11. Ray Tracing

Reading

Required:

- Watt, sections 1.3-1.4, 12.1-12.5.1, 10.6.

Further reading:

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

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**.

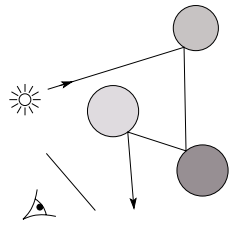
Here are the rules of geometric optics:

- 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).

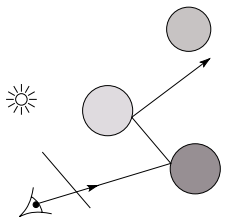
Eye vs. light ray tracing

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)

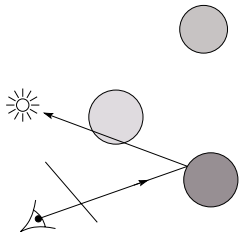


We will generally follow rays from the eye into the scene.

Precursors to ray tracing

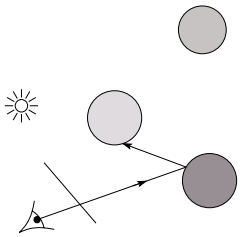
Local illumination

- Cast one eye ray, then shade according to light



Appel (1968)

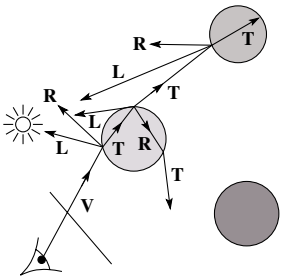
- Cast one eye ray + one ray to light



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:

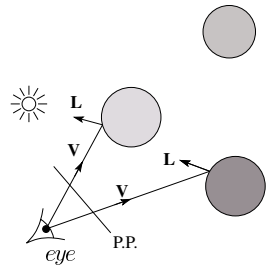
1.For each pixel, trace a **primary ray** in direction **V** to the first visible surface.

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

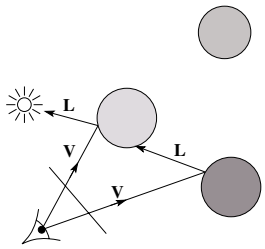
- **Shadow rays** in directions $\mathbf{L}_i$ to light sources
- **Reflected ray** in direction **R**.
- **Refracted ray** or **transmitted ray** in direction **T**.

Whitted algorithm (cont'd)

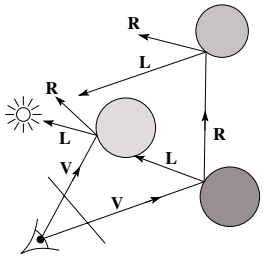
Let's look at this in stages:



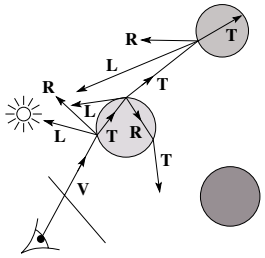
Primary rays



Shadow rays

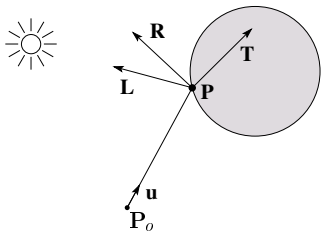


Reflection rays



Refracted rays

Shading



Let $I(P_o, \mathbf{u})$ be the intensity seen from point P_o along direction **u**:

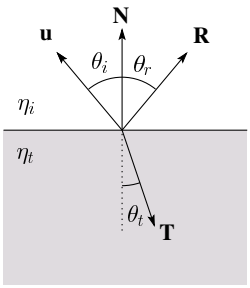
$$I(P_o, \mathbf{u}) = I_{\text{direct}} + I_{\text{reflected}} + I_{\text{transmitted}}$$

where

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

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**.

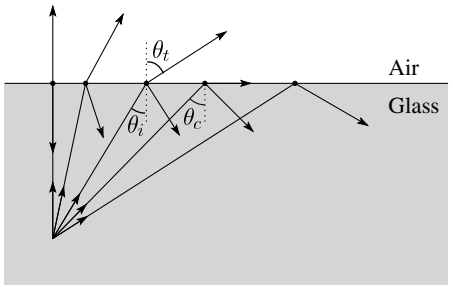
Total Internal Reflection

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

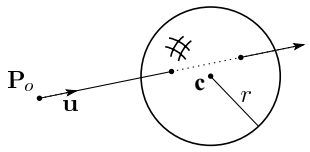
function *RayTrace*(P_o, **u**):
    (*P*, *Obj*) ← *RayCast* (P_o, **u**)
    $I \leftarrow k_e + k_a I_a$
    for each light source ℓ **do**:
        (*P'*, *Obj'*) ← *RayCast(*P, Dir(P, ℓ))
        if *Obj'* = ℓ **then**:
            $I \leftarrow I$ + (diffuse term) + (spec term)
        end if
    end for
    $I \leftarrow I + Obj.k_r * RayTrace\ (P, \mathbf{R})$
    $I \leftarrow I + Obj.k_t * RayTrace\ (P, \mathbf{T})$
    return *I*
end function

Terminating recursion

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

Possibilities:

Intersecting rays with spheres



Given:

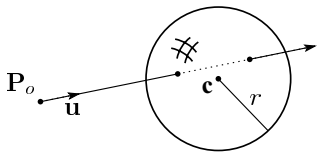
- A ray **P**(t) through initial position $\mathbf{P}_0$ in direction **u** (a unit vector):

$$\mathbf{P}(t) = \mathbf{P}_0 + t\,\mathbf{u}$$

- A sphere *S* centered at **c** with radius *r*

Find: The intersection of **P**(t) with *S*.

Intersecting rays with spheres

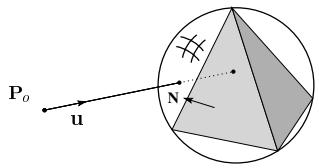


Solution:

$$\|\mathbf{P} - \mathbf{c}\| = r^2$$
$$\|\mathbf{P}_o + t\mathbf{u} - \mathbf{c}\| = r^2$$
$$[\mathbf{q} \equiv \mathbf{P}_o - \mathbf{c}]$$
$$(\mathbf{q} + t\mathbf{u})\bullet(\mathbf{q} + t\mathbf{u}) = r^2$$
$$\mathbf{q}\bullet\mathbf{q} + 2\mathbf{q}\bullet\mathbf{u}t + \mathbf{u}\bullet\mathbf{u}t^2 = r^2$$
$$at^2 + bt + c = 0$$

Q: What is the normal to the sphere?

Intersecting rays with polyhedra



To intersect a ray with a polyhedron:

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

$$\mathbf{u} \cdot \mathbf{N} < 0$$

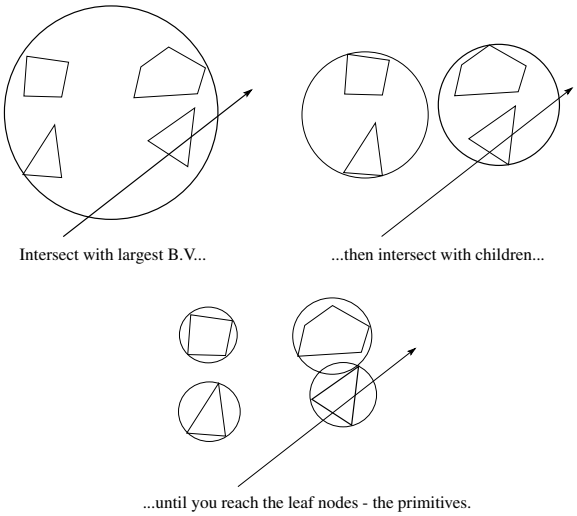
- Intersect the ray with each front face's supporting plane.
- Use a point-in-polygon test to see if the ray is inside the face.
- Sort intersections according to smallest *s*.

Acceleration: Hierarchical bounding volumes

Vanilla ray tracing is really slow!

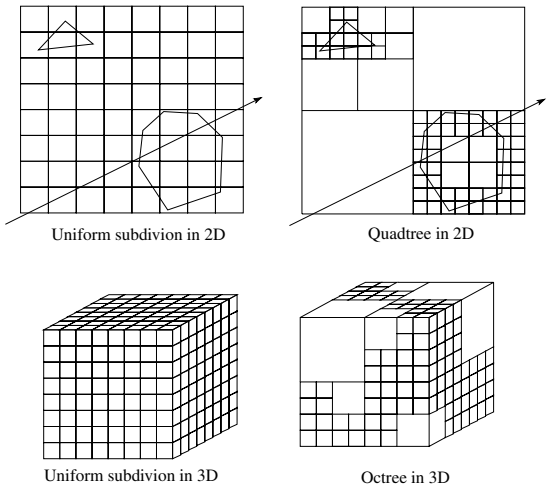
In practice, some acceleration technique is almost always used.

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



Acceleration: Spatial subdivision

Another approach is **spatial subdivision**.



Idea:

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

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

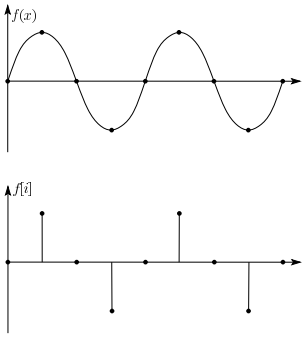
Aliasing

Ray tracing is a form of sampling and can suffer from annoying visual artifacts...

Consider a continuous function $f(x)$. Now sample it at intervals Δ to give $f[i] = f(i\Delta)$.

Q: How well does $f[i]$ approximate $f(x)$?

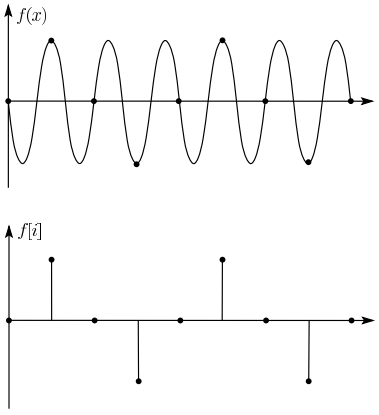
Consider sampling a sinusoid:



In this case, the sinusoid is reasonably well approximated by the samples.

Aliasing (con't)

Now consider sampling a higher frequency sinusoid

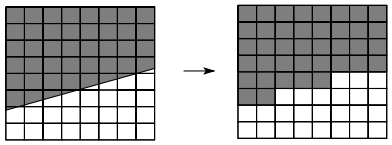


We get the exact same samples, so we seem to be approximating the first lower frequency sinusoid again.

We say that, after sampling, the higher frequency sinusoid has taken on a new "alias", i.e., changed its identity to be a lower frequency sinusoid.

Practical examples of aliasing

Drawing a polygon into the frame buffer:



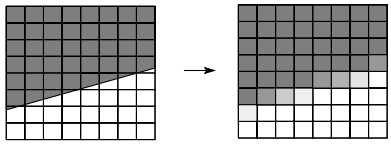
Temporal aliasing:

Anti-aliasing

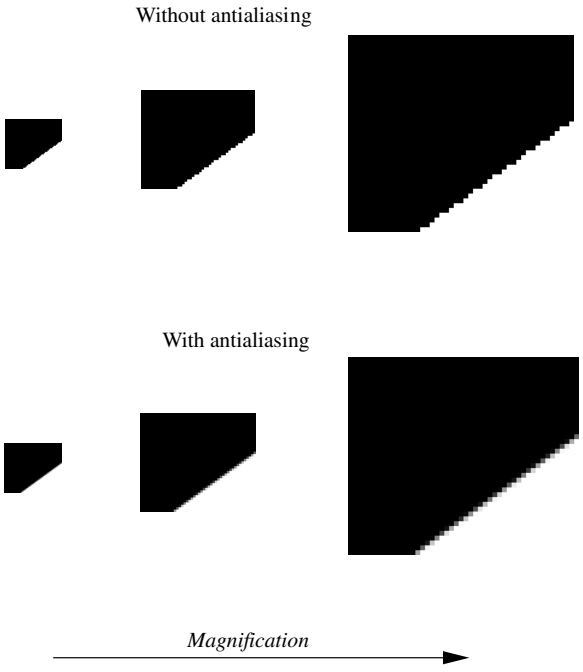
Q: How do we avoid aliasing artifacts?

1. Sampling:
2. Filtering:
3. Combination:

Example - polygon revisited:



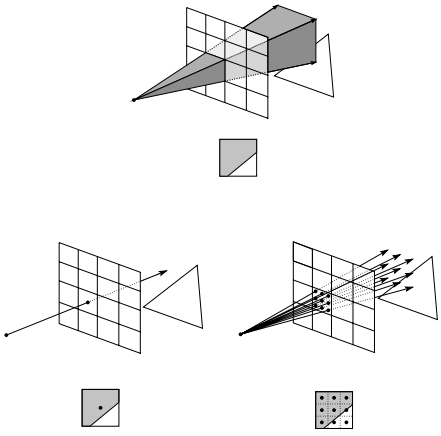
Polygon anti-aliasing



Q:What about temporal aliasing?

Antialiasing in a ray tracer

We would like to compute the average intensity in the neighborhood of each pixel.



When casting one ray per pixel, we are likely to have aliasing artifacts.

To improve matters, we can cast more than one ray per pixel and average the result.

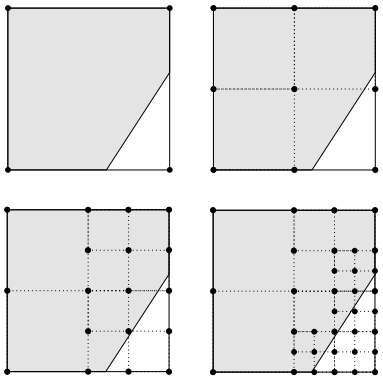
A.k.a., **super-sampling and averaging down**.

Antialiasing by adaptive sampling

Casting many rays per pixel can be unnecessarily costly.

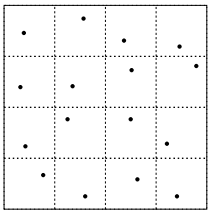
For example, if there are no rapid changes in intensity at the pixel, maybe only a few samples are needed.

Solution: **adaptive sampling**.



Q: When do we decide to cast more rays in a particular area?

Distribution ray tracing

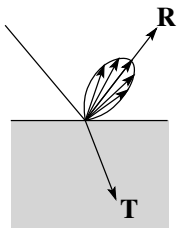


Idea:

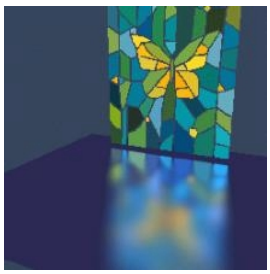
- Use non-uniform (jittered) samples.
- Replaces aliasing artifacts with noise.
- Provides additional effects if we distribute rays in other dimensions:
  - Reflection and refractions
  - Light source area
  - Camera lens area
  - Time

Originally called "distributed ray tracing," but we will call it **distribution ray tracing** so as not to confuse with parallel computing.

Distribution ray tracing (cont'd)



Distributing rays over reflection and/or refraction directions gives:

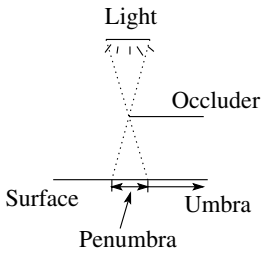


Distribution ray tracing (cont'd)

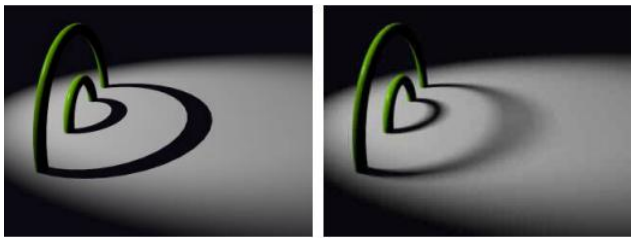
Operationally:

1. Partition the reflection directions into 16 angular regions. Assign each region a unique number between 1 and 16.
2. Partition each pixel into 16 regions. Assign each region a unique number between 1 and 16.
3. Select sub-pixel $m = 1$.
4. Cast a ray through sub-pixel m, jittered within its region.
5. After finding the first intersection, reflect into direction region m, jittered within that region. Repeat for future reflections.
6. Add result to current pixel total.
7. Increment m and, if $m \leqslant 16$, go to 4.
8. Divide by 16, store the result, choose the next pixel and go to 3.

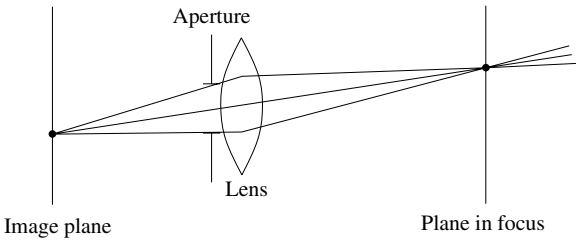
Distribution ray tracing (cont'd)



Distributing rays over light source area gives:

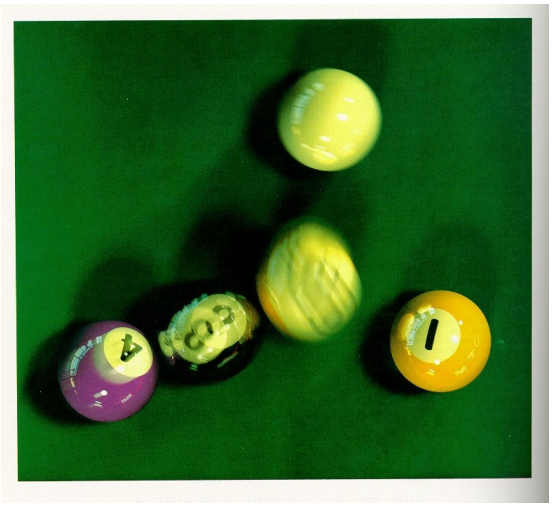


Distribution ray tracing (cont'd)



Distributing rays over a finite aperture gives:

Distribution ray tracing (cont'd)



Distributing rays over time gives:

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.
6. An intuition for what aliasing is.
7. How to reduce aliasing artifacts in a ray tracer
8. Concept of distribution ray tracing.