Collision Detection for Interactive Graphics Applications

Philip M. Hubbard

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-08
April 1995

Collision Detection for Interactive Graphics Applications

by
Philip Martyn Hubbard
A. B., Harvard University, 1988
Sc. M., Brown University, 1991

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University.

Submitted: October 1994
Revised: March 1995

Errata

Section 6.3 describes how to build a tree of spheres that approximate a polyhedron. One step, described in Section 6.3.2, takes a tree that roughly covers a polyhedron and makes the tree cover the polyhedron conservatively. This step should ensure that each set of children spheres at level j covers the part of the polyhedron covered by the parent sphere at level $j-1$. My implementation, however, used a different criteria. It ensured that for each level j, the union of all spheres at that level covers the polyhedron; the pseudocode from Figure 6.26 also uses this approach. Thus, a parent's nieces may cover what its children should cover. In theory, this oversight could produce trees that cause the collision-detection algorithm from Figure 5.4 to miss collisions. In the empirical tests from Section 7.2, though, this problem did not occur, as the algorithm from Figure 5.4 found every collision found by a reference algorithm.

Appendix B asserts that finding intersections between two-dimensional shapes (under mild restrictions) is asymptotically as efficient as finding intersections between the shapes bounding boxes. This conjecture still seems plausible, but the proof sketched in Appendix B does not cover all cases. Fortunately, this conjecture is merely a digression, and no results of this dissertation depend on it.

Abstract

Solid objects in the real world do not pass through each other when they collide. Enforcing this property of "solidness" is important in many interactive graphics applications; for example, solidness makes virtual reality more believable, and solidness is essential for the correctness of vehicle simulators. These applications use a *collision-detection algorithm* to enforce the solidness of objects. Unfortunately, previous collision-detection algorithms do not adequately address the needs of interactive applications. To work in these applications, a collision-detection algorithm must run at real-time rates, even when many objects can collide, and it must tolerate objects whose motion is specified "on the fly" by a user.

This dissertation describes a new collision-detection algorithm that meets these criteria through approximation and graceful degradation, elements of *time-critical computing*. The algorithm is not only fast but also interruptible, allowing an application to trade accuracy for speed as needed. The algorithm uses two forms of approximate geometry. The first is a four-dimensional structure called a *space-time bound*. This structure provides a conservative estimate of where an object may be in the immediate future based on estimates of the object's acceleration. The algorithm uses space-time bounds to focus its attention on objects that are likely to collide. The second form of approximate geometry is a *sphere-tree*. This structure contains a hierarchy of unions of spheres, each union approximating the three-dimensional surface of an object at a different level of detail. Sphere-trees allow the algorithm to quickly find approximate contacts between objects, and they allow the application to interrupt the algorithm in order to meet real-time performance goals.

Automatically building sphere-trees is an interesting problem in itself, and this dissertation describes several approaches. The simplest approach is based on octrees, and more sophisticated approaches use simulated annealing and approximate medial-axis surfaces. Several of the steps in these algorithms are themselves significant. One step is a simple algorithm for checking whether a union of two-dimensional shapes covers a polygon. Another step builds Voronoi diagrams for three-dimensional data points, and does so more robustly and accurately than previous algorithms.

An implementation of the collision-detection algorithm runs significantly faster than a previous algorithm in empirical tests. In particular, this implementation allows real-time performance in a sample application (a simple flight simulator) that is too slow with the previous algorithm; in some cases, the performance improves by more than two orders of magnitude. Experience with this sample application suggests that time-critical computing is not always simple to apply, but it provides enough benefits that it deserves further exploration in other contexts.

Acknowledgements

All the recent members of the Brown University Computer Graphics Group contributed to this work, whether they realize it or not. The group is intense and exciting, and it is difficult to spend much time in the group without feeling a permanent influence. Certain members of the group stand out for their particular importance to me. Henry Kaufman, Mike Natkin, and Paul Strauss helped me settle into the group during my first year. Brook Conner, Cindy Grimm, Ken Herndon, Nate Huang, Brian Knep, Tom Meyer, Dan Robbins, Scott Snibbe, Matthias Wloka and Bob Zeleznik spent countless hours sharing their ideas with me and helping me clarify my own. I particularly thank Scott for implementing the interactive system that let me play with two-dimensional Voronoi diagrams, and for modifying it to help me generate the diagrams that appear in this dissertation. Dan and Ken also deserve special mention for answering all my video questions. Lori Agresti helped with many administrative problems, and the group could not have functioned without her.

The Computer Science Department at Brown was as pleasant a setting for graduate studies as I could imagine. I thank the many people who contributed to the breadth of my education. Lloyd Greenwald and Kate Sanders, in particular, enriched my experience in many ways. I thank my early officemates, Cheryl Harkness, Richard Hughey and Rob Ravenscroft, for tolerating me while I adjusted to graduate school. My later officemates, Brook Conner, Larisa Matejic and Kate Sanders, shared the workstations generously and provided welcome camaraderie. I fondly remember playing with “the band,” and I appreciate all the people who played in it at various times: Brook Conner (vocals), Tony Davis (drums), Nate Huang (keyboards, violin, guitar, vocals), John “Spike” Hughes (guitar, vocals), Jeremy Katz (vocals, guitar), Henry Kaufman (percussion), Brian Knep (guitar, vocals), Tom Meyer (guitar), Eric Morse (guitar), Ramamurthy Ravi (vocals, guitar), and Rick Sabourin (saxophone). Intramural sports also helped keep me going, and I thank all the people, too numerous to mention, who played on various teams with me.

For giving me a home away from the department, I thank my Providence housemates through the years: Bruce Bauer, Sanji Balasuriya, Denise Chen, Jim Sauerberg, and Liddy Shriver.

People at several other schools made valuable contributions to my graduate studies. In particular, I thank Steve Drucker, Paul Dworkin and Tinsley Galyean from the Massachusetts Institute of Technology Media Laboratory, and Greg Turk and Terry Yoo from the University of North Carolina at Chapel Hill. Frederick P. Brooks, Jr., also at UNC, gave me much-needed encouragement, particularly at the beginning of this research. Stuart Semmel, from Harvard University, does not work in computer science but our discussions of scholarship and life in general have been most important.

My doctoral-committee members gave me important guidance in completing this research. Franco Preparata’s geometric insight inspired me at many times, and he was my constant companion through the book he coauthored. Jim Kajiya gently steered me away from some inappropriate ideas, and gave me confidence in the value of my other ideas.

Andy van Dam, founder of the Brown Graphics Group, was essentially a second advisor to me. I thank him for maintaining confidence in me as I learned though trial and error. Throughout my stay at Brown, I admired his determination, stamina and willingness to let anyone demonstrate their talents. His ability to anticipate important trends and articulate his visions are remarkable. I hope that some of these qualities have rubbed off on me.

This work would have been impossible without the help of my advisor, John “Spike” Hughes. Whenever I had a problem he was there to cheerfully offer sage advice. Our conversations on topics outside of mathematics and graphics were a welcome respite from the rigors of research, and I will always be grateful for the day he taught me the right way to play a ninth chord on the guitar. His approach to research and teaching, emphasizing intuitive approaches and clear explanations, is inspiring. He is an insightful, thorough and modest scholar, and graduate school will have been a success for me if I have learned to emulate even some of these virtues.

Finally, I would like to thank my parents, Paul and Sylvia, and my sister, Carol. They acted as my role models, and provided support and encouragement throughout my years in graduate school. Especially during the final weeks, I could not have finished without them.

Contents

List of Figures

Chapter 1

Introduction

An important goal in computer graphics is producing animations that believably reproduce the physical world. Many factors contribute to making an animation believable. The *geometric modeling* of three-dimensional (3D) shapes and the *rendering* of these 3D shapes into two-dimensional (2D) images must be realistic, and the computer graphics community has made many advances in these areas. An equally important factor is the *behavior* of objects as they move through the individual images or *frames* of an animation. Creating believable behavior still poses many research challenges.

One such important challenge involves collisions between objects. In most situations, an animation will seem more believable if the objects do not pass through each other like ghosts when they collide, that is, if objects exhibit "solidness." An application program that produces animations can require that human users enforce solidness manually. For example, the program might require that an artist explicitly script the motion of each object at each frame; in this case, the artist is responsible for knowing when objects will collide and what should happen after each collision.

In many contexts, however, manual enforcement of solidness is impractical or impossible. Manual enforcement does not work in an application that gives users high-level controls over object motion, controls that may affect what happens in more than one frame. A particular example of such an application is a *forward simulation*, in which users manipulate the physical properties of objects (e.g., mass, velocity, acceleration) and the application applies physical rules to determine how the objects move. These applications need a way of enforcing solidness automatically.

An application program can rely on a *collision-handling system* to enforce solidness. A collision-handling system typically consists of two components. The first component is a *collision-detection algorithm* (also known as a *detection algorithm*). It checks for objects that are starting to penetrate the surfaces of each other. If the detection algorithm finds any such objects, the collision-handling system calls its second component, a *collision-response algorithm* (also called a *response algorithm*). This algorithm corrects the behavior of the objects so that they no longer penetrate. The correction, for example, might involve making the object bounce off one another.

Both collision detection and collision response pose interesting research problems, but this dissertation focuses on collision detection. Such a focus is reasonable because the geometric issues involved in collision detection are interesting and challenging. For a thorough discussion of issues in collision response, see the work of Baraff [Bar89, Bar90, Bar91, BW92].

1.1 The Problem

An increasingly important class of application programs are those that produce *interactive* animations. In an interactive application, human users directly control the course of events and receive immediate feedback from the application. Contrast such an application with a "batch-processing" application, which waits for users to submit sequences of commands and produces no results until a complete sequence has been received. In many situations, users find the responsiveness of interactive applications more natural and comfortable.

Interactive applications share two particular characteristics. First, the application cannot know exactly what it will be doing in the future. This uncertainty is a result of human users determining the course of events of the application "on the fly." Second, the application must respond to users' actions at *real-time* rates. For an application producing an animation, an important element of real-time performance is maintaining a high and nearly-constant frame rate. Opinions vary on the minimum acceptable frame rate. Airey *et al.* [ARB90] state that a rate of six frames per second (fps) is barely tolerable for building-walkthrough applications, while Funkhouser and Séquin [FS93] suggest 10 fps; in general, Brooks [Bro88] and Fuchs and Bishop [FB92] suggest that the goal should be 20 to 30 fps. Another important aspect of real-time performance is low *latency* (also called *lag*). Latency is the time between when a user instigates an action and when the user sees that action manifested in the application. Fuchs and Bishop [FB92] state that the data on acceptable latency is currently only anecdotal, but they suggest that latency above 100 ms makes users uncomfortable and Friedmann *et al.* [FSP92] concur.

A variety of interactive applications need collision detection. One example is vehicle simulators, such as flight simulators and driving simulators [PM92]. These simulators are important because they train users to operate complicated vehicles without risking injury to the users or damage to the vehicles. Since users need to learn to operate the vehicles without crashing into other objects, collision detection is important for these simulators.

Another application is molecular modeling [Jan85, Bro88]. The effects of a drug molecule are determined, in part, by how it fits geometrically into a receptor site. Computer simulations that allow chemists to interactively test how molecules "dock" with each other are valuable tools in the design of new drugs. Detecting collisions between molecules is an important part of these simulations.

An increasingly popular class of applications is immersive virtual reality [FB92]. In these applications, the computer tries to convey to users the sense of being immersed in a new environment. Users may lose the sense of "really being there" if objects in the new environment behave in obviously unnatural ways. Objects that pass through each other will seem unnatural in most situations, so virtual reality applications need collision detection.

Researchers in graphics and robotics have published a variety of detection algorithms, but these algorithms cannot meet the demands of many interactive applications. The main problem is that the algorithms are too slow, making real-time performance impossible. As an example, Hahn [Hah88] describes a simulation system that uses a relatively sophisticated detection algorithm, an algorithm that people would still consider using today. Hahn reports the performance of this system both with and without collision detection. Adding collision detection increases the time to generate each frame by a factor of 47 on the average. Most interactive applications cannot tolerate this extra computational burden at each frame.

1.2 The Solution Strategy

This dissertation proposes a new approach to collision detection that addresses the needs of interactive applications. The key to this approach is the observation that for interactive applications, real-time performance is often at least as important as fully-accurate collision detection.

Consider the phenomenon of *simulator sickness*, which is also known as *visually induced motion sickness* or *cybersickness*. This feeling of discomfort, with symptoms resembling those of classical motion sickness, can plague users of vehicle simulators and virtual-reality applications. Simulator sickness is a complicated phenomenon and its causes are not fully understood, but evidence suggests that it is aggravated by low frame rates and high latency [Bio92, HR92, KLL$^+$92]. To alleviate the problem of simulator sickness, it is more important that an application program update frames quickly than that it detect all collisions with utmost accuracy.

Even when simulator sickness is not a problem, poor performance can limit the effectiveness of interactive applications. Forms of interaction that work well at real-time rates can be frustrating and confusing at slower speeds. Gossweiler [Gos93] observes that "if the feedback loop is too latent, control can oscillate and errors can compound (thus the paradigm shift from immersion to lockstep wait-and-see, much like trying to adjust bathtub shower temperatures)." Accurate collision detection is meaningless in an application that users find too difficult to control.

Fully-accurate collision detection may not always be necessary, even if there is time to compute it. Consider a simulator for a jet airplane. This airplane moves so quickly that a human user cannot control the simulation with pinpoint precision. Maneuvers specified by the user will not be accurate at a resolution finer than about half a meter, so it does not matter if collision detection involves a similar amount of inaccuracy. As another example, consider two complicated shapes (e.g., a trumpet and a trombone) colliding and bouncing off of each other. Many people would have difficulty predicting what motion should result from this collision, so they would not notice some inaccuracy in the collision detection. Users are also less likely to notice inaccuracy as animations become more crowded with moving objects.

These examples suggest the idea of *approximate* collision detection. The general idea is to detect collisions between approximate representations of the surfaces of objects. By using approximate representations that are faster to process than exact representations, the detection algorithm is able to trade accuracy for speed.

An important extension of this idea is *interruptible* collision detection. Computing a frame of an animation involves multiple tasks in addition to collision detection. Scheduling these tasks to meet performance goals is difficult. The scheduling problem becomes simpler if the application can interrupt a computation that starts to take too long. An interruptible detection algorithm can operate in the following manner. The algorithm first detects collisions at a coarse level of accuracy. If the application program allows it continue running, the algorithm refines the accuracy. By supporting several levels of refinement, the algorithm gives the application the flexibility to get as much accuracy as time allows.

The general idea of an interruptible computation that improves in quality with time appears by various names in other areas of graphics and computer science. Bergman *et al.* [BFGS86] introduced this idea in image rendering, calling it *adaptive refinement*. Cohen *et al.* [CCWG88] called this idea *progressive refinement* in the context of another approach to rendering. Dean and Boddy [DB88] first applied this idea to planning problems, calling the implementation an *anytime algorithm*. Van Dam [van93] uses the term *negotiated graceful degradation* for this class of techniques, and considers them to be part of the broader topic of *time-critical computing*. These ideas are important because they put the application in control of its own performance, freeing it from slavery to its subroutines. Informal user studies, conducted by Meyer and Globus [MG93] for a time-critical approach to visualizing scientific data, indicate that users appreciate being able to trade accuracy for speed. The success of these related ideas suggests that interruptible collision detection will also prove beneficial.

Approximation in collision detection and approximation in rendering can have fundamentally different effects. In rendering, inaccuracies due to approximation are not cumulative; if a rendering algorithm uses approximation in one frame, it can return to using its utmost accuracy in the next frame and the inaccuracies will disappear. Collision detection is different, in that a collision between two objects at one moment can influence the behavior of those objects from that moment forward. As an extreme example, consider a simulation of billiards. A complicated shot by an expert player may involve a series of collisions that ultimately put a ball in a pocket; if the simulation approximates any of these collisions the ball may end up nowhere near the pocket. This billiards simulation is an example of an application for which utmost accuracy is essential. Interruptible collision detection is not appropriate for these applications. A premise of this dissertation is that many applications, particularly interactive applications, cannot concentrate on accuracy alone but must balance it with performance. It is interesting to note that the class of applications that must balance speed and accuracy includes such demanding tasks as dynamic simulation of macromolecular systems, for which the popular CHARMM system [BBO+83] supports a variety of approximations even though their effects are also cumulative.

Few time-critical graphics algorithms can avoid occasional decreases in frame rate or increases in latency. The work in this dissertation is no exception. Fortunately, users do not seem to notice these fluctuations as long as they are small and rare. In this respect, the real-time response users require from interactive graphics systems imposes "soft" deadlines. This dissertation does not consider the more difficult topic of "hard" deadlines.

1.3 Contributions

The most direct contribution of this dissertation is a new approach to collision detection. This approach is an advance because it allows real-time performance in applications that could not achieve this goal using previous detection algorithms. Empirical studies show that in some situations, this approach can improve the performance of an application by more than two orders of magnitude.

In the course of developing the new detection algorithm, this dissertation describes several new results in geometric algorithms. One result is a simple way of detecting whether a union of shapes cover a polygon, an idea which can be extended to more general covering problems (Section 5.3.3). Another result is an improved algorithm for approximating the *medial-axis surface* [Blu67, NP85] of a 3D polyhedron (Chapter 6). The medial-axis surface gives a summary of the shape of an object, and has potential applications in pattern recognition and computer vision as well as computer graphics. The algorithm for building the medial-axis surface first invokes a straightforward method for distributing sample points uniformly over the surface of a polyhedron (Section 6.2.2). The algorithm then uses an improved algorithm for building a *Voronoi diagram* [PS85] for 3D data (Section 6.2.3). A Voronoi diagram is a classic structure from computational geometry that captures many useful proximity properties within a set of points. Accuracy and robustness are important issues in practical implementations of Voronoi diagrams, and the improved algorithm has advantages over previous algorithms in these respects.

A final contribution of this dissertation is its exploration of time-critical computation. This topic is becoming increasingly important in graphics applications, and it has the potential to become important throughout computer science. Since the topic is still relatively new, examples such as those in this dissertation are valuable because they suggest the important issues and demonstrate the possible benefits.

1.4 Overview

The next chapter describes related work on collision detection and time-critical computing. An important part of this chapter is a classification of three algorithmic weaknesses that can hurt performance in collision detection. Chapter 3 discusses what time-critical computing means in the context of collision detection. It introduces the mechanisms which are the focus of the remainder of this dissertation.

The first of those mechanisms, the *space-time bound*, is the subject of Chapter 4. A space-time bound uses approximate four-dimensional geometry to accelerate one phase of the detection algorithm. The second mechanism, the *sphere-tree*, is covered in Chapters 5 and 6. A sphere-tree supports the graceful degradation that is at the heart of time-critical collision detection. The latter of these chapters also discusses medial-axis surfaces and how they relate to sphere-trees. For a preview of sphere-trees, see Figures 6.27, 6.28 and 6.29, at the end of Chapter 6.

Chapter 7 analyzes the performance of the detection algorithm. The focus in this chapter is on empirical studies. Chapter 8 summarizes this dissertation and presents some possible extensions.

Chapter 2

Related Work

The research literature from computer graphics and robotics contains a wide variety of approaches to collision detection. This chapter summarizes the advantages and disadvantages of the prominent approaches. After discussing the history of collision detection, this chapter moves to the topic of time-critical computing. The summary of this topic focuses on the work most directly relevant to graphics applications.

2.1 A Basic Algorithm

One way to understand the challenges in collision detection is to examine a basic detection algorithm, to understand why it often performs poorly in practice. Figure 2.1 presents pseudocode for a basic detection algorithm and a simple application which calls it.

The application simulates the motion of N objects, $O_0, \ldots, O_{N-1}$, to produce an animation. The application has its own internal sense of *simulation time*, which runs from t_0 to t_1. Depending on the semantics of the application, this time may or may not correspond to *wallclock time*, the sense of time perceived by a human observer. For example, say the application is simulating the smashing of atoms in a particle accelerator. The period between t_0 and t_1 is extremely short for this simulation, much less than one second. A human audience will be able to appreciate the animation of this simulation only if its duration is significantly longer. The application achieves this end by making a unit of simulation time correspond to many units of wallclock time.

The application renders a frame of the animation every Δt_r simulation-time units, where Δt_r is the *rendering timestep*. To render a frame, the application first gets input from the user and updates the behavior of each object (see lines 4 and 5 of Figure 2.1); the details of what is involved in these steps are not important here. Next, the application calls the detection algorithm to determine if the updated behavior of any objects cause them to collide (line 8). If so, the application invokes a collision-response algorithm to correct the behavior (line 10). The application must actually invoke collision detection and response repeatedly (lines 7 to 11), to ensure that it handles all the collisions in the simulation-time period from the last frame to the current frame. Finally, the application renders all the objects (line 12) and advances to the next frame.

The basic detection algorithm operates in the following manner. When it is called at simulation time t_{curr}, it examines what has happened since t_{prev}, the simulation time at which it most recently

```
1   application()
2   {
3       for t ← t_0 to t_1 in steps of Δt_r {
4           get user input
5           update behavior of each object in {O_0, ... O_{N-1}} as of t
6           update any other general bookkeeping
7           do {
8               result ← detect(t, {O_0, ..., O_{N-1}})
9               if (result contains collisions)
10                  collision response
11          } while (result contains collisions)
12          render each object in {O_0, ... O_{N-1}} as of t
13      }
14  }

    /* The variable t_prev persists between calls. */
    /* Before first call, t_prev is initialized to t_0. */
15  detect(t_curr, {O_0, ..., O_{N-1}})
16  {
17      for t ← t_prev to t_curr in steps of Δt_d {
18          move each object in {O_0, ..., O_{N-1}} to its position as of t
19          for each object O_i ∈ {O_0, ..., O_{N-1}} {
20              for each object O_j ∈ {O_i, ..., O_{N-1}}
21                  if (pair-processing algorithm says O_i, O_j intersect)
22                      add O_i, O_j to collisions
23          }
24          if (collisions exist) {
25              t_prev ← t
26              return (collisions, t)
27          }
28      }
29      t_prev ← t_curr
30      return (no collisions, t_curr)
31  }
```

Figure 2.1: A basic detection algorithm for a simple application.

stopped detecting collisions (e.g., the time of the previous frame). Specifically, it samples the interval between t_{prev} and t_{curr} every Δt_d units (see line 17). The *detection timestep*, Δt_d, is distinct from the rendering timestep, Δt_r, used by the application. The code in Figure 2.1 implicitly assumes that Δt_r is an integer multiple of Δt_d. Few authors discuss the relationship between Δt_r and Δt_d. Hahn [Hah88] argues that the timesteps can be the same if they are small enough, more specifically, if there are at least 24 to 30 timesteps per second of wallclock time. Allowing Δt_d to be smaller than Δt_r is more flexible, however, in that this policy maintains some accuracy in the motion of objects even if the application cannot render that motion so accurately. At each sample t between t_{prev} and t_{curr}, the detection algorithm finds the position of each object as of t (line 18). The algorithm then cycles through all pairs of objects (lines 19 and 20) and tests each pair to see if their surfaces intersect (line 21). The steps involved in this last test constitute the *pair-processing algorithm*. If the pair-processing algorithm finds intersection, the detection algorithm adds the current pair of objects to the list of colliding pairs, which it returns when it has tested all the pairs. If it gets through the simulation-time interval from t_{prev} to t_{curr} without finding intersection, the detection algorithm returns that no collisions occurred. In either case, the algorithm sets t_{prev} to be the simulation time at which it stopped processing (lines 25 and 29) so it can resume processing at that time when it is next called (t_{prev} persists between calls to the algorithm).

This basic detection algorithm does not include a few features found in some real detection algorithms. First, the basic algorithm has no facility for *back-tracking*. Back-tracking is binary search on the simulation-time interval from t_{prev} to t_{curr} in order to more accurately determine the moment of contact between two objects. Back-tracking gets it name because it involves decrementing t to go backwards in simulation time in addition to incrementing t (as the basic algorithm does). A detection algorithm that uses back-tracking will be more accurate, but it will also be slower. For many interactive applications, the overhead in back-tracking is not worthwhile. A second feature not present in the basic algorithm is the ability to handle non-pairwise collisions. If three objects—O_a, O_b and O_c—all collide simultaneously, the algorithm will report separate collisions between O_a and O_b, O_a and O_c, and O_b and O_c. For most forms of collision response this approach will not be a problem.

2.2 Weaknesses of the Basic Algorithm

This basic detection algorithm has three weaknesses. The first weakness is in the outermost loop of the detection algorithm (starting at line 17). This outer loop breaks simulation time down into fixed timesteps of length Δt_d. The disadvantage is that choosing the size of Δt_d poses a dilemma. On the one hand, a smaller Δt_d improves the accuracy of collision detection. Consider Figure 2.2. This figure shows two objects, a circle and a square, moving towards each other along the x axis. The figure also includes an explicit simulation-time axis to illustrate how the objects move over time. The time axis is labeled with one possible choice of fixed timesteps. Notice that there are no collisions at any of the timesteps, but there is a collision that the algorithm will miss because it falls between timesteps. By using a smaller timestep the algorithm decreases the chance of this sort of error. On the other hand, the algorithm is more efficient if it uses a larger timestep. The algorithm does work at each timestep, so fewer, larger timesteps mean less work overall. A way to escape this dilemma is to make Δt_d be an *adaptive timestep* that changes according to the situation: Δt_d should become smaller when collisions are likely and larger when they are not. A detection algorithm's inability to adaptively change its timestep is the *fixed-timestep weakness*.

The second weakness of the basic algorithm is the presence of the two loops starting at line 19. These loops cycle through every pair of objects at every timestep, making the processing time spent by the algorithm increase quadratically with N, the number of objects. As users demand more detailed applications, the typical values of N will increase, hurting the performance of the algorithm. The algorithm will be more efficient if it can prune the number of pairs that it must process, to avoid processing pair of objects that are obviously far away from each other. Lacking such an optimization, the need to check every pair of objects at every timestep is the *all-pairs weakness*.

The third weakness involves the test at line 21. The subroutine that determines if the surfaces of two objects intersect as of a particular moment in simulation time is the pair-processing algorithm. Designing a pair-processing algorithm that is both efficient and accurate is a difficult problem. The difficulty of this task depends on how the algorithm represents the surface of each object. A

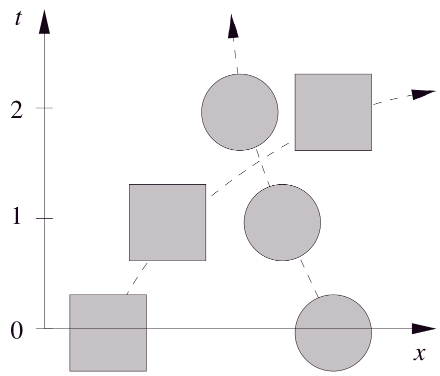


Figure 2.2: The basic detection algorithm can miss this collision.

popular representation is the polyhedron. The test for polyhedral intersection is quite complicated; all the different ways vertices, edges and faces can contact one another produce many special cases which are difficult to process efficiently. Other forms of surface representation also pose problems. Curved surfaces, such as parametric bicubic surfaces [FvFH90], can be even more difficult to process, and pair-processing algorithms for these surfaces often resort to approximating the surfaces with polyhedra. A detection algorithm suffers from the *pair-processing weakness* if its pair-processing algorithm is not robust and efficient.

2.3 Improvements on the Basic Algorithm

All the previous work on collision detection can be seen as efforts to address the weaknesses of the basic algorithm. Most of the published papers offer clever ideas that mitigate one or two of the weaknesses. For some applications this work is appropriate, because not all weaknesses matter in all applications; applications involving only a few objects are not affected by the all-pairs weakness, and applications involving many objects that collide almost all the time are not affected by the fixed-timestep weakness. In general, however, all three weaknesses do matter, and an algorithm should address as many of them as possible. A few algorithms do come close to meeting this goal, but all of them have disadvantages for interactive applications.

2.3.1 Four-Dimensional Methods

To address the fixed-timestep weakness, many authors use four-dimensional (4D) analysis, where the fourth dimension explicitly represents simulation time. One of the earliest examples of this idea is the work of Boyse [Boy79]. He derives simple equations for the motion of polyhedral edges and faces over time; a root of any such equation corresponds to the time and location of a collision. By assuming only one stationary and one moving polyhedron, with the moving polyhedron being limited to periods of only linear translation and periods of only rotation around a fixed axis, Boyse's algorithm can find the collisions without any iterative timestep. The assumptions of the algorithm limit its utility in practice, however. Lubachevsky [Lub91] presents a detailed description of a similar idea. His approach makes even more assumptions, though, limiting itself to spheres moving in straight lines on a plane ("pool balls").

Canny [Can86] uses more elaborate analysis to process more general motion. He obtains quintic polynomials whose roots, found with iterative numerical techniques, describe the collisions between convex polyhedra. The motion this algorithm can handle is still not completely general, however; objects must have constant linear velocity and nearly-constant angular velocity. For interactive applications, the main disadvantage of Canny's algorithm is the disadvantage shared by most 4D methods: the motion of all objects must be completely specified throughout simulation time. Interactive applications cannot meet this requirement, because a human user controls the motion of objects "on the fly."

Cameron [Cam90] presents an algorithm that handles more complicated shapes than Canny's approach, in particular, objects specified by the constructive solid geometry (CSG) paradigm. This algorithm is a 4D extension of earlier work [Cam89], which addresses the pair-processing weakness for CSG objects by using a technique called *S-bounds*. Viewing CSG models as trees of operations, S-bounds provide a way to prune nodes that do not matter for a possible collision between two trees. The 4D extension to S-bounds addresses the fixed-timestep weakness, but only for piecewise-linear motion that is fully specified throughout time.

To detect collisions between arbitrary time-varying parametric surfaces, Von Herzen *et al.* [VBZ90] use the mathematical technique of the *Lipshitz condition*. A Lipshitz condition maps a change in the parameters of a surface to an upper bound on the change in the surface itself. By applying this mapping while searching the parameter spaces of two objects, the algorithm can detect collisions between the objects. This approach addresses the fixed-timestep weakness because the search of the simulation-time parameter is analogous to binary search. This approach ignores the pair-processing weakness, however, and once again it assumes knowledge of how the surface changes throughout time.

Samet and Tamminen [ST85] describe a 4D approach based on recursive subdivision. The algorithm considers "cubes" of space and time, and subdivides the cubes that contain two objects. The objects in this context are represented by 4D geometry, defined by how pieces of the object's surfaces move through time. If the objects do collide, repeated subdivision rapidly locates a small cube in which the collision occurs. This algorithm processes only two objects, so it ignores the pair-processing weakness. Samet and Tamminen also give details for only the case in which the two objects have piecewise-linear motion. For more general motion, the authors suggest an approach based on interval analysis.

Mudur and Koparkar [MK84] describe one of the first applications of *interval analysis* to graphics, but the work of Snyder [Sny92], Duff [Duf92] and Snyder *et al.* [SWF$^+$93] is more relevant for collision detection. Interval analysis involves extending mathematical functions to take intervals as arguments and produce intervals as results. A detection algorithm uses functions whose argument intervals represent regions in space and time. If any objects collide within these regions, the functions return intervals that contain designated values. The detection algorithm uses a form of recursive subdivision on the argument intervals to locate regions that contain collisions. Snyder *et al.* [SWF$^+$93] describe functions that represent constraints on each pair of objects, constraints which are met only when the pair has just started to collide. To accelerate the recursive subdivision, the Snyder *et al.* algorithm applies an interval version of the Newton-Raphson method, a classic technique for finding roots of a function [PTVF92]. Duff [Duf92] uses simpler functions than Snyder *et al.*, and his algorithm does not exploit the interval Newton-Raphson method. His algorithm does not need a distinct function for each pair of objects, though, so it avoids the all-pairs weakness. Duff's algorithm is thus one of the few published algorithm that addresses all three weaknesses of the basic algorithm. Unfortunately, the algorithm shares the disadvantage of 4D analysis, that it must know each object's exact position throughout time.

Culley and Kempf [CK86] attack the fixed-timestep weakness by making the following observation: an upper bound on the relative velocity of two objects and a lower bound on their separation distance define a lower bound on the time before they can collide. An algorithm that uses this lower bound on collision time can adaptively vary its timestep. Note that unlike the other 4D methods, this approach does not require full knowledge of the object's motion ahead of time. Culley and Kempf do not describe efficient ways to make this approach work for nonlinear motion or more than two objects, but the general idea was a motivation for Chapter 4 of this dissertation.

Foisy *et al.* [FHA90] developed ideas somewhat similar to those in Chapter 4 independently. An important part of both approaches is 4D analysis based on estimates of objects' maximum accelerations. These approaches address the fixed-timestep weakness without requiring full knowledge of objects' motions throughout time. The details of the approaches differ significantly, however. The Foisy *et al.* algorithm uses a queueing scheme that allows it to compare only $O(N \log N)$ objects at each timestep. When the frequency and number of possible collisions meet certain conditions, this scheme solves the all-pairs weakness. If these conditions are not met, however, the algorithm can fail to detect some collisions; to guarantee correctness in this case, the algorithm must call some other detection algorithm.

2.3.2 Three-Dimensional Subdivision Methods

Many authors focus more on the all-pairs and pair-processing weaknesses than on the fixed-timestep weakness. A common approach is to subdivide 3D space into regions. A pair of objects must intersect the same region in order to collide with each other, so the detection algorithm can use the regions to reduce the number of pairs it must consider.

The simplest subdivision technique partitions 3D space along a uniform grid. Zyda *et al.* [ZOMP93] describe an implementation of this idea. The cells of the grid are *isothetic*[1] cubes. The axis-aligned nature of isothetic cubes allows the algorithm to quickly determine whether an object belongs in a cell.

Turk [Tur90b] uses a similar grid, and he describes two approaches for determining which objects belong in each cell. His application is a molecular-docking simulation, in which molecules are represented as collections of spheres. The width of each cubical grid cell is the diameter of the largest sphere, so any sphere can intersect at most eight cells; it is straightforward to determine these cells given the cell that contains the sphere's center. One way to implement this grid is to find the box that bounds all the spheres and then divide it into cells. Turk reports that this approach works no better than a simpler approach. The simpler approach uses spatial hashing, that is, it creates a fixed grid of cells and then uses a hash function to map any position in $\mathbb{R}^3$ to a cell in that grid. The simplest hashing function effectively tiles all of $\mathbb{R}^3$ with the fixed grid, so objects that are actually quite far from each other can sometimes occupy the same cell. In Turk's application, this problem did not adversely affect performance. This approach does not address the fixed-timestep or pair-processing weaknesses, but it has low enough overhead and prevents the all-pairs weakness well enough that it could be part of an effective detection algorithm for an interactive application. In this context, it would quickly identify pairs of objects whose bounding spheres collide; a pair-processing algorithm would then check those objects for actual collisions. Chapter 7 discusses the performance of this approach in practice.

A more sophisticated approach to space subdivision uses an *octree*, which acts like a hierarchy of grids at different resolutions. The root of an octree is an isothetic box that bounds all the objects.

[1] *Isothetic* means "aligned with the principal coordinate axes of the space." A square in $\mathbb{R}^n$ is isothetic if each of its edges is parallel to one of the coordinate axes of $\mathbb{R}^n$, and a cube is isothetic if each of its faces is an isothetic square.

The children of the root are the eight octants defined by dividing each edge of the box in half. The algorithm that performs this dividing also determines which objects belong in each octant. If an octant contains only a few objects, the detection algorithm can efficiently check pairs of those objects for collisions. If an octant contains many objects, the algorithm recursively subdivides that octant into smaller octants. This approach adaptively refines the resolution of the grid, making the cells smaller only where necessary. Hahn [Hah88], Moore and Wilhelms [MW88] and Shaffer and Herb [SH92] describe variations on this approach for collision detection. These approaches address the all-pairs weakness. Octrees also address the pair-processing weakness, as Moore and Wilhelms describe most clearly. The idea is to continue subdividing the space shared by two objects, keeping track of the parts of the objects' surfaces in each cell. If the objects have polyhedral surfaces, this approach efficiently identifies the polygons that can intersect each other. One of the earliest examples of this idea was the work of Mäntylä and Tamminen [MT83], which involves a structure that is not precisely an octree but shares many of its properties.

Octrees share the disadvantage of Turk's algorithm, that they do not address the fixed-timestep weakness. Unlike Turk's algorithm, however, octrees do not necessarily have low overhead. Moore and Wilhelm's algorithm rebuilds the octree each time it is called, and Shaffer and Herb's algorithm modifies its octree to reassign objects that have moved out of their previous cells. Either approach involves considerable computation, particularly for applications involving many moving objects; the simplicity of Turk's algorithm is likely to make it faster in many situations.

Binary space partitioning (BSP) trees provide a different form of space subdivision. These structures were introduced by Fuchs *et al.* [FKN80] as a way to remove hidden surfaces during polygon rendering. Thibault and Naylor [TN87] discuss how to use BSP trees to perform Boolean set operations on two polyhedra. Their algorithm finds intersections, so it can function as a pair-processing algorithm for collision detection. Each node of a BSP tree corresponds to the plane containing one face of a polyhedron. For the root node, this plane partitions the other faces into three sets: those in front of the plane, those behind it, and those intersecting it. For each of the first two sets, picking a face from the set to function as another partitioner defines a subtree of the root. Each face in the third set belongs in both subtrees. Repeating this process recursively defines the BSP tree for the polyhedron. Note that if some query polygon is fully on one side of the plane at the root, then that polygon cannot intersect any of the faces in one subtree. This idea leads to an efficient algorithm for finding intersections between two polyhedra given a BSP tree for one of them. This algorithm does not address the fixed-timestep or all-pairs weaknesses, but it works well as a pair-processing subroutine in a larger detection algorithm. Chapter 7 discusses the performance of this approach in practice, comparing it to the new algorithm from this dissertation.

Vaněček [Van91] extends BSP trees to form *multidimensional space partitioning* (MSP) trees, which he also calls *Brep-indices*. These structures represent the volume enclosed by polyhedra more conveniently than BSP trees. Vaněček describes an algorithm that intersects a line segment with a Brep-index, identifying the parts of the segment that are inside a polyhedron. Features of Brep-indices allow this algorithm to tolerate inaccuracy in floating-point operations. Bouma and Vaněček [BV91] use this algorithm to detect collisions between pairs of polyhedra. This approach could provide benefits over a pair-processing algorithm based on BSP trees, but I chose to use BSP trees in Chapter 7 for simplicity of implementation.

2.3.3 Other Methods

To detect collisions between two convex polyhedra, Dobkin and Kirkpatrick [DK83] present an algorithm that requires $O(log^2 n)$ operations, with n being the sum of the number of vertices in the two polyhedra. This algorithm uses the decomposition of each polyhedron into *drums*, which are slabs cut from the polyhedron by parallel planes. The simple structure of drums makes them similar to two-dimensional polygons, so an extension of an algorithm for polygons can find intersections between two drums. For convex polyhedra, a form of binary search efficiently determines pairs of drums to be tested for intersection. This approach could, in theory, detect collisions between two nonconvex polyhedra given a decomposition of these polyhedra into convex pieces. The number of pairs of these pieces could be overwhelming for objects with complicated shapes, however.

Lin and Canny [LC91] describe an efficient algorithm for finding the closest distance between two convex polyhedra. The algorithm is based on the following observation: when two convex polyhedra move only a small amount relative to each other, the points on each polyhedron closest to the other polyhedron also move only a small amount. Since the polyhedra are convex, it is not difficult to track these small movements of the closest points. Knowing the closest point between two convex objects is sufficient for detecting collisions between the objects, so this algorithm is a detection algorithm that solves the pair-processing weakness for convex objects.

Lin, Canny and Manocha [LMC93] extend this idea to handle more than two nonconvex objects. The algorithm assumes each nonconvex object is partitioned into convex pieces, and builds a tree whose leaves are the convex pieces. The interior nodes are convex hulls of unions of the pieces. The algorithm descends the tree, applying the Lin-Canny algorithm to each of the convex hulls. To handle multiple objects, the algorithm uses as queueing scheme much like the one described by Foisy *et al.* [FHA90]. These two extensions can mitigate the fixed-timestep and all-pairs weaknesses. There are situations in which these extensions would perform poorly, however, and the authors do not discuss their algorithm's actual performance in practice.

A different extension of the Lin-Canny algorithm is presented by Dworkin [Dwo93]. He augments the algorithm with four-dimensional techniques similar to Lubachevsky [Lub91] and Duff [Duf92]. Given certain assumptions about the sparseness of objects and the motion they can have, this approach yields high performance.

Sclaroff and Pentland [SP91] describe a novel solution to the pair-processing weakness. They suggest that polyhedral models be approximated by deformed superquadric ellipsoids [Bar81] whose surfaces have been modulated by a *displacement map*. These approximate models, which they call *generalized implicit functions*, have the advantage that the algorithm to detect collisions between them runs quickly. The performance figures quoted by the authors suggest that their algorithm is efficient for the class of models that can be represented by their technique. They do not characterize the members of this class, but it cannot contain some interesting shapes due to details of the technique: the displacement map cannot match any feature that is not *star-shaped* [PS85] about a point on the deformed ellipsoid, and the superquadric ellipsoid cannot match any shape with non-spherical topology. The idea of approximation is a promising one, however, and this paper is one of the first to show its practical performance benefits.

2.4 Time-Critical Computing

Time-critical computing is related to the topic of real-time systems. Halang and Sacha [HS92] indicate that these systems are important in applications like industrial process control and aviation. These systems tend to be hard-wired to meet the performance goals of specific applications, however. Halang and Sacha suggest that there are few commercial operating systems that support a variety of general real-time applications, and the support that is available is at a relatively low level.

A seminal work on time-critical computing in a more general context is the paper by Dean and

Boddy [DB88]. They discuss a decision-theoretic framework for time-critical planning, useful for applications such as robotics. They prove that this framework picks the optimal schedule of subtasks, given subtask algorithms that increase their utility with time according to specific patterns. This paper also coins the term *anytime algorithms*, which has seen wide currency since then. It is an open question whether this decision-theoretic framework is valuable for interactive graphics applications.

Computer graphics has a history of dedicated systems that provide real-time performance for specific applications, particularly vehicle simulation. Schacter [Sch81] indicates that flight simulators have been using time-critical rendering for years. These systems represent the shapes of obstacles on the ground and of other planes at various levels of detail, that is, with various numbers of polygons. The rendering components of these systems choose the appropriate number of polygons to keep the systems running at real-time rates. The shapes used by a given simulator are generally fixed, simplifying the task of building the levels of detail.

Bergman *et al.* [BFGS86] describe one of the first time-critical rendering algorithms used outside of flight simulators. This algorithm quickly presents a crude image while a user is changing the viewpoint, and then adaptively refines this image when the user pauses (to study some feature of the image). The crude image shows only the vertices of the visible objects. To refine the image, the algorithm adds polyhedral edges, flat-shaded polygons, improved shading for the polygons that need it, and anti-aliasing. Funkhouser and Séquin [FS93] present a more advanced approach. Their algorithm uses a simple, empirical model to predict the cost of rendering objects at various levels of detail. The algorithm also has a heuristic model for the benefits of rendering at each level. When it renders each frame, the algorithm uses a greedy approach to maximize the benefits of rendering while meeting a cost goal, that is, keeping the frame rate high and nearly-constant.

This algorithm assumes that it has access to the levels of detail for each object. Clark [Cla76] presents one of the earliest discussions of levels of detail, but gives no specific techniques for generating the levels. More recently, authors such as Turk [Tur92] and Schroeder *et al.* [SZL92] have described algorithms that produce these levels of detail automatically. Hoppe *et al.* [HDD$^+$93] describe what seems to be the most sophisticated approach to date. The algorithm generates each level of detail by building an approximation to the original polyhedron. It builds the approximation by solving an energy-minimization problem. The energy has terms that increase with the number of polygonal vertices and with the looseness with which the approximation fits the original. To minimize this energy, the algorithm alternates between modifying the geometry and the topology of the approximation. Hoppe *et al.* report promising results from this algorithm, so it could make time-critical rendering algorithms, like the one by Funkhouser and Séquin, more practical.

Time-critical algorithms for tasks beyond polygon rendering are starting to appear. Meyer and Globus [MG93] consider the problem of visualizing scalar field data, for example, data from computational fluid dynamics simulations. One way to visualize such data is to display an isosurface, a surface that connects locations having a particular scalar value. Meyer and Globus describe an algorithm that quickly generates the isosurface at a location of interest to the user; the algorithm then propagates the isosurface to adjacent locations until the user specifies a new location of interest. This approach preserves interactive response to changes in the location of interest, and informal user studies suggest that people find this approach helpful when exploring data.

For the problem of time-critical collision detection, the previous work on other time-critical algorithms does not provide much more than inspiration. The previous work is complementary, however, in the sense that these other algorithms would work well along side a time-critical detection algorithm in a complete application.

An application that uses multiple time-critical algorithms will be simpler to implement in a software framework that explicitly supports such algorithms. Rohlf and Helman [RH94] describe one of the first commercial software packages to strive for this goal. The system supports scene data structures that explicitly represent the geometry of objects at multiple levels of detail. The system also provides several flexible ways of synchronizing activities with frames that occur at regular time intervals. Missing from the system are advanced features, such as automatic ways to generate levels of detail and techniques of feedback or prediction to prevent activities from exceeding the time allotted per frame, but the system provides more support for real-time performance than previous systems. Wloka [Wlo93] proposes a more ambitious system. The design of this system emphasizes predictive scheduling to ensure real-time, low-latency response to users' commands. The availability of systems such as this one should make possible more widespread application of time-critical ideas.

Chapter 3

Overview of Time-Critical Collision Detection

Section 1.2 introduced the idea behind time-critical computing, that it is sometimes valuable to trade accuracy for speed. Section 2.1 described how an application program and a detection algorithm cooperate, but it did not describe how this process can be time-critical. This chapter gives an overview of a time-critical algorithm for collision detection. Chapters 4, 5 and 6 will describe in detail the mechanisms involved in the time-critical detection algorithm.

Recall from Section 2.1 that before rendering each frame of an animation, the application program calls the detection algorithm. The detection algorithm is responsible for finding collisions in the period from t_{prev}, the previous simulation time at which it stopped detecting, to t_{curr}, the current simulation time. As part of this process, the pair-processing algorithm tests pairs of object surfaces for intersection.

A modification to this process involves the pair-processing algorithm testing not the actual surfaces but *conservative approximations* to the surfaces. An approximate surface is conservative if it fully covers the actual surface, leaving no parts uncovered. Note that every collision between a pair of actual surfaces will also be a collision between the conservative approximations, but the converse is not true. Thus, this new version of the pair-processing algorithm still answers the question, "Does this pair of objects intersect?" However, it no longer returns a "yes/no" answer; it now returns a "maybe/no" answer. Figure 3.1 gives an example of the difference between these two approaches.

Subsequent chapters of this dissertation will discuss the issue of conservative approximation more fully, but for now assume that the pair-processing algorithm can test approximate surfaces for intersection more quickly than it can test actual surfaces. The new pair-processing algorithm thus allows the detection algorithm to quickly return a "maybe/no" answer when the application asks about collisions. If the detection algorithm returns "no" then the application can go ahead and render the current frame. If, on the other hand, the detection algorithm returns "maybe" then the application has some options. The application might determine that it cannot afford to devote any more processing time to collision detection. In this case, the application must make do with the answer of "maybe." The conservative policy is to treat "maybe" as "yes" and act as if the detection algorithm found a collision between the actual surfaces of the objects. The collision-response algorithm will produce an approximation to the correct behavior since it is processing a collision between approximate surfaces, but in many situations, approximate behavior that meets

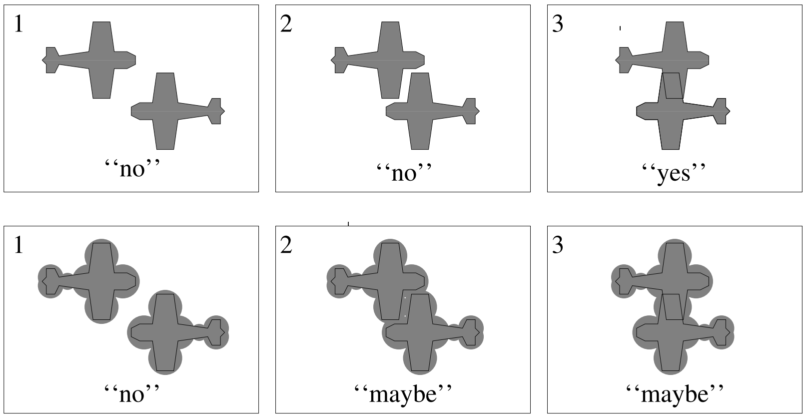


Figure 3.1: The upper three frames show "yes/no" answers; the lower three frames show the same motion with "maybe/no" answers.

performance goals will be better than correct behavior that lags behind those goals.

The application, having received "maybe" from the detection algorithm, might determine that it does have additional processing time for collision detection. In this case, the application can tell the detection algorithm to refine the details of the approximate collision. The detection algorithm can make this refinement if it can approximate the actual surface of each object at multiple *levels of detail*. Each level of detail should improve the accuracy of the previous level, and the levels should be arranged so that information about a collision at one level can be reused to detect a collision at the next level. Given these levels of detail, the detection algorithm can repeatedly refine the accuracy with which it detects collisions, at each repetition returning either a "no" or a "maybe" more accurate than the previous one. Figure 3.2 updates Figure 2.1 with pseudocode for an application that calls this sort of detection algorithm. The application controls the amount of refinement, so it has the flexibility to balance the accuracy of collision detection against the amount of available processing time. This concept is *progressive refinement*, which is a form of *negotiated graceful degradation* and an important element of time-critical computing.

3.1 Choosing Appropriate Degradation

Negotiated graceful degradation allows an application to interrupt the detection algorithm when it begins to take too much processing time. The application thus needs a policy for deciding how much time it can devote to collision detection.

The policy must consider how collision detection fits into all the activities the application performs to produce a frame. The pseudocode in Figure 3.2 includes five typical activities: getting user input, updating the behavior of objects (which may be based on the user input), updating any other application-specific bookkeeping, executing the loop that detects and responds to collisions (lines 7 through 15), and rendering the objects. Performance goals determine the maximum amount of processing time that the application can spend per frame to complete all these activities. To meet a frame rate of n frames per second, the per-frame maximum clearly must be at most $1/n$ seconds. The overhead of the operating system might dictate that the per-frame maximum should be less than $1/n$ seconds, especially if the operating system cannot provide real-time guarantees;

```
application()
{
    for t ← t_0 to t_1 in steps of Δt_r {
        get user input
        update behavior of each object in {O_0, ... O_{N-1}} as of t
        update any other general bookkeeping
        do {
            result ← detect(t, {O_0, ..., O_{N-1}})
            while ((result is "maybe") and
                    (more processing time is available))
                /* detect(t, result) knows to refine result. */
                result ← detect(t, result)
            if (result is "maybe")
                collision response
        } while (result is "maybe")
        render each object in {O_0, ... O_{N-1}} as of t
    }
}
```

Figure 3.2: A simple application that calls a time-critical detection algorithm, detect().

in practice, simple strategies to minimize the effects of time-sharing, such as those suggested by Rohlf and Helman [RH94], should minimize this overhead. Dealing with the goal of low latency (lag) is more difficult. If an input device could sample a user's actions immediately, and if the application acted on the input during the frame it was sampled, then the latency would be the processing time for one frame, $1/n$ seconds. The first assumption is reasonable for a simple input device like a desktop mouse, but not for a more sophisticated device like a tracker for head or hand motions in three dimensions. With current technology, the only way an application can compensate for latency in a tracking device is by predicting the values it will return in the immediate future. Liang *et al.* [LSG91], Friedmann *et al.* [FSP92], Deering [Dee92] and Azuma and Bishop [AB94] argue that techniques such as linear interpolation and Kalman filtering give acceptable prediction in some situations. Getting prediction to work in all situations is still a research problem, but for this dissertation I assume that prediction works and the application uses it. The latency is thus tied to the frame rate, that is, it is $1/n$ seconds.

The maximum per-frame processing time dictated by the target frame rate and latency is actually a "soft" deadline. Human users do not seem to notice small fluctuations in the frame rate or latency as long the application meets the performance goals "almost always." There seems to be no formal evidence for this conclusion, but there is anecdotal evidence. Funkhouser and Séquin [FS93], for example, report that their system is "interactive" when its frame times have a mean of 0.10 seconds, with a maximum of 0.13 and a standard deviation of 0.008.

Given a maximum amount of per-frame processing time (to be met almost always), the application must divide this time amongst the five per-frame activities mentioned earlier. Determining a division that meets some goal of optimality as the application runs is a difficult research problem. As a simple alternative, the programmers who write the application could determine a priori an acceptable division that holds throughout a run. Either approach requires a time-critical algorithm for each activity, that is, an algorithm that can terminate within a particular time limit (almost always).

The subsequent chapters of this dissertation concentrate on how to limit the time spent in collision detection, so the current section will discuss time limits for the other activities. Of these activities, rendering has received the most attention. Section 2.4 summarizes several time-critical approaches to rendering. The technique of Funkhouser and Séquin [FS93] is the only one that can adapt to a variety of time limits. This algorithm is incomplete, though, in that it assumes it has access to various levels of detail for each object, that is, it needs sets of polygons of different sizes to represent each object at different resolutions. The literature contains some work on the problem of building these levels of detail automatically. The approach of Hoppe *et al.* [HDD+93] looks most promising, but the problem is still far from solved. In practice, the application programmers will probably have to build many of the levels of detail by hand. For applications in which the visual complexity of the scene remains relatively constant, the full power of Funkhouser and Séquin's algorithm may not be necessary. These applications will render a similar population of polygons at each frame, so the rendering time will vary little between frames. The application programmers can empirically determine the rendering time, and adjust the visual complexity to meet a particular time limit.

A time limit for user input should not be a problem. Assuming that the application uses prediction to overcome input-device latency, new input is ready in the time it takes a new device sample to arrive and be processed by the prediction technique. The times involved in both steps are modest. Wloka [Wlo94] describes measurements of these times for one particular hardware configuration; similar empirical tests could determine the times for other situations.

Limiting the time involved in updating behavior is less straightforward. In sophisticated applications, the algorithms that compute behavior can be complicated. A forward simulation, for example, will involve solving differential equations numerically, and even efficient approaches like a Runge-Kutta method with adaptive step size [PTVF92] involve nontrivial amounts of computation. The application programmers will have to rely on insight into the application's semantics to pick appropriate tradeoffs between speed and accuracy. A Runge-Kutta method, for example, takes a parameter specifying the allowable level of error, and the programmers can increase this level to improve performance. As with user input, empirical tests will probably be the best way for the programmers to measure how much time they must allow for behavioral update of the appropriate quality. If these limits are conservative and behavioral update for particular frames takes less time than expected, the extra time can be absorbed by the time-critical collision detection and rendering algorithms. Evidence from previous animation systems, such as the work of Hahn [Hah88], suggests that tight limits on behavioral update times may not be so important because collision detection and rendering are often the real performance bottlenecks.

Bounding the time the application spends processing collision response is possible for simple responses. If the response algorithm simply destroys the colliding objects (perhaps replacing them with a simple pile of rubble) the processing time is minimal. Moore and Wilhelms [MW88] describe a response algorithm that temporarily inserts a stiff spring between colliding objects, to force them apart; this approach, often called a "penalty" method, takes little time if the application is already simulating the dynamics of objects. More sophisticated forms of collision response, such as those Baraff advocates [Bar89, Bar90, Bar91, BW92], unfortunately do not lend themselves to time limits.

This section has not distinguished between single-processor and multiple-processor applications. Multiple processors can help an application perform its activities more quickly, but they complicate the issue of latency. Wloka [Wlo94] discusses this problem and presents some pragmatic solutions.

To summarize this section, it is clear that the problem of bounding per-frame computations still offers many opportunities for research. Fortunately, the suggestions from this section, combined with the mechanisms from Chapters 4 through 6, do allow at least some interactive applications to run at real-time rates; Chapter 7 will describe one such example.

3.2 Implementing Degradation

The policy from the previous section allows an application to choose how much processing time it can devote to collision detection. Now the important issue is how to structure the detection algorithm so that it can be interrupted after the appropriate amount of time. The beginning of this chapter mentions the central idea, that the detection algorithm uses conservative approximations to object shapes at multiple levels of detail.

The detection algorithm's processing of these approximations divides naturally into two phases. The first phase, the *broad phase*, uses the most conservative approximation to each object, its bounding sphere. The second phase, the *narrow phase*, uses all the other, more-accurate approximations.

The central operation of the broad phase is finding the next *possible* collision between bounding spheres. Uncertainty is present in this computation because the behavior of the objects is controlled "on the fly" by human users. As a result, the broad phase cannot know exactly where each object will be in the future. Chapter 4 will argue, however, that an application can provide some information about where each object might be for the immediate future. If the application can supply an upper bound on each object's acceleration, the broad phase can build a conservative bound on the object's future position. This conservative bound is a four-dimensional structure (the fourth dimension explicitly representing simulation time) called a *space-time bound.*

Because space-time bounds are conservative, the broad phase can use their intersections to find the simulation time, t_i, of the next possible bounding-sphere collision. The broad phase then knows that there can be no collisions from the current simulation time, t, up to t_i. When the application calls the detection algorithm for each frame between t and t_i, the algorithm can return without doing any further work. The effect is as if the detection algorithm has an adaptive timestep, which stretches out because no collisions are possible. It should be intuitively clear that this approach addresses the fixed-timestep weakness from Section 2.2. The way the broad phase finds intersections between space-time bounds also addresses the all-pairs weakness. Thus, the broad phase is an improvement over the three outer loops at lines 17, 19 and 20 of the basic algorithm in Figure 2.1. Chapter 4 will discuss in detail how the broad phase avoids the fixed-timestep and all-pairs weaknesses.

The broad phase next does work when the simulation time, t_i, of the earliest possible collision arrives. At this time, the broad phase checks for intersection between the two bounding spheres it thought might collide, using for their positions the actual $t = t_i$ positions of the objects the spheres bound, O_a and O_b. The bounding spheres might not intersect, because the broad phase predicted the possible collision based on an estimate of a future. (Remember, however, that the estimate was conservative, so the algorithm will never miss a pair of bounding spheres that really do intersect.) If, on the other hand, the bounding spheres do intersect, then the broad phase has found a collision at the coarsest level of detail. The detection algorithm returns to the application a "maybe" answer (including references to O_a, O_b and the time, t_i, of collision).

As the beginning of this chapter describes, the application now has the chance to ask the algorithm to refine the accuracy of the "maybe" answer. If it does, the algorithm switches to the *narrow phase.* The narrow phase acts like the pair-processing algorithm in Figure 2.1. The narrow phase starts by checking for intersection between approximations to O_a and O_b at the second level of detail. If it again finds intersection, the narrow phase can continue to use more levels of detail as long as the application concludes that processing time is available. All the levels of detail in the approximation to an object's surface are represented by a data structure called a *sphere-tree.*

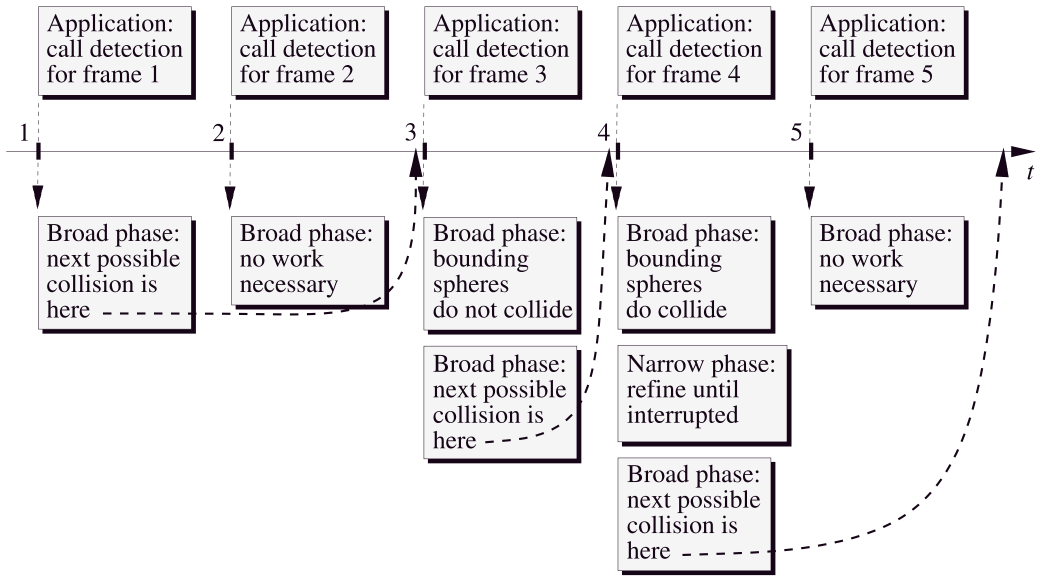


Figure 3.3: The broad phase and the narrow phase in action.

As the name suggests, this structure is a hierarchy each level of which is a union of spheres. Each such union approximates the surface of the object, and the approximations become more accurate at deeper levels of the tree. The advantage of spheres is the ease with which the narrow phase can check them for intersection. Sphere-trees thus allow the narrow phase to address the pair-processing weakness from Section 2.2. Chapters 5 and 6 explain this advantage of sphere-trees, and also discuss how to automatically build sphere-trees for objects with complicated shapes.

The narrow phase terminates when the application interrupts it, or when it exhausts the levels of detail in its sphere-trees, or when it returns a “no” answer. At this point, the broad phase begins again. The broad phase computes the time, t_i, of the next possible collision, and the whole process repeats. Figure 3.3 gives a schematic example that summarizes the process.

The algorithms for building sphere-trees involve considerable amounts of computation, so the application should not run them while it is interacting with human users. Instead, a sphere-tree for each object should be built off-line in a preprocessing step. This approach has the advantage that a sphere-tree built once can be used in many runs of the application. A further advantage is that the preprocessor executes most of the complicated code before the application starts running; the algorithm that checks two already-constructed sphere-trees for intersection at runtime is quite simple, so the chance that a user will encounter runtime errors is reduced. A disadvantage of this approach is that it can handle only rigid objects, that is, objects that do not deform as the application runs. An object that consists of several rigid components that move independently is not a problem, as it can be approximated by a set of sphere-trees, one for each component. Truly flexible objects, like those made of rubber, however, do not lend themselves to this approach. Fortunately, rigid objects are important, especially in engineering, and there are many useful applications that use only rigid objects.

Note that the places at which the application can interrupt the detection algorithm all fall within the narrow phase. This structure is disadvantageous because it reduces the application's ability to limit the algorithm's processing time. If, for example, the N objects are densely packed then many pairs of bounding spheres will collide at each frame. The broad phase will report each of these collisions, which may take a considerable amount of processing time for large values of N. The only way the application could avoiding spending this time would be if it were to allow the broad phase to ignore some bounding-sphere collisions. This change would allow some objects to pass through one another, an unacceptable result in general. In specific situations, however, this sort of nonconservative collision detection might be useful; Zyda *et al.* [ZOMP93] argue that some missed collisions are tolerable in large-scale vehicle simulations.

The broad phase and the narrow phase form a convenient division of labor in any detection algorithm, whether it is time-critical or not. (Certain four-dimensional techniques, such as Duff's interval-analysis algorithm [Duf92], do not divide neatly into two phases, but most algorithms do.) Much of the previous work summarized by Chapter 2 fits into this general framework; some previous techniques function as a broad phase and some as a narrow phase. It is thus possible to "mix and match" broad-phase algorithms and narrow-phase algorithms to create a variety of interesting hybrid algorithms. Such a hybrid algorithm could provide the best performance for a particular situation. Chapter 7 compares the algorithm using space-time bounds and sphere-trees to one hybrid algorithm, whose broad phase uses Turk's algorithm [Tur90b] and whose narrow phase is based on binary space partitioning (BSP) trees [FKN80, TN87]. Further study of hybrid algorithms is an interesting topic for future research.

Chapter 4

Space-Time Bounds

Section 3.2 introduced the broad phase of the detection algorithm. The broad phase checks the N objects to find the earliest possible collision between their bounding spheres. The key to this process is *space-time bounds*, four-dimensional structures whose fourth dimension represents simulation time. A space-time bound gives a conservative estimate of where an object's bounding sphere might be for the immediate future. This estimate is based on an upper bound on the object's acceleration for the immediate future. The definition of a space-time bound does not require the exact position of the object over time, so space-time bounds can handle objects whose motion is determined interactively by human users. To find the earliest possible collision, the broad phase finds intersections between space-time bounds. By finding these intersections, the broad phase addresses the fixed-timestep weakness, and empirical evidence suggests that the intersection algorithm also addresses the fixed-timestep weakness.

This chapter begins with the derivation of space-time bounds. It then presents two algorithms for finding intersections between space-time bounds, and compares the performance of the algorithms. It next details how these intersection algorithms fit into the broad phase. The chapter concludes by discussing acceleration bounds and how an application might compute them.

4.1 Deriving Space-Time Bounds

This section derives the space-time bound for an object, O. The derivation presents four different versions of four-dimensional bounding structures, each elaborating on the previous one. The final version is what the broad phase uses for its computations.

4.1.1 Parabolic Horns

To start the derivation, assume that object O is a point in three-dimensional (3D) space, $\mathbb{R}^3$. The position of O in $\mathbb{R}^3$ at simulation time t is described by the function $\mathbf{x}(t)$. The function $\dot{\mathbf{x}}(t)$ describes O's velocity at t, and $\ddot{\mathbf{x}}(t)$ describes its acceleration at t. Let the current simulation time be $t = 0$. As of this time, the detection algorithm knows O's current position, velocity and acceleration—$\mathbf{x}(0)$, $\dot{\mathbf{x}}(0)$ and $\ddot{\mathbf{x}}(0)$, respectively—but it does not know $\mathbf{x}(t)$, $\dot{\mathbf{x}}(t)$ or $\ddot{\mathbf{x}}(t)$ for $t > 0$.

Given this information, Taylor's Theorem [EG78] states that position of O at $t > 0$ is

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

where $\mathrm{O}(t^3)$ represents an unknown vector. This result is not precise enough to be useful, but the general idea leads to a more powerful conclusion. Say the application program has provided an upper bound on the acceleration magnitude, $|\ddot{\mathbf{x}}(t)|$, up to some future simulation time $t = 1$. In other words, the application has provided a scalar, M, such that

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

Section 4.4 will discuss how the application might compute M. This upper bound on acceleration yields a bound on $\mathbf{x}(t)$, expressed in the following lemma.

Lemma 4.1 *If* $|\ddot{\mathbf{x}}(t)| \leq M, 0 \leq t \leq 1$, *then*

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

Proof: Introducing two auxiliary functions, $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, simplifies the proof. First, define

$$\boldsymbol{\alpha}(t) \;=\; \mathbf{x}(t) - \mathbf{x}(0). \tag{4.1}$$

Note that

$$\boldsymbol{\alpha}(0) = 0, \quad \dot{\boldsymbol{\alpha}}(t) = \dot{\mathbf{x}}(t), \quad \ddot{\boldsymbol{\alpha}}(t) = \ddot{\mathbf{x}}(t).$$

Using this new function, the goal becomes proving that

$$|\boldsymbol{\alpha}(t) - \dot{\boldsymbol{\alpha}}(0)t| \;\leq\; \frac{M}{2}t^2, \quad 0 \leq t \leq 1.$$

Second, define

$$\boldsymbol{\beta}(t) \;=\; \boldsymbol{\alpha}(t) - \dot{\boldsymbol{\alpha}}(0)t. \tag{4.2}$$

Note that

$$\boldsymbol{\beta}(0) = 0, \quad \dot{\boldsymbol{\beta}}(0) = 0,$$

and

$$\ddot{\boldsymbol{\beta}}(t) \;=\; \ddot{\boldsymbol{\alpha}}(t) \;=\; \ddot{\mathbf{x}}(t). \tag{4.3}$$

The goal is now to show that

$$|\boldsymbol{\beta}(t)| \;\leq\; \frac{M}{2}t^2, \quad 0 \leq t \leq 1. \tag{4.4}$$

Applying the Fundamental Theorem of Calculus twice gives

$$\boldsymbol{\beta}(t) \;=\; \int_0^t \dot{\boldsymbol{\beta}}(u)\,du \;=\; \int_0^t \int_0^u \ddot{\boldsymbol{\beta}}(v)\,dv\,du. \tag{4.5}$$

The Triangle Inequality gives

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

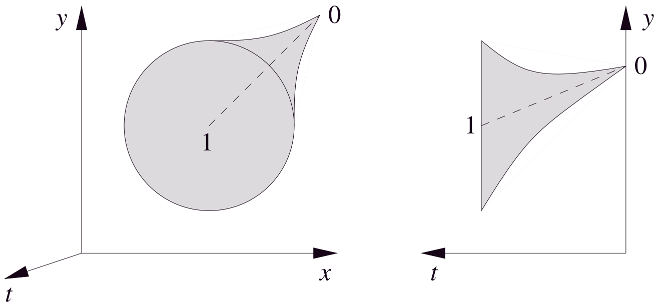


Figure 4.1: Two views of a parabolic horn for an object moving in $\mathbb{R}^2$. The point labeled "0" is $\mathbf{x}(0)$ and the point labeled "1" is $\mathbf{x}(0) + \dot{\mathbf{x}}(0)t|_{t=1}$.

The hypothesis and Equation 4.3 yield

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

The righthand integral evaluates to $(M/2)t^2$, so the proof of Inequality 4.4 is complete. □

The significance of Lemma 4.1 comes from its geometric interpretation. The lemma states that whatever the position of O at time t, it will be within a distance $(M/2)t^2$ from the point $\mathbf{x}(0)+\dot{\mathbf{x}}(0)t$. Thus, a three-dimensional bound on the position of O at time t is merely a sphere of radius $(M/2)t^2$ centered at $\mathbf{x}(0) + \dot{\mathbf{x}}(0)t$, a position which the detection algorithm knows in advance. A four-dimensional (4D) bound on the position of O over all times $t \in [0, 1]$ is the structure whose *cross section* at any particular t is that sphere. To get a better intuition for this 4D structure, consider the special case in which everything is one dimension lower, that is, O is moving in $\mathbb{R}^2$ and a structure that bounds O's motion over time has three dimensions. The cross sections of this bounding structure will now be circles of radius $(M/2)t^2$ instead of spheres. Figure 4.1 gives an example of the bounding structure in this case. Due to the factor of t^2 present in the 4D structure's definition, I call the structure a *parabolic horn*. Note that Lemma 4.1 guarantees that a parabolic horn is a *conservative* bound, in that O cannot be outside its parabolic horn for any $t \in [0, 1]$.

4.1.2 Expanded Parabolic Horns

This section relaxes the assumption that O is a point. If O is now a rigid body, its motion is more complicated because it can involve angular velocities and accelerations. At any moment in simulation time, t, however, this motion is an instantaneous rotation about a point, $\mathbf{y}(t)$, that is subject to only non-angular velocities and accelerations; if O's motion is governed by Newtonian dynamics, then $\mathbf{y}(t)$ is O's center of mass [MH82].

If O does not change its scale, then there is a 3D sphere that, when centered at $\mathbf{y}(t)$, bounds the space swept by these rotations. Let O's *rotational radius*, r, be the radius of this sphere. Building a parabolic horn for the point $\mathbf{y}$ and expanding every spherical cross section by r gives a parabolic horn that bounds O. The cross section of this expanded parabolic horn at simulation time t is a sphere of radius $(M/2)t^2 + r$. Figure 4.2 illustrates this idea. This simple approach to handling rotation can be overly conservative if the general shape of O in no way resembles a sphere. In this case, the detection algorithm could represent O by several overlapping parabolic horns, each bounding a part of O.

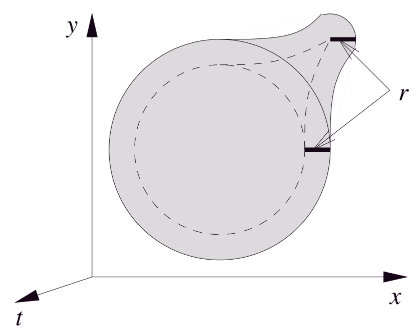


Figure 4.2: This expanded parabolic horn bounds the position through time of a non-point object moving in $\mathbb{R}^2$.

4.1.3 Hypertrapezoids

The conceptual simplicity of parabolic horns makes them an appealing way to bound objects in motion, but they have a computational disadvantage. The problem is the quadratic term, t^2. This term complicates the task of efficiently finding intersections within a set of space-time bounds, an important task for the broad phase. The only previous algorithm that can handle this task is the algorithm of Duff [Duf92], as summarized in Section 2.3. Duff's interval-analysis techniques are quite general, though, and they do not exploit the features of this particular task. To form the basis of a simpler and more efficient intersection algorithm, this section derives a linear approximation to a parabolic horn.

The linear approximation puts a tapered polyhedron around the expanded parabolic horn. The cross sections of the parabolic horn at $t = 0$ and $t = 1$ are spheres of radius r and $(M/2) + r$, respectively. The $t = 0$ and $t = 1$ cross sections of the polyhedron are the *isothetic* (axis-aligned) cubes that circumscribe these spheres. Each of the other cross sections of the polyhedron for $t \in (0, 1)$ is an isothetic cube whose size is a linear interpolation between the sizes of the $t = 0$ and $t = 1$ cubes. I call the 4D polyhedron a *hypertrapezoid* because it has pairs of parallel edges and pairs of non-parallel edges. A hypertrapezoid is guaranteed to enclose its parabolic horn because the radii of the parabolic horn's cross sections are defined by a *convex* function[1] in t. A hypertrapezoid is thus a *conservative* bound on O's position for $t \in [0, 1]$. Figure 4.3 shows the hypertrapezoid that bounds the parabolic horn from Figure 4.2.

Each cross section of a hypertrapezoid is a 3D isothetic cube. Each of these cubes has six square faces, each one normal to a coordinate axis of $\mathbb{R}^3$. The set of corresponding faces from all cross sections form a 4D structure. This structure, which is planar in $\mathbb{R}^4$ because the sizes of the cubes increase linearly, is a face of the hypertrapezoid. Just as each 3D cube is a cross section of the hypertrapezoid, each face of a cube is a cross section of a 4D face of the hypertrapezoid. The hypertrapezoid has six such faces, one for each face of a 3D cube. Note that each cross section of a particular 4D face is normal to the same axis of $\mathbb{R}^3$; it is convenient, therefore, to say that the 4D face is "normal to" that axis, too (even though the dimensions of the face and the axis do not match). This feature of 4D faces will become important in Section 4.2.

[1] A function $f(t)$ is convex on the interval $[t_1, t_2]$ if $f(t)$ is no greater than the convex combination of $f(t_1)$ and $f(t_2)$, that is, if $f(t) \leq \frac{t-t_1}{t_2-t_1} f(t_1) + \frac{t_2-t}{t_2-t_1} f(t_2)$, $t_1 \leq t \leq t_2$. A function is *convex* if it is convex on every interval.

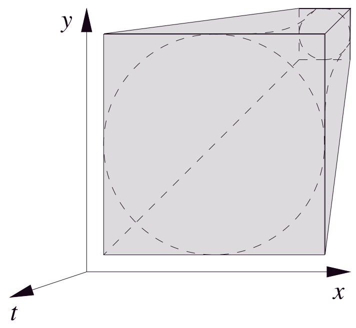


Figure 4.3: This hypertrapezoid approximates a parabolic horn for an object moving in $\mathbb{R}^2$.

4.1.4 Cutting Planes

The development of hypertrapezoids so far has not used any information about the direction of acceleration. This section shows how limits on direction remove part of a hypertrapezoid, tightening the bound it represents.

Assume, for the moment, that O is once again a point. Say the application can determine that at every time $t \in [0, 1]$, the acceleration vector $\ddot{\mathbf{x}}(t)$ for O will lie in one half of a sphere around O. Knowing this information is equivalent to knowing a vector $\mathbf{d} \in \mathbb{R}^3$ such that

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

How the application might compute $\mathbf{d}$ is the subject of Section 4.4. From this relationship between $\ddot{\mathbf{x}}(t)$ and $\mathbf{d}$ a lemma follows.

Lemma 4.2 *If $\ddot{\mathbf{x}}(t) \cdot \mathbf{d} \leq 0$, for all $t \in [0, 1]$, then*

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

Proof: This proof follows the same pattern as the proof in Section 4.1.1. Using the definitions of $\boldsymbol{\alpha}(t)$ and $\boldsymbol{\beta}(t)$ from Equations 4.1 and 4.2, respectively, the goal becomes showing that

$$\boldsymbol{\beta}(t) \cdot \mathbf{d} \ \leq \ 0, \quad 0 \leq t \leq 1. \tag{4.6}$$

Equation 4.5 still holds, so

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

By the linearity of integration,

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

Using Equation 4.3 and the hypothesis produces

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

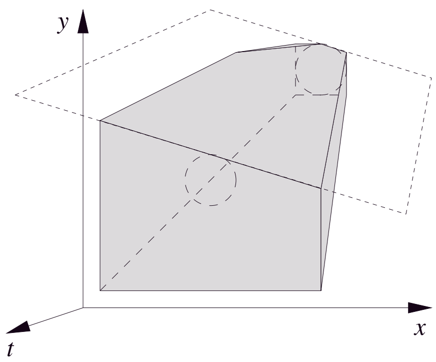


Figure 4.4: A complete space-time bound consists of a hypertrapezoid and a cutting plane.

The lefthand integral evaluates to $\boldsymbol{\beta}(t) \cdot \mathbf{d}$ and the righthand integral evaluates to 0, so the proof is complete. $\square$

To appreciate the significance of Lemma 4.2, think of $\mathbf{d}$ as the normal vector for a 3D plane that passes through the point $\mathbf{c}(t) = \mathbf{x}(0) + \dot{\mathbf{x}}(0)t$. Lemma 4.2 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. Thus, O itself will always be on the back side of the plane. Since O can never be in front of the plane, any part of a bound on O's position that lies in front of the plane can be discarded; Section 4.2 discusses how this discarding is actually implemented.

The set of 3D planes defined by $\mathbf{c}(t)$ and $\mathbf{d}$ over the time interval $[0, 1]$ are cross sections of a 4D structure. This structure is a 4D plane because the cross sections never change orientation ($\mathbf{d}$ is constant) and their movement through time is linear ($\mathbf{c}(t)$ is linear in t). I call this 4D plane a *cutting plane*, and a hypertrapezoid teamed with a cutting plane is a complete space-time bound. It would be possible to use more than one cutting plane with each hypertrapezoid, but using one seems to be an appropriate compromise between tightening a hypertrapezoid and adding complexity to the processing.

When O is not just a point, the cutting plane must be displaced away from the center of O so it will not cut through O itself. This displacement adds a constant term to the formula for $\mathbf{c}(t)$. Note that if the displacement is along the vector $\mathbf{d}$, then the geometric interpretation of Lemma 4.2 still holds. Figure 4.4 shows an example of a displaced cutting plane and a hypertrapezoid, together forming a space-time bound.

4.2 Finding Intersections Between Space-Time Bounds

If two objects are going to collide at simulation time $t \in [0, 1]$, their space-time bounds must intersect at some time $t_i \leq t$; this conclusion follows because space-time bounds are conservative over-estimates of where the objects might be. Say the detection algorithm has space-time bounds for all N objects, and it has found the earliest time t_i at which a pair of space-time bounds intersect. The algorithm can then conclude that no collisions are possible from time 0 until time t_i. Until t_i arrives, the detection algorithm has no more work to do; when the application calls it for each intervening frame it can immediately return that no collisions occurred. Section 4.3 will describe this approach more precisely, but it should be clear that this idea addresses the fixed-timestep weakness.

This section concentrates on how to find t_i while avoiding the all-pairs weakness.

The research literature contains few results on problems involving intersection of more than two objects in more than two dimensions. Section 2.3 explains that Samet and Tamminen's algorithm [ST85] can handle four-dimensional objects like space-time bounds, but it is inefficient for more than two objects. Duff's algorithm [Duf92] does not apply directly (because space-time bounds are not implicit functions), but the basic idea of using multidimensional subdivision is applicable. As Section 4.1.3 mentions, however, this approach is quite general, and only with particular optimizations is it likely to perform well for space-time bounds. This section describes an algorithm tailored to the particular features of space-time bounds. A variation on Duff's algorithm also fits into this framework, and this section compares the performance of the two algorithms.

4.2.1 Overall Strategy

The main complicating factor in the intersection algorithm is the cutting planes. Some of the complications are mitigated by the following observation: the intersection between two hypertrapezoid faces is a necessary condition for any intersection between two space-time bounds. To understand this fact, let B_1 be a space-time bound consisting of hypertrapezoid T_1 and cutting plane P_1; define B_2, T_2 and P_2 analogously. Remember that both B_1 and B_2 exist over the same t range, $0 \leq t \leq 1$. Assume that B_1 and B_2 are disjoint at $t = 0$. Consider the different ways B_1 and B_2 can intersect:

- An intersection might involve a face f_1 of T_1 and a face f_2 of T_2. Obviously, the intersection of hypertrapezoid faces is a necessary condition in this case.
- An intersection might involve a face f_1 of T_1 and the cutting plane P_2. A cutting plane can only remove volume from a hypertrapezoid, never add it; so only the part of P_2 within T_2 is important. This part of P_2 can be intersected by f_1 only if f_1 also intersects T_2, that is, if f_1 intersects a face f_2 from T_2. The region $f_1 \cap f_2$ may well be cut off by P_2, but it still must exist. Figure 4.5(B) illustrates this situation. The same argument holds for an intersection between a face f_2 from T_2 and P_1.
- An intersection might involve P_1 and P_2. By the argument from the previous case, only the parts of P_1 and P_2 within T_1 and T_2, respectively, matter. Thus, there must be an intersection between a face f_1 from T_1 and a face f_2 from T_2. See Figure 4.5(C).

There are no other ways B_1 and B_2 can intersect. It is convenient, therefore, to structure the search for intersections between space-time bounds around the search for intersections between hypertrapezoid faces. The relative simplicity and efficiency of finding intersections between faces is another reason for checking this case before checking any other cases.

Each time the intersection algorithm finds a pair of intersecting faces, it considers the cutting planes from those faces' space-time bounds. It checks to see if the intersection between the faces is cut off by one or other cutting plane (in which case the intersection does not matter), and it checks for intersections between the faces and the cutting planes or between the cutting planes themselves. Following the discussion of all these cases, Section 4.2.8 will present pseudocode for the full intersection algorithm.

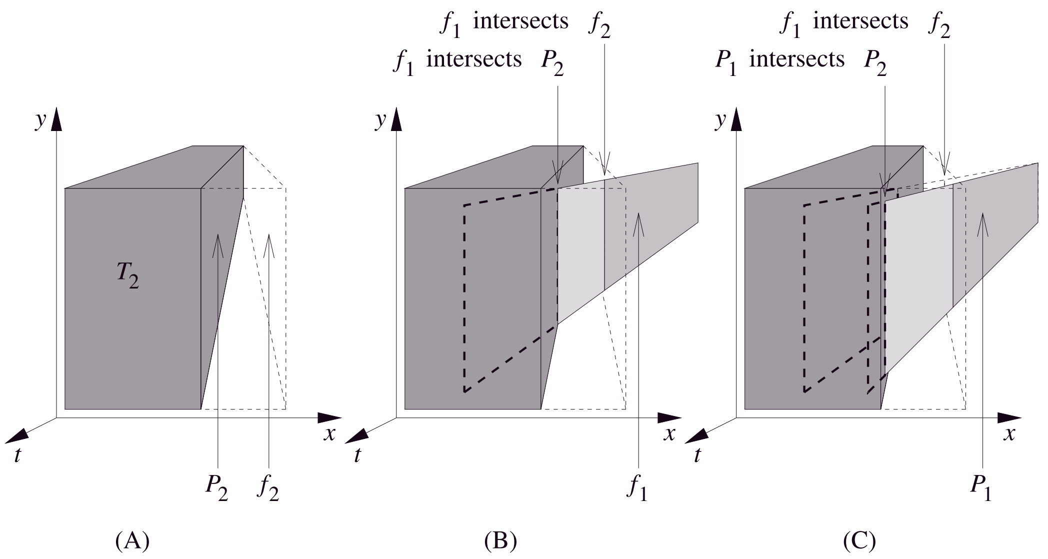


Figure 4.5: (A) A hypertrapezoid, T_2, its cutting plane, P_2, and a face, f_2, which is cut off. (B) If a face f_1 from another hypertrapezoid intersects P_2 it must first intersect a face such as f_2 from T_2. (C) If a cutting plane P_1 from another hypertrapezoid intersects P_2 there must first be an intersection between faces such as f_2 and f_1.

4.2.2 Face-Face Intersections: Preliminaries

Section 4.1.3 explained that each 4D face of a hypertrapezoid is "normal to" a principal coordinate axis of $\mathbb{R}^3$, that is, the 3D cross sections of the 4D face are all normal to that axis. The set of all 4D faces therefore can be partitioned into three subsets:

$$F_\alpha \;=\; \{f \,|\, \text{hypertrapezoid face } f \text{ is normal to axis } \alpha\}, \quad \alpha \in \{x, y, z\}. \tag{4.7}$$

These sets are important for the following reason: if two hypertrapezoids T_1 and T_2 are disjoint at $t = 0$ and intersect for the first time at $t = t' \in (0, 1]$, then there must be an intersection at t' between a face from T_1 belonging to F_α and face from T_2 also belonging to the *same* F_α, for some $\alpha \in \{z, y, z\}$. The proof of this assertion is messy but essentially trivial, so I omit it.

The contra-positive of this assertion sets the overall structure of 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 the algorithm has checked all three sets and has found no intersections, then the algorithm can conclude that there is no intersection between any hypertrapezoids.

The face-intersection algorithm needs a way to find intersections within a set F_α of faces. The next two sections describe two heuristic methods, one based on projection and the other based on subdivision. The subdivision method draws on a few ideas from the projection method, so the description starts with the latter. Section 4.2.5 compares the performance of the two methods.

4.2.3 Face-Face Intersections: The Projection Method

For each face f in F_α, each 3D constant-t cross section of f is normal to the α axis of $\mathbb{R}^3$. All points on a cross section thus have not only the same t coordinate but also the same α coordinate. As a

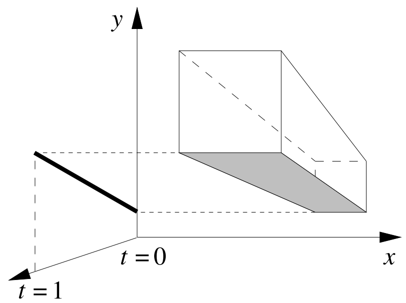


Figure 4.6: Each face in F_α projects into a straight line segment in the α-t plane. Here, $\alpha = y$.

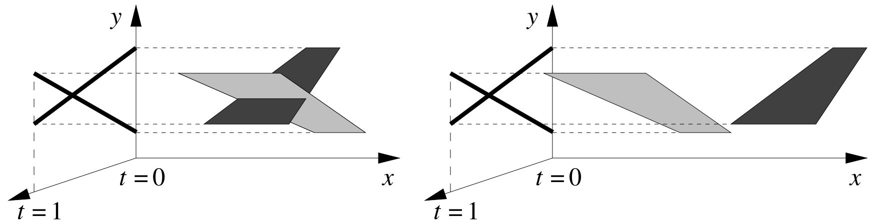


Figure 4.7: Intersecting segments are a necessary but not sufficient condition for intersecting faces. For clarity, this diagram hides the other faces of the hypertrapezoids.

consequence, the projection of f into the α-t plane is a straight line segment. Figure 4.6 illustrates this idea. Note that for every face the projected line segment will start at $t = 0$ and end at $t = 1$.

For any pair of faces f_1 and f_2 in F_α, an intersection between their projected line segments is a necessary condition for an actual intersection between the faces. It is not a sufficient condition, though, because the projection throws away some information. Figure 4.7 gives two examples that illustrate this situation. If the projections of f_1 and f_2 intersect at $t = t' \in [0, 1]$, the actual intersection between the faces, if it exists, must also be at t'. The faces do actually intersect if their $t = t'$ cross sections intersect. Since the two cross sections are isothetic squares at the same α coordinate, checking for their intersection reduces to the simple problem of finding the intersection between two squares in the β-γ plane (letting β and γ be the axes of $\mathbb{R}^3$ other than α).

The algorithm that finds this intersection needs the β and γ coordinates of the squares' corners. Recall that each square, s, is one side of a isothetic 3D cube, that cube being the $t = t'$ cross section of a hypertrapezoid, T. This s shares an edge with four other sides of the cube. It should be clear that knowing a point on each of these other sides (four points total) is enough information to compute the corner coordinates of s; this fact is a nice feature of isothetic cubes. To obtain these four points, the algorithm needs a formula for some point on a side of T's cross-sectional cube at $t = t'$, that is, some point on the $t = t'$ cross section of a face of T. Let T correspond to an object, O, with $\mathbf{x}(0)$, $\dot{\mathbf{x}}(0)$, r and M defined as in Section 4.1. Let $\mathbf{u}_\beta$, a unit vector pointing along the positive β axis, be the outward-pointing normal vector for each cross section of the face of T. A point on the face at $t = 0$ is

$$\check{\mathbf{p}} = \mathbf{x}(0) + r\mathbf{u}_\beta, \tag{4.8}$$

and a point on the face at $t = 1$ is

$$\hat{\mathbf{p}} \;=\; \mathbf{x}(0) + \dot{\mathbf{x}}(0) + \left(\frac{M}{2} + r\right)\mathbf{u}_\beta. \tag{4.9}$$

Recalling that each hypertrapezoid face is linear, a point on the face's cross section at time t is

$$\mathbf{p}(t) = \check{\mathbf{p}} + (\hat{\mathbf{p}} - \check{\mathbf{p}})\,t, \quad 0 \le t \le 1. \tag{4.10}$$

Note that if the face's outward-pointing normal vector points down the negative β axis, then Equation 4.10 still holds with only one modification to Equations 4.8 and 4.9: replace $\mathbf{u}_\beta$ with $-\mathbf{u}_\beta$.

The intersection between a pair of face cross sections, if it exists, will be an isothetic rectangle. Once the algorithm finds such an intersection, it still must consider whether the rectangle has been cut off by any cutting planes (if so, the intersection does not matter). 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 corners are behind the t' cross section of one or other cutting plane. Determining if a corner is behind this cross section involves computing the 3D vector from the corner to $\mathbf{c}(t')$, a point on the cutting plane (see Section 4.1.4); if the dot product of this vector with the cutting plane's 3D normal vector is negative, then the corner is behind the cutting plane.

The idea of projecting pairs of faces into the α-t plane leads to a complete algorithm for finding intersections in F_α. The algorithm first projects every face into a line segment. It then steps through the segment intersections in increasing t order. At each segment intersection, the algorithm checks for an actual face intersection by checking the appropriate cross-sectional squares.

A key to this algorithm is efficiently finding intersections between 2D line segments. One solution would be to test every pair of segments for intersections, but this approach suffers from the all-pairs weakness. A better alternative is to use the algorithm of Bentley and Ottmann [BO79, PS85]. This algorithm not only gives nearly-optimal asymptotic performance, but its straightforward implementation has low overhead in practice. In its full generality, this algorithm sweeps a line across the α-t plane from $t = 0$ to $t = 1$, updating its data structures whenever a new segment starts or ends, and reporting every intersection it finds. For the application of finding face intersections, a simpler version of the algorithm suffices. All the segments start at $t = 0$ and end at $t = 1$, so the algorithm does not need to worry about segments starting late or ending early. Also, the algorithm can stop as soon as it finds a segment intersection corresponding to a valid space-time-bound intersection, that is, an actual face intersection that matches one of the cases from Section 4.2.1.

The only complication in using the Bentley-Ottmann algorithm is inaccuracy in floating-point computations. As it runs, the algorithm tracks the ordering of the segments along a vertical line. In Figure 4.8, for example, the ordering as of $t = 0$ is s_4, s_2, s_1, s_3. The ordering changes at each intersection; the two intersecting segments swap and two new sets of intersections become possible. In Figure 4.8, the intersection between s_1 and s_2 at $t = t_a$ causes them to swap positions, making intersections between the pairs s_1-s_4 and s_2-s_3 possible. If either pair does actually intersect, the algorithm adds the pair to a priority queue, keyed on the intersection's t coordinate. The algorithm then advances to the intersection at the head of the queue, the one with the lowest t coordinate. When processing this "new" intersection, the algorithm must remember if it has seen it before. The intersection at $t = t_b$ in Figure 4.8, for example, causes s_1 and s_2 to become adjacent in the ordering, but they have already intersected. Surprisingly, neither published description of the algorithm [BO79, PS85] describes how to handle this problem. The algorithm cannot merely ignore the "new" intersection if its t coordinate is less than the t of the current intersection, because floating-point inaccuracy can affect the computation of the t coordinate. In Figure 4.8, when the algorithm processes the s_1-s_2 intersection at $t = t_a$, it computes t_b from the s_1-s_3 intersection; if floating-point

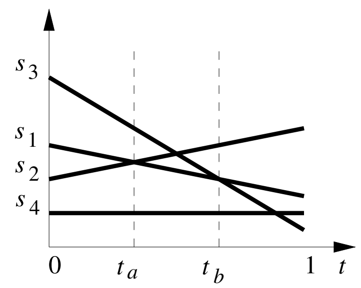


Figure 4.8: A set of four segment being processed by the Bentley-Ottmann algorithm.

inaccuracy makes $t_b < t_a$ by mistake, then the algorithm will miss the s_1-s_3 intersection altogether. A better solution is to explicitly store every processed intersection in a hash table. Managing a hash table takes longer than comparing two floating-point numbers, but execution profiles indicate that the extra time is insignificant.

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

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

since each segment from each object can intersect all the segments from the other $N-1$ objects. One can imagine pathological cases in which the algorithm processes all $2N^2 - 2N$ of these segment intersections without finding any actual face intersections. I did not encounter any of these pathological cases in the testing of the algorithm, though. In fact, for the hundreds of test runs described in Chapter 7, 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.

4.2.4 Face-Face Intersections: The Subdivision Method

Section 2.3 describes many uses of subdivision in collision detection. The work of Duff [Duf92] most closely matches the approach in this section. A good way to understand the underlying idea is through a lower-dimensional example.

Consider objects moving in a one-dimensional world, the x axis. Space-time bounds for these objects have two dimensions, and the faces of the hypertrapezoids are 2D line segments. Figure 4.9 shows a sample configuration of these faces. To find the intersections between these faces, the subdivision algorithm starts by processing isothetic square regions in x and t. If the number of faces in the current square is above the base-case threshold, n_b, the algorithm subdivides the square into quadrants. Since the quadrants are isothetic, subdividing involves merely splitting the current square along the midpoint lines for each dimension, for example, the lines $x = x_m$ and $t = t_m$ in Figure 4.9(B). For each face in the current square, the algorithm clips the face against the quadrant boundaries and assigns the resulting pieces to the quadrants that contain them. The algorithm then adds the quadrants to a priority queue, keyed on the minimum t coordinate of the quadrant. The quadrant at the head of the queue becomes the next square the algorithm processes. This quadrant has a minimum t coordinate lower than any other unprocessed quadrant, so the algorithm finds intersections with low t coordinates first.

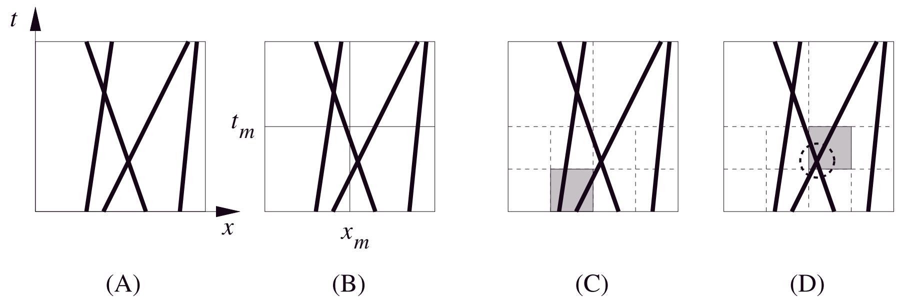


Figure 4.9: Four steps of the subdivision method, processing a set of 2D hypertrapezoid faces.

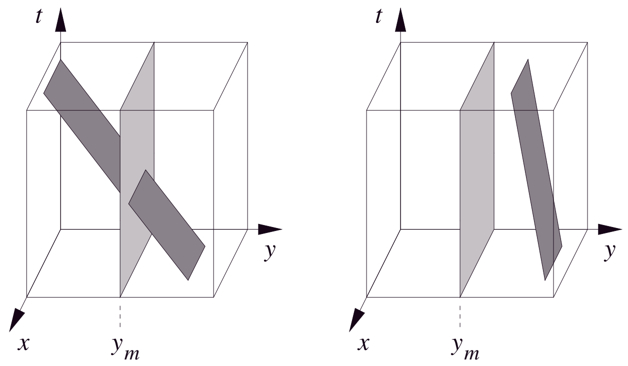


Figure 4.10: The cross sections of the face are parallel to the midpoint plane, $y = y_m$.

The algorithm will eventually find a square that contains n_b or fewer face pieces. In this base case, the algorithm tests all possible pairs of face pieces for intersection. If there are no intersections, as in the shaded quadrant from Figure 4.9(C), then the algorithm continues on to the next quadrant from the head of the queue. If, on the other hand, there are intersections, as circled in the shaded quadrant from Figure 4.9(D), then the algorithm returns the intersection with the lowest t coordinate.

In theory, it is not difficult to generalize this approach to handle 4D hypertrapezoid faces. The algorithm now processes 4D isothetic cubes, or *cells*, instead of squares. When the number of faces in a cell exceeds n_b and the algorithm subdivides the cell, the result is 16 smaller subcells.

In practice, however, it is tricky to implement the subdivision step, in particular, the clipping of faces and the assignment of the resulting pieces to the appropriate subcells. By analogy to Figure 4.9, subdividing involves splitting the cell by the four midpoint planes for the cell, $x = x_m$, $y = y_m$, $z = z_m$ and $t = t_m$. Although the splitting is symmetric, I find it simpler to treat the t dimension differently. For each face to be clipped, the algorithm finds the t intervals during which the face is on each side of the $\alpha = \alpha_m$ plane, $\alpha \in \{x, y, z\}$. These t intervals determine the face's relationship to the $t = t_m$ plane, and which of the 16 subcells contain pieces of the face.

Finding the t intervals during which a face $f \in F_\alpha$ is on each side of a plane is simplest if the plane is $\alpha = \alpha_m$. The advantage here is that the cross sections of f are parallel to the plane, so if f ever crosses the plane it does so at only one t coordinate. Figure 4.10 illustrates two cases that can occur; the figure shows 3D faces but the situation is analogous for 4D faces.

If the plane is $\beta = \beta_m$ for $\beta \neq \alpha$, then the the intersection between f and the plane is more

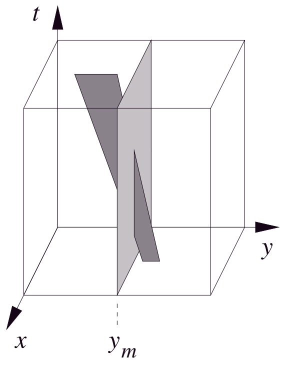


Figure 4.11: Both edges of the face intersect the midpoint plane, $y = y_m$.

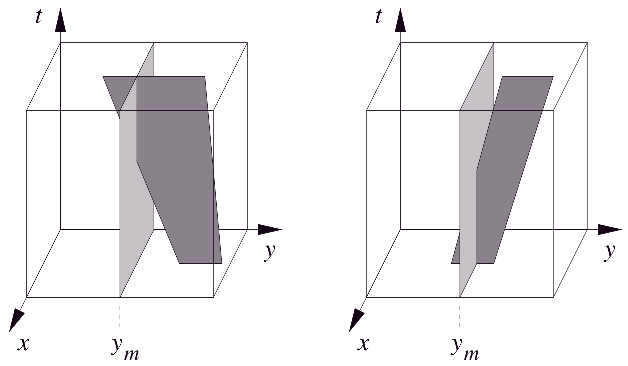


Figure 4.12: One edge of the face intersects the midpoint plane, $y = y_m$.

complicated. It helps to observe that f must adjoin two faces whose cross sections *are* parallel to the plane. The t intervals during which these two faces are on each side of the plane determine the t intervals for f because the adjacent faces define the edges of f. There are several cases, depending on which of these edges intersect the plane. Figure 4.11 shows the case in which both edges intersect the plane. Figure 4.12 shows two cases that occur when only one edge intersects the plane. Finally, Figure 4.13 shows the cases for no edge intersections. All these figures show f entering the cell at the minimum t coordinate and leaving at the maximum t. Simple variations on these cases occur if f enters late or leaves early.

When the algorithm finds a cell containing no more than n_b faces (the base-case threshold), then the algorithm checks all pairs of these faces for intersections. The simplest way to determine if two faces intersect is to use the projection method, from Section 4.2.3: project the two faces into 2D line segments, and if the segments intersect at $t = t'$ check the face cross sections at $t = t'$.

4.2.5 Comparing Face-Face Intersection Methods

Both the projection method and the subdivision method can perform poorly in the worst case. As Section 4.2.3 mentions, the projection method could process $2N^2 - 2N = O(N^2)$ segment intersections without finding any actual face intersections. Figure 4.14 shows a 3D example of the "spiral"

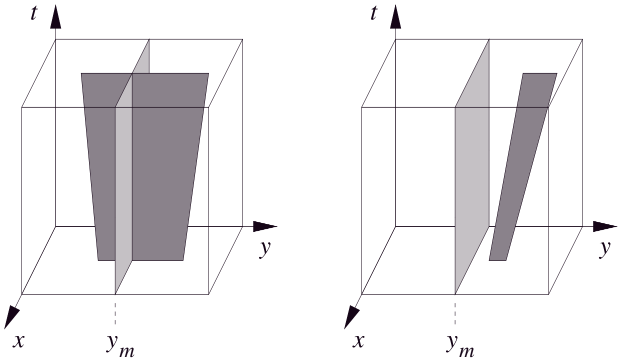


Figure 4.13: Neither edge of the face intersects the midpoint plane, $y = y_m$.

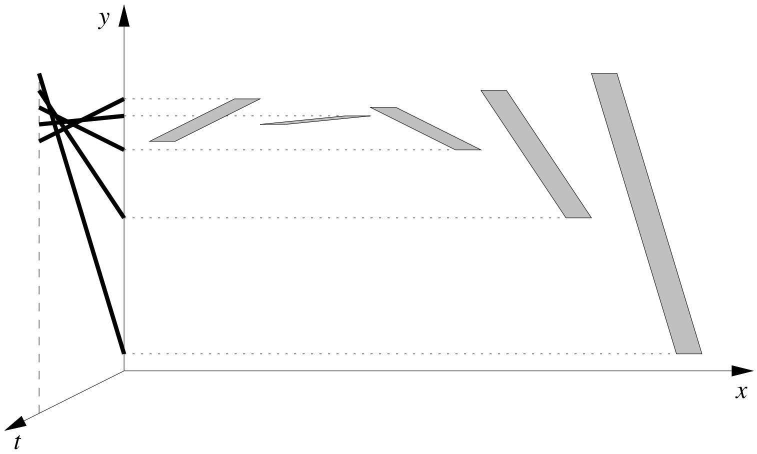


Figure 4.14: The worst case for the projection method.

of faces that creates the worst case; for clarity, the figure shows only one face per hypertrapezoid instead of two, quartering the number of intersections. Figure 4.15 illustrates a 2D version of the subdivision method's worst case; again, the figure shows only one face per hypertrapezoid. The squares in the figure are the base cases of the subdivision, the cells containing at most $n_b = 3$ faces. Notice that each of the $N = 8$ faces appears in at least N base-case cells, so the work is $\Omega(N^2)$ despite there being no face intersections.

These worst cases are rather contrived, however. To get a better idea of how the two methods might actually perform in practice, I ran empirical tests. A test consists of a number of "trials"; the test sets the general pattern for the N hypertrapezoids, and each trial is a random variation on that pattern. (Note that the tests do not involve cutting planes, since the emphasis is on the performance when finding face intersections.) Having fixed a particular set of hypertrapezoids, a trial measures how long it takes each intersection method to find the earliest intersection. The Unix "clock" routine makes the time measurements, with a stated resolution of 16.667 ms.

Every test involves 200 trials. In all the tests, the $t = 0$ size of each hypertrapezoid is drawn from the same random distribution; more specifically, each hypertrapezoid has as its rotational radius, r (see Section 4.1.2), a random number uniformly distributed over the interval $[\,0.5, 1.5\,]$. I ran all

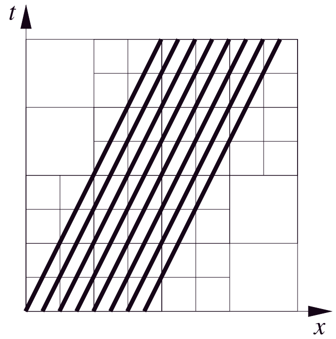


Figure 4.15: The worst case for the subdivision method.

the tests on a Sun SPARCstation 10/30. To compare the performance of the two methods, each trial of each test reports the *speedup* of the projection method over the subdivision method, which is defined as follows:

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

This speedup is a dimensionless, relative measure of performance. Note that a speedup greater than 1 means the projection method is faster than the subdivision method, while a speedup less than 1 means the subdivision method is faster.

The first set of tests involve the most randomness, hopefully giving an approximation to the "average" case. The initial position, $\mathbf{x}(0)$, of each of $N = 100$ hypertrapezoid is a random point uniformly distributed within a sphere of radius 200. Each hypertrapezoid's initial velocity, $\dot{\mathbf{x}}(0)$, has a random direction uniformly distributed over the unit sphere, and a random magnitude uniformly distributed over the interval $[5, 15]$. The upper bound on acceleration for a hypertrapezoid is uniformly distributed over $[0, 4]$. The only difference between the tests in this set is the value of the base-case threshold, n_b, for the subdivision method. Repeating the same trials for different values of n_b gives some indication of whether subdivision is actually helpful. Figure 4.16 shows the results of these tests. There is one scatter plot for each test, and each scatter plot has a circle for each trial; the abscissa of the circle is the simulation time of the earliest intersection in that trial, and the ordinate is the speedup of the projection method for detecting that intersection. The graphs use a logarithmic scale so that speedups less than 1 are not hidden. In these results, the speedup is always greater than 1, that is, the projection method is always faster. Note also that the subdivision method gets faster as n_b gets bigger, as it uses *less* subdivision. This counter-intuitive result suggests that subdivision carries with it a significant overhead, and this overhead can outweigh the advantages of subdivision in terms of reducing the number of face intersection tests.

This set of tests uses a particular number of hypertrapezoids and a particular *density* of hypertrapezoids, that is, the expected number of hypertrapezoids per unit volume. I ran similar "average" case tests using variations on these parameters, specifically, $N = 50$ and $N = 200$, and densities of half and double the original value. In all these variations, the speedups are essentially identical to those in the graphs from Figure 4.16, so I omit the graphs.

The next set of tests creates a more structured configuration of hypertrapezoids, one closer to the worst case for the projection method. In each test, the hypertrapezoids are "aligned" with the x axis. More specifically, the initial velocities of the $N = 100$ hypertrapezoids are all parallel to the x axis. The initial positions are all within 0.1 of the y-z plane. These values cause all the faces in F_x to

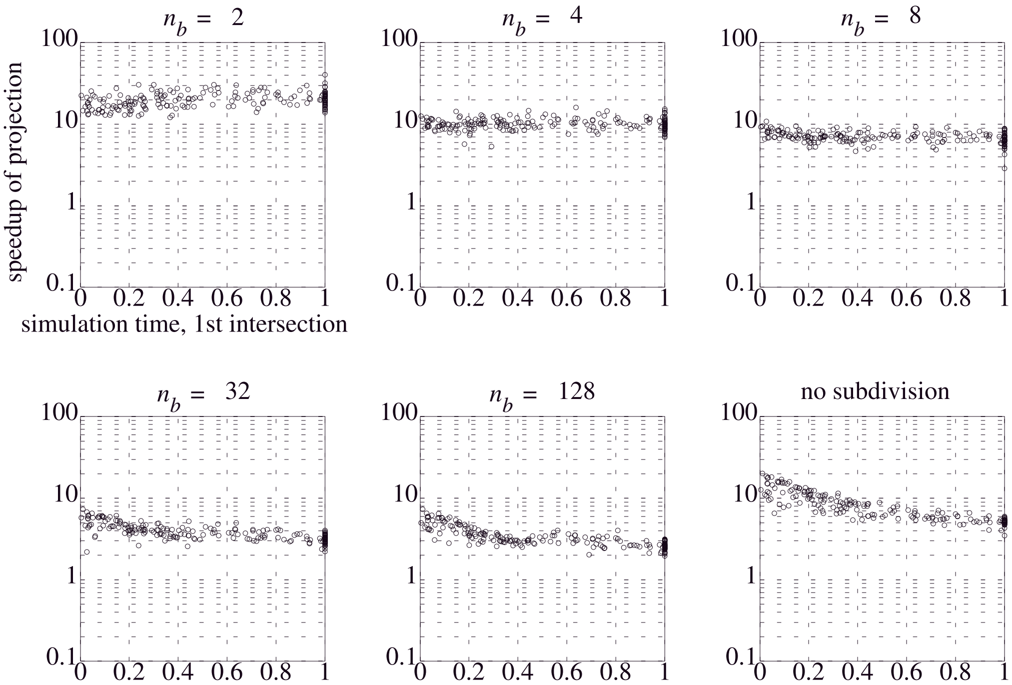


Figure 4.16: Results of the random, "average-case" tests for $N = 100$ hypertrapezoids. Each circle represents one trial.

have nearly equal x coordinates at $t = 0$. As a consequence, when the projection method turns these faces into segments in the x-t plane it finds many intersections, even though the faces themselves might not intersect at all. Figure 4.17 shows the results of these tests. As with the previous set of tests, I tried various values of n_b. The results indicate that the projection approach is faster for small values of n_b. As n_b increases, the performance of the two methods becomes similar, until for $n_b > 32$ the subdivision method is faster. I ran a similar set of tests for hypertrapezoids aligned with the y axis, and also with the z axis; the results were close enough to the tests of x-axis alignment that I omit the graphs.

An even more structured situation aligns hypertrapezoids with the x-y plane. This form of alignment gives the hypertrapezoids initial velocities uniformly distributed in the x-y plane, and initial positions lined up within 0.1 of the z axis. Due to this alignment, all the faces in F_x have nearly equal x coordinates at $t = 0$, and likewise for the y coordinates of the faces in F_y. As with the previous set of tests, the projection method should thus find many intersections between segments in the x-t plane, and also in the y-t plane, before finding any actual face intersections. Figure 4.18 graphs the speedups from these tests for $N = 100$ hypertrapezoids. The projection method is almost always faster for all values of n_b. For similar sets of tests involving alignment with the x-z plane and with the y-z plane, the speedup of the projection method was even greater; this increase is due to the projection method's processing of the face sets in the order F_x, F_y then F_z.

Note that the intersection times in Figure 4.18 are all clustered around $t = 0$. These early intersections are due to the maximum accelerations, M, making the hypertrapezoids spread quickly into each other. Configurations of aligned hypertrapezoids with $M = 0$ for all hypertrapezoids pose a challenge to both intersection methods, as the details of the alignment guarantee that the

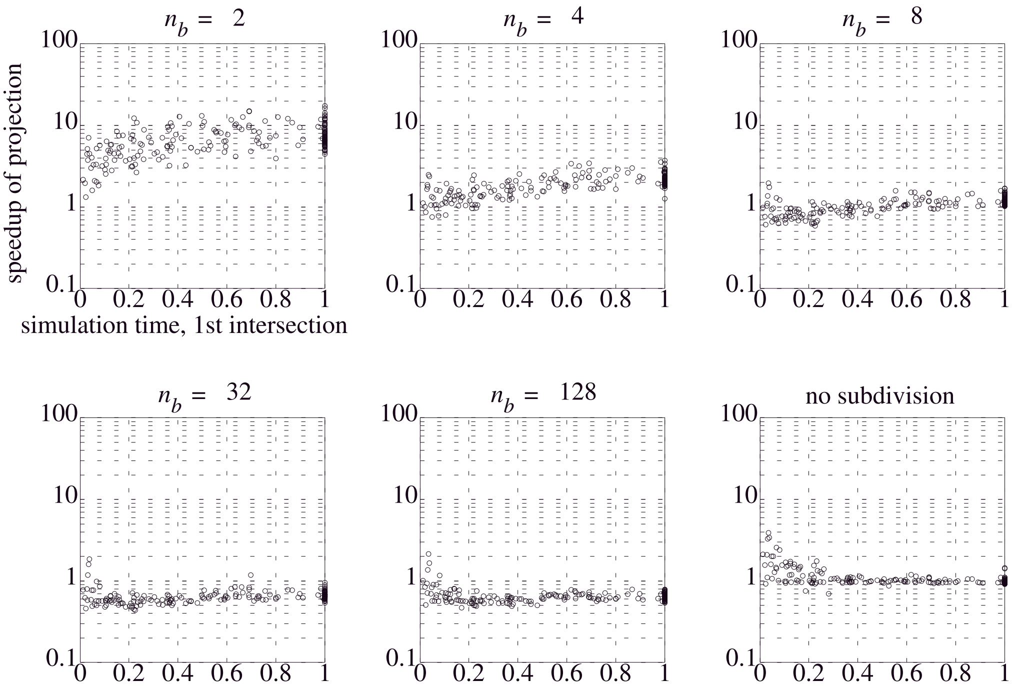


Figure 4.17: Results of the tests involving $N = 100$ hypertrapezoids aligned with the x axis. Each circle represents one trial.

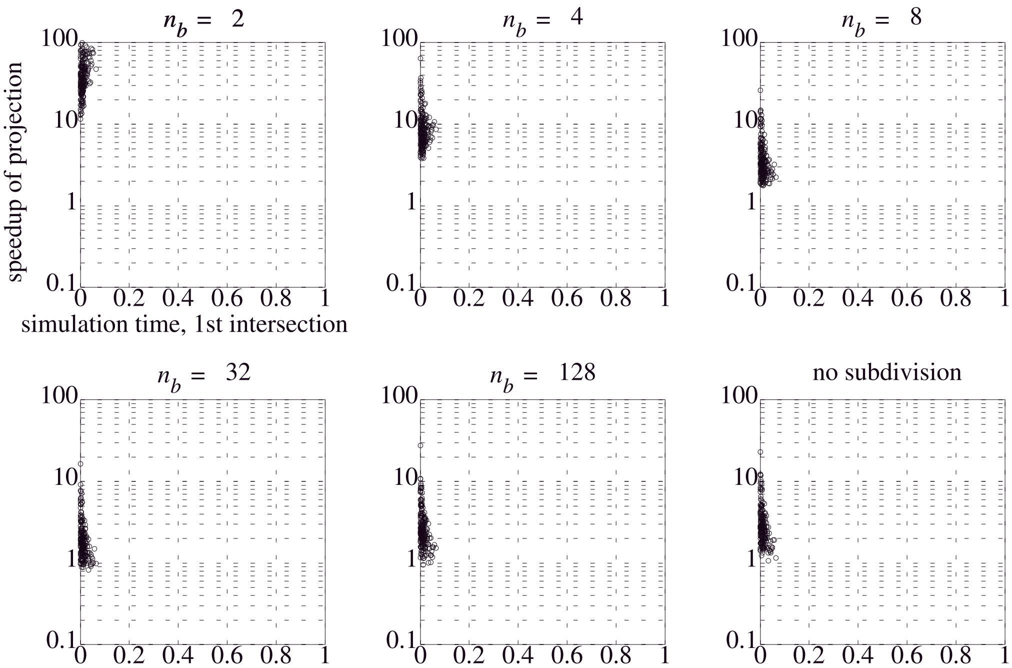


Figure 4.18: Results of the tests involving $N = 100$ hypertrapezoids aligned with the x-y plane. Each circle represents one trial.

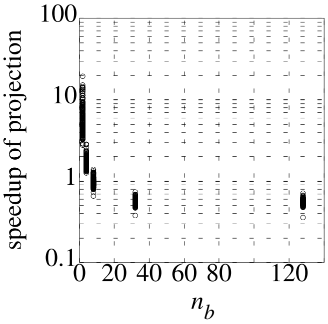


Figure 4.19: Results of the tests involving $N = 100$ hypertrapezoids aligned with the x axis with $M = 0$. Each circle represents one trial.

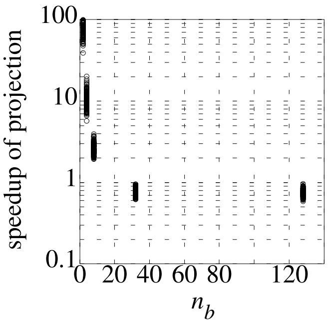


Figure 4.20: Results of the tests involving $N = 100$ hypertrapezoids aligned with the x-y plane with $M = 0$. Each circle represents one trial.

hypertrapezoids will be densely packed but never intersecting. Using $M = 0$ I repeated the tests for alignment with the x axis and with the x-y plane. Figure 4.19 shows the results for alignment with the x axis, and Figure 4.20 for alignment with the x-y plane. Note that unlike the previous graphs these scatter plots do not use intersection time as their abscissa, since no intersections occur; instead, they use n_b as the abscissa. The graphs indicate that the subdivision method is almost always faster. These types of hypertrapezoid configurations are rather pathological, however, and since they may be rare in real applications the significance of these tests is unclear.

Based on the results of all these tests, I favor the projection method over the subdivision method. The results from the tests involving alignment are inconclusive, with each method showing advantages in some of the tests. The random tests, however, show a significant advantage for the projection approach. I place more emphasis on these results, because the random tests should be a better approximation to the "average" case.

4.2.6 Face-Plane Intersections

Face-face intersections are not the only way two space-time bounds can intersect. As Section 4.2.1 describes and Figure 4.5 illustrates, a face can also intersect a cutting plane. Say the face is f_1 and the plane is P_2. Let T_1 be the hypertrapezoid that includes f_1, and let P_1 be the cutting plane associated with T_1; let T_2 be the hypertrapezoid associated with P_2. From Section 4.2.1, f_1 and

P_2 can intersect only if f_1 intersects some face, f_2, from T_2. The intersection between f_1 and P_2 matters only if the intersection between f_1 and f_2 is cut off by P_1 or P_2. To see why, note that any part of a face that is cut off at the $t = t_c$ stays cut off for $t > t_c$. Thus, if the intersection between f_1 and f_2 is not cut off, it must occur at a lower t coordinate than the intersection between f_1 and P_2. Since only the earliest intersection between space-time bounds matters, the lower t coordinate is the important one.

Assume, then, that the intersection between f_1 and f_2 has been cut off. To characterize the intersection between f_1 and P_2, it is convenient to start with the intersection between P_2 and p_1, where p_1 is the 4D plane that contains f_1. Let $\mathbf{n}_1$ and $\mathbf{n}_2$ be the 3D normal vectors for the 3D constant-t cross sections of p_1 and P_2, respectively. The line of intersection between p_1 and P_2 has direction

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

This direction remains constant throughout the simulation-time interval $[0, 1]$ because neither plane changes its orientation. Positions along this line must be defined in terms of some reference point, $\mathbf{r}(t)$, guaranteed to lie on the line at time t. Given a point, $\check{\mathbf{r}}$, on the line at $t = 0$ and a point, $\hat{\mathbf{r}}$, on the line at $t = 1$, a legitimate $\mathbf{r}(t)$ is

$$\mathbf{r}(t) \ = \ \check{\mathbf{r}} + (\hat{\mathbf{r}} - \check{\mathbf{r}})\, t, \quad 0 \leq t \leq 1. \tag{4.11}$$

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

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

The values $\check{c}_1$ and $\check{c}_2$ are the plane constants for cross sections of p_1 and P_2, respectively, at $t = 0$; recall that a plane's constant is the dot product of the plane's normal with any point on the plane, for instance, the point defined in Equation 4.10. These three equations in three unknowns (the three coordinates of $\check{\mathbf{r}}$) can be solved for the value of $\check{\mathbf{r}}$ using standard matrix techniques. Repeating this process for $t = 1$ gives $\hat{\mathbf{r}}$. Given the formula for $\mathbf{r}(t)$, each point on the intersection between p_1 and P_2 is

$$\mathbf{i}(t, k) \ = \ \mathbf{r}(t) + k\mathbf{m}, \tag{4.12}$$

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

Not every point $\mathbf{i}(t, k)$ corresponds to a valid intersection between f_1 and P_2. The point could be on the part of p_1 outside the boundaries of f_1, or on the part of P_2 outside the boundaries of T_2, or in the space cut off by P_1; in all of these cases, the point is not a real intersection point.

A solution to this problem is to derive constraints that characterize only real intersection points. The constraints come from the interpretation of a space-time bound as the intersection between seven half-spaces. To understand this interpretation, notice that a cubical cross-section of a hypertrapezoid is certainly the intersection of six half-spaces, where each half-space is the volume behind the plane containing one side of the cube. Section 4.1.3 derives a hypertrapezoid from its cross-sections in such a way that it is also the intersection of six half-spaces, each one being bounded by the 4D plane that contains a hypertrapezoid face. The cutting plane defines the seventh relevant half-space.

A half-space defines a constraint expression as follows. Consider one of the half-spaces that defines a hypertrapezoid. Let $\mathbf{n}_H$ be the outward-pointing normal vector for a 3D cross-section of the half-space's bounding plane. If $\mathbf{q}(t)$ is a point on the bounding plane (either $\mathbf{p}(t)$ from Equation 4.10 or $\mathbf{c}(t)$ from Section 4.1.4), then the point $\mathbf{i}(t,k)$ is inside that half-space only if

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

Solving for k in terms of t characterizes those points on the intersection line at time t that satisfy the constraint. Plugging in the expression for $\mathbf{i}(t,k)$ from Equation 4.12 yields

$$[(\mathbf{r}(t) + k\mathbf{m}) - \mathbf{q}(t)] \cdot \mathbf{n}_H \leq 0. \tag{4.13}$$

If $\mathbf{m} \cdot \mathbf{n}_H > 0$, then solving for k gives

$$k \leq \frac{[\mathbf{q}(t) - \mathbf{r}(t)] \cdot \mathbf{n}_H}{\mathbf{m} \cdot \mathbf{n}_H}.$$

When $\mathbf{m} \cdot \mathbf{n}_H < 0$ the result is the same with the direction of the inequality reversed. The next paragraph discusses the case in which $\mathbf{m} \cdot \mathbf{n}_H = 0$. Since both $\mathbf{q}(t)$ and $\mathbf{r}(t)$ are linear in t, the expression for k is also linear in t. The expression thus defines a half-plane in the t-k plane, that is, a line that separates the pairs (t,k) that correspond to valid intersection points $\mathbf{i}(t,k)$ from those pairs that give invalid intersection points.

If $\mathbf{m} \cdot \mathbf{n}_H = 0$, Equation 4.13 cannot be solved for k as a function of t. This dot product is zero, however, only when the intersection line is parallel to the half-space's bounding plane. If the intersection line moves behind (or, conversely, in front of) that plane at some time t, then all points on the line will become (or will stop being) valid intersection points. So the correct approach is to solve Equation 4.13 for t.

For the case of $\mathbf{q} = \mathbf{p}$, plugging in the definition of $\mathbf{p}$ from Equation 4.10 and $\mathbf{r}$ from Equation 4.11 gives

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

If $[\hat{\mathbf{r}} - \check{\mathbf{r}} - (\hat{\mathbf{p}} - \check{\mathbf{p}})] \cdot \mathbf{n}_H < 0$, then dividing by it gives a lower bound on t; this t is the time at which the intersection line moves behind the constraint half-space. Conversely, $[\hat{\mathbf{r}} - \check{\mathbf{r}} - (\hat{\mathbf{p}} - \check{\mathbf{p}})] \cdot \mathbf{n}_H > 0$ produces an upper bound on t. Either result is another half-plane in the t-k plane, one that says any value of (t,k) gives a valid intersection point $\mathbf{i}(t,k)$ as long as t is above (below) a specific value. If $[\hat{\mathbf{r}} - \check{\mathbf{r}} - (\hat{\mathbf{p}} - \check{\mathbf{p}})] \cdot \mathbf{n}_H = 0$ then $\hat{\mathbf{r}} - \check{\mathbf{r}} = \hat{\mathbf{p}} - \check{\mathbf{p}}$, that is, the intersection line and the half-space's bounding plane both move through time with the same velocity. Consequently, the intersection line is always either inside the half-space or outside it. Comparing $\mathbf{r}(0)$ against the bounding plane at $t = 0$ determines if the constraint is ever satisfied. Everything in this paragraph applies by analogy when $\mathbf{q} = \mathbf{c}$.

The result of all this work 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, the one with the lowest t coordinate. Finding this point is an example of a two-variable linear-programming problem. Preparata and Shamos [PS85] describe a solution to this problem, but I chose not to implement their algorithm. Instead, I extended my implementation of the Bentley-Ottmann algorithm [BO79, PS85] to solve this problem. The necessary extensions are quite straightforward. Whenever the Bentley-Ottmann algorithm reports that two segments have intersected, the algorithm must consider what happens to the two half-planes bounded by those segments. There are four cases, two in which the two half-planes start intersecting, and two in which they stop intersecting. For completeness, the remainder of this section describes these cases.

Consider what can happen when two half-plane-bounding segments, s_1 and s_2, intersect at $t = t'$. Without loss of generality, assume s_1 is above s_2 just before the intersection. If s_2 marks the upper

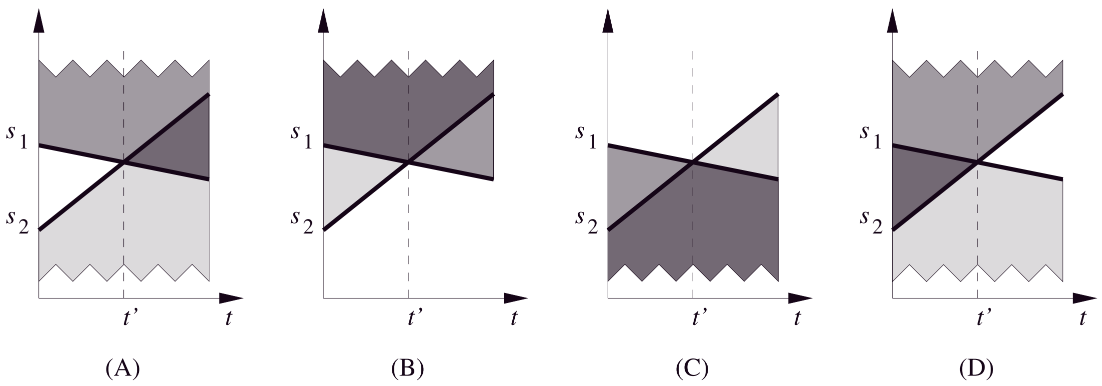


Figure 4.21: The four cases for half-plane intersections.

bound of a half-plane, H_2, and s_1 marks the lower bound of a half-plane, H_1, as in Figure 4.21(A), then the intersection of s_1 and s_2 is the first point in $H_1 \cap H_2$. Note that any points on s_1 or s_2 with $t > t'$ will also be in $H_1 \cap H_2$. The algorithm remembers this fact by adding H_1 and H_2 to the *membership* lists of s_1 and s_2. Each segment has such a list, which records the half-planes in which the segment currently lies. Since the maximum number of half-planes is 13 (six each from two hypertrapezoids and one from a cutting plane), the algorithm can store the membership list efficiently as a bit vector. If the combined memberships of s_1 and s_2 include all the half-planes, then t' is the solution to the linear-programming problem.

If s_2 marks the lower bound of half-plane H_2, then the situation is different; see Figure 4.21(B). As of $t = t'$, all the points on s_1 are no longer in H_2 (the fact that H_2 has as linear boundary means it can never "curl around" and engulf s_1 again). Thus, the algorithm updates the membership of s_1 so that it is no longer in H_2. If s_1 and s_2 are both upper-bounds, the situation is analogous; see Figure 4.21(C). In this case, s_2 has left H_1, and the algorithm updates its membership accordingly.

A final case occurs if s_1 an upper bound and s_2 is a lower bound, as depicted in Figure 4.21(D). As of this intersection, there can be no points that lie in both H_1 and H_2. Thus, the algorithm can immediately conclude that there is no solution to the linear-programming problem.

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

4.2.7 Plane-Plane Intersections

The final category of space-time-bound intersections from Section 4.2.1 involves two cutting-planes, say P_1 and P_2 from space-time bounds including hypertrapezoids T_1 and T_2, respectively. This situation is almost identical to the situation involving a face and a plane from the previous section. An argument like the one at the beginning of that section shows that P_1 can intersect P_2 only if faces of T_1 and T_2 have intersected and have been cut off. By a similar argument, it follows that an intersection between two cutting planes will always have a higher t coordinate than a valid intersection between a face and a cutting plane. Equation 4.12 characterizes the intersection between P_1 and P_2 if $\mathbf{m}$ and $\mathbf{r}(t)$ are computed from the normals and plane constants of P_1 and P_2.

```
1   $t_i \leftarrow \infty$
2   for $\alpha \in \{x, y, z\}$
3     while ((intersection method says $f_1$-$f_2$ is next intersection in $F_\alpha$, at $t = t'$) and
4            $(t' < t_i)$) {
5          for $j \in \{1, 2\}$ {
6            $T_j \leftarrow$ hypertrapezoid containing $f_j$
7            $P_j \leftarrow$ cutting plane associated with $T_j$'s space-time bound
8          }
9          if ($f_1$-$f_2$ intersection is not cut off by $P_1$ or $P_2$)
10           $t_i \leftarrow t'$
11         else {
12           intersected $\leftarrow$ FALSE
13           if ($f_1$ intersects $P_2$, satisfying constraints from $T_1$, $T_2$ and $P_1$ at $t = t''$)
14             intersected $\leftarrow$ TRUE
15           if ($f_2$ intersects $P_1$, satisfying constraints from $T_2$, $T_1$ and $P_2$ at $t = t'''$)
16             intersected $\leftarrow$ TRUE
17           if (intersected)
18             $t_i \leftarrow \min\{t_i, t'', t'''\}$
19           else
20             if ($P_1$ intersects $P_2$, satisfying constraints from $T_1$ and $T_2$ at $t = t''''$)
21               $t_i \leftarrow \min\{t_i, t''''\}$
22         }
23    }
```

Figure 4.22: This algorithm sets t_i to the time of the earliest intersection between space-time bounds.

The discussion of constraint half-spaces still holds, the only change being that the algorithm must consider at total of twelve half-spaces: six for T_1 and six for T_2.

4.2.8 Summary

Now that all the cases have been considered, Figure 4.22 presents the pseudocode for finding the simulation time, t_i, of the earliest intersection between space-time bounds. A few details of the pseudocode deserve comment. The calls to the intersection method at line 3 return face intersections in order of increasing t. At line 4, the "while" loop terminates if it finds an actual face intersection at simulation time greater than the current estimate for t_i; in this situation, there is no reason to continue processing the current F_α because the algorithm is looking for the earliest intersection time. Lines 12 through 18 use the variable "*intersected*" to indicate if either f_1 intersects P_2 or f_2 intersects P_1; it is important for the code to check both cases so it can find the intersection with the lower t coordinate.

Reviewing the entire discussion on intersections between space-time bounds, it is clear that cutting planes make the algorithms more complicated. Intuitively, cutting planes seem worth the trouble because they can significantly tighten space-time bounds. When space-time bounds are not tight, the broad phase of the detection algorithm can waste time processing intersections that have little to do with actual collisions. It would be interesting to investigate the significance of this wasted time in empirical test, but I have not conducted these tests.

```
/* Variables t_prev and overlap_possible persist between calls. */
/* Before first call, t_prev is initialized to t_0. */
/* Before first call, overlap_possible is initialized to FALSE . */
broad_phase(t_curr, {O_0, ..., O_{N-1}})
    t ← t_prev
    while (t ≤ t_curr) {
            t_build ← t
            rebuild space-time bounds for {O_0, ..., O_{N-1}} as of t_build
            if (space-time bounds intersect at t_i) {
              t ← unnormalize(t_i)
              if (t − t_build < Δt_d) {
                overlap_possible ← TRUE
                if (any objects whose space-time bounds intersected
                    also have intersecting bounding spheres as of t) {
                  t_prev ← t
                  add each pair of intersecting bounding spheres to result
                  return result so the narrow-phase algorithm can take over
                }
                t ← t_build + Δt_d
              }
              else
                overlap_possible ← FALSE
            }
            else
              t ← expiration time for space-time bounds
    }
    t_prev ← t
    return no collisions
```

Figure 4.23: A broad phase that uses space-time bounds.

4.3 Using Space-Time Bounds in the Broad Phase

This section describes how the algorithm from the previous section fits into the broad phase of the overall detection algorithm. The pseudocode for the broad phase appears in Figure 4.23.

Recall from Section 3.2 that the application calls the detection algorithm at the simulation time, t_{curr}, of the current frame. The broad phase has to check the N objects' bounding spheres for collisions. The previous time it stopped checking was t_{prev}, so it has to check the time interval from t_{prev} to t_{curr}. The broad phase starts by building a space-time bound for each object as of time t_{prev}. It uses M and $\mathbf{d}$ values supplied by the application, as Section 4.4 will discuss. The application will also indicate the amount of time beyond t_{curr} for which these values make sense, thus specifying the time at which these values and the space-time bounds built from them *expire*. After building the space-time bounds, the algorithm finds the time, t_i, of their earliest intersection, using the ideas from the previous section. That section worked in a *normalized* time scale starting at 0 and ending at 1. The broad phase, however, must work in the simulation time scale of the application, so it must unnormalize t_i to a value it calls t. Unnormalizing is a simple process, so I do not discuss its

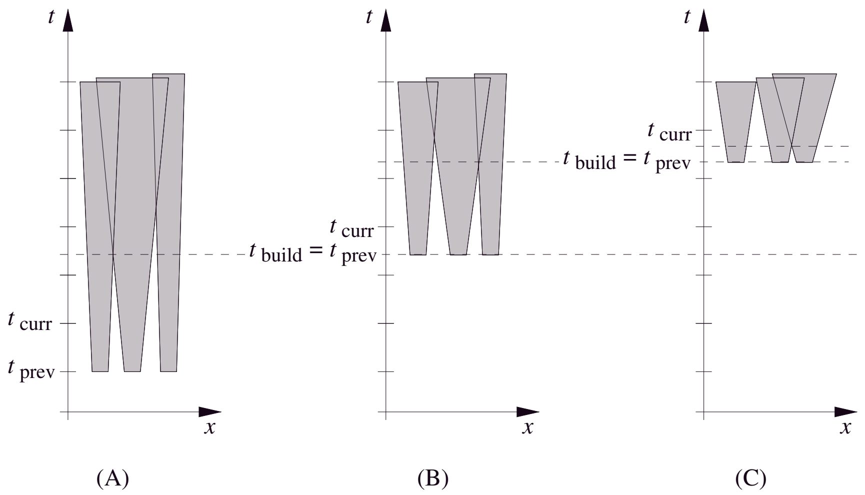


Figure 4.24: Three examples of the broad phase using space-time bounds.

details.

If the unnormalized time, t, of the earliest space-time-bound intersection is greater than the current frame time, t_{curr}, then there can be no collisions as of the current frame. Figure 4.24(A) illustrates this case. The same is true if the space-time bounds do not intersect at all before they expire. The broad phase is thus free to immediately return control to the application. In the pseudocode of Figure 4.23, this situation corresponds to skipping from line 8 to the end of the "while" loop and then immediately leaving the loop. Before returning, the algorithm saves t in t_{prev} as preparation for the next time it is called. The broad phase thus does no more work until $t_{\text{curr}} > t$, allowing it to avoid the fixed-timestep weakness.

When t_{curr} does finally exceed the earliest collision time saved in t_{prev}, then the algorithm sets t_{build} to t_{prev} and rebuilds the space-time bounds. The rebuilding uses new data provided by the application (i.e., M and $\mathbf{d}$) as of t_{build}. The algorithm then checks this new set of space-time bounds for intersections, obtaining a new t_i unnormalized to t. Hopefully, the new t is substantially larger than t_{build} so the algorithm can continue to avoid work. Figure 4.24(B) illustrates this case.

It is possible, however, that the new intersection occurs almost immediately. Figure 4.24(C) illustrates this case. The technical definition of "almost immediately" is $t - t_{\text{build}} < \Delta t_d$. The Δt_d in this expression is the *minimum temporal resolution* of the detection algorithm. If the detection algorithm finds two space-time-bound intersections within Δt_d of each other, it assumes that some pair of space-time bounds are in continuous contact with each other. In this situation, the pair of space-time bounds have provided as much accuracy as they can, and it is time for the broad phase to compare the bounding spheres of the objects associated with those space-time bounds. If the bounding spheres do intersect, then the broad phase has found a collision at the first level of detail. It returns the pairs of objects whose bounding spheres intersected (see lines 12 through 14 of Figure 4.23). Any more processing of these pairs is the responsibility of the narrow phase, which can use more levels of detail if the application decides processing time is available.

If no bounding spheres intersect, then the broad phase continues. It must advance to a time beyond the old t_{build} so it can rebuild and retest the space-time bounds. The time it advances to is $t_{\text{build}} + \Delta t_d$. The minimum temporal resolution of the broad phase is thus similar to the timestep in the basic algorithm of Figure 2.1 (hence the identical notation). The two algorithms will have the same temporal accuracy, but the broad phase uses space-time bounds to get that accuracy more efficiently. The application should choose the size of Δt_d. It should be small enough that users will not care if collisions of duration less than Δt_d are missed.

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

If the broad phase does step ahead to $t_{\text{build}} + \Delta t_d$, it must take an extra precaution when it next tests the space-time bounds for intersections. The algorithm from Section 4.2 assumes that the interiors of all the space-time bounds are initially disjoint (grazing contact between their surfaces does not matter). Stepping ahead to $t_{\text{build}} + \Delta t_d$ might violate this assumption: the two objects whose space-time bounds just touch at t_{build} could be closer together by $t_{\text{build}} + \Delta t_d$, making their space-time bounds penetrate at that time. The broad phase solves this problem by performing a special test before checking space-time bounds for intersection. This special test uses the faces in the set F_x from Equation 4.7. The x coordinates of these faces at (normalized) time 0 define a set of intervals, one for each space-time bound. If a pair of space-time bounds are to penetrate initially, the pair of corresponding intervals must intersect. The special test steps through the interval endpoints in ascending x-coordinate order, keeping track of those intervals that intersect and checking the corresponding space-time bounds for initial penetration. Line 9 of Figure 4.23 sets the flag *overlap_possible* to TRUE when the test is necessary, and the intersection algorithm in line 6 performs the test only when the flag is set. This test seems quite efficient in practice, in part because the projection method already sorts the interval endpoints in preparation for the Bentley-Ottmann algorithm [BO79, PS85]. Even so, this test could be slow in the worst case, and an alternative test that prunes the number of space-time bounds it considers would be valuable.

Line 13 of Figure 4.23 passes to the narrow phase every pair of objects whose space-time bounds intersect and whose bounding spheres intersect. It is important to realize that there can be several such pairs. The probability is small that more than one pair of bounding spheres will start intersecting at the same moment, but once a pair starts it may continue while other pairs start. This situation can occur if the pair do not intersect at the next level of detail; there will be no collision and thus no collision response, so the objects may not move apart enough to stop their bounding spheres from intersecting. Figure 4.25 shows an example. Once one pair of space-time bounds starts intersecting, the broad phase will start stepping ahead by Δt_d. Each time it does so, it will set the *overlap_possible* flag and invoke the special penetration test. This test must report all the pairs of space-time bounds that penetrate; it cannot stop at the first pair it finds.

An important part of the broad phase is the rebuilding of the space-time bounds (see line 5 of Figure 4.23). The broad phase could ask the application for $\mathbf{x}(t_{\text{build}})$, $\dot{\mathbf{x}}(t_{\text{build}})$ and the latest M and $\mathbf{d}$ for each object, and use this data to rebuild *all* the space-time bounds. This approach is correct, but it is not always the most efficient strategy. An alternative strategy is to rebuild only the space-time bounds that intersected last time. The broad phase can continue using the other space-time bounds until they expire, with only minor modifications: their early portions must be chopped off so that all the space-time bounds start at the same time coordinate, a requirement for

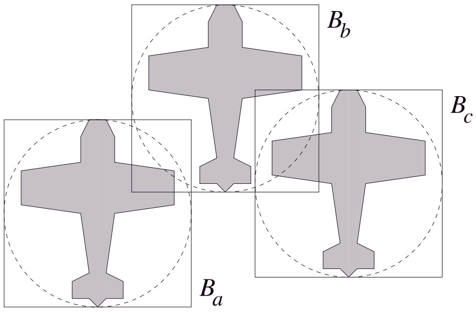


Figure 4.25: The bounding spheres inside space-time bounds B_a and B_b intersect, as do those inside B_b and B_c, but no objects actually collide. Note that this diagram shows only the cross-sections of B_a, B_b and B_c as of the current time.

the the face intersection algorithm from Section 4.2. The advantage of this strategy is the processing time saved by not completely rebuilding those other space-time bounds. This strategy does have a potential disadvantage. A space-time bound is conservative, and at t coordinates near its expiration time it could bound a large amount of space. As the current simulation time, t_{curr}, approaches the expiration time, "false-alarm" intersections are more likely. It is valuable to minimize the number of false alarms because any time the broad phase spends processing them is wasted. Empirical evidence suggests that this disadvantage is not significant, however; in pilot studies for the tests in Section 7.1, the time saved by not rebuilding all the space-time bounds outweighed the time wasted on false alarms.

4.4 Acceleration Bounds

The development of space-time bounds in this chapter has assumed that the application can supply the upper bounds on acceleration—M and $\mathbf{d}$, and their expiration time—for each object in the animation. It makes sense for the application to provide this information because it will have the most insight into how the objects might behave over the immediate future. This section discusses some issues related to these upper bounds, and gives some examples of how particular applications might estimate the bounds.

Since M and $\mathbf{d}$ are only upper bounds and do not need to be exact, the application can approach them with some flexibility. Tight bounds on M and $\mathbf{d}$ will yield more accurate space-time bounds, but even loose bounds should help the broad phase address the fixed-timestep weakness. If estimating tight bounds is computationally expensive, the application might get better overall performance from looser bounds.

No matter what acceleration bounds the application chooses, it has the freedom to *cancel* them if it realizes they are incorrect. When the application cancels the most recent set of bounds, it tells the broad phase to act as if the expiration time for the previous set of bounds has suddenly become the current time, t_{curr}. The broad phase will therefore not advance past t_{curr} without asking the application for a new, corrected set of bounds. Cancellation is effective only if the application actually notices when acceleration bounds become incorrect. To this end, the application must monitor the acceleration of each object. This additional processing should take a negligible amount of time in most applications.

The broad phase might ask the application for an updated set of acceleration bounds as of any simulation time (see line 5 of Figure 4.23). If the previous set of bounds still hold (i.e., if the objects have not exceeded them), then the application is free to supply them once again. The new space-time bounds that the broad phase builds will be an improvement, because even though they use old values for M and $\mathbf{d}$, they do incorporate the objects' latest positions and velocities.

The application should be careful about reusing the previous bounds, however, as doing so can sometimes reduce performance. The problem can arise because the expiration time of the whole set of space-time bounds is the earliest expiration time of any objects' acceleration bounds; this choice is necessary because the algorithm from Section 4.2 assumes all space-time bounds cover the same time period. Say, for example, an application knows the acceleration bounds for O_0 from time 0 to time 11, and for all the other objects from time 0 to time 10. If time 10 arrives without any collisions, the broad phase will ask the application for new acceleration bounds. Assume the application could now determine the acceleration bounds for all objects from time 10 to time 20. The application does not really need to recompute the bounds for O_0, since the known bounds have not yet expired. But reusing these bounds means the earliest expiration time would be at most one time unit later, at time 11. In this situation, the acceleration bounds are *unsynchronized.* The application would do better to go ahead and recompute the bounds of O_0 to get the later expiration time.

This section now turns to some examples of how applications compute acceleration bounds. The first example is a simple spaceship flight simulator. (The performance of this application is the subject of Chapter 7.) In this simulator, a spaceship has two controls, a throttle and a rudder. The throttle controls the thrust provided by the ship's main rocket engine. A particular setting of the throttle gives the ship a constant acceleration forward, in the direction of the ship's main axis. The rudder controls booster rockets that dictate the ship's orientation. A rudder setting determines how much the ship turns, climbs or dives. More specifically, it specifies an axis for the ship's rotation and an angular speed for that rotation; a more realistic model would let the rudder control angular acceleration, but that model is too difficult for casual users to control. When the rudder is in its neutral position, the ship retains its most recent orientation. At a neutral throttle setting the ship has no acceleration but it continues traveling forward at its most recent velocity. This model is quite simple, but it gives interesting effects reminiscent of what one expects from a spaceship.

The simulator computes a ship's M from the acceleration provided by the main engine. The simulator can set M to the magnitude of the current acceleration plus some extra "slack" value, preventing M from immediately becoming invalid if the user is increasing the acceleration. For $\mathbf{d}$, the simulator can use the current direction of the ship's main axis. This value will be valid until the user's subsequent rudder settings make the main axis rotate more than 90 degrees. By guessing that the ship will continue rotating at its current rate, the application can approximate the length of time before $\mathbf{d}$ expires. If the simulation notices that any of these guesses become incorrect, it should immediately cancel the current M and $\mathbf{d}$ and compute new values.

Vehicles like airplanes and automobiles have a different form of dynamics. As they turn, these vehicles have centripetal acceleration, directed towards the center of the turn. If the linear velocity of the vehicle is momentarily constant with magnitude $|\dot{\mathbf{x}}|$, the acceleration will have the instantaneous magnitude $|\dot{\mathbf{x}}|^2/R$, where R is the radius of the current turn [MH82]. The application can guess R from the current rudder settings of the vehicle, and as long as this guess is an underestimate then $M = |\dot{\mathbf{x}}|^2/R$ will be an upper bound. Since the acceleration will be directed from the vehicle to the center of the turn, the application can use this vector for $\mathbf{d}$. As with the spaceship model, the application can estimate the expiration time for M and $\mathbf{d}$ from the current rate of turning, and these values can be canceled if necessary.

Another important class of interactive applications allow users direct manipulation of objects.

For example, a modeling program might allow a user to select objects with a mouse cursor and "drag" them into positions that are not prohibited by collisions. It is difficult for an application to compute acceleration bounds for a directly manipulated object because the application does not have any explicit, differentiable formula for object's motion; the motion is dictated by the whims of the user. One solution is for the application to use a numerical approximation to the current acceleration; the finite difference method [PTVF92] follows from the definition of the derivative, and allows the application to estimate acceleration based on recent values of position. As an alternative solution, the application could slightly modify the meaning of direct manipulation so that an function does determine the object's acceleration. Instead of equating the position of the object to the position of the mouse cursor, the application could introduce a simulated spring between the cursor and the object. A user's cursor movements now apply forces (accelerations) to the object, with the relationship between cursor position and force being determined by Hooke's law [MH82]. This spring adds a little "lag" and "looseness" to the direct manipulation, but these effects smooth out the "jerkiness" in the object's motion, a result which is sometimes desirable.

The task of predicting acceleration bounds might be easier if the application uses a statistical prediction technique such as a Kalman filter. As Chapter 3 describes, several researchers have used Kalman filters to predict motion for the purpose of reducing input-device latency [LSG91, FSP92, AB94]. An interesting topic for future research is determining whether similar filters can make short-term predictions of acceleration magnitude and direction.

Chapter 5

Sphere-Trees

The previous chapter discussed how the broad phase efficiently handles the temporal aspect of collision detection. Space-time bounds allow the broad phase to avoid the fixed-timestep and all-pairs weaknesses while finding the simulation times of approximate collisions. These approximate collisions involve the most conservative approximations to the three-dimensional (3D) surfaces of the objects. Improving the spatial aspect of collision detection—determining if more accurate approximations also collide at the specified instant in time while avoiding the pair-processing weakness—is the task of the narrow phase.

This chapter introduces the key data structure of the narrow phase, the *sphere-tree*. As Section 3.2 mentions, the narrow phase allows an application to meet performance goals by trading accuracy for speed. The key to this process of negotiated graceful degradation is approximating the surface of each object at multiple levels of detail. This chapter begins by describing how sphere-trees provide this capability by organizing sets of 3D spheres into a hierarchy. The chapter next shows how the narrow phase uses sphere-trees. The remainder of the chapter discusses algorithms that build sphere-trees automatically. These algorithms strive to make each level of detail in a sphere-tree resemble the object as closely as possible, to allow the narrow phase to approximate collisions as accurately as possible. The first algorithm is based on the *octree* data structure, which represents a form of recursive subdivision. The next algorithm uses the more sophisticated technique of *simulated annealing*, a general approach to solving constrained optimization problems. This algorithm uses solutions to some interesting subproblems, such as determining whether a union of shapes completely cover a two-dimensional (2D) region, and measuring the distance from the surface of a sphere to the surface of a polyhedron. Chapter 6 will discuss another even more advanced (and more successful) algorithm for building sphere-trees, one based on *medial-axis surfaces*.

5.1 Approximating Objects with Spheres

The goal of approximate collision detection is to detect intersections between simplified representations of objects' surfaces. One of the simplest shapes to check for intersection is the sphere. A pair of spheres intersect if the distance between their centers is less than the sum of their radii. Figure 5.1 illustrates this fact for circles, which are 2D "spheres." For spheres centered at $\mathbf{c}_1$ and $\mathbf{c}_2$ with radii r_1 and r_2, one way to implement this test is to check for $|\mathbf{c}_1 - \mathbf{c}_2| < r_1 + r_2$. However, $|\mathbf{c}_1 - \mathbf{c}_2| = \sqrt{(\mathbf{c}_1 - \mathbf{c}_2) \cdot (\mathbf{c}_1 - \mathbf{c}_2)}$, so a more efficient approach is to check for $(\mathbf{c}_1 - \mathbf{c}_2) \cdot (\mathbf{c}_1 - \mathbf{c}_2) <$

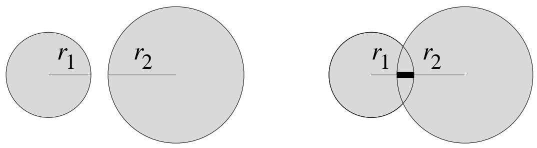


Figure 5.1: Spheres intersect if and only if the distance between their centers is less than the sum of their radii.

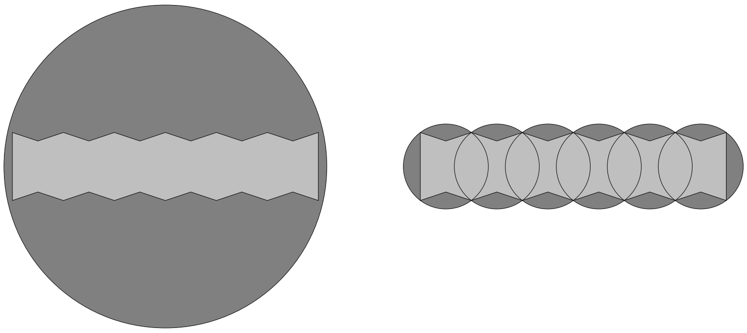


Figure 5.2: The union of six circles on the right gives a better approximation than the single circle on the left.

$(r_1 + r_2)^2$, a very fast computation.

A single sphere does not accurately approximate the shape of many objects. A union of partially overlapping spheres, however, can give a more accurate approximation. Figure 5.2 demonstrates this idea with a simple 2D example. If each object is covered by a union of spheres, a straightforward algorithm to detect approximate intersections between a pair of objects operates as follows: check for intersection between each pair of a sphere from one union and a sphere from the other union, and report any intersections found.

In general, a union involving many spheres can fit the contours of an object more closely than a union involving few spheres; the subsequent sections of this chapter and the following chapter explore ways to generate a close fit. The straightforward algorithm from the previous paragraph will thus give more accuracy if it processes unions of many spheres. On the other hand, the algorithm does more work when the unions contain more spheres. The number of spheres in the union thus determines the balance between speed and accuracy. Associating each object with multiple unions, some with fewer spheres and some with more, is thus a way to provide the multiple levels of detail needed for graceful degradation in the narrow phase.

To make graceful degradation most effective, the levels of detail must be related. More specifically, when the narrow phase detects collisions using one level of detail, it needs to be able to reuse that work to more quickly detect collisions at the next level of detail. The narrow phase can achieve this goal if it uses a *hierarchy* of sphere unions. Figure 5.3 shows an example of a three-level hierarchy. The root of the hierarchy—by convention, level zero—is the bounding sphere for the object, sphere 0 in the figure. The first level of the hierarchy is the union containing the smallest number of spheres. The subsequent levels are the other unions in order of increasing size. Each sphere from the union at level j has as its children a subset of the spheres in the union at level $j + 1$. There is no reason why every parent must necessarily have the same number of children, although an upper limit makes the data structure for a sphere-tree node simpler.

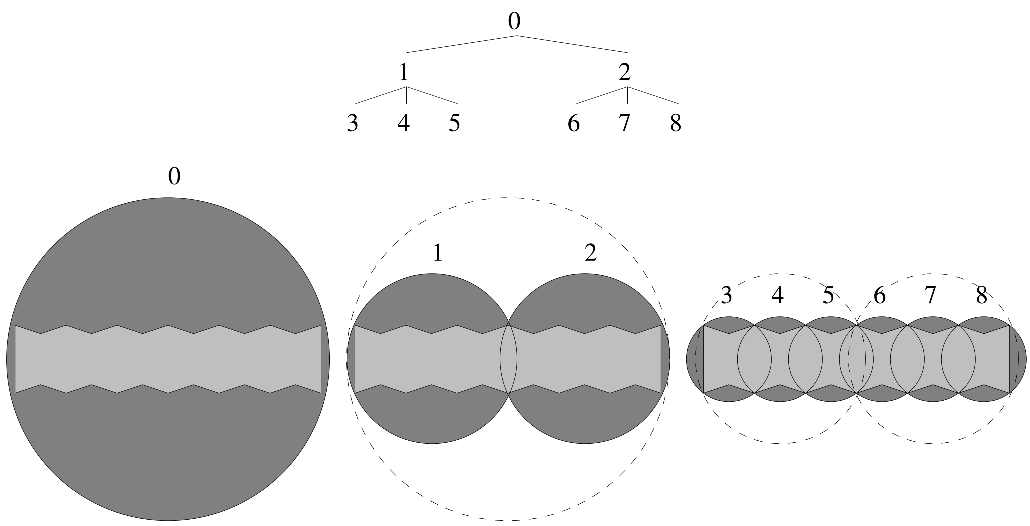


Figure 5.3: A three-level hierarchy. For each level, the previous level is superimposed with dashed lines.

It is important, though, that the children of each parent cover all parts of the object that the parent covered. Figure 5.3 shows this feature by superimposing (in dotted lines) the parents of each set of children. This feature is what makes the hierarchy useful to the narrow phase. Say the narrow phase is checking two objects' hierarchies for intersection. If the actual surfaces of two objects intersect at a region, ρ, and ρ is contained in a pair of intersecting spheres from level j of the hierarchies, then the children of these spheres will completely contain ρ. The narrow phase thus needs to check only these children for intersection. This policy guarantees that the narrow phase finds intersecting pairs of spheres at level $j+1$ that contain all of ρ, keeping the narrow phase conservative. By avoiding any spheres at level $j+1$ whose parents did not intersect, the narrow phase prunes the number of sphere pairs it must test, making it more efficient.

As long as the children of a sphere cover all parts of the object that it covers, the children are allowed to cover more of the object. In Figure 5.3, for example, sphere 5 covers part of the object that is outside its parent, sphere 1. This coverage is redundant, however, because the area outside the parent will already be covered by one of the parent's siblings and thus by that sibling's children. The hierarchy will therefore be more efficient if children cover as little outside their parents as possible. On the other hand, it is possible that the children can more tightly fit the parts of the object their parent covers if they also cover a little area outside the parent; in this situation, the extra coverage is worthwhile.

This hierarchy of sphere unions is a *sphere-tree*. The idea of hierarchy has a long history in computer graphics; Samet and Webber [SW88a, SW88b] give an overview of some hierarchical data structures and algorithms. Sphere-trees have some novel features, however, which will be developed in this chapter and the next.

Figure 5.4 presents the pseudocode for a narrow phase that uses sphere-trees. This routine takes a parameter, S, which is the set of sphere pairs that intersected at the previous level of detail. Passing an empty S causes the routine to detect intersection at the first level of detail, the bounding spheres. As Chapters 3 and 4 explain, the broad phase already detects collisions between bounding spheres; the pseudocode in Figure 5.4 retains the case for bounding spheres to make it more general,

```
        /* Call with S = ∅ for first level of detail. */
        /* Call with S = previous result to refine. */
1       narrow_phase(S)
2       {
3          result ← ∅
4          if (S = ∅)
5             if (root of sphere-tree 1 intersects root of sphere-tree 2)
6                put (root of sphere-tree 1, root of sphere-tree 2) into result
7          else
8             for (s1, s2) ∈ S {
9                if (s1 and s2 have children)
10                  for each child s1' of s1
11                     for each child s2' of s2
12                        if (s1' intersects s2')
13                           put (s1', s2') into result
14               else
15                  put (s1, s2) into result
16            }
17         return result
18      }
```

Figure 5.4: A narrow phase that uses sphere-trees.

for example, to support applications that feature only two objects. The pseudocode in Figure 5.4 uses the children of the pairs in S to find and return the intersecting pairs at the next level of detail. The application can thus invoke this algorithm repeatedly to refine the accuracy of collision detection. When the application can devote no more time to collision detection, it stops calling the algorithm and moves on to the next task. Note that algorithm returns only when it has completely refined the current level of detail. As a variation, this algorithm could return when it refined part of the level, essentially breaking down the "for" loop at line 8 of Figure 5.4. This variation would give the algorithm more control over the time spent on collision detection.

The narrow phase in Figure 5.4 does not build the sphere-trees that it uses. Instead, it assumes that the sphere-trees have been built in a preprocessing algorithm that runs before the application starts. This separation of labor is important for the efficiency of the narrow phase. The narrow phase is faster if it uses sphere-trees that fit objects tightly, but automatically generating tight-fighting sphere-trees can be expensive, so the generation should not be part of the narrow phase itself. Algorithms to automatically generate sphere-trees are the topic of following sections of this chapter and the next chapter. As Section 3.2 mentions, building sphere-trees in advance works only for rigid objects or objects consisting of articulated rigid components. A running application changes a rigid object by applying rotations and translations to it. If the application applies the same rotations and translations to its sphere-tree (or, more efficiently, to only the children of those spheres involved in intersection at the previous level), then the sphere-tree will always match the object and cover it conservatively. This idea applies to objects with articulated rigid components if each component has its own sphere-tree. The narrow phase in Figure 5.4 can process rotated and translated sphere-trees efficiently because spheres are rotationally invariant. Hierarchies composed of shapes other than spheres do not necessarily have this property. An *octree*, for example, is a hierarchy composed of isothetic 3D cuboids (which relate to cubes as rectangles to squares). Checking a pair of isothetic cuboids for intersection is essentially as efficient as checking a pair of spheres. Once the cuboids have been rotated, however, they are no longer isothetic and checking them for intersection becomes significantly more difficult and expensive. A narrow phase that approximates objects with octrees thus would not be nearly as efficient as one that uses sphere-trees.

A small modification allows the narrow phase in Figure 5.4 to use *full* accuracy as its final level of detail. The ideas is that the algorithm that builds a sphere-tree also stores, in each leaf sphere (i.e., each sphere at the deepest level), the part of the object's surface that is contained in that sphere. If, for instance, the object is a represented visually by a polyhedron, the leaf spheres store polygons. When the algorithm in Figure 5.4 finds a leaf sphere that intersects a leaf sphere from another sphere-tree (see line 15), the algorithm could compare all the polygons in one sphere with all the polygons in the other sphere, reporting any intersecting pairs of polygons. This approach should be efficient if leaf spheres are relatively small, because on the average there should be few polygons in each leaf sphere. If leaves contain more than a few polygons, the algorithm could use a more sophisticated variation that does not always compare all pairs of polygons; an approach analogous to the subdivision method from Section 4.2.4 is one possibility. Interactive applications may never need the full accuracy this modification can provide, but it is valuable to have it available if needed.

This chapter now turns to the problem of the preprocessing algorithm that automatically builds sphere-trees. The literature contains little in the way of relevant previous work. Youn and Wohn [YW93] discuss how hierarchies of simple shapes like spheres can approximate the surfaces of objects, but they do not describe how to build these hierarchies for arbitrary objects. Goldsmith and Salmon [GS87] describe how to build a hierarchy of bounding volumes given a set of leaf volumes. Their approach is interesting because it involves heuristics that optimize the quality of the hierarchy for their particular application, intersecting light rays with objects to simulate realistic images. The lack of a method for specifying leaf volumes means that this method is not directly applicable for building sphere-trees, however. Liu *et al.* [LNA88] present a way to build a hierarchy like a sphere-tree by fitting children to regular polyhedra inscribed in the parent sphere. The next section of this chapter explores a related idea in more detail. The most interesting previous work is that of Badler, O'Rourke and Toltzis [BOT79, OB79]. They describe a greedy approach to making a union of spheres fill the volume inside an object. One application of this approach is collision detection for models of the human body, although in this application they authors report needing some human intervention to obtain a desirable configuration of spheres. The authors do not consider the idea of arranging the unions into a hierarchy to support graceful degradation, but they pay more attention than other authors to the issue of making the spheres fit the object tightly. Chapter 6 discusses at length an algorithm that shares some concepts with their approach but that provides several significant advantages. The last section of that chapter compares the two approaches in detail.

5.2 Building Sphere-Trees from Octrees

An *octree* is a data structure that represents a recursive subdivision of 3D space. Octrees are based on the same principle as the subdivision method for finding intersections between hypertrapezoid faces, from Section 4.2.4. Section 2.3.2 explained how some algorithms use octrees to detect collisions between polyhedral objects. The current section describes how an octree subdivides the space occupied by an object, and how the resulting subdivisions can dictate the positions of spheres in a sphere-tree. The specific details of this section pertain only to objects whose surfaces are polyhedra. It would be possible to extend the general ideas to objects with other types of surfaces, but these extensions are beyond the scope of this dissertation.

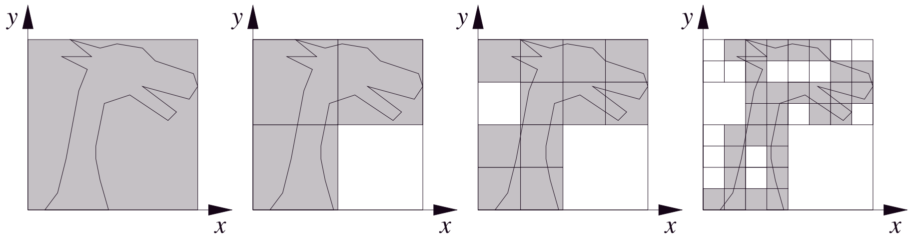


Figure 5.5: Four levels of an octree for a 2D object. The shaded squares are the nodes. The lines at each level show the subdivision of the previous level.

The literature on octrees is extensive. As the summaries by Samet and Webber [SW88a, SW88b] indicate, there are several different versions of octrees. For this section, the nodes of an octree are isothetic cuboids. Each cuboid stores the pieces of the object's surface that intersect it. If the object is represented by a polyhedron, the pieces of its surface are polygons. The algorithm to build an octree for a polyhedral object operates as follows. It first sets the root of the octree—by convention, level zero—to be the smallest isothetic rectangle that bounds all the polyhedron. To build the children of the root, level one of the octree, the algorithm subdivides the cuboid into eight isothetic octants. The algorithm associates with each octant the polygons that intersect the octant. If an octant contains any pieces of polygons, then it becomes a child of the root node; if it is empty of polygons, then it is not needed and the root has one less child. To build the next level of the tree, the algorithm recurs on each child of the root. The algorithm can continue this process indefinitely, so an octree can have an arbitrary depth. Figure 5.5 illustrates this process in two dimensions, with an object consisting of line segments instead of polygons.

The difficult part of this algorithm is the step that determines what polygons lie in each child octant. This step involves clipping polygons against the isothetic midpoint planes that define the child octants. The subdivision method for finding hypertrapezoid face intersections, from Section 4.2.4, has a similar step. In one sense, this step of the octree algorithm is simpler, because it does not need to deal with the fourth dimension. Unlike hypertrapezoid faces, however, the 3D polygons that define an object's surface do not necessarily have cross sections that are isothetic, so the octree algorithm must deal with more cases. Laidlaw *et al.* [LTH86] describe in detail how to find the intersection between two arbitrarily aligned polygons. Clipping a polygon against a midpoint plane involves almost the same steps, with a few simplifications because the plane is not arbitrarily aligned but isothetic. If the polygons are triangles, a few more simplifications are possible [Hub90].

Given an octree for an object, a straightforward way to build a sphere-tree is to put a sphere around each node of the octree. Finding these circumscribing spheres is simple since each octree node is an isothetic cuboid. The sphere-tree maintains the same parent-child relationships as the octree. A set of children are guaranteed to cover everything their parent covers because the spheres circumscribe the cuboids. The root sphere is actually a special case, because its children will cover what it covers as long as it bounds the whole object. Thus, it need not be as large as the sphere that circumscribes the root cuboid of the octree; Ritter [Rit90] describes a simple algorithm that finds a tighter bounding sphere. Figure 5.6 shows the sphere-tree corresponding to the octree from Figure 5.5. Note that two regions inside the object are not covered by spheres. The octree does not have nodes in these regions because its nodes must contain part of the object's surface, not its volume. For another object to reach one of these uncovered interior regions, it would have to pass through this object's surface, which is covered. The conservative nature of space-time bounds (Chapter 4) prohibit the other object from "jumping" through this object's surface into the uncovered region.

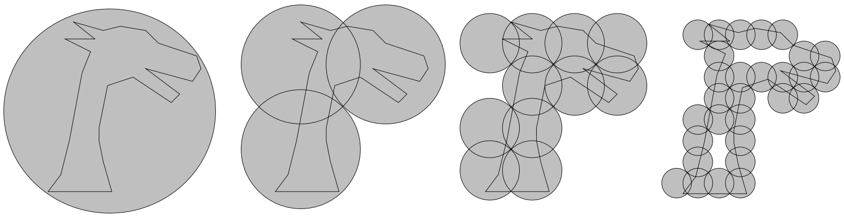


Figure 5.6: The octree from Figure 5.5 defines the sphere-tree whose four levels are shown here.

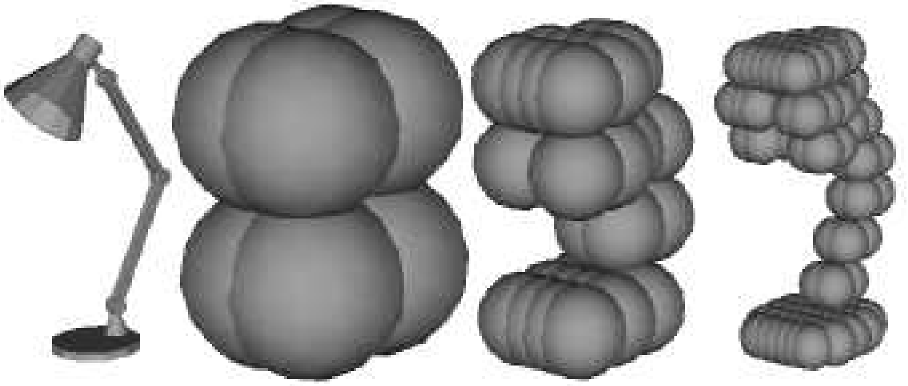


Figure 5.7: A desk lamp, and three levels of its octree-based sphere-tree.

In a sphere-tree based on an octree, the spheres at level $j+1$ have half the radius of the spheres at level j, and one-eighth the volume. (Treating the root with the special case from the previous paragraph violates this rule for level zero.) Thus, the "resolution" of the sphere-tree increases with each level. It is clear from Figure 5.6, however, that the spheres determined by an octree can fit the object quite loosely; this conclusion should be intuitively evident even without a precise definition of a sphere-tree's accuracy, which will appear in Section 5.3.

Figure 5.7 shows this "looseness" for a real 3D model. At the left of the figure is a desk lamp. This model is moderately complex, involving 626 triangular faces. Next to it are three levels of its sphere-tree, levels one through three (where the root is level zero). The spheres clearly reflect the regular subdivision of the octants on which they are based. The spheres do appear to fit the lamp more tightly at deeper levels of the tree, but the fit is still quite loose.

Building sphere-trees from octrees does have certain advantages. The octree algorithm builds this sphere-tree quickly, taking about two seconds on a Hewlett Packard 9000 Model 735. The algorithm also is relatively simple to implement. Intuitively, however, its seems that a different approach ought to yield tighter sphere-trees. The rest of this chapter and the following chapter explore approaches that do provide this advantage.

5.3 Building Sphere-Trees with Simulated Annealing

This section presents a second algorithm for building sphere-trees. This algorithm use *simulated annealing*, a general technique for solving constrained optimization problems [LA87, PTVF92]. The sphere-trees this algorithm produces can fit objects more tightly than sphere-trees produced by the octree method from the previous section. Building sphere-trees with simulated annealing does have disadvantages: it can be slow and unreliable. The algorithm from this section is important, though, because at its heart are techniques for perturbing an existing sphere-tree and checking whether this new sphere-tree covers the object and fits it more tightly. These techniques are useful in other algorithms for building sphere-trees, most notably the one from Chapter 6, which consistently produces good results in a reasonable amount of time. The current section, like Section 5.2, focuses on polyhedral objects, but the general ideas should extend to other classes of objects.

5.3.1 Basic Ideas

The motivation for simulated annealing comes from the thermodynamics of liquids [LA87, PTVF92]. The internal energy is lower in a liquid that cools slowly than in a liquid that cools quickly. Internal energy causes brittleness in a liquid that cools to a solid state. Noting this problem, metal workers increase the strength of a part cast from molten metal by cooling it slowly. This process of slow cooling is called *annealing*.

Annealing works because of what happens to the molecules in a liquid as it cools. At higher temperatures, the molecules rearrange themselves more than at lower temperatures. The rearrangement may temporarily increase the energy in the liquid, but it tends to prevent the liquid from getting stuck in local energy minima, allowing a lower energy in the long term. A liquid that cools slowly has more time for this rearrangement and so it is more likely to reach a low-energy state. A statistical model for this behavior is the *Boltzmann distribution*, which gives the probability of a particular energy as of a particular temperature. Under this inverse-exponential distribution, higher energies are more likely at higher temperatures, but there is always a chance of an energy increase at any temperature.

Simulated annealing is the application of the Boltzmann distribution in algorithms that solve minimization problems. An algorithm that uses simulated annealing creates an analogy between the problem it is trying to solve and a liquid being cooled: the quantity to be minimized corresponds to the internal energy of the liquid, and the variables that make up a solution to the problem correspond to the configuration of the liquid's molecules. To model the temperature of the liquid, the algorithm introduces a new control parameter, τ, which it gradually decreases as it searches for the minimum energy. Press *et al.* [PTVF92] calls the policy for decreasing τ the *annealing schedule*. The basic step of the algorithm involves randomly rearranging the current configuration, that is, the current estimate of the problem solution. The algorithm measures the energy of the new configuration, and chooses whether to accept it as the current solution or reject it with a probability determined by the Boltzmann distribution as of the current temperature, τ. More specifically, most algorithms use the *Metropolis criterion*, which says that the probability of accepting an energy change of ΔE at temperature τ is

$$\mathrm{Prob}(\Delta E, \tau) = \exp\left(-\frac{\Delta E}{k_B \tau}\right), \tag{5.1}$$

where k_B is a constant called the *Boltzmann constant*. The Metropolis criterion makes the algorithm more likely to accept energy-increasing rearrangements early on, when the temperature is high. By repeating this basic step a large number of times as τ gradually decreases, the algorithm should arrive at a solution close to the global minimum for the problem. This general approach seems to work well for many practical problems; Laarhoven and Aarts [LA87], for example, discuss several applications of simulated annealing to layout problems in VLSI design, and Press *et al.* [PTVF92] describe how simulated annealing generates an approximate solution to the traveling-salesman problem.

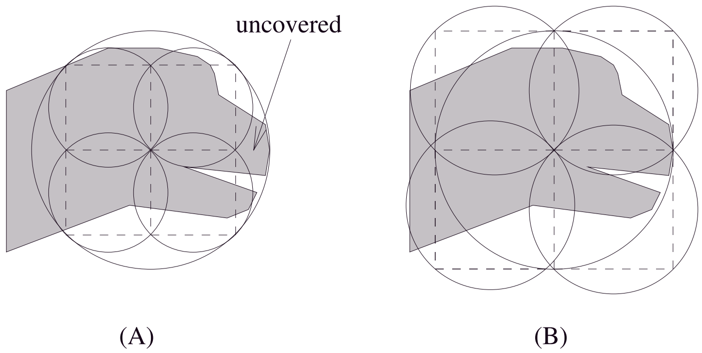


Figure 5.8: (A) These children leave uncovered part of what their parent covers. (B) The modified octree method overcomes this problem.

An algorithm for building a sphere-tree for an object can use simulated annealing to extend each node, that is, to configure a set of children so they tightly cover the part of object covered by the parent. The root has no parent, so it is just the object's bounding sphere, which can be computed using Ritter's algorithm [Rit90]. The children of the root are the first spheres to which the algorithm applies simulated annealing. The positions and radii of these spheres correspond to the configuration of molecules in the cooling liquid. To model the liquid's energy, the algorithm needs a quantity that gets smaller as the spheres more closely fit the part of the object they must cover. An appropriate quantity is the "looseness" of the spheres, that is, a measurement of how much of the spheres protrude outside the object; Section 5.3.2 will discuss measurements of "looseness" in detail. Minimizing this quantity is desirable because "loose" sphere-trees make the narrow phase less accurate. The algorithm begins by picking some initial configuration for the children of the root; a convenient choice is the eight spheres that the octree method from Section 5.2 creates. The algorithm then starts the annealing schedule and the repeated random changes to the configuration of children. When the annealing schedule is complete, the algorithm has determined positions and radii for each of the root's children. The algorithm can now recursively extend each of these children.

When extending a sphere that is not the root, the algorithm must be more careful about choosing the initial configuration of the children spheres. In particular, it cannot directly use the eight spheres that the octree method from Section 5.2 creates. The octree method assumes that the parent sphere circumscribes a cuboid, and that the parent is responsible for covering only the parts of the object within that cuboid; the children therefore need only cover that cuboid as well. Simulated annealing, however, can tighten the parent sphere around the object, so it can be responsible for parts of the object outside the cuboid it circumscribes; the children therefore may need to cover more than what the cuboid covers. Figure 5.8(A) illustrates this problem. A solution is to use a modified approach in which the children cover the cuboid that circumscribes the sphere, not vice versa. As Figure 5.8(B) shows, the resulting children are closer in size to their parent. The children thus do not initially provide as much of an improvement in resolution, but they are ready to be tightened by simulated annealing.

During the course of the annealing schedule, the algorithm repeatedly makes random changes to the configuration of the children. A configuration change consists of the algorithm picking one child and translating and scaling the child by small random amounts. After making such a change, the algorithm must check that the children still cover what their parent covers; this operation is the subject of Section 5.3.3. If the children do not maintain coverage, the algorithm undoes the current change, picks another child and tries another random change. The algorithm continues this process until it can make a change that satisfies the coverage constraint.

The annealing schedule itself is a nontrivial issue. The basic idea of what it does is straightforward: it picks an initial value for the temperature parameter, τ, and then gradually decreases τ until the minimization is complete. There are many ways to actually implement this idea, though. Laarhoven and Aarts [LA87] describe a large number of variations that are used in practice. For building sphere-trees, I chose a simple implementation. The initial value for τ should be high enough that most configuration changes will be accepted by the Metropolis criterion, Equation 5.1. One way to meet this goal is to compute the average change in energy over a number of initial trials, and then solve Equation 5.1 for the value of τ that makes this average change have a specific, high probability. In my experience, I found that 40 initial trials seem to be enough. Given an initial τ, the annealing schedule stays at that temperature for a certain number of changes to the configuration of children spheres. More specifically, it stays until it meets either an upper limit on the number of accepted changes or an upper limit on rejected changes. For both upper limits, it uses a value of 200. To actually change a sphere, my implementation translates the sphere in a random direction, with a magnitude of up to 3.5 percent of the sphere's parent's radius. It also scales the sphere by a random factor of between 0.95 and 1.053. When the number of changes reaches one of the upper limits, the annealing schedule decreases the temperature to a fraction of its current value, 90 percent in my implementation. The annealing schedule continues this pattern until it has iterated a specific number of times. Levels of the sphere-tree near the root seem to require more iterations to converge to an acceptable fit. I use 100 iterations for the first level (the children of the root), and 50 for subsequent levels. Other values for all these parameters are possible, and one weakness of simulated annealing in general is the amount of parameter-tweaking it requires. I have not found another set of parameters that gives better results for sphere-trees, though. Section 5.3.6 describes the results of the algorithm with this set of parameters.

5.3.2 Measuring Energy ("Looseness")

An important part of simulated annealing is the analogy to the cooling liquid's internal energy. This quantity is what simulated annealing minimizes. A simulated-annealing algorithm that builds a sphere-tree needs to minimize the "looseness" with which the sphere-tree fits an object, because "loose" sphere-trees make the narrow phase less accurate. This section describes an algorithm for measuring "looseness."

There are at least two ways the algorithm can quantify what "looseness" means for an individual sphere in sphere-tree. First, the algorithm can measure the volume inside the sphere that is outside the object being covered. Figure 5.9 gives an example. Second, the algorithm can measure the distance from the surface of the sphere to the object. To precisely define this distance, consider every point on the surface of the sphere outside the object; the distance from that point to the object is the *minimum* distance from the point to the surface of the object, and the *maximum* of these distances over all points is the distance from the sphere to the object. Figure 5.10 shows an example of this distance.

The definition of "looseness" for a set of spheres from a sphere-tree (e.g., the children of a parent) must be some aggregate quantity that involves each sphere in the set. The first definition of "looseness" for a single sphere, based on volume outside the object, does not lend itself to aggregation because the volume outside the object may be common to more than one sphere. It is simpler to use the definition based on distance. One idea is to sum the distances from each sphere in the set to the object, or average them. Another idea is to use the maximum of these distances. There are other possible choices, and reasons to believe that each might be the best quantity to minimize. In

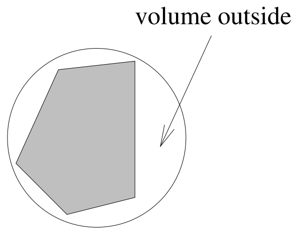


Figure 5.9: One measure of "looseness" is volume bound by the sphere that is outside the object.

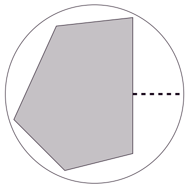


Figure 5.10: Another measure of "looseness" is the distance from the sphere to the object, as indicated by the heavy line.

my empirical tests, however, the maximum seems to give the best results; Section 5.3.6 will discuss this conclusion further.

To use any of the aggregate quantities, the algorithm needs a way to compute the distance from a sphere to an object. The definition of this distance does not directly lead to a tractable computation, since it involves iterating over an infinite number of points on the surface of the sphere. The distance from a sphere to an object is related to the distance from a point to the object. An algorithm for computing this latter distance can compute the distance from a sphere to the object exactly if the object is convex. The problem of computing the distance from a sphere to a nonconvex object is significantly more difficult. There appears to be no way to compute this distance exactly, but two approximations follow from an algorithm to compute the distance from a point to the object.

Distance from a Point to a Polyhedron

The first goal, thus, is an algorithm that computes the distance from a point $\mathbf{p} \in \mathbb{R}^3$ to an object whose surface is a triangulated polyhedron, convex or nonconvex. The simplest approach is as follows: for each face of the polyhedron, find the shortest distance from $\mathbf{p}$ to the face, and return the minimum over all faces. This algorithm takes time linear in number of faces, so it may not be the most efficient approach, but it is simple to implement.

The key to this approach is an algorithm that computes the minimum distance from $\mathbf{p}$ to a triangular face, f, of the object. Throughout this discussion, let $d(\mathbf{x}, \mathbf{y})$ be the Euclidean distance between any points $\mathbf{x} \in \mathbb{R}^3$ and $\mathbf{y} \in \mathbb{R}^3$. The algorithm must consider three cases. In the first case, $\mathbf{p}$ projects onto f. More specifically, the line through $\mathbf{p}$ parallel to f's normal vector intersects f at the point $\mathbf{p}'$. In this case, no other point on f is closer to $\mathbf{p}$ than the projected point, $\mathbf{p}'$. The proof follows from the Pythagorean theorem, as Figure 5.11 illustrates. Any other point, $\mathbf{q}$, on f forms a right triangle with $\mathbf{p}$ and $\mathbf{p}'$, and the leg from $\mathbf{p}$ to $\mathbf{q}$ must be longer than the leg from $\mathbf{p}$ to $\mathbf{p}'$. Thus, if the algorithm finds that $\mathbf{p}'$ lies inside f, it returns $d(\mathbf{p}, \mathbf{p}')$.

In the second case, $\mathbf{p}$ does not project onto f but it does project onto an edge, e, of f. More

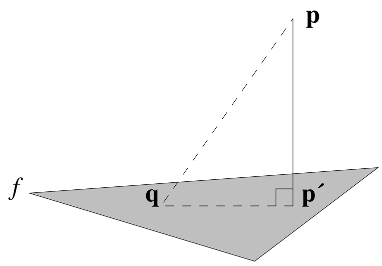


Figure 5.11: When $\mathbf{p}$ projects onto the face, f, $d(\mathbf{p}, \mathbf{p}') < d(\mathbf{p}, \mathbf{q})$.

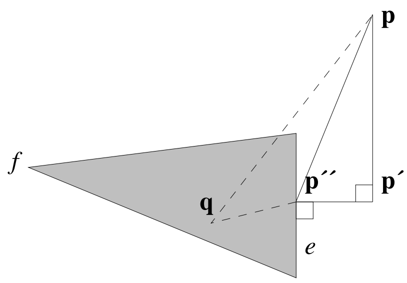


Figure 5.12: When $\mathbf{p}$ projects onto the edge, e, $d(\mathbf{p}, \mathbf{p}'') < d(\mathbf{p}, \mathbf{q})$.

specifically, let $\mathbf{p}'$ be the projection of $\mathbf{p}$ onto the plane of f, that is, the point where the line through $\mathbf{p}$ parallel to the plane's normal intersects the plane. The line in that plane through $\mathbf{p}'$ and perpendicular to e intersects e at $\mathbf{p}''$, as Figure 5.12 shows. Note that $\mathbf{p}'$ can project in this way onto only one edge of f. In this case, no other point on f is closer to $\mathbf{p}$ than point $\mathbf{p}''$. As proof, consider $\mathbf{p} - \mathbf{p}''$, the vector from $\mathbf{p}''$ to $\mathbf{p}$, and $\mathbf{q} - \mathbf{p}''$, the vector from $\mathbf{p}''$ to any other point $\mathbf{q}$ in f. Since the vector from $\mathbf{p}'$ to $\mathbf{p}''$ is perpendicular e, it follows that $(\mathbf{p} - \mathbf{p}'') \cdot (\mathbf{q} - \mathbf{p}'') \leq 0$. The law of cosines states that $d(\mathbf{p}, \mathbf{q})^2 = d(\mathbf{p}, \mathbf{p}'')^2 + d(\mathbf{p}'', \mathbf{q})^2 - 2(\mathbf{p} - \mathbf{p}'') \cdot (\mathbf{q} - \mathbf{p}'')$, so $d(\mathbf{p}, \mathbf{q}) \geq d(\mathbf{p}, \mathbf{p}'')$. Thus, the algorithm returns $d(\mathbf{p}, \mathbf{p}'')$ in this case.

The third case occurs when $\mathbf{p}$ projects onto neither f nor any edge of f. In this case, $\mathbf{p}$ is closer to a vertex, v, of f than to any other point $\mathbf{q}$ in f. Say $\mathbf{p}$ is closer to v than to the other vertices of f, and let $\mathbf{v}$ be the position of v. The proof that $d(\mathbf{p}, \mathbf{v}) < d(\mathbf{p}, \mathbf{q})$ for any other point $\mathbf{q}$ in f is similar to the proof of the previous case, as Figure 5.13 illustrates. Because $\mathbf{p}'$ does not project onto any edge of f, $(\mathbf{p} - \mathbf{v}) \cdot (\mathbf{q} - \mathbf{v}) \leq 0$. The rest of the proof follows from the law of cosines. The algorithm thus returns $d(\mathbf{p}, \mathbf{v})$, and it has covered all cases.

The overall algorithm, which computes the distance from $\mathbf{p}$ to the whole polyhedron, need not necessarily compute the distance from $\mathbf{p}$ to every face. Optimizations can cut down on the number of faces the algorithm must consider. The simplest optimizations use space subdivision. Say the algorithm has access to a 3D grid of isothetic cubes, with each cube referencing the faces that intersect its volume. Because the cubes are isothetic, the algorithm can quickly find the point, $\mathbf{p}_c$, closest to $\mathbf{p}$ on any such cube. If, for any cube, $d(\mathbf{p}, \mathbf{p}_c)$ is greater than the distance between $\mathbf{p}$ and a face the algorithm has already processed, then no part of any face inside the cube can be closest to $\mathbf{p}$. The algorithm can thus prune the faces it considers if it uses the cubes to organize its processing. More specifically, the algorithm starts in one cube and finds the closest face in that cube. The algorithm then advances to another cube to consider its faces, but the algorithm ignores the cube if it is farther away than the currently-closest face. This approach is analogous to *minimax*

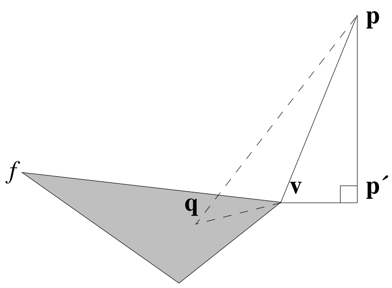


Figure 5.13: When $\mathbf{p}$ projects onto the vertex at $\mathbf{v}$, $d(\mathbf{p}, \mathbf{v}) < d(\mathbf{p}, \mathbf{q})$.

search, which is popular in artificial intelligence [CM86].

The order in which the algorithm visits the cubes affects the number of faces the optimization prunes. The algorithm should start in the cube containing $\mathbf{p}$, or the cube closest to $\mathbf{p}$. It should then process cubes in order of increasing distance from $\mathbf{p}$. This order allows the algorithm to terminate as soon as it finds one cube that is farther away than the currently-closest face. My implementation uses leaves of an octree as the set of cubes, and sorts the set by increasing distance from $\mathbf{p}$. A better implementation would avoid the full sort. It would conduct a breadth-first search of the grid, stopping as soon as the cubes at the search boundary are all farther away than the currently-closest face. Note that in all these approaches, a face can be in more than one cube. The algorithm should thus mark each face that it processes so it can avoid reprocessing. Even so, the time the algorithm spends just checking for the mark can accumulate and reduce the significance of the optimization. In practice, however, this accumulation does not seem to be a problem and the optimization is worthwhile.

Distance from a Sphere to a Convex Polyhedron

The algorithm to compute the distance from a point to the polyhedral surface of an object is important for computing the distance from a sphere to the object. For the case in which the polyhedral surface is convex, the following lemma expresses the key relationship.

Lemma 5.1 *Consider a sphere with center* $\mathbf{c}$ *and radius* r *that intersects the surface of a convex polyhedron. If* $\mathbf{c}'$ *is a point on the surface of the polyhedron closest to* $\mathbf{c}$*, then the distance from the sphere to the polyhedron is* $r + d(\mathbf{c}, \mathbf{c}')$ *if* $\mathbf{c}$ *is outside the polyhedron, and* $r - d(\mathbf{c}, \mathbf{c}')$ *if* $\mathbf{c}$ *is inside the polyhedron.*

Proof: A rigorous proof is surprisingly long and involved. Appendix A presents the details for the sake of completeness. □.

This lemma leads immediately to an algorithm that computes the distance from the sphere to the object. After computing the distance $d(\mathbf{c}, \mathbf{c}')$, all the algorithm must do is determine if the sphere's center, $\mathbf{c}$, is inside or outside the polyhedron. Since the polyhedron is convex, $\mathbf{c}$ is inside only if it is behind the planes of all the polyhedral faces, that is, if the dot product of each face normal with a vector from $\mathbf{c}$ to a point on the face is negative.

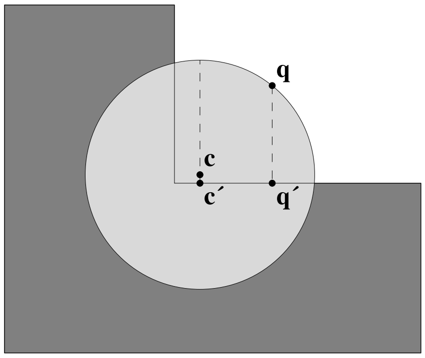


Figure 5.14: Lemma 5.1 does not work for nonconvex polyhedra.

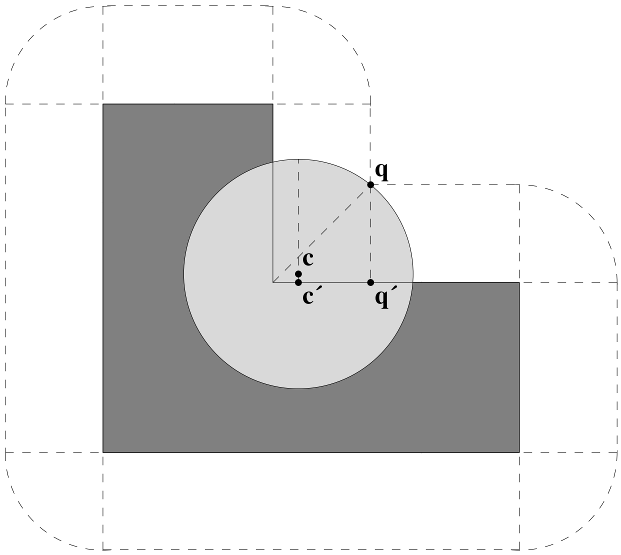


Figure 5.15: The push-off surface shows that the correct distance is $d(\mathbf{q}, \mathbf{q}')$.

Distance from a Sphere to a Nonconvex Polyhedron

Convex polyhedra have convenient mathematical properties, but few interesting objects in the real world have convex surfaces. To build sphere-trees for nonconvex objects, the simulated-annealing algorithm needs to measure the distance from spheres to nonconvex polyhedra. Lemma 5.1 does not quite work in this situation. Figure 5.14 shows an example of how Lemma 5.1 can fail. In this example, the distance $r + d(\mathbf{c}, \mathbf{c}')$ is greater than the correct distance from the sphere to the polyhedron, $d(\mathbf{q}, \mathbf{q}')$. To verify the correctness of $d(\mathbf{q}, \mathbf{q}')$, consider Figure 5.15. It shows the *push-off surface* for the polyhedron, in which all the faces have moved outward by the same distance along their normal vectors. This distance is the minimum distance from each point on the push-off surface to the polyhedron. For a point at which the push-off surface just encloses the sphere, this distance is as large as for any other point on the sphere. In Figure 5.15, $\mathbf{q}$ is such a point, so the distance from the sphere to the polyhedron is $d(\mathbf{q}, \mathbf{q}')$.

The push-off surface is helpful for visualizing the distance from a sphere to a nonconvex polyhedron, but it is not a practical computational technique. Unfortunately, there do not seem to be any alternative approaches for computing the distance exactly. The only solution I can find is to use an approximation to the distance.

The simplest approximation is to go ahead and use the distance provided by Lemma 5.1. In Figure 5.14, this distance is not much different from the correct distance. Note also that it is an overestimate of the correct distance. This property seems to hold in general, although a proof seems difficult. For building sphere-trees by simulated annealing, overestimating the distance can fool the algorithm into thinking it is improving the fit of the spheres when it is really doing the opposite. As an example, say the simulated-annealing algorithm measures the "looseness" of a set of spheres as the maximum distance to the polyhedron for a sphere in the set. Let s be the sphere with the current maximum distance, but assume that distance is too large because the algorithm measured it with Lemma 5.1. At the current instant in the annealing schedule, the algorithm randomly chooses to change some other sphere, s'. The algorithm could move s' so that it is actually further from the polyhedron than s, but the error due to Lemma 5.1 might let the algorithm think s still has the maximum distance.

To avoid this problem, the algorithm could use a more accurate distance approximation. One approach is to compute the exact distance to the polyhedron from a set of sample points on the surface of the sphere; the maximum of these sample distances approximates the distance from the sphere to the polyhedron. Note that this approach is a discrete approximation to the original definition of the distance. To make this approximation as accurate as possible on the average, the algorithm should distribute the sample points evenly across the sphere's surface. The vertices of a dodecahedron inscribed in the sphere meet this criteria. Paeth [Pae91] tabulates the positions of these twenty vertices for a unit sphere, so the algorithm need only scale these positions at runtime. A distance approximation based on samples can be an underestimate, because the location on the sphere that has the true maximum distance to the polyhedron may lie between the sample points. To compensate, the algorithm can add a correction to the maximum sample distance. This correction is the maximum distance from any point on the sphere to a sample point. The algorithm can precompute this correction distance for a unit sphere and scale it at runtime.

Approximating the distance from sample points generally gives higher accuracy than using Lemma 5.1. Unfortunately, using sample points is also slower, by a factor of the number of sample points. A few optimizations can make the approach faster in some situations. As one optimization, the algorithm can use a minimax search strategy. At each sample point, the algorithm measures the minimum distance to the faces of the polyhedron. If at any face the current minimum falls below the overall minimum for a previous sample point, the algorithm need not compute the distance from the sample point to the remaining faces; this policy works because the algorithm needs the maximum distance over the sample points. A second optimization exploits coherency inherent in simulated annealing. The simulated-annealing algorithm changes spheres by small amounts only. Thus, the sample point that had the maximum distance before a change is likely to still have the maximum distance after the change. It thus makes sense to process this sample point first. By using this strategy in conjunction with minimax search, the algorithm can often reduce the number of sample points it processes.

The extra processing required for multiple sample points makes this approximation appropriate only in situations that demand more accuracy than Lemma 5.1 can provide. One such situation is measuring the accuracy of a fully constructed sphere-tree, a topic which Section 5.3.5 discusses further. For measuring "looseness" during the course of the annealing schedule, however, the extra accuracy does not seem to make much difference. In my tests, the simulated-annealing algorithm did not produce significantly better sphere-trees when it used the sample-based approximation instead of Lemma 5.1, and the latter was definitely faster.

5.3.3 Ensuring Conservative Coverage

A sphere-tree is *conservative* if it completely covers the surface of an object. As Chapter 3 discusses, the narrow phase needs conservative sphere-trees so that it will never miss a collision that it would have detected between the actual surfaces of objects. To build conservative sphere-trees, the simulated-annealing algorithm from Section 5.3.1 must test the random changes it makes at each step of the annealing schedule. More specifically, each time it translates and scales a child sphere it must determine if that change caused any part of the object that is covered by the parent sphere to become uncovered. If so, the algorithm must reject the change and try another one. This section discusses how the algorithm checks for conservative coverage of a triangulated polyhedral object.

A triangulated polyhedron is covered if each of its triangular faces is covered. Each such triangle lies in a plane in $\mathbb{R}^3$, and the intersection of this plane with a set of spheres is a set of filled circles, or *disks*. The set of spheres covers the triangle if the union of these disks covers the triangle. Note that checking coverage is thus a collection of 2D subproblems. For the algorithm that builds sphere-trees, the set of spheres will be the children of a parent sphere. The children only need to cover what their parent covers, so the problem becomes determining if the intersection of a triangle with one disk is covered by a union of disks.

This problem is simplified by the following lemma. The lemma applies to any closed region; for the sphere-tree algorithm, the region is the intersection of a triangle with the parent's disk.

Lemma 5.2 (the boundary lemma) *A closed region is covered by a union of disks if and only if at least one disk from the union touches the region, and for each disk the part of its boundary inside the region is covered by the other disks.*

Proof: This proof is somewhat informal; a more rigorous proof is possible but it adds little to the understanding of the lemma. First, the "if" half follows by contradiction. Assume that a disk touches the region and the disks' boundaries are covered, and assume that the region is *not* covered. Some subset, U, of the region must thus be uncovered. This U must be adjacent to at least one of the disks, as illustrated in Figure 5.16. A portion of the boundary of this disk must touch U, and this portion cannot be covered by any other disk since U is uncovered. This conclusion contradicts the assumption that all the disk boundaries are covered, so the "if" half must be true. Second, the "only if" half follows by a similar contradiction. Assume that the region is covered by the disks. Clearly, at least one disk must thus touch the region. Now, assume that for some disk, the portion of its boundary inside the region is not covered. This portion of the boundary must thus define the border of subset, U, of the region which is uncovered. The existence of U violates the assumption that the region is covered, so the lemma must be true. □

The advantage of the boundary lemma is the way it converts the problem of checking a triangle's coverage from a 2D problem to a set of 1D problems. The boundary of a disk can be parameterized by the interval $[0, 2\pi]$. The portion of the boundary that is inside both the triangle and the parent is a (possible disjoint) subinterval of this interval. The part of this boundary that the other disks cover is another (possibly disjoint) subinterval. If the latter subinterval contains the former, then the appropriate part of the disk's boundary is covered by the other disks. If each child's disk passes this test, then the triangle is covered.

Figure 5.17 gives pseudocode for this algorithm. Line 4 finds the interval from a disk's boundary

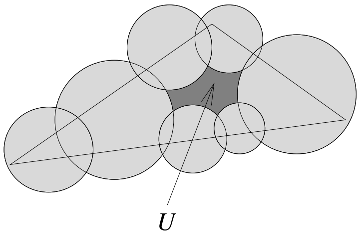


Figure 5.16: The area U is an uncovered subset of the region. For simplicity, the region is the whole triangle, that is, the triangle is fully inside the parent (not shown).

```
1   covers(parent, children, triangle)
2   {
3       for each sphere s_j in children {
4           I_j ← boundary of s_j's disk inside triangle
5           I_j ← I_j clipped to parent
6       }
7       for each pair of spheres (s_j, s_k) in children whose disks intersect {
8           subtract from I_j what is covered by s_k's disk
9           subtract from I_k what is covered by s_j's disk
10      }
11      for each sphere s_j in children
12          if (I_j ≠ ∅)
13             return FALSE
14      return TRUE
15  }
```

Figure 5.17: This algorithm uses the boundary lemma to check if the parent's children cover the part of the triangle inside the parent.

that is inside the triangle. The main operation in this step is finding the intersections between the disk and the triangle's edges. Line 5 clips this interval against the parent sphere. Doing so requires intersecting this interval with the part of the disk's boundary that is inside the parent's disk. This interval operation involves several cases, but getting it to work is not difficult. Inside the loop starting at line 7 the algorithm subtracts one boundary interval from another. Interval subtraction is comparable in difficulty to interval intersection. The loop starting at line 7 reports all pairs of children spheres whose disks intersect. The simplest way to implement this step is to compare every disk against every other disk. This approach is efficient enough because a sphere-tree node has a small number of children (e.g., eight if the initial configuration comes from an octree subdivision) so there are never many disks to compare.

When the simulated-annealing algorithm changes one child of a parent, it need not call the coverage algorithm in Figure 5.17 for every triangular face in the object's surface. The only faces that can become uncovered are those that intersected the child before it was changed, so the coverage algorithm should check only these faces. The simulated-annealing algorithm must thus store with each child a list of every face it intersects, and the algorithm must update this list when it changes a child. The algorithm must find any new faces that the changed child did not intersect previously.

The only such new faces that matter, however, are those that also intersect the parent, since the child and its siblings are responsible for covering only those faces. So the simulated-annealing algorithm checks only those faces associated with the parent for intersections with the child.

As another efficiency measure, the algorithm should execute lines 4 and 5 of Figure 5.17 only for the child that it changed. To implement this optimization, the algorithm needs to cache the boundary intervals for each pair of a sphere and a face that sphere intersects, to allow reuse of this information for unchanged spheres. The algorithm can store these caches with the list of intersected faces that it keeps for each sphere.

The boundary lemma applies to situations that are more general than the coverage of a triangle by disks. Appendix B discuss these situations in more detail.

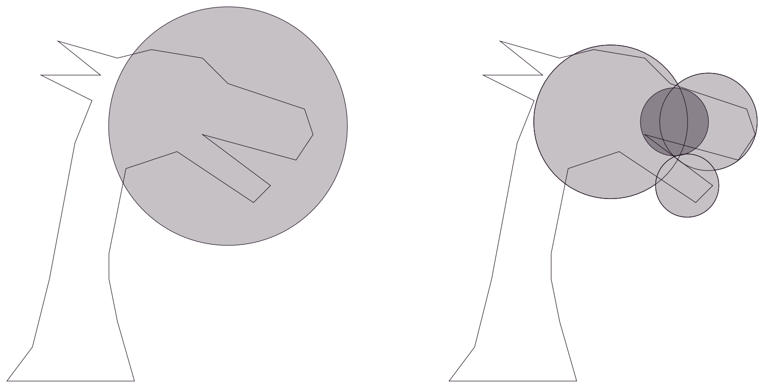
5.3.4 Eliminating Redundant Spheres

The algorithm that uses simulated annealing to tighten a set of children spheres repeatedly translates and scales these spheres. By doing so, the algorithm might cause a sphere to become redundant, that is, cause it to cover parts of the object that are completely covered by other spheres. Redundant spheres make the narrow phase less efficient, because the detection algorithm from Figure 5.4 spends time checking for intersections with these spheres but gains no additional accuracy from this processing. In theory, it might be possible to discourage the algorithm from moving spheres into redundant positions by adding to the energy measurement a term that increases with the amount of sphere overlap. In practice, however, this term would have detrimental effects, because partially-overlapping spheres often fit the object most tightly.

As an alternative, the algorithm can change the children freely and then check to see if any children are redundant. There is no reason to check for redundancy during the course of the annealing schedule, because a sphere that is temporarily redundant may subsequently become an unique and important component in the overall fit. When the annealing schedule is complete and the final configuration of a parent's children is set, then the algorithm should initiate a postprocess that finds and eliminates redundant children.

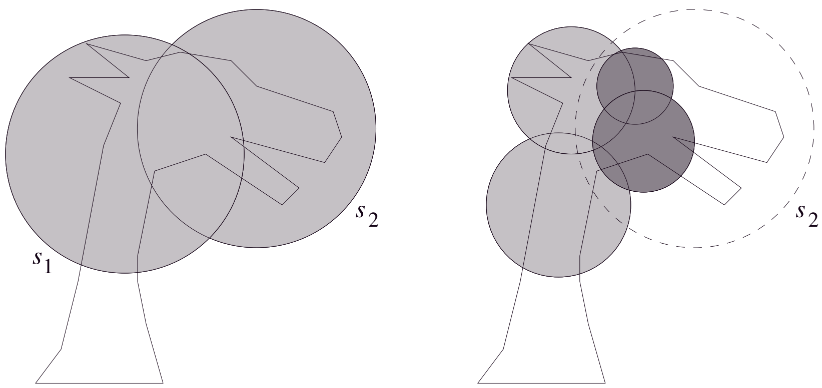
The simplest form of redundancy occurs when a child sphere is made redundant by its siblings. Figure 5.18 shows an example of this situation. Clearly, the sphere-tree will be no less accurate or conservative if this child is eliminated. A more subtle form of redundancy involves aunt spheres (siblings of the parent). In Figure 5.19(A), spheres s_1 and s_2 are siblings. Figure 5.19(B) shows s_1's children, to whom s_2 is an aunt. The two darkened children of s_1 are completely covered by s_2. As a result, any part of the object they cover will also be completely covered by s_2's children, their cousins. Any parts of other sphere-trees that intersect s_1 and, in turn, these darkened children will also intersect s_2 and, in turn, the cousins of the darkened children. The two darkened children are thus redundant, and removing them from the sphere-tree will not affect the narrow phase's correctness.

The postprocess that eliminates these redundant children borrows heavily from the ideas of the previous section, on checking for conservative coverage. Consider first the case of finding children made redundant by their siblings. A child sphere, s, is redundant if its siblings cover everything that it covers. More specifically, s, is redundant if for each triangular face of the object it intersects, the part of that face it covers is also covered by its siblings. The algorithm from Figure 5.17 will check for this condition if it takes s as the *parent* parameter and s's siblings (excluding s itself) as the *children* parameter. The situation is similar for a child made redundant by its aunts. This child is redundant if the part of each face it covers is also covered by its aunts. The algorithm from



(A) (B)

Figure 5.18: (A) A parent sphere. (B) The parent's children. The darkened child is redundant because of its siblings.



(A) (B)

Figure 5.19: (A) Two parent spheres which are siblings. (B) The children of s_1. The darkened children are redundant because of their aunt, s_2.

Figure 5.17 will check for this condition if it takes s as the *parent* parameter and s's aunts (excluding s's parent) as the *children* parameter.

A redundant sphere, s, is dependent on the set of spheres, S, that cover what it covers. This dependency means that the postprocess may legally remove s only if either it removes no spheres from S or it makes certain that any spheres it removes from S are covered by other spheres. Conceptually, the simplest policy is to not remove any sphere in S. A convenient way for the postprocess to keep track of these constraints is with an undirected graph. The graph contains a node for each redundant sphere. The node represents that sphere and the set of spheres on which it is dependent; these two pieces of data are, respectively, the "covered" sphere and "coverers" set of the node. The graph contains an edge between two nodes if the "covered" sphere of one node is a member of the "coverers" set of the other node. The postprocess thus cannot remove both the "covered" sphere from one node and the "covered" sphere from a second node connected by an edge; removing both would be deleting a redundant sphere and sphere that made it redundant. So the postprocess can remove only spheres corresponding to an *independent set* of graph: a subset of nodes such that for any edge, only one node on that edge is in the subset.

The postprocess should find as large an independent set as it can, corresponding to removing as much redundancy as possible. Unfortunately, Garey and Johnson [GJ79] show that finding the largest independent set of a graph is an NP-complete problem. A tractable alternative is to find an approximation to the largest independent set. The simplest approach is for the postprocess to use a greedy algorithm. For each node in the graph, the algorithm adds the node to the independent set if none of its neighbors are already in the set. The algorithm is likely to find larger independent sets if it goes through the nodes in order of increasing degree; this heuristic favors adding to the independent set nodes that will prevent few other nodes from joining the set. I have implemented this approach and it does seem to find sizable independent sets in practice.

The policy of not removing any sphere that makes another sphere redundant is overly conservative. Say sphere s is made redundant by spheres s_1 and s_2. If s_1 is, in turn, made redundant by spheres s_{11} and s_{12} then s_{11} and s_{12} must cover the part of s that s_1 covers. Thus, if s_{11} and s_{12} remain in the sphere-tree the postprocess may legally remove both s and s_1. The only problem is cyclic dependencies. If s_1 depends on s_2, which depends on s_3, which depends on s_1, the postprocess cannot remove all three spheres without leaving part of the object uncovered. The solution is to detect and break cycles. Depth-first search is a convenient way to find cycles in an undirected graph [Sed88]. Specifically, the algorithm works as follows. It starts a depth-first search and stops at the first node that is part of a cycle. By taking this node out of the graph, the algorithm breaks the cycle. It starts a new depth-first search of the resulting graph, and continues breaking cycles and starting new searches until the graph contains no cycles. The postprocess may then legally to remove all "covered" spheres for nodes remaining in the graph. The one case on which this approach fails is a two-node cycle. A two-node cycle is an unusual situation, however. It corresponds to a pair of spheres that by themselves cover an isolated "island" of the object, touching no other spheres. The postprocess can detect such cycles and deal with them as a special case, removing only one sphere from the pair. Although this overall approach of detecting cyclic dependencies sounds promising, I have not actually implemented it. Doing so is an interesting topic for future work.

5.3.5 Accuracy

When an application uses approximation, it is important for the application to know how much accuracy it is losing. The conservative nature of spacetime bounds and sphere-trees limits the form of inaccuracy that these structures permit. As Chapter 3 discusses, a detection algorithm using these

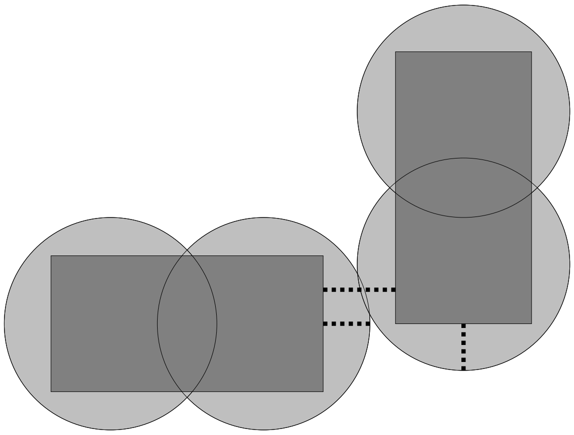


Figure 5.20: The sum of the distances from the spheres to the object is an upper bound on the distance between the objects.

structures never allows objects to start penetrating before it detects a collision. It can, however, detect a collision between objects which are close but not actually touching. A reasonable measure of the inaccuracy in this case is the shortest distance between the objects at the moment when the algorithm decides they collide.

The results of Section 5.3.2 allow the algorithm to quickly estimate this inaccuracy. Say the narrow phase finds a pair of intersecting spheres from different objects' sphere-trees. The sum of the maximum distances from these spheres to their respective objects is an upper bound on the distance between the objects, as Figure 5.20 illustrates. A tighter upper bound is this sum minus the amount the spheres overlap. If the narrow phase finds more than one pair of intersecting spheres, the tightest upper bound on separation distance is the *minimum* of the estimates over all the pairs. Using the approach from Section 5.3.2, the algorithm that builds these sphere-trees can store with each sphere an upper bound on its maximum distance. The narrow phase thus does no work to obtain these distances, and it can estimate the inaccuracy of any collision with just a few arithmetic operations. This estimate is an overestimate, but such a value is more useful to an application than an underestimate.

Since the distance from each sphere to the object can be computed by the preprocess that builds the sphere-tree, users can know roughly what inaccuracy to expect before they start running an application. To summarize the distances for all spheres in the sphere-tree, various statistics are useful. Histograms of distances, one for each level of the sphere-tree, give the most complete picture. These histograms convey a qualitative impression of how inaccuracy is distributed through the sphere-tree, suggesting the probability of various amounts of inaccuracy. A more concise summary is the maximum distance of any sphere at each level, which indicates the worst-case inaccuracy for an application that uses sphere-trees. The average distance over each level is perhaps a more useful statistic. It approximates the expected inaccuracy over all possible situations, a measurement that can be more realistic than the worst-case inaccuracy.

In order for the detection algorithm to truly implement graceful degradation, its inaccuracy must decrease as it uses more processing time. A sphere-tree supports this goal if the distances from its spheres to the object decrease for spheres at deeper levels. The next section describes specific examples of this property in sphere-trees generated by simulated annealing.

figure	level	processing time (mins)	number of nodes	fraction of octree's average inaccuracy	fraction of octree's maximum inaccuracy
5.22	1	10.9	7	0.317	0.295
	2	43.5	46	0.252	0.365
5.23	1	9.5	5	0.430	0.395
	2	30.5	27	0.295	0.510
5.24	1	10.8	6	0.210	0.326
	2	30.7	29	0.146	0.603
5.25	1	8.8	5	0.271	0.564
	2	21.1	26	0.153	0.634

Figure 5.21: A summary of the simulated-annealing results on the model of the desk lamp.

It would be useful to have a function that predicts the inaccuracy of the detection algorithm for a given amount of processing time. There is no simple way to derive such a function, though. Across different applications and different situations within one application, there is too much variance in the patterns of spheres which intersect when sphere-trees approach each other. This variance precludes accurate prediction of the processing time the algorithm will need. A worst-case analysis might be possible, but it is likely to be contrived. A better possibility is an average-case analysis based on empirical trials. The point behind sphere-trees and interruptible collision detection in general, though, is to make a predicting function less important. The application controls the time spent by the detection algorithm by telling it to stop when it reaches the limit of its allotted time.

5.3.6 Results

Figure 5.7 showed a model of a desk lamp, and the sphere-tree that the octree algorithm generates for this model. This section describes the results of the simulated-annealing algorithm on the same model. Figure 5.21 summarizes some statistics from these results, and Figures 5.22 through 5.25 show the sphere-trees themselves.

The main advantage of these sphere-trees over the octree-based sphere-tree is the improved tightness of the fit. The last two columns of the table in Figure 5.21 quantify this improvement. These columns compare the inaccuracy in the sphere-trees generated by the two methods, where inaccuracy is the distance from a sphere to the object, as the previous section discusses. In particular, one column compares the average inaccuracy for spheres at a given level, and the other column compares the maximum inaccuracy for those spheres. Note that the simulated-annealing algorithm produces sphere-trees with only a fraction of the inaccuracy, by either of these statistics. It is not clear which of these statistics provides the better summary of the "overall" inaccuracy of the sphere-tree. The average has the advantage that it approximates the expected inaccuracy over all possible collision involving the sphere-tree. The disadvantage of the average is the way it can be fooled by spheres of extreme size; one big sphere and many tiny spheres may have a low average inaccuracy but may not actually provide a good fit.

Figure 5.21 also shows a disadvantage of simulated annealing, its slowness. Generating these sphere-trees took between 29.9 and 54.4 minutes each on a Hewlett Packard 9000 Model 735. The technique from Section 5.3.4, to remove redundant spheres, does alleviate the problem somewhat; the simulated-annealing algorithm builds a sphere-tree from the top down, so reducing the number of parent spheres reduces the time spent on children spheres. The number of redundant spheres the

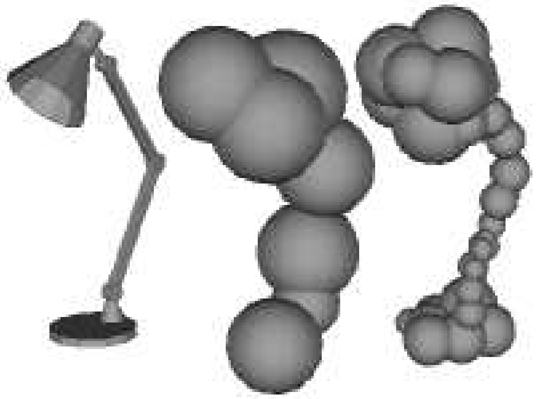


Figure 5.22: Results of simulated annealing, with the energy for a set being the maximum over its members, and the random seed being 2.

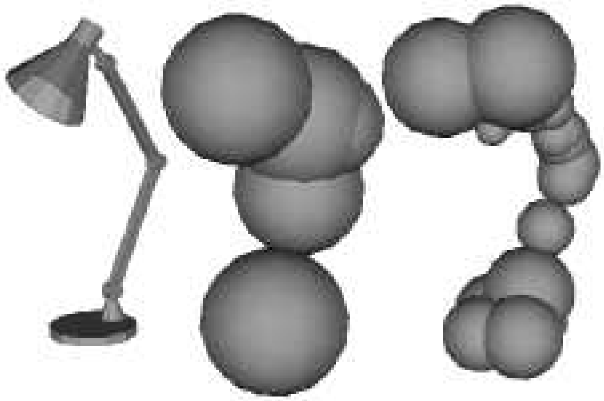


Figure 5.23: Results of simulated annealing, with the energy for a set being the maximum over its members, and the random seed being 3.

algorithm removed is indicated by the fourth column of Figure 5.21; if the algorithm did not remove any spheres, these numbers would be powers of eight, the default number of children for each sphere. Even with this optimization, the algorithm would take a long time to add a level onto one of these sphere-trees. For the example of the sphere-tree in Figure 5.22, the algorithm spent about 6 minutes on the children of each sphere at level one. If it were to spend an equivalent amount of time on each of the 46 spheres at level two, the additional time would be 4.6 hours!

Figures 5.22 through 5.25 show a final problem with simulated annealing, its unpredictability. Section 5.3.1 explains that the annealing schedule makes random changes to spheres, so the algorithm needs a random-number generator and a seed value. The algorithm also must measure the energy (i.e., "looseness") of a set of spheres, and Section 5.3.2 indicates there are several ways to compute the energy of the set from the energies of its members. The choice of random seed and energy function can have a significant impact on the results of the algorithm. For both Figures 5.22 and 5.23, the algorithm computed the energy of the set as the *maximum* energy of its members (it used Lemma 5.1 to approximate the energy for each member sphere). The algorithm used different random seeds for the two runs, though. The results of the two runs are quite different, with the second result,

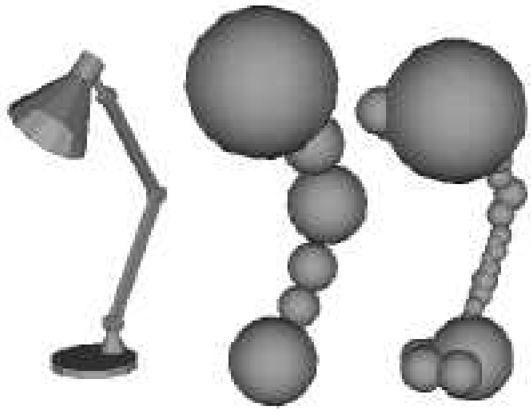


Figure 5.24: Results of simulated annealing, with the energy for a set being the sum of its members, and the random seed being 2.

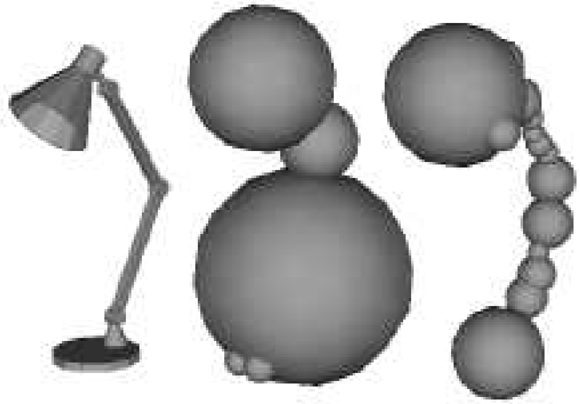


Figure 5.25: Results of simulated annealing, with the energy for a set being the sum of its members, and the random seed being 3.

Figure 5.23, being less tight and less symmetrical. Figures 5.24 and 5.25 use a different energy function, the *sum* of the energies of the set's members. In Figure 5.24, this energy function gives a better first-level fit than the previous energy function. Its second-level fit has problems, however; the fit around the shade of the lamp, in particular, is no better than the first-level fit. Figure 5.25 shows the considerable impact of a different random seed with the same energy function. Ideally, the algorithm that builds an object's sphere-tree should be able to run reliably without any human intervention. Users should not have to experiment with the algorithm's parameters, especially for an algorithm as slow as simulated annealing. These examples suggest simulated annealing, although an improvement over the octree algorithm in some ways, cannot meet this goal.

Chapter 6

Sphere-Trees and Medial-Axis Surfaces

This chapter continues the development of sphere-trees, which began in the previous chapter. In particular, this chapter focuses on an advanced method for the automatic construction of sphere-trees. At the heart of this method is the concept of the *medial-axis surface*. A medial-axis surface of an object corresponds intuitively to the "stick-figure" representation or "skeleton" of the object. The symmetries of an object around its skeleton suggest configurations of spheres that approximate the object's shape, leading to an algorithm for building a sphere-tree. This algorithm has several advantages over the simulated-annealing algorithm from Chapter 5: it is more efficient and it produces accurate sphere-trees more reliably.

The first section of this chapter provides some background on medial-axis surfaces and their relationship to sphere-trees. The next section discusses an algorithm for approximating the medial-axis surface of a polyhedral object. This algorithm generates the medial-axis surface from a *Voronoi diagram*, a structure that represents proximity relationships between sample points on the object's surface[1]. Key features of this algorithm are a method for distributing sample points uniformly over an object's polyhedral surface, a way of avoiding numerical errors in the Voronoi diagram, and a method for adding new sample points if the Voronoi diagram does not provide the detail needed for a medial-axis surface. The last part of this chapter shows how to generate a sphere-tree from a medial-axis surface. The algorithm uses the medial-axis surface to determine a union of spheres at a high level of detail, and also to simplify this union to produce lower levels of detail. This algorithm also reuses elements of the simulated-annealing algorithm from Section 5.3, such as the method for checking whether a set of spheres conservatively cover the object.

6.1 Medial-Axis Surfaces

The *medial axis* was introduced by Blum [Blu67] to describe the shapes of biological objects. He defines the medial axis of a two-dimensional (2D) shape by various physical analogies, including the "grassfire analogy": consider igniting simultaneously all the points on the boundary of a figure in a field of grass; the places where the the flames meet and die out define the medial axis of the

[1] Throughout this chapter, the term "surface" means "medial-axis surface" only when it is explicitly designated as such. The term "object's surface," for example, refers to the outer boundary of the object.

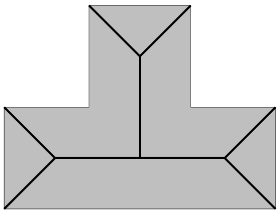
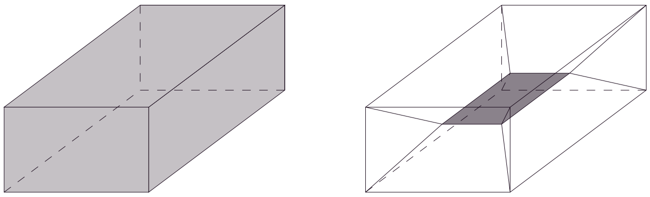


Figure 6.1: A medial axis for a simple 2D figure.



(A) (B)

Figure 6.2: (A) A simple 3D object. (B) Its medial-axis surface, with only the central part of the surface darkened.

figure. Points on the medial axis are equidistant from two sides of the boundary, so they represent a symmetrical axis or "skeleton." Figure 6.1 shows an example of a medial axis. In later work, Blum and Nagel [BN78] state another definition of the medial axis, which they also call the "symmetric axis": the locus of the centers of maximal circles that fit inside the figure, with no circle containing another. They also describe how quantitative properties of a medial axis relate to properties of the figure's shape.

Nackman and Pizer [NP85] discuss similar properties for *medial-axis surfaces* of three-dimensional (3D) objects. The definition of a 3D medial-axis surface is simple extension of the 2D definition: the locus of centers of maximal spheres that fit inside the object, with no sphere containing another. Note that these centers lie on *surfaces* rather than *lines*, as Figure 6.2 shows. Nackman and Pizer argue that a medial-axis surface naturally combines two shape-description paradigms, in the sense that it both discards inessential details of a shape and also decomposes a shape into simpler elements. These properties make medial-axis surfaces a promising starting point for the problem that interests Nackman and Pizer, automatically analyzing the shapes of human organs from computed tomography data.

The definition of a medial-axis surface, in terms of spheres fitting inside an object, suggests that the medial-axis surface for an object is related to its sphere-tree. Picking a representative set of the spheres centered on the medial axis and expanding them so they conservatively cover the object produces one level of the sphere-tree. Varying the number of spheres in the set leads to different levels of detail. Section 6.3 will make this notion more explicit. Using a medial-axis surface to guide the placement of spheres has intuitive appeal, in the sense that it corresponds to "fleshing out a skeleton." O'Rourke and Badler [OB79] note the connection between spheres that approximate an object and its medial-axis surface, but in a complementary way; their algorithm shrinks spheres until they fit inside an object, and thus produces points on the medial-axis surface as a byproduct.

6.2 Building Medial-Axis Surfaces

An algorithm that builds a sphere-tree from a medial-axis surface must first build the medial-axis surface. This problem is not a simple one, and the literature contains few algorithms that solve it. Hoffmann [Hof92] states that the only exact algorithm for the medial-axis surface of a 3D object is his algorithm, for constructive solid geometry (CSG) objects [Hof90]. This algorithm processes each pair of distinct elements from an object's surface to find points equidistant from the elements and interior to the object. It then analyzes these points to determine the structure of the object's skeleton. The steps in this algorithm are quite intricate and involve many cases, so getting the algorithm to work in practice would be a challenge. It is also unclear how efficient the algorithm would be, since it solves many systems of equations in the process of finding the equidistant points.

The application of building a sphere-tree does not require an exact medial-axis surface. An approximate medial-axis surface guides the positions of the spheres sufficiently well in most situations. There are a variety of algorithms that compute approximate medial axes of 2D figures. One example is Kwok's algorithm [Kwo88], which thins a figure in a binary image to find the pixels that approximate its skeleton. This algorithm traces the contour of the figure and slices off those pixels which cannot be part of skeleton, producing a new contour closer to the skeleton. Repeating this process ultimately identifies the pixels of the skeleton. A variation on this approach could work in three dimensions. Instead of processing a set of square pixels, the algorithm would process a set of cubical voxels that approximate the shape of the 3D object. Hoffman [Hof92] mentions this idea, but there seems to be no paper which presents an actual algorithm based on it.

6.2.1 Approximate Medial-Axis Surfaces and Voronoi Diagrams

Another approach to approximate medial-axis surfaces uses *Voronoi diagrams*, also known as *Dirichlet tessellations* [PS85]. A Voronoi diagram for a set of points identifies, for each point, the region of space which is closer to that point than to any of the other points. Figure 6.3 shows a Voronoi diagram for 2D data, but note that the definition of a Voronoi diagram applies to any dimension. The region associated with a point is a convex polyhedron that surrounds the point, and it is called the *Voronoi cell* for the point. The cell's vertices are the vertices of the diagram, or *Voronoi vertices*. There is one face of the cell between the point and each nearby point, and this face is equidistant between the points; Figure 6.3 shows this property for 2D cells, whose "faces" are line segments.

The equidistance of faces between point pairs is what relates a Voronoi diagram to a medial-axis surface. Consider a set of points distributed over the polyhedral surface of an object. For a pair of points separated by the object's interior, the intervening cell face will lie close to the middle of the object. The cell face thus approximates a portion of the object's medial-axis surface. Figure 6.4 illustrates this idea in two dimensions. This idea should work for objects whose surfaces are not polyhedra, but this section focuses on the polyhedral case.

Hoffmann [Hof90] mentions the connection between Voronoi diagrams and medial-axis surfaces in passing, and Goldak *et al.* [GYKD91] develop the idea further for polyhedral objects. They phrase their algorithm in terms *Delaunay regions*, which are duals to the Voronoi cells, but it is simpler to think in terms of the Voronoi cells directly. The algorithm identifies the Voronoi vertices from cell faces that approximate the medial-axis surface; these vertices are the ones interior to the object. It then groups nearby vertices into *segments*, which are uninterrupted sections of the medial-axis surface between places where it branches. There are two kinds of segments: *branch* segments, which touch (or nearly touch) the object's surface, and *trunk* segments, which do not. Figure 6.5 shows the 2D versions of trunk segments with solid lines and branch segments with dashed lines. The

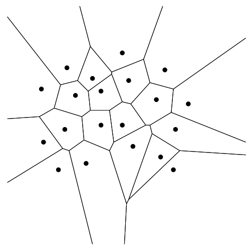


Figure 6.3: A Voronoi diagram for 2D points.

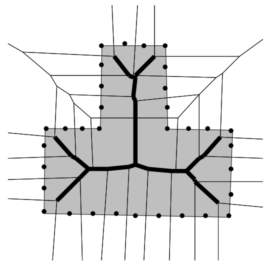


Figure 6.4: A Voronoi diagram for points on the boundary of a 2D figure, resulting in an approximation to the figure's medial axis.

algorithm differentiates between branch and trunk segments by the properties of their vertices.

For building sphere-trees, the grouping of vertices into segments is not necessary. The important parts of the algorithm are the earlier steps, which deal with the vertices themselves. The theory behind these steps seems correct, but Goldak *et al.* do not describe many of the implementation details. To get these steps to work in practice, I had to make a number of significant additions to the algorithm.

The first step of the algorithm distributes points over the polyhedral surface of the object. Goldak *et al.* acknowledge that the density of the distribution affects the results of the algorithm. They prove that the results converge to the exact medial-axis surface as the number of points increases to infinity. They state that choosing a sufficiently dense set of points is "relatively easy" in practice, but they give few guidelines on actually placing the points. Unfortunately, the algorithm is sensitive enough to the point placement that many intuitively reasonable placements do not work. My experience suggests that the best solution is to start by distributing the points as uniformly as possible, and

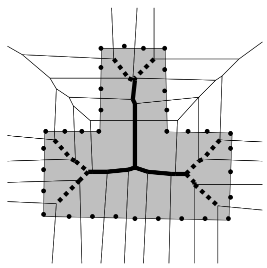


Figure 6.5: The solid segments of the medial axis are trunks, and the dashed segments are branches.

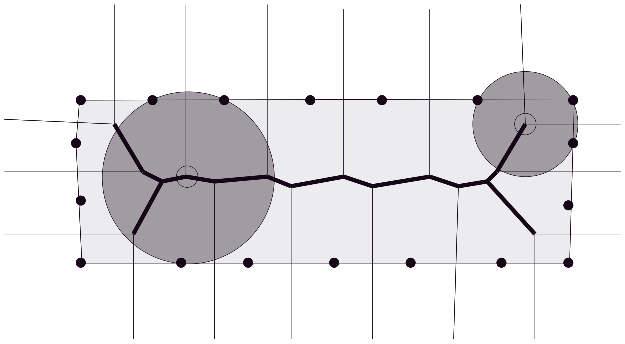


Figure 6.6: Two examples of how a Voronoi vertex is the circumcenter of its forming points.

then add more points to correct specific problems in the resulting medial-axis surface. Section 6.2.2 discusses uniform distributions of points on objects with complicated shapes. Section 6.2.4 explains the most common problems that result from point placement. These problems involve segments of the medial-axis surface "leaking" out of one part of the object, crossing a "gap" outside the object and entering the object at distant location. It is difficult to prevent these problems from occurring, but it is possible to correct them once they have occurred. The correction involves adding new points at the locations of the problems and updating the Voronoi diagram.

The second step of the algorithm builds a Voronoi diagram from the surface points. Each Voronoi vertex is at the center of a sphere that touches four of the points, that is, the vertex is the *circumcenter* of the four points. The four points are those which lie in the four Voronoi cells that adjoin the vertex, and these points are the vertex's *forming points*. Preparata and Shamos [PS85] prove the corresponding property for two-dimensional Voronoi diagrams, in which each vertex is the circumcenter of three points. Figure 6.6 shows an example of the 2D case. The sphere centered at each Voronoi vertex and the forming points lying on that sphere are the foundation of the algorithm that converts a medial-axis surface into a sphere-tree, as Section 6.3 will explain.

Actually building the Voronoi diagram can cause problems. A basic algorithm to build a 3D Voronoi diagram is not complicated, but it cannot tolerate imprecise numerical computations. Unfortunately, the arrangements of surface points that suit the needs of medial-axis surfaces tend to promote this imprecision. The risk that the basic algorithm will fail is high enough that a more robust algorithm is necessary. Section 6.2.3 discusses the difficulties of making a Voronoi-diagram algorithm more robust without sacrificing accuracy.

Not all the Voronoi vertices are part of the medial-axis surface. In particular, only the vertices interior to the object can lie on the medial-axis surface, so the next step of the algorithm identifies these vertices. There are a variety of ways to determine if a vertex, v, is inside a polyhedron. One approach is as follows. Find the position, $\mathbf{w}$, on the polyhedron closest to the position of v. Then determine if v lies behind the polyhedral faces in the local neighborhood of $\mathbf{w}$; if so v must be inside the polyhedron because no other parts of the polyhedron can intervene and change the classification. For the first step, finding $\mathbf{w}$, the methods of Section 5.3.2 apply, and the octree optimization is particularly helpful. For the second step, the analysis depends on the whether $\mathbf{w}$ lies on a polyhedral face, edge or vertex. Let $\boldsymbol{\sigma}$ be the vector from $\mathbf{w}$ to the position of v. If $\mathbf{w}$ lies on a face, then v is inside if the dot product of $\boldsymbol{\sigma}$ and the face's outward-pointing normal vector is negative. When $\mathbf{w}$ lies on an edge, the classification depends on whether the two faces adjoining the edge are convex or concave. The vertex v is behind a pair of convex faces if *both* faces' dot

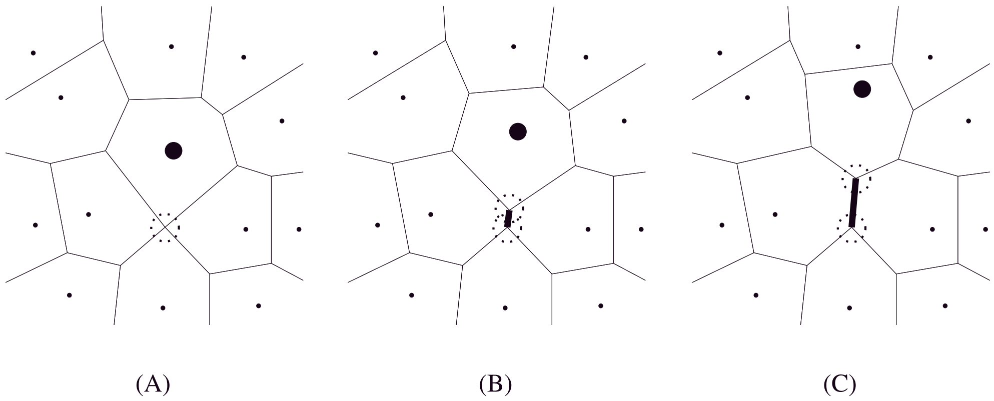


Figure 6.7: (A) The vertex in the dashed circle appears to adjoin four cells. (B)(C) Perturbing the darkened point, however, reveals that the vertex is really two coincident vertices separated by a degenerate cell boundary.

products with $\boldsymbol{\sigma}$ are negative; v is behind concave faces if *either* dot product is negative. If $\mathbf{w}$ lies on a polyhedral vertex, then v cannot project onto any of the edges or faces adjoining that vertex. In this case, v is behind the faces only if the dot product of each face normal with $\boldsymbol{\sigma}$ is negative.

Further processing of the Voronoi vertices on the medial-axis surface, such as building a sphere-tree, requires knowing which vertices are adjacent to each other on the medial-axis surface. This adjacency information is an inherent feature of the Voronoi diagram. For faces of the Voronoi cells that lie on the medial-axis surface, the edges of these faces indicate which Voronoi vertices are adjacent along the medial-axis surface. In the general case, each Voronoi vertex lies on exactly four cells and thus adjoins four edges. If five nearby points lie on a sphere, however, the Voronoi diagram will contain a location at which five cells meet. Such a location looks like a Voronoi vertex that adjoins more than four edges, but it is really two coincident Voronoi vertices and a degenerate cell face. Figure 6.7 illustrates this situation in two dimensions; at this lower dimension, four cocircular points create a degeneracy. The figure shows what happens when one of the cocircular points moves slightly: the degenerate cell face appears again and the coincident vertices separate. Coincident vertices complicate any further processing of the medial-axis surface. The best solution is to coalesce all coincident vertices into a single vertex. This new vertex adjoins the union of the edges from the coincident vertices. One way to find the coincident vertices is to treat each vertex as a sphere of radius ϵ, where ϵ is the minimum accuracy of floating-point arithmetic. The problem then becomes one of finding all the spheres that intersect. To solve this problem efficiently, I use Turk's spatial hashing scheme [Tur90b], summarized in Section 2.3.2, since I already need this algorithm for the performance tests in Chapter 7.

Once the algorithm has coalesced all the coincident vertices, it has established enough of the medial-axis surface that the algorithm to build a sphere-tree can take over. Other applications of medial-axis surfaces, however, need the latter steps of the Goldak *et al.* algorithm, which group the Voronoi vertices into segments. Certain aspects of these latter steps are problematic. I identified some improvements for these steps, but these improvements are still preliminary so this dissertation does not discuss them.

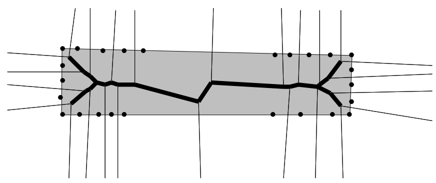


Figure 6.8: The middle of this object does not have enough Voronoi vertices to produce a good set of spheres.

6.2.2 Placing Surface Points

An algorithm that builds an object's medial-axis surface from a Voronoi diagram must first place points on the surface of the object. The arrangement of these points determines the accuracy with which the Voronoi vertices approximate the medial-axis surface. If the algorithm incorporates all the steps described by Goldak *et al.* [GYKD91] to group vertices into segments, an improper distribution of points will yield vertices that imply an erroneous set of segments.

If the algorithm is building a sphere-tree, it need not construct the segments, but the distribution of the points is still important. Recall from Section 6.1 that the Voronoi vertices define the positions of spheres that can be in the sphere-tree. Without making explicit the algorithm that builds these spheres, it should be clear that the density of spheres is limited by the density of the vertices. The density of the vertices is influenced, in turn, by the positions of the points. Figure 6.8 shows that an uneven distribution of points can create a spotty distribution of vertices, which will prevent parts of the object from having adequate coverage by spheres.

It is not difficult to distribute points evenly on the surface of broad, regular shape like a square. The problem is more difficult for a complicated polyhedron with many faces, however. Turk [Tur91] confronts this problem in a different context, generating a mesh of points to support texturing of a polyhedral surface. His solution is to use a relaxation algorithm that gradually forces the points into an even distribution. The relaxation algorithm starts by distributing points randomly across the polyhedron's surface. It then applies forces to each point, forces that push the point away from nearby points. After updating the position of each point in response to the forces, the algorithm computes a new set of forces. The algorithm terminates after a set number of these iterations. For a sufficient number of iterations, the result is an even distribution of points. Turk recommends 40 iterations, and that number works well in my tests.

The first step of Turk's algorithm is straightforward. The random distribution of initial points must ensure that every location on the polyhedral surface is equally likely to receive a point. The probability that a polyhedral face receives a point should thus be proportional to the face's area. The algorithm orders the faces and tabulates a list of partial area sums (i.e., for each face, the sum of the face's area and the areas of the faces before it in the order). To place a point, the algorithm picks a fraction between 0 and 1, finds that fraction of the total area in the list of partial area sums, and assigns the point to the corresponding face. This strategy avoids a bias against faces with small areas. Having assigned a point to a face, the algorithm gives the point a random location within the face [Tur90a].

Computing the forces on each point is more complicated. The repulsive forces a point, p, receives from other points should decrease with distance; Turk recommends a linear decrease and no force for all distances greater than a maximum *repulsive radius.* For a point, p', on the same face as p, it is simple to measure the distance. The direction of the force on p is the vector from p' to p. If p

is not on the same face as p', the situation is less clear. The force should be based on the *geodesic* path from p' to p, that is, the shortest path from p' to p along the polyhedral surface. Finding the geodesic path is difficult, however, so Turk uses a heuristic approximation. If p' is on a face that shares an edge with p's face, then the algorithm rotates p' around the shared edge to put it on the same plane as p. Otherwise, the algorithm projects p' onto the plane of p's face. For a p' more than one or two faces removed from p, this approximation introduces error.

A different heuristic offers a simple way to avoid the error in most situations. Under this heuristic, the algorithm processes independently each *cluster* of faces. A cluster is a set of adjacent coplanar faces; each square side of triangulated cube, for example, is a cluster of two triangular faces sharing a diagonal. Some edges of faces in the cluster are *boundary* edges, which adjoin faces from other clusters; the other edges are *interior* edges, which adjoin faces from the same cluster. The motivation for the heuristic is as follows. When the algorithm processes one cluster, it assumes that the points in all other clusters are (or will become) uniformly distributed. The algorithm thus assumes that the points in other clusters exert a uniform force across each boundary edge onto points in the current cluster. As a consequence, the algorithm can ignore points outside the current cluster and instead compute the force "exerted by" each boundary edge. This force points into the cluster with a direction perpendicular to edge, and it is constant along the length of the edge. A point, p, on the cluster is affected by this force if p projects onto the edge; the direction and magnitude of the force are proportional to the vector of projection.

For points within the current cluster, the algorithm computes repulsive forces with Turk's approach. Two points on the same cluster lie on the same plane, so the vector between the points approximates the geodesic path. This approximation is incorrect if the cluster is nonconvex and the vector goes outside the cluster, but this mistake does not seem to cause problems in either Turk's implementation or mine.

The forces on a point can push the point off the surface of the polyhedron. This problem occurs in any version of the relaxation algorithm, regardless of how it handles geodesic paths. As a solution, the algorithm must check each point after each iteration and push the points back onto the polyhedral surface. When using the cluster heuristic, the algorithm knows that each point must be on the plane of the cluster, so it first projects each point onto that plane. It then must detect points outside the boundary edges of the cluster, and snap them back onto these edges. The simplest solution is to build the line segment from the new position of a point to its previous position (as of the previous iteration). Assuming that the previous position was actually inside the cluster, the new position is outside the cluster only if the line segment intersects a boundary edge; in this case, the intersection closest to the previous position is the corrected new position. This corrected position must really be inside the cluster to avoid future problems. If numerical imprecision causes it to be slightly outside the cluster, the next iteration will fail to detect the need for snapping. This problem occurred only once in my tests, so I did not implement a solution.

When boundary edges exert uniform forces, it will be rare for any point to stay on a boundary edge. The Goldak *et al.* algorithm seems to work better when there are points on boundary edges, so the point-placement algorithm adds them in a separate step. This step assigns points to boundary edges by using a list of partial edge-length sums (analogous to the list of partial area sums at the beginning of this section). Knowing the total number of points per edge, it can directly determine an even spacing without resorting to relaxation.

Turk mentions that uniform space subdivision can optimize the search for points which affect a particular point, p. Recall that only points within the repulsive radius, r, around p exert any force on p. If the algorithm stores all the points in a 3D grid whose cubes have width $2r$, then the only points that can affect p are those in an eight-cube block around p. Turk's spatial hashing scheme [Tur90b], summarized in Section 2.3.2, finds these cubes quickly. A version of the algorithm based on the cluster heuristic can use a simpler version of this optimization. The forces on p come from points and edges on the same cluster as p. A cluster is planar so it defines a 2D space. The algorithm thus needs only a 2D subdivision grid, which is more efficient than a 3D grid. In my experience, the distribution of surface points is not the performance bottleneck in building a medial-axis surface, so I have not implemented this optimization.

Figure 6.9 shows some results of the algorithm with the cluster heuristic. The distribution of points on this model, a bathroom faucet, is quite even overall. Figure 6.10 shows the results for a second model, a spaceship. For this model, I specified a disproportionately high density of points along the cluster-boundary edges. The spherical regions near the front of the ship have a large number of these edges, so they receive a large number of points. Being able to control the density of points on edges and faces independently adds flexibility, but balancing the two densities can require some care. The front tip of the spaceship shows another disadvantage of the cluster heuristic. The points here form several concentric rings, an artifact which is not always desirable. Rings occur because a set of narrow clusters converge at the tip. On each edge between these clusters, the spacing of points is the same; this regularity produces the ring patterns. To avoid this problem, the algorithm could vary the spacing of the points along the edges. This solution does not seem necessary for medial-axis surfaces. In this context, the rings do not cause problems, and the overall distribution of points is even enough to meet the basic requirements of the medial-axis-surface algorithm. The algorithm cannot specify more subtle requirements until after it has built the Voronoi diagram; Section 6.2.4 will discuss how the algorithm adds new points to satisfy these requirements.

6.2.3 Robust and Accurate Voronoi Diagrams in Three Dimensions

Given a distribution of points on the object's surface, the Goldak *et al.* algorithm [GYKD91] next builds a Voronoi diagram from these points. Bowyer [Bow81] describes a relatively straightforward algorithm for the Voronoi diagram of 3D points. This algorithm implicitly assumes infinite-precision arithmetic, however. Finite-precision arithmetic causes the algorithm to enter inconsistent states from which it cannot recover. Inagaki *et al.* [ISS92] discuss modifications to the algorithm that prevent inconsistencies. These modifications make the algorithm more robust, but they also reduce the accuracy of the Voronoi diagram it produces. One might think that some inaccuracy would be tolerable because the Voronoi diagram and the medial-axis surface based on it need not be exact to produce a tight-fitting sphere-tree. Inaccuracy in the Voronoi diagram can cause subtle problems in the medial-axis surface, however, as Section 6.2.4 explains. An algorithm that builds robust and accurate Voronoi diagrams is thus an important goal. The end of this section describes a new algorithm that approaches that goal.

A Basic Algorithm

Bowyer's algorithm builds the Voronoi diagram incrementally. It starts with the Voronoi diagram for a tetrahedron of "dummy" points that bounds all the "real" points. The Voronoi diagram for this tetrahedron is trivial: the Voronoi cells for the four dummy points meet at a single Voronoi vertex in the center of the tetrahedron, and four other "dummy" vertices located at "infinity" complete the cells. The algorithm adds each of the real points one at a time, extending the diagram. By starting with the tetrahedron of dummy points, the algorithm ensures that each real point will be inside the convex hull of points already in the diagram, reducing the number of special cases. Each new point adds to the diagram one new cell, the region of space closer to that point than to all the existing points. This space was formerly divided amongst cells associated with existing points, so adding the

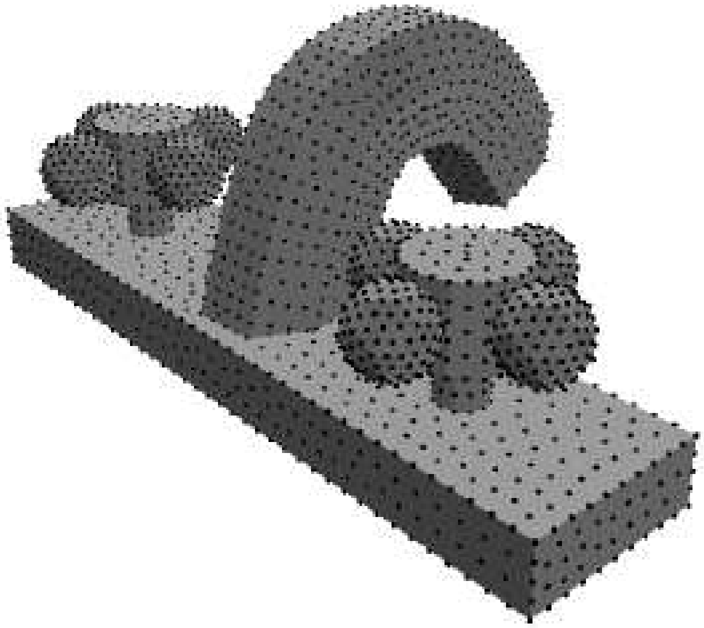


Figure 6.9: 3325 points on a bathroom faucet, placed by the algorithm using the cluster heuristic.

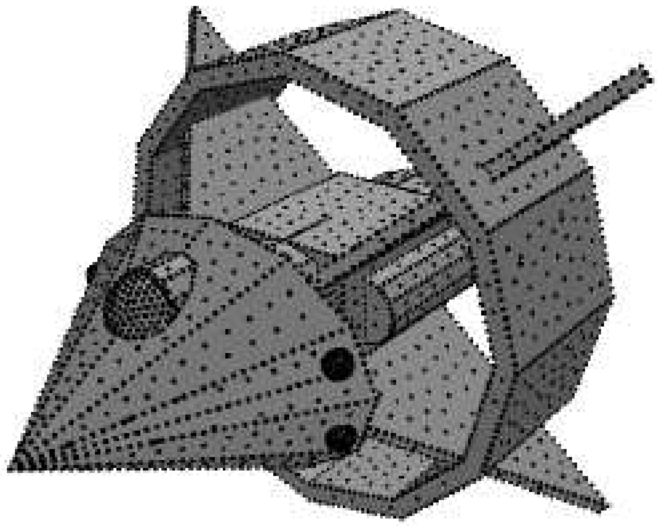


Figure 6.10: 6792 points on a spaceship, placed by the algorithm using the cluster heuristic.

new cell must cause existing cells to shrink. To shrink a cell, the algorithm deletes some vertices of the cell and replaces them with new ones. Bowyer describes the deletion and replacement in terms of 2D examples. Since the generalization to 3D situations is not always clear, I provide some details here.

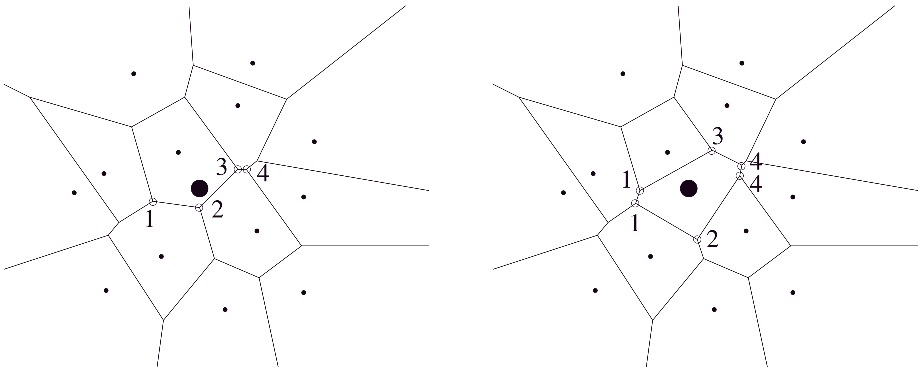
Bowyer's algorithm chooses the vertices to delete as follows. Recall from Section 6.2.1 that each vertex is at the circumcenter of four points, its forming points. In theory, the algorithm should delete every vertex that is closer to the new point than to any of its forming points (from which it is equidistant). This conclusion follows because the vertex represents, for each forming point, one limit of the space closer to that point than to any other points; this limit clearly needs to be revised if the vertex is actually closer to the new point. In practice, issues of arithmetic accuracy may dictate that the algorithm should not delete some of these vertices, but such issues are beyond the scope of Bowyer's algorithm. The algorithm initializes the set of deletable vertices with any vertex that is closer to the new point than to its forming points; the simplest-possible linear scan of the vertices suffices to find such a vertex, because this step is not the performance bottleneck in practice. It then adds to the deletable set any neighboring vertex (connected to the vertex by an edge of a cell) that is also closer to the new point. It continues this search transitively until it finds no more deletable vertices.

Having identified the deletable vertices, Bowyer's algorithm builds the new vertices that replace them, the vertices that define the cell around the new point. There will be one new vertex for each pair of a deletable vertex, v_d, and a neighbor, v_u, of v_d which is not itself deletable. The vertices v_d and v_u have in common three of their four forming points. These three points plus the new point define the position of the new vertex: its position is the circumcenter of the four points. These four points define a tetrahedron, and the circumcenter is the center of the sphere that circumscribes this tetrahedron. Schultze and Sevenoak [SS23] describe how to find the center of this sphere: for any three edges of the tetrahedron that meet at the same tetrahedral corner, build the perpendicular-bisector plane for each edge, and the intersection of these three planes is the circumcenter. Goldman [Gol90] gives a fast algorithm that computes the intersection of three planes from properties of the determinant. To illustrate the replacement of deletable vertices with new vertices, Figure 6.11 shows a 2D example. In two dimensions, v_d and v_u share two of three forming points, and the new vertex is the center of circle containing these points and the new point.

To complete the replacement of vertices, the algorithm updates some bookkeeping information. The data structure for a point stores a list of all the vertices that the point "forms," that is, the vertices for which it is a forming point. The algorithm removes each deletable vertex from the list associated with its forming points. This operation is possible because each vertex references its four forming points. A vertex also references its four neighboring vertices. For each deletable vertex, v_d, the algorithm removes the reference in each undeletable neighbor, v_u. As a replacement, v_u gets a reference to the new vertex built from v_u and v_d's shared forming points. The new vertex stores v_u as one of its neighbors. Its other three neighbors are new vertices, those with which it shares three forming points. The simplest way to find the neighboring new vertices is to use brute force: for each pair of new vertices, check all their forming points for three in common.

Bowyer's algorithm assumes that all the points it processes are unique. To make sure this assumption is not violated, the algorithm checks for any coincident points before starting to build the diagram. The end of Section 6.2.1 mentions that Turk's spatial hashing scheme [Tur90b] finds coincident vertices within a specified tolerance (to account for floating-point inaccuracy). The same approach applies to finding coincident points.

For randomly distributed points, Bowyer's algorithm performs well. Points on the surface of an object have more structure, however, and this structure can cause problems for the algorithm. The

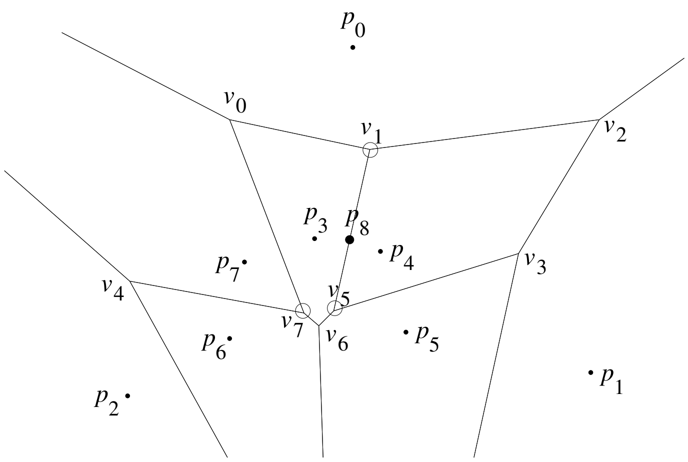


(A) (B)

Figure 6.11: (A) Adding the darkened point necessitates deleting the Voronoi vertices 1 through 4. (B) The new diagram. The number by a new vertex indicates the deleted vertex to which it corresponds.

first problem occurs during the search for deletable vertices. Recall that in theory, the algorithm should delete any vertex that is closer to the new point than it is to its forming points. At issue is what the algorithm should do for a vertex that is almost the same distance from the new point and its forming points, that is, for a *marginally-deletable* vertex. If the algorithm chooses not to delete a marginally-deletable vertex, then there may be no deletable vertices at all. In this situation, the algorithm will be unable to change any cells to accommodate the new point, which is clearly an error. If, on the other hand, the algorithm chooses to delete a marginally-deletable vertex, then the algorithm may delete too many vertices. When the algorithm creates a new vertex for each deleted vertex, it will find that some new vertices share three forming points with more than four vertices; these new vertices need to have more than four neighbors, which is an error. Figure 6.12 shows an example of this problem in two dimensions. The figure shows the Voronoi diagram before the new point, p_8, is added. The vertex v_1 is significantly closer to p_8 than to its forming points, so it is deletable. Vertices v_5, v_6 and v_7 are all marginally-deletable (assuming that the scale of this figure is close to the minimum resolution of floating-point arithmetic). If the algorithm deletes only v_1 and v_5, then there are no problems. If, however, it also deletes v_7, then some new vertices will have too many neighbors. A detailed analysis of this situation is tedious, but Figure 6.12 summarizes the details for readers interested in hand-simulating what happens. When the algorithm compares a vertex's distances to the new points and to its forming points, the algorithm should consider the distances equal if they are with in a small tolerance; such use of tolerances is common practice for floating-point computation. No single value of this tolerance solves the problem in all cases, however.

A second robustness problem for Bowyer's algorithm occurs when it computes the position of a new vertex. Recall that this position is the circumcenter of the vertex's forming points. If the new vertex results from a mistaken choice of deletable vertices, the forming points may be coplanar. In this case, the circumcenter computation will fail because it will try to find the intersection of three disjoint, parallel planes. Figure 6.13 shows a 2D analog. The figure shows a Voronoi diagram before the new point, p_2, is added. The vertex v_0 is closer to p_2 than to its forming points, so it is deletable. Vertex v_1 is marginally-deletable, as the dashed circle illustrates. If the algorithm chooses not to delete v_1, then it will create a new vertex whose forming points are p_2 and the two forming points shared by v_0 and v_1: p_0 and p_1. But p_0, p_1 and p_2 are approximately colinear (the 2D analog to coplanar), so the algorithm cannot compute their circumcenter.



deleted	undeleted	new	forming points	neighbors
v_1	v_0	v_8	p_0, p_3, p_8	$v_0, v_9, v_{11}, v_{12}, v_{14}$ (too many)
v_1	v_2	v_9	p_0, p_4, p_8	v_2, v_8, v_{10}
v_5	v_3	v_{10}	p_4, p_5, p_8	v_3, v_9, v_{11}
v_5	v_6	v_{11}	p_3, p_5, p_8	$v_6, v_8, v_{10}, v_{12}, v_{14}$ (too many)
v_7	v_0	v_{12}	p_3, p_7, p_8	$v_0, v_8, v_{11}, v_{13}, v_{14}$ (too many)
v_7	v_4	v_{13}	p_6, p_7, p_8	v_4, v_{12}, v_{14}
v_7	v_6	v_{14}	p_3, p_6, p_8	$v_6, v_8, v_{11}, v_{12}, v_{13}$ (too many)

Figure 6.12: An example in which deleting too many vertices causes the Voronoi-diagram algorithm to fail.

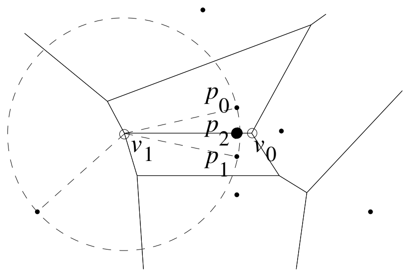


Figure 6.13: An example in which the Voronoi-diagram tries to find the circumcenter of colinear points.

A Robust Algorithm

Inagaki *et al.* [ISS92] describe a technique designed to improve the robustness of an algorithm, like Bowyer's, which builds a Voronoi diagram incrementally. The technique is based on the observation that a Voronoi diagram has three topological properties:

1. the diagram has as many cells as points;
2. each cell is simply connected (i.e., it is not the union of several disjoint volumes, nor is one volume that is "pinched" down to a neck consisting of a single point); and
3. two cells cannot share two or more faces.

Inagaki *et al.* state that when the algorithm adds a new point, it preserves these topological properties only if the deletable vertices satisfy five constraints:

1. the set of deletable vertices is nonempty;
2. the deletable vertices form a connected subgraph;
3. at least one vertex from each cell is *not* deletable;
4. the deletable vertices on each cell form a connected subgraph; and
5. the undeletable vertices on each cell form a connected subgraph.

These constraints are combinatorial rather than numerical in nature, so they are not affected by floating-point inaccuracy. An algorithm can thus remain robust in the presence of inaccuracy if it chooses deletable vertices that satisfy the five constraints. A vertex whose addition to the current set of deletable vertices does not violate the constraints is *legally deletable*.

In passing, Inagaki *et al.* also note that for each pair of a deletable vertex, v_d, and an undeletable vertex, v_u, the new vertex must lie on the edge between them. Although Inagaki *et al.* do not mention it, this observation allows the algorithm to approximate the circumcenter of coplanar forming points. The idea is to use the position of v_u as the circumcenter. As justification, note that the "true" circumcenter of four coplanar points is infinitely far away from the points. To approximate going infinitely far away while staying on the edge the algorithm must stop at v_d or v_u; v_u is preferable because it is not closer to the new point than to its forming points.

Inagaki *et al.* suggest a greedy search to find the deletable vertices that meet the five constraints. The search starts at the vertex closest to the new point being added; this vertex meets the constraints and so is deletable. From this vertex the search branches out to neighboring vertices, adding to the deletable set any which do not violate the constraints. The search continues until all neighbors violate the constraints.

This greedy search is a straightforward addition to Bowyer's basic algorithm, replacing its search for all theoretically-deletable vertices. Each time the search advances to another vertex, v, it determines whether deleting v would violate the constraints on any cell containing v. There is one such cell for each of v's four forming points. A simple traversal of a cell's undeletable vertices, skipping v, reveals whether they would remain connected with a deleted v. A similar traversal of the cell's deletable vertices, this time including v, verifies their connectivity.

Adding this search to Bowyer's algorithm, and adding the circumcenter approximation for coplanar points, does make the algorithm more robust. In my experience, this modified algorithm does not fail on any of the sets of points that make Bowyer's algorithm fail.

The disadvantage of this algorithm is its loss of accuracy. The constraints on the deletable vertices ensure that the diagram has the topological properties of a Voronoi diagram, but these constraints ignore the geometric properties, that each cell represent the space closer to one point that to any other point. When choosing vertices to designate deletable, the greedy search does not consider the impact of its decisions on the accuracy with which the diagram approximates the geometric properties. As an example, the search may choose a vertex which is marginally closer to the new point than to its forming points, and by making this vertex deletable the search may prevent itself from later choosing another vertex which is much closer to the new point. Such problems do seem to occur in practice, because the Inagaki *et al.* algorithm is significantly less accurate than Bowyer's algorithm on points that do not make the latter fail.

A Robust and Accurate Algorithm

A robust and accurate algorithm must do more than just meet the Inagaki *et al.* constraints. It must replace the greedy search with a more elaborate search, one that identifies vertices whose deletion will compromise the geometric properties of the Voronoi diagram as little as possible. One approach is to define a "value" for the deletion of a vertex, where the value increases with the likelihood that deleting the vertex will produce an accurate diagram. The search for deletable vertices thus becomes a constrained maximization problem: find the set of vertices with the highest value that also meet the Inagaki *et al.* constraints.

One definition of a vertex's value is its distance to its forming points minus its distance to the new point. This value is positive if the vertex is theoretically deletable, in the sense used by Bowyer's algorithm. The value increases with the extent to which the vertex intrudes on the cell which will be associated with the new point; a high value means the vertex is substantially closer to the new point than to its forming points, so it clearly belongs in the new point's cell and should not be part of its forming points' cells. Thus, whenever the algorithm does not delete a vertex with high value it makes the diagram less of a true Voronoi diagram.

Finding the set of legally deletable vertices with the maximum possible value is a difficult problem. A simpler problem is finding a set with maxim*al* value, meaning a set to which it is impossible to add another vertex that increases the overall value. A set with maximal value is not as useful as a set with maximum value, but it is preferable to the set found by the Inagaki *et al.* approach, which ignores value altogether.

The simplest algorithm for finding a set of legally deletable vertices with maximal value is to use a greedy algorithm. This algorithm is greedy in the sense that when choosing vertices to add to the deletable set, it always takes the highest-valued vertex that it can find. I call this strategy a *highest-first* approach to distinguish it from the greedy approach of Inagaki *et al.*, which does not consider value at all.

The highest-first algorithm's main data structure is a priority queue of vertices, which releases high-valued vertices first. The algorithm initializes this queue to the vertex with highest value, found by a simple linear search since this step is not the performance bottleneck. The main loop of the algorithm starts by taking the first vertex from the priority queue. If this vertex is not legally deletable, the algorithm adds it to a set of "illegal" vertices and takes the next vertex from the queue until it finds a legally deletable vertex. (Determining if a vertex is legally deletable is the same as in the Inagaki *et al.* algorithm.) The algorithm adds this vertex to the deletable set. It then adds to the queue the vertex's positive-valued neighbors. The last step of the main loop adds back onto the queue the set of illegal vertices found at the beginning of the loop; the algorithm should reconsider these vertices because they have high value, and it is possible some of them will no longer violate the constraints once other vertices have been added to the deletable set. The algorithm then repeats its main loop. The loop terminates when the algorithm finds no legally deletable vertex on the queue.

To compare the accuracy of the highest-first and Inagaki *et al.* approaches, I ran empirical tests. The key part of each test is a routine that estimates the inaccuracy of a finished Voronoi diagram. There are many possible ways to make this estimate; I chose one which is simple to implement. The test begins with the premise that a Voronoi cell should be a convex polyhedron which contains one of the original data points. For a line segment from that data point to a corner of the cell, every position on the segment should be closer to the data point than to any other data point. The test constructs this segment for each corner of each cell, and samples positions along the segment; the current implementation uses three evenly-spaced samples. At each sample, the test measures the distance, d, from the sample to the cell's data point. It also measures the distance from the sample to every other data point in the diagram, and finds the minimum of these distances, d'. If $d' < d$, then the diagram is inaccurate. A measure of the inaccuracy at this sample is $d - d'$, for which a higher value means more inaccuracy. The test maintains the sum of these inaccuracy measures, and reports as its results the average inaccuracy per sample. This result is not the final word on accuracy, but it does convey some useful information about the error in a Voronoi diagram.

The tests themselves involved two different sets of data. Bowyer's algorithm fails on these data sets, both when it deletes marginally-deletable vertices and when it does not. All the algorithms in these tests use double-precision arithmetic. Figure 6.9, from Section 6.2.2, shows the data points for the first accuracy test, 3325 points distributed over the surface of a bathroom faucet. For this data, the Inagaki *et al.* algorithm produces a diagram with 35205 vertices. The average inaccuracy is 0.71519 units per sample, where the dimensions of the model are 5.78 by 4.09 by 10.0 units. The highest-first algorithm produces a 20694-vertex diagram for this data, and the average inaccuracy per sample is 6.02201×10^{-5}. The inaccuracy of the highest-first algorithm is thus more than four orders of magnitude lower. The second test involved 6792 points distributed over the surface of a spaceship, as shown in Figure 6.10 from Section 6.2.2. The Inagaki *et al.* algorithm produces a diagram with 66934 vertices for this data. The average inaccuracy is 0.322203 units per sample, where the dimensions of the model are 3.10 by 2.80 by 3.50 units. The highest-first algorithm produces 46893 Voronoi vertices for this data, with an average inaccuracy per sample of 2.90961×10^{-9} units. The inaccuracy of the highest-first algorithm is thus more than eight orders of magnitude lower.

From these tests, it is clear that the highest-first algorithm is significantly more accurate. Further evidence for this conclusion comes from the number of times the algorithm has to approximate the circumcenter of coplanar forming points: it makes this approximation 52 times for the first test and 5 times for the second, whereas the Inagaki *et al.* algorithm makes it 4066 and 6452 times, respectively. The diagrams the highest-first algorithm produces are not only more accurate but also more compact; in the first test, the number of vertices it produces is 59 percent of the number produced by the Inagaki *et al.* algorithm, and in the second test it is 70 percent. For the question of performance, the two tests are not conclusive. In the first test, the highest-first algorithm is faster, taking 154 seconds on an Hewlett Packard 9000 Model 735 compared to the 240 seconds required by the Inagaki *et al.* algorithm. The highest-first algorithm is slower in the second test, however, taking 963 seconds compared to the Inagaki *et al.* algorithm's 859 seconds.

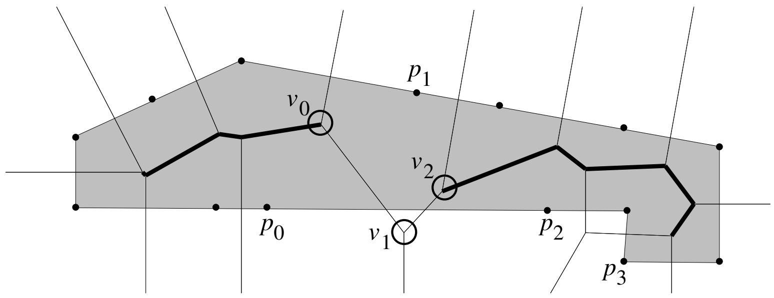


Figure 6.14: A break in the connected component of adjacency.

6.2.4 Gap-Crossing Cells

The previous section explains how to build a Voronoi diagram that accurately reflects the points on the surface of an object. Even when it is accurate in this sense, the Voronoi diagram may cause the medial-axis surface to be inaccurate. The problem stems from the surface points themselves. The algorithm from Section 6.2.2 distributes those points as evenly as possible in preparation for the Voronoi diagram, but certain problems with the distribution become apparent only after the diagram has been built. These problems and their solutions are esoteric, but I found these solutions important in practice, especially for objects with complicated shapes.

These problems involve the adjacency between Voronoi vertices. Each vertex is adjacent to its four neighbors, the vertices connected to it by an edge of a cell face. For an object in one piece (with no disconnected "islands") the adjacency between vertices should define one connected component of vertices; multiple independent components should not occur. Even though it is acceptable for this component to only approximate the contours of the object's true medial axis, the component should not deviate in a qualitative way; for example, if the object is a human figure the component should not exit the body at one knee, cross the "gap" between the legs and re-enter the body at the other knee. Section 6.3 explains why these properties of adjacency are important for sphere-trees, but it should not be difficult to appreciate that defects in adjacency are abnormal in an intuitive sense.

Figure 6.14 shows a 2D example of the first problem, a break in connectivity. Voronoi vertices v_0 to v_2 should be connected through v_1, but v_1 is outside the object so the algorithm removes it. The source of the problem is the cell around p_1. Notice that p_1 lies on one side of the object, but the cell intersects a different side (i.e., the edge between v_0 and v_1 intersects the side containing p_0, and the same for the edge between v_1 and v_2). No other cells have this property. The only cells that intersect more than one side of the object are associated with corner points, like p_3, and each such point lies on both sides that its cell intersects.

The second problem occurs when the medial-axis surface crosses a "gap" outside the object, and Figure 6.15 shows a 2D example. Voronoi vertices v_0 and v_1 are adjacent, but this adjacency jumps from one knee of the object to the other. Notice that both vertices are inside the object, so the algorithm does not break the adjacency. The problem is once again related to how cells intersect the object. In particular, the cell for p_0 intersects a side of the object on which p_0 does not lie. The same holds for p_1's cell. As in Figure 6.14, no other cells have this property.

The cells that cause both these problems are examples of *gap-crossing cells*. These cells are distinguished by the *clusters* they intersect. Recall from Section 6.2.2 that a cluster is a set of adjacent coplanar triangles from a triangulated polyhedron; the two triangles composing a side of

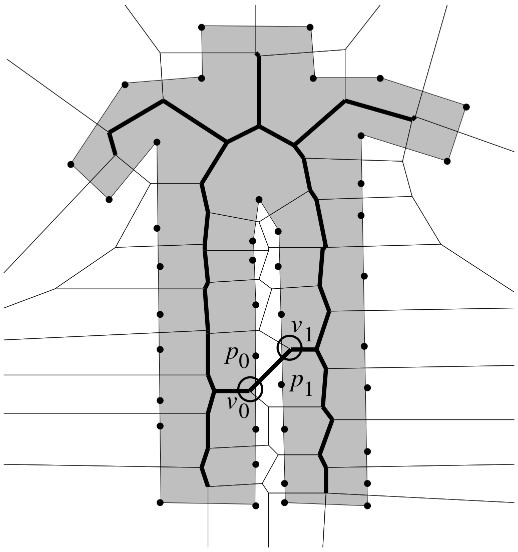


Figure 6.15: The medial axis crosses a gap.

a cube, for example, are one cluster. Say the cells around p_j and p_k are adjacent, that is, the cells share a face. By the strictest definition, these cells are gap-crossing if the face intersects a cluster that does not contain both p_j and p_k. Notice that in both Figure 6.14 and Figure 6.15 the only cells that are gap-crossing by this definition are those which cause the adjacency problems. A looser definition of gap-crossing is sometimes useful in practice. Consider once again the cells around p_j and p_k, and the clusters intersected by the face they share. The cells are loosely gap-crossing if any intersected cluster does not either contain both p_j and p_k or adjoin the clusters that do contain them. The end of this section will explain why this looser definition has efficiency advantages in practice.

Gap-crossing cells are related to the distribution of surface points. The relationship is not obvious, however. For the 2D objects in Figure 6.14 and Figure 6.15, the gap-crossing cells appear to occur near places where the points are "uneven." In three dimensions, however, it is much less clear what makes points "uneven" enough to cause gap-crossing cells. A distribution of points that is extremely dense all over the object might solve the problem, but this solution is overkill; computing a Voronoi diagram for this many points would be intractable in practice. I believe the only solution is to add a postprocess to the medial-axis-surface algorithm, one which identifies and eliminates gap-crossing cells after the Voronoi diagram has been built.

The first part of this postprocess is an algorithm that locates the gap-crossing cells. The most reliable approach is the obvious one: triangulate each face of each cell in the Voronoi diagram, find all intersections between each such face and the surface of the object, and note any intersections with clusters that make the cell gap-crossing. Triangulating a cell face is simple because it is a convex polygon. Testing whether one of these triangles intersects a triangle from the object's surface involves a modest number of steps; Laidlaw *et al.* [LTH86] describe the process for the more general case of two polygons, and a few of these steps are simpler for triangles [Hub90]. It is clearly inefficient for the algorithm to test every triangle from a cell face against every triangle from the object. To improve efficiency, the algorithm uses an octree which stores the object's triangles. The medial-axis-surface algorithm already builds an octree, which it uses to find the vertices inside the

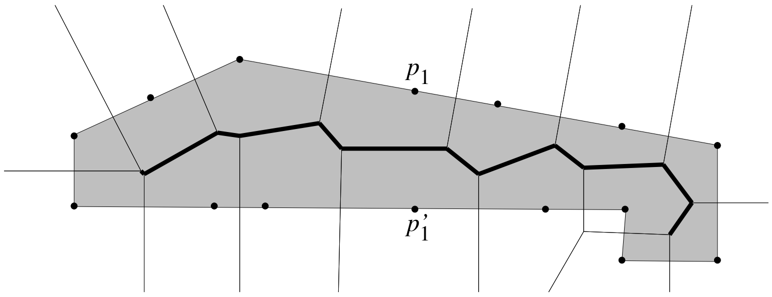


Figure 6.16: Adding a surface point fixes the break in the connected component.

object, as Section 6.2.1 explains. In the current context, the algorithm passes each cell-face triangle down through the octree as if adding the triangle to the octree. This process identifies leaf octants which contain the triangle. The algorithm then tests the triangle against only the object triangles associated with these leaves. This approach is similar to what Shaffer and Herb [SH92] do for collision detection. Another optimization follows from the sharing of a face between two adjacent cells. The algorithm need not process the triangles of this face twice. It is important to note, though, that the algorithm must consider the points in both cells to determine if either cell is gap-crossing. Figure 6.14 shows an example. The face between v_0 and v_1 separates the cells around points p_0 and p_1. This face is part of a gap-crossing cell, but the algorithm would not notice the problem if it considered only p_0.

Having found a face of a gap-crossing cell, the algorithm selectively adds new surface points and updates the Voronoi diagram. Figure 6.16 shows how adding one point, p_1', mends the break in connectivity from Figure 6.14. The position of p_1' is related to the position of the point, p_1 inside the gap-crossing cell. Specifically, p_1' is the normal projection of p_1 onto the cluster, c, that the cell intersected (i.e., the direction of the projection is normal to c). Similarly, Figure 6.17 shows how projecting p_0 and p_1 into p_0' and p_1', respectively, eliminates the gap-crossing cells from Figure 6.15.

There is a specific reason for using the normal projection of the gap-crossing cell's point, p, to determine the new point to add, p'. Adding p' to the diagram changes the gap-crossing cell so that it has a face between p and p'. The new cell face will be normal to line between p and p'. If this line is normal to the cluster that the cell had intersected, then the new cell face will be parallel to this cluster and will not intersect it. Since the cell must be convex, the rest of it must be on the other side of the new cell face; thus, the cell can no longer intersect the cluster.

A face of a gap-crossing cell may intersect more than one cluster that it should not intersect. The algorithm projects the point inside this cell, p, onto each intersected cluster. Thus, after the algorithm has processed every face of the cell and updated the Voronoi diagram, the cell can no longer be a gap-crossing cell.

Each new point, p', will create its own cell in the updated Voronoi diagram. This new cell shares a face with the cell that algorithm just corrected, so that face cannot cause the new cell to be gap-crossing. Its other faces can cause problems, however. Of particular concern are the faces the new cell shares with other new cells. The algorithm must be certain not to miss these faces just because they are associated with new points. To avoid this problem, the algorithm maintains a list of all the new points it adds. When it has finished processing the existing points, the algorithm processes the new points, checking any face shared by two new points' cells. Thus the algorithm has multiple passes, and it keeps going until none of the new points it adds are in gap-crossing cells.

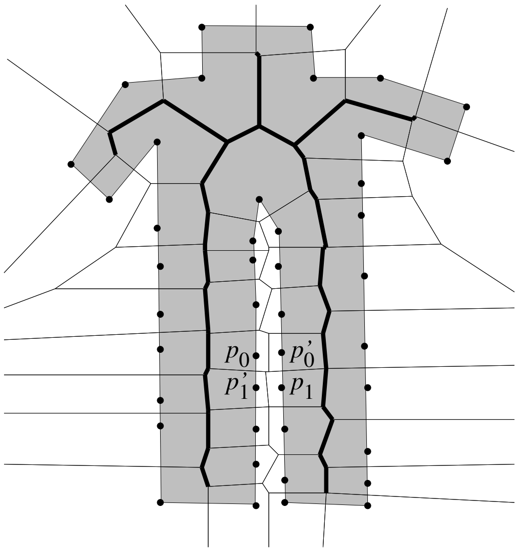


Figure 6.17: Adding a surface point fixes the gap-crossing cells.

The algorithm produces a new point by projecting an existing point onto plane of a cluster. This new point may not actually lie within the cluster, though. Instead, the point may be a *free-floating* point that nowhere touches the surface of the object. For a highly nonconvex object, a free-floating point may be quite far away from the object. Such a point will introduce errors into the medial-axis surface, so the algorithm needs a way in avoiding free-floating points. The algorithm does have some leeway in avoiding free-floating points. Recall that a face that intersects the wrong cluster and makes a cell gap-crossing is shared with another cell. The projection of one cell's point need not be free-floating even if the other is, as Figure 6.18 illustrates. The algorithm should thus try both projections in the hopes that one will not produce a free-floating point.

If both projected points are free-floating, then the algorithm must choose one and snap it to the surface of the object. There is a fundamental disadvantage to snapping, though. Recall that positioning a new point by normal projection onto a cluster gives the updated Voronoi diagram a desirable property, that the new cell face will not intersect the cluster. Snapping the projected point changes the orientation of the new cell face, however, and may cause it to intersect the cluster. Figure 6.19 shows an extreme example of this problem. Here, the face between point p_0 and p_1 intersects cluster c. Projecting p_0 and snapping it to the closest point on c produces a point coincident with the existing point, p_2. Updating the Voronoi diagram will thus produce no change whatsoever, and the gap-crossing cell will persist.

To solve this problem, the algorithm snaps the projected point to multiple locations. Instead of snapping the point to the closest location on the cluster, it snaps the point to the closest location on several individual triangles from the cluster. The particular triangles are those that were intersected by the gap-crossing cell's face. Figure 6.20 illustrates this strategy. Snapping to triangles is advantageous because triangles are convex. When the algorithm adds the snapped point to the Voronoi diagram, the gap-crossing cell will receive a new face, but the triangle will be entirely on one side of this face. The snapped point thus eliminates an intersection between the gap-crossing cell and one triangle of the cluster. Collectively, the snapped points for all the triangles eliminate all such intersections.

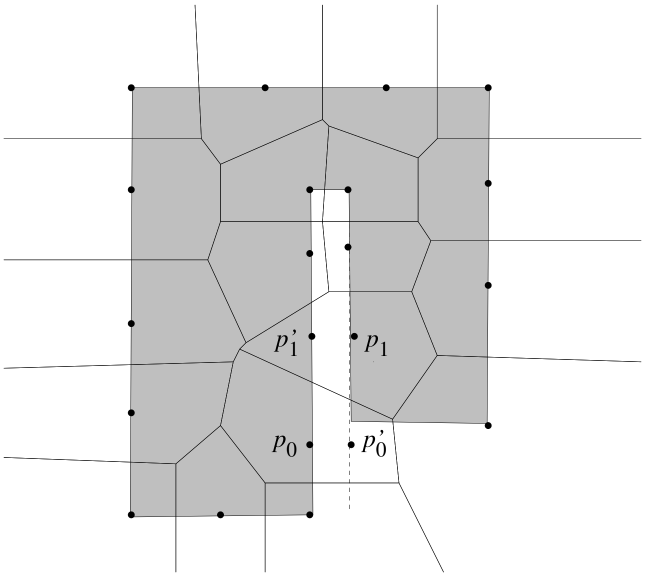


Figure 6.18: The projection of p_0, p_0', is free-floating, but the projection of p_1, p_1', is not.

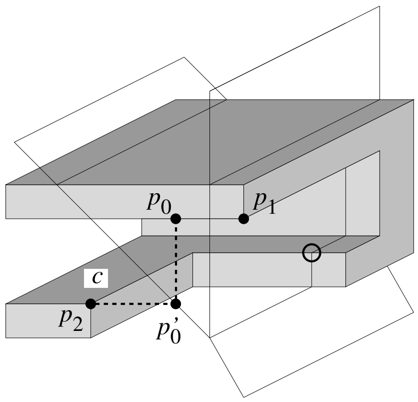


Figure 6.19: Projecting p_0 requires snapping, but it does not eliminate intersection with c.

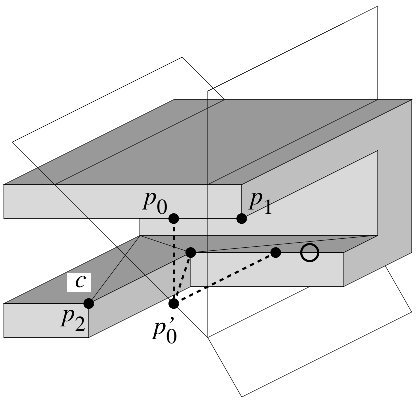
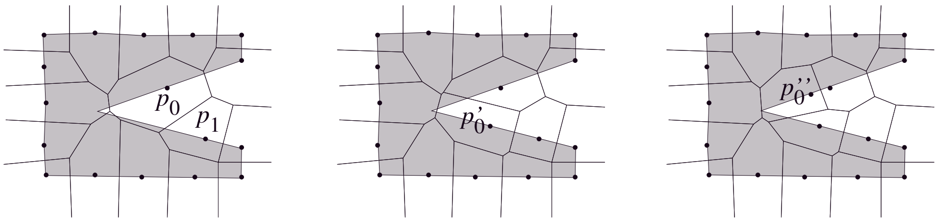


Figure 6.20: Snapping p_0 to intersected triangles of c.

To implement this strategy, the algorithm first tests whether the projected point is inside the cluster. The simplest version of this test compares the point with each triangle in the cluster; the point must be inside one such triangle to be inside the cluster. Determining if a point is inside a triangle is straightforward. If the projected point is outside the cluster, the algorithm proceeds with the snapping. Snapping a point to the closest location on a triangle is also a simple process. The essential step is finding the closest location. Section 5.3.2 has already discussed this step, in the context of energy functions for simulated annealing.

Regardless of whether or not the algorithm had to snap the new point, the point may be nearly coincident with a point already in the Voronoi diagram. There are several reasons for not adding the new point in this case. If the new point is exactly coincident, its addition will not chance the diagram at all. If the distance from the new point to the existing point is small, close to the computer's minimum resolution for arithmetic, then adding the new point can cause problems in the diagram; the robust techniques from Section 6.2.3 will not fail in this situation, but they are likely to produce inaccurate Voronoi cells. On the other hand, the algorithm chose to add the new point for a reason—to fix a gap-crossing cell—and this problem may remain unsolved if the algorithm does not add the point. Other points the algorithm adds later may also fix this gap-crossing cell as a fortunate side effect, in which case the point is not strictly necessary, but there are no guarantees. I found no real solutions to this dilemma. In my experience, nearly-coincident points are rare but they do occur. Their occurrence is probably related to inaccuracy in the Voronoi diagram. If the diagram contains a gap-crossing cell, and the algorithm deduces that a point coincident with an existing point should solve the problem, then some aspect of the cell must be incorrect (e.g., it might be nonconvex). My implementation does not add nearly-coincident points. Fortunately, the other points it adds do seem to correct all the gap-crossing cells. A better solution to the dilemma is a good topic for future research.

The current approach has another weakness. There are situations in which building p' by normal projection will cause an inordinate number of extra passes. Figure 6.21 shows a 2D example. In Figure 6.21(A), the point p_0 is inside a gap-crossing cell, which intersects the cluster adjacent to p_0's cluster. The normal projection of p_0 is p_0'. As Figure 6.21(B) shows, the cell around p_0' is also gap-crossing, intersecting the cluster that contains p_0. The normal projection of p_0' is p_0''. But as Figure 6.21(C) shows, the cell around p_0'' is still gap-crossing, and has effectively recreated the situation from Figure 6.21(A). This process can continue indefinitely, as the projected points approach the place where adjacent clusters touch.



(A) (B) (C)

Figure 6.21: A problem with normal projection.

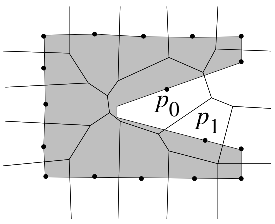


Figure 6.22: Another problem with normal projection.

Note that under the looser definition of gap-crossing, the cell around p_0 in Figure 6.21(A) is not gap-crossing because p_0 and p_1 are on adjacent clusters. The looser definition is intuitively reasonable for objects with complicated shapes. In such objects, gaps of significant scale are likely to involve more than just one pair of adjacent clusters. Allowing cells to cross smaller gaps like the one in Figure 6.21(A) should not cause problems. In general, the algorithm processes fewer cells when it uses the looser definition. Since the algorithm is time-consuming in practice, reducing the work it performs is important. My implementation uses the looser definition of gap-crossing cells to gain this efficiency advantage.

The looser definition does not fully eliminate the problem, though. Figure 6.22 shows a situation similar to Figure 6.21(A). In this case, however, the clusters are not adjacent, being the the sides of a "U" instead of a "V." The bottom of the "U" must be small for this problem to be significant, and such situations are probably rare, but it is still a concern.

If the algorithm could detect these situations, there are two steps it could take. First, it could avoid the normal projection when placing the new point. In Figures 6.21 and 6.22, for example, the position p_0' by algorithm could project p_0 straight across the "V" or "U". This projection puts the new cell face between p_0 and p_0' on the bisector of the "V" or "U". This new face cannot intersect the sides of the "V" or "U". It can intersect the bottom, though. The algorithm could explicitly check for this intersection, and if it exists the algorithm could introduce a second new point, positioned on the bottom, to prevent it. These two steps would be helpful, but it is difficult for the algorithm to detect when they are necessary. As a consequence, they are not part of my implementation.

All the ideas in this section assumes that the Voronoi diagram is fully accurate. Even the improved algorithm of Section 6.2.3 cannot always satisfy this assumption. For example, the Voronoi vertices that lie on a cell face do not always define a flat, convex polygon. In extreme situations, the polygon can twist back on itself. The algorithm which detects gap-crossing cells currently assumes this twisting cannot happen, and the algorithm can make mistakes if its assumption is incorrect. Inaccuracy may also cause new points to coincide with existing points, as mentioned earlier. I have

```
medial_axis_vertices(O)
{
    /* See Section 6.2.2. */
    randomly distribute points on O's surface
    for each iteration of relaxation
        for each cluster on O {
            accumulate forces on cluster's points
            apply forces to update positions of cluster's points
        }

    /* See Section 6.2.3. */
    P ← points with coincidents removed
    D ← Voronoi diagram for P

    /* See Section 6.2.4. */
    do {
        P' ← ∅
        for each point p ∈ P
            for each cell-face f of p not already processed
                if (f intersects inappropriate clusters)
                    add to D and to P' new points that correct gap crossing
        P ← P'
    } while (P ≠ ∅)

    /* See Section 6.2.1. */
    coalesce coincident vertices of D
    return vertices of D
}
```

Figure 6.23: Pseudocode for building Voronoi vertices on object O's medial-axis surface.

encountered these problems occasionally in practice, but they have not been severe enough to stop the algorithm from working.

Gap-crossing cells are a difficult problem to handle. The ideas from this section are not perfect, but they do seem to work well enough in practice. In all my tests, the algorithm to eliminate gap-crossing cells has corrected the problems with Voronoi vertex adjacency, allowing the sphere-tree algorithm to proceed in situations that it could not otherwise process.

6.2.5 Summary

Figure 6.23 presents pseudocode for building vertices on an object's medial-axis surface. Details of this code's performance on specific examples will appear in Section 6.3.5, which discusses the overall performance of building sphere-trees from medial-axis surfaces.

6.3 Building Sphere-Trees from Medial-Axis Surfaces

The algorithm from the previous section identifies Voronoi vertices on an object's medial-axis surface. Recall from Section 6.2.1 that a Voronoi vertex is the center of a sphere. This section describes an algorithm that turns these spheres into a sphere-tree.

The algorithm adds nodes to the sphere-tree from the top down. The root is the bounding sphere for the object, which the algorithm computes using Ritter's approach [Rit90]. The algorithm builds the children of the root by *merging* spheres associated with the Voronoi vertices. To merge a pair of adjacent spheres, the algorithm replaces them with one bigger sphere that covers that parts of the object that they cover. The algorithm chooses spheres to merge by applying a cost function, one that encourages spheres that fit the object tightly. By repeatedly merging pairs of spheres, the algorithm gradually arrives at a small set of spheres whose union approximates the shape of the object. The algorithm stops merging when the set contains the number of spheres that the root needs for children; my implementation gives each sphere eight children, to match the octree-based sphere-trees. To give one of these spheres, s, its own children, the algorithm uses almost the same process. The algorithm again merges spheres associated with Voronoi vertices, but this time it starts with a subset of all the vertices: those vertices whose spheres ultimately became merged to form s. Thus, the algorithm adds nodes to the tree from the top down, but it builds each node from the bottom up.

Section 6.3.1 describes the merging and its cost function in more detail. The spheres which result from merging approximate the shape of the object, but they do not necessarily cover it conservatively; Section 6.3.2 discusses how to grow the spheres until they do provide conservative coverage. The possibility that merging will produce redundant spheres is the topic of Section 6.3.3. Section 6.3.4 discusses the important issue of measuring how tightly the merged spheres fit. These last three sections all draw heavily on the concepts of simulated annealing from Section 5.3. Section 6.3.5 concludes the discussion of building sphere-trees from medial-axis surfaces by presenting some results and comparing the approach to the related ideas of Badler, O'Rourke and Toltzis [BOT79, OB79].

6.3.1 Merging Spheres

Typically, the algorithm from Section 6.2 generates a large number of Voronoi vertices on an object's medial-axis surface. The spheres centered at these vertices approximate the shape of the object at a high level of detail. The goal of merging is to reduce the number of spheres while maintaining as much of the detail as possible.

To merge one pair of spheres, s_1 and s_2, the algorithm replaces them with a new sphere, s_{12}. This s_{12} must cover the same part of the object's surface that s_1 and s_2 cover. For efficiency, the algorithm approximates coverage in terms of discrete points on the object's surface. Recall from Section 6.2.1 that for each Voronoi vertex on the medial-axis surface, the sphere centered at that vertex touches the object's outer surface at four places, the forming points of the vertex. Say s_1 and s_2 are two such spheres. If s_{12} bounds the forming points for s_1 and s_2, then s_{12} covers approximately the same part of the object that s_1 and s_2 cover. Figure 6.24 shows a 2D example. The algorithm uses Ritter's approach [Rit90] to make s_{12} bound the forming points. If the algorithm chooses to replace s_1 and s_2 with s_{12}, it stores with s_{12} the union of the forming points from s_1 and s_2. Any future merges involving s_{12} must cover this union of forming points. Using this policy, the algorithm guarantees that the final set of merged spheres cover all the forming points on the object's surface. The spheres may not cover some spots on the surface between the points. These spots are usually small, however, and Section 6.3.2 describes a postprocess that quickly covers these spots.

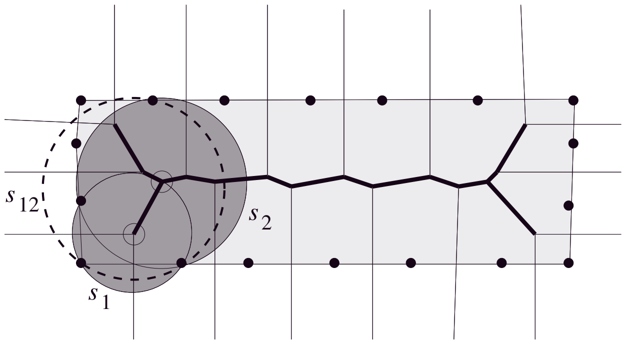
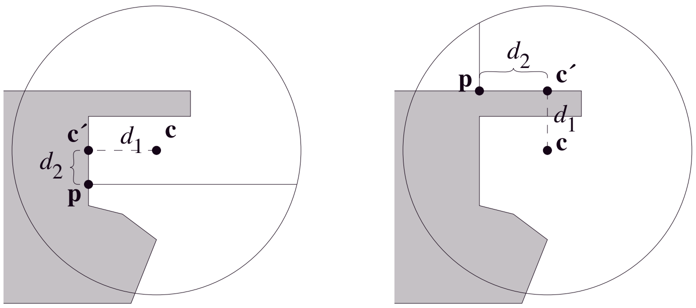


Figure 6.24: Merging s_1 and s_2 produces s_{12}, which bounds their forming points. Note that in two dimensions each vertex has three forming points.

The algorithm repeatedly merges pairs of spheres until the overall number of spheres is down to a target number. At each iteration, the algorithm solves an optimization problem: among the many pairs of spheres that could be merged, find the pair whose merger preserves the most detail. To solve this problem, the algorithm needs a set of candidate pairs to consider and a cost function to evaluate the candidates.

The algorithm uses the Voronoi diagram to identify candidate pairs for merging. Initially, each sphere is centered at a Voronoi vertex. Each such vertex is adjacent to a subset of the other vertices. Recall from Section 6.2.1 that two Voronoi vertices are adjacent if they are connected by an edge of a cell face. When the algorithm merges s_1 and s_2 to form s_{12}, it also merges adjacencies, that is, it makes s_{12} adjacent to the union of spheres adjacent to s_1 and s_2. The algorithm thus can use the heuristic of considering only adjacent pairs of spheres as candidates for merging. Such pairs are good candidates for two reasons. First, adjacent Voronoi vertices are generally closer to each other than nonadjacent vertices, and the same is true after the algorithm merges adjacencies. This property does not hold in all situations, but for a dense set of vertices on the medial-axis surface it holds frequently. The heuristic thus helps the algorithm ignore distant pairs of spheres. Ignoring these pairs makes sense intuitively because a new sphere that bounds their forming points will be quite large. The second reason for using the heuristic is that adjacent spheres respect the contours of the object more than nonadjacent spheres. If the medial-axis surface contains no gap-crossing cells, adjacency defines a single connected component of vertices that does not bypass parts of the object in unexpected ways. By merging only adjacent spheres, the algorithm follows the guidance of the connected component and avoids many undesirable merges. The techniques of Section 6.2.4 that eliminate gap-crossing cells are thus important to make merging successful.

To apply the heuristic of considering only adjacent pairs of spheres, the algorithm maintains a priority queue of sphere pairs. The queue associates each pair with the cost of merging it, and the queue is keyed so that low-cost pairs are at the head. The next paragraph discusses the cost function in detail; for now, all that is important is that low-cost mergers are desirable. Initially, the algorithm puts into the queue every pair of adjacent vertices. There are many such pairs, so this initialization is takes some time. Once these pairs are in the queue, however, the algorithm can choose spheres to merge very quickly. The pair at the head of the queue is always the best pair to merge, since the cost of merging that pair is lowest. Each time the algorithm merges the head pair, it updates some of the pairs still in the queue. If the head pair consists of s_1 and s_2 and the merge produces s_{12}, the algorithm must update any other pair involving either s_1 or s_2; such a pair must now include s_{12}, and its queue element must be keyed by the cost of merging with s_{12}. Each sphere stores a



(A) (B)

Figure 6.25: Two cases of the distance from a forming point to a sphere.

list of all queue elements for pairs containing it, so the algorithm can quickly find these pairs. Each queue element stores a pointer to its position in the queue, so the algorithm can quickly initiate the process of putting an updated pair into its new queue position. Sedgewick [Sed88] describes efficient algorithms for putting the pairs into their new positions.

The queue is keyed on the cost of merging two spheres, s_1 and s_2. A low cost means that s_{12}, the merger of s_1 and s_2, makes the sphere-tree nearly as accurate as if s_1 and s_2 were left unmerged. There are many ways that the algorithm could measure cost. One approach would be to measure the "looseness" of s_{12} around the object's surface, as described in Section 5.3.2. Instead, the merging algorithm uses a simpler and more efficient measurement. Recall that each sphere bounds a set of forming points on the object's surface. The merging algorithm uses these points as an approximation to the object's surface when computing the cost. More specifically, it measures cost as the approximate "looseness" of the sphere around these points. For each point, it computes the distance from that point to the sphere. Figure 6.25 shows how the algorithm computes the distance for a point at position $\mathbf{p}$. The algorithm first projects the sphere's center, $\mathbf{c}$, onto the plane of the object's surface at $\mathbf{p}$. It projects along the surface normal at $\mathbf{p}$, or if $\mathbf{p}$ is on an edge or vertex, along the average normal. Let $\mathbf{c}'$ be the projection of $\mathbf{c}$, d_1 be the distance from $\mathbf{c}$ to $\mathbf{c}'$, d_2 be the distance from $\mathbf{c}'$ to $\mathbf{p}$, and r be the radius of the sphere. The distance from $\mathbf{p}$ to the sphere is

$$\sqrt{r^2 - d_2^2} + d_1$$

if $\mathbf{c}$ is in front of the object's surface at $\mathbf{p}$, as in Figure 6.25(A), or

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

if $\mathbf{c}$ is behind it, as in Figure 6.25(B).

Given this distance for each forming point in the set, the algorithm uses the maximum distance as an approximation to the distance from the sphere to the object in the neighborhood of the points. Using this approximation has two main advantages. First, it is simple to implement. Second, it is efficient; computing it takes significantly less time than applying the more accurate techniques of Section 5.3.2. This approximation is only a heuristic, but it is interesting to note that the approximation can equal the true distance in some situations. Recall from Section 5.3.2 that the true distance is the *maximum* over all positions on the sphere of the *minimum* distance from the position to the object. Let $\mathbf{q}$ be a position on the sphere whose minimum distance is maximal. If one of the forming points happens to be the point on the object closest to $\mathbf{q}$, then the approximate distance will equal the true distance. The chance that this situation will occur is not large in practice, but the approximate distance is accurate enough to be useful as a cost function.

To measure the cost of merging spheres s_1 and s_2, the algorithm must build the sphere, s_{12}, that would be the result of the merger. Building s_{12} involves finding the union of s_1 and s_2's forming points, and the bounding sphere for this union of points. These operations are not extremely expensive, but there is no reason for the algorithm to repeat them more than necessary. In particular, the algorithm should not perform them once to compute the cost of the merger, and then again if the cost is low enough that the algorithm chooses to make the merger. To avoid this unnecessary work, the algorithm caches the sphere s_{12} with the pair s_1 and s_2 when it computes their cost. Note that the algorithm only computes this cost once, when adding s_1 and s_2 to the priority queue, so it stores the cache in their queue element.

6.3.2 Ensuring Conservative Covering

The merging algorithm from the previous section builds spheres that cover the forming points of the Voronoi vertices. Since the technique from Section 6.2.2 places these points uniformly across the object's surface, the merged spheres approximately cover that surface as well. There can be uncovered spots between the points, however. For the sphere-tree to be part of a conservative narrow phase, the merging algorithm needs a postprocess that makes certain these spots are covered.

There are two reasons for correcting the uncovered spots in a postprocess, rather than ensuring that they never occur at all. First, the merging algorithm is simpler and more efficient if it can process points on the object's surface instead of the surface itself. Second, the bounding spheres produced by the merging algorithm tend to have a little "slack," so uncovered spots are rather rare.

The first step in the postprocess detects the uncovered spots. Recall Section 5.3.3 and its approach to checking for a conservative covering. This approach uses the boundary lemma to detect any triangles on the object's surface that are not fully covered. The postprocess can use the same approach. The only difference is that the postprocess needs, for each uncovered triangle, a list of the spheres corresponding to uncovered boundaries.

The postprocess uses this list in its second step. This step slightly grows the spheres in each list, and rechecks the corresponding triangle for coverage. If this triangle is still uncovered, the postprocess grows the spheres again. It continues until the triangle is fully covered. When all the triangles are covered, the postprocess is complete. The amount each sphere grows at each iteration is a user-specified parameter. I have found that ten percent growth (i.e., scaling the radius by the factor 1.1) works well in practice, allowing the postprocess to complete in one or two iterations in most cases.

6.3.3 Eliminating Redundant Spheres

Section 5.3.4 describes how a sphere-tree built by simulated annealing can have redundant spheres, that is, spheres that cover only parts of the object that other spheres already cover. That section presents an algorithm that identifies redundant spheres and removes them from the sphere-tree, preventing them from becoming the roots of unnecessary subtrees. All the ideas from that section apply to sphere-trees built from medial-axis surfaces as well. Redundant spheres are less likely in this context, however. The merging algorithm from Section 6.3.1 assigns spheres to cover specific parts of the object's surface, that is, specific sets of forming points for Voronoi vertices. Voronoi vertices do share forming points, so the spheres do overlap to a certain degree. Enough overlap to make a sphere redundant is unlikely, though, because the process of merging tends to make one sphere cover the majority of points in a given area. Thus, I did not actually add the technique from Section 5.3.4 to the algorithm that builds sphere-trees from medial-axis surfaces.

6.3.4 Accuracy

Section 5.3.5 discusses the idea of computing the accuracy of a sphere-tree built by a simulated-annealing algorithm. Once it has finished positioning each sphere in the sphere-tree, the algorithm measures the final "looseness" of the sphere by the techniques of Section 5.3.2 and stores this measurement with the sphere. An application that uses this sphere-tree can thus compute the accuracy with which sphere-trees approximate collisions. The algorithm that builds sphere-trees from medial-axis surfaces can use exactly the same approach to compute accuracy. Note that this algorithm thus uses two different measurements of "looseness." When building the sphere-tree by merging spheres, the algorithm use the heuristic from Section 6.3.1, which quickly approximates "looseness" for the object from points on the object. Once the algorithm has finished building the sphere-tree, it goes back and recomputes "looseness" more accurately with the techniques from Section 5.3.2. This final step is time consuming for a deep sphere-tree, but it is worth taking this time once when the sphere-tree is built so the accuracy information will be available each time an application uses the sphere-tree. The next section discusses the accuracy of sphere-trees for several sample objects.

6.3.5 Results

The pseudocode in Figure 6.26 summarizes the algorithm that builds sphere-trees from medial-axis surfaces. Figures 6.27 through 6.29 show the results of the algorithm on three sample models. Figures 6.30 through 6.33 present some statistics for the algorithm's performance when generating these sphere-trees on a Hewlett Packard 9000 Model 735.

The medial-axis-surface algorithm produces more accurate sphere-trees than the octree algorithm, from Section 5.2. Figure 6.30 compares the inaccuracy each algorithm produces at corresponding levels of the sphere-trees. As in Section 5.3.5, the accuracy of each sphere is its distance to the object, and both the average and maximum inaccuracy over the spheres at each level give some indication of the sphere-tree's fit. Figure 6.30 shows that each level of the medial-axis-surface sphere-trees improve on the same level of the octree sphere-trees by both statistics, although the improvements in average inaccuracy are more significant. The average inaccuracy approximates how much error the sphere-tree is likely to give for a random collision. By this statistic, sphere-trees generated from medial-axis surfaces lead to more accurate collision detection than sphere-trees generated from octrees. This increased accuracy comes at the cost of increased preprocessing time (to build the sphere-trees) but not increased detection time while the application is running. In practice, the narrow phase is likely to do less work when using medial-axis-surface sphere-trees. The savings come from objects that do not actually collide but whose bounding spheres intersect. The upper levels of sphere-trees built from medial-axis surfaces fit these objects more tightly, so the narrow phase will descend fewer levels before discovering there is no collision.

The medial-axis-surface algorithm compares favorably with the simulated-annealing algorithm, from Section 5.3. The first advantage is reduced processing time. Figure 6.31 breaks down the time involved in building the medial-axis surfaces. The algorithm spends a significant portion of that time detecting gap-crossing cells and adding points to eliminate them. Figure 6.32 compares the number

```
sphere_tree(O, d, n)
{
    /* See Section 6.2. */
    M ← medial-axis surface for O
    V ← Voronoi vertices from M

    root is bounding sphere for O
    for j ← 0 to d {

        /* See Section 6.3.1. */
        for s ← each leaf at level j {
            S ← spheres centered at v ∈ V inside s
            S′ ← pairs of adjacent spheres from S
            Q ← priority queue for S′, keyed on merge cost
            while (|S| > n) {
                    (s1, s2) ← head of Q
                    s12 ← merger of (s1, s2)
                    update Q and S so s12 replaces s1 and s2
            }
            children of s are spheres from S
        }

        /* See Section 6.3.2. */
        do {
             if (any faces of O's surface are uncovered)
                grow spheres touching those faces
        } while (any faces remain uncovered)

        /* See Section 6.3.3. */
        eliminate redundant spheres at level j
    }
}
```

Figure 6.26: This algorithm builds a sphere-tree from object O's medial-axis surface. The sphere-tree has depth d and fanout (number of children at each node) n.

of added points to the number of original points. Once it has vertices on the medial-axis surface, the algorithm proceeds to build the sphere-tree. Figure 6.33 summarizes the time involved in the major steps of this process. The total time spent by the algorithm is not unreasonable, ranging from 12 to 21 minutes to build three-level sphere-trees for the various models. This processing time is significantly less than what simulated annealing requires. In particular, the medial-axis-surface algorithm builds a three-level sphere-tree for the lamp model, Figure 6.28, in less than half the time simulated annealing requires to build a two-level sphere-tree.

Building sphere-trees from medial-axis surfaces also seems more predictable and reliable than using simulated annealing. The sphere-trees in Figures 6.27 through 6.29 show none of the oddly-placed or oddly-sized spheres visible in the results of simulated annealing, Figures 5.22 through 5.25. Simulated annealing is strongly affected by the seed of the random-number generator, but it does not have much affect on medial-axis surfaces. Only the algorithm that places points on the object's

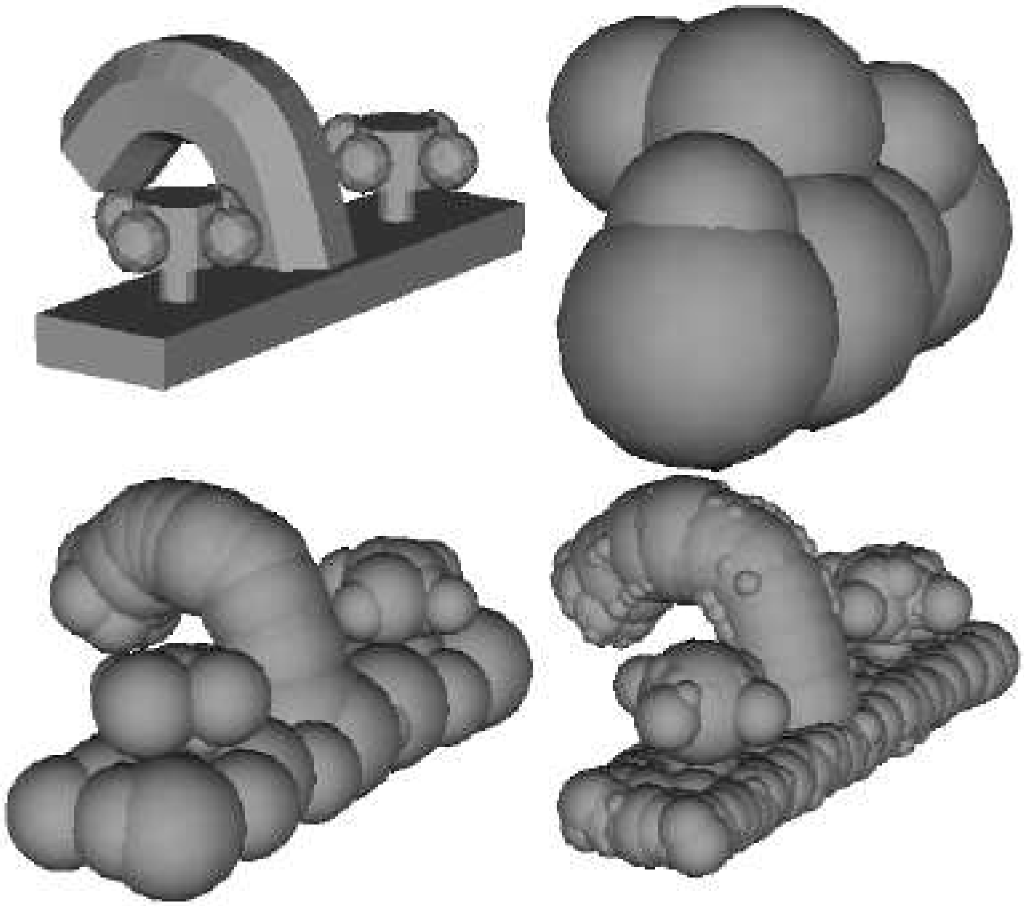


Figure 6.27: A bathroom faucet (1288 triangles), and three levels of its sphere-tree, generated from the medial-axis surface.

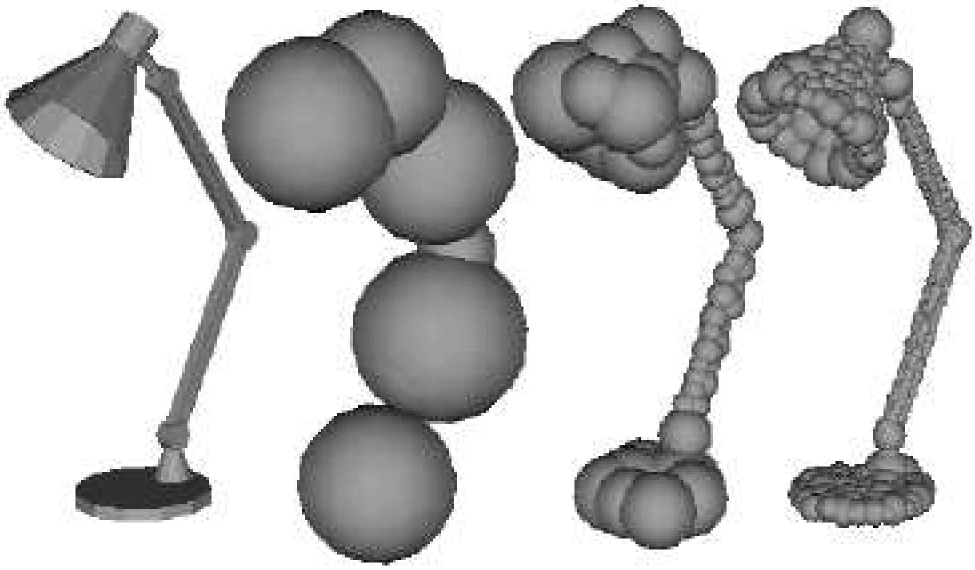


Figure 6.28: A desk lamp (626 triangles), and three levels of its sphere-tree, generated from the medial-axis surface.

surface uses randomness. It determines the initial positions of the points randomly, and then uses relaxation to spread the points more evenly. Most effects of the initial randomness disappear after the relaxation iterates a sufficient number of times, 40 in my tests.

The only other work on fitting spheres to 3D objects, other than simple approaches comparable to the octree approach from Section 5.2, is the work of Badler, O'Rourke and Toltzis [BOT79, OB79]. Their algorithm builds a set of spheres that fit just inside an object's surface. The spheres are anchored to points on the object's surface. O'Rourke and Badler [OB79] describe how to create the set of points by interpolating between cross sections of the object, but any other way of producing points, such as the technique from Section 6.2.2, would also be appropriate. Given a set of points, the algorithm repeatedly chooses a point, p, constrains a big sphere to touch p and then iteratively shrinks the sphere. At each iteration, the algorithm checks for other surface points that are inside the sphere; when it finds one, it shrinks the sphere so that it is just inside that point (and still attached to p). The iteration terminates when the sphere bounds none of the surface points. Since the points approximate the object's surface, the resulting sphere is roughly the largest sphere touching p that fits inside the object.

This approach seems like the basis for an algorithm that builds sphere-trees, but completing that algorithm would require a significant number of extensions. First, the algorithm would need a way to transform spheres that fit inside an object into spheres that conservatively cover the object. An appropriate approach is the technique from Section 6.3.2. Next, the algorithm would need a way to generate different levels of the sphere-tree, that is, sets containing various numbers of spheres that approximate the object's shape. Generating small and medium-sized sets is the most challenging problem. In this case, the algorithm must still use a large number of points on the object's surface; these points tell the algorithm how the sphere fits the surface, and an insufficient number of points can lead to large uncovered regions of the surface. The algorithm could anchor spheres to only a subset of these points, but choosing an appropriate subset is a difficult problem. The only alternative seems to be generating a large number of spheres and merging them to make fewer spheres, as in

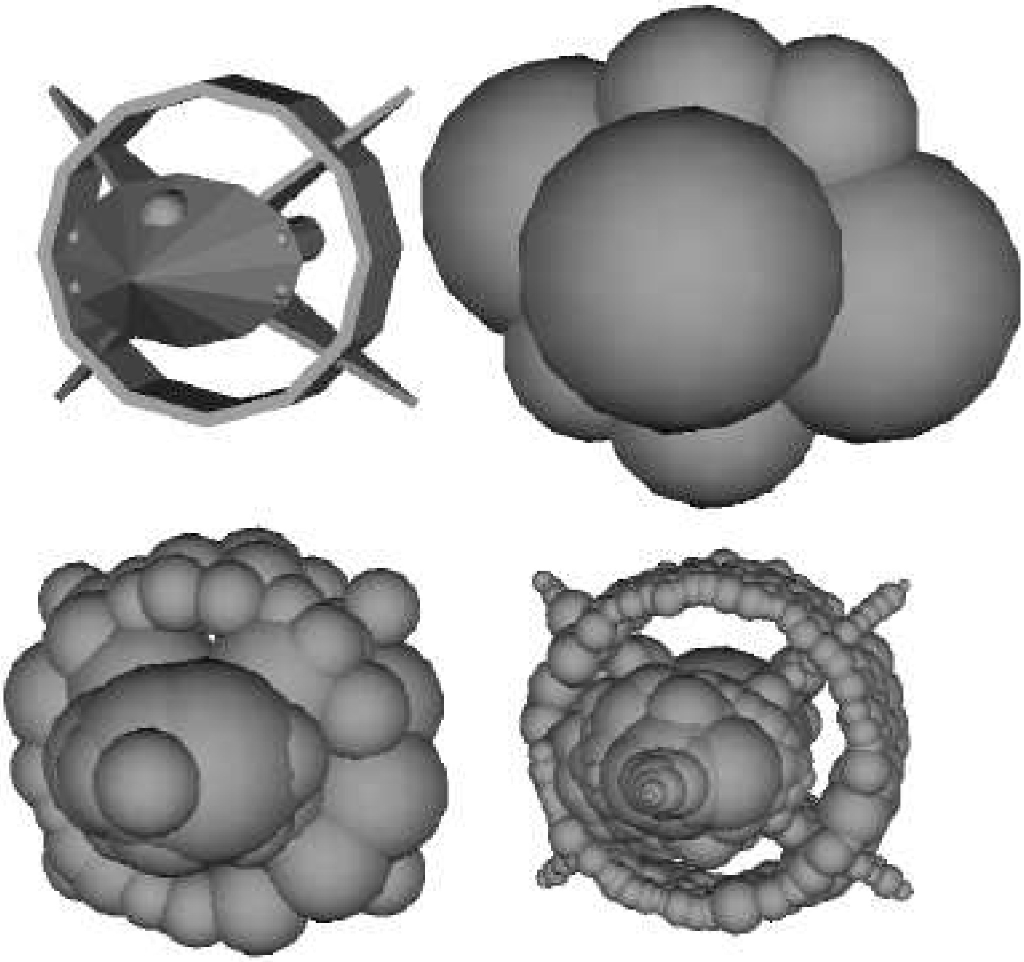


Figure 6.29: A spaceship (1420 triangles), and three levels of its sphere-tree, generated from the medial-axis surface. See Figure 6.10 for a view of the ship from a different angle.

Section 6.3.1. The technique from Section 6.3.1 relies on adjacency information, however, and the O'Rourke-Badler algorithm does not maintain such information. To obtain adjacency information efficiently, the algorithm would have to build a structure that represents proximity relationships, and one of the best choices would be a Voronoi diagram. Even with a Voronoi diagram, the algorithm would not be able to detect spheres which are adjacent across a "gap" outside the object. To avoid the errors created by merging such spheres the algorithm would need some way to perform the analysis from Section 6.2.4. Finally, the algorithm would need to measure the accuracy of its results, using ideas like those in Section 6.3.4. Thus, getting the O'Rourke-Badler algorithm to actually generate sphere-trees would require adding essentially all the techniques from this chapter. The algorithm from this chapter therefore seems like the most appropriate approach currently available for generating sphere-trees.

model	level	fraction of octree's average inaccuracy	maximum inaccuracy
faucet	1	0.492	0.552
	2	0.358	0.439
	3	0.377	0.687
lamp	1	0.442	0.494
	2	0.220	0.485
	3	0.164	0.433
ship	1	0.654	0.728
	2	0.546	0.690
	3	0.379	0.979

Figure 6.30: Accuracy of medial-axis sphere-trees compared to octree sphere-trees.

model	processing time (mins) distributing points	building Voronoi diagram	eliminating gap-crossing cells	finding interior vertices	total
faucet	0.6	2.6	1.9	0.9	6.0
lamp	0.6	3.9	4.0	0.7	9.2
ship	0.5	8.1	3.3	1.0	12.9

Figure 6.31: Timing results for building medial-axis surfaces.

model	number of surface points original	added (gap-crossing)
faucet	3285	12
lamp	3482	808
ship	5181	163

Figure 6.32: A count of the points involved in the medial-axis surfaces.

model	processing time (mins) merging spheres	making conservative	measuring accuracy	total
faucet	5.1	0.1	3.0	8.2
lamp	2.0	0.1	1.1	3.2
ship	5.5	0.1	3.3	8.9

Figure 6.33: Timing results for building sphere-trees from medial-axis surfaces.

Chapter 7

Performance

To be truly valuable, space-time bounds and sphere-trees must improve the performance of collision detection for interactive applications. This chapter argues that the two techniques do meet this goal. The supporting evidence comes from empirical studies. Another valuable form of evidence would be a predictive model of the techniques' performance over a wide variety of situations, and an analysis of this model that proves conclusively the advantages of the techniques. Unfortunately, collision detection in real applications does not lend itself to such models or analysis. Finding common features of objects' shapes and behaviors across different applications is very difficult. Even within a single application, the patterns of possible collisions vary enough to make analysis a daunting prospect. A worst-case analysis is more feasible, but it is likely to be artificially pessimistic. Figure 7.1, for instance, shows a set of N objects that touch each other $N^2/4 = \mathrm{O}(N^2)$ times. Such an example proves that the worst-case complexity of collision detection is $\Omega(N^2)$, but most realistic situations are not likely to be this bad.

My empirical studies evaluate space-time bounds and sphere-trees together, as a complete detection algorithm, but also as independent techniques. These studies are valuable because a detection algorithm could use space-time bounds in its broad phase and a technique other than sphere-trees in its narrow phase, or vice versa. Tests of the techniques in isolation give some indication of when such a "mix and match" approach might prove beneficial.

All these tests are more meaningful when they measure performance relative to previous algorithms. One of the best algorithm for the broad phase is Turk's spatial hashing scheme [Tur90b]. Section 7.1 shows that space-time bounds perform better than Turk's algorithm in a large number of randomly-generated situations. For the narrow phase, an efficient approach is to use binary space partitioning (BSP) trees. Thibault and Naylor [TN87] demonstrated that BSP trees are an efficient way to find exact intersections between polyhedra. Section 7.2 shows that sphere-trees dramatically outperform BSP trees in situations drawn from a sample application. Section 7.3 uses this sample application as a test case for complete detection algorithms. Space-time bounds and sphere-trees working together allow this application to run at a high and nearly-constant frame rate, a rate which can be more than 200 times higher than what Turk's algorithm and BSP trees can provide. This real-time performance would not be possible without the interruptible structure of sphere-trees. Experience with this application suggests, however, that time-critical computing will be more successful when an application has support from more than just one time-critical algorithm.

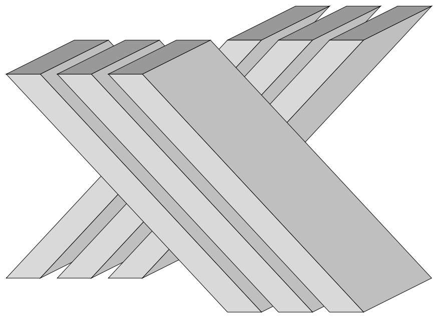


Figure 7.1: These $N = 6$ objects touch $N^2/4 = O(N^2)$ times.

7.1 Broad Phase

The broad phase detects collisions at the coarsest level of accuracy. It focuses more on the motion of objects over time than on the details of the objects' shapes. For this reason, the tests of the broad phase feature objects whose motions are more complicated than their shapes. The objects in the tests are isothetic (axis-aligned) cubes. Each cube's motion is determined by forces applied to its center of mass; there are no torques, so the cubes stay isothetic as they move. A test consists of a set of related "trials," each of which creates a population of cubes and lets them move until a pair of cubes collides; the result of a trial is a comparison of the time it takes space-time bounds and Turk's algorithm [Tur90b] to find this collision. As Section 2.3.2 explains, Turk's algorithm detects collisions between spheres; a simple modification allows it to detect collisions between isothetic cubes. Note that neither algorithm needs a narrow phase for objects as simple as isothetic cubes.

A performance comparison with Turk's algorithm is a challenge because Turk's algorithm suits interactive applications better than most previous algorithms. Recall from Section 2.3.2 that the algorithm uses a spatial hashing scheme to addresses the all-pairs weakness, which plagues many detection algorithms. The algorithm does not address the fixed-timestep weakness, but unlike most algorithms that do address that weakness, Turk's algorithm can handle objects whose future motion is not fully specified in advance. It can thus handle interactively-guided objects which they cannot handle. Turk's algorithm is also streamlined enough that it is not difficult to make it run efficiently in practice.

A test fixes certain parameters of the objects and their motion, and each trial within a test varies the other parameters randomly. The number of objects remains constant for all trials in a test. Each trial assigns initial positions to objects randomly, but the choice of positions is bounded by a sphere that remains fixed for all trials in a test. A test thus fixes the expected density of objects at their initial positions.

To move each object away from its initial position, a trial gives it a random initial velocity. The trial also picks a sequence of forces it applies to the object over the course of simulation time. Each force belongs to a subsequence of the overall sequence. The forces in a subsequence all have directions within one randomly-oriented hemisphere and magnitudes less than one randomly-chosen upper bound. A subsequence thus corresponds to one set of space-time bound parameters, $\mathbf{d}$ and M. At a particular moment in simulation time, the trial makes the M and $\mathbf{d}$ for the current subsequence (and their expiration time, the subsequence's duration) available to the algorithm that builds space-time bounds for the objects and finds their intersections. Each object has its own sequence of forces, but the subsequences all start and end at the same time across all objects. This constraint means the

M and $\mathbf{d}$ values are never unsynchronized, making space-time bounds more efficient as Section 4.4 explains. A real application would try to enforce the same constraint, so it seems reasonable to use it in these tests. Note that other than the M and $\mathbf{d}$ values for the current subsequence, the detection algorithm gets no information about the "future" of simulation time; each trial thus accurately reflects an important feature of interactive applications.

Although the forces within a subsequence are related by M and $\mathbf{d}$, these parameters vary randomly across the subsequences. Over many trials, this random variation approximates the "average" case, in a sense. Averages can be deceiving, though, so an interesting topic for future research would be tests in which the subsequences are related in various ways.

Once it has set the sequences of forces for each object, the trial proceeds as follows. It first measures the performance of the algorithm that uses space-time bounds. It calls this algorithm every $\Delta t_r = 0.02$ units of simulation time, as if it were generating frames of an animation with a rendering timestep Δt_r. As in the pseudocode of Figure 4.23, the algorithm uses a minimum temporal resolution of $\Delta t_d = \Delta t_r/2$ simulation units; collision detection thus has higher accuracy than rendering. Any time the algorithm asks an object for its position, the object computes its position from its forces with a numerical ordinary differential equation (ODE) solver, a fourth-order Runge Kutta method with adaptive stepsize [PTVF92]. To ensure that the adaptive stepsize does not become larger than it could in a real application, the trial inquires every object's position at each rendering timestep, as it would if it were actually rendering the objects. The trial ends when either the algorithm detects a collision or the current simulation time reaches an upper bound of 3. The trial then records the elapsed processing time, and repeats the process to time Turk's algorithm. During this second run, the trial calls Turk's algorithm every Δt_d—not Δt_r—units of simulation time, to ensure that it provides the same accuracy as space-time bounds. The performance of Turk's algorithm is likely to improve with the amount of memory it uses for its spatial hashing table. For fairness, Turk's algorithm uses as much memory as space-time bounds used during the first run.

The timings for the two runs measure the "overall" processing time, in the sense that they include the time spent by the ODE solver. It is also useful to know the amount of time that was devoted solely to detecting collisions. To this end, the trial also measures the ODE solver's processing time. Subtracting this time from the overall time gives the "detection-only" time. The trial uses the Unix "clock" routine for all timing measurements. Measuring the ODE solver's time requires two calls to "clock" for each inquiry of an object's position. Since "clock" does take some time itself, it is possible that these extra calls influence the detection-only time, but it is difficult to know for certain.

To compare space-time bounds and Turk's algorithm, I ran three tests. The first test involved 100 trials with 30 objects per trial, the second test 200 trials with 100 objects each, and the third test 100 trials with 300 objects each. Figures 7.2 through 7.4 show the results of these tests. These figures show the *speedup* of space-time bounds, which is defined as follows:

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

A speedup greater than 1 thus indicates that space-time bounds are faster. To avoid hiding "slowdowns" (speedups less than 1) the figures use a logarithmic scale. All tests ran on a Digital Equipment Corporation DECstation 5000/200 PGX Turbo.

In these tests, space-time bounds are almost always faster than Turk's algorithm. The reduced speedups in the overall graphs suggest that the overhead of the ODE solver is significant for both algorithms. Notice that the detection-only speedup tends to increase with the length of time before a collision occurs. This trend makes sense, because space-time bounds recognize situations in which collisions cannot soon happen, and avoid further processing until collisions are more imminent. Note

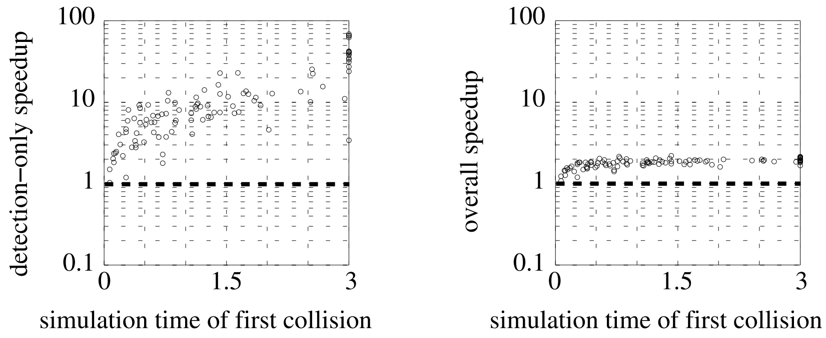


Figure 7.2: The speedup of space-time bounds over Turk's algorithm, for 100 trials with 30 objects per trial. Each circle represents one trial.

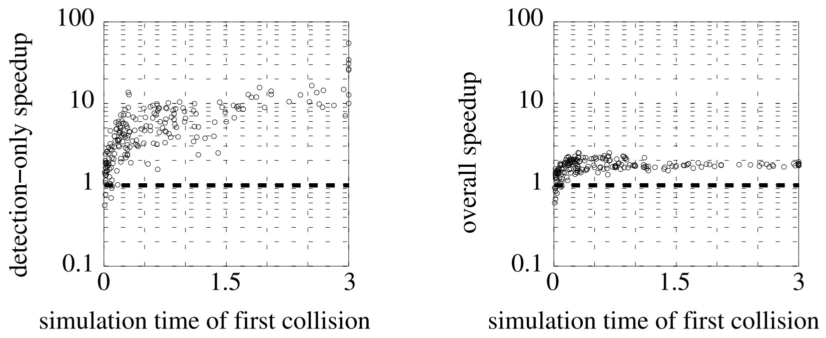


Figure 7.3: The speedup of space-time bounds over Turk's algorithm, for 200 trials with 100 objects per trial. Each circle represents one trial.

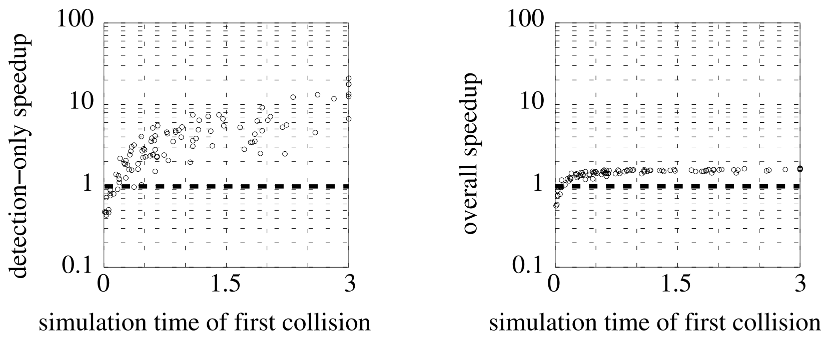


Figure 7.4: The speedup of space-time bounds over Turk's algorithm, for 100 trials with 300 objects per trial. Each circle represents one trial.

also that the detection-only speedup is not as large for the tests involving 300 agents as for the tests involving 30 agents. This difference could be evidence that space-time bounds do not scale, but it is more likely due to the difference in object density between the tests. The initial position of each object lies within a sphere whose radius is fixed for all trials in a test. The relationship of the spheres for the 30-object and 300-object tests was such that the average number of objects per unit volume was 42 times higher for the 30-object test. Turk's algorithm tends to perform better when the objects are less dense, because the expected number of objects per hash cell decreases with density. Thus, the 300-object test favored Turk's algorithm more than the 30-object test.

The speedups from Figures 7.2 through 7.4 reflect performance over a series of frames. For interactive applications, it is also important that each individual frame be fast. To study the per-frame performance of space-time bounds, I repeated the three tests with a different timing scheme. This scheme records the maximum time spent at any frame over the course of a trial. Figures 7.5 through 7.7 show the speedup of space-time bounds for this timing scheme. These graphs indicate that in most trials, the worst-case per-frame performance of Turk's algorithm was better than space-time bounds.

Further tests indicate, however, that frames nearly as slow as the slowest frame are less common with space-time bounds. Figures 7.8 through 7.10 give the results of these tests. These figures show histograms of the time each algorithm spends at each frame, with a histogram for each algorithm on each test. These histograms show that space-time bounds produces many more fast frames than Turk's algorithm, with the majority of space-time bounds' frames taking essentially no time. In the case of the 100-object test, the histogram shows that the worst of space-time bounds' worst cases is almost twice as fast as the worst of Turk's worst cases. This pattern is reversed for the 300-object test (the histogram in Figure 7.10 for space-time bounds contains a few barely-perceptible non-empty bins between 0.3 and 0.4), but the density of objects in this test favors Turk's algorithm, as mentioned earlier. These histograms suggest that any slow frames space-time bounds produce are quite rare. Many fast frames and occasional slow frames meet the requirements of "soft" real-time systems like interactive graphics applications.

7.2 Narrow Phase

The key component of the narrow phase is a pair-processing algorithm. This algorithm checks a pair of objects for intersection as of a particular moment in simulation time. Since it need not deal with multiple objects or periods of time, a performance test for a pair-processing algorithm seems simpler than the broad-phase test from the previous section. It is important, however, not to make the test overly simplistic. For example, it might seem that the test could be as simple as follows: generate a random position and orientation for each of the two objects, time the pair-processing algorithm as it checks for intersection, and then repeat the process many times for a statistical summary. This approach can give misleading information, however. In a real application, the narrow phase does not process objects with random positions and orientations. Instead, it processes objects that are drawing close to each other or just barely intersecting. Objects that penetrate each other deeply can occur randomly but do not occur in practice, because collision response would stop the penetration before it become very deep. The depth of penetration can affect the performance of the pair-processing algorithm, so it is important to avoid these unrealistic situations.

The best way to avoid unrealistic situations is to measure the performance of the narrow phase in the context of a real application. For my test, I implemented a simple spaceship flight simulator. In this simulator, a user controls one spaceship interactively and the system controls a set of "drone"

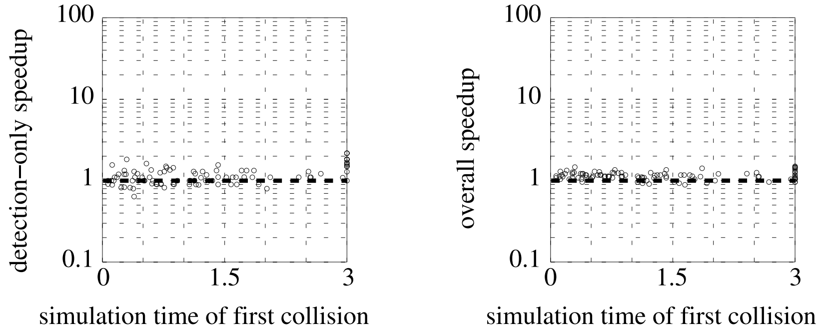


Figure 7.5: The speedup of space-time bounds' slowest frame over Turk's slowest frame, for 100 trials with 30 objects per trial. Each circle represents one trial.

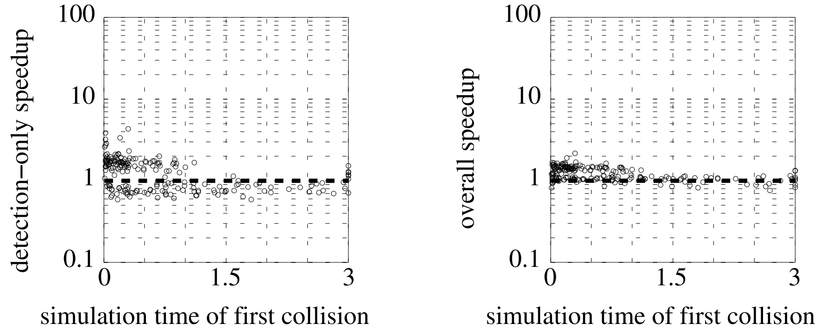


Figure 7.6: The speedup of space-time bounds' slowest frame over Turk's slowest frame, for 200 trials with 100 objects per trial. Each circle represents one trial.

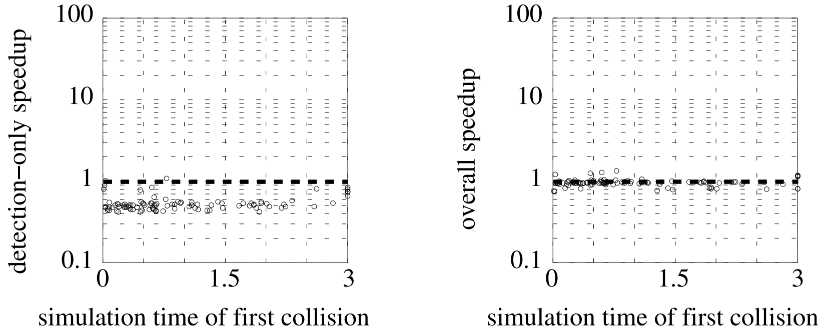


Figure 7.7: The speedup of space-time bounds' slowest frame over Turk's slowest frame, for 100 trials with 300 objects per trial. Each circle represents one trial.

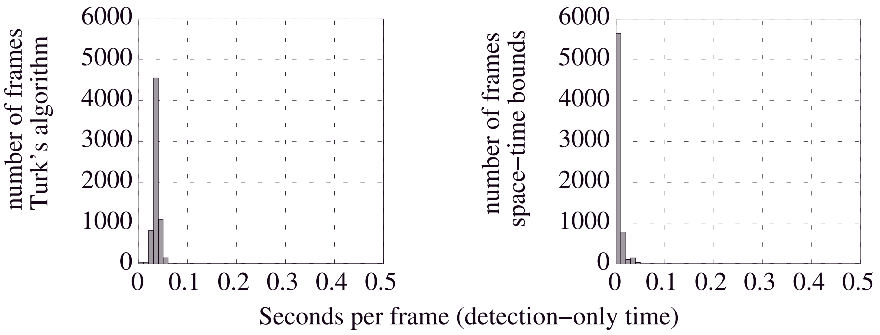


Figure 7.8: Histograms of frame times for Turk's algorithm and space-time bounds, for 100 trials with 30 objects per trial. Note that the leftmost bin for space-time bounds is very full.

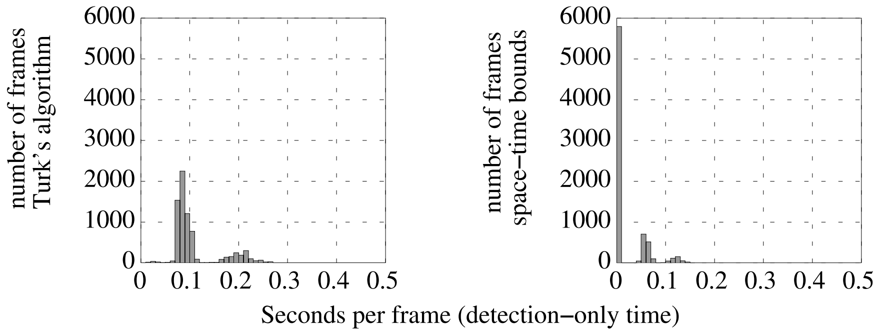


Figure 7.9: Histograms of frame times for Turk's algorithm and space-time bounds, for 200 trials with 100 objects per trial. Note that the leftmost bin for space-time bounds is very full.

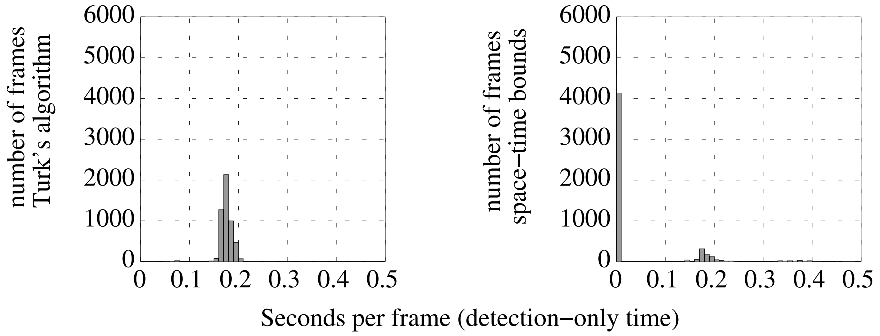


Figure 7.10: Histograms of frame times for Turk's algorithm and space-time bounds, for 100 trials with 300 objects per trial. Note that the leftmost bin for space-time bounds is very full.

ships. Section 4.4 explains the controls and simplified dynamics of the ships. For the drone ships, the system picks control values at random every few seconds. The collision response in this system is rudimentary. When a drone ship collides with anything it jumps back to its initial position and continues flying from there; the user-controlled ship is juggernaut that is not affected by any collisions. This system is not particularly easy to use, nor does it have any "purpose," but it suffices as a testbed for collision detection.

Under normal operation, this system uses sphere-trees for the pair-processing algorithm of its narrow phase. To compare sphere-trees against another pair-processing algorithm, the system performs the narrow phase twice at the same instant of simulation time, once for each algorithm. It measures the processing time for each algorithm and stores the results for later analysis. If the system instructs drone ships to stay in a small neighborhood of space, even a short run of the system generates a considerable amount of timing data.

As a comparison for sphere-trees, I implemented a pair-processing algorithm based on BSP trees. The algorithm is based on Thibault and Naylor's approach [TN87], as summarized in Section 2.3. Another interesting competitor would be an octree algorithm, as described by Moore and Wilhelms [MW88] or Shaffer and Herb [SH92]. Section 2.3 discusses the disadvantage of octree algorithms, that to retain their isothetic orientation they must be rebuilt for each new collision; this disadvantage probably makes octree algorithms slower than BSP-tree algorithms in practice, so I chose the latter.

The structure of a spaceship's BSP tree does not change during the course of the simulation, so the simulator uses BSP trees built in a preprocessing phase. To make the pair-processing algorithm as efficient as possible, the preprocessing phase uses a tree-building algorithm which optimizes the BSP trees. The goal of the optimizations is to minimize the number of BSP-tree nodes that the pair-processing algorithm must search. One heuristic is to make the tree as balanced as possible: when picking the partitioning face for a node, the tree-building algorithm chooses the face that most nearly balances the number of faces in front of it and behind it. This heuristic ignores the number of faces split by the partitioner, however. Split faces needlessly increase the overall size of the tree, increasing the work the pair-processing algorithm performs. A second heuristic is to pick the partitioning face which minimizes the number of split faces. In pilot studies, this heuristic seems to improve the pair-processing algorithm's performance more than the first heuristic. The BSP trees used by the simulator incorporate this second heuristic.

A comparison of BSP trees and sphere-trees is complicated by the differences in the results the two techniques return. BSP trees always detect polyhedral intersections with full accuracy. Sphere-trees, on the other hand, support detection at various levels of accuracy. The highest level is full accuracy; Section 5.1 explains how the leaf spheres of a sphere-tree can store pieces of the object's actual surface, thus permitting exact detection. For this level of accuracy, a direct comparison of the time spent by the two algorithms makes sense without question. Sphere-trees also detect intersections at lower levels of accuracy, however. In one sense, it is not fair to compare the time spent by the algorithms to obtain results of differing accuracy. From a broader perspective, however, such a comparison is reasonable. For an interactive graphics application, the time an algorithm spends computing its results is often as important as the accuracy of the results, so both factors are part of an overall sense of the result's correctness. A comparison of the performance of BSP trees with sphere-trees at all levels of accuracy indicates how much time an application can save by compromising on accuracy.

For the test this section describes, each "spaceship" in the simulator had the shape of the desk lamp, from Figure 6.28 of Section 6.3.5. I chose this whimsical shape over the spaceship model of Figure 6.29 for pragmatic reasons. The lamp model contains 626 triangular polygons, while the spaceship model contains 1420. Rendering the spaceship thus takes more processing time, making the effects of collision detection on performance less apparent. A pair-processing algorithm using BSP trees is also likely to do less work for fewer polygons, so the lamp model should make BSP trees more of a challenge for sphere-trees.

The test ran on a Sun Microsystems Sparcstation 10/512 ZX. To measure processing time, the system used the Unix "gethrtime" routine. This routine returns times in nanoseconds, but my empirical studies reveal that its minimum resolution is closer to 4.5 microseconds. The test involved one user-controlled lamp and 10 drone lamps. The sphere-tree for each lamp contained levels 1 through 3 depicted in Figure 6.28, plus an additional level 4. The spheres at level 4 also contain the triangles of the lamp's surface to support exact detection, which I call level 5.

During the course of the test run the narrow phase called each pair-processing algorithm 50042 times. Figures 7.11 through 7.15 show the *speedup* of sphere-trees over BSP trees for all these calls. Remember that speedup is defined as follows:

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

These figures show histograms of the speedups for each level of accuracy supported by the sphere-trees. More specifically, the histogram for level j counts the speedup for each occasion in which the sphere-tree algorithm reached level j. The algorithm may reach level j and stop there, because it is the first level at which there is no collision between the sphere-trees. On other occasions, the algorithm may reach level j and discover that the sphere-trees still intersect at that level; to the accuracy of level j the algorithm does not know if this situation is a real collision or not, so it is a "maybe" collision, using the terminology of Chapter 3. The histograms in Figures 7.11 through 7.15 do not differentiate between these occasions. The figure for each level shows three different views of that level's histogram. The leftmost view shows the entire histogram, with bins of width 100. The middle view shows in more detail the first bin, for speedups from 0 to 100; the bins in this histogram have width 5. The rightmost view shows the bin from 0 to 5 in even more detail. These detailed views are important because speedups less than 1 indicate that the BSP-tree algorithm is faster than sphere-trees.

These histograms show that sphere-trees are significantly faster than BSP trees in almost every situation. Only at level 5 (full accuracy) are BSP trees faster for more than 10 of the 50042 calls. Some of the situations in which BSP trees appear to be faster or nearly as fast for levels 2 through 4 may actually be due to timing anomalies. At about 0.3 percent of the calls, the timing data showed unusual jumps in the processing time between level 1 and level 2 of the sphere-trees. In particular, the processing time increased by a factor of 100 to 1000 even though all the algorithm did was test a few more pairs of spheres for intersection. It is possible that these jumps were the result of background processes running on the machine, or some other operating-system issue that is difficult to control. Similar anomalies might also have artificially increased the processing time for the BSP-tree algorithm in some cases. In future tests it would be valuable to eliminate these anomalies somehow. These problems are relatively rare in the current tests, though, and the overall trends of the data clearly indicate that sphere-trees are dramatically faster than BSP trees.

Remember that when the sphere-tree algorithm reaches level j, it sometimes discovers that there is no longer a collision. Figure 7.16 shows the speedup of sphere-trees in only these cases. This figure contains three views of the data, as in the previous figures. This figure does not differentiate the levels at which the sphere-tree algorithm stopped. In other words, this figure indicates the speedup obtained when the sphere-tree algorithm processes as many levels as it must to determine that there is not, in fact, any collision. Once again, this figure shows that sphere-trees are much faster than BSP trees.

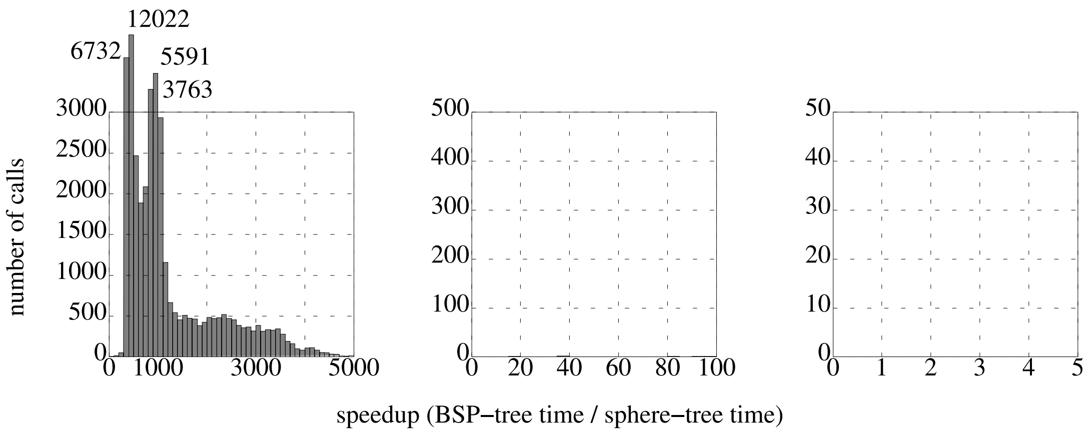


Figure 7.11: Speedups of sphere-trees over BSP trees for every narrow-phase call that reaches level 1.

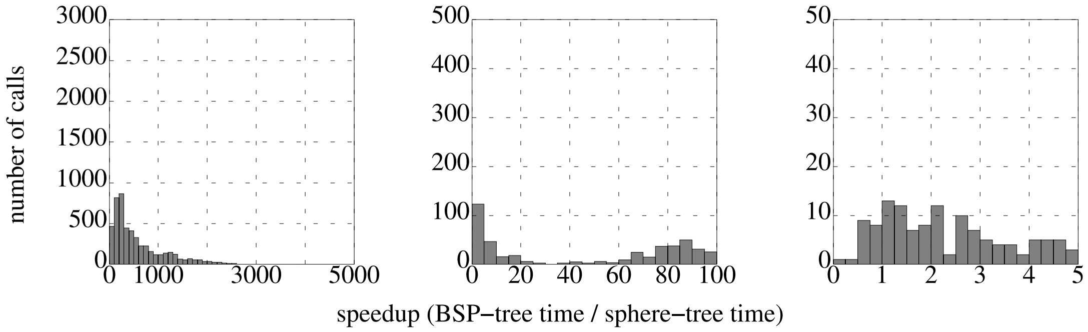


Figure 7.12: Speedups of sphere-trees over BSP trees for every narrow-phase call that reaches level 2.

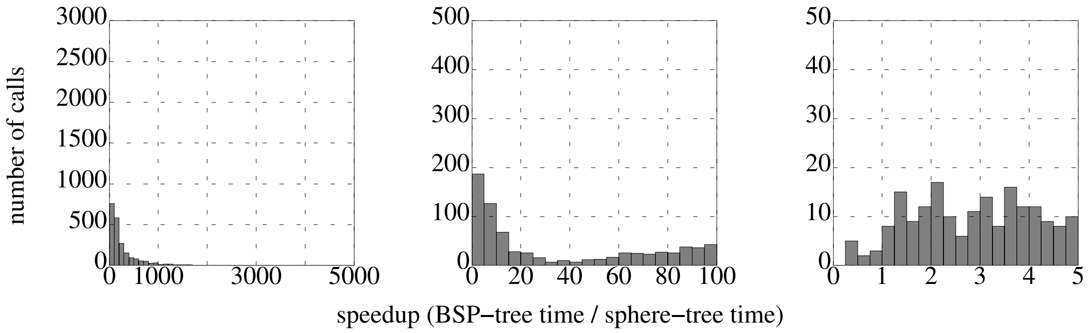


Figure 7.13: Speedups of sphere-trees over BSP trees for every narrow-phase call that reaches level 3.

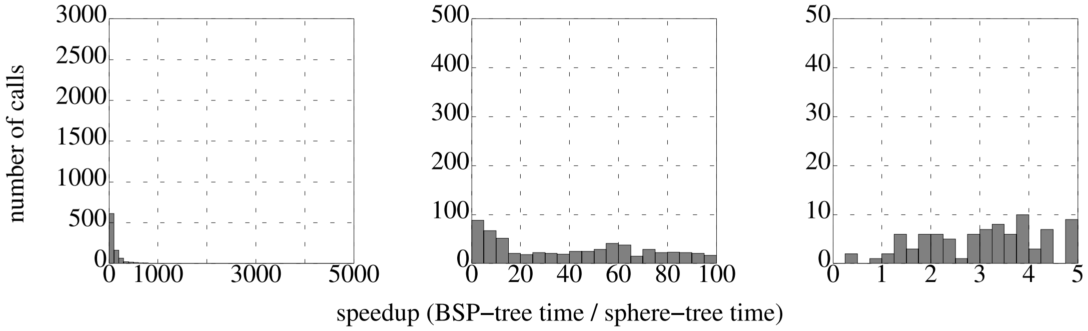


Figure 7.14: Speedups of sphere-trees over BSP trees for every narrow-phase call that reaches level 4.

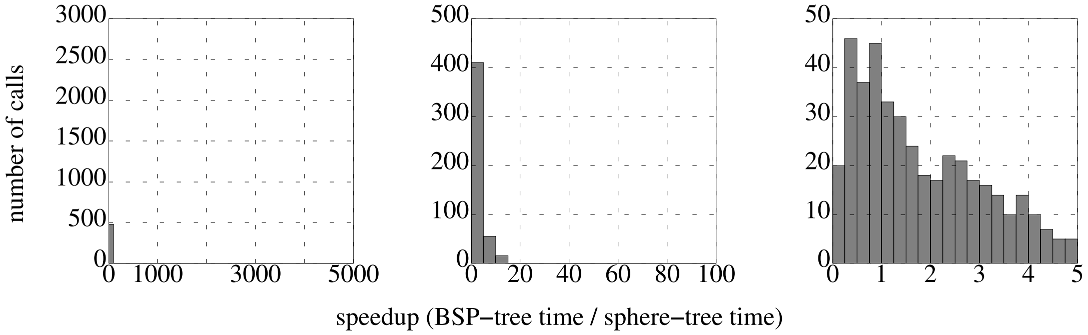


Figure 7.15: Speedups of sphere-trees over BSP trees for every narrow-phase call that reaches level 5 (exact detection).

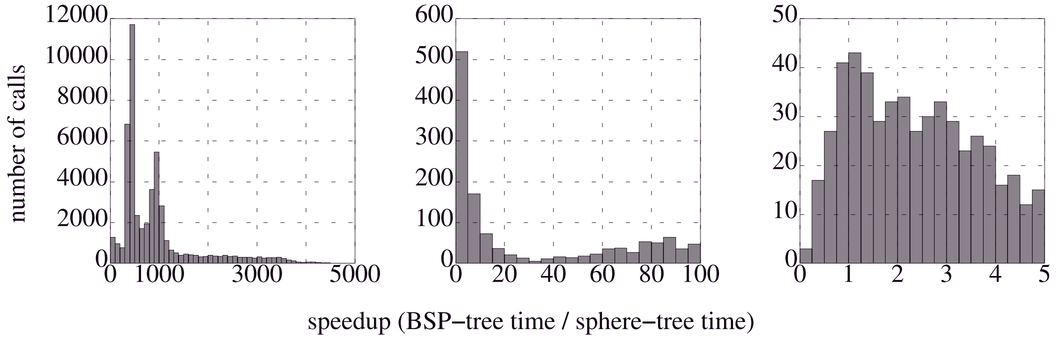


Figure 7.16: Speedups of sphere-trees over BSP trees for every narrow-phase call that found no collision as of some level.

If the sphere-tree algorithm reaches level j and discovers that some spheres still intersect, the application must decide if it should stop detection at that point and declare that there is an approximate collision. Figures 7.17 through 7.21 show speedups for only these "maybe" cases. More specifically, these histograms show the speedup the spaceship simulator could have obtained by stopping at each possible level. As in the previous analysis, sphere-trees provide dramatic speedups over BSP trees.

When the application decides to stop detection for a "maybe" collision at level j, the application accepts a certain amount of inaccuracy. Figure 7.22 shows histograms of the inaccuracy that occurred during the test run. The measure of inaccuracy is an upper bound on the separation distance between the pair of lamps, reported as a fraction of the radius of a lamp's bounding sphere. As Section 6.3.4 explains, each pair of intersecting level-j spheres provide an upper bound on the distance between the parts of the objects these spheres contain. Note that there can be more than one such intersecting pair. In this situation, the tightest upper bound on separation distance is the *minimum* of the estimates for each intersecting pair. The histograms in Figure 7.22 tabulate values of this minimum for each level. These histograms show that accuracy improved with depth in the sphere-tree. For the collisions that were still possible by level 4, the accuracy was quite high. Remember also that the values in these histograms are conservative, because the upper bound on separation distance can be loose; the true accuracy is likely to be significantly higher. These results suggest that an application does not sacrifice much accuracy when it accepts the speedup provided by sphere-trees.

7.3 Both Phases Together

The previous sections indicate that both space-time bounds and sphere-trees compare favorably to previous techniques when tested independently. This section discusses their performance when operating together. For testing the combined performance, I again used the spaceship simulator from the previous section. Figure 7.23 shows the time this simulator spends on collision detection when it uses Turk's algorithm [Tur90b] and BSP-trees [TN87] together. This figure traces the processing time for the detection algorithm through one run of the simulator. This run involved the user-controlled lamp and 10 drone lamps. Simulation time ran from 0 to 40, with the simulator rendering a frame every $\Delta t_r = 0.1$ simulation time units and calling Turk's algorithm every $\Delta t_d = 0.05$ units. Ideally, the simulator would run quickly enough that a unit of simulation time would correspond directly to a unit of real wallclock time. To meet this goal, the simulator must compute each frame in 0.1 seconds, but Figure 7.23 shows that the detection algorithm makes the simulator take substantially longer.

To test space-time bounds and sphere-trees in this context, I added one extension to the simulator. The extension is a simple policy for graceful degradation, one that tries to exploit the interruptibility of sphere-trees to make the simulator run at a high and nearly-constant frame rate. This extension does meet this goal in at least some simulator runs, but getting it to work was more difficult than I had expected.

Most of the difficulties arose because the application performs several activities at each frame, but it has access to a time-critical algorithm for only one of them, collision detection. The most significant of the other activities is rendering. Funkhouser and Séquin [FS93] describe a time-critical rendering algorithm, but I do not have an implementation of this algorithm. Even given an implementation, actually using the algorithm is complicated by the assumptions it makes. In particular, it assumes that the polygonal representation of each object is modeled at several levels of detail. Hoppe *et al.* [HDD+93] describe an algorithm that can automatically generate levels of

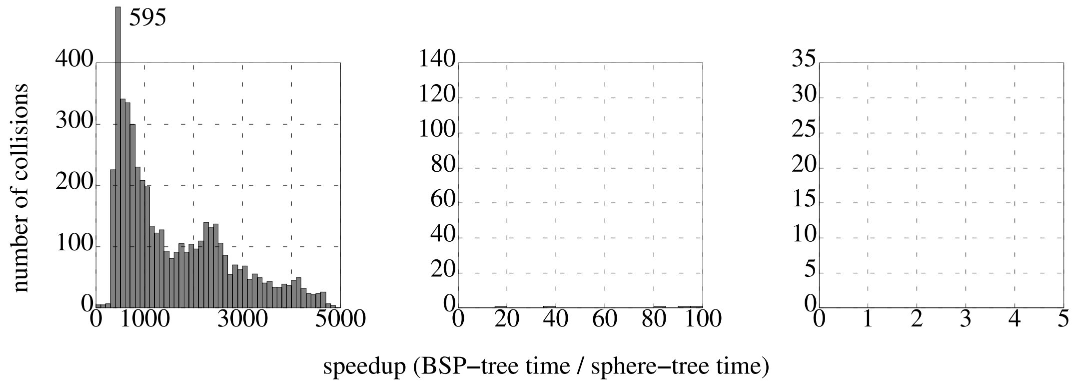


Figure 7.17: Speedups of sphere-trees over BSP trees for every narrow-phase call that stopped for a collision at level 1.

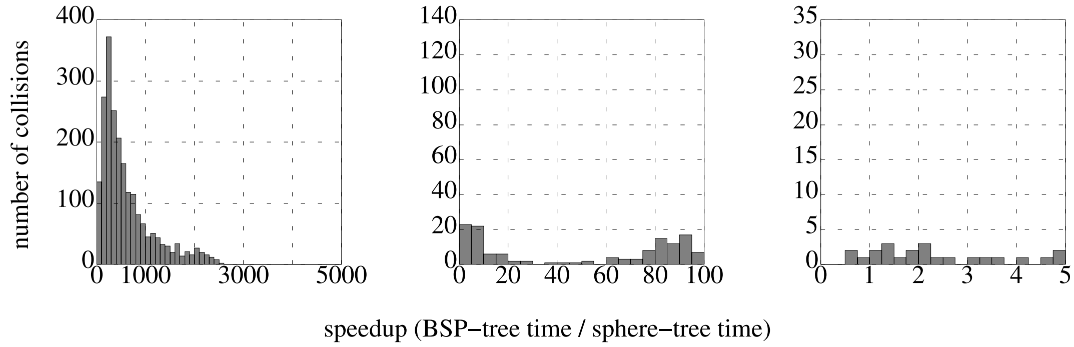


Figure 7.18: Speedups of sphere-trees over BSP trees for every narrow-phase call that stopped for a collision at level 2.

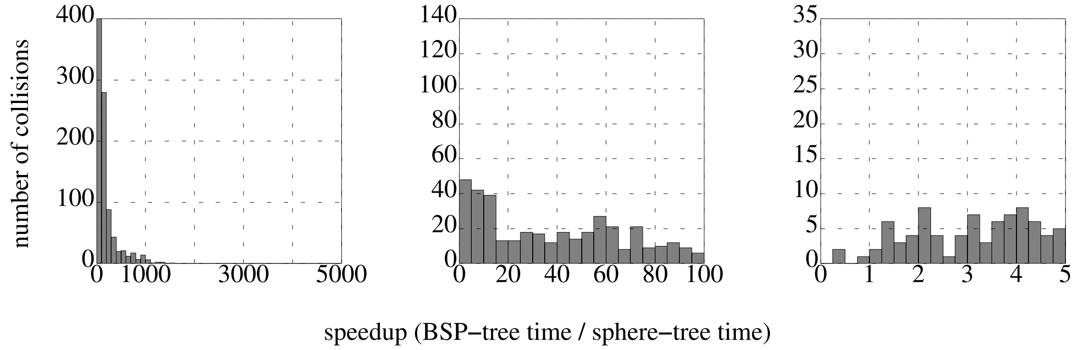


Figure 7.19: Speedups of sphere-trees over BSP trees for every narrow-phase call that stopped for a collision at level 3.

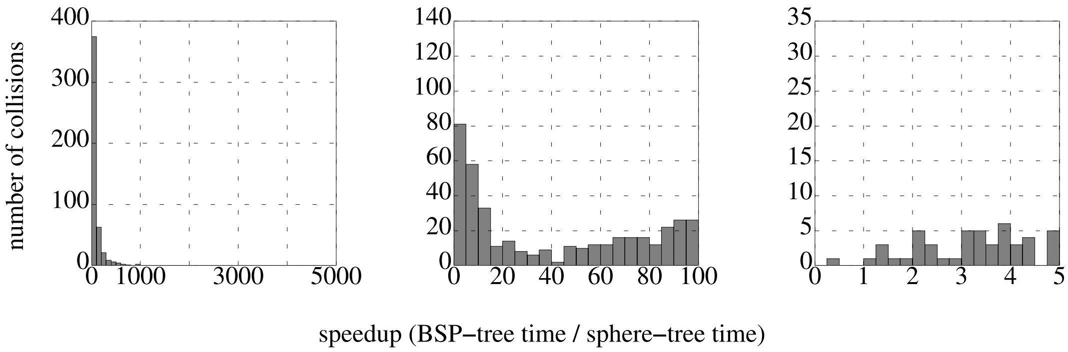


Figure 7.20: Speedups of sphere-trees over BSP trees for every narrow-phase call that stopped for a collision at level 4.

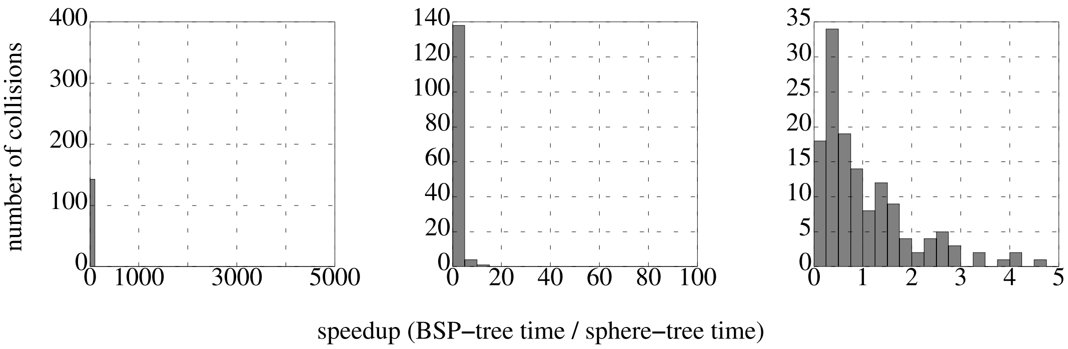


Figure 7.21: Speedups of sphere-trees over BSP trees for every narrow-phase call that stopped for a collision at level 5 (exact detection).

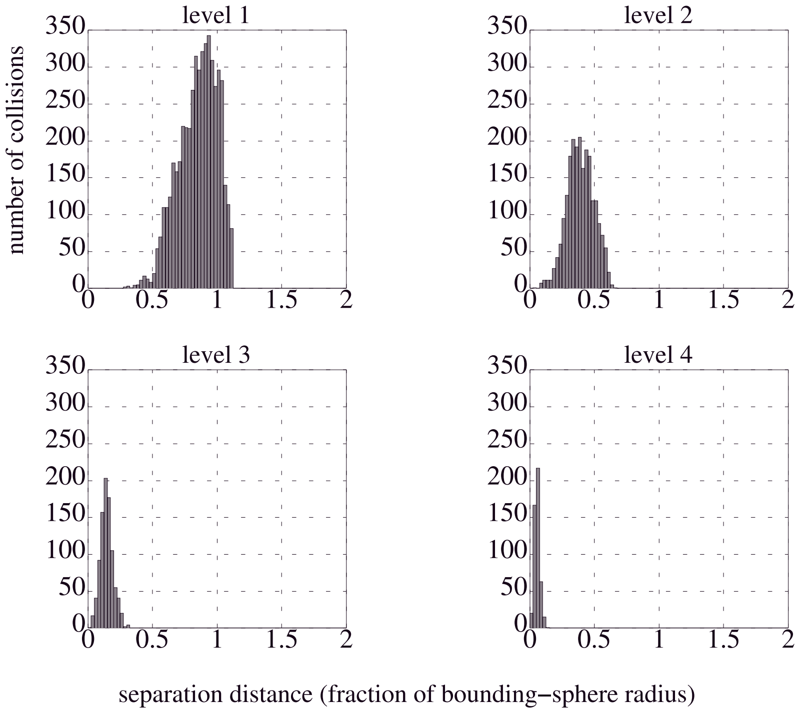


Figure 7.22: Accuracy of each narrow-phase call that stopped for a collision at some level of the sphere-trees.

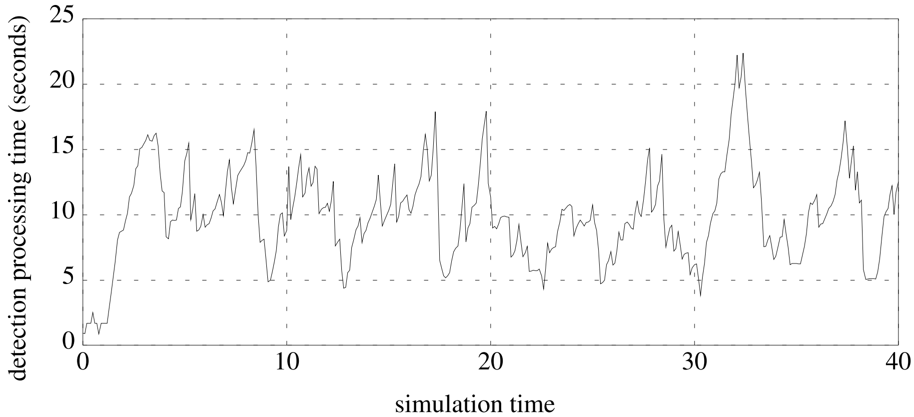


Figure 7.23: Processing times for Turk's algorithm and BSP trees at each frame of a simulator run.

detail for some objects, but implementing this algorithm would require considerable effort. The alternative, generating levels of detail by hand, would also be a laborious process.

As a simpler solution, the simulator just uses the workstation's standard rendering hardware, which makes no attempt to be time-critical. There is no doubt that this hardware takes different amounts of time for scenes of different complexity. The spaceship simulator, though, renders the same set of objects at every frame, so it is possible that the rendering time might not vary much between frames. The simulator assumes there is little variation, and it predicts that the rendering time at the current frame will be the same as for the previous frame. This prediction is important, because it tells the simulator how much time it can spend on collision detection, which precedes rendering in the computation of a frame.

This form of prediction assumes that the simulator can actually measure the rendering time from the previous frame. The simulator measures times with the Unix "gethrtime" routine, but this routine takes some time itself. To minimize measurement overhead, the simulator makes as few measurements as possible. In particular, it measures only the time it spends on collision detection and the time it spends for the whole frame. The difference of these times is the *nondetection time* for the frame. The simulator predicts that the nondetection time of the current frame will be the same as for the previous frame, and uses this prediction to determine the time it can spend on collision detection. The detection algorithm performs the broad phase, and lets the simulator control the level of refinement in the narrow phase. The simulator measures the elapsed time, and stops refining when it exceeds the time allotted to collision detection. The broad phase might detect several pairs of colliding objects. To treat all the pairs fairly, the simulator refines these collisions in a round-robin fashion, refining each by one level before it refines any by another level.

It is possible that the simulator can finish all the activities for the current frame ahead of schedule. In this situation, the simulator enters a busy loop that repeatedly calls "gethrtime" until the time for the next frame arrives. The time spent in this busy loop is the *slack time* for the frame.

Figure 7.24 shows the performance of the simulator with its simple policy for degrading collision detection. The graph tracks the processing time for a run very similar to the run from Figure 7.23. This graph shows the overall time spent on each frame, and also breaks that time down into the detection time, the nondetection time and the slack time. The sphere-trees for this run have levels 0 through 4 like those from the previous section, but the simulator never refines to level 5 (full accuracy) during this run.

The performance goal for this run is making simulation time match wallclock time, so frames must take 0.1 seconds. Space-time bounds and sphere-trees do allow the simulator to meet this goal at almost every frame. Occasional frames do take longer, but they are rare and they miss the goal by only a small amount. The performance for this run is certainly superior to the performance of the other detection algorithm, in Figure 7.23. Note that at some frames, space-time bounds and sphere-trees are more than 200 times faster than the other algorithm.

Note that the nondetection time varies considerably over the course of the run. Its minimum value is roughly two-thirds of its maximum value, and it changes value at every frame. These fluctuations are due to the rendering hardware, and probably also to the Unix operating system, which provides no real-time guarantees. The detection algorithm must compensate for this variability to keep the frame rate constant. The detection algorithm might be able to perform better if it could concentrate more fully on keeping its own processing time stable. Note also that the slack time is nonzero for many frames. This observation indicates that the system is sacrificing throughput in order to maintain a nearly-constant frame rate. It would be interesting to study whether a time-critical rendering system and operating system would allow the simulator to reduce the slack time and thus increase throughput.

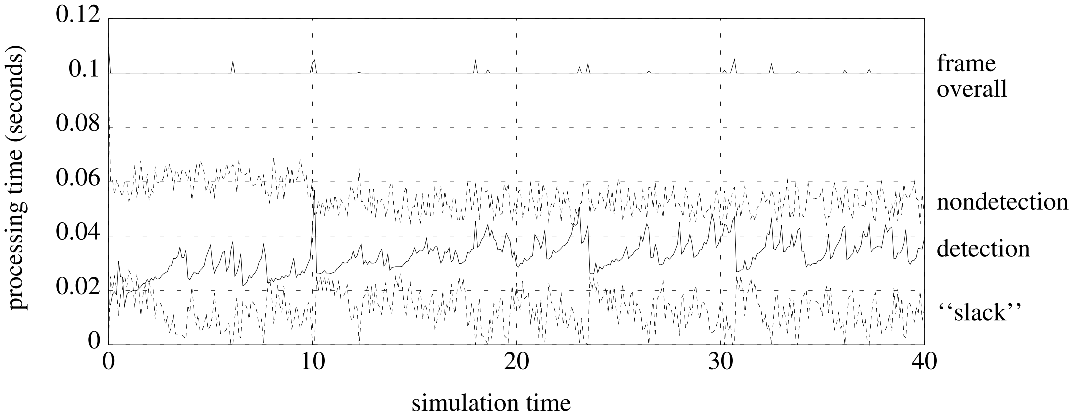


Figure 7.24: Overall times for space-time bounds and sphere-trees.

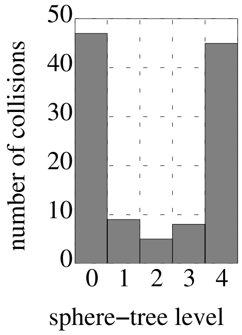


Figure 7.25: The number of times the application accepted a collision at each level of the sphere-trees.

Figure 7.25 indicates how well the simulator exploited the interruptible character of sphere-trees during the run. This histogram records all the situations in which the simulator stopped refinement when the sphere-trees still intersected, treating a "maybe" collision as a real collision; the bins count the number of times this situation occurred for each sphere-tree level. The simulator did use all the levels, but it relied most heavily on the lowest and highest levels. A more even distribution across the levels would indicate that the simulator was taking more advantage of graceful degradation. In some other runs, the simulator used the lowest and highest levels exclusively, suggesting that more levels of degradation do not always provide additional benefits. One possible explanation is that the variability in the nondetection time prevented graceful degradation from working effectively. Another possible explanation is that certain features of the spaceship simulator prevent it from fully exploiting graceful degradation. Recall that the simulator uses a very simple collision-response algorithm, which makes objects fly far apart as soon as they touch. This form of response reduces the overall number of collisions that the system must handle. With other forms of collision response, colliding objects collide repeatedly for several frames; consider, for example, objects which collide and roll off each other. This pattern of collisions would increase the amount of time spent in collision detection, and might well make the value of graceful degradation more apparent.

Another way to look at the situation is to notice that space-time bounds and sphere-trees are fast enough that collision detection is no longer the simulator's bottleneck. In Figure 7.24, the nondetection time is often more than twice the detection time. For different applications that involve more collisions, space-time bounds and sphere-trees at their deepest level will once again become a bottleneck. These applications will thus need to use more of the graceful degradation supported by sphere-trees to meet their performance goals.

Chapter 8

Conclusions

The popularity and success of computer graphics and animation sets high standards for the future of the technology. Users are no longer easily impressed, and they demand more realistic effects that they can control more naturally and directly. For many applications that depict moving objects, the absence of some form of collision detection and response will make users uncomfortable. For many applications that present information visually, the inability to manipulate that information interactively at real-time rates will limit what users can accomplish. This dissertation has explored some ways to address these problems. This section summarizes the contributions of this work, and discusses some of the many possible improvements and extensions.

8.1 Contributions

This dissertation presents two new techniques for collision detection, space-time bounds and sphere-trees. A key feature of these techniques is their support for approximation. Empirical evidence suggests that these techniques improve on the performance of some of the best previous algorithms; sphere-trees, in particular, offer dramatic speedups—often two or three orders of magnitude—to applications that can tolerate some loss of accuracy. A detection algorithm using these techniques is not only fast but also interruptible, using graceful degradation to implement an approach to time-critical computing. Tests with a sample application indicate that exploiting degradation is not always simple, but it can allow an application to meet real-time performance goals that would otherwise be impossible.

An important task is building sphere-trees automatically. As part of two solutions to this problem, this dissertation develops some interesting new geometric algorithms. One algorithm is a simple way to detect whether a union of two-dimensional shapes cover a polygon. A second algorithm improves on previous techniques for approximating medial-axis surfaces of three-dimensional (3D) polyhedra. A third algorithm addresses the important problem of building Voronoi diagrams from 3D data; empirical evidence indicates this algorithm is both more robust and more accurate than previous approaches.

In a broader context, this dissertation contributes more information about the general topic of time-critical computing. My experiences with graceful degradation in collision detection indicate that time-critical computing involves both problems and promise. Designers of time-critical algorithms and applications must confront some subtle issues. Most applications involve multiple tasks, and it can be difficult to get a time-critical algorithm for one task to compensate for "time-oblivious" algorithms for the other tasks. A single time-critical algorithm can allow an application to meet performance goals in some situations, though, and this positive result should provide encouragement for future research. Many applications could profit from interactive performance, and time-critical computing may be the best way to achieve that goal. Computer science as a whole will benefit when researchers extend preliminary results in time-critical computing to more general principles and guidelines.

8.2 Future Work

This dissertation leaves many topics underexplored. Chapter 4 describes space-time bounds as one way to avoid processing objects that cannot possibly collide, but variations on this approach are possible. Section 4.2 describes two ways to find intersections between space-time bounds, the projection method and the subdivision method. A hybrid approach drawing on both methods might be the most efficient algorithm. This hybrid would start with a few iterations of subdivision to group hypertrapezoid faces by their general location, and then it would apply the projection method to each of those groups. This approach might also work well on a multiprocessor system. The key to a multiprocessor implementation would be balancing the load between processors; the four-dimensional properties of space-time bounds might allow the algorithm to predict load in advance, allowing it to avoid unbalanced situations.

The current approach of approximating parabolic horns with hypertrapezoids and cutting planes is not the only possibility. A hypertrapezoid puts isothetic cubes around the $t = 0$ and $t = 1$ cross sections of a parabolic horn and interpolates linearly between these cubes. An alternative would be a structure that puts a cube around every cross section for $t \in [0, 1]$. This approach gives a tighter approximation to the original parabolic horn, thus reducing the number of spurious intersections. The "faces" of such a structure would no longer be planar in four dimensions. Their projections into the α-t plane would be quadratic curves, but a simple modification to the Bentley-Ottmann algorithm [BO79, PS85] allows it to find intersections between curves of this form. A new version of the projection method from Section 4.2.3 using this idea would do more work for each intersection, but the reduction in spurious intersections might improve the performance overall. Another alternative is to avoid polygonal approximation altogether, working instead with the parabolic horns directly.

At a basic level, space-time bounds could be based on maximum velocity or position instead of maximum acceleration. This reformulation might be advantageous in some cases, for applications that can estimate maximum velocity or position more easily than acceleration.

Sphere-trees also offer many opportunities for improvement. The algorithm from Chapter 6 to build sphere-trees from medial-axis surfaces performs well in its current form. An improved version could invoke a postprocess that tightens the spheres around the object. This postprocess could be similar to the simulated-annealing algorithm from Chapter 5, or it could be a random-descent algorithm, which shares many features with simulated annealing but never increases the "looseness" of the spheres.

This dissertation discusses only sphere-trees for rigid, polyhedral objects. Many of the techniques for building sphere-trees might extend to nonpolyhedral objects (e.g., objects with parametric bicubic surfaces [FvFH90]), although the algorithm for building medial-axis surfaces may not work for these objects. Flexible objects are more of a challenge. Building a sphere-tree for a flexible object could involve precomputing a sphere-tree for some "rest position" of the object and then perturbing the sphere-tree at runtime to follow the deformations of the object.

The current algorithms build sphere-trees that have the same depth everywhere. Some applications might need sphere-trees that are particularly deep (and thus accurate) in certain areas; the desk lamp of Figure 6.28, for example, has a flat base, and an application that needs this base to rest on a desk requires a particularly tight fit for the spheres on the base. It would be interesting to investigate the implications of different depths in the same sphere-tree.

An original goal of this work was a simpler approach to collision detection, one that would be easier to implement and less prone to runtime bugs. The approach in this dissertation is hardly simple. It is true that for sphere-trees, at least, the complicated aspects occur only in the preprocessing phase that builds the sphere-trees. Implementing this phase still takes considerable effort, though. A valuable contribution would be an approach to building sphere-trees which is simpler overall.

The detection algorithm that uses space-time bounds and sphere-trees is conservative. This feature is important for many applications. Some applications might tolerate less-conservative detection, however. In this case, the broad phase could become interruptible, a feature it does not currently support. In particular, the application could designate certain objects as less "important" than other objects. The broad phase could process space-time bounds for important objects first, getting back to less important objects only if time allows. With this organization the broad phase will miss some collisions involving less important objects, but the performance improvement that accompanies this decrease in accuracy may sometimes be valuable. In particular, an application might want the ability to interrupt after an arbitrarily small length of time; a less-conservative broad phase might allow the detection algorithm to come closer to meeting this goal.

Collision detection is far from being a solved problem. Space-time bounds and sphere-trees are an advance, but there are situations in which they are not the most efficient way to detect collisions. For example, to prevent a single moving object from passing through the walls of a maze, a more efficient approach is an algorithm that exploits the particular structure of the situation. It is unlikely that a single algorithm will ever be the most efficient approach for all applications. For this reason, a useful study would investigate the characteristics of different applications and algorithms to determine the best match in various situations.

Finally, one of the most important topics for future research is the practical implications of time-critical computing. It would be useful to have a better understanding of what applications can exploit graceful degradation, and what forms of degradation prove most effective. Implementations of fully time-critical systems, which use time-critical algorithms for all their components, would provide considerable insight. Controlled user studies are also essential, to determine what forms of inaccuracy are most acceptable to users, and to determine appropriate balances between speed and accuracy.

Appendix A

Proof of Lemma 5.1

This appendix proves Lemma 5.1 from Section 5.3.2. Throughout this appendix, let $d(\mathbf{x}, \mathbf{y})$ be the Euclidean distance between any points $\mathbf{x}, \mathbf{y} \in \mathbb{R}^3$. Recall that the lemma is as follows.

Lemma 5.1 *Consider a sphere, s, with center $\mathbf{c}$ and radius r that intersects the surface of a convex polyhedron, P, in $\mathbb{R}^3$. If $\mathbf{c}'$ is a point on the surface of P closest to $\mathbf{c}$, then the distance from s to P is $r + d(\mathbf{c}, \mathbf{c}')$ if $\mathbf{c}$ is outside P, and $r - d(\mathbf{c}, \mathbf{c}')$ if $\mathbf{c}$ is inside P.*

A.1 Preliminaries

To prove this lemma, several other lemmas are helpful. For any $\mathbf{x} \in \mathbb{R}^3$, let $d(\mathbf{x}, P)$ be the minimum distance from $\mathbf{x}$ to P. Define the distance function $D : \mathbb{R}^3 \to \mathbb{R}$ such that $D(\mathbf{x}) = d(\mathbf{x}, P)$. Note that in these definitions and elsewhere, P is treated as a solid, the union of the polyhedral boundary and the volume it encloses; any references to the surface of P will be explicit.

Lemma A.1 *For any $\mathbf{x} \in \mathbb{R}^3$ outside P there exists a unique $\mathbf{y} \in P$ such that $d(\mathbf{x}, \mathbf{y}) = D(\mathbf{x})$.*

Proof: The polyhedron P is closed and bounded, and $d(\mathbf{x}, \mathbf{p})$ is a continuous function of $\mathbf{p} \in P$. (The proof of the continuity would use the standard ϵ-δ argument; this argument is based on distance, so the proof is straightforward for a distance function.) Thus, there exists $\mathbf{y} \in P$ such that $d(\mathbf{x}, \mathbf{y})$ is minimal. Now, suppose that $\mathbf{y}$ were not unique, that is, suppose $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{x}, \mathbf{y}')$ for some other $\mathbf{y}' \in P$. The line segment between $\mathbf{y}$ and $\mathbf{y}'$ must be inside P because P is convex. Thus any point $\mathbf{y}''$ on the line segment is within P. Figure A.1 shows that $d(\mathbf{x}, \mathbf{y}'') < d(\mathbf{x}, \mathbf{y})$, violating the assumption that $d(\mathbf{x}, \mathbf{y})$ is minimal. The lemma thus must be true by contradiction. □

Lemma A.2 *The gradient of D, ∇D, exists outside P; for any $\mathbf{x} \in \mathbb{R}^3$ outside P with corresponding $\mathbf{y}$ as in Lemma A.1, $\nabla D(\mathbf{x})$ is parallel to $\mathbf{y} - \mathbf{x}$, and $|\nabla D(\mathbf{x})| = 1$.*

Proof: A rigorous proof that ∇D exists outside P is tedious. Informally, however, the existence of ∇D follows by analogy to the situation for a polygon. If P were a polygon in $\mathbb{R}^2$, then its distance function would have a graph in $\mathbb{R}^3$. This graph would be a smooth union of planes and cones; each

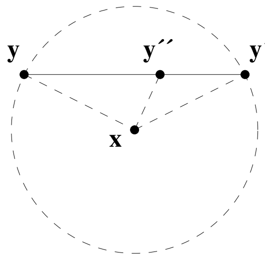


Figure A.1: Since $d(\mathbf{x},\mathbf{y}) = d(\mathbf{x},\mathbf{y}')$ it follows that $d(\mathbf{x},\mathbf{y}'') < d(\mathbf{x},\mathbf{y})$.

slice of it is a "push-off surface," as in Figure 5.15. By analogy, when P is a polyhedron, the graph of D is a smooth hypersurface in $\mathbb{R}^4$, and thus ∇D exists.

Now, let $\mathbf{u}$ be a unit vector in the direction $\mathbf{x}-\mathbf{y}$. The directional derivative of D along $\mathbf{u}$ at $\mathbf{x}$ is

$$\lim_{h\to 0} \frac{D(\mathbf{x}+h\mathbf{u}) - D(\mathbf{x})}{h} \quad = \quad \lim_{h\to 0} \frac{d(\mathbf{y},\mathbf{x}) + h - d(\mathbf{y},\mathbf{x})}{h} \quad = \quad 1.$$

For any other unit vector $\mathbf{v}$, the directional derivative of D along $\mathbf{v}$ at $\mathbf{x}$ is

$$\lim_{h\to 0} \frac{D(\mathbf{x}+h\mathbf{v}) - D(\mathbf{x})}{h} \quad \leq \quad \lim_{h\to 0} \frac{d(\mathbf{y},\mathbf{x}+h\mathbf{v}) - D(\mathbf{x})}{h}$$

by the definition of D. Thus

$$\lim_{h\to 0} \frac{D(\mathbf{x}+h\mathbf{v}) - D(\mathbf{x})}{h} \quad < \quad \lim_{h\to 0} \frac{d(\mathbf{y},\mathbf{x}) + h - d(\mathbf{y},\mathbf{x})}{h} \quad = \quad 1.$$

So the directional derivative in any direction $\mathbf{v} \neq \mathbf{u}$ is ≤ 1. The gradient must have the direction and magnitude of greatest change, so $\nabla D(\mathbf{x})$ is parallel to $\mathbf{y}-\mathbf{x}$, and $|\nabla D(\mathbf{x})| = 1$. □

Lemma A.3 *For* $\mathbf{x} \in \mathbb{R}^3$ *outside* P *with corresponding* $\mathbf{y}$ *as in Lemma A.1, if* $\mathbf{x}'$ *is on the ray from* $\mathbf{y}$ *through* $\mathbf{x}$ *then* $\nabla D(\mathbf{x}') = \nabla D(\mathbf{x})$.

Proof: First, say $D(\mathbf{x}') < D(\mathbf{x})$. In this case, $\mathbf{y}$ must be the point on P closest to $\mathbf{x}'$. The proof is by the triangle inequality, as illustrated in Figure A.2. More specifically, consider any point $\mathbf{y}'$ on P. By the triangle inequality, $d(\mathbf{y}',\mathbf{x}) \leq d(\mathbf{y}',\mathbf{x}') + d(\mathbf{x}',\mathbf{x})$. Say $\mathbf{y}'$ is the point on P closest to $\mathbf{x}'$. Then $d(\mathbf{y}',\mathbf{x}') + d(\mathbf{x}',\mathbf{x}) \leq d(\mathbf{y},\mathbf{x}') + d(\mathbf{x}',\mathbf{x}) = d(\mathbf{y},\mathbf{x})$ by the collinearity of $\mathbf{y}$, $\mathbf{x}'$ and $\mathbf{x}$. Thus $d(\mathbf{y}',\mathbf{x}) \leq d(\mathbf{y},\mathbf{x})$. But $\mathbf{y}$ satisfies Lemma A.1, so $d(\mathbf{y},\mathbf{x})$ is minimal and thus $d(\mathbf{y}',\mathbf{x}) = d(\mathbf{y},\mathbf{x})$. Thus $\mathbf{y} = \mathbf{y}'$ by the uniqueness result of Lemma A.1.

Now, say $D(\mathbf{x}') > D(\mathbf{x})$. In this case, the previous argument holds with the roles of $\mathbf{x}$ and $\mathbf{x}'$ reversed. Thus, $\mathbf{y}$ must be the point on P closest to $\mathbf{x}'$ in all cases.

Hence, Lemma A.2 states that $\nabla D(\mathbf{x}')$ is parallel to $\mathbf{y}-\mathbf{x}'$, and $|\nabla D(\mathbf{x}')| = 1$. Similarly, $\nabla D(\mathbf{x})$ is parallel to $\mathbf{y}-\mathbf{x}$, and $|\nabla D(\mathbf{x})| = 1$. Since $\mathbf{x}-\mathbf{y}$ is parallel to $\mathbf{x}'-\mathbf{y}$, it follows that $\nabla D(\mathbf{x}) = \nabla D(\mathbf{x}')$. □

Lemma A.4 *For* $\mathbf{w}$ *on the surface of the sphere,* s*, let* $\mathbf{n}$ *be the outward-pointing normal vector of* s *at* $\mathbf{w}$*. If for all* $\mathbf{w}' \in s$ $D(\mathbf{w}) \geq D(\mathbf{w}')$*, then* $\nabla D(\mathbf{w})$ *is parallel to* $\mathbf{n}$.

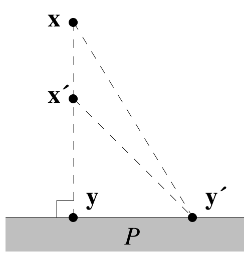


Figure A.2: The triangle inequality.

Proof: Let $\gamma(t)$ be an arbitrary smooth path on the sphere, s, with $\gamma(0) = \mathbf{w}$. Let $g(t) = D(\gamma(t))$. By hypothesis, $t = 0$ is a maximum for g, so $g'(0) = 0$. By the chain rule, $g'(0) = \nabla D(\gamma(0)) \cdot \gamma'(0) = \nabla D(\mathbf{w}) \cdot \gamma'(0)$. But $\gamma(t)$ is an arbitrary path on s, so $\nabla D(\mathbf{w})$ must be perpendicular to the tangent plane of s at $\mathbf{w}$. Thus $\nabla D(\mathbf{w})$ is parallel to $\mathbf{n}$. $\square$

A.2 c is outside P

Given these preliminary lemmas, the proof of Lemma 5.1 has two cases. The first involves $\mathbf{c}$ being outside P. In this case, the goal is to prove that the distance from s to P is $r + d(\mathbf{c}, \mathbf{c}')$.

Let $\mathbf{z}$ be the point on s furthest from P. By Lemma A.4, $\nabla D(\mathbf{z})$ is parallel to $\mathbf{n}$, the outward-pointing normal to s at $\mathbf{z}$. Let $\mathbf{y}$ be the point on P closest to $\mathbf{z}$. By Lemma A.2, $\mathbf{z} - \mathbf{y}$ is parallel to $\nabla D(\mathbf{z})$. Thus, $\mathbf{z} - \mathbf{y}$ is parallel to $\mathbf{n}$. The center, $\mathbf{c}$, of s clearly lies on the line through $\mathbf{z}$ having direction $\mathbf{n}$, so $\mathbf{c}$ lies on the ray from $\mathbf{y}$ through $\mathbf{z}$. Thus $\nabla D(\mathbf{c}) = \nabla D(\mathbf{z})$ by Lemma A.3. Recall that $\mathbf{c}'$ is a point on the surface of P closest to $\mathbf{c}$. By Lemma A.2, $\mathbf{c} - \mathbf{c}'$ is parallel to $\nabla D(\mathbf{c})$. Thus $\mathbf{c} - \mathbf{c}'$ is parallel to $\nabla D(\mathbf{z})$. Since $\nabla D(\mathbf{z})$ is parallel to $\mathbf{z} - \mathbf{y}$, $\mathbf{c} - \mathbf{c}'$ is parallel to $\mathbf{z} - \mathbf{y}$.

Thus $\mathbf{c}$ lies on the ray from $\mathbf{y}$ through $\mathbf{z}$, and it is simple to show that $d(\mathbf{z}, \mathbf{y}) = r + d(\mathbf{c}, \mathbf{c}')$, where r is the radius of s. Thus the proof of the first case, for $\mathbf{c}$ is outside P, is complete.

A.3 c is inside P

The second case involves $\mathbf{c}$ being inside P. In this case, the goal is to prove that the distance from s to P is $r - d(\mathbf{c}, \mathbf{c}')$.

By hypothesis, $\mathbf{c}'$ is a point on the surface of P such that $d(\mathbf{c}, \mathbf{c}') \leq d(\mathbf{c}, \mathbf{c}'')$, for all other $\mathbf{c}''$ on the surface of P. (Since $\mathbf{c}$ is inside P, $\mathbf{c}'$ may not be unique.) The point $\mathbf{c}'$ must be inside s, because points inside s are closer to $\mathbf{c}$ than points outside s; also, there will be at least one point of P inside s because s intersects P.

Lemma A.5 *The point* $\mathbf{c}'$ *is interior to a face of* P.

Proof: Suppose $\mathbf{c}'$ is on an edge between two faces of P, f_1 and f_2. Let s' be a sphere centered at $\mathbf{c}$ with radius $d(\mathbf{c}, \mathbf{c}')$. Since $\mathbf{c}'$ is a closest point to $\mathbf{c}$ on the surface of P, it follows that $s' \subseteq P$. Let p_1 be the plane containing f_1. Since P is convex, it is the intersection of the halfspace behind p_1 and the halfspaces behind the planes for all its other faces. Thus s' touches p_1 at $\mathbf{c}'$ and lies on one side of p_1, so p_1 must be tangent to s' at $\mathbf{c}'$. The same holds for p_2, the plane of f_2. Thus, p_1 and p_2 are coplanar, so f_1 and f_2 must be the same face. Note that the same argument shows that $\mathbf{c}'$ cannot be on a vertex of P. $\square$

A corollary of this lemma is the fact that a ray from $\mathbf{c}$ through $\mathbf{c}'$ is normal to a face of P. Now, construct the point, $\mathbf{z}'$, where this ray intersects the surface of s. Since P is convex, $\mathbf{z}'$ must be outside P. From the Lemma A.5 and its corollary, $d(\mathbf{z}', P) = r - d(\mathbf{c}, \mathbf{c}')$. Lemma A.4 and Lemma A.2 prove that $\mathbf{c}$ is on the line between $\mathbf{z}$ and $\mathbf{y}$, where $\mathbf{z}$ is the point on s furthest from P and $\mathbf{y} \in P$ is the unique point with $d(\mathbf{z}, \mathbf{y}) = D(\mathbf{z})$ (as in Lemma A.1). Thus $d(\mathbf{z}, P) = r - d(\mathbf{c}, \mathbf{y})$. By hypothesis, $d(\mathbf{z}, P) \geq d(\mathbf{z}', P)$. Thus $d(\mathbf{c}, \mathbf{y}) \leq d(\mathbf{c}, \mathbf{c}')$. Since $d(\mathbf{c}, \mathbf{c}')$ is minimal by hypothesis, $d(\mathbf{c}, \mathbf{c}') = d(\mathbf{c}, \mathbf{y})$.

The distance from s to P is $d(\mathbf{z}, P) = r - d(\mathbf{c}, \mathbf{y})$. But $d(\mathbf{c}, \mathbf{c}') = d(\mathbf{c}, \mathbf{y})$, so the distance from s to P is $d(\mathbf{z}, P) = r - d(\mathbf{c}, \mathbf{c}')$. The proof of Lemma 5.1 is thus complete.

Appendix B

The Boundary Lemma and General Covering Problems

The boundary lemma, Lemma 5.2 from Section 5.3.3, applies to situations that are more general than the coverage of a triangle by disks. Any two-dimensional (2D) region is covered by a set of arbitrary 2D shapes if and only if the portion of each shape's boundary inside the region is covered by the other shapes. A variation of the algorithm in Figure 5.17 will work for any such situation. All that is required is a way to find the part of a shape's boundary which is inside the region, and a way to find what part of that boundary is covered by another shape. The rest of the algorithm in Figure 5.17 manipulates 1D intervals, and these parts need no changes.

The running time of this algorithm will depend on how efficient it is to find intersections between a shape and the region, and intersections between two shapes. For situations involving many shapes, the loop starting at line 7 will also become an important factor. Checking every pair of shapes for intersection will no longer be efficient. A heuristic that reduces the number of shape comparisons is as follows: bound each shape with an isothetic rectangle and compare only pairs of shapes whose bounding rectangles intersect. Finding the bounding rectangles that intersect is an efficient operation; Preparata and Shamos [PS85] present a plane-sweep algorithm that uses the *interval tree* data structure to report the k_r intersecting rectangles in optimal $\theta(n \log n + k_r)$ time.

The potential disadvantage of this heuristic is the time it spends finding bounding rectangles that intersect but whose inscribed shapes do not intersect. If k_r' is the number of these intersecting rectangles, the overall performance of the algorithm is $\Omega(n \log n + k_r + k_r')$. The factor k_r' is not a problem, however, if the shapes have a bounded *aspect ratio*, that is, if the ratio of the sizes of two shapes never exceeds A. In this case, $k_r' = \mathrm{O}(n)$, which means that the performance is asymptotically no different from the case of finding only rectangle intersections (assuming the time to check two shapes for intersection depends only loosely on the number of shapes, that is, it is $\mathrm{O}(\log n)$).

The proof that $k_r' = \mathrm{O}(n)$ is as follows. The key is showing that even in the worst case, any bounding rectangle can intersect only a constant number of other rectangles without any inscribed shapes intersecting, that is, the rectangle can be involved in only a constant number of "spurious intersections." The worst case occurs when a rectangle, x, is of maximum size and every rectangle, x', that it intersects is of minimum size. In this situation, without loss of generality, x has sides of length A and x' has sides of length 1. Let X be the smallest square that bounds every x' that x intersects. The sides of X must have length $A + 2$. If y is the shape inscribed in x, then the area of $X - y$ divided by the area of x' gives an upper bound on the number of spurious intersections involving x. It is reasonable to assume that the area of a bounding rectangle will be constant factor, z, of the area of the shape it bounds. The area of $X - y$ is thus $(A+2)^2 - A^2/z$, and the area of x' is 1, so the number of spurious intersections is independent of n. Thus, the proof that $k'_r = \mathrm{O}(n)$ is complete.

An algorithm based on the boundary lemma may not be the most efficient way to detect coverage in some situations. If the boundaries of the shapes consist of a small number of curves that have monotonically increasing y coordinates, then a variation of the Bentley-Ottmann algorithm [BO79, PS85] can find the intersections between these shapes efficiently. If, additionally, the region to be covered is a polygon, then the plane-sweep of the Bentley-Ottmann algorithm could work in conjunction with a structure like an interval tree to detect any gaps in the coverage by the shapes. Further development of this idea is beyond the scope of this dissertation.

Bibliography

[AB94] Ronald Azuma and Gary Bishop. Improving Static and Dynamic Registration in an Optical See-Through HMD. In *Proceedings of SIGGRAPH '94,* published as *Computer Graphics* Proceedings, Annual Conference Series, pages 197–204, July 1994.

[ARB90] John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics (Snowbird, Utah)*, pages 41–50, 1990.

[Bar81] Alan H. Barr. Superquadrics and Angle-Preserving Transformations. *IEEE Computer Graphics and Applications*, volume 1, number 1, pages 11–23, January 1981.

[Bar89] David Baraff. Analytical Methods for Dynamic Simulation of Non-Penetrating Rigid Bodies. In *Proceedings of SIGGRAPH '89,* published as *Computer Graphics*, volume 23, number 3, pages 223–232, July 1989.

[Bar90] David Baraff. Curved Surfaces and Coherence for Non-Penetrating Rigid Body Simulation. In *Proceedings of SIGGRAPH '90,* published as *Computer Graphics*, volume 24, number 4, pages 19–29, August 1990.

[Bar91] David Baraff. Coping with Friction for Non-Penetrating Rigid Body Simulation. In *Proceedings of SIGGRAPH '91,* published as *Computer Graphics*, volume 25, number 4, pages 31–40, July 1991.

[BBO+83] Bernard R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A Program for Macromolecular Energy, Minimization and Dynamics Calculations. *Journal of Computational Chemistry*, volume 4, number 2, pages 187–217, 1983.

[BFGS86] Larry Bergman, Henry Fuchs, Eric Grant, and Susan Spach. Image Rendering by Adaptive Refinement. In *Proceedings of SIGGRAPH '86,* published as *Computer Graphics*, volume 20, number 4, pages 29–37, August 1986.

[Bio92] Frank Biocca. Will Simulation Sickness Slow Down the Diffusion of Virtual Environment Technology? *Presence*, volume 1, number 3, pages 334–343, Summer 1992.

[Blu67] 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.

[BN78] Harry Blum and Roger N. Nagel. Shape Description Using Weighted Symmetric Axis Features. *Pattern Recognition*, volume 10, pages 167–180, 1978.

[BO79] Jon L. Bentley and Thomas A. Ottmann. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Transactions on Computers*, volume C-28, number 9, pages 643–647, September 1979.

[BOT79] Norman I. Badler, Joseph O'Rourke, and Hasida Toltzis. A Spherical Representation of a Human Body for Visualizing Movement. *Proceedings of the IEEE*, volume 67, number 10, pages 1397–1403, October 1979.

[Bow81] Adrian Bowyer. Computing Dirichlet Tessellations. *The Computer Journal*, volume 24, number 2, pages 162–166, 1981.

[Boy79] John W. Boyse. Interference Detection Among Solids and Surfaces. *Communications of the ACM*, volume 22, number 1, pages 3–9, January 1979.

[Bro88] Frederick P. Brooks, Jr. Grasping Reality Through Illusion—Interactive Graphics Serving Science. In *Proceedings of CHI '88*, pages 1–11, May 1988.

[BV91] William J. Bouma and George Vaněček, Jr. Collision Detection and Analysis in Physically Based Simulation. In *Proceedings of the Second Eurographics Workshop on Animation and Simulation*, pages 191–203, September 1991.

[BW92] David Baraff and Andrew Witkin. Dynamic Simulation of Non-Penetrating Flexible Bodies. In *Proceedings of SIGGRAPH '92,* published as *Computer Graphics*, volume 26, number 2, pages 303–308, July 1992.

[Cam89] Stephen A. Cameron. Efficient Intersection Tests for Objects Defined Constructively. *International Journal of Robotics Research*, volume 8, number 1, pages 3–25, February 1989.

[Cam90] Stephen A. Cameron. Collision Detection by Four-Dimensional Intersection Testing. *IEEE Transactions on Robotics and Automation*, volume 6, number 3, pages 291–302, June 1990.

[Can86] John Canny. Collision Detection for Moving Polyhedra. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 8, number 2, pages 200–209, March 1986.

[CCWG88] Michael F. Cohen, Shengchang Eric Chen, John R. Wallace, and Donald P. Greenberg. A Progressive Refinement Approach to Fast Radiosity Image Generation. In *Proceedings of SIGGRAPH '88,* published as *Computer Graphics*, volume 22, number 4, pages 75–84, August 1988.

[CK86] 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.

[Cla76] James H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, volume 19, number 10, pages 547–554, October 1976.

[CM86] Eugene Charniak and Drew V. McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley, Reading, Massachusetts, 1986.

[DB88] Thomas Dean and Mark Boddy. An Analysis of Time-Dependent Planning. In *Proceedings of AAAI-88*, pages 49–54, 1988.

[Dee92] Michael Deering. High Resolution Virtual Reality. In *Proceedings of SIGGRAPH '92,* published as *Computer Graphics*, volume 26, number 2, pages 195–202, July 1992.

[DK83] David P. Dobkin and David G. Kirkpatrick. Fast Detection of Polyhedral Intersection. *Theoretical Computer Science*, volume 27, number 3, pages 241–253, December 1983.

[Duf92] Tom Duff. Interval Arithmetic and Recursive Subdivision for Implicit Functions and Constructive Solid Geometry. In *Proceedings of SIGGRAPH '92,* published as *Computer Graphics*, volume 26, number 2, pages 131–138, July 1992.

[Dwo93] Paul Dworkin. A New Model for Efficient Dynamic Simulation. Master's thesis, Media Laboratory, Massachusetts Institute of Technology, 1993.

[EG78] Robert Ellis and Denny Gulick. *Calculus with Analytic Geometry.* Harcourt Brace Jovanovich, New York, New York, 1978.

[FB92] Henry Fuchs and Gary Bishop. Research Directions in Virtual Environments. *Computer Graphics*, volume 26, number 3, pages 153–177, 1992.

[FHA90] 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.

[FKN80] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On Visible Surface Generation by A Priori Tree Structures. In *Proceedings of SIGGRAPH '80,* published as *Computer Graphics*, volume 14, number 3, pages 124–133, July 1980.

[FS93] 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.

[FSP92] Martin Friedmann, Thad Starner, and Alex Pentland. Device Synchronization Using an Optimal Linear Filter. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics (Cambridge, Massachusetts)*, pages 57–62, 1992.

[FvFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, Massachusetts, 2nd edition, 1990.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, New York, 1979.

[Gol90] Ronald Goldman. Intersection of Three Planes. In Andrew S. Glassner, editor, *Graphics Gems*, page 305. Academic Press, Boston, Massachusetts, 1990.

[Gos93] Rich Gossweiler. Time-Critical Rendering in an Immersive Virtual Environment. Ph.D. Dissertation Proposal, School of Engineering and Applied Science, University of Virginia, June 1993.

[GS87] Jeffrey Goldsmith and John Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications*, volume 7, number 5, pages 14–20, May 1987.

[GYKD91] 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*, volume 1, number 3, pages 327–339, 1991.

[Hah88] James K. Hahn. Realistic Animation of Rigid Bodies. In *Proceedings of SIGGRAPH '88,* published as *Computer Graphics*, volume 22, number 4, pages 299–308, August 1988.

[HDD+93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh Optimization. In *Proceedings of SIGGRAPH '93,* published as *Computer Graphics* Proceedings, Annual Conference Series, pages 19–26, August 1993.

[Hof90] 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. (Also Technical Report CSD-TR-1014, Computer Sciences Department, Purdue University.).

[Hof92] Christoph M. Hoffmann. Computer Vision, Descriptive Geometry and Classical Mechanics. In B. Falcidieno, I. Herman, and C. Pienovi, editors, *Computer Graphics and Mathematics*. Springer Eurographics Series, 1992.

[HR92] Lawrence J. Hettinger and Gary E. Riccio. Visually Induced Motion Sickness in Virtual Environments. *Presence*, volume 1, number 3, pages 306–310, Summer 1992.

[HS92] Wolfgang A. Halang and Krzysztof Sacha. *Real-Time Systems: Implementation of Industrial Computerised Process Automation.* World Scientific, Singapore, 1992.

[Hub90] Philip M. Hubbard. Constructive Solid Geometry for Triangulated Polyhedra. Technical Report CS-90-07, Department of Computer Science, Brown University, September 1990.

[ISS92] 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.

[Jan85] P. A. J. Janssen. Strategies in Pharmaceutical Research. *Endeavour*, volume 9, number 1, pages 28–33, 1985.

[KLL+92] Robert S. Kennedy, Norman E. Lane, Michael G. Lilienthal, Kevin S. Berbaum, and Lawrence J. Hettinger. Profile Analysis of Simulator Sickness Symptoms: Application to Virtual Environment Systems. *Presence*, volume 1, number 3, Summer 1992.

[Kwo88] Paul C. K. Kwok. A Thinning Algorithm by Contour Generation. *Communications of the ACM*, volume 31, number 11, pages 1314–1324, November 1988.

[LA87] Peter J. M. van Laarhoven and Emile H. L. Aarts. *Simulated Annealing: Theory and Applications.* D. Reidel Publishing Company, Dordrecht, Holland, 1987.

[LC91] 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.

[LMC93] Ming C. Lin, Dinesh Manocha, and John F. Canny. Fast Collision Detection between Geometric Models. Technical Report TR93-004, Department of Computer Science, The University of North Carolina at Chapel Hill, January 1993.

[LNA88] 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.

[LSG91] Jiandong Liang, Chris Shaw, and Mark Green. On Temporal-Spatial Realism in the Virtual Reality Environment. In *Proceedings of the 1991 ACM Symposium on User Interface Software and Technology*, pages 19–25, November 1991.

[LTH86] David H. Laidlaw, W. Benjamin Trumbore, and John F. Hughes. Constructive Solid Geometry for Polyhedral Objects. In *Proceedings of SIGGRAPH '86,* published as *Computer Graphics*, volume 20, number 4, pages 161–169, August 1986.

[Lub91] Boris D. Lubachevsky. How to Simulate Billiards and Similar Systems. *Journal of Computational Physics*, volume 94, number 1, pages 255–283, May 1991.

[MG93] Tom Meyer and Al Globus. Direct Manipulation of Isosurfaces and Cutting Planes in Virtual Environments. Technical Report CS-93-54, Department of Computer Science, Brown University, December 1993.

[MH82] Jerry B. Marion and William F. Hornyak. *Physics for Science and Engineering*. Saunders College Publishing, Philadelphia, Pennsylvania, 1982.

[MK84] S. P. Mudur and P. A. Koparkar. Interval Methods for Processing Geometric Objects. *IEEE Computer Graphics and Applications*, volume 4, number 2, pages 7–17, February 1984.

[MT83] Martti Mäntylä and Markku Tamminen. Localized Set Operations for Solid Modeling. In *Proceedings of SIGGRAPH '83,* published as *Computer Graphics*, volume 17, number 3, pages 279–287, July 1983.

[MW88] Matthew P. Moore and Jane Wilhelms. Collision Detection and Response for Computer Animation. In *Proceedings of SIGGRAPH '88,* published as *Computer Graphics*, volume 22, number 4, pages 289–298, August 1988.

[NP85] Lee R. Nackman and Stephen M. Pizer. Three-Dimensional Shape Description Using the Symmetric Axis Transform I: Theory. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume PAMI-7, number 2, pages 187–202, March 1985.

[OB79] Joseph O'Rourke and Norman Badler. Decomposition of Three-Dimensional Objects into Spheres. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume PAMI-1, number 3, pages 295–305, July 1979.

[Pae91] Alan W. Paeth. Exact Dihedral Metrics for Common Polyhedra. In James Arvo, editor, *Graphics Gems II*, pages 174–178. Academic Press, Boston, Massachusetts, 1991.

[PM92] Pieter Padmos and Maarten V. Milders. Quality Criteria for Simulator Images: A Literature Review. *Human Factors*, volume 34, number 6, pages 727–748, June 1992.

[PS85] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, New York, 1985.

[PTVF92] 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.

[RH94] 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.

[Rit90] Jack Ritter. An Efficient Bounding Sphere. In Andrew S. Glassner, editor, *Graphics Gems*, pages 301–303. Academic Press, Boston, Massachusetts, 1990.

[Sch81] Bruce J. Schacter. Computer Image Generation for Flight Simulation. *IEEE Computer Graphics and Applications*, volume 1, number 4, pages 29–68, October 1981.

[Sed88] Robert Sedgewick. *Algorithms.* Addison-Wesley, Reading, Massachusetts, 2nd edition, 1988.

[SH92] Clifford A. Shaffer and Gregory M. Herb. A Real-Time Robot Arm Collision Avoidance System. *IEEE Transactions on Robotics and Automation*, volume 8, number 2, pages 149–160, April 1992.

[Sny92] John M. Snyder. Interval Analysis for Computer Graphics. In *Proceedings of SIGGRAPH '92,* published as *Computer Graphics*, volume 26, number 2, pages 121–130, July 1992.

[SP91] Stan Sclaroff and Alex Pentland. Generalized Implicit Functions for Computer Graphics. In *Proceedings of SIGGRAPH '91,* published as *Computer Graphics*, volume 25, number 4, pages 247–250, August 1991.

[SS23] Arthur Schultze and Frank L. Sevenoak. *Plane and Solid Geometry.* The Macmillan Company, New York, New York, 1923.

[ST85] Hanan Samet and Markku Tamminen. Bintrees, CSG Trees, and Time. In *Proceedings of SIGGRAPH '85,* published as *Computer Graphics*, volume 19, number 3, pages 121–130, July 1985.

[SW88a] Hanan Samet and Robert E. Webber. Hierarchical Data Structures and Algorithms for Computer Graphics Part I: Fundamentals. *IEEE Computer Graphics and Applications*, volume 8, number 3, pages 48–68, May 1988.

[SW88b] Hanan Samet and Robert E. Webber. Hierarchical Data Structures and Algorithms for Computer Graphics Part II: Applications. *IEEE Computer Graphics and Applications*, volume 8, number 4, pages 59–75, July 1988.

[SWF+93] 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.

[SZL92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of Triangle Meshes. In *Proceedings of SIGGRAPH '92,* published as *Computer Graphics*, volume 26, number 2, pages 64–70, July 1992.

[TN87] 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*, volume 21, number 4, pages 153–162, July 1987.

[Tur90a] Greg Turk. Generating Random Points in Triangles. In Andrew S. Glassner, editor, *Graphics Gems*, pages 24–28. Academic Press, Boston, Massachusetts, 1990.

[Tur90b] 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.

[Tur91] Greg Turk. Generating Textures on Arbitrary Surfaces Using Reaction-Diffusion. In *Proceedings of SIGGRAPH '91,* published as *Computer Graphics*, volume 25, number 4, pages 289–298, August 1991.

[Tur92] Greg Turk. Re-Tiling Polygonal Surfaces. In *Proceedings of SIGGRAPH '92,* published as *Computer Graphics*, volume 26, number 2, pages 55–64, July 1992.

[Van91] George Vaněček, Jr. Brep-Index: A Multidimensional Space Partitioning Tree. *International Journal of Computational Geometry and Applications*, volume 1, number 3, pages 242–261, 1991.

[van93] 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.

[VBZ90] 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*, volume 24, number 4, pages 39–48, August 1990.

[Wlo93] Matthias M. Wloka. Time-Critical Graphics. Ph.D. Dissertation Proposal, Technical Report CS-93-50, Department of Computer Science, Brown University, November 1993.

[Wlo94] Matthias M. Wloka. Lag in Multiprocessor Virtual Reality. *Presence*, volume 3, number 4, 1994.

[YW93] Ji-Hoon Youn and K. Wohn. Realtime Collision Detection for Virtual Reality Applications. In *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pages 415–421, September 1993.

[ZOMP93] 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*, volume 4, number 1, pages 13–24, 1993.