Distribution Ray Tracing

Brian Curless
CSE 457
Spring 2010

Reading

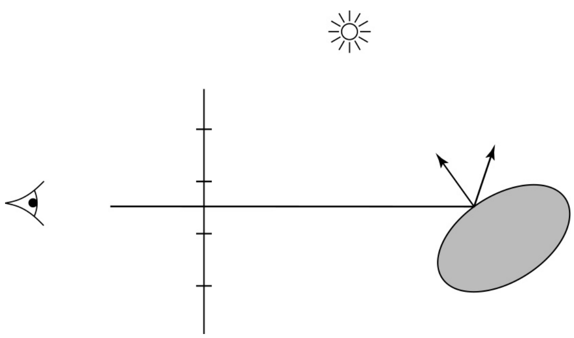
Required:

- Shirley, section 10.11

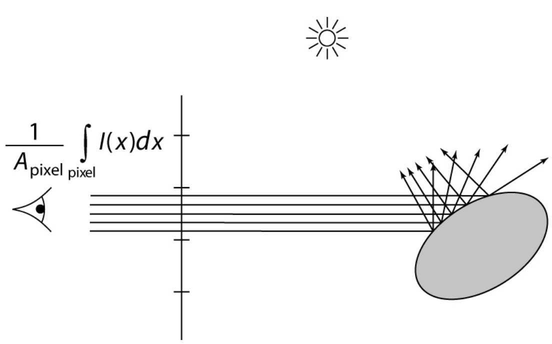
Further reading:

- 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.

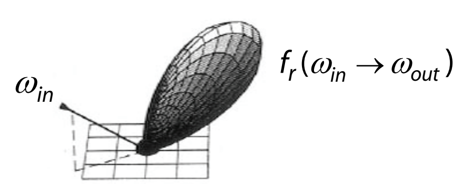
Recall that we could view light reflection in terms of the general **Bi-directional Reflectance Distribution Function** (**BRDF**):

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

Sometimes this is written as:

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

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



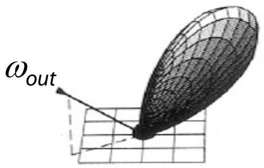
This is like a ray of light coming in at direction ω_{in} and scattering into directions ω_{out}.

Surface reflection equation

BRDF's exhibit reciprocity:

$$f_r(\omega_{in} \to \omega_{out}) = f_r(\omega_{out} \to \omega_{in})$$

This, combined with the idea of tracing rays from the viewer into the scene, means that we can turn things around:



Now, we can think of the BRDF as weighting light coming in from all directions ω_{in} and summing their effect into ω_{out}.

This idea gives rise to the surface reflection equation:

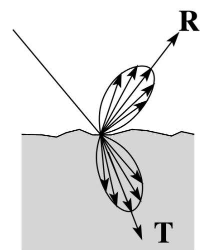
$$I(\omega_{out}) = \int_H I(\omega_{in}) f_r(\omega_{in} \to \omega_{out}) (\omega_{in} \bullet \mathbf{N}) d\omega_{in}$$

Where we are integrating over all incoming directions from the hemisphere H above the surface point.

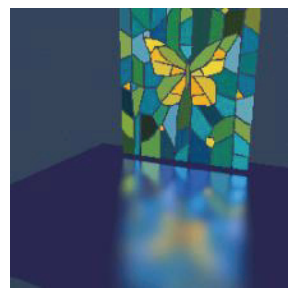
Simulating gloss and translucency

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

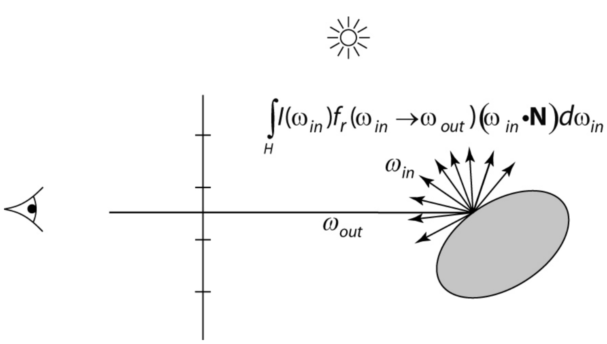
For example:



Distributing rays over reflection directions gives:

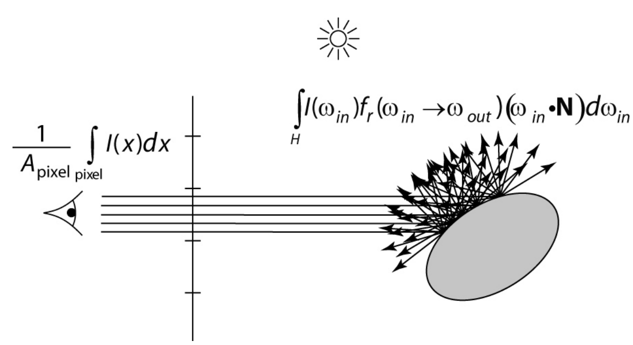


Reflection anti-aliasing



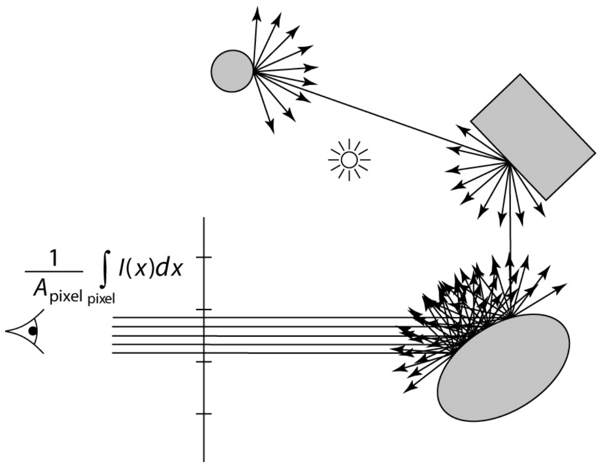
Reflection anti-aliasing

Pixel and reflection anti-aliasing



Pixel and reflection anti-aliasing

Full anti-aliasing



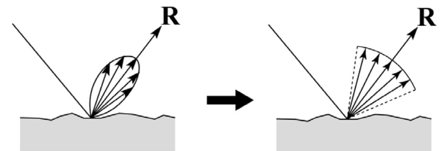
Full anti-aliasing…lots of nested integrals!

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

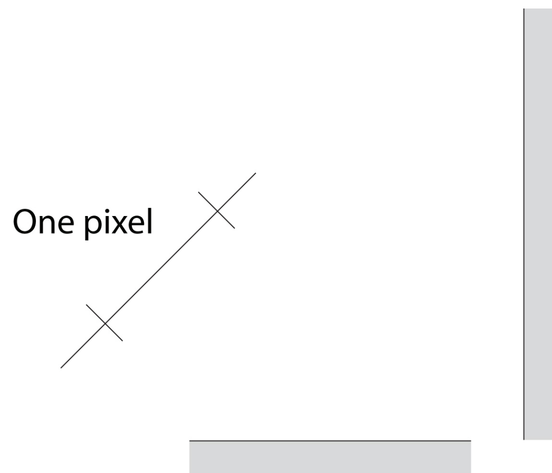
We'll look at ways to approximate high-dimensional integrals…

Glossy reflection revisited

Let's return to the glossy reflection model, and modify it – for purposes of illustration – as follows:

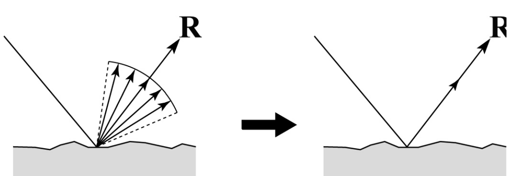


We can visualize the span of rays we want to integrate over, within a pixel:

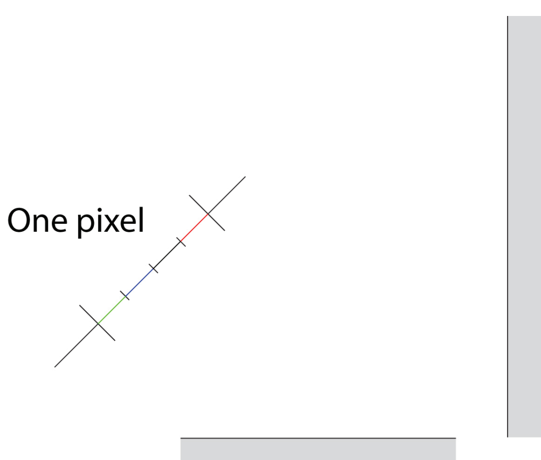


Whitted ray tracing

Returning to the reflection example, Whitted ray tracing replaces the glossy reflection with mirror reflection:

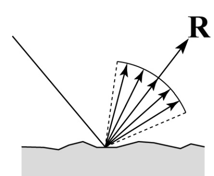


Thus, we render with anti-aliasing as follows:

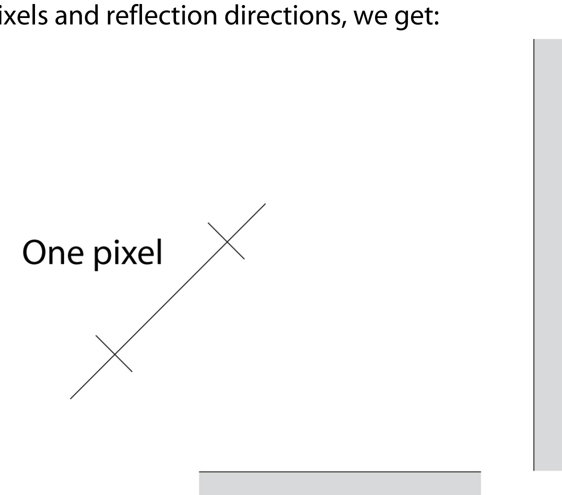


Monte Carlo path tracing

Let' return to our original (simplified) glossy reflection model:

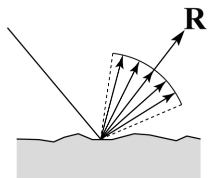


An alternative way to follow rays is by making random decisions along the way – a.k.a., Monte Carlo path tracing. If we distribute rays uniformly over pixels and reflection directions, we get:

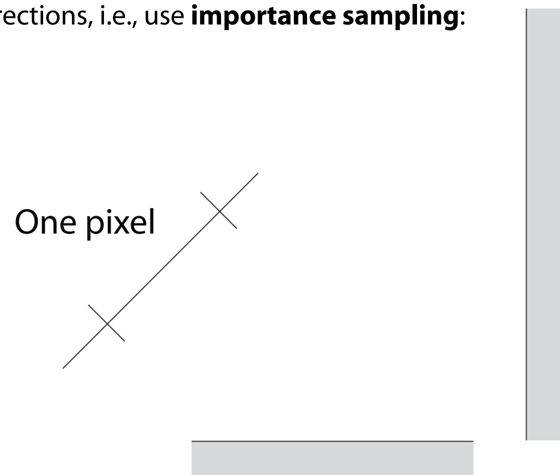


Importance sampling

The problem is that lots of samples are "wasted." Using again our glossy reflection model:

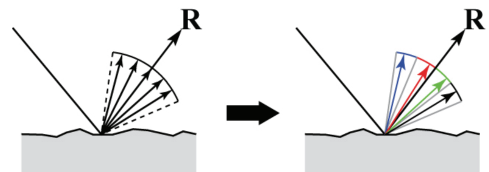


Let's now randomly choose rays, but according to a probability that favors more important reflection directions, i.e., use **importance sampling**:

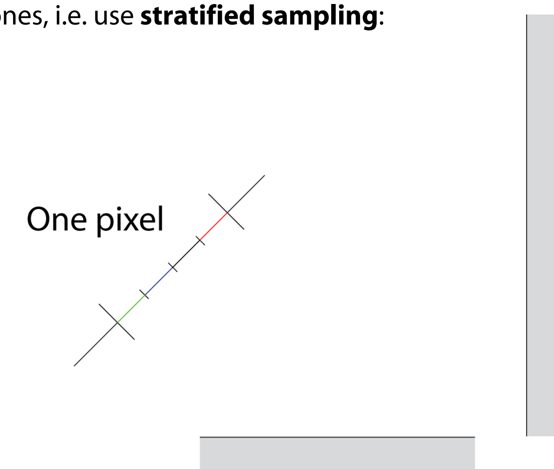


Stratified sampling

We still have a problem that rays may be clumped together. We can improve on this by splitting reflection into zones:

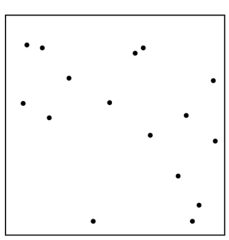


Now let's restrict our randomness to within these zones, i.e. use **stratified sampling**:

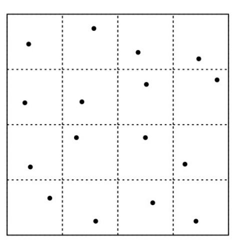


Stratified sampling of a 2D pixel

Here we see pure uniform vs. stratified sampling over a 2D pixel (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

[This approach was originally called "distributed ray tracing," but we will call it distribution ray tracing (as in probability distributions) so as not to confuse it with a parallel computing approach.]

DRT pseudocode

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

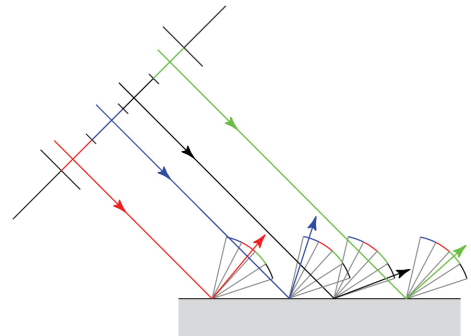
function *traceImage* (scene):
    for each pixel (i, j) in image **do**
        I(i, j) ← 0
        for each sub-pixel id in (i,j) **do**
            s ← *pixelToWorld*(jitter(i, j, id))
            p ← **COP**
            d ←(**s** - **p**).normalize()
            I(i, j) ← I(i, j) + *traceRay*(scene,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

A typical choice is numSubPixels = 5*5.

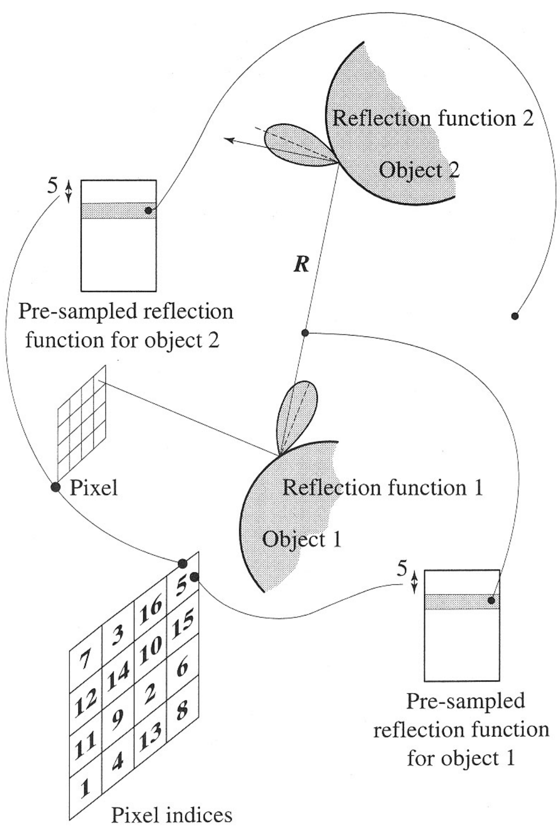
DRT pseudocode (cont'd)

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

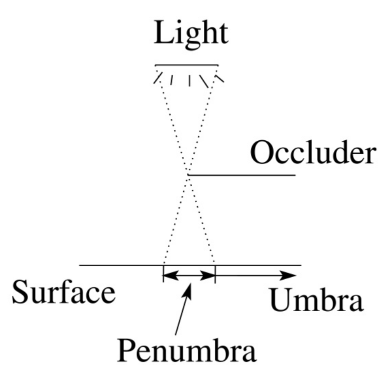
function *traceRay*(scene, **p**, **d,** id):
    (**q**, **N**, material) ← *intersect* (scene, **p**, **d**)
    I ← *shade*(…)
    R ← *jitteredReflectDirection*(**N**, -**d**, material, id)
    I ← I + material.k_r ∗ *traceRay*(scene, **q**, **R,** id)
    return I
end function



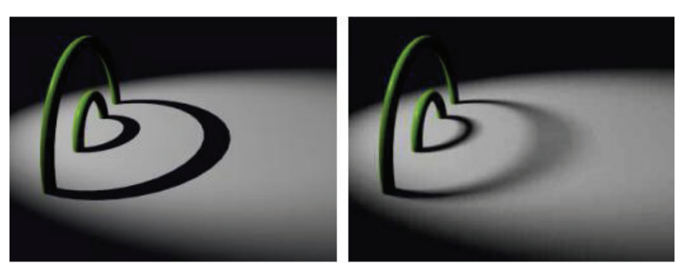
Pre-sampling glossy reflections (Quasi-Monte Carlo)



Soft shadows

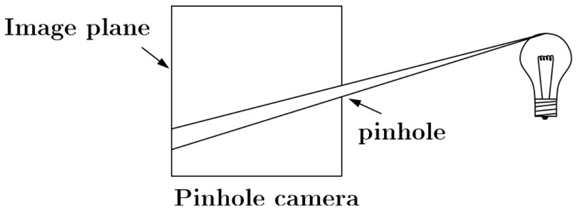


Distributing rays over light source area gives:

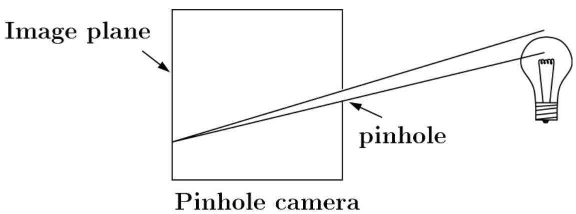


The pinhole camera, revisited

Recall the pinhole camera:

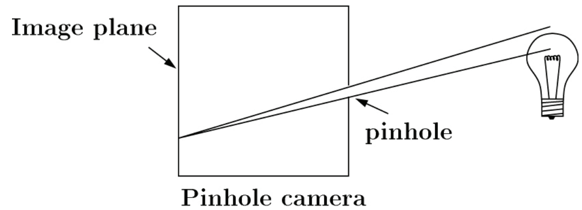


We can equivalently turn this around by following rays from the viewer:

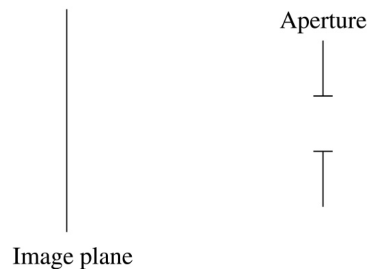


The pinhole camera, revisited

Given this flipped version:



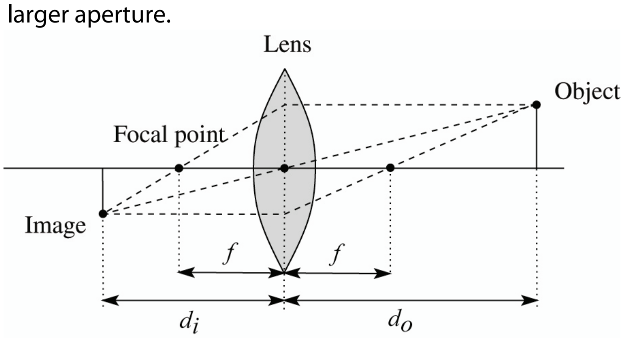
how can we simulate a pinhole camera more accurately?



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:

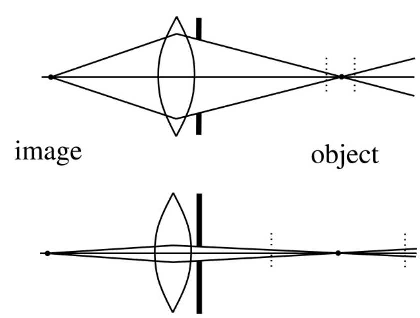
$$\frac{1}{d_i} + \frac{1}{d_o} = \frac{1}{f}$$

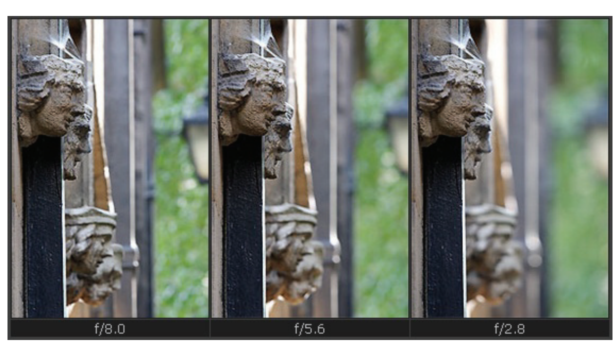
where *f* is the **focal length** of the lens.

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."

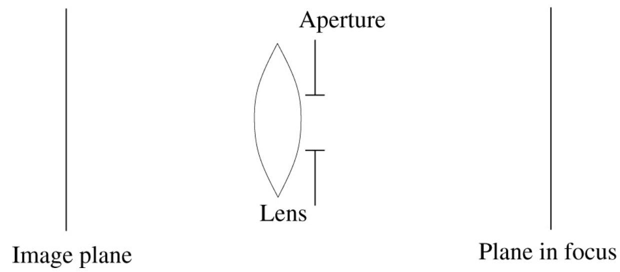




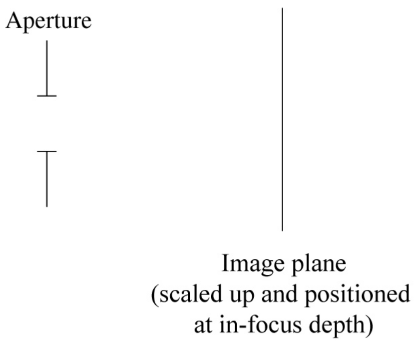
http://www.cambridgeincolour.com/tutorials/depth-of-field.htm

Simulating depth of field

Consider how rays flow between the image plane and the in-focus plane:



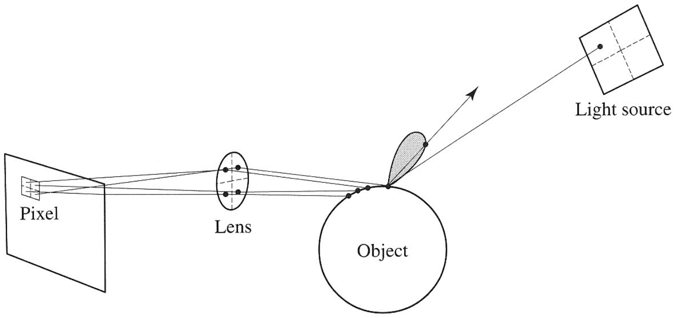
We can model this as simply placing our image plane at the in-focus location, in *front* of the finite aperture, and then distributing rays over the aperture (instead of the ideal center of projection):



Simulating depth of field, cont'd

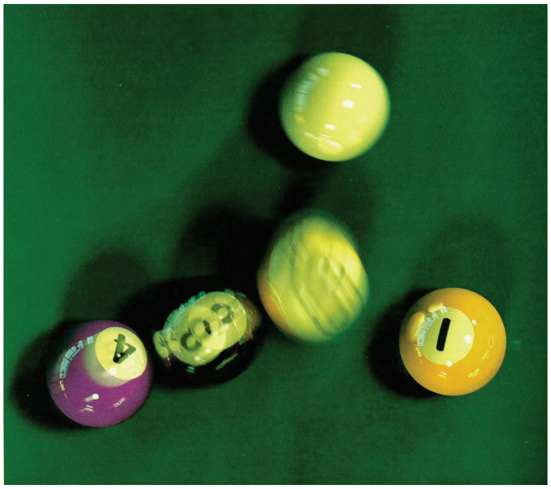
Chaining the ray id's

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.