Computers & Graphics 122 (2024) 103951

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Technical Section

3D geometric kernel computation in polygon mesh structures ☆

Merve Asiler *, Yusuf Sahillioğlu

Computer Engineering Department, Middle East Technical University, 06800 Ankara, Turkey

ARTICLE INFO

Keywords:
3D kernel
3D visibility
Mesh guidance
Star-shape

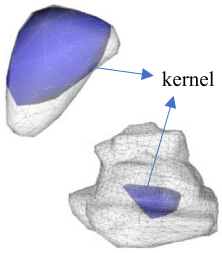
ABSTRACT

This paper introduces a novel approach to compute the geometric kernel of a polygon mesh embedded in 3D. The geometric kernel defines the set of points inside or on the shape's boundary, ensuring visibility of the entire shape. The proposed method utilizes scattered rays to identify a sufficient number of sample points on the kernel surface and subsequently leverages these points to locate as many surface vertices as possible. By computing the convex hull of these identified points, we derive an approximation of the kernel. Notably, the output of our method consists exclusively of interior or boundary points of the actual kernel. Comparative evaluations against established CGAL and Polyhedron Kernel algorithms highlight our method's superior computational speed and high approximation accuracy. The parametric structure of our solution allows for different levels of accuracy to be obtained, enabling the user to tailor the approximation to their specific needs. This property sets our algorithm apart from others and provides greater flexibility in its use. Additionally, adjusting the algorithmic settings also enables the computation of the kernel itself with a trade-off in computational speed. Furthermore, our algorithm swiftly and accurately identifies an empty kernel for non-star-shaped configurations.

1. Introduction

The geometric kernel of a mesh consists of the points from which the entire mesh is visible via a line segment that remains completely within the shape. The points may be located within the region bounded by the mesh faces or on the faces themselves. If such a single point does not exist, then the kernel is an empty set. Otherwise, the kernel has a convex shape and its host mesh is called as star-mesh.

Various geometric problems benefit significantly from kernel-based solutions. For instance, employing naturally defined visibility parameters derived from the kernel enables effective solutions to spherical parametrization [1]. In shape guarding problems, determination of guard points relies on the kernel concept [2–4]. Additionally, ordering points of a mesh based on its kernel viewpoint aids in detecting self-intersections [5,6] and ensures collision-free mesh deformation [7]. Lastly, the ratio of kernel area/volume to the total shape area/volume serves as a vital metric for evaluating shape quality [8,9].



While directly computing the kernel itself is valuable, approximate kernel computation is beneficial in various contexts. Firstly, it can be employed in cases where rapid computation is essential, and an approximation that highly aligns with the actual kernel's shape suffices. For instance, in mesh generation and mesh refinement applications requiring recurring dynamic shape quality measurements, fast approximations closely resembling the kernel's area/volume offer advantages [10]. Additionally, during the segmentation of shapes into star pieces, approximate kernel computation for each segment to be decomposed significantly optimizes the decomposition speed [3,4,11].

☆ This article was recommended for publication by Marco Attene.

* Corresponding author.
E-mail addresses: asiler@ceng.metu.edu.tr (M. Asiler), ys@ceng.metu.edu.tr (Y. Sahillioğlu).

https://doi.org/10.1016/j.cag.2024.103951
Received 25 February 2024; Received in revised form 23 May 2024; Accepted 27 May 2024
Available online 28 May 2024

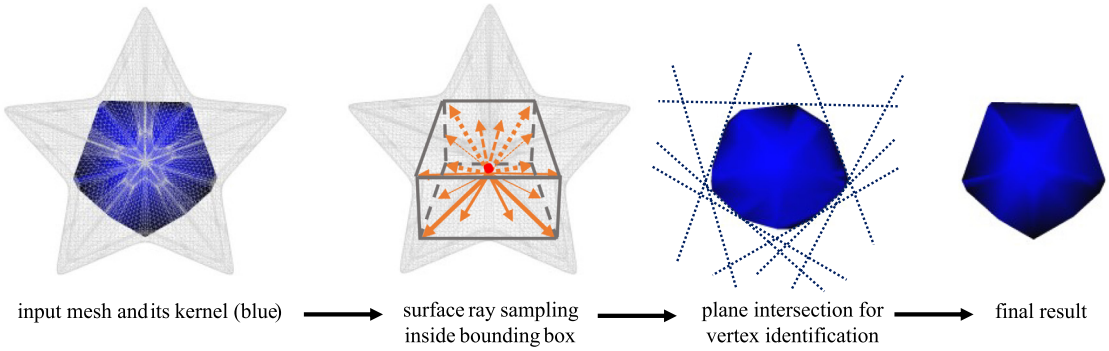


Fig. 1. Algorithm pipeline. Points on the kernel boundary are efficiently sampled by scattering rays from the center of the kernel's bounding box, guided by surface geometry. Subsequently, the approximation is enhanced via vertex identification using boundary planes of the sampled points.

Secondly, it affords the opportunity to adjust the level of details on the host mesh. For example, in both mesh simplification and mesh subdivision problems, employing different approximations of the kernel provides control over defining distinct details on the host mesh boundary, as demonstrated in studies focusing on mesh generation from kernels [12,13].

Motivated by the aforementioned applications of kernels in computer graphics, this paper presents a novel algorithm to compute the kernel of a polyhedron. Our approach employs different techniques compared to the existing solutions such as CGAL (Computational Geometry Algorithms Library) [14] and Polyhedron Kernel [15]: It begins by approximating the boundary shape of the kernel by sampling points on its surface, considering the kernel's surface geometry, and then efficiently identifying vertices along the boundary through a recursive search (Fig. 1). Our contributions are:

1. Our paper introduces a novel kernel computation algorithm for polygonal meshes embedded in 3D. The output consistently includes interior or boundary points of the kernel, ensuring the exclusion of exterior points. Under baseline settings, our algorithm effectively approximates the shape of the kernel. Furthermore, it offers the flexibility to adjust the sampling quantity and search depth, enabling users to attain their desired level of approximation, a feature not available in existing methods. Additionally, the approximation can be refined towards the kernel itself with a trade-off in execution speed.
2. We conduct comprehensive experimental evaluations comparing the performance of our proposed method with CGAL and Polyhedron Kernel. Our evaluation includes metrics encompassing execution times, visual representations, volume losses and Hausdorff distances between the outputs of each method. Our algorithm demonstrates superior efficiency and low error rates, particularly for star-shaped geometries, making it an optimal choice for various cases.
3. The proposed algorithm effectively identifies and promptly reports an empty kernel for non-star-shaped configurations.

The remainder of the paper is organized as follows: Section 2 provides a comprehensive review of existing studies related to kernels. Section 3 elucidates the theoretical and technical implementation details of the proposed solution, accompanied by complexity analysis. Subsequently, Section 4 presents numerical and visual experiment results obtained through comparative analysis of the aforementioned algorithms. Finally, Section 5 encompasses the conclusions drawn from the study and outlines potential future work.

2. Related work

In the literature, early work related to kernel computation was pioneered by Shamos et al. [16]. Their study established that the kernel of a polygon with N_f faces could be computed in $\mathcal{O}(N_f \log N_f)$ time. Subsequently, Preparata et al. introduced an $\mathcal{O}(N_f \log N_f)$ algorithm for determining the kernel of a polyhedron in their study [17]. They approached the problem by recognizing the kernel as the intersection set of all halfspaces bounding the mesh. To tackle this, they formulated a linear algebra system based on halfspace inequalities and solved it using the simplex method. Implementations of their method can be found in several renowned libraries such as CGAL [14] and Libigl [18]. Later, Lee et al. [19] presented an $\mathcal{O}(N_f)$ time method, analyzing how convex and reflex angles at vertices of a polygon, whose edges are given in order, contribute to shaping its kernel.

Recently, Sorgente et al. introduced a new algorithm in the Polyhedron Kernel study [15], approaching the problem from a geometric perspective with a computational cost bounded by $\mathcal{O}(N_v N_f)$ where N_v is number of vertices. Their methodology initiates from a rough bounding box encompassing the kernel and progressively refines it using bounding planes of the mesh until achieving the desired set.

Besides computing the entire kernel, it is possible to identify a single point on the kernel boundary using linear programming in $\mathcal{O}(N_f)$ time. This can be achieved by sweeping across the polygon and analyzing line slopes, as proposed by Gomez et al. [20]. Additionally, Berg et al. present a generalized algorithm applicable across all dimensions. They introduce a unit vector into the system and derive a point positioned in the direction of the given vector, with an expected time complexity of $\mathcal{O}(N_f)$ [21].

Research endeavors also involve the analysis of the algebraic and geometric properties of the kernel. These studies explore properties like mean value coordinates of kernel points and maximums of kernel boundaries [22], finite element formulations over star-shapes [23,24] and algebraic derivations over the kernel of free-form curves and free-form surfaces [25,26]. Utilizing the mathematical relationships and properties associated with the kernel is crucial for driving potential optimizations in kernel-related algorithms. Furthermore, it opens avenues for the development of new methods and expands the application areas of kernel-related processes.

Another relevant area within this domain is star-shape segmentation, which is partitioning the object into pieces with non-empty kernels. In addition to the studies that perform star decomposition on a given single shape [27,28], there is also a method that handles the problem in a more complex setting where compatible decompositions between two shapes are desired [29]. Yu et al. perform star decomposition in 3D with the purpose of shape guarding on meshes [3,4]. Similarly, Hiraki et al. employ star-kernel decomposition for robot viewpoint planning [30].

Keil et al. aim to achieve a decomposition with the minimum number of pieces, including star-shaped components, as discussed in their work [31]. The particular problem is known as the *Art Gallery Problem*, acknowledged as NP-Hard [32]. This problem involves achieving visibility coverage, where a solution with at least one kernel point at each component is sought from various perspectives. While some studies

offer exact solutions for art gallery problems in both polygons and terrains [33], others provide approximate solutions [34]. Amit et al. address the problem of locating guards to provide visibility coverage for polygons [35], while Banerjee et al. focus on the same concept for dynamic orthogonal art gallery scenarios [36]. Lastly, Bose et al. extend the scope of the art-gallery problem to three-dimensional scenarios by focusing on guarding polyhedral terrains [37].

3. Algorithm

In this section, we elaborate on our algorithm designed to compute the kernel of a polyhedron. We specifically consider closed manifold meshes while developing our algorithm. We outline the specific settings crucial for optimal results and emphasize the potential to enhance the approximation by adjusting these settings, converging the computed shape towards the actual kernel with an associated increase in computation time. Our methodology involves the examination of each mesh face within its corresponding plane and normal direction. Each face defines a halfspace, a concept we consistently reference alongside their respective boundary planes throughout this paper. Algebraically, the kernel is the intersection of all these halfspaces, and its surface consists of the intersection of their bounding planes [17].

3.1. Extreme point search

In our algorithm, we utilize Berg et al.'s single point finder method [21], which employs incremental linear programming to find the kernel's extreme point in a specified direction. This method operates on a set of constraints defined by the halfspaces bounding the kernel, maximizing the objective function, typically defined as the scalar product of the given direction vector with the target solution. It begins by initializing a starting vertex, v_0, at one of the corners formed by intersections of halfspaces. Then, it iterates through each halfspace, checking if the previous vertex v_{i-1} lies within it. If it does, the current vertex v_i remains the same as the previous one. Otherwise, it computes the point p on the boundary of the halfspace that maximizes the objective function, subject to the constraints imposed by the previous halfspaces. If such a point does not exist, indicating infeasibility, it reports that the kernel is empty. Conversely, if the algorithm successfully completes all iterations, it returns the last computed vertex as the extreme point within the kernel in the specified direction.

3.2. Bounding box utilization

In our study, we determine the smallest axis-aligned bounding box, referred to as the $AABB$, that encloses the kernel. Performing operations utilizing the $AABB$ enhances computational efficiency by directing the search towards the kernel's location.

We compute the $AABB$ through a simple two-step process: (i) Using the single point finder method detailed in Section 3.1, we identify the extreme kernel points in six directions: $(1,0,0)$, $(-1,0,0)$, $(0,1,0)$, $(0,-1,0)$, $(0,0,1)$, and $(0,0,-1)$. If multiple extreme points exist in a given direction, we select the first one based on lexicographical order. (ii) From the coordinates of these six points, we determine the maximum and minimum x, y, and z values. These extreme coordinates define the $AABB$, and by assigning each corner one of the eight $\langle x, y, z\rangle$ combinations, we construct the bounding box. The construction of the $AABB$ is illustrated in Fig. 2.

It is important to note that if the shape is non-star, no kernel points can be found in any direction. In such cases, the kernel computation process is terminated, and the algorithm returns an empty set. Therefore, when referring to the $AABB$, we assume a non-empty bounding box, which implies the existence of the kernel.

Claim 1. *The center C_p of the $AABB$ lies within the kernel.*

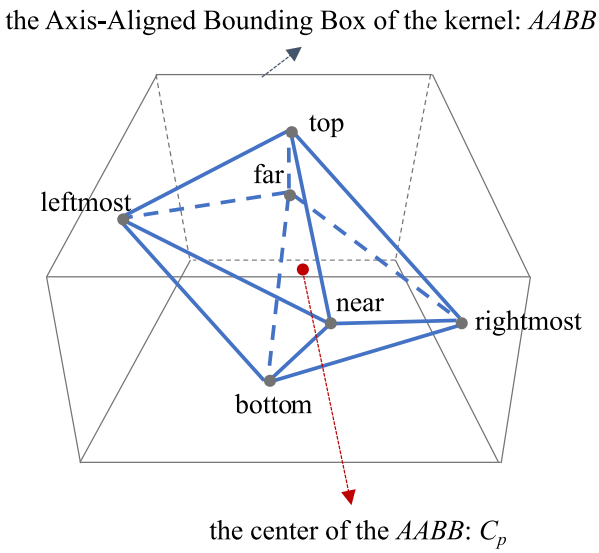


Fig. 2. The construction of the Axis-Aligned Bounding Box ($AABB$) for the kernel, defined by the maximum and minimum x, y, and z coordinates of extreme kernel points in six principal directions.

Proof. Let us assume the contrary, i.e., suppose C_p is not within the kernel. This implies the existence of a halfspace that does not contain C_p. However, since the extreme points found in the six directions are part of the kernel, they are contained by all the halfspaces. Therefore, the boundary plane of the excluding halfspace must pass between C_p and each of those extreme points, leading to a contradiction. □

3.3. Surface ray sampling

We approximate the kernel by initially sampling its surface, accomplished by sending rays towards its boundary from the center C_p (Claim 1), each directed towards a distinct direction. Rather than uniformly distributing rays in a spherical manner, we distribute them aligning better with the kernel's surface geometry. This distribution is managed as follows: Utilizing the extreme kernel points used to construct the $AABB$, we generate triangular structures surrounding the $AABB$'s corners. We create a single triangle for each corner by combining the three nearest points around it, as illustrated in Fig. 3. Although these structures mostly form triangles, they might occasionally appear as a single line or point if the same extreme point is utilized for different directions. However, this does not pose an issue since the generated structures effectively cover the kernel's interior from all directions. For simplicity, we consider each of these geometric shapes as a triangle. Subsequently, we seek to determine specific points on these triangles through which the rays traverse, thereby assigning a direction to each ray. To attain a uniformly distributed point set on the triangle, we select points utilizing barycentric coordinates, following the formula outlined by Osada et al. in [38]. Since there is no direct correlation between the area of a corner triangle in the $AABB$ and the geometric variability of the kernel surface, we distribute an equal number of rays to each triangle. For most input shapes, employing approximately thirty rays yields satisfactory outcomes. Increasing the ray count can improve the approximation quality further.

Since corner triangles are generated using extreme kernel points in the principal directions of 3D space, their composition largely conforms to the morphology of the kernel body. Additionally, each triangle aligns with the underlying surface geometry of the kernel on the corresponding side. Importantly, in cases where the kernel exhibits nearly spherical geometry, the resulting triangle sampling closely resembles spherical sampling. Our empirical study results validate the effectiveness of this approach (Section 4.4).

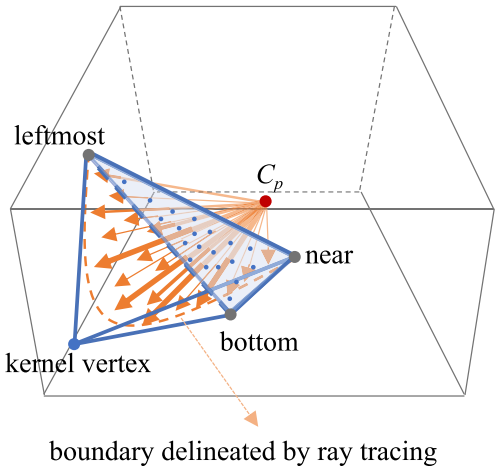


Fig. 3. Corner triangles and uniform sampling. A single triangle is formed around each corner of the $AABB$ by combining the corresponding extreme kernel points (as shown for the leftmost, near, and bottom corner). After uniform sampling within each triangle, rays are projected from C_p to the sample points. The first intersection of each ray with the planes is selected to delineate the initial boundary for the kernel.

Once the directions of the rays are established, we project them to intersect with the boundary planes and identify the points of intersection. Since the outer intersections are not included by the half-spaces giving the inner intersections, which contradicts the definition of a kernel point [17], we select the point closest to C_p for each ray. This allows us to obtain a point on the surface of the kernel.

3.4. Kernel vertex identification

This stage aims to identify as many vertices of the kernel mesh as possible to improve the approximation. We seek the planes capable of generating kernel vertices by identifying the boundary planes associated with the kernel points found in the previous step (Section 3.3). Our approach begins by computing the convex hull of the kernel points obtained via ray tracing. This step establishes adjacency relationships between points. It is crucial to note that, given the convex nature of the kernel itself, the resulting convex hull is entirely enclosed by the kernel. While it partially touches the kernel's boundary, it remains partially inside the kernel. Furthermore, if any kernel vertex is not included in the computed convex hull, its generating planes either tangentially intersect the current convex hull at its adjacent vertices or do not intersect it at all. Consequently, for each vertex on the convex hull, we analyze its neighboring vertices and utilize the associated boundary planes to generate potential kernel vertices. For every group of three planes, we check their intersections in separate recursive processes. To confirm whether an intersection denotes a kernel vertex, we project a ray from C_p to the intersection point, as illustrated in Fig. 4. If the ray reaches the target before encountering another boundary plane, we identify it as a kernel vertex. Otherwise, we substitute the obstacle plane encountered by the ray with the candidate planes in the group of three one by one, checking intersections at subsequent recursion depths. If the planes in any group do not intersect at a point, we stop that recursion step and continue with the other branches of the recursion tree. The recursive process continues until a kernel vertex is determined, limited to the third recursion depth to prevent excessively long execution times. Experimental studies demonstrate that, with the given settings, our algorithm efficiently identifies most or all of the kernel vertices.

3.5. Kernel shape extraction

In the final step, we extract the shape of the kernel by computing the convex hull formed by all detected points on the kernel's surface,

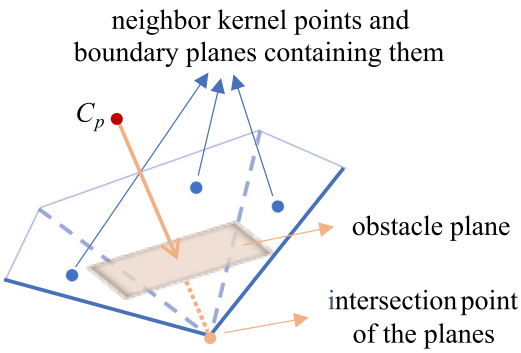


Fig. 4. Vertex identification stage. Boundary planes of previously identified kernel points are utilized to initiate a recursive intersection-checking process. A ray is sent from C_p to the intersection point. If the ray encounters a boundary plane before reaching the target point (referred as the obstacle plane), it is replaced with the previous ones at subsequent recursive depths.

including both vertices and non-vertices. Since the kernel itself is inherently convex, our approach theoretically provides the exact shape of the kernel when all kernel vertices are identified. Otherwise, the generated shape remains within the kernel boundaries, as all points within the convex hull belong to the kernel.

3.6. Complexity analysis

Our algorithm begins by identifying extreme points in six primary directions (as discussed in Section 3.2) and computing the $AABB$. Locating an extreme point in a specific direction entails an expected complexity of $\mathcal{O}(N_f)$ [21], where N_f denotes the number of boundary planes. Subsequently, rays are cast towards the triangles constructed near the corners of the $AABB$. For each ray, we determine its intersection with all boundary planes to find the innermost intersection point, considered as a point on the kernel's surface. This phase requires $\mathcal{O}(N_f N_r)$ time, where N_r represents the total number of rays.

After the ray casting process, we obtain N_r points on the kernel's surface and record each point along with its corresponding boundary plane detected during the intersection computations. In cases where more than one boundary plane passes through a kernel point, we record all of them. We assume that each kernel point corresponds to one boundary plane, as it is typically the case. However, in rare instances where multiple boundary planes exist at a given kernel point, we account for them in the scenario of increased recursive depth, as part of the worst-case scenarios explained later.

Moving on to the vertex identification phase, we compute the convex hull of the N_r kernel points, yielding a complexity of $\mathcal{O}(N_r \log N_r)$. Subsequently, we compute the intersection of boundary planes carrying neighbor points. In a worst-case scenario, a point might be a neighbor to all other remaining points, although this occurrence is rare. Our observations suggest that a point generally has considerably fewer neighbors than N_r, resulting in only a few potential plane combinations to form a kernel vertex. Therefore, determining whether the intersection of the selected three planes in each combination constitutes a point belonging to the kernel results in a total computational complexity of $\mathcal{O}(N_f N_r)$. If the point lies outside the kernel, we generate new candidate plane triples and continue the process recursively, limiting the recursion to a depth of three. In the worst-case scenario, where a point is adjacent to all other points on the convex hull or the recursive process continues until reaching a kernel vertex, the maximum cost could reach $\mathcal{O}({N_f}^3)$. This complexity arises from the brute-force intersection check of all possible triple combinations of boundary planes.

Finally, we compute the convex hull of newly detected vertices along with the previously discovered points on the kernel's surface. Since the kernel is the intersection of all the boundary halfspaces of

Table 1
The execution times (ms) and our error rates for various sample star-shaped meshes. The last two rows present the average values for the *Thingi* and *Princeton* datasets, where the number of star-shaped objects is provided instead of the number of faces.

Input	# Faces	CGAL	Polyhedron kernel	Ours	Volume loss (%)	Hausdorff distance
Crossing Cubes	29994	1242	668	**300**	0.012	0.0025
Cube on Cylinder	29986	1377	1128	**405**	0.0094	0.0046
Cylinder Trio	23062	1167	1116	**538**	0.0045	0.0018
Wringer	17514	753	141	**135**	0	0
Acorn	8224	443	337 743	**366**	0.254	0.063
Ball	1316	**76**	766	633	0.22	0.19
Cross	7824	2130	10 814	**203**	0.00037	0.0023
Flex	1664	292	**28**	84	0	0
Muffin	17940	**880**	151 049	1830	1.48	0.39
Plus	892	133	**6**	15	0	0
Thingi (AVG)	320	252	5132	**92**	0.38	0.17
Princeton (AVG)	12	1028	1159	**376**	0.002	0.0007

the input mesh, there can be at most N_f many faces of the output kernel, resulting in approximately half as many vertices based on Euler's formula. Therefore, the cost of the final convex hull is $\mathcal{O}((N_r + N_f)\log(N_r+N_f))$. Our observations consistently reveal that the number of detected kernel vertices remains significantly lower than N_r, given a sufficiently large N_r. Consequently, we conclude that $\mathcal{O}(N_f N_r)$ serves as an upper limit for the complexity of our implementation under the specified conditions to ensure optimality. In the worst-case scenario, the complexity increases to $\mathcal{O}({N_f}^3)$ when N_r is less than or equal to ${N_f}^2$, and $\mathcal{O}(N_f N_r)$ otherwise.

4. Experiments

The experiments were conducted on a Windows machine equipped with a 2.20 GHz Intel(R) Core(TM) i7-8750H CPU and 16 GB of RAM. All implementations were performed using C++. The source code for the implementation is accessible at https://github.com/merveasiler/Geometric-Kernel-Approximation.git.

Our algorithm was tested on a reduced version of the *Thingi10K* dataset provided by Sorgente et al. [39] and the *Princeton* dataset [40]. The reduced *Thingi10K* dataset was specifically prepared for meaningful inputs with non-empty finite kernels as well as non-star shapes, including data with one connected component, genus zero, Euler characteristic greater than zero, closed shapes, non-degenerate structures, all of which are smaller than 1MB in size.

To quantitatively assess the output quality of our algorithm, we measured the volume loss and symmetric Hausdorff distance. These metrics were used to evaluate the accuracy and similarity of our algorithm's output compared to ground truth data. For this purpose, we utilized CGAL's halfspace intersection algorithm provided by Preparata et al. [17].

Statistical results were classified based on whether the input data represented a star-shaped object or not. Following this, an experimental study was carried out by refining the same object to observe performance changes concerning speed and approximation accuracy for our algorithm. Additionally, we assessed the impact of each phase of our algorithm, utilizing various settings, on both computation time and approximation quality.

During comparison tests with other existing algorithms, Polyhedron Kernel ran in shuffle mode, while our proposed solution employed the optimal settings outlined in Section 3.

4.1. Star-shaped objects

Table 1 presents statistical data obtained from the given algorithms applied to a selection of star-shaped objects from both the *Thingi* and *Princeton* datasets. The objects chosen include Acorn (ThingiID: 815480), Ball (ThingiID: 58238), Cross (ThingiID: 313882), Flex (ThingiID: 827640), Muffin (ThingiID: 101636), Plus (ThingiID: 1120761) from the *Thingi* dataset [39] and Crossing Cubes (PrincetonID: 325), Cube on Cylinder (PrincetonID: 326), Cylinder Trio (PrincetonID: 337), and Wringer (PrincetonID: 350) from the *Princeton* dataset [40]. Corresponding visuals for each shape, demonstrating their actual kernels by CGAL and the kernels using our algorithm, are illustrated in Fig. 5.

Table 1 provides clear evidence of our algorithm's high accuracy in yielding quality results within short computation times for sample inputs. Notably, Polyhedron Kernel's efficiency diminishes for larger inputs such as Acorn, Cross, and Muffin, as expected due to its design for smaller-sized data. However, for meshes comprising numerous planar or nearly planar faces, Polyhedron Kernel performs equally or even faster than CGAL, despite having a large number of faces, as observed in cases like Crossing Cubes, Cube on Cylinder, Cylinder Trio, and Wringer.

Concerning the computation time, our algorithm demonstrates fast kernel computation for both large and small meshes. While it generally outperforms both CGAL and Polyhedron Kernel in most cases, there might be instances, particularly with smaller-sized inputs like Flex and Plus, where it slightly lags behind Polyhedron Kernel. Additionally, for kernels exhibiting highly rounded surfaces like Ball and Muffin, it might slightly fall short compared to CGAL. Our algorithm may also experience a slight increase in volume difference for round-shaped kernels due to the challenge of detecting kernel vertices by assessing plane intersections in specific areas. This difficulty arises when the curvature of the kernel surface increases, leading to many closely positioned planes that complicate identifying the correct generator planes in the search for a vertex. Nevertheless, Fig. 5 demonstrates that the kernels computed by our method closely resemble the actual kernels, even for those with curvilinear surfaces. Notably, our algorithm precisely identifies the kernel for inputs Flex, Plus, and Wringer.

Fig. 6 showcases the timing results obtained across the entire *Thingi* and *Princeton* datasets. For the *Thingi* dataset, our algorithm shows comparable computation speeds to CGAL and Polyhedron Kernel for meshes containing fewer than 1000 faces (Fig. 6). However, for larger meshes, our method significantly outperforms the other algorithms in terms of computation time. Similarly, for the *Princeton* dataset, our algorithm consistently computes kernels faster than CGAL and Polyhedron Kernel. As indicated in Table 1, our algorithm maintains faster computation times for both datasets on average.

Table 2 provides statistics for each group of results concerning volume loss. Our algorithm successfully identifies all kernels with a volume loss of at most 0.012% and accurately determines the kernel itself for 8 out of 12 star-shaped meshes in the *Princeton* dataset. In the *Thingi* dataset, it precisely computes the kernel for 121 meshes and approximates the kernel for 81 meshes with a volume loss below 0.01%. Additionally, it achieves volume differences of lower than 1% for 72 meshes, below 3% for 41 meshes, and below 6% for the remaining 5 meshes. The table also presents average timings for each group, with our algorithm consistently demonstrating the highest speed. Finally,

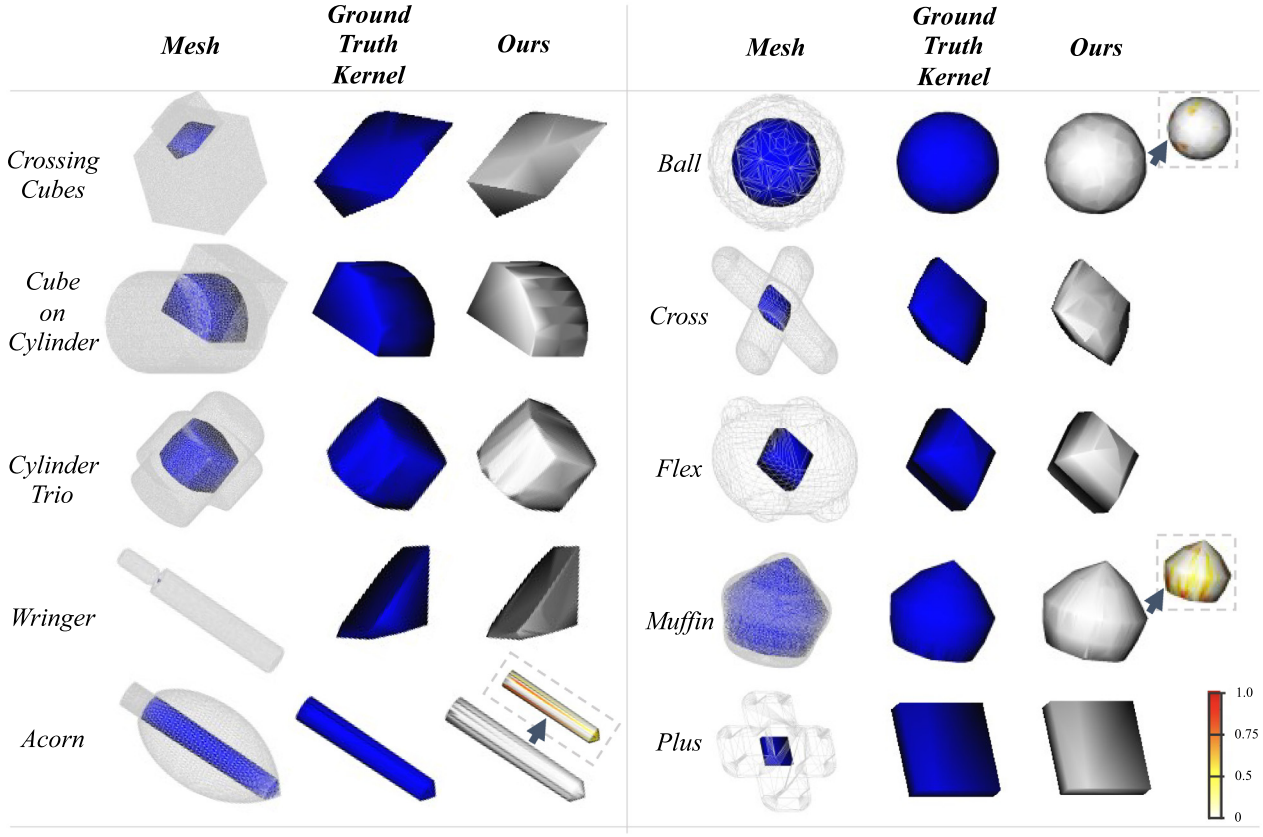


Fig. 5. The kernels computed by our method, as detailed in Table 1. To enhance clarity, both the ground truth kernels and our computed kernels were scaled up at the same ratio. The visuals highlight the close resemblance between our computed kernels and the actual kernels. Furthermore, pointwise errors (Euclidean distance from the ground truth) are visualized for kernels obtained with a volume loss exceeding 0.2%. These error values are normalized for each individual output, with hot colors indicating larger errors.

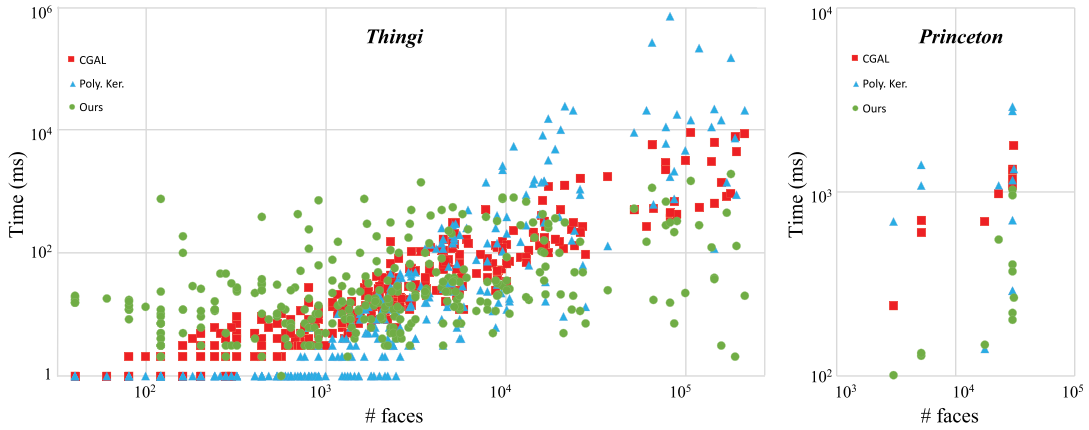


Fig. 6. A comprehensive overview of timing results obtained across the *Thingi* and *Princeton* datasets.

our empirical results did not suggest any correlation between the varying number of input mesh faces across different geometries and the accuracy of the approximation. Rather, the primary determinant of accuracy appears to be the inherent geometry of the input mesh itself.

In conclusion, our kernel computation algorithm demonstrates significant advantages over CGAL and Polyhedron Kernel, especially for larger inputs, considering its high accuracy and rapid computation time.

4.2. Non-star-shaped objects

Our algorithm initiates by computing extreme points along the primary axis directions to derive the $AABB$ encompassing the kernel. The absence of any identified point signifies an empty kernel set. Utilizing Berg et al.'s expected $\mathcal{O}(N_f)$ time algorithm [21] (where N_f denotes the number of mesh faces) enables a swift execution, returning results in nearly 0 ms for numerous input meshes. For contextual reference, we computed the average execution time of our algorithm. The results are given in Table 3. Across 1482 non-star shapes in the *Thingi* dataset, the average execution time stands at 0.00001 ms, whereas it registers at 0.0004 ms for 368 non-star shapes within the *Princeton* dataset.

In contrast, both CGAL and Polyhedron Kernel algorithms employ their standard intersection computation methods until encountering an empty set for non-star shapes. As delineated in Sorgente et al.'s research [15], the Polyhedron Kernel algorithm demonstrates superior

Table 2
The average execution times (ms) categorized by volume losses of our approximated kernels. While Polyhedron Kernel aims to provide exact kernels, minor volume differences in the outputs due to precision errors occurring in arithmetical operations may be observed, typically not exceeding 1e-6%.

Thingi Vol. loss (%)	# Shapes	CGAL	Poly. ker.	Ours
0	121	132	318	**38**
<0.01	81	164	408	**40**
< 1	72	400	12110	**172**
< 3	41	520	6263	**219**
< 6	5	275	4929	**138**
Princeton Vol. loss (%)	# Shapes	CGAL	Poly. ker.	Ours
0	8	892	750	**172**
<0.012	4	1218	1730	**662**

Table 3
The average execution times (ms) for non-star-shaped meshes.

Dataset	# Shapes	CGAL	Poly. ker.	Ours
Thingi	1482	788	402	**0.00001**
Princeton	368	1307	626	**0.0004**

performance over CGAL for all non-star models. Based on our experimental study, Polyhedron Kernel reports empty kernels with an average duration of 402 ms and 626 ms for the *Thingi* dataset and *Princeton* dataset, respectively. In comparison, CGAL's reporting time averages at 788 ms for the *Thingi* dataset and 1307 ms for the *Princeton* dataset.

4.3. Evaluation on increasing mesh faces

In this section, we examine how our method behaves as the number of mesh faces increases. We look at situations with fixed and changing numbers of distinct boundary planes. For our study, we use the *Spiral* (ThingiID: 60246) and *Vase* (ThingiID: 85580) shapes from the *Thingi* dataset. Additionally, we consider five iterations of each shape, where the number of mesh faces increases fourfold in each iteration, as done in the study by Sorgente et al. [15].

Fig. 7 shows how CGAL, Polyhedron Kernel, and our algorithm perform in terms of computation time for the *Spiral* model and its iterations. Each iteration is obtained by subdividing the triangles of the previous model into four. Therefore, the number of distinct faces remains unchanged between iterations, resulting in consistent kernel shapes. Consequently, our algorithm consistently identifies the same output for each iteration, which is the kernel itself. As shown in the chart, the time taken for computation gradually increases as the number of mesh faces grows, following similar trends observed in Polyhedron Kernel, always remaining lower than the computation time taken by CGAL.

Furthermore, Fig. 8 illustrates the changes in speed and approximation accuracy for the *Vase* model and its iterations. The refined versions of this model utilize different boundary planes, impacting the quality of approximations. As the refined *Vase* model contains more kernel vertices, the accuracy of our approximation slightly decreases due to the increased number of missed vertices. Nevertheless, our algorithm provides reliable approximations close to the true kernels. The computation time also increases similarly to the *Spiral* model, outperforming Polyhedron Kernel and consistently bettering CGAL.

To summarize, our algorithm consistently experiences longer computation times as the count of mesh faces increases. However, the accuracy may vary depending on the presence of different mesh faces, demonstrating adaptability to diverse mesh configurations.

4.4. Ablation studies

Impact of the ray count: Our method employs a ray casting technique to sample points on the kernel surface by projecting rays from a point

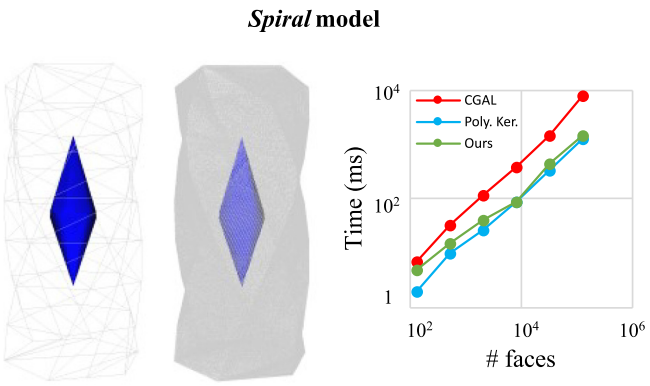


Fig. 7. The computation time comparisons for iterations of the *Spiral* model. In the current model, our algorithm successfully identifies the kernel itself.

inside the kernel in various directions (Section 3.3). Increasing the number of sample points enhances the potential for improved approximation in subsequent stages. Augmenting the number of rays, as shown in Fig. 9a, results in a greater number of sampled points on the kernel surface, capturing finer details and producing convex hulls closely resembling the true kernel. Conversely, reducing the number of rays leads to faster computation but a decrease in approximation accuracy. These results represent average values over the *Thingi* dataset, with the vertex identification stage limited to the third recursive search depth.

Fig. 10 illustrates the impact of changing ray count on Capsule (ThingiID: 58439), Diamond (ThingiID: 313917), and Tower (ThingiID: 472035) objects. We can see that the most significant changes occur particularly in regions exhibiting higher circularity.

For scenarios requiring an approximate kernel, users have the flexibility to intuitively adjust the number of rays according to the desired level of accuracy. In contrast, direct-result algorithms lack such interpretability, as their design does not allow for parameter adjustments.

Impact of the recursive search depth: We conducted experiments to assess the impact of different recursion depths in the vertex identification stage. As depicted in Fig. 9b, notable changes in accuracy are observed at depth 2, with a decrease in volume loss averaging under 0.4%. The results for two objects exhibiting the most visible changes, Cap (ThingiID: 40359) and Dome (ThingiID: 789073), are presented in Fig. 11. Continued recursive processing leads to increased surface detail in the outputs. These results were obtained using 240 rays in each run.

Similar to adjusting the ray count, recursion depth can also be utilized as a parameter to modulate the approximation accuracy. Deeper recursive searches naturally yield better approximation with a trade-off in computation time.

Impact of the vertex identification: To assess the impact of the vertex identification phase, we conducted experiments both with and without this stage, limiting the depth of the recursive search to three. Table 4 presents related statistics averaged over the *Thingi* dataset. The inclusion of the vertex identification stage notably improves output accuracy, yielding improvements of approximately 20 times, 7 times, and 2.5 times in terms of volume loss when using 240, 4800, and 96000 rays, respectively. However, this enhancement comes at the cost of increased computation time, which rises approximately 15, 19, and 5.6 times when including vertex identification with the specified number of rays.

As the number of rays increases, the discrepancy in output accuracy between versions with and without vertex identification diminishes. Consistent coloring in the table indicates similar results obtained by our algorithm's default mode (240 rays with vertex identification, recursive search limited to 3) and versions where vertex identification is not activated. To achieve a comparable speed performance to the default mode without vertex identification, 4800 rays are required. However, even using 20 times more rays fails to match the accuracy obtained

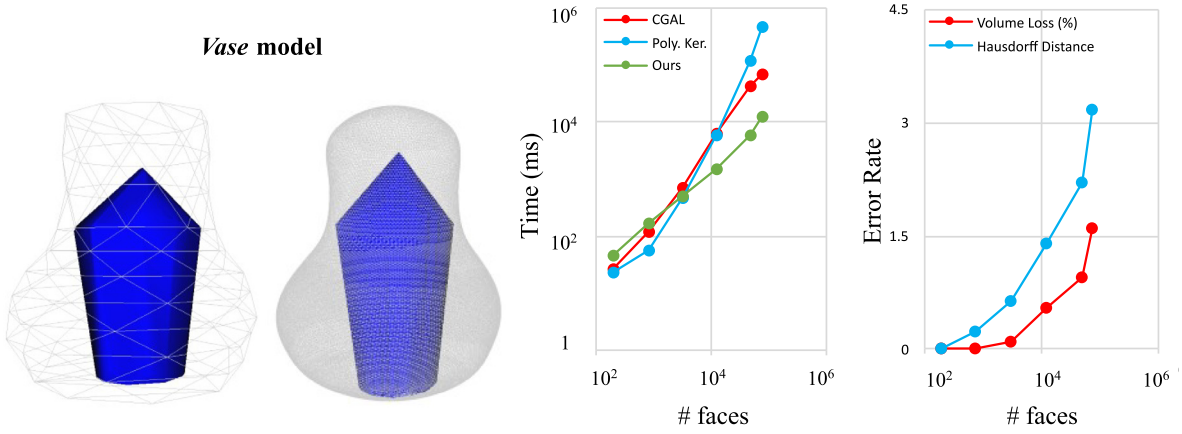


Fig. 8. The experimental results for iterations of the *Vase* model. While the accuracy decreases slightly with the increased vertex count in the refinements of the model, our algorithm consistently produces dependable approximations.

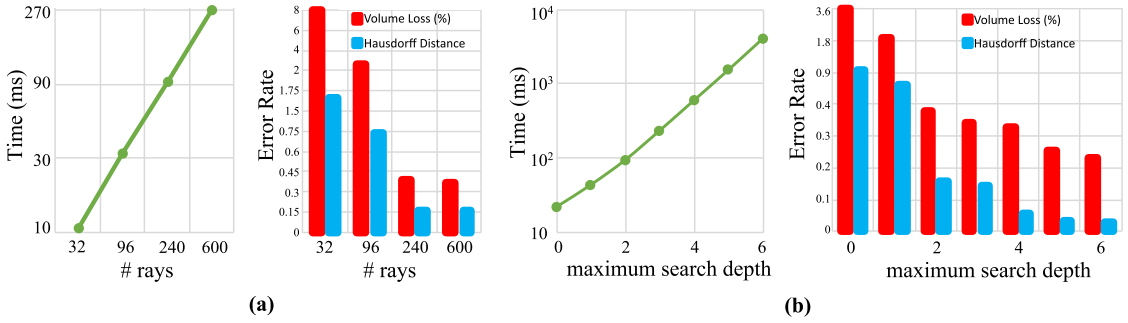


Fig. 9. Ablation Studies: (a) Examination of the impact of varying ray counts on execution time and output accuracy. Results are derived from runs incorporating vertex identification. (b) Analysis of execution time and output accuracy with different recursive search depths for vertex identification. Depth 0 signifies the initial search involving the first candidate generator planes, excluding subsequent recursive processes with obstacle planes.

Table 4
The average execution times and error rates of our algorithm obtained by different number of rays, with and without vertex identification. 'RA' and '+VI' stand for the terms 'rays alone' and 'vertex identification included', respectively. Consistent coloring indicates similar results with the default mode (the second column).

	240 Rays		4800 Rays		96000 Rays	
	RA	+VI	RA	+VI	RA	+VI
Time (ms)	6	92	94	1800	1615	9149
Vol. Loss (%)	7.6	0.38	1.96	0.27	0.43	0.16
Hausdorff Dist.	1.70	0.17	0.58	0.07	0.19	0.037

Table 5
The comparison between the average results obtained by spherical distribution and surface geometry-based distribution of the rays.

	Spherical distr.	Surface geometry-based distr.
Time (ms)	**88**	92
Vol. Loss (%)	3.31	**0.38**
Hausdorff Dist.	1.83	**0.17**

by vertex identification (volume loss: 1.96% vs. 0.38%). Additionally, attempting to achieve a similar quality result solely by increasing the number of rays, we observe that using 96000 rays (400 times more rays) brings a closer approximation accuracy to the default mode. Nevertheless, this significantly extends computation time, averaging 1615 ms compared to the default mode's 92 ms.

In conclusion, achieving similar or superior performance solely with rays, without vertex identification, appears unattainable, thus affirming the optimality of our default setting.

Similarly, in Fig. 12, we provide visual results showcasing the impact of the vertex identification phase on the Cocoon (ThingiID: 195696) and Wing (ThingiID: 41104) meshes. The figure displays outputs obtained with 240 rays alone, 2400 rays alone, and the default settings. We observe that the vertex identification can accurately detect small ridges and sharp corners, a capability not achievable solely through ray sampling. This underscores the advantage of including the vertex identification.

Comparison of spherical and surface geometry-based ray distribution: A comparative analysis between the algorithm versions employing spherical and surface geometry-based ray distribution yields insightful findings. Table 5 displays the average numerical results obtained with the default settings (240 rays and vertex search depth limited to 3) over the *Thingi* dataset. While both distribution methods exhibits comparable execution times, surface geometry-based distribution clearly provides superior accuracy in approximating the actual kernel.

Furthermore, Fig. 13 presents the output kernels obtained using both spherical distribution and surface geometry-based distribution for the Peg Top (ThingiID: 72581), Stopper (ThingiID: 69928), and Super Ellipse (ThingiID: 40172) meshes. It is evident that the surface geometry-based distribution tends to more accurately compute kernels, effectively capturing the general structures. Due to the non-uniform mapping of the kernels onto the unit sphere passing through their centers, deducing the complete kernel outline is more challenging with spherical ray distribution. Despite the implementation of vertex identification operations, insufficient point sampling in densely structured surface regions may lead to significant details being omitted from the kernel boundary. Consequently, notable discrepancies are observed,

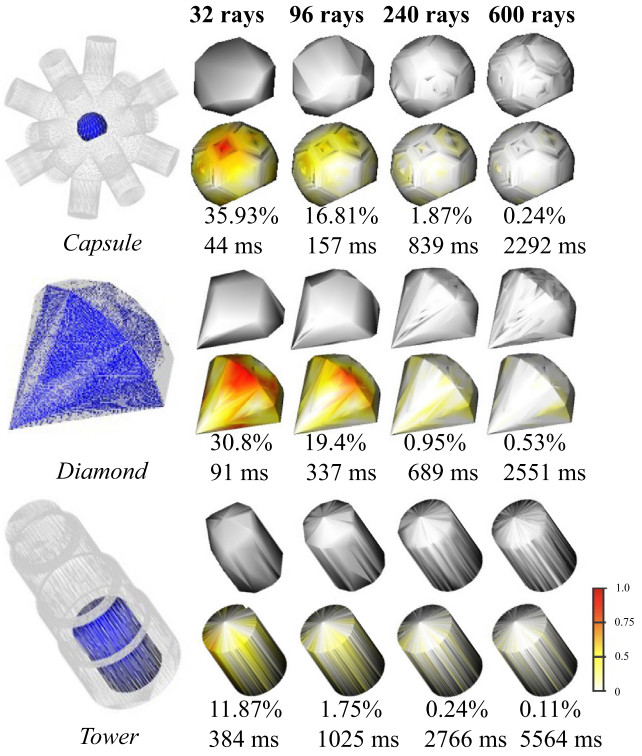


Fig. 10. Comparison of different ray counts. For each input shape, the first row displays the computed kernels, while the second row presents the normalized errors relative to the true kernel, with hot colors indicating larger errors.

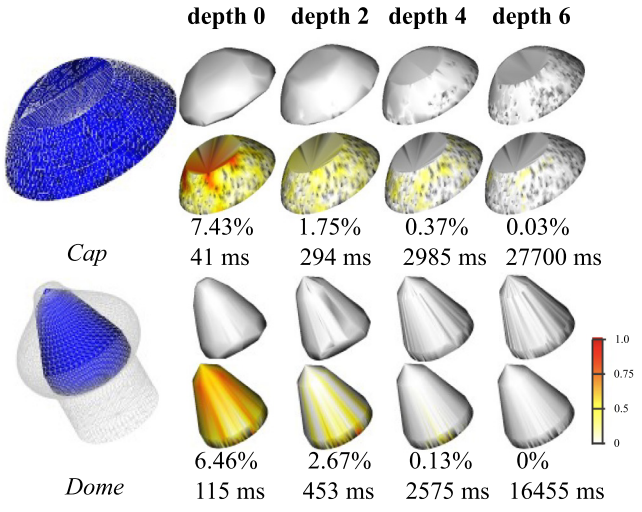


Fig. 11. Comparison of different recursion depths in vertex identification.

such as a more cornered kernel for Peg Top instead of a circular outline, a distorted output kernel for Stopper due to missing characteristic vertices, and a partial outline for Super Ellipse resulting from sparse ray sampling, which skips crucial endpoints of the kernel.

Additionally, experiments were conducted using both surface-geometry based rays and uniformly distributed rays via spherical distribution simultaneously. However, almost no discernible additional impact of the spherical distribution rays was observed beyond that of the surface-geometry based rays, as the latter already account for the geometric structure of the kernel.

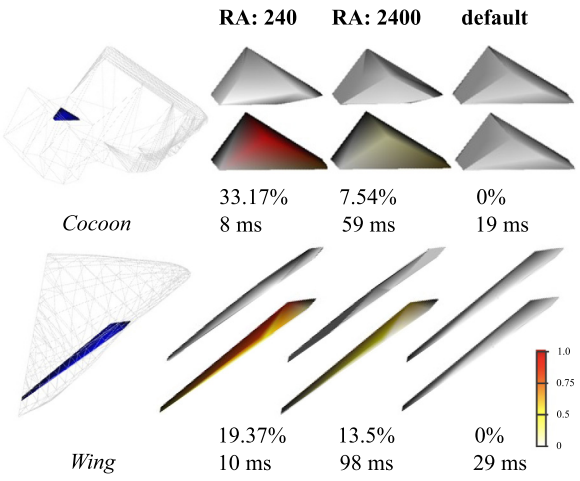


Fig. 12. Impact of the vertex identification. The approximation is significantly improved when the vertex identification stage is activated (left column vs. right column). It effectively captures details on the kernel boundaries, outperforming the version using more rays alone (middle column vs. right column).

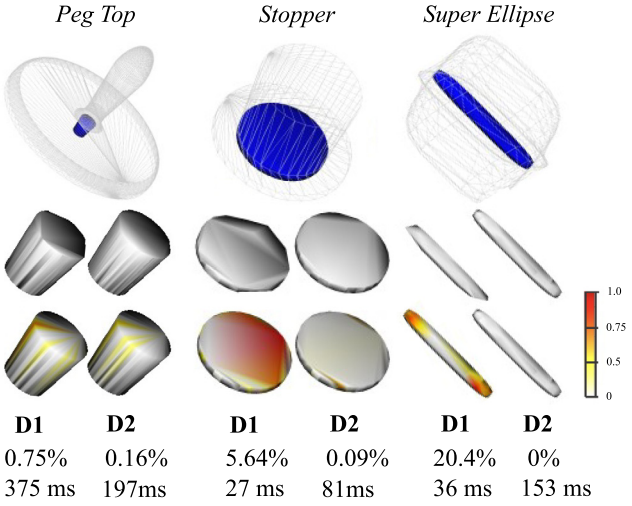


Fig. 13. Spherical (D1) vs. surface geometry-based (D2) ray distribution.

4.5. Evaluation

In terms of computational efficiency, our algorithm outperforms established methods like CGAL and Polyhedron Kernel as the number of mesh faces increases. However, when the number of rays and input mesh faces are relatively close, CGAL and Polyhedron Kernel are expected to run faster. This is because all three algorithms primarily involve plane-to-plane intersection operations, and a relatively close number of rays to the number of faces diminishes the computational efficiency of our algorithm by significantly increasing the number of intersections. This effect is more pronounced when the ray count serves as a larger coefficient in our algorithm, amplifying the impact of the number of faces, whereas the coefficient is constrained by the number of faces itself in other algorithms. We observed this phenomenon around 1000 faces in the *Thingi* dataset during our experiments. Up to nearly 1000 faces, the ratio between the number of faces and our fixed ray count (240) remains relatively constant. Consequently, our algorithm may lag behind CGAL and Polyhedron Kernel or show comparable performance. However, beyond 1000 faces, we observe significantly higher computational efficiency compared to the others. This observation aligns with the respective complexities of CGAL and

Polyhedron Kernel, which are $\mathcal{O}(N_f \log N_f)$ and $\mathcal{O}(N_f N_v)$, whereas ours is $\mathcal{O}(N_f N_r)$, where N_f, N_v and N_r are the number of faces, vertices and rays, respectively. To ensure more efficient computational performance compared to previous algorithms, users should keep the number of rays low relative to the number of mesh faces.

Also, our experiments have shown that our method accurately approximates the kernel shape. This close similarity can be attributed to two critical factors:

1. Surface geometry-based sampling: Our technique involves selecting triangles around the corners of the $AABB$ for surface sampling, effectively exploring boundary points. This approach aligns better with the geometry of the kernel's surface compared to spherical sampling.
2. Smart vertex identification: By topologically selecting candidate generator planes from nearby regions, we significantly enhance the probability of detecting a kernel vertex through their intersection. This leads to efficient vertex identification within a few attempts most of the time.

On the other hand, our algorithm's accuracy may decrease slightly when dealing with rounded-shaped kernels. Curved geometries require numerous nearly-coplanar faces to delineate their rounded surfaces, resulting in a multitude of plane intersections, which in turn leads to an increased number of vertices along their boundaries. Consequently, the recursive search in the vertex identification phase often fails to terminate early, leading to increased execution times. Additionally, our default search depth of three may prove insufficient to explore all kernel vertices, resulting in marginally higher volume losses.

In conclusion, with our algorithm's default settings of 240 rays and a recursive depth limited to three, we achieved high accuracy in both the *Thingi* and *Princeton* datasets. Users can intuitively adjust these settings to achieve their desired balance between accuracy and execution time, or to obtain an approximate kernel with their desired precision, a feature not available in other kernel computation algorithms.

5. Conclusion and future work

In this study, we introduce a novel kernel computation algorithm designed for complex star-shapes. The algorithm employs different techniques than the existing CGAL and Polyhedron Kernel algorithms. It begins by strategically sampling points on the kernel's surface via ray casting based on the kernel's surface geometry. Subsequently, it refines the approximation accuracy by identifying vertices on the surface of the kernel. The resulting output exclusively utilizes the interior or boundary points of the kernel. Our comprehensive analysis underscores the efficacy of the proposed solution. The experimental outcomes highlight the algorithm's remarkable performance in kernel computation across the *Thingi* and *Princeton* datasets, showcasing both high accuracy and efficiency, even for complex mesh structures. Especially for large-sized inputs, it outperforms existing methods, notably excelling in computational efficiency while maintaining a high level of approximation quality. Lastly, it instantly returns an empty kernel for non-star shapes without executing the entire process.

Our investigation into distinct ray tracing strategies and vertex identification methodology yields valuable insights into their influence on output accuracy and computational efficiency. These findings underscore the flexibility of our algorithm, which stands out as a distinguishing feature missing in the other methods, enabling users to adjust settings to achieve desired outcomes for scenarios requiring rapid computations or approximate solutions, such as dynamic shape quality measurement, mesh simplification, or decomposition.

Future improvements to this work will include analyzing the distribution of the kernel's surface at intermediate stages to strategically direct rays towards areas where greater geometric variation is expected. The objective is to enhance output accuracy without incurring additional computational costs, potentially through a parallel implementation [41] or a dynamic approach implementation such as the incremental convex hull algorithm [42]. Additionally, our future plans involve incorporating parallel computing strategies to enhance the algorithm's scalability. These improvements aim to further optimize the algorithm's performance, rendering it more adept at handling larger-scale datasets with increased efficiency. Moreover, our forthcoming endeavors will include extensive validation of the algorithm's applicability through rigorous testing in real-world applications, spanning domains like biomedical imaging and computer graphics. These validation exercises will serve to confirm the algorithm's utility in practical real-world scenarios.

CRediT authorship contribution statement

Merve Asiler: Writing – original draft, Visualization, Validation, Software, Methodology, Investigation. **Yusuf Sahillioğlu:** Writing – review & editing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: The second author reports financial support was provided by Scientific and Technological Research Council of Turkey.

Data availability

The data I utilized is publicly available, and I have properly cited its sources in my research.

Acknowledgments

This work was supported by TUBITAK (The Scientific and Technological Research Council of Turkey) under the project EEEAG-119E572.

References

[1] Kent JR, Carlson WE, Parent RE. Shape transformation for polyhedral objects. ACM SIGGRAPH 1992;26(2):47–54.
[2] Worman C, Keil JM. Polygon decomposition and the orthogonal art gallery problem. Internat J Comput Geom Appl 2007;17(02):105–38.
[3] Yu W, Li X. Computing 3d shape guarding and star decomposition. Comput Graph Forum 2011;30(7):2087–96.
[4] Yu W, Li M, Li X. Optimizing pyramid visibility coverage for autonomous robots in 3D environment. In: 2013 8th int. conference on computer science & education. IEEE; 2013, p. 1023–8.
[5] Schvartzman SC, Pérez ÁG, Otaduy MA. Star-contours for efficient hierarchical self-collision detection. ACM SIGGRAPH 2010;26(2):1–8.
[6] Wong SK, Lin WC, Hung CH, Huang YJ, Lii SY. Radial view based culling for continuous self-collision detection of skeletal models. ACM Trans Graph 2013;32(4):1–10.
[7] Shapira M, Rappoport A. Shape blending using the star-skeleton representation. IEEE Comput Graph Appl 1995;15(2):44–50.
[8] Sorgente T, Biasotti S, Manzini G, Spagnuolo M. The role of mesh quality and mesh quality indicators in the virtual element method. Adv Comput Math 2022;48(1):3.
[9] Sorgente T, Biasotti S, Manzini G, Spagnuolo M. Polyhedral mesh quality indicator for the virtual element method. Comput Math Appl 2022;114:151–60.
[10] Sorgente T, Biasotti S, Manzini G, Spagnuolo M. A survey of indicators for mesh quality assessment. In: Computer graphics forum, vol. 42. Wiley Online Library; 2023, p. 461–83.
[11] Sorgente T, Vicini F, Berrone S, Biasotti S, Manzini G, Spagnuolo M. Mesh quality agglomeration algorithm for the virtual element method applied to discrete fracture networks. Calcolo 2023;60(2):27.
[12] Subedi B. Generating kernel aware polygons [Ph.D. thesis], Las Vegas: University of Nevada; 2019.
[13] Gewali L, Subedi B. Random generation of visibility aware polygons. In: Int. conference on systems engineering. Springer; 2020, p. 151–9.
[14] CGAL Editorial Board. The computational geometry algorithms library. 2022, Sitewide ATOM URL https://www.cgal.org/.

[15] Sorgente T, Biasotti S, Spagnuolo M. Polyhedron kernel computation using a geometric approach. Comput Graph 2022.
[16] Shamos MI, Hoey D. Geometric intersection problems. In: 17th annual symposium on foundations of computer science. IEEE; 1976, p. 208–15.
[17] Preparata FP, Muller DE. Finding the intersection of n half-spaces in time O (n log n). Theoret Comput Sci 1979;8(1):45–55.
[18] Jacobson A, Panozzo D. Libigl: Prototyping geometry processing research in c++. ACM SIGGRAPH Asia 2017;1–172.
[19] Lee DT, Preparata FP. An optimal algorithm for finding the kernel of a polygon. J ACM 1979;26(3):415–21.
[20] Gómez F, Hurtado F, Ramaswami S, Sacristán V, Toussaint G. Implicit convex polygons. J Math Model Algorithms 2002;1:57–85.
[21] De Berg M, Van Kreveld M, Overmars M, Schwarzkopf O. Computational geometry. Springer; 1997, p. 1–17.
[22] Floater MS, Kós G, Reimers M. Mean value coordinates in 3D. Comput Aided Geom Design 2005;22(7):623–31.
[23] Natarajan S, Ooi ET, Saputra A, Song C. A scaled boundary finite element formulation over arbitrary faceted star convex polyhedra. Eng Anal Bound Elem 2017;80:218–29.
[24] Ooi E, Saputra A, Natarajan S, Ooi E, Song C. A dual scaled boundary finite element formulation over arbitrary faceted star convex polyhedra. Comput Mech 2020;66(1):27–47.
[25] Elber G, Johnstone JK, Kim MS, Seong JK. The kernel of a freeform surface and its duality with the convex hull of its tangential surface. Int J Shape Model 2006;12(02):129–42.
[26] Hong QY, Elber G. Detection and computation of conservative kernels of models consisting of freeform curves and surfaces, using inequality constraints. Comput Aided Geom Design 2022;94:102075.
[27] Avis D, Toussaint GT. An efficient algorithm for decomposing a polygon into star-shaped polygons. Pattern Recognit 1981;13(6):395–8.
[28] Chun S, Hong K, Jung K. 3D star skeleton for fast human posture representation. World Acad Sci Eng Technol 2008;2:2603–12.
[29] Etzion M, Rappoport A. On compatible star decompositions of simple polygons. IEEE Trans Vis Comput Graphics 1997;3(1):87–95.
[30] Hiraki T, Hayase T, Ike Y, Tsuboi T, Yoshiwaki M. Viewpoint planning of projector placement for spatial augmented reality using star-kernel decomposition. In: IEEE conference on virtual reality and 3D user interfaces abstracts and workshops. IEEE; 2021, p. 583–4.
[31] Keil JM. Decomposing a polygon into simpler components. SIAM J Comput 1985;14(4):799–817.
[32] Lee D, Lin A. Computational complexity of art gallery problems. IEEE Trans Inform Theory 1986;32(2):276–82.
[33] Kröller A, Baumgartner T, Fekete SP, Schmidt C. Exact solutions and bounds for general art gallery problems. J Exp Algorithmics 2012;17:2.3.2.1–23.
[34] Ghosh SK. Approximation algorithms for art gallery problems in polygons. Discrete Appl Math 2010;158(6):718–22.
[35] Amit Y, Mitchell JS, Packer E. Locating guards for visibility coverage of polygons. Internat J Comput Geom Appl 2010;20(05):601–30.
[36] Banerjee D, Inkulu R. Vertex guarding for dynamic orthogonal art galleries. Internat J Comput Geom Appl 2021;31(02n03):123–40.
[37] Bose P, Shermer T, Toussaint G, Zhu B. Guarding polyhedral terrains. Comput Geom 1997;7(3):173–85.
[38] Osada R, Funkhouser T, Chazelle B, Dobkin D. Shape distributions. ACM Trans Graph 2002;21(4):807–32.
[39] Supplemental material for the paper "A geometric approach for computing the kernel of a polyhedron" by T. Sorgente, S. Biasotti and M. Spagnuolo. — github.com. 2022, URL https://github.com/TommasoSorgente/polyhedron_kernel. [Accessed 29 August 2022].
[40] Chen X, Golovinskiy A, Funkhouser T. A benchmark for 3D mesh segmentation. ACM SIGGRAPH 2009;28(3).
[41] Stein A, Geva E, El-Sana J. CudaHull: Fast parallel 3D convex hull on the GPU. Comput Graph 2012.
[42] Acar UA, Blelloch GE, Tangwongsan K, Türkoğlu D. Robust kinetic convex hulls in 3D. In: 16th annual European symposium on algorithms, proc. 16. Springer; 2008, p. 29–40.

Merve Asiler is a research and teaching assistant and a Ph.D. Candidate in the Department of Computer Engineering at Middle East Technical University. She received her BS degree in the Department of Mathematics at Middle East Technical University with a double major in the Department of Computer Engineering and received her M.Sc. degree from the Department of Computer Engineering, Middle East Technical University, Turkey. Her research interests include computer graphics, digital geometry processing and discrete differential geometry. Contact her at asiler@ceng.metu.edu.tr or visit http://www.ceng.metu.edu.tr/~asiler.

Yusuf Sahillioğlu is a professor in the Department of Computer Engineering at Middle East Technical University. He is also an associate editor of The Visual Computer journal. His research interests include computer graphics and digital geometry processing. He has a Ph.D. in Computer Science from Koç University, Turkey. Contact him at ys@ceng.metu.edu.tr or visit http://www.ceng.metu.edu.tr/~ys.