# Proximity Queries and Penetration Depth Computation on 3D Game Objects

Gino van den Bergen

Not a Number
Meerenakkerplein 16
5652 BJ Eindhoven
The Netherlands
gino@acm.org

## Abstract

This paper discusses methods for performing proximity queries (collision detection, distance computation) and penetration depth computation on a large class of convex objects. The penetration depth of a pair of intersecting objects is the shortest vector over which one object needs to be translated in order to bring the pair in touching contact. The class of objects includes convex primitives such as polytopes (line segments, triangles, convex polyhedra) and quadrics (spheres, cones, cylinders), as well as shapes derived from these primitives by affine transformation and spherical expansion (inflating an object by a given offset).

# 1. Introduction

Interactions between 3D game objects usually involve contact or close proximity of the objects. Determining which pairs of objects are in contact or at close proximity is a complex task in many game environments. Additionally, for resolving contacts, a description of the contact area is often required. A useful entity for describing the contact area is *penetration depth*. The penetration depth of a pair of intersecting objects is the shortest vector over which one object needs to be translated in order to bring the pair in touching contact.

In this paper, we focus on methods for performing collision detection, distance computation, and penetration depth computation on convex objects. Several solutions for detecting collisions between convex objects have been offered in the past.

For collision detection of simple polytopes, such as line segments, triangles and boxes, the fastest results are achieved by applying the separating axis theorem [Ebe00]. The *separating axis theorem* states that for a pair of non-intersecting polytopes there exists a separating axis that is either orthogonal to a facet of either polytope, or orthogonal to an edge from each polytope. So, a separating axis can be found simply by testing all facet orientations and all combinations of edge directions to see if one of these yields a separating axis. If none of the axes is a separating axis, then the polytopes must intersect. A nice example of this test is the oriented box test used in OBB trees [GLM96].

For polytopes that have a large number of features, the separating axis theorem is no longer an option, as the number of axes to be tested is quadratic in the number of features. For these cases it is better to switch to an incremental collision detection algorithm that exploits frame coherence. Examples of such algorithms are Lin-Canny [LC91], Chung-Wang [CW96], and GJK-based methods [Cam97,vdB99]. Incremental algorithms have the benefit that, when frame coherence is high, their performance is almost insensitive to the complexity of the objects. Lin-Canny and GJK can be used also for computing the distance between a pair of polytopes.

Methods for computing the penetration depth are less common. An algorithm for determining the penetration depth of a pair of convex polyhedra has been presented in [CC86]. This algorithm has quadratic space and time bounds since it requires an explicit representation of the configuration space obstacle (Minkowski sum). In [Cam97], Cameron presents a faster method, however this method returns an estimate for the magnitude of the penetration depth. The returned penetration depth may point considerably away from the actual penetration depth.

**Contribution**

In this paper, we offer a novel algorithm for determining the exact penetration depth of a pair of convex objects. This algorithm is closely related to the Gilbert-Johnson-Keerthi (GJK) algorithm, so we feel it is appropriate to first present an overview of GJK applied to distance computation and collision detection. The main strength of the methods presented here is the fact that they are applicable to a large class of convex objects. Besides polytopes, the class also contains quadric primitives (spheres, cones, cylinders), as well as shapes derived from these primitives by affine transformation and Minkowski summation. Minkowski sums are useful for representing sphere-swept volumes. Quadrics and sphere-swept volumes are often more convenient for modeling game objects than polytopes, since they require less storage, and allow exact representation of features such as wheels, rods, and rounded edges.

## 2. Convex Objects and Support Mappings

This section defines the class of geometrical objects on which the proximity queries and penetration depth computation are performed. Here, an object is a closed convex point set of arbitrary dimension. The single piece of information regarding the geometry of the objects that is used by the queries is a support mapping. A support mapping of an object $A$ is a function $s_A$ that maps vectors to points, according to

$$s_A(\mathbf{v}) \in A, \quad \mathbf{v} \cdot s_A(\mathbf{v}) = \max\{\mathbf{v} \cdot \mathbf{x} : \mathbf{x} \in A\}$$

The value of a support mapping for a given vector is called a support point. Note that a support mapping of a given object may not be uniquely determined. The choice of support mapping does not matter in the applications that we will encounter. A support mapping fully describes the geometry of a convex object, and can thus be viewed as an implicit representation of the object.

The class of objects we consider is recursively constructed from:

1. Convex primitives, such as:
   a. Polytopes: convex hulls of finite point sets, e.g., line segments, triangles, and convex polyhedra.
   b. Quadrics: spheres, cones, cylinders.
2. Minkowski sums of pairs of convex objects.
3. Affine transformations of convex objects.

For each primitive type we need to supply a support mapping. The support mappings for Minkowski sums and affine transformations can be derived from the support mappings of their child objects. We will discuss how to do this further on.

For quadric objects, a support point can be computed in constant time [CF90, vdB99]. It can be seen that for polytopes, there exists a support mapping that returns vertices only. For a given vector, the corresponding support point is the vertex for which the dot product with this vector is the greatest. So, without preprocessing, the computation of a support point for a polytope takes an amount of time that is linear in the number of vertices. It is shown in [CW96] that a support point can be found in $O(\log n)$ time for two- and three-dimensional polytopes represented using the hierarchical representation by Dobkin and Kirkpatrick [DK90].

In interactive applications, there is usually a lot of frame coherence. It has been mentioned in a number of publications [CC86,CW96,Cam97,OG97] 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 for 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*. The adjacency graph of polytope vertices can be obtained by computing the convex hull, for instance, using Quickhull [BDH96].

Recently, a hybrid technique has been presented that allows hill climbing to be performed on a hierarchical polytope representation, thus maintaining a worst-case $O(\log n)$ time bound, while allowing near constant computation cost when frame coherence is high [GHZ99,EL00].

**Minkowski Sum**

The Minkowski sum of objects $A$ and $B$ is defined as

$$A + B = \{\mathbf{a} + \mathbf{b} : \mathbf{a} \in A, \mathbf{b} \in B\}.$$

It is not very hard to show that if $A$ and $B$ are convex, then $A + B$ is also convex. Furthermore, it can be shown that the mapping

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

is a proper support mapping for $A + B$.

Minkowski sums are useful for representing swept volumes. Most commonly used are sphere-swept volumes. A sphere-swept volume is the result of adding a sphere to an arbitrary convex object. Figure 1 shows the sphere-swept volume resulting from adding a sphere to a box.

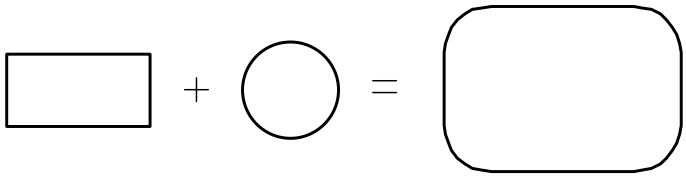


Figure 1: The Minkowski sum of a box and a sphere.

Sphere-swept volumes are typically useful for estimating the contact plane in physics-based simulations. For this purpose, a skin-bone technique is applied. Collision detection is performed on sphere-swept objects. When a collision is found, the motion integration time step is subdivided until a configuration is attained in which the sphere-swept objects (the skins) overlap, but the plain objects (the bones) do not overlap. For such a configuration, the closest points pair of the plain objects can be used as an approximation of the contact plane.

Further on, we will discuss the computation of penetration depth, which is better suited for approximating the contact plane in interactive applications, since it does not require the time step to be subdivided.

## Affine Transformation

The position, orientation, and non-uniform scaling of game object are represented by affine transformations. For an affine transformation $T(\mathbf{x}) = \mathbf{B}\mathbf{x} + \mathbf{c}$, the matrix $\mathbf{B}$ represents the orientation and scaling of the object's local coordinate system, and the vector $\mathbf{c}$ represents the position of the object's local origin.

Given $s_A$, a support mapping of object $A$, the function

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

is a support mapping for $T(A)$, the image of $A$ under $T$ [vdB99]. Here, $\mathbf{B}^{\mathrm{T}}$ denotes the transpose of $\mathbf{B}$. Note that $\mathbf{B}^{\mathrm{T}}\mathbf{v} = (\mathbf{v}^{\mathrm{T}}\mathbf{B})^{\mathrm{T}}$ is simply $\mathbf{v}$ left-multiplied with $\mathbf{B}$.

## Bounding Boxes

A lot of techniques for speeding up collision detection of 3D objects make use of axis-aligned bounding boxes (AABBs). One example is an incremental sorting technique, often referred to as *Sweep and Prune,* that maintains a list of pairs of

overlapping bounding boxes [Bar92,CLMP95]. Another example is the AABB tree, a bounding box hierarchy that is used for speeding up collision detection between complex shapes [vdB97].

The axis-aligned bounding box of a convex object can be computed straightforwardly by using a support mapping. Let $\mathbf{e}_i, i = 1,2,3$ be the axes of the coordinate system to which the bounding box is aligned. Then, the projection of the object $A$ onto $\mathbf{e}_i$ is given by the interval

$$[\mathbf{e}_i \cdot s_A(-\mathbf{e}_i), \mathbf{e}_i \cdot s_A(\mathbf{e}_i)].$$

The bounding box is the Cartesian product of the intervals on the three coordinate axes.

For an object $A$ whose placement is given by an affine transformation $T(\mathbf{x}) = \mathbf{B}\mathbf{x} + \mathbf{c}$, the computation of the bounding box of $T(A)$ with respect to the standard (world) coordinate system can be further optimized. The projection of $T(A)$ on $\mathbf{e}_i$ is

$$[\mathbf{e}_i \cdot T(s_A(-\mathbf{B}^{\mathrm{T}}\mathbf{e}_i)), \mathbf{e}_i \cdot T(s_A(\mathbf{B}^{\mathrm{T}}\mathbf{e}_i))].$$

For the standard coordinate system, we have $\mathbf{e}_1 = (1,0,0)$, $\mathbf{e}_2 = (0,1,0)$, $\mathbf{e}_3 = (0,0,1)$. We see that the vector $\mathbf{B}^{\mathrm{T}}\mathbf{e}_i = (\mathbf{e}_i^{\mathrm{T}}\mathbf{B})^{\mathrm{T}}$ is simply the $i^{\text{th}}$ row of $\mathbf{B}$. Furthermore, $\mathbf{e}_i \cdot T(\mathbf{x}) = \mathbf{e}_i \cdot \mathbf{B}\mathbf{x} + \mathbf{e}_i \cdot \mathbf{c}$ is equal to $\mathbf{b}_i \cdot \mathbf{x} + c_i$, where $\mathbf{b}_i$ is the $i^{\text{th}}$ row of $\mathbf{B}$, and $c_i$ is the $i^{\text{th}}$ component of $\mathbf{c}$. We can reduce the projection of $T(A)$ onto $\mathbf{e}_i$ to

$$[\mathbf{b}_i \cdot s_A(-\mathbf{b}_i) + c_i, \mathbf{b}_i \cdot s_A(\mathbf{b}_i) + c_{i,}].$$

For Minkowski sums of convex objects we have the following useful property. The AABB of $A + B$ is equal to the Minkowski sum of the AABBs of $A$ and $B$. This can be verified by observing that for intervals $[b_1, e_1]$ and $[b_2, e_2]$, we have

$$[b_1, e_1] + [b_2, e_2] = [b_1 + b_2, e_1 + e_2].$$

Practical use of this property is found in collision detection of sphere-swept objects. For instance, we have a rigid object represented by a complex shape such as a triangle mesh. For the shape we compute an AABB tree as a preprocessing step. For some tasks it is necessary to add a sphere to the shape, in order to ‘fatten’ the object, while others use the plain object. Instead of maintaining two hierarchies, one for the plain and one for the sphere-swept object, we add the sphere during the traversal of the AABB tree. For each visited node during the traversal we compute the Minkowski sum of the node’s AABB

and the AABB of the sphere, and use the resulting AABB for the rejection test. This technique is particularly useful in the skin-bone technique for estimating the contact plane.

## Configuration Space

The collision detection, distance computation, and penetration depth computation problems are expressed in terms of the *configuration space obstacle* of the query objects. For this purpose we introduce a negation operation to the Minkowski sum:

$$-B = \{-\mathbf{b} : \mathbf{b} \in \mathbf{B}\}.$$

It can be seen that the mapping

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

is indeed a support mapping for $-B$.

The *configuration space obstacle* (CSO) of objects $A$ and $B$ is the object $A + (-B)$, which we will abbreviate to $A$ - $B$. The mentioned queries are expressed in terms of $A$ - $B$ in the following way. The collision detection problem is expressed as

$$A \cap B \neq \varnothing \Leftrightarrow \mathbf{0} \in A - B.$$

Here, $\mathbf{0}$ denotes the zero vector or origin of the configuration space. The distance $d(A,B)$ between $A$ and $B$ can be expressed as follows

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

Similarly, the magnitude of the penetration depth $p(A,B)$ of $A$ and $B$ can be expressed as

$$p(A,B) = \inf\{\|\mathbf{x}\| : \mathbf{x} \notin A - B\}.$$

We use *infimum*, i.e., greatest lower bound, rather than *minimum* since $A - B$ is a closed set. It can be seen that for intersecting objects the penetration depth must be a point on the boundary of $A - B$. Figure 2 illustrates the relation between the objects and their CSO.

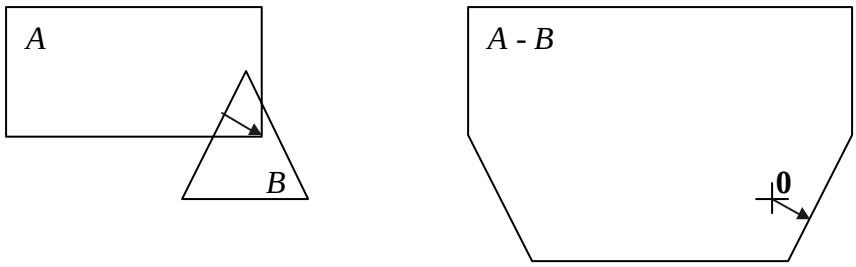



Figure 2: A pair of intersecting objects and their corresponding CSO. The arrow represents the penetration depth.

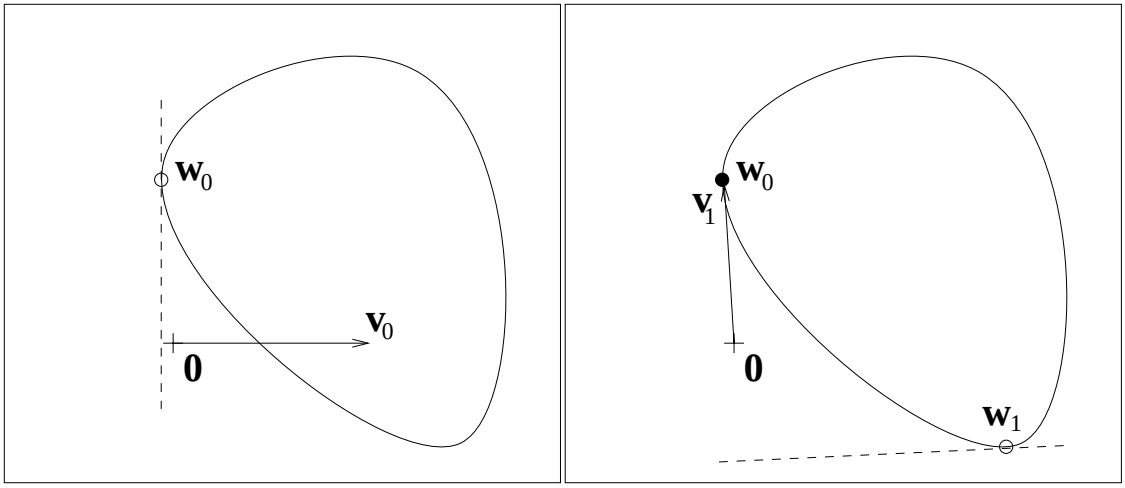
## 3. The Gilbert-Johnson-Keerthi Algorithm

This section presents an overview of the Gilbert-Johnson-Keerthi (GJK) algorithm [GJK88,GF90]. The GJK algorithm is an iterative method for computing the distance between a pair of convex objects. A more in-depth discussion as well as details on how to improve the performance and robustness of GJK can be found in [vdB99].

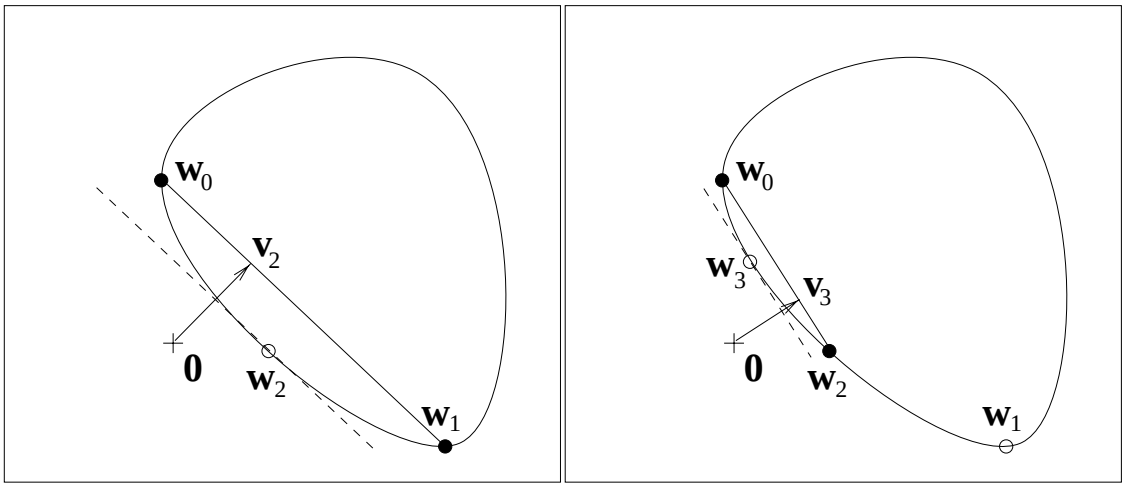
GJK is essentially a descent method for finding the point in the CSO closest to the origin. In each iteration, a simplex is constructed that is contained in the CSO and lies closer to the origin than the simplex constructed in the previous iteration. The algorithm terminates as soon as the estimated error in the computed distance is less than a given tolerance.

We define $W_k$ as the set of vertices of the simplex constructed in the $k^{\text{th}}$ iteration, and $\mathbf{v}_k$ as the point in the simplex closest to the origin. Initially, we have $W_0 = \varnothing$, and $\mathbf{v}_0$, an arbitrary point in $A$ - $B$. Since each $\mathbf{v}_k$ is contained in the CSO, the length of $\mathbf{v}_k$ must be an upper bound for the distance.

GJK generates a sequence of simplices in the following way. In each iteration step, a vertex $\mathbf{w}_k = s_{A-B}(-\mathbf{v}_k)$ is added to the simplex. The new $\mathbf{v}_{k+1}$ is the point in the convex hull of $W_k \cup \{\mathbf{w}_k\}$ closest to the origin. For $W_{k+1}$, we take the smallest sub-simplex of $W_k \cup \{\mathbf{w}_k\}$ that contains $\mathbf{v}_{k+1}$. The point $\mathbf{v}_{k+1}$ and the simplex $W_{k+1}$ are computed using Johnson's algorithm [GJK88,vdB99]. Figure 3 shows a sequence of iterations of the GJK algorithm in 2D.




(a) $k = 0, W_0 = \varnothing$ (b) $k = 1, W_1 = \{\mathbf{w}_0\}$




(c) $k = 2, W_2 = \{\mathbf{w}_0, \mathbf{w}_1\}$ (d) $k = 3, W_3 = \{\mathbf{w}_0, \mathbf{w}_2\}$

Figure 3: A sequence of iteratons of the GJK algorithm. The dashed lines represent the support planes. The continuous lines connecting the black dots represent the current simplex.

The error in the length of $\mathbf{v}_k$ can be estimated if we also maintain a lower bound for the distance. As lower bound we use

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

However, since $\delta_k$ is not necessarily monotonic, we use the maximum $\delta_k$ over all iterations, and start with zero as initial lower bound.

The computation of $\|\mathbf{v}_k\|$ involves the evaluation of a square root, which is not very good for the performance. The lower and upper bound are therefore better expressed in terms of squared distance. Note however that only for non-negative $\delta_k$, we have

$$(\max\{\delta_k\})^2 = \max\{\delta_k{}^2\} = \max\{(\mathbf{v}_k \cdot \mathbf{w}_k)^2 / \|\mathbf{v}_k\|^2\},$$

so for the case $\mathbf{v}_k \cdot \mathbf{w}_k < 0$, the squared lower bound should not be updated.

**Collision Detection**

For deciding whether or not two objects intersect, we do not need to know the distance between them. We merely need to know whether the distance is equal to zero or not. So, as soon as the lower bound for the distance becomes positive, the algorithm may terminate returning a non-intersection. 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 fewer iterations for finding a separating axis of a pair of non-intersecting objects than for computing the distance. For instance in Figure 3, the vector $\mathbf{v}_k$ is a separating axis for the first time when $k = 2$. For intersecting objects, the algorithm terminates on $\mathbf{v}_k = \mathbf{0}$, i.e., a simplex containing the origin is found.

The collision detection version of GJK is particularly useful when there is a lot of frame coherence. Similar to distance tracking algorithms, such as Lin-Canny [LC91], and Cameron's Enhanced GJK algorithm [Cam97], an incremental version of the GJK separating-axis algorithm shows almost constant time complexity per frame for convex objects of arbitrary complexity, when 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. When the degree of coherence between frames is high, 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.

## 4. Penetration Depth

In this section, we present an iterative method for computing the penetration depth of a pair of intersecting objects. This method is closely related to GJK. It also uses only support mappings as geometric representation of the objects, and is therefore applicable to the same class of objects as GJK. Moreover, the method uses the output of GJK, i.e., a simplex containing the origin, as input.

We saw that the penetration depth of a pair of intersecting objects $A$ and $B$, is a point on the boundary of $A - B$ closest to the origin. We say "a point" since the penetration depth is not necessarily unique. The basic strategy to finding such a point is to start with a polytope that contains the origin and has vertices that lie on the boundary, and 'blow it up' by adding vertices that lie on the boundary. In each iteration step, we select the facet of the polytope closest to the origin and subdivide it using support points as additional vertices. Note that for non-zero $\mathbf{v}$, the support point $s_{A-B}(\mathbf{v})$ is indeed a point on the boundary[1].

For reasons of clarity, we will first explain the algorithm in 2D, and then generalize it to 3D. In the 2D version, we blow up a convex polygon by splitting the edges. We start off with a simple polygon, such as a triangle, that contains the origin and has its vertices on the boundary of the CSO. For each edge of the polygon we compute the point $\mathbf{v}$ on the affine hull of the edge (the line through the edge's vertices) that lies closest to the origin. The length of the shortest $\mathbf{v}$ is a lower bound for the magnitude of the penetration depth, since the polygon is contained in the CSO. The edge with the shortest $\mathbf{v}$ is going to be split.
Note that since the polygon is convex, the shortest $\mathbf{v}$ must be an internal point of this edge. Figure 4 illustrates this property.

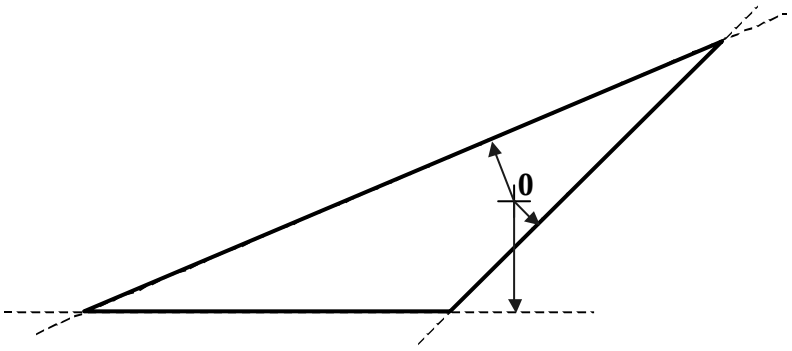
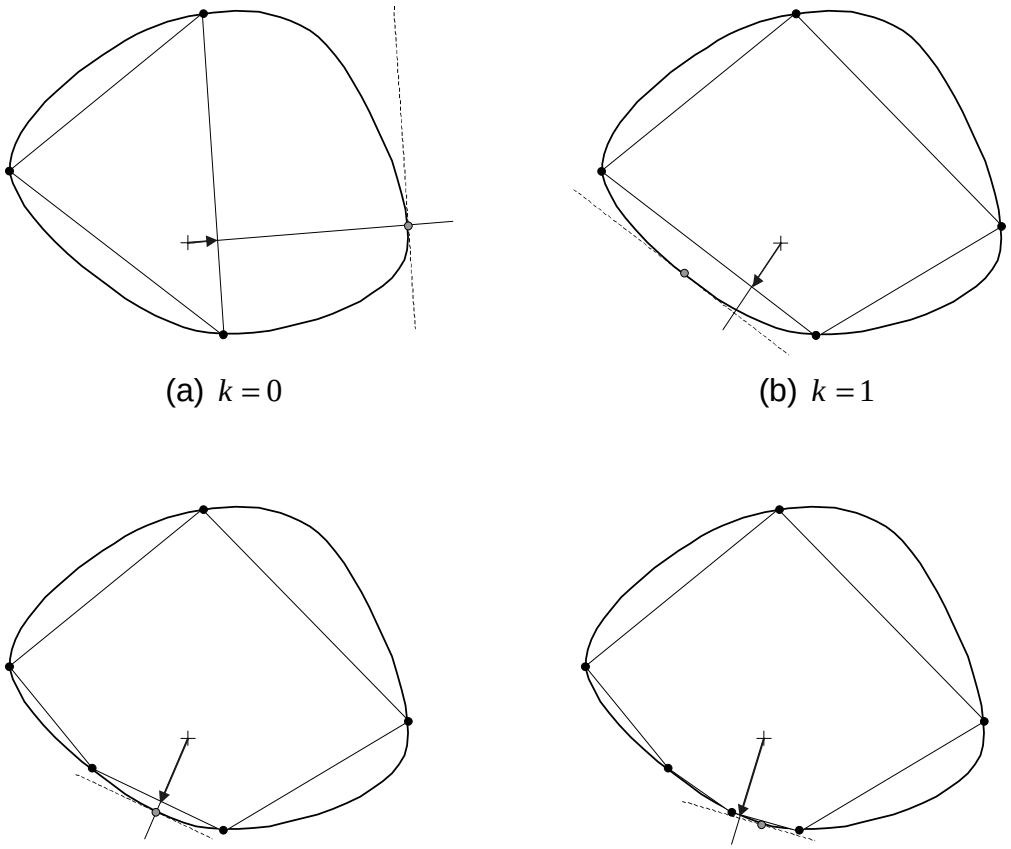



Figure 4: An arrow denotes the point on the line through an edge, that lies closest to the origin. The shortest arrow always points to an internal point of the corresponding edge.

[1] For the zero vector this does not have to be the case, however, the requirement that a support point should always lie on the boundary can easily be satisfied.

In each iteration step, the edge with the shortest $\mathbf{v}$ is split by inserting the support point $\mathbf{w} = s_{A-B}(\mathbf{v})$ as new vertex. For the two new edges we compute the point on the affine hull closest to the origin, and repeat this procedure until the shortest $\mathbf{v}$ lies sufficiently close to the penetration depth. Figure 5 shows a sequence of iterations of our method, which we will refer to as the *expanding polytope algorithm*.



(a) $k = 0$ (b) $k = 1$

(c) $k = 2$ (d) $k = 3$

Figure 5: A sequence of iterations of the *expanding polytope algorithm*. An arrow denotes the shortest vector $\mathbf{v}$. An open dot denotes the new vertex $\mathbf{w}$. A dashed line represents the support plane of the vector $\mathbf{v}$.

The algorithm terminates as soon as the error in $\mathbf{v}$ drops below a given tolerance. As upper bound for the magnitude of the penetration depth we take the distance from the support plane through $\mathbf{w}$ to the origin, which is $\mathbf{v} \cdot \mathbf{w} / \|\mathbf{v}\|$. Again, we save

ourselves the evaluation of square roots by squaring the distance measures. We store the edges of the polygon together with their corresponding $\mathbf{v}$ vectors in a priority queue, using the squared length of the vector as key value (shortest length first). Note that edges whose $\mathbf{v}$ is not an internal point of the edge, will never be the closest edge of a convex polygon, and can therefore be left out of the priority queue. Priority queues can be implemented efficiently in C++ by using the binary heap operations from the *Standard Template Library*.

We observe that when the origin lies close to the center of the CSO, the algorithm 'pokes around' wildly before converging to the penetration depth. In the extreme case where the CSO is circular and centered at the origin, the algorithm will converge extremely slowly, since the penetration depth has an infinite number of solutions. These cases are better avoided or dealt with in a different way. Examples of such cases are pairs of concentric spheres or cylinders.

We are now ready to tackle penetration depth computation in 3D. The difference with the 2D algorithm is that we need to inflate a convex polyhedron in 3D, and thus have to subdivide triangles rather than line segments. The naïve way of splitting a triangle would be to add a single vertex as depicted in Figure 6.

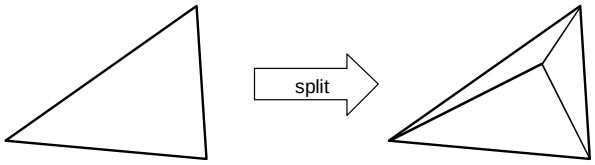



Figure 6: A naïve way to split a triangle.

This is not a very good solution for two reasons. Firstly, as we proceed in splitting the triangles, the resulting triangles will become gradually more oblong. For oblong triangles, the computation of the closest point suffers more from numerical problems due to round-off errors in floating point arithmetic. Secondly, since the edges of the triangle are never broken, the algorithm will have a hard time approaching the surface when the penetration depth is located near an edge of the initial polytope.

We also need to split the edges of the triangle in order to get triangle fragments that lie closer to the boundary. For this purpose, we compute auxiliary support points for the edges. First, we compute for each edge $e$, the point $\mathbf{v}_e$ on the edge closest to the origin. Next, we split the edge by inserting a new vertex $\mathbf{w}_e = s_{A-B}(\mathbf{v}_e)$. A triangle is thus split into six fragments as depicted in Figure 7.

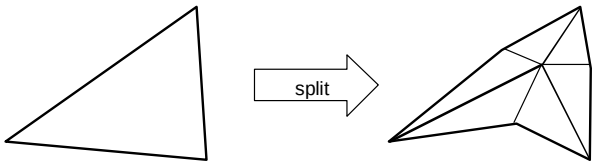



Figure 7: A better way to split a triangle.

However, edges that are part of the boundary of the CSO need not be split. This case may occur when the CSO has flat faces, for instance, if either or one of the query objects is a polytope. An edge is part of the boundary if the point $\mathbf{v}_e$ lies in the corresponding support plane through $\mathbf{w}_e$. This is the case iff $\mathbf{v}_e \cdot \mathbf{w}_e = \|\mathbf{v}_e\|^2$, as can easily be verified. So the ultimate solution is to split only the edges of the triangle that are not part of the boundary, as depicted in Figure 8.

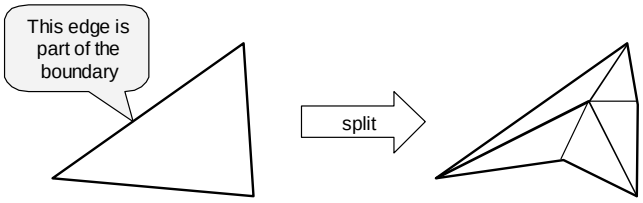



Figure 8: The ultimate solution. Edges that are part of the boundary are not split.

The point $\mathbf{v}$ on the affine hull of a simplex closest to the origin is computed in the following way. Let $\{\mathbf{w}_1,\ldots,\mathbf{w}_n\}$ be the set of vertices of the simplex. We express the point $\mathbf{v}$ as an *affine combination* of $\{\mathbf{w}_i\}$, i.e., $\mathbf{v} = \lambda_1\mathbf{w}_1 + \cdots + \lambda_n\mathbf{w}_n$, for $\lambda_1 + \cdots + \lambda_n = 1$. The vector $\mathbf{v}$ points to the closest point iff it is orthogonal to the affine hull, i.e., $\mathbf{v} \cdot (\mathbf{w}_i - \mathbf{w}_1) = 0$, for $i = 2,\ldots,n$. This results in solving a linear system of equations $\mathbf{A}(\lambda_i)^{\mathrm{T}} = \mathbf{b}$, where

$$\mathbf{A} = \begin{pmatrix} 1 & \cdots & 1 \\ \mathbf{w}_1 \cdot (\mathbf{w}_2 - \mathbf{w}_1) & \cdots & \mathbf{w}_n \cdot (\mathbf{w}_2 - \mathbf{w}_1) \\ \cdots & \cdots & \cdots \\ \mathbf{w}_1 \cdot (\mathbf{w}_n - \mathbf{w}_1) & \cdots & \mathbf{w}_n \cdot (\mathbf{w}_n - \mathbf{w}_1) \end{pmatrix}, \text{ and } \mathbf{b} = \begin{pmatrix} 1 \\ 0 \\ \cdots \\ 0 \end{pmatrix}.$$

This system of equations is easily solved by computing $\mathbf{A}^{-1}$, the inverse of $\mathbf{A}$, since $(\lambda_i)^{\mathrm{T}} = \mathbf{A}^{-1}\mathbf{b}$. The point $\mathbf{v}$ is an internal point of the simplex iff for all $i = 1, \ldots, n$, the value of $\lambda_i$ is positive.

As mentioned earlier, we start off the algorithm with a polytope that contains the origin and has its vertices on the boundary of the CSO. We have seen that for intersecting objects, GJK returns a simplex that contains the origin. This simplex will in most cases be a tetrahedron, which is just what we need for the expanding polytope algorithm. However, in some cases, GJK may also return a line segment or a triangle as simplex. We may deal with these degenerate cases in two ways. We either add support points to the simplex, thus constructing a proper polytope, or we recognize a special case and solve it using a dedicated penetration depth computation method.

In the case where the simplex returned by GJK is a triangle, we can simply add the support points $s_{A-B}(\mathbf{n})$ and $s_{A-B}(-\mathbf{n})$, where $\mathbf{n}$ is a normal to the triangle. In this way, we construct a hexahedron (two tetrahedra glued together) containing the origin. In the case where the simplex is a line segment, we are dealing most likely with a pair of intersecting spheres. Other configurations of intersecting objects that may result in GJK returning a line segment are cubes or cylinders that are aligned along their diagonals. In all cases where GJK returns a line segment, we simply return the vertex of the line segment that lies closest to the origin, which will be the correct penetration depth in most of these degenerate cases.

The penetration depth can be used as an approximation of the contact plane's normal. The expanding polytope algorithm can be tailored to return also the contact points. At termination, we have a description of the penetration depth as $\mathbf{v} = \lambda_1\mathbf{w}_1 + \cdots + \lambda_n\mathbf{w}_n$, where $\lambda_1 + \cdots + \lambda_n = 1$ and $\lambda_i > 0$. Each vertex is computed as $\mathbf{w}_i = \mathbf{p}_i - \mathbf{q}_i$, where $\mathbf{p}_i$ and $\mathbf{q}_i$ are support points of respectively $A$ and $B$. Let $\mathbf{a} = \lambda_1\mathbf{p}_1 + \cdots + \lambda_n\mathbf{p}_n$ and $\mathbf{b} = \lambda_1\mathbf{q}_1 + \cdots + \lambda_n\mathbf{q}_n$. Since $A$ and $B$ are convex, it is clear that $\mathbf{a} \in A$ and $\mathbf{b} \in B$. Furthermore, since $\mathbf{a} - \mathbf{b} = \mathbf{v}$, the points $\mathbf{a}$ and $\mathbf{b}$ are proper contact points.

## 5. Conclusion

In this paper, we have discussed the Gilbert-Johnson-Keerthi algorithm, an iterative method for computing the distance and detecting collision between arbitrary convex objects, and offered a novel algorithm, referred to as the *expanding polytope algorithm*, for computing the penetration depth of a pair of convex objects. The main strength of these methods is the fact that they are applicable to a large class of convex objects, including polytopes, quadrics, as well as shapes derived from these primitives by affine transformation and Minkowski summation.

A C++ implementation of GJK for collision detection and distance computation is released as part of SOLID 2.0. Also, a standalone demo package, called *GJK-engine*, has been made available. The complete C++ source code for both packages can be downloaded from http://www.win.tue.nl/~gino/solid/.

Experiments have shown that the performance of these methods warrants their use at interactive rates. Most notably, ISA-GJK, the collision detection version of GJK, is one of the fastest collision detection methods for convex objects currently available.

## Reference


[Bar92] David Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD thesis, Computer Science Department, Cornell University, 1992. Technical Report 92-1275.

[BDH96] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hull. *ACM Transactions on Mathematical Software*, 22:469-483, 1996.

[Cam97] 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.

[CC86] 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.

[CLMP95] 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.

[CW96] 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.

[DK90] D. P. Dobkin and D. G. Kirkpatrick. Determining the separation of preprocessed polyhedra - a unified approach. In *Proc. 17th Int. Coll. Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 400-413. Springer-Verlag, 1990.

[Ebe00] David H. Eberly. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[EL00] S. A. Ehmann and M. C. Lin. Accelerated proximity queries between convex polyhedra by multi-level voronoi marching. In *Proc. Int. Conf. on Intelligent Robots and Systems*, 2000.

[GF90] 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.

[GHZ99] L. J. Guibas, D. Hsu, and L. Zhang. H-walk: Hierarchical distance computation for moving convex bodies. In *Proc. 15th Annual ACM Symposium on Computational Geometry*, pages 265-273, 1999.

[GJK88] 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.

[GLM96] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: a hierarchical structure for rapid interference detection. In *proc. SIGGRAPH '96*, pages 171-180, 1996.

[LC91] 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.

[OG97] 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.

[vdB97] Gino van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools*, 2(4):1-14, 1997.

[vdB99] Gino van den Bergen. A fast and robust GJK implementation for collision detection of convex objects. *Journal of Graphics Tools*, 4(2):7-25, 1999.