Rafael Padilla Perez - U1592528

rpadiper@gmail.com

Austin H Kim - U1545784

liliumahk@gmail.com

Conner Murray - U1197807

conner.murray@utah.edu

Path Tracing Renderer Profiler (RayStats)

GitHub Repository: https://github.com/Rafapp/raystats

Background

A renderer is a software program that transforms three-dimensional scene descriptions into two-dimensional images for display. It takes scene data—including objects, materials, lights, and camera parameters—and mathematically converts this 3D information into a flat image that can be shown on screen.

Path tracing is a sophisticated rendering algorithm that achieves photorealistic results by simulating how light behaves in the real world. The technique works by casting multiple light rays from the camera through each pixel into the scene. As these rays travel, the renderer calculates their interactions with surfaces: how much light is absorbed, reflected, or transmitted by different materials. The algorithm traces these light paths as they bounce between objects, accumulating lighting information along the way. Finally, all this gathered data is processed to determine the correct RGB color value for each pixel, which is then written to create the final rendered image.

Motivation

For other coursework this semester, we are building a path tracing renderer from scratch, which presents an excellent opportunity for performance measurement and optimization. Since images have a width w and height h, the total number of pixels to calculate is $w \times h$. This operation scales quadratically with image dimensions—for a square image of side length n, the pixel count becomes n^2, resulting in $O(n^2)$ computational complexity.

Performance is absolutely critical in path tracing because the complex ray calculations must be executed for every pixel. In production environments like animated feature films, where render times directly impact project timelines and costs, returning images as quickly as possible isn't just useful—it's essential. Modern animated films often render at 4K resolution (3840 × 2160 pixels) or higher, meaning over 8 million pixels per frame, with each pixel requiring hundreds or thousands of ray samples for photorealistic quality.

To guarantee optimal renderer performance, systematic measurement is crucial for identifying computational bottlenecks. Different stages of the path tracing pipeline—ray generation, scene intersection testing, material evaluation, and light sampling—can exhibit varying performance characteristics depending on scene complexity. This is why we propose creating a comprehensive data visualization dashboard for our renderer to monitor and analyze performance metrics in real-time.

Objectives

1. *Identify Critical Performance Bottlenecks*: Determine which computational metrics (ray intersection tests, memory allocation, sample convergence rates) provide the most actionable insights for path tracer optimization across different scene complexities.

2. *Design Effective Data Visualizations*: Research and evaluate visualization techniques for rendering performance data, comparing our approach to industry solutions like Hyperion's Renderer Statistics Viewer to establish best practices for intuitive data representation.

3. *Build a Real-Time Performance Dashboard*: Develop a comprehensive web-based interface that aggregates and displays path tracing metrics with responsive charts and interactive controls for live performance monitoring.

4. *Achieve Measurable Performance Improvements*: Use the dashboard's insights to identify and resolve performance bottlenecks, resulting in quantifiable speedups in our path tracer's rendering times.

5. *Establish a Reusable Analysis Framework*: Create a systematic approach to performance profiling that can be applied to future rendering projects and provide educational value for understanding path tracing optimization principles.

Data

Our data will be collected from a path tracing renderer built in CS 6620. During the rendering of a scene, the renderer will log our desired performance metrics on a scene level, pixel level, and ray level. These metrics will be collected into a CSV file for easy aggregation and filtering.

Data Processing

We expect to do moderate data cleanup, mainly to normalize or filter out incomplete or corrupted frame statistics, remove outliers, and ensure consistency across multiple frames.

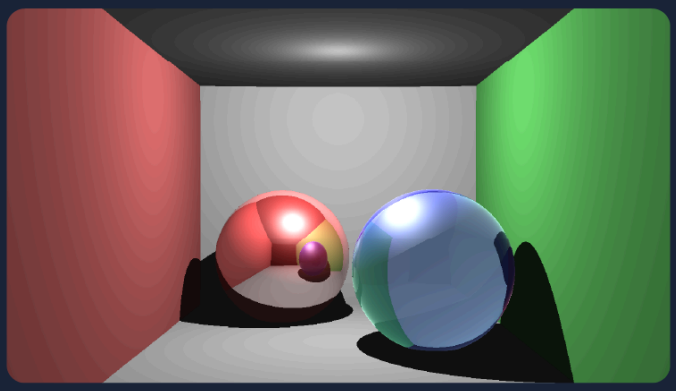
From the raw rendering data, we plan to derive a variety of quantities that provide insight into both spatial and pipeline-level performance, including:

- **Histograms** of pixel sampling counts, cache hits, ray bounces, and shading times to understand distribution patterns and variance.
- **Aggregated frame statistics**, such as average rays per pixel, per-frame rendering time, memory usage, and sample convergence rates.
- **Scene statistics**, including object counts, triangle counts, levels of detail, and per-object or per-material rendering cost.

Visualization Design

RayStats: V1

SCENE



name	CornellBox.usd
objects	7
materials	2
lights	3
scene file size	27 kb

STATS

Ray

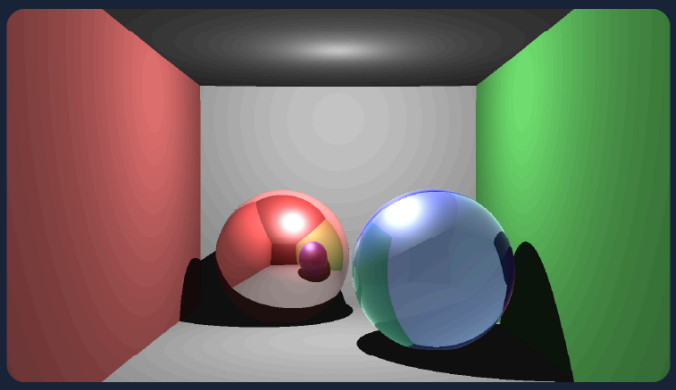
rays	1.783 M
bounces	2.437 M
samples	8.783 M
samples	9.846 M
...	...

Shader

most hit	blinn-phong
least hit	PXRVolume
dot products	8.45 M
square roots	1.692 K
...	...

RayStats: V2

SCENE

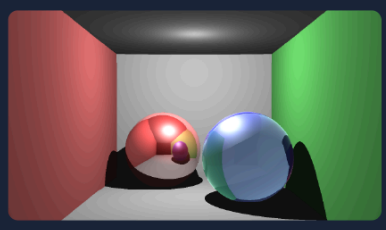


name	CornellBox.usd
objects	7
materials	2
lights	3
scene file size	27 kb

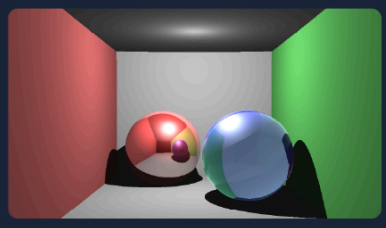
Ray | Shader | Material

Ray

rays	1.783 M
bounces	2.437 M
samples	8.783 M
samples	9.846 M
...	...



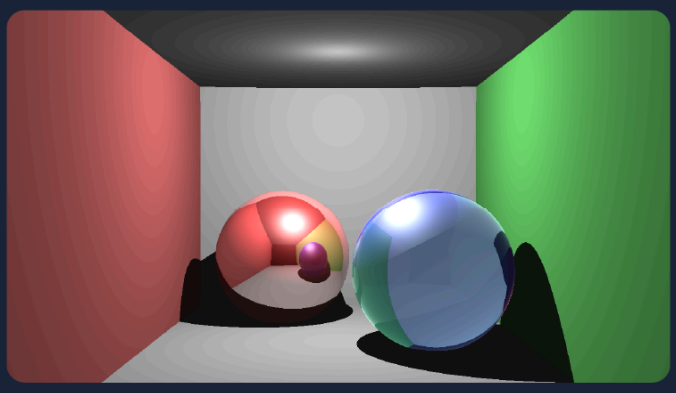
slowest frame 1.2 s



fastest frame 196 ms

RayStats: V3

SCENE

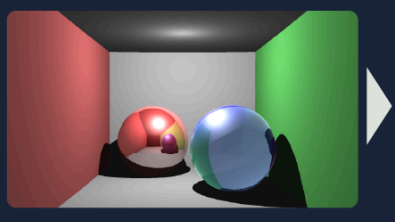


name	CornellBox.usd
objects	7
materials	2
lights	3
scene file size	27 kb

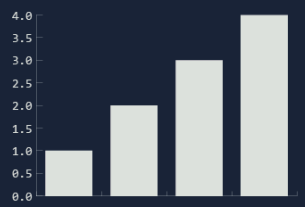
Ray | Shader | Material

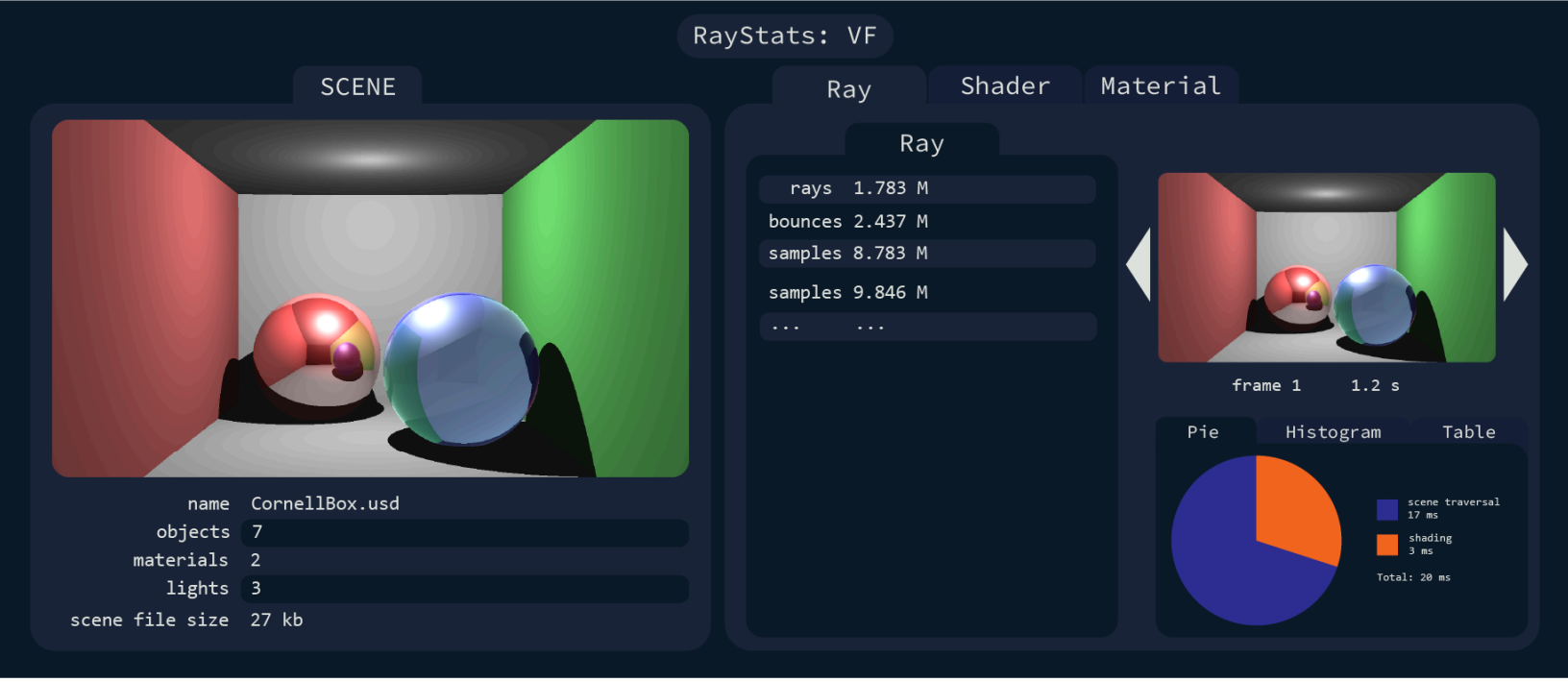
Ray

rays	1.783 M
bounces	2.437 M
samples	8.783 M
samples	9.846 M
...	...



frame 1 1.2 s





V1 - Shows all data as tabular data

V2 - Shows statistics broken down into ray, shader, and material specific categories. Also provides frame comparisons.

V3 - Demonstrates more performance metrics through graphed performance indicators. Users will be able to switch between different visualizations including frames that took the longest to render, heat mapped scenes, and z-buffer data.

V4 - A suite of different graph visualizations of performance indicators for each category and frame visualization.

Must-have features

- The ability to collect and summarize metrics such as rays per pixel, shading time, and convergence rates. Without this, the visualizer cannot provide concrete data required for analyzing performance.

- Generate heatmaps or other visualizations showing which areas of the scene or require the highest computational cost. Without this, identifying local bottlenecks in the scene is impossible.
- The visualization must be interactive, allowing the user to explore performance trends through zooming, filtering, and selecting regions of the performance visualizations.

Optional features

It would be nice to be able to break down performance on a per-object or per-material basis. That way, we could analyze more specific elements of a scene. Having side-by-side analysis of scenes rendered with different configurations would enable us to compare and contrast different optimizations. If we had time sampled data, such as a moving camera or animated object, tracking the performance of these through animated visualizations could also be useful.

Project Schedule (weekly)

Week 1-2: Setup, Plan, and Mockups

Set up a repository and research other rendering and GPU profilers. Figure out the scope of our projects and see what we should absolutely include and what features we can leave out. Create basic mockups of what we want and improve upon the visualization over the coming weeks. Research the design and features of existing industry level profilers if available such as Disney's Hyperion.

Week 3-4: Demo Raytracer

Create a ray tracer that features multiple light bounces, reflections, refractions, and shadows. Towards the end of the project, we should have path tracers alongside more robust data to conduct our final tests on. We will still demo on our ray tracers until then.

Weeks 5-7: Profiling Infrastructure

Include methods to collect data in our renders, which should include the total number of rays cast, number of bounces, traversal time, shading time, traversal speed, and CPU Time. These will be implemented over time as our ray tracers become more robust. Create a mockup that is closer to our intended vision

Weeks 7-9: Visualizations

Optimize our renderers by including accelerated structures, BVH, and multithreading since we will be including models with more geometry into our scenes. Start implementing visualization outputs of our data including charts, tabular data, and heatmaps. The heatmap may take longer depending on how we want it to be displayed.

Week 10: Create Demos

We should run larger test scenes and a stress-test on our profiler. Create multiple test scenes and have some of the profiling results available to our webpage.

Week 11: Refining the UI

Clean up the logging format and how the rest of the data is being displayed with the test scenes we created a week prior. Focus on refining the UI and create a more stable and refined profiler.

Week 12: Integration

Run integration test to see if it works across all three of our renderers. Use this opportunity to check if the data we are receiving are consistent with our expectations. We should have a fully working profiler to demo by the end of the week.

Week 13: Upgrading to Path Tracers

Complete our Path Tracers to include global illumination to our scenes. Considering the significantly larger sample size of rays we will get compared to the original ray tracers, we will run the profiler and use this as our finals tests.

Week 14-15: Final Imaging & Evaluations

Apply our profiler to our Utah Teapot Competition Renders and demo everything on it for our final product. The profiler should work for all of our renderers and should be consistent in outputting data for all three of our renders.

Week 16: Final Reports

Double check the demo, webpage, and the write up if there is one. Review and see if we think we all did sufficiently throughout the project and what we would do next time if we decide to improve upon the profiler.