# Path Tracing with Monte Carlo Sampling

## Abstract

Path Tracing is a basic implementation of the render equation, which tracks all of the light paths to calculate the luminance of each pixel. And Monte Carlo sampling (MC) is a simple method of path tracing. With sampling, we can reduce the integration to a summation, which make a proper approximation to the actual result. Monte Carlo Path Tracing (MCPT) consists of 3 parts: Ray casting, Sampling and Integration. In this article, we will describe how MCPT works, analyzing the algorithm and giving a brief demonstration of this method.

## Introduction

The render equation tells how the luminance is transported on each surface. We collect the light from all the directions, multiplied by a weight depending on the input direction, the material itself, the output direction, etc. The final result is the integration of the input light within a small solid angle. However, it is hard for computer to do the integration, so MC provided an approximation to the integration. We use some random variables to sample the integrand. For each sampled integrand, we get one value depending on the MC integration rule, and finally, we sum them up and calculate the average to decide the final color of this pixel.

We sample on different surfaces, including diffuse surface, specular surface and Fresnel reflect and refract surface. On different surfaces, we use different distribution of samples to simplify the calculation of final summation and the average, which is called importance sampling in MC. The information of such surfaces is stored in Material files; meanwhile the vertices and the normals are stored in Object files. We read these files in run-time, and prepare them for the renderer to use.

Another important part is intersection detection. We use rays to do the path tracing, so we should know whether a ray is intersected with any object. If the ray and one object intersect, we must obtain the description of the hit-point (which triangle, which material, etc.). In this demonstration, we do not apply any accelerating methods since the number of triangles is not that high, and we run this demonstration on GPU using general purposed programming language CUDA.

We will show some results of this demonstration and make some explanation about these results.

## Rays

This MCPT is based on directional light beam called rays. In out model, each ray consists of two main parts: The Starting Point and The Direction . Using this definition, every single point that on the straight light path can be described as: , where is the parameter.

### 2.1 Ray Generation

Out tracing ray begins at the camera, and ends at the light source probably. So the first step is to generate the ray from the camera. Our initial data is each pixel’s coordinates on the screen, and the goal is restore the pixel’s coordinates in world space. This could be simply done using the projection matrix and the view matrix.

Consider we are using such projection matrix :

Where is the width of the screen, is the height of the screen, and are far plane and near plane and is the Field-of-View. This matrix projects one point in view space to the screen space (whole viewport), and the depth for each point will be mapped to , with deeper points have less depth value. Suppose the position of the pixel in view space is and the position on screen is with divided. We have:

But the problem is that we cannot derive from and . In fact, what we need is just a direction, and the position of camera in view space is exactly , so every point with matches our requirement. According to this, we can set to , and calculate and , and normalize the direction to make its Euclidean metric to .

The next part is to transform the direction from view space to world space. We have the view matrix , where is rotation and is translation. The equation is:

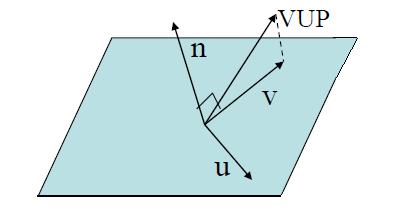
With

Vector , and are vectors of a UVN camera system (Figure 1). Since belongs to 3-Dementional Special Orthogonal Group (SO(3)), ’s inverse is equal to its transpose, and we don’t need to recalculate the inverse matrix again.

### 2.2 Ray Casting

With the generated ray, we can now cast the ray to objects. Our objects are formed with triangles, so the casted ray will intersect with triangles. We have already known that points on the ray can be represented as:

Figure 1 UVN camera system, with n(0,1,0) as up vector, v(0,0,-1) as look-at vector and u(1,0,0) as right vector. Note that we are using the right-hand coordinate system, so the z value of v is -1 instead of 1



And the next thing is to represent every point on the triangle. One thing is that all the points on the triangle is on the same plane, to make use of this, we let the 3 vertices be , and , vector and are linearly independent, hence each point on the plane can be written like: . To make the equation more common the point can also be like:

With (4) and (5) we can solve the formula and get the parameters.

In addition, we should also keep track of the material of the hit triangle for the sampling.

## Monte Carlo Integration

The basic idea of MC integration is to use samples to use the expectation to approximate the integration:

Integrand is any real function, is the distribution function of a random variable .

Using (6), we can rewrite the integration part of the render equation:

### 3.1 Diffuse Sample

Diffuse material reflect light to all directions equally, which means . To make things simple, we think there is no loss during the reflection. Hence if we sample as , then the result .

Furthermore, if we want to eliminate the cosine part, we can choose another and . Thus, we get our sampled direction . However, this direction is in the tangent space of the hit plane, and we should change it to world space. This could easily be done by creating orthogonal basics and rotate the direction like View matrix. Since we’ve apply the cosine part in the sample, the luminance of the light is unchanged.

### 3.2 Specular Sample

We use Blinn-Phong lighting model to sample specular surfaces. Again, to make the summation has only term, we apply the phong lobe to the sample, which means . Referring to the model, , where is the half vector of the light vector and the view vector. So, the sampled direction in this model is the half vector. Then the same thing is to rotate this direction from normal vector’s tangent space to world space. And the output direction is symmetric with the input direction by the half vector:

### 3.3 Fresnel

This kind of material is something different. We do not sample the output ray’s direction, instead, we sample only one variable to decide whether the ray is reflected or refracted.

According to Fresnel’s law, we can get the refract rate of the input ray [1]:

In this equation means the probability of the light to be refracted (not concerning total reflection), so if we sample a value , means the light is refracted and means the light is reflected. is the basic transparent value, and we treat it as the value of in the .mtl file. and are light direction and surface normal. This means the more bias the light against the normal, the more likelihood the light is to be reflected.

In fact, because of the total reflection, not all the light going through the refract pass will be actual refracted. When the total reflection happens, the light is still reflected on the surface.

## Environment and Results

Our demonstration is running on a desktop PC, with Intel i7 6700k, 16G RAM, and a Nvidia GTX1070 graphics card.

The software we use is Visual Studio 2017 15.6.2 with vc140 c++ compiling tool chain and 10.0.16299.0 windows SDK. CUDA version is 9.1 with all of the three patches.

For visualization, we use the OpenCV library to show the temporary result of the MC method and save the final image to a .png file.

We started 600 blocks for each row and 800 threads in each block for each column.

We first render the Cornell Box (Figure 2)

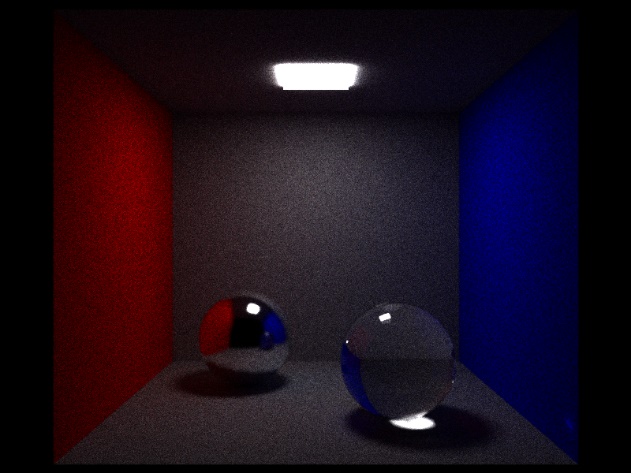
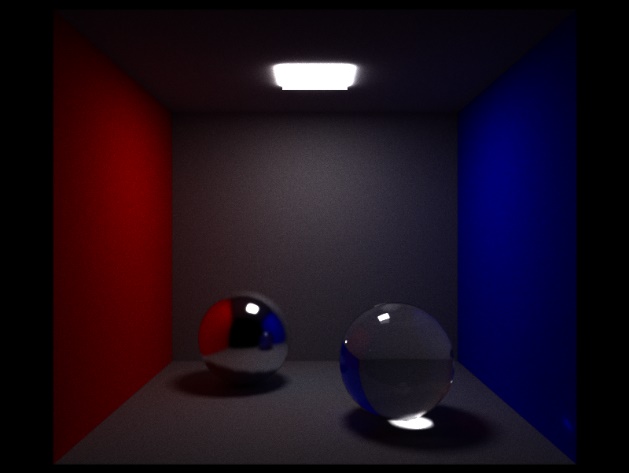


Figure 2 Rendered Cornell Box. Light source luminance is 30, and the number of samples is 1000 (left) and 10000 (right)

Since we are shading the image by choosing a random subset of pixels per frame, if the number of samples is not large enough there would be much noise such as the image in left of the Figure 2, however, as the sample count grows, the noise is filtered, although there are still non-smooth color changes in an area where the color is supposed to be the same.

The walls are all diffuse surfaces and the left sphere is a specular sphere with an exponent of 1000, the right sphere is with Fresnel material and the basic transparent rate is 0.9.

Figure 3 Scene 2 with luminance 10 and 10000 samples (Blinn-Phong model)

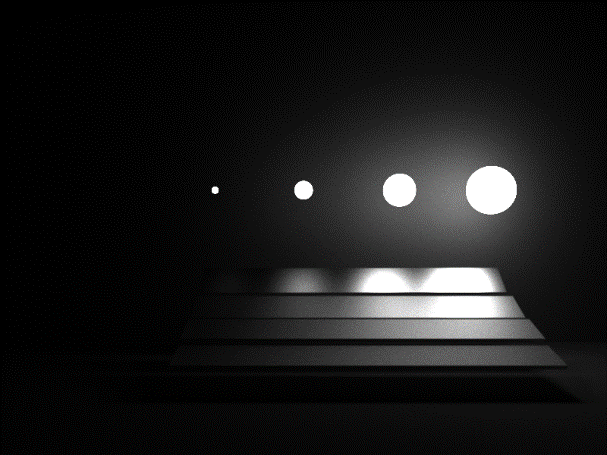
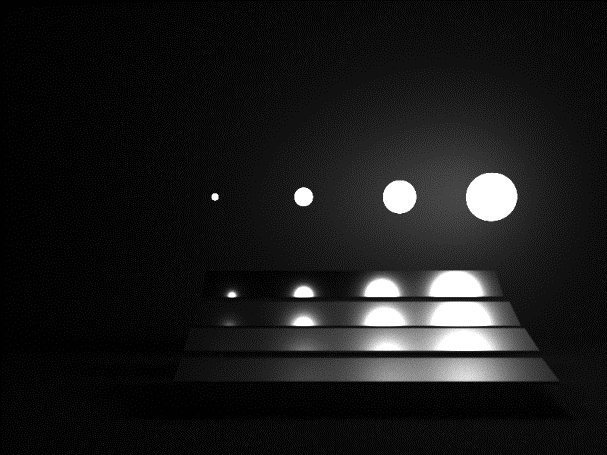


Figure 4 Scene 2 with luminance 10 and 10000 samples (Phong model)



Another result is about the specular material with different exponents, and light sources with different size.

Figure 3 shows the result of the scene. The four spheres are light sources with different size, and the four cubes are with specular materials. Upper cube has higher exponents and will result in light spots on the cube. The lower exponent, the larger the light spot, and less focused light.

There is a fact that the smallest light source hardly appears on the cube, which tells us that the sampled ray is too hard to hit the light source. But the result with Phong model is quite different (Figure 4).

The performance of the demonstration is listed as below:

|  |  |  |  |
| --- | --- | --- | --- |
| Scene | Triangles | Samples | Runtime |
| 1 | 863 | 1000 | 200sec |
| 1 | 863 | 10000 | 33min |
| 2 | 1733 | 100 | 34sec |
| 2 | 1733 | 10000 | 58min |

It’s clear that more triangles will cost more time on searching for intersection. Giving each object with a bounding box could probably reduce the searching time if there is no intersection. However, calculation on CUDA kernel with tens of thousands of threads is limited, and there is an issue that when applying bounding box, the CUDA kernel will not launch. So we decide not to add accelerating methods, and for such demonstration, this is tolerable.

## Conclusion

MC is a useful method of realistic graphics and gives a good result when applied to some optimizations. However, sampling causes noise and without post processes, the result look noisy. Nvidia recently provided a deep learning based method to reduce noise and with that method deployed, ray tracing should perform better.

## Reference

[1] 冯乐乐. Unity Shader 入门精要. 10.1.5 菲涅耳反射, Schlick近似