To appear in ACM Symposium on Virtual Reality Software and Technology 1998 (VRST' 98).

Incremental 3D Collision Detection with Hierarchical Data Structures

Tsai-Yen Li
Computer Science Department,
National Chengchi University,
Taipei, Taiwan, R.O.C.
886-2-29387642
li@cs.nccu.edu.tw

Jin-Shin Chen
Computer Science Department,
National Chengchi University,
Taipei, Taiwan, R.O.C.
886-2-29387642
s8213@cs.nccu.edu.tw

1. ABSTRACT

3D collision detection is the most time-consuming component of many geometric reasoning applications. Any improvements on the efficiency of the collision detection module may have a great impact on the overall performance of these applications. Most efficient collision detection algorithms in the literature use some sort of hierarchical bounding volumes, such as spheres or oriented bounding boxes, to reduce the number of calls to expensive collision checks between polygons. In this paper, we propose an incremental scheme that takes advantage of spatial coherence to improve the performance of this class of algorithms. Experiments have been conducted on a sphere-tree structure for several moving objects. Consistent improvements ranging from 70 to 90 percents were observed. These numbers are actually very close to the theoretical upper bound for such improvements.

1.1 Keywords

Collision Detection, Incremental Algorithm, Hierarchical Bounding Volumes, Shape Approximation

2. INTRODUCTION

3D collision detection has been a core component in many geometric reasoning applications. Given two or more geometric models with boundary representation, the goal is to check whether they overlap at a given time instance. It is a special case of distance determination, where the distance between objects is equal to or less than zero.

Most researches on collision detection are in the fields of robotics, CAD/CAM, and computer graphics. In robotics, collision detection modules are used to plan the motion of mobile robots or robot arms[2][12]. In many cases, 90% of overall planning time is spent in detecting collisions or computing distances. In CAD/CAM, collision detection modules are used to help designers assess manufacturability or maintainability of the designed product[4][14]. The cost of collision detection routines is even more significant in the problems encountered in maintainability studies, where objects are in close proximity of each other[4]. In graphics applications, especially in 3D interactive graphics, collision detection routines are crucial for plausible dynamic simulation and for realistic 3D manipulation or navigation in a virtual environment[10].

Many collision detection algorithms have been proposed in the literature. The most naive approach is to check all pairs of geometric primitives, such as polygons, between two objects. This so-called "all-pairs weakness" deems to be very inefficient[8]. Therefore, most research efforts focus on reducing the number of calls to exact collision checks between two geometric primitives. A common characteristic of these algorithms is that they all use some sort of bounding volumes (BV) organized in a tree structure to avoid as many unnecessary exact collision checks as possible. The incurred performance costs include maintaining the data structure and performing overlap checks between the BV' s. Fortunately, these approaches have been shown to be effective because only a very small portion of primitive pairs need to be checked in almost all applications.

In this paper, we present a novel approach that maintains a separation list for a typical recursion tree in detecting collisions to reduce the number of overlap checks between the BV' s. This approach can be used to improve the performance of the class of algorithms utilizing hierarchical data structures. The upper bound on the performance improvement of such an approach is also analyzed. Experimental results are presented to demonstrate its effectiveness.

The rest of the paper is organized as follows. In Section 3, we give a general description for the class of algorithms utilizing hierarchical data structures and analyze their computation costs. In Section 4, we will propose our modifica-

tions to the way that two bounding-volume trees are checked for overlaps. Experimental results on performance improvements are presented in Section 5. Extensions and future work are concluded in the last section.

3. ALGORITHMS WITH HIERACHICAL DATA STRUCTURES

3.1 Related work

The problem of collision detection has been extensively studied in the literature. Good results have been obtained for determining distances between geometric primitives. For example, the closest features between two convex polyhedra can be tracked in constant time as shown in [13]. Another approach based on linear programming was presented in [6]. With slight modifications, this algorithm is shown to have a constant-time complexity as well [3]. All of these algorithms utilize spatial and temporal coherence to achieve constant-time complexity. They have been shown to be effective for large virtual environments consisting of convex polytopes.

However, in many applications, especially in CAD/CAM and robotics, geometric models are not convex, and the existence of contaminated data (as called "polygon soups") are also common. Many general algorithms for non-convex objects have been proposed, and almost all approaches utilize some sort of BV hierarchy. For example, sphere trees were developed in [8][15][16]. Although spheres do not bound a polygon as tight as other BV's, sphere trees are easy to implement and efficient at rejection tests. Oriented bounding box (OBB) is another good BV shown to be effective[7]. Spherical shell is a BV possessing even better tightness and smaller update cost[11]. However, a common problem with these approaches is that they require considerable amount of memory storage. Discrete Orientation Polytope (DOP) is another good choice of BV that requires smaller memory storage than OBB[9][17]. A scaleable scheme utilizing bit addressing to detect possible collisions among thousand of objects with various resolutions has also been proposed in [5]. In addition to these empirical proposals, theoretical results on a class of bounding volume hierarchies called BOXTREE have also been presented in [1].

3.2 Performance analysis

Any collision detection algorithms based on hierarchical BV's involve a one-time preprocessing cost and run-time costs for collision queries. The primary task in the preprocessing step is to build a tree of hierarchical BV's. We enclose an object with the underlying BV's of various sizes and organize them into a tree structure. An important property of such trees is that a node in the upper level will enclose the geometry covered by its child nodes in lower levels. Therefore, if no overlaps of BV's were found at a certain level, there is no need to explore any of its children. Although the running time and memory requirement of this preprocessing step may vary greatly for different algorithms, most performance analyses in the literature do not focus on this step because it is only a one-time cost.

```
boolean CheckBVCollision (node N1, node N2)
if both N1 and N2 are leaves then
    return CheckPrimitiveCollision(N1, N2)
elseif N2 is larger than N1 then
    for each child N2[i] of N2
        if CheckBVCollision (N1,N2[i]) then
            return TRUE
else
    for each child N1[i] of N1
        if CheckBVCollision (N1[i], N2) then
            return TRUE
return FALSE
```

Figure 1. Basic recursive collision check algorithm

In most applications, the run-time collision queries are more important because they constitute the majority of the system's overall running time. According to [7][11], the total cost for interference detection is given by the following cost equation:

$$T = N_u \times C_u + N_v \times C_v + N_p \times C_p \quad (1)$$

where

T: total cost for interference detection,
N_u: no. of BV's updated,
C_u: cost of updating a BV,
N_v: no. of BV overlap tests,
C_v: cost of testing two BV's for overlaps,
N_p: no. of primitive pairs tested for interference,
C_p: cost of testing two primitives for interference.

The choice of BV affects these cost parameters. For example, C_u and C_v are all very small for sphere trees, but their N_v and N_p are larger than other trees. OBB trees and spherical-shell trees are typically smaller in the number of nodes due to their faster convergence (quadratic and cubic, respectively) to the enclosed geometry. Their good tightness also reduces N_p, the total number of calls to interference checks between primitives. The ratio of the times spent on BV overlap tests (N_v x C_v) and primitive interference tests (N_p x C_p) varies greatly for different applications at different instances. For problems with close proximity configurations, such as those encountered in assembly maintainability studies, a typical ratio for these two times is about 1:1. In contrast, when objects are far from each other, primitive interference tests can be totally eliminated. In any cases, reducing N_v, may have a great impact on the overall running time, and it is also the objective of this paper.

3.3 Hierarchical collision detection schemes

The pseudo code in Figure 1 outlines the basic routine in most collision detection algorithms based on hierarchical BV's. Assume that A and a are the root nodes of the BV trees ($BVTree_1$ and $BVTree_2$) for obj_1 and obj_2, respectively. The returned value of CheckBVCollision(A, a) will

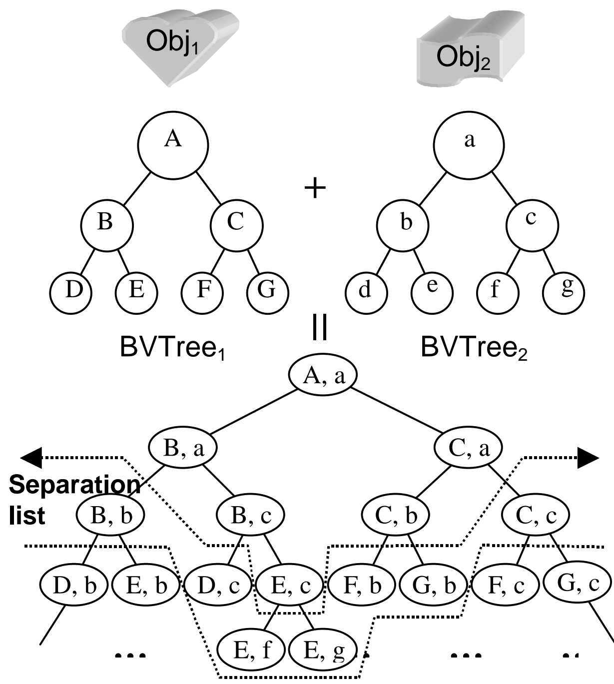

Figure 2. CheckBVCollision's recursion tree and separation list

indicate if obj1 and obj2 are in collision. Note that the metric chosen to compare the size of two nodes is not crucial as long as it decreases monotonically when a node splits.

The recursion tree for the CheckBVCollision routine is depicted in Figure 2. A leaf node of the tree implies an interference check between a pair of primitives. Assume that obj1 and obj2 contain m and n primitives, respectively. Then, there are at least $O(mn)$ leaf nodes at the bottom of the recursion tree no matter what kind of BV is used. The key idea of the algorithms based on hierarchical BV's is that only a very small portion or even none of the leaf nodes (and their enclosing primitives) need to be checked before we can answer a collision query. In fact, it is very unlikely that this recursion tree needs to be fully traversed. The numbers of internal and leaf nodes traversed correspond to N_v and N_p in the cost equation (1), respectively. A BV of better tightness traverses less deeper down the recursion tree than other BV's before it can answer a collision query.

4. The Proposed Incremental Algorithm

Based on an observation from the spatial coherence between two consecutive collision queries, we propose an incremental scheme that can be applied to the existing algorithms to reduce the number of overlap tests between two BV's (N_v in equation (1)).

4.1 The observation

According to the CheckBVCollision routine in Figure 1, the recursion tree in Figure 2 is traversed in a depth-first fashion. When the underlying BV's of a node do not overlap, it returns to its parent immediately, and no further traversal down to its subtree is necessary. Therefore, for each collision query, we can draw a line that cuts the recursion tree into two parts. The BV's of nodes above the cut overlap while those below the cut do not. We call the set of nodes right below this cut a *separation list* for a given collision query. Each node in the list is called a *separation node*. A key observation on this separation list is that when objects do not jump very far between two consecutive collision queries, most of the separation list remains unchanged or only changes locally. Such spatial coherence does exist for most applications in path planning or interactive graphics.

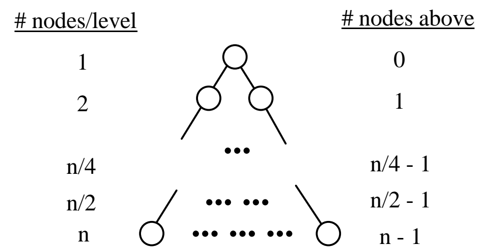

Figure 3. Number of nodes in a recursion tree

4.2 Maintaining a separation list

By keeping track of this separation list, we hope to reduce the number of internal nodes that need to be traversed. We use the list from the previous query as a starting point to generate the separation list for the current query. If coherence between two consecutive queries exists, then only a small portion of the list needs be updated at each time.

Three types of results may occur when updating a separation node. A node may (a) move down, if the BV's of the node overlap. In this case, we lower the portion of the list by calling the CheckBVCollision routine on this node and recording new separation nodes during the traversal. If the BV's of the current node do not overlap, the node may (b) stay still, or (c) move up in the recursion tree. The node stays still if and only if its parent node is in collision. Otherwise, the node will be moved up (maybe several levels) to an appropriate parent node where no collisions occur. In this case, all children of this new node must be excluded from the separation list.

Consider a binary recursion tree, such as the one depicted in Figure 3. The number of nodes doubles when we move the list down by one level. Note that the number of nodes in the list is of the same order as the total number of nodes above it. Assume that the current list contains n nodes. Then, how much can we save by maintaining such a separation list instead of always starting traversing from the root? First, we consider case (a), where the nodes move down. The number of overlap tests we saved is n-1, which is the total number of nodes above the list. Second, for case (b), where the nodes in the current list (n) are not in collision, we must at least check the nodes in one level up (n/2) to know that there is no need to move the list up. The total number of

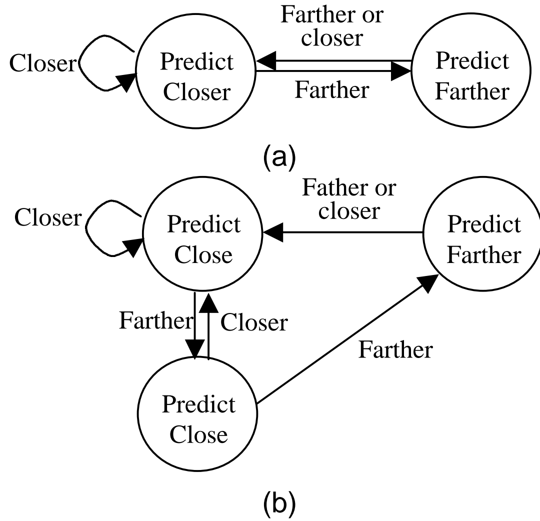

Figure 4. State diagram for updating a separation list

nodes we checked is $1\frac{1}{2}n\,(=n+\frac{1}{2}n)$, which is better than traversing from the root ($2n$). Third, consider case (c), where the list is actually moved up. The number of nodes we need to visit depends on the number of levels that the list moves up. For example, suppose that the list moves up by one level, then the number of nodes that we need to visit is $1\frac{3}{4}n$ $(=n+\frac{1}{2}n+\frac{1}{4}n)$. However, the number is only n $(=\frac{1}{2}n+\frac{1}{2}n)$ if we traverse from the root down. Suppose the probabilities that a node moves down, stays still, and moves up are p_d , p_s, and p_u, respectively. The expected number of nodes that we can save by updating such a list is $n\times p_d+\frac{1}{2}n\times p_s-\frac{3}{4}n\times p_u$. If we give these three types of results equal chance of occurrence (i.e., $p_d=p_s=p_u=\frac{1}{3}$), then the expected number of nodes that we save is only $\frac{1}{4}n$ $(=(n+\frac{1}{2}n-\frac{3}{4}n)/3)$. Therefore, maintaining such a separation list may not result in as good saving as one may expect.

4.3 Predicting motion of a separation list

A main extra cost of updating such a list is on the overhead of checking whether we should move a list up or not. In this type of moves, traversing from the root may actually be a cheaper solution. Therefore, we propose the following hybrid strategy, as shown in Figure 4(a), to maintain the separation list. We predict the motion of the separation list by examining its size (number of nodes) in two consecutive queries. If the size increases, we predict that the objects are getting closer to each other, and we will keep on using this growing list to generate the next separation list. On the other hand, whenever the list stop growing (i.e., its size does not change in two consecutive queries), we predict that they are moving away from each other, and the separation list will be rebuilt from scratch (from the root) in the next query. With this strategy, we do not examine the parent nodes of the current list in any cases, and the list is updated to the correct separation list from time to time when it is rebuilt.

In each of the first two cases (case (a) and (b)), we save n nodes of traversals (the total number of nodes above the list). In the third case, we may not save or waste any nodes. For example, if the list is moved up by one level, it also takes an order of n nodes to get to the list from the root. In all cases, the overall performance of the proposed strategy should be as good as the original one. The expected number of nodes that we can save is $n\times p_d+n\times p_s-0\times p_u$, which is non-negative for any probability distributions. The actual time saving depends on the probability distributions of these updates as well as the number of levels that a list may move up. Nevertheless, experimental results presented in the next section will show that this indeed is a good strategy.

4.4 Deferring rebuilding a separation list

As shown in the previous subsection, the performance of updating the previous separation list can be at least as good as traversing from the top in any cases. However, there are costs to pay in the next query if the list should shrink but it did not. If the number of nodes does not increase in the previous query, the list could be in case (b) (staying still) or case (c) (moving up). Case (c) is the only break-even case, where the costs of traversing the list and the portion of tree above it are the same. We may start to pay the cost and visit extra nodes in the next query if we were in case (c) and the list continues to move up in the next update. On the other hand, it may not worth the effort to rebuild the list if it may move down again in the next query. Therefore, we propose a modified strategy that defers the decision by one query, as shown in the state diagram in Figure 4(b). If the list grows after the update, it means that we were actually in case (b), and it was a good choice not to rebuild the list. If the number of nodes remains the same as in the previous query, then we could be in one of the five cases out of nine possible outcomes for two updates: (*up*, *up*), (*up*, *still*), (*up*, *down*), (*still*, *up*), and (*still*, *still*). The actual separation lists for three of these cases are above the currently maintained list. Therefore, we will choose to rebuild the separation list from scratch in the next query. The overall effect is that the separation list is rebuilt if the list does not grow for two consecutive queries. This strategy, as depicted in Figure 4(b), is very similar to the branch prediction strategy used in pre-fetching pipelined instructions in modern CPU designs. Its effectiveness will be shown in the next section.

5. EXPERIMENTAL RESULTS

5.1 Implementation

We have implemented a collision detection library based on the sphere-tree structure proposed in [15]. The code was written in the C++ language and runs on most UNIX oper-

Run #	Top-Down (sec.)	Using SL		Deferred SL	
		time (sec.)	speedup	time (sec.)	speedup
1	1507.9	1075.7	40%	759.2	99%
2	1744.0	1252.2	39%	1021.7	71%
3	1552.8	1077.7	44%	781.5	99%
4	1713.0	1177.3	46%	980.1	75%
5	1548.4	1080.5	43%	808.2	92%
6	1653.4	1168.3	42%	969.5	71%
7	1403.0	1021.2	37%	687.5	104%
8	1570.5	1120.2	40%	831.4	89%
9	1564.9	1163.1	35%	900.4	74%
10	1580.5	1139.2	39%	865.7	83%
ave.	1583.8	1127.5	40%	860.5	84%

Table 1. Performance comparisons of the improved algorithms

ating systems. The separation list of the recursion tree was implemented as a linked list of nodes comprised of sphere pairs. The code is also used in a path planner based on artificial potential fields (similar to the planner in [4]).

5.2 Performance Evaluation

Independent experiments have been conducted on several geometric models. The experiment reported in this paper was based on a scenario consisting of 9 bunny models of 500 polygons (covered by 3747 spheres) each. Each of these models performs 1000 random motions possibly in all translational and rotational dimensions. The translational increment is the radius of the smallest sphere, and the rotational increment is 5 degrees. In most path planning or graphics applications, the goal of detecting collision is to avoid it. In our experiments, we ensured that objects remain collision-free after any motions by ignoring those moves that cause collisions. In each query, pairwise interference checks among these 9 objects were performed. Therefore, there are 36 separation lists to maintain in each query.

Experiments were conducted on a PC with a 133MHz AMD K5 processor running Linux 2.0.7. The running times of ten experiments using different seeds for random motions are shown in Table 1. The running times of the original algorithm using sphere trees are shown in the second column, while the running times for the improved algorithms using separation lists (without and with deferrals) are shown in the third and fifth columns. The average speedup (as defined in most computer architecture textbooks) for the first improvement without deferring rebuilding the separation list is about 40% while the average speedup for the second improvement with deferrals is 84%. From the analysis in the previous section, we know that the maximal number of nodes that we can save from traversing is the same as the number of nodes in the separation list. This implies that the maximal speedup resulting from such improvements is 100%. 84% is actually very close to this optimal speedup. Note that the best speedup (occurred in the 7th run) is over 100%, which may look impossible. However, we think that this extra improvement is due to the increased spatial locality and the lighter function overhead of using the linear data structure instead of the original recursive function calls.

Examples of various complexities have been used to compare the performances of the original and the modified algorithms. The speedups of the modified algorithms may vary according to the degree of spatial coherence between consecutive queries. Nevertheless, the modified algorithms show consistent improvements on average performance. The strategy of deferring rebuilding the separation list for one query almost always outperforms the one without deferrals. However, our experiments show that deferring for more than one query is not a good idea since the performance starts to degrade.

6. CONCLUSION AND FUTURE WORK

In many applications, collision detection is such a crucial module that any improvements on its running time may have a great impact on the overall performance of the application. Most efficient collision detection algorithms for general geometric models use some sort of hierarchical bounding volumes. In this paper, we have proposed a simple modification to this class of algorithms that can potentially reduce the number of overlap tests between two bounding-volume trees. The modified algorithms try to capture spatial coherence by maintaining a separation list generated from the previous query. We have implemented a collision detection algorithm based on a sphere-tree structure to do our experiments. Experimental results show consistent 70-90% improvements over various geometric models.

The performance improvement of the proposed modification depends on the degree of spatial coherence for the problem at hand. The performance of the modified algorithm could be worse than the original algorithm if objects perform large or random jumps. Therefore, for the result to be fruitful, we need to quantitatively relate the degree of spatial coherence to performance improvement. The application designers can then use this guideline to decide when to use the modified algorithm. In addition, we are doing more experiments on different types of bounding volumes, such as OBB, to measure the degree of improvements on these data structures.

7. ACKNOWLEDGMENTS

This work was partially supported by grants from NSC under contract numbers NSC 86-2213-E-004-006 and NSC 87-2213-E-004-007. The authors also would like to thank Hung-Kai Ting for his help on collecting experimental data.

8. REFERENCES

[1] G. Barequet, B. Chazelle, L.J. Guibas, J. S. B. Mitchell, and A. Tal, "BOXTREE: A Hierarchical

Representation for Surfaces in 3D," *Computer Graphics Forum*, 15(30), pp387-396, Sept. 1996.

[2] J. Barraquand, L. Kavraki, J.C. Latombe T.Y. Li, and P. Raghavan, "A Random Sampling Scheme for Path Planning," *International Journal of Robotics Research*, 16(6), pp759-774, Dec. 1997.

[3] S. Cameron, "Enhancing gjk: Computing Minimum and Penetration Distance between Convex Polyhedra," *Proceedings of International Conference on Robotics and Automation*, 1997.

[4] H.S. Chang and T.Y. Li, "Assembly Maintainability Study with Motion Planning," *Proceedings of 1995 IEEE International Conference on Robotics and Automation*, Nagoya, Japan, May 1995.

[5] K.M. Fairchild, T. Poston, and W. Bricken, "Efficient Virtual Collision Detection for Multiple Users in Large Virtual Spaces," in Proceedings of ACM Symposium on Virtual Reality Software Technology, pp271-285, 1994.

[6] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi, "A Fast Procedure for Computing the Distance between Objects in Three-Dimensional Space," *IEEE Journal of Robotics and Automation*, 4(2), pp193-203, April 1988.

[7] S. Gottschalk, M. Lin, and D. Manocha, "OBB-Tree: A Hierarchical Structure for Rapid Interference Detection," *SIGGRAPH 96 Conference Proceedings, Annual Conference Series*, pp171-180, ACM SIGGRAPH, Addison Wesley, Aug. 1996.

[8] P. M. Hubbard, "Interactive Collision Detection," In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*, October 1993.

[9] J. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan, "Efficient Collision Detection Using Bounding Volume Hierarchies of K-DOP's," *Siggraph'96 Visual Proceedings*, 1996.

[10] Y. Koga, K. Kondo, J. Kuffner, and J.C. Latombe, Planning Motions with Intentions. *Proceedings of SIGGRAPH'94*, pp395-408, 1994.

[11] S. Krishnan, A. Pattekar, M. C. Lin, and D. Manocha, "Spherical Shell: A Higher Order Bounding Volume for Fast Proximity Queries," in *Proceedings of Workshop on Algorithmic Foundation for Robotics '98,* 1998.

[12] J. C. Latombe, "*Robot Motion Planning*," Kluwer Academic Publishers, 1991.

[13] M. C. Lin, "*Efficient Collision Detection for Animation and Robotics*," Ph.D. thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, December 1993.

[14] M. C. Lin, D. Manocha, M. K. Ponamgi and J. D. Cohen, "A Coherence-based Hierarchical Approach to Interference Problems for Virtual Prototyping", in *Product Modeling for Computer Integrated Design and Manufacture*, edited by M. Pratt, R. D. Sriram, and M. J. Wozny, 1996.

[15] S. Quinlan, "Efficient Distance Computation between Non-Convex Objects," *Proceedings of International Conference on Robotics and Automation*, pp3324-3329, San Diego, CA, 1994.

[16] I. J. Palmer and R. L. Grimsdale, "Collision Detection for Animation Using Sphere-Tree," *Computer Graphics Forum*, 14(2), pp105-116, June 1995.

[17] G. Zachmann, "Rapid Collision Detection by Dynamically Aligned DOP-Trees," *Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS '98),* Atlanta, Georgia, March 1998.