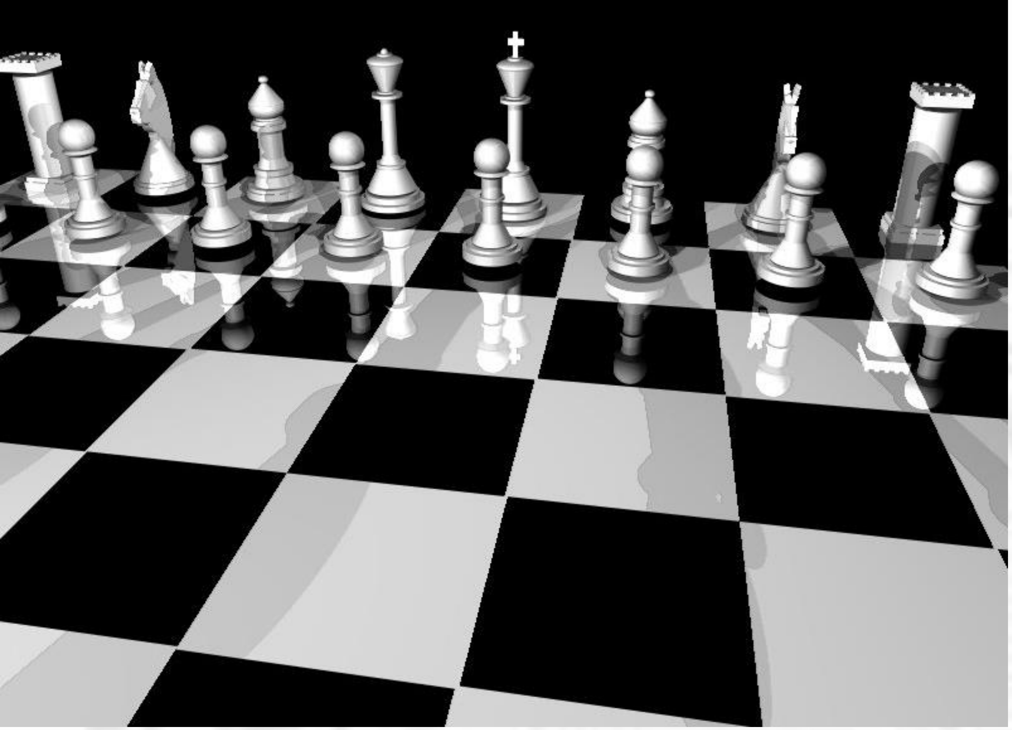
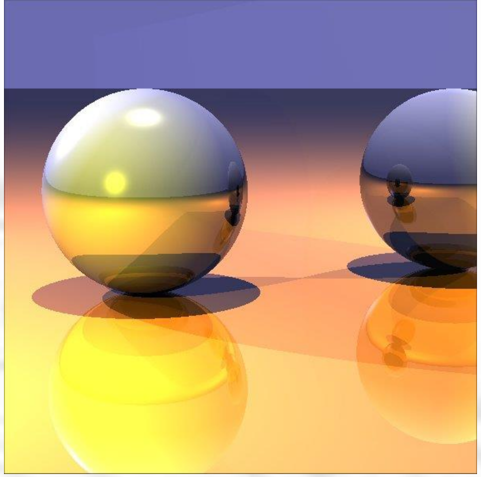
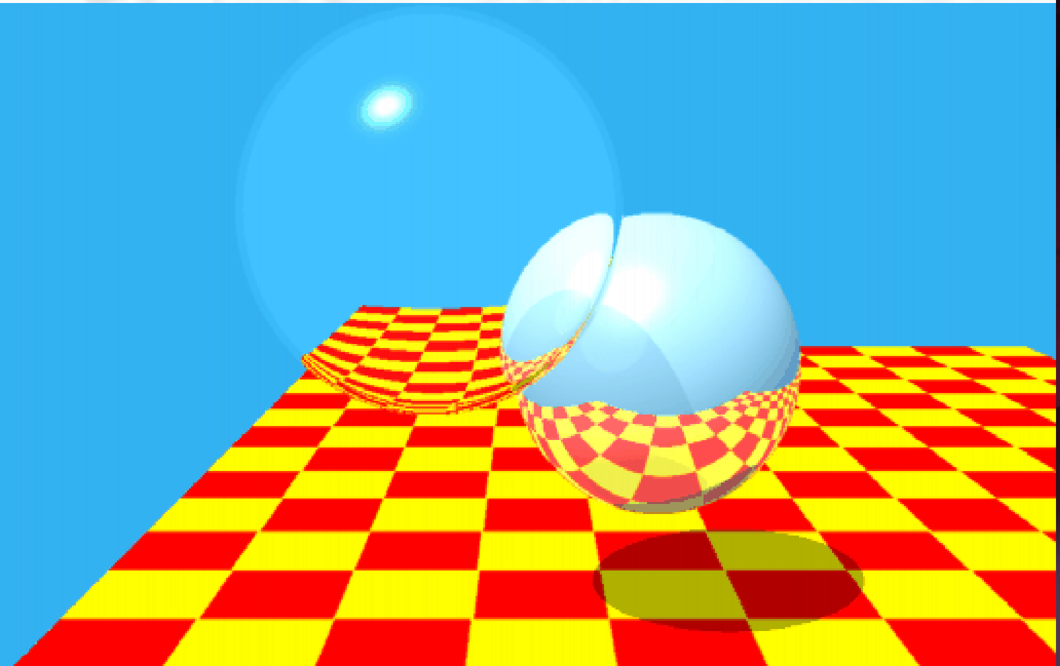
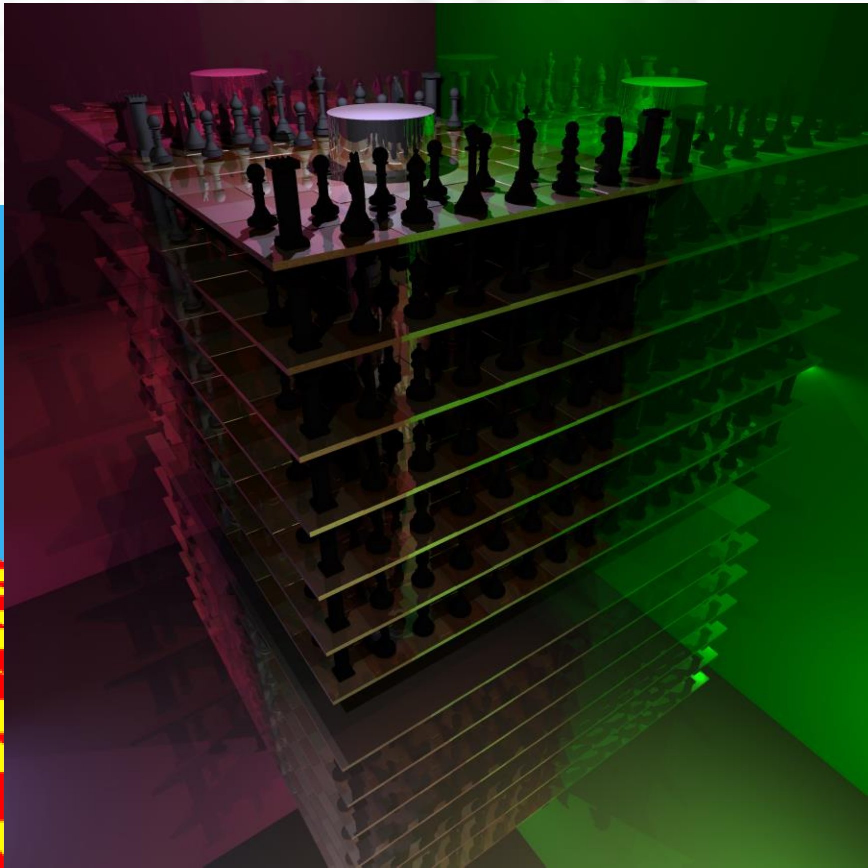
RayTracing

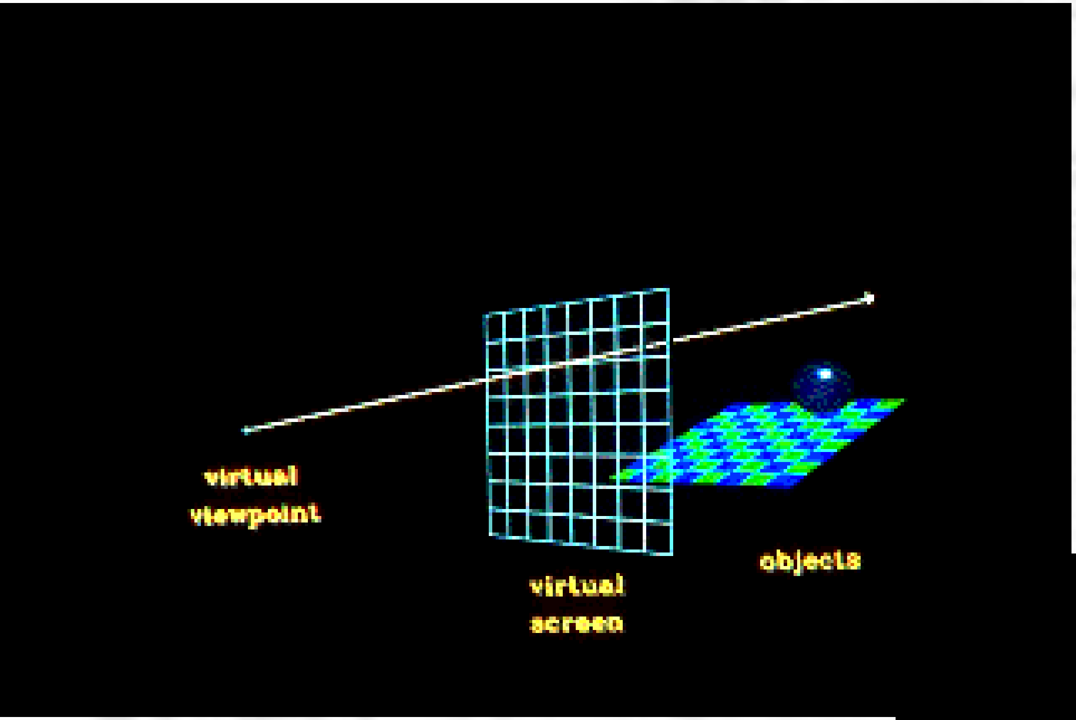
POV-Ray

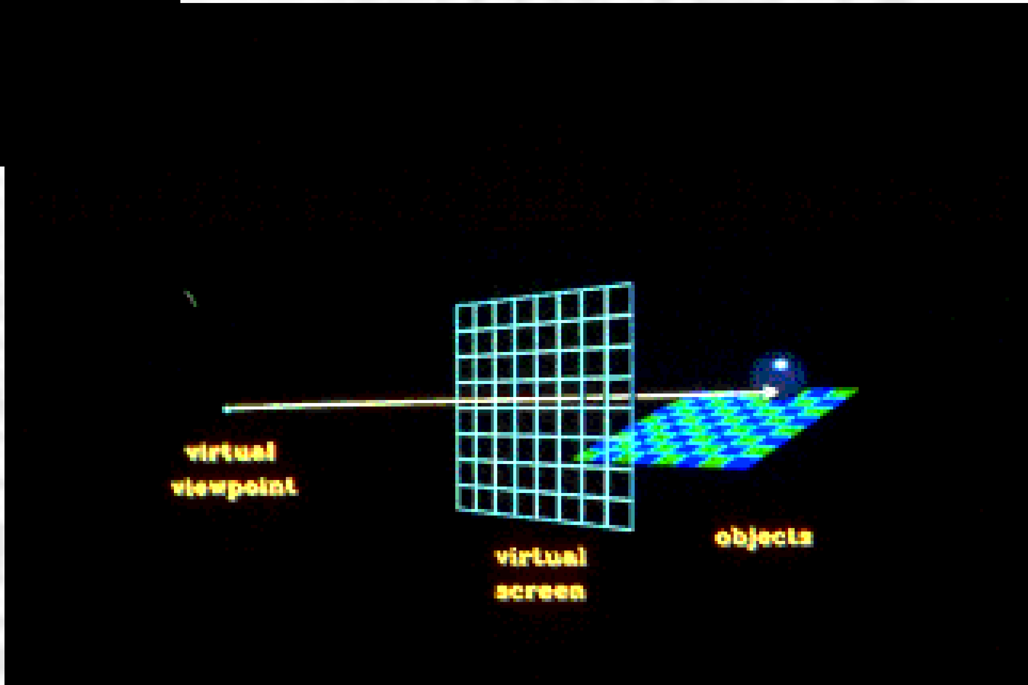
- Full-featured raytracer
- Free

Ray Tracing Basics

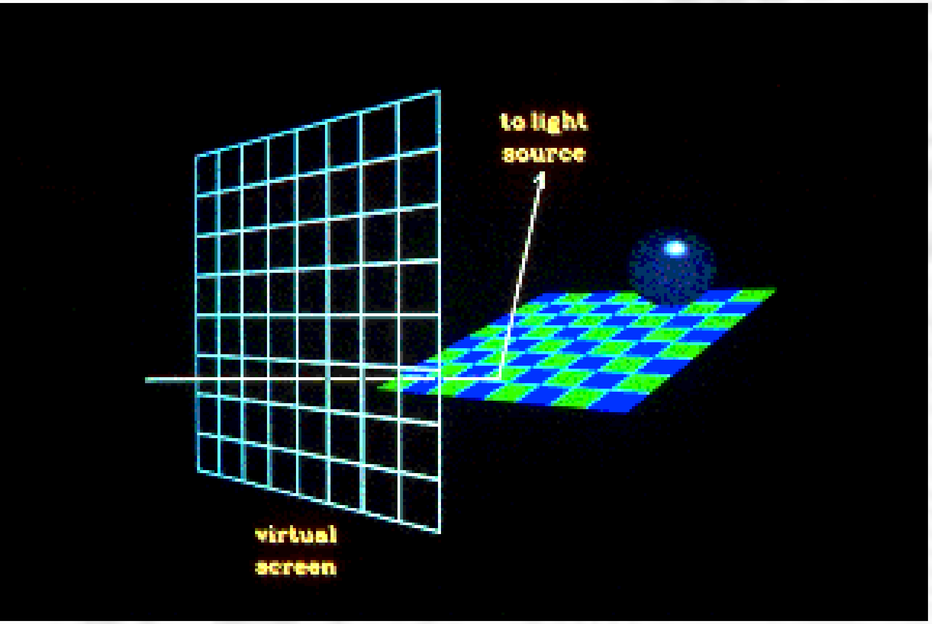
- Shoot ray in the *reverse* direction (from eyes to light instead of from light to eyes)
- Miss
- Hit
  - Shadow ray (to the light)
  - Reflected ray (on the same side)
  - Refracted ray (on the opposite side)

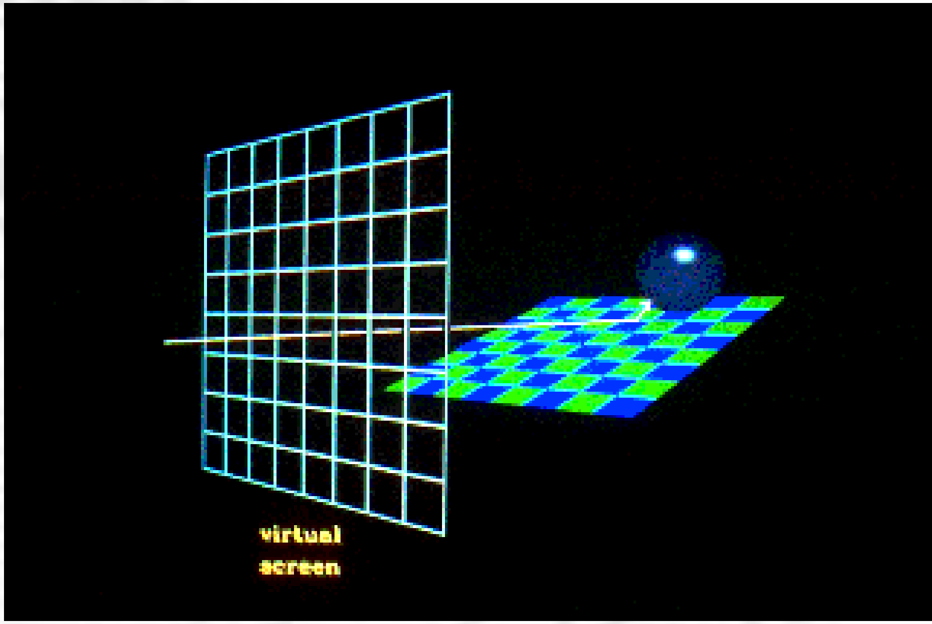
Hit and Miss


virtual
viewpoint
virtual
screen
objects

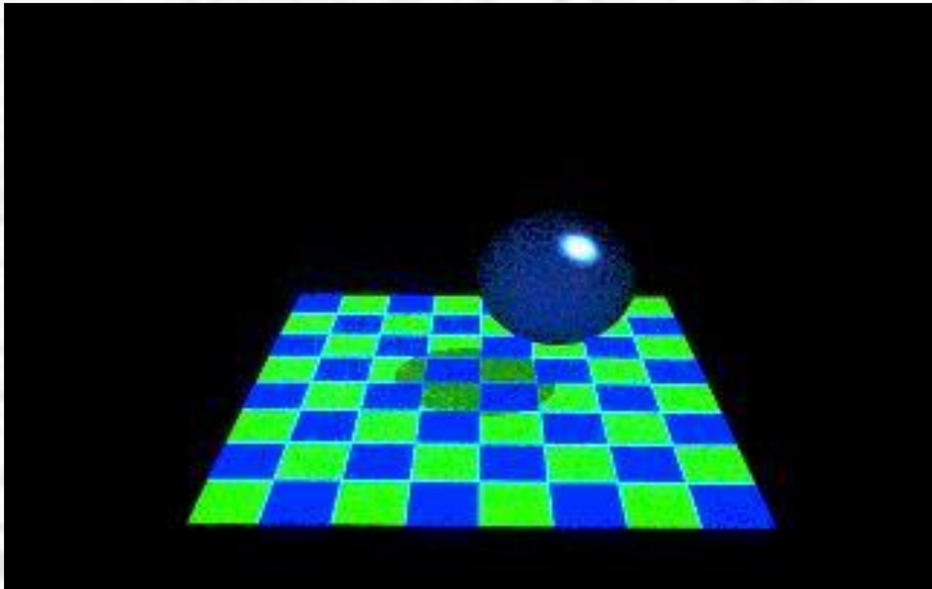

virtual
viewpoint
virtual
screen
objects

Shadow Ray

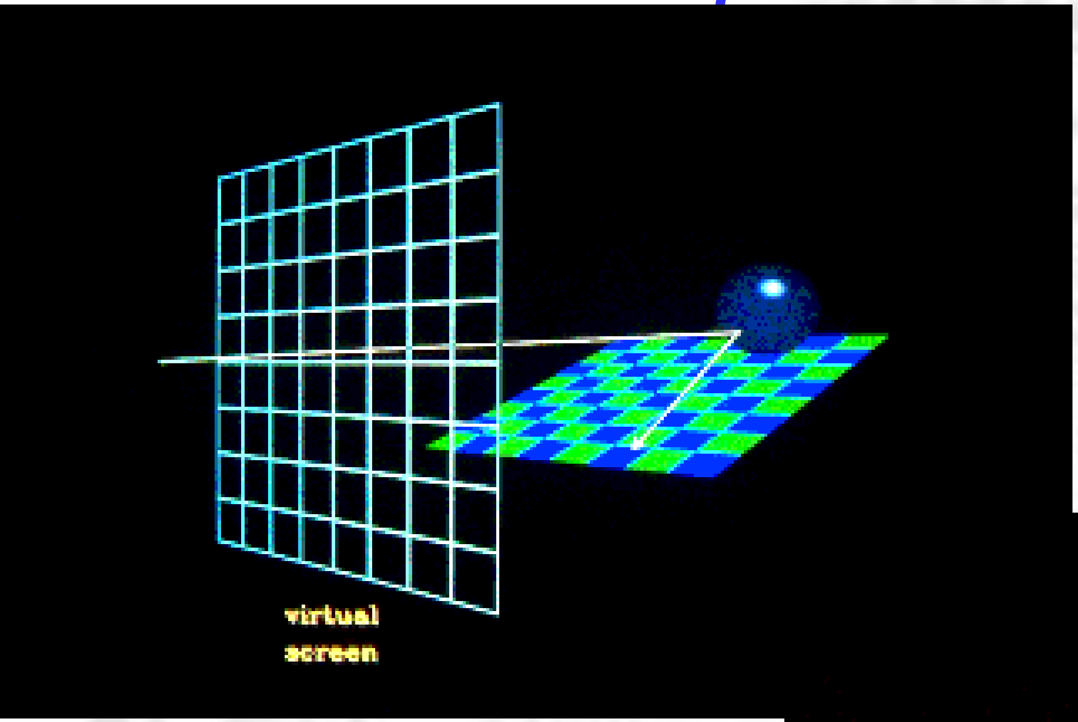




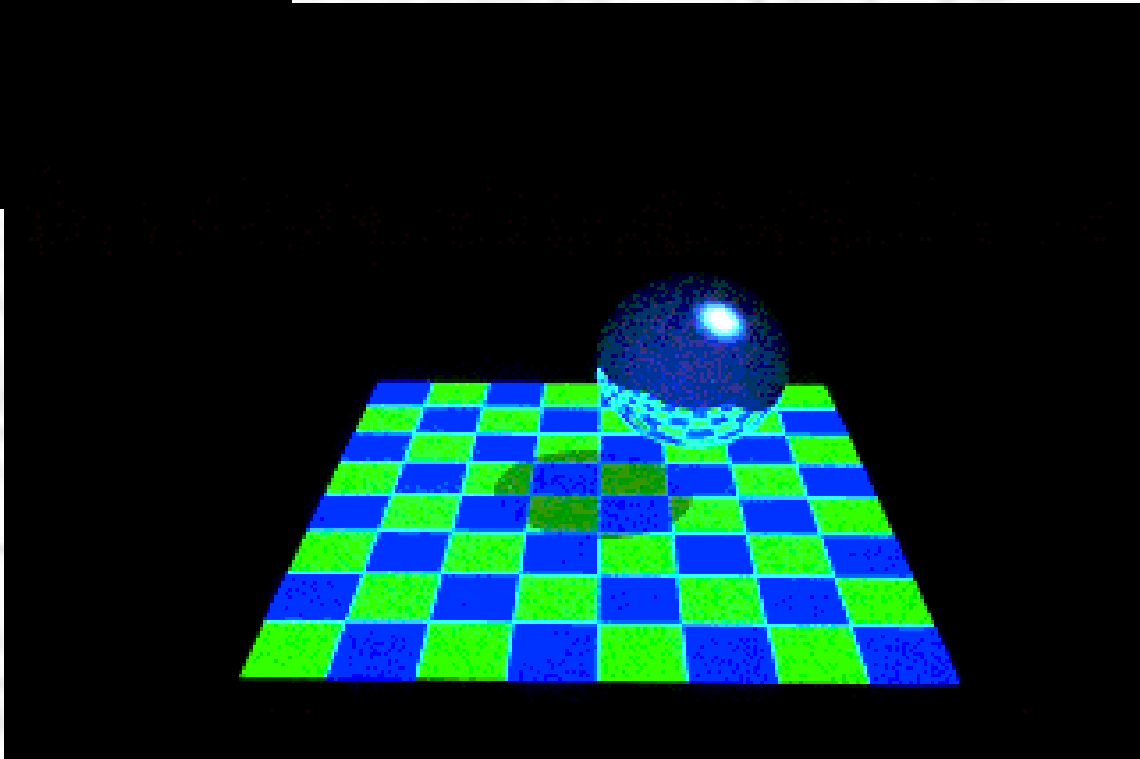
- Shadow ray
  - Blocked – in shadow
  - Not blocked



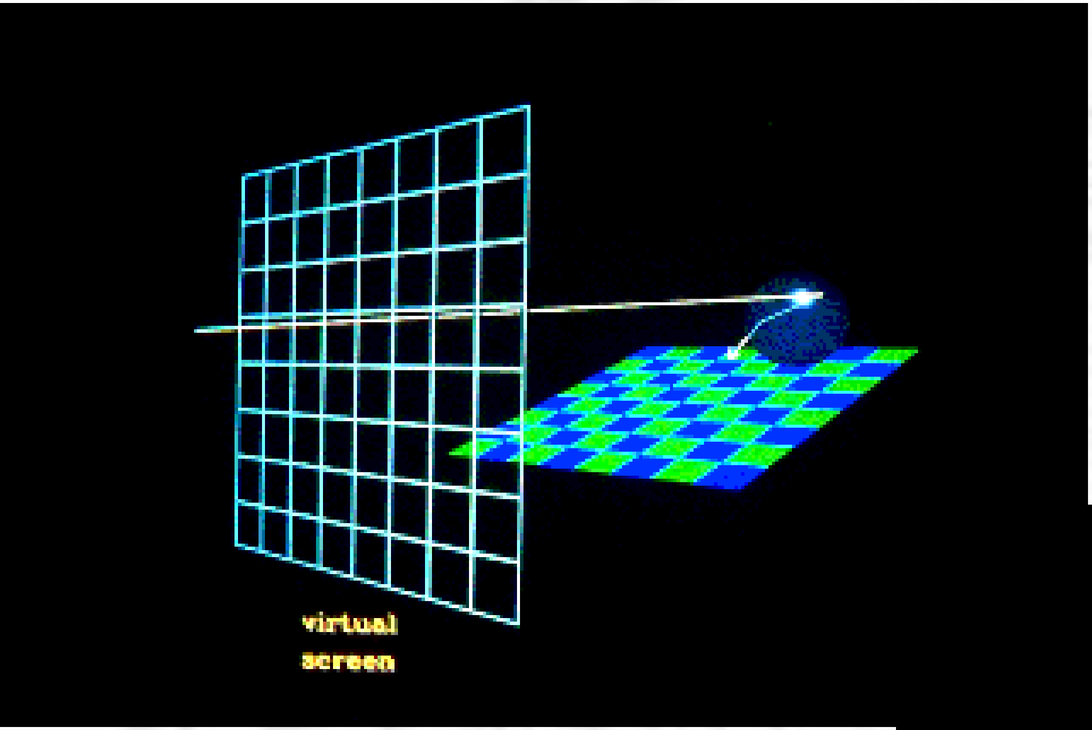
Reflected Ray

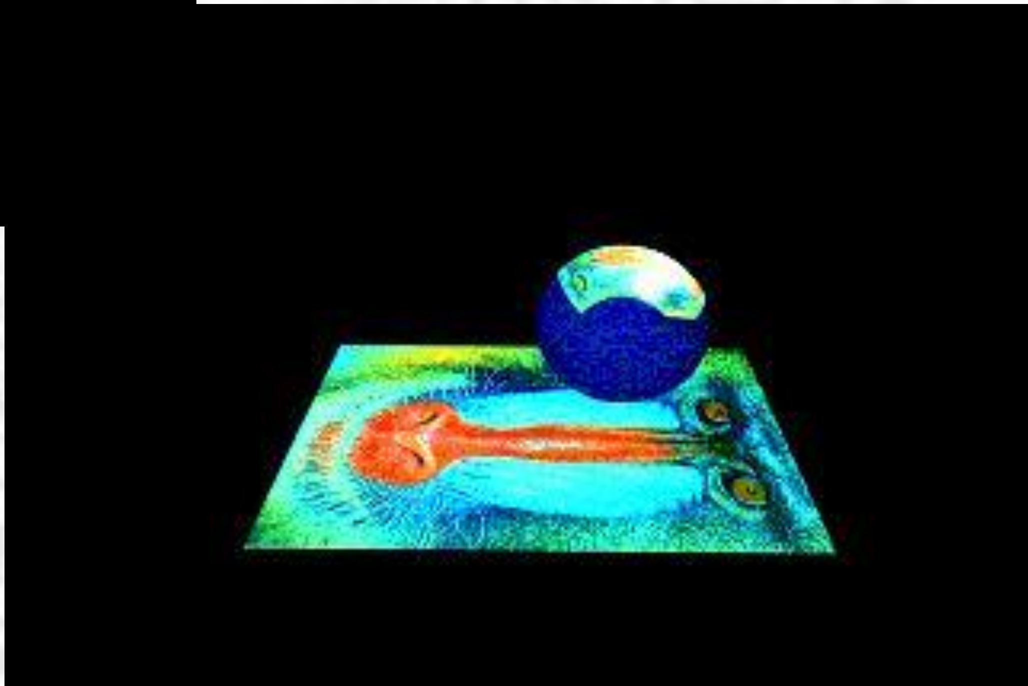


- Pick up color of objects on the same side



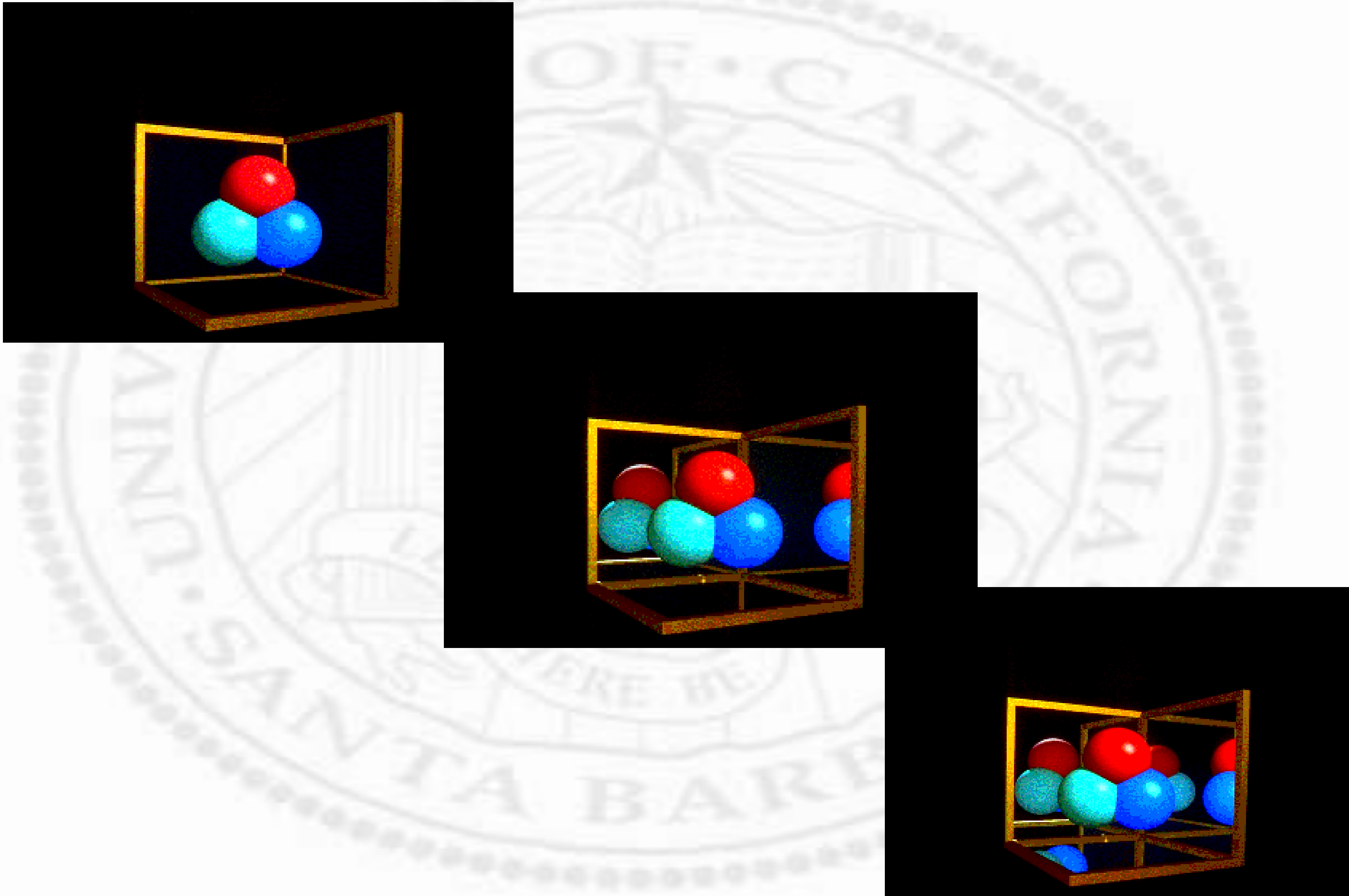
Refracted Ray





- Pick up color of objects on the opposite side

Multiple Levels of R/R

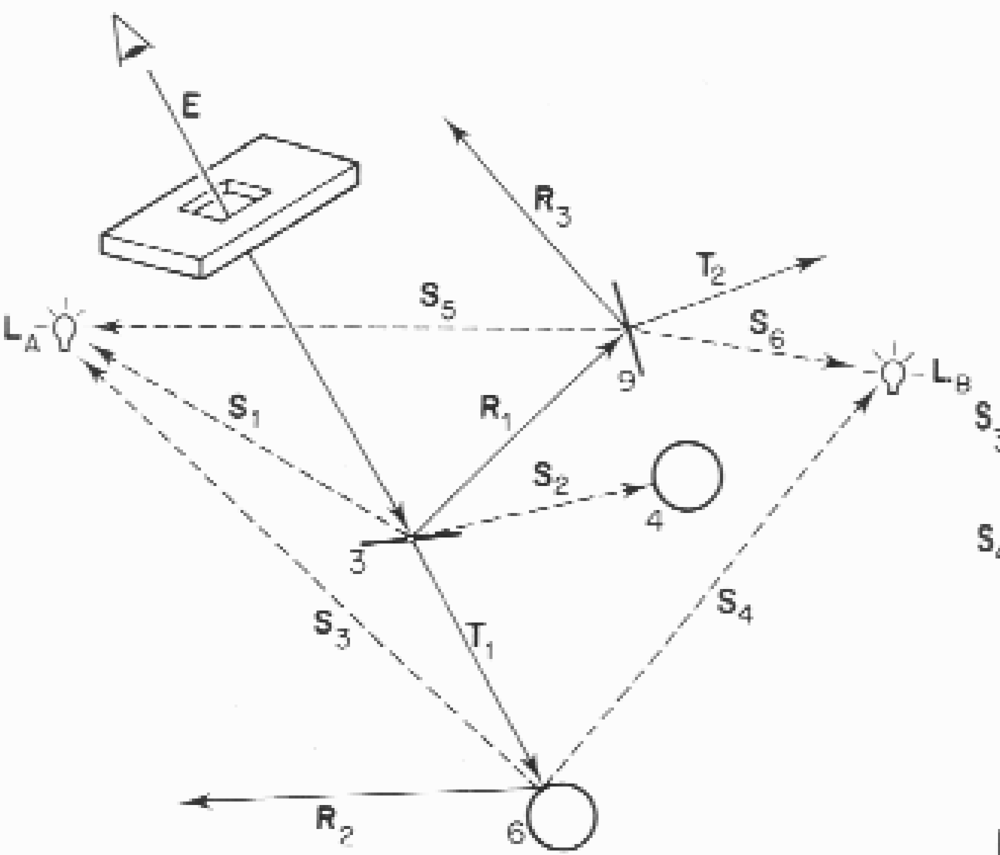


Visible Surface Ray Tracing

```
for (each scan line) {
    for (each pixel in scan line) {
        compute ray direction from COP (eye) to pixel
        for (each object in scene) {
            if (intersection and closest so far) {
                record object and intersection point        // a hit
            }
            accumulate pixel colors (one level)
                    - shadow ray color
                    - reflected ray color (recursion)
                    - refracted ray color (recursion)
        }
    }
```

Details

- $I = I_{local} + K_r * R + K_t * T$
- Build tree top-down
- Fill in values bottom-up



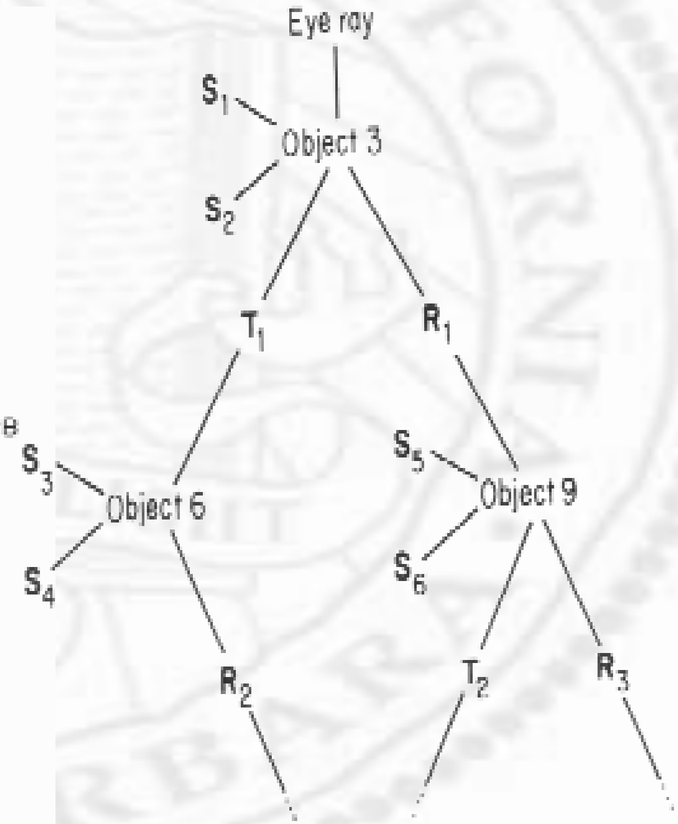
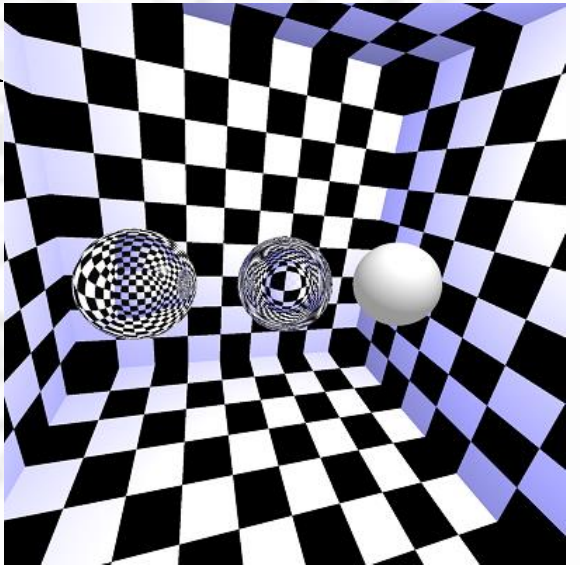


Fig. 12. The ray tree in schematic form.

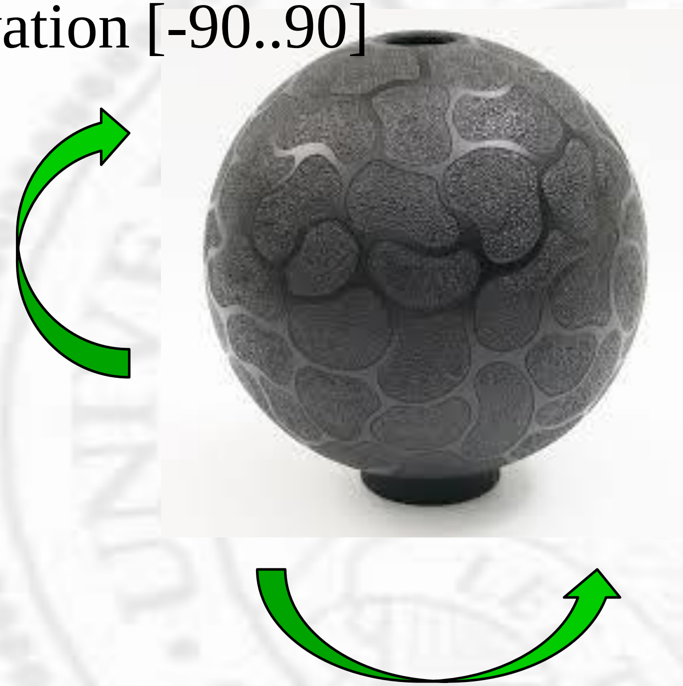
Local Color

- A single color [r, g, b] – no brainer
  - {r,g,b}_local = {r,g,b}_light * {r,g,b}_material *cos(θ) for each light not in shadow
  - Add one extra term {r,g,b}_ambient* {r,g,b}_material for background emission
  - This reflects a local diffuse model
- A texture image – every pixel can have a different color, more interesting



Texture Mapping

- Important to preserve aspect ratio so as not to distort content
  - Not always possible with sphere



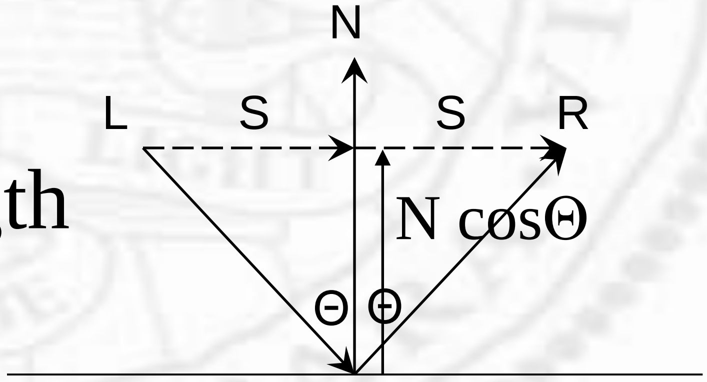

```
if xmax >= ymax,
        width = xmax
        height = ymax
else
        width = ymax
        height = xmax
end
if width >= 2*height
        wrange = 2*height
        hrange = height
else
        wrange = widith
        hrange = width/2
end
```

Computing Reflected Ray

- $S = N \cos\Theta + L = N\,(N.L) + L$

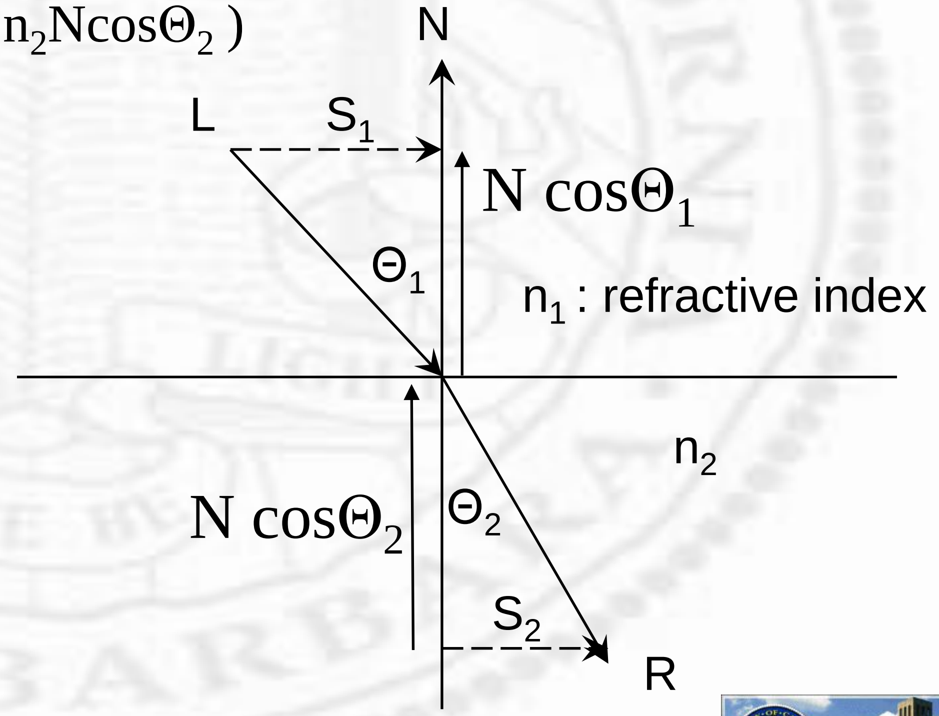
- $R = N \cos\Theta + S$
- $= N\,(N.L) + N\,(N.L) + L$
- $= 2\,N\,(N.L) + L$

- All vectors are UNIT length



Computing Refracted Ray

- $n_1 \sin\Theta_1 = n_2 \sin\Theta_2$ (Snell's law)
- $S_1 = L + N \cos\Theta_1 = L + N\,(N.L)$
- $S_2 = N \cos\Theta_2 + R$
- $S_1 / S_2 = \sin\Theta_1 / \sin\Theta_2 = n_2 / n_1$
- $= (L + N \cos\Theta_1) / (N \cos\Theta_2 + R)$
- $R = 1/n_2\,(n_1 L + n_1 N \cos\Theta_1 - n_2 N \cos\Theta_2)$



Ray-Object Intersection

- Implicit definition (f(P)=0)
  - f(x,y) = x^2+y^2-R^2
  - f(x,y,z) = Ax+By+Cz+D
  - f(x,y,z)=x^2+y^2+z^2-R^2
- Starting from a point **P** in space
- Go in the direction of **d**
- Point on ray is **P** + t**d**
- f(**P** + t**d**)=0
- Quadratic equations to solve (circle, sphere)

Ray-Object Intersection

- When and where?
  - Before normalization transform and projection
  - In the world coordinate system (in fact, often in object's own coordinate system)
  - Normalization transform won't help to simplify the math
    - Lights can be anywhere
    - Objects can be anywhere
    - Normalization only help with clipping and projection (a viewer centered operation)
- Hint: for HW, do it in the world coordinate

2D ray-circle intersection example

Consider the eye-point *P* = (-3, 1), the direction vector ***d*** = (.8, -.6) and the unit circle given by:

$$f(x,y) = x^2 + y^2 - R^2$$

A typical point of the ray is:

$$Q = P + t\mathbf{d} = (-3,1) + t(.8,-.6) = (-3 + .8t, 1 - .6t)$$

Plugging this into the equation of the circle:

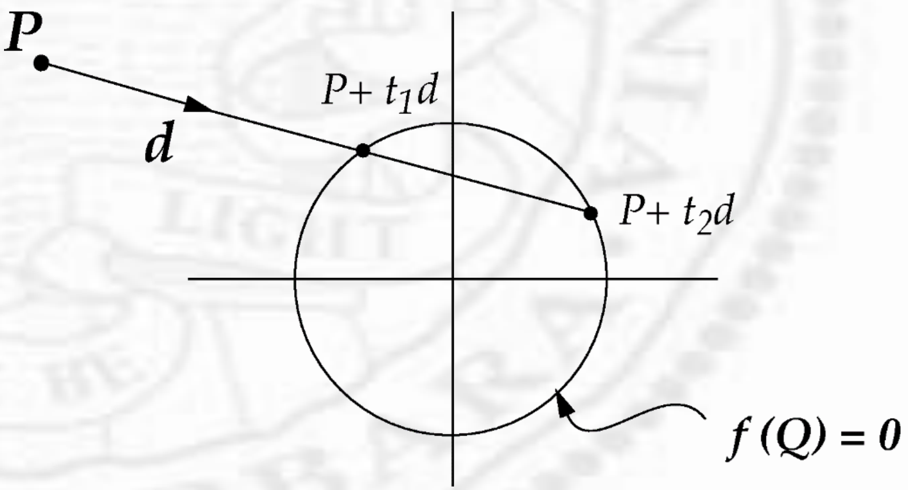
$$f(Q) = f(-3 + .8t, 1 - .6t) = (-3+.8t)^2 + (1-.6t)^2 - 1$$

Expanding, we get:

$$9 - 4.8t + .64t^2 + 1 - 1.2t + .36t^2 - 1$$

Setting this to zero, we get:

$$t^2 - 6t + 9 = 0$$



2D ray-circle intersection example (cont.)

Using the quadratic formula:

$$roots = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

We get:

$$t = \frac{6 \pm \sqrt{36 - 36}}{2}, \qquad t = 3, 3$$

Because we have a root of multiplicity 2, ray intersects circle at one point (i.e., it's tangent to the circle)

We can use discriminant $D = b^2 - 4ac$ to quickly determine if a ray intersects a curve or not

- if $D < 0$, imaginary roots; no intersection
- if $D = 0$, double root; ray is tangent
- if $D > 0$, two real roots; ray intersects circle at two points

Smallest non-negative real t represents intersection nearest to eye-point

Implicit objects-multiple conditions

For objects like cylinders, the equation

$$x^2 + z^2 - 1 = 0$$

in 3-space defines an infinite cylinder of unit radius, running along the y-axis

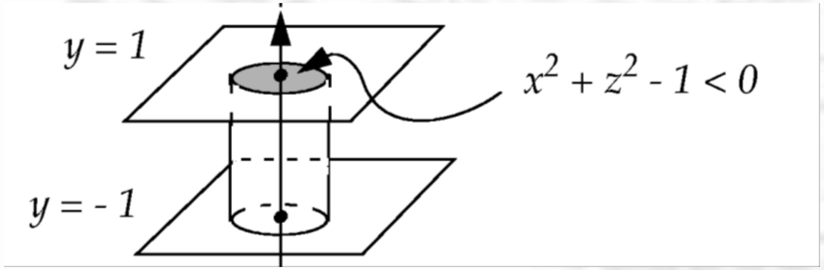
Usually, it's more useful to work with finite objects, e.g. such a unit cylinder truncated with the limits

$$y \leq 1$$
$$y \geq -1$$

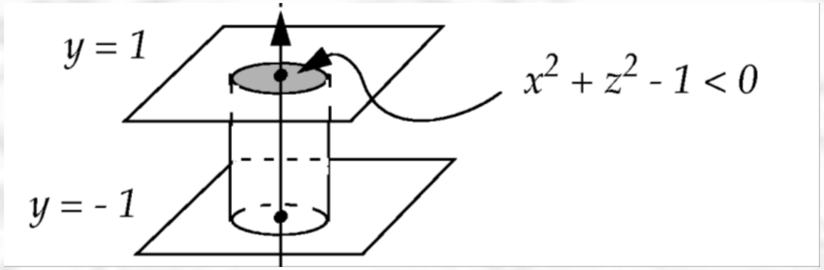
But how do we do the "caps?"

The cap is the inside of the cylinder at the y extrema of the cylinder

$$x^2 + z^2 - 1 < 0, \; y = \pm 1$$



Multiple conditions (cont.)



We want intersections satisfying the cylinder:

$$x^2 + z^2 - 1 = 0$$

$$-1 \le y \le 1$$

or top cap:

$$x^2 + z^2 - 1 \le 0$$

$$y = 1$$

or bottom cap:

$$x^2 + z^2 - 1 \le 0$$

$$y = -1$$

Multiple conditions-cylinder pseudocode

Solve in a case-by-case approach

```
Ray_inter_finite_cylinder(P,d):
// Check for intersection with infinite cylinder
t1,t2 = ray_inter_infinite_cylinder(P,d)
          compute P + t1*d, P + t2*d
// If intersection, is it between "end caps"?
if y > 1 or y < -1 for t1 or t2, toss it

// Check for intersection with top end cap
Compute ray_inter_plane(t3, plane y = 1)
Compute P + t3*d
// If it intersects, is it within cap circle?
if x^2 + z^2 > 1, toss out t3

// Check intersection with other end cap
Compute ray_inter_plane(t4, plane y = -1)
Compute P + t4*d
// If it intersects, is it within cap circle?
if x^2 + z^2 > 1, toss out t4

Among all the t's that remain (1-4), select the smallest non-negative
one
```

$$sphere: (X-a)^2 + (Y-b)^2 + (Z-c)^2 - r^2 = 0$$

$$ray: \begin{cases} X = X_o + t\Delta X \\ Y = Y_o + t\Delta Y \\ Z = Z_o + t\Delta Z \end{cases}$$

$$X^2 - 2aX + a^2 + Y^2 - 2bY + b^2 + Z^2 - 2cZ + c^2 - r^2 = 0$$

$$(X_o + t\Delta X)^2 - 2a(X_o + t\Delta X) + a^2 +$$
$$(Y_o + t\Delta Y)^2 - 2b(Y_o + t\Delta Y) + b^2 +$$
$$(Z_o + t\Delta Z)^2 - 2c(Z_o + t\Delta Z) + c^2 - r^2 = 0$$

$$(\Delta X^2 + \Delta Y^2 + \Delta Z^2)t^2 +$$
$$2\{\Delta X(X_o - a) + \Delta Y(Y_o - b) + \Delta Z(Z_o - c)\}t +$$
$$(X_o - a)^2 + (Y_o - b)^2 + (Z_o - c)^2 - r^2 = 0$$

$$(\Delta X^2 + \Delta Y^2 + \Delta Z^2)t^2 +$$
$$2\{\Delta X(X_o - a) + \Delta Y(Y_o - b) + \Delta Z(Z_o - c)\}t +$$
$$(X_o - a)^2 + (Y_o - b)^2 + (Z_o - c)^2 - r^2 = 0$$

$$At^2 + Bt + C = 0 \quad \Delta = B^2 - 4AC$$

$$t\begin{cases} \Delta > 0 & \text{insersecti ng} \\ \Delta = 0 & \text{grazing} \\ \Delta < 0 & \text{non insersecti ng} \end{cases}$$

$$plane: aX + bY + cZ + d = 0$$

$$ray: \begin{cases} X = X_o + t\Delta X \\ Y = Y_o + t\Delta Y \\ Z = Z_o + t\Delta Z \end{cases}$$

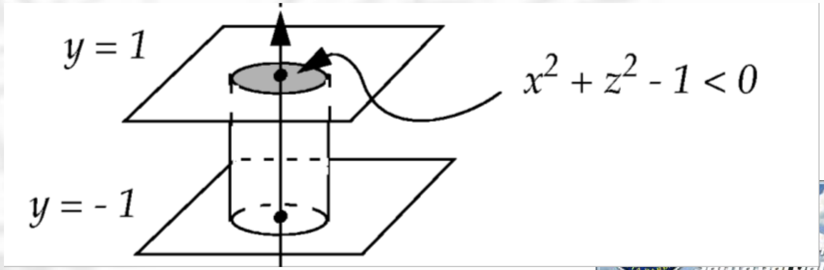
$$a(X_o + t\Delta X) + b(Y_o + t\Delta Y) + c(Z_o + t\Delta Z) + d = 0$$

$$t = -\frac{aX_o + bY_o + cZ_o + d}{a\Delta X + b\Delta Y + c\Delta Z}$$

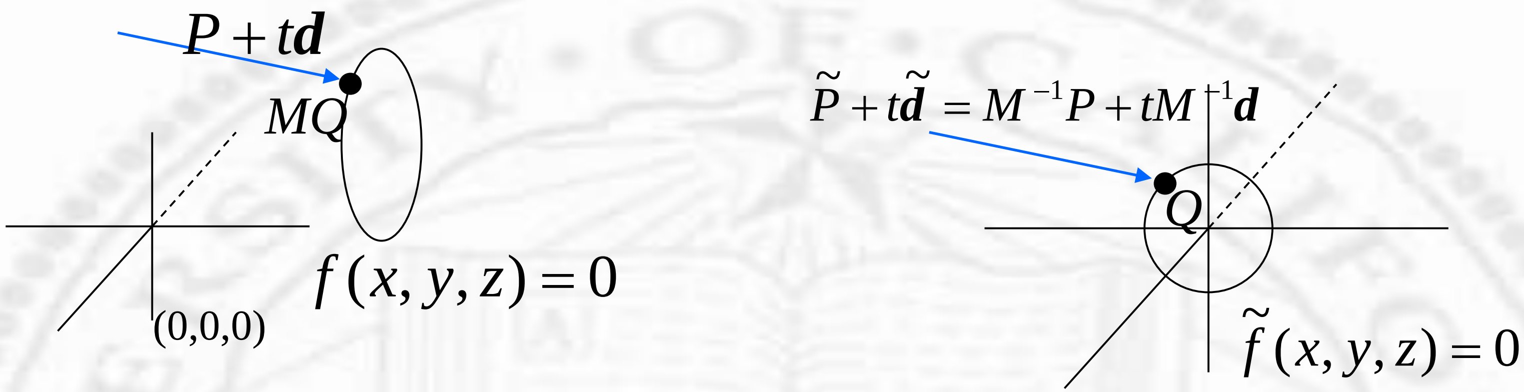
- There will be a reasonable t value, unless the denominator is zero (the line and the plane are parallel)
- But is the intersection point actually inside the polygon?

One Final Detail

- A cylinder $x^2+z^2-1=0$ is "simple" only in its own coordinate system
- Modeling transform can destroy that simplicity
- How to intersect with a general quadratic equation $ax^2+bxy+cy^2+dx+ey+f = 0$?



Object-Space Intersection



- World system
  - Complicated shape equations
  - $ax^2+bxy+cy^2+dx+ey+f = 0$
  - Ray equation is P+td always
- Object system
  - Simple shape equation
  - $x^2+z^2-1=0$

Shading – Normal Vector

- For illumination, you need the normal at the point of intersection in world space
- Two step process:
  - solving for point of intersection in the object's own space and computing normal there;
  - then transform the object space normal to the world space

Normal Vectors)

- Normal vectors need for shading
- A two-step process:
  - solving for point of intersection in the object's own space and computing normal there;
  - then transform the object space normal to the world space
  - Surface: f(x,y,z)=0, interior f(x,y,z)<0, then

$$\boldsymbol{n} = \nabla f(x, y, z)$$

$$\nabla f(x, y, z) = \left(\frac{\partial f}{\partial x}(x, y, z), \frac{\partial f}{\partial y}(x, y, z), \frac{\partial f}{\partial z}(x, y, z) \right)$$

Normal Vectors at Intersection Points

Sphere normal vector example (2/4)

For the sphere, the equation is

$$f(x,y,z) = x^2 + y^2 + z^2 - 1$$

The partial derivatives are

$$\frac{\partial f}{\partial x}(x, y, z) = 2x$$
$$\frac{\partial f}{\partial y}(x, y, z) = 2y$$
$$\frac{\partial f}{\partial z}(x, y, z) = 2z$$

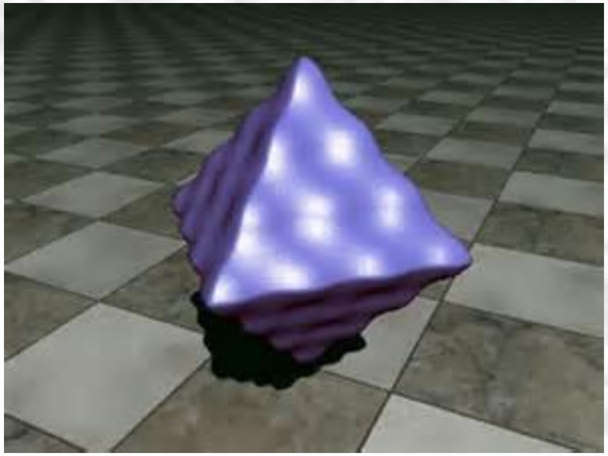
So the gradient is

$$\boldsymbol{n} = \nabla f(x, y, z) = (2x, 2y, 2z)$$

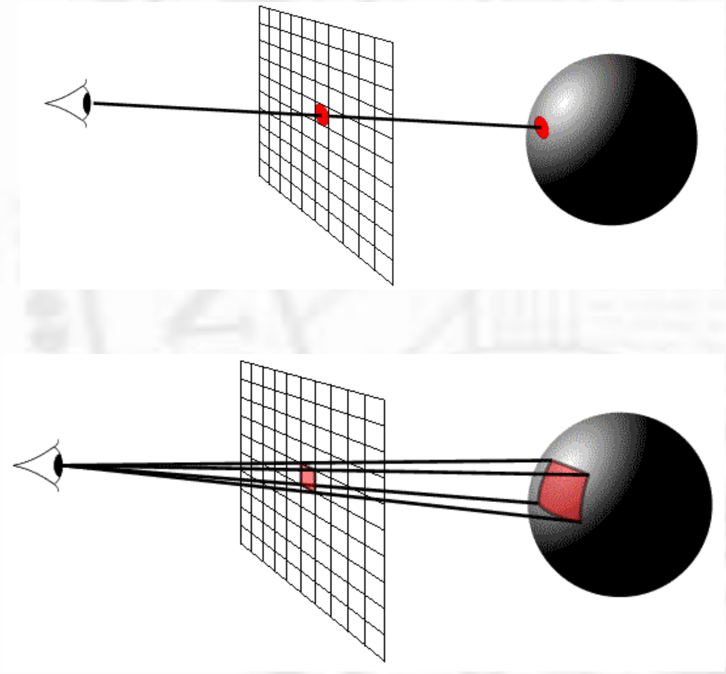
Normalize ***n*** before using in dot products!

In some degenerate cases, the gradient may be zero, and this method fails…use nearby gradient as a cheap hack

Special Effects



Practical Issues - Realism



Sampling

- In the simplest case, choose our sample points at pixel centers
- For better results, can *supersample*
  - e.g., at corners and at center
- Even better techniques do *adaptive sampling*: increase sample density in areas of rapid change (in geometry or lighting)
- With *stochastic sampling*, samples are taken probabilistically; converges faster than regularly spaced sampling
- For fast results, can *subsample*: fewer samples than pixels
  - take as many samples as time permits
  - *beam tracing*: track a bundle of neighboring rays together
- How to convert samples to pixels? Filter to get weighted average of samples

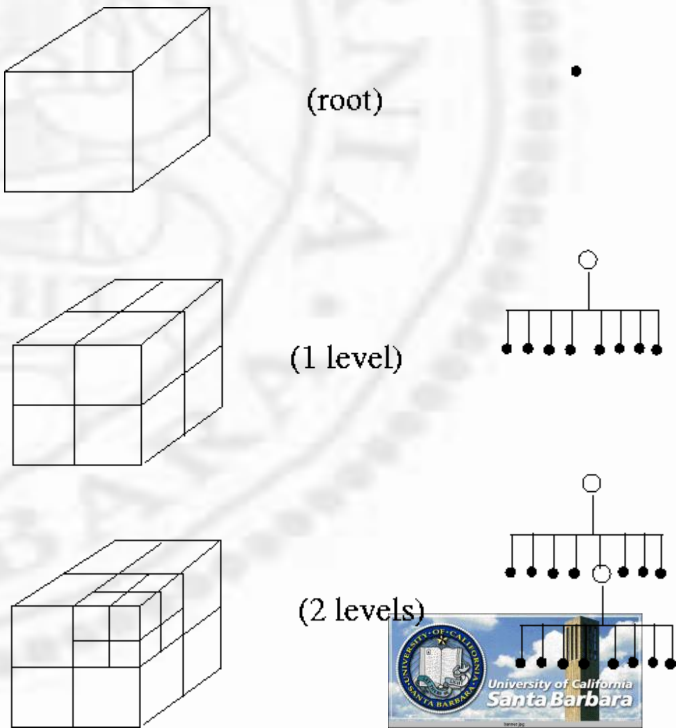
Practical Issue - Speed

- Very expensive
- Yet embarrassingly parallel
- Avoid unnecessary intersection tests

Space Partition

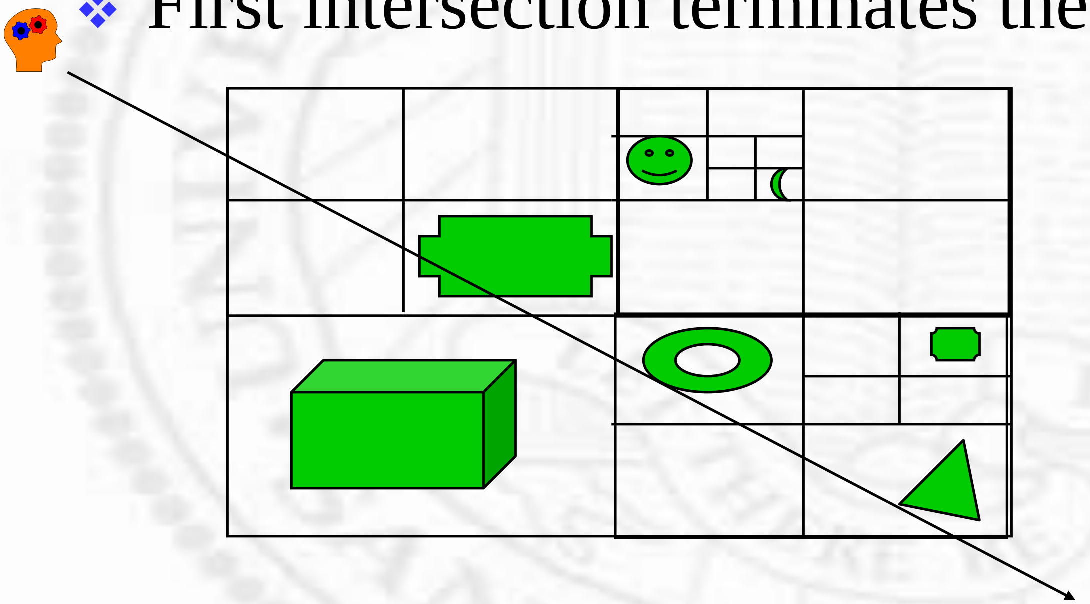
- During raytracing, the number of outstanding rays are usually over 100k.
- Building the Octree
- Create one cube represent the world and put all the triangles inside
- Recursively subdivide a cube into 2x2x2 cubes if the number of triangles is over a threshold

- Ray triangle intersection
- If the cube has children
- recursively intersects
- all its children cube
- intersect against all triangles



Spatial Partitioning

- Ray can be advanced from cell to cell
- Only those objects in the cells lying on the path of the ray need be considered
- First intersection terminates the search



Bounding Volume

$$Slab: aX + bY + cZ + d = 0$$

$$ray: \begin{cases} X = X_o + t\Delta X \\ Y = Y_o + t\Delta Y \\ Z = Z_o + t\Delta Z \end{cases}$$

$$t = -\frac{aX_o + bY_o + cZ_o + d}{a\Delta X + b\Delta Y + c\Delta Z} = \frac{A + D}{B}$$

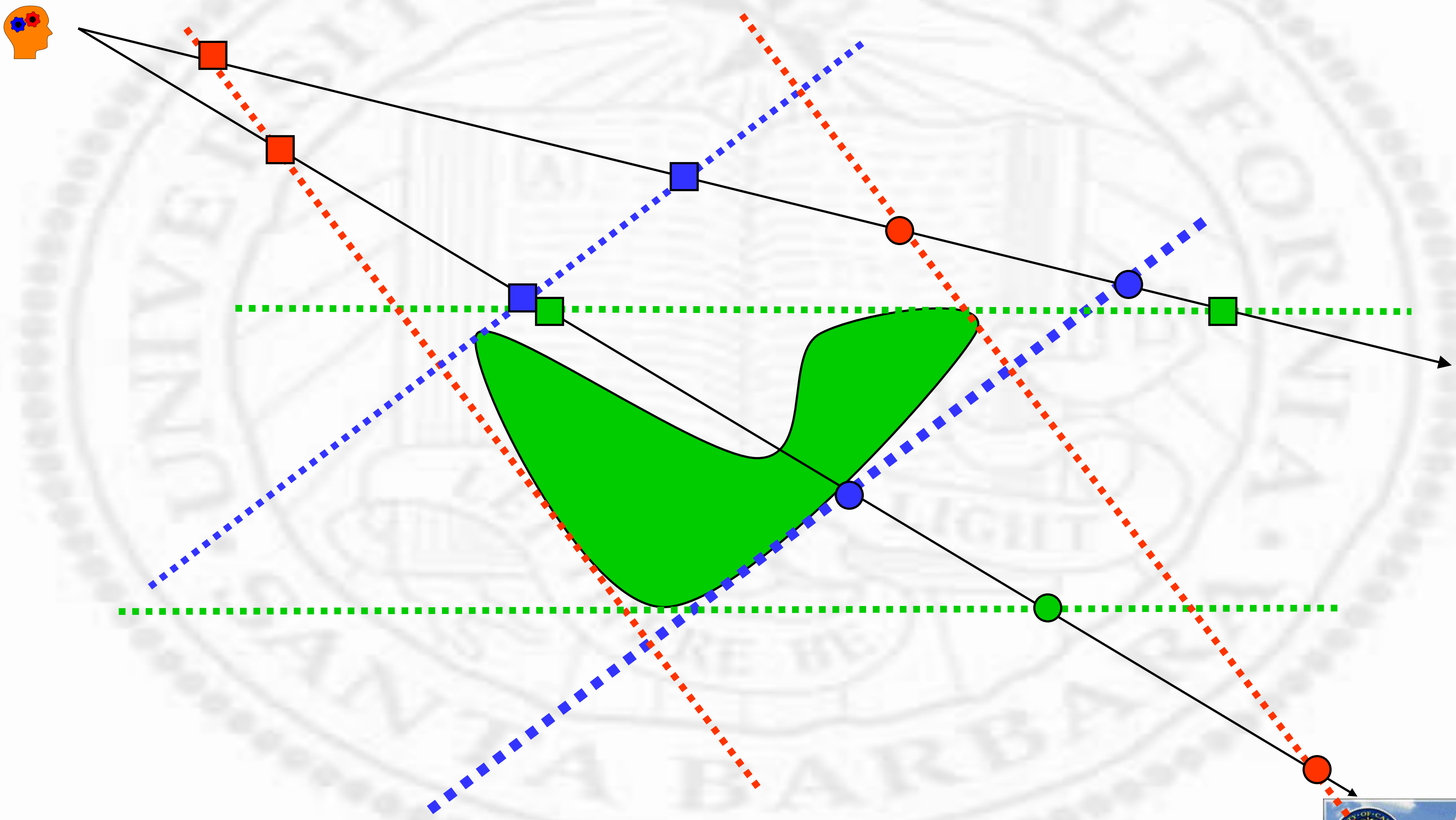
A: per ray per slab set

B: per ray per slab set

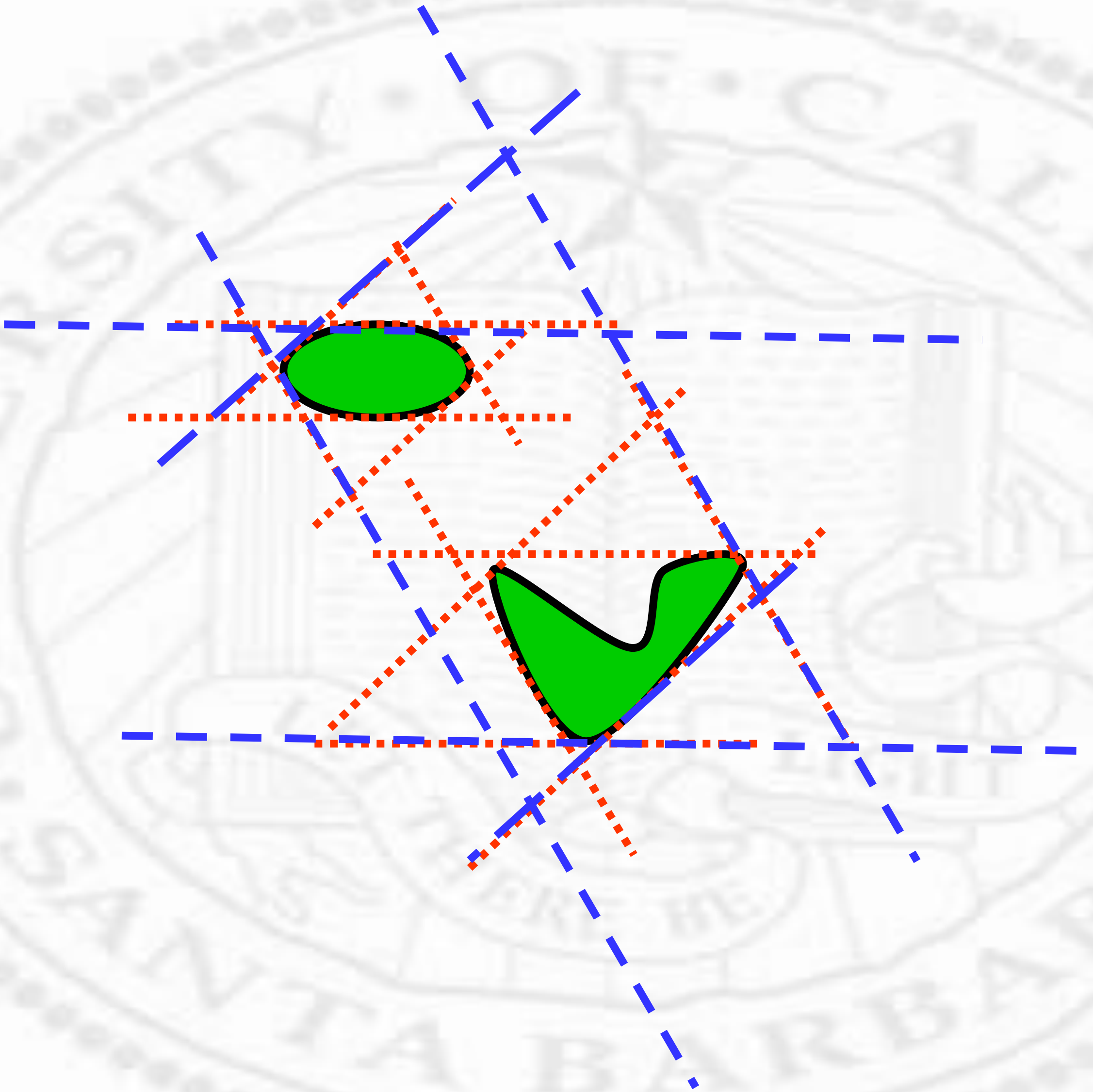
D: per slab

Bounding Volume (cont.)

- All the maximum (circle) intersections must be after all the minimum (square) intersections



Hierarchical Bounding Volume



Batch vs. Interactive

- Batch
  - Build a whole tree (<= certain depth)
  - At the leaf level
    - Nothing (background color)
    - Object (its intrinsic color)
    - Proceed backward to fill in the info
- Interactive
  - Build a tree to, say, depth 1
  - At leaf level
    - Nothing (background color)
    - Object (color computed from previous iteration)
    - Proceed backward to fill in the inof

  Extend to the next depth, and repeat