

# Time of Impact Dataset for Continuous Collision Detection and a Scalable Conservative Algorithm

DAVID BELGROD, New York University, USA

BOLUN WANG, KAUST, Saudi Arabia

ZACHARY FERGUSON and XIN ZHAO, New York University, USA

MARCO ATTENE, IMATI - CNR, Italy

DANIELE PANOZZO, New York University, USA

TESEO SCHNEIDER, University of Victoria, Canada

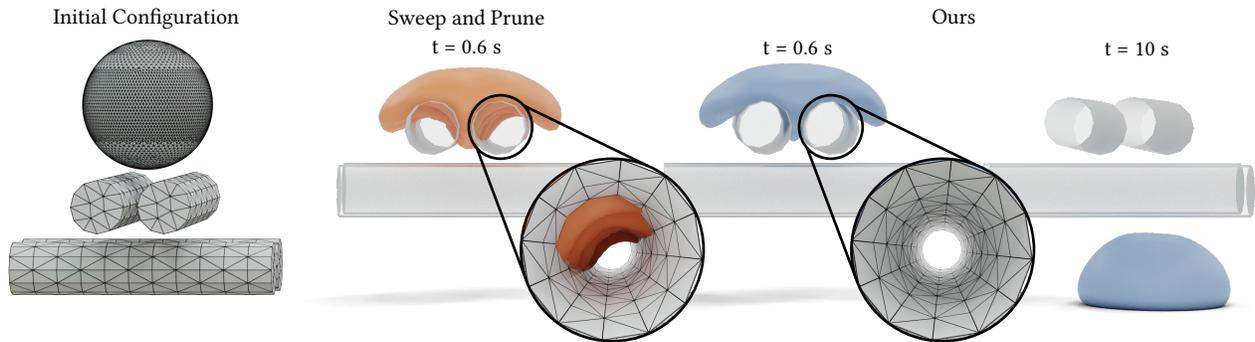


Fig. 1. An *approximate* collision detection can lead to poor simulation results. We simulate this scene using Incremental Potential Contact [Li et al. 2020] which provides an intersection-free guarantee, but approximate collision detection (either broad- or narrow-phase) can break this guarantee. Here we see using an approximate broad-phase method (sweep and prune) results in missed collisions and intersections (see inset). In contrast, a conservative or exact method allows the ball to squeeze through the rollers and come out the bottom.

We introduce a large-scale benchmark for broad- and narrow-phase continuous collision detection (CCD) over linearized trajectories with exact time of impacts and use it to evaluate the accuracy, correctness, and efficiency of 13 state-of-the-art CCD algorithms. Our analysis shows that several methods exhibit problems either in efficiency or accuracy.

To overcome these limitations, we introduce an algorithm for CCD designed to be scalable on modern parallel architectures and provably correct when implemented using floating point arithmetic. We integrate our algorithm within the Incremental Potential Contact solver [Li et al. 2021] and evaluate its impact on various simulation scenarios. Our approach includes a broad-phase CCD to quickly filter out primitives having disjoint bounding boxes and a narrow-phase CCD that establishes whether the remaining primitive pairs indeed collide. Our broad-phase algorithm is efficient and scalable thanks to the experimental observation that sweeping along a coordinate axis performs surprisingly well on modern parallel architectures. For narrow-phase CCD, we re-design the recently proposed interval-based algorithm of Wang et al. [2021] to work on massively parallel hardware.

To foster the adoption and development of future linear CCD algorithms, and to evaluate their correctness, scalability, and overall performance, we release the dataset with analytic ground truth, the implementation of all the algorithms tested, and our testing framework.

CCS Concepts: • **Computing methodologies** → **Collision detection**; *Physical simulation*.

Authors' addresses: David Belgrod, db2762@nyu.edu, New York University, USA; Bolun Wang, bolun.wang@kaust.edu.sa, KAUST, Saudi Arabia; Zachary Ferguson, zfergus@nyu.edu; Xin Zhao, xz3752@nyu.edu, New York University, USA; Marco Attene, marco.attene@cnr.it, IMATI - CNR, Italy; Daniele Panozzo, panozzo@nyu.edu, New York University, USA; Teseo Schneider, teseo@uvic.ca, University of Victoria, Canada.

Additional Key Words and Phrases: Collision Detection and Response

## 1 INTRODUCTION

Continuous collision detection (CCD) is used extensively in graphics, engineering, and scientific computing for the simulation of rigid and deformable objects, and in geometry processing to ensure that self-intersections are not introduced in parameterization or deformation applications. Objects are typically represented by triangle meshes. In this work, we focus on the common case where mesh vertices move along linear trajectories. Hence, collisions can occur either when an edge hits another edge or when a vertex hits a triangle [Wang et al. 2021].

CCD is usually divided into two steps: (1) broad-phase, which is a conservative filter that identifies candidate colliding pairs, and (2) narrow-phase, which validates each pair with an accurate and more computationally intensive algorithm. While the narrow-phase is local and involves only a pair of primitives, the broad-phase usually relies on acceleration data structures to prune unnecessary pairs and avoid the quadratic complexity of a brute-force evaluation on all possible pairs. Both problems have been extensively studied in graphics, engineering, and scientific computing in the last three decades (Section 2). An ideal linear CCD algorithm takes the start and end points of a linear trajectory and determines if, at any point along the trajectory, the geometries intersect and, if they do, it tells us at which time the first collision happens. This might seem a trivial requirement for a CCD algorithm; however, Wang et al. [2021]

show that many popular *narrow-phase* CCD implementations are *not* correct (Figure 1). Ensuring that an implementation is correct is a subtle and surprisingly difficult challenge since most of the intermediate computations use floating-point numbers, while the corresponding algorithms and their proof of correctness usually disregard floating-point rounding errors.

*Two-Pronged Evaluation Approach.* Validating the correctness of algorithms implemented in floating-point is a major challenge, especially when multiple operating systems and architectures are considered. Indeed, floating-point computations might slightly differ due to either hardware specifics or modern compilers trying to reorder operations to improve performance. This makes it challenging to have correct code and especially hard to keep it up to date as compilers and architectures evolve. To provide a way to validate an implementation on a specific system, we have collected a large dataset of CCD queries evaluated using exact computations (Section 4), and we provide a large statistical experimental validation of several methods. Differently from other CCD datasets [Serpa and Rodrigues 2020; Wang et al. 2021], we generate *ground truth time of impact* for each successive pair of frames from each scene, using a combination of symbolic computation and conservative filtering. This allows us to evaluate CCD as a whole, with broad and narrow phases coupled, and measure how conservative the methods are.

*Evaluation.* We use exact computation to evaluate the correctness of 13 broad phase algorithms on a large dataset (Section 5). To our surprise, most of the implementations are incorrect (even if the methods described in the papers are when implemented with exact arithmetic) and miss collisions, making them unusable for interior point optimization.

*Broad-Phase.* The broad-phase of a CCD algorithm usually aims at detecting collision between (axis-aligned) boxes around primitives. Several existing methods simplify the problem by either checking for collisions only at the end of the time interval (i.e., discrete collision detection) or by assuming that only a small fraction of the scene moves. These simplifications allow for faster algorithms but may lead to unrealistic results when dealing with elastic bodies. Many acceleration data structures exist (e.g., hash grids, spatial trees, bounding volumes hierarchies) to avoid checking “far away” boxes, each providing different advantages in different situations. However, most of these structures are complex to parallelize, in particular on graphics processing units (GPUs) where dynamic memory allocation is not an option. Additionally, these structures’ complexity makes it hard to verify and ensure correctness when using floating-point computations. In our work, we discovered that the simplest strategy, sweep along the most varying principal axis, is the most effective on GPUs. The algorithm requires only a parallel sort for the sweep, then every GPU core will compare pairs of boxes. This strategy is not only massively parallel, but it is also trivial to ensure correctness: the only computation performed on the boxes is a comparison between floating-point numbers, which is exact.

*Narrow-Phase.* Snyder et al. [1993] introduced a conservative narrow-phase CCD algorithm that properly handles numerical errors by employing interval arithmetic. Tight-Inclusion (TI) CCD [Wang et al. 2021] used a similar idea and developed a faster CCD method

by replacing intervals with inclusion predicates. Unfortunately, TI cannot be directly translated to GPU as it contains several branches, dynamic allocation, and high register use<sup>1</sup>. We propose a novel algorithm based on the same idea as TI, redesigned to be GPU friendly.

*Contributions.* In our work, we introduce a dataset for CCD on linearized trajectories of five scenes obtained from different simulators containing between 50 thousand to half a million primitives (i.e., vertex-face and edge-edge). For every successive pair of frames, we compute the ground truth Boolean result and time of impact using a symbolic solver [Wolfram Research Inc. 2020]. We use this dataset to validate the output of several CCD algorithms.

Additionally, we introduce a novel GPU CCD implementation. Given two meshes for the start and end of a step, our method returns the time at which the impact occurs. Our pipeline includes the novel parallel broad-phase algorithm we call *Sweep and Tiniest Queue* as well as a GPU-friendly variant of the TI algorithm.

## 2 RELATED WORK

We present an overview of existing CCD datasets and the broad- and narrow-phase collision detection algorithms benchmarked in our study. We refer to [Serpa and Rodrigues 2020] for a detailed review of broad-phase algorithms and to [Wang et al. 2021] for narrow-phase algorithms.

### 2.1 Datasets

The UNC Dynamic Scene Benchmarks [Curtis et al. 2012] features keyframes from simulation data and is commonly used throughout collision detection works as a source of benchmark data. This dataset covers a variety of simulation methods, materials (e.g., deformable and rigid), and physics (e.g., cloth and fracturing solids). We borrow three scenes (cloth-funnel, cloth-ball, and n-body simulation) from this dataset. We enrich these scenes with ground truth Boolean results and symbolic time of impacts which was not included in the original dataset.

Serpa and Rodrigues [2020] benchmark several classic broad-phase collision detection algorithms. In doing so, they provide not only reference implementations of these algorithms but also a benchmarking framework and procedurally generated scenarios. These benchmark scenes focus on simple primitives (e.g., cubes and spheres) in free fall or undergoing random rigid motion. In contrast, we focus on the more general case of deformable triangle meshes. Serpa and Rodrigues [2020] focuses primarily on static collision detection, while our work evaluates broad-phase methods on continuous collision detection scenarios.

Wang et al. [2021] introduced a large scale dataset for narrow-phase CCD algorithms. The dataset is designed to cover common cases extracted from simulation scenarios and challenging degenerate cases. Wang et al. [2021] uses the dataset to evaluate the accuracy (the number of false positives), correctness (the number of false negatives), and efficiency (the average runtime) of different narrow-phase CCD algorithms. However, the large-scale dataset proposed by Wang et al. [2021] contains only the queries and the ground truth Boolean results, but not the collision time for each query. We

<sup>1</sup>On modern GPU architectures, the number of registers per core is extremely limited, and this puts a limit on how many local variables can be used in every function.

propose a new benchmark dataset of over 4M collisions combined with their time of impact (Section 4). This allows us to evaluate the accuracy of different methods in addition to their correctness.

## 2.2 Broad-Phase

We discuss the 12 methods benchmarked in [Serpa and Rodrigues 2020] and additionally include the spatial hash data structure used in [Li et al. 2020]. We note that broad phase methods can be used for both static or continuous collision detection by just changing the geometric proxies used: instead of building a proxy, such as a bounding box, around a static object, it is possible to build the proxy around the linearized trajectory in a time-step. Their performance is, however, very different in these two settings due to the much larger number of overlaps in the continuous collision detection case.

*BF.* A simple *CPU parallel brute-force* check where every box in the list is checked against every other box. To avoid any concurrent accesses to the resulting candidates, we use a synchronized vector [Dagum and Menon 1998], and the algorithm is accelerated using Advanced Vector Extensions (AVX) instructions. This algorithm is simple, but its complexity is  $O(n^2)$ , with  $n$  the number of primitives: it is not practical for large scenes, but it is viable for smaller ones.

*SAP.* A serial and parallel standard implementation of the *sweep and prune algorithm* [Baraff 1992; Capannini and Larsson 2016a,b, 2018]: it starts by performing an intersection check between the  $x$ -axes of the boxes by sorting the boxes along  $x$ . For every  $x$ -intersecting box, the algorithm proceeds by checking  $y$  and  $z$ , every time sorting and pruning the axis. This algorithm improves over the simple brute force as its complexity is  $O(n \log(n))$ . SAP comes also with an OpenCL GPU implementation from Bullet 3 [Coumans and Bai 2019] based on [Liu et al. 2010]. Note that we cannot provide a full comparison with the reference implementation because of the incomplete state of their code.

*iSAP.* A serial implementation of *incremental sweep and prune algorithm* [Coumans and Bai 2019]. It is an improvement over SAP: For every intersecting box along the  $x$ -axis, the pair of overlapping boxes is added to a list. Three lists are built to keep track of intersecting boxes along all three axes:  $x$ ,  $y$ , and  $z$ . Finally, a pass through all three arrays is done to find pairs of boxes that intersect along all three axes.

*Grid.* The scene is divided into voxels (or cells) of uniform size  $v$ . The voxel size  $v$  is chosen based on a target number of boxes per voxel (we use the default: 200 boxes per voxel). Every input bounding box is assigned to the cells intersecting it. We detect intersections between boxes by iterating over the sparse cells. This algorithm is efficient and easily parallelizable (on both CPU and GPU); however, its main disadvantage is the choice of  $v$ : a small  $v$  will lead to many boxes, and an explosion in memory due to duplicate collision candidates, and a large  $v$  leads to many boxes inside each cell (e.g., if there is only one voxel the grid reduces to brute force (BF)). The choice of  $v$  is particularly problematic for large displacements. For instance, if the objects are moving apart, the grid (and the number of boxes) will grow in size, potentially

leading to an exploding number. For the GPU algorithm, we use the OpenCL parallel implementation based on Bullet [Coumans and Bai 2019]. We note that it requires the same delicate choice of  $v$  that is even exacerbated as GPUs have less memory than CPUs.

*GSAP.* This method is similar to the Grid method, but uses a sweep and prune (SAP) inside each cell instead of a brute force check. It suffers from similar issues as Grid: A small  $v$  will result in many duplicate candidates and large memory usage, but, unlike Grid, a large  $v$  reduces to SAP and is, therefore, more efficient than Grid’s BF comparisons.

*SH.* A parallel CPU implementation of Grid, that encodes the grid implicitly using a *spatial hashing* function [Li et al. 2020; Tang et al. 2018a,b]. Each candidate box is rasterized in the grid, and for each voxel, a hash value is computed. These hash values are used to store the elements IDs in a hash map (mapping from voxel indices to a vector of element IDs contained inside the voxel). The candidate collisions can then be found by rasterizing the query element and looking up the voxel indices. Our implementation is based on the code from [Li et al. 2020], which has been modified to produce all collision candidates in one parallel loop and include axis-aligned bounding box checks of elements, to make it comparable with the other approaches. This algorithm has the same shortcomings as Grid: a wrong choice of  $v$  might lead to either slow performances or excessive memory usage. We use a heuristic for the voxel size equal to two times the maximum of the average edge lengths and the average displacement length. This ensures the average element fits within a single voxel.

*BVH.* A *bounding volume hierarchy*. The boxes are divided into two sets: query and target. The target queries are sorted using Morton encoding to optimize spatial locality. The sorted boxes are grouped into pairs, each forming a larger box. By recursively iterating the process, we obtain a *binary tree*, where the root is a box containing the whole space-time scene. Every query box traverses the tree by recursively checking its intersection with the box at the tree’s node until it reaches the leaf. The BVH can be updated “bottom-up” (i.e., if a leaf box grows, it can update its parent until the root), dramatically reducing the update cost in dynamic scenes. We use a *deferred BVH (DBVT-D)*, which performs a single tree-tree query. Additionally, Bridson et al. [2002] proposes to use numerical tolerances to account for rounding error in floating-point computation, which we also apply to all methods we compare against to mitigate (but as we will show in Section 5 not reliably address) the effect of rounding errors. We also include an OpenCL GPU implementation of the Linear BVH in Bullet [Coumans and Bai 2019]. Similar to the BVH, the tree is organized using Morton encoding. By default, Serpa and Rodrigues [2019] assumes a maximum of  $18n$  possible intersections, with  $n$  input bounding boxes, and discards any successive one. Changing the default size for all our scenes required a trade-off between performance for smaller scenes in exchange for more collisions in larger scenes. To avoid this unnecessary shortcoming, which introduces false negatives, we changed the algorithm to process all intersections in an appropriate amount of time: If the list of candidates reaches the maximum, we stop storing them and count their number. Once we know the total number of candidates,

we re-execute the algorithm with the correct preallocated size. This change affects only a few cases where the number of candidates exceeds  $18n$ .

*KDT.* An optimized *KD-Tree* designed to handle static scenes based on the efficient implementation in [Serpa and Rodrigues 2019]. The spatial subdivision is designed to adaptively partition the space and have a small number of boxes attached to each cell. We note that when using the automatic box inflation present in the implementation [Serpa and Rodrigues 2019], the algorithm does not report any collision (i.e., it fails to detect true positives). We thus alter the code to disable this feature and use our default 1% inflation, which leads to reasonable results suggesting a bug in the auto inflation code.

*Tracy.* The parallel method of Tracy et al. [2009], which builds off the incremental SAP of Baraff [1992] and Cohen et al. [1995] by including the ability to insert AABBs without the need to perform a full sort of the axes.

*CGAL.* The CGAL implementation of an interval-tree SAP algorithm designed to handle  $d$ -dimensional axis-aligned boxes (in our experiments, we use  $d = 3$ ) [Kettner et al. 2016; Zomorodian and Edelsbrunner 2000]. This method works by using SAP on the first axis and then using range and interval trees on the subsequent axes.

### 2.3 Narrow-Phase

*Numerical Root-Finding.* Narrow-phase CCD can be reduced to root-finding: the roots of a carefully designed function correspond to the times of impact. CCD of linear trajectories without minimal separation equates to finding the roots of a cubic polynomial [Provot 1997]. Many methods focus on solving these cubic polynomials using numerical methods [Provot 1997]. Provot [1997] introduce the most common strategy of finding a time of coplanarity and then performing an inside check. This idea has since been expanded to solve both rigid [Kim and Rossignac 2003; Redon et al. 2002] and deformable collisions [Bridson et al. 2002; Hutter and Fuhrmann 2007; Tang et al. 2011].

The downside of these methods is that they assume infinite precision. When implemented using floating-point numbers, these methods can both miss collisions (false negatives) and report non-existent collisions (false positives).

*Inclusion-Based Root-Finding.* Alternatively to numerical root-finding algorithms, some propose using inclusion-based root-finding algorithms to determine if a root exists in the co-domain of our function with some tolerance [Redon et al. 2002; Snyder 1992; Snyder et al. 1993; Von Herzen et al. 1990; Wang et al. 2021]. This can either be done using interval arithmetic [Snyder 1992] or by designing custom inclusion functions [Wang et al. 2021]. The latter has the benefit of producing tighter inclusion functions than general interval arithmetic and can be performed in floating-point with specially crafted error bounds. These methods avoid false negatives but produce false positives, which add extra numerical padding to simulated objects and can result in worse convergence when used in line-search-based implicit solvers.

*Conservative Advancement.* Originally introduced for CCD between rigid convex objects [Mirtich 1996], conservative advancement incrementally estimates the time of impact through a series of minimum distance queries. This work has been subsequently extended to non-convex [Zhang et al. 2006], articulated [Zhang et al. 2007], and polygon-soup models [Tang et al. 2009]. Most notably, in the context of this work, Tang et al. [2010] proposed a method of “Local Advancement” for CCD between deformable triangles. Li et al. [2021] propose a conservative advancement method for CCD between triangles. They claim numerical robustness as they attempt to conservatively underestimate the time of impact. As this work postdates the benchmark and analysis done by Wang et al. [2021], we investigate here their claims and determine if it is a viable alternative to the TI algorithm of Wang et al. [2021]. As shown in Section 5.2, the reference implementation provided by the authors misses collisions. We speculate that this is due to floating-point rounding errors in distance computations (involving square roots), which make the CCD algorithm in Li et al. [2021] not conservative and thus unsuitable for interior point optimization.

*Exact Methods.* Both Brochu et al. [2012] and Tang et al. [2014] introduce exact root-parity CCD methods. Root parity is insufficient for CCD, as it cannot distinguish between 0 or 2 collisions within a timestep. In addition, their algorithm is not handling certain corner cases, making the root parity check not exact [Wang et al. 2022, 2021].

The only method exact CCD method we are aware of is using symbolic root finding [Wolfram Research Inc. 2020]. Unfortunately, this is computationally expensive (seconds for each query) and thus impractical in a simulation setting. We use the ground truth generated by this method to verify the results of all queries used in this work.

## 3 PRELIMINARIES

We formally introduce the CCD problem and classifications of CCD algorithms, which we will use in our evaluation. A CCD algorithm considers a scene with objects (typically triangles) moving from an initial position at time  $t = 0$  to a final position at time  $t = 1$ . If two objects intersect during the movement, the algorithm should report COLLISION and return the time  $t^* \in [0, 1]$  at which such a collision occurs ( $t^* = \infty$  if there are no collisions). As mentioned, algorithm implementations may not be robust. If an algorithm reports COLLISION but no pair of objects actually intersect, we call the report a *false positive*. When an intersection actually occurs, but the algorithm does not report COLLISION, we call the report a *false negative*.

*Definition 3.1.* A CCD algorithm (or implementation of a CCD algorithm) is *exact* if it reports COLLISION if and only if the geometries intersect.

Hence, an exact algorithm never reports false positives and/or negatives. This property can be achieved using exact or symbolic calculations as explained in Section 4, though this is intractably slow in real applications. Nonetheless, most algorithms can tolerate few false positives; thus, we introduce a new definition.

*Definition 3.2.* A CCD algorithm (or an implementation of a CCD algorithm) is *conservative* if, when the geometries intersect, it reports COLLISION.

Therefore, a conservative algorithm never reports false negatives but may report false positives. This property is fundamental when false negatives cannot be tolerated, such as in a new trend of contact models and corresponding simulators [Ferguson et al. 2021; Li et al. 2020, 2021] which guarantee (and also assume), by construction, that no interpenetrations are present in the scene at any moment in time. These algorithms provide robust and accurate modeling of contact but are unable to tolerate CCD imprecision: if a CCD query misses a collision, a self-intersection will appear, breaking the assumption of a self-intersection-free state and thus prevents the simulation from terminating (Figure 1). This imprecision includes floating-point rounding errors, which need to be accounted for in a conservative CCD algorithm. For this class of algorithms, we measure their *accuracy* (i.e., we measure how close they are to be exact) by counting the number of false positives they produce.

Not all contact models require conservative CCD. In some cases, for instance, the model introduces contact forces to remove intersections between primitives. Since the forces are introduced after a collision to remove the intersections, small numerical errors in the CCD queries are unlikely to affect the overall simulation, and it is thus common to sacrifice numerical guarantees in the CCD algorithm, favoring approximations that lead to a lower computation cost.

*Definition 3.3.* A CCD algorithm (or an implementation of a CCD algorithm) is *approximate* if, most of the time, it reports COLLISION when the geometries intersect; however, some collisions may be missed.

In this case, it is important to measure both false positives (to see how close to exact they are) and false negatives (to ensure that only a few collisions are missed).

## 4 DATASET GENERATION

Our dataset is composed of five simulated scenes (Figure 2 top row) and the corresponding ground truth data for continuous collisions between frames. From the UNC Dynamic Scene Benchmark [Curtis et al. 2012], we include two co-dimensional cloth simulations with a large number of self-collisions (Cloth-Ball and Cloth-Funnel) and a simulation of a large number of rigid spherical bodies (N-Body). We also include two elastodynamic scenes featuring large compression and nonlinear buckling simulated using the method of Li et al. [2020] (Armadillo-Rollers and Rod-Twist).

*Ground Truth.* We generate ground truth for each successive pair of frames from each scene, using a combination of symbolic computation and conservative filtering. While this might seem similar to the dataset of narrow-phase ground truth introduced by Wang et al. [2021], they only provide a set of queries with no global mesh, nor are the queries separated by time step. Therefore, the dataset of Wang et al. [2021] is only suited for evaluating narrow-phase CCD algorithms, and this necessitates introducing a new dataset to evaluate broad-phase methods or CCD as a whole.

We first enumerate all possible collision pairs (collision candidates) through a brute-force approach. We only consider point-triangle and edge-edge pairs as these pairs capture the first collisions between triangles [Provot 1997] (not including points that are vertices of the triangles and edges that share a common endpoint). For each collision candidate, we determine if the pair collides using the provably conservative CCD of Wang et al. [2021]. While this CCD algorithm is guaranteed to not have false negatives, it may produce false positives. To eliminate false positives, we find the exact Boolean solutions of the CCD query using the symbolic solver in Mathematica [Wolfram Research Inc. 2020]. Mathematica uses symbolic computations combined with exact arithmetic to produce a symbolic expression for the time of impact. We use Mathematica’s exact predicate computation to determine if this is a valid time of impact ( $t \in [0, 1]$ ) and, therefore, a collision.

While we could skip the middle step and directly use the symbolic CCD, this would be prohibitively slow, as the symbolic solvers take several seconds per query. Instead, the method of Wang et al. [2021] quickly filters the majority of the collision pairs, leaving a smaller number of candidates to validate with the symbolic solver.

*Time of Impact Expressions.* In addition to the Boolean ground truth, our dataset is the first to include the symbolic expressions for the valid roots (as Wolfram Language MX files). It is necessary to save the symbolic expressions (instead of a real or rational number) as they may include operators and functions that cannot be evaluated exactly using floating point or rational numbers (e.g., square roots). For example, Equation (1) shows that Mathematica stores the time of impact as the roots of a cubic polynomial which can be solved analytically (e.g., by using Cardano’s method [Cardano 1545]) but requires irrational and complex arithmetic.

$$\begin{aligned}
 t = \text{Root}(&32438097225180438401964874473761976929223 x^3 \\
 &- 2546681458666122439357688750343089434416006 x^2 \\
 &+ 2471187477357729479707126378291971630585896 x \\
 &- 266132451806881357156163391768279366354706, 2)
 \end{aligned} \tag{1}$$

In total, this includes over 3.3M edge-edge and 728K vertex-face contacts. For reproducibility, we provide scripts to rerun the validation on other architectures and compilers.

## 5 COMPARISON

We run all our experiments on an AMD Ryzen™ Threadripper™ PRO 3995WX 64-Cores @ 2.7 GHz processor with 64 threads, 512 GB of RAM, and an NVIDIA® 3080 Ti. Note that, to avoid duplicated plots, we include the results of our method; we refer to Section 6 for a detailed explanation of what it does and to Section 7 for a discussion. The BVH method provides a function to update the data structure with new positions instead of rebuilding it from scratch at every frame, which we use in our experiments. For all other methods, the acceleration data structure is rebuilt at each frame.

### 5.1 Broad-Phase

Figure 3 presents detailed statistics of twelve implementations of a broad-phase collision detection algorithm run on our benchmark

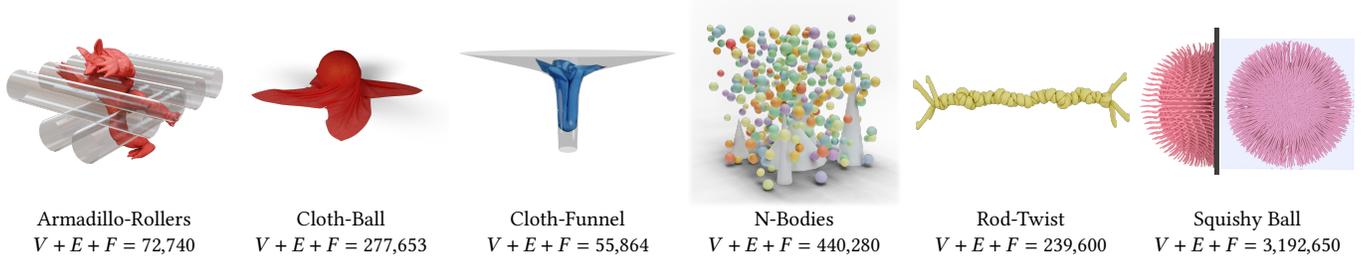


Fig. 2. Overview of the six scenes in our dataset.

scenes. For each broad-phase method, we compare the list of candidates with the ground truth data. We report the number of false positives (collisions detected by the method but are not colliding) and false negatives (collisions not detected by the method but are colliding), as well as the running time and maximal memory usage. The number of false positives is quite stable among the methods, with a lower number for the Grid-based and the BVH-based approaches. All methods use a similar amount of memory, except for the BVH (for small scenes) and Tracy (for medium scenes). We note that for SH, memory (and runtime) vary heavily based on the choice of the grid size (Appendix A). The runtime varies significantly across the different methods, but the parallel SAP, SH, and GPU BVH are faster.

Since the goal of a broad-phase CCD is to filter unnecessary collision pairs, it is expected to introduce false positives, but it should be conservative; that is, it should *never* have false negatives. In our experiments, despite inflating the bounding boxes by 1% to ameliorate numerical problems, we found that the public implementations we use of GSAP, GpuGrid, GpuSAP, KDT, SAP, and CGAL have false negatives<sup>2</sup>. The false negatives produced by the GpuGrid and GpuSAP methods are at least in part due to the implementation choice of allocating a fixed-size array for candidates and ignoring any additional candidates. We believe that it is the case, as these methods are designed for rigid bodies where the number of boxes is proportional to the number of objects in the scene (typically not above tens of thousands), while for deformable bodies, the number of boxes depends on the complexity of the surface (i.e., one box per triangle, edge, and vertex) leading to a much larger number of boxes (the smallest scene in our dataset has 50 thousand boxes). This problem could be fixed by a more complex and less efficient implementation.

## 5.2 Narrow-Phase

For each narrow-phase algorithm, we only record the time of impact for the whole timestep. For the narrow-phase, we compare only with the original CPU TI [Wang et al. 2021] and the Additive CCD (ACCD) method of Li et al. [2021] as it has been shown that other existing algorithms are either not conservative or produce a large number of false positives [Wang et al. 2021, Table 1]. Table 1 summarizes the results and shows that ACCD is not conservative: as with many

<sup>2</sup>Note that CGAL `box_self_intersection_d` is fast but has a bug. The `box_self_intersection_all_pairs_d` method is correct, but its quadratic complexity makes it intractable.

Table 1. Average narrow-phase runtime (milliseconds), total number of false positives (in thousands), and total number of false negatives for ACCD, TI, and ours on GPU.

Scene	Method	Runtime (ms)	FP (1K)	FN
Armadillo-Rollers	ACCD	2	8594	0
	TI	32	8594	0
	Ours	7	8594	0
Cloth-Ball	ACCD	7	18637	0
	TI	103	18637	0
	Ours	15	18637	0
Cloth-Funnel	ACCD	1	2995	2470
	TI	12	3417	0
	Ours	5	3417	0
N-Bodies	ACCD	150	232991	258080
	TI	402	232965	0
	Ours	46	232965	0
Rod-Twist	ACCD	4	457479	0
	TI	195	457479	0
	Ours	24	457479	0

other CCD methods, ACCD focuses on performance at the cost of correctness.

## 6 ALGORITHM

Our CCD algorithm for triangle meshes, summarized in Algorithm 1, takes as input two triangle meshes  $M_0, M_1$  whose vertex positions are given at time  $t = 0$  and  $t = 1$  and are assumed to move along linear trajectories in between. The algorithm returns the earliest time of impact  $t^*$  ( $t^* = \infty$  if there are no collisions between  $M_0$  and  $M_1$ ). Vertex coordinates are represented using floating-point numbers. Our parallel algorithm is implemented both on GPU and CPU and strives to vectorize the computations.

Our broad and narrow phase algorithms are inspired by the results (Section 5): for broad phase, the SAP algorithms are among the fastest and simplest (we exclude SH as its runtime depends on the grid size, Figure 14), and only TI is conservative for the narrow phase. Thus we develop our algorithm on their core principle while redesigning it to target vectorized architectures and flexible memory management.

*Overview.* We first build a set  $B = \{b_i \mid i = 1, \dots, k\}$  of  $k$  boxes  $b_i = (b_i^m, b_i^M)$  (where  $b_i^m$  and  $b_i^M$  are the minimum and maximum corner of the box respectively) on CPU around every moving primitive (triangles, edges, and vertices at both  $t = 0$  and  $t = 1$ ) on  $M_0$

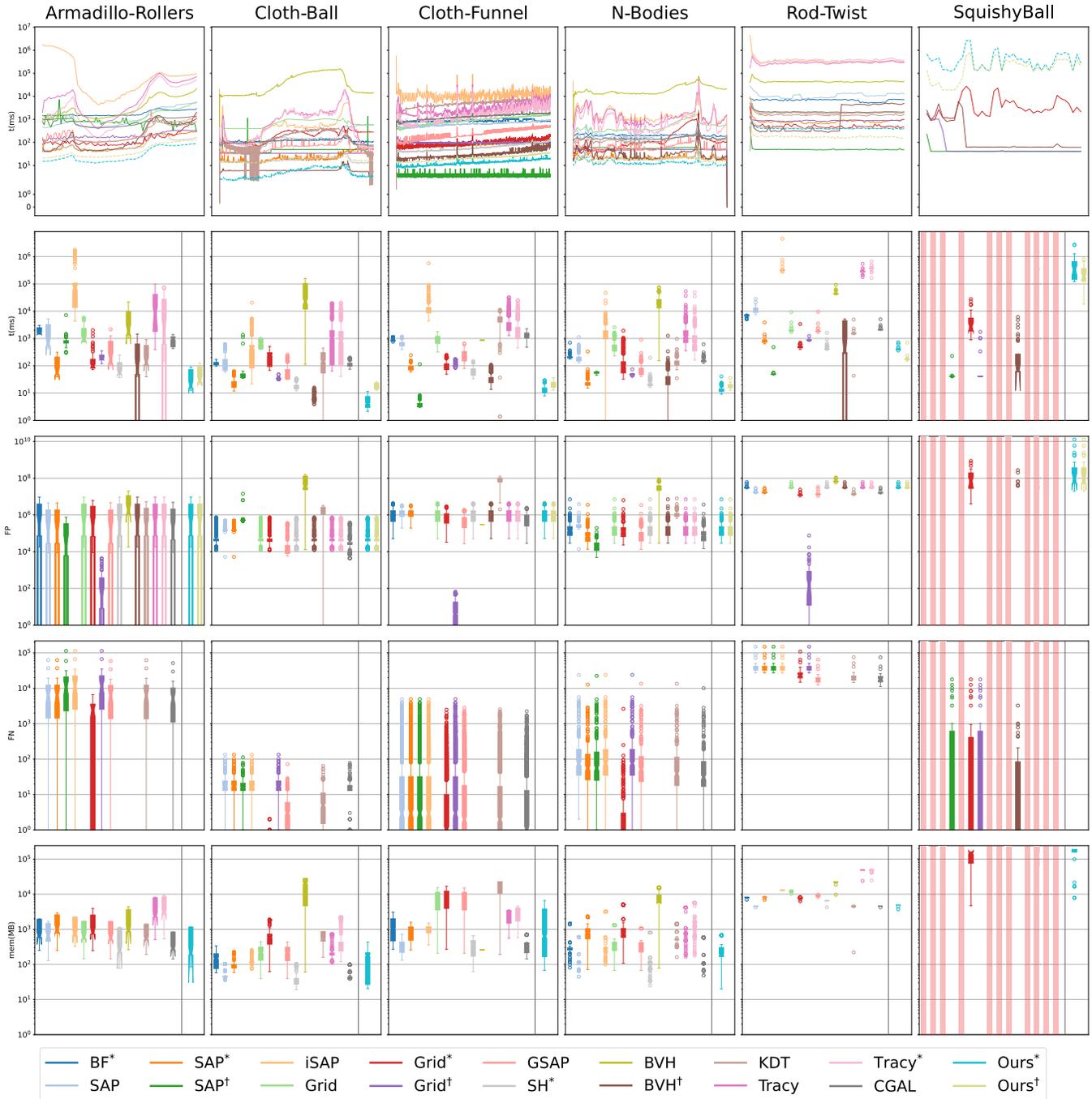


Fig. 3. Results of the 13 different methods for our six different scenes (columns). For each scene (Figure 2), we report the performance for every simulation frame (first row), timings box plot (second row), false positive box plot (third row), false negative box plot (fourth row), and memory box plot (fifth row). The star\* depicts parallel CPU methods, while we use the dagger† for GPU methods. For many methods, the SquishyBall experiment ran out of memory; we discarded them and marked them in red. The box plot shows aggregated statistics: the box extends from the first to the last quartile, the line in the middle is the median, and the lines (whisker) extend to the largest/smallest non-outlier point. Outliers are plotted as circles and defined as data points laying outside the 1.5 times interquartile range.

**Algorithm 1** Overview of our CCD algorithm.

---

```

1: function CCD( $M_0, M_1$ )
2:    $B \leftarrow$  BUILDBOXES( $M_0, M_1$ )            $\triangleright$  CPU, Section 6.1
3:    $C \leftarrow$  BROADPHASE( $B$ )                  $\triangleright$  GPU, Section 6.2
4:    $t^* \leftarrow$  NARROWPHASE( $C, M_0, M_1$ )      $\triangleright$  GPU, Section 6.3
5:   return  $t^*$ 
6: end function

```

---

and  $M_1$ . We use the CPU for this step as its running time is negligible. Additionally, we need to have access to the whole scene that might not fit on GPU. When moving the boxes to GPU, we represent them in *single* precision while ensuring that every input (in double precision) is *exactly* contained in its box  $b_i$  (Section 6.1). This phase is fast and takes around 10% of the runtime (Figure 9).

Then we pass  $B$  to our broad phase algorithm to discard any far away candidate (Section 6.2). The broad phase produces a set of  $n$  candidates intersection pairs  $C = \{(l_i, r_i) \mid 1 \leq i \leq n\}$ , where every pair  $(l_i, r_i)$  indicates that the primitives in the boxes  $b_{l_i}$  and  $b_{r_i}$  potentially intersect. This stage takes about 50% of the runtime (Figure 9). The main idea of our algorithm is similar to Sweep and Prune [Baraff 1992; Cohen et al. 1995], but we vectorize the operation and avoid assumptions on the number of intersections to avoid memory allocation. In particular, SAP [Liu et al. 2010] uses the length of the box as a heuristic to determine the number of threads used to process a box; instead, we use a *consumer queue* which keeps the work balanced. While there are similarities between SAP and our algorithm, our version is conservative and is 3.76 $\times$  faster on average (Figure 3).

Finally, to obtain the time of impact  $t^*$ , we run our narrow phase algorithm using the collision candidates  $C$ , and the input meshes  $M_0$  and  $M_1$  (Section 6.3). The core idea is the same as in [Wang et al. 2021], but we redesigned the algorithm to avoid recursion and used a *worker queue* paradigm to make it GPU parallelizable. In comparison, we implemented [Wang et al. 2021] in CUDA and observed that it does not scale at all with multiple threads, due to branching, and high-registry use, making it slower than its parallel CPU counterpart implemented by Wang et al. [2021]. Note that, on both CPU and GPU, we execute the narrow-phase using the same precision as the input (e.g., double precision for all our experiments).<sup>3</sup>

In Section 6.5 we show that our overall algorithm is *conservative*; that is, it never misses collisions. Section 5 shows that our algorithm has similar performance as state-of-the-art and Section 7 details the performance of our method.

### 6.1 Construction of the Boxes

To construct a *tight* single precision box  $b = (b^m, b^M)$  around a primitive (i.e., a triangle, edge, or vertex), we first compute the extent of the box  $b^m, b^M$  in double precision using the min and max of the coordinates of the primitive during its entire movement. Because trajectories are linear, considering  $t = 0$  and  $t = 1$  is sufficient. For instance, for a triangle,  $b^m$  is the minimum of the  $x, y, z$ -coordinates of the three vertices, each at either  $t = 0$  or

<sup>3</sup>Modern consumer GPUs have very limited support for double computation, but this is not an issue for our purposes, as the narrow-phase is memory bound and the lower number of double-precision ALUs does not affect the algorithm performance.

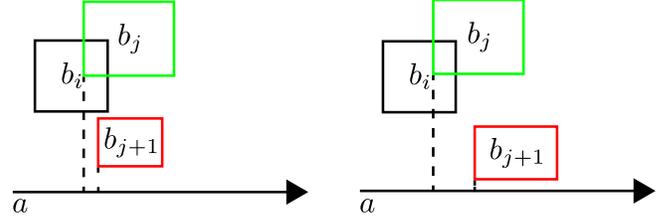


Fig. 4. The boxes  $b_i, b_j$ , and  $b_{j+1}$  are sorted along the axis  $a$ . Since  $b_i$  and  $b_j$  intersect, we append the pair to  $C$  and check  $b_{j+1}$ . If  $b_{j+1}$  intersects  $b_i$  on the axis  $a$  (left) we append the  $(b_i, b_{j+1})$  to  $Q'$ , if not (right) we discard the pair. Because the boxes are sorted along the axis  $a$ , if  $b_{j+1}$  does not overlap with  $b_i$ , then  $b_k$  for all  $k > j + 1$  do not intersect  $b_i$ , and we can skip checking them.

$t = 1$ . To *conservatively* convert  $b^m$  in single precision, for each coordinate (e.g.,  $x$ ), we first round  $b_x^m$  to its nearest single precision value and check if  $b_x^m < b_x^m$ , in case it is not, we decrease  $b_x^m$  to its previous representable single precision value using the function `nextafterf`<sup>4</sup>. The procedure for  $b^M$  is similar. When running our algorithm on GPU, we need to copy the boxes on the device; in our experiments, this time, B2G, is negligible (Figure 9).

### 6.2 Broad-Phase

Our Sweep and Tiniest Queue (STQ) algorithm is based on the Sweep and Prune algorithm [Baraff 1992; Cohen et al. 1995]. We observe that on modern architectures, due to their memory layout and a large number of ALUs that favor heavier computation with structured memory access, brute-force checking all possible pairs is not only easy to parallelize but extremely fast. Unfortunately, this simple approach has a runtime quadratic with respect to the number of pairs and cannot be applied to large scenes. To overcome this limitation, we borrow ideas from the SAP to limit the average complexity of our algorithm (Figure 5).

*Algorithm.* Our STQ starts by computing the variance  $\sigma$  of the boxes' centers  $B_C$  and finding the most varying axis  $a$  (Line 4) [Liu et al. 2010]. We then sort the boxes  $B$  along the axis  $a$  based on their minimum  $a$  coordinate and initialize a queue  $Q$  that will hold pairs of boxes that overlap on the  $a$ -axis. Since  $a$  is the axis of maximum variance, this will lead to the smallest possible queue among the three axes if the data is uniformly distributed.

In the first step, for every box  $b_i$  we check if it intersects its next box  $b_{i+1}$  along the  $a$ -axis; if it does, we append the pair  $(b_i, b_{i+1})$  to  $Q$  (Line 8).

Then STQ extracts a pair  $(b_i, b_j)$  from  $Q$  and checks if it intersects in the remaining two axes  $a^c$  (Line 17). If they do, we append the pair to the output  $C$  (global). Finally we add the pair  $(b_i, b_{j+1})$  to an output queue  $Q'$  if  $(b_i, b_{j+1})$  intersects along the  $a$ -axis (Line 21, Figure 4).

*Implementation Remarks.* We remark that the STQ algorithm creates queues with at most  $k - 1$  elements, as each pass on  $Q$  pushes at most one pair. Additionally, as not all pairs are always added, the sizes are monotonically decreasing. On GPU, we exploit this

<sup>4</sup><https://en.cppreference.com/w/c/numeric/math/nextafter>

**Algorithm 2** Overview of the broad-phase.

---

```

1: function BROADPHASE( $B$ )
2:    $B_C \leftarrow \{c = (b^m + b^M)/2 \mid b \in B\}$             $\triangleright$  Box centers
3:    $\sigma \leftarrow \sigma(B_C)$                               $\triangleright$  Variance of the boxes centers
4:    $a \leftarrow \operatorname{argmax}_{i \in \{x,y,z\}} \sigma^i$ 
5:    $a^c \leftarrow \{x,y,z\} \setminus a$ 
6:    $B' \leftarrow \operatorname{SORT}(B, \operatorname{ORDER}_a)$                   $\triangleright$  in parallel
7:    $Q \leftarrow \{\}$ 
8:   for all  $i \in \{1, \dots, |B'| - 1\}$  do
9:     if  $b_{B'_i}^a \cap b_{B'_{i+1}}^a \neq \emptyset$  then            $\triangleright$  Boxes intersect along  $a$ 
10:       $Q \leftarrow Q \cup (b_{B'_i}, b_{B'_{i+1}})$ 
11:    end if
12:  end for
13:
14:  while  $Q \neq \emptyset$  do
15:     $Q' \leftarrow \{\}$ 
16:    for all  $(b_i, b_j) \in Q$  do
17:      if  $b_i^{a^c} \cap b_j^{a^c} \neq \emptyset$  then
18:         $C \leftarrow C \cup (b_i, b_j)$ 
19:      end if
20:      if  $b_i^a \cap b_{j+1}^a \neq \emptyset$  then
21:         $Q' \leftarrow Q' \cup (b_i, b_{j+1})$ 
22:      end if
23:    end for
24:     $Q \leftarrow Q'$ 
25:  end while
26:  return  $C$ 
27: end function
28:
29: function ORDER $_a(b_i, b_j)$ 
30:  return  $b_i^{m_a} < b_j^{m_a}$             $\triangleright$  Order by min value along  $a$ 
31: end function

```

---

observation to pre-allocate the correct queue size and guarantee that we will always have the necessary space. To efficiently parallelize STQ on GPU, we exploit shared memory per thread block. In our experiments, the most efficient strategy consists of splitting the sorted boxes  $B'$  into  $m$  blocks containing 32 queries each (i.e.,  $m = k/32$ ). Since a warp consists of 32 threads, choosing a smaller number will introduce unnecessary branching. Using larger thread blocks can lead to more imbalanced workloads and longer runtime of branching in the code. The GPU thread-block scheduler performs well even with a larger grid size of thread blocks.

When running our algorithm on GPU, we need to prepare the data for the narrow-phase; this requires:

- (1) splitting the set  $C$  into edge-edge and vertex-face cases (SO),
- (2) transforming the pairs in  $C$  into narrow-phase data (CD),
- (3) coping the vertex coordinates to the device (V2G).

Similar to copying the boxes to GPU, these intermediate stages are negligible (Figure 9).

**6.3** Narrow-Phase

Since our algorithm builds upon [Wang et al. 2021], we first provide a self-contained overview of the original algorithm.

*Summary of [Wang et al. 2021].* The algorithm uses inclusion functions to detect collisions. It starts by constructing the interval  $I = I_t \times I_{uv} \subseteq [0, 1]^3$ , where  $I_t = [0, 1]$  is the time interval and  $I_{uv}$  is the parameterization of the space. For a point-face query  $I_{uv} = \{(u, v) \mid 0 \leq u, v \leq 1 \wedge u + v \leq 1\}$ , where  $u$  and  $v$  are the triangle’s barycentric coordinates. For an edge-edge query  $I_{uv} = [0, 1]^2$ , where  $u$  and  $v$  are the first and second segment parameter. Using  $I$ , the algorithm defines the box  $B = B_F(i)$  [Wang et al. 2021, Equation (4)] that, if it intersects with a tolerance box  $C_\epsilon$  [Wang et al. 2021, Equation (5)]<sup>5</sup>, determines if the interval  $I$  contains a root or not. In case it does, the algorithm recursively splits  $I$  into two subintervals [Wang et al. 2021, Algorithm 2] until they are either too small or completely contained in  $C_\epsilon$ .

*Algorithm.* The method of Wang et al. [2021] can be trivially parallelized on a CPU by adding a parallel loop around the candidates  $C$ . This strategy works well, but it is unfortunately not suitable with GPU architectures, which for good performance requires all threads to perform exactly the same operations and have similar memory access patterns, and this is not the case for Wang et al. [2021], as each query requires a different number and type of subdivisions.

To overcome this limitation, we observe that the core of the algorithm processes intervals and not queries (Algorithm 3). We can thus parallelize over interval splits instead of queries. This observation leads to performing the same operations on every thread independently from the candidate<sup>6</sup>.

We start by constructing, for every collision candidate  $C$ , the initial interval  $I = [0, 1]^3$  (Line 2) and append them to the input queue  $Q$ . We then process in parallel all intervals in  $Q$  and produce the output intervals’ queue  $Q'$  (Line 6). In the end, we swap the roles of the two queues (Line 15) and continue alternating until  $Q$  is empty.

*Time of Impact.* We modified how we process a single interval (Line 20) with respect to [Wang et al. 2021] to account for the time of impact. We first check if  $I_t^l$  (i.e., left-hand-side of the time interval of  $I$ ) is larger than the current time of impact  $t^*$ . In this case,  $I$  can safely be skipped (Line 22). Then we proceed as in [Wang et al. 2021] and, if the box  $B$  constructed from  $I$  does not intersect  $C_\epsilon$ ,  $I$  can be discarded as it does not contain a root (Line 26). Finally, if the width  $w(I)$  of  $I$  is smaller than a user-provided tolerance  $\delta$  (or if  $B$  is contained in  $C_\epsilon$ ), we report a collision by returning  $I_t^l$  (Line 29). If it is not the case, we split  $I$  into a left  $I^l$  and right  $I^r$  interval and return the current time-of-impact as optimal.

<sup>5</sup> $C_\epsilon$  relies on floating-point operations to be compliant with the IEEE 754 standard, which is the case for GPU (see <https://docs.nvidia.com/cuda/floating-point/index.html>). Our algorithm is designed to account for the rounding error produced by these operations. Our floating-point filters are conservative in the sense that no rounding can make them fail. Possible fused multiply-add (FMA) contractions would only make the results of single operations more precise, and therefore our filters still work.

<sup>6</sup>An additional subtle benefit of this algorithm is that it makes the GPU kernel shorter, reducing the number of registers used, which is a common performance bottleneck on GPUs, where each streaming multiprocessor (SM) has a very small pool of registers available.

**Algorithm 3** Overview of the narrow-phase.

---

```

1: function NARROWPHASE( $C, M_0, M_1$ )
2:    $Q \leftarrow \text{BUILDINTERVALS}(C, M_0, M_1)$ 
3:    $t^* \leftarrow \infty$ 
4:   while  $Q \neq \emptyset$  do
5:      $Q' \leftarrow \{\}$ 
6:     for all  $I \in Q$  do                                 $\triangleright$  In parallel
7:        $t^*, I^l, I^r \leftarrow \text{PROCESSINTERVAL}(I, t^*)$ 
8:       if  $I^l \neq \emptyset$  then
9:          $Q' \leftarrow Q' \cup \{I^l\}$ 
10:      end if
11:      if  $I^r \neq \emptyset$  then
12:         $Q' \leftarrow Q' \cup \{I^r\}$ 
13:      end if
14:    end for
15:     $Q \leftarrow Q'$ 
16:  end while
17:  return  $t^*$ 
18: end function
19:
20: function PROCESSINTERVAL( $I, t^*$ )
21:    $t \leftarrow I_t^l$ 
22:   if  $t \geq t^*$  then                                 $\triangleright$  Current interval is after  $t^*$ 
23:     return  $t^*, \emptyset, \emptyset$ 
24:   end if
25:    $B \leftarrow B_F(I)$ 
26:   if  $B \cap C_\epsilon = \emptyset$  then                         $\triangleright$   $I$  does not have collision
27:     return  $t^*, \emptyset, \emptyset$ 
28:   end if
29:   if  $w(B) < \delta$  or  $B \subseteq C_\epsilon$  then                 $\triangleright$  Collision found
30:     return  $t, \emptyset, \emptyset$ 
31:   end if
32:    $I^l, I^r \leftarrow \text{SPLIT}(I)$                         $\triangleright$   $I$  gets refined
33:   return  $t^*, I^l, I^r$ 
34: end function

```

---

*Discussion.* The algorithm has only two necessary synchronizations: 1) the update of  $t^*$  and 2) appending intervals to  $Q'$ . We update  $t^*$  using a mutex as `atomicMin` does not support floating-point numbers. To efficiently append intervals, we keep track of the size of  $Q'$  and use `atomicAdd` to increase the size counter when appending new elements.

## 6.4 Batching

Running CCD on a GPU is particularly challenging as the amount of required memory might easily exceed the device’s physical memory. For instance, the queries necessary to run the scene in Figure 10 cannot fit in 12 GB. While splitting the work in *batches* is possible for our method, it is challenging for other methods which use more complex spatial data structures. While other methods could have implemented batching, we are not aware of any existing CCD implementation that has this feature.

As our method checks every possible pair, we can schedule their execution in batches whose size depends on the available memory.

To ensure that we have enough memory, we measure the size in bytes of the different data structures. Namely, let

- $S_P$  the size of the input parameters for the narrow phase (56 bytes);
- $S_Q$  the size of the narrow phase queries (24 doubles per query);
- $S_I$  the size of the narrow phase interval (252 bytes);
- $S_i$  the size of two integers used to store a colliding pair (8 bytes).

We assume we can fit all boxes  $b_i$  and the scene in CPU memory. Our batching strategy requires storing all boxes (but not collision pairs) on the GPU: this is not an issue in our experiments as we can fit 97,612,893 boxes on 4 GB of memory, an unlikely scenario even for large scenes. In case the scene is larger, our code falls back to a CPU implementation. Once we construct the boxes, we estimate the available memory  $\mathcal{M}$  using a cuda function. To ensure that we can run our algorithm safely, we construct the output queue  $C$  containing the colliding pairs of maximum size  $S_C = (\mathcal{M} - S_P) / (S_Q + 3S_i)$ . To compute the maximum size  $S_C$ , we subtract the necessary memory for storing parameters  $S_P$ ; then we divide by the memory necessary to run every query in the narrow phase. Every query requires  $S_i$  bytes to store the results of the broad phase (vertex-face and edge-edge mixed),  $2S_i$  bytes to conservatively separate the output of the broad phase into the two primitive pair, and  $S_Q$  bytes to store the input query for the narrow phase. This ensures that if the broad phase appends at most  $S_C$  results, we can *always* run the narrow phase (even if the queue  $Q$  might overflow).

While running our broad phase, we append the colliding pair to the  $C$  only if it does not overflow (in this case, we can run the whole algorithm without batching). If it does, we stop the broad phase and divide the input boxes into two batches and restart the algorithm until all batches can append all pairs to  $C$ .

In the narrow phase, the only problem that might occur is that the working queue  $Q$  might overflow as we split the input intervals. We allocate  $Q$  of size  $\mathcal{M}/S_I$ , where  $\mathcal{M}$  is the available memory after running the broad phase. If  $Q$  overflows, we discard the results and re-run the narrow phase using half of the pairs  $C$ . We repeat this procedure until  $Q$  does not overflow. We note that we always keep the most accurate time of impact when we batch

## 6.5 Guarantees

Our broad-phase and narrow-phase algorithms are both *conservative*. Section 7.3 empirically confirms that the final time of impact is also *conservative* (i.e., less or equal to the exact value) while providing a measure of its difference from the ground truth.

*Broad-phase.* In our algorithm (Algorithm 2), any intersection check amounts to only comparing floating point numbers. No other operation and/or rounding is involved in the process, meaning that intersection checks are all exact and  $C$  contains all and only the intersecting pairs. However, the boxes themselves might be slightly larger than necessary due to the rounding from double to single precision. Thus, our broad phase detection is *conservative* because  $C$  might include intersecting pairs that would not intersect without rounding. However, since the distortion is as small as machine

precision, these spurious pairs are too few (if any) to have any perceivable impact on the overall performances.

*Narrow-phase.* As explained in Algorithm 3, if the two meshes collide, our process returns a finite time of impact (Line 17). This occurs if at least one root was found during the interval processing (Line 30). In turn, this occurs if the condition on Line 29 is verified. This condition is exactly the same as used in Wang et al.’s [2021] algorithm. Our algorithm differs from Wang et al. [2021] because some additional intervals are discarded, whereas some others are added. However, we discard an additional interval only if the current time of impact is finite (Line 22), meaning that a collision was already detected. Stated differently, our additional discards cannot determine a false negative. Similarly, Wang et al.’s [2021] method does not process an interval that we process only if the algorithm exits before having processed all the pairs and their intervals, which happens only if a collision is detected. That is, our additional intervals cannot determine a collision unless Wang et al.’s [2021] method does the same. In essence, our algorithm returns a finite time of impact if and only if Wang et al.’s [2021] algorithm reports a collision and, since Wang et al.’s [2021] algorithm is conservative, we can conclude that ours is equivalently conservative.

## 7 RESULTS

### 7.1 Comparison

Overall, our GPU algorithm is up to 20 times faster for larger scenes than the best existing combination (BVH for broad-phase and CPU parallel TI); 3 times faster for Armadillo-Rollers, 10 for Cloth-Ball, 1.6 for Cloth-Funnel, 22 for N-Bodies, and 17 Rod-Twist.

*Broad Phase.* As for the other BP methods (Figure 8), our method has similar accuracy to any other algorithm. The performance of our broad-phase methods is mostly independent of the scene: ours is consistently among the fastest methods. Our method on GPU is faster than BVH on large scenes (N-Bodies and Rod-Twist) and has a comparable time for smaller ones. On CPU, our method has a similar performance to the spatial hash; however, it does not require tweaking the cell size. Our method uses slightly more memory than BF, as it does not need to store any data structure.

*Narrow Phase.* ACCD is 3.5× faster than our method, but it fails to detect collisions, making it not suitable for contact methods using interior point optimization (e.g., IPC). TI is consistently slower than our method (between 2.4 and 8.7 times faster).

### 7.2 Scaling

While our algorithm is inspired by a brute force approach which has quadratic complexity with respect to the number of boxes, we borrow ideas from the SAP to limit the average complexity of our algorithm. Figure 5 shows that the run time of both of our algorithms grows linearly with the number of boxes.

To assess the parallel scalability of our broad-phase method, we run the last ten frames of Rod-Twist on CPU, varying the number of threads from 1 to 32 (Figure 6). Our algorithm scales well with respect to the number of threads: with 8 threads, the broad-phase is 7.5 times faster, and it gets 22 times faster with 32 threads. Constructing the boxes scales the worst: 3.7 times faster with 8 threads

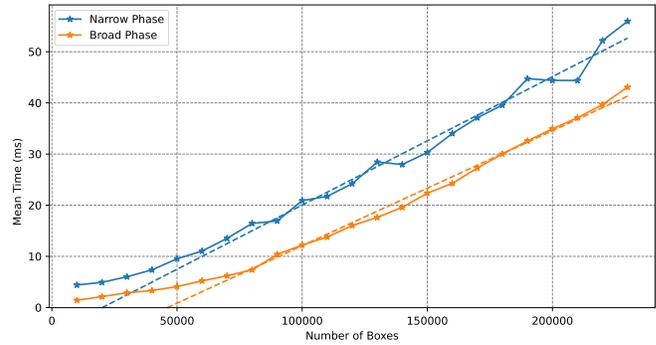


Fig. 5. Run time of our algorithm with respect to the number of queries. To generate a varying number of queries, we select a random sub-sample of all possible boxes from the last 10 frames of Rod-Twist.

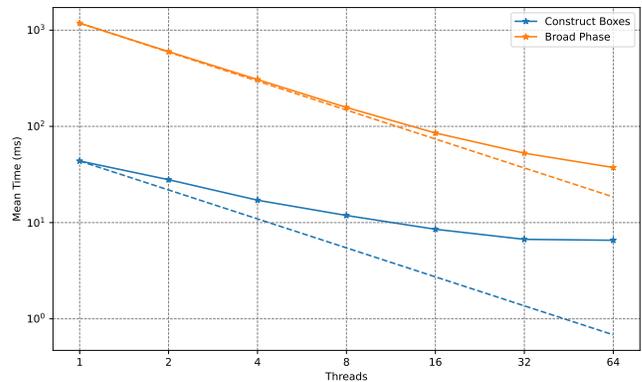


Fig. 6. Strong scaling of our method for the last 10 frames of Rod-Twist. The dashed lines show the perfect scaling.

and 6.5 faster for 32 threads. However, it is around one order of magnitude faster than the broad-phase, making it negligible.

### 7.3 Time of Impact Validation and Accuracy

Taking a step size greater than the exact time of impact (TOI) will lead to intersections, so we confirm our predicted TOI is less than or equal to the symbolic expression (Section 4). This confirms our method is conservative. Additionally (and not used to verify correctness), we estimate the error of our time of impact by evaluating the difference using 128 bits of precision (Figure 7).

The mean error for all queries is 0.0023 with a standard deviation of 0.018 and a median error of  $1.03 \times 10^{-6}$ . We note that this distribution varies between scenes (e.g., cloth-ball has a mean error of  $8.33 \times 10^{-6}$  compared to 0.032 for rod-twist). This indicates a dependency between the types of contact and the accuracy of the time of impact.

### 7.4 Different Architectures

We also run our method on different “workstation” hardware (Figure 8); two CPUs: CPU1 a consumer architecture (Intel® Core™ i7-5930K CPU @ 3.5 GHz) and CPU2 a professional CPU (AMD

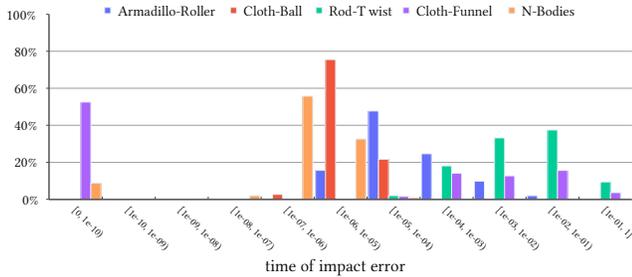


Fig. 7. Using the symbolic ground truth for the time of impact, we compute an error per query per scene and plot them as a histogram in log scale. The error is computed as the absolute difference between the earliest root reported by the symbolic root finder and our code (the error is approximately computed using 128 bits of precision). All TOIs computed by our method are verified symbolically to be less than the ground truth.

Ryzen™ Threadripper™ PRO 3995WX 64-Cores @ 2.7 GHz) and two GPUs: GPU1 a consumer-grade card (NVIDIA® 3080 Ti) and GPU2 a professional-grade card (NVIDIA® v100). Using this naming convention, the results in Section 5 are run on CPU2 and GPU1. For every CPU run, we limit the number of threads to 12. We note that for many computations, a CPU implementation on CPU2 has an execution time comparable to a GPU implementation:<sup>7</sup> the hardware is, however, expensive as CPU2 costs around ten times more than a top-of-the-line GPU.

The narrow phase on GPU is faster than the broad phase (just barely for the Rod-Twist and 9 times for the N-Bodies); on CPU, the two phases are more comparable (narrow-phase is 3 times slower on Rod-Twist and 3 times faster on Cloth-Funnel). The difference comes from the fact that the narrow-phase on GPU is up to 80 times faster than CPU, while the broad-phase peaks at 10 times faster. As expected for our method CPU2 (orange) has a similar performance to GPU1 (green).

Figure 9 shows the cutoff of the different phases of our algorithm; the broad phase dominates the computation, in particular on the slower CPU. For scenes with complex contacts (Rod-Twist and N-Bodies) the narrow phase becomes more prominent. When switching to a faster architecture (CPU1 to CPU2 and GPU1 to GPU2), both phases obtain a similar speedup. The narrow phase benefits more when switching from CPU to GPU. The other parts of our algorithm (e.g., memory copy, allocation, etc.) are negligible.

## 7.5 Batching

To evaluate the overhead of our batching strategy (Section 6.4), we artificially limited the available memory on the GPU between 1GB and 12GB (Figure 10). As we decrease the memory, our algorithm becomes slightly slower. For the narrow phase at 6GB, one of the early batches finds a small time of impact, leading to all other chunks to quickly terminate.

Table 2. Performance of our new CCD and broad phase for the unit tests of Erleben [2018] in [Li et al. 2020, Figure 11], the five cube stack, mat-twist, and mat-knives simulations. For tiny scenes the overhead of our method worsens performance, but in general leads to a performance increase in both the CCD and simulation as a whole. For larger scenes, the bottleneck shifts to the linear solves and Hessian matrix assembly leading to a smaller overall improvement of running time.

Scene	#V	#T	CCD Time (s)		Total Time (s)		CCD Speed-Up	Total Speed-Up
			SH+TI	Ours	SH+TI	Ours		
Spikes	5	2	0.01	0.73	0.12	0.90	0.01×	0.13×
Spike and Wedge	5	2	0.01	0.79	0.13	0.89	0.02×	0.14×
Spike in a Hole	5	2	2.47	4.65	2.93	5.29	0.53×	0.55×
Wedge in a Crack	6	3	5.11	6.17	5.75	7.17	0.83×	0.80×
Sliding Spike	5	2	0.82	0.62	0.86	0.73	1.31×	1.18×
Spike in a Crack	5	2	5.45	3.58	5.80	4.13	1.52×	1.40×
Cliff Edges	8	6	3.98	1.66	4.46	2.24	2.40×	1.99×
Internal Edges	8	6	6.31	2.57	6.93	3.38	2.45×	2.05×
Sliding Wedge	6	3	1.78	0.50	1.85	0.58	3.59×	3.21×
Wedges	6	3	7.57	1.95	7.81	2.33	3.88×	3.36×
5 Cubes (Figure 11)	40	30	34.3	5.88	36.0	7.52	5.83×	4.78×
Mat-Twist (Figure 12)	3.2K	9.1K	7.92	2.60	2567.06	2368.87	3.05×	1.08×
Mat-Knives (Figure 13)	3.2K	9.1K	86.1	25.2	178.1	146.7	3.42×	1.21×

## 8 SIMULATIONS

We utilize our GPU CCD algorithm inside the IPC algorithm [Li et al. 2020] implemented in PolyFEM [Schneider et al. 2019] by running several simulations on CPU2 with 8 threads for the simulation and GPU1 for the CCD (see Section 7.4 for a description of the architectures). We note that IPC requires computing the distances between primitives at the beginning of every time step, which we accelerate using our STQ broad phase algorithm. We run all the unit tests of Erleben [2018] presented in [Li et al. 2020, Figure 11] using the original implementation (using a spatial hash for broad phase and the parallel TI narrow phase CCD) and compare with our method (Table 2).

We run the five-cube stack example (Figure 11) that contains several resting contacts. Similar to the unit tests, as the meshes are extremely coarse when using our method, the simulation is 4.5× faster. When using denser meshes (Figure 12 has 9K tetrahedra) and the elastic deformations become more challenging, the non-linear elastic solver dominates the IPC runtime, and the speedup provided when using our method becomes less prominent; only 8% times faster overall. Our method naturally support CCD between codimensional object (Figure 13); for this scene, we again see similar speedups.

We also run the five-cube stack example (Figure 11), which contains several resting contacts. Similar to the unit tests, as the meshes are extremely coarse when using our method, the simulation is 4.5× faster. In the mat-twist shown in Figure 12, the denser mesh with more challenging elastic deformations causes the linear solver to dominate the overall runtime, and the speedup provided when using our method becomes less prominent (only 8% faster overall).

To stress-test our CCD algorithm, we show our method is able to handle CCD between codimensional objects in Figure 13. In this scene, we again see similar speedups as Figure 12 with a 21% speedup overall.

<sup>7</sup><https://opendata.blender.org/>

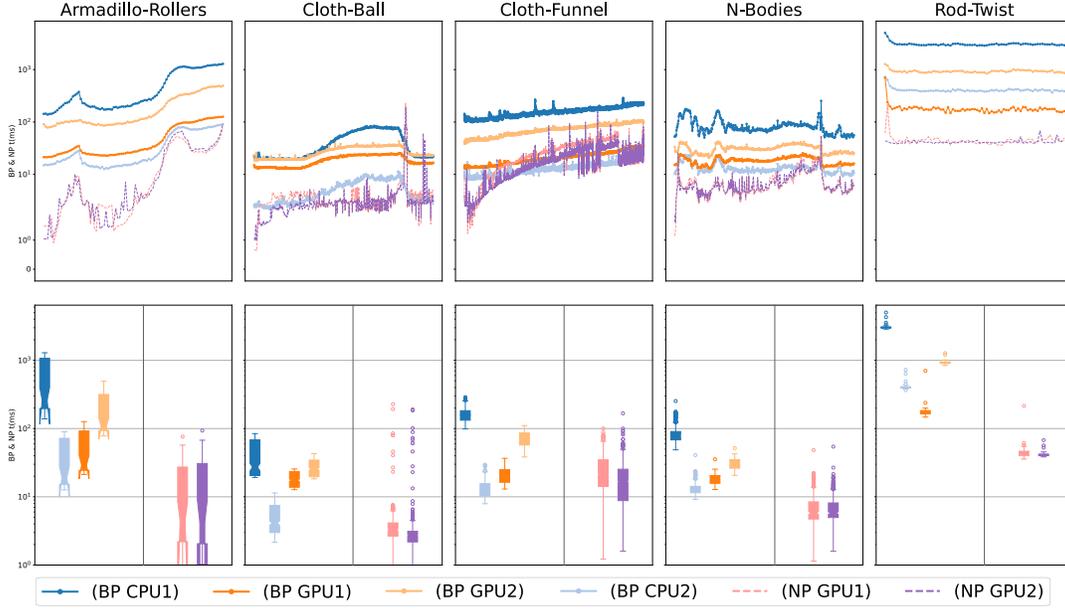


Fig. 8. Results of our method on the consumer architectures (CPU1 and GPU1) and the professional architectures (CPU2 and GPU2) for five different scenes (columns). For each scene (Figure 2), we report the performance for every simulation frame (first row) and timings box plot (second row).

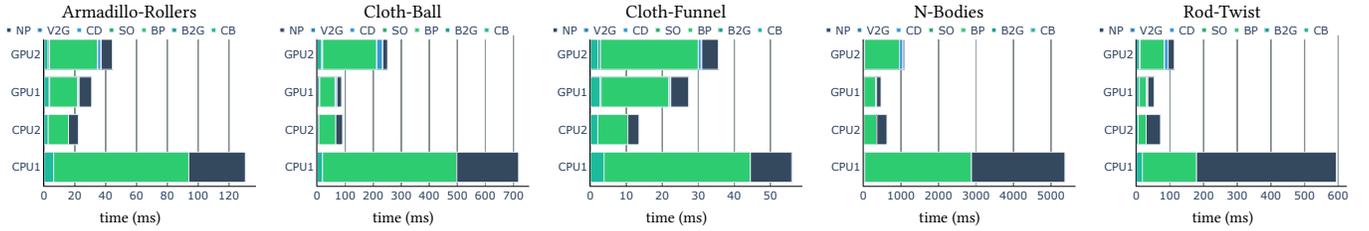


Fig. 9. Cutoff of the runtime for every scene for our method on the different architectures. NP runs the narrow phase, V2G copies the vertices to the GPU; SO splits the queries into vertex-face and edge-edge, BP is the broad phase; CD constructs the data list; B2G copies the boxes to the GPU; and CB constructs the boxes. Section 5 details the different architectures.

*Avoiding Time of Impacts Equal to Zero.* An important caveat of using a conservative CCD method inside the IPC algorithm is that it should not produce a time of impact  $t^*$  of zero. A  $t^*$  of zero causes the non-linear solver to stagnate because IPC uses the  $t^*$  to determine the maximum step size allowable inside the optimization-based implicit time-stepping. Additionally, IPC guarantees every step results in an intersection-free state, so  $t^*$  cannot be zero (i.e., not initially intersecting).

An exact CCD method would provide this guarantee. However, we use a conservative method, so even if the objects are not touching, a naïve implementation can produce a time of impact of zero.

To avoid  $t^* = 0$ , we make slight modifications to Algorithms 1 and 3 (Appendix B). As part of the strategy to avoid  $t^* = 0$ , the IPC algorithm uses a minimum separation in the CCD to prevent taking a step that results in parts exactly touching. We choose the minimum separation relative to the initial distance  $d_0$  between the query’s primitives. We use  $0.2d_0$  in all our experiments. To implement minimum separation CCD, we use the same strategy

as TI [Wang et al. 2021]: we enlarge the box  $C_\epsilon$  by the minimum separation distance.

## 9 CONCLUSION

We introduce a novel dataset that provides a way to check for the correctness of CCD codes and their time of impact in different settings. We believe it is a realistic and practical approach to evaluate the conservativeness of CCD implementations, even if passing the benchmark is not a formal proof of correctness. It helped us design our algorithm; using it, we found counter-examples for other CCD codes. The benchmark is easy to extend, and we plan to keep it up to date and add more scenes and challenging queries in the following years.

Based on the benchmark, we designed a novel scalable CCD algorithm combining broad and narrow phase collision detection. Our algorithm is provably conservative, and our implementation has been tested on multiple combinations of recent operating systems

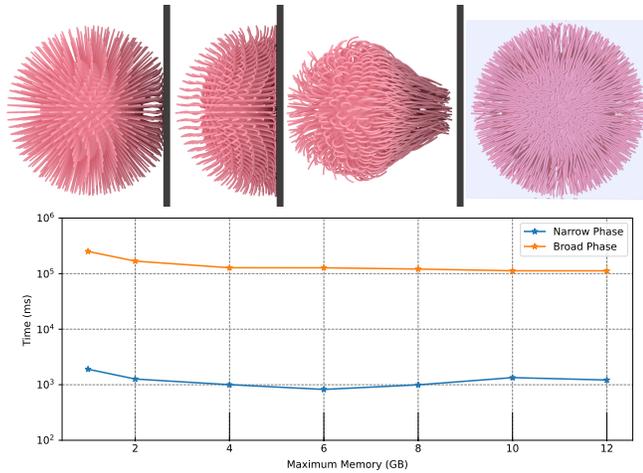


Fig. 10. Runtime of our algorithm for a scene with 13 billion queries (top) for different amounts of memory (bottom). The original scene data and renderings are courteous of Li et al. and were generated as part of [Li et al. 2020] using the IPC method.

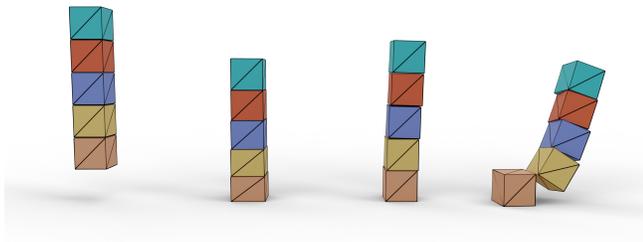


Fig. 11. Several frames of the five-cube stack example. The whole scene has only 30 tetrahedra and is, therefore, CCD bound. We see a 5.83× speed-up in CCD and 4.78× overall compared to using SH and TI.

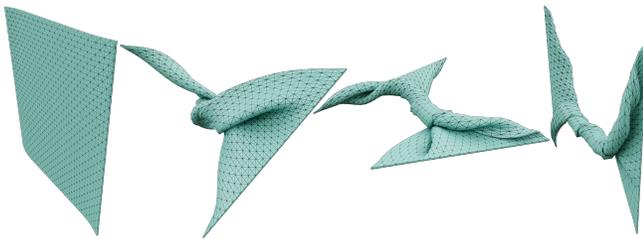


Fig. 12. Several frames of a mat-twist simulation utilizing our scalable continuous collision detection (CCD) algorithm. We see a 3.05× speed-up in the CCD and a 1.08× speed-up overall compared to using SH and TI.

and hardware architectures. Our algorithmic contribution specifically targets parallel architectures with high memory bandwidth (and high latency), which have very different requirements than traditional serial architectures. Our algorithm scales well to GPU hardware: an NVIDIA<sup>®</sup> 3080 Ti GPU (MSRP ~1.2K USD) achieves a speed comparable to a CPU server chip with 64 cores/128 threads

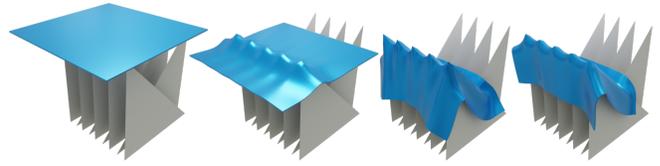


Fig. 13. Several frames of a codimensional simulation. The mat has 9K tetrahedra, while the codimensional triangles are not deformable. We see a 3.42× speed-up in the CCD and a 1.21× speed-up overall compared to using SH and TI.

(MSRP ~10K USD). We believe that our GPU algorithm could be extended to run on multi-GPU. Our preliminary experiments show that the broad phase becomes 2.3 times faster when using 4 GPUs for the N-Bodies scene.

When integrated with the state-of-the-art solver IPC, our approach reduces the overall simulation time, which we believe is of practical relevance to the graphics and simulation community. Our implementation will be released on GitHub with the MIT license to foster its adoption in academia and industry.

## REFERENCES

- David Baraff. 1992. *Dynamic simulation of non-penetrating rigid bodies*. Technical Report. Cornell University.
- Robert Bridson, Ronald Fedkiw, and John Anderson. 2002. Robust Treatment of Collisions, Contact and Friction for Cloth Animation. *ACM Transactions on Graphics* 21, 3 (July 2002), 594–603.
- Tyson Brochu, Essex Edwards, and Robert Bridson. 2012. Efficient geometrically exact continuous collision detection. *ACM Transactions on Graphics* 31, 4, Article 96 (July 2012), 7 pages.
- G. Capannini and T. Larsson. 2016a. Adaptive Collision Culling for Large-Scale Simulations by a Parallel Sweep and Prune Algorithm. In *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization (Groningen, The Netherlands) (EGPGV '16)*. Eurographics Association, Goslar, DEU, 1–10.
- Gabriele Capannini and Thomas Larsson. 2016b. Efficient Collision Culling by a Succinct Bi-Dimensional Sweep and Prune Algorithm. In *Proceedings of the 32nd Spring Conference on Computer Graphics (SCCG '16)*. Association for Computing Machinery, New York, NY, USA, 25–32.
- Gabriele Capannini and Thomas Larsson. 2018. 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. <https://doi.org/10.1109/TVCG.2017.2709313>
- Girolamo Cardano. 1545. *Artis Magnae, Sive de Regulis Algebraicis Liber Unus*.
- Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav Ponamgi. 1995. I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics (Monterey, California, USA) (I3D '95)*. Association for Computing Machinery, New York, NY, USA, 189–ff.
- Erwin Coumans and Yunfei Bai. 2016–2019. PyBullet, a Python module for physics simulation for games, robotics and machine learning. <http://pybullet.org>.
- Sean Curtis, Russ Gayle, Naga Govindaraju, Ilknur Kabul, Ming Lin, Simon Pabst, Stephane Redon, Avneesh Sud, Min Tang, Sung-eui Yoon, Jieyi Zhao, and Dinesh Manocha. 2012. UNC Dynamic Scene Benchmarks. <http://gamma.cs.unc.edu/DYNAMICB>.
- Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE* 5, 1 (1998), 46–55.
- Kenny Erleben. 2018. Methodology for Assessing Mesh-Based Contact Point Methods. *ACM Transactions on Graphics* 37, 3, Article 39 (July 2018), 30 pages.
- Zachary Ferguson, Minchen Li, Teseo Schneider, Francisca Gil-Ureta, Timothy Langlois, Chenfanfu Jiang, Denis Zorin, Danny M. Kaufman, and Daniele Panozzo. 2021. Intersection-Free Rigid Body Dynamics. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 40, 4, Article 183 (July 2021), 16 pages.
- Marco Hutter and Arnulph Fuhrmann. 2007. Optimized continuous collision detection for deformable triangle meshes. (July 2007).
- Lutz Kettner, Andreas Meyer, and Afra Zomorodian. 2016. Intersecting Sequences of dD Iso-oriented Boxes. [https://doc.cgal.org/latest/Box\\_intersection\\_d/index.html](https://doc.cgal.org/latest/Box_intersection_d/index.html)

- Byungmoon Kim and Jarek Rossignac. 2003. Collision prediction for polyhedra under screw motions. *Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications*, 4–10.
- Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M. Kaufman. 2020. Incremental Potential Contact: Intersection-and Inversion-Free, Large-Deformation Dynamics. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 39, 4, Article 49 (July 2020), 20 pages.
- Minchen Li, Danny M. Kaufman, and Chenfanfu Jiang. 2021. Codimensional Incremental Potential Contact. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 40, 4, Article 170 (2021).
- Fuchang Liu, Takahiro Harada, Youngeun Lee, and Young J Kim. 2010. Real-time Collision Culling of a Million Bodies on Graphics Processing Units. *ACM Transactions on Graphics* 29, 6 (Dec. 2010), 1–8.
- Brian Vincent Mirtich. 1996. *Impulse-Based Dynamic Simulation of Rigid Body Systems*. Ph. D. Dissertation.
- Xavier Provot. 1997. Collision and self-collision handling in cloth model dedicated to design garments. In *Computer Animation and Simulation*. Springer, 177–189.
- Stephane Redon, Abderrahmane Kheddar, and Sabine Coquillart. 2002. Fast Continuous Collision Detection between Rigid Bodies. *Computer Graphics Forum* 21 (May 2002).
- Teseo Schneider, Jérémie Dumas, Xifeng Gao, Denis Zorin, and Daniele Panozzo. 2019. Polyfem. <https://polyfem.github.io/>.
- Ygor Rebouças Serpa and Maria Andréia Formico Rodrigues. 2019. Flexible use of temporal and spatial reasoning for fast and scalable CPU broad-phase collision detection using KD-Trees. In *Computer Graphics Forum*, Vol. 38. Wiley Online Library, 260–273.
- Ygor Rebouças Serpa and Maria Andréia Formico Rodrigues. 2020. Broadmark: A Testing Framework for Broad-Phase Collision Detection Algorithms. *Computer Graphics Forum* 39, 1 (2020), 436–449.
- John M. Snyder. 1992. Interval Analysis for Computer Graphics. *Computer Graphics (Proceedings of SIGGRAPH)* 26, 2 (July 1992), 121–130.
- John M. Snyder, Adam R. Woodbury, Kurt Fleischer, Bena Currin, and Alan H. Barr. 1993. Interval Methods for Multi-Point Collisions between Time-Dependent Curved Surfaces. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (Anaheim, CA) (SIGGRAPH '93)*. Association for Computing Machinery, New York, NY, USA, 321–334.
- Min Tang, Young Kim, and Dinesh Manocha. 2009. C<sup>2</sup>A: Controlled Conservative Advancement for Continuous Collision Detection of Polygonal Models. 849–854.
- Min Tang, Young J. Kim, and Dinesh Manocha. 2010. Continuous collision detection for non-rigid contact computations using local advancement. In *2010 IEEE International Conference on Robotics and Automation*. 4016–4021.
- Min Tang, Zhongyuan Liu, Ruofeng Tong, and Dinesh Manocha. 2018a. PSCC: Parallel Self-Collision Culling with Spatial Hashing on GPUs. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 1, Article 18 (July 2018), 18 pages.
- Min Tang, Dinesh Manocha, Sung-eui Yoon, Peng du, Jae-Pil Heo, and Ruofeng Tong. 2011. VolCCD: Fast continuous collision culling between deforming volume meshes. *ACM Transactions on Graphics* 30 (Jan. 2011), 111.
- Min Tang, Ruofeng Tong, Zhendong Wang, and Dinesh Manocha. 2014. Fast and Exact Continuous Collision Detection with Bernstein Sign Classification. *ACM Transactions on Graphics* 33 (Nov. 2014), 186:1–186:8. Issue 6.
- Min Tang, tongtong wang, Zhongyuan Liu, Ruofeng Tong, and Dinesh Manocha. 2018b. I-Cloth: Incremental Collision Handling for GPU-Based Interactive Cloth Simulation. *ACM Transactions on Graphics* 37, 6, Article 204 (Dec. 2018), 10 pages.
- Daniel J Tracy, Samuel R Buss, and Bryan M Woods. 2009. Efficient large-scale sweep and prune methods with AABB insertion and removal. In *2009 IEEE Virtual Reality Conference*. IEEE, 191–198.
- Brian Von Herzen, Alan H. Barr, and Harold R. Zatz. 1990. Geometric Collisions for Time-Dependent Parametric Surfaces. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques (Dallas, TX, USA) (SIGGRAPH '90)*. Association for Computing Machinery, New York, NY, USA, 39–48.
- Bolun Wang, Zachary Ferguson, Xin Jiang, Marco Attene, Daniele Panozzo, and Teseo Schneider. 2022. Fast and Exact Root Parity for Continuous Collision Detection. *Computer Graphics Forum* (2022). <https://doi.org/10.1111/cgf.14479>
- Bolun Wang, Zachary Ferguson, Teseo Schneider, Xin Jiang, Marco Attene, and Daniele Panozzo. 2021. A Large-Scale Benchmark and an Inclusion-Based Algorithm for Continuous Collision Detection. *ACM Transactions on Graphics* 40, 5, Article 188 (Sept. 2021), 16 pages.
- Wolfram Research Inc. 2020. *Mathematica 12.0*. <http://www.wolfram.com>
- Xinyu Zhang, Minkyung Lee, and Young J. Kim. 2006. Interactive Continuous Collision Detection for Non-Convex Polyhedra. *Vis. Comput.* 22, 9 (sep 2006), 749–760.
- Xinyu Zhang, Stephane Redon, Minkyung Lee, and Young J. Kim. 2007. Continuous Collision Detection for Articulated Models Using Taylor Models and Temporal Culling. *ACM Transactions on Graphics* 26, 3 (July 2007), 15–es.
- Afra Zomorodian and Herbert Edelsbrunner. 2000. Fast Software for Box Intersections. In *Proceedings of the Sixteenth Annual Symposium on Computational Geometry (Clear Water Bay, Kowloon, Hong Kong) (SCG '00)*. Association for Computing Machinery,

New York, NY, USA, 129–138.

## A SPATIAL HASH VOXEL SIZE

The performance and memory footprint of SH heavily depends on the voxel size (Figure 14). We use  $v = 2 \max(d_0, d_1)$ , with  $d_0$  the average edge length and  $d_1$  the average displacement as an empirical heuristic for our experiments. Using smaller or larger voxels leads to an increase in runtime (with large voxels being the worst); however, too small voxels consume too much memory and eventually run out of memory (for  $v/20$ ).

## B ZERO TIME OF IMPACT AND MINIMUM SEPARATION

To avoid  $t^* = 0$ , we make slight modifications to Algorithms 1 and 3. As in [Li et al. 2020], if Algorithm 1 return  $t^* = 0$  we perform the narrow-phase again but set the minimum separation to 0 and enable a no zero ToI strategy.

This no-zero ToI strategy dictates that if  $I_t^l = 0$ ,  $I$  should always be split (ignoring user tolerances and the maximum number of splits). We note that under floating-point division this split might not be possible, but this has not been encountered in practice and would most likely require a degenerate case involving tiny distances (which IPC does a good job of preventing thanks to its barrier potential method of handling contacts). In the end, because the minimum separation was disabled, the resulting  $t^*$  is multiplied by a scaling factor less than 1 to avoid exactly touching after the step (we use 0.8 in our examples).

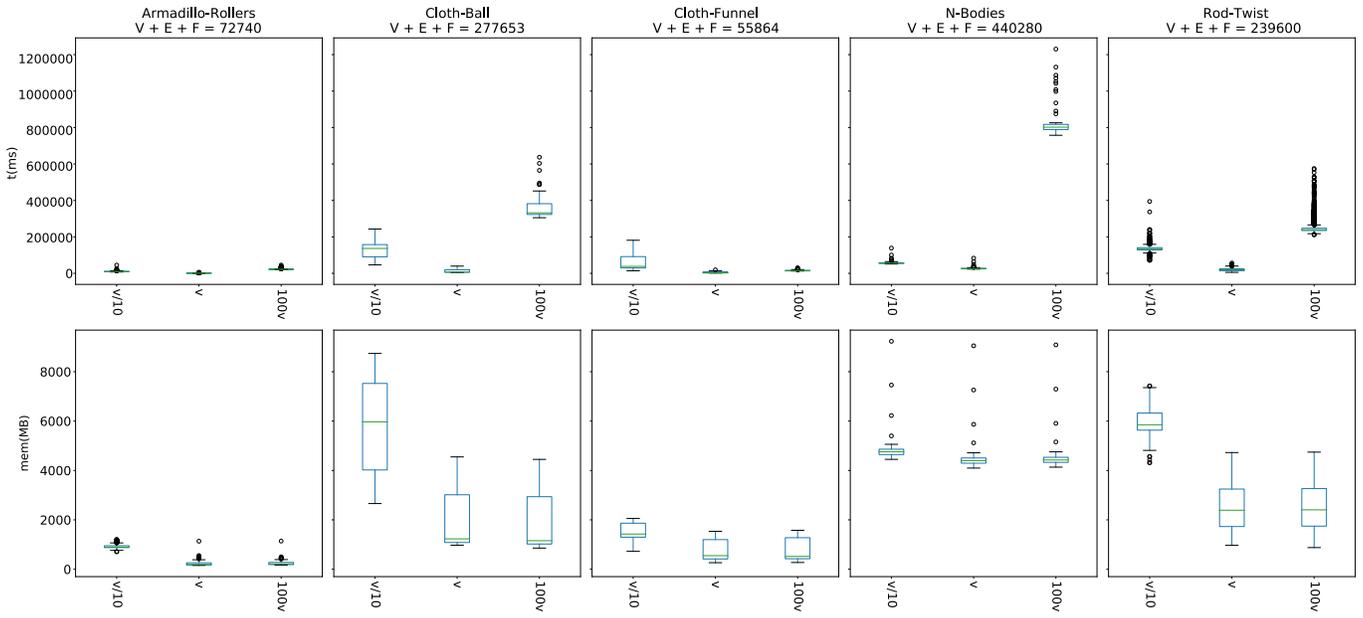


Fig. 14. **SH Voxel Size.** We benchmark SH using three different values for the voxel size and plot the timing (top) and memory (bottom). We use a voxel size of  $v/10$ ,  $v$ , and  $100v$ , where  $v$  is the heuristic size used throughout all of our experiments. We also experiment with smaller voxel sizes but run out of memory for even  $v/20$  (max 64 GB) due to many duplicate collision candidates.