ACM Transactions on Graphics, Vol. 15, No. 3, July 1996, pp. 179–210

Approximating Polyhedra with Spheres for Time-Critical Collision Detection *

Philip M. Hubbard
Program of Computer Graphics
Cornell University†

July 26, 1996

Abstract

This paper presents a method for approximating polyhedral objects to support a *time-critical* collision-detection algorithm. The approximations are hierarchies of spheres, and they allow the time-critical algorithm to progressively refine the accuracy of its detection, stopping as needed to maintain the real-time performance essential for interactive applications. The key to this approach is a preprocess that automatically builds tightly fitting hierarchies for rigid and articulated objects. The preprocess uses *medial-axis surfaces*, which are skeletal representations of objects. These skeletons guide an optimization technique that gives the hierarchies accuracy properties appropriate for collision detection. In a sample application, hierarchies built this way allow the time-critical collision-detection algorithm to have acceptable accuracy, improving significantly on that possible with hierarchies built by previous techniques. The performance of the time-critical algorithm in this application is consistently 10 to 100 times better than a previous collision-detection algorithm, maintaining low latency and a nearly-constant frame rate of 10 frames per second on a conventional graphics workstation. The time-critical algorithm maintains its real-time performance as objects become more complicated, even as they exceed previously reported complexity levels by a factor of more than 10.

Categories and Subject Descriptors: I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*geometric algorithms, languages and systems; object hierarchies; physically based modeling*; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*animation; virtual reality.*

General Terms: algorithms, human factors, performance.

Additional Key Words and Phrases: approximation, collision detection, interactive systems, medial-axis surfaces, spheres, time-critical computing.

1 Introduction

Performance is paramount for most interactive graphics applications, such as virtual reality systems and vehicle simulators. As Brooks [6] discusses for the case of virtual reality, these applications will

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

not be successful unless they respond to users' actions at real-time rates. In particular, frame rates must be high and nearly constant, and latency ("lag") must be low.

Realistic modeling, rendering and animation are also important in these applications. Collision detection and response, for example, prevent moving objects from passing through each other, making the objects seem more natural and believable. Despite recent advances, traditional collision-detection algorithms ("detection algorithms") are not fast enough for most interactive settings. Most interactive applications are thus forced to do without collision detection.

The most promising way to make collision detection possible for more interactive applications is to use *time-critical computing*. The essence of this approach, which van Dam [49] also calls *negotiated graceful degradation*, is trading accuracy for speed. A time-critical detection algorithm checks for collisions between successively tighter approximations to the objects' real surfaces. After any step of this progressive refinement, the application can stop the algorithm if it exceeds its time budget.

By processing approximations to objects' surfaces, a time-critical algorithm can maintain consistent, real-time performance as the surfaces become more complicated. This approach does degrade the accuracy of detection, but small inaccuracies will be acceptable in many situations. Most other detection algorithms produce their own form of inaccuracy, because these algorithms sample time discretely. These algorithms do not, however, adjust their accuracy to ensure real-time performance, so they do not provide the advantages of a time-critical algorithm. Section 2 elaborates on these points.

The key to time-critical collision detection is the method for automatically approximating an object's surface. This paper presents a preprocess that approximates rigid or articulated polyhedral objects with sets of spheres. It produces multiple *levels of detail* arranged in a hierarchy, as Figure 1 exemplifies. To place the spheres, the preprocess uses the *medial-axis surface*, which represents an object in skeletal form. The medial-axis surface guides an optimization process that matches the spheres to the object's shape. During a run of an application, these levels of detail take the place of the object's real surface for approximate but fast collision detection and response. Note the difference between this approach and the traditional use of hierarchies to focus on collisions between the real surfaces.

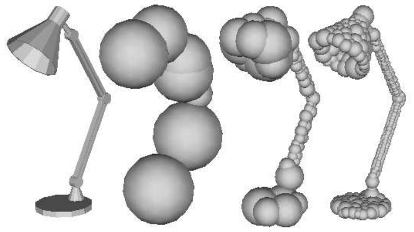


Figure 1: A desktop lamp (626 triangles), and three levels of detail using spheres.

The time-critical detection algorithm is kindred in spirit to recent time-critical algorithms for other graphics problems; examples include rendering algorithms for static walk-throughs by Funkhouser and Séquin [13] and Maciel and Shirley [29], the IRIS Performer application framework of Rolhf and Helman [40] and the human-figure animation algorithm of Granieri *et al.* [17]. This paper extends our earlier paper [20], which introduced the idea of time-critical collision detection but presented a less sophisticated approach to hierarchies. Our companion papers [23, 22]—to our knowledge, the only other descriptions of time-critical collision detection—complement the current paper, focusing more on the framework for the time-critical algorithm and less on important details of building and using the sphere hierarchies. The novel contributions of the current paper are as follows:

- The preprocess that builds hierarchies of spheres specifically addresses the goal of maximizing collision accuracy at each level of detail for a fixed number of spheres. No previous work on hierarchies addresses this goal. Empirical evidence suggests that the improvement in accuracy over previous work is significant.

- Empirical tests demonstrate that the time-critical collision-detection algorithm provides acceptable accuracy while keeping latency low and the frame rate high and nearly constant. No previous work in collision detection demonstrates real-time performance for so many nearly simultaneous collisions between objects with such complicated, highly non-convex shapes. Specifically, the tests show a significant speedup over a previous algorithm based on BSP trees, and an ability to maintain real-time performance as objects become more complex.

We do not suggest that time-critical collision detection is appropriate in every situation. The maximum possible accuracy will be needed for some applications, such as a simulation that verifies the precise fit of parts in a mechanical assembly, or one that predicts behavior dependent on sustained contact between parts. Nevertheless, the time-critical algorithm should increase the number of applications which can use collision detection.

The remainder of this paper proceeds as follows. Section 2 motivates the idea of trading accuracy for speed. Section 3 surveys the benefits of previous work on collision detection, and explains why this work does not make trading accuracy for speed unnecessary. Section 4 presents the structure of the time-critical detection algorithm. Sections 5 through 9 describe how to build the hierarchies of spheres that approximate objects at multiple levels of detail, and Section 10 shows some examples of these hierarchies. Section 11 presents results from empirical tests of the detection algorithm. These results indicate that the algorithm can balance performance and accuracy effectively. Section 12 summarizes the paper and discusses possible extensions.

2 The Need for Time-Critical Collision Detection

The state of the art in collision detection is improving. Some detection algorithms can provide real-time performance in some challenging test cases, as Section 3 describes. The demands of interactive applications are also increasing, however, and there are several arguments that only a time-critical detection algorithm can meet these demands.

A detection algorithm must maintain real-time performance as an application's geometric characteristics change. Some changes occur as an application runs: users perform actions that bring multiple objects together into colliding and nearly-colliding configurations, increasing the geometric complexity in regions between objects. Other changes occur between runs: designers upgrade the application by adding more objects with more detail, increasing the geometric complexity of future runs. A time-critical algorithm copes with these changes by approximating object surfaces. For approximations with resolution independent of the real surfaces, performance does not degrade as geometric complexity increases. Traditional (non-time-critical) detection algorithms, on the other hand, process the real surfaces of objects, so their processing times must necessarily increase with geometric complexity. Such time growth is unacceptable for interactive applications; increases in the complexity of the objects do not make users care less about speed and responsiveness.

Because it uses approximation, a time-critical detection algorithm does decrease accuracy. The simplest measure of a detection algorithm's inaccuracy is the separation distance between two objects it considers to be colliding. Accuracy is also affected by what the algorithm reports as the relative orientation of the colliding surfaces. Collision response when the separation distance is not exactly zero or the orientation is incorrect does alter the course of future frames. This cumulative inaccuracy will sometimes cause intolerable qualitative changes, but there are reasons to believe that it is often acceptable. With our time-critical algorithm, the inaccuracy at the collision itself is usually small, as Section 11 demonstrates. For colliding objects which are steered by users, the cumulative inaccuracy will quickly disappear in many cases, because humans are skilled at correcting subtle changes unconsciously (as when riding a bicycle). The effects on unsteered objects may not be noticeable to users, as the correct behavior after a collision between complicated shapes may be difficult to predict (consider, for example, a trumpet hitting a trombone). Even if inaccuracies are noticeable, they may be better than the alternative, degraded performance with improved accuracy. Hettinger and Riccio [18] report that users of vehicle simulators seem to suffer motion sickness more frequently when latency is high. When poor performance does not cause "simulator sickness," it can render interactive applications unresponsive and thus ineffective; Pausch *et al.* [35], for example, cite studies indicating that latency decreases operator performance in vehicle simulators.

Full accuracy is also uncommon in the alternatives to the time-critical algorithm. Most traditional algorithms produce temporal inaccuracy by sampling object positions discretely; although in theory the application can choose the sampling rate, even the fastest of these algorithms can support real-time performance only when limited to one sample per frame. Temporal inaccuracy thus creates spatial inaccuracy equal to the distance an object travels between frames. This effect is present even for slow-moving objects. A human figure walking slowly at 2.5 miles per hour, for example, travels 4.4 inches per frame, given 10 frames per second; inaccuracy of 4.4 inches is significant on the scale of the figure. Adaptive sampling could reduce temporal inaccuracy, but current adaptive techniques generally cannot increase the sampling rate without sacrificing real-time performance. In the presence of temporal inaccuracy, the time a traditional algorithm spends checking for exact surface collisions at each time sample may be wasted. A better approach is to seek only as much collision accuracy as real-time constraints allow, which is the approach taken by the time-critical algorithm.

Every application involves a variety of tasks other than collision detection. The application itself, or the people using it, should control how processing time is distributed among the tasks to balance the overall speed and effectiveness of the application. A detection algorithm should not have the power to delay the rest of the application while it produces the accuracy it alone deems important. The time-critical algorithm produces more or less accuracy to fit the time it is given, so it gives an application flexibility to handle competing demands.

3 Related Work

The literature on collision detection is extensive. The published algorithms incorporate many important techniques that improve performance. This section argues, however, that these algorithms cannot meet the goals from the previous section: maintaining the real-time performance required by sophisticated interactive applications.

To put the previous work in perspective, it helps to consider an idealized interactive application that calls a simple detection algorithm, as in Figure 2. This application generates frames on a *simulation time* scale, which may not correspond to the *wall-clock time* we experience. The detection algorithm is accurate to only Δt_d simulation time units, its *minimum temporal resolution*.

The simple detection algorithm has several weaknesses, as we describe in more detail elsewhere [20, 22]. The fixed time step on line 9 can cause inaccuracy or inefficiency, and the all-pairs loop on line 10 can reduce performance. The intersection test on line 11 can also be a significant performance bottleneck.

```
1  application()
2      for $t \leftarrow t_0$ to $t_1$ in steps of $\Delta t_r$
3          get user input
4          update each object's behavior as of $t$
5          while (collision_detection($t$, $collisions$))
6              respond to $collisions$
7          render each object

8  collision_detection($t_{\mathsf{curr}}$, $collisions$)
9      for $t_d \leftarrow t_{\mathsf{prev}}$ to $t_{\mathsf{curr}}$ in steps of $\Delta t_d$
10         for each pair of objects $(O_1, O_2)$
11             if ($O_1$ intersects $O_2$ as of $t_d$)
12                 add $(O_1, O_2)$ to $collisions$
13         if ($collisions$)
14             $t_{\mathsf{prev}} \leftarrow t_d$; return TRUE
15     $t_{\mathsf{prev}} \leftarrow t_d$; return FALSE
```

Figure 2: An idealized interactive application and a simple detection algorithm.

Several algorithms address these weaknesses by using geometry with an extra dimension that explicitly represents simulation time. Samet and Tamminen [41] apply recursive subdivision to the four dimensions of space and time. Canny [8] derives quintic polynomials whose roots represent the time and location of collisions. Cameron [7] extends the approach of Samet and Tamminen, adding a mechanism that prunes parts of objects that cannot collide. Von Herzen *et al.* [50] use Lipshitz conditions to accelerate a form of binary search through space and time. Duff [11] applies interval analysis to generalize the idea of recursive subdivision. Snyder *et al.* [45] combine interval analysis with the Newton-Raphson root-finding technique, providing the most accurate detection to date for collisions involving curved surfaces. All of these techniques assume knowledge of every object's exact position throughout simulation time. This information is available to applications that generate prescripted animations off-line, and these algorithms work well in that setting. Interactive applications, however, feature objects whose motion is specified "on the fly" by human users, so these algorithms cannot be used directly. To work in this context, these algorithms would need to predict the future positions of objects; we discuss related ideas later in this section.

Other algorithms do improve on the basic algorithm without making assumptions about objects' motions. Moore and Wilhelms [32] and Shaffer and Herb [43] use recursive subdivision in the form of an octree. By repeatedly subdividing regions of space that contain more than one object, an octree helps these algorithms avoid testing distant parts of objects for collisions. These algorithms must update the octree when objects move, however, and the associated computation can be significant.

Most other algorithms appropriate for interactive applications view collision detection as two phases. Replacing lines 9 and 10 in Figure 2 is the *broad phase*, which finds collisions between simplified forms of the objects, such as bounding boxes or spheres. Line 11 is the *narrow phase*, and it checks for exact intersections between individual pairs of objects whose simplified forms collide at time t_d.

For the broad phase, Turk [47] and Zyda *et al.* [52] use a regular grid to identify objects that are close to each other. Baraff [3] and Cohen *et al.* [9] describe *sweep-and-prune* techniques which exploit inter-frame coherence to efficiently sort bounding boxes, identifying those that intersect. Several authors describe ways to adaptively change the broad phase's time step using predictions of objects' future positions, predictions which often are possible for interactively guided objects. Mirtich and Canny [31] use upper bounds on linear and angular velocity in a priority queue that tracks the next possible collision between convex polyhedra. We use bounds on maximum acceleration to derive *space-time bounds* [20, 22], four-dimensional structures whose intersections predict bounding-sphere collisions. Foisy *et al.* [12] also use maximum accelerations to predict collisions, employing a queuing scheme to efficiently update the predictions. In our experience, adaptive techniques must clamp the time step at a minimum temporal resolution, Δt_d, to avoid slowing below real-time rates, so these techniques do not eliminate the temporal inaccuracy mentioned in Section 2. Several of these broad-phase algorithms perform well in empirical tests, so we use space-time bounds in our time-critical algorithm and we concentrate on the narrow phase in the remainder of this paper.

An early narrow-phase algorithm of theoretical importance is described by Dobkin and Kirkpatrick [10]. This algorithm detects the collision of two polyhedra in $O(\log^2 n)$ time, where n is the total number of vertices in the polyhedra. A practical disadvantage of this algorithm, however, is that it returns insufficient information for many forms of collision response (it reports only one collision point even if multiple parts of the objects collide). Baraff [2] presents an algorithm that exploits inter-frame coherence to efficiently detect collisions between pairs of convex objects. Non-convex objects must be treated as a union of convex pieces, which will cause inefficiency for objects with complicated shapes. Sclaroff and Pentland [42] improve the narrow phase's performance by approximating each object with a new representation, a deformed superquadric ellipsoid whose surface is modulated by a displacement map. This approach works well but applies to only some types of objects, those with spherical topology and "star-shaped" [37] surface features. Recursive subdivision allows an algorithm to process objects with more general shapes. One of the earliest examples of this idea is the work of Mäntylä and Tamminen [30]. Kitamura *et al.* [26] present a more modern variation that uses octrees.

Three recent algorithms with narrow phases that use subdivision techniques—the work of Smith *et al.* [44], Garcia-Alonso *et al.* [14] and Ponamgi *et al.* [36]—deserve particular attention. Although their broad phases create the temporal inaccuracy discussed in Section 2, these algorithms have the advantage that they achieve real-time performance for some challenging situations. In the Smith *et al.* algorithm, the narrow phase builds an octree for a subset of the faces of polygonal objects, those faces within intersecting bounding boxes found by the broad phase. In a sample run that finds the first collision among 15 space shuttles (528 faces each), the algorithm performs well, taking about 0.03 seconds at the slowest time step (which occurs at the first collision). Few real applications end at the first collision, however. It is unclear how the algorithm would perform after collision response, which can cause multiple objects to collect in colliding and nearly-colliding configurations for many time steps.

The Garcia-Alonso *et al.* [14] algorithm precomputes a one-level grid subdivision for each object. The narrow phase uses these grids to search for intersecting faces within the overlap of object bounding boxes. When detecting interference in an unfolding satellite antenna (with 50 jointed objects and 1500 total faces), the algorithm allows near-real-time animation at 5 frames per second. The generality of these results is unclear, though, for several reasons. The satellite animation stops at the first collision, which can affect performance as noted above. For objects connected by joints, the narrow phase uses an inexpensive test of joint limits, and the number of satellite components handled this way is unspecified. For objects not handled this way, the narrow phase must deal with rotations to the objects and thus to their precomputed grids. Its solution is to replace a rotated grid cell with its axis-aligned bounding box. Unfortunately, such a box can bound considerably more than the original cell: when we generated 500,000 random rotations of a unit cube, we found that its bounding box increased in volume by a factor of 3.4 on the average and 4.7 in the worst case. Thus, rotations reduce the grid's expected efficiency for localizing collisions.

Ponamgi *et al.* [36] use a narrow phase that builds on two previous approaches. It first uses the incremental algorithm of Lin and Canny [27] to check the convex hulls of the two objects for intersection. An intersection here may involve part of a hull that covers a concavity. In this case, the narrow phase descends a precomputed octree subdivision of the concavity, using a hierarchical version of the Cohen *et al.* sweep-and-prune technique [9] to exploit inter-frame coherence. For a sample run in which eight interlocked tori (400 faces each) bounce against each other, this approach gives real-time performance, taking 0.038 seconds per frame. The single, highly regular hole of a torus makes the octree approach particularly effective. Multiple irregular holes will pose more of a challenge, however, especially those involving skinny faces; the precomputed octree leaves must bound these faces in all possible orientations, so the octrees will be looser and less effective at localizing collisions. Half the faces of a torus lie on its convex hull, and the interlocked configuration of the tori could cause nearly half the collisions to involve only these hull faces; the algorithm is optimized to detect these collisions very quickly, but how it would perform for more general situations is unclear.

The best evaluation of a new detection algorithm involves running it and previous algorithms in the same application. Section 11 makes a first step towards this goal by describing a comparison between the time-critical algorithm and one previous algorithm, an algorithm based on binary space partitioning (BSP) trees as described by Thibault and Naylor [46]. Another interesting comparison would involve one of the subdivision algorithms, for example, the work of Ponamgi *et al* [36]. This algorithm and the BSP algorithm both traverse trees, and the subdivision algorithm has the advantage that it can sometimes detect the absence of collisions before reaching the leaves. On the other hand, the BSP algorithm makes better use of preprocessing. A precomputed BSP tree transforms naturally to match a moving rigid object, which is not true of a precomputed octree or axis-aligned grid. The preprocess that builds the BSP tree can also optimize the tree's collision-localizing properties, as Naylor describes [33]. We are aware of no reports that subdivision algorithms outperform the BSP algorithm, but further comparisons between the time-critical algorithm and other approaches like the subdivision algorithms would be interesting future work.

4 A Time-Critical Detection Algorithm

The majority of the algorithms from the previous section process the real surfaces of objects, so they do more work as the surfaces become more complicated. This paper now turns to how a time-critical detection algorithm avoids this problem.

4.1 Progressive Refinement

The time-critical algorithm assumes each object is approximated by a hierarchy of spheres, which represents the object at multiple levels of detail. Level 0 is the object's bounding sphere. Subsequent levels are unions of successively more spheres, approximating the object at higher resolutions. Spheres are rotationally invariant, so for a rigid object, the hierarchy is built once by a preprocess; a running application applies to this hierarchy the same linear transformations it applies to the object. For articulated objects, the same approach applies to each articulated component individually. Sections 5 through 9 describe the hierarchy-building preprocess, focusing on how it maximizes the accuracy in each level of detail.

The time-critical detection algorithm uses these hierarchies to implement progressive refinement. When called by the application, the algorithm detects collisions between the level-0 spheres of the hierarchies, the objects' bounding spheres. Any broad-phase approach from Section 3 will suffice for this step. Should the broad phase find any level-0 spheres that collide as of time step t_d, the algorithm enters its narrow phase. Each step of the narrow phase descends one level in the hierarchies for one pair of objects (still at their t_d positions). Descending one level involves the obvious operations: the algorithm checks the colliding spheres at the current level of the two hierarchies to see if their children collide. Spheres are simple enough shapes that this collision checking is very efficient. If no spheres from the two hierarchies collide at the current level, then that pair of objects need no further processing as of t_d. Otherwise, the algorithm returns the colliding spheres at the current level to the application. If the application can devote more of the current frame's processing time to detection, it returns control to the algorithm to proceed with the next refinement step. The algorithm can continue these steps as long as the hierarchies have the levels and the application can spare the time.

The application is free to stop the refinement at any level if available time is exhausted. If spheres from two objects still collide at this point, the application should invoke collision response (e.g., make the objects bounce), using the colliding spheres as an approximation to the objects' surface contact; the accuracy of the detection and ensuing response depends on how tightly the colliding spheres fit the real surfaces of the objects. The application then proceeds to its next task for the current frame.

The processing time for this algorithm depends on the resolutions of the sphere hierarchies. It is independent of the objects' geometric complexity because the algorithm does not use the hierarchies to find parts of the objects' real surfaces to test for collisions. If time allows, however, the algorithm can test the real surfaces within colliding leaf spheres as a final level of refinement.

Note that this algorithm restricts progressive refinement to the narrow phase, with none occurring in the broad phase. Section 12 explains how this restriction can sometimes limit performance, and suggests future work that may provide a solution.

4.2 Choosing the Amount of Refinement

To use the approach from the previous section, the application must determine the time it can spend per frame on collision detection. The answer depends on the application's performance goals and the set of activities it performs at each frame. These activities are summarized in the code of the application from Figure 2. Ideally, the application would have time-critical algorithms for all these activities, so it could allot a specific amount of time for each one.

At the current state of the art, though, time-critical algorithms are rare. For our tests, we thus used traditional algorithms for the tasks other than collision detection. The main difficulty was predicting the time needed by the rendering algorithm. The simplest approach is to use the average rendering time over some number of previous frames. From this prediction the application estimates the time which will be unused during the current frame, and it allots this time to detection. Should rendering be faster than predicted, the application idles for the "slack" time at the end frame, to keep the frame from finishing too quickly. Section 11 presents results from this approach in practice.

The detection algorithm may find multiple objects whose sphere hierarchies collide at the current frame. The application should refine all these collisions in a round-robin fashion, to avoid spending all available time on just one collision. In practice, we found it useful to require that every collision be refined at least once (to level 1, one level better than an object's single bounding sphere); this approach prevented variations in predicted rendering time from forcing undue inaccuracy in collision detection.

5 Sphere Hierarchies

The hierarchies of spheres discussed in Section 4 involve several design decisions. First, spheres at different levels need some sort of relationship. An obvious approach is to make the hierarchy a tree, a *sphere-tree*. The algorithm for one step of the narrow phase, from Section 4.1, is thus straightforward and similar to hierarchical algorithms in other areas of graphics; see Figure 3 for pseudocode. This algorithm is efficient because testing two spheres for intersection is a very fast operation, and the hierarchy prunes the number of spheres to test. Another form of relationship (which we have not implemented) is a directed acyclic graph, a *sphere-DAG*, in which parents can share children. In this case, the detection algorithm uses a variation of the pseudocode from Figure 3, marking nodes to avoid repeated traversals of shared children.

```
refine_detection((s_1, s_2), S)
   S_1 ← children of s_1 that intersect s_2
   S_2 ← children of s_2 that intersect s_1
   for each pair (s'_1, s'_2), s'_1 ∈ S_1, s'_2 ∈ S_2
      if (s'_1 intersects s'_2)
         add (s'_1, s'_2) to S
```

Figure 3: This algorithm refines the detection of a collision involving spheres (s_1, s_2) from different sphere-trees, putting the result in S.

The next design decision involves how the sphere hierarchy covers an object's surface. The strictest policy is *fully conservative coverage*, under which children must collectively cover all parts of the object their parent covers. A looser policy is *sample-based coverage*, in which children must cover a set of specific points that their parent covers. Fully conservative coverage is simpler to implement for a sphere-DAG than a sphere-tree, as Section 9 discusses.

The final design decision is how to automatically generate the particular spheres in the hierarchy. This question is the topic of Sections 6 through 9.

6 Building Sphere Hierarchies

The success of the time-critical detection algorithm depends on the preprocess that builds the sphere hierarchies. The preprocess must meet three requirements. It must be automatic, building useful hierarchies without user intervention. It must ensure that each hierarchy is an effective search structure, with each level pruning the parts of the next level that could be visited by the detection algorithm. Finally, it must generate hierarchies in which each level fits the object as tightly as possible. This requirement is critical, because the detection algorithm could stop at any level and pass the colliding spheres to collision response. Response based on these spheres will be more accurate if the spheres' surfaces are closer to the objects' real surfaces.

6.1 Previous Approaches

The final requirement of tightness is the most challenging to meet, and it limits the applicability of previous work on hierarchies. Octree-like recursive subdivision is the basis for hierarchy-building algorithms described by Liu *et al.* [28] and by us in a previous paper [20]. An octree for an object defines a sphere-tree if each occupied octant is circumscribed by a sphere. A preprocess based on this approach has the advantages that it is straightforward to implement and quick to execute. For the sphere-tree in Figure 4, the preprocess took 2 seconds on a Hewlett Packard 9000/755. The disadvantage of this approach is that it does not often produce hierarchies that fit tightly. Figures 4 illustrates this problem, and Figures 11(a), 12(a) and 13(a) from Section 10 give further examples. These hierarchies prevent the time-critical detection algorithm from reaching acceptable accuracy, as empirical tests from Section 11 demonstrate.

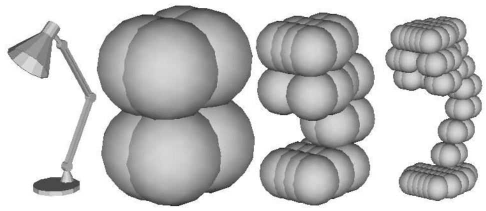


Figure 4: A desktop lamp (626 triangles), and three levels of its octree-based sphere-tree.

Another class of algorithms builds a bounding hierarchy from the leaves up, with leaf spheres enclosing "primitive" pieces of an object. These algorithms require an appropriate set of primitives and a way to designate siblings at each hierarchy level. Youn and Wohn [51] and Rolhf and Helman [40] assume that the designer of an object addresses both of these issues by modeling the object as a hierarchy of pieces; these algorithms thus do not meet the requirement of producing hierarchies automatically. Ray-tracing renderers commonly use bounding hierarchies to reduce the number ray-object intersection tests. Algorithms that build hierarchies for that application are not appropriate in the context of collision detection, though, for two reasons. First, these algorithms are meant to process scenes of many objects with clearly defined primitives; the algorithms are not capable of breaking a single object into primitives that yield an effective hierarchy. Second, these algorithms optimize the hierarchy for the characteristics of ray-object intersection tests, an approach which does not necessarily lead to the tightly fitting sphere hierarchies needed for collision detection. Kay and Kajiya [25], for example, build bounding shapes that are cheap to intersect with rays, but these shapes do not have the rotational invariance necessary for moving objects. Goldsmith and Salmon [16], as another example, show that a hierarchy prunes ray-object intersection tests most effectively if it minimizes surface area; this criteria does not necessarily encourage a tight fit around an object, though, as it can create artifacts like "caps" over concavities.

The most successful approach for tightly approximating an object with spheres is the work of O'Rourke and Badler [34]. Their algorithm fits spheres to a polyhedron by anchoring big spheres to points on the polyhedron and shrinking the spheres until they just fit inside the polyhedron. Badler, O'Rourke and Toltzis [1] extend this approach to build two-level hierarchies, but they do not consider the more general hierarchies of greater depth necessary for a time-critical detection algorithm. Section 10 examines this approach further, comparing it to a new approach whose details are the topic of the intervening sections.

6.2 A New Approach

Inasmuch as no previous hierarchy-building algorithms fully satisfy all three requirements of the time-critical detection algorithm, we present a new approach. The motivation for this approach is Blum's *medial-axis* [4], which corresponds to a "skeleton" or "stick figure" representation of a two-dimensional (2D) object. Figure 5(a) shows an example. A more technical definition involves the locus of points equidistant from two sides of the object. The three-dimensional (3D) version is

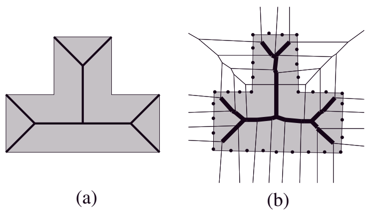


Figure 5: (a) The heavy lines are the 2D medial axis of the grey polygon. (b) A Voronoi diagram approximating the medial axis.

a *medial-axis surface*. This structure contains surfaces rather than lines, but it remains analogous to a skeleton.

The symmetries of an object around its medial-axis surface suggest that the latter could guide the placement of spheres that approximate the object. Spheres placed in this manner often correspond closely to the spheres a person would choose when approximating the object manually. Section 8 describes an algorithm that centers many spheres on the medial-axis surface and then "merges" them to reduce the number while optimizing the accuracy with which the object is approximated. As a precursor to this algorithm, the Section 7 discusses the building of medial-axis surfaces.

7 Building Medial-Axis Surfaces

An algorithm that builds a sphere hierarchy from a medial-axis surface must first build the medial-axis surface. This problem is not simple, and the literature contains few solutions. Hoffmann [19] presents the only exact algorithm, but it is complicated and limited to constructive solid geometry (CSG) objects. Fortunately, building sphere hierarchies does not require exact medial-axis surfaces; approximations suffice.

One way to approximate a medial-axis surface uses a *Voronoi diagram* [37]. The Voronoi diagram for a discrete set of points identifies, for each point, the region of space closer to that point than to any of the other points. The regions are called *Voronoi cells.* For 2D points the cells are convex polygons; for 3D points they are convex polyhedra. Each face of a cell is equidistant between two points in the discrete set. Thus, for a set of points, P, on the surface of a polyhedron, the Voronoi diagram's cells have faces lying roughly on the medial axis. Figure 5(b) shows an analogous 2D situation. This idea could work for nonpolyhedral objects, but this paper focuses on the polyhedral case.

Goldak *et al.* [15] develop this idea into an algorithm. Their algorithm identifies the *Voronoi vertices* (corners of Voronoi cells) that lie on the medial-axis surface. In their description of the algorithm, Goldak *et al.* emphasize the general properties at the expense of specific details. The remainder of this section summarizes the practical enhancements to the algorithm necessary for a successful implementation.

The algorithm's first step places the set of points, P, on the polyhedron's surface. The number of points is a parameter set by the user. In our experience, the algorithm works better when P covers the polyhedron uniformly, and the simplest way to achieve this goal is to use Turk's point-placement algorithm [48]. It first creates a random distribution of points and then applies a relaxation technique to make the distribution even.

The second step is building a Voronoi diagram for the points. The literature contains several algorithms for 3D Voronoi diagrams. Bowyer [5] presents a straightforward algorithm that incrementally adds points to the diagram. This algorithm is not numerically robust, however, so Inagaki *et al.* [24] rephrase the algorithm to use topological properties rather than necessarily-inexact geometric computations. Unfortunately, these extensions compromise accuracy. As a remedy, we extend the algorithm to choose between equally-valid topological situations based on estimated accuracy [21].

Each Voronoi vertex is the center of a sphere on which lie four points from the set P. The four points thus associated with a vertex are its *forming points*. Figure 6 shows a 2D example, in which spheres are replaced by circles and the number of forming points is reduced to three. The spheres

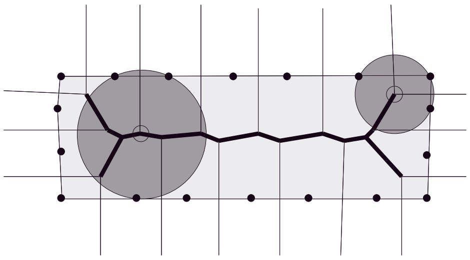


Figure 6: Each of these two circles are centered at Voronoi vertices and touch the object at three forming points.

centered at Voronoi vertices and the associated forming points are the foundation of the algorithm for building hierarchies from medial-axis surfaces, as Section 8 will explain.

Only the Voronoi vertices interior to the polyhedron lie on the medial-axis surface. The third step of the algorithm identifies these vertices. Inclusion testing is a standard geometric problem, and extensions of 2D algorithms [37] apply.

Vertices in the Voronoi diagram are adjacent to each other if they lie on the same face of a Voronoi cell. These faces may legitimately have zero area, in which case vertices coincide. In the algorithm from Section 8, it is convenient to treat a set of coincident vertices as one vertex that inherits the adjacency of the set. To identify coincident vertices, simple space subdivision [47] is helpful. Once adjacency is recorded, the medial-axis surface is sufficiently complete for the algorithm from Section 8.

There is the possibility of aliasing problems in the medial-axis surface. A low density of points from P on narrow “necks” of the polyhedron or along narrow “gaps” can cause the medial-axis surface to disconnect or bridge a gap. Preventing these problems in advance seems difficult. In our experience, the best solution is to detect the problems after they occur and then add more points to correct them. This process involves finding certain intersections between Voronoi-cell faces and the polyhedron, in particular, intersections with parts of the polyhedron that are not adjacent to the point $p \in P$ contained in the cell. For each such intersection, P gets an additional point, p', the projection of p onto the region of intersection. Complete details of this process appear elsewhere [21].

8 Building Sphere Hierarchies from Medial-Axis Surfaces

The previous section’s algorithm generates a large set of Voronoi vertices on a polyhedral object’s medial-axis surface. Each vertex and its forming points define a sphere, and these spheres tightly approximate the polyhedron. The topic now is reducing the number of spheres while preserving as much tightness of the approximation as possible. After discussing this reduction process, this section will describe how the hierarchy-building preprocess applies it multiple times to produce a hierarchy with multiple levels of detail.

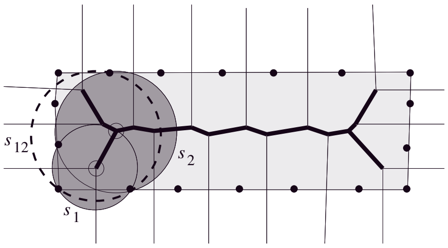


Figure 7: The merger of spheres s_1 and s_2 forms s_{12}.

The strategy for reducing the number of spheres is to "merge" adjacent spheres. Merging two spheres, s_1 and s_2, involves replacing them with a new sphere, s_{12}, that covers the parts of the polyhedron they cover. For efficiency, the preprocess approximates what a sphere covers in terms of its forming points from the set, P, of points on the polyhedron's surface (as defined in the previous section). Thus, s_{12} is the bounding sphere for the forming points associated with s_1 and s_2; we use Ritter's method [39] to compute a nearly-optimal bounding sphere. Figure 7 illustrates a 2D example of merging. After the merge, s_{12} stores the union of the forming points from s_1 and s_2, and any future merge involving s_{12} must cover these points.

Producing a level of detail involves repeatedly merging pairs of spheres. Choosing the pairs to merge is an optimization problem: at each repetition, choose the minimum-cost merger, that is, the candidate pair whose merger most preserves the level's tightness around the polyhedron. Note that this approach of optimizing each repetition independently is a greedy approach, so it does not guarantee that the final result will have the tightest possible fit; nevertheless, this approach has produced tight fits in every test we have conducted. To identify candidate pairs for merging, the preprocess uses the adjacency properties of the medial-axis surface (as defined in the previous section). Initially, all spheres are centered at Voronoi vertices on the medial-axis surface, and a pair of spheres is a merging candidate if their vertices are adjacent in the Voronoi diagram. When s_1 and s_2 merge to become s_{12}, then s_{12} becomes adjacent to the spheres adjacent to s_1 and s_2. By restricting attention to only adjacent spheres, the preprocess avoids considering mergers of spatially distant spheres, the results of which would be undesirably large spheres. Adjacent spheres also reflect the medial-axis surface's ability to trace the skeleton of a polyhedral object, so mergers of adjacent spheres tend to respect the conceptual organization of the object.

The cost function for the optimization must return a low value when the merger of candidates s_1 and s_2 to form s_{12} would preserve tightness around the polyhedron. A useful approach is based on the *Hausdorff distance* [37] from s_{12} to the polyhedron. This distance is defined as the *maximum*, over all points on s_{12}'s surface, of the *minimum* distance from that point to the polyhedron. A cost function computing this distance would return low costs in the appropriate cases, but we have found no way to compute this distance exactly for nonconvex polyhedra; as we explain elsewhere [21], a straightforward algorithm (albeit one with a complicated correctness proof) computes this distance exactly for convex polyhedra, but the algorithm seems to overestimate the distance for nonconvex polyhedra, even when expressed as unions of convex pieces. We thus use a function that approximates the Hausdorff distance. For each forming point of s_1 or s_2, the function measures the distance to s_{12}'s surface along the normal direction for the polygon containing the point; the maximum of these distances is the cost returned by the function. To compute the distance for a forming point at position $\mathbf{p}$, the function first projects s_{12}'s center, $\mathbf{c}$, onto the plane of the polygon containing $\mathbf{p}$.

Letting $\mathbf{c}'$ be the projection of $\mathbf{c}$, d_1 be the distance from $\mathbf{c}$ to $\mathbf{c}'$, d_2 be the distance from $\mathbf{p}$ to $\mathbf{c}'$,

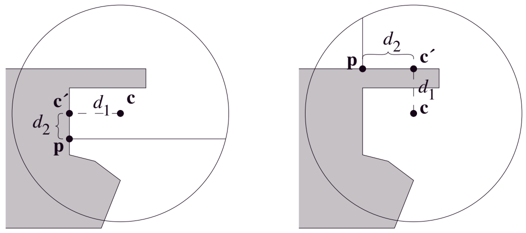


Figure 8: Computing the distance from $\mathbf{p}$ to a sphere.

and r be the radius of s_{12}, then the distance from $\mathbf{p}$ to s_{12} is

$$\sqrt{r^2 - d_2^2} \pm d_1.$$

Whether to add or subtract d_1 depends on the relative positions of $\mathbf{p}$ to $\mathbf{c}$, as illustrated in Figure 8. This form of the cost function strikes a balance between efficiency and accuracy that works well for the merging operation. Section 9.2 discusses a different, more accurate approximation which is useful for measuring the tightness of a hierarchy when the merging is complete.

To control the repetitions of merging, the preprocess uses a priority queue. This queue ranks pairs of adjacent spheres according to their merging cost, allowing the preprocess to quickly find the optimal merge. After each merge the preprocess updates the queue to reflect changes in adjacency. Building the queue initially takes some time, but updating it is very efficient.

Building a full hierarchy requires that the preprocess apply the repeated-merging technique in multiple passes. For a sphere-tree, the preprocess first merges all the spheres centered at Voronoi vertices to produce a fixed number of children for the root (the polyhedron's bounding sphere). To subsequently build children of a parent anywhere in the hierarchy, the preprocess starts over with all the Voronoi vertices whose spheres merged to form the parent; it repeatedly merges these spheres until the desired number of children remain (we give each parent eight children, to match the branching factor of an octree). For a sphere-DAG, the preprocess merges the spheres centered at all Voronoi vertices down to the appropriate number for each level independently.

The merging process only guarantees that forming points are covered. The rest of the polyhedron tends to get covered as well, but to ensure fully conservative coverage the techniques from Section 9 are necessary. Section 10 shows sphere-trees built with the merging process and discusses the preprocessing time involved.

9 Completing the Hierarchies

Once the preprocess has used merging to build a sphere hierarchy that approximates a polyhedral object, two extra steps can give the hierarchy some useful properties.

9.1 Ensuring Conservative Coverage

The algorithm from Section 8 tends to produce spheres whose coverage of the object is nearly conservative. Fully conservative coverage is also valuable, though. With this form of coverage, the time-critical algorithm can support a final level of exact detection: leaf spheres store the object's polygons they intersect, and the algorithm compares the polygons in colliding leaves from different hierarchies.

Conservative coverage has different requirements for different types of hierarchies. For a sphere-tree, the safest policy is for children to collectively cover everything their parent covers. For a sphere-DAG, all the spheres at hierarchy level ℓ must collectively cover the whole polyhedron (assuming the parents at level $\ell - 1$ have as children every level-ℓ sphere they intersect). In each case, the basic operation is determining if each polyhedral face—assumed to be a triangle—is covered by a set of spheres. Checking a triangle for coverage is a 2D problem: each sphere that intersects the triangle corresponds to solid 2D disk in the triangle's plane, and the union of these disks must cover the triangle. For sphere-trees, the disks need only cover the part of the triangle inside another "clipping disk," corresponding to the parent sphere.

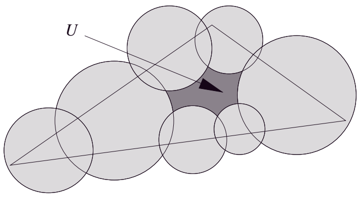


Figure 9: The boundary lemma.

The outer circles or boundaries of these solid disks figure prominently in the solution to the 2D problem. Specifically, the following *boundary lemma* simplifies the problem: the triangle (within the clipping disk) is covered by the disks if an only if for each disk, the part of its boundary inside the triangle (and within the clipping disk) is covered by other disks. We prove this lemma elsewhere [21], but Figure 9 gives an intuitive justification; note that the uncovered region, U, is ringed by uncovered portions of disk boundaries. Implementing a coverage-checker based on the boundary lemma is straightforward, because the book-keeping involves primarily set operations on one-dimensional intervals representing disk boundaries. If any face is found to be not covered, the simplest remedy is to iteratively enlarge the spheres that intersect the face until there is coverage.

In our experience, applying the safest policy to sphere-trees made them noticeably "looser." We thus chose to apply the sphere-DAG policy to sphere-trees, which at least guarantees that every polygon is fully covered by some set of spheres. Section 11 gives empirical results which suggest that this approach can be as conservative as the safest policy in practice.

9.2 Measuring Accuracy

A running application might want the time-critical detection algorithm to report the inaccuracy of each collision as it is detected. Recall from Section 2 that an important factor in the inaccuracy of a detection algorithm is the separation distance between two objects it designates as colliding. Unfortunately, computing this distance exactly is expensive enough that doing so negates the benefits of time-critical collision detection.

An upper bound on this distance is quickly computed, however, if the preprocess stores with each sphere of the hierarchy the distance from the sphere to the polyhedron. This distance is defined in Section 8 to be the Hausdorff distance: the *maximum*, over all points on the sphere's surface, of the *minimum* distance from such a point to the polyhedron. When two spheres from different hierarchies collide, the sum of their Hausdorff distances is an upper bound on the enclosed polyhedra's separation distance, as illustrated in Figure 10. A tighter upper bound is this sum minus the overlap of the

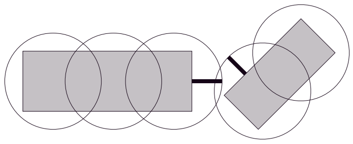


Figure 10: The separation distance is less than the sum of the Hausdorff distances (heavy lines).

spheres. For collisions involving multiple pairs of spheres from the two hierarchies, the best estimate of the separation distance is the minimum of the upper bounds over all the pairs. This approach gives a true upper bound only if the sphere hierarchy covers the object conservatively, but the technique from Section 9.1 will make any hierarchy at least nearly conservative, so we advocate this view of accuracy in general.

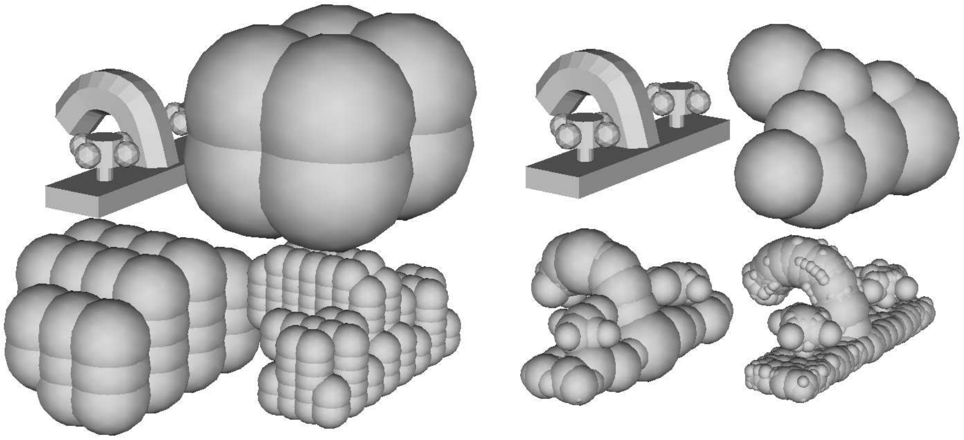
As Section 8 mentions, there seems to be no way to compute the exact Hausdorff distance from a sphere to a nonconvex polyhedron. That section derived an inexpensive approximation to the distance that works well for the merging operation. For the purposes of measuring separation distance, a more expensive and accurate approximation is useful; the extra expense is justified because the preprocess performs this computation only once, when the hierarchy is complete. The approach involves computing the minimum distance to the polyhedron from a discrete set of points on the sphere, such as the vertices of an inscribed dodecahedron. The maximum of these distances approximates the true Hausdorff distance. Adding a correction term—the maximum distance from the sphere's surface to the nearest dodecahedron vertex—makes the approximation a true upper bound. Computing the minimum distance from a dodecahedron vertex to the polyhedron is straightforward, involving computing the minimum distance to each polyhedral face. Space subdivision techniques can prune the set of faces, making the computation more efficient.

10 Sphere Hierarchy Results

The hierarchy-building algorithm from Sections 7 through 9 worked well in our tests. Figures 1, 11(b), 12(b) and 13(b) show some results. All four sphere-trees are conservative according to the sphere-DAG policy from Section 9.1. On a Hewlett Packard 9000/735, building the sphere-tree for the lamp took 12.4 minutes, the faucet 14.2 minutes, the rocket 21.8 minutes and the truck chassis 2.7 hours. These times included all operations, including measuring accuracy as in Section 9.2. Note that these preprocessing times are amortized each time an object is used in an application.

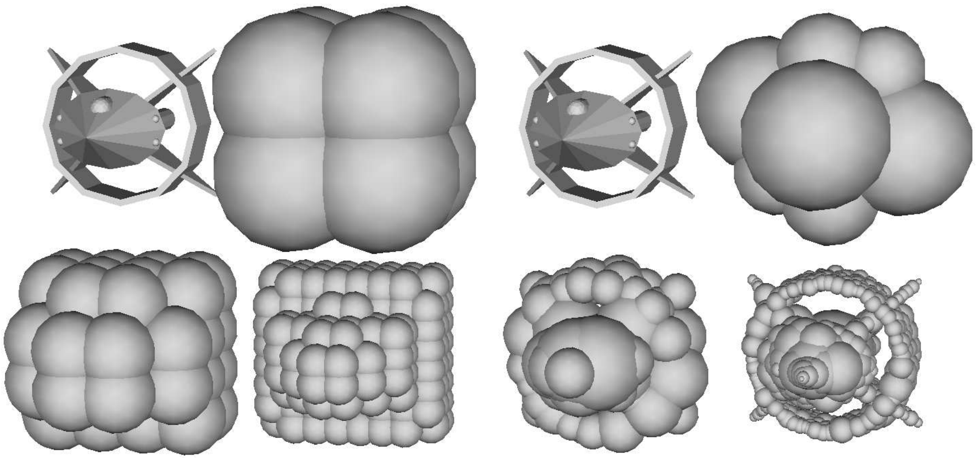
The results of the hierarchy-building algorithm compare favorably to an octree-based algorithm in terms of accuracy. Figures 4, 11(a), 12(a) and 13(a) show that the results of the octree algorithm are visibly less accurate. For a more quantitative comparison, we used the technique of Section 9.2 to measure the Hausdorff distance to the polyhedron for each sphere in each hierarchy. For a given level of the hierarchy, two measures of the inaccuracy are the average and maximum Hausdorff distances over all spheres at that level. Figure 14 gives these measures for the four objects. By these measures, the inaccuracy of the sphere-trees built with our algorithm is only a fraction of the inaccuracy for octree-based sphere-trees.

It is worth comparing the algorithm from Sections 7 through 9 to the work of Badler, O'Rourke and Toltzis [1, 34]. Their algorithm also begins with a set of points distributed over the surface of the polyhedron. The algorithm repeatedly chooses a point, anchors a big sphere to it and shrinks the sphere until it bounds none of the other points; this process produces a set of spheres that fit just



(a) Octree. (b) Medial-axis surface.

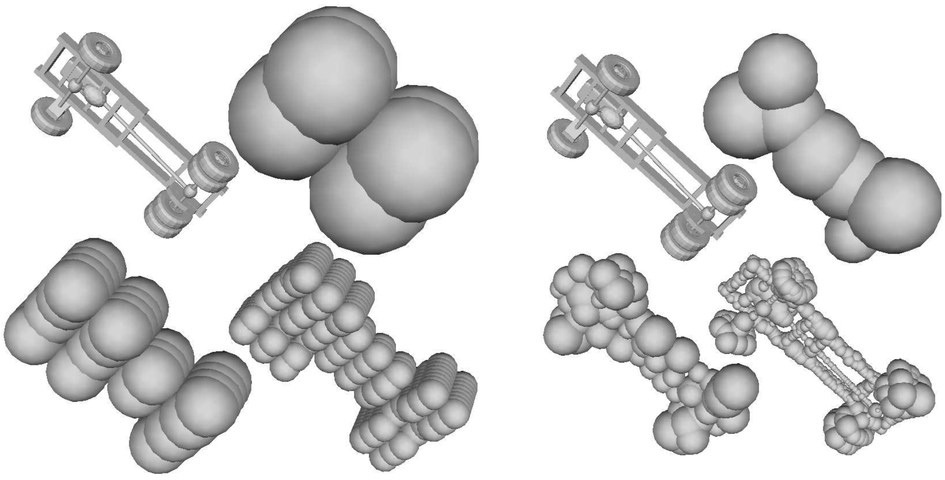
Figure 11: A bathroom faucet (1288 triangles), and two forms of sphere-trees.



(a) Octree. (b) Medial-axis surface.

Figure 12: A rocket (1420 triangles), and two forms of sphere-trees.

inside the polyhedron. For an articulated object, the algorithm bounds the spheres associated with each jointed component with another sphere, creating a two-level hierarchy. This work is important as early evidence that objects can be successfully approximated with spheres. Our work continues in this tradition, and adds several significant improvements: building hierarchies of more than two levels, optimizing the tightness of the spheres' fit, checking for conservative coverage, measuring accuracy and using all these techniques in a time-critical context. These improvements are essential for a detection algorithm that meets the needs of interactive applications.



(a) Octree. (b) Medial-axis surface.

Figure 13: A truck chassis, seen from below (10,171 triangles), and two forms of sphere-trees.

model	level	fraction of octree's average inaccuracy	maximum inaccuracy
lamp	1	0.442	0.494
	2	0.220	0.485
	3	0.164	0.433
faucet	1	0.492	0.552
	2	0.358	0.439
	3	0.377	0.687
rocket	1	0.654	0.728
	2	0.546	0.690
	3	0.379	0.979
truck	1	0.571	0.671
	2	0.389	0.450
	3	0.286	0.531

Figure 14: Accuracy of medial-axis sphere-trees compared to octree sphere-trees.

11 Collision Detection Performance

To see how well the time-critical detection algorithm exploits the sphere hierarchies, we tested its performance empirically, comparing it to an algorithm based on BSP trees [46]. These tests demonstrated that hierarchies built from medial-axis surfaces provide acceptable accuracy (improving on hierarchies built from octrees) and significant speedups. Most importantly, the hierarchies allowed interactive performance that was not otherwise possible, and they maintained this performance as objects became more complex.

11.1 Sample Application

The context of the tests was a simple spaceship simulator. This simulator allows a user to interactively control a ship flying among autonomous drone ships. The user controls her ship's forward acceleration and rotational velocity, and the simulator computes its motion according to a simplified dynamics model, solving the ordinary differential equations using a second-order Runge-Kutta method with adaptive step size [38]. The drones move according to the same model and pick their control parameters at random every few seconds. Each ship is free to collide with each other ship, and the simulator detects all collisions using one of the detection algorithms, or it calls both to compare their performance.

The ships can use as geometry any of the models from Figures 1, 11, 12 or 13. We built sphere-trees and BSP trees for the particular geometry as preprocessing. The sphere-trees had levels 1 through 3 depicted in the figures plus an additional level 4. The BSP trees were optimized to avoid face-splitting, a heuristic which pilot studies indicated makes the BSP trees more efficient. At runtime, both detection algorithms used a broad phase based on space-time bounds [20, 22] to find bounding-sphere collisions.

The simulator's collision response is quite simple. When two ships collide, the response algorithm determines if the ships are converging by checking the relative velocities of the colliding spheres (for BSP trees, it uses the bounding spheres). If they are converging, it applies energy and momentum conservation to change the ships' velocities, causing them to bounce. It ignores rotational velocity for simplicity. The user's ship has infinite mass, so it affects other ships but is not itself affected. This collision response improves on what we used in our companion paper [22], which "teleports" the colliding objects to their non-colliding positions from the start of the simulation.

11.2 Performance at Each Hierarchy Level

The first set of tests studied the performance available from each level in the sphere-trees. The ships used the lamp geometry from Figure 1; we chose this shape because the deep concavities create complicated collision patterns. The simulator application called both detection algorithms at each frame. It refined the sphere-trees to the deepest colliding level; it also used the idea from Section 9.1 for comparing polygons stored in leaf spheres, giving an extra level of refinement designated level 5. For each level ℓ, it recorded the *speedup* of sphere-trees, defined as:

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

Note that a speedup greater than 1 indicates that the sphere-trees are faster. These tests ran on Hewlett Packard 9000/755, with the user-controlled ship and ten drones. The total number of calls to each algorithm (each call corresponding to one pair of ships whose bounding spheres collided) was 86,083, and the calls featured ships in a wide variety of relative orientations.

Histograms of the speedups for all 86,083 calls appear in Figure 15. The histograms show that, not surprisingly, the speedup was greater when the detection algorithm had not descended so far down the sphere-tree. Even so, sphere-trees were significantly faster than BSP trees at all levels except level 5, the level of the objects' real surfaces; even at this level, sphere-trees were faster in 75.1 percent of the calls.

If the application were to stop the refinement at a level with colliding spheres, it would invoke collision response on objects that are not quite touching, creating some inaccuracy. As Section 9.2 discusses, the algorithm can report its inaccuracy in this situation, giving an upper bound on the separation distance between the objects. The histograms in Figure 16 show how this measure of

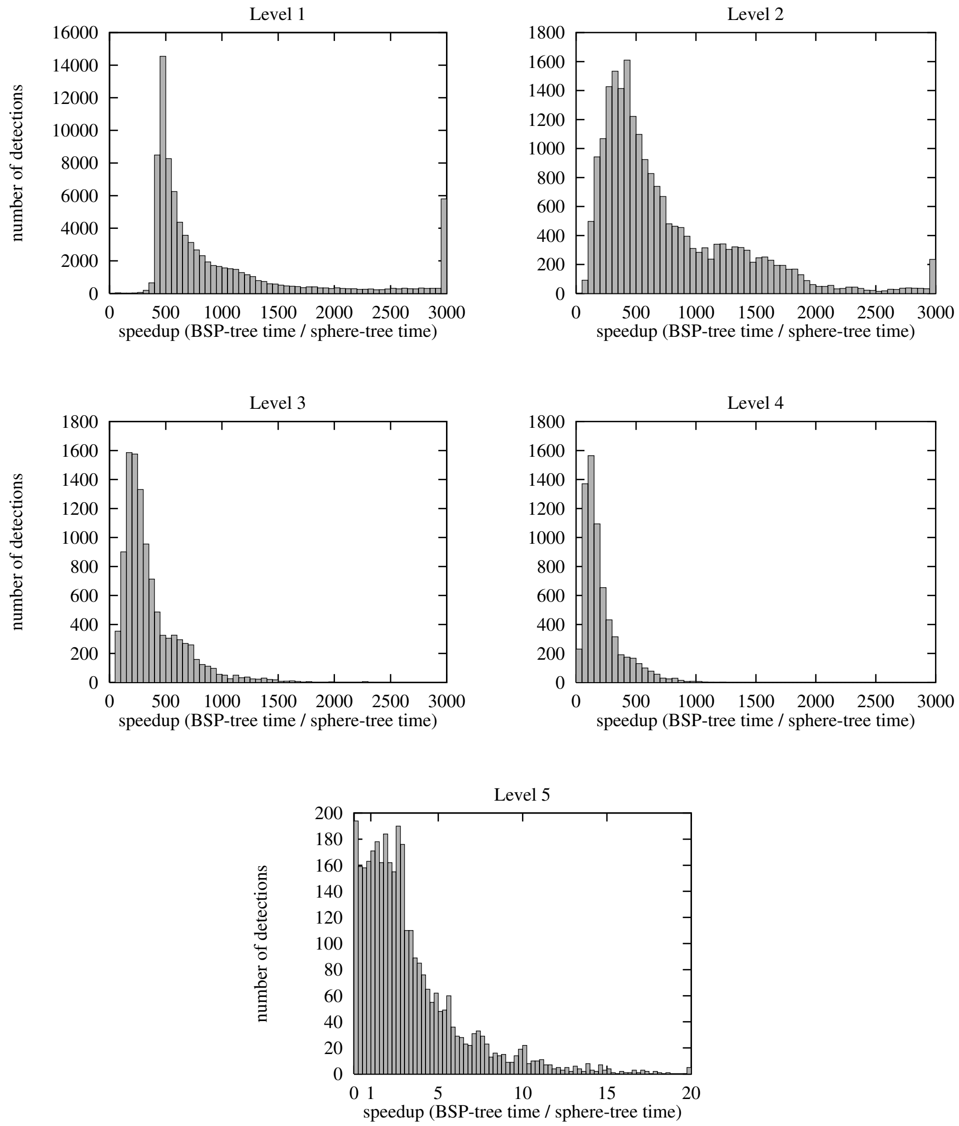


Figure 15: Histograms of the speedup of sphere-trees over BSP trees.

inaccuracy improved for deeper levels in the sphere-trees. These results and the histograms from Figure 15 suggest that time-critical collision detection does allow a flexible trade-off between speed and accuracy.

The tests also suggest an interesting conjecture about conservative coverage. Using level 5 (testing the polygons stored in the level-4 spheres), the sphere-tree algorithm found the same set of 804 collisions that the BSP-tree algorithm detected. One would expect this result only if the sphere-tree does actually cover the object conservatively. So applying the simpler sphere-DAG policy for conservative coverage to sphere-trees (see Section 9.1) may be as useful in practice as generating full coverage.

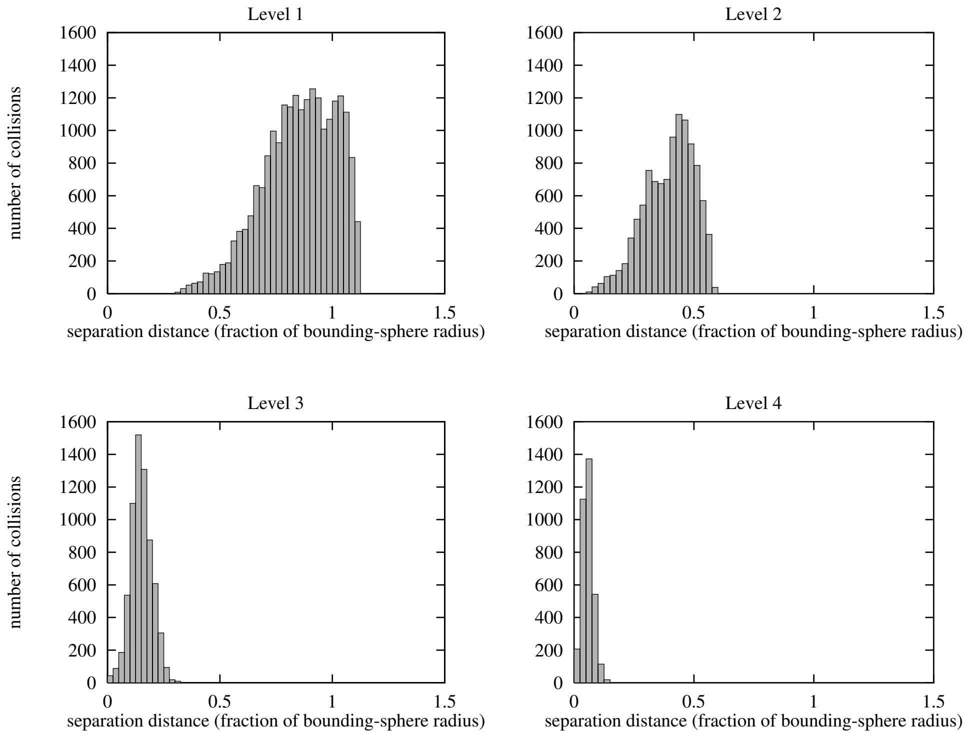


Figure 16: Histograms of sphere-tree accuracy.

11.3 Sustainable Real-Time Performance

The second set of tests evaluated the detection algorithms' ability to maintain real-time performance. We ran the application once using BSP trees, then again using two versions of sphere-trees. For the runs using sphere-trees, the application used the strategy from Section 4.2, stopping the detection algorithm's progressive refinement in order to meet a target frame rate.

These tests ran on a Hewlett Packard 9000/755 with TVRX T4 graphics acceleration. No time-critical operating system or rendering were available. Having these components would likely improve the performance of the application, but it is interesting to see how time-critical collision detection alone affects performance.

The first set of tests involved the user's ship plus nine drones. As in the tests from Section 11.2, each ship used the 626-triangle lamp geometry from Figure 1. Simulation time ran from 0 to 40, with the application rendering frames every $\Delta t_r = 0.1$ time units. The minimum temporal resolution for detection was $\Delta t_d = 0.05$ units. Note that the detection algorithms thus sample each object's position twice per frame. The application's performance goal was making simulation time match wall-clock time, that is, computing each frame in 0.1 seconds.

For a run involving 780 collisions[1], Figure 17 tracks per-frame processing time for BSP trees. The time includes times for the broad phase and collision response, but these times were insignificant.

[1]The number of collisions was rather large for 400 frames because the simple collision response required several frames to fully eliminate some collisions.

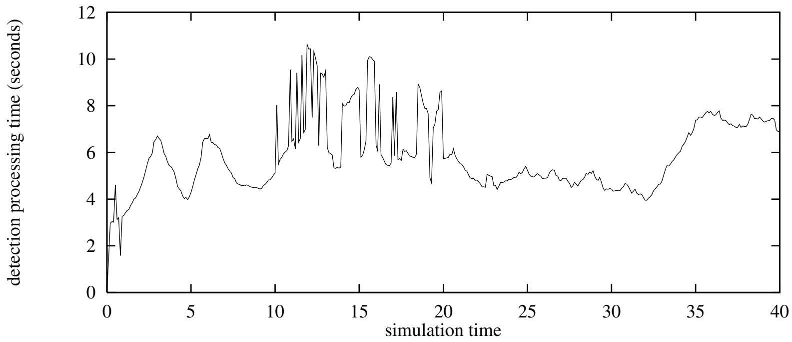


Figure 17: Peformance with BSP trees for the test involving ten lamps. For detection time, mean = 5.825, std. dev. = 1.586, max. = 10.627.

This graph indicates that BSP trees caused the application to miss its target frame time at almost all frames, often being between 10 and 100 times too slow. An MPEG animation of this run is available on the World Wide Web at URL http://www.acm.org/pubs/tog/hubbard96/; follow the link labeled “Run 1”.

The next run used the time-critical algorithm with octree-based sphere-trees, as depicted in Figure 4. Performance was much closer to the target of 0.1 seconds per frame, but accuracy was poor. Due to the “looseness” of the octree-based sphere-trees, objects separated by large distances still had colliding sphere-trees when the detection algorithm exhausted its allowable processing time; thus, collision response made objects bounce when they were nowhere near touching. The World Wide Web page mentioned above shows an MPEG animation of this run; follow the link labeled “Run 2”. This animation demonstrates the disadvantages of hierarchy-building techniques that do not address the specific needs of time-critical collision detection.

In the third run, the time-critical algorithm used sphere-trees built from medial-axis surfaces. The run involved 274 collisions; Figure 18 categorizes these collisions by the sphere-tree levels at which collision response was invoked. A performance profile of the run appears in Figure 19. The graph shows the time spent by collision detection (including the broad phase and collision response) and rendering, as well as the “slack” time (see Section 4.2) and the overall frame time[2]. Note that the frame time meets the target of 0.1 seconds at most frames, and the deviations that do occur are small. The accuracy is also quite reasonable, as can be seen in an MPEG animation from the World Wide Web page mentioned above; follow the link labeled “Run 3”. Figure 21 shows a frame from this animation. This test confirms that the time-critical algorithm can provide real-time performance not possible with the BSP algorithm.

For a final test, we ran the application with significantly more complex geometry. Nine drone ships each used the 10,171-triangle truck geometry from Figure 13, and the user’s ship used the lamp geometry. The total number of triangles was thus 92,165, which is more than ten times the number tested by Smith *et al.* [44], the maximum in the literature for nonconvex objects. With this many triangles, rendering became the bottleneck on our hardware, taking more than 0.9 seconds at almost every frame. The application thus aimed for a target of 1 frame per second; this target is clearly not interactive performance, but it limits collision detection to a time budget that would allow interactive performance with faster rendering. For 43 collisions (all at level 4), the application met the target

[2]To measure the time spent solving the differential equations for motion control would have required too many time-consuming calls to the system clock, so portions of this time are included in both the detection and rendering times. Pilot studies indicated that the time the application spent on motion control was insignificant, however.

level	collisions
1	82
2	25
3	26
4	141

Figure 18: Sphere-tree levels at which collision response was invoked, for the test involving ten lamps.

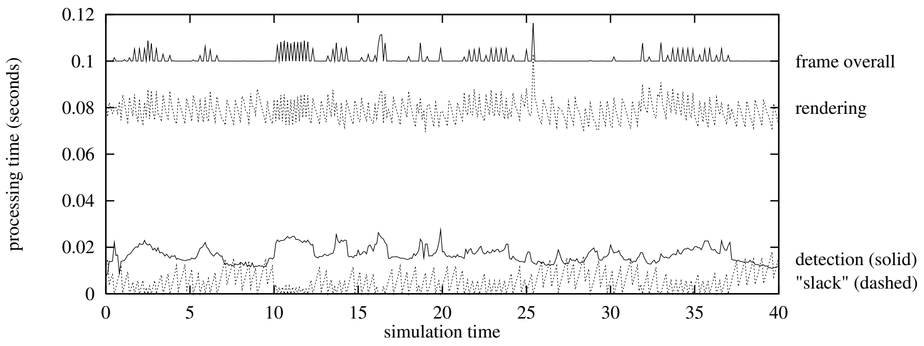


Figure 19: Performance with sphere-trees from medial-axis surfaces for the test involving ten lamps. For frame time, mean = 0.101, std. dev. = 0.002, max. = 0.116. For detection time, mean = 0.019, std. dev. = 0.003, max. = 0.028.

frame rate, as illustrated in the profile from Figure 20. Figure 22 shows a snapshot of the run, and an MPEG animation showing more appears on the World Wide Web page mentioned above; follow the link labeled "Run 4". Note that the detection algorithm used an average of only 0.018 seconds per frame. These results are evidence that time-critical collision detection does maintain real-time performance as objects become more complicated.

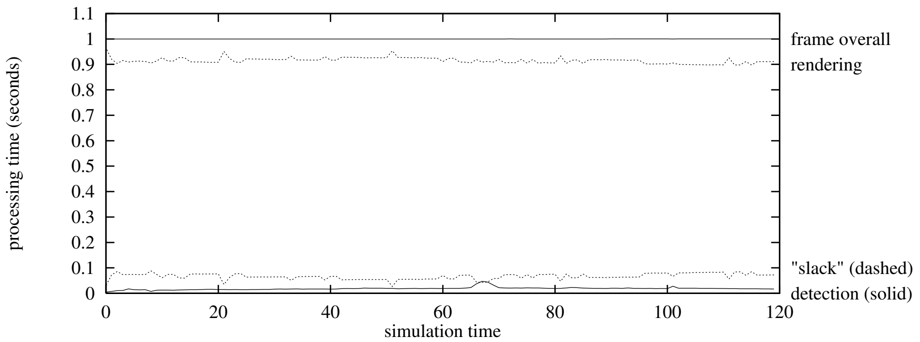


Figure 20: Performance with sphere-trees from medial-axis surfaces for the test involving one lamp and nine truck chasses. For detection time, mean = 0.018, std. dev. = 0.005, max.= 0.047.

12 Conclusions and Future Work

This paper presents a time-critical collision-detection algorithm that trades accuracy for speed. The foundation of the algorithm is a preprocess that builds sphere hierarchies automatically from medial-axis surfaces of polyhedral objects; this preprocess specifically optimizes the tightness with which each hierarchy level approximates an object. Empirical results indicate that this preprocess improves on previous hierarchy-building techniques in meeting the needs of time-critical collision detection. Tests with a sample application demonstrate that the time-critical detection algorithm provides acceptable accuracy while maintaining real-time performance that is not possible with a previous algorithm. These tests also indicate that the time-critical algorithm can preserve real-time performance even as geometric complexity increases.

This work suggests several extensions and improvements. The hierarchy-building algorithm works well but it was not simple to implement. The most complicated part is the algorithm from Section 7 for medial-axis surfaces, so a simpler approach to this subproblem would help. A version of the algorithm to build sphere-DAGs is also worth implementing. Another issue to explore is redundancy, that is, spheres whose removal from the hierarchy does not affect conservative coverage. The boundary lemma from Section 9.1 would help to detect redundancy, but we have not yet experimented with an implementation based on it. Hierarchies of spheres may not be the best way to approximate some objects. A flat wall, for example, poses problems in that many spheres are required to form a tight approximation for a few triangles. One solution to this problem might be to use hybrid hierarchies that incorporate boxes (or some other flat-side primitive) in addition to spheres. Regardless of the hierarchy primitives, the algorithm that traverses the hierarchies may be able to exploit inter-frame coherence. Remembering the hierarchy nodes that collided at the previous frame may reduce the work required to find the colliding nodes for the current frame.

Because all of the algorithm's progressive refinement occurs in the narrow phase, there may be situations in which the algorithm cannot guarantee real-time performance. If many pairs of objects are simultaneously colliding or nearly colliding, then the broad phase may not have enough time to detect all collisions between the objects' bounding spheres. Solving this problem requires some sort of progressive refinement in the broad phase. One approach would involve allowing the broad phase to selectively ignore collisions between user-designated "unimportant" objects. The broad phase would devote its time to objects that are important for the particular application, moving on to other objects only when it has extra time.

13 Acknowledgements

The majority of this work 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. Andy van Dam, Jim Kajiya, Franco Preparata, Peter Shirley and Don Greenberg also made important contributions. Jim Arvo deserves special mention for implementing the tests of random cube rotations from Section 3. At Brown and Cornell, this work was supported in part by the NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization, and by Hewlett Packard. At Brown, additional support was provided by: Sun Microsystems; Autodesk; Taco, Inc.; ONR grant N00014-91-J-4052 ARPA order 8225; NCR; IBM; Digital Equipment Corporation; Apple; and Microsoft.

References

[1] Norman I. Badler, Joseph O'Rourke, and Hasida Toltzis. A spherical representation of a human body for visualizing movement. *Proceedings of the IEEE*, 67(10):1397–1403, October 1979.

[2] David Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. In *Proceedings of SIGGRAPH '90,* published as *Computer Graphics*, 24(4):19–29, August 1990.

[3] David Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies.* PhD thesis, Department of Computer Science, Cornell University, March 1992.

[4] Harry Blum. A transformation for extracting new descriptors of shape. In Weiant Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Form*, pages 362–380. MIT Press, Cambridge, Massachusetts, 1967.

[5] Adrian Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[6] Frederick P. Brooks, Jr. Grasping reality through illusion—interactive graphics serving science. In *Proceedings of CHI '88*, pages 1–11, May 1988.

[7] Stephen A. Cameron. Collision detection by four-dimensional intersection testing. *IEEE Transactions on Robotics and Automation*, 6(3):291–302, June 1990.

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

[9] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav 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.

[10] David P. Dobkin and David G. Kirkpatrick. Fast detection of polyhedral intersection. *Theoretical Computer Science*, 27(3):241–253, December 1983.

[11] Tom 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.

[12] André Foisy, Vincent Hayward, and Stéphane Aubry. The use of awareness in collision prediction. In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation*, pages 338–343, 1990.

[13] Thomas A. Funkhouser and Carlo 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, August 1993.

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

[15] John A. Goldak, Xinhua Yu, Alan Knight, and Lingxian Dong. Constructing discrete medial axis of 3-D objects. *International Journal of Computational Geometry and Applications*, 1(3):327–339, 1991.

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

[17] John P. Granieri, Jonathan Crabtree, and Norman I. Badler. Production and playback of human figure motion for 3D virtual environments. In *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pages 127–135, March 1995.

[18] Lawrence J. Hettinger and Gary E. Riccio. Visually induced motion sickness in virtual environments. *Presence*, 1(3):306–310, Summer 1992.

[19] Christoph M. Hoffmann. How to construct the skeleton of CSG objects. In Adrian Bowyer and J. Davenport, editors, *The Mathematics of Surfaces IV.* Oxford University Press, Oxford, 1990. Available as Technical Report CSD-TR-1014, Computer Sciences Department, Purdue University.

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

[21] Philip M. Hubbard. *Collision Detection for Interactive Graphics Applications.* PhD thesis, Department of Computer Science, Brown University, October 1994. Available at ftp://ftp.cs.brown.edu/pub/techreports/95/cs95-08.ps.Z.

[22] Philip M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, September 1995.

[23] Philip M. Hubbard. Real-time collision detection and time-critical computing. In *Proceedings of the First ACM Workshop on Simulation and Interaction in Virtual Environments*, pages 92–96, July 1995.

[24] Hiroshi Inagaki, Kokichi Sugihara, and Noboru Sugie. Numerically robust incremental algorithm for constructing three-dimensional voronoi diagrams. In *Proceedings of the Fourth Canadian Conference on Computational Geometry*, pages 334–339, 1992.

[25] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In *Proceedings of SIGGRAPH '86,* published as *Computer Graphics*, 20(4):269–277, August 1986.

[26] Yoshifumi Kitamura, Haruo Takemura, Narendra Ahuja, and Fumio Kishino. Efficient collision detection among objects in arbitrary motion using multiple shape representations. In *Proceedings 12th IAPR International Conference on Pattern Recognition*, pages 390–396, October 1994.

[27] Ming C. Lin and John F. Canny. A fast algorithm for incremental distance calculation. In *Proceedings 1991 IEEE International Conference on Robotics and Automation*, pages 1008–1014, 1991.

[28] Yunhui Liu, Jiroshi Noborio, and Suguru Arimoto. Hierarchical sphere model (HSM) and its application for checking an interference between moving robots. In *Proceedings of the IEEE International Workshop on Intelligent Robots and Systems*, pages 801–806, 1988.

[29] Paulo W. C. Maciel and Peter 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.

[30] Martti Mäntylä and Markku Tamminen. Localized set operations for solid modeling. In *Proceedings of SIGGRAPH '83,* published as *Computer Graphics*, 17(3):279–287, July 1983.

[31] Brian Mirtich and John Canny. Impulse-based simulation of rigid bodies. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics (Monterey, California)*, pages 181–188, 1995.

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

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

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

[35] Randy Pausch, Thomas Crea, and Matthew Conway. A literature survey for virtual environments: Military flight simulator visual systems and simulator sickness. *Presence*, 1(3):344–363, Summer 1992.

[36] Madhav K. Ponamgi, Dinesh Manocha, and Ming C. Lin. Incremental algorithms for collision detection between solid models. In *Proceedings of the Third ACM Symposium on Solid Modeling and Applications*, pages 293–304, May 1995.

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

[38] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, England, 2nd edition, 1992.

[39] Jack Ritter. An efficient bounding sphere. In Andrew S. Glassner, editor, *Graphics Gems*, pages 301–303. Academic Press, Boston, Massachusetts, 1990.

[40] John Rohlf and James Helman. IRIS performer: A high performance multiprocessing toolkit for real-time 3D graphics. In *Proceedings of SIGGRAPH '94,* published as *Computer Graphics* Proceedings, Annual Conference Series, pages 381–394, July 1994.

[41] Hanan Samet and Markku Tamminen. Bintrees, CSG trees, and time. In *Proceedings of SIGGRAPH '85,* published as *Computer Graphics*, 19(3):121–130, July 1985.

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

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

[44] Andrew Smith, Yoshifumi Kitamura, Haruo Takemura, and Fumio 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, March 1995.

[45] John M. Snyder, Adam R. Woodbury, Kurt Fleischer, Bena Currin, and Alan 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, August 1993.

[46] William C. Thibault and Bruce 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.

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

[48] Greg Turk. Generating textures on arbitrary surfaces using reaction-diffusion. In *Proceedings of SIGGRAPH '91,* published as *Computer Graphics*, 25(4):289–298, August 1991.

[49] Andries van Dam. VR as a forcing function: Software implications of a new paradigm. In *Proceedings of the 1993 IEEE Symposium on Research Frontiers in Virtual Reality*, pages 5–8, October 1993.

[50] Brian Von Herzen, Alan H. Barr, and Harold R. Zatz. Geometric collisions for time-dependent parametric surfaces. In *Proceedings of SIGGRAPH '90,* published as *Computer Graphics*, 24(4):39–48, August 1990.

[51] Ji-Hoon Youn and K. Wohn. Real-time collision detection for virtual reality applications. In *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pages 415–421, September 1993.

[52] Michael J. Zyda, William D. Osborne, James G. Monahan, and David R. Pratt. NPSNET: Real-time vehicle collisions, explosions and terrain modifications. *The Journal of Visualization and Computer Animation*, 4(1):13–24, 1993.

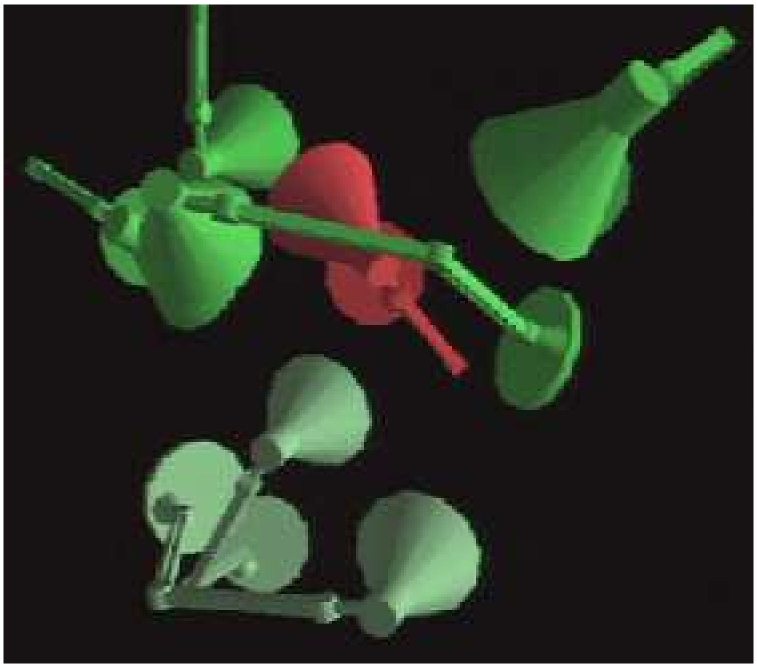


Figure 21: A frame from the run profiled in Figure 19. The light-colored lamps have just collided.

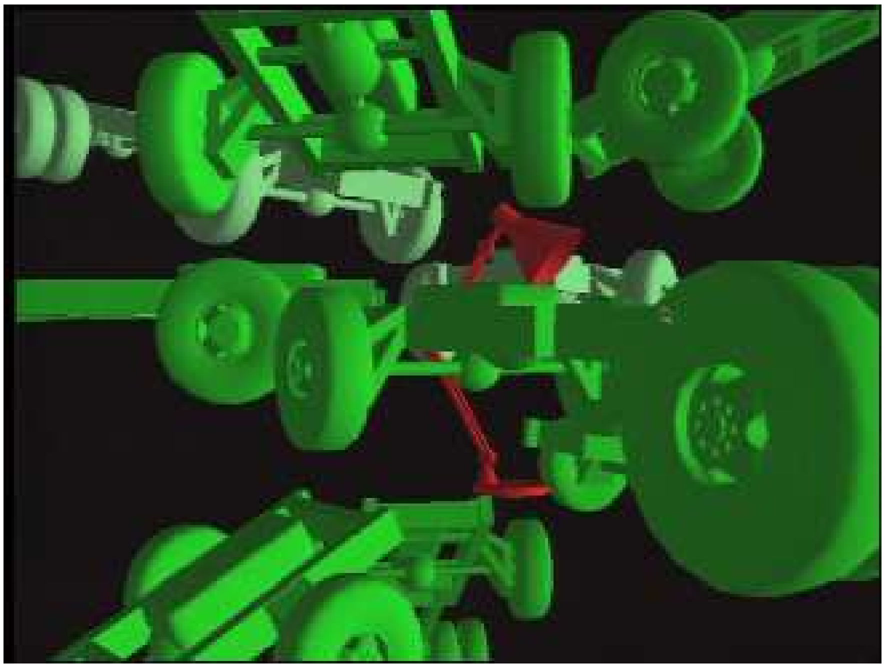


Figure 22: A frame from the run profiled in Figure 20. The light-colored trucks have just collided.