DOI: 10.1111/cgf.15137

Eurographics Symposium on Geometry Processing 2024
R. Hu and S. Lefebvre
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 43 (2024), Number 5

KerGen: A Kernel Computation Algorithm for 3D Polygon Meshes

M. Asiler and Y. Sahillioğlu

Dept. of Computer Engineering, Middle East Technical University, Turkey
asiler@ceng.metu.edu.tr, ys@ceng.metu.edu.tr

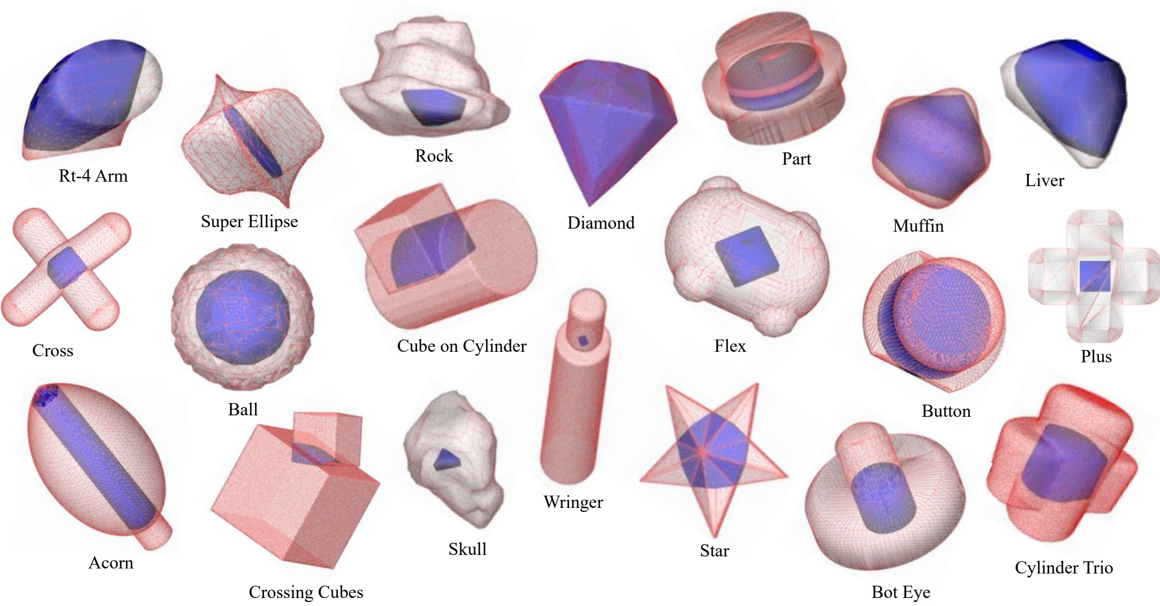


Figure 1: *A gallery of 3D kernel extraction results (blue) by our algorithm KerGen for a variety of input polygon meshes.*

Abstract

We compute the kernel of a shape embedded in 3D as a polygon mesh, which is defined as the set of all points that have a clear line of sight to every point of the mesh. The KerGen algorithm, short for Kernel Generation, employs efficient plane-plane and line-plane intersections, alongside point classifications based on their positions relative to planes. This approach allows for the incremental addition of kernel vertices and edges to the resulting set in a simple and systematic way. The output is a polygon mesh that represents the surface of the kernel. Extensive comparisons with the existing methods, CGAL and Polyhedron Kernel, demonstrate the remarkable timing performance of our novel additive kernel computation method. Yet another advantage of our additive process is the availability of the partial kernel at any stage, making it useful for specific geometry processing applications such as star decomposition and castable shape reconstruction.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Mesh geometry models;***

1. Introduction

The geometric kernel consists of points within or on the shape boundary that satisfy a crucial property: any line segment drawn from any point of the shape to any point within its kernel remains entirely inside the shape. In other words, all points of the shape are visible from any point within the kernel. If the kernel is non-empty, it includes the interior and/or the boundary points of the shape, which classifies the geometry as star-shaped. Convex ob-

© 2024 Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd.

delivered by EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

jects are inherently star-shaped, and the kernel of a convex shape corresponds to the shape itself. In Figure 1, kernels computed fully-automatically by our KerGen algorithm are demonstrated for various polygon meshes. Red lines are the wireframe overlaying over the gray shape boundaries, whereas blue regions are the kernels.

For 3D kernels, there are three use cases requiring (1) a kernel emptiness decision, (2) a single kernel point, and (3) an explicit kernel representation. Determining if a shape has a non-empty kernel is useful in applications such as star decomposition [YL11] and self-intersection detection [SPO10]. While these applications may involve a simple emptiness check, the first use case can extend to shape guarding [YLL13], and the second can be employed in collision-free deformations [SR95; WLH*13] by searching for an arbitrary kernel point. An injective planar mapping [Liv24] can also be performed over a single kernel point. Although spherical parameterization [KCP92] requires a single kernel point, the process can be repeated with different kernel points to minimize angle distortion. Similarly, polar refinement in remeshing [SDG*19] may benefit from various kernel point initializations. Moreover, explicit kernel representation is useful in mesh quality assessment [SBMS22], with applications in finite element [PCS*22] and virtual element [BBC*13] methods, mesh generation and refinement [SBMS23], and mesh quality assessment [SVB*23] algorithms. Additionally, in a robot target tracking system [MB23], navigation computations can be performed using the explicit kernel of the area of motion.

In this paper, we propose KerGen, abbreviation for Kernel Generation, as a novel and efficient algorithm to compute the explicit kernel representation of a 3D shape. The idea behind our algorithm centers on recovering the kernel boundary by identifying the closest plane intersections from a reference kernel point, thereby minimizing unnecessary plane intersections outside the kernel. Briefly, we determine an initial 3D kernel point automatically, then we iteratively find the locations of the edges and vertices of the output mesh corresponding to the kernel's boundary surface. Essentially, we generate the kernel boundary by identifying the adjacencies of the current kernel edges and vertices at each iteration. Geometric predicates [She97] are employed for classifying point positions.

Our algorithm has several appealing aspects compared to recent works such as the CGAL (Computational Geometry Algorithms Library) algorithm [FP09] and the Polyhedron Kernel algorithm [SBS22a], the only two methods for computing the 3D kernel. First, it relies on efficient plane intersections with other planes and lines, as well as the analysis of point positions relative to planes, leading to significantly faster execution times compared to existing methods. Second, it allows kernel generation from any point within the kernel. From a reverse perspective, it provides flexibility in specifying any desired point as a kernel point and computing the kernel which corresponds to the desired visibility coverage. Assuming the total shape body is covered by the set of such kernels, this feature could offer novel solutions to computing robot motion coverage and navigation based on the specific guards as well as designing kernel-aware shapes. Third, unlike existing subtractive methods, KerGen adopts an additive approach. This ensures that the partially generated shape at each iteration always remains within the kernel and allows for exploring the details of the kernel's geometry structure during generation. Consequently, users can obtain partial kernels or make compatible modifications to the kernel and input mesh geometry at any stage. This adaptability renders KerGen well-suited for future tasks such as star decomposition of shapes or castable shape reconstruction from rigid molds.

We demonstrate the efficiency of our method on various groups of meshes: (i) the ones having a non-empty kernel, i.e., star-shapes and (ii) remeshed versions of some meshes in (i). Additionally, we provide an experimental study for (iii) meshes with an empty kernel, i.e., non-star shapes. Comprehensive experiments in Section 5 reveal our state-of-the-art performance in comparison with the other existing methods.

2. Related Work

Kernel Computation. Kernel computation is one of the less-explored topics in digital geometry processing. First study by Shamos and Hoey assert that the kernel of a polygon can be found in $\mathcal{O}(N_e \log N_e)$ time where N_e is the number of edges [SH76]. Later, Preparata and Muller prove the previous claim in 3D by suggesting an algebraic solution in $\mathcal{O}(N_f \log N_f)$ time where N_f is the number of faces [PM79]. Their method has been integrated into various well-known libraries, including CGAL [FP09] and Libigl [JP17]. Recently, Sorgente et al. address the problem from a geometric perspective [SBS22a] with a faster computation time than the algebraic one. Their algorithm takes $\mathcal{O}(N_f N_v)$ time where N_v is the number of vertices in the input mesh. Unlike existing parameter-free approaches, including ours, which produce the kernel itself, Asiler and Sahillioğlu [AS24] approximate the 3D kernel of a given polyhedron by fine-tuning two parameters: ray count and recursion depth. Their method operates in expected time $\mathcal{O}(N_f N_r)$, where N_r is the number of rays. There also exist some 2D studies that can compute the kernel in $\mathcal{O}(N_e)$ time. In [LP79], Lee and Preparata deduce the shape of the kernel by computing the reflex and convex angles between consecutive edges. Moreover, in [Sac19], a similar discussion is made over line positions in 2D to obtain the extreme kernel point in a given direction. Mainly, they use the randomized algorithm presented by Seidel [Sei91] to solve the linear programming constrained by line equations. For the same purpose, Berg et al. solve a linear equation system in 2D [DVOS97], yet they generalize their idea to all dimensions.

Kernel Analysis. Given the kernel of a shape, finding out properties of the kernel is another topic of interest. In their study, Gardner and Kallay reveal the face properties of the kernel boundary by mathematical proofs [GK92]. In [FKR05], Floater et al. define mean-value coordinates of the star polyhedra over the equations constructed on the properties satisfied by kernel. Additionally, there exist studies proposing new finite element methods which are based on the kernel of a polyhedron [OSN*20]. Other than that, given a point set, the description of being a kernel point is used in order to generate the polygon that accepts the largest kernel [Sub19; GS20].

Star-Shape Decomposition. Another kernel-related problem is star shape decomposition, which deals with partitioning the object into pieces with non-empty kernels. In addition to the studies that perform star decomposition on a given single polygon [AT81; FM84; CHJ08], there is also a method that handles the problem in a more complex setting where compatible decomposition between

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

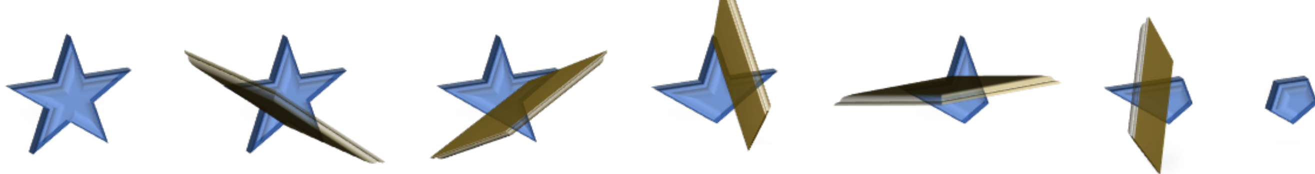


Figure 2: *An illustration of kernel extraction from the bounding planes of the Star model. Kernel is shown at the rightmost column.*

two polygons are sought [ER97]. Keil et al., on the other hand, aim to obtain such a decomposition with minimum number of pieces [Kei85]. An associative problem to this one is familiarly known as the *Art Gallery Problem* which is NP-Hard [LL86]. Yu et al. perform star decomposition in 3D with the purpose of shape guarding on meshes [YL11; YLL13].

Usage of Kernel. Geometric kernels find extensive application across various domains. The determination of kernel emptiness is crucial for detecting self-intersections within shapes [SPO10]. Similarly, to ensure collision-free deformation, input shapes are interpolated using a function that maps mesh vertices to unique angle-distance tuples computed from a single kernel point [SR95; WLH*13]. This approach can also be extended to transform star-shapes into spherical [KCP92] and planar [Liv24] parameterizations. Moreover, in spherical parameterization, different kernel points may be utilized to minimize angle distortion, while polar refinement in remeshing [SDG*19] could benefit from various kernel point initializations. Furthermore, explicit representation of the kernel allows for assessing shape quality, particularly concerning the object's kernel size in finite element [PCS*22] and virtual element [BBC*13] methods. Sorgente et al. propose a metric quantifying the ratio of kernel area/volume to total shape area/volume [SBMS22], which is evaluated for mesh generation and refinement processes [SBMS23]. Their recent work [SVB*23] introduces a novel agglomeration algorithm that optimizes an energy function utilizing the aforementioned kernel-based metric. Also, robot target tracking systems rely on the explicit kernel for the navigation computation [MB23]. Lastly, explicit kernel computation may find application in determining the castability of a shape, as both problems share a common algebraic framework [DVOS97]. Stein et al. illustrate this with a castable shape design algorithm in [SJG19].

3. Preliminaries

3.1. Kernel Basics

Definition 3.1 Formally, kernel K of a shape S is defined as follows:

$$K = \{p \in S \mid (1-t)p + tq \in S, \;\; \forall q \in S, \;\; \forall t \in [0,1]\}.$$

Corollary 3.1 If K is a non-empty set, then K has a convex shape.

Definition 3.2 S is called star-shape, if its K is a non-empty set.

Corollary 3.2 All convex shapes are star-shapes.

Corollary 3.3 All star-shapes are connected.

Corollary 3.4 For a star-shape S, its K is connected.

Theorem 3.1 For a polyhedral shape S with the boundary ∂S, let H_F be the half-space whose boundary is the plane carrying the face F of ∂S. Let also H_F depict the inner side of ∂S. Then kernel of S can be found by taking the intersections of all H_F $\forall F$. That is, $K = \bigcap_{F=1}^{N} H_F$, where N is the number of faces.

Proof See [PM79]. □

In the rest of the paper, we call those half-spaces as **the bounding half-spaces** and its boundaries as **the bounding planes**. Figure 2 illustrates how kernel of the given 3D shape is found by intersecting the bounding planes.

Obviously, in 3D space, when we take the intersection of three non-parallel planes, we get a single point provided that the planes are not coinciding along a line. Then, by Theorem 3.1, intersection of some of the bounding planes gives vertices of the kernel. However, it is not trivial to decide which planes should be intersected to obtain the kernel vertices and also there are in total $\binom{N_f}{3}$ possible plane combinations to check for intersection where N_f is the number of faces of the input mesh. Hence, it takes $\mathcal{O}({N_f}^3)$ time which is highly demanding. Our output-sensitive algorithm, on the other hand, requires only $\mathcal{O}(N_f K_e)$ time where K_e is the number of edges in the output kernel mesh.

Lemma 3.1 Since each half-space is represented by an inequality, the set of all bounding half-spaces of a mesh constructs an inequality system. As a result, Theorem 3.1 yields to the fact that all points of the kernel must satisfy this inequality system.

3.2. Point Classification

In our algorithm, we classify points obtained through intersection operations based on their positions relative to boundary planes. These planes are defined by using vertices of the polygons on the input mesh. A point is labeled as `BELOW` or `ABOVE` the plane P depending on whether it lies on the side where the normal of P points or on the opposite side, respectively. If a point lies on the plane P, it is labeled as `INTER`.

To perform the labeling, we utilize Shewchuk's geometric predicates [She97] . The labels `BELOW`, `ABOVE`, and `INTER` are determined based on the output of Shewchuk's predicate function `orient3d()` provided by CinoLib [Liv19]. It is worth noting that in their Polyhedron Kernel algorithm [SBS22a], Sorgente et al. also employ the same labeling system for the points computed by the similar intersection operations.

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

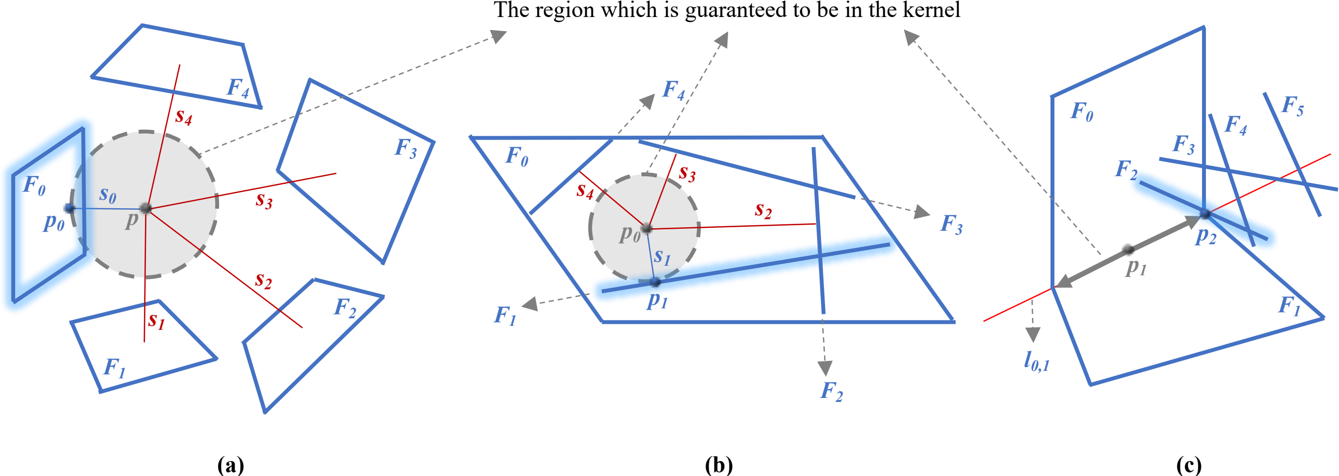


Figure 3: *The process of determining the initial faces whose intersection yields a kernel vertex: Firstly, we identify the closest plane to p, which becomes our initial plane, denoted as F_0. We then project p onto the selected plane, resulting in point p_0 (a). We proceed by intersecting F_0 with each of the other planes one-by-one. From the resulting lines, we choose the one closest to p_0, which becomes F_1. Similarly, we project p_0 onto the intersection line $l_{0,1}$ of F_0 and F_1, obtaining point p_1 (b). Lastly, we find the closest plane to p_1 on $l_{0,1}$, which is F_2, and the intersection point of F_2 and $l_{0,1}$, denoted as p_2 (c). Note that the final point p_2 is also a vertex on the kernel surface.*

4. Method

4.1. Theoretical Background

KerGen is designed to identify the closest boundary elements of the kernel by iteratively selecting them from various reference kernel points. Assuming that p is a point located inside the kernel, it is possible to find an initial face belonging to the kernel in 3D using the following approach.

Since the kernel is defined as the intersection of bounding half-spaces (as stated in Theorem 3.1), the faces of the kernel surface must lie on the planes that act as boundaries to these half-spaces. Therefore, we can select the initial face F_0 as the plane closest to p by calculating the perpendicular distance from p to each bounding plane. It is important to note that the plane itself is not considered a face, but it is tangent to the kernel. For simplicity, we refer to this plane as a face.

Lemma 4.1 The point obtained by projecting p to F_0 is guaranteed to be on the kernel boundary.

Proof For a point to be inside the kernel, we require it to satisfy the inequality system of the bounding half-spaces. Let the perpendicular distance from p to F_0 be s_0. We already know that s_0 is the shortest distance among the distances to other faces. That means, whichever direction we go from p in a distance smaller than s_0, we cannot reach any bounding planes since the shortest distances to them are certainly higher than s_0. That means, when we go for a distance of s_0 and in the direction of the normal of F_0 (which gives the perpendicular direction), we still remain inside the kernel, i.e., satisfy the inequality system. □

According to Lemma 4.1, we can deduce that the projection of p onto F_0, denoted as p_0, is also a point within the kernel. Utilizing this insight, we can apply a similar technique to identify a second face that is tangent to the kernel. By projecting all elements onto F_0 and reducing the problem to a 2D space, we can proceed as follows:

After computing the perpendicular distance from p_0 to the intersection line of each bounding plane with F_0, we find that the projection of p_0 onto the line with the minimum distance lies within the kernel. Let this projected point be denoted as p_1, the corresponding plane as F_1, and the line resulting from the intersection of F_0 and F_1 as $l_{0,1}$. At this point, we can apply the 1D case of Lemma 4.1 to $l_{0,1}$ and p_1.

Since the next point we seek is the intersection of three planes, namely F_0, F_1, and F_2, which represents the closest plane to p_1 along $l_{0,1}$, we are able to determine an initial vertex on the surface of the kernel. By applying the 1D version of Lemma 4.1 to the previously identified kernel elements, we can easily determine the remaining kernel vertices, edges, and faces. Figure 3 provides a visual representation of the process for finding F_0, F_1, and F_2.

4.2. Algorithm Description

In the beginning of the process, we derive the equations of the bounding planes (and consequently bounding half-spaces) by utilizing the polygons comprising the surface of the input mesh. These equations are essential for conducting computations involving plane-plane and line-plane intersections. Additionally, we retain the vertices of each polygon associated with the planes to facilitate predicate checks, enabling us to determine the positions of points relative to these planes accurately.

Our algorithm requires an initial kernel point regardless of its location, which is then snapped to the closest kernel vertex on the kernel surface. We can reach such a kernel vertex from the initial

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

kernel point by applying all versions of Lemma 4.1 successively from 3D to 1D. This approach is particularly useful in scenarios where the kernel of the shape, or a segment of it, is desired to be formed around a specific point of the shape. In our implementation, we begin by determining an initial kernel vertex using Berg et al.'s incremental linear programming solution [DVOS97], which finds the extreme kernel point in a given direction. Although multiple extreme points may exist in the specified direction, the point identified is guaranteed to be a kernel vertex, as the algorithm selects the first one based on lexicographical order in all coordinates. Our empirical observations indicate no significant variation in computation time based on the direction of the vector used to find the extreme point. Therefore, we consistently employ the same direction vector, $(0,0,1)$, for all tests. Once the desired kernel vertex is identified, its generator planes are easily detected based on the `INTER` label returned by Shewchuk's predicate function (refer to Section 3.2).

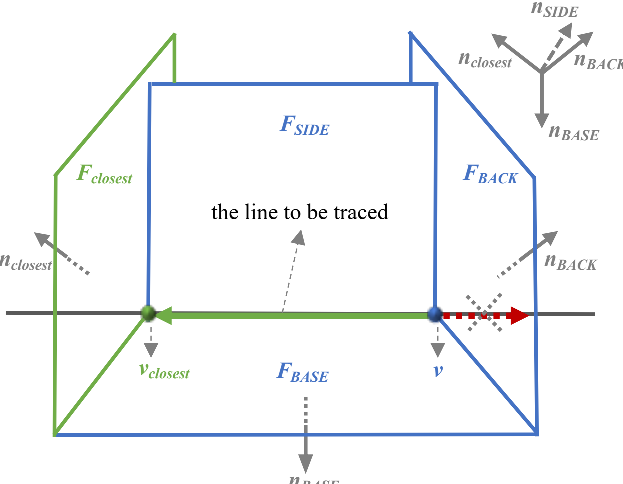


Figure 4: *Illustration of labeling the generator planes of a vertex v before tracing the line started from it. The correct tracing direction is indicated by green, which points to the* `ABOVE` *side of* F_{BACK}. *We identify the closest plane to v along the line, whose intersection is either* `ABOVE` *or* `INTER` *all other planes intersecting the line.*

Next, we proceed to find the other vertices on the kernel surface by tracing along the lines that represent the kernel edges. Given a kernel vertex v and its associated generator planes, our objective is to identify the additional vertices of the kernel that lie on the intersection lines of these generators. Initially, we assume that a vertex is associated with only three generator planes, while addressing the exceptional cases of having more than three generators at a later stage. It is important to note that each distinct pair of generators intersect at a line. Based on the choice of which line to track, we typically designate its generators as F_{BASE} and F_{SIDE}, while the remaining generator of v is referred to as F_{BACK}. The setup is illustrated in Figure 4.

During the process of tracing along a line, we make use of 1D version of Lemma 4.1. Specifically, we search for the plane whose intersection point with the line is closest to the vertex v and lies on the `ABOVE` side of F_{BACK}. The determination of the intersection point's position with respect to F_{BACK} is achieved through the utilization of Shewchuk predicates. Furthermore, to find the closest intersection point, we rely on the predicates rather than comparing distances: At each step of the iterative process, where the intersection of the line with a new plane is considered, we perform a check to determine if the previously computed intersection point is `ABOVE` or `INTER` the current plane. If it is, we can skip the computation of the intersection point between the current plane and the line, as it would have a larger distance from v. Conversely, in the case where the previously computed point is `BELOW` the current plane, we update the previous point with the new intersection point obtained by intersecting the current plane with the line. This process allows us to come up with the other kernel vertex on the line.

Algorithm 1: LINETRACKING

Input: An initial kernel vertex: v,
The set of generator planes of v: $\{F_v\}$
The set of bounding planes: $\{F\}$.
Output: The tuple of $< E_K, V_K >$ where E_K is the set of kernel edges and V_K is the set of kernel vertices.

```
1  begin
       // identify adjacent lines to v
2      Q := IdentifyLines(v, {F_v}, E_K)
3      while Q ≠ ∅ do
4          Line l ← Pop Q
5          F_BASE, F_SIDE, F_BACK ← l.planes
6          v ← l.vertex
7          {F_v} ← v.generators
8          v_closest := null , {F_closest} := ∅
9          foreach F_i ∈ {F}/{F_v} do
               // define initial closest vertex
10             v_i := F_i ∩ l
11             if v_i ≠ null and v_i is ABOVE F_BACK then
12                 v_closest ← v_i
13             end
               // update closest elements
14             if v_closest ≠ null then
15                 if v_closest is INTER F_i then
16                     {F_closest} ∪= {F_i}
17                 end
18                 if v_closest is BELOW F_i then
19                     v_closest := F_i ∩ l
20                     {F_closest} ← {F_i}
21                 end
22             end
23         end
           // update the sets, find new lines
24         if v_closest ∉ V_K then
25             Add v_closest into V_K
26             Q ∪= IdentifyLines(v_closest, {F_closest}, E_K)
27         end
28         Edge e :=< v, v_closest >, e.line ← l,
29         Add e into E_K
30     end
31     return < E_K, V_K >
32 end
```

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

Upon the discovery of a new vertex, additional lines are introduced, which guide the subsequent exploration. To facilitate this process, we employ a queue data structure to store the newly discovered lines. Each line is associated with the new vertex, as well as the corresponding planes: F_{BASE}, F_{SIDE}, and F_{BACK}. For each line popped from the queue we repeat the above process until the queue becomes empty. Note that each line should be added into the queue only once, otherwise the same lines are met over and over again and the queue never becomes empty. To determine whether the two lines are the same, we check if their generator planes, F_{BASE} and F_{SIDE}, are identical instead of checking their equations or endpoints to prevent false detections due to numerical inaccuracies. Given that multiple instances of a plane can exist in the bounding plane set, we use predicate checks involving the defining vertices to verify the sameness of the planes. Algorithm 1 gives the pseudo code of the line tracking procedure.

An important consideration is how to determine which lines to follow next and select appropriate F_{BASE}, F_{SIDE}, and F_{BACK} planes when there are more than three generators for the vertex v. We employ the following strategy to ensure correct line and plane assignments: For each candidate pair for F_{BASE} and F_{SIDE}, we designate a test point on the next candidate line that they generate, on both the upper and lower sides from v. Subsequently, we examine both test points to determine whether they are located on the `ABOVE` or `INTER` sides of the other generator planes, utilizing the Shewchuk predicates. If either of the points does not fall on the `BELOW` side of any of the other generators, then the selected pair of planes and their common line are considered valid, and the planes are labeled as F_{BASE} and F_{SIDE}. It is important to note that although the test points used may not belong to the kernel, the selected line contains a kernel edge (Lemma 4.2).

Lemma 4.2 The line selected according to the above specifications contains a kernel edge.

Proof Since v is a kernel vertex, the kernel edges adjacent to v must be intersection lines of some of the generator planes of v. Assume l is one of those lines adjacent to v. Then any other bounding plane which is not a generator of v already includes some portion of l started from v. This follows from the fact that the corresponding half-space of any bounding plane which does not pass through v already contains some volume around v, including some segment of l. Therefore, it is sufficient to check the properties of l with respect to only the generator planes of v to understand whether it contains a kernel edge. Note that any plane passing through v either aligns with the line l itself or intersects l at v. This means that at least one of the two sides of l starting from v completely resides in the corresponding half-spaces of the generator planes of v, validating predicate tests for any test point on each side. □

It is also possible that no plane pairs could be labeled as F_{BASE} and F_{SIDE}. This situation occurs when the kernel boundary ends at the current vertex because the resulting kernel is not a polyhedron, but rather consists of a single point, a single line segment, or a single polygon. Additionally, there may be more than one pair of planes suitable for the F_{BASE} and F_{SIDE} labels, indicating that the current kernel vertex has a degree larger than three. In such cases, all identified pairs are evaluated as lines spanning a kernel edge.

Next, we need to choose a face as F_{BACK} from among the remaining generator planes. Any generator plane could be used as F_{BACK} provided that only one of the two test points is located on the `ABOVE` side of it whereas the other is located on the `BELOW` side. Once the selection is complete, we add the computed lines into the queue. However, we ensure that the lines being added have not been previously pushed into the queue. Figure 5 illustrates the line selection process for the two possible types of plane neighborhoods sharing vertex v. Also, Algorithm 2 gives the pseudo code of the line selection and plane labelling procedure.

Algorithm 2: IDENTIFYLINES

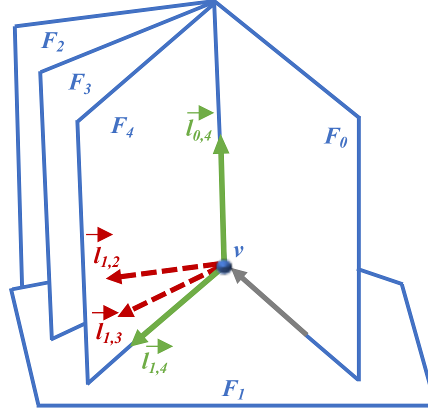
Input: A kernel vertex: v,
The set of generator planes of v: $\{F_v\}$
The set of kernel edges: $\{E_K\}$.
Output: The yet-to-be-traced lines adjacent to v: Q_v

1 **begin**
2 $Q_v := \emptyset$
3 **foreach** $F_i, F_j \in \{F_v\}$ **do**
4 $l_{i,j} := F_i \cap F_j$
`// ensure` $l_{i,j}$ `is not revisited`
5 **if** $l_{i,j}$ is not a line **or** $l_{i,j} \in Q_v$ **or** $\exists e \in E_K$ on $l_{i,j}$ **then**
6 **continue**
7 **end**
8 $F_{BASE} := F_i$, $F_{SIDE} := F_j$, $F_{BACK} :=$ null
`// define test points` p_1`,` p_2 `on` $l_{i,j}$
9 $\overline{F}$:= The plane with the following properties: $l_{i,j} \perp \overline{F}$ and v is `INTER` $\overline{F}$
10 p_1 := A point $p \in l_{i,j}$ such that p is `ABOVE` $\overline{F}$
11 p_2 := A point $p \in l_{i,j}$ such that p is `BELOW` $\overline{F}$
12 $p_1.sign :=$ true, $p_2.sign :=$ true
`// determine` $p_1.sign$`,` $p_2.sign$ `and` F_{BACK}
13 **foreach** $p_{test} \in \{p_1, p_2\}$ **do**
14 **foreach** $F_k \in \{F_v\}$ **do**
15 **if** p_{test} is `BELOW` F_k **then**
16 $p_{test}.sign \leftarrow$ false
17 $F_{BACK} \leftarrow F_k$
18 **break**
19 **end**
20 **end**
21 **end**
`// decide on` $l_{i,j}$ `to trace`
22 **if** $p_1.sign \oplus p_2.sign$ **then**
23 $l_{i,j}.planes \leftarrow F_{BASE}, F_{SIDE}, F_{BACK}$
24 $l_{i,j}.vertex \leftarrow v$
25 Push $l_{i,j}$ into Q_v
26 **end**
27 **end**
28 **return** Q_v
29 **end**

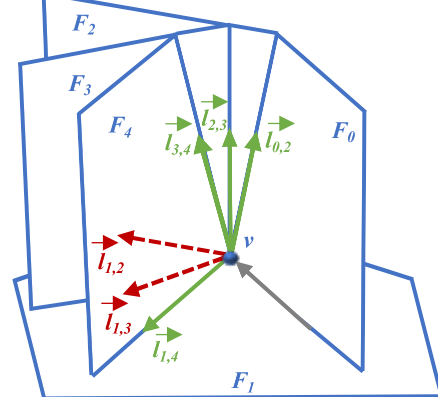
Lemma 4.3 The proposed method accurately computes the kernel.

Proof The output mesh consists of vertices and the edges connecting them, all of which are verified to belong to the kernel via predicate checks. Thus, the produced shape is a part of the kernel. Moreover, due to the connectedness of the kernel of a star-shape, the resulting mesh is guaranteed to include all the boundary vertices

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.



(a) Planes intersecting along a line



(b) Planes coinciding at a single point

Figure 5: *In the scenarios where the vertex v has more than three generator planes, gray arrows represent incoming traversal at v, while red and green arrows indicate candidate next lines to follow. In both examples, the lines indicated by green arrows are the correct lines to trace, while those indicated by red arrows are not. To illustrate F_{BACK} determination, in both scenarios, F_0 can chosen for line $l_{1,4}$, while F_1 can be selected for lines $l_{0,4}$ (a) and $l_{3,4}$, $l_{2,3}$, and $l_{0,2}$ (b).*

and edges of the kernel. This can be summarized as follows: The algorithm begins with a kernel vertex and iteratively identifies the remaining kernel vertices and edges by traversing along the kernel's boundary lines. At each vertex, the algorithm identifies all the kernel edges adjacent to it. Subsequently, each edge is traversed exactly once until reaching the vertex at the opposite end, thus ensuring prevention of infinite loops over previously traversed edges. Therefore, the resulting mesh accurately captures the geometry of the kernel itself. □

Finally, Figure 6 illustrates the formation of the kernel on *Star* and *Ball* models with the screenshots taken at some intermediate stages.

4.3. Complexity Analysis

The algorithm starts with selection of an initial kernel vertex utilizing Berg et al.'s $\mathcal{O}(N_f)$ expected time algorithm where N_f is the number of faces on the input mesh. Next, the algorithm proceeds with LINETRACKING module in which we traverse on each line containing a kernel edge (Algorithm 1). To identify the lines passing through a kernel vertex v and assign appropriate labels to its generator planes, we call IDENTIFYLINES module (Algorithm 2). In most cases, v has exactly three generator planes, enabling efficient operations in constant time. However, if v has more than three generators, we need to check each pair of planes that intersect at a line for unprocessed edges. This is accomplished by evaluating the positions of two test points on the line relative to the other generator planes. Although checking the intersection of each pair of planes results in quadratic complexity, we observe that this part is amortized in constant time since the number of generator planes for v is much smaller than the total number of planes in the input mesh. Consequently, the practical impact of this operation is negligible.

While tracking on a line, we search for the closest plane to v. As long as a data structure like a kd-tree is not used to define locations of the planar sections, this operation requires labeling point positions with respect to each plane, which is $\mathcal{O}(N_f)$. Consequently, the time complexity for tracking a single line is $\mathcal{O}(N_f)$. Since the line walking operation is executed exactly once for every edge of the kernel mesh, the computational complexity of LINETRACKING module results in $\mathcal{O}(N_f K_e)$ where K_e is the total number of kernel edges. As a result, the computational complexity of our KerGen algorithm is output-sensitive and equal to $\mathcal{O}(N_f K_e)$.

While it is not feasible to provide an exact formula for the relationship between K_e and N_f (or N_v, the number of vertices on the input mesh) to compare the complexity of KerGen with those of CGAL ($\mathcal{O}(N_f \log N_f)$) and Polyhedron Kernel ($\mathcal{O}(N_f N_v)$), we can roughly estimate an upper limit using Euler's formula. This estimation is based on the assumption that all polygonal faces of the mesh are subdivided into triangles. Since the kernel of an input mesh is the intersection of the bounding half-spaces, its boundaries may consist of at most N_f faces, resulting in half as many vertices and 1.5 times as many edges. Although this imposes an upper limit that turns into quadratic based on N_f for both Polyhedron Kernel and KerGen, our empirical analysis, which is based on the star-shapes from the *Thingi* dataset [SBS22b] and the *Princeton* dataset [CGF09], shows that in the majority of input shapes, the value of K_e is significantly smaller than N_f, as evidenced by Table 1 and Figure 8. This characteristic of K_e allows for efficient performance of KerGen.

5. Experimental Results

In this section, we conduct qualitative and quantitative evaluations of the proposed method to comprehensively demonstrate its effectiveness. We implement and experiment on a computer with a 2.20GHz i7 CPU and 16GB memory. We also provide the source

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

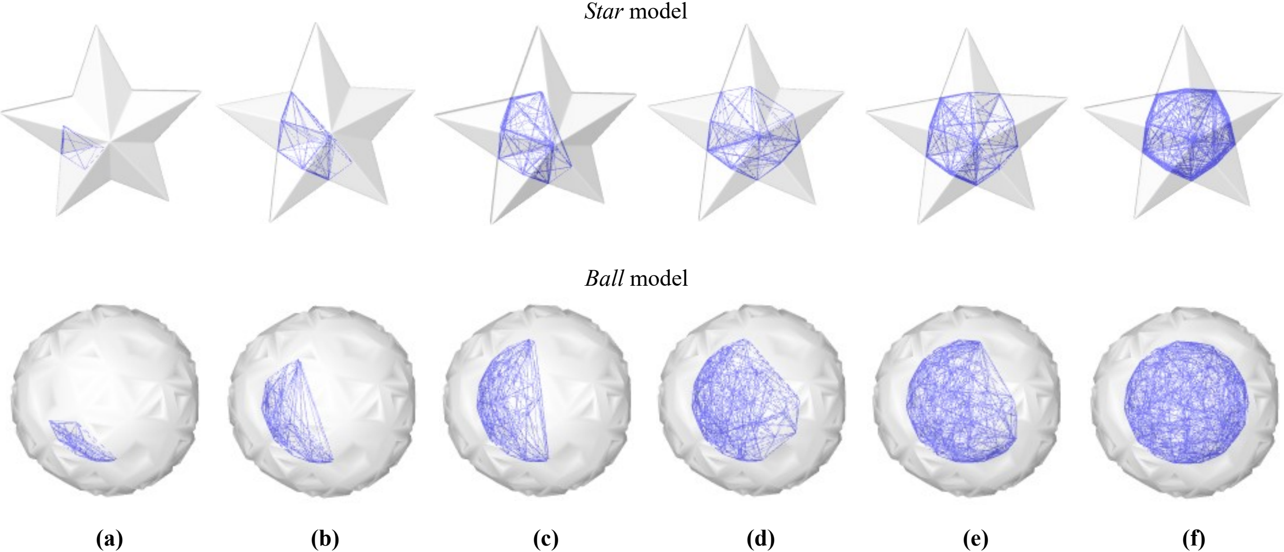


Figure 6: *(Top) Formation of the kernel of the Star model. The figure shows the snapshots taken during the construction of the kernel at the 12th (a), 30th (b), 60th (c), 85th (d), 115th (e) and 165th (f) edge. (Bottom) Similar formation is shown for the Ball model where little cavities over the surface affects the kernel drastically. Final kernels of these models can also be seen in Figure 1.*

code in C++ at `https://github.com/merveasiler/Geometric-Kernel-Generation`, along with a supplementary video for further qualitative assessment.

We compare the running time performance of KerGen with both CGAL [FP09] and Polyhedron Kernel [SBS22a]. In order to compute kernels, CGAL uses the half-space intersection functionality that is based on the algebraic approach given in [PM79]. Using the fact that convex hull and half-space intersection problems are duals of each other, it solves the inequality system constructed from the input mesh boundaries by computing the convex hull of the sets obtained through dual transformations. On the other hand, Polyhedron Kernel uses a geometric subtractive approach which produces the kernel after clipping an initial candidate box with each bounding plane one-by-one.

In performance comparisons of all three algorithms, the results demonstrated correspond to equivalently configured implementations. For CGAL's algorithm, its `halfspace_intersection_with_constructions()` function is invoked with half-spaces defined by explicit inequalities using `CGAL::Simple_cartesian<double>`, contributing to its efficient performance. Similarly, for both Polyhedron Kernel and KerGen, which utilize the same algorithmic tools of plane intersections and point identifications, a comparable implementation to CGAL's is employed, utilizing inexact constructions with limited precision and CinoLib's inexact predicates [Liv19]. However, we further discuss the performance and accuracy variations when employing other exact or implicit constructions along with exact predicates in Section 6.2.

We evaluate the algorithms using the *Princeton* [CGF09] and *Thingi* [SBS22b] datasets. The latter is a subset of the *Thingi10K* [ZJ16] dataset, filtered to include only one-connected closed manifold input meshes with both empty and non-empty finite kernels. We conduct experiments on three types of data: star-shapes, non-star shapes, and remeshed star-shapes. The timing results represent the average of ten successive runs, ensuring statistical reliability. Our experiments clearly demonstrate that KerGen yields accurate results and outperforms both CGAL and Polyhedron Kernel across the *Princeton* and *Thingi* datasets.

5.1. Comparison Over Star-Shaped Meshes

As star-shaped data, we use the so-called *Acorn* (ThingiID: 815480), *Ball* (ThingiID: 58238), *Bot Eye* (ThingiID: 37276), *Button* (ThingiID: 1329185), *Cross* (ThingiID: 313882), *Diamond* (ThingiID: 313917), *Flex* (ThingiID: 827640), *Muffin* (ThingiID: 101636), *Part* (ThingiID: 472063), *Plus* (ThingiID: 1120761), *Rt-4 Arm* (ThingiID: 39353), *Star* (ThingiID: 313883) and *Super Ellipse* (ThingiID: 40172) from *Thingi* dataset and *Crossing Cubes* (PrincetonID: 325), *Cube on Cylinder* (PrincetonID: 326), *Cylinder Trio* (PrincetonID: 337) and *Wringer* (PrincetonID: 350) from *Princeton* dataset. The visuals of the kernels computed by KerGen is given in Figure 1. The blue volume inside the meshes represent their kernels. Also, the execution times of all algorithms are given in Table 1.

For the majority of the input star-shaped meshes listed in Table 1, KerGen exhibits significantly faster kernel computation compared to CGAL and Polyhedron Kernel. However, in the cases of the *Liver*, *Acorn*, *Ball*, *Muffin*, and *Star* meshes, KerGen falls slightly behind the other algorithms in terms of timing. This discrepancy can be attributed to the presence of numerous edges on the rounded boundary of the kernels for meshes like *Acorn*, *Ball*, and *Muffin*, as well as diverse mesh surfaces on the rounded boundary of the in-

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

Table 1: *Execution times (ms) of CGAL, Polyhedron Kernel and KerGen algorithms for 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	# Mesh faces	# Kernel edges	CGAL	Poly. Ker.	KerGen
Crossing Cubes	29994	72	836	668	**294**
Cube on Cylinder	29986	108	802	1128	**503**
Cylinder Trio	23062	156	985	1116	**94**
Wringer	17514	42	405	141	**52**
Liver	112	106	10	**2**	5
Rock	916	24	32	10	**3**
Skull	2110	42	47	15	**7**
Acorn	8224	1601	**370**	337743	656
Ball	1316	714	**55**	766	62
Bot Eye	902	177	37	444	**21**
Button	2450	258	48	269	**42**
Cross	7824	153	1974	10814	**87**
Diamond	14080	869	2889	11289	**255**
Flex	1664	80	260	28	**7**
Muffin	17940	1464	**784**	151049	952
Part	10760	1038	6329	14196	**1369**
Plus	892	18	86	6	**3**
Rt-4 Arm	1306	526	72	239	**54**
Star	19262	165	4112	**893**	1021
Super Ellipse	576	168	16	28	**8**
Princeton (AVG)	12		790	1159	**164**
Thingi (AVG)	320		214	5132	**79**

put meshes themselves. In such cases, CGAL may outperform both KerGen and Polyhedron Kernel due to its computational efficiency, as it is less dependent on the shape of the input mesh or its kernel compared to geometric methods. Conversely, Polyhedron Kernel shows higher performance for *Liver* and *Star* meshes, thanks to its efficient handling of small-sized meshes and meshes with many co-planar bounding faces, respectively. Aside from these specific cases, KerGen demonstrates higher efficiency for the remaining sample star-shapes, even those with a high number of mesh faces, due to its unique additive progress that avoids unnecessary computations.

Table 1 also reveals that, for half of the inputs, Polyhedron Kernel outperforms CGAL, while the reverse is true for the other half. The average timings for both datasets indicate that KerGen computes kernels approximately 5 times faster than CGAL and 7 times faster than Polyhedron Kernel for the *Princeton* dataset, and nearly 3 and 65 times faster, respectively, for the *Thingi* dataset.

Additionally, Figure 7 illustrates the timings of all algorithms across the entire *Thingi* and *Princeton* datasets, clearly showcasing KerGen's superior computational efficiency for input meshes of all sizes.

In terms of accuracy, we have not observed a notable difference between the computed kernels and the results obtained by exact computations. All three algorithms consistently achieved accurate kernels, with volumetric discrepancies averaging around 1e-5% and similar Hausdorff distance measurements compared to exact outputs. Further details on numerical precision are provided in Section 6.2.

In conclusion, the performance of kernel computation algorithms is significantly affected by the geometry of the input mesh and output kernel, as well as the number of bounding planes. Polyhedron Kernel exhibits an advantage for small-sized meshes and large-sized meshes with many co-planar faces, while CGAL is preferable for meshes whose kernel features a rounded shape. However, KerGen generally outperforms both algorithms, particularly for input meshes with small-sized kernels as a result of its efficient output-sensitive structure.

5.2. Comparison Over Non-Star-Shaped Meshes

We also provide performance comparisons using shapes with empty kernels, as identifying non-star-shaped input is crucial for efficient star decomposition processes. Thanks to our efficient approach, our additive algorithm, initiated with a kernel point, immediately determines non-star-shaped input at the first stage. We utilize Berg et al.'s linear programming solution [DVOS97] for this stage, which runs instantly. In contrast, competitor algorithms like CGAL and Polyhedron Kernel, which rely on algebraic transformations and carving-based subtractions, respectively, reach this determination at a later stage. To provide further insights, we present the average timings obtained across the *Thingi* and *Princeton* datasets in Table 2. In the *Thingi* dataset, consisting of 1482 non-star shapes, the average execution time of the initial point finding algorithm is $1e-5$ milliseconds, while for the 368 non-star shapes in the *Princeton* dataset, it is $4e-4$ milliseconds.

In contrast, both CGAL and Polyhedron Kernel algorithms employ their standard intersection computation methods until encountering an empty set for non-star shapes. As indicated in Sorgente et

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

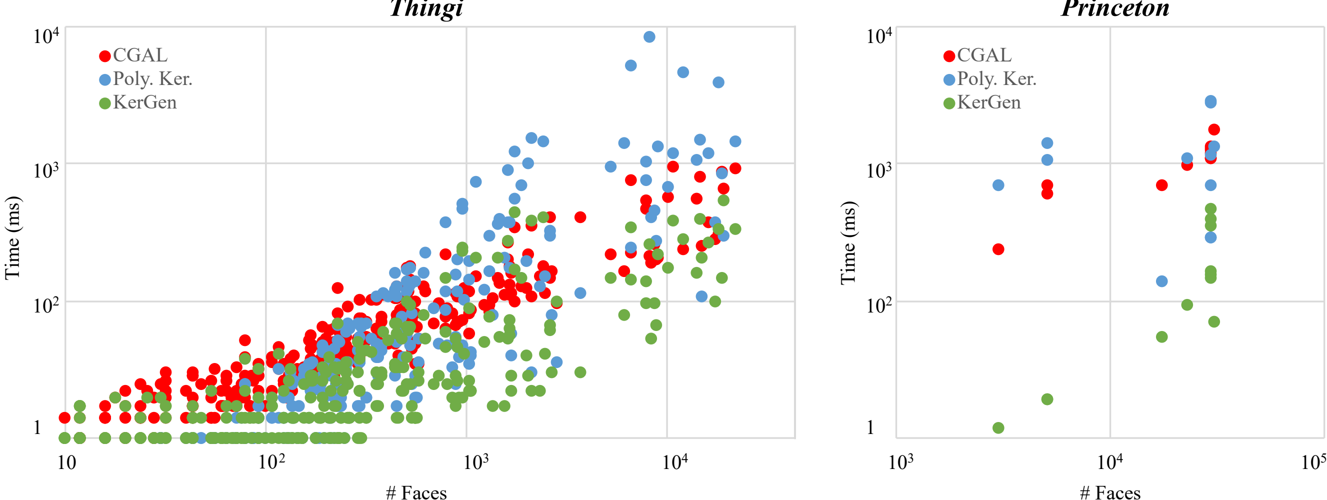


Figure 7: *The statistics for the star-shaped meshes from the Thingi and Princeton datasets. The results clearly demonstrate KerGen's state-of-the-art performance.*

al.'s study [SBS22a], the Polyhedron Kernel algorithm outperforms CGAL for all non-star models. According to our experiments, Polyhedron Kernel detects empty kernels in an average time of 402 milliseconds and 626 milliseconds for the *Thingi* and *Princeton* datasets, respectively. In comparison, CGAL's detecting time averages 540 milliseconds for the *Thingi* dataset and 1071 milliseconds for the *Princeton* dataset.

Table 2: *The average execution times (ms) for non-star shapes.*

Dataset	# Shapes	CGAL	Poly. Ker.	KerGen
Thingi	1482	540	402	**1e − 5**
Princeton	368	1071	626	**4e − 4**

5.3. Performance Analysis Over Remeshed Star-Shaped Meshes

We finally investigate the sensitivity of KerGen to variations in polygons and resolutions using remeshed models. We selected the *Spiral* (ThingiID: 60246) and *Vase* (ThingiID: 85580) models from the *Thingi* dataset, as they are commonly used for similar analyses [SBS22a]. Each model has six remeshed versions, with each subsequent version containing four times more faces than the previous one.

To remesh the *Spiral* model, we divided each mesh face into four sub-pieces by linking the center of the face to its corners. This approach maintains the same number of bounding planes despite the increase in the total face count. Conversely, we utilized the Loop's subdivision algorithm to remesh the *Vase* model, potentially generating faces lying on entirely different planes.

Figure 8 depicts the kernels computed by KerGen for each remeshed shape alongside the execution times of all three algorithms. While the output kernel remains unchanged for the *Spiral* model across its remeshed versions due to the consistent use of bounding planes, a slight variation in the output kernel boundary is observed for the *Vase* model. However, this difference is minimal and difficult to perceive visually. Notably, the number of output kernel edges demonstrates nearly linear growth across the different remeshed versions. Our implementation of KerGen accurately produces the kernels of the refined versions of both models without any degeneracy.

The execution time for kernel computation gradually increases for both models as the number of faces grows, which is consistent with our complexity analysis in Section 4.3. The charts clearly demonstrate the computational superiority of KerGen over both CGAL and Polyhedron Kernel. In the case of the *Spiral* model, KerGen proves to be 2 to 30 times faster than CGAL across varying resolutions of remeshing, with a relatively consistent ratio compared to Polyhedron Kernel, which ranges from 3 to 5. Conversely, for the *Vase* model, KerGen outperforms CGAL by a factor of 5 to 20, with more pronounced ratios compared to Polyhedron Kernel, ranging from 4 to 80 as the resolution of the remeshing increases.

6. Discussion

6.1. Evaluation

Both the Polyhedron Kernel and KerGen algorithms approach the problem from a geometric perspective, utilizing plane-plane and line-plane intersections, and point classifications in relation to planes. However, they differ in their fundamental strategies for extracting the kernel: Polyhedron Kernel adopts a subtractive approach, initiating from the axis-aligned bounding box of the shape and progressively eliminating its non-kernel portions by sequentially filtering each half-space requirement. Conversely, KerGen employs an additive method, beginning with an initial kernel point and iteratively determining the kernel boundaries by following adjacency relationships. Consequently, the resulting object consistently resides within the kernel throughout the process, whereas our

© 2024 Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd.

competitor may include non-kernel components until completion. This brings in KerGen a remarkable advantage over Polyhedron Kernel, particularly in utilizing partial kernel data from specific regions to address other geometry processing challenges, as elaborated in Section 7.

On the other hand, CGAL adopts an algebraic approach involving operations and constructions interpreted in 4D. It converts the half-spaces defined by homogeneous coordinates into their corresponding dual points. Subsequently, it computes the convex hulls of the point sets, performing linear transformations in the intermediate steps, and projects the outcome back from dual space to the original space. While CGAL's computational complexity ($\mathcal{O}(N_f log N_f)$) is lower than that of geometric approaches (which tend to be quadratic), its solution is insensitive to variations in input and output mesh geometries. Consequently, it does not allow for the possibility of truncating its execution, which would otherwise reduce its operational cost. Empirical studies suggest that geometric methodologies may outperform the algebraic approach in various practical scenarios. In essence, the efficiency of operations in KerGen, particularly concerning only the kernel boundary while excluding non-kernel parts, proves more performance-effective. Intuitively, when the number of edges on the kernel boundary are expected to be minimal, KerGen demonstrates remarkable computational efficiency.

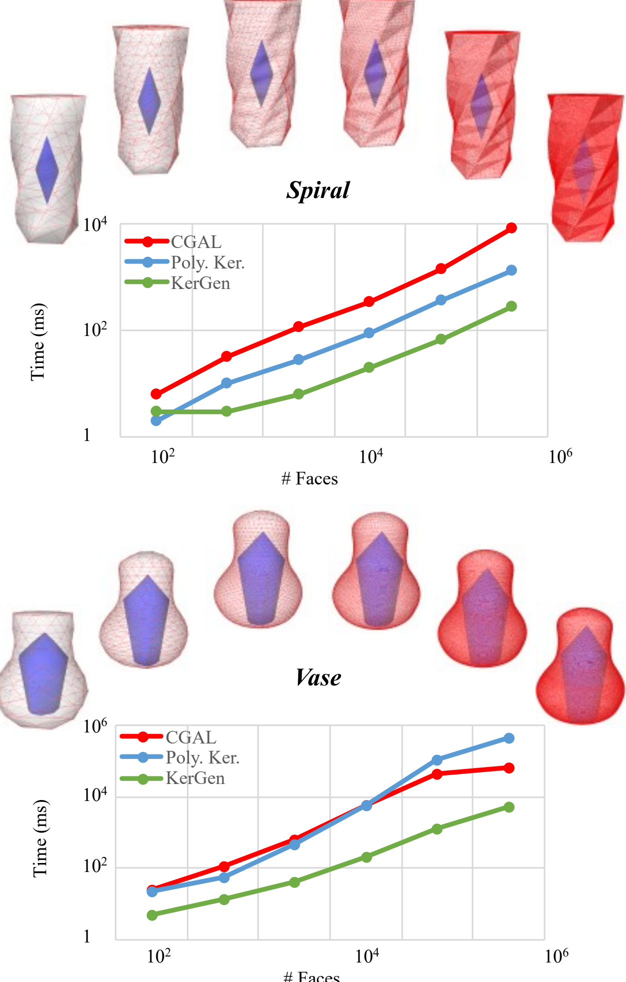


Figure 8: *The kernels computed by our KerGen algorithm on six different remeshed versions of the Spiral and Vase models. Additionally, timings are plotted, revealing linear growth.*

6.2. Numerical Precision

The precision adjustment of outputs across all three algorithms can be facilitated through various computational tools. To ensure fair comparisons, we have evaluated the timing performance of each algorithm under identical settings, as detailed in Table 1, utilizing implementations employing inexact predicates with explicit inexact constructions. While this methodology enables us to achieve efficient results in terms of both execution times and output accuracies, we also present a concise analysis of outcomes obtained through alternative configurations.

In our quantitative accuracy assessment, we conducted a benchmark comparison against results derived from CGAL's `Exact_predicates_exact_constructions` option which exhibited an average slowdown of approximately 2.4 times. This involved evaluating volumetric discrepancies and Hausdorff distances between the computed kernels and their exact forms for each input mesh in the *Thingi* and *Princeton* datasets. Across these datasets, all three algorithms consistently achieved accurate kernels, with volumetric discrepancies averaging around 1e-5% and similar Hausdorff distance measurements. Our analysis included meshes featuring numerous co-planar and nearly co-planar faces such as *Crossing Cubes*, *Cube on Cylinder*, *Wringer*, *Acorn*, *Diamond*, *Muffin*, and *Star*, revealing outputs characterized by visually imperceptible deviations. In many instances, metric values approached zero, indicating a high level of accuracy.

Furthermore, CGAL offers a half-space intersection functionality without explicit constructions, resulting in greater reliability compared to inexact constructions. Additionally, it achieves shorter computation durations than exact computations. Our investigation revealed that this functionality resulted in timings nearly 1.8 times longer on average compared to those detailed in Table 1, with minimal error rates. A similar strategy could be explored for geometric approaches through the utilization of indirect predicates introduced by Attene [Att20], promising robustness with a modest increase in execution times.

It is important to note that while KerGen using inexact configurations produced high-accuracy kernels, it may not always generate a convex watertight 2-manifold output mesh converging to the kernel itself. Inexact predicate checks and plane intersections can lead to incorrect point classifications, plane labeling, and false line generations and tracings, resulting in gaps, non-manifold edges, or divergences. The first issue could be mitigated by computing the convex hull of the identified kernel points to extract the final shape of the output kernel, while the second could be limited by restricting the computation within the input mesh boundaries. However, incorrect kernels may still occur in both applications. Similar erroneous kernels may also be observed in other existing methods due to inexactness. A straightforward solution to prevent these issues

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

is using exact computations, albeit with an appreciable increase in execution time. Alternatively, indirect predicates can be used by replacing current data structures with indirect ones that define kernel vertices and lines over the generator planes.

In conclusion, our findings highlight the notable quality of our approach, characterized by high accuracies achieved within short timeframes, outperforming both CGAL and Polyhedron Kernel. However, we also acknowledge that its implementation using inexact configurations with explicit constructions may compromise robustness, although this concern could potentially be mitigated by employing indirect predicates with a reasonable trade-off in computational speed. We argue that the potential performance variations do not justify the significant gap in other application advantages of KerGen, such as its additive process, which allows for partial kernels to be generated at any time, and its dynamic feature extraction capabilities on the kernel's geometry.

6.3. Limitations

KerGen assumes that each edge in the input mesh is shared by exactly two polygons and that the shape is oriented, thereby excluding non-manifold shapes. It relies on well-defined inside/outside concepts, which are satisfied by unique regions or volumes of the mesh. Consequently, KerGen may encounter failure with inputs lacking clear inside/outside definitions, such as non-manifold meshes. However, non-star meshes with non-zero genus or disconnected components pose no issue for KerGen, as it does not proceed after determining the non-existence of a single kernel point in such cases. For manifolds with boundaries, KerGen can still be applied by utilizing bounding planes aligned with the boundary edges, perpendicular to the shape along the edge, instead of employing algebraic definitions of those edges.

7. Conclusions and Future Works

In this paper, we propose a novel and fast solution named KerGen for computing kernels of 3D shapes represented as polygon meshes. By relying solely on plane-plane and line-plane intersections, as well as point identifications based on their positions with respect to planes, we have successfully computed the kernel of a shape. While the computation of kernels has been extensively studied in 2D, it is still in its infancy when it comes to the 3D domain. There are only two existing methods that can find 3D kernels, and we have shown clear advantages over both of them. Specifically, we provide an additive formulation enabling significantly faster execution times while producing very accurate kernels.

We believe that KerGen has significant potential in various computer graphics applications such as remeshing, star decomposition, shape guarding, mesh quality measurement, and both spherical and planar parametrization, which may benefit from an explicit kernel representation (Section 1). In the remainder of this section, we further expand upon the application context of geometric kernels, leveraging the properties of our KerGen algorithm as a future tool.

Parallelization. Our current implementation relies on a serial mesh data structure, making it single-threaded. However, due to the capability of our KerGen algorithm to initiate the process with a single kernel point in any direction, parallelization could be easily achieved by initiating multiple processes with extreme points in different directions safely. To enhance efficiency and prevent repeated edge traversals by different threads, a data structure like a kd-tree could be utilized to partition the volume enclosed by the input mesh. This would allow for the shared processing of explored kernel vertices among threads based on their locations, thereby managing the traversal of adjacent kernel edges more effectively. Idle threads responsible for non-kernel locations could be seamlessly reassigned to other potential kernel locations.

Robustness. As mentioned in Section 6.2, KerGen's current configuration relies on inexact computations, which may pose concerns regarding robustness. As a potential solution, similar to the applications presented in [CLSA20] and [CPAL22], indirect predicates [Att20] could be employed for operations dependent on line generating intersections. Although those predicates support one level of indirection, the kernel line and vertex constructions employed in our method KerGen can be defined solely based on the generator planes, whose identifications are not affected by cascading computations. We believe that with this approach, KerGen offers high-performance results while preserving robustness.

Star-shape Decomposition. 3D art gallery problem could be addressed by decomposing the object into star-pieces [AT81; YL11; YLL13]. Unlike previous subtractive methods, our additive approach in KerGen expands the kernel elements one by one, enabling us to initiate kernel generation for a connected subset of the shape and halt the generation at any stage that overlaps with the kernel of a neighboring subset. Furthermore, akin to the concept presented in [KYD*18], this approach facilitates the remeshing of the shape, approximating it to a form that is closer to being star-shaped. As a result, it yields a more suitable model for spherical parametrization, which is advantageous for achieving a smoother, interpolating model, as elaborated in [SGU17].

Robot Motion Planning. For robot motion planning problems, specific connected regions which are desired to visibly cover a certain domain can be defined as kernels, allowing the current visibility to be re-defined while accommodating the predefined kernel [GS20; MB23]. Thanks to the incremental structure of KerGen, kernel-aware shapes can be dynamically generated by tracing the boundaries of the region intended to be the kernel. This approach enhances flexibility in designing area of motion for robots based on their standing points.

Acknowledgments

This work was supported by TUBITAK under the project EEEAG-119E572.

References

[AS24] ASILER, MERVE and SAHILLIOĞLU, YUSUF. "3D geometric kernel computation in polygon mesh structures". *Computers & Graphics* (2024), 103951 2.

[AT81] AVIS, DAVID and TOUSSAINT, GODFRIED T. "An efficient algorithm for decomposing a polygon into star-shaped polygons". *Pattern Recognition* 13.6 (1981), 395–398 2, 12.

[Att20] ATTENE, MARCO. "Indirect predicates for geometric constructions". *Computer-Aided Design* 126 (2020), 102856 11, 12.

[BBC*13] BEIRÃO DA VEIGA, L, BREZZI, FRANCO, CANGIANI, ANDREA, et al. "Basic principles of virtual element methods". *Mathematical Models and Methods in Applied Sciences* 23.01 (2013), 199–214 2, 3.

© 2024 Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd.

[CGF09] CHEN, XIAOBAI, GOLOVINSKIY, ALEKSEY, and FUNKHOUSER, THOMAS. "A Benchmark for 3D Mesh Segmentation". *ACM Transactions on Graphics* 28.3 (Aug. 2009) 7, 8.

[CHJ08] CHUN, SUNGKUK, HONG, KWANGJIN, and JUNG, KEECHUL. "3D star skeleton for fast human posture representation". *World Acad. Sci. Eng. Technol* 2 (2008), 2603–2612 2.

[CLSA20] CHERCHI, GIANMARCO, LIVESU, MARCO, SCATENI, RICCARDO, and ATTENE, MARCO. "Fast and robust mesh arrangements using floating-point arithmetic". *ACM Transactions on Graphics (TOG)* 39.6 (2020), 1–16 12.

[CPAL22] CHERCHI, GIANMARCO, PELLACINI, FABIO, ATTENE, MARCO, and LIVESU, MARCO. "Interactive and robust mesh booleans". *arXiv preprint arXiv:2205.14151* (2022) 12.

[DVOS97] DE BERG, MARK, VAN KREVELD, MARC, OVERMARS, MARK, and SCHWARZKOPF, OTFRIED. *Computational geometry*. 1997 2, 3, 5, 9.

[ER97] ETZION, MICHAL and RAPPOPORT, ARI. "On compatible star decompositions of simple polygons". *IEEE Transactions on Visualization and Computer Graphics* 3.1 (1997), 87–95 3.

[FKR05] FLOATER, MICHAEL S, KÓS, GÉZA, and REIMERS, MARTIN. "Mean value coordinates in 3D". *Computer Aided Geometric Design* 22.7 (2005), 623–631 2.

[FM84] FOURNIER, ALAIN and MONTUNO, DELFIN Y. "Triangulating simple polygons and equivalent problems". *ACM Transactions on Graphics (TOG)* 3.2 (1984), 153–174 2.

[FP09] FABRI, ANDREAS and PION, SYLVAIN. "CGAL: The computational geometry algorithms library". *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*. 2009, 538–539 2, 8.

[GK92] GARDNER, RJ and KALLAY, MICHAEL. "Subdivision Algorithms and the Kernel of a Polyhedron". *Discrete & Computational Geometry* 8.4 (1992), 417–427 2.

[GS20] GEWALI, LAXMI and SUBEDI, BIBEK. *Random Generation of Visibility Aware Polygons*. Springer, 2020 2, 12.

[JP17] JACOBSON, ALEC and PANOZZO, DANIELE. "Libigl: Prototyping geometry processing research in c++". *SIGGRAPH Asia 2017 courses*. ACM New York, NY, USA, 2017, 1–172 2.

[KCP92] KENT, JAMES R, CARLSON, WAYNE E, and PARENT, RICHARD E. "Shape transformation for polyhedral objects". *ACM SIGGRAPH Computer Graphics* 26.2 (1992), 47–54 2, 3.

[Kei85] KEIL, J MARK. "Decomposing a polygon into simpler components". *SIAM Journal on Computing* 14.4 (1985), 799–817 3.

[KYD*18] KHAN, DAWAR, YAN, DONG-MING, DING, FAN, et al. "Surface remeshing with robust user-guided segmentation". *Computational Visual Media* 4 (2018), 113–122 12.

[Liv19] LIVESU, MARCO. "cinolib: a generic programming header only C++ library for processing polygonal and polyhedral meshes". *Transactions on Computational Science XXXIV* (2019), 64–76 3, 8.

[Liv24] LIVESU, MARCO. "Advancing Front Surface Mapping". *Computer Graphics Forum* (2024) 2, 3.

[LL86] LEE, D and LIN, ARTHURK. "Computational complexity of art gallery problems". *IEEE Trans. Info. Theory* 32.2 (1986), 276–282 3.

[LP79] LEE, DER-TSAI and PREPARATA, FRANCO P. "An optimal algorithm for finding the kernel of a polygon". *Journal of the ACM (JACM)* 26.3 (1979), 415–421 2.

[MB23] MANDAL, SHASHWATA and BHATTACHARYA, SOURABH. "Relay Pursuit for Multirobot Target Tracking on Tile Graphs". *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2023, 7691–7698 2, 3, 12.

[OSN*20] OOI, ET, SAPUTRA, A, NATARAJAN, S, et al. "A dual scaled boundary finite element formulation over arbitrary faceted star convex polyhedra". *Computational Mechanics* 66.1 (2020), 27–47 2.

[PCS*22] PIETRONI, NICO, CAMPEN, MARCEL, SHEFFER, ALLA, et al. "Hex-mesh generation and processing: a survey". *ACM transactions on graphics* 42.2 (2022), 1–44 2, 3.

[PM79] PREPARATA, FRANCO P. and MULLER, DAVID E. "Finding the intersection of n half-spaces in time O (n log n)". *Theoretical Computer Science* 8.1 (1979), 45–55 2, 3, 8.

[Sac19] SACRISTÁN, VERA. *Intersecting Half-Planes and Related Problems*. 2019 2.

[SBMS22] SORGENTE, TOMMASO, BIASOTTI, SILVIA, MANZINI, GIANMARCO, and SPAGNUOLO, MICHELA. "The role of mesh quality and mesh quality indicators in the virtual element method". *Advances in Computational Mathematics* 48.1 (2022), 3 2, 3.

[SBMS23] SORGENTE, TOMMASO, BIASOTTI, SILVIA, MANZINI, GIANMARCO, and SPAGNUOLO, MICHELA. *A Survey of Indicators for Mesh Quality Assessment*. Wiley Online Library, 2023 2, 3.

[SBS22a] SORGENTE, TOMMASO, BIASOTTI, SILVIA, and SPAGNUOLO, MICHELA. "Polyhedron kernel computation using a geometric approach". *Computers & Graphics* 105 (2022), 94–104 2, 3, 8, 10.

[SBS22b] SORGENTE, TOMMASO, BIASOTTI, SILVIA, and SPAGNUOLO, MICHELA. *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*. [Accessed 29-Aug-2022]. 2022 7, 8.

[SDG*19] SCHNEIDER, TESEO, DUMAS, JÉRÉMIE, GAO, XIFENG, et al. "Poly-spline finite-element method". *ACM Transactions on Graphics (TOG)* 38.3 (2019), 1–16 2, 3.

[Sei91] SEIDEL, RAIMUND. "Small-dimensional linear programming and convex hulls made easy". *Discrete & Comp. Geom.* 6 (1991), 423–434 2.

[SGU17] SCHMITTER, DANIEL, GARCÍA-AMORENA, PABLO, and UNSER, MICHAEL. "Smooth shapes with spherical topology: Beyond traditional modeling, efficient deformation, and interaction". *Computational Visual Media* 3 (2017), 199–215 12.

[SH76] SHAMOS, MICHAEL IAN and HOEY, DAN. *Geometric intersection problems*. IEEE, 1976 2.

[She97] SHEWCHUK, JONATHAN R. "Adaptive precision floating-point arithmetic and fast robust geometric predicates". *Discrete & Computational Geometry* 18 (1997), 305–363 2, 3.

[SJG19] STEIN, ODED, JACOBSON, ALEC, and GRINSPUN, EITAN. "Interactive design of castable shapes using two-piece rigid molds". *Computers & Graphics* 80 (2019), 51–62 3.

[SPO10] SCHVARTZMAN, SARA C, PÉREZ, ÁLVARO G, and OTADUY, MIGUEL A. "Star-contours for efficient hierarchical self-collision detection". *ACM SIGGRAPH 2010 papers*. 2010, 1–8 2, 3.

[SR95] SHAPIRA, MICHAL and RAPPOPORT, ARI. "Shape blending using the star-skeleton representation". *IEEE Computer Graphics and Applications* 15.2 (1995), 44–50 2, 3.

[Sub19] SUBEDI, BIBEK. "Generating kernel aware polygons". PhD thesis. University of Nevada, Las Vegas, 2019 2.

[SVB*23] SORGENTE, TOMMASO, VICINI, FABIO, BERRONE, STEFANO, et al. "Mesh quality agglomeration algorithm for the virtual element method applied to discrete fracture networks". *Calcolo* 60.2 (2023), 27 2, 3.

[WLH*13] WONG, SAI-KEUNG, LIN, WEN-CHIEH, HUNG, CHUN-HUNG, et al. "Radial view based culling for continuous self-collision detection of skeletal models". *ACM Transactions on Graphics (TOG)* 32.4 (2013), 1–10 2, 3.

[YL11] YU, WUYI and LI, XIN. "Computing 3d shape guarding and star decomposition". *Computer Graph. Forum* 30.7 (2011), 2087–2096 2, 3, 12.

[YLL13] YU, WUYI, LI, MAOQING, and LI, XIN. "Optimizing pyramid visibility coverage for autonomous robots in 3D environment". *Int. Conf. on Computer Science & Education*. IEEE. 2013, 1023–1028 2, 3, 12.

[ZJ16] ZHOU, QINGNAN and JACOBSON, ALEC. "Thingi10k: A dataset of 10,000 3d-printing models". *arXiv preprint arXiv:1605.04797* (2016) 8.

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.