Collision Detection Based on Fuzzy Scene Subdivision

David Mainzer[1] and Gabriel Zachmann[2]

[1] Clausthal University, Germany
dm@tu-clausthal.de
[2] University of Bremen, Germany
zach@cs.uni-bremen.de

Abstract. We present a novel approach to perform collision detection queries between rigid and/or deformable models. Our method can handle arbitrary deformations and even discontinuous ones. For this, we subdivide the whole scene with all objects into connected but totally independent parts by a fuzzy clustering algorithm. Following, for every part our algorithm performs a Principal Component Analyses to achieve the best sweep direction for the Sweep-Plane step, which reduces the number of false positives greatly. Our collision detection algorithm processes all computations without the need of a bounding volume hierarchy or any other acceleration data structure. One great advantage of this is that our method can handle the broad phase as well as the narrow phase within one single framework. Our collision detection algorithm works directly on all primitives of the whole scene, which results in a simpler implementation and can be integrated much more easily by other applications. We can compute inter-object and intra-object collisions of rigid and deformable objects consisting of many tens of thousands of triangles in a few milliseconds on a modern computer. We have evaluated its performance by common benchmarks.

Keywords: Collision Detection, Fuzzy Clustering, Physics based Animation, Computer Animation, Cloth Simulation

1 Introduction

Collision detection between rigid, and/or soft bodies is important for many fields of computer science, e.g. for physically-based simulations, medical applications like virtual surgery, and cloth simulation. The underlying collision detection needs to check if collisions occur between a pair of objects as well as self-collisions among deformable objects. In many applications, an additional requirement is that the collision detection has to be calculated within milliseconds. Penalty-based physical simulations, for example, typically perform a number of iterations for a single rendering frame, requiring collision detection at $n \times 30$Hz, if the scene is rendered at 30Hz.

There exist various approaches that propose spatial subdivision for collision detection or approximate the surface of rigid and soft bodies. These algorithms employ axis-aligned bounding boxes (AABB) [22], oriented bounding boxes (OBB) [5] or Inner Sphere Trees (IST) [23] to reduce the computation time.

Most of the earlier efficient collision detection algorithms were sequential ones, which are perfect for devices that can execute only one instruction at a time. The current

trend in computer architecture focuses on multi-core CPUs and many-core GPUs, and so many parallel collision detection algorithms have been proposed in the last years. The collision detection algorithm we present in this Chapteris a fast, fully GPU-based algorithm that can exploit data and thread-level parallelism.

Modern GPUs can be thought of as many-core stream processors, and such streaming architectures have significant implications on algorithm design, especially when applied to general purpose tasks because they were initially designed for graphics manipulations. Because of this, many prior GPU-based collision detection algorithms [8, 7, 15] or hybrid combinations of CPU and GPU [6, 11, 16] have been developed. A lot of well-known culling methods for collision detection algorithms exist, which include *Sort and Sweep* [1], also known as *Sweep and Prune* [3], to limit the number of pairs of primitives that need to be checked for collision. Without using these culling methods, a huge amount of computation time is wasted and additional memory access is needed, which takes a lot of time especially when accessing global memory on GPUs.

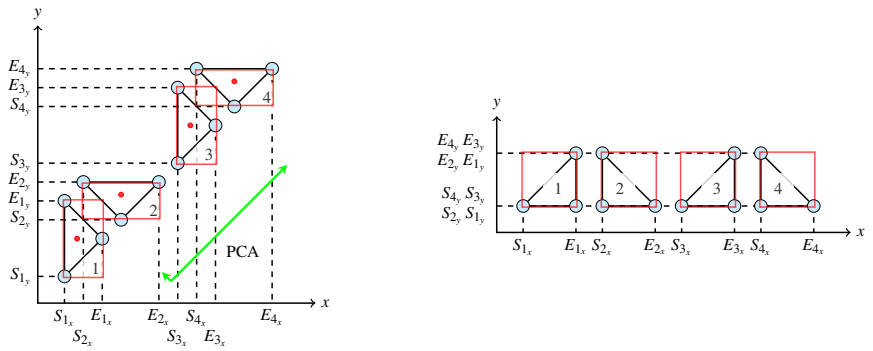
1.1 Our Contributions

Our novel *Collision Detection Based on Fuzzy Scene Subdivision* algorithm is designed for interactive and exact collision detection in complex environments and can handle objects movement and deformation at the same time. To achieve these features, our algorithm subdivides the whole scene, with all objects, into independent, overlapping parts in the first step. For the segmentation process, we use a GPU-based clustering algorithm called *fuzzy C-means* (see Section 4). For all clusters, we can execute the collision detection steps independently, and this offers the possibility to distribute the collision detection computation for the clusters to different GPUs. To reduce the number of false positives we use an adapted version of the *Sweep and Prune* approach in combination with *Principal Component Analysis* (see Section 3). This has the advantage that our algorithm does not need to distinguish between a broad and narrow phase.

Our novel approach is as fast as state-of-the-art collision detection algorithms but with the additional advantage that our collision detection can be distributed easily to more than only one GPU, because we subdivides the whole scene into independent but connected parts; thus it scales very well with the number of GPUs. Also, our collision detection algorithm works directly on all primitives (e.g. triangles) of the whole scene, which results in a simpler implementation and can be implemented much more easily by other applications. In addition to that, working on all primitives directly avoids approximate errors.

2 Previous Work

Since collision detection is a fundamental technique in many simulations, it has been extensively investigated by researchers over the last decades. As a result, a large number of different techniques for collision detection queries and handling exist [20]. In this section, we focus on those approaches only, that can handle collisions between deformable objects.



(a) The initial scene consisting of a number of triangles with corresponding bounding boxes and the result of the Principal Component Analysis. As can clearly be seen the bounding boxes of triangle 1 and 2, and triangle 3 and 4 intersect.

(b) Initial scene from Fig. 1a, rotated so that the direction of the first component of the Principal Component Analysis points along the x-axis. As can clearly be seen, in this example the number of overlapping bounding boxes reduced to zero.

Fig. 1: Improvement of sweep-plane approach via Principal Component Analysis

2.1 Approaches Using Bounding Volume Hierarchies

Using Bounding Volume Hierarchies (BVH) is the most common approach to speed up collision detection of rigid and deformable objects [4]. Govindaraju et al. [6] used precomputed chromatic decomposition of a mesh to check for collisions between non-adjacent primitives. A limitation of this approach is that the connectivity of the mesh has to be fixed. Consequently, this approach is not applicable when you want to simulate ripping or cutting a virtual object, which have main importance in simulations like virtual surgery and advanced cloth animation. Greß et al. [7] used stenciled geometry images to generate GPU-optimized BVH in real-time. This approach is optimized for collision and self-collision detection for NURBS models or other types of rigid or deformable parameterized surfaces. This approach is limited to a few thousand NURBS patches. Kim et al. [11] presented a hybrid CPU-GPU parallel continuous collision detection (HPCCD) method. HPCCD is based on a BVH and performs efficient reconstructions for selective parts of the BVH. Because they do the BVH reconstruction on the CPU, there is a significant communication between GPU and CPU. A GPU-based linear BVH approach was presented by Lauterbach et al. [12]. Their approach used thread and data parallelism to perform fast hierarchy operations. The linear BVH is used to check for collisions between two disjoint objects as well as self-collisions for deformable objects. Updating these LBVH over more than one GPU is difficult and leads to a huge communication overhead. Tang et al. [18] presented a GPU-based streaming algorithm for collision detection between deformable models. Their approach used BVH as culling technique and reduces the computation to generating different streams. This technique can not be easily extended to use more than one GPU.

2.2 GPU-based Collision Detection

Most modern collision detection algorithms using BVH are GPU based. However, there are some approaches which use distance fields, space subdivision or image-space techniques to improve their performance. Teschner et al. [21] presented a new approach to collision and self-collision detection of dynamically deforming objects that consist of tetrahedrons. This proposed algorithm employs a hash function for compressing a potentially infinite regular spatial grid. This hash function maps 3D cells to a hash table, thus realizing a very efficient spatial subdivision. This approach is limited to objects that consist of tetrahedrons only. Heidelberger et al. [9] proposed a simple and efficient algorithm based on Layered Depth Images (LDI). They use a discrete representation of the intersection volume which allows for volume-based collision queries. The accuracy of this method corresponds with the LDI resolution and the depth-buffer resolution. Because the LDI provides only a discrete representation of the underlying objects in some cases collision may be missed. Morvan et al. [15] presented an algorithm for proximity queries between a closed rigid object and an arbitrary mesh, for example, deformable, polygonal mesh. They sampled the distance field of the rigid object over the arbitrary mesh. One downside of this approach is that one object has to be a rigid body so, they can not simulate collisions between two soft bodies, for example. A hybrid CPU/GPU collision detection technique based on spatial subdivision was presented by Pabst et al. [16]. They prune away non-colliding parts of the scene by using an adapted highly parallel spatial subdivision method. Mainzer and Zachmann [14] presented a new approach to collision and self-collision detection which is completely GPU based. Therefore, they subdivide the scene into independent parts by fuzzy clustering. However, the thread and memory management can be improved which results in a less memory consuming implementation.

3 Sweep-Plane Technique Using PCA for Collision Detection

Due to the fact that our collision detection approach treats all objects in a scene at the same time, we do not differentiate between individual objects in the rest of this paper. Furthermore, we tread all primitives, whether from the same or from a different object, as equals which ensures that our approach detects inter-object and intra-object collisions. A majority of computer animation and simulation use triangles as their fundamental modeling primitive and therefore we choose triangles as primitive for our collision detection approach too. However, our approach can be extended to use other primitives easily.

During the collision detection process we use an adapted version of the standard Sweep and Prune approach, a 1D version, hereafter referred to as *sweep-plane* technique. We compute the bounding box for every primitive. Each bounding box spans an interval $[S_i, E_i]$ for each primitive T_i on the x-axis. Sorting all intervals along the x-axis provides information about possible colliding bounding boxes because, two bounding boxes collide iff one of the four cases $[S_a, S_b, E_b, E_a]$, $[S_b, S_a, E_a, E_b]$, $[S_a, S_b, E_a, E_b]$, or $[S_b, S_a, E_b, E_a]$ occurs (see Fig. 1).

Fig. 1a depicts an example of a downside of using bounding volumes, like AABB's or OBB's. If, for example, primitives are moving then in a significant amount of cases

a huge number of false positives may occur, when we choose any of the *fixed* world coordinate axes as sweep direction. In our case, the best sweep direction is the one, that allows projection to separate the primitives as much as possible. In order to achieve the best sweep direction, even if the primitives move through 3D spaces, we compute the *Principal Component Analysis* (PCA) [10, 13] in every frame, because the direction of the first principal component maximizes the variance of primitives, after projection [13].

The type of covariance analysis we perform is commonly used for dimension reduction and statistical analysis of data [4]. As data points we use the centroid C_i of every primitive in the scene. The covariance matrix $\mathbf{Cov} = [h_{ij}]$ for all centroid points $C_1, C_2, \ldots, C_n$ is given by

$$h_{ij} = \frac{1}{n} \sum_{k=1}^{n} \left(C_{k,i} - mean_i\right) \cdot \left(C_{k,j} - mean_j\right), \tag{1}$$

with $mean_i$ and $mean_j$ is the mean of the i-th and the j-th coordinate value of all the centroid points.

In Fig. 1b we move the direction of the first principal component on the x-axis. Now we compute the bounding box intervals $[S_i, E_i]$ and use the x-axis, more specifically the direction of the first component of the PCA, respectively, as sweep direction. Comparing Fig. 1a with Fig. 1b depicts the advantages of using the first principal component as sweep direction. The number of false positives great reduce.

As a consequence, combining sweep-plane and PCA reduces the number of primitive pairs tested for intersection and thus significantly reduces the calculation time.

3.1 Thread Management

In this Section we depict how we determine the minimal number of working (CUDA) threads, which are needed for identifying all possible colliding pairs. Additionally, we compute the worst case memory usage, i.e., the space needed to store all possible colliding primitives, at the same time.

In the first step we sort all start (S_i) and end (E_i) points of the bounding box intervals along longest principal axis. Additionally an array "Type" with the information if at position j is a start ($S_j \rightarrow \text{Type} == 1$) or an end ($E_j \rightarrow \text{Type} == 0$) point is created at the same time (see Fig. 2 upper part).

On account of the fact that we want to avoid counting overlapping bounding boxes twice, we only consider the start point (S_i) of the bounding box intervals i. If this is not taken into account, and we consider both the start (S_i) and end point (E_i) of the bounding box interval, for example in the case of $[S_a, S_b, E_a, E_b]$, we will receive two intersections. Primitive a intersects with primitive b, and vice versa. So, when we consider the start point (S_i) solely, we will get an intersection between primitive a and b only, because S_b is in the interval $[S_a,E_a]$, whilst S_a is not in the interval $[S_b,E_b]$.

To identify the number of working threads needed to do all intersection tests for a primitive, we need the amount of bounding box intersections between the bounding box of a primitive and all other bounding boxes for all primitives. Therefore, a very suitable

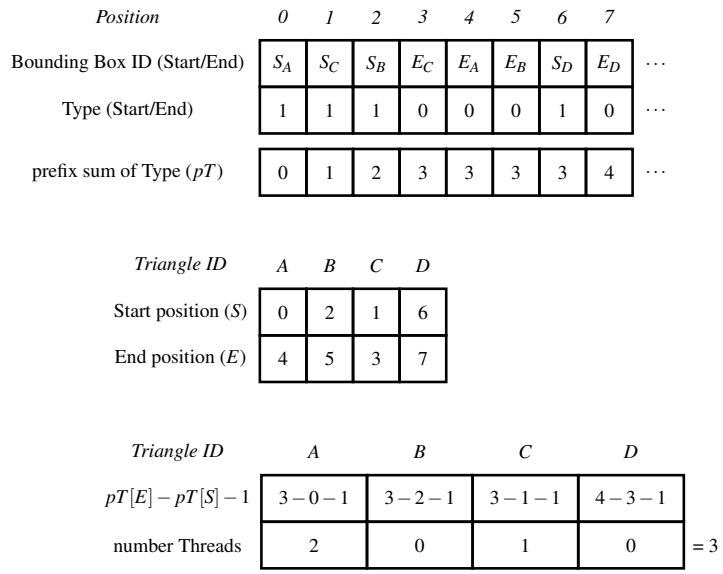


Fig. 2: Determination of the minimal number of threads needed to identify all possible colliding primitive pairs and the worst case memory usage to store all these pairs.

solution is the prefix sum algorithm from the Thrust[3] library using the "Type" array as input (see Fig. 2 upper part).

The resulting array pT can be used to compute the working threads needed for a primitive to do all possible intersection tests. Therefore, we calculate $pT[E_i] - pT[S_i] - 1$ for a primitive i which generates the number of threads needed for each primitive. The total amount of threads is equal to the number of the worst-case memory usage required to store all possible colliding primitive pairs.

4 Object Subdivision Using Fuzzy C-Means

Using the first principal component as sweep direction only, will nevertheless produce false positives, because of the dimensional reduction in the sweep-plane step. The sweep-plane technique, used to separate the primitives, projects all 3D bounding volumes to 1D points. This means, for example, that in some cases primitives of the *front side* and primitives of the *backside* of an object will be recognized as potentially colliding pairs, even if there is a large distance between them. This recognition will results in an amount of unwanted false positives.

To eliminate this kind of false positives we subdivide the scene (see Fig. 3 for some examples) into connected components using *fuzzy C-means* (FCM) algorithm [2, 17]. We use a fuzzy clustering algorithm because the primitives, which are located on the border between two clusters, have to be in both clusters. If adjoining clusters are not

[3] http://thrust.github.com/

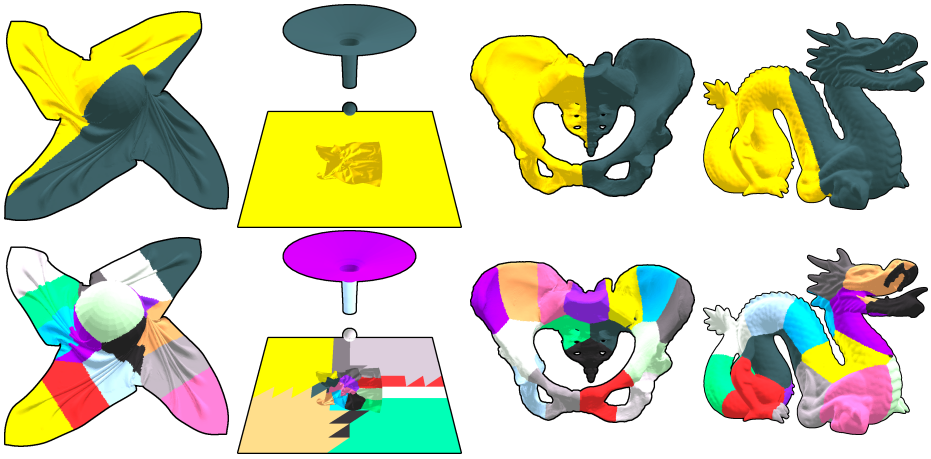


Fig. 3: Examples of some high-detail objects, partitioned by fuzzy C-means into two (top row) and 16 clusters, respectively. From left to right: Cloth on Ball (92k triangles), Funnel (18k triangles), Model of the Female Pelvis (200k triangles), and Dragon (202k triangles).

connected, then in some cases collisions across the border of the clusters would not be taken into account (see Fig. 4).

The FCM algorithm is a soft, or fuzzy, version of the well-know k-means clustering algorithm. In the classic k-means clustering algorithm, every data point is associated with only the nearest cluster center point. In the fuzzy version of the k-means algorithm, fuzzy C-means, every data point has a membership value in the range of 0 and 1 for every cluster. The algorithm tries to minimize the total error, which is the sum of the squared distances of each data point to each cluster center, if we use the euclidean distance, weighted by the membership of the data point to each cluster, for all data points.

Another advantage is that the fuzzy c-means algorithm can be run incrementally thus exploiting temporal coherence that is inherent in most real scenes. For the next iteration the algorithm uses the last computation result as starting point and iteratively minimizes the total error with the new data points. This approach takes advantage of the fact that the scene changes not very much from one frame to the next one.

Assuming we want to subdivide the scene into c clusters, we compute a sum of dispersion between the data points x_i and a set of prototypes (cluster center points) $v_1, v_2, \ldots, v_c$

$$Q = \sum_{i=1}^{c} \sum_{k=1}^{n} u_{ik}^{t} d(x_k, v_i) \tag{2}$$

with $d(x_k, v_i)$ being a given fixed distance function (e.g. Euclidean distance, or any l_p-Norm in general) between the data points x_k and v_i, the center point of cluster i.

Furthermore, Eq. 2 contains the fuzziness factor t, $t > 1$, and a partition matrix $U = [u_{ik}]$, $i = 1,2,\ldots,c$, $k = 1,2,\ldots,n$, which allocate the data points to the clusters.

A fuzziness factor $t = 1$ means that the algorithm is doing a hard clustering, like fuzzy k-means, and if $t \to \infty$ the membership will be equal in all clusters. The fuzzy clustering algorithm will iteratively optimize Eq. 2. In each iteration, all elements u_{ik} of the partition matrix U are updated using Eq. 3.

$$u_{ik} = \frac{1}{\sum_{j=1}^{c} (\frac{d_{ki}}{d_{ji}})^{\frac{2}{t-1}}} \tag{3}$$

In the next step the algorithm updates the cluster centers v_k:

$$v_k = \frac{\sum_{i=1}^{n} u_{ik}^t \cdot x_i}{\sum_{i=1}^{n} u_{ik}^t}. \tag{4}$$

The algorithm repeats these steps until the center points converge.

In the initialization phase, we choose the stop criterion much smaller than during runtime. We also limit the number of iterations for the clustering process to a fixed number at runtime because it is not necessary to get a perfect clustering. These properties ensure that the time, needed for clustering, will not rise dramatically when the scene changes drastically.

5 GPU-based Collision Detection

Algorithm 1 GPU-based Collision Detection
Each line is mapped to a massively parallel computation kernel

Input: primitives of all objects
Output: intersecting pairs of primitives
1: subdivide scene into c clusters using fuzzy C-means
2: **for all** clusters **do in parallel**
3: compute and apply PCA
4: sort AABBs along longest principle axis
5: collect all overlapping intervals
6: **for all** overlapping intervals **do in parallel**
7: **if** AABB intersect along y-axis **then**
8: do primitive-primitive intersection test
9: **end if**
10: **end for**
11: **end for**

In this section, we show how our method combines all previously introduced techniques. Algorithm 1 provides a short overview of the pipeline of our collision detection approach with the main procedures, which are mapped to a set of computation kernels.

First of all, we subdivide the whole scene into independent, overlapping parts by fuzzy clustering. Thus, we use the centroid of all primitives to decide to what cluster a primitive belongs to. Using a well-chosen stop criterion and a maximum number of

iterations for the clustering process, limits the time needed for clustering, even when the scene changes significantly. The stop criterion determines when the clustering process has reached an almost steady state, that means the movement of the cluster center point of all clusters is smaller than the predefined criterion.

Now we can do the following steps for every cluster independently. As described in Section 3, we do a PCA using the centroid of the primitives of the cluster. The result of the PCA is applied to the primitives of the cluster, which means that the direction of the first component of the PCA points along the x-axis (step "Clustering and PCA" in Fig. 7 and 8).

We are now use the x-axis as sweep-plane direction because this direction maximizes the variance of primitives after projection. Therefore, we compute the bounding box of all primitives of this cluster. We calculate the bounding box for the x-dimension and y-dimension in the same step. In this way we can exploit the fact that we can get completely coalesced memory access, which results in a lower computation time (step "Compute AABBs" in Fig. 7 and 8). We have coalesced memory access because, for example primitive k, will be adapted by the thread with $tid = k$, which can read all vertices from position k and write the result to memory at position k, and consequently there is no discontinuous read or write access to the memory. We do not compute the bounding box for the z-dimension because our approach only use the x- and y-dimension for the bounding box intersection test. We explain the fact why we omit the z-dimension bounding box intersection test in the following section.

After computing the bounding boxes for all primitives of this cluster, we sort them along the x-axis using a highly-tuned Radix Sort algorithm from the Thrust library.

The next challenge is to collect all bounding box intervals which intersects in the x-dimension. In order to avoid counting overlapping bounding boxes twice, which would increase computation time and memory needed for the collision detection, we only consider the start point (S_i) of a bounding box interval. In order to receive the required memory and the position where to put all possible colliding pairs, we use the prefix sum (or so called scan) algorithm from the Thrust library. This step, see "Collect overlapping intervals" in Fig. 7 and 8, takes up the most computation time in our collision detection algorithm. The problem is that it is not possible to access the memory completely coalesced, which slows down the computation process.

After collecting all possible colliding pairs, whose intervals overlap in x-dimension, we verify whether the bounding boxes of both primitives overlap in the y-dimension or not. We omit an bounding box overlap test for the z-dimension, because it takes more time to read the bounding box information from memory and to compare the values, than using the primitives vertices, which may potentially needed further in the case both primitives intersect, to test if the primitives overlap in the z-dimension. In the case of using a complex polygon as primitive the algorithm will not omit the z-dimension bounding box test. If that is the case, and both primitives overlap in all three dimensions, the algorithm performs a primitive-primitive intersection test.

Our collision detection algorithm compute all colliding primitive pairs and, if needed, the intersection point or line, respectively.

5.1 Accuracy and Limitations

Our collision detection algorithm will recognize all intersections between all primitives. Therefore, our approach perform bounding box intersection tests with all primitives of a cluster, to detect all colliding primitive pairs. However, in the case of significant differences in the size of the primitives, it could happen that a primitive is completely assigned to one cluster, but collides with a primitive which is completely assigned to an adjoining cluster. The reason for this is that our approach use the centroid, which represents a primitive for the clustering process. To prevent this, we have to decrease the membership value in the clustering step. This results in a higher degree of overlap between adjoining clusters (see Fig. 4). The size of the overlap has to be at least as large as the overall maximum distance from primitive's centroid to one of its vertices:

$$\max_{i=1,2,\ldots,n} \left(\max_{k=0,1,2} \left(\|C_i - vertex_{i,k}\|_2 \right) \right) \tag{5}$$

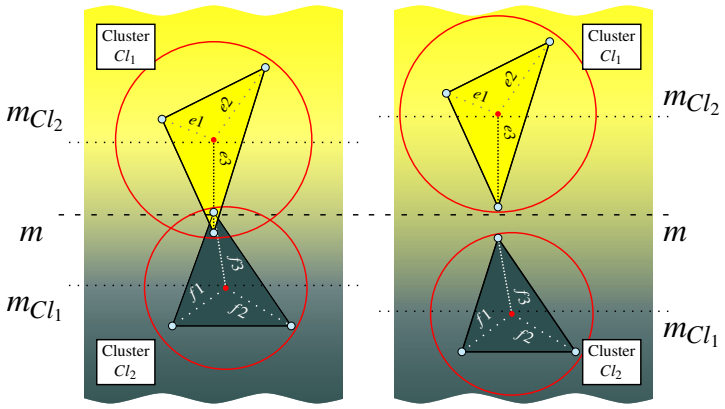


Fig. 4: The Figure shows two adjoining clusters with two triangles, one colored in yellow and one in grey. The yellow triangle is completely assigned to the yellow cluster Cl_1 and the grey triangle is completely assigned to the grey cluster Cl_2. On the left side of the Figure we choose the overlap $d(m,m_{Cl_1}) == d(m,m_{Cl_2}) < \|f_3\|_2 < \|e_3\|_2$. Accordingly, like you can see in the Figure, it is possible that the yellow triangle intersect with the grey one. In this case this collision will not be recognized by our collision detection. On the left side of the Figure we increase the overlap such that $d(m,m_{Cl_1}) == d(m,m_{Cl_2}) < \|f_i\|_2, i = 1,2,3$ and $d(m,m_{Cl_1}) == d(m,m_{Cl_2}) < \|e_i\|_2, i = 1,2,3$. As a result it is impossible that triangles, which are completely assigned to a different cluster, can intersect.

From this follows one small restriction for our approach. The large overlap between clusters can affect the performance in some scenarios, because of a higher number of

collision computations. This limitation can be avoided by virtually subdividing huge primitives. The virtual primitives are used for clustering and sorting instead of the initial primitive.

If the size of all primitives is more or less equal, than our algorithm chooses a membership value so that the overlap between adjoining clusters consists of exactly two primitives.

6 Results

We have implemented our collision detection algorithm on a NVIDIA GeForce GTX 480 using the CUDA toolkit 5.0 as development environment. Because our collision detection algorithm is purely GPU-based, components like CPU and RAM do not have effect on the running time. However, for the sake of completeness, we will provide the key data of our system. Our collision detection algorithm is implemented in C++/CUDA. The platform for benchmarking consists of a PC running Gentoo Linux with an Intel Core i5-2500K 3.30GHz CPU and 8GB of memory. For sorting and prefix computation steps we used Thrust, a parallel algorithms library.

6.1 Benchmarking

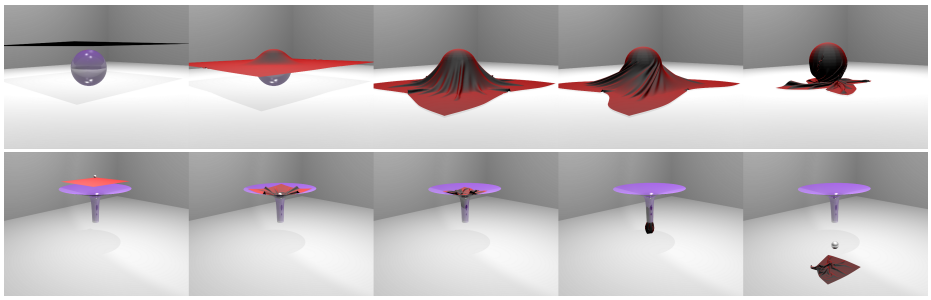


Fig. 5: The upper row shows the frames 0, 10, 40, 60 and 93 of the Cloth on Ball benchmark. The lower row shows the frames 0, 125, 200, 375 and 500 of the Funnel benchmark.

To evaluate the performance of our collision detection algorithm in different situations, we choose some often used collision detection benchmarks to compare our results against other approaches. Experiments have shown that subdividing the scene into 2 respectively 4 clusters, when the objects are far apart from each other, for a single GPU provides the best performance. Therefore, in the following benchmarks we subdivided the scene into 2 clusters.

In Fig. 6 we show the average collision detection time needed for all benchmarks compared with state-of-the-art collision detection algorithms. Our approach is slightly

slower than the CStreams [18] technique but this approach can not be easily extended to more than one GPU. Comparing our approach to the hybrid CPU/GPU collision detection techniques [16, 11] and the multi-core collision detection approach [19] shows that our technique performs better.

Bench.	Our	CSt.	Pab.	HP	MC
Cl. on Ball	20.24	18.6	36.6	23.2	32.5
Funnel	6.53	4.4	6.7	–	–

Fig. 6: Collision detection computation times in milliseconds. The timings include both external and self-collision detection. CStreams (CSt.) – GPU-based streaming algorithm for collision detection [18], Pab. – a hybrid CPU-GPU collision detection technique based on spatial subdivision [16], HP – a hybrid CPU-GPU parallel continuous collision detection [11], MC – a multi-core collision detection algorithm running on a 16 core PC [19]

Cloth on Ball In this benchmark, a cloth (92k triangles) drops down on a rotating ball (760 triangles) (see Fig. 5 upper row). Thereby the cloth has a huge number of self-collisions. This benchmark is subdivided into 93 frames. Our collision detection algorithm needs for this benchmark 20.24ms in average (see Table 6).

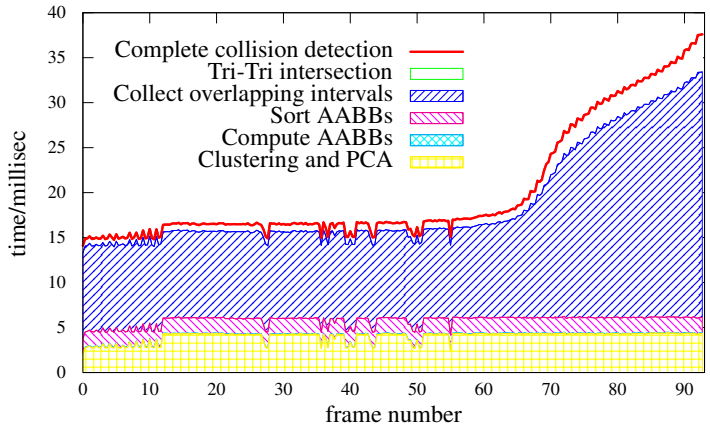


Fig. 7: Collision detection time needed for Cloth on Ball (92k triangles) Benchmark.

Fig. 7 shows that the collision detection time needed to compute all collisions from frame 60 onwards increase because the number of self-collisions increase heavily like

you can see on the Fig. 5 (upper row). Our collision detection algorithm needs more time to collect all possible colliding triangles and has to do more intersection tests between them. The benchmark, provided by the UNC Dynamic Scene Benchmarks collection, itself contains self intersecting triangles, which means that real collisions occur, like you can see at frame 93.

Funnel A cloth (14.4k triangles) falls into a funnel (2k triangles) and passes through it, due to the force applied by a ball (1.7k triangles). The ball slowly increased in volume over the time (see Fig. 5 lower row). Our collision detection algorithm needs for this benchmark 6.53ms in average (see table 6).

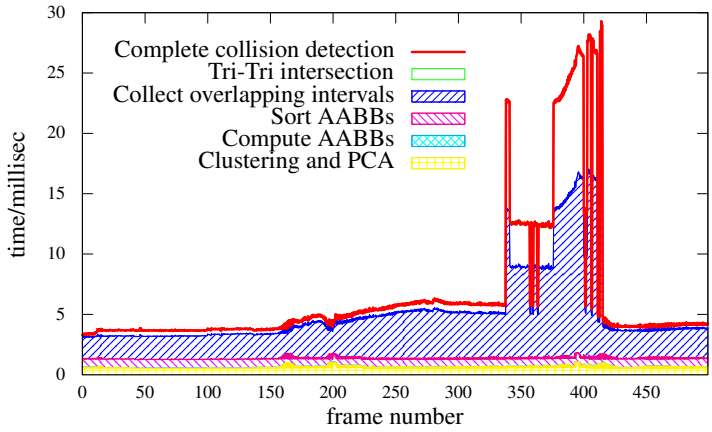


Fig. 8: Collision detection time needed for Funnel (18.5k triangles) Benchmark.

Fig. 8 depicts that the collision detection time needed to compute all collisions increase slightly between frame 150 and frame 345. In these frames the cloth hit the funnel and slides a little bit into the funnel. From frame 345 onwards the ball push the cloth trough the funnel, and produces a huge number of self-collisions which results in an higher computation time needed for collision detection.

7 Conclusions and Future Work

We presented a novel, accurate and fast collision detection algorithm which is completely GPU-based and does not require additional communication between host (CPU) and device (GPU). Our *Collision Detection Based on Fuzzy Scene Subdivision* technique can perform collision queries between rigid and/or deformable models consisting of many tens of thousands of triangles in a few milliseconds. One great advantage of this is that our method can handle the broad phase as well as the narrow phase within one single framework. Arguably, our method is much easier to implement than many

other GPU-based deformable collision detection approaches, because we do not need any BV hierarchy or other acceleration data structure. Our results show that our collision detection algorithm is as fast as state-of-the-art approaches. However, because of the subdivision process our collision detection approach can be distributed easily to more GPUs.

A multi-GPU version of our algorithm is currently being implemented to evaluate the speed improvement. We believe that we can further improve the performance of our algorithm by improving the PCA process, to reduce the number of false positives, even when the objects are deform intensive or closely intertwined. An interesting extension would certainly be to handle triangles which size significantly differ. To realize this we can use virtual subdivision for the degenerated triangles. Finally, we will extend the approach to perform other proximity queries, including distance and penetration depth or volume queries.

Acknowledgments The Cloth on Ball and Funnel simulation benchmarks are courtesy of the UNC Dynamic Scene Benchmarks collection and was provided by Naga Govindaraju, Ilknur Kabul, Stephane Redon and Simon Pabst.

References

1. D. Baraff. *Dynamic simulation of non-penetrating rigid body simulation*. PhD thesis, PhD thesis, Cornell University, 1992.
2. J.C. Bezdek. *Pattern recognition with fuzzy objective function algorithms*. Kluwer Academic Publishers, 1981.
3. J.D. Cohen, M.C. Lin, D. Manocha, and M. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 symposium on Interactive 3D graphics*. ACM, 1995.
4. C. Ericson. *Real-time collision detection*. Morgan Kaufmann, 2004.
5. S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A Hierarchical Structure for Rapid Interference Detection. *Computer Graphics*, 30(Annual Conference Series):171–180, 1996.
6. N.K. Govindaraju, D. Knott, N. Jain, I. Kabul, R. Tamstorf, R. Gayle, M.C. Lin, and D. Manocha. Interactive collision detection between deformable models using chromatic decomposition. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 991–999. ACM, 2005.
7. A. Greß, M. Guthe, and R. Klein. Gpu-based collision detection for deformable parameterized surfaces. In *Computer Graphics Forum*, volume 25, pages 497–506. Wiley Online Library, 2006.
8. Alexander Greß and Gabriel Zachmann. Object-space interference detection on programmable graphics hardware. In M. L. Lucian and M. Neamtu, editors, *SIAM Conf. on Geometric Design and Computing*, pages 311–328, Seattle, Washington, November 13–17 2003. Nashboro Press.
9. Bruno Heidelberger, Matthias Teschner, and Markus Gross. Real-time volumetric intersections of deforming objects. In *Proceedings of Vision, Modeling and Visualization*, volume 3, 2003.
10. I. Jolliffe. *Principal component analysis*. Wiley Online Library, 2005.
11. D. Kim, J.P. Heo, J. Huh, J. Kim, and S. Yoon. Hpccd: Hybrid parallel continuous collision detection using cpus and gpus. In *Computer Graphics Forum*, volume 28, pages 1791–1800. Wiley Online Library, 2009.

12. C. Lauterbach, Q. Mo, and D. Manocha. gproximity: Hierarchical gpu-based operations for collision and distance queries. In *Computer Graphics Forum*, volume 29, pages 419–428. Wiley Online Library, 2010.
13. Fuchang Liu, Takahiro Harada, Youngeun Lee, and Young J Kim. Real-time collision culling of a million bodies on graphics processing units. In *ACM Transactions on Graphics (TOG)*, volume 29, page 154. ACM, 2010.
14. David Mainzer and Gabriel Zachmann. CDFC: Collision Detection Based on Fuzzy Clustering for Deformable Objects on GPUs. In *WSCG 2013 - POSTER Proceedings*, volume 21, pages 5–8, Plzeň, Czech Republic, 7 2013. Poster.
15. T. Morvan, M. Reimers, and E. Samset. High performance gpu-based proximity queries using distance fields. In *Computer graphics forum*, volume 27, pages 2040–2052. Wiley Online Library, 2008.
16. S. Pabst, A. Koch, and W. Straßer. Fast and scalable cpu/gpu collision detection for rigid and deformable surfaces. In *Computer Graphics Forum*, volume 29, pages 1605–1612. Wiley Online Library, 2010.
17. Witold Pedrycz. *Knowledge-based clustering: from data to information granules*. Wiley-Interscience, 2005.
18. M. Tang, D. Manocha, J. Lin, and R. Tong. Collision-streams: fast gpu-based collision detection for deformable models. In *Symposium on Interactive 3D Graphics and Games*, pages 63–70. ACM, 2011.
19. Min Tang, Dinesh Manocha, and Ruofeng Tong. Mccd: Multi-core collision detection between deformable models using front-based decomposition. *Graphical Models*, 72(2):7–23, 2010.
20. M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M. p. Cani, F. Faure, N. Magnenat-thalmann, W. Strasser, and P. Volino. Collision detection for deformable objects. In *Computer Graphics Forum*, pages 61–81, 2004.
21. Matthias Teschner, Bruno Heidelberger, Matthias Müller, Danat Pomeranets, and Markus Gross. Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of Vision, Modeling, Visualization VMV'03*, pages 47–54, 2003.
22. G. Van Den Bergen. Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools*, 2(4):1–13, 1997.
23. Rene Weller and Gabriel Zachmann. Inner sphere trees for proximity and penetration queries. In *Robotics: Science and Systems Conference (RSS)*, Seattle, WA, USA, June/July 2009.