A Fast and Robust GJK Implementation for Collision Detection of Convex Objects

GINO VAN DEN BERGEN

Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
E-mail: gino@win.tue.nl

July 6, 1999

Abstract

This paper presents an implementation of the Gilbert-Johnson-Keerthi algorithm for computing the distance between convex objects, that has improved performance, robustness, and versatility over earlier implementations. The algorithm presented here is especially fit for use in collision detection of objects modeled using various types of geometric primitives, such as boxes, cones, and spheres, and their images under affine transformation, for instance, as described in VRML.

1 Introduction

The Gilbert-Johnson-Keerthi distance algorithm (GJK) is an iterative method for computing the distance between convex objects [9]. The attractiveness of GJK lies in its simplicity, which makes it fairly easy to implement, and its applicability to general convex polytopes [8].

This paper presents a GJK that has improved performance, robustness, and versatility over earlier implementations [13, 3]. The performance improvements are the result of: (a) data caching and smarter selection of sub-simplices in the

GJK subalgorithm, (b) early termination on finding a separating axis, in the application of GJK to intersection detection, and (c) exploitation of frame coherence by caching the separating axis. With these improvements, our GJK implementation detects intersections between convex polyhedra roughly five times faster than the implementation of the Lin-Canny closest-feature algorithm [10] used in I-COLLIDE [7]. Regarding robustness, we present a solution to a termination problem in the original GJK due to rounding errors, which was noted by Nagle [11]. Finally, regarding versatility, we show how GJK can be applied to a large family of geometric primitives, which includes boxes, spheres, cones and cylinders, and their images under affine transformation, thus demonstrating the usefulness of GJK for collision detection of objects described in VRML [2].

The GJK implementation presented in this paper is incorporated in the Software Library for Interference Detection (SOLID) version 2.0. The C++ source code for our GJK implementation is also released as a separate package called *GJK-engine*[1].

The rest of this paper is organized as follows. Section 2 describes the general GJK distance algorithm. Readers already familiar with this algorithm may safely skip this section. Section 3 discusses the application of GJK on a number of convex primitives and their images under affine transformation. Section 4 discusses the performance improvements of our implementation. And finally, Section 5 shows what causes the numerical problems of GJK and how to tackle these problems.

2 Overview of GJK

This section describes the extended GJK for general convex objects, first presented in [8]. Readers unfamiliar with the concepts and notations used here are referred to Appendix A.

The GJK algorithm computes the distance between a pair of convex objects. The *distance* between objects A and B, denoted by $d(A, B)$, is defined by

$$d(A, B) = \min\{\|\mathbf{x} - \mathbf{y}\| : \mathbf{x} \in A, \mathbf{y} \in B\}.$$

The algorithm can be tailored to return a pair of closest points, which is a pair of points $\mathbf{a} \in A$ and $\mathbf{b} \in B$ for which $\|\mathbf{a} - \mathbf{b}\| = d(A, B)$.

[1]The GJK-engine source code as well as information on obtaining the complete C++ source code and documentation for SOLID 2.0 is available online at *http://www.acm.org/jgt/papers/vanDenBergen99/.*

We express the distance between A and B in terms of their Minkowski sum $A - B$ as

$$d(A, B) = \|v(A - B)\|,$$

where $v(C)$ is defined as the point in C nearest to the origin, i.e.,

$$v(C) \in C \quad \text{and} \quad \|v(C)\| = \min\{\|\mathbf{x}\| : \mathbf{x} \in C\}.$$

Clearly, for $\mathbf{a} \in A$ and $\mathbf{b} \in B$ a pair of closest points, we have $\mathbf{a} - \mathbf{b} = v(A - B)$.

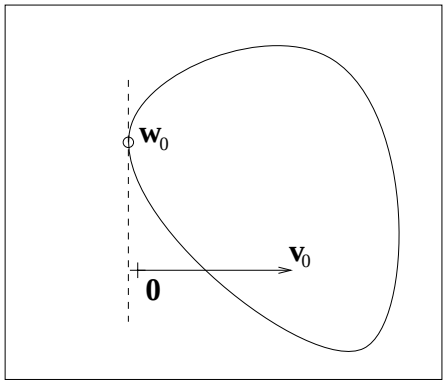
GJK is essentially a descent method for approximating $v(A - B)$ for convex A and B. In each iteration a simplex is constructed that is contained in $A - B$ and lies nearer to the origin than the simplex constructed in the previous iteration. We define W_k as the set of vertices of the simplex constructed in the k-th iteration ($k \geq 1$), and $\mathbf{v}_k$ as $v(\text{conv}(W_k))$, the point in the simplex nearest to the origin. Initially, we take $W_0 = \emptyset$, and $\mathbf{v}_0$, an arbitrary point in $A - B$. Since $A - B$ is convex and $W_k \subseteq A - B$, we see that $\mathbf{v}_k \in A - B$, and thus $\|\mathbf{v}_k\| \geq \|v(A - B)\|$ for all $k \geq 0$.

GJK generates the sequence of simplices in the following way. Let $\mathbf{w}_k = s_{A-B}(-\mathbf{v}_k)$, where s_{A-B} is a support mapping of $A - B$. We take $\mathbf{v}_{k+1} = v(\text{conv}(W_k \cup \{\mathbf{w}_k\}))$, and as W_{k+1} we take the smallest set $X \subseteq W_k \cup \{\mathbf{w}_k\}$, such that $\mathbf{v}_{k+1}$ is contained in $\text{conv}(X)$. It can be seen that exactly one such X exists, and that it must be affinely independent. Figure 1 illustrates a sequence of iterations of the GJK algorithm in two dimensions. We refer to [8] for a proof of global convergence of the sequence $\{\|\mathbf{v}_k\|\}$.

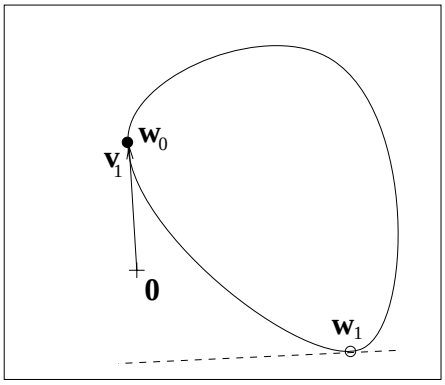
For polytopes, GJK arrives at $\mathbf{v}_k = v(A - B)$ in a finite number of iterations, as shown in [9]. For non-polytopes this may not be the case. For these type of objects, it is necessary that the algorithm terminates as soon as $\mathbf{v}_k$ lies within a given tolerance from $v(A - B)$. The error of $\mathbf{v}_k$ is estimated by maintaining a lower bound for $\|v(A - B)\|$. As a lower bound we may take the signed distance from the origin to the supporting plane $H(-\mathbf{v}_k, \mathbf{v}_k \cdot \mathbf{w}_k)$, which is

$$\delta_k = \mathbf{v}_k \cdot \mathbf{w}_k / \|\mathbf{v}_k\|.$$

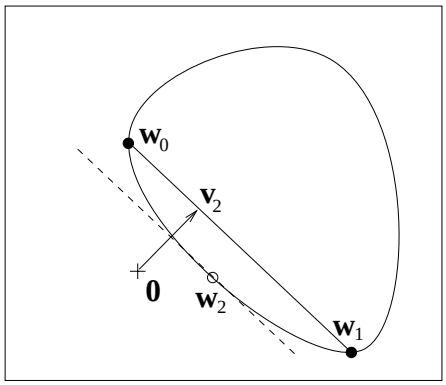
This is a proper lower bound since for positive δ_k, the origin lies in the positive halfspace, whereas $A - B$ is contained in the negative halfspace of the plane. In Figure 1 we see that δ_k is positive in the cases where the dashed line crosses the arrow.



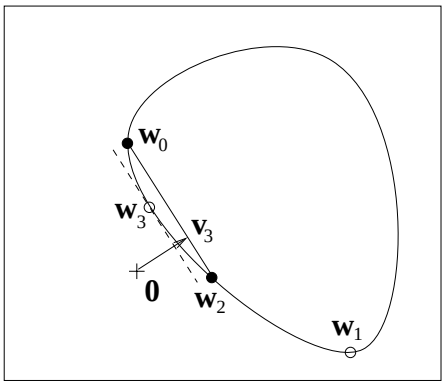
(a) $k = 0$, $W = \emptyset$, $\delta < 0$, $\mu = 0$



(b) $k = 1$, $W = \{\mathbf{w}_0\}$, $\delta < 0$, $\mu = 0$



(c) $k = 2$, $W = \{\mathbf{w}_0, \mathbf{w}_1\}$, $\delta > 0$, $\mu = \delta$



(d) $k = 3$, $W = \{\mathbf{w}_0, \mathbf{w}_2\}$, $\delta > 0$, μ unchanged

Figure 1: Four iterations of the GJK algorithm (See Algorithm 1). The dashed lines represent the support planes $H(-\mathbf{v}_k, \mathbf{v}_k \cdot \mathbf{w}_k)$. The points of W_k are drawn in black.

Contrary to $\|\mathbf{v}_k\|$, the lower bound δ_k may not be monotonic in k, i.e., it is possible that $\delta_j < \delta_i$ for $j > i$. Furthermore, 0 is a trivial lower bound for $\|v(A - B)\|$. Hence, we use

$$\mu_k = \max\{0, \delta_0, \ldots, \delta_k\}$$

as a lower bound, which is often tighter than δ_k.

A monotonic lower bound is needed for the following reason. For certain configurations of objects (in particular objects that have flat boundary elements), δ is *ill-conditioned*, i.e., a small change in $\mathbf{v}$ may result in a large change in δ. Since the computation of $\mathbf{v}$ with finite precision arithmetics inevitably suffers from rounding errors, the computed value for δ may be considerably smaller than its actual value. The relative error in $\mathbf{v}$ is larger for sets W that are close to being affinely dependent, as we will see Section 5. GJK has a tendency to generate simplices that are progressively more oblong, i.e., closer to being affinely dependent, as the number of iterations increases. Hence, for large k, the computed values for δ_k may be less reliable.

Given ε, a tolerance for the absolute error in $\|\mathbf{v}_k\|$, the algorithm terminates as soon as $\|\mathbf{v}_k\| - \mu_k \leq \varepsilon$. Algorithm 1 presents pseudo-code for the GJK distance algorithm.

We now focus on the computation of $\mathbf{v} = v(\text{conv}(Y))$ for an affinely independent set Y and the determination of the smallest $X \subseteq Y$ such that $\mathbf{v} \in \text{conv}(X)$. These operations are performed by a single subalgorithm. The requested subset $X = \{\mathbf{x}_0, \ldots, \mathbf{x}_n\}$ of Y is characterized by

$$\mathbf{v} = \sum_{i=0}^{n} \lambda_i \mathbf{x}_i \quad \text{where} \quad \sum_{i=0}^{n} \lambda_i = 1 \quad \text{and} \quad \lambda_i > 0.$$

This subset X is the largest of all nonempty $Z \subseteq Y$ for which each λ_i of the point $v(\text{aff}(Z))$ is positive. The requested point $\mathbf{v}$ is the point $v(\text{aff}(X))$.

It remains to explain how to find the λ_i values for $v(\text{aff}(X))$, where X is affinely independent. We observe that the vector $\mathbf{v} = v(\text{aff}(X))$ is perpendicular to $\text{aff}(X)$, i.e., $\mathbf{v} \in \text{aff}(X)$ and $\mathbf{v} \cdot (\mathbf{x}_i - \mathbf{x}_0) = 0$ for $i = 1, \ldots, n$. Hence, the λ_i values are found by solving a system of linear equations. We apply Cramer's rule to solve these systems of equations. Since we need to find solutions for all nonempty subsets of Y, we exploit the recursion in Cramer's rule. Let $Y = \{\mathbf{y}_0, \ldots, \mathbf{y}_n\}$, where $n \leq d$, the dimension of the space. In our case, $d = 3$. Each nonempty $X \subseteq Y$ is identified by a nonempty $I_X \subseteq \{0, \ldots, n\}$ such that $X = \{\mathbf{y}_i : i \in I_X\}$.

Algorithm 1 The GJK distance algorithm

```
v := "arbitrary point in A − B";
W := ∅;
μ := 0;
close_enough := false;
while not close_enough and v ≠ 0 do begin
   w := s_{A−B}(−v);
   δ := v · w/‖v‖;
   μ := max{μ, δ};
   close_enough := ‖v‖ − μ ≤ ε;
   if not close_enough then begin
      v := v(conv(W ∪ {w}));
      W := "smallest X ⊆ W ∪ {w} such that v ∈ conv(X)";
   end
end;
return ‖v‖
```

We obtain the following recursively defined solutions. For each subset X, we have $\lambda_i = \Delta_i(X)/\Delta(X)$, where $\Delta(X) = \sum_{i \in I_X} \Delta_i(X)$, and

$$\begin{aligned} \Delta_i(\{\mathbf{y}_i\}) &= 1 \\ \Delta_j(X \cup \{\mathbf{y}_j\}) &= \sum_{i \in I_X} \Delta_i(X)(\mathbf{y}_i \cdot \mathbf{y}_k - \mathbf{y}_i \cdot \mathbf{y}_j), \end{aligned}$$

where $j \notin I_X$ and k is an arbitrary but fixed member of I_X, for instance $k = \min(I_X)$. The smallest $X \subseteq Y$ such that $\mathbf{v} \in \text{conv}(X)$ can now be characterized as the subset X for which (i) $\Delta_i(X) > 0$ for each $i \in I_X$, and (ii) $\Delta_j(X \cup \{\mathbf{y}_j\}) \leq 0$, for all $j \notin I_X$. The subalgorithm successively tests each nonempty subset X of Y until it finds one for which (i) and (ii) hold. In Section 4 we discuss how the subalgorithm is further optimized in order to improve performance.

Finally, a pair of closest points is computed as follows. At termination, we have a representation of $\mathbf{v} \approx v(A - B)$ as

$$\mathbf{v} = \sum_{i=0}^{n} \lambda_i \mathbf{y}_i \quad \text{where} \quad \sum_{i=0}^{n} \lambda_i = 1 \quad \text{and} \quad \lambda_i > 0.$$

Each $\mathbf{y}_i = \mathbf{p}_i - \mathbf{q}_i$, where $\mathbf{p}_i$ and $\mathbf{q}_i$ are support points of respectively A and B. Let $\mathbf{a} = \sum_{i=0}^{n} \lambda_i \mathbf{p}_i$ and $\mathbf{b} = \sum_{i=0}^{n} \lambda_i \mathbf{q}_i$. Since A and B are convex, it is clear that $\mathbf{a} \in A$ and $\mathbf{b} \in B$. Furthermore, it can be seen that $\mathbf{a} - \mathbf{b} = \mathbf{v}$. Hence, $\mathbf{a}$ and $\mathbf{b}$ are closest points of A and B.

3 Support Mappings

In order to use GJK on a given class of objects, all we need is a support mapping for that class. In this section we discuss the computation of the support points for a number of geometric primitives and their images under affine transformation.

Polytope

The set of polytopes includes simplices (points, line segments, triangles, and tetrahedra), convex polygons, and convex polyhedra. For a polytope A, we may take $s_A(\mathbf{v}) = s_{\text{vert}(A)}(\mathbf{v})$, i.e.,

$$s_A(\mathbf{v}) \in \text{vert}(A) \quad \text{where} \quad \mathbf{v} \cdot s_A(\mathbf{v}) = \max\{\mathbf{v} \cdot \mathbf{x} : \mathbf{x} \in \text{vert}(A)\}.$$

Obviously, a support point of a polytope can be computed in linear time with respect to the number of vertices of the polytope. However, it has been mentioned in a number of publications [4, 5, 3, 12] that by exploiting frame coherence, the cost of computing a support point of a convex polyhedron can be reduced to almost constant time. For this purpose, an adjacency graph of the vertices is maintained with each polytope. Each edge on the polytope is an edge in the graph. In this way, a support point that lies close to the previously returned support point can be found much faster using local search. This technique is commonly referred to as *hill climbing*. In our implementation, we use *Qhull* [1] for computing the adjacency graph of a polytope.

Box

A Box primitive is a rectangular parallelepiped centered at the origin and aligned with the coordinate axes. Let A be a Box with extents $2\eta_x$, $2\eta_y$, and $2\eta_y$. Then, we take as support mapping for A,

$$s_A((x, y, z)^{\mathrm{T}}) = (\text{sgn}(x)\eta_x, \text{sgn}(y)\eta_y, \text{sgn}(z)\eta_z)^{\mathrm{T}},$$

where $\text{sgn}(x) = -1$, if $x < 0$, and 1, otherwise.

Sphere

A Sphere primitive is a ball centered at the origin. The support mapping of a Sphere A with radius ρ is

$$s_A(\mathbf{v}) = \begin{cases} \frac{\rho}{\|\mathbf{v}\|}\mathbf{v} & \text{if } \mathbf{v} \neq \mathbf{0} \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

Cone

A Cone primitive is a capped cone that is centered at the origin and whose central axis is aligned with the y-axis. Let A be a Cone with a radius of ρ at its base, and with its apex at $y = \eta$ and its base at $y = -\eta$. Then, the for the top angle α we have $\sin(\alpha) = \rho/\sqrt{\rho^2 + (2\eta)^2}$. Let $\sigma = \sqrt{x^2 + z^2}$, the distance from $(x, y, z)^{\mathrm{T}}$ to the y-axis. We choose as support mapping for A, the mapping

$$s_A((x, y, z)^{\mathrm{T}}) = \begin{cases} (0, \eta, 0)^{\mathrm{T}} & \text{if } y > \|(x, y, z)^{\mathrm{T}}\| \sin(\alpha) \\ (\frac{\rho}{\sigma}x, -\eta, \frac{\rho}{\sigma}z)^{\mathrm{T}} & \text{else, if } \sigma > 0 \\ (0, -\eta, 0)^{\mathrm{T}} & \text{otherwise.} \end{cases}$$

Cylinder

A Cylinder primitive is a capped cylinder that again is centered at the origin and whose central axis is aligned with the y-axis. Let A be a Cylinder with a radius of ρ, and with its top at $y = \eta$ and its bottom at $y = -\eta$. We find as support mapping for A the mapping

$$s_A((x, y, z)^{\mathrm{T}}) = \begin{cases} (\frac{\rho}{\sigma}x, \operatorname{sgn}(y)\eta, \frac{\rho}{\sigma}z)^{\mathrm{T}} & \text{if } \sigma > 0 \\ (0, \operatorname{sgn}(y)\eta, 0)^{\mathrm{T}} & \text{otherwise.} \end{cases}$$

Affine Transformation

Given a class of objects for which we have a support mapping, the following theorem yields a method for computing support points for images under affine transformations of objects of this class.

Theorem 1. *Given s_A, a support mapping of object A, and $\mathbf{T}(\mathbf{x}) = \mathbf{B}\mathbf{x} + \mathbf{c}$, an affine transformation, a support mapping for $\mathbf{T}(A)$, the image of A under $\mathbf{T}$, is*

$$s_{\mathbf{T}(A)}(\mathbf{v}) = \mathbf{T}(s_A(\mathbf{B}^{\mathrm{T}}\mathbf{v})).$$

Proof. A support mapping $s_{\mathbf{T}(A)}$ is characterized by

$$\mathbf{v} \cdot s_{\mathbf{T}(A)}(\mathbf{v}) = \max\{\mathbf{v} \cdot \mathbf{T}(\mathbf{x}) : \mathbf{x} \in A\}.$$

We rewrite the right member of this equation using the following deduction.

$$\mathbf{v} \cdot \mathbf{T}(\mathbf{x}) = \mathbf{v} \cdot \mathbf{B}\mathbf{x} + \mathbf{v} \cdot \mathbf{c} = \mathbf{v}^{\mathrm{T}}\mathbf{B}\mathbf{x} + \mathbf{v} \cdot \mathbf{c} = (\mathbf{B}^{\mathrm{T}}\mathbf{v})^{\mathrm{T}}\mathbf{x} + \mathbf{v} \cdot \mathbf{c} = (\mathbf{B}^{\mathrm{T}}\mathbf{v}) \cdot \mathbf{x} + \mathbf{v} \cdot \mathbf{c}.$$

This equation is used in the steps marked by $(*)$ in the following deduction.

$$\begin{aligned} \max\{\mathbf{v} \cdot \mathbf{T}(\mathbf{x}) : \mathbf{x} \in A\} &\overset{(*)}{=} \max\{(\mathbf{B}^{\mathrm{T}}\mathbf{v}) \cdot \mathbf{x} + \mathbf{v} \cdot \mathbf{c} : \mathbf{x} \in A\} \\ &= \max\{(\mathbf{B}^{\mathrm{T}}\mathbf{v}) \cdot \mathbf{x} : \mathbf{x} \in A\} + \mathbf{v} \cdot \mathbf{c} \\ &= (\mathbf{B}^{\mathrm{T}}\mathbf{v}) \cdot s_A(\mathbf{B}^{\mathrm{T}}\mathbf{v}) + \mathbf{v} \cdot \mathbf{c} \\ &\overset{(*)}{=} \mathbf{v} \cdot \mathbf{T}(s_A(\mathbf{B}^{\mathrm{T}}\mathbf{v})) \end{aligned}$$

Hence, $s_{\mathbf{T}(A)}(\mathbf{v}) = \mathbf{T}(s_A(\mathbf{B}^{\mathrm{T}}\mathbf{v}))$ is a support mapping of $\mathbf{T}(A)$. □

4 Speed

This section presents a fast implementation of the subalgorithm and a collision detection algorithm derived from the GJK distance algorithm.

Subalgorithm

It is hinted in [9] that by caching and reusing results of dot products, substantial performance improvement can be obtained. Since some or all vertices in W_k reappear in W_{k+1}, many dot products from the k-th iteration are also needed in the $k+1$-th iteration. We will show how this caching of dot products is implemented efficiently.

In order to minimize the caching overhead, we assign an index number to each new support point, which is invariant for the duration that the support point is a member of $W_k \cup \{\mathbf{w}_k\}$. Since $W_k \cup \{\mathbf{w}_k\}$ has at most four points, and each point that is discarded will not reappear, we need to cache data for only four points. The support points are stored in an array $\mathbf{y}$. The index of each support point is its array index. The set W_k is identified by a subset of $\{0, 1, 2, 3\}$, which is implemented as a bit-array b, i.e., $W_k = \{\mathbf{y}[i] : b[i] = 1, i = 0, 1, 2, 3\}$. The index number of the new support point $\mathbf{w}_k$ is the smallest i for which $b[i] = 0$. Note that a free 'slot'

for $\mathbf{w}_k$ is always available during iterations, because if W_k has four elements, then $\mathbf{v}_k = v(\text{conv}(W_k))$ must be zero, since W_k is affinely independent, in which case the algorithm terminates immediately without computing a support point.

The dot products of all pairs $\mathbf{y}[i], \mathbf{y}[j] \in W_k \cup \{\mathbf{w}_k\}$ are stored in a 4×4 array d, i.e., $d[i, j] = \mathbf{y}[i] \cdot \mathbf{y}[j]$. In each iteration, we need to compute the dot products of the pairs containing $\mathbf{w}_k$ only. The other dot products are already computed in previous iterations. For a W_k containing n points, this takes $n + 1$ dot product computations.

We further improve the performance of the subalgorithm by caching all the $\Delta_i(X)$ values. Let $Y = W_k \cup \{\mathbf{w}_k\}$. For many of the $X \subseteq Y$, the $\Delta_i(X)$ values are needed in several iterations, and are therefore better cached and reused rather than recomputed. For this purpose, each subset X is identified by the integer value of the corresponding bit-array. For instance, for $X = \{\mathbf{y}[0], \mathbf{y}[3]\}$, we use bit-array 1001, corresponding to integer value $2^0 + 2^3 = 9$. The values of $\Delta_i(X)$ for each subset X are stored in a 16×4 array D. The element $D[x, i]$ stores the value of $\Delta_i(X)$, where x is the integer value corresponding with subset X. Only the elements $D[x, i]$ for which bit i of bit-array x is set, are used. Similar to the dot product computations, we only need to compute, in each k-th iteration, the values of $\Delta_i(X)$ for the subsets X containing the new support point $\mathbf{w}_k$, since the other values are computed in previous iterations.

Another improvement concerning the subalgorithm is based on the following theorem.

Theorem 2. *For each k-th iteration, where $k \geq 1$, we have $\mathbf{w}_k \in W_{k+1}$.*

Proof. Suppose $\mathbf{w}_k \notin W_{k+1}$, then $v(W_{k+1}) = v(W_k)$, and thus $\mathbf{v}_{k+1} = \mathbf{v}_k$. But since $\|\mathbf{v}_{k+1}\| < \|\mathbf{v}_k\|$, this yields a contradiction. □

Consequently, only those subsets X, for which $\mathbf{w}_k \in X$ need to be tested by the subalgorithm. This reduces the number of subsets from $2^{n+1} - 1$ to 2^n, where n is the number of elements in W_k.

Separating Axis

For deciding whether two objects intersect, we do not need to have the distance between them. We merely need to know whether the distance is equal to zero or not. Hence, as soon as the lower bound for the distance becomes positive, the algorithm may terminate returning a nonintersection. The lower bound is positive iff $\mathbf{v}_k \cdot \mathbf{w}_k > 0$, i.e., $\mathbf{v}_k$ is a *separating axis* of A and B. In general, GJK

needs less iterations for finding a separating axis of a pair of nonintersecting objects than for computing an accurate approximation of $v(A - B)$. For instance in Figure 1, the vector $\mathbf{v}_k$ is a separating axis for the first time when $k = 2$. If the objects intersect, then the algorithm terminates on $\mathbf{v}_k = \mathbf{0}$, returning an intersection. Algorithm 2 shows an algorithm for computing a separating axis, which is derived from the GJK distance algorithm. Besides requiring fewer iterations

Algorithm 2 The GJK separating-axis algorithm

$\mathbf{v} :=$ "arbitrary vector";
$W := \emptyset$;
repeat
  $\mathbf{w} := s_{A-B}(-\mathbf{v})$;
  if $\mathbf{v} \cdot \mathbf{w} > 0$ then return false;
  $\mathbf{v} := v(\text{conv}(W \cup \{\mathbf{w}\}))$;
  $W :=$ "smallest $X \subseteq W \cup \{\mathbf{w}\}$ such that $\mathbf{v} \in \text{conv}(X)$";
until $\mathbf{v} = \mathbf{0}$;
return true

in case of nonintersecting objects, the separating-axis algorithm performs better than the distance algorithm for another reason. Notice that the value of $\|\mathbf{v}\|$ is not needed in the GJK separating-axis algorithm. The computation of $\|\mathbf{v}\|$ involves evaluating a square root, which is an expensive operation. A single iteration of the separating-axis algorithm is therefore significantly cheaper than an iteration of the distance algorithm.

Also, note that in the separating-axis algorithm, $\mathbf{v}$ does not need to be initialized by a point in $A - B$, since the length of $\mathbf{v}$ does not matter. This feature is convenient for exploiting frame coherence.

Similar to closest-point tracking algorithms, such as the Lin-Canny closest feature algorithm [10], and Cameron's Enhanced GJK algorithm [3], an incremental version of the GJK separating-axis algorithm shows almost constant time complexity per frame for convex objects of arbitrary complexity, if frame coherence is high. The incremental separating-axis GJK algorithm, further referred to as ISA-GJK, exploits frame coherence by using the separating axis from the previous frame as initial vector for $\mathbf{v}$. If the degree of coherence between frames is high, then the separating axis from the previous frame is likely to be a separating axis in the current frame, in which case ISA-GJK terminates in the first iteration.

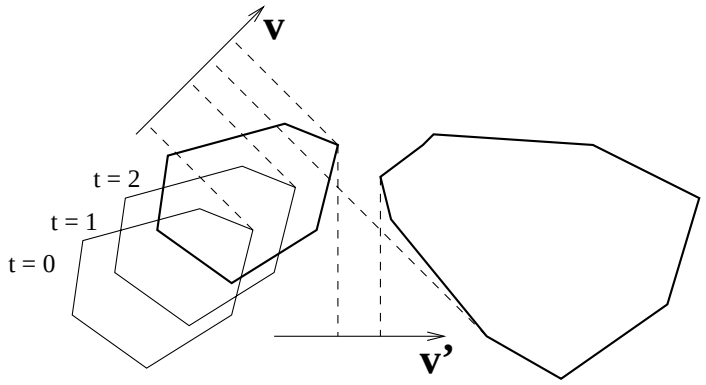


Figure 2: Incremental separating axis computation using ISA-GJK. The separating axis $\mathbf{v}$ from $t = 0$ is also a separating axis for $t = 1$. However, $\mathbf{v}$ fails to be a separating axis for $t = 2$. A new separating axis $\mathbf{v}'$ is computed using $\mathbf{v}$ as initial axis.

Figure 2 shows the behavior of ISA-GJK for a smoothly moving object. We saw in Section 3 that a support point can be computed in constant time for quadrics and, if coherence is high, in nearly constant time for arbitrary polytopes. Hence, in these cases, ISA-GJK takes nearly constant time per frame.

Experiments show that, in the application of collision detection between convex polyhedra, ISA-GJK is roughly five times faster than Lin-Canny. We did not find significant differences in accuracy for these experiments. Lin-Canny occasionally misses a collision detected by ISA-GJK, although the differences are minimal. The results of these experiments are discussed in Appendix B.

5 Robustness

Numbers represented by a machine have finite precision. Therefore, arithmetic operations will introduce rounding errors. In this section we discuss the implications of rounding errors for the GJK algorithm, and present solutions to problems that might occur as a result of these.

Termination Condition

Let us review the termination condition $\|\mathbf{v}\| - \mu \leq \varepsilon$ of the GJK distance algorithm. We see that for large $\|\mathbf{v}\|$ the rounding error of $\|\mathbf{v}\| - \mu_k$ can be of the same magnitude as ε. This may cause termination problems. We solve this problem by terminating as soon as the relative error, rather than the absolute error, in the computed value of $\|\mathbf{v}\|$ drops below a tolerance value $\varepsilon > 0$. Thus, as termination condition we take $\|\mathbf{v}\| - \mu \leq \varepsilon\|\mathbf{v}\|$.

Moreover, for $\mathbf{v} \approx \mathbf{0}$, we see that the right member of the inequality might underflow and become zero, which in turn will result in termination problems. This problem is solved by terminating as soon as $\|\mathbf{v}\|$ drops below a tolerance ω, where ω is a small positive number.

We would like to add that our experiments have shown that for quadric objects, such as spheres and cones, the average number of iterations used for computing the distance is $O(-\log(\varepsilon))$, i.e., the average number of iterations is roughly linear in the number of accurate digits in the computed distance. For polytopes, the average number of iterations is usually less than for quadrics, regardless of the complexity of the polytopes, and is not a function of ε (for small values of ε).

Backup Procedure

The main source of GJK's numerical problems due to rounding errors is the computation of $\Delta_i(X)$. Each nontrivial $\Delta_i(X)$ is the product of a number of factors of the form $\mathbf{y}_i \cdot \mathbf{y}_k - \mathbf{y}_i \cdot \mathbf{y}_j$. If $\mathbf{y}_k$ is almost equal to $\mathbf{y}_j$ in one of these factors, i.e., X is close to being affinely dependent, then the value of this factor is close to zero, in which case the relative rounding error in the machine representation of this factor may be large due to numerical cancellation. This results in a large relative error in the computed value of $\Delta_i(X)$, causing a number of irregularities in the GJK algorithm.

One of these irregularities was addressed in the original paper [9]. Due to a large relative error, the sign of the computed value of $\Delta_i(X)$ may be incorrect. As a result of this, the subalgorithm will not be able to find a subset X that satisfies the stated criteria. The original GJK uses a backup procedure to compute the best subset. Here, the best subset is the subset X for which each $\Delta_i(X)$ is positive and $v(\text{aff}(X))$ is nearest to the origin.

In our experiments, we observed that, in the degenerate case where the backup procedure needs to be called, the difference between the best vector returned by the backup procedure and the vector $\mathbf{v}_k$ from the previous iteration is negligible.

Hence, considering the high computational cost of executing the backup procedure, we chose to leave it out and return the vector from the previous iteration, after which GJK is forced to terminate. Should the algorithm continue iterating after this event, then it will infinitely loop, since each iteration will result in the same vector being computed.

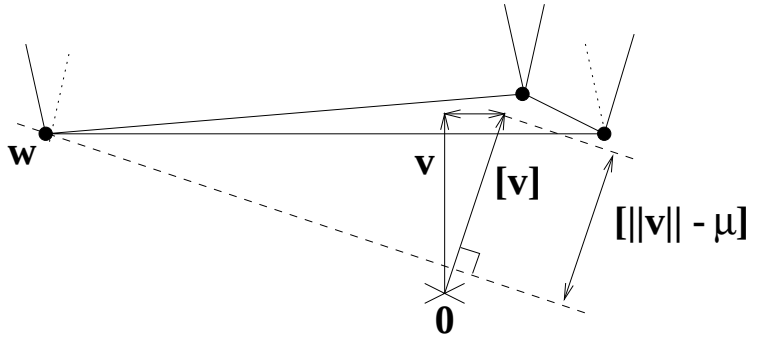
Ill-conditioned Error Bounds

Despite these precautions, the algorithm may still encounter configurations of objects that cause it to loop infinitely, as noted by Nagle [11]. This problem may occur when two polytopes that differ a few orders of magnitude in size are in close proximity of each other. Due to the difference in size, the Minkowski sum of the objects has extremely oblong shaped facets. Let us examine a scenario in which the current simplex conv(W) is an oblong shaped triangle, and $\mathbf{v} = v(A - B)$ is an internal point of the triangle.

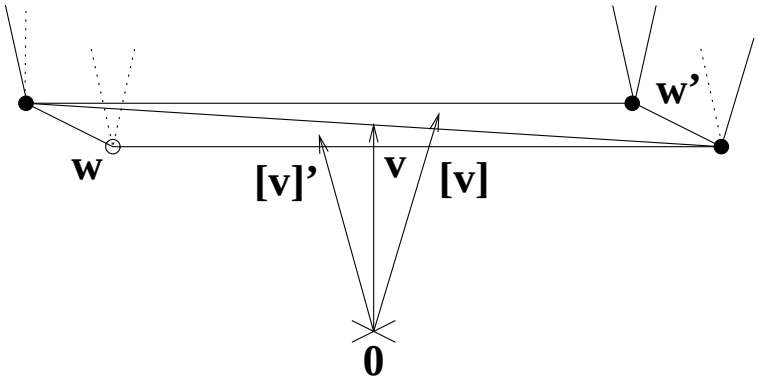
First we note that two of the triangle's vertices lie close to each other. This may cause a large relative rounding error in the computation of $\Delta_i(X)$ for some subsets X. Hence, the computed value $[\mathbf{v}]$ of $\mathbf{v}$ might suffer from this error. Note that the subalgorithm always computes λ_i values that are positive and add up to one. Thus, $[\mathbf{v}]$ is also an interior point of the triangle, yet located at some distance from $\mathbf{v}$. Figure 3(a) depicts the effect of an error in $[\mathbf{v}]$. We see that a small error in $[\mathbf{v}]$ may result in a large error in $[\|\mathbf{v}\| - \mu]$, the computed error bound of $\|\mathbf{v}\|$. The algorithm should terminate at this point since the actual $\|\mathbf{v}\| - \mu$ is zero. However, the error in $[\|\mathbf{v}\| - \mu]$ causes the algorithm to continue iterating. Since the support point $\mathbf{w}$ for $[\mathbf{v}]$ is already a vertex of the current simplex, the algorithm will find the same $[\mathbf{v}]$ in each following iteration, and thus, will never terminate.

Another problem occurs when $v(A-B)$ lies close to the diagonal of an oblong quadrilateral facet in $A - B$, as depicted in Figure 3(b). Again, the large error in $[\|\mathbf{v}\| - \mu]$ causes the algorithm to continue iterating. Only this time, GJK alternately returns the diagonal's opposing vertices $\mathbf{w}$ and $\mathbf{w}'$ as support points. In each iteration, one of the vertices is added to the current simplex and the other is discarded, and vice versa. The remaining two vertices of the current simplex are the vertices of the facet lying on the diagonal. We see that for the simplex containing $\mathbf{w}$, the value $[\mathbf{v}]'$ is computed, which results in the vertex $\mathbf{w}'$ being added to the current simplex. For the simplex containing $\mathbf{w}'$, the value $[\mathbf{v}]$ is computed, which again will cause $\mathbf{w}$ to be added to the current simplex.

Both degenerate cases are tackled in the following way. In each iteration, the support point $\mathbf{w}_k$ is tested whether it is a member of $W_{k-1} \cup \{\mathbf{w}_{k-1}\}$. This can



(a) Same support point in each iteration



(b) Alternating support points

Figure 3: Two problems in the original GJK, resulting from ill-conditioned error bounds.

only be true as a result of one of the degenerate cases. If a degenerate case is detected, then the algorithm terminates and returns $[\mathbf{v}_k]$ as the best approximation of $v(A - B)$ within the precision bounds of the machine.

We have done some extensive bench tests on random input in order to compare the robustness of our enhanced algorithm with the original GJK. The tests showed that with this extra termination condition, our algorithm terminates properly for all tested configurations of polytopes, regardless of the size of the tolerance for the relative error in the computed distance, whereas the original GJK occasionally failed to terminate on the same input.

A Convex Analysis Mini-primer

The *Minkowski sum*[2] of objects A and B is defined as

$$A - B = \{\mathbf{x} - \mathbf{y} : \mathbf{x} \in A, \mathbf{y} \in B\}.$$

Although the Minkowski sum of a pair of objects is a set of vectors, it is regarded as a point set. The space of this point set has the zero vector $\mathbf{0}$ as its origin. It can be shown that if A and B are convex, then $A - B$ is also convex.

A *support mapping* is a function s_C that maps a vector to a point of an object C, according to

$$s_C(\mathbf{v}) \in C \quad \text{and} \quad \mathbf{v} \cdot s_C(\mathbf{v}) = \max\{\mathbf{v} \cdot \mathbf{x} : \mathbf{x} \in C\}.$$

The value of a support mapping for a given vector is called a *support point*. It can be shown that the mapping

$$s_{A-B}(\mathbf{v}) = s_A(\mathbf{v}) - s_B(-\mathbf{v})$$

is a support mapping of $A - B$.

The *affine* and *convex hulls* of a finite point set $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ are the sets of, respectively, *affine* and *convex combinations* of points in X, denoted by

$$\begin{aligned} \text{aff}(X) &= \left\{ \sum_{i=1}^{n} \lambda_i \mathbf{x}_i : \sum_{i=1}^{n} \lambda_i = 1 \right\} \\ \text{conv}(X) &= \left\{ \sum_{i=1}^{n} \lambda_i \mathbf{x}_i : \sum_{i=1}^{n} \lambda_i = 1, \lambda_i \geq 0 \right\}. \end{aligned}$$

[2]Although Minkowski *difference* seems more appropriate, we avoid using this term, since it is defined differently in many geometry texts, namely as $(A^* - B)^*$, i.e., the complement of the Minkowski sum of A's complement and B (shrink one object by the other).

The convex hull of a finite point set is called a *polytope*. A set of points is called *affinely independent* if none of the points can be expressed as an affine combination of the other points. In three-dimensional space, an affinely independent set has at most four points. A *simplex* is the convex hull of an affinely independent set of points. These points are referred to as the *vertices* of the simplex. A simplex of one, two, three, and four vertices is, respectively, a point, a line segment, a triangle, and a tetrahedron.

For $\mathbf{v} \in \mathbb{R}^3 \setminus \{\mathbf{0}\}$ and $\delta \in \mathbb{R}$, the *plane* $H(\mathbf{v}, \delta)$ is a set of points defined by

$$H(\mathbf{v}, \delta) = \{\mathbf{x} \in \mathbb{R}^3 : \mathbf{v} \cdot \mathbf{x} + \delta = 0\}$$

The vector $\mathbf{v}$ is referred to as a *normal* of the plane. For $\|\mathbf{v}\| = 1$, the value of δ is the *signed distance* from $H(\mathbf{v}, \delta)$ to the origin. The positive *halfspace* of a plane $H(\mathbf{v}, \delta)$ is defined as

$$H^+(\mathbf{v}, \delta) = \{\mathbf{x} \in \mathbb{R}^3 : \mathbf{v} \cdot \mathbf{x} + \delta \geq 0\}$$

The negative halfspace $H^-(\mathbf{v}, \delta)$ is defined similarly. Obviously, for nonnegative δ the origin lies in the positive halfspace.

B Empirical Results

In order to compare the performance of ISA-GJK with existing algorithms, we conducted the following experiment. As benchmark we took the multi-body simulation from I-COLLIDE [7]. This is a simulation of a number of polyhedra that move freely inside a cubic space. The number, complexity, density, and translational and rotational velocities of the objects in the space can be varied in order to test the algorithms under different settings. The simulation has a simple type of collision response. It exchanges the translational velocities of each colliding pair of objects, thus simulating a pseudo-elastic reaction. Objects also bounce off the walls of the cubic space in order to constrain them inside the space.

Using this benchmark, we compared the performance of ISA-GJK to Lin-Canny's. For testing Lin-Canny we used I-COLLIDE [6], whereas for testing ISA-GJK, we replaced the Lin-Canny test in I-COLLIDE by a C++ implementation of our ISA-GJK intersection test.

The tests were performed on a Sun UltraSPARC-I (167MHz), compiled using the GNU C/C++ compiler with '-O2' optimization. As default setting we used 20 objects, each having 20 vertices. The default density was set at 5% of the space

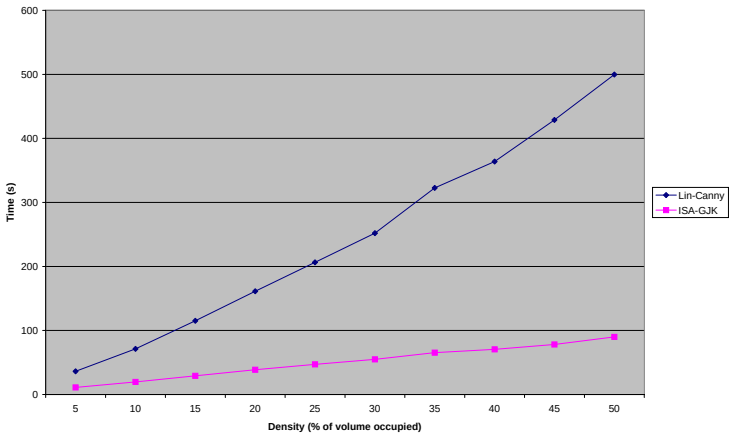


Figure 4: Performance under density variations

being occupied by objects. The translational velocity of an object is expressed in the percentage of its radius the object is displaced in each frame. The default value is 5%. The default rotational velocity is 10 degrees per frame. For each setting, we measured the times for the three algorithms by simulating 50,000 frames.

We experimented with different densities, translational velocities, and rotational velocities. The results of this experiment are shown in Figure 4, 5, and 6.

References

[1] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hull. *ACM Transactions on Mathematical Software*, 22:469–483, 1996.

[2] G. Bell, R. Carey, and C. Marrin. VRML97: The Virtual Reality Modeling Language. http://www.vrml.org/Specifications/VRML97, 1997.

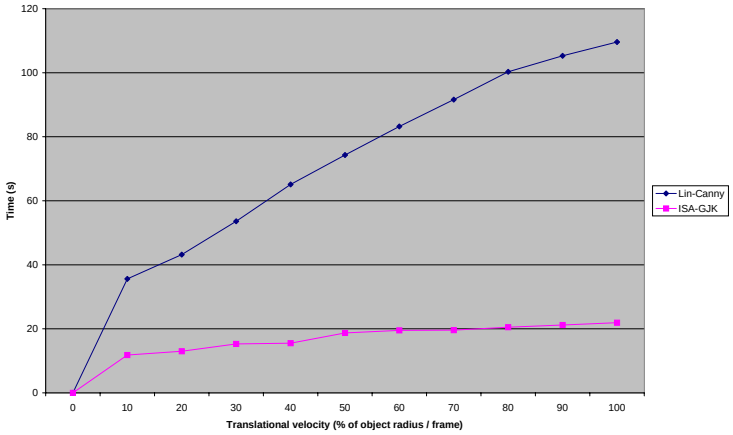


Figure 5: Performance under translational-velocity variations

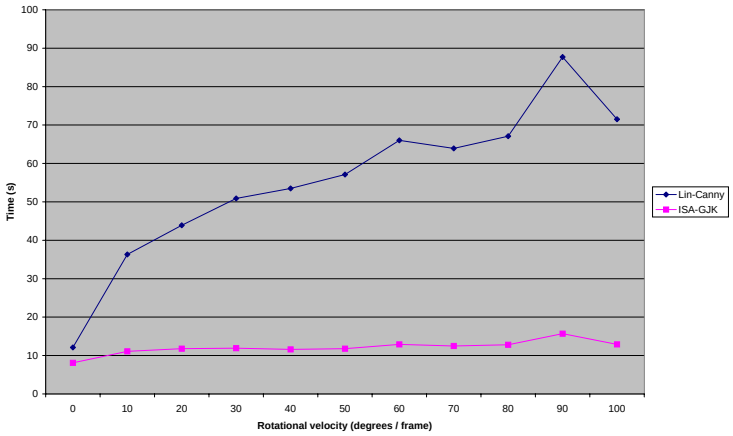


Figure 6: Performance under rotational-velocity variations

[3] S. Cameron. Enhancing GJK: Computing minimum and penetration distances between convex polyhedra. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3112–3117, 1997.

[4] S. A. Cameron and R. K. Culley. Determining the minimum translational distance between convex polyhedra. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 591–596, 1986.

[5] K. Chung and W. Wang. Quick collision detection of polytopes in virtual environments. In *Proc. ACM Symposium on Virtual Reality Software and Technology*, pages 125–131, 1996.

[6] J. Cohen, M. C. Lin, D. Manocha, B. Mirtich, M. K. Ponamgi, and J. Canny. I-COLLIDE: Interactive and exact collision detection library. http://www.cs.unc.edu/˜geom/I_COLLIDE.html, 1996. software library.

[7] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proc. ACM Symposium on Interactive 3D Graphics*, pages 189–196, 1995.

[8] E. G. Gilbert and C.-P. Foo. Computing the distance between general convex objects in three-dimensional space. *IEEE Transactions on Robotics and Automation*, 6(1):53–61, 1990.

[9] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 4(2):193–203, 1988.

[10] M. C. Lin and J. F. Canny. A fast algorithm for incremental distance computation. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 1008–1014, 1991.

[11] J. Nagle. GJK collision detection algorithm wanted. posted on the *comp.graphics.algorithms* newsgroup, Apr. 1998.

[12] C. J. Ong and E. G. Gilbert. The Gilbert-Johnson-Keerthi distance algorithm: A fast version for incremental motions. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 1183–1189, 1997.

[13] R. Rabbitz. Fast collision detection of moving convex polyhedra. In P. Heckbert, editor, *Graphics Gems IV*, pages 83–109. Academic Press, Boston, MA, 1994.