DOI: 10.1111/cgf.13529

COMPUTER GRAPHICS *forum*
Volume 00 (2018), number 0 pp. 1–14

Flexible Use of Temporal and Spatial Reasoning for Fast and Scalable CPU Broad-Phase Collision Detection Using KD-Trees

Ygor Rebouças Serpa and Maria Andréia Formico Rodrigues

Programa de Pós-Graduação em Informática Aplicada (PPGIA), Universidade de Fortaleza (UNIFOR), Brazil
{ygor.reboucas, andreia.formico}@gmail.com

Abstract

Realistic computer simulations of physical elements such as rigid and deformable bodies, particles and fractures are commonplace in the modern world. In these simulations, the broad-phase collision detection plays an important role in ensuring that simulations can scale with the number of objects. In these applications, several degrees of motion coherency, distinct spatial distributions and different types of objects exist; however, few attempts have been made at a generally applicable solution for their broad phase. In this regard, this work presents a novel broad-phase collision detection algorithm based upon a hybrid SIMD optimized KD-Tree and sweep-and-prune, aimed at general applicability. Our solution is optimized for several objects distributions, degrees of motion coherence and varying object sizes. These features are made possible by an efficient and idempotent two-step tree optimization algorithm and by selectively enabling coherency optimizations. We have tested our solution under varying scenario setups and compared it to other solutions available in the literature and industry, up to a million simulated objects. The results show that our solution is competitive, with average performance values two to three times better than those achieved by other state-of-the-art AABB-based CPU solutions.

Keywords: collision detection, animation

ACM CCS: Computer Graphics [Computing Methodologies]: Animation-Collision detection

1. Introduction

Realistic computer animations rely on physics simulations to generate plausible motions and effects. At its core, the collision detection (CD) module is an essential component to the illusion that digital objects are solid. On the other hand, it is one of the most time-consuming stages of the pipeline. Due to the complexity of designing such systems, applications often rely on physics libraries, such as the *Bullet Physics* [Cou17] or the *PhysX* [NVi17], to implement both the CD and response steps.

Poor or absent CD makes for notoriously unrealistic simulations. Missed or false collisions are readily visible to users. In a dual manner, the CD module has to be fast enough to produce high frame rates, which, in turn, are key elements for achieving visual realism. Moreover, current simulations steadily grow in the number of simulated objects and shape complexity, while computational resources are squished even more towards other components, such as the data related to logic and visual representations. To cope with these requirements, the CD is architecturally divided into two sequential phases: the broad and narrow phases [Eri04]. The former works globally and inaccurately, selecting the most likely colliding object pairs over all possible N^2 pairs, whereas the latter acts locally and accurately, properly testing each selected pair for collisions, yielding the final result. Traditionally, the broad phase performs the CD between bounding volumes only [DLAG12], which despite being inaccurate provides perfect recall, whereas the narrow phase acts pair-by-pair, considering the actual objects' geometries. While the narrow phase can differ considerably from simulation to simulation, when dealing with different types of objects (e.g. rigid and deformable bodies, particles, etc.), the broad-phase implementation does not differ significantly. Actually, the broad phase only depends on the choice of bounding volumes, how many of them there are, their sizes and their motion.

Broad-phase algorithms approach the inaccurate global intersection detection by means of shape simplification (as previously mentioned), spatial reasoning and temporal reasoning. Altogether, these three techniques allow for fast proximity detection of objects, outputting the list of object pairs that are close enough to be likely colliding. The first technique consists of completely bounding

each object with simple geometric primitives that provide fast intersection checking. If a pair of bounding volumes does not intersect, the pair can be safely discarded. Commonly, Axis-Aligned Bounding Boxes (AABBs) are the primitive of choice, but several others have also been used, such as spheres, Oriented Bounding Boxes and K-Discrete Oriented Polytopes [Eri04]. While the shape simplification acts locally for each object, the spatial reasoning acts globally, reasoning over the distribution of objects and identifying relationships among them, such as clusters, separating axes and order. Often, this is approached by either using space partitioning data structures, spatial sorting or hashing. Finally, the temporal reasoning resides in exploiting previous knowledge from the simulation and in anticipating its future behaviour, such as identifying static objects, reusing computations and structures from previous frames or even extrapolating trajectories to anticipate collisions. The more coherent a simulation is, the more temporal reasoning opportunities exist. In some applications, this can yield orders of magnitude faster executions [TB12].

Modern broad-phase algorithms employ all these three techniques to reach competitive performance. However, although very fast broad-phase algorithms exist, most solutions tend to excel in just a handful of specific scenarios, failing to deliver the promised performance on several others. Game-oriented solutions, for instance, often assume that most objects will be static, and thus, are optimized for coherency. However, incoherent behaviour may arise from the players actions and will lead to a severely reduced performance. An adaptive or general solution would provide a more consistent performance across several application domains, better handling these changes in behaviours. In another direction, several authors have explored the GPU for a faster broad phase [LHLK10, LLCC13, AGA12]. However, the latter approach is contrary to the generality argument, as it requires a programmable GPU to run, which might not be the case for low-end desktops, browsers and many mobile devices. Additionally, it is still questionable whether the GPU can outperform the CPU for mostly static scenes. In such scenarios, the workload is considerably sparse and barely arithmetic, which is undesirable for GPU devices. Some works exploit the temporal coherence on the GPU [LHLK10], but their improvements are not as competitive as those reported on CPU [TB12].

Main Results: we present a novel broad-phase CD method based on a KD-Tree designed to handle several objects distributions and degrees of motion coherence. The algorithm is based upon an SIMD-optimized KD-Tree and sweep-and-prune (SAP) [BW92] hybrid algorithm that is able to perform complete and incremental CD, respectively, handling coherent and incoherent setups with uniformly and non-uniformly sized objects. Additionally, our solution is also capable of self-adapting from incremental to complete CD on the fly, dealing with scenarios that may alternate behaviour from coherent to incoherent. The features previously mentioned are made possible by an efficient and idempotent two-step tree optimization algorithm based on a linear arrangement of objects. The complete solution was tested under varying scenario setups and compared to other solutions available in the literature and industry [Cou17, KMZ16, TBW09], using up to a million simulated objects. The results show that our solution (coloured black in Figure 1) is competitive from a low to high number of simulated objects, with average performance values two to three times better than those achieved by other state-of-the-art AABB-based CPU solutions.

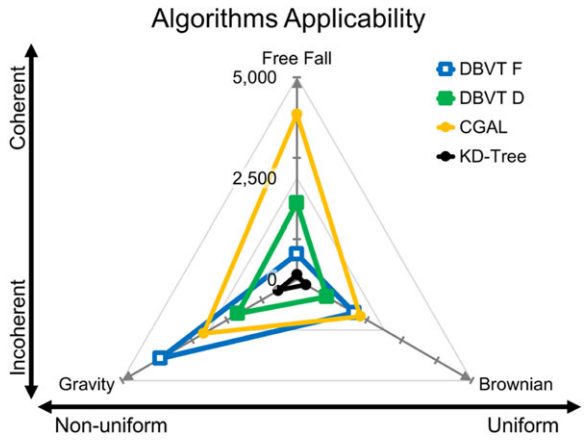


Figure 1: *Radar chart comparing the algorithms performance in three distinct scenarios (*Free Fall*,* Brownian *and* Gravity*), positioned according to their temporal coherency and object spatial distribution. An algorithm is said to be general or specific whether it performs consistently or inconsistently across all scenarios, respectively. Data collected from one million object simulations and time in milliseconds per frame. Our solution is coloured in black.*

2. Related Work

Over the years, the continuous effort to improve CD has led to several improvements to both algorithms and spatial data structures. Many data structures have been explored, such as Grids [AGA12, LLCC13], Bounding Volumes Hierarchies (BVHs) [Cou08, LGS*09], KD-Trees, Octrees and Binary Space Partitioning Trees (BSP-Trees) [Gla05, LCF05, SN15, Wal07] and sorted vectors [TBW09, DSC05]. However, few works have explored the applicability of their solutions beyond simple scenarios, and often provide only basic time statistics, such as the total execution time for an evenly distributed set of randomly moving objects. A deeper study of the algorithms behaviour under several setups is needed to assess their readiness for industry use.

The broad-phase solutions applicability is directly related to their use of spatial reasoning: grids excel when all objects are of similar size and are uniformly distributed [LLCC13]; tree-based algorithms can better handle differently sized objects and non-uniform distributions, but often have to trade partitioning flexibility for an easier build and balancing (two extremes are the BSP and the Octree) [LCF05] and algorithms based on sorting, such as the well-known SAP [BW92], are often either made for highly coherent scenarios [TBW09] or completely ignore coherency [ZE00]. By itself, the SAP algorithm does not scale well [TBW09]; however, it is often used in conjunction with other methods [LHLK10, ZE00, TBW09], as it can be used in-place. Moreover, many authors have attempted to further improve the execution speed by developing GPU algorithms [LHLK10, LLCC13, AGA12]. However, these are often grid-based and only applicable to modern desktop computers, excluding several potential devices, such as mobile phones. Regarding CPU parallel solutions, we refer to the survey by Dodier-Lazaro, Avril and Gouraton [AGA11].

For an algorithm to be equally applicable to a variety of scenarios, it has to be indifferent to any scene property. Consequently, its time complexity must only depend on the size of its inputs. Such an algorithm exists and is known as the brute force method. It employs no coherency assumption or favours any given object distribution over another; however, it is unusable in practice, due to its quadratic time complexity. Actually, the number of colliding pairs is often much closer to N than to N^2. The work of Weller *et al.* [WFZ13, WDZ17] develops from that notion, presenting novel ways to enumerate collisions between sphere packs and between triangle meshes in linear time. However, it is left open how the authors theories could be applied to the N-bodies case.

We assume that to be widely applicable and efficient, an algorithm has to self-adapt to the workload. For that, its spatial reasoning should be flexible and its temporal reasoning be controllable, so that it can act as a specialized solution for a wide range of possible applications. In view of this, we propose a novel broad-phase CD algorithm based on the KD-Tree, the SAP algorithm and the incremental CD, focusing on flexibility and adaptiveness. Results show that our solution can be efficiently updated and queried for both coherent and incoherent scenarios, as well as on uniform and non-uniform distributions with equal or different object sizes.

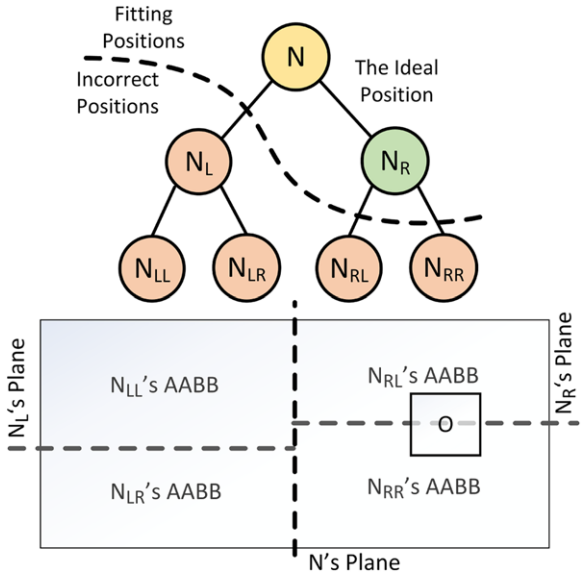


Figure 2: *The ideal, fitting and incorrect positions of an object 'o'.*

3. KD-Tree-Based Broad Phase

In general terms, our approach consists of a variable depth KD-Tree that automatically grows or shrinks and self-tunes to improve its partitioning. The execution is divided into two parts: the tree update and CD. The former is responsible for all tree manipulation and optimization tasks, while the latter is responsible for finding all colliding pairs of objects within it. The complete solution is governed by the following three parameters: T, which controls the tree growth; ε, which controls the static/dynamic object classification and K, which controls the adaptivity mechanism. In our algorithm, objects are simplified by AABBs, as it is a standard practice for broad-phase algorithms.

Throughout the entire text, by referring to KD-Tree, we refer to a strictly binary tree in which nodes have a plane, an AABB and a list of objects. The plane of a node is an axis aligned plane that subdivides the space into two halves, its AABB is the region delimited by all the planes of the node's ancestors and finally, the list of objects is the list of all objects within the node. While leaf nodes do not have planes, the root node has an arbitrarily large AABB, since it has no ancestors to delimit it. By subdividing the set of objects into groups, the tree provides enough information to assert that no collisions exist between objects of different tree branches. Pairs that can be discarded are referred to as *pruned pairs*, whereas pairs that have to be tested for intersection are considered as *candidate pairs*, as there is not enough information to prune them. Implementation-wise, the nodes AABBs can either be stored explicitly or recovered during the tree traversals, we opted for the caching approach, as it simplifies the implementation.

Given the detailed structure, we define an *object arrangement* as being the positioning of all objects within the structure. For any such object, we define its *ideal position* as the deepest node that fully contains it within its AABB. Additionally, an object is said to be in an *incorrect position* when the node that houses it does not fully contain it. Finally, an object is said to be in a *fitting position* when its position is neither ideal nor incorrect, i.e. the node fits it completely, but it is not the deepest one. These three concepts are shown in Figure 2. With regard to the tree itself, we define its *topology* as being the set of nodes, planes and AABBs that it is composed of. For CD purposes, all objects need to be, at least, in a fitting position. However, to reach the lower bound of the candidate pair generation, all objects must be housed within their ideal nodes, characterizing an *optimized arrangement*. Additionally, the more evenly distributed the objects are across leaf nodes, the more pairs are pruned. Thus, the topology plays a key role in pruning pairs. Finally, the subtree *population* is the total number of existing objects in all of its nodes.

Taking into consideration the previous description and terminology, the following subsections describe our CD process, the two-phase tree update algorithm, the use of SAP and SIMD instructions, the use of temporal reasoning and the adaptivity mechanism.

3.1. Collision detection

Given a tree (as previously described) and an optimized object arrangement over it, the CD process can be performed top-down by identifying all pairs that cannot be pruned by the structure. From top to bottom, each node generates two types of candidate pairs: internal and external. The former is the set of all possible pairs among the node's objects and the latter is the set of all pairs formed by the node's objects (internal objects) and objects that intercept the node AABB (external objects).

While internal pairs are trivial to generate, external pairs require an algorithm to identify all the external objects of a node. The key idea behind this algorithm is that, given a node and its external objects, we can easily identify which of these objects are also external to the node's left and right children. Firstly, all objects of a node are external to the node children, as these objects lie on the

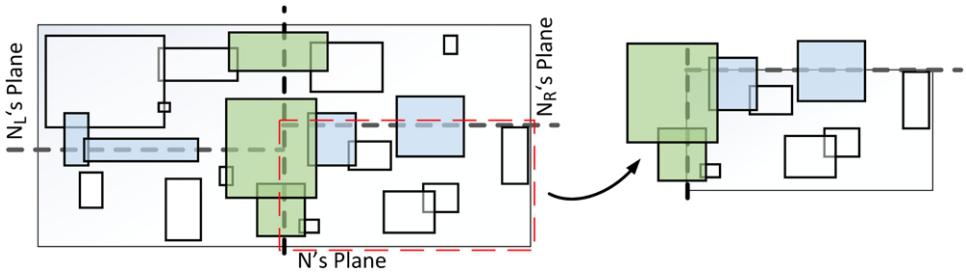


Figure 3: *On the left and right, respectively, a scene with several objects, and the internal and external objects of the N_{RR} node. Note that the topmost object of the root node was not considered external to the N_{RR}'s AABB, as it is not completely/partly to the right of the N_R's plane.*

splitting plane of the node. Secondly, all external objects that are completely/partly to the left of the node plane are also external to the left child, and, analogously, all external objects completely/partly to the right are also external to the right child. This relationship is exemplified in Figure 3. Finally, as the root node AABB is infinite, no object can be external to it. Using all the previously mentioned properties, it is possible to recursively enumerate the external objects of the nodes using a regular top-down traversal, and thus, generate all internal and external pairs.

3.2. Tree update

The goal of the tree update algorithm is to turn any previous topology and arrangement into a more efficient topology with an optimized object arrangement, to maximize the amount of pruned pairs. This process is constrained by a single parameter, named T, which limits how many objects there may be in a leaf node and the minimum population for an internal node. When this constraint is violated, the node is either subdivided (in case it is a leaf node) or collapsed (in case it is an internal node). This constraint alone is able to build an initial topology for which the update algorithm will act. The parameter T, in contrast to fixing the height to a specific constant, enables the tree to grow/shrink as needed and to be tuned without a specific number of objects in mind. Thus, the algorithm can be run unchanged for several settings of numbers of objects, as well as adapting seamlessly to scenarios that may add/remove a significant amount of objects at runtime.

To efficiently manipulate the objects in memory, the algorithm employs a single array to store all objects' references following an in-order traversal, as shown in Figure 4. In more detail, the list of objects of each node is a contiguous block of object references which are arranged following an in-order traversal. Thus, instead of each node storing its own separate list, they all share the same array. To keep track of its object block, each node stores its starting and ending indices in the shared array. The object blocks of a subtree will appear in a sequential order in memory when using the in-order layout. This scheme greatly optimizes subtree traversal operations, such as enumerating its objects or collapsing them into a single node. This layout is made possible by the design choice of having objects on both internal and leaf nodes. This choice removes any object redundancies on the structure, which allows for a fixed size array, despite the underlying tree topology. The drawback of using this representation (as shown in Figure 5) is that moving a set of objects from a node to another requires that the in-between

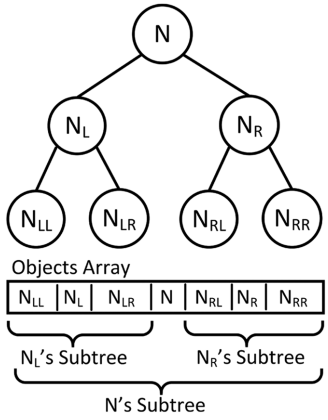


Figure 4: *Object array memory layout (in-order traversal).*

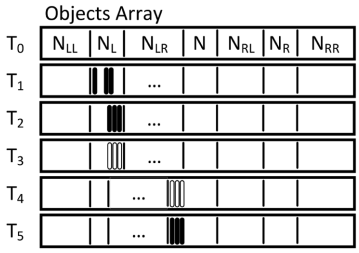


Figure 5: *Example of transferring three objects from N_L to N. Step-by-step, the objects are grouped at the end of the objects block (T_2), copied to an auxiliary buffer (T_3), the intermediary objects are moved (T_4) and the objects are placed at the destination (T_5).*

objects be moved as well. A similar arrangement of objects outlining a space partitioning structure can be found in the work of Mora [Mor11]. However, in contrast to the mentioned work, we still store the structure in memory.

The updated algorithm is subdivided into two phases: the refit and optimization. Both phases traverse the tree only once. The former does it bottom-up, moving unfitting positioned objects to fitting positions and the latter traverses the tree top-down, optimizing the topology and moving objects towards their ideal positions. To mitigate the cost of updating the begin and end indices of nodes

whenever objects are moved, we employ a correction term. When moving x objects from the node N_L to the node N, we need to adjust all indices of the N_{LR} subtree by $-x$ positions. Therefore, we store $-x$ as a correction term of N_{LR}, leaving all N_{LR} indices in an inconsistent state. Later on, during optimization, when N_{LR} is visited, the algorithm applies the correction term to the node's indices and propagates it to the node's children. Thus, indices are left inconsistent during the refit, but are corrected, during optimization.

Refit: Bottom-up, for each node, we traverse its objects list for incorrectly positioned objects, moving them to their parent node. Gradually, all objects reach a fitting position, which in the worst case is the root node.

Optimization: Top-down, for each node, a set of operators is used to improve and optimize the arrangement and topology, altering the cutting planes when necessary and moving objects downwards from fitting positions to their ideal positions. The complete process uses six distinct operators: (1) *Filter*, responsible for filtering and moving objects downwards to improve their positioning; (2) *Partition*, in charge of branching leaf nodes; (3) *Collapse*, in charge of collapsing subtrees back into leaf nodes; (4) *Evaluate*, tasked with evaluating how appropriate the planes of the nodes are; (5) *Translate*, the plane optimization operator and (6) *Erase*, responsible for deleting a node and one of its branches in favour of the other. Combined, these operators carry out all the remaining tasks of the update phase. A graphical representation of these operators is shown in Figure 6.

The execution starts by pushing the root node into a stack. From this point on, the algorithm repeats until the stack is empty (*main loop*), popping the first node of the stack and performing either of the following actions based on the node type:

- **Leaf node:** If the node has more than T objects, it triggers the *Partition* operator and then pushes the recently created nodes into the stack; otherwise, nothing happens, returning to the main loop.
- **Internal node:** Initially, the node population is queried for the *Collapse* operator. If the subtree has less than T objects, it is collapsed and the algorithm returns to the main loop; otherwise, the execution continues by applying the *Filter* and *Evaluate* operators. If the evaluation is positive, no further work is needed for this node, and if it is negative, the *Translate* operator is applied. To validate if this operator was successful in improving the evaluation, the *Filter* and *Evaluate* operators are reapplied. If the evaluation is now positive, the operator succeeded, else, the *Erase* operator is applied. At the end of this procedure, the node siblings are pushed into the stack.

This algorithm is shown as a flow chart in Figure 7. At first sight, the algorithm previously described can be seen as conservative, since it attempts to improve the structure by simple means, applying the *Filter*, *Partition*, *Collapse* and *Translate* operators. However, when all these operators fail, i.e. a node is negatively evaluated twice, the algorithm goes on a more drastic approach, discarding the node and its least populated branch. In sequence, the process is repeated for each remaining child node, which may end up being erased as well. This conservative/aggressive duality performs little to no update at all for coherent/static simulations, but also quickly throws away most of its structure, if need be (e.g. during incoherent/dynamic situations). Moreover, the algorithm can quickly build an entire subtree from scratch, by applying the *Partition* operator to a leaf node and then pushing the new nodes to the stack, allowing them to be partitioned. Thus, our structure is lazily constructed [HBHS05]. Hence, the algorithm seamlessly merges conservative, aggressive and rebuilding strategies into a single one-pass execution that is deterministic and produces an optimized tree. In the following, we provide more detail on these operators.

Filter: This operator (Figure 6a) receives an internal node as input and filters it to separate those that lie completely to the left of the plane, exactly on the plane and completely to the right (these objects have to be moved, respectively, to the left branch, stay on the node and be moved to the right branch). In addition, during its action, we update the nodes AABBs and population counts, as well as the correction of its indices using the delayed offset.

Partition and Collapse: These operators (Figure 6b) control the size of the tree, increasing it when leaves are too crowded, or decreasing it when subtrees are underpopulated. Both operators rely on the T parameter as its threshold for action. When triggered, the *Partition* gathers the objects mean and variance statistics, positions the new plane at the mean of the axis with greatest variance and creates the two new child nodes. Finally, it applies the *Filter* operator to distribute the objects and update the nodes AABBs, population counts and indices. Inversely, when triggered, the *Collapse* operator moves all objects from the subtree back to the node, erases both left and right subtrees, deletes the cutting plane and updates its indices to reflect the acquired objects.

Evaluate: This operator is the key decision-making process of our algorithm. The goal is to evaluate how proper the cutting plane is by calculating two measures: *Cost* and *Balance*. The *Cost* metric is an estimate that assumes the worst-case scenario of $\left(\frac{n}{2}\right)$ tests for n objects. Given the subtree rooted in N, we have n objects at the node, p_L objects at the left subtree population and p_R objects at the right subtree population. In total, each group has to generate its internal pairs and external pairs. Assuming N has no external objects, we obtain the total number of tests t, as follows:

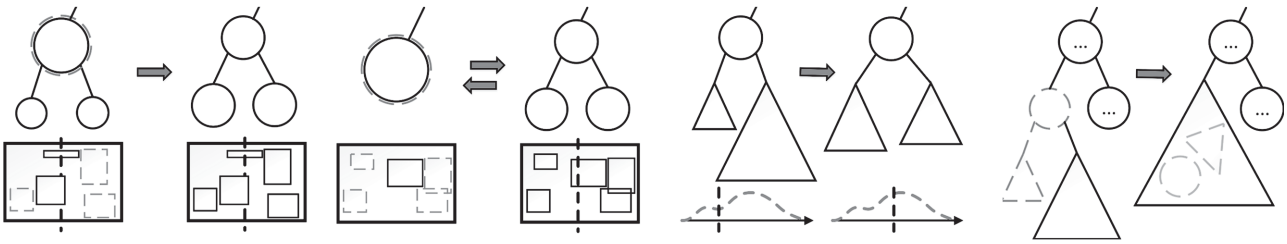
$$t = \left(\frac{n}{2}\right) + \left(\frac{p_L}{2}\right) + \left(\frac{p_R}{2}\right) + n(p_L + p_R). \tag{1}$$

The worst-case scenario for Equation (1) is when all objects are together, i.e. $t_{\max} = \frac{n+p_L+p_R}{2}$, and the best case is when there is a perfect 50/50 split, that is, $t_{\min} = 2 * \frac{(n+p_L+p_R)/2}{2}$. Thus, we can define our final cost measure by normalizing the expected number of tests t with its lower and upper bounds, as shown in Equation (2):

$$Cost = \frac{t - t_{\min}}{t_{\max} - t_{\min}} \quad , \quad t_{\max} \neq t_{\min}. \tag{2}$$

Contrastingly, the *Balance* metric can be simply computed as the difference between the lowest populated subtree population to the rest of the node population. Formally, it is defined in Equation (3) as follows:

$$Balance = \frac{\min(p_L, p_R)}{n + \max(p_L, p_R)}. \tag{3}$$



(a) Filter. **(b)** Partition and Collapse. **(c)** Translate. **(d)** Erase.

Figure 6: *Operators used during the tree optimization step.*

If $Cost > Balance$, the node is negatively evaluated. In general terms, the deeper the node is in the tree, the closer its *Cost* will be to the actual number of generated object pairs. Therefore, this operator tends to evaluate more negatively the nodes closer to the root, that is, this metric focuses on performing large operations up the tree and small adjustments down.

Translate: This operator (Figure 6c) optimizes the plane positioning of a given node by shifting it on its current axis towards the new objects mean. To perform this action, the algorithm pulls all objects from the subtree to the node, computes the new mean, translates the plane and then applies the *Filter* operator. While pulling all objects up may seem excessive, it avoids having to redo the refit phase for the subtree. Additionally, the *Filter* operator repositions all objects on-the-go and also propagates changes to the node AABBs. Moreover, after being pulled up, all objects will go down the tree at the same pace as the algorithm. Thus, further *Translates* or *Erases* will have all objects at nearby nodes, optimizing their execution. Finally, the use of the array representation simplifies the operation of unifying the whole population to a set of index and delayed offset operations. Additionally, there are no intermediate objects after this point during the whole top-down traversal of the subtree, avoiding memory shift operations.

Erase: Whenever the *Translate* operator is not sufficient to improve the node evaluation, this operator (Figure 6d) is called. The *Erase* operator removes the current node and its less populated branch from the tree, and inserts their content on the more populated branch. This effectively removes an entire branch of the tree and leaves the remaining branch to continue the effort of optimizing the topology.

Finally, the insertion and removal of objects can be easily done and has only a marginal impact on the algorithm's performance: any object can be inserted at the end of the objects vector (rightmost node of the tree), and any object can be removed from the structure by swapping it with the last element of the vector. During the update phase, the inserted or swapped object will be moved to its ideal positions.

3.3. Sweep-and-prune and SIMD

The CD using the KD-Tree alone is able to prune away most of the search space. However, there is a limit to what the tree can do. The taller the tree, the more branches it has, and thus, more pairs can be pruned. This is efficient up to the point that the AABBs of leaf nodes are as small as the objects themselves. At this point, the overhead of such a structure takes over as the bottleneck. To handle this, a second level of pruning that is not based on space partitioning is required. In this regard, we use the SAP algorithm.

The SAP algorithm [TBW09, BW92, ZE00] projects all objects into an axis and sorts them by their minima. This allows for an efficient uni-dimensional pruning of collisions. Object pairs that do collide on the uni-dimensional case remain as candidate pairs. This algorithm can be easily merged with any space-partitioning algorithm by using it on each of the structure's partitions independently. In our implementation, the axes are selected using the objects variance and the projections are sorted using the quicksort algorithm. For external pairs, we use the bipartite version of the SAP algorithm, as described in [ZE00]. In essence, both lists are traversed at the same time and evaluated against each other.

When applying the KD-Tree and SAP algorithms sequentially, we obtain a set of candidate pairs to be tested. Such a test can be further optimized with the use of SIMD instructions. If w is the SIMD-width available to the CPU architecture, up to w AABB pairs can be tested simultaneously for collisions [Eri04]. On most modern hardware, both SSE and AVX instructions are present, allowing for four and eight pairs to be tested at once, respectively.

3.4. Coherency

The algorithm described thus far makes limited use of temporal reasoning: it updates the tree instead of rebuilding it. For pruning purposes, this property should also be accounted for. Thus, for coherent simulations, we propose a variant of our algorithm that searches for collisions *incrementally*, i.e. it only looks for pairs that no longer collide and for newly colliding pairs.

Any collision algorithm can be converted from a complete search to an incremental one by introducing two elements: a static/dynamic object classification function and a colliding pairs cache. The key idea is to store collisions among static objects and limit the search to dynamic–dynamic and dynamic–static pairs. In more detail, at the beginning of each frame, the cache has to be cleaned of any collisions involving dynamic objects. Then, the CD is performed as usual, but no static–static candidate pairs are tested. Within our algorithm, this takes place as an additional pruning step that filters out static–static pairs before testing.

For our solution, objects are classified as static or dynamic by augmenting their AABBs by a small constant ε, e.g. 0.01, which merely acts as a numerical margin. For each object, when updated, the received non-augmented AABB is tested against the object's current augmented AABB. If the received AABB lies completely inside the current augmented AABB, then it can be considered as static and the current AABB is kept. If not, the object is considered dynamic and the received AABB is augmented and used. This approach is conservative and will not miss any collisions at the expense that the added margin value may lead to false candidate pairs.

Although the incremental CD *per se* is not a novelty of this work, our approach differs from previous works in the classification method. The Dynamic Bounding Volumes Tree (DBVT) algorithm [Cou08] uses a BVH structure with one object *per* leaf; however, the leaf AABB is slightly larger than the object's AABB. As long as the object fits inside the leaf volume, it will not be updated nor tested for collisions, thus is considered static. The work described in [LHLK10] fully describes the incremental approach, but does not explain how objects are classified in the first place or even if this is done automatically by the algorithm. Additionally, physics libraries such as Bullet [Cou17] and PhysX [NVi17] employ a similar technique (sleeping bodies) to improve the system performance and stability for time integration and collision response tasks. This, however, does not apply to the CD. Moreover, these classification methods are often based on object properties that are not visible/relevant to the broad phase, such as the mass-normalized kinetic energy measurement. Our solution is fully automated and works at the AABB level.

This version of the algorithm will be referred to in this work as the *Incremental* approach, whereas the one previously described will be referred to as the *Complete* CD.

3.5. Adaptivity

The Complete and Incremental algorithms can be used for their specific optimization targets; however, some scenarios may alternate between coherent and incoherent. With this in mind, we propose the following simple adaptivity mechanism: compute the percentage of static objects within the simulation, if it is above a threshold K (defined empirically), then use the Incremental algorithm, otherwise use the Complete algorithm. For this to work, we need to have the collision cache and static/dynamic classification at all times, even when performing the Complete algorithm. This is the sole drawback of this approach to adaptivity. With regard to the collision cache, it should be emptied when using the Complete CD, so as to avoid duplicated entries.

3.6. The complete solution

The algorithm previously described is a three-stage pruning solution over an efficient KD-Tree structure. The pruning stages are: KD-Tree, SAP algorithm and Incremental approach. This pruning combination is the key element for our solution performance. Individually, the first two do not scale well: the KD-Tree alone needs many levels to achieve sufficient pruning and the SAP alone does not scale on large scenes, as evaluated by Tracy [TBW09]. However, when both are combined, it is feasible to use a much smaller tree paired with many small SAPs, a much more efficient and scalable alternative. In sequence, the Incremental CD pruning acts on static–static pairs, further reducing the number of candidate pairs. This step is crucial to the algorithm's competitiveness on coherent scenarios and is transparently turned on or off by the adaptivity mechanism. Finally, the remaining candidate pairs are tested for collisions in SIMD batches.

Regarding the structure's complexity, the tree height H is proportional to $\log(\frac{N}{T})$ (up to T objects *per* leaf) and occupies memory proportional to N (there is no object redundancy). Both refit and optimization steps, and thus the complete update phase, are proportional to N, in the best case, and to $N \cdot H$ in the worst case. For both, the best case is when no object/node requires action and the worst case is when a full rebuild is required. Individually, the operators run in either linear time (*Partition*, *Translate* and *Filter*) or constant time (*Collapse*, *Evaluate* and *Erase*). The key element to the update performance is the contiguous memory layout, which eases all subtree related operations, such as deleting subtrees or enumerating its objects, while also improving data locality dramatically.

While our work shares some resemblance to the work of Luque *et al.* [LCF05], we differ in several key aspects. The most important differences are the contiguous memory layout and the KD-Tree over the BSP-Tree. These changes allow most of the operators to run in constant or linear time, while avoiding any need for redundant object references. This greatly simplifies the algorithm and allows all optimizations to be carried out whenever possible (as opposed to scheduled). The scheduler is completely replaced by the refit and optimization steps, which are an idempotent algorithm to correct and streamline the entire structure. This has a deep influence on our algorithm's capacity to handle incoherent and large-scale scenarios as well as static ones. Finally, our solution requires only three easily configurable parameters, whereas the solution presented by Luque *et al.* [LCF05] uses several non-trivial parameters, which are considerably harder to tune.

4. Benchmark

To assess our solution performance and to contrast it with the state-of-the art CPU CD, we designed three distinct scenarios: *Free Fall*, a coherent simulation of free falling cubes; *Brownian*, an incoherent simulation of uniformly distributed objects moving in random directions and *Gravity*, an incoherent simulation of densely packed objects over a variable gravitational field. In essence, they explore different scenario properties. *Free Fall* is similar to what can be expected in games, a coherent motion with little to no external influence. *Brownian* is a commonly used benchmark problem: uniformly distributed randomly moving objects unaffected by gravity. Finally, *Gravity* employs a variable gravitational field that constantly stacks all objects over different regions of the application space. This scenario poses a great challenge to algorithms, since most of the application space is unused, many collisions occur and the motion is highly unpredictable. To investigate the objects' geometry impact on the algorithms performance, we execute these scenarios with equally sized objects (cubes) and with objects of varying sizes (bars, planks, squares and cubes). It makes little sense to increase the objects' complexity further (such as triangle meshes) because, at the broad-phase level, only bounding volumes are used. These variants

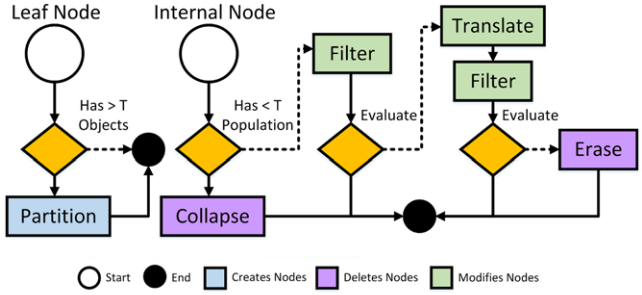
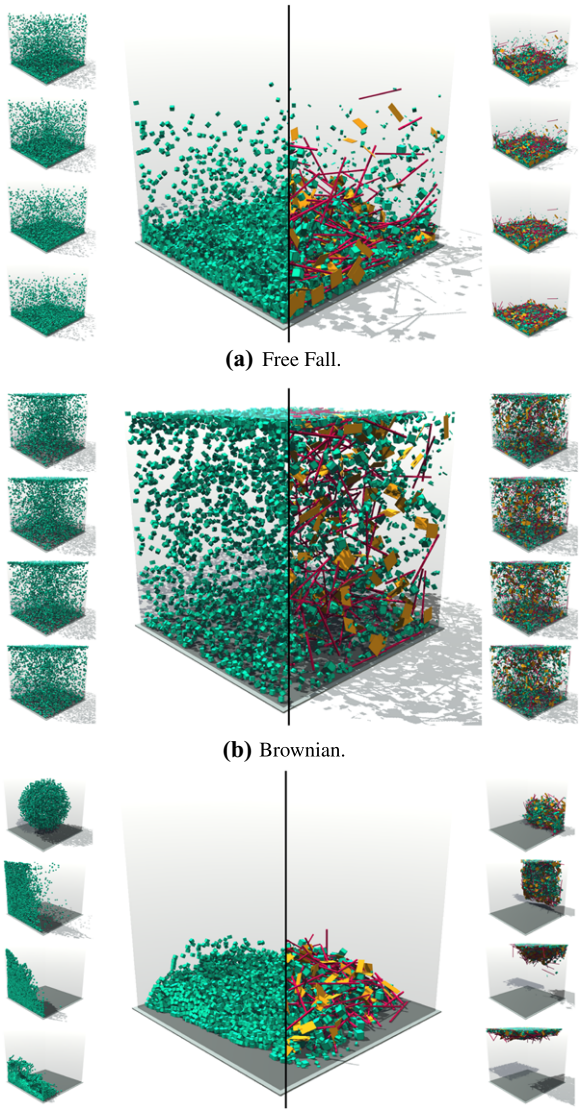


Figure 7: *Tree optimization algorithm as a flow chart.*

will be referred to as *Cubes* and *Assorted*. To better illustrate each scenario, nine sample frames of each, for 4000 objects, are shown in Figure 8. Complete animations can be found in [SR17].

All simulations were built using the Unity Game Engine [Uni17] rigid body physics and recorded to disk, so that all algorithms were executed with the same inputs. To ensure that all simulations are similar, despite the number of objects, the application space was chosen as a cube of volume equal to 50 times the combined volume of all objects and the gravity vector was set to 10% of the application space side length (pointing downwards). For the *Cubes* simulations, cubes with size 1 are used. For the *Assorted* simulations, we define six object classes: bars (very thin and long), planks (thin and rectangular), squares (thin and square), big cubes, small cubes and random cubes. The sizes of these objects are chosen so as to ensure that the combined objects volume is the same for both *Cubes* and *Assorted* scenarios. Altogether, these choices ensure that every aspect of the simulation remains the same when varying the number of objects. In total, 1000 frames were recorded for each simulation in time steps of $\frac{1}{30}$ s, yielding approximately 33 s of animation. To evaluate scalability, each simulation was performed for several numbers of objects, ranging from one thousand to one million objects.

To compare our solution, we selected several open-source algorithms from the industry and literature, focusing on state-of-the-art libraries and recent publications with available source code. From the Bullet Library [Cou17], the algorithms DBVT and AxisSweep were selected. The former is a BVH-based algorithm that has two variants, one designed for coherent and the other for incoherent scenarios, whereas the latter is an SAP-based algorithm similar to [BW92]. These solutions will be referred to as DBVT F, DBVT D and AxisSweep. Apart from Bullet, the Computational Geometry Algorithms Library (CGAL) [The16] provides a CD algorithm [KMZ16] based on the algorithm originally proposed by Zomorodian and Edelsbrunner [ZE00]. This algorithm uses a hybrid of interval/segment trees with the SAP technique and will be referred to as CGAL. Finally, the work presented by Tracy *et al.* [TBW09] is also available and will be named after its first author. The algorithm is a hybrid solution that uses a grid and the SAP algorithm to prune pairs. It is important to mention that both Bullet's AxisSweep and Tracy's algorithm are based on Baraff's approach to the SAP algorithm [BW92]. In essence, it is an incremental CD algorithm based on managing sorted lists of object projections, one list per axis. Whenever projections are moved within a list, they may add/remove collisions to/from the cache. This approach is optimal



(a) Free Fall.

(b) Brownian.

(c) Gravity.

Figure 8: *From top left to bottom right, nine sample execution frames for each of the three detailed scenarios (*Free Fall, Brownian *and* Gravity*) for* 4000 *objects. On the left, simulations using uniformly sized objects and, on the right, the non-uniform case.*

when only a handful of objects move, as its complexity is proportional to the number of operations to re-sort the lists. In comparison, our SAP implementation is more closely related to that of Liu *et al.* [LHLK10], which is a complete CD algorithm based on sorting and sweeping projections on a single axis. A thoughtful analysis of the SAP algorithm is presented by Tracy *et al.* [TBW09].

The system was implemented in C++ and simulations were carried out on Unity 2017.1. All tests were executed on a Windows 10 ×64 machine running on an Intel Core i7 7700 with 8GB of RAM and an NVIDIA GT 1060 GPU. Finally, all time measurements were

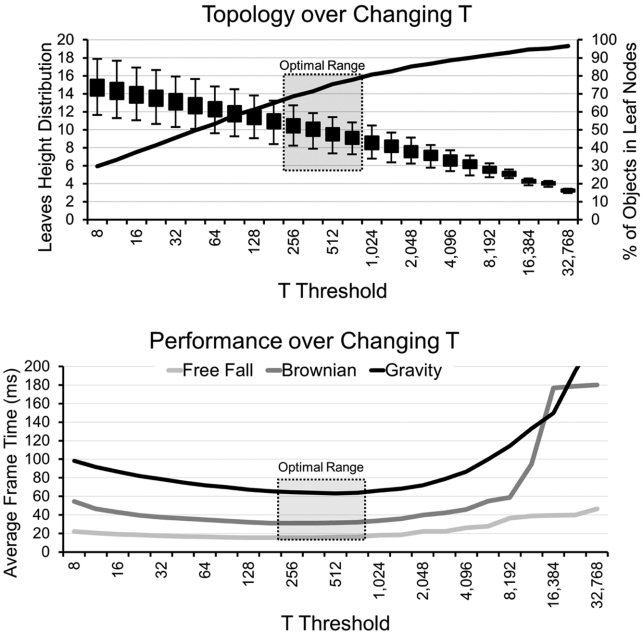


Figure 9: *Tree topology and performance over changing the threshold T. On the top, the height distribution of leaf nodes and the respective percentage of objects on leaf nodes (averaged over all scenarios). On the bottom, the respective performance achieved. Data collected for 128 000 object simulations.*

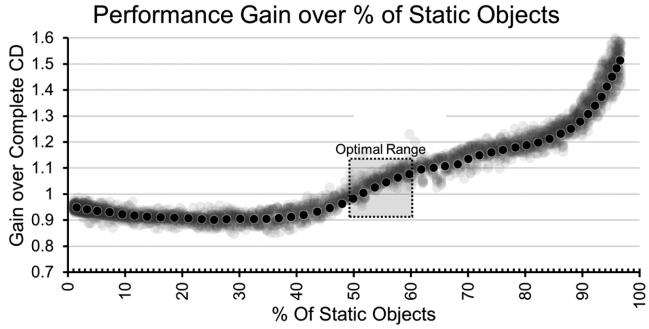


Figure 10: *Comparison of the Incremental and Complete approaches with regard to the percentage of static objects. The plot shows the relative gain of the Incremental approach for all frames of all* Free Fall *executions [clamped to the (1%, 99%) and (0.7, 1.6) range]. The darker shades correlates to more data points. The data moving average is shown as black dots.*

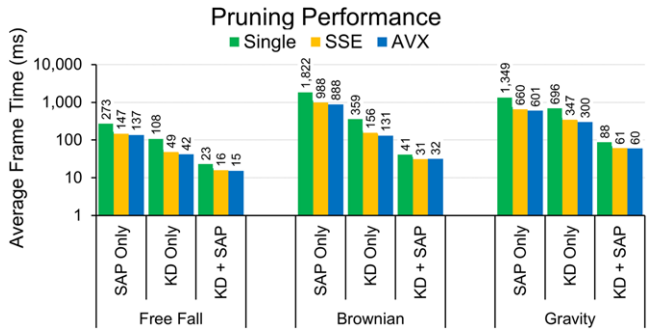


Figure 11: *Performance of the individual pruning and SIMD optimizations (128 000 object simulations), logarithmical scale.*

made as precise as possible and correspond to the time spent by the algorithms only, no other tasks were included in the measurements.

5. Tests and Results

In Figure 9, focusing on our KD-Tree algorithm, we plot the T value influence on the tree topology and performance. Near the extremes, a relationship between the tree's height and performance can be noted. A tall or short tree leads to a worse performance than an intermediary one ($T = 512$, for instance). The algorithm performance is practically constant around $T = [256$ to $1024]$. These T values allow the algorithm to make the most of the SAP and the SIMD optimization. For really high T values (2048 onwards), the pruning becomes insufficient, to the point that it significantly lowers the performance, despite the use of a second pruning step. The *Brownian* scenario, of all three, is the most affected by this, due to its more uniform object distribution. This shows up as a flat line on $T = [16\,384$ to $32\,768]$, as these values of T end up generating similarly sized trees.

From a user standpoint, the T parameter is relatively straight forward to tune, since it has a wide range of similarly performing values and a single global minimum plateau. Contrastingly, the ε parameter is application-dependent. However, it only depends on the objects' dimensions. If μ is the average object size, one can use $\varepsilon = \frac{\mu}{100}$ as a default value. This value is sufficient to detect static objects, but not large enough to generate many false negatives.

Finally, to tune K, we present a comparative analysis of the Incremental and Complete approaches for the *Free Fall* scenario in Figure 10. The plot shows the relative gain of the Incremental approach for all frames of all *Free Fall* executions. The data reveal several key insights on the algorithm behaviour over the amount of static objects. Firstly, the gain/loss of using the Incremental variant is consistent across the entire spectrum, despite the data being collected from 1000 to 1 024 000 objects simulations, with the absolute variation being at most 5% for any percentage of static objects. Secondly, the spectrum can be clearly divided into tree separate regions: from 0% to 50% static objects, the Complete approach is consistently 5% to 15% faster; from 50% to 60%, both variants are roughly equals in performance; however, from 60% to 100% of static objects, the Incremental approach is considerably faster. Based on this data, we justify the importance of the adaptivity mechanism to leverage both the Complete and Incremental algorithms performance and we define $K = 60\%$. Unless otherwise stated, all figures in this work use $T = 512$, $\varepsilon = \frac{\mu}{100}$ and $K = 60\%$.

In Figure 11, we present the pruning performance of each individual pruning stage and how they relate to the use of the SIMD optimization. The plot clearly shows that using the *SAP Only* or the *KD-Tree Only* solutions is not optimal compared to the combined

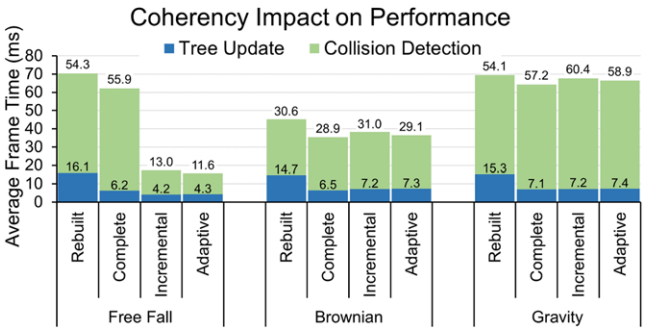


Figure 12: *Performance comparison of temporal reasoning techniques (128 000 object simulations).*

algorithm. In fact, the limited pruning effectiveness of the individual solutions is apparent when comparing the SIMD widths. For the individual cases, the SIMD test leads to a major performance improvement over the base case. For the combined solution, its relative improvement is minor, since there are considerably less candidate pairs to be tested. With regard to the SIMD optimization, we found that AVX instructions have the biggest improvement overall, but its improvement over the SSE instructions is marginal. In particular, the number of AABB tests batched into AVX instructions follows a positively skewed distribution. For instance, considering the *Gravity* scenario with 128 000 cubes, the average number of tests batches is 15.8, but around 48% of the batches have eight or less tests. This shows that many batches generate unsubstantial work to fully use the wider instructions, explaining the marginal gain over simpler SSE instructions. As regards memory, for any of the test scenarios, we did not find convincing evidence that the algorithm is memory bound. Applicability-wise, SIMD optimizations are of special interest as they are orthogonal to any scenario property, besides the raw number of collisions.

The performance of the *Complete*, *Incremental* and *Adaptive* approaches are shown in Figure 12. Additionally, we include a *Rebuilt* variant that consists of using the Complete CD and no tree update algorithm, rebuilding the tree entirely every frame. Of all four

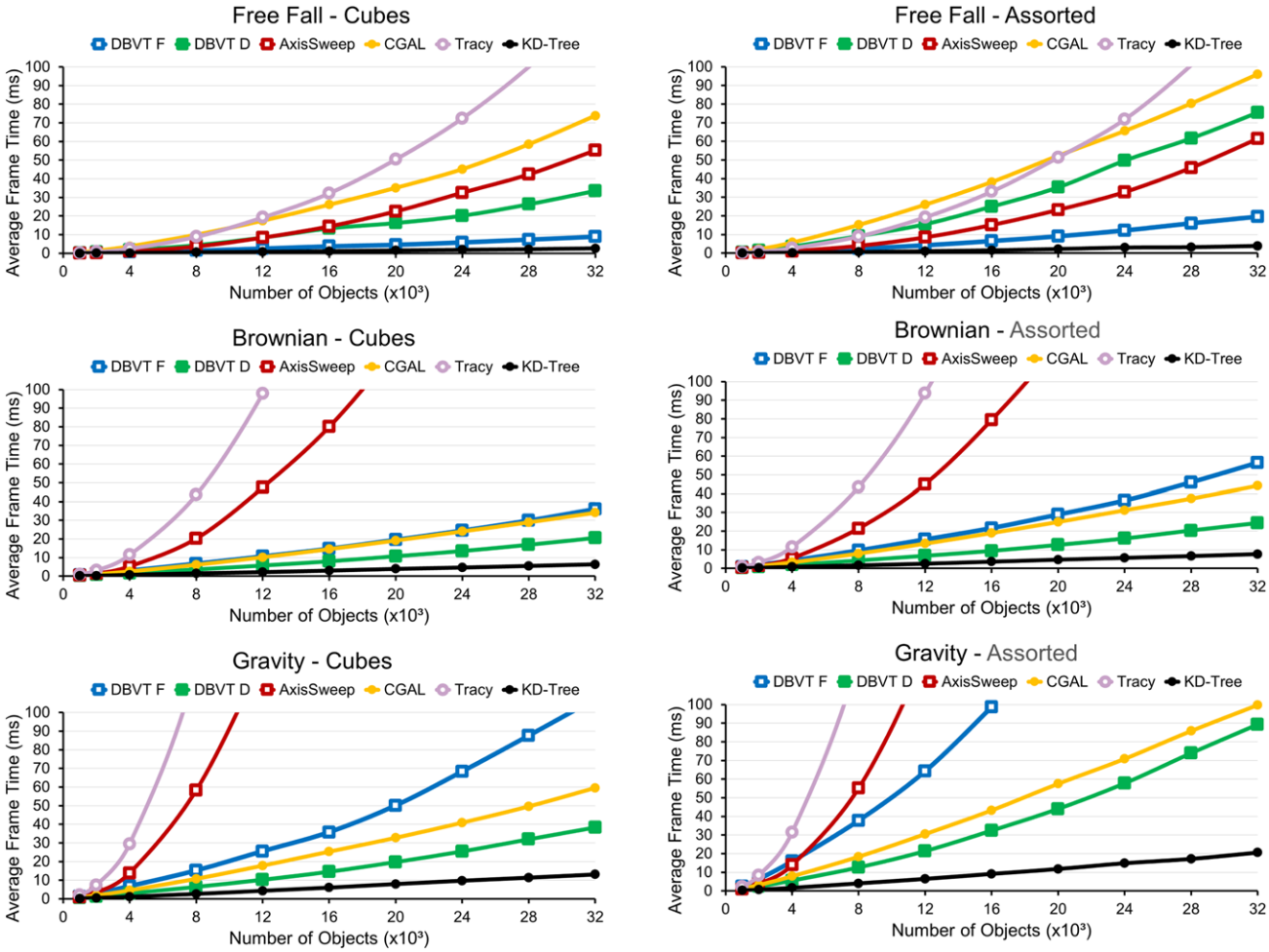


Figure 13: *Performance comparison of broad-phase algorithms for the uniformly sized objects case (left) and for the non-uniform case (right). Bullet's algorithms have square markers and coherency-optimized algorithms have white-background markers.*

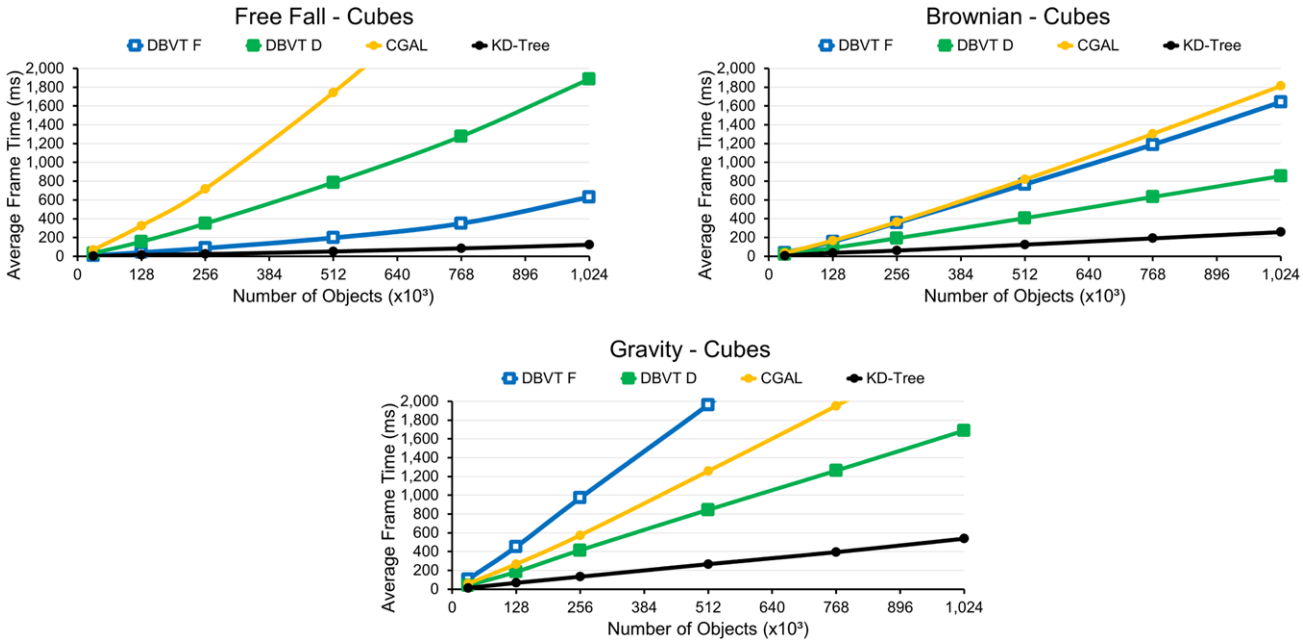


Figure 14: *Continued performance comparison for up to a 1 024 000 uniformly sized objects for the four best performing algorithms.*

variants, the *Rebuilt* approach, which is completely indifferent to coherency, performs the least, due to its increased update cost, even for the most incoherent simulations. This shows the efficiency of our update algorithm at handling incoherent setups. In sequence, the *Complete* and *Incremental* versions are the best approaches for their specific goals, with the *Adaptive* method being a close second in all scenarios. Clearly, the use of coherency optimization is invaluable when handling the *Free Fall* scenario, for this reason, we judge the adaptivity to be a strength of our algorithm.

In Figure 13, we compare our solution to the CPU state of the art with uniformly and non-uniformly sized objects. For all scenarios using the cubes setting, our solution consistently performs twice as better than the second best algorithm. In sequence, the DBVT solution is the second best algorithm for all scenarios (using the D or F variant). The CGAL algorithm is notable for its quasi-linear scalability, despite being less efficient. Finally, the Tracy and AxisSweep algorithms provide both the worst performances and scalability. For the Assorted scenarios, a similar picture is seen overall, but several significant differences can be noted. The most contrasting curves are those generated by the DBVT D and DBVT F algorithms, which are based on the BVH structure. These algorithms present a significantly lower performance/scalability for the assorted case, specially for the *Gravity* scenario. The AxisSweep algorithm is also significantly affected by the more complex objects, although, not as much as the DBVT algorithms. All other algorithms are only marginally affected by this change, showing that they are more resilient to changes in the objects' complexities.

In order to extend our scalability comparison, Figure 14 shows the four best performing solutions for up to one million cubes. Although having greatly increased the number of objects, a similar picture is shown, being that the KD-Tree the fastest, followed by the DBVT and CGAL algorithms. The AxisSweep and Tracy algorithms were not included because in our scenarios, they did not scale well beyond 32 000 objects. Additionally, assorted objects simulations for more than 32 000 objects were not feasible to generate. Numerically, the performance data for cubes and assorted simulations are detailed, respectively, in Tables 1 and 2. For cube scenes, the data show that our solution is, on average, two to three times faster than the other tested solutions. For assorted scenes, the gains are even higher, as the assorted objects affected much more the other algorithms than ours.

To visualize the solutions' applicability, Figure 1 shows a radar plot of the best performing algorithms in each of the three application scenarios. This plot conveniently displays the individual worst-case and best-case scenarios of each solution. The DBVT F algorithm clearly performs better for the coherent case and worse for the incoherent ones, specially *Gravity*. In contrast, both CGAL and DBVT D display some preference for the incoherent-uniform axis and a considerably worse performance on *Free Fall*. The KD-Tree algorithm, however, consistently provides the best performance of all tested algorithms, with a small preference for the coherent and incoherent-uniform cases. However, for all algorithms, the trend towards disfavouring non-uniform distributions can be explained by the fact that these distributions lead to a higher number of collisions, as more objects are stacked together, whereas uniform ones allow for more spatial reasoning.

Although BVHs are considered more suited for dynamic scenes, the KD-Tree solution performed better on all test scenarios. The main reason for its superior performance is the lower tree height. With $T = 512$, our structure can be up to nine levels shorter than a one object *per* leaf tree. This greatly reduces the overhead of tree traversals and the cost of updating the structure. By pairing

Table 1: *Numerical results for the four best performing algorithms in Cubes scenarios. All time measurements are in milliseconds.*

Thousands of objects		1	2	4	8	16	32	64	128	256	512	1024
Free Fall	DBVT F	0.12	0.30	0.68	1.47	3.76	8.90	26.56	42.78	88.62	198.04	630.89
	DBVT D	0.29	0.74	1.84	4.43	13.43	33.57	92.69	156.33	348.99	786.65	1887.57
	CGAL	0.52	1.38	3.75	9.99	26.12	73.86	172.23	325.47	717.84	1743.76	4085.13
	KD-Tree	0.04	0.10	0.21	0.55	1.15	2.68	6.05	12.80	25.19	53.40	123.45
	Gain	193%	206%	219%	167%	226%	232%	339%	234%	252%	271%	411%
Brownian	DBVT F	0.67	1.36	3.07	6.78	14.79	36.08	101.68	155.66	355.57	765.34	1641.21
	DBVT D	0.32	0.72	1.63	3.64	8.00	20.53	63.71	90.24	192.09	406.57	854.59
	CGAL	0.37	1.00	2.51	6.07	14.51	34.13	84.28	168.14	365.53	818.28	1813.24
	KD-Tree	0.16	0.29	0.75	1.55	3.01	6.38	13.44	29.22	60.43	123.75	258.52
	Gain	101%	152%	118%	135%	166%	222%	374%	209%	218%	229%	231%
Gravity	DBVT F	1.18	2.90	6.82	15.22	35.78	107.32	292.75	452.40	971.81	1962.63	3898.99
	DBVT D	0.52	1.27	2.91	6.40	14.56	38.41	106.84	183.30	413.32	843.91	1688.11
	CGAL	0.73	1.88	4.58	10.76	25.32	59.53	132.26	267.01	573.87	1256.41	2666.89
	KD-Tree	0.21	0.50	1.12	2.65	6.06	13.16	28.19	62.70	132.95	266.79	539.27
	Gain	145%	152%	159%	141%	140%	192%	279%	192%	211%	216%	213%

Table 2: *Numerical results for the four best performing algorithms in assorted scenarios. All time measurements are in milliseconds.*

Thousands of objects		1	2	4	8	16	32
Free Fall	DBVT F	0.17	0.43	1.05	2.51	6.53	19.53
	DBVT D	0.52	1.41	3.63	9.24	26.05	75.42
	CGAL	0.82	2.19	5.74	15.17	38.21	96.01
	KD-Tree	0.05	0.13	0.30	0.68	1.56	3.86
	Gain	210%	228%	252%	267%	318%	406%
Brownian	DBVT F	0.78	1.84	4.24	9.63	21.63	56.52
	DBVT D	0.37	0.85	1.94	4.32	9.44	24.31
	CGAL	0.52	1.32	3.30	8.02	19.01	44.15
	KD-Tree	0.16	0.34	0.86	1.67	3.60	7.69
	Gain	134%	146%	126%	159%	162%	216%
Gravity	DBVT F	2.48	6.44	15.96	37.77	98.70	281.58
	DBVT D	1.00	2.40	5.62	12.66	32.40	89.34
	CGAL	1.34	3.26	7.90	18.41	43.29	99.75
	KD-Tree	0.31	0.71	1.70	4.09	9.20	20.65
	Gain	218%	235%	231%	209%	252%	333%

the solution with the SAP algorithm, we provide a second level of pruning that more than compensates the reduced tree. We also highlight that, even though a reduced BVH algorithm is possible, the KD-Tree offers several key advantages over such a tree. Firstly, KD-Trees can be built top-down more naturally than BVHs, which usually resort to an implicit KD-Tree building [Wal07]. Secondly, on a reduced BVH traversal tree [TMT10], each leaf–leaf node intersection would generate up to T^2 external pairs, whereas on a KD-tree, we can easily store objects in internal nodes and use the external objects algorithm to generate a minimal number of external pairs. Lastly, update algorithms based on refitting nodes such as those used by the DBVT algorithm [Cou08]) would not work properly on a reduced BVH due to the lack of a deep structure.

Finally, we believe that the given analysis supports the claim of efficiency, general applicability and ease of use of our solution. Figures 1, 13 and 14 and Tables 1 and 2 show the performance gain of our solution over the state-of-the-art CPU algorithms, in all three scenarios and for uniformly and non-uniformly sized objects. Additionally, the results shown in Figures 11 and 12 point to the use of our adaptivity mechanism and AVX instructions in all these scenarios. For the ease of use, Figures 9 and 10, as well as our analysis, show suitable default values for the T, ε and K variables. Further tuning will not lead to significant performance improvements, regardless of scenario.

6. Conclusions and Future Work

We have presented two main contributions to the field of CD. Firstly, we successfully proposed, implemented and tested an efficient novel algorithm for broad-phase CD, which is, on average, two to three times more efficient than the state-of-the-art CPU algorithms. Secondly, our solution provides the field with a general and easy-to-use solution. We have shown that the structure can handle scenes that vary from static to completely dynamic, uniform and non-uniform object distributions, uniform and non-uniform object sizes and coherent and incoherent motion. Our solution is of special interest to application domains where: (1) a custom algorithm would be too costly, (2) multiple scenarios exist and (3) scenarios change over time. In general, these domains share a restricted resources/time budget for the development of specialized algorithms and would profit from a generalized solution.

Although we designed our algorithm to perform broad-phase CD in physics simulations, it can be used in several other applications whose goal is to find interactions between pairs of objects belonging to a set, such as: n-body simulations, proximity queries, massively multi-player online (MMO) games, surface self-intersections, etc. Within the field of CD, the solution can be extended to support ray, sphere and AABB casts as well as techniques such as collision filtering. Additionally, the solution can be easily extended to the d-dimensional case, but it is unknown if the solution would still be competitive for higher dimensional problems. Furthermore, decoupled from the CD, the proposed structure can be used as an alternative method of k-dimensional organization of dynamic objects

and be applied to other still challenging areas of computer graphics, such as visibility culling and real-time ray tracing.

As future work, we anticipate the interest in enhancing the proposed solution. Of special interest, we plan to study the multi-core CPU execution and/or to use the GPU for improved performance and scalability. Several works provide parallel solutions for Grids, BVHs and KD-Trees, showing that all three approaches can be built, updated and traversed in parallel with almost linear speedups [DLAG12, Wal07]. However, we are unaware of any work focused on a general, highly applicable, parallel solution for the broad-phase problem. We highlight that different parallel schemes may differ substantially with regard to the objects distribution and motion coherence. Thus, it indicates the need to do further research on this problem. Additionally, the update algorithm could be tested and tuned for other application domains, as previously mentioned. Moreover, some of these concepts might be applied to BVH and BSP structures. Finally, we would like to integrate our solution within the *Bullet* library, making it accessible to the research community.

Acknowledgements

Ygor Rebouças Serpa and Maria Andréia Formico Rodrigues would like to thank the Brazilian Agency CAPES (under grants 88887.176617/2018-00 and 88881.120921/2016-01, respectively) for their financial support. The authors are also very grateful to the referees for providing insightful comments and suggestions to improve the manuscript.

References

[AGA11] Avril Q., Gouranton V., Arnaldi B.: Dynamic adaptation of broad phase collision detection algorithms. In *Proceedings of 2011 IEEE International Symposium on VR Innovation* (March 2011), pp. 41–47.

[AGA12] Avril Q., Gouranton V., Arnaldi B.: Fast collision culling in large-scale environments using GPU mapping function. In *Eurographics Symposium Proceedings* (2012), pp. 71–80.

[BW92] Baraff D., Witkin A.: Dynamic simulation of non-penetrating flexible bodies. In *SIGGRAPH '92: Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1992), ACM, pp. 303–308.

[Cou08] Coumans E.: btDBVT documentation. bulletphysics.org/mediawiki-1.5.8/index.php?title=BtDbvt_dynamic_aabb_tree, 2008.

[Cou17] Coumans E.: Bullet Physics library 2.85. bulletphysics.org, 2017.

[DLAG12] Dodier-Lazaro S., Avril Q., Gouranton V.: Dynamic spatial subdivision for n-body collision detection on multi-CPU and multi-GPU architectures. *Bibliographic Study* (2012). https://scholar.google.com.br/scholar?hl=en&as_sdt=0%2C5&q=dodier-lazaro+s+dynamic+spatial+subdivision+for+n-body&btnG=

[DSC05] Daniel S., Coming O. G. S.: Kinetic sweep and prune for collision detection. In *Proceedings of the 2nd Workshop in Virtual Reality Interactions and Physical Simulations (VRIPHYS'05)* (November 2005).

[Eri04] Ericson C.: *Real-Time Collision Detection*. Elsevier, Amsterdam, 2004.

[Gla05] Glass K.: *Analysis of Broad-Phase Spatial Partitioning Optimizations in Collision Detection*. Tech. rep., Rhodes University, Grahamstown, South Africa, 2005.

[HBHS05] Havran V., Bittner J., Herzog R., Seidel H. -P.: Ray maps for global illumination. In *EGSR '05: Proceedings of the 16th Eurographics Conference on Rendering Techniques* (Aire-la-Ville, Switzerland, Switzerland, 2005), Eurographics Association, pp. 43–54.

[KMZ16] Kettner L., Meyer A., Zomorodian A.: Intersecting sequences of dD iso-oriented boxes. In *CGAL User and Reference Manual*, 4.9 ed. CGAL Editorial Board, 2016.

[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* (New York, NY, USA, 2005), ACM, pp. 179–186.

[LGS*09] Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D.: Fast BVH construction on GPUs. In *Computer Graphics Forum* (2009), vol. 28, Wiley Online Library, pp. 375–384.

[LHLK10] Liu F., Harada T., Lee Y., Kim Y. J.: Real-time collision culling of a million bodies on graphics processing units. *ACM Transactions on Graphics (TOG) 29*, 6 (December 2010), 1–8.

[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 (TOMACS) 23*, 3 (July 2013), 1–22.

[Mor11] Mora B.: Naive ray-tracing: A divide-and-conquer approach. *ACM Transactions on Graphics 30*, 5 (October 2011), 117:1–117:12.

[NVi17] NVidia: Physx 3.3. developer.nvidia.com/gameworks-physx-overview, 2017.

[SN15] Schauer J., Nüchter A.: Collision detection between point clouds using an efficient K-D tree implementation. *Advanced Engineering Informatics 29*, 3 (2015), 440–458.

[SR17] Serpa Y. R., Rodrigues M. A. F.: Flexible use of temporal and spatial reasoning for fast and scalable CPU broad phase collision detection using KD-trees. https://www.youtube.com/watch?v=1eJVuaKm6Mo, 2017. Last accessed on 19 july 2018.

[TB12] Tracy D. J., Brown S.: Accelerating physics in large, continuous virtual environments. *Concurrency and Computation: Practice & Experience 24*, 2 (February 2012), 125–134.

[TBW09] Tracy D. J., Buss S. R., Woods B. M.: Efficient large-scale sweep and prune methods with AABB insertion and removal. In *VR'09: Proceedings of the 2009 IEEE Virtual Reality Conference* (Washington, DC, USA, 2009), IEEE CS, pp. 191–198.

[The16] The CGAL Project: *CGAL User and Reference Manual*, 4.9 ed. CGAL Editorial Board, 2016.

[TMT10] Tang M., Manocha D., Tong R.: Mccd: Multi-core collision detection between deformable models using front-based decomposition. *Graphical Models 72*, 2 (2010), 7–23.

[Uni17] Unity Technologies: Unity 3D v5.5. unity3d.com, 2017.

[Wal07] Wald I.: On fast construction of sah-based bounding volume hierarchies. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 33–40.

[WDZ17] Weller R., Debowski N., Zachmann G.: kDet: Parallel constant time collision detection for polygonal objects. *Computer Graphics Forum 36*, 2 (2017), 131–141.

[WFZ13] Weller R., Frese U., Zachmann G.: Parallel Collision Detection in Constant Time. In *Proceedings of the Workshop on Virtual Reality Interaction and Physical Simulation* (2013). J. Bender, J. Dequidt, C. Duriez and G. Zachmann (Eds.), The Eurographics Association.

[ZE00] Zomorodian A., Edelsbrunner H.: Fast software for box intersections. In *SCG'00: Proceedings of the 16th Annual Symposium on Computational Geometry* (New York, NY, USA, 2000), ACM, pp. 129–138.