# Chunked Bounding Volume Hierarchies for Fast Digital Prototyping using Volumetric Meshes

Robert Schmidtke and Kenny Erleben

**Abstract**—We present a novel approach to using Bounding Volume Hierarchies (BVHs) for collision detection of volumetric meshes for digital prototyping based on accurate simulation. In general, volumetric meshes contain more primitives than surface meshes, which in turn means larger BVHs. To manage these larger BVHs, we propose an algorithm for splitting meshes into smaller chunks with a limited-size BVH each. Limited-height BVHs make guided, all-pairs testing of two chunked meshes well-suited for GPU implementation. This is because the dynamically generated work during BVH traversal becomes bounded. Chunking is simple to implement compared to dynamic load balancing methods and can result in an overall two orders of magnitude speedup on GPUs. This indicates that dynamic load balancing may not be a well suited scheme for the GPU. The overall application timings showed that data transfers were not the bottleneck. Instead, the conversion to and from OpenCL friendly data structures was causing serious performance impediments. Still, a simple OpenMP acceleration of the conversion allowed the GPU solution to beat the CPU solution by 20%. We demonstrate our results using rigid and deformable body scenes of varying complexities on a variety of GPUs.

**Index Terms**—Collision Detection, Bounding Volume Hierarchies, Parallel Tandem Traversal, Scheduling Algorithm

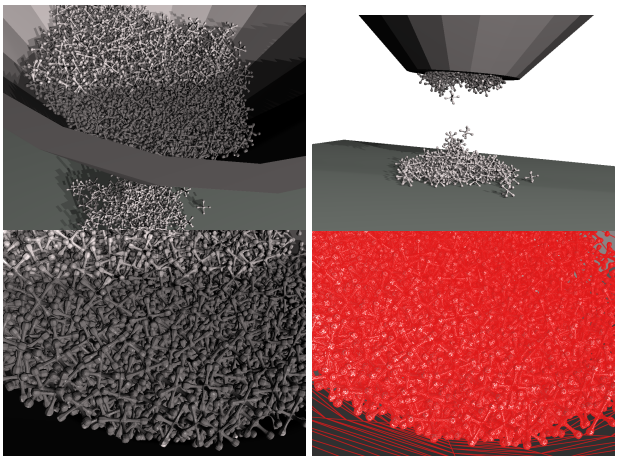

Fig. 1: Detailed contact modeling in digital Prototyping. 2000 jack shapes jamming in a funnel with an average of $5.5 \cdot 10^5$ contact points.

## 1 Introduction

Bounding Volume Hierarchies (BVHs) present an efficient way of reducing the complexity of exact collision tests between two objects. Traversing the hierarchies is easily parallelizable, offering a significant performance improvement for collision detection in large scale simulations. Because of their high instruction and memory throughput, as well as their versatility, GPUs are becoming increasingly popular platforms for parallelizing BVH traversal. The challenge is to properly utilize the GPU's processing units because parallel traversal of two BVHs may generate work dynamically, making it difficult to evenly distribute the workload.

Our interest is the accurate physical simulation of structures subject to external forces, taking their respective materials, densities and rigidities into account. This is required in digital prototyping to analyze jamming in assembly lines, safety and risk assessments from impact scenarios, cultural preservation of building structures, or compaction of chalk grains. Digital prototyping cases are shown in Figure 2 and supplementary movies.[1] Another area where internal object structure is vital to the correctness of a simulation is exact force computation on deforming bodies, as found in biomechanical modeling of tissues or structural components subject to external forces, exemplified by the cantilever tower scene in Figure 6e.

We present an algorithmic contribution that removes the need for run-time workload balancing on many-core systems such as GPUs. We achieve consistent hardware utilization and performance by distributing enough work over all processing units of a GPU, while limiting the amount of work that is allowed to grow dynamically. We show that this approach is superior in performance to the more commonly employed approach of active workload balancing, outperforming it in all cases in terms of kernel invocations, application execution times and overall pair-wise collision detection times. It also addresses the possibly reduced pruning capabilities of BVHs for volumetric objects because of the larger number of primitives they contain.

**Our contributions:**

- Division of volumetric objects into layers of spatially close fixed-size chunks for building multiple BVHs per object.
- Scheduling chunks of two layers for all-pairs testing for proper GPU hardware utilization.


- *R. Schmidtke is with Zuse Institute Berlin, Germany. E-mail: schmidtke@zib.de*
- *K. Erleben is with University of Copenhagen, Denmark. E-mail: kenny@di.ku.dk*


*Manuscript received April 19, 2005; revised August 26, 2015.*

1. https://www.youtube.com/playlist?list=PLNtAp--NfuirBjOiudmmnzhn4U-3n6Bcl

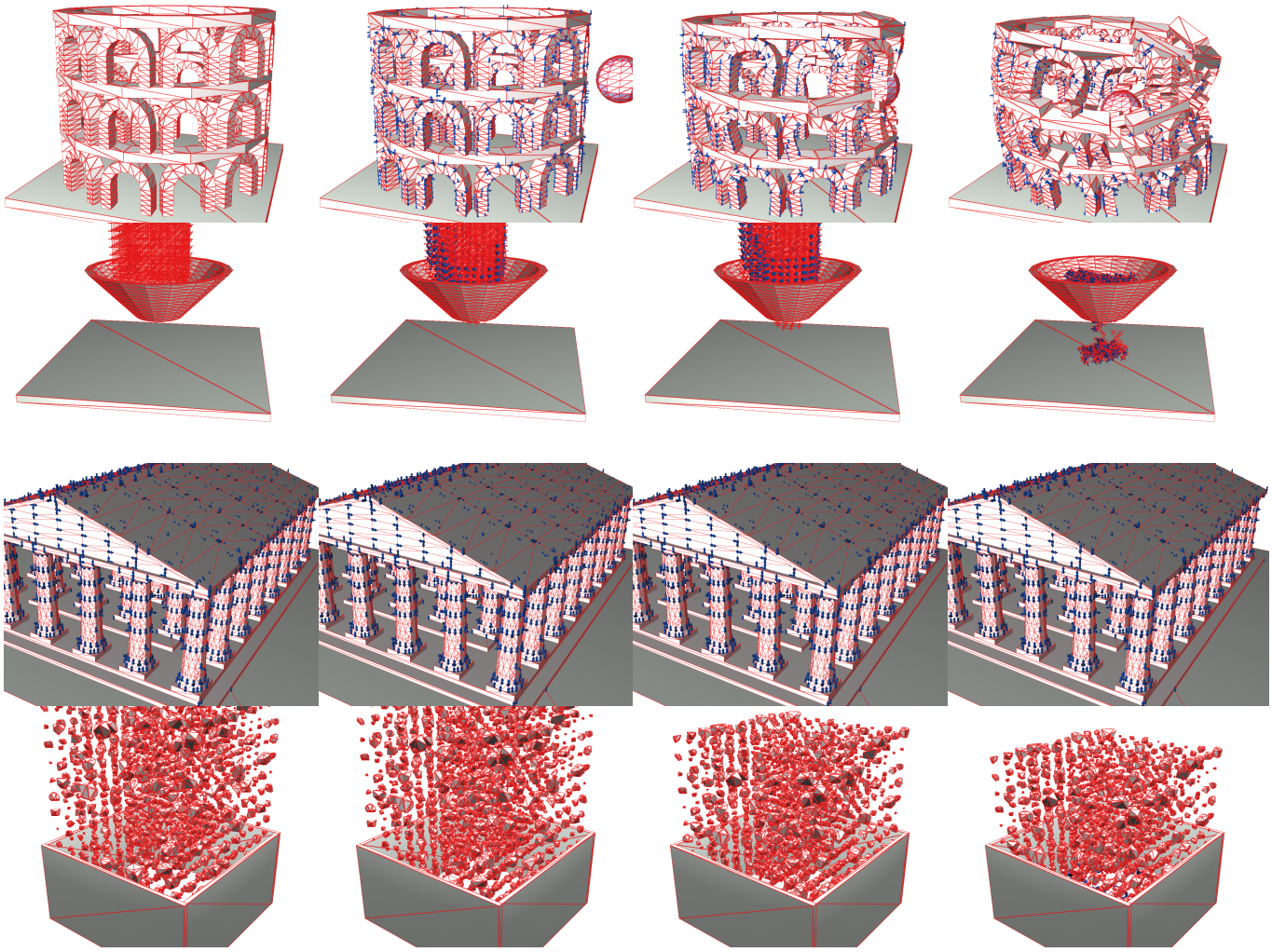

Fig. 2: Large scale rigid body simulations; top: Meteor Impact Risk Assessment, mid-upper: Jamming in Digital Prototyping, mid-lower: Cultural Preservation Building Collapse, and bottom: analysis of grain chalk compaction. The meteor scene has 1082 rigid bodies comprised of $1 \cdot 10^4$ tetrahedrons in total with an average of $8 \cdot 10^4$ contacts; the funnel jamming scene has 1002 rigid bodies and $2.3 \cdot 10^6$ tetrahedrons in total with an average of $3.51 \cdot 10^5$ contacts.

- Avoiding complex workload balancing schemes through traversal of fixed-size BVHs, addressing the issue of having to distribute work dynamically at runtime.
- Greatly reducing kernel invocation overhead by dispatching up to three orders of magnitude fewer traversal kernels that current approaches require for workload balancing.

In addition to the performance advantages, our algorithm is simpler, hence straightforward to implement, and therefore less exposed to logical errors, easier to debug and deploy.

Related work on BVHs and GPU-based collision detection is presented in Section 2. Section 3 presents the details of reordering a tetrahedral mesh, addresses aspects of *chunking* and gives details on how to use OpenCL to implement parallel BVH traversal. Section 4 describes benchmark results of an OpenCL implementation of our approach compared to an OpenCL implementation of another popular and widely used approach. Section 5 concludes this article and suggests promising areas for future work.

## 2 Related Work

The field of collision detection covers a large body of work, as presented in surveys [1], [2]. However, this work considers only mid-phase and narrow-phase collision detection, whereas we focus solely on BVHs, as we consider them to be agile and versatile when applied to general non-convex geometries, whether rigid or deformable. Furthermore, their accuracy is only limited by machine precision. BVHs are hence a perfect match for general purpose digital prototyping based on accurate simulation with general non-convex shapes. Our goal is to push for fast accurate contact modeling. This is important in digital prototyping, robotics and training simulators, as the predicted motion needs to be more accurate than the usual desired plausible motion quality known from entertainment oriented contexts. Hence we omit methods based on approximations [3]–[6] and image-based approaches [7] from further analysis.

BVHs are often used to improve performance of collision detection for both rigid body and deformable object simulation [4], [8], [9]. However, there are also examples of other

application areas, such as ray tracing [10], [11] and motion planning [12]. Most work on BVH collision detection for simulation concerns surface representations such as triangle meshes. This work focuses on tetrahedral mesh representations which are common for finite element/volume method simulations [13], [14] and other applications [15], [16]. The major advantage of the volumetric mesh approach lies in exploiting the volume information for more robust contact point generation. This has been recognized for years by the gaming community, as their approach to non-convex general shapes decomposes them into smaller convex ones. The extreme version of the approach would be to consider a general non-convex rigid body as a decomposition into tetrahedrons as done in Bullet [17]. This approach is typical for many fast simulation applications. If we only cared about actual intersection testing, then the volumetric approach would be uncalled for. However, we found the volumetric approach to be easy to work with for contact point generation, as many others have [14], [18]–[23].

Past work on BVHs is extensive, and we refer the reader to the surveys for comprehensive past coverage [1], [2]. More recent work covers approximating sphere-trees [4] and redundant element-wise tests [24], continuous collision detection expressing element-wise motions by higher order polynomials [25] and Bernstein basis to ensure reliable testing [26]. Extensions to deal with topological changes are found in [27]. "Chunking" of surface meshes can be found in [6]. This work embeds a triangle mesh in a coarse tetrahedral mesh with an associated smooth surface representation for contact normal generation. The triangle surface mesh is clipped against the embedding tetrahedron and a small "chunked" bounding volume hierarchy is created.

In terms of traversing BVHs on GPUs, algorithms such as gProximity [28] and its derivatives [29] are the only sensible choice to compare the approach presented in this work to, as it provides insight on active load balancing versus static load balancing using spatial decomposition. We present a detailed comparison to gProximity using a variety of benchmarks in Section 4. To our knowledge, the class of gProximity methods is still the de-facto representative of active load balancing algorithms for BVH tandem traversals on GPUs. Another approach without explicit load balancing is presented in [30].

There has been a lot of work targeting GPUs as main computing platforms for collision detection implementations. Early attempts use visibility queries to determine whether objects are in close proximity [31], [32]. Hybrid CPU/GPU approaches have been proposed using BVHs [33] and spatial subdivision [34]. State-of-the-art papers apply layered depth images (LDIs) [5], however such techniques are limited by image-space resolution and tend to address approximating contact manifolds. Works before the advent of general purpose programming frameworks for GPUs (like CUDA and OpenCL) model the collision detection problem in terms of graphics domains (e.g. representing input as textures) to exploit the computing power of the graphics pipeline, e.g. to build distance fields [35], [36].

Considerable work on (possibly multi-level) BVHs, to which our approach is similar, has been performed in the past, especially within the area of ray tracing using the surface area heuristic (SAH) [37]–[39]. Using GPU support, in [40], the objects' BVHs are used as leaves in one BVH for the entire scene, effectively implementing a broad phase. A combination of SAH-trees on top of Morton clusters produced by the HLBVH approach is presented in [41]. Bonsai [42] uses rough partitioning into triangle groups to build mini-SAH-trees, on top of which a regular BVH is constructed. Recent approaches include partial merging of low-level BVHs to reduce their overlaps in the high-level BVH [43], as well as using a quickly constructed auxiliary BVH of the entire scene to build a final, high-quality BVH through refinement [44].

## 3 Chunked Bounding Volume Hierarchies

Collision detection is a major bottleneck in simulation applications [1], [2], [45]. Because GPUs have long outpaced CPUs in theoretical maximum memory and instruction throughput [28], [29], [46], they form a suitable platform to target this bottleneck. BVH accelerated algorithms allow computation of high-detailed contact information, while not limiting the number of possible applications: they can be used for rigid/deformable bodies, continuous/discrete collision detection, intra/inter-object intersections and surface/volume meshes. We chose $k$-DOPs in order to have a single parameter family of volumes where $k$ can intuitively be increased to provide a tighter fitting volume. Empirically we determined $k = 8$, which we observed to fit memory layout well while providing good all-around pruning capability for our specific cases. Note that the specific choice of $k$ is irrelevant in regards to our contribution in this work, and our algorithms can be used with any volume type.

BVHs need to cover more geometry for volumetric meshes compared to the surface-only counterparts, due to the added interior geometric elements. This results in deeper hierarchies, and if they are not carefully laid out, many unnecessary tests need to be performed between non-overlapping internal parts of two objects, which reduces overall performance through many false positives. We obtain efficient memory access by sorting data in a way that favors contiguous memory access. Specifically, the memory sorted array of a mesh's tetrahedrons is *chunked* into $C$ fixed sized chunks, each covering $L$ tetrahedrons. By recursively using chunking in a bottom up fashion (combining $L$ contiguous chunks into one super-chunk), we create a kind of $C$-nary BVH data structure where its nodes are binary $k$-DOP BVHs. The large $C$-nary branch factor in higher levels combined with a simultaneous descend rule (see Section 3.2) increases the workload per work unit. Further, super-chunking provides an efficient accelerated mid-phase of our approach. By limiting each chunk's size to $\mathcal{O}(L)$, the fast shared limited-size memory of a GPU is guaranteed to not overflow (see Section 3.1). This reduces costly kernel relaunches because of memory exhaustion.

Note that while other works have employed multi-level BVHs as well [40], [42], [43], their focus was on two levels: the lower level for the objects in the scene, the higher level for connecting them into the scene, which may result in deep hierarchies. Our approach, on the other hand, explicitly takes into account the hardware specifications, chunking the scene into appropriately sized BVHs to allow for efficient processing on GPUs with as little host-device memory transfer as possible, resulting in possibly more than two levels.

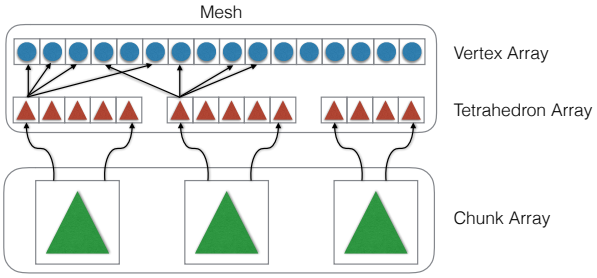


Fig. 3: Graphical illustration of how mesh data and chunks are ordered in memory. Each green triangle represents a binary $k$-DOP BVH – a chunk. A Level 0 chunk covers $L$ tetrahedrons, illustrated by red triangles.

This is opposed to previous GPU-accelerated multi-level BVH approaches [6], [42].

We achieve memory and topological locality by a reordering of tetrahedrons, such that tetrahedrons that are spatially close in the mesh are also stored in the same area of memory. Because of this reordering, the mesh can be trivially split into multiple connected chunks of tetrahedrons while ensuring that each chunk occupies a contiguous part in memory as well. Figures 3, 4 and 5 graphically illustrate the layout of the data structure.

### 3.1 Building and Updating Chunked Bounding Volume Hierarchies

Our approach is inspired by a bottom-up hierarchy construction. A new level in the hierarchy is created by merging spatially close nodes from the current level into disjoint sets. A tight enclosing volume is created for each individual set and becomes the new parenting node in the new level. Solving this problem optimally in terms of a classical tightness measure favoring fat non-overlapping volumes [47] is equivalent to solving the weighted set packing problem and hence is an intractable NP-complete problem. To be efficient we settle for a suboptimal solution: We exploit the spatial adjacency of tetrahedrons. We transform the problem of finding subsets into mapping the mesh elements into a linear order and sorting them, before creating equal-sized chunks by chopping up the sorted elements in contiguous sequences of same size. This is simple and fast while keeping suboptimal localization. Furthermore, we use the new order to actually re-order elements in memory to ensure fast memory access. We experimented with graph traversals such as breadth first traversals of the tetrahedrons to exploit adjacency between tetrahedrons. However, we observed that generated volumes tended to lose the overall desired fatness and non-overlapping properties. Instead we decided to use space-filling curves. We found Z-curves to be more implementation friendly and easy to apply in 3D than Hilbert curves for example, while still providing us with sufficient localization of the tetrahedral elements as others too have discovered [37] (see Figure 5 for a comparison between breadth-first and Z-ordering).

Initially, the mesh is stored as an unordered array of tetrahedrons, each containing indices to a vertex array. The $M$ tetrahedrons (and their vertices) are reordered along the *Z-order curve* using *Morton codes*, which is a common technique to obtain spatial proximity between primitives [37], [41], [42], [44]. Sorting the tetrahedrons according to their Morton codes results in an array where neighboring tetrahedrons in memory are not too far away from neighboring tetrahedrons in the mesh. The now ordered array can then be split into $C$ *chunk*s of $L$ tetrahedrons each, where each chunk represents a spatially close part of the mesh due to choice of using a space-filling curve, see Figure 5. Having found the chunks, we have identified the leaves of our BVHs. Following this we switch to a top-down approach for generating the connectivity of the BVH, which we will detail in the following paragraphs.

To calculate the optimal chunk size $L$, and the number of chunks $C$, we first determine the maximum number of BVH nodes we can fit in memory of size `memsize` bytes:

$$N_{\text{max}} = \left\lfloor \frac{\text{memsize}}{\text{node size in bytes}} \right\rfloor . \tag{1}$$

Next we divide $N_{\text{max}}$ by two because we want to be able to fit two BVHs into memory to be able to traverse them simultaneously. `memsize` is manually chosen taking into account the local memory size on the GPU (see Section 3.3). The *perfect* number of BVH nodes needed in a complete balanced binary tree (BBT) is then the next smallest power of 2, minus 1:

$$N_{\text{perfect}} = 2^{\lfloor \log(N_{\text{max}}/2) \rfloor} - 1 \tag{2}$$

The number of tetrahedrons in each chunk is bounded by the number of leaves in a BBT with $N_{\text{perfect}}$ nodes:

$$L = \frac{N_{\text{perfect}} + 1}{2} \tag{3}$$

A mesh with $M$ tetrahedrons is split into $C$ chunks:

$$C = \left\lceil \frac{M}{L} \right\rceil \tag{4}$$

A BBT is built for each chunk top-down by halving the chunk's tetrahedron array at each step. At the bottom level, each tetrahedron's $k$-DOP is computed. The bounding volumes (BVs) for each node in a chunk's hierarchy are constructed in a bottom-up manner by combining their minimum and maximum extents along each of the $k$ axes. This process is repeated on the newly created chunks, grouping $L$ of them together in a new layer of *super chunk*s with a BVH each, until the root level contains less than $L$ chunks. This way, very large objects with millions of tetrahedrons can be accounted for. The root node $k$-DOP BVs are combined into a single $k$-DOP that encloses the entire mesh, see Figure 4.

This way, instead of constructing a *single* BVH, *multiple* BVHs are constructed for each sub mesh, each BVH covering a *fixed size* subset of the tetrahedrons. These subsets contain tetrahedrons within sufficiently close proximity such that the BVHs do not cover overlapping parts of the overall mesh. For each BVH leaf node, an index into the tetrahedron array is stored so the bounded tetrahedron can be referred to later on. Because the tetrahedrons are in Z-order, the chunks and super chunks are in a semi Z-order as well, because due to contiguity of chunks, the super chunks inherit some spatial locality from the chunks they each cover. Because the topology of the bodies we employ stays fixed even for

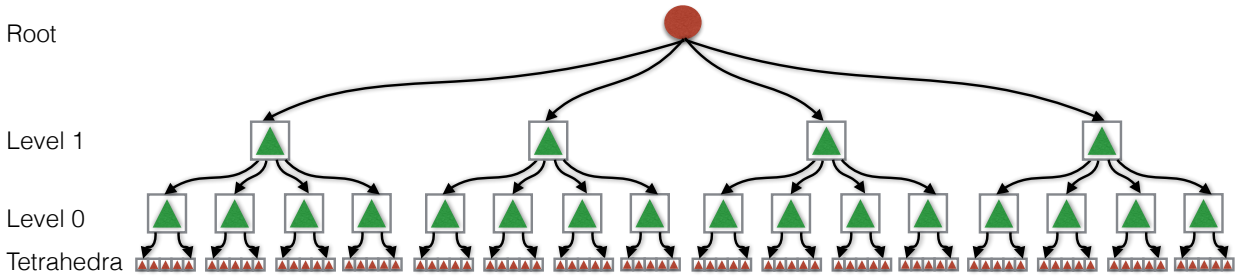


Fig. 4: Graphical illustration of the super chunks data structure: each green triangle represents a binary $k$-DOP BVH – a chunk. A Level 0 chunk covers $L$ tetrahedrons and a chunk on level $n > 0$, we call a super chunk, covers $L$ child chunks. The circle is a single root $k$-DOP. Each level is stored in arrays and has a large branching factor between levels.

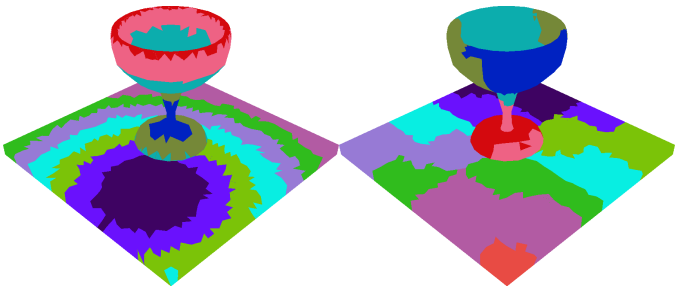

Fig. 5: Mesh ordering comparison for 256 KiB large chunks. Left: breadth first traversal results in chunks with large overlaps in bounding volumes. Right: Z-order curve chunking in 3D gives sufficient fat and disjoint subset of tetrahedrons.

deformable bodies, as we do not allow breaking or self-intersection, the division of meshes into objects remains the same and does not require updating. The whole process is outlined in Algorithms 1, 2 and 3.

**Data:** $T$: Array of tetrahedron elements;
**Result:** $L_0, L_1, \ldots, L_{n-1}$: Arrays of BVHs; $R$ root volume;
$L_0 \leftarrow$ Create-level-0 from $T$;
$n \leftarrow 1$;
**while** *number of chunks in* $L_{n-1} > L$ **do**
  $L_n \leftarrow$ Create-level-$n$ from $L_{n-1}$;
  $n \leftarrow n + 1$;
**end**
$R \leftarrow$ generate root node of $L_{n-1}$;
Update all BVs bottom-up;

**Algorithm 1:** Build-chunked-BVH. Notice the mixing of top-down and bottom-up approaches. The ordering allows for using top-down splitting by halving sub-arrays recursively for generating the connectivity structure. The actual geometry of bounding volumes are computed in bottom-up fashion after the structure has been created.

Our construction method supports a fast bottom-up update of the BVHs, exploiting memory locality during the update. This is unlike other work describing fast dynamic updates of $k$-DOP trees [48]. Our approach is more similar to what is typically done for axis-aligned bounding boxes (AABBs) when applied to deformable models [9]. We have

**Data:** $T$: Array of tetrahedron elements;
**Result:** $L_0$: Array of BVHs;
Allocate space for $L_0$;
Reorder $T$ array by Z-curve;
Chop $T$ array into sub-arrays;
**foreach** *sub-array* $A$ **do**
  $B \leftarrow$build BVH by top-down splitting of $A$;
  Insert $B$ to end of $L_0$;
**end**

**Algorithm 2:** Create-level-0 algorithms reorders mesh data and create a sequence of fixed size chunks that forms level 0 in the super chunk data structure.

**Data:** $L_{n-1}$: Array of BVHs;
**Result:** $L_n$: Array of BVHs;
Allocate space for $L_n$;
Chop $L_{n-1}$ array into sub-arrays;
**foreach** *sub-array* $A$ **do**
  $B \leftarrow$build BVH by top-down splitting of $A$;
  Insert $B$ to end of $L_n$;
**end**

**Algorithm 3:** Create-level-$n$ works similar to how level 0 is created. The idea is applied recursively to chunks instead of tetrahedrons.

implemented updating of our BVHs in OpenCL, with results reported as *BVH refitting* times in Section 4. We exploited the inherently parallel nature of this task and used a version similar to the common partial sum reduction technique for refitting the BVHs bottom-up.

Using binary BVHs rather than hierarchies of higher cardinality is motivated mainly by the ease of implementation. We use the tree cardinality when computing $N_{\text{perfect}}$ in equation (2), so changing it will not affect our algorithm, as long as two BVHs fit into a limited memory region simultaneously. In Section 3.3 we give a brief argument that supports favoring low-cardinality BVHs due to memory efficiency.

To assess the quality of the generated BVHs, in ray tracing the surface area heuristic (SAH) is used [38], [41], [42], [44]. Often, the underlying primitives are sorted using Morton codes as well. However, as noted in [37], the resulting BVHs are not always optimized for ray tracing, because primitives along the Morton curve may not always be spatially close.

This can be seen in Figure 5, where 256 KiB large chunks may sometimes span a large area. In practice, we use 8 KiB chunks, giving a far more fine-grained segmentation of objects, reducing the risk for any individual chunk to be overly large. In our approach, tandem traversal is needed instead of single traversal, with the probability of intersection increasing with the volumes of objects (and the granularity of the volumes' representations), rather than their surfaces. Furthermore, penetrations are allowed in our scenarios to determine contact response.

This is opposed to ray tracing applications, which are mostly concerned with rendering, rather than physically accurate simulation, as can be seen in commonly used benchmarks,[2] where each frame is precisely defined. Nonetheless, we are aware of the problem using the Morton curve as sole measure for determining the residents of each chunk, and therefore suggest to evaluate a hybrid SAH approach in future work.

### 3.2 Tandem Traversal

In order to define units of work in the traversal that can be easily processed in parallel, we use a bounding volume test tree (BVTT) [47], [49]. Unlike the common approach of descending into the largest volume first [50], we employ a simultaneous descend rule. This reduces thread divergence through shorter paths to the leaves of the BVTT, while at the same time providing enough work to parallelize. We focus on accelerating the tandem traversal using GPUs. Usually, when two objects' root BVs overlap, the root nodes' children are tested in an all-pairs manner, making up the BVTT. Using our approach, there are multiple options to seed the tandem traversal. All "level-zero" chunks of the two objects could be scheduled in an all-pairs manner; all "level-one" chunks etc. The higher the level of which the chunks are scheduled for pair-wise testing, the fewer initial pair-wise tests, see Figure 4.

The chunk level to seed pair-wise testing is determined based on the resulting number of pair-wise tests. If both objects contain more than the one chunk level, and the number of pair-wise tests in that level is "too high", the level above is chosen. This is repeated until either one object has no higher super chunk level, or the number of pair-wise tests has fallen below a threshold value. We experimentally defined this threshold for GPUs to be $10^7$ (e.g. $10^3 \times 10^4$ for objects of different sizes).[3] The resulting number of initial pair-wise tests therefore ranges between $10^7/L$ and $10^7$. This efficiently implements a *mid phase* algorithm to collision detection for large objects. For each pair of chunks, we have to traverse two BVHs in parallel, as part of either the mid phase or the narrow phase of collision detection, depending on the chunk level used. For two objects' levels $A$ and $B$ with $C_A$ and $C_B$ chunks, respectively, there are $C_A \times C_B$ BVHs to test for overlap using tandem traversal. The mid phase algorithm is illustrated in Algorithm 4.

Instead of processing each pair of chunks one after the other, all chunk pairs of the appropriate levels are collected for all collision candidates first. They are then used as input to the GPU tandem traversal kernel as BVH root node pairs

2. http://gamma.cs.unc.edu/DYNAMICB/
3. Section 3.3 discusses the impact of changing this threshold.

**Data:** $A, B$: Chunked BVHs; $K$: Threshold for initial pair-wise tests
**Result:** $S$: Array of pair-wise tests
$n \leftarrow \max$ level of $A$ and $B$;
$C_A \leftarrow$ number of chunks in level $n$ of $A$;
$C_B \leftarrow$ number of chunks in level $n$ of $B$;
**while** $n > 0$ *and* $C_A \times C_B > K$ **do**
  $C_A \leftarrow$ number of chunks in level $n$ of $A$;
  $C_B \leftarrow$ number of chunks in level $n$ of $B$;
  $n \leftarrow n - 1$;
**end**
$S \leftarrow$ all pairs from level $n$ of $A$ and $B$.

**Algorithm 4:** Mid phase collision detection for seeding tandem traversal.

(BVTT root nodes). The kernel's output are BVTT leaf nodes that either reference pairs of tetrahedrons or pairs of lower-level chunk pairs. Pairs of tetrahedrons are processed in a dedicated contact point generation kernel, while BVTT nodes referencing lower-level chunks are used as input to the tandem traversal kernel again. This situation arises when super chunks are chosen to initially seed the tandem traversal. Note that keeping the data on the GPU as long as possible has been used in previous approaches as well, e.g. using collision streams [30].

For sufficiently large scenes with many objects in close proximity, multiple kernel launches cannot be avoided. This is because the hardware memory is limited, which prevents keeping all BVTT nodes in device global memory at once (a limitation in [30]). This problem is inherent to collision detection algorithms that use BVTTs to guide BVH traversals, especially a parallel implementation working on many BVTT nodes simultaneously. To help this all-pairs memory problem, we have applied chunking recursively, motivated by the pruning capabilities of BVHs on multiple levels, with sufficiently locality within chunks and super chunks because of the underlying Z-order.

### 3.3 OpenCL Implementation of Tandem Traversal

We have adopted OpenCL terminology in accordance with [51], equivalent CUDA terms are provided where the two terminologies are conflicting.

A work item is defined as testing two BVs for overlap and subsequently scheduling the appropriate child overlap tests (if needed). In other words, a work item is the procession of one BVTT node. Since OpenCL does not allow recursion, we place stacks that hold BVTT nodes into a work group's local memory (in CUDA: shared memory). Initially, each work item fetches one BVTT root node from global memory (CUDA: device memory) using its global identifier, and pushes it onto its local work stack. As long as there is work on the local stack, each work item pops a test pair $\{a, b\}$ and processes it. If $a$ and $b$ overlap, the work item generates the appropriate child test pairs $\{c_a, c_b\}$ and pushes them onto its stack. If the stack is empty, either the work item fetches the next BVTT root node from global memory, or exits if there is no more work to do. Each work item has access to a dedicated region of the stack, removing the need for inter-work-item synchronization using local atomics and barriers. Work items access

this region in a strided fashion to reduce the risk of bank conflicts, using their local identifier as offset and the work group size as step size. Overlapping BVTT nodes are collected in dedicated global memory using a global append buffer. A dedicated contact point generation kernel uses the append buffer as input, and finds overlaps between tetrahedrons. If a pair of super chunk levels has been chosen initially, BVTT leaf nodes reference chunks in a lower chunk layer. If the overlap test is positive, the chunk leaf test pairs are collected in a global memory region, using a global append buffer. Future traversal kernel invocations will process these pairs in order to not overflow the local stack.

Using a stack to store intermediate nodes effectively implements a depth-first traversal of a BVTT. The BVTT's size is bounded by $\mathcal{O}(\delta \log_\delta n)$, where $\delta$ is the degree of the BVTT (4 in our case since we use binary BVHs), and $n$ is the number of nodes in the BVTT. This places a bound on the maximum amount of local memory that a work item uses. We note that small $\delta$ and $n$ decrease the per work item maximum stack memory requirements. This allows work groups with many work items (recall all work items have their stacks in a work group's local memory (CUDA: shared memory)), which is desirable for proper hardware utilization. Using chunks of limited size (thereby limiting the BVHs' sizes and ultimately the BVTTs' sizes, $n$) and binary BVHs (small $\delta$) achieves precisely that.

Chunking generates sufficiently many limited-size BVTTs to provide enough lightweight work for the hardware. This avoids having too many idle threads, which in turn removes the need for active workload balancing and costly kernel relaunches. With gProximity, a separate workload balancing kernel is launched whenever it is detected that too many threads sit idle while others have too much work (see Section 4).

We use a global append buffer for copying BVTT nodes back to global memory. gProximity collects BVTT nodes in dedicated global memory regions for each work group, ending up with possibly scattered memory that needs compaction. We obtain a contiguous memory segment that can be used directly as input to the contact point computation kernel or to another invocation of the tandem traversal kernel, and therefore can remain on the device. This comes at the cost of using global atomics for the append buffer. The overhead of an additional compaction kernel in the pipeline must therefore be balanced with the performance loss by global atomics. However, recent GPUs have drastically improved execution of global atomic operations [52].

## 4 Results

We compare an OpenCL implementation of our approach to an OpenCL version of gProximity [28] which we have implemented ourselves using the CUDA source code made publicly available.[4] As argued in Section 2, gProximity is the most opposite alternative approach to load balancing compared to our work. Furthermore, as far as we are aware it is still considered to be one of the fastest approaches and is well documented [29]. For our gProximity implementation we chose to use one $k$-DOP BVH covering the whole reordered surface mesh. We use the same construction approach as we use for the level-0 chunks to eliminate bias due to hierarchy construction when comparing static vs. dynamic load balancing schemes.

4. gProximity: http://gamma.cs.unc.edu/GPUCOL/

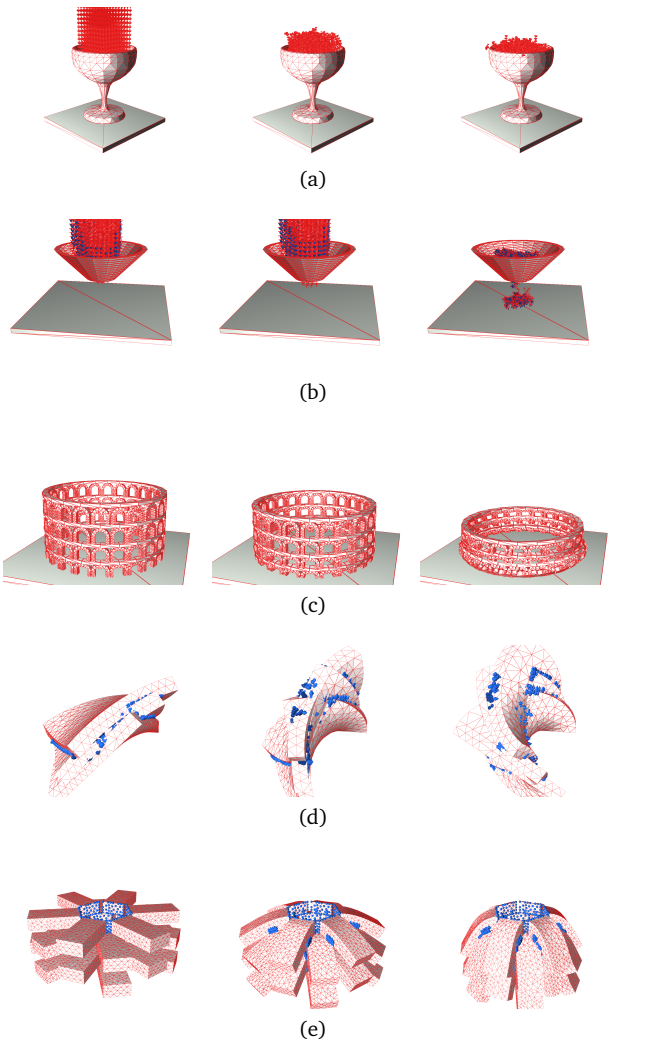


Fig. 6: Our test scenes: (a) Falling **Glasses** (rigid), (b) **Funnel** jamming (rigid), (c) **Colosseum** collapse (rigid), (d) **Plates** stack (deformable), and (e) cantilever **Tower** (deformable).

While OpenCL and CUDA may exhibit different performance characteristics on NVIDIA hardware due to their levels of abstraction, we have opted for an OpenCL rewrite of gProximity instead of integrating the CUDA version. The first reason behind this is that had we not wanted to be platform-independent, we would have implemented our kernels in CUDA as well. This might have resulted in better performance, both for the gProximity version as well as ours, however we believe that the differences in performance would not have changed dramatically. Another reason is that the relatively small part of gProximity that we are comparing against was portable to OpenCL in a straightforward way, whereas adapting our kernels, as well as host-device interactions with gProximity, would have been considerably more involved.

Because we focus on computing correct physical behavior between objects, our implementation describes scenes using

TABLE 1: Test scene specifications. The columns show the total number of objects, vertices, tetrahedrons in the scene, as well as the total number of contacts and the median of contacts per simulation step.

| Name | Objects | Vertices ($10^3$) | Tetrahedrons ($10^3$) | Contacts ($10^3$) | Med. Contacts per Step ($10^3$) |
|---|---|---|---|---|---|
| Glasses | 2002 | 1102.6 | 3365.7 | 437558.4 | 1279.7 |
| Funnel | 2002 | 1596.7 | 4676.1 | 333400.4 | 1276.8 |
| Colosseum | 1729 | 14.8 | 16.1 | 68260.2 | 162.9 |
| Plates | 3 | 1.7 | 5.2 | 15521.2 | 18.7 |
| Tower | 24 | 10.2 | 37.8 | 398058.2 | 22.1 |

TABLE 2: GPU specifications partially collected from `clinfo` and CUDA device query, and product homepages. All numbers have been truncated to lowest whole unit.

| Specification | Tesla C2075 | Tesla K20c | Tesla K40c | Firepro W8100 |
|---|---|---|---|---|
| Compute Units | 14 | 13 | 15 | 40 |
| CUDA Cores | 448 | 2496 | 2880 | 2560 |
| Max Clock Frequency (MHz) | 1147 | 705 | 745 | 824 |
| Global Memory (GB) | 5 | 5 | 11 | 8 |
| Local Memory (KiB) | 48 | 48 | 48 | 32 |
| Memory Bandwidth (GB/s) | 144 | 208 | 288 | 320 |
| Single Precision (GFLOPS) | 1030 | 3524 | 5040 | 4300 |

volumetric objects. That is why input that can be properly tetrahedralized is required. Accordingly, we present results for three rigid body benchmarks (Figures 6a through 6c) and two deformable body benchmarks (Figure 6d and Figure 6e) of varying complexity as shown in Table 1, and on multiple GPUs shown in Table 2.

We have measured overall application timings and overall computation sub-tasks using CPU timers. Pre- and post-processing times for conversion to/from OpenCL friendly data structures from/to native internal data structures of the simulator are measured using CPU timers as well. Kernel execution times and GPU read and write times are measured using OpenCL counters. The simulation framework we use is based on the work in [53].

Table 3 shows that up to 70-80% of time can be spent on collision detection for complex scenes (see Table 1). For the less complex Colosseum scene with two orders of magnitude fewer vertices and tetrahedrons, as well as an order of magnitude fewer contacts, we observe an inversion of percentages for the time spent in the solver and in collision detection. This is because of the wide spread of objects during the collapse of the building, where the broad phase is able to eliminate many pair-wise tests, as there are only few objects within close proximity. In the glasses and funnel scenes there are many objects within close proximity at all times, resulting in a lot of time spent in collision detection compared to the solver.

The solver runs single-threadedly on the host and iterates for a high number of times until convergence. Simulation times are 3-4 hours for one test scene of 5 seconds of simulated time (500 simulation steps of 0.01 seconds each). Using double precision would likely double the GPU timings or worse because of the additional memory needed on the device, entailing more kernel relaunches, as well as the reduced throughput compared to single precision, eventually causing substantially longer running times and skewing the ratio between solver and collision detection even more. Hence, from a practical viewpoint, single precision is the only real option due to the still large discrepancy between peak performance of double and single precision.

TABLE 3: Total time and percentages for overall processes in the simulator for different test scenes using our Chunked approach. Initialization includes transformation and refit computations. Narrow phase includes traversal and contact generation times. Note that generally speaking the whole collision detection takes up to 70-80% of the computation time for complex scenes. This proves that a large percentage of computation time is spent on collision detection. If the solver was GPU accelerated instead of using a single-threaded CPU core, then the ratio might be even higher. The analysis clearly indicates that it is the narrow-phase collision detection that is the major bottleneck.

| Name | Arch. | Total time (min.) | Solver | Initialization | Broad Phase | Narrow Phase | Contact Reduction |
|---|---|---|---|---|---|---|---|
| | | | (% of Total time) | | | | |
| Glasses | C2075 | 213.32 | 27.42 | 4.44 | 0.03 | 58.5 | 4.65 |
| | K20c | 211.48 | 27.63 | 4.47 | 0.03 | 58.14 | 4.74 |
| | K40c | 127.7 | 26.87 | 7.03 | 0.03 | 58.3 | 2.71 |
| | W8100 | 99.22 | 26.48 | 4.51 | 0.04 | 61.73 | 2.12 |
| Colosseum | C2075 | 13.94 | 58.08 | 1.98 | 0.35 | 21.09 | 7.19 |
| | K20c | 13.83 | 58.6 | 1.17 | 0.35 | 21.12 | 7.32 |
| | K40c | 7.26 | 57.52 | 4.85 | 0.39 | 20.51 | 4.31 |
| | W8100 | 6.33 | 58.66 | 0.82 | 0.38 | 24.21 | 4 |
| Funnel | C2075 | 243.9 | 16.37 | 5.36 | 0.04 | 72.34 | 2.67 |
| | K20c | 245.6 | 16.32 | 5.3 | 0.04 | 72.43 | 2.68 |
| | K40c | 158.83 | 13.5 | 7.67 | 0.04 | 74.49 | 1.33 |
| | W8100 | 134.95 | 16.16 | 4.43 | 0.04 | 75.4 | 1.1 |

Table 3 illustrates why collision detection of general shaped polygonal models is still a major concern in terms of being a performance bottleneck. Our work favors the GPU acceleration positively, and still the numbers are poor.

Table 4 shows speedup factors of Chunked BVH compared to gProximity. Detailed time measurements of individual computational tasks and read/write data transfers are shown in Table 5. Chunked BVH outperforms gProximity due to two major reasons. The *dynamic descend biggest volume first rule* used in gProximity is going to avoid more unnecessary tests to be done. In comparison the *descend all simultaneously rule* used in Chunked BVH is going to generate a few more bounding volume pair tests than necessary, but this descend rule makes code simpler, and provides for better hardware utilization. This implies we expect slightly larger traversal timings when using Chunked BVH than gProximity. However, Chunked BVH is superior as it does not need to do any dynamic load balancing and can save the time for this part. The gProximity method has a significant read-overhead from the many kernel launches done by the dynamic balancing. For this reason Chunked BVH clearly outperforms gProximity.

Note the traversal and contact times for the Glasses scene: Here both gProximity and Chunked exhibit very large values. This is likely due to the fact that, as shown in Figure 9, the number of contacts in the examined steps 20 to 122 is a lot higher for the glasses scene compared to the funnel

TABLE 4: Speedup comparison between gProximity and Chunked BVH methods. GPU read and write measure total data transfer times for each method. Narrow phase is the overall speedup of the whole narrow phase collision detection, and total is the whole simulation time. Speedup factors are rounded to one decimal place. Kernel launches indicate the factor by which Chunked BVH uses fewer launches compared to gProximity.

| Name | Arch. | Total | Narrow Phase | GPU Read | GPU Write | Kernel Launches |
|---|---|---|---|---|---|---|
| Funnel | C2075 | 5.8 | 14.8 | 1 | 39.4 | 34.9 |
| | K20c | 8.7 | 23.2 | 1 | 63.5 | 94.7 |
| | K40c | 16.4 | 54.1 | 0.9 | 106.3 | 94.3 |
| | W8100 | 3.2 | 6.3 | 1 | 17 | 4.6 |
| Glasses | C2075 | 27.1 | 30.8 | 1 | 364.4 | 381.3 |
| | K20c | 43.3 | 49.2 | 1.2 | 594.8 | 1666.2 |
| | K40c | 102.1 | 124.8 | 1.1 | 923.5 | 1748.3 |
| | W8100 | 9.9 | 11.1 | 1 | 141.2 | 16.1 |
| Plates | C2075 | – | 22.9 | 1.4 | 1963.4 | 10.5 |
| | K20c | – | 26.3 | 2 | 2761.2 | 19.2 |
| | K40c | – | 60.8 | 2.1 | 3590.6 | 17.9 |
| | W8100 | – | 41.7 | 1.2 | 200.1 | 2.6 |
| Tower | C2075 | – | 33.9 | 1.2 | 600.3 | 16 |
| | K20c | – | 52.5 | 2.1 | 1045.9 | 96.4 |
| | K40c | – | 132.8 | 1.6 | 1831.4 | 85.8 |
| | W8100 | – | 6.1 | 1.2 | 208.3 | 2.1 |

scene, resulting in a lot more overall work. We observe that contact times are different between gProximity and Chunked. This is curious as both algorithms produce the exact same pairs of tetrahedrons that are fed into the contact generation kernel. The large discrepancy in values is caused by measuring the overhead due to many more contact kernel invocations in the gProximity variant, plus underutilization of the hardware. Compared to the Chunked algorithm, after each traversal/balancing kernel invocation pair, the gProximity variant will check if the contact generation kernel needs to be invoked. This implies the contact generation kernel could be invoked as many times as traversal/balance kernels are invoked. In the Chunked variant all data is generated before invoking the contact generation kernel.

Figures 7 and 8 clearly show that the bottleneck of gProximity is the traversal write data transfers timings and their relation to the number of kernel launches.

The Chunked BVH method behaves quite different than gProximity and has quite different performance bottlenecks. Comparing Figure 9 and Figure 10, we notice that the timing of all contact and traversal related tasks appear to scale monotonically with the number of contacts. Further we notice that transform and refit tasks appear to be constant and independent of the number of contact points, suggesting these depend on the number of objects in the scene.

Further, Chunked BVH does not suffer from being penalized by read and write data transfer times as shown in Table 6. Data transfers are never more than 5%-7% of the total running time. Clearly the W8100 has better transfer times and all NVIDIA architectures have similar characteristics. The K40c is slightly better than the K20c and the C2075. Again comparing to Table 2 we expect only a small difference between the W8100, the K40c and the K20c. However, there should be a big difference between the C2075 and any of the others by around a factor of two. We do observe a proper relative ranking W8100 < K40c < K20c, but unexpectedly we notice K20c $\approx$ C2075.

Table 7 shows that the actual traversal computation is the bottleneck of the GPU computations. The overall performance bottleneck is the traversal pre-processing as Figures 10 and 11 show.

Figure 12 confirms that AMD and NVIDIA architectures are working differently for the Chunked BVH method as well. This is expected as the W8100 has more compute units and slightly smaller memory, see Table 2. Figure 11 shows that it is mostly pre- and post-processing and data transfers that make the difference between the W8100 and K40c platforms. The actual GPU computations are very similar in magnitudes as expected due to the similar compute power.

Looking naively at single precision peak performance in Table 2 suggests the K20c should be three times faster than the C2075, and the K40c and W8100 should be about 1.5 times faster than the K20c. Surprisingly, the bar plot in Figure 10 shows that the C2075 performs almost as well as the K20c. The K40c and W8100 do appear almost twice as fast as the K20c, as expected. This suggests that the hardware is under-utilized.

We evaluated different pair-wise test thresholds for seeding the tandem traversal (Section 3.2) for the Glasses scene on the W8100: 1, 10, 1000, $10^4$, $10^5$, $10^7$ and $10^9$. Results are shown in Figure 13. We observe a step up in traversal times when switching from $10^3$ to $10^4$ as a seed threshold value, because the objects contain 53 chunks each, and $10^3 < 53^2 < 10^4$, so a higher level is chosen. Obviously in this scene, there are not many pair-wise contacts, so it is beneficial to bail out of the traversal early, instead of scheduling more lower level BVTT nodes for testing. A more versatile implementation with respect to the scene's content would take into account all objects when determining the individual test-pairs' chunk levels to seed traversal, instead of one test-pair at a time. This way, given sufficient GPU memory, more high-level test pairs could be scheduled, instead of many low-level test-pairs, giving similar hardware utilization while avoiding unnecessary overlap checks. We estimate this to be a promising area of future work to improve generality of our approach.

## 5 Discussion and Conclusions

In our subjective opinion, Chunking is a simpler algorithm to implement and hence reduces both implementation time and risk for introducing bugs. Chunking results in overall better kernel performance and GPU write results and makes BVHs more versatile in regard to hardware utilization. Key-ingredients to achieve these results are chunking and reordering of volumetric meshes by Z-curves, simultaneous descents, intelligent seeding of traversals by determining pair-wise chunk-levels fitting the underlying hardware's computing power and memory capacity for initial mid-phase using an all-pairs manner seeding of the tandem traversals.

Our approach comes without the need for complex active workload balancing between threads and cores of a GPU, which greatly reduces kernel invocation overhead, resulting in a reduction of total narrow phase times down to 11% – 69%. At the same time, pure kernel execution times achieved

TABLE 5: gProximity comparison table. All timings show total time in units of seconds rounded to one decimal. We skipped first 20 steps to avoid side-effects from kernel compiling and then used the following 102 steps for our data.

| Name | Arch. | Method | Transform (s) | Refit (s) | Traversal (s) | Contact (s) | Balance (s) | Preprocess (s) | Postprocess (s) | Read (s) | Write (s) | Launches (#) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Funnel | C2075 | gProximity | 0.08 | 19.7 | 4.23 | 1.88 | 1.63 | 224.78 | 25.93 | 17.48 | 1206.57 | 19178 |
| | | Chunked | 0.08 | 11.91 | 3.87 | 1.89 | – | 157.03 | 19.95 | 18.37 | 28.86 | 518 |
| | K20c | gProximity | 0.05 | 14.9 | 3.77 | 0.9 | 2.83 | 227.39 | 25.71 | 18.35 | 1964.95 | 48801 |
| | | Chunked | 0.05 | 14.34 | 4.86 | 0.88 | – | 158.05 | 20.01 | 19.15 | 29.49 | 476 |
| | K40c | gProximity | 0.04 | 11.98 | 2.55 | 0.7 | 2.37 | 169.2 | 39.15 | 15.3 | 2693.37 | 42573 |
| | | Chunked | 0.04 | 11.44 | 1.35 | 0.65 | – | 88.56 | 14.01 | 15.78 | 23.09 | 408 |
| | W8100 | gProximity | 0.24 | 8.06 | 0.16 | 0.39 | 0.69 | 121.61 | 17.75 | 3.94 | 130.81 | 4475 |
| | | Chunked | 0.24 | 7.22 | 1.85 | 0.39 | – | 76.54 | 14.42 | 4.01 | 7.31 | 929 |
| Glasses | C2075 | gProximity | 0.07 | 10.99 | 28.49 | 5.46 | 4.5 | 1007.93 | 17.41 | 14.44 | 26222.22 | 210598 |
| | | Chunked | 0.07 | 7.75 | 52.3 | 2.02 | – | 774.91 | 17.11 | 13.73 | 72.21 | 566 |
| | K20c | gProximity | 0.04 | 9.18 | 78.13 | 6.39 | 33.07 | 1020.81 | 17.49 | 16.48 | 42525.13 | 876635 |
| | | Chunked | 0.04 | 9.39 | 60.95 | 1.06 | – | 778.57 | 17.01 | 14.22 | 71.51 | 528 |
| | K40c | gProximity | 0.03 | 7.27 | 53.64 | 4.84 | 26.69 | 631.29 | 26.97 | 14.34 | 68378.13 | 767814 |
| | | Chunked | 0.03 | 7.37 | 17.06 | 0.52 | – | 487.92 | 21.86 | 13.05 | 76.05 | 428 |
| | W8100 | gProximity | 0.16 | 6.42 | 1.64 | 2.87 | 3.57 | 522.04 | 13.41 | 2.78 | 2293.01 | 26059 |
| | | Chunked | 0.16 | 7.25 | 20.6 | 0.44 | – | 398.11 | 12.12 | 2.7 | 16.22 | 1696 |
| Plates | C2075 | gProximity | – | 0.08 | 0.47 | 0.15 | 0.48 | 0.22 | 0.6 | 0.03 | 43.65 | 3344 |
| | | Chunked | – | 0.06 | 0.71 | 0.15 | – | 0.2 | 0.55 | 0.02 | 0.02 | 306 |
| | K20c | gProximity | – | 0.06 | 0.48 | 0.09 | 0.51 | 0.21 | 0.57 | 0.04 | 60.08 | 6036 |
| | | Chunked | – | 0.07 | 1.1 | 0.1 | – | 0.21 | 0.56 | 0.02 | 0.02 | 306 |
| | K40c | gProximity | – | 0.05 | 0.3 | 0.06 | 0.44 | 0.1 | 0.23 | 0.03 | 95.93 | 5624 |
| | | Chunked | – | 0.06 | 0.56 | 0.06 | – | 0.1 | 0.22 | 0.01 | 0.03 | 306 |
| | W8100 | gProximity | – | 0.07 | 0.06 | 0.08 | 0.35 | 0.1 | 0.32 | 0.03 | 46.06 | 2392 |
| | | Chunked | – | 0.06 | 0.14 | 0.1 | – | 0.1 | 0.24 | 0.02 | 0.03 | 918 |
| Tower | C2075 | gProximity | – | 0.14 | 0.71 | 0.11 | 0.19 | 11.15 | 0.37 | 0.13 | 340.04 | 4890 |
| | | Chunked | – | 0.1 | 0.75 | 0.08 | – | 8.07 | 0.37 | 0.11 | 0.58 | 306 |
| | K20c | gProximity | – | 0.1 | 1.64 | 0.09 | 1.79 | 11.21 | 0.36 | 0.22 | 549.77 | 29524 |
| | | Chunked | – | 0.11 | 1.18 | 0.05 | – | 8.05 | 0.37 | 0.1 | 0.52 | 306 |
| | K40c | gProximity | – | 0.1 | 1.13 | 0.07 | 1.54 | 6.47 | 0.12 | 0.18 | 812.07 | 26250 |
| | | Chunked | – | 0.1 | 0.62 | 0.03 | – | 4.36 | 0.15 | 0.11 | 0.42 | 306 |
| | W8100 | gProximity | – | 0.06 | 0.08 | 0.03 | 0.38 | 7.02 | 0.16 | 0.04 | 32.34 | 2010 |
| | | Chunked | – | 0.06 | 0.14 | 0.05 | – | 9 | 0.35 | 0.03 | 0.13 | 918 |

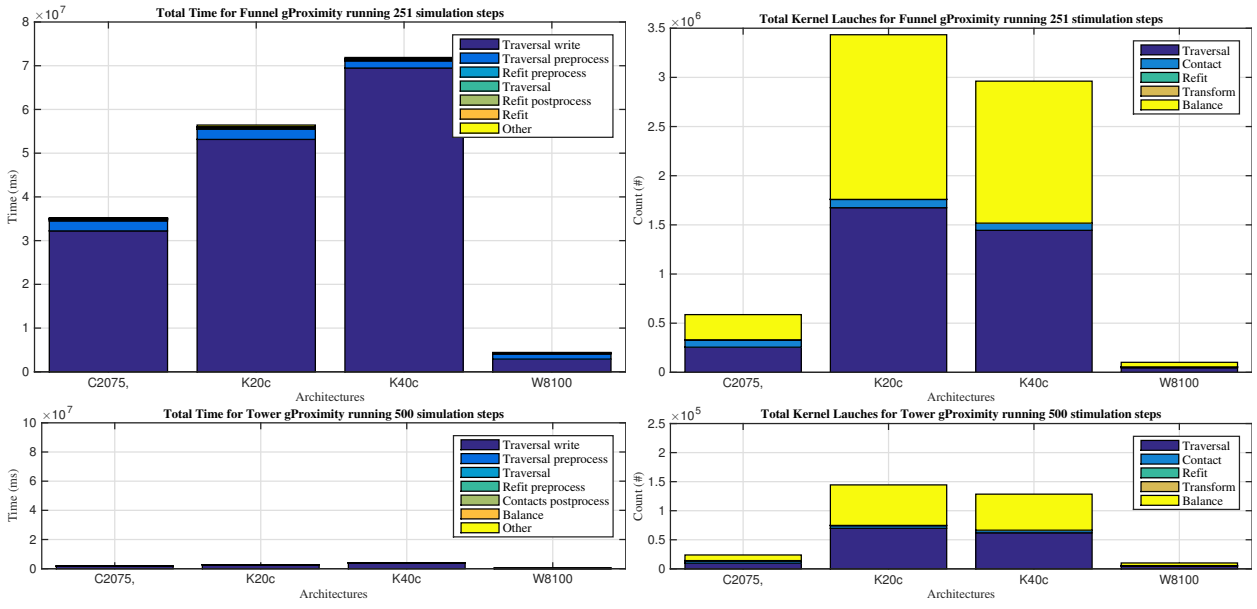


Fig. 7: gProximity bottlenecks analysis. Left shows the 6 major bottlenecks measured as a fraction of total running time. Right shows the total number of various kernel launches over the total running time. Clearly traversal write times are dominating gProximity due to the many kernel launches. Firepro W8100 behaves surprisingly different than any of the NVIDIA architectures.

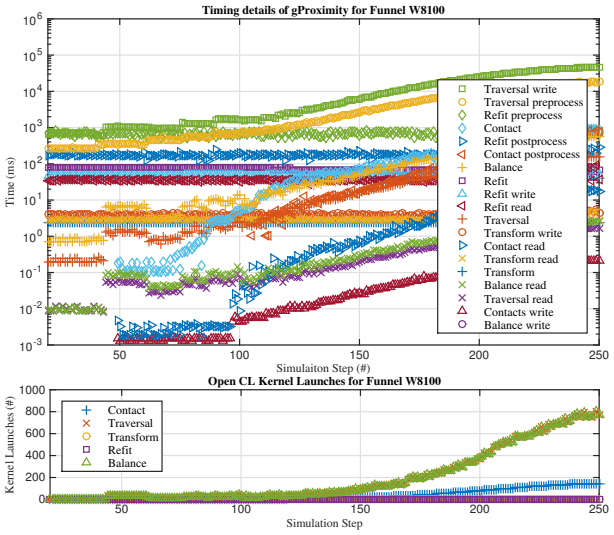


Fig. 8: gProximity detailed analysis for Funnel test scene on the best architecture. Top shows timings of all tasks as a function of simulation steps, and bottom shows corresponding number of kernel launches. In alignment with Figure 7, we observe the traversal write to be the performance bottleneck in each time step and the Firepro W8100's different IO performance shows clearly to be an advantage.

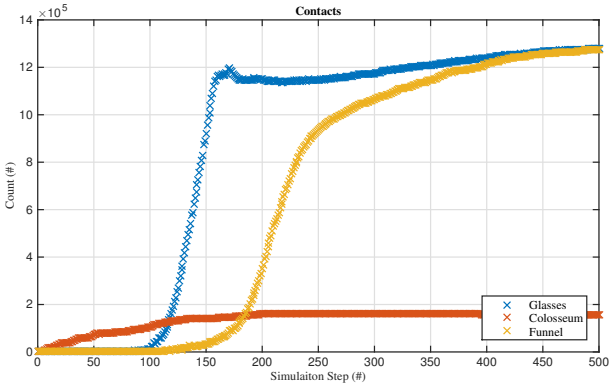


Fig. 9: Detailed contact statistics for selected test scenes. Comparing shapes of plots with detailed timings in Figure 11, we observe that device computations and read/write times scale with the number of contacts as would be expected. Hence the number of contacts is a good indicator of the problem size.

are on par with approaches that employ workload balancing for tandem traversal of BVHs for discrete collision detection on GPUs [28], outperforming them even when taking into account memory transfer times as well.

Using BVTT front tracking, an effectively similar approach to implicit workload balancing is presented in [30], working on collision streams to avoid host-device transfers. However, the entire front, in addition to all model data, needs to be stored on the GPU. Our approach ultimately works on streams as well (see Section 3.2), however it allows for BVTT fronts that, in conjunction with the model data, exceed the GPU's memory. They report approximately 9 % of their time is spent on refitting and updating BVHs and exact tests, while our approach spends approximately 21 % of the time on these

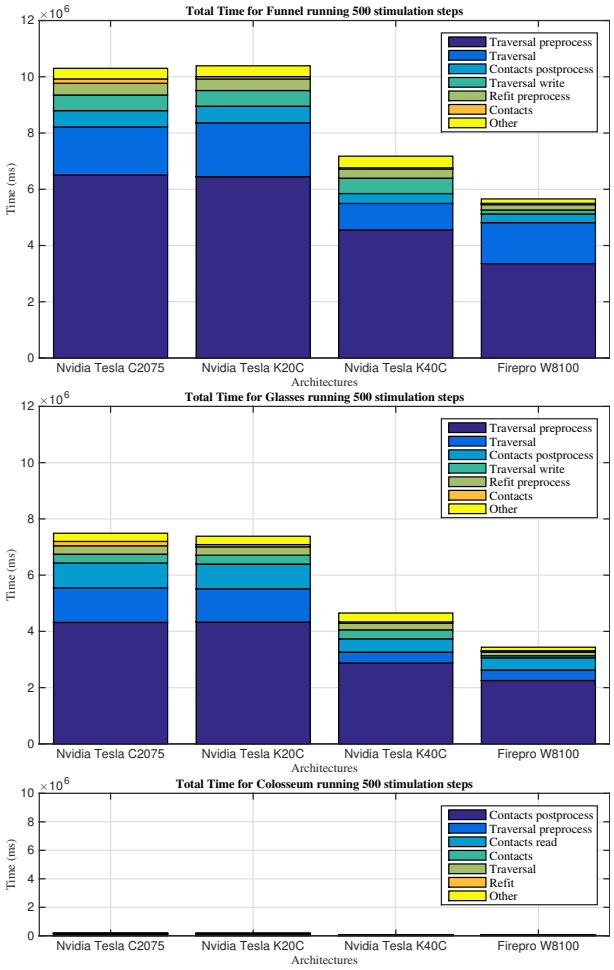


Fig. 10: Comparison of timings for different scenes and different architectures. Top: Funnel jamming, mid: Glasses, and bottom: Colosseum structure analysis. Bars show total accumulated time computed from the average of 10 simulations each consists of 5 seconds of simulated time. Only the 6 most expensive costs are shown to keep figures simple. The most expensive GPU computation is by far the BVH traversal. The traversal write is expensive on Tesla platforms but substantially lower on Firepro platform. We observe that host-size post- and pre-processing varies as is expected due to different host CPUs. However for GPU computations only Firepro W8100 and NVIDIA K40c appears to be similar in performance. K20c and C2075 appears to deliver similar performance. This is surprising as K20c have five times more CUDA cores and 3-4 times better peak performance. Interestingly the Firepro W8100 has much lower data transfer times than any of the NVIDIA test platforms we used (See also Table 6).

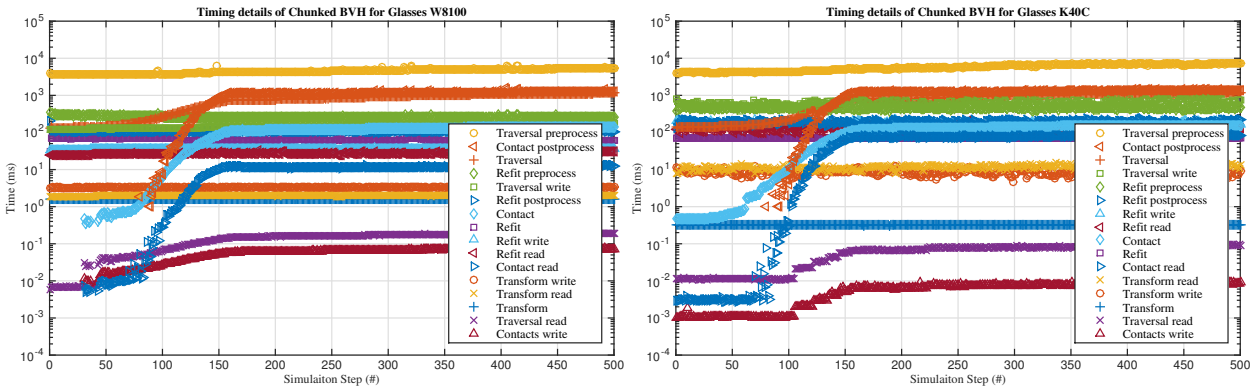


Fig. 11: Detailed comparison of timings for Funnel jam and falling Glasses (from top to bottom). Left column shows timings for Firepro W8100 and right column shows NVIDIA K40c. Average measurements from 10 simulation runs are shown. It is apparent that the two GPUs from two different vendors behave similar in terms of computations but the Firepro platform is better at handling data-transfers.

TABLE 6: Data transfer times for reading from device to host and writing from host to device. Observe that NVIDIA platforms all have similar timings and the AMD platform appears faster, particular for reading. In general when comparing to the total running time, data transfer times are not the worst bottlenecks, and they seem bounded by app. 5%-7% of the total running time. Compared to pre- and post-processing there appears to be little potential in optimizing data transfers further.

| Name | Arch. | Total time | Read | Write | Read | Write |
|---|---|---|---|---|---|---|
| | | (min) | | | (% of Total time) | |
| Glasses | C2075 | 213.32 | 1.73 | 6.54 | 0.81 | 3.07 |
| | K20c | 211.48 | 1.75 | 6.55 | 0.83 | 3.1 |
| | K40c | 127.7 | 1.65 | 6.67 | 1.29 | 5.23 |
| | W8100 | 99.22 | 0.32 | 1.54 | 0.32 | 1.55 |
| Colosseum | C2075 | 13.94 | 0.09 | 0.01 | 0.67 | 0.04 |
| | K20c | 13.83 | 0.1 | 0.01 | 0.7 | 0.04 |
| | K40c | 7.26 | 0.07 | – | 0.9 | 0.07 |
| | W8100 | 6.33 | 0.01 | 0.01 | 0.23 | 0.1 |
| Funnel | C2075 | 243.9 | 1.93 | 11.02 | 0.79 | 4.52 |
| | K20c | 245.6 | 1.91 | 11.08 | 0.78 | 4.51 |
| | K40c | 158.83 | 1.94 | 10.97 | 1.22 | 6.9 |
| | W8100 | 134.95 | 0.43 | 2.81 | 0.32 | 2.09 |

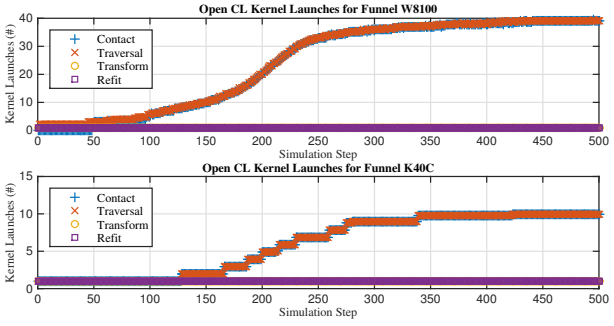


Fig. 12: Comparison of kernel launches for different architectures. Clearly the K40c uses far less launches than the other NVIDIA platforms. Interestingly, the Firepro W8100 that has similar compute power as the K40c but faster data transfer times, appears to use many more kernel launches than the NVIDIA platforms.

TABLE 7: Total times for each computational sub task running on each device. The timings are accumulated from average over 10 simulation runs. Only GPU computation tasks are shown, data transfers and pre- and post-processing are not shown here. Transforming the mesh data into proper world space position is insignificant, timings for refitting of $k$-DOP BVHs are quite small. Traversal timings appear to be in the order of 10 times larger than the contact point generation timings. Hence, the traversal process is the computational bottleneck on the device.

| Name | Arch. | Transform | Refit | Traversal | Contact |
|---|---|---|---|---|---|
| | | (min) | | | |
| Glasses | C2075 | 0.01 | 0.63 | 20.57 | 2.56 |
| | K20c | – | 0.77 | 19.77 | 1.34 |
| | K40c | – | 0.61 | 6.42 | 0.86 |
| | W8100 | 0.01 | 0.54 | 6.24 | 0.79 |
| Colosseum | C2075 | – | 0.08 | 0.09 | 0.09 |
| | K20c | – | 0.04 | 0.1 | 0.05 |
| | K40c | – | 0.03 | 0.04 | 0.04 |
| | W8100 | – | – | 0.02 | 0.05 |
| Funnel | C2075 | 0.01 | 0.97 | 28.48 | 2.61 |
| | K20c | – | 1.17 | 31.92 | 1.39 |
| | K40c | – | 0.94 | 15.64 | 0.86 |
| | W8100 | 0.02 | 0.49 | 24.26 | 0.82 |

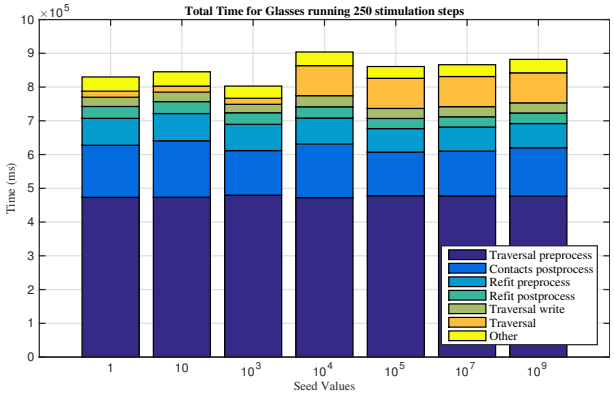


Fig. 13: Comparison of timings for different settings of the seed threshold value. Bars show total accumulated time computed from the average of 3 simulations.

tasks (see Figure 10). Note that no further distinction is given in [30], and our approach uses different phases as well as a contact force solver, which makes subsuming these numbers difficult.

Our tandem traversal currently is implemented to cover inter-object collisions only, however we believe it can be readily extended to cover self-collisions. Chunk size was set manually by human operator which may require some intelligent tuning. Automatic tuning of chunk-size by system introspection could make this more user friendly. Inspecting the scene to determine an automatic cutoff value for seeding the tandem traversal is likely to improve performance for general scenes as well.

Memory transfer times are an actual problem not to be dismissed, one possibility of addressing this is to move whole simulation onto the GPU. However, memory size is a concern for how large polygonal models and objects such a simulator would be capable of handling. We currently only use static splitting across a Z-curve on Morton codes. However, approaches from the field of ray tracing using the surface area heuristic (SAH), are promising performance gains for a future hybrid approach.

We have been concerned with the OpenCL data conversion costs that our implementation suffered from. These bottlenecks were so severe that the overall application performance of a GPU accelerated simulator using Chunked BVHs was comparable to a CPU version. When we accelerated the OpenCL data conversion using OpenMP, we could achieve a 20% speedup of the GPU-accelerated version compared to the CPU-version.

The consequence is that if one wants to benefit from a GPU acceleration in an existing CPU based simulator, then one must either cache and reuse OpenCL friendly data structures to avoid doing needless data conversions, reduce the bottleneck by parallelism, or redesign the simulator to use the OpenCL data structures everywhere. As we did not want to alter the internals of an existing and quite complex simulator, we opted for simple OpenMP acceleration of the data conversion, yielding a 20% overall application speedup factor of the GPU-acceleration compared to a CPU-only version, for the examples we tested.

## ACKNOWLEDGMENT

The authors would like to thank Stefan Sommer (UCPH), Sarah Niebe (UCPH), Jeff Trinkle (RPI) and Ming C. Lin (UNC) for helpful discussions and feedback.

The simulations were performed with resources provided by the North-German Supercomputing Alliance (HLRN).

## REFERENCES

[1] M. C. Lin and S. Gottschalk, "Collision detection between geometric models: A survey," in *In Proc. of IMA Conference on Mathematics of Surfaces*, 1998, pp. 37–56.

[2] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino, "Collision detection for deformable objects," *Computer Graphics Forum*, vol. 24, no. 1, pp. 61–81, 2005.

[3] P. M. Hubbard, "Approximating polyhedra with spheres for time-critical collision detection," *ACM Trans. Graph.*, vol. 15, no. 3, pp. 179–210, Jul. 1996.

[4] D. L. James and D. K. Pai, "Bd-tree: Output-sensitive collision detection for reduced deformable models," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 393–398, Aug. 2004.

[5] J. Allard, F. Faure, H. Courtecuisse, F. Falipou, C. Duriez, and P. G. Kry, "Volume contact constraints at arbitrary resolution," *ACM Trans. Graph.*, vol. 29, no. 4, pp. 82:1–82:10, Jul. 2010.

[6] O. Civit-Flores and A. Susín, "Fast contact determination for intersecting deformable solids," in *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games*, ser. MIG '15. New York, NY, USA: ACM, 2015, pp. 205–214.

[7] G. Baciu and W. S.-K. Wong, "Image-based collision detection," in *Integrated Image and Graphics Technologies*, D. D. Zhang, M. Kamel, and G. Baciu, Eds. Norwell, MA, USA: Kluwer Academic Publishers, 2004, pp. 75–94.

[8] S. Redon, A. Kheddar, and S. Coquillart, "Fast Continuous Collision Detection between Rigid Bodies," *Computer Graphics Forum*, vol. 21, no. 3, pp. 279–287, 2002.

[9] G. van den Bergen, "Efficient collision detection of complex deformable models using aabb trees," *J. Graph. Tools*, vol. 2, no. 4, pp. 1–13, Jan. 1998.

[10] J. Goldsmith and J. Salmon, "Automatic creation of object hierarchies for ray tracing," *IEEE Computer Graphics and Applications*, vol. 7, no. 5, pp. 14–20, May 1987.

[11] I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies," *ACM Transactions on Graphics (TOG)*, vol. 26, no. 1, p. 6, 2007.

[12] J. Pan, C. Lauterbach, and D. Manocha, "Efficient nearest-neighbor computation for GPU-based motion planning," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2010, pp. 2243–2248.

[13] M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets, and M. Gross, "Optimized spatial hashing for collision detection of deformable objects," in *Proceedings of the Vision, Modeling, and Visualization Conference 2003*, Nov 2003, pp. 47–54.

[14] B. Heidelberger, M. Teschner, R. Keiser, M. Müller, and M. H. Gross, "Consistent penetration depth estimation for deformable collision response," in *Vision, modeling and visualization 2004 : Proceedings, November 16-18, 2004, Stanford, USA*, B. Girod, M. A. Magnor, and H.-P. Seidel, Eds., Berlin, 2004, pp. 339–346.

[15] J. F. O'Brien and J. K. Hodgins, "Graphical modeling and animation of brittle fracture," in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. ACM Press/Addison-Wesley Publishing Co., 1999, pp. 137–146.

[16] M. Tang, D. Manocha, S.-E. Yoon, P. Du, J.-P. Heo, and R.-F. Tong, "VolCCD: Fast continuous collision culling between deforming volume meshes," *ACM Transactions on Graphics (TOG)*, vol. 30, no. 5, pp. 111:1–111:15, Oct 2011.

[17] E. Coumans, "Contact generation," Online slides from presentation at Game Developers Conference 2010, https://bullet.googlecode.com/files/GDC10_Coumans_Erwin_Contact.pdf.

[18] G. Hirota, S. Fisher, and M. Lin, "Simulation of non-penetrating elastic bodies using distance fields," University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, Tech. Rep., 2000.

[19] S. Fisher and M. C. Lin, "Deformed distance fields for simulation of non-penetrating flexible bodies," in *Proceedings of the Eurographic Workshop on Computer Animation and Simulation*. New York, NY, USA: Springer-Verlag New York, Inc., 2001, pp. 99–111.

[20] G. Hirota, S. Fisher, A. State, C. Lee, and H. Fuchs, "An implicit finite element method for elastic solids in contact," in *The Fourteenth Conference on Computer Animation, CA 2001, Seoul, South Korea, November 7-8, 2001*, 2001, pp. 136–254.

[21] E. Guendelman, R. Bridson, and R. Fedkiw, "Nonconvex rigid bodies with stacking," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 871–878, Jul. 2003.

[22] J. Spillmann and M. Teschner, "Contact surface computation for coarsely sampled deformable objects," in *Proc. Vision, Modeling, Visualization VMV'05*, Erlangen, Germany, November 2005, pp. 189–296.

[23] B. Smith, D. M. Kaufman, E. Vouga, R. Tamstorf, and E. Grinspun, "Reflections on simultaneous impact," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 106:1–106:12, Jul. 2012.

[24] S. Curtis, R. Tamstorf, and D. Manocha, "Fast collision detection for deformable models using representative-triangles," in *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, ser. I3D '08. New York, NY, USA: ACM, 2008, pp. 61–69.

[25] H. Wang, "Defending continuous collision detection against errors," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 122:1–122:10, Jul. 2014.
[26] M. Tang, R. Tong, Z. Wang, and D. Manocha, "Fast and exact continuous collision detection with bernstein sign classification," *ACM Trans. Graph.*, vol. 33, no. 6, pp. 186:1–186:8, Nov. 2014.
[27] L. He, R. Ortiz, A. Enquobahrie, and D. Manocha, "Interactive continuous collision detection for topology changing models using dynamic clustering," in *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, ser. i3D '15. New York, NY, USA: ACM, 2015, pp. 47–54.
[28] C. Lauterbach, Q. Mo, and D. Manocha, "gProximity: Hierarchical GPU-based operations for collision and distance queries," *Computer Graphics Forum*, vol. 29, no. 2, pp. 419–428, Jun 2010.
[29] J. Pan and D. Manocha, "GPU-based parallel collision detection for real-time motion planning," in *Algorithmic Foundations of Robotics IX*, 2011, vol. 68, pp. 211–228.
[30] M. Tang, D. Manocha, J. Lin, and R. Tong, "Collision-streams: Fast GPU-based collision detection for deformable models," in *Symposium on Interactive 3D Graphics and Games*. ACM, 2011, pp. 63–70.
[31] N. K. Govindaraju, S. Redon, M. C. Lin, and D. Manocha, "CULLIDE: nteractive collision detection between complex models in large environments using graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. Eurographics Association, 2003, pp. 25–32.
[32] N. K. Govindaraju, M. C. Lin, and D. Manocha, "Quick-CULLIDE: Fast inter-and intra-object collision culling using graphics hardware," in *Virtual Reality, 2005. Proceedings. VR 2005. IEEE*. IEEE, Mar 2005, pp. 59–66.
[33] D. Kim, J.-P. Heo, J. Huh, J. Kim, and S.-E. Yoon, "HPCCD: Hybrid parallel continuous collision detection using CPUs and GPUs," *Computer Graphics Forum (Pacific Graphics)*, vol. 28, no. 7, pp. 1791–1800, 2009.
[34] S. Pabst, A. Koch, and W. Straßer, "Fast and scalable CPU/GPU collision detection for rigid and deformable surfaces," *Computer Graphics Forum*, vol. 29, no. 5, pp. 1605–1612, Jul 2010.
[35] A. Sud, M. A. Otaduy, and D. Manocha, "Difi: Fast 3d distance field computation using graphics hardware," *Computer Graphics Forum*, vol. 23, no. 3, pp. 557–566, Sep 2004.
[36] A. Sud, N. Govindaraju, R. Gayle, and D. Manocha, "Interactive 3d distance field computation using linear factorization," in *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. ACM, 2006, pp. 117–124.
[37] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH construction on GPUs," *Computer Graphics Forum*, vol. 28, no. 2, pp. 375–384, Apr 2009.
[38] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst, "Embree: A kernel framework for efficient cpu ray tracing," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 143:1–143:8, Jul. 2014.
[39] O. Mattausch, J. Bittner, A. Jaspe, E. Gobbetti, M. Wimmer, and R. Pajarola, "Chc+rt: Coherent hierarchical culling for ray tracing," *Comput. Graph. Forum*, vol. 34, no. 2, pp. 537–548, May 2015.
[40] M. Tang, S. Curtis, S.-E. Yoon, and D. Manocha, "Iccd: Interactive continuous collision detection between deformable models using connectivity-based culling," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 4, pp. 544–557, Jul. 2009.
[41] J. Pantaleoni and D. Luebke, "Hlbvh: Hierarchical lbvh construction for real-time ray tracing of dynamic geometry," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010, pp. 87–95.
[42] P. Ganestam, R. Barringer, M. Doggett, and T. Akenine-Möller, "Bonsai: Rapid bounding volume hierarchy generation using mini trees," *Journal of Computer Graphics Techniques (JCGT)*, vol. 4, no. 3, pp. 23–42, 2015.
[43] C. Benthin, S. Woop, I. Wald, and A. T. Áfra, "Improved two-level bvhs using partial re-braiding," in *Proceedings of High Performance Graphics*, ser. HPG '17. New York, NY, USA: ACM, 2017, pp. 7:1–7:8.
[44] J. Hendrich, D. Meister, and J. Bittner, "Parallel bvh construction using progressive hierarchical refinement," *Comput. Graph. Forum*, vol. 36, no. 2, pp. 487–494, May 2017.
[45] J. Bender, K. Erleben, and J. Trinkle, "Interactive simulation of rigid body dynamics in computer graphics," *Computer Graphics Forum*, vol. 33, no. 1, pp. 246–270, 2014.
[46] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, Mar 2007.
[47] S. Gottschalk, "Collision queries using oriented bounding boxes," Ph.D. dissertation, The University of North Carolina, 2000.
[48] G. Zachmann, "Rapid collision detection by dynamically aligned DOP-trees," in *Proceedings of the Virtual Reality Annual International Symposium*. IEEE Computer Society, 1998, pp. 90–97.
[49] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha, "Fast proximity queries with swept sphere volumes," Technical Report TR99-018, Department of Computer Science, University of North Carolina, Tech. Rep., 1999.
[50] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser *et al.*, "Collision detection for deformable objects," *Computer Graphics Forum*, vol. 24, no. 1, pp. 61–81, Mar 2005.
[51] A. Munshi, "The OpenCL specification version: 1.1, revision 44," Khronos OpenCL Working Group, Tech. Rep., Jun 2011.
[52] NVIDIA, "NVIDIA's next generation CUDA compute architecture: Kepler GK110," NVIDIA Corporation, Tech. Rep., Jan 2013.
[53] K. Erleben, "Rigid body contact problems using proximal operators," in *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*, K. Yin and B. Thomaszewski, Eds. The Eurographics Association, 2017.

**Robert Schmidtke** is a PhD student at the Zuse Institute Berlin, Germany. In his master's thesis he developed an OpenCL implementation for collision detection using chunked bounding volume hierarchies. Currently he is investigating the fusion of Big Data technologies and High Performance Computing hardware, focusing on distributed data management using multi-level memory hierarchies, many-core processors and high-speed interconnects.

**Kenny Erleben** is an Associate Professor in Department of Computer Science, University of Copenhagen. He completed his PhD in 2005. His research interests are Computer simulation and numerical optimization with particular interests in computational contact mechanics of rigid and deformable objects, inverse kinematics for computer graphics and robotics, computational fluid dynamics, computational biomechanics, foam simulation, interface tracking meshing.