Visual Comput (2008)
DOI 10.1007/s00371-008-0246-8

ORIGINAL ARTICLE

Hanyoung Jang
JungHyun Han

Fast collision detection using the A-buffer

H. Jang · J. Han (✉)
Game Research Center, Korea University, Seoul, Korea
jhan@korea.ac.kr

Abstract This paper presents a novel and fast image-space collision detection algorithm with the A-buffer, where the GPU computes the potentially colliding sets (PCSs), and the CPU performs the standard triangle intersection test. When the bounding boxes of two objects intersect, the intersection is passed to the GPU. The object surfaces in the intersection are rendered into the A-buffer. Rendering into the A-buffer is up to eight-times faster than the ordinary approaches. Then, PCSs are computed by comparing the depth values of each texel of the A-buffer. A PCS consists of only two triangles. The PCSs are read back to the CPU, and the CPU computes the intersection points between the triangles. The proposed algorithm runs extremely fast, does not require any preprocessing, can handle dynamic objects including deformable and fracturing models, and can compute self-collisions. Such versatility and performance gain of the proposed algorithm prove its usefulness in real-time applications such as 3D games.

Keywords Real-time collision detection · A-buffer · Deformable object

1 Introduction

Collision detection refers to the process of determining if two objects are intersecting, and is a fundamental problem in many applications such as 3D games. It is a very time-consuming step in most physics simulations, and therefore is a primary target for optimization to achieve real-time performance. Numerous algorithms for collision detection have been proposed, and they are often classified into object-space algorithms and image-space algorithms.

For the last decade, the computational power of graphics hardware has made enormous leaps, in both speed and functionality, and the APIs, Direct3D and OpenGL, have accordingly evolved. Especially, DirectX 10 [3], as an API for the 4th-generation GPU, provides increased generality and flexibility through the techniques of geometry shader stage, stream output stage, etc.

This paper proposes an image-space collision detection algorithm, the key techniques of which are implemented in DirectX 10. The proposed algorithm has many strengths. It can handle both closed and open objects, and can take as input various dynamic objects including deformable and fracturing models. It does not require any preprocessing, and is simple enough to be fully hardware-accelerated. Unlike many of the image-space collision detection algorithms, the one proposed in this paper rarely suffers from readback and rendering overheads. Finally, and most importantly, it shows superior performance. The proposed algorithm is attractive for real-time applications such as 3D games.

The structure of this paper is as follows. Section 2 reviews the related works, discusses their advantages and disadvantages in depth, and derives the *raison d'être* of the approach proposed in this paper. Section 3 gives the overview of the proposed approach, and Sects. 4 and 5 present the main algorithms. Section 6 discusses several optimization techniques. Section 7 presents the test results, and finally Sect. 8 concludes the paper.

2 Related work

The collision detection algorithms based on triangulated models can be classified into two broad categories. One is the object-space approach and the other is the image-space approach. In the object-space approach, most of the proposed algorithms are accelerated by utilizing spatial data structures, i.e. the objects are hierarchically organized using the bounding volumes such as bounding spheres [16, 23], axis-aligned bounding boxes (AABBs) [26, 28], oriented bounding boxes [4], discrete orientation polytopes [18], and quantized orientation slabs with primary orientations [11]. These data structures are used to cull away portions of an object that are not in close proximity. However, the spatial data structures do not help a lot in identifying the closest features between pairs of objects in close proximity, especially for dynamic environments and deformable objects, where both of the hierarchy and bounding volumes should be updated. Some algorithms proposed for handling deformable objects either can handle simple objects only or have been designed for a limited class of objects such as cloth [20]. Another class of algorithms in the object-space approach includes the spatial hashing [25] and spatial subdivision methods [9].

It is worth analyzing the object-space algorithm, named steaming AABB, found in the work of Zhang and Kim [29]. Using the leaf nodes of a mesh's AABB tree, the 1D stream of AABBs is constructed. In the stream, an AABB comprises a set of triangles, and is represented by min/max vertices. Then, all possible pairwise combinations from two AABB streams are examined for possible overlap. The Boolean results of the overlap test constitute the potentially colliding sets (PCSs), and are read back to the CPU for a primitive-level intersection test. As the input mesh models deform, the AABB streams are updated. The proposed algorithm runs quite fast and is accurate. However, it is not suitable for fracturing objects. When the triangles in an AABB fall apart due to fracture, the AABB stream often has to be restructured. Such restructuring hampers real-time performance. Without restructuring, the AABB would become unacceptably large, and the number of PCSs is accordingly increased. As a result, the readback overhead is increased, and the CPU is burdened with extensive computation, i.e. real-time performance is hampered.

The image-space approach typically measures the volumetric ranges of objects along the viewing direction, and then compares the ranges to detect collision. Since the seminal work of Shinya and Forgue [24], various algorithms for the image-space approach have been proposed, attempting to maximally utilize the powerful rasterization capability of the GPUs [1, 14, 22, 27]. Recent efforts in the image-space approach include the works of Heidelberger et al. [12, 13]. They proposed to compute the layered depth images (LDIs), one for each object, where an LDI stores entry and leaving points of parallel viewing rays with respect to an object. Then, collision is detected through Boolean intersection on LDIs. Those approaches can handle concave objects. LDI generation also requires a considerable amount of time for objects with complex geometry, due to the rendering and readback overhead. Jang et al. [17] proposed an algorithm, named alternate surface peeling, where the object surfaces are rendered layer by layer, and the depth disparities among the surfaces are analyzed to compute the PCSs.

In general, the advantages of the image-space approach can be listed as follows. Unlike the object-space approach which requires non-trivial preprocessing for computing bounding volumes and their hierarchy, the image-space approach rarely requires preprocessing. Partly due to the absence of the preprocessing, the image-space approach is easy to implement. It can also effectively handle deformable objects and dynamic environments. Moreover, it usually employs the GPU which has been evolving at a rate faster than Moore's law, while the object-space approach usually performs the collision tests on the CPU. Therefore, an efficiently designed algorithm can show superior performance.

However, the image-space approach also reveals disadvantages. First of all, its effectiveness is limited by the image-space resolution, and therefore the image-space approach often misses overlapping primitives. Therefore, it is not suitable for applications requiring *accurate* collision detection, but can be used for *approximate* collision detection. An example application that can be satisfied with such approximate collision detection is the 3D game.

Virtually all of the image-space algorithms proposed so far perform collision tests using both the CPU and GPU, and suffer from the limited bandwidth between them, i.e. the readback problem. In the LDI-based algorithm of Heidelberger et al. [12, 13], for example, the CPU reads the LDIs from the GPU's back buffers, and then tests the LDIs for Boolean intersection. Because of the limited bandwidth, the sampling resolution (LDI resolution) is usually made low, 32×32 through 128×128. Note that, however, the accuracy of the collision detection is governed by the LDI precision, and low resolutions lead to inaccurate detection of collision for complex objects.

As an effort to alleviate the readback problem, Govindaraju et al. [8] proposed an algorithm named CULLIDE.

(There have been a few extensions of the original CULLIDE algorithm [6, 7]. In this paper, we collectively call all of them just 'CULLIDE.'). The CULLIDE algorithm tests the visibility of an object O with respect to a set S of objects. A PCS is computed through a hardware visibility query, which checks if any part of O is occluded by S. If not occluded, O does not collide with S, and therefore is not included in the PCS. CULLIDE usually partitions an object into a set of sub-objects at the preprocessing stage, and the sub-object hierarchy is traversed for the visibility query at run-time. The number of the sub-objects determines the number of rendering calls. An efficient preprocessing technique for handling self-collision using the CULLIDE algorithm, named chromatic decomposition [5], has been suggested, but it also has limitations. For example, the objects to be tested for collision are limited to polygonal meshes with fixed connectivity. For a fracturing model the topology of which may vary frame by frame, the time-consuming preprocessing has to be executed per each frame.

Like the works of CULLIDE in the image-space approach and streaming AABB in the object-space approach, the proposed algorithm uses the GPU to compute the PCSs. Unlike CULLIDE and streaming AABB, however, the proposed algorithm makes the GPU take the major computation load off the CPU. As a result, each PCS passed to the CPU is of the minimum size, i.e. a pair of triangles. The computing capability of the 4th-generation GPU makes the strategy quite attractive. In addition, the proposed algorithm does not suffer from the readback overhead problem, and can freely handle deformable and fracturing models. These are the distinctions from CULLIDE and streaming AABB.

3 Overview of the approach

The algorithm proposed in this paper utilizes both the CPU and GPU for collision detection, and the coordination between them is illustrated in Fig. 1. Each object in the scene is associated with an axis-aligned bounding box (AABB). If the AABBs of two objects O_1 and O_2 intersect, the intersection is passed to the GPU as a *region of interest* (*ROI*). An example is shown in Fig. 2a. Given an ROI, the GPU computes the potentially colliding sets (PCSs). A PCS is a pair of triangles, one from O_1 and the other from O_2. Two PCSs are obtained in Fig. 2b. Given such PCSs, the CPU performs the traditional triangle intersection test to obtain the intersection points.

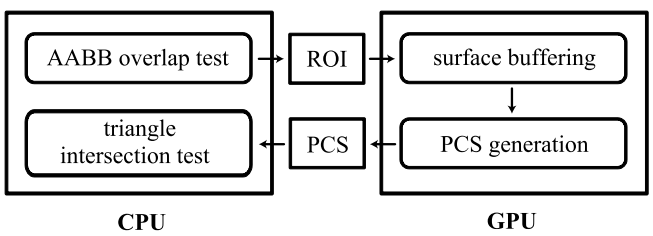

Fig. 1. System architecture

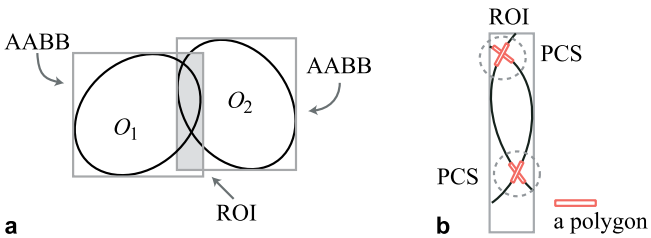

Fig. 2. a Object AABBs and the ROI, **b** PCSs in the ROI

The coordination of CPU and GPU aims at both performance and accuracy. The streaming processor is suitable for performing a simple operation with massive data, and hence the GPU can prune away non-intersecting triangles quickly. On the other hand, the collision information provided by the CPU is in the triangle level accuracy, and enables the collision response module to perform realistic physical simulation.

The framework is similar to that of CULLIDE [8], and also that of streaming AABB [29], in the sense that PCSs are computed by the GPU and passed to the CPU. However, the key feature that makes our algorithm distinguishable from CULLIDE and streaming AABB is that a PCS consists of exactly two triangles while a PCS in CULLIDE and streaming AABB comprises many triangles. When a PCS consists of n triangles, we need $O(n^2)$ triangle intersection tests. Note that, per PCS, the CPU in our method performs just a single test for triangle intersection. Our strategy is to maximally utilize the computing power of the GPU and pass the minimum-sized PCSs to the CPU so as to reduce the CPU's computational load. The revolutionary growth of the GPU's computing power makes this strategy very attractive, as will be discussed in Sect. 4.

In the framework of Fig. 1, the CPU's tasks, i.e. AABB overlap test and triangle intersection test, are quite obvious, and existing methods are adopted for them. The major contributions of this paper are found in the GPU's tasks, surface buffering and PCS generation, which are discussed in Sects. 4 and 5, respectively.

4 Surface buffering

In the rendering area, a few algorithms for saving *multi-layer depth images* have been proposed [2, 19]. The al-

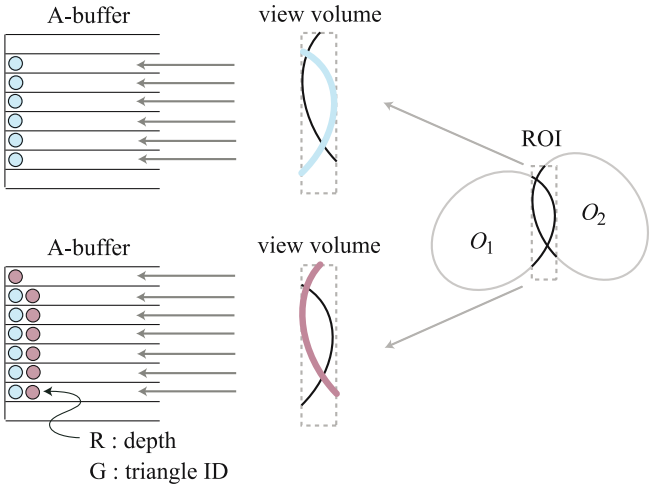

Fig. 3. Surface rendering into the A-buffer

gorithms have adopted the so-called *K-buffer* which can capture multiple layers in a single pass, using the stencil routing method. Each pixel of the K-buffer stores a list of fragments. Myers has presented an extension of the K-buffer, named the *A-buffer*[1] [21]. Currently, the A-buffer captures up to eight fragments in a rendering pass, and is supported by DirectX 10 only. The A-buffer can be easily implemented, and is quite fast.

Figure 3 illustrates how the surfaces, in the ROI of Fig. 2b, are rendered into the A-buffer. Orthographic projection is used for rendering, and the ROI is set to its view volume. Then, O_1 is processed to make the light fragments in the A-buffer, as shown in Fig. 3. Next, O_2 is processed and the dark fragments are accumulated into the A-buffer. Note that such a multi-layer depth image is created in a single pass of rendering. As the A-buffer captures up to eight fragments in a pass, the rendering process is up to eight times faster than those of the other GPU-based image-space algorithms.

As illustrated at the bottom of Fig. 3, a fragment in the A-buffer contains two values for surface information: a *depth* value in the red channel and *triangle ID* in the green channel. The triangle ID indicates the owner triangle of the fragment, and is allocated using the *SV_PrimitiveID* semantic in DirectX 10 [3]. To distinguish between the fragments from O_1 and those from O_2, a bit in the triangle ID is used as a flag.

In the image-space algorithm described above, more accuracy can be obtained if we increase the A-buffer size because the surfaces in the ROI are then sampled more densely. This issue will be discussed in Sect. 7.

[1] The K-buffer is often taken as a hazardous operation, because it uses the render target texture as a read-modify-write texture and the GPUs process multiple fragments at the same time. The A-buffer is a non-hazardous version of the K-buffer.

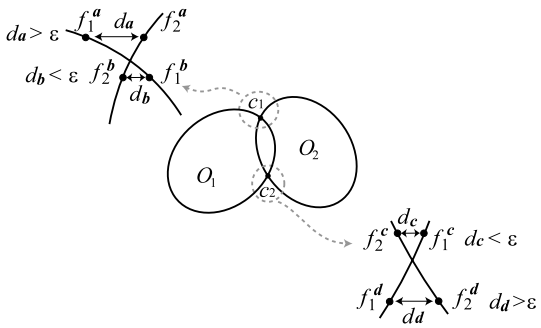

Fig. 4. PCS generation using the distance threshold

5 PCS Generation

In order to compute PCSs, a simple test is invoked for each pixel of the A-buffer. Out of the fragment list, one from O_1, named f_1, and the other from O_2, named f_2, are selected. If the distance d between f_1 and f_2 is less than the threshold ϵ, the triangle IDs are retrieved from the fragments, and then passed to the CPU as a PCS. In Fig. 4, the fragment pairs (f_1^b, f_2^b) and (f_1^c, f_2^c) constitute the PCSs.

The PCSs are initially recorded in a render target texture. The PCSs should be *read back* to the main memory so that the CPU can compute the intersection points, c_1 and c_2, in Fig. 4. The readback operation is expensive on commodity graphics hardware, and therefore readback of the entire texture significantly degrades the system performance. Note that, in general, only a small fraction of the texels in the render target texture contains the information of PCSs while the remaining texels contain no information at all. It is desirable to extract only the PCSs. We call the process *stream reduction*, which is a special algorithm for removing unnecessary data elements or changing the sequence of the elements.

Stream reduction is a fundamental issue in many GPGPU applications, and a few works on it have been reported [10, 15]. Fortunately, the *geometry shader* in DirectX 10 enables us to trivially remove unwanted elements from a stream of input. The reduced elements can also be written to a memory resource which can be copied to a staging resource for readback to the CPU.

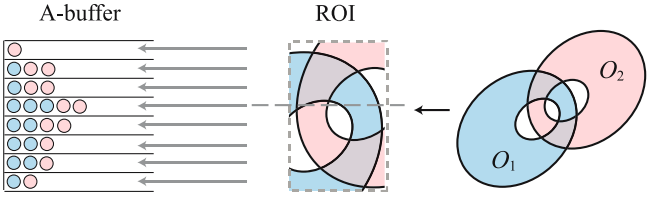

Fig. 5. Surface buffering for complicated collisions

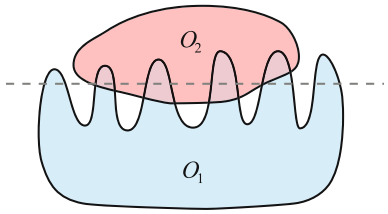

Fig. 6. A-buffer overflow

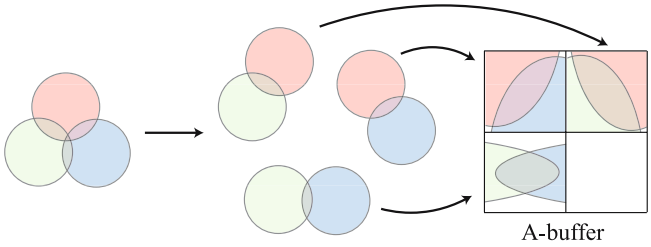

Fig. 7. Partitioning of the A-buffer into a set of cells

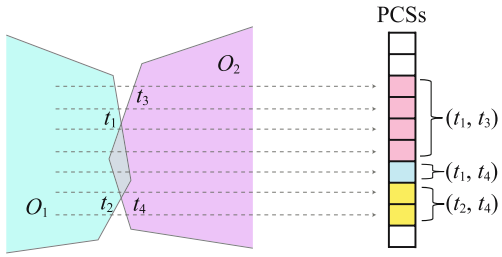

Fig. 8. Multiple instances of a single PCS are often produced

In DirectX 10 with shader model 4.0, we have two rendering passes for PCS generation. The first pass computes the PCSs in the pixel shader, and the second pass performs stream reduction in the geometry shader. Note that, in DirectX 10.1 with shader model 4.1, those can be performed in a single pass with the vertex shader and geometry shader, because the vertex shader can access multi-sample textures.

So far, we have used a simple example and discussed the skeleton of the proposed algorithm. Figure 5 shows a more complicated case, where more than two fragments are accumulated in a pixel position of the A-buffer. At the 4th pixel, five fragments are accumulated, three from O_1 and two from O_2. In this case, all possible fragment pairs from O_1 and O_2, six pairs in the 4th pixel, are checked in the PCS generation phase.

Recall that the contemporary A-buffer captures up to eight fragments in a rendering pass. As shown in Fig. 6, however, more than eight fragments may be created for a pixel position in the A-buffer. Such an overflow can be handled by creating multiple A-buffers. Fortunately, our experiment reports that eight fragments per pixel are enough for almost all cases, i.e. a single A-buffering is enough.

6 Optimization techniques

6.1 A-buffer partitioning

Collision detection among more than two objects can be handled by reformulating it into multiple instances of collision detection between two objects, as shown in Fig. 7. Such multiple instances can be processed simultaneously by partitioning the A-buffer. In Fig. 7, each ROI is scaled and translated to fit to a cell of the A-buffer. To prevent an ROI from affecting the other cells, the scissor test is enabled for each cell. A-buffer partitioning significantly enhances the overall system performance because the overhead of context switching is reduced[2].

[2] A-buffer partitioning reduces the number of the batch calls for initialization. The A-buffer has eight sub-pixels in a pixel, and each sub-pixel has a different stencil value. To assign the stencil values to all sub-pixels, a call for the stencil buffer initialization and subsequent seven renderings are needed.

6.2 PCS coherence

The neighboring texels in the render target texture may often contain the same PCS information. In Fig. 8, seven PCSs are computed between O_1 and O_2, but among them, for example, four PCSs are identical, i.e. four instances of (t_1, t_3) indicate that the triangles t_1 of O_1 and t_3 of O_2 would collide. Note that the four PCSs appear continuously. This is called PCS coherence. Our algorithm eliminates such duplicate PCSs through a small effort of checking the neighbors of a PCS in the CPU, and avoids unnecessary tests for triangle intersection.

7 Experimental result and analysis

The proposed algorithm has been implemented in DirectX 10 with shader model 4.0 on a PC with a 2.4 GHz Intel Core2 CPU and 2 GB memory. The PC is equipped with a NVIDIA GeForce 8800GTS GPU which has a 500 MHz core, 640 MB memory, and PCI-Express 1.1×16 interface. Various functionalities of DirectX 10 are exploited, such as multisample antialiasing (MSAA) texture for A-buffer and SV_Primitive semantic for PCS generation.

7.1 Performance analysis

Figure 9 shows a series of screen shots for a scene of multiple collisions. Table 1 shows the performance statistics of the test in Fig. 9. The GPU time is spent for A-buffer initialization, surface buffering, and PCS generation. The

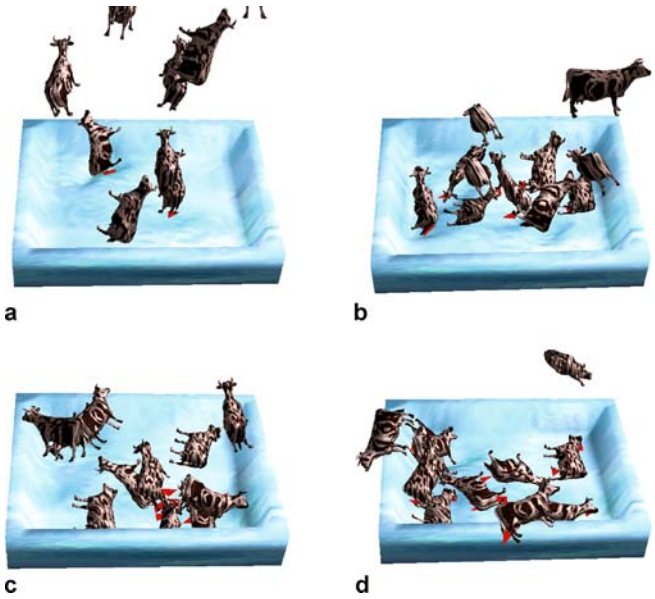

Fig. 9a–d. A scene of 11 cows (each with 3.3 K triangles) falling on the ground (of 1.9 K triangles)

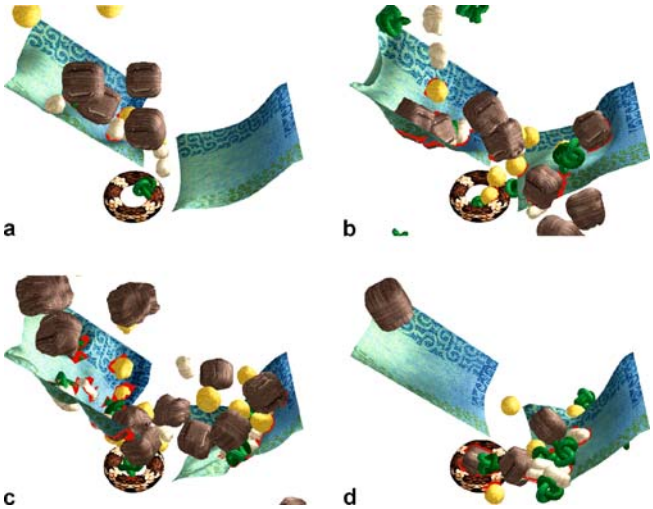

Fig. 10a–d. Multiple deforming objects (56 K triangles in total): a torus (of 840 triangles), two pieces of cloth (each with 1.2 K triangles), 16 torus-knots (each with 1.3 K triangles), 24 spheres (each with 320 triangles), 21 chamfer-boxes (each with 800 triangles), and 19 capsules (each with 360 triangles)

Table 1. Performance evaluations for Fig. 9 (times in ms)

Fig. 9	PCSs	Collisions	GPU	CPU	Total
a	58	16	0.98	0.32	1.30
b	238	52	2.18	0.61	2.79
c	976	132	2.22	0.75	2.97
d	1688	197	2.35	0.86	3.21

CPU time is spent for the AABB overlap test, readback, and triangle intersection test. The total processing time is proportional to the model complexity and the number of PCSs.

Figure 10 shows a series of screen-shots for a scene with a variety of deforming objects. The performance of the A-buffer algorithm is compared with that of CULLIDE. As discussed in Sect. 2, CULLIDE requires off-line preprocessing for defining sub-objects. In the experiments[3], an object is partitioned into a set of sub-objects, each of which contains 15 triangles. In Fig. 11a, CULLIDE shows an outstanding performance at the initial stage, which corresponds to the scene in Fig. 10a, with few collisions. When multiple collisions start to occur, however, CULLIDE's performance degrades significantly. It is caused by both the CPU and GPU. Recall that a PCS passed to the CPU is a set of triangles, not a pair of triangles. Given n triangles in a PCS, n^2 triangle intersection tests should be performed. Figure 11b illustrates the performance gap between A-buffer and CULLIDE in the triangle intersection test. In PCS computation on the GPU side, the A-buffer algorithm also shows a better performance than CULLIDE because, in principle, a rendering call

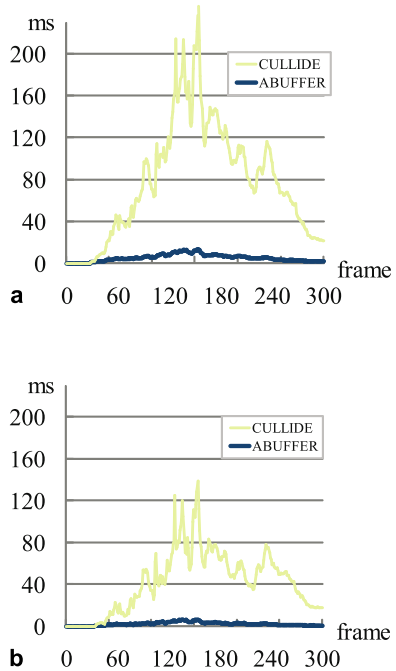

Fig. 11a,b. Performance comparisons: **a** total time, **b** time for triangle intersection test

[3] The visibility test is done three times along the principal axes. For the purpose of optimization, rendering is done in the partitioned areas of the render target texture, which is similar to the A-buffer partitioning in our approach.

is needed for an object in the A-buffer algorithm while multiple calls are needed in CULLIDE due to the existence of sub-objects.

Figure 12 shows a test with deforming and fracturing objects, where the cloth patch is being torn by the falling sharp objects. When the cloth is torn, its mesh connectivity is changing. Further, new triangles are dynamically added in the areas being split, to achieve more natural simulation. The collision detection algorithm proposed in this paper does not require any extra time for handling such deforming and fracturing objects. In contrast, handling this kind of real-time simulation is not possible in the ordinary bounding volume hierarchy methods including streaming AABB.

Recall that, in the proposed approach, the A-buffer saves all fragments of objects in a pixel position. It makes the proposed algorithm extend directly to detection of self-collisions. Figure 13 shows a simple example of self-collision detection with a cloth patch of 22×22 vertices. This simulation is performed within 0.5 ms.

7.2 Accuracy

A well-known problem of the image-space collision detection algorithms is that its accuracy is limited by the image-space resolution and the viewing direction. As a result, an image-space algorithm often misses overlapping primitives. Suppose that q is the number of colliding triangle pairs and p is that computed by a collision detection algorithm. Then, p/q is defined to be the *hit ratio*, and $1 - p/q$ is the *miss ratio*. Figure 14 shows the hit ratio curve for the scene of Fig. 10, where a 64×64 size A-buffer is assigned per ROI[4].

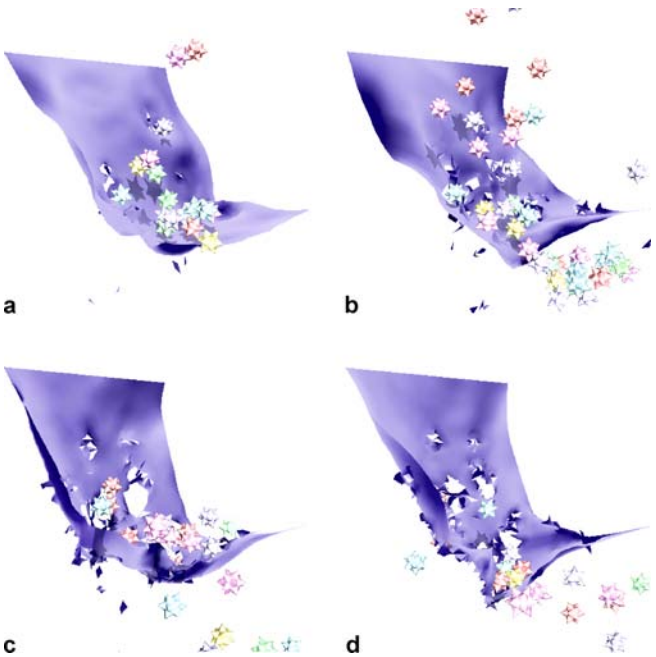

Fig. 12a–d. Collision detection among deforming and fracturing models: a cloth patch (of 1.6 K triangles) and 40 sharp objects (each with 2.1 K triangles)

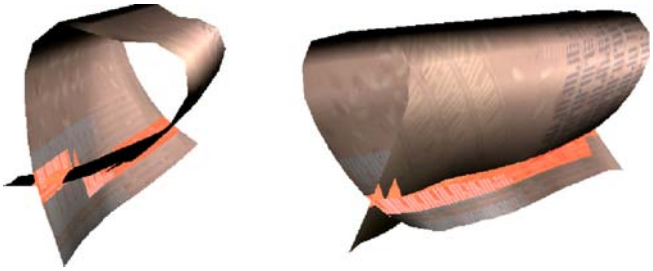

Fig. 13. Self-collision detection: The red-colored triangles represent collisions

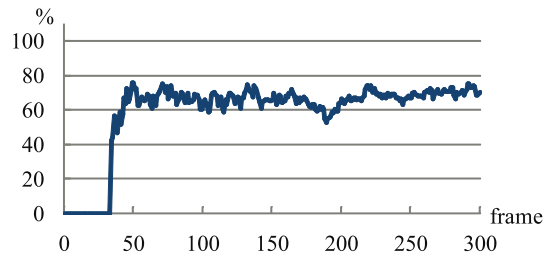

Fig. 14. Hit ratio curve for Fig. 10

Fig. 15. Collision between bunny (2.7 K triangles) and dragon (2.9 K triangles)

Figure 15 shows a worst-case example, where the bunny and dragon are deeply interpenetrated and therefore the ROI size becomes large. If a fixed-size A-buffer is assigned to an ROI independently of the ROI size, the precision of sampling becomes lower as the ROI grows. With a 64×64 size A-buffer, 307 pairs of triangles actually collide, but our algorithm detects 134 pairs out of 928 PCSs, i.e. the miss ratio is about 56%. An obvious solution to this problem is to enlarge the size of the A-buffer.

Figure 16 shows the graphs of the miss ratio and collision detection time, as functions of the A-buffer size. As

[4] For the same scene, CULLIDE achieves a higher hit ratio, about 95% on average, as it is hard for the visibility test of CULLIDE to miss colliding sub-objects. Recall, however, that CULLIDE requires expensive CPU computation, i.e. n^2 triangle intersection tests, discussed in Sect. 7.1.

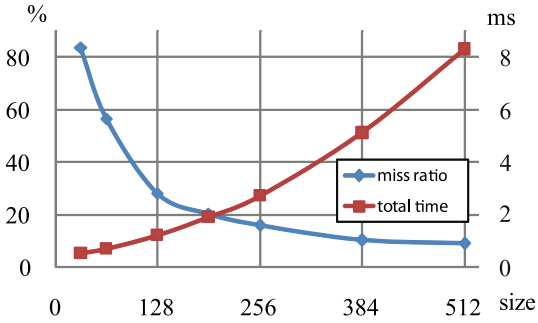

Fig. 16. The graphs of the miss ratio and the collision detection time imply the trade-off between accuracy and efficiency

the A-buffer size increases, the miss ratio drops rapidly. For example, with a 128×128 size A-buffer, 221 colliding pairs are obtained, and the miss ratio falls under 30%. As the A-buffer size increases, however, the collision detection time also increases though the increase is gradual. This is because both the A-buffering rendering cost and the number of PCSs increase.

In actuality, the graphs in Fig. 16 show the trade-off between accuracy and efficiency. Depending on the requirements of the applications at hand, the most appropriate size of A-buffer can be determined. For example, the miss ratios over 50% would be admissible in 3D games. This is because game players are generous in moderate inaccuracy if real-time performance is guaranteed. In such a case, the A-buffer size can be made small.

A-buffer partitioning brings us a useful tool for accuracy control, i.e. the precision of the collision detection can be made proportional to the degree of the user's visual perception. Suppose that you have two ROIs in a scene: one is closer to the viewpoint and the other is located far. Then, it is a good strategy to assign a small cell of the A-buffer to the far ROI, and a large cell to the closer one. Even though the far ROI leads to fairly inaccurate collision detection and response, the user may not be able to perceive it.

8 Conclusion

This paper presented an efficient image-space algorithm for real-time collision detection. In the current implementation, shader programs compute the PCSs, and the CPU performs the primitive-level intersection test. Thanks to the computing power and various functionalities of the contemporary GPU, the algorithm can handle a variety of dynamic objects including fracturing meshes, and perform fast self-collision detection. This method rarely suffers from both rendering and readback overheads, and in the near future the performance will be upgraded by DirectX 10.1 and PCI-Express 2.0. The experimental results show the feasibility of the shader-based collision detection and its performance gain in real-time applications such as 3D games.

The proposed algorithm also has disadvantages. It works in a synchronous mode between CPU and GPU, i.e. the CPU waits until the GPU computes the PCSs. The problem of missing collisions for complex objects has to be resolved. The proposed algorithms are being extended to overcome these disadvantages.

Acknowledgement This research was supported by MKE, Korea under ITRC IITA-2008-(C1090-0801-0046), and also by grant No. R01-2006-000-11297-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

References

1. Baciu, G., Wong, S.K., Sun, H.: RECODE: An image-based collision detection algorithm. J. Vis. Comput. Animation **10**(4), 181–192 (1999)
2. Bavoil, L., Callahan, S.P., Lefohn, A., Comba, J.L.D., Silva, C.T.: Multi-fragment effects on the gpu using the k-buffer. In: I3D '07: Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games, pp. 97–104. ACM, New York, NY (2007)
3. Blythe, D.: The direct3d 10 system. ACM Trans. Graph. **25**(3), 724–734 (2006)
4. Gottschalk, S., Lin, M.C., Manocha, D.: OBBTree: A hierarchical structure for rapid interference detection. Computer Graphics (SIGGRAPH '96 Proceedings) **30**, 171–180 (1996)
5. Govindaraju, N.K., Knott, D., Jain, N., Kabul, I., Tamstorf, R., Gayle, R., Lin, M.C., Manocha, D.: Interactive collision detection between deformable models using chromatic decomposition. ACM Trans. Graph. **24**(3), 991–999 (2005)
6. Govindaraju, N.K., Lin, M.C., Manocha, D.: Fast and reliable collision culling using graphics hardware. In: VRST '04: Proc. of ACM Symposium on Virtual Reality Software and Technology, pp. 2–9. ACM, New York, NY (2004)
7. Govindaraju, N.K., Lin, M.C., Manocha, D.: Quick-CULLIDE: Fast inter- and intra-object collision culling using graphics hardware. In: VR '05: Proc. of IEEE Virtual Reality 2005, pp. 59–66. IEEE Computer Society, Washington, DC (2005)
8. Govindaraju, N.K., Redon, S., Lin, M.C., Manocha, D.: CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In: HWWS '03: Proc. of ACM SIGGRAPH/Eurographics Conference on Graphics Hardware, pp. 25–32. Eurographics Association, Aire-la-Ville, Switzerland (2003)
9. Grand, S.L.: Broad-phase collision detection with CUDA. In: Nguyen, H. (ed.) GPU Gems 3, chap. 32, pp. 697–722. Addison-Wesley (2007)
10. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with CUDA. In: Nguyen, H. (ed.) GPU Gems 3, chap. 39, pp. 851–876. Addison-Wesley (2007)
11. He, T.: Fast collision detection using QuOSPO trees. In: I3D '99: Proc. of the 1999 Symposium on Interactive 3D Graphics, pp. 55–62. ACM, New York, NY (1999)
12. Heidelberger, B., Teschner, M., Gross, M.: Realtime volumetric intersections of deforming objects. In: Ertl, T. (ed.) VMV '03: Proc. of the 2003 Vision, Modeling, and Visualization Conference,

pp. 461–468. Aka GmbH, München (2003)
13. Heidelberger, B., Teschner, M., Gross, M.: Detection of collisions and self-collisions using image-space techniques. J. Winter School Comput. Graph. **12**(3), 145–152 (2004)
14. Hoff, K.E., Zaferakis, A., Lin, M.C., Manocha, D.: Fast 3D geometric proximity queries between rigid and deformable models using graphics hardware acceleration. Tech. Rep. TR02–004, Department of Computer Science, University of North Carolina (2002)
15. Horn, D.: Stream reduction operations for gpgpu applications. In: Pharr, M. (ed.) GPU Gems 2, chap. 36, pp. 573–589. Addison-Wesley (2005)
16. Hubbard, P.M.: Interactive collision detection. In: Proc. of IEEE Symposium on Research Frontiers in Virtual Reality 1993, pp. 24–32. IEEE Computer Society, San Jose, CA (1993)
17. Jang, H., Jeong, T., Han, J.: Image-space collision detection through alternate surface peeling. In: ISVC '07: Proc. of International Symposium on Visual Computing. Lect. Note Comput. Sci., vol. 4841, pp. 66–75. Springer (2007)
18. Klosowski, J.T., Held, M., Mitchell, J.S.B., Sowizral, H., Zikan, K.: Efficient collision detection using bounding volume hierarchies of *k*-DOPs. IEEE Trans. Vis. Comput. Graph. **4**(1), 21–36 (1998)
19. Liu, B., Wei, L.Y., Xu, Y.Q.: Multi-layer depth peeling via fragment sort. Tech. Rep. MSR-TR-2006-81, Microsoft Research Asia (2006)
20. Mezger, J., Kimmerle, S., Etzmuss, O.: Hierarchical techniques in collision detection for cloth animation. J. Winter School Comput. Graph. **11**(2), 322–329 (2003)
21. Myers, K., Bavoil, L.: Stencil routed A-buffer. In: SIGGRAPH '07: ACM SIGGRAPH 2007 Sketches, p. 21. ACM, New York, NY (2007)
22. Myszkowski, K., Okunev, O.G., Kunii, T.L.: Fast collision detection between computer solids using rasterizing graphics hardware. Visual Comput. **11**(9), 497–511 (1995)
23. Palmer, I., Grimsdale, R.: Collision detection for animation using sphere-trees. Comput. Graph. Forum **14**(2), 105–116 (1995)
24. Shinya, M., Forgue, M.: Interference detection through rasterization. J. Vis. Comput. Animation **2**(4), 131–134 (1991)
25. Teschner, M., Heidelberger, B., Mueller, M., Pomeranets, D., Gross, M.: Optimized spatial hashing for collision detection of deformable objects. In: Ertl, T. (ed.) VMV '03: Proc. of the 2003 Vision, Modeling, and Visualization Conference, pp. 47–54. München (2003)
26. Van den Bergen, G.: Efficient collision detection of complex deformable models using aabb trees. J. Graph. Tools **2**(4), 1–13 (1997)
27. Vassilev, T., Spanlang, B., Chrysanthou, Y.: Fast cloth animation on walking avatars. Comput. Graph. Forum **20**(3), 260–267 (2001)
28. Zachmann, G., Felger, W.: The boxtree: Enabling realtime and exact collision detection of arbitrary polyhedra. In: Proc. of Workshop on Simulation and Interaction in Virtual Environments 1995, pp. 104–113 (1995)
29. Zhang, X., Kim, Y.: Interactive collision detection for deformable models using streaming aabbs. IEEE Trans. Vis. Comput. Graph. **13**(2), 318–329 (2007)

HANYOUNG JANG is a Ph.D. student at Korea University. He received both his M.S. and B.S. degrees in the College of Information and Communications at Korea University. Since 2005, he has been working in the fields of collision detection and accessibility analysis for robotics. Currently, his primary research interest lies in real-time physics simulation.

JUNGHYUN HAN is an associate professor in the Department of Computer Science and Engineering at Korea University, where he directs the Interactive 3D Media Laboratory and Game Research Center supported by the Korea Ministry of Culture, Sports, and Tourism. Prior to joining Korea University, he worked at the School of Information and Communications Engineering of Sungkyunkwan University, in Korea, and at the Manufacturing Systems Integration Division of the US Department of Commerce National Institute of Standards and Technology (NIST). Dr. Han is currently serving as the project manager of the digital content area in Korea Ministry of Information and Communication. He received a B.S. degree in Computer Engineering at Seoul National University, an M.S. degree in Computer Science at the University of Cincinnati, and a Ph.D. degree in Computer Science at USC. His research interests include real-time simulation and animation for games.