Introduction

The most time consuming part in molecular dynamics simulations of arbitrary size is the collision detection. Usually, this problem is solved by restricting the shape of the particles to spheres. We will present an algorithm, originally developed for virtual reality visualizations by D.Baraff(1993) and M.C.Lin(1993), that allows the use of complex polyhedra (up to 920 faces and more). The expected run time is $\mathcal{O}(\mathrm{N})$, where N is the number of particles in the simulation. Neither complexity nor shape of the particles affect the run time.

How can this be achieved, if the run time for sorting general lists is $\mathcal{O}(\mathrm{N}\log\mathrm{N}+\mathrm{k})$? The crucial feature of the algorithm is to use information from the last time step.

The algorithm consists of two parts. In the first step, the algorithm looks for collisions of the particles bounding boxes. These are detected by resorting the list of all boxes. Insertion–sort allows sorting in $\mathcal{O}(\mathcal{N})$ operations. The second step is a fast method to compute the distance between two polyhedra by finding and tracking the closest points. In MD simulations, this algorithms is so efficient that the amount of required CPU time is independent of the particle shape.

Bounding Boxes

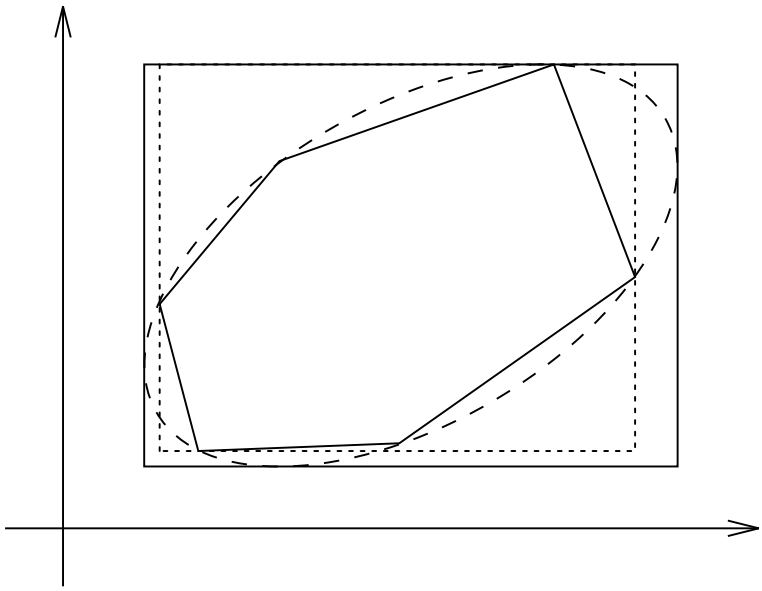

Figure 1: Different kind of bounding boxes for a Polygon. Choosing the "best one" asks to weigh up size versus calculation time

Generally any box B_A can be called a bounding-box of particle A, if A is inside of B_A ($A \subset B_A$). Here we assume rectangular bounding boxes, aligned with the coordinate axes.

The sense of using bounding boxes is to enable a fast check for NON-collision of two particles A and A'. If there is no point $x \in \mathcal{R}^3$ with $x \in B_A \wedge x \in B_{A'}$ then there is no point $x \in A \wedge x \in A'$ because of $A \subset B_A$,$A' \subset B_{A'}$.

The one-dimensional case

First we treat the problem in one dimension. The bounding boxes are now intervals $[b_i, e_i]$ for the ith particle. We have now to search for every pair of intersecting intervals $[b_i, e_i]$ and $[b_j, e_j]$

This can be solved with a sort and sweep algorithm (see Baraff(1993)) by generating a sorted list of all b_i and e_i values and sweeping the list. By means of this, one obtains a list of active intervals. When sweeping the list, you have to follow two rules:

1. If some value b_i is encountered, all intervals on the active list are known to be overlapping i and i is added to the active list.
2. If some value e_i is encountered, i is removed from the active list.

For the example in figure 2 we will get the list $b_8 - b_4 - b_5 - e_5 - b_2 - \ldots - e_7$ as input to the sort and sweep algorithm.

i	L_i	action	$\mathrm{L_{act.}(old)}$	collisions found
1	b_8	insert 8	—	—
2	b_4	insert 4	8	(4;8)
3	b_5	insert 5	$8-4$	(5;8), (5;4)
4	e_5	delete 5	$8-4-5$	no check
5	b_2	insert 2	$8-4$	(2;8) (2;4)
			⋮	
16	e_7	delete 7	7	no check

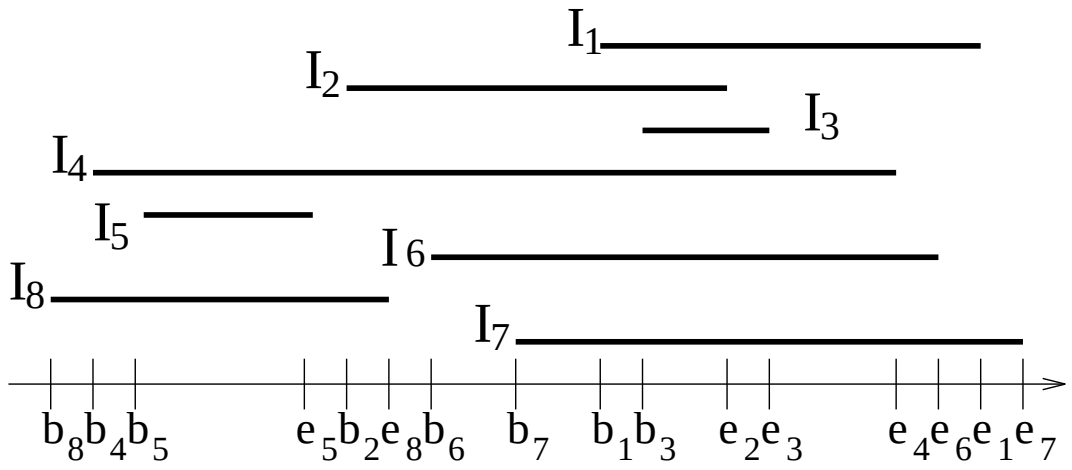

Figure 2: Example of intervals in one dimension.

The total cost of the process is $\mathcal{O}(n \log n)$ to sort the list, $\mathcal{O}(n)$ to sweep through the list, and $\mathcal{O}(k)$ to output each overlap. Obviously the problem's most time consuming part is the sorting algorithm.

In a MD-simulation, the particles will not move very far between two time steps, so the sorted list of intervals will not change dramatically.

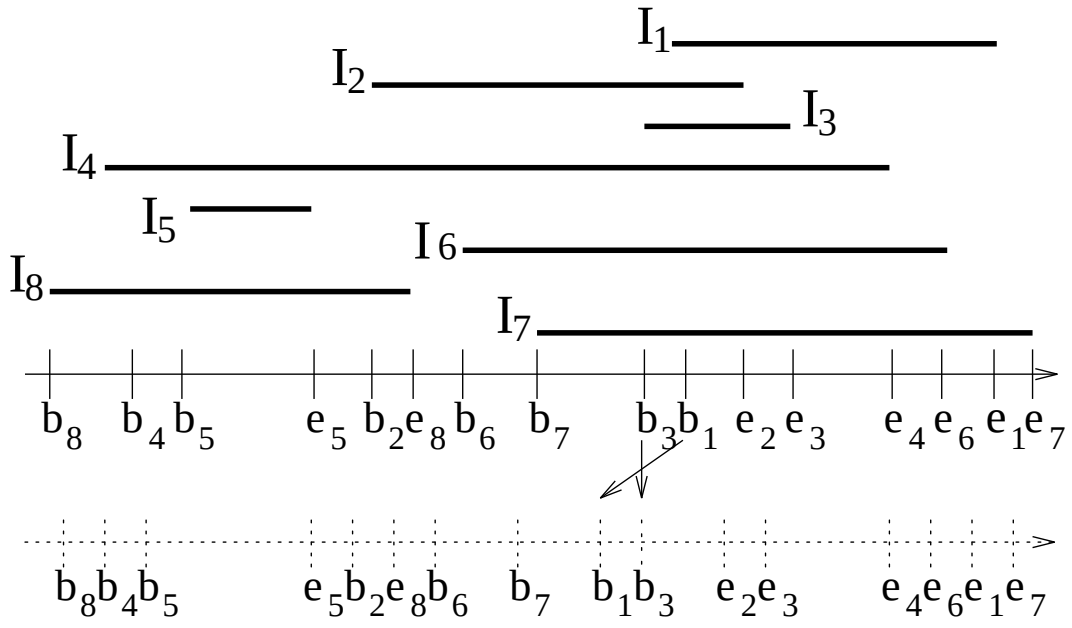

Figure 3: Moving intervals in one dimension.

There is a sorting routine, which in particular cases can be better than $\mathcal{O}(n \log n)$. Insertion sort insertion sort[1] can be very fast for nearly sorted lists. Such an algorithm takes time $\mathcal{O}(n+c))$ where c is the number of exchanges necessary. We can maintain a table of overlapping intervals at each time step. This table can be updated with a total cost of $\mathcal{O}(n+c)$. In a normal MD-simulation the number of exchanges c will be close to the number k of changes in the overlap status, and the extra $\mathcal{O}(c-k)$ work will be negligible.

Extension to three dimensions

The three dimensional problem is more complicated than the one-dimensional case. We have three independent intervals aligned along coordinate axis. Now, our first step is to sort three lists, one for every axis. This will need $\mathcal{O}(n+c)$ operations. For every exchange in any list, we will check the change in the overlap status. The cost of this operation is $\mathcal{O}(n+c)$. The extra work for each pair, which doesn't change the status, will be $\mathcal{O}(n-k)$, but in real simulations the extra work can be found to be completely negligible. The relationship $k \ll n$ holds so the algorithm will perform essentially with $\mathcal{O}(n)$.

[1] In general insertion sort may need up to $\mathcal{O}(n^2)$ operations!

Closest-feature-algorithm

The main idea is to utilize convexity to establish some local applicability criteria for verifying the closest features of two polyhedra. This algorithm was developed by Lin (1993) for problems in animation and robotics.

Voronoi regions

A Voronoi region V_f associated with a feature f of a polyhedron P is a set of points exterior to P which are closer to that feature than to any other.

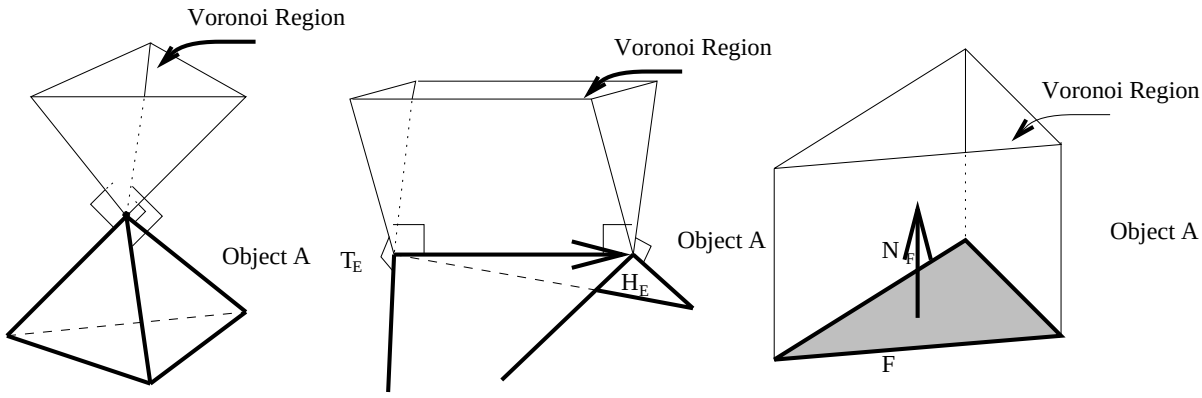

Figure 4: Typical Voronoi region for a vertex, an edge and a face.

If a point p on object P_A lies inside the Voronoi region of f_B on object P_B, then f_B is a closest feature to the point P and vice versa for an Voronoi region of f_A. If we have a pair of features fulfilling the above condition, we have a pair of closest features.

The algorithm

We are looking on two features f_A and f_b on two polyhedra P_A and P_B.

1. Calculate the Voronoi regions V_A and V_B
2. Calculate a point p_A on f_A that is the closest to f_B and a point p_B on f_B that is the closest to f_a
3. Check for $p_A \in V_B$. If not: choose new f_A and restart algorithm
4. Check for $p_B \in V_A$. If not: choose new f_B and restart algorithm

If this algorithm terminates, we have two features f_A and f_b which are the closest possible pair for the polyhedra P_A and P_B. If step 2 or 3 fail, we must choose a feature, so that the distance $d(f_A, f_b^{new} < d(f_A, f_b^{old})$. In this case, we are moving around the minimum of the distance function, in which we may become trapped. There is a proof by Lin (1993), that it is possible, to find these features in constant time and that for any pair f_A and f_B the algorithm will converge in $\mathcal{O}(n^2)$ time (n = number of features).

The main problem is now to choose a new f so that $d(f_A, f_b^{new} < d(f_A, f_b^{old})$ is valid. The main idea is to chose the feature belonging to the violated Voronoi plane. But there are also a few special cases which aren't discussed here.

Real simulations

In a normal MD-simulation we can store the closest pair and reuse it as starting point for the next time step. In this case, since we have a local test, we will normally have only 1 or 2 iterations for the algorithm, the run time will be independent of the polyhedra's shape. This can be observed in numerical simulations. For the table below, the algorithm calculates the distance between two rotating particles 100000 times. The particle size and the number of iterations is shown. The calculations where performed on an IBM Power 2 Workstation.

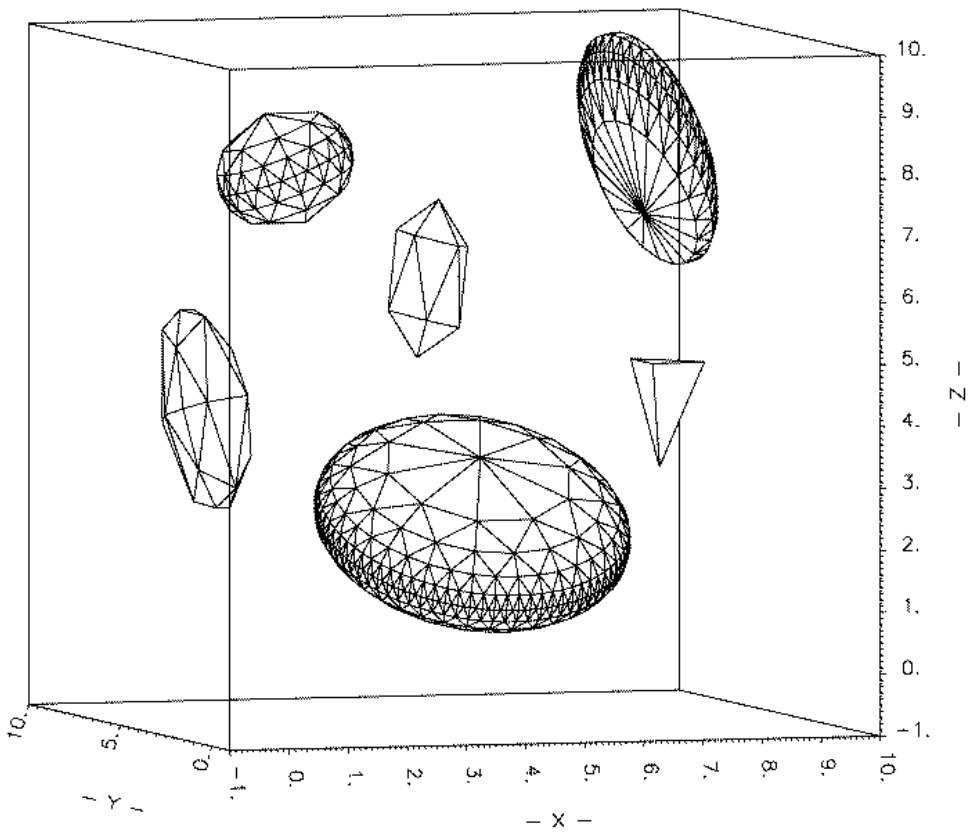

Figure 5: Some typical shapes (up to 462 faces).

faces	tot.	1 it.	2 it.	3 it.	4- it.	μs
4	14	98290	1430	259	20	63
8	26	99803	196	0	2	83
20	62	99497	465	36	2	71
32	98	99703	296	0	1	69
40	122	99702	222	75	1	72
80	242	99164	737	97	2	76
100	302	99563	292	144	1	70
140	422	99486	436	77	1	63
160	482	99442	549	8	1	70
200	602	99361	630	8	1	78
400	1202	98907	1057	35	1	74
600	1802	98271	1667	57	5	78
920	2762	97661	2217	112	10	58

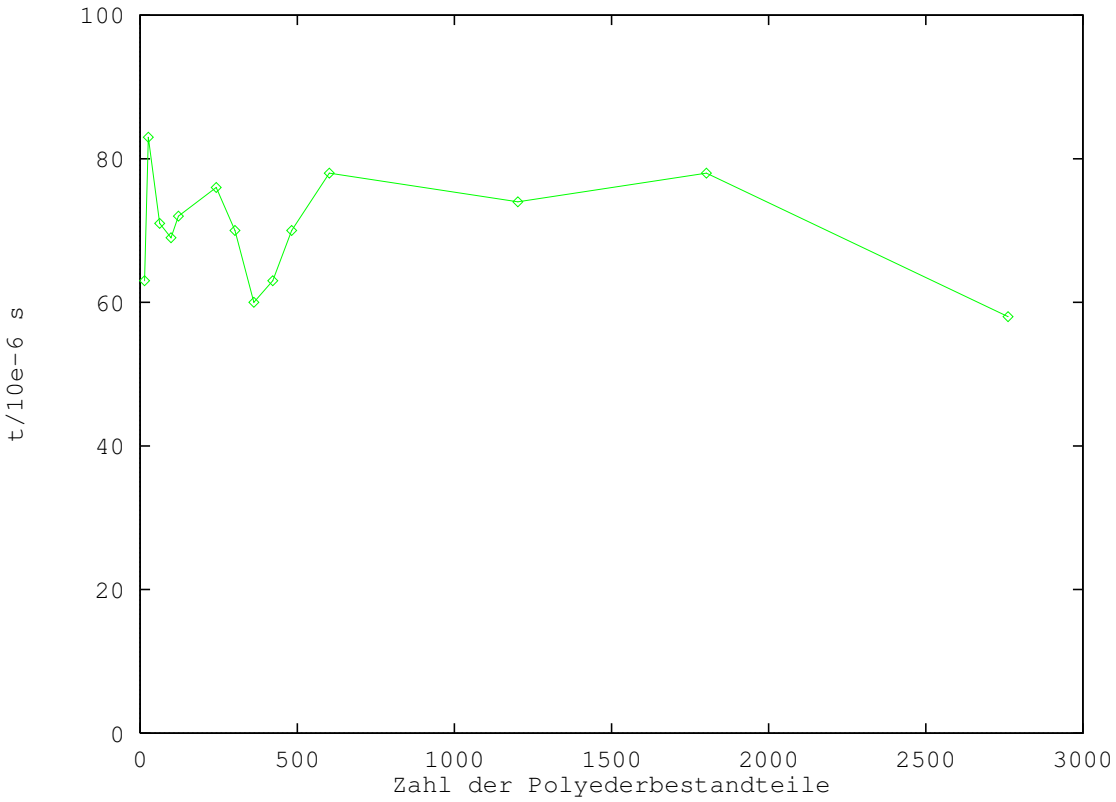

Figure 6: Computing time for different number of polyhedra features.

Conclusion

We have presented an algorithm, originally developed for virtual reality visualizations by D.Baraff and M.C.Lin, that enables us to use arbitrary convex polyhedra. The expected run time is $\mathcal{O}(\mathrm{N})$. The algorithm consists of two parts. In the first step, collisions of the particles' bounding boxes are detected by resorting the list of all boxes. The second step is a fast method to compute the distance between two polyhedra by finding and tracking the closest points. The expected constant time consumption results from the fact that each update of the closest feature pair involves only the constant number of neighboring features. The algorithm results from the incremental nature of Lin's approach and the geometric coherence between successive, discrete movements.

References

Baraff D (1993), Course notes 60 for the ACM SIGGRAPH 1993, An Introduction to Physically Based Modeling

Lin M.C. (1993), Dissertation, Efficient Collision Detection for Animation and Robotics

Schinner A. (1995), Diploma thesis, Numerische Simulationen für granulare Medien