Monte Carlo Path Tracing with CUDA Acceleration

Shreya Bhaumik, Nikhil Johny, Yuka Murata, Nisha Mariam Thomas
University of Southern California

***Abstract*—Path tracing is a physically-based rending algorithm for three-dimensional scenes in computer graphics such that the global illumination generates a photorealistic image [1]. The algorithm integrates over all the illuminance arriving at a single point on the surface of an object. Accurate interaction of light with material surfaces can render high fidelity images. The goal of our project is to recreate the famous Cornell box scene by developing a path tracer using Monte Carlo integration and CUDA for GPU acceleration.**

***Keywords*—CUDA, Global Illumination, Importance Sampling, Monte Carlo, Path Tracing.**

I. INTRODUCTION

Rasterization is a process of rendering each object based on its position in the scene. It converts the objects of the 3D model into pixels on a 2D screen. It builds an image just like a painter paints a painting, object by object.

Ray tracing is another rendering method that shoots rays from the camera to the scene. In a basic ray tracer, the rays reflect from the objects to the light source to compute the color value at the pixel position. Ray tracing can model the actual behavior of light in a scene, but it is much more computationally expensive than rasterization.

A path tracer tries to solve the rendering equation proposed by Kajiya [5]. It uses the same technique of ray tracing but unlike it, the albedo is calculated by the lights arriving from all possible directions each of which continuously bounced off from other objects in the scene. This simulates a more natural behavior of light leading to effects such as soft shadows, caustics, ambient occlusion, and indirect lighting. Due to its accuracy and unbiased nature, path tracing is used to generate reference images when testing the quality of other rendering algorithms.

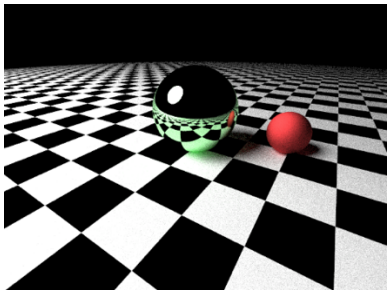

Figure 1.1. Photorealistic path tracer output (rendered using our path tracer)

.

II. TECHNOLOGY STACK

2.1 Programming Language

The code of the entire project has been built from scratch using C++.

2.2 Libraries Used

i. Stb_image library – to read image for image texture mapping.

ii. CUDA – for GPU acceleration.

III. IMPLEMENTATION

3.1 Construction of the basic path tracer

We defined a class to represent 3D geometric vectors, which includes vector operations. A ray is implemented as $p(t) = A + t * B$ where p is a 3D position on a 3D line whose origin is A and direction is B [2] .

Firstly, we place the camera at (0, 0, 0) and define the coordinates of our screen space. For a given sphere with center C = (C_x, C_y, C_z) and radius R, if our ray intersects the sphere, there will be some t for which p(t) will satisfy the equation:

$$t^2 \cdot dot(B,B) + 2t \cdot dot(B, A-C) + dot(A-C, A-C) - R^2 = 0. \quad (1)$$

We can use this to find t and color the pixel that hits the sphere with a designated color [2].

Next, we define surface normals so that we can shade the sphere. We take normal to be a unit vector and map its x, y, z to r, g, b in order to visualize the normal as we still don't have a light source defined.

We apply antialiasing to remove the jagged edges with super-sampling. For every pixel we compute colors by averaging the colors of rays at a pixel passing through several randomly generated samples.

3.2 Introducing Materials – Lambertian, Metal and Dielectric

Lambertian/Diffuse objects which don't emit light take the color of the surroundings and adjust it with their own color[2]. Light reflects from a diffuse surface in random directions. We define a sphere with unit radius at the hit point and pick any random point in this sphere to get the direction of the reflected ray.

Metals are smooth, and they don't scatter rays randomly. The reflected ray for metals, have the direction of R which can be calculated as:

$$V - 2 * dot(V, N) * N \qquad (2)$$

where V is the incoming light direction and N is the normal at the intersection [2].

Dielectrics are clear materials like glass, water etc. When a light ray hits them, it divides into a reflected and a refracted ray [2]. On interaction with these materials we randomly choose either a reflected or refracted ray. In case it is the reflected ray, we use (2), while for the refracted ray, we apply the Snell's law: $n.sin(\theta) = n'.sin(\theta')$, where n and n' are refractive indices of the respective materials.

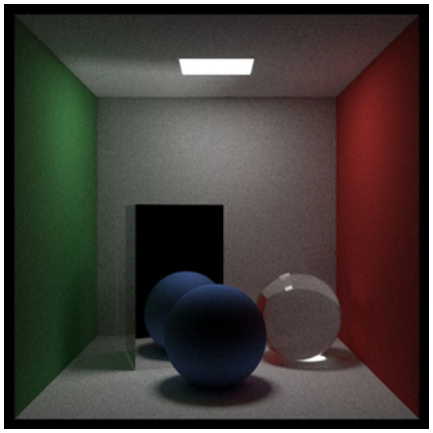

Figure 3.2. Objects with different Materials

3.3 Camera Properties

We place our camera at a position, *look-from* and point it to another position, *look-at* [2]. To handle tilt of the camera, we specify an *up-vector*. The orthonormal basis to denote camera's coordinates, (u, v, w) can be calculated using cross-products on the vectors defined.

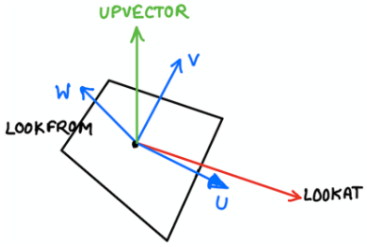

Figure 3.3. Camera Coordinates

3.4 Camera Effects – Defocus Blur and Motion Blur

Defocus blur causes only the object in focus to be in focus and the rest to be blurred [2]. We start our rays from the lens' surface and project them onto the virtual film plane which is at a distance such that the film is perfectly focused.

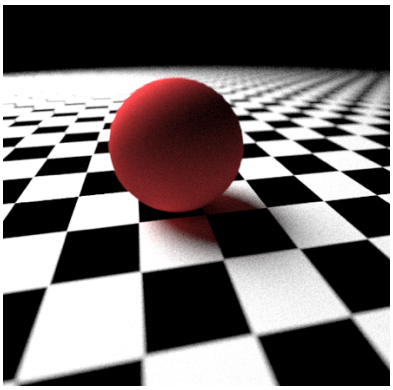

Figure 3.4.1. Defocus Blur

Motion blur occurs when the shutter stays open for a time interval and the objects and/or camera moves [3]. To implement this, we make our virtual camera produce rays at random times between a time interval: t1 and t2. Our sphere object is made to move vertically from initial center position at t1 to final center position at t2.

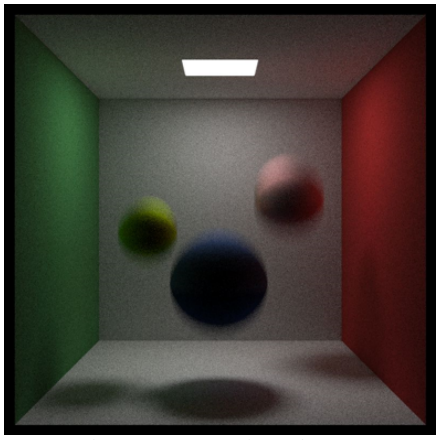

Figure 3.4.2. Motion Blur

3.5 Bounding Volume Hierarchies

In a path tracer, each intersection test for a ray with objects is a performance bottleneck. We build a hierarchy of objects based on its bounding volumes. This is implemented using axis-aligned bounding boxes (AABBs). We perform intersection tests against the nodes from top to bottom. If a ray misses the bounding volume at a level, then it misses all the node's children [3].

3.6 Textures

Checker texture can be created by multiplying sine functions for each of all the three dimensions and deciding the color based

on the sign of the result.

Perlin noise creates a reproducible noise pattern given a 3D point, it returns the same random number [3]. So, we use hashing to randomize and then interpolation to smoothen it.

To produce turbulence, we get a summation of multiple calls of Perlin noise [3].

Image texture mapping is done by reading an image and finding a texel position that corresponds to the (u, v) position of the object.

Figure 3.6. Objects with different textures

3.7. Cornell Box

To make the Cornell Box, we define 5 walls and rectangular light using the rectangle object. Next we set the camera to appropriate position. But three walls of the box are not visible as their normal face the wrong direction. So, we flip the normals to get the correct shade. We then add 2 boxes which are basically axis-aligned blocks made out of 6 rectangle objects and then we translate and rotate the boxes so as to match the classic Cornell Box scene.

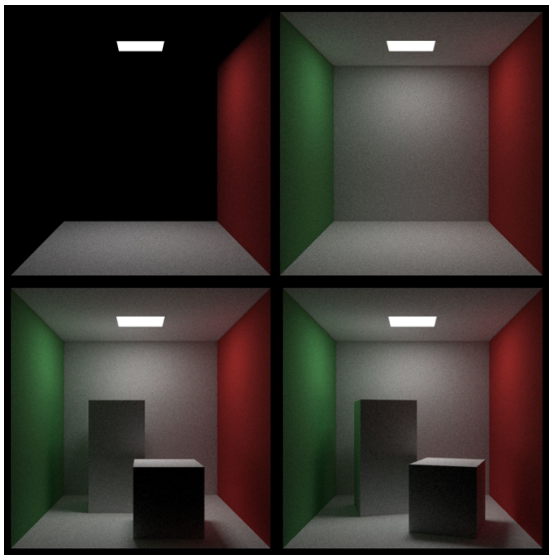

Figure 3.7 Formation of Cornell Box

3.8 Volumes

Instead of reflecting the ray from the surface of an object, we can use the boundaries and find a random reflection point within the boundaries. A ray can entirely pass through the volume of the object or it may scatter in any direction at some point. The denser the volume, the more likely it is for the ray to scatter [3]. This creates the effect of fog or mist.

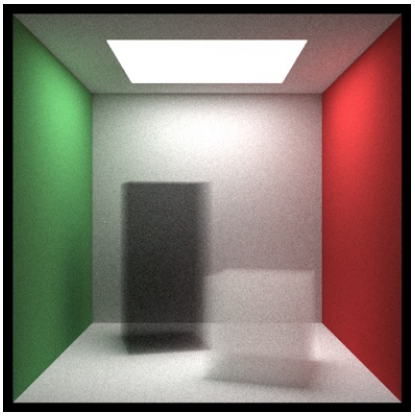

Figure 3.8. Volumes

3.9 Monte Carlo and Importance Sampling

We can achieve the statistical estimation of Kajiya's rendering equation [5] by using the Monte Carlo integration. We compute four probability density functions (pdf) for a ray scattered on a diffuse surface, a ray directly going to an area light, a ray going to a sphere, and a ray reflecting on a metal surface. Each pdf is then used to compute attenuation with the corresponding reflected ray. We further implemented importance sampling by combining sampling methods along weighted average of the related pdfs to reduce noise in the rendered image.

The generalized form of the Monte Carlo integration is shown by Jensen [8]:

$$\int_{x \in S} g(x) d\mu \approx \frac{1}{N} \sum_{i=1}^{N} \frac{g(x_i)}{p(x_i)}. \quad (3)$$

We use this estimation to calculate the color value of each pixel. Each randomly reflected ray is weighted with its pdf [4]. For the rays scattered on a diffuse surface, the pdf is a *cos(θ)* where the angle is between the normal and the reflected ray on the hemisphere:

$$pdf_{diffuse}(direction) = \frac{\cos\theta}{\pi}. \quad (4)$$

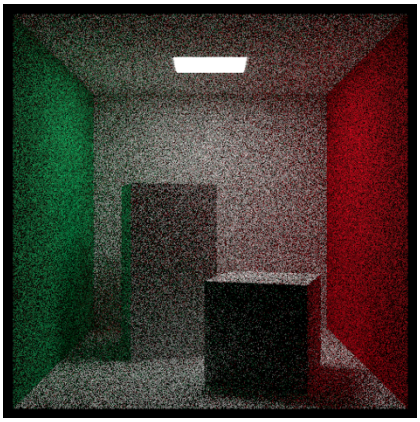

Figure 3.9.1. Cosine Sampling

For the direct light sampling, we map the pdf over a hemisphere knowing the area of the light, angle, and the distance between the intersection and light source:

$$pdf_{areadlight}(direction) = \frac{distance\ (p,q)^2}{\cos\alpha\ \cdot A}. \quad (5)$$

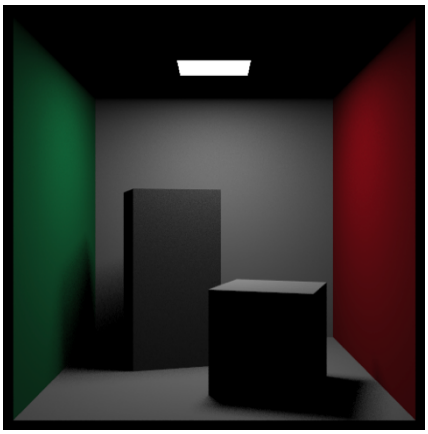

Figure 3.9.2. Direct Light Sampling

The pdf for the sampling through a glass sphere is simply a solid angle which is equal to an area of a unit sphere as follows:

$$pdf_{sphere}(direction) = \frac{1}{solid\_angle}. \quad (6)$$

$$solid\_angle = \int_0^{2\pi}\int_0^{\theta_{max}} \sin\theta\, d\theta d\phi = 2\pi\ \cdot (1 - \cos\theta_{max}). \quad (7)$$

We can use an implicit sampling or a uniform pdf for the metal surface:

$$pdf_{metalbox}(direction) = 1. \quad (8)$$

Finally, we mix the probability densities of the cosine sampling (4) and direct light sampling (5) for the importance sampling as seen in Figure 3.9.4 to reduce noise in the image by taking a weighted average of these pdfs:

$$pdf_{mix}(direction) = \frac{1}{2}\, pdf_{diffuse}(direction) + \frac{1}{2}\, pdf_{arealight}(direction). \quad (9)$$

Our importance sampling shows significant improvement on noise reduction seen below.

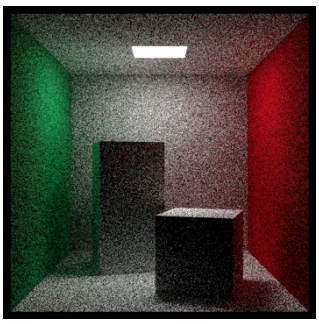

Figure 3.9.3. Random Sampling (100 samples/pixel)

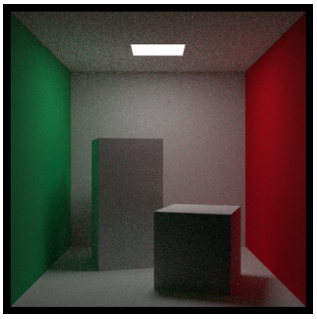

Figure 3.9.4. Importance Sampling (100 samples/pixel)

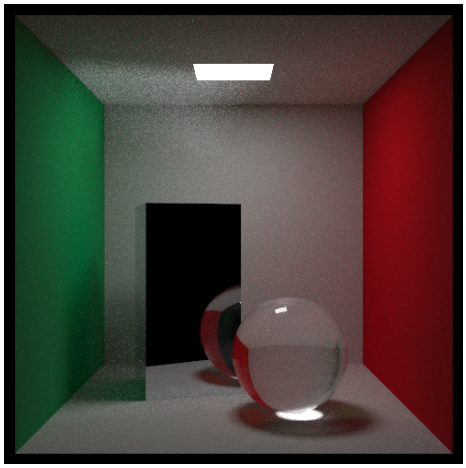

Figure 3.9.5. Glass and metal using Monte Carlo (1000 samples/pixel)

3.10 GPU Acceleration using CUDA

Prerequisite: Setup NVIDIA CUDA on device [6]. (We used CUDA 10.0.).

In CUDA, the scheduler takes blocks of threads and schedules them on the GPU. The frame buffer is defined on the host, allowing communication between the CPU and the GPU.

'cudaMallocManaged' allocates unified memory, which allows us to rely on the CUDA runtime to move the frame buffer on demand to the GPU for rendering and to the CPU for outputting the PPM image. 'cudaDeviceSynchronize' lets the CPU know that the GPU is done rendering [7].

After multiple trial and error, we came to a conclusion of using 8x8 thread blocks to develop an 800x800 image.

Image	Time taken by original code	Time taken by code with GPU acceleration using CUDA	% increase	Speed Increase
1. 500 random spheres with 100 samples (1200x600)	50 mins	3.9 mins	92.2 %	12.8x
2. Cornell box with 100 samples (800x800)	80 mins	5 mins	93.75 %	16x
3. Cornell box with 500 samples (800x800)	7 hrs	27 mins	93.48 %	15.35x

The average improvement is 93.14 % decrease in time taken, i.e. 15x speed.

IV. CONCLUSION

We have successfully recreated the famous Cornell box image shown in Figure 4.1.2. Our render of the Cornell box is shown in Figure 4.1.1. Notably, we have added further scene properties, namely, different materials like Lambertian, metal, dielectric and isometric and camera effects like motion blur and defocus blur. Moreover, we have improved on the basic path tracer and integrated Monte Carlo and added sampling techniques like random and importance sampling. Furthermore, we enhanced the performance of the path tracer by implementing CUDA and bounding box hierarchy.

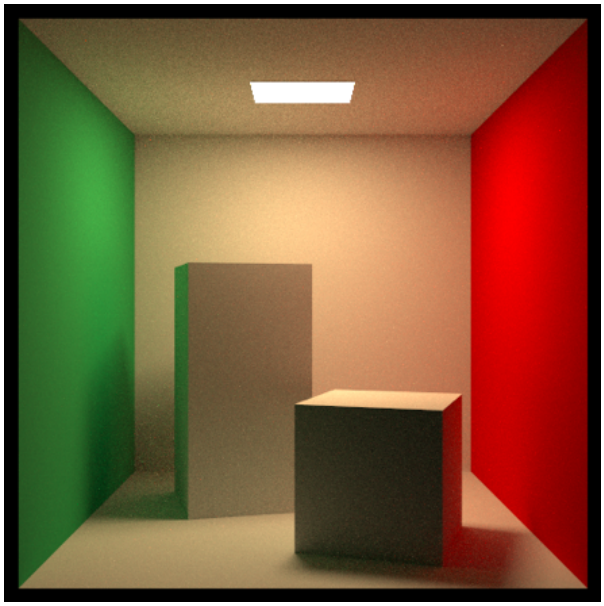

Figure 4.1.1. Our Monte Carlo path tracer (1000 samples/pixel)

Figure 4.1.2. Cornell box ground truth

REFERENCES

[1] "Path tracing." [online]. Available: https://en.wikipedia.org/wiki/Path_tracing

[2] P. Shirley, "Ray tracing in one weekend." [online]. Available: https://raytracing.github.io/books/RayTracingInOneWeekend.html

[3] P. Shirley, "Ray tracing the next week." [online]. Available: https://raytracing.github.io/books/RayTracingTheNextWeek.html

[4] P. Shirley, "Ray tracing the rest of your life." [online]. Available: ttps://raytracing.github.io/books/RayTracingTheRestOfYourLife.html

[5] J. T. Kajiya, "The rendering equation," in Computer Graphics, vol. 20, no. 4, Aug. 1986, pp. 143-150.

[6] "Gpu accelerated computing with c and c++." [online]. Available: https://developer.nvidia.com/how-to-cuda-c-cpp

[7] "Accelerated ray tracing in one weekend in cuda." [online]. Available: https://devblogs.nvidia.com/accelerated-ray-tracing-cuda/

[8] H. W. Jensen et al., "State of the art in monte carlo ray tracing for realistic image synthesis," in SIGGRAPH 2001 Course 29, Aug. 2001. [online]. Available: http://cseweb.ucsd.edu/~viscomp/classes/cse274/fa18/readings/course29sig01.pdf