# 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 1.

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)

