1. Introduction

In this thesis, we present a narrow-phase algorithm for efficient distance computation between two flexible objects. The algorithm will allow for arbitrary deformations on the modeled objects. It will also allow for approximate distance computation with run-time settable error rates in order to provide a continuum of information between Boolean collision checking and exact distance computation. Moreover, this algorithm is designed to provide real-time performance on useful-sized models.

Collision detection algorithms are an essential component of computer-based physical simulations. In nature, objects naturally prevent themselves from occupying the same volume (through the electromagnetic force). Computer simulations of physical objects have no such natural behavior, and thus algorithms to determine explicitly the collision state of two bodies are required. Moreover, accurate simulations also require collision response algorithms. These determine the simulated behavior of the objects to prevent physical overlap. Most often, the collision response is simply a repulsive force that avoids continued collision. In this thesis, we discuss only collision detection and its close relative, distance computation.

Distance computation algorithms are very similar to collision detection algorithms. While collision detection algorithms return a Boolean true/false in response to a collision query, distance checkers return a number representing the minimum distance separating two bodies. It is easy to observe that a distance-checking algorithm can be used to implement collision detection – it would simply return true if the distance is zero. Quinlan observed that a continuum of behavior exists between pure collision detection and pure distance computation [Qui94].

The collision detection problem is usually posed as follows: "Given a list of objects, determine which pairs, if any, are in collision." This definition implies that collision detection should be performed in two phases. Typically they are known as the broad phase and the narrow phase. The broad phase is responsible for determining which pairs of objects should be more closely inspected for pair-wise collisions. After pairs have been identified as needing further processing, the narrow phase determines if a specific pair of object is indeed in collision. This thesis will deal with only the narrow phase problem.

2. Previous Work

Collision detection algorithms for polygonal objects can roughly be classified in two distinct groups: feature-based algorithms and hierarchical bounding volume algorithms. Each of the two methods has distinct advantages and drawbacks. Feature-based algorithms are better able to exploit temporal coherence in the model. This means that small motions of the simulated objects will require minimal correction to determine the new closest feature pairs. However, to exploit coherence, the models are generally required to be convex or built of a small number of convex pieces. This is a significant limitation for the simulation of flexible objects. Hierarchical bounding volume algorithms require little of the underlying geometric models of the objects to be simulated. The most pathologically non-convex objects, polygon soups, are easily handled by hierarchical bounding volume approaches. Since the geometries allowed by bounding volume algorithms do not easily lend themselves to coherence-based speedups, incremental collision checks are just as costly as the first.

Feature-based algorithms represent objects as a collection of features -- points, line segments, and facets. This construction leads to a topological map of the object. Moreover, they typically maintain that this collection of features is the convex hull of the object. This

assumption leads directly to the coherence-based speedups that make these algorithms attractive. The convexity assumption allows algorithms to traverse features in search of globally optimal closest pairs without being trapped in local minima. One of the first feature-based algorithms was that presented in [Bar90]. The Lin-Canny closest features algorithm is a considerably more sophisticated feature based algorithm that allows distance computation between the polyhedra as well. V-Clip [Mir98] is an enhancement of Lin-Canny that eliminates numerical instabilities, increases efficiency, and allows for the computation of penetration distance for intersecting polyhedra.

The other main class of collision and distance algorithms uses hierarchies of bounding volumes to limit the search space. Hierarchies were formally introduced in [DK90] where the asymptotic run time was found to be $O(\log^2 n)$. In previous work these volumes have been axis aligned bounding boxes (AABBs) [CLMP95], oriented bounding boxes (OBBs) [GLM96], spheres [Qui94, Hub96], and even sphere shells [KPLM98]. As can be expected, the bounding box approaches work better for "flat" objects while spheres work better for rounded objects. Moreover, the OBBs and sphere shells trade "closer fits" to the objects, and hence a smaller search space, for a more complicated distance operation.

Some previous works exists in the area of collision detection for flexible objects [BW92,]. However, these algorithms were designed for particular simulation domains and were not intended as general-purpose algorithms. This is by no means an exhaustive survey of previous work. A more complete survey can be found in [Lin99].

3. Basic Algorithm

The algorithm presented here has several constraints that influence its design. Most importantly, this collision detector/distance checker must handle flexible objects well.

Moreover, for generality, these flexible objects might be pathologically non-convex. This restriction eliminates the feature-based trackers from consideration. This limits the selection to the bounding volume methods. Since the underlying model will be changing – often frequently – the bounding volume hierarchy will have to be changed frequently. Spheres require few computations to fix the hierarchy. The basic algorithm we present to solve this problem is similar to that of Quinlan. As in that method we use a sphere tree as a bounding method.

Unlike Quinlan, we do not tile each triangle with many equal sized spheres. Our algorithm assigns each triangle in the model one, variable sized, leaf sphere. This design decision was made to simplify the process of adjusting the hierarchy when a triangle moves. Unlike this algorithm, Quinlan's does not need to keep track of triangles after they have been tessellated into spheres. Since we do not allow for structural changes to be made to the hierarchy after its construction, we must also assure that each triangle's tessellation does not change (i.e., no spheres may be added or deleted from a triangle's representation). Tiling triangles with many equal sized spheres does not make sense in this context since the spheres' sizes would have to change if the triangle's size or shape changes. Under our semantics, if a triangle changes, the tree's structure will remain the same and the change will be propagated through only one leaf sphere.

In this algorithm the assumption is made that each hierarchy sphere node will contain both of its children spheres. This rule makes it local and simple to update a particular sphere if its children have been modified. This is different from Quinlan's rule that a sphere node need only contain its descendant triangles. It would be very difficult to efficiently maintain this quality in all the sphere nodes as the triangles are modified.

3.1. Initial Tree Construction

Perhaps the most expensive operation in this algorithm is the initial construction of the sphere tree. It is important that the sphere tree meet some important criteria for collision or distance queries to be efficient. The tree must be approximately balanced combinatorially to assure that the triangle primitives are $O(lg\ n)$ deep. This is necessary to make the pruning operation work more effectively. The tree should also encode as much geometric information as possible. Specifically this means that a nodes two children should be as geometrically as distinct as possible. If the children have much spatial overlap, then descending in the tree will not result in much pruning, the operation that provides this algorithm's speed.

To build a tree that satisfies these criteria we sort the triangles in a top down fashion. The recursive procedure operates as follows: split the triangles into two, geometrically separated and equal sized, groups; recursively build trees for each of these two groups; call our children the spheres surrounding each of these sub-trees; and then our sphere is the sphere surrounding both children spheres. The most important step is to effectively divide the triangle list into two groups, as it is responsible for satisfying the two criteria. This step need not be particularly efficient, as this operation is performed only once. If the lists are not about equal, the resultant tree will not be combinatorially balanced. If the lists are not geometrically separated, the tree will not encode the geometry properly.

3.2. Collision Query

Like the tree construction, the distance computation is a recursive algorithm. Given two spheres, the distance procedure will determine the distance separating the underlying models. In collision detection mode, it operates as follows: if the spheres are non-intersecting, it reports that no collision occurs; otherwise it recurses in checking one sphere

against each of the children of the other sphere; in the base case of triangle against triangle, the collision state is determined directly. As in Quinlan, we use the heuristic of splitting the larger sphere when recursion is necessary.

3.3. Tree Maintenance

Since the primary goal of this thesis is to have a collision detection algorithm that can handle deformations, we must have a method to handle these changes. The simplest way to handle these is propagate the changes from the leaf to the root every time a deformation occurs. Only the chain of spheres from the root of the tree to the sphere enclosing the triangle is affected by the change. The naïve propagation operation for any node in the tree is quite simple: resize our sphere to fit our children (either two spheres or a triangle), than send a message to our parent to do the same thing. This, however, is not the optimal strategy. A better method is discussed in section 4.1.

The naïve version of tree maintenance results in location neutral costs for deformation. That means that the cost of maintaining the hierarchy is directly proportional to only the number of nodes that are deformed, assuming the tree is combinatorially balanced. This is simply because a deformation results in a traversal of the path from the leaf to the root. If the tree is balanced, deforming any node will have cost proportional to the depth of the tree. The enhanced propagation method results in different cost characteristics. The pathological case for deformation is simultaneous deformation to all nodes in the model. Such a deformation will result in $O(n \lg n)$ sphere adjustments, roughly the same cost of building the model from scratch.

4. Enhancements

Although the basic algorithm will correctly compute the distance between two time-varying flexible objects, a couple of core optimizations help speed computation considerably. The basic algorithm's amortized performance deteriorates rapidly as more changes are made to the model between distance queries. The following modifications significantly improve its running time by reducing the number of required updates.

4.1. Batch processing of changes

In the basic algorithm described above, every time a vertex in the model is modified, changes will propagate all the way to the root of the tree. If two nearby triangles are changed, each will result in a full propagation to the root sphere. Also, if the same triangle is modified twice, this propagation will occur twice, even if no intermediate distance queries are made. These two problems can be solved by careful batch processing of deformations of the object.

A naïve solution would be simply to trap all deformations and apply only the final modification to any particular triangle when a distance query is made. This approach will solve the second problem: multiple changes made to a single triangle will result in only one propagation of the change. However, changes made to two triangles may still result in overlaps of their two propagation chains.

A better solution is to order the propagations by depth in the tree. The basic data structure required for this is a priority queue where deeper nodes are extracted first. We can easily add an extra attribute variable to each node in the tree to cache its depth. Consider the following pseudocode to handle propagation:

```
while notempty(pq)
     node = extract(pq)
     fix(node)
```

```
        insert(pq, parent(node))
    end
```

Instead of following a continuous path from a node to the root, the algorithm explores completely each level of the tree, deepest first. Only nodes whose children have been modified are added to the tree. This results in no unnecessary repeated updating. Moreover, this method has another advantage. If we make the assumption that neighbors at a particular level will be near each other in the system's memory, the computer's cache will better exploit the locality of memory references. However, to assure this condition, the hierarchy must be built more carefully.

When using this method of deformation propagation, certain types of deformations are more preferable than others. Specifically, this method performs best when deformations are made to a localized area in the tree (i.e., a local geometric area). This method results in smaller benefits as deformations are spread randomly throughout the geometry. However, this is not very likely in practice since changes are generally made in localized regions. As in the naïve case, the worst scenario is if all vertices are deformed. Unlike that method, the enhancement will result in a cost of only $O(n)$, since each sphere will be visited only once.

4.2. Loose bounding

If many of the triangles are moving simultaneously, the update propagation will be very time consuming. Moreover, models in many domains have small regions of triangles that move frequently. The idea of loose bounding attempts to address the frequent propagation operations that result from these types of models. The intuition behind loose bounding is that if we let a sphere in the tree be slightly larger than necessary, it will be able to absorb small variations of its children's positions without varying itself.

Unlike batch processing of changes, loose bounding does not always increase the performance of the distance computation algorithm. This is because of the algorithm's reliance upon accurate information from spheres to prune the search. Loose bounding can only have a negative impact on actual distance queries. Batch processing, on the other hand, has no effects on the queries, and can only have positive effects on the updating of the hierarchy. Another drawback is that loose bounding requires more parameters to be tuned for acceptable, let alone optimal performance.

5. Results

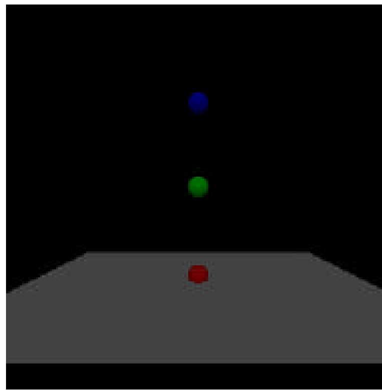
In this section we discuss the behavior and performance of the implementation of the described algorithm. The algorithm was implemented in C++, and was compiled with maximum optimization on Microsoft Visual C++. All data was gathered on a 400 MHz Pentium II, with 256 MB memory running Windows 2000. Since the algorithm was designed to decouple deformations from collision or distance queries, the analyses will be divided in two sections.

5.1. Query Performance

Our primary metric for this algorithm's query performance is the average time required to execute a given query for two objects a given distance apart. This is a reasonable basis for examination since, as a bounding volume algorithm, the number of spheres that may be pruned increases as the objects are separated, thereby quickening the collision or approximate distance query.

For these performance tests, we sought to mimic situations commonly encountered by collision detection algorithms. The test includes one fixed object and one moveable object, as

this algorithm only deals with the narrow-phase of collision detection. The fixed object, a flat and square mesh, is tessellated with 8192 triangles, and is 8 by 8 units. This is meant to represent a face of a large obstacle or wall in the workspace. The moveable object is a ball that is tessellated with 1024 triangles and is 2 units in diameter. All tests involve the ball being moved from 64 units separation to one unit penetration (minimum-extraction distance), at the center of the mesh. The following image is of the scene used to measure query performance: the red ball is in penetration by 0.5 units, the green ball has a separation distance of 8 units, and the blue ball has a separation distance of 16 units.

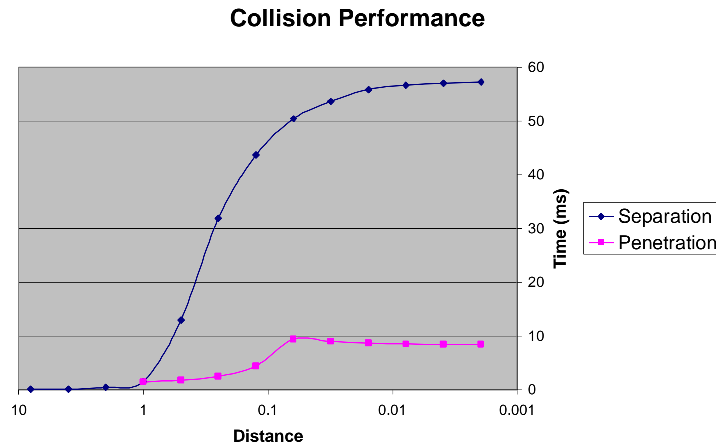


5.1.1. Collision Performance

The first test is of pure collision performance. Here we only expected the algorithm to accurately determine whether or not the two objects were in collision. As can be expected, when the objects are far apart, the query is extremely fast, on the order of tenths of milliseconds. However, as the objects are brought closer together, the time required to determine that the objects still were separated grew quickly to an asymptote of just under 60 milliseconds. Again, this is to be expected since more of the spheres must have been examined to rule out a possible collision.

Once the objects were in collision, the query time dropped sharply, to under 10 milliseconds. Since the algorithm only needs a witness pair of features to the collision, this

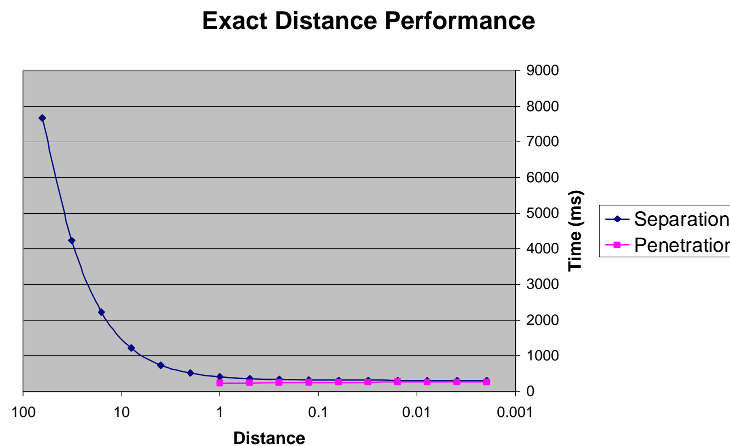
should be fast. As the objects are brought further into collision, it becomes even easier to find such a pair of witnesses, and consequently the query time follows further.



5.1.2. Exact Distance Performance

At the other end of the spectrum from collision checks are exact distance queries. In these types of events the algorithm is expected to return the exact distance separating two objects as well as a pair of features that yields this distance. This is understandably a much harder query to execute than a collision check. Consequently the behavior of the algorithm is considerably different for this type of query. Rather than get harder as the two objects approach, performance actually increases dramatically as separation distance decreases for a distance operation. This property can be explained as follows: as the objects are brought closer together, it is easer to determine which features from each object must be considered to determine the minimum distance. Specifically, as the two objects are separated any pair of features is roughly the same distance as any other pair of features, and therefore it becomes difficult to prune the search. It can be expected that the distance query time will be limited above by an asymptote representing the time required to compare all possible pairs of features as separation increases.

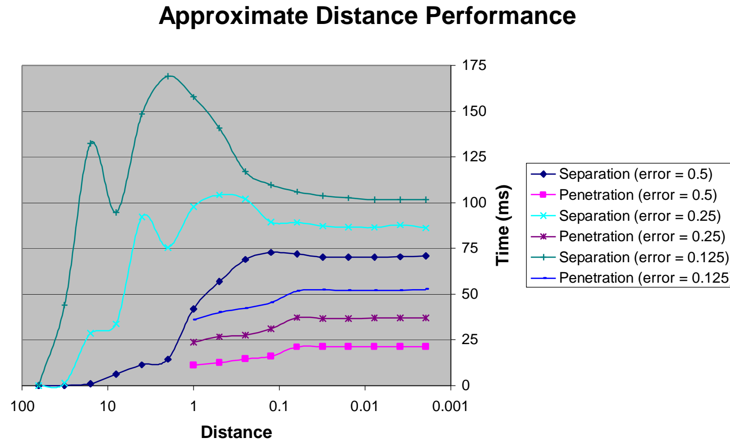
As in the case of collision detection, queries when the objects are in collision were reported quickly. These queries were not as fast as in the collision detection case since the search tree was not pruned as quickly. The times ranged from 260 milliseconds for slight penetration to 238 milliseconds for the deepest possible penetration. The quickest queries for separation took about 310 milliseconds.



5.1.3. Approximate Distance Performance

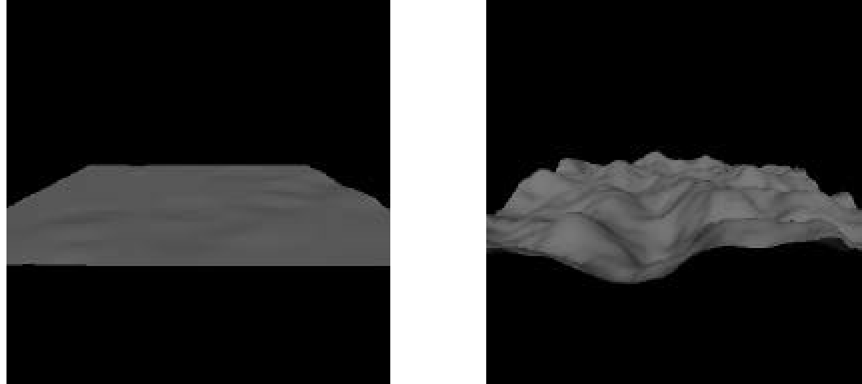
The final query test examined how the algorithm performed when it was allowed to report a distance within a certain error bound. Specifically, if the algorithm was asked to return the true distance *d* with error *e*, it could respond with a distance *d'* in the range [*d*, *d*(1+*e*)], so that *d* is a lower bound for *d'*. Moreover, collisions will always be properly reported. By allowing for errors in the reported distance, the queries can be executed more quickly since the search tree can be pruned more easily.

As can be expected, approximate queries behave as a hybrid of collision and exact distance queries. Similar to collision checks, approximate distance computations are extremely quick when the objects are greatly separated, and grow as the objects are brought closer together. However, just as in exact distance computations, approximate distance queries decrease in cost as the objects are brought yet closer.

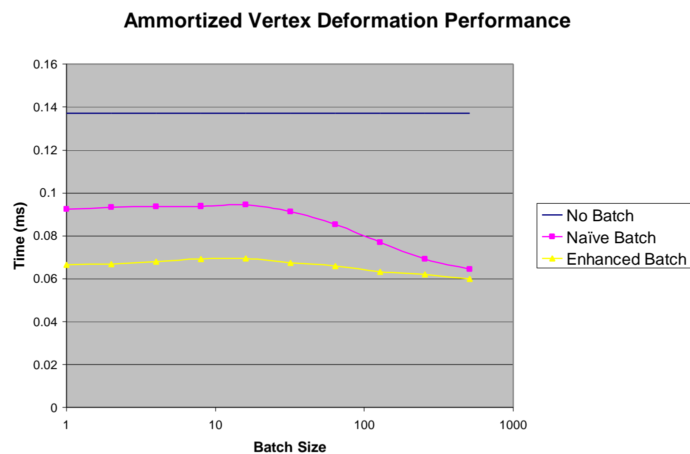


5.2. Deformation Propagation Performance

Besides knowing how long it takes to perform a collision check or distance computation, it is also important to know how quickly the model can be altered to take a deformation into account. Three main tests were performed to evaluate the various propagation settings: no batch processing, naïve batch processing, and enhanced batch processing. To compare these three modes we identify the time it takes each algorithm to perform a specified battery of deformations. In order to accurately measure the relative performances, each test is composed of the exact same sequence of deformations. We start with a flat 100x100 grid, with each square split into two triangles. Then we select a center of deformation and radius of deformation between one and ten. Next all vertices within the specified radius of the specified center are deformed a distance proportional to its distance from the center. This process is repeated a given number of times. The results are reported as the amortized performance per deformed vertex: we divide the total time for the battery by the number of vertices in the battery to yield the time per vertex. The following images illustrate the deformations caused by the test. On the left is the surface after it has been deformed 100 times. The right picture shows a surface after 10000 random deformations.



Without any enhancements, each deformed vertex requires about 0.13 milliseconds of computation. If deformations are just batch-processed, the per vertex time drops to about 0.09 milliseconds. The largest gain occurs when the batches are processed using the priority queue, where it costs only 0.06 milliseconds to process each vertex. This data demonstrates that, with the enhancements, this algorithm can efficiently maintain the sphere hierarchy while the objects in the model are being deformed. A surprising observation from these tests is that the average time remains relatively constant regardless of the size of the batch. This would seem to indicate that most of the cost is incurred by adding each vertex to the batch, especially since the per-vertex time is not much lower even when batches are never processed.



6. Future Work

This research has only addressed a small part of the task of modeling flexible objects. However, within the topic of collision detection, there is much more work that can be done to make this algorithm more broadly useful. Specifically, by increasing the efficiency and speed of this algorithm, it could be better used for extremely time sensitive applications, such as haptics, where collision checking must run at 1 kHz.

As was noted in the analysis of results, collision checks when the two objects are in collision tend to be completed within a small time bound. This interesting fact can be used to help guarantee a higher rate of collision checks than would normally be possible. By carefully setting a timeout for queries, one could be reasonably sure that if no collision had been detected within the allotted time, the objects are separated. The small risk of an undetected collision might be acceptable in some simulation domains to justify the added speed. This is a fundamentally different method of guaranteeing time bounds than in [Hub96].

Perhaps a more significant enhancement to this algorithm would be a method to allow it to handle massively deforming polygon soups. As the algorithm currently exists, if too much deformation occurs, nearby triangles in the sphere tree may actually be very far apart in the model. This will result in the sphere tree losing its information about the geometry (i.e., descending the tree will not allow many spheres to be pruned, as they will all be nearly the same size). In a sense, the tree has lost its geometric balance. This shortcoming was alluded to in [Lin99]. To maintain this balance, spheres will have to be rearranged in the tree to maintain the encoded geometric information. However, these changes must also be made to maintain the combinatorial balance of the tree, so that it does not grow too deep. An ideal algorithm would make the tree always look as optimal as if it were just created.

7. Conclusion

The algorithm presented in this thesis is an efficient method for collision detection and distance computation between two deformable objects. By decoupling the actual collision queries from the hierarchy maintenance, this algorithm maintains the speed and elegance of traditional hierarchy collision detectors. Moreover, this separation makes this a more general-purpose method than any previous work. Prior rigid body collision detectors could not handle deformations while those particularly suited to flexible objects performed poorly with rigid ones. Furthermore, this architecture better allows extensions to this algorithm, such as the enhancement proposed to deal with massive deformations.

8. Bibliography

[Bar90] D. Baraff. *Curved surfaces and coherence for non-penetrating rigid body simulation*. Computer Graphics, 24(4):19-28 (August 1990).

[BW92] David Baraff and Andrew Witkin. *Dynamic simulation of non-penetrating flexible bodies*. Computer Graphics, 26(2):303-308 (July 1992).

[CLMP95] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. *I-COLLIDE: an interactive and exact collision detection system for large-scale environments*. Proc. ACM Interactive 3D Graphics Conf., 189-196 (1995).

[DK90] D. P. Dobkin and D. G. Kirkpatrick. *Determining the separation of preprocessed polyhedra -- a unified approach*. Proc. 17th Internat. Colloq. Automata Lang. Program. Springer-Verlag, Lecture Notes Comput. Sci. 443, 400-413 (1990).

[GLM96] S. Gottschalk, M. C. Lin and D. Manocha. *OBB-tree: a hierarchical structure for rapid interference detection*. Proc. SIGGRAPH '96, 171-180 (1996).

[Hub96] Philip M. Hubbard. *Approximating polyhedra with spheres for time-critical collision detection*. ACM Trans. Graph., 15, 179-210 (1996).

[KPLM98] S. Krishnan, A. Pattekar, M. Lin, and D. Manocha. *Spherical shell: a higher order bounding volume for fast proximity queries*. Proc. Work. Algorithmic Found. Robotics (1998).

[Lin93] M. C. Lin. *Efficient collision detection for animation and robotics*. Ph.D. thesis. University of California (1993).

[Lin99] M. C. Lin. *Fast and Accurate Collision Detection for Virtual Environments*. Proc. IEEE Scientific Visualization Conference (1999).

[Mir97] Mirtich, B. *V-Clip: Fast and Robust Polyhedral Collision Detection*, ACM Transactions on Graphics, 17(3), pp. 177-208 (1998).

[Qui94] Sean Quinlan. *Efficient distance computation between non-convex objects*. Proc. Intern. Conf. on Robotics and Automation, 3324-3329 (1994).