Efficient Collision Detection of Complex Deformable Models using AABB Trees

GINO VAN DEN BERGEN

Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
E-mail: gino@win.tue.nl

November 6, 1998

Abstract

We present a scheme for exact collision detection between complex models undergoing rigid motion and deformation. The scheme relies on a hierarchical model representation using axis-aligned bounding boxes (AABBs). In recent work, AABB trees have been shown to be slower than oriented bounding box (OBB) trees. In this paper, we describe a way to speed up overlap tests between AABBs, such that for collision detection of rigid models, the difference in performance between the two representations is greatly reduced. Furthermore, we show how to quickly update an AABB tree as a model is deformed. We thus find AABB trees to be the method of choice for collision detection of complex models undergoing deformation. In fact, because they are not much slower to test, are faster to build, and use less storage than OBB trees, AABB trees might be a reasonable choice for rigid models as well.

Keywords: computer animation, collision detection, hierarchical data structures, deformable models

1 Introduction

Hierarchies of bounding volumes provide a fast way to perform exact collision detection between complex models. Examples of volume types that are used for this purpose are spheres [8, 5], oriented bounding boxes (OBBs) [4], and discrete-orientation polytopes (DOPs) [6, 12]. In this paper, we present a collision detection scheme that relies on a hierarchical model representation using axis-aligned bounding boxes (AABBs). In the AABB trees as we use them, the boxes are aligned to the axes of the model's local coordinate system, thus, all the boxes in a tree have the same orientation.

In recent work [4], AABB trees have been shown to yield a worse performance than OBB trees for rigid models. In this paper, however, we present a way to speed up overlap testing between relatively oriented boxes of a pair of AABB trees. This results in a performance for the AABB tree that is close to the OBB tree's performance for collision detection of rigid models.

Furthermore, we show how to quickly update an AABB tree as a model is deformed. Updating an AABB tree after a deformation is considerably faster than rebuilding the tree, and results in a tight-fitting hierarchy of boxes for most types of deformations. Since updating an OBB tree is significantly more complex, we find AABB trees to be the method of choice for collision detection of complex models undergoing deformation. In fact, because they are not much slower to test, are faster to build, and use less storage than OBB trees, AABB trees might be a reasonable choice for rigid models as well.

In comparison to a previous algorithm for deformable models presented in [10], the algorithm presented here is expected to perform better for deformable models that are placed in close proximity. For these cases, both algorithms show a time complexity that is roughly linear in the number of primitives. However, our approach has a smaller constant (asymptoticly 48 arithmetic operations per triangle for triangle meshes). Moreover, our algorithm is better suited for collision detection among a mix of rigid and deformable models, since it is linear in the number of primitives in the deformable models only.

The C++ source code for the scheme presented here is released as part of the Software Library for Interference Detection (SOLID) version 2.0[1].

[1]Information on how to obtain the complete C++ source code and documentation for SOLID 2.0 is available at *http://www.acm.org/jgt/papers/vanDenBergen98*.

2 Building an AABB Tree

The AABB tree that we consider is, as the OBB tree described in [4], a binary tree. The two structures differ with respect to the freedom of placement of the bounding boxes: AABBs are aligned to the axes of the model's local coordinate system, whereas OBBs can be arbitrarily oriented. The added freedom of an OBB is gained at a considerable cost of storage space. An OBB is represented using 15 scalars (9 scalars for a 3×3 matrix representing the orientation, 3 scalars for position, and 3 for extent), whereas an AABB only requires 6 scalars (for position and extent). Hence, an AABB tree of a model requires roughly half as much storage space as an OBB tree of the same model.

An AABB tree is constructed top-down, by recursive subdivision. At each recursion step, the smallest AABB of the set of primitives is computed, and the set is split by ordering the primitives with respect to a well-chosen partitioning plane. This process continues until each subset contains one element. Thus, an AABB tree for a set of n primitives has n leaves and $n - 1$ internal nodes.

At each step, we choose the partitioning plane orthogonal to the longest axis of the AABB. In this way, we get a 'fat' subdivision. In general, fat AABBs, i.e., cube-like rather than oblong, yield a better performance in intersection testing, since under the assumption that the boxes in a tree mutually overlap as little as possible, a given query box can overlap fewer fat boxes than thin boxes.

We position the partitioning plane along the longest axis, by choosing δ, the coordinate on the longest axis where the partitioning plane intersects the axis. We then split the set of primitives into a negative and positive subset corresponding to the respective halfspaces of the plane. A primitive is classified as positive if the midpoint of its projection onto the axis is greater than δ, and negative otherwise. Figure 1 shows a primitive that straddles the partitioning plane depicted by a dashed line. This primitive is classified as positive. It can be seen that by using this subdivision method, the degree of overlap between the AABBs of the two subsets is kept small.

For choosing the partitioning coordinate δ we tried several heuristics. Our experiments with AABB trees for a number of polygonal models showed us that, in general, the best performance is achieved by simply choosing δ to be the median of the AABB, thus splitting the box in two equal halves. Using this heuristic, it may take $O(n^2)$ time in the worst case to build an AABB tree for n primitives, however, in the usual case where the primitives are distributed more or less uniformly over the box, building an AABB tree takes only $O(n \log n)$ time. Other heuristics we have tried, that didn't perform as well, are: (a) subdividing the set of

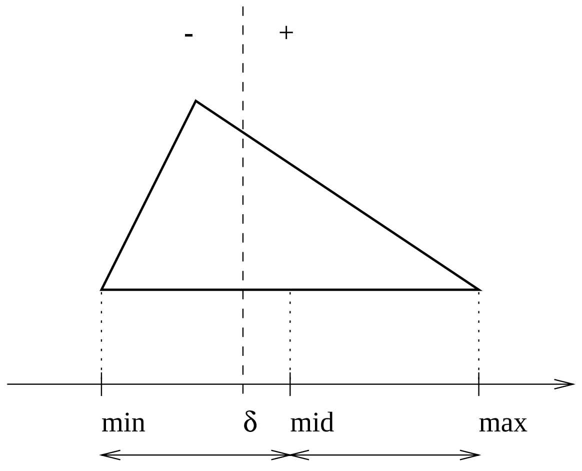


Figure 1: The primitive is classified as positive, since its midpoint on the coordinate axis is greater than δ.

primitives in two sets of equal size, thus building an optimally balanced tree, and (b) building a halfbalanced tree, i.e., the larger subset is at most twice as large as the smaller one, and the overlap of the subsets' AABBs projected onto the longest axis is minimized.

Occasionally, it may occur that all primitives are classified to the same side of the plane. This will happen most frequently when the set of primitives contains only a few elements. In this case, we simply split the set in two subsets of (almost) equal size, disregarding the geometric location of the primitives.

Building an AABB tree of a given model is faster than building an OBB tree for that model, since the estimation of the best orientation of an OBB for a given set of primitives requires additional computations. We found that building an OBB tree takes about three times as much time as building an AABB tree, as is shown in Section 5.

3 Intersection Testing

An intersection test between two models is done by recursively testing pairs of nodes. For each visited pair of nodes, the AABBs are tested for overlap. Only the nodes for which the AABBs overlap are further traversed. If both nodes are leaves then the primitives are tested for intersection and the result of the test is passed back. Otherwise, if one of the nodes is a leaf and the other an internal node, then the leaf node is tested for intersection with each of the children of the internal node. Finally, if both nodes are internal nodes then the node with smaller volume is tested for intersection with the children of the node with the larger volume. The latter heuristic choice of unfolding the node with the largest volume results in the largest reduction of total volume size in the following AABB tests, thus the lowest probability of the following tested boxes overlapping.

Since the local coordinate systems of a pair of models may be arbitrarily oriented, we need an overlap test for relatively oriented boxes. A fast overlap test for oriented boxes is presented by Gottschalk in [4]. We will refer to this test as the *separating axes test* (SAT). A separating axis of two boxes is an axis for which the projections of the boxes onto the axis do not overlap. The existence of a separating axis for a pair of boxes sufficiently classifies the boxes as disjoint. It can be shown that for any disjoint pair of convex three-dimensional polytopes a separating axis can be found that is either orthogonal to a facet of one of the polytopes, or orthogonal to an edge from each polytope [3]. This results in 15 potential separating axes that need to be tested for a pair of oriented boxes (3 facet orientations per box plus 9 pairwise combinations of edge directions). The SAT exits as soon as a separating axis is found. If none of the 15 axes separate the boxes, then the boxes overlap.

We refer to the original paper for details on how the SAT is implemented such that it uses the least number of operations. For the following discussion, it is important to note that this implementation requires the relative orientation represented by a 3×3 matrix, and its absolute value, i.e., the matrix of absolute values of matrix elements, to be computed before performing the 15 axes tests.

In general, testing two AABB trees for intersection requires more box overlap tests than testing two OBB trees of the same models, since the smallest AABB of a set of primitives is usually larger than the smallest OBB. However, since each tested pair of boxes of two OBB trees normally has a different relative orientation, the matrix operations for computing this orientation and its absolute value are repeated for each tested pair of boxes, whereas for AABB trees the relative orientation is the same for each tested pair of boxes, and thus needs to be computed only once. Therefore, the performance of an AABB tree might not be as bad as we would expect. The empirical results in Section 5 show that, by exploiting this feature, intersection testing using AABB trees usually takes only 50% longer than using OBB trees in cases where there is a lot of overlap among the models.

For both tree types, the most time consuming operation in the intersection test is the SAT, so let us see if there is room for improvement. We found that,

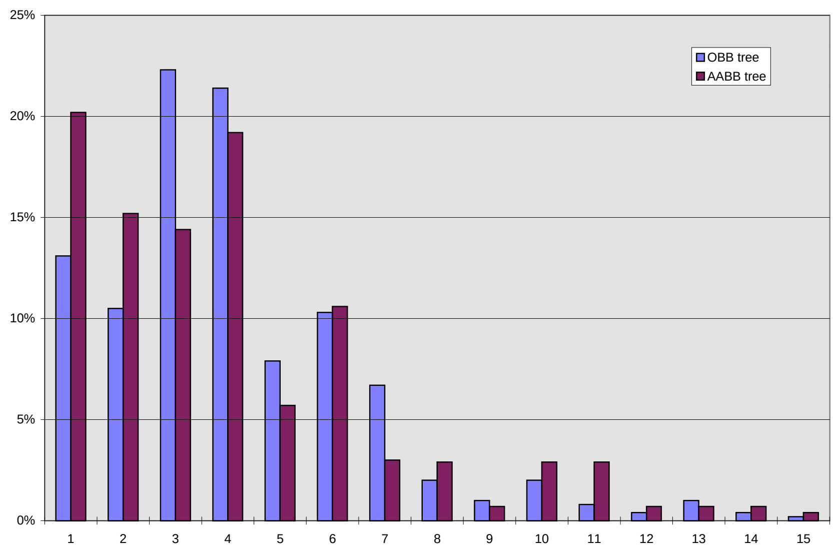


Figure 2: Distribution of axes on which the SAT exits in case of the boxes being disjoint. Axes 1 to 6 correspond to the facet orientations of the boxes, and axes 7 to 15 correspond to the combinations of edge directions.

in the case where the boxes are disjoint, the probability of the SAT exiting on an axis corresponding to a pair of edge directions is about 15%. Figure 2 shows a distribution of the separating axes on which the SAT exits for tests with a high probability of the models intersecting. Moreover, for both the OBB and the AABB tree we found that about 60% of all box overlap tests resulted in a positive result. Thus, if we remove from the SAT the nine axis tests corresponding to the edge directions, we will get an incorrect result only 6% (40% of 15%) of the time.

Since the box overlap test is used for quick rejection of subsets of primitives, exact determination of a box overlap is not necessary. Using a box overlap test that returns more overlaps than there actually are, results in more nodes being visited, and thus more box overlap and primitive intersection tests. Testing fewer axes in the SAT reduces the cost of a box overlap test, but increases the number of box and primitive pairs being tested. Apparently, there is a trade-off of per-test cost against number of tests, when we use a SAT that tests fewer axes.

In order to examine whether this trade-off is in favor of the performance, we repeated the experiment using a SAT that tests only the six facet orientations. We refer to this test as the *SAT lite*. The results of this experiment are shown in Section 5. We found that the AABB tree's performance benefits from a cheaper but sloppier box overlap test in all cases, whereas the OBB tree shows hardly any change in performance. This is explained by the higher cost of a box overlap test for the OBB tree due to extra matrix operations.

4 AABB Trees and Deformable Models

AABB trees lend themselves quite easily to be used for deformable models. In this context, a deformable model is a set of primitives in which the placements and shapes of the primitives within the model's local coordinate system change over time. A typical example of a deformable model is a triangle mesh in which the local coordinates of the vertices are time-dependent.

Instead of rebuilding the tree after a deformation, it is usually a lot faster to refit the boxes in the tree. The following property of AABBs allows an AABB tree to be refitted efficiently in a bottom-up manner. Let S be a set of primitives and S^+, S^-, subsets of S such that $S^+ \cup S^- = S$, and let B^+ and B^- be the smallest AABBs of respectively S^+ and S^-, and B, the smallest AABB enclosing $B^+ \cup B^-$. Then, B is also the smallest AABB of S. This property is illustrated in Figure 3. Of all bounding volume types we have seen so far, AABBs share this property only with DOPs.

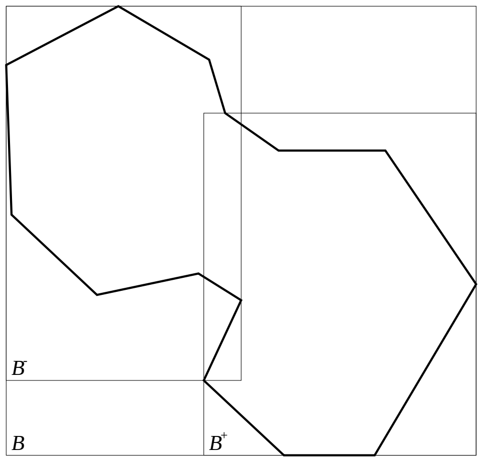
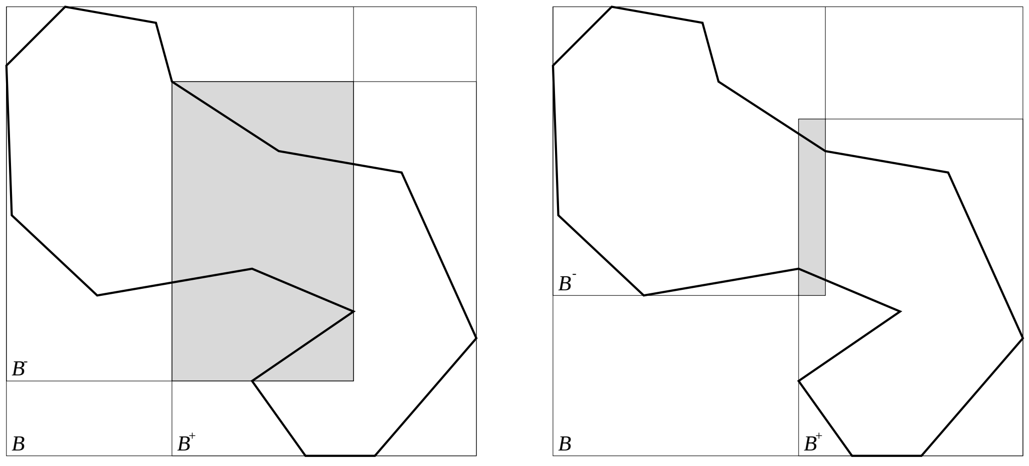


Figure 3: The smallest AABB of a set of primitives encloses the smallest AABBs of the subsets in a partition of the set.

This property of AABBs yields a straightforward method for refitting a hierarchy of AABBs after a deformation. First the bounding boxes of the leaves are recomputed, after which each parent box is recomputed using the boxes of its children in a strict bottom-up order. This operation may be implemented as a posorder tree traversal, i.e., for each internal node, the children are visited first, after which the bounding box is recomputed. However, in order to avoid the overhead of recursive function calls, we implement it differently.

In our implementation the leaves and the internal nodes of an AABB tree are allocated as arrays of nodes. We are able to do this, since the number of primitives in the model is static and a priori known. Furthermore, the tree is built such that each internal child node's index number in the array is greater than its parent's index number. In this way, the internal nodes are refitted properly by iterating over the array of internal nodes in reversed order. Since refitting an AABB takes constant time for both internal nodes and leaves, an AABB tree is refitted in time linear to the number of nodes. Refitting an AABB tree of a triangle mesh takes less than 48 arithmetic operations per triangle. Experiments have shown that for models composed of over 6000 triangles, refitting an AABB tree is about ten times as fast as rebuilding it.

There is, however, a drawback to this method of refitting. Due to relative position changes of primitives in the model after a deformation, the boxes in a refitted tree may have a higher degree of overlap than the boxes in a rebuilt tree.



(a) Refitted

(b) Rebuilt

Figure 4: Refitting vs. rebuilding the model in Figure 3 after a deformation

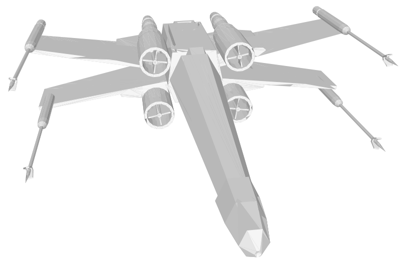
Figure 4 illustrates this effect for the model in Figure 3. A higher degree of overlap of boxes in the tree results in more nodes being visited during an intersection test, and thus, a worse performance for intersection testing.

We observe a higher degree of overlap among the boxes in a refitted tree mostly for radical deformations such as excessive twists, features blown out of proportion, or extreme forms of self-intersection. However, for deformations that keep the adjacency relation of triangles in a mesh intact, i.e., the mesh is not torn up, we found no significant performance deterioration for intersection testing, even for the more severe deformations. This is due to the fact that the degree of overlap increases mostly for the boxes that are maintained high in the tree, whereas most of the boxes that are tested are the ones that are maintained close to the leaves.

5 Performance

The total cost of testing a pair of models represented by bounding volume hierarchies is expressed in the following cost function [11, 4]:

$$T_{total} = N_b * C_b + N_p * C_p,$$


(a) X-wing

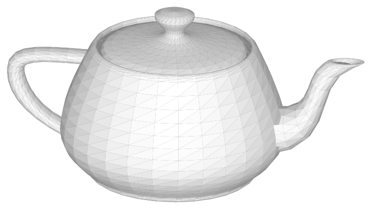

(b) Teapot

Figure 5: Two models that where used in our experiments

where

T_{total}	is the total cost of testing a pair of models for intersection,
N_b	is the number of bounding volume pairs tested for overlap,
C_b	is the cost of testing a pair of bounding volumes for overlap,
N_p	is the number of primitive pairs tested for intersection, and
C_p	is the cost of testing a pair of primitives for intersection.

The parameters in the cost function that are affected by the choice of bounding volume are N_b, N_p, and C_b. A tight-fitting bounding volume type, such as the OBB, results in a low N_b and N_p, but has a relatively high C_b, whereas an AABB will result in more tests being performed, but the value of C_b will be lower.

In order to compare the performances of the AABB tree and the OBB tree, we have conducted an experiment, in which a pair of models were placed randomly in a bounded space and tested for intersection. The random orientations of the models were generated using the method described by Shoemake in [9]. The models were positioned by placing the origin of each model's local coordinate system randomly inside a cube. The probability of an intersection is tuned by changing the size of the cube. For all tests, the probability was set to approximately 60%.

OBB tree							
Model	N_b	C_b	T_b	N_p	C_p	T_p	T_{total}
Torus	10178961	4.9	49.7	197314	15	2.9	52.6
X-wing	48890612	4.6	223.8	975217	10	10.2	234.0
Teapot	12025710	4.8	57.6	186329	14	2.7	60.3

AABB tree							
Model	N_b	C_b	T_b	N_p	C_p	T_p	T_{total}
Torus	32913297	3.7	122.3	3996806	7.2	28.7	151.0
X-wing	92376250	3.1	288.8	8601433	7.1	61.3	350.1
Teapot	25810569	3.3	84.8	1874830	7.4	13.9	98.7

Table 1: Performance of the AABB tree vs. the OBB tree, both using the SAT. N_b and N_p are respectively the total number box and triangle intersection tests, C_b and C_p the per-test times in microseconds for respectively the box and triangle intersection test, $T_b = N_b * C_b$ is the total time in seconds spent testing for box intersections, $T_p = N_p * C_p$ is the total time used for triangle intersection tests, and finally T_{total} is the total time in seconds for performing 100K intersection tests.

For this experiment we used Gottschalk's RAPID package [2] for the OBB tree tests. For the AABB tree tests, we used a modified RAPID, in which we removed the unnecessary matrix operations. We experimented with three models: a torus composed of 5000 triangles, a slenderly shaped *X-wing* space craft composed of 6084 triangles, and the archetypical teapot composed of 3752 triangles, as shown in Figure 5. Each test performed 100K random placements and intersection tests, resulting in approximately 60K collisions for all tested models. Table 1 shows the results of the tests for both the OBB tree and the AABB tree. The tests were performed on a Sun UltraSPARC-I (167MHz), compiled using the GNU compiler with '-O2' optimization.

An AABB tree requires approximately twice as much box intersection tests as an OBB tree, however, the time used for intersection testing is in most cases only 50% longer for AABB trees. The exception here is the torus model, for which the AABB tree uses almost three times as much time as the OBB tree. Apparently, the OBB tree excels in fitting models that have a smooth surface composed of uniformly distributed primitives. Furthermore, we observe that, due to its tighter fit, the OBB tree requires much fewer triangle intersection tests (less than two triangle intersection tests per placement, for the torus and the teapot).

OBB tree							
Model	N_b	C_b	T_b	N_p	C_p	T_p	T_{total}
Torus	13116295	3.7	47.9	371345	12	4.4	52.3
X-wing	65041340	3.4	221.4	2451543	9.3	22.9	244.3
Teapot	14404588	3.5	50.8	279987	13	3.5	54.3

AABB tree							
Model	N_b	C_b	T_b	N_p	C_p	T_p	T_{total}
Torus	40238149	2.4	96.1	5222836	7.4	38.4	134.5
X-wing	121462120	1.9	236.7	13066095	7.0	91.3	328.0
Teapot	30127623	2.1	62.5	2214671	7.0	15.6	78.1

Table 2: Performance of AABB tree vs. OBB tree, both using the SAT lite

We repeated the experiment using a separating axes test that tests only the axes corresponding to the six facet orientations, referred to as *SAT lite*. The results of this experiment are shown in Table 2. We see a performance increase of about 15% on average for the AABB tree, whereas the change in performance for the OBB tree is only marginal.

We also ran some tests to see how the time used for refitting an AABB tree for a deformable model compares to the intersection testing time. We found that on our testing platform, refitting a triangle mesh composed of a large number ($>$ 1000) of triangles takes 2.9 microseconds per triangle. For instance, for a pair of models composed of 5000 triangles each, refitting takes 29 milliseconds, which is more than 10 times the amount of time it takes to test the models for intersection. Hence, refitting is likely to become the bottleneck if many of the models in a simulated environment are deformed and refitted in each frame. However, for environments with many moving models, in which only a few are deformed in each frame, refitting will not take much more time in total than intersection testing.

We conclude with a comparison of the performance of the AABB tree vs. the OBB tree for deformable models. Table 3 presents an overview of the times we found for operations on the two tree types. We see that for deformable models, the OBB's faster intersection test is not easily going to make up for the high cost of rebuilding the OBB trees, even if only a few of the models are deformed. For these cases, AABB trees, which are refitted in less than 5% of the time it takes to rebuild an OBB tree, will yield a better performance, and are therefore the preferred method for collision detection of deformable models.

Operation	Torus	X-wing	Teapot
Build an OBB tree	0.35 s	0.46 s	0.27 s
Build an AABB tree	0.11 s	0.18 s	0.08 s
Refit an AABB tree	15 ms	18 ms	11 ms
Test a pair of OBB trees	0.5 ms	2.3 ms	0.6 ms
Test a pair of AABB trees	1.3 ms	3.3 ms	0.8 ms

Table 3: Comparing the times for a number of operations

6 Implementation Notes

In SOLID 2.0, AABB trees are used both for rigid and deformable models. In order to comply with the structures and motions specified in VRML [1], SOLID allows, besides translations and rotations, also nonuniform scalings on models. Note that a nonuniform scaling is not considered a deformation, and hence, does not require refitting. However, in order to be able to use nonuniformly scaled models, some changes in the AABB overlap test are needed.

Let $\mathbf{T}(\mathbf{x}) = \mathbf{B}\mathbf{x} + \mathbf{c}$ be the relative transformation from a model's local coordinate system to the local coordinate system of another model, where $\mathbf{B}$ is a 3×3 matrix, representing the orientation and scaling, and $\mathbf{c}$ is a vector representing the translation. For nonuniformly scaled models, we can not rely on the matrix $\mathbf{B}$ being orthogonal, i.e., $\mathbf{B}^{-1} = \mathbf{B}^{\mathrm{T}}$. However, for the SAT lite both $\mathbf{B}$ and $\mathbf{B}^{-1}$, and their respective absolute values are needed. Hence, in our implementation we compute these four matrices for each intersection test of a pair of models, and use them for each tested pair of boxes. The added cost of allowing nonuniformly scaled models is negligible, since $\mathbf{B}^{-1}$ and its absolute value is computed only once for each tested pair of models.

Finally, it is worth mentioning that for AABB trees a larger percentage of the time is used for primitive intersection tests than for OBB trees (28% vs. 5%). In this respect, it might be good idea to use the triangle intersection test presented by Möller in [7], which is shown to be faster than the one used in RAPID.

References

[1] G. Bell, R. Carey, and C. Marrin. VRML97: The virtual reality modeling language. http://www.vrml.org/Specifications/VRML97, 1997.

[2] S. Gottschalk. RAPID: Robust and accurate polygon interference detection system. http://www.cs.unc.edu/~geom/OBB/OBBT.html, 1996. software library.

[3] S. Gottschalk. Separating axis theorem. Technical Report TR96-024, Dept. of Computer Science, UNC Chapel Hill, 1996.

[4] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. In *Proc. SIGGRAPH '96*, pages 171–180, 1996.

[5] P. M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, July 1996.

[6] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of *k*-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[7] T. Möller. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30, 1997.

[8] I. J. Palmer and R. L. Grimsdale. Collision detection for animation using sphere-trees. *Computer Graphics Forum*, 14(2):105–116, 1995.

[9] K. Shoemake. Uniform random rotations. In D. Kirk, editor, *Graphics Gems III*, pages 124–132. Academic Press, Boston, MA, 1992.

[10] A. Smith, Y. Kitamura, H. Takemura, and F. Kishino. A simple and efficient method for accurate collision detection among deformable polyhedral objects in arbitrary motion. In *Proc. IEEE Virtual Reality Annual International Symposium*, pages 136–145, 1995.

[11] H. Weghorst, G. Hooper, and D. P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, Jan. 1994.

[12] G. Zachmann. Rapid collision detection by dynamically aligned DOP-trees. In *Proc. IEEE Virtual Reality Annual International Symposium*, pages 90–97, 1998.