Distribution Ray Tracing: Summary

April 24, 2021

Distribution (or distributed) ray tracing refers to the concept of simulating various visual effects at little to no extra cost that you already pay for multi-sampling. The technique is first introduced by Cook et al. in 1984 in a Siggraph paper [1]. The effects that it includes are:

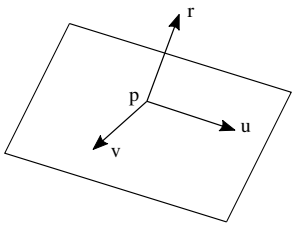
- Motion blur
- Depth of field
- Glossy (imperfect mirror) reflections
- Soft shadows (requires area lights)

Motion blur: This effect can be produced by adding a random *time* parameter to each of your rays. We usually use the symbol xi (ξ) to refer to a random number uniformly distributed between zero and one (i.e. $\xi \in [0, 1)$). As scary as this symbol looks, fear not, it just represents a single number. So to simulate this effect we set the time parameter of each ray to ξ. You can use any respected pseudo-random number generator for this purpose.

Secondly, we must define a motion vector for each object that we want to exhibit motion blur. In general this can be an arbitrary transformation but for simplicity we will assume that a motion vector represents only a translation.

With these tools in place, it is easy to implement motion blur. If a primary ray has a time parameter of 0, it will intersect the moving object at the beginning of its motion. On the other extreme, if a ray has a time parameter of 1, it will intersect the object at the end of its motion. Rays with times in between will intersect the object at positions linearly scaled in between these two extremes. It is important to note that the time parameter is randomly defined only for primary rays. Other rays spawned from the primary rays should have the same time parameter of its mother ray.

Glossy reflections. This effect is generally used to simulate metallic objects that are not polished. Brushed metal is an example. It allows a reflection ray to be sent at an angle that is somewhat off from the perfect reflection direction. This effect again involves randomness. Assume that $\mathbf{r}$ is a unit vector along the perfect reflection vector. We must define an orthonormal basis around this vector as shown in the following figure.



The vectors $\mathbf{u}$ and $\mathbf{v}$ are on the same plane and they are perpendicular to each other as well as to $\mathbf{r}$. Furthermore they are also unit vectors. Such a coordinate system is known as a local orthonormal basis created at point p. We can create this basis as follow:

1. Set the minimum (in absolute value sense) component of $\mathbf{r}$ to 1 to create a new vector $\mathbf{r}'$. This creates a vector that is not colinear with $\mathbf{r}$.
2. $\mathbf{u} = normalize(\mathbf{r} \times \mathbf{r}')$
3. $\mathbf{v} = \mathbf{r} \times \mathbf{u}$

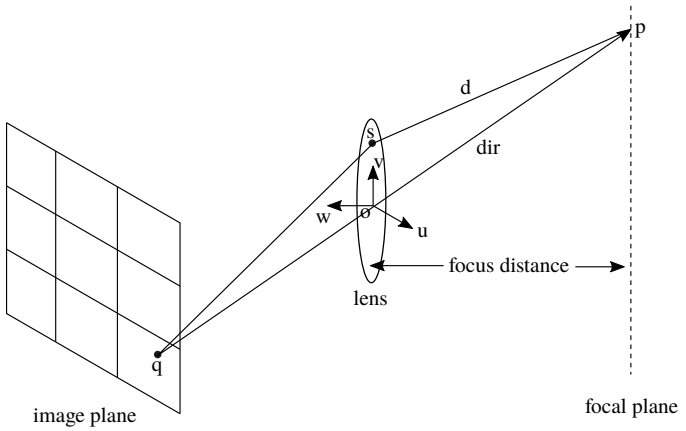
Normalization of v is not necessary if $\mathbf{r}$ and $\mathbf{u}$ are already normalized. Note that $\mathbf{u}$ and $\mathbf{v}$ vectors are unique up to a rotation around $\mathbf{r}$. Once this basis is created the real reflection vector can be calculated as:

$$\mathbf{r_{real}} = normalize\left(\mathbf{r} + roughness\left(\xi_1\mathbf{u} + \xi_2\mathbf{v}\right)\right). \tag{1}$$

Here ξ's are normalized uniform random numbers as before and *roughness* is a parameter of the surface. The greater the roughness value the more rough the metallic surface should appear.

Depth of field. Real cameras have finite aperture as opposed to the pinhole model we have been using so far. Without a proper lens, a finite aperture camera is guaranteed to produce blurry images on its image plane. A lens is a glass contraption which allows focusing objects at a certain distance from it to a single point behind the lens. This is known as the focal or focus distance of the lens. Photographers typically put their main subject at this distance to create an effect where the subject is sharp but the background is blurry. In distributed ray tracing we can simulate this effect to make our renderings as if they are coming from a real camera.

This effect is best explained using a physical model where the image plane (e.g. the film or the sensor) is behind the lens. Normally, we prentend that the image plane is *in front of* the sensor to avoid the burden of flipping our images after being rendered. The setup is shown in the following figure.



In this setup, our goals are to compute **q**, **s**, and **d**. **q** represents one sample's position within a pixel. **s** is the position on the aperture through which the ray from the sample will pass. **d** is the direction of this ray *after* it bends through the lens. Computing **q** is easy – just sample a random point within the pixel. Computing **s** is also easy if we assume the aperture to be a square window instead of a circular one. This assumption will make a minor difference in the image being rendered. Computing **d** is a bit more involved. This is actually why we drew the straight line from point **q** through **o** and until **p**. Here, **o** represents the center of the lens (i.e. our camera origin). Lenses have a property that the direction of a ray that passes through its center remains unaltered. Given these, we first compute the **dir** vector:

$$\mathbf{dir} = normalize(\mathbf{o} - \mathbf{q}). \tag{2}$$

We then compute at what t value, the ray $r(t) = \mathbf{o} + t.\mathbf{dir}$ passes through the focal plane. This can be found by:

$$t_{\text{fd}} = \frac{focusDistance}{dot(\mathbf{dir}, -\mathbf{w})}. \tag{3}$$

Once t is found, we can compute point **p**:

$$\mathbf{p} = r(t_{\text{fd}}) \tag{4}$$

By using **p**, we can find the bent direction of the ray that originates at **q** and passes through the aperture at **s**:

$$\mathbf{d} = \mathbf{p} - \mathbf{s} \tag{5}$$

While doing this derivation, we assumed that the image plane is behind the lens. In practice we put the image plane in front of the lens. But you can verify yourself that this suprisingly does not change the math at all. To make this work in practice, you will have to compute two random points, one on the pixel and one on the lens. Then find d as explained above. Finally, create your primary ray as $r(t) = \mathbf{s} + t.\mathbf{d}$ and cast it into this wild world of ray tracing.

Area lights: For area lights, please see the other PDF document as well as the lecture slides.

References

[1] Robert L Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 137–145, 1984.