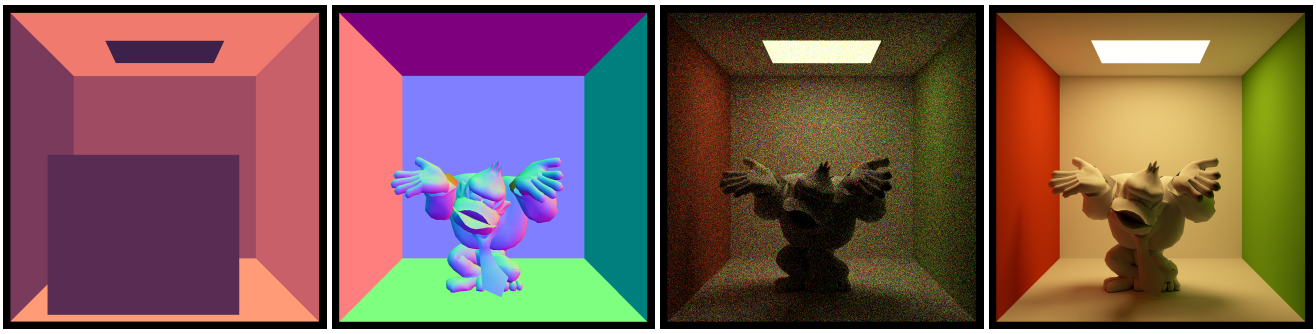
CRT: CUDA Ray Tracer

Raj Sugavanam
Washington University in St. Louis

Junseo Shin
Washington University in St. Louis



(a) Axis aligned bounding box. (b) Normal map (smooth shading). (c) 16 samples/pixel (d) 16384 samples/pixel

Figure 1: Progressive steps for computing a physically based render of a Cornell scene at 1440x1440 resolution. (c) and (d) uses the spectral data from the original Cornell box test scene with a 4745 triangle Donkey Kong model [3].

Abstract

We present a spectral path tracer that showcases the capabilities and limits of the CUDA programming model in the field of physically based rendering. The simplest ray tracing algorithm requires the GPU to dispatch rays from the camera into the scene, and calculate the interaction of the rays with the scene geometry in order to render an output image of the camera's view. Our implementation focuses on rendering a Cornell box test scene to demonstrate the accuracy and performance of the CUDA ray tracer (CRT). Representing the scene geometry with triangle meshes presents the ability to interopt professional 3D modeling software with the ray tracer, but also explores the limitations of our implementation.

1 Introduction

The computational complexity of simulating physical phenomena compels high performance computing and parallel processing. Many sub-categories of ray tracing suggest widely different approaches, and focusing on different areas of tradeoff between timing and simulation accuracy leaves room for many optimizations and algorithms.

2 Problem Statement

We address the computational problem of non-real-time, physically-based path tracing that is facilitated through parallelism. We specifically make the tradeoff to forgo simulation time in favor of more accurate light bounces. We only considered meshes composed of triangular faces, as every polygonal mesh can be decomposed to fit this. Furthermore, it greatly simplifies how we represent models in the code, which permits further optimizations.

3 Approach

Model setup. We begin by setting up a simple perspective camera and read a sequence of object files. We include a scene manager to handle multiple objects. We consider distances of triangles to the camera during rendering to ensure the output looks correct.

Path Tracing Strategy. We follow a widely used method in this respect by tracing light backwards, ie from the camera to the light source. As fully accurate ray tracers are almost impossible to feasibly run, our approach made heavy use of estimation techniques.

Color. During our implementation we found an opportunity to use wavelength-based colors rather than only RGB. This implies the usage of conversions from wavelength to RGB, as monitors can only feasibly display RGB. To this effect, we experimented with different color representations.

4 Implementation

.obj Inputs. We only use the host to read in .obj files, as doing so requires use of system calls, which cannot be done on the device for obvious reasons.

Thread Mapping. Each thread handles a single output pixel of the result image.

Camera Setup. Our implementation of perspective projection uses a point-based camera origin with a given focal length and FOV. This gives rise to a viewing plane in the scene, which can also be translated to a pixel grid (i.e., image output) through simple use of basis vectors to facilitate the bijection/translation between scene and image.

Object Algorithms. We use the Möller-Trumbore ray-triangle intersection to detect which rays hit models, and where. Each thread performs multiple calls to this algorithm, to serve the purpose of handling (multiple) light path bounces. This is inefficient from under-utilization of the device with smaller images, however a simple optimization of distributing several sampled paths for a given pixel across multiple threads can easily alleviate this problem. We solved z-fighting by only rendering the closest triangle for a given ray.

Light Bounces. A fully realistic model of ray tracing would consider infinite amounts of light rays that bounce from an incident ray, each of which differ by a differential element. This is not feasible on a computer; it is more reasonable to simulate this by bouncing a certain k number of rays for every incident ray, however for a maximum of n bounces, assuming a triangle mesh of $O(m)$ size and intersection performance of $O(m)$, we get a worst-case of $O(k^n m)$ computation time for a single ray for a single pixel on the camera. This is also infeasible despite the parallelization.

We opt to eliminate the exponential nature of this procedure by considering "paths" of light. In other words, we set $k = 1$, so our computation time remains linear in the amount of mesh triangles, which is a reasonable amount of work per thread. However, to give us more control of simulation accuracy, we define a P set of simulated light paths, per pixel. Each such $p_i \in P$ is create by using randomized bounce angles at every triangle intersection.

Suppose P is a set of light paths generated using a specific pixel. If $p_i \in P$ is a sequence $[r_1, r_2, \ldots, r_N]$ of rays, r_1 in p_i is equal to r_1 in p_j (ray incident from a pixel), however it is only infinitesimally likely that $r_k \in p_i$ is equal to $r_k \in p_j$.

P, in effect, has non-exponential granularity for how accurate we wish to make our ray tracer. Suppose $\|P\| = k$; we still get k bounces for the first triangle intersection from the camera's incident ray, but only $O(kNm)$ polynomial runtime to compute all intersections.

5 Performance Measurements

Table 1: Reduction performance measurements for arrays of arbitrary length N on the CPU and GPU.

N	CPU Min	CPU Max	GPU Min	GPU Max
5,000	34,806 ns	59,253 ns	96,714 ns	19,877 ns
10^4	0.693 ms	0.770 ms	0.087 ms	0.014 ms
10^6	7.368 ms	7.039 ms	0.156 ms	0.064 ms
10^8	714.632 ms	711.017 ms	4.705 ms	4.617 ms

In the reduction performance measurements from Table 1, the GPU incurrs a slight overhead for small arrays for reduction for arrays of size $N = 5,000$ and below, but the GPU is significantly faster for massive arrays such as a 151x speedup for $N = 10^8$ elements.

Table 2: AABB calculation performance for triangle meshes with M triangles on the CPU, GPU and GPU with CUDA streams. Using `sphere.obj` ($M = 960$), `donkey_kong.obj` ($M = 4,745$) and `dragon.obj` ($M = 100,000$).

M	CPU	GPU	CUDA streams
960	0.089 ms	0.177 ms	0.262 ms
4,745	0.398 ms	0.294 ms	0.279 ms
100,000	8.187 ms	1.737 ms	0.284 ms

Table 2 shows the performance of the reduction algorithm being inefficient for very small triangle mesh objects with less than 1,000 triangles, but the GPU starts to outperform the CPU for any triangle mesh larger than 5,000 triangles. Furthermore using CUDA streams further improves the performance of calculating the AABB for increasingly larger triangle meshes.

Table 3: Triangle intersection test performance with and without AABB culling for the `donkey_kong.obj` model in the Cornell Box scene.

Method	Rendering Time
No AABB	93.963 ms
With AABB	22.664 ms

For the Donkey Kong Cornell Box scene, AABB culling adds a 4.1x speedup compared to the naive triangle intersection test (Table 3). Although this greatly speed up the rendering time for raycasting the rays onto the scene, the performance improvement of AABB culling is dependent on the how much of the scene is taken up by the large triangle mesh object. For example, if the bounding box of the triangle mesh takes up the entire screen, then the AABB culling will not be able to filter out any rays from performing the computationally expensive triangle intersection tests for all the triangles in the mesh. However, if the triangle mesh is small and far away from the camera, then the AABB culling will be able to filter out most of the rays from performing the triangle intersection test on the triangle mesh.

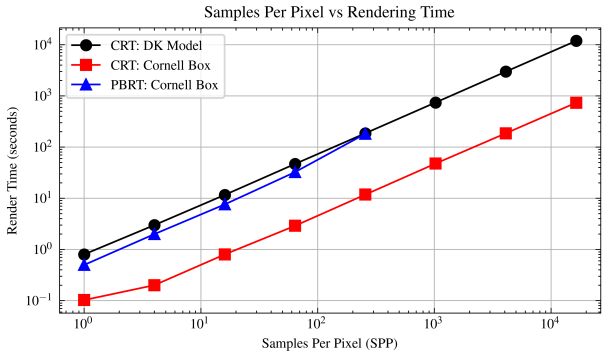


Figure 2: Log-log plot of SPP vs rendering times for the CRT and PBRT CPU renderer. All renders have 5 max bounces and output a 1440x1440 image. The PBRT renderer uses the CPU `SimplePathIntegrator` and `IndependentSampler`. the `donkey_kong.obj` model. The PBRT CPU renderer is run on a Mac Air M3 with multi-threading enabled for 8 threads.

Our CRT renderer has a 10x average speedup over the PBRT [7] CPU renderer for the Cornell Box test scene as shown in Figure 2. In a further test, doubling the dimensions of the output image to 2880x2880 resolution (or quadrupling the number of pixels in the image) increased the PBRT CPU rendering time from 2.0s 14.2s compared to the CRT renderer which only increased from 0.2s to 0.8s for a render using 4 samples per pixel. Although CRT renderer has a well define linear scaling with the number of pixels in the image, the PBRT CPU renderer suffers greatly from larger resolution renders despite its native multi-threading support.

6 Optimizations

One primary optimization we used was AABBs (axis-aligned bounding boxes) to compute a top-level intersection box that rays which have a chance to strike a triangle are guaranteed to hit. The intersection of this box is low cost and far easier to initially compute than iterate through every mesh triangle, therefore we use it as an initial filter to determine which pixels should proceed to multi-bounce path trace. This improves our compute time for smaller/farther distance meshes. The effect of this optimization reduces as more of the mesh fills the viewing range of the camera.

We also used several optimizations exclusive to the CUDA programming model. Due to the complex nature of simulating physical light, several parts of our code were written with object-oriented programming in mind. This requires us to enable separable compilation and linking device code to the host code. One caveat of this is that having separately compiled device code requires device link time optimization to be enabled via the `CMAKE_INTERPROCEDURAL_OPTIMIZATION` option in CMake. This allows the compiler to optimize the device code at link time rather than at compile time, thus allowing function inlining and other optimizations to be performed across translation units.

For the AABB calculation, we first used a Structure of Arrays (SoA) memory layout to allow for memory coalescing we perform reduction on shared memory to calculate the minimum and maximum bounds of a triangle mesh. Furthermore, due to the dynamic nature of the AABB calculation, our reduction implementation allows for an arbitrary length array of triangles to be passed in, and completes the final reduction between the blocks of a single kernel using custom `atomicMin()` and `atomicMax()` implemented based on the CUDA `atomicCAS()` function. Finally, CUDA streams were used to allow for multiple reduction to be performed in parallel on the GPU since the AABB calculation requires 9 reductions (each triangle has three vertices each with an x, y, z component stored as a `float*`) to be performed on a triangle mesh object. Although the SoA layout greatly improves AABB calculation performance, the memory accesses during the traversal of a rays through a scene is not coalesced very well since the path of each ray diverges greatly, so moving from using a Array of Structures (AoS) to a SoA layout only improved the ray tracing performance by about 1.12x.

Since ray tracing is a highly memory bound operation, we used constant memory to store the material properties of each triangle mesh object. This included the reflectance spectrum of each triangle mesh object to be accessable via `reflectance_id` and the emission spectrum of the light object. In addition, we also used constant memory to store the discrete values of the color matching functions (CMFs) to compute the CIE 1931 XYZ color space values and its corresponding XYZ to RGB transformation matrix. This greatly reduces the number of global memory accesses to the GPU already bounded by the triangle intersection test.

7 Results

We direct the reader to Figure (1) for the output `.ppm` images for our program. Figures (1a) and (1b) indicate our representations for AABBs and smooth shading. Smooth shading is a technique that calculates interpolated normals at any point on a triangle, using vertex normals. That is, if we represent point P in space with barycentric coordinates of triangle ABC by

$$P = \lambda_1 A + \lambda_2 B + \lambda_3 C,$$

then we calculate the surface normal *at point* $(\lambda_1, \lambda_2, \lambda_3)$ to be

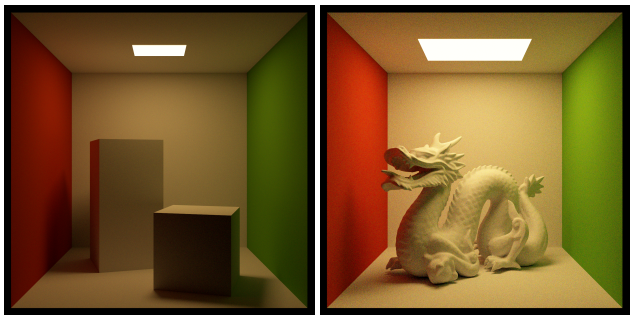
$$N = \lambda_1 N_i + \lambda_2 N_j + \lambda_3 N_k,$$

where N_i, N_j, and N_k are vertex normals of A, B, and C respectively.

If we had used a face normal based method, where each triangle gets assigned *one* N regardless of any barycentric point $(\lambda_1, \lambda_2, \lambda_3)$, we would see 'sharp' shading with more pronounced edges.

As RGB color can be encoded as a 3-dimensional vector, we see that the surface color of Figure (1b) represents the direction of our calculated N.

Soft shadows, seen in Figures (1c) and (1d), are a result of our ray tracing model. Figure (1c) has a far greater amount of noise due to an extremely low sample count, which cannot rely on the law of large numbers to accurately color an output pixel with volume of light paths.



(a) Rendering time: 700s **(b) Rendering time: 15,000s**

Figure 3: (a) Cornell box with 16,384 samples per pixel (b) 100,000 triangle `dragon.obj` [8] with 1024 samples per pixel

8 Discussion

Although the CRT renderer is capable of rendering a Cornell box scene with a triangle mesh object (Figure 3a), the perfomance greatly suffers for increasingly complex triangle meshes. For simple geometry, the CRT can render the original Cornell Box test scence with 16,384 samples per pixel in 700s compared to 12,000s for the Donkey Kong Cornell box scene. For even more complex geometry used in professional 3D modeling software, our implementation does not scale to the same computational speed as the PBRT GPU renderer (Figure 3b). This is due to our CRT computing only indirect lighting and not using any further acceleration structures to speed up the ray-triangle intersection test. Despite the limitations of our implementation, the CRT renderer is capable of taking `.obj` files from professional 3D modeling software such as Blender and render physically based images using spectral data which is not readily available as a core feature in most ray tracing software (PBRT natively supports spectral rendering).

9 Conclusion

One approach we considered was to use the `curand` interface to generate random numbers. We had a partial implementation of

this, but ended up using our own XORShift pseudo-random number generator. We still sample with a cosine-weighted distribution. We could have seen further optimization from usage of RT cores via NVIDIA Optix (speed-up with ray-triangle intersection, etc). We discussed implementing more material properties for objects; specifically, reflective, specular, and caustic. With enough time, we may have developed a protocol to utilize corresponding `.mtl` files, which we currently ignore.

Overall, our model is successful in feasibly simulating the basic behavior of light. Further developments would focus on enhancing the visual complexity of our renders.

References

[1] International Electrotechnical Commission. 2003. IEC 61966-2-1:1999 Amendment 1:2003 – Multimedia systems and equipment – Colour measurement and management – Part 2-1: Colour management – Default RGB colour space – sRGB. https://webstore.iec.ch/publication/6173.

[2] Joey de Vries. 2020. Learn opengl: Learn modern opengl graphics programming in a step-by-step fashion. https://learnopengl.com/.

[3] Nintendo. 2001. Super Smash Bros. Melee. https://www.models-resource.com/gamecube/ssbm/model/38652/.

[4] Steve Hollasch Peter Shirley, Trevor David Black. 2025. Ray Tracing in One Weekend. https://raytracing.github.io/books/RayTracingInOneWeekend.html

[5] Steve Hollasch Peter Shirley, Trevor David Black. 2025. Ray Tracing: The Next Week. https://raytracing.github.io/books/RayTracingTheNextWeek.html

[6] Steve Hollasch Peter Shirley, Trevor David Black. 2025. Ray Tracing: The Rest of Your Life. https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html

[7] Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2023. Physically Based Rendering: From Theory to Implementation, 4th ed. https://pbr-book.org/4ed/.

[8] Stanford University. 2009. Stanford 3D Scanning Repository. http://graphics.stanford.edu/data/3Dscanrep/.

[9] A. Wilkie, S. Nawaz, M. Droske, A. Weidlich, and J. Hanika. 2014. Hero Wavelength Spectral Sampling. *Computer Graphics Forum* 33, 4 (2014), 123–131. doi:10.1111/cgf.12419 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12419