Collision Detection as a Fundamental Technology in VR Based Product Engineering

Gabriel Zachmann

Clausthal University, Germany, zach@tu-clausthal.de

1 Introduction

In the product development process, prototyping is an essential step. Prototypes represent important features of a product, which are to be investigated, evaluated, and improved. They are used to prove design alternatives, to do engineering analysis, manufacturing planning, support managment decisions, and often just to show a product to the customers.

The vision of virtual prototyping is to use virtual reality techniques for design evaluations and presentations based on a digital model instead of physical prototypes.

In order to make the virtual objects behave exactly like real objects, fast and exact collision detection of polygonal objects undergoing rigid motions or deformations is an an enabling technology in many virtual prototyping applications (and many other simulations in computer graphics and virtual reality). It is a fundamental problem of the dynamic simulation of rigid bodies, simulation of natural interaction with objects, and haptic rendering. For example, in virtual assembly simulation, parts should be rigid and slide along each other.

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. For instance, force-feedback is a very demanding simulation because it requires the collision detection algorithms to handle at least 1000 collision queries per second.

Virtually all approaches to this problem utilize some kind of acceleration data structure. One particular requirement in the context of product engineering and virtual prototyping is that this data structure can be built fairly fast and efficiently. This is important in virtual prototyping because the manufacturing industries do not want to store any auxiliary data structures in the product data management system.

In the following we will describe some of the approaches that we have developed in recent years, targeted at differing scenarios and conditions.

2 Related Work

Bounding volume hierarchies have proven to be a very efficient data structure for rigid collision detection, and, to some extent, even for deformable objects.

One of the design choices with BV trees is the type of BV. In the past, a wealth of BV types has been explored, such as spheres [17, 36], OBBs [14], DOPs [26, 47], Boxtrees [48, 2], AABBs [44, 28], and convex hulls [11].

Another alternative are space-subdivision approaches, for instance by an octree [20] or a voxel grid [32]. In general, non-hierarchical data structures seem to be more promising for collision detection of

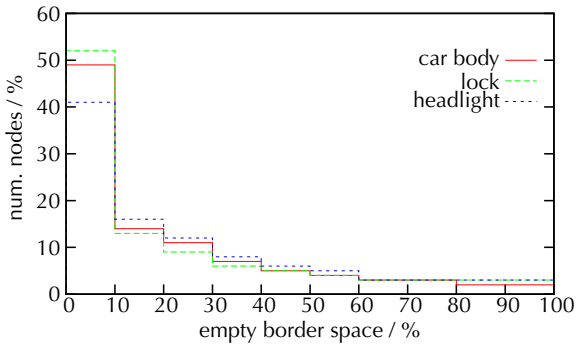

Fig. 1. We have found that for most nodes in an AABB tree there is very little empty space between them and their parent on most sides.

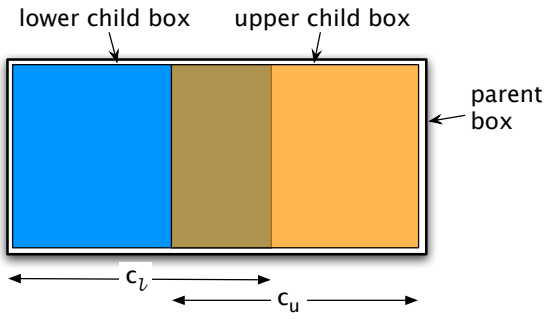

Fig. 2. Child nodes are obtained from the parent node by splitting off one side of it. The drawing shows the case where a parent has a lower and an upper child that have coplanar splitting planes.

deformable objects [4, 18, 13], although some geometric data structures suggest a natural BV hierarchy [29]. Deformable collision detection is not the focus of our work presented here.

Some good surveys on collision detection can be found in [42, 41, 30, 35]. In addition, there two books [12, 45].

A clever way to utilize graphics hardware was presented by [27]. Based on the observation that an intersection can occur if and only if an edge of one object intersects the other one, they render edges of one object and polygons of the other. This even works for deformable geometry. Unlike many previous approaches, objects do not need to be convex. However, they must still be closed. Furthermore, it seems to work robustly only for moderate polygon counts.

A hybrid approach was proposed by [15]. Here, the graphics hardware is used only to detect potentially colliding objects, while triangle-triangle intersections are performed in the CPU. While this approach alleviates previous restrictions on object topology, its effectiveness seems to degrade dramatically when the density of the environment increases.

The approach presented by [3] can compute the penetration depth using graphics hardware, but only for convex objects.

Earlier image-based methods include [40, 33, 5, 31, 6].

3 Minimal Hierarchical Collision Detection

In this section, we present a bounding volume hierarchy (BVH) that allows for extremely small data structure sizes while still performing collision detection as fast as other classical hierarchical algorithms in most cases [48]. 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. It has been invented independently several times and under different names [48, 34, 46].

We also propose a criterion to be used during the construction of the BVHs that is formally derived. The idea of the argument is general and can be applied to other bounding volume hierarchies as well.

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 means that a parent box touches

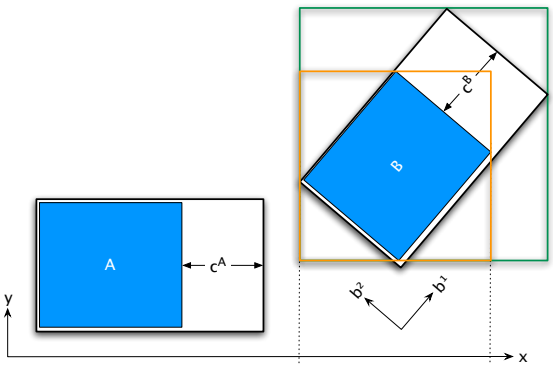

Fig. 3. Only one value per axis needs to be recomputed for the overlap test.

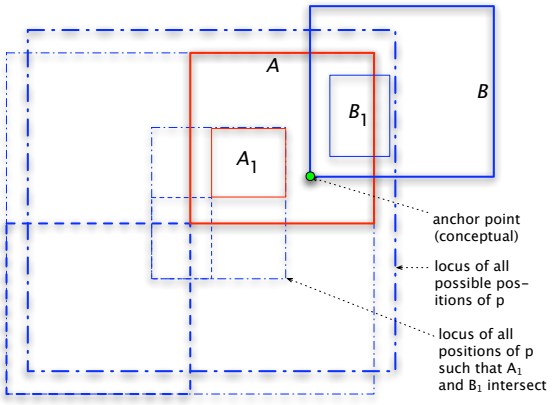

Fig. 4. By estimating the volume of the Minkowski sum of two BVs, we can derive an estimate for the cost of the split of a set of polygons associated with a node.

each child box on 3 sides on average, because of the way the set of polygons is partitioned (usually alog one axis). We have tried to quantify this observation further: in the AABB tree of three representative objects, we have measured the empty space between each 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%.

Consequently, our hierarchy never stores a box explicitly. Instead, each node stores only one plane that is perpendicular to one of the three axes. Overall, a node consists of just one float, representing the distance from one of the sides of the parent box (see Figure 2), plus the ID of the axis, plus one pointer to the left child (assuming siblings are stored contiguously).

Because each box in such a hierarchy is restricted on most sides, we call this a *restricted boxtree*.

3.2 Overlap Test

The overlap test between a pair of BVs is the elementary step in all hierarchical collision detection algorithms, which accounts for the majority of the runtime.

With the BVs of the restricted boxtree, there are a number of algorithms possible to test a given pair for overlap. In the following, we will present the one that we have found to be the most efficient one. Basically, it is the so-called "SAT lite" test [44], the application of which to the restricted boxtree will be detailed in the following.

Assume we are given two boxes A and B, and that (b^1, b^2, b^3) is the coordinate frame of box B represented in A's object space (see Figure 3).

First, we compute an axis-aligned box (in the coordinate system of A) that tightly encloses B, which is specified by $(l, h) \in \mathbb{R}^3 \times \mathbb{R}^3$ (this is equivalent to projecting B onto the three axes of A).

We already know that the parent boxes of A and B must overlap. Notice that we need to compute only 3 values of (l, 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 $b \in \{b^1, b^2, b^3\}$, which is distance c away from the corresponding upper side of the parent box (i.e., B is a lower child).

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

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

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

and

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

Similarly, l_x, h_x can be computed if B is an upper child, and analogously the new value along the other 2 axes.

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 comparison along the x axis to be done is

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

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.

Eventually, we have thus performed the "SAT lite" test on 3 axes. Then, we reverse the roles of A and B, which completes the test along all 6 axes.

4 A General Heuristic for Constructing Good BVHs

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.

The generally adopted aproach is top-down, i.e., we start with the complete set of polygons. In the first step, we enclose this by a BV of the chosen type. Then, we choose a so-called *splitting axis* (this could be one of the coordinate axes). Then, we make one plane sweep along that axis, where each plane position yields a partition of the given set of polygons. Once we have determined the right plane position, we can recursively process the two sets of polygons.

The main question during this process is which plane position yields the best partition of the set of polygons, in the sense of fastest collision queries on average.

In the following, we will derive a heuristic, that can guide the determination of the plane's position along the splitting axis.

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

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

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).

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

$$\begin{aligned} C(A,B) =& 1 + P(A_1,B_1) + P(A_1,B_1)P(A_{11},B_{11}) \\ &+ P(A_1,B_1)P(A_{12},B_{11}) + \ldots + \\ &P(A_1,B_2) + P(A_1,B_2)P(A_{11},B_{21}) \\ &+ \ldots \end{aligned} \tag{2}$$

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 and approximating it by the number of primitives, which gives

$$C(A,B) \approx \sum_{i,j=1,2} P(A_i,B_j)\cdot N(A_i)\cdot N(B_j) \tag{3}$$

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 4.[2] Because B_1 is a child of B, we know that the anchor point of B_1 must lie somewhere in the Minkowski sum $A \oplus B \oplus d$, where $d = \text{anchor}(B_1) - \text{anchor}(B)$. Since A_1 is inside A and B_1 inside B, we know that $A_1 \oplus B_1 \subset A \oplus B \oplus d$. So, for arbitrary convex BVs the probability of overlap is

$$P(A_1,B_1) = \frac{\text{Vol}(A_1 \oplus B_1)}{\text{Vol}(A \oplus B \oplus d)} = \frac{\text{Vol}(A_1 \oplus B_1)}{\text{Vol}(A \oplus B)} \tag{4}$$

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

$$\begin{aligned} \text{Vol}(A) + \text{Vol}(B) + \frac{2}{\alpha}\sqrt{\text{Vol}(A)\,\text{Vol}(B)} \leq \\ \text{Vol}(A \oplus B) \leq \\ \text{Vol}(A) + \text{Vol}(B) + 2\alpha\sqrt{\text{Vol}(A)\,\text{Vol}(B)} \end{aligned} \tag{5}$$

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

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

yielding

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

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

[2] 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.

5 Object-Space Interference Detection on Programmable Graphics Hardware

Currently, the performance of graphics hardware (GPUs) is progressing faster than general-purpose CPUs. The main reason is an architecture that combines *stream processing* [19] and SIMD processing. In addition, the programmability of the GPU has increased drastically over the past few years. Overall, today a programmer can write *kernels* for all stages of the graphics pipeline that are automatically executed in parallel on an indefinite number of processing units. This has led many researchers to investigate exploitation of the GPU for other computations, such as matrix computations, ray tracing, distance field computation, etc.

Many algorithms have been proposed to utilize graphics hardware for the problem of collision detection. They can be classified into techniques that make use of the depth and stencil buffer tests, and those that compute discrete distance fields. In any case, the problem is approached in *image space*, i.e., it is discretized.

To our knowledge, we have presented the first algorithm that performs collision detection completely on the GPU and in *object space* [16]. Our method is based on previous hierarchical collision detection algorithms. However, all computations during this traversal, including the final triangle intersection tests, are performed in vertex and fragment programs on the GPU. The algorithm has no requirements on the shape, topology, or connectivity of the polygonal input models.

This method will be outlined in the following. We will skip the details of the triangle intersection test and the box overlap test on the GPU and refer the interested reader to [16].

5.1 Hierarchical Interference Detection

Instead of traditional depth-first traversals for collision detection on the CPU, we use a breadth-first traversal scheme. To be able to traverse the AABB trees efficiently, the trees have to be balanced. Furthermore, since leaf nodes will be handled differently than inner nodes by our algorithm, we require that there are no leaf nodes in the tree other than at the lowest hierarchy level

5.2 Outline of the Algorithm

Since we traverse the tree breath-first and since at each hierarchy level only certain node pairs are to be visited, we have to store the indices of these node pairs temporarily during the traversal. For this purpose, we use a 2D buffer, which we will refer to in the following as *node pair index map*.

This buffer contains an array of index sets as follows. Let $L_j = \{i \mid \text{AABB}(S_i) \text{ overlaps } \text{AABB}(T_j)\}$. Putting the contents of set L_j in the 2D buffer at row j, stored successively starting at the first pixel in this row, the complete buffer consists of m horizontal lines of different lengths. The lengths of all these lines (or, more precisely, their start and end points) are stored in a vertex array.

In addition, we require a second temporary 2D buffer, that we call *overlap count map*. This buffer consists of multiple levels, exactly as much as there are hierarchy levels in the AABB trees. As the node pair index map, also each level of the overlap count map consists of m horizontal lines of different lengths. The contents of the overlap count map are constructed during the AABB tree traversal at each hierarchy level L as follows.

At first, all those AABB node pairs that are to be visited at the considered level L are checked for overlap. Each such node pair corresponds to one entry in the overlap count map at level L. If the AABB overlap test of a certain node pair was positive and thus the corresponding child nodes are to be visited when processing the next hierarchy level, the number of these child nodes is written into the corresponding entry of the overlap count map. Otherwise the entry of the overlap count map is set to 0. How this is done using the GPU is described in [16].

If this step results in a map containing only 0-entries (what can be determined using an occlusion query), all AABB overlap tests have been negative, and therefore the two objects definitively do not collide.

Otherwise, the AABB tree traversal is continued as follows. Before the iteration proceeds to the next hierarchy level, the node pair index map has to be updated, as well as the vertex array containing the start and end points of the horizontal lines contained in this map. This is done in two steps. First, the vertex array is updated, as well as levels $0, \ldots, L-1$ of the overlap count map. Second, the information contained in levels $0, \ldots, L$ of the overlap count map is used to construct the node pair index map required as input for processing the next hierarchy level. These two steps are explained in detail in Section 5.3.

The whole process is repeated for all hierarchy levels as long as there are positive AABB overlap tests. If the last hierarchy level is reached, instead of testing AABB overlaps, triangle intersection tests are performed on the GPU for all leaf node pairs that have to be visited according to the node pair index map. Using an occlusion query, we obtain the number of intersecting triangle pairs for the considered objects. If required, the actual list of intersecting triangles can be obtained via read-back from graphics memory.

The outline of the overall algorithm is summarized in Fig. 5.

5.3 Generating the Node Pair Index Map

We construct the new node pair index map and the corresponding vertex array in multiple passes using the following technique.

The length of each horizontal line in the new node pair index map corresponds to the number of nodes of level $L+1$ for whose parent nodes the AABB overlap test was positive. By construction, this number is equal to the sum of all values in level L of the overlap count map at the corresponding row.

Therefore, we can construct the vertex array by summing up these values. In analogy to the construction of 1D MIP maps on the GPU, we do this by constructing the levels $L-1, \ldots, 0$ of the overlap count map as follows. Each level $i = L-1, \ldots, 0$ of this map consists of $2^i \times m$ entries, each of which is calculated by summing up two values from level $i+1$ (see Fig. 6).

We store the entries of level 0 of the overlap count map, each of which corresponds to the sum of all values in the corresponding row of level L, in a vertex array, such that they can be used as vertex program input for the AABB overlap tests at hierarchy level $L+1$. In our current implementation, this is accomplished by transferring the data from the render target texture directly to the vertex array using the `EXT_pixel_buffer_object` OpenGL extension.

Next, we construct the new node pair index map for hierarchy level $L+1$ in L passes as follows.

The basic idea is that for every row of the node pair index map the nth entry corresponds to the nth AABB tree node of level $L+1$ that is to be visited. This node can be found be traversing the AABB tree starting at the root node. Note, that the first entry of level 1 of the overlap count map contains the number of nodes of level $L+1$ to be visited that are reached from the root node via its first child node. Therefore, this value decides whether we must proceed to the first or to the second child of the root node to reach the searched node. Then, this step is repeated using levels $2, \ldots, L$ of the overlap count map.

This technique to construct the new node pair index map is realized on the GPU as follows. We need a temporary buffer of the same size as the node pair index map that we are going to construct, consisting of two components per entry. The first component, called *current node index* in the following, is used to store the indices of AABB tree nodes that are visited during the traversal. The second component, called *current child index* in the following, corresponds to n if we search the nth node of level $L+1$ to be visited that is reached from the node specified by the current node index.

At the beginning, this temporary buffer is initialized as follows: In each row of the texture, the current child index is numbered consecutively starting with 0. The current node index is initialized to 0

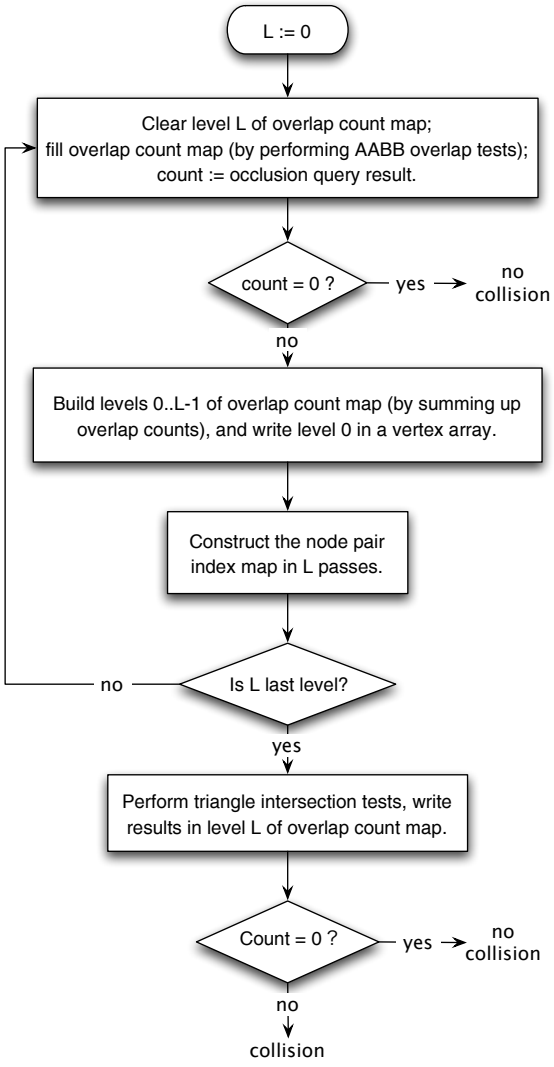

Fig. 5. The outline of our approach to object-space collision detection on the GPU.

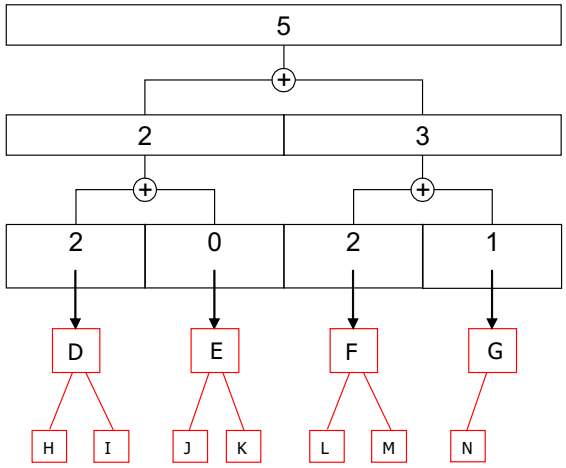

Fig. 6. Summing up the overlap counts.

and thus addresses the root node. Each row of the temporary buffer is assumed to be of the corresponding length stored in the vertex array.

We use a fragment program that does the following for each pass $i = 1, \ldots, L$: First, the value from the overlap count map of level i at the index that equals 2*current node index* is read. Then, this value (*overlap count*) is compared against the current child index. If it is greater, the current node index is replaced by 2*current node index* in the temporary buffer, and the current child index remains unchanged. Otherwise, the current node index is replaced by 2*current node index*$+1$, and the overlap count is subtracted from the current child index, see Fig. 7.

After these L passes, the new node pair index map can be obtained based on the values in the temporary buffer and the current contents of the node pair index map as follows. For each entry of the map, we identify the actual corresponding AABB tree node by accessing the current contents of the node pair index map at the index specified by the current node index from the temporary buffer. Then we store the

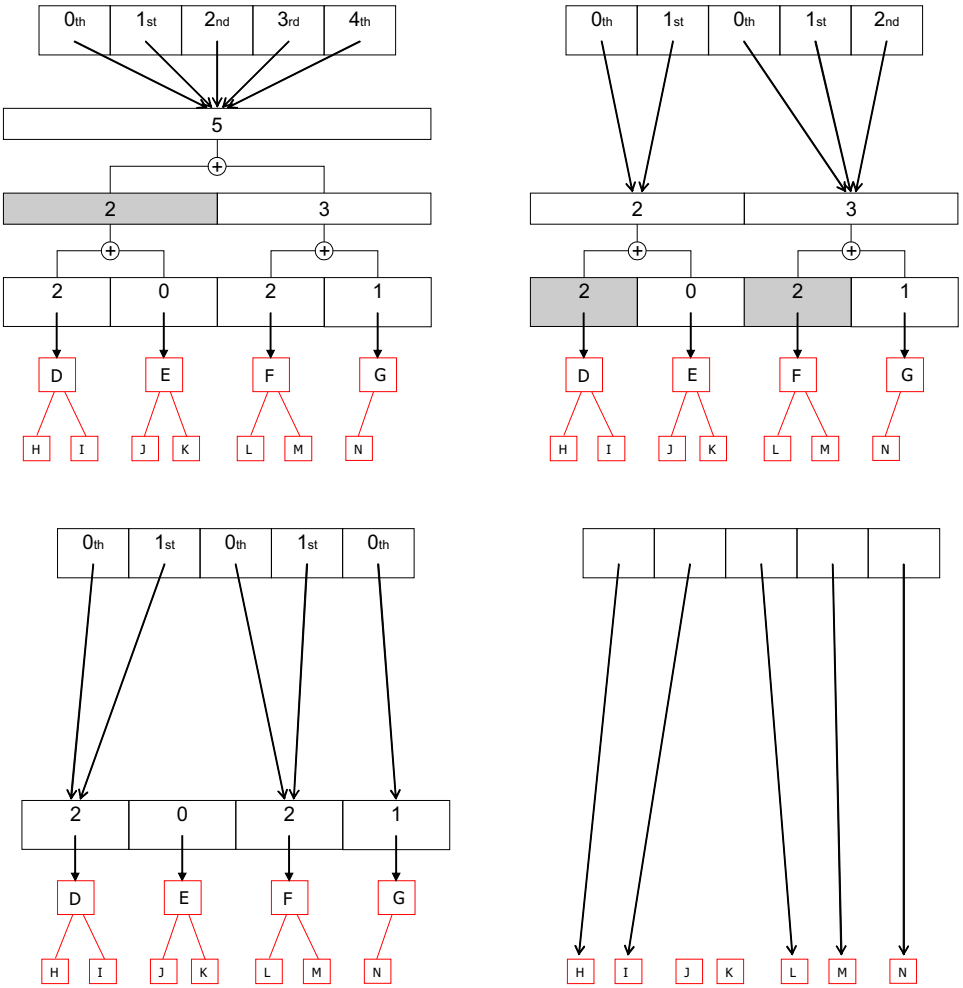

Fig. 7. Construction of the node pair index map in multiple passes. The values on gray background are the overlap counts that are compared to the current child indices shown at the top row.

index of its first or second child, depending on the current child index from the temporary buffer, into the new node pair index map. This step is shown in the lower right of Fig. 7.

Instead of doing this last step in an additional render pass, it can be incorporated directly into the fragment programs used for box overlap and triangle intersection testing, such that the construction of the node pair index map requires L passes in total.

Note that the multi-pass construction technique described above is also suited for graphics hardware with dependent texture read limit (like the current ATI Radeon GPUs). On hardware without such a limit, the number of passes could be reduced by combining multiple of these passes into a single one, provided that total number of textures used in such a pass does not exceed the corresponding hardware limit.

5.4 Results

We implemented the method in C++ using OpenGL on a NVIDIA GeForce FX 5900 GPU.

We tested the performance of our algorithm using a test scenario similar to that of [49]: two identical objects placed at some distance from each, one of them rotating and slowly approaching the other.

This test was performed on a set of CAD objects with varying complexities. To compare the performance of GPU and CPU based collision detection methods, we also implemented the AABB tree traversal approach on the CPU using an identical traversal scheme and ran the same tests on that implementation.

The timings include the determination of all intersecting triangle pairs as well as the read-back of its indices from graphics memory in case of the GPU implementation.

When comparing the performance of the individual steps of the algorithm between its GPU and CPU implementations, it turns out that in the GPU implementation the box overlap and triangle intersection tests perform up to four times faster especially at the lower hierarchy levels. However, this speed-up is reduced by the overhead of the node pair index map generation, which is larger in the GPU implementation. Overall, in general our current GPU implementation is slightly faster than the CPU implementation.

For more details, we would like to refer the interested to [49].

6 Time-Critical Collision Detection Using ADB-Trees

It has often been noted previously, that the *perceived quality* of a virtual environment and, in fact, most interactive 3D applications, crucially depends on the real-time response to collisions [43]. At the same time, humans cannot distinguish between physically correct and *physically plausible* behavior of objects (at least up to some degree) [8]. [3]

Therefore, we have introduced the novel framework of collision detection using an average-case approach, thus extending the set of techniques for plausible simulation [22, 21]. To our knowledge, this is the first time that the *quality* of collision detection can be decreased in a controlled way (while increasing the speed), such that a numeric *measure* of the quality of the results is obtained (which can then be related to the perceived quality). The methods presented in this section can be applied to virtually any hierarchical collision detection algorithm.

Conceptually, the main idea of the new algorithm is to consider *sets of polygons* at inner nodes of the BV hierarchy, and then, during traversal, check pairs of sets of polygons. However, we neither check pairs of polygons derived from such a pair of polygon sets, nor store any polygons with the nodes. Instead, based on a small number of parameters describing the *distribution* within the polygon sets, we will derive an estimation of the probability that there *exists* a pair of intersecting polygons. This has two advantages:

1. The application can control the runtime of the algorithm by specifying the desired "quality" of the collision detection (to be defined later).
2. The probabilities can guide the algorithm to those parts of the BV hierarchies that allow for faster convergence of the estimate.

6.1 Overview of our Approach

The idea of our algorithm is to guide and to abort the traversal by the *probability* that a pair of BVs contains intersecting polygons. The design of our algorithm was influenced by the idea to develop an algorithm that works well and efficient for most practical cases — in other words, that works well in the average case. Therefore, we estimate the probability of a collision within a pair of BVs by some

[3] Analogously to rendering, a number of human factors determine whether or not the "incorrectness" of a simulation will be noticed, such as the mental load of the viewing person, cluttering of the scene, occlusions, velocity of the objects and the viewpoint, point of attention, etc.

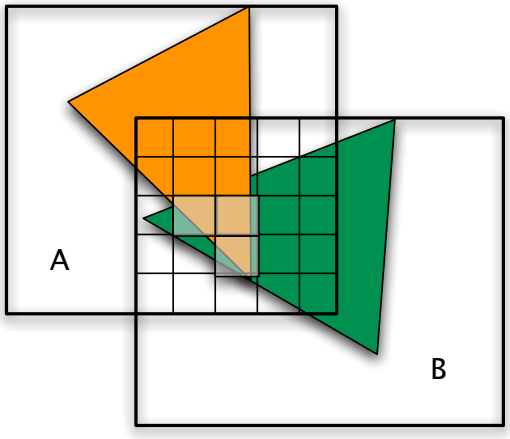

Fig. 8. We partition the intersection volume by a grid. Then, we determine the probability that there are *collision cells* where polygons of different objects *could* intersect (highlighted in grey). For the sake of illustration, only one polygon of each BV is shown.

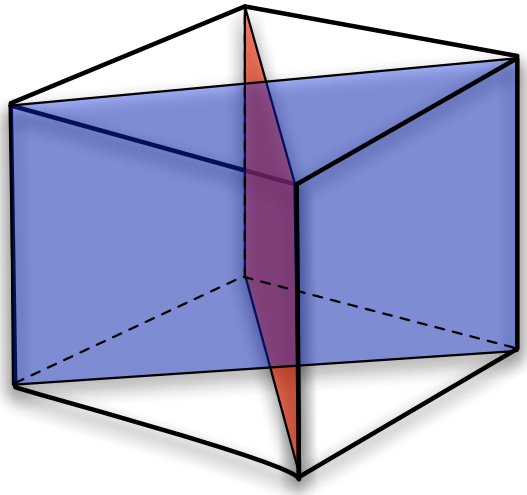

Fig. 9. A cubic collision cell c with side length a. $\mathrm{Area}_c(A)$ and $\mathrm{Area}_c(B)$ must be at least $\mathrm{MaxArea}(c) = a^2\sqrt{2}$, which is exactly the area of the two quadrangles.

characteristics about the average distribution of the polygons, but we do not use the exact positions of the polygons during the collision detection.

Conceptually, the intersection volume of BVs A and B, $A \cap B$, is partitioned into a regular grid (see Figure 8). If a cell contains *enough* polygons of one BV, we call it a *possible collision cell* and if a cell is a possible collision cell with respect to A and also with respect to B, we call it a *collision cell* (a more precise definition is given in Section 6.2). Given the total number of cells in $A \cap B$, the number of possible collision cells from A and B, resp., lying in $A \cap B$, we can compute the probability that there are at least x collision cells in $A \cap B$. This probability can be used to estimate the probability that the polygons from A and B intersect. For the computations, we assume that the probability of being a possible collision cell is evenly distributed among all cells of the partitioning because we are looking for an algorithm that works well in the average case where the polygons are uniformly distributed in the BVs.

An outline of our traversal algorithm is shown in Figure 10. Function computeProb estimates the probability of an intersection between the polygon sets of two BVs. By descending first into those subtrees that have highest probability, we can quickly increase the confidence in the result and determine the end of the traversal. Basically, we are now dealing with priorities of pairs of nodes, which we maintain in a priority queue. It contains only pairs whose corresponding polygons can intersect. The queue is sorted by the probability of an intersection. Instead of a recursive traversal, our algorithm just extracts the front node pair of the queue and inserts a number of child pairs.

The quality and speed of the collision detection strongly depends on the accuracy of the probability computation. Several factors contribute to that, such as the kind of partitioning and the size of the polygons relative to the size of the cells.

There are two other important parameters in our traversal algorithm, p_{min} and k_{min}, that affect the quality and the speed of the collision detection. Both can be specified by the application every time it performs a collision detection. A *pair of collision nodes* is found if the probability of an intersection between their associated polygons is larger than p_{min}. A collision is reported if at least k_{min} such pairs

```
traverse(A, B)
priorityQueue q;  k:=0
q.insert(A, B, 1)
while q is not empty do
    A, B := q.pop;
    for all children A[i] and B[j] do
        p := computeProb(A[i], B[j])
        if  p ≥ p_min  then
            k ++
            if  k ≥ k_min then
                return "collision"
            end if
        end if
        if  p > 0 then
            q.insert(A[i], B[j], p)
        end if
    end for
end while
return "no collision"
```

Fig. 10. Our algorithm traverses two BV hierarchies by maintaining a priority queue of BV pairs sorted by the probability of an intersection.

have been found. The smaller p_{min} or k_{min}, the shorter is the runtime and, in most cases, the more errors are made.

6.2 Notation and Definitions

For the sake of accuracy and conciseness, we introduce the following notation and definitions. We treat the terms *bounding volume* (BV) and *node* of a hierarchy synonymous. A and B will always denote BVs of two different hierarchies.

All polygons of the object contained in BV A or intersecting A are denoted as $P(A)$. Let c be a cell of the partitioning of $A \cap B$. The total area of all polygons in $P(A)$ clipped against cell c is denoted as $\text{Area}_c(A)$. $\text{MaxArea}(c)$ denotes the area of the largest polygon that can be contained completely in cell c.

Definition 1 (possible collision cell) *Given a BV* A *and a cell* c. c *is a possible collision cell, if* $\text{Area}_c(A) \geq \text{MaxArea}(c)$.

Definition 2 (collision cell) *Given two intersecting BVs* A *and* B *as well as a partitioning of* $A \cap B$. *Then,* A *and* B *have a (common) collision cell iff* $\exists c : \text{Area}_c(A) \geq \text{MaxArea}(c) \wedge \text{Area}_c(B) \geq \text{MaxArea}(c)$ *(with suitably chosen* $\text{MaxArea}(c)$*).*

Definitions 1 and 2 are actually the first steps towards computing the probability of an intersection among the polygons of a pair of BVs. In particular, definition 2 is motivated by the following observation. Consider a cubic cell c with side length a, containing exactly one polygon from A and B, resp. Assuming $\text{Area}_c(A) = \text{Area}_c(B) = \text{MaxArea}(c)$, then we must have exactly the configuration shown in Figure 9, i.e., an intersection, if we choose $\text{MaxArea}(c) = a^2\sqrt{2}$. Obviously, a set of polygons is not planar (usually), so even if $\text{Area}_c(A) > \text{MaxArea}(c)$ there might still not be an intersection. But since almost all practical objects have bounded curvature in most vertices, the approximation by a planar polygon fits better and better as the polygon set covers smaller and smaller a surface of the object.

Definition 3 ($LB(c_{A\cap B})$) *Given an arbitrary collision cell* c *from the partitioning of* $A\cap B$. *A lower bound for the probability that a collision occurs in* c *is denoted as* $LB(c_{A\cap B})$.

Let us conclude this subsection by the following important definition.

Definition 4 ($Pr(c(A\cap B)\geq x)$) *The probability that at least* x *collision cells exist in* $A\cap B$ *is denoted as* $Pr[c(A\cap B)\geq x]$.

Overall, given the probability $Pr[c(A\cap B)\geq 1]$, a lower bound for the probability that the polygons from A and B intersect is given by

$$Pr[P(A)\cap P(B)\neq\varnothing]\geq Pr[c(A\cap B)\geq 1]\cdot LB(c_{A\cap B}). \tag{7}$$

A better lower bound is given below.

6.3 ADB-Trees

As mentioned before, our approach is applicable to virtually all BV hierarchies by augmenting them with a simple description of the distribution of the set of polygons. We call the resulting hierarchies *ADB trees*. In the following, we explicitly mention the type of BV only if necessary.

Our function computeProb(A, B) needs to estimate the probability $Pr[c(A\cap B)\geq x]$ that is defined in the previous section. However, partitioning $A\cap B$ during runtime is too expensive.

Therefore, we partition each BV during the construction of the hierarchy into a fixed number of cuboidal cells, and then we count the number of *possible collision cell* according to Definition 1 and store it with the node. Note that, thanks to our average-case approach making the assumption that each cell of the partitioning has the same probability to be a possible collision cell, we are not interested in exactly which cells are possible collision cells, but only in their number. As a consequence, this additional parameter per node incurs only a very small increase in the memory footprint of the BV hierarchy, even when utilizing very "light-weight" nodes such as spheres [17] or restricted boxes [48]. It is, of course, computed during preprocessing after the construction of the BV hierarchy.

Note that we do not need to store any polygons or pointers to polygons in inner nodes! A possible intersection is determined solely based on the probabilities described so far.

In addition to the ADB-trees, we will need a number of lookup tables in order to compute $Pr[c(A\cap B)\geq x]$ efficiently (see Section 6.4). Fortunately, they do not depend on the objects nor on the type of BV, so we need to precompute the lookup tables only once.

6.4 Probability Computations

In this section, we explain the computation of the probability $Pr[c(A\cap B)\geq x]$ and its usage. It can be computed from the following 3 parameters only:

$$\begin{aligned} s &= \#\text{ cells contained in } A\cap B,\\ s_A &= \#\text{ possible collision cells from } A \text{ in } A\cap B,\\ s_B &= \#\text{ possible collision cells from } B \text{ in } A\cap B. \end{aligned}$$

For the details about how to compute these during the hierarchy traversal, we would like to refer the interested reader to [22, 21].

Given a partitioning of $A\cap B$ and the numbers s, s_A, s_B, the question is: what is the probability that at least x of the s cells are possible collision cells of the s_A cells and are *also* possible collision cells of the s_B cells? This is the probability that at least x collision cells exit.

Note that the $s_A + s_B$ possible collision cells are randomly but not independently distributed among the s cells: obviously, it can never happen that two or more of the s_A or s_B, resp., possible collision cells are distributed on the same cell, i.e., s_A possible collision cells are distributed on exactly the same number of cells of the partitioning. This problem can be stated more abstractly and generalized by the following definition.

Definition 5 ($\Pr(\#\ \text{filled bins} \geq x)$) *Given u bins, v blue balls, and w red balls. The balls are randomly thrown into the u bins, whereby a bin never gets two or more red or two or more blue balls. The probability that at least x of the u bins get a red and a blue ball is denoted as* $\Pr[\#\ \textit{filled bins}\ \geq x]$.

If $u = s, v = s_A$ and $w = s_B$, this definition is related to our original problem by the following observation, because we assume that each cell of the partitioning has the same probability of being a possible collision cell.

Observation 1
$\Pr[c(A \cap B) \geq x] \approx \Pr[\#\ \textit{filled bins}\ \geq x]$.

Now, let us determine $\Pr[\#\ \text{filled bins}\ \geq x]$. The probability, that exactly t of the u bins get a red and a blue ball, is[4]

$$\frac{\binom{w}{t}\binom{u-w}{v-t}}{\binom{u}{v}}$$

Thus, the probability that at least x of the u bins get a red and a blue ball, is

$$\Pr[\#\ \text{filled bins}\ \geq x] = 1 - \sum_{t=0}^{x-1} \frac{\binom{w}{t}\binom{u-w}{v-t}}{\binom{u}{v}} \tag{8}$$

Until now, for computing a lower bound for $\Pr[P(A) \cap P(B) \neq \varnothing]$ (see Equation 7) we have only used the probability that at least *one* collision cell exists in $A \cap B$. Although the algorithm achieves very good quality using only that probability, we can improve the lower bound by using the probability that *several* collision cells are in the intersection, i.e., by using $\Pr[c(A \cap B) \geq x]$, $x > 1$.

Obviously, $\Pr[c(A \cap B) \geq x]$ decreases as x increases. But the more collision cells (with high probability) in the intersection volume are, the higher the probability is that a collision really takes place in the pair of BVs.

Let a partitioning of $A \cap B$ be given. Then, a lower bound for the probability $\Pr[P(A) \cap P(B) \neq \varnothing]$ can be computed by

$$\Pr[P(A) \cap P(B) \neq \varnothing] \geq \max_{x \leq \min\{s_A, s_B\}} \left\{ \Pr[c(A \cap B) \geq x] \cdot \left(1 - (1 - LB(c_{A\cap B}))^x\right) \right\} \tag{9}$$

because $\left(1 - (1 - LB(c_{A\cap B}))^x\right)$ denotes a lower bound for the probability that in at least one of the x collision cells a collision takes place. Note that, if we use the approximation shown in Observation 1, this is not a lower bound any longer, but *only* a good estimation of it.

In practice, it is sufficient to evaluate Equation 9 for small x, because for realistic values of s, s_A, s_B, and $LB(c_{A\cap B})$ it assumes the maximum at a small x. Consequently, we bound x by a small number (e.g., 10) in Equation 9.

[4] Explanation: Let us assume, the w red balls have already been thrown into the u bins. Now, the question is: what is the probability that t of the v blue balls are thrown into bins containing a red ball? $\binom{u}{v}$ is the number of possibilities to distribute the v blue balls to the u bins. The number of possibilities that t of the v blue balls are distributed to bins already filled with a red ball is $\binom{w}{t}$. And the remaining $v - t$ blue balls have to be distributed simultaneously to the $u - w$ empty bins.

Overall, in order to get a better lower bound for the collision probability, $Pr[P(A) \cap P(B) \neq \emptyset]$ can be computed by Equation 9 instead of Equation 7.

It remains to derive $LB(c_{A\cap B})$, which denotes a lower bound for the probability of an intersection in an arbitrary collision cell from the partitioning of $A \cap B$ (see Defintion 3).

For most real-world models, we can assume that the curvature of the surfaces is bounded (possibly except in a finite number of curves on the surface). Now consider a collision cell $c \subset A \cap B$, i.e., it contains two sets of polygons, $P(A)$ and $P(B)$. Because we are looking for a lower bound, we can only assume that $Area_c(A)$ and $Area_c(B)$ are equal to $MaxArea(c)$.

Suppose the cell c is large compared to the size of the objects. Then, the possible curvature of the surface parts in $P(A)$ and $P(B)$ can lead to convex hulls of $P(A)$ and $P(B)$ that are small with respect to c. Therefore, the probability of an intersection is small.

On the other hand, as the traversal of the BV hierarchies reaches lower levels, c becomes small compared to the size of the objects. Then, because of the bounded curvature, $P(A)$ and $P(B)$ can be approximated better and better by two plane polygons. In the extreme, we reach exactly the situation shown in Figure 9. Therefore, we estimate the lower bound $LB(c_{A\cap B})$ by

$$LB(c_{A\cap B}) \approx \frac{d_A + d_B}{d_{max_A} + d_{max_B}}$$

where d_A, d_B are the depth of node A, B in their respective BV hierarchies, and d_{max_A}, d_{max_B} are the maximum depths. In other words, the larger the depth of the nodes of A and B, the smaller the BVs, and the larger is the probability that the polygons in a collision cell intersect.

Note that if we approximate $LB(c_{A\cap B})$ as described above, the lower bound given in Equation 9 is not a lower bound any longer, but *only* a good estimation of it.

6.5 Intersection Volume

Since the probability is computed once per node pair during the hierarchy traversal, we need a fast way to compute $Vol(A \cap B)$. However, an exact computation is prohibitively expensive for most BVs (except spheres), even for cubes, because they are not aligned with each other. So we need to resort to approximations.

The idea of our proposed method is shown in Figure 11.

Given two bounding boxes of (nearly) the same size at a certain distance d that are not necessarily aligned with each other. Then, an upper bound of their intersection volume is given by two BVs of the same size with the same distance d, that are aligned as one of the 3 cases shown in the Figure 11. So we only need to tentatively compute the intersection volume V_i for each of them. Then, $\max\{V_1, V_2, V_3\}$ is an upper bound of the intersection volume. In the following, let a, b, and c denote the side lengths of the BVs, where $a \geq b \geq c$. Then

$$V_1 = a \cdot b \cdot c \cdot (1 - \frac{d}{\sqrt{a^2}})^1 = (a - d) \cdot b \cdot c$$
$$V_2 = a \cdot b \cdot c \cdot (1 - \frac{d}{\sqrt{a^2 + b^2}})^2$$
$$V_3 = a \cdot b \cdot c \cdot (1 - \frac{d}{\sqrt{a^2 + b^2 + c^2}})^3$$

To prove this claim, one has to perform two steps. First of all, assume that the boxes are axis-aligned. Without loss of generality, let one box be centered at the origin and the other centered at $P = (x, y, z)$. Then, the intersection volume is $V = (a-x)(b-x)(c-x)$, which has to be maximized under the constraint

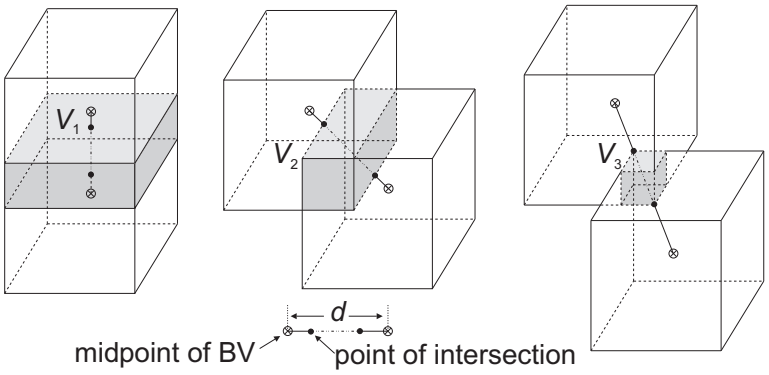

Fig. 11. We estimate the intersection volume for two not necessarily aligned BVs by the maximum of three corresponding aligned cases, $\max\{V_1, V_2, V_3\}$, which is an upper bound. Note that, also for non-cubic BVs, the line through the midpoints of the two BVs has to intersect the vertices and/or edges of the BV of the intersection volume as shown.

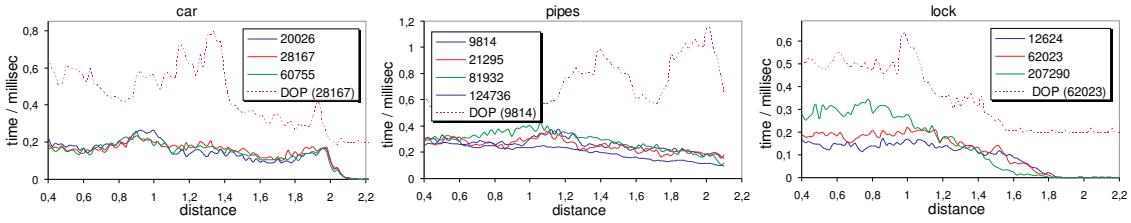

Fig. 12. Timings for different models and different polygon counts ($k_{min} = 10$ and $p_{min} = 0.99$). Also, a runtime comparison to a DOP tree is shown.

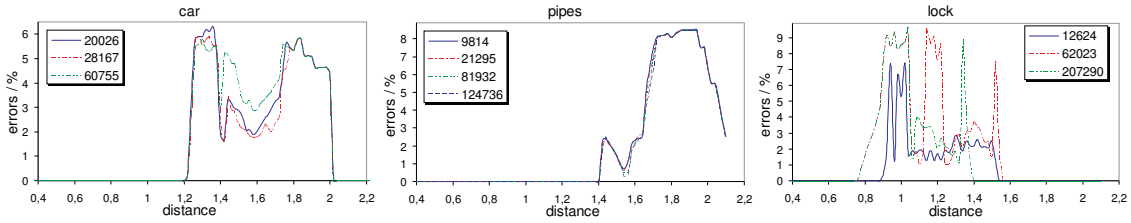

Fig. 13. Error rates corresponding to the timings in Figure 12.

that $x^2 + y^2 + z^2 = d^2$. Then, one has to show that $V \leq \max\{V_1, V_2, V_3\}$ always holds. In the second step, one has to prove that for a rotated BV the intersection volume is smaller or equal than when aligning that BV while keeping the same distance. If the difference of the size of the two bounding boxes is above a certain threshold, we use the intersection volume of the two bounding spheres as an estimate. In our experience this seems to work well.

6.6 Results

Each plot in Figure 12 shows the runtime for a model of varying complexity (the legend gives the number of polygons per object). In most cases, the runtime is fairly independent of the complexity.

Figure 13 shows the error rates corresponding to the timings in Figure 12. Here, the error is defined as the percentage of wrong detections. For measuring them, we have compared our results with an exact

approach. Only collision tests are considered where at least the outer BVs, which enclose the whole objects, intersect. Apparently, the error rates are always relatively low and mostly independent of the complexities: on average, only 1.89% (sharan), 1.54% (door lock), and 2.10% (pipes) wrong collisions are reported if the objects have a distance between 0.4 and 2.1, and about 3.19% (sharan), 1.71% (door lock), and 3.15% (pipes) wrong collisions are reported for distances between 1 and 2.

Further statistics and measurements can be found in [22, 21].

7 Collision Detection of Point Clouds

Point sets, on the one hand, have become a popular shape representation over the past few years. This is due to two factors: first, 3D scanning devices have become affordable and thus widely available [38]; second, points are an attractive primitive for rendering complex geometry for several reasons [37, 39, 50, 7].

Interactive 3D computer graphics, on the other hand, requires object representations that provide fast answers to geometric queries. Virtual reality applications and 3D games, in particular, often need very fast collision detection queries. This is a prerequisite in order to simulate physical behavior and in order to allow a user to interact with the virtual environment.

So far, however, little research has been presented to make point cloud representations suitable for *interactive* computer graphics. In particular, there is virtually no literature on determining collisions between two sets of points.

In this section, we present our algorithm to check whether or not there is a collision between two point clouds. The algorithm treats the point cloud as a representation of an implicit function that approximates the point cloud.

Note that we never explicitly reconstruct the surface. Thus, we avoid the additional storage overhead and an additional error that would be introduced by a polygonal reconstruction.

We also present a novel algorithm for constructing point hierarchies by repeatedly choosing a suitable subset. This incorporates a hierarchical sphere covering, the construction of which is motivated by a geometrical argument.

This hierarchy allows us to formulate two criteria that guide the traversal to those parts of the tree where a collision is more likely. That way, we obtain a *time-critical* collision detection algorithm that returns a "best effort" result should the time budget be exhausted. In addition, the point hierarchy makes it possible that the application can specify a maximum "collision detection resolution", instead of a time budget.

7.1 Overview of our Approach

A point cloud P_A can be viewed as a way to define a function $f_A(x)$ such that the implicit $f_A(x) = 0$ approximates P_A. Given two point clouds P_A and P_B, we pursue a hierarchical approach to quickly determine points x such that $f_A(x) = f_B(x) = 0$ by exploiting the spatial knowledge about the surface.

The idea of our algorithm is to create a hierarchy where the points are stored in its leaves. At each inner node, we store a sample of the point cloud underneath, a simple BV (such as a box), and a sphere covering for the part of the surface corresponding to the node (see Fig. 14). The point cloud samples effectively represent a simplified surface, while the sphere coverings define a neighborhood around it that contains the original surface.

The sphere coverings, on the one hand, can be used to quickly eliminate the possibility of an intersection of parts of the surface. The simplified point clouds, on the other hand, together with the sphere coverings, can be used to determine kind of a likelihood of an intersection between parts of the surface.

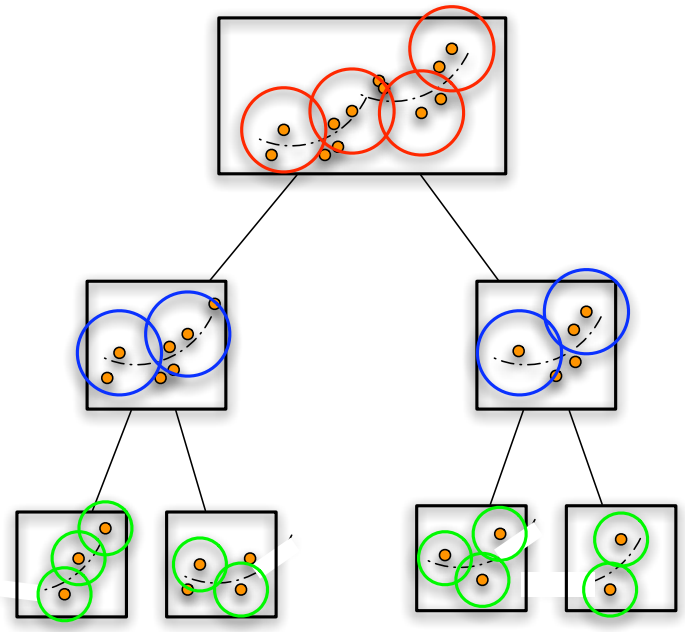

Fig. 14. Our approach constructs a point hierarchy, where each node stores a sample of the points underneath, which yields different levels of detail of the surface. In addition, we store a sphere covering of the surface of each node. Note that in our implementation we compose a sphere covering of many more spheres.

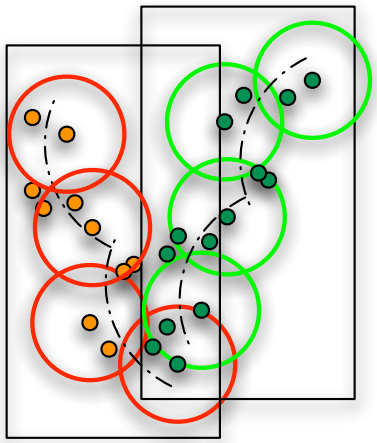

Fig. 15. Using the BVs and sphere coverings stored for each node, we can quickly exclude intersections of parts of the surfaces.

traverse(A, B)
if simple BVs of A and B do not overlap **then**
  return
end if
if sphere coverings do not overlap **then**
  return
end if
if A and B are leaves **then**
  return approx. distance between surfaces inside
end if
for all children A_i and B_j **do**
  compute priority of pair (A_i, B_j)
end for
traverse(A_i, B_j) with largest priority first

Fig. 16. Outline of our hierarchical algorithm for point cloud collision detection.

Given two such point cloud hierarchies, two objects can be tested for collision by simultaneous traversal (see Fig. 16), controlled by a priority queue. For each pair of nodes that still needs to be visited, our algorithm tries to estimate the likelihood of a collision, assigns a priority, and descends first into those pairs with largest priority. A pair of leaves is interrogated by a number of test points.

In order to make our point hierarchy memory efficient, we do not compute an optimal sphere covering, nor do we compute an optimal sample for each inner node. Instead, we combine both of them so that the sphere centers are also the sample.

7.2 Terminology

In the following, we treat the terms *bounding volume* (BV) and *node* of a hierarchy synonymous. A and B will always denote BVs of two different hierarchies.

For the sake of accuracy and conciseness, we introduce the following definitions.

Definition 6 (Cloud point) *Each point of a given point cloud is denoted as a cloud point. The set of cloud points lying in BV A or its r_ε-border (see previous section) is denoted as P_A.*

Definition 7 (Sample point) *Each inner node A of our hierarchy stores a sample of all cloud points lying in A or its r_ε-border. These sample points are denoted as P'_A ($P'_A \subset P_A$).*

Definition 8 (Test point) *A test point is an arbitrary point that is not necessarily contained in a given point cloud.*

7.3 Surface Definition

We define the surface of a point cloud implicitly based on weighted least squares. For sake of completeness, we will give a quick recap of that surface definition in this section; please refer to [25, 24, 23] for the details.

Let N points $p_i \in \mathbb{R}^3$ be given. Then, the implicit function $f : \mathbb{R}^3 \to \mathbb{R}$ describes the distance of a point x to a plane given by a point $a(x)$ and the normal $n(x)$:

$$f(x) = n(x) \cdot (a(x) - x) \tag{10}$$

The point $a(x)$ is the weighted average

$$a(x) = \frac{\sum_{i=1}^{N} \theta(\|x - p_i\|) p_i}{\sum_{i=1}^{N} \theta(\|x - p_i\|)} \tag{11}$$

and the normal $n(x)$ is defined by weighted least squares, i.e., $n(x)$ minimizes

$$\sum_{i=1}^{N} \big(n(x) \cdot (a(x) - p_i)\big)^2 \theta(\|x - p_i\|) \tag{12}$$

for fixed x and under the constraint $\|n(x)\| = 1$. This is exactly the smallest eigenvector of matrix B with

$$b_{ij} = \sum_{k=1}^{N} \theta(\|x - p_k\|)(p_{k_i} - a(x)_i)(p_{k_j} - a(x)_j) \tag{13}$$

In the following, we will use the kernel

$$\theta(d) = e^{-d^2/h^2} \tag{14}$$

where the global parameter h (called *bandwidth*) allows us to tune the decay of the influence of the points, which is theoretically unbounded. It should be chosen such that no holes appear, yet details are preserved.

In practice, we consider a point p_i only if $\theta(\|x - p_i\|) > \theta_\varepsilon$, which defines a *horizon of influence* for each p_i. However, now there are regions in $\mathbb{R}^3$ where only a small number of p_i are taken into account for computing $a(x)$ and $n(x)$. We amend this by dismissing points x for which the number c of p_i taken into account would be too small.Note that c and θ_ε are independent parameters. (We remark here that [1] proposed an amendment, too, although differently specified and differently motivated.)

Overall, the surface S is defined as the constrained zero-set of f, i.e.,

$$S = \left\{ x \mid f(x) = 0,\ \#\{p \in P : \|p - x\| < r_\varepsilon\} > c \right\} \tag{15}$$

where Equ. 14 implies $r_\varepsilon = h \cdot \sqrt{|\log \theta_\varepsilon|}$.

We approximate the distance of a point x to the surface S by $f(x)$. Because we limit the region of influence of points, we need to consider only the points inside a BV A plus the points within the r_ε-border around A, if $x \in A$.

7.4 Point Cloud Hierarchy

In this section, we will describe a method to construct a hierarchy of point sets, organized as a tree, and a hierarchical sphere covering of the surface.

In the first step, we construct a binary tree where each leaf node is associated with a subset of the point cloud. In order to do this efficiently, we recursively split the set of points by a top-down process. We create a leaf when the number of cloud points is below a threshold. We store a suitable BV with each node to be used during the collision detection process. Since we are striving for maximum collision detection performance, we should split the set so as to minimize the volume of the child BVs [48].

Note that so far, we have only partitioned the point set and assigned the subsets to leaves.

In the second step, we construct a simplified point cloud and a sphere covering for each level of our hierarchy. Actually, we will do this such that the set of sphere centers are exactly the simplified point cloud. One of the advantages is that we need virtually no extra memory to store the simplified point cloud.

In the following, we will derive the construction of a sphere covering for one node of the hierarchy, such that the centers of the spheres are chosen from the points assigned to the leaves underneath. In order to minimize memory usage, all spheres of that node will have the same radius. (This problem bears some relationship to the general mathematical problem of thinnest sphere coverings, see [9] for instance, but here we have different constraints and goals.)

More specifically, let A be the node for which the sphere covering is to be determined. Let $L_1, \ldots, L_n$ be the leaves underneath A. Denote by P_i all cloud points lying in L_i or its r_ε-border, and let $CH(P_i)$ be its convex hull. Let $P_A = \bigcup P_i$.

For the moment, assume that the surface in A does not have borders (such as intentional holes). Then

$$\forall x \in L_i : a(x) \in CH(P_i).$$

Therefore, if $x \in A$ and $f(x) = 0$, then x must be in $H = \bigcup_i CH(P_i)$.

So instead of trying to find a sphere covering for the surface contained in A directly, our goal is to find a set $K = \{K_i\}$ of spheres, centered at k_i, and a common radius r_A, such that $\mathrm{Vol}(K) = \mathrm{Vol}(\bigcup K_i)$ is minimal, with the constraints that $k_i \in P_A$, K covers H, and bounded size $|K| \leq c$. This problem can be solved by a fast randomized algorithm, which does not even need an explicit representation of the convex hulls (see below).

Our algorithm first tries to determine a "good" sample $P'_A \subset P_A$ as sphere centers k_i, and then computes an appropriate r_A. In both stages, the basic operation is the construction of a random point within the convex hull of a set of points, which is trivial.

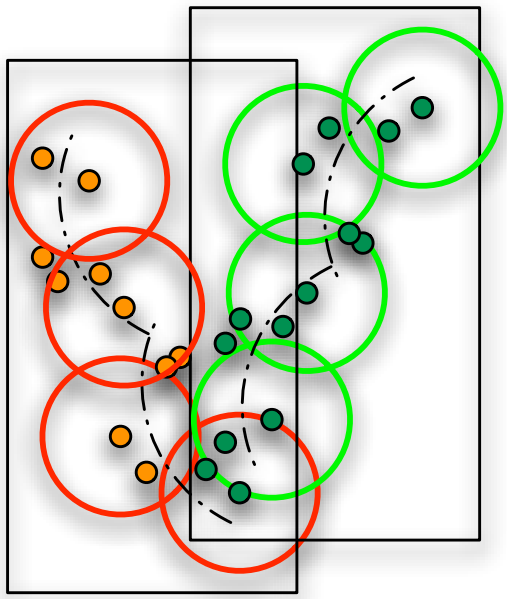

Fig. 17. Using the BVs and sphere coverings stored for each node, we can quickly exclude intersections of parts of the surfaces.

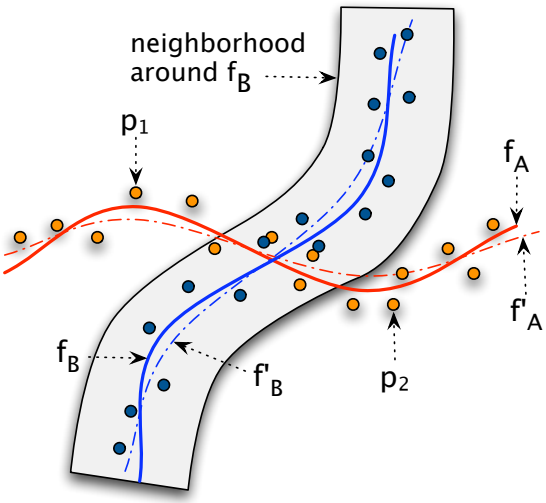

Fig. 18. Using the sample of two nodes and their r-neighborhoods, we can efficiently determine whether or not an intersection among the two nodes is likely.

The idea is to choose sample points $k_i \in P_A$ in the interior of H so that the distances between them are of the same order. Then, a sphere covering using the k_i should be fairly tight and thin.

We choose a random point q lying in BV A; then, we find the closest point $p \in P_A$ (this is equivalent to randomly choosing a Voronoi cell of P_A with probability depending on its size); finally, we add p to the set P'_A. We repeat this random process until P'_A contains the desired number of sample points. In order to obtain more evenly distributed k_i's, and thus a better P'_A, we can use quasi-random number sequences.

Since we want to prefer random points in the interior over points close to the border of H, we compute q as the weighted average of *all* points P_i of a randomly chosen L_i.

Conceptually, we could construct the Voronoi diagram of the k_i, intersect that with $H = \bigcup_i CH(P_i)$, determine the radius for the remainder of each Voronoi cell, and then take the maximum. Since the construction of the Voronoi diagram in 3D takes $O(n^2)$ (n = number of sites) [10], we propose a method similar to Monte-Carlo integration as follows.

Initialize r_A with 0. Generate randomly and independently test points $q \in H$. If $q \notin K$, then determine the minimal distance d of q to P'_A, and set $r_A = d$. Repeat this process until a sufficient number of test points has been found to be in K.

In other words, we continuously estimate

$$\frac{\text{Vol}(K \cap H)}{\text{Vol}(H)} \approx \frac{\#\text{ points} \in K \cap H}{\#\text{ points} \in H} \tag{16}$$

and increase r_A whenever we find that this fraction is less than 1. In order to improve this estimate, we can apply kind of a stratified sampling: when $q \notin K$ was found, we choose the next r test points in the neighborhood of q (for instance, by a uniform distribution confined to a box around q).

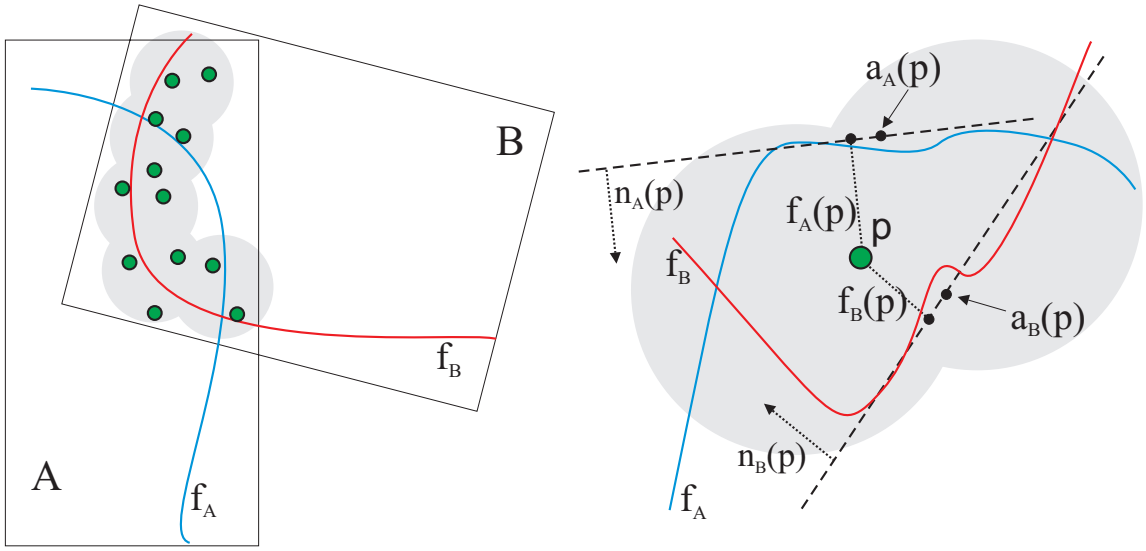

Fig. 19. In order to efficiently estimate the distance between the surfaces contained in a pair of leaves, we generate a number of random test points (left) and estimate their distance from A and B (right).

7.5 Simultaneous Traversal of Point Cloud Hierarchies

In this section we will explain the details of the algorithm that determines an intersection, given two point hierarchies as constructed above.

Utilizing the sphere coverings of each node, we can quickly eliminate the possibility of an intersection of parts of the surface (see Fig. 17). Note that we do not need to test all pairs of spheres. Instead, we use the BVs of each node to eliminate spheres that are outside the BV of the other node.

As mentioned above, we strive for a time-critical algorithm. Therefore, we need a way to estimate the likelihood of a collision between two inner nodes A and B, which can guide our algorithm shown in Fig. 16.

Assume for the moment that the sample points in A and B describe closed manifold surfaces $f_A = 0$ and $f_B = 0$, resp. Then, we could be certain that there is an intersection between A and B, if we would find two points on f_A that are on different sides of f_B.

Here, we can achieve only a heuristic. Assuming that the points P'_A are close to the surface, and that f'_B is close to f_B, we look for two points $p_1, p_2 \in P'_A$ such that $f'_B(p_1) < 0 < f'_B(p_2)$ (Fig. 18).

In order to improve this heuristic, we consider only test points $p \in P'_A$ that are outside the r_B-neighborhood around f_B, because this decreases the probability that the sign of $f_B(p_1)$ and $f_B(p_2)$ is equal.

Overall, we estimate the likelihood of an intersection proportional to the number of points on both sides.

This argument holds only, of course, if the normal $n_B(x)$ in Equation 10 does not "change sides" within a BV B.

When the traversal has reached two leaf nodes, A and B, we would like to find a test point p such that $f_A(p) = f_B(p) = 0$ (where f_A and f_B are defined over P_A and P_B, resp.).

In practice, such a point cannot be found in a reasonable amount of time, so we generate randomly and independently a constant number of test points p lying in the sphere covering of object A (see left of Fig. 19). Then we take

$$d_{AB} \approx \min_p \{|f_A(p)| + |f_B(p)|\} \tag{17}$$

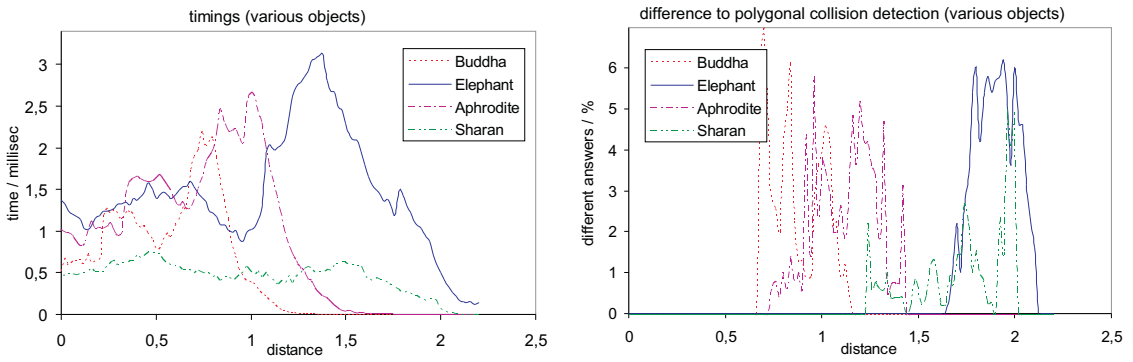

Fig. 20. Left: timings for different objects. Right: differences to polygonal collision detection of the objects; note that the polygonal models are *not* a tessellation of the true implicit surface, but just a tessellation of the point cloud. The results for the teddy are very similar to that of the sharan, and are therefore omitted.

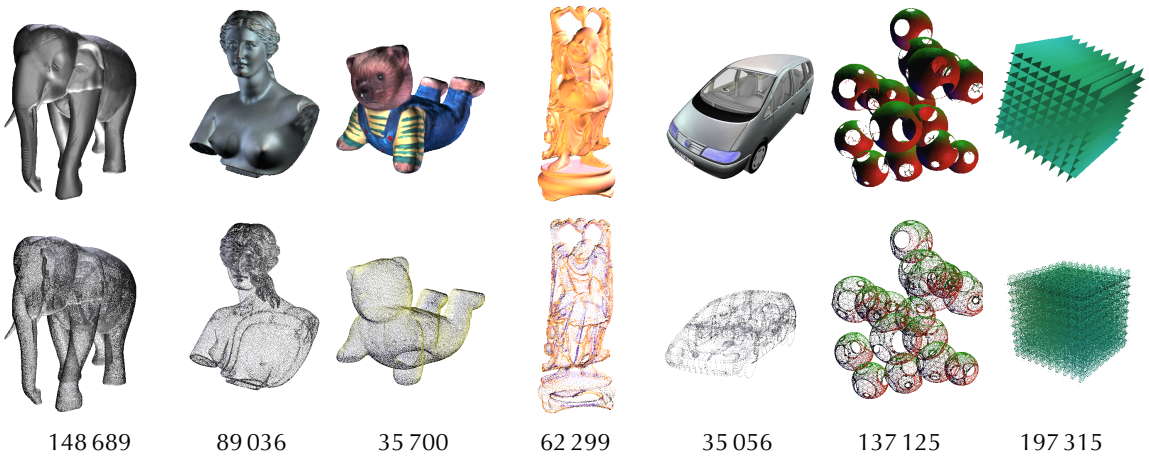

Fig. 21. Some of the models of our test suite, by courtesy of (left to right): Polygon Technology Ltd, Stanford, Volkswagen. The two artificial models (spheres and grid) show that our approach works well with non-closed geometry, too. The numbers are the sizes of the respective point clouds.

as an estimate of the distance of the two surfaces (see right of Fig. 19).

7.6 Results

We implemented our algorithm in C++. As of yet, the implementation is not fully optimized. In the following, all results have been obtained on a 2.8 GHz Pentium-IV with 1 GB main memory.

For timing the performance and measuring the quality of our algorithm, we have used a set of objects (see Figure 21), most of them with varying complexities (with respect to the number of points). Benchmarking is performed by the procedure proposed in [48], which computes average collision detection times for a range of distances between two identical objects.

Each plot in Figure 20 shows the average runtime for a model of our test suite, which is in the range 0.5–2.5 millisec. This makes our new algorithm suitable for real-time applications, and, in particular, physically-based simulation in interactive applications.

8 Conclusion

In this paper, we have reviewed some of our work in the area of collision detection.

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

We have presented a method for interference detection using programmable graphics hardware. Unlike previous GPU-based approaches, it performs all calculations in object-space rather than image-space, and it imposes no requirements on shape, topology, or connectivity of the polygonal input models.

We have also presented a general method to turn a conventional hierarchical collision detection algorithm into one that uses probability estimations to decrease the quality of collision detection in a controlled way. Thus, using this method, any hierarchical collision detection algorithm can be made time-critical, i.e., it computes the best answer possible within a given time budget.

And, finally, we have presented our approach to collision detection of point clouds, which is, to the best of our knowledge, still the only one. It works even for non-closed surfaces, and it works directly on the point cloud (and the implicit function defined by that), i.e., there is no polygonal reconstruction.

9 Acknowledgements

This work was partially supported by DFG grant ZA292/1-1.

In addition, I would like to thank Prof. Doru Talaba for inviting me to the 2-nd Advanced Study Institute on Product Engineering, 2007, held within the FP6 SSA project VEGA and the Network of Excellence INTUITION.

Finally, I would like to thank the development team of OpenSG.

References

1. A. Adamson and M. Alexa, *Approximating Bounded, Non-orientable Surfaces from Points*, in Shape Modeling International, 2004. accepted. 19
2. P. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar, and H. Haverkort, *Box-Trees and R-Trees with Near-Optimal Query Time*, Discrete and Computational Geometry, 28 (2002), pp. 291–312. 1
3. P. Agarwal, S. Krishnan, N. Mustafa, and S. Venkatasubramanian, *Streaming Geometric Optimization Using Graphics Hardware*, in 11th European Symposium on Algorithms, 2003. 2
4. P. 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. 2
5. G. Baciu and W. S.-K. Wong, *Hardware-assisted self-collision for deformable surfaces*, in Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST), 2002, pp. 129–136. ISBN 1-58113-530-0. `http://doi.acm.org/10.1145/585740.585762`. 2
6. ———, *Image-Based Techniques in a Hybrid Collision Detector*, IEEE Transactions on Visualization and Computer Graphics, 9 (2003), pp. 254–271. 2
7. K. Bala, B. Walter, and D. P. Greenberg, *Combining edges and points for interactive high-quality rendering*, in Proc. of SIGGRAPH, vol. 22, July 2003, pp. 631–640. ISSN 0730-0301. 17
8. R. Barzel, J. Hughes, and D. N. Wood, *Plausible Motion Simulation for Computer Graphics Animation*, in Proceedings of the Eurographics Workshop Computer Animation and Simulation, R. Boulic and G. Hégron, eds., 1996, pp. 183–197. 10
9. J. H. Conway and N. J. A. Sloane, *Sphere Packings, Lattices, and Groups*, Springer-Verlag, New York, 2 ed., 1993. 20

10. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, Berlin, Germany, 2nd ed., 2000. 21
11. S. A. Ehmann and M. C. Lin, *Accurate and Fast Proximity Queries Between Polyhedra Using Convex Surface Decomposition*, in Computer Graphics Forum, vol. 20, 2001, pp. 500–510. ISSN 1067-7055. 1
12. C. Ericson, *Real-Time Collision Detection*, Morgan Kaufman, 2004. ISBN 978-1-55860-732-3. 2
13. S. Fisher and M. Lin, *Fast Penetration Depth Estimation for Elastic Bodies Using Deformed Distance Fields*, in Proc. International Conf. on Intelligent Robots and Systems (IROS), 2001. 2
14. S. Gottschalk, M. Lin, and D. Manocha, *OBB-Tree: A Hierarchical Structure for Rapid Interference Detection*, in SIGGRAPH 96 Conference Proceedings, H. Rushmeier, ed., ACM SIGGRAPH, Aug. 1996, pp. 171–180. held in New Orleans, Louisiana, 04-09 August 1996. 1
15. N. Govindaraju, S. Redon, M. C. Lin, and D. Manocha, *CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments Using Graphics Hardware*, in Proc. of Graphics Hardware, San Diego, California, July 2003. `http://graphics.stanford.edu/papers/photongfx/`. 2
16. A. Gress and G. Zachmann, *Object-Space Interference Detection on Programmable Graphics Hardware*, in SIAM Conf. on Geometric Design and Computing, M. L. Lucian and M. Neamtu, eds., Seattle, Washington, Nov. 13–17 2003, pp. 311–328. ISBN 0-0-9728482-3-1. 6
17. P. M. Hubbard, *Approximating Polyhedra with Spheres for Time-Critical Collision Detection*, ACM Transactions on Graphics, 15 (1996), pp. 179–210. ISSN 0730-0301. 1, 13
18. S. Huh, D. N. Metaxas, and N. I. Badler, *Collision Resolutions in Cloth Simulation*, in IEEE Computer Animation Conf., Seoul, Korea, Nov.2001. 2
19. U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, *Programmable Stream Processors*, IEEE Comput., (2003), pp. 54–61. 6
20. Y. Kitamura, A. Smith, H. Takemura, and F. Kishino, *A Real-Time Algorithm for Accurate Collision Detection for Deformable Polyhedral Objects*, Presence, 7 (1998), pp. 36–52. 1
21. J. Klein and G. Zachmann, *ADB-Trees: Controlling the Error of Time-Critical Collision Detection*, in 8th International Fall Workshop Vision, Modeling, and Visualization (VMV), University München, Germany, Nov. 19–21 2003, pp. 37–45. ISBN 1-58603-393-X. `http://www.gabrielzachmann.org/`. 10, 13, 17
22. ———, *Time-Critical Collision Detection Using an Average-Case Approach*, in Proc. ACM Symp. on Virtual Reality Software and Technology (VRST), Osaka, Japan, Oct. 1–3 2003, pp. 22–31. `http://www.gabrielzachmann.org/`. 10, 13, 17
23. ———, *Nice and Fast Implicit Surfaces over Noisy Point Clouds*, in SIGGRAPH Proc., Sketches and Applications, Los Angeles, Aug. 2004. `http://www.gabrielzachmann.org/`. 19
24. ———, *Point Cloud Surfaces using Geometric Proximity Graphs*, Computers & Graphics, 28 (2004), pp. 839–850. ISSN 0097-8493. `http://dx.doi.org/10.1016/j.cag.2004.08.012`. 19
25. ———, *Proximity Graphs for Defining Surfaces over Point Clouds*, in Symposium on Point-Based Graphics, M. Alexa, M. Gross, H.-P. Pfister, and S. Rusinkiewicz, eds., ETHZ, Zürich, Switzerland, June 2–4 2004, pp. 131–138. `http://www.gabrielzachmann.org/`. 19
26. J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowrizal, and K. Zikan, *Efficient Collision Detection Using Bounding Volume Hierarchies of* k*-DOPs*, IEEE Transactions on Visualization and Computer Graphics, 4 (1998), pp. 21–36. 1
27. D. Knott and D. K. Pai, *CInDeR: Collision and Interference Detection in Real-Time Using Graphics Hardware*, in Proc. of Graphics Interface, Halifax, Nova Scotia,Canada, June11–13 2003. 2
28. T. Larsson and T. Akenine-Möller, *Collision Detection for Continuously Deforming Bodies*, in Eurographics, 2001, pp. 325–333. short presentation. 1
29. R. Lau, O. Chan, M. Luk, and F. Li, *A Collision Detection Method for Deformable Objects*, in Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST), Nov.11–13 2002, pp. 113–120. ISBN 1-58113-530-0. `http://doi.acm.org/10.1145/585740.585760`. 2
30. M. Lin, D. Manocha, J. Cohen, and S. Gottschalk, *Collision Detection: Algorithms and Applications*, in Proc. of Algorithms for Robotics Motion and Manipulation, J.-P. Laumond and M. Overmars, eds., 1996, pp. 129–142. 2
31. J.-C. Lombardo, M.-P. Cani, and F. Neyret, *Real-time collision detection for virtual surgery*, in Proc. of Computer Animation, Geneva, Switzerland, May26-28 1999. 2

32. W. A. McNeely, K. D. Puterbaugh, and J. J. Troy, *Six Degrees-of-Freedom Haptic Rendering Using Voxel Sampling*, Proceedings of SIGGRAPH 99, (August 1999), pp. 401–408. ISBN 0-20148-560-5. Held in Los Angeles, California. 1
33. K. Myszkowski, O. G. Okunev, and T. L. Kunii, *Fast collision detection between complex solids using rasterizing graphics hardware*, The Visual Computer, 11 (1995), pp. 497–512. ISSN 0178-2789. 2
34. B. Ooi, K. McDonell, and R. Sacks-Davis, *SpatiaJ kd-tree: An indexing mechanism for spatial databases*, in IEEE COMPSAC, 1987, pp. 433–438. 2
35. J. P., T. F., and T. C., *3D Collision Detection: a Survey*, Computers and Graphics, 25 (2001). 2
36. I. J. Palmer and R. L. Grimsdale, *Collision Detection for Animation using Sphere-Trees*, Proc. of EUROGRAPHICS, 14 (1995), pp. 105–116. ISSN 0167-7055. 1
37. H. Pfister, M. Zwicker, J. van Baar, and M. Gross, *Surfels: Surface Elements as Rendering Primitives*, in Proc. of SIGGRAPH, 2000, pp. 335–342. `http://visinfo.zib.de/EVlib/Show?EVL-2000-69`. 17
38. S. Rusinkiewicz, O. Hall-Holt, and M. Levoy, *Real-time 3D model acquisition*, ACM Transactions on Graphics, 21 (2002), pp. 438–446. ISSN 0730-0301. 17
39. S. Rusinkiewicz and M. Levoy, *QSplat: A Multiresolution Point Rendering System for Large Meshes*, in Proc. of SIGGRAPH, 2000, pp. 343–352. `http://visinfo.zib.de/EVlib/Show?EVL-2000-73`. 17
40. M. Shinya and M.-C. Forgue, *Interference detection through rasterization*, The Journal of Visualization and Computer Animation, 2 (1991), pp. 132–134. ISSN 1049-8907. 2
41. M. Teschner, B. Heidelberger, D. Manocha, N. Govindaraju, G. Zachmann, S. Kimmerle, J. Mezger, and A. Fuhrmann, *Collision Handling in Dynamic Simulation Environments*, in Eurographics Tutorial # 2, Dublin, Ireland, Aug. 29 2005, pp. 1–4. `http://www.gabrielzachmann.org/`. 2
42. M. Teschner, S. Kimmerle, G. Zachmann, B. Heidelberger, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnetat-Thalmann, and W. Strasser, *Collision Detection for Deformable Objects*, in Proc. Eurographics State-of-the-Art Report, Grenoble, France, 2004, pp. 119–139. `http://www.gabrielzachmann.org/`. 2
43. S. Uno and M. Slater, *The sensitivity of presence to collision response*, in Proc. of IEEE Virtual Reality Annual International Symposium (VRAIS), Albuquerque, New Mexico, Mar.01–05 1997, p. 95. 10
44. G. van den Bergen, *Efficient Collision Detection of Complex Deformable Models using AABB Trees*, Journal of Graphics Tools, 2 (1997), pp. 1–14. 1, 3
45. G. van den Bergen, *Collision Detection in Interactive 3D Environments*, Morgan Kaufman, 2003. ISBN 155860801X. 2
46. C. Wächter and A. Keller, *Instant Ray Tracing: The Bounding Interval Hierarchy*, in Eurographics Workshop/Symposium on Rendering, T. Akenine-Möller and W. Heidrich, eds., Nicosia, Cyprus, 2006, pp. 139–149. ISBN 3-905673-35-5. ISSN 1727-3463. `http://www.eg.org/EG/DL/WS/EGWR/EGSR06/139-149.pdf`. 2
47. G. Zachmann, *Rapid Collision Detection by Dynamically Aligned DOP-Trees*, in Proc. of IEEE Virtual Reality Annual International Symposium; VRAIS '98, Atlanta, Georgia, Mar. 1998, pp. 90–97. 1
48. ———, *Minimal Hierarchical Collision Detection*, in Proc. ACM Symposium on Virtual Reality Software and Technology (VRST), Hong Kong, China, Nov. 11–13 2002, pp. 121–128. `http://www.gabrielzachmann.org/`. 1, 2, 13, 20, 23
49. G. Zachmann and G. Knittel, *High-Performance Collision Detection Hardware*, Tech. Rep. CG-2003-3, University Bonn, Informatik II, Bonn, Germany, Aug. 2003. `http://www.gabrielzachmann.org/`. 10
50. M. Zwicker, H. Pfister, J. van Baar, and M. Gross, *EWA Splatting*, IEEE Transactions on Visualization and Computer Graphics, 8 (2002), pp. 223–238. ISSN 1077-2626. `http://csdl.computer.org/comp/trans/tg/2002/03/v0223abs.htm;http://csdl.computer.org/dl/trans/tg/2002/03/v0223.htm;http://csdl.computer.org/dl/trans/tg/2002/03/v0223.pdf`. 17