11. Ray Tracing

Reading

Required:

- Watt, sections 1.3-1.4, 12.1-12.5.1.

Further reading:

- Watt, chapter 14 and the rest of chapters 10 and 12.
- A. Glassner. An Introduction to Ray Tracing. Academic Press, 1989. [In the lab.]
- T. Whitted. An improved illumination model for shaded display. Communications of the ACM 23(6), 343-349, 1980. [In the reader.]

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**.

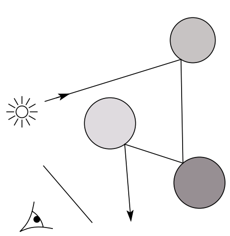
Here are the rules of geometric optics:

- 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).

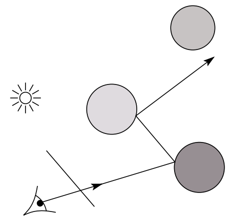
Eye vs. light ray tracing

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)

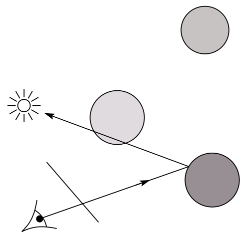


We will generally follow rays from the eye into the scene.

Precursors to ray tracing

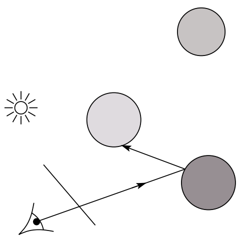
Local illumination

- Cast one eye ray, then shade according to light



Appel (1968)

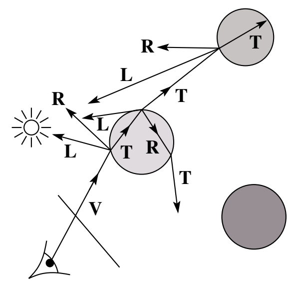
- Cast one eye ray + one ray to light



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:

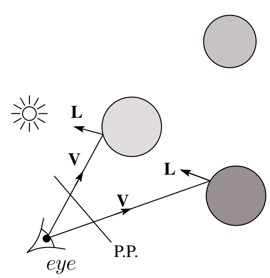
1.For each pixel, trace a **primary ray** in direction **V** to the first visible surface.

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

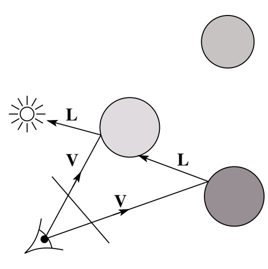
- **Shadow rays** in directions $\mathbf{L}_i$ to light sources
- **Reflected ray** in direction **R**.
- **Refracted ray** or **transmitted ray** in direction **T**.

Whitted algorithm (cont'd)

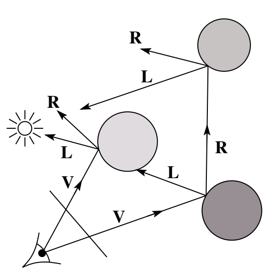
Let's look at this in stages:



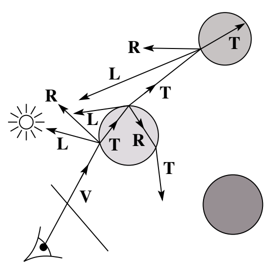
Primary rays



Shadow rays

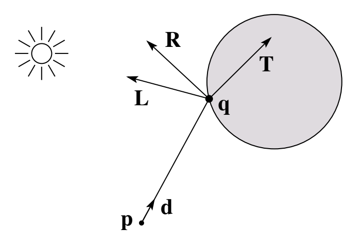


Reflection rays



Refracted rays

Shading



A ray is defined by an origin **p** and a unit direction **d** and is parameterized by t:

$$\mathbf{p} + t\mathbf{d}$$

Let $I(\mathbf{p}, \mathbf{d})$ be the intensity seen along that ray. Then:

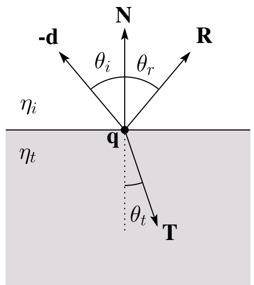
$$I(\mathbf{p}, \mathbf{d}) = I_{\text{direct}} + I_{\text{reflected}} + I_{\text{transmitted}}$$

where

- I_{direct} is computed from the Phong model
- $I_{\text{reflected}} = k_r I(\mathbf{q}, \mathbf{R})$
- $I_{\text{transmitted}} = k_t I(\mathbf{q}, \mathbf{T})$

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**.

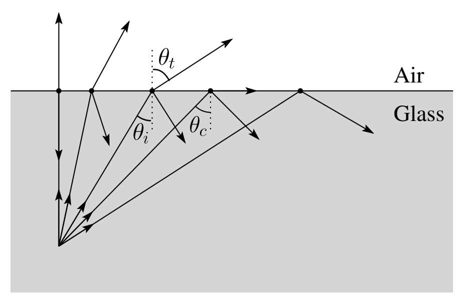
Total Internal Reflection

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, material) ← intersect (scene, p, d)
    q ← ray (p, d) evaluated at t
    I = shade(                                   )
    R = reflectDirection(              )
    I ← I + material.k_r * traceRay(scene, q, R)
    if ray is entering object then
        n_i = index_of_air
        n_t = material.index
    else
        n_i = material.index
        n_t = index_of_air
    if (notTIR (                              )) then
        T = refractDirection (                          )
        I ← I + material.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(scene, material, q, N, d):
    I ← material.k_e + material.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
```

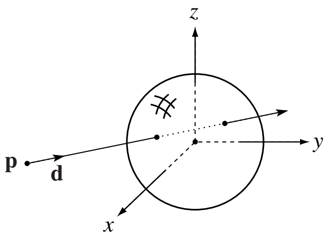

Shadow atttenuation

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

```
function shadowAttenuation(scene, p)
    d = (p - ℓ.position).normalize()
    (q, N, material) ← intersect(scene, p, d)
    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?

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

$$a = d_x^2 + d_y^2 + d_z^2$$
$$b = 2(p_x d_x + p_y d_y + p_z d_z)$$
$$c = d_x^2 + d_y^2 + d_z^2 - 1$$

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?

Intersecting with xformed geometry

What if the sphere were transformed by a matrix M (e.g., to make a rotated, translated, ellipsoid)?

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?

Epsilons

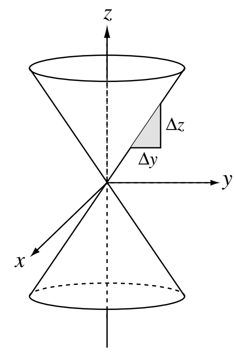
Due to finite precision arithmetic, we do not always get the exact intersection at a surface.

Q: What kinds of problems might this cause?

Q: How might we resolve this?

Intersecting rays with cones

A cone centered at the origin:



Has the form:

$$x^2 + y^2 = \beta^2 z^2$$

where:

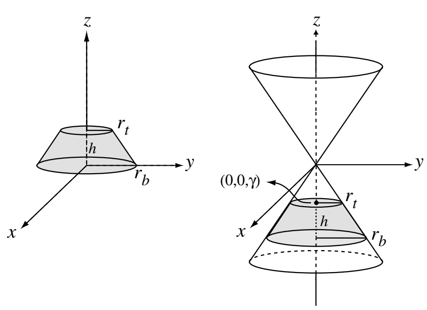
$$\beta = \left(\frac{\Delta y}{\Delta z}\right)_{x=0}$$

Intersection with such a cone can be computed the same way we did with spheres.

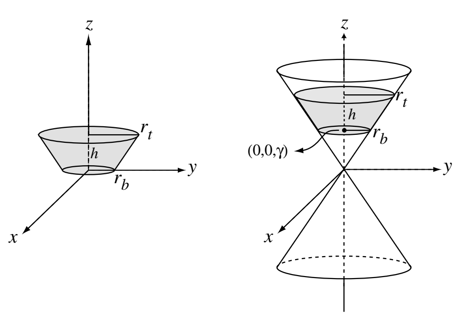
Intersecting rays with capped cones

For capped cones, we define top and bottom radii and a height. These are really two shifted types of cones.

If the top radius is smaller:



Else:



Intersecting rays with capped cones

In this case, we can solve by shifting the cone down by γ:

$$x^2 + y^2 = \beta^2 (z + \gamma)^2$$

Your assignment is to:

- Solve for β and γ (and store these statically)
- Intersect a ray with the cone and its caps

Q: What is the normal to the cone?

It turns out that if you can write a surface as:

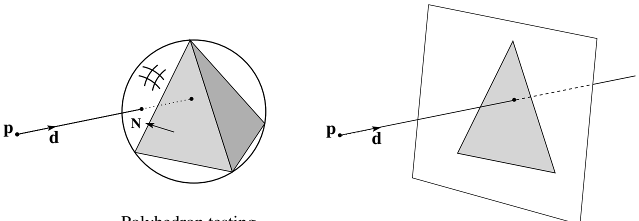
$$f(x, y, z) = 0$$

(a.k.a. an **implicit surface**), then:

$$\mathbf{N} \sim \nabla f(x, y, z)$$

So, the implicit equation and normal for a cone are:

Intersecting rays with polyhedra



Polyhedron testing   Polygon testing

To intersect a ray with a polyhedron:

- Test intersection of ray with bounding sphere.
- Locate the "front-facing" faces of the polyhedron with

$$\mathbf{d} \cdot \mathbf{N}$$

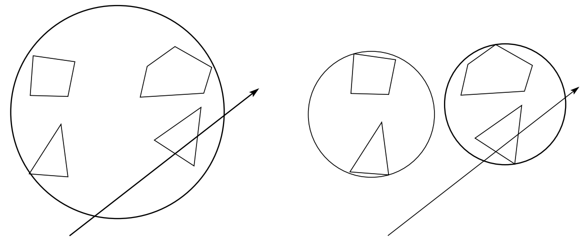
- Intersect the ray with each front face's supporting plane.
- Use a point-in-polygon test to see if the ray is inside the face.
- Sort intersections according to smallest *t*.

Acceleration: Hierarchical bounding volumes

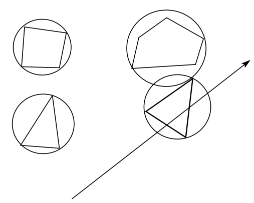
Vanilla ray tracing is really slow!

In practice, some acceleration technique is almost always used.

One approach is to use **hierarchical bounding volumes**.



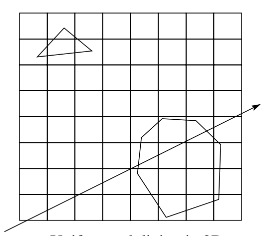
Intersect with largest B.V...   ...then intersect with children...



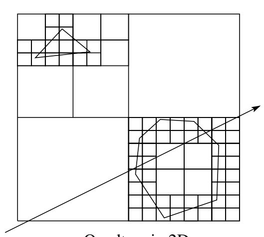
...until you reach the leaf nodes - the primitives.

Acceleration: Spatial subdivision

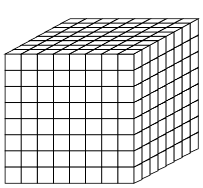
Another approach is **spatial subdivision**.



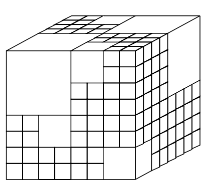
Uniform subdivion in 2D



Quadtree in 2D



Uniform subdivion in 3D



Octree in 3D

Idea:

- Partition objects spatially.
- Trace ray through voxel array.

Partition can be uniform or adaptive (e.g., octrees).

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.
5. Basic acceleration strategies.