EUROGRAPHICS 2022 / R. Chaine and M. H. Kim
(Guest Editors)

Volume 41 (2022), Number 2

Fast and Exact Root Parity for Continuous Collision Detection

Bolun Wang[1], Zachary Ferguson[2], Xin Jiang[1,3], Marco Attene[4], Daniele Panozzo[2], and Teseo Schneider[5]

[1] LMIB & School of Mathematical Sciences & Shenyuan Honor College, Beihang University
[2] New York University
[3] Zhengzhou Aerotropolis Institute of Artificial Intelligence
[4] IMATI - CNR
[5] University of Victoria

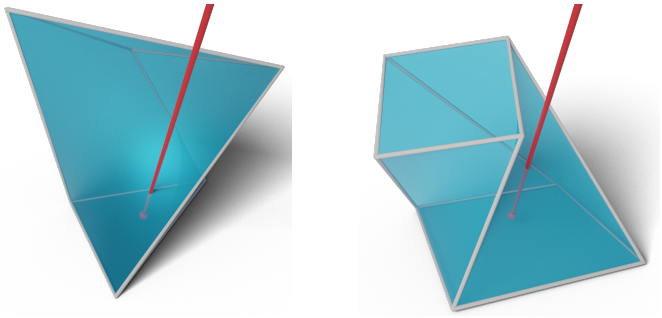

Figure 1: *The parity of the roots between a moving triangle and a point (left) and between two moving edges (right), can be computed by counting the parity of the intersections of a ray (red) casted from the origin (red sphere) with the surface of a prism for the vertex-triangle case (left) or a cube for the edge-edge case (right).*

Abstract

We introduce the first exact *root parity counter for continuous collision detection (CCD). That is, our algorithm computes the parity (even or odd) of the number of roots of the cubic polynomial arising from a CCD query. We note that the parity is unable to differentiate between zero (no collisions) and the rare case of two roots (collisions).*
Our method does not have numerical parameters to tune, has a performance comparable to efficient approximate algorithms, and is exact. We test our approach on a large collection of synthetic tests and real simulations, and we demonstrate that it can be easily integrated into existing simulators.

CCS Concepts
• ***Computing methodologies*** → ***Collision detection;*** • ***Mathematics of computing*** → *Mathematical software;*

1. Introduction

Continuous Collision Detection (CCD) between two objects is a fundamental tool to guarantee that the two moving objects will not pass through each other (tunneling). Among several object representations (e.g., spline patches, point clouds, etc.) and trajectories, we focus on the popular case of triangle meshes moving along piecewise linear trajectories. Using this assumption, CCD can be reduced to CCD between two triangles. Since two triangles moving along linear trajectories will either intersect when a vertex touches a face (including its boundary) or when an edge crosses another edge, CCD can be further simplified to vertex-triangle detection and edge-edge collision detection.

With the aforementioned assumptions (triangular meshes with vertices moving on linear trajectories), CCD queries can be solved by finding roots of low-degree cubic polynomials. Theoretically, any collision can be detected if the root-finding is accurate. However, under a computer's floating-point calculation, rounding errors often cause inaccurate results, which may cause false positives

(meaning a collision is reported when there is no collision) or false negatives (meaning a collision is not reported when there is a collision).

Many methods [BEB12, TTWM14] attempt to *exactly* find the roots of the low-degree polynomial to detect collisions; however, none succeeds [WFS*21]. Among all existing *ostensibly exact* methods, [BEB12] reformulates the parity of the CCD roots as a geometric problem: the CCD roots have the same parity as the parity of the number of intersections between a ray and bilinear surfaces (Figure 1). Thus, it constructs the co-domain (which are polyhedrons, prisms and hexahedra, with bilinear faces) of the collision equations, casts a ray from the origin, and counts the parity of the intersections. As with many geometric approaches, the main challenge of this approach is to account both for inexact floating-point representations and enumerate all possible degenerate configurations. According to our study, [BEB12] misses important corner cases and relies on a complex arithmetic expansion to account for rounding errors.

The only downside of our method, as noted by [BEB12], is that it cannot distinguish between zero and two roots. This is practically a minor issue as these cases only occur in very contrived configurations. To construct such examples, we considered very large time steps (which have only become practical with the IPC simulator [LFS*20]); in our dataset this occurs only in seven cases out of 60 million (Table 1). Additionally, as noted in [BEB12], this is not an issue for closed meshes [BEB12, Section 3].

Our work is inspired by [BEB12] with a runtime comparable to the original method. Our method sidesteps the arithmetic expansion by using a numerical shifting method and detects and processes all the degenerate cases. Our approach requires rounding *only* the input scene (and no other subsequent modifications to the simulations is required) based on its bounding-box and guarantees that all subsequent time-steps are *exact* and do not require any rounding. Fortunately, the rounding error is on the order of 10^{-15}, Figure 5; thus its impact on the simulation is practically none.

In this paper, we generalize the construction in [BEB12] by:

- **Floating-Point Construction.** We introduce a shifting procedure that allows constructing the polyhedron using standard floating-point numbers. This reduces the computational cost and, at the same time, simplifies the implementation.
- **Degeneracies.** We enumerate all the degeneracies of the bilinear faces (such as flips or collapse to a line or point) and robustly handle them, avoiding tunneling failures of [BEB12].
- **Ray-Bilinear Face Intersection Parity.** We provide a floating-point arithmetic-based algorithm to determine the parity of ray-bilinear face intersections. In our algorithm, we can not only determine the parity of intersections between a ray and a non-degenerate bilinear face but can also detect the intersection between a ray and a degenerate bilinear face using a very simple XOR detector. The ability to process degenerate cases makes our root-parity counting algorithm robust.

Overall, our algorithm using an exact and robust root parity counting method has a runtime comparable to the most efficient inaccurate method published for CCD, while returning no false positives when counting the parity.

We test our algorithm on the large-scale benchmark provided by [WFS*21] to prove its efficiency, stability, and accuracy in Section 5. The comparison results also show that our algorithm is able to handle challenging inputs not supported by the original algorithm provided by [BEB12]. We provide our reference implementation with scripts to reproduce all our results (Section 5), and our framework to foster quick adoption of our technique. Finally, we describe the minimal changes required in solvers to use our CCD algorithm (Section 6) and demonstrate its performance when integrated into popular collision response algorithms. We note that, similarly to [BEB12], our algorithm is only able to count the parity of the roots and might miss collisions when there are two roots. These cases are rare (7 cases out of 60 million queries) in applications. Additionally, double roots can be reduced by using a smaller simulation time-step. On the positive side, our method is, to the best of our knowledge, the only one that has a reasonable runtime (Section 5) and can *exactly* count the parity of the CCD roots.

2. Related Work

Here we present a short overview of continuous collision detection methods for deformable triangle meshes with linear vertex-trajectories. We refer to [WFS*21] for a complete overview.

CCD problems can be represented as finding roots of special low order polynomials. Thus the majority of research focuses on developing efficient and accurate cubic polynomial solvers. [Pro97] introduced the most popular method which first solves a cubic equation to check coplanarity, then checks for overlap. If there is no root to coplanarity, then there is no collision; otherwise overlapping will be validated to determine if a collision occurs. Based on this idea, refined constructions have been introduced to solve rigid body collision [RKC02, KR03] and deformable body collision [HF07, TMY*11]. All of these methods are based on floating-point arithmetic, which is efficient but introduces numerical errors. In fact, the roots of the cubic polynomials are in general irrational numbers thus cannot be represented as floating-point results exactly. However, even if numerical thresholds are applied to account for the rounding errors, these root finders only guarantee to be robust for some specific scenarios and still suffer from false positives.

False positives can be regarded as adding an extra numerical padding layer above the objects during simulation that is not related to any physical quantity. [TMT10] propose a filter that can conservatively detect roots and effectively reduce the number of elementary tests and false positives. [Wan14] and [WTTM15] improve the reliability of algorithms by introducing forward error analysis, in which error bounds for floating-point computation are used to reduce false positives.

Inclusion-based root-finding algorithms are another family of conservative CCD methods [Sny92, SWF*93, RKC02, VHBZ90, WFS*21]. Snyder [Sny92] applied interval arithmetic to computer graphics for collision finding. Recently, [WFS*21] introduced an algorithm totally depending on floating-point arithmetic to accelerate the hierarchical root-finding algorithm. However, while providing no false negatives, these algorithms are not exact, often producing false positives.

Relying on exact arithmetic, [BEB12] and [TTWM14] provide

exact continuous collision detection methods. However, the two algorithms cannot always provide exact answers, according to the careful research of [WFS*21]. [BEB12] introduces a root-parity counting algorithm to detect collisions, thus sidestepping the actual computation of the roots. The algorithm constructs the co-domain of the multi-variate equation of the CCD query as a polyhedron, and computes the parity of the roots, by casting a ray from the origin and counting the parity of the number of intersections between the ray and the surface of the co-domain. If rational numbers or arithmetic expansion is used, while making the algorithm much slower in performance, polyhedron construction, ray casting, and intersection can be done exactly. However, as shown in [WFS*21], the algorithm is not robust to degenerate configurations.

The linear assumption, as we described in Section 1, meaning all vertices of the meshes move in linear trajectories during each time step, is well established and commonly used. However, rigid motions [TKM09, RKC02, Can86, ZRLK07] and polynomial trajectories [PZM12] also require corresponding CCD algorithms to avoid approximation errors caused by linearization.

3. Preliminaries and Notation

We briefly overview continuous collision detection, and in particular, the geometrical interpretation introduced in [BEB12] to make our paper self-contained.

The goal of exact CCD is to provide an exact predicate in floating-point coordinates telling us if two objects, whose vertices move in a *linear trajectory*, collide. This is a common task during simulation or animation of moving objects, to detect if and when they come in contact and how they deform. Assuming that the objects are represented using triangular meshes, the first collision between moving triangles can happen either when a vertex hits a triangle, or when an edge hits another edge. We will focus on the former for simplicity as the latter is a minor variant.

Given a point p and a triangle with vertices v_1, v_2, v_3 at two distinct timesteps t^0 and t^1 (we use the superscript notation to denote the time, i.e., p^0 is the position of p at t^0), the goal is to determine if at any point in time between t^0 and t^1 the point is contained in the moving triangle. [BEB12] reduces this problem to a geometric intersection parity counting problem by introducing the map

$$F(t,u,v) = p(t) - \big((1-u-v)v_1(t) + uv_2(t) + vv_3(t)\big), \qquad (1)$$

which maps the domain $\Omega = [0,1] \times \{u,v \geq 0 | u+v \leq 1\}$ to a triangular prism P with bilinear faces. Equipped with this definition, [BEB12] showed that the roots of (1) have the same parity as the number of intersections between P and a ray cast from the origin. The edge-edge case leads to a formula similar to Equation (1), but maps the domain to a hexahedron bounded by 6 bilinear faces instead of a prism.

4. Method

To solve vertex-triangle CCD, [BEB12] represents the root parity problem of Equation (1) as an intersection-parity counting algorithm: First, the co-domain of Equation (1) is presented as a prism P in $\mathbb{R}^3$ by constructing its vertices. While observing that the faces of the prism are bilinear surfaces, the algorithm casts a random ray R from origin $O = (0,0,0)$ to intersect the bilinear faces (Figure 1, left). In the end, if the number of intersections between the ray and the bilinear faces is odd, then the algorithm returns true (as (1) has an odd number of roots), which means collision happens; otherwise, the algorithm returns false, which means there are either two collisions or zero collisions in this time step. The edge-edge CCD follows the same algorithm, other than constructing a hexahedron with six bilinear faces instead of a prism with five bilinear faces (Figure 1, right). We note that, since we are only able to determine the parity of the number of root, our algorithm is unable to distinguish between two and zero roots, thus producing false negatives. As pointed out by [BEB12], false negatives are rare (only 7 cases on our simulation dataset) and can be reduced by shrinking the time-step in the simulations.

Besides the inability to distinguish between zero and two roots, [BEB12] is not exact for two main reasons: (1) The vertices of the prism P need to be computed *exactly* (which can only be done using rational number arithmetic) and (2) several degenerate cases are not properly handled.

Following the idea of [BEB12], we provide an exact and efficient root-parity counter (i.e., we do not rely on rational computations) for continuous collision detection. A key property of our method is that the results of our predicates are not affected by rounding errors in the floating-point computation, a necessary property required by the simulation algorithms that assume that the scene is intersection-free at every step. To achieve this goal, our final answer should not rely on intermediate floating-point calculations which might be rounded, and hence inexact. We thus rely only on exact input values, and base our final answer on exact predicates computed from these values. We note that, to avoid the complexity of dealing with floating-point predicate construction, one could consider the easy solution of building the ray and bilinear faces of $F(\Omega)$ explicitly and checking their intersection using exact computation. This solution requires implementing the whole algorithm using rational arithmetic, which is simple, but can be impractically slow.

4.1. Algorithm Overview

The input to our algorithm is the generalized prism $P = F(\Omega)$ and a ray R cast from the origin $O = (0,0,0)$, and our algorithm returns a Boolean indicating if there are an odd or even number of intersections between R and the surfaces of P which is equivalent to the parity of the roots of Equation (1) [BEB12].

Constructing the corners of P requires evaluating Equation (1) at the corners of Ω, which may introduce numerical rounding errors. We show in Section 4.2 how to compute them exactly using only floating-point operations by a shifting algorithm. The entire boundary of P is then explicitly parametrized by bilinear patches, whose equations depend only on the corner vertices, which are exactly represented in floating-point with our method.

To count the parity of intersections between R and the bilinear surfaces b of P, we enumerate four possible relative configurations between R and a bilinear face b (Figure 2):

1. O is exactly on b, Figure 2 (1);

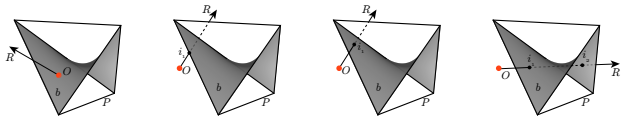

Figure 2: *Illustration of the four possible relative configurations between a ray R cast from the origin O and a bilinear surface b of a generalized prism P; i_1 and i_2 are the intersection points between R and b.*

2. O is not on b, and R intersects the boundary of b, Figure 2 (2);
3. O is not on b, and R intersects b in the interior of b once, Figure 2 (3);
4. b and R have zero or two intersections, Figure 2 (4).

[WFS*21] shows that the axis aligned bounding box BB of the vertices of P is the tightest axis aligned bounding box of P. So, our algorithm 1 first checks if O intersects BB (line 5), as a quick culling method for acceleration: if O and BB do not intersect, the parity is even, then there is no collision, and our algorithm returns NO_COLLISION.

The algorithm then checks the intersection between R and b, while dealing with all possible degenerate cases. If the case falls into Case 1 (line 12), then the algorithm directly return COLLISION since Case 1 means the parity is odd. Case 2 is extremely difficult since intersection is degenerate and the parity is impossible to detect. For instance R might be collinear with an edge of b and thus generate infinite intersections. Fortunately this case is extremely rare; Thus if it happens, as suggested by [BEB12], we cast another ray in a random direction and retry (line 15).

If Case 3 or Case 4 happens we first check if the patch is degenerate (line 19); If it is, we identify the case and use a simple XOR to count the number of intersections (line 21). In the case of a non-degenerate patch, we count the intersections between R and b (line 23). We repeat this process for every bilinear face until all the bilinear faces are checked (line 11). After counting the number of roots for every face, we return the parity of the number of intersections (line 26).

The key challenge is distinguishing between Case 3 and Case 4: counting the intersections between a ray R and a prism face b (which can be either a triangle, a bilinear patch, or degenerated). Counting the intersections can be done as explained in [BEB12] or using standard predicates (Section 4.3). However, computing the coordinates of the corners of the prism P using standard floating-points arithmetic introduces potential floating-point rounding errors. Additionally, several degenerate configurations might occur (e.g., the bilinear quad becomes flat and concave) that need to be carefully enumerated and handled (Section 4.4). All these problems (i.e., numerical and geometrical) may cause the failure of the algorithm. Thanks to the rounding (Section 4.2) and handling of degenerate cases (Section 4.4), our method can use standard orientation predicates (since the vertices are *exact* in floating-point numbers) thus making it a robust and efficient root-parity counter.

Algorithm 1 Exact CCD

```
1:  function CCD(p^0, v_1^0, v_2^0, v_3^0, p^1, v_1^1, v_2^1, v_3^1)
2:      P ← PRISM(p^0, v_1^0, v_2^0, v_3^0, p^1, v_1^1, v_2^1, v_3^1)      ▷ Equation (2)
3:      BB ← PRISMBOUNDINGBOX(P)
4:
5:      if O ∩ BB = ∅ then ▷ Check origin and prism bounding-box
6:          return NO_COLLISION
7:      end if
8:
9:      C ← 0
10:     R ← RANDOMRAY
11:     for b ∈ P do                                   ▷ Loop over the faces of P
12:         if O ∩ b ≠ ∅ then
13:             return COLLISION
14:         end if
15:         if R intersects b on the boundary then
16:             Cast another random ray and retry, line 10.
17:         end if
18:
19:         if ISDEGENERATE(b) then
20:             T ← DEGENERATETYPE(b)
21:             C ← C+ XOR(b, T)                       ▷ Section 4.4
22:         else
23:             C ← C+ RAYBILINEARCHECK(b, R)          ▷ Section 4.3
24:         end if
25:     end for
26:     if C% 2 = 0 then
27:         return COLLISION
28:     else
29:         return NO_COLLISION
30:     end if
31: end function
```

4.2. Reduced Precision

The computation of the corners of the prism $P = F(\Omega)$ introduces potential floating-point rounding errors. In [BEB12], this is tackled using a custom construction that uses higher precision to represent these points. We opt for a simpler and more efficient approach, by rounding the input vertices to ensure that evaluating F on the corners will not introduce errors. This construction adds a minor hurdle when integrating our method in existing solvers, since it has to be performed on the solver side. The input to our algorithm needs to be already rounded and we cannot handle inputs with full precision. In Section 6 we discuss in more detail how to achieve this with only a minor modification, and no perceptible impact on the simulation.

We need to evaluate Equation (1) at 6 corners, resulting in the following expressions

$$\begin{aligned} \tilde{v}_1 &= p^0 - v_1^0, & \tilde{v}_2 &= p^0 - v_2^0, & \tilde{v}_3 &= p^0 - v_3^0, \\ \tilde{v}_4 &= p^1 - v_1^1, & \tilde{v}_5 &= p^1 - v_2^1, & \tilde{v}_6 &= p^1 - v_3^1. \end{aligned} \tag{2}$$

All these expressions are similar and involve a single difference of the input coordinates.

To make the evaluation of the difference exact, we round the

input vertex coordinates so as to make them lose a bit of precision [Ste74]. This precision loss is in the order of 10^{-15} on average (Figure 5) and it is thus comparable to the rounding errors that are introduced by the numerical solvers used in simulators. We perform this rounding by shifting the whole mesh away from the origin by a certain amount. The shifting amount is calculated by exploiting the so-called Sterbenz theorem [Ste74].

Let a and b be two floating-point values. Sterbenz's theorem states that their exact difference $a-b$ is representable as a floating-point number if $1/2 \leq a/b \leq 2$ (Sterbenz's condition). According to the IEEE-754 standard, the result of a floating-point operation is the rounding of the exact result. Hence, if the exact result is representable, the floating-point result must coincide with such an exact result. Stated differently, if Sterbenz's condition holds then $a-b = \text{round}(a-b)$.

If Sterbenz's condition does not hold, the question is how to round a and b so that their difference is exact. We observe that (1) if either operand is zero the difference is exact and (2) if $a = b$ the difference is exact (Sterbenz condition holds). If none of these conditions hold, we need to find a value e such that:

$$a' = a+e, \quad b' = b+e, \quad \text{and} \quad 1/2 \leq a'/b' \leq 2.$$

If $e = b-2a$ we have

$$a'/b' = (a+e)/(b+e) = (b-a)/(2b-2a) = 1/2.$$

Similarly, if $e = a-2b$ we have

$$a'/b' = (a+e)/(b+e) = (2a-2b)/(a-b) = 2.$$

Therefore, any of the two options for e is appropriate. Since we must coherently shift a whole set of values using a unique displacement e, we conservatively select the maximum over the two possibilities. Hence, for a set of pairs (a_i, b_i) we calculate a corresponding set of displacements $e_i = \max((b_i - 2a_i), (a_i - 2b_i))$ and pick the maximum. Note that this makes sense because the violation of Sterbenz's condition implies that $e_i > 0$ independently of the sign of a_i and b_i: this can be easily proved by analyzing each of the six possible cases (four sign combinations with $a/b < 1/2$, plus another two with $a/b > 2$). The value of e and all the displaced values can be easily calculated using floating-point arithmetic if the rounding mode is set to `plus_infinity`.

To avoid having a different rounding over time, we compute the bounding box of the scene, then compute a *global* shift e for the x, y, and z-component based on the eight vertices of the bounding box.

4.3. Ray Shooting

To enable early termination of our algorithm, we rely on ray shooting as in [BEB12]. Note that we handle all the degenerate cases as explained in Section 4.4 and, if the ray hits a vertex or an edge we discard it and retry. Differently from [BEB12] we avoid using intervals and expansions and rely on an efficient `ray_plane` predicate (Appendix A).

The ray-triangle predicate checks if a ray R defined by an oriented pair of points (s_1, s_2) intersects a triangle $T = (t_1, t_2, t_3)$. This is similar to the line-triangle intersection (Appendix A). The only difference is that we need to check if the three $v_i, i = 1,2,3$ have the same sign as $o_1 = \texttt{orient3d}(s_1, t_1, t_2, t_3)$ to ensure that the ray is pointing towards the plane spanned by T and not in the opposite direction.

4.4. Degenerate Cases

In the previous sections we compute intersections between rays and a bilinear quad, where special care is needed since the bilinear patch can degenerate. We enumerate these cases and discuss how to handle them in a unified manner.

We call a bilinear quad degenerate when the volume of the tetrahedron T is zero, that is when the four points are coplanar or collinear. For this specific case we define the plane D spanned by the four points. Figure 3 illustrates and enumerates all such possible degenerate configurations, ranging from a simple regular quad, passing through a "butterfly" configuration, to a line or a point.

Let us enumerate the four vertices of the quad with numbers from 1 to 4. With this enumeration we can decompose the quad in two triangles having distinct configurations: either connect 1 with 3, or 2 with 4. We call the two configurations c_{13} and c_{24}. To differentiate the degenerate cases, we pick a random point p not coplanar with D and construct two pairs of tetrahedra (T^1_{13}, T^2_{13} and T^1_{24}, T^2_{24}) by connecting the two pairs of triangles with p. Each specific degenerate case can then be uniquely identified by the four signs of the volumes of the two pairs, which we compute using `orient3d` [Lév19, She97, Att20]. We note that the position of p with respect to D is not relevant as we compare the signs. That is, by selecting a point on the other side of D all signs will flip but their relative sign will not change. Figure 3 illustrates all possible combinations of signs and which degenerate cases they form. For instance, if the four volumes are positive (or all negative) we have a regular quad or if only one volume is negative (or only one is positive) we have a concave quad.

When all four volumes are positive or zero we can simply check if R intersects b by using the pair generated by c_{13} and check for two `ray_triangle` intersections. When one of the volumes is negative, we select the configuration containing it, check if R intersects the two triangles using `ray_triangle`, and use a XOR to decide if R intersects b (Figure 4).

5. Results

Our algorithm is implemented in C++ and uses Eigen [GJ*10] for the linear algebra routines and [She97] for the standard orientation predicate. We run our experiments on a 2.35 GHz AMD EPYC™ 7452. We attach the reference implementation and the data used for our experiments and we will release it as an open-source project.

5.1. Datasets

To compare our method with existing CCD algorithms we tested our algorithm using a large scale dataset provided in [WFS*21]. The dataset contains a *handcrafted* dataset that contains over 12 thousand point-triangle and 15 thousand edge-edge queries, and a

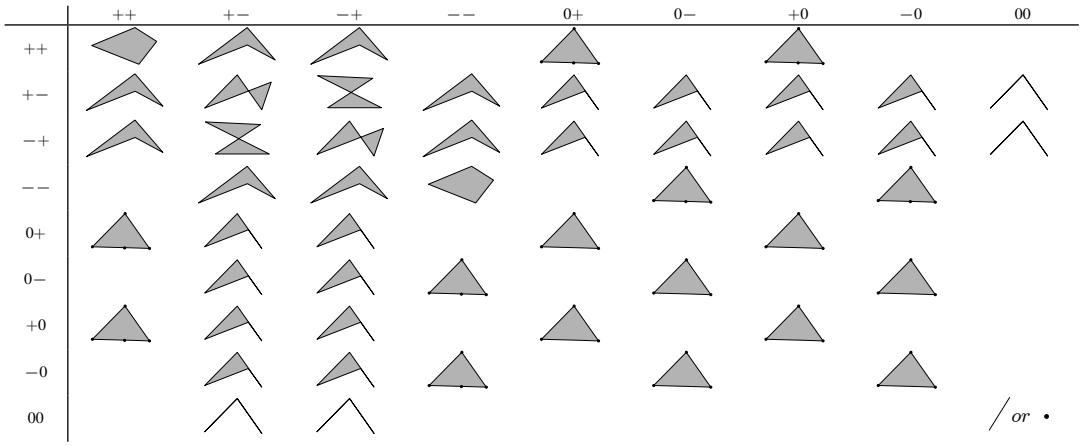

Figure 3: *Enumeration of the possible degenerate cases. All of them can be decided by the two signs of T_{13} (columns) and T_{24} (rows).*

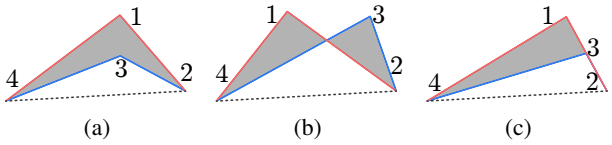

Figure 4: *Illustration of how a simple XOR between the positive and negative configuration can decide the three cases (up to symmetry) where one of the volumes is negative and the configuration c_{24} is selected. For each of them, the grey area depicts the interior of the degenerate bilinear patch b bounded by the blue and red edges. If R intersects both triangles t_{123} and t_{134}, or none, then the ray does not intersect b. If R intersects only one of triangles, then the ray intersects b. This operation can be summarized with: R needs to intersect one XOR the other triangle.*

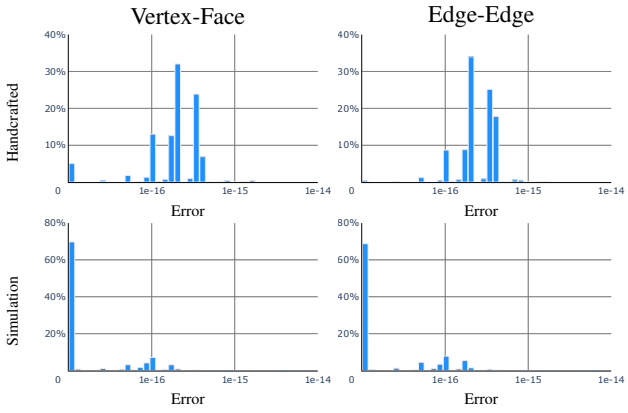

Figure 5: *Histograms of ε for our method.*

Handcrafted Dataset (12K) – Vertex-Face CCD

	IRF	UIRF	FPRF	TCCD	RP	RRP	BSC	MSRF	TI	Ours
T	14674.60	134494.00	3.12	0.40	2.82	1042.59	113.62	9.47	1605.72	1.07
TP	46748.97	220967.74	3.07	1.06	16.02	882.80	96.27	2.77	28004.70	4.72
TN	12784.93	129119.67	3.12	0.31	2.16	1050.59	114.52	9.66	94.06	0.88
FP	92	127	6	902	0	0	42	10	74	0
FN	0	0	98	0	5	5	26	254	0	5

Handcrafted Dataset (15K)– Edge-Edge CCD

	IRF	UIRF	FPRF	TCCD	RP	RRP	BSC	MSRF	TI	Ours
T	11116.80	16294.00	0.68	0.34	2.53	1362.76	79.88	2.69	3110.59	3.29
TP	95241.05	203781.84	4.65	1.15	36.90	1099.73	990.90	1.19	77870.26	10.82
TN	7820.85	7767.89	0.60	0.29	1.46	1370.81	48.92	2.71	78.58	3.06
FP	118	209	6	370	4	0	128	14	138	0
FN	0	0	170	0	8	8	86	292	0	8

Simulation Dataset (18M) – Vertex-Face CCD

	IRF	UIRF	FPRF	TCCD	RP	RRP	BSC	MSRF	TI	Ours
T	96.83	4872.95	9.85	0.29	0.43	1053.85	34.38	64.98	0.79	0.31
TP	949.77	115.92	1.25	1.16	8.35	518.91	264.56	1.07	63.70	2.95
TN	95.94	4877.90	9.86	0.28	0.43	1054.41	33.84	65.00	0.73	0.31
FP	2	18	0	96102	0	0	23013	0	2	0
FN	0	0	5103	0	0	0	0	15007	0	0

Simulation Dataset (41M) – Edge-Edge CCD

	IRF	UIRF	FPRF	TCCD	RP	RRP	BSC	MSRF	TI	Ours
T	146.58	648.50	0.37	0.27	0.62	1240.22	12.61	13.10	0.81	1.14
TP	49945.51	1995.81	5.87	1.30	17.30	1139.63	542.03	1.54	162.13	5.15
TN	137.68	647.71	0.37	0.27	0.62	1240.24	12.45	13.10	0.78	1.14
FP	71	16791	0	82708	0	0	4757	0	17	0
FN	0	0	2318	0	7	7	17	4712	0	7

Table 1: *Summary of the average runtime in µs (T), average runtime of positive queries (TP), and average runtime of negative queries (TN), number of false positives (FP), and number of false negatives (FN), for the competing methods.*

simulation dataset that contains over 18 million point-triangle and 41 million edge-edge queries. Our method requires the vertices of the prism or hexahedron to be representable in floating-point coordinates which we achieve by rounding the input (Section 4.2). To compare other methods with ours, we take the initial datasets and apply the necessary rounding for each component (x, y, z) and query individually. We will release the *rounded* dataset in the same format as [WFS*21], with the only modification that we export the

number of roots (we use -1 for infinite roots) instead of a Boolean flag. For each query we measure the error ε as

$$\varepsilon = |a + e - a'|,$$

where a is the original number (i.e., one the x, y, z coordinate of the query), a' is the rounded number, and e is the displacement (Section 4.2). Figure 5 shows the distribution of ε introduced by our method for the four datasets: the maximum is close to machine epsilon and the average is negligible.

5.2. Comparison

We compare our method with the interval root-finder (IRF) [Sny92], the univariate interval root-finder (UIRF) [Sny92, RKC02], the floating-point root finder (FPRF) [VHTG10], TightCCD (TCCD) [WTTM15], Root Parity (RP) [BEB12], a rational implementation of Root Parity (RRP) with the degenerate cases properly handled [BEB12, WFS*21], Bernstein Sign Classification (BSC) [TTWM14], the minimum separation floating-point root finder (MSRF) [HPSZ11] and Tight Inclusion (TI) [WFS*21]. We collect the average query time T, average positive query time PT, average negative query time NT, the number of false positives FP (i.e., there is no collision but the method detects one), and the number of false negatives FN (i.e., there is collision but the method misses it).

Table 1 summarizes the results of the comparison; for fairness we included the average rounding time in our timings (as a reference the rounding takes $0.33\mu s$ for the handcrafted dataset and $0.30\mu s$ for the simulation dataset). The runtime of our method is comparable with TCCD, while being an *exact* root-parity counter. Thanks to the rounding, our method can use standard orientation predicates (since the vertices are *exact* in floating-point numbers) thus making it faster. Comparing with TI, IRF and BSC, our method has a more stable runtime (i.e., the runtime is similar independent of whether the algorithm returns true or false), as TI, IRF, and BSC are significantly slower in returning true. We note that both our algorithm and RP have few false negative (collision missed) for the handcrafted dataset, where [WFS*21] fabricated examples with multiple roots. Multiple roots appear only in 7 queries in the real data, as the time step in simulation is usually small. However only our method (and its variations RP, RRP) are *exact*; that is, they do not produce any false positive.

6. Integration in Existing Simulators

We examine using our CCD in elastodynamic simulations during a line search to prevent intersections [LFS*20]. Our CCD algorithm can replace existing standard CCD algorithms (for sufficiently small time-steps) since it uses the same interface. However, it has one additional requirement: its input needs to be at reduced precision to allow us to evaluate Equation (2) exactly (Section 4.2). This is a minor but fundamental change that needs to be made to existing simulator codes, since our algorithm cannot consume full precision inputs (even if the difference is in the order of machine precision).

6.1. Consistent Rounding Across Time Steps

Our algorithm requires rounding the coordinates of the mesh at lower precision in the simulator, by calling the rounding procedure (Section 4.2) on the mesh vertices before using our CCD. The rounding procedure depends on the displacement e, and, up to this point, we selected the largest e across all candidate collision pairs. While this choice is valid for one time step, it might introduce different rounding across timesteps, thus potentially introducing intersections (given that the rounding is in the order of machine precision, this is unlikely but still a concern).

To avoid this problem, we simply compute a single displacement e for each component valid for the entire simulation by estimating a bounding box of the scene over all frames (conservatively specified to ensure no objects will leave the box during the simulation). In this way, the starting position at step $i+1$ will be identical to the previously rounded final positions at step i (since the displacement e is the same), ensuring an *exact* result. This equates to a single shift of the entire scene once at the beginning of the simulation (a simple modification to existing codes). Note that in the following section we use prime to denote rounded variables (e.g., x' is the rounded version of x).

In our experiments, the error introduced by the rounding is slightly larger than the query experiments in Figure 5 because of the larger scale used to conservatively bound the simulation world: The scenes in figures 6 and 7 have a rounding error of 1.42×10^{-14} and 2.66×10^{-15} respectively. We could not observe any change in the simulation behaviour due to the additional rounding.

6.2. Line Search

Algorithm 2 Time of impact through bisection

```
1:  function CCDBISECTION(x'_0, x'_1, ε_t)
2:      t_0, t_1 ← 0, 1
3:      while |t_1 − t_0| > ε_t do
4:          t ← (t_0 + t_1)/2
5:          x' ← (x'_1 − x'_0)t + x'_0
6:          if CCD(x'_0, x') = COLLISION then
7:              t_1 ← t
8:          else
9:              t_0 ← t
10:         end if
11:     end while
12:     return t_0        ▷ Most conservative value within tolerance
13: end function
```

It is possible to build upon an existing line search in an implicit time-stepping algorithm, that solves for an update Δx to the state variables x, to incorporate our CCD. Taking a full step in the update direction, in general, can lead to difficulties in convergence [NW06], and, in the particular case of collision handling, intersecting states. To avoid these problems one can use a line search to find an appropriate step-size α along the update direction that satisfies both the convergence criterion and the non-penetration constraints. The optimal step-size α is the earliest time of impact that, when using our method, can be approximate using a simple

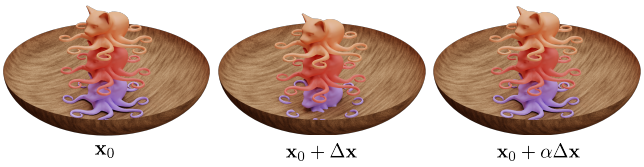

Figure 6: *We drop three elastic Octocats into a bowl and use our CCD within a filtered line search to keep the meshes intersection-free (simulated using IPC [LFS*20] with a timestep of* 0.07 s*). From the current position (left), taking a full step in the update direction leads to serious intersections between the bottom Octocat and bowl (middle). Using Algorithm 2, we determine a step-size of* $\alpha = 0.0625$ *that prevents intersections (right).*

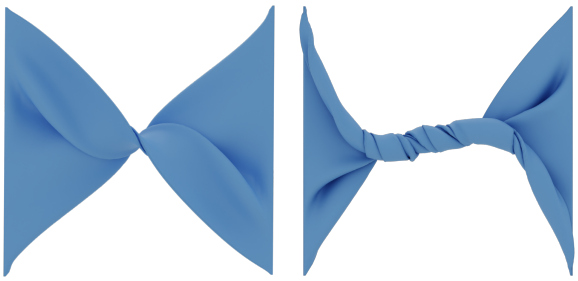

Figure 7: *We run the mat-twist experiment from [LFS*20] using our CCD method in the filtered line-search and timestep of* 0.04 s*. The mat is* $1\,\text{m} \times 1\,\text{m} \times 8\,\text{mm}$ *and our algorithm conservatively shifts the world by* $[11.75, 11.75, 12.24]$ *which results in a rounding error of* 2.66×10^{-15}.

bisection method (Algorithm 2). The bisection starts from a valid rounded configuration x_0' and a final rounded configuration (possibly with collision) x_1', then it iteratively splits (line 4) the timestep and checks for CCD (line 6). If a collision is found the algorithm repeats the same computation for the first half of the timestep (line 7), otherwise for the second half (line 9). The algorithm terminates when the size of the step is smaller than an input numerical tolerance ε_t (line 3). In our simulation experiments (Figures 6 and 7) we use a constant $\varepsilon_t = 10^{-3}$.

Figure 6 illustrates that our CCD used in a filtered line-search is able to find a collision-free α, while Figure 7 shows our method works well even for challenging scenes like twisting cloth.

7. Limitations and Concluding Remarks

We revisited the classical problem of continuous collision detection. By assuming an input with reduced precision and designing an algorithm using floating-point predicates, we show it is possible to solve CCD queries exactly at a cost comparable to commonly used, inaccurate root finders.

Our algorithm has three limitations: (1) it cannot efficiently compute the time of impact (in our experiments we use a naïve bisection algorithm), (2) it requires (minor) changes in the simulation solver to reduce the input precision, and (3) it can only detect the parity of the roots (thus it is unable to distinguish zero from two roots).

The first limitation could probably be addressed by deriving an estimate from the point in the image of $F(\Omega)$ closest to the origin: this is an important avenue for future work. The second limitation is only an implementation concern since the rounding does not affect the simulation accuracy or the runtime in a noticeable way. The third limitation is the most severe: despite the very low probability of this happening (7 cases out of 60 million for the real data), it leads to false negatives, which are problematic for certain solvers [LFS*20]. Addressing this limitation, while still ensuring that no false positives are introduced, is an interesting avenue for future work.

An interesting avenue for future work is the design of an exact algorithm with minimum separation and for non-linear trajectories.

We release the open-source reference implementation of our technique with an MIT license to foster adoption of our technique by existing commercial and academic simulators.

Acknowledgements

We thank the NYU IT High Performance Computing for resources, services, and staff expertise. This work was partially supported by the NSF CAREER award under Grant No. 1652515, the NSF grants OAC-1835712, OIA-1937043, CHS-1908767, CHS-1901091, National Key Research & Development Program of China Grant No. 2020YFA0713701, Natural Science Foundation of China Grants No. 12171023 & No. 12001028, NSERC DGECR-2021-00461 and RGPIN-2021-03707, EU ERC Advanced Grant CHANGE No. 694515, a Sloan Fellowship, a gift from Adobe Research, a gift from nTopology, and a gift from Advanced Micro Devices, Inc.

References

[Att20] ATTENE M.: Indirect predicates for geometric constructions. *Computer-Aided Design* (2020). 5

[BEB12] BROCHU T., EDWARDS E., BRIDSON R.: Efficient geometrically exact continuous collision detection. *ACM Transactions on Graphics 31*, 4 (July 2012), 96:1–96:7. 2, 3, 4, 5, 7, 9

[Can86] CANNY J.: Collision detection for moving polyhedra. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-8*, 2 (1986), 200–209. 3

[GJ*10] GUENNEBAUD G., JACOB B., ET AL.: Eigen v3, 2010. URL: http://eigen.tuxfamily.org. 5

[HF07] HUTTER M., FUHRMANN A.: Optimized continuous collision detection for deformable triangle meshes. 2

[HPSZ11] HARMON D., PANOZZO D., SORKINE O., ZORIN D.: Interference-aware geometric modeling. *ACM Transactions on Graphics 30*, 6 (Dec. 2011), 1–10. 7

[KR03] KIM B., ROSSIGNAC J.: Collision prediction for polyhedra under screw motions. pp. 4–10. 2

[Lév19] LÉVY B.: Geogram, 2019. URL: http://alice.loria.fr/index.php/software/4-library/75-geogram.html. 5

[LFS*20] LI M., FERGUSON Z., SCHNEIDER T., LANGLOIS T., ZORIN D., PANOZZO D., JIANG C., KAUFMAN D. M.: Incremental potential contact: Intersection- and inversion-free large deformation dynamics. *ACM Transactions on Graphics 39*, 4 (2020). 2, 7, 8

[NW06] NOCEDAL J., WRIGHT S. J.: *Numerical Optimization*, second ed. Springer, New York, NY, USA, 2006. 7

[Pro97] PROVOT X.: Collision and self-collision handling in cloth model dedicated to design garments. In *Computer Animation and Simulation*. Springer, 1997, pp. 177–189. 2

[PZM12] PAN J., ZHANG L., MANOCHA D.: Collision-free and smooth trajectory computation in cluttered environments. *The International Journal of Robotics Research 31*, 10 (2012), 1155–1175. 3

[RKC02] REDON S., KHEDDAR A., COQUILLART S.: Fast continuous collision detection between rigid bodies. *Computer Graphics Forum 21* (May 2002). 2, 3, 7

[She97] SHEWCHUK J. R.: Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry 18*, 3 (1997), 305–363. 5

[Sny92] SNYDER J. M.: Interval analysis for computer graphics. *Computer Graphics (Proceedings of SIGGRAPH) 26*, 2 (July 1992), 121–130. 2, 7

[Ste74] STERBENZ P. H.: *Floating-point computation*. Prentice-Hall series in automatic computation. Prentice-Hall, Englewood Cliffs, NJ, 1974. 5

[SWF*93] SNYDER J. M., WOODBURY A. R., FLEISCHER K., CURRIN B., BARR A. H.: Interval methods for multi-point collisions between time-dependent curved surfaces. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1993), SIGGRAPH '93, Association for Computing Machinery, p. 321–334. 2

[TKM09] TANG M., KIM Y., MANOCHA D.: C^2A: Controlled conservative advancement for continuous collision detection of polygonal models. pp. 849–854. 3

[TMT10] TANG M., MANOCHA D., TONG R.: Fast continuous collision detection using deforming non-penetration filters. pp. 7–13. 2

[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 Transactions on Graphics 30* (Jan. 2011), 111. 2

[TTWM14] TANG M., TONG R., WANG Z., MANOCHA D.: Fast and exact continuous collision detection with Bernstein sign classification. *ACM Transactions on Graphics 33* (Nov. 2014), 186:1–186:8. 2, 7

[VHBZ90] VON HERZEN B., BARR A. H., ZATZ H. R.: Geometric collisions for time-dependent parametric surfaces. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1990), SIGGRAPH '90, Association for Computing Machinery, p. 39–48. 2

[VHTG10] VOUGA E., HARMON D., TAMSTORF R., GRINSPUN E.: Asynchronous variational contact mechanics. *Computer Methods in Applied Mechanics and Engineering 200* (July 2010), 2181–2194. URL: https://github.com/evouga/collisiondetection. 7

[Wan14] WANG H.: Defending continuous collision detection against errors. *ACM Transactions on Graphics 33* (July 2014), 1–10. 2

[WFS*21] WANG B., FERGUSON Z., SCHNEIDER T., JIANG X., ATTENE M., PANOZZO D.: A large scale benchmark and an inclusion-based algorithm for continuous collision detection. *ACM Transactions on Graphics* (2021). 2, 3, 4, 5, 6, 7

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

[ZRLK07] ZHANG X., REDON S., LEE M., KIM Y. J.: Continuous collision detection for articulated models using taylor models and temporal culling. *ACM Transactions on Graphics 26*, 3 (July 2007), 15–es. 3

Appendix A: Intersection Predicates

Line-Triangle

The line-triangle predicate checks if a straight line L defined by two points (s_1, s_2) intersects a triangle $T = (t_1, t_2, t_3)$. We compute the sign of the three volumes of the tetrahedra formed by (s_1, s_2) and each edge of the triangle T, that is

$$\begin{aligned} v_1 &= \texttt{orient3d}(s_1, s_2, t_1, t_2), \\ v_2 &= \texttt{orient3d}(s_1, s_2, t_2, t_3), \quad \text{and} \\ v_3 &= \texttt{orient3d}(s_1, s_2, t_3, t_1). \end{aligned}$$

If the three $v_i, i = 1,2,3$ have the same sign we return true. Note that this predicate can also detect if the intersection is on an edge of T by checking if only one of the v_i is zero and the other two have same sign.

Ray-Triangle

The ray-triangle predicate checks if a ray R defined by an oriented pair of points (s_1, s_2) intersects a triangle $T = (t_1, t_2, t_3)$. This is similar to the line-triangle intersection (Appendix A). The only difference is that we need to check if the three $v_i, i = 1,2,3$ have the same sign as $o_1 = \texttt{orient3d}(s_1, t_1, t_2, t_3)$ to ensure that the ray is pointing towards the plane spanned by T and not in the opposite direction.

Appendix B: Definition of the function ϕ

The implicit function $\phi(x)$ [BEB12] of a bilinear quad b is defined as

$$\phi(x) = h_{23}(x) - h_{14}(x),$$

where the indices 1 to 4 refer to vertices of b, and $h_{23}(x)$ and $h_{14}(x)$ are defined as

$$\begin{aligned} h_{23}(x) &= g_{123}(x) g_{143}(x), \\ h_{14}(x) &= g_{243}(x) g_{124}(x), \\ g_{pqr}(x) &= (x - x_p) \cdot (x_q - x_p) \times (x_r - x_p). \end{aligned}$$

Then the zero level set of $\phi(x)$ contains the bilinear quad b. Moreover, the sign of $\phi(x)$ changes across the zero level set, dividing space into positive and negative regions.