GPU-based Image-space Approach to Collision Detection among Closed Objects

Han-Young Jang
jhymail@gmail.com

TaekSang Jeong
nanocreation@gmail.com

JungHyun Han
jhan@korea.ac.kr

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

Abstract

This paper presents an image-space algorithm for real-time collision detection, which is run completely by GPU. For multiple objects with no collision, the front and back faces appear alternately along the view direction. However, such alternation is violated when objects collide. Based on these observations, the algorithm has been devised, and the implementation utilizes the state-of-the-art functionalities of GPU such as framebuffer objects and occlusion query. The experimental results show the feasibility of GPU-based collision detection and its performance gain in real-time applications such as 3D games.

1. Introduction

Collision detection is a fundamental problem in many applications such as computer graphics and animation, 3D games, virtual reality, physically-based simulation, and robotics. It is often the major computational bottleneck in real-time simulation of complex and dynamic systems. A lot of algorithms for collision detection have been proposed, and 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*.

This paper proposes a new technique for image-space approach. The proposed algorithm is quite simple, is easy to implement using shaders, and shows superior performance. The simplicity and efficiency of the algorithm are attractive for real-time applications such as 3D games.

The structure of this paper is as follows. Section 2 reviews the related work, and discusses the advantages and disadvantages of the traditional approach. Section 3 describes the features of collision among closed objects. Based on the features, Section 4 presents the collision detection algorithm, which overcomes the disadvantages of the traditional approach. Section 5 extends the algorithm presented in Section 4 for handling complex scenes. Section 6 discusses the strength of the proposed algorithm. Section 7 shows the test results, and Section 8 concludes the paper.

2. Related Work

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 such as bounding spheres [11, 16], axis-aligned bounding boxes [20, 22], oriented bounding boxes [3], discrete orientation polytopes [12], and quantized orientation slabs with primary orientations [7]. 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 [19, 14].

In contrast with the object-space approach, the image-space approach typically measures the volumetric ranges of objects along the viewing direction, and then compares the ranges to detect collision. The trend started with the work of Shinya and Forgue [18], where the depth layers of convex objects are rendered into depth buffers, and then compared for interference checking. Since then, various algorithms for image-space approach have been proposed, and have attempted to maximally utilize the graphics hardware's functionality [1, 15, 10, 21].

The most recent efforts in the image-space approach include the work by Heidelberger *et al.* [8, 9], which is useful

to discuss both of the advantages and disadvantages of the image-space approach. In their work, layered depth images (LDIs) are computed, 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. This approach can handle concave objects, and shows real-time performance for simple objects. However, LDI generation requires a considerable amount of time for objects with complex geometry.

In general, the advantages of the image-space approach can be listed as follows. Unlike the object-space approach which requires non-trivial pre-processing for computing bounding volumes and their hierarchy, the image-space approach does rarely require pre-processing. Partly due to absence of the pre-processing, the image-space approach is easy to implement. It can also effectively handle deformable objects and dynamic environments. Moreover, it usually employs graphics hardware or GPU which has been evolving at a rate faster than Moore's law, while the object-space approach performs virtually all collision tests on CPU.

The image-space approach also reveals disadvantages. First of all, virtually all of the image-space algorithms proposed so far perform collision tests using both CPU and GPU, and suffer from the limited bandwidth between them, i.e. the *readback* problem. In the LDI-based algorithm by Heidelberger *et al.* [8, 9], for example, the CPU reads the LDIs from the GPU's back buffers, and then tests the LDIs for Boolean intersection. Due to the limited bandwidth, the sampling resolution (LDI resolution) is usually made low, 32x32 through 128x128. Note that, however, the accuracy of the collision detection is governed by the LDI precision, and low resolutions do not guarantee accurate detection of collision for complex objects. As an effort to overcome the readback problem, Govindaraju *et al.* [5] proposed an algorithm named CULLIDE, which computes a potentially colliding set (PCS) through hardware visibility query. However, it requires off-line pre-processing or setup stage, which is quite complex. Recently, an efficient pre-processing technique for the CULLIDE algorithm, named chromatic decomposition [4], 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 deformable mesh the topology of which may vary frame by frame, the time-consuming pre-processing has to be executed per each frame.

Another major disadvantage of the image-based approach lies in the rendering cost because it generally renders the entire surface areas of the objects to be tested for collision. The computational bottleneck may lie in GPU for arbitrarily-shaped complex objects.

Finally, the input to most of the image-space algorithms is limited to a pair of objects, not many objects in a large scene. Knott and Pai [13] proposed an algorithm that can handle large number of objects. However, their algorithm is based on wireframe rendering, and therefore is inherently fragile, i.e. may often miss obvious collision.

In order to resolve the problems of the traditional image-space approach, this paper proposes a GPU-based algorithm, where the entire collision test is run by GPU and readback is minimized. Assuming no self-collision, the rendering cost has been dramatically reduced. Further, it detects collision of the entire scene, not of only a pair of objects.

3. Features of Collision among Closed Objects

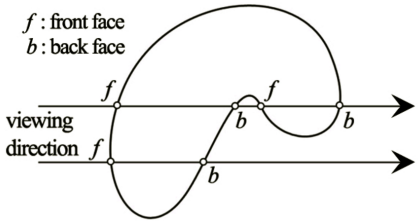


Figure 1: Front-back face pairing in a closed object

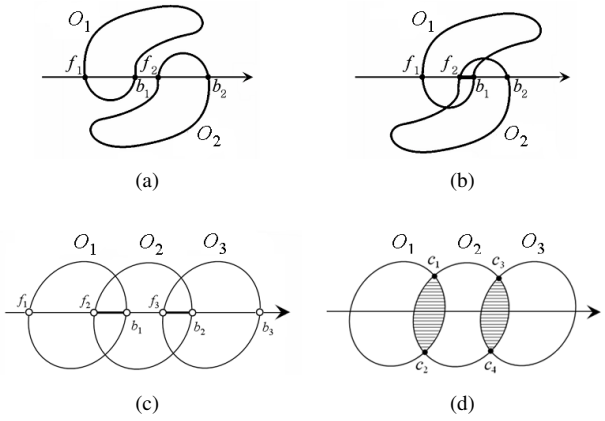


Figure 2: Front-back face pairing in multiple objects: (a) consistent pairs (b) inconsistent pair (c) inconsistent pairs (d) collision area

The image-space collision detection algorithm proposed in this paper handles only closed objects. Collision between closed objects reveals distinct features, which have been observed in many image-based algorithms. The front and back faces of a closed mesh appear alternately along a viewing ray, as illustrated in Fig. 1. The observation can be generalized for a scene of multiple objects with no collision, as illustrated in Fig. 2-(a). More precisely, a front face of an

object is paired with a back face of the same object. In Fig. 2-(a), we can find two such pairs, (f_1,b_1) of O_1 and (f_2,b_2) of O_2. We call them *consistent pairs*.

Collision takes place when an object penetrates or touches another object. In Fig. 2-(b), O_1 penetrates O_2, and then consistent pairing is violated, i.e. right in front of b_1 lies f_2, not f_1. We call (f_2,b_1) an *inconsistent pair*. Such observation holds for multiple object collision, as illustrated in Fig. 2-(c), where we can find two inconsistent pairs, (f_2,b_1) and (f_3,b_2). Then, collision detection resorts to the task of finding inconsistent pairs (f_i,b_j), $i \neq j$, which tells us that O_i collides with O_j. The shaded areas in Fig. 2-(d) encompass all inconsistent pairs. They are used to compute the *collision points* (the actual intersections between objects) denoted by c_1 through c_4 in the figure.

4. Image-space Collision Detection and Determination

Collision handling is often divided into three parts: *collision detection* which judges whether tow or more objects collide, *collision determination* which computes the collision points, and *collision response* which determines the actions to be taken in response to the collision. Many people call the combination of the first two parts simply 'collision detection,' but this section distinguishes between them. Specifically, the collision detection module computes the inconsistent pairs, which are taken as input for the collision determination module. In the proposed system, both of collision detection and collision determination are done by GPU while collision response is by CPU.

Each object in the scene is associated with an axis-aligned bounding box (AABB). The potentially colliding set (PCS) is computed using the AABBs. The AABBs of the PCS determine the dimension of the *virtual orthographic view volume*, with which the image-space collision test is invoked. The proposed algorithm can be described as *virtual ray casting* in the sense that collision is detected for a set of parallel rays within the orthographic view volume.

4.1. Front-face Rendering through Depth Peeling

In the proposed algorithm, a subset of the front faces in the PCS is rendered layer by layer into textures. Such rendering is named *depth peeling*, and Fig. 3 illustrates an example with two objects (O_1 and O_2) and three 32-bit floating-point textures (`texture #1`, `texture #2`, and `texture #3`). At the initialization stage, all textures are filled with the depth value of the scene background. Then, the scene is rendered with the depth test enabled, and its depths are recorded into `texture #1`. As a result, `texture #1` contains the depth values of the first-layer

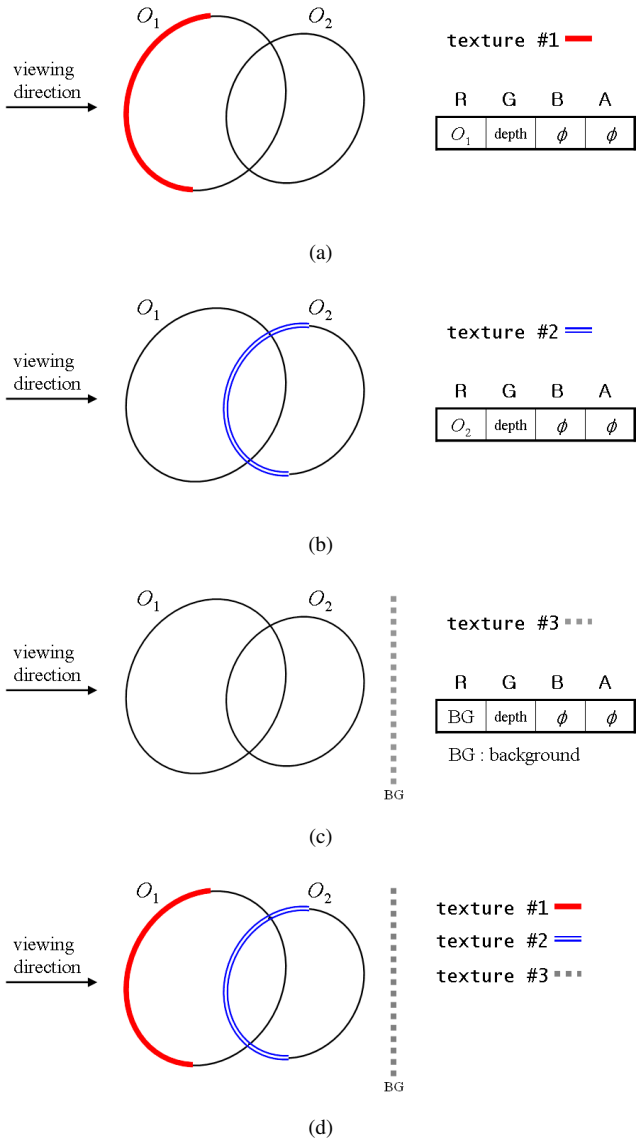


Figure 3: Depth peeling iteration: (a) 1st-layer front faces (b) 2nd-layer front faces (c) background (d) front faces in textures

front faces, as illustrated in Fig. 3-(a). Rendering is done by a shader, and each texel of the texture contains one more piece of information: the object ID of the rendered pixel. As shown in Fig. 3-(a), the object ID and the depth are stored in color channels R and G, respectively.

A new shader is used for filling `texture #2` and `texture #3`. The shader renders the scene using the result of the previous stage (currently `texture #1`), and discards a pixel if it is not deeper than the texel in the texture or its object ID is identical to that of the texel. Then, only the second-layer front faces survive. Their depth values are stored into `texture #2`, as shown in Fig. 3-(b).

Finally, the scene is rendered again in order to update `texture #3`. However, there is no more front face. Nothing is rendered. As shown in Fig. 3-(c), `texture #3` is not updated at all, and all of its texels contain the background depth. Fig. 3-(d) shows the result of the depth peeling process.

The depth peeling process is implemented using *frame-buffer object* (FBO) [6], which has recently been specified as a collection of local buffers such as color, depth, stencil, and accumulation buffers. In this new specification, rendering destinations can be off-screen renderbuffers or textures. They can be shared among FBOs. Therefore, the texture rendered in a stage will be available for the next stage, at the minimum cost of context switching.

4.2. Collision Detection: Computation of Inconsistent Pairs

Either the front faces or the back faces of a scene can be rendered by changing the *culling mode*. When the front-face depth peeling is done and three textures (`texture #1`, `texture #2`, and `texture #3`) are obtained, the culling mode is changed to process back faces.

A back-face pixel p can be located either between `texture #1` and `texture #2`, as shown in Fig. 4-(a), or between `texture #2` and `texture #3`, as shown in Fig. 4-(b). In both cases, the front-face texel t along p's viewing ray is retrieved from the texture in front of p: from `texture #1` in Fig. 4-(a), and from `texture #2` in Fig. 4-(b). Then, the IDs of t and p are compared. If they are identical, (t,p) is a consistent pair, as shown in Fig. 4-(a). Otherwise, it is an inconsistent pair, as shown in Fig. 4-(b), which indicates that O_1 collides with O_2.

Every inconsistent pair (t,p) is recorded. As the front face information required for an inconsistent pair has already been recorded in three textures, only the back face information (ID and depth) needs to be stored. When p is located between `texture #1` and `texture #2`, `texture #3` is used to store the back face information. In contrast, `texture #1` is used as output when p is located between `texture #2` and `texture #3`, as illustrated in Fig. 4-(b). Note that p's ID and depth are stored in color channels B and A because color channels R and G have been filled with front face information. The shaded area in Fig. 4-(c) represents all inconsistent pairs.

Fig. 5 illustrates the inconsistent pair computing process for a different configuration of two objects. Note that, unlike the case of Fig. 4, the inconsistent pairs take only a small subset of the viewing rays. In fact, Fig. 4 shows a special case. In general, inconsistent pairs are found in a small area, and the collision determination algorithm discussed in Section 4.3 computes the *collision points* within the area.

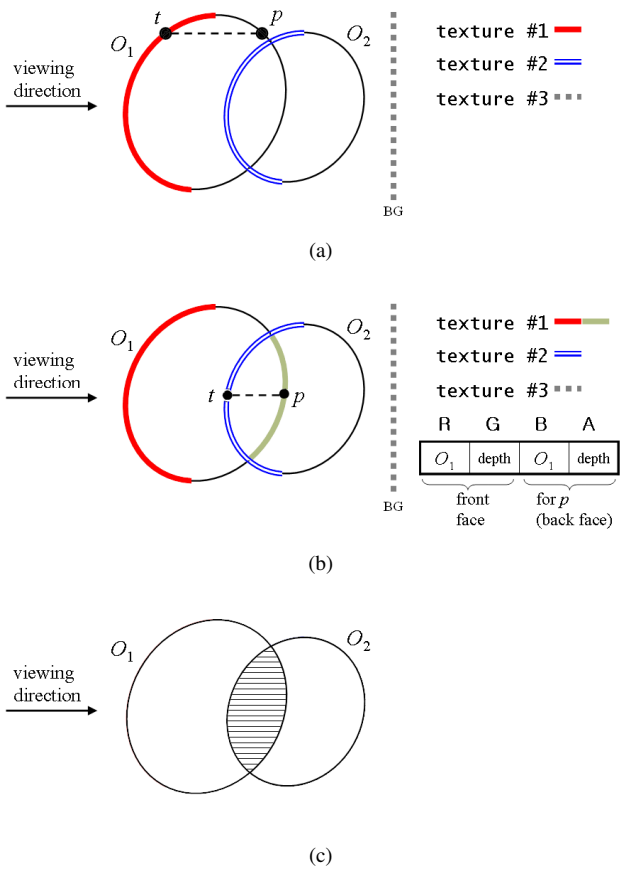


Figure 4: Collision detection: (a) consistent pair (b) inconsistent pair (c) collision area

4.3. Collision Determination

Note that `texture #1`, `texture #2` and `texture #3` contain front and back face information required for all the inconsistent pairs. More precisely, color channels R and G of the three textures store the front face information while color channels B and A of `texture #1` and `texture #3` store the back face information. Using the textures, the collision determination stage computes the *collision points*.

For computing the collision points, the culling mode is disabled and all faces in the scene are rendered. For each pixel p, first of all, the front-face information stored in the textures is retrieved. Suppose a texel t is selected from a front face. The pixel shader performs three tests with p and t: (1) if p and t have different IDs, (2) if t is not deeper than p, and (3) the distance between p and t is less than a given threshold. Fig. 6-(a) shows examples: (t_1,p_1) does not pass the first test; (t_2,p_2) passes the first test, but fails the second test; (t_3,p_3) passes both of the first and second tests, but fails the third test; (t_4,p_4) passes all the tests, and a collision point c=(t_4,p_4) is obtained.

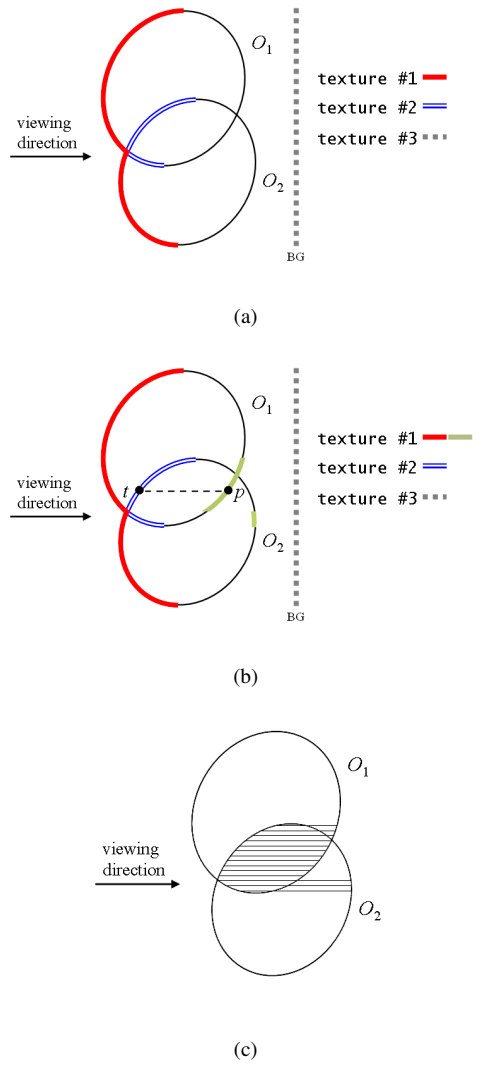


Figure 5: Collision detection: (a) depth peeling result (b) inconsistent pair (c) collision area

In Fig. 6-(a), p_4 is a back-face pixel, and t_4 is a front-face pixel. Therefore, c is a point of collision 'between back and front faces.' However, p is not necessarily a back-face pixel. In Fig. 6-(b), p is a front-face pixel, and the three tests lead to a point of collision 'between front faces.'

In the collision determination stage, each pixel p has to be tested not only with the front faces but also with the back faces. The tests are quite similar. For a texel t from a back face, p is tested (1) if its ID is different from t's, (2) if it is not deeper than t, and (3) the distance from t is less than a given threshold. Fig. 6-(c) shows a collision point c=(t,p) which passes the three tests. It is found that, for the object configuration of Fig. 6-(a) and -(c), the collision determination algorithm computes two collision points.

Finally, the collision determination algorithm can compute a point of collision 'between back faces,' as shown in Fig. 6-(d). Two collision points are computed for the configuration of Fig. 6-(b) and -(d).

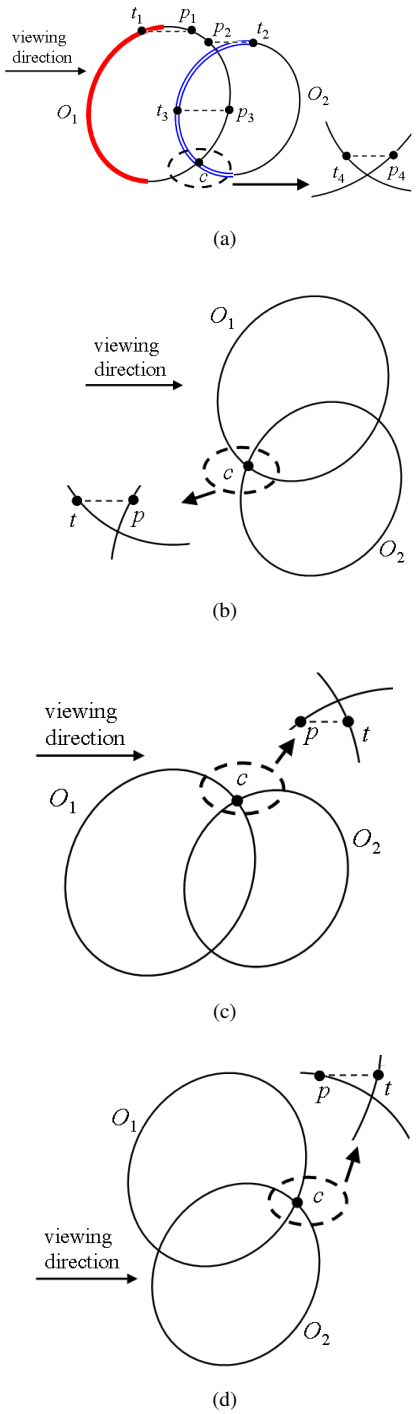


Figure 6: Collision determination: (a) back-front collision (b) front-front collision (c) front-back collision (d) back-back collision

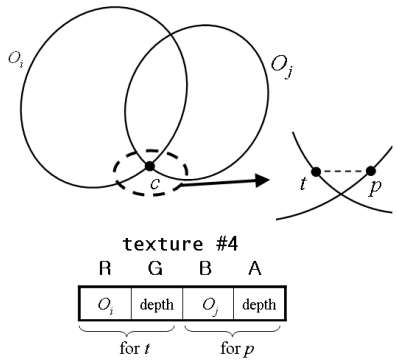


Figure 7: Collision point representation

A collision point c represents a pair of points from two colliding objects O_i and O_j, and therefore is recorded as two points in `texture #4`, as shown in Fig. 7. The set of collision points is readback to CPU for collision response. (In the collision point representation, each ID includes a flag indicating whether the point belongs to a front face or a back face. The flag is useful for the collision response stage.)

5. Iteration for a Complex PCS

Contemporary graphics hardware supports up to four *render targets*, and the proposed algorithm utilizes all of them: `texture #1`, `texture #2` and `texture #3` to store the front and back face information for the inconsistent pairs, and `texture #4` to store the collision points. For a simple PCS, four textures would be enough. For a complex PCS such as the one in Fig. 8, however, more textures will be needed. The problem is resolved by iterating the collision detection and determination algorithms.

Fig. 8-(a) shows the result of depth peeling with three textures. The collision determination algorithm considers only the pixels located between `texture #1` and `texture #3`, and will return two collision points, c_1 and c_2, as shown in Fig. 8-(b). The collision points are readback to CPU. Then, the second iteration of depth peeling starts by taking `texture #3` of the first iteration as `texture #1`. Fig. 8-(c) shows the result of depth peeling. Finally, the collision determination algorithm computes four collision points, c_3 through c_6, as shown in Fig. 8-(d).

To decide if a new iteration is needed, the system checks the result of the *occlusion query* [17], which has been issued for the front faces in the previous iteration. The occlusion query returns the number of pixels that have passed the depth test. If the return value is 0, no more iteration is needed.

A more complex PCS is shown in Fig. 9. The first iteration computes the collision points, c_1, c_2 and c_3 in Fig. 9-(a), and the second iteration computes the collision point c_4 in Fig. 9-(b). All the required collision points are extracted.

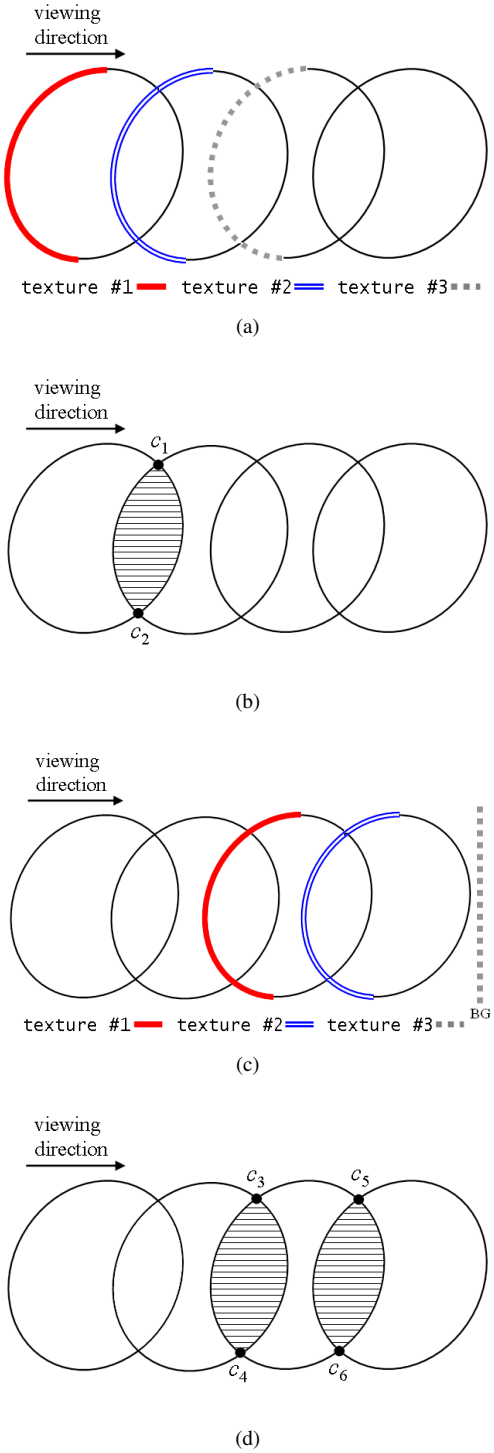


Figure 8: Iterated collision test: (a) 1st iteration: depth peeling (b) 1st iteration: collision points (c) 2nd iteration: depth peeling (d) 2nd iteration: collision points

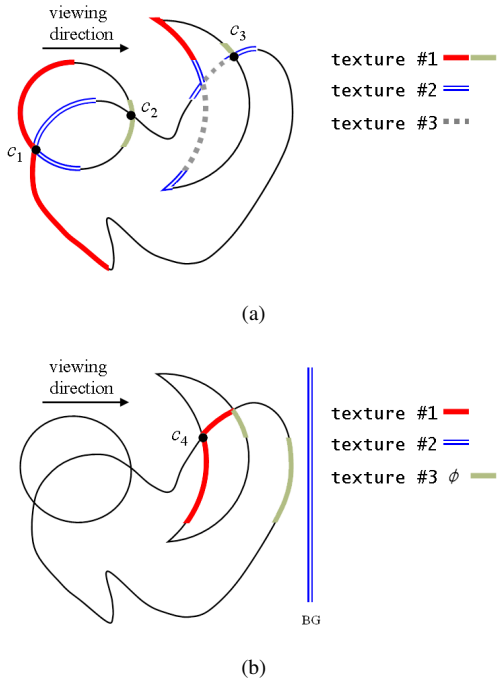


Figure 9: Iterated collision test: (a) 1st iteration (b) 2nd iteration

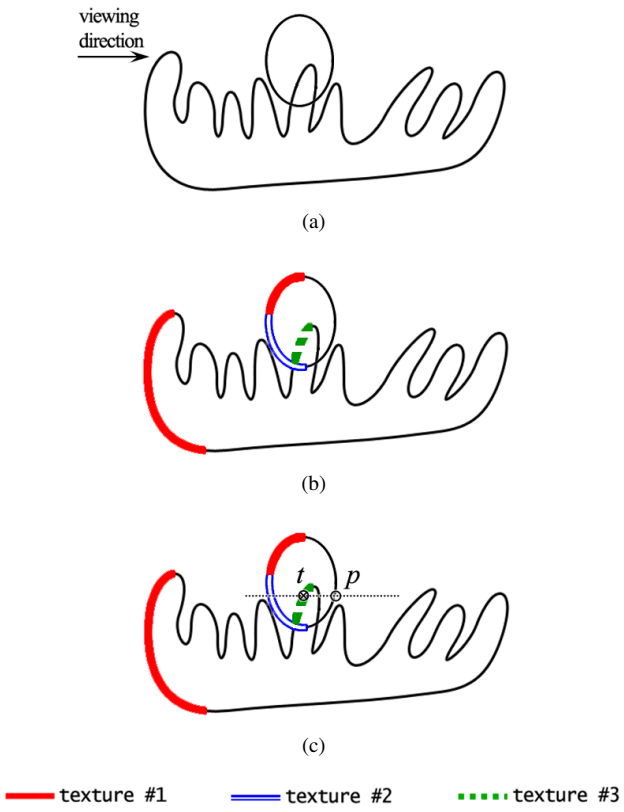


Figure 10: Collision example: (a) objects (b) depth peeling results (c) inconsistent pair

6. Discussions

As discussed in Section 2, the traditional image space approach suffers from three major drawbacks. The algorithm proposed in this paper resolves them successfully. First of all, in the current implementation, CPU computes only the potentially colliding set (PCS), and the entire collision test is run by GPU. The authors believe that such an approach is the first of its kind in collision test research field. As discussed in Section 5, CPU reads the collision points only once for an iteration. The algorithm does not suffer from the readback problem.

The second drawback of the traditional approach has also been resolved: the proposed image-space algorithm does not necessarily render the entire surface of an object. For an arbitrarily-shaped complex object, it significantly increases efficiency. Fig. 10 and Fig. 11 illustrate the collision test with two objects in different configurations. In both cases, two iterations are enough, and just three layers of the front faces are rendered in total. In contrast, for example, the LDI approach computes nine pairs of entry and leaving points (volumetric ranges) by rendering the entire surface, for the cases of Fig. 10 and Fig. 11.

Third, the proposed algorithm can detect collision of the entire scene, not of only a pair of objects. The performance gain obtained by processing the entire scene at a time is especially useful for real-time applications such as 3D games.

A noteworthy fact is that the texture resolution governing the accuracy of collision detection can be adaptively set depending on the PCS size in the screen. Then, the precision of the collision test becomes compatible with the user's visual perception. This makes the proposed algorithm distinguished from other real-time image-space collision detection algorithms.

For collision response, the impulse-based simulation method proposed by Guendelman *et al.* [2] is used. One of the factors determining the impulse is the surface normal at the collision point. CPU computes the surface normal using the collision points provided by the collision determination module.

Collision between two objects is classified into a penetration case and a contact case. For the penetration case, multiple collision points are extracted. In contrast, a single collision point can be extracted for the contact case where an object simply touches the other. Our collision response module considers only the penetration case, and computes the surface normal using three neighboring non-linear collision points of the same object ID. The contact case is naturally handled because the simulator of [2] anticipates the object configuration of the next frame, computes the im-

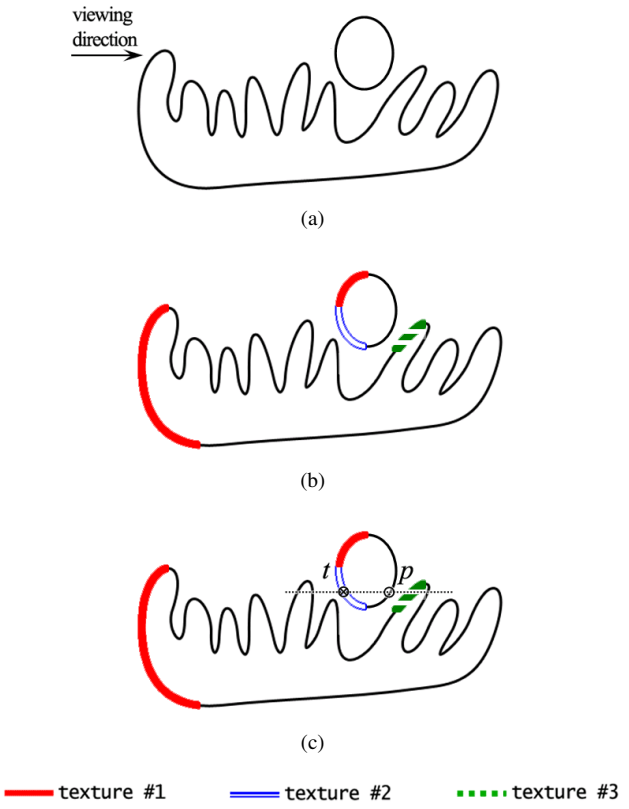


Figure 11: Non-collision example: (a) objects (b) depth peeling results (c) consistent pair

pulse based on the collision points, and then updates the current frame.

7. Implementation

The proposed algorithm has been implemented in C++, OpenGL and Cg on a PC with 3.2 GHz Intel Pentium4 CPU, 2GB memory, and NVIDIA GeForce 7800GTX 256MB GPU. Various functionalities of the graphics hardware are exploited, e.g. the GL_NV_occlusion_query for depth peeling, GL_ARB_vertex_buffer_object for vertex buffer, EXT_framebuffer_object for off-screen rendering, etc. 32-bit floating-point textures are used as render targets. FACE semantic of OpenGL NV_fragment_program 2.0 profile is used for checking whether a pixel is on a front face or a back face.

The proposed algorithm has been tested with various objects, and shown in Fig. 12 are complex models among them. Fig. 13 lists the measured execution time for both collision detection and collision determination between pairs of the complex models. In each sub-figure, the right one shows heavy intersection while the left one shows light intersection. When heavily intersected, more

(a) Dragon (902K triangles) (b) Goldman (512K triangles)

(c) Buddha (106K triangles) (d) Dinosaur (108K triangles)

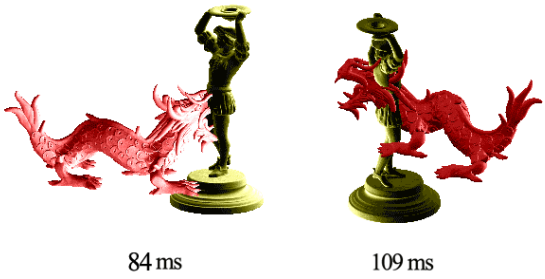
Figure 12: Test objects

time is need for collision detection. Fig. 13-(a) through -(d) are listed in the deceasing order of mesh complexity, e.g. 1423K triangles in Fig. 13-(a) and 214K triangles in Fig. 13-(d).

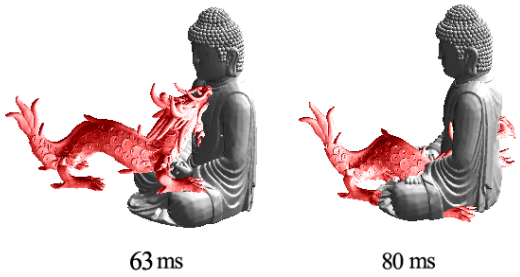
In Fig. 14, the upper-right vertex of each curve corresponds to Fig. 13-(a) while the lower-left vertex corresponds to Fig. 13-(d), i.e. execution time is proportional to the mesh complexity. Collision detection and determination performances are also evaluated by changing the texture resolution. Three curves in Fig. 14 show that the texture resolution does not have a great impact on the performance. Recall that the accuracy of the collision detection is governed by the texture/image precision. Therefore, in the proposed approach, highly accurate collision test can be achieved in real-time even with 1024×1024 texture.

Table 1: Performance evaluation for collision among multiple objects

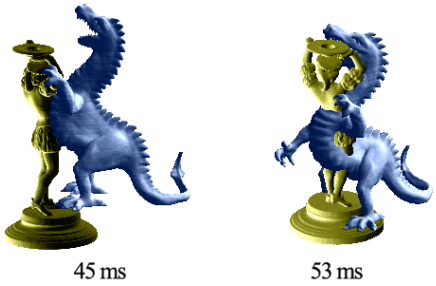
# objects	collision test time per frame	collision response time per frame	fps
100	20ms	2ms	40
50	9ms	1ms	84



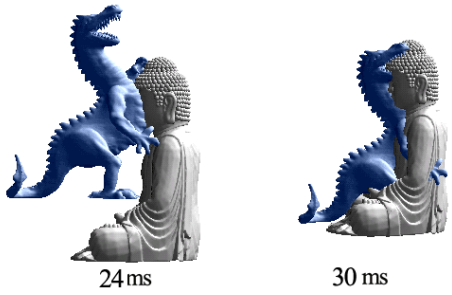
(a) Dragon and Goldman



(b) Dragon and Buddha



(c) Dinosaur and Goldman



(d) Dinosaur and Buddha

Figure 13: Performance evaluation for collision test between two objects

Fig. 15 shows a cube in which multiple objects are moving and colliding with each other. Each model consists of thousands of triangles, and an object has 3K polygons on average. Table 1 shows the performance evaluation measured by varying the number of objects. Collision test and

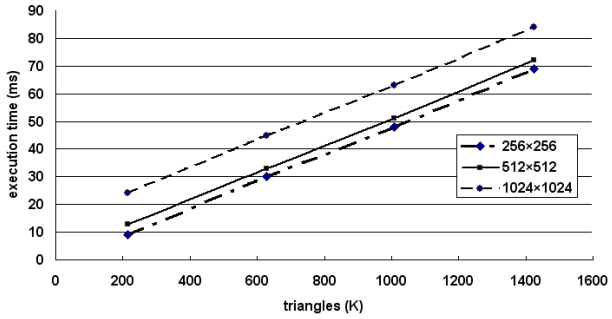


Figure 14: Performance evaluation with different texture resolutions

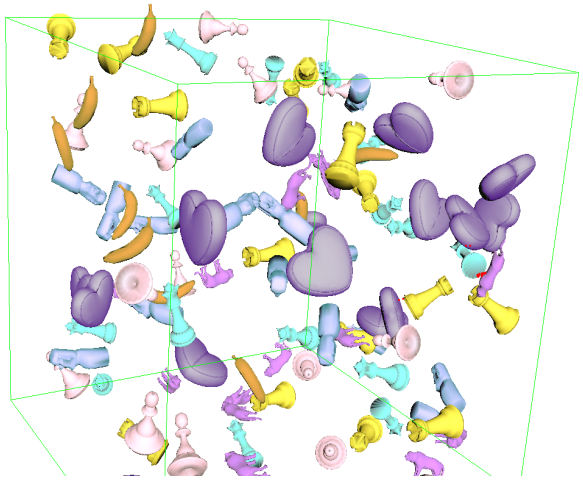


Figure 15: Multiple objects in a cube

response show satisfactory performances despite the high complexity of the scene in Fig. 15. Finally, the proposed algorithms are tested with ragdoll simulation and its collision with rigid body objects, as shown in Fig. 16. The collision test also shows quite satisfactory performance.

8. Conclusion

This paper presented an efficient image-space algorithm to real-time collision detection. In the current implementation, CPU computes only the potentially colliding set (PCS), and the entire collision test is run by GPU. The algorithm does rarely suffer from the readback problem. Further, the proposed algorithm does not necessarily render the entire surface of an object, and therefore the rendering cost has been reduced. The algorithm's efficiency is achieved by detecting collision of the entire scene, not of only a pair of objects. The algorithm maximally utilizes the state-of-the-art functionalities of GPU such as framebuffer objects and occlusion query. The experimental results show the feasi-

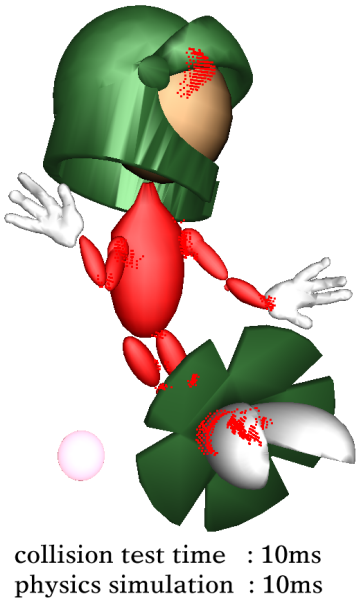


Figure 16: Ragdoll simulation

bility of GPU-based collision detection and its performance gain in real-time applications such as 3D games.

9. Acknowledgement

This research was supported by the Ministry of Information and Communication, Korea under the Information Technology Research Center support program supervised by the Institute of Information Technology Assessment, IITA-2005(C1090-0501-0019).

References

[1] G. Baciu, S. K. Wong, and H. Sun. RECODE: An image-based collision detection algorithm. *Journal of Visualization and Computer Animation*, 10(4):181–192, 1999.

[2] R. F. E. Guendelman, R. Bridson. Nonconvex rigid bodies with stacking. In *Proceedings of ACM SIGGRAPH*, pages 871–878, 2003.

[3] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. In *Proc. of ACM SIGGRAPH*, pages 171–180, 1996.

[4] N. K. Govindaraju, D. Knott, N. Jain, I. Kabul, R. Tamstorf, R. Gayle, M. C. Lin, and D. Manocha. Interactive collision detection between deformablemodels using chromatic decomposition. In *Proc. of ACM SIGGRAPH 2005*, pages 991–999, 2005.

[5] N. K. Govindaraju, S. Redon, M. Lin, and D. Manocha. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In *Proc. of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 25–32, 2003.

[6] S. Green. The opengl framebuffer object extension. Game Developers Conference 2005, 2005.

[7] T. He. Fast collision detection using QuOSPO trees. In *Symposium on Interactive 3D Graphics*, pages 55–62, 1999.

[8] B. Heidelberger, M. Teschner, and M. Gross. Realtime volumetric intersections of deforming objects. In *Proc. of Vision, Modeling and Visualization*, pages 461–468, 2003.

[9] B. Heidelberger, M. Teschner, and M. Gross. Detection of collisions and self-collisions using image-space techniques. In *Proc. Computer Graphics, Visualization and Computer Vision WSCG'04*, pages 145–152, 2004.

[10] K. Hoff, A. Zaferakis, M. Lin, and D. Manocha. Fast 3D geometric proximity queries between rigid and deformable models using graphics hardware acceleration. Technical report, Department of Computer Science, University of North Carolina, 2002.

[11] P. M. Hubbard. Interactive collision detection. In *Proc. of IEEE Symposium on Research Frontiers in Virtual Reality*, pages 24–32, 1993.

[12] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[13] D. Knott and D. Pai. CinDeR: Collision and interference detection in real–time using graphics hardware. In *Proc. of Graphics Interface '03*, pages 73–80, 2003.

[14] J. Mezger, S. Kimmerle, and O. Etzmuss. Hierachical techniques in collision detection for cloth animation. *Journal of WSCG*, 11(2):322–329, 2003.

[15] K. Myszkowski, O. Okunev, and T. Kunii. Fast collision detection between computer solids using rasterizing graphics hardware. *Visual Computer*, 11:497–511, 1995.

[16] I. Palmer and R. Grimsdale. Collision detection for animation using sphere-trees. *Computer Graphics Forum*, 14(2):105–116, 1995.

[17] A. Rege. Occlusion - hp and nv extensions. Game Developers Conference 2002, 2002.

[18] M. Shinya and M. Forgue. Interference detection through rasterization. *Journal of Visualization and Computer Animation*, 2(4):131–134, 1991.

[19] M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets, and M. Gross. Optimized spatial hashing for collision detection of deformable objects. In *Proc. of Vision, Modeling, Visualization*, pages 47–54, 2003.

[20] G. van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools*, 2(4):1–14, 1997.

[21] T. Vassilev, B. Spanlang, and Y. Chrysanthou. Fast cloth animation on walking avatars. *Computer Graphics Forum (Proc. of Eurographics01)*, 20(3):260–267, 2001.

[22] G. Zachmann and W. Felger. The boxtree: enabling real-time and exact collision detection of arbitrary polyhedra. In *Proc. of Workshop on Simulation and Interaction in Virtual Environments, SIVE 95*, pages 104–113, 1995.