Collision Detection Using Axis Aligned Bounding Boxes

Panpan Cai, Chandrasekaran Indhumathi, Yiyu Cai, Jianmin Zheng, Yi Gong, Teng Sam Lim and Peng Wong

Abstract Collision detection plays a critical role in real-time applications such as game, simulation, and virtual reality. Collision avoidance is important in robotics path planning. Industrial safety, especially in construction and building, has a close linkage with the concept of contact avoidance. This chapter is interested in the investigation of collision detection problem using hardware graphics acceleration. Axis aligned bounding boxes (AABB) technique will be applied also for fast collision detection.

Keywords Axis aligned bounding boxes · Collision detection · Simulation

1 Introduction

Collision detection plays a critical role in real-time applications such as game, simulation, and virtual reality. Collision avoidance is important in robotics path planning. Industrial safety, especially in construction and building, is closely related to contact avoidance. High-accuracy and real-time collision detection for physically based simulations is often computationally expensive (Fauer et al. 2008). Apart from making the simulation more intelligent and realistic, the collision detection engine is also the basis of many applications such as collision response and haptic feedback.

Usually, collision detection algorithms are divided into two phases: the *broad phase* and the *narrow phase* (Govindaraju et al. 2003). The broad phase aims at

P. Cai · C. Indhumathi · Y. Cai (✉) · J. Zheng
Nanyang Technological University, Singapore, Singapore
e-mail: myycai@ntu.edu.sg

Y. Gong · T. S. Lim · P. Wong
PEC Limited, Jurong, Singapore

culling possible colliding pairs or groups. An acceleration data structure called bounding volume hierarchies (BVH) is frequently used to perform the culling (Gottschalk et al. 1996; Tang et al. 2010, 2011). Spatial hierarchical structures such as Octrees (Hunter 1978; Jung and Gupta 1996; Zhou et al. 2010) are also one of those famous acceleration tools. The narrow phase often contains primary interference checks (Govindaraju et al. 2003; Tang et al. 2010, 2011; Ericson 2005) using direct primary tests, separating axes tests, separating plane tests, or proximity tests. As the set of candidate triangles can be very large, traditional narrow phase algorithms are often computationally intensive and time-consuming.

To speed up the algorithms, attempts in developing parallelized collision detection algorithms have been made using multi-core CPUs (Tang et al. 2010) and distributed computers. But none of these platforms could have a stronger parallelization power as modern GPUs do. Researchers such as Fauer and Govindaraju have investigated the potential of relying on GPU platforms to speed up collision detection and have achieved impressive results (Fauer et al. 2008; Govindaraju et al. 2003; Heidelberger et al. 2003). GPUs have a nature parallel-styled design with tremendous computational horsepower and high memory bandwidth to support graphic applications such as shading and rendering (NVIDIA 2010). In order to meet the requirement of more complex general processing and computing, GPUs evolves into General Purpose GPUs (GPGPUs). GPGPUs provide a functional complete set of operations which works on data of arbitrary length. With parallel computing language libraries like CUDA C provided by NVIDIA, programmers are able to allocate GPU memories and run parallel functions called "kernels" in a C/C++ like environment (NVIDIA 2010; Sanders and Kandrot 2011). These new hardware and software designs have brought enormous prosperousness to GPU-enabled computing and processing including collision detection.

Most of the hardware-assisted image-space collision detection methods make use of OpenGL buffers such as depth buffer, stencil buffer, and color buffer (Baciu et al. 1999, Vassilev et al. 2001). But these methods require data read-back, which is often time-consuming due to the asymmetric accelerated graphics port buses in common graphic cards. Moreover, these methods can only be applied in convex objects. Cai et al. (2006) proposed an image-space method using multiple projections to handle convex objects. But the algorithm relies highly on the shape of objects. To achieve correct collision results for complex shapes, the number of projection screens required could be very large.

The *collision detection algorithm* that will be used in this research addresses the following points:

(1) Correctly and efficiently handling arbitrary shapes;
(2) Avoid data read-back in the collision detection process;
(3) Use a single rasterization process to perform collision detection for each component; and
(4) Obtain collision position information during the collision detection process.

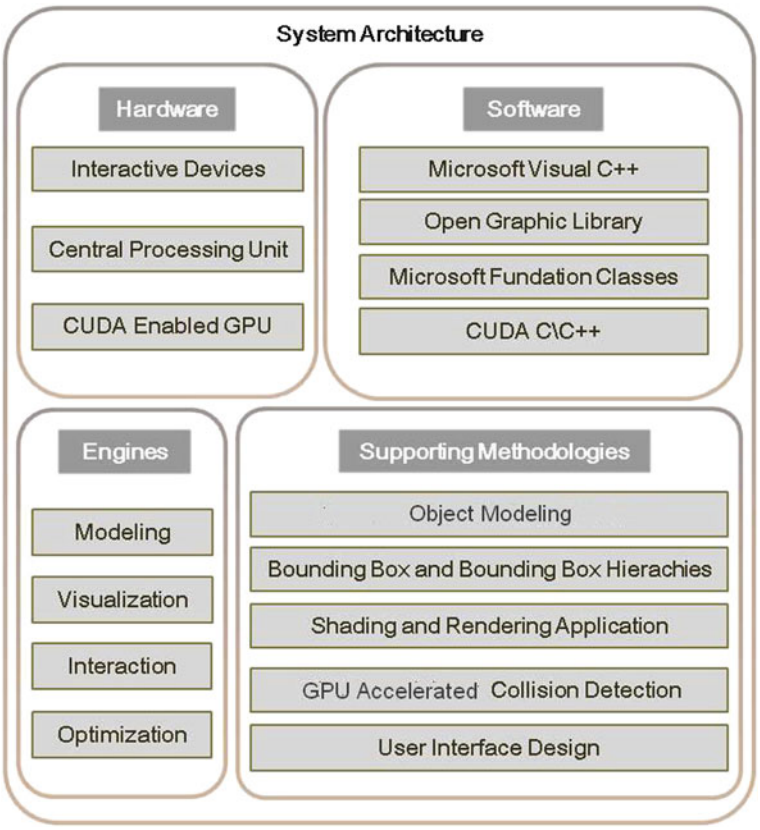

Fig. 1 Simulation system architecture

2 Collision Detection Supported Simulation System Design

Figure 1 illustrates the *overall system structure* for simulation application which consists of three components: *hardware*, *software*, and *methodology*.

2.1 Hardware Component

The fundamental computer hardware used for the system development is a workstation equipped with 1. 86 GHz Intel Core 2 Duo processor E6300 and 4 GB system memory based on windows operating system. The basic simulation operations include *modeling*, *visualization* and *interactions* using central processing unit (CPU). The most challenging aspect to develop a simulator is the *efficient real-time collision detection*. The use of GPU in collision detection can significantly improve the performance of the simulator. In our system design, CUDA enabled GPU accelerated computing is achieved by using NVIDIA GeForce GTX 560Ti graphic card with 1 GB graphic memory. With 8 multiprocessors and 384 CUDA cores, GeForce GTX 560Ti card provides exceptional performance in real-time scientific rendering and processing. Users are able to launch at most 1,024 threads in a block with a maximum of 65,535 blocks per dimension in a grid.

2.2 Software Component

We develop real-time simulator using Visual C++ compatible on both Windows XP and Windows seven platforms. Open Graphic Library (OpenGL) is used as the Graphic Library for the purposes of model transformation, rasterization and display. Microsoft Foundation Class (MFC) is used for designing the Graphical User Interface (GUI). CUDA is a parallel programming platform supported by NVIDIA graphic cards which enable dramatic increases in computing performance by harnessing the power of the GPU. GPU-based collision test algorithms are developed using CUDA C\C++ in order to optimize the real-time performance.

2.3 Engines and Supporting Methodologies

We have four engines developed taking care of modeling, visualization, interaction, and optimization.

2.3.1 Modeling Engine

Any simulation model is constructed using the *standard geometric modeling techniques*. To optimize the collision detection, rendering, and visualization processes, a hierarchical scene graph is built for the virtual object model from which parameters for users' interaction may be deduced. A scene graph is a tree structure that represents objects or nodes in an order defined by the tree layout. A hierarchical tree structure makes modeling easier because it allows one to construct scene elements based on a hierarchy of objects. By inheriting a coordinate system from their parents, the children automatically follow their parent when the parent is moved. Figure 2 shows a two-party simulation system with the hierarchical tree structure representing each party of object models. Objects are organized into two groups: Party-1 and Party-2. Each of the groups has several components of objects either static or dynamic.

2.3.2 Visualization Engine

3D objects are represented in triangular (polygonal) meshes. The entire vertex coordinates and the vertex indices of triangular faces are extracted from input files

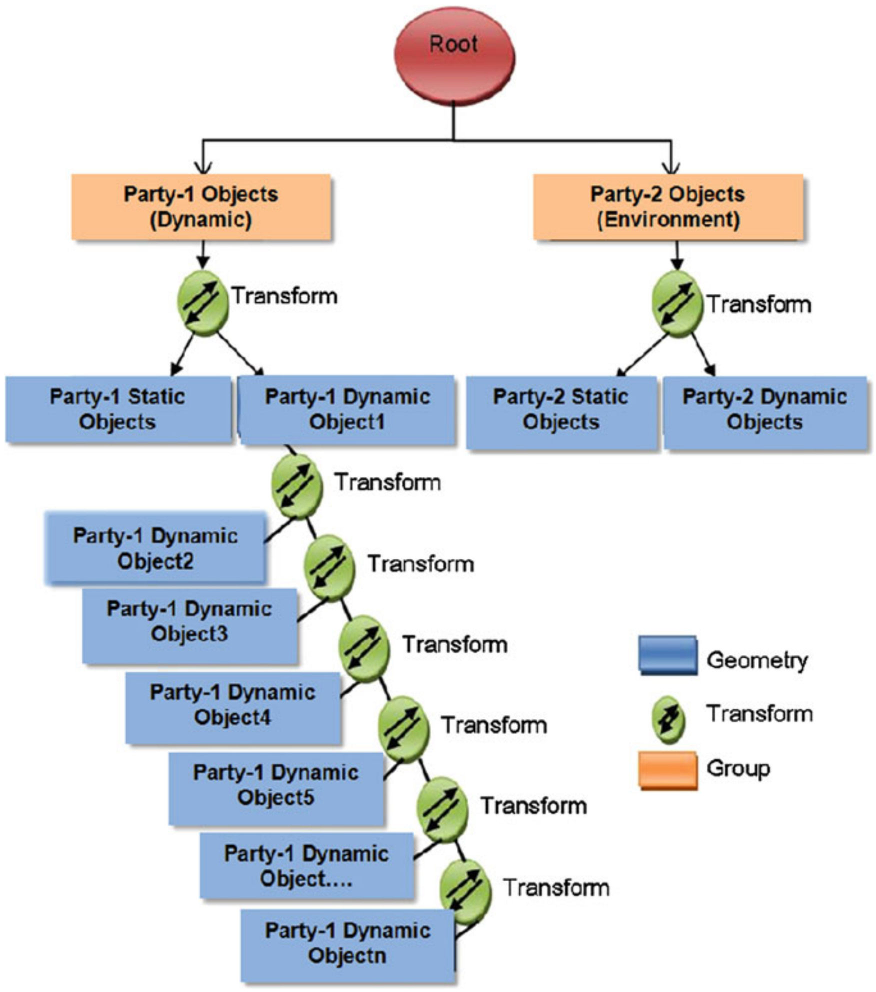

Fig. 2 Illustration of hierarchical tree structure of the two parties

and organized into triangular form to define the geometry of each object. Our system uses the OpenGL to render the 3D object models represented by triangular meshes.

2.3.3 Interaction Engine

Interaction relies on both software and hardware devices. Virtually all interactive devices can be used in the simulation. Selected interactive devices are identified due to the functions required for the simulation purpose. On top of that an interactive GUI is designed using MFC which provides a friendly and convenient manner for users to perform control or manipulation. Typical graphical interactions will include

(1) Translation
(2) Rotation
(3) Scaling
(4) Viewpoint changes (Iso view, front view, side view, etc.)
(5) Undo and redo
(6) Navigation or steering

Along with mouse and keyboard, joystick is also used as input device for our simulation. Generally, joystick device is more suitable for manipulation and transformation providing an interactive experience closer to reality.

2.3.4 Optimization Engine

Parallel features of GPU are explored to speed up the simulator, especially the collision detection engine. The whole collision detection algorithm is implemented in GPU exploiting its capability in rasterization, image compaction, and parallel sorting. Although parallel feature of GPUs is very powerful, performance of the algorithm is still restricted by issues such as huge GPU memory usage caused by high rasterization resolution. To solve this problem, bounding boxes and bounding box hierarchies are introduced to reduce the number of candidate triangles and diminish the raster image size before the rasterization stage. Axis Aligned Bounding Boxes (AABBs) (Ericson 2005) are built for all objects and a bounding volume hierarchy (BVH) is constructed for some object models if needed. As the BVH (Ericson 2005) for the objects contains both AABBs and Oriented Bounding Boxes (OBBs), we name it as a "hybrid BVH."

3 GPU-Enabled Collision Detection

The main idea of the collision detection algorithm is to cast vertical sample rays from a chosen plane and investigate object intersections along the rays to estimate the situation of the scene (Ericson 2005). The process is implemented by rasterization application and the "Sort_Search_Pair" algorithm described in Sect. 3.4.

3.1 Algorithm Overview

The work flow of the GPU collision detection algorithm is as follows:

(1) Update transformation matrices, AABBs and the Party-2 distance array;
(2) Filter Party-2 objects according to the distance map;
(3) Do collision checks between filtered Party-2 objects and the hybrid BVH of the Party-1 objects.

The process of the third step is stated in the following diagram:

Algorithm 1 : Pseudo-code of step 3 in the algorithm overview

```
for (each layer of the hybrid BVH)
{
    Launch Kernel1, $thd_i$ work on $Object_i$ in $O_r$ (set of Party-2 objects) and $AABB_i$ in $A_r$ (set of Party-2 AABBs)
    {
        if ($AABB_i$ not intersect with the Party-1 component AABB in the current node)
        {
            Remove $Object_i$ from $O_r$;
            Remove $AABB_i$ from $A_r$;
        }
    }
    if ($O_r$ is empty)
    {
        Report "No Collision";
        break for;
    }
    else
    {
        Launch Kernel2, $thd_i$ works on $Object_i$ in $O_r$ and $AABB_i$ in $A_r$

        {
            if ($AABB_i$ not intersect with the Party-1 component OBB linked to the current node)
            {
                Remove $Object_\text{i}$ from $O_r$;
                Remove $AABB_i$ from $A_r$;
            }
        }
    }
    if ($O_r$ is empty)
        continue for;
    else
    {
        Store triangles of objects in $O_r$ into $T_r$;
        Launch Kernel3, $thd_i$ works on $Triangle_i$ in $T_r$
        {
            if ($Triangle_i$ does not intersect with the Party-1 component OBB linked to the current node)
                Remove $Triangle_i$ from $T_r$; Compact $T_r$;
        }
    }
    if ($T_r$ is empty)
        continue for;
    else
    {
        Combine $T_r$ and triangles from Party-1 component mesh data linked with the current node into $T_c$ (set of candidate triangles);
        Pre-rasterize triangles in $T_c$; (see Section 2.3.4)
        Rasterize triangles in $T_c$.;
        Launch the "Sort_Search_Pair" algorithm (see Section 2.3.4);
        if (collision detected)
            Report ObjectID and TriangleID;
        else
            continue for;
    }
}
```

3.2 Collision Detection Data Sets

With a big collection of stream processors, the Single Instruction Multiple Data (SIMD) model of GPU (NVIDIA 2010) has significant advantages in processing uniformly represented data. Thus, in this collision detection algorithm, geometric mesh data, bounding boxes, and transformation matrices are stored as GPU styled linear data sets (Fig. 3).

Linear data sets included in the algorithm are:

(1) Party-1 Vertex Set: The coordinates of vertices in the Party-1 object mesh data.
    This set is static as the transformations are represented in the matrix streams defined later.
(2) Party 1 AABB Set and Party-1 OBB Set: The corresponding information about AABBs and OBBs maintained for components of the Party-1 objects.
    The OBBs will not change once built but the AABBs need to be updated at each rendering time step;
(3) Party-1 Matrix Set: The transformation matrices for Party-1 components dynamically modified from user manipulation.
    Vertices in the Party-1 vertex set and OBB information in the Party-1Matrix Set are multiplied by corresponding matrices before use.
(4) Party-2 Vertex Set: The coordinates of vertices in Party-2 object mesh data.
    These vertices will also not be changed during the whole lifespan of the simulation.
(5) Party-2 AABB Set and Party-2 Distance Set: AABB representations and Distance value from the center of AABB to the center of Party-1 objects.
    These two sets serve as initial sifters for candidate triangles for rasterization.

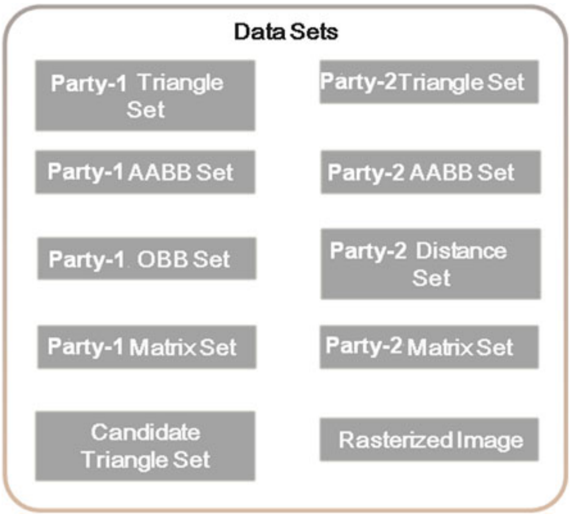

Fig. 3 Linear data sets in the collision detection engine

(6) Party-2 Matrix Set: The transformation matrices for Party-2 objects. The function of this set is similar to the Party-1 Matrix Stream.
(7) Candidate Triangle Set: Vertex references, ID of the objects that the triangles belong to, ID of the triangles in the related objects.
(8) Rasterized Image Set: The image data prepared for rasterization result. The size of the image may change for different Party-1 components.

3.3 Updating the Hybrid BVH

As stated in the algorithm overview, the hybrid BVH is used to cull Party-2 objects and triangles in Party-2 object meshes. The overall structure of the hybrid BVH is a tree with one AABB node and one OBB node at each layer as child nodes of the AABB node in the previous layer. Geometric primitives of Party-1 components are linked to their OBB nodes. The sequence of OBBs in each layer is pre-defined regarding the properties of these Party-1 components. AABBs in layers are built for the set of Party-1 components down from the current layer. The structure of the hybrid BVH is shown in Fig. 4.

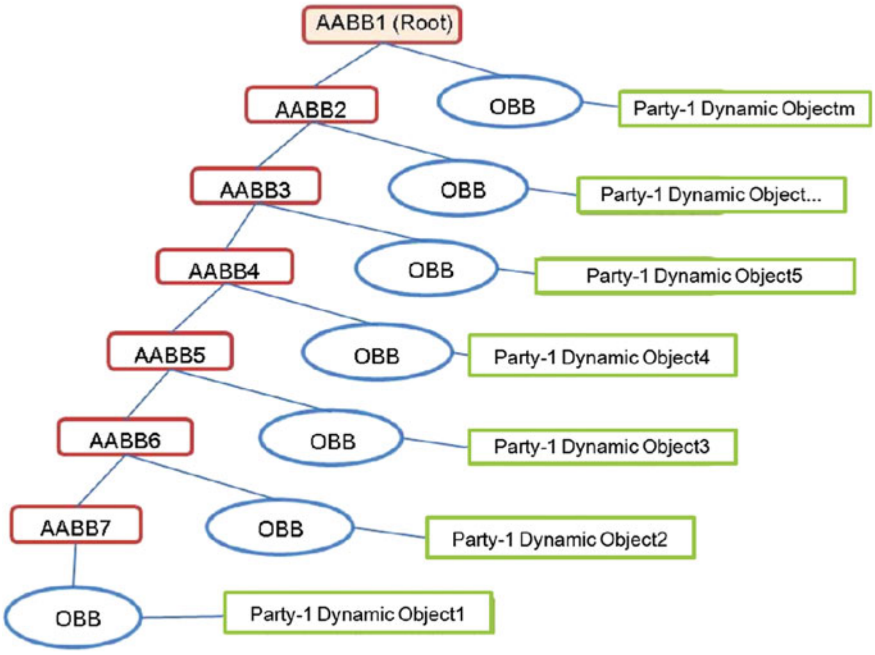

Fig. 4 The hybrid BVH of the crane. *Red blocks* stand for the AABB of the set of all Party-1 components (from object 1 to object m) in lower layers. *Blue blocks* stand for the OBB of the Party-1 component linked to the *blue boxes*

In the BVH updating process, OBBs are first transformed by transformation matrices recording its current direction and location. Then, the system will start the tree traversal to update AABBs via querying for maximum and minimum values of OBB vertices. This can be done in a single query as if we start the search from the bottom in the scene graph shown in Fig. 2.

3.4 Rasterization and the "Sort_Search_Pair" Algorithm

The specially designed rasterization process is the core of the GPU collision detection algorithm. The work flow of it is shown in Fig. 5. Data sets related here are the candidate triangle set and the pixel image set.

First, all the candidate triangles are transformed into the rasterization plane aligned coordinate system. Then a projection matrix is applied on triangle vertices to get their position on the rasterization plane. Depth information is also correspondingly computed.

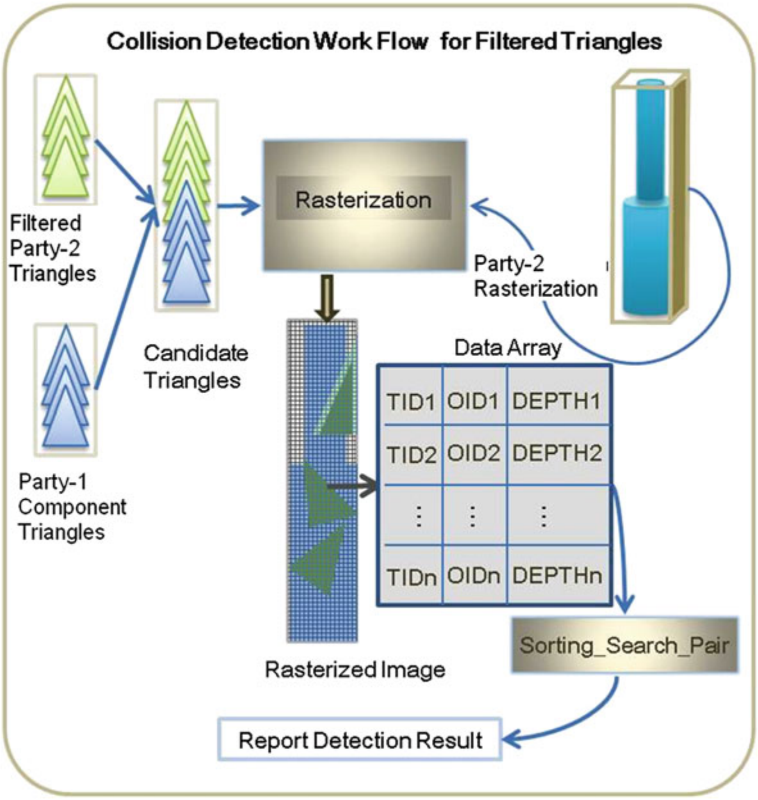

Fig. 5 The rasterization-based collision detection process for filtered Party-2 triangles

In the second step, edge equations of candidate triangles are computed. Functionality of edge equations is to decide the relationships between pixel center points and triangles. Then, for each triangle, a block-shaped GPU parallel kernel is launched for the planar AABB of projected triangle vertices. The algorithm will decide whether pixels in the AABB region are in the overlapped zone of the considered triangle. If the center of a pixel is inside the triangle, the triangle ID, object ID information, and interpolation depth are stored in the corresponding positions in the rasterized image. This stage is referred to as the pixel rasterization stage.

As the numbers of triangles intersecting pixels are not uniform, deciding the size of required memory for the rasterizaed image is a problem. Our solution is to introduce a pre-rasterized process conducted before the pixel rasterization stage to count the amount of triangles overlapping each pixel. Memory needed for rasterization can then be allocated according to the counted result.

The last step is the "Sort_Search_Pair" algorithm which is denoted in Algorithm 2 below:

Algorithm 2: Pseudo-code of the "Sort_Search_Pair" Algorithm

```
Launch Kernel, $thd_{i,j}$ works on $pixel_{i,j}$ in image I (defined in Algorithm 1.1)
{
        Sort the data array $d_{i,j}$ linked to $pixel_{i,j}$ by descending depth value;
        index ← 1;
        flag ← 1;
        while (index is no larger than the length of $d_{i,j}$)
        {
                if ($d_{i,j}$[index].ObjectID (see in Algorithm 1.1) belongs to Party-1)
                        flag ← –flag ;
                else
                {
                        if (flag equals -1)
                        {
                                Output $d_{i,j}$[index].ObjectID ;
                                Output $d_{i,j}$[index].TriangleID ;
                                break for;
                        }
                }
                index ← index+1;
        }
        Report "no collision in $pixel_{i,j}$";
}
```

4 Results

The collision detection part in the interaction engine is not fully implemented yet. Shown in Fig. 6 are some simple collision detection results using the described image-space algorithms for rasterization and collision detection. The blue boxes are Party-1 component and the gray ones denote Party-2 objects. The blue image at the bottom-right corner indicates the rasterized image with the red portion

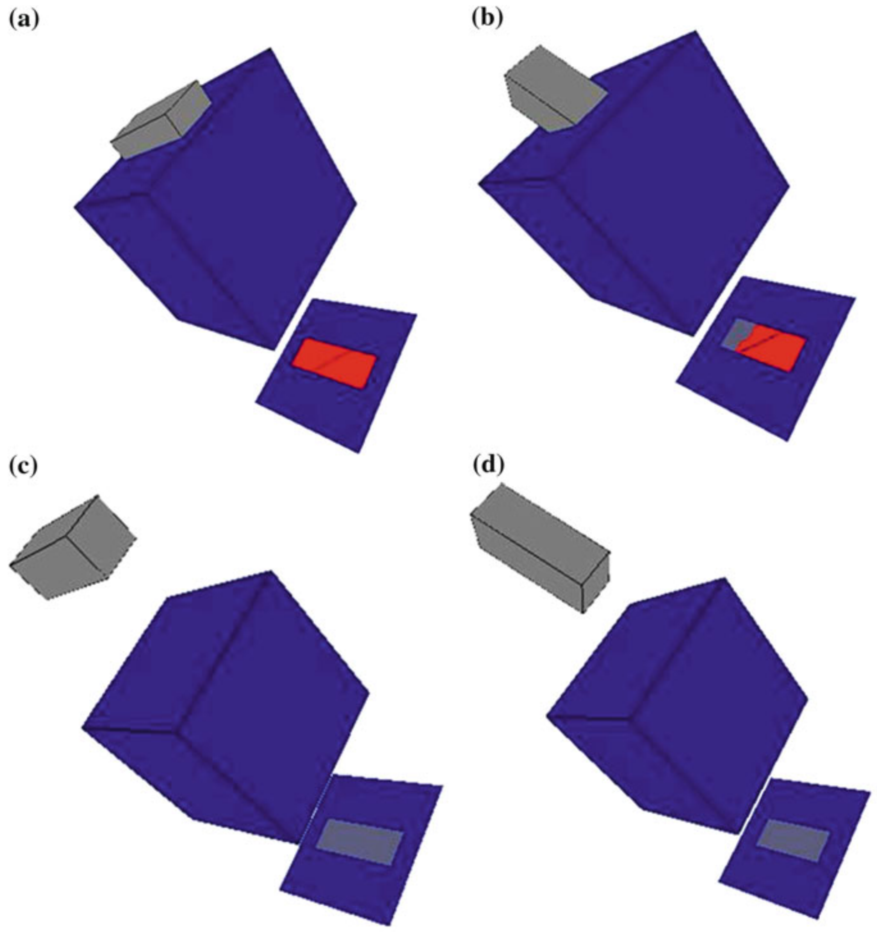

Fig. 6 Simple image rasterization and collision detection results. The *blue box* is one of the Party-1 components and the *gray box* denotes a Party-2 object. The blue image in the bottom-right corner indicates the rasterized image with the red portion referring to collision regions: **a** and (**b**) Collisions detected between the Parties; **c** and (**d**) Situations with no collision

referring to collision regions. Figures 6a, 6b show the cases where the Party-1 component collides with the Party-2 object, while Fig. 6c, 6d shows situations with no collision involved. The blue line in the red region is the overlapping of adjacent triangles in the blue box.

5 Conclusion

In this chapter we investigate the collision detection problem focusing on the use of AABBs technique and the GPU acceleration. Basic issues of simulation modeling, visualization, interaction and user interface are discussed as well. The modeling and hierarchical structure described here enables real-time computing benefiting from the latest GPU hardware acceleration. In particular, image-based algorithm with a hybrid BVH and distance set implemented in GPU is introduced to accelerate collision detection.

There are many applications for collision detection. Robotic path planning is an issue of collision avoidance. Real-time collision detection plays a central role in game, simulation, animation, and computer graphics.

In the future, we will optimize the data structure and the rasterization process to improve the efficiency and numerical accuracy. Implementation of the collision detection engine will be further modified for different industrial needs. Other minor issues such as rasterization plane choosing strategy might be altered to achieve better performance.

References

Baciu G, Wong W, Sun H (1999) RECODE: an image-based collision detection algorithm. J Vis Comput Anim 10(4):181–192

Cai Y, Fan Z, Wan H, Gao S, Lu B, Lim K (2006) Hardware-accelerated collision detection for 3D virtual reality games. Simul Gaming 37(4):476–490

Ericson C (2005) Real-time collision detection. Published in, Elsevier

Fauer F, Sebastien B, Jeremie A, Florent F (2008) Image-based collision detection and response between arbitrary volume objects. Eurographics 2008:155–162

Govindaraju N, Redon S, Lin M, Manocha D (2003) CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. Graph Hardware 2003:25–32

Gottschalk S, Lin M, Manocha D (1996) OBBTree: A hierarchical structure for rapid interference detection. In Proc. of ACM SIGGRAH'96, pp 171–180

Heidelberger B, Teschner M, Gross M (2003) Real-time volumetric intersections of deformable objects. In Proc Vision Model Vis 2003:461–468

Hunter G (1978) Efficient computation and data structure for graphics. Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ Ph.D. dissertation, US

Jung D, Gupta K (1996) Octree-based hierarchical distance maps for collision detection. Int Proc Robot Autom 1996:454–459

NVIDIA (2010) NVIDIA CUDA C Programming Guide. NVIDIA CUDA^{TM}

Sanders J, Kandrot E (2011) Cuda by example: an introduction to general-purpose GPU programming. NVIDIA Corporation

Tang M, Manocha D, Tong R (2010) Multi-core collision detection between deformable models using front-based decomposition. Graph Models 72(2):7–23

Tang M, Manocha D, Lin J Tong R (2011) collision streams: fast GPU-based collision detection for deformable models. Symposium on interactive 3D graphics and games, pp 63–70

Vassilev T, Spanlang B, Chrysanthou Y (2001) Fast cloth animation on walking avatars. Comput Graph Forum 20(3):260–267

Zhou M, Gong M, Huang X, Guo B (2010) Data-parallel octrees for surface reconstruction. IEEE Trans Visual and Comput Graphics voi no. pp 669–681