A Benchmarking Suite for Static Collision Detection Algorithms

Sven Trenkel
TU Clausthal, Germany
trenkel@tu-clausthal.de

Rene Weller
TU Clausthal, Germany
weller@in.tu-clausthal.de

Gabriel Zachmann
TU Clausthal, Germany
zach@in.tu-clausthal.de

ABSTRACT

In this paper, we present a benchmarking suite that allows a systematic comparison of pairwise static collision detection algorithms for rigid objects. The benchmark generates a number of positions and orientations for a predefined distance. We implemented the benchmarking procedure and compared a wide number of freely available collision detection algorithms.

Keywords: Collision Detection, Benchmarking

1 INTRODUCTION

Fast algorithms for collision detection between polygonal objects are needed in many fields of computer science, e.g. in physically based simulations, computer games, or robotics. In many of these applications, collision detection is the computational bottleneck. In order to gain a maximum speed of applications, it is essential to select the best suited algorithm.

There are a number of algorithms for collision detection between rigid objects. Unfortunately, it is extremely difficult to evaluate and compare collision detection algorithms, because in general they are very sensitive to specific scenarios, i.e. to the relative size of the two objects, the relative position to each other, the distance, etc.

The design of a standardized benchmarking suite for collision detection would make fair comparisons between algorithms much easier. Such a benchmark must be designed with care, so that it includes a broad spectrum of different and interesting contact scenarios. However, there are no standard benchmarks available to compare different algorithms. As a result, it is nontrivial to compare two algorithms and their implementations.

In this paper, we propose a simple benchmark procedure which eliminates these effects. It has been kept very simple so that other researchers can easily reproduce the results and compare their algorithms.

The user only has to specify a small number of parameters, namely: The objects he wants to test, the number of sample points, and, finally, a set of distances. Our algorithm then generates the required number of test positions and orientations by placing the object in the given distances.

Our benchmarking suite is flexible, robust, and it is easy to integrate other collision detection libraries. Moreover, the benchmarking suite is freely available and could be downloaded together with a set of objects in different resolutions that cover a wide range of possible scenarios for collision detection algorithms, and a set of precomputed test points for these objects [1].

2 RELATED WORK

There does not exist much work about special benchmarking suites for collision detection algorithms. Most authors simply choose some objects and test them in a not further described way, or they restrict their explorations just to some special scenarios. A first approach for a comprehensive and objective benchmarking suite was given by [Zac98]. The code for the benchmark is freely available. However, it does not guarantee to produce results with practical relevance, because the objects interpenetrate heavily during the benchmark, but collision detection is mostly used to avoid interpenetrations. In many simulations, objects are allowed to collide only a little bit, and then the collision handling resolves the collision by backtracking or a spring-damping approach.

[OL03] chose a set of physically based simulations to test their collision detection algorithms. This scenarios are a torus falling down a spiral peg, a spoon in a cup, and a soup of numbers in a bowl. [vdB97] positioned two models by placing the origin of each model randomly inside a cube. The probability of an intersection is tuned by changing the size of the cube. The problem here is that it is stochastic, and that a lot of large and irrelevant distances are tested.

[1] http://cg.in.tu-clausthal.de/research/colldet_benchmark/

[CRM02] presented a comparison with the special focus on motion planing. They used different scenes in their probabilistic motion planner for the benchmark. However, such a benchmarking suite compares the collision detection algorithms only in a special scenario. In this paper, we present a more flexible and general benchmarking suite which produces more reliable comparisons.

3 THE BENCHMARKING ALGORITHM

Nearly all collision detection libraries for static collision detection between rigid objects are based on bounding volume hierarchies (BVHs). If the bounding volume (BV) of an object does not intersect a volume higher in the tree, then it cannot intersect any object below that node. So, they are all rejected very quickly. If two objects overlap, the recursive traversal during the collision check should quickly converge towards the colliding polygon pair. So, it is most time consuming if the BVHs overlap, but the objects do not.

Therefore, in most collision detection algorithms, the testing time depends mainly on the configuration of the two objects and their shapes, i.e. the positions, orientations and distance, and to a lesser amount on their complexity. Therefore, it seems to be reasonable for a well-balanced benchmarking procedure to test as many configurations for a given distance as possible.

3.1 The Search Space

Without loss of generality, it is sufficient to rotate only one of the objects in order to get all possible configurations, because we can simply transform one of the objects into the coordinate system of the other. This does not change the relative position of the objects. Therefore, our search space has 6 dimensions.

As even a 6D search space is too big to be tested, we have to reduce it by sampling. In order to find a large number of sampling points, we propose two different methods in our benchmarking suite. We call them *sphere method* and *grid method*. The sphere method is faster, but could miss some interesting configurations, while the grid method is more accurate. Both methods start with a fixed rotation. After a cycle of method-specific translations, the moving object is rotated and the next cycle can start until a user specified number of rotations is reached.

The Grid Method The first method uses a simple axis-aligned grid to find the translations. The center of the moving object is moved to the center of all cells. For each of these, the object is moved towards the fixed object until the required distance is reached. Then, the configuration is stored. Unfortunately, it is not possible to know the number of configurations found by this method in advance.

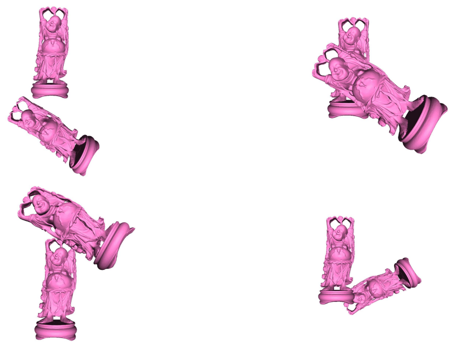
Figure 1: Our sphere-method uses a fixed rotation for every cycle. The moving object is rotated around the fixed object. After a cycle is finished, the rotation is changed.

The Sphere-Method The main idea of this method is to reduce the time for finding possible configurations. To this end, the 3D search space is reduced to 2 dimensions by using spherical coordinates. Nevertheless, it might happen to miss some interesting configurations. Within this method, we place the moving object on a sphere around the fixed object. The sphere should be bigger than the required distance. In the next step, we move the object towards the fixed object on a straight line through the center of the sphere until we reach the required distance. Because there could be several points that match the required distance on the straight line, it is possible to miss some configurations. In addition to the higher speed of this method, it is possible to define the number of located configurations in advance, because every straight line leads to exactly one configuration (see Fig. 1).

At the end of this procedure, we have got a large number of configurations for a user specified number of object-object-distances. This has to be done only once as preprocessing step, even if we add another collision detection library to the set later, or if we move to other platforms.

3.2 Benchmarking

The bulk of the work has been done in the previous step. In order to actually perform the benchmark, we just load the set of configurations. For each object-object distance we start the clock, set the transformation matrix of the moving object to all the configurations associated with the distance, and perform a collision test for each of them. After that, we can compute an average collision detection time for this distance.

3.3 Distance Computing

One method to determine the distance between two objects is to use the collision detection algorithms itself. We can build an offset object from the fixed object where the offset equals the specified distance. Then, we can conduct a binary search until we find a point where the moving object is just touching the offset object. However, offset objects can get very large for complex objects.

That is why we propose another method: The PQP-library offers the possibility to compute the distance between two objects by using swept spheres. With a given distance, we can also do a binary search until we find a point which matches the specified distance.

However, distance computing is more complicated than collision detection. Thus, this method is more time consuming. On the other hand, it is more accurate and less memory intensive than the offset object method. Therefore, we prefer this method for our benchmark. Another advantage of this method is that we know the exact distance between the objects during the binary search. We can use this information to delete cells in the grid method with a higher distance than the specified one. This accelerates the search for configurations.

Indeed, our benchmarking suite supports both methods for distance computing, because PQP is not Open Source software and, therefore, it is not possible to deliver it directly with our benchmarking suite.

Another problem that arises during distance computation concerns numerical stability. Because we are forced to floating point accuracy, it is not possible to find configurations with an exact distance while doing binary search. On account of this, we use an accuracy of 0.001% relative to the size of the fixed object in our benchmark. Of course, this accuracy can be changed by the user.

4 IMPLEMENTATION

Most collision detection libraries use proprietary internal data structures for data representation. Therefore, it is not possible to pass all kinds of objects directly to the algorithms. We chose OpenSG, a freely available scenegraph system for object management, because it offers support for many file formats, it is portable to many operating systems and, its data structures are well documented and easy to use. We wrote a wrapper for every collision detection library in order to convert the OpenSG data to the specific required data structures of the collision detection libraries. During initialization, our benchmark simply checks if the dynamically linked libraries are available and, if so, loads them.

We tested a wide variety of freely available collision detection libraries, precisely:

V-Collide: V-Collide, proposed by [HLC$^+$97], is a wrapper with a simple interface for I-Collide and the RAPID library. In a first step, a sweep-and-prune algorithm is used to detect potentially overlapping pairs of objects. In a second step, the RAPID library is used for the exact pairwise test between a pair of objects. It uses an oriented bounding box test to find possibly colliding pairs of triangles.

PQP: PQP [GLM96] [LGLM99] is also based on the RAPID library. As with RAPID, PQP uses oriented bounding boxes. Furthermore, PQP is also able to compute the distance between the closest pair of points. For distance and tolerance queries, a different BV type, the so-called swept spheres, is used.

FreeSolid: FreeSolid, developed by [vdB99], uses axis-aligned bounding boxes (AABBs) for collision detection. For a fast collision test between the AABB hierarchies, the acceleration scheme described in [vdB97] is used. FreeSolid could also handle deformations of the geometry.

Opcode: Opcode, introduced by [Ter01], is a collision detection library for pairwise collision tests. It uses AABB hierarchies with a special focus on memory optimization. Therefore, it uses so-called no-leaf, i.e., BVHs of which the leaf nodes have been removed. For additionally acceleration it uses primitive-BV overlap tests during recursive traversal, whereas all other libraries described in this paper only use primitive-primitive-tests and BV-BV-tests. Like Freesolid, Opcode also supports deformable meshes.

BoxTree: The BoxTree, described in [Zac95], is a memory optimized version of the AABB trees. Instead of storing 6 values for the extents of the boxes, only two splitting planes are stored. For the acceleration of n-body simulations, the libraries offers support for a grid.

Dop-Tree: The Dop-Tree [Zac98] uses discrete oriented polytopes (k-DOPs, where k is the number of orientations) as BVs. k-DOPs are a generalization of axis aligned bounding boxes. The library supports different numbers of orientations. In [Zac98] it is shown that $k = 24$ guarantees the highest performance. Therefore, we also chose this number for our measurements. The set of orientations is fixed. This library also supports n-body simulation via grids.

When running the configuration space exploration (see section 3), the user simply specifies the objects he wants to test, the size of the grid, if he wants to use the grid-method or a step size for the spherical coordinates of the sphere-method. Moreover, a step size for the rotation of the moving object must be given and, finally, a distance. Then, our benchmark automatically generates a set of sample points for these specified parameters and benchmarks all available algorithms. It measures the times with an accuracy of 1 msec. Moreover, our benchmarking suite also offers scripts for the automatical generation of diagrams to plot the results of the benchmark.

5 RESULTS

Besides the distance between the objects, the performance of collision detection libraries mainly depends on the complexity and the shape of the objects. We used

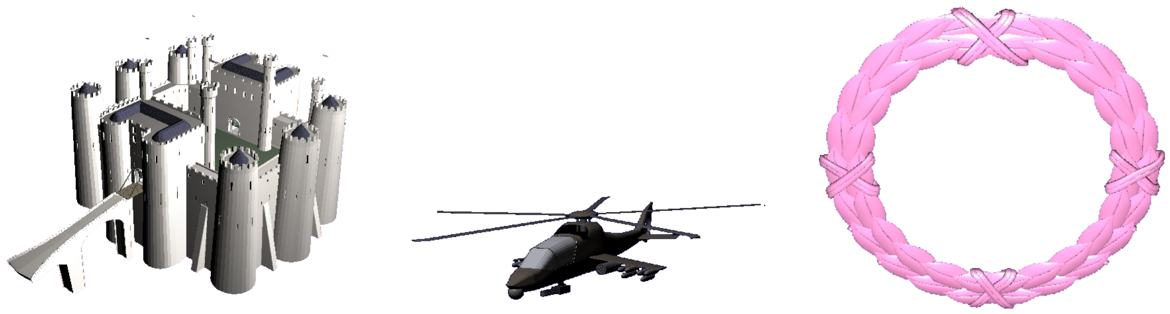
Figure 2: Some of the objects we used to test the collision detection libraries: A model of a castle, a helicopter and a laurel wreath

20 different objects in several resolutions in order to cover a wide range of use cases. All of the objects are in the public domain and can be accessed on our website. In particular, we used models of the Apollo 13 capsule and the Eagle space transporter, because they are nearly convex but have a lot of small details on the surface. To test the performance of the libraries on extremely concave objects we chose models of a helicopter, a luster, a chair, an ATST-walker and a set of pipes. Moreover, we used a laurel wreath to test intricate geometries. A Buddha model, a model of the Deep Space 9 space station, a dragon, and the Stanford Bunny were tested as examples of very large geometries. A model of a castle consists of very small, but also very large triangles. We used it to test the performance at unequal geometries. Accurate models of a Ferrari, a Cobra, and a door lock represent typical complex objects for industrial simulations. Finally, synthetic models of a sphere, a grid, a sponge, and a torus were used. Figures 2 and 3 show some of these objects.

Within our benchmarks, we simply tested a model against a copy of itself. However, our benchmark also supports the use of two different objects, but the first method is sufficient to make conclusions about the performance of the libraries.

We tested the libraries on a Pentium D CPU with 3 GHz and 1 GB of DDR2-RAM running Linux. All source code was compiled with gcc 4.0.2. We used the sphere-method with PQP for distance computing. We chose a step size of 15° for the spherical coordinates and a step size of 60° per axis for the rotations of the objects. With these values, we generated a set of 38000 sample configurations for every distance. We computed sample configurations for distances up to 40% of the object size in 1% steps, because in all example cases, there was no significant time spent on collision detection for larger distances. All these configurations can be downloaded from our web site[2].

The first reasonable finding of our measurements is that those algorithms, which use the same kind of BVH, behave very similar. Our second finding is that all algorithms have their special strength and weakness in different scenarios. E.g., the AABB-based algorithms like FreeSOLID, Opcode and the BoxTree were very well suited for regular meshes like the grid or the lustre, but also for meshes with very unequal triangle sizes, like the castle (see Fig. 7). In these cases, they were up to 4 times faster than the OBB-based libraries or the DopTree. This is because in these test cases, AABBs fit the objects very well and therefore, the algorithms can benefit from their faster collision check algorithm.

When we used extremely concave and sparse objects, like the lustre or the ATST, or objects with lots of small details, like the Apollo capsule, the situation changed completely and the OBB-based algorithms, namely PQP and V-Collide, performed much better than the AABB-based libraries (see Fig. 5). This is, because with these kinds of objects, a tight fitting BVH seems to gain more than a fast BV test.

A special role played the Dop-Tree which combines the fast BV tests of the AABB-based algorithms with the tight BVs of the OBB-based libraries. As expected, this BVH is placed between the other two kinds of algorithms in most of the test scenarios.

Another interesting aspect we wanted to benchmark is the dependency on the complexity of the objects. Therefore, we tested all our models in different resolutions. The surprising result was, that there was no general dependency on the complexity for the algorithms we tested. E.g., in the lustre scene, the times increase nearly linearly with the number of polygons, for the AABB-based libraries, whereas it is nearly constant for the OBB-based algorithms. In the grid scenario, the increase is about $O(n \log n)$ for all algorithms (see Fig. 6). In the castle scene, the collision detection time seems to be independent from the complexity and in the chair scene, the collision detection time decreased for all algorithms with an increasing object complexity (see Fig. 7).

Summarizing, there is no all-in-one device suitable for every purpose. Every algorithms has its own strength in special scenarios. Therefore, the users should check their scenario carefully when choosing a special collision detection algorithm. A good compro-

[2] http://cg.in.tu-clausthal.de/research/colldet_benchmark/

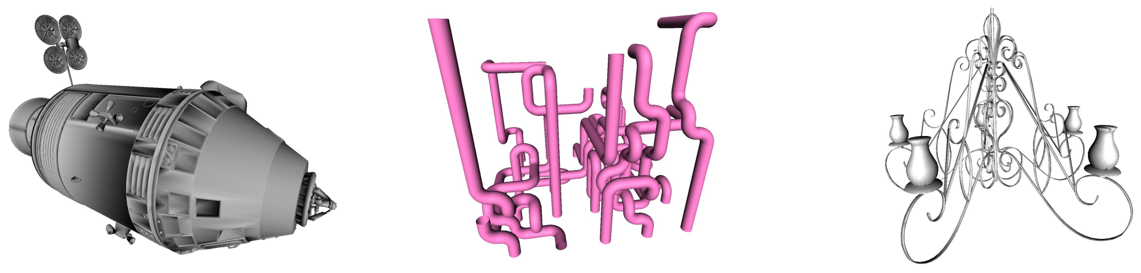
Figure 3: Some more of the test objects: A model of the Apollo 13 capsule, a set of pipes and a lustre.

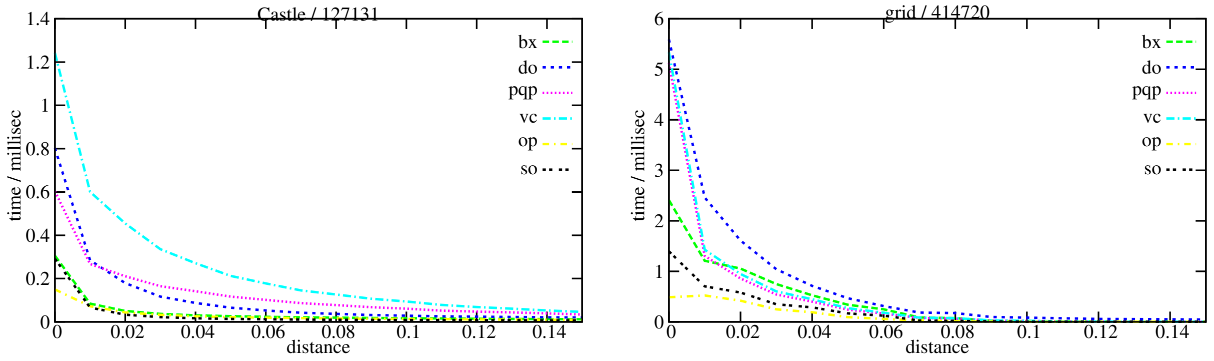

Figure 4: The results of the benchmark for the castle scenario in resolutions with 127 131 vertices and in the grid scene with 414 720 vertices. The x-axis denotes the relative distance between the objects, where 1.0 is the size of the object. Distance 0.0 means that the objects are almost touching but do not collide. The abbreviations for the libraries are as follows: bx=BoxTree, do=Dop-Tree, pqp=PQP, vc=V-Collide, op=Opcode, so=FreeSOLID. The AABB-based algorithms perform best in this kind of scenarios.

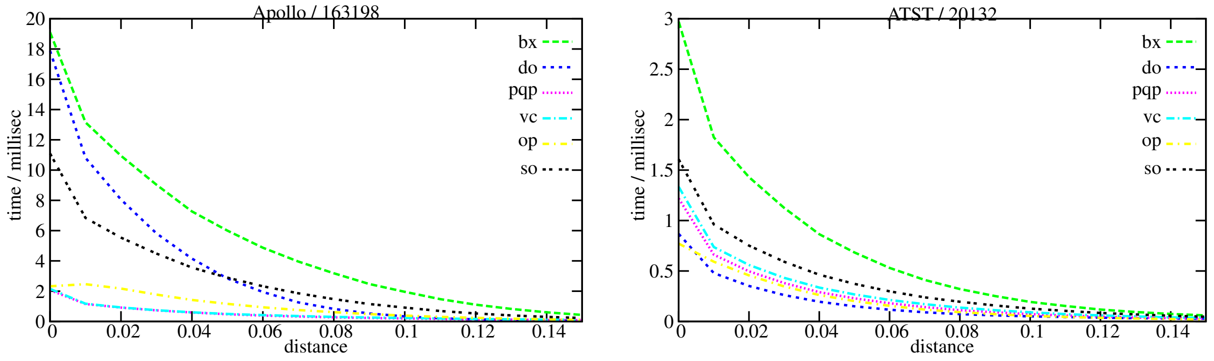

Figure 5: The results of the benchmark for the Apollo capsule with 163 198 vertices and the ATST walker with 20132 vertices. In these test cases, the OBB-based algorithms are much faster than the AABB-based libraries.

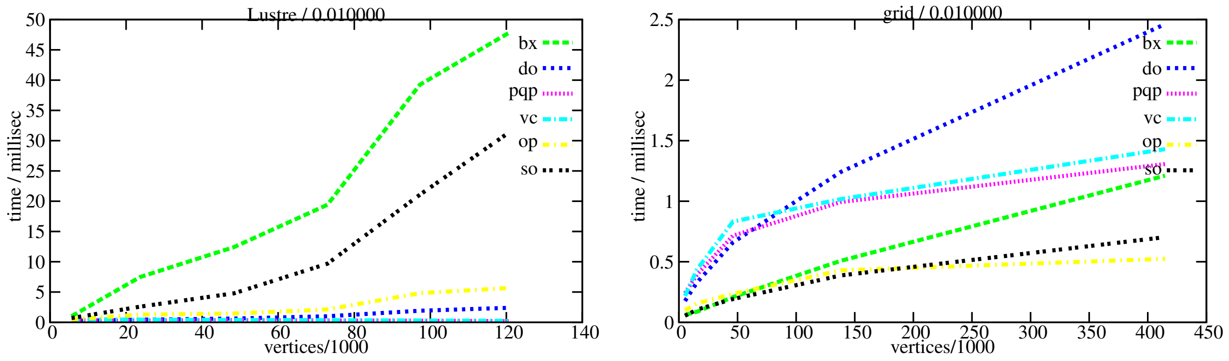

Figure 6: The results of the benchmark for the lustre scene and the grid scene for a distance of 1% relative to the object size. The x-axis denotes the number of vertices divided by 1000. The time for collision detection in the lustre scene increases nearly linearly for the AABB-based algorithms, whereas it seems to increase in $O(n \log n)$ for all algorithms in the grid scene.

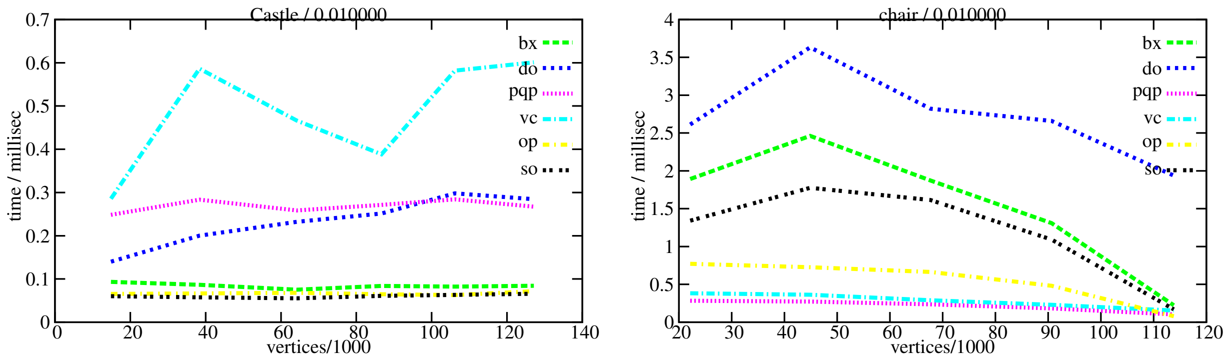

Figure 7: The dependency of the collision detection time from the complexity of the models in the castle and the chair scenes. The distance is fixed to 1% of the object size. In the castle scene, the collision detection time seems to be independent of the complexity, while in the chair scene, the time for collision detection even decreases with increasing complexity.

mise seems to be the Dop-Tree, because it combines tight BVs with fast BV tests. Moreover, in some cases, it could be helpful to increase the complexity of the model in order to decrease the time for collision detection, but this does not work in all cases. However, in nearly all test cases, all libraries are fast enough to perform real time collision checks even for very complex objects.

6 CONCLUSIONS

We presented an easy to use benchmarking method and a representative suite for benchmarking objects for static collision detection algorithms for rigid objects. Our benchmark is robust, fast, flexible, and it is easy to integrate other collision detection libraries. We used our benchmarking suite to test several freely available collision detection libraries with a wide variety of objects.

Our benchmarking suite is helpful for users to figure out the best fitting collision detection scheme to meet their specific requirements. The comparison of several algorithms yields a simple rule for choosing the optimal algorithm.

In the future, we plan to extend our benchmarking suite also for penetrating objects. Therefore, we will have to determine the penetration depth of a pair of objects. Another promising future project is the design of a benchmarking suite for more than 2 objects and for continuous collision detection algorithms. Moreover, a standardized benchmarking suite for deformable objects is still missing and could be very helpful for users.

REFERENCES

[CRM02] Stefano Caselli, Monica Reggiani, and M. Mazzoli. Exploiting Advanced Collision Detection Libraries in a Probabilistic Motion Planner. In *WSCG*, pages 103–110, 2002.

[GLM96] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A Hierarchical Structure for Rapid Interference Detection. *Computer Graphics*, 30(Annual Conference Series):171–180, 1996.

[HLC$^+$97] Thomas C. Hudson, Ming C. Lin, Jonathan Cohen, Stefan Gottschalk, and Dinesh Manocha. V-COLLIDE: Accelerated Collision Detection for VRML. In *VRML 97: Second Symposium on the Virtual Reality Modeling Language*, Rikk Carey and Paul Strauss, Eds. ACM Press, New York City, NY, 1997.

[LGLM99] E. Larsen, S. Gottschalk, M. Lin, and D. Manocha. Fast proximity queries with swept sphere volumes. In *Technical Report TR99-018*, 1999.

[OL03] Miguel A. Otaduy, and Ming C. Lin. CLODs: Dual Hierarchies for Multiresolution Collision Detection. In *Symposium on Geometry Processing*, pages 94–101, 2003.

[Ter01] Pierre Terdiman. Memory-optimized bounding-volume hierarchies. 2001. `http://www.codercorner.com/Opcode.htm`.

[vdB97] Gino van den Bergen. Efficient Collision Detection of Complex Deformable Models using AABB Trees. *Journal of Graphics Tools: JGT*, 2(4):1–14, 1997.

[vdB99] Gino van den Bergen. A Fast and Robust GJK Implementation for Collision Detection of Convex Objects. *Journal of Graphics Tools: JGT*, 4(2):7–25, 1999.

[Zac95] G. Zachmann. The BoxTree: Exact and Fast Collision Detection of Arbitrary Polyhedra. In *SIVE Workshop*, pages 104–112, July 1995.

[Zac98] Gabriel Zachmann. Rapid Collision Detection by Dynamically Aligned DOP-Trees. In *Proc. of IEEE Virtual Reality Annual International Symposium; VRAIS '98*, pages 90–97. Atlanta, Georgia, March 1998.