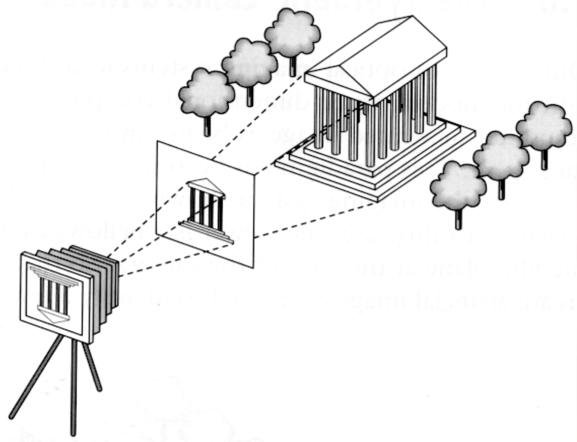
Ray Tracing

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 form the light sources to the eye, but the physics is invariant under path reversal (reciprocity).

Synthetic pinhole camera

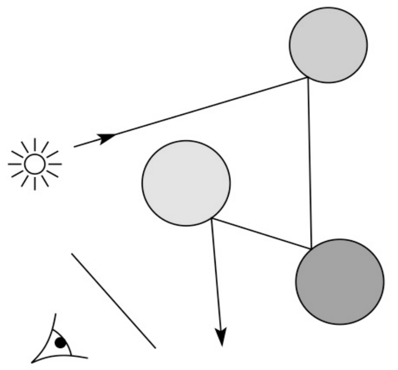
- The most common imaging model in graphics is the synthetic pinhole camera: light rays are collected through an infinitesimally small hole and recorded on an **image plane**.



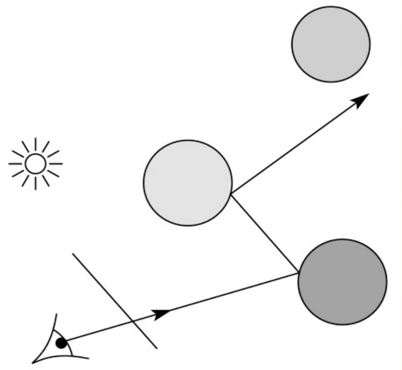
- For convenience, the image plane is usually placed in front of the camera, giving a non-inverted 2D projection (image).
- Viewing rays emanate from the **center of projection** (COP) at the center of the lens (or pinhole).
- The image of an object point *P* is at the intersection of the viewing ray through *P* and the image plane.

Eye vs. light ray tracing

- Where does light begin?
- At the light: light ray tracing (a.k.a., forward ray tracing or photon tracing)



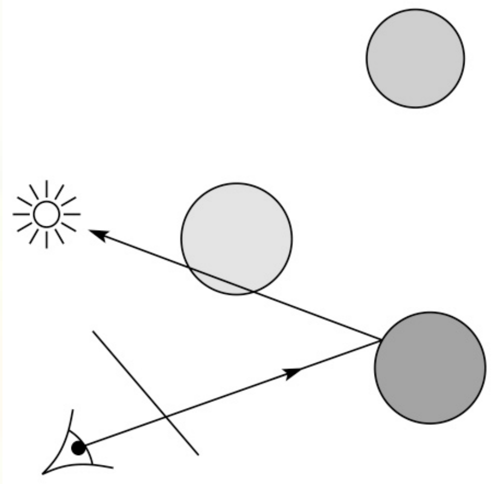
- At the eye: eye ray tracing (a.k.a., backward ray tracing)



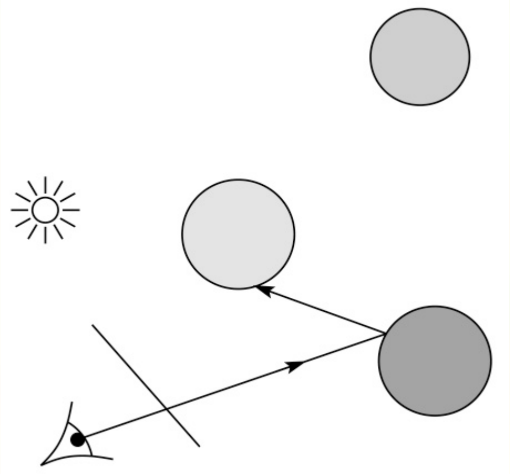
- We will generally follow rays from the eye into the scene.

Precursors to ray tracing

- Local illumination
  - Cast one eye ray,
    then shade according to light

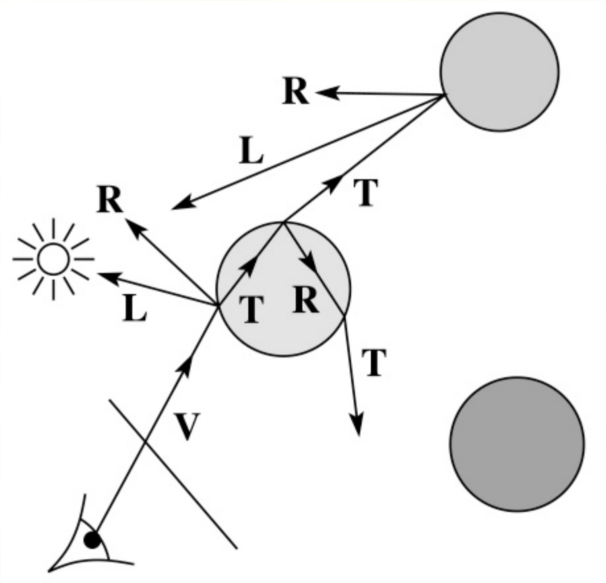


- Appel (1968)
  - Cast one eye ray + one ray to light



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

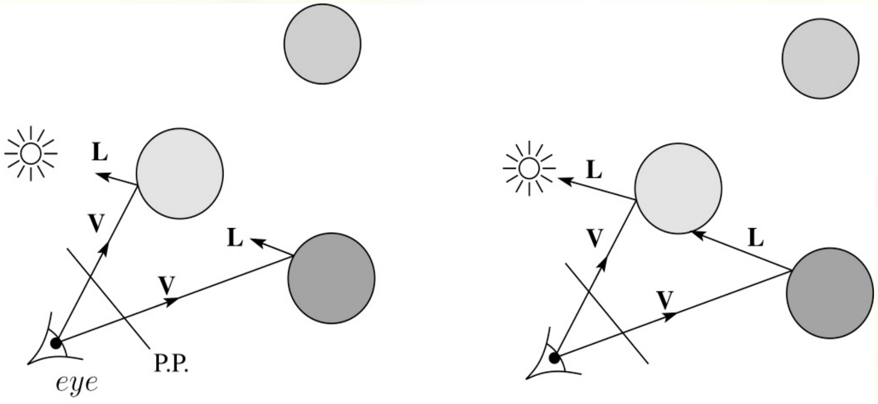


- Algorithm:

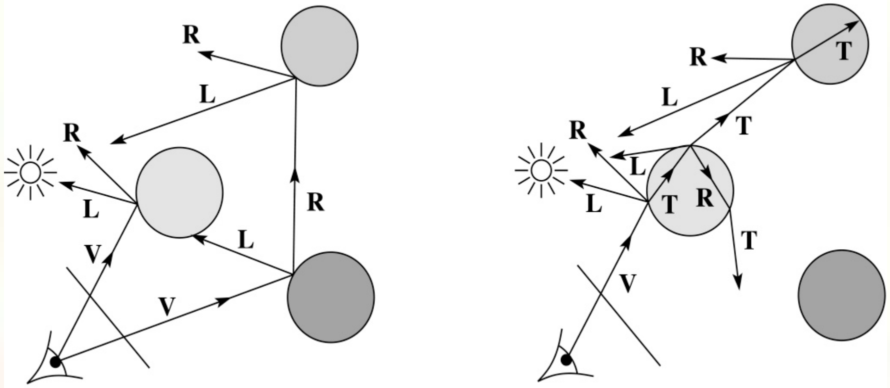
1. For each pixel, trace a **primary ray** in direction **V** to the first visible surface.
2. For each intersection, trace **secondary rays**:
   - **Shadow rays** in directions $\mathbf{L}_i$ to light sources
   - **Reflected ray** in direction **R**.
   - **Refracted ray** or **transmitted ray** in direction **T**.

Whitted algorithm (cont'd)

Let's look at this in stages:

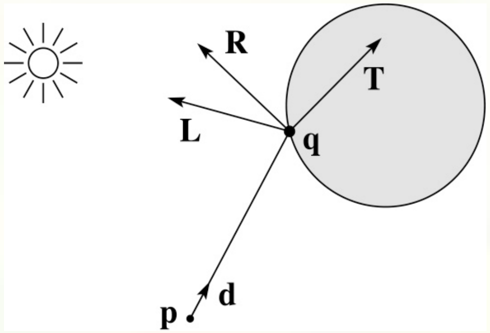


Primary rays Shadow rays



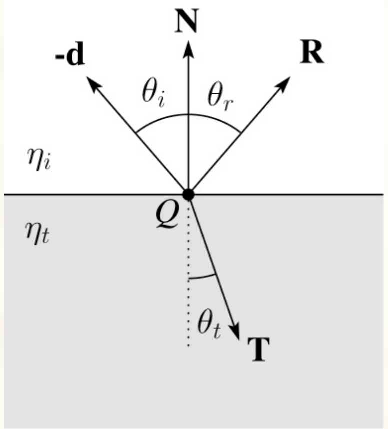
Reflection rays Refracted rays

Shading



- A ray is defined by an origin **P** and a unit direction **d** and is parameterized by *t*:
  - $P + t\mathbf{d}$
- Let $I(P, \mathbf{d})$ be the intensity seen along that ray. Then:
- $I(P, \mathbf{d}) = I_{\text{direct}} + I_{\text{reflected}} + I_{\text{transmitted}}$
- where
  - I_{direct} is computed from the Phong model
  - $I_{\text{reflected}} = k_r I\,(Q, \mathbf{R})$
  - $I_{\text{transmitted}} = k_t I\,(Q, \mathbf{T})$
- Typically, we set $k_r = k_s$ and $k_t = 1 - k_s$.

Reflection and transmission

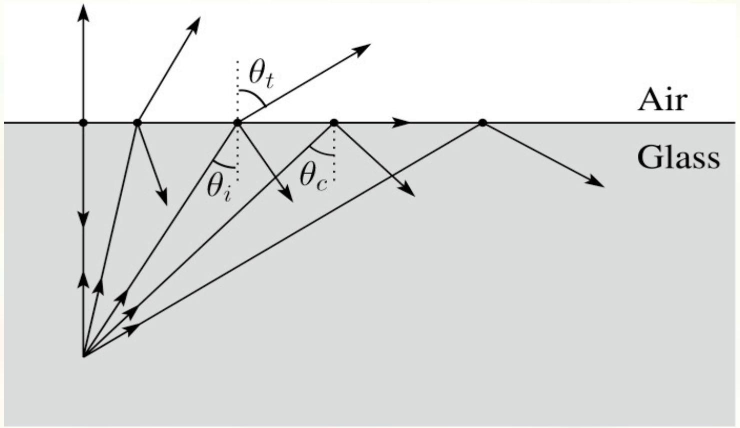


- Law of reflection:
  - $\theta_i = \theta_r$
- Snell's law of refraction:
  - $\eta_\mathrm{i} \sin\theta_\mathrm{I} = \eta_\mathrm{t} \sin \theta_\mathrm{t}$
- where η_i , η_t are **indices of refraction**.

Total Internal Reflection

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

- What happens 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.



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
        $S = pixelToWorld(i,j)$
        $P = \mathbf{COP}$
        $\mathbf{d} = (S - P)/\| S - P \|$
        $I(i,j) = traceRay(scene, P, \mathbf{d})$
    end for
end function
```

Ray-tracing pseudocode, cont'd

```
function traceRay(scene, P, d):
    (t, N, mtrl)  ← scene.intersect (P, d)
    Q ← ray (P, d) evaluated at t
    I = shade(q, N, mtrl, scene)
    R = reflectDirection(N, -d)
    I ← I + mtrl.$k_r$ ∗ traceRay(scene, Q, R)
    if ray is entering object then
        n_i = index_of_air
        n_t = mtrl.index
    else
        n_i = mtrl.index
        n_t = index_of_air
    if (mtrl.k_t > 0 and notTIR (n_i, n_t, N, -d)) then
        T = refractDirection (n_i, n_t, N, -d)
        I ← I + mtrl.$k_t$ ∗ traceRay(scene, Q, T)
    end if
    return I
end function
```

Terminating recursion

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

Shading pseudocode

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

```
function shade(mtrl, scene, Q, N, d):
    I ← mtrl.$k_e$ + mtrl. $k_a$ * scene->$I_a$
    for each light source $\lambda$ do:
        atten = $\lambda$ -> distanceAttenuation( Q ) *
        $\lambda$ -> shadowAttenuation( scene, Q  )
        I ← I + atten*(diffuse term + spec term)
    end for
    return I
end function
```

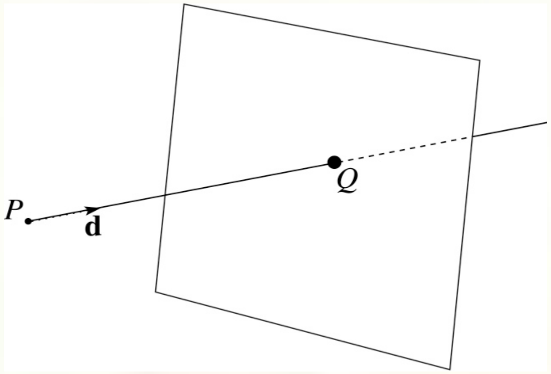
Shadow attenuation

- 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, P)
    d = (λ.position - P).normalize()
    (t, N, mtrl) ← scene.intersect(P, d)
    Q ← ray(t)
    if Q is before the light source then:
        atten = 0
    else
        atten = 1
    end if
    return atten
end function
```

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

Ray-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, **n** = [*a b c*]^T. Thus, we can re-write the plane equation as:

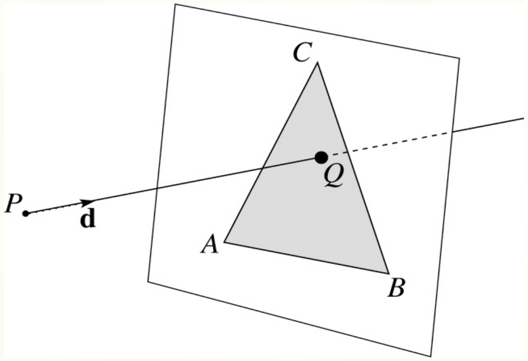
$$\mathbf{n} \bullet \mathbf{p}(t) + d = 0$$

$$\mathbf{n} \bullet \left(\mathrm{P} + t\mathbf{d}\right) + d = 0$$

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

$$t = -\frac{\mathbf{n} \bullet P + d}{\mathbf{n} \bullet \mathbf{d}}$$

Ray-triangle intersection



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

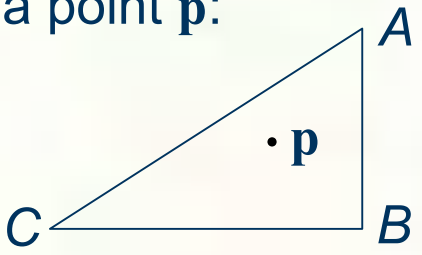
$$\mathbf{n} = (A - C) \times (B - C)$$

$$d = -(\mathbf{n} \bullet A)$$

- Then, we need to decide if the point is inside or outside of the triangle.
  - Solution 1: compute barycentric coordinates from 3D points.
  - What do you do with the barycentric coordinates?

Barycentric coordinates

A set of points can be used to create an affine frame. Consider a triangle *ABC* and a point **p**:



We can form a frame with an origin *C* and the vectors from *C* to the other vertices:

$$\mathbf{u} = A - C \quad \mathbf{v} = B - C \quad \mathbf{t} = C$$

We can then write *P* in this coordinate frame $\mathbf{p} = \alpha\mathbf{u} + \beta\mathbf{v} + \mathbf{t}$

The coordinates (α, β, γ) are called the **barycentric coordinates** of **p** relative to *A*, *B*, and *C*.

Computing barycentric coordinates

For the triangle example we can compute the barycentric coordinates of P:

$$\alpha A + \beta B + \gamma C = \begin{bmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} \mathbf{p_x} \\ \mathbf{p_y} \\ 1 \end{bmatrix}$$

Cramer’s rule gives the solution:

$$\alpha = \frac{\begin{Vmatrix} \mathbf{p_x} & B_x & C_x \\ \mathbf{p_y} & B_y & C_y \\ 1 & 1 & 1 \end{Vmatrix}}{\begin{Vmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ 1 & 1 & 1 \end{Vmatrix}} \quad \beta = \frac{\begin{Vmatrix} A_x & \mathbf{p_x} & C_x \\ A_y & \mathbf{p_y} & C_y \\ 1 & 1 & 1 \end{Vmatrix}}{\begin{Vmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ 1 & 1 & 1 \end{Vmatrix}} \quad \gamma = \frac{\begin{Vmatrix} A_x & B_x & \mathbf{p_x} \\ A_y & B_y & \mathbf{p_y} \\ 1 & 1 & 1 \end{Vmatrix}}{\begin{Vmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ 1 & 1 & 1 \end{Vmatrix}}$$

Computing the determinant of the denominator gives:

$$B_x C_y - B_y C_x + A_y C_x - A_x C_y + A_x B_y - A_y B_x$$

Cross products

Consider the cross-product of two vectors, **u** and **v**. What is the geometric interpretation of this cross-product?

A cross-product can be computed as:

$$
\begin{aligned}
\mathbf{u} \times \mathbf{v} &= \begin{Vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{Vmatrix} \\
&= (u_y v_z - u_z v_y)\mathbf{i} + (u_z v_x - u_x v_z)\mathbf{j} + (u_x v_y - u_y v_x)\mathbf{k} \\
&= \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}
\end{aligned}
$$

What happens when **u** and **v** lie in the *x*-*y* plane? What is the area of the triangle they span?

Barycentric coords from area ratios

Now, let’s rearrange the equation from two slides ago:

$$B_xC_y - B_yC_x + A_yC_x - A_xC_y + A_xB_y - A_yB_x$$
$$= (B_x - A_x)(C_y - A_y) - (B_y - A_y)(C_x - A_x)$$

The determinant is then just the *z*-component of

(B-A) × (C-A), which is two times the area of triangle *ABC*!

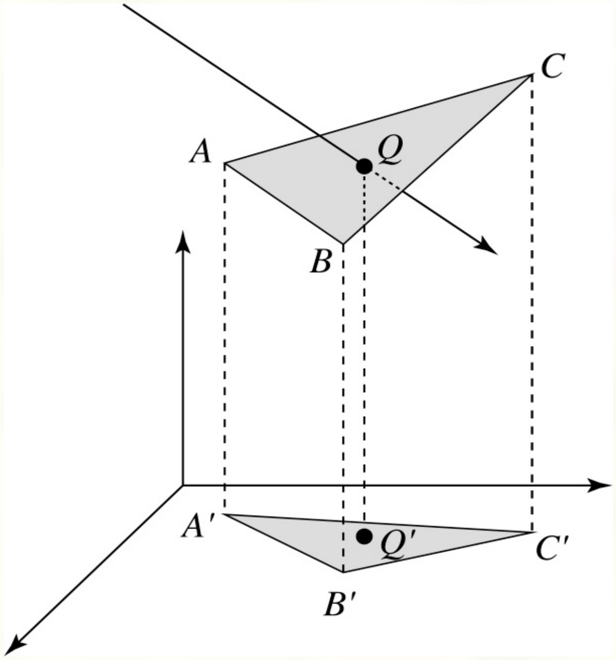
Thus, we find:

$$\alpha = \frac{\text{SArea}(\mathbf{p}BC)}{\text{SArea}(ABC)} \quad \beta = \frac{\text{SArea}(A\mathbf{p}C)}{\text{SArea}(ABC)} \quad \gamma = \frac{\text{SArea}(AB\mathbf{p})}{\text{SArea}(ABC)}$$

Where SArea(RST) is the signed area of a triangle, which can be computed with cross-products.

Ray-triangle intersection

- Solution 2: project down a dimension and compute barycentric coordinates from 2D points.



- Why is solution 2 possible? Why is it legal? Why is it desirable? Which axis should you "project away"?

Interpolating vertex properties

- The barycentric coordinates can also be used to interpolate vertex properties such as:
  - material properties
  - texture coordinates
  - normals
- For example:

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

- Interpolating normals, known as Phong interpolation, gives triangle meshes a smooth shading appearance. (Note: don't forget to normalize interpolated normals.)

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!

Intersecting with xformed geometry

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