Eurographics Symposium on Parallel Graphics and Visualization (2016)
W. Bethel, E. Gobbetti (Editors)

Adaptive Collision Culling for Large-Scale Simulations by a Parallel Sweep and Prune Algorithm

G. Capannini and T. Larsson

Mälardalen University, Västerås, Sweden

Abstract

We propose a parallel Sweep and Prune algorithm that solves the dynamic box intersection problem in three dimensions. It scales up to very large datasets, which makes it suitable for broad phase collision detection in complex moving body simulations. Our algorithm gracefully handles high-density scenarios, including challenging clustering behavior, by using a dual-axis sweeping approach and a cache-friendly succinct data structure. The algorithm is realized by three parallel stages for sorting, candidate generation, and object pairing. By the use of temporal coherence, our sorting stage runs with close to optimal load balancing. Furthermore, our approach is characterized by a work-division strategy that relies on adaptive partitioning, which leads to almost ideal scalability. Experimental results show high performance for up to millions of objects on modern multi-core CPUs.

Categories and Subject Descriptors (according to ACM CCS): F.2.2 [Analysis Of Algorithms And Problem Complexity]: Nonnumerical Algorithms and Problems—Geometrical problems and computations; I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures;

1. Introduction

Many applications in computer graphics, animation, and visualization include different forms of multi-body or n-body simulations. Such simulations are also used to study various types of phenomena in computational branches of physics, chemistry, and biology [SSW01]. In large-scale scenarios, a particularly challenging and crucial feature is the ability to detect collisions in real-time. In this paper, a scalable solution is developed which can be used as a general building block in simulations of millions of bodies.

The Sweep and Prune (SaP) algorithm has proven to be particularly useful in practice as a coarse grained collision culling method [CLMP95, Ber03, Eri04, PCM12]. By sorting the bounding boxes of the objects along one or several axes, and examining their overlap status by sweeping, it outputs the set of colliding box pairs. The algorithm aims to eliminate a vast majority of all the possible $O(n^2)$ combinations of box pairs. As such, the SaP method is an example of a top-level or broad phase collision search [Hub96].

Other competing algorithms that can be used to determine a similar spatial ordering of the objects often rely on spatial subdivision using data structures such as BSP trees, k-d trees, octrees, and bounding volume hierarchies [LCF05, Ben75, VCC98, Sam05]. Bucketing approaches based on uniform grids, non-uniform grids, and spatial hashing have proven to be useful as well [Ove92, THM*03]. These techniques aim to localize collision searches to gain performance. Uniform subdivision is the simplest, but its effectiveness is severely reduced when objects of widely varying sizes are used [MHN11]. Recently, several parallelization strategies for broad phase collision detection on both multi-core CPU and many-core GPU architectures have been considered. These methods mainly focus on SaP, uniform grid partitioning, and brute force testing [LHLK10, GTT13, AGA12, LLCC13].

The purpose of the broad phase is to find the set of colliding box pairs. Usually, since the boxes are used to approximate more complex and arbitrarily shaped objects contained inside them, this output represents a potentially colliding set, where each pair of objects has been found to be sufficiently close to each other to warrant a more detailed investigation of their overlap status. In such cases, a narrow phase follows which produces the exact colliding set by more fine-grained checks [Hub96]. In fact, more or less the same data structures can be used also in this case. It is the division into

DOI: 10.2312/pgv.20161177

a two-phased approach that has proven to be very practical. Each phase can then be implemented and optimized for more specific operational circumstances. A commonly utilized combination is to use SaP for the approximate broad phase testing, and bounding volume hierarchies for exact narrow phase testing [CS08, PCM12]. What combination of data structures that gives the best overall performance is an open and challenging research question, which is complicated by the fact that the best choices are often application and scenario dependent. For a broader picture of the collision detection problem, interested readers can consult published surveys (e.g., [TKH*05], [KHI*07], or chap. 2 in [Wel13]).

This work extends our previous paper that presents an efficient sequential SaP algorithm [CL16]. We show how all the major parts can be efficiently parallelized, which lead to an attractive scalable solution for simulating millions of moving objects, even in dense simulation environments. Our main contributions are: (i) A parallel index sorting algorithm based on radix sort that exploits temporal coherence to enable dynamic load balancing. (ii) A parallel dual-axis sweeping approach that uses succinct tree structures for cache friendliness and scenario-adaptive load balancing. (iii) A heuristic method for the dynamic choice of sweeping axes based on temporal coherence to efficiently handle clustering scenarios. (iv) A detailed experimental evaluation of the proposed solution.

2. Prior work

David Baraff introduced the original "Sort and Sweep" algorithm [Bar92]. It works by projecting the bounding boxes onto one of the main axes to obtain an array of lower and upper interval bounds. Overlapping pairs can then be determined by sorting the array and tracking interval overlaps by a scanning procedure. When an interval overlap is found, the full box-box overlap status is determined by considering also the other two axes. Insertion sort can be used to re-sort the array for each discrete frame of the simulation. By temporal coherence, it can be assumed that the array is kept almost sorted throughout the simulation, and if so, the dynamic box overlapping problem can be solved in $O(n+s)$ time, where s is the number of swaps used in the sorting. Thus, it can be argued that this algorithm is very fast when $s-k$ is small, where k denotes the size of the output. Interestingly, there are theoretical results in computational geometry showing that the box intersection problem can be solved in worst-case $O(n \log n + k)$ time [PS85].

Another early account of sweep and prune is presented for the I-Collide system [CLMP95]. In this realization, three lists, one for each principal coordinate axis, are kept sorted using insertion sort, and the swaps trigger changes in a table of $O(n^2)$ bit flags that represents the overlap status of all possible pairs of intervals. While this may be efficient in some cases, the storage cost prohibits large-scale simulations.

Unfortunately, these classical SaP methods suffer from two drawbacks. First, the sorting is performed using insertion sort in $O(n)$ time under the assumption that the arrays are almost sorted, which can be expected due to temporal coherence. However, when the coherence is lost, the sorting turns into a major bottleneck. Second, in large scale simulations, the number of false positives along sweep axes tends to be superlinear in n [TBW09].

In an unpublished document, Terdiman gives an overview of different SaP methods, including some improvements that addresses these issues, such as Multi-SaP, which combines SaP with spatial subdivision [Ter07]. An improved and more detailed analysis of SaP further motivates such hybrid SaP/spatial subdivision approaches [TBW09]. Efforts to parallelize the computations usually focus on similar hybrid strategies. In particular, GPU-based collision culling approaches have been proposed that rely on a combination of space subdivision and SaP [LHLK10, MZ15]. Also, as an attempt to reduce the number of false positives during the sweep phase, a preferred sweep direction can be selected based on principal component analysis (PCA).

To support continuous collision detection, event-driven SaP methods can be designed. By utilizing known motion trajectories of the bodies, per-frame overhead can be reduced [CS06], and further, by introducing a velocity-aligned bounding volume, objects with high velocities can be handled more efficiently [CS08]. Furthermore, although SaP is used mainly as a broad-phase collision culling method, it is applicable also in contexts where more fine-grained intersection determination is needed. The SaP method can be called recursively level-wise during dual traversal of object hierarchies [PML97]. Another variant targets self-collisions within a breakable or exploding polygon mesh [LAM06]. Recently, yet another adaptation was proposed that aims to handle all parts of the collision detection process [MZ15].

3. Sequential Bi-dimensional SaP

This section briefly presents our sequential bi-dimensional SaP approach [CL16]. We first explain the idea behind the method in terms of the well-known SaP approach by Baraff [Bar92], then we give the details of our implementation and the data structures used.

In general, SaP deals with finding overlapping pairs of objects in a 3D world. Each simulated object is bounded by an axis-aligned box (AABB). Let a denote the projection of the AABB related to a given object on a coordinate axis. We consider a as a *closed interval* represented by an ordered pair of values $[a^+, a^-]$ with $a^+ < a^-$, also called *low-endpoint* and *high-endpoint*, respectively. Clearly, two AABBs collide iff the corresponding projections overlap on all the three coordinate axes.

Every time a low-endpoint is picked, the original SaP algorithm inserts the corresponding object in a data structure

called *activelist*. Such an object remains *active* until the corresponding high-endpoint is picked, then it is removed from the activelist. Finally, when an object is inserted, its projection overlaps with the ones of all the other objects currently stored in the activelist.

In contrast, the bi-dimensional approach splits the computation in two phases performed on two different coordinate axes. Given a primary coordinate axis, the first phase sweeps the sorted list of endpoints to localize, for each object, the ones (referred as *candidates*) that possibly collide with it since their AABBs overlap on the current axis. Each candidate set is defined as a range of objects of which the boundaries are retrieved from the activelist respectively when the low- and high-endpoint of an object are picked. In particular, the low boundary corresponds to the least recent object stored in the activelist when low-endpoint is picked, while the upper boundary corresponds to the last object added to the activelist before the high-endpoint is picked.

In the second phase, since for every object the candidate set represents a superset of its collisions, we sweep the ordered set of endpoints of a different coordinate axis. In particular, only the objects belonging to the intersection of the candidate set (computed on the primary axis) and the activelist (computed on the secondary axis) are tested.

Algorithm 1 describes in more details the bi-dimensional SaP computation. Here, the pseudo-code slightly differs from the description above, but they are equivalent. In particular, the pseudo-code makes use of the *rank* (defined for each object i as the position of i^+ in the ordered set of low-endpoints of the primary axis) to represent an object in the activelist and to keep its position during the whole computation. Ranks are defined in the array $R[]$, while $L[]$ and $U[]$ store the candidate-set boundaries for each object $i \in \Omega$. The first phase is computed in the loop on Line 6. Here, for each object i, $L[i]$ and $U[i]$ are retrieved when i^+ and i^- are respectively picked. The second phase begins on Line 16. For each new picked object i, the related range of candidates, i.e., $[L[i], U[i])$, is used to filter the activelist S by means of a range query (Line 18). On each returned object, a full box-to-box test is performed to find a new possible collision for the object i.

The bi-dimensional approach needs to be supported by an efficient implementation of the activelist during the two phases. The operations required in Algorithm 1 are: inserting and removing objects (on Lines 11, 14, 21, and 23), retrieving the minimum (on Line 10), and performing range queries (on Line 18). As a consequence, we defined a succinct data structure, called SuccTree. Assuming the w-bit word-RAM model [CR72], a SuccTree consists of a w-ary tree with n leaves, where n equals the number of simulated objects. Each level of a SuccTree is implemented by a bit vector of which entries are the nodes. The value of a bit is used as a flag of presence for the node: in this way nodes have a fixed position (which simplifies to locate parent, children and sibling of a node) and they can be easily added and removed by flipping the corresponding bit. Each leaf is associated to a value in $\Omega = \{0, \ldots, n-1\}$ so that the activelist is represented by the set bits in the vector associated with the bottom level of the tree, while the upper levels are used to efficiently move from an active leaf to the next one. In particular, the bit corresponding to a node i (which is not a leaf) is set iff at least one of its children is set. To this end, the parent of the node at position i at some level d is the node at position $\lfloor i/w \rfloor$ at level $d+1$. According to this rule, the procedure for adding (or removing) an element in a SuccTree, sets the corresponding bit at the bottom level of the tree, then updates the ancestors when it is required. The operation used to iterate through the stored items is $succ(i)$ which returns the position of the least leaf greater than the i-th one (if it exists) by searching the lowest common ancestor between i and $succ(i)$. When such a node has been found, a path is computed to its least descendant greater than i, i.e., $succ(i)$. All these operations act by traversing at most two paths of the SuccTree so that they are performed in $O(\log_w n)$ time. This is possible since the SuccTree uses the bitwise instruction set of the CPU to exploit bit-level parallelism and performs in $O(1)$ time operations on w-bit words that otherwise

Algorithm 1 Sequential bi-dimensional SaP.

Input: $\Omega = \{0, \ldots, n-1\}$ ▷ object *id*s
Input: $X[2n]$ ▷ 1st axis endpoints
Input: $Y[2n]$ ▷ 2nd axis endpoints
Input: $Z[2n]$ ▷ 3rd axis endpoints
Output: C ▷ list of *id* pairs of colliding objects

```
1:  L[n], U[n]                          ▷ candidate-set boundaries
2:  R[n], R⁻[n]                         ▷ object rank and inverse of rank
3:  S ← SuccTree(n)                     ▷ init. ∅
4:  c ← 0                               ▷ rank counter
5:  I_x ← IDXSORT(X)
6:  for each i ∈ I_x do
7:    if i < n then
8:      R[i] ← c                        ▷ assign rank
9:      R⁻[c++] ← i                     ▷ assign inverse of rank
10:     if S = ∅ then L[i] ← c else L[i] ← S.min()
11:     S.ins(R[i])                     ▷ activate i by means of R[i]
12:   else
13:     U[i − n] ← c                    ▷ i − n is the id of the object
14:     S.del(R[i − n])
15: I_y ← IDXSORT(Y)
16: for each i ∈ I_y do
17:   if i < n then
18:     for each r ∈ S ∩ [L[i], U[i]) do   ▷ range query
19:       j ← R⁻[r]                     ▷ retrieve candidate id
20:       if OverlapTest(i, j) then C.add(i, j)
21:     S.ins(R[i])
22:   else
23:     S.del(R[i − n])
```

have $O(w)$ complexity. Finally, since a range query is performed by iterating the *succ*() operation between two given boundaries, its complexity is $O(\ell \cdot \log_w n)$ with ℓ equal to the number of values returned by the query.

4. Parallel Bi-dimensional SaP

Here we describe the parallel version of the approach shown in Section 3. In particular, we divide Algorithm 1 in three parts which are treated independently: sorting on Lines 5 and 15, generating the candidate intervals in the loop starting on Line 6, and pairing phase in the loop starting on Line 16.

4.1. Sorting

Our sorting approach makes use of *temporal coherence* to fairly split an input set of endpoints into partitions which can be sorted in parallel independently. To this end, through the SaP iterations, for each axis to sort, a set B of boundaries is used for splitting the input at the beginning of the procedure, then B is updated before the procedure ends. The algorithm used for sorting each partition is the sequential sorting applied in Algorithm 1. It is a variant of the stable Least Significant Digit Radix Sort (LSDR), which returns the indexes of the sorted items instead of permuting the input points. The $2n$ endpoints of each coordinate axis are stored in separate arrays and, for each object $i \in \Omega$, the corresponding i^+ and i^- are respectively placed at position i and $i+n$ of each array[†]. Firstly, the parallel sorting assigns each endpoint to the proper bucket according to B, then the sequential LSDR is applied to each partition.

Algorithm 2 Parallel sorting.

Input: X ▷ endpoint set to sort with $|X| = 2n$
Input: m ▷ number of partitions
Input: B ▷ boundaries set
Output: I ▷ sorted X indexes
1: **for each** $x \in X$ **do in parallel** ▷ setup
2: $i \leftarrow 1 + \sum_{j=1}^{m-1} \text{INT}(x > b_j)$
3: $P_i \leftarrow P_i \cup \{x\}$ ▷ P_i init. $\varnothing$
4: **for each** $i \in [1,m]$ **do in parallel** ▷ sorting
5: $I_i \leftarrow \text{IDXSORT}(P_i)$
6: **for each** $i \in [1,m-1]$ **do** ▷ update B
7: $b_i \leftarrow X[I[i \cdot \lfloor 2n/m \rfloor]]$
8: **return** I

Algorithm 2 shows in more details the parallel procedure described above. Let X denote the set of endpoints to reorder with $|X| = 2n$. The set X is firstly divided into m partitions, $P_{1..m}$, by means of $m-1$ values, $B = \{b_1, \ldots, b_{m-1}\}$ (Line 2). Let $b_0 = -\infty$ and $b_m = +\infty$, then we have that $P_i = \{x \in X : b_{i-1} \leq x < b_i\}$. Values in B are extracted from the ordered list of points sorted at the previous iteration. In fact, they correspond to the points stored at distance $\lfloor 2n/m \rfloor$ to each other (Line 7). Since the position of the simulated objects does not vary drastically between two consecutive SaP iterations, the partitions calculated in this way are almost equally sized. On the first iteration, instead, B is initialized by computing the span of X as $\text{span}(X) = \max(X) - \min(X)$, and setting $b_i = i \cdot \text{span}(X)/m + \min(X)$ for each $i \in \{1, \ldots, m-1\}$. Once the setup of the partitions is done, m threads sort the defined partitions in parallel (Line 4) using the LSDR algorithm. Indexes related to each sorted partition P_i are stored in m contiguous arrays I_i (Line 5) such that $|I_i| = |P_i|$ and $I = \langle I_1, I_2, \ldots, I_m \rangle$. The final part of the algorithm updates B to be used in the next iteration.

4.2. Candidates Generation

The main problem for parallelizing this phase deals with the management of the SuccTree S in Algorithm 1. In each iteration of the loop on Line 6, S is modified by adding or removing values so that S in the current iteration depends on the previous one. Since candidates are a function of S, we determine S at any given point of the computation before we can compute the candidate intervals. Then we can parallelize the sequential loop in different independent parts so that each thread can compute the candidate boundaries for the assigned part by using a private instance of S. Algorithm 3 briefly describes such a parallel approach by skipping the details that are identical in the sequential version.

In Algorithm 1, S is initially empty and, at the beginning of each iteration of the loop on Line 6, S contains the rank of the active objects. Such a loop consists of $2n$ iterations and our preliminary goal is to identify the objects stored in S every $\lfloor 2n/m \rfloor$ iterations starting from the first one. In Algorithm 3, we firstly compute (in parallel) the object ranks by means of the following steps: dividing the set I of sorted indexes into m equally-sized intervals $I_{1..m}$ (Line 2); counting the low-endpoints in each partition (Line 5); computing a prefix sum on such values (Line 7); calculating the object ranks of each partition (Line 10). Once the ranks are computed, we split the sorted input array I in m partitions, $I_{1..m}$, made up of consecutive endpoints. Moreover, for each partition I_p, an empty SuccTree, S_p, is instantiated. Such partitions are then swept by activating and deactivating in S_p the object-rank of the endpoints belonging to I_p (loop at Line 13). At the end, each S_p contains the ranks of the objects having only one endpoint in the corresponding partition I_p. Now, we merge the values of two consecutive SuccTrees, S_p and S_{p+1}, using the *symmetric difference* operator $\oplus$ defined as: $S_p \oplus S_{p+1} = (S_p \cup S_{p+1}) \setminus (S_p \cap S_{p+1})$. By means of $\oplus$, which is known to be associative, we compute the *all-prefix-sums* on the set of SuccTrees on Line 15. Each resulting S_p contains the ranks of the objects i such that i^+ be-

[†] This means that we can easily disambiguate the type of an endpoint-index by comparing its value with n (e.g., Lines 7 and 17 in Algorithm 1) and calculate the object *id* of an high-endpoint by subtracting n (e.g., Lines 13 and 23 in Algorithm 1).

Algorithm 3 SaP – Parallel candidates generation.

Input: $\Omega = \{0,\dots,n-1\}$ ▷ object *id*s
Input: I ▷ array of $2n$ indexes returned by sorting
Input: m ▷ number of partitions
1: $\Delta \leftarrow 2n/m$ ▷ partition size, assume $2n$ is multiple of m
2: **for each** $p \in [1,m]$ **do**
3: $I_p \leftarrow I[(p-1)\Delta,\dots,p\Delta-1]$
4: $c_{1..m} \leftarrow 0$ ▷ m local rank-counters
5: **for each** $i \in I_p$ **do in parallel**
6: **if** $i < n$ **then** c_p++
7: AllPrefixSums($c_1,\dots,c_m$)
8: **for each** $p \in [1,m]$ **do** ▷ make a copy of $c[]$
9: $c'_p \leftarrow c_p$
10: **for each** $i \in I_p$ **do in parallel**
11: **if** $i < n$ **then** $R[i] \leftarrow c'_p$++
12: $S_{1..m} \leftarrow \varnothing$ ▷ m SuccTrees
13: **for each** $i \in I_p$ **do in parallel**
14: **if** $i < n$ **then** $S_p.ins(R[i])$ **else** $S_p.del(R[i-n])$
15: AllPrefixSums($S_1,\dots,S_m$)
16: **for each** $i \in I_p$ **do in parallel**
17: **if** $i < n$ **then**
18: $R^-[c_p{+}{+}] \leftarrow i$
19: **if** $S_p = \varnothing$ **then** $L[i] \leftarrow c_p$ **else** $L[i] \leftarrow S_p.min()$
20: $S_p.ins(R[i])$
21: **else**
22: $U[i-n] \leftarrow c_p$
23: $S_p.del(R[i-n])$

longs to a partition $I_{p'<p}$ and i^- belongs to a partition $I_{p''\geq p}$. In other words, S_p stores the ranks of the active objects at the point of the computation when I_p begins. Note that also the computation of the all-prefix-sums is performed in parallel by following the tree-based approach shown in [Ble90]. Once the all-prefix-sums operation is computed, the rest of the procedure computes the sets of candidates on each partition in parallel similarly as done in Algorithm 1.

4.3. Pairing Phase

The pairing phase dominates the runtime, in particular, in the most challenging scenarios with high density. As a consequence, parallelizing the loop on Line 16 in Algorithm 1 is the most crucial point of our solution.

Similarly to other solutions presented in the literature and discussed in Section 1 and 2, also ours can be regarded as a *spatial partitioning* technique. In contrast with several existing approaches, however, Algorithm 1 has been parallelized by *adaptively* dividing the space in non-uniform portions made up of an equal number of objects. In particular, the subdivision is calculated directly during the pairing phase by splitting the candidate set of each object. In this way longer intervals (e.g. due to clustered objects) are "spread" on more threads so as to improve the workload balance and the overall throughput.

Algorithm 4 SaP – Parallel pairing phase.

Input: $\Omega = \{0,\dots,n-1\}$ ▷ object *id*s
Input: I ▷ array of $2n$ sorted indexes
Input: m ▷ number of partitions
Output: C ▷ list of *id* pairs of colliding objects
1: $\Delta \leftarrow n/m$ ▷ partition size, assume n multiple of m
2: **for each** $p \in [0,m)$ **do in parallel**
3: $S_p \leftarrow \varnothing$ ▷ local SuccTree
4: $\beta \leftarrow p \cdot \Delta$ ▷ partition begin
5: $\varepsilon \leftarrow \beta + \Delta$ ▷ partition end
6: **for each** $i \in I$ **do**
7: **if** $i < n$ **then**
8: $l \leftarrow L[i]$
9: $u \leftarrow U[i]$
10: **if** $l < \varepsilon \wedge \beta \leq u$ **then**
11: **if** $\lfloor l/\Delta \rfloor = p$ **then** $l \leftarrow l - \beta$ **else** $l \leftarrow 0$
12: **if** $\lfloor u/\Delta \rfloor = p$ **then** $u \leftarrow u - \beta$ **else** $u \leftarrow \Delta$
13: **for each** $r \in [l,u) \cap S_p$ **do**
14: $j \leftarrow R^-[r+\beta]$
15: **if** OverlapTest(i, j) **then** $C.add(i,j)$
16: **if** $\lfloor R[i]/\Delta \rfloor = p$ **then** $S_p.ins(R[i]-\beta)$
17: **else**
18: **if** $\lfloor R[i-n]/\Delta \rfloor = p$ **then** $S_p.del(R[i-n]-\beta)$

All steps of the parallel pairing phase are shown in Algorithm 4. The set Ω of objects is divided in $P_{0..m-1}$ partitions that will be computed in parallel by m threads. We firstly compute the size of a partition as $\Delta = n/m$ so that objects of which rank belongs to $[i\Delta, i\Delta+\Delta)$ are assigned to the partition P_i and, for each thread, a Δ-sized SuccTree S_p is instantiated. Each thread sweeps independently the set of sorted endpoints I by discovering the collisions between an object, when it is activated, and the corresponding candidates covered by the thread partition. In particular, when the i-th thread picks an index related to the low-endpoint of an object, if the intersection between the object candidates and P_i is not empty (Line 10), the thread shrinks the boundaries of such an interval to fit in the portion covered by P_i (Lines 11 and 12) and performs the usual range query by means of such local boundaries (loop starting on Line 13). Furthermore, for each index picked from I, the related object rank is represented only in the SuccTree of the partition P_i to which the rank belongs (conditional steps on Lines 16 and 18). As a consequence, objects colliding with a given object a are discovered (when a is activated) by more than one partitions (depending on the length of the candidate interval), but only the thread in which a is represented can discover the collisions of a with the objects activated between a^+ and a^-. Hence, no duplicates are added to the final set of collisions C by the threads and no reduction phase is needed at the end of the procedure.

5. Dynamic Choice of Sweeping Axes

The key point of the approach discussed in Section 4.1 is that the bucketing phase allows to almost fairly divide the workload among the threads due to temporal coherence. However, there are certain cases when this approach is not effective. For example, when endpoints get severely clustered along a sorting axis, the position of many of them can coincide so that the setup phase can gather most of them in one partition, which unbalances the workload of the threads. Such situations can degrade both the sorting throughput and, in particular, the performance of the entire computation. In such scenarios, however, the third unused axis is likely to be less clustered than the other two. In these cases, there is an opportunity to switch the most clustered axis with the unused one. Intuitively, severe clustering along all three axes at a single time instant can only happen if the objects can interpenetrate. In realistic simulations, the collision response mechanism is supposed to prevent such configurations.

To decide when it is advantageous to swap an axis, we calculate a measure of relative data dispersion D. In particular, it measures the unbalance of the number of elements per bucket (computed in the sorting phase) relatively to the ideal one. To this end, D is defined as a function of the number of objects n, the number of buckets m, and the statistical variable $\mathcal{X}$ describing the number of elements per bucket. Let δ denote the mean deviation[‡] equal to the expected value $E[|\mathcal{X}-\mu|]$ with μ equal to $2n/m$. Since the value of δ is unrelated to the bucket size, the measure of dispersion we adopt is $D = \delta/\mu$, which is computed as follows:

$$D = \frac{\delta}{\mu} = \frac{E[|\mathcal{X}-\mu|]}{\mu} = \frac{\sum_{i=1}^{m} |x_i-\mu|/m}{2n/m} = \frac{\sum_{i=1}^{m} |x_i-\mu|}{2n} \tag{1}$$

During the computation, when partitions turn out to be unbalanced on one of the two main axes (i.e., D is greater than a given threshold) we switch that axis with the unused one. Computing D costs $O(m)$ while the axis switching requires B to be reinitialized for the new axis which costs $O(n)$ (for retrieving the maximum and minimum of the new set of endpoints and compute $b_{1..m}$ as explained in Section 4.1).

6. Experiments

All tests have been run on a dual 2.40 GHz Intel(R) Xeon(R) CPU E5-2630 v3 having 16 physical cores equipped with 64 GB of RAM and using Ubuntu 14.04 with gcc 5.3.0.

In the following we present the results of running different simulations specifically designed to compare performances of our parallel bi-dimensional SaP and its sequential version. In each simulation, n boxes of varying sizes moved freely in a 3D space delimited by a world cube. To challenge our approach, we let colliding boxes pass through each other and bounce on the world boundaries. In this way, we created highly clustered contexts with many deeply overlapping boxes although such scenarios are unrealizable in a simulation of non-penetrating objects. In the beginning, the boxes were spread uniformly in space with density d computed as the ratio of the sum of the boxes' volumes to the volume of the world. In each iteration of a simulation, SaP is firstly used to discover the collisions, then the objects move. Each simulation consisted of 100 iterations and we validated the results by checking that the set of collisions discovered at each iteration was the same for the two solutions. In the following, we present the results of each parallelized part: sorting, candidate generation, and pairing. Finally, we show the details of how these parts affect the overall performance.

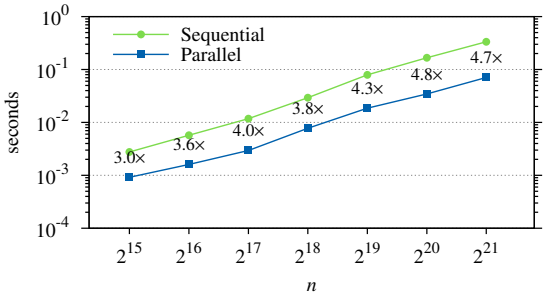

Figure 1: *Elapsed time and speedup of the sorting phase by varying the input size n.*

6.1. Sorting Evaluation

Results in Figure 1 show the performance of the two different versions used for sorting. The sequential one is based on the LSDR algorithm while the parallel method corresponds to Algorithm 2 proposed in Section 4.1. Algorithm 2 mainly consists of two phases: setup and sorting the partitions (time spent in updating the set B is negligible and can be left out of this analysis). The time complexity of the setup phase is linear in the number of objects n since the computation of the object partitions is performed in parallel by m threads, but computing each of them requires $O(m)$ operations. Moreover, even if the parallel time complexity of the second phase is $O(n/m)$, such a phase is more time-consuming as it performs several passes made up only of memory copies to reorder the data. As a consequence of the overhead introduced by the setup phase as well as the memory intense nature of radix sort, the scalability reached in this phase was modest. However, since sorting is not the bottleneck of the sequential computation, the lower scalability is compensated by the performance of the other phases. Figure 1 shows the results related to just one object density (i.e., $d = 0.35$ that is the highest tested value), since sorting is only slightly affected by this parameter.

Finally, we measured also the workload balance of our

[‡] The mean deviation (also called the mean absolute deviation) is the mean of the data's absolute deviations around the data's mean.

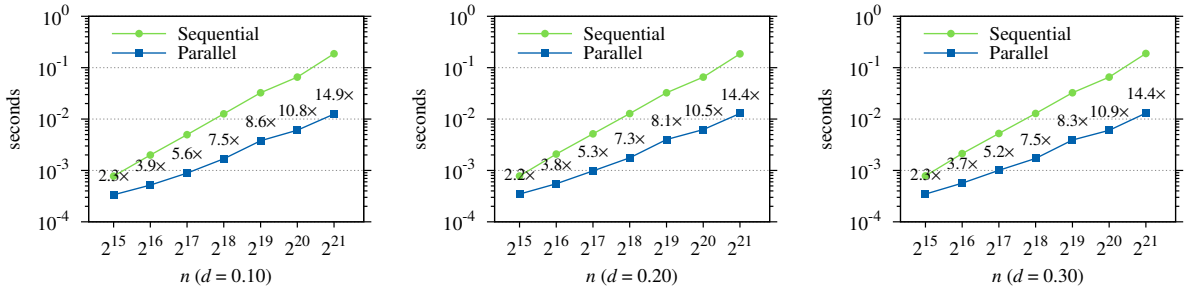

Figure 2: *Elapsed time and speedup for generating candidates by varying the input size n and the object density d.*

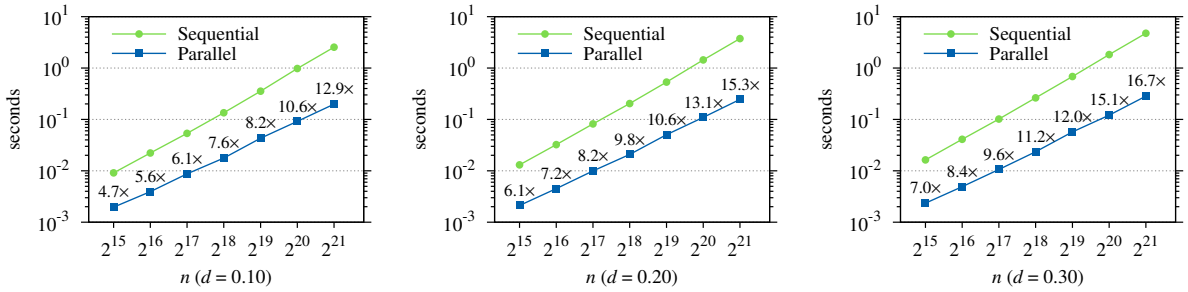

Figure 3: *Elapsed time and speedup of the pairing phase by varying the input size n and the object density d.*

parallel method to evaluate the effectiveness of computing the boundary set B by exploiting temporal coherence. To this end, we measured D, as defined in Section 5, in each frame of the simulation. The results showed that D was always lower than 0.01, which means that the number of endpoints to sort assigned to the buckets missed the perfectly balanced size by less than 1%.

6.2. Evaluation of Candidates Generation

Figure 2 shows that generating the object candidates in parallel as described in Section 4.2 has overall good performance and speedup. Given a specific input size, we observed that, in some cases, the measured speedup slightly decreased as the object density increased. In fact, since more object projections overlap each other on the primary axis in such cases, the average number of endpoints falling in between the endpoints of an object increases. As a consequence, the probability that both endpoints belong to the same thread partition is lower so that the average size of the m SuccTrees (on which the all-prefix-sum operation is computed in Algorithm 3) grows, which slows down the phase.

6.3. Pairing Phase Performance

Figure 3 shows the performance of the parallel pairing phase. Our adaptive space-subdivision approach exhibits a good scalability, which increases as the number of simulated objects and their density grow. This is mainly due to the fact that the computation is synchronization-free and no reduction phase is required at the end of the process. Last but not the least, the almost perfect load balancing observed in the experiment led to a better throughput since it minimized the average response time of each thread. In fact, we measured the thread workload in all simulations by collecting the percentages of the number of collisions detected by the various threads. The results showed small values of standard deviation, σ, of the percentages, which means that workload was almost perfectly distributed among the threads. We also repeated the same tests by varying the number of threads from 4 to 64 and we obtained similar results, i.e., $\sigma < 1.5\%$.

Finally, we observed that, in high-density large-scale simulations, we obtained a super-linear scalability. This is a side-effect of the adaptive space-subdivision which acts as an early-exit condition in some cases. When long candidate intervals are spread out on more than one partition, each thread checks that the assigned part of the original interval is overlapping with its own partition (Line 10). When such a test fails, the current iteration ends immediately and no access is done to the SuccTree. In the sequential case, instead, at least one access to the SuccTree is made for every low-endpoint picked. Furthermore, the SuccTree instances used in the parallel case are smaller (because only a fraction of the the entire set of objects is managed by each of them) which reduces the complexity of the SuccTree operations.

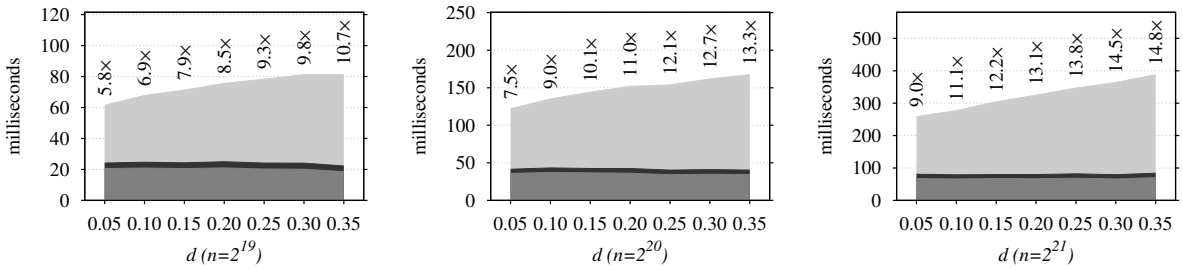

Figure 4: *Elapsed total time of the three phases of our parallel SaP: sorting (the gray layer at the bottom), candidates generation (the dark gray layer in the middle), and pairing (the light gray layer at the top) computed by varying the object density for the tested input sizes. For each density and each input size, speedup of the overall parallel computation calculated with respect to the sequential runtime is shown above the gray areas.*

6.4. Overall Parallel Performance

Figure 4 presents how each part of the computation affected the global performance of our parallel SaP. We show the time spent for sorting the primary and secondary axes, generating the candidates, and pairing. In this case, we present the results of the experiment by varying, in each chart, the object density on the abscissa and selecting three different input sizes: 2^{19}, 2^{20}, and 2^{21} (roughly half, one, and two millions). As expected, the time spent in the first two phases remained almost constant since the object density affected the performance of sorting and the candidate generation only slightly. On the other hand, the pairing phase dominated the elapsed time of the entire algorithm. Moreover, as the scenario became more dense, the time spent in this phase grew due to the increasing number of collisions.

7. Clustering Scenarios

To further challenge our algorithm, two additional experiments were designed to examine the behavior under different kinds of clustering of the objects. In the first case, the objects were randomly positioned using a uniform distribution inside a large ball. The velocity vectors were directed towards the center of this ball with the speeds adjusted so to let each object reach the center simultaneously, should they not hit anything on the way. However, a collision response method was used to prevent objects from passing through each other. In this way, the objects formed a dense ball-shaped cluster before a massive number of collisions forced them to spread out in all directions. The whole simulation was run for 500 frames. To illustrate the motion, Figure 5 shows five screen captures of the scenario. The plots under the images give the collision detection times of our parallel algorithm together with the corresponding sequential runtimes for $n = 2^{19} = 524\,288$. The parallel speed-up during the most intense part of this scenario, i.e., frames 75–150 varied in the range 7.5–11.3×. The average time for all 500 frames was 87.4 ms, which compared to the average sequential runtime gave a speedup of 6.5×.

In the next experiment, $n = 750^2 = 562\,500$ equally sized cubes were distributed in a large cube. To define the midpoints of the boxes, we used a regular grid spacing for two of the coordinates, whereas the third coordinate, call it u, was chosen at random to place the objects uniformly distributed above and below the plane $u = 0$. The motions of the objects were then chosen so to let all the objects pass through the plane $u = 0$ simultaneously in the middle of the simulation. An illustration of this scenario is given in Figure 6. The size of the cubes was set so that the cubes almost touched their neighbours while moving through the plane (see the middle image). This means that the entire scenario was free from collisions, but note that the clustering of the objects within a single plane can be a very difficult case causing severe bottlenecks for certain algorithms.

Since our algorithm uses a dual-axis approach, it is less sensitive to clustering in general compared to single-axis approaches. Nevertheless, if clustering occurs along either the primary or the secondary axis, a performance bottleneck occurs, given that the chosen sweeping axes remain fixed throughout the simulation. However, if we use our axis swapping heuristic (described in Section 5), such problems can be avoided in most cases. The performance plots in Figure 6 show the obtained results when we ran our parallel algorithm with and without dynamic axis swapping. In all cases, the algorithm used the x-axis as the primary axis, and the y-axis as the secondary axis to begin with. Without axis swapping, we saw that run-times increased substantially during the most intense part of the plane clustering in frames 225–275. This happened both when the clustering occurred in the x direction ($u = x$) and the y direction ($u = y$), although the negative effect was more pronounced for $u = y$. When our heuristic for dynamic axis selection was used, however, this problem was avoided completely. After the axis swap, which occurred on frame 214, the whole scenario ran at more or less a constant speed (about 40 ms per frame).

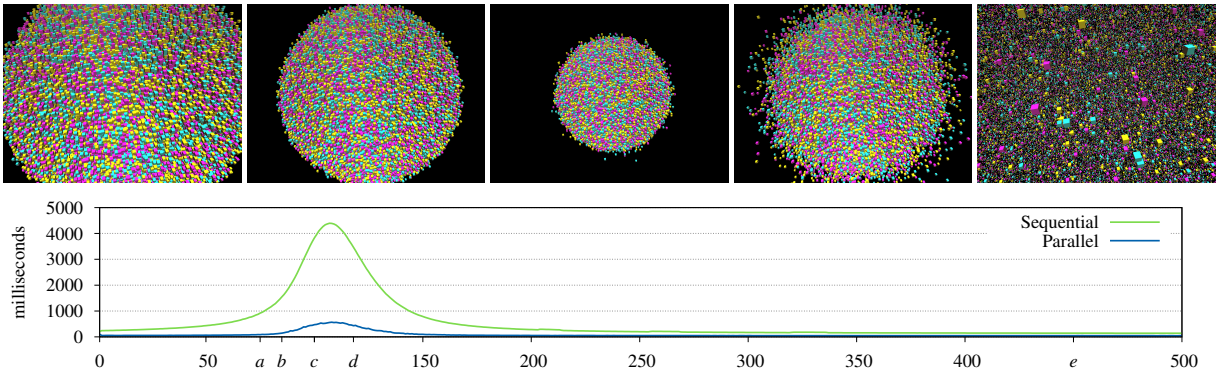

Figure 5: *Ball clustering scenario ($n = 524\,288$). The visualized frames are: $a = 75$, $b = 85$, $c = 100$, $d = 118$, and $e = 450$.*

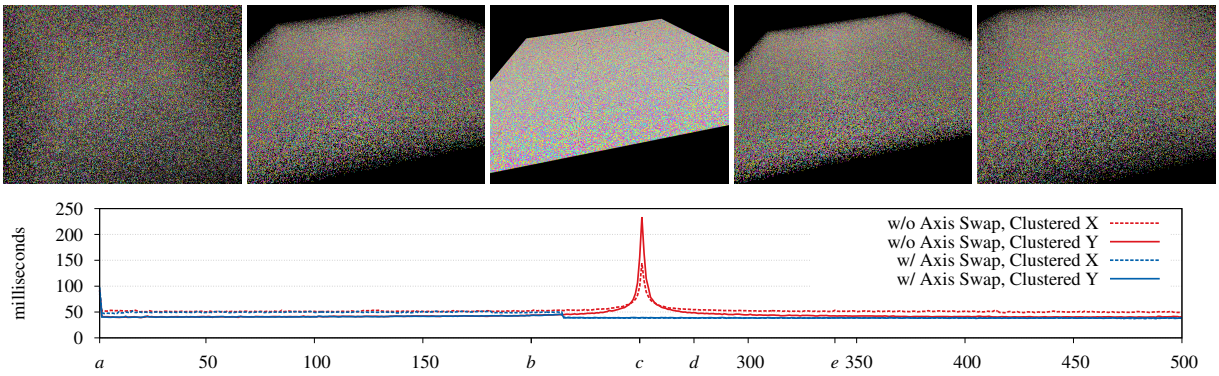

Figure 6: *Plane clustering scenario ($n = 562\,500$). The visualized frames are: $a = 0$, $b = 200$, $c = 250$, $d = 275$, and $e = 340$.*

8. Comparisons to other Algorithms

Our parallel SaP method can be compared with some other algorithms for large-scale simulation scenarios in the literature. In particular, there are some parallel GPU approaches that give high performance. Liu et al. used a hybrid SaP/ uniform grid approach that showed high performance up to one million objects [LHLK10]. They reported a query time of 161 ms for 960K objects using a Tesla C1060. The subdivision method presented by Lo et al. was able to handle 10^7 boxes within one second using a Tesla C2070 [LLCC13].

These examples illustrate that very high rates of collision culling are possible on massively parallel GPU architectures. Our CPU-based algorithm was able to handle one million objects within a range of 122–167 ms depending on the object density d of the simulated scenarios (see the middle plot in Figure 4). If we only consider parallel solutions targeting multi-core CPUs, we are not aware of papers that reports higher performance than we do. Therefore, we conclude that our algorithm is a highly competitive choice for simulations running on CPUs.

9. Conclusions

We have presented a fully parallelized SaP algorithm for the dynamic box intersection problem. By exploiting the architecture of modern CPUs, we realized an efficient, cache-oriented, multi-core solution that scaled up to large datasets. Furthermore, our algorithm was able to handle challenging clustering scenarios without severe performance drops. The experimental results confirm its good qualities in practice resulting in a remarkable boost in the collision culling performance. We achieved almost the ideal speedup, 16×, despite that we did not use SIMD instructions (except for computing the axis span as described in Section 4.1). Thus, there is a chance that performance can be improved further by using, e.g., AVX instructions.

An interesting future line of research would be to port our parallel algorithm onto GPUs. Such devices possess great computational power and they are characterized by a high level of data parallelism. A possibility might be to adapt our SuccTree to the GPU architecture to make it possible to run many instances of it asynchronously. Also, since several high performance collision culling methods targeting GPUs are already known [LHLK10, LLCC13], an interest-

ing next step would be to aim for a heterogeneous parallelization utilizing a combination of CPUs and GPUs. In this case, a technique to automatically tune the workload among the available devices based on their computing power would be needed as well.

Acknowledgments

This research was supported by the Swedish Foundation for Strategic Research (grant IIS11-0060). We are also indebted to the HPC Lab, which is part of ISTI CNR in Pisa, for allowing us to run benchmarks on their hardware.

References

[AGA12] AVRIL Q., GOURANTON V., ARNALDI B.: Fast Collision Culling in Large-Scale Environments Using GPU Mapping Function. In *Eurographics Symposium on Parallel Graphics and Visualization* (2012), pp. 71–80. 1

[Bar92] BARAFF D.: *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD thesis, Cornell University, 1992. 2

[Ben75] BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517. 1

[Ber03] BERGEN G. V. D. (Ed.): *Collision Detection in Interactive 3D Environments*. The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann, 2003. 1

[Ble90] BLELLOCH G. E.: Prefix sums and their applications. In *Synthesis of parallel algorithms*. Morgan Kaufmann, 1990, pp. 35—60. 5

[CL16] CAPANNINI G., LARSSON T.: Efficient collision culling by a succinct bi-dimensional sweep and prune algorithm. In *Proceedings of the 32nd Spring Conference on Computer Graphics* (2016). 2

[CLMP95] COHEN J. D., LIN M. C., MANOCHA D., PONAMGI M.: I-Collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the Symposium on Interactive 3D Graphics* (1995), pp. 189–196. 1, 2

[CR72] COOK S. A., RECKHOW R. A.: Time-bounded random access machines. In *Proceedings of the fourth annual ACM symposium on Theory of computing* (1972), pp. 73–80. 3

[CS06] COMING D. S., STAADT O. G.: Kinetic sweep and prune for multi-body continuous motion. *Computers & Graphics 30*, 3 (2006), 439–449. 2

[CS08] COMING D. S., STAADT O. G.: Velocity-aligned discrete oriented polytopes for dynamic collision detection. *IEEE Transactions on Visualization and Computer Graphics 14*, 1 (2008), 1–12. 2

[Eri04] ERICSON C.: *Real-Time Collision Detection*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. 1

[GTT13] GELERI F., TOSUN O., TOPCUOGLU H.: Parallelizing broad phase collision detection algorithms for sampling based path planners. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on* (2013), pp. 384–391. 1

[Hub96] HUBBARD P. M.: Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics 15*, 3 (1996), 179–210. 1

[KHI*07] KOCKARA S., HALIC T., IQBAL K., BAYRAK C., ROWE R.: Collision detection: A survey. In *IEEE International Conference on Systems, Man and Cybernetics* (2007), pp. 4046–4051. 2

[LAM06] LARSSON T., AKENINE-MÖLLER T.: A dynamic bounding volume hierarchy for generalized collision detection. *Computers & Graphics 30*, 3 (2006), 450–459. 2

[LCF05] LUQUE R. G., COMBA J. A. L. D., FREITAS C. M. D. S.: Broad-phase collision detection using semi-adjusting BSP-trees. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games* (2005), I3D '05, ACM, pp. 179–186. 1

[LHLK10] LIU F., HARADA T., LEE Y., KIM Y. J.: Real-time collision culling of a million bodies on graphics processing units. *ACM Trans. Graph. 29*, 6 (2010), 154:1–154:8. 1, 2, 9

[LLCC13] LO S. H., LEE C. R., CHUNG I. H., CHUNG Y. C.: Optimizing pairwise box intersection checking on GPUs for large-scale simulations. *ACM Transactions on Modeling and Computer Simulation 23*, 3 (2013), 19:1–19:22. 1, 9

[MHN11] MAZHAR H., HEYN T., NEGRUT D.: A scalable parallel method for large collision detection problems. *Multibody System Dynamics 26*, 1 (2011), 37–55. 1

[MZ15] MAINZER D., ZACHMANN G.: Collision detection based on fuzzy scene subdivision. In *GPU Computing and Applications*. Springer, 2015, pp. 135–150. 2

[Ove92] OVERMARS M. H.: Point location in fat subdivisions. *Inf. Process. Lett. 44*, 5 (1992), 261–265. 1

[PCM12] PAN J., CHITTA S., MANOCHA D.: FCL: A general purpose library for collision and proximity queries. In *IEEE International Conference on Robotics and Automation (ICRA)* (2012), pp. 3859–3866. 1, 2

[PML97] PONAMGI M. K., MANOCHA D., LIN M. C.: Incremental algorithms for collision detection between polygonal models. *IEEE Transactions on Visualization and Computer Graphics 3*, 1 (1997), 51–64. 2

[PS85] PREPARATA F. P., SHAMOS M. I.: *Computational Geometry: An Introduction*. Springer-Verlag, 1985. 2

[Sam05] SAMET H.: *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2005. 1

[SSW01] SIGURGEIRSSON H., STUART A., WAN W.-L.: Algorithms for particle-field simulations with collisions. *Journal of Computational Physics 172*, 2 (2001), 766–807. 1

[TBW09] TRACY D. J., BUSS S. R., WOODS B. M.: Efficient large-scale sweep and prune methods with AABB insertion and removal. In *Proceedings of the 2009 IEEE Virtual Reality Conference* (2009), IEEE Computer Society, pp. 191–198. 2

[Ter07] TERDIMAN P.: Sweep-and-prune. Online, Sept 2007. 2

[THM*03] TESCHNER M., HEIDELBERGER B., MÜLLER M., POMERANERTS D., GROSS M.: Optimized spatial hashing for collision detection of deformable objects. In *Proc. Vision, Modeling, Visualization VMV 2003* (2003), pp. 47–54. 1

[TKH*05] TESCHNER M., KIMMERLE S., HEIDELBERGER B., ZACHMANN G., RAGHUPATHI L., FUHRMANN A., CANI M.-P., FAURE F., MAGNENAT-THALMANN N., STRASSER W., VOLINO P.: Collision detection for deformable objects. *Computer Graphics Forum 24*, 1 (2005), 61–81. 2

[VCC98] VEMURI B. C., CAO Y., CHEN L.: Fast collision detection algorithms with applications to particle flow. *Computer Graphics Forum 17*, 2 (1998), 121–134. 1

[Wel13] WELLER R.: *New Geometric Data Structures for Collision Detection and Haptics*. Springer, 2013. 2