IEEE Transactions on Visualization and Computer Graphics, 1(3), Sept. 1995, pp. 218–230

Collision Detection for Interactive Graphics Applications *

Philip M. Hubbard
Program of Computer Graphics
Cornell University†

Abstract

Collision detection and response are important for interactive graphics applications such as vehicle simulators and virtual reality. Unfortunately, previous collision-detection algorithms are too slow for interactive use. This paper presents a new algorithm for rigid or articulated objects that meets performance goals through a form of time-critical computing. The algorithm supports progressive refinement, detecting collisions between successively tighter approximations to object surfaces as the application allows it more processing time. The algorithm uses simple four-dimensional geometry to approximate motion, and hierarchies of spheres to approximate three-dimensional surfaces at multiple resolutions. In a sample application, the algorithm allows interactive performance that is not possible with a good previous algorithm. In particular, the new algorithm provides acceptable accuracy while maintaining a steady and high frame rate, which in some cases improves on the previous algorithm's rate by more than two orders of magnitude.

Index items: collision detection, time-critical computing, real-time performance, interaction, four dimensions, approximation.

1 Introduction

The physical world is filled with solid objects. When solid objects collide, they do not penetrate one another (unless they flex or break into pieces). A computer simulation of the physical world will seem more natural and believable if its objects exhibit this property. To enforce the "solidness" of objects, a simulation system relies on a *collision-handling algorithm*.

Researchers in computer graphics and robotics have developed a variety of collision-handling algorithms. Typically, such an algorithm consists of two parts. The *detection algorithm* determines whether simulated objects would penetrate one another if the collision-handling algorithm were to leave them alone. The *response algorithm* corrects the behavior of the objects that would penetrate. Both parts of a collision-handling algorithm pose interesting problems, but this paper focuses on detection algorithms. For a discussion of response algorithms, see the work of Baraff [1].

Despite the wealth of literature on detection algorithms, no published algorithms adequately address the needs of interactive applications, such as vehicle simulators or virtual reality. To work in an interactive application, a detection algorithm must satisfy two criteria. First, the algorithm must support real-time performance (a high and nearly-constant frame rate, and low latency), even when many complex objects can collide. Second, the algorithm must tolerate objects whose motion is guided by a user, that is, objects whose motion is not fully pre-specified.

This paper presents a detection algorithm for rigid or articulated objects that meets these criteria through approximation. The algorithm detects collisions between simplified representations of objects. These representations support progressive refinement; having detected collisions between representations that fit objects with one level of accuracy, the algorithm can advance to more accurate representations with further processing. After each accuracy improvement, the algorithm allows itself to be interrupted by the application program. The application thus has the flexibility to degrade accuracy when speed is important, for example, to avoid latency, which may give users a feeling of motion sickness [15].

†Mailing address: 580 Frank H. T. Rhodes Hall, Cornell University, Ithaca, NY 14853-3801. Email address: pmh@graphics.cornell.edu.

This approach is analogous, in essence, to rendering at multiple levels of quality, as explored by Bergman *et al.* [2], Funkhouser and Séquin [8], and Maciel and Shirley [17]. The success of these rendering techniques suggests the value of this approach in collision detection. All these ideas are elements of *time-critical computing*, which stresses that the time an algorithm spends producing its results can be as important as the correctness of those results.

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

These two forms of approximation cooperate to make the detection algorithm significantly faster than previous algorithms. In a sample application, one of the best previous algorithms often requires more than 10 seconds per frame, preventing the application from maintaining real-time performance. The new algorithm, in comparison, allows low latency and a nearly-constant frame rate of 10 frames per second; note that this rate can exceed the previous algorithm's rate by more than two orders of magnitude.

The remainder of this paper is structured as follows. Section 2 reviews the collision-detection literature. Section 3 gives an overview of the new algorithm. Section 4 introduces space-time bounds, and their use in the detection algorithm is developed in Sections 5 and 6. Section 7 describes sphere-trees, and Section 8 sketches an algorithm that builds sphere-trees automatically. Some aspects of the latter topic are subtle, so we omit details that appear elsewhere [14]; the current paper thus emphasizes the overall structure of the detection algorithm. Section 9 discusses the performance of the algorithm. Finally, Section 10 summarizes this work and some issues it raises.

2 Previous Work

2.1 A Basic Algorithm

One way to evaluate previous detection algorithms is to note how they improve on a basic algorithm. Consider a simulation that animates N objects, $O_0, \ldots, O_{N-1}$. The system that runs this simulation maintains an internal sense of *simulation time*, distinct from the *wallclock time* experienced by a human observer; simulation time runs from t_0 to t_1. Fig. 1 sketches pseudocode for the simulation system. It renders a frame of the animation every Δt_r units of simulation time. Before rendering each frame, it calls a basic detection algorithm. The next three paragraphs discuss three problems with this basic algorithm.

The first problem concerns the outermost loop of the detection algorithm. This loop increments the time variable, t, by a fixed timestep Δt_d (assumed to be smaller than the rendering timestep, Δt_r). A larger Δt_d makes the algorithm faster, because it samples simulation time less frequently. A smaller Δt_d, however, makes the algorithm more accurate, because it is less likely to miss collisions between samples. The only way to get speed and accuracy is to use an adaptive timestep: the size of the timestep should become smaller when collisions are likely. The basic algorithm's inability to adaptively change its timestep is the *fixed-timestep weakness*.

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

The basic algorithm's third problem involves the "if" statement within the innermost loop. An algorithm that checks two objects' surfaces for penetration at a particular time t is a *pair-processing algorithm*. Implementing such an algorithm can be tricky. For example, consider the case of objects with polyhedral surfaces. All the ways that faces, edges, and vertices can contact one another complicate the algorithm, giving it many special cases that are inefficient to execute and difficult to debug. A detection algorithm suffers from the *pair-processing weakness* if its pair-processing algorithm is not robust and efficient.

```
simulation()
    for t ← t_0 to t_1 in steps of Δt_r
        get user input
        update behavior of objects {O_0, ..., O_{N-1}}
        if (detect(t, {O_0, ..., O_{N-1}}) finds collision)
          do collision response
        render objects {O_0, ..., O_{N-1}}

/* Variable t_prev persists between calls. */
detect(t_curr, {O_0, ..., O_{N-1}})
    for t ← t_prev to t_curr in steps of Δt_d
        for each object O_i ∈ {O_0, ..., O_{N-1}}
            move O_i to its position at time t
        for each object O_i ∈ {O_0, ..., O_{N-1}}
            for each object O_j ∈ {O_{i+1}, ..., O_{N-1}}
                if (O_i penetrates O_j)
                  collision occurs at simulation time t
    t_prev ← t_curr
```

Figure 1: A basic detection algorithm and a simulation that uses it.

2.2 Basic Algorithm Improvements

Many authors address the basic algorithm's weaknesses. To do justice to all this previous work would require a full paper in itself, so this section presents only a few highlights. The solutions discussed here are ingenious, but few of them address all three weaknesses. Furthermore, the algorithms that most nearly solve all three impose restrictions on the shapes or motion of objects, making these algorithms unsuitable for interactive applications.

To eliminate the fixed-timestep weakness, some algorithms use 4D geometry, the fourth dimension being simulation time. Canny [3], for example, derives functions of a time variable that express how convex polyhedra change over time; the roots of these functions denote collisions. Duff [6] uses interval analysis to quickly identify and refine the regions in space and time that could contain collisions. This approach is one of the few to address all three weaknesses. Unfortunately, it assumes that objects' motions over time are fully known in advance. Thus, an interactive application cannot use this algorithm. Recent advances in interval analysis, such as the work of Snyder *et al.* [27], still retain this disadvantage. Variations on these algorithms would apply to 4D geometry that estimates where objects could be in the future without assuming fully pre-specified motion; this idea is related to the approach we develop in this paper, and Section 5.2.2 will discuss one use of interval analysis in this context.

Some researchers address the fixed-timestep weakness without assuming fully pre-specified motion. Culley and Kempf [5] use bounds on velocity and distance to increase Δt_d when collisions cannot occur. Their technique cannot handle nonlinear motion, nor does it address the all-pairs weakness, but the idea inspired the concept of space-time bounds, to be introduced in Section 4. Foisy *et al.* [7] use 4D structures somewhat similar to space-time bounds to solve the fixed-timestep weakness. Their approach also addresses the all-pairs weakness in some—but not all—situations. Lin *et al.* [16] use similar 4D analysis in concert with an efficient technique for tracking the distance between two convex polyhedra. The algorithm incorporates extensions for nonconvex objects, but the authors do not analyze the performance of the algorithm in practice.

Space subdivision is a popular way to mitigate the all-pairs weakness (although $\Omega(N^2)$ worst cases still exist for some object shapes). The algorithms of Moore and Wilhems [18], Shaffer and Herb [25] and Smith *et al.* [26] exemplify the use of octrees by many researchers. Turk [29] presents a simpler technique that divides space uniformly; while Turk's application was molecular modeling using spheres, the approach works for the bounding spheres or boxes of arbitrary objects, efficiently reducing the number of objects whose surfaces must be tested for intersection (by some other method). Cohen *et al.* [4] use simple spatial sorting to find intersections between bounding boxes; this sorting exploits interframe coherence to reduce its workload. Although these algorithms do not address the fixed-timestep or pair-processing weaknesses, they are flexible enough to handle motion that is not fully pre-specified. Turk's algorithm and the Cohen *et al.* algorithm, in particular, are efficient because of their low overhead.

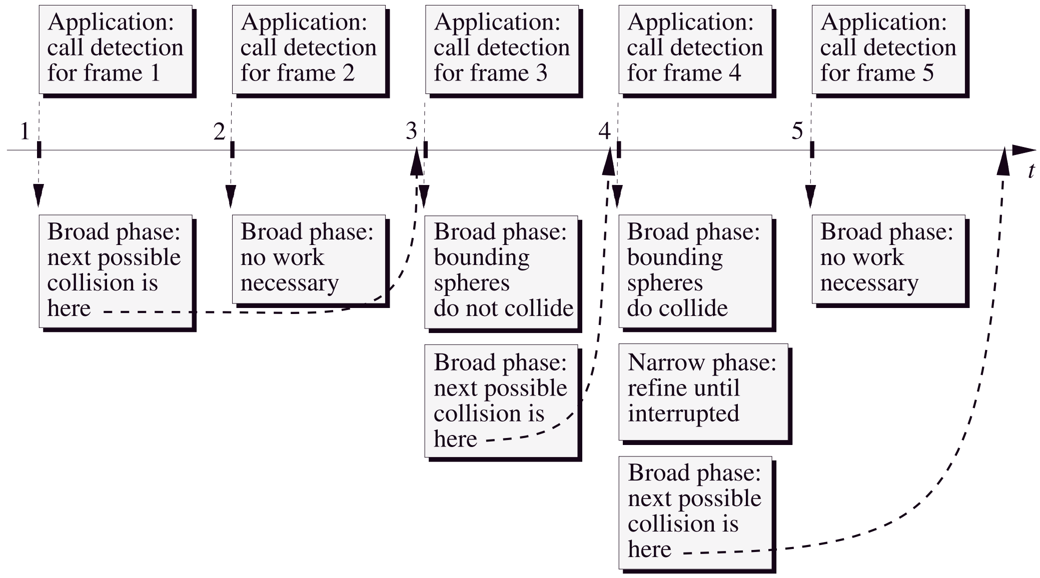

Figure 2: The broad phase and narrow phase in action.

Thibault and Naylor [28] show how binary space partitioning (BSP) trees address the pair-processing weakness for polyhedral objects. An advantage of BSP trees is that for rigid objects, the structure of a tree can be optimized to improve performance; Naylor [19] describes several optimization techniques. To accelerate their pair-processing algorithm, Garcia-Alonso *et al.* [9] associate a precomputed regular grid with each object; as objects rotate, however, the grids lose some of their efficiency. Sclaroff and Pentland [24] present a pair-processing algorithm that approximates surfaces with deformed superquadrics. The authors do not demonstrate that their approach works for a wide variety of surfaces, but they are the first authors to show that approximation can make collision detection more efficient.

To understand the performance of a new detection algorithm, the best evaluation is direct, empirical comparison with previous algorithms. These comparisons require considerable effort, however, so they are not common in the literature. Our contribution in this respect is modest, as we compare our algorithm against only two previous approaches, Turk's algorithm and a BSP-tree algorithm, in Section 9. The efficient organization of these algorithms makes them both straightforward to implement and quick to execute. Although we know of no published demonstrations that other algorithms improve on these algorithms in a variety of situations, further comparisons of our algorithm to other algorithms is a worthwhile area of future work.

3 Algorithm Overview

Section 1 introduced a detection algorithm that progressively refines its accuracy. The algorithm operates as a sequence of steps, allowing itself to be interrupted by the application after each step.

The first step of the algorithm is the *broad phase*. The broad phase uses upper bounds on the objects' accelerations to build a set of *space-time bounds*, the 4D structures that Section 4 will derive. A space-time bound gives a conservative estimate of where an object could be in the future. The lowest time coordinate, $t = t_i$, at which a pair of space-time bounds intersect is the earliest time at which a collision between objects is possible. Until t_i arrives, the broad phase need not do any more work; it can immediately conclude there is no collision at each intervening frame. Space-time bounds thus address the fixed-timestep weakness. Section 5 will explain how space-time bounds also address the pair-processing weakness.

When time t_i arrives, the broad phase focuses on the two objects, O_1 and O_2, whose space-time bounds intersect. It checks for intersection between these objects' bounding spheres, located at the objects' $t = t_i$ positions. If the bounding spheres do not intersect then there is no collision; the broad phase thus builds a new set of space-time bounds, computes a new t_i and repeats. If, on the other hand, the bounding spheres do intersect, then the broad phase has found a collision between O_1 and O_2 at the coarsest level of detail.

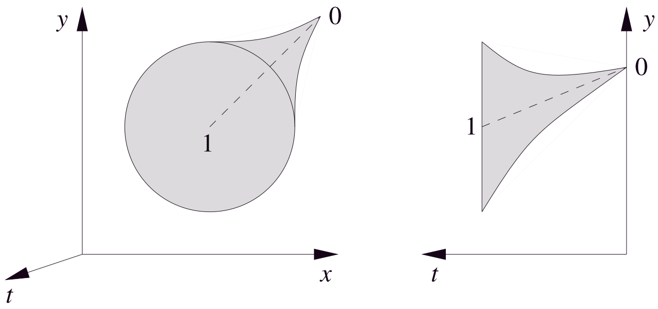

Figure 3: Two views of a parabolic horn for an object moving in $\mathbb{R}^2$. The point labeled "0" is $\mathbf{x}(0)$ and the point labeled "1" is $\mathbf{x}(0) + \dot{\mathbf{x}}(0)t|_{t=1}$.

In this case, the algorithm switches to the *narrow phase*. The narrow phase progressively refines the accuracy of the $t = t_i$ collision. More specifically, the narrow phase repeatedly determines if approximations to the surfaces of O_1 and O_2 intersect at $t = t_i$, using progressively more accurate approximations. After each repetition, the narrow phase allows itself to be interrupted by the application, so the application gets accuracy proportional to the time it can spare for collision detection. For each object, the approximations are sets of spheres arranged in a hierarchy called a *sphere-tree*, as Section 7 will describe. Each level of a sphere-tree fits its object more tightly than the previous one, and the hierarchy allows the narrow phase to base more accurate intersection tests on the results of less accurate ones. Sphere-trees make each narrow-phase step efficient and straightforward to implement, and thus address the pair-processing weakness.

Once the narrow phase terminates—when a refinement step finds no collision between O_1 and O_2, or the application interrupts—the algorithm returns to the broad phase and the cycle repeats. Fig. 2 shows a schematic diagram of the broad phase and narrow phase in action. Section 6 will describe this algorithm more precisely.

4 Space-Time Bounds

This section defines *space-time bounds*, the 4D structures used by the broad phase. For the moment, assume that each object, O, is a point. Let $\mathbf{x}(t), \dot{\mathbf{x}}(t), \ddot{\mathbf{x}}(t) \in \mathbb{R}^3$ denote the position, velocity and acceleration, respectively, of O at simulation time t. An interactive application knows these vectors at the current time—say, $t = 0$—but not for the future. If, however, the application knows a scalar, M, such that

$$|\ddot{\mathbf{x}}(t)| \;\leq\; M$$

holds until some future time—say, $t = 1$—then the unknown $\mathbf{x}(t)$ is subject to the following inequality:

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

This assertion is related to Taylor's theorem; a proof is not difficult and it appears elsewhere [14].

Inequality 1 is important for its geometric interpretation. It states that the unknown position $\mathbf{x}(t)$ will be within a distance $(M/2)t^2$ from the known position $\mathbf{x}(0) + \dot{\mathbf{x}}(0)t$. Thus, a 3D bound on O's position at t is a sphere of radius $(M/2)t^2$ centered at $\mathbf{x}(0) + \dot{\mathbf{x}}(0)t$. A 4D bound on O's position over all times $t \in [\,0, 1\,]$ is the structure whose cross section at t is that sphere. To better understand this 4D structure, consider an analogy in one less dimension: O is now in $\mathbb{R}^2$, and a structure that bounds O's movement over time has three dimensions. This structure's cross sections will be circles of radius $(M/2)t^2$ instead of spheres. Fig. 3 shows a bounding structure in this case. Due to the factor of t^2 present in the definition, the bounding structure is a *parabolic horn*. Note that it is a *conservative* bound on O's position over time, in that O cannot be outside the horn for $t \in [\,0, 1\,]$.

An extension of this idea covers an object, O, that is not merely a point. As it moves, O will rotate around a point, $\mathbf{y}(t)$, which for physically based motion is O's center of mass. All possible rotations are bounded by a sphere of radius r, the radius of O. The parabolic horn for O is the parabolic horn for $\mathbf{y}(t)$ with all its cross sections expanded by r. This simple approach to handling rotation can be overly conservative if the shape of O is distended. In this case, O's representation could be several overlapping parabolic horns, each bounding part of O.

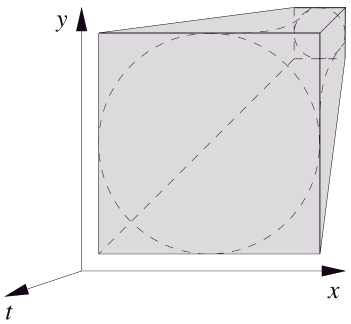

Figure 4: A hypertrapezoid for motion in $\mathbb{R}^2$.

As Section 5 will discuss, the detection algorithm addresses the fixed-timestep weakness by finding intersections between 4D bounding structures. Parabolic horns contain a quadratic factor t^2 that makes them costly to intersect. An alternative is to build a simple 4D polyhedron, called a *hypertrapezoid*, that encloses an object's parabolic horn. The cross sections of a hypertrapezoid at $t = 0$ and $t = 1$ are the isothetic[1] cubes that bound the $t = 0$ and $t = 1$ cross sections, respectively, of the parabolic horn. The cross sections of the hypertrapezoid at $t \in (0, 1)$ are defined by linear interpolation between the endpoint cubes. Fig. 4 shows a hypertrapezoid for an object in $\mathbb{R}^2$, in which case the hypertrapezoid is a 3D frustum. For an object in $\mathbb{R}^3$, the hypertrapezoid has six 4D faces, one for each 3D face of its cross-sectional cubes. (A hypertrapezoid needs no "bottom" or "top" face—corresponding to $t = 0$ and $t = 1$, respectively—because any intersection involving these faces implies an intersection involving the other six faces.) Each constant-t cross section of a 4D face is an isothetic 3D square. All such cross sections for a given 4D face are normal to the same axis of $\mathbb{R}^3$; we therefore say the 4D face is "normal" to that 3D axis, too. This feature of 4D faces will become important in Section 5. A hypertrapezoid is guaranteed to enclose the parabolic horn because the radius of the horn's cross sections is a convex function in t; thus a hypertrapezoid is also a conservative bound.

So far, the derivation has assumed that an object's acceleration vector could point in any direction. If the acceleration is limited to only certain directions, this approach is overly conservative, causing a hypertrapezoid to bound space that the object cannot possibly occupy. Knowledge that only certain directions are possible allows part of a hypertrapezoid to be trimmed away, tightening the bound it represents. Say the application knows that for $t \in [0, 1]$ the acceleration vector $\ddot{\mathbf{x}}(t)$ will lie in a hemisphere around O, that is, the application knows a vector $\mathbf{d} \in \mathbb{R}^3$ such that

$$\ddot{\mathbf{x}}(t) \cdot \mathbf{d} \le 0 \quad t \in [0, 1].$$

This sort of limit arises when O moves away from a particular direction, for example, if O is avoiding an obstructing wall. This relationship between $\ddot{\mathbf{x}}(t)$ and $\mathbf{d}$ leads to the inequality

$$(\mathbf{x}(t) - [\mathbf{x}(0) + \dot{\mathbf{x}}(0)t]) \cdot \mathbf{d} \le 0, \quad t \in [0, 1]. \tag{2}$$

The proof is almost identical to the proof of Inequality 1. To appreciate the significance of Inequality 2, think of $\mathbf{d}$ as the outward-pointing normal vector for a 3D plane that passes through $\mathbf{c}(t) = \mathbf{x}(0) + \dot{\mathbf{x}}(0)t$. Inequality 2 then states that at time t, the vector from O's position to $\mathbf{c}(t)$ will always be on the back side of that plane. The inequality still holds if O is not a point and $\mathbf{c}(t)$ is displaced by O's radius, r. Thus, O can never be in front of the plane, and any part of a bound on O's position that lies in front of the plane is unnecessary; Section 5 will describe how the algorithm implements this trimming. The set of 3D planes defined by $\mathbf{c}(t)$ and $\mathbf{d}$ for all $t \in [0, 1]$ are cross sections of a 4D *cutting plane*. A hypertrapezoid teamed with a cutting plane is a complete space-time bound. Fig. 5 shows an example.

The algorithms that use space-time bounds assume that M and $\mathbf{d}$ will be provided by the application program. The application must also provide the time at which these values *expire*, which corresponds to $t = 1$ in the earlier derivations. The application uses its specialized knowledge of its objects to compute these values. As a first example, consider a spaceship flight simulator. The simulator can compute M from the current acceleration provided by the ship's main engine; adding some extra "slack" to the value prevents M from immediately becoming invalid if the user increases the acceleration. For $\mathbf{d}$, the simulator can use the current direction of the ship's main axis; this value

[1] *Isothetic* means "axis aligned." For example, a 2D square is isothetic if each of its edges is parallel to a coordinate axis of $\mathbb{R}^2$, and a 3D cube is isothetic if each of its faces is an isothetic square.

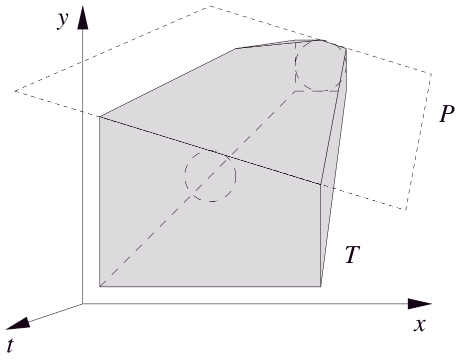

Figure 5: A complete space-time bound (hypertrapezoid T and cutting plane P) for motion in $\mathbb{R}^2$.

expires when the user makes the ship rotate more than 90 degrees, and the application can estimate this expiration time based on the current rotation rate. If the simulator notices that any of these guesses become incorrect, it can force the broad phase to treat the current time as the expiration time for M and $\mathbf{d}$. As a second example, consider an application with vehicles like airplanes or automobiles. As they turn, these vehicles have centripetal acceleration, directed towards the center of the turn. For a vehicle whose current speed is $|\dot{\mathbf{x}}|$, an upper bound on the centripetal acceleration is $|\dot{\mathbf{x}}|^2/R$, where R is lower bound on the turning radius. A corresponding $\mathbf{d}$ points towards the center of the turn. Again, the application can estimate the expiration time for these values and notify the broad phase if they expire early. As a final example, consider an application in which a user selects and "drags" objects with a mouse. This case is difficult because the dragging is kinematic instead of dynamic. One solution is slightly modify the dragging paradigm, simulating attachment of the object to the mouse cursor with a spring. This approach adds a little "lag," but it also smooths out the "jerkiness" in the object's motion, a result which is sometimes desirable.

5 Space-Time Bound Intersections

If two objects collide at simulation time t, their space-time bounds must intersect at some $t_i \leq t$; this conclusion follows because space-time bounds are conservative. A detection algorithm can thus compute the earliest t_i from all objects' space-time bounds, and do no further work from time 0 until time t_i. It should be intuitively clear that this approach addresses the fixed-timestep weakness. Section 6 will present the algorithm more precisely. The current section concentrates on how to compute t_i while avoiding the all-pairs weakness.

5.1 Overall Strategy

The main source of complexity in the intersection algorithm is the cutting planes. The following observation simplifies the algorithm: an intersection between two hypertrapezoid faces is necessary for any intersection between two space-time bounds.

To understand this observation, let B_1 be a space-time bound consisting of hypertrapezoid 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; the observation holds trivially.
- An intersection might involve a face, f_1, of T_1 and the cutting plane P_2. Recall that B_2 is the intersection of T_2 and the space behind P_2. Thus, only the part of P_2 inside T_2 matters. To get inside T_2, f_1 must intersect a face, f_2, of T_2.
- An intersection might involve P_1 and P_2. As in the previous case, only the parts of P_1 and P_2 within T_1 and T_2, respectively, matter. Thus, a face, f_1, from T_1 must intersect a face, f_2, from T_2.

There are no other ways B_1 can intersect B_2. It is convenient, therefore, to structure the search for intersections between space-time bounds around the search for face intersections.

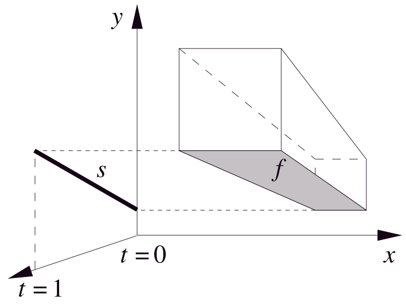

Figure 6: Each face $f \in F_\alpha$ projects into a straight line segment, s, in the α-t plane. Here, $\alpha = y$.

5.2 Face-Face Intersections

Recall that each 4D hypertrapezoid face is "normal" to one standard axis of $\mathbb{R}^3$, that is, each 3D cross section of the 4D face is normal to that axis. The set of all faces thus consists of three subsets:

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

These sets are important for the following reason: if two hypertrapezoids, T_1 and T_2, intersect for the first time at $t_i \in [0, 1]$, then there must be an intersection at t_i between a face of T_1 in F_α and a face of T_2 in the *same* F_α, for some $\alpha \in \{x, y, z\}$. The proof of this assertion is messy but straightforward.

The contra-positive of this assertion guides the 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 hypertrapezoids.

For problems like finding intersections within a set F_α, the literature contains no methods that guarantee good worst-case behavior. The next two sections present two heuristic methods. The following section compares the performance of these methods.

5.2.1 The Projection Method

One way to find intersections within F_α is to project 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 into the α-t plane is thus a 2D line segment, s, as Fig. 6 illustrates. Intersection between two faces' segments at $t = t'$ is a necessary but not sufficient condition for intersection between the faces at $t = t'$. The faces do actually intersect if their $t = t'$ cross sections intersect. Since the two cross sections are isothetic squares with the same α coordinate, checking for their intersection requires 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, the algorithm needs the β and γ coordinates of the squares' corners. Recall that each square, σ, is one side of a 3D isothetic cube, that cube being the $t = t'$ cross section of a hypertrapezoid, T. This σ shares an edge with four other sides of the cube. Since the cubes are isothetic, it should be clear that the algorithm can compute the required corner coordinates of σ given a point on each of these adjoining sides (four points total). Thus, the algorithm needs a formula for a point on a side of T's cross section at time t, that is, a point on a 4D face of T. Recall from Section 4 that a hypertrapezoid is defined by linear interpolation, so an appropriate point is

$$\mathbf{p}(t) = \check{\mathbf{p}} + (\hat{\mathbf{p}} - \check{\mathbf{p}})\, t, \quad t \in [\,0, 1\,], \tag{3}$$

where $\check{\mathbf{p}}$ and $\hat{\mathbf{p}}$ are points on the face at $t = 0$ and $t = 1$, respectively. To derive $\check{\mathbf{p}}$ and $\hat{\mathbf{p}}$, let T correspond to an object O with $\mathbf{x}(0)$, $\dot{\mathbf{x}}(0)$, r and M defined as in Section 4. Let the 4D face in question have 3D cross sections with outward-pointing normal $\mathbf{u}_\beta$, a unit vector pointing along the $+\beta$ axis. Equation 1 and the idea of expanding a parabolic horn by O's radius, r, yield

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

and

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

Similar analysis holds if the normal is $-\mathbf{u}_{\beta}$.

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 must consider whether the rectangle has been cut off by any cutting planes. The only relevant cutting planes 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' cross section of one or other cutting plane.

The algorithm applies these ideas as it finds intersections between all the 2D segments. It finds these segment intersections by using the technique of Bentley and Ottmann [21]. This technique sweeps a line across the α-t plane (from $t = 0$ to $t = 1$), reporting every intersection it finds. The broad phase needs the earliest (lowest t) segment intersection that corresponds to a real face intersection, so sweeping stops as soon it finds this intersection.

The Bentley-Ottmann algorithm runs in $\mathrm{O}([m + k]\log m)$ time for m segments that intersect k times. For processing F_{α}, $m = 2N$ because F_{α} contains two faces for each of the N objects. The worst-case value of k is $\mathrm{O}(N^2)$. One can imagine cases in which the algorithm processes $\mathrm{O}(N^2)$ segment intersections without finding any real face intersections. These cases should be unusual, though, because they require that the objects align themselves and coordinate their motions quite precisely. We have not encountered any of these cases in our tests. In fact, for the hundreds of test runs Section 9 will describe, 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 the projection method effectively eliminates the all-pairs weakness in practice.

5.2.2 The Subdivision Method

An alternative way to find face intersections is to use an approach similar to Duff's algorithm [6]. This approach subdivides space and time much like a 4D octree. The subdivisions are isothetic 4D cubes, so the algorithm can efficiently determine which faces intersect each cube. The algorithm subdivides recursively, the base case being cubes that intersect a small number, n_b, of faces. In each such cube, the algorithm tests all pairs of faces for intersection. The process continues until it finds the earliest intersection. Like the projection method, the subdivision method can spend $\mathrm{O}(N^2)$ time to notice that there are no face intersections. This worst case should be uncommon, however, and the subdivision method is usually efficient in practice.

5.2.3 Comparison

To compare the projection and subdivision methods, we ran empirical tests. The tests involved various distributions of hypertrapezoids, as detailed elsewhere [14]. The current section presents only the results for randomly distributed hypertrapezoids, which, in a sense, approximate the "average" case.

We conducted 200 tests, each involving a different set of 100 hypertrapezoids. Each test found the first intersection in the set by applying the projection method and the subdivision method with several values of n_b. For each value of n_b, the test measured the *speedup* of the projection method, defined as

$$\text{speedup} = \frac{\text{processing time for subdivision}}{\text{processing time for projection}}.$$

A speedup greater than 1 means that the projection method is faster. Fig. 7 graphs the speedup against the simulation time of the first intersection, for $n_b = 2$ and $n_b = 32$. The graphs use a logarithmic scale to avoid hiding "slowdowns" (speedups less than 1). The projection method is faster in all these tests, so we recommend it as the way to find face intersections.

5.3 Face-Plane Intersections

Section 5.1 mentioned that the second category of space-time-bound intersections involves a face, f_1, and a cutting plane, P_2. This intersection can occur only if f_1 intersects a face, f_2, from T_2. Only the part of P_2 inside T_2 is significant, so the intersection between P_2 and f_1 must involve greater t coordinates than the intersection between f_2 and f_1.

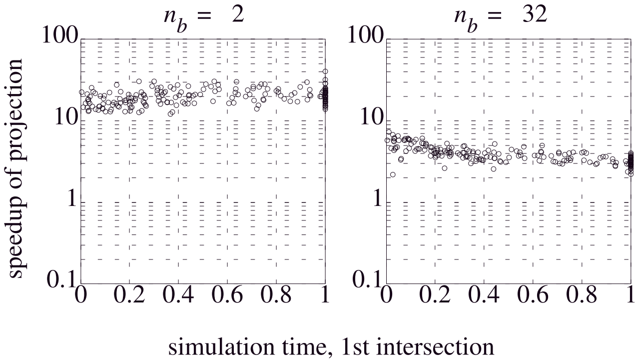

Figure 7: The speedup of the projection method over the subdivision method.

To characterize the intersection between f_1 and P_2, it is convenient to start with the intersection between P_2 and the 4D plane, p_1, that contains f_1. Let $\mathbf{n}_1 \in \mathbb{R}^3$ and $\mathbf{n}_2 \in \mathbb{R}^3$ be the normal vectors for the constant-t cross sections of p_1 and P_2, respectively. The line of intersection between p_1 and P_2 has direction

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

A reference point on the line at t, $\mathbf{r}(t)$, is defined by linear interpolation between points on the line at $t = 0$ and $t = 1$; those points must lie on both p_1 and P_2 so their positions follow from linear algebra. Given $\mathbf{r}(t)$, any point on the intersection between p_1 and P_2 is

$$\mathbf{i}(t,k) \;=\; \mathbf{r}(t) + k\mathbf{m}$$

for some $t \in [\,0, 1\,]$ and $k \in \mathbb{R}$.

Some points $\mathbf{i}(t,k)$ do not correspond to real intersections between f_1 and P_2. Such points are 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. The algorithm avoids these points by imposing constraints on $\mathbf{i}(t,k)$ that characterize real intersections. The constraints come from the interpretation of a space-time bound as the intersection between seven half-spaces: one defined by each face of the hypertrapezoid, and one for the cutting-plane.

A half-space defines a constraint expression as follows. Let $\mathbf{n}$ be the outward-pointing 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 (e.g., from Equation 3), then the point $\mathbf{i}(t,k)$ is inside that half-space only if

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

This inequality can be solved for k in terms of t because both $\mathbf{i}(t,k)$ and $\mathbf{p}(t)$ are linear functions of t; complete details appear elsewhere [14]. The result is a set of constraint half-planes in the t-k plane. The pairs, (t,k), within the intersection of these half-planes satisfy the constraints and correspond to real intersection points $\mathbf{i}(t,k)$. The face-plane intersection algorithm must find the earliest such point. Finding this point is an example of a two-variable linear-programming problem. Preparata and Shamos [21] describe a solution to this problem. An alternative solution involves straightforward extensions to the Bentley-Ottmann algorithm, and this approach is very efficient for the small number of constraints involved.

5.4 Plane-Plane Intersections

The final category of space-time-bound intersections from Section 5.1 involves two cutting-planes, P_1 and P_2, associated with hypertrapezoids T_1 and T_2. This situation is identical to the situation in Section 5.3, with P_1 replacing p_1. Note that the $\mathbf{i}(t,k)$ thus obtained is subject to twelve constraint half-spaces: six from T_1 and six from T_2.

The discussion of the ways space-time bounds can intersect is now complete. The pseudocode in Fig. 8 summarizes the whole process.

```
t_i ← ∞
for α ∈ {x, y, z}
   while ((intersection method says f_1-f_2 is next intersection in F_α, at t = t′) and
         (t′ < t_i))
         for j ∈ {1, 2}
            T_j ← hypertrapezoid containing f_j
            P_j ← cutting plane associated with T_j
         if (f_1-f_2 intersection is not cut off by P_1 or P_2)
            t_i ← t′
         else
           if ((f_1 intersects P_2, satisfying constraints from T_1, T_2 and P_1 at t = t″) or
               (f_2 intersects P_1, satisfying constraints from T_2, T_1 and P_2 at t = t‴))
              t_i ← min{t_i, t″, t‴}
           else
             if (P_1 intersects P_2, satisfying constraints from T_1 and T_2 at t = t⁗)
               t_i ← min{t_i, t⁗}
```

Figure 8: This algorithm finds the time, t_i, of the earliest intersection in a set of the space-time bounds. The "intersection method" can be either the projection method or the subdivision method.

```
t ← t_prev
while (t ≤ t_curr)
        t_build ← t
        rebuild space-time bounds as of t_build
        if (space-time bounds intersect at t_i)
           t ← unnormalize(t_i)
           if (t − t_build ≤ Δt_d)
              if (objects with intersecting space-time bounds
                  also have intersecting bounding spheres at t)
                 t_prev ← t
                 add these objects to result
                 return result to start narrow phase
              t ← t_build + Δt_d
        else
           t ← expiration time for space-time bounds
t_prev ← t
return no collisions
```

Figure 9: A broad phase that detects collisions as of simulation time t_{curr}.

6 Using Space-Time Bounds in the Broad Phase

This section describes how the detection algorithm's broad phase uses the results of Section 5. The essential idea appeared in Fig. 2 from Section 3, which shows the broad phase doing no work when collisions are not possible. If the earliest intersection between space-time bounds is at simulation time t_i, and $t_i > t_{\mathrm{curr}}$, t_{curr} being the time of the current frame, then the broad phase concludes no collisions are currently possible. Fig. 9 gives pseudocode that implements this idea.

In more detail, the broad phase sets the variable t_{build} to t_{prev}, the previous time at which it stopped detecting. It rebuilds the space-time bounds as of t_{build} and uses the intersection algorithm from Section 5 to find t_i, the next time at which the space-time bounds intersect. Note that Section 5 works in a *normalized* time scale, which assumes $t_{\mathrm{build}} = 0$ and $t_i \leq 1$; mapping between unnormalized and normalized time is straightforward, and the broad phase sets t to the unnormalized value of t_i.

If $t > t_{\mathrm{curr}}$, then no collisions are currently possible. The detection algorithm is thus free to return immediately.

Before returning, it stores t in the variable t_{prev} (which persists across frames), allowing it to immediately return at future frames until $t_{\text{curr}} \geq t_{\text{prev}}$. The broad phase can also return immediately if the space-time bounds built at t_{build} do not intersect at all. In this case, the algorithm from Section 5 sets t_{prev} to the simulation time at which the space-time bounds *expire*, that is, the future time at which the application's estimates of the acceleration bounds, M and $\mathbf{d}$, become invalid.

If $t \leq t_{\text{curr}}$, then a collision is possible as of the current frame and the broad phase must do more work. The default behavior in this case is to set t_{build} to t (the time of the possible collision), rebuild the space-time bounds and check for new intersections. When, however, $t - t_{\text{build}}$ is below a small threshold, Δt_d, then the default behavior has diminishing returns. The broad phase thus gets the bounding spheres of the two objects whose space-time bounds intersect, and checks these spheres for intersection. If these spheres intersect, then the broad phase has reached the limits of its accuracy. It returns the two objects so the narrow phase can take over to detect collisions with greater accuracy.

Once the narrow phase is complete, the broad phase resumes its operation. It must advance to a simulation time beyond t_{build} so it can rebuild and retest the space-time bounds. The rebuilding uses new values of M and $\mathbf{d}$ provided by the application; these new values reflect any discontinuous changes in objects' trajectories due to collision response. The simulation time to which the broad phase advances is $t_{\text{build}} + \Delta t_d$, with Δt_d as in the previous paragraph. Thus, Δt_d is the *minimum temporal resolution* of the broad phase. The value of Δt_d is chosen by the application. It should choose Δt_d small enough so that a user will not care if collisions of duration less than Δt_d are missed[2]. Note the similarity between Δt_d here and in the basic algorithm of Fig. 1. Both algorithms have the same minimum temporal resolution; the algorithm using space-time bounds has the advantage that its timestep is not fixed at Δt_d.

If the broad phase advances to $t_{\text{build}} + \Delta t_d$, objects may draw close enough to each other that their new space-time bounds are not disjoint at their initial time coordinate, $t_{\text{build}} + \Delta t_d$. The algorithm from Section 5, however, assumes that the space-time bounds are initially disjoint. To resolve this problem, the algorithm performs an extra test for overlap between the space-time bounds' cross sections as of time $t_{\text{build}} + \Delta t_d$. Each such cross section is defined by the space-time bound's two faces in any of the sets F_α, say for $\alpha = x$. These faces will already by sorted for the Bentley-Ottmann processing from Section 5.2.1, so the algorithm can iterate over them to find overlapping intervals. This iteration is quite efficient in practice.

7 The Narrow Phase and Sphere-Trees

This paper now turns to the narrow phase, which checks pairs of objects for intersection at a particular instant in simulation time. The narrow phase is based on the observation that spheres are among the simplest shapes to check for intersection. If all objects were spheres, the narrow phase's pair-processing algorithm would be trivial. Few interesting shapes are closely approximated by a single sphere, but a union of partially-overlapping spheres can provide a better approximation. To support progressive refinement, the narrow phase uses hierarchies of unions of spheres, which we call *sphere-trees*. Deeper levels in the hierarchy use more spheres to approximate the object more exactly. The children of a level-j sphere, s, are the spheres at level $j+1$ that cover the parts of the object that s covers. Thus, when the pair-processing algorithm checks two sphere-trees for intersection, it need check only pairs of spheres whose parents intersect. A simple recursive algorithm that descends the hierarchy can therefore find intersection at any level without wasting time on unnecessary sphere comparisons.

At each level of the recursion, the algorithm allows itself to be stopped by the application. By stopping at a particular hierarchy level, j, the algorithm detects approximate collisions. More specifically, the intersecting level-j spheres from two objects approximate the true collision between those objects (assuming that the recursion proceeds in a breadth-first manner, finding all level-j intersections before descending to level $j+1$). The accuracy of the approximate collision (and the ensuing collision response) depends on the tightness with which the level-j spheres fit the objects. Section 8 describes how to build sphere-trees in which deeper levels use more spheres to increase the tightness. Deeper descent thus increases the accuracy, and also increases the processing time. The narrow phase therefore provides progressive refinement, allowing the application to control the balance between accuracy and speed through its choice of the number of levels descended.

[2] Minimum temporal resolution is common in detection algorithms from the literature. Interval-analysis techniques [6, 27] provide the best alternative: at their deepest level of subdivision, they find small intervals of time that definitely contain collisions, although the exact instants of the collisions within those intervals are unknown.

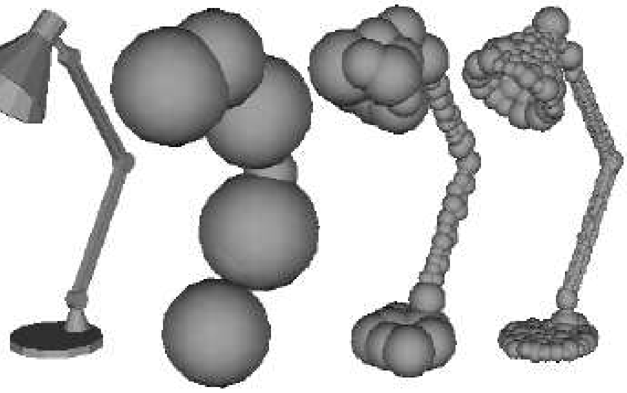

Figure 10: A desktop lamp (626 triangles) and three levels of its sphere-tree, built from the medial-axis surface.

Hierarchical algorithms are common in computer graphics, so it is important to note how this approach differs from the use of hierarchies in previous detection algorithms. First, the new approach avoids work inherent in algorithms that use octree hierarchies, for example, the algorithm of Shaffer and Herb [25]. At every frame, these algorithms incur the cost of rebuilding the octree or of moving objects between octants. Sphere-trees, in contrast, transform the same way as rigid objects, due to the rotational invariance of spheres; sphere-trees can thus be built once by a preprocess. This idea applies to articulated objects as well, if each articulated component has its own sphere-tree.

A second difference is that previous algorithms descend to the deepest nodes of their hierarchies and then check for exact intersections between the object's real surfaces (e.g., sets of polygons). The processing time for these algorithms thus grows with the geometric complexity of the surfaces. The new approach, on the other hand, can stop without processing the object's real surfaces. Its processing time thus depends on the resolution of the sphere-trees, which is independent of the objects' geometric complexity (when sphere-trees are built as Section 8 describes). The accuracy will decrease as geometric complexity increases, but the application will have the ability to balance speed and accuracy as it deems appropriate. The new approach can act like the previous ones (checking for real-surface intersections in the deepest intersecting spheres) if the application has time for that level of accuracy. This feature requires that each sphere-tree be *conservative*, in other words, that each level cover the object's surface completely.

8 Building Sphere-Trees

Building a sphere-tree requires generating multiple hierarchy levels, each using different numbers of spheres to approximate an object at different resolutions. Multiresolution modeling is a difficult topic in general, as Heckbert and Garland [12] discuss in the context of rendering. For collision detection, the particular challenge is making each level fit the object tightly; tightness is necessary because the application could stop the detection and invoke collision response on the colliding spheres at any level. This section gives an overview of techniques for building tight sphere-trees; more details appear elsewhere [14].

The literature contains few references on tight hierarchies. An algorithm that builds an octree representation of an object [23] can also build a sphere-tree, by circumscribing each occupied octant with a sphere. The resolution of the resulting sphere-tree increases with depth; even so, sphere-trees produced this way tend to fit an object loosely. Goldsmith and Salmon [11] present an algorithm that builds efficient bounding hierarchies for ray tracing. Optimizing a hierarchy for ray-object intersections does not directly promote the tightness needed for object-object collisions, however. O'Rourke and Badler [20] fit spheres inside a polyhedron by attaching them to a subset of its vertices. This approach produces tightly-fitting sets of spheres, but it offers little control over the resolution of a set, and it does not link multiple sets into a hierarchy.

We have experimented with two more elaborate techniques for building sphere-trees. One algorithm starts with an octree-based sphere-tree and tightens the spheres around the object. This tightening is a constrained minimization problem: minimize the looseness of the spheres while maintaining conservative coverage of the object. One way to solve such a problem is to apply *simulated annealing* [22]. In this context, simulated annealing repeatedly proposes random changes to the spheres, and accepts all changes that produce a tighter fit; it also accepts some changes that loosen the fit, to avoid local minima. Measuring the looseness of spheres around a nonconvex object is a nontrivial problem. The best approach we have found measures the Hausdorff distance [21] from each sphere to the object's surface, but there seems to be no way to measure this distance without making approximations. Simulated annealing does improve the accuracy of octree-based sphere-trees in many cases, but it is slow and it sometimes produces highly irregular distributions of spheres.

The most successful algorithm for building sphere-trees uses *medial-axis surfaces.* A medial-axis surface corresponds to the "skeleton" of an object. Building the exact medial-axis surface for a 3D polyhedron is difficult, but Goldak *et al.* [10] describe an algorithm that produces a satisfactory approximation. This algorithm distributes points on the outer surface of the object and computes the *Voronoi diagram* [21] for the points; the vertices of the Voronoi diagram that are inside the object lie on its medial-axis surface. Each such vertex defines a sphere that touches the object's outer surface at four points. Typically, there are many vertices and associated spheres, so generating multiple levels of detail requires "merging" adjacent spheres. To merge a pair of spheres, the sphere-tree-building algorithm replaces the pair with one sphere that covers the parts of the object the pair cover. The algorithm chooses which pair to merge by applying a cost function, one that encourages a tight fit around the object. By applying the merging operation repeatedly, the algorithm can give each level an arbitrary number of spheres; in our tests, level j has 8^j spheres, making the sphere-tree's fan-out match that of an octree. Note that the resolution of a sphere-tree is thus independent of the object's geometric complexity. The merged spheres do not necessarily cover the object conservatively, but they usually leave only tiny patches of the object's surface uncovered; a postprocess which scales spheres adjacent to uncovered patches corrects the problem.

To work reliably in this context, the Goldak *et al.* algorithm requires some extensions. The algorithm can have aliasing problems, and a postprocess must check the algorithm's results and correct any problems. The corrections involve selectively adding points on the object's outer surface and updating the Voronoi diagram. With these extensions, the algorithm to build a sphere-tree from a medial-axis surface works well in practice. Fig. 10 shows the results for a model of a lamp. This model features some long, flat triangles which require many small spheres for tight coverage. Even so, the sphere-tree allows efficient collision detection, as Section 9 reports. Long, flat triangles should also become less common as models increase in complexity and detail. Building the sphere-tree for the 626-triangle lamp takes 12.4 minutes on a Hewlett Packard 9000 Model 735. The increase in time is not unreasonable for larger models; for example, a model with more than 1400 triangles takes under 22 minutes. It is also important to remember that this time occurs in preprocessing, before an application begins running.

9 Performance

To evaluate the detection algorithm empirically, we tested it against one of the best previous algorithms. This algorithm is a hybrid, which uses Turk's uniform space subdivision [29] to find pairs of objects whose bounding cubes intersect, and then uses BSP trees [28] to process each pair. A complete description of the tests appears elsewhere [14]; this section presents highlights of the results.

The first set of tests compare space-time bounds to Turk's algorithm. The test program generates random configurations of isothetic cubes and applies forces to make them move. Each cube's forces are grouped into sets, each set being associated with acceleration bounds M and $\mathbf{d}$; the number of sets and the variations within each set are random. To simulate interactive conditions, the program gives neither algorithm any "future" information except for M, $\mathbf{d}$ and their expiration time. The Unix "clock" routine measures the time each algorithm needs to find the first collision.

These tests ran on a DECstation 5000/200 PGX Turbo. Fig. 11 gives the results for 200 tests with 100 objects per test. The graphs show the *speedup* of space-time bounds, defined as

$$\text{speedup} = \frac{\text{processing time for Turk's algorithm}}{\text{processing time for space-time bounds}}.$$

A speedup greater than 1 means space-time bounds are faster. The graphs use a logarithmic scale to avoid hiding "slowdowns" (speedups less than 1). The "detection-only" speedup is a comparison of only the time spent by the detection algorithms, ignoring the time spent by the ordinary differential equation (ODE) solver; the "overall" speedup includes the ODE solver's runtime.

Fig. 11 shows that space-time bounds are almost always faster than Turk's algorithm. Space-time bounds are sometimes slower when the terminating collision occurs almost immediately; in these cases, estimating future collisions

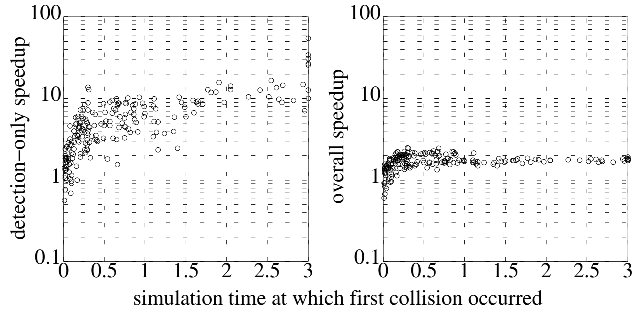

Figure 11: The speedup of space-time bounds over Turk's algorithm.

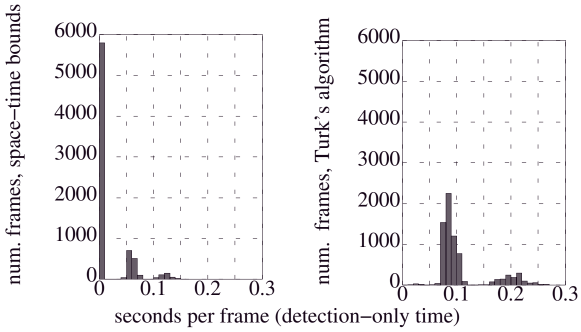

Figure 12: Frame timings for space-time bounds and Turk's algorithm.

provides no advantage so the overhead of space-time bounds is a penalty. The "overall" graph suggests that the ODE solver (a fourth-order Runge-Kutta method [22]) is a bottleneck; the "overall" speedups could be closer to the "detection-only" speedups given a faster ODE solver.

The speedups from Fig. 11 reflect performance over many frames. For interactive applications, it is also important that each individual frame be fast. To study per-frame performance, we repeated the tests, recording the time spent by each algorithm at each frame. In many of the individual tests, the slowest frame for space-time bounds is slower than the slowest frame for Turk's algorithm. On the other hand, frames nearly as slow as the slowest frame are less common with space-time bounds. Evidence for this conclusion comes from histograms of the "detection-only" frame times for each algorithm, which include all frames (not just the slowest) from all tests; see Fig. 12. Note, in particular, the fullness of the first bin, indicating that space-time bounds spend almost no time on the majority of the frames. By providing many fast frames with only occasional slower frames, space-time bounds meet the requirements of "soft" real-time systems like interactive graphics applications.

The next set of tests compare sphere-trees to BSP trees. BSP trees give exact results, so these tests indicate how much speed an application gains by accepting degraded accuracy. These tests run in the context of a sample application, a simple spaceship simulator; while not a real application, this simulator features objects with geometric and behavioral complexity that is representative of real applications. A user controls one ship by applying forward thrust and rotational velocity. The user's ship shares space with drone ships that periodically pick control values at random. The simulator can call two detection algorithms at each frame to compare their performance. For these tests, one algorithm uses sphere-trees and the other uses BSP trees. A preprocessor builds both data structures before the start of the tests. To make the BSP trees more efficient, the preprocessor minimizes the number of extra nodes created by faces that straddle partitioning planes; Naylor [19] discusses other ways to optimize BSP trees. The "ships" in these tests are shaped like lamps, as in Fig. 10. The sphere-trees have a fourth, more accurate level of spheres not shown in Fig. 10, and these spheres store the lamp's polygons, allowing a fifth level of exact detection. All tests ran on a Sun Sparcstation 10/512 ZX, with timings by the Unix "gethrtime" routine.

The tests measure the *speedup* of sphere-trees over BSP trees, defined as

$$\text{speedup} = \frac{\text{processing time for BSP trees}}{\text{processing time for sphere-trees}}.$$

level	all cases number	mean speedup
1	50042	1049.3
2	5079	581.6
3	2180	243.2
4	915	112.8
5 (exact)	483	2.7

Figure 13: The speedup of sphere-trees over BSP trees, for all cases that reached sphere-tree level j.

level	intersecting cases number	mean speedup
1	5079	1576.7
2	2180	607.1
3	915	181.3
4	483	83.1
5 (exact)	143	1.3

Figure 14: The speedup of sphere-trees over BSP trees, for cases that intersect at sphere-tree level j.

Fig. 13 shows the mean speedup for all cases in which the narrow phase reached level j of the sphere-trees. In some of these cases, level j is the shallowest level at which the sphere-trees do not intersect; the detection algorithm can thus conclude that the objects do not collide. In other cases, the sphere-trees still intersect at level j. If the application stops the narrow phase when there is still intersection (at levels shallower than 5, exact detection) then it invokes collision response on objects that might not quite collide. Fig. 14 shows the speedup in these cases. Fig. 15 shows an upper bound on the distance between the not-quite-colliding objects in these cases. These distances indicate a certain amount of inaccuracy, but it decreases rapidly with sphere-tree depth. When this modest inaccuracy is acceptable, Figs. 13 and 14 show that sphere-trees offer dramatic speedups over BSP trees.

The final set of tests compare the overall algorithms, space-time bounds and sphere-trees versus Turk's algorithm and BSP trees. These tests also use the spaceship simulator, again with "ships" shaped like lamps. The ten drone ships start in a circle around a user-controlled ship, but the drones soon break formation and move independently. Collision response is simple, causing a ship to jump back to its initial position after each collision. Fig. 16 tracks the processing time spent by Turk's algorithm and BSP trees for one run of the simulator. Notice that the time oscillates considerably, and it is almost always greater than 5 seconds per frame. Fig. 17 shows a similar run for space-time bounds and sphere-trees. For this run, the simulator tries to meet a target of 0.1 seconds per frame (acceptable "interactive" performance) by interrupting the narrow phase. Fig. 17 tracks the time devoted to collision detection[3] and to all other "nondetection" activities (e.g., rendering); it also tracks "slack" time, periods of waiting that keep the frame time from being less than 0.1 seconds. The detection algorithm allows the application to keep the overall frame time on target for almost all frames. The speedup over Turk's algorithm and BSP trees is more than 200 at some frames. At all frames, space-time bounds and sphere-trees allow interactive performance which is impossible with the other algorithm.

[3] We did not time the broad and narrow phases separately because the extra calls to the system clock take time themselves, enough time to have noticeable effects on the overall frame rate in our pilot studies.

level	mean distance
1	0.85
2	0.39
3	0.15
4	0.06

Figure 15: For cases that intersect at level j, an upper bound on the separation distance, expressed as a fraction of the lamp's bounding-sphere radius.

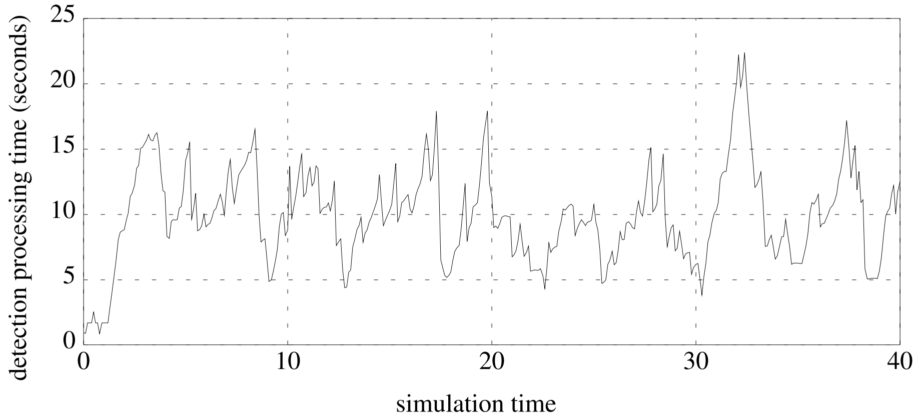

Figure 16: The processing time for Turk's algorithm and BSP trees, for one run of the spaceship simulator.

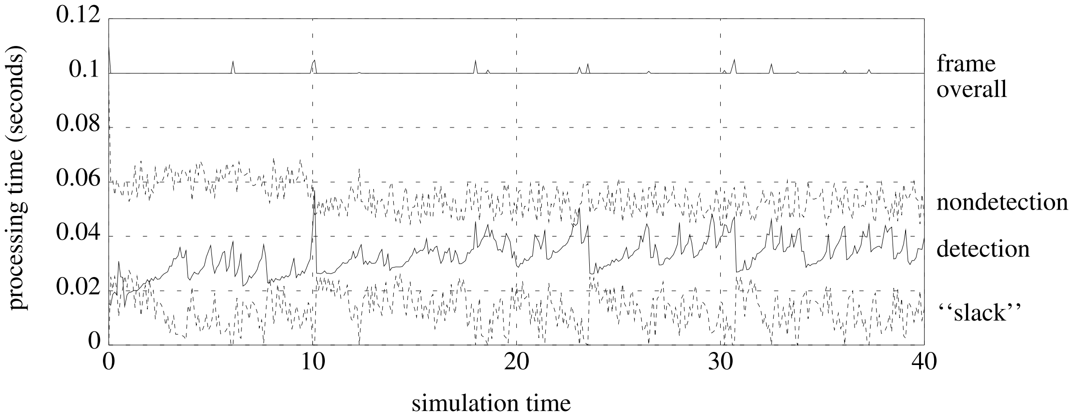

Figure 17: The processing time for space-time bounds and sphere-trees, for one run of the spaceship simulator.

10 Conclusions

This paper presents a time-critical approach to collision detection. The algorithm addresses three weaknesses common in detection algorithms, and allows interactive applications to trade accuracy for speed as needed. An implementation significantly outperforms a good previous algorithm in empirical tests, allowing interactive performance that would otherwise be impossible.

This work offers many opportunities for extensions and improvements. The narrow phase currently does not exploit interframe coherence; starting traversals of sphere-trees at levels suggested by the previous frame could make the narrow phase even faster than it is now. Space-time bounds based on maximum acceleration work well, but 4D structures similar to space-time bounds can also be derived from limits on velocity or position; this reformulation might be advantageous in some cases, for applications that can estimate velocity or position more easily than acceleration. The current broad phase is not interruptible because it is conservative, that is, it never ignores bounding-box collisions. It could thus take an inordinate amount of time in some situations, such as when the objects are packed densely enough that every object collides with multiple objects at every frame. One way to make the broad phase interruptible is to allow it to selectively ignore objects that the application temporarily designated "less important"; an interesting question is whether speed gained in this manner would be useful. Parallel processing could accelerate both phases of the algorithm: in the broad phase, the predictive properties of space-time bounds could assist load-balancing schemes, and in the narrow phase, the independence of paths to sphere-tree leaves could allow simultaneous processing. Finally, the general idea of trading accuracy for speed deserves further research. Important advances would be systems that integrate time-critical algorithms for multiple tasks, and studies to determine what approximations users find most appropriate for particular tasks.

11 Acknowledgements

A preliminary, abbreviated version of this paper appeared at the IEEE Symposium on Research Frontiers in Virtual Reality [13]. This research was conducted as part of the author's doctoral studies at Brown University. The guidance of John "Spike" Hughes was essential to the completion of this work, and Andy van Dam and the rest of the Brown Computer Graphics Group also made important contributions. At Brown, this work was supported in part by: the NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization; Sun Microsystems; Autodesk; Taco, Inc.; ONR grant N00014-91-J-4052 ARPA order 8225; NCR; Hewlett Packard; IBM; Digital Equipment Corporation; Apple; and Microsoft. At Cornell, Peter Shirley gave valuable encouragement and advice on the structure of this paper.

References

[1] D. Baraff. Coping with friction for non-penetrating rigid body simulation. In *Proceedings of SIGGRAPH '91,* published as *Computer Graphics*, 25(4):31–40, July 1991.

[2] L. Bergman, H. Fuchs, E. Grant, and S. Spach. Image rendering by adaptive refinement. In *Proceedings of SIGGRAPH '86,* published as *Computer Graphics*, 20(4):29–37, Aug. 1986.

[3] J. Canny. Collision detection for moving polyhedra. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(2):200–209, Mar. 1986.

[4] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics (Monterey, California)*, pages 189–196, 1995.

[5] R. K. Culley and K. G. Kempf. A collision detection algorithm based on velocity and distance bounds. In *Proceedings 1986 IEEE International Conference on Robotics and Automation*, pages 1064–1069, 1986.

[6] T. Duff. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. In *Proceedings of SIGGRAPH '92,* published as *Computer Graphics*, 26(2):131–138, July 1992.

[7] A. Foisy, V. Hayward, and S. Aubry. The use of awareness in collision prediction. In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation*, pages 338–343, 1990.

[8] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of SIGGRAPH '93,* published as *Computer Graphics* Proceedings, Annual Conference Series, pages 247–254, Aug. 1993.

[9] A. Garcia-Alonso, N. Serrano, and J. Flaquer. Solving the collision detection problem. *IEEE Computer Graphics and Applications*, 14(3):36–43, May 1995.

[10] J. A. Goldak, X. Yu, A. Knight, and L. Dong. Constructing discrete medial axis of 3-D objects. *International Journal of Computational Geometry and Applications*, 1(3):327–339, 1991.

[11] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.

[12] P. S. Heckbert and M. Garland. Multiresolution modeling for fast rendering. In *Proceedings of Graphics Interface '94*, pages 43–50, May 1994.

[13] P. M. Hubbard. Interactive collision detection. In *Proceedings of the 1993 IEEE Symposium on Research Frontiers in Virtual Reality*, pages 24–31, Oct. 1993.

[14] P. M. Hubbard. *Collision Detection for Interactive Graphics Applications.* PhD thesis, Department of Computer Science, Brown University, Oct. 1994.

[15] R. S. Kennedy, N. E. Lane, M. G. Lilienthal, K. S. Berbaum, and L. J. Hettinger. Profile analysis of simulator sickness symptoms: Application to virtual environment systems. *Presence*, 1(3), Summer 1992.

[16] M. C. Lin, D. Manocha, and J. F. Canny. Fast collision detection between geometric models. Technical Report TR93-004, Department of Computer Science, The University of North Carolina at Chapel Hill, Jan. 1993.

[17] P. W. C. Maciel and P. Shirley. Visual navigation of large environments using textured clusters. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics (Monterey, California)*, pages 95–102, 1995.

[18] M. P. Moore and J. Wilhelms. Collision detection and response for computer animation. In *Proceedings of SIGGRAPH '88,* published as *Computer Graphics*, 22(4):289–298, Aug. 1988.

[19] B. F. Naylor. Constructing good partitioning trees. In *Proceedings of Graphics Interface '93*, pages 181–191, May 1993.

[20] J. O'Rourke and N. Badler. Decomposition of three-dimensional objects into spheres. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(3):295–305, July 1979.

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

[22] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, Cambridge, England, 2nd edition, 1992.

[23] H. Samet and R. E. Webber. Hierarchical data structures and algorithms for computer graphics, part I: Fundamentals. *IEEE Computer Graphics and Applications*, 8(3):48–68, May 1988.

[24] S. Sclaroff and A. Pentland. Generalized implicit functions for computer graphics. In *Proceedings of SIGGRAPH '91,* published as *Computer Graphics*, 25(4):247–250, Aug. 1991.

[25] C. A. Shaffer and G. M. Herb. A real-time robot arm collision avoidance system. *IEEE Transactions on Robotics and Automation*, 8(2):149–160, Apr. 1992.

[26] A. Smith, Y. Kitamura, H. Takemura, and F. Kishino. A simple and efficient method for accurate collision detection among deformable objects in arbitrary motion. In *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pages 136–145, Mar. 1995.

[27] J. M. Snyder, A. R. Woodbury, K. Fleischer, B. Currin, and A. H. Barr. Interval methods for multi-point collisions between time-dependent curved surfaces. In *Proceedings of SIGGRAPH '93,* published as *Computer Graphics* Proceedings, Annual Conference Series, pages 321–334, Aug. 1993.

[28] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. In *Proceedings of SIGGRAPH '87,* published as *Computer Graphics*, 21(4):153–162, July 1987.

[29] G. Turk. Interactive collision detection for molecular graphics. Technical Report TR90-014, Department of Computer Science, The University of North Carolina at Chapel Hill, Jan. 1990.