ANISH KAMBHAMPATI[1], DURGA MANDARAPU[1], MILIND KULKARNI[2]

Analyzing the Performance of GPU-Based Point-in-Polygon Algorithms

Read the full paper

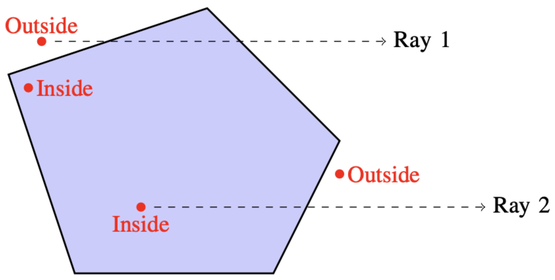
PURDUE UNIVERSITY
John Martinson Honors College

[1] *Department of Computer Science, Purdue University*
[2] *Elmore Family School of Electrical and Computer Engineering, Purdue University*

Acknowledgements: *We would like to thank Durga Mandarapu and Dr. Milind Kulkarni for their invaluable mentorship. We also thank the John Martinson Honors College for their funding support.*

OVERVIEW

The Point-in-Polygon (PIP) query determines if a point lies inside a polygon with applications spanning computer graphics, geographic information systems, and scientific simulations. While computations have traditionally been on CPUs, the emergence of GPU computing has enabled high-performance alternatives. We benchmark three GPU-based PIP algorithms—CUDA ray casting, RayJoin, and s-ray—on NVIDIA hardware. Our results show trade-offs between query and preprocessing times, providing a foundation for practitioners to select and/or develop their own PIP algorithms.



An example of the point-in-polygon (PIP) query using the ray casting method.

INTRODUCTION

Traditional algorithms for the PIP problem leveraged simple geometric principles. Examples include the ray-casting (shown above) and the winding number algorithms [1, 2]. However, the rise of GPUs have brought novel features that can be used to enhance or even improve upon these. One such feature is the CUDA paradigm which enables developers to write high-performant, parallel code that can run in GPU kernels. Another feature is hardware-accelerated ray tracing (RT), which simulates the behavior of light by tracing rays as they interact with objects in the scene. While typically used in graphics, RT capabilities have found uses in high-performance computing (HPC) applications, aided by NVIDIA's OptiX framework [3]. We aim to compare the performance of three algorithms which leverage some of these advanced GPU capabilities.

REFERENCES

[1] D. Horvat and B. Zalik, "Ray-casting point-in-polyhedron test," Proceedings of the CESCG, 2012.

[2] K. Hormann and A. Agathos, "The point in polygon problem for arbitrary polygons," Computational Geometry, vol. 20, no. 3, pp. 131–144, 2001, doi: https://doi.org/10.1016/S0925-7721(01)00012-8.

[3] S. G. Parker et al., "Optix: a general purpose ray tracing engine," Acm transactions on graphics (tog), vol. 29, no. 4, pp. 1–13, 2010.

[4] L. Geng, R. Lee, and X. Zhang, "RayJoin: Fast and Precise Spatial Join," in Proceedings of the 38th ACM International Conference on Supercomputing, 2024, pp. 124–136. doi: 10.1145/3650200.3656610.

METHODOLOGY

The three algorithms we compared are the following:

1. CUDA ray casting — this algorithm works exactly like the CPU-based ray casting algorithm. The only difference is that the ray casting for each point is done in parallel.
2. RayJoin — this algorithm is a state-of-the-art (SOTA) GPU-based spatial join library. By reframing the PIP query as a ray tracing problem, RayJoin is able to take advantage of hardware RT acceleration [4].
3. s-ray — this algorithm is similar to RayJoin in that both make use of RT cores. The difference lies in how the build indices are created. In the former, they are constructed around polygons, while in the latter, they are constructed around polygon edges. This is a novel, work-in-progress algorithm.

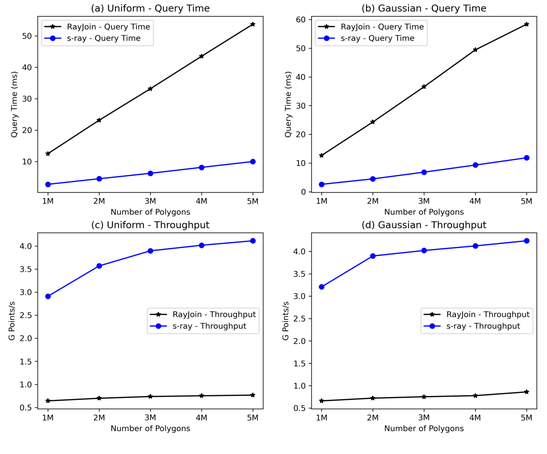
To compare the performance of all three, we evaluate them on two distinct scenarios. The first has significantly more points (~117M) than polygons (117). The second contains a more balanced distribution of points (~50M) and polygons (1M-5M).

RESULTS

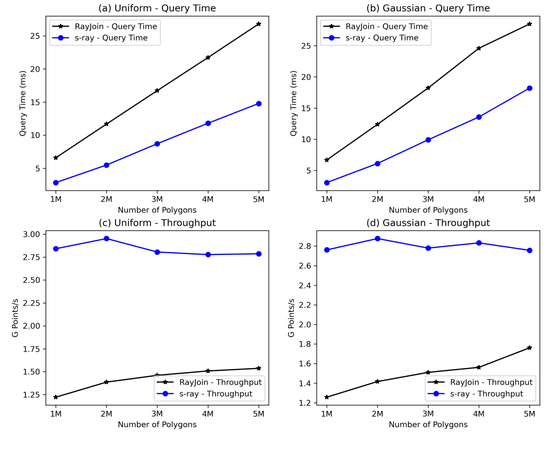
In the first scenario (shown right & tested on an NVIDIA RTX 4070Ti), we find that while the CUDA ray casting approach does not have a build step, it has by far the slowest query time. RayJoin does require a build step, but it has a much faster query time. Notably, s-ray (despite its slow build time) is twice as fast as RayJoin.

We tested the second scenario (shown below) on both an RTX 4070Ti and RTX 6000 GPU as well as on both uniformly and normally distributed datasets. Due to CUDA ray casting's poor performance in the previous scenario, we opted to only test RayJoin and s-ray here. From the results, we see that s-ray has significantly higher throughput than RayJoin. However, RayJoin has much lower build times, which leads to it being the faster algorithm. One promising sign for both algorithms is that their query time appears to scale linearly with the number of polygons, which indicates that both should work on larger datasets. A counterintuitive observation is that s-ray's throughput is higher on the 4070Ti than the 6000. We have yet to explore the reasons behind this discrepancy.

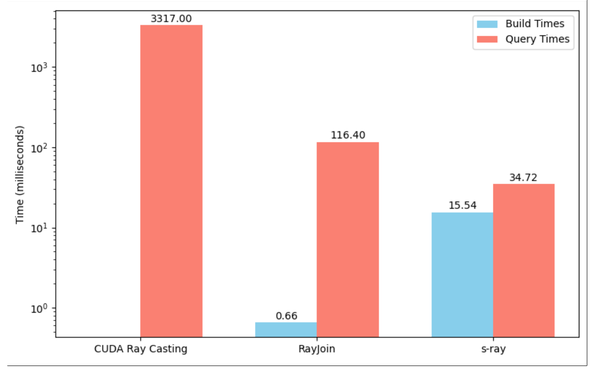
QUERY TIME, RTX 4070TI



QUERY TIME, RTX 6000



QUERY TIME, NYC DATASET



DISCUSSION

CUDA ray casting is the slowest algorithm, and this has to do with RT's hardware acceleration. In contrast, the CUDA-based algorithm executes intersection tests in software. While both RayJoin and s-ray are faster, the former has a faster build time and the latter has a faster query time. This has to with RayJoin's adaptive grouping (AG) step, which lowers the number of primitives in the bounding volume hierarchy (BVH). RayJoin was overall slower than s-ray in scenario one but faster in scenario two, indicating the potential of both algorithms in HPC applications. We also note that both algorithms performed slower with normally distributed datasets. This has to do with spatial locality — uniformly distributed workloads allow BVH traversals to be more even and avoid clustering around the mean.

In terms of future work, one path would be implementing AG for s-ray. This would drastically lower build times (which in scenario two could be over 3x RayJoin's). We would, however, need to ensure that our query times do not go up significantly. Another path would be to test polygons in parallel for the CUDA ray casting algorithm. Currently, only points are tested in parallel which renders the algorithm impracticable on datasets with millions of polygons.