Proceedings of DETC'97
1997 ASME Design Engineering Technical Conferences
September 14-17, 1997, Sacramento, California

DETC97/CIE-4306

REAL-TIME AND EXACT COLLISION DETECTION FOR INTERACTIVE VIRTUAL PROTOTYPING

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

ABSTRACT

Many companies have started to investigate Virtual Reality as a tool for evaluating digital mock-ups. One of the key functions needed for interactive evaluation is real-time collision detection.

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.

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

The algorithm is fairly simple to implement and it is described in great detail in an "ftp-able" appendix to facilitate easy implementation. Pre-computation of auxiliary data structures is very simple and fast.

The efficiency of the approach is shown by timing results and two real-world digital mock-up scenarios.

Keywords: digital mock-up, interference detection, virtual reality, hierarchical data structures.

INTRODUCTION

Virtual prototyping, namely digital mock-ups (DMU), are becoming more and more important to help reduce the time-to-market, and thus the costs of a new model or product. It is said that each day of delay in producing a new car model costs about 2 million dollars (Dai and Reindl, 1986).

Many companies, especially in the automotive and aircraft industries, have started to evaluate Virtual Reality (VR) as a back-end to CAD and CAE in order to investigate a DMU of a new design. The idea is to allow designers, manufacturing planners, stylists, and analysts to evaluate several aspects of the new product interactively and immersively. All relevant parts including functional, descriptive, and other properties can be converted into a Virtual Environment (VE). Then, appearance, serviceability, packaging, variants, safety, and other aspects can be studied immersively and interactively by one or many engineers at the same time, possibly at different locations.

Collision detection in virtual prototyping scenarios One of the main goals of using a VR system for design evaluation is the potentially high degree of "reality" which can be experienced when immersed in a VE. In order to achieve this, the VR system needs to be able (among other things) to simulate realistic and natural object behavior at interactive frame rates.

In order to simulate a natural VE (Magnenat-Thalmann and Thalmann, 1994), the VR system must inhibit mutual penetration of objects. It should also make them "slide" on the surface of other objects when the user moves them to a position where penetration would occur. When the user interacts with the VE using a data-glove it should also be possible to grab or push objects just like in the real world.

Other tasks of a VR system in the context of DMU are geometrical and spatial analyses. In a fitting simulation a designer might want to check interactively, if a slightly larger, different part would fit in the place of the original part (see Figure 11. Or he might want to scale or shift a part while the system checks all relevant safety distances. In order to check serviceability of a part, the VR system has to track the work space necessary for removing the part by a human worker and for the tools, and it has to report both intentional as well as "forbidden" collisions. During an assembly or disassembly simulation it is often necessary to simulate kinematics in order to render a sensible design study.

When tools are to be used in an interactive serviceability test, the VR system must check geometric and spatial relationships between the tool and the object being manipulated by the worker. For instance, when the worker has placed a wrench on a screw, the system must maintain both objects coaxial while the worker unscrews it.

Handling collisions is at the core of all of the above mentioned functionalities. Two major parts can be identified in collision handling: *collision detection* and *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, (Moore and Wilhelms, 1988; Bouma and Vanecek, Jr., 1991).

Requirements and solution The requirements on a collision detection algorithm for interactive virtual prototyping are very demanding. Under all circumstances, the collision detection must be real-time in order to retain the effect of immersion. The algorithm must be able to handle arbitrary polyhedra, since most polynomial geometry data, converted from CAD data, are usually not "well-formed" in the following sense: there might be gaps between polygons belonging to the same object, polygons could overlap, and almost all polyhedra are not convex, some are even not closed. Furthermore, most collision handling modules must be given at least one point of intersection in order to take reasonable steps. Finally, the algorithm must be able to detect collisions for large object complexities at interactive speed, since polygon counts for typical CAD data range from 5,000 to 50,000 polygons per object.

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

The BoxTree-algorithm meets the above mentioned requirements: it can handle all objects which are just a collection of plane polygons. Objects may even be self-overlapping. It is fast enough to provide for interactive collision detection rates. If two such polyhedra intersect at a given time, the algorithm will find two (or more) witnesses (an edge and a polygon).

The BoxTree data structure is a binary tree, which is a hierarchical, non-uniform, adaptive space subdivision. The leaves of a BoxTree contain edges and polygons which define the associated polyhedron. Based on some heuristics, a tree construction algorithm builds a near-optimal tree with respect to the collision detection algorithm.

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

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.

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

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 (Muller and Preparata, 1978; Mehlhorn and Simon, 1985). Later, researchers realized that the *detection problem* is interesting by itself and can be solved more efficiently than the construction problem (Dobkin and Kirkpatrick, 1985; Reichling, 1988). The algorithms are very efficient in the asymptotical worst-case, however, they seem to be only of theoretical interest, because the hidden constant is probably very large. No implementation is known to us.

In the field of robotics, a completely different approach has been pursued: collisions are detected in *configuration space* (see (Erdmann and Lozano-Pérez, 1987), for example). This approach seems to be well suited for path-planning. However, no real-time implementation seems feasible.

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 (Navazo et al., 1986; Naylor et al., 1990; Thibault and Naylor, 1987).

For collision avoidance systems, an *approximate collision detection* is quite appropriate (Clifford A. Shaffer, 1992).

(Hubbard, 1995) 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. (García-Alonso et al., 1994) partition the set of polygons of an object by a uniform grid. In general, hierarchical schemes outperform their non-hierarchical counterpart, if they don't have to be re-built dynamically.

(Gilbert et al., 1988) compute the *distance* between convex polyhedra (or its spherical extension) with approximately linear complexity. (Lin and Manocha, 1991) present an *incremental* distance algorithm for convex polyhedra. Recently, (Ponamgi et al., 1995) developed a hierarchy of convex bounding volumes. However, the algorithms are quite complicated to implement. A separating plane is used to compute the distance between convex polyhedra by (Heckbert, 1994, I.8).

An approach which computes the *exact time of collision* was given by (Canny, 1986), who use quaternions to represent orientations and formulate the problem in 7-dim. configuration space. However, this is neither relevant for VR systems nor can it be computed in real-time. In this paper, we will not consider the issue of finding the exact time of primal contact between two polyhedra.

Octrees have been considered by (Yu et al., 1996). They have presented a fast method for simultaneous traversal of axis-aligned octrees. However, octrees are very time-consuming to build, so they are not suitable for real-time collision detection in dynamic environments.

THE BOXTREE ALGORITHM

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 can be improved by 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 an $O(n^2)$ algorithm in the worst-case.

Profilings have shown that most of the time of the simple algorithm presented above is spent in the inner loop (which is called 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 is inspired by BSP trees, k-d trees, and *balanced bipartitions* (known in the area of VLSI layout algorithms).

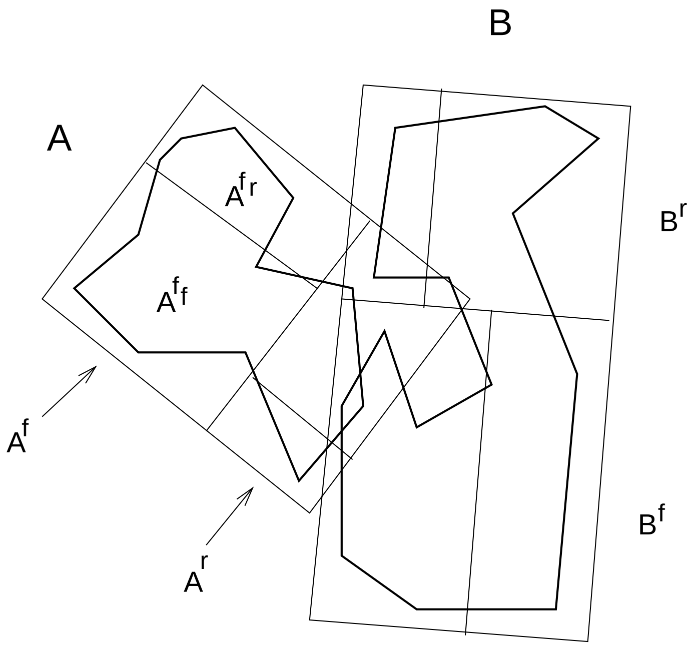

Figure 1. ONLY FACES AND EDGES OF OVERLAPPING BOXES HAVE TO BE CHECKED FOR INTERSECTION. FOR EXAMPLE, EDGES OF A.L DON'T HAVE TO BE CHECKED WITH POLYGONS OF B.L .

Outline of the algorithm

The simple algorithm as given above will be improved by the following divide-&-conquer approach (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 need to check edges against polygons only, if their boxes do 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"). The check between an empty box and another (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).

BoxTrees will be constructed in *object space*, i.e., no transformations are applied to the object. When objects are trans-

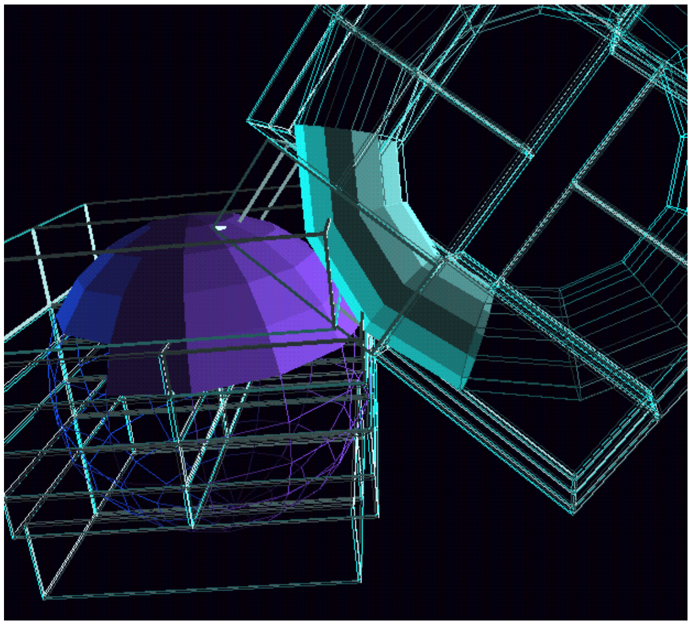

Figure 2. THIS VISUALIZATION OF THE BOXTREE ALGORITHM SHOWS, HOW MANY AND WHICH POLYGONS ARE ACTUALLY CONSIDERED FOR INTERSECTION. THE LEAVES OF THE BOXTREE ARE DEPICTED GRAPHICALLY BY BOXES.

formed during a simulation, the boxes of their BoxTrees have to be transformed as well. However, it turns out that 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 (Liang and Barsky, 1984). 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.

Simultaneous recursive traversal of BoxTrees

Simultaneous recursive traversal of two BoxTrees consists of two phases: an initialization phase and a traversal phase. By “simultaneous” we mean that the two trees of both objects are traversed synchronously.

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

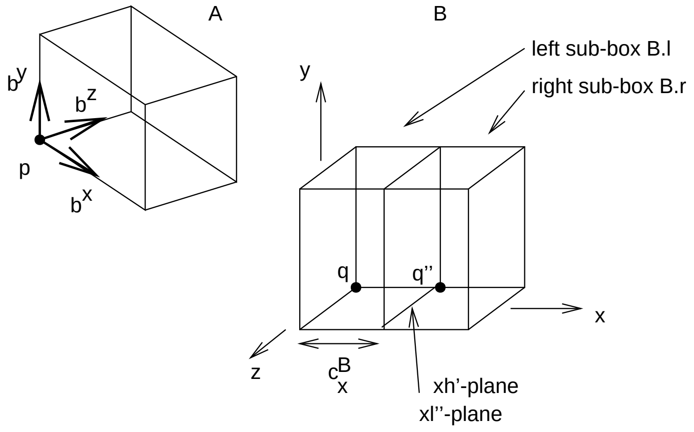

Figure 3. SPLITTING BOX B PERPENDICULAR TO ITS X-EDGES BOUNDS THE LINE INTERVALS OF EDGES OF A.

```
Simultaneous traversal of BoxTrees
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)
```

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 the “*elementary operation*”, which provides the “semantics” of the overall operation (see (Naylor et al., 1990) for a similar point of view regarding BSP trees).

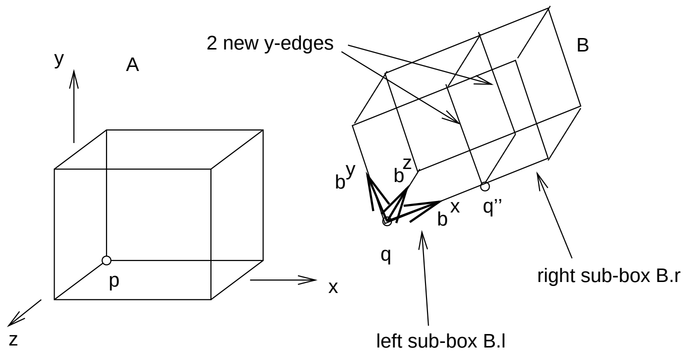

Figure 4. SPLITTING BOX B PERPENDICULAR TO ITS X-EDGES YIELDS 2 NEW Y-INTERVALS AND 2 NEW Z-INTERVALS. ALL OTHER INTERVALS CAN BE RE-USED.

A single traversal step. We will not discuss the details of one step of a simultaneous traversal of two BoxTrees due to limited space. Interested readers can find a thorough description in (Zachmann, 1995) and in `ftp://ftp.igd.fhg.de/pub/doc/-techreports/zach/BoxTree-appendix.ps.gz`. (Although the algorithm has been improved a lot, the mathematical details in (Zachmann, 1995) are still valid concerning one traversal step.)

One step of the traversal algorithm corresponds conceptually to splitting one box of a pair of boxes (a,b) (see Figures 3, 4) and calculating the overlap status of the two new pairs of boxes. Suffice it to say here, that such a step can be performed with at most 72 multiplications and 72 additions!

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 below.

The following discussion will discuss 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 number of elementary (i.e., edge-polygon) intersection tests with the set of polygons is minimized on average. In the following, we will derive some heuristics for an optimal partitioning.

Whenever the collision detection algorithm steps down one level in the BoxTree, and it discards one of the sub-boxes, we want it to discard as many polygons as possible. This leads to a space subdivision scheme which tries to balance the tree in terms

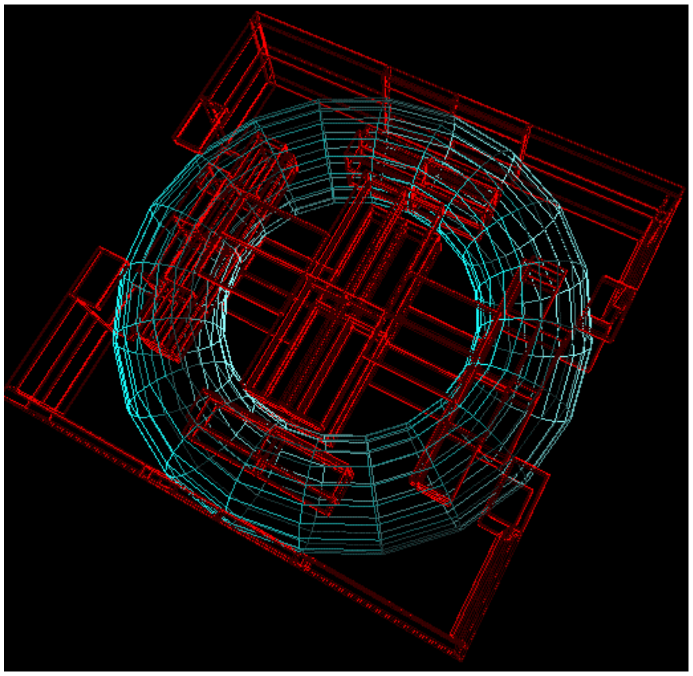

Figure 5. THIS SHOWS ALL THE EMPTY BOXES OF THE BOXTREE FOR A TORUS. DURING INTERSECTION TESTS, THESE CAN BE REJECTED TRIVIALLY. THE OBJECT'S COMPLEXITY IS RATHER LOW (400 POLYGONS), SO ONLY 23% OF ITS BOUNDING BOX IS COVERED BY EMPTY BOXES. WITH LARGER COMPLEXITIES 40%–60% ARE COVERED, TYPICALLY.

of polygon counts.

In general, there will be always polygons which are contained 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

$$p(c) = |n_l - n_r| + \gamma n_c \tag{1}$$

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*

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

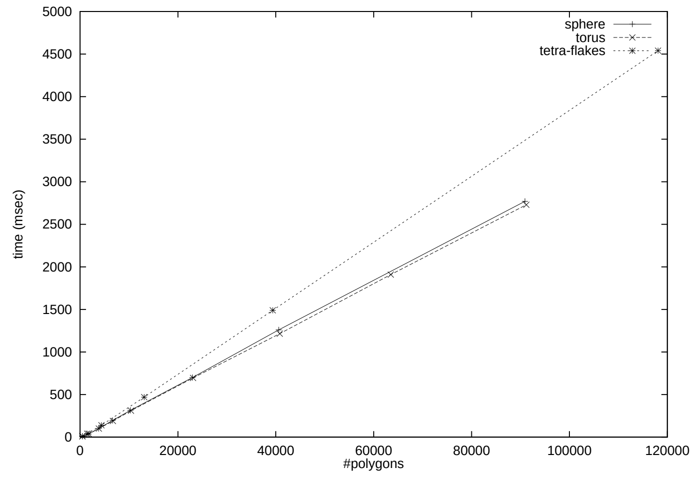

Figure 6. EXPERIMENTS INDICATE THAT BUILDING BOXTREES IS IN $O(N)$ AVERAGE RUNNING TIME. THE GRAPH SHOWS TIMINGS FOR BUILDING THE BOXTREE FOR SPHERES AND HYPERBOLOIDS. TIMING WAS DONE ON AN R4400/200MHZ.

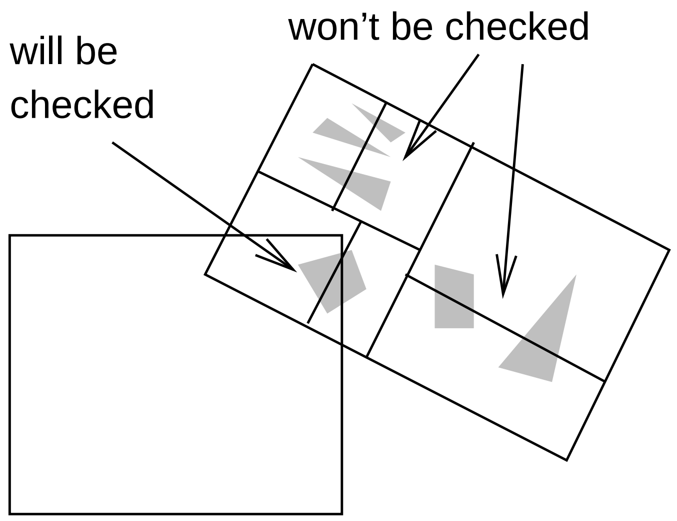

Figure 7. IF CROSSING POLYGONS ARE STORED AT LEAVES OF THE BOXTREE, TOO, THEY CAN BE DISCARDED DURING THE SIMULTANEOUS TRAVERSAL LIKE "NON-CROSSING" POLYGONS.

We use the simpler function $p(c) = |n_l - n_r|$, which is monotonic. Thus we can find the minimum by *interval bisection*, and the BoxTrees yielded by this function 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" (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 the balanced cut, 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.

The box bisection 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. $\lambda \approx 0.8$).

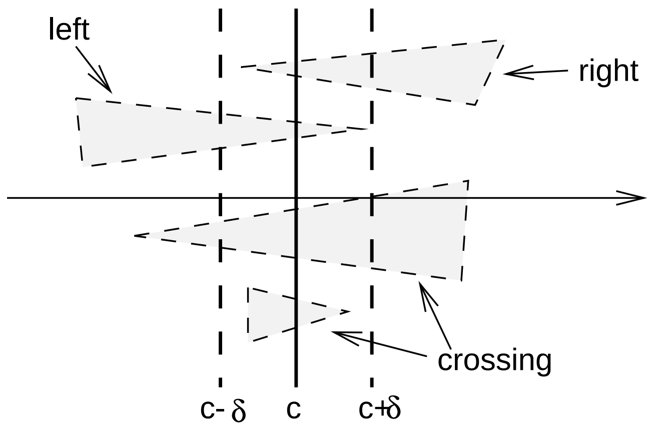

Figure 8. FOR SPLITTING A SET OF POLYGONS BY A PLANE, GEOMETRICAL ROBUSTNESS CAN BE ACHIEVED BY GIVING THE CUT-PLANE A CERTAIN "THICKNESS".

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

It should be evident now, why we did not choose octrees: sub-octants of a cell of the octree are not balanced, in general. Also, implementing a simultaneous traversal of octrees is much more complicated.

It can be shown that under certain assumptions the complex-

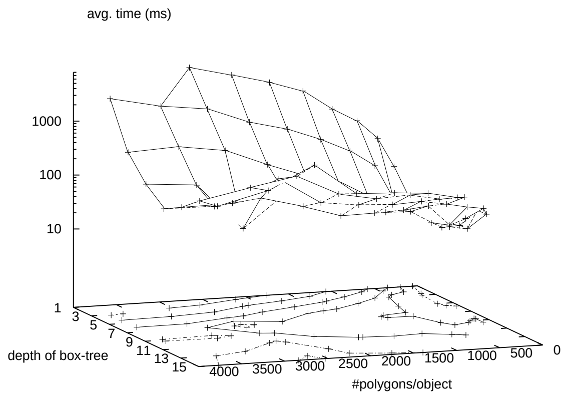

Figure 9. SEARCH FOR THE OPTIMAL DEPTH OF A BOXTREE. THIS IS THE GRAPH FOR TWO SPHERES. TESTING TWO TORI OR TWO TETRA-FLAKES YIELDED VERY SIMILAR RESULTS. EACH SAMPLE IS AN AVERAGE OVER 10×500 FRAMES.

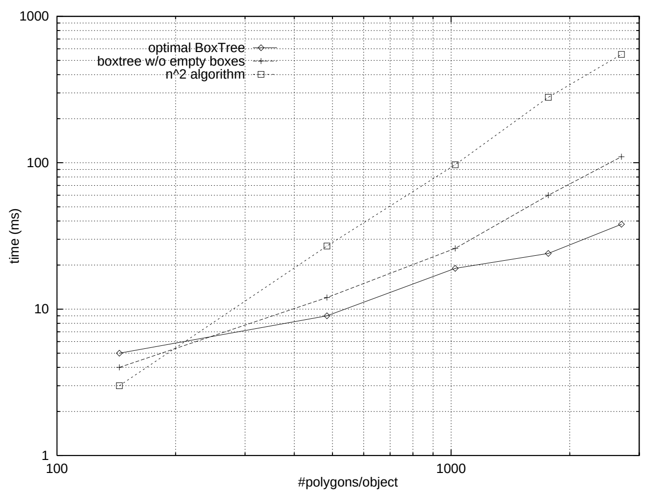

Figure 10. COMPARISON OF THE BOXTREE ALGORITHM WITH THE SIMPLE ALGORITHM. SCENARIO: TWO TORI BOUNCING OFF EACH OTHER IN A FAIRLY TIGHT CAGE. OTHER OBJECT TYPES (SPHERE AND TETRA-FLAKE) YIELDED SIMILAR RESULTS WITH SLIGHTLY DIFFERENT THRESHOLDS.

ity of computing a BoxTree is in $O(n \log n)$, where n is the number of polygons. Experiments indicate an even better average running time of $O(n)$ (see Figure 6).

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 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. This issue is of great importance, as experiments have showed clearly. This is especially true for polygonal 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 routine is not robust, the BoxTree will be not balanced at all.

The problem is: when do we consider a polygon to be on the left, the right, or on both sides of 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.

RESULTS

Timing

For timing tests we chose the following scenario: two objects move inside 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, tetra-flakes, 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 determined *optimal 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 ob-

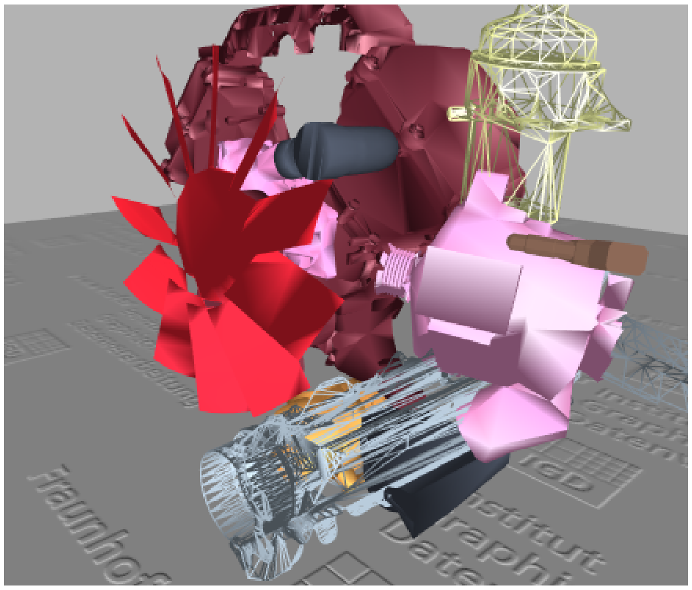

Figure 11. DURING AN INTERACTIVE FITTING SIMULATION IN A VIRTUAL ENVIRONMENT, THE SYTEM HIGHLIGHTS ALL OBJECTS COLLIDING WITH THE ALTERNATOR (DATA COURTESY AIT CONSORTIUM).

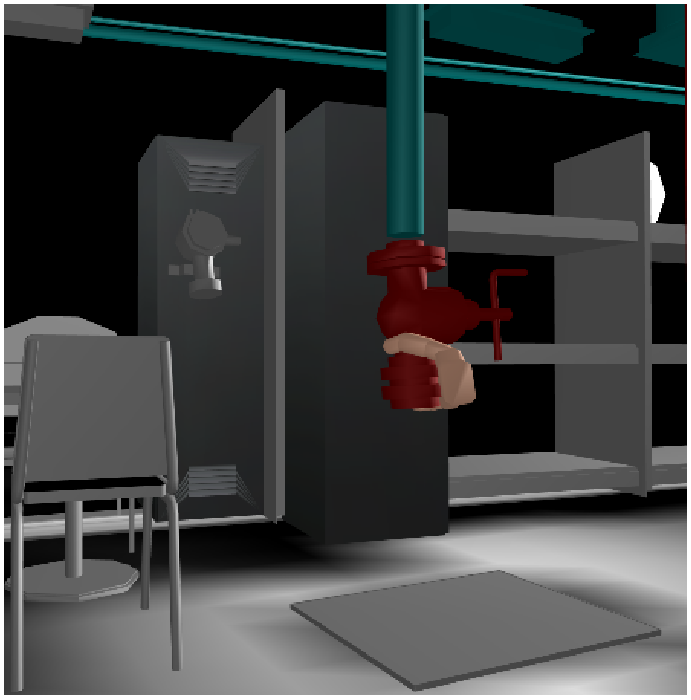

Figure 12. NEW PIPES CAN BE DESIGNED FROM BUILDING BLOCKS. HERE THE USER ATTACHES A VALVE TO THE END OF A NEW PIPE. WHEN THE VALVE TOUCHES THE PIPE AND IT HAS NEARLY THE “CORRECT” POSITION, THE SYSTEMS SNAPS IT TO THE PIPE. DURING POSITIONING, COLLISIONS BETWEEN VALVE AND PIPE ARE HIGHLIGHTED BY RENDERING THE PIPE IN WIREFRAME. (DATA COURTESY .)

ject 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 “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 for the optimal maximum BoxTree depth 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 ; 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.

Applications

The algorithms presented in this paper have been integrated with our proprietary VR system “Virtual Design II” (Astheimer et al., 1995). Several applications have been implemented with it, mainly for automotive companies (Dai et al., 1996).

An early DMU prototype has been described by (Dai and Reindl, 1986). This is one of the first attempts to simulate (among other things) a complete service maintainance of a car’s alternator by a digital mock-up: the user wearing a head-mounted display and data-glove interacts with a scene of about 40,000 polygons representing the front of engine compartment, which is rendered at about 20 frames/sec. He has to open the hood of the car first. Then he has to accomplish the following steps in order:

1. remove the fan,
2. tilt the oil filter,
3. push the cooling hose to the side,
4. unscrew the fixing wheel of the V-belt,
5. grab the alternator and take it out.

Although this is still a rather simplified scenario of a real maintainance operation, the VR system has to provide quite a few functionalities for object manipulation and object behavior. Each step and each functionality including the car hood involves collision detection!

Variants of parts can be tried and fitted interactively in place of the original ones. Figure 11 shows an example: all objects colliding with the new part will be highlighted on-line by switching their rendering to wireframe.

Another example of collision detection for digital mock-ups involves mostly pipes, here in the interior of a ship. It is an experimental application where a user can verify the design of all kinds of pipes in a ship. Furthermore, he can modify the existing layout or even design new pipes (see Figure 12). New pipes can be designed from building blocks such as straight segments, curved segments, valves, T-segments, etc. The system aids the designer by a snapping mechanism which attaches parts at each other when they are positioned touching each other. This requires fast and exact collision detection to achieve interactivity.

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.

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).

An incremental simultaneous traversal might save a lot of box-box checks during tree traversal. Unfortunately, it is not yet clear to us, how such an incremental algorithm could be implemented efficiently.

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 (e.g., distance computations). 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 (the BoxTree) is a hierarchical, non-uniform space decomposition, which can be pre-computed quite efficiently at start-up time. An algorithm for that has been presented and it has been tested thouroughly. Parameters have been determined which yield a near-optimal object partitioning with respect to fast simultaneous traversal.

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

Both the collision detection algorithm and the BoxTree construction algorithm are quite easy to implement.

The algorithms presented have been integrated with our proprietary VR software (Dai et al., 1996), which is being used for virtual prototyping in German automotive industry. The efficiency of the approach has been verified in several real-world digital mock-up scenarios.

ACKNOWLEDGEMENTS

I would like to take the opportunity to thank Prof. Dr. h.c. Dr.-Ing. J. L. Encarnação and Dr. Stefan Müller for the great working environment, and Dr. Fan Dai for his valuable comments and encouragement.

REFERENCES

Astheimer, P., , Dai, F., Felger, W., Göbel, M., Haase, H., Müller, S., and Ziegler, R. (1995). Virtual Design II – an advanced VR system for industrial applications. In *Proc. Virtual Reality World ’95*, pages 337–363.

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

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

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

Dai, F., Felger, W., Frühauf, T., Göbel, M., Reiners, D., and Zachmann, G. (1996). Virtual prototyping examples for automotive industries. In *Proc. Virtual Reality World*, Stuttgart.

Dai, F. and Reindl, P. (1986). Enabling digital mock-up with virtual reality techniques - vision, concept, demonstrator. In *ASME Design for Manufacturing Conferences*, Irvine, CA.

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

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

García-Alonso, A., Serrano, N., and Flaquer, J. (1994). Solving the collision detection problem. *IEEE Computer Graphics and Applications*, pages 36–43.

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

Heckbert, P. S., editor (1994). *Graphics Gems IV*. Academic Press, Inc., Cambridge, MA.

Hubbard, P. M. (1995). 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. University of Iowa, informal proceedings.

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

Lin, M. C. and Manocha, D. (1991(?)). *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.

Magnenat-Thalmann, N. and Thalmann, D., editors (1994). *Realism in Virtual Reality*, pages 189–210. Wiley & Sons.

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

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

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

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

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

Ponamgi, M. K., Cohen, J. D., Lin, M. C., and Manocha, D. (1995). 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. University of Iowa, informal proceedings.

Reichling, M. (1988). 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.

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

Yu, Y., Wu, M., and Zhou, J. (1996). An octre algorithm for dynamic interference detection using space partitioning. In *Proc. of The 1996 ASMEDesing Engineering Technical Conference and Computers in Engineering Conference*, pages 96–DECT/DAC–1046, Irvine, CA.

Zachmann, G. (1995). The BoxTree: Enabling real-time and exact collision detection of arbitrary polyhedra. In *Informal Proc. First Workshop on Simulation and Interaction in Virtual Environments, SIVE 95*, University of Iowa, Iowa City. The OX Association for Computing Machinery.