Distribution Ray Tracing

Brian Curless
CSE 557
Fall 2014

Reading

Required:

• Shirley, 13.11, 14.1-14.3

Further reading:

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

Pixel anti-aliasing

No anti-aliasing

Pixel anti-aliasing

All of this assumes that inter-reflection behaves in a mirror-like fashion...

BRDF

The diffuse+specular parts of the Blinn-Phong illumination model are a mapping from light to viewing directions:

\[ I = \begin{bmatrix} I_d \left( \mathbf{N} \cdot \mathbf{L} \right) + k_r \left( \mathbf{N} \cdot \frac{\mathbf{L} + \mathbf{V}}{\| \mathbf{L} + \mathbf{V} \|} \right) n_r \end{bmatrix} \]

\[ = f_r \left( \mathbf{L}, \mathbf{V} \right) \]

The mapping function \( f_r \) is often written in terms of incoming (light) directions \( \omega_{in} \) and outgoing (viewing) directions \( \omega_{out} \):

\[ f_r \left( \omega_{in}, \omega_{out} \right) \quad \text{or} \quad f_r \left( \omega_{in} \rightarrow \omega_{out} \right) \]

This function is called the Bi-directional Reflectance Distribution Function (BRDF).

Here’s a plot with \( \omega_{in} \) held constant:

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

BRDF’s can be quite sophisticated…
Light reflection with BRDFs

BRDF's exhibit Helmholtz reciprocity:

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

That means we can take two equivalent views of reflection. Suppose \( \omega_{in} = L \) and \( \omega_{out} = V \):

\[ f_r(L \rightarrow V) \]

\[ f_r(V \rightarrow L) \]

We can now think of the BRDF as weighting light coming in from all directions, which can be added up:

\[ \eta(V) = \int \eta(L)f_r(L \rightarrow V)(L \cdot N) dL \]

Or, written more generally:

\[ \eta(\omega_{out}) = \int \eta(\omega_{in})f_r(\omega_{in} \rightarrow \omega_{out})(\omega_{in} \cdot N) d\omega_{in} \]

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

Pixel and reflection anti-aliasing
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 (or is not easy to integrate), but can be readily evaluated, then we can approximate the integral by sampling.

Our approximate integral value is then something like:

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

where we have sampled $n$ times at spacing $\Delta x$.

Evaluating an integral in this manner is called quadrature.

Q: What happens if we have integrals over $d$ dimensions, sampling $n$ times in each dimension?

Approximating integrals (cont’d)

We can also do uneven sampling:

Our approximate integral value is then something like:

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

where we have sampled $n$ times at variable spacing $\Delta x_i$.

We can think of the $\Delta x_i$ as weights on the samples.

Q: When the $x_i$ are more closely spaced, do they get larger or smaller weight?

Q: Which $x_i$ should we pick?

A stochastic approach

A good approach to distributing samples unevenly across many dimensions is to do so stochastically.

Quick review first… For 1D, 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). Then:

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

$$ V[X] = E[X^2] - (E[X])^2 = \int x^2 \, p(x) \, dx - (\int x \, p(x) \, dx)^2 $$

$$ E[eX] = \int e^x \, p(x) \, dx = e^{E[X]} $$

$$ V[eX] = \int (e^x)^2 \, p(x) \, dx - (\int e^x \, p(x) \, dx)^2 = e^{E[X]} - E[eX]^2 $$

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

$$ V[g(X)] = \int (g(x))^2 \, p(x) \, dx - E[g(X)]^2 $$

We can also show that for independent random variables $X$ and $Y$:

$$ E[X + Y] = \int (x + y) \, p(x) \, p(y) \, dx \, dy $$

$$ V[X + Y] = V[X] + V[Y] $$
Monte Carlo integration

Consider this function of a random variable:

\[
\frac{f(X)}{p(X)}
\]

where \( f(x) \) is the function we’d like to integrate. What is its expected value?

\[
\mathbb{E}\left\{ \frac{f(X)}{p(X)} \right\} = \int \frac{f(x)}{p(x)} p(x) \, dx = \int f(\omega) \, d\omega = F
\]

Computing this expected value requires the analytic integral \( F \), which is not available.

Monte Carlo integration (cont’d)

Instead, we can estimate the expected value of \( f(X)/p(X) \) (and thus the integral \( F \)) as:

\[
\tilde{F} = \frac{1}{n} \sum_{i=1}^{n} \frac{f(X_i)}{p(X_i)}
\]

where the \( X_i \) are independent and identically distributed (i.i.d.) random variables, each with distribution \( p(x) \).

This procedure is known as Monte Carlo integration.

Note that \( \tilde{F} \) is a random variable built by summing over functions of random variables \( f(X_i)/p(X_i) \), where each of these functions of a random variable is itself a random variable. What is its expected value of \( \tilde{F} \)?

\[
\mathbb{E}[\tilde{F}] = \mathbb{E}\left[ \frac{1}{n} \sum_{i=1}^{n} \frac{f(X_i)}{p(X_i)} \right] = \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}\left[ \frac{f(X_i)}{p(X_i)} \right] = \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}\left[ \frac{f(X_i)}{p(X_i)} \right]
\]

Uniform sampling

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

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

Suppose we now try to estimate the integral of \( f(x) \) with uniform sampling over an interval of width \( w \) (i.e., choosing \( X \) from a uniform distribution):

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

where \( w \geq a \).

If we can get the variance to 0, then \( \tilde{F} \) must be a single value which must be equal to \( F \). Getting to 0 is hard, but getting to low variance is an important goal.

What variables and/or functions are under our control here? \( X, p(x) \)
**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.

\[
\frac{f(x)}{f(x)p(x)} = \frac{1}{E_x} = 0
\]

\[
\int p(x)dx = 1 = \int f(x)dx = K\int f(x)dx = \int f(x)dx
\]

\[K = \frac{1}{\int f(x)dx}\]

**Why don’t we just do that?**

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:

\[I_{\text{pixel}} = \frac{1}{n} \sum_i I_i\]

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

\[I_i = \sum_j I_{ij} f_{ij}(t_j)\]

Substituting back in:

\[I_{\text{pixel}} = \frac{1}{n} \sum_i \sum_j I_{ij} f_{ij}(t_j)\]

**Summing over ray paths**

We can incorporate tertiary rays next:

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

\[I_{\text{pixel}} = \frac{1}{n} \sum_j \sum_k \sum_i I_i f_{ij}(t_j) f_{jk}(t_k)\]

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

\[r_i \rightarrow r_j \rightarrow r_k\]

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.
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:

One pixel

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

Thus, we render with anti-aliasing as follows:

One pixel

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:

One pixel

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:

One pixel
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 diagram]

Stratified sampling of a 2D pixel

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

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:

```plaintext
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)

Distributing rays over light source area gives:

Soft shadows

In 3D, we can visualize the blur induced by the pinhole (a.k.a., \textit{aperture}):

Q: How would we reduce blur?

The pinhole camera

The first camera - "camera obscura" - known to Aristotle.
Shrinking the pinhole

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

The pinhole camera, revisited

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

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 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):
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 **motion blur**

Distributing rays over time gives: