Detecting the intersection of two convex shapes by searching on the 2-sphere

Samuel Hornus

Detecting the intersection of two convex shapes by searching on the 2-sphere

Samuel Hornus *

Inria, Villers-lès-Nancy, F-546000, France
Université de Lorraine, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54506, France
CNRS, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54506, France

Abstract

We take a look at the problem of deciding whether two convex shapes intersect or not. We do so through the well known lens of Minkowski sums and with a bias towards applications in computer graphics and robotics. We describe a new technique that works explicitly on the unit sphere, interpreted as the sphere of directions. In extensive benchmarks against various well-known techniques, ours is found to be slightly more efficient, much more robust and comparatively easy to implement. In particular, our technique is compared favourably to the ubiquitous algorithm of Gilbert, Johnson and Keerthi (GJK), and its decision variant by Gilbert and Foo. We provide an in-depth geometrical understanding of the differences between GJK and our technique and conclude that our technique is probably a good drop-in replacement when one is not interested in the actual distance between two non-intersecting shapes.

Keywords: intersection detection, GJK, numerical robustness, collision detection, Minkowski sum, Gauss map

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 [9], computer animation [12] 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 [8] and the survey of Jiménez *et al.* [17].

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 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 [1] and the other references within). This essentially closes the (theoretical) problem for the case of convex polyhedra. It is not clear if their technique is amenable to an efficient implementation since the data-structures used are not simple and it requires knowledge of the connectivity between the polyhedron neighboring facets.

In this paper however, we only consider techniques that do not use excessive or complex pre-processing [1] and are asymptotically slower, but very fast in practice.

Of particular interest is the beautiful algorithm developed by Gilbert, Johnson and Keerthi (GJK) for computing the distance $d(A, B)$ between two convex polyhedra A and B [10]. It only needs access to the vertices of the polyhedra and typically uses just a few iterations over them to compute the distance $d(A, B)$. It was later generalized and adapted to the decision version of the problem by Gilbert and Foo [9] to handle a broader class of convex objects. We describe these algorithms in Section 3.

In this paper we view the decision problem as that of finding an *oriented* plane that sets A on its negative and B on its positive side. Writing $\mathcal{S}^-$ for the set of normals to the planes achieving separation, we design, in Section 4, an algorithm to find a direction in $\mathcal{S}^-$ or decide that $\mathcal{S}^-$ is empty (when A and B do touch each other). Our algorithm iteratively prunes parts of the unit sphere $\mathcal{S}$ so that the remaining part, a convex spherical polygon, provides

* samuel.hornus@inria.fr

[1] Except, briefly, in Section 6.4.

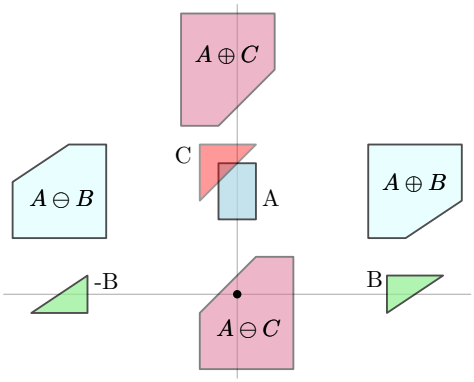

Figure 1: Three convex shapes A, B and C lie in the Euclidean plane whose origin is marked. Some Minkowski sums and differences are drawn. Note how the origin lies in the Minkowski difference of A and C, proving that these two shapes intersect.

an increasingly tight superset of $\mathcal{S}^-$. We call it *Decision Sphere Search*, or DSS for short.

Section 5 gives a theoretical analysis of DSS and an extensive comparison of DSS with GJK. In particular, we show that DSS optimally aggregates the information gathered about the Minkowski difference $A \ominus B$ during the successive iterations.

In Section 6, we benchmark our implementations of a "naive", quadratic algorithm, of DSS and of GJK. Each benchmark considers a specific type of objects and measures the performance of the algorithm with respect to the "collision density," *ie* the ratio of the number of tested pairs of convex objects that actually intersect to the total number of tested pairs. Our DSS technique appears to be faster than GJK in general, numerically more robust and easier to implement. 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 than the other. In that case, we show that a hybrid technique combining DSS and GJK gives the overall best results and we explain why.

2. Preliminaries

It is simpler to work with an alternative view of the geometry of the intersection detection problem, thereby reducing it to deciding whether a closed convex set contains the origin or not. The rest of the present section describes this well known alternative view.

The *Minkowski sum and Minkowski difference* of two subsets A and B of $\mathbb{R}^3$ are (see Figure 1):

$$A \oplus B = \{a + b \mid a \in A, b \in B\} \text{ and} \quad (1)$$
$$A \ominus B = \{a - b \mid a \in A, b \in B\}. \quad (2)$$

Using this definition, A and B have non-empty intersection if and only if $A \ominus B$ contains the origin $\mathbf{o}$:

$$A \cap B \neq \emptyset \quad \Leftrightarrow \quad \mathbf{o} \in A \ominus B. \quad (3)$$

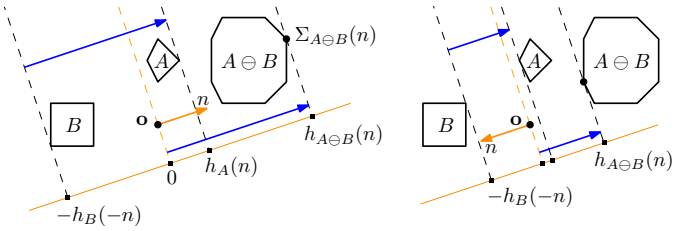

Figure 2: Two illustrations of equation (8) with opposite normal vectors n. In each case, the blue vectors are identical. On the left, $\max_{x \in A \ominus B} n \cdot x = h_{A\ominus B}(n)$ is positive. On the right, it is negative, which certifies that $A \ominus B$ does not contain the origin.

We write $d(A, B)$ for the distance between the two subsets A and B:

$$d(A, B) = \min_{a \in A, b \in B} |a - b|. \quad (4)$$

When A and B are closed convex sets, their Minkowski difference is a closed convex set as well. In equation (3), we have thus reduced our question on the existence of an intersection between A and B to a question regarding a single convex object $P = A \ominus B$ and the origin. In the rest of the paper we most often consider this single closed convex set P and its relation with the origin $\mathbf{o}$, instead of considering A and B separately.

We now define three functions that play an important role in the algorithms discussed in this paper. For a closed convex subset P of $\mathbb{R}^3$, we define the *support function* h_P that maps a unit vector n of the unit sphere $\mathcal{S}$ to a real number defined as

$$h_P(n) = \max_{p \in P} (n \cdot p). \quad (5)$$

The support function is closely tied to the study of convex shapes and serves as a tool to represent and manipulate them. The usefulness of the support function for shape modeling operations was recognized by Sabin who showed how offset and convolution are easily expressed with it [20]. A more technical overview with an application to the computation of Minkowski sums is given, *eg* by Šír *et al.* [21]. We also define the *extremal function* Σ_P as

$$\Sigma_P(n) = \arg\max_{p \in P} (n \cdot p), \quad (6)$$

ie, $h_P(n) = n \cdot \Sigma_P(n)$. For a given direction n, several points on P might realize the largest dot-product with n. In this case we choose a point arbitrarily. When the convex set P is not bounded, we implicitly restrict the functions h_P and Σ_P to the portion of $\mathcal{S}$ where they are well defined. The *closest-point function* ν maps P to the unique point in P that realizes the distance from P to the origin, *ie* $\nu(P) \in P$ and $|\nu(P)| = d(P, \{\mathbf{o}\})$.

The case $P = A \ominus B$ is of particular importance for us.

Algorithm 1 The GJK algorithm.

```
1:  function GJK(P, V)    ▷ If we test A and B, then P = A ⊖ B
                                                        ▷ |V| ≤ 4
2:      v ← ν(H(V))   ▷ v is the point of H(V) closest to the origin
3:      if v = o then              ▷ d(P, {o}) = 0 since o ∈ H(V) ⊂ P
4:          return 0 (for the distance problem)
5:          or return INTERSECTION (for intersection detection)
6:      p ← Σ_P(−v/|v|)
7:      if v · p = |v|² then
8:          return |v| or DISJOINT        ▷ o ∉ P and d(P, {o}) = |v|
9:      V̂ ← the smallest subset of V such that v ∈ H(V̂)  ▷ |V̂| ≤ 3
10:     return GJK(P, V̂ ∪ {p})
```

In this case, the functions h_P and Σ_P are computed as:

$$h_{A\ominus B}(n) = \max_{a\in A}(n\cdot a) - \min_{b\in B}(n\cdot b) \quad (7)$$

$$= h_A(n) + h_B(-n) \text{ and} \quad (8)$$

$$\Sigma_{A\ominus B}(n) = \Sigma_A(n) - \Sigma_B(-n) \text{ (see Figure 2).} \quad (9)$$

The convex shape $P = A \ominus B$ contains the origin if and only if its support function h_P, takes a non-negative value over the whole unit sphere:

$$A\cap B \neq \emptyset \;\Leftrightarrow\; \mathbf{o}\in P \;\Leftrightarrow\; \forall n\in\mathcal{S}, h_P(n)\geq 0, \text{ or} \quad (10)$$

$$A\cap B = \emptyset \;\Leftrightarrow\; \mathbf{o}\notin P \;\Leftrightarrow\; \exists n\in\mathcal{S}, h_P(n) < 0. \quad (11)$$

3. Related work

Research on practical intersection detection techniques for *static* convex shapes has not been very active in recent years, so we can refer the reader to the survey of Jiménez *et al.* [17] and the excellent book of Ericson [8]. We also refer the reader to our technical report [16], where we propose a review of previous work on intersection detection problem for small convex polyhedra with or without specific symmetries, through the unifying lens of Minkowski sums and the Gauss map, which both play a central role as soon as the relation between two convex objects is sought after; see for example the references [18, 5].

Among the various existing techniques, that of Gilbert, Johnson and Keerthi stands out as a very general and typically very fast method to compute the distance between two convex polyhedra. As such it is an indispensable tool in software that deals with interacting shapes (see Section 5.4). We devolve this section to describing the GJK technique [10] and its cousin, developed by Gilbert and Foo [9] that generalizes it to a more general class of convex shapes and adapt it to the decision version of the problem.

3.1. The GJK algorithm

The technique of Gilbert, Johnson and Keerthi (called GJK hereafter) incrementally evolves a set $\mathcal{V}$ of d vertices of P ($d \in [1, 4]$) whose convex hull $\mathcal{H}(\mathcal{V})$ forms increasingly accurate approximation of $A \ominus B$ closer and closer to the origin [10]. See Algorithm 1. The first call to GJK is passed the parameters $P = A \ominus B$ and $\mathcal{V} = \{x\}$ where x is any point in P. In each iteration, $\mathcal{H}(\mathcal{V})$ serves as a low complexity proxy to the full convex P. The iteration of the GJK algorithm proceeds as follows. The closest point of $\mathcal{H}(\mathcal{V})$ to the origin is computed and stored in variable v. If $v = \mathbf{o}$ then $\mathbf{o} \in P$ because $\mathcal{H}(\mathcal{V}) \subset P$ and this proves that P contains the origin (or that A and B do intersect); the iteration terminates. Otherwise the support point on P along the direction of $-v$ is computed and stored in variable p. The authors show (and it is easy to see) that if $v \cdot p = |v|^2$ then we have found $\nu(P) = v$ so the iteration terminates ($d(A, B) = |\nu(P)|$). Finally, a new simplex $\hat{\mathcal{V}} \cup \{p\}$ is formed which is a better approximation of P closer to the origin and the iteration continues. The authors provide a proof that the algorithm does indeed terminate after a finite number of iterations when P is a polyhedron.

For the decision version of GJK, we can return DISJOINT as soon as we find a plane separating P from $\mathbf{o}$. To do so, we replace lines 7 and 8 by

```
if v · p > 0 then
    return DISJOINT
```

The main difficulty with GJK is the computation of $\nu(\mathcal{H}(\mathcal{V}))$ when $\mathcal{V}$ is a tetrahedra, starting at the beginning of the fourth iteration. First, this computation is time consuming and not easy to implement correctly. Second, the chain of floating point computations starting at the input point coordinates are long, which increases the numerical inaccuracies. The later are exacerbated by the geometry of the tetrahedra itself which tends to be close to degenerate, being almost as flat as a triangle. This behavior forces an implementation to monitor the number of iterations and decide that P contains the origin when that number exceeds a fixed threshold. (A large number of iterations is usually due to a configuration where P and the origin are difficult to separate.) Another minor defect of GJK is its inability to take advantage of all the information gathered along the iterations. In particular, after the fourth iteration, each iteration loses one of the vertices of P computed in line 6 of Algorithm 1. This loss is on purpose, so that $\mathcal{H}(\mathcal{V})$ never get more complex than a tetrahedron.

In contrast, our new technique DSS keeps all the relevant information of previous iterations compactly in the form of a convex spherical polygon. Long chains of floating point computations are not required, thereby greatly decreasing numerical inaccuracies. This makes the technique much less prone to reaching the maximum number of iterations and taking, then, an arbitrary decision, as demonstrated in our benchmarks. In addition our DSS technique has a more stable way to pick a new candidate test direction v, which leads to a smaller number of iterations than GJK on average. Finally, DSS is also easier to implement since its most complex subroutine is the clipping of a 2D convex polygon against a half-plane.

The GJK technique was then generalized by Gilbert

and Foo [9] to arbitrary convex shapes X as long as the corresponding extremal function Σ_X can be computed on them. In the same reference, the decision version of the algorithm is also presented, as described above.

More recent work has focused on computing the penetration depth of intersecting objects [18, 22, 23] or bringing time into the problem: maintaining the intersection status as the objects move or deform and doing so over complex hierarchies of shapes, see *eg* [14] and the huge body of work at the Gamma research group at UNC. We discuss some of these aspects in Section 5.4.

3.2. Projection onto Convex Sets

While DSS and GJK are closely related, there is another, much more general family of techniques for computing a point in the intersection of two or more convex shapes by alternate projection of a point on the shapes until convergence. These *Projection onto Convex Sets* (POCS) techniques are typically used in the case of many convex shapes in higher dimensional spaces. Their basic operation is the projection operator, while DSS and GJK use the extremal function Σ. Using POCS for deciding if two three-dimensional convex shapes have an intersection point seems less efficient than using GJK or DSS. The interested reader may consult the survey by Bauschke and Borwin [2].

4. Our Decision Sphere Search algorithm

Our algorithm is similar in spirit to the GJK/GF algorithm: A sequence of candidate directions are generated in which the convex P and the origin $\mathbf{o}$ are tested for separation using equations (11) and (8) until a final decision can be taken. Our algorithm differs from theirs in the way a new test direction is generated (they search a point closest to the origin on a simplex inside P, whereas 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 [9]). Note that equation (8), as well as our technique, described below, applies to any pair of closed convex objects, not necessarily polyhedra.

Let us define the *separating set* $\mathcal{S}^-(P)$ of convex P as

$$\mathcal{S}^-(P) = \{n \in \mathcal{S} \mid h_P(n) < 0\}, \quad \text{(see Figure 3)}. \tag{12}$$

It is the set of normals of the oriented planes tangent to P with P on their non-positive side and $\mathbf{o}$ on their positive side. Then

$$A \cap B \neq \emptyset \quad \Leftrightarrow \quad \mathcal{S}^-(A \ominus B) = \emptyset, \text{ or} \tag{13}$$

$$\mathbf{o} \in P \quad \Leftrightarrow \quad \mathcal{S}^-(P) = \emptyset. \tag{14}$$

Our algorithm follows this idea and searches over the unit sphere for a direction n in which $h_{A\ominus B}(n)$ is negative, or decides that $h_{A\ominus B}$ is everywhere non-negative. When v is a vector in $\mathbb{R}^3$, let $v^\uparrow$ denote the north-hemisphere of $\mathcal{S}$ whose north pole is in the direction of v: $v^\uparrow = \{w \in \mathcal{S} \mid v \cdot w \geq 0\}$ and $v^\downarrow$ denote its complement: $v^\downarrow = \mathcal{S} \setminus v^\uparrow$. (As an exception, $\mathbf{o}^\uparrow$ is equal to $\mathcal{S}$; it is not a hemisphere.) To design our search procedure, we use the following

Algorithm 2 Our DecisionSphereSearch algorithm.

```
1: function DecisionSphereSearch(P, S)
2:     n ← a center point of S                    ▷ It holds that n ∈ Ŝ
3:     p ← Σ_P(n)                                 ▷ h_P(n) = n · p
4:     if h_P(n) < 0 then return Disjoint         ▷ o ∉ P
5:     S' ← S ∩ p↓                                ▷ Now, n ∉ Ŝ' since n · p ≥ 0
6:     if Ŝ' is empty then return Intersection    ▷ o ∈ P
7:     return DecisionSphereSearch(P, S')
```

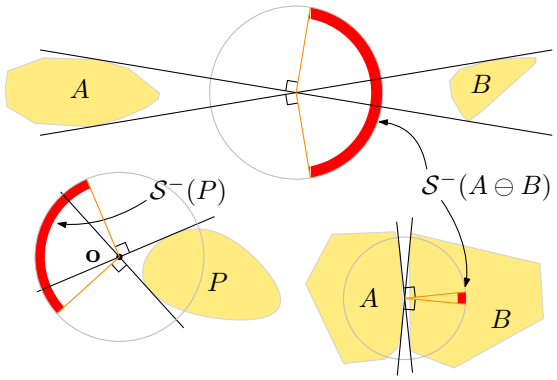

Figure 3: In 2D, the separating set of $A \ominus B$ or P, drawn red, is a circular arc on the unit circle.

Lemma 1. *Let p be a point in $P \setminus \{\mathbf{o}\}$. Then $h_P(\frac{p}{|p|}) > 0$ and, for all $n \in p^\uparrow$, $h_P(n) \geq 0$.*

Proof. Clearly, $p \cdot p$ is positive, which proves that $h_P(\frac{p}{|p|})$ is positive as well. If n is a vector in $p^\uparrow$, then $n \cdot p \geq 0$. Therefore $h_P(n) \geq 0$. □

The lemma above states that the support function h_P takes a non-negative value on any direction $n \in p^\uparrow$. We use this property to prune parts of the unit sphere in which we can not find a direction n making h_P negative. Our search procedure (Algorithm 2) takes as parameters a *search polygon* S: a convex spherical polygon whose interior, $\mathring{S}$, is guaranteed to contain $\mathcal{S}^-(P)$, and a direction $n \in \mathring{S}$:

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 $p \leftarrow a - b$ and $n \leftarrow \frac{-p}{|p|}$ and call DecisionSphereSearch($A \ominus B$, n, $n^\uparrow$).

Initially, the spherical polygon S is set to a hemisphere, and the first test direction is the center n (or pole) of that hemisphere. If $h_P(n) \geq 0$, then direction n fails to separate B from A and a new direction must be tested. The extremal point $p \in P$ ($p = a - b$, $a \in A$ and $b \in B$) that was computed in line 3 is put to use to prune a part of the search polygon: since we know, by Lemma 1, that

h_P takes a non-negative value over $p^\uparrow$ the search polygon can be reduced to $S \cap p^\downarrow$.

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, the hemisphere $p^\downarrow$, see 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* [6]. 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 DSS algorithm shares several interesting properties with that of Gilbert and Foo [9]:

- The only operation needed on the convex object is the computation of the extremal function Σ 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 DSS becomes an exact frustum culling algorithm for spheres. Algorithms to compute Σ_X for various classes of shapes X, including ellipsoids, are given in [9, 7]. Zonotopes (which include oriented bounding boxes) are tight-fitting bounding volumes for which the extremal function is easy to compute; see [15, 16].

- DSS 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.[2] This limits the extrema to finite points only without any restriction since extrema at infinity always lead to a positive infinite value of support function h.

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 stages of the algorithm. This lowers the average number of directions to be tested. See Section 5.

- Importantly, DSS has a much lower failure rate than GJK, where a failure means entering in an infinite loop because of numerical inaccuracy. See Section 5.3.

[2] The support, or domain, of the Gauss map of a convex shape is the union of the unit normal vectors of its tangent planes. For example, 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 the Gauss map of a view-pyramid is a convex quadrilateral (on the unit sphere).

The last detail of our algorithm is the computation of $S \cap p^\downarrow$. 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.

4.1. Remark on the complexity of DSS

In this section, we assume that we are able to find a centerpoint n of a convex spherical polygon S efficiently. Thus, there exists a constant $\mu > 1$ such that $n \cdot v \geq 0 \Rightarrow \text{area}(S \cap v^\downarrow) \leq \text{area}(S)/\mu$. Note that our implementation of DSS does *not* satisfy this assumption: it computes the average of the vertices of the polygon, which is not guaranteed to produce a centerpoint.

After the k-th iteration, if a decision has not yet been reached, the search polygon S must still contain $\mathcal{S}^-(A \ominus B)$. Thus, $\text{area}(\mathcal{S}^-(A \ominus B)) \leq \text{area}(S) \leq 2\pi\mu^{-k}$. This implies

$$k \leq \log_\mu \left(\frac{2\pi}{\text{area}(\mathcal{S}^-(A \ominus B))} \right). \tag{15}$$

Under the above assumption, the number of iterations of DSS is bounded by the logarithm of the inverse of the area of the separating set of $A \ominus B$. When A and B are far away from each other, this area is large (close to 2π, Figure 3 *top*) and thus the number of iterations is small. When A and B are close to tangent to each other, the area of the separating set is very small (Figure 3 *bottom-right*) and the maximal number of iterations is correspondingly larger.

5. Understanding DSS v.s. GJK

5.1. A characterisation of the separating planes

This section derives a characterisation of the planes separating two convex objects that we use in Section 5.2 to understand the differences between our algorithm and GJK. This characterization, embodied in definition (12) and Lemma 2 below.

First, recall that testing that A and B touch each other is equivalent to testing that the origin $\mathbf{o}$ lies in the Minkowski difference $A \ominus B$. In this section, in order to simplify the exposition, we therefore consider the geometrically (but not computationally) equivalent problem of testing that an object P contains the origin $\mathbf{o}$. We assume that P is convex, which is the case when $P = A \ominus B$ and both A and B are convex.

Let us define the *silhouette* of P, $\text{sil}(P)$, as the set of points $p \in \partial P$ such that P admits a tangent plane in p with (outward) normal n so that $h_P(n) = 0$.

When P is a polyhedron, its silhouette $\text{sil}(P)$ is a subset of its faces (vertices, edges and facets). The separating set of P *depends only* on its silhouette vertices:

Algorithm 3 An abstract view of GJK and DSS.

```
1:  function ConvexContainsOrigin(P, P̃_i)
2:      ▷ P̃_i is a convex polyhedron that approximates P: P̃_i ⊂ P.
3:              ▷ P is a convex object. It is assumed that o ∉ P̃_i.
4:      n ← a vector in S⁻(P̃_i), the separating set of P̃_i
5:                                  ▷ S⁻(P̃_i) ≠ ∅ because o ∉ P̃_i
6:      v_{i+1} ← Σ_P(n)                          ▷ h_P(n) = n · v_{i+1}
7:      if h_P(n) < 0 then return Disjoint                ▷ o ∉ P
8:      P̃_{i+1} ← better approximation of P using P̃_i and v_{i+1}
9:      if o ∈ P̃_{i+1} then return Intersection      ▷ o ∈ P̃_{i+1} ⊂ P
10:     return ConvexContainsOrigin(P, P̃_{i+1})       ▷ o ∉ P̃_{i+1}
```

Lemma 2. *When P is a convex polyhedron not containing the origin, $\mathcal{S}^-(P)$ is a non-empty convex spherical polygon on $\mathcal{S}$:*

$$\mathcal{S}^-(P) = \bigcap_{v,\ vertex\ of\ \mathrm{sil}(P)} v^\downarrow = \bigcap_{v,\ vertex\ of\ P} v^\downarrow. \qquad (16)$$

We use Lemma 2, whose proof is given in Appendix A, in the following in order to compare the DSS and GJK algorithms.

5.2. *Comparing* DSS *and GJK*

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 (Algorithm 3):

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 6 of Algorithm 3. The algorithms differ in the polyhedron $\tilde{P}_i$ used to approximate P and in lines 4, 8 and 9 of Algorithm 3. We now examine these differences in turn.

5.2.1. *The polyhedron $\tilde{P}_i$*

In DSS (Algorithm 2 page 4), $\tilde{P}_i$ 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 2 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, $\tilde{P}_{\mathrm{GJK}} \subset \tilde{P}_{\mathrm{DSS}}$, *ie* GJK considers approximations of P of lesser quality (they are smaller, so their separating sets are larger than those of DSS). Also, there is no guarantee that the sequence of considered approximations is increasing, while DSS 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 $\mathcal{S}^-(\tilde{P}_{\mathrm{GJK}})$ having at most 4 vertices (because $\tilde{P}_{\mathrm{GJK}}$ has at most 4 silhouette vertices and Lemma 2). This is a direct evidence that our algorithm is able, in practice as well as in theory, to use more information during its execution.

5.2.2. *Line 9: 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 DSS, we know, by Lemma 2, that the origin lies in $\tilde{P}_{i+1}$ simply when its separating set, S, is empty, which is trivial to check.

5.2.3. *Line 4: picking a new test direction*

In DSS, we pick a vector n as a (approximate) centerpoint of $\mathcal{S}^-(\tilde{P}_i)$. 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: $n = -\nu(\tilde{P}_i)$. 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 $n \mapsto h_{\tilde{P}_i}(n)$.

5.2.4. *Line 8: 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 4 disappears from $\mathcal{S}^-(\tilde{P}_{i+1})$ (by Lemma 1), but only DSS 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 DSS, the separating set of $\tilde{P}_{i+1}$ is computed as the intersection of the separating set of $\tilde{P}_i$ (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 $\mathbf{o} \notin \tilde{P}_i$, let f be the unique facet of $\tilde{P}_i$ that contains $\nu(\tilde{P}_i)$ in its interior. 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 algorithm 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 taken in less than four iterations. The iterations of DSS 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 reach a decision (typically less than four) and GJK is faster on average (see the frustum culling test in Section 6.5). In our experiments, the polyhedron $P = A \ominus B$ is often more complex and DSS is slightly faster. Furthermore, DSS is less prone to numerical inaccuracies caused by floating-point computation, thanks in part to its less complicated implementation. Section 5.3 describes this phenomenon and Section 6 shows experimentally the robustness of DSS.

5.3. Numerical issues in GJK *and* DSS

GJK and DSS are iterative techniques. A typical implementation uses non-exact floating point numbers, so that it is possible that the implementations of GJK or DSS 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, decide 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 than $\Theta - 1$ (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.

In this context, we will see in Section 6 that DSS is more stable than GJK, in the sense that our DSS implementation fails less often than our GJK implementation. In fact, while the failure rate of both implementation is rather small, the failure rate of DSS is more than a thousand times smaller than that of GJK. DSS is also almost never slower than GJK. This let us argue that DSS might be a good candidate to replace GJK in several applications.

Our DSS 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 [19].

5.4. DSS *in physics simulation*

The decision version of GJK (that returns a yes/no answer) is often used in software library for physics simulation (*eg* the *Bullet* Physics Library [3]) since it applies equally well to all kinds of convex shapes, a large variety of which are typically used in such software (from *Bullet*'s class hierarchy: spheres, convex hulls, convex polyhedra, cones, capsules, Minkowski sums, cylinders, triangles, tetrahedra and boxes).

Our DSS algorithm is simpler to implement (see accompanying code). The benchmarks in Section 6 indicate that DSS is also about 10 % faster and fails much less often (see Section 5.3 and Section 6). Thus we believe that DSS can provide a useful replacement for the decision version of GJK (but not for computing the distance between two non-intersecting convex objects, see Section 7).

In a physics simulator, after A and B are found to intersect, the penetration depth (the shortest translation required to separate A from B) is computed, if desired, using the so-called *Expanding Polytope Algorithm* (EPA) [23]. The EPA technique starts from the simplex $\sigma \subset A \ominus B$ containing the origin $\mathbf{o}$, as computed by GJK. It then iteratively expands it away from the closest point on the boundary of σ to $\mathbf{o}$ until an approximation of $A \ominus B$ is reached that does contain the closest point to $\mathbf{o}$ on the boundary of $A \ominus B$, giving the penetration depth. (The EPA technique is used only when A and B are known to intersect because it is costly, since it basically amounts to an incremental convex hull construction that can lead to a polytope with a large number of faces.)

As described, DSS can not be used to compute the penetration depth. However, just like GJK, the spherical polygon S maintained by DSS can be used to compute a starting polytope for EPA. (A similar idea is used in Section 6.5.1.) Indeed the edges of the spherical polygon S do correspond to 3D vertices of $A \ominus B$. Let V be the set of 3D vertices thus representing the edges of S just prior to S becoming empty; $S = (\bigcap_{p \in V} p^{\downarrow}) \neq \emptyset$. And let v be the vertex of $A \ominus B$ such that $S \cap v^{\downarrow} = \emptyset$. Then the convex hull $\mathcal{H}(V \cup \{v\})$ is a polytope that can be used as a starting point for EPA: $\mathbf{o} \in \mathcal{H}(V \cup \{v\}) \subset A \ominus B$.

There are other efficient techniques for computing the penetration depth, such as that of Kim *et al.* [18], which can similarly be efficiently initialized using the vertex of $A \ominus B$ found at the last iteration of the DSS run.

6. Benchmarks

We have compared our implementations of DSS, GJK and other algorithms, 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 a "naive" technique[4] that we described in [16, §3], **Tetra** is an implementation of **Generic** specialized to pairs of tetrahdra [16, §4] and **DSS** is an implementation of our DSS algorithm. Our implementation of GJK

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

[4] The naive technique tests all facets as potential separating plane, then all pairs of edges,one in A and one in B, that are silhouette edges with respect to each other.

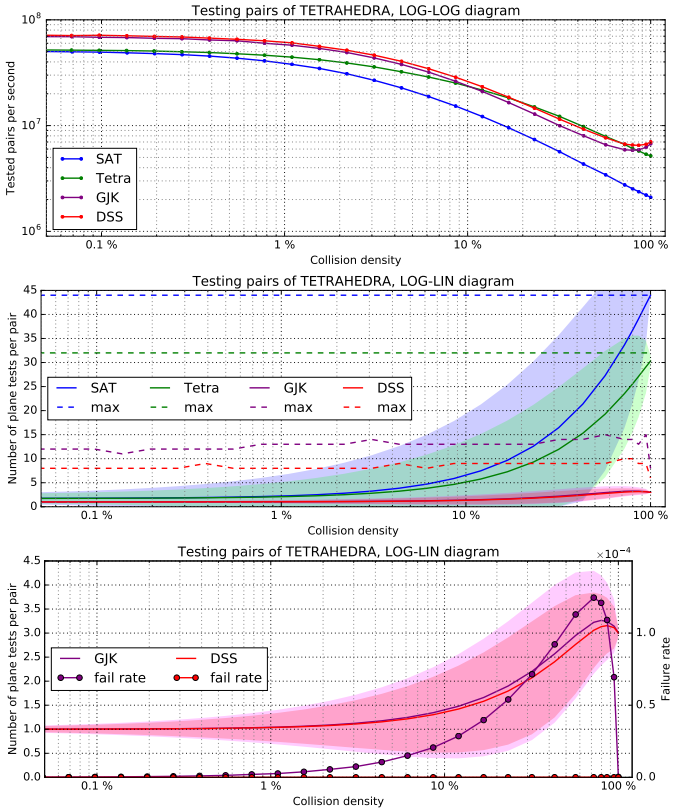

Figure 4: Statistics for the 'random tetrahedra' test.

(called **GJK**) follows reference [22]. The literature describes various algorithmic optimizations of the computation of $\Sigma_P(n)$ (*eg* "hill-climbing" along the edges of P when P is a polyhedron [4], or starting the search from the previous extremal position when P is moving [14]). We have not used these optimizations since they apply equally well to GJK and DSS and only when P is a polyhedron. **SAT** is an implementation of the *Separating Axis Test* that has been taken directly from Gottschalk's PhD manuscript [11]. The abscissa axis is always logarithmic. The ordinate axis is linear or logarithmic as indicated above each diagram.

All the implementations use 32-bits floating-point arithmetic. We have carefully optimized all our tested implementation, but have refrained from using SIMD instructions, which would however clearly help in optimizing further. In particular, the subroutines that find the extremal point on $A \ominus B$ in a given direction are shared by the **GJK** and **DSS** implementations, so that their optimization with SIMD instructions would benefit both equally. Regarding the non-shared parts of these implementations, that of **DSS** (2D polygon clipping) is probably the one that would benefit the most from a "SIMD treatment" (computing the position of points relative to a clipping line, four points at a time). The specifics of **GJK** seem much harder to optimize with SIMD instructions, so that we believe SIMD instruction would mostly be in favor of our new technique, DSS.

The benchmarks are run on a desktop computer with an Intel Core i7-4770K CPU clocking at 3.5 GHz, with 16 GB of RAM clocking at 1.6 GHz. The software is compiled using g++ 4.9 (`c++ -O3 -DNDEBUG`). The OS is Debian Linux.

Some of the statistics below report the average number of plane tests evaluated per pair of objects tested for intersection. This number is independent of the implementation and is therefore valuable for comparing the various techniques.

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

6.1. 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 random distance 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 **Tetra** and **SAT** techniques 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 than **Tetra** in the high-density regime.

DSS and **GJK** have a very similar behavior but **DSS** is consistently faster when density is in the range $[1\,\%, 100\,\%]$. Note that the minimum of the graphs for **DSS** 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 or non-intersecting tetrahedra (because the origin is close to the boundary of $A \ominus B$). In contrast, and contrary to **Tetra** and **SAT**, frank intersections at density 100 % are easier to detect for **GJK** and **DSS**.

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 **Tetra** (32) is always achieved at any collision density since all the tests are required to confirm that two tetrahedra touch each other. Our **DSS** implementation never performed more than 10 plane tests in this benchmark. At the lowest density, **Tetra** and **SAT** require a bit less than two plane tests on average while **GJK** and **DSS** require just one, since the heuristic chooses a initial plane which is separating when tetrahedra are far away from each other. At density 100 %, **SAT** computes 44 interval overlap tests; **Tetra** 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 than the plane test.) **GJK** and **DSS** 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 **DSS**, 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 configurations 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 DSS, the construction of a new test direction does not depend much on the actual geometry of the problem and our **DSS** implementation enjoys a much lower failure rate. For example, at density 72 %, **GJK** failed 24923 times while testing 1.999×10^8 pairs of tetrahedra while **DSS** failed 14 times (at density 56 %). 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 than one per ten thousand pairs.

6.2. 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 $N = 2000$.

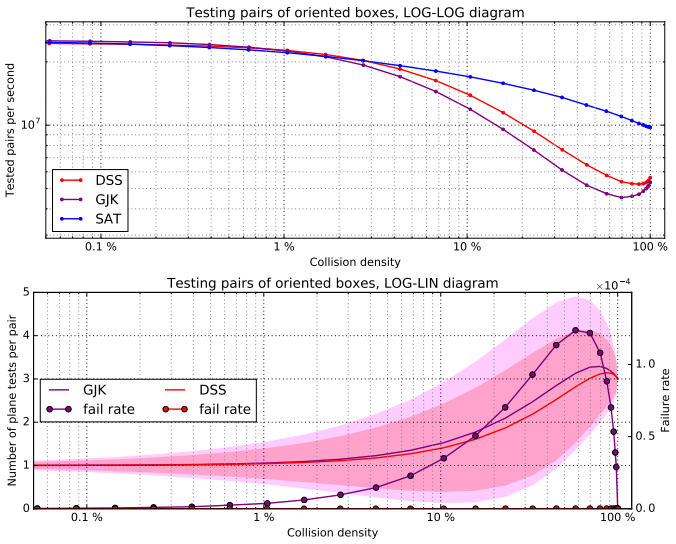

Figure 5: Statistics for the 'random oriented boxes' test.

Figure 5–top diagram. As expected, the specialized **SAT** implementation is faster than **GJK** or **DSS** when the collision density becomes non-negligible. Indeed, while the **GJK** and **DSS** implementations also take advantage of the symmetrical nature of the boxes to accelerate the computation of $h_{A\ominus B}(n)$, 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 **DSS** is 2.04. The largest speed ratio of **SAT** to **GJK** is 2.46.

In this benchmark, **DSS** is faster than **GJK** in the density range $[1\,\%, 100\,\%]$ by as much as 25 %.

Figure 5–bottom diagram. As for tetrahedra, the most striking observation is how our **DSS** technique is able to use fewer test planes than **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 **DSS**.

6.3. 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 **DSS** and $N = 300$ for **Generic**.

Each plane test in the first phase of the **Generic** implementation (see [16, §3]) requires to loop over the vertices of a *single* polytope, instead of looping over the vertices of *both* in order to compute $h_{A\ominus B}(n)$ in **GJK** and **DSS**. 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 **DSS** and **GJK**, we see a trend very similar to the oriented boxes benchmark. **DSS** is again faster and

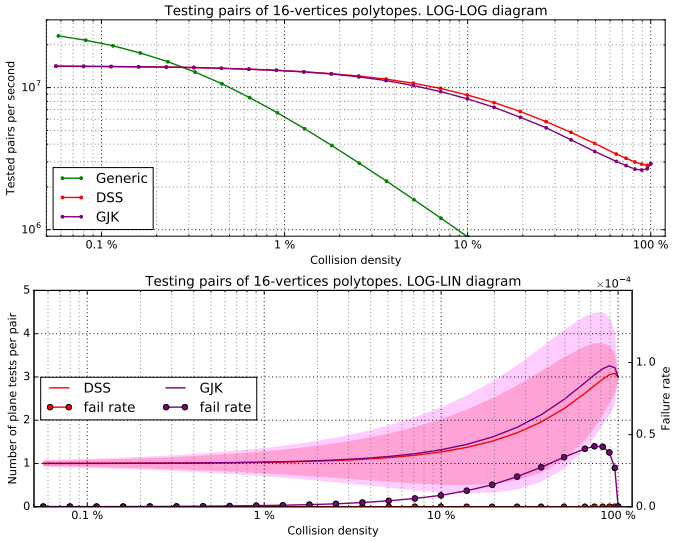

Figure 6: Statistics for the 'random polytopes with 16 vertices' test.

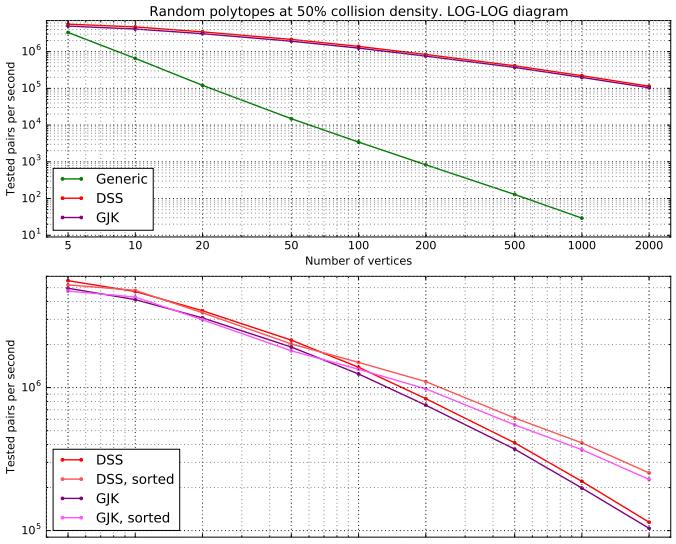

Figure 7: Statistics for the 'random polytopes at 50 % collision density' test.

more robust than **GJK** in the density range $[1\,\%, 100\,\%]$ and up to 14 % faster. In the next benchmark, we fix the collision density at 50 % and vary the number of vertices of the polytopes.

6.4. Varying the number of vertices at 50 % density

In this benchmark, we generate random polytopes in the same way as in Section 6.3, 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 shown in the top diagram, the **Generic** technique exhibits an inverse quadratic dependency on the number v of vertices. The **DSS** 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 **DSS**, 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 **DSS** technique. This explains the constant ratio, of about 1.12, between **DSS** and **GJK**. (The corresponding parallel curves are more easily seen in the zoomed-in 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 **DSS** and **GJK** and more effective with an increasing number of vertices.

6.5. Frustum culling: influence of the relative size

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 (10^8 tested pairs for each sample point). We decrease the collision density by increasing the parameter σ.

At the bottom of Figure 8 we have added a comparison with an implementation of the technique of Greene. The ratio of Greene's technique to the **DSS** technique ranges from 4.13 at low collision density to 2.73 at high density. This ratio is easily explained because Greene's technique is specialized to testing the intersection of an axis-aligned box and a polyhedron. It also pre-computes three sets of silhouette edges on the polyhedra [13]. On the other hand, the **DSS** and **GJK** techniques are fully general and do not pre-process the input convex shapes.

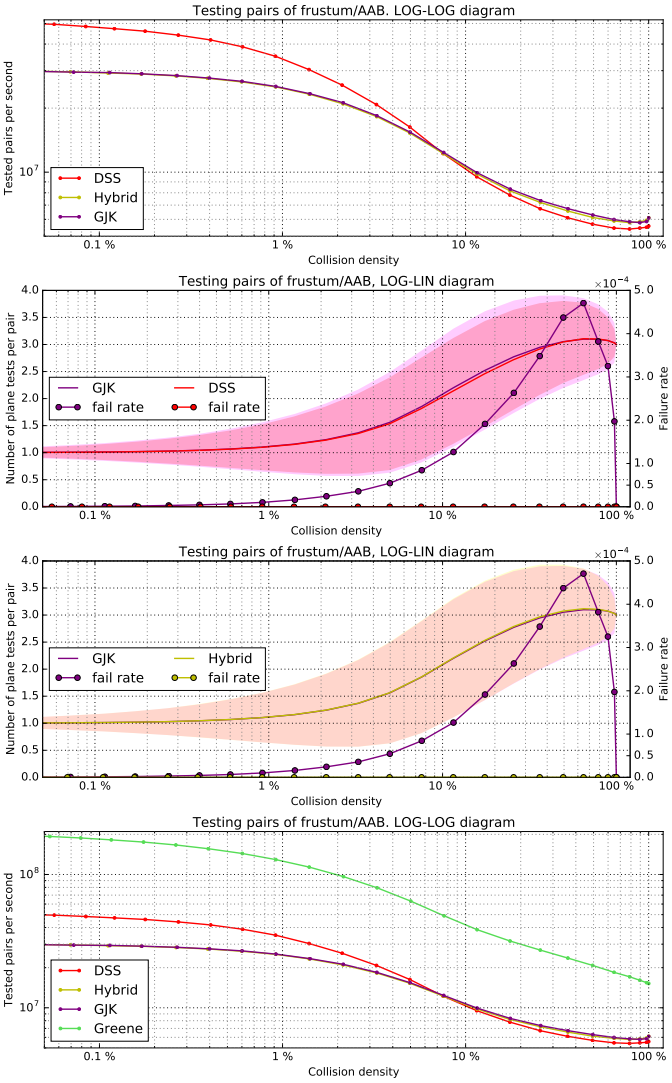

Figure 8: Statistics for frustum culling of axis-aligned boxes. *Bottom.* Comparison with the specialized technique of Greene.

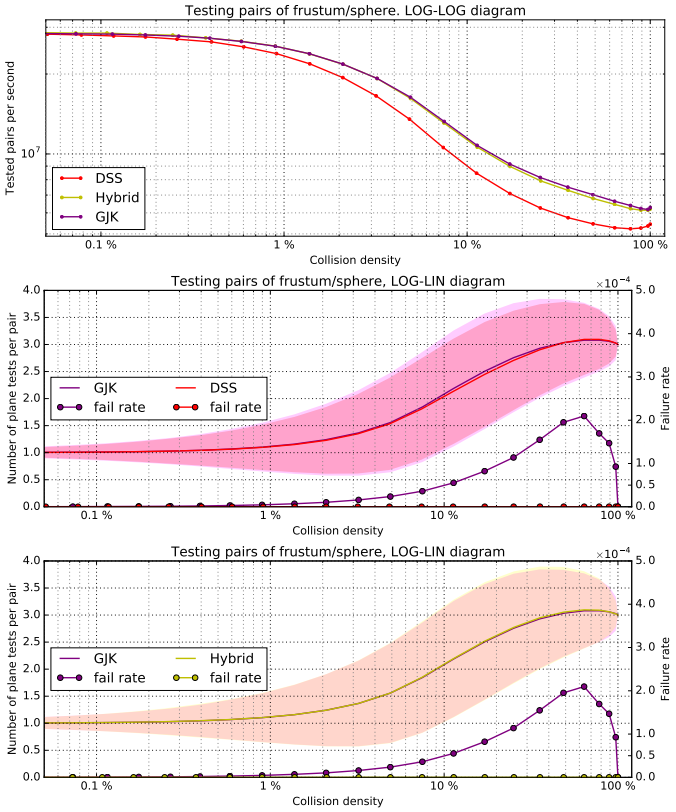

Figure 9: Statistics for frustum culling of spheres.

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 **DSS** technique is slower than **GJK** on almost the whole density range. However, the diagram below indicates that **GJK** still fails far more often than **DSS**.

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

1. Since B is much smaller than 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 [10].

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, DSS 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 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 **DSS** to conclude quickly that A and B are disjoint (equivalently $\mathbf{o} \notin A \ominus B$); **DSS** 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 support function $h_{A \ominus B}$ has a large gradient. This is especially pronounced when A and B are very close to a tangential configuration. In this context, **DSS** needs more iterations to find $\mathcal{S}^-(A \ominus B)$; its dichotomic search works better with a smoother support function.

6.5.1. 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 **DSS**.

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

$$S = \bigcap_{p,p \text{ is a vertex of } \tau} p^{\downarrow} \tag{17}$$

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

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,[5] we obtain again the same behavior as, *eg*, in the oriented-boxes benchmark in Section 6.2, where **DSS** 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 **DSS**. Remarkably, **Hybrid** fails (in the sense given in Section 5.3) even less often that **DSS**, typically once or twice per benchmark, which involves 10^8 tested pairs. We set as future work the task of understanding this interesting behavior.

7. Concluding remarks

We have developed a new algorithm, DSS, to decide the disjointness of two convex shapes, by searching on the 2-sphere. Just as for GJK, DSS simply assumes that the extremal function Σ (page 2) is computable on the shapes at hand. Compared to GJK, DSS is *i)* easier to implement, *ii)* numerically more robust, *iii)* typically a little bit faster and never much slower.

We have however considered only the disjointness decision problem. But GJK can compute the actual distance between the two shapes. It is possible to use DSS in a binary search process that converges to the distance $d(P, \mathbf{o})$, since the *signed* distance from $\mathbf{o}$ to the boundary of $P \oplus \mathbb{B}(r)$ (for $r \geq 0$) is decreasing (as a function of r), is zero precisely at $r = d(P, \mathbf{o})$ and it is possible to decide if $P \oplus \mathbb{B}(r)$ contains the origin using $\Sigma_{P \oplus \mathbb{B}(r)}(n) = \Sigma_P(n) + rn$. While our implementation of this idea works well, we found it difficult to make it competitive with GJK.

Acknowledgment. This work was partially supported by ERC grant ShapeForge (StG-2012-307877).

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

References

[1] Barba, L., Langerman, S., 2015. Optimal detection of intersections between convex polyhedra. In: Proceedings of the Annual Symposium on Discrete Algorithms (SODA). ACM-SIAM. URL `http://arxiv.org/abs/1312.1001`

[2] Bauschke, H. H., Borwein, J. M., 1996. On projection algorithms for solving convex feasibility problems. SIAM Review 38 (3), 367–426.

[3] Bullet, 2015. The Bullet physics library. URL `http://bulletphysics.org/wordpress/`

[4] Cameron, S., 1997. Enhancing GJK: Computing minimum and penetration distances between convex polyhedra. In: Proceedings of International Conference on Robotics and Automation. pp. 3112–3117.

[5] Choi, Y.-K., Li, X., Rong, F., Wang, W., Cameron, S., 2010. Determining the directional contact range of two convex polyhedra. Computers-Aided Design 42 (1), 27–35.

[6] Clarkson, K. L., Eppstein, D., Miller, G. L., Sturtivant, C., Teng, S.-T., 1996. Approximating center points with iterated radon points. International Journal of Computational Geometry & Applications 6 (3).

[7] Eberly, D., 2008. Intersection of ellipsoids. Tech. rep., Geometric Tools, LLC. URL `http://www.geometrictools.com/Documentation/\IntersectionOfEllipsoids.pdf`

[8] Ericson, C., 2004. Real-Time Collision Detection. CRC Press.

[9] Gilbert, E. G., Foo, C.-P., Feb. 1990. Computing the distance between general convex objects in three-dimensional space. IEEE Journal of Robotics And Automation 6 (1), 53–61.

[10] Gilbert, E. G., Johnson, D. W., Keerthi, S., Apr. 1988. A fast procedure for computing the distance between complex objects in three-dimensional space. IEEE Journal of Robotics And Automation 4 (2).

[11] Gottschalk, S., 2000. Collision queries using oriented bounded boxes. Ph.D. thesis, UNC Chapel Hill.

[12] Gottschalk, S., Lin, M. C., Manocha, D., 1996. OBBTree: A hierarchical structure for rapid interference detection. In: Proc. SIGGRAPH. Annual Conference Series. ACM, ACM.

[13] Greene, N., 1994. Detecting intersection of a rectangular solid and a convex polyhedron. In: Heckbert, P. S. (Ed.), Graphics Gems IV. Academic Press, Ch. I.7, pp. 74–82.

[14] Guibas, L. J., Hsu, D., Zhang, L., 2000. A hierarchical method for real-time distance computation among moving convex bodies. Computer Geometry, Theory and Applications 15 (1–3), 51–68.

[15] Guibas, L. J., Nguyen, A., Zhang, L., 2003. Zonotopes as bounding volumes. In: Proceedings of the Annual Symposium on Discrete Algorithms (SODA). ACM-SIAM.

[16] Hornus, S., 2015. Intersection detection via gauss maps; a review and new techniques. Tech. Rep. 8730, Inria.

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

[18] Kim, Y. J., Lin, M. C., Manocha, D., 2004. Incremental penetration depth estimation between convex polytopes using dual-space expansion. IEEE Transactions on Visualization & Computer Graphics 10 (2), 152–163.

[19] Pion, S., Fabri, A., Apr. 2011. A generic lazy evaluation scheme for exact geometric computations. Science of Computer Programming 76 (4), 307–323. URL `https://hal.inria.fr/inria-00562300`

[20] Sabin, M., 1974. Surfaces closed under five important geometric operations. Tech. Rep. VTO/MS/207, British Aircraft Corporation.

[21] Šír, Z., Gravesen, J., Jüttler, B., 2007. Computing convolutions and minkowski sums via support functions. In: Curve and Surface Design. Nashboro Press, pp. 244–253.

[22] van der Bergen, G., 1999. A fast and robust GJK implementation for collision detection of convex objects. Journal of Graphics Tools 4 (2), 7—25.

[23] van der Bergen, G., 2001. Proximity queries and penetration

depth computation on 3D game objects. In: In Game Developers Conference.

Appendix A. Proof of Lemma 2

Lemma 3. $\mathcal{S}^-(P) = \bigcap_{p\in P} p^\downarrow$.

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

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 $h_P(n) = 0$. Lemma 4 below shows that the separating set of P depends only on the silhouette of P.

Lemma 4. *If* $\mathbf{o}$ *is not in* 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, by Lemma 3, 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 $p = \alpha s_1 + \beta s_2$ where s_1 and s_2 belong to $\mathrm{sil}(P)$, $\alpha \geq 0$, $\beta \geq 0$ and $\alpha + \beta > 0$ (since $p \neq \mathbf{o}$). Since $n \in s_1^\downarrow \cap s_2^\downarrow$, we have that $n \cdot s_1 < 0$ and $n \cdot s_2 < 0$. Therefore $n \cdot p < 0$ and $n \in \mathcal{S}^-(P)$. □

When P is a polyhedron, its silhouette $\mathrm{sil}(P)$ is a subset of its faces (vertices, edges and facets). The separating set of P depends only on its silhouette vertices:

Lemma 2. *When* P *is a convex polyhedron not containing the origin,* $\mathcal{S}^-(P)$ *is a non-empty convex spherical polygon on* $\mathcal{S}$*:*

$$\mathcal{S}^-(P) = \bigcap_{v,\ vertex\ of\ \mathrm{sil}(P)} v^\downarrow = \bigcap_{v,\ vertex\ of\ P} v^\downarrow. \quad \text{(A.1)}$$

Proof. The left equality is obtained from Lemma 4 by considering and simplifying the contribution of each silhouette edge. The right equality follows directly from Lemma 3. □