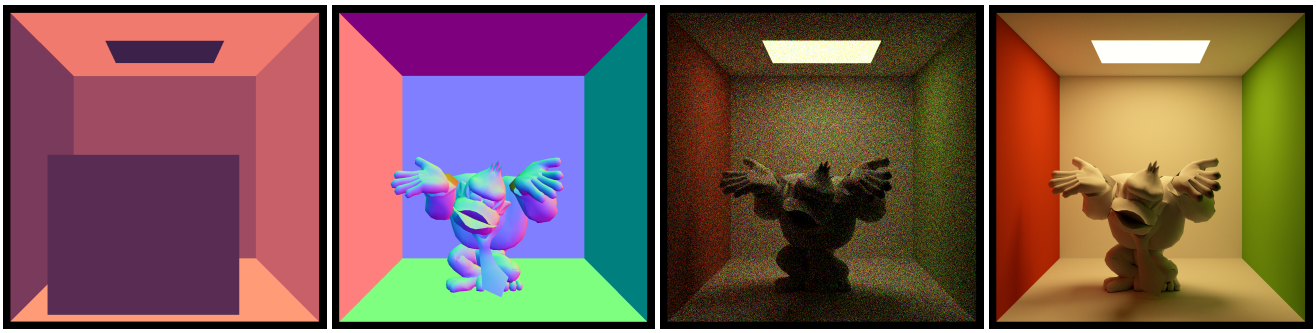
CRT: CUDA Ray Tracer

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 [cite model ref].

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 allows the GPU to perform multiple reductions on the triangle mesh object which uses Structure of Arrays (SoA) to store 9 separate arrays accounting for each component of a triangle's vertices.

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.

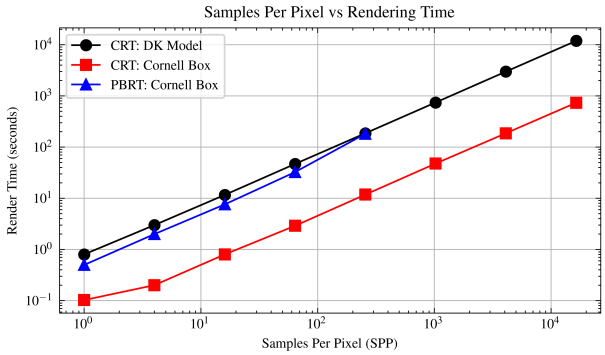


Figure 2: Log-log plot of SPP vs rendering times for the CRT and PBRT CPU renderer. All renders have 5 max bounces and output a 1440x1440 image. The PBRT renderer uses the CPU `SimplePathIntegrator` and `IndependentSampler`. the `donkey_kong.obj` model. The PBRT CPU renderer is run on a Mac Air M3 with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

Our CRT renderer has a 10x average speedup over the PBRT 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 restructured the arrays of vectors for a mesh to prioritize components first, rather than vectors first. Our array structure will use a major order with all vertices' x-components first, then y, then z. This makes coalescing memory accesses between threads far easier.

7 Section

Paragraph here

8 Section

Paragraph here

9 Section

Paragraph here

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