Efficient Collision Prediction Among Many Moving Objects

Vincent Hayward, Stéphane Aubry, André Foisy and Yasmine Ghallab

The International Journal of Robotics Research 1995; 14; 129
DOI: 10.1177/027836499501400203

Vincent Hayward
Stéphane Aubry
André Foisy
Yasmine Ghallab
Electrical Engineering Department and
Center for Intelligent Machines
McGill University
Montréal, Québec, Canada H3A 2A7

Efficient Collision Prediction Among Many Moving Objects

Abstract

We consider the problem of flagging all collisions between a large number of dynamic objects. Because the number of possible collisions grows quadratically with the number of objects, a brute force approach is not applicable with finite computational resources.

Hence, we propose a scheduling mechanism that reduces the computational load by exploiting the coherence of the world throughout time. This mechanism has a very simple structure and easily lends itself to distributed processing. It considers all pairwise interactions between objects and maintains a structure that reflects the imminence, or urgency, of collision for each pair. Bounds on the urgency of collisions can be computed given minimal knowledge of the system dynamics. For example, we represent physical objects by their positions and by bounds on their relative speeds and accelerations. These are assumed to be available at all times. If the environment does not change too rapidly, the mechanism flags all collisions. False alarms may also be generated but can be eliminated with a specialized exact collision post-processor.

We address the question of how often to perform the collision checks while guaranteeing that all collisions will be caught. Given the large number of possible environments and motions, no general optimal answer can be provided. Yet the soundness and efficiency of the proposed algorithm is experimentally verified in the case of a simple world consisting of many spheres moving simultaneously and randomly.

1. Introduction

Consider a world made up of N objects, each moving along an arbitrary, possibly unknown, trajectory. We wish to design a system capable of flagging all collisions between them before they occur. A familiar example of this problem could be that of a traffic controller, placed in the middle of a busy intersection, who must process sensor information fast enough to prevent all collisions. It may also correspond to the case of a robot moving a collection of limbs, abstracted to a collection of objects, subject to unpredictable velocity demand signals and for which we want to automate a collision prevention behavior.

For each of the $\binom{N}{2} = O(N^2)$ object pairs, we compute a time value τ. τ is the amount of time (measured from the instant at which it is computed) for which we are guaranteed no collision will occur for the pair. It can be computed as long as the rate of change of the world is guaranteed not to exceed some bounds. This is, of course, the case of the physical world at a macroscopic scale.

We further assume that this computation can be repeated for each pair with a period of Δt. A brute force approach to solving the problem of flagging collisions then leads to a simple algorithm: τ is recomputed every Δt seconds. Whenever a pair's τ value is less than Δt, the pair is flagged, indicating a possible collision. This algorithm takes $O(N^2)$ computations at each sample time, which is unacceptable in most practical situations.

Our intuition tells us that we do not apply this suggested simple algorithm to avoid collisions in a crowded environment (on the street, for example). Instead, we exploit the coherence through time of the world and we handle the situation with a presumably smaller amount of computations. We evaluate all the perceived relationships with the objects that surround us and grade them on a scale of urgency. According to that scale, we dispense a variable amount of attention to each relationship. In addition, we dynamically adjust our assessment as the situation evolves. τ provides such an assessment, or measure of urgency. Thus, it not only conveys the frequency with which we must flag for a possible collision, but it also conveys the frequency with which τ must be recomputed!

The Dynamic Urgency Algorithm (DUA), which this article presents, uses that strategy. At initialization, τ is evaluated for all pairs of objects. The ones in greater

The International Journal of Robotics Research,
Vol. 14, No. 2, April 1995, pp. 129–143,

danger of colliding are given high priority and hence are reevaluated more frequently, until we are satisfied they no longer pose an imminent danger. At that time we let the frequency of their reevaluation drop in favor of other pairs whose measure of urgency may have overtaken theirs. The net effect of the strategy, compared with the exhaustive simple collision flagging scheme, is a great reduction in the load of the computational resources, which are necessarily finite in time and space.

The urgency function used to compute τ depends on the available a priori knowledge. In the case of physical objects, the laws of mechanics apply and provide such knowledge. A simple but representative case is worked out in this article.

We claim that DUA is an example of a very general strategy, designed to exhibit a gracefully degrading behavior when faced with a seemingly intractable problem of resource allocation with on-line input acquisition.

Section 2 gives a review of "classic" collision detection, whereas Section 3 introduces our paradigm, which we call collision prediction. The rest of the article is devoted to DUA, a collision predictor. Section 4 gives a brief outline, Section 5 describes the model of the environment, Section 6 describes the urgency function, Section 7 describes the algorithm itself, and Section 8 presents experimental simulations made for DUA.

2. A Review of Collision Detection

Collision detection is the set of techniques used to ascertain whether moving objects collide with each other, given their shapes and trajectories. In the robotics field, such techniques may be used to plan legal trajectories using a generate-and-test approach. This is in contrast to the so-called global path planning techniques, which perform a search through the graph of legal configurations (Avnaim et al. 1988; Brooks 1983; Brooks and Lozano-Pérez 1983; Lozano-Pérez 1983, 1987; Lozano-Pérez and Wesley 1979).

In collision detection, there exist two paradigms. In the first one, the trajectory of the moving object(s) is written in terms of a parameter, and root-finding techniques are applied to compute the value of the parameter at which collisions occur. In the other paradigm, the time scale is discretized, and static collision tests are applied at each step. These two approaches are reviewed in the following subsections.

2.1. Root-Finding Techniques

In three-dimensional polyhedral worlds, only two types of nondegenerate collisions exist: vertex-face (VF) contacts and edge-edge (EE) contacts. This is the basis of most of the approaches described in this section. These approaches also assume that the objects under study are convex or that they have been decomposed into convex subsets on which the techniques can be applied.

The early work of Boyse exhaustively found the roots of the algebraic constraints describing all possible VF and EE contacts and checked whether they were within the constraints expressed by the polyhedra's boundaries. Boyse (1979) considered only purely rotational or purely translational movements, obtaining quadratic and linear equations, respectively.

Canny (1988) later generalized the technique to general trajectories by using a quaternion representation. For two objects, each moving along a straight line in $\Re^3 \times SO^3$, collisions are detected by solving a quintic. Because the tests were also exhaustive over the set of possible VF and EE contacts, the computational complexity was basically the same as that given by Boyse—namely, of the order of the product of the number of vertices of each object pair.

Gilbert and Hong (1990) used an iterative technique based on supporting planes and minimum distance computations and, after extensive experimentation, reported an expected computational complexity linear in the total number of vertices of the polyhedra.

Kawabe et al. (1988) recursively split trajectories into cubic segments until the segments closely approximated the exact trajectories. The method yielded a sixth-degree equation for the case of two moving objects. The advantage of the method seems to be its great simplicity and ease of implementation. Obviously, one is free to choose the order of the polynomial segments approximating the trajectories.

Cameron (1985) chose to use linear path segments and described a hierarchical four-dimensional time-space data structure based on extrusions. Extruded polyhedra form hyperpolyhedra in four-space. The data structure was geared toward quickly checking for intersections in four-space by pruning entire sections of that space.

Finally, through linear programming, Barford (1989) gave a simple and efficient solution for objects moving under pure translations. The theoretical worst-case complexity was linear in the total number of vertices, which was confirmed by experiments.

2.2. Time Increment Techniques

Time increment techniques are an alternative to finding the time of collision for a specified trajectory: It is easier to detect a static collision than it is to compute where on the trajectory a collision may occur. More importantly, there are occasions when trajectories are not known or only partially known a priori, which is the case the present article is tackling. For example, object positions and velocities could be dynamically acquired from sensors.

The question of whether a collision occurs is then answered by the determination of the nullity of the distance separating the objects. Such a determination can be performed by an exhaustive or near-exhaustive search through all pairs of features (vertex, edge, or face), taken one from each object (Red 1983). More efficient methods are discussed in the two next subsections.

2.2.1. *Distance Computation with Computational Geometry Methods*

These methods optimize worst-case asymptotic computational complexity, as a function of the polyhedra sizes, through the use of appropriate data structures.

Dobkin and Kirkpatrick (1985) gave a $\theta(n)$ algorithm, based on a recursive linear-space tree-like representation of polyhedra. The crucial property of the representation is that each level of the tree is made up of a number of vertices, at most a constant fraction α ($0 < \alpha < 1$) of the distance to its predecessor. Since the algorithm takes a linear amount of time at each level of the tree, the ensuing geometric series ensures that the overall complexity is also linear.

If preprocessing of the polyhedra is allowed, sublinear algorithms can be obtained. For example, Dobkin and Kirkpatrick (1983) used a complex data structure for polyhedra, the drum structure, to design an $O(\log^2 n)$ collision detection algorithm, with $O(n^2)$ preprocessing.

2.2.2. *Distance Computation With Iterative Methods*

Iterative methods "walk" along the boundaries of the objects, minimizing a quadratic objective function—namely, Euclidean distance.

Hurteau and Stewart (1988) pose the problem as a quadratic programming program. The objective function to minimize is $\|\mathbf{s} - \mathbf{t}\|$, such that $\mathbf{s}$ and $\mathbf{t}$ are constrained to lie in their respective closed polyhedra. The objective function is positive semidefinite in the sixth-dimensional $(\mathbf{s}, \mathbf{t})^T$ space, and the number of constraints is equal to the total number of faces making up the objects. This problem can be solved using a number of standard techniques and is trivially extended to any number of dimensions (Luenberger 1969).

Bobrow (1989) uses a similar technique based on the Kuhn-Tucker conditions, which relate the active constraints and the (linear) gradient of the objective function. Bobrow reports a linear behavior in the number of constraints, although the worst-case complexity is probably quadratic.

Gilbert, Johnson, and Keerthi (1988) present a technique based on support properties of convex sets, and on Minkowski sums. The basic algorithm is as follows:

1. Get an initial four-vertex set from $S \ominus T$, where S is the vertex set of the first polyhedron, T is the vertex set of the second polyhedron, and $\ominus$ is the Minkowski difference. Set $t = 0$.
2. Calculate ν_t, the furthest point of the set from the origin.
3. Among remaining vertices in $S \ominus T$, find ν_{t+1}, the one with maximum dot product in direction $-\nu_t$.
4. Update the set to include ν_{t+1} and the minimal vertex set needed to express ν_t. Set $t = t + 1$. Go back to step 2.

This algorithm is proven to generate a converging sequence in an n-dimensional space (Gilbert and Johnson 1985). Further, it converges finitely in the case of polytopes. Extensive testing indicates a computational complexity of about 20 flops per point, with an empirically established, almost linear growth. Although contrived examples can be found with quadratic growth, the error in such cases is exceedingly small after a few iterations. We note that the method presented by Faverjon and Tournassoud (1988) is similar to that of Gilbert et al., even though their nomenclature is quite different.

In a later article, Gilbert and Foo (1990) also generalize the method to quadrics using the Lagrange multiplier rule, yielding a great improvement in efficiency.

Finally, we mention the Roider method, which constructs a witness of disjointness—that is, a cone whose apex is at a boundary point of one object and whose interior entirely contains the other object. Unfortunately, no comparison with other methods is offered (Roider and Stifter 1987; Stifter 1988).

Although the methods cited in this section greatly differ in their implementation, most of them report running times that are linear in the number of constraints—namely, the sum of the number of faces of the objects. This is in agreement with previously reported quadratic programming's empirical complexity results (Powell 1985).

2.3. *Iterative Methods and Coherence Through Time*

The iterative methods presented in the last section are generally simpler than those presented in Section 2.1. Still, one of their drawbacks is that the same repetitive tests must be performed anew at each time step, even though it is likely that the state of the world changes little between adjacent steps.

Some recent iterative methods exploit such coherence across the time axis by picking the previous step's solution as the current step's initial solution (Gilbert et al. 1988). This leads to faster expected convergence, as measured in the number of iterations, because the initial guess is generally "close" to the actual solution. The

speed increase causes a large reduction in the number of computations over the course of a complete trajectory.

Using this idea of coherence, Lin and Canny (1991) present an algorithm that solves the collision detection problem at time $n + 1$ in expected constant time, provided the solution at time n is known. They carefully enumerate the type (vertex, edge, or face) of the features realizing the closest distance between the polyhedra under consideration and their connectivity with neighboring features. At a given time step, they perform a graph search around the previous solution's features. The search provably converges monotonically toward the solution. Empirical results confirm the efficiency claim.

A similar idea is presented by Baraff (1990), using separating hyperplanes. It is well known that disjoint convex figures can be separated by a hyperplane. As with the Roider method, the hyperplane is used as a witness of disjointness. The witness is found at initialization and is updated in sublinear expected time as the motion progresses.

In summary, although the methods of Section 2.2.1 are provably "optimal," it is not clear how efficient they are in practice for polyhedra with relatively small numbers of vertices. Empirical results show that iterative methods are very efficient on average, particularly if coherence is taken advantage of.

3. Collision Detection versus Collision Prediction

The collision detection techniques described in the last section are concerned with one pair of objects only. As is often mentioned in the literature, the basic collision detection check must be repeated a number of times, and this number grows quadratically with the number of moving objects in the space, regardless of the technique in use. In general, most of the computational time is spent checking for collisions between pairs of objects with low colliding urgency. The large number of pairs to be checked may easily overwhelm even the most efficient collision detector.

One way to reduce this potentially enormous load is to perform collision detection only for the pairs that are likely to collide within a given time horizon. Culley and Kempf (1986) introduced the notion of a "driving pair,"—namely, the pair of objects that are nearest neighbor among all pairs. This driving pair was used to determine the time increment until the next invocation of the collision detector.

This concept can be developed much further if the coherence of the world throughout time is explicitly exploited to schedule the pairs to be tested. Note that this use of coherence among object pairs is different from the expected coherence of the solution for a given object pair mentioned in Section 2.3.

Sensing and processing resources will be used more effectively if attention is allocated to pairs of objects that are near and/or fast approaching, rather than to those that are immobile, far, and/or separating. Figure 1 makes this clear.

Instead of a collision detector, we present here a collision predictor (Fig. 2). A collision predictor must either flag all collisions before they occur, thereby generating alarms, or indicate its inability to keep up with the dynamically changing environment. However, it may also predict collisions that will never occur. Such collisions are called false alarms. Obviously a good collision predictor should generate as few false alarms as possible.

As shown in Figure 2, the predictor may also act as a preprocessing filter for a "pure" or exact collision detector. In such an arrangement, the role of the predictor is to cull out the safe pairs from consideration by applying a simple and conservative test. An appropriate collision detector then determines whether the selected events (pairs) are true of false alarms.

By exploiting the coherence through time of which object pairs are in danger of colliding, the predictor acts as a scheduler for pair testing. Hence, one of its components is a specialized data structure that dynamically reflects the value of the urgency measure τ. DUA implements such a collision predictor. We describe its main components in the next subsection.

4. Outline of DUA for Collision Prediction

The algorithm proceeds as follows. The entire set of $O(N^2)$ pairs is considered at initialization time. A measure of urgency, τ, is computed for each pair as a function of instantaneous relative distance and velocity and relative velocity and acceleration bounds.

The set of pairs is partitioned according to the value of τ, and pairs are placed in equivalence classes of unequal cardinality. Class cardinalities grow as a geometric series, from those containing pairs that are deemed to be very

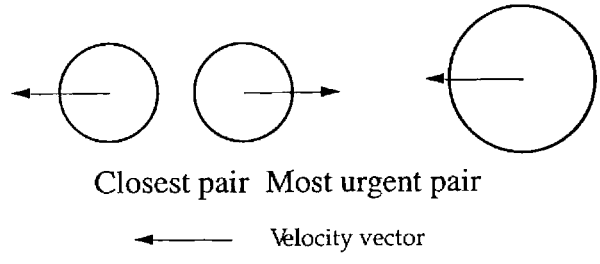

Fig. 1. The pair on the left is closer, but the one on the right is deemed more urgent because of its relative velocity vectors.

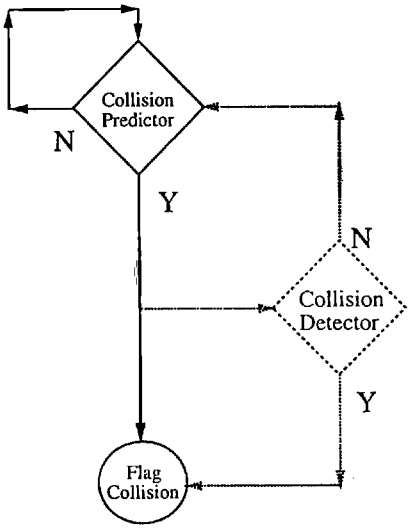

Fig. 2. Block diagram of collision prediction. The dashed lines indicate that the collision predictor can replace the detector entirely, at the expense of the possible generation of false alarms.

"urgent" to those containing pairs that are not. Classes are implemented as subarrays that are called buckets.

The algorithm then proceeds by testing more frequently the pairs belonging to the buckets of lower cardinality. Pairs percolate from bucket to bucket according to their time-evolving value of τ.

Data Structures

- Time-varying array $A(t)$ of maximum length $\binom{N}{2}$: Each element of A is a structure containing all the information pertaining to a given pair of objects. Within A, these pair structures are grouped into buckets of different size.
- Time-varying array $W(t)$ of maximum length $\lceil \log_2 \binom{N}{2} \rceil$. $W(t)$ is used to store and access the pairs under investigation at time t.

Algorithmic Steps

- Initial bucket allocation: For each pair, an underestimate τ of its possible collision time is computed, and a partial sort of the pairs is performed to initialize and populate the buckets.
- Each time sample: One element of each bucket is picked from A and transferred into W. The underestimate τ is then recomputed for each element of W. The structure W is completely sorted by the new value of τ and its elements are reinserted into the buckets. If a collision may occur, it is reported as an alarm.

The reinsertion can proceed in different manners, and the proposed scheme is as follows: the ith element of the sorted structure W is put in the corresponding ith bucket in the position of the previously withdrawn element. As time varies, the reinsertion location is not always the same. This way of reinserting the elements is equitable, in the sense that all elements of a bucket are evaluated an equal number of times.

5. Spatial Decomposition and Relative Motion

Consider a set $R = \{R_1, \ldots, R_N\}$ of N moving objects and a conservative covering $\mathcal{R}$ of the elements of R. $\mathcal{R}$ is conservative if and only if the closure of the elements of $\mathcal{R}$ is a superset of the closure of the elements of R. In the following the size of $\mathcal{R}$ is assumed to be $O(N)$.

To each $r \in \mathcal{R}$, a coordinate frame is associated, and a position function $\mathbf{f_r}(t)$ is defined on it, twice-differentiable with respect to time in a given interval $[0, T]$.

Let $P = \{(r_i, r_j) \mid r_i, r_j \in \mathcal{R}, i, j \in \{1, \ldots, N\}, i < j\}$. To each element $p = (r_1, r_2) \in P$, a relative position function $\mathbf{x_p}(t) = \mathbf{f_{r_2}}(t) - \mathbf{f_{r_1}}(t)$, a relative velocity function $\mathbf{v_p}(t) = d\mathbf{x_p}/dt$, and a relative acceleration function $\mathbf{a_p}(t) = d^2\mathbf{x_p}/dt^2$, all defined on $[0, T]$, are associated (Fig. 3).

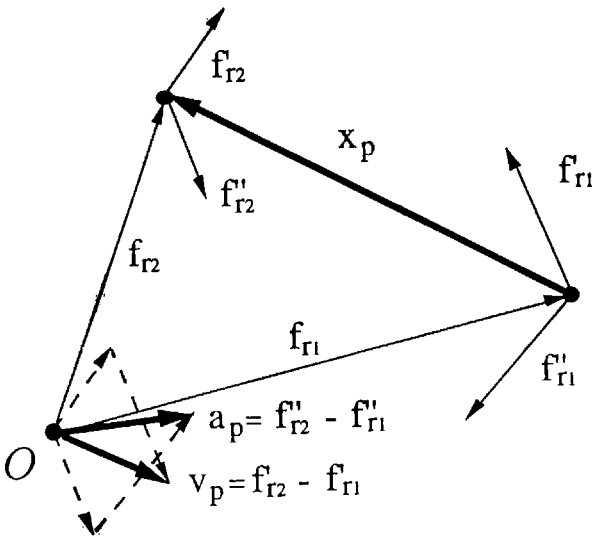

Fig. 3. Every pair of P defines a relative position, a relative velocity, and a relative acceleration vector function. O is the origin.

If p consists of two spheres of radii ϵ_1 and ϵ_2, a collision occurs at time t if

$$\|\mathbf{x_p}(t)\| \leq \epsilon_1 + \epsilon_2. \tag{1}$$

The minimum acceptable distance between elements of a pair p is this sum, augmented by a safety margin ϵ_p for errors made in estimating the relative position function and its derivatives.

Spheres are a simple but effective way to model arbitrary three-dimensional objects (Hayward and Aubry 1987; O'Rourke and Badler 1979), and they easily allow for a conservative representation. We argue that richer object primitives, such as polyhedra, are not warranted for a collision predictor. If necessary, these primitives should instead be used by the later-stage collision detector, whenever the predictor flags a collision for a given pair.

The following relative motion equation always holds for all pairs:

$$\mathbf{x_p}(t) = \left(\int_{t_0}^{t} \left\{ \int_{t_0}^{t} \mathbf{a_p}(s)\, ds + \mathbf{v_p}(t_0) \right\} ds \right) + \mathbf{x_p}(t_0), \tag{2}$$

$$\forall t, t_0 \geq 0 \quad \text{s.t.} \quad t_0 \leq t.$$

Given $\mathbf{x_p}(t_0)$, $\mathbf{v_p}(t_0)$, a bound V_p on $\|\mathbf{v_p}(t)\|$ and a bound A_p on $\|\mathbf{a_p}(t)\|$, we can use (2) to solve the minimization problem

$$\min_{t_0 \leq t \leq t_0 + \Delta t} \|\mathbf{x_p}(t)\|, \tag{3}$$

for all possible relative trajectories. The solution is a conservative relative distance for p, valid over the range $[t_0, t_0 + \Delta t]$. It indicates the smallest relative distance the pair can assume in the time interval, given the initial distance and velocity and the acceleration and speed bounds.

In general, these bounds are given by the physics of the collision problem we are solving. The acceleration bound may result from bounds on the active forces being exerted on the object, while the speed bound may be given by a measure of viscosity of the environment or other physical characteristics. In general, the relative bounds for the object pairs are the sum of the individual bounds on the objects themselves. If the bounds are uniform, then the relative bounds are twice the individual bounds.

Alternatively, (2) can be used to determine a lower bound for the smallest time that can elapse until a collision occurs. The object is then to find

$$\tau(p, t_0) = \min_{t_0 \leq t \leq t_0 + \Delta t} t \tag{4}$$

$$\text{s.t.} \quad \|\mathbf{x_p}(t)\| \leq \epsilon_p, \tag{5}$$

for all possible relative trajectories, subject to the same conditions as above. A sufficient condition for a pair not to generate a collision is

$$\tau(p, t_0) \geq \Delta t. \tag{6}$$

We choose $\tau(p, t_0)$ as our urgency function, and we show in the next section how to calculate it.

6. Calculating the Urgency Function τ

Setting $t_0 = 0, \mathbf{x_p}(0) = \mathbf{x_0}$ and $\mathbf{v_p}(0) = \mathbf{v_0}$ in (2), and otherwise dropping the reference to the particular pair p, we get

$$\mathbf{x}(t) = \left(\int_{t_0}^{t} \int_{t_0}^{t} \mathbf{a}(s)\, ds ds \right) + \mathbf{v_0}\, t + \mathbf{x_0}. \tag{7}$$

6.1. *Case without a Speed Bound*

The higher the acceleration of a body, the further it travels. Thus, it is intuitive that the minimum-time trajectory has the constant bound value A for the norm of its acceleration vector $\mathbf{a}(t)$. Variational calculus predicts that maximum range is achieved when the acceleration is constant in both norm and direction (Bryson and Ho 1987; Lawden 1963). Keeping the acceleration norm constant and maximum, (7) yields

$$\mathbf{x}(t) = \left(\int_{0}^{t} \int_{0}^{t} \mathbf{a}\, ds ds \right) + \mathbf{v_0}\, t + \mathbf{x_0} \tag{8}$$

$$= \frac{1}{2}\mathbf{a}t^2 + \mathbf{v_0}\, t + \mathbf{x_0} \tag{9}$$

where $\mathbf{a} = \mathbf{a}(t_0)$. From (9), the locus of attainable relative positions at time t is a sphere S centered at $\mathbf{x_0} + \mathbf{v_0}t$ and of radius $\frac{1}{2}At^2$. The boundary of S is the locus of relative positions attained when the relative motion has maximum constant acceleration, as in (9). Because the radius of S grows with the square of t and its center shifts linearly, any point in space is reachable given a long enough period of time (Fig. 4). From (5), a collision may occur during the time interval Δt if and only if

$$\exists t \in [0, \Delta t], \quad \|\mathbf{x}(t)\| \leq \epsilon. \tag{10}$$

From equation (9), the earliest time at which the above inequality can be verified occurs when $\mathbf{a}$ is of opposite direction with $\mathbf{x_0} + \mathbf{v_0}t$ (Fig. 5). Let $\mathbf{U}$ be a unit vector also of opposite direction from $\mathbf{a}$. Taking ϵ into account in equation (9), a collision occurs when

$$\mathbf{u}\epsilon = \frac{1}{2}\mathbf{a}t^2 + \mathbf{v_0}\, t + \mathbf{x_0}. \tag{11}$$

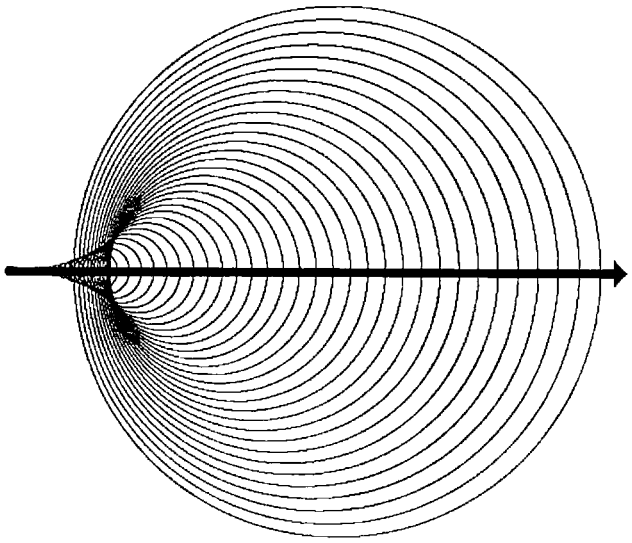

Fig. 4. Case with no speed bound. Two-dimensional projections onto a plane containing $\mathbf{v_0}$ *of the loci* $S(t)$ *of feasible positions at time* $t = \Delta t$ *for several values of* Δt. $\mathbf{v_0}$ *is shown as an arrow. The tail of the arrow lies on* $\mathbf{x_0}$.

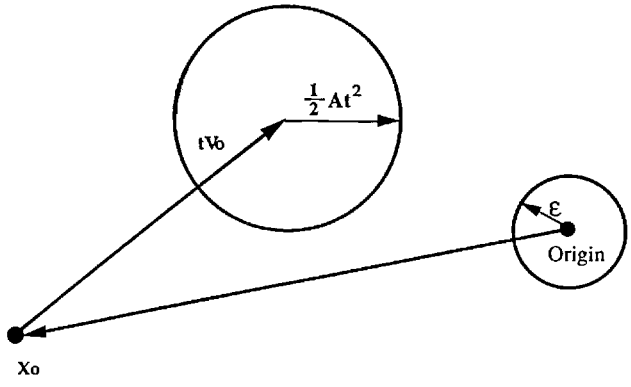

Fig. 5. Graphical illustration of the four vector terms of (10) at some time t *during the acceleration component.*

Rearranging and squaring (11) yields

$$\epsilon^2 - \epsilon t^2 \langle \mathbf{a}, \mathbf{u} \rangle + \frac{A^2}{4} t^4 = \|\mathbf{v_0}\|^2 t^2 + 2t \langle \mathbf{v_0}, \mathbf{x_0} \rangle + \|\mathbf{x_0}\|^2, \quad (12)$$

with $\langle \mathbf{a}, \mathbf{u} \rangle = A$. Solving for t, we find that the earliest possible collision time τ is the smallest positive root of the fourth-degree polynomial

$$-\frac{A^2}{4} t^4 + (\|\mathbf{v_0}\|^2 - \epsilon A) t^2 + 2\langle \mathbf{v_0}, \mathbf{x_0} \rangle t + (\|\mathbf{x_0}\|^2 - \epsilon^2) = 0. \quad (13)$$

6.2. *Case With a Speed Bound*

We now include a bound V on the relative speed. As above, we represent the locus of attainable positions at time t by a closed geometric figure S' (Fig. 6). A collision may occur if and only if the origin is in the closure of S'. Of course, the results of the previous section hold over any given time interval Δt, if the maximum speed is never achieved over it.

Suppose that the maximum speed is achieved during a certain time interval. Assuming the initial relative speed is less than V, the relative motion generating the earliest possible collision is split into:

1. An acceleration component, during which the movement has maximum constant acceleration $\mathbf{a}$, of magnitude A, until the speed V is reached.
2. A velocity component, during which the movement has zero acceleration and maximum speed $V = \|\mathbf{v}\|$.

This result is quite intuitive but is left to be proved.

We first determine T, the time at which the relative speed reaches V. Differentiating (9), we get $\mathbf{v}(t) = \mathbf{a}t + \mathbf{v_0}$. Hence,

$$\|\mathbf{v}(t)\| = \|\mathbf{a}t + \mathbf{v_0}\|. \quad (14)$$

Solving (14) for $\|\mathbf{v}(T)\| = V$, we obtain a second-degree equation with roots of opposite signs. By choosing T positive, we get

$$T = \frac{-\langle \mathbf{a}, \mathbf{v_0} \rangle + \sqrt{A^2(V^2 - \|\mathbf{v_0}\|^2) + \langle \mathbf{a}, \mathbf{v_0} \rangle^2}}{A^2}. \quad (15)$$

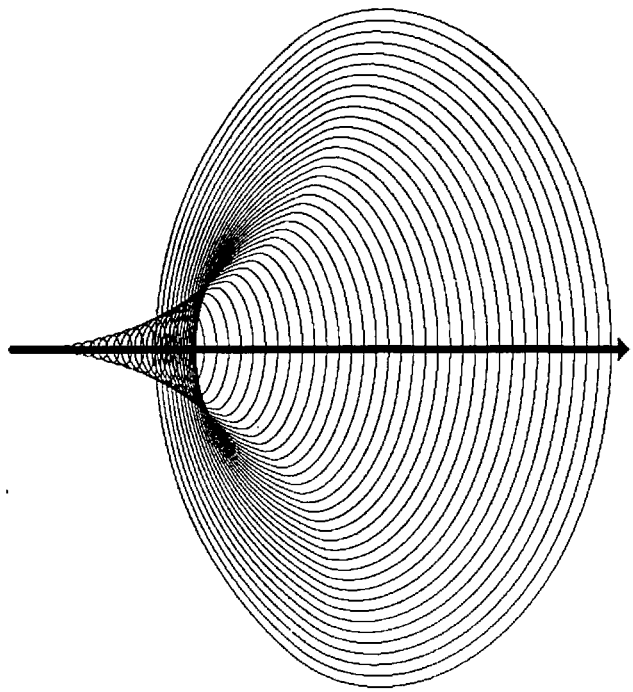

Fig. 6. Case with a speed bound. Two-dimensional projections onto a plane containing $\mathbf{v_0}$ *of the loci* $S(t)$ *of feasible positions at time* $t = \Delta t$ *for several values of* Δt. $\mathbf{v_0}$ *is shown as an arrow. The tail of the arrow lies on* $\mathbf{x_0}$.

The speed bound is encountered at time T. Of course, if $\Delta t \leq T$, it will not be reached, and we revert back to the case with no speed bound.

Note that T varies with the direction of $\mathbf{a}$. For example, from (15), we can isolate the two special cases where $\mathbf{a}$ is collinear with $\mathbf{v_0}$. This yields $T = (V - \|\mathbf{v_0}\|)/A$ if $\mathbf{a}$ and $\mathbf{v_0}$ point in the same direction, and $T = (V + \|\mathbf{v_0}\|)/A$ if they point in opposite directions.

Summing up both motion components, we find that τ is the smallest positive solution, greater or equal than T, of

$$\mathbf{x}(t) = (\mathbf{x_0} + \mathbf{v_0}T + \frac{1}{2}\mathbf{a}T^2) + \mathbf{v}(T)(t - T). \quad (16)$$

Because $\mathbf{v}(T) = \mathbf{v_0} + \mathbf{a}T$, (16) becomes

$$\mathbf{x}(t) = \mathbf{a}\left(Tt - \frac{T^2}{2}\right) + \mathbf{v_0}t + \mathbf{x_0}. \quad (17)$$

From (5), a collision may occur if and only if

$$\exists t \in [T, \Delta t], \qquad \|\mathbf{x}(t)\| \leq \epsilon. \quad (18)$$

The earliest time at which the above inequality can be verified occurs when $\mathbf{a}$ is collinear with $\mathbf{x_0} + \mathbf{v_0}t$ and of opposite direction. Equations (17) and (18) yield

$$\|\mathbf{v_0}t + \mathbf{x_0}\| - A(Tt - \frac{T^2}{2}) \leq \epsilon. \quad (19)$$

Rearranging terms and squaring and solving for t, we obtain a second-degree equation

$$(\|\mathbf{v_0}\|^2 - A^2T^2)t^2 + (T^3A^2 + 2\langle \mathbf{x_0}, \mathbf{v_0}\rangle - 2AT\epsilon)t + (\|\mathbf{x_0}\|^2 + \epsilon AT^2 - \epsilon^2 - \frac{A^2T^4}{4}) = 0. \quad (20)$$

The earliest possible collision time τ is the smallest positive root of the previous second-degree equation. We have then completely specified τ as a function of T. However, because T is itself a function of $\mathbf{a}$, whose direction is not known a priori, τ is still not completely determined. We need an additional equation to convey the fact that the two conditions on the motion components are verified only when $\mathbf{v}(T)$ points toward the origin. Hence, $\mathbf{v}(T)$ and $\mathbf{x}(T)$ are collinear:

$$\mathbf{x}(T) \times \mathbf{v}(T) = \left(\frac{1}{2}\mathbf{a}T^2 + \mathbf{v_0}T + \mathbf{x_0}\right) \times (\mathbf{a}T + \mathbf{v_0}) \quad (21)$$

$$= 0. \quad (22)$$

Expanding (22), we obtain a second-degree equation in T. This equation is solved for T for the smallest positive root, and the root is equated with the one found in equation (15) to get an expression relating $\mathbf{a}$ to the known values $A, V, \mathbf{v_0}, \mathbf{x_0}$. This expression can be quite complex in the general case and is more easily solved using numerical methods.

The net effect of including the velocity bound in our calculations is to reduce the loci of feasible relative positions attainable after a given time interval, as can be observed from Figures 4 and 6. This is desirable because it makes the function less "conservative." Namely, the more refined the urgency function, the less conservative it is and the fewer the false alarms. As expected, however, we find that more involved computations are the price to pay for less conservative measures.

Referring to Figure 2, it is clear that the more complex the urgency function is, the more work gets shifted away from the collision detector onto the collision predictor. How complex the urgency function should be, therefore, depends not only on the available dynamic model, but also on the desired distribution for the computational resources.

6.3. *Active and Inactive Pairs*

As described, the urgency measure τ does not take into account special relationships that may exist between objects forming pairs. The most obvious of these relationships is that of pairs consisting of elements belonging to the same rigid object. Such pairs cannot generate collisions and should not be included in the data structure A, thus eliminating them from consideration at the onset of the algorithm.

Other relationships may also arise as motion progresses. For example, when a robot is required to graze or contact objects, DUA should alter its behavior. The considered pairs should be ignored as long as contact is desired. This is not only for reasons of efficiency, but also because desired low distance values for the pairs must be maintained.

This concludes the description of the urgency function. The next section gives a detailed description of DUA.

7. The Dynamic Urgency Algorithm

7.1. *Spatiotemporal Sequencing Data Structure*

Given a time interval Δt, we organize the $O(N^2)$ pairs into a time-varying sequencing structure $A(t)$ that indicates the urgency with which those pairs should be checked for a possible collision. $A(t)$ serves the following purposes:

1. To determine the most likely pairs that may generate a collision. The algorithm tests these pairs more often.
2. To sequence the collision checks among elements of P that are equally likely to generate a collision, thereby reducing the number of computations.

7.1.1. *Initial Bucket Partitioning*

For each pair we calculate τ, which we defined in the last section as a measure of collision urgency. We first use τ to perform initial collision checks and flag for possible collisions for all pairs. We describe these tests in Section 7.2.4.

Then, using τ as a key, we partition the pairs into buckets of unequal cardinality, where each bucket groups pairs of similar collision urgency.

Here we propose a binary partitioning scheme in which the cardinalities of neighboring buckets differ by a power of 2. Other partitioning schemes with similar size relationships—based, for example, on the Fibonacci series—may also be used. The crucial property is that the size of adjacent buckets grows as a geometric series. This guarantees that the number of buckets grows as a logarithmic function of the original input size.

Suppose the cardinality of the set of pairs P is M and that $\lceil \log_2 M \rceil + 1 = L$. For efficiency, only a partial sort of the pairs is performed, using τ as the key, and these are inserted into the array $A(0)$, such that

$$\begin{aligned} &\forall k \in \{1,\ldots,(L-1)\}, \\ &\forall i \in \{1,\ldots,2^k - 1\}, \\ &\forall j \in \{2^k,\ldots,M\}, \\ &\tau(A(0)[i]) \leq \tau(A(0)[j]). \end{aligned} \tag{23}$$

In other words, the value of the key τ for any of the first $2^k - 1$ elements must be less than that of any of the remaining elements of the array. A partial sort can be implemented using a median algorithm, which is faster than a general sorting algorithm (see Section 7.3). Because $A(t)$ only implements equivalence classes, the relative values of τ for pairs belonging to the same class, or bucket, are inconsequential.

Each bucket $\{A(t)[2^{i-1}],\ldots,A(t)[2^i - 1]\}$ of pairs is denoted by B_i. Hence, the cardinality of B_i is 2^{i-1}, and there are L buckets. Note that bucket B_1 contains one pair and only the first $M - 2^{\lfloor \log_2 M \rfloor}$ elements of bucket B_L are valid pairs, the rest being filled with dummy variables. We say that bucket B_i is larger than bucket B_j if and only if $i > j$ (B_i contains more elements than B_j).

7.1.2. *Bucket Selection and Update*

After the initial bucket construction, we select at every time interval one pair from each bucket only, for a total of L pairs. The pairs are temporarily held in $W(t)$. Because there are fewer pairs in the smaller buckets, these pairs will be tested more often: this implements purpose 1 above. Furthermore, the computational load will remain constant at each step, because the number of selected pairs is constant: this implements purpose 2.

The justification for the sequencing scheme should now be clear: because τ indicates that the pairs from the larger buckets are less likely to collide than the ones from the smaller buckets, they need not be checked as often. Figure 7 shows the selection process through several time steps.

7.2. *Algorithm's Description*

We now describe a full algorithm iteration. We assume that $A(t_0)$ has been properly initialized such that the pairs are partially sorted with respect to the proximity measure τ, as explained in Section 7.1.1. The algorithm is described in pseudo-code in the Appendix.

At each iteration, we perform the following operations:

1. Select pairs from $A(t)$ and insert them into $W(t)$.
2. Evaluate τ for all pairs of $W(t)$.
3. Sort $W(t)$ using the new value for τ as the key.
4. Using the new order, check whether any pair of $W(t)$ may generate a collision before its next update. This depends on the cardinality of the bucket in which the pair is to be reinserted, which in turn depends on the order of $W(t)$.
5. Reinsert the elements of $W(t)$ back into $A(t)$.

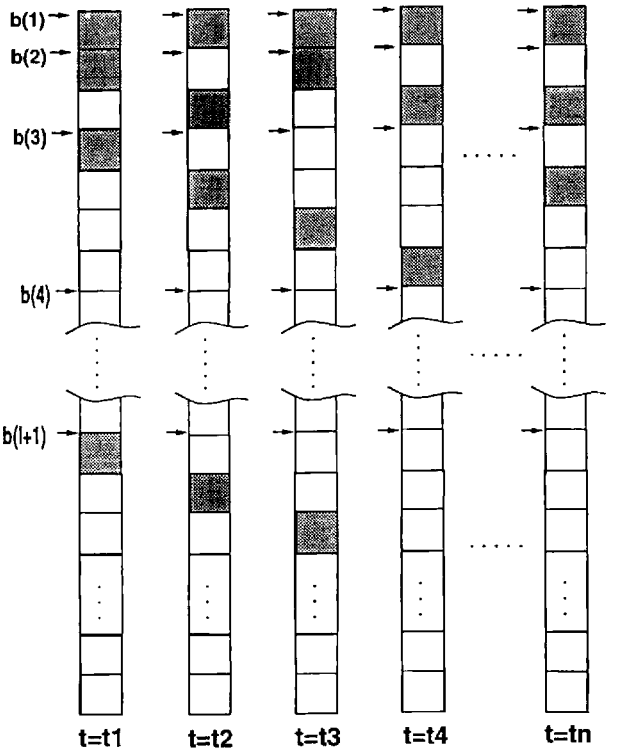

Fig. 7. Array $A(t)$ at different times t_i. The buckets' upper limits are indicated by the arrows, and the array positions selected for insertion into $W(t)$ are shown by the shaded regions.

7.2.1. Pair Selection

Let $W(t)$ be the array of pairs to be examined at a given time $t = (\Delta t)n$. $W(t)$ is populated in the following manner: each element $W(t)[i]$ is the current head of the corresponding circular queue of bucket B_i. This is written as:

$$W(t) = \langle A(t)[1],\ A(t)[2^1 + (n \bmod 2^1)],\ \ldots,\ A(t)[2^i + (n \bmod 2^i)],\ \ldots,\ A(t)[2^L + (n \bmod 2^L)]\rangle. \quad (24)$$

Note that the last element of the above set does not always exist for all values of n.

7.2.2. Evaluation of τ

For each pair p in $W(t)$, we read in the current values for $\mathbf{x_p}(t)$ and $\mathbf{v_p}(t)$. As discussed in the beginning of Section 2.2, these values are assumed to be available in real time, either from sensor readings or from trajectory computations. We use these values to recalculate τ using either of the methods described in Section 6 or any other urgency function.

7.2.3. Sorting W

The elements of $W(t)$ are sorted according to the most recent value of τ that was calculated in the previous step. The new positions of the pairs within $W(t)$ now reflect the current relative urgency for those pairs.

7.2.4. Performing the Collision Checks

The new value for τ is then used to determine whether a collision should be flagged. If p is a pair under consideration, p cannot generate any collision during the time interval τ.

Let $p = W(t)[i]$, where $W(t)$ has been updated as explained in the previous subsection. From Figure 7 it is clear that if p is reinserted into $A(t)$, it will next be selected after $2^i - 1$ iterations.

Suppose now that the following holds:

$$\tau > 2^i \Delta t. \quad (25)$$

Equation (25) means that p is guaranteed not to generate any collision for an amount of time equal to its update time. This is because p will be updated after a number of time increments exactly equal to the cardinality of the bucket into which p is to be reinserted. To guarantee that p will not collide until the next update is completed, we need a slightly more restrictive condition. This condition must take into account the time needed to complete the current iteration and the time needed to complete the next iteration in which p will be selected. Hence, (25) becomes

$$\tau > (2^i + 1)\Delta t. \quad (26)$$

Condition (26) expresses a sufficient condition for p to be reinserted into $A(t)$ while guaranteeing that any collision generated by p will be flagged by subsequent iterations. Hopefully, a large number of pairs will satisfy it. If (26) does not hold, we generate an alarm for pair p. As seen in Figure 2, whether a collision is actually about to occur can be checked by an exact collision detection module, if such a module is available.

7.3. Complexity Analysis

The method complexity is driven by the number of pairs. There are originally $M = O(N^2)$ pairs, where N is the number of spheres induced by the covering $\mathcal{R}$.

The original preprocessing partitioning step using τ takes $O(N^2)$. This can be seen considering that $A(t_0)$ is obtained by repeated application of the linear-complexity median-finding algorithm over an exponentially decreasing set, the size of the input being $O(N^2)$ (Aho et al. 1982).

At every time increment, we need to consider one pair from each bucket. There are $O(\log N^2) = O(\log N)$ buckets. Updating the array using a true sorting algorithm has a complexity of $O(\log N \log\log N)$. Note that if a partial sort such as that performed at preprocessing time is chosen, the on-line complexity drops to $O(\log N)$.

7.4. Completeness

We say that the algorithm is complete if we can guarantee that all collisions are flagged. Let Q be the time necessary to completely process the $O(\log N)$ pairs in the main loop of the algorithm. A sufficient condition for the algorithm to be complete is that

$$Q < \Delta t. \quad (27)$$

If Δt is chosen too small, (27) shows that the processor may not be able to perform all the overhead associated with every iteration: memory accesses, recomputation of τ, and resorting of the pairs. Thus collisions may be missed and DUA is not complete.

However, if Δt is chosen too large, (26) shows that an excessive number of "collisions" may be flagged. Most of them would be false alarms. If an exact detection collision module is available, an inordinate number of false alarms would overload it, leading to system degradation. If, instead of being post-processed, alarms cause preventive shutdown of the system, an excessive number of false alarms would also be very undesirable.

Even though we have not yet attempted to calculate an optimal time step, choices can be made based on intuitive arguments. For example, Culley and Kempf (1986) determine the time interval length between static collision checks as $t = d_{\min}/V_{\max}$, where $d_{\min}$ is the current minimum relative distance among all object pairs and $V_{\max}$ is the maximum relative speed between the objects.

In the next section, we present simulation results demonstrating the effectiveness of DUA and discuss its limitations. Rather than choosing an a priori optimal value for Δt, as would have to be done in a real-time application, we elect to let DUA process pairs as fast as the computational resources allow it, thus empirically establishing Δt as $Q + 1$ instruction cycles. This guarantees that, once initialized, the simulation satisfies (27).

8. Experiments

As explained in Section 5, we assume that three-dimensional moving objects are modeled as collections of spheres. Here, the spheres are constrained to move in a cube. A collision occurs when the distance between the centers of any two spheres falls below a certain threshold. The motion of the spheres is generated by assigning random but bounded values to the acceleration vectors, thus generating relative position and velocity values for all pairs. This represents a worst-case scenario as far as the coherence of the system is concerned. In most practical situations, we would expect a larger amount of coherence to be present and hence an even better performance of DUA.

The problem that DUA addresses presents a large number of parameters. We elected to perform tests showing the effectiveness of the scheduling mechanism and of the urgency function.

8.1. *Description of the Tests*

For purposes of comparison, we implemented three algorithms using Common Lisp and ran them on a SPARC 470:

1. DUA: The algorithm discussed in this article, with the urgency function set as described in Section 6.1 (no speed bound).
2. SA(τ) (Select-All): The simple algorithm referred to in Section 1: at every time step, we recalculate τ and check for possible collisions for all pairs. No sequencing structure or sorting mechanism is necessary.
3. SA(d): Same as SA(τ), except that the urgency function is the ordinary Euclidean distance.

8.1.1. *Effectiveness of DUA's Sequencing Technique*

We ran DUA and SA(τ) for sets of 5, 10, 15, 20, 30, and 40 spheres over 25, 35, 50, 65, and 75 iterations. We use the naive SA(τ) algorithm as a benchmark for assessing DUA's effectiveness in terms of its:

Efficiency: Efficiency is measured in CPU cycles per algorithm iteration, as a function of the number of spheres.

Precision: Precision measures the number of false alarms DUA generates. The smaller the number, the more precise the algorithm. Because the simulation methodology guarantees completeness (see the comment at the end of Section 7.4), we know that DUA flags all collisions. However, the more precise DUA is, the fewer false alarms it generates.

Urgency Belief: DUA maintains a hierarchy of its belief of pair urgency through the structure $A(t)$. As time progresses, the belief may become flawed (for example, if the world is too incoherent for DUA to update its hierarchy fast enough).

8.1.2. *Efficiency*

Not surprisingly, SA(τ) requires much more CPU time per iteration. SA(τ) exhibits a growth quadratic in the number of spheres, whereas DUA exhibits a sublinear growth in the same number. The results are charted in Figure 8 and further detailed in Table 1. In Figure 9, the experimental computational growth of DUA is seen to closely follow the theoretical $O(\log n \log\log n)$ growth.

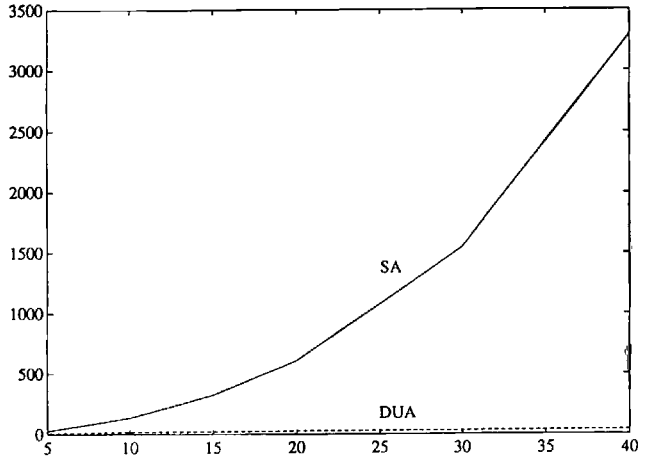

Fig. 8. Experimental running times for DUA and SA in CPU cycles, with respect to the number of spheres present in the scene.

Table 1. CPU Cycle Time for DUA and SA

No. of spheres	5	10	15	20	30	40
DUA	12.01	18.74	22.42	26.65	31.14	39.69
SA	29.53	135.133	323.90	607.45	1543.02	3294.18

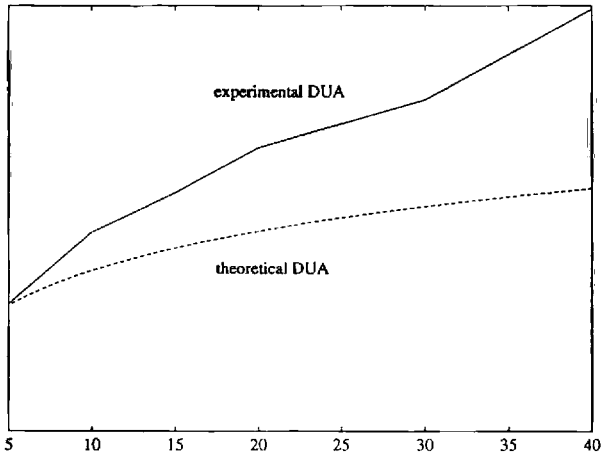

Fig. 9. Comparison of the theoretical computational growth (dotted line) of DUA versus its experimental growth (solid line), in scaleless units.

8.1.3. Precision

In the tests we ran, the number of collisions between the spheres turned out to be zero in all cases. Hence, all the alarms generated because equation (26) did not hold were false alarms. The numbers of such alarms for both DUA and SA(τ) are charted in Figure 10 and further detailed in Table 2.

Although DUA does generate false alarms, we found the results quite convincing when compared with SA(τ). The absolute number of alarms is always lower for DUA, and its growth appears to be sublinear with respect to the number of spheres. In contrast, the number of false alarms for SA(τ) seems to be following at least a quadratic growth.

A repetitive alarm is an alarm for a given pair that is flagged at successive iterations. It is clear that repetitive alarms should be counted only once, because they occur only because of our artificial time discretization: Objects collide only once, but the collision predictor "flags" a collision as long as the pair's urgency measure remains too low (i.e., low earliest time to collision). We also assume that in a more realistic situation, corrective action is taken to remedy alarms, thereby preventing repetitive alarms. SA(τ) generates many repetitive alarms, and they were filtered out from the results of Figure 10. DUA also generates repetitive alarms, but because pairs are normally not selected repeatedly over neighboring iterations, filtering DUA's repetitive alarms is a more subtle task. In the result we present for DUA, we did not attempt to filter out the repetitive alarms. Hence, actual results for DUA are better than those shown.

8.1.4. Urgency Belief

We want to ascertain whether DUA's hierarchy is accurate. For doing so, we chose an extremely simple test. At every iteration, we check whether DUA's belief of the most urgent pair is correct. SA(τ) maintains an accurate hierarchy of urgency and we used its hierarchy as a reference against which to check DUA's.

We varied the number of iterations from 25 to 250, while keeping the number of spheres at 10. For more accuracy, we applied the algorithms for 10 different random movement sequences for each test. We then computed statistics on DUA's belief of the most urgent pair. Overall, we found that DUA correctly predicted the most urgent pair 67% of the time, a rather high agreement measure (Table 3).

We also tested the effectiveness of the urgency function we used. We applied the same test—of the most urgent pair—to compare the effectiveness of SA(τ) against that of SA(d). There, the number of agreements between the two algorithms was 20% (Table 3). We believe this low number vindicates our use of an urgency function that incorporates knowledge about the pairs' positional derivatives. Figure 2 gave a simple case well handled by the urgency function $\tau(p,t)$ but not by $d(p,t)$. We also visually verified the prevalence of such cases on graphical displays, where the color of the spheres was used as an indicator of the magnitude of τ.

8.2. Summary of Test Results

By comparing the performance of DUA to a naive exhaustive collision-prediction algorithm, we found it performed significantly better. The growths of both the computational cost and the number of false alarms as a function of the number of spheres decreased from approximately quadratic to sublinear. These results are correlated, because the precision of DUA is linked to its ability to process information speedily.

Table 2. Number of False Alarms of DUA and SA.*

No. of spheres	5	10	15	20	30	40
DUA:Δt	0.2	0.31	0.37	0.44	0.52	0.66
DUA: No. of iterations						
25	0	52	**96**	124	164	**196**
35	**0**	**75**	133	174	226	**270**
50	**0**	**104**	189	248	319	**379**
65	**0**	**134**	246	323	410	**485**
75	0	153	284	370	471	557
SA:Δt	0.48	2.23	5.33	10.12	25.31	53.75
SA: No. of iterations						
25	0	88 (102)	496 (1203)	897 (3581)	2193 (8317)	3988 (15489)
35	0	109 (125)	655 (1516)	1144 (4955)	2822 (11462)	5118 (21593)
50	0	127 (146)	914 (1944)	1538 (6937)	3746 (16076)	6717 (30649)
65	0	153 (172)	1199 (2398)	1988 (8909)	4763 (20822)	8539 (39892)
75	0	177 (196)	1368 (2671)	2315 (10168)	5433 (23927)	9668 (45958)

*In the case of SA, the first number is the count with repetitive alarms excluded, while the number in parentheses includes all alarms.

Table 3. Agreement of DUA With SA(τ) and of SA(τ) With SA(d) for the Classification of the Most Urgent Pair in the Case of 10 Spheres.*

No. of iterations	25	50	75	100	150	200	250	Overall
SA(τ) versus DUA								
Mean	63.6	64.2	67.87	70.9	68.73	68.4	64.56	66.89
Standard deviation	27.29	19.58	19.30	19.61	18.46	16.61	20.94	20.26
SA(d) versus SA(τ)								
Mean	16.8	15.4	19.73	19.4	21.67	24.0	21.6	19.8
Standard deviation	17.66	13.98	10.73	8.56	9.25	8.92	9.13	11.18

*The sample mean and the standard deviation of the number of agreements between the algorithms are shown.

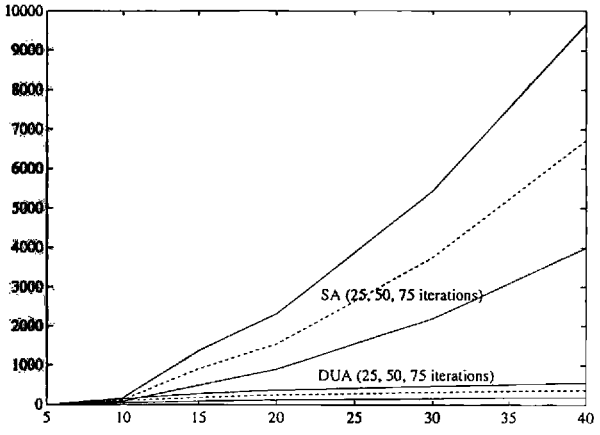

Fig. 10. The number of false alarms charted against the number of spheres is shown for 25, 50, and 75 iterations for both DUA and SA(τ). Repetitive alarms are only counted once.

According to an arbitrary simple measure, that of the belief of the most urgent pair, we showed that the sequencing mechanism maintains a high degree of agreement (2/3) with the current state of the world. With the same simple measure, we also showed that using the prediction function generally benefits (four times out of five) from the incorporation of the positional derivatives in its computation.

9. Conclusions

We presented a general, gracefully degrading sequencing scheme for computational problems characterized by large input sizes, limited resources, and a dynamically changing environment by assuming that the rate of change of the environment obeys some bounds. We also generated some experiments to illustrate the validity of the approach.

We called the sequencing scheme DUA, for dynamic urgency algorithm, and we applied it to the problem of collision prediction for many moving objects. In particular, we introduced an urgency measure based on the objects' positions and positional derivatives. The measure represents the shortest possible time before a collision, given the available information. We also showed how it could be used to implement a collision prediction behavior.

Given an environment made up of N spheres, the naive method of testing all pairwise interactions at each time sample has a complexity of $O(N^2)$. In contrast, DUA has an initial classification step of complexity $O(N^2)$ and a steady-state complexity of $O(\log N \log\log N)$ per time sample. We gave a condition for DUA to guarantee the flagging of all collisions (the completeness property), but because of its generality we could not specify when such a condition could be guaranteed.

Within the general framework of DUA, many variations are possible. Examples of critical implementation considerations are:

Serial computing: If a serial computer is used, larger buckets can be assigned to memories of increasing capacity and decreasing access speed. In such a context, the pair sampling algorithm mimics scheduling algorithms used in time-sharing operating systems.

Parallel computing: In a parallel computing environment, a computing unit could be assigned to each bucket, because the computing requirements are the same for all buckets. Further, higher speed links between the central unit, whose job is to manage alarms, and the larger buckets could be arranged, thus reflecting the urgency hierarchy right in the hardware configuration. A more extreme case would be for each pair to be assigned its own processing unit. In this case no bucket scheduling scheme is necessary, but an urgency hierarchy would still be desirable to reduce the number of connections between the central collision flag manager and the $O(N^2)$ pairs. Finally, the collision predictor and detector of Figure 2 can easily be implemented on separate and communicating computing units.

Partitioning: As mentioned in Section 7.1.1, the size of buckets can follow other geometric series, such as the Fibonacci series. The partitioning strategy may in general have to be tuned to the particular application.

Environment Knowledge: The urgency function τ we developed is by no means restrictive. The system's ability to acquire available information and process it in real time should determine which function DUA uses. It is clear that the more information is available, the more accurate τ is.

Appendix: Dynamic Urgency Algorithm

Initialization

$t = t_0$;
for-each $p \in P$
  $\mathbf{x_0}(p), \mathbf{v_0}(p)$:=READ ; calculate $\tau(p, t_0)$;
end-for-each
$\mathcal{P} = P$;
while(CARDINALITY($\mathcal{P}$) > 1)
  m :=MEDIAN($\mathcal{P}$) with respect to τ;
  $\mathcal{P}' := \{p \in \mathcal{P}, \tau(p, t_0) > \tau(m, t_0)\}$;
  Fill A bottom-up with $\mathcal{P}'$;
  $\mathcal{P} = \mathcal{P} \setminus \mathcal{P}'$;
end-while

Main Loop

$t := t_0$;
for-ever
  $W(t)$:=SELECT(P);
  for-each $p \in W(t)$
    $\mathbf{x_0}(p, t), \mathbf{v_0}(p, t)$:=READ;
    calculate $\tau(p, t)$;
  end-for-each
  $W(t)$:=SORT($W(t)$) by $\tau(p, t)$;
  for-each $p = W[i] \in W(t)$
    if $(\tau(p, t) < (2^i + 1)\,\Delta t)$
      Report a possible collision;
    end-if
  end-for-each
  $t := t + \Delta t$;
end-for-ever

Acknowledgments

Work described in this article has been supported in part by a research contract with Spar Aerospace Ltd., Toronto, Canada, by a grant from the Natural Sciences and Engineering Research Council of Canada and Fonds Formation des Chercheurs et l'Aide à la Recherche, Québec. Additional support has been provided by the Centre de Recherche en Informatique de Montréal from research conducted at McGill University on behalf of the Institut de Recherche d'Hydro-Québec, Montréal.

References

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. 1982. *Data Structures and Algorithms*. Reading, MA: Addison-Wesley.

Avnaim, F., Boissonnat, J.-D., and Faverjon, B. 1988 (May, Philadelphia). A practical exact motion planning algorithm for polygonal objects amidst polygonal

obstacles. *Proc. IEEE International Conference on Robotics and Automation,* pp. 1656–1661.

Baraff, D. 1990. Curved surfaces and coherence for non-penetrating rigid body simulation. *Computer Graphics* 24(4):19–38.

Barford, D. A. 1989 (November, Tempe, AZ). Fast detection of collisions between moving bodies. *Proc. SIAM Conference on Geometric Design.*

Bobrow, J. E. 1989. A direct minimization approach for obtaining the distance between convex polyhedra. *Int. J. Robot. Res.* 8(3):65–76.

Boyse, J. W. 1979. Interference detection among solids and surfaces. *Comm. ACM* 22(1):3–9.

Brooks, R. A. 1983. Solving the find-path problem by good representation of free space. *IEEE Trans. Sys. Man Cybernet.* 13:190–197.

Brooks, R. A., and Lozano-Pérez, T. 1983 (Karlsruhe, Germany). A subdivision algorithm in configuration space for findpath with rotation. *Proc. International Joint Conference on Artificial Intelligence*, pp. 799–806.

Bryson A. E., and Ho, Y.-C. 1975. *Applied Optimal Control.* New York: Halsted Press.

Cameron, S. 1985 (March, St. Louis). A study of the clash detection problem in robotics. *Proc. IEEE International Conference on Robotics and Automation,* pp. 488-493.

Canny, J. 1988. *The Complexity of Robot Motion Planning.* Cambridge, MA: MIT Press.

Culley, R. K., and Kempf, K. G. 1986 (April, San Francisco). A collision detection algorithm based on velocity and distance bounds. *Proc. IEEE International Conference on Robotics and Automation,* pp. 1064–1069.

Dobkin, D. P., and Kirkpatrick, D. G. 1983. Fast detection of polyhedral intersection. *Theoretical Comp. Sci.* 27:241–253.

Dobkin, D. P., and Kirkpatrick, D. G. 1985. A linear algorithm for determining the separation of convex polyhedra. *J. Algorithms* 6:381–392.

Faverjon, B., and Tournassoud, P. 1988 (December). A practical approach to motion-planning for a manipulator with many degrees of freedom. Rapport de Recherche 951, INRIA, Sophia Antipolis, France.

Gilbert, E. G., and Foo, C. P. 1990 (May, Scottsdale, AZ). Computing the distance between smooth objects in three dimensional space. *Proc. IEEE International Conference on Robotics and Automation,* pp. 158–163.

Gilbert, E. G., and Hong, S. M. 1990 (May, Scottsdale, AZ). A new algorithm for detecting the collision of moving objects. *Proc. IEEE International Conference on Robotics and Automation,* pp. 8–14.

Gilbert, E. G., and Johnson, D. W. 1985. Distance functions and their application to robot path planning in the presence of obstacles. *IEEE Trans. Robot. Automation* 1(1):21–30.

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

Hayward, V., and Aubry, S. 1987 (October, Cambridge, MA). On describing a robotic scene. *Proc. SPIE Conference on Intelligent Robots and Computer Vision: Sixth in a Series* Bellingham, WA: SPIE, pp. 525–532.

Hurteau, G., and Stewart, N. F. 1988. Distance calculation for imminent collision indication in a robot system simulation. *Robotica* 6:47–51.

Kawabe, S., Okano, A. K., and Shima, K. 1988. Collision detection among moving objects in simulation. In Bolles, R. C., and Roth, B. (eds.): *Fourth International Symposium on Robotics Research.* Cambridge, MA: MIT Press, pp. 489–496.

Lawden, D. F. 1963. *Optimal Trajectories for Space Navigation.* London: Butterworths.

Lin, M. C., and Canny, F. F. 1991 (April, Sacremento, CA). A fast algorithm of incremental distance calculation. *Proc. IEEE International Conference on Robotics and Automation,* pp. 1008–1014.

Lozano-Pérez, T. 1983. Spatial planning: A configuration space approach. *IEEE Trans. Computers* 32:108–120.

Lozano-Pérez, T. 1987. A simple motion-planning algorithm for general robot manipulators. *IEEE Trans. Robot. Automation* 3(3):224–238.

Lozano-Pérez, T., and Wesley, M. A. 1979. An algorithm for planning collision-free paths among polyhedral obstacles. *Comm. ACM* 22:560–570.

Luenberger, D. G. 1969. *Optimization by Vector Space Methods.* New York: Wiley.

O'Rourke, J., and Badler, N. 1979 (July). Decomposition of three-dimensional objects into spheres. *IEEE Transactions of Pattern Analysis and Machine Intelligence,* 1(1):295–305.

Powell, M. J. D. 1985. On the quadratic programming algorithm of Goldfarb and Idnani. *Math. Program. Study* 25:46–61.

Red, W. E. 1983. Minimum distances for robot task simulation. *Robotica* 1:231–238.

Roider, B., and Stifter, B. 1987. Collision of convex objects. In Davenport, J. H. (ed.): *Eurocal.* (LNCS). New York: Springer-Verlag, pp. 258–259.

Stifter, S. 1988. A generalization of the Roider method to solve the robot collision problem. In Gianni, P. (ed.): *Proc. International Symposium on Symbolic and Algebraic Computation.* New York: Springer-Verlag, pp. 332–343.