**Reading**

Required:
- Shirley, 13.11, 14.1-14.3

Further reading:

---

**Pixel anti-aliasing**

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)**:

\[ f_r(\omega_{\text{in}}, \omega_{\text{out}}) \]

Sometimes this is written as:

\[ f_r(\omega_{\text{in}} \rightarrow \omega_{\text{out}}) \]

Which we could visualize for a given \( \omega_{\text{in}} \):

\[ \frac{1}{A_{\text{pixel}}} \int l(x)dx \]

This is like a ray of light coming in at direction \( \omega_{\text{in}} \) and scattering into directions \( \omega_{\text{out}} \).
**Surface reflection equation**

BRDF’s exhibit reciprocity:

\[ f_r(\omega_{in} \rightarrow \omega_{out}) = f_r(\omega_{out} \rightarrow \omega_{in}) \]

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

[Diagram of BRDF](https://example.com/diagram)

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

This idea gives rise to the surface reflection equation:

\[
l(\omega_{out}) = \int_H l(\omega_{in}) f_r(\omega_{in} \rightarrow \omega_{out})(\omega_{in} \cdot \mathbf{N}) d\omega_{in}
\]

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:

[Diagram of glossy surface](https://example.com/diagram)

Distributing rays over reflection directions gives:

[Diagram of ray distribution](https://example.com/diagram)

---

**Reflection anti-aliasing**

[Diagram of reflection anti-aliasing](https://example.com/diagram)

**Pixel and reflection anti-aliasing**

[Diagram of pixel and reflection anti-aliasing](https://example.com/diagram)
Full anti-aliasing

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

We'll look at ways to approximate high-dimensional 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 \( \Delta x \). If these samples are distributed over an interval \( w \), then

\[ \Delta x = \frac{w}{n} \]

and the summation becomes:

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

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 (non-negative, integrates to unity).

Now let’s consider a function of that random variable, \( f(X) \) and define another function (also of that random variable) as:

\[ g(x) = \frac{f(x)}{p(x)} \]

What is the expected value of this new random variable \( g(X) \)?

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 a result 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 over an interval of width $w$ (i.e., choosing $X$ from a uniform distribution):

$$p(x) = \begin{cases} 1/w & |x| \leq w/2 \\ 0 & \text{otherwise} \end{cases}$$

Uniform sampling, cont’d

Suppose that the unknown function we are integrating happens to be a normalized box function of width $a$:

$$f(x) = \begin{cases} 1/a & |x| \leq a/2 \\ 0 & \text{otherwise} \end{cases}$$

Importance sampling

A better approach, if $f(x)$ is non-negative, 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 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:

\[ I_{\text{pixel}} = \frac{1}{n} \sum_{r} I(r) \]

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

\[ I(r) = \sum_{j} (r_{ij}) f_{r_{ij}} (r_{ij} \rightarrow r) \]

Substituting back in:

\[ I_{\text{pixel}} = \frac{1}{n} \sum_{i} \sum_{j} (r_{ij}) f_{r_{ij}} (r_{ij} \rightarrow r) \]

**Problem**: too expensive to sum over all paths.

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

---

**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:
Let’s 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:

Monte Carlo 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:

Importance 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:

Stratified sampling in path tracing

Here we see pure uniform vs. stratified sampling over a 2D pixel (here 16 rays/pixel):

Stratified sampling of a 2D pixel

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.

Random Stratified
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.kr * traceRay(scene, q, R, id)
  return I
end function

Pre-sampling glossy reflections (Quasi-Monte Carlo)

Now consider traceRay(), modified to handle (only) opaque glossy surfaces:
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: What happens as we continue to shrink the aperture?

Shrinking the pinhole, cont’d

Diffraction
The pinhole camera, revisited

We can think in terms of light heading toward the image plane:

![Diagram of pinhole camera with light rays](image1)

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

![Diagram of pinhole camera with rays](image2)

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 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.”

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:

![Diagram of image plane and lens with an aperture]

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):

![Diagram of aperture and image plane scaled and positioned at in-focus depth]

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:

![Diagram of ray tracing from light source to object and pixel]

DRT to simulate ________________

Distributing rays over time gives: