PSCC: Parallel Self-Collision Culling with Spatial Hashing on GPUs

Min Tang
Zhejiang University
Alibaba-Zhejiang University Joint Institute of Frontier Technologies
tang_m@zju.edu.cn

Zhongyuan Liu
Zhejiang University
lzy_work@foxmail.com

Ruofeng Tong
Zhejiang University
trf@zju.edu.cn

Dinesh Manocha
University of North Carolina at Chapel Hill
Zhejiang University
dm@cs.unc.edu

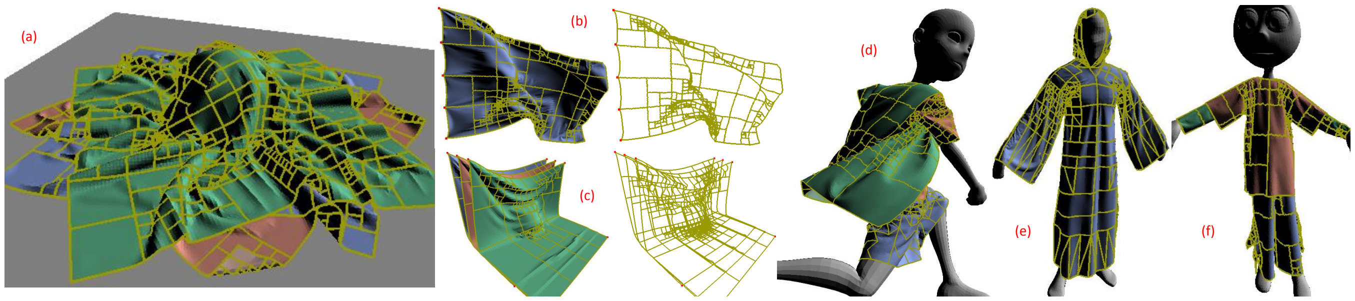


Figure 1: Our novel GPU-based collision detection algorithm is used to simulate clothes with regular (a-c) or irregular (d-f) shapes, which may correspond to a single layer (b, e) or multiple layers (a, c, d, f). We use a parallel spatial hashing algorithm to perform inter-object and intra-object collision checking on a GPU along with normal cone culling. The yellow line segments highlight all the areas that have no self-collisions and can be culled away. As compared to prior GPU-based collision detection algorithms, we observe $6-8X$ speedup.

ABSTRACT

We present a GPU-based self-collision culling method (P-SCC) based on a combination of normal cone culling and spatial hashing techniques. We first describe a normal cone test front (NCTF) based parallel algorithm that maps well to GPU architectures. We use sprouting and shrinking operators to maintain compact NCTFs. Moreover, we use the NCTF nodes to efficient build an enhanced spatial hashing for triangles meshes and use that for inter-object and intra-object collisions. Compared with conventional spatial hashing, our approach provides higher culling efficiency and reduces the cost of narrow phrase culling. As compared to prior GPU-based parallel collision detection algorithm, our approach demonstrates $6-8$X speedup. We also present an efficient approach for GPU-based cloth simulation based on PSCC. In practice, our GPU-based cloth simulation takes about one second per frame on complex scenes with tens or hundreds of thousands of triangles, and is about $4-6$X faster than prior GPU-based simulation algorithms.

KEYWORDS

self-collision culling, normal cone, spatial hashing, GPU, cloth simulation

ACM Reference Format:
Min Tang, Zhongyuan Liu, Ruofeng Tong, and Dinesh Manocha. 2018. PSCC: Parallel Self-Collision Culling with Spatial Hashing on GPUs. In *Proceedings of I3D*. ACM, New York, NY, USA, 11 pages. https://doi.org/XXXX

1 INTRODUCTION

Fast collision detection between deformable models is a key problem in physically-based simulation. In particular, reliable collision checking is important for many applications including cloth and surgical simulation. In many cases, collision detection is one of the major bottlenecks [Jiang et al. 2017; Selle et al. 2009; Sutherland et al. 2006; Tang et al. 2016].

Most algorithms for collision detection between deformable models are based on bounding volume hierarchies (BVHs) [Heo et al. 2010; Kim et al. 2009; Zhang and Kim 2014] or spatial hashing [Fan et al. 2011; Pabst et al. 2010; Weller et al. 2017]. As the models undergo deformation, these hierarchies/hashing tables are updated or reconstructed and used to cull away non-overlapping pairs. In order to accelerate the computations, many parallel algorithms that utilize multiple cores on a CPU or a GPU have been proposed [Fan et al. 2011; Heo et al. 2010; Kim et al. 2009; Pabst et al. 2010; Tang et al. 2010a; Weller et al. 2017; Zhang and Kim 2014]. However, there are two main challenges in terms of using these algorithms on complex models related to memory overhead and use of high-level culling methods.

Many BVH-based parallel algorithms maintain or update a front of the underlying BVHs to accelerate the computations [Klosowski et al. 1998; Li and Chen 1998; Tang et al. 2010b; Zhang and Kim 2014]. However, the storage overhead of these fronts can be high and can require several GBs of memory for complex models composed of tens or hundreds of triangles. Moreover, this storage overhead affects the performance on commodity or mobile GPUs, which typically have a few GBs of memory.

One of the key issues in terms is self-collision culling. Many high-level culling methods based on normal cones have been proposed to perform such culling [Provot 1997; Schvartzman et al. 2010; Tang et al. 2009; Wang et al. 2017]. However, they have been limited to sequential collision-detection methods. In fact, most parallel GPU-based collision detection and cloth simulation systems do not use self-collision culling based on normal cones [Tang et al. 2011, 2016; Weller et al. 2017]. As a result, the culling efficiency may not be high and results in a large number false-positive elementary tests.

Main Results: In this paper, we present a novel GPU-based collision detection algorithm (PSCC) to address these issues. This includes a parallel normal culling algorithm that uses a combination of shrinking and sprouting operators and has lower memory overhead (Section 3). The resulting hierarchy traversal performs more effective culling with lower runtime overhead. Moreover, we present an enhanced spatial-hashing method that is used for inter-object and intra-object collisions (Section 4). A key aspect of our approach is a novel workload scheme to distribute the computations evenly among different GPU threads. Based on our collision detection algorithm, we present an improved collision-handling scheme for cloth simulation that can reuse broad-phrase test results (Section 5).

We have implemented these algorithms on different GPUs and evaluated their performance on many complex benchmarks. Our novel combination of parallel normal cone culling with spatial hashing results in the following benefits:

- **Lower memory overhead:** For scenes with hundreds of thousands of triangles, the storage overhead reduces from several gigabytes [Tang et al. 2011] to about $500M$ bytes.
- **Faster collision detection:** As compared to prior GPU-based parallel collision detection algorithm [Tang et al. 2014; Weller et al. 2017], our approach demonstrates $6-8$X speedup.
- **Parallel cloth simulation:** In practice, our GPU-based cloth simulation takes about one second per frame on complex benchmarks with tens or hundreds of thousands of triangles, and is about $4-6$X faster than prior GPU-based algorithms [Tang et al. 2016].

2 RELATED WORK

There is considerable work on collision detection, cloth simulation, and parallel GPU-based algorithms to accelerate collision detection. In this section, we give a brief overview of prior work on collision culling and parallel algorithms.

Self-collision culling: Provot [1997] presented an efficient method using normal cones for self-collision culling. It is used to remove a large number of redundant tests at relatively "flat" areas of the deformable models. Conventionally, the normal cones associated with the internal nodes of a BVH are computed in a bottom-up manner. Self-collision detection can then be performed in a top-down manner. Schvartzman et al. [2010] combined this method with self-collision test trees (SCTT) to accelerate discrete self-collision queries for general deformable models with $O(n)$ complexity. Tang et al. [2009] extended normal cone culling to continuous collision detection. However, its complexity is $O(n^2)$, where n is the number of boundary edges. Wang et al. [2017] presented a conservative normal cone-based culling algorithm with $O(n)$ complexity. Other techniques include energy-based methods [Barbič and James 2010; Zheng and James 2012] and radial-based culling methods [Wong and Cheng 2014; Wong et al. 2013].

Spatial hashing on GPU: Spatial hashing is a classic algorithm for collision detection, and can be easily parallelized on GPUs [Lefebvre and Hoppe 2006]. A spatial hashing algorithm has a constant time complexity on querying all the triangles in proximity. Pabst et al. [2010] used a uniform grid to perform the broad phase culling on GPUs for triangle meshes. In order to handle the non-uniform distribution of scene geometry primitives, Faure et al. [2012] extended uniform grids to two-layer grids. Recently, hierarchical grids have been used to further improve the efficiency of spatial subdivision-based CD algorithms [Faure et al. 2012; Weller et al. 2017; Wong et al. 2014]. Although hierarchical grids can address the non-uniform distribution of scene primitives, they cannot cull the triangles that belong to "relatively flat" areas of cloth or deformable models. The previous spatial hashing algorithms [Faure et al. 2012; Weller et al. 2017; Wong et al. 2014] tend to treat soft bodies and rigid bodies uniformly. This can result in a high number of redundant self-collision tests, i.e., false positives.

Parallel cloth simulation on multi-core/many-core processors: Selle et al. [2009] designed a parallel cloth simulation algorithm for multi-core platforms. That algorithm can take up to 30 minutes per frame for a cloth mesh with $500K$ triangles on a 16-core workstation. Tang et al. [2013] proposed a streaming algorithm for regular-shaped high-resolution cloth simulation on GPUs. The algorithm has been extended for cloth simulation with arbitrary topology structures [Tang et al. 2016], and can take up to 35 seconds per frame for cloths with $1M$ triangles on a commodity GPU. These parallel algorithm maintain a bounding volume traversal tree (BVTT) front for parallel collision checking, which can have a large memory overhead. Recently, Jiang et al. [2017] used a Lagrangian approach to simulate the anisotropic elastoplasticity of cloth, and their implementation takes about 2 minutes per frame for a cloth mesh with $1.8M$ triangles.

3 PARALLEL NORMAL CONE CULLING

In this section, we present our parallel normal cone culling algorithm. We first introduce the notation used in the rest of the paper.

3.1 Notation

We mainly focus on accurate collision detection for general deformable objects, including self-collisions. We mainly focus on continuous collision detection (CCD), while parallel scheme can also be used for discrete collision detection (DCD). Our CCD algorithm assumes linearly interpolating trajectories [Bridson et al. 2002; Provot 1997; Tang et al. 2009] between the mesh vertices corresponding to two successive simulation instances. We use the following acronyms in the rest of the paper: *BV* (bounding volume), *BVH* (bounding volume hierarchy), *NCTT* (normal cone test tree), and *NCTF* (normal cone test front).

3.2 Parallel Culling

Our parallel normal culling algorithm is based on BVH-based culling with BVTT (Bounding volume testing tree) fronts [Tang et al. 2011]. The BVH quality is critical for the culling efficiency of normal cone tests. For cloth simulation, we construct BVH in its material space [Wang et al. 2011] (Figure 4(a)) instead of the geometry space (Figure 4(b)). As a result, we observe much higher culling efficiency for normal cone tests, especially for areas with multiple layers of cloth (see the upper part of the body in Figure 4). Conceptually, any normal cone testing algorithm [Provot 1995; Schvartzman et al. 2010; Tang et al. 2009; Volino and Thalmann 1994; Wang et al. 2017] can be used in Algorithm 1. During the construction step, we also compute a normal cone and gather its contour edges for each BVH node. This is useful for the cases where all the underlying triangles are connected with each other. The normal cone is computed by merging all the normal vectors of these triangles.

In this work, we use a recent algorithm by Wang et al. [Wang et al. 2017] based on unprojected contour test with $O(n)$ complexity, due to its support for both CCD and DCD, and its simplicity for implementation.

We use a NCTF to record all the BVH nodes where normal cone tests are terminated during the tree traversal at the last time step (as shown in Figure 2). By utilizing the spatial-temporal coherence between simulation time steps, we perform various normal cone tests in parallel starting from all the nodes recorded in the front. As shown in Algorithm 1, all the nodes of the NCTF can be tested and updated independently. For those nodes that do not pass the normal cone tests, sprouting operators are used to traverse down and add their descendents into the NCTF. In this manner, high-level self-collision culling is performed in parallel on a GPU.

Sprouting operator: In Algorithm 1, we first perform the normal cone test on each node in the NCTF (lines 2-3). For those nodes that fail the test, we use the sprouting operator (Algorithm 2) to traverse to their descendents (line 4-5).

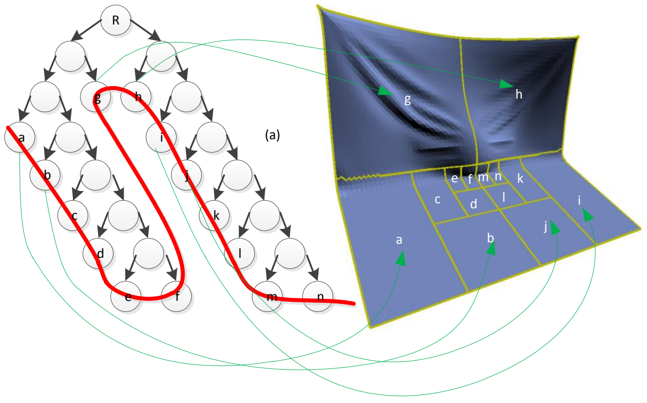


Figure 2: Normal Cone Test Front (NCTF): For the cloth model in (b), we store all the BVH nodes where normal cone tests are terminated during the tree traversal at the last time step (e.g. the red curve in (a)) as the normal cone test front. The green arrows highlight the correspondences between culled areas and BVH nodes. Here, $a, b, ..., n$ are the NCTF node IDs. All its descendent triangles share the same NCTF ID.

Algorithm 1 SelfCollideParallel(F): Perform self-collision in parallel with a NCTF F.

```
1: for all $n^i \in F$ do
2:     if IsLeaf($n^i$) or NormalConeTest($n^i$) == true then
3:         return  // Skip nodes without self-collisions.
4:     end if
5:     Delete($F$, $n^i$)
6:     Sprouting($F$, $n^i$ →left)
7:     Sprouting($F$, $n^i$ →right)
8: end for
```

Algorithm 2 Sprouting(F, n^i): Update the normal cone test front F by sprouting from a BVH node n^i.

```
1: if IsLeaf($n^i$) or NormalConeTest($n^i$) == true then
2:     Insert($F$, $n^i$)
3: else
4:     Sprouting($F$, $f^i$ →left)
5:     Sprouting($F$, $f^i$ →right)
6: end if
```

Ultimately the nodes that pass the normal cone tests are added to the NCTF (line 2).

Shrinking operator: In Algorithm 1, we only use the sprouting operator to update the front. In practice, the underlying deformable mesh or the cloth may change from a wrinkling status to a flat status. In order maintain a compact NCTF, we use a shrinking operator to re-scan the front nodes after several time steps. During the simulation, some tangled areas may become flat again. So the front needs to shrink upward If two sibling front nodes are in the front, we remove

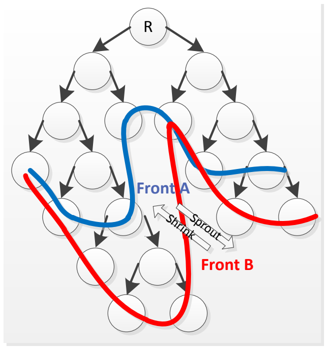


Figure 3: Front Updating: The NCTF remains compact by updating with sprouting and shrinking operators. During different time steps, the regions that can be culled with normal cone tests change. Therefore, the front needs to be updated from Front A to Front B. Both A and B are compact. Sprouting and shrinking operators are used to maintain the compact fronts (i.e. A, B).

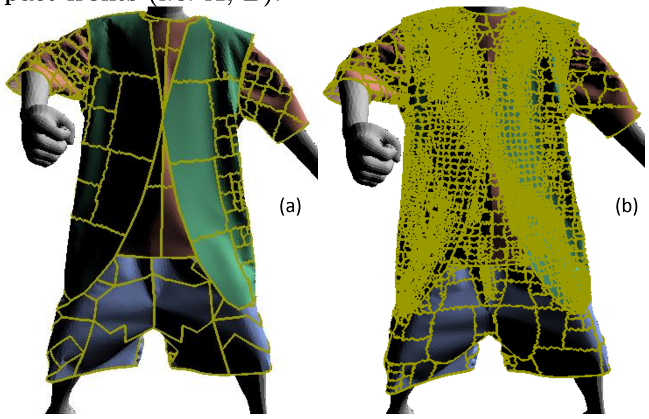


Figure 4: The tightness of the BVH is important for NC culling. Here we highlight the culling efficiency with different BVH construction algorithms. Compared with the BVH constructed in geometry space (b), the BVH constructed in material space (a) results in better normal cone culling efficiency, especially for multi-layer clothes (see the upper part of the body).

all these sibling nodes and insert their parent into the front. In this way, the front will shrink upward in various areas which become flat again. As shown in Figure 3, the front can be fully updated in parallel with sprouting (Algorithm 2) and shrinking operators.

Memory overhead: The number of nodes in NCTF is bounded by $2 * N + 1$, where N is the number of triangles. This $O(N)$ memory requirement is much less than the memory overhead of the BVTT front [Tang et al. 2011], which can be $O(N^2)$. As a result, we observe reduced memory usage and better caching behavior on complex benchmarks.

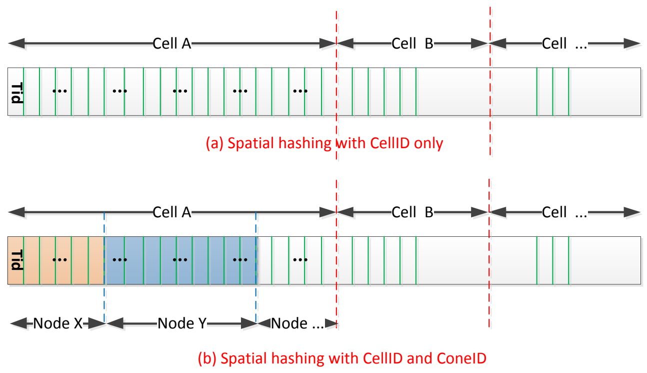

Cell A
Cell B
Cell ...
Tid
(a) Spatial hashing with CellID only
Cell A
Cell B
Cell ...
Tid
Node X
Node Y
Node ...
(b) Spatial hashing with CellID and ConeID

Figure 5: Hash Key: Both spatial information and NCTF node information are used as the hash keys. We show the the conventional hashing key (a). Our novel hashing key (b) splits the triangles in the same cell into groups with the same NCTF node ID, to enable self-collision culling.

Self-collision culling efficiency: NCTF is designed to perform parallel normal cone culling, while BVTT front is used for parallel collision detection between BVH nodes. For self-collision detection, NCTF is more efficient than BVTT due to its ability to remove large "flat" areas. On the other hand, the BVTT needs to be traversed to leaf level, and the resulting algorithm needs to check all the triangle pairs with overlapping bounding volumes for exact collisions. Most of these overlap tests tend to be false positives, as the adjacent triangle pairs cannot be cull out with bounding volume tests. As highlighted in Figure 1, all these areas bounded by the yellow line segments are skipped for self-collision checking with NCTF. If we use BVTT, all these triangle pairs have to be checked. Therefore, NCTF tends to be more efficient than BVTT for self-collision detection.

4 SPATIAL HASHING WITH NORMAL CONE FRONT

The high-level culling algorithm for self-collisions needs to be combined with low-level collision culling methods to reduce the number of false positive elementary tests. We combine parallel normal culling with extended spatial hashing for improved performance. A key aspect of this integrated scheme is the use of paired hashing key. We split the bounding volume of the entire simulation scene into cells and assign each cell a unique Cell ID. For each triangle, we keep track of the corresponding Cell IDs for those cells that overlap with its BV. Each triangle also has a unique NCTF node ID, which is computed from the parallel normal cone tests (e.g., $a, b, c, ..., n$ in the Fig. 2). We use both, spatial information (Cell IDs) and NCTF node IDs, as the hashing keys. Logically, the triangles of a mesh can be organized into a workload distribute table, as shown in Figure 7. All the triangles with the same $\{CellID, NodeID\}$ are skipped in terms of performing

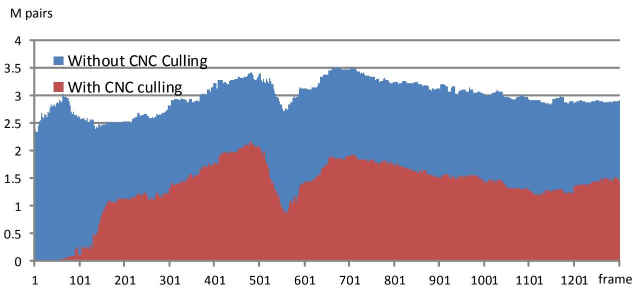


Figure 6: Comparison between the number of output triangle pairs from broad phase culling: The red and blue areas represent the number of triangle pairs output by the broad phase with and without CNC culling, respectively. With fewer output triangle pairs, the exact overlap tests performed during the narrow using spatial hashing are reduced.

intersecting tests, because they are in same flat region and do not need to be tested for intra-object collisions.

With conventional spatial hashing, only $\{CellID\}$ are used as the hashing keys. The intersecting number in one spatial cell, N_{inter}, is:

$$N_{inter} = n_t * (n_t - 1)/2,$$

where n_t is the number of triangles in that cell. With the new hashing key $\{CellID, NodeID\}$, the new number, N'_{inters}, becomes:

$$N'_{inter} = \sum_{i \neq j} n_i * n_j,$$

where n_i and n_j are the numbers of triangles with the same Node IDs in the cell, and $\sum n_i = n_t$. As long as $n_i >= 1$, we have:

$$\begin{aligned} N_{inter} - N'inter &= n_t * (n_t - 1)/2 - \sum_{i \neq j} n_i * n_j \\ &= [(\sum n_i)^2 - \sum n_i - 2 * \sum_{i \neq j} n_i * n_j]/2 \\ &= (\sum n_i^2 - \sum n_i)/2 >= 0 \end{aligned}$$

Based on the paired hashing key, we do not need to perform self-collision detection between the small group of triangles that have the same NodeID. Due to our formulation, the number of candidate triangle pairs is reduced.

Self-culling with spatial hashing decreases the number of threads, as each test is performed by a separate thread and PSCC results in fewer triangle-triangle tests (see (Figure 6). This is useful in the narrow phase testing of the collision detection pipeline. We use two hash tables to build our workload distribute table, and modify the workload distribute algorithm [Fan et al. 2011] for our approach to balance the computational load of GPU threads. Finally, a token position technique [Fan et al. 2011] is used to avoid an output of duplicate triangle pairs after the broad phase.

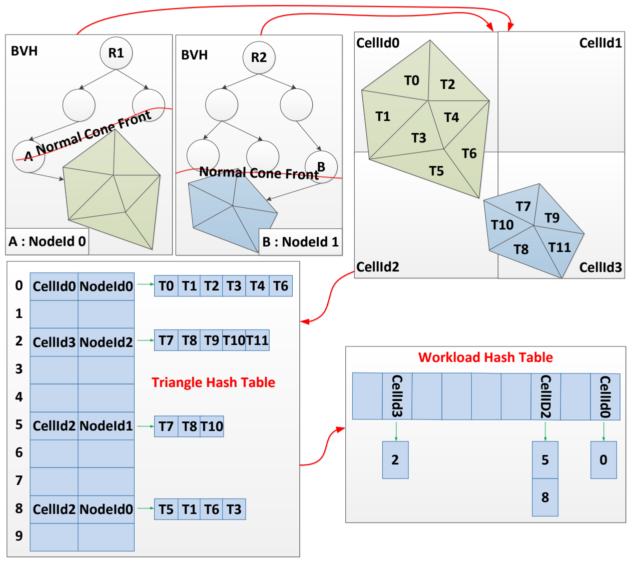


Figure 7: Build workLoadHashTable: Two hash tables are computed for building the workLoad-HashTable. As the figure show, prior spatial hashing algorithms [Pabst et al. 2010] would need to perform higher number of tests, which is 43 tests (10 for CellId0, 21 for CellId3, and 12 for CellId2) for this benchmark. Based on the integration of spatial hashing with normal cone culling (PSCC), we only perform 12 tests (for CellId2) in this scenario.

4.1 Workload Distribution

Corresponding to the paired hash keys, we use a novel workload distribution to distribute the computations evenly among GPU threads. Before workload distribution, we need two pieces of information. For each spatial cell, we need to collect all the triangles that overlap with that cell. We separate all the triangles that lie into the same cell into different sets by NodeID. As shown in Algorithm 3, two hash tables are used to implement this approach. In our algorithm, for all triangles that fall into the same cell and have the same NodeID, no extra intersection testing is needed. Therefore, *triangleHashTable* is used to separate the triangles with different CellIDs and NodeIDs into many different groups. Next, *workLoadHashTable* collects all the groups indexed with the same CellID. Finally, through these two hash tables, we can copmute all the triangle pairs that are needed to perform the BV overlap tests (Figure 7). In this way, we can perform workload distribution among all the GPU threads.

We represent our spatial-hashing table as a row-major sparse matrix. We can insert a {key, value} pair by hashing its key to find the corresponding row index, and put the value to a value array on this row. The hash table construction algorithm is a two stage process very similar to the sparse matrix assembly method used in [Tang et al. 2016].

Algorithm 3 SpatialHashingCD(T,$NodeIds$): Perform spatial hashing collision detect on a triangle set T with a NCTF node ID $NodeIds$.

```
1:  for all $t^i \in T$ do
2:      for all $cell^j \in cellsOverlapped^i$ do
3:          // Make the unique id $uuid$ with nodeid and cellid for
            every cell the triangle $t^i$ overlapped.
4:          $uuid$ = MakeUuid($nodeid^i$, $cellid^j$)
5:          // Use $uuid$ as the key to insert triangle index $tIndex^i$
            into the $triangleHashTable$.
6:          Insert($triangleHashTable$, $uuid$, $tIndex^i$)
7:      end for
8:  end for
9:  // For every row $row^i$ in triangleHashTable, use the $cellid^i$
    of this row as the key to insert the row index $i$ into workLoad-
    HashTable.
10: for all $row^i \in triangleHashTable$ do
11:     Insert($workLoadHashTable$, $cellid^i$, $i$)
12: end for
13: // Count the total amount of BV tests.
14: CountTotalThreadNum($workLoadHashTable$,
    $triangleHashTable$)
15: // For every BV test, a thread is used to balance the compu-
    tation load.
16: for all $threadidx^i \in totalThread$ do
17:     DecodePairsAndTest($threadidx^i$,$workLoadHashTable$,
        $triangleHashTable$, $T$)
18: end for
```

4.2 Hashing Function

We use NCTF node IDs and cell IDs to make a uuid as the hash key. The $\{Key\}$ is calculated by:

$$\begin{aligned} Key &= uuid \bmod hashTableSize \\ &= (CellId + NodeId * CellIdMax) \bmod hashTableSize \end{aligned}$$

and

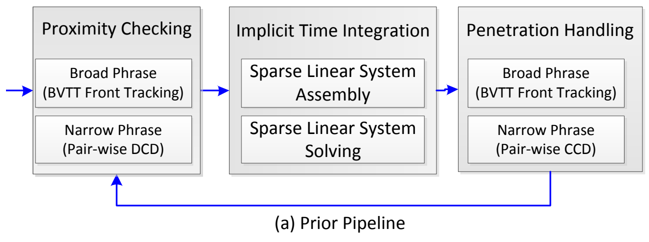
$$CellId = Id_x * N_y * N_z + Id_y * N_z + Id_z,$$

where $N_{x,y,z}$ and $Id_{x,y,z}$ are the number of cells and the ID of a current cell in the x,y,z axis of the spatial grid, respectively. The CellId can be any other hashing function such as Morton codes [Morton 1966] or DJB2 [Eitz and Gu 2007].

Grid size: We currently use a uniform grid for space partition. Therefore, the grid size is an important factor in the overall performance. For scenes with unevenly sized triangles, a grid size that is too large may decrease the culling efficiency of the spatial-hashing method, while a grid size that is too small would lead to many redundant detections between big triangle pairs. We therefore choose the gird size as:

$$Size_{cell} = k * BoxAxisLen_{aver},$$

where k is a heuristic parameters and $BoxAxisLen_{aver}$ is the average length of all triangles' bounding boxes' axes. In practice, this grid size works well for most of our benchmarks. The running time of the broad-phrase culling is about 6% – 10% of the overall running time of cloth simulation pipeline described in Section 5.


Proximity Checking
Broad Phrase (BVTT Front Tracking)
Narrow Phrase (Pair-wise DCD)
Implicit Time Integration
Sparse Linear System Assembly
Sparse Linear System Solving
Penetration Handling
Broad Phrase (BVTT Front Tracking)
Narrow Phrase (Pair-wise CCD)
(a) Prior Pipeline

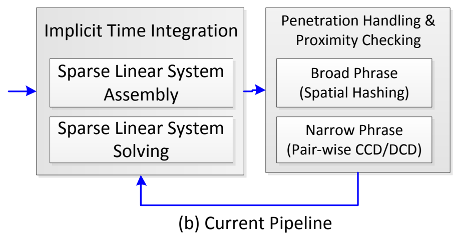

Implicit Time Integration
Sparse Linear System Assembly
Sparse Linear System Solving
Penetration Handling & Proximity Checking
Broad Phrase (Spatial Hashing)
Narrow Phrase (Pair-wise CCD/DCD)
(b) Current Pipeline

Figure 8: Collision Handling Pipeline: Compared to the prior pipeline (a) in [Tang et al. 2016], the current pipeline (b) reuses the testing results from the broad phrase for both pair-wise DCD and CCD.

Extension to rigid bodies: Our spatial hashing algorithm can easily be extended for collision detection between rigid bodies by simply assigning a unique NodeID for each rigid body. In this manner, all self-collision tests among the triangles belonging to the same rigid body can be avoided.

4.3 Overall PSCC Algorithm

Our collision detection algorithm is a combination of Algorithm 1 (high level culling) and Algorithm 3 (low level culling). We first execute Algorithm 1 to compute the NCTF node IDs, then execute Algorithm 3 for inter-object and intra-object collision culling. The paired hash keys in Algorithm 3 are based on the NCTF node IDs computed by Algorithm 1.

5 COLLISION HANDLING FOR CLOTH SIMULATION

We use our novel parallel collision detection algorithm to improve the performance of GPU-based cloth simulation. The original pipeline for a fast GPU-based algorithms is shown in Figure 8(a), which first performs proximity checking with DCD, collects all the proximity constraints for implicit time integration, performs penetration handling with CCD to ensure there are no penetrations between triangle pairs. In [Tang et al. 2016], BVTT front tracking is used as broad phrase culling, and is performed twice for DCD and CCD.

We notice that the vertices used for CCD in the current time step are the same as those for DCD at the next time step. This is due to the fact that the penetration-free mesh output from last time step is directly used as the input mesh for the next simulation time step, i.e. they have the same vertices. Therefore, the results of broad-phrase culling for last time step can be reused for next time step (as shown in Fig. 8(b)). We therefore perform spatial hashing culling

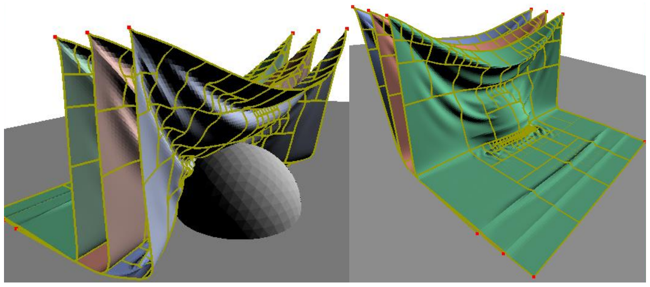


Figure 9: Benefits of the new collision handling pipeline: For this frame of Benchmark Sphere, with the new collision handling pipeline, we are able to reduce the number of BV tests and the running time of broad phrase culling to 51.1% and 53.3%, respectively.

Resolution (triangles)	Bench-marks	Time Steps(s)	CAMA (K40c)	Our (K40c)	Our (1080)	Our (1080 TI)
200k	Sphere	1/200	2.84	1.62	0.94	0.82
200k	Twisting	1/200	2.92	1.72	0.97	0.87
80K	Flag	1/100	3.39	0.81	0.35	0.20
127K	Andy	1/25	2.42	1.48	0.84	0.57
172K	Falling	1/30	2.49	0.98	0.51	0.39
124K	Bishop	1/30	3.19	1.82	0.94	0.82

Figure 10: Performance: This figure shows the average running time for a single frame of our algorithm on four different configurations (i.e., CAMA on Telsa K40C, our system on Tesla K40C, our system on GTX 1080, and our system on GTX 1080 Ti, respectively). We observe significant speedups (up to $16X$) over CAMA.

during the broad phrase culling, and execute it only once per time step. The culling results are used for the narrow phrase (both for DCD and CCD). During the narrow phrase with CCD, we detect all the penetrations between Vertex-Face/Edge-Edge pairs. The CCD computation is performed until all penetrations are handled. After CCD computation, another pass of narrow phrase testing with DCD is performed to collect proximity constraints for the next time step. In this way, we save the running time in terms of performing another pass during broad phrase culling by reusing its output.

In the Benchmark Sphere (Figure 9), we are able to reduce the number of BV tests and the running time of broad phrase culling by 51.1% and 53.3%, respectively.

6 IMPLEMENTATION AND RESULTS

Implementation: We have implemented our PSCC algorithm on three commodity GPUs, an NVIDIA Tesla K40c (with 2880 cores at 745MHz and 12G memory), an NVIDIA GeForce GTX 1080 (with 2560 cores at 1.6 GHz and 8G memory), and an NVIDIA GeForce GTX 1080 Ti (with 3584 cores at 1.58 GHz and 11G memory), respectively. We use these GPUs with varying number of cores to test the parallel performance of our approach. Our implementation is based on CUDA toolkit 8.0 and Visual Studio 2013 as the underyling development environment. We used a standard PC (Windows 7 Ultimate 64 bits/Intel I7 CPU@3.5G Hz/8G RAM) as the testing environment and perform single-precision floating-point arithmetic for all the computations performed on a GPU. Moreover, we used Thrust for the prefix-sum operator and cuBLAS/cuSPARSE for linear system solving.

Benchmarks: We used three different benchmarks for regular-shaped cloth simulation:

- **Twisting:** Three pieces of cloth with a total of $200K$ triangles twist severely as the underlying ball rotates (Fig. 1(a)).
- **Flag:** A waving flag with $80K$ triangles is blowing in the wind (Fig. 1(b)).
- **Sphere:** Three pieces of hanging cloth with a total of $200K$ triangles are hit by a forward/backward moving sphere (Fig. 1(c)).

These benchmarks contain many inter- and intra-object collisions. We used three other benchmarks for garment simulation:

- **Andy:** A boy wearing three pieces of clothing (with $127K$ triangles) is practicing Kung-Fu (Fig. 1(d)).
- **Falling:** A man wearing a robe (with $172K$ triangles) falls down rapidly under strikes (Fig. 1(e)).
- **Bishop:** A swing dancer wears three pieces of clothing (with $124K$ triangles) (Fig. 1(f)).

These are complex benchmarks with multiple pieces, layers, and wrinkles, which result in a high number of collisions. Our algorithm can handle inter- and intra-object collisions reliably (see video).

Performance: Figure 10 shows the resolutions and time steps for different benchmarks, and highlights the performance of our algorithm on these benchmarks. This includes the average frame time of our GPU-based algorithm on three commodity GPUs with different numbers of cores. These results demonstrate that our cloth simulation algorithm works well on different GPU architectures and that the performance is proportional to the number of cores. Compared with the CAMA performance [Tang et al. 2016] on an NVIDIA Telsa K40C, we observe significant speedups (up to $16X$) on an NVIDIA GeForce GTX 1080 Ti.

Running time ratios: Figure 11 shows the running time ratios of different computing stages: time integration, broad phrase testing, narrow phrase testing, and penetration handling. These data are collected by running our system for Benchmark Sphere on the NVIDIA GeForce GTX 1080. As shown in the figure, time integration takes almost constant running time for all the time steps. Collision detection (broad phrase and narrow phrase) and penetration handling appear to be the most computationally expensive parts, especially when the cloths are tangled.

Memory overhead: Figure 12 compares the memory overhead of our algorithm (spatial hashing-based) and CAMA (BVH-based). We list all the memory breakdown for the Benchmark Sphere running using two different algorithms.

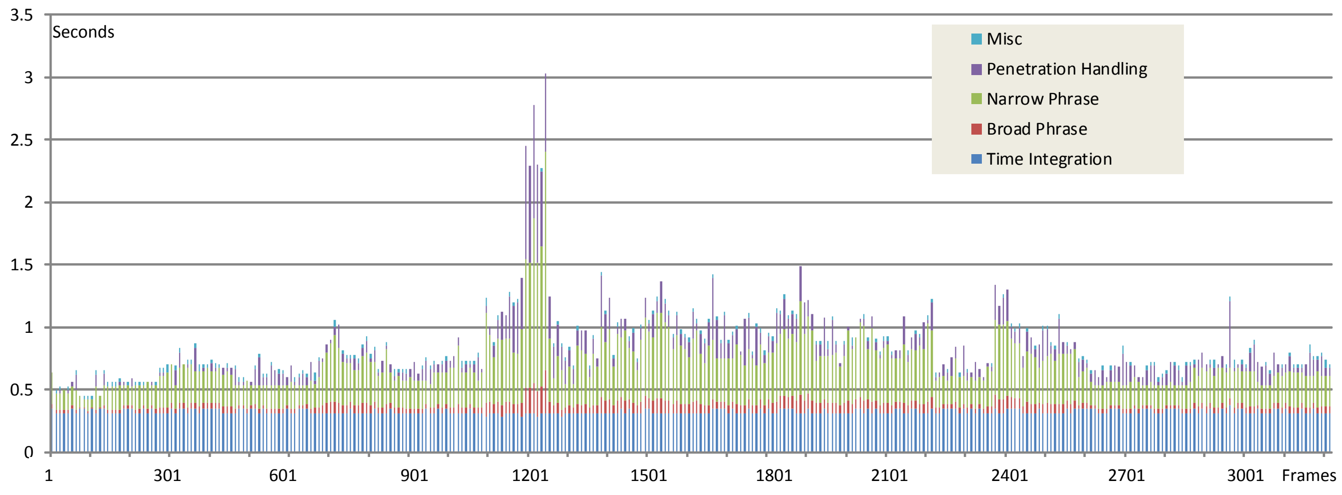


Figure 11: Ratios of different stages of our system: This figure shows the running time ratios of different computing stages: time integration, broad phrase testing, narrow phrase testing, and penetration handling, respectively. These data are collected by running our system for the Benchmark Andy on the NVIDIA GeForce GTX 1080.

Compared with our approach, CAMA [Tang et al. 2016] uses much more memory as the storage for BVTT front and triangle pairs (passed broad phrase culling). Our approach requires much less memory for spatial hashing and normal cone front. Due to the culling effects of normal cones, our approach also needs much less memory for the triangles pairs. CAMA [Tang et al. 2011, 2016] needs more than $2G$ GPU memory for this benchmark, while our system needs approximately $500M$ GPU memory. We also observe similar memory reduction in other benchmarks using PSCC.

7 COMPARISON AND ANALYSIS

In this section, we compare the features and performance of our approach with prior methods.

GPU-based collision detection algorithm and cloth simulation system: As compared with priort GPU-based cloth simulation systems [Tang et al. 2013, 2016], the main benefits of our algorithm include much less memory overhead (Figure 12) and improved runtime performance on the same GPU (Figure 10). With the enhanced spatial hashing data structure, our approach is capable of performing large-area self-collision culling. Figure 16 shows the running time ratios of different computing stages of the CAMA system [Tang et al. 2016]. This performance data is collected by running CAMA for Benchmark Sphere on the NVIDIA GeForce GTX 1080. As shown in the figure, CAMA needs much more running time than our system for collision detection and handling. All the speedup are due to the improvements in the collision detection algorithm (broad phrase testing, narrow phrase testing, and revised pipeline), reduced memory overhead and the improved cloth simulation pipeline. On the same GPU platform (NVIDIA GeForce GTX 1080), our collision

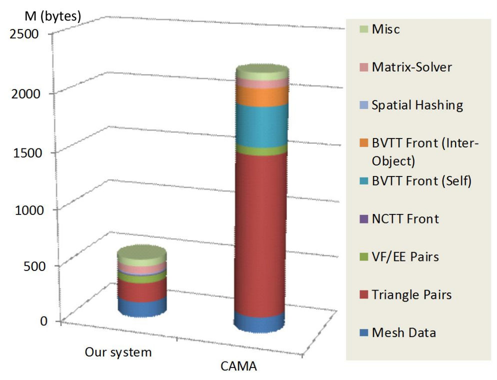


Figure 12: Memory Overhead: This figure compares the memory occupation ratios of our system (spatial hashing based) and CAMA (BVH-based). We list all the memory occupation details for the Benchmark Sphere running on the two different systems. Compared with our system, CAMA used much more memory as the storage for BVTT front and triangle pairs (that passed broad phrase culling). CAMA [Tang et al. 2011, 2016] needs more than $2G$ GPU memory for this benchmark, while our system needs approximately $500M$ GPU memory.

detection algorithm provides $6-8$X speedup over prior GPU-based algorithms [Tang et al. 2011] , and results in $4-6$X speedup over prior cloth simulation algorithm [Tang et al. 2016].

CPU-based collision culling algorithms: The recent work of Wang et al. [2017] combines normal cone culling

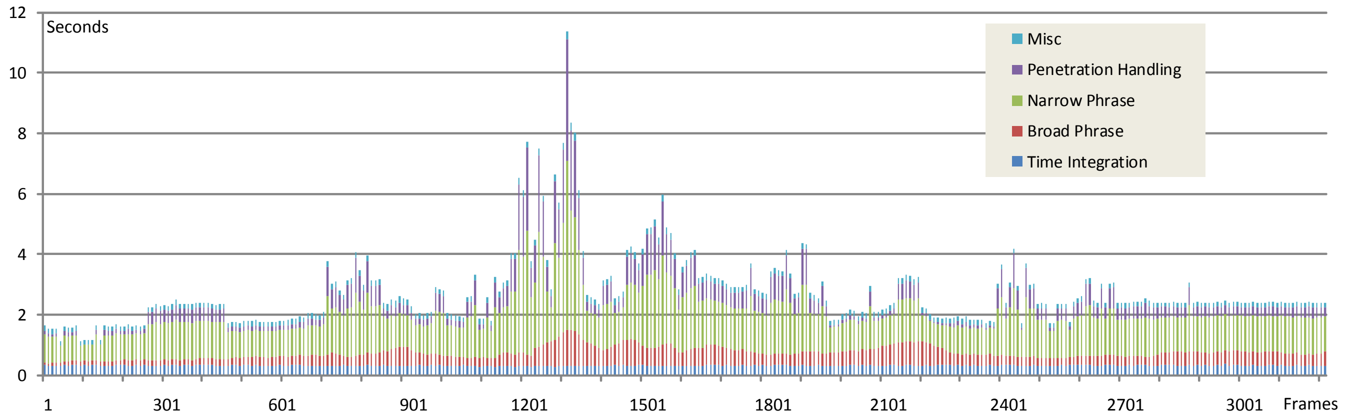


Figure 16: Ratios of different stages of the CAMA system: This figure shows the ratios of different computing stages: time integration, broad phrase testing, narrow phrase testing, etc. These data are collected by running the CAMA system for the Benchmark Andy on the NVIDIA GeForce GTX 1080.

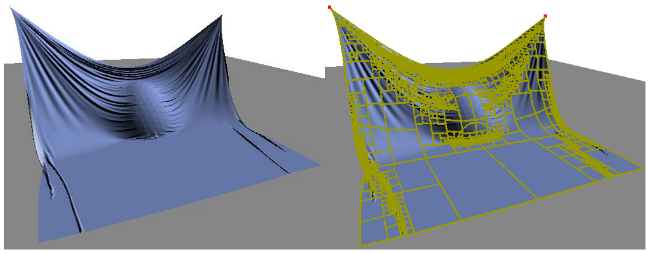


Figure 13: High-resolution cloth simulation: A rectangular cloth (with $1M$ triangles) falls on the top of a moving sphere.

Resolution (triangles)	Bench-marks	Time Steps(s)	CAMA (K40c)	Our (K40c)	Our (1080)	Our (1080 Ti)
1M	Sphere	1/200	35.75	7.60	4.20	3.28

Figure 14: Performance for high-resolution cloth simulation: This figure shows the average running time for a single frame of our algorithm on a benchmark with $1M$ triangles. We observe 8.5X speedup over CAMA on the GeForce GTX 1080.

Resolution (triangles)	Bench-marks	Hierarchical Grid(KDet)			Ours		
		BV-test	Inter-pairs	Self-pairs	BV-test	Inter-pairs	Self-pairs
127K	Andy	38584 k	389 k	3831 k	21681 k	389 k	1931 k
124K	Bishop	30552 k	553 k	2648 k	18774 k	553 k	1514 k

Figure 15: Comparison between KDet and our method: This figure shows the average amount of BV-tests and output triangle pairs of the broad phase in different methods and benchmarks.

with BVTT front tracking. However, it is a serial algorithm designed for single thread implementation, it is hard to parallelize and compare the performance with a GPU-based parallel approach. Another drawback of this algorithm is that it needs a lot of CPU memory to store the BVTT front.

Spatial Hashing Only: As a comparison, we also implemented the hierarchical spatial hashing algorithm KDet [Weller et al. 2017]. Figure 15 shows the total amount of BV-tests and output triangle pairs of the broad phase. The BV-tests' numbers indicate the spatial hashing method's efficiency and the triangle pairs' output indicates the workload of the narrow phase. Our approach achieves better performance at both the broad phase and narrow phase.

Performance on high-resolution cloth simulation: Figure 14 shows the average running time for a single frame of our algorithm on a benchmark with $1M$ triangles (Figure 13). We observe 8.5X speedups (on a GTX 1080) over CAMA (on Tesla K40c). The recent work of Jiang et al. [Jiang et al. 2017] takes about 2 minutes per frame for a similar benchmark with 1.8M triangles, while our system takes about 13s per frame on an NVIDIA GeForce GTX 1080.

8 CONCLUSION AND LIMITATIONS

We present a GPU-based self-collision culling method, PSCC, based on a combination of normal cone culling (Algorithm 1) and spatial hashing techniques (Algorithm 3). We use a NCTF-based parallel algorithm, along with sprouting and shrinking operators to maintain compact NCTFs on GPU. This algorithm perform high-level self-collision culling and compute the NCTF node IDs. We use the NCTF node IDs and spatial location information to build an enhanced spatial hashing for low-level culling between triangles. This enhanced spatial hashing technique can perform inter-object collisions and also used for self-collision checking based on normal cone tests. We also redesign the collision handling pipeline of a GPU-based cloth simulation system by reusing the computation of broad phrase culling. We have demonstrated its performance on many complexly layered cloth benchmarks containing 80 − 300K triangles. We observe significant speedups (4 − 6 X) and much less memory overhead, as compared to prior GPU-based systems.

Our approach has several limitations. For tangled, collision detection and penetration handling still remain a major bottleneck in terms of the overall running time (Figure 11). We need better techniques to exploit frame-to-frame coherence since only a few of the vertices involved in the penetrations tend to move between the frame. Also, our approach uses normal cone culling for self-collision culling. For meshes undergoing topological changes, the normal cones and their associated contour edges need to be updated on-the-fly.

There are many avenues for future research. In addition to overcoming the limitations, we feel that it is possible to further improve the performance by exploiting the memory hierarchy and cache of modern GPUs (e.g. combined with [Wang et al. 2018]). Also, it will be interesting to explore the computation potential of multiple GPUs by performing data/task partitioning wisely and perform interactive cloth and deformable simulations.

ACKNOWLEDGEMENTS

This research is supported in part by the National Key R&D Program of China (2017YFB1002703), NSFC (61732015, 61572423, 61572424), the Science and Technology Project of Zhejiang Province (2018C01080), and Zhejiang Provincial NSFC (LZ16F020003).

REFERENCES

Jernej Barbič and Doug L. James. 2010. Subspace Self-Collision Culling. *ACM Trans. on Graphics (SIGGRAPH 2010)* 29, 4 (2010), 81:1–81:9.

Robert Bridson, Ronald Fedkiw, and John Anderson. 2002. Robust treatment of collisions, contact and friction for cloth animation. *ACM Trans. Graph. (SIGGRAPH)* 21, 3 (July 2002), 594–603.

Mathias Eitz and Lixu Gu. 2007. Hierarchical Spatial Hashing for Real-time Collision Detection. In *IEEE International Conference on Shape Modeling and Applications*. 61–70.

Wenshan Fan, Bin Wang, JeanClaude Paul, and Jiaguang Sun. 2011. A Hierarchical Grid Based Framework for Fast Collision Detection. *Computer Graphics Forum* 30, 5 (2011), 1451–1459.

Xavier Faure, Florence Zara, Fabrice Jaillet, and Jean-Michel Moreau. 2012. An Implicit Tensor-Mass Solver on the GPU for Soft Bodies Simulation. In *Proceedings of VRIPHYS*. 1–10.

Jae-Pil Heo, Joon-Kyung Seong, DukSu Kim, Miguel A. Otaduy, Jeong-Mo Hong, Min Tang, and Sung-Eui Yoon. 2010. FASTCD: Fracturing-Aware Stable Collision Detection. In *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*, MZoran Popovic and Miguel Otaduy (Eds.). The Eurographics Association.

Chenfanfu Jiang, Theodore Gast, and Joseph Teran. 2017. Anisotropic Elastoplasticity for Cloth, Knit and Hair Frictional Contact. *ACM Trans. Graph.* 36, 4, Article 152 (July 2017), 14 pages.

Duksu Kim, Jae-Pil Heo, Jaehyuk Huh, John Kim, and Sung-eui Yoon. 2009. HPCCD: hybrid parallel continuous collision detection using CPUs and GPUs. 28 (10 2009), 1791–1800.

James T. Klosowski, Martin Held, Joseph S. B. Mitchell, Henry Sowizral, and Karel Zikan. 1998. Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics* 4, 1 (Jan. 1998), 21–36.

Sylvain Lefebvre and Hugues Hoppe. 2006. Perfect Spatial Hashing. In *ACM SIGGRAPH 2006 Papers (SIGGRAPH '06)*. ACM, New York, NY, USA, 579–588.

Tsai-Yen Li and Jin-Shin Chen. 1998. Incremental 3D Collision Detection with Hierarchical Data Structures. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST '98)*. 139–144.

G. M. Morton. 1966. A computer oriented geodetic data base and a new technique in file sequencing. *Physics of Plasmas* 24, 7 (1966), 159–173.

Simon Pabst, Artur Koch, and Wolfgang Straßer. 2010. Fast and Scalable CPU/GPU Collision Detection for Rigid and Deformable Surfaces. *Comp. Graph. Forum* 29, 5 (2010), 1605–1612.

Xavier Provot. 1995. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In *Proc. of Graphics Interface*. 147–154.

X. Provot. 1997. Collision and Self-collision Handling in Cloth Model Dedicated to Design Garments. In *Graphics Interface*. 177–189.

Sara C. Schvartzman, Álvaro G. Pérez, and Miguel A. Otaduy. 2010. Star-contours for Efficient Hierarchical Self-collision Detection. *ACM Trans. Graph.* 29, 4, Article 80 (July 2010), 8 pages.

Andrew Selle, Jonathan Su, Geoffrey Irving, and Ronald Fedkiw. 2009. Robust High-Resolution Cloth Using Parallelism, History-Based Collisions, and Accurate Friction. *IEEE Trans. Vis. Comp. Graph.* 15, 2 (March 2009), 339–350.

Leanne M. Sutherland, Philippa F. Middleton, Jeffrey Hamdorf, Patrick Cregan, David Scott, and Guy J. Maddern. 2006. Surgical Simulation: A Systematic Review. *Annals of Surgery* 243, 3 (2006), 291–300.

Min Tang, Sean Curtis, Sung-Eui Yoon, and Dinesh Manocha. 2009. ICCD: Interactive Continuous Collision Detection between Deformable Models Using Connectivity-Based Culling. *IEEE Transactions on Visualization and Computer Graphics* 15 (2009), 544–557.

Min Tang, Dinesh Manocha, Jiang Lin, and Ruofeng Tong. 2011. Collision-Streams: Fast GPU-based collision detection for deformable models. In *Proceedings of I3D*. 63–70.

Min Tang, Dinesh Manocha, and Ruofeng Tong. 2010a. MCCD: Multi-Core collision detection between deformable models using front-based decomposition. *Graphical Models* 72, 2 (2010), 7–23.

Min Tang, Dinesh Manocha, and Ruofeng Tong. 2010b. MCCD: Multi-Core collision detection between deformable models using front-based decomposition. *Graphical Models* 72, 2 (2010), 7–23.

Min Tang, Ruofeng Tong, Rahul Narain, Chang Meng, and Dinesh Manocha. 2013. A GPU-based Streaming Algorithm for High-Resolution Cloth Simulation. *Comp. Graph. Forum (Pacific Graphics)* 32, 7 (2013), 21–30.

Min Tang, Ruofeng Tong, Zhendong Wang, and Dinesh Manocha. 2014. Fast and Exact Continuous Collision Detection with Bernstein Sign Classification. *ACM Trans. Graph. (SIGGRAPH Asia)* 33 (November 2014), 186:1–186:8. Issue 6.

Min Tang, Huamin Wang, Le Tang, Ruofeng Tong, and Dinesh Manocha. 2016. CAMA: Contact-Aware Matrix Assembly with Unified Collision Handling for GPU-based Cloth Simulation. *Computer Graphics Forum (Proceedings of Eurographics 2016)* 35, 2 (2016), 511–521.

P. Volino and N. M. Thalmann. 1994. Efficient Self-Collision Detection on Smoothly Discretized Surface Animations using Geometrical Shape Regularity. *Comp. Graph. Forum* 13, 3 (1994), 155–166.

Huamin Wang, Ravi Ramamoorthi, and James F. O'Brien. 2011. Data-Driven Elastic Models for Cloth: Modeling and Measurement. *ACM Trans. Graph. (SIGGRAPH)* 30, 4 (July 2011), 71:1–11.

Tongtong Wang, Zhihua Liu, Min Tang, Ruofeng Tong, and Dinesh Manocha. 2017. Efficient and Reliable Self-Collision Culling Using Unprojected Normal Cones. *Computer Graphics Forum* 36, 8 (2017), 487–498.

Xinlei Wang, Min Tang, Dinesh Manocha, and Ruofeng Tong. 2018. Efficient BVH-based Collision Detection Scheme with Ordering and Restructuring. *Computer Graphics Forum (Proceedings of Eurographics 2018)* 37, 2 (2018).

René Weller, Nicole Debowski, and Gabriel Zachmann. 2017. kDet: Parallel Constant Time Collision Detection for Polygonal Objects. *Computer Graphics Forum* 36, 2 (2017), 131–141.

Sai-Keung Wong and Yu-Chun Cheng. 2014. Continuous Self-Collision Detection for Deformable Surfaces Interacting with Solid Models. *Computer Graphics Forum* 33, 6 (2014), 143–153.

Sai-Keung Wong, Wen-Chieh Lin, Chun-Hung Hung, Yi-Jheng Huang, and Shing-Yeu Lii. 2013. Radial View Based Culling for Continuous Self-collision Detection of Skeletal Models. *ACM Trans. Graph.* 32, 4, Article 114 (July 2013), 10 pages.

Tsz Ho Wong, Geoff Leach, and Fabio Zambetta. 2014. An adaptive octree grid for GPU-based collision detection of deformable objects. *Visual Computer* 30, 6-8 (2014), 729–738.

Xinyu Zhang and Y. J. Kim. 2014. Scalable Collision Detection Using p-Partition Fronts on Many-Core Processors. *IEEE Transactions on Visualization and Computer Graphics* 20, 3 (March 2014), 447–456.

Changxi Zheng and Doug L. James. 2012. Energy-based Self-Collision Culling for Arbitrary Mesh Deformations. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2012)* 31, 4 (Aug. 2012), 98:1–98:12.