A GRID-BASED PARALLEL COLLISION DETECTION ALGORITHM

BY

ORION SKY LAWLOR

B.S., University of Alaska at Fairbanks, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

Table of Contents

1 Introduction

Two physical objects cannot occupy the same space at the same time. Simulated physical objects do not naturally obey this constraint. Instead, we must detect when two objects have collided and rectify the situation.

For example, when using CAD/CAM to design a machine, we must ensure the simulated machine parts never pass though one another. If the simulated parts collide, we must correct the design.

In computer graphics and animation, we often want to ensure that objects behave in a physically plausible way. This means checking if the simulated objects penetrate one another– if they do, the modeling tool or animator will want to know.

In motion planning for robotics, we must check if a robot arm will collide with anything as it executes a proposed command. The command will have to be modified if a collision is possible.

In simulating a car crash or tearing metal, at each timestep we must check if any objects intersect. If they do, we must deform or displace the objects.

Collision detection, also known as contact or interference detection, is the problem of determining whether a given set of objects overlap. If the objects overlap, in some situations we need to find exactly which parts of the objects overlap and the depth of the overlap region. The problem can be formulated in 2D, 3D, or higher dimensions; can vary with time; and, as shown above and in Figure 1.1, is relevant to many domains.

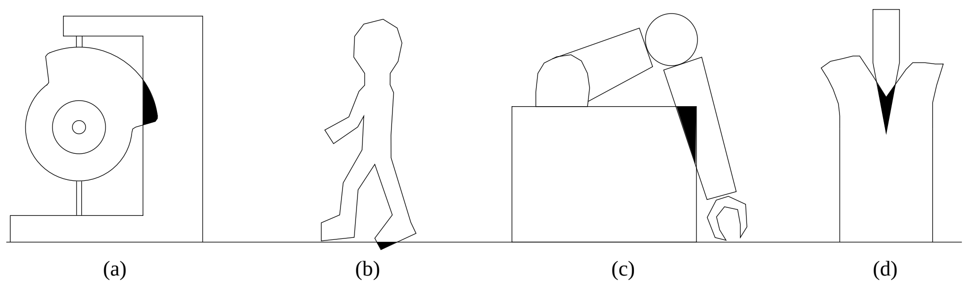

Figure 1.1: Collision detection for: (a) CAD/CAM (b) Computer animation (c) Robotics (d) Physical simulation

In this work, we only consider collision detection. Determining the appropriate response to a collision is an interesting problem, but the solution completely depends on the particular domain.

1.1 Summary

In this work, we present a scalable high-level parallel solution to a large subclass of collision detection problems. Our approach is to divide space into a sparse grid of regular axis-aligned voxels distributed across the parallel machine. Objects are then sent to all the voxels they intersect. Once all the objects have arrived, each voxel becomes a self-contained subproblem, which is then solved using standard serial collision detection approaches.

The voxels efficiently and naturally separate many objects that cannot ever collide, by placing them in separate voxels. Simultaneously, it brings together adjacent objects that may intersect.

The basic voxel algorithm is theoretically independent of the object type and serial collision detection method used. In practice, the method works best when there is not too much scale disparity– objects much larger than a voxel will be sent to several voxels, wasting space and time; while structures much smaller than a voxel must be dealt with entirely via the serial approach. Further, the serial collision detection method used must be able to gracefully handle objects entering and leaving the problem space as time progresses.

For problems which conform to these limitations, however, the serial and parallel performance of the voxel scheme is excellent. Subject to certain reasonable assumptions, the theoretical time taken by the voxel algorithm for n objects on p processors is the optimal $O(n/p)$– that is, linear serial time and perfect parallel speedup. In this work, we present and analyze an implementation of the voxel method.

1.2 Prior Work

Many researchers, from many domains, have addressed the problem of collision detection. Following Lin and Gottschalk in [21], we separate out two main areas of work: n-body collision detection, the "broad phase" which determines which pairs among n objects may intersect; and pairwise collision detection, the "narrow phase" which decides whether a single pair of objects collide. Any n-body algorithm depends on a pairwise algorithm to perform the final tests.

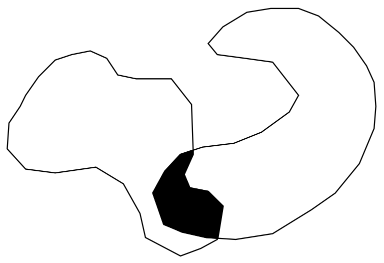

Figure 1.2: Two colliding objects, with their intersection shown in black

Finally, although the two problems are distinct, the common and literature usage of the term "collision detection" can refer to work in either n-body or pairwise detection. The voxel method, whose presentation forms the bulk of this work, addresses the n-body problem.

1.2.1 Pairwise collision detection

The basic problem of pairwise collision detection is to determine whether two given objects collide, as illustrated in Figure 1.2. Hubbard [10] points out that most collision detection research ignores the time dependence of collision problems. Although methods for fully four dimensional "spacetime" collision detection have been known since Boyse's work in 1979 [2], the usual approach is to ignore time dependence and treat each timestep or frame of the simulation as a conceptually separate problem[1].

To answer the question "Do these two objects collide?", we must first decide what we mean by "objects." Computational geometers use a surprising number of fundamentally different ways to represent objects. The most basic division is between solid and boundary representations– a solid representation directly indicates which points are inside the object and which are outside; a boundary representation only gives the outer shell of an object.

With an *implicit surface*, the value of a (typically continuous) function $f : \Re^3 \mapsto \Re$ indicates whether a point is inside or outside the solid. Then $I = \{\vec{x} \in \Re^3 | f(\vec{x}) < 0\}$. Without restrictions on f, the pairwise collision detection problem is not solvable.[2] Popular restricted classes of functions include quadrics and more general polynomials, for which efficient collision tests are known [6].

Parametric surfaces (such as NURBS or Bézier surfaces [17]) represent

[1] Although many collision detection algorithms use temporal coherence, they use it only to speed up each timestep. The basic problem is still posed in terms of timesteps, which are not ideal.

[2] Consider a surface defined by a noncomputable function f!

the boundary of an object as the image of a lower-dimension shape under a mapping function such as $f : \Re^2 \mapsto \Re^3$, so $P = \{f(\vec{x}) | \vec{x} \in \Re^2\}$. Pairwise collision detection for parametric surfaces are typically based on subdivision.

The *constructive solid geometry* (CSG) representation constructs solid objects from simple fundamental shapes via set operations such as union, intersection, and difference. Pairwise collision depends on the fundamental shapes chosen, but these are typically chosen to ensure collisions are efficient. A useful formalism for this case is "S-Bounds," described by Cameron in [4].

Polyhedral decompositions approximate a solid object as the disjoint union of convex polytopes. Several well-known results on polyhedra are those of Lin-Canny [20] and Gilbert-Johnson-Keerthi [8], which use convex optimization techniques to achieve performance as good as $O(\lg m)$, where m is the number of vertices in the polytope. Further, using temporal coherence, these algorithms exhibit near constant-time performance for slow-moving objects. Sadly, this result is often misinterpreted to mean the n-body collision detection problem can be solved in logarithmic time, which is not the case.

The lowest common denominator boundary representation is *triangles*, often (imprecisely) called polygons. Pairwise collision between triangles takes constant time.

Finally, if the objects intersect, depending on the application, we may also want to know the penetration distance and direction, or the size of the intersection region. For simple shapes, this is often available at little or no extra effort.

1.2.2 n-body collision detection

Given an algorithm to determine if a given pair of objects collide, the obvious way to determine which of n objects collide is simply to try all $\binom{n}{2}$ possible pairs. Because $\binom{n}{2} = O(n^2)$, this approach requires $O(n^2)$ pair tests.

In fact, *any* correct n-body collision detection algorithm has a worst-case time in $O(n^2)$. This is because it is possible to arrange n objects (see Figure 1.3) so that every object collides with every other object, for $O(n^2)$ separate collisions.

However, because physical objects cannot pass though one another, nearly any plausible physical situation actually has at most $O(n)$ collisions, as in Figure 1.4(a). Many situations found in practice have as few as $O(1)$ collisions, as shown in Figure 1.4(b).

Since these rare-collision cases are common, it is quite often possible to

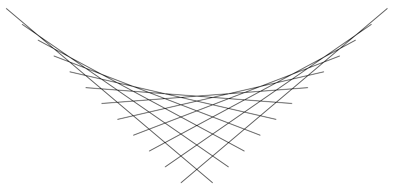

Figure 1.3: n line segments arranged to have $\binom{n}{2}$ collisions

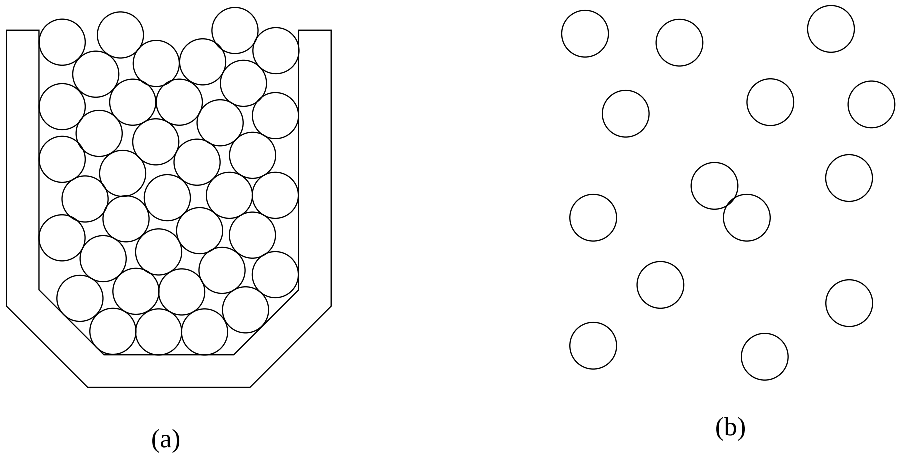

Figure 1.4: n circles; (a) with $O(n)$ collisions; (b) with $O(1)$ collisions

do collision detection in faster than $O(n^2)$ time. There are several ways to do this, each with their own advantages. For rigid objects with bounded velocities and accelerations, the collision scheduling method of [19] can be effective.

Most other n-body approaches use some sort of bounding volume. Bounding volumes, although shown quite effective in practice, are subject to rather unlikely worst cases, such as the nested 'L' shapes shown in Figure 1.5. Suri, Hubbard, and Hughes [23] show that if the object aspect ratio and "scale factor"[3] are bounded, then bounding boxes introduce no more than a linear amount of overhead. Zhou and Suri [25] extend this result to include bounded *average* aspect ratio and scale factor. These theoretical results confirm the practical effectiveness of bounding boxes.

The two major approaches to n-body collision detection [11] are object subdivision, where we divide up objects into parts that cannot intersect; and spatial subdivision, where we divide up space to separate objects that cannot intersect.

[3]Ratio of the volume of the bounding boxes of the smallest and largest object.

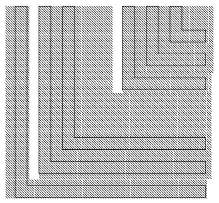

Figure 1.5: n nested 'L' figures do not collide, but every bounding box intersects every other bounding box.

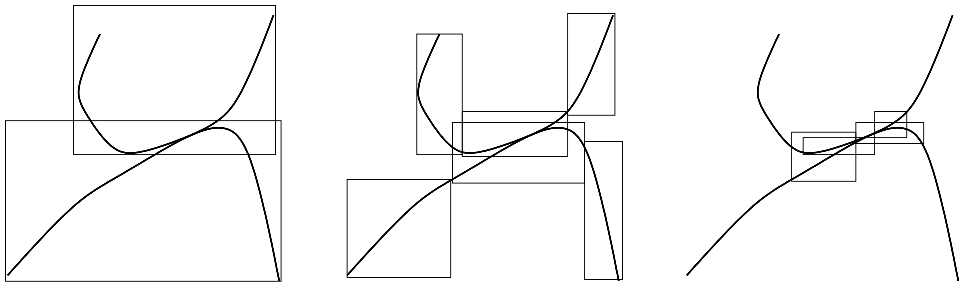

Figure 1.6: Object subdivision to intersect two curves. From left to right: the curves' top-level bounding boxes overlap; so the algorithm recursively examines the curves' subsections; overlapping subsections are examined further

Object subdivision

Object subdivision methods traverse a hierarchical bounding volume tree containing the objects. Some objects are naturally hierarchical; but often the bounding hierarchy must be constructed, typically at a cost in $O(n \lg n)$. The hierarchy is extended upward until there are a small number of top-level objects.

Collision detection begins at this top level. If no top-level bounding volumes intersect, the algorithm is finished; otherwise the next level bounding boxes for the offending objects are examined. The process recursively descends the object tree wherever needed until the lowest level is reached, when the algorithm falls back to the simple pairwise collision detection of Section 1.2.1. See Figure 1.6 for an example.

One major advantage of this approach is that for repeated queries on rigid objects, the hierarchical representation need only be built once. If there are few collisions, so only a few parts of the tree are examined, sublinear performance is possible.

A variety of bounding volumes have been examined in the literature. Axis-aligned bounding boxes and spheres are well known [10]. Gottschalk et

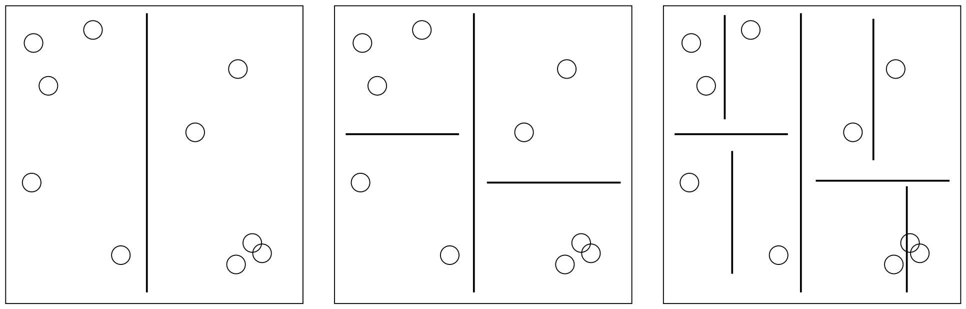

Figure 1.7: Recursive coordinate bisection on a set of small spheres. From left to right: problem is bisected once horizontally; recursively bisected vertically; then recursively bisected horizontally again

al. [9] examines oriented bounding boxes; Klosowski et al. [15] uses discrete-orientation polytopes (k-DOPs); and Krishnan et al. [16] use a higher-order bounding volume that can closely approximate curved surfaces.

For deformable objects, object subdivision schemes must rebuild the bounding volume hierarchy at each timestep. For engineering and simulation applications, this is a significant limitation.

Spatial subdivision

Spatial subdivision divides objects based on their location, rather than a prearranged hierarchy. The classic spatial subdivision method is *recursive coordinate bisection*, or RCB. In this method, the domain is repeatedly halved by choosing an axis-aligned cutting plane and dividing the objects on one side from those on the other side– see Figure 1.7. Eventually, the subproblems are small enough that the all-pairs method is appropriate.

With axis-aligned cutting planes, we can determine which side an object belongs simply by examining its axis-aligned bounding box, so dividing the problem takes $O(n)$ time. Ignoring objects that straddle the cutting plane (and thus must be kept on both sides), and assuming each division cuts the problem into two nearly equal-sized halves, the RCB method requires $O(\lg n)$ divisions and hence runs in $O(n \lg n)$ time.

Plimpton, Attaway, Hendrickson et al. [22] use a parallel RCB scheme to perform collision detection in a transient dynamics application. The parallel implementation of RCB requires at least $O(\lg p)$ synchronized communication steps, however, so the approach has a rather high synchronization overhead[4].

[4]Around 25 milliseconds in [22].

Allowing the cutting planes to have arbitrary orientation results in a *binary space partition*, first described by Fuchs et al. in [7]. The theoretical runtime is the same as that of RCB.

Another spatial subdivision scheme is the *sweep-line method*, often used for line segment intersection in 2D. This method first sorts the significant points of each object by x coordinates, then maintains a sorted y neighbor list as it proceeds along x, testing each new object against its neighbors. The sorting and neighbor lists make this algorithm $O(n \lg n)$ average case, like RCB.

Several other spatial subdivision schemes have been described, such as octrees [1] and methods derived from ray tracing. By reduction to sorting, however, any subdivision scheme based on comparing object locations will require at least $O(n \lg n)$ time; with parallel implementations that require multiple synchronization steps. The voxel method, described next, in some situations achieves $O(n)$ performance with excellent parallel efficiency.

2 The Voxel Method

The n-body, spatial subdivision approach we examine in the remainder of this work is the *voxel method.* A regular, axis-aligned[1] grid of voxels is imposed over the problem domain. Each object is added to all the voxels it touches, and the voxels are then collided independently. Any collision detection subscheme may be used to detect collisions within a voxel, from the simple all-pairs test to a recursive application of the voxel method on a smaller domain.

The voxel method was first described by Turk [24] for molecules, but his approach hashed the grid locations into buckets, ignoring hash collisions, and used the naive all-pairs algorithm on each bucket. Zyda et al. [26] describe the NPSNET system, which tests for vehicle/vehicle and vehicle/terrain collision using a parallel 2D grid structure quite similar to the voxel method. Zyda also uses the naive all-pairs approach within each grid cell.

When the objects to be collided are of nearly uniform size, the voxel method is ideal and gives excellent results. For objects much smaller than a voxel, the division into voxels does not help much, and the voxel method degenerates to the subscheme used within a voxel.

Objects much larger than a voxel will be duplicated across many voxels, as shown in Figure 2.1. In this case, a careful implementation can avoid duplicating collision tests across voxels by using a convention to decide which voxel is responsible for testing these shared objects. For example, only the voxel that contains the midpoint between the object centers need test the objects for collision. Even with this optimization, however, large objects still waste space and time.

Voxels should be implemented as a sparse grid, via a hash table in most cases. Sparseness is especially important if the problem domain is not naturally bounded, objects are distributed in a nonuniform fashion, or if memory usage is important. A sparse grid can also improve efficiency, because empty cells are automatically ignored.

Although normally implemented in 3D, gridding works well with any number of dimensions. High-dimension problems may consume large quantities of memory, however. The common timestep "sweeping" approach

[1] Or in the terminology of [10], *isothetic.*

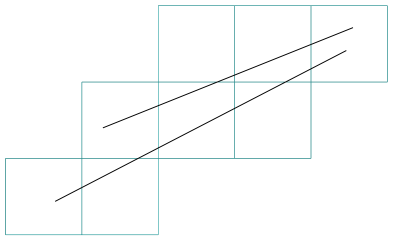

Figure 2.1: Large objects may span several voxels

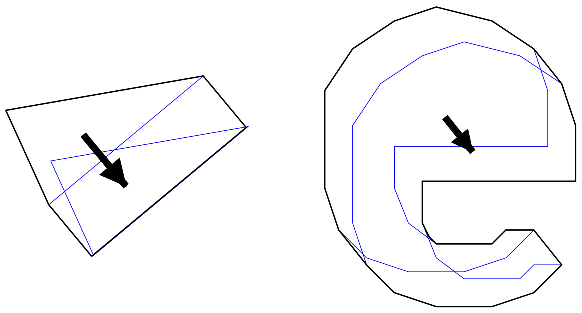

Figure 2.2: Sweeping objects between timesteps corresponds to a coarse timestep grid in 4D

illustrated in Figure 2.2 can be seen as a kind of gridding along the time axis of spacetime.

The voxel approach is easiest to implement via the sloppy strategy of adding each object to *all* the voxels touched by the object bounding box[2]. For elongated objects, whose bounding box does not fit well, this can be wasteful; the 3D digital differential analyzer approach of [1] could improve performance.

However, since most objects should fit inside a single voxel, gridding only the bounding box normally works well. The fast gridding method described in Appendix B is ideal for this case.

[2]This works because any irrelevant objects will be ignored by the collision detection subscheme

3 Parallel Implementation

Unlike many other object- or space-division collision detection schemes, the voxel method is naturally parallel. The description of a parallel implementation of the voxel algorithm is given below.

3.1 Charm++

The sparse voxel grid is implemented on top of the portable parallel runtime system CHARM++, presented in [12]. CHARM++ targets both shared- and distributed-memory parallel machines, and runs on machines from workstations to clusters to supercomputers. CHARM++ is a parallel library for C++, but includes bindings for C and FORTRAN90.

A parallel application in CHARM++ consists of a number of relatively small, self-contained *parallel objects*. These objects communicate via remote method invocation, similar to Java RMI or RPC[1], but asynchronous. Because each processor houses several objects, while one object is waiting for data, other local objects can use the CPU.

This data-driven approach thus automatically and transparently overlaps communication and computation, leading to better performance and easier application development. This approach compares quite favorably with the lower-level primitives provided by MPI, the dominant distributed memory parallel programming interface.

Parallel objects are implemented as ordinary C++ classes. They communicate with other parallel objects using a local *communication proxy* object, another regular C++ class. Proxies are automatically generated from an interface file that lists each classes' remote methods.

In this sense, CHARM++ has much the same interface as CORBA. However, CHARM++ targets tightly coupled parallel machines and aims for extremely efficient intra-process communication. The software messaging and scheduling overhead for CHARM++ is only a few microseconds.

The CHARM++ array framework, presented by the author in [18], supports parallel objects called *array elements* that can be dynamically created and destroyed, participate in reductions and broadcasts, and migrate from

[1]Remote procedure call, a standard protocol used by UNIX machines

one processor to another. Array elements are identified across the parallel machine by an *array index*, which need not be a contiguous range of integers–it can be a sparse user-defined data structure such as a multidimensional location, bitvector, or string.

Since array elements are migrateable, CHARM++ can improve load balance by occasionally migrating some objects from heavily loaded processors to less loaded processors. Migration-based load balancing can automatically compensate for application-induced load imbalance, minimize communication volume, handle variation among machines, or accommodate background load. The system is described in [3].

CHARM++ is an excellent foundation for parallel programming design, research and implementation. Several major, highly scalable parallel applications have been developed using CHARM++, including [13].

3.2 Interface

The input to this implementation of the voxel algorithm is a set of triangles presented on each processor. That is, the input is expected to already be distributed; although the method will still work if all the triangles are presented on a single processor. The details of the input format are described in Appendix A.

Triangles are widely used in computer graphics because they are a flexible, simple to handle lowest common denominator. It is easy to exactly triangulate any piecewise flat object; and curved objects can be approximated with any desired (but finite) degree of precision.

A collision is considered to have occurred if two triangles penetrate each other's planes. This definition does not require any "inside/outside object" tests, so no assumptions need be made about the underlying object topology. In particular, the triangles representing an object may form cracks, gaps, or holes without affecting the algorithm. This is important for supporting automatically generated or acquired geometric models, in which such irregularities may abound.

3.3 The Algorithm

The voxel method's parallel implementation is perhaps most succinctly described by its CHARM++ interface file:

```
module parCollide {
  message triListMsg;

  array [3D] collideVoxel {
    entry collideVoxel();
    entry [createhere] void add(triListMsg *);
    entry void startCollision(void);
  };

  group collideMgr : syncReductionMgr {
    entry collideMgr(CkArrayID collideVoxels);
    entry void voxelMessageRecvd(void);
  };
};
```

`collideVoxel` is a grid of voxels– a sparse three-dimensional CHARM++ parallel array. This allows voxels to be created on any processor, receive messages from any processor, and migrate from processor to processor.

`collideVoxel::add` is called to add a list of objects to the voxel. Because `add` is declared `createhere`, if an `add` message is sent to a non-existent grid location, CHARM++ will create a new `collideVoxel` object on the sending processor to handle the request. `triListMsg`, the "message" parameter to `add`, contains a list of triangles to be collided by the voxel.

`collideVoxel::startCollision` runs the serial collision detection subscheme on all the voxel's accumulated objects. We use the RCB method described in Section 1.2.2; although for triangles the BSP method can be nearly as efficient. An interesting approach might be to recursively apply the voxel method, especially if many objects lie in a single voxel.

`collideMgr` is a CHARM++ *group*, a special collection of parallel objects with exactly one object on every processor. The `collideMgr` sequences the collision detection computation.

3.3.1 Steps

The steps in one collision detection computation are as follows.

1. Contributors give the local `collideMgr` their objects to be collided.
2. The local `collideMgr` determines which voxels each object touches, and calls `collideVoxel::add` for the appropriate voxels.
3. Once the voxels have received all their objects, the `collideMgr` objects synchronize.
4. `startCollision` is broadcast to all voxels, which then run their serial collision detection algorithm.
5. The resulting lists of collisions are collected.

In step 1, the contributors use the input format described in Appendix A. As usual in CHARM++, because contributors are almost always parallel objects themselves, there can be several contributors per processor, or none at all on a particular processor.

In step 2, the local `collideMgr` actually accumulates the triangles to be sent off, then finally sends all the triangles for a voxel at once. This local accumulation is much more efficient than sending a separate message for each triangle, and allows us to use the indexed triangle representation. Splitting up indexed triangle lists efficiently is the subject of Appendix A.1.

The message sends in step 2 actually *create* the voxels, if they haven't already been created. By default, the creation happens on the sending machine– this makes all future communication between this `collideMgr` and the voxel efficient. If the triangles contributed on a processor are clustered well, most will never be sent over the network.

The synchronization in step 3 is needed because voxels may receive objects from several processors. Since any processor can send a triangle to any voxel, "Have all triangles been delivered?" is a global property, obtainable only via a global operation. Recall that in CHARM++, however, other local objects will automatically use the CPU during this single synchronization.

After the synchronization, each voxel runs a separate serial collision detection algorithm and reports the results.

3.3.2 Parallel Details

The CHARM++ parallel array framework used above abstracts away several interesting implementation details.

Consider step 2 of the parallel algorithm above. Triangles destined for voxel $(17, 9, -4)$ can be given to the `collideMgr` on any processor. Somehow, each processor must determine if that voxel already exists somewhere on the parallel machine. If so, the triangles need to be sent to the voxel. If not, a new voxel must be created, ideally on the same processor.

Further, triangles destined for the same, nonexistent voxel may be delivered to two different processors at the same time. Exactly one of the processors must create a new voxel; and the other's triangles must be sent there.

It's easy to imagine using a centralized voxel registry to map grid locations to processors and serialize the creation race condition. However, a centralized registry is is not scalable and would quickly become a serial bottleneck. The natural solution, then, is to use a distributed registry.

This is the approach used by the parallel array framework. To deliver a message to an unknown grid location, it computes a "home" processor[2] for that location via a simple hash function. This home processor will either inform the sender of the voxel's location, or authorize the sender to create a new local voxel. Home processors thus act as a distributed registry to keep track of which voxels exist, and where. Processors cache the location of recently referenced voxels, eliminating the overhead of contacting the home processor for repeated communication.

Home processors also provide an efficient, distributed means to support migrating array elements. The array framework also properly supports broadcasts, used in step 4 above; as well as reductions, used in step 5. The details of how to support these operations with ongoing migrations are presented by the author in [18].

3.4 Load Balancing

The Charm++ automatic load balancer [3] measures the load and communication patterns of each parallel object. It then uses a load balancing "strategy"[3] to determine where each object should be migrated for optimal performance. Performing load balancing automatically at run-time enables an application to react quickly to irregularity in the problem structure.

In fact, since collision detection is often only a small part of a much larger parallel program, perfect load balance for the collision library is actually not necessary. As long as the program is structured to allow the execution of different libraries to interleave (as is always the case in Charm++), different processors may have different amounts of collision work yet all have the same amount of work overall.

[2]The same concept is used in most distributed shared memory cache coherence schemes

[3]Several built-in strategies exist; or a custom strategy can be written

Consider the work done by a parallel machine during an unsynchronized period– that is, between two global synchronizations. In symbols, let A_i and B_i represent the amount of work needed by two libraries A and B on processor i. Then for load balance we require that each processor have the same amount of work overall, i.e.:

$$\forall_{i,j}\ A_i + B_i = A_j + B_j \tag{3.1}$$

This is often ensured by requiring that every library impose the same amount of work on every processor, or:

$$\forall_{i,j}\ A_i = A_j \wedge B_i = B_j \tag{3.2}$$

It is clear that 3.2 implies 3.1, but 3.1 does not imply 3.2. That is, library load balance is sufficient but not necessary for global load balance. Thus the usual parallel programming approach of load balancing every library is not necessary. A load balancer can use the extra degrees of freedom obtained by replacing 3.2 with 3.1 to minimize communication overhead.

For the voxel method, the CHARM++ automatic load balancer is free to migrate both the contributors, which do work before the synchronization; and the voxels, which do work afterwards. Hence the load balancer can balance the collision algorithm in the context of the larger program.

4 Performance

The voxel algorithm presented in this work has excellent theoretical as well as practical performance.

4.1 Theoretical Performance

Let n denote the number of objects. If the average number of voxels spanned by each object is bounded, the voxel method collides $O(n)$ objects and hence imposes no asymptotic overhead on the intra-voxel subscheme.

If, further, the intra-voxel subscheme is linear time, then the voxel method is also linear time overall. If the subscheme is not linear; but the number of objects in each voxel is bounded, the subscheme runs in constant time and the voxel method is again linear time.

In the parallel implementation, after a global synchronization step each voxel forms an independent serial subproblem. Thus assuming enough voxels that good load balance is possible[1] the time required is $O(n/p + \lg p)$. For large enough n, the first term dominates and the time required is $O(n/p)$.

4.2 Serial Performance

The 871,414 triangle dragon model of Figure 4.1 presented in [5] took 3.132 seconds on an Origin 2000 [2].

The remaining tests use a much simpler model– a tessellated plane perturbed by small harmonic waves. This simple model is easy to scale to any desired number of triangles; while retaining much of the character of an actual surface. No collisions actually occur in the simple model.

Figure 4.2 shows the wall-clock time per complete collision detection for a range of different numbers of triangles on a Linux PC [3]. The smallest figure shown is 1,024 triangles, which take 1.4 milliseconds or 1.4 microseconds per triangle. The largest figure is 1,048,576 triangles, which take 2.1 seconds or 2.1 microseconds per triangle. The much larger dataset cannot fit entirely in cache and is hence slightly slower.

[1] The degree of parallelism is limited by the number of voxels.

[2] 195MHz MIPS R10000, IRIX 6.5

[3] 400MHz AMD K6-3, Linux 2.4.2

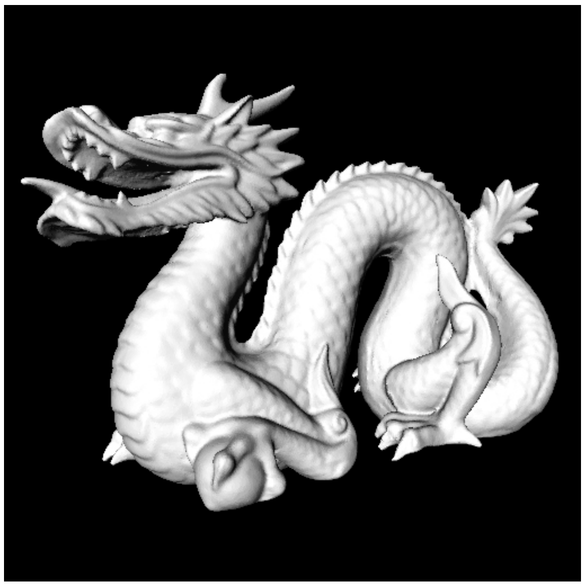

Figure 4.1: Dragon model.

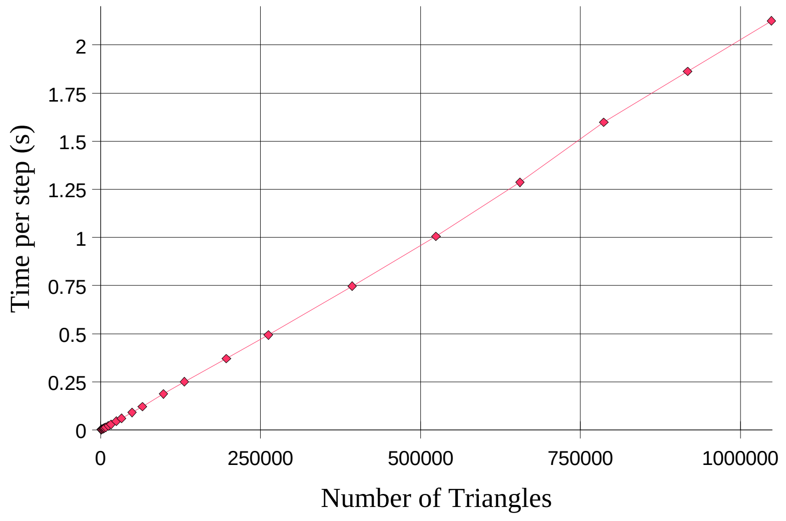

Figure 4.2: Time per step for serial benchmark

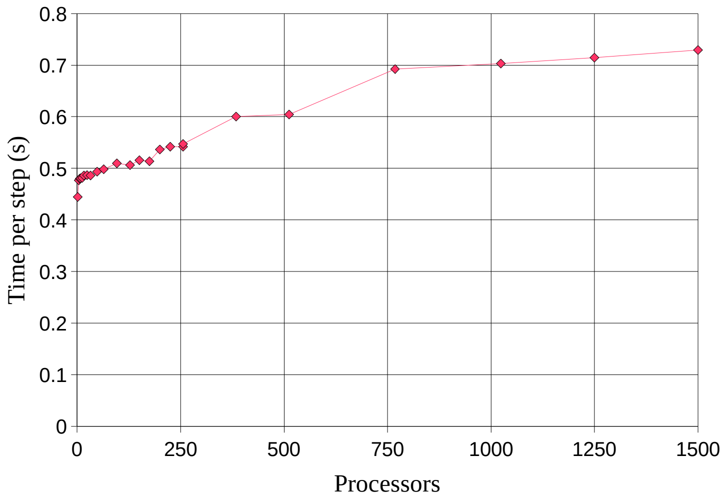

Figure 4.3: Time per step for a scaling parallel benchmark, with a fixed 65,536 polygons per processor.

This serial result is competitive with the figure of 2 microseconds per triangle given for the serial algorithm of Gottschalk et al. in [9]. Further, the observed performance of the voxel algorithm is indeed linear in the number of triangles.

4.3 Parallel Performance

On ASCI Red, an Intel Paragon machine, we ran a simple scaling benchmark with 65,536 polygons per processor. The wall-clock time per timestep for various numbers of processors is shown in Figure 4.3. A program with perfect speedup would have a constant time per step. Instead, we see a slow, logarithmic rise in the time per step due to the $O(\lg p)$ synchronization overhead.

The smallest run shown, 65,536 triangles on a single processor, takes 0.44 seconds per step. The largest run shown, 65,536 triangles on each of 1,500 processors or 98.3 million triangles, takes 0.73 seconds per step, for a speedup of 915 or a parallel efficiency of 60 percent.

The observed parallel performance is indeed excellent. It also compares quite favorably with the result of 1 second per timestep for 8,000 objects per processor for the parallel RCB scheme described in [22].

The parallel implementation also scales down for smaller models and fast response time, such as for interactive applications. 32 processors of a 195 MHz Origin2000 system can handle 300,000 triangles at the good interactive rate of 30 milliseconds per step.

5 Conclusions and Future Work

We have demonstrated a parallel collision detection algorithm based on regular space subdivision. We have implemented the algorithm in CHARM++ and extensively analyzed its performance. The results are theoretically and practically quite competitive with published work.

5.1 Future Work

The most fundamental limitation of the voxel method is that all the voxels are the same size and orientation, which limits performance for extremely small or large objects. A promising area of future research is to use a nonuniform grid, ideally determined automatically from the object density. A multi-level grid such as an octree would be an easy way to achieve this end, although it may be difficult to maintain $O(n)$ performance. A more effective method might be to use the local object size or density, possibly from the previous step, to efficiently adjust the grid resolution in a spatially dependent manner. A per-axis application of this method is shown in Figure 5.1.

A much simpler goal is automatic determination of the ideal voxel size. Voxels that are too small have too much overhead; voxels that are too large do not create enough parallelism and lump together too many objects. The

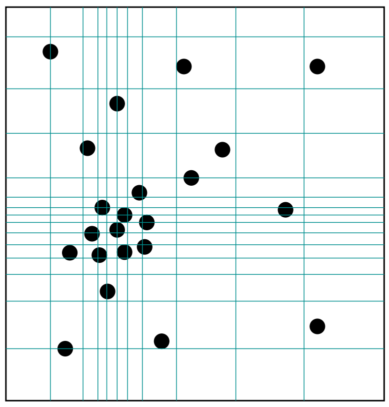

Figure 5.1: Objects could warp the grid in a relativistic fashion.

tradeoff should be better analyzed and, if possible, automatic.

Voxels need not form a regular grid at all. There are several other regular and irregular ways to tile 3D space. A less angular grid cell, with a higher volume to surface area than a cube, would be less likely to bring together objects that will never collide.

Although the voxel algorithm applies to arbitrary objects, the implementation is currently limited to triangles. A useful extension would be to include more complex primitives. The implementation could also make much better use of temporal and spatial coherence, as suggested in [20] and [19].

The implementation currently requires one barrier operation per collision, which could become a bottleneck for small problems on large machines. Some sort of global operation is needed to ensure all triangles have been delivered; but a systolic approach, continually summing the current send and receive counts, might be more proactive.

We currently only determine which objects intersect. This is sufficient for some applications, such as motion planning, clearance checking, and many graphics applications. However, a more complete solution would go on to determine the entire set of penetrating points, the penetration distance, and the penetration direction for each point. Kawachi and Suzuki [14] use a "discrete closest feature list" to efficiently determine these quantities via an algorithm similar to the voxel method.

A Input Format and Manipulation

The input to any collision detection algorithm is a set of objects. The current implementation is restricted to triangles[1].

The usual mathematical description of a triangle is simply three noncollinear points. Rather than this, we use the indexed point list representation, common in computer graphics, shown in Figure A.1. In this representation, a triangle's vertices are actually indices into a shared list of points.

Because isolated triangles are rare, most points are shared between several triangles. In fact, most geometric models have, in total, about half as many points as triangles! Thus using the indexed representation often saves a substantial amount of memory[2], which tends to improve cache utilization and hence overall performance.

The indexed representation also allows us to store an extra property of triangles, the *prio* field. This is the "collision priority", which is eventually used by the final collision detection scheme. To prevent duplicate collision checks, collision tests are only performed against objects with a lower priority. If every object has a different priority, this simply avoids duplicate collision checks.

Because objects with the same priority are never collided, we can also use the priority field to prune off unnecessary checks. For example, if we know there are no self-intersections in a rigid model, we can prevent the algorithm from ever colliding model triangles against one another by giving all the model's triangles the same priority. Or, if we aren't interested in collisions among "static" entities, such as trees and buildings, we can give

[1]It is common to abuse terminology by using "triangle" and "polygon" interchangeably.

[2]Memory usage is important because many engineering applications already demand enormous amounts of storage

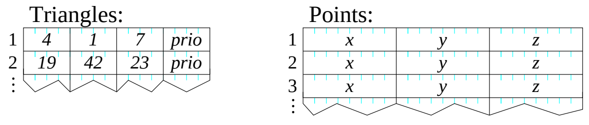

Figure A.1: The input data is a list of triangles and an array of points.

all static entities the same priority. The RCB algorithm used includes an early-exit test if all a region's objects have the same priority.

Finally, note that the input consists of two arrays, each of which contain data of the same type. This makes bindings for other languages, such as C or FORTRAN, much simpler. In fact, the only difference between accepting C and FORTRAN triangles is that the point indices start at 0 in C, and 1 in FORTRAN.

A.1 Efficiently Splitting Indexed Triangle Lists

The task of the `collideMgr` object is to split these indexed triangle lists into a single message for each voxel. However, this task is rather difficult to perform efficiently, because we must somehow collect all the points referenced by a each set of triangles.

Let there be p total (unsplit) points, v voxels, and t triangles per voxel. Since the number of voxels can be large, and triangles should be distributed among voxels relatively evenly, p is normally much larger than t.

The brute-force solution– for each point, check to see if any relevant triangle references it– is clearly $O(pt)$. Since the process will be repeated once for every voxel, the brute force approach is $O(vpt)$.

We of course can do better by first marking the points used by each triangle, then collecting all the marked points. If we mark points using a table, this approach is $O(t+p)$ for each voxel, because it takes $O(p)$ time to clear all the marks initially, $O(t)$ time to mark some of the points, and then $O(p)$ again to collect the marked points. Repeating for every voxel gives $O(v(t+p))$ overall.

Theoretically, we could eliminate the factor of p by marking each voxel's $O(t)$ referenced points with a hashtable. In practice, the large constant factor associated with a hashtable access[3] makes this approach unattractive.

A much better solution is to mark all referenced points in a linked array structure. Each entry of this array indicates whether the corresponding point has been marked, and if so, links to the next marked point. Then marking a new point takes (a very small) constant time; and all referenced points can be collected by following the links, in $O(t)$ time. We can also clear all the marks by following the links. Thus the only $O(p)$ operation needed is initialization, for a total run time across all voxels of $O(vt+p)$. Since both v and p can be large, this is a substantial improvement.

[3]Especially since there will be three accesses per triangle

B Efficient Gridding

The inner loop of the voxel method outlined in this work computes the grid locations spanned by an object, and then inserts the object into the object lists at each grid location. For simplicity, we compute the grid locations based on the object bounding box. Hence one extremely common operation is to convert the coordinate of each face of a bounding box, a floating-point number, into a grid location, an integer. However, on modern architectures, converting floating-point numbers to integers is often slow.

In certain situations, a novel but simple technique can be used to dramatically speed up floating-point to integer conversions. In particular, given a double-precision source and integer destination array, we may use either the first or the second two C statements below:

```
dest[i] = (int)floor(src[i]);  /* Conventional version*/
float f = (float)(src[i] + convert_offset);
dest[i] = *(int32 *)&f;    /* Optimized version*/
```

The reason is that an IEEE single precision floating-point number's mantissa field overlaps the low bits of an integer, as shown in Figure B.1. If the grid size is a power of two, we can convert a coordinate into a grid location by shifting and extracting out the appropriate bits of the coordinate.

Floating-point numbers are always normalized into the form $2^m 1.xxxx_2$, so we can *shift* the mantissa of a floating point number by simply adding a constant. For example, given a floating-point coordinate with binary representation $xxxx.yyyy_2$, after adding $2^{23}1.1000_2$, we get

$$2^{23}1.\underbrace{1000000000000000000xxxx}_{23\text{bits}}$$

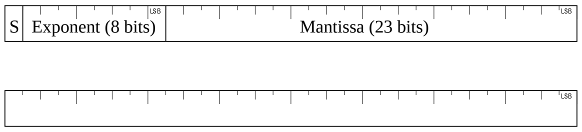

Figure B.1: An IEEE single precision floating-point number and a 32-bit integer

By adding a constant, we have shifted the grid location *xxxx* into the low bits, and by rounding to single precision we have removed the excess bits *yyyy* that represent the sub-gridcell location.

The resulting floating-point number can be compared, incremented, and decremented as if an integer. The IEEE exponent field corrupts the high bits of this integer, but these can easily be subtracted off. Since the origin of the grid coordinates is arbitrary anyway, we leave the exponent. The mantissa is only 23 bits, but this is still enough to represent 8 million grid cells along each axis, or 10^{20} cells total. Converting a double-precision floating point number to a 64-bit integer in the same way would yield a 52-bit grid index.

By adding a carry-under protection bit to the constant, this method can accept both positive and negative input coordinates. By subtracting the coordinate origin, we can accomplish a shifting as well as scaling. We can also scale the constant to extract out a different set of bits, simulating any power-of-two grid size. Finally, we must compensate for IEEE rounding, which rounds to the nearest bitpattern rather than truncating. The ideal constant for this method thus has the form:

$$(\underbrace{1.5}_{\text{carryprotect}} * \overbrace{2^{23}}^{\text{shift}} - \underbrace{0.5}_{\text{round}}) * gridsize - origin$$

Thus given a power-of-two grid size and origin, the optimized C code to compute the conversion constant and then convert a floating point value `src` to an integer grid cell index `dest` is as follows.

```
double convert_offset = (1.5*(1<<23)-0.5)*gridsize-origin;
float f = (float)(src + convert_offset);
int dest = *(int32 *)&f;
```

Of course, the same method can be applied in any language that allows us to quickly interpret the bits of one data type as bits of another data type.

The table on the next page shows the time per floating point to integer conversion for both the conventional and optimized approaches. The performance difference between the two is dramatic– a factor of 2 to 40, depending on the architecture. Because mapping one bounding box onto a grid requires six such conversions, the total savings by using the optimized method is a microsecond or two per object– a significant speedup.

Machine	*Conventional*[1]	*Optimized*[1]
195 MHz MIPS R10000[2]	109 ns	21 ns
300 MHz SPARC Ultra 10[3]	245 ns	30 ns
332 MHz PowerPC 604e[4]	281 ns	127 ns
500 MHz Pentium 3 Xenon[5]	236 ns	8 ns
400 MHz AMD K6-3[6]	368 ns	64 ns
300 MHz AMD K6-3[7]	491 ns	84 ns
240 MHz PA-RISC 8200[8]	648 ns	15 ns

Table B.1: Comparing conventional and optimized conversion times.

[1]Computed as wall clock time for 100 million conversions divided by 100 million

[2]IRIX64 6.5, gcc 2.95.2 -O3

[3]Solaris 2.6, gcc 2.95.2 -O3

[4]AIX for IBM SP, gcc 2.95.2 -O3/VisualAge AIX 5

[5]Linux 2.4.2, gcc 2.96 -O3

[6]Linux 2.4.2, egcs-2.91.66 -O3

[7]Linux 2.2.18, egcs-2.91.66 -O3

[8]HP-UX 10.20, gcc 2.95.2 -O3

References

[1] S. Bandi and D. Thalmann. An adaptive spatial subdivision of the object space for fast collision of animated rigid bodies. In *Proceedings of Eurographics '95*, pages 259–270, August 1995. http://ligwww.epfl.ch/ thalmann/.

[2] J. Boyse. Interference detection among solids and surfaces. *Communications of the ACM*, 22, number 1:3–9, January 1979.

[3] R. Brunner and L. Kalé. Adapting to load on workstation clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112, February 1999.

[4] S. Cameron. Approximation hierarchies and s-bounds. In *Proceedings of Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 129–137, 1991.

[5] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *Proceedings of ACM Siggraph '96*, pages 303–312, 1996.

[6] R. Farouki, C Neff, and M. O'Connor. Automatic parsing of degnerate quadric-surface intersections. *ACM Transactions on Graphics*, 8:174–203, 1989.

[7] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Proceedings of ACM Siggraph*, pages 124–133, 1980.

[8] E. Gilbert, D. Johnson, and S. Keerthi. A fast procedure for computing the distance between objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, RA-4:193–203, 1988.

[9] S. Gottschalk, M. Lin, and D. Manocha. Obb-tree: A heirarchical structure for rapid interference detection. In *Proceedings of ACM Siggraph '96*, pages 171–180, 1996.

[10] P. Hubbard. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, Brown University, 1994.

[11] P. Jiménez, F. Thomas, and C. Torras. Collision detection algorithms for motion planning. In *Robot Motion Planning and Control*. Springer, 1998.

[12] L. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In *Parallel Programming using C++*, pages 175–213. 1996. http://charm.cs.uiuc.edu/.

[13] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[14] K. Kawachi and H. Suzuki. Distance computation between non-convex polyhedra at short range based on discrete voronoi regions. In *Proceedings of Geometric Modeling and Procesing*, pages 123–128, Hong Kong, 2000. IEEE.

[15] J. Klosowski, M. Held, J. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volumes of k-dops. In *Siggraph '96 Visual Proceedings*, page 151, 1996.

[16] S. Krishnan, A. Pattekar, M. Lin, and D. Manocha. A higher order bounding volume for fast proximity queries. In *Proceedings of Third International Workshop on Algorithmic Foundations of Robotics*, 1998.

[17] J. Lane and R. Reisenfeld. A theoretical development for the computer generation and display of piecewise polynomial surfaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(1):150–159, 1980.

[18] O. Lawlor and L. Kalé. Supporting dynamic parallel object arrays. In *Proceedings of International Symposium on Computing in Object-oriented Parallel Environments*, Stanford, CA, Jun 2001. http://charm.cs.uiuc.edu/papers/ArrayMigISCOPE01.html.

[19] M. Lin. *Efficient Collision Detection for Animation and Robotics.* PhD thesis, University of California, Berkeley, 1993.

[20] M. Lin and J. Canny. Efficient algorithms for incremental distance computation. *IEEE Conference on Robotics and Automation*, pages 1008–1014, 1991.

[21] M. Lin and S. Gottschalk. Collision detection between geometric models: A survey. In *Proceedings of IMA Conference on Mathematics of Surfaces*, 1998. http://www.cs.unc.edu/ dm/collision.html.

[22] S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, and D. Gardner. Transient dynamics simulations: Parallel algorithms for contact detection and smoothed particle hydrodynamics. In *Proceedings of Supercomputing*, 1996.

[23] S. Suri, P. Hubbard, and J. Hughes. Analyzing bounding boxes for object intersections. *ACM Transactions on Graphics*, 18 no. 3:257–277, July 1999.

[24] G. Turk. Interactive collision detection for molecular graphics. Master's thesis, University of North Carolina at Chapel Hill, 1989.

[25] Y. Zhou and S. Suri. Analysis of a bounding box heuristic for object intersection. *Journal of the ACM*, 46 no. 6:833–857, November 1999.

[26] M. Zyda, W. Osborne, J. Monahan, and D. Pratt. Npsnet: Real-time collision detection and response. *Journal of Visualization and Computer Animation*, 4, number 1:13–24, 1993.