# 15618 Project Milestone Report: CUDA Ray Tracer

**Jingguo Liang** and **Ruben Partono**

Project Webpage: https://cuda-ray-tracer.netlify.app/

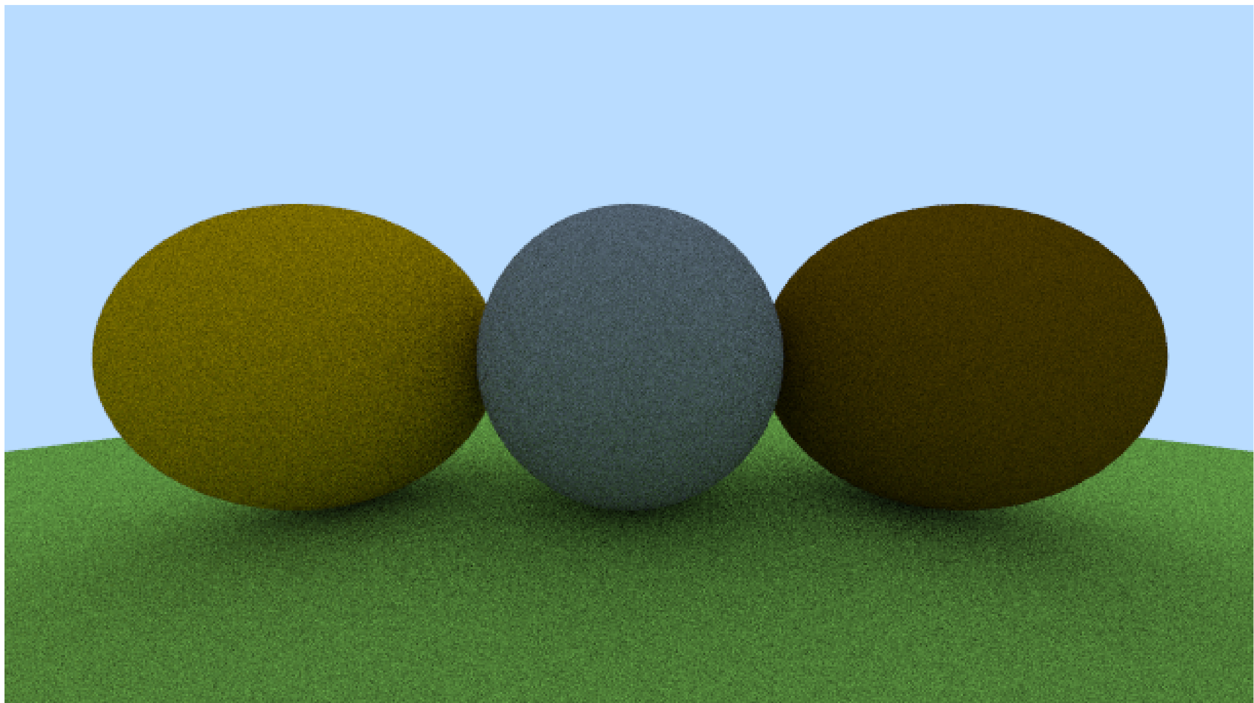
## Summary

In this project, we aim to implement a ray tracer from scratch in CUDA to run on our personal computers equipped with Nvidia RTX 2060 GPUs.

## Summary of Progress

We have implemented a CPU serial ray tracer based on the method of Monte Carlo rendering, which renders affine-transformed primitives with Lambertian materials. Below is an output image from our software renderer.

We are now about half a week behind our original schedule. We are currently working on the naive parallelization of our serial ray tracer using CUDA. We have made several unsuccessful attempts. The primary challenge in this step is transferring and representing the data structures in GPU devices. The lack of features such as std::shared_ptr and STL containers makes this task rather tricky. As from our resource of reference, our next strategy will be to allocate arrays of device pointers in host, and create shapes and materials in device.



## Changes in Ray Tracing Algorithm

After some initial research into the different ray tracing algorithms, we decided to switch from our initially planned method that spawns multiple rays per hit into a particular Monte Carlo method that can work with only spawning a single ray per hit. The initial method looks for light sources and computes refraction and reflection directions deterministically at each hit, while the Monte Carlo method we are looking at simply scatters a single random ray according to some material-dependent distribution function. (Note that there is no real requirement of only shooting out a single ray per hit, but we choose to do that as it is better for our purpose of parallelization.) With that, the Monte Carlo method we use approximates pixel colors by sampling rays for that pixel multiple times, with each sample having different light scattering on hit. The overall algorithm for Monte Carlo rendering is as follows:

```
For each pixel in image:
      For each sample:
            Intersect ray with scene
            If hit:
                  Sample scattered ray based on material
                  Calculate attenuation for the scattered ray
                  Compute color of the scattered ray (recursion)
                  Accumulate attenuation * scattered color to the pixel
      Average samples
```

We have not started working on the bounding volume hierarchy, which is something we wanted to leave for well after the milestone. We therefore have no changes to report for our plan for this BVH.

## Goals and Deliverables

### Plan to Achieve

As specified in the proposal, we still aim to have by the end of the project both a serial ray tracer as well as a CUDA parallel ray tracer that achieves a respectable speedup over the serial version. We still also aim to have a bounding volume hierarchy accelerate both serial and parallel versions for scenes with large numbers of geometry primitives.

We still plan to show results by running a handful of test scenes against the raytracers and displaying the rendered images alongside their rendering runtimes and any other useful statistics we can put in.

We also plan to measure and discuss the workload imbalance across pixels. We should be able to reference literature for potential solutions, although implementing them will be a far-reaching stretch goal.

### Hope to Achieve

We still hope to attain real-time framerates on many-triangle scenes. We would also like to add interactivity such as the ability to walk around the scene, which will involve presenting an interactive application instead of just static images or videos.

We would also like to see if there is anything we can implement to help with the workload imbalance.

## Platform Choice

The sheer amount of pixels (1920 x 1080 > 2 million) that can independently be processed makes the GPU a good candidate for speeding up ray tracing. Nowadays, it is reasonable to expect that personal laptop computers equipped with Nvidia RTX 20XX GPUs can handle ray tracing workloads well.

## Original Schedule

Week 1:
- Obtain geometry and scenes to test on (Stanford bunny, Dragon, Cornell box, etc.)
- Begin implementation of the serial ray tracer.

Week 2:
- Finish the serial ray tracer and start to work on the CUDA version.

Week 3:
- Finish the CUDA ray tracer.
- Collect milestone statistics on workload imbalance and performance in general.
- Attempt to set up an interactive application.

Week 4:
- Optimize (most likely with a BVH), with the goal of reaching real-time performance on respectable scenes.
- Finish interactive application.

Week 5:
- Fine tune scenes for presentation.
- Collect final statistics.

## Updated Schedule

Weeks 1 and 2 (done):
- Explore programming platforms.
    - Visual Studio turns out to be the best platform for our goal of having this run on our personal Windows laptops while maintaining focus on development and not fiddling with build processes too much. There is, however, a learning curve to this featureful IDE that we are not very familiar with.
- Design and implement simple scenes with spheres. Other geometry primitives such as triangles and boxes (useful for more common models like the Stanford bunny or scenes like the Cornell box) are not implemented yet.
- Begin implementation of the serial ray tracer.

- Finish the serial ray tracer and start to work on the CUDA version.

Week 3 (current week):

- Finish the CUDA ray tracer. (Not done yet.)
    - A challenge that we faced was that the convenient C++ data structures such as smart pointers and vectors are not compatible with CUDA which entails much restructuring.
    - As of now, the progress for our CUDA “ray tracer” has been mainly setting up the CUDA framework (which is a huge part) and it now renders the image below.

- Implement other geometry primitives on both serial and CUDA versions.
- The milestone report will not contain any statistics, only image results.

Week 4:

- Attempt to set up an interactive application.
    - This will involve more exploring the features that Visual Studio provides.
- Optimize (most likely with a BVH), with the goal of reaching real-time performance on respectable scenes.

Week 5:

- Finish interactive application if things go well. We are less optimistic about achieving this “nice to have” goal.
- Fine-tune scenes for presentation.
- Collect final statistics.