A review of polyhedral intersection detection and new techniques

Samuel Hornus

▶ **To cite this version:**

Samuel Hornus. A review of polyhedral intersection detection and new techniques. [Research Report] RR-8730, Inria Nancy - Grand Est (Villers-lès-Nancy, France). 2015, pp.35. hal-01157239v1

HAL Id: hal-01157239

https://inria.hal.science/hal-01157239v1

Submitted on 3 Jun 2015 (v1), last revised 12 Jun 2015 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

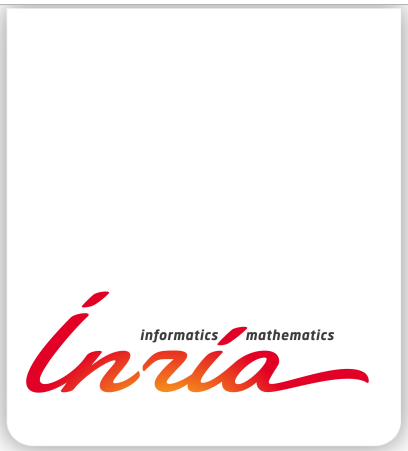
Distributed under a Creative Commons Attribution - ShareAlike 4.0 International License

A review of polyhedral intersection detection and new techniques

Samuel Hornus

RESEARCH REPORT
N° 8730
Mai 2015
Project-Teams Alice

ISSN 0249-6399 ISRN INRIA/RR--8730--FR+ENG



A review of polyhedral intersection detection and new techniques

Samuel Hornus

Project-Teams Alice

Research Report n° 8730 — Mai 2015 — 35 pages

Abstract: This paper delves into the problem of detecting the intersection of two convex polyhedra. It does so through the lens of Minkowski sums and Gauss maps, and with a bias towards applications in computer graphics and robotics. In the first part, we show how Minkowski sums and Gauss maps come into play, give a brief survey of techniques for pairs of simple shapes and describe a low-level optimization of a naive algorithm for convex polyhedra, which is applied to tetrahedra. Novel applications to the ray casting problem are also given. In the second part, we take a more abstract approach to the problem and describe a new and very efficient and robust algorithm for detecting the intersection of two convex shapes. The new technique works directly on the unit sphere, interpreted as the sphere of directions. In particular, it is compared favourably to the ubiquitous algorithm of Gilbert, Johnson and Keerthi.

Key-words: computer graphics, computational geometry, intersection detection, robotics, collision, GJK

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Détection de l'intersection de deux convexes : revue et nouvelles techniques

Résumé : Cet article discute du problème (décisionnel) de la détection de l'intersection de deux polyèdres convexes. Il porte particulièrement sur les applications de ce problème en informatique graphique et en robotique. La discussion s'y fait du point de vue des sommes de Minkoswki et de l'application de Gauss. Dans la première partie, nous rappellons le rôle de ce point de vue dans la compréhension de la géométrie du problème. Nous donnons un bref aperçu des techniques conçues pour certaines paires de formes simples, et nous proposons un algorithme naïf mais optimisé, traitant des polyèdres convexes quelconques. Nous traitons en exemple une application aux paires de tétraèdres et une application au problème du lancer de rayons. En deuxième partie, nous approchons le problème de manière plus abstraite et décrivons un nouvel algorithme robuste et rapide pour la détection de l'intersection de deux objets convexes (non nécessairement polyédrique). Ce nouvel algorithme travaille directement sur la sphère unité que nous interprétons comme l'espace des directions. En particulier, notre nouvelle technique est comparée favorablement à celle, fort répandue, de Gilbert, Johnson et Keerthi.

Mots-clés : informatique graphique, géométrie algorithmique, détection d'intersection, robotique, collision, GJK

1 Introduction

This paper is concerned with the problem of detecting the intersection of two convex objects. Given two convex objects A and B in $\mathbb{R}^3$, we ask whether A and B have a point in common or not. When that is the case we say that "they intersect" or "they touch each other." The intersection detection problem is a component of collision detection for more general shapes, which plays a major role in robotics [8], computer animation [11] and mechanical simulation for example.

Intersection detection also plays a role in computer graphics in general as an ingredient in acceleration data structures, such as bounding volume hierarchies or Kd-trees. In the latter case, an object of interest (eg a ray, a view frustum, or another hierarchy) is tested for intersection against the geometric shapes that bound each node in the hierarchy. These bounding shapes are simple shapes (boxes, spheres) for which fast intersection detection techniques exist. For an in-depth exposition to intersection and collision detection, we refer the reader to the book of Ericson [7] and the survey of Jiménez *et al.* [15]. In robotics or computer graphics, we often limit ourselves to constant-size or small convex objects and techniques that do not use pre-processing, but the intersection detection problem has several variants and have also been studied by theoreticians as well.

Computational geometers have recently developed an optimal solution for general convex polyhedra: Given any collection of convex polyhedra in $\mathbb{R}^3$, one can pre-process them in linear time, independently of each other, so that the intersection of any two polyhedra P and Q from the collection can be tested in optimal time $O(\log|P| + \log|Q|)$ (see [2] and the other references within). This essentially closes the problem for the case of convex polyhedra.

In this paper however, we only consider techniques that do not use pre-processing [1] and are asymptotically slower, but very fast in practice. In addition to general polyhedra, we also consider techniques that are tailored to specific convex shapes (tetrahedra, spheres, axis-aligned boxes, oriented boxes, frusta, unbounded pyramids, zonohedra).

A common theme, throughout the paper, is the well known and beautiful interplay between the intersection detection problem and Minkowski sums and the overlay of Gauss maps, which we use extensively.

This paper is divided in two parts. The first part (Sections §2 to §5) focuses on the problem of detecting the intersection of two convex polyhedra. It shows how Minkowski sums and Gauss maps come into play, gives a brief survey of techniques for pairs of simple shapes and describes a low-level optimization of a naive algorithm for convex polyhedra, which is applied to tetrahedra. Novel applications to the ray casting problem are also given. The second part (Sections §6 to §8) takes a more abstract approach to the problem. It leverages our understanding of Gauss maps developed in the first part in order to describe a new and very efficient and robust algorithm for detecting the intersection of two convex shapes. The new technique works directly on the unit sphere, interpreted as the sphere of directions.

Detailed content of the paper In §2 we describe in detail the equivalence of detecting the intersection of two convex shapes A and B and checking that the origin belongs to their Minkowski difference $A \ominus B$. For polyhedra, the Minkowski difference is a polyhedron as well, and we show how to compute information about this difference via the overlay of Gauss maps. This let us describe an abstract algorithm (equation 10) for intersection detection. The rest of this section reviews a number of well known techniques for simple polyhedra and relates them to the abstract algorithm.

[1] Except, briefly, in §8.4.

In §3 we make the abstract algorithm concrete and describe low-level technique and optimization for this generic algorithm. Although the resulting algorithm is quadratic in nature, we show in §4 that it is quite efficient in practice for small polytopes, namely tetrahedra. (The reader will find more extensive benchmarks in §8.)

In §5, we specialize equation 10 to intersection detection problems that appear in packet ray casting: object A is a (non truncated) view-pyramid that encloses the rays going through a rectangular piece of the image plane. Object B is sometimes an axis-aligned box bounding some geometry in a hierarchy, sometimes a triangle from the geometrical description of the scene. We show how our new techniques improve the performance of the packet ray tracing technique of Wald *et al.* [18].

In this first part, we consider sections 3, 4 and 5 as novel contributions.

We view the second part, that describes and analyzes the new SphereSearch algorithm, as the more important contribution of this paper. We define a signed distance function $D : \mathcal{S} \mapsto \mathbb{R}$ such that the preimage of the negative numbers under D, $\mathcal{S}^-(A \ominus B) = D^{-1}(\mathbb{R}^-)$ is precisely the set of directions along which A and B can be separated (by an orthogonal plane). The intersection detection problem reduces to the problem of finding a point $n \in \mathcal{S}$ whose image by D is negative or deciding that D is everywhere non-negative. In §6, we design such an algorithm, which we call SphereSearch. It iteratively prunes parts of the unit sphere $\mathcal{S}$ so that the remaining part, a convex spherical polygon, converges towards $\mathcal{S}^-(A \ominus B)$.

Section 7 gives a theoretical analysis of SphereSearch and an extensive comparison of SphereSearch with GJK [9]. In particular, it shows that SphereSearch optimally aggregates the information gathered about $A \ominus B$ during the successive iterations.

In §8, we benchmark our implementations of the "naive", quadratic algorithm from §3, of SphereSearch and of GJK. Each benchmark considers a specific type of objects and measures the performance of the algorithm with respect to the "collision density", the ratio of the number of tested pairs of convex objects that actually intersect to the total number of tested pairs. We have taken care to analyse a large variety of situations including very uneven ones, such as frustum-culling where one object (the frustum) is much larger that the other. In that case, we show that a hybrid technique combining SphereSearch and GJK gives the overall best results.

Sections 6 to 8 are largely independent from sections 3 to 5.

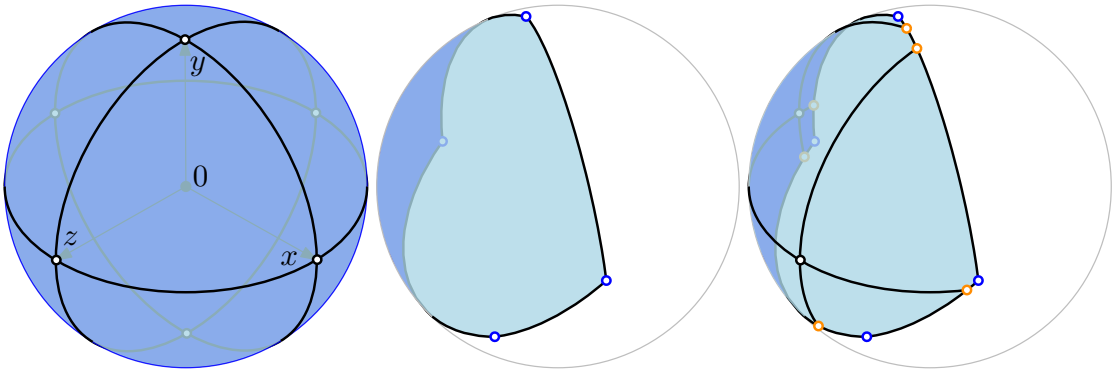


Figure 1: *Left.* The Gauss map of an axis-aligned box has 6 dual-points, 12 dual-arcs and 8 dual-faces; it fully covers the unit sphere. *Middle.* The Gauss map of a view pyramid is shaded blue. It has 4 dual-points, 4 dual-arcs and 1 dual-face. Since the view pyramid is unbounded, its Gauss map does *not* fully cover the unit sphere. *Right.* The overlay of both Gauss maps. There are 4 normal vectors (or dual-points) in N_A (blue), 2 in N_{-B} (black) and 6 in N_{AB} (orange).

2 The point-in-polyhedron point of view

2.1 Reduction to a point-in-polyhedron test

The Minkowski sum of two subsets A and B of $\mathbb{R}^3$ is

$$A \oplus B = \{a + b \mid a \in A, b \in B\}. \tag{1}$$

Writing $-B = \{-b \mid b \in B\}$, their Minkowski difference is

$$A \ominus B = A \oplus (-B) = \{a - b \mid a \in A, b \in B\}. \tag{2}$$

From equation (2), we deduce that A and B have non-empty intersection if and only if $A \ominus B$ contains the origin:

$$A \cap B \neq \emptyset \quad \Leftrightarrow \quad 0 \in A \ominus B. \tag{3}$$

From now on, we consider only convex polyhedra. When A and B are (convex) polyhedra, their Minkowski sum or difference is a convex polyhedron as well. A polyhedron is the intersection of closed half-spaces bounded by a set of planes. We always orient these planes so that the polyhedron that they bound is the intersection of theirs negative sides. While a polyhedron admits infinitely many bounding planes, we only consider the necessary ones: the supporting planes of the facets of the polyhedron. The intersection test (3) amounts to checking that the origin lies on the negative side of each plane bounding $A \ominus B$.

In the next section, we show how one might obtain the planes supporting the facets of $A \ominus B$ from the vertices of the overlay of two Gauss maps. We may already notice that, by omitting some bounding planes from consideration, one obtains a *conservative* intersection test, with possible false positives but no false negative: A and B may be found to intersect each other while they in fact do not.

2.2 The overlay of two Gauss maps

The Gauss map $\mathcal{G}(P)$ of a polyhedron P gives us a duality between the arrangement of the vertices, edges and facets of P and an arrangement on the unit sphere $\mathcal{S} = \{x \in \mathbb{R}^3 \mid |x| = 1\}$. It maps planes tangent to P (with P on their negative side) to their unit normal vector. The dual of a face (vertex, edge or facet) of P is the image of the planes tangent to P at points in that face:

- A facet f of P has an isolated dual-point on $\mathcal{S}$, which is the unit normal vector of the facet f.
- An edge e of P has a dual-arc $e^\star$ on $\mathcal{S}$ between two dual-points. This dual-arc lies on the great-circle of the unit sphere whose corresponding "north pole" is the direction of edge e. The two end-dual-points of $e^\star$ are the dual-points of the two facets of P adjacent to edge e.
- Finally, a vertex v of P has a convex spherical dual-polygon on $\mathcal{S}$ bounded by dual-arcs. The latter are the dual-arcs of the edges of P adjacent to v.

The Gauss map is a kind of dual of a polyhedron in the same way a Voronoï diagram is the dual of a Delaunay triangulation. Figure 1 (left) illustrates the Gauss map of an axis-aligned box. When a polyhedron is unbounded in some direction v, it has no tangent plane with outward normal v. Thus, its Gauss map does not cover point v on the unit sphere. In general,

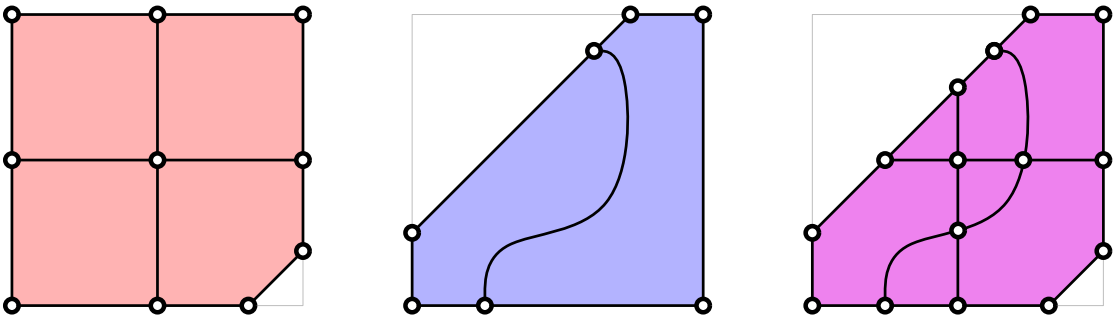


Figure 2: A visual definition of an overlay: *Left & Middle.* Two maps in a square. *Right.* Their overlay. The red map has 10 vertices, 13 edges and 4 faces. The blue map has 7 vertices, 8 edges and 2 faces. Their purple overlay has 15 vertices, 21 edges and 7 faces. A point belongs to the overlay of two maps only if it belongs to both maps.

the Gauss map of an unbounded polyhedron does not cover the full unit sphere, as illustrated in Figure 1 (middle).

The link between Gauss maps and Minkowski sum is that the Gauss map of the Minkowski sum $A \oplus B$ is the overlay of the Gauss maps of A and B [3]:

$$\mathcal{G}(A \oplus B) = \text{Overlay of } \mathcal{G}(A) \text{ and } \mathcal{G}(B), \tag{4}$$

$$\mathcal{G}(A \ominus B) = \text{Overlay of } \mathcal{G}(A) \text{ and } \mathcal{G}(-B). \tag{5}$$

Figure 2 gives a visual definition of an overlay of two maps. For our purpose, the overlay is a map on $\mathcal{S}$ whose support is the intersection of the supports of the Gauss maps $\mathcal{G}(A)$ and $\mathcal{G}(B)$ and whose edges are obtained by superimposing the edges of $\mathcal{G}(A)$ over those of $\mathcal{G}(B)$. The overlay of two maps is therefore a refinement of both maps restricted to the part of the unit sphere where they are both defined.

From the definition of the Gauss map, we see that the normals of the facets bounding $A \ominus B$ are the vertices of $\mathcal{G}(A \ominus B)$. From (5), these are:

- the sets N_A (resp. N_{-B}) of normals of facets of A (resp. $-B$), that lie in $\mathcal{G}(-B)$ (resp. $\mathcal{G}(A)$) and
- the set N_{AB} of proper intersections of a dual-arc of $\mathcal{G}(A)$ and a dual-arc of $\mathcal{G}(-B)$. See Figure 1 (right).

Let us write $N_{A\ominus B}$ for the set of normals to the facets of $A \ominus B$:

$$N_{A\ominus B} = N_A \cup N_{-B} \cup N_{AB}. \tag{6}$$

We then derive

$$\begin{aligned} A \cap B \neq \emptyset \quad &\Leftrightarrow \quad 0 \in A \ominus B && (7)\\ &\Leftrightarrow \quad \forall n \in N_{A\ominus B}, \quad \max_{x \in A \ominus B} (n \cdot x) \geq 0 && (8)\\ &\Leftrightarrow \quad \forall n \in N_{A\ominus B}, \quad \max_{a \in A}(n \cdot a) - \min_{b \in B}(n \cdot b) \geq 0. && (9) \end{aligned}$$

Note that the extrema in (9) are finite since both A and B admit a tangent plane with any normal in $N_{A\ominus B}$. Writing h_n^A for the equation of the plane bounding A, tangent to A with normal n, we finally obtain

$$A \cap B \neq \emptyset \quad \Leftrightarrow \quad \forall n \in N_{A\ominus B}, \quad \min_{b\in B}(h_n^A(b)) \leq 0. \tag{10}$$

The intersection test (10) can be interpreted as creating a set of oriented planes tangent to A: $H(A,B) = \{h_n^A \mid n \in N_{A\ominus B}\}$, and testing that B touches the negative side of each of these planes. Of course, this is strictly equivalent to testing that the origin lies inside the Minkowski difference $A \ominus B$. Formulating the test as in (10) has advantages:

- It is in the same familiar form as the well-known conservative test that consists in testing the position of B with respect to the sides of A.
- The planes equations in $H(A,B)$ depend only on the Gauss map of B and not on the geometry of B. When B ranges over a set of axis-aligned boxes, which all have the same Gauss map, the set $H(A,B)$ is the same for, say, a fixed view-frustum A and any axis-aligned box B, and can therefore be precomputed. This corresponds exactly to the algorithm proposed by Greene [12]; see §2.5.

For an implementation of the form (10), polyhedra A and B and interchangeable, so ideally, we should label them A and B in such a way that it is easy to compute the constant term of the plane equation h_n^A given n, and more importantly, fast to minimize this affine function over B.

For any normal $n \in N_{A\ominus B}$, it is also possible to switch to the alternative equivalent test

$$\min_{a\in A}(h_{-n}^B(a)) \leq 0, \tag{11}$$

since $-n$ belongs to $N_{B\ominus A}$.

If we restrict test (10) to the set of normal vectors N_A instead of $N_{A\ominus B}$, we obtain the well known conservative test that checks the position of B with respect to the sides of A. The derivation of (10) demonstrates that by simply considering a few more planes tangent to A, the test becomes exact. These additional planes do not support a face of A but are tangent to A along an edge or a vertex.

Zones In order to compute the vertices of the overlay of two Gauss maps, we may try to exploit the particular structure of the maps at hand. In this context, a *zone* of length n on a polyhedron P is a sequence $(e_0, f_0, e_1, f_1, \ldots, f_{n-1}, e_n)$ such that

- the e_i are edges of P and pairwise parallel,
- the f_i are facets of P and pairwise distinct,
- $e_n = e_0$ and
- e_i and e_{i+1} are edges of facet f_i.

A polyhedron may or may not exhibit a zone. The existence of a zone on polyhedron P is equivalent to the existence of a set of dual-edges of $\mathcal{G}(P)$ whose union forms a complete great-circle on $\mathcal{S}$.

2.3 Intersection of two boxes

In the rest of Section 2, we review intersection detection techniques for boxes, that are ubiquitous in computer graphics, and see how they fit with equation 10. Boxes in particular have well structured Gauss maps.

2.3.1 The Gauss map of zonohedra

A zonohedron is a polyhedron obtained as the Minkowski sum of a set of line segments. The Gauss map of a line segment splits the sphere in two dual-hemispheres separated by a great-circle. Therefore, the Gauss map of a zonohedron generated by k line segments is an arrangement of k great-circles on $\mathcal{S}$. Equivalently, there exist a partition of the edges of a zonohedron such that each subset forms a zone.

We can then immediately derive the structure of the Gauss maps of oriented and axis-aligned boxes. An oriented box is a zonohedron generated by three pairwise orthogonal line segments, and its Gauss map is therefore the arrangement of three pairwise orthogonal great circles. An axis-aligned box is an oriented box whose generator directions coincide with the x, y and z axes of an orthogonal frame of reference; its Gauss map is the arrangement of the three great circles on the $x = 0$, $y = 0$ and $z = 0$ planes; see Figure 1 (left).

2.3.2 Intersection of two axis-aligned boxes

Since the Gauss maps of two axis-aligned boxes A and B are identical, their overlay is identical as well: $\mathcal{G}(A \ominus B) = \mathcal{G}(A) = \mathcal{G}(-B)$. Indeed, the Minkowski difference of two axis-aligned boxes is also an axis-aligned box. Therefore, as is very well known, six plane tests are sufficient and necessary to complete an exact intersection test between A and B. Furthermore, the six normal vectors in $N_{A\ominus B}$ are pairwise opposite and parallel to a base axis, which lets us interpret the six plane tests as three interval overlap tests, one along each base axis. This symmetry was exploited by Gottschalk [11, 10] for implementing an efficient intersection test between two oriented boxes; see §2.3.4.

2.3.3 Shaft culling

Shaft culling is a technique to accelerate visibility queries between two objects in a scene, due to Haines and Wallace [14]. It works by first constructing the shaft, which is the convex hull of the axis-aligned bounding boxes of both objects, and second, hierarchically culling the scene against the shaft.

They test for intersection between a shaft S and a bounding box B using equation (10) but only using the sides of S, eg the planes h_n^S with normal n in N_S. The authors write: "*We have constructed an informal proof showing that shafts do not give false positives for box testing, but have not formally studied what are necessary and sufficient conditions to avoid mis categorization.*" By examining the Gauss map of a shaft, we can easily prove that it is indeed sufficient (and necessary) to test the box against the sides of the shaft to obtain an exact intersection test:

Since a shaft is the convex hull of two axis-aligned boxes, it must contain three zones, the edges of which are parallel to each reference axis, respectively. (We can visualize one such zone by wrapping a tangent plane parallel to a fixed axis (say the x axis) around the shaft and observe that at any time, the tangent plane must contain one or two edge(s) parallel to the axis.) Therefore, the Gauss map $\mathcal{G}(S)$ of shaft S is a subdivision of the Gauss map $\mathcal{G}(-B)$ of the axis-aligned box $-B$. This implies that their overlay is equal to $\mathcal{G}(S)$ and $N_{S\ominus B} = N_S$, thereby proving, via equation (10), that the intersection test used by Haines and Wallace is indeed exact. Using more generally oriented bounding boxes (in the role of B) would require, however, additional planes tangent to the shaft in order to guarantee an exact intersection test.

2.3.4 Intersection of two oriented boxes and the separating axis test

The case of two oriented boxes has been studied in depth by Gottschalk [11, 10]. Let us consider two oriented boxes A and B. The overlay of their Gauss maps is the arrangement of six great-

circles. It contains $2 \times \binom{6}{2} = 30$ vertices, which come in 15 pairs of antipodal vertices.

A pair of antipodal vertices in $\mathcal{G}(A \ominus B)$ is responsible for two planes with opposite normals n and $-n$ in $H(A, B)$. The restriction of equation (10) to these two planes is equivalent to an interval overlap test on a line with direction n: Define the projection $p_n(X) = \{x \cdot n \mid x \in X\}$. Then,

$$\min_{b \in B}(h_n^A(b)) \leq 0 \quad \text{and} \quad \min_{b \in B}(h_{-n}^A(b)) \leq 0 \quad \Leftrightarrow \quad p_n(A) \cap p_n(B) \neq \emptyset. \tag{12}$$

This is the *separating axis test (SAT)* devised by Gottschalk. It becomes computationally interesting when, in addition, the interval overlap test can be optimized. This is the case when A and B are center symmetric. In that case, the overlap of $p_n(A)$ and $p_n(B)$ can be tested by comparing the distance between the projection of the centers of A and B and the sum of their projected radii, which affords a particularly efficient implementation when A and B are oriented boxes [10].

A word of caution It should be noted that the SAT is particularly well suited to testing the intersection of two oriented boxes, but may not be optimal for other shapes. In particular, it makes essential use of the fact that the vertices in $\mathcal{G}(A \ominus B)$ are all part of an antipodal pair. Since the interval overlap test along some direction n is equivalent to two plane tests (equation (12)), we see that this test performs unnecessary work when the direction n has no antipodal partner in the overlay, which is the case in general in the overlay of the Gauss maps of most pairs of shapes.

The SAT is often described as a generic method for detecting the intersection of two convex polyhedra. It is applied with the interval overlap test along the directions given by the normals of the facet of each shape, as well as the cross product of each normal of one shape with each normal of the other shape. This amounts to extending the dual-edge of each edge in both shapes to a full great-circle and using equation (10) on the resulting "augmented Gauss map"; one can see right away that redundant computations appear.

For example, consider two tetrahedra T and T'. Given an edge e of T, the edges of $-T'$ whose dual-edge crosses $e^\star$ in the Gauss maps overlay are silhouette edges of T' in the viewing direction e; see §2.4. The number of such silhouette edges is 3 or 4. So, the dual-edge of e intersects at most 4 dual-edges of $\mathcal{G}(-T')$. We then obtain an intersection test (10) using at most $4 + 4 + 6 \times 4 = 32$ plane tests. On the other hand, a direct application of the separating axis test leads to $4 + 4 + 6 \times 6 = 44$ interval overlap tests, mathematically equivalent to 88 plane tests. Of course, there are other computational factors that may mitigate this harsh comparison (for example, the vertices of the overlay are more difficult to compute than simply taking all the possible cross-products of any two edges), but this example clearly shows that the separating axis test is not the definitive answer to testing the intersection of two convex polyhedra in general (without preprocessing). We give a faster technique for pairs of tetrahedra in §4.

A generalization to zonohedra We have seen that zonohedra form a family of polyhedra whose Gauss map is an arrangement of great-circles on the unit sphere. Zonohedra are also center symmetric, so that the arguments that we used in favor of the interval overlap test for a pair of oriented boxes carry naturally to pairs of zonohedra. The facets of a zonohedron A generated by a set of n generators $G(A)$ are parallelograms generated by some pairs of generators from $G(A)$. The set of normal directions N_A is $\{g \times g' \mid g \neq g', g \text{ and } g' \in G(A)\}$. Let B be another zonohedron generated by m line segments and define $D_{AB} = \{g \times g' \mid g \in G(A), g' \in G(B)\}$. Note that $N_{AB} = D_{AB} \cup (-D_{AB})$. The set of directions along which the interval overlap test should be performed is $N_A \cup N_B \cup D_{AB}$. The projected radius of each zonohedron is half the sum of the projected length of the generators. We deduce that a fast implemention of the SAT

is probably feasible for small zonohedra. Guibas *et al.* designed more efficient algorithms that are required when n or m is large [13].

2.4 A silhouette condition

Let A and B be general convex polyhedra again. Let e_A be an edge of A and e_B an edge of B. With a slight abuse of notation, we write $e_B^\star$ for the dual-edge of $-e_B \in (-B)$. The first necessary condition for the dual-edges $e_A^\star$ and $e_B^\star$ to cross (thereby forming a test normal in N_{AB}) is that the end-dual-points of $e_A^\star$ (resp. $e_B^\star$) lie on either sides of the great circle supporting $e_B^\star$ (resp. $e_A^\star$). (A second necessary condition is detailed later in §3.1 but is not relevant here.) Consequently, each primal edge must be a silhouette edge of its polyhedron with respect to the view direction given by the other edge. This observation was leveraged by Greene as we describe now.

2.5 Intersection of a box and a view-frustum or a convex polyhedron

Greene proposes an algorithm for exactly testing the intersection of an axis-aligned box and a convex polyhedron [12]. A major application for this test is (hierarchical) view-frustum culling wherein the polyhedron is a 6-sided truncated pyramid. Let A be a convex polyhedron and B an axis-aligned box.

The separation axis test is used for testing the 6 plane tests with normals in N_B, which are implemented by computing the axis-aligned bounding box of polyhedron A and comparing its extents with those of B.

The plane tests with normals in N_A are not specifically optimized. However, Greene introduces a now famous optimization for testing the position of an axis-aligned box with respect to a plane, that considers only one or two vertices of B, instead of eight, via an examination of the signs of the coordinates of the normal vectors of N_A.

The plane tests with normals in N_{AB} are optimized as follows. In addition to recalling the silhouette condition of §2.4, Greene observes that an edge of A is silhouette with respect to an edge of B if and only if it is a boundary edge of the orthogonal projection of A on one of the three axis-aligned planes. The 3D plane tests then reduce to testing the position of a 2D axis-aligned rectangle against the sides of a 2D convex polygon, in the three projections along the x, y and z axes. When several boxes are tested against the polyhedron A, both the bounding box of A and its silhouette edges with respect to the x, y and z directions are precomputed to speed up the intersection test.

3 A low level optimization

In this section, we develop a generic and fast implementation of test (10) for two small convex polyhedra, A and B, when no particular assumption on their shapes can be made.

Let us look at the normal vectors in $N_{A\ominus B}$. If $n \in N_A$ then the associated plane test $\min_{b\in B}(h_n^A(b)) \leq 0$ simply corresponds to testing the position of B with respect to (the plane supporting) the facet of A whose normal is n. Similarly, if $n \in N_{-B}$ then the associated plane test $\min_{b\in B}(h_n^A(b)) \leq 0$, being equivalent to $\min_{a\in A}(h_{-n}^B(a)) \leq 0$, simply corresponds to testing the position of A with respect to (the plane supporting) the facet of B whose normal is $-n$. (Recall that $n \in N_{-B} \Leftrightarrow -n \in N_B$.) So, the intersection test (10) restricted to normals in $N_A \cup N_{-B}$ is the well known conservative test that consists in testing the position of A with respect to the sides of B, and *vice versa*. In order to obtain an exact test we must augment the set of test

normals with the ones in N_{AB} that arise as the proper intersections of two dual-edges in the Gauss maps of A and $-B$. The pseudo-code for the generic algorithm is given in Algorithm 1.

Algorithm 1 A generic intersection test for small convex polyhedra.

```
1:  function QuadraticIntersectionTest(A,B)          ▷ Returns True iff A ∩ B ≠ ∅
2:      for all facet f of A do
3:          if B lies in the positive side of the support plane of f then return False
4:      for all facet f of B do
5:          if A lies in the positive side of the support plane of f then return False
6:      for all edge e_B of B do
7:          for all edge e_A of A do
8:              if TwoEdgesSeparation(e_A,e_B) then return False
9:      return True
10: function TwoEdgesSeparation(e_A,e_B)
11:                  ▷ We abuse the notation and write e*_B for the dual-edge of edge −e_B ∈ (−B).
12:     if dual-edges e*_A and e*_B intersect then
13:         Let n = e*_A ∩ e*_B, v_A ∈ e_A and v_B ∈ e_B.
14:         Compute plane equation h^A_n using v_A
15:         if h^A_n(v_B) > 0 then return True
16:     return False
```

The first two **for** loops are straightforward. The function TwoEdgesSeparation is more involved. We focus on the dual-edge intersection computation on line 12.

3.1 Computing the intersection of two dual-edges

Let n_A, n'_A, n_B, n'_B be the normal vectors of the facets adjacent to edges e_A and e_B respectively. They are also the dual-end-points of $e^\star_A$ and $e^\star_B$. We fix an arbitrary orientation for edges e_A and e_B and see them as vectors. Then we can write the silhouette condition from §2.4 like so:

$$d_A d'_A < 0 \quad \text{and} \quad d_B d'_B < 0 \tag{13}$$

$$\text{where} \quad d_A = n_A \cdot e_B, \quad d'_A = n'_A \cdot e_B, \quad d_B = n_B \cdot e_A \quad \text{and} \quad d'_B = n'_B \cdot e_A. \tag{14}$$

Recall that the cross-product of e_A and e_B gives a vector that is collinear to the potential intersection of $e^\star_A$ and $e^\star_B$ but may have the opposite orientation, since we have no control over the orientation of the edges. Instead of using a cross-product, we build the intersection x_A of $e^\star_A$ with the great-circle supporting $e^\star_B$ and then check that it belongs to the dual-edge $e^\star_B$:

$$x_A \quad = \quad \frac{u}{||u||} \quad \text{where} \quad u = |d_A| n'_A + |d'_A| n_A. \tag{15}$$

The vector x_A exists—and is computed—only when e_A is silhouette. As a positive linear combination of n_A and n'_A, the vector x_A is guaranteed to be orthogonal to e_A and the tangent plane $h^A_{x_A}$ is tangent to A along edge e_A; in other words, $x_A \in e^\star_A$. The coefficients of the combination also make x_A orthogonal to e_B. It is then immediate to see that

$$e^\star_A \cap e^\star_B \neq \emptyset \quad \Leftrightarrow \quad x_A \in e^\star_B \quad \Leftrightarrow \quad e^\star_A \cap e^\star_B = x_A. \tag{16}$$

(In practice, it is not necessary to normalize x_A.) We now need to test if x_A lies in $e^\star_B$. Let f_B and f'_B be the facets of B sharing edge e_B, with normal vectors n_B and n'_B respectively. Let v_B

be a vertex on e_B, w_B a vertex of f_B not on e_B and w'_B a vertex of f'_B not on e_B. Using the fact that x_A is orthogonal to e_B, It is simple to check that

$$x_A \in e_B^\star \quad \Leftrightarrow \quad (x_A \cdot (v_B - w_B) < 0 \quad \text{and} \quad x_A \cdot (v_B - w'_B) < 0) \tag{17}$$

$$\Leftrightarrow \quad (x_A \cdot v_B < x_A \cdot w_B \quad \text{and} \quad x_A \cdot v_B < x_A \cdot w'_B) \tag{18}$$

If x_A is found to be in $e_B^\star$, we set n to x_A (line 13 above). Let v_A be a vertex on e_A. Then, the plane h_n^A separates A and B if and only if $n \cdot (v_B - v_A) > 0$ which is equivalent to $h_n^A(v_B) > 0$ (line 15 above). The pseudo-code for TwoEdgesSeparation is shown in Algorithm 2. Note

Algorithm 2 An implementation of TwoEdgesSeparation.

```
1: function TwoEdgesSeparation(e_A,e_B)
2:     d_A ← n_A · e_B
3:     d'_A ← n'_A · e_B
4:     if d_A d'_A ≥ 0 then return False                ▷ e_A is not silhouette w.r.t. e_B
5:     x_A ← |d_A| n'_A + |d'_A| n_A
6:     Let v_A, v_B, w_B and w'_B be defined as above
7:     if (x_A · v_B ≥ x_A · w_B  or  x_A · v_B ≥ x_A · w'_B) then
8:         return False                                 ▷ e*_A and e*_B do not intersect
9:     return x_A · v_B ≥ x_A · v_A                     ▷ is there a separating plane?
```

that in line 9, the minimization of the plane equation over B is implicit; Indeed, we already know that the linear equation is minimized (over B) at vertex v_B, since we know exactly which edges (e_A and e_B) are responsible for the appearance of the normal vector x_A on the overlay of the Gauss maps. Thus, TwoEdgesSeparation takes constant time (assuming we can query the vertices involved in constant time) and the loop over the pairs of edges takes time $O(E_A E_B)$ where E_X is the number of edges of polyhedron X.

Alternatively, we can simply skip checking that x_A is indeed a vertex of the overlay and replace lines 6 to 9 with any other minimization scheme. In particular, we will see in §4 that for pairs of tetrahedra, it is a little bit faster to explicitly compute the value of h_n^A at the four vertices of B.

Both alternatives benefit from a simple arithmetic optimization using the fact that x_A is a linear combination of normal vectors of facets of A, letting us avoid all the 3D dot-products that appear in function TwoEdgesSeparation loop and replace them with 2D dot-products or simple subtractions. We describe this optimization next.

3.2 An arithmetic optimization

We define the matrix Δ of the dot-products of normals of A and vertices of B:

$$\Delta_{ij} \quad = \quad n_i^A \cdot v_j^B, \tag{19}$$

where n_i^A is the outward normal vector of the i-th facet of A and v_j^B is the j-th vertex of B. We also define the vector $M = (M_i)$ as

$$M_i \quad = \quad \max_{a \in A} (n_i^A \cdot a) \quad = \quad n_i^A \cdot v(n_i^A), \tag{20}$$

where $v(n_i^A)$ is any vertex of the facet of A whose normal is n_i^A.

Both Δ and M are readily computed when we test the position of B with respect to the sides of A (Algorithm 1, lines 2-3). Indeed, for the facet of A with normal n_i^A, the corresponding plane test can be stated like so:

$$n_i^A \cdot v(n_i^A) \quad \leq \quad \min_j (n_i^A \cdot v_j^B). \tag{21}$$

The left-hand side forms the i-th element of vector M while the dot-products on the right-hand side form the i-th row of matrix Δ. We therefore just need to allocate memory for Δ and M and store their elements as they are computed in Algorithm 1, lines 2-3.

It turns out that all the dot-products in Algorithm 2 can be expressed using Δ and M. Assume that edge e_B connects vertices v_k^B and v_l^B, and that edge e_A is adjacent to facets f_i^A and f_j^A with normal vector n_i^A and n_j^A respectively. Then Algorithm 2, lines 2 and 3 become

$$d_A \leftarrow \Delta_{il} - \Delta_{ik}, \tag{22}$$
$$d'_A \leftarrow \Delta_{jl} - \Delta_{jk}. \tag{23}$$

The dot-product of x_A with any vertex v_k^B of B can be computed as

$$\begin{aligned} x_A \cdot v_k^B &= (|d_A| n_j^A + |d'_A| n_i^A) \cdot v_k^B && (24)\\ &= |d_A| n_j^A \cdot v_k^B + |d'_A| n_i^A \cdot v_k^B && (25)\\ &= |d_A| \Delta_{jk} + |d'_A| \Delta_{ik}. && (26) \end{aligned}$$

Finally, the dot-product of x_A with a vertex v_A of edge e_A can be computed as follows:

$$\begin{aligned} x_A \cdot v_A &= (|d_A| n_j^A + |d'_A| n_i^A) \cdot v_A && (27)\\ &= |d_A| n_j^A \cdot v_A + |d'_A| n_i^A \cdot v_A && (28)\\ &= |d_A| n_j^A \cdot v(n_j^A) + |d'_A| n_i^A \cdot v(n_i^A) && (29)\\ &= |d_A| M_j + |d'_A| M_i. && (30) \end{aligned}$$

Equation (29) holds because vertices v_A and $v(n_j^A)$ both lie on facet f_j^A whose normal vector is n_j^A and similarly for v_A and $v(n_i^A)$. Since all dot-products with x_A are now expressed in terms of Δ and M, it is not even necessary to compute the 3D vector x_A.

We find it noteworthy that the arithmetic optimization describe here let us obtain similar "2D computations" as with the projections along the three canonical axes in Greene's algorithm (see §2.5). Our optimization works in a generalized context that puts no constraint on either polyhedra, beside being convex.

In the rest of the paper, we refer to our implementation of the technique described in this section as the **Generic** implementation.

4 Application to pairs of tetrahedra

In this section, we test a specialized implementation of the generic algorithm described in §3 for detecting intersecting pairs of tetrahedra. The specialization is two-fold. First, tetrahedra always have the same combinatorics, so we store it only once in static arrays, and take advantage of its structure as much as possible. Second, since tetrahedra have few vertices, we found it faster to evaluate the plane equation explicitly on all four vertices, in Algorithm 2 using equation (26), instead of evaluating it for just the three vertices v_B, w_B and w'_B (line 7). We believe that the

explicit computation on four vertices is faster because their indices are hard-coded (0, 1, 2 and 3) instead of being stored in variables which incurs a slower indirect memory addressing.

We compare three implementations, all of which use the hard-coded combinatorics for tetrahedra. The **tetra0** implementation implements the algorithm described in §3.2. The **tetra1** implementation explicitly minimizes the plane equation over the four vertices. Both **tetra0** and **tetra1** are specialized versions of the **Generic** implementation. The **SAT** implementation tests all pairs of edges using the separation axis theorem.

The test generates $N = 10^4$ random tetrahedra in the unit cube and then tests all $\binom{N}{2}$ pairs of distinct tetrahedra for intersection:

N	% hits	**SAT**	**tetra0**	**tetra1**
10^4	61.17 %	25.73 s.	15.54 s.	13.06 s.
ratio to **SAT**		1.0	0.6	0.51

Among all the pairs, 61.17 % are intersecting. The specialized technique **tetra1** is about twice faster than **SAT**. We have also implemented an exact implementation of the test using the CGAL library, which lets us test whether our floating-point implementations succeeded in computing the correct result or not:

	SAT succeeds	**SAT** fails
tetra1 succeeds	49994974	2
tetra1 fails	23	1

The table above says, for example, that **SAT** gave the wrong answer 3 times, while **tetra1** did so 24 times out of $\binom{N}{2} = 49995000$. The implementation **tetra1** fails more often than **SAT** because of the larger degree of the arithmetic expression it uses, when combining the precomputed entries of matrix Δ and vector M (§3.2). But the proportion of failures is still relatively low and is probably acceptable if one is not willing to use slower but exact geometric predicates. We provide more extensive benchmarks in §8.

5 Application to packet ray tracing

Frustum culling is the process of testing parts of a 3D scene against the viewing volume of a virtual camera (the *view frustum*) in order to avoid drawing objects guaranteed to not be visible. To make the culling efficient, simple bounding volumes are tested in place of the complex geometry they enclose. The fastest methods use hierarchies of bounding volumes and a corresponding recursive frustum culling algorithm [1].

Since the view frustum is most often a truncated pyramid, a convex polyhedron, a quick way to test a volume against the frustum is to check its relation with respect to each side of the volume. If the volume if found to lie outside (the support plane of) a side of the frustum, then the volume is guaranteed to not intersect the frustum and the "content" of the volume can be ignored. This is a *conservative test* since it can decide that a volume touches the frustum although it does not, in which case some time is wasted in processing invisible parts of the scene. Clearly an exact test would cull more bounding volumes and would be of interest if it can be made fast enough. Fast exact intersection tests are known for some pairs of shapes [12, 11, 14].

In this section, we focus on casting rectangular packets of rays. For this purpose we replace the frustum by an unbounded view-pyramid. We use equation 10 to develop a specialized technique for testing the intersection of such a view-pyramid against axis-aligned (bounding) boxes and triangles. We apply the resulting technique to the tracing of rectangular packets of rays as used in high performance ray tracers (see [18] and the references within). The primary rays that

correspond to a rectangular region of the image plane, can be bounded by a *view-pyramid*: a four-sided unbounded pyramid with apex at the camera position. Groups of shadow-rays with a same origin can similarly be bounded by such a view-pyramid.

A view-pyramid is then a simplified view-frustum without near- or far-planes, and we can adapt Greene's algorithm (§2.5) to test for the intersection of a view-pyramid A and an axis-aligned box B. In the following, we detail this procedure and show that it improves performance of the technique of Wald *et al.* [18].

The Gauss map $\mathcal{G}(A)$ of the view-pyramid A is a single spherical quadrangle; see Figure 1(middle). As such, it does not cover the full unit sphere. Therefore, parts of the Gauss map of B will not contribute to its overlay with $\mathcal{G}(A)$ (Figure 1(right)).

We will also use the four rays with origin the apex of A and directed along the four edges of A. These are often called the four *corner rays* of A.

5.1 Conservative pyramid-AABB test

In computer graphics applications, it is typical to implement a *conservative* intersection test for pyramid A and box B by using only the set of normals N_A, instead of implementing an exact test using all the normals in $N_{A\ominus B}$. This conservative test checks the position of B relative to the four sides of A. Note that all the planes used depend only on the geometry of the pyramid, and so are computed once and used to test several boxes.

5.2 Exact pyramid-AABB test

To perform an exact test, one needs to compute the additional planes tangent to A with normals in N_{-B} and N_{AB}. Since all the axis-aligned boxes have the same Gauss map, we can also precompute these planes once, for a given pyramid, and use them to test any number of axis-aligned boxes. Note in particular that when B is an axis-aligned box, $\mathcal{G}(-B) = \mathcal{G}(B)$.

5.2.1 N_{-B}

N_{-B} is the set of normals to facets of B that do lie inside the Gauss map of A (the spherical quadrangle). The normals of facets of B are the three orthonormal basis vectors and their three opposites, so that we can find which do lie in $\mathcal{G}(A)$ by looking at the signs of the coordinates of the four corner rays. For example, if all four corner rays' directions have a negative y component, then $e_{+y} = (0, +1, 0)^T \in N_{-B}$ and the plane test associated with e_{+y} simply consist in comparing the y coordinate of the apex of A with the lowest y coordinate over B. We remark that N_{-B} contains at most three vectors when A is a view-pyramid, while it contains *exactly* 6 vectors when A is a bounded view-frustum.

Our experiments indicate that these planes have a little culling power, because the facets of $A \ominus B$ that they generate have size equal to the size of the facets of B and are gathered around the viewpoint which is most often far from surfaces. Thus, we choose to omit them in our implementation. This implies that our more precise test is also conservative instead of being completely exact.

5.2.2 N_{AB}

By contrast, the planes with normals in N_{AB} are very effective at culling more boxes, because their corresponding facets on $A \ominus B$ are unbounded and thus have an influence on all boxes close to the pyramid. To compute them efficiently, we take advantage of the simple geometry of the Gauss map of an axis-aligned box: a dual-arc of an edge of A crosses a dual-arc of an

edge of B only if its two dual-endpoints have different signs for some component x, y or z. See Figure 1 (middle). When that is the case, a linear combination of the endpoints gives the direction vector of a normal in N_{AB}. As in §2.5 each normal vector in N_{AB} is orthogonal to a world basis vector and the corresponding plane test can thus be carried in 2D.

From the silhouette condition, we deduce that the number of normals in N_{AB} is either 0, 2, 4, or 6. N_{AB} is probably empty when the view-pyramid has a very large field of view, which makes its Gauss map very small, in which case it is unlikely to intersect any dual-edge of $\mathcal{G}(B)$. On the other hand, N_{AB} is likely to contain 6 normal vectors when the view-pyramid has a very small field of view, in which case its Gauss map spans almost a half-sphere and its boundary probably crosses the three great-circles in $\mathcal{G}(B)$ twice each.

5.3 Intersection of a view-pyramid and a triangle

In this section, A is again a view-pyramid, and B is a triangle. For implementing a conservative test, we use the four normals in N_A and, given a plane equation $h_n^A, n \in N_A$, we compute $\min_{b\in B}(h_n^A(b))$ by computing the values of h^A at the three corners of the triangle. For the same reasons as for axis-aligned boxes, we may avoid considering the normal vector(s) in N_{-B}, without loosing much culling power.

The Gauss map of a triangle has three dual-arcs (Figure 3). Since these arcs are different for two different triangles, we can not precompute the set of planes $\{h_n^A, n \in N_{AB}\}$. Instead, to complete our almost-exact test, we combine the computation of N_{AB} and the minimization of the corresponding plane equations into a single procedure. The procedure considers the three dual-arcs of the Gauss map of the triangle in turn, and compute the normals that it contributes to N_{AB} with the four dual-arcs of $\mathcal{G}(A)$ at once, using SIMD instructions. By comparing the sign of the dot product of these normal vector with the pyramid apex and the triangle vertex not on the current triangle edge, we can compute the result of the intersection test.

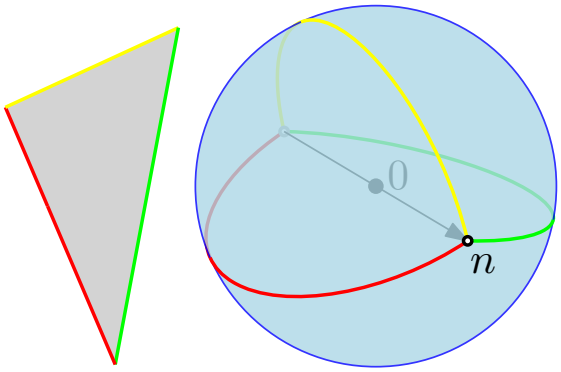


Figure 3: A triangle and its Gauss map.

This is equivalent to intersecting the pyramid with the plane supporting the triangle and testing the resulting polygon against each edge of the triangle, on that plane, as is already known (eg [17]). However, our version casts the test in a more general setting and its implementation does not need any projection or division and does not need the triangle's normal.

⋆ Our C++ code for testing the triangle against the planes with normal in N_{AB} is available as supplemental material.

5.4 Integration in Wald *et al.* traversal procedure

The Bounding Volume Hierachy (BVH) traversal algorithm of Wald *et al.* is given below. It traverses the BVH for a packet (a set) of rays P, which, in our implementation, corresponds to a 16×16 square block of pixels. The view-pyramid in the function tightly bounds the packet P. The integer `far` is the index of the first ray in the packet that touches node n (the `first active ray`).

1: **function** W+IA(BVH Node n, int `far`, Packet P)

```
2:    if P[far] does not hit n's bounding box then
3:        if n does not hit the view-pyramid then return
4:        while P[far] does not hit n's bounding box do
5:            increment far
6:            if far≥ |P| then return
7:    if n is a leaf node then
8:        for r =far to |P| − 1 do
9:            if P[r] touches n's bounding box then
10:               Intersect ray P[r] with the triangle
11:   else                                          ▷ n is an inner node.
12:       for all child n′ of n do W+IA(n′, far, P)
```

At line 3, the view-pyramid test is performed using interval arithmetic [18]. We modify the traversal as follows to obtain our W+**exact** traversal:

- We replace the test in line 3 with an exact view-pyramid/box test.
- Before line 8, we insert an exact view-pyramid/triangle test and exit if the test fails.

5.4.1 A note on packets and on the BVH

We have obtained best performance with packets of 16×16 primary rays. We use this packet size throughout this section. For these large packets, we have found that it is better to refine the BVH down to a single triangle per leaf. This is why a single triangle is tested in line 10. In all our experiments, the BVH is built using the algorithm of Wald *et al.* [18] which we modify slightly to fully refine the hierarchy.

5.4.2 Comparison of W+IA and W+exact

In this section, we experiment on a laptop with hardware similar to that used in [18]: a late 2007 MacBookPro laptop with an Intel Core 2 Duo CPU clocking at 2.6 Ghz, with DDR2 RAM clocking at 667 MHz. (The cited paper uses a desktop PC equipped with a dual-core Opteron clocking at 2.6 GHz.)

The **while** loop at line 4 ensures that all the nodes traversed are hit by at least one ray from packet P. Therefore, in both W+IA and W+**exact**, the number of nodes traversed is exactly the same. Performance improvement can come from a more precise view-pyramid culling of nodes at line 3 or a fast view-pyramid culling of triangles just before line 8.

Our experiments indicate that culling *nodes* with interval arithmetic is very precise: it culls barely less nodes that our almost exact test. We attribute this very good behavior to the arithmetic simplicity of the slab test as implemented with interval arithmetic. Indeed, the only approximation in the input is that of the set of inverse ray directions with a bounding cube (a cartesian product of 3 intervals). Further, the interval arithmetic operations of a slab test (one exact subtraction and one interval multiplication) do not introduce additional source of approximation [4].

Therefore, in the table below, we only vary the *triangle* culling technique inserted before line 8 and measure its effectiveness. In particular we compare our exact culling of triangle (W+**exact**) with the well known culling using the four planes of the view-pyramid (called "W+simple" in the table). Note that W+IA does not cull triangles explicitly. In line 1 of the table, we show the success rate of this simple conservative triangle test. Line 2 uses our almost-exact test for triangles, which is able to cull twice more triangles. Line 3 and 4 compare the original and our implementation of W+IA: Ours is faster, which is an indication of the fairness of our comparison.

The figures in the table were obtained for an output image resolution of 1024×1024, constant color, diffuse shading, no shadow and two concurrent threads.

		conference	**fairy**
1	W+simple, % tris culled	16.6 %	17.3 %
2	Our W+**exact**,% tris culled	32.4 %	33.2 %
3	W+IA from [18] FPS	10.5	6.1
4	W+IA FPS	15.1	12.7
5	W+simple FPS	16.3	13.7
6	Our W+**exact** FPS	18	15.1

We believe that the difference in FPS between the original code and our implementation of W+IA is due to the fully refined BVH, the possible difference in the amount of code making use of SSE instructions and the difference in hardware (our laptop seems to have a larger L2 cache).

6 The optimization point of view: SphereSearch

In this section, we use the knowledge of the overlay of Gauss maps that we developed above to design a novel algorithm for testing the intersection of two convex objects. Our algorithm is similar in spirit to the algorithms of Gilbert, Johnson and Keerthi [9] and Gilbert and Foo [8], which we call GJK for short: a sequence of candidate directions are generated in which the two convex objects A and B are tested for separation using equation (9) until a final decision can be taken. Our algorithm differs from their in the way a new test direction is generated (they search a point closest to the origin on a tetrahedron inside $A \ominus B$, we pick a point in a convex spherical polygon) and in that we do not seek the actual distance between the two convex objects but only whether they touch or not (as also studied in [8]).

Let D be the function defined over the unit sphere $\mathcal{S}$ as

$$D : \mathcal{S} \to \mathbb{R}, n \mapsto \max_{a \in A}(n \cdot a) - \min_{b \in B}(n \cdot b). \tag{31}$$

Then, A and B intersect if and only if D takes a non-negative value all over the full sphere:

$$A \cap B \neq \emptyset \quad \Leftrightarrow \quad \forall n \in \mathcal{S}, D(n) \geq 0 \tag{32}$$

$$A \cap B = \emptyset \quad \Leftrightarrow \quad \exists n \in \mathcal{S}, D(n) < 0. \tag{33}$$

Note that these equations apply to any convex objects, not necessarily polyhedra. Our algorithm follows this idea and searches over the unit sphere for a direction n in which $D(n)$ is negative, or decides that D is everywhere non-negative. When v is a non-zero vector, let $v^\uparrow$ denote the north-hemisphere of $\mathcal{S}$ whose north pole is $v/|v|$: $v^\uparrow = \{n' \in \mathcal{S} \mid v \cdot n' \geq 0\}$. To design our search procedure, we use the following

Lemma 1. *Let $a \in A$ and $b \in B$. Let $n = \dfrac{a-b}{|a-b|}$ so that $n \in \mathcal{S}$. Then $D(n) \geq 0$ and, for all $n' \in n^\uparrow$, $D(n') \geq 0$.*

Proof. Clearly, $n \cdot a - n \cdot b$ is non-negative, which proves that $D(n)$ is non-negative as well. If $n' \cdot n \geq 0$ then $n' \cdot a - n' \cdot b = n' \cdot (|a-b|n) \geq 0$. Therefore $D(n') \geq 0$. □

The lemma above states that function D takes a non-negative value on any direction $n \in (a-b)^\uparrow$. We use this property to prune parts of the unit sphere in which we can not find a direction n making D negative. Our search procedure takes as parameters a *search polygon* S: a convex spherical polygon whose interior, $\mathring{S}$, is guaranteed to contain $D^{-1}(\mathbb{R}^-)$ (the preimage by D of the negative numbers), and a direction $n \in \mathring{S}$:

1: **function** SphereSearch(A, B, n, S) ▷ It holds that $n \in \mathring{S}$
2: $a \leftarrow \arg\max_{p \in A} n \cdot p$
3: $b \leftarrow \arg\min_{p \in B} n \cdot p$ ▷ $D(n) = n \cdot (a-b)$
4: **if** $D(n) < 0$ **then return** False ▷ A and B do not intersect
5: $S' \leftarrow S \cap (b-a)^\uparrow$ ▷ Now, $n \notin \mathring{S}'$ since $n \cdot (b-a) \leq 0$
6: **if** S' is empty **then return** True ▷ A and B do intersect
7: $n' \leftarrow$ a center point of S'
8: SphereSearch(A, B, n', S')

To test if A and B intersect, we first pick some points $a \in A$ and $b \in B$ (the respective centers of A and B might be good candidates). If $a = b$ then we are done. Otherwise, we compute $n \leftarrow b - a$ and call SphereSearch(A, B, $n/|n|$, $n^\uparrow$).

Initially, the spherical polygon S is set to an hemisphere, and the first test direction is the center n (or pole) of that hemisphere. If $D(n) \geq 0$, then direction n fails to separate B from A and a new direction must be tested. The extremal points $a \in A$ and $b \in B$ that were computed while computing $D(n)$ are put to use to prune a part of the search polygon: since we know, by Lemma 1, that D takes a non-negative value over $(a-b)^\uparrow$ the search polygon can be reduced to $S \cap (b-a)^\uparrow$.

For the search to be as quick as possible, we should prune as much of S as possible. We should ideally choose the test direction $n \in S$ in such a way that any hemisphere that contains n (in particular $(b-a)^\uparrow$ in line 5) contains at least a constant fraction of the area of S. The solution for a discrete version of this problem is known as the *centerpoint* [5]. For our convex spherical problem, we don't know how to find such an "area centerpoint." Our implementation approximates it by the re-normalized average of the vertices of S, which behaves well in practice.

The SphereSearch algorithm shares several interesting properties with that of Gilbert and Foo [8]:

- The only operation needed on the convex object is to find the extremal point in a given direction. Therefore, it works on any kind of convex objects for which the extremal point can be effectively computed, not just polyhedra. For example, take A as a sphere and B as a view-frustum and SphereSearch becomes an exact frustum culling algorithm for spheres.

- It works just as well on unbounded convex objects such as lines, rays or view-pyramid. In the case of unbounded polyhedra, we avoid treating infinitely far extremal points as a special case by simply initializing the spherical polygon S to the intersection of the supports of the Gauss maps of A and B. (The Gauss map of a ray with direction n is the hemisphere $(-n)^\uparrow$; the Gauss map of a line is reduced to a single great-circle whose north pole is in the direction of the line, and we have already seen that the Gauss map of a view-pyramid is a convex quad.) This limits the extrema to finite points only without any restriction since extrema at infinity always lead to a positive infinite value of D.

Compared to the GJK algorithm, our algorithm

- can not compute the distance between A and B, but only gives a yes/no answer; this lets it conclude that A and B do not intersect using less test directions since the actual distance between A and B is not needed.

- is able to use more of the information computed in previous stage of the algorithm. This lowers the average number of directions to be tested. See §7.
- Importantly, SphereSearch has a much lower failure rate than GJK, where a failure means entering in an infinite loop because to numerical inaccuracy. See §7.3.

The last detail of our algorithm is the computation of $S \cap (b-a)^{\uparrow}$. This intersection is simple to compute since any algorithm for clipping a convex polygon with a half-plane can be adapted to clip a convex spherical polygon with a hemisphere, with a tiny extension to account for lunes: spherical polygons with two sides only.

6.1 Application to ellipsoids and zonohedra

With more degrees of freedom, ellipsoids and zonohedra provide tighter bounding volumes for complex geometry, compared to bounding boxes. Yet, tight bounding volumes should be coupled with a fast technique for detecting intersection of these volumes with each other or with other shapes.

Testing pairs of ellipsoids for intersection with an algebraic approach is much more involved than for spheres, for example, as it requires the computation of the sign of the roots of polynomials of degree at least four [19, 6]. In contrast, our algorithm only requires a procedure to compute the extremal point on the ellipsoid in a given query direction, which is easier to compute. Assuming the ellipsoid E is modeled as an affine transformation of the unit sphere, $E = c_E + M(\mathcal{S})$, where M is an 3×3 matrix, then the extremal points of E along the query direction $n \in \mathcal{S}$ are

$$\arg\max_{p \in E} (p \cdot n) = c_E + r \quad \text{and} \quad \arg\min_{p \in E} (p \cdot n) = c_E - r \tag{34}$$

$$\text{where} \quad r = \frac{M'n}{\sqrt{n^T M' n}} \quad \text{and} \quad M' = MM^T. \tag{35}$$

It is similarly easy to apply our SphereSearch algorithm to zonohedra (see §2.3.1 and §2.3.4). Let Z be a zonohedron generated by k centrally symmetric line segments: $Z = c_Z + (s_1 \oplus s_2 \oplus \cdots \oplus s_k)$ where $s_i = \{\lambda v_i \mid \lambda \in [-1, 1]\}$ and $v_i \in \mathbb{R}^3$. Let us define the sign function $\text{sign}(x) = 1$ if $x \geq 0$ and $\text{sign}(x) = -1$ otherwise. Then

$$\arg\max_{p \in Z} (p \cdot n) = c_Z + r \quad \text{and} \quad \arg\min_{p \in Z} (p \cdot n) = c_Z - r \tag{36}$$

$$\text{where} \quad r = \sum_{i=1}^{k} \text{sign}(v_i \cdot n) v_i. \tag{37}$$

Using these simple calculations inside the SphereSearch algorithm, hierarchies of bounding ellipsoids or zonohedra might become competitive with bounding box hierarchies. We set experiments along these lines as future work.

7 SphereSearch v.s. GJK

7.1 A characterisation of the separating planes

This section derives a characterisation of the planes separating two convex objects that we use in §7.2 to understand the differences between our algorithm and GJK.

Note that we consider a trivial variant of GJK that does not care about computing the actual distance between A and B but returns as soon as it is possible to decide whether the objects do intersect or not.

First, recall that testing that A and B touch each other is equivalent to testing that the origin O lies in the Minkowski difference $A \ominus B$. In this section, in order to simplify the exposition, we therefore consider the geometrically [2] equivalent problem of testing that an object P contains the origin O. We assume that P is convex, which is the case when $P = A \ominus B$ and both A and B are convex. Following the earlier definition of function D, we define

$$D(P, n) = \max_{p \in P} (n \cdot p). \tag{38}$$

Recall that $0 \notin P \Leftrightarrow \exists n, D(P, n) < 0$. Lemma 1 tells us that for all $p \in P$, it holds that $D(P, p) \geq 0$ and $\forall n \in p^{\uparrow}, D(P, n) \geq 0$. (As a special case, we set $O^{\uparrow} = O^{\downarrow} = \mathcal{S}$.) Define

$$v^{\downarrow} = (-v)^{\uparrow}, \tag{39}$$

$$\mathcal{S}^{-}(P) = \{n \in \mathcal{S} \mid D(P, n) \leq 0\} \quad \text{(the } \textit{separating set} \text{ of } P\text{)}. \tag{40}$$

The set $\mathcal{S}^{-}(P)$ is the set of normals of the planes tangent to P with P on their negative side and O on their positive side. For this reason, we call $\mathcal{S}^{-}(P)$ the *separating set* of P.

Lemma 2. $\mathcal{S}^{-}(P) = \bigcap_{p \in P} p^{\downarrow}$.

Proof. $n \in \mathcal{S}^{-}(P) \Leftrightarrow D(P, n) \leq 0 \Leftrightarrow \forall p \in P, n \cdot p \leq 0 \Leftrightarrow \forall p \in P, n \in p^{\downarrow} \Leftrightarrow n \in \bigcap_{p \in P} p^{\downarrow}$. □

Define the *silhouette* of P, $\mathrm{sil}(P)$ as the set of points $p \in \partial P$ such that P admits a tangent plane in p with (outward) normal n that satisfies $D(P, n) = 0$. Lemma 3 below shows that the separating set of P depends only on the silhouette of P.

Lemma 3. *If O is not in the interior of P then $\mathcal{S}^{-}(P) = \bigcap_{p \in \mathrm{sil}(P)} p^{\downarrow}$. (Otherwise $\mathcal{S}^{-}(P)$ is empty.)*

Proof. We have $\mathrm{sil}(P) \subset P$ which implies that $\mathcal{S}^{-}(P) \subset \bigcap_{p \in \mathrm{sil}(P)} p^{\downarrow}$. In the other direction, let $n \in \bigcap_{p \in \mathrm{sil}(P)} p^{\downarrow}$. Any point $p \in P$ can be expressed as $\alpha s_1 + \beta s_2$ where s_1 and s_2 belong to $\mathrm{sil}(P)$, $\alpha \geq 0$ and $\beta \geq 0$. Since $n \in \bigcap_{p \in \mathrm{sil}(P)} p^{\downarrow}$, we have that $n \cdot s_1 \leq 0$ and $n \cdot s_2 \leq 0$. Therefore $n \cdot p \leq 0$ and $n \in \mathcal{S}^{-}(P)$. □

When P is a polyhedron, the separating set of P depends only on its silhouette vertices:

Lemma 4. *When P is a polyhedron, its silhouette* $\mathrm{sil}(P)$ *is a subset of its faces (vertices, edges and facets). Its silhouette vertices are sufficient to define $\mathcal{S}^{-}(P)$: If O is not in the interior of P then $\mathcal{S}^{-}(P)$ is a non empty convex spherical polygon on $\mathcal{S}$:*

$$\mathcal{S}^{-}(P) = \bigcap_{v,\ \textit{vertex of}\ \mathrm{sil}(P)} v^{\downarrow} = \bigcap_{v,\ \textit{vertex of}\ P} v^{\downarrow}. \tag{41}$$

Proof. By considering and simplifying the contribution of each silhouette edge. □

[2] but not computationally.

7.2 Comparing SphereSearch and GJK

Technically, the GJK algorithm computes the distance between the origin and a convex polyhedron P, but it is straightforward to adapt it to simply test that O lies in P for any convex object P, as was suggested in [8]. In order to compare the GJK algorithm with ours, it is convenient to see both under the same light. To do so, we can describe both algorithms abstractly as follows:

1: **function** ConvexContainsOrigin(P, $\tilde{P}_i$) ▷ It is assumed that $O \notin \tilde{P}_i$.
2: ▷ P is a convex object, $\tilde{P}_i$ is a convex polyhedron that approximates P: $\tilde{P}_i \subset P$
3: $n \leftarrow$ a vector in the separating set of $\tilde{P}_i$ ▷ $\mathcal{S}^-(\tilde{P}_i) \neq \emptyset$ because $O \notin \tilde{P}_i$
4: $v_{i+1} \leftarrow \arg\max_{v \in P} n \cdot v$ ▷ $D(P, n) = n \cdot v_{i+1}$
5: **if** $D(P, n) < 0$ **then return** False ▷ $O \notin P$
6: $\tilde{P}_{i+1} \leftarrow$ better approximation of P using $\tilde{P}_i$ and v_{i+1}
7: **if** $O \in \tilde{P}_{i+1}$ **then return** True ▷ $O \in \tilde{P}_{i+1} \subset P$
8: ConvexContainsOrigin(P, $\tilde{P}_{i+1}$) ▷ $O \notin \tilde{P}_{i+1}$

Both algorithms are called initially using $\tilde{P}_0 = \{v_0\}, v_0 \in P$. We will also consider the set of all known constructed points of P: $V_i = \{v_0, v_1, v_2, \cdots, v_i\} \subset P$ generated in line 4. The algorithms differ in the polyhedron $\tilde{P}_i$ used to approximate P and in lines 3, 6 and 7. We now examine these differences in turn.

7.2.1 The polyhedron $\tilde{P}_i$

In SphereSearch (page 19), $\tilde{P}_i$ is simply the convex hull of all the known points of P: $\tilde{P}_i = \mathcal{H}(V_i)$. Note however that the algorithm does not store $\tilde{P}_i$ explicitly but stores a spherical polygon S that is guaranteed to contain $\mathcal{S}^-(P)$. Lemma 4 proves that indeed S is the separating set of $\mathcal{H}(V_i)$: $S = \mathcal{S}^-(\mathcal{H}(V_i))$. Importantly, $\mathcal{H}(V_i)$ is the best approximation of P that we can have knowing only the subset V_i of P, and therefore S is the tightest approximation of $\mathcal{S}^-(P)$ that one can construct with the knowledge that we have at this stage.

In GJK, $\tilde{P}_i$ is stored explicitly and is either a vertex, a line segment, a triangle or a tetrahedron. It is the convex hull of at most four points taken in V_i and including v_i. As such, $\tilde{P}_{GJK} \subset \tilde{P}_{\text{SphereSearch}}$, ie GJK considers approximations of P of lesser quality (they are smaller, so their separating sets are larger that those of SphereSearch). Also, there is no guarantee that the sequence of considered approximations is increasing, while SphereSearch does guarantee that $\tilde{P}_i \subset \tilde{P}_{i+1}$. Dually, our algorithm guarantees that $\mathcal{S}^-(\tilde{P}_{i+1})$ is a better approximation of $\mathcal{S}^-(P)$ than $\mathcal{S}^-(\tilde{P}_i)$, while GJK offers no such guarantee.

In our implementation, we regularly find separating sets S with 5 or 6 vertices while GJK can only produce spherical polygons $\mathcal{S}^-(\tilde{P}_{GJK})$ having at most 4 vertices (because $\tilde{P}_{GJK}$ has at most 4 silhouette vertices and Lemma 4). This is a direct evidence that our algorithm is able, in pratice as well as in theory, to use more information during its execution.

7.2.2 Line 7: testing for intersection

This line tests whether the origin lies in the approximation $\tilde{P}_{i+1}$ of P. In GJK, this geometric test is performed when $\tilde{P}_{i+1}$ is a tetrahedron as part of the picking of a new test direction (see below). In our algorithm SphereSearch, we know, by lemma 4, that the origin lies in $\tilde{P}_{i+1}$ simply when its separating set, S, is empty, which is trivial to check.

7.2.3 Line 3: picking a new test direction

In SphereSearch, we pick a vector n as a (approximate) center of $\mathcal{S}^-(\mathcal{H}(V_k))$. As we have seen earlier in the description of the algorithm, this ensures that a large part of S is pruned if n

fails to produce a separating plane (ie $n \notin \mathcal{S}^-(P)$) thereby heuristically accelerating the search for the separating set of P.

In contrast, GJK was originally designed to actually compute the closest point of P to the origin, To ensure that it is eventually found and that $\tilde{P}_i$ stays tractable (with 4 or fewer vertices), the vector n is chosen as the opposite of the closest point of $\tilde{P}_i$ to the origin: $n = -p$ where $p = \arg\min_{x \in \tilde{P}_i} |x|$. The vector n is indeed a direction in the separating set of $\tilde{P}_i$, but it is not necessarily centrally located in it. It is however locally optimal in the sense that is minimizes $n \mapsto D(\tilde{P}_i, n)$.

7.2.4 Line 6: updating the approximation $\tilde{P}_i$

In both algorithms we know that the silhouette of $\tilde{P}_{i+1}$ is different from that of $\tilde{P}_i$ and the vector $n \in \mathcal{S}^-(\tilde{P}_i)$ picked in line 3 disappears from $\mathcal{S}^-(\tilde{P}_{i+1})$ (by lemma 1), but only SphereSearch guarantees that $\tilde{P}_i \subset \tilde{P}_{i+1}$, or equivalently, $\mathcal{S}^-(\tilde{P}_{i+1}) \subset \mathcal{S}^-(\tilde{P}_i)$. In particular in GJK the vector n might appear again in a subsequent approximation $\tilde{P}_j, j > i+1$.

In SphereSearch, the separating set of $\tilde{P}_{i+1}$ is computed as the intersection of the separating set of $\tilde{P}_i$ (modeled as a spherical convex polygon) with the half-sphere $v_{i+1}^{\downarrow}$. This is algorithmically akin to polygon clipping in the plane.

In GJK, assuming that $O \notin \tilde{P}_i$, let f be the unique facet of $\tilde{P}_i$ that contains, in its interior, the closest point of $\tilde{P}_i$ to the origin. Then $\tilde{P}_{i+1}$ is set to $\mathcal{H}(f \cup \{v_{i+1}\})$ which is the convex hull of at most 4 affinely independent points.

Which is faster is not an easy question to answer to. GJK is very fast at first when $\tilde{P}_i$ has less than four vertices, but slower when $\tilde{P}_i$ is a tetrahedron. However the tetrahedron stage is seldom reached as a decision is often reached in less that four iterations. The iterations of SphereSearch all cost roughly the same. We then expect to see GJK perform best in easy cases, when the object P is close to being a polyhedron with very few facets and far from being "round". In that case, few iterations are required to reached a decision (typically less than four) and GJK is faster on average. In our experiments, the polyhedron $P = A \ominus B$ is often more complex and SphereSearch is slightly faster (except for frustum culling, see §8.5). Furthermore, SphereSearch is less prone to numerical inaccuracies caused by floating-point computation, thanks in part to its less complicated implementation. The next section describes this phenomenon and §8 shows experimentally the robustness of SphereSearch.

7.3 Numerical issues in GJK and SphereSearch

While the **Generic** and **SAT** techniques have a definite maximal number of plane tests to perform, this is not the case for the **GJK** and SphereSearch techniques in which the next test plane is iteratively constructed.

Typical implementation will use non-exact floating point numbers, so that it is possible that the implementations of **GJK** or SphereSearch enter an infinite loop. To remedy this problem, we force the implementation to exit when a maximal number of iterations, Θ, has been reached.[3] When this happens, we consider the intersection test to *have failed* and, conservatively, that the pair of convex objects at hand do intersect. Note that a failure may happen also when the objects do not actually intersect, in which case a wrong answer is reported.

In our statistics over a large number of tested pairs of objects, the second largest number of iterations is strictly smaller that $\Theta - 1$ (the largest one being Θ). This makes us confident that Θ is large enough to almost surely detect that the routine has entered an actual infinite loop.

[3] Our implementations limit the number of iterations to $\Theta = 20$ for SphereSearch and **GJK**.

In this context, we will see in §8 that SphereSearch is more stable than **GJK**, in the sense that our SphereSearch implementation fails less often than our **GJK** implementation. In fact, while the failure rate of both implementation is rather small, the failure rate of SphereSearch is more than a thousand times smaller than that of **GJK**. SphereSearch is also almost never slower than **GJK**. This let us argue that SphereSearch might be a good candidate to replace **GJK** in several applications.

Our SphereSearch algorithm also has the advantage, over **GJK**, to be more easily amenable to a fast and exact implementation. Indeed, the only operation that is required is the computation of the signs of the determinant of 3 by 3 matrices, a predicate for which several very efficient exact implementations exist [16].

8 Benchmarks

We have compared our implementations of SphereSearch (or **Sphere**, for short), **GJK** and other algorithms mentioned earlier, over a few kinds of randomly generated data sets: random pairs of tetrahedra, oriented boxes and polytopes, and frustum culling of random spheres and axis-aligned boxes.

In the figures below, **Generic** is an implementation of the technique described in §3. **Sphere** is an implementation of our SphereSearch algorithm.

We have carefully optimized all our tested implementation, but have refrained from using SIMD instructions, which would however clearly help in optimizing further. All the implementations use 32-bits floating-point arithmetic.

The benchmarks are run on a late 2007 MacBookPro laptop with an Intel Core 2 Duo CPU clocking at 2.6 Ghz, with DDR2 RAM clocking at 667 MHz. The software is compiled using Apple's C++ compiler from XCode 6.3. The OS is MacOSX 10.10.3.

The C++ code and Python scripts that were used to generate all the figures are available as companion files to this paper.

8.1 Random tetrahedra

In this test, N tetrahedra are generated as the convex hull of four uniformly random point on the unit sphere. Each tetrahedron is translated along the x axis by a value of $x \in [0, \sigma]$ where σ represents the "spread" of the set of tetrahedra. Then all $\binom{N}{2}$ pairs of tetrahedra are tested for intersection using different techniques. The collision density is the fraction of intersecting pairs. A fixed σ implies a fixed average collision density, and the larger the spread is, the lower the collision density. We ensure that all tetrahedra contain the origin when $\sigma = 0$, so as to guarantee a collision density of 100 % in that case. Figure 4 plots the statistics of this test, run against a varying value of σ. Each sample point is the average of 100 runs with $N = 2000$.

Figure 4–top diagram This diagram shows the number of pairs of tetrahedra tested per second against the collision density, for a variety of algorithms. The general trend of the graphs shows, as expected, that the performance of all the different techniques lowers as the collision density increases.

The **Generic** and **SAT** algorithms both test a predetermined set of separating planes in a predetermined order. This explains their similar performance in the low-density regime, where one or two plane tests are sufficient on average. For pairs of tetrahedra, the **SAT** algorithm performs more arithmetic operations and thus is predictably slower that **Generic** in the high-density regime.

SphereSearch and **GJK** have a very similar behavior but SphereSearch is consistently faster when density is in the range $[5\,\%, 90\,\%]$. Note that the minimum of the graphs for **Sphere** and **GJK** is not at 100 % density, but between 80 % and 90 %. This is the density that maximizes the number of pairs of almost-tangent tetrahedra. These pairs require more work from the algorithm to distinguish between intersecting of non-intersecting tetrahedra (because the origin is close to the boundary of $A \ominus B$). In contrast, and contrary to **Generic** and **SAT**, frank intersections at density 100 % are easier to detect for **GJK** and **Sphere**.

Figure 4–middle diagram This diagram shows the average number of plane tests (or interval overlap tests for **SAT**) per pair of tetrahedra (solid lines with shaded standard deviation) as well as the overall maximal number of plane tests reached during the benchmark operation (dashed lines, excluding the pairs for which the intersection detection failed by reaching Θ iterations).

The maximal possible number of tests for **SAT** (44) and **Generic** (32) is always achieved at any collision density since all the tests are required to confirm that two tetrahedra touch each other. Our **Sphere** implementation never performed more than 10 plane tests in this benchmark. At the lowest density, **Generic** and **SAT** require a bit less than two plane tests on average while **GJK** and **Sphere** require just one, since the heuristic choses a initial plane which is separating when tetrahedra are far away from each other. At density 100 %, **SAT** computes 44 interval overlap tests; **Generic** performs a bit less that 32 plane tests since only pairs of edges that are silhouette of each other incur a plane test. (The silhouette condition is tested for the 36 pairs of edges, but this is much faster that the plane test.) **GJK** and **Sphere** perform 3 plane tests only, on average (see the bottom diagram), which are sufficient to conclude that the tetrahedra touch each other.

Figure 4–bottom diagram For comparing **GJK** and **Sphere**, the most important diagram is the bottom one. It shows the same average number of plane tests as in the middle diagram and also shows the *rate of failure* of each technique. This rate is the probability that the testing of a pair will reach the maximum number of iterations allowed, Θ. This limit, Θ, is required because limited floating-point precision may give wrong results in configuration that are close to degenerate. This is the case when the two tetrahedra barely touch each other, and explains the peak failure rate at density $\approx 72\,\%$ for **GJK**. In SphereSearch, the construction of a new test direction does not depend much on the actual geometry of the problem and our **Sphere** implementation enjoys a much lower failure rate. For example, at density 72 %, **GJK** failed 24779 times while testing 1.999×10^8 pairs of tetrahedra while **Sphere** failed 15 times. The **GJK** technique needs to find a closest point on a simplex to the origin. This is numerically more sensitive and our implementation of **GJK** can reach a failure rate of more that one per ten thousand pairs.

8.2 Random oriented boxes

In this test, N oriented boxes are generated randomly. The three edge half-lengths are uniformly random in $[0, 1]$, the center of each box is uniformly random in a zero-centered cube of side length σ and the box is randomly oriented. Then all $\binom{N}{2}$ pairs of boxes are tested for intersection using different techniques. Figure 5 plots the statistics of this test, run against a varying value of σ. Each sample point is the average of 100 runs with $N = 2000$. The **SAT** implementation has been taken directly from Gottschalk's PhD manuscript [10].

Figure 5–top diagram As expected, the specialized **SAT** implementation is faster than **GJK** or **Sphere** when the collision density becomes non-negligible. Indeed, while the **GJK** and

Sphere implementations also take advantage of the symmetrical nature of the boxes to accelerate the computation of $D(n)$, they can not exploit the algebraic simplifications stemming from the specific choice of test normal vectors that **SAT** uses. The largest speed ratio of **SAT** to **Sphere** is 2.21. The largest speed ratio of **SAT** to **GJK** is 2.58.

In this benchmark, **Sphere** is faster than **GJK** in the density range $[0.6\,\%, 100\,\%]$ by as much as 24 % at density 45 %.

Figure 5–bottom diagram As for tetrahedra, the most striking observation is how our **Sphere** technique is able to use fewer test planes that **GJK**, although these numbers are already very close to optimal. Here again, while relatively small, the failure rate of **GJK** is still about a thousand times larger than that of **Sphere**.

8.3 Random polytopes with 16 vertices

We now move to somewhat larger convex polytopes generated as the convex hull of 16 random points on the unit sphere and translated by a random amount in $[0, \sigma]$ along the x axis. Figure 6 plots the statistics of this test, run against a varying value of σ. Each sample point is the average of 100 runs with $N = 1600$ for **GJK** and **Sphere** and $N = 300$ for **Generic**.

Each plane test in the first phase of Algorithm 1 (page 11, lines 2-5) requires to loop over the vertices of a single polytope, instead of looping over the vertices of both in order to compute $D(n)$ in **GJK** and **Sphere**. This explains why **Generic** is faster at very low collision density. The performance of **Generic** falls dramatically at higher density because of its quadratic time complexity.

Comparing **Sphere** and **GJK**, we see a trend very similar to the oriented boxes benchmark. **Sphere** is again faster and more robust than **GJK** in the density range $[1\,\%, 100\,\%]$ and up to 12 % faster at density 65 %. In the next benchmark, we fix the collision density at 50 % and vary the number of vertices of the polytopes.

8.4 Varying the number of vertices at 50 % density

In this benchmark, we generate random the random polytopes in the same way as in §8.3, but set the spread σ so that the collision density is always approximately 50 %. We then vary the number of vertices of the polytopes and test the same three techniques. Figure 7 plots the statistics of this benchmark.

Unsurprisingly, as show in the top diagram, the **Generic** technique exhibits a inverse quadratic dependency on the number v of vertices. The **Sphere** and **GJK** techniques shows a performance only inversely proportional to v, and a bit better than that for $v \leq 100$, perhaps thanks to cache memory.

Regarding **GJK** and **Sphere**, when the number of vertices increases, the cost of a single plane test becomes dominant compared to the cost of updating the respective data-structure maintained by these two techniques from one iteration to the next. Therefore, the time to perform one test increasingly depends only on the average number of plane tests, which is lower, at this collision density, for our **Sphere** technique. This explains the constant ratio, of about 1.12, between **Sphere** and **GJK** when the number of vertices is larger than 20. (The corresponding parallel curves are more easily seen in the bottom diagram.)

When the number of vertices is large, it might become interesting to add a hierarchy on top of the vertices of a polytope in order to accelerate the maximization of a linear function over the polytope. We have experimented with such an acceleration scheme and show the result in the

bottom diagram of Figure 7. As expected, the scheme is effective for both **Sphere** and **GJK** and more effective with an increasing number of vertices.

8.5 Frustum culling: size matters

We haven't yet looked at a case of convex objects that are relatively simple but show a strong size discrepancy. To analyse this case, we look at frustum culling. The frustum is a six-sided truncated pyramid. Against a frustum, we cull either axis-aligned boxes (Figure 8) or spheres (Figure 9), since these shapes are typically used as bounding shapes of more complex geometric data.

All frustums are generated with a constant horizontal field-of-view of 80° and a 16 : 9 aspect ratio. The near plane is 0.1 units away from the frustum apex and the far plane 100 units away. We compute the center C and radius ρ of the largest inscribed sphere of a frustum and translate the frustum so that the center C coincide with the world origin. Each frustum is then randomly rotated.

The radii of the spheres and the edge-lengths of the boxes are uniformly random in $[0, 1]$. Their centers are uniformly random in the ball of center C and radius $\sigma\rho$ where the spread parameter σ is never smaller than 1. We generate N frustums and N boxes or spheres and test all N^2 pairs for intersection. N is set to 1000 and the statistics are averaged over 100 runs (5×10^7 tested pairs for each sample point). We decrease the collision density by increasing the parameter σ.

When we look at the top diagrams of Figures 8 and 9, the situation is different from the previous benchmarks. For frustum culling small objects (w.r.t. the frustum), the **Sphere** technique is slower than **GJK** on almost the whole density range. However, the middle diagrams indicate that **GJK** still fails far more often that **Sphere**.

We need to make two observations in order to understand why **Sphere** is slower in that case. Let A be the frustum and B a box of a sphere.

1. Since B is much smaller that A, $A \ominus B$ is approximately equal to A, that is, $A \ominus B$ is very close to the shape of a frustum, which is a quite simple geometric shape.

2. The **GJK** algorithm builds local approximations of $A \ominus B$ closer and closer to the origin [9].

Since $A \ominus B$ is a simple shape, the first triangle that **GJK** builds in $A \ominus B$ is highly likely to approximate the face of $A \ominus B$ closest to the origin very well, say within Hausdorff distance 1, the size of B. This approximation is often largely enough for the subsequent plane test to succeed or for the generated tetrahedron to include the origin, letting the algorithm decide that A and B intersect.

This behavior explains the smaller standard deviation of **GJK** (bottom diagrams) and why it is faster: because it can more often avoid the distance minimization step over a tetrahedron.

In contrast, SphereSearch works on the unit sphere of directions, and generates a new test direction as an approximate center n of the current spherical polygon S'. Now consider the plane with normal vector n (toward its positive side) and tangent to $A \ominus B$, with $A \ominus B$ on its negative side. Any small variation of n makes this tangent plane rotate around some point of $A \ominus B$. During this rotation, *at the other end of* $A \ominus B$, far from the rotation center, the local distance of the plane to $A \ominus B$ varies widely, with respect to the size of B. This will often put the origin in the negative side of the plane, preventing **Sphere** to conclude quickly that A and B are disjoint (equivalently $0 \notin A \ominus B$); **Sphere** will require more iterations to align its test plane with a face of $A \ominus B$.

In other words, in the case of frustum culling, the separating set $\mathcal{S}^-(A \ominus B)$ is small and the function D has a large gradient. This is especially pronounced when A and B are very close to

a tangent configuration. In this context, **Sphere** needs more iterations to find $\mathcal{S}^-(A \ominus B)$; its dichotomic search works better for smoother D.

8.5.1 A hybrid technique

As evidenced in the bottom diagrams of Figures 8 and 9, despite being the faster contender for frustum culling small objects, **GJK** shows again a rate of failure much larger than that of **Sphere**.

It turns out that we can combine **GJK** and **Sphere** into a technique, called **Hybrid** in the two figures, that is just as fast as **GJK** and fails just as seldom as **Sphere**. To do so, we start with **GJK** for the first four iterations. After the first tetrahedron τ has been built, we build the spherical polygon

$$S = \bigcap_{p, p \text{ is a vertex of } \tau} p^{\downarrow} \tag{42}$$

and switch to **Sphere** iterations. The statistics for our **Hybrid** implementation are shown in Figures 8 and 9.

$\diamond$

This hybrid technique is only useful when two objects of very different size are tested for intersection. When we enlarge the spheres or boxes to roughly the size of the frustum,[4] we obtain again the same behavior as, eg, in the oriented-boxes benchmark in §8.2, where **Sphere** is faster than **GJK** in almost the whole collision density range. In this case as well, **Hybrid** performs just like **GJK** and is therefore slower that **Sphere**. Remarkably, **Hybrid** fails (in the sense given in §7.3) even less often that **Sphere**, typically once or twice per benchmark, which involves about 5×10^7 tested pairs. It would be interesting to understand this, and we set this task as future work.

9 Concluding remarks

By looking at the intersection detection problem with Gauss maps, we obtained a unifying view of several techniques and we were able to design a fast technique for view-pyramid culling of axis-aligned boxes and triangles with applications in ray casting. A more abstract look at function D led to our new SphereSearch algorithm with is in general a bit faster and more robust that the ubiquitous **GJK** technique. We expect the new algorithm to be useful in many applications.

References

[1] Ulf Assarsson and Tomas Möller. Optimized view frustum algorithms for bounding boxes. *J. Graphics Tools*, 5(1), 2000.

[2] Luis Barba and Stefan Langerman. Optimal detection of intersections between convex polyhedra. In *Proceedings of the Annual Symposium on Discrete Algorithms (SODA)*. ACM-SIAM, 2015.

[3] Eric Berberich, Efi Fogel, Dan Halperin, Michael Kerber, and Ophir Setter. Arrangements on parametric surfaces II: Concretizations and applications. *Math. in Comp. Sci.*, 4(1), 2010.

[4] We have benchmarked this case but do not include the diagrams, which are similar to earlier diagrams on boxes or small polytopes.

[4] Solomon Boulos, Ingo Wald, and Peter Shirley. Geometric and arithmetic culling methods for entire ray packets. Technical Report UUCS-06-10, University of Utah, 2006.

[5] Kenneth L. Clarkson, David Eppstein, Gary L. Miller, Carl Sturtivant, and Shang-Tua Teng. Approximating center points with iterated radon points. *International Journal of Computational Geometry & Applications*, 6(3), 1996.

[6] David Eberly. Intersection of ellipsoids. Technical report, Geometric Tools, LLC, 2008.

[7] Christer Ericson. *Real-Time Collision Detection*. CRC Press, 2004.

[8] Elmer G. Gilbert and Chek-Peng Foo. Computing the distance between general convex objects in three-dimensional space. *IEEE Journal of Robotics And Automation*, 6(1):53–61, February 1990.

[9] Elmer G. Gilbert, Daniel W. Johnson, and Sathiya Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics And Automation*, 4(2), April 1988.

[10] Stefan Gottschalk. *Collision Queries using Oriented Bounded Boxes*. PhD thesis, UNC Chapel Hill, 2000.

[11] Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. OBBTree: A hierarchical structure for rapid interference detection. In *Proc. SIGGRAPH*, Annual Conference Series. ACM, ACM, 1996.

[12] Ned Greene. Detecting intersection of a rectangular solid and a convex polyhedron. In Paul S. Heckbert, editor, *Graphics Gems IV*, chapter I.7, pages 74–82. Academic Press, 1994.

[13] Leonidas J. Guibas, An Nguyen, and Li Zhang. Zonotopes as bounding volumes. In *Proceedings of the Annual Symposium on Discrete Algorithms (SODA)*. ACM-SIAM, 2003.

[14] Eric A. Haines and John R. Wallace. Shaft culling for efficient ray-cast radiosity. In *Proceedings of the Second Workshop on Rendering*. Eurographics, 1991.

[15] P. Jiménez, F. Thomas, and C. Torras. 3D collision detection: A survey. *Computers & Graphics*, 21(2):269–285, 2001.

[16] Sylvain Pion and Andreas Fabri. A generic lazy evaluation scheme for exact geometric computations. *Science of Computer Programming*, 76(4):307–323, April 2011.

[17] Alexander Reshetov. Faster ray packets-triangle intersection through vertex culling. In *Proceedings of the Symposium on Interactive Ray Tracing*. IEEE, 2007.

[18] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1), 2007.

[19] Wenping Wang, Jiaye Wang, and Myung-Soo Kim. An algebraic condition for the separation of two ellipsoids. *Computer Aided Geometric Design*, 18(6):531–539, 2001.

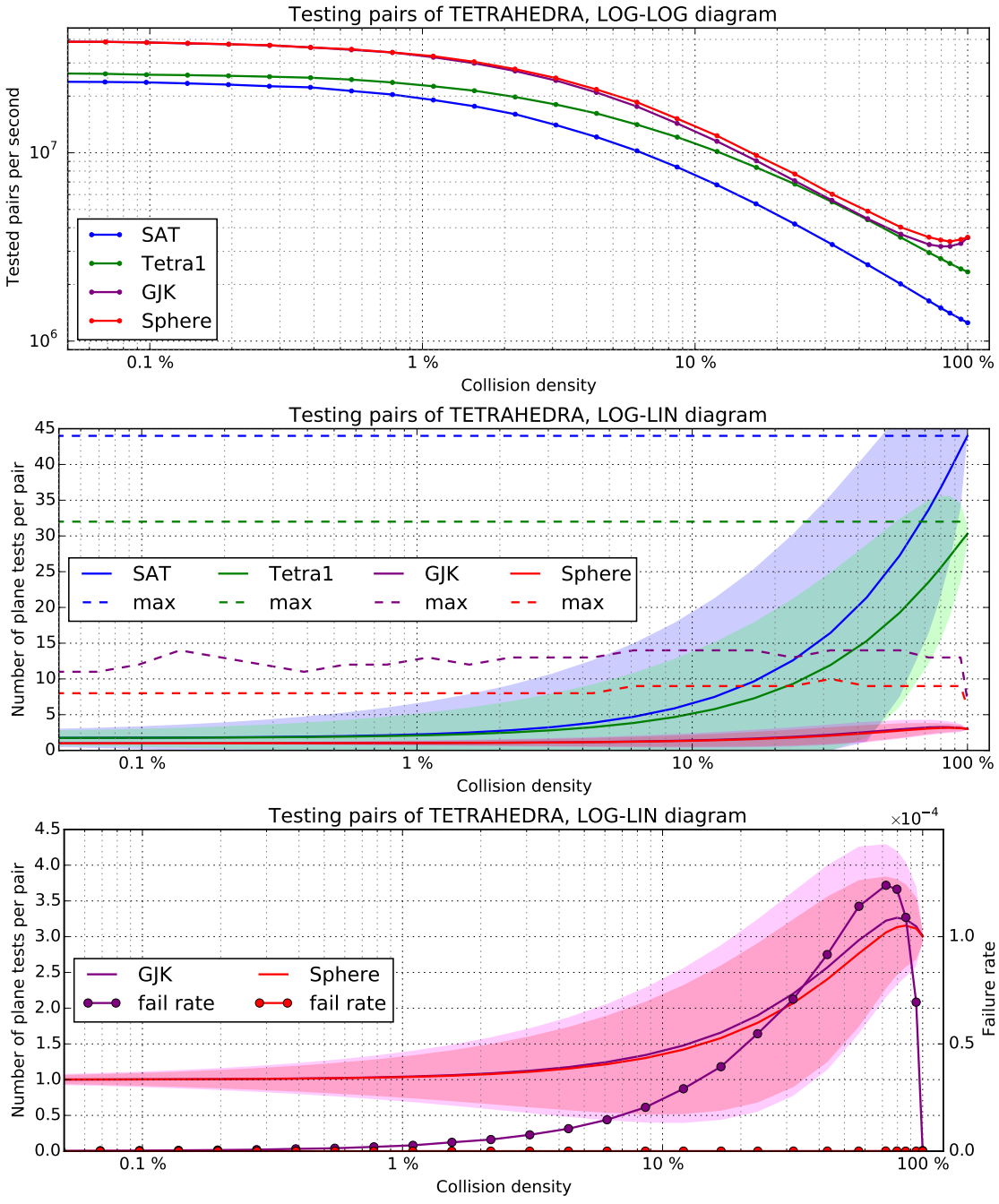


Figure 4: Statistics for the 'random tetrahedra' test.

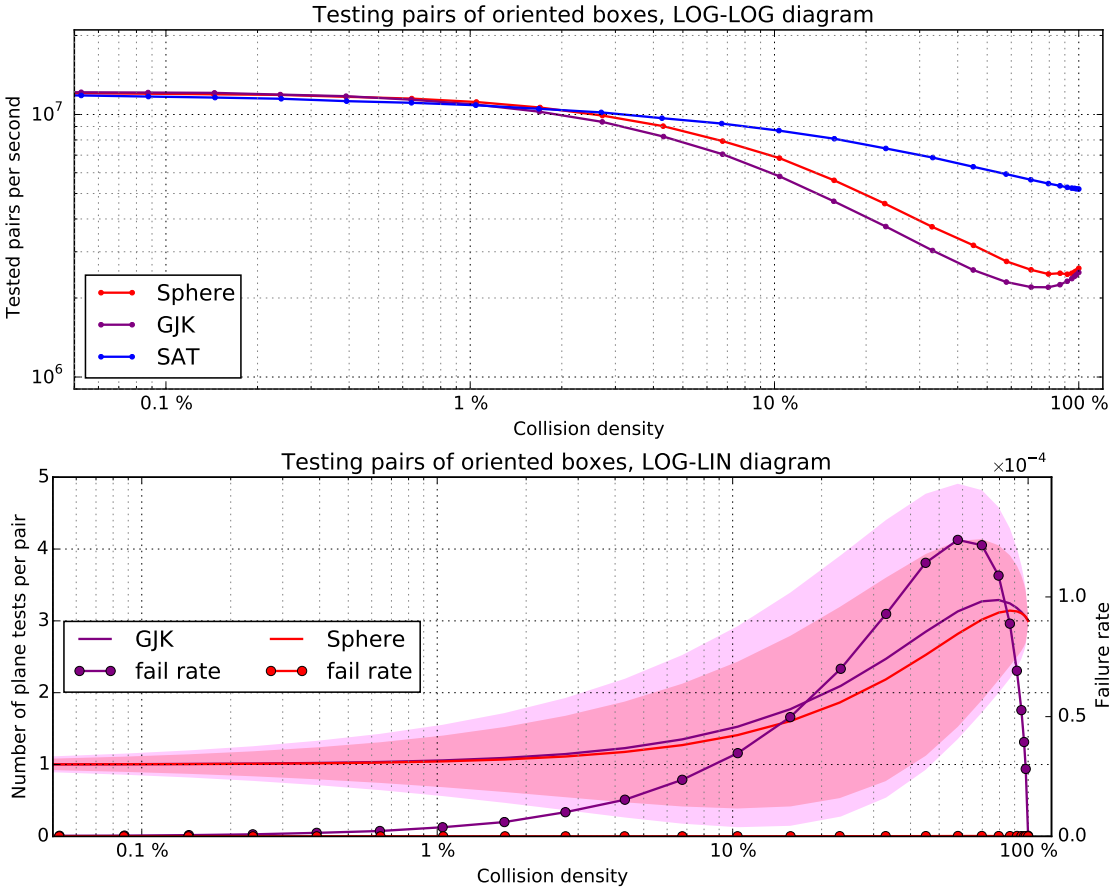


Figure 5: Statistics for the 'random oriented boxes' test.

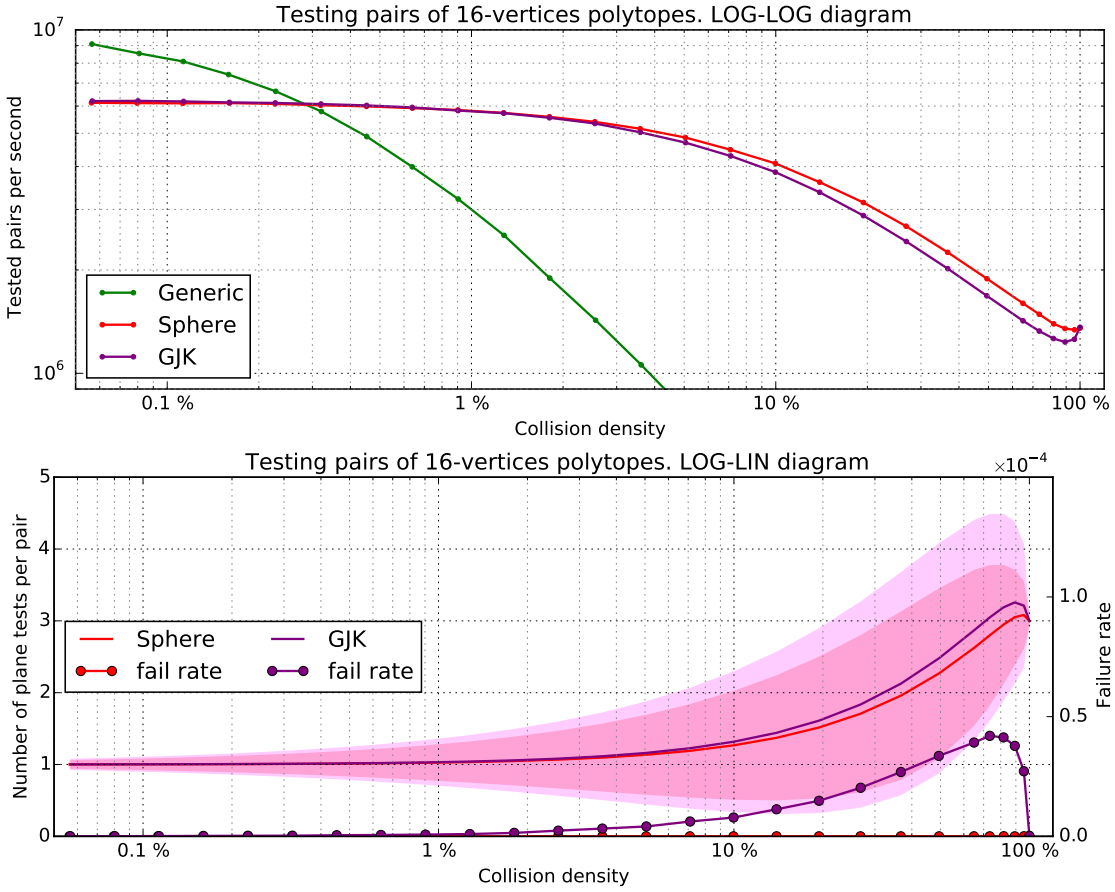


Figure 6: Statistics for the 'random polytopes with 16 vertices' test.

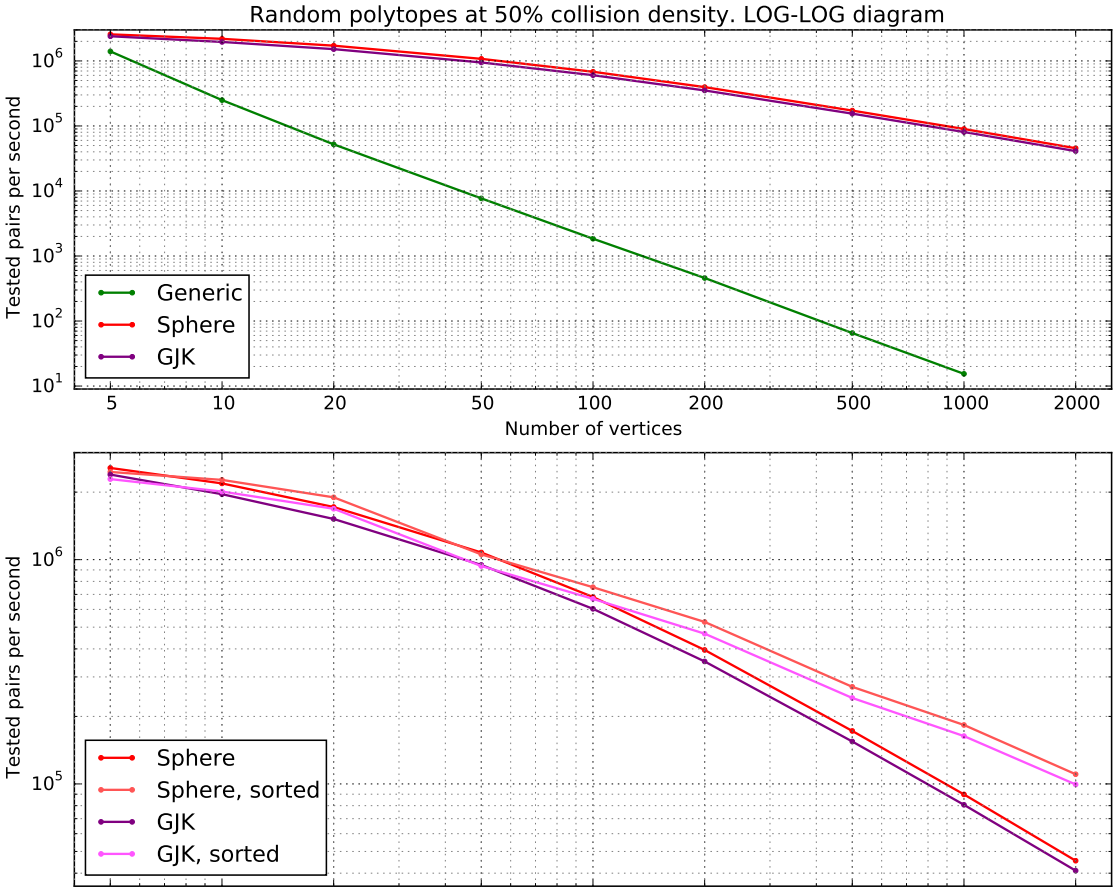


Figure 7: Statistics for the 'random polytopes at 50 % collision density' test.

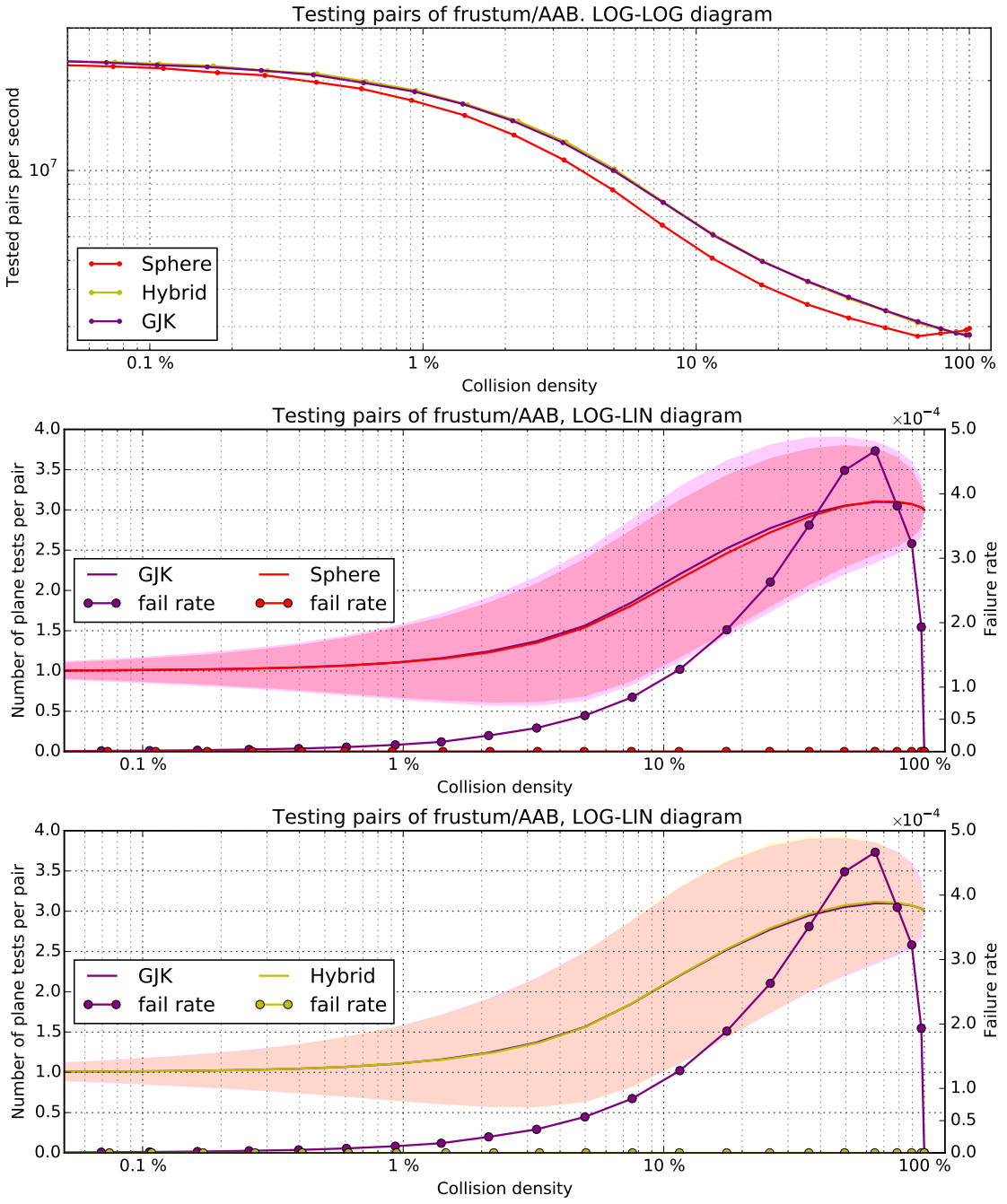


Figure 8: Statistics for frustum culling of axis-aligned boxes.

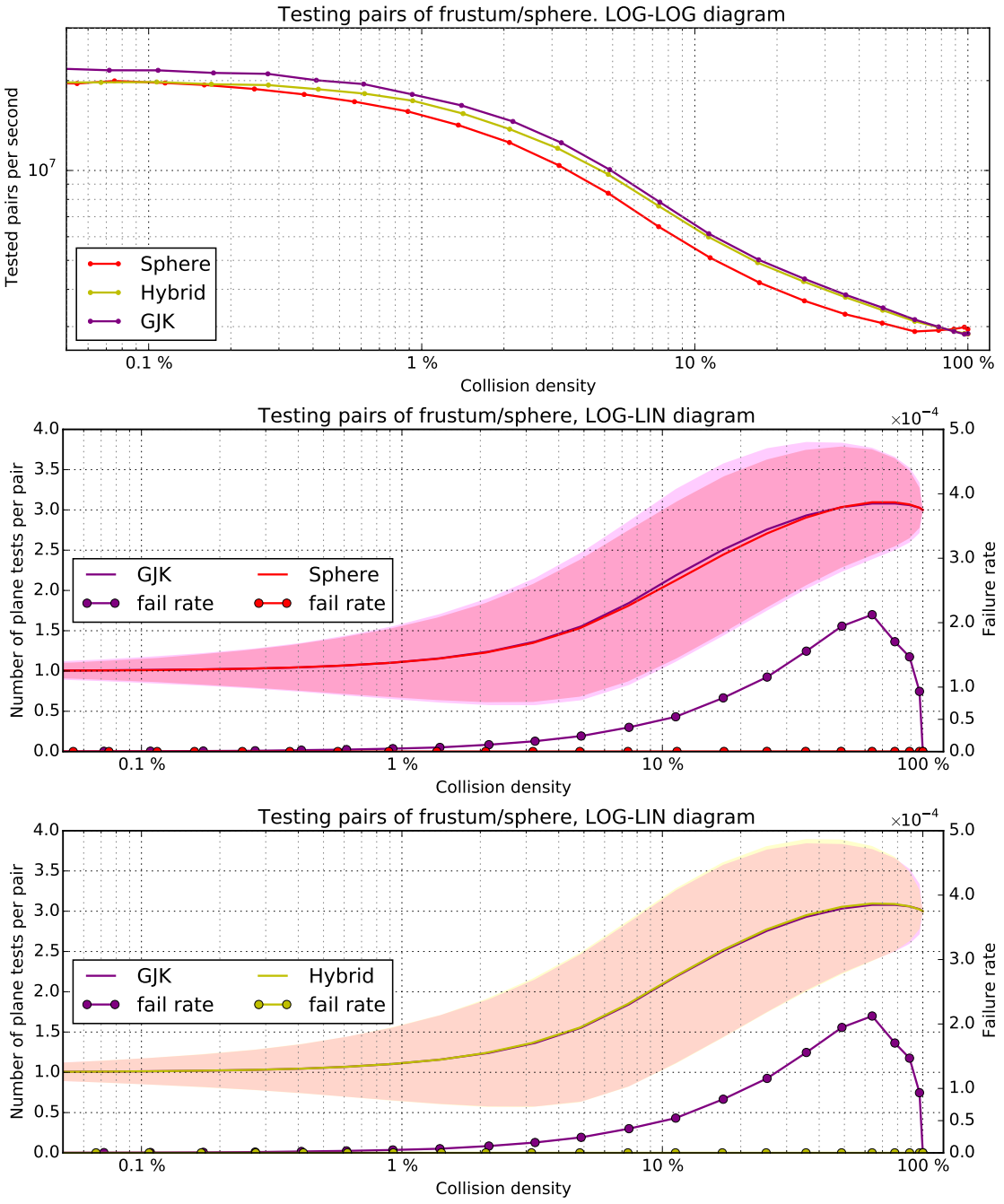
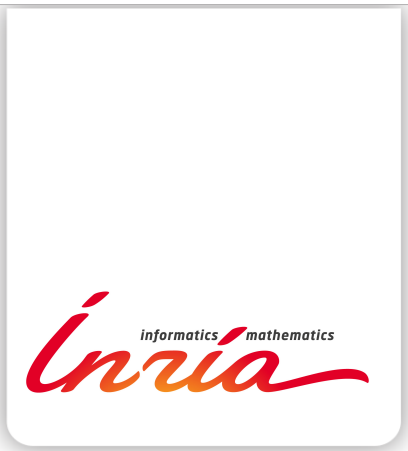


Figure 9: Statistics for frustum culling of spheres.



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399