Distribution Ray Tracing

Reading

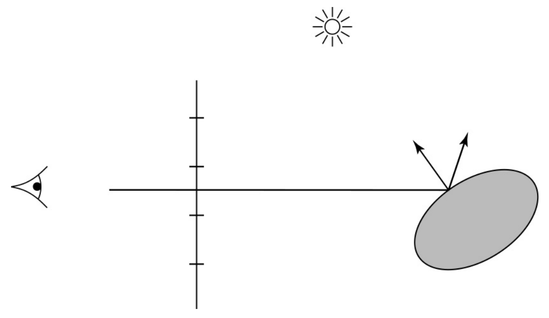
Required:

- Watt, sections 10.6 ,14.8. (see handouts)

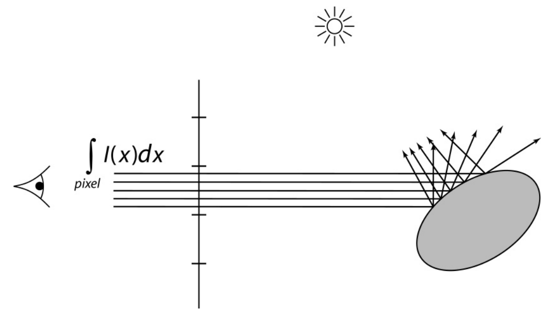
Further reading:

- Watt, sections 10.4-10.5
- A. Glassner. An Introduction to Ray Tracing. Academic Press, 1989. [In the lab.]
- Robert L. Cook, Thomas Porter, Loren Carpenter. "Distributed Ray Tracing." Computer Graphics (Proceedings of SIGGRAPH 84). *18 (3)*. pp. 137-145. 1984.
- James T. Kajiya. "The Rendering Equation." Computer Graphics (Proceedings of SIGGRAPH 86). *20 (4)*. pp. 143-150. 1986.

Pixel anti-aliasing



No anti-aliasing



Pixel anti-aliasing

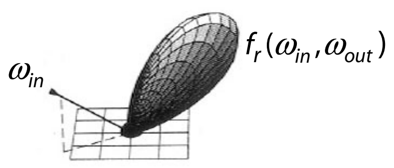
BRDF, revisited

The reflection model on the previous slide assumes that inter-reflection behaves in a mirror-like fashion.

Recall that we could view light reflection in terms of the general **Bi-directional Reflectance Distribution Function** (**BRDF**):

$$f_r(\omega_{in}, \omega_{out})$$

Which we could visualize for a given ω_{in}:



Surface reflection equation

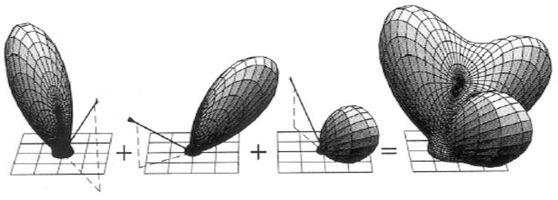
To compute the reflection from a real surface, we would actually need to solve the **surface reflection equation**:

$$I(\omega_{out}) = \int_H I(\omega_{in}) f_r(\omega_{in}, \omega_{out}) d\omega_{in}$$

For a directional light with intensity L_1 coming from direction direction, ω_1, we can view the remaining directions as contributing zero, giving:

$$I(\omega_{out}) = L_1 f_r(\omega_1, \omega_{out})$$

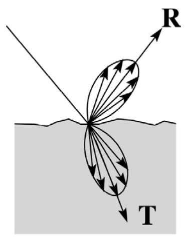
We can plot the reflected light as a function of viewing angle for multiple light source contributions:



Simulating gloss and translucency

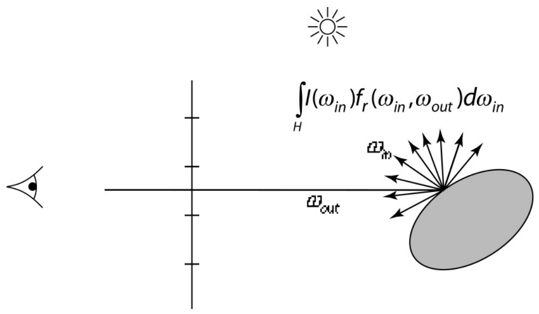
The mirror-like form of reflection, when used to approximate glossy surfaces, introduces a kind of aliasing, because we are undersampling reflection (and refraction).

For example:



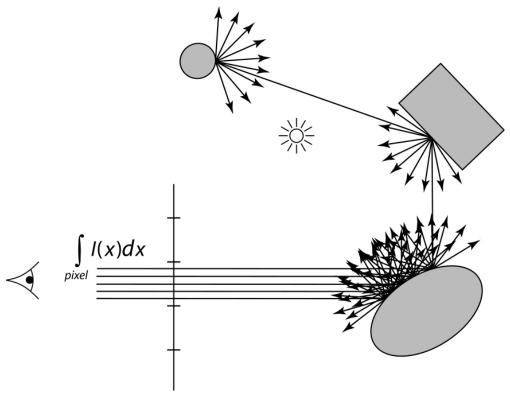
Distributing rays over reflection directions gives:

Reflection anti-aliasing


$\int_H I(\omega_{in}) f_r(\omega_{in}, \omega_{out}) d\omega_{in}$
ω_{in}
ω_{out}

Reflection anti-aliasing

Full anti-aliasing



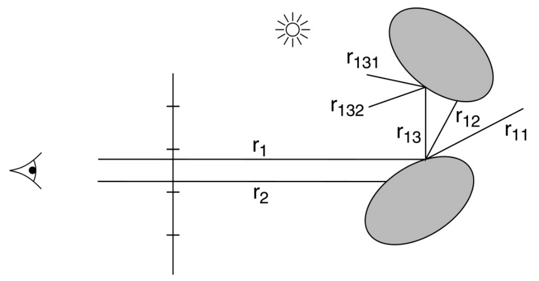
Full anti-aliasing…lots of nested integrals!

Computing these integrals is prohibitively expensive, especially after following the rays recursively.

We'll look at ways to approximate high-dimensional integrals…

Summing over ray paths

We can think of this problem in terms of enumerated rays:



The intensity at a pixel is the sum over the primary rays:

$$I_{pixel} = \frac{1}{n}\sum_i^n I(r_i)$$

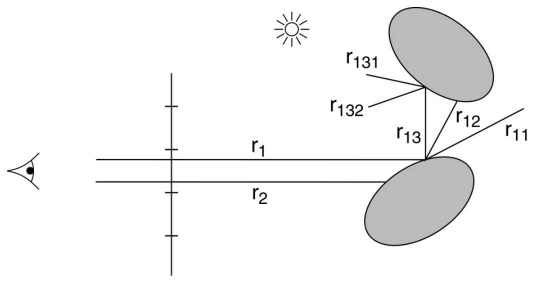
For a given primary ray, its intensity depends on secondary rays:

$$I(r_i) = \sum_j I(r_{ij}) f_r(r_{ij} \rightarrow r_i)$$

Substituting back in:

$$I_{pixel} = \frac{1}{n}\sum_i \sum_j I(r_{ij}) f_r(r_{ij} \rightarrow r_i)$$

Summing over ray paths



We can incorporate tertiary rays next:

$$I_{pixel} = \frac{1}{n}\sum_i \sum_j \sum_k I(r_{ijk}) f_r(r_{ijk} \rightarrow r_{ij}) f_r(r_{ij} \rightarrow r_i)$$

Each triple i,j,k corresponds to a ray path:

$$r_{ijk} \rightarrow r_{ij} \rightarrow r_i$$

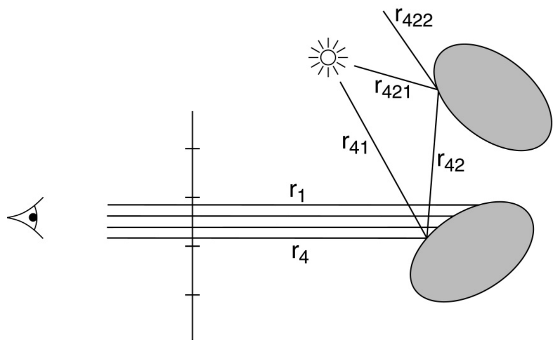
So, we can see that ray tracing is a way to approximate a complex, nested light transport integral with a summation over ray paths (of arbitrary length!).

Problem: too expensive to sum over all paths.

Solution: choose a small number of "good" paths.

Whitted integration

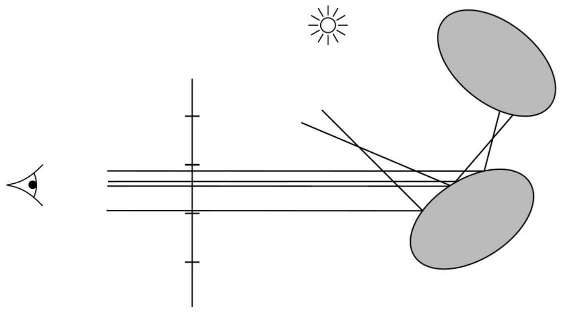
An anti-aliased Whitted ray tracer chooses very specific paths, i.e., paths starting on a regular sub-pixel grid with only perfect reflections (and refractions) that terminate at the light source.



One problem with this approach is that it doesn't account for non-mirror reflection at surfaces.

Monte Carlo path tracing

Instead, we could choose paths starting from random sub-pixel locations with completely random decisions about reflection (and refraction). This approach is called **Monte Carlo path tracing** [Kajiya86].



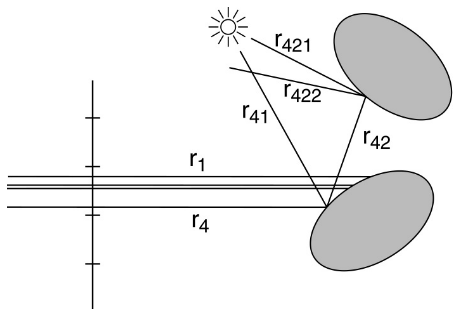
The advantage of this approach is that the answer is known to be unbiased and will converge to the right answer.

Importance sampling

The disadvantage of the completely random generation of rays is the fact that it samples unimportant paths and neglects important ones.

This means that you need a lot of rays to converge to a good answer.

The solution is to re-inject Whitted-like ideas: spawn rays to the light, and spawn rays that **favor** the specular direction.

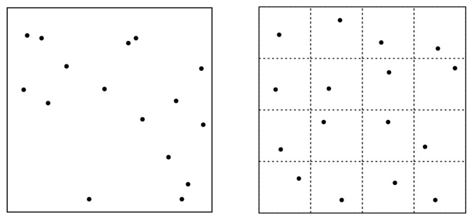


Stratified sampling

One problem is that samples may still clump together and give uneven results, i.e., require more samples to get a good answer. How can we make sure they spread out well?

Answer: **stratified sampling**.

E.g., for sub-pixel samples (here 16 rays/pixel):



Random Stratified

The stratified pattern on the right is also sometimes called a **jittered** sampling pattern.

One interesting side effect of these stochastic sampling patterns is that they actually injects noise into the solution (slightly grainier images). This noise tends to be less objectionable than aliasing artifacts.

Distribution ray tracing

These ideas can be combined to give a particular method called **distribution ray tracing** [Cook84]:

- uses non-uniform (jittered) samples.
- replaces aliasing artifacts with noise.
- provides additional effects by distributing rays to sample:
  - Reflections 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.]

DRT pseudocode

TraceImage() looks basically the same, except now each pixel records the average color of jittered sub-pixel rays.

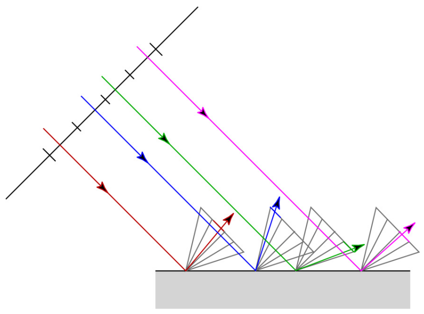
```
function traceImage (scene):
    for each pixel (i, j) in image do
        I(i, j) ← 0
        for each sub-pixel id in (i,j) do
            s ← pixelToWorld(jitter(i, j, id))
            p ← COP
            d ←(s - p).normalize()
            I(i, j) ← I(i, j) + traceRay(scene, p, d, id)
        end for
        I(i, j) ← I(i, j)/numSubPixels
    end for
end function
```

A typical choice is numSubPixels = 5*5.

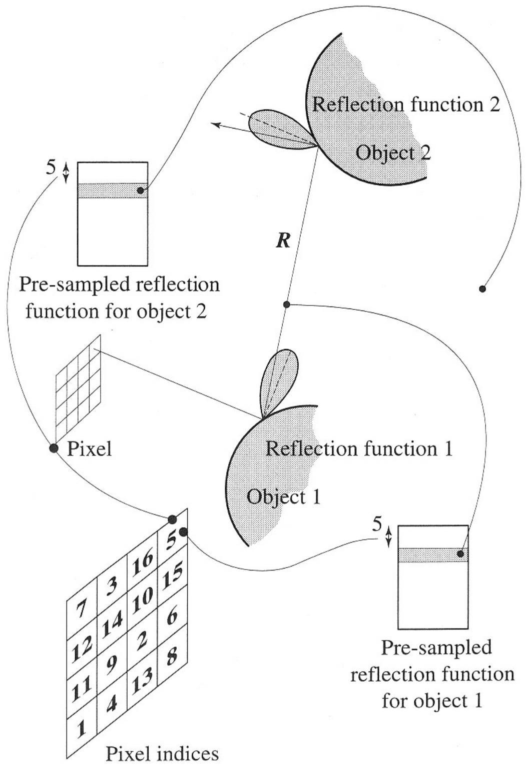
DRT pseudocode (cont'd)

Now consider *traceRay*(), modified to handle (only) opaque glossy surfaces:

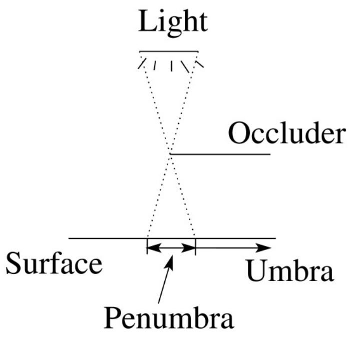
```
function traceRay(scene, p, d, id):
    (q, N, material) ← intersect (scene, p, d)
    I ← shade(…)
    R ← jitteredReflectDirection(material, N, -d, id)
    I ← I + material.$k_r$ * traceRay(scene, q, R, id)
    return I
end function
```



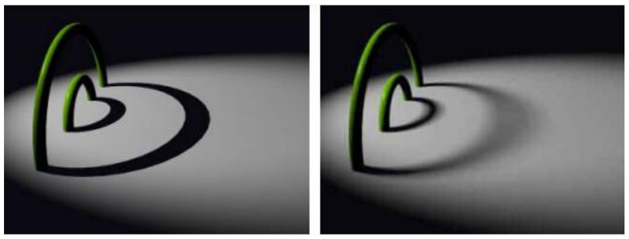
Pre-sampling glossy reflections



Soft shadows

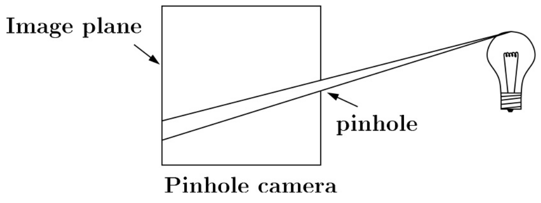


Distributing rays over light source area gives:

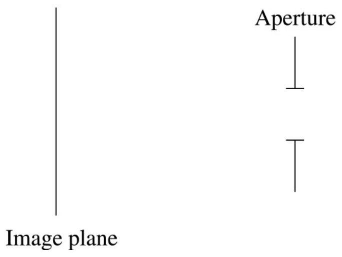


The pinhole camera, revisited

Recall the pinhole camera:



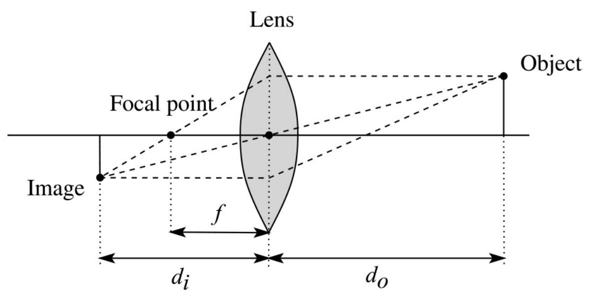
Q: How can we simulate a pinhole camera more accurately?



Lenses

Pinhole cameras in the real world require small apertures to keep the image in focus.

Lenses focus a bundle of rays to one point => can have larger aperture.



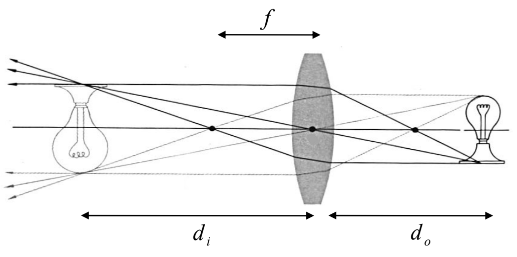
For a "thin" lens, we can approximately calculate where an object point will be in focus using the the Gaussian lens formula:

$$\frac{1}{d_i} + \frac{1}{d_o} = \frac{1}{f}$$

where *f* is the **focal length** of the lens.

Lenses (cont'd)

An image is formed of the whole object by collecting bundles of rays from every point on the object:

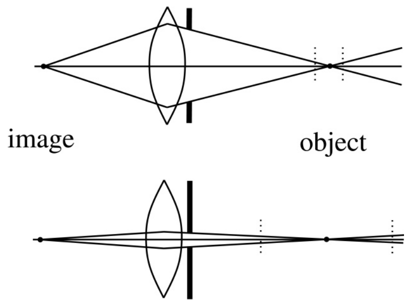


Depth of field

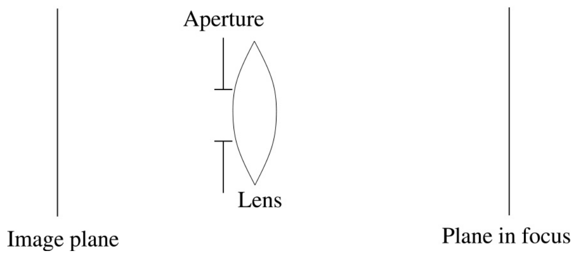
Lenses do have some limitations.

The most noticeable is the fact that points that are not in the object plane will appear out of focus.

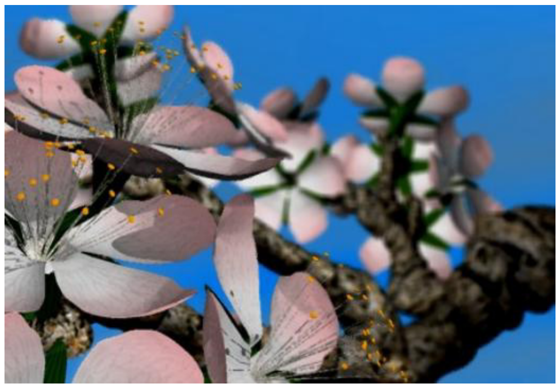
The **depth of field** is a measure of how far from the object plane points can be before appearing "too blurry."



Simulating depth of field

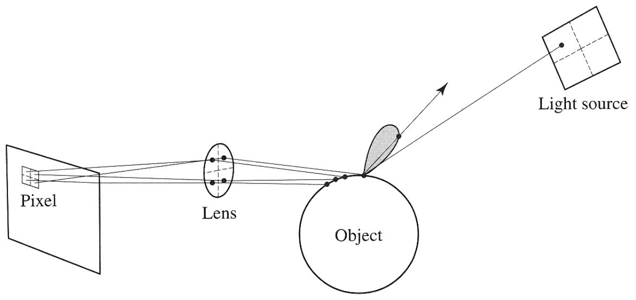


Distributing rays over a finite aperture gives:



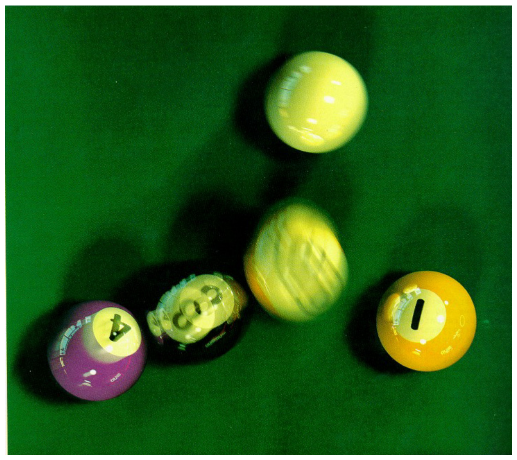
Chaining the ray id's

In general, you can trace rays through a scene and keep track of their id's to handle *all* of these effects:



DRT to simulate ____________________

Distributing rays over time gives:



Summary

What to take home from this lecture:

1. The limitations of Whitted ray tracing.
2. How distribution ray tracing works and what effects it can simulate.