Anish Kambhampati

ECE 496 Semester 2 Final Report

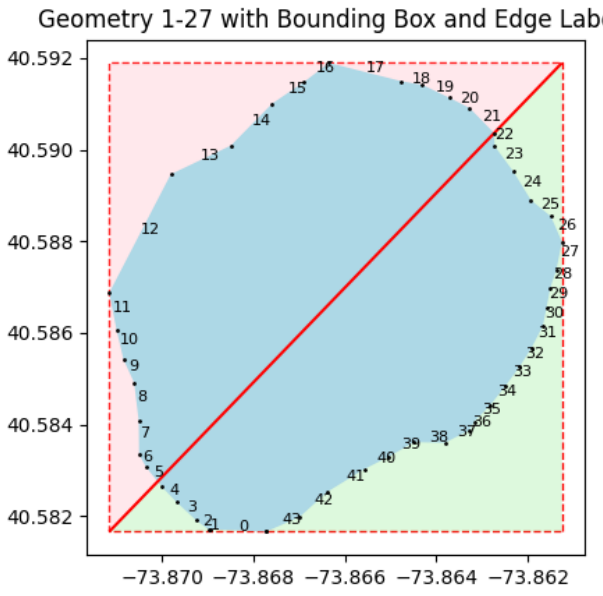
Introduction

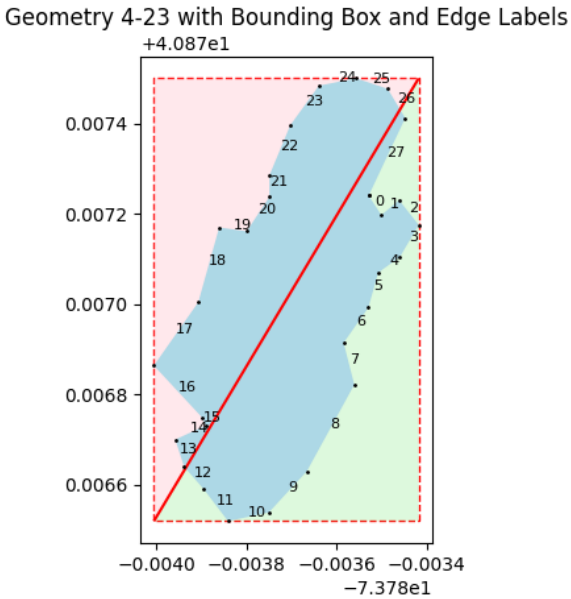
Last semester, I wrote a much more formal report detailing the work that I did. I structured it like a research paper, clearly describing my contributions in the larger context of the field. For this semester's report, I will be much more informal. The reasons for this are twofold. First, I only took one credit of research instead of the three from before, meaning that the work I did was a whole lot less. Second, there was no underlying theme in what I did, meaning that writing a cohesive report would be impossible. In this report, I describe the three main things I did. First, I discuss an RT-based PIP (point-in-polygon) algorithm that Durga (my PhD mentor) wanted me to write. Second, I describe an ear-clipping CUDA PIP algorithm, which while I technically wrote last semester, was not mentioned at all in my report then. Considering that she wanted that work now, I felt it appropriate to discuss it here. Finally, I will briefly touch on my experience at the National Conference on Undergraduate Research (NCUR), where I presented the research I worked on last semester.

RT-Based PIP Algorithm

At the end of last semester (and early this semester), the s-ray algorithm that Durga and I were working on was faster than RayJoin for PIP in terms of query time but slower in terms

of build time. As a refresher, by build time, I mean the time it takes to construct the BVH, and by query time, I mean the time it takes to traverse the BVH. One of the goals for this semester was to reduce the number of primitives in the BVH to reduce build time. Durga's idea for this was simple — for each polygon, construct its bounding box and then divide it along the diagonal to create two triangles per polygon. By doing this for every polygon and using those triangles as the BVH primitives, the number of primitives scales linearly with the number of polygons and not the number of polygon edges. (The reason why we focus on triangles is because they come with native RT hardware acceleration.) Durga further wanted to record which edges of each polygon were in each triangle, with the hope to somehow check only those edges once a point-triangle intersection was detected. She tasked me with the implementation of this idea. I initially wrote the bounding box algorithm in Python with geopandas, working with the NYC borough boundaries dataset, starting with a script to pictorially show the bounding box and edges. Two examples are below.



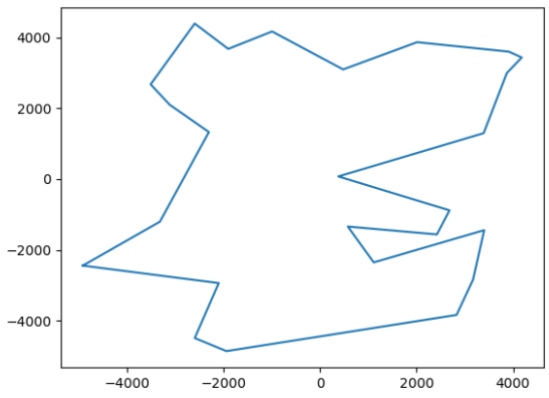


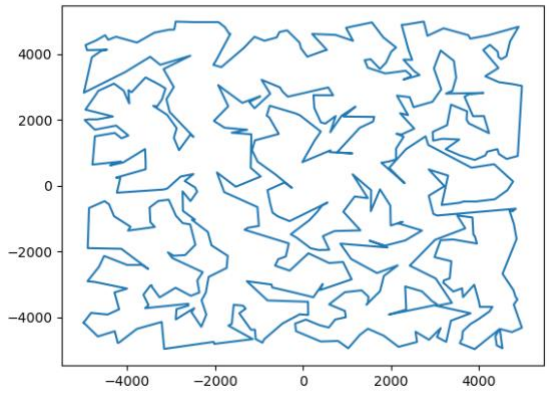
My Python script does not actually compute PIP queries. All it does is return each polygon, its two triangles, and the edges partially or fully contained with each triangle. My next step was to convert this to C++. I ported over the edge computations using the GDAL C++ API. I didn't port over the plot creations for obvious reasons. I then verified that my C++ and Python scripts returned the same output.

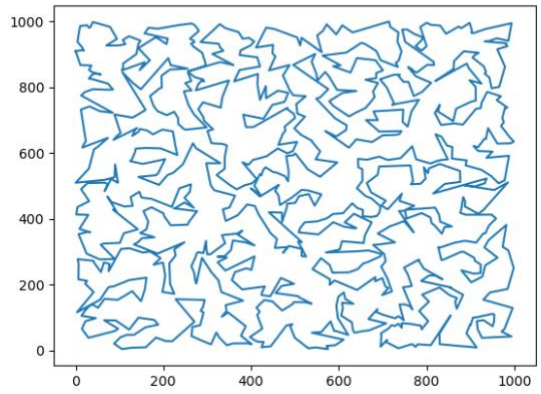
The final step was to use OWL (OptiX Wrapper Library) to perform the RT-accelerated PIP queries. I set each point's z coordinate to -0.1 and set each triangle slightly offset (1e-4) from every other triangle in the +z direction. Next, I shot a ray from each point in the +z direction. I needed to use an anyhit shader as opposed to a closesthit as it was possible for two triangles from different polygons to overlap. For this step, while I chose to send the edge info to the GPU device, the shader itself doesn't make use of this information. It simply computes the true intersection status using a ray-casting approach which I extensively described in my report from last semester. In any case, I was able to successfully write this program and execute it. The build time was ~15.048 milliseconds and the query time was ~1.3 seconds. This is way worse than s-ray, which had a build time of ~15.54 milliseconds and a query time of ~34.72 milliseconds. I suspect the reason for this is both the highly expensive ray-casting computation in the shader. Another reason is that on this specific dataset, the small number of polygons (117) in comparison with the number of points (~112M) meant that any build time was insignificant in comparison with the query time. I suspect that the implementation I wrote will perform better on datasets with more polygons, but I did not have time to test that.

Ear-Clipping Algorithm

This was something I worked on last semester but did not get a chance to touch on at all. As I mentioned previously, RT has hardware acceleration support for triangles. So, one possible approach would be to deconstruct each polygon into a bunch of triangles and build a BVH around that. This problem is known as polygon triangulation, and while it can be solved easily for convex polygons, doing it for all simple polygons is much harder. I used the Earcut library to perform the triangulation for me. I unfortunately did not test this with RT acceleration, instead opting to only test it using CUDA and a fast point-in-triangle implementation. I tested this against my CUDA ray casting approach and made sure that both approaches yielded the same results. I noticed that when using single-precision floating point (FP) numbers for calculations, some results would differ between the implementations. These discrepancies went away when using double-precision FP numbers, but the time taken was significantly longer. I tested both algorithms with just a single polygon with a varying number of edges, as well as a varying number of points. I generated the polygons using a polygon generation library from two students at Princeton University. The three polygons I used had 25 (left), 500, and 1000 (below) edges.







I generated points uniformly at random from the range [-5000, 5000] in both the x and y directions. I generated 5 datasets, with 1×10^3, 5×10^4, 2.5×10^5, 1×10^6, and 1×10^7 points. The resulting queries times are summarized in the below table. For ear clipping, I report two times — the first is the time it takes to compute the polygon triangulation (the build time) and the second is the query time. In addition, all times are in milliseconds.

Points	**25 Edges**		**500 Edges**		**1000 Edges**	
	Ray Casting	Ear Clipping	Ray Casting	Ear Clipping	Ray Casting	Ear Clipping
1,000	2.861	0.009 0.041	2.751	0.238 0.627	3.308	0.522 1.262
50,000	2.942	0.009 0.070	3.399	0.238 1.264	3.246	0.522 2.516
250,000	2.971	0.010 0.216	4.627	0.240 4.397	4.567	0.520 8.798
1,000,000	3.720	0.009 0.823	11.022	0.241 17.496	9.669	0.530 35.051
10,000,000	9.192	0.010 7.981	83.536	0.241 172.421	70.165	0.518 345.436

From the above table, we see that the ear clipping approach works with a smaller number of points and/or polygon edges. However, it does not scale as effectively as the ray casting

approach, rendering it suboptimal on larger datasets. I expect that future work will create RT-accelerated approach and test it against other GPU-based algorithms.

NCUR Presentation

Along with the work I did this semester, I also presented my GPU PIP algorithm comparison at NCUR 2025 in Pittsburgh, PA. It was my first research conference, and I really enjoyed the opportunity to present my work to a wider audience. I met a lot of wonderful people who were interested in my work. I even met an older gentleman who previously worked in the aviation industry and used spatial queries in his work. We had a great conversation about why I was chasing performance gains, especially since he felt that existing algorithms were fast enough. I found it insightful to step back and consider the “why” of my research. This experience would not have been possible without grants from the John Martinson Honors College (JMHC), and for that I am very grateful.

Final Thoughts

I thoroughly enjoyed working on this project over the past two semester. This was my first exposure to research, and it taught me a lot about the process of having an idea, testing an idea, and refining on those ideas. I would say that my biggest takeaway from all of this is the need to understand and verify everything. I would use ChatGPT for a lot of my code, but even if it worked, Durga would question my understanding of the code. She made sure that if I claimed to have written a certain piece of code, then I could defend every line of that

code. This is a skill that I have taken with me and have applied elsewhere. I'd like to thank her, my faculty advisor Professor Milind Kulkarni, and the JMHC for making this whole experience possible. It truly transformed my college experience.