# Minimal Hierarchical Collision Detection *

Gabriel Zachmann
University Bonn
zach@cs.uni-bonn.de

## Abstract

We present a novel bounding volume hierarchy that allows for extremely small data structure sizes while still performing collision detection as fast as other classical hierarchical algorithms in most cases. The hierarchical data structure is a variation of axis-aligned bounding box trees. In addition to being very memory efficient, it can be constructed efficiently and very fast.

We also propose a criterion to be used during the construction of the BV hierarchies is more formally established than previous heuristics. The idea of the argument is general and can be applied to other bounding volume hierarchies as well. Furthermore, we describe a general optimization technique that can be applied to most hierarchical collision detection algorithms.

Finally, we describe several box overlap tests that exploit the special features of our new BV hierarchy. These are compared experimentally among each other and with the DOP tree using a benchmark suite of real-world CAD data.

Keywords:

Interference detection, virtual prototyping, hierarchical partitioning, hierarchical data structure, shape approximation, physically-based modeling, R-trees.

## 1 Introduction

Fast and exact collision detection of polygonal objects undergoing rigid motions is at the core of many simulation algorithms in computer graphics. In particular, virtual reality applications such as virtual prototyping need exact collision detection at interactive speed for very complex, arbitrary "polygon soups". It is a fundamental problem of dynamic simulation of rigid bodies, simulation of natural interaction with objects, and haptic rendering. It is very important for a VR system to be able to do all simulations at interactive frame rates. Otherwise, the feeling of immersion or the usability of the VR system will be impaired.

The requirements on a collision detection algorithm for virtual prototyping are: it should be real-time in all situations, it should not make any assumption about the input, such as convexity, topology, or manifoldedness (because CAD data is usually not "well-behaved" at all), and it should not make any assumptions or estimations about the position of moving objects in the future. Finally, polygon numbers are very high, usually in the range from 10,000 up to 100,000 polygons per object.

The performance of any collision detection based on hierarchical bounding volumes depends on two factors: (1) the tightness of the bounding volumes (BVs), which will influence the number of bounding volume tests, and (2) the simplicity of the bounding volumes, which determines the efficiency of an overlap test of a pair of BVs [GLM96]. In our algorithm, we sacrifice tightness for a fast overlap test.

Our hierarchical data structure is a tree of boxes, which are axis-aligned in object space. Each leaf encloses exactly one polygon of the object. Unlike classical AABB trees, however, the two children of a box cannot be positioned arbitrarily (hence we call this data structure "restricted boxtree", or just "boxtree"). This allows for very fast overlap tests during simultaneous traversal of "tumbled" boxtrees.

Because of the restriction we place on the relation between child and parent box, each node in the tree needs very little memory, and thus we can build and store hierarchies for very large models with very little memory. This is important as the number of polygons that can be rendered at interactive frame rates seems to increase currently even faster than Moore's Law would predict.

We also propose a very efficient algorithm for constructing good boxtrees. This is important in virtual prototyping because the manufacturing industries want all applications to compute auxiliary and derived data at startup time, so that they do not need to be stored in the product data management system. With our algorithm, boxtrees can be constructed at load-time of the geometry even for high complexities.

*Extended VRST'02 version

In order to guide the top-down construction of bounding volume hierarchies, a criterion is needed to determine a good split of the set of polygons associated with each node. In this paper, we present a more formal argument than previous heuristics did to derive such a criterion that yields good hierarchies with respect to collision detection. The idea of the argument is generally applicable to all hierarchical collision detection data structures.

We also propose a general optimization technique, that can be applied to most hierarchical collision detection algorithms.

Our new BV hierarchy can also be used to speed up ray tracing or occlusion culling within AABBs.

The rest of the paper is organized as follows. Section 2 gives an overview of some of the previous work. Our new data structure and algorithms are introduced in Section 3, while Section 4 describes the efficient computation of boxtrees. Results are presented in Section 5.

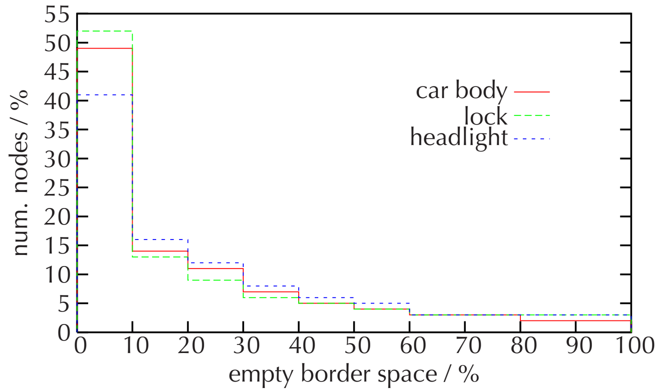


Figure 1: We have found that for most nodes in an AABB tree there is very little empty space between them and their parent on most sides.

## 2 Related Work

Bounding volume trees seem to be a very efficient data structure to tackle the problem of collision detection for rigid bodies.

Hierarchical spatial decomposition and bounding volume data structures have been known in computational geometry, geometrical data bases, and ray tracing for a long time. Some of them are k-d trees and octrees [PS90], R-trees [BKSS90], and OBB trees [AK89].

For collision detection, sphere trees have been explored by [Hub96] and [PG95]. [GLM96] proposed an algorithm for fast overlap tests of oriented (i.e., non-axis-parallel) bounding boxes (OBBs). They also showed that an OBB tree converges more rapidly to the shape of the object than an AABB tree, but the downside is a much more expensive box-box intersection test. In addition, the heuristic for construction of OBB trees as presented in [GLM96] just splits across the median of the longest side of the OBB.

DOP trees have been applied to collision detection by [KHM+98] and [Zac98]. AABB trees have been studied by [Zac97, vdB97, LAM01]. All of these data structures work quite efficiently in practice, but their memory usage is considerably higher than needed by our hierarchy, even for sphere trees.

More recently, hierarchical convex hulls have been proposed for collision detection and other proximity queries by [EL01]. While showing excellent performance, the memory footprint of this data structure is even larger than that of the previously cited ones. This is even further increased by the optimization techniques the authors propose for the collision detection algorithm.

Non-hierarchical approaches try to subdivide object space, for instance by a voxel grid [MPT99] or Delaunay triangulation [Gei00]. In particular, non-hierarchical approaches seem to be more promising for collision detection of deformable objects [ABG+00, HMB01, FL01].

Regarding the name of our data structure, we would like to point out that [BCG+96] presented some theoretical results on a class of bounding volume hierarchies, which they called BOXTREE, too. However, their data structure is substantially different from ours, and they do not provide any run-time results.

Bounding volume hierarchies are also used in other areas, such as nearest-neighbor search and ray tracing. For point k-d trees, [DDG00] have shown that a longest-side cut produces optimal trees in the context of nearest-neighbor searches. However, it seems that this rule does not apply to collision detection.

## 3 Data Structure and Algorithms

Given two hierarchical BV data structures for two objects, the following general algorithm scheme can quickly discard sets of pairs of polygons which cannot intersect:

```
traverse(A,B)
if A and B do not overlap then
    return
end if
if A and B are leaves then
    return intersection of primitives
            enclosed by A and B
else
    for all children A[i] and B[j] do
        traverse(A[i],B[j])
    end for
end if
```

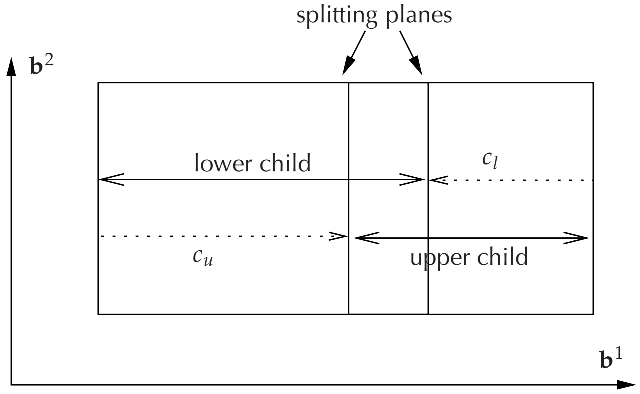


Figure 2: Child nodes are obtained from the parent node by splitting off one side of it. The drawing shows the case where a parent has a lower and an upper child that have coplanar splitting planes.

| BV hierarchy | Bytes | FLOPS |
|---|---|---|
| Restr. Boxtree (3.2.1) | 9 | 12 |
| Restr. Boxtree (3.2.4) | 9 | 24 |
| Restr. Boxtree (3.2.3) | 9 | 82 |
| sphere trees | 16 | 29 |
| AABB tree | 28 | 90 |
| OBB tree | 64 | 243 |
| 24-DOP tree | 100 | 168 |

Table 1: This table summarizes the amount of memory per node needed by the various BV hierarchies, and the number of floating point operations per node pair in the worst case during collision detection. The number of bytes also includes one pointer to the children. If the optimization technique from Section 3.2.5 is applied, then all FLOPS counts can be further reduced (about a factor 2 for boxtrees).

Almost all hierarchical collision detection algorithms implement this traversal scheme in some way. It allows to quickly "zoom in" on pairs of close polygons. The characteristics of different hierarchical collision detection algorithms lie in the type of BV used, the overlap test for a pair of nodes, and the algorithm for construction of the BV trees.

In the following, we will first introduce our type of BV, then we will present several algorithms to check them for overlap.

## 3.1 Restricted Boxtrees

In a BV hierarchy, each node has a BV associated that completely contains the BVs of its children. Usually, the parent BV is made as tight as possible. In binary AABB trees, this usually means that a parent box touches each child box on at least 3 sides. We have pursued this observation further: in the AABB tree of three representative objects, we have measured the empty space between all of its nodes and their parent nodes.[1] Figure 1 shows that for about half of all nodes the volume of empty space between its bounding box and its parent's bounding box is only about 10%.

The reason for the closeness between child and parent box along most directions is the way we split a set of polygons. One can, of course, imagine other splitting procedures that will lead to boxtrees not exhibiting this property. But the usefulness of such construction schemes seems to be questionable.

Consequently, it would be a waste of memory (and computations during collision detection), if we stored a full box at each node. Therefore, our hierarchy never stores a box explicitly. Instead, each node stores only one plane that is perpendicular to one of the three axes, which is the (almost) least possible amount of data needed to represent a box that is sufficiently different from its parent box. We store this plane using one float, representing the distance from one of the sides of the parent box (see Figure 2). The reason for this will become clear below.[2] In addition, the axis must be stored (2 bits) and we need to distinguish between two cases: whether the node's box lies on the lower side or on the upper side of the plane (where "upper" and "lower" are defined by the local coordinate axes of the object).

Because each box in such a hierarchy is restricted on most sides, we call this a *restricted boxtree*. Obviously, this restriction makes some nodes in the boxtree hierarchy slightly less tight than in a regular AABB tree. But, as we will see, this does not hurt collision detection performance. On the contrary, the resulting hierarchical data structure will allow very fast traversal.

The observation we made above for AABB trees is probably also true for other hierarchies utilizing some kind of axis-aligned BV (such as DOPs). If a DOP tree hierarchy is built according to the heuristics proposed in [Zac98], or according to the one proposed for OBB trees in [GLM96], then it is very likely that about half of the sides of each child DOP touch the corresponding side of the father DOP (and are, in fact, a subset of it). Therefore, the technique proposed above, together with the one below for testing the overlap, could be applied there as well.

Section 4 will describe in detail our algorithm to construct a restricted boxtree.

[1] For all nodes, *one* side was excluded in this calculation, which was the side where the construction performed the split.

[2] One could picture the resulting hierarchy as a cross between k-d trees and AABB trees.

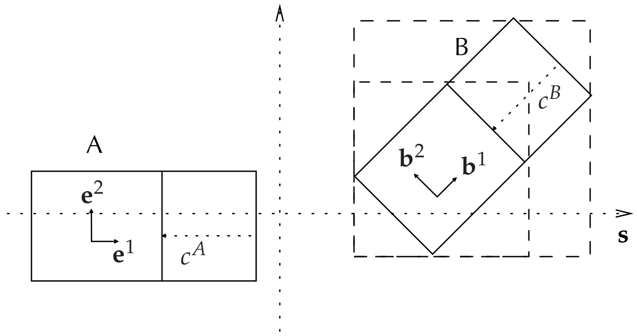


Figure 3: Only one value per axis $\mathbf{s}$ needs to be recomputed for the overlap test.

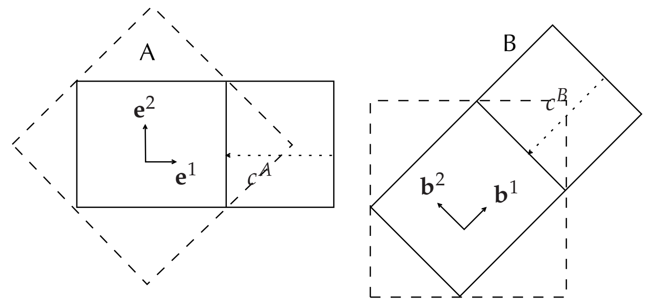


Figure 4: A reduced version of the separating axis test seems to be a good compromise between the number of false positives and computational effort.

## 3.2 Box Overlap Tests

The basic step in each hierarchical collision detection algorithm is the overlap test of two nodes of the hierarchy. There are usually two contradictory requirements: it should be fast, and it should return few or no false positives.[3]

In the following, we will describe several variations of an algorithm to test the overlap of two restricted boxes, exploiting the special properties of the restricted boxtree. All algorithms will assume that we are given two boxes A and B, and that $(\mathbf{b}^1, \mathbf{b}^2, \mathbf{b}^3)$ is the coordinate frame of box B with respect to A's object space.

### 3.2.1 Axis alignment

Since axis-aligned boxes offer probably the fastest overlap test, the idea of our first overlap test is to enclose B by an axis-aligned box $(\mathbf{l}, \mathbf{h}) \in \mathbb{R}^3 \times \mathbb{R}^3$, and then test this against A (which is axis-aligned already). In the following, we will show how this can be done with minimal computational effort for restricted boxtrees.

We already know that the parent boxes of A and B must overlap (according to the test proposed here). Notice that we need to compute only 3 values of $(\mathbf{l}, \mathbf{h})$, one along each axis. The other 3 can be reused from B's parent box. Notice further that we need to perform only one operation to derive box A from its parent box.

Assume that B is derived from its parent box by a splitting plane perpendicular to axis $\mathbf{b} \in \{\mathbf{b}^1, \mathbf{b}^2, \mathbf{b}^3\}$, which is distance $c$ away from the corresponding upper side of the parent box (i.e., B is a lower child).

We have already computed the parent's axis-aligned box, which we denote by $(\mathbf{l}^0, \mathbf{h}^0)$. Then, $l_x, h_x$ can be computed by (see Figure 3)

$$h_x = \begin{cases} h_x^0 - cb_x & \text{if } b_x > 0 \\ h_x^0 & \text{if } b_x \leq 0 \end{cases}$$

and

$$l_x = \begin{cases} l_x^0 & \text{if } b_x > 0 \\ l_x^0 - cb_x & \text{if } b_x \leq 0 \end{cases}$$

Similarly, if B is an upper child, $l_x, h_x$ can be computed by

$$h_x = \begin{cases} h_x^0 & \text{if } b_x > 0 \\ h_x^0 + cb_x & \text{if } b_x \leq 0 \end{cases}$$

and

$$l_x = \begin{cases} l_x^0 + cb_x & \text{if } b_x > 0 \\ l_x^0 & \text{if } b_x \leq 0 \end{cases}$$

Analogously, the new value along the other 2 axes can be computed.

In addition to saving a lot of computations, we also save half of the comparisons of the overlap tests of aligned boxes. Notice that we need to compare only 3 pairs of coordinates (instead of 6), because the status of the other 3 has not changed. For example, the only comparison we need to do along the x axis is

$$B \text{ and } A \text{ do } \textit{not} \text{ overlap if } \begin{cases} h_x < l_x^A & \text{if } b_x > 0 \\ l_x > h_x^A & \text{if } b_x \leq 0 \end{cases}$$

where $l_x^A, h_x^A$ are the x-coordinates of box A. Note that the decision $b_x \lesseqgtr 0$ has been made already when computing $h_x$ or $l_x$. Of course, these comparisons are precomputed, so that during traversal we only check a precomputed boolean value.

[3] These requirements are somewhat related to the ones on the type of BV itself, i.e., it should allow for fast overlap tests, and it should be as tight as possible. Both sets of requirements can be expressed in the total cost function for interference detection $T = N_{\text{bv}}C_{\text{bv}} + N_{\text{tr}}C_{\text{tr}}$ [GLM96].

Overall, this method needs 12 floating point operations (3 mul. + 4 add. + 5 comp.) for checking the overlap of a pair of nodes of a restricted boxtree.

### 3.2.2 Lookup tables

Since the $\mathbf{b}^i$ are fixed for the complete traversal of two boxtrees, and the $c$'s can be only from a range that is known at the beginning of the traversal, it would seem that the computations above could possibly be sped up by the use of a lookup table.

At the beginning of the traversal, we compute 3x3 lookup tables $L^i_j$ with 1000 entries each, one table for each component $b^i_j$. Then, during traversal, a term of the form $h_x = h^0_x - cb^s_x$ is replaced by $h_x = h^0_x + L[s][x][c]$, where $c$ now is an integer and $s$ is the splitting axis.

See below for results and discussion.

### 3.2.3 Separating axis test

The separating axis test (SAT) is a different way to look at linear separability of two convex polytopes: two convex bodies are disjoint if and only if we can find a separating plane, which is equivalent to finding an axis such that the two bodies projected onto that axis (yielding two line intervals) are disjoint (hence "separating axis"). For boxes, [GLM96] have shown that it suffices to consider at most 15 axes.

We can apply this test to the nodes in our restricted boxtree.[4] As previously, we do not need to compute all line intervals from scratch. Instead, we modify only those ends of the 15 line intervals that are different from the ones of the parent.

Let $\mathbf{s}$ be one of the candidate separating axes and assume that B is a lower box (see again Figure 3). As above, for a lower box we compute only

$$\begin{cases} h_s = h^0_s - c(\mathbf{b}\cdot\mathbf{s}) & \text{if } (\mathbf{b}\cdot\mathbf{s}) > 0 \\ l_s = l^0_s - c(\mathbf{b}\cdot\mathbf{s}) & \text{if } (\mathbf{b}\cdot\mathbf{s}) \le 0 \end{cases}$$

(and analogously for an upper box). Of course, we can precompute all 3x15 possible products $\mathbf{b}^i \cdot \mathbf{s}^j$.

The advantage of this test is that it is precise, i.e., there are no false positives. One disadvantage of this test is that there are 9 axes that are not perpendicular to A nor to B (these are the edge-edge cross products). For instance, in order to update the line intervals of B on the 3 separating axes given by $\mathbf{b}^i$, we do not need any multiplication at all. This is not possible with the 9 edge-edge axes.

In total, this method needs 82 FLOPS in the worst case, which is much higher than the method above, but still less than 200 FLOPS worst case for OBBs.[5]

### 3.2.4 SAT lite

As we have seen in the section above, the 9 candidate axes obtained from all possible cross products of the edge orientations do not lend themselves to some nice optimizations. Therefore, a natural choice of a subset of the 15 axes would be the set of 6 axes consisting of the three coordinate axes of A and B resp. (see Figure 4). This has been proposed by [vdB97] (who has called it "SAT lite"). Here, we apply the idea to restricted boxtrees and show that even more computational effort can be saved.

This variant can also be viewed as the first variant being executed two times (first using A's then B's coordinate frame). The total operation count is 24.

### 3.2.5 Further Optimizations

In this section, we will describe several techniques to further improve the speed of restricted boxtrees and hierarchical collision detection in general.

The first optimization technique is a general one that can be applied to all algorithms for hierarchical collision detection if the overlap test of a pair of nodes involves some node-specific computations that can be performed independently for each node (as opposed to pair-specific computations). Examples of this are the matrix multiplication at each node of a OBB hierarchy traversal, and the computations needed for the axis alignment or separating axis intervals during a restricted boxtree traversal.

The idea is to shift the node-specific computations up one level in the traversal. Assume that the costs of a node pair overlap test consist of a node-specific part $c_1$ and an overlap-test-specific part $c_2$. Then, the costs for a pair $(A,B)$ are $C(A,B) = 2c_1 + c_2 + 4(2c_1 + c_2) = 10c_1 + 5c_2$, because if (A,B) are found to overlap, all 4 pairs of children need to be checked. However, we can compute the node-specific computations already while still visiting (A,B) and pass them down to the children pairs, which reduces the costs to $C(A,B) = c_2 + 2c_1 + 4c_2 = 2c_1 + 5c_2$.

If the node-specific computations have to be applied only to one of the two hierarchies (or the node-specific costs of the other hierarchy are neglectible), then the difference is even larger, i.e., $9c_1 + 5c_2$ vs. $1c_1 + 5c_2$.

Depending on the actual proportions of $c_1$ and $c_2$, this can result in dramatic savings. In the case of restricted boxtrees with our first overlap test (axis alignment), this technique reduces the number of floating point operations

[4] In fact, our test by axis alignment can be considered a variant of the separating axis test, where only a special subset of axes is being considered instead of the full set.

[5] The worst case, of course, actually happens for exactly half of all node pairs being visited during a simultaneous traversal.

to 1.5 multiplications, 2 additions, and 5 comparisons! In the case of restricted boxtrees, this suggests to actually store the splitting plane offset of a node with its parent, so that each node holds two splitting planes. An additional advantage is that we can spend the storage of the leaves for triangle information, and we still have a proper restricted box available for them.

While this technique might already be implemented in some collision detection code, it has not been identified as a general optimization technique for hierarchical collision detection.

As usual, we arrange the children of a common parent node contiguously in memory, so that we need to keep only one pointer in the parent to the first of its children. This is not so much a necessity with other hierarchies, the nodes of which already have a large "net" memory footprint, but it does save considerable memory with our restricted boxtrees.

# 4 Construction of Boxtrees

The performance of any hierarchical collision detection depends not only on the traversal algorithm, but also crucially on the quality of the hierarchy, i.e., the construction algorithm.

Our algorithm pursues a top-down approach, because that usually produces good hierarchies and allows for very efficient construction. Other researchers have pursued the bottom-up approach [BCG+96], or an insertion method [GS87, BKSS90].

## 4.1 A General Criterion

Any top-down construction of BV hierarchies consists of two steps: given a set of polygons, it first computes a BV (of the chosen type) covering the set of polygons, then it splits the set into a number of subsets (usually two).

Before describing our construction algorithm, we will derive a general criterion that can guide the splitting process, such that the hierarchy produced is good in the sense of fast collision detection.

Let $C(A,B)$ be the expected costs of a node pair $(A,B)$ under the condition that we have already determined during collision detection that we need to traverse the hierarchies further down. Assuming binary trees and unit costs for an overlap test, this can be expressed by

$$C(A,B) = 4 + \sum_{i,j=1,2} P(A_i,B_j) \cdot C(A_i,B_j) \qquad (1)$$

where $A_i, B_j$ are the children of A and B, resp., and $P(A_i,B_j)$ is the probability that this pair must be visited (under the condition that the pair $(A,B)$ has been visited).

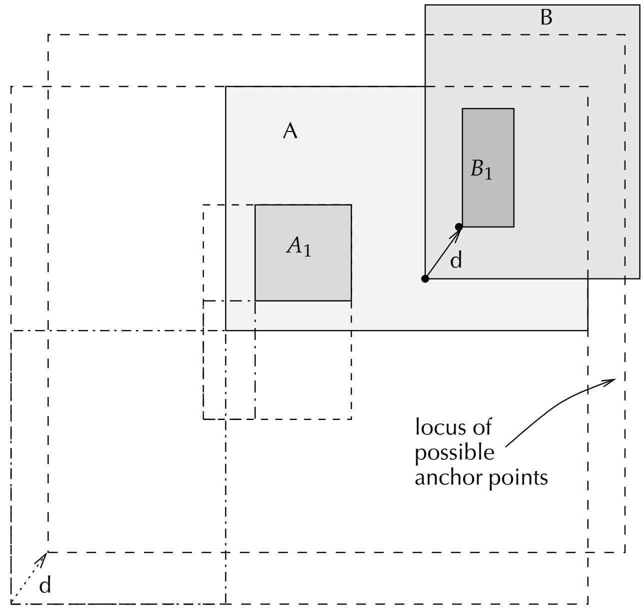


Figure 5: By estimating the volume of the Minkowski sum of two BVs, we can derive an estimate for the cost of the split of a set of polygons associated with a node.

An optimal construction algorithm would need to expand (1) down to the leaves:

$$\begin{aligned} C(A,B) = &P(A_1,B_1) + P(A_1,B_1)P(A_{11},B_{11}) \\ &+ P(A_1,B_1)P(A_{12},B_{11}) + \ldots + \\ &P(A_1,B_2) + P(A_1,B_2)P(A_{11},B_{21}) \\ &+ \ldots \end{aligned} \qquad (2)$$

and then find the minimum. Since we are interested in finding a local criterion, we approximate the cost function by discarding the terms corresponding to lower levels in the hierarchy, which gives

$$C(A,B) \approx 4\big(1 + P(A_1,B_1) + \ldots + P(A_2,B_2)\big) \qquad (3)$$

Now we will derive an estimate of the probability $P(A_1,B_1)$. For sake of simplicity, we will assume in the following that AABBs are used as BVs. However, similar arguments should hold for all other kinds of convex BVs.

The event of box A intersecting box B is equivalent to the condition that B's "anchor point" is contained in the Minkowski sum $A \oplus B$. This situation is depicted in Figure 5.[6] Because $B_1$ is a child of $B$, we know that the anchor point of $B_1$ must lie somewhere in the Minkowski sum $A \oplus B \oplus \mathbf{d}$, where $\mathbf{d} = \text{anchor}(B_1) - \text{anchor}(B)$.

[6] In the figure, we have chosen the lower left corner of $B$ as its anchor point, but this is arbitrary, of course, because the Minkowski sum is invariant under translation.

Since $A_1$ is inside $A$ and $B_1$ inside $B$, we know that $A_1 \oplus B_1 \subset A \oplus B \oplus \mathbf{d}$. So, for arbitrary convex BVs the probability of overlap is

$$P(A_1, B_1) = \frac{\mathrm{Vol}(A_1 \oplus B_1)}{\mathrm{Vol}(A \oplus B \oplus \mathbf{d})} = \frac{\mathrm{Vol}(A_1 \oplus B_1)}{\mathrm{Vol}(A \oplus B)} \quad (4)$$

In the case of AABBs, it is safe to assume that the aspect ratio of all BVs is bounded by $\alpha$. Consequently, we can bound the volume of the Minkowski sum by

$$\mathrm{Vol}(A) + \mathrm{Vol}(B) + \frac{2}{\alpha}\sqrt{\mathrm{Vol}(A)\,\mathrm{Vol}(B)} \leq \mathrm{Vol}(A \oplus B) \leq \mathrm{Vol}(A) + \mathrm{Vol}(B) + 2\alpha\sqrt{\mathrm{Vol}(A)\,\mathrm{Vol}(B)} \quad (5)$$

So we can estimate the volume of the Minkowski sum of two boxes by

$$\mathrm{Vol}(A \oplus B) \approx 2(\mathrm{Vol}(A) + \mathrm{Vol}(B))$$

yielding

$$P(A_1, B_1) \approx \frac{\mathrm{Vol}(A_1) + \mathrm{Vol}(B_1)}{\mathrm{Vol}(A) + \mathrm{Vol}(B)} \quad (6)$$

Since $\mathrm{Vol}(A) + \mathrm{Vol}(B)$ has already been committed by an earlier step in the recursive construction, Equation 3 can be minimized only by minimizing $\mathrm{Vol}(A_1) + \mathrm{Vol}(B_1)$. This is our criterion for constructing restricted boxtrees.

## 4.2 The Algorithm

According to the criterion derived above, each recursion step will try to split the set of polygons so that the cost function (3) is minimized. This is done by trying to find a good splitting for each of the three coordinate axes, and then selecting the best one. Along each axis, we consider three cases: both subsets form lower boxes with respect to its parent, both are upper boxes, or one upper and one lower box.

In each case, we first try to find a good "seed" polygon for each of the two subsets, which is as close as possible to the outer border that is perpendicular to the splitting axis. Then, in a second pass, we consider each polygon in turn, and assign it to that subset whose volume is increased least. In order to prevent "creeping greediness",[7] we run alternatingly through the array of polygons. An alternative would be to access them in a random order

[7] This happens, for instance, when the sequence of polygons happens to be ordered such that each polygon increases one subset's volume just a little bit, so that the other subset never gets a polygon assigned to.

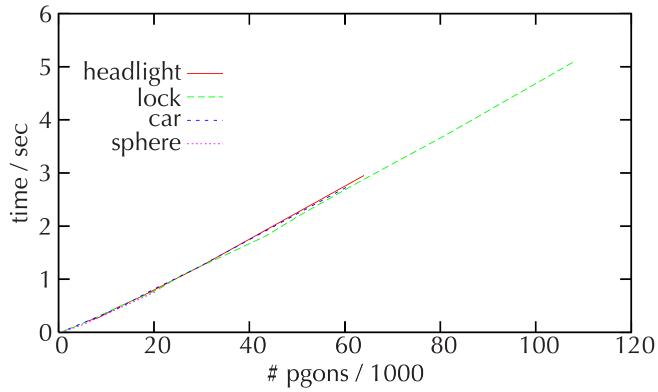


Figure 6: This plot shows the build time of restricted boxtrees for various objects.

After several good splitting candidates have been obtained for all three axes, we just pick the one with least total volume of the subsets.

The algorithm and criterion we propose here could also be applied to construct hierarchies utilizing other kinds of BVs, such as OBBs, DOPs, and even convex hulls. We suspect that the volume of AABBs would work fairly well as an estimate of the volume of the respective BVs.

We have also tried a variant of our algorithm, which considers only one axis (but all three cases along that axis). This was always the axis corresponding to the longest side of the current box. This experiment was motivated by a recent result for k-d trees [DDG00]. For the result, see Section 5.

In our current implementation, the splitting planes of both children are coplanar. We have not yet explored the full potential of allowing perpendicular splitting planes, too.

Our algorithm has proven to be geometrically robust, since there is no error propagation. Therefore, a simple epsilon guard for all comparisons suffices.

## 4.3 Complexity

The complexity of constructing a boxtree is in $O(n \log n)$, where $n$ is the number of polygons. This is supported by experiments (see Section 5).

Our algorithm takes a constant number of passes over all polygons associated with a node in order to split a set of polygons $\mathcal{F}$. Each pass is linear in the number of polygons. Every split will produce two subsets such that $\mathcal{F} = \mathcal{F}_1 \dot{\cup} \mathcal{F}_2$. Therefore, $T(n) = cn + T(\alpha n) + T((1-\alpha)n)$.

In other words, each polygon gets visited exactly $cn$ times on each level. Assuming there are $\log n$ levels, the total work over all levels is $O(n \log n)$.

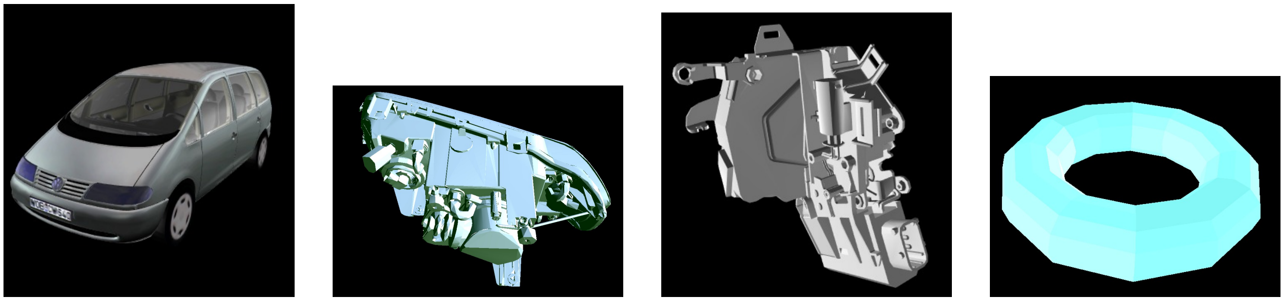

Figure 7: Some of the objects of our test suite. They are (left to right): body of a car, a car headlight, the lock of a car door (and a torus). (Data courtesy of VW and BMW)

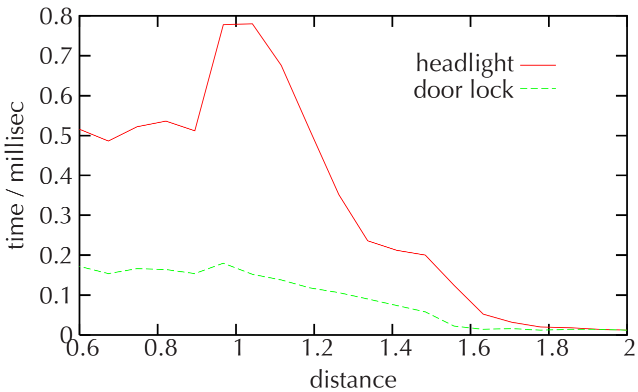


Figure 8: Our benchmark procedure computes the average collision detection time for a range of distances for each object in the test suite. In this example, the objects are the headlight with 64000 polygons and the door lock with 80000 polygons; the algorithm is the SAT lite.

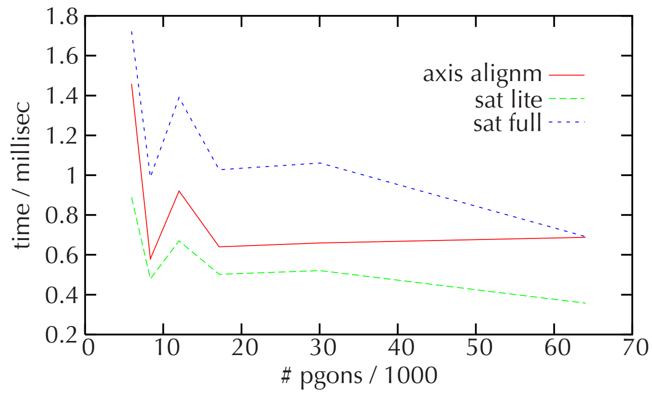


Figure 9: A comparison of the different overlap tests for pairs of boxtree nodes shows that the "SAT lite" test seems to offer the best performance.

# 5 Results

Memory requirements of different hierarchical data structures can be compared by calculating the memory footprint of one node, since a binary tree with $n$ leaves always has $2n - 1$ nodes.[8] Table 1 summarizes the number of bytes per node for different BV hierarchies (only inner nodes are considered, because leaf nodes often store information about the polygon instead of a BV).

Table 1 also compares the number of floating point operations needed for one node-node overlap test by the methods described above and three other fast hierarchical collision detection algorithms (OBB, DOP, and sphere tree). This does not, of course, imply the actual performance of any of these algorithms.

In the following, all results have been obtained on a Pentium-III with 1 GHz and 512 MB. All algorithms have been implemented in C++ on top of the scene graph OpenSG. The compiler was gcc 3.0.4.

For timing the performance of our algorithms, we have used a set of CAD objects, each of which with varying complexities (see Figure 7), plus some synthetic objects like sphere and torus. Benchmarking is performed by the following scenario: two identical objects are positioned at a certain distance $d = d_{\text{start}}$ 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 by a fixed, large number of small steps (5000). 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.

Figure 8 shows, as an example, the output of our benchmarking procedure for two objects. In order to get a better overview when comparing different algorithms, we sum-

[8] The per-node memory consumption is also a sensible measure for comparing k-ary trees, provided each BVH stores the same number of polygons per leaf.

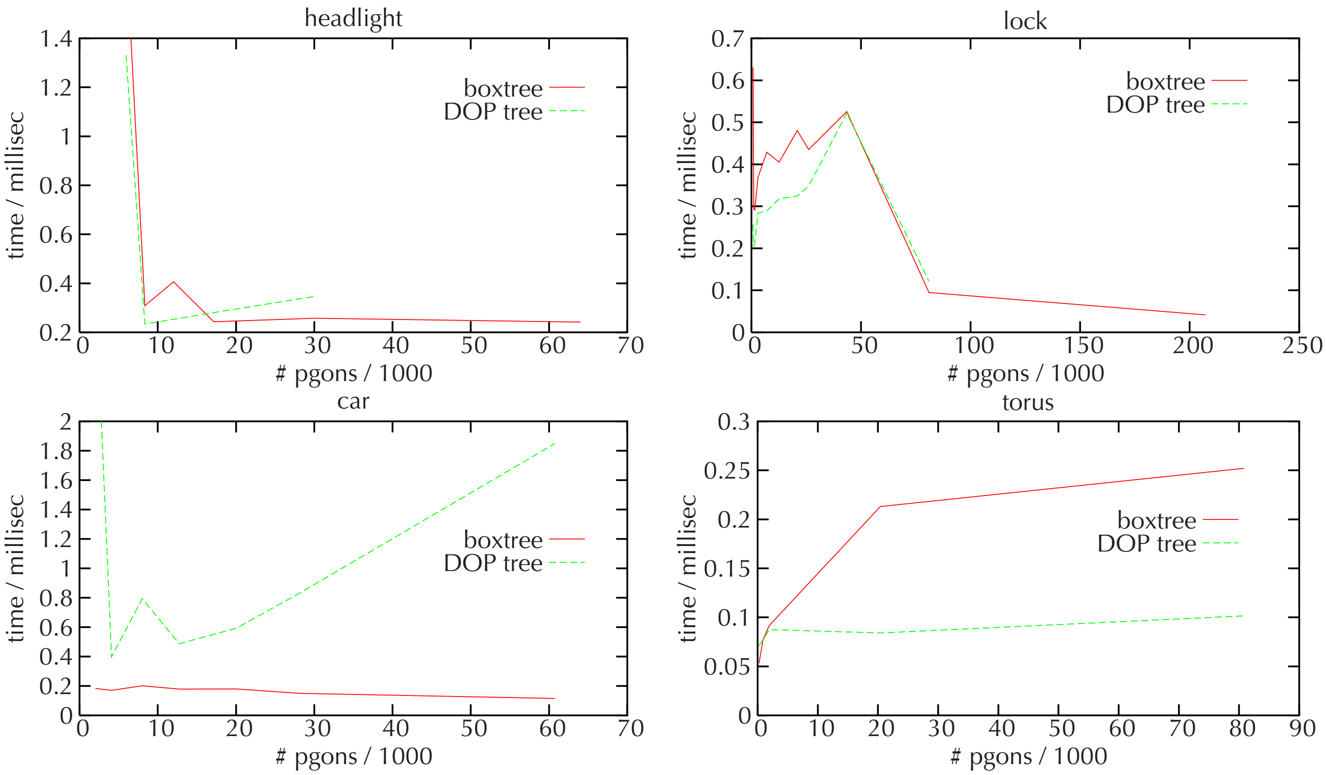


Figure 10: Runtime comparison between the restricted boxtree and the DOP tree for various objects and polygon complexities.

marize such a plot by the average time taken over that range of distances which usually occur in practical applications, such as physically-based simulation.

Figure 9 compares the performance of the various box overlap tests presented in Section 3.2 for one of the CAD objects. Similar results were obtained for all other objects in our suite. Although the full separating axis test can determine the overlap of boxes without false positives, it seems that the computational effort is not worth it, at least for axis-aligned boxes ([vdB97] has arrived at a similar conclusion for general AABB trees). From our experiments it seems that the "SAT lite" offers the best performance among the three variants.

A runtime comparison between our boxtree algorithm and DOP trees for various objects can be found in Figure 10. It seems that boxtrees offer indeed very good performance (while needing much less memory). This result puts restricted boxtrees in the same league as DOP trees [Zac98] and OBB trees [GLM96].

For sake of brevity, we have omitted our experiments assessing the performance of the lookup table approach. It has turned out that lookup tables offer a speedup of at most 8%, and they were even slower than the non-lookup table version for the lower polygon complexities because of the setup time for the tables. The reason might be that floating point and integer arithmetic operations take almost the same number of cycles on current CPUs.

As shown in Section 4, boxtrees can be built in $O(n)$. Figure 6 reveals that the constant is very small, too, so that the boxtrees can be constructed at startup time of the application.

# 6 Conclusion

We have proposed a new hierarchical BV data structure (the *restricted boxtree*) that needs arguably the least possible amount of memory among all other BV trees while performing about as fast as DOP trees. It uses axis-aligned boxes at the nodes of the tree, but it does not store them explicitly. Instead, it just stores some "update" information with each node, so that it uses, for instance, about a factor 7 less memory than OBB trees.

In order to construct such restricted boxtrees, we have developed a new algorithm that runs in $O(n)$ ($n$ is the number of polygons) and can process about 20,000 polygons per second on a 1 GHz Pentium-III.

We also propose a better theoretical foundation for the criterion that guides the construction algorithm's splitting procedure. The basic idea can be applied to all BV hierarchies.

A number of algorithms have been developed for fast collision detection utilizing restricted boxtrees. They gain their efficiency from the special features of that BV hierarchy. Benchmarking them has shown that one of them seems to perform consistently better than the others.

Several optimization techniques have been presented that further increases the performance of our new collision detection algorithm. The most important one can also be applied to most other hierarchical collision detection algorithms, and will significantly improve their performance, too.

Finally, using a suite of CAD objects, a comparison with DOP trees suggested that the performance of restricted boxtrees is about as fast in most cases.

## 6.1 Future Work

While BV trees work excellently with rigid objects, it is still an open issue to extend these data structures to accommodate deforming objects.

Our new BV hierarchy could also be used for other queries such as ray tracing or occlusion culling. It would be interesting to evaluate it in those application domains.

As stated above, most BV trees are binary trees. However, as [ES99] have observed, other arities might yield better performance. This parameter should be optimized, too, when constructing boxtrees.

So far, we have approximated the cost equation 2 only to first order (or rather, first level). By approximating it to a higher order, one could possibly arrive at a kind of "look-ahead" criterion for the construction algorithm, which could result into better hierarchies.

# 7 Acknowledgements

I would like to thank Gerrit Voss for his patient and responsive help with OpenSG and for converting some models.

# References

[ABG+00] Pankaj K. Agarwal, J. Basch, L. J. Guibas, J. Hershberger, and L. Zhang. Deformable free space tiling for kinetic collision detection. In *Proc. 4th Workshop Algorithmic Found. Robot.*, 2000. To appear.

[AK89] J. Avro, and D. Kirk. A survey of ray tracing acceleration techniques. In *An Introduction to Ray Tracing*, A. Glassner, Ed., pages 201–262. Academic Press, San Diego, CA, 1989. ISBN 0-12-286160-4.

[BCG+96] Gill Barequet, Bernard Chazelle, Leonidas J. Guibas, Joseph S. B. Mitchell, and Ayellet Tal. BOX-TREE: A Hierarchical Representation for Surfaces in 3D. *Computer Graphics Forum*, 15(3):C387–C396, C484, September 1996. ISSN 0167-7055.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 322–331, 1990.

[DDG00] M. Dickerson, C. Duncan, and M. Goodrich. K-D Trees Are Better when Cut on the Longest Side. In *LNCS 1879*, *ESA 2000*, pages 179–190, 2000.

[EL01] Stephan A. Ehmann, and Ming C. Lin. Accurate and Fast Proximity Queries Between Polyhedra Using Convex Surface Decomposition. In *Computer Graphics Forum*, vol. 20, pages 500–510, 2001. ISSN 1067-7055.

[ES99] Jens Eckstein, and Elmar Schömer. Dynamic Collision Detection in Virtual Reality Applications. In *Proc. The 7-th Int'l Conf. in Central Europe on Comp. Graphics, Vis. and Interactive Digital Media '99 (WSCG'99)*, pages 71–78. University of West Bohemia, Plzen, Czech Republic, February 1999.

[FL01] Susan Fisher, and Ming Lin. Fast Penetration Depth Estimation for Elastic Bodies Using Deformed Distance Fields. In *Proc. International Conf. on Intelligent Robots and Systems (IROS)*, 2001.

[Gei00] Bernhard Geiger. Real-Time Collision Detection and Response for Complex Environments. In *Computer Graphics International*. Geneva, Switzerland, June19-23 2000.

[GLM96] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. OBB-Tree: A Hierarchical Structure for Rapid Interference Detection. In *SIGGRAPH 96 Conference Proceedings*, Holly Rushmeier, Ed., pages 171–180. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[GS87] Jeffrey Goldsmith, and John Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications*, 7(5): 14–20, May 1987.

[HMB01] Suejung Huh, Dimitris N. Metaxas, and Norman I. Badler. Collision Resolutions in Cloth Simulation. In *IEEE Computer Animation Conf.* Seoul, Korea, November2001.

[Hub96] P. M. Hubbard. Approximating Polyhedra with Spheres for Time-Critical Collision Detection. *ACM Trans. Graph.*, 15(3):179–210, July 1996.

[KHM+98] James T. Klosowski, Martin Held, Jospeh S. B. Mitchell, Henry Sowrizal, and Karel Zikan. Efficient Collision Detection Using Bounding Volume

Hierarchies of *k*-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, January 1998.

[LAM01] Thomas Larsson, and Tomas Akenine-Möller. Collision Detection for Continuously Deforming Bodies. In *Eurographics*, pages 325–333, 2001. short presentation.

[MPT99] William A. McNeely, Kevin D. Puterbaugh, and James J. Troy. 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.

[PG95] I. J. Palmer, and R. L. Grimsdale. Collision Detection for Animation using Sphere-Trees. *Computer Graphics Forum*, 14(2):105–116, June 1995. ISSN 0167-7055.

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

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

[Zac97] Gabriel Zachmann. Real-time and Exact Collision Detection for Interactive Virtual Prototyping. In *Proc. of the 1997 ASME Design Engineering Technical Conferences*. Sacramento, California, September 1997. Paper no. CIE-4306.

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