6. Ray Tracing

Reading

Required:

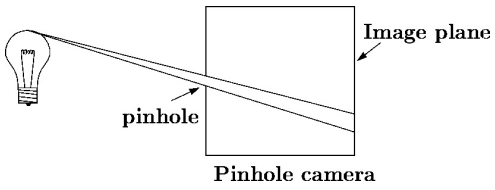
- Watt, sections 1.3-1.4, 12.1-12.5.1.
- T. Whitted. An improved illumination model for shaded display. Communications of the ACM 23(6), 343-349, 1980. [In the reader.]

Further reading:

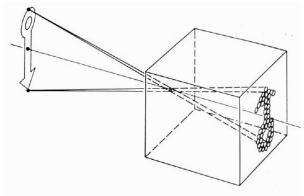
- A. Glassner. An Introduction to Ray Tracing. Academic Press, 1989. [In the lab.]
- K. Turkowski, "Properties of Surface Normal Transformations," Graphics Gems, 1990, pp. 539-547. [In the reader.]

The pinhole camera

The first camera - "camera obscura" - known to Aristotle.

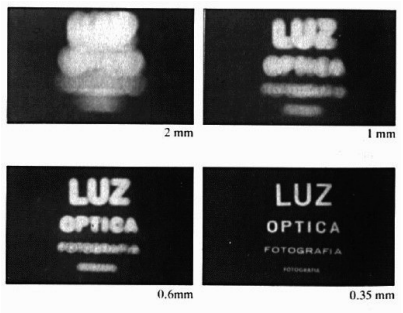


In 3D, we can visualize the blur induced by the pinhole (a.k.a., **aperture**):



Q: How would we reduce blur?

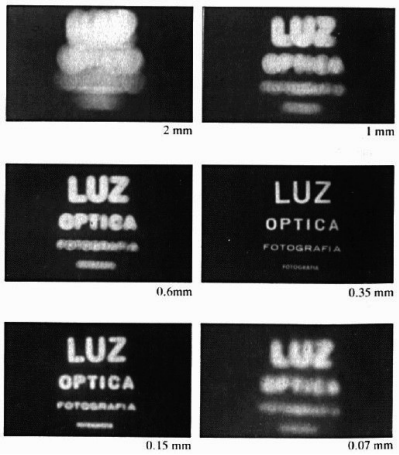
Shrinking the pinhole



Q: What happens as we continue to shrink the aperture?

Shrinking the pinhole, cont'd

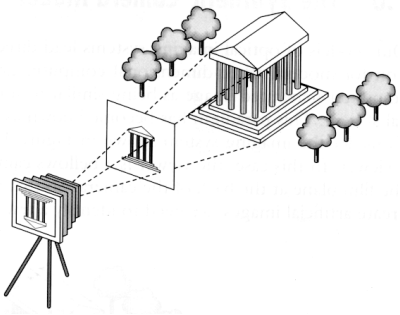
Diffraction



Imaging with the synthetic camera

In practice, pinhole cameras require long exposures, can suffer from diffraction effects, and give an inverted image.

In graphics, we have none of these are a problem.



The image is rendered onto an **image plane** (usually in front of the camera).

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.

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**.

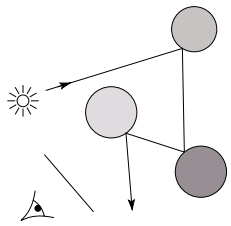
Here are the rules of geometric optics:

- 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).

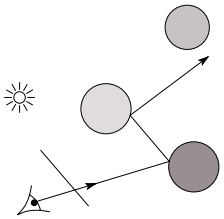
Eye vs. light ray tracing

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)

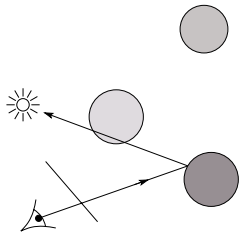


We will generally follow rays from the eye into the scene.

Precursors to ray tracing

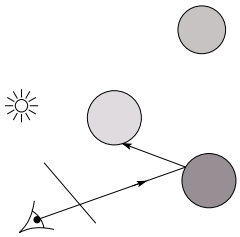
Local illumination

- Cast one eye ray, then shade according to light



Appel (1968)

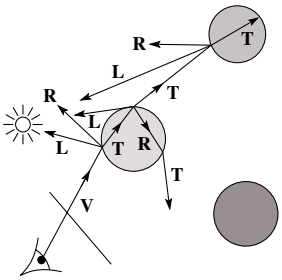
- Cast one eye ray + one ray to light



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:

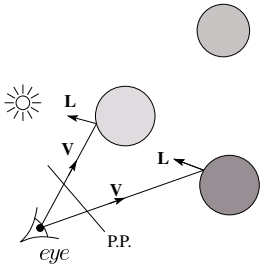
1.For each pixel, trace a **primary ray** in direction **V** to the first visible surface.

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

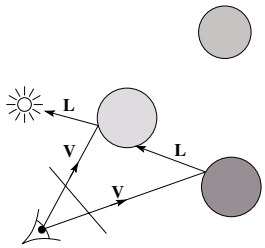
- **Shadow rays** in directions $\mathbf{L}_i$ to light sources
- **Reflected ray** in direction **R**.
- **Refracted ray** or **transmitted ray** in direction **T**.

Whitted algorithm (cont'd)

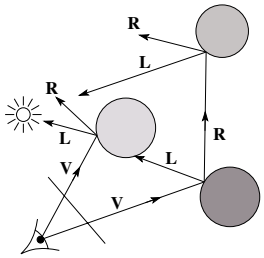
Let's look at this in stages:



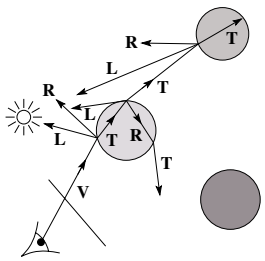
Primary rays



Shadow rays

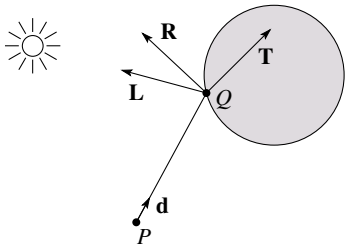


Reflection rays



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:

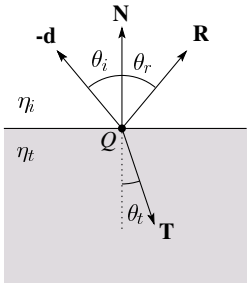
$$I(P, \mathbf{d}) = I_{\text{direct}} + I_{\text{reflected}} + I_{\text{transmitted}}$$

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:

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

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

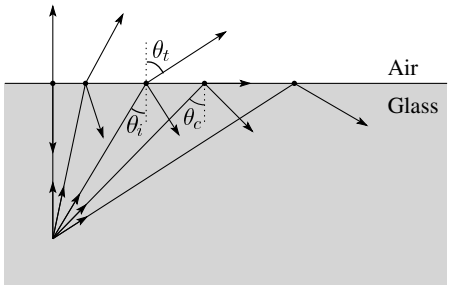
Total Internal Reflection

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.



Error in Watt!!

In order to compute the refracted direction, it is useful to compute the cosine of the angle of refraction in terms of the incident angle and the ratio of the indices of refraction.

On page 24 of Watt, he develops a formula for computing this cosine. Notationally, he uses μ instead of η for the index of refraction in the text, but uses η in Figure 1.16(!?), and the angle of incidence is ϕ and the angle of refraction is θ.

Unfortunately, he makes a grave error in computing $\cos\theta$.

The last equation on page 24 should read:

$$\cos\theta = \sqrt{1-\mu^2(1-\cos^2\phi)}$$

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 = **COP**
        d = (*S* - *P*)/|| *S* – *P*||
        I(i,j) = *traceRay*(scene, *P*, **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(                          )
    R = reflectDirection(               )
    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 (notTIR (                              )) then
        T = refractDirection (                )
        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 ℓ do:
        atten = ℓ -> distanceAttenuation(              ) *
                ℓ -> shadowAttenuation(              )
        I ← I + atten*(diffuse term + spec term)
    end for
    return I
end function
```


Shadow attenuation

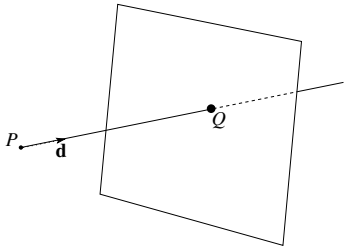
Computing a shadow can be as simple as checking to see if a ray makes it to the light source.

For a point light source:

```
function PointLight::shadowAttenuation(scene, P)
    d = (ℓ.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



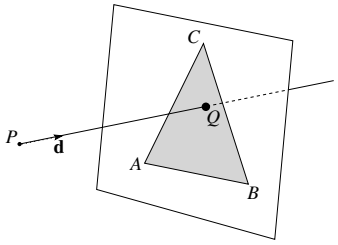
We can write the equation of a plane as:

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

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

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

Ray-triangle intersection



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

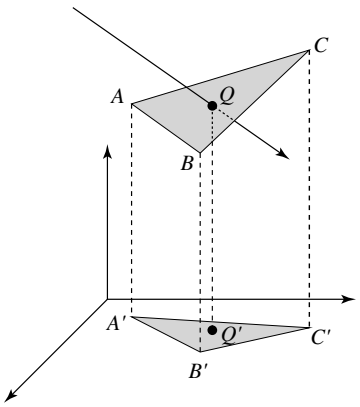
Then, we need to decide if the point is inside or outside of the triangle.

Solution 1: compute barycentric coordinates from 3D points.

What do you do with the barycentric coordinates?

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:

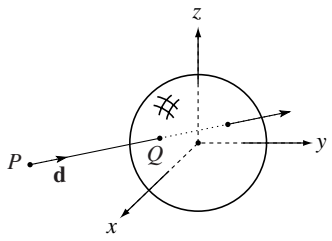
- material properties
- texture coordinates
- normals

For example:

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

Interpolating normals, known as Phong interpolation, gives triangle meshes a smooth shading appearance.

Intersecting rays with spheres



Given:

- The coordinates of a point along a ray passing through *P* in the direction **d** are:

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

- A unit sphere *S* centered at the origin defined by the equation:

Find: The *t* at which the ray intersects *S*.

Intersecting rays with spheres

Solution by substitution:

$$x^2 + y^2 + z^2 - 1 = 0$$
$$(P_x + td_x)^2 + (P_y + td_y)^2 + (P_z + td_z)^2 = 0$$
$$at^2 + bt + c = 0$$

where

$$a = d_x^2 + d_y^2 + d_z^2$$
$$b = 2(P_x d_x + P_y d_y + P_z d_z)$$
$$c = P_x^2 + P_y^2 + P_z^2 - 1$$

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?

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?

Summary

What to take home from this lecture:

1. The meanings of all the boldfaced terms.
2. Enough to implement basic recursive ray tracing.
3. How reflection and transmission directions are computed.
4. How ray--object intersection tests are performed.