

10. Ray Tracing

- Generate a ray for each pixel and trace the ray **backwards** to its origin

10.1 Basic Ray Tracing Algorithm

- Trace each ray to **FIRST** object hit by ray
- Structure of basic ray tracing program:

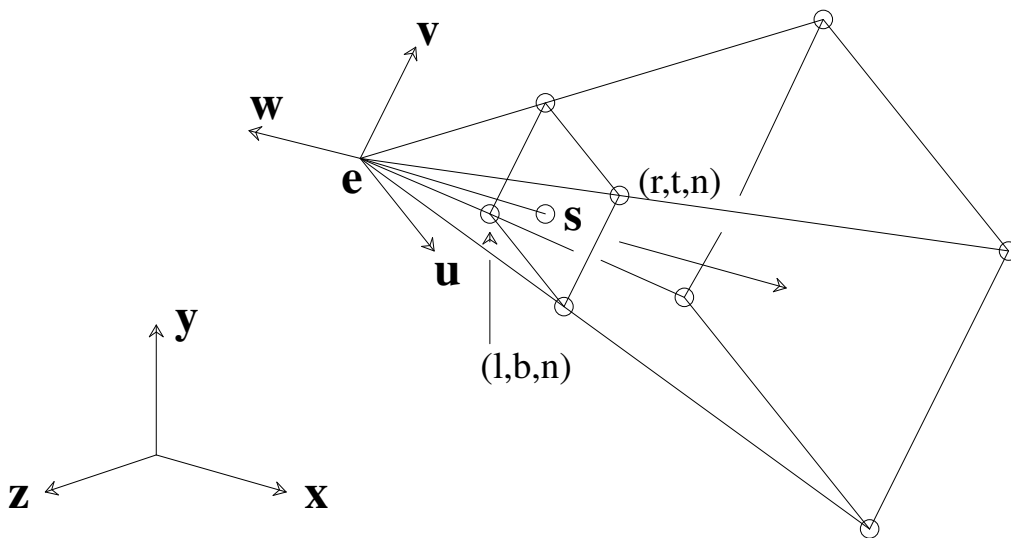
Compute \mathbf{u} , \mathbf{v} , \mathbf{w} basis vectors

for each pixel **do**

 compute viewing ray

 find first object hit by ray and its surface normal \mathbf{n}

 use material, light, and \mathbf{n} to compute pixel color



10.2 Computing Viewing Rays

- Parametric representation of a ray:

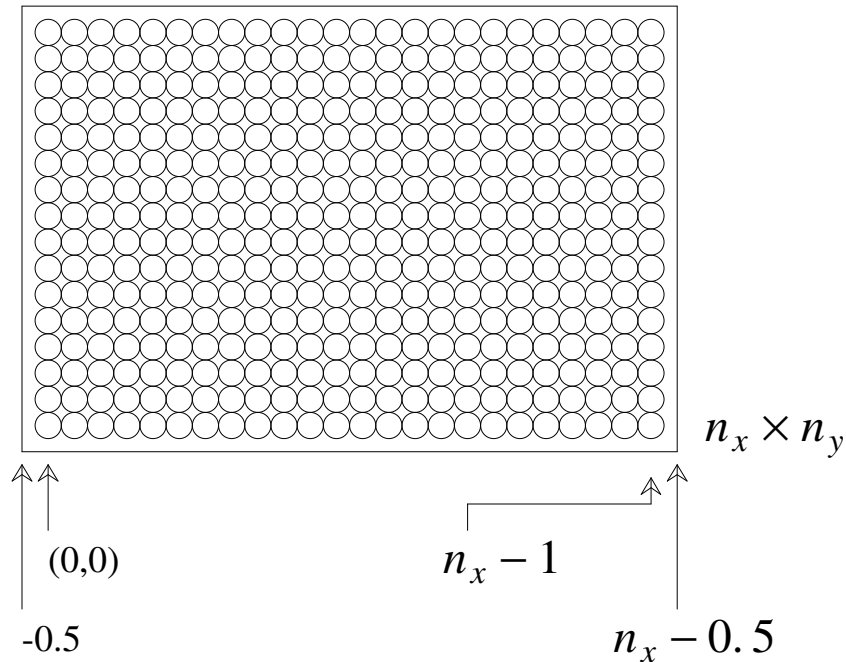
$$\mathbf{p}(t) = \mathbf{e} + t(\mathbf{s} - \mathbf{e}), \quad t \geq 0$$

where \mathbf{e} is eye point and \mathbf{s} is a point on the screen

- Computation of \mathbf{s} :

if resolution of the screen is $n_x \times n_y$ and coordinates of the 3D window on the view plan ($w = n$) are (l, b, n) and (r, t, n) defined in the uvw coordinate system, then

$$\mathbf{s} = \mathbf{e} + u_s \mathbf{u} + v_s \mathbf{v} + w_s \mathbf{w}$$



where

$$w_s = n$$

$$u_s = l + (r - l) \frac{i + 0.5}{n_x}$$

$$v_s = b + (t - b) \frac{j + 0.5}{n_y}$$

and (i, j) are pixel indices satisfying

$$0 \leq i \leq n_x - 1$$

$$0 \leq j \leq n_y - 1$$

- in matrix form:

$$\begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & x_e \\ 0 & 1 & 0 & y_e \\ 0 & 0 & 1 & z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & x_w & 0 \\ y_u & y_v & y_w & 0 \\ z_u & z_v & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_s \\ v_s \\ w_s \\ 1 \end{bmatrix}$$

10.3 Ray-Object Intersection

- Ray-sphere intersection

Ray: $\mathbf{p}(t) = \mathbf{e} + t\mathbf{d}$

Sphere: $f(\mathbf{p}) = (x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - R^2 = 0$

Set $f(\mathbf{p}(t)) = f(\mathbf{e} + t\mathbf{d}) = 0$. We get

$$(\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0$$

$$(\mathbf{e} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{e} + t\mathbf{d} - \mathbf{c}) - R^2 = 0$$

$$(\mathbf{d} \cdot \mathbf{d})t^2 + 2\mathbf{d} \cdot (\mathbf{e} - \mathbf{c})t + (\mathbf{e} - \mathbf{c}) \cdot (\mathbf{e} - \mathbf{c}) - R^2 = 0$$

a quadratic equation in t . Can solve it for t .

- Sphere normal at \mathbf{p} is $\mathbf{n} = 2(\mathbf{p} - \mathbf{c})$

- Ray-triangle intersection

Ray: $\mathbf{p}(t) = \mathbf{e} + t\mathbf{d}$

Plane containing triangle:

$$f(\beta, \gamma) = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

where \mathbf{a} , \mathbf{b} and \mathbf{c} are vertices of the triangle

Set

$$\mathbf{e} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}) \quad (*)$$

The hit point is in the triangle iff $\beta > 0$, $\gamma > 0$ and $\beta + \gamma < 1$.

Rewrite (*) as

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_e \\ y_a - y_e \\ z_a - z_e \end{bmatrix}$$

then use Cramer's rule to compute β , γ and t .

- Ray-polygon intersection

Ray: $\mathbf{p}(t) = \mathbf{e} + t\mathbf{d}$

Polygon: $\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_m$, and polygon normal \mathbf{n}

First, compute the intersection point between the ray and the plane containing the polygon
($f(\mathbf{p}) = (\mathbf{p} - \mathbf{p}_1) \cdot \mathbf{n} = 0$)

Set $f(\mathbf{e} + t\mathbf{d}) = 0$ and solve for t . We get

$$t = \frac{(\mathbf{p}_1 - \mathbf{e}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

Use this t we can compute \mathbf{p} . Then determine if \mathbf{p} is inside the polygon.

How do you tell if \mathbf{p} is inside the polygon???

10.4 A Ray-Tracing Program

- Using ray-tracing technique to produce images similar to the z-buffer or BSP-tree codes

```
for each pixel do
    compute viewing ray;
    if (ray hits an object with  $t \in [0, \infty)$ ) then
        compute n;
        evaluate lighting equation and set pixel to that
        color;
    else
        set pixel color to background color;
```

The statement "if ray hits an object" can be implemented as a function that tests for hits in the interval $t \in [t_0, t_1]$, as follows

```
hit = false
for each object o do
    if (object is hit at ray parameter  $t$  &  $t \in [t_0, t_1]$  )
        then
            hit = true
            hitobject = o
             $t_1 = t$ 
return hit
```

Note:

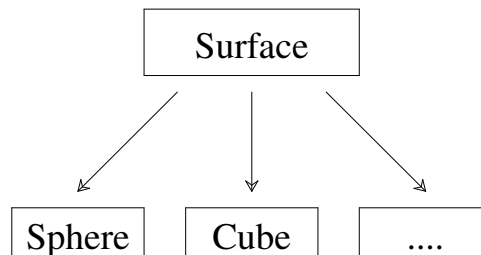
(1) In actual implementation, we should return a reference to the object that is hit. This can be done by passing a record/structure such as

hit-record rec

(2) should define a super class "surface" with subclasses such as *shpere*, *cone*, *cube*, ... to cover all the objects that could be intersected by a ray.

Object-Oriented Design for a Ray Tracer

- Develop a class hierarchy:



that not only include everything that can be hit by a ray, but also efficiency structures, such as bounding volume hierarchies

- Definition of the class *surface*:

```
class surface  
virtual bool hit(ray  $\mathbf{e} + t\mathbf{d}$  , real  $t_0$ , real  $t_1$ , hit-record rec)  
virtual box bounding-box()  
.  
.
```

(t_0, t_1) is the interval on the ray where hits will be returned.

- Implementation of "*bounding-box()*" for a sphere:

```
box sphere::bounding-box()
vector3 min = center - vector3(radius, radius, radius)
vector3 max = center + vector3(radius, radius, radius)
return box(min, max)
```

- Should also define a class called "material" to hold properties of an object and build links between objects and materials.

10.5 Shadows

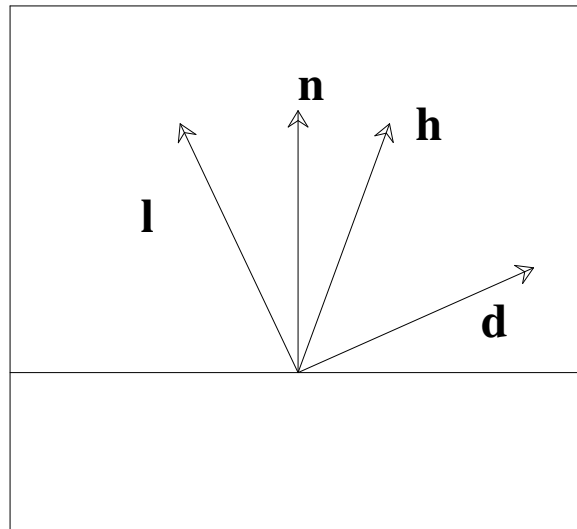
- How to add shadows into the above algorithm?
- Use *shadow rays* (different from *viewing rays*) to determine if a point is in shadow
- implement shadow rays for Phong lighting:

```
-----  
function raycolor( ray  $\mathbf{e} + t\mathbf{d}$ , real  $t_0$ , real  $t_1$  )  
hit-record rec, srec  
if ( scene  $\rightarrow$  hit( $\mathbf{e} + t\mathbf{d}$ ,  $t_0$ ,  $t_1$ , rec ) ) then  
     $\mathbf{p} = \mathbf{e} + \text{rec}.t\mathbf{d}$   
    color  $I = I_a \text{rec}.k_d$   
    if ( not scene  $\rightarrow$  hit( $\mathbf{p} + s\mathbf{l}$ ,  $\varepsilon$ ,  $\infty$ , srec ) ) then  
        vector3  $\mathbf{h} = \text{normalized}(\text{normalized}(\mathbf{l}) + \text{normalized}(-\mathbf{d}))$   
         $I = I + I_p \text{rec}.k_d \max(0, \text{rec}.\mathbf{n} \cdot \mathbf{l}) + I_p \text{rec}.W(\theta)(\mathbf{h} \cdot \text{rec}.\mathbf{n})^{\text{rec}.q}$   
    return  $I$   
else  
    return background-color  
-----
```

Note that usually the specular reflection is computed as

$$I = I_p \cdot W(\theta) \cdot \cos \alpha$$

where α is the angle between \mathbf{n} and \mathbf{r} with $\mathbf{r} = -\mathbf{l} + 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n}$.



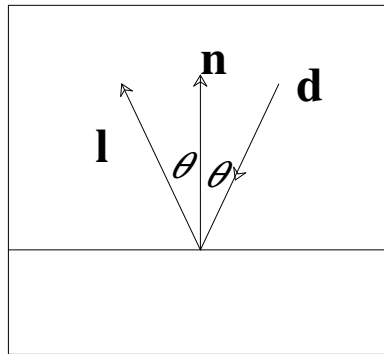
An alternative is to use the unit vector halfway between \mathbf{l} and \mathbf{d} to compute $\cos \alpha$:

$$\mathbf{h} = \frac{\mathbf{d} + \mathbf{l}}{\|\mathbf{d} + \mathbf{l}\|}$$

10.6 Specular Reflection

- Add *specular reflection* to ray tracing
- For the view direction \mathbf{d} , compute its specular reflection \mathbf{r}

$$\mathbf{r} = \mathbf{d} + 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$$



- Use \mathbf{r} to recursively compute contribution from specular reflection

$$color \ I = I + I_s \ raycolor(\mathbf{p} + s\mathbf{r}, \epsilon, \infty)$$

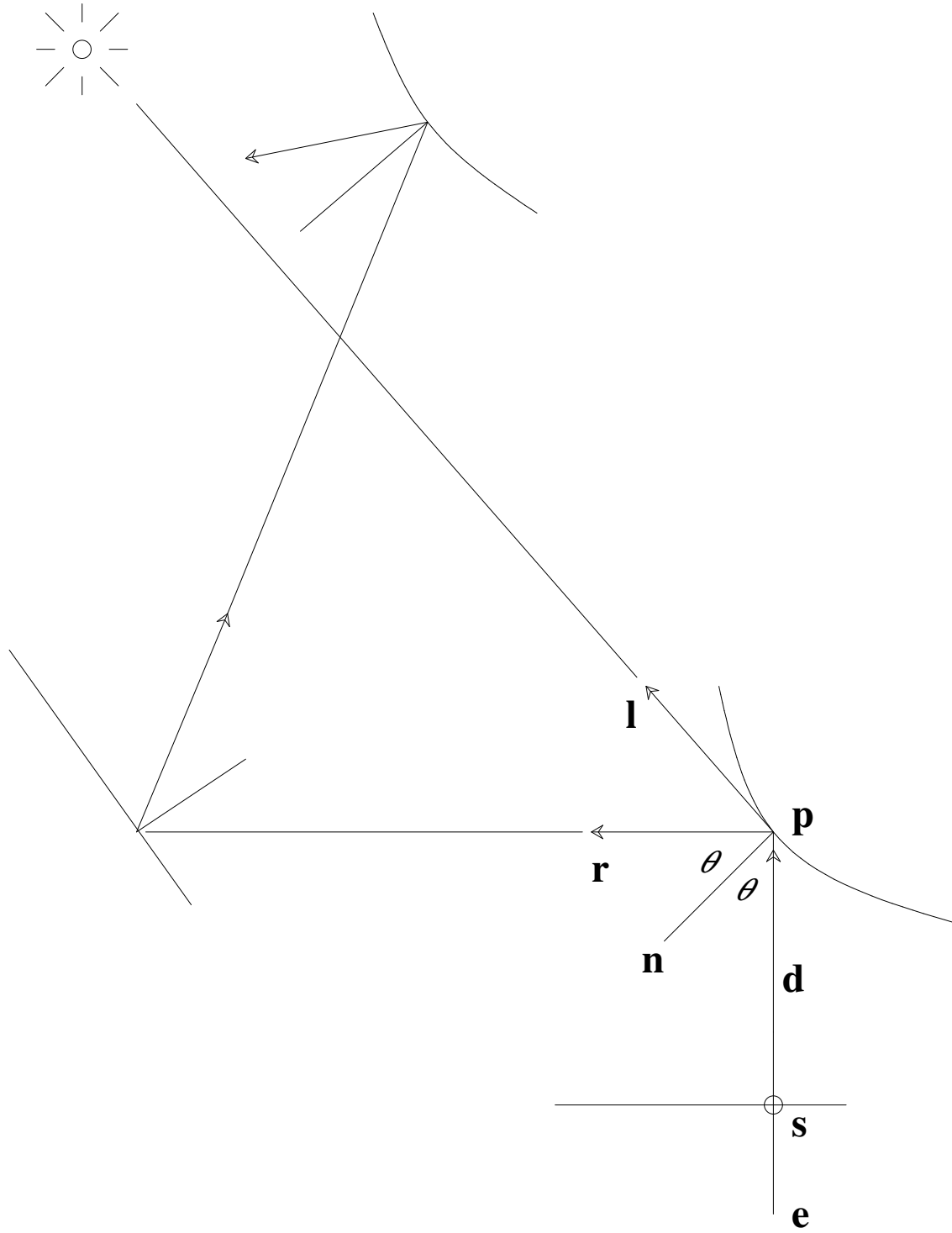
where I_s is the specular RGB color.

- To avoid an infinite loop in the above recursive call, set up a maximum recursive depth.

- Include specular reflection in "raycolor" (making it a real ray-tracing process):

```
function raycolor( ray  $\mathbf{e} + t\mathbf{d}$ , real  $t_0$ , real  $t_1$  )
hit-record rec, srec
if ( scene→hit( $\mathbf{e} + t\mathbf{d}$ ,  $t_0$ ,  $t_1$ , rec ) ) then
     $\mathbf{p} = \mathbf{e} + \text{rec.}t\mathbf{d}$ 
    color  $I = I_a \text{ rec.}k_d$ 
    if ( not scene→hit( $\mathbf{p} + s\mathbf{l}$ ,  $\varepsilon$ ,  $\infty$ , srec ) ) then
        vector3  $\mathbf{h} = \text{normalized( normalized}(\mathbf{l}) + \text{normal-}$ 
             $\text{ized}(-\mathbf{d}))$ 
         $I = I + I_p \text{ rec.}k_d \max(0, \text{rec.} \mathbf{n} \cdot \mathbf{l}) + I_p \text{ rec.} W(\theta)(\mathbf{h} \cdot \text{rec.} \mathbf{n})^{\text{rec.}q}$ 
         $\mathbf{r} = -\text{rec.} \mathbf{d} + 2(\text{rec.} \mathbf{d} \cdot \text{rec.} \mathbf{n})\text{rec.} \mathbf{n}$ 
         $I = I + k_s \text{ raycolor}(\mathbf{p} + s\mathbf{r}, \varepsilon, \infty)$ 
    return  $I$ 
else
    return background-color
```

- Illustration:

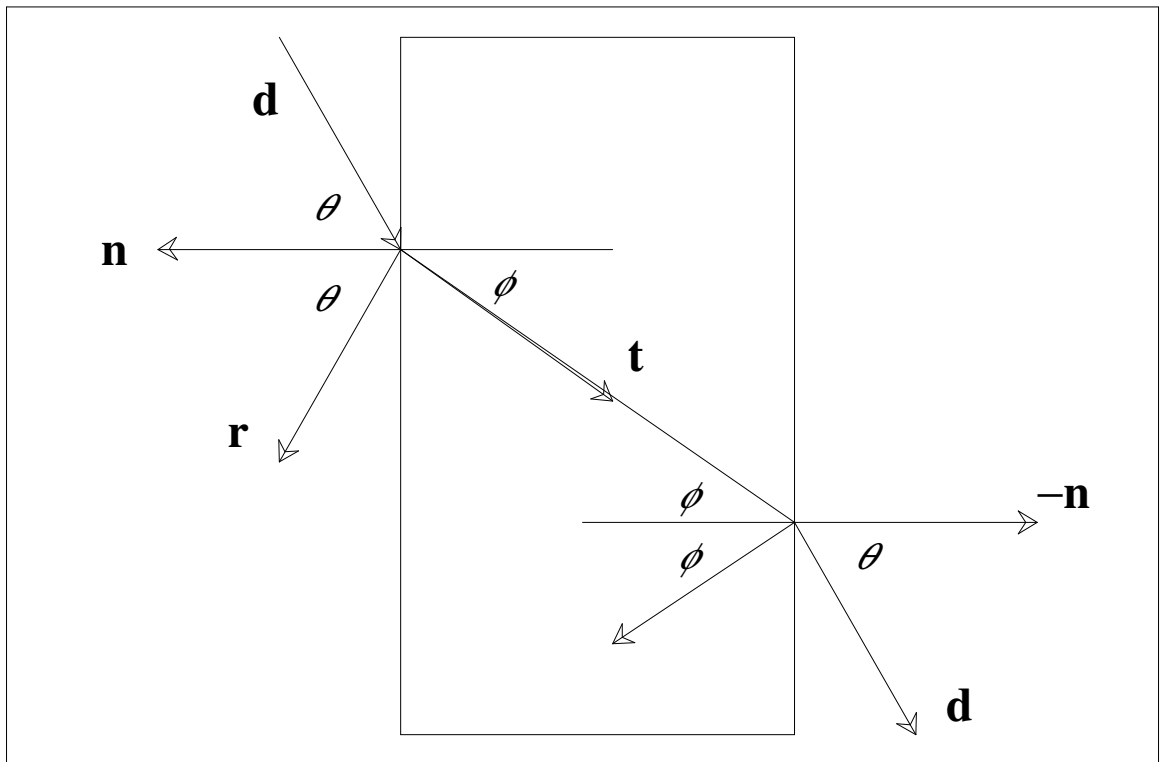


10.7 Refraction

- Add *refraction* to ray tracing
- Light is refracted when it enters a dielectric (transparent material that refracts light). The refraction follows *Snell's law*

$$n \sin \theta = n_t \sin \phi$$

where n and n_t are the refractive indices of the objects



- $\cos \phi$ can be computed as follows:

$$\cos^2 \phi = 1 - \frac{n^2(1 - \cos^2 \theta)}{n_t^2}$$

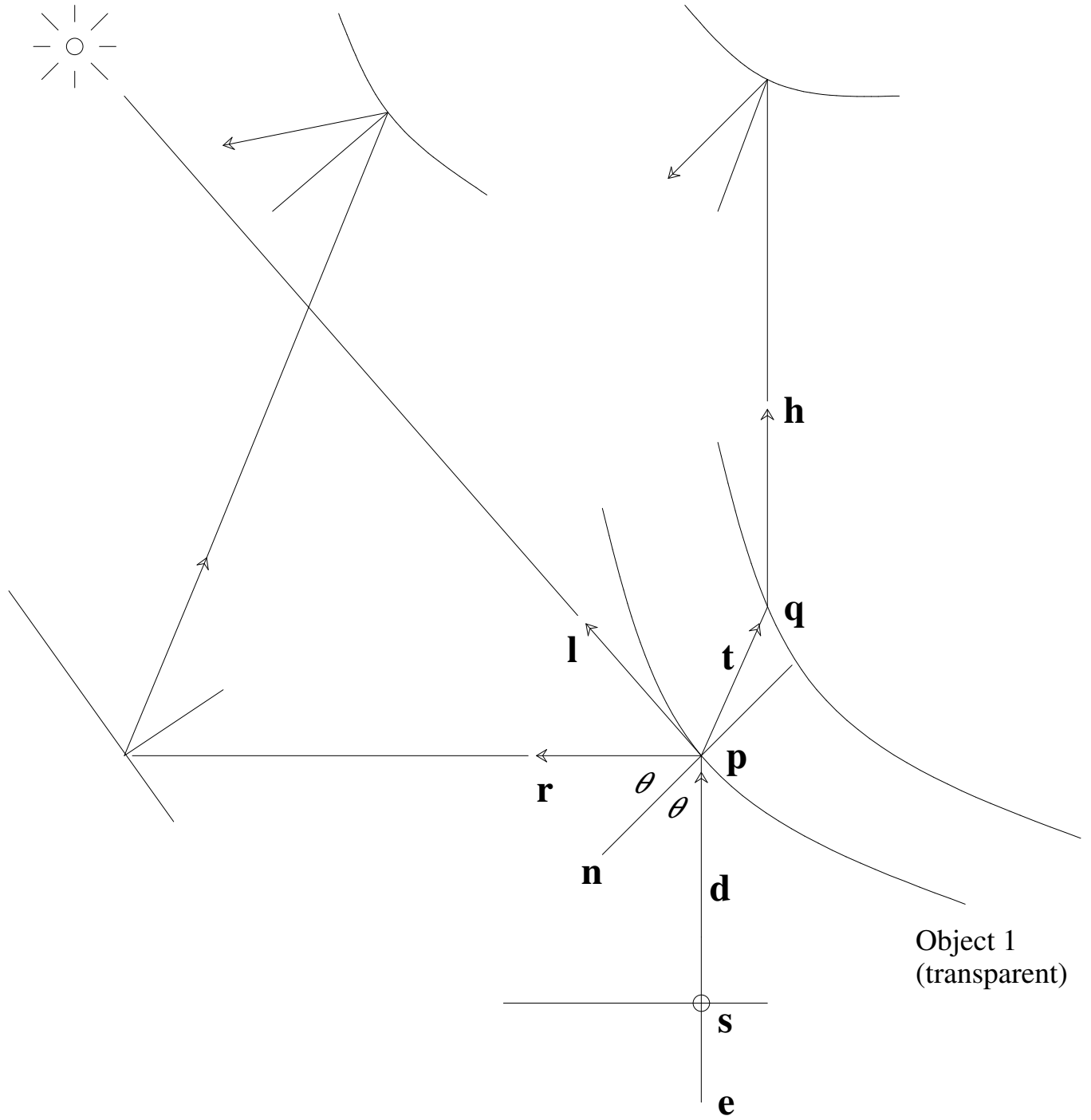
- The refraction vector \mathbf{t} can be computed as follows:

$$\mathbf{t} = \frac{n(\mathbf{d} - \mathbf{n}(\mathbf{d} \cdot \mathbf{n}))}{n_t} - \mathbf{n} \sqrt{1 - \frac{n^2(1 - (\mathbf{d} \cdot \mathbf{n})^2)}{n_t^2}}$$

when the number under the square root is negative, there is no refracted ray and all the energy is reflected.

- How should *refraction* be included in "raycolor"?
Conceptually (see figure on next page),
 - 1) Compute the vector \mathbf{t}
 - 2) Compute the intersection point \mathbf{q}
 - 3) Compute the vector \mathbf{h}
 - 4) Call "raycolor" with the ray $\mathbf{q} + u\mathbf{h}$ as the parameter to compute the intensity I' of the object at \mathbf{q}
 - 5) Combine I' with I to get the final color for s

- Illustration:

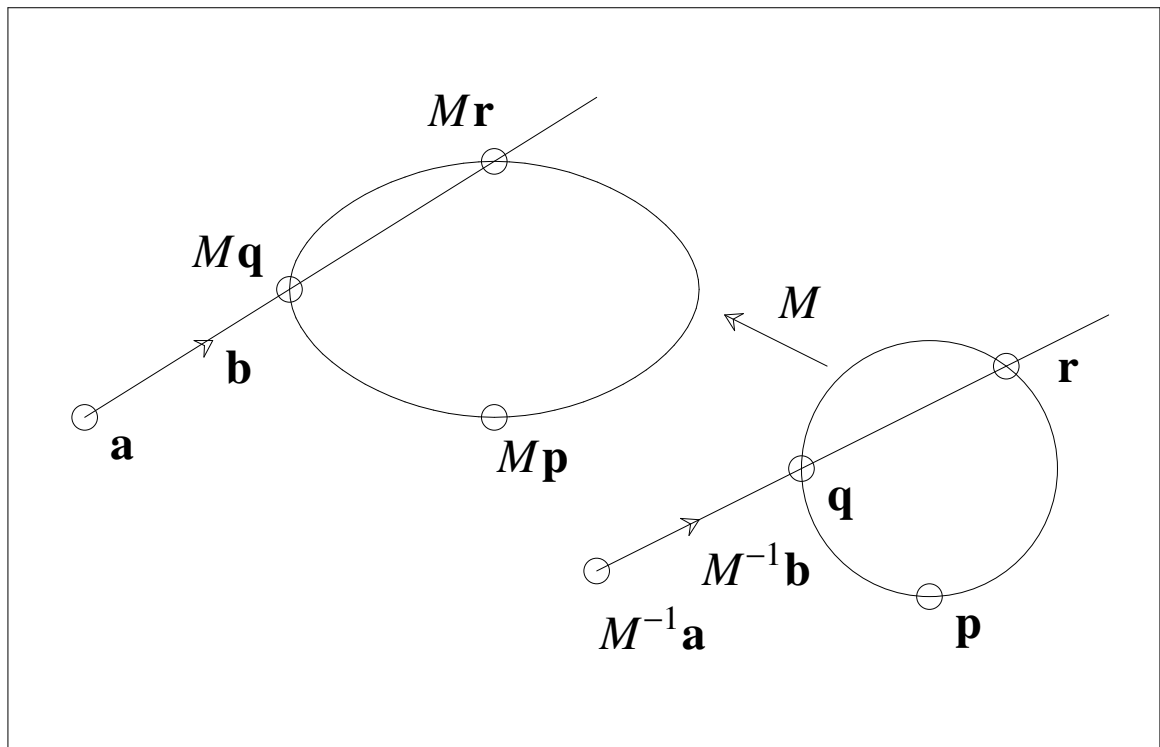


- Include refraction in "raycolor" (making it a 100 ray-tracing process):

```
function raycolor( ray  $\mathbf{e} + t\mathbf{d}$ , real  $t_0$ , real  $t_1$  )
hit-record rec, srec
if ( scene→hit( $\mathbf{e} + t\mathbf{d}$ ,  $t_0$ ,  $t_1$ , rec ) then
     $\mathbf{p} = \mathbf{e} + \text{rec}.t\mathbf{d}$ 
    color  $I = I_a \text{rec}.k_d$ 
    if (not scene→hit( $\mathbf{p} + s\mathbf{l}$ ,  $\varepsilon$ ,  $\infty$ , srec )) then
        vector3  $\mathbf{h} = \text{normalized}(\text{normalized}(\mathbf{l}) + \text{normalized}(-\mathbf{d}))$ 
         $I = I + I_p \text{rec}.k_d \max(0, \text{rec}.\mathbf{n} \cdot \mathbf{l}) + I_p \text{rec}.W(\theta)(\mathbf{h} \cdot \text{rec}.\mathbf{n})^{\text{rec}.q}$ 
         $\mathbf{r} = -\text{rec}.\mathbf{d} + 2(\text{rec}.\mathbf{d} \cdot \text{rec}.\mathbf{n})\text{rec}.\mathbf{n}$ 
         $I = I + k_s \text{raycolor}(\mathbf{p} + s\mathbf{r}, \varepsilon, \infty)$ 
        compute the vector  $\mathbf{t}$ 
        find the intersection point  $\mathbf{q}$ 
        compute the vector  $\mathbf{h}$ 
         $I = (1 - k_t)I + k_t \text{raycolor}(\mathbf{q} + u\mathbf{h}, \varepsilon, \infty)$ 
    return  $I$ 
else
    return background-color
```

10.8 Instancing

- How should we ray trace an instance of an object transformed by a matrix M ?
- The matrix could be the accumulation of several transformations
- Ray tracing the transformed object can be done in the space of the untransformed object



- One needs to transform the ray to the space of the untransformed object, do the tracing there, then transform the results to the space where the transformed object is in.
- If we define an instance class of type *surface*, we need to create a *hit* function:

```
-----  
instance::hit(ray  $\mathbf{a} + t\mathbf{b}$  , real  $t_0$ , real  $t_1$ , hit-record  $rec$ )  
ray  $\mathbf{r}' = M^{-1}\mathbf{a} + tM^{-1}\mathbf{b}$   
if (base-object→hit( $\mathbf{r}'$ ,  $t_0$ ,  $t_1$ ,  $rec$ )) then  
     $rec.\mathbf{n} = (M^{-1})^T rec.\mathbf{n}$   
    return true  
else  
    return false  
-----
```

10.9 Sub-Linear Ray-Object Intersection

- Three techniques to speed up ray-tracing:
 - *bounding volume hierarchies*
 - *uniform spatial subdivision*
 - *binary space partitioning*

10.9.1 Bounding Boxes

- Only need to know if the ray hits the box; do not need to know where
- Bounding box: $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$
- Ray: $\mathbf{e} + t\mathbf{d}$
- Compute the ray parameters

$$t_{x\min}, \quad t_{x\max}, \quad t_{y\min}, \quad t_{y\max}$$

where the ray hits the line

$$x = x_{\min}, \quad x = x_{\max}, \quad y = y_{\min}, \quad y = y_{\max},$$

respectively. The ray hits the box iff the intervals $[t_{x\min}, t_{x\max}]$ and $[t_{y\min}, t_{y\max}]$ overlap.

10.9.2 Hierarchical Bounding Boxes

- Bounding boxes can be nested by creating boxes around subsets of the nodes
- Use top-down ray-box testing and bottom-up parameter-merging

