

Fast and robust GPU-based point-in-polyhedron determination



Jing Li ^{a,b,*}, Wencheng Wang ^{b,c}

^a State Key Laboratory of Integrated Information System Technology, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China

^b State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China

^c The University of Chinese Academy of Sciences, Beijing 100049, China

ARTICLE INFO

Article history:

Received 12 August 2015

Accepted 12 February 2017

Keywords:

Point-in-polyhedron test

GPU

3D uniform grids

Ray-crossing

ABSTRACT

This paper presents a fast and robust GPU-based point-in-polyhedron determination method. The method partitions the bounding box of the polyhedron into a grid with $O(N)$ cells, where N is the number of polyhedron faces, and predetermines the inclusion property of the grid cells' center points. Then, a line segment is generated from the query point to the center point of its related grid cell to determine the inclusion property of the query point by counting the faces intersected by the line segment. Such a localization treatment is further exploited to optimize the visiting pattern in using GPUs, by which a high increase in the testing speed is achieved. We also provide a unified solution to all singular cases, especially those caused by localization, to guarantee the efficiency and robustness for point-in-polyhedron tests. The results show that our proposed method can be faster than both state-of-the-art serial and parallel methods by several orders of magnitude, with only a slight increase in storage requirement.

© 2017 Elsevier Ltd. All rights reserved.

1. Introduction

Point-in-polyhedron determination is a fundamental problem in computational geometry. It is widely used in many fields like computer-aided design (CAD), computer graphics, and geographic information systems (GIS) to support animation, realistic rendering, collision detection, and point location in 3D GIS. Obviously, its efficient implementation is important to high-level algorithms. Currently, in various applications, the amount of query points is rapidly increasing, and an increasingly higher processing speed is required, with demands to be met even in real time, and the models may change their shapes frequently. For example, more and more points are needed to produce more realistic effects in games, and the amount of mobile equipment with location and navigation sensors is rapidly increasing. Thus, this brings a huge challenge to fast execute point-in-polyhedron determination. In this paper, we present a method to execute these inclusion tests using GPUs, in an attempt to address this challenge.

To our knowledge, GPUs have been extensively applied to accelerate various basic geometrical algorithms. Unfortunately, only few research studies are known to apply GPUs to the point-in-polyhedron test. Normally, ray-crossing can be easily implemented on a GPU by assigning a thread to each query point for its determination in parallel. However, the time complexity for a query

is still $O(N)$, where N is the number of polyhedron faces. When the number of points is far greater than the number of processing units of a GPU, the running time is still very high. Though there are methods with lower time complexities (e.g., $O(\log N)$), they are typically based on a structure including hierarchical trees to decrease the time complexity, which are generally inconvenient to implement on a GPU.

In our method, we attempt to confine the point-in-polyhedron determination within a local region, and locally apply the ray-crossing method. This is by constructing a uniform grid with $O(N)$ grid cells according to the bounding box of the polyhedron, and predetermining the inclusion property of the cells' center points as priors for point-in-polyhedron determination, which means to determine the cells' center points as inside or outside of the polyhedron before answering query points. Thus, for a query point, we only need to process the faces in the grid cell containing the query point, to count the faces that intersect with the line from the query point to the center point of the cell, by which ray-crossing is applied locally, which is discussed in detail in Section 3. Thus, we can reduce the time complexity by avoiding multiple ray-face tests and conveniently use the GPU for point-in-polyhedron determination. Furthermore, we develop novel measures to considerably reduce the time for constructing auxiliary structures and thereby speed up point-in-polyhedron determination. Thus, we can well treat the applications that use a large number of points and exhibit dynamic changes in polyhedron geometry. Experimental results show that our method needs only 8 ms to determine 1M points against a gear box model with 165K faces from [1], whereas the

* Corresponding author at: State Key Laboratory of Integrated Information System Technology, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China.

E-mail addresses: lijing2015@iscas.ac.cn, superredlark@163.com (J. Li).

parallel global ray-crossing method takes 100 s. This is a speed up of over five magnitudes. Fig. 1 shows an example of determining 1M points against a deforming model in real time.

The rest of this paper is organized as follows. In Section 2, we discuss related works. We describe our method using GPUs in Section 3 and analyze our method in Section 4. Then, we compare our method with some state-of-the-art methods in Section 5, and finally, conclude the paper in Section 6.

2. Related works

Compared with the works on 2D point-in-polygon determination, there are fewer works on 3D point-in-polyhedron tests. These works report either computing some parameters globally or localizing point-in-polyhedron tests using certain auxiliary structures, and are discussed in the following paragraphs.

The methods computing parameters globally are typically simple to implement [2–6]. For example, ray-crossing is the fastest one of this type [7]; it generates a ray from the query point in a certain direction to intersect the faces of the polyhedron. By counting the intersections, it is known the query point is inside/outside the polyhedron if the count number is odd/even. However, similar to the other methods of this type, ray-crossing needs checking against every face for a query, in a time complexity $O(N)$. Evidently, the computations are too slow when applied to a large number of query points, even if they are accelerated using a GPU (because a GPU has a limited number of processing units).

To reduce the time complexity for acceleration, researchers proposed several methods to localize point-in-polyhedron tests using auxiliary structures, by either reorganizing the polyhedron elements in layers or constructing a spatial partition structure.

Layer-based representation of 3D models has been studied extensively. In this method, the elements of the polyhedron are sorted orderly, which is helpful to speed up the search of the most related elements for operations. For example, layered depth images are used for solid modeling by decomposing the model into stripes by their depths to improve Boolean operations between models [8], volumetric intersections of deforming objects [9], collision detection between models [10], and so on. In particular, several specific layer structures have been presented for point-in-polyhedron tests [11–13]. Among them, the method reported in [11] is easy and fast to construct and use layer representation because it employs orthogonal projection to obtain the occlusion between faces for classifying them in layers, unlike the others to employ perspective projection to sort the decomposed tetrahedrons from the polyhedron into layers. These methods can facilitate the use of the binary search for speeding up inclusion tests, with a reduction in time complexity between $O(N)$ and $O(\log N)$. However, their preprocessing is not easy, and their complicated layer structures and binary search are difficult to implement on the GPU, which prevents them from further acceleration and use for models with frequent change in their shapes.

The methods using spatial partition structures construct either hierarchical structures or grids for performing inclusion tests locally. Commonly used hierarchical structures include BSP-tree [14], octree [1], quadtree [15], and kd-tree [16]. Although face-aligned BSP is very efficient in terms of determination [14], its long preprocessing time and large memory requirement hamper practical applications. Using an octree, Liu et al. [1] found the triangle that is relatively closest to the query point, called the *relative closest triangle* (RCT), and determined the point using its relative orientation to the RCT. However, although the computations are confined to a limited number of nodes, these methods need to search in different tree levels. Such vertical traverses in the octree weaken the gain from localization computation, and increase computation cost very much when the visited tree nodes

differ much in their levels. This seriously affects the query speed. Moreover, tree traversal typically uses stacks and involves many branches. This does not suit the GPU architecture and is detrimental to implementation.

The grid is a popular structure for acceleration. Haines [17] proposed a 2D method that combines a uniform grid and ray-crossing. It first predetermines the corner points of grid cells, to know their inclusion properties (inside or outside) in advance. Then, the inclusion test is based on these predetermined corner points to run ray-crossing locally. This method is very fast because computations are largely localized. However, it has not been extensively studied, and its 3D extension has not been proposed. Its potential in terms of parallel computing (aided by localized computations) has not been exploited. Ooms et al. [15] proposed another method to use a uniform grid, which is an extension of the 2D-CBCA algorithm in [18] and [19]. It preclassifies grid cells as being completely inside, outside or on the boundary of the polyhedron, and determines a point according to the status of the cell containing the point. If the cell is completely inside or outside the polyhedron, the point has the same property as the cell. Otherwise, the point is determined by employing ray-crossing. To speed up preprocessing and inclusion tests, this method manages faces using an extra 3D quad-tree. However, it must test all the leaves that a ray passes when determining ray-face intersections, which lowers the degree of localization. According to the experimental results in [16], its accurate determination is too slow (no statistics were reported). Currently, voxelization of solid models has been studied extensively [20,21], which can be used for point-in-polyhedron tests. By positioning the voxel containing the query point, the inclusion property of the point can be obtained instantly because the voxel is either in the polyhedron or outside the polyhedron. Though this method is shown to be very effective for applications by voxelizing the model at the resolution of 1024^3 cells, it is still an approximate approach, and its high storage requirement prevents its use for large polyhedrons, which are being increasingly used in various applications.

Few researchers have focused on exact point-in-polyhedron tests based on a GPU. Rueda et al. [22] implemented a method based on the method involving decomposed tetrahedrons [6] on a GPU. However, at the thread level, this method must still visit all the tetrahedrons and has a $O(N)$ query time complexity.

In comparison with the existing methods, our proposed method is based on a grid structure. It seems similar to the method in [17]. However, we extend it to treat 3D inclusion tests, where more singular cases need to treat, and we provide a uniform solution to them, unlike the method in [17] to take a heuristic strategy to avoid singular cases, which may be very trouble. In addition, we predetermine the center points of grid cells, not corner points of grids cells. Thus, in answering a query point, we only need to check the faces in the cell containing the query point, not the faces in the cells sharing the related corner point. This can save much computation. Moreover, we develop measures to quickly determine the center points of grid cells, and make an effective implementation on GPUs, which are not studied in [17].

3. Our proposed method

As discussed in Section 1, our method tries to perform inclusion tests by applying ray-crossing locally. Considering efficiency, we adopt the grid instead of hierarchical trees as our spatial partition structure because it is easy to construct, fast to position the cell containing the query point, and convenient to implement on GPUs. In the following subsections, we will describe our method in detail, where our GPU implementation is introduced correspondingly.

Before discussing the details of our method, we first clarify the premise. In this paper, we assume that a polyhedron is represented



Fig. 1. Real-time collection of points inside a dynamic hand model, where 1,000,000 falling points are determined. The points fall down from the top of the model and accumulate within the model's bounding box (a). The hand model has 15K faces (b). The points outside the model are set to be transparent, and only the points inside the model are colored (c)–(e). The process is handled at a speed of 40 fps on the average, with a screen resolution of 512×512 pixels on a desktop PC equipped with a NVIDIA Quadro 6000 GPU. (The frame rate in the attached video was affected by the screen capture software; hence, it is approximately 30 fps. <http://tappole.com/3DGCAP.avi>.)

by a triangle mesh, which is one of the most common boundary representations used in practice. However, our method is not confined to a triangle mesh. It can be easily generalized to a polygon mesh. We also assume that a polyhedron is manifold and closed or watertight. Any non-manifold or unclosed model should be fixed before processing.

3.1. Constructing a uniform grid

In constructing a grid structure for a polyhedron, we first obtain the bounding box of the polyhedron and decide the grid resolution, and then sort the faces of the polyhedron into grid cells.

In generating the bounding box of the polyhedron, we inspect all vertices of the polyhedron to compute the point with the minimum x , y , and z coordinates and the point with the maximum x , y , and z coordinates to define the minimum axis aligned bounding box of the polyhedron. To avoid numerical errors, we expand the tightly enclosed bounding box outward a little to set the grid bounding box. Then, we adopt the strategy in [23] to obtain the grid resolution as described in the following:

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

where d_x , d_y , and d_z are the lengths of the grid's bounding box, $V = d_x \times d_y \times d_z$, 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.

In assigning the faces into grid cells, we follow the method reported in [24] that first finds face–cell pairs, where a face may exist in multiple pairs as it may overlap multiple cells, then sorts these pairs in parallel using radix sort according to the cell index, and finally extracts the faces that are overlapped by each cell. Considering efficiency, we adopt the method reported in [23] to replace the sorting step by atomic operations on the GPU. A brief pseudo code is as follows (see [23] for details).

Algorithm 1. Uniform grid construction.

```

int res[3] = calculate the grid resolution
int nCellCount = res[0] * res[1] * res[2]
int nPairCount = Calculate the total number of face–cell pairs
intArray triID[nPairCount]; //the triangle list for face–cell pairs
intArray cellID[nPairCount]; //the cell list for face–cell pairs
1: Compute coarse cell–triangle pairs and populate triID in parallel
2: Compute the exact cell–triangle pairs and populate cellID in parallel
intArray triList[nPairCount]; //triangle list ordered by cell ID
intArray gridStart[nCellCount]; // the starting address in triList for each cell
intArray gridEnd[nPairCount]; // the ending address in triList for each cell
3: Extract grid structure (triList, gridStart, gridEnd) from cell–triangle pair (triID, cellID) in
parallel using atomic operations

```

In the implementation, we load the whole geometrical information of both vertices and faces from CPU to GPU at the beginning, and the subsequent processions are executed totally on the GPU to avoid frequent data transmission between CPU and GPU, a serious problem in using GPUs. As our method requests moderate storage cost, such a loading strategy is feasible, as shown by both analysis and experiment results in subsequent sections. Steps 1, 2, and 3 in the algorithm are realized by separate CUDA functions.

3.2. Predetermining center points of grid cells

In determining the center points, we also apply the ray-crossing method locally. For two neighboring center points, say O_1 and O_2 , we connect them with a line segment. If the inclusion property of O_1 is known, we can derive the inclusion property of O_2 by counting the intersections between the line segment and the faces of the polyhedron, as listed in Table 1. If O_2 overlaps a vertex, an edge or a face of the polygon, it is not sure if it is inside or outside the polyhedron. In this case, we mark O_2 as a singular one.

According to the above discussion, we can connect neighboring center points one by one with the line segments parallel to the axes, to form traversal paths, and extend every path to a point outside the grid-bounding box. As the point outside the grid-bounding box is surely outside the polyhedron, we can determine the center points on the path by counting the intersections between the path and faces of the polyhedron. An example is illustrated in Fig. 2, where only a path is formed to connect all center points of the grid with $4 \times 3 \times 3$ cells, and its three slices are displayed separately to show the traverse path clearly. In the path, point O is outside the polyhedron.

For using parallel computation for high speed, we do not form complicated paths as shown in Fig. 2. We just form paths parallel to the axes, e.g. the paths from the left to the right parallel to the x axis (which will be used to describe our method in the following, without loss of generality), in our implementation. In this way, many paths can be generated to run in parallel, and every path can be treated easily.

Starting from the outside point of a path, its center points are determined one by one from left to right along the path, supposed that the path is parallel to the x axis. At first, we compute the count of intersections for the line segments between current center point and its preceding center point (the outside point precedes the first center point) on the path, say S_i , $i = 1, 2, \dots$, which is the sum of intersection numbers between the line segment and its intersected faces, to be discussed in Section 3.2.1. As the line segments are all parallel to the x -axis, the intersection numbers for such a line segment can be easily obtained by only checking the faces in the two cells containing the line segment. Then, with accumulation computation, we can obtain the count of intersections from the i th center point in the path to the outside point, say S_i , in the following equations.

$$S_i = \sum_{j=1}^i S_j \quad (1)$$

$$S_j = \sum_{k=1}^{T(j)} a_k. \quad (2)$$

In Eq. (2), $T(j)$ is the number of faces in the two cells containing the j th line segment of the path and a_k is the intersection number between the segment and the k th face in the two cells.

When S_i is obtained, the inclusion property of the center point can be determined. Though it may lie on the boundary of a polyhedron to be singular, this would cause no problem to compute

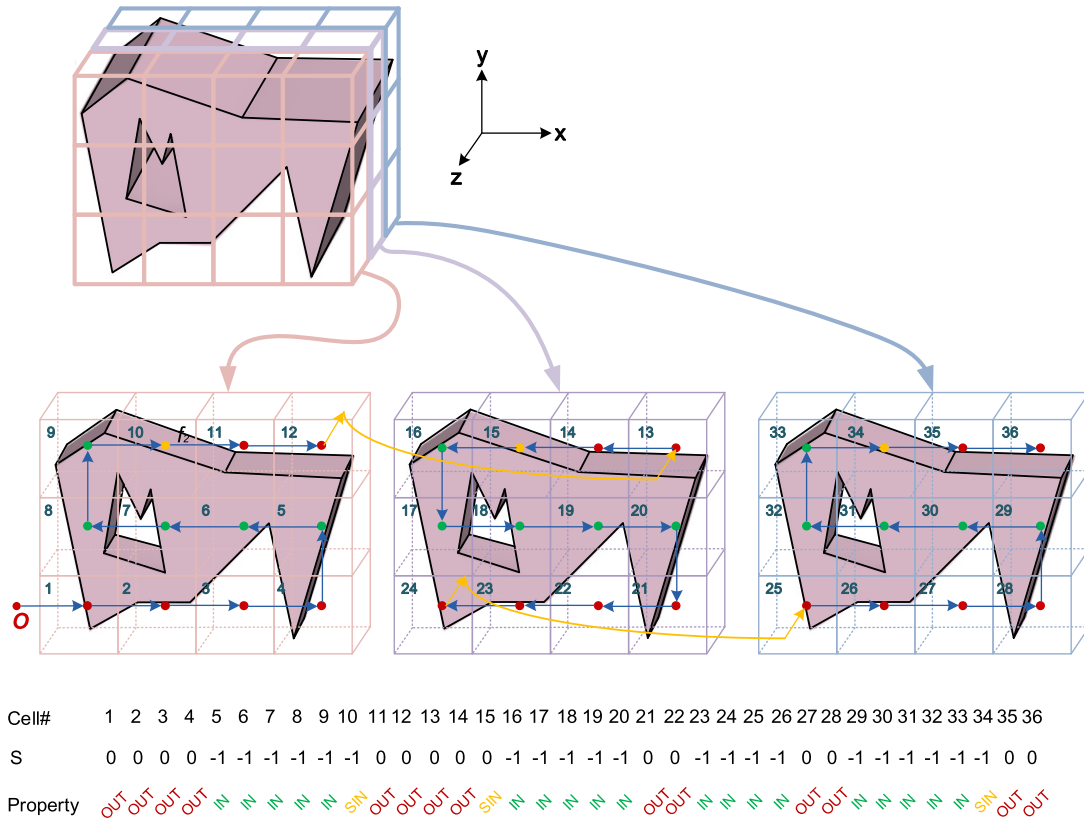


Fig. 2. Center points are determined by the path connecting them. Inside, outside, and singular center points are denoted by green, red, and orange points and marked IN, OUT and SIN in the Property list, respectively. S is a parameter for center point determination, explained by Eq. (1). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

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

intersections, as we adopt the method in [25] to correctly compute intersections, even in singular cases. This is discussed in detail in the following.

3.2.1. Computation of intersection numbers

To obtain S_i , it is necessary to compute a_k . However, compared with the 2D intersection computation, there are more singular cases for a line segment to intersect a 3D face. Therefore, instead of simply setting $a_k = 0$ or $a_k = 1$ to indicate whether there is an intersection, we use the method in [25] to count the intersections, where singularities are well considered. In the ray-crossing method in [25], a ray emitted from a query point is imaged to be a very thin cylinder. For each face, a_k is assigned a value to indicate the extent by which the ray enters ($a_k < 0$) or exits ($a_k > 0$) the polyhedron at this face. a_k is set to the fraction of the cross-section area of the thin cylinder which gets into or out of the polyhedron. In this way, singular cases are handled in a uniform way by using fraction numbers in counting the intersections.

Therefore, in Eq. (2), the value of a_k for a line segment O_1O_2 with a face f_k is computed as follows. Suppose that the vector of O_1O_2 is u , and the outer normal of f_k is v , then we compute a_k by the following four cases, as illustrated in Fig. 3,

- (1) If O_1O_2 is parallel to or overlaps the plane containing f_k , including the case that O_1O_2 passes through a vertex or edge while being co-planar, then $a_k = 0$. See Fig. 3(a).
- (2) If O_1O_2 intersects with the plane containing f_k and the intersection point x lies in the interior of f_k , then $a_k = \text{sign}(u \cdot v)$. See Fig. 3(b).
- (3) If x overlaps an edge of f_k , $a_k = 0.5 \text{ sign}(u \cdot v)$. See Fig. 3(c).
- (4) If x overlaps a vertex of f_k , $a_k = (\alpha/2\pi) \text{ sign}(u \cdot v)$, where α is the internal angle of the projection of f_k on γ at x , where γ is the plane passing through x and perpendicular to u . See Fig. 3(d).

Here, a special case needs attention. That is, the center point overlaps a polyhedron's face. For such a case, the value of S for this center point can be still calculated normally. In addition, we mark this center point singular, by which we will take a special treatment for inclusion tests. Because such a center point may be connected to generate two line segments, with its front and behind center point respectively, its related intersection number should be avoided to accumulate repeatedly in computing S . In our implementation, if such a singular case occurs, the number of intersections is only accumulated to S for the center points after the singular center point.

As illustrated in Fig. 3, an intersection point may relate to one or more faces. In [25], for an intersection point, the sum of a of these related faces can only be 1, -1 or 0 because in a watertight polyhedron, the cylinder of the emitted ray covers $\pm 100\%$ area of the cross section or 0 in the case of tangency. According to Eqs. (1) and (2), the value of S_i should only be 1, -1 or 0 too. However, in 3D cases, there exists an exception that a can be any fraction number between 0 and 1 (or -1), as shown in Fig. 3(e). Here, face f_1 is perpendicular to both faces f_2 and f_3 and these three faces intersect at x_1 , O_1O_2 passes through x and overlaps the intersection line of f_2

and f_3 . In this case, the cross section of O_1O_2 at x_1 does not form a complete circle, which induces a fraction number.

Fortunately, this will cause no problem to our method in treating manifold and closed polyhedrons by the following discussion. For an intersection with a fraction number (e.g. x_1 in Fig. 3(e)), it will be followed by many singular intersection points, which are all on some faces connected together, until meeting a singular intersection point for the path to leave these connected faces. With the singular intersection points processed, the summed cross section of the path at these singular intersection points either returns to zero or forms a complete circle, so that these intersection numbers are accumulatively summed either to 0 (Fig. 3(e)), meaning the path tangent with the polyhedron, or $-1/1$ (Fig. 3(f)) for entering/exiting the polyhedron. As the singular intersection points are marked singular, and the sum of the intersection numbers for a group of continuous singular intersection points is 0, 1 or -1 , the nonsingular center points can be correctly determined, guaranteeing the robustness of our method.

3.2.2. GPU implementation

We realize the process of center point determination by a CUDA function. Its pseudo-code is listed in Algorithm 2. In our implementation, traverse paths are set parallel to the x-axis, and the cells with same y and z index are assigned a thread. As the number of cells traversed for each thread (e.g. $res[0]$) is not very high, we use normal serial additions to compute $S[k]$ instead of parallel prefix sum operations, which have relatively high cost for elements in a small number.

Algorithm 2. Center point determination.

```

intArray centerProp[nCellCount]
for each  $c \in [1, res[1] \times res[2]]$  in parallel
    int  $i$  = current index in y direction for  $c$ 
    int  $j$  = current index in z direction for  $c$ 
    floatArray  $S[res[0]]$ 
    boolArray  $singular[res[0]]$ 
     $S[1]$  = the number of intersections between an out point and  $O_{1,i,j}$ 
     $singular[1] \leftarrow$  determine if  $O_{1,i,j}$  overlaps a face
    for  $k=2$  to  $res[0]$ 
         $s$  = the number of intersections for the segment between  $O_{k-1,i,j}$  and  $O_{k,i,j}$ 
         $S[k] = S[k-1] + s$ ;
         $singular[k] \leftarrow$  determine if  $O_{k,i,j}$  overlaps a face
    endfor
    for  $k=1$  to  $res[0]$ 
         $centerProp[k][i][j] = (singular[k] == \text{true}) ? \text{SIN} : (S[k] == 0) ? \text{OUT} : \text{IN}$ ;
    endfor
Endfpar

```

3.3. Point-in-polyhedron tests

With the constructed grid for the polyhedron, we classify a query point in the following steps. First, the grid cell containing the query point is decided quickly by the coordinates of the point and the size of grid cells. Then, the center point of the cell is connected to the query point to form a line segment. By counting the intersections between the line segment and the faces in the cell, using the measures described in Section 3.2, the inclusion property of the query point can be determined by the rules in Table 1.

According to the discussion in Section 3.2, the center point of the cell containing the query point may be singular. With regard to this, we abandon it and search the neighboring cells gradually from near to far until we find a nonsingular center point. Then, we generate a line segment from the query point to the found nonsingular center point, and perform the inclusion test for the query point as described in the above paragraph, where the faces

in the cells containing the line segment should be all checked. By algorithm analysis in Section 4, we do not need to search many cells (typically no more than two) to obtain a nonsingular center point. Thus, our method can run in a high degree of localization. Note that the special case in Fig. 3(e) may also occur for the line segment between a query point and a center point, making the number of intersection be a fraction. With similar reasoning, such a case can be solved by substituting the current center point with a nonsingular one to avoid the fraction value.

In applications, it is always required to answer many query points. Considering efficiency, we use two ways to distribute the work load based on the number of query points. If there are very few query points, the local faces are grouped and distributed evenly to each thread. Intersection results are recorded for each query point and summed in parallel to produce the final results for them. If there are many query points, a thread is assigned to each query point and responsible for all its intersection tests. In this paper, we focus on the situation of many query points, so we use the second way. Before starting the test, we bind the polyhedron's geometrical information and grid structure to textures to accelerate the access speed by the cache scheme of texture memory. This is benefited from our low-storage requirement, to be discussed in Sections 4 and 5, to effectively use GPUs.

Fig. 4 illustrates our treatment for inclusion tests. For query point Q_0 , test segment Q_0O_{13} crosses faces f_0 and f_1 . The total number of intersections is $1 + (-1) = 0$; thus, Q_0 has the same property as O_{13} . For Q_1 , because O_{21} is singular, O_{20} is selected to generate a test segment. Q_1O_{20} crosses cells 20 and 21. It only intersects face f_2 ; thus, the number of intersections is 1. Therefore, Q_1 and O_{20} have different properties, and Q_1 is thus inside the polyhedron.

4. Algorithm analysis

4.1. Probability for singular center points

The probability that a center point is singular equals the probability that it overlaps a face, denoted by δ . In practice, we typically decide whether a point overlaps a face by determining if the distance from the point to the face is less than a threshold, ε . If a center point has a distance less than ε away from a face, it is a singular center point. For a face, such a region is the volume expanded from the face by ε . Therefore, δ is the volume ratio of the sum of the expanded regions of all faces against the volume of the grid. Because ε is very small, δ is typically also very small. Suppose that the expected number of singular center points before finding a nonsingular center point is N_c , then N_c is 2 at most, because δ is typically not greater than 0.5.

Although very rare, there exists an extreme case that many center points overlap faces. In this case, we use the method proposed in [17] for improvement, which reconstructs the grid by modifying the resolution or adjusting the size of the grid bounding box. Fortunately, we have not encountered such a case in our tests.

4.2. Time and space complexities

Suppose that the total number of grid cells is M , and a face intersects with an average of r cells. Then, a cell contains an average of rN/M faces. Using our algorithm, we need $O(M)$ space to store the property of center points. Considering the space for basic geometrical information ($O(N)$) and the space for references to faces ($O(rN)$), the complexity of the total space requirement is $O(M) + O(N) + O(rN)$. Because $r > 1$, this is equal to $O(M + rN)$.

Our preprocessing stage consists of two parts: creating the uniform grid and predetermining the center points. In the first

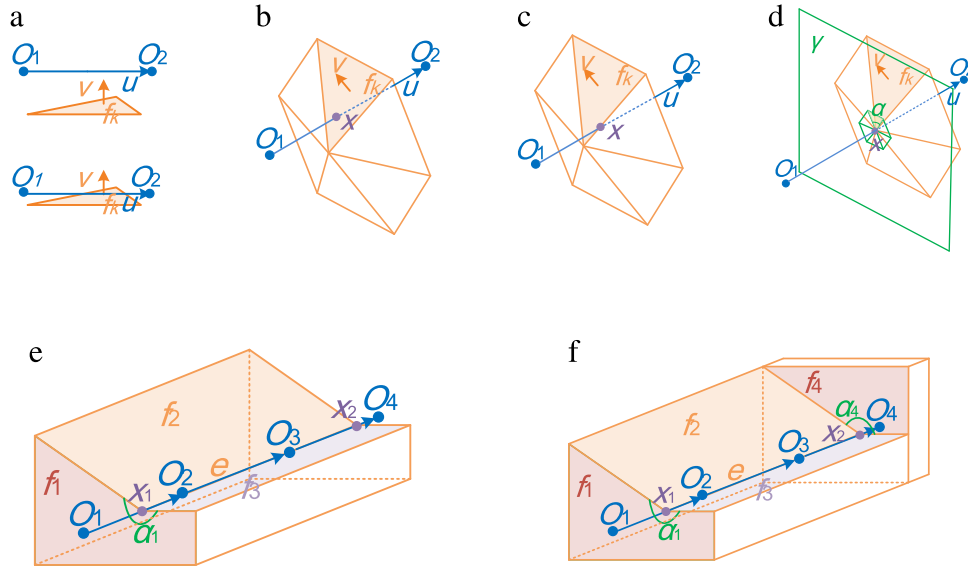


Fig. 3. The four cases that arise in the intersection tests (a)–(d). (e) and (f) show a special case wherein the intersection number for an intersection point is a fraction number.

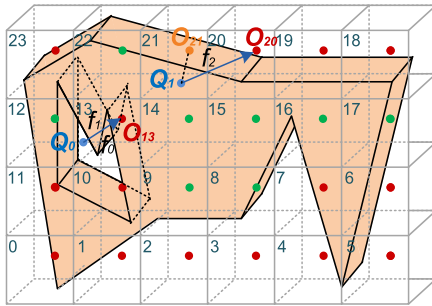


Fig. 4. Point-in-polyhedron determination using predetermined center points.

stage, according to [23], the time complexity for creating a uniform grid is $O(M + rN)$. In the second, the faces in a cell are visited two times at most, one for determining the current center point, and the other for determining its adjacent center point. Therefore, all faces are visited $2r$ times, leading to $O(rN)$ time for predetermining center points. Thus, the total time complexity for preprocessing is $O(M + rN)$.

In a point-in-polyhedron test, the algorithm must find a nonsingular center point near the query point. The above analysis implies that the number of cells to be searched is typically not more than 2. Thus, the expected time complexity for a query is $O(rN/M)$ because each cell contains an average of $O(rN/M)$ faces. For the extreme case that a cell contains N faces, e.g., all faces of a cone are linked to the apex, our algorithm still needs $O(N)$ time for answering a query point in the cell.

In summary, our algorithm needs $O(M + rN)$ space, $O(M + rN)$ preprocessing time, and $O(rN/M)$ time for a query in treating ordinary polyhedrons without their faces assembled in a small region. When $M = O(N)$, the three complexities are $O(rN)$, $O(rN)$, and $O(r)$, respectively. Although r is $O(N)$ at worst (for extreme cases), r is typically $O(1)$ in practical applications, which we tested using experiments. Thus, the three complexities would be $O(N)$, $O(N)$, and $O(1)$ for practical applications, which are shown by our experimental results. In comparison with other methods based on grids, our method has the same complexities with them. As we use highly efficient local operations, it is much faster than the others, even in $O(1)$ time, as shown in Section 5.

5. Results and discussion

To test the performance of our method, we ran various experiments on a desktop computer configured with 3.1 GHz Intel (R) Xeon (R) CPU, 16 GB RAM, and a NVIDIA Quadro 6000 GPU. Because our method is based on grid center points, we call our method 3D-GCP in the following. We selected six models from [1], which are either typical CAD models or models often tested in literatures. Table 2 lists these models and shows the grid resolutions used with our method. In all tests, we generated 1,000,000 random query points within the minimum bounding box of each model.

In our current implementation, the faces in the cells are checked individually to obtain the intersections for ray-crossing. As discussed in [26,27], such computation can be accelerated by a layer peeling technique. As this needs extra storage and processing time for the layer structure, and the faces in a cell are always in a small number, we do not adopt this technique now. In the future, when integrating our method with solid modeling, we would adopt this technique for efficient implementation.

The size of the thread block is an important factor for performance when using a GPU. Therefore, we first ran tests using 3D-GCP with various block sizes and different models to determine the suitable sizes of the thread block for various tasks in our method. Our results suggested that the algorithm performs best when the block sizes are 256, 64, and 128 threads for creating the uniform grid, setting the center points, and running the point-in-polyhedron tests, respectively. The results for 3D-GCP in the following tests were obtained under this configuration.

5.1. Comparison with other parallel methods

We first compared 3D-GCP with our implementation of three parallel algorithms on a GPU. The first one was based on decomposed tetrahedrons (denoted by Tetra) [22]. It is the only published GPU-based exact method we can find for point-in-polyhedron tests. The second was a parallel implementation of classical ray-crossing method (denoted by RC). Tetra and RC are typical $O(N)$ methods. The third was a parallel method that simply combines ray-crossing and a uniform grid, denoted by URC. For an inclusion test, it emits a ray from the query point along an axis to intersect the faces in the cells that the ray traverses through until the ray exits the grid. Note that the only difference between URC and

Table 2

Test models and grid resolutions used by 3D-GCP.







						
Name	Bunny	Dragon	Horse	Gear box	Pump shell	Drill handle
Vertex No#	5664	30342	138352	82556	26493	32241
Face No#	11316	60680	276680	165140	53106	64542
Grid resolution	$48 \times 48 \times 37$	$124 \times 115 \times 33$	$191 \times 135 \times 85$	$224 \times 95 \times 61$	$60 \times 76 \times 91$	$298 \times 41 \times 41$

Table 3

Performance comparison between 3D-GCP and three existing methods with parallel computation on a GPU, to answer 1M query points.

	Tetra		RC		URC		3D-GCP		
	\hat{T}_q	$\#I$	\hat{T}_q	$\#I$	\hat{T}_q	$\#I$	\hat{T}_q	$\#I$	sT_t
Bunny	18921	11316	9800	11316	895	6.9	7	0.7	15
Horse	94590	60680	52645	60680	876	5.6	8	0.5	26
Dragon	430512	276680	245360	276680	4468	14.2	8	0.4	61
Gear box	260156	165140	100371	165140	4710	18.2	8	0.6	47
Pump shell	83908	53106	35873	53106	4274	14.6	8	0.7	30
Drill handle	101349	64542	49642	64542	4199	22.1	9	1.0	36

 \hat{T}_q : the total query time for answering 1M query points without preprocessing included (ms). $\#I$: the average number of intersection tests for a query point. sT_t : The time for answering 1M query points with the preprocessing time included (ms).

3D-GCP is that URC does not use any prior knowledge. For these three methods, each query point was assigned a thread in the inclusion test. The block sizes for Tetra, RC, and URC were 256, 256, and 128 threads, respectively, which are their optimal settings in our implementation.

In the tests, we collected the query time without preprocessing (T_q) for these methods to answer 1M query points, and recorded the average numbers of intersection tests for a query point (I) with these methods respectively, which usually reflects the degree of localization for a point-in-polyhedron test. A smaller I value means a higher degree of localization, also corresponding to a quicker test. According to the statistics in Table 3, RC and Tetra were the slowest, as they use global operations. URC has a moderate degree of localization because it uses a uniform grid. Thus, it was much faster than RC and Tetra, but slower than 3D-GCP. 3D-GCP has the highest degree of localization because it uses a grid and prior knowledge. Therefore, it was much faster than the other methods. In Table 3, we also list the total time of 3D-GCP (T_t) including both the preprocessing time and the query time. Statistics shows that even including the preprocessing time, 3D-GCP is still much faster than the other methods. This provides a solid base to treat the polyhedrons with their shapes changed frequently.

5.2. Comparison with serial algorithms

Because we did not aim to compare all existing serial point-in-polyhedron methods, we only compared our technique to two typical state-of-the-art serial methods. One is based on face layers (denoted by Layer) [11], which is a typical method of reorganizing geometrical information for fast inclusion tests. To determine the basic performance of the method, we did not combine the Layer with an octree. The other is RCT [1], which is typical in using a spatial partition structure. We obtained the implementations of both methods from their authors.

According to the statistics in Table 4, 3D-GCP significantly outperforms the competitors in terms of both preprocessing time and

total query time. The acceleration ratios for the preprocessing are 7–35 for RCT and 36–158 for Layer. The improvement in query time is even more obvious and obtains an acceleration of 2–4 orders of magnitude. In terms of the auxiliary space cost, 3D-GCP has less storage requirements than RCT, but more than Layer. However, compared with the improvement in speed, the increase in storage is small (at most twice). Furthermore, we can decrease the space costs by decreasing the grid resolution.

5.3. Performance in terms of time and storage

To further verify the performance of 3D-GCP with respect to the number of faces, we created 10 torus knot models that contain 200–102,400 faces respectively. The results for 3D-GCP, Layer, and RCT with respect to the number of faces are illustrated in Fig. 5. For the preprocessing time (Fig. 5(a)) and storage cost (Fig. 5(c)), although all the curves have similar shapes for these 10 models, the 3D-GCP curve has the minimum degree of inclination. For the query time, Layer and RCT are unstable because their auxiliary structures are sensitive to the model shape. For example, even with the same number of faces, complex shapes with many face layers are much slower than simple shapes with fewer layers. In comparison, 3D-GCP is very stable and fast. Its query time complexity is almost $O(1)$ for the tested models and the query time for 1,000,000 points is only 8 ms.

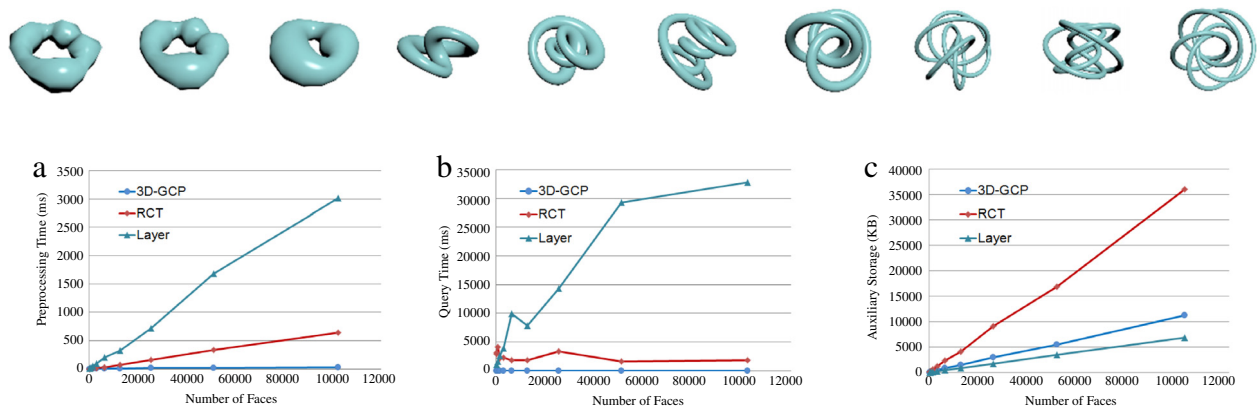
6. Conclusion

In this paper, we present a new method for point-in-polyhedron determination. By constructing a uniform grid for managing the faces of the polyhedron and predetermining the center points of grid cells as priors, the method can perform inclusion tests by applying ray-crossing locally, where a unified solution is presented to handle all possible singular cases. Moreover, we develop measures to quickly predetermine the center points of grid cells and effectively implement on GPUs. Consequently, we can perform

Table 4

Performance comparison between 3D-GCP and the serial algorithms Layer [11] and RCT [1].

	$\#T_p$			$\$T_q$			$\&S$		
	Layer	RCT	3D-GCP	Layer	RCT	3D-GCP	Layer	RCT	3D-GCP
Bunny	301	68	8	12471	1625	7	770	3580	1245
Horse	1339	363	18	26986	1349	8	4238	18692	6429
Dragon	8389	1881	53	85013	3119	8	18503	83904	29551
Gear Box	6004	1005	39	49686	1568	8	11055	45708	18618
Pump Shell	2144	350	22	28500	1930	8	3599	15964	6105
Drill Handle	2804	422	27	33938	3259	9	4346	20260	7873

 $\#T_p$: the preprocessing time (ms). $\$T_q$: the total query time for answering 1M query points without preprocessing included (ms). $\&S$: the auxiliary space cost (KB).**Fig. 5.** Performance curves for 10 torus knot models using Layer, RCT, and our 3D-GCP.

inclusion tests against 3D polyhedrons robustly and achieve an acceleration of several orders of magnitude over the existing methods, as attested by experimental results. Therefore, we can well address the challenge in applications, quickly answering many query points against large polyhedrons, even for polyhedrons with frequently changing shapes.

Acknowledgments

We appreciate the valuable suggestion from anonymous reviewers. We thank the authors of Ref. [1] for providing us their programs and test models. This work is partially supported by the National Natural Science Foundation of China (Nos. 60873182, 61379087, U1435220, and 61661146002), and the People Programme (Marie Curie Actions) of the European Unions Seventh Framework Programme (FP7/2007–2013) under Grant 612627.

References

- [1] Liu JF, Chen YQ, Maisos JM, Luta G. A new point containment test algorithm based on preprocessing and determining triangles. *Comput-Aided Des* 2010;42(12):1143–50.
- [2] Kalay YE. Determining the spatial containment of a point in general polyhedra. *Comput Graph Image Process* 1982;19(4):303–34.
- [3] Lane J, Magedson B, Rarick M. An efficient point in polyhedron algorithm. *Comput Vis Graph Image Process* 1984;26(1):118–25.
- [4] Carvalho PCP, Cavalcanti PR. Point in polyhedron testing using spherical polygons. In: Paeth AW, editor. *Graphics Gems V*. Academic Press; 1995. p. 42–9.
- [5] Horn WP, Taylor DL. A theorem to determine the spatial containment of a point in a planar polyhedron. *Comput Vis Graph Image Process* 1989;45(1):106–16.
- [6] Feito FR, Torres JC. Inclusion test for general polyhedra. *Comput Graph* 1997;21(1):23–30.
- [7] Moller T, Haines E. *Realtime rendering*. 2nd ed. A. K. Peters; 2002. p. 349–53 583–6, 596–7.
- [8] Wang CCL. Approximate Boolean operations on large polyhedral solids with partial mesh reconstruction. *IEEE Trans Vis Comput Graphics* 2011;17(6):836–49.
- [9] Heidelberger B, Teschner M, Gross M. Real-time volumetric intersections of deforming objects. In: *Proceedings of vision, modeling and visualization*; 2003. p. 461–8.
- [10] Faure F, Barbier S, Allard J, Falipou F. Image-based collision detection and response between arbitrary volume objects. In: *Proceedings of eurographics/ACM SIGGRAPH symposium on computer animation*; 2008. p. 155–62.
- [11] Wang W, Li J, Sun H, Wu E. Layer-based representation of polyhedrons for point containment tests. *IEEE Trans Vis Comput Graphics* 2008;14(1):73–83.
- [12] Rueda AJ, Feito FR, Ortega LM. Layer-based decomposition of solids and its applications. *Vis Comput* 2005;21(6):406–17.
- [13] Rueda AJ, Feito FR. EL-REP: A new 2D geometric decomposition scheme and its applications. *IEEE Trans Vis Comput Graphics* 2011;17(9):1325–36.
- [14] Ogayar CJ, Segura RJ, Feito FR. Point in solid strategies. *Comput Graph* 2005;29(4):616–24.
- [15] Ooms K, De Maeyer P, Neutens T. A 3D inclusion test on large dataset. In: Neutens T, DeMaeyer P, editors. *Developments in 3D geo-information sciences. Lecture notes in geoinformation and cartography*. New York: Springer; 2010. p. 181–99.
- [16] Horvat D. Ray-casting point-in-polyhedron test. In: *Proceedings of CESC 2012: The 16th central european seminar on computer graphics*. 2012.
- [17] Haines E. Point in polygon strategies. <http://erich.realtimerendering.com/ptinpoly/>.
- [18] Zalik B, Kolingerova I. A cell-based point-in-polygon algorithm suitable for large sets of points. *Comput Geosci* 2001;27(10):1135–345.
- [19] Gombosi M, Zalik B. Point-in-polygon tests for geometric buffers. *Comput Geosci* 2005;31(10):1201–12.
- [20] Sud A, Govindaraju N, Gayle R, Manocha D. Interactive 3D distance field computation using linear factorization. In: *Proceedings of ACM symposium on interactive 3D graphics and games*; 2006. p.117–24.
- [21] Li W, McMains S. Voxelized minkowski sum computation on the GPU with robust culling. *Comput-Aided Des* 2010;43(10):1270–83.

- [22] Rueda AJ, Ortega L. Geometric algorithms on CUDA. In: Proceedings of GRAPP; 2008. Paper No. 59.
- [23] Taranta EM, Pattanaik SN. Macro 64-regions for uniform grids on GPU. *Vis Comput* 2014;30(6–8):615–24.
- [24] Kalojanov J, Slusallek P. A parallel algorithm for construction of uniform grids. In: Proceedings of the conference on high performance graphics; 2009. p. 23–8.
- [25] Linhart J. A quick point-in-polyhedron test. *Comput Graph* 1990;14(3–4):445–7.
- [26] Wang CCL, Leung Y-S, Chen Y. Solid modeling of polyhedral objects by layered depth-normal images on the GPU. *Comput-Aided Des* 2010;42(6):535–44.
- [27] Leung Y-S, Wang CCL. Conservative sampling of solids in image space. *IEEE Comput Graph Appl* 2013;33(1):14–25.