DOI: 10.1111/cgf.13884

COMPUTER GRAPHICS *forum*
Volume 00 (2019), number 00 pp. 1–14

Broadmark: A Testing Framework for Broad-Phase Collision Detection Algorithms

Ygor Rebouças Serpa and Maria Andréia Formico Rodrigues

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

Abstract

Research in the area of collision detection permeates most of the literature on simulations, interaction and agents planning, being commonly regarded as one of the main bottlenecks for large-scale systems. To this day, despite its importance, most subareas of collision detection lack a common ground to test and validate solutions, reference implementations and widely accepted benchmark suites. In this paper, we delve into the broad-phase of collision detection systems, providing both an open-source framework, named Broadmark, to test, compare and validate algorithms, and an in-deep analysis of the main techniques used so far to tackle the broad-phase problem. The technical challenges of building this framework from the software and hardware perspectives are also described. Within our framework, several original and state-of-the-art implementations of CPU and GPU algorithms are bundled, alongside three benchmark scenes to stress algorithms under several conditions. Furthermore, the system is designed to be easily extensible. We use our framework to bring out an extensive performance comparison among assembled solutions, detailing the current CPU and GPU state-of-the-art on a common ground. We believe that Broadmark encompasses the principal insights and tools to derive and evaluate novel algorithms, thus greatly facilitating discussion about successful broad-phase collision detection solutions.

Keywords: open-source framework, collision detection, broad phase, state-of-the-art implementations, CPU and GPU algorithms

ACM CCS: Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [Computing Methodologies]: Animation–Collision detection

1. Introduction

Collision detection algorithms deal with the relative configuration of objects within a given space, which cannot penetrate one another during their simulated motion. These algorithms focus on relevant relations, such as simple overlaps, intersection of geometries and, more generally, spatial or proximity relationship between geometric objects [MCLK17]. Various groups of authors have developed algorithms for collision detection in motion simulation of both rigid and deformable bodies, clothes, particles and fluids, among others. Essentially, their primary focus has been on developing efficient algorithms for collision detection [Eri04]. However, only a few of these works have made source code publicly available, employ similar testing methodologies or even compare against standard solutions of their respective interest areas. This hinders further advancements in the collision detection area, making it difficult to assess the relative effectiveness of algorithms. Moreover, it generates duplication of effort, insufficient testing, failure to test against state-of-art and poor choice of parameters.

Collision detection is often broken down into two sequential phases: broad phase and narrow phase. The former searches for all bounding volume (BV) overlaps, while the latter queries if the bounded objects also overlap [Mir97]. Thus, it is basically a coarse-fine search. While the broad phase prunes the set of all object pairs to a small set of potential collisions, the narrow phase performs the actual overlap tests. Without the broad phase, performing the narrow phase would be intractable, hence its importance. This separation also favours code reuse, since the broad phase can be employed unchanged for any set of boundable objects, and, if precision is not a concern, the narrow phase can be simplified or even absent. In this work, we focus on the broad phase and, particularly, on using

axis-aligned bounding box (AABB) as BVs. In other fields in computer science, this task is also called the box intersection checking problem [KMZ16, LLCC13, FM17].

Main contribution In this paper, we seek to establish a ground base for future broad-phase collision detection research, providing a complete testing framework, named Broadmark, and an in-deep analysis of the main broad-phase techniques. Our testing framework comprises several original and state-of-the-art implementations of CPU and GPU algorithms and a set of benchmark scenes. We use our framework to bring out an extensive performance comparison among assembled solutions. In addition, we detail the current state-of-the-art on a common ground and relate the single, multi and many threaded solutions. We believe that Broadmark encompasses the principal insights and tools to derive and evaluate novel algorithms, thus greatly facilitating discussion about successful broad-phase collision detection solutions. The Broadmark source code and detailed documentation are available on GitHub at https://github.com/ppgia-unifor/Broadmark.

The remainder of this paper is structured as follows: Section 2 surveys the related work on broad-phase collision detection, stressing how many authors validate their respective solutions and what sort of issues they usually investigate. Section 3 describes the typical structure of a broad-phase algorithm and delves on the standard approaches to solve this problem. Section 4 describes the developed benchmark tool, its built-in scenes and the bundled algorithms. In Section 5, we present a complete scalability analysis of the implemented algorithms. Finally, Section 6 concludes with a summary of our main contributions, alongside a discussion about the future work possibilities arising from these studies and findings.

2. Related Work

Several authors present comparative studies to validate their novel algorithm's efficiency. Broadly, these works can be divided on whether synthetic data [LLCC13, TBW09, CL16a, CL18] or physical simulations [LCF05, SR17, LHLK10, TB12, WM09] have been used to perform collision detection tests. Studies using synthetic data generally consist of constant-speed objects that bounce on the scene walls. Usually, each object is both initialized with a random direction and allowed to pass through other objects. These scenes are considered coherent/predictable, show a very uniform object distribution and, despite being artificial, they appear quite frequently in the literature. On the contrary, physically based scenes are much less predictable, give rise to non-uniform distributions and closely match real-world usage scenarios. However, they are harder to setup and costly to generate, especially in scenarios with a massive number of objects.

With regard to analysis, the majority of existing works focus on scalability and include analysis of some other properties relevant to the developed approach. Algorithms that make use of temporal techniques commonly analyse performance across the number of moving objects [LHLK10, SR17, TB12, CL18], showing their speedups on predictable scenes. Works based on Grid-like spatial subdivisions usually present some form of non-uniformly distributed objects analysis [LLCC13, AGA12]. On the multi-core and many-core literature, the number of cores or GPUs and other related costs have been also covered [LLCC13, AGA12, BMPS09, AGA14, CL18, FM17]. Some works aim to use multiple application scenarios [ZE00, LHLK10, SR17] or focus on specific problems, such as the cost of adding/removing objects [TBW09], the behaviour under largely static scenes [TB12], the use of custom hardware [WDM07, WM17] or specific constraints [CL16b, FM17, She14]

In addition, physics engines, such as Bullet [Cou18] and PhysX [Nvi19a], are generally not focused on benchmarks, outside their own algorithms and engine versions. Bullet 2 had a broad-phase benchmark, which has been used by some authors as state-of-the-art [LHLK10, LLCC13, TBW09]; however, it is no longer available. PhysX library developers maintain the Physics Engine Evaluation Lab (PEEL), which is a suite of tests to evaluate, compare and benchmark physics engines. Even so, PEEL is actually dedicated to compare engine features, stability and correctness, but not collision detection performance [Ter17]. In the literature, Woulfe and Manzke [WM09] present one of the few attempts at a general testing framework: a physically based simulator, a highly configurable initial state and three broad- phase algorithms from Bullet. However, although it is a noteworthy effort, it has two major limitations: the initial state is the sole point of customization and objects are re-simulated for each test, which is unfeasible for large simulations.

In this work, we propose to the collision detection community a framework, named Broadmark, for the study and benchmark of broad-phase algorithms. This addresses the extinction of the Bullet 2 benchmark, the lack of a Bullet 3 successor and is concerned specifically with this problem, unlike PEEL. Furthermore, our simulation generator is highly customizable and is fully decoupled from the algorithms execution. In more detail, the framework comprises a set of reference implementations and representative scenes, which have been developed using the physical simulation approach. We have designed this system towards investigating the algorithms behaviour under uniform/non-uniform distributions and on static/dynamic scenes, as these are the most commonly investigated features. Within the framework, several algorithms from the literature [ZE00, TBW09, SR17] and industry [Cou08a, Cou08b, Cou14b, Cou14a, Cou08a], as well as original implementations, were bundled [Brute force (BF), sweep-and-prune (SAP) and Grid], being some of these GPU based or multi-core. The complete solution has been designed to be extensible, allowing new scenes and algorithms to be added, as well as scoped for addressing the needs of most authors on designing new solutions.

3. Collision Detection Algorithms

At a high level, broad-phase algorithms can be classified by their use of spatial, temporal and sorting techniques. Spatial techniques seek to divide-and-conquer the problem using spatial data structures, such as Grids [LLCC13], BVHs [Cou08b], KD-Trees [SR17], Octrees and BSP-Trees [LCF05]. On the other hand, temporal techniques explore the behaviour of the simulation to mitigate the workload. Common use cases are the reuse of data structures [TBW09, Cou08b], incremental search methods [LHLK10, SR17, Cou08a] and future behaviour prediction [DSC05]. Lastly, many algorithms use sorting to extract relationships of locality/proximity to uncover collisions [BW92, CL16a, CL18, TBW09, ZE00, LHLK10, AGA10, SR17, Cou08a].

Most of the previously mentioned techniques reduce the search space by pruning pairs, being collectively referred to as pruning techniques. To reach competitive performance, most works employ more than one of these strategies. Initially, some form of divide-and-conquer approach is usually used, followed by a sorting-based search [ZE00, LHLK10, SR17, TBW09]. In particular, special care must be given to avoid over-pruning. For small search spaces, the cost of pruning is commonly higher than simply testing each pair of objects. With recent CPUs, single instruction, multiple data instructions (SIMD), and GPUs, this fact is even more pronounced. Similar issues arise in other areas of computer graphics, for instance, the cost of rendering a triangle on modern hardware has decreased so significantly that over-culling is currently a significant issue [SR18].

Within the domain of temporal techniques, the incremental search deserves special attention. The general idea is to take the set of collisions from the previous frame, find all new collisions in the scene and remove all ceased collisions. In other words, the search is focused on finding the difference, in terms of collisions, between the previous frame and the current frame. This technique can yield significant gains in performance when a moderate to high number of objects is considered static [SR17, LHLK10, TB12].

In the following subsections, we present the main algorithms and techniques that have been used on the broad-phase collision detection. More specifically, we discuss these approaches according to: (1) the algorithms/techniques in their canonical form, (2) how they have been used so far, from an algorithm design perspective and (3) their respective relevant state-of-the-art.

3.1. Brute force

Algorithm The BF algorithm is based on exhaustively testing all possible pairs of a set of objects (Figure 1a). If applied to all objects, it is named BF broad phase and has a time complexity of $\mathcal{O}(n^2)$, due to the $\binom{n}{2}$ tests. If applied to a subset of all objects, it becomes an *operator* of the encompassing solution, for instance, to test collisions for each cell of a Grid subdivision [AGA12, LLCC13]. Finally, for small sets of objects, this is the fastest known algorithm to enumerate collisions. It is also worth mentioning that it lends itself nicely to SIMD vectorization [Eri04], p. 547].

From an algorithm design perspective The BF method is always used as an operator to conclude the search. Basically, the main challenge when using it is to avoid under- or over-pruning. For Grid or tree-based solutions, this boils down to finding the optimal Grid size or tree height, respectively. Multilevel Grids or unbalanced trees can also be used to more finely control the pruning. Within the GPU context, the BF approach is an important design asset due to its high predictability (which can be exploited to efficiently balance the workload across threads), and the sheer parallel power, which can drastically mitigate the quadratic nature of the algorithm [AGA12, LLCC13].

State-of-the-art Due to its algorithmic complexity, no modern algorithm is based solely on this technique. As shown by Geleri *et al.*, a highly optimized GPU BF can be efficient for small to medium-sized scenes, due to the sheer computing power; however, it does not scale as efficiently as other techniques [GTT13]. As an operator, it has been used on many GPU algorithms in conjunction with a Grid subdivision strategy, such as that available in Bullet's *b3GpuGridBroadphase* [Cou14b], and those described by Avril *et al.* [AGA12] and Lo *et al.* [LLCC11, LLCC13].

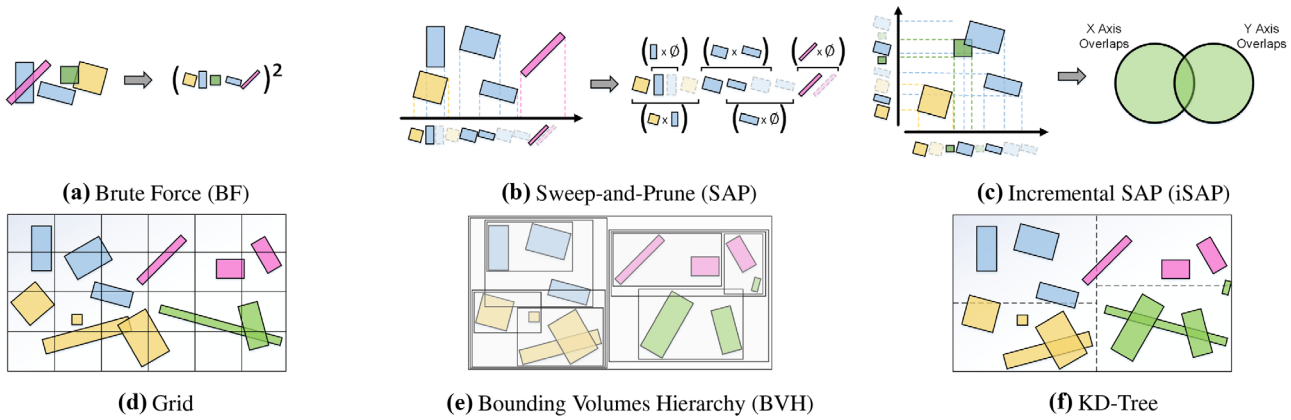
3.2. Sweep-and-prune

Algorithm The SAP algorithm is based on first performing a one-dimensional collision detection, outputting all overlapping pairs for the chosen dimension, followed by testing each of these pairs on the remaining axes and outputting the subset of pairs that overlap on all dimensions (Figure 1b). That is, it considers that for two objects to overlap in 3D, they must overlap along every 1D axis. Thus, any initial axis will lead to correct results. For efficiency, the 1D step, or *sweep step*, is performed by sorting the objects' minimum and maximum intervals, which are defined by two end points along the chosen axis, and reporting which objects are within the minimum/maximum of each other.

With regard to time complexity, for the general case, the SAP algorithm belongs to the class of $\mathcal{O}(nlog(n) + nk)$ algorithms, being $nlog(n)$ the cost of the sorting operation and k the average number of 1D overlaps per object, which is typically assumed to be a small constant. Alternatively, if the objects are supposed to be uniformly distributed, each object will have roughly $k = n^{\frac{2}{3}}$ 1D overlaps, leading to the $\mathcal{O}(n^{\frac{5}{3}})$ class, or simply $\mathcal{O}(n^{1.66})$ [TBW09]. As regards the space complexity, it can be performed in-place and, thus, belongs to the $O(1)$ class.

From an algorithm design perspective This technique can be used as-is for small to medium-sized scenes or used as an operator when tackling large to massive scenarios. For optimum performance, careful consideration must be given to the choice of sweeping axis. Some well-known strategies include using the axis of greatest variance [SR17], approximated principal components analysis [LHLK10] and context-aware heuristics [CL18]. Regarding the sorting algorithm, usual choices are the Quick sort and Radix sort, being the latter frequently used on parallel settings.

State-of-the-art Modern algorithms develop on the basic idea by either subdividing the space into many SAPs or further pruning object pairs after the sweep pass. The work of Serpa and Rodrigues employs the SAP algorithm as an operator of a KD-Tree structure [SR17], whereas Liu *et al.* use it on the GPU over a Grid subdivision [LHLK10]. Alternatively, Zomorodian and Edelsbrunner propose a combination of range and segment trees to further process the object pairs on the second and third dimensions [ZE00], while Capannini and Larsson describe a method to incorporate a second sweep step, named by the authors as *bi-dimensional SAP* [CL16a]. The same aforecited authors also explore optimizations when all objects are known to be same sized [CL16b]. Within the context of parallel hardware, for multi-core CPUs, the authors Avril *et al.*, as well as Capannini and Larsson, describe a parallelization strategy for the SAP and bi-dimensional SAP strategies, respectively [AGA10, CL18], while for GPUs Liu *et al.*, Geleri *et al.* and Grand describe methodologies to port the SAP algorithm to the many-core architecture [LHLK10, GTT13, LG07]. In libraries, the Zomorodian and Edelsbrunner's algorithm is packaged with the CGAL library [KMZ16], which was later used by Batista *et al.* to develop a simple parallel version of the algorithm [BMPS09]. In the PhysX library, the *multi-box-pruning*



(a) Brute Force (BF) **(b)** Sweep-and-Prune (SAP) **(c)** Incremental SAP (iSAP)

(d) Grid **(e)** Bounding Volumes Hierarchy (BVH) **(f)** KD-Tree

Figure 1: *The BF algorithm performs all object-object tests. This is the slowest and simplest broad-phase algorithm.*

and *automatic-box-pruning* algorithm are based on applying SAP to multiple user defined regions or over an automatic Grid subdivision, respectively [Nvi19b].

3.3. Incremental sweep-and-prune

Algorithm The rationale behind the incremental sweep-and-prune (iSAP) is to perform individual 1D passes for each axis, followed by selecting the object pairs that overlapped on all axes (Figure 1c). As proposed by Baraff, the main idea behind this algorithm is to keep three lists of sorted projections, one for each Cartesian axis, and adding the *min* and *max* edges of each projection as two separate entities [BW92]. During execution, when the AABB of the objects is updated, each *min* and *max* edges is resorted, adding and removing overlaps according to the following four general rules:

1. a *min* edge moving *down* and crossing a *max* edge is considered a new overlap,
2. a *min* edge moving *up* and crossing a *max* edge removes an overlap,
3. a *max* edge moving *down* and crossing a *min* edge removes an overlap, and
4. a *max* edge moving *up* and crossing a *min* edge is considered a new overlap.

At the end of the sorting step, all object pairs with three overlaps are considered colliding. For efficiency, overlaps are usually stored on a hashed structure with $O(1)$ insertions and removals. With regard to time complexity, Baraff classifies the algorithm as $\mathcal{O}(n+s)$, being s the number of move (or swap) operations needed [BW92]. For largely static scenes, s is assumed to be proportional to n, and for dynamic scenes s can easily grow to the order of n^2. In terms of space complexity, the algorithm clearly falls into the $\Theta(n)$ class, as it needs three n-sized lists.

From an algorithm design perspective The main advantage of this approach is that it builds upon the previous frame overlaps and uses a sorting strategy optimized for nearly sorted lists, thus, being ideal for largely static scenes. However, this approach is about three times slower than the default SAP algorithm for dynamic scenes, which only sorts one axis. As a complete solution, this algorithm is frequently found in libraries, as it tends to perform well in practice for small to medium-sized scenes, with many static objects. As an operator, iSAP is hardly used, since managing many instances of this algorithm is challenging.

State-of-the-art Optimized implementations of this algorithm can be found in the Bullet and PhysX libraries, under the names *btAxisSweep3* [Cou08a] and *sweep-and-prune* [Nvi19b], respectively. As an operator, Tracy *et al.* propose a Grid superstructure to host several iSAP instances. To manage the issue of moving objects from one iSAP to another, these authors describe a novel segmented-interval list structure to efficiently handle the insertions and removals [TBW09].

3.4. Grid

Algorithm The Grid space subdivision strategy is based on uniformly dividing the space into regions (or cells) and adding each object to all cells where they overlap (Figure 1d). Once the Grid is constructed, the broad phase is carried out by searching all overlaps for each cell. This algorithm is considerably simple to implement and lends itself nicely to parallelization, as well as to the use of other broad-phase collision detection approaches as operators.

From an algorithm design perspective Grids provide an easy and intuitive way to subdivide the scene into multiple independent regions, which can be operated in parallel. At run-time, Grids are very predictable and, for that reason, they are very popular in the parallel algorithms literature. When using Grids, two main design questions must be answered: how many cells will be used and how to handle *large objects*, i.e. objects that overlap many cells. For the first question, most authors target finding a value for objects per cell [TBW09, LHLK10, AGA12], while some others pose the problem as a maximization/minimization problem [LLCC13]. For the second question, the most common approaches so far have been to allow redundancy or, instead, to use some hierarchical/multilevel approach [SFC*19]. If objects are known to be same sized, this problem can be avoided altogether [FM17, WFZ13].

State-of-the-art From the Bullet Library, the *b3GpuGridBroadphase* algorithm uses a Grid plus BF approach

on the GPU. To handle large objects, a second BF step is used to find large-small and large-large intersections [Cou14b]. Similarly, the work of Avril *et al.* uses the same combination, but develops on the workload balancing issue, introducing a novel GPU mapping function to map threads to pairs [AGA12]. More broadly focused, the work of Lo *et al.* thoughtfully describes an elaborate GPU Grid construction scheme, a workload balanced BF, an output-compression strategy and a CPU-GPU pipelined execution to hide communication latency. In addition, also based on BF as an operator, Stroobant *et al.* propose a uniform and multi-level Grid approach for the GPU collision detection of nanoscale bodies. Outside of the Grid plus BF scope, several authors have proposed Grid plus SAP [LHLK10, LG07, Nvi19b] and Grid plus iSAP [TBW09] algorithms.

3.5. Bounding volume hierarchy

Algorithm The bounding volume hierarchy (BVH) uses BV to hierarchically bound groups of objects, from one BV per object to a single BV for the entire scene (Figure 1e). These are usually built as binary trees, but are not limited to them. To find overlaps, a single tree-tree test or n object-tree tests are performed. The former is faster as it performs a minimal number of BV tests, but the latter allows the algorithm to be incremental, for example performing object-tree tests for dynamic objects only. A common enhancement is to store w objects/leaf, being w the system SIMD width. This allows for object-tree and leaf-leaf tests to be carried using SIMD instructions.

As regards the tree itself, there are three fundamental strategies to construct a BVH: *bottom-up*, grouping nodes two-by-two, according to some proximity criteria; *top-down*, recursively subdividing the set of objects into disjoint groups; and *incrementally*, accommodating one object at a time. To update a BVH tree, many strategies exist, each with its sets of pros and cons. Notably, scheduled rebuilds, tree rotations and objects reinsertions are some of the most common ideas. To mitigate the number of objects in need of updating, several authors employ slightly larger BVs than necessary. This provides a certain threshold to move objects before requiring a repositioning.

From an algorithm design perspective The BVH algorithm is very popular, fast and easy to construct, and is widely regarded as the most cost-effective tree to handle dynamic objects. Usually, the main topics of discussion for BVHs are the algorithms to build and update the structure, and, to a lesser extent, the memory layout and optimization techniques performed to the tree-tree test, such as front-tracking. In the CPU literature, BVHs are usually built either top-down, searching the median point of objects, or incrementally, followed by some balancing algorithm; whereas in the GPU literature, the most widely used technique is the algorithm based on Morton codes, which enables a fully parallel bottom-up build [LGS*09]. Morton code-based BVHs are widely regarded as low quality, but the tree building speed more than compensates for it. As a component, BVHs are too cumbersome to be used as an operator, and too fine-grained to be used as super-structures, thus, for these reasons most BVH solutions are pure, in the sense that they do not employ any other major techniques.

State-of-the-art Within the Bullet Library, the *btDBVT* algorithm is a CPU-based single-core BVH broad-phase solution, heavily based on the reinsertion idea. The tree is built incrementally and features loose BVs. The tree quality is maintained by reinserting objects that moved beyond their BVs or by reinserting a random sample of objects each frame. The algorithm features an incremental detection option, which considers reinserted objects as dynamic and uses object-tree tests, and a complete detection one, using a full tree-tree test [Cou08b]. Also from the Bullet library, the *b3GpuParallelLinearBvh* broad phase is a GPU algorithm based both on the Morton Code algorithm and tree-tree tests. Due to the speed of the Morton Code-based build, no update algorithm is used and, therefore, the tree is simply rebuilt every frame [Cou14a].

3.6. Space partitioning trees

Algorithm Space partitioning trees work by recursively subdividing the world into regions. At each node, Octrees split the world into eight octants, KD-Trees divide it into two axis-aligned halves (Figure 1f) and BSP-trees can perform any binary split. In general, the more flexible a partitioning, the higher the costs to find optimal plane cuts, being the BSP the extreme case. For these trees, the top-down construction is the most effective, as it allows more informed plane cuts. Objects mean and median are commonly used measures to define the position of the split plane. The splitting axis, if relevant, is usually based on the objects variance.

Unlike BVHs, space partitioning trees do not handle objects directly, that is they have implementation-dependant characteristics. Commonly, objects that fit within a leaf node region are stored therein, whereas objects that cross several regions are either redundantly stored on each leaf or stored on the deepest shared parent. In addition, for consistency, these placements must be updated every frame. As regards the tree updating schemes, most authors break down their solutions into operators that merge, split and alter nodes/subtrees and a core algorithm to decide when to use each operator. Finally, to find overlaps, each leaf node presents an independent broad-phase problem, which can be handled by simpler algorithms, such as the BF. If objects are allowed to reside on internal nodes, a traversal is necessary to include these objects within the relevant sub-problems.

From an algorithm design perspective Space partitioning trees provide a flexible way to partition the space and an intuitive hierarchical representation. In comparison to Grids, these structures adapt to the workload more finely, at the expense of more costly updating algorithms. As to BVHs, these are much easier to build and maintain, but do not prune so well. A major design consideration for such trees is their height, which should be carefully chosen in order to avoid under- or over-pruning. In this regard, some authors use the SAP algorithm as an operator, allowing shallower trees to be used, while some others use variable-depth trees, generally focusing on a maximum number of objects per leaf target. It is noteworthy that these techniques are not mutually exclusive.

State-of-the-art Two major proponents of space partitioning trees to the broad phase are Luque *et al.* [LCF05] and Serpa and Rodrigues [SR17]. Specifically, Luque *et al.* use a BSP tree with five update operators, partial updates, object redundancy and the SAP algorithm as operator. In addition, to mitigate the prohibitive cost of a full BSP-tree update, these authors describe a scheduled

semi-adjusting approach that defers operators to be used over the course of several frames [LCF05]. Contrastingly, Serpa and Rodrigues use a KD-Tree structure, six light operators, full updates per frame, objects at internal nodes and an SIMD-optimized SAP algorithm as operator. In addition, these latter authors also highlight the benefits of the KD-Tree plus SAP approach, its impact on the performance of the update algorithm and the need for more generic solutions to the broad-phase problem, capable to cope with both static and dynamic scenes competitively [SR17].

4. Testing Framework

In this section, we describe the major aims of the open source testing framework we have developed, named Broadmark. It is focused on providing a common foundation to test and analyse broad-phase collision algorithms, validate and measure these solutions, as well as developing test scenarios, test cases and test scenes to be used as benchmarks. Moreover, Broadmark has been conceived to be extendable, for example with new algorithms and new scenes. Basically, our main goal with Broadmark is to foster the research of new broad-phase algorithms and provide standardized data testing to be used by other researchers in future work, when performing new tests with new algorithms.

The complete system is divided into two main components: the *scenario generator* and the *broadmark system*. The former is responsible for generating 3D scenes and baking them to disc, while the latter is responsible for running broad-phase collision detection algorithms on baked scenes. In the following subsections, we describe these both components and present implementation details, as well as their current limitations.

4.1. Simulation generator

The simulation generator tool is developed in the Unity Engine as an Unity Project, using the *C#* language, and is aimed at creating 3D scenes with several thousand dynamic objects to stress the broad-phase algorithms performance. In the following, we highlight the main aspects of the system and the rationale behind them.

Physics engine independence The system has a physics engine/body interface that completely separates it from the underlying engine. Currently, we support the *PhysX* and *Bullet* engines, for physically based simulation; a *Viewer* engine, used to replay generated scenes; and a *Splitter* engine, for creating large simulations out of smaller ones.

Object types Out-of-the-box, we provide spheres, cubes and assorted objects, being the last a set of shapes, such as bars, planks, squares and random cubes. These object types can be used to validate/stress the algorithms according to their geometrical complexity. At start-up, these objects are initialized at random positions within the scene and with random initial velocities.

Built-in scenes We include three built-in scenes, named *Free Fall*, *Brownian* and *Gravity*, which, respectively, aim at modeling a static/predictable scene, a uniformly distributed chaos and a completely dynamic scene with many collisions. Some of the resulting frames of the simulations are shown in Figure 2. These scenes have been shown in our previous work [SR17] to be an effective benchmark to investigate the algorithms' performance and their plausibility as a general solution. In more detail, the *Free Fall* scene merely lets each object follow its path, colliding with other objects and eventually reaching the scene's floor. At the end of the simulation, all objects are static. In contrast, the *Brownian* scene interferes with the objects velocities, assigning new ones periodically. Therefore, no object ever reaches a resting state. Finally, the *Gravity* scene acts on the gravity vector, rotating it smoothly over the scene's duration, forcing the objects to travel around the scene in a predictable manner while constantly clashing against each other.

Parametric generation Each scene is fed with a *SimulationParameters* object that contains data such as the number of objects, number of frames, time-step size, number of substeps, objects type and density, among other parameters. This greatly allows customization. In particular, special attention must be given to the objects density, as it controls the relative size of the world in relation to the objects. This mitigates the influence of the number of objects on the resulting motion. Figure 3 shows, for several number of objects, the same camera frame of the *Gravity* scene, demonstrating overall the same motion/behaviour. Parameters, such as the initial position, speed and gravity, are also scaled with respect to the objects density, further mitigating the effects of changing the number of objects on the generated motion.

Baked results During the execution, scenes are saved to disc in a simple binary format (*.aabbs*), which stores the scene's name, objects type, number of objects, number of frames and the AABB of each object for each simulation frame. This fully decouples the simulation generation from the algorithms execution and allows for other authors to develop their own simulation generation tools, while still being able to communicate with the broadmark system. Additionally, scenes are also recorded in a similar format (*.posrot*), which stores the position and rotation of each object instead of its AABBs. This alternative format can be used to replay the original simulation within Unity and is useful for debugging and movie recording.

Generation wizard The system comes with a generation wizard to help users generate baked results for several setups in a batch manner, for instance baking the *Free Fall* and *Brownian* scenes for 1, 2, 4, 8, 16 and 32000 cubes and spheres. For convenience, we also package a pre-built version of this tool to aid non-Unity users to bake built-in scenes.

4.2. Broadmark system

The broadmark system is our key component to run and measure algorithms. It bundles several algorithms (which, in turn, are divided into algorithm families) and implements the benchmark functionality needed to analyse them. The system uses as input a parameters file (which defines the algorithm) and a simulation test scene. It outputs the results obtained after running the entire process in form of a per-frame summary of the execution, including the time spent updating objects and structures, as well as finding collisions. For automation, we have developed a small *Python* package to batch the execution of algorithms, parse the results and generate data plots.

Currently, the system bundles the following 13 algorithm families: BF, SAP, Grid BF, Grid SAP, AxisSweep, DBVT F, DBVT D,

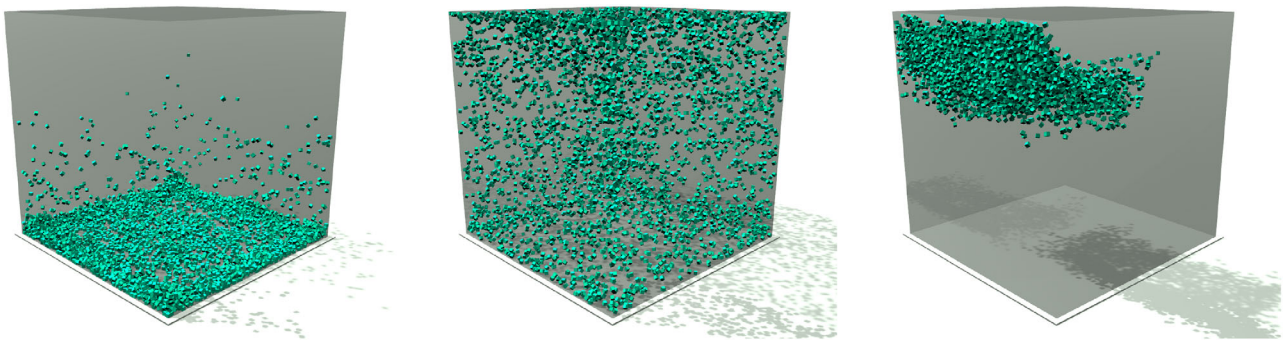


Figure 2: *From left to right, respectively, the built-in scenes developed to benchmark algorithms:* Free-Fall*,* Brownian *and* Gravity.

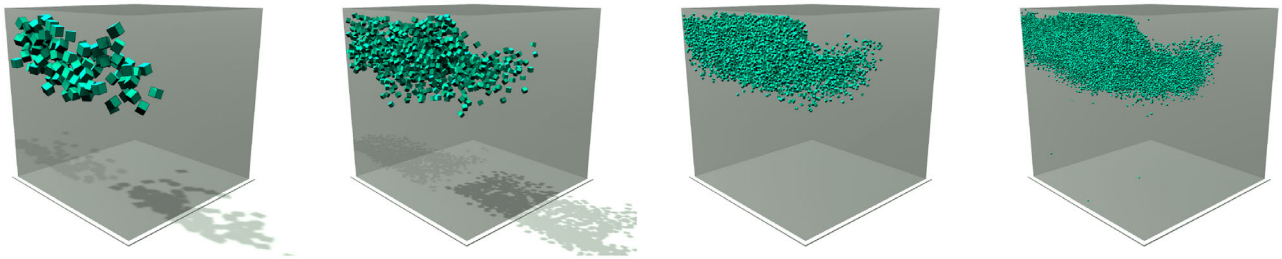


Figure 3: *From left to right, respectively, constant object densities on the* Gravity *scene for 100 – 1000 – 8000 and 32000 objects.*

CGAL, Tracy, KD-Tree, GPU Grid, GPU LBVH, and GPU SAP. Of those, the BF, SAP and Grid families are original implementations, the AxisSweep and DBVT solutions came from the Bullet 2 library, the Tracy and KD-Tree implementations were provided by the own authors, the CGAL algorithm was taken from the CGAL Library and the remaining three GPU algorithms were taken from the Bullet 3 OpenCL branch.

In the following, we briefly describe each algorithm family, stressing their relationship to the canonical algorithms aforementioned in Section 3. Table 1 summarizes the key characteristics of each algorithm.

- **BF:** A baseline implementation, closely matching the description in Section 3.1. This algorithm comes in four variants: standard, SIMD, multi-threaded (MT) and SIMD + MT. For the SIMD version, we store objects following a structure-of-arrays (SoA) memory layout and perform up to eight AABB tests in parallel, using AVX instructions. For the MT version, we define to each thread an individual collision cache and roughly the same number of AABB tests to perform. The implementation is based on OpenMP and is compatible with the SIMD implementation, allowing an SIMD + MT version.
- **SAP:** A simple, STL-sort based, SAP implementation, as described in Section 3.2. As the BF algorithm, this method also comes in four variants: standard, SIMD, MT, and SIMD + MT, using the same optimizations as the BF implementation with the addition of the STL Parallel Sort, for the MT variants.
- **Grid BF and Grid SAP:** A baseline Grid implementation. We follow the standard approach where each object may occupy more than one cell, as mentioned in Section 3.4. For the collision detection, we provide BF-based and SAP-based implementations, both allowing redundant collisions to be found. For each variant, we also include a parallel version of the algorithm, exploiting the cell-level independence. As for the previous algorithms, each thread has its own collision cache and the workload is balanced by assigning each thread a cell range with roughly the same number of objects.
- **AxisSweep and DBVT (F and D):** Incremental-SAP and BVH-based broad-phase algorithms taken from the Bullet 2 library [Cou18]. The former closely follows the iSAP description described in Section 3.3, with the addition of quantized coordinates to avoid floating-point comparisons, and a hash-map-based collisions cache implementation. The latter uses a BVH tree, which is incrementally built and optimized via re-insertions. This algorithm detects collisions using either a *Forward* approach, which is based on doing object-tree queries for objects that moved above a certain threshold, or a *Deferred* one, based on a single tree-tree query. We refer to both variants as DBVT F and DBVT D, respectively.
- **CGAL:** A range-tree, interval-tree and SAP hybrid algorithm designed to handle massive and dynamic scenes. Originally proposed by Zomorodian and Edelsbrunner [ZE00] and carefully implemented in the *Computational Geometry Algorithms Library*'s (CGAL) "Intersecting Sequences of dD Iso-oriented Boxes" package [KMZ16]. It relates to the SAP algorithm described in Section 3.2.

- **Tracy:** A Grid and iSAP hybrid algorithm designed to handle most of the static scenes. When updating objects, a custom segmented-list structure is used to ease the migration of objects between Grid cells. In this work, we named this algorithm Tracy, in honour of its author's name [TBW09].
- **KD-Tree:** A SIMD-optimized KD-Tree and SAP hybrid algorithm to better handle static scenes, which is based on a novel memory layout, an efficient two-pass tree update algorithm and an adaptive incremental collision detection [SR17]. Its focus on being both scalable and general-purpose.
- **GPU Grid, GPU LBVH and GPU SAP:** Implementations taken from the Bullet 3 Library OpenCL branch. These algorithms include a 3D Grid, a Linear-BVH [LGS*09] and a simple SAP algorithm, similar to the work of Liu *et al.* [LHLK10]. Collectively, they have been implemented using the OpenCL language and feature optimizations commonly found in GPU algorithms, such as the use of Radix Sort as the sorting algorithm, Morton Codes to extract proximity information and parallel primitives, such as map, reduce and prefix-sum.

4.3. Extending broadmark

In this section, we highlight three major directions in which the system can be easily extended by other authors. In order of importance, (1) new algorithms could be developed using our framework or be integrated into it, allowing for comparative studies to be conducted; (2) new scenes could be included, introducing other behaviour types or simulating specific applications; and (3) new engines could be added, such as new or improved physics packages or non-physical algorithms. In the following, we develop on the specific steps needed for each of those additions. Figure 4 shows the Broadmark system architecture and data flow.

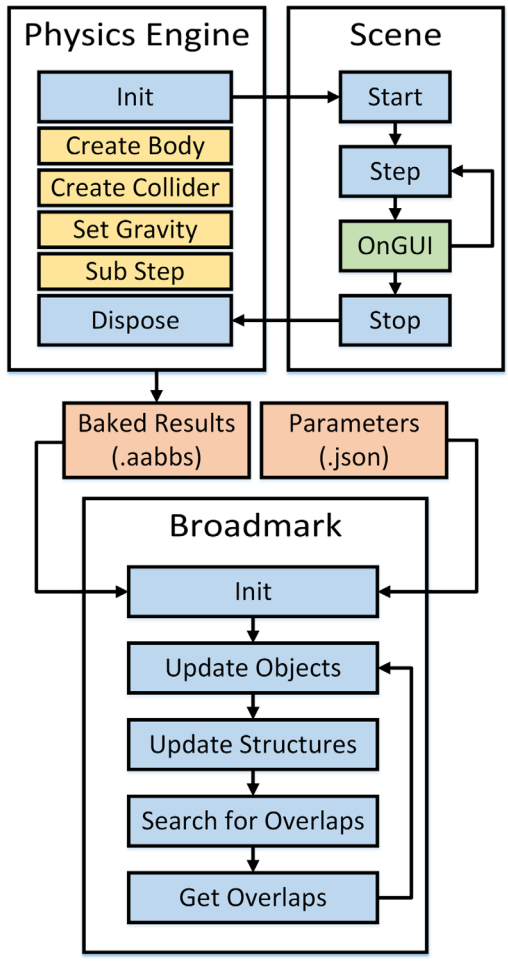


Figure 4: *The system flowchart, including its major components and dataflow. Nodes are colour-coded to indicate the data flow (*blue*), engine methods (*yellow*), Broadmark inputs (*orange*) and utilities (*green*).*

New algorithms Within the Broadmark system, the *BroadphaseInterface* is the basic API for all broad-phase algorithms and is the interface used by the system to communicate with each solution. The interface comprises virtual methods, such as *Initialize*, *Update Objects*, *Update Structures*, *Search for Overlaps* and *GetOverlaps*. The *Initialize* method is used to pass tuning parameters, simulation meta-data and initial AABBs to algorithms, the *Update Objects* receives a contiguous array with new AABBs and the *GetOverlaps* method has to return a valid *PairCache* pointer, containing the algorithm's results. This set of methods is the backbone of any broad-phase algorithm and mimics the usual setting in which these algorithms are applied. To integrate a new algorithm, it has to be registered in the *Algorithms.cpp* file.

Inheriting the *BroadphaseInterface* directly is the lowest level alternative available. On top of it, we include a *BaseAlgorithm* class, which handles storing and updating objects, as well as storing a *PairCache*. This largely reduces the amount of boilerplate code and, for that reason, is the recommended starting point for adding new solutions. For maximum flexibility, the *Object* and *PairCache* classes are template arguments. In the case of *SIMD* optimized solutions, a third option exists, named *SoAAlgorithm*. This class augments the *BaseAlgorithm* class by also storing the Objects' AABBs in a *Structure-of-Arrays* layout, which is convenient for data parallel solutions.

New scenes Within the *Unity* portion of the system, the *Simulation* class defines the interface for new scenes. In total, five virtual methods are defined: *Start*, *Step*, *Stop*, *Dispose* and *OnGUI*. The first three methods are dedicated to handle the life-cycle of the simulation, the fourth is meant for any unmanaged resources disposal and the last one is *Unity* specific and allows for custom GUI code to be added during simulation generation, which can be handy to visualize internal data or to implement user interactions with the simulation. It is worth mentioning that every simulated object is an independent *Unity* object and, thus, can have specific scripts attached to it at any moment.

New engines To simulate non-physical applications, implementing a new engine might be interesting. For instance, an agent planning or a bird-oid objects (BOIDS) simulator could be implemented on our system. To do so, two interfaces must be inherited: *PhysicsEngine* and *PhysicsBody*. The former exposes methods to control the life-cycle of the simulated world and the creation of bodies and static colliders, while the latter exposes object manipulation

Table 1: *Summary of the bundled algorithms.*

	IMPLEMENTATION					COMPLEXITY	
Algorithms	Principle	Optimizations	Incremental	Remarks	Source	Time	Space
BF	BF	SIMD + MT	–	Naive	Original	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
SAP	SAP	SIMD + MT	–	STL Sort	Original	$\mathcal{O}(n^{5/3})$	$\mathcal{O}(1)$
Grid BF	Grid	MT	–	T objects/cell	Original	$\mathcal{O}(n^2/t)$	$\mathcal{O}(nt)$
Grid SAP	Grid + SAP	MT	–	T objects/cell	Original	$\mathcal{O}(n^2/t)$	$\mathcal{O}(nt)$
AxisSweep	iSAP	–	Yes	Insertion Sort	Bullet 2	$\mathcal{O}(n+s)$	$\mathcal{O}(n)$
DBVT F	BVH	–	Yes	Persistent Tree	Bullet 2	$\mathcal{O}(nlog(n))$	$\mathcal{O}(n)$
DBVT D	BVH	–	–	Persistent Tree	Bullet 2	$\mathcal{O}(nlog(n))$	$\mathcal{O}(n)$
CGAL	Tree + SAP	–	–	Stateless	CGAL	$\mathcal{O}(nlog^3(n))$	$\mathcal{O}(n)$
Tracy	Grid + iSAP	MT	Yes	Insertion Sort	Authors	$\mathcal{O}(n+s)$	$\mathcal{O}(n)$
KD-Tree	Tree + SAP	SIMD	Yes	Adaptive, Persistent Tree	Authors	$\mathcal{O}(nlog(n))$	$\mathcal{O}(n)$
GPU Grid	Grid	GPU	–	OpenCL	Bullet 3	N/A	N/A
GPU LBVH	BVH	GPU	–	OpenCL	Bullet 3	N/A	N/A
GPU SAP	SAP	GPU	–	OpenCL	Bullet 3	N/A	N/A

Time complexity was derived considering a uniformly distributed set of boxes.

routines, such as changing its speed, position and activation state. We highlight that new engines are an effective way to create development tools. For instance, our simulation viewer is implemented as an engine that creates and moves objects according to a baked simulation data file.

4.4. Limitations

The current version of our simulation generator and Broadmark system is feature complete and covers most situations that arise in practice. However, some use cases are not yet covered by our framework. Primarily, the simulation generator is currently limited to a fixed number of objects per simulation and the Broadmark system supports only AABBs, one of the most widely used BVs, which are the industry standard [Cou18, Nvi19a, Hav19]. While support for variable number of objects and other input shapes could be added in the future, these features would significantly increase the system complexity, especially the latter.

Regarding the broad-phase precision, all bundled algorithms are complete, i.e. they report all colliding AABB pairs. However, some solutions may also report some non-colliding pairs, namely, the AxisSweep and KD-Tree algorithms, which use slightly enlarged AABBs and, thus, over-report pairs. In this context, the Broadmark system cannot measure the negative impact of over-reporting, as it is restricted to the broad phase. The system can, however, quantify the amount of over-reporting, for instance, the aforementioned solutions detect 5% more pairs, in average. These data can be used, in turn, as a rough estimate for the extra time that will be spent at the narrow phase.

Specifically to game-like applications, broad-phase structures are commonly used to accelerate other tasks, such as ray-casts, boxcasts, sphere-sweeps and others. However, superior performance can often be achieved using dedicated acceleration structures to process such queries [Nvi19a]. Furthermore, decoupling the simulation code from the scene query system allows both components to run asynchronously, which better exploits modern parallel hardware. For this reason, we have not included support for scene queries benchmark in our system.

5. Comparison of Performance and Behaviour of Algorithms

In this section, we present the methods, analyse and discuss the results of testing the performance and behaviour of the algorithms described in Section 4.2, on the following benchmark scenarios: *Brownian Scene*, *Free Fall* and *Gravity*. Under a uniform distribution, the *Brownian Scene* illustrates a typical case useful for testing the overall performance of the algorithms. To examine the algorithms behaviour as the objects stand still, we used the *Free Fall* scene. Finally, the *Gravity* scene was chosen to stress test the algorithms. It is important to note that our system also offers the possibility of generating simulations with assorted objects (e.g. varied sized boxes and spheres); however, in this work, we opted not to include these scenarios, since in a previous work [SR17] we have shown that this addition impacts most algorithms similarly and, thus, provides little additional insight. In the previous testing algorithms, the DBVT solution (which is BVH based) was the only tested solution to show a significantly reduced performance on these scenes, taking two to three times longer in these cases, while the other tested solutions, i.e. KD, Tracy and CGAL, took only 0.2 to 0.4 times longer.

To focus on the most competitive solutions, we chose to cap our analysis to the 0.5 s/frame mark. Any algorithm performing beyond this point can be safely disregarded as competitive. In Table 2, we show the results for all algorithms in each scenario, up to the first configuration that crossed the 0.5 s mark. For completeness, we present data starting from 1000 objects, despite it being 0.00 for all algorithms, as it shows that, for practical reasons, all algorithms perform similarly in this range. Out of all algorithms, the GPU SAP was the only one to exhibit problems: it failed to run for more than 512,000 objects. For that reason, we have not included it on the performance analysis. However, just

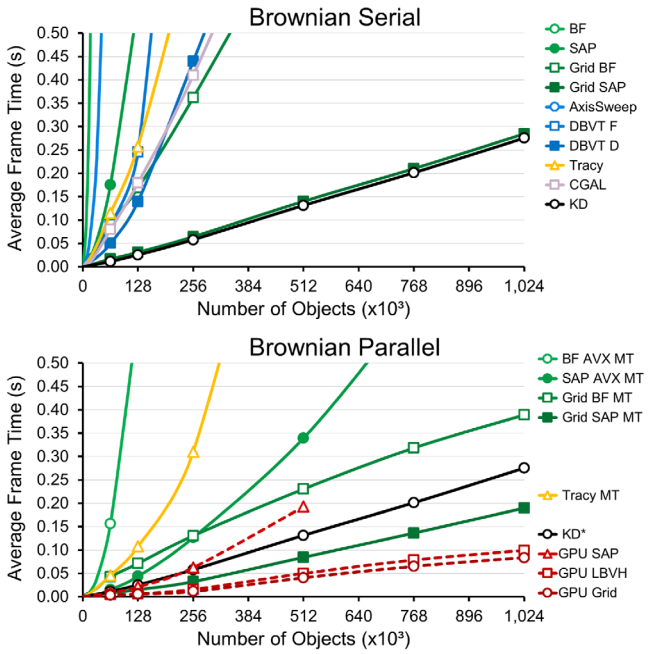


Figure 5: *Scalability results for all algorithms (single-threaded, multi-threaded and GPU), under the* Brownian *scene for one million objects.*

for comparative reference, we opted to include it in the plots and tables.

In all plots shown in Figures 5–7, we assigned a semantic colour/style scheme to each of the solutions to help identify and relate algorithms. In short, colours indicate the algorithms source, marker shapes identify algorithms, unfilled and solid marker shapes relate to algorithm variants and continuous and dashed lines, respectively, indicate whether the solution is CPU or GPU based. For instance, the Grid BF and Grid SAP algorithms are original implementations (depicted in green), CPU based (displayed as continuous lines) and share the same square marker shape, with one being unfilled (Grid BF) and the other solid (Grid SAP). On the other hand, all three GPU algorithms come from the Bullet 3 OpenCL branch (red coloured), are GPU based (depicted as dashed lines) and correspond to distinct algorithms, being graphically represented by three different marker shapes (triangle, square and circle). The colouring schema used for drawing the line plots were defined as follows: *green* represents original implementations, *blue* and *red,* respectively, stand for algorithms available in the Bullet 2 and 3 libraries and the remaining colours (*yellow*, *lavender* and *black*) indicate independently sourced algorithms from the literature.

In Figures 5–7, the single-threaded algorithm with the best performance was included among the parallel ones to ease performance comparison.

All results presented in this section were gathered on a *Intel Core i7 7700k* machine, with *32GB* of *RAM* and a *Nvidia GTX 1060* GPU with 1759 MHz and 1280 threads. For multi-threaded algorithms, four threads were used, mapped to the four physical cores of the CPU. With regard to time measurements, all broad-phase steps were accounted for, such as the time updating objects and structures, sending/receiving data from the GPU, as well as searching for collisions and managing the collisions cache. In particular, no step outside the broad-phase spectrum was measured or included in the measurements. Finally, to be statistically significant, all presented values are the average over 1000 execution frames.

In the following, we present the per-scene analysis from 1000 to one million objects. Table 2 shows the numerical data points used to draw each line plot.

5.1. Brownian tests

Figure 5 shows the scalability of all algorithms for the *Brownian* scene divided into serial and parallel solutions. In performance order, on the single-threaded plot, the most basic approaches, i.e. the BF and SAP algorithms, exhibit the worst performance for this benchmark. In the multi-threaded plot, we show that these both algorithms perform much better when using AVX instructions and CPU threads, reaching up to 40 and 12 times the performance of their respective single-threaded implementations. In sequence, the DBVT F and Tracy algorithms are the third and fourth least performing algorithms for this benchmark, which is justifiable by their incremental nature, which benefits from static scenes. Appearing next, the DBVT D and CGAL algorithms, which benefit from dynamic scenes, are up to three times faster than the previous discussed incremental algorithms. Out of these, the Tracy implementation, particularly, has a multi-threaded variant that achieves a speed up of 2.2 times over its serial variant. In sequence, the Grid and Grid SAP algorithms illustrate the performance gain when using the SAP algorithm as an operator, being the Grid SAP, on average, times faster than the Grid BF approach. On the multi-threaded plot, both algorithms are two to three times faster due to their parallel execution. The KD CPU algorithm shows similar results to the single-threaded Grid SAP solution. Finally, the GPU Grid and the GPU LBVH algorithms reach the best results among all tested algorithms for this benchmark.

Comparing algorithms by hardware on this benchmark, in Figure 5 (the bottom plot), we show that the best GPU algorithms (i.e. GPU LBVH and GPU Grid) are only two times faster than the best multi-threaded algorithm (Grid SAP MT) and only three times faster than the best single threaded algorithm (KD). In light of this, we believe that there is still plenty of room for improvement on the multi-threaded and GPU algorithms. Assuming a hypothetical parallel KD implementation with perfect scaling, it would only take three threads to match the best results obtained on GPU. In addition, the BF and SAP implementations also provide further evidence that multi-core optimizations are still far unexplored. Using four cores and eight AVX lanes, up to 40 and 12 times gains were obtained on the BF and SAP algorithms, respectively. This leads us to conclude that custom tailored algorithms for SIMD and multi-core parallelization might achieve similar gains. In contrast, the KD algorithm, which is only partly SIMD optimized, becomes only 0.3 times faster when using such instructions [SR17].

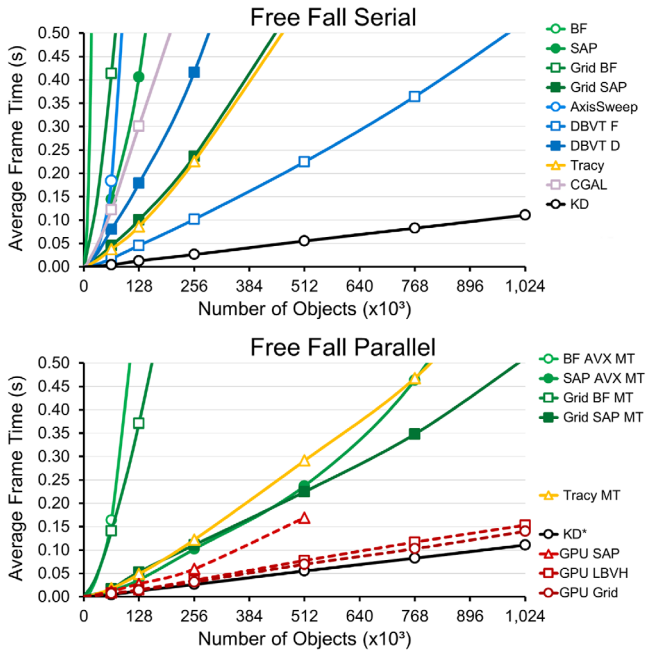


Figure 6: *Scalability results for all algorithms (single-threaded, multi-threaded and GPU), under the* Free Fall *scene for one million objects.*

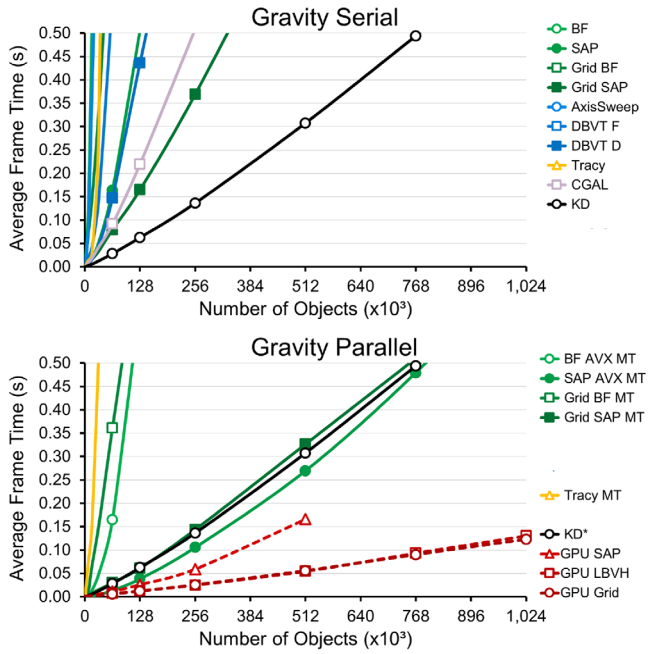


Figure 7: *Scalability results for all algorithms (single-threaded, multi-threaded and GPU) under the* Gravity *scene for one million objects.*

5.2. Free fall tests

As can be seen in Figure 6, unlike the previous scene, the *Free Fall* shows a mostly static behaviour. This greatly favours the algorithms with temporal optimizations (AxisSweep, Tracy, DBVT F and KD). Moreover, most objects lie densely packed on the scene's floor, which undermines the potential of the Grid-based solutions (Grid BF, Grid SAP and Tracy). Overall, this plot shows significantly different results for most algorithms. Some of them thrived (AxisSweep, Tracy, DBVT F and KD), some were negatively affected (Grid BF, Grid SAP, CGAL and DBVT D) and others did not deviate significantly (BF, SAP and all three GPU solutions). When comparing algorithms, we note that the DBVT F and KD have benefited the most from this scenario. In addition, we obtained an interesting behaviour for some algorithms: the DBVT F (previously, one of the least performing solutions), in this scene has reached the one million objects mark; and the KD algorithm currently outperforms even GPU algorithms, despite being single threaded.

These findings stress, once more, the importance of exploiting temporal optimizations on static scenes. Moreover, the *Free Fall* simulation highlights some key features regarding the possible benefits of including temporal optimizations in broad-phase algorithms, although more data are needed to further illuminate this interesting behaviour. As mentioned by Serpa and Rodrigues, many algorithms are either fully focused on static scenes (e.g. AxisSweep) or completely disregard it (e.g. CGAL), being few of them focused on being applicable to both (e.g. KD and DBVT) [SR17]. This analysis also sheds some light on the perceived generality by force of some algorithms. For instance, both GPU Grid and GPU LBVH are still very performant on this scene, despite being void of any temporal optimization. This suggests that their performance comes mostly from sheer pair-wise testing power than from their pruning strategy. While this may seem negative, it is, nonetheless, a form of generality and should be noted as a positive feature. In fact, for hard to prune scenes, this generality by force might be the sole feasible solution for an efficient broad phase.

5.3. Gravity tests

The *Gravity* scene is the hardest benchmark for all algorithms, posing challenges to all the methods we have tested. SAP-based algorithms are less effective due to densely packed objects, iSAP alternatives have no coherence to exploit for sorting performance, Grids perform poorly under highly clustered settings and, finally, tree structures have to deal with massive changes to the objects positions in each frame. Figure 7 shows severely reduced timings for almost all algorithms, with the exception of the GPU algorithms. Actually, no single or multi-threaded algorithm reached the one million objects mark. This by itself is a proof that the *Gravity* scene is challenging and that it can be effectively used as a practical upper bound for stress-testing broad-phase collision detection algorithms. With regard to the GPU-based Grid and LBVH algorithms, this scene further reinforces the generality by force explanation. Numerically, these algorithms perform similarly on all three benchmarks, despite the many differences (e.g. static, dynamic, uniform and non-uniform distributions) among the benchmark scenarios.

Table 2: *Numerical data for all presented plots.*

BROWNIAN

	BF		SAP		Grid BF		Grid SAP		Axis	DBVT		Tracy				GPU		
# objects	ST	MT	ST	MT	ST	MT	ST	MT	Sweep	F	D	ST	MT	CGAL	KD	SAP	LBVH	Grid
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8	0.11	0.00	0.01	0.00	0.02	0.01	0.00	0.00	0.02	0.01	0.00	0.01	0.00	0.01	0.00	0.00	0.00	0.00
16	0.42	0.01	0.02	0.00	0.03	0.01	0.00	0.00	0.06	0.01	0.01	0.02	0.01	0.01	0.00	0.00	0.00	0.00
32	1.60	0.04	0.06	0.01	0.04	0.02	0.01	0.00	0.23	0.04	0.02	0.05	0.02	0.03	0.01	0.00	0.00	0.00
64		0.16	0.18	0.02	0.09	0.04	0.02	0.01	1.09	0.09	0.05	0.12	0.05	0.08	0.01	0.01	0.00	0.00
128		0.60	0.57	0.04	0.17	0.07	0.03	0.02		0.25	0.14	0.26	0.11	0.18	0.03	0.02	0.01	0.01
256				0.13	0.36	0.13	0.06	0.03		1.42	0.44	0.70	0.31	0.41	0.06	0.06	0.02	0.01
512				0.34	0.78	0.23	0.14	0.08			1.04		1.19	0.95	0.13	0.19	0.05	0.04
768				0.63		0.32	0.21	0.14							0.20	N/A	0.08	0.07
1,024						0.39	0.28	0.19							0.28		0.10	0.08

FREE FALL

	BF		SAP		Grid BF		Grid SAP		Axis	DBVT		Tracy				GPU		
# objects	ST	MT	ST	MT	ST	MT	ST	MT	Sweep	F	D	ST	MT	CGAL	KD	SAP	LBVH	Grid
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8	0.11	0.00	0.01	0.00	0.04	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00
16	0.42	0.01	0.01	0.00	0.07	0.02	0.01	0.00	0.01	0.00	0.01	0.01	0.00	0.02	0.00	0.00	0.00	0.00
32	1.65	0.05	0.04	0.00	0.15	0.05	0.02	0.01	0.04	0.01	0.03	0.01	0.01	0.05	0.00	0.01	0.00	0.00
64		0.16	0.14	0.01	0.41	0.14	0.05	0.02	0.18	0.02	0.08	0.04	0.02	0.12	0.00	0.01	0.01	0.01
128		0.68	0.41	0.04	1.05	0.37	0.10	0.05	1.09	0.05	0.18	0.09	0.05	0.30	0.01	0.03	0.02	0.01
256			1.30	0.10		0.94	0.24	0.11		0.10	0.42	0.23	0.12	0.65	0.03	0.06	0.04	0.03
512				0.24			0.60	0.22		0.22	1.09	0.57	0.29		0.06	0.16	0.08	0.07
768				0.46				0.35		0.36			0.47		0.08	N/A	0.12	0.10
1,024				0.85				0.51		0.52			0.71		0.11		0.15	0.14

GRAVITY

	BF		SAP		Grid BF		Grid SAP		Axis	DBVT		Tracy				GPU		
# objects	ST	MT	ST	MT	ST	MT	ST	MT	Sweep	F	D	ST	MT	CGAL	KD	SAP	LBVH	Grid
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8	0.13	0.00	0.01	0.00	0.07	0.01	0.01	0.00	0.09	0.02	0.01	0.01	0.08	0.01	0.00	0.00	0.00	0.00
16	0.51	0.01	0.02	0.00	0.14	0.06	0.02	0.01	0.36	0.04	0.02	0.03	0.15	0.02	0.01	0.00	0.00	0.00
32		0.05	0.05	0.01	0.34	0.15	0.03	0.02	0.89	0.15	0.05	0.29	0.52	0.04	0.01	0.01	0.00	0.00
64		0.17	0.16	0.01	0.84	0.36	0.08	0.03		0.57	0.15	2.90		0.09	0.03	0.01	0.01	0.01
128		0.63	0.51	0.04		0.76	0.17	0.06			0.44			0.22	0.06	0.03	0.01	0.01
256				0.11			0.37	0.14			0.99			0.51	0.14	0.06	0.02	0.03
512				0.27			0.82	0.33							0.31	0.17	0.05	0.06
768				0.48				0.51							0.49	N/A	0.09	0.09
1,024				0.72											0.68		0.13	0.12

All measurements are in seconds and the number of objects are in thousands.

6. Conclusion and Future Work

In this work, we present to the collision detection community two major contributions: (1) a comprehensive summary of the main techniques for the development of broad-phase collision detection algorithms, and (2) a benchmark system for the broad-phase collision detection problem.

Many reasons make us believe that this work can be very useful and especially used as ground base for future research in the collision detection area: it provides in-depth insights on the usage of many common and well-known broad-phase collision detection techniques; it extensively refers to and comments on the relevant literature; it includes the *broadmark* system, which assembles many pre-existing and some original implementations into a single, easy-to-use, framework; and it presents an extensible and feature-rich testing environment, which can be reused on future works to help standardize comparisons.

Through use of the *broadmark* system, we could evaluate the current state-of-the-art for single, multi and many threaded algorithms, and compare the implemented solutions among each other.

Notably, the *Brownian* scene unveiled that the best GPU implementations are only two and three times faster than the best parallel and serial CPU implementations, respectively, which suggests that there is still plenty of room for improvement on the GPU side. In addition, it illustrates the performance divide between incremental and non-incremental algorithms on a dynamic scene, as well as between the BF and SAP algorithms used as complete solutions and as operators of a high-level pruning structure. The results obtained in the benchmark scenarios *Free Fall* and *Gravity* clearly show the available opportunities for improvement from temporal optimizations on static scenes and the amount of performance degradation possible when objects are densely packed and move continuously. These two scenes also provide evidence, which suggests that GPU algorithms can be general-purpose by force, instead of by design. This feature should be further investigated as a possible solution for hard to prune scenes, such as the *Gravity* one. Altogether, these scenes demonstrate the plurality of broad-phase collision detection algorithms and the many variables affecting their performance. We firmly believe that novel broad-phase algorithms should be designed with these ideas in mind, being conceptualized with not just one target application, but multiple contrasting scenarios.

As future work, we plan to develop novel multi-threaded and GPU algorithms to further advance the field and support even larger simulations. Moreover, we aim to investigate the effective use of SIMD and multi-threading on the broad-phase problem, focusing on achieving scalable results regarding the number of threads and SIMD lanes. Finally, further research is also required to explore the mixed use of the CPU and GPU and could prove quite beneficial to achieve a greater level of generality to algorithms, exploiting both the generality by design (possible on the CPU) and by force (possible on the GPU). Regarding our framework, we plan to expand it in order to support variable number of objects, continuous collision detection and other use cases related to the broad-phase problem.

Acknowledgements

Ygor Rebouças Serpa and Maria Andréia Formico Rodrigues would like to thank the Brazilian Agencies CAPES/FUNCAP and CNPq for their financial support, under grants 88887.176617/2018-00 and 439067/2018-9, respectively. We are also grateful to the referees for providing insightful comments and suggestions to improve the manuscript.

References

[AGA10] Avril Q., Gouranton V., Arnaldi B.: A broad phase collision detection algorithm adapted to multi-cores architectures. In *Proceedings of the 2010 Virtual Reality International Conference* (Laval, France, 2010), Laval Virtual, pp. 95–101.

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

[AGA14] Avril Q., Gouranton V., Arnaldi B.: Collision detection: Broad phase adaptation from multi-core to multi-GPU architecture. *Journal of Virtual Reality and Broadcasting 6*, 11 (2014), 1–13.

[BMPS09] Batista V. H., Millman D. L., Pion S., Singler J.: Parallel geometric algorithms for multi-core computers. In *Proceedings of the 25*th *Annual Symposium on Computational Geometry* (Aarhus, Denmark, 2009), ACM, pp. 217–226.

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

[CL16a] Capannini G., Larsson T.: Efficient collision culling by a succinct bi-dimensional Sweep and Prune algorithm. In *Proceedings of 32nd Spring Conference on Computer Graphics* (Smolenice Castle, Slovakia, 2016), ACM, pp. 25–32.

[CL16b] Capannini G., Larsson T.: Output sensitive collision detection for unisize boxes. In *Proceedings of the 2016 Swedish Chapter of Eurographics (SIGRAD)* (Visby, Sweden, 2016), Linköping University Electronic Press, pp. 22–27.

[CL18] Capannini G., Larsson T.: Adaptive collision culling for massive simulations by a parallel and context-aware Sweep and Prune algorithm. *IEEE Transactions on Visualization and Computer Graphics 24*, 7 (2018), 2064–2077.

[Cou08a] Coumans E.: btaxissweep3. github.com/bulletphysics/bullet3/blob/master/src/BulletCollision/BroadphaseCollision/btAxisSweep3.h (2008).

[Cou08b] Coumans E.: btdbvtbroadphase. github.com/bulletphysics/bullet3/blob/master/src/BulletCollision/BroadphaseCollision/btDbvtBroadphase.h (2008).

[Cou14a] Coumans E.: b3gpuparallellinearbvhbroadphase. github.com/bulletphysics/bullet3/blob/master/src/Bullet3OpenCL/BroadphaseCollision/b3GpuParallelLinearBvhBroadphase.h (2014).

[Cou14b] Coumans E.: b3gridgpubroadphase. https://github.com/bulletphysics/bullet3/blob/master/src/Bullet3Collision/BroadPhaseCollision/b3DynamicBvhBroadphase.h (2014).

[Cou18] Coumans E.: Bullet Physics library 2.88. github.com/bulletphysics/bullet3 (2018).

[DSC05] Daniel S., Coming O. G. S.: Kinetic Sweep and Prune for collision detection. In *Proceedings of the* 2*nd Workshop in Virtual Reality Interactions and Physical Simulations* (Pisa, Italy, 2005), The Eurographics Association, pp. 1–10.

[Eri04] Ericson C.: *Real-Time Collision Detection*. CRC Press, 2004.

[FM17] Franklin W. R., Magalhães S. V.: Parallel intersection detection in massive sets of cubes. In *Proceedings of the* 6*th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial)* (Redondo Beach, CA, USA, 2017), ACM, pp. 20–26.

[GTT13] Geleri F., Tosun O., Topcuoglu H.: Parallelizing broad phase collision detection algorithms for sampling based path planners. In *Proceedings of the* 21*st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)* (Pavia, Italy, 2013), IEEE, pp. 384–391.

[Hav19] Havok: Havok Physics. havok.com/physics/ (2019).

[KMZ16] Kettner L., Meyer A., Zomorodian A.: Intersecting sequences of dD iso-oriented boxes. In *CGAL v4.13.1: User and Reference Manual*. 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 (I3D)* (Washington, DC, USA, 2005), ACM, pp. 179–186.

[LG07] Le Grand S.: Broad-phase collision detection with CUDA. In *GPU Gems 3*. Hubert Nguyen (Ed.). Addison-Wesley Professional (2007), ch. 32, pp. 697–722.

[LGS*09] Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D.: Fast BVH construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), 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 29*, 6 (2010), 1–8.

[LLCC11] Lo S.-H., Lee C.-R., Chung Y.-C., Chung I.-H.: A parallel rectangle intersection algorithm on GPU+CPU. In *Proceedings of the 2011 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (Newport Beach, CA, USA, 2011), IEEE, pp. 43–52.

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

[MCLK17] Ming C., Lin D. M., Kim Y. J.: Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*, 3rd ed. Jacob E. Goodman, Joseph O'Rourke and Csaba D. Tóth (Eds.). CRC Press (2017), ch. 39, pp. 1029–1056.

[Mir97] Mirtich B.: *Efficient Algorithms for Two-Phase Collision Detection*. Mitsubishi Electric Research Laboratory (MERL), 1997, 1–26.

[Nvi19a] Nvidia: Physx 4.1. github.com/NVIDIAGameWorks/PhysX (2019).

[Nvi19b] Nvidia: Physx 4.1 documentation: Broad phase algorithms. gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Manual/RigidBodyCollision.html#broad-phase-algorithms (2019).

[SFC*19] Stroobant P., Felicetti L., Colle D., Tavernier W., Femminella M., Reali G., Pickavet M.: Parallel algorithms for simulating interacting carriers in nanocommunication. *Nano Communication Networks 20* (2019), 20–30.

[She14] Shellshear E.: 1D sweep-and-prune self-collision detection for deforming cables. *Visual Computer 30*, 5 (2014), 553–564.

[SR17] Serpa Ygor R., Rodrigues M. A. F.: Flexible use of temporal and spatial reasoning for fast and scalable CPU broad-phase collision detection using KD-Trees. *Computer Graphics Forum 38*, 1 (2017), 1–14.

[SR18] Serpa, Yvens R., Rodrigues M. A. F.: A draw call-oriented approach for visibility of static and dynamic scenes with large number of triangles. *Visual Computer 35*, 4 (2018), 549–563.

[TB12] Tracy D. J., Brown S.: Accelerating physics in large, continuous virtual environments. *Concurrency and Computation: Practice & Experience 24*, 2 (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 *Proceedings of the 2009 IEEE Virtual Reality Conference* (Lafayette, LA, USA, 2009), IEEE, pp. 191–198.

[Ter17] Terdiman P.: Physics engine evaluation lab (PEEL). github.com/Pierre-Terdiman/PEEL (2017).

[WDM07] Woulfe M., Dingliana J., Manzke M.: Hardware accelerated broad phase collision detection for realtime simulations. In *Proceedings of the 2007 Workshop in Virtual Reality Interactions and Physical Simulations* (Dublin, Ireland, 2007), J. Dingliana and F. Ganovelli (Eds.), The Eurographics Association, pp. 1–10.

[WFZ13] Weller R., Frese U., Zachmann G.: Parallel collision detection in constant time. In *Proceedings of the 2013 Workshop on Virtual Reality Interaction and Physical Simulations* (Lille, France, 2013), J. Bender , J. Dequidt, C. Duriez, G. Zachmann (Eds.), The Eurographics Association, pp. 1–10.

[WM09] Woulfe M., Manzke M.: A framework for benchmarking interactive collision detection. In *Proceedings of the* 25*th Spring Conference on Computer Graphics* (Budmerice, Slovakia, 2009), ACM, pp. 205–212.

[WM17] Woulfe M., Manzke M.: A hybrid fixed-function and microprocessor solution for high-throughput broad-phase collision det*ection*. *EURASIP Journal on Embedded Systems 2017*, 1 (2017), 1–15.

[ZE00] Zomorodian A., Edelsbrunner H.: Fast software for box intersections. In *Proceedings of the* 16*th Annual Symposium on Computational Geometry* (Clear Water Bay, Hong Kong, 2000), ACM, pp. 129–138.