Image-Space Collision Detection Through Alternate Surface Peeling

Han-Young Jang, TaekSang Jeong, and JungHyun Han

Game Research Center, College of Information and Communications, Korea University, Seoul, Korea

Abstract. This paper presents a new image-space algorithm for real-time collision detection, where the GPU computes the potentially colliding sets, and the CPU performs the standard triangle/triangle intersection test. The major strengths of the proposed algorithm can be listed as follows: it can handle dynamic models including deforming and fracturing objects, it can take both closed and open objects, it does not require any preprocessing but is quite efficient, and its accuracy is proportional to the visual sensitivity or can be controlled on demand. The proposed algorithm would fit well to real-time applications such as 3D games.

1 Introduction

Collision detection is a fundamental problem in many applications such as 3D games, virtual reality, medical simulation, physically-based simulation, and robotics. A lot of algorithms for collision detection have been proposed. The algorithms based on triangulated models can be classified into two broad categories. One is *object-space approach* and the other is *image-space approach.*

In the object-space approach, most of the proposed algorithms are accelerated by utilizing spatial data structures which are often hierarchically organized and are based on bounding volumes [1,2]. A state-of-the-art algorithm in the object-space approach is found in the work of Zhang and Kim [3]. The algorithm performs AABB overlap test which is accelerated by GPU. The algorithm runs quite fast enough to handle deformable objects with high accuracy. 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.

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 [4], various algorithms for image-space approach have been proposed, attempting to maximally utilize the powerful rasterization capability of the GPUs. Recent efforts in the image-space approach include the work of Heidelberger *et al.* [5] and Govindaraju *et al.* [6]. The approach of Heidelberger *et al.* can handle concave objects, but requires a considerable amount of time for collision detection of objects with complex geometry, due to the rendering and readback overhead. As an effort to alleviate the readback problem, Govindaraju *et al.* proposes CULLIDE algorithm, which can reduce the readback overhead using occlusion query. However CULLIDE algorithm requires an off-line setup stage, which defines the sub-objects used for each occlusion query. In case of a fracturing model, the topology may vary frame by frame. The time-consuming pre-processing has to be executed per each frame.

This paper proposes a new algorithm for the image-space approach. Its key components are implemented in shader programs. Like many of the image-space collision detection algorithms, the proposed algorithm can handle deformable models. The unique strength of the proposed algorithm is its versatility: it can handle both closed and open objects, and more importantly it can take as input various dynamic models including fracturing meshes. Moreover, our algorithm overcomes not only the readback and rendering overhead but also the accuracy problem of the ordinary image-space approach. The proposed algorithm does not require any pre-processing, is simple to implement, and shows superior performance. Such an algorithm is attractive for real-time applications such as 3D games.

2 Overview of the Approach

This paper proposes to compute potentially colliding sets(PCSs) using GPU and leave the primitive-level intersection test to CPU. This framework is similar to the state-of-the-art work in the image-space approach, CULLIDE [6], and that in the object-space approach, the work of Zhang and Kim [3]. Unlike CULLIDE, however, the proposed algorithm computes the PCSs always at the primitive level, and further it resolves the major drawbacks of CULLIDE discussed in the previous section. Unlike the work of Zhang and Kim [3], the proposed algorithm maintains the trade-off between accuracy and efficiency, and can handle fracturing objects.

Fig. 1 shows the flow chart of the proposed image-space collision detection algorithm. 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 GPU as a *region of interest* (*ROI*). Fig. 2 illustrates three examples, each with a pair of objects, their AABBs, and the ROI. No ROI is found in Fig. 2-(a) whereas ROIs are found and passed to GPU in Fig. 2-(b) and -(c). Given an ROI, GPU computes PCSs. A PCS consists of two triangle sets, one from O_1 and the other from O_2. No PCS is computed in Fig. 2-(b) whereas two PCSs

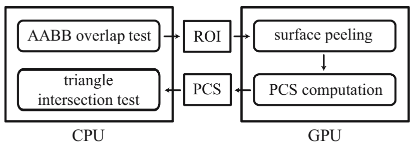

Fig. 1. System architecture

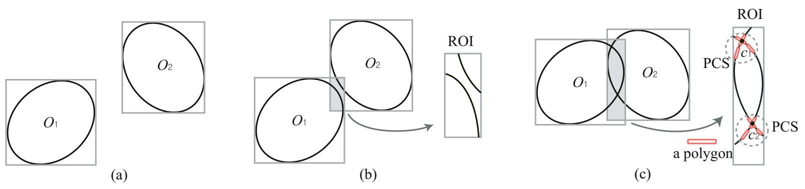

Fig. 2. Object AABBs and ROIs. (a) no ROI, (b) ROI with no collision, (c) ROI with collision.

are computed in Fig. 2-(c). Given the PCSs in Fig. 2-(c), CPU performs the traditional triangle intersection test to obtain the intersection points c_1 and c_2.

3 Surface Peeling and PCS Computation

Given an ROI passed by CPU, GPU computes PCSs by rendering the ROI surfaces into textures. Rendering is done in a layer-by-layer fashion, and we call it *surface peeling*. (The idea of surface peeling is not new, and its root comes from the area of transparent surface rendering [7]. It has been named *depth peeling*. There have been lots of applications of depth peeling, and in fact the work of Heidelberger *et al.* [5] described in Section 1 adopted the depth peeling algorithm. In this paper, we use the terminology 'surface peeling' instead of 'depth peeling' for stressing the differences between the two approaches, which will be discussed later.)

As in many image-space collision detection algorithms, orthographic projection along a viewing direction is used for rendering. It can be described as casting of parallel rays. Fig. 3 illustrates the surface peeling process with the example of Fig. 2-(c). Between the two objects, the one farther from the viewpoint is first rendered. It is O_2 in the example. The rendering result is stored in a 32-bit floating-point texture, `texture #1`, as illustrated in Fig. 3-(a). In the texture, the R channel stores the *depth* value of the rendered pixel, and the G channel stores the *triangle ID* of the pixel, which is the ID of O_2's triangle hit by the ray. (Triangle identification will be discussed in Section 5).

In the next phase, a new shader program renders O_1 through the *depth test* with the output of the previous stage (currently, stored in `texture #1`). The shader program discards a pixel of O_1 if it is not deeper than the corresponding texel in `texture #1`. The result is stored in `texture #2`, as shown in Fig. 3-(b), where the depth values and triangle IDs are stored into color channels R and G.

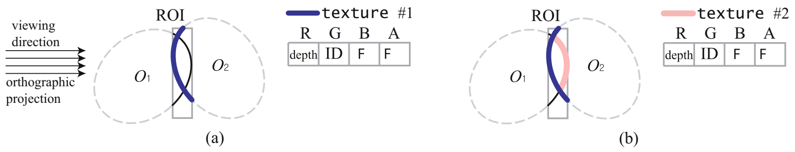

Fig. 3. Surface peeling. (a) phase 1, (b) phase 2.

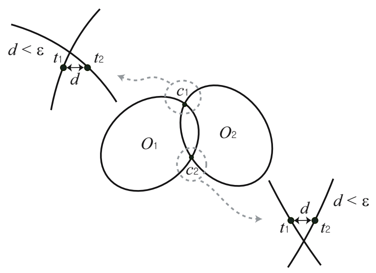

Fig. 4. PCS computation

In order to compute PCSs, a simple test is invoked for the space between `texture #1` and `texture #2` in Fig. 3. If the distance d between texel t_1 from `texture #1` and texel t_2 from `texture #2` is less than the threshold ϵ, as shown in Fig. 4, the triangle IDs are retrieved from t_1 and t_2, and then passed to CPU as PCSs. Finally, given the PCSs, CPU computes the intersection points, c_1 and c_2 in Fig. 4.

4 Alternate Surface Peeling

The ROI is composed of a pair of two objects, and in general PCS computation requires the two objects to be *alternately rendered*. Let us discuss the alternate rendering using the example in Fig. 5. O_2 is deeper than O_1, and therefore O_2 is rendered first to create `texture #1`, as shown in Fig. 5-(a). O_1 is then rendered to create `texture #2`, as shown in Fig. 5-(b). Surface peeling does not stop here. The shader program again renders O_2 through the depth test with `texture #2`, i.e. the shader program discards a pixel of O_2 if it is not deeper than the corresponding texel of `texture #2`. The result is stored in `texture #3`, shown in Fig 5-(c).

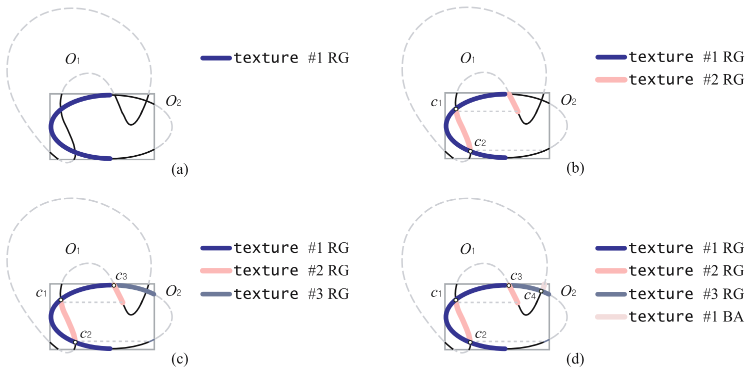

Fig. 5. Alternate surface peeling with three textures. (a) phase 1, (b) phase 2, (c) phase 3, (d) phase 4.

Suppose that the current implementation uses a graphics chip with four render target textures. (The state-of-the-art graphics hardware such as GeForce 8800 series supports eight render targets.) Among the four textures, three are used for surface peeling, and the remaining one is for recording PCSs. For efficient recording and readback of the PCSs, a hierarchical technique proposed by [8] is employed.

Note that only color channels R and G have been used so far to store the depth values and triangle IDs. Color channels B and A are empty. It is time to render O_1 through the depth test with `texture #3`. The rendering result is recorded at color channels B and A of `texture #1`, as shown in Fig. 5-(d). No more surface remains in the ROI, and the alternate rendering stops. (Occlusion query is used to decide when to stop rendering.) If we had more surfaces to render, B and A channels of `texture #2` and then `texture #3` would be used.

For a complex configuration of objects, however, more than six times of surface peeling may be needed. Then, the texture storing the PCSs is read back to CPU, and the second stage of the surface peeling is started, where the 6th surface peeled in the first stage is used for depth test.

Note that, as shown in Fig. 5-(d), not all surfaces in the ROI are rendered. In contrast, the Heidelberger′s algorithm based on depth peeling [5] renders *all* surfaces in the ROI, and all of the rendered surfaces are read back to CPU, which causes a serious overhead. Recall that our algorithm computes PCSs from the rendered surfaces, and passes the PCSs to CPU, not the rendered surfaces themselves. Given objects with complex geometry, the alternate surface peeling algorithm proposed in this paper is superior in performance to the Heidelberger′s algorithm.

PCSs are created between two adjacent textures. For example, in Fig. 5-(b), two PCSs are obtained between `texture #1` and `texture #2`, and eventually lead to the intersection points c_1 and c_2. Similarly, intersection point c_3 in Fig. 5-(c) is computed by CPU from the PCS obtained between `texture #2` and `texture #3`. Finally, c_4 in Fig. 5-(d) is computed from the PCS between `texture #3` and `texture #1`.

5 Triangle Identification for PCS Computation

A well-known problem of image-space collision detection is that its effectiveness is limited by the image-space resolution and the image-space approach often misses overlapping primitives. See Fig. 6 where two objects, O_1 and O_2, collide. O_1 includes the triangles (polygons) p_1, p_2 and p_3, and O_2 includes p_4 and p_5. Along the upper ray, t_1 is first recorded into the texture, but no pairing is possible because the intersection between the ray and p_1 lies in front of t_1. Therefore, no PCS is obtained. In contrast, the mid ray detects $<t_2,t_4>$ and identifies the PCS, $\{(p_2),(p_5)\}$. In actuality, p_2 and p_5 do not intersect, and CPU computes no intersection point. The lower ray detects $<t_3,t_5>$ and identifies the PCS, $\{(p_3),(p_5)\}$. Then, the CPU will compute the intersection point c_2. Note that the other intersection point c_1 is not found. Unless the image-space resolution reaches the infinity, overlapping primitives can be missed in the proposed algorithm.

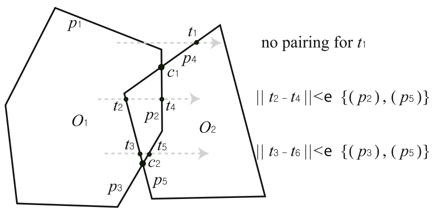

Fig. 6. Sampling error alleviation

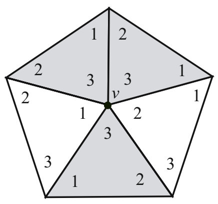

Fig. 7. Triangle ID: The center vertex v is assigned a gray color. In each triangle, vertex ordering is denoted by numbers. All of the three gray-colored triangles have v as the last vertex.

In order to alleviate (not solve) the inaccuracy problem, the proposed algorithm adopts a trick based on *flat shading*, where a triangle's color is set to the color of its last vertex. (It is OpenGL convention. In DirectX, the color of the first vertex determines the triangle color.) In the proposed scheme, a distinct color c is associated with each vertex v. Therefore, every flat-shaded triangle whose last vertex is v is colored in c, as illustrated in Fig. 7. For each pixel stored in the texture during the surface peeling process, its color is taken as the triangle ID. Note that the triangle ID is not unique, i.e. multiple triangles may have an ID. For example, the three gray-colored triangles in Fig. 7 will be given an identical ID.

Such non-unique IDs alleviate the problem caused by an insufficient image-space resolution. Let us revisit the mid ray in Fig. 6, and suppose p_4 and p_5 of O_2 have an identical ID. Then, the PCS will be $\{(p_2),(p_4,p_5)\}$, not just $\{(p_2),(p_5)\}$, and the intersection point c_1 can be obtained.

Fig. 8 illustrates the ratio, which is named *miss ratio*, of the missing to all intersection points. The miss ratios depend on viewing directions. In Fig. 8, a pair of θ (longitude) and ϕ (latitude) defined in a spherical coordinate system represents a viewing direction, where θ is sampled at intervals of 30^o and ϕ is at 10^o. Given the configuration of bunny and dragon in Fig. 8-(a), Fig. 8-(b) shows the miss ratios when the ROI is assigned a 256×256-sized texture. The algorithm adopts the non-unique ID scheme. The average miss ratio is 5.02% while the largest miss ratio is 12.7% and the smallest is 0.65%.

Fig. 8-(c) shows the miss ratios when the algorithm adopts the unique ID scheme, i.e. a triangle is given a unique ID. The miss ratios become high. The

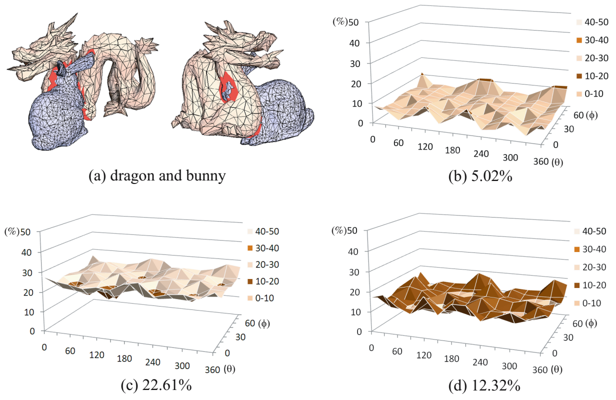

Fig. 8. Miss ratios: dragon (2.9K triangles) and bunny (2.7K triangles)

average miss ratio is 22.61% while the largest miss ratio is 29.31% and the smallest is 14.98%.

Fig. 8-(d) shows the result of using a 128×128-sized texture, where non-unique ID scheme is used. The average miss ratio is 12.32% while the largest miss ratio is 20.52% and the smallest is 5.86%. Compare the miss ratios of Fig. 8-(b) and Fig. 8-(d): 5.02% and 12.32%. Obviously, the miss ratio increases when a smaller texture is assigned to an ROI.

Recall that a smaller texture is assigned to an ROI when there are many ROIs in the scene. In such a scene of crowded colliding objects, each ROI generally takes just a small fraction in the screen. Therefore, inaccuracy in collision detection and consequent collision response would not be easily perceived. In contrast, suppose that the scene consists of the two objects in Fig. 8-(a). Then, the entire 512×512-sized texture is assigned to the ROI, and the miss ratio drops significantly, leading to realistic collision response. In summary, the precision of the collision detection is roughly proportional to the user's visual sensitivity. This feature makes the proposed algorithm distinguished from other image-space algorithms.

6 Implementation and Test

The proposed algorithm has been implemented in C++, OpenGL and Cg on a PC with 2.4 GHz Intel Core2 Duo CPU, 2GB memory, and NVIDIA GeForce 7900GTX GPU with 512 video memory and PCI-Express interface. Various functionalities of the graphics hardware are exploited, e.g. the GL_NV_occlusion_query for surface peeling, EXT_framebuffer_object for off-screen rendering, etc.

As discussed earlier, the proposed algorithm can handle a variety of dynamic objects. Fig. 9 shows a scene where a deforming object, lizard, walks through the balls on the billiard table. (See the attached video.) Table 1 shows the performance statistics for 4 configurations in Fig. 9. Each configuration has a distinct

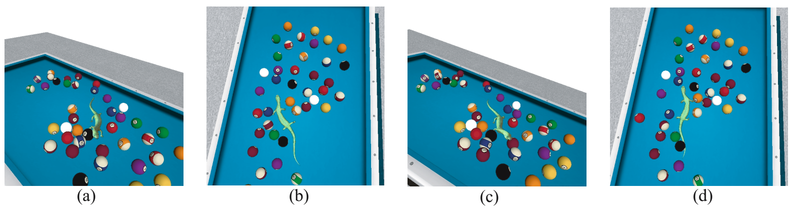

Fig. 9. Test #1: deforming lizard (2.1K triangles) and 40 billiard balls (each of 4.5K triangles)

Table 1. Performance evaluation for lizard and billiard table simulation (times in ms)

	nCTP	AABB	GPU	readback	CPU	total
(a)	15	0.07	3.11	0.11	0.07	3.36
(b)	33	0.07	3.08	0.13	0.14	3.42
(c)	96	0.07	3.28	0.15	0.44	3.94
(d)	173	0.07	3.02	0.36	0.68	4.13

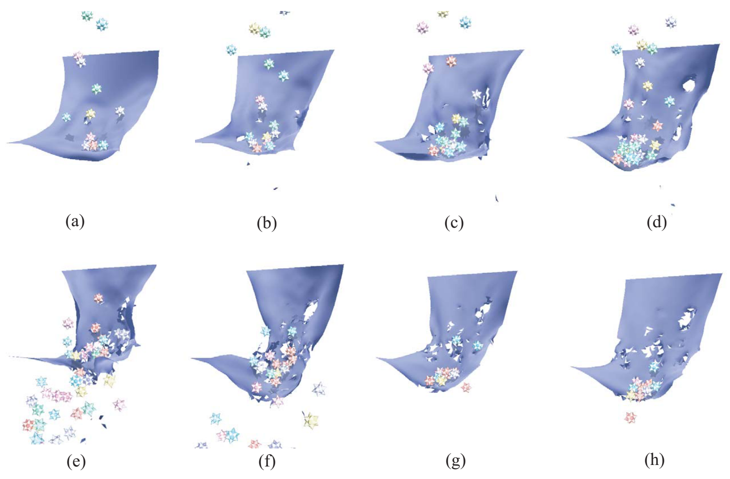

Fig. 10. Test #2: Fracturing cloth (1.6K triangles) and 40 falling objects (each of 2.1K triangles)

number of colliding triangle pairs (nCTP). AABB construction requires negligible amount of time. GPU time is spent for surface peeling and PCS computation. Note that the GPU times remain almost constant for varying nCTPs. In contrast, the readback time is proportional to the PCS size. So is the CPU time spent for triangle/triangle intersection test.

Fig. 10 shows simulation of a deforming and fracturing object, cloth, which is being torn by the falling sharp rigid objects. (See the attached video.) When the cloth is torn, its mesh connectivity changes. Furthermore, new triangles are

Table 2. Performance evaluation for cloth simulation (time in ms)

	nCTP	AABB	GPU	readback	CPU	total
(a)	16	0.03	2.66	0.12	0.10	2.91
(b)	115	0.03	3.05	0.68	0.66	4.42
(c)	96	0.03	4.26	0.39	0.48	5.16
(d)	39	0.03	2.39	0.14	0.36	2.93
(e)	125	0.03	3.22	0.73	0.67	4.65
(f)	222	0.03	2.38	0.84	0.95	4.20
(g)	339	0.03	3.15	0.92	1.66	5.75
(h)	262	0.03	2.90	0.71	1.17	4.81

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 a dynamic and fracturing object. In contrast, handling this kind of simulation would be difficult in the state-of-the-art collision detection algorithms such as those of Govindaraju *et al.* [6] (CULLIDE) and Zhang and Kim [3]: CULLIDE requires each sub-object to be a single triangle, which would lead to a huge number of occlusion queries, and the algorithm by Zhang and Kim requires an AABB to contain a single triangle, which would lead to a serious readback overhead.

Table 2 shows the performance statistics for 8 configurations in Fig. 10. Cloth simulation itself takes about 2 ms for all configurations. Collision detection among the falling rigid objects is done using RAPID 2.0 [9] which is better for rigid body collision detection.

7 Conclusion

This paper presented an efficient image-space algorithm for real-time collision detection. In the current implementation, shader programs compute the PCSs, and CPU performs the primitive-level intersection test. The algorithm can handle a variety of dynamic objects including fracturing meshes, and does rarely suffer from the readback problem. 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. CPU waits until GPU computes the PCSs. The proposed algorithm cannot handle self-collision. The proposed algorithms are being extended for overcoming these disadvantages.

Acknowledgement

This work was supported by grant No. R01-2006-000-11297-0(2007) from the Basic Research Program of the Korea Science & Engineering Foundation, and MIC, Korea under ITRC IITA-2006-(C1090-0603-0046).

References

1. van den Bergen, G.: Efficient collision detection of complex deformable models using aabb trees. Journal of Graphics Tools 2, 1–13 (1997)
2. Gottschalk, S., Lin, M.C., Manocha, D.: OBBTree: A hierarchical structure for rapid interference detection. In: Proc. of ACM SIGGRAPH 1996, pp. 171–180. ACM Press, New York (1996)
3. Zhang, X., Kim, Y.: Interactive collision detection for deformable models using streaming AABBs. IEEE Transaction on Visualization and Computer Graphics 13, 318–329 (2007)
4. Shinya, M., Forgue, M.: Interference detection through rasterization. Journal of Visualization and Computer Animation 2, 131–134 (1991)
5. Heidelberger, B., Teschner, M., Gross, M.: Detection of collisions and self-collisions using image-space techniques. In: Proc. of Winter School of Computer Graphics 2004, pp. 145–152 (2004)
6. Govindaraju, N.K., Redon, S., Lin, M.C., Manocha, D.: CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In: Proc. of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003, pp. 25–32. ACM Press, New York (2003)
7. Everitt, C.: Interactive order-independent transparency. NVIDIA corporation technical report (2001), http://www.cs.unc.edu/~{}geom/obb/obbt.html
8. Choi, Y., Kim, Y., Kim, M.: Self-CD: Interactive self-collision detection for deformable body simulation using GPUs. In: Proc. of Asian Simulation Conference, pp. 187–196 (2005)
9. Gottschalk, S.: RAPID: Robust and accurate polygon interference detection system (1996), http://www.cs.unc.edu/~{}geom/obb/obbt.html