Space-Time Bounds for Collision Detection

Philip M. Hubbard

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-04
Febuary 1993

Space-Time Bounds for Collision Detection

Philip M. Hubbard
Department of Computer Science
Box 1910, Brown University
Providence, RI 02912
pmh@cs.brown.edu

January 27, 1993

Abstract

Collision detection and response is an important but costly component of computer animation. We identify three basic reasons why collision-detection algorithms can be slow. We present a new collision-detection algorithm that directly addresses two of these problems and that prepares us for future work on the third. Our algorithm is based on a four-dimensional structure that puts a conservative bound on motion through time, even if that motion is specified interactively by a user. The empirical results we present suggest that our algorithm performs significantly better than previous 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 break into pieces). A computer simulation of the physical world needs to enforce this property of solid objects if it is to appear realistic. We use the term *collision-handling algorithm* for the part of the simulation system that enforces the solidness of objects.

Researchers in computer graphics and robotics have developed a variety of collision-handling algorithms. Typically, a collision-handling algorithm consists of two parts. The *collision-detection algorithm* determines whether objects in the simulation would penetrate one another if the collision-handling algorithm were to leave them alone. The *collision-response algorithm* processes the objects that would penetrate, correcting their behavior to prevent them from penetrating. For simplicity, we call these algorithms the *detection algorithm* and the *response algorithm*, respectively. Both parts of a collision-handling algorithm pose interesting problems, but we focus on detection algorithms in this paper. For a discussion of response algorithms, see the work of Baraff [Bara89, Bara90, Bara91, Bara92a, Bara92b]

The disadvantage associated with the detection algorithms in the literature is their slowness. Slowness is undesirable in any application, but in applications that require interaction with users at real-time rates (e.g. virtual reality) slowness can be unacceptable. We have developed a new technique to alleviate this slowness. The foundation of our work is a four-dimensional geometric structure, the fourth dimension representing time. This *space-time bound* provides a conservative estimate of where a moving object will be in the future. Our detection algorithm uses this estimate to avoid processing objects that cannot possibly collide. To build a space-time bound for an object, we require upper bounds on the object's future acceleration but not a complete specification of the object's future motion. Space-time bounds are thus compatible with the user-guided objects from interactive applications. Experiments indicate that space-time bounds make our detection algorithm more efficient than previous algorithms.

The remainder of this paper is structured as follows. Section 2 discusses detection algorithms that appear in the literature. Section 3 defines space-time bounds. Sections 4 and 5 detail the role of space-time bounds in our detection algorithm. Section 6 describes the performance of our algorithm. Finally, Section 7 outlines some extensions of our algorithm.

```
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 (the geometric representations of A_i and A_j penetrate)
              then a collision occurs at time t
```

Figure 2.1: A naive collision-detection algorithm.

2 Previous Work

2.1 A Naive Algorithm

One way to evaluate previous work on detection algorithms is to note how the published algorithms improve on a naive algorithm. Consider a simulation involving N "agents" (objects that can move) $A_1, \ldots, A_N$. In Figure 2.1, we present a naive algorithm that attempts to detect collisions between the agents in the period $[0, \hat{t}]$. In the next three paragraphs, we discuss three problematic aspects of this algorithm.

The first problem concerns the outermost loop of the algorithm. This loop increments the time variable t by a fixed timestep Δt. To use this algorithm, one must pick an appropriate value for Δt. A large value of Δt makes the algorithm progress quickly, but decreases its accuracy. The algorithm can be up to Δt time units late in its detection of collisions, or it can miss entirely any collision which lasts less than Δt time units. To increase the accuracy, one can choose a smaller value of Δt, but then the algorithm will be slower. A good way to escape this dilemma is to use an *adaptive timestep* that changes according to the situation: the size of the timestep should (somehow) be inversely proportional to the likelihood that agents will collide. 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 the time spent by the algorithm increase quadratically with N, the number of agents. As users demand more detailed simulations, the typical values of N will increase and the performance of the naive algorithm will worsen. We would prefer an algorithm that does not inherently need to check all pairs of agents to detect collisions, and we call the need to check these pairs the *all-pairs weakness*.

The naive algorithm's third problem involves the "if" statement within the innermost loop. It can be difficult to efficiently and accurately determine whether the geometric representations of two agents penetrate at a particular time t. For example, consider the case in which the geometric representations are polyhedra. One must consider all the different ways that faces, edges and vertices can contact one another; it is difficult enough to enumerate all the special cases, let alone write and debug code to process them efficiently. We use the term *pair-processing algorithm* for the code that checks a pair of geometric representations for penetration. We say a detection algorithm suffers from the *pair-processing weakness* if the pair-processing algorithm it calls does not work robustly and efficiently.

The work we discuss starting in Section 3 focuses on the fixed-timestep and all-pairs weaknesses. In Section 7 we sketch an extension to our work that addresses the pair-processing weakness.

2.2 Improvements on the Naive Algorithm

Many authors have described solutions to the naive algorithm's problems. Few authors, however, have addressed all three weaknesses. Furthermore, the algorithms that most nearly solve all three weaknesses impose

restrictions on the agents that make these algorithms less useful, particularly for interactive applications.

The literature addressing the fixed-timestep weakness includes several attempts to dispense with the timestep Δt altogether. Lubachevsky [Luba91] notes that when agents move in straight lines in a plane it is simple to directly compute the time at which their paths cross. This approach does not generalize to more complicated situations, however. Boyse [Boys79] derives simple equations for the motion of polyhedral edges and faces over time; a root of any such equation corresponds to the time and location of a collision. He places fewer restrictions on motion than Lubachevsky, but he does assume one stationary and one moving polyhedron, with the moving polyhedron being limited to periods of only linear translation and periods of only rotation around a fixed axis. Canny [Cann86] uses more elaborate analysis to process more general motion. He obtains quintic polynomials whose roots, found with iterative numerical techniques, describe the collisions between convex polyhedra. The motion he can handle is still not completely general, however: agents must have constant linear velocity and "nearly-constant" angular velocity. Canny's algorithm also requires the motion of all agents to be completely specified in advance, making his algorithm unusable for applications in which a user interactively specifies the motion.

Samet and Tamminen [Same85] address the fixed-timestep weakness by working in a four-dimensional space that includes a temporal dimension. Their algorithm recursively subdivides the space like an octree, making more subdivisions in regions that could contain a pair of agents. The net effect is that the algorithm focuses on times when collisions are likely. The algorithm works only for agents represented by the Constructive Solid Geometry (CSG) modeling paradigm, though, and the authors are vague about how to handle agents whose motion is not linear. Duff [Duff92] uses the formalism of interval arithmetic as the basis of an algorithm similar to Samet and Tamminen's algorithm. Interval arithmetic allows Duff's algorithm to handle nonlinear motion, but it cannot handle motion defined by ordinary differential equations (e.g., physically-based motion). Snyder [Snyd92] presents another application of interval arithmetic to solving the fixed-timestep weakness. His approach is based on the approach of Von Herzen [VonH89, VonH90], which uses the Lipschitz condition. The Lipschitz condition gives bounds on the space a surface can occupy over a particular interval of time; Von Herzen uses these bounds in a recursive-subdivision algorithm analogous to Samet and Tamminen's algorithm. Like Canny's algorithm, the algorithms of Von Herzen, Snyder and Duff all require that motion be prespecified.

Culley and Kempf [Cull86] attack the fixed-timestep weakness by making the following observation: an upper bound on the relative velocity of two agents and a lower bound on their separation distance define a lower bound on the time before they can collide. Culley and Kempf describe an algorithm that uses this lower bound on collision time to adaptively vary its timestep. Cameron [Came85] sketches a similar idea in less detail. None of these authors present any efficient ways to make their algorithms handle non-linear motion or more than two agents. The spirit of this approach seems valid, however, and it served as an inspiration for our work on space-time bounds.

The all-pairs weakness has received less attention in the literature than the fixed-timestep weakness. Herman [Herm86] uses an octree to group agents according to their location in space, thus avoiding the processing of pairs of agents that are separated by large distances. The algorithm suffers from the same limitations as Boyse's algorithm, though, in that only one agent can be moving and its motion must consist of independent translations and rotations. Duff's subdivision algorithm has the properties of an octree algorithm, but also puts restrictions on the allowable motion as we mentioned earlier. Moore and Wilhelms [Moor88] describe another algorithm based on octrees. The only restrictions they place on an agent's motion is that it can be considered linear between timesteps, but they explicitly ignore a few special cases of motion that could produce collisions. They also present no empirical results to verify that the time saved by using an octree outweighs the time spent building the octree at each timestep; knowing the results of this tradeoff is particularly significant because Moore and Wilhelms do not address the fixed-timestep weakness, so the number of timesteps their algorithm uses could be large.

Lubachevsky's approach to the all-pairs weakness is to subdivide space into equal-sized squares he calls "sectors." Turk [Turk90] uses a similar approach in his algorithm to detect collisions between molecules, with his sectors being cubes. The algorithm compares only pairs of agents that occupy the same sector; the goal is to obtain the advantages of an octree without the overhead of building it. Turk's empirical data

suggests that his approach works well, at least for the application of molecular modeling, even though he does not address the fixed-timestep weakness. We will discuss Turk's algorithm in more detail in Section 6.

The pair-processing weakness is a popular topic in the literature. Several of the algorithms that address the fixed-timestep or all-pairs weaknesses also provide some sort of solution to the pair-processing weakness. For pairs of polyhedral models, the goal is to avoid the brute-force technique of comparing every polygon from one model with every polygon from the other model. The subdivision techniques of Samet and Tamminen, Duff, Herman, and Moore and Wilhelms do avoid this sort of comparison. No one has presented empirical data comparing these algorithms against less sophisticated solutions to the pair-processing weakness, but theoretical arguments suggest that the expected performance of the subdivision techniques is better. Snyder and Von Herzen apply subdivision techniques to models made of parametric patches instead of polygons. Their algorithms seem to be an efficient way to find contacts between a single pair of patches, but they do not address models made from more than one patch.

Hahn [Hahn88] suggests that he uses an octree in a manner similar to Moore and Wilhelms, but he provides no details of his algorithm. Cameron [Came89] presents an algorithm for a pair of CSG models. Viewing a CSG model as a tree of operations, he shows how bounds on the partial results at internal nodes allow collisions between trees to be found quickly. He reports speedups of "a factor of about 100" over brute-force techniques. In a later paper [Came90], Cameron extends his bounds to work in four dimensions. The result is a solution to the fixed-timestep weakness. He assumes piecewise-linear motion, however, and this motion must be fully specified in advance. In this paper, Cameron also addresses the case of more than two CSG models, but he does not fully eliminate the all-pairs weakness. Dobkin and Kirkpatrick [Dobk83] describe another approach for a pair of convex polyhedra. By slicing the polyhedra into a particular form of cross-sections, the algorithm can applying two-dimensional techniques to find collisions. The authors prove that the algorithm has good asymptotic performance, but they admit that it has never been implemented and evaluated empiricly. Their algorithms addresses neither the fixed-timestep nor the all-pairs weaknesses.

Sclaroff and Pentland [Scla91, Pent91] describe a novel solution to the pair-processing weakness. They suggest that polyhedral models be approximated by deformed superquadric ellipsoids whose surfaces have been modulated by a "displacement map"; these approximate models have the advantage that contacts between them are simpler to detect. The performance figures quoted by the authors suggest that their algorithm is fast for the class of models that can be represented by their superquadrics. They do not characterize the members of this class, though, leaving us to wonder whether it includes many useful models. The idea of approximation is a promising one, however, and we will explore it further in Section 7.

After we had developed our space-time bounds, we discovered a paper by Foisy *et al.* [Fois90] that contains some similar ideas. An important element of their work is a queueing scheme that allows the algorithm to compare only $O(\log N)$ pairs of agents at each timestep. When the frequency and number of possible collisions meet certain conditions, this scheme solves the all-pairs weakness. If these conditions are not met, however, the algorithm can fail to detect some collisions; to guarantee correctness in this case, the algorithm must call some other detection algorithm.

Although there are many interesting ideas in the papers we have just summarized, none of the proposed algorithms solve all three weaknesses with the naive algorithm. The absence of a fully-satisfying detection algorithm motivated us to develop our own approach.

3 Space-Time Bounds

Our detection algorithm is based on four-dimensional (4D) geometry, the fourth dimension representing time. The use of an explicit temporal dimension is not new; Canny [Cann86], Samet and Tamminen [Same85], Cameron [Came90] and Duff [Duff92] exploit this extra dimension, as we discuss in Section 2. Unlike these other researchers, we do not use 4D geometry to exactly represent each agent moving through time. Instead, our 4D structures are bounds that give conservative over-estimates on the space an agent could occupy. Our detection algorithm uses these *space-time bounds* to prune the set of agents; agents that cannot collide are eliminated quickly, making it cheaper to process those agents that can collide. The approximate nature of

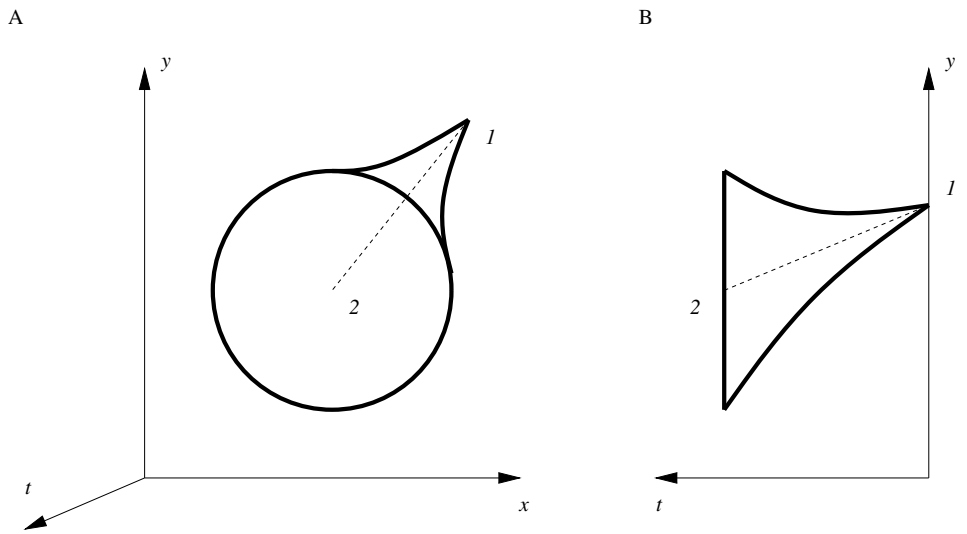

Figure 3.1: Two views of a parabolic horn. This parabolic horn bounds the position through time of a point agent moving in $\mathbb{R}^2$. The point labeled 1 is $\mathbf{x}(0)$ and the point labeled 2 is $\mathbf{x}(0) + \dot{\mathbf{x}}(0)\hat{t}$.

space-time bounds allows us to build them efficiently without placing strong restrictions on the motion an agent may exhibit; we can even handle interactive motion that is not completely specified in advance.

In this section, we start with a simple idea and then gradually augment it to arrive at the general definition of a space-time bound. We show how to use the space-time bounds for a set of agents in Sections 4 and 5. These sections will describe how space-time bounds address the fixed-timestep and all-pairs weaknesses.

We begin the derivation by assuming an agent A is a point. Let the position of A in $\mathbb{R}^3$ at time t be described by the function $\mathbf{x}(t)$; let $\dot{\mathbf{x}}(t)$ denote the velocity and $\ddot{\mathbf{x}}(t)$ denote the acceleration of A at t. If we know $\mathbf{x}(0)$, $\dot{\mathbf{x}}(0)$ and $\ddot{\mathbf{x}}(0)$ then Taylor's Theorem [Elli78] tells us that the position of A at time $t \geq 0$ is

$$\mathbf{x}(t) \;=\; \mathbf{x}(0) + \dot{\mathbf{x}}(0)t + \ddot{\mathbf{x}}(0)\frac{t^2}{2} + O(t^3).$$

This result is not precise enough for our current purposes, but we can use this general idea to develop a more useful result. Say we know $\mathbf{x}(0)$, $\dot{\mathbf{x}}(0)$, and a scalar M such that

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

We assume M will be supplied by the application program using our detection algorithm, as we discuss in Section 5.3. With this upper bound on the magnitude of A's acceleration, we can say the following about $\mathbf{x}(t)$:

$$|\mathbf{x}(t) - [\mathbf{x}(0) + \dot{\mathbf{x}}(0)t]| \;\leq\; \frac{M}{2}t^2, \quad 0 \leq t \leq \hat{t}. \tag{3.1}$$

A proof of this assertion can be found in Appendix A.

The value of Inequality 3.1 comes from its geometric interpretation. The inequality states that whatever the position of A at time t, it will be within a distance $(M/2)t^2$ from the point $\mathbf{x}(0) + \dot{\mathbf{x}}(0)t$. Thus, a three-dimensional (3D) bound on the position of A at time t is merely 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 get a better intuition for this 4D structure, drop this entire discussion down one dimension: A is now moving in $\mathbb{R}^2$, and a structure that bounds A's movement over all time has three dimensions. The cross-sections of the bounding structure will now be circles of radius $(M/2)t^2$ instead of spheres. Figure 3.1 gives an example of the bounding structure in this case. Due to the factor of t^2 present in the 4D structure's definition, we call the structure a *parabolic horn*.

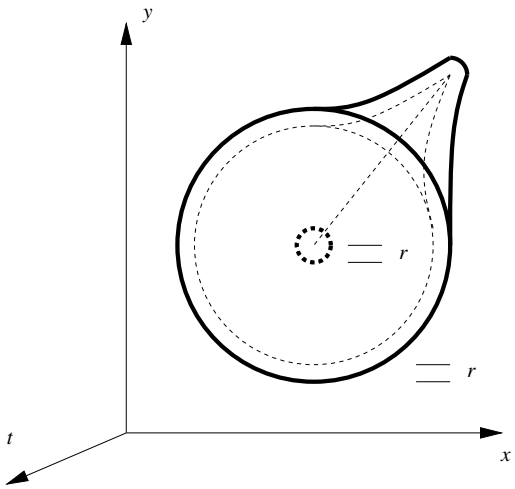

Figure 3.2: This parabolic horn bounds the position through time of a non-point agent moving in $\mathbb{R}^2$.

We can extend the idea of a parabolic horn to cover an agent A that is a rigid body (larger than a point). The motion of such an agent is complicated because it can involve angular velocities and accelerations. At any moment in time, however, we can treat this motion as instantaneous rotation about a point $\mathbf{y}$ that is subject to only non-angular velocities and accelerations; if we are using Newtonian dynamics, $\mathbf{y}$ is A's center of mass [Gold50]. If A does not change its scale, there is a 3D sphere that, when centered at $\mathbf{y}(t)$, bounds the space swept by these rotations. Let r be the radius of this sphere; we call r the *radius* of A. If we build a parabolic horn for the point $\mathbf{y}$ and expand every spherical cross-section by r, we have a parabolic horn that bounds A. Figure 3.2 illustrates this idea (dropping down one dimension as in Figure 3.1). This simple approach to handling rotation can be overly conservative if the general shape of A in no way resembles a sphere. In this case, we could represent A by several overlapping parabolic horns, each bounding only a part of A. If A is a long rectangular box, for example, we could use a set of parabolic horns whose spherical cross-sections lie along the central axis of the box.

The conceptual simplicity of parabolic horns makes them an appealing way to bound agents in motion, but they have a computational disadvantage. The problem is the quadratic term, t^2. The number of algorithms for processing non-linear geometry is small compared to the wealth of algorithms for linear geometry [Prep85]. We need an algorithm to efficiently determine which of the N bounds intersect, as we will discuss in Section 4. So that we can use a published algorithm for this intersection problem, we develop a linear approximation to parabolic horns.

Our approach is to build a simple 4D polyhedron that encloses an agent's parabolic horn. All the 3D cross-sections of our polyhedron will be isothetic[1] cubes. The $t = 0$ and $t = \hat{t}$ cross-sections are the bounding boxes of the $t = 0$ and $t = \hat{t}$ cross-sections of the parabolic horn, respectively. The cross-sections for $t \in (0, \hat{t})$ are defined by linear interpolation between the endpoint cubes. The polyhedron has six faces, one for each face of its cubical cross-sections. Note that every cross-section of a particular face is normal to the same axis of $\mathbb{R}^3$; we therefore say the face is normal to that axis, too. We will exploit this feature of faces in Section 4. We call our 4D polyhedron a *hyper-trapezoid* because each face has pairs of parallel edges and pairs of non-parallel edges. A hyper-trapezoid is guaranteed to enclose the parabolic horn because the horn's cross-sections increase in size monotonically with t. Figure 3.3 shows the hyper-trapezoid that bounds the parabolic horn from Figure 3.2.

In our development of hyper-trapezoids, we have not used any information about the direction of acceleration. If we do know some limits on the direction, we can remove part of a hyper-trapezoid, tightening

[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.

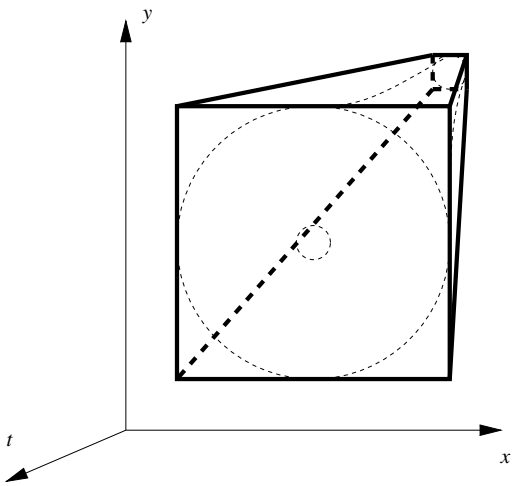

Figure 3.3: This hyper-trapezoid approximates a parabolic horn for an agent moving in $\mathbb{R}^2$.

the bound it represents. Assume, for the moment, that agent A is a point. Say we know that at every time $t \in [0, \hat{t}]$, the acceleration vector $\ddot{\mathbf{x}}(t)$ of A will lie in one half of a sphere around A. This sort of limit arises when A wants to move away from a particular direction, e.g., if A is moving away from an obstructing wall. Knowing this information is equivalent to knowing a vector $\mathbf{d} \in \mathbb{R}^3$ such that

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

As with the magnitude bound M, we assume the application program will supply $\mathbf{d}$; see Section 5.3. Given this relationship between $\ddot{\mathbf{x}}(t)$ and $\mathbf{d}$, we can conclude that

$$(\mathbf{x}(t) - [\mathbf{x}(0) + \dot{\mathbf{x}}(0)t]) \cdot \mathbf{d} \ \leq \ 0, \quad 0 \leq t \leq \hat{t}. \tag{3.2}$$

A proof can be found in Appendix B. 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 3.2 then 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. Thus, A itself will always be on the back side of the plane. Since we know that A can never be in front of the plane, any part of a bound on A's position that lies in front of the plane can be discarded; we will discuss how this discarding is actually implemented in Section 4. 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 structure. This structure is a 4D plane because the cross-sections never change orientation ($\mathbf{d}$ is constant) and their movement through time is linear ($\mathbf{q}(t)$ is linear in t). We call this 4D plane a *cutting plane*, and we consider a hyper-trapezoid teamed with a cutting plane to be a complete space-time bound.

When A is not a point, we need to displace the 4D cutting plane away from the center of A so it will never cut through A itself. Note that if we displace along $\mathbf{d}$, the interpretation of Inequality 3.2 still holds. Figure 3.4 shows an example of a displaced cutting plane and a hyper-trapezoid, together forming a space-time bound.

4 Finding Space-Time Bound Intersections

Now that we have defined a space-time bound, we show why space-time bounds are valuable in a collision-detection algorithm. If two agents are going to collide at time $t \in [0, \hat{t}]$, their space-time bounds must intersect at some time $t_i \leq t$; this conclusion follows because space-time bounds are conservative over-estimates. Say our detection algorithm has space-time bounds for all N agents, and it has found the earliest

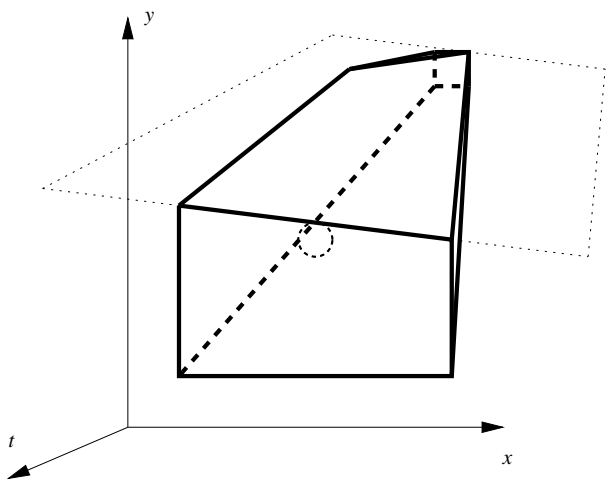

Figure 3.4: Here is a complete space-time bound consisting of a polyhedral bound and a cutting plane to tighten the bound.

time $\check{t}_i$ at which a pair of space-time bounds intersect. Then the algorithm can conclude that no collisions can possibly occur from time 0 until time $\check{t}_i$. The algorithm can thus wait until $\check{t}_i$ arrives before it has to do any more work. We save the precise statement of this approach for Section 5, but it should be clear at an intuitive level that we have a solution to the fixed-timestep weakness. We now concentrate on how to determine $\check{t}_i$ while avoiding the all-pairs weakness.

Throughout this section, we assume $\hat{t} = 1$. By making this assumption, we simplify the computations we must perform to find space-time-bound intersections. In a real implementation, it is simple to change to this normalized time scale. For completeness, we describe the normalizing process in Appendix C.

4.1 Overall Strategy

The main source of complexity in the intersection algorithm is the presence of cutting planes in space-time bounds. We can simplify the algorithm by making the following observation: the intersection between two hyper-trapezoid faces is a necessary condition for any intersection between two space-time bounds. To understand this fact, let B_1 be a space-time bound consisting of hyper-trapezoid T_1 and cutting plane P_1; define B_2, T_2 and P_2 analogously. Assume that T_1 and T_2 are disjoint at $t = 0$. Consider the different ways B_1 and B_2 can intersect:

- An intersection might involve a face f_1 of T_1 and a face f_2 of T_2. Obviously, the intersection of hyper-trapezoid faces is a necessary condition in this case.
- An intersection might involve a face f_1 of T_1 and the cutting plane P_2. We know, however, that a cutting plane can only remove volume from a hyper-trapezoid, never add it; so we care about only the part of P_2 within T_2. Thus, f_1 can intersect P_2 only if f_1 also intersects T_2, i.e., if f_1 intersects a face f_2 from T_2. The region $f_1 \cap f_2$ may well be cut off by P_2, but it still must exist. Figure 4.1A illustrates this situation. The same argument holds for an intersection between a face f_2 from T_2 and P_1.
- An intersection might involve P_1 and P_2. By the same argument we used in the last case, we care about only the parts of P_1 and P_2 within T_1 and T_2, respectively. Thus there must be an intersection between a face f_1 from T_1 and a face f_2 from T_2. See Figure 4.1B.

There are no other ways B_1 can intersect B_2. We find it convenient, therefore, to structure our search for intersections between space-time bounds around the search for intersections between hyper-trapezoid

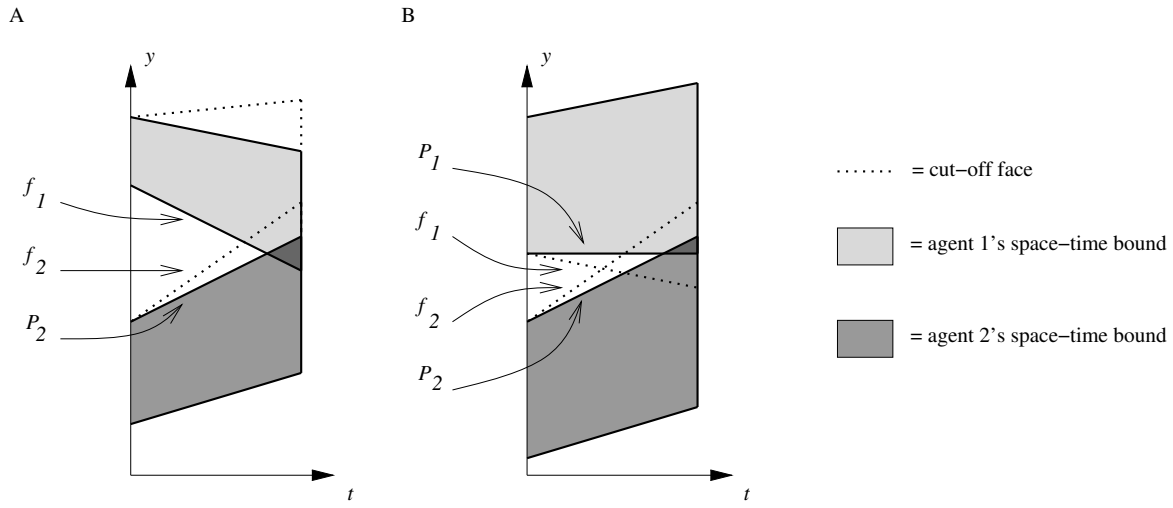

Figure 4.1: Part A shows that if face f_1 intersects cutting plane P_2, then f_1 also intersects face f_2. Part B shows that if cutting planes P_1 and P_2 intersect, then face f_1 and f_2 also intersect. In both parts, the faces are shown edge-on, as line segments.

faces. The relative simplicity and efficiency of finding intersections between faces, discussed in Section 4.2, is another reason for checking this case before checking any other cases. Once we have considered the details of all the cases, we will present the pseudo-code for our full intersection algorithm.

4.2 Face-Face Intersections

Recall that each cross-section of a hyper-trapezoid face is normal to a standard axis of $\mathbb{R}^3$, i.e., the face is normal to that axis. We can therefore partition the set of all faces into three subsets:

$$F_\alpha \;=\; \{f \,|\, f \text{ is a face from a hyper-trapezoid, and } f \text{ is normal to axis } \alpha\}, \quad \alpha \in \{x, y, z\}. \tag{4.1}$$

These sets are important for the following reason: if two hyper-trapezoids T_j and T_k are disjoint at $t = 0$ and intersect for the first time at $t = t_i \in (0, 1]$, then there must be an intersection at t_i between a face from T_j belonging to F_α and face from T_k also belonging to the *same* F_α, for some $\alpha \in \{z, y, z\}$. The proof of this assertion is messy but essentially trivial, so we omit it.

The contra-positive of this assertion guides our face-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 all three sets have been considered and no intersection has been found, then the algorithm can conclude that there is no intersection between any hyper-trapezoids.

To find intersections within one 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. Note that the segment starts at $t = 0$ and ends at $t = 1$. The algorithm builds one such segment for each face and looks for intersections between these segments. 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 the simple problem of finding the intersection between two squares in the β-γ plane (letting β and γ be the standard axes of $\mathbb{R}^3$ other than α).

To find this intersection, we need the β and γ coordinates of the squares' corners. Recall that each square is one side of a 3D isothetic cube, that cube being the $t = t_i$ cross-section of a hyper-trapezoid T. This side shares an edge with four other sides of the cube. It should be clear that given a point on each of these

other sides (four points total), we can compute the corner coordinates we need; this fact is a nice feature of *isothetic* cubes. Thus, we need a formula for some point on a side of T's cross-section at time t, i.e., on a face of T. Let T correspond to an agent A with $\mathbf{x}(0)$, $\dot{\mathbf{x}}(0)$, r and M defined as in Section 3. Let the face be normal to $\mathbf{u}_\beta$, a unit vector pointing along the $+\beta$ axis. This face lies in some 4D plane, p. A point on the face at time 0 is

$$\check{\mathbf{p}} \;=\; \mathbf{x}(0) + r\mathbf{u}_\beta, \tag{4.2}$$

and a point on the face at time 1 is

$$\hat{\mathbf{p}} \;=\; \mathbf{x}(0) + \dot{\mathbf{x}}(0) + \left(\frac{M}{2} + r\right)\mathbf{u}_\beta. \tag{4.3}$$

Thus, a point on the face at time t is

$$\mathbf{p}(t) = \check{\mathbf{p}} + (\hat{\mathbf{p}} - \check{\mathbf{p}})\,t, \quad 0 \le t \le 1. \tag{4.4}$$

Note that if the face is normal to $-\mathbf{u}_\beta$, Equation 4.4 still holds with only one modification to Equations 4.2 and 4.3: replace $\mathbf{u}_\beta$ with $-\mathbf{u}_\beta$.

The intersection between a pair of face cross-sections, if it exists, will be an isothetic rectangle. Once the algorithm finds such an intersection, it still must consider whether the rectangle has been cut off by any cutting planes. The only cutting planes that matter are those from the two space-time bounds that contain the intersecting faces. The intersection rectangle is cut off only if all four of its vertices are behind the t_i cross-section of one or other cutting plane.

We are left with the problem of efficiently finding intersections between 2D segments. One solution would be to test every pair of segments for intersections, but we want to avoid this version of the all-pairs weakness. Instead, we use the algorithm of Bentley and Ottmann [Bent79, Prep85]. In its full generality, this algorithm sweeps a line across the α-t plane (from low t coordinates to high t coordinates), updating its data structures whenever a new segment starts or ends, and reporting every intersection it finds. We were able to simplify the algorithm in our application. All our segments start at $t = 0$ and end at $t = 1$, so we do not need to worry about segments starting late or ending early. Also, we care about only the earliest (lowest t coordinate) intersection that corresponds to a real face intersection, so we can stop the algorithm as soon as the algorithm finds this intersection. The only complication we encountered concerns round-off error in some of the algorithm's numerical calculations; we discuss our solution to this problem in Appendix D.

The Bentley-Ottmann algorithm runs in $O([m+k]\log m)$ time for m segments that intersect k times. In our application, $m = 2N$ because F_α contains two faces for each of the N agents. The worst-case value of k is

$$k \;=\; \frac{2N(2[N-1])}{2} \;=\; 2N^2 - 2N,$$

since each segment from each agent can intersect all the segments from the other $N-1$ agents. One can imagine pathological cases in which our algorithm processes all $2N^2 - 2N$ of these segment intersections without finding any real face intersections. We did not encounter any of these pathological cases in the testing of our algorithm, though. In fact, for the hundreds of test runs described in Section 6, 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 in practice.

4.3 Face-Plane Intersections

As we mentioned in Section 4.1, the second category of space-time-bound intersections involves a face and a cutting plane. Say the face is f_1 and the plane is P_2. Let T_1 be the hyper-trapezoid that includes f_1, and let P_1 be the cutting plane associated with T_1; let T_2 be the hyper-trapezoid associated with P_2. We argued in Section 4.1 that f_1 and P_2 can intersect only if f_1 intersects some face f_2 from T_2. We now argue that the intersection between f_1 and P_2 matters only if the intersection between f_1 and f_2 is cut off by P_1 or P_2.

If a face is cut off at the t coordinate t_c, it stays cut off for all $t > t_c$. Thus, if the intersection between f_1 and f_2 is not cut off, it must occur at a lower t coordinate than the intersection between f_1 and P_2. Since we are trying to find the earliest intersection, we care about only the lower t coordinate.

Assume, then, that the intersection between f_1 and f_2 has been cut off. We need to characterize the intersection between f_1 and P_2. We start by considering the intersection between P_2 and p_1, where p_1 is the 4D plane that contains f_1. Let $\mathbf{n}_1$ and $\mathbf{n}_2$ be the normal vectors for p_1 and P_2, respectively. The line of intersection between p_1 and P_2 has direction

$$\mathbf{m} = \mathbf{n}_1 \times \mathbf{n}_2.$$

This direction remains constant throughout the time interval $[0, 1]$ because neither plane changes its orientation. Now we need $\mathbf{r}(t)$, a reference point on the intersection line at time t. If we know a point $\check{\mathbf{r}}$ on the line at $t = 0$ and a point $\hat{\mathbf{r}}$ on the line at $t = 1$, we can define $\mathbf{r}(t)$ as

$$\mathbf{r}(t) \;=\; \check{\mathbf{r}} + (\hat{\mathbf{r}} - \check{\mathbf{r}})\, t, \quad 0 \le t \le 1. \tag{4.5}$$

Linear interpolation is appropriate because both p_1 and P_2 move linearly through time, so their intersection must also move linearly. To compute $\check{\mathbf{r}}$, we note that it must be on both p_1 and P_2 at $t = 0$. These constraints yield the following equations:

$$\begin{aligned} \check{\mathbf{r}} \cdot \mathbf{n}_1 &= \check{c}_1 \\ \check{\mathbf{r}} \cdot \mathbf{n}_2 &= \check{c}_2 \\ \check{\mathbf{r}} \cdot \mathbf{m} &= 0. \end{aligned}$$

The values $\check{c}_1$ and $\check{c}_2$ are the plane constants for p_1 and P_2, respectively, at $t = 0$; we compute a plane's constant by taking the dot product of the plane's normal with any point on the plane, e.g., the point defined in Equation 4.4. Since we have three equations in three unknowns (the three coordinates of $\check{\mathbf{r}}$), we can solve for the value of $\check{\mathbf{r}}$ using standard matrix techniques. We can repeat this process for $t = 1$ to compute $\hat{\mathbf{r}}$. The formula for $\mathbf{r}(t)$ allows us to express any point on the intersection between p_1 and P_2 as

$$\mathbf{i}(t, k) \;=\; \mathbf{r}(t) + k\mathbf{m}, \quad 0 \le t \le 1,\ k \in \mathbb{R}. \tag{4.6}$$

Not every point $\mathbf{i}(t, k)$ corresponds to a valid intersection between f_1 and P_2. The point could be on the part of p_1 outside the boundaries of f_1, or on the part of P_2 outside the boundaries of T_2, or in the space cut off by P_1; in all of these cases, the point is not a real intersection point. We need an algorithm for finding the point $\mathbf{i}(t, k)$ with the lowest t coordinate that falls under none of these cases. Our approach is to impose constraints on $\mathbf{i}(t, k)$ that permit only real intersections. The constraints come from the interpretation of a space-time bounds as the intersection between seven half-spaces. To understand this interpretation, notice that a cubical cross-section of a hyper-trapezoid is certainly the intersection of six half-spaces, where each half-space is the volume behind the plane containing one face. We built a hyper-trapezoid from its cross-sections in such a way that it is also the intersection of six half-spaces, each one being bounded by the 4D plane that contains a face. The cutting plane defines the seventh relevant half-space.

We build a constraint expression from a half-space as follows. Consider one of the half-spaces that defines a hyper-trapezoid. Let $\mathbf{n}$ be the normal vector to a 3D cross-section of the half-space's bounding plane. If $\mathbf{p}(t)$ is a point on the bounding plane (see Equation 4.4), then the point $\mathbf{i}(t, k)$ is inside that half-space only if

$$[\mathbf{i}(t, k) - \mathbf{p}(t)] \cdot \mathbf{n} \;\le\; 0.$$

We can solve for k in terms of t to characterize those points on the intersection line at time t that satisfy the constraint. Plugging in the expression for $\mathbf{i}(t, k)$ from Equation 4.6 yields

$$[(\mathbf{r}(t) + k\mathbf{m}) - \mathbf{p}(t)] \cdot \mathbf{n} \;\le\; 0. \tag{4.7}$$

If $\mathbf{m} \cdot \mathbf{n} > 0$, then we can solve for k, getting

$$k \;\le\; \frac{[\mathbf{p}(t) - \mathbf{r}(t)] \cdot \mathbf{n}}{\mathbf{m} \cdot \mathbf{n}}.$$

When $\mathbf{m} \cdot \mathbf{n} < 0$ we get the same expression with the direction of the inequality reversed. We discuss the case in which $\mathbf{m} \cdot \mathbf{n} = 0$ in the next paragraph. Since both $\mathbf{p}(t)$ and $\mathbf{r}(t)$ are linear in t, our expression for k is also linear in t. We can therefore view our expression as a half-plane in the t-k plane, i.e., a line that separates the pairs $\langle t, k \rangle$ that correspond to valid intersection points $\mathbf{i}(t, k)$ from those pairs that give invalid intersection points.

If $\mathbf{m} \cdot \mathbf{n} = 0$, we cannot solve Equation 4.7 for k as a function of t. This dot product is zero, however, only when the intersection line is parallel to the half-space's bounding plane. If the intersection line moves behind (or, conversely, in front of) that plane at some time t, then all points on the line will become (or will stop being) valid intersection points. So we solve Equation 4.7 for t. Plugging in the definitions from Equations 4.4 and 4.5 gives us

$$t\,[\hat{\mathbf{r}} - \check{\mathbf{r}} - (\hat{\mathbf{p}} - \check{\mathbf{p}})] \cdot \mathbf{n} \;\leq\; (\check{\mathbf{p}} - \check{\mathbf{r}}) \cdot \mathbf{n}.$$

If $[\hat{\mathbf{r}} - \check{\mathbf{r}} - (\hat{\mathbf{p}} - \check{\mathbf{p}})] \cdot \mathbf{n} < 0$, then we can divide through to get a lower bound on t; this t is the time at which the intersection line moves behind the constraint half-space. For $[\hat{\mathbf{r}} - \check{\mathbf{r}} - (\hat{\mathbf{p}} - \check{\mathbf{p}})] \cdot \mathbf{n} > 0$ we get an upper bound on t. We can view either case as a another half-plane in the t-k plane, one that says any value of $\langle t, k \rangle$ gives a valid intersection point $\mathbf{i}(t, k)$ as long as t is above (below) a specific value. We discuss the case in which $[\hat{\mathbf{r}} - \check{\mathbf{r}} - (\hat{\mathbf{p}} - \check{\mathbf{p}})] \cdot \mathbf{n} = 0$ later in this section.

We now have a set of constraint half-planes in the t-k plane. The pairs $\langle t, k \rangle$ within the intersection of these half-planes satisfy the constraints and correspond to real intersection points $\mathbf{i}(t, k)$. We want to find the earliest such point, i.e., the one with the lowest t coordinate. Finding this point is an example of a two-variable linear-programming problem. Preparata and Shamos [Prep85] describe a solution to this problem, but we chose not to implement their algorithm. Instead, we extended our implementation of the Bentley-Ottmann algorithm [Bent79, Prep85] to solve this problem. The necessary extensions are quite simple and straightforward. Whenever the Bentley-Ottmann algorithm reports that two segments have intersected, we must consider what happens to the two half-spaces bounded by those segments. There are four cases, two in which the two half-spaces start intersecting, and two in which they stop intersecting. For completeness, we describe these cases in Appendix E.

We still must consider what happens when $\mathbf{m} \cdot \mathbf{n} = 0$ and $[\hat{\mathbf{r}} - \check{\mathbf{r}} - (\hat{\mathbf{p}} - \check{\mathbf{p}})] \cdot \mathbf{n} = 0$. In this case, the intersection line is parallel to the half-space's bounding plane and $\hat{\mathbf{r}} - \check{\mathbf{r}} = \hat{\mathbf{p}} - \check{\mathbf{p}}$, i.e., the intersection line and the bounding plane both move through time with the same velocity. Consequently, the intersection line is always either inside the half-space or outside it. We need only compare $\mathbf{r}(0)$ against the bounding plane at $t = 0$ to determine if the constraint is ever satisfied. In our implementation, we perform this test before solving the linear programming problem we discussed above; if this test fails, then it is impossible for all the constraints to be satisfied, so the intersection between f_1 and P_2 is always invalid.

4.4 Plane-Plane Intersections

The final category of space-time-bound intersections from Section 4.1 involves two cutting-planes, say P_1 and P_2 from space-time bounds including hyper-trapezoids T_1 and T_2, respectively. This situation is almost identical to the situation we discussed in Section 4.3. Arguing as we did at the beginning of that section, we know that P_1 can intersect P_2 only if faces of T_1 and T_2 have intersected and have been cut off. By a similar argument, we know that an intersection between two cutting planes will always have a higher t coordinate than a valid intersection between a face and a cutting plane. Equation 4.6 characterizes the intersection between P_1 and P_2 if we use the normals and plane constants from P_1 and P_2 to compute $\mathbf{m}$ and $\mathbf{r}(t)$. The discussion of constraint half-spaces still holds, the only change being that we must consider at total of twelve half-spaces: six for T_1 and six for T_2.

We have now considered all the ways two space-time bounds can intersect. To summarize the whole process, we give pseudo-code for the intersection algorithm in Figure 4.2.

```
for α ∈ {x, y, z}
    while (s1 and s2 are segments from faces in Fα that Bentley-Ottmann says intersect at ti)
        for j ∈ {1, 2}
            fj ← face associated with sj
            Tj ← hyper-trapezoid containing fj
            Pj ← cutting plane associated with Tj 's space-time bound
        if (ti cross-sections of f1 and f2 intersect)
            if (intersection rectangle is not cut off by P1 or P2)
              intersection occurs at ť_i = ti
            else
              if (f1 intersects P2, satisfying constraints from T1, T2, and P1 at t'_i)
                intersected ← TRUE
              if (f2 intersects P1, satisfying constraints from T2, T1, and P2 at t''_i)
                intersected ← TRUE
              if (intersected)
                intersection occurs at ť_i = min(t'_i, t''_i)
              else
                if (P1 intersects P2, satisfying constraints from T1 and T2 at t'''_i)
                  intersection occurs at ť_i = t'''_i.
```

Figure 4.2: This algorithm finds the time $\check{t}_i$ of the earliest intersection in a set of the space-time bounds.

5 Detecting Collisions Using Space-Time Bounds

The algorithm to find intersections between space-time bounds is the foundation of our collision-detection algorithm. In Section 5.1, we describe how the detection algorithm uses space-time bounds. Sections 5.2 and 5.3 discuss how the detection algorithm builds the space-time bounds it needs.

5.1 The Detection Algorithm

Our detection algorithm will ultimately become part of a larger graphics system, and the way our algorithm fits into that system influences our algorithm's structure[2]. The naive algorithm of Figure 2.1 suggests that the detection algorithm drives the system by repeatedly advancing the current position in "simulation time." A detection algorithm actually plays a more subservient role in the graphics system we have implemented [Zele91, Hubb91]. The current position in simulation time is controlled by another part of this system, which calls the detection algorithm whenever it needs to render a frame. When called, the detection algorithm determines if any collisions have occurred since the time of the previous frame. It returns details of any collisions it finds so the system can call the response algorithm.

To make the animation look as smooth as possible, the system usually tries to render frames at a constant rate. The calls to a detection algorithm thus come at evenly-spaced points in simulation time. As we stated in Section 2, however, the detection algorithm should not sample the agents' positions at a constant rate when checking for collisions. The detection algorithm must therefore superimpose its own adaptive sampling rate over the constant rate at which it is called. Our detection algorithm achieves this goal by repeatedly building space-time bounds and remembering the earliest time at which they intersect.

Let t denote the current time, the time at which the system calls our algorithm. When $t = 0$, the start of the simulation, our 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 3.

[2] We realize that collision detection is important for other disciplines besides computer graphics. We concentrate on the needs of graphics applications, though, and we expect that needs of other applications will be compatible.

After building, our algorithm invokes the intersection algorithm from Section 4. The intersection algorithm returns $\check{t}_i$, the normalized 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 $\check{t}_i = 1$, the normalized expiration time. Our detection algorithm sets t_{end} to be the unnormalized value of $\check{t}_i$.

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

When $t \geq t_{\text{end}}$, our detection algorithm sets t_{build} to t_{end} and rebuilds the space-time bounds. Note that the rebuilt space-time bounds incorporate data (e.g., the M and $\mathbf{d}$ values) as of time t_{build}. Our detection algorithm then calls the intersection algorithm to obtain a new t_{end}. We hope that the new t_{end} is substantially larger than t_{build} so the 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; we discuss the choice of Δt in the next paragraph. 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 should call a pair-processing algorithm on A_1 and A_2. We could use any of the techniques we summarized in that section, although we prefer the new idea we sketch in Section 7.

Say the pair-processing algorithm finds that A_1 and A_2 do not interpenetrate. In this case, our detection algorithm must advance to a time beyond t_{build} so it can 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 that is 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 2.1. Both algorithms have the same minimum temporal resolution; our algorithm has the advantage that its timestep is not fixed at Δt.

Note that if our algorithm steps from t_{build} to $t_{\text{build}} + \Delta t$ it can go past t, the simulation time at which the system called our algorithm. In this case, our algorithm can conclude that there is no collision at t within the minimum temporal resolution. Our algorithm is thus free to stop processing. Even if there were actually a collision at t—and a user could see that the system did not detect this collision when it rendered the frame at t—our algorithm would not have made an error. Our algorithm would have been accurate to within the specified Δt, which is all that we can really expect. We could add a provision to our algorithm so that if it steps over t, it explicitly invokes the pair-processing algorithm at t. We chose not to add this feature because we prefer to have our algorithm's accuracy depend only on Δt.

If our detection algorithm does step ahead to $t_{\text{build}} + \Delta t$, it must take an extra precaution when it next tests space-time bounds for intersections. The algorithm from Section 4 assumes that the interiors of all the space-time bounds are initially disjoint (contact between their surfaces does not matter). We might violate this assumption by stepping ahead: the two agents whose space-time bounds touch at t_{end} could be closer together by $t_{\text{build}} + \Delta t$, making their space-time bounds penetrate at that time. We solve this problem by performing a special test before checking space-time bounds for intersection. This special test uses the faces in the set F_x we defined in Equation 4.1. The x coordinates of these faces at normalized time 0 define a set of intervals, one for each space-time bound. If a pair of space-time bounds are to penetrate initially, the pair of corresponding intervals must intersect. Our special test steps through the interval endpoints in ascending x-coordinate order, keeping track of those intervals that intersect and checking the corresponding space-time bounds for initial penetration. This process is quite efficient, especially because we already need to sort the interval endpoints before using the Bentley-Ottmann algorithm [Bent79, Prep85] to find face intersections. Even so, we want to avoid running this special test when it is unnecessary. To that end, our detection algorithm sets the flag *could_overlap* to TRUE only when the special test is needed, and we assume the space-time-bound intersection algorithm invokes the special test only when that flag is set.

Our discussion so far has implicitly assumed that only two agents can collide at any given time. This assumption is not actually correct. As we mentioned earlier in this section, space-time bounds describe collisions to within a certain accuracy, and a pair-processing algorithm gives greater accuracy. It is possible for several pairs of agents to collide at the accuracy of their space-time bounds but not at the accuracy of

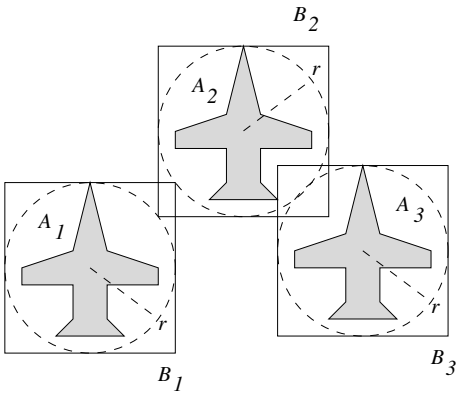

Figure 5.1: Agents A_1, A_2 and A_3 each have radius r. At the current time, the space-time bounds B_1 and B_2 intersect, as do B_2 and B_3, but none of the agents actually collide. Note that this diagram shows only the cross-sections of B_1, B_2 and B_3 as of the current time.

```
t_build ← t_end
while (t ≥ t_end)
    rebuild space-time bounds as of t_build
    ť_i ← earliest intersection between space-time bounds
    t_end ← unnormalize(ť_i)
    if (t_end − t_build < Δt)
      could_overlap ← TRUE
      if (ť_i < 1, i.e., a pair of space-time bounds really intersect)
        for each pair of intersecting space-time bounds B_1, B_2
            if (pair-processing algorithm finds that B_1's and B_2's agents penetrate at t_build)
              then collision occurs at t_build
      t_end ← t_build + Δt
    else
      could_overlap ← FALSE
    t_build ← t_end
there is no collision as of t
```

Figure 5.2: This algorithm determines if a set of agents have collided as of time t.

the pair-processing algorithm. Figure 5.1 gives an example. The detection algorithm must call the pair-processing algorithm for every pair of agents whose space-time bounds intersect. Once one pair of space-time bounds starts intersecting, the detection algorithm will start stepping ahead by Δt. Each time it does so, it will set the *could_overlap* flag and invoke the special penetration test. This test reports all the pairs of space-time bounds that penetrate, allowing the detection algorithm to pass their agents to the pair-processing algorithm.

We have now described the structure of our detection algorithm. We summarize the algorithm in the pseudo-code of Figure 5.2.

5.2 Rebuilding Strategies

An important part of the detection algorithm is the rebuilding of the space-time bounds. The algorithm could ask the graphics system for $\mathbf{x}(t_{\text{build}})$, $\dot{\mathbf{x}}(t_{\text{build}})$ and the latest M and $\mathbf{d}$ for each agent, and use this data to rebuild *all* the space-time bounds. This approach is always correct, but it is not always the most efficient strategy.

A second approach is to rebuild only the agents whose space-time bounds intersected last time. For the other agents, we can continue using their current space-time bounds until they expire. The advantage of this approach is the time we save by not rebuilding those other space-time bounds. Rebuilding an agent's space-time bound involves two main tasks: computing the position and velocity of the agent, and updating the data structures that represent the space-time bound itself. The first task can be time-consuming if the motion of each agent is determined by ordinary differential equations (ODEs), e.g., if we are simulating Newtonian physics. In this case, when our algorithm asks the graphics system for an agent's $\mathbf{x}(t_{\text{build}})$ and $\dot{\mathbf{x}}(t_{\text{build}})$, the system will invoke a numerical method to solve the ODEs.

The common ODE solvers [Pres88] iterate through simulation time, building solutions at later times from solutions at earlier times. Knowing that ODE solvers work in this manner, one might be surprised that the computation of $\mathbf{x}(t_{\text{build}})$ and $\dot{\mathbf{x}}(t_{\text{build}})$ adds any extra processing time. The simulation time t_{build} will fall between two times at which the system renders frames; letting δt be the frame rate, we call these frame times $j\,\delta t$ and $(j+1)\delta t$. When the ODE solver computes the values at t_{build}, it iterates from the values at $j\,\delta t$ that it has already computed. One might think it would take no longer to iterate from the values at $j\,\delta t$ to t_{build} and then on to $(j+1)\delta t$ than to iterate from $j\,\delta t$ directly to $(j+1)\delta t$. In our tests, however, the latter is consistently faster. This speed probably comes from a characteristic of the ODE solver we use. This solver, a fourth-order Runge Kutta method [Pres88], chooses the size of its iteration steps adaptively. When it is iterating over a longer interval of simulation time, it has more opportunity to stretch out its steps without losing accuracy. When it is forced to stop at t_{build}, however, it takes more steps, resulting in poorer performance.

Rebuilding only the space-time bounds of the agents that collided has a potential disadvantage. A space-time bound is conservative, and at t coordinates near its expiration time it is likely to bound a large amount of space. As the current simulation time gets nearer the expiration time, false-alarm intersections are more likely. We would like our algorithm to avoid wasting time on false alarms when possible. In an attempt to overcome this disadvantage, we implemented a third rebuilding strategy. The idea is to rebuild the space-time bounds of the colliding agents as of t_{build}, and rebuild the space-time bounds of the other agents as of $j\,\delta t$, the previous frame time. Since our ODE solver will have already computed its values as of $j\,\delta t$, this strategy does not force the ODE solver to make extra stops that inhibit its adaptive stepping. In our tests, this third strategy was faster than the basic strategy of rebuilding all space-time bounds, but it was slightly slower than our second strategy. The slowness was due to the overhead of manipulating the data structures for all N space-time bounds instead of just a few of them. The extra false alarms introduced by the second strategy were not as big a factor as we had feared.

Both the second and third rebuilding strategies create some space-time bounds that start at a later unnormalized t coordinate than the other space-time bounds. The intersection algorithm from Section 4 assumes that all the space-time bounds start at the same t coordinate. We arrive at this common starting coordinate by chopping off the early part of the other space-time bounds. This chopping process is simple to implement, so we do not describe it.

5.3 Application-Specific Information

An application which uses our detection algorithm must be able to give the algorithm some specific information. The minimum temporal resolution Δt (see Section 5.1) could be set to a generic default value by our algorithm, but an application is likely be more successful if the application writer chooses the value of Δt. The acceleration bounds M and $\mathbf{d}$ (see Section 3) are more important. The application must be able to supply these values for any agent every time our detection algorithm rebuilds the space-time bounds.

We cannot guarantee that every application will be able to compute these values simply and efficiently. But many applications should be able to approximate these values well enough to take advantage of our detection algorithm. Here, we summarize a few broad classes of applications and why we believe each one can compute M and $\mathbf{d}$:

- Before a new vehicle is built, a design team might want to create a simulator that allows users to "test drive" the vehicle. Since the designers will want to know if a user can control the vehicle without hitting other objects, the simulator should call a collision-detection algorithm. Most vehicles give the user some means of specifying straight-ahead motion and turning. When the user chooses to go straight ahead, it should be simple for the simulator to infer values for M and $\mathbf{d}$. When the user chooses to turn, computing M and $\mathbf{d}$ is more complicated. The simulator could compute these values from an estimate of the vehicle's centripetal acceleration. The centripetal acceleration will always point towards the center of the turn. Thus, a $\mathbf{d}$ orthogonal to the vehicle's velocity at the beginning of the turn will hold until vehicle turns 180 degrees. The magnitude of the centripetal acceleration increases with the sharpness of the turn. The simulator can set M based on a guess of the sharpness; if the user ever exceeds this guessed sharpness, the simulator can force the detection algorithm to rebuild the space-time bounds with an updated value.

- The general concept of "docking" is important in the real world. For example, engineers need to know how pilots can bring spacecraft together in flight, and chemists need to understand the way drug molecules fit into receptor sites [Turk90, Broo90]. Computer simulations allow users to gain insight into docking tasks, and these simulations should incorporate collision detection to provide the most realism. In these simulations, users directly manipulate objects with a mouse or some other input device. The simulation system might have difficulty getting acceleration bounds for a directly-manipulated object. As a solution, the system could simulate attachment of the manipulated object to the user's cursor by means of a virtual spring. The virtual spring allows the system to convert cursor displacements into the needed acceleration bounds according to Hooke's law [Gold50]. The disadvantage of the virtual spring is the extra lag it introduces, but we expect this lag would not be a problem in most applications; sometimes this lag is actually desirable, as it smoothes out the jitters in the manipulated object's motion.

- Some computer simulations require a user to navigate through a crowded space of agents. If the system is to detect collisions between the user-controlled viewpoint and the other agents, then it will need acceleration bounds for the user's position. Mackinlay *et al.* [Mack90] have published an appealing approach to navigation. The user points at a destination with a mouse cursor and the system moves the user's viewpoint to that destination; the speed with which the viewpoint moves decreases logarithmically as the destination draws nearer. This characteristic of the viewpoint's speed will allow the system to compute acceleration bounds for the viewpoint.

The accuracy with which these applications can compute acceleration bounds will have some effect on the performance of our detection algorithm. Tighter acceleration bounds will give better performance, but we believe our algorithm will still perform well if an application uses simpler calculations to arrive at looser bounds.

The length of time before the acceleration bounds expire also effects our algorithm's performance. Our algorithm does no work between the times at which it has to rebuild the space-time bounds. If acceleration bounds expire infrequently, the time between rebuildings is likely to be longer. The expiration time of the whole set of space-time bounds is the earliest expiration time of any agent's acceleration bounds; this choice is necessary because the algorithm from Section 4 assumes all space-time bounds cover the same time period. An application should therefore make an effort to maximize the earliest expiration time. It should also be careful about reusing acceleration bounds it computed earlier. For example, say an application knows the acceleration bounds for A_1 from time 0 to time 11, and for $A_2, \ldots, A_N$ from time 0 to time 10. If time 10 arrives without any collisions, our algorithm will ask the application for new acceleration bounds. Assume the application could now determine the acceleration bounds for all agents from time 10 to time 20. The application does not really need to recompute the bounds for A_1, since the known bounds have not yet

expired. But reusing these bounds means the earliest expiration time would be at most one time unit later, at time 11. The application would do better to go ahead and recompute the bounds of A_1 to get the later expiration time. We say the acceleration bounds are *unsynchronized* if the application sacrifices a later expiration time in a misguided attempt to avoid some recomputation.

6 Performance

The best test of our detection algorithm will come from a real application. Applications take time to develop, however, so we decided to conduct some simpler but more contrived tests. We feel these tests give some insight into the average-case performance of our algorithm. The test results are positive enough that we believe our algorithm will perform well in real applications.

We have no idea what a truly "average" case is for a detection algorithm. The applications that use collision detection are so diverse that we could think of no unifying characteristics. Consequently, we decided to randomize as many details of our tests as possible. By repeatedly generating random configurations of agents and timing our algorithm as it detected the collisions, we hoped to get a general impression of the performance we could expect in any real application.

The number of agents involved in a test configuration affects the overall amount of work our detection algorithm must perform during that test. We thus ran sequences of tests, with the number of agents per test remaining constant throughout each sequence. The details of a particular test in a sequence are as follows. For each agent, we generated a random width and represented the agent as an isothetic cube of that width; we chose this simple geometry so we would not have to implement an elaborate pair-processing algorithm. We then gave the agent a random initial position within a sphere (the sphere's radius also remained constant throughout the sequence). We chose the initial velocity vector of each agent randomly, with the direction coming from the unit sphere. To generate these and all other random numbers, we used the UNIX `drand48` routine, which draws numbers from a uniform distribution.

To control the agent's motion, we generated a random sequence of linear accelerations to affect the agent's center (we ignored rotational motion for simplicity). Each acceleration had a random duration, and started acting on the agent as soon as the previous acceleration stopped acting. For the purpose of computing the acceleration directions and magnitudes, we grouped the accelerations into a random number of subsequences (between one and six subsequences per agent in our tests). All the accelerations in a subsequence have directions within one randomly-oriented hemisphere and magnitudes less than one upper bound; a subsequence is thus a group of accelerations for which we know M and $\mathbf{d}$. We made the subsequences of all agents start and stop at the same points in simulation time, thus avoiding the unsynchronized acceleration bounds we discussed in Section 5.3. Although we pre-specified the accelerations affecting each agent throughout the test, we did not give the detection algorithm any "future" information (other than M and $\mathbf{d}$). We therefore believe we accurately simulated how a user might interactively control an agent by periodically applying control forces (accelerations).

To put the performance of our algorithm in perspective, we compared it with the performance of Turk's algorithm [Turk90] on the same test configurations. We chose Turk's algorithm for several reasons. Recall from Section 2 that Turk's algorithm solves the all-pairs weakness by dividing space into fixed-sized sectors and comparing only pairs of agents within the same sector. Turk's empirical results suggest that this approach is quite efficient. It is also quite simple to implement, since a spatial hashing function assigns agents to the appropriate sectors. Turk's algorithm does not address the fixed-timestep weakness—it involves an outer loop exactly like the loop in the naive algorithm of Figure 2.1—but none of the published algorithms that do address this weakness can handle agents whose motion is determined by accelerations.

Each test we ran involved the following procedure. We called our detection algorithm once every δt units of simulation time, δt being the frame rate (which remained constant throughout the sequence of tests). We stopped when either the end of simulation time arrived or our algorithm detected a collision; we did not invoke a collision-response algorithm because our work focuses on detection. For a minimum temporal resolution, we used $\Delta t = \delta t/2$. We are not certain if we chose correct the relationship between δt and Δt, but

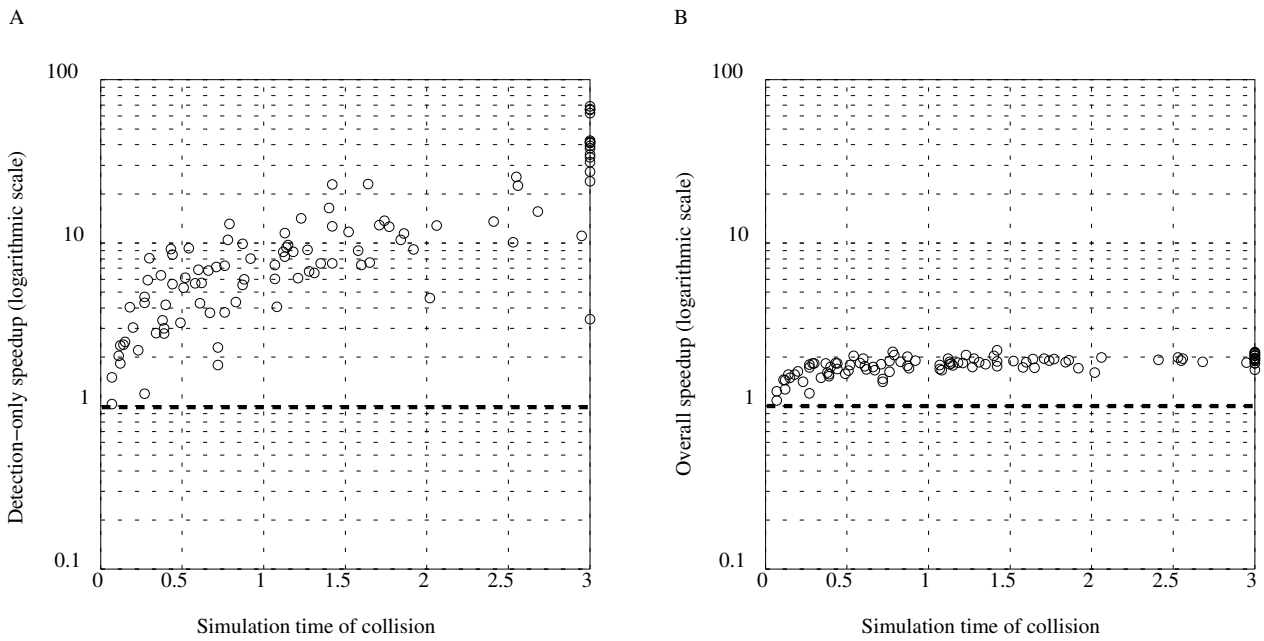

Figure 6.1: Figure A shows the detection-only speedup of our algorithm over Turk's algorithm for 100 tests with 30 agents per test. Figure B shows the overall speedup. Remember that speedup is defined to be the time spent by Turk's algorithm divided by the time spent by our algorithm.

we think it is sensible to use more accuracy when computing motion than when viewing it since a mistake in the computed motion affects the entire future course of the simulation. After every call to our algorithm, we also simulated the rendering of the frame by computing the position of each agent; these computations are important to make certain the ODE solver behaves the way it would behave in a real application (see Section 5.2). We timed the whole process by calling the UNIX routine `clock` before the first call and after the last call. We then timed Turk's algorithm being called every Δt—not δt—time units; we used Δt as the time step in order to make certain Turk's algorithm had the same minimum temporal resolution as ours. In another attempt at fairness, we let Turk's algorithm use as much memory for its hash table as our algorithm used for all its data structures.

The timings we obtained from this procedure measure the overall time of each simulation. The overall time includes the time spent by the ODE solver to compute the position of each agent. Since the Runge Kutta ODE solver we used performs a substantial amount of computation, we made additional measurements to get a feeling for how much of the overall time was spent actually detecting collisions. By bracketing each call to the ODE solver with calls to `clock`, we computed the time spent solving ODEs. We subtracted this time from the overall time to get the "detection-only" time. Originally, we feared the time spent by `clock` itself might skew our results, but execution profiles indicate that this problem did not occur.

We ran three test sequences. The machine we used was a DECstation 5000/200 PGX Turbo with 40MB RAM. All our code was written in C++ and compiled by an AT&T 3.0.1 compiler with −O2 optimizations. The first sequence involved 100 tests with 30 agents per test, the second sequence 200 tests with 100 agents per test, and the third sequence 100 tests with 300 agents per test. In all three tests, we used the frame rate $\delta t = 0.02$ time units (giving $\Delta t = 0.01$) and we used 3.0 as the end of simulation time. Figures 6.1, 6.2 and 6.3 graph the *speedup* of our algorithm for each test in each sequence. Following Patterson and Hennessy [Patt90], we define the speedup of our algorithm to be the time spent by Turk's algorithm divided by the time spent by our algorithm. Thus, any speedup greater than 1 indicates an advantage for our algorithm, and the larger the speedup the better. We use a logarithmic scale for these graphs to avoid hiding "slowdowns" (i.e., speedups less than 1).

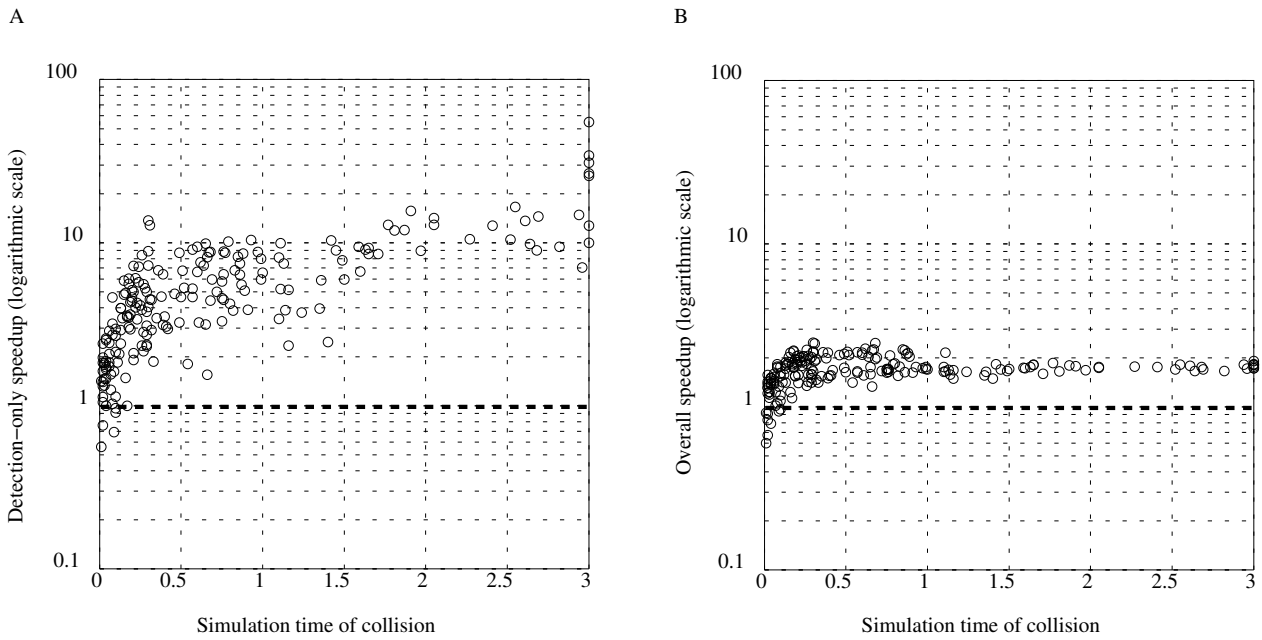

Figure 6.2: Figure A shows the detection-only speedup of our algorithm over Turk's algorithm for 200 tests with 100 agents per test. Figure B shows the overall speedup. Remember that speedup is defined to be the time spent by Turk's algorithm divided by the time spent by our algorithm.

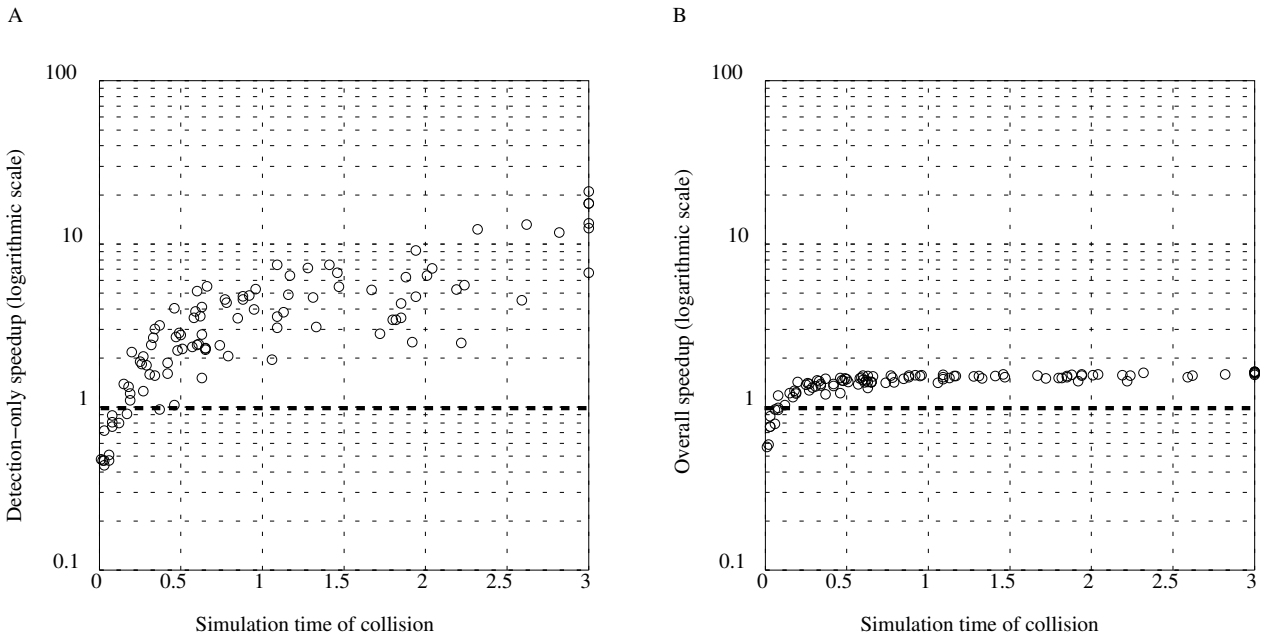

Figure 6.3: Figure A shows the detection-only speedup of our algorithm over Turk's algorithm for 100 tests with 300 agents per test. Figure B shows the overall speedup. Remember that speedup is defined to be the time spent by Turk's algorithm divided by the time spent by our algorithm.

The graphs show that our algorithm was almost always faster than Turk's algorithm. The detection-only graphs show a particularly striking pattern of large speedups. The less dramatic speedups in the overall graphs suggest to us that the performance of the ODE solver is still an important factor in the performance of a simulation system. We anticipate that the overall speedup of a system using our algorithm would increase if the system used a faster ODE solver (e.g., a second-order Runge Kutta method). Notice that the detection-only speedup tends to increase with the length of time before a collision occurs. This trend makes sense because a later collision time lengthens the period over which the cost of our algorithm is amortized. Note also that the detection-only speedup is not as large for the tests involving 300 agents as for the tests involving 30 agents. We do not interpret this difference as evidence that our algorithm will not scale. Instead, we believe it shows that our tests did not adequately explore the effects of agent density. As we mentioned earlier, the initial position of each agent in a test was bounded by a sphere. The radius of this sphere remained fixed throughout all tests in a sequence, but the relationship of the radius between sequences was quite arbitrary. In particular, the average number of agents per unit volume in our 30-agent tests was 42 times the number for our 300-agent tests. Turk's algorithm is likely to perform better when the agents are less dense because the expected number of agents per sector decreases with density. Thus, our 300-agent tests provided a situation particularly suited to Turk's algorithm. In the future, we would like to conduct another sequence of 300-agent tests with a higher density of agents to see if the speedup of our algorithm improves.

The goal of our research is making collision detection fast enough for interactive applications. In these applications, we care not only about the performance of the system summed over all frames but also about the performance at each individual frame. In particular, we would like the worst-case per-frame performance to be good enough that the graphics system can maintain a constant and high frame rate. To study the per-frame performance of our detection algorithm, we repeated the three sequences of tests with a different timing scheme. In these tests, our system called `clock` before and after it processed each individual frame. We recorded the maximum time spent on any one frame over the course of a single test. Once again, we subtracted the time spent by the ODE solver to obtain the detection-only time. Figures 6.4, 6.5 and 6.6 graph speedup of our algorithm's maximum frame time.

These graphs indicate that in most tests, the worst-case per-frame performance of Turk's algorithm was better than ours. We suspected, however, that frames nearly as slow as the slowest frame are less common with our algorithm. To test this suspicion, we recorded the time each algorithm spent on each frame and put these times into histograms. We built one histogram per algorithm for each test sequence. Figures 6.7, 6.8 and 6.9 give the histograms for the 30-agent, 100-agent and 300-agent sequences, respectively. These histograms show that our algorithm produces many more fast frames than Turk's algorithm. In the case of the 100-agent tests, the histogram shows that the worst of our worst cases is almost twice as fast as the worst of Turk's worst cases. This pattern is reversed for the 300-agent tests, but as we mentioned earlier, we believe the density of the agents was a factor in this sequence of tests. We draw two conclusions from these histograms. First, the 100-agent tests suggest that our algorithm's slowest frame can be faster than Turk's when we consider a broad sampling of situations. Second, the tests as a whole suggest that the cases in which our algorithm produces slower frames than Turk's algorithm are indeed rare. We hope that users will tolerate these occasional slower frames, but we cannot be certain until we have tried our algorithm in a real application.

7 Conclusions and Extensions

In this paper, we have presented a new approach to collision detection. Section 2 identified the weaknesses of previous algorithms that we addressed in our work. We argued in Section 4.2 that our algorithm avoids the all-pairs weakness by using the Bentley-Ottmann algorithm [Bent79, Prep85] to find hyper-trapezoid intersections. In Section 5.1, we showed how our algorithm avoids the fixed-timestep weakness by tracking the intersections of space-time bounds. The timings from Section 6 indicate that our algorithm does indeed perform well in some test situations.

We are currently incorporating our detection algorithm into a sample application, a spaceship flight-

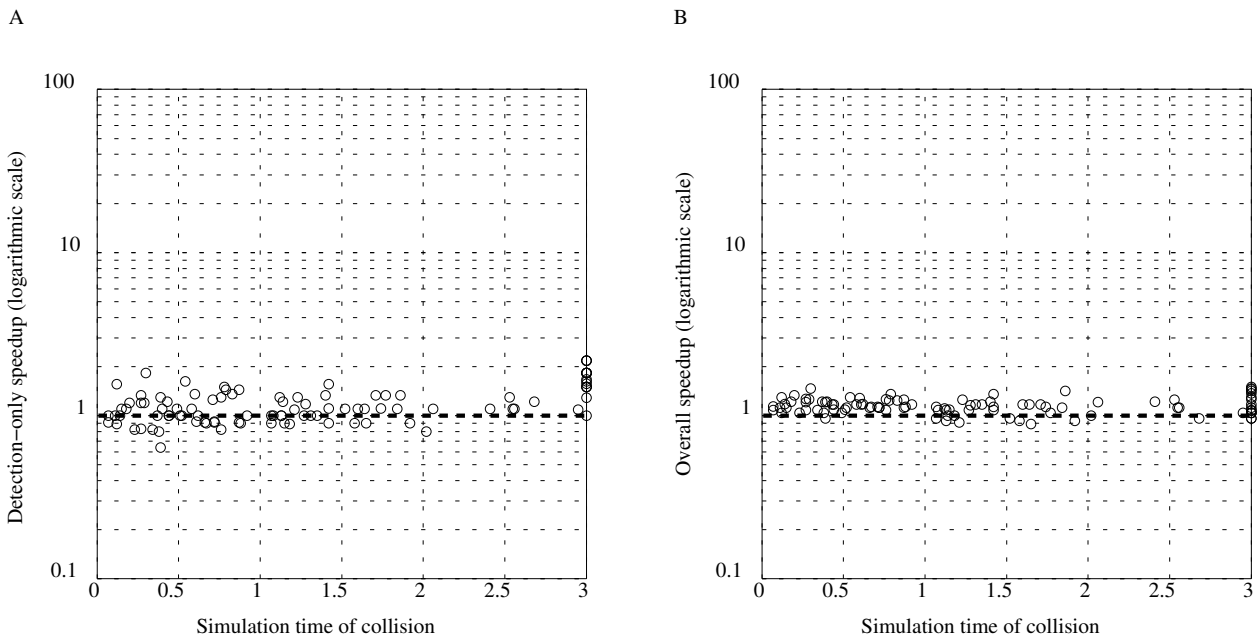

Figure 6.4: Figure A shows the speedup of our algorithm's maximum single-frame detection-only time for 100 tests with 30 agents per test. Figure B shows the speedup for maximum single-frame time including ODE-solving time. Remember that speedup is defined to be the time spent by Turk's algorithm divided by the time spent by our algorithm.

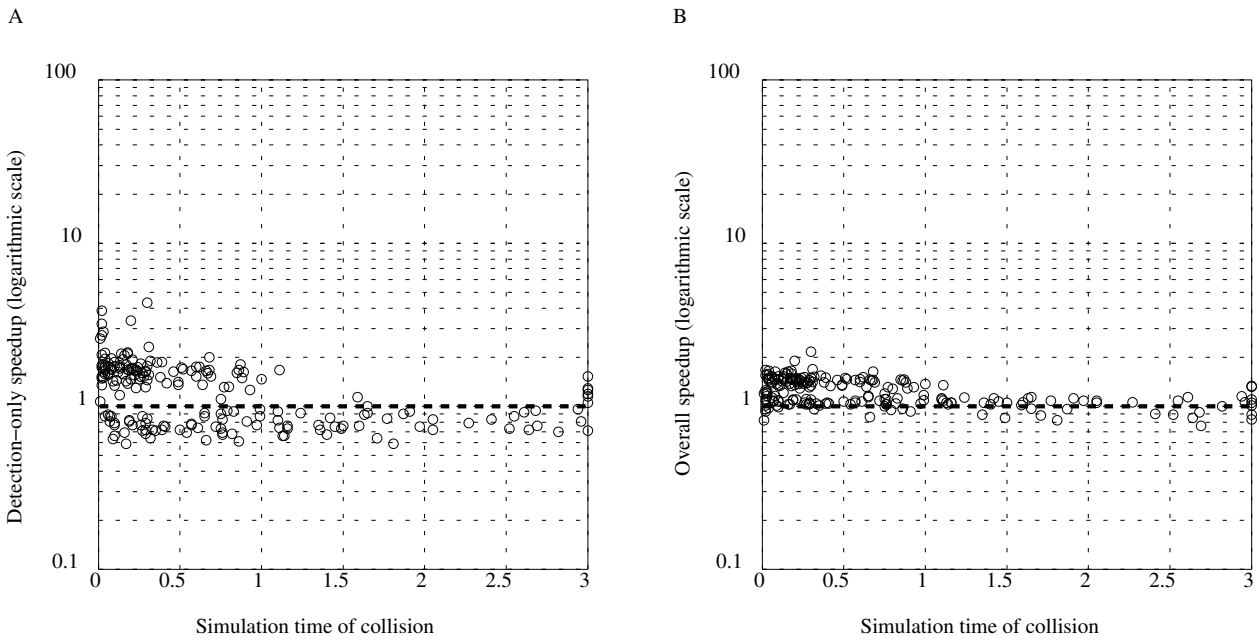

Figure 6.5: Figure A shows the speedup of our algorithm's maximum single-frame detection-only time for 200 tests with 100 agents per test. Figure B shows the speedup for maximum single-frame time including ODE-solving time. Remember that speedup is defined to be the time spent by Turk's algorithm divided by the time spent by our algorithm.

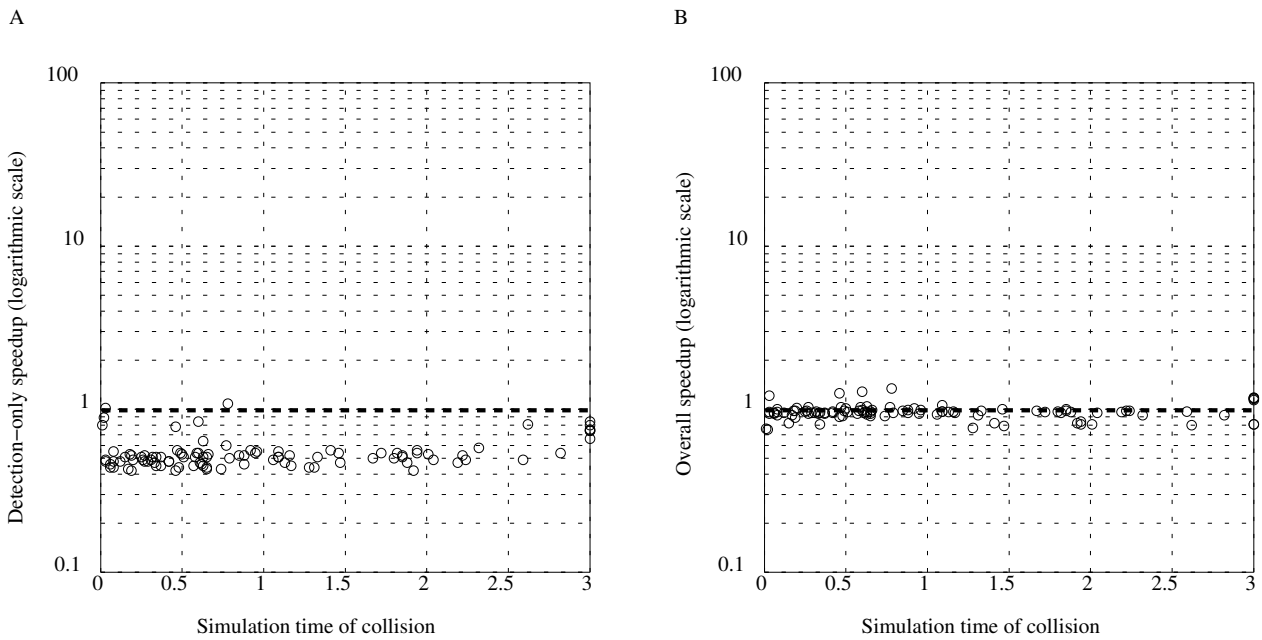

Figure 6.6: Figure A shows the speedup of our algorithm's maximum single-frame detection-only time for 100 tests with 300 agents per test. Figure B shows the speedup for maximum single-frame time including ODE-solving time. Remember that speedup is defined to be the time spent by Turk's algorithm divided by the time spent by our algorithm.

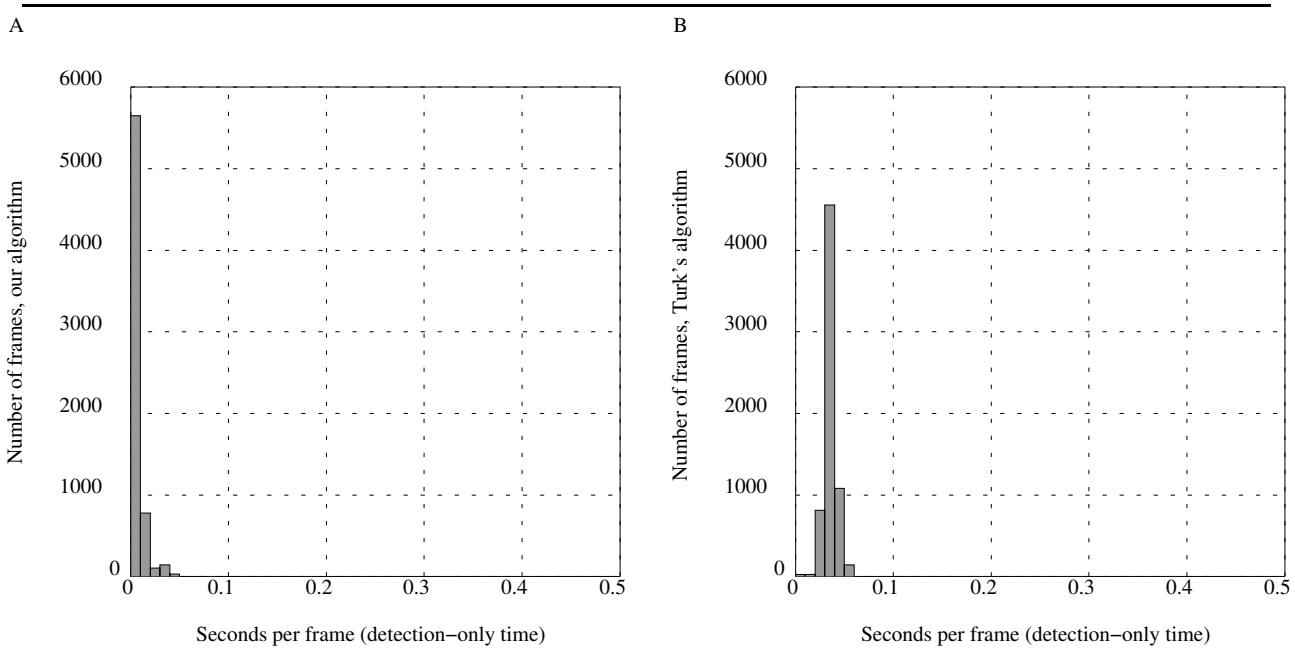

Figure 6.7: Figure A shows our algorithm's detection-only time for every frame of the 100 30-agent tests. Figure B shows the times for Turk's algorithm.

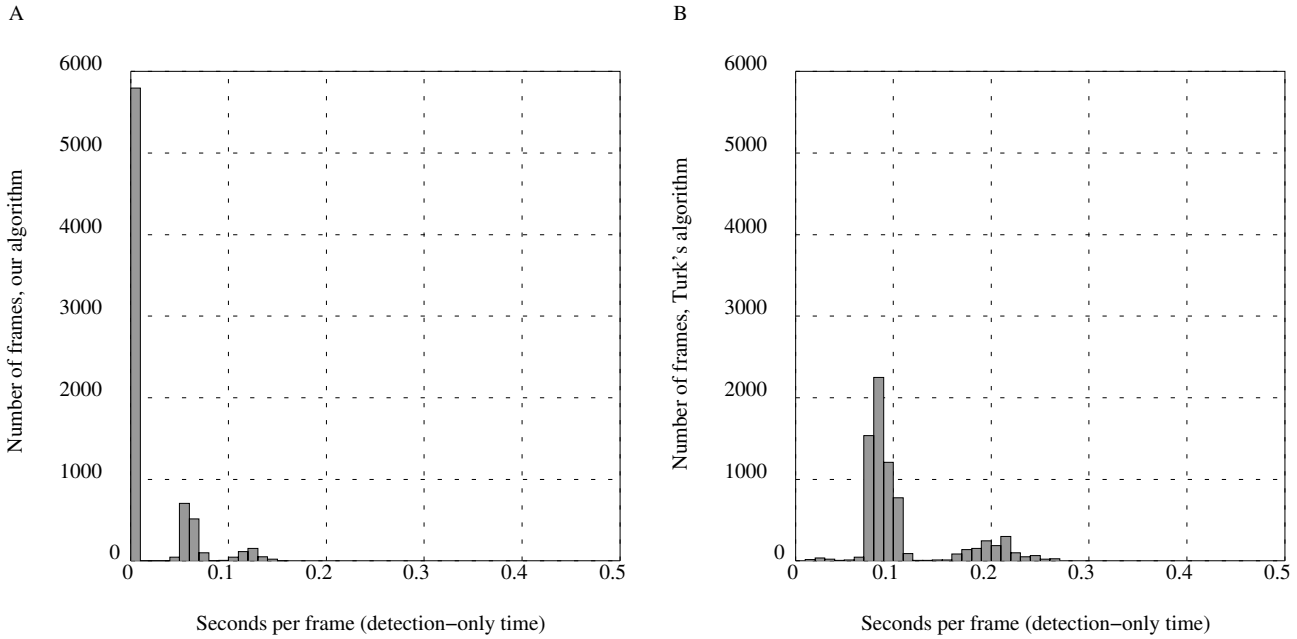

Figure 6.8: Figure A shows our algorithm's detection-only time for every frame of the 200 100-agent tests. Figure B shows the times for Turk's algorithm.

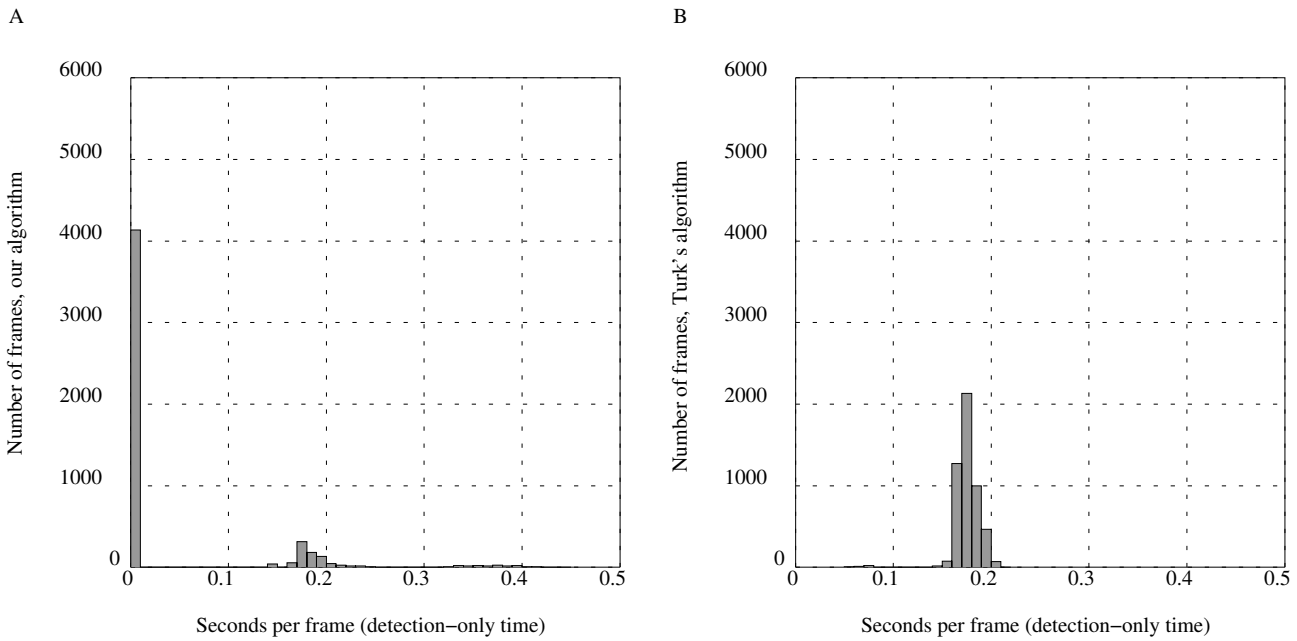

Figure 6.9: Figure A shows our algorithm's detection-only time for every frame of the 100 300-agent tests. Figure B shows the times for Turk's algorithm.

simulator. This application will provide useful information about the performance of our algorithm. It will also test our assumption that applications can compute the acceleration bounds M and $\mathbf{d}$ without much difficulty.

We also intend to explore variations on the algorithm we present in this paper. It may be possible to support other forms of 4D bounds that have advantages over space-time bounds in some situations. For example, consider an application in which vehicles drive through a maze. If a wall of the maze is not isothetic, its space-time bound will not provide a tight bound. The looseness of the bound will increase the number of false-alarm collisions detected by our algorithm when vehicles drive near the wall. A better bound for the wall would be a 4D plane, like a cutting plane without the associated hyper-trapezoid. Another variation is space-time bounds built from velocity or position bounds instead of acceleration bounds. Consider a tethered object rotating in a circle. If the object is rotating quickly, its centripetal acceleration will be large. The tether bounds the object's position to the interior of a circle, however, so we know it can never collide with anything outside that circle.

The one weakness with the naive algorithm that we have not addressed is the pair-processing weakness. We believe the best way to attack this problem is to use approximation. A detection algorithm should not always compute the inter-penetration between two agents with the utmost accuracy, ignorant of the processing time required. Instead, a detection algorithm should be responsive to the needs of the application that is calling it. When the application discovers that its other tasks are taking too long, it should be able to tell the detection algorithm to spend less time computing a less-accurate, approximate result. Notice the analogy between this idea and progressive refinement in rendering [Berg86, Cohe88]; the success of that approach to rendering gives us hope that our idea will be useful.

To support this sort of operation, a detection algorithm must have a special structure. We believe the algorithm should consist of a sequence of steps. Each step will characterize the collisions or lack thereof to a certain degree of accuracy, so the application can interrupt the algorithm after any step. Each step will also refine the results of the previous step, so the quality of the results will improve if the application decides to allow more steps to be completed. The application might base this decision on the "importance" of the agents involved, where importance is some application-specific concept. As an example, one measure of importance is the proximity of the colliding agents to the agent being controlled by the user.

The detection algorithm we described in this paper could be the first step in the sequence, since it finds collisions to the accuracy of a cube around each agent. If our algorithm discovers that the cubes around two agents do collide, the next step could compare the geometry of these agents with more accuracy. This next step should not immediately look for the exact contact between the two agents. Instead, it should use an approximation to the geometry of the agents that is a little more accurate than the cubes. If the agents still collide at this level of approximation, the next step would use an even more accurate approximation. Ideally, the geometric approximations would eventually converge on the real geometry of the agents.

We have not found this sort of geometric approximation discussed anywhere in the literature. The work of Sclaroff and Pentland [Scla91, Pent91] introduces one way to approximate an agent's geometry (recall Section 2), but the authors do not discuss how this approximation might be made more or less accurate to fit the available processing time. We believe the topic of geometric approximation deserves further investigation.

8 Acknowledgements

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

A Appendix: Proof of Inequality 3.1

We are given $\mathbf{x}(0)$, $\dot{\mathbf{x}}(0)$, and a scalar M such that

$$|\ddot{\mathbf{x}}(t)| \;\leq\; M, \quad 0 \leq t \leq \hat{t}. \tag{A.1}$$

Our goal is to prove that

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

Start by defining a new function

$$\mathbf{a}(t) \;=\; \mathbf{x}(t) - \mathbf{x}(0). \tag{A.2}$$

Note that

$$\begin{aligned} \mathbf{a}(0) &= 0, \\ \dot{\mathbf{a}}(t) &= \dot{\mathbf{x}}(t), \\ \ddot{\mathbf{a}}(t) &= \ddot{\mathbf{x}}(t). \end{aligned}$$

Using this new function, the goal becomes proving that

$$|\mathbf{a}(t) - \dot{\mathbf{a}}(0)t| \;\leq\; \frac{M}{2}t^2, \quad 0 \leq t \leq \hat{t}.$$

Define a second new function

$$\mathbf{b}(t) \;=\; \mathbf{a}(t) - \dot{\mathbf{a}}(0)t. \tag{A.3}$$

Note that

$$\begin{aligned} \mathbf{b}(0) &= 0, \\ \dot{\mathbf{b}}(0) &= 0, \end{aligned}$$

and

$$\ddot{\mathbf{b}}(t) \;=\; \ddot{\mathbf{a}}(t) \;=\; \ddot{\mathbf{x}}(t). \tag{A.4}$$

Our goal is now to show that

$$|\mathbf{b}(t)| \;\leq\; \frac{M}{2}t^2. \tag{A.5}$$

By applying the Fundamental Theorem of Calculus twice, we get

$$\mathbf{b}(t) \;=\; \int_0^t \dot{\mathbf{b}}(u)\,du \;=\; \int_0^t \int_0^u \ddot{\mathbf{b}}(v)\,dv\,du. \tag{A.6}$$

The Triangle Inequality allows us conclude that

$$|\mathbf{b}(t)| \;=\; \left|\int_0^t \int_0^u \ddot{\mathbf{b}}(v)\,dv\,du\right| \;\leq\; \int_0^t \int_0^u \left|\ddot{\mathbf{b}}(v)\right|\,dv\,du.$$

Our hypothesis (Inequality A.1) and the equality in Equation A.4 yield

$$\int_0^t \int_0^u \left|\ddot{\mathbf{b}}(v)\right|\,dv\,du \;\leq\; \int_0^t \int_0^u M\,dv\,du.$$

This integral evaluates to $(M/2)t^2$, so we have reached our goal of Inequality A.5.

B Appendix: Proof of Inequality 3.2

We are given $\mathbf{x}(0)$, $\dot{\mathbf{x}}(0)$, and a vector $\mathbf{d}$ such that

$$\ddot{\mathbf{x}}(t) \cdot \mathbf{d} \;\leq\; 0, \quad 0 \leq t \leq \hat{t}. \tag{B.1}$$

Our goal is to prove that

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

This proof follows the same pattern as the proof in Appendix A. Using the definitions of $\mathbf{a}(t)$ and $\mathbf{b}(t)$ from Equations A.2 and A.3, respectively, our goal becomes showing that

$$\mathbf{b}(t) \cdot \mathbf{d} \;\leq\; 0, \quad 0 \leq t \leq \hat{t}. \tag{B.2}$$

Equation A.6 still holds, so

$$b(t) \cdot \mathbf{d} \;=\; \left[\int_0^t \int_0^u \ddot{\mathbf{b}}(v)\, dv\, du\right] \cdot \mathbf{d}.$$

Since integration is linear, we can push the dot product inside to get

$$\left[\int_0^t \int_0^u \ddot{\mathbf{b}}(v)\, dv\, du\right] \cdot \mathbf{d} \;=\; \int_0^t \int_0^u \left[\ddot{\mathbf{b}}(v) \cdot \mathbf{d}\right] dv\, du.$$

Using Inequality B.1 and Equation A.4 produces

$$\int_0^t \int_0^u \left[\ddot{\mathbf{b}}(v) \cdot \mathbf{d}\right] dv\, du \;\leq\; \int_0^t \int_0^u 0\, dv\, du.$$

Since this integral evaluates to 0, we have arrived at our goal of Inequality B.2.

C Appendix: Normalizing Time

In Section 3, we considered how to build a space-time bound when we know $\mathbf{x}(0)$ and $\dot{\mathbf{x}}(0)$ and when M is defined over the time interval $[0, \hat{t}]$. We now want to be able to get the same space-time bound when we consider the time interval to be not $[0, \hat{t}]$ but $[0, 1]$.

First, we consider the velocity $\dot{\mathbf{x}}(0)$. Since this vector lies in $\mathbb{R}^3$, our change of the time scale will not affect its direction. So we need only rescale its magnitude. Originally, $|\dot{\mathbf{x}}(0)|$ specified that a certain distance d will be traveled in $\hat{t}$ time units. Now, we want the same distance to be traveled in 1 time unit. So we want the new magnitude to be $\Delta x \,/\, 1 = |\dot{\mathbf{x}}(0)|\, \hat{t}$.

The only other quantity to be rescaled is the acceleration bound M. Since M is a constant acceleration, it specifies that velocity will change by a certain magnitude Δv in $\hat{t}$ time units. We now want the same change to occur in 1 time unit. Thus, we want to use the new acceleration bound $M' = \Delta v \,/\, 1 = M\hat{t}$.

D Appendix: Round-Off Errors in the Bentley-Ottmann Algorithm

The Bentley-Ottmann algorithm [Bent79, Prep85] finds all the intersections within a set of 2D line segments. The foundation of the algorithm is a linear ordering on the segments at any abscissa t. To define the ordering, place a vertical line at t and rank the segments according to where they intersect the line. In Figure D.1, for example, the ordering at the abscissa $t = 0$ is s_4, s_2, s_1, s_3. Two segments can intersect only if they are adjacent in the ordering at some abscissa. Since we are considering only segments that start at $t = 0$ and end at $t = 1$, the ordering changes only at the intersection of two segments.

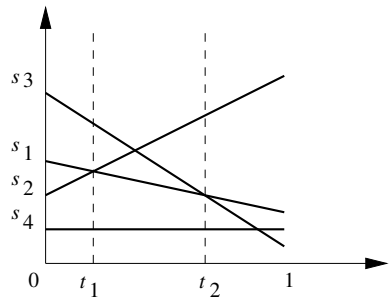

Figure D.1: A set of four segment being processed by the Bentley-Ottmann algorithm.

The algorithm starts by comparing the adjacent segments in the $t = 0$ ordering. The earliest intersection between two such segments—call them s_1 and s_2—is the first point at which the ordering changes. At the abscissa of this intersection, s_1 and s_2 swap places in the ordering. After this swap, two new pairs of segments may intersect: s_1 can intersect s_4, the segment below s_2 in the old ordering, and s_2 can intersect s_3, the segment above s_1 in the old ordering. Figure D.1 shows an example of this situation at abscissa t_1. If either pair of segments intersects, the algorithm has found another abscissa at which the ordering changes. The algorithm puts this abscissa in a priority queue with all the other intersections it has detected but not processed. The algorithm then advances to the earliest abscissae from this queue and repeats the whole process.

When the algorithm processes an ordering swap and finds a pair of segments that intersect, the algorithm must remember if it has already found this "new" intersection. As an example, consider abscissa t_2 in Figure D.1. Here, s_1 and s_2 become adjacent in the ordering, but their intersection has already been processed, at t_1. Surprisingly, neither published description of the Bentley-Ottmann algorithm describes how to accurately determine if an intersection has already been processed. One might be tempted to use the abscissa of the "new" intersection as follows: if it is less than the abscissa of the most-recently-processed intersection, one would conclude that the "new" intersection has already been processed. Unfortunately, this approach is prone to round-off errors [Pres88]. When the algorithm computes an intersection abscissa, it uses floating-point computations that are not guaranteed to be exactly correct. To see how these inaccuracies can cause problems, refer again to Figure D.1. When the algorithm is processing the intersection at t_1, the value it computes for t_2 might incorrectly be less than t_1. In this case, the algorithm would skip the intersection at t_2 entirely. Thus, the algorithm needs a different method of detecting when it has already processed an intersection.

Our implementation uses a hash table to explicitly store every processed intersection. The algorithm ignores a "new" intersection if the hash table already contains a record for that intersection. The hash key is based on the data-structure addresses for the two intersecting segments, so the algorithm can never miss an intersection. Managing a hash table takes longer than comparing two floating-point numbers, but our execution profiles indicate that the extra time does not significantly change the overall performance of our detection algorithm.

E Appendix: Using the Bentley-Ottmann Algorithm for Two-Variable Linear Programming

The particular linear-programming problem we are trying to solve is as follows: given up to 13 constraint half-spaces in $\mathbb{R}^2$, find the point within the intersection of the half-spaces with the lowest abscissa t. The maximum number of half-spaces is 13 because we must consider the constraints from at most two hyper-trapezoids and one cutting plane.

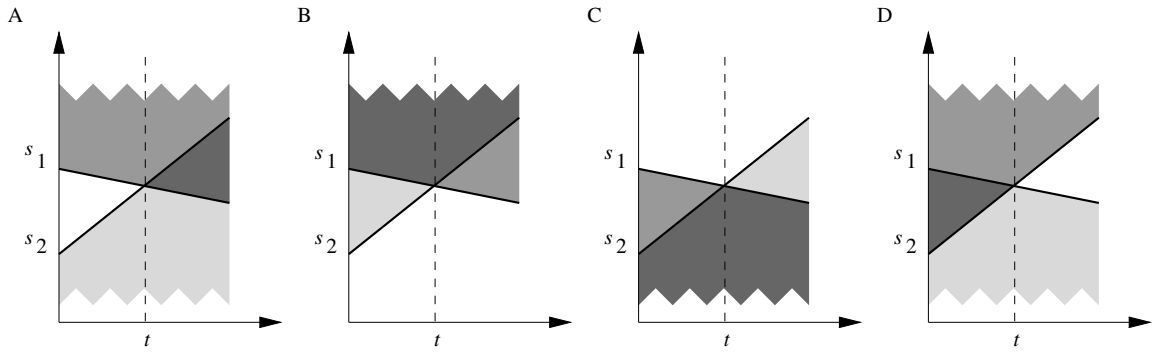

Figure E.1: The four cases for half-space intersections.

Each half-space is defined by a bounding line segment. If two half-spaces are not already intersecting at $t = 0$, the place where they start intersecting corresponds to the intersection between their bounding segments. Thus, the point with the lowest abscissa that lies within all the half-spaces must be the point at which some pair of bounding segments intersect. As we discuss in Appendix D, the Bentley-Ottmann algorithm [Bent79, Prep85] finds the intersections between segments in order of increasing abscissa; the algorithm thus provides most of the functionality we need to solve our linear-programming problem.

To get the remaining functionality, we must make the algorithm keep track of half-spaces as well as segments. Consider what can happen when two bounding segments s_1 and s_2 intersect at abscissa t. Without loss of generality, assume s_1 is above s_2 just before the intersection. If s_2 marks the upper bound of a half-space H_2 and s_1 marks the lower bound of a half-space H_1, as in Figure E.1A, then the intersection of s_1 and s_2 is the first point in $H_1 \cap H_2$. Note that any points on s_1 or s_2 with abscissa greater than t will also be in $H_1 \cap H_2$. The algorithm remembers this fact by adding H_1 and H_2 to the *membership* lists of s_1 and s_2. Each segment has such a list, which records the half-spaces in which the segment currently lies. Since the maximum number of half-spaces is 13, we can implement a membership list efficiently as a bit vector. If the combined memberships of s_1 and s_2 include all the half-spaces, then the algorithm has solved the linear-programming problem.

If s_2 marks the lower bound of half-space H_2, then the situation is different; see Figure E.1B. As of abscissa t, all the points on s_1 are no longer in H_2 (the fact that H_2 has as linear boundary means it can never "curl around" and engulf s_1 again). Thus, the algorithm updates the membership of s_1 so that it is no longer in H_2. If s_1 and s_2 are both upper-bounds, the situation is analogous; see Figure E.1C. In this case, s_2 has left H_1, and the algorithm updates its membership accordingly.

A final case occurs if s_1 an upper bound and s_2 is a lower bound, as depicted in Figure E.1D. As of this intersection, there can be no points that lie in both H_1 and H_2. Thus, the algorithm can immediately conclude that there is no solution to the linear-programming problem.

It is possible for the bounding segment of a half-space to be a vertical line. Our algorithm handles these half-spaces as a special case. Considering only these half-spaces, the feasible region (if it exists) will be a single interval of abscissae. Our algorithm essentially computes this interval and uses it to check the validity of a solution to the other constraints.

References

[Bara89] Baraff, David, "Analytical Methods for Dynamic Simulation of Non-Penetrating Rigid Bodies," the Proceedings of SIGGRAPH '89, published as *Computer Graphics*, Vol. 23, No. 3, July 1989, pp. 223–232.

[Bara90] Baraff, David, "Curved Surfaces and Coherence for Non-Penetrating Rigid Body Simulation," the Proceedings of SIGGRAPH '90, published as *Computer Graphics*, Vol. 24, No. 4, August 1990, pp. 19–29.

[Bara91] Baraff, David, "Coping with Friction for Non-Penetrating Rigid Body Simulation," the Proceedings of SIGGRAPH '91, published as *Computer Graphics*, Vol. 25, No. 4, July 1991, pp. 31–40.

[Bara92a] Baraff, David, *Dynamic Simulation of Non-Penetrating Rigid Bodies*, Ph.D. Dissertation, Department of Computer Science, Cornell University, Technical Report 92-1275, March 1992.

[Bara92b] Baraff, David and Andrew Witkin, "Dynamic Simulation of Non-Penetrating Flexible Bodies," The Proceedings of SIGGRAPH '92, published as *Computer Graphics*, Vol. 26, No. 2, July 1992, pp. 303–308.

[Bent79] Bentley, Jon L. and Thomas A. Ottmann, "Algorithms for Reporting and Counting Geometric Intersections," *IEEE Transactions on Computers*, Vol. C-28, No. 9, September 1979, pp. 643–647.

[Berg86] Bergman, Larry, Henry Fuchs, Eric Grant and Susan Spach, "Image Rendering by Adaptive Refinement," The Proceedings of SIGGRAPH '86, published as *Computer Graphics*, Vol. 20, No. 4, August 1986, pp. 29–37.

[Boys79] Boyse, John W., "Interference Detection among Solids and Surfaces," *Communications of the ACM*, Vol. 22, No. 1, January 1979, pp. 3–9.

[Broo90] Brooks, Frederick P., Jr., Ming Ouh-Young, James J. Batter and P. Jerome Kilpatrick, "Project GROPE: Haptic Displays for Scientific Visualization", Proceedings of SIGGRAPH '90, published as *Computer Graphics*, Vol. 24, No. 4, August 1990, pp. 177–185.

[Came85] Cameron, Stephen A., "A Study of the Clash Detection Problem in Robotics," *Proceedings 1985 IEEE International Conference on Robotics and Automation*, pp. 488–493.

[Came89] Cameron, Stephen A., "Efficient Intersection Tests for Objects Defined Constructively," *International Journal of Robotics Research*, Vol. 8, No. 1, February 1989, pp. 3–25.

[Came90] Cameron, Stephen A., "Collision Detection by Four-Dimensional Intersection Testing," *IEEE Transactions on Robotics and Automation*, Vol. 6, No. 3, June 1990, pp. 291–302.

[Cann86] Canny, John, "Collision Detection for Moving Polyhedra," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 8, No. 2, pp. 200–209, March 1986.

[Cohe88] Cohen, Michael F., Shenchang Eric Chen, John R. Wallace and Donald P. Greenberg, "A Progressive Refinement Approach to Fast Radiosity Image Generation," The Proceedings of SIGGRAPH '88, published as *Computer Graphics*, Vol. 22, No. 4, August 1988, pp. 75–84.

[Cull86] Culley, R. K., and K. G. Kempf, "A Collision Detection Algorithm Based on Velocity and Distance Bounds," *Proceedings of the 1986 IEEE International Conference on Robotics and Automation*, Vol. 2, 1986., pp. 1064–1069.

[Dobk83] Dobkin, D. P. and D. G. Kirkpatrick, "Fast Detection of Polyhedral Intersection," *Theoretical Computer Science*, Vol. 27, 1983, pp. 241–253.

[Duff92] Duff, Tom, "Interval Arithmetic and Recursive Subdivision for Implicit Functions and Constructive Solid Geometry," The Proceedings of SIGGRAPH '92, published as *Computer Graphics*, Vol. 26, No. 2, July 1992, pp. 131–138.

[Elli78] Ellis, Robert and Denny Gulick, *Calculus with Analytic Geometry*, Harcourt Brace Jovanovich (New York), 1978.

[Fois90] Foisy, André, Vincent Hayward and Stéphane Aubry, "The Use of Awareness in Collision Prediction," *Proceedings of the 1990 IEEE International Conference on Robotics and Automation*, pp. 338–343.

[Gold50] Goldstein, Herbert, *Classical Mechanics*, Addison-Wesley (Reading, Massachusetts), 1950.

[Hahn88] Hahn, James K., "Realistic Animation of Rigid Bodies," The Proceedings of SIGGRAPH '88, published as *Computer Graphics*, Vol. 22, No. 4, August 1988, pp. 299-308.

[Herm86] Herman, Martin, "Fast, Three-Dimensional, Collision-Free Motion Planning," *Proceedings of the 1986 IEEE International Conference on Robotics and Automation*, Vol. 2, 1986, pp. 1056-1063.

[Hubb91] Hubbard, Philip M., Matthias M. Wloka and Robert C. Zeleznik, "UGA: A Unified Graphics Architecture," Technical Report CS-91-30, Department of Computer Science, Brown University, July 1991.

[Luba91] Lubachevsky, Boris D., "How to Simulate Billiards and Similar Systems," *Journal of Computational Physics*, Vol. 94, No. 1, May 1991, pp. 255-283.

[Mack90] Mackinlay, Jock D., Stuart K. Card and George G. Robertson, "Rapid Controlled Movement Through a Virtual 3D Workspace," Proceedings of SIGGRAPH '90, published as *Computer Graphics*, Vol. 24, No. 4, August 1990, pp. 171-176.

[Moor88] Moore, Matthew and Jane Wilhelms, "Collision Detection and Response for Computer Animation," The Proceedings of SIGGRAPH '88, published as *Computer Graphics*, Vol. 22, No. 4, August 1988, pp. 289-298.

[Patt90] Patterson, David A. and John. L. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kauffmann (San Mateo, California), 1990.

[Pent91] Pentland, Alex and Stan Sclaroff, "Closed-Form Solutions for Physically Based Shape Modelling and Recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 13, No. 7, July 1991, pp. 715-729.

[Prep85] Preparata, Franco P. and Michael I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag (New York), 1985.

[Pres88] Press, William H., Brian P. Flannery, Saul A. Teukolsky and William T. Vetterling, *Numerical Recipes in C*, Cambridge University Press (Cambridge), 1988.

[Same85] Samet, Hanan and Markku Tamminen, "Bintrees, CSG Trees, and Time," The Proceedings of SIGGRAPH '85, published as *Computer Graphics*, Vol. 19, No. 3, July 1985, pp. 121-130.

[Scla91] Sclaroff, Stan and Alex Pentland, "Generalized Implicit Functions for Computer Graphics," The Proceedings of SIGGRAPH '91, published as *Computer Graphics*, Vol. 25, No. 4, July 1991, pp. 247-250.

[Snyd92] Snyder, John M., "Interval Analysis for Computer Graphics," The Proceedings of SIGGRAPH '92, published as *Computer Graphics*, Vol. 26, No. 2, July 1992, pp. 121-130.

[Turk90] Turk, Greg, "Interactive Collision Detection for Molecular Graphics," Technical Report TR90-014, Computer Science Department, The University of North Carolina at Chapel Hill, March 1990.

[VonH89] Von Herzen, Brian, *Applications of Surface Networks to Sampling Problems in Computer Graphics*, Ph.D. Dissertation, California Institute of Technology, Computer Science Department, Caltech-CS-TR-88-15, 1989.

[VonH90] Von Herzen, Brian, Alan H. Barr and Harold R. Zatz, "Geometric Collisions for Time-Dependent Parametric Surfaces," The Proceedings of SIGGRAPH '90, published as *Computer Graphics*, Vol. 24, No. 4, August 1990, pp. 39-48.

[Zele91] Zeleznik, Robert C., D. Brookshire Conner, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard, Brian Knep, Henry Kaufman, John F. Hughes and Andries van Dam, "An Object-Oriented Framework for the Integration of Interactive Animation Techniques," The Proceedings of SIGGRAPH '91, published as *Computer Graphics*, Vol. 25, No. 4, July 1991, pp. 105-112.