Monte Carlo Path Tracer

Ziyi Lu

Course Project of *Advanced Computer Graphics——Realistic Image Synthesis Winter Semester 2022* [Course Page]

1. Abstract
2. Theory
3. Implementation
4. Results

1 Abstract

This project (**Tira**, short for Tiny ray tracer) is a tiny physically based renderer that features **Monte Carlo Path Tracer**. The core functionalities of this project include basic ray tracing utilities, acceleration structures, material apperances, integrators. Both CPU multi-thread and GPU (OpenGL compute shader) version are provided.

2 Theory

2.1 Monte Carlo Path Tracing

The process of rendering can be described as solving the following rendering equation:

$$L_o(p,\omega_o) = \int_{\mathcal{S}^2} f(p,\omega_o,\omega_i) L_i(p,\omega_i)|\cos\theta_i| d\omega_i$$

The equation can be approximated by Monte-Carlo integration:

$$L_o(p,\omega_o) \approx \frac{1}{N}\sum_{j=1}^{N} \frac{f(p,\omega_o,\omega_j) L_i(p,\omega_j)|\cos\theta_j|}{p(\omega_j)}$$

Where $p(\omega_j)$ is the Probability Distribution Function (PDF), which describes the distribution of irradiance.

Therefore, the process of Monte-Carlo integration can be described as the following pseudo code:

```
Li(p, wo):
    Sample a wi from a distribution
    Intersect the ray(p, wi) with the scene at q
    Return 0 if hit nothing
    If the ray hit a light:
        Return Li * brdf * cosine / pdf(wi)
```

```
Else:
    Return Li(q, -wi) * brdf * cosine / pdf(wi)
```

By evaluating Monte-Carlo estimator, we approach the integral (nonbias) by sampling a large number of paths for each pixel.

2.2 Multiple Importance Sampling

To reduce variance, it is a common strategy to take samples from several distributions since they are extremely hard to be described in just one distribution. Multiple importance sampling (MIS) is the certain strategy to construct a robust Monte-Carlo estimator with these samples from different distritbutions.

The core idea behind MIS is to weight the contributions by a heuristic based on PDF of the samples. The balanced heuristics is as follow:

$$\hat{w}_i(x) = \frac{c_i p_i(x)}{\sum_j c_j p_j(x)}$$

Where $\hat{w}_i$ is the weight for the sample drwan from the i-th distribution, c_i is the sample count of the i-th distribution and p_i is the PDF. Also there are other heuristics such as power heuristics and so on.

The MIS is used in Monte-Carlo path tracing for sampling area light and sampling the BRDF, in bidirectional path tracing for weighting each connected path.

2.3 Bidirectional Path Tracing

Bidirectional path tracing trace rays from both camera and light, generating a number of vertices (intersection) with certain probabilities. The bidirectional method then connect these vertices forming a number of paths that contribute to camera with certain PDF. By MIS, we are able to combine these paths together.

Bidirectional method has advantage in rendering effects like caustics. Also it greatly reduce the computation since the vertices are reused multiple times forming new paths rather than trace a new path.

3 Implementation

3.1 Math and Geometry

I implemented simple vectors and matrices in column major. I tried SIMD accleration, but have not seen significance in performance improvement. The SIMD option is turned off by default.

In geometry part, I implemented ray-triangle, ray-sphere, ray-AABB intersection.

3.2 Scene

To load the provided scene, I use thirdparty liberaries like `tinyobjloader`, `stb` and `pugixml`.

To correctly load the official scene, the `tinyobjloader` has to be modified in order to load `Tr` as transmittance (with 3 components) correctly.

Also, for convenience, I extended the xml file so as to load additional parameters as follows:

- Extra lightings info (directional light & envmap)
- Integrator settings (spp, max bounce and so on)
- Scene settings (acceleration structure type and so on)
- Extra primitives (sphere)
- Kernel settings (for GPU version)

3.3 Material

This project mainly uses two material models:

- Phong Model
- Glass Model

Phong Model can be described by the following equations:

$$f(\omega_i, \omega_o) = k_d \frac{1}{\pi} + k_s \frac{n+2}{2\pi} \cos^n \alpha$$

Where k_d is the diffuse reflectivity (`Kd` in .mtl file), k_s is the specular reflectivity (`Ks` in .mtl file), n is the specular exponent (`Ni` in .mtl file). α is the angle between the perfect reflection direction (`reflect(-wi, n)`) and the outgoing direction ω_o.

For Glass material, there is chance for both reflection and refraction determined by fresnel term. Fresnel term decides the ratio of reflection and refraction. The scene will automatically decide whether this material is glass or not.

3.4 Acceleration Structure

I implemented two acceleration structures (BVH and Octree) and use BVH by default.

The BVH is partitioned by SAH (surface area heuristics). I do SAH search for all three axes. To avoid taking too much time building BVH, it takes fixed steps on each axes so as to reduce the total search time from $O(N \log N)$ to $O(N)$.

The octree only out performs the BVH w/o SAH for certain scenes, therefore is turned of by default.

3.5 Integrator

The Integrator sturct is an abstraction of integrating the rendering equation to render a realistic image. To implement certain path tracing algorithm, override the `get_pixel_color` function and here is its declaration:

```
virtual float3 get_pixel_color(
    int x, int y, int sample_id, Scene const& scene) = 0;
```

`x` and `y` are pixel coordinates, `sample_id` is used for random generator seed, and `scene` provided everything needed for integrating the rendering equation.

There are three integrator implementation:

1. Whitted style path tracer (took reference from *pbrt*)
2. Monte Carlo path tracer (unidirectional)
3. Bidirectional path tracer (under development...)

3.6 GPU Acceleration

To accelerate the integration part, I implemented a GPU version with OpenGL compute shader. The scene and acceleration structure is constructed by CPU and passed to storage buffers. For performance, the GPU version renders in tiles with adaptive samples per kernel invocation.

Theoretically, OpenCL or CUDA are more suitable for such heavy load task and OpenGL is more likely to crash on such task. But I only choose OpenGL for convenience for this tiny renderer, and it does speed up the calculation significantly comparing to any CPU.

4 Result

All the images in this document is compressed, for the original images and other images, see the supplement material.

4.1 Default scenes

The following session I will demonstrate some images rendered by this renderer. Most of them are rendered with GPU version, which is identical to CPU version given the same size and samples.

For the default scenes provided in the course page, I doubled the image size to get better results. (Figure 1 ~ 3)

4.2 Other scenes

For my own scenes, I select some of the representative ones that each demonstrates some of the features of my renderer. (Figure 4 ~ 8)

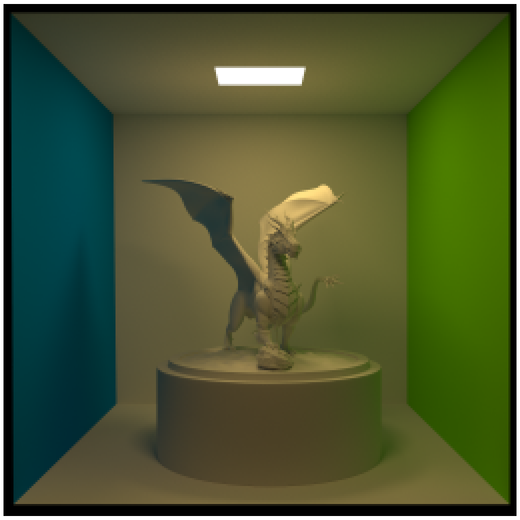

Figure 1: cornell-box, rendered in 2048x2048 with 4096 SPP, by RTX 2070s in 41 mins.

Figure 2: staircase, rendered in 2560x1440 with 512 SPP, by RTX 2070s in 24 mins.

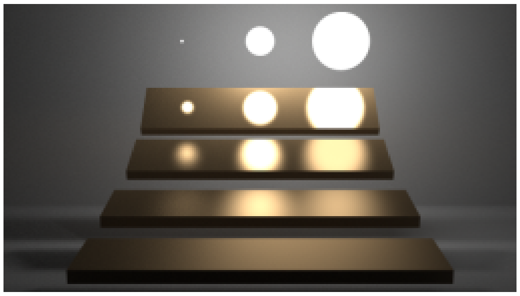

Figure 3: veach-mis, rendered in 2560x1440 with 8192 SPP, by RTX 2070s in 3 hours 20 mins.

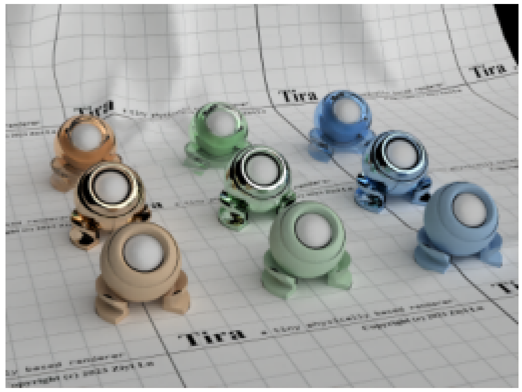

Figure 4: Test orbs by Yasutoshi Mori, a demonstration of pure diffuse, glossy metallic and colored glass materials respectively. The picture is rendered in 1600x1200 with 256 SPP in 5.6 hours by RTX 2070s.

Figure 5: Oak tree in cornell box, a demonstration of alpha testing in intersection. Rendered in 1024x1024 with 64 SPP, by RTX 2070s in 5 mins 24 secs.

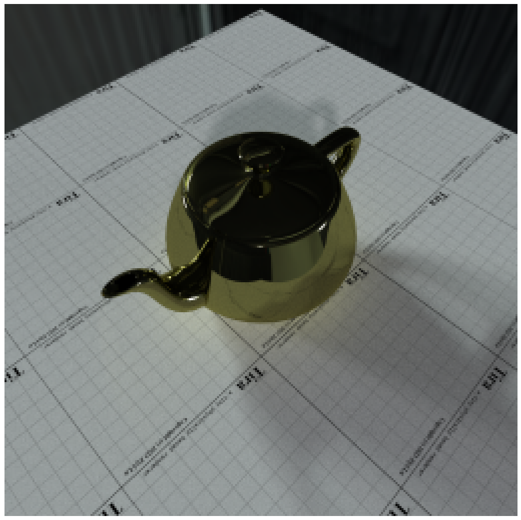

Figure 6: Utah teapot lit by envmap (blender `studio.exr`), rendered in 2048x2048 with 4096 SPP, by RTX 2070s in 48 mins.

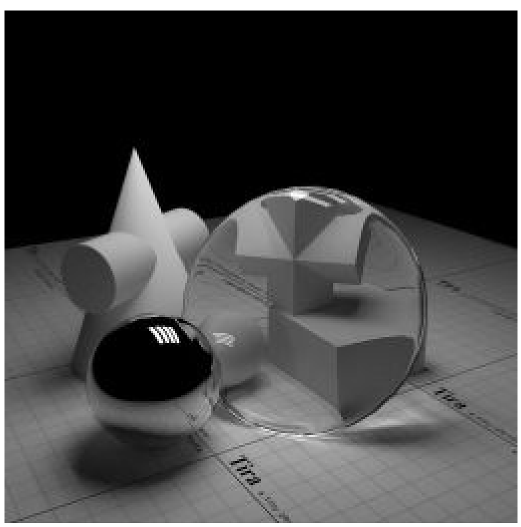

Figure 7: Geometry sets created by blender, rendered in 2048x2048 with 2048 SPP, by GTX 960 in 36 mins.

Figure 8: sponza lit by sunlight (directional light), rendered in 2400x1600 with 64 SPP, by RTX 2070s in 23 mins

Reference

[1] Peter Shirley et al., Ray Tracing in One Weekend Series, https://raytracing.github.io/

[2] Matt Pharr et al., Physically Based Rendering: From Theory To Implementation, https://www.pbr-book.org/

[3] Jason Lawrence, Importance Sampling of the Phong Reflectance Model, https://www.cs.princeton.edu/courses/archive/fall16/cos526/papers/importance.pdf

[5] Brian Karis, Specular BRDF Reference, http://graphicrants.blogspot.com/2013/08/specular-brdf-reference.html

[6] Walt Disney Animation Studios, BRDF Explorer, https://github.com/wdas/brdf

[7] Eric Veach, Robust Monte Carlo Methods for Light Transport Simulation, Ph.D. dissertation, Stanford University, December 1997, http://graphics.stanford.edu/papers/veach_thesis/