13 Distribution Ray Tracing

In Distribution Ray Tracing (hereafter abbreviated as "DRT"), our goal is to render a scene as accurately as possible. Whereas Basic Ray Tracing computed a very crude approximation to radiance at a point, in DRT we will attempt to compute the integral as accurately as possible. Additionally, the intensity at each pixel will be properly modeled as an integral as well. Since these integrals cannot be computed exactly, we must resort to numerical integration techniques to get approximate solutions.

Aside:
When originally introduced, DRT was known as "Distributed Ray Tracing." We will avoid this name to avoid confusion with distributed computing, especially because some ray-tracers are implemented as parallel algorithms.

13.1 Problem statement

Recall that, shading at a surface point is given by:

$$L(\vec{d}_e) = \int_\Omega \rho(\vec{d}_e, \vec{d}_i(\phi,\theta))\, L(-\vec{d}_i(\phi,\theta))\, (\vec{n}\cdot\vec{d}_i)\, d\omega \tag{1}$$

This equation says that the radiance emitted in direction $\vec{d}_e$ is given by integrating over the hemisphere Ω the BRDF ρ times the incoming radiance $L(-\vec{d}_i(\phi,\theta))$. Directions on the hemisphere are parameterized as

$$\vec{d}_i = (\sin\theta\sin\phi, \sin\theta\cos\phi, \cos\theta) \tag{2}$$

The differential solid angle $d\omega$ is given by:

$$d\omega = \sin\theta d\theta d\phi \tag{3}$$

and so:

$$L(\vec{d}_e) = \int_{\phi\in[0,2\pi]}\int_{\theta\in[0,\pi/2]} \rho(\vec{d}_e, \vec{d}_i(\phi,\theta))\, L(-\vec{d}_i(\phi,\theta))\, (\vec{n}\cdot\vec{d}_i)\ \sin\theta d\theta d\phi \tag{4}$$

This is an integral over all incoming light directions, and we cannot compute these integrals in closed-form. Hence, we need to develop numerical techniques to compute approximations.

Intensity of a pixel. Up to now, we've been engaged in a fiction, namely, that the intensity of a pixel is the light passing through a single point on an image plane. However, real sensors — including cameras and the human eye — cannot gather light at an infinitesimal point, due both to the nature of light and the physical properties of the sensors. The actual amount of light passing through any infinitesimal region (a point) is infinitesimal (approaching zero) and cannot be measured. Instead light must be measured within a region. Specifically, the image plane (or retina) is divided up into an array of tiny sensors, each of which measures the total light incident on the area of the sensor.

As derived previously, the image plane can be parameterized as $\bar{p}(\alpha, \beta) = \bar{p}_0 + \alpha\vec{u} + \beta\vec{v}$. In camera coordinates, $\bar{p}_0^c = (0, 0, f)$, and the axes correspond to the x and y axes: $\vec{u}^c = (1, 0, 0)$ and $\vec{v}^c = (0, 1, 0)$. Then, we placed pixel coordinates on a grid: $\bar{p}_{i,j}^c = (L+i\Delta i, T+j\Delta j, f) = \bar{p}_0+\alpha$, where $\Delta i = (R-L)/n_c$ and $\Delta j = (B-T)/n_r$, and L, T, B, R are the boundaries of the image plane.

We will now view each pixel as an area on the screen, rather than a single point. In other words, pixel (i, j) is all values $\bar{p}(\alpha, \beta)$ for $\alpha_{min} \leq \alpha < \alpha_{max}$, $\beta_{min} \leq \beta < \beta_{max}$. The bounds of each pixel are: $\alpha_{min} = L + i\Delta i, \alpha_{max} = L + (i + 1)\Delta i, \beta_{min} = T + j\Delta j$, and $\beta_{max} = T + (j + 1)\Delta j$. (In general, we will set things up so that this rectangle is a square in world-space.) For each point on the image plane, we can write the ray passing through this pixel as

$$\vec{d}(\alpha, \beta) = \frac{\bar{p}(\alpha, \beta) - \bar{e}}{||\bar{p}(\alpha, \beta) - \bar{e}||} \tag{5}$$

To compute the color of a pixel, we should compute the total light energy passing through this rectangle, i.e., the flux at that pixel:

$$\Phi_{i,j} = \int_{\alpha_{min} \leq \alpha < \alpha_{max}} \int_{\beta_{min} \leq \beta < \beta_{max}} H(\alpha, \beta) d\alpha d\beta \tag{6}$$

where $H(\alpha, \beta)$ is the incoming light (irradiance) on the image at position α, β. For color images, this integration is computed for each color channel. Again, we cannot compute this integral exactly.

Aside:
An even more accurate model of a pixel intensity is to weight rays according to how close they are to the center of the pixel, using a Gaussian weighting function.

13.2 Numerical integration

We begin by considering the general problem of computing an integral in 1D. Suppose we wish to integrate a function $f(x)$ from 0 to D:

$$S = \int_0^D f(x) dx \tag{7}$$

Visually, this corresponds to computing the area under a curve. Recall the definition of the integral. We can break the real line into a set of intervals centered at uniformly-spaced points $x_1, ..., x_N$. We can then define one rectangle on each interval, each width D/N and height $f(x_i)$. The total area of these rectangles will be approximately the same as the area under the curve. The area of each rectangle is $f(x_i)D/N$, and thus the total area of all rectangles together is:

$$S_N = \frac{D}{N}\sum_i f(x_i) \tag{8}$$

Hence, we can use S_N as an approximation to S. Moreover, we will get more accuracy as we increase the number of points:

$$\lim_{N\to\infty} S_N = S \tag{9}$$

There are two problems with using uniformly-spaced samples for numerical integration:

- Some parts of the function may be much more "important" than others. For example, we don't want to have to evaluate $f(x)$ in areas where it is almost zero. Hence, you need to generate many, many x_i values, which can be extremely slow.
- Uniformly-spaced samples can lead to *aliasing artifacts*. These are especially noticable when the scene or textures contain repeated (periodic) patterns.

In ray-tracing, each evaluation of $f(x)$ requires performing a ray-casting operation and a recursive call to the shading procedure, and is thus very, very expensive. Hence, we would like to design integration procedures that use as few evaluations of $f(x)$ as possible.

To address these problems, randomized techniques known as **Monte Carlo integration** can be used.

13.3 Simple Monte Carlo integration

Simple Monte Carlo addresses the problem of aliasing, and works as follows. We randomly sample N values x_i in the interval $[0, D]$, and then evaluate the same sum just as before:

$$S_N = \frac{D}{N}\sum_i f(x_i) \tag{10}$$

It turns out that, if we have enough samples, we will get just as accurate a result as before; moreover, aliasing problems will be reduced.

Aside:
Formally, it can be shown that the expected value of S_N is S. Moreover, the variance of S_N is proportional to N, i.e., more samples leads to better estimates of the integral.

In the C programming language, the random sampling can be computed as `rand() * D`.

Aside:
Monte Carlo is a city near France and Italy famous for a big casino. Hence, the name of the Monte Carlo algorithm, since you randomly sample some points and gamble that they are representative of the function.

13.4 Integration at a pixel

To compute the intensity of an individual pixel, we need to evaluate Equation 6). This is a 2D integral, so we need to determine K 2D points (α_i, β_i), and compute:

$$\Phi_{i,j} \approx \frac{(\alpha_{max} - \alpha_{min})(\beta_{max} - \beta_{min})}{K} \sum_{i=1}^{K} H(\alpha_i, \beta_i) \tag{11}$$

In other words, we pick N points withnin the pixel, cast a ray through each point, and then average the intensities of the rays (scaled by the pixel's area $(\alpha_{max} - \alpha_{min})(\beta_{max} - \beta_{min})$. These samples can be chosen randomly, or uniformly-spaced.

Example:
The simplest way to compute this is by uniformly-spaced samples (α_m, β_n):

$$\alpha_m = (m-1)\Delta\alpha, \quad \Delta\alpha = (\alpha_{max} - \alpha_{min})/M \tag{12}$$

$$\beta_n = (n-1)\Delta\beta, \quad \Delta\beta = (\beta_{max} - \beta_{min})/N \tag{13}$$

and then sum:

$$\Phi_{i,j} \approx \Delta\alpha\Delta\beta \sum_{m=1}^{M} \sum_{n=1}^{N} H(\alpha_m, \beta_n) \tag{14}$$

However, Monte Carlo sampling — in which the samples are randomly-spaced — will usually give better results.

13.5 Shading integration

Our goal in shading a point is to compute the integral:

$$L(\vec{d}_e) = \int_{\phi\in[0,2\pi]} \int_{\theta\in[0,\pi/2]} \rho(\vec{d}_e, \vec{d}_i(\phi,\theta))\, L(-\vec{d}_i(\phi,\theta))\, (\vec{n} \cdot \vec{d}_i)\, \sin\theta \, d\theta d\phi \tag{15}$$

We can choose uniformly-spaced values of ϕ and θ values as follows:

$$\theta_m = (m-1)\Delta\theta, \quad \Delta\theta = (\pi/2)/M \tag{16}$$

$$\phi_n = (n-1)\Delta\phi, \quad \Delta\phi = 2\pi/N \tag{17}$$

This divides up the unit hemisphere into MN solid angles, each with area approximately equal to $\sin\theta\Delta\theta\Delta\phi$. Applying 2D numerical integration gives:

$$L(\vec{d}_e) \approx \sum_{m=1}^{M} \sum_{n=1}^{N} \rho(\vec{d}_e, \vec{d}_i(\phi,\theta))\, L(-\vec{d}_i(\phi,\theta))\, (\vec{n} \cdot \vec{d}_i)\, \sin\theta\, \Delta\theta\, \Delta\phi \tag{18}$$

Once you have all the elements in place (e.g., the ray-tracer, the BRDF model, etc.), evaluating this equation is actually quite simple, and doesn't require all the treatment of special cases required for basic ray-tracing (such as specular, diffuse, mirror, etc.). However, it is potentially much slower to compute.

13.6 Stratified Sampling

A problem with Simple Monte Carlo is that, if you use a small number of samples, these samples will be spaced very irregularly. For example, you might be very unlucky and get samples that don't place any samples in some parts of the space. This can be addressed by a technique called stratified sampling: divide the domain into K-uniformly sized regions, and randomly sample J points x_i within each region; then sum $\frac{D}{N}\sum_i f(x_i)$ as before.

13.7 Non-uniformly spaced points

Quite often, most of the radiance will come from a small part of the integral. For example, if the scene is lit by a bright point light source, then most of the energy comes from the direction to this source. If the surface is very shiny and not very diffuse, then most of the energy comes from the reflected direction. In general, it is desirable to sample more densely in regions where the function changes faster and where the function values are large. The general equation for this is:

$$S_N = \sum_i f(x_i) d_i \tag{19}$$

where d_i is the size of the region around point x_i. Alternatively, we can use stratified sampling, and randomly sample J values within each region. How we choose to define the region sizes and spaces depends on the specific integration problem. Doing so can be very difficult, and, as a consequence, deterministic non-uniform spacing is normally used in graphics; instead, importance sampling (below) is used instead.

13.8 Importance sampling

The method of **importance sampling** is a more sophisticated form of Monte Carlo that allows non-uniform sample spacing. Instead of sampling the points x_i uniformly, we sample them from another probability distribution function (PDF) $p(x)$. We need to design this PDF so that it gives us more samples in regions of x that are more "important," e.g., values of $f(x)$ are larger. We can then approximate the integral S as:

$$S_N = \frac{1}{N} \sum_i \frac{f(x_i)}{p(x_i)} \tag{20}$$

If we use a uniform distribution: $p(x) = 1/D$ for $x \in [0, D]$, then it is easy to see that this procedure reduces to Simple Monte Carlo. However, we can also use something more sophisticated, such as a Gaussian distribution centered around the point we expect to provide the greatest contribution to the intensity.

13.9 Distribution Ray Tracer

```
for each pixel (i,j)
    < choose $N$ points $\bar{x}_k$ within the pixel's area >
    for each sample $k$
        < compute ray $\vec{\mathbf{r}}_k(\lambda) = \vec{\mathbf{p}}_k + \lambda\vec{\mathbf{d}}_k$ where $\vec{\mathbf{d}}_k = \vec{\mathbf{p}}_k - \vec{\mathbf{e}}$ >
        $I_k$ = rayTrace( $\vec{\mathbf{p}}_k$, $\vec{\mathbf{d}}_k$, 1)
    end for
    setpixel(i, j, $\Delta i \Delta j \sum_k I_k/N$)
end for
```

The rayTrace and findFirstHit procedures are the same as for Basic Ray Tracing. However, the new shading procedure uses numerical integration:

```
distRtShade(OBJ, $\vec{\mathbf{p}}$, $\vec{\mathbf{n}}$, $\vec{\mathbf{d}}_e$, depth)
    < choose $N$ directions $(\phi_k, \theta_k)$ on the hemisphere >
    for each direction $k$
        $I_k$ = rayTrace($\vec{\mathbf{p}}$, $\vec{\mathbf{d}}_k$, depth+1)
    end for
    return $\Delta\theta\Delta\phi \sum_k \rho(\vec{\mathbf{d}}_e, \vec{d}_i(\phi_k, \theta_k)) I_k \sin\theta_k$
```