Implementing a GPU-Accelerated Ray-Tracing Renderer

Henry Heiberger
Harry Heiberger

6.S894: Accelerated Computing

December 11, 2024

1 Introduction and Background

When thinking of a GPU, one thing that obviously comes to mind is computer graphics. For years, Harry and I have always known that having a computer with a stronger graphics card was critical in order to be able to render very detailed or realistic virtual scenes. However, coming into this class, we only had a vague understanding of how GPUs worked and never really considered how today's nearly photorealistic video game scenes and animations were actually created. Thus, for our 6.S894 Accelerated Computing final project, we knew that we wanted to pursue something that improved our understanding of computer graphics and allowed us to be able to see for ourselves just how much of an effect a GPU had in accelerating a graphics renderer.

Given its high computational complexity and relatively recent emergence as a powerful technique for creating photorealistic images, we decided to specifically make ray-tracing the focus of our final project. As outlined in **Figure 1**, this technique simulates the physical properties of light by casting a series of rays from a virtual camera through each pixel into a scene. By keeping track of how these cast rays interact with every virtual object they collide with, graphics designers are able to use ray tracing to simulate stunningly realistic shadows, reflections, and other light effects.

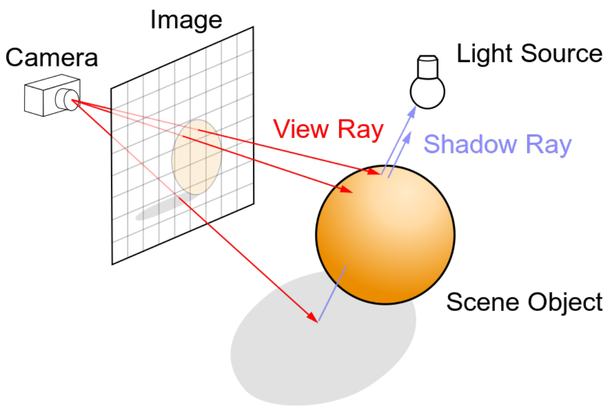


Figure 1: A diagram of the ray-tracing algorithm. Rays are cast from a virtual camera through each pixel and into the scene. The rays interact with the virtual objects within the scene, influencing the color value of the related pixel [1].

2 Related Work

Since neither of us had ever implemented a ray-tracing algorithm or any other computer graphics procedure before, our project started by spending a large portion of time researching how such graphics techniques are approached. Resources such as a broad definition of ray-tracing from Nvidia and a high-level article from OpenSystems Media on the algorithmic techniques clearly defined the ray-tracing problem for us and allowed us to start thinking about how to solve it [2, 3]. When it came to learning how to implement a graphics renderer entirely from the ground up, we found Gabriel Gambetta's *Computer Graphics From Scratch*, Peter Shirley's *Ray Tracing in One Weekend*, and the *Scratchapixel* website incredible resources for guiding us through the process in-depth [4–6].

3 Technical Approach

We approached the technical implementation of our final project in two main stages. First, we developed a fully functional C++ ray-tracing renderer primarily following the model provided within Shirley's book series [5]. Then, we sought to accelerate this CPU implementation by adapting the renderer to work with NVIDIA's CUDA model and applying many of the optimization techniques discussed in class over the course of the semester.

3.1 Implementing a C++ Ray-Tracing Renderer

The first stage of our final project involved developing a fully-functional C++ ray-tracing renderer. Because this would serve as a CPU baseline to compare against our later optimizations, we wanted to make sure that this renderer could produce scenes that were both clean to look at and computationally intense enough that a pure CPU implementation would take significant time to render. As a disclaimer, we are both new to graphics pipelines and rendering. As such, the specific ray-tracing algorithms we chose to implement in our CPU implementation focused on simplicity and understandability, following the approach taken in Shirley's book series, rather than being the fastest version in the literature [5].

Like most renderers, our implementation functions through the abstractions of a camera, viewport, and scene. The viewport sits between our camera and scene, providing a 2D rectangular mapping between the 3D objects in our world and the pixels in our output image. The camera functions by shooting rays through the viewport into our scene. The output image pixels correspond to the viewport location in which the camera's rays cross through the viewport, and they are colored based on that ray's interactions within the scene.

A pixel's specific color is chosen based on the movements of the ray within the scene. When a ray passes through our viewport, we loop over the list of all objects in the scene and determine which object, if any, is the closest object that the ray strikes. As our renderer currently only supports spheres, collisions can be calculated using the sphere equation and the characteristics of the incoming ray. Once the closest to the camera collision is determined, we then calculate what should happen with the ray after impact. This highly depends on the material characteristics of the impacted sphere. If the ray is not absorbed, the new trajectory of the ray is calculated based on the local reflection or refraction properties, and our raycolor function recurses to determine how this newly transmitted ray will interact with the scene. In our implementation, this process continues until either the ray misses all objects (colliding with the sky), is absorbed by a material, or has bounced a maximum of 50 times. The final output color of each pixel is a weighted sum of the various bounces where each bounce contributes a decreasingly smaller component to the final color based on impact material properties.

In our renderer, we implemented three distinct materials: a diffuse (matte) material, a metal material, and a glass material. These material classes each have member functions that hold the various calculations for determining how rays interact on collision. Various graphics approximation formulas such as Snell's Law and Schlick's approximation were used to make the materials interact as realistically as possible.

Besides materials, we also implemented a variety of computer graphics techniques to improve the realism of our scene. The most computationally intense of these is anti-aliasing. Recall that the real world operates in a continuous color space where the edges of one object blur into the next object. However, as we are creating a renderer based on pixels, we operate in a discrete space where there are distinct pixel boundaries between objects. This results in a jagged appearance that hurts realism. Anti-aliasing resolves this by setting a pixel's color as the average of a randomly selected subset of a location's surrounding pixels, simulating a continuous space. Our specific implementation samples 22 randomly selected surrounding pixels, determining the color (and potentially many light bounces) for each.

3.2 Accelerating with CUDA

After obtaining a vanilla C++ ray-tracing renderer to use as a CPU baseline, the second stage of our final project was to accelerate it as much as possible using our GPU. To do this, our first major step was to convert our pure-C++ implementation into one that could work with the CUDA parallel programming model. This involved tagging functions with the appropriate host or device tags, refactoring our various functions so that they can be called via a main CUDA kernel call, and allocating our various objects into GPU memory. This resulted in the creation of a create_world and free_world kernel call that builds up our scene on the GPU. As some of our calculations utilized random values, we also needed to create a rand_init kernel that utilized the cuRAND library to allocate a random state variable for each thread. After performing all of these changes, we ended up with an unoptimized Cuda renderer that was able to generate an identical output to the CPU renderer.

Ray tracing is inherently a very parallelizable problem. As such, there are a variety of ways it can be optimized using CUDA. Our first and biggest optimization was splitting up our rendering process into blocks where the threads in each block each compute a pixel of the output image. As pixels are computed independently of each other, there was no risk of race conditions for this optimization. The specific size of our blocks was tuned to maximize performance. The various sizes tested and their resulting speeds when rendering our reference image are shown in the table below.

Table 1: Render Time for Various Block Dimensions

Block Dimensions (pixels)	Render Time (sec)
4 x 2	3.72
4 x 4	1.67
8 x 2	2.28
8 x 4	1.43
8 x 8	2.12
16 x 2	1.22
16 x 4	3.22
16 x 8	4.89
16 x 16	4.00
32 x 2	3.53
32 x 4	4.26
32 x 8	2.56
32 x 16	2.94
32 x 32	3.87

Beyond breaking the problem into parallel computed CUDA blocks, we also performed a series of additional optimizations in order to further increase the performance of our GPU-accelerated renderer. One way in which this was done was searching for ways to add register reuse and instruction-level parallelism to our renderer. Recursive implementations of functions were transformed into iterative approaches, and most of our computations and looping structures were refactored in order to provide as many opportunities for resue as possible.

Likewise, our implementation also took advantage of shared memory. Each pixel and anti-aliasing sample needs to loop over all objects in the scene in order to determine the output color. Thus, to save memory access time on these scene objects, we decided to store them in a shared memory buffer that is generated before the render process and is now used for all color calculations.

Finally, we also examined our various mathematical calculations for potential optimizations. Double-precision math in Cuda is inherently slower than single-precision math, so we converted all double-precision operations, giving us a small performance boost. We also implemented some approximations that didn't visually change the output appearance but resulted in faster computation.

4 Results

By the conclusion of our project, we successfully implemented both a vanilla C++ and GPU-accelerated CUDA ray-tracing renderer that were both able to generate images similar to the one displayed in **Figure** 2. While the initial renderers are only able to support generating sphere objects with opaque, reflective, or refractive material, they are easily expandable and could quickly be adjusted to support other shapes, materials, or textures.

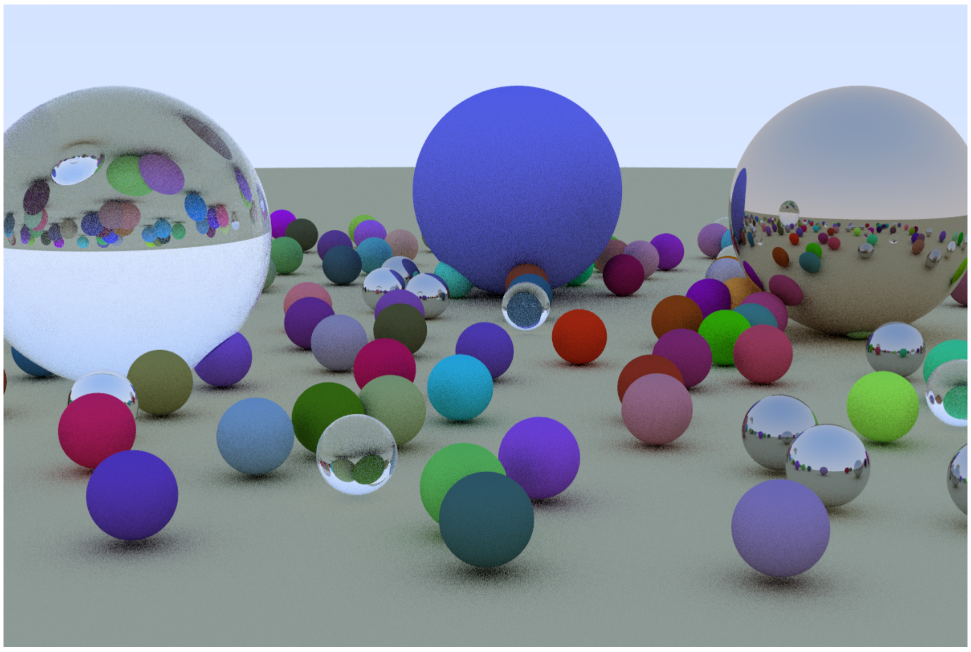
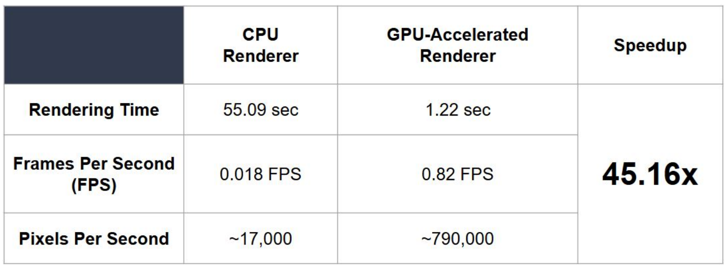


Figure 2: An example scene rendered by our CPU and GPU-accelerated ray-tracing renderer. The scene contains a collection of spheres composed of three materials, an opaque material that absorbs light, a metal material that reflects light, and a glass material that refracts light.

Regarding performance, our basic CPU baseline renderer was able to render the scene displayed in **Figure** 2 in approximately **55 seconds**. In contrast, after implementing the same renderer in CUDA and performing optimizations, we were able to reduce this time to only a little over **1 second**, resulting in greater than a **45x speedup**. While this performance is still a little slow to generate frames in real time, for a renderer that only outputs still images, this one second wait is far more reasonable than nearly one minute. A more detailed summary of this performance comparison can be seen in **Figure** 3.



	CPU Renderer	GPU-Accelerated Renderer	Speedup
Rendering Time	55.09 sec	1.22 sec	**45.16x**
Frames Per Second (FPS)	0.018 FPS	0.82 FPS	
Pixels Per Second	~17,000	~790,000	

Figure 3: A table comparing the performance of our vanilla C++ CPU baseline and our GPU-accelerated CUDA ray-tracing renderer when rendering the scene displayed in **Figure** 2

.

5 Reflection and Next Steps

Over the course of this final project, we were able to implement a functional C++ ray-tracing renderer that supported rendering simple scenes and then optimize it to achieve render times of around one second. While originally we had hoped to create a GPU-accelerated ray-tracing renderer that could achieve real-time rendering (24+ FPS) performance, coming from almost no background in graphics and completing this project at the same time as two other major final projects, we are quite satisfied with the end result we were able to obtain. Our current renderer is a strong baseline that can easily be expanded to include additional objects and materials, and its performance can continue to be improved by researching more advanced graphics techniques. Additionally, throughout the process of working on it, we both feel that our understanding of computer graphics has greatly improved and are prepared to take on even larger and more complicated graphics projects.

Regarding next steps, one particularly interesting area involving ray-tracing that we would like to explore are the performance gains brought by the RT Cores included in recent NVIDIA GPUs. NVIDIA describes these cores as accelerator units dedicated to performing ray-tracing operations with "extraordinary efficiency" [7]. Interestingly, these RT cores are not accessible through vanilla CUDA, even by directly writing PTX instructions. Instead, they can only be accessed implicitly through three major ray-tracing APIs: NVIDIA OptiX [8], Microsoft DXR [9], and the Vulkan Ray Tracing Extensions [10]. In a forum post from 2023 describing why this is the case, A NVIDIA moderator stated that a PTX instruction set for the RT Cores hasn't been exposed yet because the cores are frequently changing between GPU generations and making them directly accessible would limit their advancement [11]. When we were first thinking about this project, we thought it would be interesting to compare our vanilla CUDA-accelerated ray-tracing renderer with an equivalent one implemented using the NVIDIA OptiX engine. While we were not able to get to this done during our project due to the other implementations being more complicated than expected and the OptiX engine having a fairly large learning curve, given additional time, we would love to investigate the performance gains of these RT Cores.

Furthermore, in this project, we only implemented a handful of the effects that can be produced by a ray-tracing renderer. By exploring more sophisticated techniques for features such as motion blur, ambient occlusion, variable light sources, and texture maps, our renderer can continue to produce more visually interesting scenes.

References

[1] Wikipedia contributors. *Ray tracing (graphics) — Wikipedia, The Free Encyclopedia*. [Online; accessed 10-December-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Ray_tracing_(graphics)&oldid=1253778958.

[2] NVIDIA Developer. *Ray Tracing*. NVIDIA, 2024. URL: https://developer.nvidia.com/discover/ray-tracing (visited on 12/09/2024).

[3] Embedded Computing Design. “Ray tracing for beginners”. In: *Embedded Computing Design* (Aug. 2015). URL: https://embeddedcomputing.com/technology/software-and-os/ray-tracing-for-beginners.

[4] G. Gambetta. *Computer Graphics from Scratch: A Programmer’s Introduction to 3D Rendering*. San Francisco: No Starch Press, May 2021, p. 248. ISBN: 9781718500761.

[5] S. H. Peter Shirley Trevor David Black. *Ray Tracing in One Weekend*. https://raytracing.github.io/books/RayTraci
Aug. 2024. URL: https://raytracing.github.io/books/RayTracingInOneWeekend.html.

[6] Scratchapixel. *Scratchapixel*. Accessed: 2024-12-09. 2024. URL: https://www.scratchapixel.com/.

[7] NVIDIA. *NVIDIA RTX Ray Tracing Technology*. Real-time ray tracing technology powered by NVIDIA RT Cores. NVIDIA Corporation, 2024. URL: https://developer.nvidia.com/rtx/ray-tracing.

[8] S. G. Parker et al. “OptiX: a general purpose ray tracing engine”. In: *ACM Trans. Graph.* 29.4 (July 2010). ISSN: 0730-0301. DOI: 10.1145/1778765.1778803. URL: https://doi.org/10.1145/1778765.1778803.

[9] N. Developer. *Introduction to NVIDIA RTX and DirectX Ray Tracing*. NVIDIA, 2018. URL: https://developer.nvidia.com/blog/introduction-nvidia-rtx-directx-ray-tracing/ (visited on 12/09/2024).

[10] Khronos Group. *Ray Tracing in Vulkan*. 2024. URL: https://www.khronos.org/blog/ray-tracing-in-vulkan (visited on 12/10/2024).

[11] NVIDIA Developer Forum. *Any Lower Access to RT Core than OptiX?* 2023. URL: https://forums.developer.nvidia.com/t/any-lower-access-to-rt-core-than-optix/262955 (visited on 12/10/2024).