IFI TECHNICAL REPORTS

Institute of Computer Science,
Clausthal University of Technology

IfI-06-16

Clausthal-Zellerfeld 2006

Kinetic Bounding Volume Hierarchies for Collision Detection of Deformable Objects

Gabriel Zachmann* TU Clausthal

Rene Weller† TU Clausthal

Abstract

We present novel algorithms for updating bounding volume hierarchies of objects undergoing arbitrary deformations. Therefore, we introduce two new data structures, the kinetic AABB tree and the kinetic BoxTree.

The event-based approach of the kinetic data structures framework enables us to show that our algorithms are optimal in the number of updates. Moreover, we show a lower bound for the total number of BV updates, which is independent of the number of frames.

Furthermore, we present a kinetic data structures which uses the kinetic AABB tree for collision detection and show that this structure can be easily extended for continuous collision detection of deformable objects.

We performed a comparison of our kinetic approaches with the classical bottom-up update method. The results show that our algorithms perform up to ten times faster in practically relevant scenarios.

1 Introduction

Bounding volume hierarchies for geometric objects are widely employed in many areas of computer science to accelerate geometric queries. Such acceleration data structures are used in computer graphics for ray-tracing, occlusion culling and collision detection, to name but a few; They are also used in other areas such as geographical databases, molecular simulation, or robotics. Usually, a bounding volume hierarchy is constructed in a pre-processing step which is suitable as long as the objects are rigid.

However, deformable objects play an important role, e.g. for creating virtual environments in medical applications, entertainment, and virtual prototyping [Teschner et al., 2005]. If the object deforms, the pre-processed hierarchy becomes invalid.

In order to still use this well-known method for deforming objects as well, it is necessary to update the hierarchies after the deformation happens.

Most current techniques do not make use of the temporal and spatial coherence of simulations and just update the hierarchy by brute-force at every time step or they sim-

*e-mail: zach@in.tu-clausthal.de

†e-mail:weller@in.tu-clausthal.de

ply restrict the kind of deformation in some way, in order to avoid the time consuming per-frame update of all BVs.

On the one hand, we all know that motion in the physical world is normally continuous. So, if animation is discretized by very fine time intervals, a brute-force approach to the problem of updating BVHs would need to do this at each of these points in time. On the other hand, changes in the *combinatorial* structure of a BVH only occur at discrete points in time. Therefore, we propose to utilize an event-based approach to remedy this unnecessary frequency of BVH updates.

According to this observation, we present two algorithms to update hierarchies in a more sensitive way: we only make an update if it is necessary. In order to determine exactly when it is necessary, we use the framework of kinetic data structures (KDS). To use this kind of data structures, it is required that a *flightplan* is given for every vertex. This flightplan may change during the motion, maybe by user interaction or physical events (like collisions). Many deformations caused by simulations satisfy these constraints, like keyframe animations and many other animation schemes.

Beyond this, our algorithms can handle all objects for which we can build a bounding volume hierarchy, including polygon soups, point clouds, and NURBS models. Our algorithms are even flexible enough for handling insertions or deletions of vertices or edges in the mesh during run-time.

In the following, we first present a kinetization of an AABB tree and show that the associated update algorithm is optimal in the number of BV updates (This means that every AABB hierarchy which performs less updates must be invalid at some point of time).

Moreover, we prove an asymptotic lower bound on the total number of update operations in the worst case which holds for every BVH updating strategy. This number is independent from the length of the animation sequence under certain conditions.

In order to reduce the number of update operations, we propose a kinetization of the BoxTree. A BoxTree is a special case of an AABB, where we store only two splitting axis per node. On account of this, we can reduce the overall number of events.

Furthermore, we use the kinetic data structure framework not only to update the hierarchies, but also for the collision detection. Therefore we present the kinetic incremental collision detection. We use a separation list to keep track of the spatial and temporal coherence during collision detection.

Finally, we present the results of a comparison to the running times of hierarchical collision detection based on our novel kinetic BVHs and conventional bottom-up updating, resp.

2 Related Work

Many methods using bounding volume hierarchies have been developed for collision detection of rigid bodies and have been also adopted for deformable objects, including axis-aligned bounding volumes (AABBs) [van den Bergen, 1997, Provot, 1997], *k*-

Dops [Klosowski et al., 1998], OBBs [Gottschalk et al., 1996] and spheres [Palmer and Grimsdale, 1995]. Since the objects deform, the hierarchies must be updated regularly and the cost of these updates can be high. [van den Bergen, 1997] showed that updating is about ten times faster compared to a complete rebuild of an AABB hierarchy, and as long as the topology of the object is conserved, there is no significant performance loss in the collision check compared to rebuilding.

Several techniques to speed up the updates during each time step were proposed, including top-down, bottom-up updates and hybrid strategies [Bergen, 1998]. [Mezger et al., 2003] accelerated the update by omitting the update process for several time steps. Therefore, the BVs are inflated by a certain distance, and as long as the enclosed polygon does not move farther than this distance, the BV need not to be updated. There also exist some stochastic methods [Klein and Zachmann, 2003, Lin, 1993] for deformable collision detection, but they can not guarantee to find exact collisions and even a single missed collision can result in an invalid simulation.

[Knott and Pai, 2003] used hardware frame buffer operations to implement a ray-casting algorithm to detect static interferences between polyhedral objects. Therefore, the precision is constrained by the dimension of the viewport. Another hardware-based approach is given by [Heidelberger et al., 2004]. They use layered depth images with additional information on face orientation for the collision detection. Govindaraju et al [Govindaraju et al., 2005] use chromatic decompositions and the GPU to speed up the triangle tests using 2.5D overlap tests. However, for the broad phase, they use bottom-up updates of an AABB hierarchy. Furthermore, the algorithm is restricted to polygonal meshes with fixed connectivity.

[Lau et al., 2002] proposed a collision detection framework for deformable NURBS surfaces using AABB hierarchies. They reduce the number of updates by looking for special deformation regions.

Another approach for the special case of morphing objects [Larsson and Akenine-Moeller, 2003], where the objects are constructed by interpolating between some morphing targets, is to construct one BVH and fit this to the other morph targets, such that the corresponding nodes contain exactly the same vertices. During runtime, the current BVH can be constructed by interpolating the BVs. [Fisher and Lin, 2001] use deformed distance fields for the collision detection between deformable objects.

[James and Pai, 2004] introduced the BD tree which uses spheres as BVs and leads to a sub-linear-time algorithm for models which represent the deformation as linear superposition of precomputed displacement fields. However, the deformation is restricted to reduced deformable objects.

There also exist first approaches of collision detection using the event-based kinetic data structures (KDS): [Erickson et al., 1999] describes a KDS for collision detection between two convex polygons by using a so-called boomerang hierarchy. [Agarwal et al., 2002] and [Speckmann, 2001] developed a KDS using pseudo triangles for a decomposition of the common exterior of a set of simple polygons for collision

detection. However, all these approaches could not be extended to 3D-space or are much too expensive in practice.

3 Overview of our Approach

In this section we start with a quick recap of the kinetic data structure framework and its terminology.

The kinetic data structure framework (KDS) is a framework for designing and analyzing algorithms for objects (e.g. points, lines, polygons) in motion which was invented by [Basch et al., 1997]. The KDS framework leads to event-based algorithms that samples the state of different parts of the system only as often as necessary for a special task. This task can be for example the computing of the convex hull of a set of moving points and it is called the *attribute* of the KDS.

A KDS consists of a set of elementary conditions, called *certificates*, which prove altogether the correctness of the attribute. Those certificates can fail as a result of the motion of the objects. This certificate failures, the so-called *events*, are placed in an *event-queue*, ordered according to their earliest failure time. If the attribute changes at the time of an event, the event is called *external*, otherwise the event is called *internal*. Thus sampling of time is not fixed, but determined by the failure of some certain conditions.

As an example we can assume the bounding box of a set of moving points in the plane. The bounding box is the attribute we are interested in. It is build by four points $P^t_{\{max,min\},\{x,y\}}$ which have the maximum and minimum x- and y-values at a certain time point t. For every inner point P^t_i we have $P^t_i[x] < P^t_{max,x}[x]$, $P^t_i[y] < P^t_{max,x}[y]$, $P^t_i[x] > P^t_{min,x}[x]$ and $P^t_i[y] > P^t_{min,y}[y]$. These four simple inequations are the certificates in our example. If an inner point moves out if the bounding box due to its motion, e.g. $P^{t_2}_i[x] > P^{t_2}_{max,x}[x]$, this causes an external event at the point of time $t + \Delta t$ when $P^{t+\Delta t}_i[x] = P^{t+\Delta t}_{max,x}[x]$ (see Fig. 1). If $P^t_i[x] > P^t_j[x]$ and $P^{t_3}_i[x] < P^{t_3}_j[x]$ for points that are not in $P^t_{\{max,min\},\{x,y\}}$, this causes an internal event.

We measure the quality of a KDS by four criteria: A good KDS is *compact*, if it requires only little space, it is *responsive* if we can update it quickly in case of a certificate failure. It is called *local*, if one object is involved in not too many events. This guarantees that we can adjust changes in the flighplan of the objects quickly. And finally, a KDS is *efficient*, if the overhead of internal events with respect to external events is reasonable.

In our case, the objects are a set of m polygons with n vertices. Every vertex p_i has a flightplan $f_{p_i}(t)$. This might be a chain of line segments in case of a keyframe animation or algebraic motions in case of physically based simulations. The flightplan is assumed to use $O(1)$ space and the intersection between two flightplans can be computed in $O(1)$ time. The flightplan may change during simulation by user interaction or physical phenomena, including collisions. In this case, we have to update all events

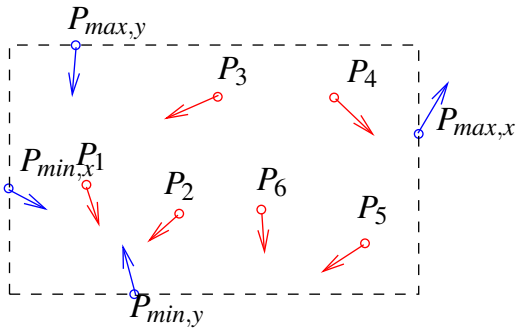

Figure 1: Assume a set of moving points in the plane. $P_{\{max,min\},\{x,y\}}$ for the current bounding volume of this points. At some time, P_5 will become smaller than $P_{min,y}$, this causes an event.

Algorithm 1: Simulation Loop

```
while simulation runs do
    calc time t of next rendering
    e ← min events in event-queue
    while e.timestamp < t do
        processEvent(e)
        e ← min events in event-queue
    check for collisions
    render scene
```

The attribute is, in case of the kinetic AABB tree and the kinetic BoxTree, a valid BVH for a set of moving polygons. An event will happen, when a vertex moves out of its BV.

The kinetic data structures we will present have some properties in common, which will be described as follows.

Algorithm 2: Check{*BV a of object A, BV b object B*}

```
if overlap ( a, b ) then
    if a and b are leaves then
        test_primitives( a, b )
    else
        forall children a[i] of a do
            forall children b[j] of b do
                Check( a[i], b[j] )
else
    return
```

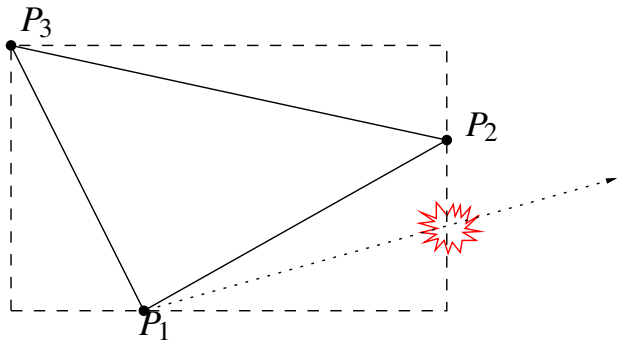

Figure 2: When P_1 becomes larger than the maximum vertex P_2, a Leaf-Event will happen.

They all use an event-queue for which we use an AVL-Tree, because with this data structure we can insert and delete events as well as extract the minimum in time $O(\log k)$ where k is the total number of events.

Both algorithms run within the same framework for kinetic updates, which is explained in Algorithm 1.

Furthermore, our algorithms use although the same procedure for the collision check (see Algorithm 2).

4 Kinetic AABB-Tree

In this section, we present a kinetization of the well known AABB tree. We build the tree by any algorithm which can be used for building static BVHs and store for every node of the tree the indices of these points that determine the bounding box. It is only required that the height of the BVH is logarithmic.

After building the hierarchy, we traverse the tree again to find the initial events.

There are three kinds of different events:

- *Leaf-Event*: Assume that P_1 realizes the BVs maximum along the x-axis. The a leaf event happens, when the x-coord of one of the other points P_2 or P_3 becomes larger than $P_{1,x}$ (see Fig. 2).
- *Tree-Event*: Let K be an inner BV with its children K_l and K_r and $P_2 \in K_r$ is the current maximum of K on the x-axis. A Tree-Event happens when the maximum of K_l becomes larger than P_2 (see Fig. 3). Analogously Tree-Events are generated for the other axis and the minima. other axis and the minima.
- *Flightplan-Update-Event*: Every time a flightplan of a point changes, we get a Flightplan-Update-Event.

So after the initialization we have stored six events with each BV. In addition, we put the events in the event queue, sorted by time-stamp.

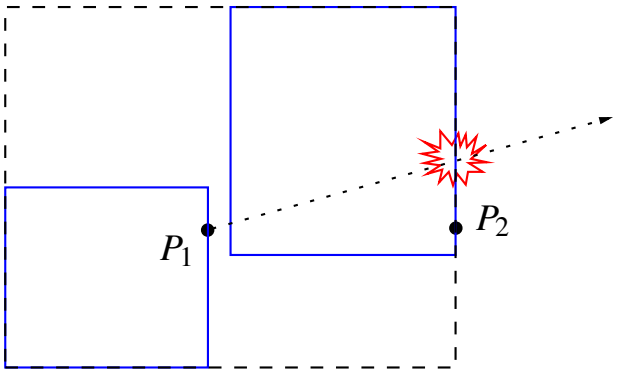

Figure 3: When P_1, the maximum of the left child-box becomes larger than the overall maximum vertex P_2, a Tree-Event will happen.

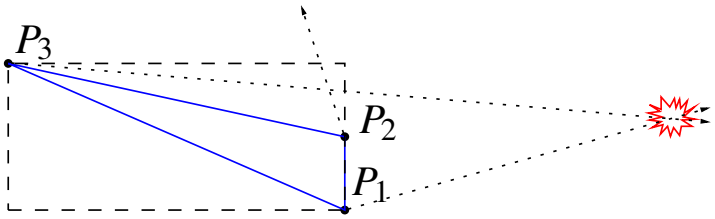

Figure 4: To keep the hierarchy valid when a Leaf-Event happens, we have to replace the old maximum P_2 by the new maximum P_1, and compute the time, when one of the other vertices of the polygon, P_2 or P_3 will become larger than P_1. In this example this will be P_3.

During runtime, we perform an update according to Algorithm 1 before each collision check. In order to keep the BV hierarchy valid, we have to handle the events as follows:

- *Leaf-Event*: Assume in a leaf BV B, realized by the vertices P_1, P_2 and P_3, the maximum extend along the x-axis has been realized by P_2. With the current event, P_1 takes over, and becomes larger than $P_{2,x}$. In order to maintain the validity of the BV hierarchy, in particular, we have to associate P_1 as the max x extent of B. In addition, we have to compute a new event, i.e., we have to compute all the intersections of the flightplans of all other vertices in B with P_1 in the xt-plane. The, an event is inserted into the queue for that pair with the earliest intersection time (Fig. 4).
  But that is not necessarily sufficient for keeping the BVH valid. In addition, we have to propagate this change in the BVH to the upper nodes. Assume B be the right son of its father V, so we have check whether P_2 had been the maximum of V too. In this case, we have to replace P_2 by the new maximum P_1. In addition, the corresponding event of V is not valid any more because it was computed with P_2. So we have to

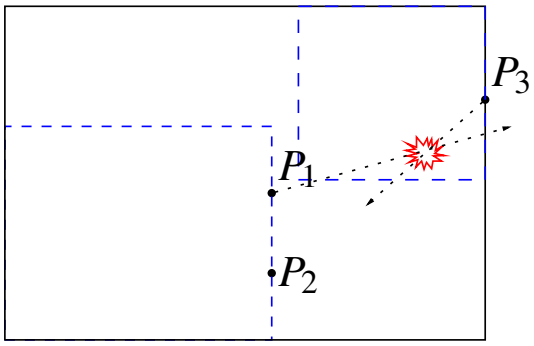

Figure 5: In Addition we have to propagate the change to upper Bev's in the hierarchy after a Tree-Event. After replacing the P_2 old maximum by the new maximum P_1 in the lower left box, we, we have to compute the event between P_1 and P_3, which is the maximum of the father BV.

delete this event from the event-queue and compute a new event between P_1 and the maximum of the left son of V.
Similarly we have to proceed up the BVH until we find the first predecessor $\overline{V}$ with $max_x\{\overline{V}\} \neq P_2$, or until we reach the root. In the first case we only have to compute another event between $max_x\{\overline{V}\}$ and P_1 and stop the recursion. (Fig. 5).

- *Tree-Event*: Let K be an inner node of the BVH and P_2 be the maximum along the x-axis. Assume further, P_2 is also the maximum of the left son. When a Tree-Event happens, P_2 will be replaced by P_1, which is the maximum of the right son of K (see Fig. 5). In addition, we have to compute a new event between P_1 and P_2 and propagate the change to the upper nodes in the BVH in the same way as described above.
- *Flightplan-Update-Event*: When the flightplan of a vertex changes, we have to update all events it is involved in. If a Flightplan-Update-Event happens, we traverse all of them and update the timestamps.

For measuring the theoretical performance of our algorithm we use the four criteria of quality given for every KDS.

In addition, we want to show that our data structure is a BVH, even if the object deforms. Therefore, we need the following definition.

Definition 1 *We call a kinetic AABB tree* valid*, if every node in the tree is a bounding volume for all polygons in its subtree.*

Theorem 1 *The kinetic AABB tree is compact, local, responsive and efficient. Furthermore, if we update the BVHs in the manner described above, then the tree is valid at every point of time.*

- *Compactness*: For a BVH we need $O(m)$ BVs. With every BV we store at most six Tree- or Leaf-Events. Therefore, we need space of $O(m)$ overall. Thus, our KDS is *compact*.

- *Responsiveness*: We have to show that we can handle certificate failures quickly.
  - Leaf-Events: In the case of a leaf event we have to compute new events for all points in the polygon, thus the responsiveness depends on the number of vertices per polygon. If this number is bounded we have costs of $O(1)$. When we propagate the change to upper nodes in the hierarchy, we have to delete an old event and compute a new one, which causes costs of $O(\log m)$ per inner node for the deletion and insertion in the event-queue, since the queue contains $O(m)$ events. In the worst case we have to propagate the changes until we reach the root. Thus the overall cost is $O(\log^2 m)$ for a leaf event.
  - Tree-Events: Analogously, for tree events, we get costs of $O(\log^2 m)$.

  Thus the KDS is also *responsive*.
- *Efficiency*: The efficiency measures the ratio of the *inner* to the *outer* events. Since we are interested in the validity of the whole hierarchy, every event is an inner event because every event changes our attribute. So, the *efficiency* is automatically given.
- *Locality*: The locality measures the number of events one vertex is participating in. For sake of simplicity, we assume that the degree of every vertex is bounded. Thus, every vertex can participate in $O(\log m)$ events. Therefore, a flightplan update can cause costs of $O(\log^2 m)$. Thus, the KDS is *local*.

We show the second part of the theorem by induction over time.

After the creation of the hierarchy, the BVH is apparently valid. The validity will be violated for the first time, when the combinatorial structure of the BVH changes, this means, a vertex flies out of its BV.

In case of a leaf, every vertex in the enclosed polygon could be considered to such an event. The initial computation of the Leaf-Events guarantees, that there exists an event for the earliest time point this can happen. For the inner nodes, it is sufficient to consider only the extremal vertices of the children: Assume a BV B with P_1 maximum of the left son B_{left} along the x-axis and P_2 maximum of the right son B_{right} along the x-axis. This means, all vertices in B_{left} have smaller x-coords than P_1 and all vertices in B_{right} have smaller x-coords than P_2. Thus, the maximum of B along the x-axis must be $max\{P_1, P_2\}$. Assume w.l.o.g. P_1 is the maximum. The vertex P_{next} which could become larger than P_1 could be either P_2 or a vertex of a BV in a lower level in the hierarchy becomes invalid before an event at B could happen. Assume P_{next} is in the right subtree, than P_{next} must become larger than P_2 and therefore B_{right} has become invalid sometime before. If P_{next} is in the left subtree, it must become larger than P_1 and thus B_{left} has become invalid before.

Summarised, we get a valid BVH after the initialisation, and the vertex which will violate the validity of the BVH for the first time triggers an event.

We still have to show, that the hierarchy stays valid after an event happens and that the next vertex which violates the validity also triggers an events.

- *Leaf-Event:* Assume B the affected leaf and P_2 becomes larger than P_1, which is the current maximum of B. As described above, we replace P_1 by P_2, therefore, B stays valid. Furthermore, we check for all other vertices in the polygon, which is the next to become larger than P_2 and store an event for that vertex, for which this happens

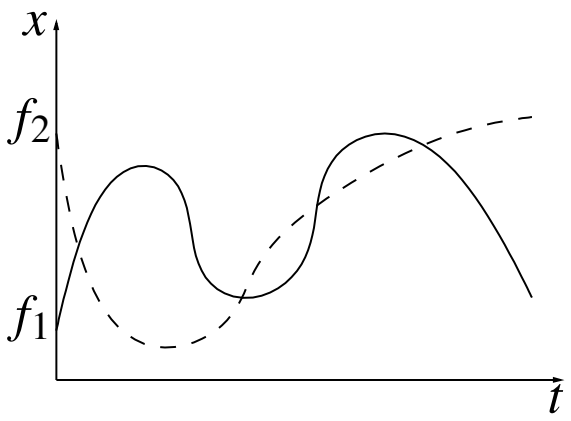

Figure 6: The flighplans are functions f_1 and f_2 in the xt-plane and similarly, in the yt- and zt-planes.

first. This guarantees, that we will find the next violation of the validity of this BV correctly.
In addition, all predecessors of B on the path up to the root which have P_1 as maximum become invalid too. Due to the algorithm described above, we replace all occurrences of P_1 on this path by P_2. Thus, BVH stays valid. The recomputing of the events on the path to the root ensures, that the vertex which will violate the validity provokes a suitable event.

- *Tree-Event:* Assume B the affected inner node. When an event happens, e.g. P_2 becomes larger than P_1 which is the current maximum of B, we once again replace P_1 by P_2, and therefore B stays valid. For the computation of the new event it is sufficient to consider only the two child BVs of B as described above. The propagation to the upper nodes happens analogously to the Tree-Event.
- *Flightplan-Update-Event:* A Flightplan-Update-Event does not change the combinatorial structure of the BVH, thus the BVH stays valid after such an event happens. But it is possible that the error times of some certificate failures change. To ensure that we find the next violation of the BVH, we have to recompute all affected events.

Recapitulating, we have show that we have a valid BVH after the initialisation and the first vertex that violates the validity provokes an event. If we update the hierarchy as described above, it stays valid after an event happens and we compute the next times when an event can happen correctly.

Note that by this theorem, the BVH is valid at *every* time point, not only at the moments we check for a collision as it is the case with most other update algorithms, like bottom-up or top-down approaches.

5 Optimality of the kinetic AABB-Tree

In the previous section we have proven that our kinetic AABB can be updated efficiently. Since there are no internal events, we would also like to determine the overall number of events for a whole animation sequence, in order to estimate the running time of the algorithm more precisely.

Theorem 2 *Given n vertices P_i, we assume that each pair of flightplans, $f_{P_i}(t)$ and $f_{P_j}(t)$, intersect at most s times. Then, the total number of events is in nearly $O(n \log n)$.*

We consider all flightplans along each coordinate axis separately (see Fig. 6). We reduce the estimation of the number of events on the computation of the upper envelope of a number of curves in the plane. This computation can be done by an algorithm using a combination of divide-and-conquer and sweep-line for the merge step. The sweep-line events are the sections of the sub-envelopes (we call them the edge-events) and intersections between the two sub-envelopes (which we call the intersection-events). Each sweep-line event corresponds to an update in our kinetic BVH.

The total number of sweep-line events depends obviously on s. We define $\lambda_s(n)$ as maximum number of edges of the upper envelope of n functions, whereas two of these functions intersects at most s times.

For the number of edge-events we get:

$$2\lambda_s\left(\left[\frac{n}{2}\right]\right) \leq \lambda_s(n),$$

since the two sub-envelopes are envelopes of $\left[\frac{n}{2}\right]$ flightplans.

Furthermore, we get an new edge in the envelope for every intersection-event, thus, this could be at most $\lambda_s(n)$. Therefore we can estimate the total number of events by the following recursion-equation:

$$T(2) = CT(n) \leq 2T\left(\frac{n}{2}\right) + C\lambda_s(n)$$

for some constant C. Overall we get:

$$T(n) \leq \sum_{i=0}^{\log n} 2^i C\lambda_s(\frac{n}{2^i})$$

In order to resolve the inequation, we have to know more about $\lambda_s(n)$. Since a total analysis of $\lambda_s(n)$ is to lengthy to be done here, we refer the interested reader to [Agarwal and Sharir., 1995].

One special characteristic of $\lambda_s(n)$ is:

Theorem 3 *For all $s, n \geq 1$ we have: $2\lambda_s(n) \leq \lambda_s(2n)$.*

With this theorem, we can solve the recursion equation and get:

Theorem 4 *For the computation of the upper envelope of n functions we need at most $O(\lambda_s(n) \log n)$ events.*

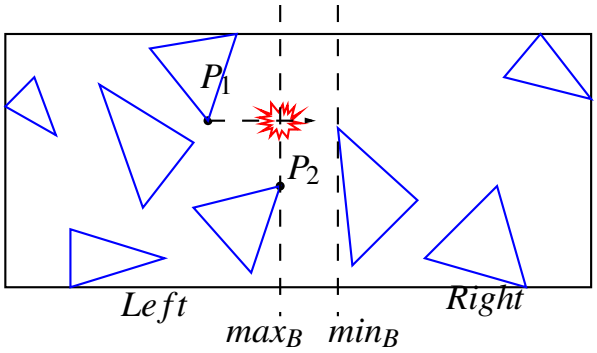

Figure 7: If a vertex in the left subtree becomes larger than the maximum P_2, a Tree-Event will happen.

Furthermore it is true, that $\lambda_s(n)$ behaves nearly linear; more precisely $\lambda_s(n) \in O(n \log^* n)$ where $\log^* n$ is the smallest number m for which the m-th iteration of the logarithm is smaller than 1. For example, $\log^* n \leq 5$ for all $n \leq 10^{20000}$.

Furthermore, it can be shown that the problem of computing the upper envelope is in $\Theta(n \log n)$, which shows that our algorithm is optimal in the worst case.

This demonstrates one of the strengths of the kinetic AABB tree: with classical update strategies like bottom-up, we need $O(kn)$ updates, where k is the number of frames. However, with our kinetic BVH, we can reduce this to nearly $O(n \log n)$ updates in the worst case. Furthermore, it is totally independent of the number of frames the animation sequence consists of (or, the frame rate), provided the number of intersections of the flightplans depends only on the length of the sequence in "wall clock time" and not on the number of frames.

Moreover, our kinetic AABB is updated only if the vertices that realize the BVs change; if all BVs in the BVH are still realized by the same vertices after a deformation step, nothing is done. As an extreme example, consider a translation of all vertices. A brute-force update would need to update all BVs — in our kinetic algorithm, nothing needs to be done, since no events occur. Conversely, the kinetic algorithm never performs more updates than the brute-force update, even if only a small number of vertices has moved.

6 Kinetic Boxtree

The kinetic AABB tree needs up to six events for every BV. In order to reduce the total number of events, we kinetized another kind of BVH, the BoxTree [Zachmann, 1995], which uses less memory than the kinetic AABB tree. The main idea of a BoxTree is to store only two splitting planes per node instead of six values for the extends of the box. To turn this into a KDS we proceed as follows:

In the pre-processing step, we build a BoxTree as proposed in [Zachmann, 1995], but similarly to the kinetization of the AABB tree, we do not store real values for the splitting planes. Instead, we store that vertex for each plane that realizes it (see Fig. 7). We continue with the initialization of the events:

There are only two kinds of events:

- *Tree-Event*: Assume B is an inner node of the hierarchy with splitting plane $e \in \{x, y, z\}$ and assume further min_B is the minimum of the right subtree (or max_B the maximum of the left subtree). A Tree-Event happens, when a vertex of the right subtree becomes smaller than min_B with regard to the splitting axis e, or a vertex of the left subtree becomes larger than max_B (see Fig. 7).
- *Flightplan-Update-Event*: Every time if the flightplan of a vertex changes, a Flightplan-Update-Event happens.

During runtime, we perform an update according to Algorithm 1 before every collision check. For keeping the BVH valid, we have to handle the events as described in the following:

Tree-Event: Let K be the node, where the Tree-Event happens and let P_{new} be the vertex in the left subtree of K that becomes larger than the current maximum K_{max}.

In this case we have to replace K_{max} by P_{new} and compute a new event for this node. Computing a new event is more complicated than in the case of a kinetic AABB tree. This is because the number of possibilities of different splitting planes and because of the fact that the extends of the BVs are given implicitly.

For simplicity, we first assume that all BVs have the same splitting axis. In this case, we have to look for event candidates, i.e. vertices, which can become larger than the maximum, in a depth-first search manner (see Fig. 8). Note that we do not have to look in the left subtree of the left subtree, because those vertices would generate an earlier event stored with one of the nodes in the subtree.

If more than one splitting axis is allowed, we first have to search for the nodes, with the same splitting axis (see Fig. 9).

Then we have to propagate the change to the upper nodes: First we have to search a node above K in the hierarchy with the same splitting axis. If its maximum is also K_{max}, we have to replace it and compute a new event for this node. We have to continue recursively until we reach a node O with the same splitting axis but $O_{max} \neq K_{max}$, or until we reach the root.

Flightplan-Update-Event: If the flightplan of a point changes, we have to update all events it is involved in. Therefore, we once again start at the leaves and propagate it to the upper nodes.

In order to show the performance of the algorithm, we have to show the four quality criteria for KDS again. Unfortunately, we will see that the kinetic BoxTree is responsive only in the one-dimensional case.

Theorem 5 *The kinetic BoxTree is compact, local and efficient. The responsiveness holds only in the one-dimensional case. Furthermore, if we use the strategies described above to update the BVH, we get a valid BVH at every point of time.*

We start with the proof of the first part of the theorem:

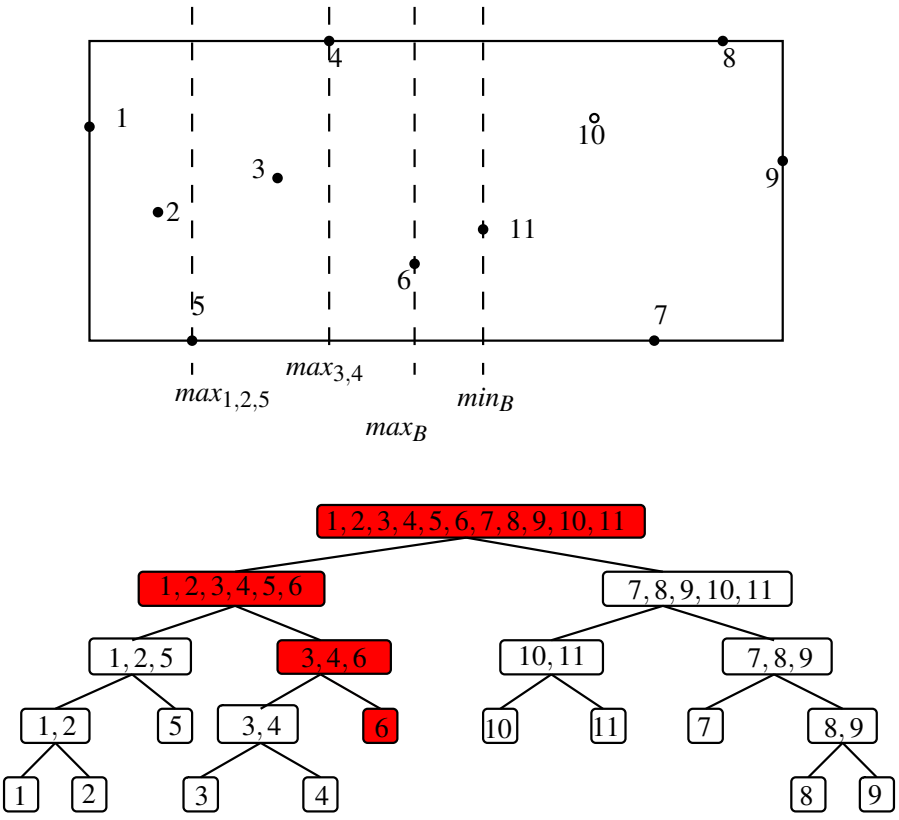

Figure 8: In order to compute a new event, we have to look which vertex can become larger than max_l. In the first level, this could be the maximum of the left subtree, the vertex 5, and all vertices in the right subtree of (1,2,3,4,5,6). On the next level it could be the maximum of the left subtree of (3,4,6), thus the vertex 4, and all vertices in the left subtree.

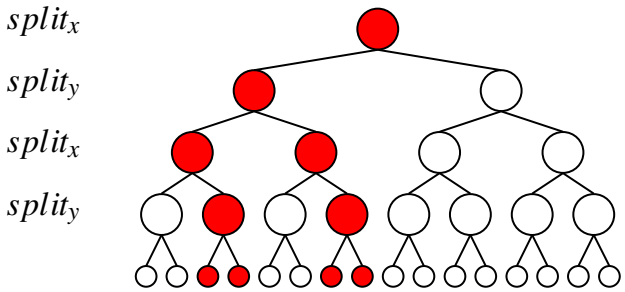

Figure 9: If more than one splitting axis are allowed, we have to search for the next level with the same splitting axis, when we want to look for the next candidates for an event. We have to visit the red marked nodes when we compute a new event for the root box.

- *Compactness*: We need space of $O(m)$ for storing the kinetic BoxTree. In addition, we get at most two events per node, so we have $O(m)$ events overall, so, the kinetic BoxTree is *compact*.
- *Efficiency*: Since we are interested in the validity of the whole hierarchy and every event leads to a real change of the combinatorial structure of the hierarchy, our KDS is also *efficient*.
- *Locality*: Assuming the tree is not degenerated, one polygon can be involved in at most $O(\log m)$ events, thus the KDS is local.
- *Responsiveness*: Not so clear is the responsiveness of our KDS, which is due to the costly computation of new events, where we have to go down the tree in dfs-manner. If all nodes have the same splitting axis, the computation of a new event costs at most $O(\log m)$, because of the length of a path from the root to a leaf in the worst case.
  But if the other nodes are allowed to use other split-axis too, it could be much more expensive. Assume that the root BV has the x-axis as split-axis and all other nodes have y as split-axis (Fig. 10). If an event appears at the root, we have to traverse the whole tree to compute the next event, so we have costs of $O(m \log m)$ and thus, the KDS is not responsive.
  Deletion and insertion of an event in the event-queue generate costs of $O(\log m)$ and in the worst case we have to propagate the change up to the root BV.
  Therefore, the overall cost for computing an event is $O(\log^2 m)$ in the 1D- and $O(m \log^2 m)$ in the multidimensional case, thus, the KDS is *responsive* only in 1D.

The total number of events is nearly in $O(n \log n)$ which follows analogously to the kinetic AABB tree.

We show the second part of the theorem by induction over time. W.l.o.g. we restrict to prove it only for the maxima, the arguments for the minima follows analogously. After building the BVH we have a valid hierarchy. It can become invalid if a vertex P gets larger along some axis than the maximun of some inner node K, i.e., if a Tree-

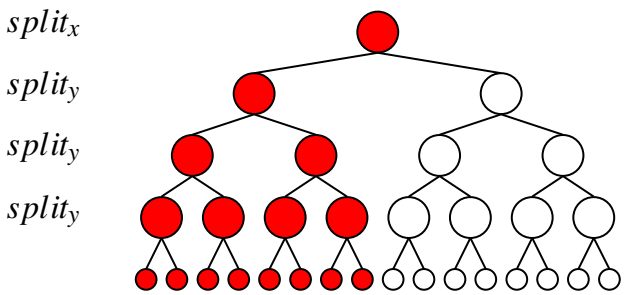

Figure 10: In the worst case, all levels have the same split axis, instead of the root. If we now want to compute a new event for the root, we have to traverse the whole tree.

Event happens. Since we calculate the Tree-Events for every inner node correctly, we will recognize the first time, when the hierarchy becomes invalid.

We still have to show, that the hierarchy stay valid after an event happen, and that we find the next event as well.

If a Tree-Events happens, this means some vertex P becomes larger than the maximum K_{max} of a node K, we have to replace all occurrences of K_{max} on the way from K to the root box by P and recalculate the events. This guarantees, that the hierarchy is still valid and we will find the next violation of the validity of the BVH correctly.

In the case of a Flightplan-Update-Event, the validity of the BVH does not change, but the error times of the events may change. Thus we have to recompute the times for all events, the vertex is involved in.

Summarized, the hierarchy is valid after initialisation and the first violation of the validity is stored as event in the BVH. After an event happens, the hierarchy is valid again an it is guaranteed, that we find the next violation of the validity. Thus, the BVH is valid at every point of time.

Recapitulating, we have a second KDS for fast updating a BVH which uses less events than the kinetic AABB tree but the computation of one single event is more complicated.

7 Kinetic Incremental Collision Detection

Our kinetic AABB tree and our kinetic BoxTree use the spatial and temporal coherence only for the updates of the hierarchies. Now we use the KDS framework also for the proper collision detection. Therefore we use an incremental technique to speed up the collision check which was first proposed in [Chen and Li, 1999] for collision detection of rigid bodies.

Our so-called *kinetic incremental collision detection* uses the kinetic AABB trees. Given two kinetic AABB-trees of two objects O_1 and O_2, we first traverse them for the initialization of the kinetic incremental collision detection as described in Algorithm 2

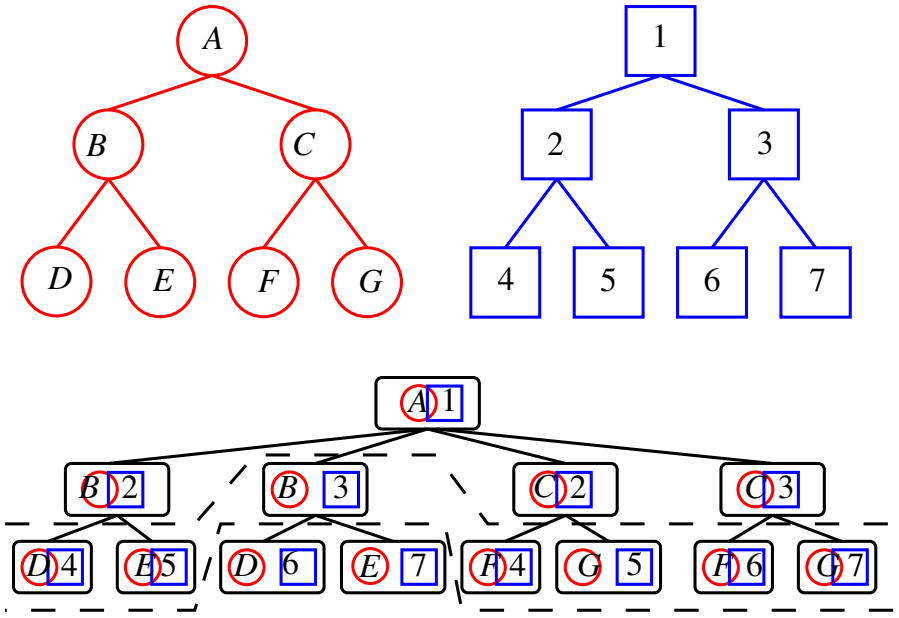

Figure 11: Given two BVHs, we get a BV-test-tree from the collision check. Those BV-pairs, where the traversal stops, build a list in this tree. We call it the separation list. This list may contain inner nodes, which do not overlap (B, 3), leave nodes which do not overlap (g, 5) and overlapping leaves (f, 6).

for only one time. Thereby we get a list, the so-called separation list, of colliding BVs in the BV-test-tree (See Fig. 11). This list contains pairs of BVs from O_1 and O_2 with nodes that do not overlap (we will call them the *inner nodes*), with leaves that do not overlap and finally, with leaves that do overlap.

During running time of the simulation, this configuration changes, and one of the following events may happen:

- *BV-Overlap-Event*: This event happens, if two BVs in the separation list which did not overlap before so far, now overlap. Thus, this event can happen only at inner nodes and not overlapping leaves (see Fig. 12).
- *Fathers-do-not-Overlap-Event*: This event happens, if the father of an inner node or a not overlapping leaf do not overlap anymore in the BV-test-tree (see Fig. 13).
- *Leaves-do-not-overlap-Event*: The Fathers-do-not-Overlap-Event cannot occur to overlapping leaves, because if their fathers do not overlap, even the leaves cannot overlap. Moreover, they must become not overlapping leaves before, and therefore, we introduce the Leaves-do-not-Overlap-Event.
- *BV-Change-Event*: Finally, we have an event which remarks changes of the BV-hierarchies. This event is comparable to the flightplan-updates of the kinetic AABB-

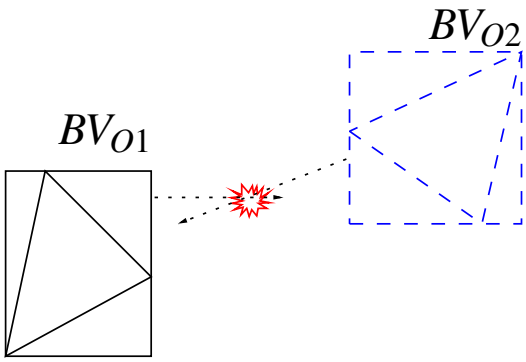

Figure 12: If the BVs move so that they overlap, we get an BV-Overlap-Event.

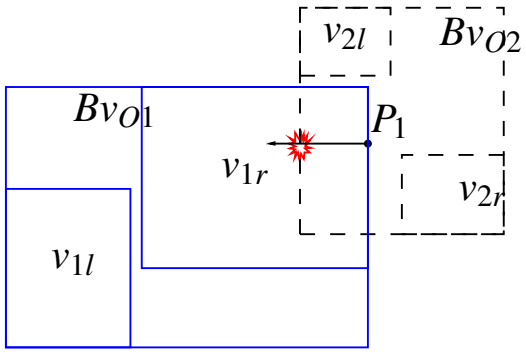

Figure 13: If the fathers Bv_{O1} and Bv_{O2} of the BVs v_{1r}, v_{1l}, v_{2r} and v_{2l} do not overlap anymore, we get a Fathers-do-not-Overlap-Event.

tree or BoxTree, but it is not exactly the same: This is, because an object in the separation list is composed of two BVs of different objects O_1 and O_2 and the flight-plans are attributes to the vertices of only one single object. Therefore, not every flightplan-update of an object has effect on the separation list (see Fig. 14).
In addition, a BV-Change-Event happens, if the combinatorial structure of a BV in the separation list changes. Since we use kinetic AABB trees as BVH for the objects, this can happen is a Tree-Event or a Leaf-Event in the BVH of an object happens. Surly, not all events cause changes at the separation list.

So, for example, if the BVs of the object do not overlap at the beginning of the simulation, the separation list only consists of one node which contains the root-BVs of the hierarchies.

During running time, we have to update the separation list every time an event happens according to the following rules:

- *BV-Overlap-Event*:

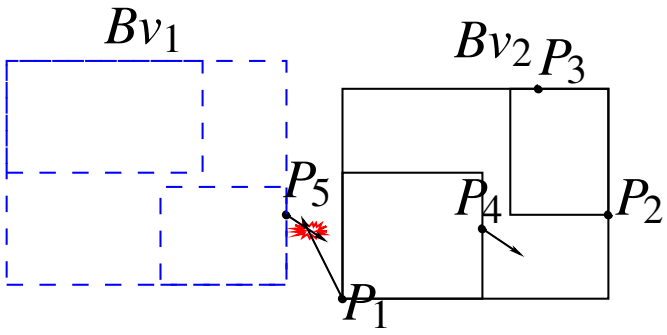

Figure 14: If the flightplan of P_4 changes, this has no effect on the separation list, and thus, no BV-Change-Event will happen due to this change.

  - node K is inner node: Let K be the inner node with BVs V_1 of object O_1 and V_2 of object O_2. In order to keep the separation list valid after the event happens, we have to delete K from it and insert the child-nodes into the BV-test-tree instead. This means, if V_1 has the children V_{1L} and V_{1R}, and V_2 has the children V_{2L} and V_{2R} we have to put 4 new nodes, namely $K_1 = (V_{1L}, V_{2L})$, $K_2 = (V_{1L}, V_{2R})$, $K_3 = (V_{1R}, V_{2L})$ and $K_4 = (V_{1R}, V_{2R})$ into the list. Then we have to compute the next time point t, when (V_1, V_2) do not overlap. Furthermore, we have to compute the times t_i for the new nodes, when they do overlap. If $t_i < t$ we put a BV-Overlap-Event in the queue, otherwise a Father-Do-Not-Overlap-Event (see Fig. 15).
  - if the node K is a not overlapping leaf: In this case we just have to turn the node into an overlapping leaf an compute the next Leaves-do-not-Overlap-Event.
- *Fathers-do-not-Overlap-Event*: In this case, we have to delete the corresponding node from the separation list, and insert his father from the BV-test-tree instead. Furthermore, we have to compute the new Fathers-do-not-Overlap-Event and BV-Overlap-Event for the new node and insert the event which will happen first into the event queue (See Fig. 16).
- *Leaves-do-not-overlap-Event*: If such an event happens, we have to turn the overlapping leaf into a non overlapping leaf, compute a new Fathers-do-not-Overlap-Event or rather a BV-Overlap-Event and put it into the event-queue.
- *BV-Change-Event*: If something in a BV in the separation list changes, e.g. the fligthplan of a vertex or the maximum or minimum vertex of a BV, we have to recompute all events, the BV is involved in.

Analogously to the theorems about the kinetic AABB tree and the kinetic BoxTree, we get a similar theorem for the kinetic incremental collision detection. First we have to define the "validity" of a separation list:

Definition 2 *We call a separation list "valid", if it contains exactly the not overlapping BVs of the lowest level in the BV-test-tree and the overlapping leaves.*

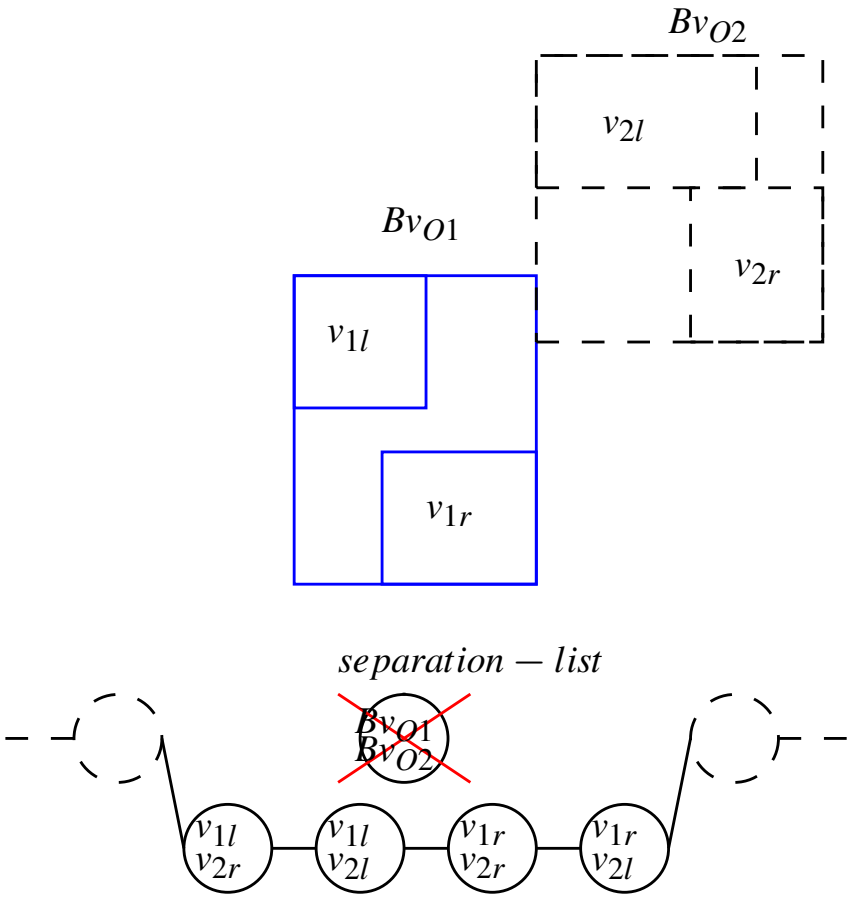

Figure 15: If the BVs Bv_{O1} and Bv_{O2} overlap due to an BV-overlap-Event, we have to remove them from the separation list and insert the pairs of their child-BVs v_{1r}, v_{1l}, v_{2r} and v_{2l}

Theorem 6 *The kinetic incremental collision detection is compact, local, responsive and efficient. Furthermore, we get a valid separation list at every point of time if we update the separation list as described above.*

In order to prove the first part of the theorem, we assume, w.l.o.g, that both objects O_1 and O_2 have the same number of vertices n and the same number of polygons m:

- *Compactness*: In order to evaluate the compactness, we have to define the attribute, we are interested in. In the case of the incremental collision detection, this is the separation list. In the worst case, every polygon of object O_1 collides with every polygon of O_2, thus, the size of a proof of correctness of the attribute, may have size $O(n^2)$.

For every node in the separation list, we store an event in the event queue, this will be at most $O(n^2)$ in total.
Furthermore, we have to store for every BV the nodes in the separation list in which it is participating, this could be at most $O(n^2)$ too. Summarized, the storage we need does not exceed the asymptotic size of the proof of correctness and thus, the data structure is *compact*.

- *Responsiveness*:
  - Leaves-do-not-overlap-Event: We have to delete the leaf from the overlapping-leaves-list and insert it into the separation list. This could be done in $O(\log(n^2))$ if we organize the lists as AVL-trees. The computation of a new event costs time $O(1)$, and the insertion in the event-queue of the new event could be done in $O(\log(n^2))$.
  - BV-Overlap-Event: The insertion of a new node into the separation list and deletion of the old node needs time $O(\log(n^2))$. In addition we have to delete the links from the old BV to the old node in the separation list and insert the new ones. If we organise this lists of links as AVL-tree, we get costs of $O(\log n)$.
  - Fathers-do-not-Overlap-Event: The deletion of nodes and events takes time of $O(\log(n^2))$ again. In addition, we have to look at the separation list, if we have already inserted the father, since Fathers-do-not-Overlap-Events always happen for four children at the same time and it could be possible, that we already processed the Fathers-do-not-Overlap-Event of another child pair. This takes time of $O(\log(n^2))$.

  Overall, our data structure is *responsive* in all cases.
- *Efficiency*: To determine the efficiency is more complicated, because it is not clear which events we have to treat as inner and outer events. Clearly, Leaves-Do-not-overlap-Events, BV-Overlap-Events and Fathers-do-not-Overlap-Event cause a real change of the attribute, the separation list, so this events will be outer events. But classifying the BV-Change-Events is more difficult. Those which happen due to flightplan updates clearly do not count, because they happen due to user interactions and could not be counted in advance. But there are also BV-Change-Events which happen due to changes of the BV-hierarchies, and they could be regarded as inner events.
  Since we use the kinetic AABB tree, there exist at most $O(n \log n)$ of these events. On the other hand, there may be $O(n^2)$ outer events and thus the KDS is still *responsive*, even if we treat the BV-Change-Events as inner events.
- *Locality*: We also have to pay attention when showing the locality of our data structure. The objects of the kinetic data structure, are the BVs, not the BV-pairs in the separation list. And one BV could participate $O(n)$ nodes in the separation list, so an update would have costs in the worst case of $O(n)$. This is compared to $O(n^2)$ total nodes in the separation list small and our data structure is also *local*.

In order to show the second part of the theorem, we use induction over time once again.

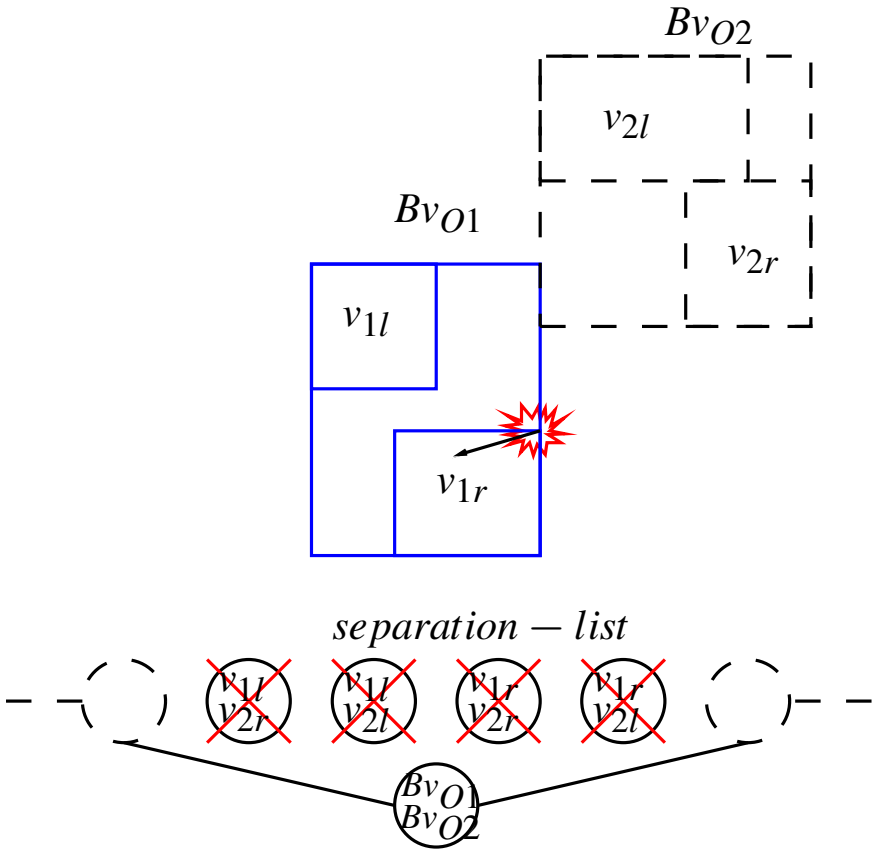

Figure 16: If an Fathers-do-not-Overlap-Event happens, this means Bv_{O1} and Bv_{O2} do not overlap anymore, we have to remove their child-BVs from the separation list and put the new node (Bv_{O1}, Bv_{O2}) into it.

After the first collision check, we get a valid separation list
The hierarchy becomes invalid, if either the BVs of an inner node or of a not overlapping leaf do not overlap anymore, or if the father of one of this kind of nodes do not overlap anymore.

Furthermore it could happen, that the BVs of an overlapping leaf do not overlap anymore. During initialisation, we compute this points of time as events and store them sorted by time in the event queue. Thus, we will notice the first point in time, when the hierarchy becomes invalid.

We have to show, that the separation list is updated correctly if an event happens and that the next point in time when it becomes invalid provokes an event.

- *BV-overlap-Event*: If a BV-overlap-Event happens, the separation list becomes invalid because the BVs of an inner node overlap. To repair the defect, we have to remove the node from the list and replace it by its four children. In order to determine the next time when one of this new nodes becomes invalid we have to calculate the events and insert them into the event queue..
- *Fathers-do-not-Overlap-Event*: In case of a Fathers-do-not-Overlap-Event the list becomes invalid because the BVs of a node K overlapped before and do not overlap anymore. Thus, K is not the deepest node in the hierarchy whose BVs do not overlap. So, K must be replaced by its parent node VK.
  The hierarchy can became invalid at node VK for the next time, if the BVs of VK overlap or the predecessor of VK does not overlap anymore. So, we have to compute what happens first and generate an event and insert it into the event queue. This guarantees that we will find the next time, when VK becomes invalid.
- *Leaves-do-not-Overlap-Event*: A Leaves-do-not-Overlap-Event does not affect the validity separation list. It is sufficient to turn the node into a not overlapping leaf.
  In order to recognize the next point of time when this node may violate the validity, we have to look if either a BV-Overlap-Event or a Fathers-do-not-Overlap-Event will happen first for this node and insert the corresponding event into the event queue.
- *BV-Change-Event*: A BV-Change-Event does not affect the validity of the separation list. But it is necessary to recompute the event times for the corresponding BVs in the list.

Overall, the validity of the hierarchy is guaranteed at all points of time.

If we want to check for a collision at any time, we only have to test the primitives in the overlapping leaves for collision.

Though our data structure fulfills all quality criteria of a kinetic data structure, the bounds of used storage $O(n^2)$ or update costs of $O(n)$ for flightplan updates of one single vertex do not look good. On the other hand these are worst-case-scenarios and only hold, if all polygons of one object overlap with all polygons of another object. This case does not happen in real-world applications. In most applications, the number of overlapping polygons could be shown to be nearly linear.

8 Calculation of the Events

The calculation of the events depends on the motion of the objects. We tested our algorithms with keyframe animations and simple velocity fields. The calculation of events will be described here for these scenarios.

8.1 Velocity Fields

In a velocity field, we simply assign a velocity vector to every vertex. During simulation, we just add the vector to the position of the vertex. Thus, the computation of the events is given by:

- *kinetic AABB tree and kinetic BoxTree:* We get an event if a vertex P become larger than another vertex Q along some axis. Therefore, the computation of an event corresponds to a line intersection in 2D.
  More precisely: Assume two vertices P and Q with velocity vectors p respectively q and a time point t is given with $P_x(t) < Q_x(t)$. In order to get the next point of time $\bar{t}$ when P becomes larger than Q along the x-axis, we first have to check if $qx \geq p_x$. In this case, there will never happen an event since P_x never will get larger than Q_x. Otherwise, we get $\bar{t} = \frac{Q_x(t)-P_x(t)}{p_x-q_x}$.
- *kinetic incremental Collision Detection*: We get events if two BV overlap or do not overlap anymore.
  Assume two BVs A and B with extreme points P^A_{imax} respective P^B_{imax} and minimum points P^A_{imin} respective P^B_{imax} with $i \in \{x,y,z\}$ at time point t.
  - Assume further A and B overlap at time t and we want to get the point of time $\bar{t}$ when they do not overlap anymore. Surly, A and B do not overlap $\Leftrightarrow$ it exists an axis $i \in \{x,y,z\}$ with $P^A_{imax}(\bar{t}) < P^B_{imin}(\bar{t})$ or $P^B_{imax}(\bar{t}) < P^A_{imin}(\bar{t})$.
    Thus, we have to compute the point of time $\bar{t_i}$ for every axis $i \in \{x,y,z\}$ when P^A_{imax} becomes smaller than P^B_{imin} and P^B_{imax} becomes smaller than P^A_{imin}. Then we initialise an event with the minimum if this $\bar{t_i}$.
  - If A and B do not overlap at time t, we have to look for the time point $\bar{t}$, when they overlap. We have, A and B overlap $\Leftrightarrow P^A_{imax}(\bar{t}) \geq P^B_{imin}(\bar{t})$ and $P^B_{imax}(\bar{t}) \geq P^A_{imin}(\bar{t})$ for all axis $i \in \{x,y,z\}$.
    Thus we have to compute the point of time $\bar{t_i}$ for all $i \in \{x,y,z\}$, when P^A_{imin} gets larger than P^B_{imax} and P^B_{imin} gets larger than P^A_{imax} too. We generate an event for the maximum of the $\bar{t_i}$.

8.2 Keyframe Animations

In key frame scenarios we get paths of line segments as motion of the vertices if we interpolate linearly between two keyframes. Since the motion is linear in sections, we can reuse the results from the section above.

Assume k keyframes $K_0, \ldots, K_k$ and l remarks the number of interpolated frames between two keyframes. We want to compute for the vertices P and Q with positions

$P(t)$ respectively $Q(t)$ at time t, when the next event between these points will happen, e.g. when P will become larger along the x-axis than Q.

Therefore we first have to detect the actual keyframe K_f with $l*f \leq t \leq l*(f+1)$. The we get the actual velocity p_f and q_f for the two points by $\overline{p_f} = P(l*(f+1)) - P(l*f)$ and $\overline{q_f} = Q(l*(f+1)) - Q(l*f)$.

Now we can compute $\bar{t}$ when P gets larger than Q, as described in the section before. If $\bar{t} \leq m*(f+1)$ we get the event for P and Q. But if $\bar{t} > l*(f+1)$ we have to look at the next keyframe if the paths of P and Q intersects, and so on (See Algorithm 3). Thus, we have to compute $O(l)$ line intersections for one single event in the worst case.

Algorithm 3: Calc Event{*start time t, vertices P, Q*}

Compute f with $l*f \leq t \leq l*(f+1)$;
$\bar{t} = l*(f+1)$;
while $t > l*f$ **do**
  $\overline{p} = P_{l*(f+1)} - P_{l*f}$;
  $\overline{q} = Q_{l*(f+1)} - Q_{l*f}$;
  $p_f = \frac{\overline{p}}{l}$;
  $q_f = \frac{\overline{q}}{l}$;
  Compute $\bar{t}$ when P gets larger than Q;
  $f = f+1$;

9 Results

We implemented our algorithms in C++ and tested the performance on a PC running Linux with a 3.0 GHz Pentium IV with 1 GB of main memory. We used two different types of test scenarios, keyframe animations and simple velocity fields.

There are three scenes with keyframe animations, the first one shows a tablecloth falling on a table. We tested this scene with several resolutions of the cloth, ranging from 2k to 16k faces. This scene show the behaviour of our algorithms under heavy deformation. The two other keyframe scenarios show typical cloth animations. The first one shows a male avatar with a shirt in resolutions from 32k to 326k deforming triangles (Fig. 17), the other one a female avatar with a dress reaching from 65k to 580k deforming triangles (see Fig. 14).

For measuring the speed of updates when the flightplan changes, we used a benchmark with two spheres. Every point of a sphere is given a velocity vector which points away from the midpoint, so that the spheres expand regularly. When they collide, the velocity vectors of the colliding triangles are reversed. We tested this scene with resolutions from 2k to 40k triangles.

We compared the performance of our algorithms with a bottom-up updating strategy.

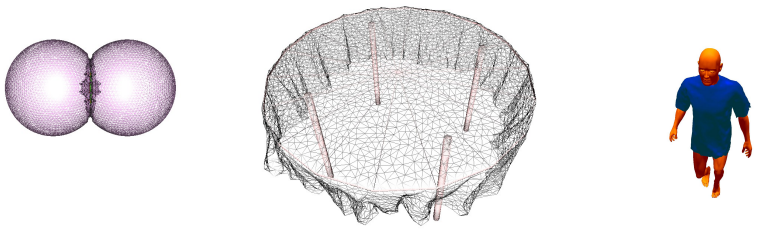

Figure 17: The scenes, with which we tested our algorithm: Two expanding spheres, a tablecloth falling down, and a cloth animation scene.

First, we consider the number of events. In the high-resolution tablecloth scene, we have about 400 events per frame and have to update only 1000 values for the kinetic AABB tree, and even less for the kinetic BoxTree (Fig. 18). In contrast, the bottom-up approach has to update 60 000 values. Since the computation costs for an event are relatively high, this results in an overall speed-up of about factor 5 for updating the kinetic AABB tree. The number of events rises nearly linearly with the number of polygons, which supports our lower bound for the total number of events of nearly $O(n \log n)$ (see Fig. 18).

The figure also shows that we need less events for the kinetic BoxTrees, but the the proper collision check takes more time since the kinetic BoxTree is susceptible for deformations.

A high amount of flightplan updates does not affect performance of the data structures, they are still up to 5 times faster than the bottom-up updates (see Fig. 21).

In the cloth animation scenes, the gain of the kinetic data structures is highest, because the the objects undergo less deformation than the tablecloth, and thus we have to perform less events. In this scenarios we see a performance gain of a factor about 10 (Fig. 21). From Theorem 2, it is clear that this factor increases with the number of interpolated frames between two keyframes. This is, because the performance of the event based kinetic data structures only depends on the number of keyframes and not on the total length of the scene.

Overall, the kinetic AABB performs best, and the running time of the updating operations is independent from the sampling frequency. This means, for example, if we want to render a scene in slow motion, maybe ten times slower, the costs for updating are still the same, while they increase for the bottom-up-update by a factor of ten.

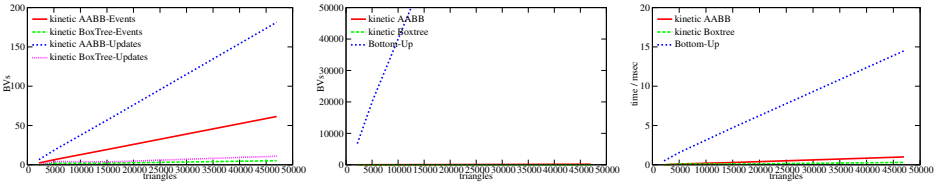

Figure 18: The left diagram shows the average number of events and updates per frame. The kinetic BoxTree has, as expected, the smallest total number of events and the smallest number of total updates per event. The diagram in the middle shows that the total number of updates is significantly lower than the updates needed by the bottom-up-strategy. Unfortunately, due to the relatively high deformation of the tablecloth and the high costs for the event-computation, the gain is lesser than expected, but there is still a significant gain for the kinetic AABB tree and the BoxTree (right diagram).

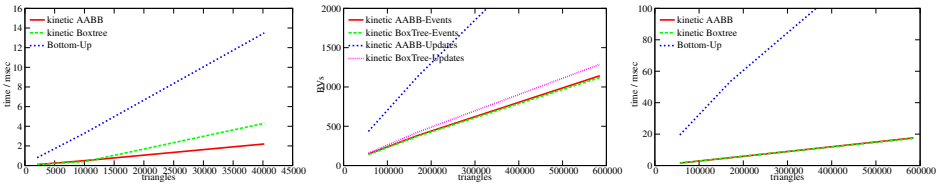

Figure 19: The left diagram shows the average total update time for the sphere scenes. This scene seems to be more appropriate for the KDSs than the tablecloth scene, despite the high amount of flightplan updates. The gain of the kinetic data structures compared to the bottom-up approach is more than a factor of five. The diagram in the middle shows the average number of events and updates for the cloth animation with the women. The ratio seems to be nearly the same as in the tablecloth scene. The diagram on the right shows the update times in the animation scene with the women. In this scene we have an overall gain of a factor about 10 for the kinetic AABB compared to the bottom-up-update.

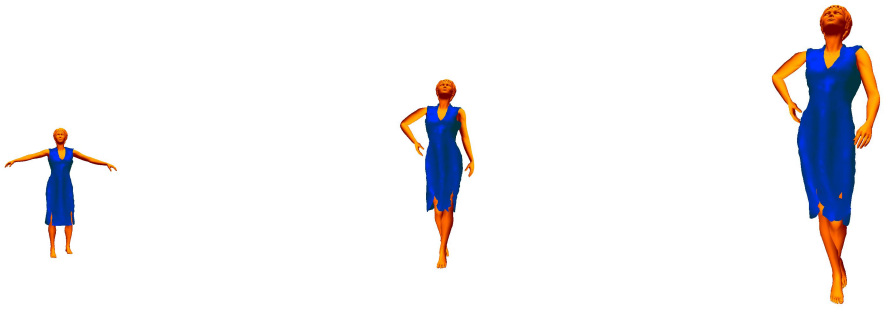

Figure 20: The second cloth animation scene shows a walking female avatar with a dress

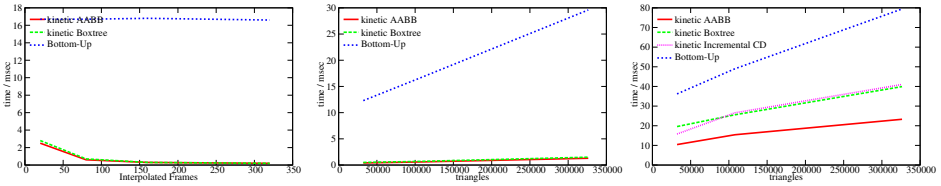

Figure 21: The left diagram shows the average update time for the cloth animation scene with the man, depending on the number of interpolated frames between two key frames. Since the number of events only depends on the number of key frames and not on the number of interpolated frames, so, the average update time decreases if we increase the total number of frames. The diagram in the middle shows the average update time for the cloth animation scene with the man and shows total time for this scene. In this scene we have an overall gain of a factor about 10 for the kinetic AABB compared to the bottom-up-update. The right diagram shows the total time, this means the time for updates and the proper check time.

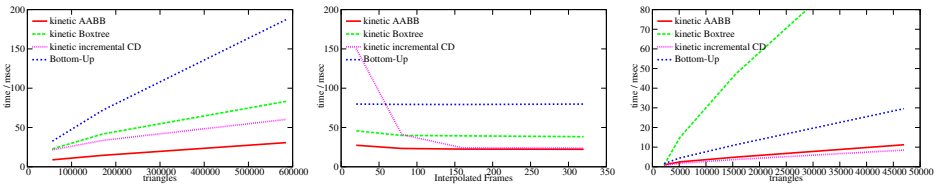

Figure 22: The left diagram shows the average total time for the updates and the collision checks cloth animation scene with the female avatar. The gain of the kinetic incremental collision detection compared to the bottom-up approach is about a factor of 5. Due to the high ammount of BVs in the separation list, it is slower than the kinetic AABB tree. The diagram in the middle shows the average total time for the same scene depending on the number of interpolated frames between two keyframes. The kinetic incremental collision detection benefits of the same phenomena as our other kinetic data structures. The diagram on the right shows the total times in tablecloth scene. In this scene we have an overall gain of a factor about 3 for the kinetic incremental collision detection compared to the bottom-up-update.

10 Conclusions and Future Work

We introduced two novel data structures and algorithms for updating a BVH over deformable objects fast and efficient and one data structure especially for collision detection between deformable objects.

We presented a theoretic and experimental analysis showing that our new algorithms are fast and efficient both theoretically and in practice. We used the kinetic data structure framework to analyze our algorithms, and we have shown an upper bound of nearly $O(n \log n)$ for the updates that are required at most to keep a BVH valid. We also showed that the kinetic AABB tree and kinetic BoxTree are optimal in the sense that they only need to make $O(n \log n)$ updates.

Our kinetic data structures update bounding volumes or find collisions more than 10 times faster than a bottom-up approach in practically relevant cloth animation scenes. Even in scenarios with heavy deformations of the objects or many flightplan updates we have a significant gain by our algorithms.

We believe that the kinetic data structures are a fruitful starting point for future work on collision detection for deformable objects. We will try to improve the performance by using trees of higher order than binary trees.

The BoxTree uses less events, but is susceptible to deformations. So it could be a good strategy to rebuild parts of a BoxTree if the deformation is too strong. This could also speed up the algorithm.

It is very easy to expand our kinetic incremental collision detection for continuous collision detection, since we have a valid separation list at every time point. We just have to add an algorithm for the triangles.

Finally we plan to use our algorithms in other kinds of motion, including physically based simulations and other animation schemes and other applications like ray-tracing or occlusion culling.

References

[Agarwal et al., 2002] Agarwal, P. K., Basch, J., Guibas, L. J., Hershberger, J., and Zhang, L. (2002). Deformable Free-Space Tilings for Kinetic Collision Detection. *I. J. Robotic Res.*, 21(3):179 – 198.

[Agarwal and Sharir., 1995] Agarwal, P. K. and Sharir., M. (1995). Davenport–schinzel sequences and their geometric applications. Technical Report Technical report DUKE–TR–1995–21.

[Basch et al., 1997] Basch, Guibas, and Hershberger (1997). Data Structures for Mobile Data. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*.

[Bergen, 1998] Bergen, G. V. D. (1998). Efficient Collision Detection of Complex Deformable Models using AABB Trees.

[Chen and Li, 1999] Chen, J.-S. and Li, T.-Y. (1999). Incremental 3D Collision Detection with Hierarchical Data Structures.

[Erickson et al., 1999] Erickson, J., Guibas, L. J., Stolfi, J., and Zhang, L. (1999). Separation-sensitive collision detection for convex objects. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 327 – 336, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[Fisher and Lin, 2001] Fisher, S. and Lin, M. C. (2001). Deformed distance fields for simulation of non-penetrating flexible bodies. In *Proceedings of the Eurographic workshop on Computer animation and simulation*, pages 99–111, New York, NY, USA. Springer-Verlag New York, Inc.

[Gottschalk et al., 1996] Gottschalk, S., Lin, M. C., and Manocha, D. (1996). OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 30(Annual Conference Series):171–180.

[Govindaraju et al., 2005] Govindaraju, N. K., Knott, D., Jain, N., Kabul, I., Tamstorf, R., Gayle, R., Lin, M. C., and Manocha, D. (2005). Interactive collision detection between deformable models using chromatic decomposition. *ACM Trans. Graph.*, 24(3):991–999.

[Heidelberger et al., 2004] Heidelberger, B., Teschner, M., and Gross, M. H. (2004). Detection of collisions and self-collisions using image-space techniques. In *WSCG*, pages 145–152.

[James and Pai, 2004] James, D. and Pai, D. (2004). Bd-tree: Output-sensitive collision detection for reduced deformable models. *ACM Transactions on Graphics (SIGGRAPH 2004)*, 23(3).

[Klein and Zachmann, 2003] Klein, J. and Zachmann, G. (2003). Adb-trees: Controlling the error of time-critical collision detection. In Ertl, T., Girod, B., Greiner, G., Niemann, H., Seidel, H.-P., Steinbach, E., and Westermann, R., editors, *Vision, Modeling and Visualisation 2003*, pages 37–46. Akademische Verlagsgesellschaft Aka GmbH, Berlin.

[Klosowski et al., 1998] Klosowski, J. T., Held, M., Mitchell, J. S. B., Sowizral, H., and Zikan, K. (1998). Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36.

[Knott and Pai, 2003] Knott, D. and Pai, D. (2003). Cinder: Collision and interference detection in real–time using graphics hardware.

[Larsson and Akenine-Moeller, 2003] Larsson, T. and Akenine-Moeller, T. (2003). Efficient collision detection for models deformed by morphing. *The Visual Computer*, 19(2-3):164–174.

[Lau et al., 2002] Lau, R. W., Chan, O., Luk, M., and Li, F. W. (2002). Large a collision detection framework for deformable objects. In *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 113–120, New York, NY, USA. ACM Press.

[Lin, 1993] Lin, M. C. (1993). *Efficient collision detection for animation and robotics*. PhD thesis. Chair-John F. Canny.

[Mezger et al., 2003] Mezger, J., Kimmerle, S., and Etzmuß, O. (2003). Hierarchical Techniques in Collision Detection for Cloth Animation. *Journal of WSCG*, 11(2):322–329.

[Palmer and Grimsdale, 1995] Palmer, I. J. and Grimsdale, R. L. (1995). Collision detection for animation using sphere-trees. *j-CGF*, 14(2):105–116.

[Provot, 1997] Provot, X. (1997). Collision and self-collision handling in cloth model dedicated to design garments. pages 177 – 189.

[Speckmann, 2001] Speckmann, B. (2001). *Kinetic Data Structures for Collision Detection*. PhD thesis.

[Teschner et al., 2005] Teschner, M., Kimmerle, S., Heidelberger, B., Zachmann, G., Raghupathi, L., Fuhrmann, A., Cani, M.-P., Faure, F., Magnenat-Thalmann, N., Strasser, W., and Volino, P. (2005). Collision detection for deformable objects. *Computer Graphics forum*, 24(1):61–81.

[van den Bergen, 1997] van den Bergen, G. (1997). Efficient collision detection of complex deformable models using aabb trees. *J. Graph. Tools*, 2(4):1–13.

[Zachmann, 1995] Zachmann, G. (1995). The boxtree: Exact and fast collision detection of arbitrary polyhedra. In *SIVE Workshop*, pages 104–112.

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Wojciech Jamroga
Contact: wjamroga@in.tu-clausthal.de
URL: http://www.in.tu-clausthal.de/~wjamroga/techreports/
ISSN: 1860-8477

The IfI Review Board