15618 Project Proposal:
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.

Background

Ray tracing is a graphics rendering technique in which pixel colours are computed by shooting rays out from the camera, hitting objects in the scene, and tracing further through the reflections and refractions. This is to be contrasted with an alternative rendering method called rasterization, where geometry primitives are drawn on the canvas, which is a process that has been highly optimized by rendering pipelines that we can program with OpenGL.

The input to the ray tracer will be a scene, which has to describe:

- Scene objects (with implicit geometries such as spheres and cubes, or explicit ones such as triangle meshes) along with their material properties
- Light sources
- Camera parameters

The output from the ray tracer will be a rendering of the input scene, which would be a 2D RGB image. In our project, we ideally want to render 1080p (1920 by 1080 pixel) images.

Here is a high-level pseudocode description of what the ray tracer does, using recursion:

To compute the color of a pixel:

- Cast a ray (or multiple rays if we are antialiasing) out from the camera location towards the direction that the pixel represents.
- Compute the color of that ray. (Or compute the colors of the multiple rays and average them if we are antialiasing.)

To compute the color of ray R:

- Intersect R with the scene.
- If there is no intersection, then the color of R is the background color at R's direction.
- If there is an intersection at point P, then the color of R is the sum of several components. The weights of these components simply depend on the material properties of the object at point P.
    - Light directly from light sources:
        - For each light source, we have to determine whether it shines directly on P by casting a ray from P to the light sources and intersecting these rays with the scene. We can use a Phong lighting model to calculate the contribution.

- Reflection:
  - If we have hit the recursion depth limit, do not compute reflections.
  - Otherwise, cast a ray that bounces off the object surface and recursively compute the color of that ray. The direction of the ray will depend on the object's normal vector at P.
- Refraction:
  - If we have hit the recursion depth limit, do not compute refractions.
  - Cast a ray that goes through the object surface and recursively compute the color of that ray. The direction of the ray will depend on the object's normal vector at P as well as its refractive index.

There will be at least one initial ray to cast per pixel, and even more with reflections and refractions down the recursion tree. With so many pixels in a 1080p scene, processing all these rays will be the most computationally intensive part of the algorithm, and it is also the major part that we want to parallelize due to our ability to process separate pixels independently.

In fact, once two different rays have been generated, they can be processed independently from each other. The dependencies come in the recursive generation of rays: we only know what rays to generate the next level of recursion after processing the ray at the previous level to see how it reflects of refracts off of objects. From the pseudocode above, it might look like a ray has to wait for its children rays to finish processing, but in reality their contributions can be computed separately, which is something that we could try in order to achieve more parallelism.

A final thing to mention is that scenes with large numbers of geometric primitives might need acceleration structures such as a Bounding Volume Hierarchy (BVH) to render at acceptable speeds. The BVH is a tree structure that can be cheaply intersected with a ray, in which each leaf represents a bounding volume that encloses actual scene geometry objects. Good BVHs will rule out expensive geometry intersection operations before they happen.

The Challenge

The problem is naturally recursive, and there is great variation in recursion depth among rays as they take their individual paths reflecting, refracting, and escaping the scene. Thus, a great challenge will be the distribution of workload and work balancing. We will also need to make decisions about how to deal with recursion: whether we should use an explicit recursion stack, how to divide the work across recursion levels among different threads, etc.

Resources

We plan to write the ray tracer from scratch with no starter code. We might use existing libraries, however, for other tasks such as OBJ file loading and interactive GUI applications. There are also multiple tutorials on CUDA ray tracing that are available online for our guidance, such as this technical blog by NVIDIA: Accelerated Ray Tracing in One Weekend in CUDA | NVIDIA Technical Blog .

The paper “Understanding the Efficiency of Ray Traversal on GPUs” by Timo Aila and Samuli Laine from NVIDIA Research discusses the gap between the theoretical best achievable parallel performance for ray tracing and the actual performance that we observe. They also provide a solution to narrow that gap, although that might be too much for our project.

There are resources available online that speak about the BVH, such as the Tree Traversal part of the “Thinking Parallel” blog series by NVIDIA.

Goals and Deliverables

Plan to Achieve

By the end of the project, we aim to have both a serial ray tracer and a CUDA parallel ray tracer that achieves a respectable speedup over the serial version. We also aim to have a bounding volume hierarchy accelerate both serial and parallel versions for scenes with large numbers of triangles.

To show the results, we plan to run a handful of test scenes against the raytracers and display the rendered images alongside the 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 point towards possible solutions from the literature, although implementing them might be a different story (for the stretch goals perhaps).

Hope to Achieve

If our optimizations are successful, we hope to attain real-time framerates (30+ fps on 1080p) on scenes with large numbers of triangles. 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.

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.