Distribution Ray Tracing

Reading

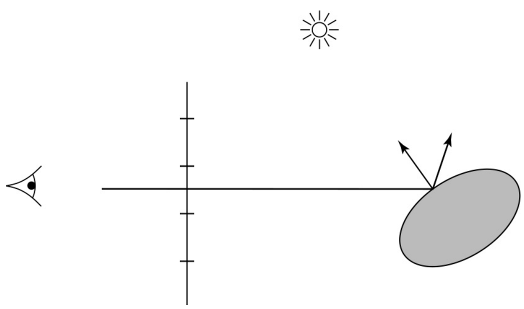
Required:

- Shirley, 10.11, 14.1-14.3

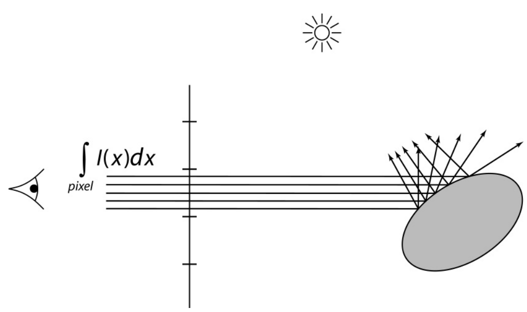
Further reading:

- 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

Surface reflection equation

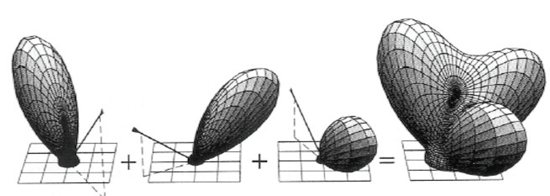
In reality, surfaces do not reflect in a mirror-like fashion.

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

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

How might we represent light from a single direction?

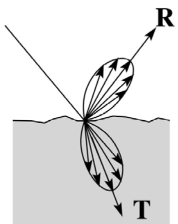
We can plot the reflected light as a function of viewing angle for multiple light source contributions:



Simulating gloss and translucency

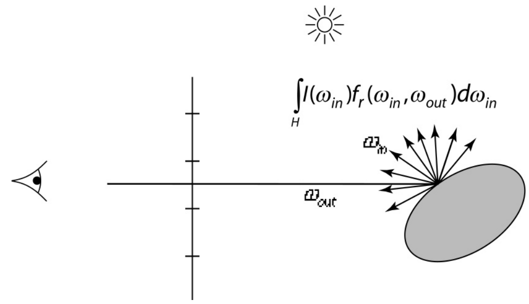
The mirror-like form of reflection, when used to approximate glossy surfaces, introduces a kind of aliasing, because we are under-sampling reflection (and refraction).

For example:



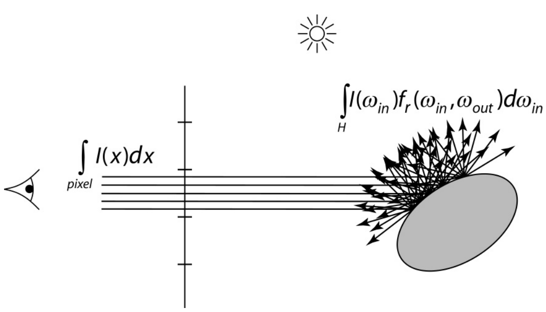
Distributing rays over reflection directions gives:

Reflection anti-aliasing



Reflection anti-aliasing

Full anti-aliasing



Full anti-aliasing…lots of nested integrals!

Computing these integrals is prohibitively expensive.

We'll look at ways to approximate integrals…

Approximating integrals

Let's say we want to compute the integral of a function:

$$F = \int f(x)dx$$

If $f(x)$ is not known analytically, but can be evaluated, then we can approximate the integral by:

$$F \approx \sum_{i=1}^{n} f(i\Delta x)\Delta x$$

where we have sampled n times at spacing Δx. We can think of these samples distributed over an interval w:

$$\Delta x = \frac{w}{n}$$

and re-write the summation as:

$$F \approx \frac{w}{n} \sum_{i=1}^{n} f(i\Delta x)$$

Evaluating an integral in this manner is called **quadrature**.

Integrals as expected values

An alternative to distributing the sample positions regularly is to distribute them **stochastically**.

Let's say the position in *x* is a random variable *X*, which is distributed according to *p*(*x*), a probability density function (strictly positive that integrates to unity).

Now let's consider a function of that random variable, *f*(*X*)/*p*(*X*). What is the expected value of this new random variable?

First, recall the expected value of a function *g*(*X*):

$$E[g(X)] = \int g(x)p(x)dx$$

Then, the expected value of *f*(*X*)/*p*(*X*) is:

Monte Carlo integration

Thus, given a set of samples positions, X_i, we can estimate the integral as:

$$F \approx \frac{1}{n}\sum_{i=1}^{n}\frac{f(X_i)}{p(X_i)}$$

This procedure is known as **Monte Carlo integration**.

The trick is getting as accurate as possible with as few samples as possible.

More concretely, we would like the variance of the estimate of the integral to be low:

$$V\left[\frac{f(X)}{p(X)}\right] = E\left[\left(\frac{f(X)}{p(X)}\right)^2\right] - E\left[\frac{f(X)}{p(X)}\right]^2$$

The name of the game is **variance reduction**…

Uniform sampling

One approach is uniform sampling (i.e., choosing *X* from a uniform distribution):

Importance sampling

A better approach, if *f*(*x*) is positive, would be to choose *p*(*x*) ~ *f*(*x*). In fact, this choice would be optimal.

Why don't we just do that?

Alternatively, we can use heuristics to guess where *f*(*x*) will be large and choose *p*(*x*) based on those heuristics. This approach is called **importance sampling**.

Stratified sampling

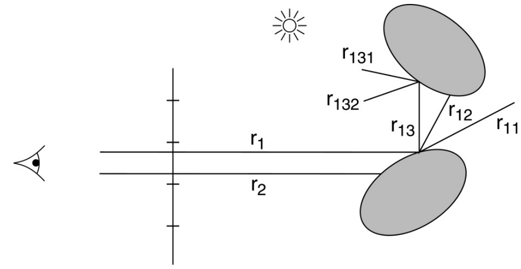
An improvement on importance sampling is **stratified sampling**.

The idea is that, given your probability function:

- You can break it up into bins of equal probability area (i.e., equal likelihood).
- Then choose a sample from each bin.

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:

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

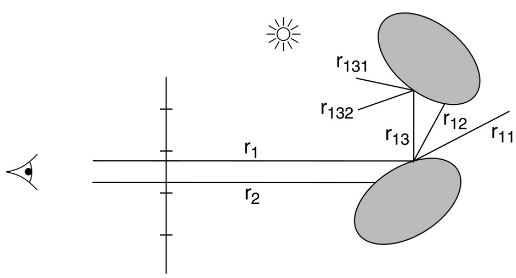
For a given primary ray, its intensity depends on secondary rays:

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

Substituting back in:

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

Summing over ray paths



We can incorporate tertiary rays next:

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

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

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

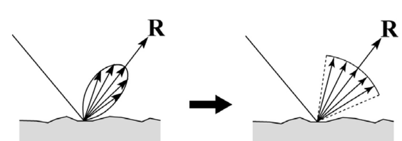
So, we can see that ray tracing is a way to approximate a complex, nested light transport integral with a summation over ray paths (of arbitrary length!).

Problem: too expensive to sum over all paths.

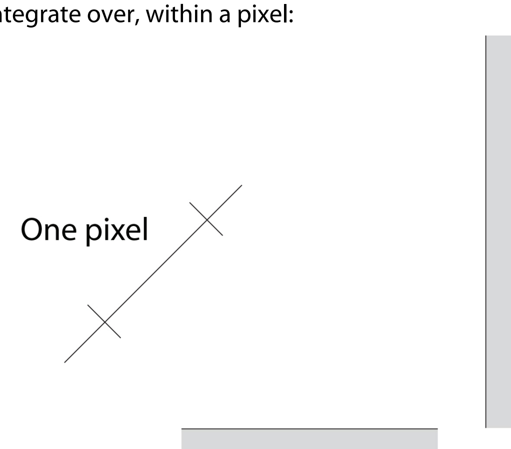
Solution: choose a small number of “good” paths.

Glossy reflection revisited

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

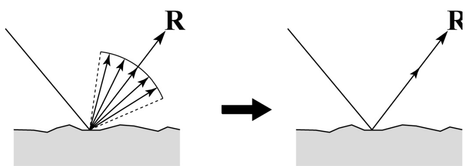


We can visualize the span of rays we want to integrate over, within a pixel:

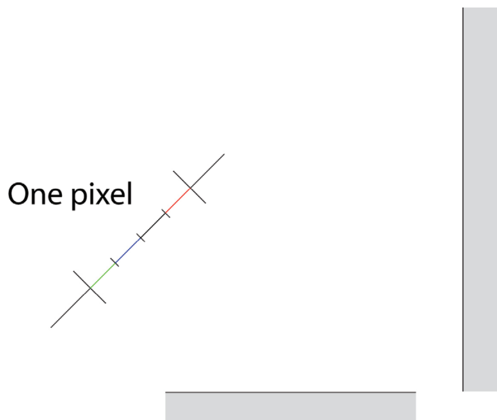


Whitted ray tracing

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

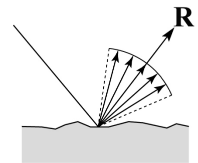


Thus, we render with anti-aliasing as follows:

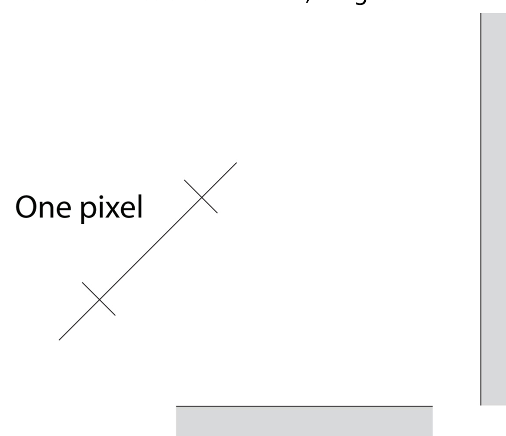


Monte Carlo path tracing

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

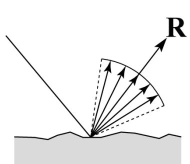


An alternative way to follow rays is by making random decisions along the way – a.k.a., Monte Carlo path tracing. If we distribute rays uniformly over pixels and reflection directions, we get:

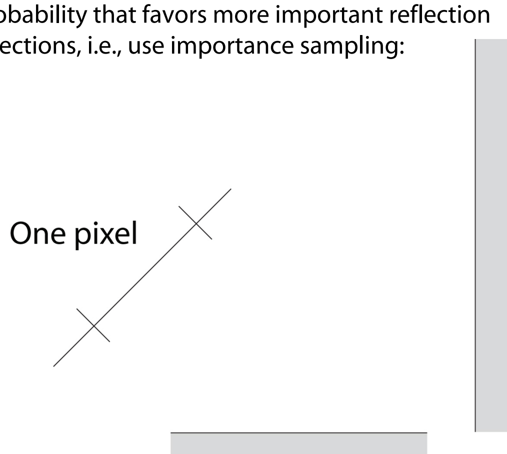


Importance sampling in path tracing

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

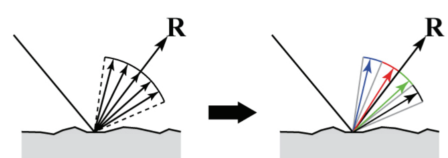


Let's now randomly choose rays, but according to a probability that favors more important reflection directions, i.e., use importance sampling:

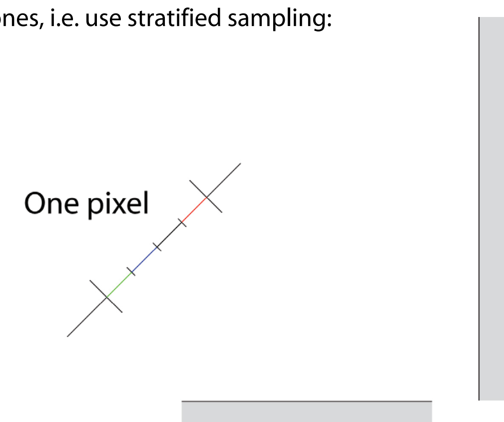


Stratified sampling in path tracing

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

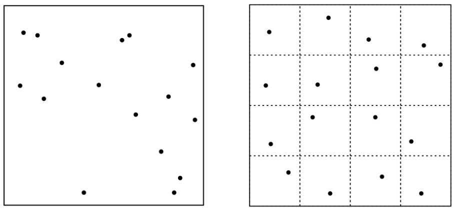


Now let's restrict our randomness to within these zones, i.e. use stratified sampling:



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.

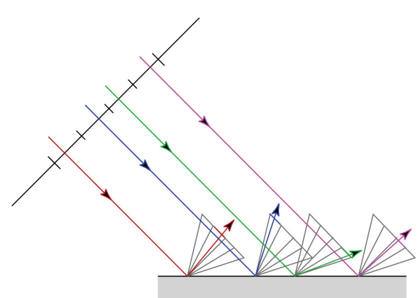
function *traceImage* (scene):
    for each pixel (i, j) in image **do**
        I(i, j) ← 0
        for each sub-pixel id in (i,j) **do**
            s ← *pixelToWorld*(jitter(i, j, id))
            p ← **COP**
            d ←(**s** - **p**).normalize()
            I(i, j) ← I(i, j) + *traceRay*(scene,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

A typical choice is numSubPixels = 4*4.

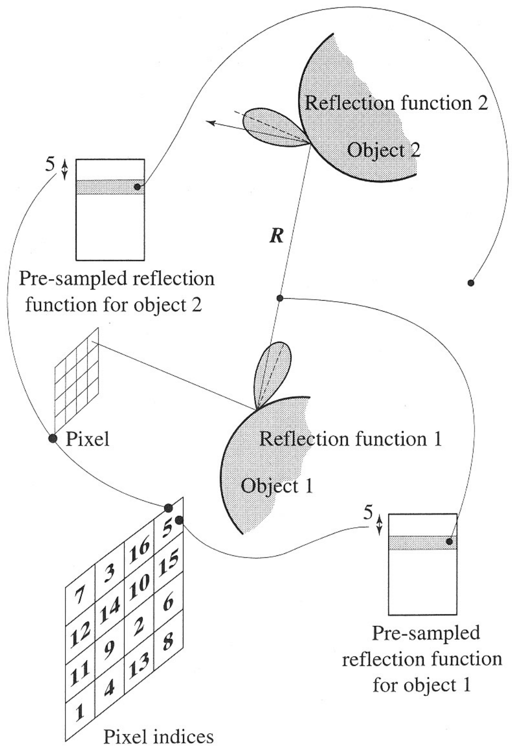
DRT pseudocode (cont'd)

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

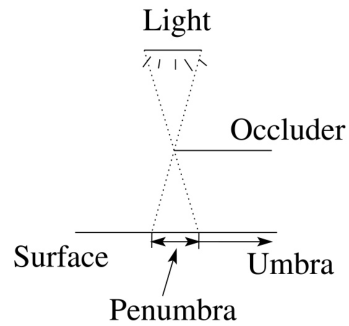
function *traceRay*(scene, **p**, **d,** id):
    (**q**, **N**, material) ← *intersect* (scene, **p**, **d**)
    I ← *shade*(…)
    R ← *jitteredReflectDirection*(**N**, -**d**, material, id)
    I ← I + material.k_r ∗ *traceRay*(scene, **q**, **R,** id)
    return I
end function



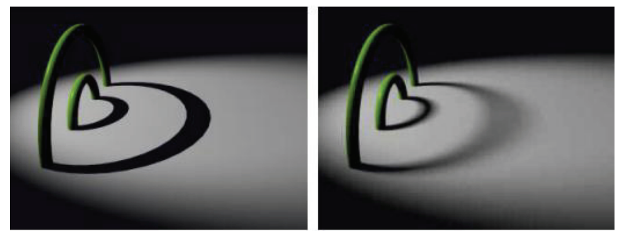
Pre-sampling glossy reflections (Quasi-Monte Carlo)



Soft shadows

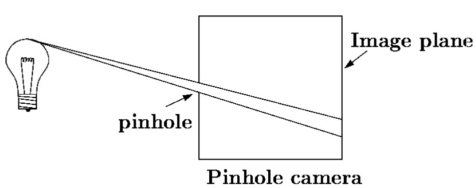


Distributing rays over light source area gives:

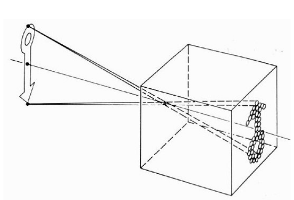


The pinhole camera

The first camera - “camera obscura” - known to Aristotle.

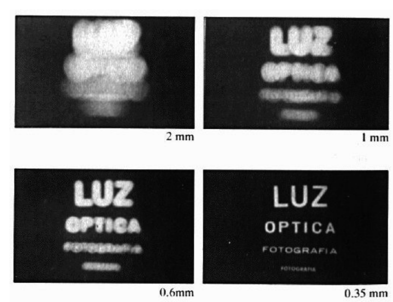


In 3D, we can visualize the blur induced by the pinhole (a.k.a., **aperture**):

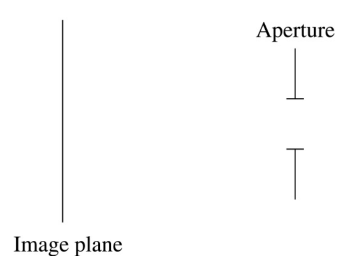


Q: How would we reduce blur?

Shrinking the pinhole

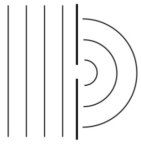


Q: How can we simulate a pinhole camera more accurately?

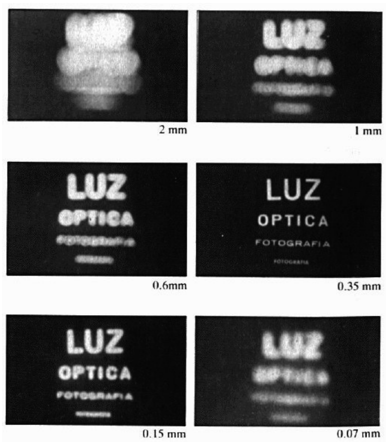


Q: What happens as we continue to shrink the aperture?

Shrinking the pinhole, cont’d



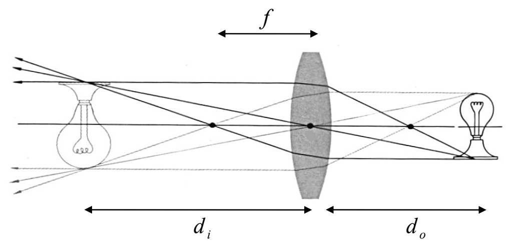
Diffraction



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:

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

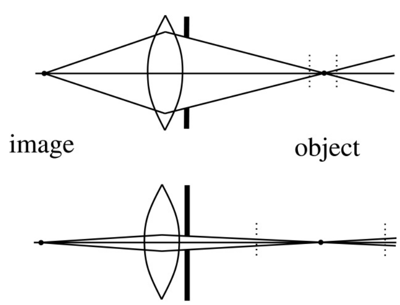
where f is the **focal length** of the lens.

Depth of field

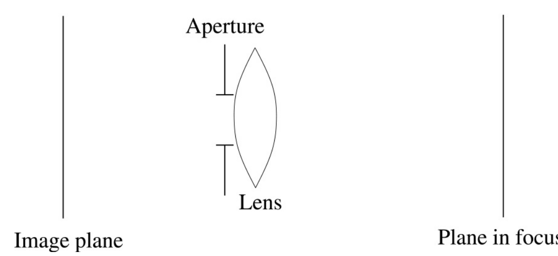
Lenses do have some limitations.

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.”



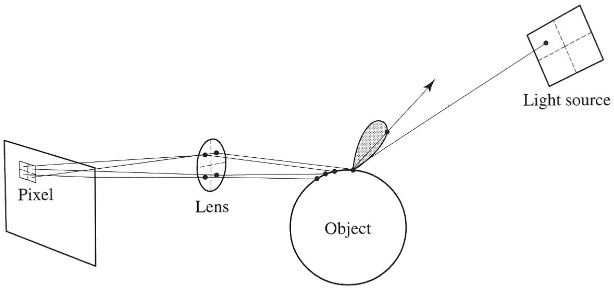
Simulating depth of field



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: