The BoxTree: Exact and Fast Collision Detection of Arbitrary Polyhedra

Gabriel Zachmann

Fraunhofer Institute for Computer Graphics
Wilhelminenstrasse 7
64283 Darmstadt, Germany
email: zach@igd.fhg.de

Abstract

An algorithm for exact collision detection is presented which can handle arbitrary non-convex polyhedra efficiently. The approach attains its speed by a hierarchical adaptive space subdivision scheme, the BoxTree, and an associated divide-and-conquer traversal algorithm, which exploits the very special geometry of boxes. Boxes were chosen, because they offer much tighter space partitioning than spheres.

The traversal algorithm is generic, so it can be endowed with other semantics operating on polyhedra.

The algorithm is fairly simple to implement and it is described in great detail in an "ftp-able" appendix to facilitate easy implementation. The construction of the data structures is very simple and fast. Timing results show the efficiency of this approach.

Keywords: collision detection, virtual reality, hierarchical data structures.

1 Introduction

Collision detection is at the core of many different applications; for example, in *physically based simulation* and *virtual reality*. *Animation* is one of many applications of physically based simulation. Other areas are *path planning* of robots [8], *tele-operation* [5], and *NC milling*. Each of these areas need some kind of collision avoidance.

There are two major parts characterizing *collision handling*: the *collision detection* and the *collision response*. Although both parts pose interesting problems, this paper will focus only on the collision detection part. For further reading on the collision response part see, for example, [21, 2, 12].

In virtual reality, collision detection can be used to facilitate intuitive interaction [1], natural manipulation of the environment, any kind of physically based simulation, and modeling. In general, collision detection with appropriate collision response can make a virtual reality application look more believable [15]. However, the requirements are most severe. Under all circumstances, the collision detection must be realtime in order to attain the effect of immersion.

While good results have been achieved for convex polyhedra, non-convex, arbitrary polyhedra still present a "hard" problem under real-time constraints.

The algorithm presented in this paper can deal with arbitrary polyhedra, meaning just a bunch of polygons here. If two such polyhedra intersect at a given time, the algorithm will find two (or more) witnesses (an edge and a polygon). In this paper, we will not consider the issue of finding the exact time of primal contact between two polyhedra.

The BoxTree data structure is a binary tree, which is a non-uniform, adaptive space subdivision. Because it is non-uniform, we suspect it to out-perform approaches using uniform subdivisions, like grids (as in [9]). Furthermore, since it is adaptive, it improves the expected runtime performance.

The leaves of a BoxTree contain edges and polygons which define the associated polyhedron. A tree construction algorithm tries to build an optimal tree with respect to the collision detection algorithm.

The results show that the BoxTree algorithm performs much better than simple (potentially $O(n^2)$) algorithms when object complexity is above a certain level ($\approx$ 200 polygons/object).

Due to the recursive refinement nature of the algorithm, it can be interrupted at any stage should the application choose to do so in order to insure a constant frame rate. So, this algorithm is a good candidate for adaptive workload balancing.

The hierarchical data structure is built only once for every object. It does not have to be transformed as the object moves.

Outline of the paper. Section 2 describes previous work done in the field. Section 3 introduces our new algorithm, while Section 4 provides a detailed description of the algorithm to build the associated data structure. Results are presented in Section 5. The paper concludes with an outlook in Section 6, and conclusions in Section 7.

2 Previous Work

Collision detection seems to have attracted much attention over the past 15 years. In the beginning, researchers seem to have come from the area of robotics and computational geometry. Later on, physically based modeling and animation had a special need for exact collision detection. Despite its comparatively long history, real-time exact collision detection has not been tackled except for the past few years.

Computational geometry first focused on the *construction* of the intersection of two polyhedra [22, 20]. Later, researchers realized that the *detection problem* is interesting by itself and can be solved more efficiently than the construction problem [7, 6, 27].

In the field of robotics, a completely different approach has been pursued: collisions are detected in *configuration space* (see [8], for example). This approach seems to be well suited for path-planning.

As stated earlier [9], the representation of objects has great impact on collision detection algorithms. *Non-b-rep* representations, e.g., octree, BSP, CSG, etc., allow/need quite different approaches [4, 23, 24, 30].

For collision avoidance systems, an *approximate collision detection* is quite appropriate [5].

[15], [16] present an *object partitioning* approach somewhat similar to ours using spheres instead. However, the construction of the auxiliary data structures is much more involved, plus the covering of space with spheres is inherently redundant. [9] partition the set of polygons of an object by a uniform grid.

[11] compute the *distance* between convex polyhedra (or its spherical extension) with approximately linear complexity. [18, 19] present an *incremental* distance algorithm for convex polyhedra. Recently, [26] added a hierarchy of convex bounding volumes. However, the algorithms are much more complicated to implement. A separating plane is used to compute the distance between convex polyhedra by [13, I.8].

Collision detection of *flexible objects* is needed for physically based simulation and for animation. Typical objects are "soft" objects like clouds, clothes, drops, etc. Flexible objects are treated by [21, 10, 29, 31, 28].

An approach which computes the *exact time of collision* was given by [3], who use quaternions to represent orientation and formulate the problem in 7-dim. configuration space.

3 The BoxTree algorithm

3.1 Motivation for BoxTrees

Here is a very simple algorithm for arbitrary objects with traditional speed-up improvements:

Check every edge of polyhedron A if it intersects any of the polygons of polyhedron B, and vice versa. (It is *not* sufficient to check only the edges of A against polygons of B. It is also *not* sufficient to check vertices for being interior.)

Of course, the algorithm above is improved by doing some pre-checks: in a pre-phase, we collect all polygons of B which are in the bounding box of A. Then, edges of A are checked only against those polygons of B which have passed this pre-check. This "filtering" is done merely on the basis of bounding boxes, so it is fast enough to improve overall performance. (The speed-up gained by this phase is about a factor of 1.5.)

Another very simple pre-check is to test if the edges e of A are in the bounding box of B. There is no need to do this in a pre-phase, since every edge is considered exactly once.

In the following, this algorithm will be called the "simple" algorithm. It is potentially an $O(n^2)$ algorithm. However, to our experience this never happens in "real" cases: if the objects don't intersect, then their bounding boxes don't overlap much, thus many polygons and edges won't pass the bounding box checks; if the objects do intersect, then there are probably quite a few intersecting edges/polygons, and chances are good that we find one pair of those quite early.

The BoxTree-algorithm will be compared to this "simple" algorithm in order to give a feeling for the improvements.

Profilings have shown that most of the time of the simple algorithm presented above is spent in the inner loop (the *all pairs weakness*). Within this inner loop, most of the time is spent with the loop construct itself plus the bounding box check!

The idea is to use a *divide-&-conquer* approach. It was inspired by BSP trees, k-d trees, and *balanced bipartitions* (known in the area of VLSI layout algorithms). Of course, this involves a pre-processing step; so this algorithm cannot be used if geometry changes often during run-time.

Sphere trees [15, 16] don't seem to be as well suited as a tree of boxes, since spheres usually have to overlap very much in order to cover all polygons. Also, constructing a sphere tree doesn't seem to be as simple as constructing a BoxTree [16]. ([25] constructs non-hierarchical sphere coverings for a set of vertices.) Another advantage of BoxTrees is that they don't have to be transformed as the associated object moves, whereas an object's transformation has to be applied to its sphere tree.

One can also use a regular grid to partition an object's bounding box [9]. But usually, hierarchical schemes outperform their regular flat counterpart, if they don't have to be re-built dynamically.

It is highly desirable that a collision detection algorithm can handle arbitrary polyhedra, since most geometry data, coming from CAD systems, are usually not well-formed objects: there might be gaps between polygons belonging to the same object, polygons could overlap, and almost all polyhedra are not convex by themselves.

The BoxTree-algorithm can handle these polyhedra: objects which are just a collection of plane polygons. Objects may even be self-overlapping.

However, it avoids the disadvantages of the sphere tree algorithm, because it uses another partitioning scheme. Since it is hierarchical, it avoids also the disadvantages of simple grids.

3.2 Outline of the algorithm

The simple algorithm as given above will be improved by a divide-&-conquer approach which is as follows (see Figure 1): we divide the bounding boxes of A and B into two parts, not necessarily of equal size (we call them "left" and "right" sub-box); we partition the set of edges of A into two sets depending whether they are in the left or the right sub-box; in the same manner, we partition the set of polygons of B. When checking edges of A and faces of B for intersection, we first check whether bbox(A) intersects bbox(B) (the non-aligned ones!); if they don't, we're finished. If they do, we check all 4 pairs of sub-boxes of A and B, resp., for intersection. Obviously, we don't have to check edges against polygons, for which their boxes don't intersect.

Of course, the sub-box pre-processing is done *recursively*, which is why we will call the whole data structure a *BoxTree*.

Sometimes, it is more efficient if we split a box such that one of the sub-boxes doesn't contain any polygons at all (such a box will be called "empty"). Then, the check between an empty box and another

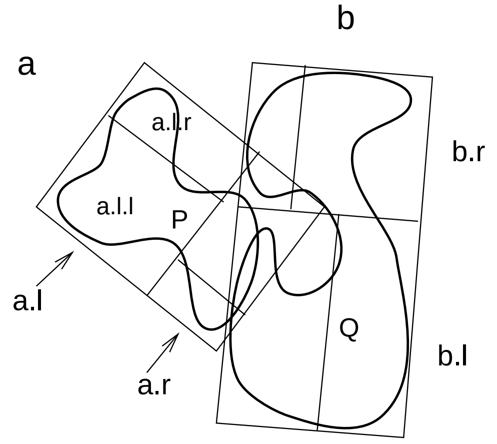

Figure 1: Only faces and edges of overlapping boxes have to be checked for intersection. For example, edges of a.l don't have to be checked with polygons of b.l .

(non-empty) one is trivial. Of course, "chipping off" an empty sub-box is not always possible, nor is it always sensible (criteria will be derived below in 4).

BoxTrees will be constructed in *model space*, i.e., when no transformations are applied to the object. When objects are transformed during the simulation, those boxes have to be transformed as well. However, we need to transform only the root boxes. We do that by setting up the recursive traversal appropriately. Then, no further transformations (of sub-boxes) have to be done.

The intersection test of two boxes could be done by the Liang-Barsky algorithm [17]. However, exploiting the very *special geometry of boxes* allows a much more efficient intersection test for two boxes: we will clip all box-edges parallel to each other at the same time. This will enable us to re-use many results during one box/box-check, plus we can re-use *all* of the arithmetical computations when descending down one level in the BoxTree. Special features of boxes are: the faces form three sets of two parallel faces each, the edges form three sets of four parallel edges each, when a box is divided by a plane perpendicular to an edge, all edges retain their entering/leaving status.

3.3 Simultaneous Recursive Traversal of BoxTrees

Simultaneous Recursive Traversal of two BoxTrees consists of two phases: an initialization phase and a traversal phase. Here, by "simultaneous" we mean

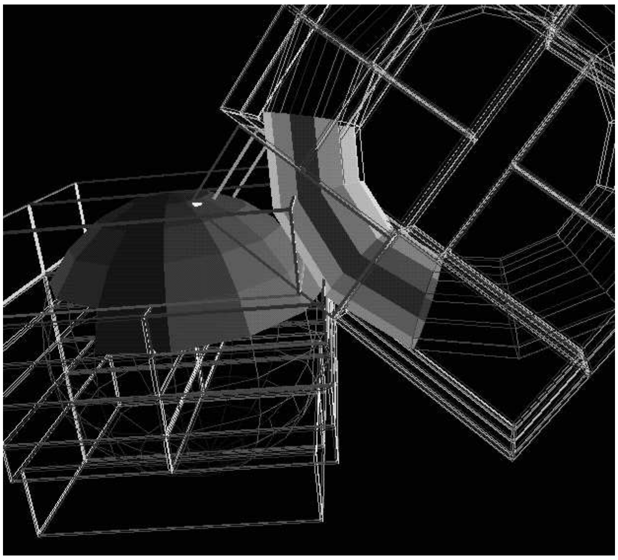

Figure 2: This visualization of the BoxTree algorithm shows, how many and which polygons are actually considered for intersection. The leaves of the BoxTree are depicted graphically by boxes.

that the two trees of both objects are traversed synchronously.

The algorithm (see also Figure 2) has the following pseudo-code outline:

<u>Simultaneous traversal of BoxTrees</u>

```
a = box in A's BoxTree, b = box in B's BoxTree
a.l, a.r are left and right sub-boxes of a

traverse(a,b):
a, b don't intersect → return
a or b is empty → return
b leaf →
  a leaf →
    elementary operation on BoxTree leaves
    return
  a not leaf →
    a.l,b intersect → traverse(a.l,b)
    a.r,b intersect → traverse(a.r,b)
b not leaf →
  a leaf →
    a,b.l intersect → traverse(a,b.l)
    a,b.r intersect → traverse(a,b.r)
  a not leaf →
    a.l,b intersect →
      a.l,b.l intersect → traverse(a.l,b.l)
      a.l,b.r intersect → traverse(a.l,b.r)
    a.r,b intersect →
      a.r,b.l intersect → traverse(a.r,b.l)
      a.r,b.r intersect → traverse(a.r,b.r)
```

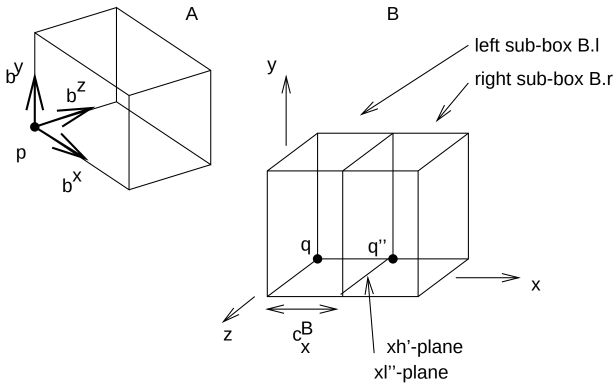

Figure 3: Splitting box B perpendicular to its x-edges bounds the line intervals of edges of A.

For collision detection, the "*elementary operation*", which operates on two leaves of the BoxTree, is the simple detection algorithm. However, the simultaneous traversal of BoxTrees could be used for other functions, too: the only part that would have to be re-defined is that "*elementary operation*", which provides the "semantics" of the overall operation (see [24] for a similar point of view regarding BSP trees).

The construction of BoxTrees will be explained in Section 4.

In the remainder of this section we will assume that the reader is familiar with the Liang-Barsky algorithm.

For a given pair (a, b) of boxes, all the information on their intersection status is given by two sets of 3×4 line parameter intervals for the edges of a and b, resp. If all intervals of one polyhedron are empty, then (a, b) do not overlap.

Initialization phase. This phase computes the initial intervals for the root boxes of two polyhedra A and B. Conceptually, we have to set up 2×3 tables Each entry in those tables can be calculated using no more than one multiplication and one addition. Each column of a table yields the line parameter interval of one edge.

The calculations of this phase can be done by the same routines which do the calculations for the traversal phase.

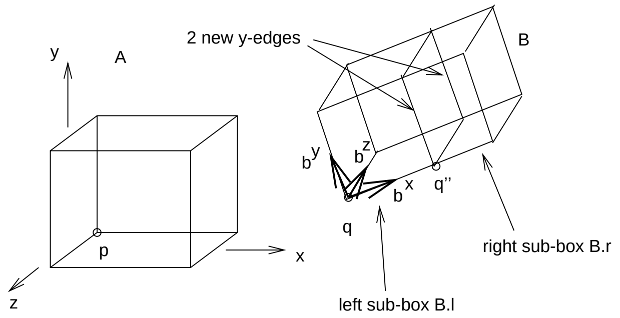

Figure 4: Splitting box B perpendicular to its x-edges yields 2 new y-intervals and 2 new z-intervals. All other intervals can be re-used.

Traversal phase. The basic step of the traversal is the test "`a,b.l` *intersect*" and "`a,b.r` *intersect*". We will do this by bisecting the box b into its left and right sub-box, which is equivalent to computing two new sets of $2 \times (3 \times 4)$ line parameter intervals, one describing $(a, b.l)$, the other describing $(a, b.r)$.

This seems like a lot of computational work; however, all of the information stored in a set of intervals for (a, b) can be re-used, one half for $(a, b.l)$, the other half for $(a, b.r)$ (see Figures 3 and 4).

Math-ematical details It is not worthwhile to present all the tedious mathematical details here. However, figuring them out can be quite cumbersome and error prone, so interested readers can obtain them via anonymous ftp from `ftp://ftp.igd.fhg.de/pub/-doc/techreports/zach/BoxTree-appendix.ps.gz`.

Suffice it to say here, that $2 \times (3 \times 4)$ line parameter intervals (as mentioned above) amount to $2{\times}(3{\times}$(24mul.+24add.)) (actually, its 3div.+21mul.). However, half of those have been computed in a step earlier, so during the tree traversal we need 72 mul.'s and 72 add.'s per box split.

4 Constructing the BoxTree

The BoxTrees being constructed here are inspired by *k-d trees* and *balanced bipartitions* from VLSI layout algorithms.

We do not construct octrees because they are too inflexible. In fact, octrees are just a special case of our data structure. Here, we want to construct balanced trees for reasons which will become clear in a moment.

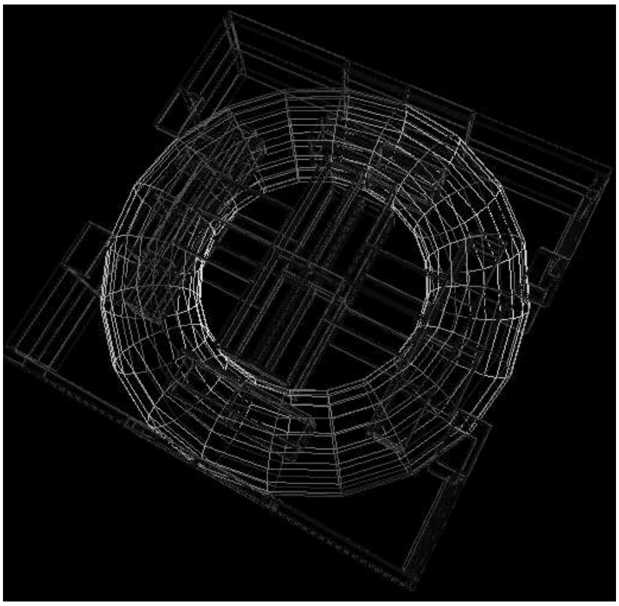

Figure 5: This shows all the empty boxes of the Box-Tree for a torus. During intersection tests, these can be rejected trivially. The object's complexity is rather low (400 polygons), so only 23% of its bounding box are covered by empty boxes. With larger complexities 40%–60% are covered, typically.

The following discussion will be restricted to the construction of BoxTrees for a set of polygons. Everything carries over to edges quite analogously.

The goal is to partition recursively the set of polygons in such a way that the intersection test between two such partitions involves a minimum number of elementary (i.e., edge-polygon) intersection tests. In the following, we will try to derive some heuristics for an optimal partitioning.

Whenever the collision detection algorithm steps down one level, and it discards one of the sub-boxes, we want as many polygons as possible to be discarded. This leads to a space subdivision scheme which tries to balance both parts (in terms of polygon counts, not in terms of space!).

In general, there will be always polygons which are in both sub-boxes, though. During a collision check, we have to deal with those (at least) twice. This leads to the heuristic that a bisection of a box should cut as few polygons as possible.

We start with a given set of n polygons. Given a cut-plane c perpendicular to the x-axis (w.l.o.g.), we denote the number of polygons to the left, the right, and crossing c by n_l, n_r, and n_c, resp. According to the heuristic proposed above, we define a *penalty* *function* for c by

$$p(c) = |n_l - n_r| + \gamma n_c$$

where γ is the factor by which a crossing polygon is worse than an unbalanced one. (Note: in general, $n_l + n_r + n_c \geq n$.)

The basic step for building a BoxTree is to find the cutplane c for a given set of polygons such that c realizes the *global minimum*

$$\min \begin{Bmatrix} \min\{p(c_x) \mid c_x \perp \text{x-axis}, c_x \in [x_{\min}, x_{\max}]\} \\ \min\{p(c_y) \mid c_y \perp \text{y-axis}, c_y \in [y_{\min}, y_{\max}]\} \\ \min\{p(c_z) \mid c_z \perp \text{z-axis}, c_z \in [z_{\min}, z_{\max}]\} \end{Bmatrix}$$

We use the simpler function to be $p(c) = |n_l - n_r|$ which is monotonic. Thus we can find the minimum by *interval bisection*, and results have been satisfactory.

After we have found a cut-plane, we divide the input array of polygons into two; crossing polygons are copied into both (for reasons which will be made clear below). Then we start the process over again for the two new arrays.

As mentioned above, "empty" boxes are "good", too (see Figure 5). By splitting off empty boxes during the tree construction, the non-empty boxes will approximate the boundary more closely. However, an empty box won't pay off if it is too small, so we introduce an empty-box-threshold.

Before trying to find the cutplane c which realizes kind of a median, we try to find a cutplane e, such that one of the two sub-boxes is empty, and which realizes the maximum empty sub-box. If the quotient of the volume of that empty sub-box and the volume of its father is greater than the empty-box-threshold, then we use the cutplane e.

In the case of an empty sub-box the recursion is trivial: the input array of polygons is passed to the non-empty sub-box.

The box splitting recursion will stop when one of the following conditions holds:

- depth $\geq d_{\max}$.
- # polygons in the box currently considered for splitting $\leq$ Min.
- $n_l > \lambda n$ or $n_r > \lambda n$ (it doesn't make sense to split the box, if one of the sub-boxes contains almost as many polygons as the father; typ. $\lambda \approx 0.8$).

When the recursion stops, we attach the array of polygons to the corresponding leaf of the BoxTree.

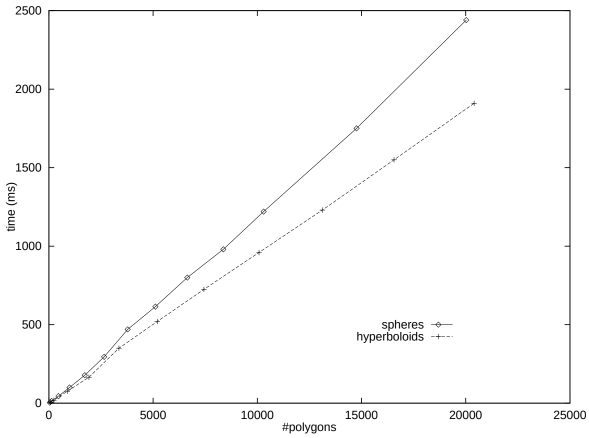

Figure 6: Experiments seem to indicate that building BoxTrees is in $O(n)$ average running time. The graph shows timings for building the boxtree for spheres and hyperboloids. Timing was done on an R4400/200MHz.

It should be evident now, why we did not choose octrees: In the simplest form, octrees start with a cube, whereas we want to start with a tight-fitting bounding box. Even if we start our octree with a tight-fitting box, and even if we choose the "midpoint" off-center, the siblings will not be balanced, in general. Finally, an implementation of a simultaneous traversal of octrees is much more complicated.

Complexity. The complexity of constructing BoxTrees is in $O(n \log n)$, where n is the number of polygons, under certain assumptions. Quite similarly, the memory complexity can be shown to be in $O(n \log n)$. Experiments indicate an even better average running time of $O(n)$ (see Figure 6).

We assume that cutting a box takes s =const passes over all polygons in that box; we assume further that every cut will split the box's polygons into 3 sets of equal size: left, right, and crossing polygons. This means that a sub-box gets $\frac{2}{3}$ of the box's polygons. Thus, the depth of a BoxTree will be $d = \log_{\frac{3}{2}}(n)$.

Let $T(n)$ be the time needed to build a BoxTree for n polygons. Then, $T(3k) = c(3k) + 2T(2k)$, and $T(3) = c$. Assume $n = 3^d$, then $T(3^d) \leq c\sum_0^d 2^i 3^{d-i} \leq c' d 3^d \in O(n \log n)$.

Crossing polygons. What should we do with crossing polygons (polygons which are on both sides of the cut plane)? The approach we have taken is to store polygons only at leaves. So, crossing polygons will be put in both sub-boxes. This avoids some

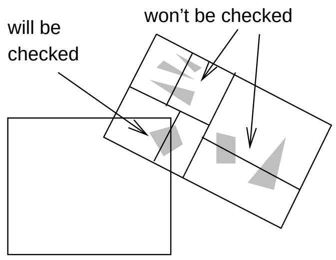

Figure 7: If crossing polygons are stored at leaves of the BoxTree, too, they can be discarded during the simultaneous traversal like "non-crossing" polygons.

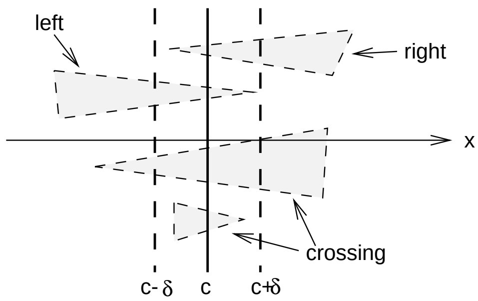

Figure 8: For splitting a set of polygons by a plane, geometrical robustness can be achieved by giving the cut-plane a certain "thickness".

disadvantages if we would store them at iner nodes. Of course, polygons can be stored multiple times at leaves, this way. However, this does not cause any memory problems: tests have shown that a BoxTree contains by a factor of $1.2 - 1.6$ more pointers to edges/faces than there really are.

Geometrical robustness. Although this seems to be of minor importance, experiments showed its necessity very early. This is especially true for objects which are computer-generated and expose a high symmetry, like spheres, tori, extruded and revolved objects, etc. These objects usually have very good cut-planes, but if the splitting algorithm is not robust, the BoxTree will be a mess.

Here, the problem is: when do we consider a polygon to be on the left, the right, or crossing the cut-plane? Because of numerical inconsistencies, many polygons might be classified "crossing" even though they only touch the cut-plane (see Figure 8). The idea is simply to give the cut-plane a certain "thickness" 2δ. Then, we'll still consider a polygon left of a cut-plane c, even if one of its edges is right of c, but left of $c + \delta$. All the possible cases are depicted in Figure 8.

5 Results

For timing tests we chose the following scenario: two objects move in a cage. Initial positions, initial translational and rotational velocities are chosen randomly at start-time. When the two objects collide, they bounce off each other based on simple heuristics (e.g., by exchanging translational and/or rotational velocities). The size of the cage is chosen so as to "simulate" a dense environment, i.e., most of the time there are only "almost-collisions", which is the "bad" case for most algorithms. In general, the cage size was chosen $1.5 - 2$ time the radius of the test-objects, so that collisions will happen fairly often (but large enough so that the two objects will not "get stuck"). The test objects were regular ones, like spheres, tori, tetraflakes, etc., and real-world data (e.g., an alternator). Rendering was switched off, of course. This scenario was chosen in order to exclude any side-effects, e.g., by doing any bbox checks.

First, we tried to find *optimum parameters* for a BoxTree, namely the maximum depth, the minimum number of polygons/edges per box, and the threshold for an "empty-box" split. To this end, we ran several tests with different objects and different choices of those parameters. The problem is actually to find a global optimum in 4-space for each polygon count and each object type. This would require a lot of tests taking days or weeks of CPU time! However, several timing experiments indicated that one can indeed search for the optima of all three parameters independently. Figure 9 shows the timing tests for finding the optimal maximum depth (on an R4000/50MHz Indigo) when the minimum number of polygons per box is 1. It turned out that the optimal minimum number of polygons per box yields about the same maximum depth. We also ran tests with the fixed "optimal" maximum depth while varying the minimum number of polygons; these tests suggested that said optimal maximum depth, together with 1 being the minimum number of polygons per box, is actually the best choice of those two parameters.

Similar tests were done to find the optimal threshold for when to split off an empty box. They yielded similar results in that there seems to be an optimal threshold which is independent of the other parameters. Furthermore, the "near-optimal" range seems to be fairly broad. We also checked experimentally that empty boxes do actually give some speed-up (see Figure 10).

It also turned out (fortunately), that optimal boxtree parameters do not depend much on the type of the object. The timing tests described above have been conducted for spheres, tori, cylinders, and

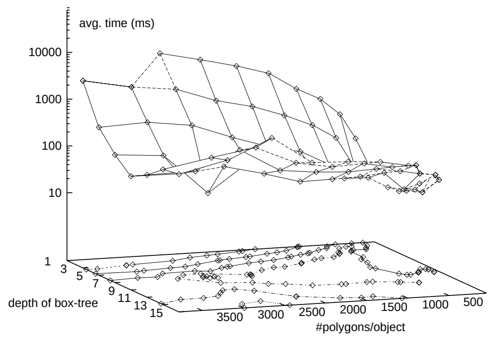

Figure 9: Search for the optimal depth of a BoxTree. This is the graph for two spheres. Testing two tori or two tetra-flakes yielded very similar results. Each sample is an average over 10×500 frmaes.

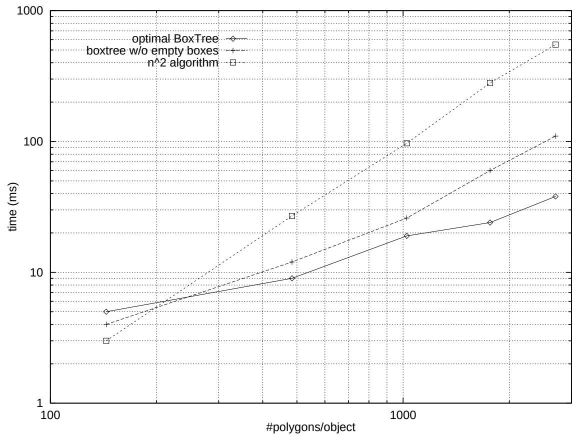

Figure 10: Comparison of the BoxTree algorithm with the simple algorithm. Scenario: two tori bouncing off each other in a fairly tight cage. Other object types (sphere and tetra-flake) yielded similar results with slightly different thresholds.

"tetra-flakes" (a tetrahedron which has small tetrahedra placed recursively on its sides). They showed that the optimal maximum tree depth, for example, varies by about ± 1 across different object types.

The following table was obtained, which is used for generating near-optimal BoxTrees:

#p'gons	100	300	700	1300	2000	3000
depth	4	5	6	7	8	9

Next, we *compared* the BoxTree algorithm (using optimal parameters for the BoxTree construction) to the simple algorithm as described in Section 3.1; the result for two tori is shown in Figure 10. The same scenario as above was used. Each sample is an average over 20×2000 frames. The tests were run on an R4400/200MHz.

As expected, BoxTrees are much faster when object complexity is above a certain threshold, but slower for small objects. As can be seen from the graph, a collision check of two fairly close 1000-polygon-tori takes about 20 msec on average. The threshold (for tori) is about 100 polygons, below which a simple algorithm out-performs the sophisticated one.

Proposal. We would like to make a general remark here. In the past, most researchers have provided timing results on their algorithms. However, little has been done to compare different exisiting algorithms (some comparisons have been published by [14]). Perhaps this is, because it is very cumbersome to re-implement other algorithms in order to compare them to one's own. To alleviate the comparison of new algorithms to exisiting ones, we would like to propose the following method: new algorithms are compared against a certain, commonly agreed upon, "calibration" algorithm. That calibration algorithm *and* the test scenario(s) should be simple and very easy to implement. In our oppinion, the scenario and the simple algorithm (see 3.1) presented here might be a possible candidate. Thus, many machine dependent side effects could be eliminated.

Of course, there should be a sufficiently "rich" suite of scenarios with different characteristics, e.g., fast/slow moving objects, many simple objects vs. few complex ones, mostly collisions vs. mostly "almost"-collisions, etc.

6 Future work

The algorithm presented above offers many more possibilities for further speed-up.

One could try a simultaneous traversal of *axis-aligned boxes*. They can be computed on-the-fly from the ones on the level above together with the information stored with each BoxTree node. Still, we would build the BoxTree as described in this paper.

So far, we have considered only the bounding box of polygons when constructing the boxtree. This is ok for "small" polygons (in terms of the diameter of their bounding sphere). However, it might tend to insert large polygons in sub-boxes with which they don't intersect at all. So it might be worthwhile to do a true polygon-box-intersection test when building the BoxTree, which would add some additional costs only to the pre-processing phase.

Is there a *relationship* between the average number of polygons per leaf box and the collision detection time? If so, this could be used for an even better heuristic to build the optimum BoxTree.

The algorithm seems to be particularly well suited for parallelization. Each recursion can be processed in parallel on up to 4 processes (depending on how many box-pairs have to be checked).

The BoxTree algorithm should be compared to other algorithms which are capable of checking collisions between arbitrary polyhedra, like [16]'s sphere tree algorithm.

7 Conclusion

An algorithm has been presented which allows real-time and exact collision detection for complex arbitrary polyhedra. This is achieved by a recursive divide-&-conquer approach, which is generic and can be furnitured with other semantics as well very easily. The recursion step basically consists of an intersection test of non-axis-aligned boxes, which gains its efficiency by exploiting the special geometry of boxes and by re-using all results from previous steps.

The associated data structure is a hierarchical space decomposition, which

An algorithm has been developed for the construction of the these data structures. It has been tested thouroughly and parameters have been determined which yield a near-optimal object partitioning. The construction of that data structure is very simple to implement and it is efficient.

The collision detection algorithm is simple to implement and very efficient. Two 1000-polygon-tori in close proximity, but not touching, can be checked in 20 msec on average on a R4400/200MHz.

If we do a breadth-first tree traversal, the collision detection algorithm it can be interrupted at any stage should the application choose to do so in order to insure a constant frame rate. In that case, collision/no-collision can be returned based on simple heuristics. So, this algorithm is a good candidate for adaptive workload balancing.

The algorithms presented can be combined with any global space partitioning method or any other method avoiding the all-pairs weakness on object-level.

Acknowledgements

I would like to take the opportunity to thank Prof. Dr. h.c. Dr.-Ing. J. L. Encarnação. At the same time, I would also like to thank all veople at the Beckman Institute in Urbana/Champaign, Illinois, where most of this work has been done, namely Dona Cox and Bill Sherman, who have made possible my stay there.

References

[1] P. Astheimer, F. Dai, M. Göbel, R. Kruse, S. Müller, and G. Zachmann. *Realism in Virtual Reality*, pages 189–210. Wiley & Sons, 1994.

[2] W. Bouma and G. Vanecek, Jr. Collision detection and analysis in a physical based simulation. In *Eurographics Workshop on Animation and Simulation*, pages 191–203, 1991.

[3] J. Canny. Collision detection for moving polyhedra. *IEEE Transactions an Pattern Analysis and Machine Intelligence*, PAMI-8(2):200–209, March 1986.

[4] I. Carlbom and I. Chakravaty. A hierarchical data structure for representing the spatial decomposition of 3-d objects. Manuscript, ??, ??

[5] G. M. H. Clifford A. Shaffer. A real-time robot arm collision avoidance system. *IEEE Transactions on Robotics and Automation*, 8(2), April 1992.

[6] D. P. Dobkin and D. G. Kirkpatrick. Fast detection of polyhedral intersection. *Theoret. Comput. Sci.*, 27:241–253, 1983.

[7] D. P. Dobkin and D. G. Kirkpatrick. A linear algorithm for determining the separation of convex polyhedra. *J. Algorithms*, 6:381–392, 1985.

[8] M. Erdmann and T. Lozano-Pérez. On multiple moving objects. *Algorithmica*, 2:477–521, 1987.

[9] A. García-Alonso, N. Serrano, and J. Flaquer. Solving the collision detection problem. *IEEE Computer Graphics and Applications*, ?(?):36–43, May 1994.

[10] M. Gascuel. An implicit formulation for precise contact modeling between flexible solids. In J. T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 313–320, Aug. 1993.

[11] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 4(2):193–203, 1988.

[12] J. K. Hahn. Realistic animation of rigid bodies. In J. Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 299–308, Aug. 1988.

[13] P. S. Heckbert, editor. *Graphics Gems IV*. Academic Press, Inc., Cambridge, MA, 1994.

[14] M. Held, J. T. Klosowski, and J. S. B. Mitchell. Evaluation of collision detection methods from virtual reality fly-throughs. In *Proc. 7th Canad. Conf. Comput. Geom.*, pages 205–210, 1995.

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

[16] P. M. Hubbard. Real-time collision detection and time-critical computing. In *SIVE 95, The First Worjshop on Simulation and Interaction in Virtual Environments*, number 1, pages 92–96, Iowa City, Iowa, July 1995. University of Iowa, informal proceedings.

[17] Y.-D. Liang and B. A. Barsky. A new concept and method for line clipping. *ACM Trans. Graphics (USA)*, 3:1–22, Jan. 1984.

[18] M. C. Lin and J. F. Canny. A fast algorithm for incremental distance calculation. In *Proc. of the 1991 IEEE International Conference onRobotics and Automation*, pages 1008–1014, April 1991.

[19] M. C. Lin and D. Manocha. *Efficient Contact Determination Between Geometric Models.* PhD dissertation, University of California, University of North Carolina Chapel Hill, URL: ftp://ftp.cs.unc.edu/pub/techreports/94-024.ps.Z, 1991(?).

[20] K. Mehlhorn and K. Simon. Intersecting two polyhedra one of which is convex. In L. Budach, editor, *Proc. Found. Comput. Theory*, volume 199 of *Lecture Notes in Computer Science*, pages 534–542. Springer-Verlag, 1985.

[21] M. Moore and J. Wilhelms. Collision detection and response for computer animation. In J. Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 289–298, Aug. 1988.

[22] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7:217–236, 1978.

[23] I. Navazo, D. Ayala, and P. Brunet. A geometric modeler based on the exact octree representation of polyhedra. *Computer Graphics Forum*, 5(2):91–104, June 1986.

[24] B. Naylor, J. Amanatides, and W. Thibault. Merging BSP trees yields polyhedral set operations. In F. Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 115–124, Aug. 1990.

[25] J. O'Rourke and N. Badler. Decomposition of three-dimensional objects into spheres. *IEEE Trans. On Pattern Analysis and Machine Intelligence*, PAM-1:295–305, 417, July 1979.

[26] M. K. Ponamgi, J. D. Cohen, M. C. Lin, and D. Manocha. Incremental algorithms for colision detection between polyhedral models. In *SIVE 95, The First Worjshop on Simulation and Interaction in Virtual Environments*, number 1, pages 84–91, Iowa City, Iowa, July 1995. University of Iowa, informal proceedings.

[27] M. Reichling. On the detection of a common intersection of k convex polyhedra. In *Computational Geometry and its Applications*, volume 333 of *Lecture Notes in Computer Science*, pages 180–186. Springer-Verlag, 1988.

[28] Smith, Andrew, Kitamura, Yoshifumi, Takemura, Haruo, Kishino, and Fumio. A simple and efficient method for accurate collision among deformable polyhedral objects in arbitrary motion. In *Virtual Reality Annual International Symposium*, pages 36–145, North Carolina, USA, 1995.

[29] J. M. Snyder, A. R. Woodbury, K. Fleischer, B. Currin, and A. H. Barr. Interval method for multi-point collision between time-dependent curved surfaces. In J. T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 321–334, Aug. 1993.

[30] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. In M. C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 153–162, July 1987.

[31] P. Volino and N. Magnenat-Thalmann. Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. In M. Daehlen and L. Kjelldahl, editors, *Eurographics 1994*, number 3, pages C–155–C–166, Oslo, Sept. 1994. Eurographics Association, Blackwell Publishers.