9. Distribution Ray Tracing

Reading

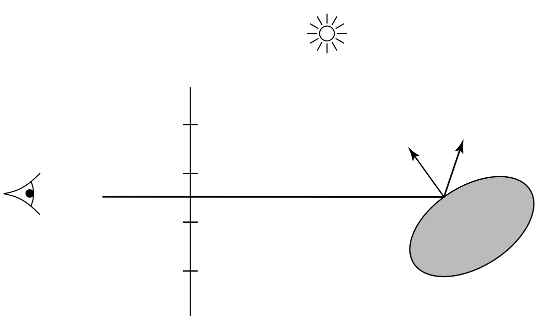
Required:

- Watt, sections 10.6 ,14.8.

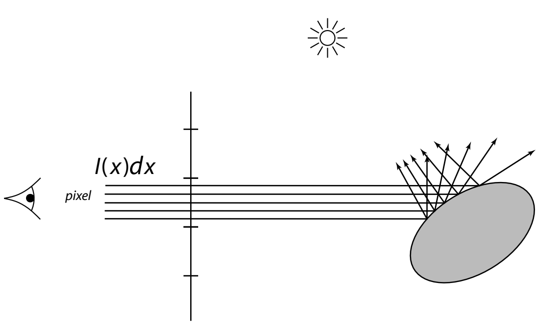
Further reading:

- A. Glassner. An Introduction to Ray Tracing. Academic Press, 1989. [In the lab.]

Pixel anti-aliasing



No anti-aliasing

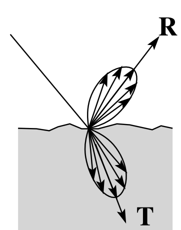


Pixel anti-aliasing

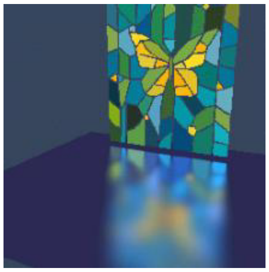
Simulating gloss and translucency

The resulting rendering can still have a form of aliasing, because we are undersampling reflection (and refraction).

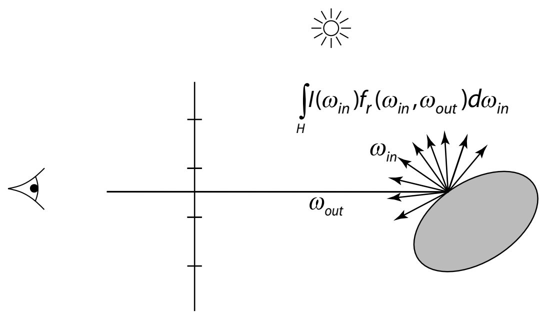
For example:



Distributing rays over reflection directions gives:

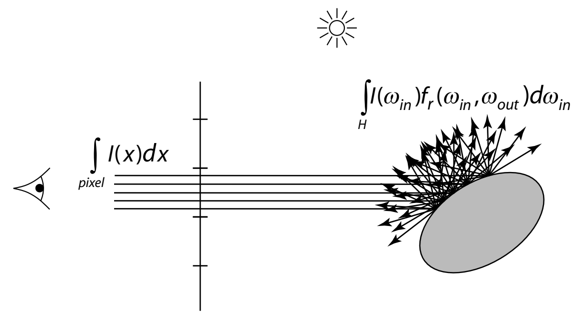


Reflection anti-aliasing



Reflection anti-aliasing

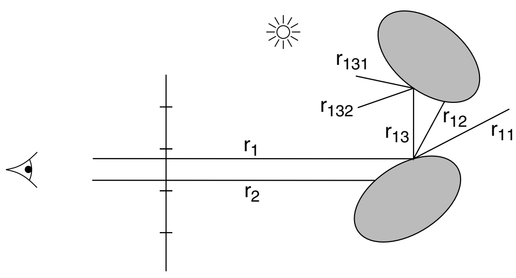
Full anti-aliasing



Full anti-aliasing

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:

$$I_{pixel} = \sum_i I(r_i)$$

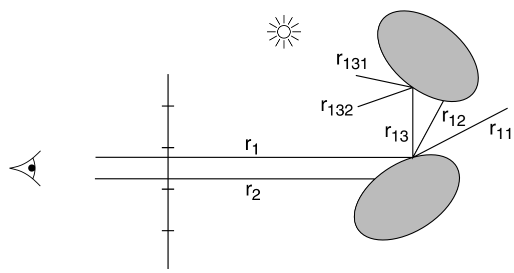
For a given primary ray, its intensity depends on secondary rays:

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

Substituting back in:

$$I_{pixel} = \sum_i \sum_j I(r_{ij}) f_r(r_{ij} \to r_i)$$

Summing over ray paths



We can incorporate tertiary rays next:

$$I_{pixel} = \sum_i \sum_j \sum_k I(r_{ijk}) f_r(r_{ijk} \to r_{ij}) f_r(r_{ij} \to r_i)$$

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

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

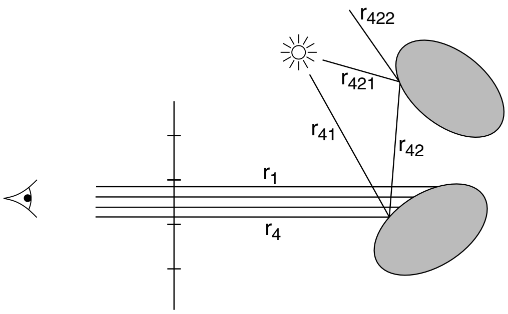
So, we can see that ray tracing is a way to approximate a complex, nested light transport integral with a summation over ray paths (of arbitrary length!).

Problem: too expensive to sum over all paths.

Solution: choose a small number of “good” paths.

Whitted integration

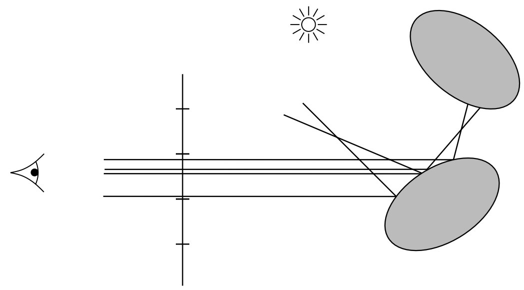
An anti-aliased Whitted ray tracer chooses very specific paths, i.e., paths starting on a regular sub-pixel grid with only perfect reflections (and refractions) that terminate at the light source.



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**.



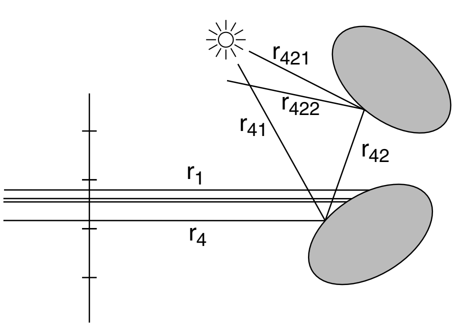
The advantage of this approach is that the answer is known to be unbiased and will converge to the right answer.

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.

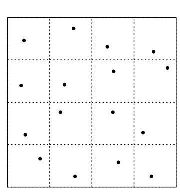
The solution is to re-inject Whitted-like ideas: spawn rays to the light, and spawn rays that **favor** the specular direction.



Stratified sampling

Another method that gives faster convergence is **stratified sampling**.

Notice, for example, that rays cast through a pixel can clump together. Here's an improved sampling pattern:



We call this a **jittered** sampling pattern.

One interesting side effect is that this randomness actually injects noise in the solution (slightly grainier images). This noise is actually more visually appealing than aliasing artifacts.

Distribution ray tracing

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

- 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.

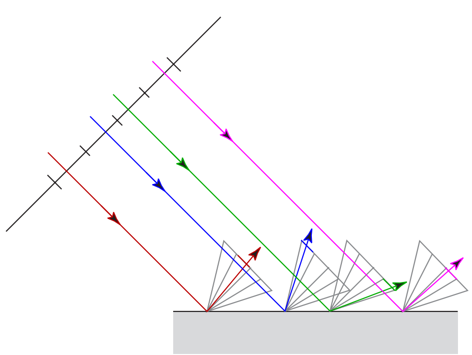
function *traceImage* (scene):
    for each pixel (i, j) in image **do**
        I(i, j) ← 0
        for each sub-pixel id in (i,j) **do**
            s ← *pixelToWorld*(jitter(i, j, id))
            p ← **COP**
            d ←(**s** - **p**).normalize()
            I(i, j) ← I(i, j) + *traceRay*(scene,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

A typical choice is numSubPixels = 4*4.

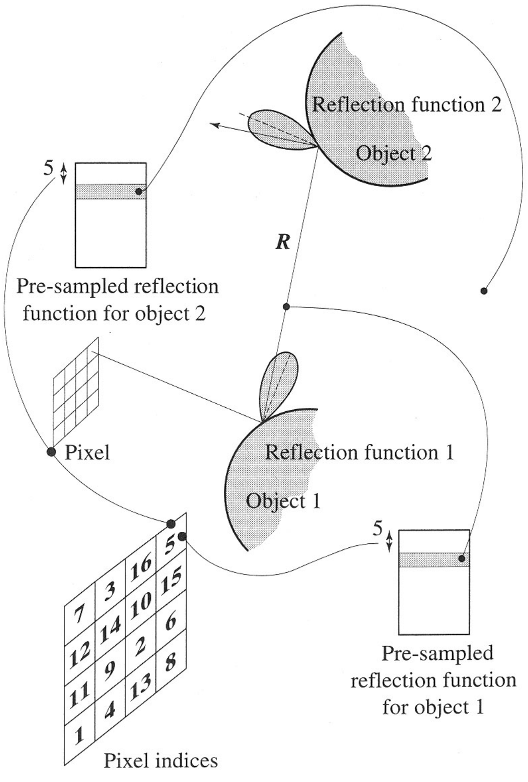
DRT pseudocode (cont'd)

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

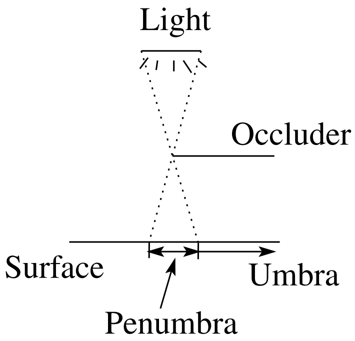
function *traceRay*(scene, **p**, **d,** id):
    (**q**, **N**, material) ← *intersect* (scene, **p**, **d**)
    I ← *shade*(…)
    R ← *jitteredReflectDirection*(**N**, -**d**, id)
    I ← I + material.k_r ∗ *traceRay*(scene, **q**, **R,** id)
    return I
end function



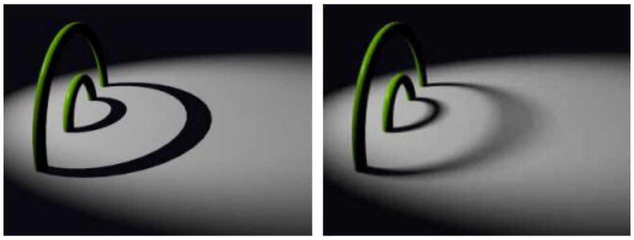
Pre-sampling glossy reflections



Soft shadows



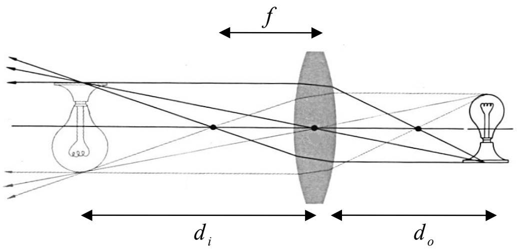
Distributing rays over light source area gives:



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:

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

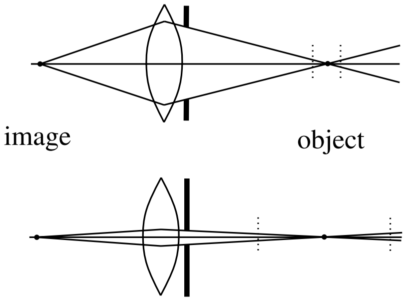
where *f* is the **focal length** of the lens.

Depth of field

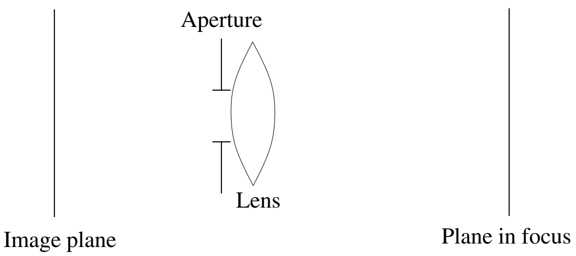
Lenses do have some limitations.

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.”



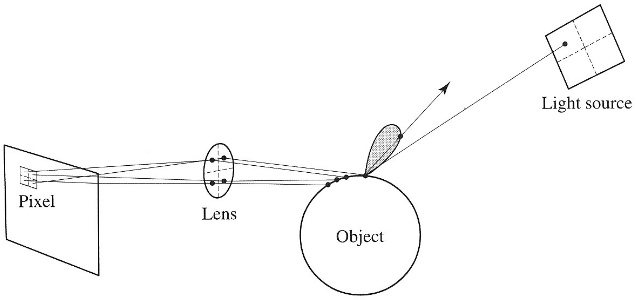
Simulating depth of field



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: