Computational Geometry 15 (2000) 51–68

Computational Geometry
Theory and Applications

www. elsevier.nl/locate/comgeo

A hierarchical method for real-time distance computation among moving convex bodies [☆]

Leonidas J. Guibas, David Hsu [1], Li Zhang [*]

Department of Computer Science, Stanford University, Stanford, CA 94305, USA

Abstract

This paper presents the *Hierarchical Walk*, or *H-Walk* algorithm, which maintains the distance between two moving convex bodies by exploiting both motion coherence and hierarchical representations. For convex polygons, we prove that *H-Walk* improves on the classic Lin–Canny and Dobkin–Kirkpatrick algorithms. We have implemented *H-Walk* for moving convex polyhedra in three dimensions. Experimental results indicate that, unlike previous incremental distance computation algorithms, *H-Walk* adapts well to variable coherence in the motion and provides consistent performance. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Distance computation; Collision detection; Moving objects; Hierarchical methods; Temporal coherence

1. Introduction

Distance computation is an important problem in many applications, for example, virtual environment simulation, computer animation, and robot motion planning. Distance information can be used for collision detection, which is crucial in creating realistic virtual environments, as collision leads to either incorrect behavior or visual artifacts. In motion planning, collision causes damage to robots. Distance information can also be used to guide sampling in randomized motion planning algorithms.

The distance computation problem has been studied extensively in the literature (see, for example, [2,4,5,7,8,10,12,14]). In particular, computing the distance between convex objects has received much attention because it serves as a fundamental building block in many collision detection packages. Two techniques are most commonly employed to obtain fast distance computation between two moving convex objects: hierarchical data structures and incremental distance computation by exploiting the

☆ This research is partially supported by National Science Foundation grant CCR-9623851, U.S. Army Research Office MURI grant DAAH04-96-1-0007, and a grant from the U.S.–Israeli Binational Science Foundation.

* Corresponding author.

E-mail address: lizhang@cs.stanford.edu (L. Zhang).

1 Partially supported by a Microsoft Graduate Fellowship.

0925-7721/00/$ – see front matter © 2000 Elsevier Science B.V. All rights reserved.
PII: S0925-7721(99)00042-5

coherence of motion. Previous work on this problem usually applies one of the two techniques; the former performs better for complex objects exhibiting low level of coherence, while the latter performs better for simple objects exhibiting high level of coherence. However, in real-time collision detection applications, what is required is consistently good performance across different levels of coherence. In this paper, we show that the two techniques can be effectively combined to achieve this goal.

Early work on distance computation typically considers the distance between a pair of static convex objects. Dobkin and Kirkpatrick give a linear-time algorithm to determine the distance between two convex polyhedra [3]. The algorithm of Gilbert, Johnson and Keerthi (*GJK*), which is well known in the robotics community, treats a polyhedron as the convex hull of a set of points and performs operations on the simplices defined by subsets of these points [7]. It works well for simple objects.

If there are many distance queries for the same pair of objects, one may wish to preprocess the objects in order to improve the query time. Edelsbrunner gives an algorithm that answers distance queries between two convex polygons in logarithmic time [5]. Dobkin and Kirkpatrick propose an algorithm (*Dobkin–Kirkpatrick*) that preprocesses two convex polyhedra in linear time so that for any translation and rotation of the polyhedra, the distance between them can be obtained in poly-logarithmic time [4]. The key data structure they use is a balanced inner hierarchical representation of a convex polyhedron known as the Dobkin–Kirkpatrick hierarchy.

In practice, the objects that we deal with are often in continuous motion. A standard way to handle this situation is to discretize time, and at each time step, compute the distance between the objects according to their current position and orientation. If the time step is small enough, the closest pair of features (vertices, edges or faces) between two polyhedra will not move very far from one time step to the next. This coherence, together with convexity, motivates the approach of tracking the closest pair of features instead of computing it from scratch at every time step. Lin and Canny proposed the first algorithm (*Lin–Canny*) that exploits coherence [12]. Their algorithm starts from the closest pair computed in the last time step, and “walks” on the surface of polyhedra until reaching the new closest pair. Convexity guarantees that we can determine locally whether a pair of features is the closest pair, and if not, a neighboring pair that is closer. Empirically the query time for *Lin–Canny* is nearly constant provided that the coherence is high. Later on, Cameron proposed an enhanced version of *GJK* [1], which possesses properties similar to those of *Lin–Canny*. More recently, Mirtich formulated the *V-Clip* algorithm, which is based on *Lin–Canny*, but more efficient and robust in practice [13].

Lin–Canny and the other algorithms that exploit the coherence between successive queries are simple to implement and perform very well when coherence is high. Ideally, if c is the minimum number of features that we need to traverse in order to go from the previous to the current closest pair, one can hope that *Lin–Canny* will work in time $\mathrm{O}(c)$. However, the performance of *Lin–Canny* depends critically on the level of coherence, which in turn depends on the combinatorial complexity, the shape, and the motion of objects. When the coherence becomes low, its performance quickly deteriorates. Actually one can show that in the worst case, if there are a total of n features on the polyhedra in question, *Lin–Canny* may have to walk $\Omega(n^2)$ steps before reaching the closest pair. In contrast to *Lin–Canny*, *Dobkin–Kirkpatrick* has guaranteed poly-logarithmic bound, but it does not exploit coherence and is likely to be slower than *Lin–Canny* when coherence is high. In addition, *Dobkin–Kirkpatrick* is more difficult to implement because of the subtle geometric computation involved.

This paper presents the *Hierarchical Walk*, or *H-Walk*, algorithm, which applies *Lin–Canny* type of walking on hierarchically represented polyhedra to improve on the classic Lin–Canny and Dobkin–Kirkpatrick algorithms. *Lin–Canny* always walks on the surface of polyhedra and therefore may walk a long distance before reaching the closest pair. The problem can be fixed by allowing the walk to go inside the polyhedra and take shortcuts to achieve better performance. To be able to do so, we need to introduce new features, in addition to those already on the surface. The inner hierarchy defined by Dobkin and Kirkpatrick provides a good set of features for our purpose. We can think of the new algorithm as walking in the Dobkin–Kirkpatrick hierarchy instead of on the surfaces of a polyhedron. Intuitively, this new walking strategy should be more efficient, especially when objects are complex and the coherence is low. Our experimental results indicate that it is indeed the case. In fact, we can prove that in two dimensions, *H-Walk* runs in $O(\log c)$ time, an improvement over *Lin–Canny*'s $O(c)$ and *Dobkin–Kirkpatrick*'s $O(\log n)$, but we are unable to prove the same theoretical bound in three dimensions. Furthermore, *H-Walk* offers consistent performance across different levels of coherence, a desirable property in many real-time applications. Like *Lin–Canny*, *H-Walk* is very simple to implement. After building the Dobkin–Kirkpatrick hierarchy the only primitive operation that it performs is to walk from one feature to an adjacent feature.

The rest of the paper is organized as follows. Section 2 describes the basic ideas for incremental distance computation and data structures needed for implementing *H-Walk*. Section 3 presents our algorithms in two dimensions. Section 4 presents our algorithms in three dimensions, the experimental results. Finally, Section 5 summarizes the results and points out direction for future research.

2. Preliminaries

2.1. Definitions and notations

Let P be a convex polyhedron in $\mathbb{R}^d$ for $d = 2$ or 3. A feature of P is a vertex, an edge or a face (if $d = 3$) on the boundary of P. We assume that edges and faces are closed sets. A closest pair of features between two convex polyhedra P and Q is a pair of features that contain a closest pair of points, i.e., a pair of points that achieve the minimum Euclidean distance between P and Q. Notice that the closest pairs may not be unique. The strategy adopted by most distance computation algorithms is to locate the closest pair of features.

Since edges and faces are assumed closed, features may thus overlap with one another. Two features are *adjacent* if they have nonempty intersection. Two adjacent features are also called *neighbors*. A sequence of features $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k$ forms a *path* between $\mathcal{F}_1$ and $\mathcal{F}_k$, if $\mathcal{F}_i$ and $\mathcal{F}_{i+1}$ are adjacent for $i = 1, 2, \ldots, k-1$. The *traversal distance* between two features $\mathcal{F}$ and $\mathcal{F}'$, denoted by $\tau(\mathcal{F}, \mathcal{F}')$, is the minimum length of all paths between $\mathcal{F}$ and $\mathcal{F}'$.

Let $\mathcal{F}$ and $\mathcal{F}'$ be two features on a convex polygon, and $|C(\mathcal{F}, \mathcal{F}')|$ be the number of features on $C(\mathcal{F}, \mathcal{F}')$, the counter-clockwise polygonal chain between $\mathcal{F}$ and $\mathcal{F}'$. The traversal distance between $\mathcal{F}$ and $\mathcal{F}'$ is then $\tau(\mathcal{F}, \mathcal{F}') = \min(|C(\mathcal{F}, \mathcal{F}')|, |C(\mathcal{F}', \mathcal{F})|)$. For convex polyhedra in $\mathbb{R}^3$, the traversal distance can be computed by searching for the shortest path between two features in the graph induced by the features of a polyhedron. The notion of traversal distance is essential for analyzing feature-based distance computation algorithms, such as those that we are going to discuss.

2.2. Feature-based incremental algorithms

There are several ways to verify that a pair of features forms the closest pair. One possibility is to use the notion of *Voronoi regions*. The Voronoi region of a feature $\mathcal{F}$ on a polyhedron P, denoted by

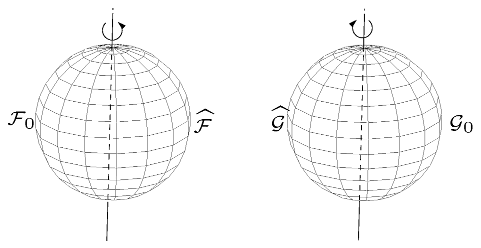

Fig. 1. A bad case for choosing the last closest pair for $(\mathcal{F}_0, \mathcal{G}_0)$. The initial pair $(\mathcal{F}_0, \mathcal{G}_0)$ and the current closest pair $(\widehat{\mathcal{F}_0}, \widehat{\mathcal{G}_0})$ are located on opposite sides of spheres.

$\mathrm{VOR}(\mathcal{F})$, is the set of points outside P that is closer to $\mathcal{F}$ than to any other feature on P. Now we can characterize the closest pair of features as follows [12,13].

Theorem 1. *Let $(\mathcal{F}, \mathcal{G})$ be a pair of features from two disjoint convex polyhedra, and $x \in \mathcal{F}$ and $y \in \mathcal{G}$ be the closest pair of points between $\mathcal{F}$ and $\mathcal{G}$. If $x \in \mathrm{VOR}(\mathcal{G})$ and $y \in \mathrm{VOR}(\mathcal{F})$, then (x, y) is a globally closest pair of points between the polyhedra, and $(\mathcal{F}, \mathcal{G})$ is a closest pair of features.*

Both *Lin–Canny* and *V-Clip*, an improvement over the former, are based on searching for a pair of features that satisfy the conditions of Theorem 1. The search starts with a pre-selected pair of features $(\mathcal{F}_0, \mathcal{G}_0)$. At each step, it tests whether the current pair of features satisfies the conditions, and if not, one of the features is updated to its neighbor to shorten their distance. [2] It can be shown that the search always terminates with the closest pair $(\widehat{\mathcal{F}}, \widehat{\mathcal{G}})$, unless it is stuck in a local minimum, which is handled as a special case (see [13] for details).

Clearly a lower bound of the running time of the above algorithm is $d = \tau(\mathcal{F}_0, \widehat{\mathcal{F}}) + \tau(\mathcal{G}_0, \widehat{\mathcal{G}})$, the sum of the traversal distance between $\mathcal{F}_0$ and $\mathcal{F}$ and that between $\mathcal{G}_0$ and $\mathcal{G}$. If d is small, the algorithm may terminate after a few updates; otherwise, it can take a long time to converge. Hence the choice of $(\mathcal{F}_0, \mathcal{G}_0)$ has a significant impact on the performance of the algorithm. In the worst case, the algorithm may take $\Theta(n^2)$ time to locate the closest pair.

If two objects move continuously and the distance between them is computed at each time step, a good heuristic is to set $(\mathcal{F}_0, \mathcal{G}_0)$ to be the closest pair from the previous time step. This strategy works very well if the coherence is high, that is, the closest pairs of features between successive invocations are close to each other. In fact, high level of coherence is a critical condition for the empirically observed nearly constant-time performance of most feature-based distance computation algorithms. Of course, there are situations where the condition does not hold, either because objects have high complexity or because they move very fast. Consider, for example, two spheres displaced at a fixed distance from each other and rotating at constant speed. If between two successive time steps, each of them turns 180°, the new closest feature pair will now be located at exactly the opposite side of the spheres (Fig. 1). A classic feature-based algorithm has to traverse all the way from one side to the other side of a sphere in order to get the correct answer, which can take a long time if the spheres are densely tessellated. A possible cure for this problem is to build hierarchical representations of objects, and tunnel through the inside of the objects by traversing in hierarchies instead of creeping on their surfaces.

[2] In this paper, the distance between two features $\mathcal{F}$ and $\mathcal{G}$ is understood to be the shortest Euclidean distance between all points in $\mathcal{F}$ and all points in $\mathcal{G}$.

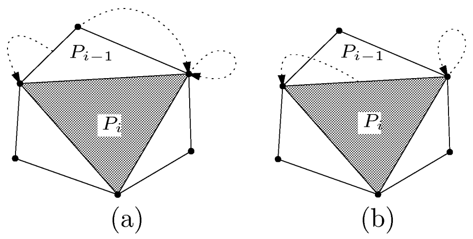

Fig. 2. In and out pointers in the Dobkin–Kirkpatrick hierarchy of a convex polygon. Only representative pointers are shown. (a) In pointers. (b) Out pointers.

2.3. Dobkin–Kirkpatrick hierarchy

The Dobkin–Kirkpatrick hierarchy for a convex polyhedron P is a sequence of convex polyhedra $P_0 = P, P_1, P_2, \ldots, P_k$ where P_{i+1} is contained in P_i, and P_k is a simplex (triangle in $\mathbb{R}^2$ and tetrahedron in $\mathbb{R}^3$). We refer to $P_0, P_1, \ldots, P_k$ as layers. Computing the Dobkin–Kirkpatrick hierarchy for a convex polygon P is straightforward. Suppose that P has n vertices $u_1, u_2, \ldots, u_n$. Choose every other vertices $u_1, u_3, u_5, \ldots$ and form a new convex polygon. Continue this process until a triangle is left. We thus end up with a sequence of polygons $P_0, P_1, \ldots, P_k$, where $k = \mathrm{O}(\log n)$. For a convex polyhedron, each layer P_{i+1} is obtained from P_i by removing an independent set of vertices of P_i of low degree, together with their adjacent features, and retriangulating the holes thus created. A combinatorial lemma shows that in each P_i, we can always find a constant fraction of independent low-degree vertices to remove, and so the hierarchy has again $\mathrm{O}(\log n)$ layers [4].

The Dobkin–Kirkpatrick hierarchy is used in [4] to compute the distance between two convex polygons in $\mathrm{O}(\log n)$ time and the distance between two convex polyhedra in $\mathrm{O}(\log^2 n)$ time. We also use this hierarchy to facilitate the distance computation, but in a different way. With each feature $\mathcal{F}$ of P_i for $1 \leqslant i \leqslant k-1$, we store an *in* pointer $\mathcal{F}_{\mathrm{in}}$, and an *out* pointer $\mathcal{F}_{\mathrm{out}}$ to connect $\mathcal{F}$ to features of adjacent layers in the hierarchy (Fig. 2). A feature of the outermost layer P_0 has only the in pointer. A feature of the innermost layer P_k has only the out pointer. Specifically, if $\mathcal{F}$ is a vertex in $P_i \backslash P_{i+1}$, then set $\mathcal{F}_{\mathrm{in}}$ to be any vertex on the boundary of the hole created by removing $\mathcal{F}$ and its neighbors. Otherwise, there are two cases. If $\mathcal{F}$ is in both P_i and P_{i+1}, set $\mathcal{F}_{\mathrm{in}}$ to its own copy in P_{i+1}. If $\mathcal{F}$ is adjacent to a feature of P_{i+1}, set $\mathcal{F}_{\mathrm{in}}$ to a feature of P_{i+1} adjacent to $\mathcal{F}$. The out pointers have these same two cases. We set up $\mathcal{F}_{\mathrm{out}}$ similarly. The in and out pointers link all layers of the hierarchy together while preserving, to some extent, the local neighborhood information. They allow us to walk conveniently up and down the hierarchy inside a polyhedron.

2.4. Overview of H-Walk

The two methods described above exploits two different aspects of the problem to gain efficiency. The feature-based method takes advantage of the coherence of motion; the Dobkin–Kirkpatrick algorithm takes shortcuts by going through the interior of polytopes in order to reach the solution more quickly. We propose a new approach that combines the advantages of both. Our algorithm builds a Dobkin–Kirkpatrick hierarchy for each polyhedron and walks in the hierarchies. At each step, the algorithm may either walk to a nearby feature at the same level of the hierarchy as a feature-based method does, or move an adjacent level of the hierarchy as the Dobkin–Kirkpatrick algorithm. Fig. 3 shows schematic drawings

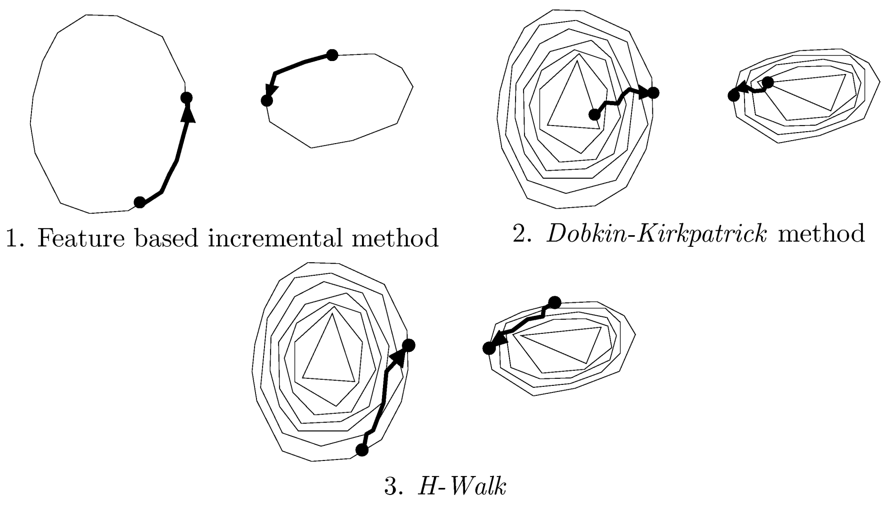
1. Feature based incremental method
2. *Dobkin-Kirkpatrick* method
3. *H-Walk*

Fig. 3. A comparison of three methods. The closed curves are schematic drawings of polyhedra (at different levels of a hierarchy). The thick lines indicate the paths taken by each method.

that compare the three approaches. A feature-based incremental method always walks on the boundaries of polyhedra. *Dobkin–Kirkpatrick* always starts from the innermost layers of hierarchies and tracks the closest feature-pair from layer to layer. In contrast, *H-Walk* starts on the boundaries of polyhedra, walks on the same layers for a few steps, and then drops to inner layers in order to take shortcuts. After locating a closest feature-pair at some level, *H-Walk* tracks it until returning to the outermost layers of the hierarchies. Intuitively when the coherence is high, a few steps of update at the same levels of the hierarchies will quickly bring us to the solution; however, when the coherence is low, we need to dive deeper into the hierarchies to take shortcuts through the interior of polyhedra.

3. Convex polygons

In two dimensions, we can treat the Dobkin–Kirkpatrick hierarchy as a tree whose parent and child pointers correspond to the in and out pointers of the hierarchy, respectively. The features on the same level of a hierarchy are linked into a circular list. The resulting data structure resembles a *skip list* (Fig. 4), where each layer of the hierarchy corresponds to a level in the skip list. Links connecting nodes in the same level correspond to polygon edges. Links connecting nodes between two adjacent levels correspond to in and out pointers. In a skip list, there is an $\mathrm{O}(\log c)$ path between any two bottom level nodes whose indices differ by c, just as in the case of a *finger tree* [9]. This is basically the reason why *H-Walk* takes time logarithmic in the traversal distance between an initial and a closest pair of features by walking in the Dobkin–Kirkpatrick hierarchy. In the next two subsections, we first prove the result for the simple case of computing the distance between a point and a convex polygon, and then treat the more complicated case of two convex polygons.

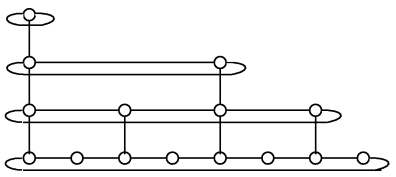

Fig. 4. A skip list representation of the Dobkin–Kirkpatrick hierarchy of a convex polygon. (Although the coarsest level of Dobkin–Kirkpatrick hierarchy has three nodes, this schematic drawing shows levels with one and two nodes to avoid a cluttered picture.)

3.1. Computing distance between a point and a convex polygon

The algorithm for computing the distance between a point q and convex polygon P is based on the following well-known fact.

Lemma 2. *For any point $q \notin P$ and any two vertices u, v of P, we can decide in* $\mathrm{O}(1)$ *time whether the closest feature in P to q is on the polygonal chain $C(u, v)$.*

To locate $\widehat{\mathcal{F}}$, the closest feature to q, we perform a two-stage binary search that resembles locating an element in a sorted list. In the first stage, we start from some initial feature $\mathcal{F}_0$ and check features with traversal distance $1, 2, 4, 8, \ldots$ away from $\mathcal{F}_0$ in the counter-clockwise (CCW) direction, until we reach a feature $\mathcal{G}$ such that $\widehat{\mathcal{F}} \in C(\mathcal{F}_0, \mathcal{G})$. In the second stage, we do a binary search on the chain $C(\mathcal{F}_0, \mathcal{G})$ to locate $\widehat{\mathcal{F}}$. Clearly, the two-stage binary search takes $\mathrm{O}(\log |C(\mathcal{F}_0, \widehat{\mathcal{F}})|)$ checks, and each check takes constant time according to Lemma 2. To obtain an overall $\mathrm{O}(\log |C(\mathcal{F}_0, \widehat{\mathcal{F}})|)$ bound, we just need to access features whose traversal distance to any given feature form a geometric progression roughly. The set of edges provided by Dobkin–Kirkpatrick hierarchy offers one possibility to make such accesses: the traversal distance between any two adjacent vertices on layer i is approximately 2^i, and we can move between layers by following in and out pointers in the hierarchy.

By definition, the traversal distance $\tau(\mathcal{F}_0, \widehat{\mathcal{F}})$ is the minimum of $|C(\mathcal{F}_0, \widehat{\mathcal{F}})|$ and $|C(\widehat{\mathcal{F}}, \mathcal{F}_0)|$. We, therefore, need to decide whether to walk in the CCW or CW direction in stage one of the algorithm. A simple way to get around the problem is to run the algorithm simultaneously in both directions, and stop as soon as $\widehat{\mathcal{F}}$ is found. Hence the algorithm takes time logarithmic in the traversal distance between $\mathcal{F}_0$ and $\widehat{\mathcal{F}}$.

Theorem 3. *Given any feature $\mathcal{F}_0$ in a convex polygon P as a starting point, we can compute the closest feature $\widehat{\mathcal{F}}$ in P to any point $q \notin P$ in* $\mathrm{O}(\log \tau(\mathcal{F}_0, \widehat{\mathcal{F}}))$ *time.*

The binary search procedure described above easily provides a logarithmic bound in terms of the traversal distance, but cannot be generalized to three dimensions, since there is no linear ordering in three dimensions. We now describe a similar, but slightly different technique which can be extended to handle the three-dimensional case. Instead of performing a binary search, we can "walk" in the Dobkin–Kirkpatrick hierarchy. The new technique maintains a feature $\mathcal{F}$ together with $P_{\mathcal{F}}$, the layer that $\mathcal{F}$ is on, and update $\mathcal{F}$ until the closest feature $\widehat{\mathcal{F}}$ is found. Starting from a given initial feature $\mathcal{F}$ on P, it checks if $\mathcal{F}$ is the closest feature on $P_{\mathcal{F}}$ to q while counting the number of updates performed on that layer. If $\mathcal{F}$ is not the closest feature, we walk to the counter-clockwise (CCW) neighbor of $\mathcal{F}$ and continue until either the number of updates exceed a prescribed constant s or the closest feature on that layer is found. In the former case, we follow $\mathcal{F}_{\text{in}}$, the in pointer of $\mathcal{F}$ to descend to an inner layer of P. In the latter case, we start to ascend in the hierarchy. During the ascension, we always walk on the same layer in the direction in which the distance to q decreases until the closest feature to q on that layer is reached. We then follow the out pointer $\mathcal{F}_{\text{out}}$ to move to an outer layer, and continue until we return to the outermost layer $P_0 = P$ and obtain the closest feature on P to q. Again, the procedure has to be performed simultaneously in both the CW and CCW direction. A nice property of the new procedure is that all the operations are local, which makes it possible to generalize to three dimensions.

3.2. Computing distance between two convex polygons

Computing the distance between two convex polygons P and Q is a little bit more complex but the underlying idea is the same. Our algorithm makes use of a key lemma due to Edelsbrunner [5].

Lemma 4. *Let $C(u_1, u_2)$ be a polygonal chain on P, and $C(v_1, v_2)$ be polygonal chain on Q. If $C(u_1, u_2)$ and $C(v_1, v_2)$ contain the closest pair of features between P and Q, then for any vertex $u \in C(u_1, u_2)$ and $v \in C(v_1, v_2)$, we can decide in* $\mathrm{O}(1)$ *time to discard at least one of four chains $C(u_1, u)$, $C(u, u_2)$, $C(v_1, v)$ and $C(v, v_2)$ so that the closest pair of features is contained in the remaining chains.*

If a subchain of a chain C is discarded, we say that C is *refined*.

Our algorithm tries to maintain two chains that contain the closest pair and refine them until the closest pair is located. In each step, we choose an appropriate vertex, called a *pivot*, on each chain and refine the two chains with respect to the pivots according to Lemma 4. If we always choose the middle vertex of a chain as a pivot, we will obtain an $\mathrm{O}(\log n)$ time algorithm, as in [5]. To achieve a bound logarithmic in the traversal distance between an initial and a closest pair of features, we follow a strategy similar to the one in the previous section. We first identify two chains, one on each polygon, that contain a closest pair of feature and then apply binary search to them to localize the closest pair.

Define the *antipodal* vertex $\dot{u}$ of a vertex $u \in P$ to be the vertex in P such the number of features between u and $\dot{u}$ in the CCW direction is $\lfloor |P|/2 \rfloor$, that is, $|C(u, \dot{u})| = \lfloor |P|/2 \rfloor$. Let us assume, for now, that the closest pair of features is contained in $C(u_0, \dot{u}_0)$ and $C(v_0, \dot{v}_0)$ for some initial pair of vertices $u_0 \in P$ and $v_0 \in Q$. The refinement strategy consists of two stages and is exactly the same for $C(u_0, \dot{u}_0)$ and $C(v_0, \dot{v}_0)$. Without loss of generality, we describe it for $C(u_0, \dot{u}_0)$ only. In stage one, we successively choose vertices with distance $s, 2s, 4s, \ldots$ away from u_0 in the CCW direction as the pivot u, where s is some constant, and refine the current chain. If $C(u, \dot{u}_0)$ is discarded instead of $C(u_0, u)$, we know that the closest feature is contained in the chain $C(u_0, u)$ and enter stage two. In stage two, the algorithm always picks the middle vertex of the current chain as the pivot, and continue refining until we get a chain of length one and find the closest feature on P. Note that the algorithm chooses the pivots independently for P and Q, depending on which stages P or Q is in. Furthermore, Lemma 4 guarantees that pivots always exist and the refinement process can proceed until we end up with two chains of length one.

The procedure described above is essentially a binary search. As we have seen in Section 3.1, binary search on a convex polygon is very similar to walking in the Dobkin–Kirkpatrick hierarchy. We have chosen to describe the algorithm as binary search because it simplifies the presentation of analysis, but as before, walking in the Dobkin–Kirkpatrick hierarchy extends more easily to three dimensions.

The running time of the algorithm is proportional to the total number of refinements required to locate the closest pair $(\widehat{\mathcal{F}}, \widehat{\mathcal{G}})$ between the two polygons P and Q. First let us consider the refinements for P. In stage one, every time we pick a new pivot u, the distance from u to u_0 doubles. After $\mathrm{O}(\log|C(u_0, \widehat{\mathcal{F}})|)$ steps, stage one ends with a pivot ξ such that the chain $C(u_0, \xi)$ is guaranteed to contain $\widehat{\mathcal{F}}$. The number of refinements in stage two is clearly bounded by $\mathrm{O}(\log|C(u_0, \xi)|)$ because we perform a binary search on $C(u_0, \xi)$. Combining the results for both stage one and two, we conclude that the number of refinement steps for P is bounded by $\mathrm{O}(\log|C(u_0, \widehat{\mathcal{F}})|)$. The same analysis applies to Q. So the total number of refinements is $\mathrm{O}(\log|C(u, \widehat{\mathcal{F}})| + \log|C(v, \widehat{\mathcal{G}})|)$.

So far, we have assumed that $C(u_0, \dot{u}_0)$ and $C(v_0, \dot{v}_0)$ contains the closest pair of features, because we always move in the CCW direction starting from u_0 and v_0. Just as in Section 3.1, we can move in either CCW or CW direction at both u_0 and v_0. By running the algorithm simultaneously for all four possibilities, we obtain the following result.

Theorem 5. *Let P and Q be two convex polygons. Given a pair of feature $\mathcal{F}_0 \in P$ and $\mathcal{G}_0 \in Q$ as a starting point, we can compute the closest pair of features $(\widehat{\mathcal{F}}, \widehat{\mathcal{G}})$ between P and Q in time* $\mathrm{O}(\log\tau(\mathcal{F}_0, \widehat{\mathcal{F}}) + \log\tau(\mathcal{G}_0, \widehat{\mathcal{G}}))$ *time.*

4. Convex polyhedra

4.1. Algorithm

Our algorithm in three dimensions employs the same basic idea of walking in hierarchically represented polyhedra. Suppose that we have pre-computed the Dobkin–Kirkpatrick hierarchies $P_0 = P, P_1, \ldots, P_k$ and $Q_0 = Q, Q_1, \ldots, Q_l$ for two convex polyhedra P and Q. The algorithm starts with an initial pair features $(\mathcal{F}_0, \mathcal{G}_0)$, and again proceeds in two stages. In stage one, it walks a constant number s of steps at each level of the hierarchies until it reaches the bottom level. In each step, the algorithm checks to see whether the current pair of features is a closest pair for the current layers. If it fails to find the closest pair after s steps, it then follows the in pointers of the current feature-pair to descend one level in the two hierarchies simultaneously, if possible, and continue until a closest pair of features is found between two polyhedra P_i and Q_j. In stage two, the algorithm walks as many steps on the same level as needed in order to find the closest pair $(\widehat{\mathcal{F}}_i, \widehat{\mathcal{G}}_j)$ between polyhedra P_i and Q_j. It then follows the out pointers of $\widehat{\mathcal{F}}_i$ and $\widehat{\mathcal{G}}_i$ to ascend one level in the hierarchies and continues the walk. When we return the outermost layers P_0 and Q_0, the closest pair between P and Q is found. The reader may refer to Fig. 3 for an illustration of the idea.

If the coherence is high, then in stage one, we will be likely to descend only a small number of levels. In stage two, since P_i and Q_j are approximations to P_{i-1} and Q_{j-1}, respectively, $(\widehat{\mathcal{F}}_i, \widehat{\mathcal{G}}_j)$ is likely to be near $(\widehat{\mathcal{F}}_{i-1}, \widehat{\mathcal{G}}_{j-1})$, and hence we will walk only a small number of steps at each level. By going down and then going up the hierarchies, we avoid the potentially expensive cost of walking a long way on the boundaries of P_0 and Q_0.

Choosing an initial pair of features $(\mathcal{F}_0, \mathcal{G}_0)$ remains an issue here, since there are many choices available. During the last invocation of *H-Walk*, we have found a sequence of closest pairs between two polyhedra approximating P and Q; any one of them may serve as a candidate for the initial pair.

If we choose $(\mathcal{F}_0, \mathcal{G}_0)$ from the innermost layers P_k and Q_l, the behavior of *H-Walk* is similar to that of *Dobkin–Kirkpatrick*, since both start with the innermost layers, move one level at a time towards the outermost layers to find the closest pair between P and Q. However, each step of *H-Walk* is simple and efficient: just walk from a pair of features to an adjacent one or follow the out pointers to ascend one level. *Dobkin–Kirkpatrick*, on the other hand, uses a very elaborate scheme every time it has to go up one layer, in order to guarantee the worst-case bound. As a result, each step of *Dobkin–Kirkpatrick* takes much longer to execute, and it is also less robust because of the more subtle geometric computation involved.

Starting the search from the innermost layers basically ignores the coherence of motion. To take advantage of coherence, we may, for example, pick the closest pair between the outmost layers P_0 and Q_0, if we expect to descend and ascend only a small number of levels in the hierarchies before reaching the closest pair. However, if the coherence is low, we must descend deeply into the hierarchies and ascend all the way up. Time spent on descending is wasted. In this case, it is much more efficient to start with the closest pair at some deeper level and pay only the cost of ascending.

As we mentioned previously, there are actually a sequence of closest pairs at different levels of the hierarchies from the last invocation of *H-Walk* to choose for $(\mathcal{F}_0, \mathcal{G}_0)$. Depending on the level of coherence, one of them might be better than the others. We will look at the impact of these choices on the behavior of the resulting algorithm in the experiments.

4.2. Experiments

We use a pair of identical objects in our experiments. Since only the relative motion of the two objects matters, one of them is fixed at the origin, and the other orbits around it and rotates about some axis according to the pseudo-code below. This test scheme is due to Mirtich [13].

```
**Procedure 1**
**Input**: angular velocity $\omega$, and constant $A$ chosen to avoid penetration of objects 1 and 2
1.  Fix object 1 at the origin.
2.  **for** $i = 1$ to 10 **do**
3.      $v \leftarrow$ a vector sampled uniformly at random from the unit sphere.
4.      **for** $j = 1$ to 100 **do**
5.          $\theta \leftarrow j\omega$.
6.          $x \leftarrow (A\cos\theta, A\sin\theta, A\cos\theta)$.
7.          $\rho \leftarrow$ rotation by $\theta$ about axis $v$.
8.          Set object 2 at $x$ with orientation $\rho$.
9.          Compute distance between objects 1 and 2.
```

For every test, we ran the algorithm once for each method of picking the initial feature pair of features. In particular, we pick the closest pair at layer 0, 4, 8, 16, and the innermost layer (labeled “h-walk 0”, “h-walk 4”, “h-walk 8”, “h-walk 16” and “h-walk ∞” in Figs. 5–7) to initialize *H-Walk*. If an object is too simple to have enough number of layers, we simply skip those layers. We also ran *V-Clip*, a very efficient implementation of *Lin–Canny*, for comparison.

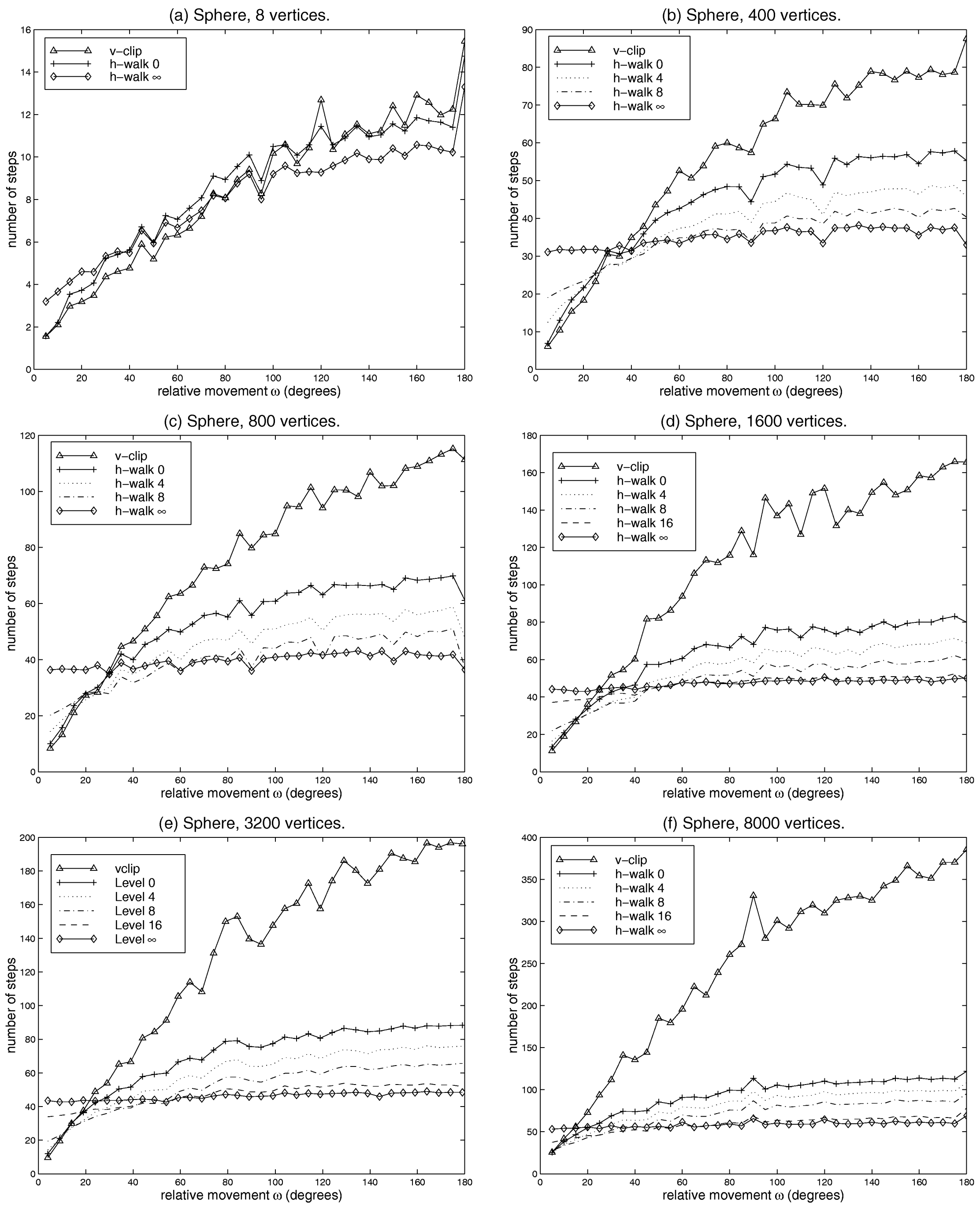

Fig. 5. Operation counts for spheres tessellated at various densities.

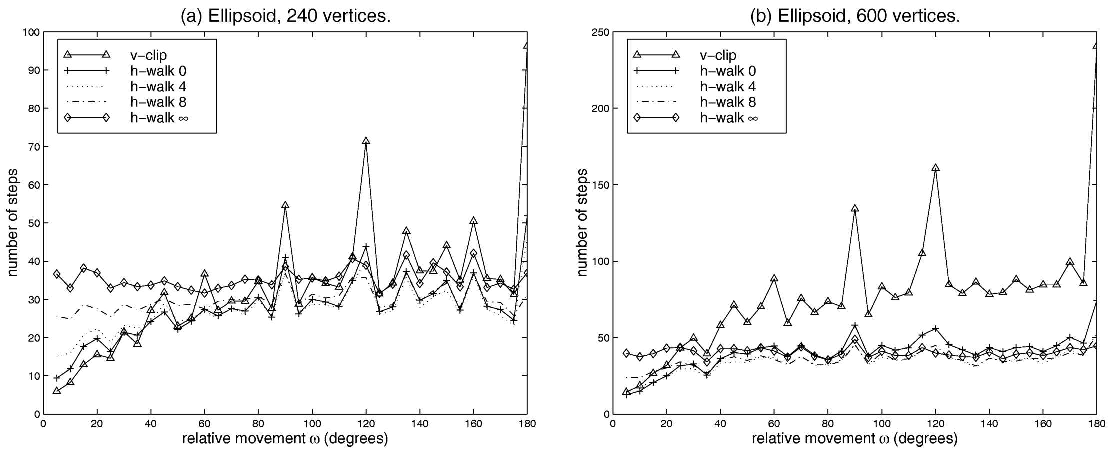

Fig. 6. Operation counts for ellipsoids with two different shapes and tessellation densities.

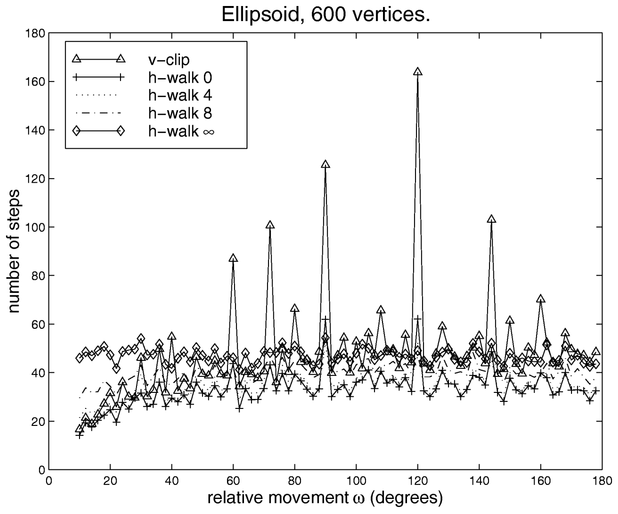

Fig. 7. Operation counts for two ellipsoids placed in close proximity.

The parameter s, the number of steps to walk on each level, was set to be 4 for all the results shown here, but according to our experiments, the choice of s does not affect the performance of the algorithm significantly. A small constant usually works well.

Coherence is a key factor that determines the efficiency of feature-based incremental distance computation algorithms, including *H-Walk*. It is difficult to define coherence precisely, but usually high coherence means that objects have low combinatorial complexity, regular geometric shape, and low velocity of motion. Thus, in the above test scheme, the greater the angular velocity ω, the lower the coherence is.

Figs. 5–7 plot the average number of steps walked per invocation to reach the closest pair versus the coherence parameter ω.

The first set of tests consists of pairs of spheres tessellated at various densities. Fig. 5(a) shows the results for a pair of spheres tessellated with eight vertices. Since the objects are extremely simple, hierarchical data structures do not help much. All algorithms have similar performance and do not differ by more than a few steps. Fig. 5(b) plots the result for spheres tessellated with 400 vertices. The graph shows that when the coherence is high (ω is small), initializing with the closest feature-pair near the outermost layer gives better performance, because coherence helps us to avoid the cost of descending to the bottom level. As coherence gets lower and lower, initializing with the closest feature-pair on the innermost layer eventually becomes the best choice. Figs. 5(c)–(e) show the plots for spheres tessellated with 800, 1,600 and 3,200 vertices, respectively. The results are similar to the previous one except that the trend is now much clearer. In the four cases shown in Figs. 5(b)–(e), when the coherence is high ($\omega < 20$), the performance of *V-Clip* is comparable with that of *H-Walk* initialized with the closest feature-pair near the outermost layers, and is better than *H-Walk* initialized with the closest feature-pair near the innermost layers. However, *V-Clip*'s performance quickly deteriorates as coherence gets lower. When $\omega > 30$, *H-Walk* shows better performance than *V-Clip* regardless of the method of initialization. The computational cost of *V-Clip* grows almost linearly with respect to ω, the coherence parameter; the growth of *H-Walk* is much smaller. Notice also that when the combinatorial size of spheres quadruples (compare, for example, Figs. 5(b) and (d)), *V-Clip* almost doubles the amount of computation time needed, while *H-Walk* shows only moderate increase of computation time. Finally, Fig. 5(f) gives an extreme example that contains a pair of spheres tessellated with 8,000 vertices. *H-Walk* is significantly more efficient than *V-Clip* in this case.

The second set of tests uses more elongated shapes. Fig. 6(a) shows the result for a pair of ellipsoids, which contain 240 vertices each and have axes of length 1, 0.1 and 0.1. Fig. 6(b) shows the result for another pair of ellipsoids, which contain 600 vertices each and have axes of length 1, 0.04 and 0.04. Again initializing with the closest feature-pair near the top level is advantageous if the coherence is high, though this turns to be a good choice for a larger range of coherence levels here. *V-Clip* has a slight edge at high level of coherence for the simpler ellipsoids (Fig. 6(a)), but *H-Walk* performs better generally for the more complex and elongated ellipsoids (Fig. 6(b)). Notice that the elongated shape of the objects makes the performance curves more erratic than those for the spheres, especially for *V-Clip*. As one would expect, this phenomenon is accentuated (Fig. 7) if we move the two ellipsoids closer together, that is, to reduce the size of input parameter A in Procedure 1. The performance curve of *V-Clip* contains sharp "spikes" when the closest pair jumps a long way from one place to another. In contrast, the performance of *H-Walk* is much more consistent.

These controlled experiments demonstrate that when the coherence is high, both *V-Clip* and *H-Walk* perform well. If the complexity of objects or the velocity of motion increases, *V-Clip*'s performance starts to deteriorate, because it has to walk through more and more features. *H-Walk* also slows down, but not nearly as much, because the hierarchical data structure keeps the number of steps that it has to walk in control. Also, irregular shapes causes more problems for *V-Clip* than for *H-Walk*. For two elongated objects, a small change in the position or orientation of objects may cause the closest pair to jump from one end of the object to the other. *V-Clip* has to walk a long way before reaching the new closest pair. *H-Walk* can do it much faster by using the hierarchies.

To test the performance of *H-Walk* in more realistic scenarios, we built a simple simulator for rigid-body objects flying in three-dimensional space (see Fig. 8 for an example). In a larger system, a simulator like this one may be integrated with dynamic simulation or motion planners to generate the motion of objects. At each time step, our simulator computes the distance between every pair of convex objects

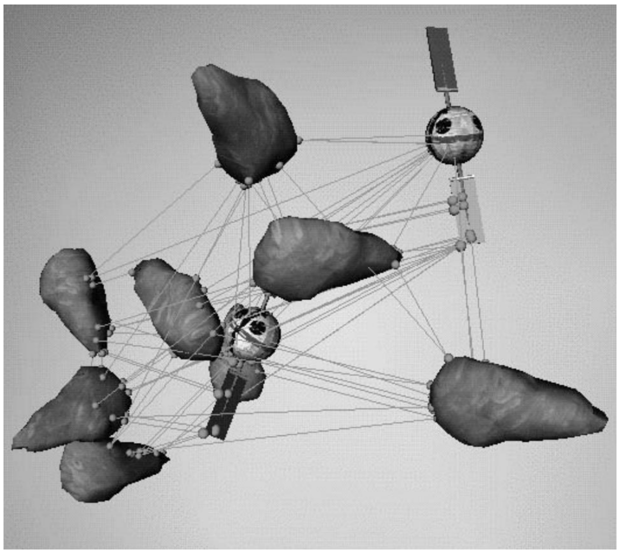

Fig. 8. Satellites flying amidst asteroids. The small spheres on the surface of objects mark the closest features.

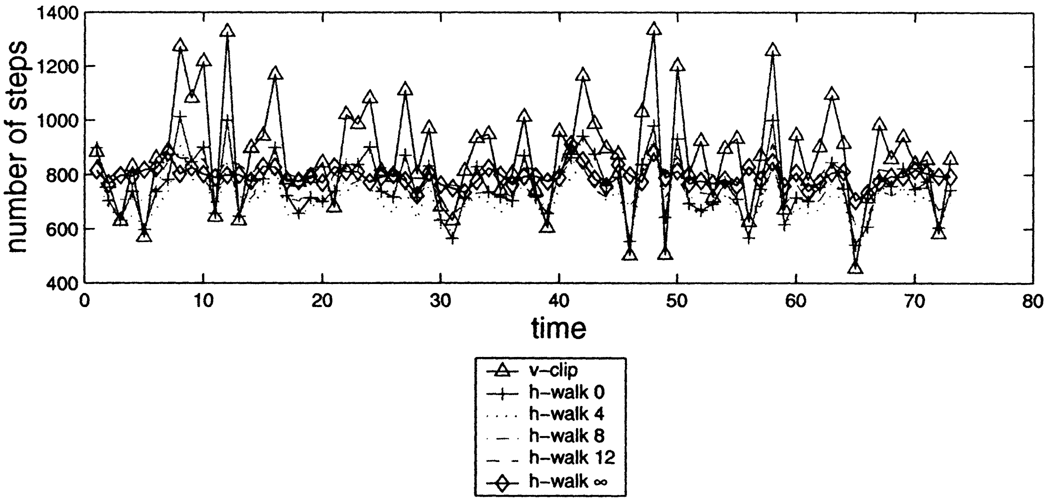

Fig. 9. Running times for *H-Walk* and *V-Clip*.

in the scene. Fig. 9 plots the algorithms' performance versus time for a test run in which a number of satellites fly among moving asteroids. All objects in the test move at about the same speed. Fig. 10 reports the mean and standard deviation of running times for *H-Walk* and *V-Clip*. Although these results were obtained from a specific set of test data, it is nevertheless interesting to note that as features at deeper levels of the hierarchies are used to initialize *H-Walk*, the mean execution time first decreases and then starts to rise again, but the standard deviation decreases monotonically. This is an interesting trade-off in practical applications: after a certain point, we may achieve more consistent performance, but only by sacrificing the average running time, or vice versa.

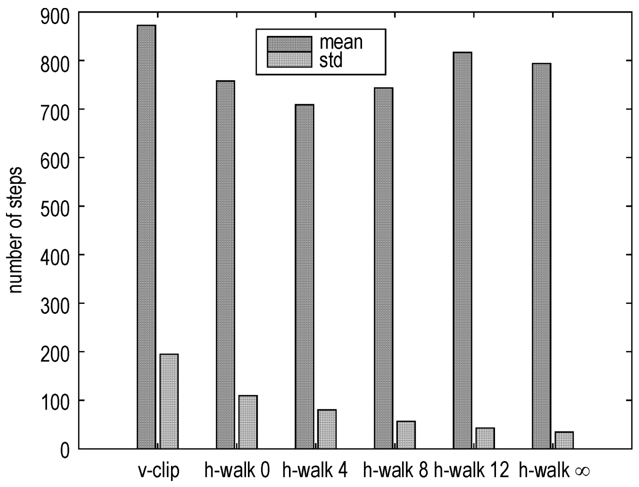

Fig. 10. The mean and standard deviation of the running times for *H-Walk* and *V-Clip*.

4.3. Discussion

In addition to showing its efficiency, the experimental results demonstrate a number of other benefits of *H-Walk* in practical applications.

Time-critical collision detection. The performance of *H-Walk* is much more consistent than the traditional feature-based incremental methods, which depend heavily on the motion coherence. This consistency allows for more accurate predication of the time for each distance computation, which is important in the time-critical simulation when the computation resource is allocated for each time step and not allowed to exceed the given limit [11].

Robustness. We have also learned in our experiments that *H-Walk* is very robust. It is not a surprise as the only geometric computation involved in the feature-based incremental algorithms is to check whether two features forms the closest pair. As described in [13], such computation can be done robustly. The original Dobkin–Kirkpatrick method use more complex geometric computation such as computing the separation planes and cutting a polyhedron by a plane, which are more prone to numerical errors. Furthermore a numerical error in the computation in *H-Walk* may only affect the efficiency of the algorithm but not the correctness. However, in *Dobkin–Kirkpatrick*, a numerical error more likely results in an incorrect answer.

The experimental results also suggest some possibilities for further improvement of *H-Walk.*

Adapting to coherence levels. The experimental data in the previous section show that the best method for choosing $(\mathcal{F}_0, \mathcal{G}_0)$ depends on the level of coherence. For instance, “h-walk ∞”, whose performance characteristic is similar to that of *Dobkin–Kirkpatrick,* does not take much advantage of the coherence. Its performance is the best at low levels of coherence, but is the worst at high levels of coherence. In many applications, the level of coherence is unknown in advance and changes over time. It is thus desirable for an algorithm to take into account the level of coherence and operate efficiently at different levels of coherence. By using a suitable strategy for picking the initial pair of features $(\mathcal{F}_0, \mathcal{G}_0)$, *H-Walk* can respond to changes in the level of coherence. When the coherence is high, we choose for $(\mathcal{F}_0, \mathcal{G}_0)$ the closest pair from the last time step that is near the outermost layer. When the coherence is low, we use the closest pair near the innermost layer. Furthermore, the behavior of *H-Walk* actually gives a hint on the level of coherence. Note that the algorithm first descends and then ascends in the hierarchies. If at some point, the algorithm only ascends from the initial pair, it means that the coherence might be higher than expected. In the next time step, we may then choose the initial pair closer to the outermost layer. On the other hand, if the algorithm descends too many levels, then we may want to choose a closest pair at deeper levels as the starting point. Therefore, *H-Walk* provides an easy way to estimate the level of coherence and adapts its own behavior correspondingly.

Theoretical bounds. Although the experiments indicate that the computation time of our algorithm grows slowly as the coherence gets lower, we are unable to obtain a theoretical bound. There seems to be two aspects of the difficulty. First, if a feature $\mathcal{F}$ has m adjacent features, the information-theoretic lower bound for finding the closest feature $\widehat{\mathcal{F}}$ to a point is $\Omega(\log m)$ even if $\widehat{\mathcal{F}}$ is adjacent to $\mathcal{F}$. Since in a convex polyhedron P, m can be linear in n, the size of P, a nice theoretical bound is possible only if the graph G induced by the features of P has bounded degree. Second, in the graph theory setting, our problem is basically to add a small number of edges to G in order to reduce the length of the shortest path between any two nodes in G. More specifically, if the length of the shortest path between two nodes of G is c, then we would like the length of the shortest path in the augmented graph to be $\log c$. The set of edges provided by Dobkin–Kirkpatrick hierarchy does not seem to have this property. In addition, even if we can construct such an augmented graph, we still need a strategy for finding such a shortest path by performing local search only.

Different hierarchies. In the paper, we use the *Dobkin–Kirkpatrick* hierarchy as shortcuts to quickly go through the interior of a convex polytope. However, *Dobkin–Kirkpatrick* hierarchy is built entirely combinatorially and the geometry of object is ignored when building such a hierarchy. There are other types of hierarchies which also takes the geometric shape into account [6]. In the hierarchies proposed in that paper, not only does the hierarchy enjoy the same combinatorial property as *Dobkin–Kirkpatrick* hierarchy, but also the layers in the hierarchy are "close" to each other as the Hausdorff distance between adjacent layers is bounded in terms of the diameter of the object. Such hierarchies can help us to obtain an estimate on the distance at the end of stage one, like what is done in [6]. When we only need to detect collision, we can safely claim that two objects are disjoint once we know that the distance is positive. Even if for the distance computation, since two adjacent layers have similar shapes, we expect that each walking in the second stage will take fewer steps. We also feel that the use of different hierarchies may be necessary for establishing a theoretical bound in three dimensions.

5. Conclusion

This paper presents the *H-Walk* algorithm, which computes the distance between two moving convex objects by exploiting both the coherence of motion and hierarchical representation. For convex polygons, we have proven that *H-Walk* takes time logarithmic in the traversal distance between an initial pair of features and a closest pair. For convex polyhedra in three dimensions, experimental results indicate that unlike previous incremental distance computation algorithms, *H-Walk* is very efficient at different levels of coherence, and the performance of the algorithm has a trend similar to that has been proven in two dimensions. It can also adapt its behavior automatically to adjust to changes in the level of coherence in order to maintain the best performance.

Proving the theoretical bound for the algorithm in three dimensions remains an interesting open question. The lack of a linear ordering on the surface of a polyhedron makes it difficult to obtain such a bound. A more "regular" hierarchical structure may be needed in order to achieve the goal.

So far, our discussion has focused on the traversal distance, an entirely combinatorial parameter. It would be interesting to know whether we can adopt the approach in [6] to take the geometric shape into account when we build the hierarchy so as to derive bounds dependent on the parameters of motion and the sampling rate of time.

Acknowledgements

We would like to thank Brian Mirtich for helpful discussion of the collision detection problem and for providing us an implementation for the V-Clip algorithm.

References

[1] S. Cameron, Enhancing GJK: Computing minimum and penetration distances between convex polyhedra, in: Proc. IEEE Int. Conf. on Robotics and Automation, 1997.

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

[3] D.P. Dobkin, D.G. Kirkpatrick, A linear algorithm for determining the separation of convex polyhedra, J. Algorithms 6 (1985) 381–392.

[4] D.P. Dobkin, D.G. Kirkpatrick, Determining the separation of preprocessed polyhedra – A unified approach, in: Proc. 17th Internat. Colloq. Automata Languages Programming, Lecture Notes in Computer Science, Vol. 443, Springer, Berlin, 1990, pp. 400–413.

[5] H. Edelsbrunner, Computing the extreme distances between two convex polygons, J. Algorithms 6 (1985) 213–224.

[6] J. Erickson, L.J. Guibas, J. Stolfi, L. Zhang, Separation-sensitive convex collision detection, in: Proc. 10th Annual ACM–SIAM Symp. on Discrete Algorithms, 1999, pp. 327–336.

[7] E.G. Gilbert, D.W. Johnson, S.S. Keerthi, A fast procedure for computing the distance between complex objects, IEEE J. Robotics Autom. 4 (2) (1988).

[8] S. Gottschalk, M. Lin, D. Manocha, OBB-Tree: A hierarchical structure for rapid interference detection, in: Computer Graphics (SIGGRAPH '96 Proceedings), 1996, pp. 171–180.

[9] L.J. Guibas, E.M. McCreight, M.F. Plass, J.R. Roberts, A new representation for linear lists, in: Proc. ACM Symp. on Theory of Computing, 1977, pp. 49–60.

[10] P.M. Hubbard, Collision detection for interactive graphics applications, IEEE Trans. Visual. Comput. Graphics 1 (3) (1995) 218–230.

[11] P.M. Hubbard, Approximating polyhedra with spheres for time-critical collision detection, ACM Trans. Graphics 15 (3) (1996) 179–210.

[12] M.C. Lin, J.F. Canny, A fast algorithm for incremental distance calculation, in: Proc. IEEE Int. Conf. on Robotics and Automation, Vol. 2, 1991, pp. 1008–1014.

[13] B. Mirtich, V-Clip: fast and robust polyhedral collision detection, Technical Report TR-97-05, Mitsubishi Electrical Research Laboratory, 1997.
[14] S. Quinlan, Efficient distance computation between non-convex objects, in: Proc. IEEE Int. Conf. on Robotics and Automation, 1994, pp. 3324–3329.