9. Distribution Ray Tracing

Reading

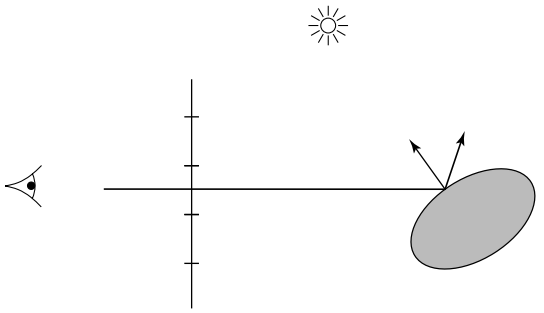
Required:

- Watt, sections 10.6 ,14.8.

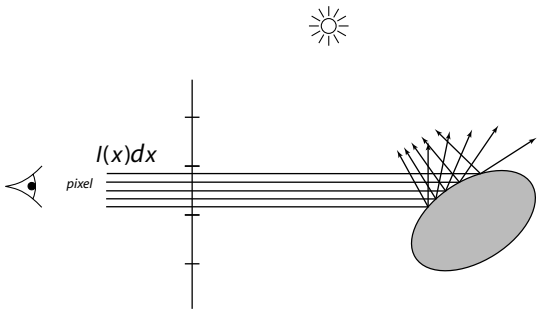
Further reading:

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

Pixel anti-aliasing



No anti-aliasing

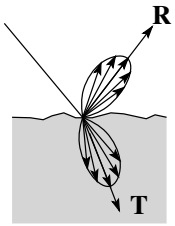


Pixel anti-aliasing

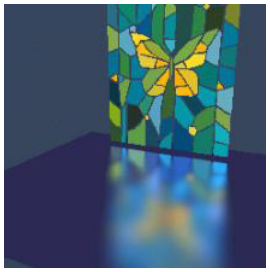
Simulating gloss and translucency

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

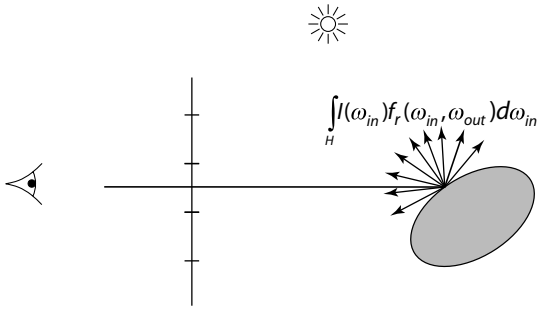
For example:



Distributing rays over reflection directions gives:

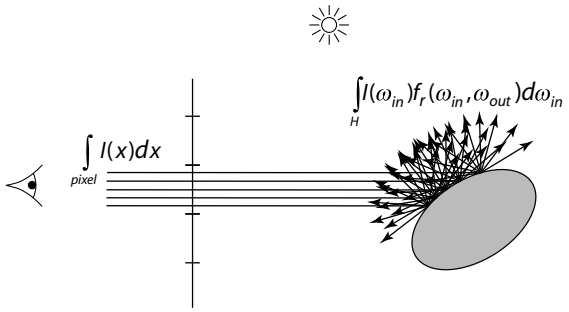


Reflection anti-aliasing



Reflection anti-aliasing

Full anti-aliasing



Full anti-aliasing…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:

$$F \approx \frac{1}{n}\sum f(i\Delta x)$$

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:

$$F \approx \frac{1}{n} \sum \frac{f(X_i)}{p(X_i)}$$

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

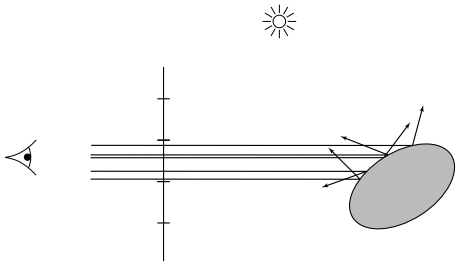
A better approach, if $f(x)$ is positive, would be to choose $p(x) \sim f(x)$. In fact, this choice would be optimal.

Why don't we just do that?

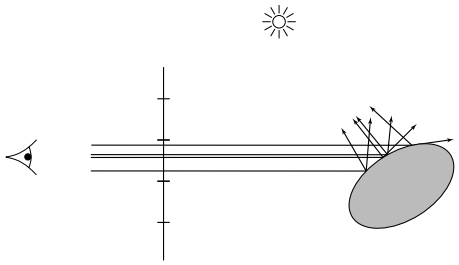
Alternatively, we can use heuristics to guess where $f(x)$ will be large. This approach is called **importance sampling**.

Importance sampling

We can apply this idea to ray tracing where we treat each ray path as a sample. This approach is called **Monte Carlo path tracing**.



Uniform sampling

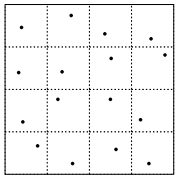


Importance sampling

Stratified sampling

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

E.g., for sub-pixel samples:



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

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.

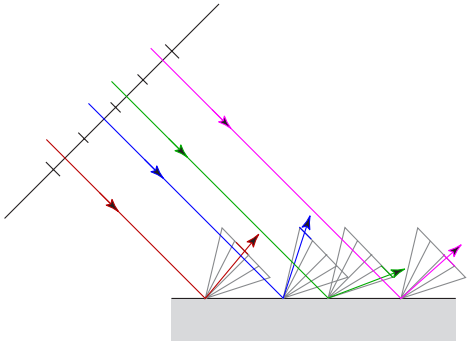
```
function traceImage (scene):
    for each pixel (i, j) in image do
        I(i, j) ← 0
        for each sub-pixel id in (i,j) do
            s ← pixelToWorld(jitter(i, j, id))
            p ← COP
            d ←(s - p).normalize()
            I(i, j) ← I(i, j) + traceRay(scene, p, d, id)
       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

A typical choice is numSubPixels = 4*4.

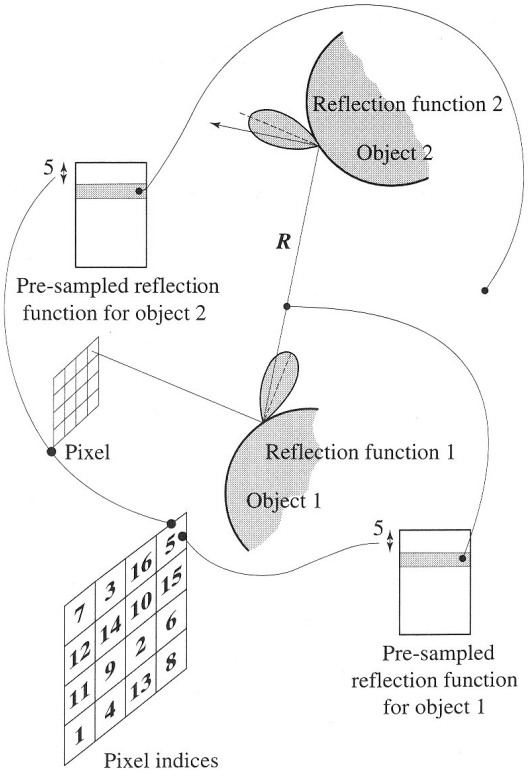
DRT pseudocode (cont'd)

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

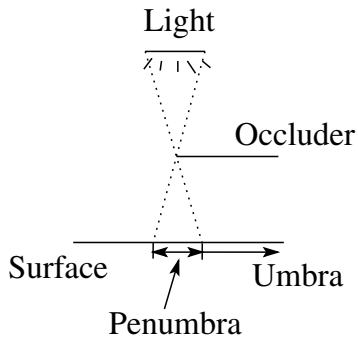
```
function traceRay(scene, p, d):
    (q, N, material) ← intersect (scene, p, d)
    I ← shade(…)
    R ← jitteredReflectDirection(N, -d, id)
    I ← I + material.k_r * traceRay(scene, q, R)
    return I
end function
```



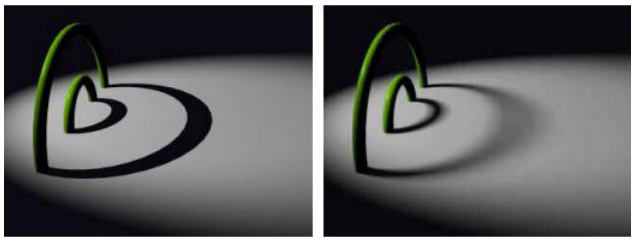
Pre-sampling glossy reflections



Soft shadows



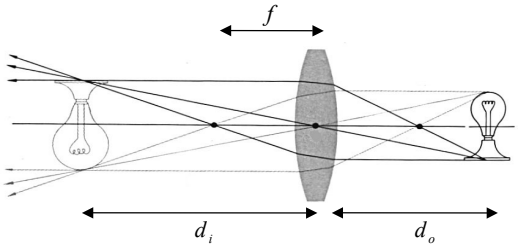
Distributing rays over light source area gives:



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:

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

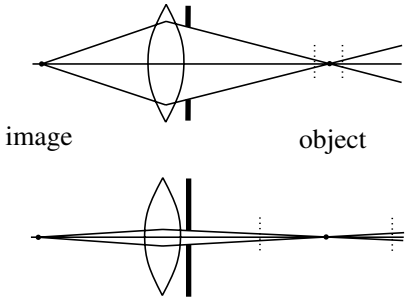
where *f* is the **focal length** of the lens.

Depth of field

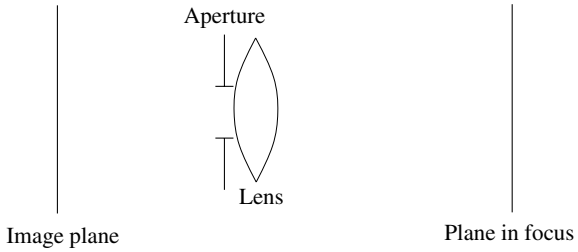
Lenses do have some limitations.

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."



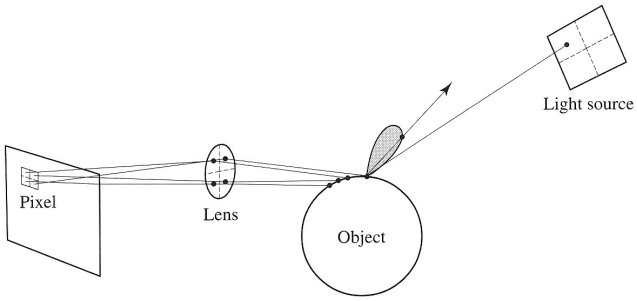
Simulating depth of field



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: