Optimizing the Collision Detection Pipeline

Gabriel Zachmann
Fraunhofer Institute for Computer Graphics
Rundeturmstraße 6
64283 Darmstadt, Germany
email: zach@igd.fhg.de

Abstract

A general framework for collision detection is presented. Then, we look at each stage and compare different approaches by extensive benchmarks. The results suggest a way to optimize the performance of the overall framework.

A benchmarking procedure for comparing algorithms checking a pair of objects is presented and applied to three different hierarchical algorithms. A new convex algorithm is evaluated and compared with other approaches to the neighbor-finding problem.

Keywords: Convex hulls, incremental collision detection, hierarchical data structures, grid, octree.

1 Introduction

Collision detection is one of the enabling technologies for many types of physically-based simulation. Such simluations become increasingly important in VR applications like virtual prototyping in order to make them functionally more rich and mature. Other areas are animation systems, robotics, and games.

While there is a wealth of literature on collision detection algorithms, only few papers describe and evaluate a complete collision detection system. In addition, there are only few evaluations as to which algorithms should be combined in order to maximize performance.

In this paper we will present the collision detection pipeline which serves as a framework for collision detection systems (see Section 3). It is generic in that most algorithms presented in the literature can be applied in the respective stage.

We will then look at each stage of the pipeline and evaluate and compare various approaches. Section 4 evaluates several hierarchical algorithms, while Section 5 presents different approaches to neighbor-finding, including a new algorithm for convex objects. Finally, results on parallelization are presented.

In the following, all experiments have been carried out on a SGI R10000 194 MHz. All times are (unless otherwise noted) in millseconds.

2 Related work

Research so far has mainly focused on algorithms for the collision detection problem, given a pair of objects. Hierarchical algorithms for polygonal objects have been presented by [GLM96, KHM$^+$98, Zac98, Hub95, OD99, NAT90]. Non-hierarchical approaches have been presented by [MPT99, Gei00]. Incremental convex algorithms have been presented by [CLMP95, vdB99, Chu96].

Algorithms for reducing the number of pairs to be tested have been presented by [CLMP95, BF79, PS90, YW93, Hub93].

A system solving the dynamic pre-fetching of objects and auxiliary data, if the complete environment does not fit in main memory, has been described by [WLML99], while [AGT99] have presented a collision detection system for haptic rendering.

3 The pipeline

Collision detection can be regarded as a pipeline of successive filters. This concept is somewhat similar to the concept of a rendering pipeline ([AJ88, Bar97, MEP92]) or visualization pipeline. The input of the collision detection pipeline is a set of objects, while the output is a set of pairs of objects (and possibly polygons).

The front end of the pipeline consists mainly of a queue which stores objects, registration commands, and registration queries. Modules interested in collisions will enter commands and objects registering them with

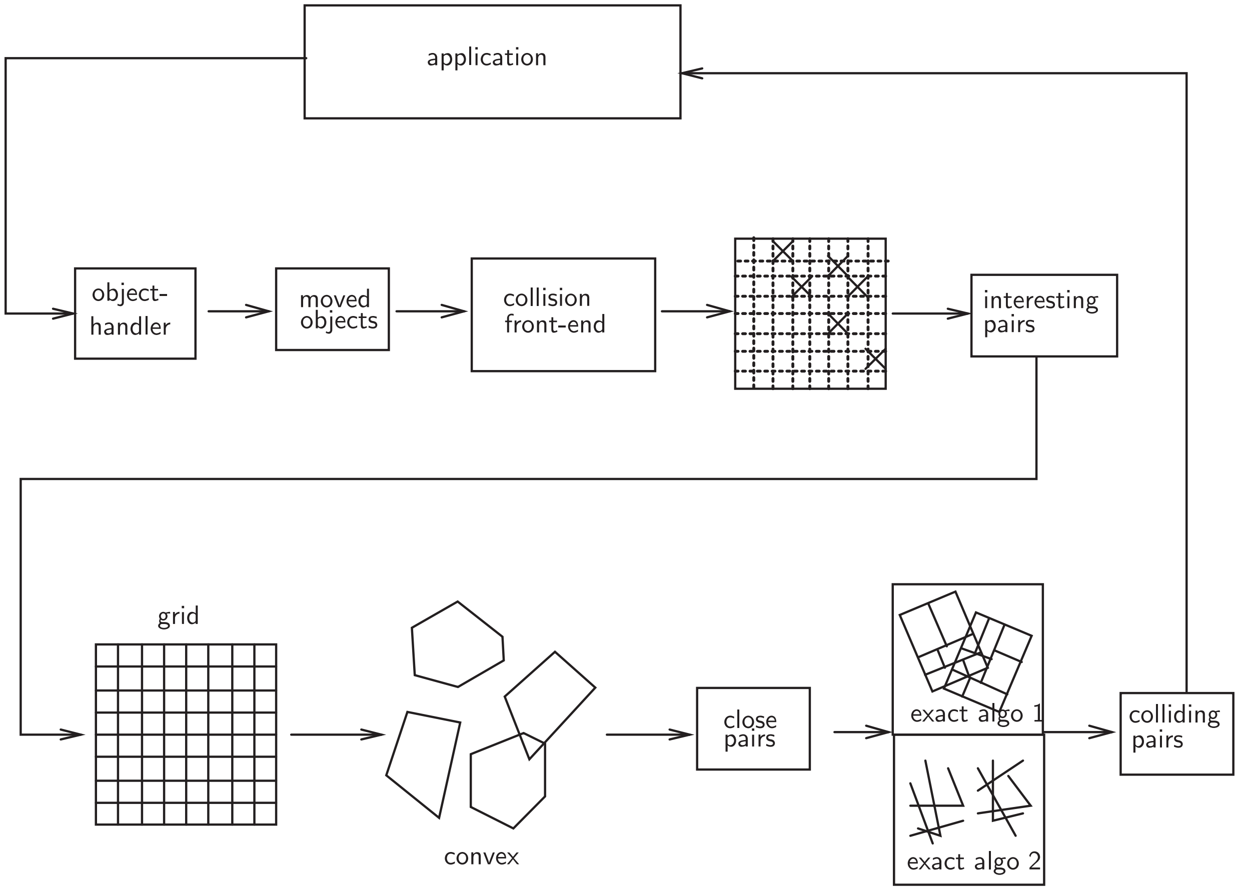

Figure 1: Collision detection can be considered as a pipeline of successive filters for pairs of objects. The front end consumes single objects while the back end produces pairs of colliding objects.

the collision detection module. This is usually done during the initialization of the system, but can be useful at run-time, too. Likewise, modules can register and unregister interest in the collision of pairs of objects at any time.

During run-time, objects, together with their current position, are entered in the queue when they have been moved. Exactly when they are entered is determined by either of two cases: either the object handler does it sort of "anonymously" whenever a transformation of that object is changed, or each module does it itself when a new position of the object has been calculated. The former case is usually more convenient if an object can be moved by different independent modules, while the latter case is more convenient if an object is solely under the control of one module.

Conceptually, the front end passes on all possible pairs of objects down the collision detection pipeline (see Figure 1). After the front end, the collision interest matrix filters out all object pairs of no interest. After that, object pairs are filtered further by two neighbor-finding stages and finally by an exact collision test (polygonal collision algorithms can be considered another filter before the polygon-polygon test). Why and when we propose two neighbor-finding stages will become clear in Section 5.

In the following, we will look at each stage in more detail, beginning with the back end.

4 Object level

At this level, we are given one pair of objects and have to determine their collision status. Research has shown that hierarchical algorithms can solve this problem very efficiently.

Although a lot of hierarchical algorithms have been published in recent years, it has not been clear how they

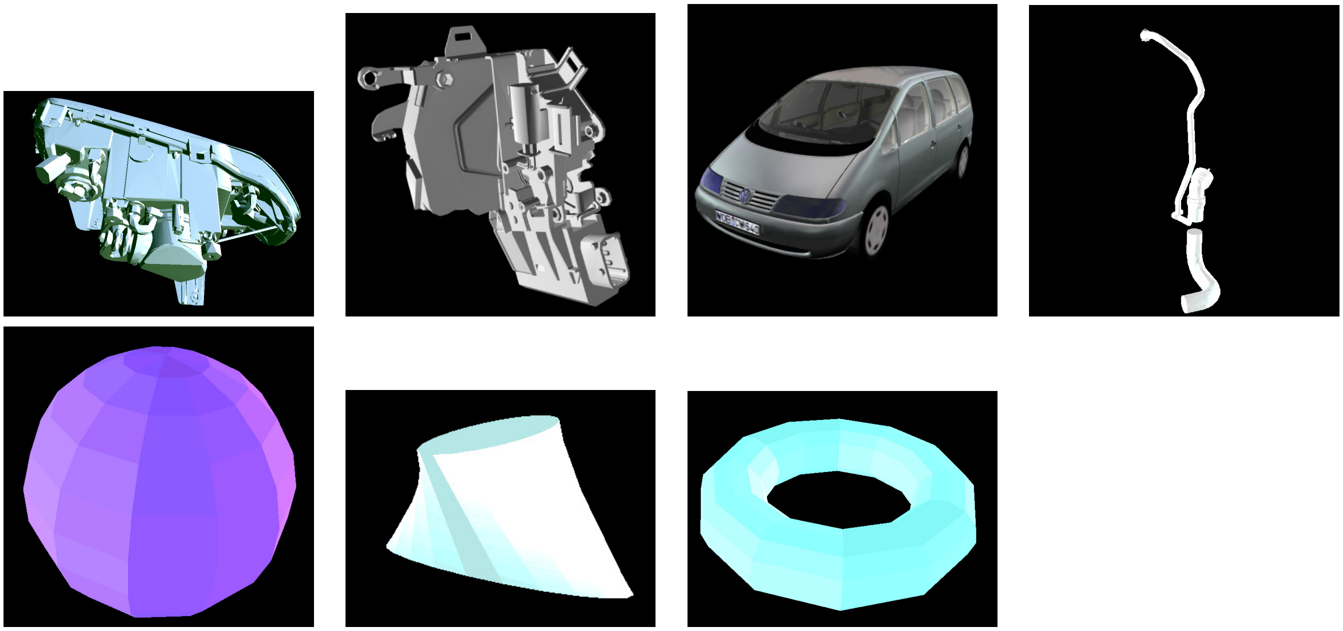

Figure 2: Our suite of test objects. They are (left to right): a car headlight, the lock of a car door, body and seats of a car, hose of a car engine, sphere, hyperboloid, torus. (Data courtesy of VW and BMW)

compare to each other. In order to optimize the collision detection pipeline, one would like to know which algorithm to use at the back end. Maybe there is no single best algorithm, and different algorithms perform best for different complexities or types of objects.

In this section, we will compare three algorithms: OBB-trees [GLM96], BV-trees of k-dops [KHM^{+}98], and DOP-Trees [Zac98]. The latter two use the same type of bounding volume, but different algorithms for checking the overlap of two BVs. The former uses oriented bounding boxes. In contrast to [Zac98], we have determined 24 as the optimal number of orientations. For the comparison, we used the implementations Rapid and QuickCD [Got97, KHM99].

We have chosen these three algorithms because they work on the same polygonal representation of objects. If algorithms using different representations are to be compared (such as those utilizing point clouds [MPT99, Gei00]), then great care must be exercised so that the error induced by the different representations is the same with respect to the original curved surfaces.

4.0.1 Benchmark procedure

We have found, that it is extremely difficult to compare collision detection algorithms, because in general they are very sensitive to conditions and scenarios, such as the relative size of the two objects, the relative position to each other, the distance, etc. Even the orientation of the object with respect to its object frame can have a significant impact on the BV-tree and hence on the efficiency [Zac00, Fig. 3.36].

Our test scenario involves two identical objects which are positioned at a certain distance $d = d_{\mathsf{start}}$ from each other. The distance is computed between the centers of the bounding boxes of the two objects; objects are scaled uniformly so they fit in a cube of size 2^3. Then, one of them performs a full tumbling turn about the z- and the x-axis in a fixed, large number of small steps (here 2000). With each step a collision query is done, and the average collision detection time for a complete revolution at that distance is computed. Then, d is decreased, and a new average collision detection time is computed, which yields graphs such as those shown in `http://www.igd.fhg.de/~zach/coldet/index.html`.

This procedure is repeated for several different initial object orientations, i.e, object frames, which can make a difference when constructing the BV tree. The object frame is rotated about the axis $(1, 1, 1)$ so as to neutralize "alignments" of the geometry. When plotting the average collision detection time, we average over all "interesting" distances and all object frames. Here, we have chosen the range from 2 distance steps *before* the

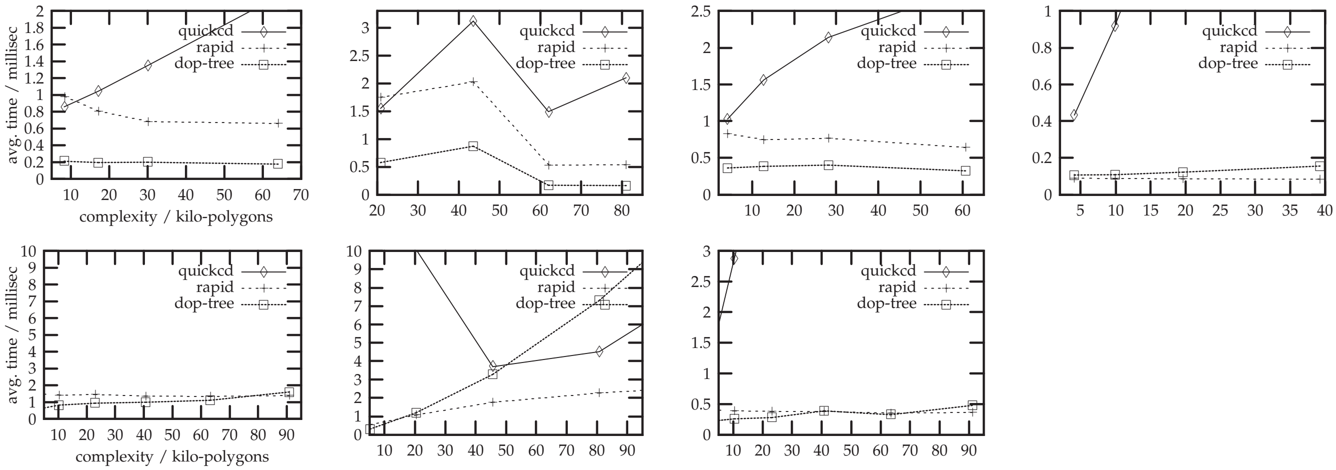

Figure 3: Results for the suite of benchmark object as shown in Figure 2 (headlight, headlight, door lock, car body, hose, sphere, hyperboloid, torus). All times have been obtained on a R10000 194MHz, averaging over distance, orientation, and object frame.

contact distance through 2 steps *after*. We believe this reflects representative situations for collision detection.

We have carried out extensive experiments with both synthetic and real-world CAD objects (see Figure 2). All timings include vertex and normal transforms.

4.1 Results

Figure 3 shows the results of our benchmark suite. The QuickCD algorithm performs much worse than Rapid and our DOP-tree. In addition, it depends much more on the number of polygons. In contrast, Rapid and DOP-tree depend very little on the number of polygons, where Rapid depends less; in some cases, collision detection time even decreases slightly as the number of polygons increases. For most objects, the DOP-tree is faster, for some it is significantly faster, while Rapid is slightly faster for others.

With one object (the door lock), there is a remarkable decrease in collision detection time; we are not sure why that is, but we suspect this is because with a finer tesselation the BV-tree construction algorithms get a better chance of producing good BV hierarchies. Another reason might be that with this particular object the bulk of the polygons are interior ones.

5 Global level

At the global level, we have the "all-pairs" problem similar to the object level. Here, the basic primitives are objects. So, in an early stage of the pipeline we need a "neighbor-finding" filter (or several). At this stage, objects must be represented by some simple bounding volume, usually its bounding box or convex hull. Unfortunately, methods suitable for the object level, where the primitives are polygons, are not suitable here, because usually objects maintain no spatial relationship to each other.

There are basically two methods to solve the problem, both of which exploit the assumption of a fairly uniform distribution of objects throughout the "universe". By this assumption, each object has a small number of "neighbors" (in some sense). One class of methods utilizes space partitioning data structures (possibly hierarchical ones), the other class is "object-oriented" in the sense that objects are maintained in a hierarchy or sorted in some order.

A hierarchical approach has been proposed by [YW93]. The idea is to improve neighbor-finding performance by a hierarchy of bounding volumes such as the scene graph. The complexity is at least $n \log n$ bounding volume tests. The problem with this approach is that hierarchies generally need to be rearranged as objects move around in order to stay efficient. Because the hierarchy degrades all over, the hierarchy cannot be updated incrementally.

The approach pursued by [CLMP95, BF79, PS90] is the sweeping plane. It works at "object precision" and has a complexity of at least $o(n)$. Since bounding volumes are sorted along one axis, the data structures lend itself quite well to incremental updates. A problem is

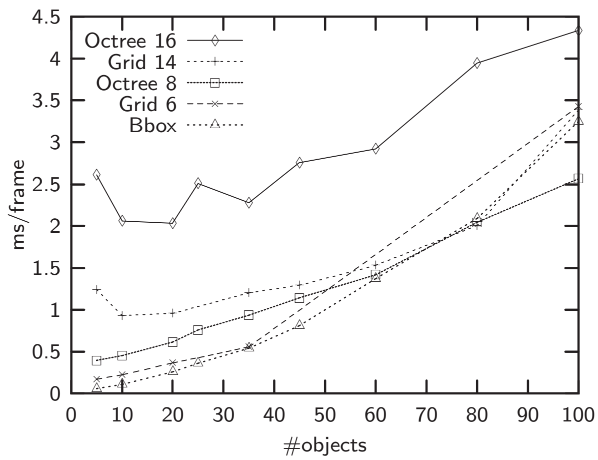

Figure 4: Comparison of grid, octree, and $\frac{n^2}{2}$ bounding box tests. The graph labeled "grid 14" corresponds to a grid with 14^3 cells; similarly for the other graphs.

that the sweep plane could potentially intersect many bounding volumes although they are very far from each other.

We have evaluated two space indexing approaches, namely the grid and octree [Ove88, GA93, MSH+92, HT92]. The advantage of space indexing approaches is that they work "locally" (defined by the cell size), with the price that they do not work at object precision but at cell precision. Our implementations of grid and octree work incremental so that only those cells have to be visited which need to be changed [Zac00]. The complexity depends on the size of objects, the granularity of the data structure, and the number of neighbors.

Figure 4 shows the results of a coparison of octrees, grids, and the naïve n^2-method. The scenario is n objects moving inside a cube (no exact collision detection). In order to keep the density[1] constant as the number of objects increases, the size of the cube has been increased accordingly.

In dynamic environments, octrees seem to be always less efficient than grids. This is in contrast to results obtained by [MSH+92] for ray tracing, which suggest octrees in favor over grids. Another result is that with very small numbers of objects, the n^2-method performs better than the grid.

[1] Here, density is defined as the average number of bounding volume overlaps per frame.

5.1 Convex hull test

Convex hulls are much tighter bounding volumes than bounding boxes, so we tried to determine whether or not a convex hull test would gain any performance.

We have chosen to use a convex hull algorithm which trades accuracy for speed, i.e., it is probabilistic. Since in our case there is always an exact collision detection check after the convex hull test, this does not introduce wrong results.

The convex algorithm We have developed an algorithm which just needs the set of vertices of the convex hull and its adjacency map. The algorithm is based on the notion of linear separability: P and Q do not intersect iff there is a plane h such that all vertices of P and Q are on different sides. Such a plane is called a *separating plane*. Let $P = \{p^1, \ldots, p^n\}, Q = \{q^1, \ldots, q^m\} \subseteq \mathbb{R}^3$; then, P and Q are linearly separable, iff $\exists w \in \mathbb{R}^3, w_0 \in \mathbb{R}\, \forall i, j : (p^i, -1) \cdot (w, w_0) > 0\ ,\ (-q^j, 1) \cdot (w, w_0) > 0$.

The algorithm is based on the perceptron learning rule [HKP91]. Let $Z = \{z^k\} := \{(p^i, -1), (-q^j, 1)\}$ be the set of vertices, and w^0 an initial plane. If w^l is not a separating plane, i.e., $\exists z : z \cdot w^l < 0$, then a new plane is computed by $w^{l+1} := w^l + \eta \cdot z$. η is a "temperature" which is used for simulated annealing. The algorithm terminates when a separating plane has been found, or when the maximum number of loops has been reached.

By saving the plane w for each pair of objects, this algorithm can be turned into an incremental algorithm very easily. In addition, checking whether or not a plane is separating can be done very quickly by hill-climbing, because the two sets of vertices are known to be convex.

We have found that a maximum number of 300 loops is sufficient, i.e., the algorithm either has determined a separating plane or it will never determine one, no matter how many more loops are being performed.

Comparison with Lin-Canny We compared the separating planes algorithm with the Lin-Canny algorithm as implemented in I_collide [CLM+]. The benchmarking procedure is comprised of two spheres, one of them stationary, the other orbiting around the first with various distances. For the separating planes algorithm, the maximum number of steps was set to 300. Times are averaged over 5,000 samples for each distance. At all distances, there were only 0–3 wrong results, except for distance 2.000060 which yielded 1256 wrong results.

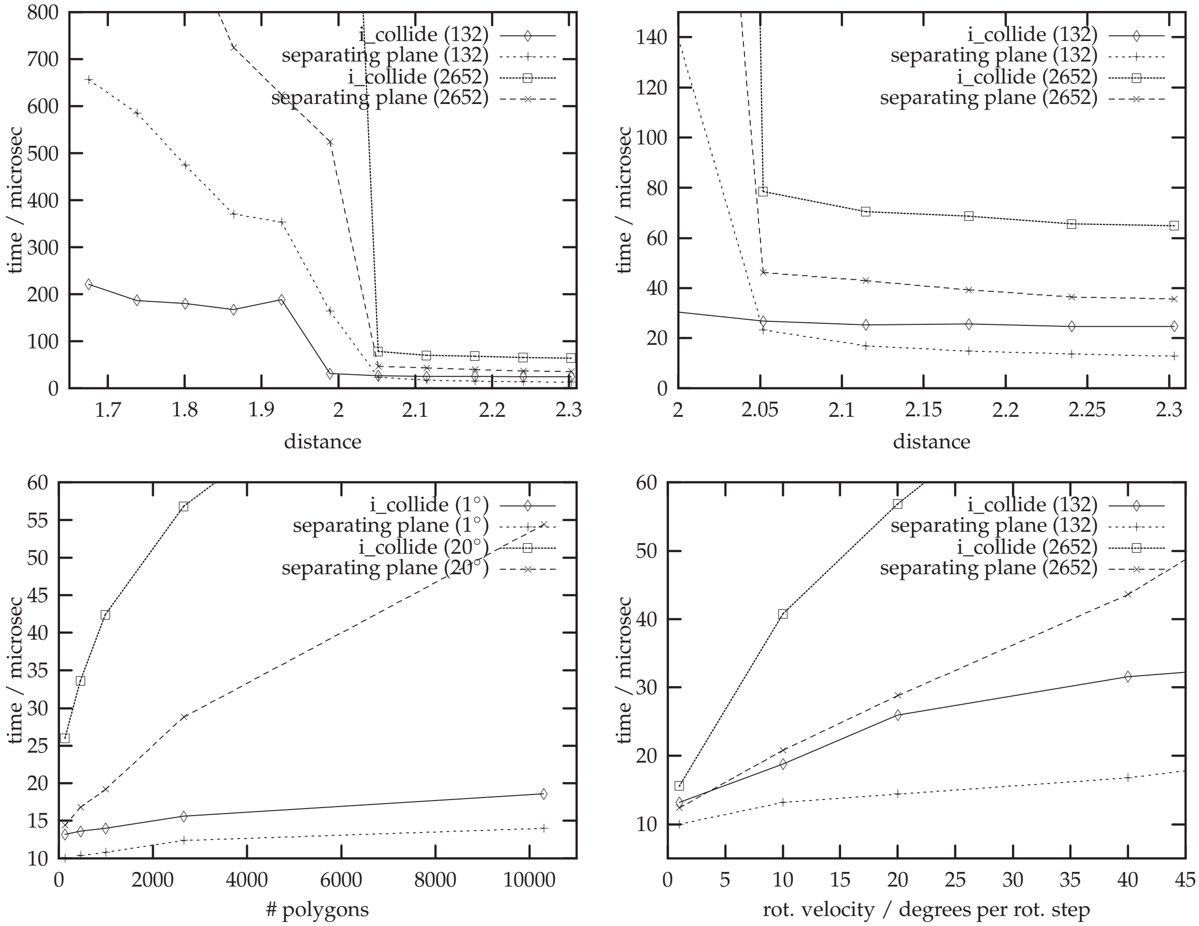

Figure 5: Comparison of I_collide and our separating planes algorithm with respect to distance, number of polygons, and rotational velocity. In the upper two graphs, the numbers in parentheses denote the number of polygons of each object. In the lower left graph, the numbers in parentheses denote rotational velocity. In the lower right graph, the numbers denote polygon count.

Figure 5 shows how the two algorithms depend on various parameters, namely distance, number of polygons, and rotational velocity.

In addition to being about twice as fast, it seems that the separating plane algorithm is much more robust than I_collide [Zac00, Sec. 3.4.3]. Maybe these problems persist because Voronoi regions are magnified a little bit in order to avoid other problems.

5.2 Comparison of neighbor-finding algorithms

In order to determine the optimal neighbor-finding algorithm, one has to benchmark each one *with* subsequent exact collision detection,[2] because there are a lot of interactions.

On the one hand, grids are essentially in $O(n)$, while the separating planes algorithm and I_collide are in $O(n^2)$. On the other hand, there is quality: by this we understand how often a neighbor-finding algorithm passes a pair of objects to the exact collision detection although they do not collide. Obviously, the grid has a lower quality than convex hull based algorithms. Another property is the dependence on object complexity: the grid is independent of object complexity, while convex hull based algorithms usually depend on it.

[2] If one would do a timing of only the neighbor-finding algorithms, then the trivial one, which passes *all* pairs on to the exact collision detection, would win.

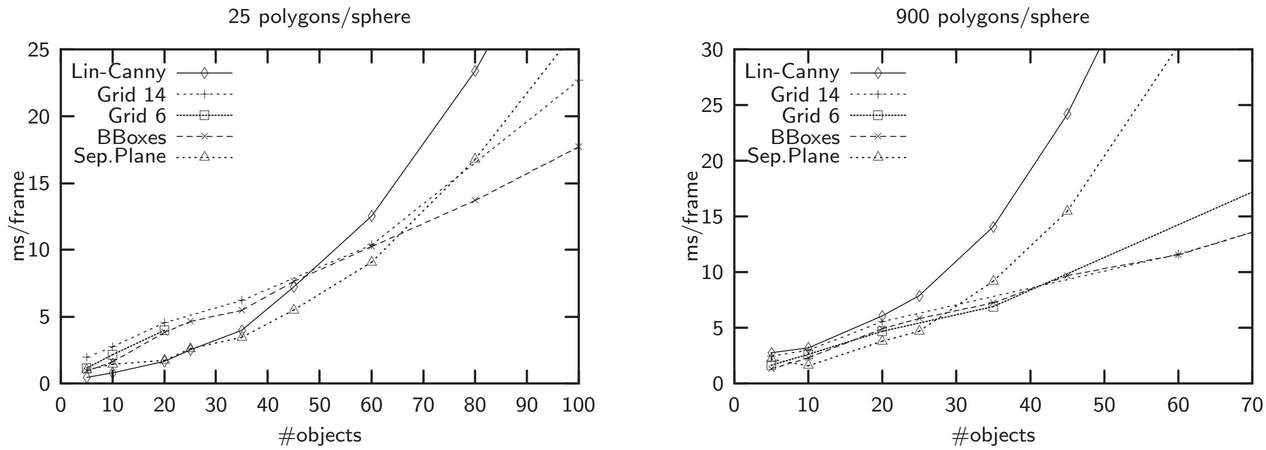

Figure 6: A comparison of various neighbor-finding algorithms. Exact collision detection (DOP-Tree) has been performed if a pair was found to be "close enough", and it is part of the timing. Scenario: n spheres bouncing off each other inside a cube. The density (defined as the number of collisions per frame) is constant through all object counts.

Figure 6 shows that there is a break-even point: with only very few objects, the convex hull based algorithms perform better because of their better quality; with many objects, the grid performs better because of its $O(n)$ complexity. It seems also, that for few objects with very low complexity, the Lin-Canny algorithm is better suited than the separating planes algorithm for neighbor-finding. Surprisingly, the brute-force approach testing $\frac{n^2}{2}$ bounding boxes performs very well; it seems to have a very small constant factor, so that it will exhibit the n^2-characteristic only with many more objects than 100.

5.3 Implications

Assuming that objects have at least a moderate polygon count ($\geq$ 1000), the collision detection pipeline should implement both the separating planes algorithm and a grid for the neighbor filtering stage.

With 5 or less objects, the naïve approach testing $\frac{n^2}{2}$ bounding boxes should be used. With 10–30 objects, the separating plane algorithm alone should be used. And with more than 50 objects a grid should be used. Of course, these thresholds will depend somewhat on object complexity.

Since grids are outperformed by the separating planes algorithm with low object numbers, we believe that a further performance gain can be achieved by combining the two: object pairs passing the former should be further filtered by the latter. Assuming that the first neighbor-finding stage needs time $T_s(n)$, i.e., assuming it has inear complexity,[3] such an approach is more efficient if

$$P_s(n) < \frac{n^2}{2} - \frac{T_s(n)}{T_p}$$

where $P_s(n)$ is the number of pairs passing the first neighbor-finding stage (like the grid) and T_p is the time needed for one neighbor-test of the second stage (like a convex algorithm).

6 Parallelization

The collision detection pipeline offers several possibilities of parallelization: pipelining, concurrency, coarse- and fine-grain parallelization. We have not yet investigated a parallel pipeline, i.e., running each stage of the pipeline concurrently to the others. Making the whole collision detection pipeline concurrent to other modules of the system is very easy: since our framework already provides a queue at the front-end, this only needs to be implemented as a double-buffer with access control.[4]

At the back-end, several pairs of objects can be checked simultaneously, using dynamic workload allocation, which we call coarse-grain parallelization. This yields very good speedups, provided there are always enough pairs passing through the neighbor-filtering

3 Grids meet this assumption.

4 We have, of course, implemented our collision detection module concurrently, and the results are entirely satisfying.

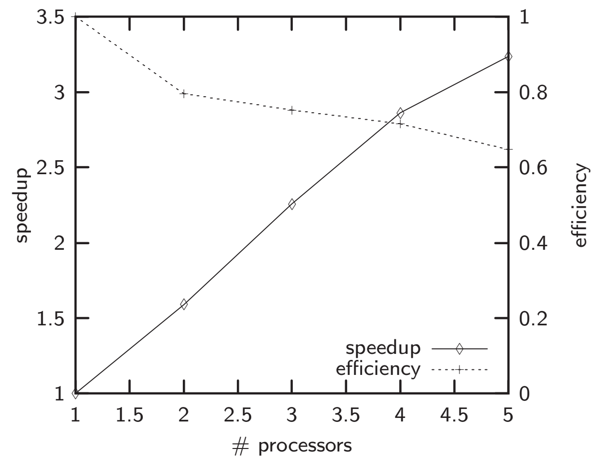

Figure 7: Coarse-grain parallelization of the back-end yields good speed-ups, if there are enough object pairs to be checked with each collision frame.

stage, i.e., the environment is dense enough. Figure 7 shows some timing results for a scenario where several objects are bouncing off each other inside a cube. The timing includes all stages of the collision detection pipeline, although only the back-end has been parallelized. It has been carried through on a 6-processor 194 MHz R10000 Onyx.

Sometimes, only one object pair passed through the neighbor-filter. In that case, a parallel version of the exact collision detection algorithm itself should be used. For hierarchical algorithms, a dynamic load-balancing scheme should be used, because different branches of the BV-tree need to be descended to highly different levels. The implementation must be careful, otherwise synchronization overhead will be too high.

7 Conclusion and future work

We have presented the pipeline as a general framework for collision detection systems. This pipeline consists of several stages, filtering lists of pairs of objects.

We have compared three different hierarchical collision detection algorithms. Our extensive experiments suggest that the DOP-tree algorithm [Zac98] is to be preferred over Rapid [GLM96] and QuickCD [KHM+98] in most cases, where it is about a factor 2 faster.

A probabilistic algorithm for collision detection of convex objects has been presented and compared to the Lin-Canny algorithm [LC92]. We have found that our probabilistic algorithm performs about twice as fast, while producing only about ‰ wrong answers.

In order to evaluate the importance of the quality of a neighbor-finding method versus its speed, we have compared four different methods. Our experiments indicate that with few objects, a more precise but a bit slower convex algorithm should be preferred, while for large object numbers, the grid or bounding box method performs better.

Finally, we have evaluated the performance of a parallelized back-end of the pipeline. Our results show that if the density of the environment is large compared to the number of processors, then good speed-ups can be yielded, although we have not parallelized the rest of the pipeline.

So far, our parallelization still contains some sequential sections and barrier synchronizations. It would be interesting to investigate further speed-ups by other schemes such as a parallel pipeline, or by extending our scheme to all stages of the pipeline.

For the neighbor-finding stage, the sweep-and-prune algorithm proposed by [CLMP95] would be interesting to compare to the algorithms considered in this paper. In addition, we would like to verify our hypothesis explained above, that combining a grid with a subsequent convex algorithm (such as our separating planes algorithm) would improve the performance of the neighbor-finding stage for large object numbers.

Besides comparing further hierarchical algorithms using our benchmark suite (like sphere trees [Hub95]), non-hierarchical algorithms like [MPT99, Gei00] should be considered as well. In order to do that fairly, errors resulting from different representations must be taken into account.

References

[AGT99] A. Gregroy, M. Lin, S. G., and Taylor, R. A framework for fast and accurate collision detection for haptic interaction. In *Proceedings of IEEE Virtual Reality Conference*, 1999.

[AJ88] Akeley, K., and Jermoluk, T. High-performance polygon rendering. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, J. Dill, Ed., vol. 22, pages 239–246, August 1988.

[Bar97] Barkans, A. C. High-quality rendering using the talisman architecture. In *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, S. Molnar and B.-O. Schneider, Eds., pages 79–88. ACM SIGGRAPH / Eurographics, ACM Press, New York City, NY, August 1997. ISBN 0-89791-961-0.

[BF79] Bentley, J. L., and Friedman, J. H. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, December 1979.

[Chu96] Chung, K. Quick collision detection of polytopes in virtual environments. In *Proc. of the ACM Symposium on Vir-*

tual Reality Software and Technology (VRST'96), M. Green, Ed., pages 125–131, 1996.

[CLM+] Cohen, J., Lin, M. C., Manocha, D., Mirtich, B., Ponamgi, M. K., and Canny, J. I_COLLIDE. URL `http://www.cs.unc.edu/~geom/I_COLLIDE.html`. Software.

[CLMP95] Cohen, J. D., Lin, M. C., Manocha, D., and Ponamgi, M. K. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *1995 Symposium on Interactive 3D Graphics*, P. Hanrahan and J. Winget, Eds., pages 189–196. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7.

[GA93] Gargantini, I., and Atkinson, H. H. Ray tracing an octree: Numerical evaluation of the first intersection. *Computer Graphics forum*, 12(4):199–210, 1993.

[Gei00] Geiger, B. Real-time collision detection and response for complex environments. In *Computer Graphics International*. Geneva, Switzerland, June19-23 2000.

[GLM96] Gottschalk, S., Lin, M., and Manocha, D. OBB-Tree: A hierarchical structure for rapid interference detection. In *SIGGRAPH 96 Conference Proceedings*, H. Rushmeier, Ed., Annual Conference Series, pages 171–180. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[Got97] Gottschalk, S. Rapid library, 1997. URL `http://www.cs.unc.edu/~geom/OBB/OBBT.html`. Vers. 2.01.

[HKP91] Hertz, J., Krogh, A., and Palmer, R. G. *Introduction to the Theory of Neural Computing*. Addison-Wesley, 1991.

[HT92] Hsiung, P.-K., and Thibadeau, R. H. Accelerating ARTS. *The Visual Computer*, 8(3):181–190, March 1992.

[Hub93] Hubbard, P. M. Interactive collision detection. In *IEEE Symposium on Research Frontiers in VR, San José, California*, pages 24–31, October 25–26 1993.

[Hub95] Hubbard, P. M. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, September 1995. ISSN 1077-2626.

[KHM+98] Klosowski, J. T., Held, M., Mitchell, J. S., Sowrizal, H., and Zikan, K. Efficient collision detection using bounding volume hierarchies of *k*-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, January 1998.

[KHM99] Klosowski, J., Held, M., and Mitchell, J. QuickCD, software library for efficient collision detection, 1999. URL `http://www.ams.sunysb.edu/~jklosow/quickcd/QuickCD.html`. Vers. 1.00.

[LC92] Lin, M. C., and Canny, J. F. Efficient collision detection for animation, September 1992.

[MEP92] Molnar, S., Eyles, J., and Poulton, J. Pixelflow: High-speed rendering using image composition. *Proc. SIGGRAPH '92 Computer Graphics*, 26(2):231–240, July 1992.

[MPT99] McNeely, W. A., Puterbaugh, K. D., and Troy, J. J. Six degrees-of-freedom haptic rendering using voxel sampling. *Proceedings of SIGGRAPH 99*, pages 401–408, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[MSH+92] McNeill, M. D. J., Shah, B. C., Hébert, M.-P., Lister, P. F., and Grimsdale, R. L. Performance of space subdivision techniques in ray tracing. *Computer Graphics forum*, 11(4): 213–220, 1992.

[NAT90] Naylor, B., Amanatides, J. A., and Thibault, W. Merging BSP trees yields polyhedral set operations. *Comput. Graph.*, 24(4):115–124, August 1990. Proc. SIGGRAPH '90.

[OD99] O'Sullivan, C., and Dingliana, J. Real-time collision detection and response using sphere-trees. In *15th Spring Conference on Computer Graphics*, pages 83–92. Budmerice, Slovakia, April 1999. ISBN 80-223-1357-2.

[Ove88] Overmars, M. H. Efficient data structures for range searching on a grid. *J. Algorithms*, 9:254–275, 1988.

[PS90] Preparata, F. P., and Shamos, M. I. *Computational Geometry: An Introduction*. Springer-Verlag, 3rd ed., October 1990. ISBN 3-540-96131-3.

[vdB99] van den Bergen, G. J. A. *Collision Detection in Interactive 3D Computer Animation*. PhD dissertation, Eindhoven University of Technology, 1999.

[WLML99] Wilson, A., Larsen, E., Manocha, D., and Lin, M. C. Partitioning and handling massive models for interactive collision detection. In *Computer Graphics Forum, Eurographics'99*, vol. 18, pages 319–330. Blackwell Publishers, September 1999. ISSN 1067-7055.

[YW93] Youn, J.-H., and Wohn, K. Realtime collision detection for virtual reality applications. In *IEEE Virtual Reality Annual International Symposium*, pages 415–421, September 18–22 1993.

[Zac98] Zachmann, G. Rapid collision detection by dynamically aligned DOP-trees. In *Proc. of IEEE Virtual Reality Annual International Symposium; VRAIS '98*. Atlanta, Georgia, March 1998.

[Zac00] Zachmann, G. *Virtual Reality in Assembly Simulation — Collision Detection, Simulation Algorithms, and Interaction Techniques*. PhD dissertation, Darmstadt University of Technology, Germany, Department of Computer Science, May 2000. URL `http://www.igd.fhg.de/~zach/ftp/diss.html`.