An efficient space partitioning technique based on linear kd-trees for simulation of short-range interactions in particle methods

Laurent Poirrier

November 22, 2009

Abstract

We present an efficient pruning algorithm for selecting candidate short-range interaction pairs in particle-based discretization methods such as the discrete element method or smoothed particle hydrodynamics. Geometrically, our technique is based on a recursive bisection of space, forming a particular type of kd-tree. But instead of relying on a tree data structure, our algorithm traverses an implicit tree stored as an associative array. We study its time and memory complexity, and compare the performance of our implementation with that of other algorithms (namely Sweep and Prune and Delaunay triangulation). Preliminary tests show that the linear kd-tree performs more than fifty times faster than the other methods on conventional computers, for highly dynamic simulations with a few thousand elements or more. We also conclude that simulations of several million elements are feasible on a single processor in less than ten seconds per time step.

Keywords particle simulation, kd-tree, linear kd-tree, broad phase, collision detection

1 Particle simulation and collision pruning

The term "particle-based simulation" designates a family of simulation paradigms in which the solution process involves keeping track of moving discrete points (or particles) in space, and evaluating model equations at these points, as opposed to mesh-based methods, in which evaluations are performed at fixed coordinates. Various types of particle-based simulations are encountered in wide range of domains. The most common examples are found in physics (in N-body simulation), computer graphics (in physics engines) and applied mechanics (in meshfree methods such as discrete element method and smoothed particle hydrodynamics).

Two basic operations are performed in these simulations: *(i)* detecting interactions between objects and *(ii)* handling the dynamics of such interactions. Here, we will discuss only the first point, which is called collision detection or interaction detection depending on the application domain. Furthermore, we will limit our study to models featuring *local* interactions (i.e. interactions with finite range), in which case the computation of the set of pairs of interacting particles can be accelerated by splitting the process in two separate passes:

The "broad phase" pass performs a coarse selection of possible interaction pairs, using fast rejection tests. Given this subset, the "narrow phase" pass uses exact element geometry to determine the true interaction status for each candidate pair, and computes interaction dynamics if needed (here, the fact that such an interaction effectively occurs will be interchangeably called a *collision*, an *interaction* or an *intersection* of respective element volumes). Not having this separation would be equivalent to carrying out "narrow phase" tests on all $\frac{1}{2}N(N-1)$ possible pairs, where N is the number of elements. Thus, the addition of a broad phase is performed in the hope to reduce the computational complexity of the overall process to a subquadratic behaviour.

In this paper, we will focus on "broad phase" algorithms (sometimes also referred as "collision pruning" or "collision culling" algorithms), which we believe are bound to be the computational bottleneck of simulations for sufficiently large N, since . Therefore, we will consider that we are given means to represent our elements' geometry, but we will not examine precise interaction determination and handling any further.

Fast rejection tests are achieved adopting a coherence view of the simulation. Roughly, *temporal* coherence relates to the fact that two elements that are close (resp. far away) at a given time of the simulation are likely to stay close (resp. far away) after a few time steps, and thus may interact (resp. probably will not interact). Although other algorithms efficiently take advantage of temporal coherence, we will only consider *geometric* coherence here. It can be stated as such: Given two elements e_1 and e_2 and their respective interaction volumes (or interaction ranges) E_1 and E_2, if there exists two disjoint subspaces S_1 and S_2 such that

$$E_1 \subset S_1 \quad E_2 \subset S_2$$

then E_1 and E_2 do not intersect, and therefore e_1 and e_2 may not interact.

The most immediate way to achieve faster collision detection using geometric coherence is to enclose all complex elements into simpler *bounding volumes* (e.g. a bounding sphere, Figure 1). Then the broad phase consists in testing every bounding volume against each other, passing only pairs with non-empty intersection to the slower narrow phase.

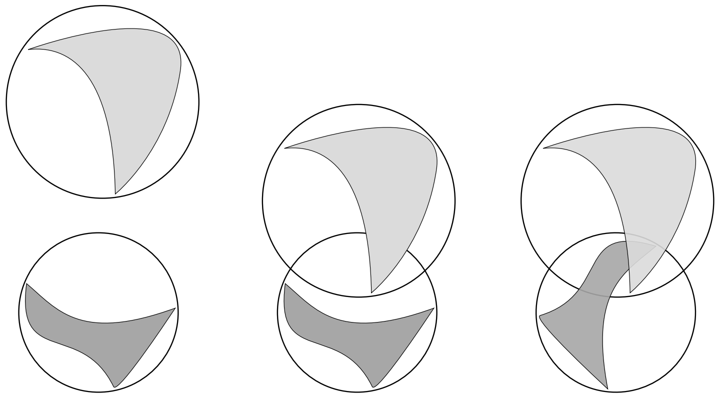


Figure 1: Bounding sphere culling

Such a technique proves very fast in practice for a few thousand elements. But it still shows quadratic behaviour since all possible pairs must be examined, that is $\frac{1}{2}N(N-1) = O(N^2)$ tests.

——- Other methods have subquadratic (average) time-complexity. The most common include spatial partitioning (see [4] for a comprehensive introduction), sweep and sort [1] (also referred as sweep and prune in [3]) and its variants [7], hierarchical pruning [6], and Delaunay triangulation ([5]).

Also note that besides time-complexity, the main characteristics of a "broad phase" algorithm include the proportion of false positives reported as candidate pairs. Both these important characteristics will be tracked in our later comparisons.

2 kd-tree structure

For our purposes, we define a kd-tree (Figure 2) as a tree data structure in which nodes represent a cuboid subspace of $\mathbb{R}^3$, called a *cell*. The root node spans over the interest space. Every non-leaf node has exactly two children, each uniquely comprising one half of its parent's cell, thus together forming a partition of it. Walking the tree downwards from the root to a leaf, the "cutting plane" dividing each cell is successively chosen orthogonal to one of the three main axis.

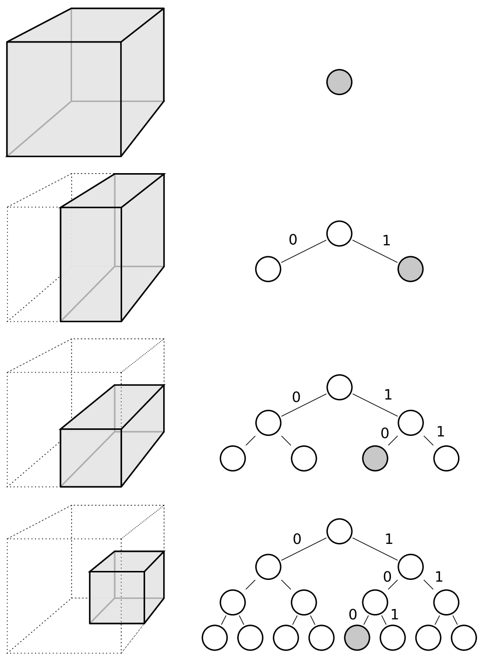


Figure 2: kd-tree

Note that the data structure we are describing here is only a particular subclass of kd-trees, but still we will refer to it as kd-tree for conciseness.

Let us assume that we associate every element with a kd-tree node that entirely encloses its volume. Two element volumes may intersect only if there is a path from one of them to the other in the directed tree. That is, if one of their respective kd-tree nodes is an ancestor of the other or if they coincide. Otherwise, their bounding volumes are disjoint.

Therefore, given a kd-tree in which we placed our elements, one possible way to generate a set containing all possible intersections would be to associate each element with every element in its node and in the sub-tree below it.

3 Using the deepest kd-tree node

When constructing a kd-tree for collision culling, our primary objective is to minimize the number of candidate interactions that will need to be examined in the subsequent traversal. To achieve this, we chose to place each element into the smallest cell (the deepest node) in the tree that entirely encloses it. It is obvious that with any other choice of cell, the number of candidate pairs generated would be greater or equal. Note, however, that this is only one policy among others.

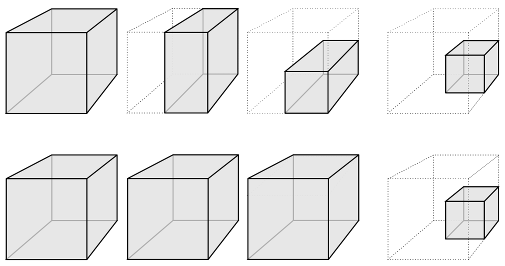


Figure 3: Corresponding bounding volumes in a kd-tree (upper cells) and in an octree (lower cells)

Due to the succession of different cutting plane directions, the structure of a kd-tree is intrinsically anisotropic. In particular, the chosen sequence of cutting directions (among the three available) has an impact on the resulting tree. Here, we will just state that the ability of a kd-tree to minimize cell volume is equivalent or better than that of an analogous octree, which treats each axis equally (Figure 3).

4 Improving kd-tree efficiency by splitting elements

One common problem with space partitioning techniques is dealing with elements that cross a cutting plane. In our case, such a situation is not permitted: we have to place elements in a sufficiently large cell (i.e. low-depth node), so that no crossing occurs. In particular, relatively small objects located just across a low-depth cutting plane (Figure 4, left) imply a heavy penalty, since their enclosing cells have a relatively large volume.

One solution is to systematically split the elements into two sub-elements along the concerned cutting planes, until the achieved total enclosing volume is considered “small enough” (Figure 4, center

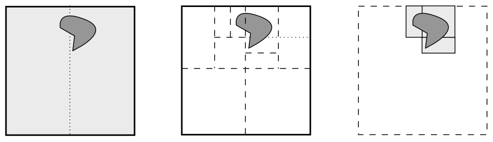


Figure 4: Cell volume reduction with element splitting

and right). Obviously, it is necessary to strike a balance between the desired volume reduction and the resulting increase in the number of sub-elements.

4.1 Axis-aligned bounding boxes

Our proposed criterion is to allow object splitting once along each of the three cutting plane directions. Thus, the overall number of sub-elements is limited to eight times the original number of elements, and we have the following interesting property:

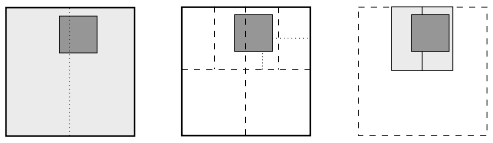


Figure 5: Cell volume reduction with box splitting

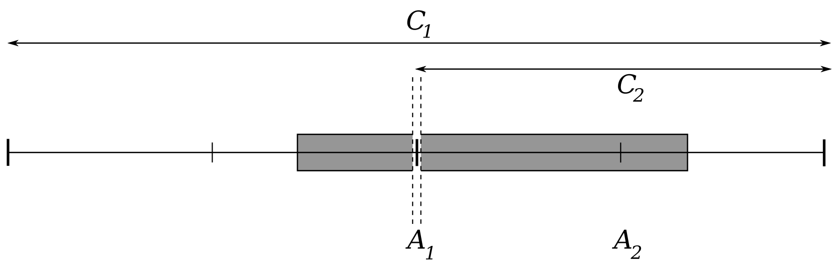


Figure 6: 1-D box splitting

Property 1. *If the elements to place in a kd-tree are axis-aligned boxes, and if we allow splitting them once along each axis, then the resulting sub-boxes will span over more than half of their respective enclosing cell along at least one axis (assuming there is no limit on the depth of the tree, i.e. no lower bound on the size of cells).*

Proof. *Let A be the cutting plane crossed by a (sub-)element in a cell C. Either the element has not been cut by a parallel plane yet (Figure 6, A_1), then it is split in two along A, and we can place the two sub-elements in cells smaller than C. Or the element has already been cut by a parallel plane (Figure 6, A_2), then it is aligned with one of the boundaries of C (Figure 6, C_2). Therefore, it spans over more than a half of C. In this latter case, our placement algorithm has reached the* *deepest enclosing node and stops, so the property is only ensured along the final splitting direction A.* □

As we will see later, considering axis-aligned boxes makes sense in practice, since we will first construct an axis-aligned bounding box (AABB) for each element, and take advantage of a fast method for placing AABBs in our kd-tree (see Section 5.3).

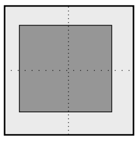


Figure 7: Splitting that leads to no kd-tree improvement

Also note that in our implementation, we suppress any split that does not lead to an overall cell volume reduction (Figure 7).

4.2 Effect on collision culling

Despite being simple, this method leads to dramatic improvements in kd-tree performance. As an illustration, we carried a simple experiment (Table 1) with the following setup:

We generate N same-sized spheric elements. The pseudo-random distribution of the sphere centers is uniform over some cuboid subspace of $\mathbb{R}^3$. The dimensions of that subspace are computed so that the ratio

$$\frac{\text{total volume of elements}}{\text{volume spanned by distribution}} \approx \frac{N\,V_{\text{sphere}}}{\Delta x\,\Delta y\,\Delta z}$$

approximates a given parameter d. Non-interpenetration of spheres is not enforced. Then we place the elements in a kd-tree, with and without element splitting.

In both cases, we measure

- The resulting number of sub-elements, which is always N without splitting, and up to $8N$ with splitting.
- A volume ratio V_n/V_e, where V_n is the overall volume of all enclosing cells (a single node being possibly counted several times), and V_e is the overall volume of all elements. A lower ratio (i.e. closer to 1) corresponds to a higher efficiency at reducing bounding volume.
- The number of candidate interaction pairs selected by the algorithm (described in Section 5.5).
- The number of effective interactions (true positives) among these pairs.

For a given d, the instances of sphere distribution used are identical in both contexts. Therefore, we could expect the resulting number of true positives to be the same. However, with splitting enabled, our algorithm may count a single candidate pair multiple times, since one or both of

d	without splitting				with splitting			
	sub-elements	V_n/V_e	selected pairs	true positives	sub-elements	V_n/V_e	selected pairs	true positives
0.01	N	$8.0275 \,.\, 10^6$	1261197	389	$6.37 \,.\, N$	10.9362	4928	429
0.05	N	$1.5836 \,.\, 10^6$	1834155	1901	$6.67 \,.\, N$	10.8227	25643	2660
0.1	N	$1.3439 \,.\, 10^6$	2300212	3715	$6.24 \,.\, N$	13.4065	62148	6235
0.5	N	$2.9324 \,.\, 10^6$	4193598	16227	$6.35 \,.\, N$	13.4745	280736	63454
1.0	N	$4.0032 \,.\, 10^6$	5552230	28144	$6.01 \,.\, N$	12.5170	492790	72176

Table 1: Effect of bounding box splitting on collision culling ($N = 10000$)

its members may be present in several different nodes of the tree. Hence, these numbers will be different.

It is clear from Table 1 that element splitting represents a broad enhancement in the efficiency of kd-trees at selecting candidate interaction pairs. In our experiment, for the cost of multiplying by 6.5 the number of elements, we reduced by almost a factor 10^6 the average enclosing volume, and by a factor 10 to 100 the number of candidate pairs.

5 kd-tree representation and sweep

So far we have described our kd-trees from a geometrical point of view. We will now cover their representation as a data structure.

5.1 Data structure representation

We do not use a tree structure to hold our kd-tree data in memory. Instead, we build an array of small records, each containing a reference to an element and a *locational code* (or *path*) describing its enclosing cell in the kd-tree. The number of records ranges from N to $8N$ with the splitting policy given previously.

Such a structure will be called a “linear kd-tree” here, as it shares much of the underlying concepts with linear octrees. However, in our linear kd-trees, the records identify elements, instead of nodes in a linear octree (as described in [4]). Therefore, a given locational code may appear several times in the array: this simply means that several elements were placed in the same kd-tree node. Moreover, since it provides no explicit representation of nodes in memory, there is no direct mean to query all elements intersecting a given node of a linear kd-tree.

5.2 Binary construction of a locational code for a point

Any node in the kd-tree may be designated by its path from the root of the tree, i.e. the list of successive branches chosen to get to it from the root node. We adopt the convention to denote by “0” the lower-value half cell and by “1” the higher one. If we bound the depth of the tree, there is

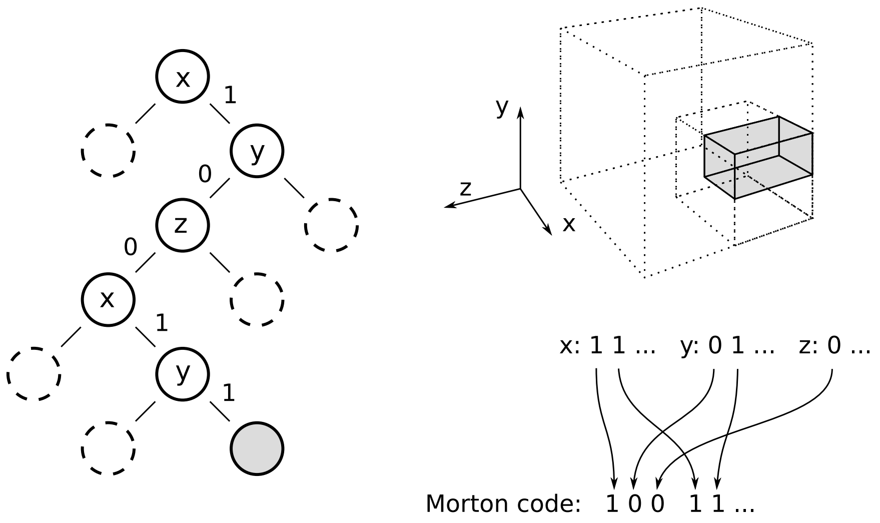


Figure 8: Morton codes in a kd-tree

an efficient method for computing the locational code of the node enclosing a point (x, y, z). Here is its outline (a more detailed explanation can be found in [4] for linear octrees):

Given the interest space (or root cell) C

$$C \equiv\ [x_0, x_1)\ \times\ [y_0, y_1)\ \times\ [z_0, z_1)\ \ \subset \mathbb{R}^3$$

we can construct a surjective mapping from $a \in [a_0, a_1)$ to the B bits integer $a' \in \{0, 1, 2, \ldots, 2^B - 1\}$:

$$a' = \left\lfloor (a - a_0) \frac{2^B}{a_1 - a_0} \right\rfloor$$

where a is one of $\{x, y, z\}$. The ith bit of a' indicates on which side of the ith cutting plane orthogonal to axis a the point (x, y, z) lies (Figure 8). Therefore, if we interleave the bits of x', y' and z', we get a $3B$ bits sequence, known as *Morton code* or *Z-order curve*, which describes the desired locational code L.

5.3 Binary construction of a locational code for an AABB

The locational code L_S of the smallest cell C enclosing an axis-aligned subspace

$$S \equiv\ [x_l, x_h]\ \times\ [y_l, y_h]\ \times\ [z_l, z_h]\ \ \subset C$$

is obtained by computing the locational codes L_l for the point (x_l, y_l, z_l) and L_h for (x_h, y_h, z_h). L_S is the longest common prefix of L_l and L_h. This means that if ith bit is the first (the most significant) that differs in L_l and in L_h, then (x_l, y_l, z_l) and (x_h, y_h, z_h) lie on different sides of the ith cutting plane, while they lie on the same side of all the previous ones. Therefore, the length of L_S is $(i - 1)$ bits.

The whole operation of computing the locational code for an AABB can be done with an assembly code containing only one branching instruction [ref to code] on architectures featuring a bit-search instruction (such as *bsr* on x86 and x86_64 [ref to intel manual]). This leads to an extremely efficient CPU pipelining for that operation [ref].

As seen before, this result is especially useful since the objects we want to place in the kd-tree are axis-aligned bounding boxes (AABBs) enclosing our elements.

Note that to get an octree structure instead of a kd-tree, we would simply have to truncate the length of all locational codes to the nearest inferior integer multiple of three. However this would not exactly give a linear octree as described in the literature (see Section 5.1).

5.4 Properties of a sorted linear kd-tree

Once the construction of our array is completed, the next stage is to sort it by locational codes. We define the following *order* on them:

Let L_1 and L_2 be two locational codes. L_1 is *before* L_2 if L_1 is a prefix of L_2 or if the first (most significant) bit that differs between the two is "0" in L_1 and "1" in L_2.

Given this ordering, we have the following properties on the sorted array:

Property 2. *Elements placed in the same node are consecutive in the array.*

Property 3. *Elements placed in the subtree below a given node immediately follow in the array the elements placed in that node.*

Property 4. *A sweep of the array from its first entry to its last is equivalent to a listing of elements during a pre-order (or "depth-first") traversal of the corresponding kd-tree.*

These properties lead to a simple algorithm for selecting candidate pairs:

5.5 Sweep algorithm

```
**for** $i = \{0, 1, \ldots, N-2\}$ {
    $j = i + 1$
    **while** $L(i)$ is a prefix of $L(j)$ {
        submit candidate interaction pair $(i, j)$
        $j = j + 1$
    }
}
```

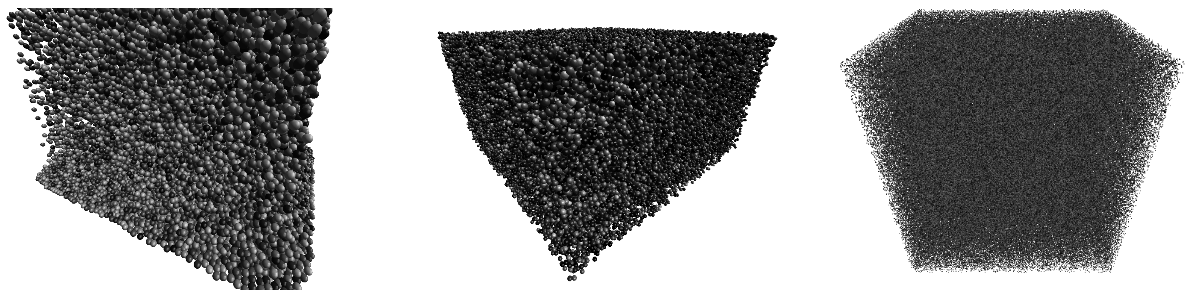
The "is a prefix of" test on the locational codes is the binary counterpart of the "is an ancestor of" test on the nodes of the tree. Thus it can be seen that this algorithm is a direct translation of the tree traversal method described in Section 2.

6 Results

Our benchmarks were based on the experiment described in Section 4.2. We still have N spheric elements initially placed randomly according to a uniform distribution such that

$$d \approx \frac{\text{total volume of elements}}{\text{volume spanned by initial distribution}}$$

In our tests, we used $d = 0.1$ (Figure 9).

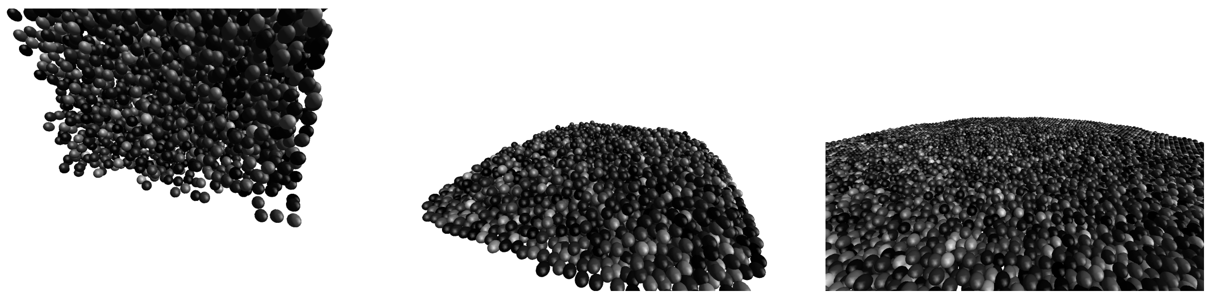


$N = 10^5$ $N = 10^5$ $N = 10^6$

Figure 9: Visualization of simulations at an early stage

To approximate real-life simulation conditions, we applied a constant downward acceleration on each element, and added a lower constraint preventing elements from shifting downwards indefinitely. Non-interpenetration is enforced pairwise (not globally) at each time step. This means that after each time step integration, we generate a list of interactions, and handle them pairwise, possibly generating more unhandled interactions. This scheme is not realistic, but it achieves our target of generating correlated random distributions of elements exhibiting some degree of time-coherence.

This setup resulted in elements getting more and more packed, and thus the number of interactions increasing with time, until some stable configuration is reached (Figure 10).



after one time step after 50 time steps after 750 time steps

Figure 10: Visualization of a simulation at different stages ($N = 10^4$)

The simulation was performed successively with four different algorithms

- A *quadratic* method effectively using no broad phase algorithm and testing each element against each other.
- A *Sweep & Prune* implementation based on V-Collide [3] and modified to handle spheric elements only.
- A *Delaunay triangulation* implementation using the CGAL library [2].
- Our implementation of *linear kd-trees* [10].

We measured average CPU timings over 11 time steps and 4 runs of the same simulation. The results are shown in Figure 11, Figure 12 and Table 2.

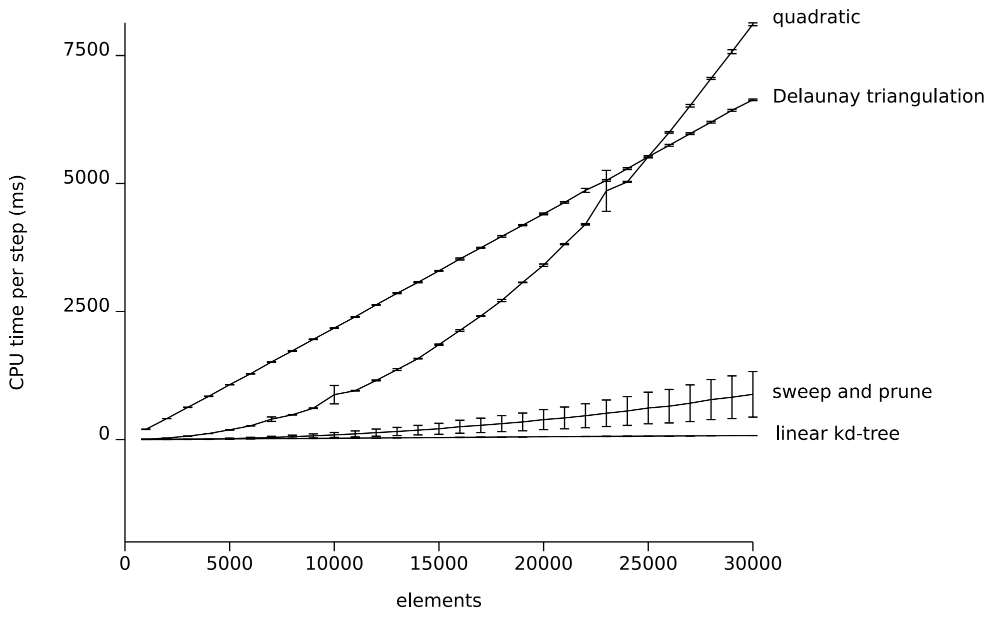


Figure 11: Average time and standard deviation on 4 runs and 11 steps (0-10), on an Intel E2180 at 2GHz

Figures 11 and 12 describe results for simulations with a small number of elements N. These are included mainly for comparison between existing alternatives, as the timings for our kd-tree-based algorithm do not appear distinctively on them. On Figure 11, we focused on the first 11 time steps after initialization. In that context, we had relatively few interactions taking place, and thus rather low dynamic behaviour (or high temporal coherence). Conversely, on Figure 12, at a later state of the simulation, we have a highly dynamic situation.

These figures highlight the seemingly linear (in fact loglinear) behaviour of Delaunay triangulation in terms of N, the quadratic order of the naive approach, and their similar efficiency at the two different points of the simulation.

The conclusions are quite different for Sweep & Prune. While this method exhibits good results for small N, especially on the first few time steps, its efficiency decreases with increasing N and

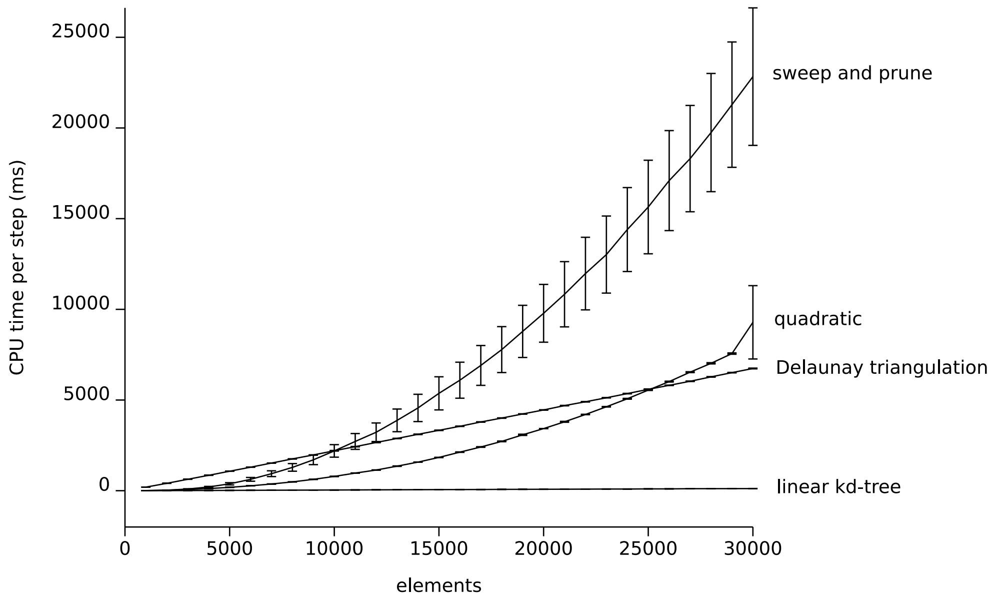


Figure 12: average time and standard deviation on 4 runs and 11 steps (50-60), on an Intel E2180 at 2GHz

simulation time. What is not shown on these figures is that for extremely low N, or more static simulations, Sweep & Prune dramatically outperforms all other methods. This is why its use is so widespread in haptics and virtual reality [3]. However, as discussed in [9, pages 47-49], its heavy reliance on temporal coherence makes it less fit for more dynamic simulations.

The performance of our algorithm based on linear kd-trees [10] is shown more explicitly on Table 2. We can see that the CPU time needed for computing a time step with our method is much smaller than with other methods. There are two main reasons for this. First, the computation of locational codes consists only in a few binary operations which execute extremely fast on general-purpose computers. Secondly, the sort is performed on a simple one-dimensional list of integers, which is both more CPU- and cache-friendly than larger structures. This sort stage is also the most time-consuming part, both from a practical and a theoretical point-of-view, needing $O(N \log N)$

	CPU time per step (ms)				Candidate pairs			
N	quadr.	S&P	Delaunay	**kd-tree-1**	quadr.	S&P	Delaunay	**kd-tree-1**
$50 . 10^3$	23818	64662	11350	211	$1.25 . 10^9$	471044	365396	$3.30 . 10^6$
$100 . 10^3$	94813	266840	23616	434	$5.00 . 10^9$	972680	732000	$5.84 . 10^6$
$500 . 10^3$	/	/	118408	2261	$125 . 10^9$	/	$3.67 . 10^6$	$35.8 . 10^6$
$1 . 10^6$	/	/	240563	4194	$500 . 10^9$	/	$7.34 . 10^6$	$78.9 . 10^6$
$5 . 10^6$	/	/	/	18885	$12.5 . 10^{12}$	/	/	$518 . 10^6$

Table 2: Performance comparison for large N (average on 4 runs, steps 50-60) on an Intel E2180 at 2GHz

	CPU time per step (ms)		Candidate pairs		Resident memory (approx.)		
N	kd-tree-1	kd-tree-2	kd-tree-1	kd-tree-2	Delaunay	kd-tree-1	kd-tree-2
$50 \,.\, 10^3$	211	297	$3.30 \,.\, 10^6$	277806	32 Mb	9 Mb	18 Mb
$100 \,.\, 10^3$	434	620	$5.84 \,.\, 10^6$	565180	76 Mb	17 Mb	35 Mb
$500 \,.\, 10^3$	2261	3288	$35.8 \,.\, 10^6$	$3.01 \,.\, 10^6$	312 Mb	82 Mb	193 Mb
$1 \,.\, 10^6$	4194	6153	$78.9 \,.\, 10^6$	$6.01 \,.\, 10^6$	623 Mb	144 Mb	353 Mb
$5 \,.\, 10^6$	18885	24205	$518 \,.\, 10^6$	$30.4 \,.\, 10^6$	/	558 Mb	1.5 Gb

Table 3: Performance (average on 4 runs, steps 50-60) and memory footprint on an Intel E2180 at 2GHz

operations.

It is however crucial to mention the main drawback of our first kd-tree algorithm (kd-tree-1): Any candidate interaction pair may be selected up to eight times. While this was not a problem in our benchmark, this is not desirable for most simulation models. In that situation, it is necessary to generate a list of unique interaction pairs. Such a list may be generated in linear time (more precisely, in $O(N\,c)$ operations, where c is the average number of interactions per element), but also requires $O(N\,c)$ additional memory, which might be prohibitive in some cases (kd-tree-2, Table 3).

Also, as we can see in the right-hand part of Table 2, the selection of candidate pairs is much less tight with our kd-tree than with the other algorithms. Although this did not penalize our benchmark results, this could be a problem if rejecting false positives were more expensive. Thus, for our method to be efficient in practice, we need our narrow-phase tests to be simple, *or* we need to have an efficient intermediate phase performing fast rejection tests (e.g. based on bounding spheres, as used in kd-tree-2). In case the specific geometry of our simulation does not allow us such efficient false-positives handling, our algorithm might be unsuited for collision culling.

7 Future directions

Although our comparative results may be questioned due to the necessary choice of a particular implementation for each alternative, our raw performance achievements open some promising possibilities for large-scale simulations. It is clear however that our method will shortly need extensions.

7.1 Exploiting temporal coherence

The first limitation of our algorithm is its inability to take advantage of temporal coherence. Simply stated, it would be convenient to find a way to not have to perform a full $O(N \log N)$ sort on the list of locational codes. Some other methods (e.g. sort and sweep [1]) do this by using special sort algorithms which are almost $O(N)$ for mostly-sorted lists such as the insertion sort [8], re-using the same list through different time steps. But the very structure of our algorithm is inherently unsuited for such an approach. More precisely, the fact that we preform element splitting before computing and sorting locational codes implies that each initial element corresponds to a variable number of sub-element locational codes in the sorted list. It is therefore not obvious to achieve temporal coherence between two consecutive lists, even if the underlying physical system shows little motion.

A foreseeable way to take temporal coherence into account would be to force our element-splitter to always give its maximal number of sub-elements (i.e. eight), possibly duplicating some sub-elements. It is however important to note that any such technique would tend to penalize simulations in a highly dynamic (i.e. low temporal coherence) environment.

7.2 Parallel processing

Targeting large-scale simulations, distributed computing is a necessary feature for practical use. Our first implementation does not currently allow it (nor even multithreaded operation).

However, both the locational code construction stage and the sort stage can be parallelized easily. The first one consists in a lot of small independent tasks. The second one is well covered in the literature [11].

The sweep stage can also be performed in parallel (different processes starting at different places in the list), but it is then necessary to take special care of data synchronization, especially if the memory is not shared by the computing resources.

7.3 Geometric scale and anisotropy

Two important parameters were left unexplained in the previous sections: the size of the interest space (the root node cell), and the maximal bit length of locational codes ($3B$). While the first one is generally believed to be problem-related, and the second one more hardware-related (typical implementations will use 32 or 64 bits integers to represent nodes), these parameters directly affect the geometry of all cells. More specifically, they should be chosen to match as much as possible the scale of the simulation, and the relative scale of axis (if not identical). But for this, a more rigorous study of anisotropy and ideal leaf node size is necessary, for each particular simulation model.

7.4 Conclusions

We have developed an efficient method for accelerating the computation of limited-range interactions in highly dynamic, large-scale simulations. We have demonstrated its usefulness with simplified benchmarks.

Due to inherent limitations, its range of application may not cover all particle simulations. However, when suited for a given problem, we have shown it to perform more than fifty times faster than its nearest competitors.

Most promising applications of such an algorithm include any three-dimensional simulation method involving limited-range potentials, such as discrete elements methods (DEM) and smoothed particle hydrodynamics (SPH).

The last remaining step before we can get a first working simulation infrastructure is to parallelize the implementation of the process and make it ready for cluster computing.

References

[1] David Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies.* PhD thesis, Cornell University, Ithaca, NY, 1992.

[2] CGAL. Computational geometry algorithms library. `http://www.cgal.org`, 2009.

[3] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav K. Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 189–196, 1995.

[4] Christer Ericson. *Real-Time Collision Detection.* The Morgan Kaufman series in interactive 3D technology. Morgan Kaufmann, 2005.

[5] Jean-Albert Ferrez. *Dynamic Triangulations for Efficient 3D Simulation of Granular Materials.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2001.

[6] Florian Fleissner and Peter Eberhard. Parallel load-balanced simulation for short-range interaction particle methods with hierarchical particle grouping based on orthogonal recursive bisection. *International Journal for Numerical Methods in Engineering*, 74:531–553, 2007.

[7] C.F. Li, Y.T. Feng, and D.R.J. Owen. Smb: Collision detection based on temporal coherence. *Computer Methods in Applied Mechanics and Engineering*, 195:2252–2269, 2005.

[8] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures.* Springer Berlin Heidelberg, 2008.

[9] Brian Vincent Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems.* PhD thesis, University of California, Berkeley, 1996.

[10] Laurent Poirrier. Linear kd-tree library and benchmarks. `http://www.montefiore.ulg.ac.be/~poirrier/particle`, 2009.

[11] Douglas R. Smith. Derivation of parallel sorting algorithms. In *Parallel Algorithm Derivation and Program Transformation.* Springer US, 1993.