Interactive Collision Detection

Philip M. Hubbard
Department of Computer Science
Box 1910, Brown University
Providence, RI 02912

Abstract

Collision detection and response can make a virtual-reality application seem more believable. Unfortunately, existing collision-detection algorithms are too slow for interactive use. We present a new algorithm that is not only fast but also interruptible, allowing an application to trade quality for more speed. Our algorithm uses simple four-dimensional geometry to approximate motion, and sets of spheres to approximate three-dimensional surfaces. The algorithm allows a sample application to run 5 to 7 times faster than it runs with existing algorithms.

1 Introduction

The physical world we live in is filled with solid objects. When solid objects collide, they do not penetrate one another (unless they flex or break). A computer simulation of a virtual world will seem more believable if it exhibits this property. We use the term *collision-handling algorithm* for the part of the simulation system that enforces the solidness of objects.

Researchers have developed a variety of collision-handling algorithms. The typical algorithm has two parts. The *detection algorithm* determines whether simulated objects would penetrate if the collision-handling algorithm were to leave them alone. The *response algorithm* corrects that behavior of the objects that would penetrate. Both parts of a collision-handling algorithm pose interesting problems, but we focus on detection algorithms. For a discussion of response algorithms, see the work of Baraff [1].

Despite the wealth of literature on detection algorithms, we have found no algorithms that adequately address the needs of interactive applications such as virtual reality. To work in an interactive application, a detection algorithm must run at real-time rates, even when many objects can collide, and it must tolerate objects whose motion is guided by a user, i.e., objects whose motion is not fully pre-specified.

We present a new detection algorithm that meets these criteria through approximation. In essence, our algorithm computes an approximate characterization of the presence or absence of collisions. The accuracy of the approximation is controlled by the application program that calls our algorithm. The algorithm refines the approximation until it is interrupted by the program; the accuracy of the approximation improves with the length of time before interruption. This approach gives the application program flexibility, allowing it to degrade the quality of detection whenever an increase in speed is important.

Our approach is analogous to *progressive refinement* in rendering [3]. The success of this rendering technique suggests that our technique will prove valuable.

Our algorithm is based on two forms of approximate geometry. The first is a four-dimensional (4D) structure we call a *space-time bound*. A space-time bound provides a conservative estimate of where an object may be in the future. The algorithm uses space-time bounds to focus its attention on objects that are likely to collide. We call the second form of approximate geometry a *sphere-tree*. This structure contains several sets of spheres, each set approximating the three-dimensional (3D) surface of an object at a different level of detail. Sphere-trees allow our algorithm to quickly find approximate contacts between objects.

2 Previous work

One way to evaluate previous detection algorithms is to note how they improve on a naive algorithm. Consider a simulation involving N "agents" (objects that can move) $A_1, \ldots, A_N$. Figure 1 presents a naive algorithm that attempts to detect collisions between the agents during the time period $[0, \hat{t}]$. The next three paragraphs discuss three problems with this algorithm.

The first problem concerns the outermost loop of the algorithm. This loop increments the time variable t by a fixed timestep Δt. Notice that a larger Δt

```
for t ← 0 to t̂ in steps of Δt
    for each agent A_i ∈ {A_1, ..., A_N}
        move A_i to its position at time t
        for each agent A_j ∈ {A_{i+1}, ..., A_N}
            move A_j to its position at time t
            if (surfaces of A_i, A_j penetrate)
                then a collision occurs at time t
```

Figure 1: A naive detection algorithm.

makes the algorithm faster, but a smaller Δt makes the algorithm more accurate. The only way to get speed and accuracy is to use an adaptive timestep that is inversely proportional to the likelihood of a collision. We call the naive algorithm's inability to adaptively change its timestep the *fixed-timestep weakness.*

The second problem with the naive algorithm is the presence of the two inner loops. These loops cycle through all pairs of agents, making processing time increase quadratically with the number of agents. We call the need to check every pair at every timestep the *all-pairs weakness.*

The naive algorithm's third problem involves the "if" statement within the innermost loop. Determining whether surfaces penetrate can be tricky. For example, polyhedral intersections involve many special cases that are complicated to process. We use the term *pair-processing algorithm* for the code that checks a pair of surfaces for penetration at a given time t. A detection algorithm suffers from the *pair-processing weakness* if the pair-processing algorithm it calls is not robust and efficient.

Many authors describe solutions to these weaknesses. Few authors, however, address all three. Furthermore, the algorithms that most nearly solve all three impose restrictions on the agents, making these algorithms unsuitable for interactive applications.

To eliminate the fixed-timestep weakness, many researchers use geometry with an explicit time dimension. Cameron [2] describes 4D bounds for agents modeled with Constructive Solid Geometry. Von Herzen *et al.* [16] use a Lipschitz condition to bound the space occupied by time-varying surfaces. Duff [5] couples octree-like subdivision with interval arithmetic. All of these techniques have limitations, however. Only Cameron and Duff address the all-pairs weakness. Only Von Herzen *et al.* and Duff consider motion involving acceleration. Most importantly, none of these techniques can handle motion that is not fully pre-specified. Thus, an interactive application cannot use any of these algorithms.

Some researchers address the fixed-timestep weakness without assuming pre-specified motion. Culley and Kempf [4] show how to increase Δt when collisions cannot occur. Their technique cannot efficiently handle curved paths, nor does it eliminate the all-pairs weakness, but it inspired our work. Foisy *et al.* [6] use 4D structures somewhat similar to our space-time bounds to solve the fixed-timestep weakness. Their approach will also remedy the all-pairs weakness in some—but not all—situations.

Space subdivision is a popular way to mitigate the all-pairs weakness (although an $\Omega(N^2)$ worst case still exists). The algorithm of Shaffer and Herb [13] exemplifies the use of octrees by many researchers. Turk [15] presents a simpler technique that divides space uniformly. Although these algorithms do not address the fixed-timestep or pair-processing weaknesses, they are flexible enough to handle motion that is not fully pre-specified. Turk's algorithm, in particular, is a viable approach because of its low overhead. We compare this algorithm to ours in Section 8.

For pair processing, Thibault and Naylor [14] show how binary space partitioning (BSP) trees should accelerate polyhedral-penetration testing on the average. Sclaroff and Pentland [12] present an interesting pair-processing algorithm that approximates surfaces with deformed superquadrics. The authors do not demonstrate that their approach works for a wide variety of surfaces, but to our knowledge they are the first authors to show that approximation can make collision detection more tractable.

3 Algorithm overview

As Section 1 mentions, our detection algorithm implements a form of progressive refinement. The algorithm operates as a sequence of steps, allowing itself to be interrupted after each step. The first step is the *broad phase.* The broad phase uses details of the agents' current motion to determine the earliest time at which a collision is possible. The algorithm executes no more steps until this time arrives. At this time, the algorithm can execute the remaining steps, called collectively the *narrow phase.* The narrow phase processes only the agents involved in the possible collision, determining if they do actually collide. Once the narrow phase terminates—by reporting either a collision or the lack thereof—the algorithm returns to the broad phase and the cycle repeats. Section 6 states this algorithm more precisely.

The broad phase accurately detects collisions between the agents' bounding boxes. Each step of the narrow phase approximates the geometry of the possibly-colliding agents with twice the accuracy of the preceding step, providing twice the detection accuracy should the application program choose not to interrupt. Each geometric approximation is conservative, i.e., any collision involving true geometry corresponds to a collision involving approximate geometry.

To avoid the fixed-timestep and all-pairs weaknesses, the broad phase uses space-time bounds, the 4D structures we describe in the next three sections. Sphere-trees, which we describe in Section 7, allow the narrow phase to avoid the pair-processing weakness.

4 Space-time bounds

This section defines *space-time bounds*, the 4D structures used by the broad phase of our algorithm. For pedagogical purposes, we begin by assuming that an agent A is a point; we will soon relax this assumption. Let the position of A in $\mathbb{R}^3$ at time t be $\mathbf{x}(t)$; let $\dot{\mathbf{x}}(t)$ denote the velocity and $\ddot{\mathbf{x}}(t)$ denote the acceleration of A at t. Say we know $\mathbf{x}(0)$, $\dot{\mathbf{x}}(0)$, and a scalar M such that

$$|\ddot{\mathbf{x}}(t)| \leq M, \quad 0 \leq t \leq \hat{t}.$$

From this upper bound we can conclude that

$$|\mathbf{x}(t) - [\mathbf{x}(0) + \dot{\mathbf{x}}(0)t]| \leq \frac{M}{2}t^2, \quad 0 \leq t \leq \hat{t}. \qquad (1)$$

This assertion is related to Taylor's theorem; a proof is not difficult and we present it elsewhere [8].

The importance of Inequality 1 comes from its geometric interpretation. The inequality states that the position of A at time t will be within a distance $(M/2)t^2$ from the point $\mathbf{x}(0) + \dot{\mathbf{x}}(0)t$. Thus, a 3D bound on the position of A at t is a sphere of radius $(M/2)t^2$ centered at $\mathbf{x}(0) + \dot{\mathbf{x}}(0)t$. A 4D bound on the position of A over all times $t \in [0, \hat{t}]$ is the structure whose cross section at any particular t is that sphere. To understand this 4D structure, consider the analogous 3D structure for A moving in $\mathbb{R}^2$. The cross sections of this 3D bounding structure will be circles of radius $(M/2)t^2$ instead of spheres. Figure 2 gives an example of this 3D structure. Due to the factor of t^2 present in the structure's definition, we call the bounding structure a *parabolic horn*.

We can extend the idea of a parabolic horn to cover an agent A that is not merely a point. As it moves, this agent will rotate around a point $\mathbf{y}(t)$. The space possibly swept by this rotation will be bounded by a sphere of radius r, the *radius* of A. The parabolic horn for A is the parabolic horn for $\mathbf{y}(t)$ with its cross

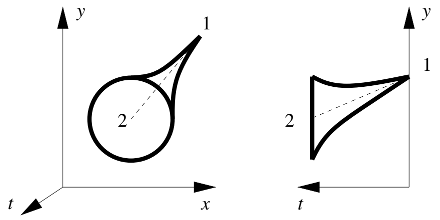

Figure 2: Two views of a parabolic horn for a point in $\mathbb{R}^2$. Point 1 is $\mathbf{x}(0)$ and point 2 is $\mathbf{x}(0) + \dot{\mathbf{x}}(0)\hat{t}$.

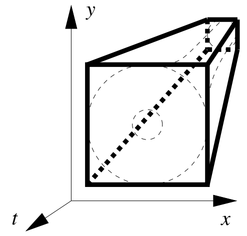

Figure 3: A hyper-trapezoid for motion in $\mathbb{R}^2$.

sections expanded by r. This simple approach can be overly conservative if A's shape is distended. In this case, we could represent A by a union of parabolic horns.

As we will see in Section 5, our detection algorithm needs to find the intersections between our 4D bounding structures. Parabolic horns contain a quadratic factor t^2 that makes them costly to intersect. Our solution is to build a simple 4D polyhedron, called a *hyper-trapezoid*, that encloses a parabolic horn. The cross sections of a hyper-trapezoid at $t = 0$ and $t = \hat{t}$ are the isothetic[1] cubes that bounds the $t = 0$ and $t = \hat{t}$ cross sections, respectively, of the parabolic horn. The cross sections of a hyper-trapezoid at $t \in (0, \hat{t})$ are defined by linear interpolation between the endpoint cubes. Note that a hyper-trapezoid has six 4D faces, one for each 3D face of its cross-sectional cubes. Each cross section of a 4D face is an isothetic 3D square. All such cross sections of this face are normal to the same axis of $\mathbb{R}^3$; we therefore say the 4D face is normal to that axis, too. This feature of 4D faces will become important in Section 5. A hyper-trapezoid is guaranteed to enclose the parabolic horn because the horn's cross sections increase in size monotonically with t. Figure 3 shows a hyper-trapezoid.

So far, we have used no information about the direction of acceleration. If we do know some limits on the direction, we can remove part of a hyper-trapezoid.

[1] A rectangle in $\mathbb{R}^n$ is *isothetic* if each of its edges is parallel to one of the coordinate axes of $\mathbb{R}^n$. A cube is isothetic if each of its faces is an isothetic rectangle.

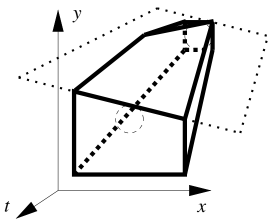

Figure 4: A space-time bound for motion in $\mathbb{R}^2$.

Say we know a vector $\mathbf{d} \in \mathbb{R}^3$ such that

$$\ddot{\mathbf{x}}(t) \cdot \mathbf{d} \ \leq \ 0, \quad 0 \leq t \leq \hat{t}.$$

This sort of limit arises when A moves away from a particular direction, e.g., if A is avoiding an obstructing wall or bouncing off another agent. Given this limit, we can conclude that

$$(\mathbf{x}(t) - [\mathbf{x}(0) + \dot{\mathbf{x}}(0)t]) \cdot \mathbf{d} \ \leq \ 0, \quad 0 \leq t \leq \hat{t}. \qquad (2)$$

The proof of this assertion is almost identical to the proof of Inequality 1. To appreciate the significance of this inequality, think of $\mathbf{d}$ as the normal vector for a 3D plane that passes through the point $\mathbf{q}(t) = \mathbf{x}(0) + \dot{\mathbf{x}}(0)t$. Inequality 2 states that at time t, the vector from A's position to $\mathbf{q}(t)$ will always be on the back side of that plane. The inequality still holds if A is not a point and we displace $\mathbf{q}(t)$ by A's radius, r. Thus A can never be in front of the plane, and A's hyper-trapezoid can be clipped at this plane; Section 5 describes how we implement this clipping. The set of 3D planes defined by $\mathbf{q}(t)$ and $\mathbf{d}$ over the time interval $[0, \hat{t}]$ are cross sections of a 4D plane. We call this 4D plane a *cutting plane*. A hyper-trapezoid teamed with a cutting plane is a complete space-time bound. Figure 4 shows an example.

The algorithms that use space-time bounds assume that M and $\mathbf{d}$ will be provided by the application program. For example, consider a driving or flight simulator. When the user starts turning her vehicle, the simulator knows that the centripetal acceleration will point to the center of the turn; thus it knows a $\mathbf{d}$ that holds until the vehicle has turned halfway around. The simulator can estimate M based on the sharpness of the turn.

5 Space-time bound intersections

If two agents are to collide at time $t \in [0, \hat{t}]$, their space-time bounds must intersect at some time $t_i \leq t$ because space-time bounds are conservative over-estimates. A detection algorithm can thus compute $\check{t}_i$, the earliest t_i from all the agents, and do no further work from time 0 to $\check{t}_i$. It should be intuitively clear that this approach solves the fixed-timestep weakness. Section 6 presents the precise statement of this approach. The current section sketches how to compute $\check{t}_i$ while avoiding the all-pairs weakness; the details of this computation can be found elsewhere [8].

The main source of complexity in the intersection algorithm is the presence of cutting planes in space-time bounds. We simplify the algorithm by observing that an intersection between two hyper-trapezoid faces is a necessary condition for any sort of intersection between two space-time bounds. A proof relies on the fact that cutting planes remove volume from hyper-trapezoids (and never add volume).

It is convenient, therefore, to structure the search for intersections between space-time bounds around the search for face intersections. Recall that each 3D cross section of a 4D hyper-trapezoid face is normal to one standard axis of $\mathbb{R}^3$, i.e., the face is normal to that axis. We can therefore partition the set of all 4D faces into three subsets:

$$F_\alpha \ = \ \{f \,|\, \text{face } f \text{ is normal to axis } \alpha\}, \ \alpha \in \{x, y, z\}.$$

These sets are important for the following reason: if two hyper-trapezoids T_j and T_k intersect for the first time at $t_i \in [0, 1]$, then there must be an intersection at t_i between a face of T_j in F_α and a face of T_k in the *same* F_α, for some $\alpha \in \{z, y, z\}$.

The contra-positive of this assertion guides our intersection algorithm. Specifically, the algorithm considers F_x, F_y and F_z in turn, testing only faces within the current F for intersection. If no F contains an intersection, then no hyper-trapezoids intersect.

To find intersections within one set, F_α, our algorithm projects each face $f \in F_\alpha$ into the α-t plane. Since f is normal to the α axis, all the points on f with the same t coordinate have the same α coordinate; the projection of f is thus a 2D line segment. If two such segments intersect at $t = t_i$, the corresponding faces *may* also intersect at $t = t_i$. The faces do actually intersect if their $t = t_i$ cross sections intersect. Since the two cross sections are isothetic squares with the same α coordinate, checking for their intersection reduces to a simple two-dimensional problem.

We use the Bentley-Ottmann algorithm [10] to find intersections between the 2D segments corresponding to F_α. This algorithm sweeps a line across the α-t plane (from low t coordinates to high t coordinates), reporting every intersection it finds.

The Bentley-Ottmann algorithm runs in $O([m + k] \log m)$ time for m segments that intersect k times. For N agents, the worst-case value of k is $O(N^2)$. We

```
t_build ← t_end /* t_end = 0 before first call */
while (t ≥ t_end)
    rebuild space-time bounds as of t_build
    ť_i ← earliest inter. between bounds, B_1, B_2
    t_end ← ť_i
    if (t_end − t_build < Δt)
      could_overlap ← TRUE
      if (B_1 and B_2 really inter. before expiring)
        if (pair-processing algorithm finds that B_1's
                and B_2's agents penetrate at t_build)
          return collision at t_build
      t_end ← t_build + Δt
    else
      could_overlap ← FALSE
    t_build ← t_end
return no collision as of t
```

Figure 5: The detection algorithm (called at time t).

have not encountered this worst case in practice, however. In fact, for the hundreds of test runs described in Section 8, the average number of segment intersections processed before a real face intersection was found was less than 0.07 percent of the worst-case value. This empirical evidence suggests that we have effectively eliminated the all-pairs weakness.

The remainder of our intersection algorithm deals with cutting planes. Whenever the algorithm finds two faces that intersect, it checks for intersections involving those faces and the cutting planes from their space-time bounds. The computation involves intersecting two 4D planes and determining when this intersection is inside the two hyper-trapezoids. The latter subproblem is a two-variable linear-programming problem [10]; since only a few constraints are involved, we solve this problem efficiently with straight-forward modifications to the Bentley-Ottmann algorithm.

6 Detecting collisions

This section describes how our detection algorithm uses the results of Section 5. We assume that the algorithm is part of a larger graphics system that calls the algorithm before rendering each frame. When called, the algorithm determines if any agents have collided since the previous frame. It returns details of any collisions it finds so the system can call a response algorithm.

Figure 5 presents the pseudo-code for our algorithm. The following paragraphs explain the details.

Let t denote the current time, the time at which the system calls our algorithm. When $t = 0$, the start of the simulation, the algorithm builds space-time bounds for all N agents. These space-time bounds will *expire* at some time in the future, the time at which we no longer know the values M and $\mathbf{d}$ from Section 4. After building, the algorithm invokes the intersection algorithm from Section 5. The intersection algorithm returns $\check{t}_i$, the time at which some pair of space-time bounds B_1 and B_2 intersect; if no space-time bounds intersect before they expire, then $\check{t}_i$ is the expiration time. For reasons that will soon become apparent, the detection algorithm saves $\check{t}_i$ in a variable, t_{end}.

If $t < t_{\text{end}}$ then there can be no collision before the current time. The detection algorithm is thus free to return immediately. When the system calls the algorithm again with the next value of t, the algorithm can again return immediately as long as $t < t_{\text{end}}$ (t_{end} persists between calls). The algorithm thus does no more work until $t \geq t_{\text{end}}$.

When $t \geq t_{\text{end}}$, our detection algorithm sets a local variable, t_{build}, to t_{end} and rebuilds the space-time bounds. Note that the rebuilt space-time bounds incorporate M and $\mathbf{d}$ as of time t_{build}. Our detection algorithm then calls the intersection algorithm to obtain a new t_{end}. Hopefully, the new t_{end} is substantially larger than t_{build} so our algorithm can continue to avoid work. It is possible, however, that B_1's and B_2's agents—call them A_1 and A_2—are close enough at t_{build} that the updated B_1 and B_2 intersect again almost immediately. We define "almost immediately" as $t_{\text{end}} - t_{\text{build}} < \Delta t$, for some small value Δt; the next paragraph discusses the choice of Δt. If $t_{\text{end}} - t_{\text{build}} < \Delta t$, we have reached the limits of the accuracy space-time bounds provide. For more accuracy, we must call a pair-processing algorithm on A_1 and A_2. We will describe our pair-processing algorithm in Section 7. Calling this pair-processing algorithm corresponds to entering the interruptible narrow phase, as described in Section 3.

Say the pair-processing algorithm finds that A_1 and A_2 do not inter-penetrate In this case, the detection algorithm must advance to a time beyond t_{build} to rebuild and retest the space-time bounds. The time it advances to is $t_{\text{build}} + \Delta t$. The value of Δt is thus the *minimum temporal resolution* of our algorithm. We assume Δt will be chosen by the application using our algorithm. The application should choose Δt small enough so that a user will not care if collisions of duration less than Δt are missed. Note the similarity between Δt here and in the naive algorithm of Figure 1. Both algorithms have the same minimum temporal resolution; our algorithm has the advantage that its timestep is not fixed at Δt.

If our detection algorithm does advance to $t_{\text{build}} + \Delta t$, agents A_1 and A_2 might become close enough that their space-time bounds penetrate immediately. The algorithm from Section 5, however, assumes that the space-time bounds are initially disjoint (with only grazing contact). To resolve this problem, our detection algorithm sets the flag *could_overlap* and performs a pre-processing test. This test iterates over the faces in F_x to find any penetration at the current time. This iteration is quite efficient, and is performed relatively rarely. The advance to $t_{\text{build}} + \Delta t$ may have allowed a pair of agents other than A_1 and A_2 to start penetrating; the pre-processing test detects these additional pairs, too. We discuss this subtle detail elsewhere [8].

7 Sphere-trees

Spheres are among the simplest geometric structures to check for penetration. If all agents were spheres, a pair-processing algorithm would be trivial. Few agents are closely approximated by a single sphere, but a set of partially-overlapping spheres can provide a more reasonable approximation. A pair-processing algorithm for agents represented as sets of spheres is also trivial.

This algorithm will be faster for small sphere sets, but small sets of spheres do not approximate agents as accurately. A solution to this dilemma is to use a hierarchy of sets for each agent. We call such a hierarchy a *sphere-tree*. Deeper levels in the hierarchy use more spheres to approximate the agent more exactly. The children of a sphere at level i are all the spheres at level $i + 1$ that it bounds. Note that when the sphere-trees for two agents penetrate, no spheres can penetrate each other unless their parents also penetrate. A recursive algorithm that descends the hierarchy can thus find penetration at any level without wasting time on spheres that cannot possibly penetrate.

Hierarchical space-subdivision of this general form is common in computer graphics. A popular technique is the octree. Sphere-trees have advantages over octrees for collision detection, however. Detection algorithms that use octrees [13] must incur the cost of rebuilding their octrees or of moving agents between octants every time they call the pair-processing algorithm. A detection algorithm that uses sphere-trees, on the other hand, builds each agent's sphere-tree only once. To update the sphere-trees, it applies to them the same rigid-body transformations that the application program applies to the agents.

Our detection algorithm uses sphere-trees for pair processing. The hierarchical algorithm for finding sphere-tree penetrations supports the goal of progressive refinement (see Section 3). Descending one level in the hierarchy corresponds to one step of the narrow phase; the application program can stop the descent at any level, and quality improves with each level. Each lowest-level sphere also stores the part of the agent's surface it intersects, allowing the last step of the narrow phase (if reached) to detect penetration exactly.

The problem remains of building the sphere-tree for each agent. There is little relevant work in the literature. O'Rourke and Badler [9] cover a polyhedron by attaching spheres to its vertices. Their approach gives little control over the number of spheres in a set and produces redundant spheres. Goldsmith and Salmon [7] present an algorithm for improving bounding hierarchies for ray tracing. To build a sphere-tree, however, this algorithm must be given the lowest-level sphere set, so it is not a full solution. To remedy these problems, we developed our own approach. We build an octree for the agent (in some default orientation), and use the circumscribing spheres of the occupied octants at level j as the sphere set at level j. Notice that the resolution of the approximation doubles with each level.

The sphere sets generated by an octree are not guaranteed to be optimal in any sense. Thus, we are exploring ways to tweak them with a simulated annealing algorithm [11]. The objective function measures the maximum distance from the agent's surface to the sphere set being tweaked. Note that this objective function also quantifies the accuracy provided by the detection algorithm.

Even without tweaking, the sphere-trees we generate seem satisfactory in practice. We have tried our approach only on polyhedral agents, however; applying the approach to other representations (e.g., spline surfaces) may prove challenging. In general, building good sphere-trees remains an interesting research problem.

8 Performance

We have evaluated the performance of our detection algorithm in several ways. First, we compared the broad phase to the broad phase of Turk's algorithm [15]. (Turk's algorithm needs no narrow phase since his application involves simple geometry.) The test program generates random configurations of isothetic cubes and applies forces to make them move. Each cube's forces are grouped into sets, each set associated with an acceleration bound M and $\mathbf{d}$; the

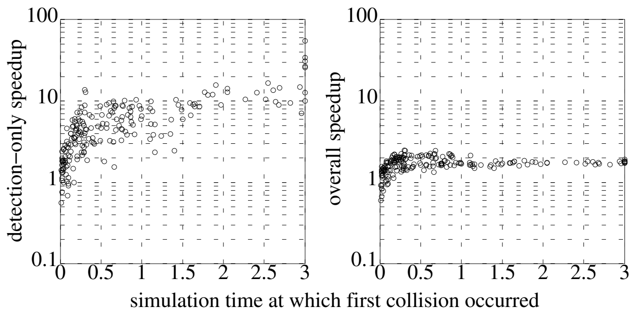

Figure 6: The speedup of our broad phase over Turk's algorithm.

number of sets and the variations within each set are random. To simulate interactive conditions, the program gives neither algorithm any "future" information except for M, $\mathbf{d}$ and their expiration time. The Unix `clock` routine measures the time each algorithm needed to find the first collision.

We ran these tests on a DECstation 5000/200 PGX Turbo with 40Mb RAM. Although we tested various numbers of agents [8], we present here only the results for 200 individual tests with 100 agents per test. Figure 6 graphs the *speedup* of our algorithm: the time spent by Turk's algorithm divided by the time spent by our algorithm. Any speedup greater than 1 indicates an advantage for our algorithm, and the larger the speedup the better. These graphs use a logarithmic scale to avoid hiding "slowdowns" (i.e., speedups less than 1). The "detection-only" speedup is a comparison of only the time spent by the detection algorithms, ignoring the time spent by the ordinary differential equation (ODE) solver; the "overall" speedup includes the ODE solver's runtime.

The graphs show that our algorithm was almost always faster than Turk's algorithm. The "overall" graph suggests that the ODE solver we used (a fourth-order Runge-Kutta method [11]) is a bottleneck; we anticipate that the "overall" speedups will be closer to the "detection-only" speedups if we use a faster ODE solver (e.g., a second-order Runge-Kutta method).

Interactive applications care about not only the performance of the system summed over all frames but also about the performance at each individual frame. To study our per-frame performance, we repeated the tests and measured the time spent by each algorithm at each frame. In many of the individual tests, our slowest frame was slower than Turk's slowest. It seems, however, that frames nearly as slow as the slowest frame are less common with our algorithm. Evidence for this conclusion comes from a histogram of the "detection-only" frame times for each algorithm

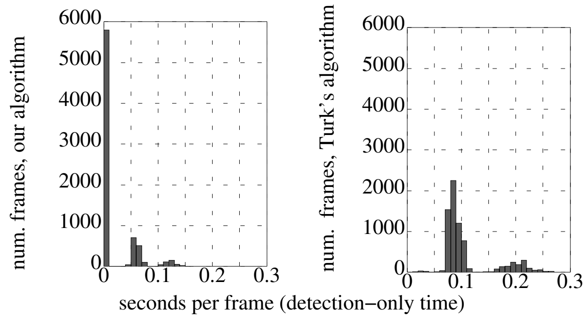

Figure 7: Frame timings for our broad phase and Turk's algorithm.

	average speedup for	
depth	*all 11,831 cases*	*132 positive cases*
2	8.905	19.532
3	8.631	14.141
4	8.310	8.419

Figure 8: The speedup of our narrow phase over a BSP algorithm.

over all tests; see Figure 7. These results give us hope that our algorithm will prove useful for interactive applications.

To evaluate our narrow phase and our full detection algorithm, we implemented a sketch of an application, a spaceship simulator. In this simulator, a user controls one ship by applying forward thrust and rotational velocity. The user's ship shares space with drone ships that periodically pick control values at random. The simulator calls our algorithm to find all the collisions between the ships. We run this simulator on a Sun SPARCstation 10/30 GT with 32Mb RAM.

The test of our narrow phase went as follows. We ran the simulator with our detection algorithm calling two pair-processing algorithms: our sphere-tree algorithm and an algorithm based on BSP trees [14]. The BSP algorithm gives exact rather than approximate results, so this test indicates what speed can be gained in return for degraded quality. Figure 8 gives the average speedup of the sphere-tree algorithm for one run of the simulator (which made 11,831 calls to each pair-processing algorithm). The "depth" refers to the maximum depth that our algorithm descended into the sphere-trees. Notice that interrupting the algorithm at a shallower depth gains the most speed on "positive" cases, i.e., cases in which the BSP algorithm detected actual penetration.

The simulator also indicates how our complete detection algorithm (broad and narrow phases) compares to existing algorithms. When the simulator calls our algorithm it executes a set number of simulation time units 5 to 7 times more quickly than when it calls Turk's algorithm coupled with the BSP algorithm.

9 Conclusions

In this paper, we have presented a new approach to collision detection. Our algorithm addresses three weaknesses common in detection algorithms, and allows interactive applications to trade quality for speed as needed. Our implementation significantly outperforms existing algorithms in empirical tests.

We believe our work raises some interesting questions. As we mention in Section 3, our broad phase is conservative, i.e., it never ignores bounding-box collisions. The algorithm could be faster if the broad phase selectively ignored agents that the application temporarily designated "unimportant." Would speed gained in this manner be useful? In general, what tradeoff between speed and accuracy do users find most appropriate, and for what tasks? These questions make topics for interesting future work.

10 Acknowledgements

This paper would not have been possible without the cheerful guidance of John "Spike" Hughes. Bob Zeleznik and Andy van Dam also played important roles in this research. This work was supported in part by the NSF/DARPA Science and Technology Center for Computer Graphics and Scientific Visualization, IBM, NCR, Sun Microsystems, DEC, HP, and ONR grant N00014-91-J-4052 ARPA order 8225.

References

[1] Baraff, D. and A. Witkin, "Dynamic Simulation of Non-Penetrating Flexible Bodies," SIGGRAPH '92 Proceedings, *Computer Graphics*, Vol. 26, No. 2, July 1992, pp. 303–308.

[2] Cameron, S., "Collision Detection by Four- Dimensional Intersection Testing," *IEEE Transactions on Robotics and Automation*, Vol. 6, No. 3, June 1990, pp. 291–302.

[3] Cohen, M. F., S. E. Chen, J. R. Wallace and D. P. Greenberg, "A Progressive Refinement Approach to Fast Radiosity Image Generation," SIGGRAPH '88 Proceedings, *Computer Graphics*, Vol. 22, No. 4, August 1988, pp. 75–84.

[4] Culley, R. K., and K. G. Kempf, "A Collision Detection Algorithm Based on Velocity and Distance Bounds," *Proceedings 1986 IEEE International Conference on Robotics and Automation*, Vol. 2, 1986., pp. 1064–1069.

[5] Duff, T., "Interval Arithmetic and Recursive Subdivision for Implicit Functions and Constructive Solid Geometry," SIGGRAPH '92 Proceedings, *Computer Graphics*, Vol. 26, No. 2, July 1992, pp. 131–138.

[6] Foisy, A., V. Hayward and S. Aubry, "The Use of Awareness in Collision Prediction," *Proceedings 1990 IEEE International Conference on Robotics and Automation*, Vol. 1, pp. 338–343.

[7] Goldsmith, J. and J. Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," *IEEE Computer Graphics and Applications*, Vol. 7, No. 5, May 1987, pp. 14–19.

[8] Hubbard, P. M., "Space-Time Bounds for Collision Detection," Tech. Report CS-93-04, Department of Computer Science, Brown University, February 1993.

[9] O'Rourke, J. and N. Badler, "Decomposition of Three-Dimensional Objects into Spheres," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 1, No. 3, July 1979, pp. 295–305.

[10] Preparata, F. P. and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag (New York), 1985.

[11] Press, W. H., B. P. Flannery, S. A. Teukolsky and W. T. Vetterling, *Numerical Recipes in C*, Cambridge University Press (Cambridge), 1988.

[12] Sclaroff, S. and A. Pentland, "Generalized Implicit Functions for Computer Graphics," SIGGRAPH '91 Proceedings, *Computer Graphics*, Vol. 25, No. 4, July 1991, pp. 247–250.

[13] Shaffer, C. A. and G. M. Herb, "A Real-Time Robot Arm Collision Avoidance System," *IEEE Transactions on Robotics and Automation*, Vol. 8, No. 2, April 1992, pp. 149–160.

[14] Thibault, W. C. and B. F. Naylor, "Set Operations on Polyhedra Using Binary Space Partitioning Trees," SIGGRAPH '87 Proceedings, *Computer Graphics*, Vol. 21, No. 4, July 1987, pp. 153–162.

[15] Turk, G., "Interactive Collision Detection for Molecular Graphics," Tech. Report TR90-014, Computer Science Department, The University of North Carolina at Chapel Hill, March 1990.

[16] Von Herzen, B., A. H. Barr and H. R. Zatz, "Geometric Collisions for Time-Dependent Parametric Surfaces," SIGGRAPH '90 Proceedings, *Computer Graphics*, Vol. 24, No. 4, August 1990, pp. 39–48.