Vol.2015-CG-161 No.19
Vol.2015-CVIM-199 No.19
2015/11/7

Real-time Ray-traced Collision Detection for Deformable Objects

SUIYAN LI[†1] YASUSI YAMAGUCHI[†2]

Abstract: In real-time physical simulation, penetration based collision detection is a popular method. The study focuses on ray tracing based collision detection, which is one of the penetration based collision detection methods. The advantage of the ray tracing based collision detection is that it calculates collision rays which can be used for accelerating the physical response computation. Our method is based on Lehericey et al.'s work [4], which exploits spatial and temporal coherency, to speedup this algorithm. But their method cannot be used for deformable objects. We proposed a pipeline that can be entirely used for deformable objects. We also present a new implementation of prediction algorithm that can detect possible and even close collisions whenever two objects have potentially colliding vertices. Our implementation can calculate ray-triangle intersection faster than previous implementation. In addition, strategies for self-collision is discussed. The proposed method is compared with the state-of-the-art conventional algorithm for several scenes containing deformable objects. The results show that our method achieves competitive time and quality performance in comparison of the conventional algorithm in the narrow phase, and perform better in the physical response phase.

Keywords: Collision Detection, Ray Tracing, Deformable Objects, Real-Time Physics-based Simulation

1. Introduction

Collision detection (CD) is an important task in the applications of 3D physical simulation. In general, collision detection is divided into two phases: broad phase and narrow phase [1]. The broad phase takes all the objects from the scene and finds a set of potential pairs of objects that may collide. The narrow phase takes these pairs as input and outputs the ones that actually collide.

Recent narrow phase algorithms use parallel GPU computation to achieve a speedup. Ray traced collision detection is one of the image-based narrow phase algorithms. It has the merit that the information of collision rays can be used immediately to calculate the physical response, which are not directly available when using conventional collision detection algorithms. This may leads to reduction of total time of the physical simulation pipeline.

However, naive ray traced collision detection cannot achieve ideal speed performance for deformable objects. The major problem for handling deformable objects is that the acceleration data structures for ray tracing are difficult to update for every time step, which is necessary for deformable objects. We propose a pipeline able to update the acceleration structure for deformable objects.

Secondly, previous study proposed to use predictive rays for their iterative algorithm to improve the quality of collision detection by preventing deep penetration. The problem is that naive implementation may lead to twice amount of the rays to be cast, meaning twice computation time. We propose an optimal implementation of predictive rays.

Thirdly, it is difficult to deal with self-collision in the context of ray traced collision detection. The major problem is that when we want to detect self-collision, then we need to cast rays from almost all the vertices in the object. We discuss how to deal with self-collision in the case of open meshes for cloth simulation.

In our experimental evaluation, we use several scenes to investigate the time performance and the quality of collision detection of our approach, and compare the results with that when using traditional algorithms.

The remaining part of this paper is organized as follows: Section 2 introduces related work. Section 3 explains our proposed pipeline of ray traced collision detection. Section 4 explains our improvement of predictive rays. Section 5 explains the algorithm to deal with self-collision of open meshes, such as cloth. Section 6 shows our experimental results. Section 7 concludes this paper and mentions future research issues.

2. Related work

There are several broad phase and narrow phase algorithms which have different processes and can be used for different types of objects. For the broad phase, BVH is the most popular method for acceleration, which is a tree structure on a set of geometric objects. By arranging the bounding volumes into a bounding volume hierarchy, the time complexity (the number of tests performed) can be reduced to logarithmic in the number of objects.

There are several mainly four types of narrow phase algorithms. BVH may be the most popular narrow phase algorithm. Feature based narrow phase algorithms work on geometric primitives of objects, of which the most famous are Lin-Canny, V-Clip, SWIFT, and polygon intersection of Moore and Wilhelms. This type of algorithms focus on finding the two closest points between two objects, and regard as colliding if each point is inside the other object. Simplex based narrow phase algorithms work on the convex hull of objects, which cannot be correct for concave objects without using more complex algorithms. The most famous simplex based narrow phase algorithm is GJK (Gilbert-Johnson-Keerthi) [2], which works on the Minkowski difference instead of the two objects immediately. Image based narrow phase algorithms exploit image rendering techniques in GPUs to perform collision detection. There are two types of image-based approaches, depth-peeling and ray-casting.

This study focuses on ray tracing based collision detection for

†1 The University of Tokyo
†2 The University of Tokyo / JST CREST

narrow phase [3], whose basic idea is that cast rays from vertices of a source object in the direction of the inward normal, and then collision can be detected if a ray hits another object inside of the source object (shown in Figure 1). This method is one of the type of so-called penetration based collision detection.

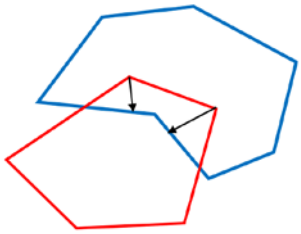

Fig.1 Illustration of the ray tracing based collision detection

Lehericey et al. [4] proposed a method to accelerate the ray-triangle intersection computation of naive ray tracing based collision detection exploiting the temporal and special coherence. However, to avoid the low speed of using ray tracing, Lehericey et al. use an acceleration data structure to get a speedup from naive ray tracing. However, this method cannot be used for deformable objects, because it is difficult to update the acceleration data structure at each time step. The study aims to make ray tracing based collision detection can be entirely used for deformable objects.

3. Pipeline of real-time ray traced collision detection for deformable objects

Ray-traced collision detection algorithms proposed in [3], [4] can be used on multi-CPUs or multi-GPUs to achieve high performance for complex scenes. Usually, only one ray-tracing algorithm is used to detect collisions in a simple scene. But in complex scenes with objects of different nature we can employ several ray-tracing algorithms to optimize each nature of object. With rigid objects we can use algorithms with static data structures. With deformable objects we need data structures that can be updated at each time-step. In the case of topology changes we need to reconstruct the ray-tracing data structures occasionally. Iterative ray-tracing can be used to accelerate all of these algorithms.

To make ray tracing based collision detection available for deformable objects, we proposed a novel pipeline for real-time ray-traced collision detection. The most important feature of this pipeline is the capability of updating acceleration data structure in every time-step for both broad phase and ray-object intersection computation. Several efficient acceleration data structures proposed recently for ray tracing rendering can be used for accelerating ray traced collision detection. Figure 2 shows the whole pipeline of real-time ray-traced collision detection for deformable objects.

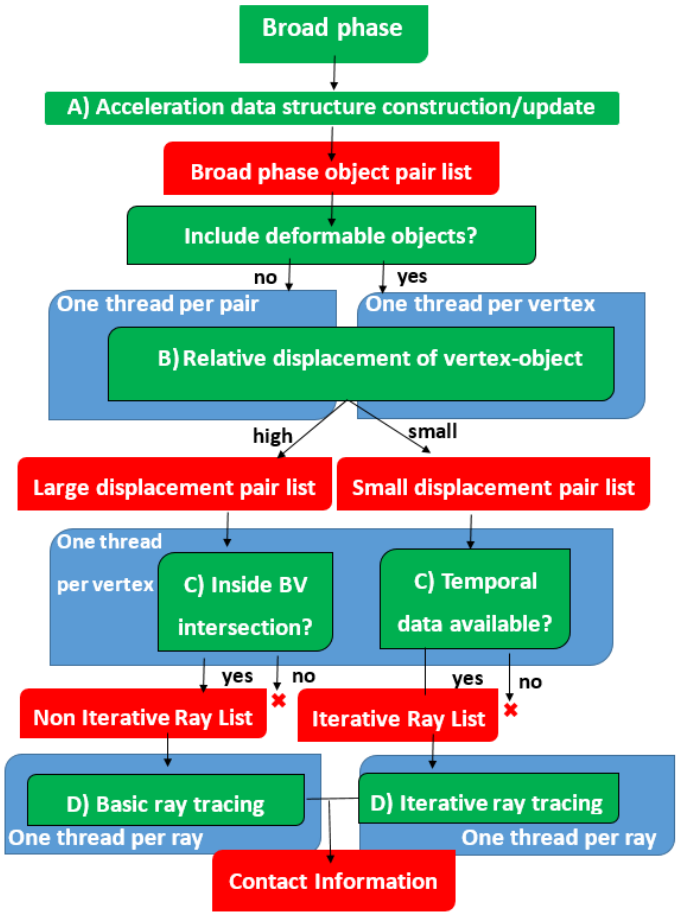

Fig.2 Pipeline of real-time ray traced collision detection for deformable objects

As shown in Figure 2, four main tasks have to be finished to achieve real-time ray traced collision detection for deformable objects: A) Building or updating acceleration data structure for ray tracing; B) **Measuring the displacement** on the pair of objects to check if the pair can be processed by iterative ray tracing algorithm; C) **Culling vertices** that are not inside the intersecting region of bounding volumes of the objects or whose temporal data are unavailable; D) **Executing ray tracing** according to the type of the ray list.

When applying the displacement measurement, there are the case of including deformable objects or not. If there is a deformable object in the object pair, then displacement measurement should be applied per vertex because of internal deformations. For the pair of rigid objects, the displacement measurement can be applied globally. Object pairs with small displacement will use the iterative ray-tracing algorithm. We need a relative displacement measurement and a threshold that indicates when the non-iterative algorithm must be used.

Vertex culling removes vertices that do not need to be tested with ray-tracing because a simple test can discard collision. For the object pairs checked with large displacement, we check if the vertex is located inside the intersection of the bounding volumes of the two objects. For the vertices checked with small displacement we test if temporal data is available.

Ray-tracing is executed for each of the remaining vertices. We

use the iterative algorithm when selected; otherwise, we use a basic ray tracing algorithm. Further improvements are possible, such as applying different types of ray tracing according to nature of the target.

3.1 Acceleration data structure

Acceleration structures are important tools for speeding up the traversal and intersection queries for ray tracing. Most of successful acceleration structures represent a hierarchical decomposition of the object or scene geometry. This hierarchy is then used to quickly cull space region not intersected by the ray.

The most important factor of acceleration data structures for ray traced collision detection of deformable objects is the speed of construction/update. We choose two acceleration data structures with high construction speed. The first one is Hierarchical Linear Bounding Volume Hierarchy (HLBVH) [5]. The structure is an optimization of Linear BVH (LBVH), which can be constructed very quickly on the GPU. The other one is Treelet Restructuring BVH (TrBVH) [6]. In fact, TrBVH is not an acceleration data structure, but an approach to transform a low-quality BVH, which can be constructed in a matter of milliseconds, into a high-quality one that is close to the gold standard in ray tracing performance.

3.2 Classification of threads for GPU implementation

Our pipeline has three different kinds of steps for GPU thread scheduling in terms of their processing objects. Each kind of step is designated by a blue box in Figure 2.

3.2.1 Per-pair step

The per-pair step takes the list of object pairs from the broad phase as input. In the step one thread corresponds to one pair of objects, and the major work is to apply measurement of displacement for the pairs with only rigid objects.

This step separate the vertices in the pairs with small displacement that will use the iterative algorithm, and vertices with large displacement that will use a basic ray-tracing algorithm. The pairs that contain at least one deformable object cannot be checked in this step. In such cases displacement needs to be measured locally to take into account internal deformations.

3.2.2 Per-vertex step

In the per-vertex step, one thread is executed for each vertex of each object of each pair. This step has two types of tasks with different input.

The first type takes the pairs containing deformable objects as input, and applies a displacement measurement on the vertices to separate the ones with small displacement that will use the iterative algorithm and the others that will use a basic algorithm.

The second type is executed for each vertex of each object of each pair. For the vertices with large displacement that will use a basic algorithm, we check if the vertex is inside the intersection of the bounding volumes of the two objects. If not, we can discard this vertex as it cannot collide. For the vertices with small displacement that will use the iterative algorithm, we test if temporal data is available from the previous time-step. If no temporal data is available we can discard the vertex.

After the vertex culling task, each remaining vertex generates a ray that will be cast on the other object of the pair. Parameters needed to cast these rays are stored in two buffers as different input for iterative and non-iterative ray tracing.

3.2.3 Per-ray step

The per-ray step takes the lists of rays as input and performs ray tracing. Each thread casts one ray. Each ray-tracing algorithm is implemented and executed in a separate kernel to avoid branch divergence. The ray-tracing algorithms output contact information for computing the physical response.

4. Implementation of Predictive Ray/Triangle Intersection

The main demerit of the iterative ray tracing [4] is that it only works with previously detected rays. When new vertices collide, they will be processed only in the next standard step. This may postpone the collision detection of a pair of objects for several time steps, which will lead to a deep interpenetration.

The first row of Figure 3 shows an example of this situation. At $t = 0$ a non-iterative algorithm is executed and no collision is detected. At $t = 1$ the previous rays are updated, as the two objects were not colliding at $t = 0$ there are no rays to update. In this case the iterative algorithm fails to detect the collision. At t = 2 a non-iterative algorithm is executed and the collision is detected. In this example the detection is postponed for one step, but in practical cases it may be more.

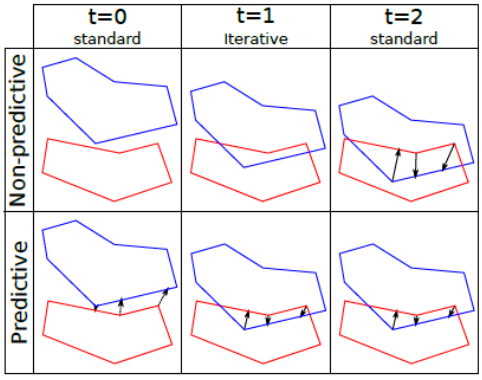

Fig.3 Predicted iteration from [7]

Lehericey et al. [7] proposed to solve the problem by performing a predictive ray triangle intersection. If a ray report no collision, then a predictive ray can be cast from the same vertex but in the direction of the normal, opposite to the ray which has been cast. If this predictive ray hits a triangle and if the distance is short, the corresponding vertex may hit that triangle (or the neighboring ones) in a near future. The predictive rays are injected in the iterative algorithm as candidates for the next steps.

The second row of Figure 3 shows how predictive rays prevent postponement of collision detection. At $t = 0$ a non-iterative algorithm is executed and no collision is detected, predictive rays are cast in the outward direction of the objects, and several predictive rays kept. At $t = 1$ the predictive rays are updated and collision is successfully detected, at $t = 2$ a physical response can be applied to prevent further interpenetration.

However, casting a predictive ray in the opposite direction

double the cost of the non-iterative ray tracing algorithm theoretically. We proposed a method of implementation to reduce the cost of performing predictive ray tracing. In practical case we do not need to cast a ray twice. This process is executed one thread per ray. We can process the predictive ray in the same thread of the non-collision ray because they share the same vertex and are in the same line. When using parallelization technique, the threads processing non-collision rays will terminate rapidly and have to wait for other threads. Processing predictive ray in these idle threads exploits the idle computation resources that are wasted.

Furthermore, we have cast rays from the vertices that are inside the intersection of the bounding volumes, which is an optimization to reduce the number of rays. In the context of predictive rays this optimization may discard predictive rays and delay the collision detection. To avoid this problem we extend the intersection of the bounding volumes by a certain distance. This ensures that we do not discard predictive rays as long as the relative displacement between the two objects in inferior to the certain distance. It value must be minimized for better performances because higher values increase the number of ray cast thus increasing computation.

5. Strategies for self-collision of open meshes

5.1 Self-collision problems of ray traced collision detection

Except of the problem of acceleration data structures to achieve ray traced collision detection for deformable objects, previous studies [3], [4] also face the problem of self-collision. Any primitive can potentially collide with any other primitives of the same object. This is the first reason to avoid self-collision detection. It tends to be more expensive than inter-object collision, because we need to trace rays from every vertex in deformable objects. One solution is to select a subset of vertices and not tracing rays from all of them, however the final result quality of this strategy is uncertain.

The second reason is that ray tracing for self-collision detection needs different processes with the inter-object one, and it may lead to more problems. Hermann et al. [3] cast rays in the direction of the outward vertex normal, namely the opposite direction of usual ray casting. If a ray hit another point of the same object and the normal at the point has the same orientation as the ray then a self-collision is detected. The reason is that they want to quickly discard rays that will not cross any face of the object. By tracing outward rays they exploit the fact that an octree (Hermann et al. use an octree as acceleration data structure) is usually much sparser in the regions outside of an object than those inside of the object. A ray that will not cross any other face of the same object will quickly reach a large cell that does not contain the object, and then be discarded. However, casting rays in the direction of vertex normal, i.e. outwards, cannot be directly used for physical response as shown in Figure 4. The correct ray to get the physical response should be the penetrating path, i.e. the inward ray between the origin vertex and the inward intersection point.

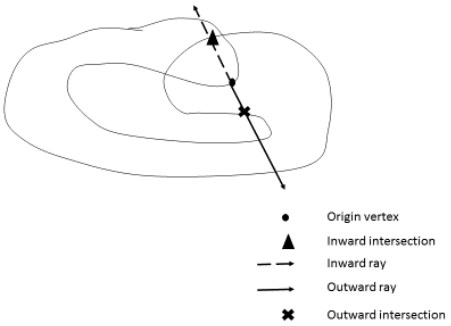

Fig.4 Inward and outward ray of self-collision

Furthermore, in the case of open meshes, such as cloth, it is difficult to judge which direction is inward/outward of a face. For this reason, we need a new strategy to work for self-collision of open meshes.

5.2 Strategies for open meshes

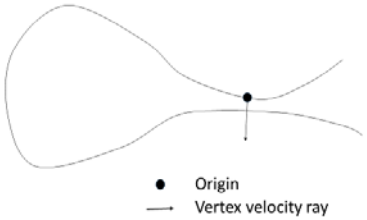

Fig.5 Ray traced self-collision for cloth

Instead of casting rays in the direction of vertex normal, we proposed to cast short rays in the direction of vertex velocity as shown in Figure 5. The maximum length of the ray should be smaller than a threshold, such as a maximum displacement in one time-step.

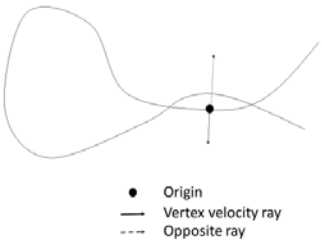

Fig.6 “Penetrating” case of self-collision of cloth

However, in practical cases like that shown in Figure 6 may happen. A vertex may go through the other part of the cloth during a time-step along the direction of velocity. We can cast an opposite ray to deal with this “penetrating” case. If the opposite ray hits within a threshold, a self-collision is detected.

Algorithm 1 shows the pseudo code of the process of dealing with self-collision of open meshes.

Algorithm 1 Self-collision of open meshes

```
1: procedure SELFCD(objectA)
2:     for each vertex v in A do
3:         direction ← directionOfVelocity(v)
4:         ray1 ← newRay(v, direction)
5:         intersection ← closestHit(ray1, A)
6:         if isHit(intersection) and notExceedThreshold(intersection) ther
7:             report new self-collision intersection
8:         end if
9:         ray2 ← newRay(v, −direction)
10:        intersection ← closestHit(ray2, A)
11:        if isHit(penetration) and notExceedThreshold(intersection) then
12:            report new penetration type self-collision intersection
13:        end if
14:    end for
15: end procedure
```

6. Experiment and Evaluation

In order to construct the proposed pipeline of real-time ray traced collision detection for deformable objects, we implement the system based on the open source physic engine, Bullet Physics. The majority of the implementation is about the narrow phase algorithm of ray traced collision detection, and several acceleration data structure for ray tracing.

The implemented system can take different types of objects as input and visualize the results of physical simulation. We have evaluated several combinations of ray traced collision detection algorithms on several scenes. Furthermore, we implement the physical response phase directly using the rays, which allows to improve the time performance of the full physical simulation pipeline.

6.1 Experimental setup

Our test platform is a quad-core Intel Core i7 4700MQ and NVIDIA GeForce GTX 770M. The first experimental scene (shown in Figure 7) contains 100 tori falling on a planar ground, where each torus is composed of 300 vertices and 600 triangles. The second experiment scene (shown as Figure 8) contains 100 bunnies falling on a planar ground, where each bunny is composed of 453 vertices and 902 triangles. At the end of both scene, all 100 objects of 60k/90k triangles in total exist in the scene.

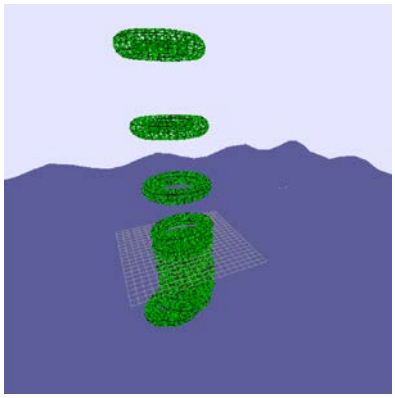

Fig.7 100 tori falling on the ground

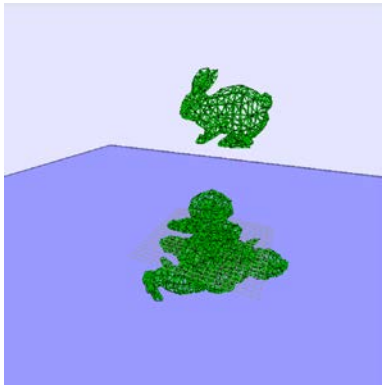

Fig.8 100 bunnies falling on the ground

We implement the experimental scenes using Bullet Physics. The broad-phase is executed on the CPU. We implement the acceleration data structure and ray tracing collision detection algorithm using CUDA, which can be executed on GPU. The physical response phase is executed on the CPU. This setup leads to memory transfer between the CPU and GPU. To improve the performance of the full pipeline, in practical cases, the whole system should be implemented on GPU or CPU to avoid memory transfers.

6.2 Comparison of different acceleration data structures of ray traced collision detection

Before we implement the ray traced collision detection algorithms, we test the time performance of acceleration data structures.

We implemented HLBVH and TrBVH introduced in section 3.1. The traversal performance of these structures may not be as significant as others, but they are suitable for deformable objects because of the need of rebuilding the data structure in every time-step. Table 1 shows the time for updating acceleration data structure and the time of ray tracing traversal when using the two acceleration data structures for ray traced collision detection in Scene 1 (100 tori). HLBVH updates faster than TrBVH, but TrBVH performs better in ray tracing traversal of the narrow phase, which results in that TrBVH spends less total time than HLBVH during collision detection. In the rest of experiments, we use TrBVH as the acceleration data structure for ray traced collision detection.

Table 1 Time performance of using TrBVH and HLBVH for ray traced CD in scene 1 (100 tori)

Algortihm	Ray Traced CD (TrBVH)	Ray Traced CD (HLBVH)
Updating Data Structure	3.1ms	0.7ms
Narrow phase	56.7ms	72.0ms
Total	59.8ms	72.7ms

6.3 Iterative ray tracing performance

We implement the iterative ray traced algorithm and make it work for deformable objects. Table 2 shows simulation time of the narrow phase with/without iterative ray tracing for the two scenes.

The Iterative algorithm can work faster during time-steps exploiting the temporal and spatial coherency. The results show that different scenes/objects may lead to different rates of performance improvements with the iterative algorithm.

Table 2 Iterative ray tracing performance

scene	Scene 1(100 Tori)	Scene 2 (100 Bunnies)
TrBVH without iterative	56.7ms	129.2ms
TrBVH with iterative	36.2ms	116.3ms

6.4 Predictive Ray performance

As introduced in section 4, native implementation of predictive ray for iterative ray-traced algorithm requires twice the amount of rays. We proposed to process the predictive ray in the same thread of the non-collision ray, which may usually be in an idling state because it waits for the termination of the threads of collision rays. The results of the experiment are shown in Table 3. Our implementation of predictive ray spent almost the same time as the ray-traced algorithm without predictive ray.

Table 3 Predictive ray performance

scene	Scene 1(100 Tori)	Scene 2 (100 Bunnies)
TrBVH with iterative	36.2ms	116.3ms
TrBVH with iterative + predictive	37.2ms	118.6ms

We also compare the average penetration depth for Scene 1 (100 tori) with/without predictive rays. Figure 9 shows that during the simulation time, the average penetration depth when using predictive rays is always smaller than that without predictive rays. It means that predictive ray can effectively prevent deep penetration, which can lead to improvement of the quality of collision detection.

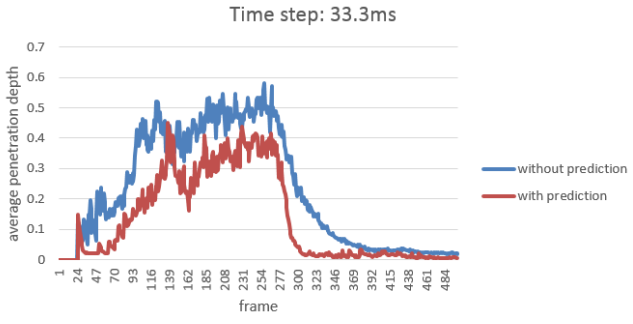

Fig.9 Average penetration depth with/without predictive ray for Scene 1 (100 tori)

6.5 Comparison with the traditional method

The traditional narrow phase collision detection algorithm compared with the proposed algorithm is the well-known GJK algorithm introduced in section 2. We also compares the time of calculating physical response in ray traced collision detection to a traditional one, i.e., the expanding polytope algorithm (EPA) for penetrating depth.

Table 4 time performance of traditional and ray traced CD in Scene 1 (100 tori)

Algortihm	Traditional	Ray Traced CD
Updating Data Structure	-	3.1ms
Narrow phase	70.3ms (dynamicBVH + GJK)	56.7ms (TrBVH)
Physical Response	6.6ms (EPA)	1.2ms
Total	77.0ms	61.0ms

Table 4 shows the time performance of ray traced algorithms against GJK + EPA algorithm in Scene 1 (100 tori). The results show that the proposed collision detection algorithm achieves competitive narrow phase performance in comparison of the traditional algorithm. And as a benefit of ray traced collision detection is that it can use the information of contact rays immediately for physical response, the ray-traced method achieves better time performance of physical response against traditional EPA algorithm.

As an algorithm of penetration type of collision detection, our ray traced collision detection method may discard some collision in some time steps. To evaluate the quality of the ray traced collision detection, we compare the number of collision pairs in each frame against traditional GJK algorithm.

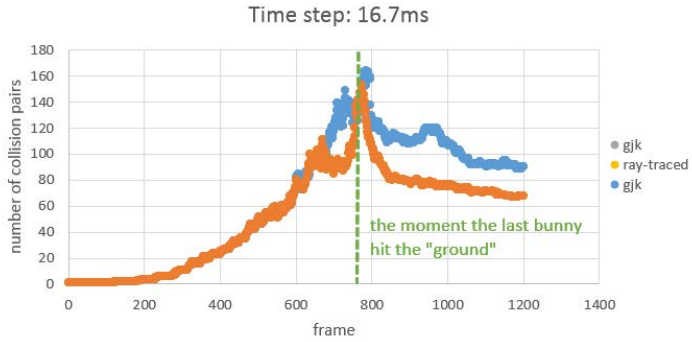

Fig.10 Number of collision pairs in each frame in the scene of 30 bunnies

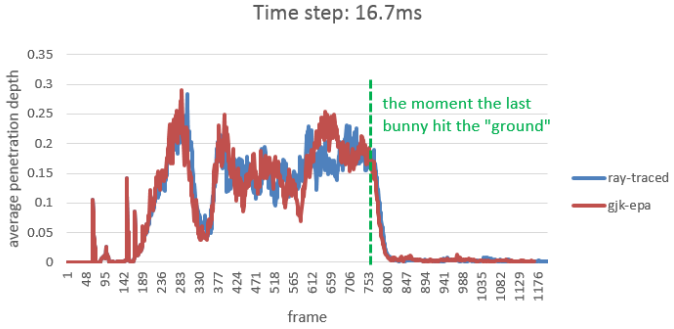

Fig.11 Average penetration depth of ray traced CD vs. GJK-EPA in the scene of 30 bunnies

Figure 10 and 12 show the number of colliding pairs in each frame. In both scenes two curves has a high concordance. After 770th frame in Figure 10, there exists a difference between two CD, but it makes no big difference in the result because those frames are actually quiet scene so that no bunnies are moving. It means that the quality of ray traced collision detection is competitive with GJK algorithm in the view of collision number.

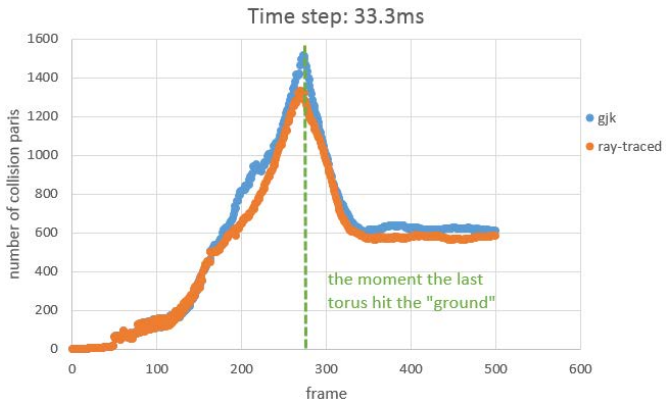

Fig.12 Number of collision pairs in each frame in the scene of 100 tori

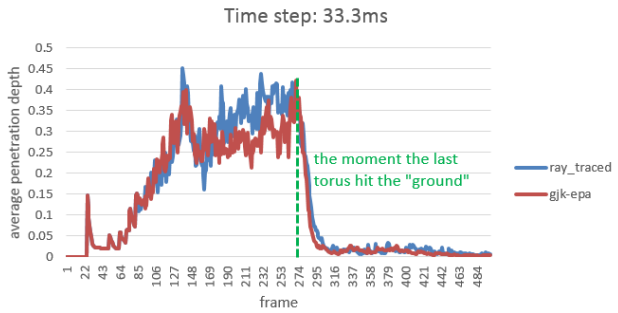

Fig.13 Average penetration depth of ray traced CD vs. GJK-EPA in the scene of 100 tori

We also compare the average penetration depth during the simulation in the case of our method against that in the case of using traditional algorithms (GJK + EPA). Figure 11 and 13 show that when using our method, the average penetration depth is just a little larger than that when using traditional algorithms (GJK + EPA) during the whole simulation. We can conclude that in the view of average penetration depth, our ray traced collision detection achieves a competitive quality against the traditional method.

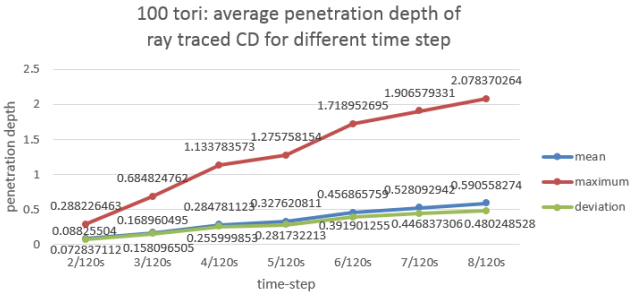

Fig.14 Average penetration depth for different time-step

As penetration depth is mainly decided by the velocity and the simulation time-step, collision detection with high quality should report near linear relationship between the penetration depth and the simulation time-step. We change the simulation time-step from 2/120s, 3/120s, 4/120s, 5/120s, 6/120s, 7/120s, 8/120s, and watch how the penetration depth changes. Figure 14 shows the average penetration depth during the simulation (from the time 3 seconds after starting to the time 10 seconds when the last torus hits the ground). Three lines represent the mean of the penetration depth, the mean of the maximum of penetration depth, and the mean of the standard deviation. All three lines show a near linear change with the time-step.

6.6 Evaluating self-collision of open meshes

In order to evaluate the quality of self-collision of open meshes such as cloth, we use a scene (as shown in Figure 15) to measure the maximum penetration depth for every frame. The experiment scene is that a cloth falls onto a capsule, and then the self-collision happens.

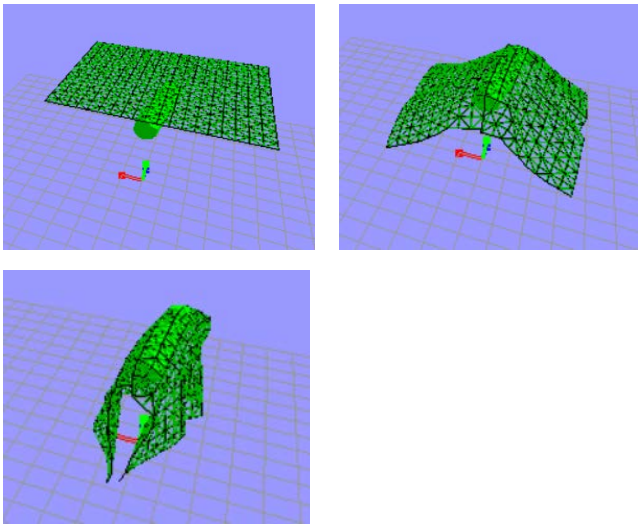

Fig.15 Self-collision of cloth when it falls onto a capsule

If the maximum penetration depth always keeps small enough, then our method introduced in section 5 is effective. Figure 16 shows the results. The maximum of penetration depth during the simulation is about 5.8651e-4, which is about 1/680 of the unit node distance 0.4 (distance of the 20*20 nodes which the cloth consists of). It is a value that small enough to show that our method can effectively detect self-collision of open meshes.

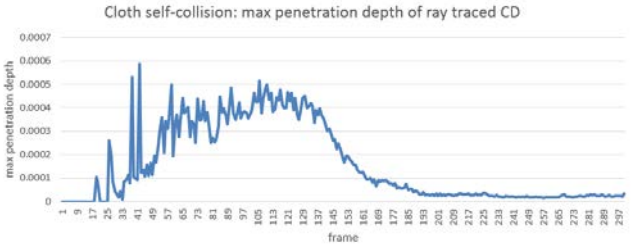

Fig.16 Max penetration depth of self-collision of a cloth

7. Conclusion

In this study, we present our idea on the pipeline of real-time ray traced collision detection for deformable objects. This method is based on building acceleration data structure of ray tracing for deformable objects in every time-step. Predictive rays are used to prevent deep penetrations. We propose a method to keep the time performance when using predictive rays by exploiting the characteristic of parallel computation. We also propose strategies to achieve self-collision for ray traced collision detection in the case of open meshes, such as cloth.

We implement the pipeline using the physic engine Bullet Physics. Traditional and ray traced algorithms are compared for several scenes containing deformable objects. The results show that ray traced collision detection for deformable objects achieve competitive performance in comparison of the traditional algorithm in the narrow phase, and perform better in the physical response phase because the information of contact rays can be used immediately. Moreover, in the view of the number of collision pairs and average penetration depth, our method achieves a competitive quality of collision detection against the traditional method of collision detection.

Our future works may include proposing and evaluating self-collision strategies for closed meshes. We may compare our method against some traditional self-collision algorithms to check if our method get a near number of self-collision objects as traditional algorithms.

Reference

[1] Hubbard P M. Interactive collision detection[C]//Virtual Reality, 1993. Proceedings. IEEE 1993 Symposium on Research Frontiers in. IEEE, 1993: 24-31.

[2] Gilbert E G, Johnson D W, Keerthi S S. A fast procedure for computing the distance between complex objects in three-dimensional space [J]. Robotics and Automation, IEEE Journal of, 1988, 4(2): 193-203.

[3] Hermann E, Faure F, Raffin B. Ray-traced collision detection for deformable bodies[C]//GRAPP 2008-3rd International Conference on Computer Graphics Theory and Applications. INSTICC, 2008: 293-299.

[4] Lehericey F, Gouranton V, Arnaldi B. New iterative ray-traced collision detection algorithm for gpu architectures[C]//Proceedings of the 19th ACM Symposium on Virtual Reality Software and Technology. ACM, 2013: 215-218.

[5] Pantaleoni J, Luebke D. HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry[C]//Proceedings of the Conference on High Performance Graphics. Eurographics Association, 2010: 87-95.

[6] Karras T, Aila T. Fast parallel construction of high-quality bounding volume hierarchies[C]//Proceedings of the 5th High-Performance Graphics Conference. ACM, 2013: 89-99.

[7] Lehericey F, Gouranton V, Arnaldi B. Ray-traced collision detection: interpenetration control and multi-gpu performance[C]//Proceedings of the 5th Joint Virtual Reality Conference. Eurographics Association, 2013: 33-40.