Fast Gpu Point in Polyhedron Test

EECE course project 2018-2019

Ahmad Mustapha

***Abstract*—Point in Polyhedron (PiP) is an important problem that have applications in different domains. With the increasing requests for higher quality and lower latency real-time software applications, implementing software applications serially on CPU is no more suitable. Viewing the problem from this point of view we managed to implement a GPU based PiP application using CUDA framework and inspired by the paper in [1].**

***Keywords*—component, formatting, style, styling, insert (*key words*)**

I. Introduction

Point in polyhedron is an important problem that have a number of applications in different domains such as computer aided design (CAD), computer graphics, geographic information systems (GIS), etc. [1]

With the rise of GPGPU (General Purpose Graphics Processing Unit) many geometric algorithms have been redesigned and implemented to work on GPUs in order to achieve a better performance. However with respect to the point in polyhedron problem few research studies have tackled the application of GPUs on it.

Usually point in polyhedron is performed on a large set of points. Given that ray crossing is one of the most efficient way to tackle the problem. One can easily parallelize point queries such a way that each thread handles a ray crossing operation in parallel. However the time complexity of the query still of O (N) where N is the number of polyhedron faces. In [2] the authors proposed a decomposed tetrahedrons [3] based GPU implementation. However the method still undergo an O (N) complexity as each thread will have to visit all the tetrahedrons.

In the paper "Fast and robust GPU-based point-in-polyhedron determination" [1] the authors made use of preprocessing step to achieve a much faster - from time complexity perspective - inclusion test. In my project I am aiming to implement this method and then I will see - if I had time - how I can achieve a better approach either by optimization or by another approach.

II. Background

Research studies have tackled the problem of point in polygon problem (the 2D case) much more than the point in polyhedron (the 3D case) problem. However many of the algorithms that are used for polygons can be generalized for polyhedrons.

Four tests (other may exist) can be used to check whether a point is in a polygon (also generalizable for 3D) or not.

- Ray Crossing Test
- Winding Number Test
- Triangle Method
- Angle Summation

The first two are the most used ones (for they need less computations). In [1] the authors made use of the first method hence I will be using it without neglecting winding number if I had time.

Aside from the test used there is another concept when dealing with inclusion tests. The concept is whether to use:

- Global approach
- Local approach

The global approach is the direct application of the chosen test for point queries. The local approach divides the polyhedron into different local regions and for a given point the inclusion test is done in the regions that contain the point (say like indexing for structured data). Local approach requires a preprocessing step.

The division of the polyhedron in the local approach can be achieved using different data structures. Some used trees, other used layers, and others used uniform grids. The authors in [1] used a uniform grid.

III. Ray crossing

Ray crossing a well-known inclusion test. The idea is to create a ray from the query point and count the number of intersections with the polyhedron faces (or polygon edges). If the intersection number is even then the point is out of the polyhedron if it is odd then the point is inside (see figure 1). The test soundness is based on Jordan Curve Theorem.

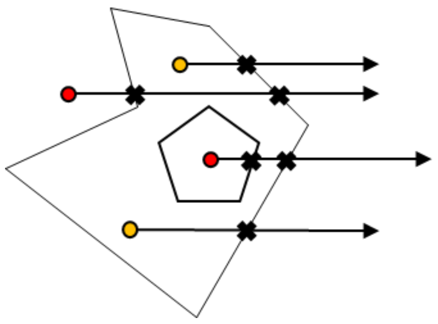

Fig. 1. Ray Casting. In polygon points are orange. Out polygon are red.

IV. Algorithm

In this section the algorithm proposed in [1] is explained. The explanation will be limited to the general procedures without mentioning tiny details as interested readers are advised to check the paper in [1]. Also illustrations of different steps will make use of polygons rather than polyhedrons for convenience without losing generality.

The proposed algorithm for point in polyhedron test can be divided into three steps.

1. Grid generation and initialization step
2. Preprocessing step
3. Point in polyhedron test step

A. Grid Generation

The first step in grid generation is choosing the grid dimensionality. The authors used the method in [4] to solve this step as described in the following:

$$N_x = d_x\sqrt[3]{\frac{\lambda N}{V}}, \qquad N_y = d_y\sqrt[3]{\frac{\lambda N}{V}}, \qquad N_z = d_z\sqrt[3]{\frac{\lambda N}{V}},$$

"Where dx, dy, and dz are the lengths of the grid's bounding box, V = dx$\times$dy$\times$dz, N is the number of faces, and λ is the density of the grid (determined by the user). In our implementation, λ was set to 8." [1] Figure 2 illustrates a grid over a polygon and a bounding box.

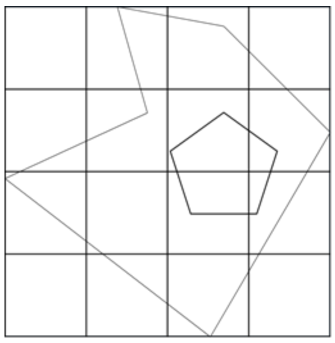

Fig. 2. A grid example

After choosing the dimensions the next step is assigning faces into grid cells. The authors considered again the proposal in [4]. This step is performed on GPU and by its end one can directly find out the faces that exists in a certain cell.

B. Prepocessing Step

The next step is the most important step in the preprocessing phase. In this step the inclusion state of cells center points are computed in parallel. Center points are set to be inside the polyhedron, outside the polyhedron, and singular. Singular points represents the points that fell on the polyhedron faces. Singular points can't be utilized during the inclusion test of the query points.

To find the state of center points a segment is created from a point outside the bounding box and the first center point. The segment intersection number with faces in the cell are counted. Based on the intersection number and as the segment first point is outside the polyhedron we can guess the state of the center points. Now that we have the inclusion state of the first center point we can apply the same methodology for the next center point by creating a segment from the first center point. And we keep on until finding all center points. Figure 3 illustrates this procedure. Note that for singular point its intersection is only counted for the points that follow it.

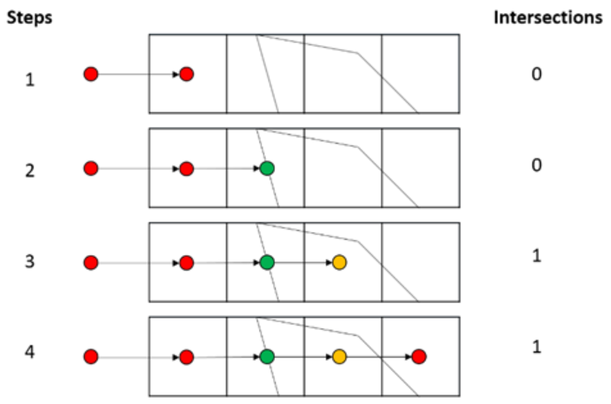

Fig. 3. A Center points inclusion state computation steps. Red are points out the polygon. Orange are inside. Green are singular.

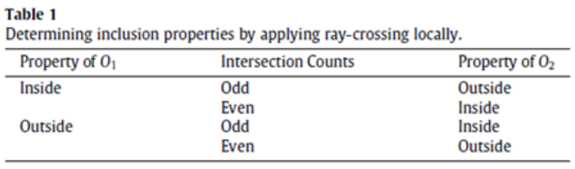

Table 1
Determining inclusion properties by applying ray-crossing locally.

Property of O_1	Intersection Counts	Property of O_2
Inside	Odd	Outside
	Even	Inside
Outside	Odd	Inside
	Even	Outside

Fig. 4. How to determine the inclusion state of a center point based on the previous point and the number of intersections.

Table 1 summarizes how to set the inclusion point of a certain point based on the previous point state and the number of intersection. This assignment is intuitive. Consider that we have a point O_1 outside the polyhedron and we want to figure the state of the next center point O_2. If the intersections of O_1O_2 are odd then it is intuitive to guess that O_2 is inside the polyhedron and outside otherwise.

Note that this procedure is done in parallel and on the GPU for each grid row parallel to x-axis (can be y or z axes based on user determination). This procedure and its parallelism might be a possible optimization region for me. As choosing a different parallelism approach might lead to a better performance. By the end of this step all the center points are assigned an inclusion state. As in the below figure:

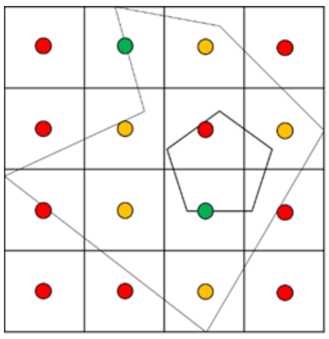

Fig. 5. The entire grid with the center points inclusion property assigned. Red points are out the polygon. Orange are in. Green are singular

C. *Point in polyhedron test step*

Now that we have found the inclusion property of each cell center point, we can start accepting point queries and found them in very low time complexity.

When a point – say P is queried whether it is inside or outside a polyhedron we first find the cell that contains the point. Then we create the segment PO_C where O_c is the central point of the containing cell. We then apply the same reasoning we applied to find the inclusion properties of central point. We count the number of intersection of PO_c with the polyhedron faces that are assigned to this cell. And according to the number of intersections and because we already know the inclusion property of the O_c we can guess the inclusion property of P.

Note that if the central point of the cell containing the query point is central we search for the first non-singular central point to apply the inclusion test reasoning on it. And it is because for singular central points no matter what is the number of the intersections of the segment between it and the query point we can't guess the nature of the query point it might be inside or outside.

Let M be the total number of cells in the generated grid. Let N be the total number of faces in the polyhedron. Let r be the average number of cells a polyhedron face overlap (a single polyhedral face might overlap multiple cells). The in average we have rN/M faces in a given cell. So In the best case scenario our method time complexity is O (rN/M) as for each point we are only counting its intersection with the faces that are assigned to the containing cell.

The following figure illustrates a query point inclusion test based on a prior knowledge represented by the inclusion properties of central points.

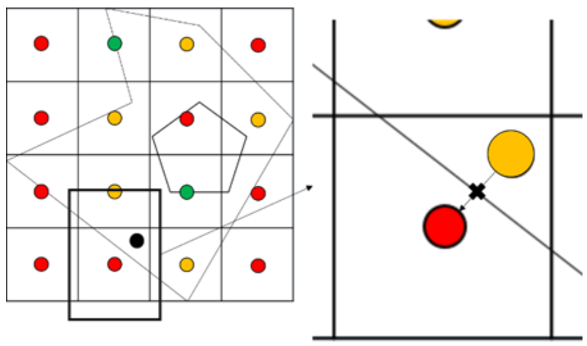

Fig. 6. Inclusion test for a query point. Black query point. Red out polygon point. Orange in polygon point. Green Singular Point. In the enlarged image we see how we figure that the query point is inside the polygon based on ray crossing.

V. IMPLEMENTATION RESULTS

We implemented the aforementioned algorithm on CUDA framework on a computer with an NVIDIA GTX 1080 GPU. In order to make sense of the implementation performance we implemented also a baseline counterpart. The base line does not include a preprocessing step. A CUDA block is launched for every point (or for multiple points if the points are more than the GPU maximum capacity to launch blocks). In each block each thread will be responsible to collect the intersection value of the block corresponding point with multiple triangle in a way that a block visits the entire triangles in a polyhedron.

We run a set of trials and we recorded the time required to obtain the inclusion state of a different number of points by each algorithm (grid based approach vs baseline). For each type we recorded execution time for 100, 1000, 10,000, 100,000, and 1,000,000 points against the same polyhedron. The polyhedron used is the Stanford-dragon 3d model. Note that for the baseline the GPU resources couldn't handle the task and fired an "unknown launch failure kernel error". Also for the grid based approach we not only varied the number of points but also the grid density. Grid density – as aforementioned – is a parameter controlled by the user for the grid generation algorithm. The higher the density the lesser (and larger in terms of volume) the number of cells. In other words by increasing the density we will be decreasing the number of the launched threads during preprocessing step but increasing the computation (as more triangles will be overlapping the cell) on each thread during point in polyhedron testing. We recorded execution time for 2, 4, 6, and 8 grid density values. The below figure shows visualize the results.

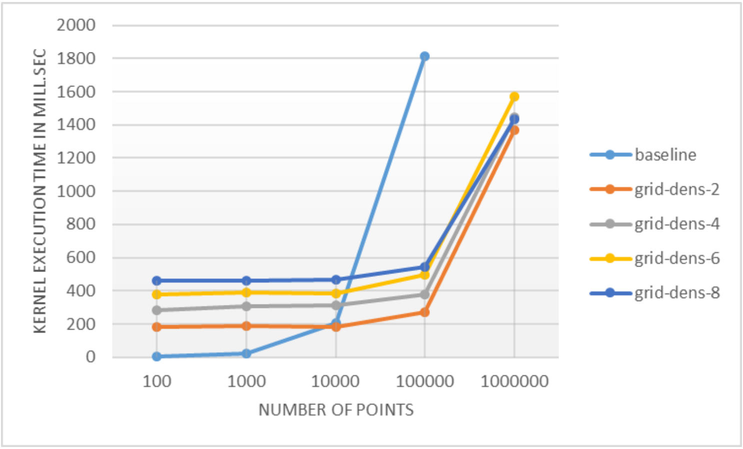

Fig. 7. The rate of change of kernel execution time as we increase the number of points to be tested for five different kernels. X-axis shows the number of points. Y-axis shows the kernel execution time in milli seconds. Kernels are 1 baseline kernel and 4 grid based pip kernels with varying grid density.

From figure 7 we can see that when the number of points is relatively small the baseline approach performs better however as the number of points increase the baseline undergoes an exponentially increase in execution time while the grid based approach maintains a linear increase. Note that the number of points scale is logarithmic.

With respect to the effect increasing grid density on execution time. It seemed that for the polyhedron under test it is better to have large number of small cells with little computations than to have small number of large cells with many computations. However as the number of points increases the effect of grid density becomes negligible.

VI. Optimization Possibilities

From algorithmic point of view it seems that optimization efforts will have a small return in gain. One possible different approach is that during the preprocess step rather than launching threads equal to the cross of two axis, one can launch threads equal to the number of cells and each thread will stay idle until one of its neighbors have computed it's cell center point inclusion state. In this case all the cells on the borders will be computed in the first iteration compared to only an axis slice in the parallel path based kernel.

From implementation view the kernel we implemented is far from optimized. All the computations included in the grid generation and in the preprocessing step store and accessed global memory. Using texture memory will give the kernel a huge performance increase. Also using CUDA streams is another direction for optimization.

I. Conclusion

In this short manuscript we presented a fast parallel point in polyhedron algorithm. We then showed that such an algorithm even if not optimized outperformed the proposed optimized baseline. We finally mentioned some optimization directions from both implementation and algorithmic points of view.

References

[1] L. Jing and W. Wencheng, "Fast and robust GPU-based point-in-polyhedron determination," *Computer Aided Design,* 2017.

[2] R. R. J. Antonio and O. M. Lidia, "Geometric Algorithms on CUDA," in *GRAPP*, Funchal, Madeira - Portugal , 2008.

[3] F. Feito and J. Torres, "Inclusion test for general polyhedra," *Computers & Graphics,* vol. 21, no. 1, pp. 23-30, 1997.

[4] T. I. M. Eugene and P. N. Sumanta, "Macro 64-regions for uniform grids on GPU," *Vis Comput,* vol. 30, pp. 615-624, 2014.