Efficient and Reliable Self-Collision Culling using Unprojected Normal Cones

Tongtong Wang[1], Zhihua Liu[1], Min Tang[1], Ruofeng Tong[1], Dinesh Manocha[2,1] †

[1]Zhejiang University, [2]University of North Carolina at Chapel Hill
http://gamma.cs.unc.edu/SCC/

Abstract

We present an efficient and accurate algorithm for self-collision detection in deformable models. Our approach can perform discrete and continuous collision queries on triangulated meshes. We present a simple and linear time algorithm to perform the normal cone test using the unprojected 3D vertices, which reduces to a sequence point-plane classification tests. Moreover, we present a hierarchical traversal scheme that can significantly reduce the number of normal cone tests and the memory overhead using front-based normal cone culling. The overall algorithm can reliably detect all (self) collisions in models composed of hundred of thousands of triangles. We observe considerable performance improvement over prior CCD algorithms.

1. Introduction

Fast and accurate collision detection is important for generating realistic deformations. At a broad level, prior work can be classified into discrete (DCD) methods that check for collisions at an instant of time, and continuous (CCD) techniques that check for colliding regions within a time interval. The latter are used to avoid missing any collisions and to perform reliable simulations by maintaining intersection-free meshes.

Complex mesh models composed of hundreds of thousands of triangle primitives are frequently used in cloth or FEM simulation. Collision detection is regarded as one of the major bottlenecks in these applications and it is important to accurately check for collisions between all the primitives [BFA02, BEB12, Wan14, TTWM14].

Some of the commonly used algorithms use bounding volume hierarchies (BVHs) to accelerate collision detection. These techniques work well in terms of inter-object collision detection. However, self-collision checking in deformable models can be challenging as many adjacent or nearby primitives of a deforming mesh are in close proximity and not culled by bounding volume tests. Even if a mesh is intersection-free, checking all the primitives for self-collisions can be expensive.

The most commonly used methods for self-collisions are based on normal cones for DCD [VT94, Pro97, SPO10]. However, the computational overhead of normal cone tests during BVH traversal

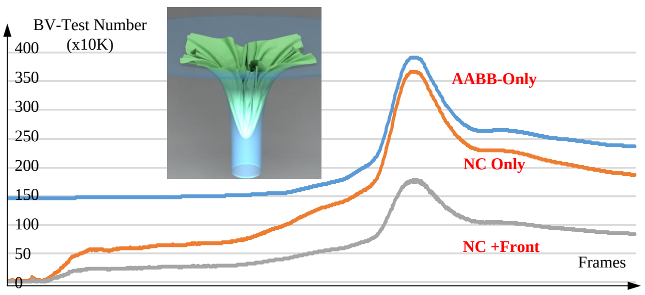


Figure 1: ***Improved collision culling:*** *We demonstrate the benefits of our self-collision algorithm on the Funnel benchmark (64K triangles). The three curves highlight the culling efficiencies of various algorithms. These include prior schemes that do not perform self-collision culling (i.e. AABB-only) vs. two variants of our algorithm. Our novel CCD algorithm can significantly reduce the number of false positives.*

can be high. This approach has been extended to CCD [TCYM09], but the additional cost of performing continuous normal cone culling slows down the overall CCD algorithm due to the quadratic complexity of the continuous contour test. Other techniques include energy-based methods [BJ10, ZJ12] and radial-based CCD culling for skeletal models [WLH*13]. However, self-collision culling continues to remains a bottleneck, especially for CCD between arbitrary deformable models.

Main Results: We present a fast and accurate approach for self-collision detection in triangulated models. Our formulation is based

† Tongtong Wang and Zhihua Liu are joint first authors. Min Tang is the corresponding author.

© 2017 The Author(s)
Computer Graphics Forum © 2017 The Eurographics Association and John Wiley & Sons Ltd. Published by John Wiley & Sons Ltd.

on normal cones and is applicable to DCD as well as CCD. In order to reduce the overhead of normal cone tests during BVH traversal, we present two novel algorithms:

1. **Linear time normal cone test using unprojected contours:** We present a novel contour test algorithm that can directly perform the queries on the 3D vertices without computing any projections on a plane. Our formulation reduces to performing point-plane orientation tests on the 3D vertices. The resulting computation is simple and accurate, and reduces to evaluating the sign of algebraic expressions for DCD as well as CCD (Section 3).
2. **Front-based normal cone culling:** We present an incremental BVH traversal algorithm that combines the normal cone test with the BVTT (bounding volume test tree) front computation. This reduces the hierarchy traversal overhead based on spatial and temporal coherence, and also the memory overhead (Section 4).

The resulting algorithms are simple to implement and can accurately detect all the collisions. It has no preprocessing overhead and can perform fast collision queries on complex benchmarks on a single CPU core (Section 5). We observe an order of magnitude improvement in the performance of the normal cone tests over prior approaches. We also highlight the overall speedups in cloth simulation due to our novel CCD algorithm.

2. Related Work

In this section, we give a brief overview of prior work on collision detection between deformable models. The simplest culling algorithms use BVHs (bounding volume hierarchies) that are based on k-DOPs or AABBs. These can be combined with self-collision culling techniques [Pro97, MKE03, TCYM09, SPO10, ZJ12].

Low level culling: Many low-level culling techniques have been proposed to reduce the number of elementary tests between the triangle pairs for CCD, such as removing redundant elementary tests [GKJ*05, CTM08, WB06, TMT10a, TMY*11] or using bounding volumes of the primitives [HF07]. Our approach can be combined with these low-level acceleration techniques.

Clustering: Many mesh decomposition and clustering strategies have been proposed to compute tighter fitting hierarchies to improve the culling efficiency. Most of these techniques are used as a preprocess [EL01, WB14]. Schvartzman et al. [SPO10] presented a self-collision test tree (SCTT) that is precomputed to accelerate the self-collision queries for general deformable models. By executing hierarchically, their cost can be reduced to $O(1)$. Wong et al. [WLH*13] presented a continuous self-collision detection algorithm for skeletal models and extended it to check for collisions between a deformable surface and a simple solid model [WC14]. He et al. [HOEM15] presented a dynamic clustering algorithm for topology changing models. Our approach can be easily combined with these methods to obtain the fastest CCD query performance.

Reliable collision queries: Most earlier methods for DCD and CCD are implemented using floating point computations and numerical tolerances. However, numerical errors in arithmetic operations, along with the tolerances, can impact their accuracy, especially for elementary tests. Recently, many reliable algorithms have been proposed for elementary tests based on exact arithmetic operations [BEB12, TTWM14] or conservative float-point computation with tight bounds [Wan14, WTTM15]. It is also important to ensure that the self-collision culling tests are reliable.

Front based traversal: BVTT (Bounding Volume Test Tree) front tracking has been used to accelerate collision detection [LC98]. By performing tests using the BVTT front generated from the last time step, these methods reduce the runtime overhead and make it easier to parallelize on CPUs and GPUs [TMLT11, ZK14]. However, the memory overhead for storing the BVTT front can be high. We combine our normal cone test with BVTT-based front to significantly reduce the time and storage complexity.

3. Unprojected Normal Cone Test

In this section, we introduce the notation and present our unprojected normal cone test for self-collision culling.

3.1. Background and Notation

In this paper, we consider discrete (DCD) as well as continuous collision detection (CCD) problems on models represented as 2-manifold triangle meshes, possibly with boundaries. Our approach uses the mesh connectivity information to perform collision culling. DCD deals with checking whether any distinct triangle primitives overlap at a given time instant. On the other hand, CCD algorithms model the motion of each object or triangle using a continuous trajectory between the successive instants of a simulator and check for collisions along the trajectory. Our CCD culling algorithm also uses linearly interpolating trajectories [Pro97, BFA02].

We use the following notation in the rest of the paper: Lower case letters in normal fonts (e.g., a, b, a_i) represent scalar variables and upper case letters (e.g., L, $J(t)$) represent scalar functions. Lower case letters in bold face fonts (e.g., $\mathbf{a}$, $\mathbf{b_t}$) represent vector quantities and points, and upper case letters in bold face fonts (e.g., $\mathbf{L}$, $\mathbf{J}(\mathbf{t})$) represent vector-valued functions. The operators '$*$', '$\cdot$', and '$\times$' denote the usual scalar multiplication, dot product, and cross product, respectively. We use following acronyms in the rest of the paper: *PPC:* stands for point-plane classification; *BV, BVH, BVTT:* stand for bounding volume, bounding volume hierarchy, and bounding volume test tree, respectively.

3.2. Normal Cone Test

Some of the widely used algorithms for self-collision detection are based on the normal cone test [VT94]. Given a continuous surface, S, bounded by a contour, C, a sufficient criterion for no self-collision is based on the following two conditions:

1. **Bounds on the normals:** There is a vector, $\mathbf{v}$, such that $(\mathbf{N} \cdot \mathbf{v}) > 0$ for every point of the surface, S, where $\mathbf{N}(S)$ is the normal vector at the point S on the surface.
2. **No boundary self-intersections:** The projection of the contour C along the vector $\mathbf{v}$ does not have any self-intersections on the projected plane.

The first condition is called the *surface normal test* and the second condition is also known as the *contour test*. Provot [Pro97]

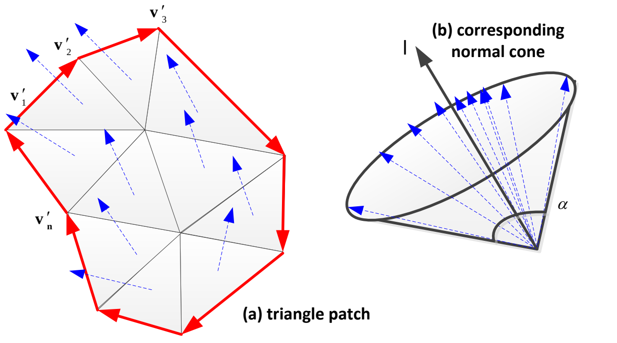


Figure 2: ***Normal Cone:*** *For an input triangle mesh (a), its contour edges are defined by an order list of vertices* $\mathbf{v}'_1, \mathbf{v}'_2, \ldots, \mathbf{v}'_n$ *(the red arrows). Its corresponding normal cone* $C_N(\alpha, \mathbf{l})$ *contains all the normal vectors of the triangles (b), where l is the axis, and* α *is the apex angle.*

presented an efficient method to evaluate the first condition based on normal cones, which corresponds to a bound on the Gauss map of S. The normal cone for a mesh of triangles can be computed by combining the normal vectors of individual triangles. Given an input normal cone $C_N(\alpha, \mathbf{l})$ bounds all the normal vectors of the triangles of a given triangle patch, where α is the apex angle, and $\mathbf{l}$ is the middle axis of the cone (see Fig. 2). The normal cone test has been used for DCD [Pro97,SPO10], and has been extended to CCD by Tang et al. [TCYM09].

Complexity of Contour Tests: In practice, the contour test tends to be more expensive as compared to surface normal test and can take up to 60% of the running time [SPO10] for DCD computations. This test typically involves computing a projection of the contour of S and checking for self-intersections on the resulting plane. Recently, Schvartzman et al. [SPO10] used a line-search star-shaped test to detect star-shaped projection, which guarantees that there will be no self-intersections. Furthermore, they used a pre-computed self-collision test tree (SCTT) to accelerate the computation. As a result, the resulting algorithm has linear time complexity in the number of contour edges. However, the projection computations can be expensive and take up to 37% of total running time [SPO10]. Furthermore, they are prone to floating-point errors and can result in accuracy issues during the contour tests. Some prior self-collision algorithms either omit the contour test computation for certain cases [VT94] or use some approximations [Pro97]. Heo et al. [HSK*10] described an approximate scheme to avoid the projection computation.

Continuous Queries: It is not clear whether the linear time algorithm for contour test in [SPO10] extends to CCD. The resulting formulation based on kernel tests reduces to checking for overlaps between intervals, as opposed to line intersections for DCD. Instead, the only known algorithm for contour tests for CCD reduces to performing $O(n^2)$ EE elementary tests, where n is the number of contour edges [TCYM09]. This is based on projecting the continuously deforming edges to a plane and checking each pair for overlap by solving a cubic equation. In such cases, the contour test becomes a major bottleneck in the overall algorithm.

3.3. Contour Test Using Point-Plane Classification

We present a novel algorithm for the contour test that does not involve any projection computations. The resulting contour test can be performed in $O(n)$ time for DCD as well as CCD, where n is the number of contour edges in C. Furthermore, our algorithm only needs to perform sign evaluation in terms of 3D point-plane side tests, i.e. determining the orientation of a point with respect to a plane in 3D.

We initially present our contour test for DCD, and later extend it to CCD computations. We first describe the notion of a PPC (point-plane classification) based contour test on a 2D projection plane, then extend it to the unprojected 3D contour points.

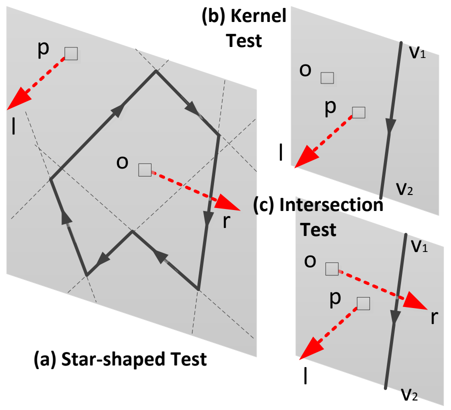


Figure 3: ***Star-shaped Test:*** *The projection plane is defined by a point* $\mathbf{p}$ *and a normal vector* $\mathbf{l}$*. A star-shaped contour polygon in the plane* $\{\mathbf{p}, \mathbf{l}\}$ *has: (1) at least one kernel point* $\mathbf{o}$ *located inside the half-spaces of all the contour edges; (2) any ray starting from* $\mathbf{o}$ *along the direction* $\mathbf{r}$ *can intersect the boundary only once. Both these conditions can be tested by computing the orientation of a point with respect to different lines.*

Given a contour, we first project the points onto a projection plane, which is defined by a point $\mathbf{p}$ and a normal vector $\mathbf{l}$ and represented as $\{\mathbf{p}, \mathbf{l}\}$ (see Fig. 3 (a)). If the projected contour is a star-shaped polygon, it guarantees no self-intersections [SPO10]. The star-shaped test is performed in two steps:

1. **Kernel Test:** There is at least one kernel point (e.g., point $\mathbf{o}$) located at the inside of the half-spaces of all the contour edges (Fig. 3 (b)).
2. **Intersection Test:** A ray starting from the kernel point in the direction $\mathbf{r}$ can intersect the boundary only once (Fig. 3 (c)).

We assume that all the contour edges are oriented in a clockwise or counter-clockwise direction, with respect to $\mathbf{l}$.

We define a scalar function $\mathbf{Side}(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{l})$ and a sign evaluation function $\mathbf{SideSign}(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{l})$, where $\mathbf{a}, \mathbf{b}, \mathbf{c}$ are three points that lie on the projection plane $\{\mathbf{p}, \mathbf{l}\}$:

$$\mathbf{Side}(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{l}) = ((\mathbf{a} - \mathbf{c}) \times (\mathbf{b} - \mathbf{c})) \cdot \mathbf{l}$$

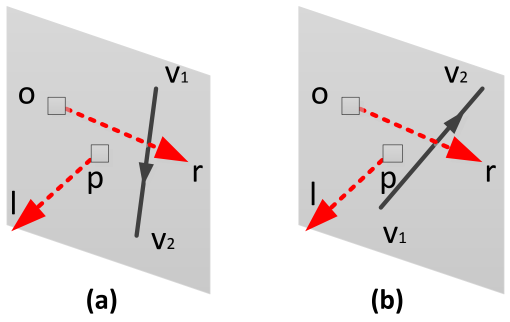


Figure 4: ***Intersection Test:*** *The two different cases (a) and (b), in which there is an intersection between the ray* $\{\mathbf{o},\mathbf{r}\}$ *and a contour edge* $\{\mathbf{v_1},\mathbf{v_2}\}$*, can be tested by computing the orientations of points with respect to different lines.*

$$\mathbf{SideSign}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{l}) = \begin{cases} 1, & \mathbf{Side}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{l}) > 0, \\ -1, & \mathbf{Side}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{l}) < 0, \\ 0, & \mathbf{Side}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{l}) = 0. \end{cases}$$

Note that the choice of **p** does not affect the values of **Side**() and **SideSign**().

The kernel test on the projected plane is performed using PPC between **o** and all the projected contour edges. For an edge defined by the two vertices $\mathbf{v_1}$ and $\mathbf{v_2}$, the orientation between them is computed using $\mathbf{SideSign}(\mathbf{o},\mathbf{v_1},\mathbf{v_2},\mathbf{l})$. Therefore, the kernel test can be expressed as checking the signs of n algebraic expressions[†]:

$$\begin{aligned} \mathbf{SideSign}(\mathbf{o},\mathbf{v_1},\mathbf{v_2},\mathbf{l}) &== \mathbf{SideSign}(\mathbf{o},\mathbf{v_2},\mathbf{v_3},\mathbf{l}) \\ == \ldots &== \mathbf{SideSign}(\mathbf{o},\mathbf{v_n},\mathbf{v_1},\mathbf{l}), \end{aligned}$$

where $\mathbf{v_1},\mathbf{v_2},\ldots,\mathbf{v_n}$ are the projected vertices of the contour.

The intersection test is also performed using sign evaluations. Let us consider the cases (a) and (b) (Figure 4), in which there is an intersection. For both cases, $\mathbf{v_1}$ and $\mathbf{v_2}$ lie on the different sides of the ray $\{\mathbf{o},\mathbf{r}\}$, and side of **o** with respect to $\overrightarrow{\mathbf{v_1}\mathbf{v_2}}$ is the same side of $\mathbf{v_2}$ with respect to the ray. We can check for these conditions based on evaluating the following expressions:

$$\begin{aligned} \mathbf{SideSign}(\mathbf{o},\mathbf{v_1},\mathbf{v_2},\mathbf{l}) &= \mathbf{SideSign}(\mathbf{v_2},\mathbf{o},\mathbf{o}+\mathbf{r},\mathbf{l}) \quad \textit{and} \\ \mathbf{SideSign}(\mathbf{v_1},\mathbf{o},\mathbf{o}+\mathbf{r},\mathbf{l}) &\neq \mathbf{SideSign}(\mathbf{v_2},\mathbf{o},\mathbf{o}+\mathbf{r},\mathbf{l}) \end{aligned}$$

Only when both conditions are fulfilled, we count an intersection between the ray and the contour edge. The idea behind the two equations is that, the first equation is to ensure there is an intersection between a line (defined by **o** and **r**) and a line segment (defined by $\mathbf{v_1}$ and $\mathbf{v_2}$). The second equation is to ensure the intersection is on the right side of the line (i.e. on the ray).

We compute the intersection number between the ray and each contour edge. The intersection test holds only when the intersection number equals to 1. For the case that the intersection number is greater than 1, the test can directly return a false answer.

[†] We ignore the case that $\mathbf{SideSign}(\ldots) = 0$, as the test returns an answer of false.

Overall, the kernel test and the intersection test are reduced to computing dot and cross products, and these tests perform PPC using SideSign() functions.

3.4. Unprojected Contour Test

The contour test for the 2D projected vertices can be extended to directly operate on the unprojected 3D contour vertices, i.e., the original vertices of the input mesh contours. Let $\mathbf{a}',\mathbf{b}',\mathbf{c}'$ be the original (unprojected) 3D contour vertices, and $\mathbf{a},\mathbf{b},\mathbf{c}$ be the corresponding projections on the projection plane $\{\mathbf{p},\mathbf{l}\}$. We use the following theorem to perform conservative contour tests on the original vertices:

Unprojected Orientation Test Theorem: *Given three 3D points* $\mathbf{a}',\mathbf{b}',\mathbf{c}'$*, and a projection direction* $\mathbf{l}$*. Let their projections*

Algorithm 1 UnprojectedContourTest(C_N): Perform unprojected contour test on the boundary contour of a given normal cone C_N.
Input: A normal cone $C_N(\alpha,\mathbf{l})$, where α is the apex angle, and $\mathbf{l}$ is the axis of the cone, and its boundary contour is defined by an ordered list of vertices, i.e., $\mathbf{v'_1},\mathbf{v'_2},\ldots,\mathbf{v'_n}$.
Output: **true** for no self-intersection on the projected contour, **false** otherwise.

```
// Prepare parameters for kernel test and intersection test.
$\mathbf{o} = \frac{\sum_1^n \mathbf{v'_i}}{n}$;  $\mathbf{l}$ is middle axis of input normal cone $C_N$
if $\mathbf{l}$ parallel to vector $\{\mathbf{0},\mathbf{1},\mathbf{0}\}$ then
    $\mathbf{r} = \{\mathbf{1},\mathbf{0},\mathbf{0}\}$
else
    $\mathbf{r} = \mathbf{l} \times \{\mathbf{0},\mathbf{1},\mathbf{0}\}$
end if
// Initialize intersection number to zero.
$intsNum = 0$
// Get side sign at the first contour segment.
$s_0 = \mathbf{SideSign}(\mathbf{o},\mathbf{v'_1},\mathbf{v'_2},\mathbf{l})$
if $s_0 == 0$ then
    return false;
end if
// Perform kernel test and intersection test on each contour segment.
for each edge $\mathbf{v'_i},\mathbf{v'_{i+1}}$ do
    // Perform side test.
    if $s_0 \neq \mathbf{SideSign}(\mathbf{o},\mathbf{v'_i},\mathbf{v'_{i+1}},\mathbf{l})$ then
        return false;
    end if
    // Perform intersection test.
    $s_1 = \mathbf{SideSign}(\mathbf{v'_i},\mathbf{o},\mathbf{o}+\mathbf{r},\mathbf{l})$
    $s_2 = \mathbf{SideSign}(\mathbf{v'_{i+1}},\mathbf{o},\mathbf{o}+\mathbf{r},\mathbf{l})$
    if $s_1 == 0$ or $s_2 == 0$ then
        return false; // Can't determine the intersection.
    end if
    if $s_2 = s_0$ and $s_1 \neq s_2$ then
        $intsNum++$
        if $intsNum > 1$ then
            return false; // More than one intersection.
        end if
    end if
end for
return true;
```

on a plane perpendicular to $\mathbf{l}$ *be* $\mathbf{a},\mathbf{b},\mathbf{c}$, *respectively. In this case,* $\mathbf{SideSign}(\mathbf{a}',\mathbf{b}',\mathbf{c}',\mathbf{l}) = \mathbf{SideSign}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{l})$.

Proof By the definition of $\mathbf{a},\mathbf{b},\mathbf{c}$, we have:

$$\mathbf{a} = \mathbf{a}' + u*\mathbf{l}; \quad \mathbf{b} = \mathbf{b}' + v*\mathbf{l}; \quad \mathbf{c} = \mathbf{c}' + w*\mathbf{l}.$$

where u, v,and w are three scalars. By substituting them into the definition of $\mathbf{Side}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{l})$, we obtain:

$$\begin{aligned}
&\mathbf{Side}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{l}) \\
=\ & ((\mathbf{a}-\mathbf{c}) \times (\mathbf{b}-\mathbf{c})) \cdot \mathbf{l} \\
=\ & (((\mathbf{a}'-\mathbf{c}') + (u-w)*\mathbf{l}) \\
\times\ & ((\mathbf{b}'-\mathbf{c}') + (v-w)*\mathbf{l})) \cdot \mathbf{l} \\
=\ & ((\mathbf{a}'-\mathbf{c}') \times (\mathbf{b}'-\mathbf{c}')) \cdot \mathbf{l} \\
=\ & \mathbf{Side}(\mathbf{a}',\mathbf{b}',\mathbf{c}',\mathbf{l})
\end{aligned}$$

□

Based on this theorem, the kernel test and intersection test, which are based on **Side**(), can be directly evaluated using the original 3D coordinates of the contour vertices. As a result, there is no need to perform projection computations for the conservative star-shape based contour test.

The pseudo-code for the unprojected contour test is given in Algorithm 1. Given an input normal cone $C_N(\alpha,\mathbf{l})$ bounds all the normal vectors of the triangles of a given triangle patch, where α is the apex angle, and $\mathbf{l}$ is the middle axis of the cone. The boundary contour of $C_N(\alpha,\mathbf{l})$ is defined by a ordered list of vertices, i.e., $\mathbf{v}_1',\mathbf{v}_2',\ldots,\mathbf{v}_n'$. Our algorithm for the contour test will return **true** if there are no self-intersections on the projected contour, but will return **false** otherwise. We first define the parameters to perform the kernel test and intersection test (Line 1-9, Algorithm 1).

- **Projection Direction l:** We use the axis $\mathbf{l}$ of the input normal cone as the projection direction.
- **Kernel Point o:** We use the average of the input contour vertices as the kernel point, i.e., $\mathbf{o} = \frac{\sum_i^n \mathbf{v}_i'}{n}$.
- **Ray Direction r:** In theory, it can be any vector perpendicular to $\mathbf{l}$. In practice, we choose it to be $\mathbf{l} \times \{\mathbf{0},\mathbf{1},\mathbf{0}\}$ if $\mathbf{l}$ is not parallel to vector $\{\mathbf{0},\mathbf{1},\mathbf{0}\}$, otherwise, we set $\mathbf{r}$ to be $\{\mathbf{1},\mathbf{0},\mathbf{0}\}$ (Lines 3-7, Algorithm 1).

After specifying all the parameters, we perform the kernel tests and intersection tests over all the contour edges (Line 9-34). The intersection test is performed at Lines 21-32. If the intersection number exceeds 1, the algorithm returns **false**. The contour test returns true only if the kernel test and the intersection test return true.

Query Performance: The worst case for the unprojected contour tests occurs when it returns a true answer and there is no early exit. In that case, it perform $3n$ **SideSign**() evaluations for DCD, where n is the number of contour edges. Each **SideSign**() evaluation requires one cross product and one dot product computation between two 3D vectors. So the total operation count is bounded by $3n$ cross products and $3n$ dot products for each contour test.

Algorithm 2 SelfCollide(N): Normal Cone Culling using Unprojected Contour Test (NC). By traversing the BVH recursively, meshes that satisfy normal cone test are culled.
Input: A node N on the BVH. C_N is the normal cone associated with N.
Output: No return value.

```
1:  if IsLeaf(N) then
2:      return; // Traversal terminated.
3:  end if
4:
5:  if ApexAngle($C_N$) < $\pi$ then
6:      if UnprojectedContourTest($C_N$) = true then
7:          return; // The corresponding mesh has no self-collisions.
8:      end if
9:  end if
10: // Check the descendants.
11: SelfCollide(N → LeftChild)
12: SelfCollide(N → RightChild)
13: Collide(N → LeftChild, N → RightChild)
```

3.5. Normal Cone Culling

Based on Algorithm 1, the overall algorithm for normal cone culling is shown in Algorithm 2. Our algorithm involves no precomputation and updates a BVH to perform hierarchical computations. The self-collision checking starts at the root node of the BVH, and traverses in a top-down manner. For a node N of the BVH, and its associated normal cone C_N, we check whether the apex angle of C_N is less than π and also perform the unprojected contour test. If these two tests are satisfied, then we do not need to traverse to the children of N to check for self-collisions.

3.6. Normal Cone Culling for CCD

The contour test and normal cone culling algorithms described above can be extended to CCD. One of the main issues is to compute a bound on the surface normals that varies during the time interval based on linearly interpolating triangle vertices. Moreover, we need to perform the unprojected contour test on the continuously varying vertices of the contour. We compute a conservative normal bound for each deforming triangle, and merge these bounds in a bottom-up manner on the BVH to update the apex angles of all normal cones. In order to perform a continuous unprojected contour test, we assume that a vertex $\mathbf{p_i}$ is moving with the constant velocity $\mathbf{v_i}$ during a given time interval $[0,1]$.

We first define the function **CSide1**() as:

$$\mathbf{CSide1}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{l}) = ((\mathbf{a}-\mathbf{c}) \times (\mathbf{b}-\mathbf{c})) \cdot \mathbf{l}$$

where $\mathbf{b} = \mathbf{b_0} + \mathbf{v_b}*t$, $\mathbf{c} = \mathbf{c_0} + \mathbf{v_c}*t$. However, $\mathbf{a}$ and $\mathbf{l}$ are unchanged during the time interval $[0,1]$. $\mathbf{b_0}$ and $\mathbf{c_0}$ are the position of two vertices at $t = 0$. $\mathbf{v_b}$ and $\mathbf{v_c}$ are their moving velocities during the time interval ($t \in [0,1]$). Therefore:

$$\begin{aligned}
&\mathbf{CSide1}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{l}) \\
=\ & ((\mathbf{b_0}+\mathbf{v_b}*t-\mathbf{a}) \times (\mathbf{c_0}+\mathbf{v_c}*t-\mathbf{a})) \cdot \mathbf{l} \\
=\ & k_0 * B_0^2(t) + k_1 * B_1^2(t) + k_2 * B_2^2(t),
\end{aligned}$$

where $B_i^2(t)'s$ are second order polynomials in Bernstein basis.

The detailed derivation of k_i is as follows:

$$\begin{aligned} k_0 &= ((\mathbf{b_0}-\mathbf{a})\times(\mathbf{c_0}-\mathbf{a}))\cdot\mathbf{l}, \\ k_1 &= \frac{((\mathbf{v_b}\times(\mathbf{c_0}-\mathbf{a})+(\mathbf{b_0}-\mathbf{a})\times\mathbf{v_c})\cdot\mathbf{l}}{\mathbf{2}} \\ &+ ((\mathbf{b_0}-\mathbf{a})\times(\mathbf{c_0}-\mathbf{a}))\cdot\mathbf{l}, \\ k_2 &= ((\mathbf{b_0}-\mathbf{a})\times(\mathbf{c_0}-\mathbf{a}))\cdot\mathbf{l} \\ &+ (\mathbf{v_b}\times(\mathbf{c_0}-\mathbf{a})+\mathbf{v_c}\times(\mathbf{b_0}-\mathbf{a}))\cdot\mathbf{l} \\ &+ (\mathbf{v_b}\times\mathbf{v_c})\cdot\mathbf{l}. \end{aligned}$$

With this representation, the value of **CSide** is bounded based on the coefficients k_0, k_1, and k_2 for $0 \le t \le 1$. Based on the formulation of **CSide1**(), we define:

$$\mathbf{CSideSign1}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{l}) = \begin{cases} 1, & k_0 > 0, k_1 > 0, k_2 > 0, \\ -1, & k_0 < 0, k_1 < 0, k_2 < 0, \\ 0, & otherwise. \end{cases}$$

This formulation is conservative, but provides a sufficient condition to perform the kernel and intersection tests for CCD.

For the case $\mathbf{a}$ that is moving under constant velocity, i.e., $\mathbf{a} = \mathbf{a_0} + \mathbf{v_a} * t$, and $\mathbf{b}, \mathbf{c}$ are unchanged during the time interval, we define the function:

$$\begin{aligned} &\mathbf{CSide2}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{l}) \\ =\ &((\mathbf{a_0}+\mathbf{v_a}*t-\mathbf{c})\times(\mathbf{b}-\mathbf{c}))\cdot\mathbf{l} \end{aligned}$$

Similarly, the value of $\mathbf{CSide2}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{l})$ is bounded by two constants for all $t \in [0,1]$:

$$\begin{aligned} k'_0 &= ((\mathbf{a_0}-\mathbf{c})\times(\mathbf{b}-\mathbf{c}))\cdot\mathbf{l} \\ k'_1 &= ((\mathbf{a_0}+\mathbf{v_a}-\mathbf{c})\times(\mathbf{b}-\mathbf{c}))\cdot\mathbf{l} \end{aligned}$$

So we define:

$$\mathbf{CSideSign2}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{l}) = \begin{cases} 1, & k'_0 > 0, k'_1 > 0, \\ -1, & k'_0 < 0, k'_1 < 0, \\ 0, & otherwise. \end{cases}$$

By replacing the functions **SideSign**() in Algorithm 1 with new sign functions **CSideSign1**() (Line 12,19) and **CSideSign2**() (Line 23, 24), respectively, the contour test algorithm for DCD can be easily extended to perform conservative contour tests during the time interval $[0,1]$ for CCD (as shown in Algorithm 3).

Query Performance: In the worst case, the unprojected contour test for CCD performs n **CSideSign1**() and $2n$ **CSideSign2**() evaluations, where n is the number of contour edges. Each **CSideSign1**() evaluation requires 8 cross products and 6 dot products computation between two 3D vectors. Each **CSideSign2**() evaluation requires 2 cross products and 2 dot products. So the totally operation count is bounded by $12n$ cross products and $10n$ dot products for each contour test.

4. Front-Based Normal Cone Culling

The hierarchical traversal algorithm based on normal cone tests presented above can accelerate the performance of each normal

Algorithm 3 UnprojectedContourTestForCCD(C_N): Perform unprojected contour test on the boundary contour of a given normal cone C_N.
Input: A normal cone $C_N(\alpha, \mathbf{l})$ where α is the apex angle, and $\mathbf{l}$ is the axis of the cone, and its boundary contour which is defined by a ordered list of vertices, i.e., $\mathbf{v'_1}, \mathbf{v'_2}, ..., \mathbf{v'_n}$.
Output: **true** for no self-intersection on the projected contour, **false** for undeterminable.

```
 1: // Prepare parameters for kernel test and intersection test.
 2: $\mathbf{o} = \frac{\sum_1^n \mathbf{v'_i}_{t=0}}{n}$
 3: $\mathbf{l}$ is middle axis of input normal cone $C_N$
 4: if $\mathbf{l}$ parallel to vector $\{\mathbf{0},\mathbf{1},\mathbf{0}\}$ then
 5:     $\mathbf{r} = \{\mathbf{1},\mathbf{0},\mathbf{0}\}$
 6: else
 7:     $\mathbf{r} = \mathbf{l} \times \{\mathbf{0},\mathbf{1},\mathbf{0}\}$
 8: end if
 9: // Initialize the intersection number to zero.
10: $intsNum = 0$
11: // Get side sign at the first contour segment.
12: $s_0 = \mathbf{CSideSign1}(\mathbf{o},\mathbf{v'_1},\mathbf{v'_2},\mathbf{l})$
13: if $s_0 == 0$ then
14:     return false;
15: end if
16: // Perform kernel test and intersection test on each contour segment.
17: for each edge $\mathbf{v'_i},\mathbf{v'_{i+1}}$ do
18:     // Perform side test.
19:     if $s_0 \ne \mathbf{CSideSign1}(\mathbf{o},\mathbf{v'_i},\mathbf{v'_{i+1}},\mathbf{l})$ then
20:         return false;
21:     end if
22:     // Perform intersection test.
23:     $s_1 = \mathbf{CSideSign2}(\mathbf{v'_i},\mathbf{o},\mathbf{o}+\mathbf{r},\mathbf{l})$
24:     $s_2 = \mathbf{CSideSign2}(\mathbf{v'_{i+1}},\mathbf{o},\mathbf{o}+\mathbf{r},\mathbf{l})$
25:     if $s_1 == 0$ or $s_2 == 0$ then
26:         return false; // Can't determine the intersection.
27:     end if
28:     if $s_2 = s_0$ and $s_1 \ne s_2$ then
29:         $intsNum++$
30:         if $intsNum > 1$ then
31:             return false; // More than one intersection.
32:         end if
33:     end if
34: end for
35: return true;
```

cone test during the hierarchical traversal. However, the resulting algorithm may perform a large number of normal cone tests while handling complex benchmarks represented using hundreds of thousands of triangles. In this section, we present a novel front-based normal cone culling algorithm that can significantly reduce the number of normal cone tests and traversal overhead. Furthermore, it reduces the memory requirements of storing the front.

The basic idea behind our algorithm is shown in Fig. 5. For a scene with deformable objects, we compute its BVH as shown in the upper right corner of Fig. 5. The self-collision detection algorithm corresponds to the traversal of its BVTT (Bounding volume test tree), as shown in Fig. 5. The main goal of using BVTT is to compute the front during the traversal from the previous frame [TMT10b, TMLT11]. By starting the traversal of the current frame directly from the BVTT front, we exploit the spatial and temporal coherence to reduce the traversal cost. However, the memory overhead of this front can be high. For example, for a mesh with 4 K triangles, it takes about 17 MB memory to store the BVTT front between successive frames [TMLT11].

A node $A*B$ in the BVTT represents collision checking between the nodes A and B of the given BVH. In the worst case, the BVTT can have $O(m^2)$ nodes, where m is the number of nodes in the BVH. The front-based algorithms keep track of a subset of BVTT that corresponds to overlapping nodes during the current frame. Instead of storing a global BVTT front (i.e. the thick line in Fig. 5), we decompose it into many sub-segments, as shown by the thin line in Fig. 5. Each sub-segment of the BVTT is associated with a $X*X$ node, which corresponds to checking for self-collisions among all the nodes beneath the internal node X of BVH. Instead of traversing the children of this node hierarchically, we use our normal cone culling algorithm to check for self-collisions. This is shown as the segmented areas in Fig. 5. In particular, we associate each BVTT front sub-segment for $X*X$ with the internal node X on the BVH.

We use the normal cone guided BVTT front tracking algorithm (Algorithm 4), to significantly reduce the traversal overhead as well as size of the front. By traversing the BVH recursively, if a node X satisfies the normal cone test, its sub-segment can be directly skipped for testing. Without the normal cone tests, we will need to track the nodes in this sub-segment to perform self-collision detection. During the BVH tracking, all the visited BVTT nodes are stored into the sub-segment and reused for subsequent frames [TMLT11].

Based on the front-based normal cone algorithm, we obtain the benefits of BVTT front based culling as well as normal cone based culling. With BVTT front tracking, we can perform collision detection in an efficient manner by utilizing temporal and spatial coherence. Furthermore, we combine them with the normal cone test, and this can significantly reduce the size of the BVTT front, as we do not need to store the BVTT front segments for the areas that are culled by our normal cone tests. Figure 6 highlights the reduced size of the normal cone guided BVTT front for the benchmark FlowingCloth. In this case, the size of our normal cone guided BVTT front (the upper curve) is only about 8% of the conventional BVTT front in [TMLT11] (the lower curve).

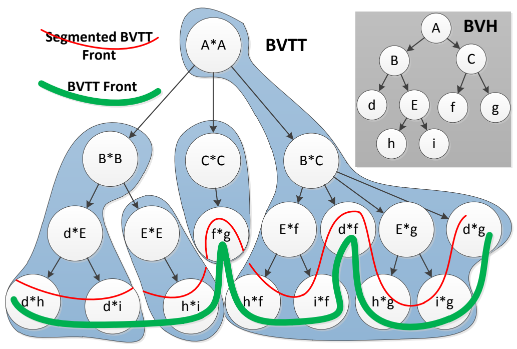


Figure 5: ***Normal cone guided BVTT front tracking:*** *We associate each BVTT front sub-segment for $X*X$ with the internal node X on the BVH. In order to perform self-collision culling, we use a front-based normal cone traversal approach that uses the BVTT front (Algorithm 4). By traversing the BVH recursively, if a node X passes the normal cone test, its associated BVTT front sub-segment is culled. Our approach can reduce the traversal overhead and runtime memory footprint.*

Algorithm 4 SelfCollideWithGuidedFrontTracking(N): a normal cone guided BVTT front tracking algorithm for self-collision detection (NC + Front).

Input: A node N on the BVH. C_N is the normal cone associated with N. $Front_N$ is the BVTT front sub-segment associated with node N.

Output: No return value.

```
 1: if IsLeaf(N) then
 2:     return; // Traversal terminated.
 3: end if
 4:
 5: if ApexAngle($C_N$) < $\pi$ then
 6:     if UnprojectedContourTest($C_N$) = true then
 7:         return; // The region is self-collision free.
 8:     end if
 9: end if
10: // Check the descendants.
11: SelfCollide(N → LeftChild)
12: SelfCollide(N → RightChild)
13: FrontTracking($Front_N$)
```

4.1. Reliable Computation

Our overall collision detection algorithm has three main steps: (1) update BVs and normal cones; (2) perform front-based BVH traversal and normal cone culling (Algorithm 4); (3) perform elementary tests between triangle pairs and may use lower level culling methods for acceleration. For stage (1), we compute a conservative bound on the normal cones during the merging step. Stage (2) of our algorithm only needs to evaluate the sign of algebraic expressions corresponding to dot products and cross products, as well as other expressions; Stage (3) can be performed with geometric exact elementary test algorithms [BEB12, TTWM14], or conservative CCD algorithms [Wan14, WTTM15], and these computations can be accelerated using floating-point filters. As a result, the overall

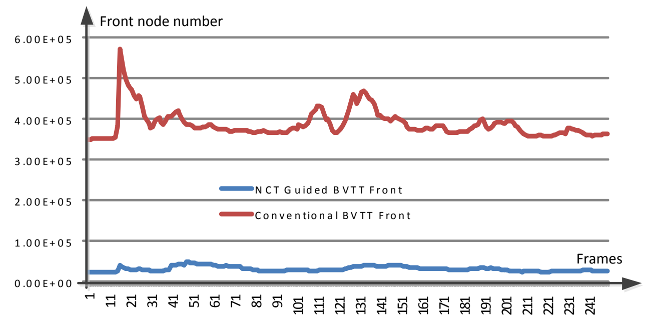


Figure 6: ***Front-based normal cone culling:*** *We highlight the size of our NCT guided BVTT front (the lower curve), which is about 8% of the conventional BVTT front in [Tang et al. 2011] (the upper curve) for the benchmark FlowingCloth. Our algorithm can save considerable memory and runtime overhead.*

collision detection algorithm is accurate and not susceptible to errors.

5. Implementation and Results

In this section, we describe our implementation and highlight the performance of our algorithm on several benchmarks.

5.1. Implementation

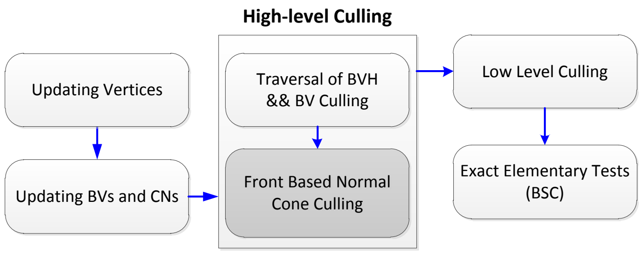
We have implemented our algorithms on a standard PC (Intel i7-3770K CPU @3.5GHz, 4GB RAM, 64-bits Window 7 OS) with C++ and all the results are generated using a single CPU core. The various components of our overall collision detection system are shown in Fig. 7.

Our current implementation is limited to deformable models without topology changes. We first construct a bounding volume hierarchy for the entire scene in a top-down manner [TCYM09]. During each time step, we update the bounding volumes and normal cones, and perform high level culling based on these data structures. After these high-level culling operations, we perform low-level culling operations that reduce the number of primitive tests using non penetration filters. Finally, we perform triangle-triangle intersection tests for DCD and exact elementary tests for CCD [TTWM14].

We use AABBs as the underlying bounding volume in the hierachy. It is possible to use tighter bounding volumes such as k-DOPs, but that increases the overhead of hierarchy update. Instead, we use non-penetration filters [TMT10a] along with AABBs and they provide similar culling efficiency as compared to k-DOPs or tight fitting bounding volumes.

5.2. Benchmarks

In order to evaluate the performance of our DCD and CCD algorithms, we used seven different benchmarks that came from different deformable simulation scenarios and have been used by other researchers.



High-level Culling
Updating Vertices
Updating BVs and CNs
Traversal of BVH && BV Culling
Front Based Normal Cone Culling
Low Level Culling
Exact Elementary Tests (BSC)

Figure 7: ***Collision Detection System:*** *For every time step, we update the bounding volumes and normal cones; perform high-level culling that includes bounding volume culling and front-based normal cone culling. We perform low-level culling to eliminate duplicate elementary tests (for CCD) and reliable primitive tests for each step.*

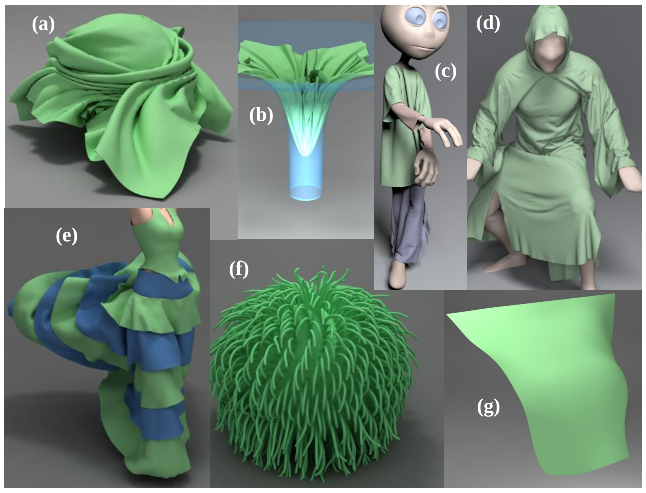


Figure 8: ***Benchmarks:*** *We use seven challenging benchmarks arising from deformable and cloth simulations. We compare the performance of our DCD and CCD algorithms with prior methods.*

- **Twisting:** It corresponds to complex cloth simulation with twists, as the ball rotates. It has 64K triangles. This benchmark has a high number of self-collisions (Figure 8(a)).
- **Funnel:** A cloth with 64K triangles falls into a funnel and folds to fit into the funnel with many self-collisions (Figure 8(b)).
- **Bishop:** A swing dancer wearing three pieces of cloth (with 124K triangles) with self-collisions (Figure 8(c)).
- **Falling:** A man wearing a robe (with 172K triangles) falls down rapidly, that introduces wrinkles and self-collisions in the cloth (Fig. 8 (d)).
- **Flamenco:** A fiery Flamenco dancer wearing a colorful skirt with ruffles. This benchmark (49K triangles) has a high number of self-collisions (Figure 8(e)).
- **SquishyBall:** A squishy ball with 820 tentacles and over 1M triangles squishes and bounces on the ground, inducing numerous small inter-penetrations [ZJ12]. (Figure 8(f)).

Bench-marks	CT Time (DCD)		CT Time (CCD)	
	Our	SCT	Our	CBC
Twisting	0.41	1.58	1.09	5.39
Funnel	0.63	2.29	1.17	7.56
Falling	1.4	8.07	17.8	55.6
Bishop	1.12	6.23	5.85	21.53
Flamenco	1.87	7.53	4.22	20.49
SquishyBall	141.1	196.5	454.3	668.5
FlowingCloth	0.033	0.14	0.8	9.37

Figure 9: ***Performance of Contour Tests:*** *We highlight the average running time (in ms) of our unprojected contour test algorithm by comparing with SCT [Schvartzman et al. 2010] and CBC [Tang et al. 2009]. The contour test can take a significant fraction of the overall collision query time shown in Fig. 10.*

Bench-marks	Qry Time (DCD)			Qry Time (CCD)				
				Prior Methods			Our Method	
	Our	AABB Only	SCT	AABB Only	CBC	DC	NC+ Front	NC Only
Twisting	56.79	80.08	60.15	105.55	95.56	96.45	82.6	92.19
Funnel	53.37	78.52	58.76	82.27	79.46	75.94	61.5	64.96
Falling	177.3	278.3	206.2	783.1	643.1	702.4	590.5	639.3
Bishop	75.24	109.23	90.93	124.98	111.9	119.32	97.76	109.5
Flamenco	83.1	102.29	95.4	272.18	267.4	252.31	210.6	215.5
SquishyBall	1842	2484	1921	6553	5749	5697	5142	5609
FlowingCloth	21.27	46.64	23.43	55.22	40.1	34.3	25.03	27.01

Figure 10: ***Performance and Comparison:*** *We compare the performance of our algorithm with prior techniques for DCD and CCD queries. We report the average time taken by these collision queries (in ms) for each benchmark. Each of these methods performs the same low-level culling operations and different high-level culling operations. For our method, we also highlight the relative benefit of combining BVTT front with normal cone culling (NC). We observe considerable speedups with (NC + Front) over prior CCD algorithms.*

- **FlowingCloth:** A flowing cloth with 50K triangles [ZJ12] (Figure 8(g)) is hanging with two corners fixed.

Four of these benchmarks (Twisting, Bishop, Falling, and Funnel) are generated using a cloth simulation system. The input for the Flamenco is given as discrete keyframes. We use the linearly interpolating motion of the vertices between successive key-frames and check for inter-object and self-collisions. Figure 10 highlights the performance of our algorithm for DCD and CCD queries on these benchmarks.

5.3. Cloth Simulation

We integrated our collision detection algorithm into a cloth simulation system. The underlying simulator performs implicit integration and use repulsion forces along with CCD computations to avoid interpenetrations. We compared the overall performance of the cloth simulation by using our collision detection algorithm vs. prior method that is based on AABB-only and performs no self-collision culling. We used the simulator to generate the entire simulations corresponding to these four benchmarks: Funnel, Twisting, Falling, and Bishop. Our faster CCD algorithm results in 1.2X speedup in the overall performance of the cloth simulator.

Bench -marks	Twisting	Funnel	Falling	Bishop	Flamenco	Squishy-Ball	Flowing-Cloth
DCD	34.48%	28.71%	25.5%	37.39%	26.23%	12.21%	8.01%
CCD	37.10%	31.40%	27.4%	39.50%	29.30%	11.40%	9.20%

Figure 11: ***Reduction in Front Size:*** *We highlight the reduction in the memory overhead of the BVTT front due to our novel traversal algorithm based on normal cones (see Section 4). Due to normal cone culling, the size of the BVTT front reduces by $8-37\%$ in our benchmarks.*

6. Comparison and Analysis

6.1. Comparison

We compare the performance of our algorithm with the prior techniques that use hierarchical methods or normal cones for DCD and CCD.

1. SCT: This corresponds to the implementation of Star-Contours based normal cone algorithm of [SPO10] for DCD. This is based on line-search star-shaped contour test and precomputing the self-collision test tree.

2. AABB-only: These algorithms use AABB as the underlying bounding volumes and perform no self-collision culling. They use low-level culling algorithms to eliminate duplicate elementary tests and perform reliable elementary tests using BSC [TTWM14].

3. CBC: This is the continuous normal cone algorithm for CCD [TCYM09] along with AABB culling and BSC elementary tests and performs the $O(n^2)$ exact contour test.

4. DC: This corresponds to the dynamic clustering algorithm [HOEM15] for CCD. It computes new clusters at each frame as the objects deform that is guaranteed to be collision free using observation point [WLH*13]. It is combined with the AABB hierarchy and BSC elementary tests.

5. NPF: The main contribution of the algorithm in [TMT10a] is the use of a non-penetration filter (NPF), which is a light-weight filter (or culling scheme) that can remove most of the false positives before performing exact elementary tests. The use of that filter resulted in 1.7x-3.5x speedups. In this paper, we also used the NPF to reduce the number of false positives (as mentioned at the end of Section 5.1). In other words, the self-collision culling algorithm presented in this paper is complementary to the use of NPF.

6. VolCCD and BSC: The main contribution of VolCCD [TMY*11] is in terms of low-level collision culling between volumetric elements; it is orthogonal to the use of normal cones. The main contribution of BSC [TTWM14] is related to performing reliable and exact elementary tests for CCD computations. The improved normal cone algorithm presented in this paper is orthogonal to these methods and can be combined with them. Some of the algebraic formulations used in our normal cone test is similar to that used in these prior papers, but the overall goal is different.

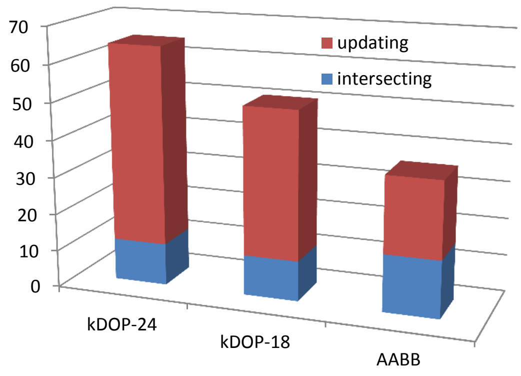


Figure 12: ***AABB vs. kDOP:*** *We compare the running times (in seconds) of the Funnel benchmark (64K triangles) by using 24-DOPs, 18-DOPs, and AABBs as bounding volumes, respectively. As shown by the figure, the time spent in the intersection computations is reduced with the use of tighter bounding volumes, but the updating time increases considerably.*

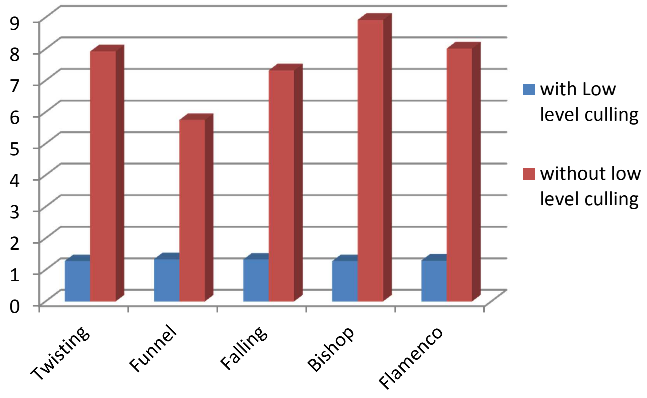


Figure 13: ***CCD accelerations with/without low-level culling:*** *We compare the acceleration rates for CCD between our method vs. AABB-only with/without performing low-level culling. We observe much higher speedups without low-level culling.*

We also compare the running times of the Funnel benchmark (64K triangles) by using 24-DOPs, 18-DOPs, and AABBs as bounding volumes, respectively. As shown in Fig. 12, the time spent in the intersection tests is reduced with tighter bounding volumes, the time to update the hierarchy (updating time) increases considerably more. Overall, the algorithms based on AABBs provide the best overall performance on the collision queries.

6.2. Analysis

As compared to the contour test presented in [SPO10], our algorithm is simpler and has lower overhead because we don't perform any projection computations. We observe up to 5.8X improvement in the performance of the contour test for DCD (Fig. 9) and up to 1.2X improvement (Figure 10) in the performance of overall DCD algorithm, as compared to [SPO10].

We observe considerable performance improvement for CCD computations, over techniques that either use no self-collisions (AABB-only) or exact continuous normal cone tests [TCYM09]. This is due to the fact that our normal cone test has linear complexity, whereas prior methods had quadratic complexity with a higher constant [TCYM09]. As a result, we observe up to 11.7X improvement in the performance of the contour test for CCD (Fig. 9). In terms of the overall CCD algorithm, the improved normal cone test (NC) results in up to 2.2X improvement over AABB-only. We obtain higher speedups for CCD as compared to DCD, because the normal cone tests take a larger fraction of the average frame time.

The combination of front-based culling with our improved normal cone test leads to considerable speedups. We observe additional performance improvement by combining front-based culling with normal cone (Front + NC) over using only normal cone (NC) in Figure 10. The front-based traversal reduces the cost of hierarchy traversal and the number of self-collision tests. It doesn't reduce the number of false positives in terms of elementary tests. As a result, the relative benefits of normal cone tests and front-based culling are somewhat complementary and we obtain high speedups by combining them. Fig. 1 highlights the improved culling with unprojected normal cone tests. Furthermore, we observe considerable improvements in the size and memory overhead of the BVTT front, when it is combined with normal cone culling (see Fig. 11).

In all the implementations (SCT, AABB-only, CBC, and DC), we used low-level culling methods (orphan sets [TCYM09] and non-penetration filters [TMT10a]) for CCD to remove redundant elementary tests. So the overall performance improvement in CCD query is moderate. If we do not perform low-level culling and only compare the relative performance with BVH culling only, we observe significantly higher speedups, as shown in Fig. 13. Many other approaches only seem to compare the relative speedups with only BVH-culling [SPO10,ZJ12]. Basically, these low-level culling methods can significantly reduce the number of exact elementary tests being performed between the primitives. The fastest CCD algorithms use a combination of low-level and high-level culling schemes, as shown in Fig. 10.

7. Limitations, Conclusions and Future Work

We present a fast and reliable algorithm to perform self-collision culling on complex deformable models. There is a general perception that the overhead of normal cone tests is high and its applications has mostly been limited to DCD. We presented a novel unprojected contour test, that provides $10-30X$ improvement over prior continuous contour tests for CCD. Furthermore, we described a novel traversal scheme using front-based normal cone culling that reduces the time and space overhead. The combination of these two methods can accelerate the performance of CCD queries by an order of magnitude.

Our approach has some limitations. The unprojected contour test tends to be more conservative than prior methods. The current formulation is limited to linearly interpolating triangles for CCD. Self-collision culling based on normal cones works well when the resulting meshes do not have high variations in curvature. As a result, their performance depends on how a large mesh is decomposed into sub-meshes at different nodes of the BVH or the underlying clustering criteria. Our current implementation is limited to deformable models that do not undergo topology changes.

There are many avenues for future work. It would be useful to combine our approach with fast, dynamic clustering schemes that can improve the culling efficiency of normal cones and also used for adaptive meshes. We would like to parallelize the approach on multi-core CPUs and GPUs, similar to [GLM05, SGG*06, TMLT11, ZK14, TWT*16], and our reduced-size BVTT front should improve the parallel performance. Finally, we would like to integrate our algorithm with FEM and hair simulation systems.

Acknowledgements: This research is supported in part by NSFC (61572423, 61572424), Zhejiang Provincial NSFC (LZ16F020003), and the Doctoral Fund of Ministry of Education of China (20130101110133).

References

[BEB12] BROCHU T., EDWARDS E., BRIDSON R.: Efficient geometrically exact continuous collision detection. *ACM Trans. Graph. 31*, 4 (July 2012), 96:1–96:7. 1, 2, 7

[BFA02] BRIDSON R., FEDKIW R., ANDERSON J.: Robust treatment of collisions, contact and friction for cloth animation. *ACM Trans. Graph. 21*, 3 (July 2002), 594–603. 1, 2

[BJ10] BARBIČ J., JAMES D. L.: Subspace self-collision culling. *ACM Trans. on Graphics (SIGGRAPH 2010) 29*, 4 (2010), 81:1–81:9. 1

[CTM08] CURTIS S., TAMSTORF R., MANOCHA D.: Fast collision detection for deformable models using representative-triangles. In *SI3D '08: Proceedings of the 2008 Symposium on Interactive 3D graphics and games* (2008), pp. 61–69. 2

[EL01] EHMANN S. A., LIN M. C.: Accurate and fast proximity queries between polyhedra using convex surface decomposition. In *Computer Graphics Forum* (2001), pp. 500–510. 2

[GKJ*05] GOVINDARAJU N. K., KNOTT D., JAIN N., KABUL I., TAMSTORF R., GAYLE R., LIN M. C., MANOCHA D.: Interactive collision detection between deformable models using chromatic decomposition. *ACM Trans. Graph. 24*, 3 (July 2005), 991–999. 2

[GLM05] GOVINDARAJU N. K., LIN M. C., MANOCHA D.: Quick-cullide: Fast inter-and intra-object collision culling using graphics hardware. In *Virtual Reality, Proceedings. IEEE VR* (2005), pp. 59–66. 11

[HF07] HUTTER M., FUHRMANN A.: Optimized continuous collision detection for deformable triangle meshes. In *Proc. WSCG '07* (2007), pp. 25–32. 2

[HOEM15] HE L., ORTIZ R., ENQUOBAHRIE A., MANOCHA D.: Interactive continuous collision detection for topology changing models using dynamic clustering. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games* (2015), i3D '15, pp. 47–54. 2, 9

[HSK*10] HEO J.-P., SEONG J.-K., KIM D., OTADUY M. A., HONG J.-M., TANG M., YOON S.-E.: FASTCD: Fracturing-aware stable collision detection. In *Proceedings of the 2010 ACM SIGGRAPH / Eurographics Symposium on Computer Animation* (2010), pp. 149–158. 3

[LC98] LI T.-Y., CHEN J.-S.: Incremental 3D collision detection with hierarchical data structures. In *Proceedings of the ACM VRST* (1998), VRST '98, pp. 139–144. 2

[MKE03] MEZGER J., KIMMERLE S., ETZMUβ O.: Hierarchical techniques in cloth detection for cloth animation. *Journal of WSCG 11*, 1 (2003), 322–329. 2

[Pro97] PROVOT X.: Collision and self-collision handling in cloth model dedicated to design garments. In *Graphics Interface* (1997), pp. 177–189. 1, 2, 3

[SGG*06] SUD A., GOVINDARAJU N., GAYLE R., KABUL I., MANOCHA D.: Fast proximity computation among deformable models using discrete voronoi diagrams. *ACM Trans. Graph. 25*, 3 (July 2006), 1144–1153. 11

[SPO10] SCHVARTZMAN S. C., PÉREZ A. G., OTADUY M. A.: Star-contours for efficient hierarchical self-collision detection. *ACM Trans. Graph. 29*, 4 (July 2010), 80:1–80:8. 1, 2, 3, 9, 10

[TCYM09] TANG M., CURTIS S., YOON S.-E., MANOCHA D.: ICCD: interactive continuous collision detection between deformable models using connectivity-based culling. *IEEE Transactions on Visualization and Computer Graphics 15* (2009), 544–557. 1, 2, 3, 8, 9, 10

[TMLT11] TANG M., MANOCHA D., LIN J., TONG R.: Collision-streams: Fast GPU-based collision detection for deformable models. In *Proceedings of the 2011 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* (2011), pp. 63–70. 2, 7, 11

[TMT10a] TANG M., MANOCHA D., TONG R.: Fast continuous collision detection using deforming non-penetration filters. In *Proceedings of ACM Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), ACM, pp. 7–13. 2, 8, 9, 10

[TMT10b] TANG M., MANOCHA D., TONG R.: MCCD: Multi-core collision detection between deformable models using front-based decomposition. *Graphical Models 72*, 2 (2010), 7–23. 7

[TMY*11] TANG M., MANOCHA D., YOON S.-E., DU P., HEO J.-P., TONG R.: VolCCD: Fast continuous collision culling between deforming volume meshes. *ACM Trans. Graph. 30* (May 2011), 111:1–111:15. 2, 9

[TTWM14] TANG M., TONG R., WANG Z., MANOCHA D.: Fast and exact continuous collision detection with bernstein sign classification. *ACM Trans. Graph. 33* (November 2014), 186:1–186:8. 1, 2, 7, 8, 9

[TWT*16] TANG M., WANG H., TANG L., TONG R., MANOCHA D.: 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. 11

[VT94] VOLINO P., THALMANN N. M.: Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. *Computer Graphics Forum 13*, 3 (1994), 155–166. 1, 2, 3

[Wan14] WANG H.: Defending continuous collision detection against errors. *ACM Trans. Graph. 33*, 4 (July 2014), 122:1–122:10. 1, 2, 7

[WB06] WONG W. S.-K., BACIU G.: A randomized marking scheme for continuous collision detection in simulation of deformable surfaces. *Proc. of ACM VRCIA* (2006), 181–188. 2

[WB14] WONG S.-K., BACIU G.: Continuous collision detection for deformable objects using permissible clusters. *The Visual Computer* (2014), 1–13. 2

[WC14] WONG S.-K., CHENG Y.-C.: Continuous self-collision detection for deformable surfaces interacting with solid models. *Computer Graphics Forum 33*, 6 (2014), 143–153. 2

[WLH*13] WONG S.-K., LIN W.-C., HUNG C.-H., HUANG Y.-J., LII S.-Y.: Radial view based culling for continuous self-collision detection of skeletal models. *ACM Transactions on Graphics (ACM SIGGRAPH) 32*, 4 (2013). 1, 2, 9

[WTTM15] WANG Z., TANG M., TONG R., MANOCHA D.: TightCCD: Efficient and robust continuous collision detection using tight error bounds. *Computer Graphics Forum 34* (2015), 289–298. 2, 7

[ZJ12] ZHENG C., JAMES D. L.: 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. 1, 2, 8, 9, 10

[ZK14] ZHANG X., KIM Y.: Scalable collision detection using p-partition fronts on many-core processors. *Visualization and Computer Graphics, IEEE Transactions on 20*, 3 (March 2014), 447–456. 2, 11