Distribution Ray Tracing

Brian Curless
CSE 457
Autumn 2010

Reading

Required:
- Shirley, section 10.11

Further reading:
- Watt, sections 10.4-10.5

Pixel anti-aliasing

No anti-aliasing

Pixel anti-aliasing

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

BRDF, revisited

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

\[ f_r(\omega_i \rightarrow \omega_o, \omega_{os}) \]

BRDFs exhibit reciprocity:

\[ f_r(\omega_o \rightarrow \omega_{os}, \omega_i) = f_r(\omega_{os} \rightarrow \omega_o, \omega_i) \]

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

\[ f_r(L \rightarrow V) \quad f_r(V \rightarrow L) \]

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

\[ \hat{r}(V) = \int \hat{r}(L) f_r(L \rightarrow V) \hat{L} \cdot \hat{N} \, d\omega_i \]

Or, written more generally:

\[ \hat{r}(\omega_{os}) = \int \hat{r}(\omega_o) f_r(\omega_i \rightarrow \omega_{os}) (\omega_{os} \cdot \hat{N}) \, d\omega_i \]
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

\[ \int \frac{\rho(x, y) \cdot \mathbf{N}}{A_{\text{pixel pile}}} \, dx \, dy \]

Pixel and reflection anti-aliasing

\[ \frac{1}{A_{\text{pixel pile}}} \int \rho(x) \, dx \]

\[ \int \rho(x) \, dx \]

Pixel and reflection anti-aliasing

Reflection 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:

One 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:

One pixel

Monte Carlo path tracing

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

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:

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**.

One pixel

---

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 inject noise into the solution (slightly grainer 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 distribution) 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.

```plaintext
function TraceImage (scene):
    for each pixel (i, j) in Image do
        l(i, j) ← 0
        for each sub-pixel id in (i, j) do
            s ← pixelToWorld(jitter(l(i, j), id))
            p ← COP
            d ← (s - p).normalize()
            l(i, j) ← l(i, j) + TraceRay (scene, p, d, id)
        end for
    end for
end function
```

A typical choice is numSubPixels = 5 x 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)
    l ← shade(...)
    R ← l*metalReflectDirection(N, d, material, id)
    l ← 1 + material.k_r * traceRay(scene, q, R, id)
    return l
end function

Pre-sampling glossy reflections
(Quasi-Monte Carlo)

The pinhole camera, revisited

Recall the pinhole camera:

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

Soft shadows

Distributing rays over light source area gives:
The pinhole camera, revisited

Given this flip-flop version:

[Image of a pinhole camera diagram]

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.

[Image of a lens diagram]

For a "thin" lens, we can approximately calculate where an object point will be in focus using the Thin Lens formula:

\[
\frac{1}{f} = \frac{1}{d_i} + \frac{1}{d_o}
\]

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."

[Image of depth of field diagram]

Simulating depth of field

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

[Image of a lens diagram with rays]

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):

[Image of aperture and in-focus plane diagram]
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:

Light source

Pixel

Lens

Object

DRT to simulate **motion blur**

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.