) opaque glossy surfaces:

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

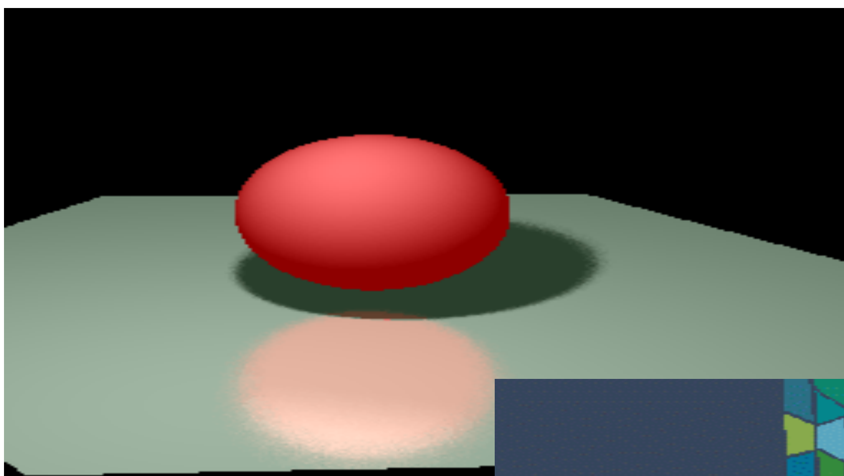
Pre-sampling glossy reflections

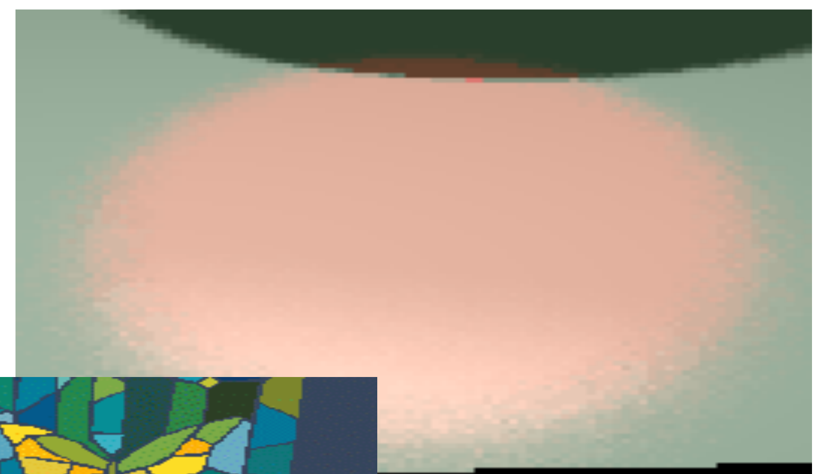


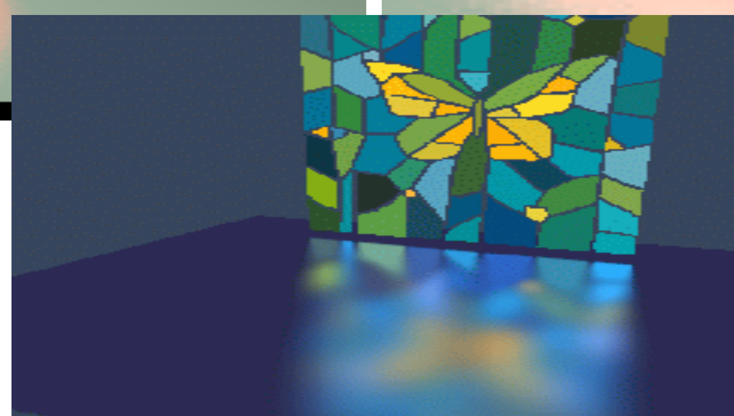
Distributing Reflections



- Distributing rays over reflection direction gives:

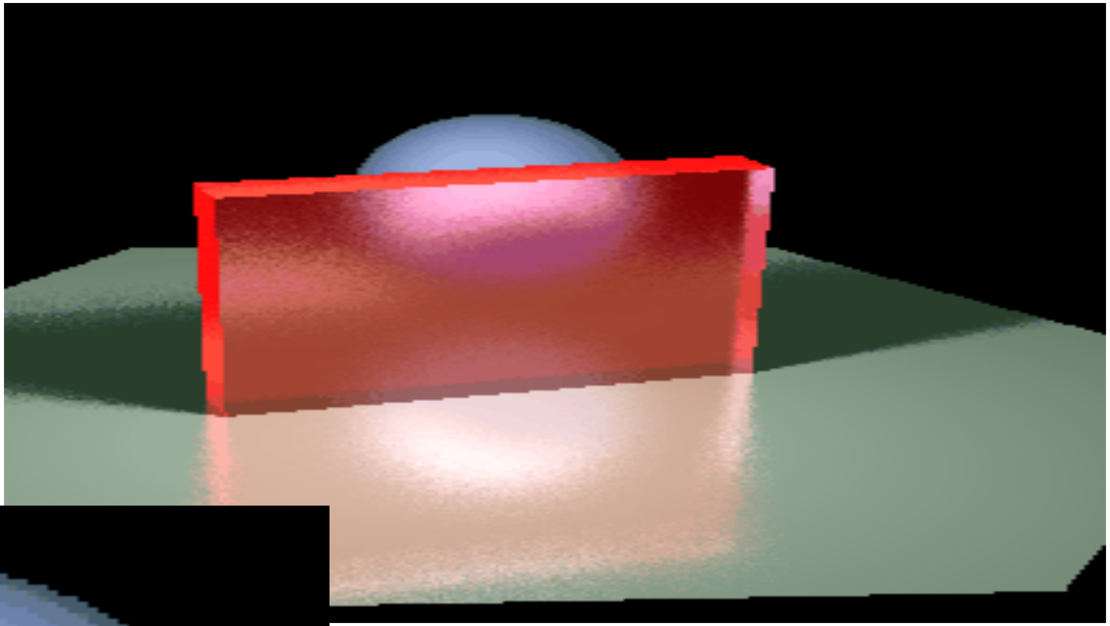


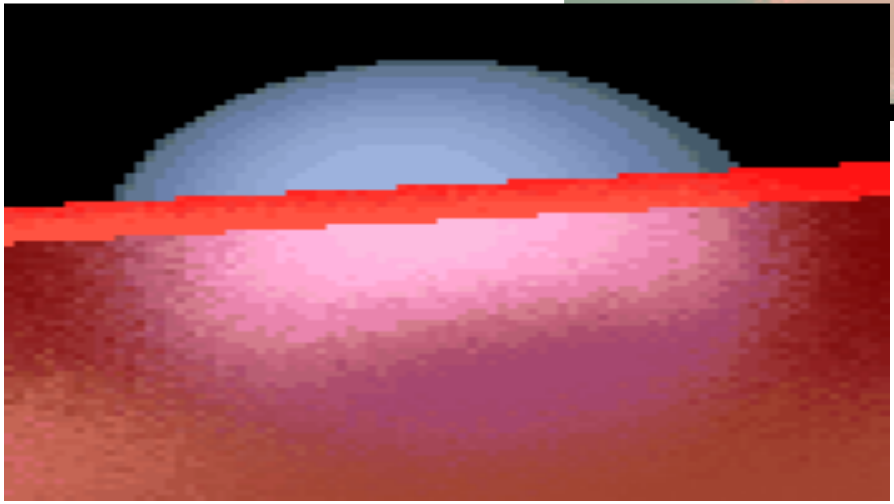




Distributing Refractions

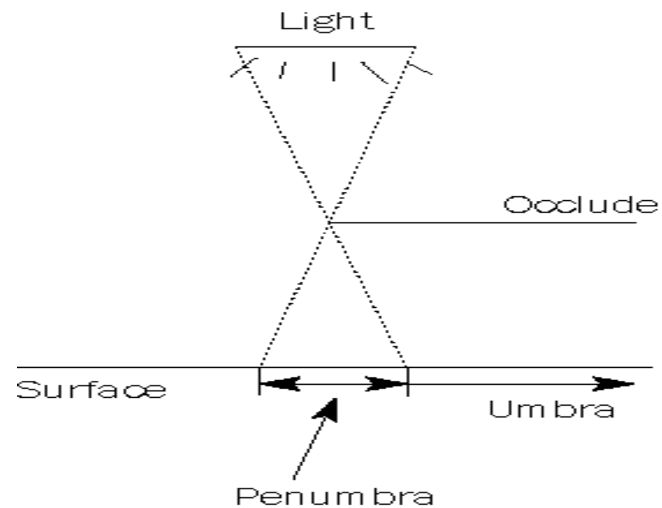
- Distributing rays over transmission direction gives:

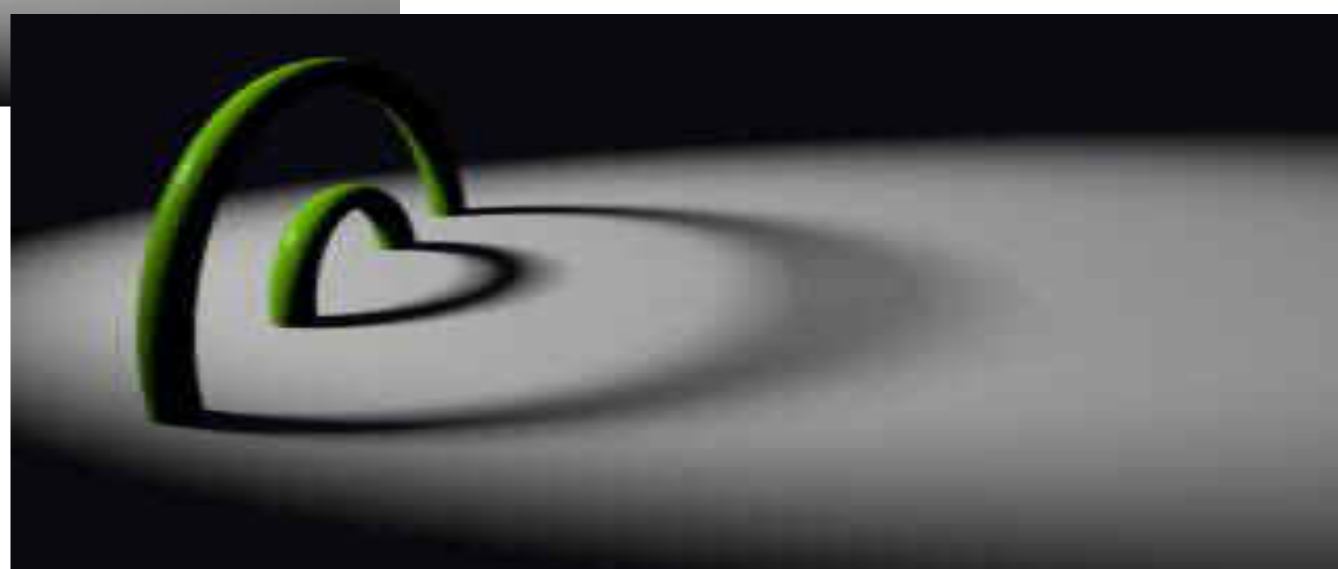




Distributing Over Light Area

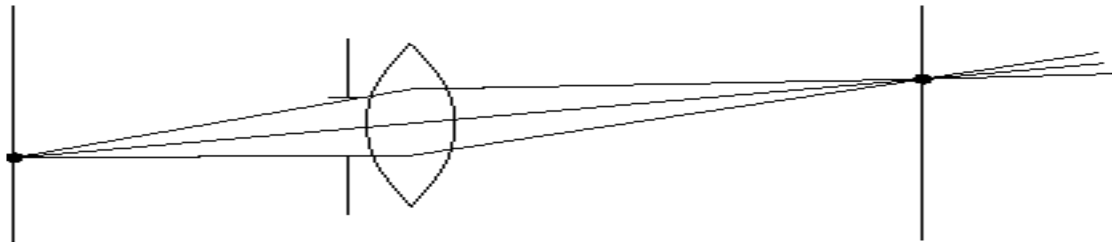
- Distributing over light area gives:

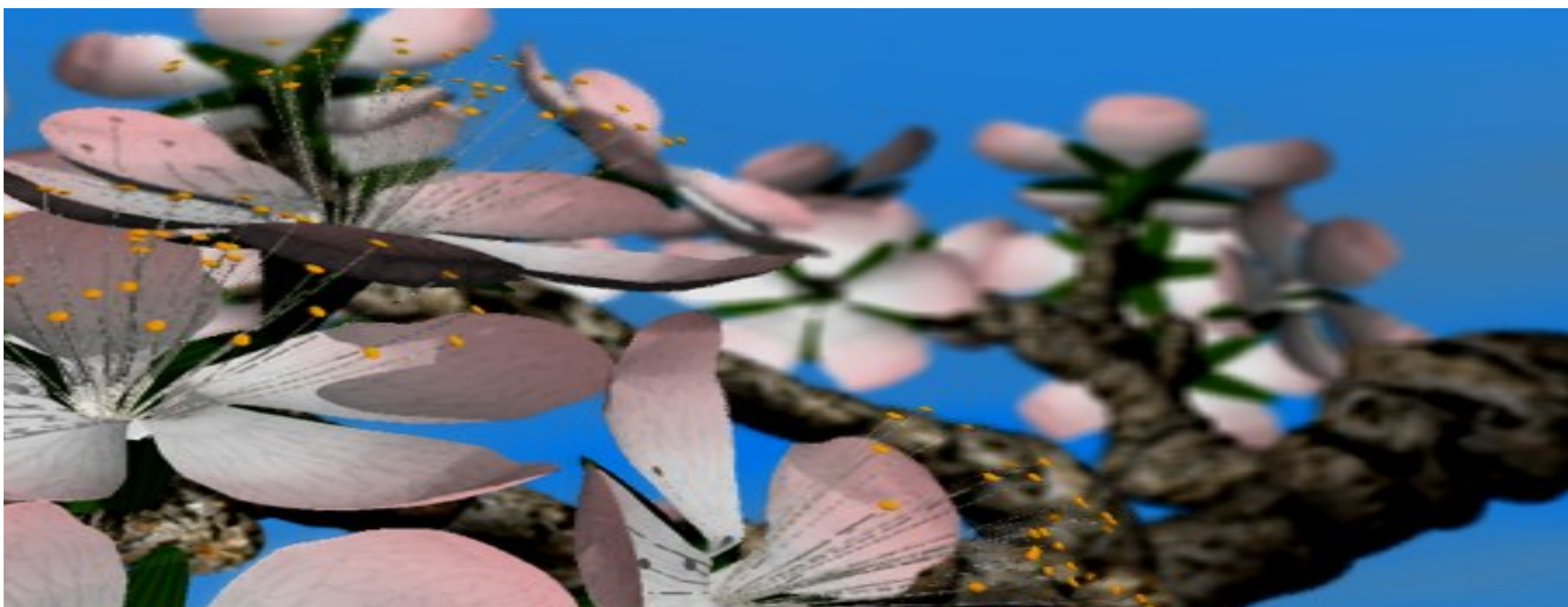




Distributing Over Aperature

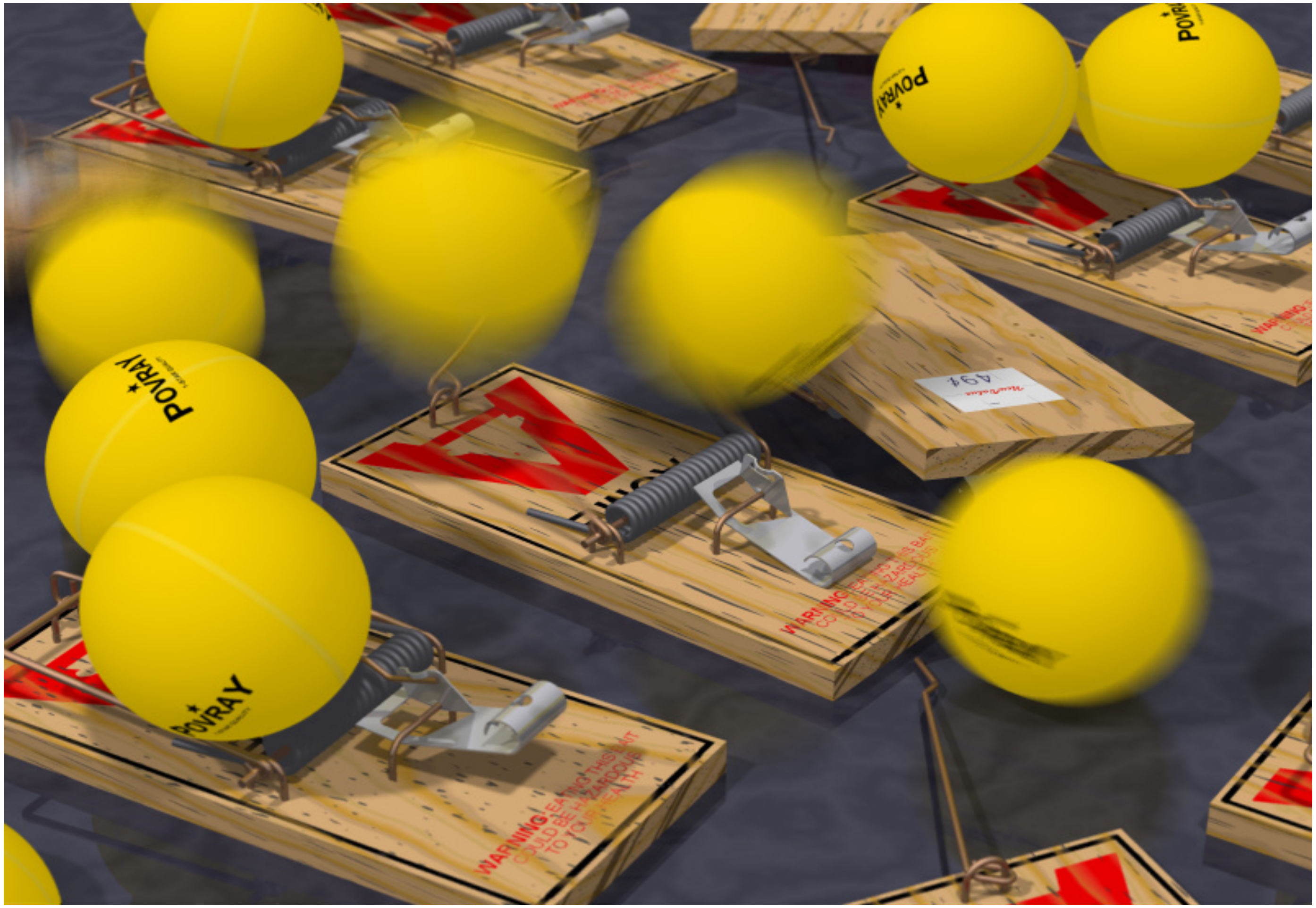
- We can fake distribution through a lens by choosing a point on a finite aperature and tracing through the "in-focus point".

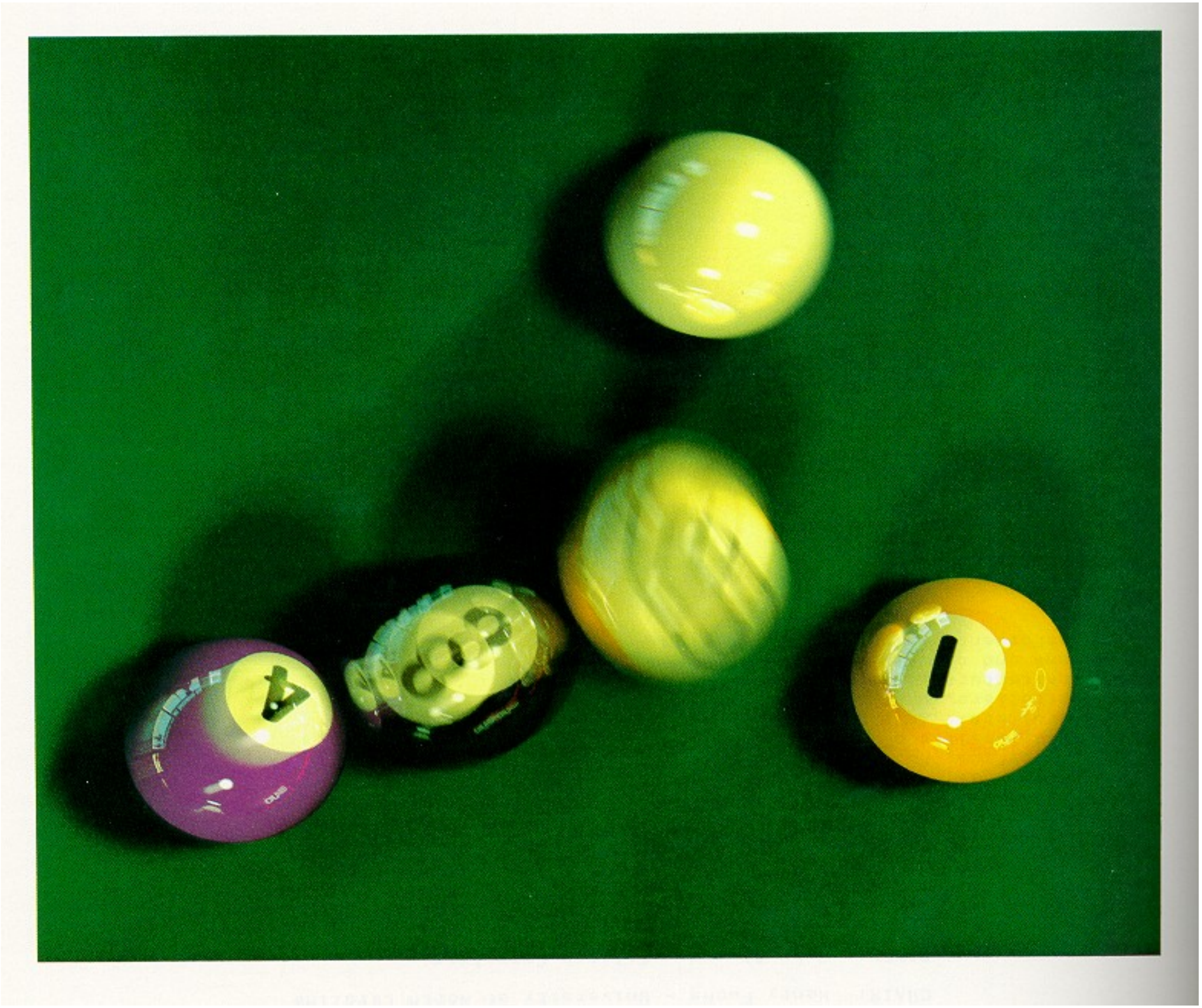




Distributing Over Time

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





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: