Fast Ray-Triangle Intersection Computation Using Reconfigurable Hardware

Sung-Soo Kim, Seung-Woo Nam, and In-Ho Lee

Digital Content Research Division,
Electronics and Telecommunications Research Institute,
305-700, 161 Gajeong-dong, Yuseong-gu, Daejeon, South Korea

Abstract. We present a novel FPGA-accelerated architecture for fast collision detection among rigid bodies. This paper describes the design of the hardware architecture for several primitive intersection testing components implemented on a multi-FPGA Xilinx Virtex-II prototyping system. We focus on the acceleration of ray-triangle intersection operation which is the one of the most important operations in various applications such as collision detection and ray tracing.

Our implementation result is a hardware-accelerated ray-triangle intersection engine that is capable of out-performing a 2.8 GHz Xeon processor, running a well-known high performance software ray-triangle intersection algorithm, by up to a factor of seventy. In addition, we demonstrate that the proposed approach could prove to be faster than current GPU-based algorithms as well as CPU based algorithms for ray-triangle intersection.

Keywords: Collision Detection, Graphics Hardware, Intersection Testing, Ray Tracing.

1 Introduction

The problem of fast and reliable collision detection has been extensively studied [4]. Despite the vast literature, real-time collision detection remains one of the major bottlenecks for interactive physically-based simulation and ray tracing [1][12]. One of the challenges in the area is to develop the *custom hardware* for collision detection and ray tracing. However, one major difficulty for implementing hardware is the multitude of collision detection and ray tracing algorithms. Dozens of algorithms and data structures exist for hierarchical scene traversal and intersection computation. Though the performance of these algorithms seems to be similar to software implementations, their applicability to hardware implementation has not yet been thoroughly investigated.

Since collision detection is such a fundamental task, it is highly desirable to have hardware acceleration available just like 3D graphics accelerators. Using specialized hardware, the system's CPU can be freed from computing collisions.

Main Results: We present a novel FPGA-accelerated architecture for fast collision detection among rigid bodies. Our proposed custom hardware for collision detection supports 13 intersection types among rigid bodies. In order to evaluate the proposed hardware architecture, we have performed the VHDL implementation for various intersection computations among collision primitives.

A. Gagalowicz and W. Philips (Eds.): MIRAGE 2007, LNCS 4418, pp. 70–81, 2007.

We demonstrate the effectiveness of our hardware architecture for collision queries in three scenarios: (a) ray-triangle intersection computation with 260 thousands of static triangles, (b) the same computation with dynamic triangles and (c) dynamic sphere-sphere intersection tesing. The performance of our FPGA-based hardware varies between 30 and 60 msec, depending on the complexity of the scene and the types of collision primitives. In order to evaluate our hardware performance for large triangle meshes, we also present our hardware to different benchmark models. For our comparative study we also analyze three popular *ray-triangle intersection algorithms* to estimate on the size of hardware resource. More details are given in Section 4. As compared to prior methods, our hardware-accelerated system offers the following advantages:

- Direct applicability to collision objects with dynamically changing topologies since geometric transformation can be done in our proposed hardware;
- Sufficient memory in our board to buffer the ray-intersection input and output data and significant reduction in the number of data transmission;
- Up to an order of magnitude faster runtime performance over prior techniques for ray-triangle intersection testing;
- Interactive collision query computation on massively large triangulated models.

The rest of the paper is organized as follows. We briefly survey previous work on collision detection in Section 2. Section 3 describes the proposed hardware architecture for accelerating collision detection. We present our hardware implementation of ray-triangle intersection in Section 4. Finally, we analyze our implementation and compare its performance with prior methods in Section 5.

2 Related Work

The problems of collision detection and distance computation are well studied in the literature. We refer the readers to recent surveys [4]. In this section, we give a brief overview of related work on the collision detection, programmable GPU-based approaches, and the custom hardware for fast collision detection.

Collision Detection: Collision detection is one of the most studied problems in computer graphics. *Bounding volume hierarchies* (BVHs) are commonly used for collision detection and separation distance computation. Most collision detection schemes involve updates to bounding volumes, pairwise bounding volume tests, and pairwise feature tests between possibly-intersecting objects. Complex models or scenes are often organized into BVHs such as sphere trees [7], OBB-trees [5], AABB-trees, and k-DOP-trees [8]. Projection of bounding boxes extents on the coordinate axes is the basis of the sweep-and-prune technique [4]. However, these methods incur overhead for each time interval tested, spent *updating* bounding volumes and collision pruning data structures, regardless of the occurrence or frequency of collisions during the time interval.

Programmable GPU: With the new programmable GPU, tasks which are different from the traditional polygon rendering can explore their parallel programmability. The GPU can now be used as a general purpose SIMD processor, and, following this idea,

a lot of existing algorithms have been recently migrated to the GPU to solve problems as global illumination, linear algebra, image processing, and multigrid solvers in a fast way. Recently, GPU-based ray tracing approaches have been introduced [11]. Ray tracing was also mapped to rasterization hardware using programmable pipelines [11]. However, according to [12] it seems that an implementation on the GPU cannot gain a significant speed-up over a pure CPU-based implementation. This is probably because the GPU is a *streaming architecture*. Another disadvantage which they share with GPUs is the *limited memory*. Out-of-core solutions are in general not an alternative due to the high bandwidth needed.

Custom Hardware: The need for custom graphics hardware arise with the demand for interactive physically simulations and real-time rendering systems. The AR350 processor is a commercial product developed by Advanced Rendering Technologies for accelerating ray tracing [3]. Schmittler et al. proposed hardware architecture (SaarCOR) for real time ray tracing and implemented using an FPGA [14]. The performance of the SaarCOR depends on a number of scene-space-subdivisions.

The first publications of work on dedicated hardware for collision detection was presented in [15]. They focused on a space-efficient implementation of the algorithms, while we aim at maximum performance for various types of collision queries in this paper. In addition, they presented only a functional simulation, while we present a full VHDL implementation on an FPGA chip.

3 Hardware Architecture

In this section, we give an overview of hardware architecture for accelerating the collision detection. Our hardware architecture is based on a modular pipeline of collision detection. The proposed architecture includes three key parts such as *input registers*, the *collision detection engine*, and the *update engine* in the Fig. 1.

3.1 Input Registers and Transformer

Our proposed hardware has three inputs which are *counter register*, *primary data register file*, and *secondary data register file*. The *transformer* provides the geometric transformation functions for secondary objects to improve the performance. The counter register contains the number of primary objects and the number of secondary objects. The geometries of the primary objects are stored in the primary data register file. The secondary data register file also holds geometries of the secondary objects for collision queries. In our research, we suppose that the primary objects $\mathcal{P}$ change for each time. On the other hand, the secondary objects $\mathcal{S}$ does not change their geometries in local coordinate system. Therefore, the $\mathcal{S}$ just can be applied the geometric transformations such as translation and rotation. For instance, the triangulated models are $\mathcal{S}$ and rays are $\mathcal{P}$ to perform the intersection computations in ray tracing applications. More specifically, $\mathcal{S}$ denotes as $\mathcal{S} = \{(\mathcal{T}_1, ..., \mathcal{T}_n) |\ n \geq 1\}$, where $\mathcal{T}$ is a triangle defined by three vertices $V_j \in \mathbf{R}^3, j \in \{0, 1, 2\}$. The $\mathcal{P}$ is the set of rays which contain their origins O and directions D. When testing the intersection between the primary objects and secondary objects, we perform the following processing steps. First, we upload the

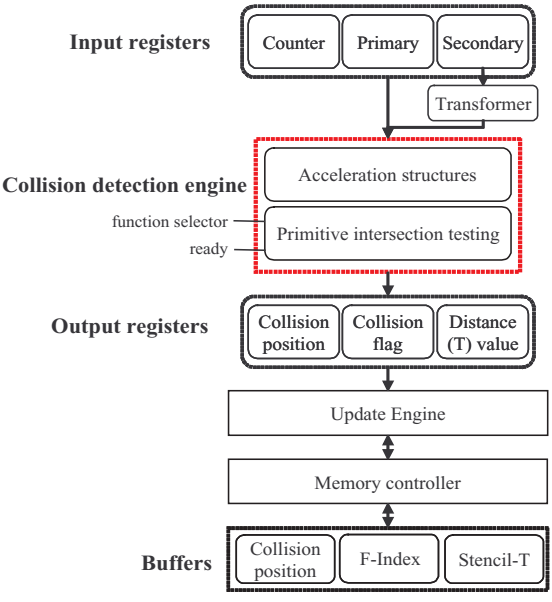


Fig. 1. The proposed hardware architecture

secondary objects to on-board memory at once through direct memory access (DMA) controller. Second, we transfer the primary objects to on-chip memory in the *collision detection engine* (CDE). To do this step, we use the register files which are packet data of the primary object to reduce the feeding time for the CDE. Finally, we invoke the ray-triangle intersection module in the CDE to compute the intersection between primary objects and secondary objects.

One of the primary benefits of the *transformer* in our architecture is to reduce the number of re-transmission for the secondary objects from main memory to on-board memory. If certain objects from the geometry buffer have to be reused, they can be transformed at the transformer without re-transmission from main memory. Therefore, we can reduce the bus bottleneck since we reduce the number of re-transmission. The bus width from *secondary register file* to CDE is 288 ($= 9 \times 32$) bits. We can transfer 288 bits to the collision detection engine in every clock. The ultimate goal of our work is applying our results to physically-based simulation. So, we use single precision for representing a floating point to provide more accurate results.

3.2 Collision Detection Engine

The collision detection engine (CDE) is a modular hardware component for performing the collision computations between $\mathcal{P}$ and $\mathcal{S}$. The CDE consists of the acceleration structures and primitive intersection testing components. As already discussed earlier in Section 2, a wide variety of acceleration schemes have been proposed for collision detection over the last two decades. For example, there are octrees, general BSP-trees, axis-aligned BSP-trees (kd-trees), uniform, non-uniform and hierarchical grids, BVHs,

and several hybrids of several of these methods. In our hardware architecture, we can adapt hierarchical acceleration structures for collision culling as shown in Fig. 1. However, we could not implement the acceleration structure due to the FPGA resource limit. But if we use the hierarchical acceleration structure, we can search the index or the smallest T-value much faster.

The primitive intersection testing component performs several operations for intersection computations among collision primitives. In order to provide various operations for intersection computations, we classified 13 types of intersection queries according to the primary and secondary collision primitives: ray-triangle, OBB-OBB, triangle-AABB, triangle-OBB, sphere-sphere, triangle-sphere, ray-cylinder, triangle-cylinder, cylinder-cylinder, OBB-cylinder, OBB-plane, ray-sphere, and sphere-OBB intersection testing. We have implemented hardware-based collision pipelines to verify these intersection types. The proposed hardware contains the 13 collision pipes, and more pipes can be available if hardware resources are sufficient. The CDE selects one collision pipe which is ready to working among 13 collision pipes by the *function selector* signal. Each pipe can be triggered in parallel by the *ready signal* of each pipe. However, it is difficult to execute each pipeline in parallel due to limitation of the input *bus width* and *routing* problems. Thus, our proposed hardware reads input packet from on-board memory and stores in the *register file* which contains two or more elements.

We use a *pipelined technique* in which multiple instructions are overlapped in execution. This technique is used for real hardware implementation in order to improve performance by *increasing instruction throughput*, as opposed to decreasing the execution time of an individual instruction. There are four outputs which are *collision flag* (F-value), *collision position* (CP), *index*, and *separation distance* or *penetration depth* (T-value). In order to get these outputs, the CDE performs the intersection testing between $\mathcal{P}$ and $\mathcal{S}$. If a collision occurs, CDE will store output values for CP, index, T-value and F-value. The CP denotes a collision position of the object pair and index is the triangle ($\mathcal{T}$) index of the triangulated mesh. The T-value denotes the penetration depth between two objects and F-value is set true. Otherwise, CP and index have invalid value, T-value is the separation distance between two objects and F-value is set false.

3.3 Update Engine

We can simplify routing data lines and make memory controller efficient by coupling buffers such as F-index buffer and two stencil-T buffers as shown in Fig. 1. We compare old T-value from stencil-T buffer0 (or 1) with new T-value from CDE and update smaller T-value from stencil-T buffer1 (or 0) of the two values within one clock. We do not transfer T-values from the stencil-T buffer to CPU in order to find the smallest or the largest T, which makes it possible to reduce transmission time. Stencil value in the stencil-T buffer is used for masking some regions of the F-index buffer to save searching time for the index of the collision object.

We use single precision floating point of IEEE standard 754 for representing each element of the vertex or vector and T-value in order to compare with the speed of the CPU arithmetic. One of the main reasons that we use single precision floating point is to provide more accurate results in physically-based simulation systems. So, we create many floating point arithmetic logics with CoreGen library supported by Xilinx tool

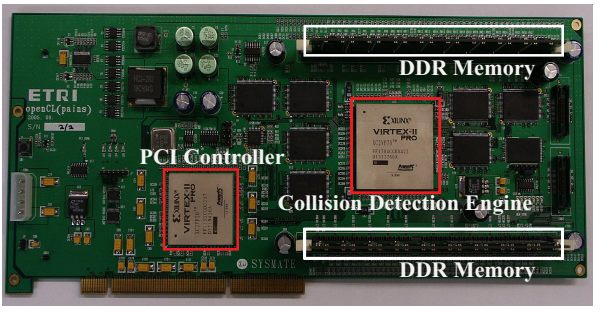


Fig. 2. The acceleration board with 64bits/66MHz PCI interface. On the board, there are Xilinx V2P20 for PCI controller, Xilinx V2P70 for memory and the collision detection logic. This board also includes two 1 GB DDR memories with 288 bit input bus, seven 2 MB SRAMs with 224 bit output bus.

ISE. We use two types of memories on the board. One is uploading-purpose memory which is consists of two DDR SDRAMs. The other is storing-purpose memory which is consist of six SRAMs to store output results (see Fig. 2). Block RAMs on the FPGA is used for buffering the $\mathcal{P}$. Primary register file matches the block RAM on the FPGA.

In our ray-triangle intersection computation, the primary object data $\mathcal{P}$ contains an origin point O and a direction vector D of a ray. Total 256 rays can be transferred from main memory to block RAMs on the FPGA at a time. Each secondary object data in $\mathcal{S}$ is a triangle $\mathcal{T}$ which contains three vertices. When the number of the rays is more than 256, the rays are divided by a packet which contains 256 rays and packets are transferred one by one at each step. We define this step as processing collision detection between a packet of primary object and all secondary objects. The bigger size of the block RAMs is, the better performance of the CDE is. While FPGAs usually have several small memories, the advantage of using such a memory is that the several memory blocks can be accessed in parallel. Each triangle of the secondary object is represented using 288 (9×32)-bit data. Nearly 55 million triangles can be transferred from main memory to two DDR SDRAMs on the board through the DMA controller. So, we designed the large bus width of the secondary object data to eliminate input bottleneck of the CDE. Therefore, we are able to read one triangle data from the queue of the DDR SDRAM in each hardware clock.

4 Analysis of Ray-Triangle Intersection Algorithms

In this section we present the analysis results for ray-triangle intersection algorithms in terms of hardware resources. We have investigated three major ray-triangle intersection algorithms, the first one is Badouel's algorithm [2], the second one is Möller and Trumbore's algorithm [9], and the last one is the algorithm using Plücker coordinates [10]. We review Möller and Trumbore's algorithm since this algorithm requires smaller hardware resources in terms of hardware implementation than others. We will skip Badouel's algorithm and the algorithm using Plücker coordinates and refer to the original publications instead [2][10].

Möller-Trumbore's Algorithm: The algorithm proposed by Möller and Trumbore does not test for intersection with the triangle's embedding plane and therefore does not require the plane equation parameters [9]. This is a big advantage mainly in terms of memory consumption – especially on the GPU and the custom hardware – and execution performance. The algorithm goes as follows:

1. In a series of transformations the triangle is first translated into the origin and then transformed to a right-angled unit triangle in the $y-z$ plane, with the ray direction aligned with x. This can be expressed by a linear equation

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{P \cdot E_1} \begin{pmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{pmatrix} \tag{1}$$

   where $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$, $T = O - V_0$, $P = D \times E_2$ and $Q = T \times E_1$.
2. This linear equation can now be solved to find the barycentric coordinates of the intersection point (u, v) and its distance t from the ray origin.

We compared these algorithms in terms of the latency, the number of I/O and hardware resources as shown in Table 1 and 2. We could not use Plücker test which contains too many multipliers and inputs relative to Möller's algorithm and Badouel's algorithm. Preprocessing of Plücker reduces the number of inputs and the latency of the hardware pipeline. However, it still needs more storage than others. Möller's algorithm is similar to Badouel's one in terms of the latency of the hardware pipeline, the number of I/O, and hardware resources as shown in Table 1 and 2. Möller's algorithm has been more efficient than Badouel's algorithm in view of the processing speed and usage of storage [9]. Therefore, we choose the Möller's algorithm for VHDL implementation for real circuit on the FPGA.

Table 1. Comparison of ray-triangle intersection algorithms in terms of the number of inputs, the number of outputs and latency for hardware implementation

Algorithms	The number of inputs	The number of outputs	Latency
Badouel's	9	6	16
Möller's	9	6	10
Plücker's	15	6	17

Table 2. Analysis of the hardware resources for ray-triangle intersection algorithms

Hardware Components	**Badouel's**	**Möller's**	**Plücker's**
Multiplier	27	27	54
Divider	2	1	1
Adder	13	12	31
Subtractor	23	15	17
Comparator	6	8	3
AND	3	2	2

5 Implementation and Analysis

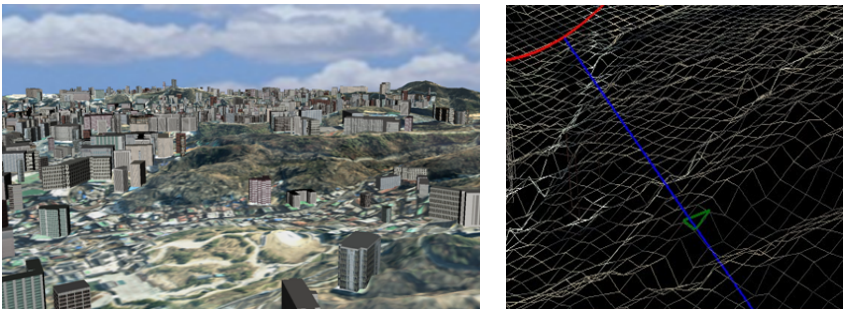
In this section we describe the implementation of our collision detection hardware and highlight its application to perform ray-triangle intersection testing for massive triangulated meshes.

5.1 Implementation

We have implemented ray-triangle collision detection engine with VHDL and simulated it with ModelSim by Mentor Graphics. The ray-triangle intersection algorithm which we used is Möller's algorithm. In order to evaluate our hardware architecture, we created this algorithm as circuits on an FPGA. In our experiments, the primary input is a dynamic ray and triangulated terrain which contains 259,572 triangles for secondary objects in Fig. 3(a). The origin of the ray moves on the flight path shown as a red curve and direction of the ray changes randomly in every frame in Fig. 3(b). We have evaluated our hardware on a PC running Windows XP operating system with an Intel Xeon 2.8GHz CPU, 2GB memory and an NVIDIA GeoForce 7800GT GPU. We used OpenGL as graphics API and Cg language for implementing the fragment programs [13]. We can classify three configurations of collision detections according to the properties of collision primitives. A *static object* is the object which the topology is not changed in the scene. On the other hand, a *dynamic object* is an object which the topology is changed in the scene for each frame.

Static Objects vs. Static Objects: In this scenario, the performance depends on the number of primary objects due to limitation of the block RAMs on an FPGA. Thus, we choose the objects which have small number of objects in our architecture. If the number of the objects is larger than the size of the block RAM, then data transmission from main memory to block RAM occurs in two or more times.

Static Objects vs. Dynamic Objects: We choose dynamic objects as the secondary object. Since the transformation is performed in our hardware, we do not need to



(a) (b)

Fig. 3. Our test terrain model: (a) terrain: 259,572 triangles (b) a ray (blue line) is shot on the triangulated terrain in arbitrary direction for each frame

retransfer data of dynamic objects except that objects are disappeared or generated newly. Position and orientation of the dynamic objects can be transformed by transformer in Fig. 3. We expect the performance is comparable to above case.

Dynamic Objects vs. Dynamic Objects: Our hardware architecture only supports transformation function for secondary objects. In this scenario, transmission time is defined by the number of the primary objects which are transformed in the CPU. Thus, the performance depends on the number of the primary objects and the CPU processing speed. We will evaluate performance of our proposed architecture in each case comparing with that of CPU and GPU. The proposed hardware checks 259,572 ray-triangle collision tests per frame, which takes 31 milliseconds including the ray data transmission time, while it takes 2,100 milliseconds for CPU based software implementation as shown in Fig. 4(a). Our hardware was about 70 times faster than CPU-based ray-triangle implementation. To compare with GPU-based approach, we have implemented the efficient ray-triangle intersection tests on the GPU using the Nvidia Cg shading language [6][13]. The proposed hardware is four times faster than the GPU-based ray-triangle intersection approach. For dynamically moving vertices of the triangles on the terrain, the proposed hardware was 30 times faster than the CPU-based approach as shown in Fig. 4(b).

We also performed another experiment for dynamic sphere-sphere collision detection. In this scenario, one thousand of sphere move dynamically in every frame. The input data contains a center point and a radius of the sphere which is represented four 32-bit floating points. In collision detection among dynamically moving spheres, our hardware is 1.4 times faster than CPU based implementation since sphere-sphere intersection algorithm consists very simple operations. In order to evaluate our hardware performance for large triangle meshes, we also have applied our hardware to different benchmarks as shown in Table 3. We measured the average computation time of ray-triangle intersection for each benchmark model. Our approach provides significant performance improvement for huge triangle meshes.

Fig. 6 shows snapshots of intersection trajectories for our benchmark models.

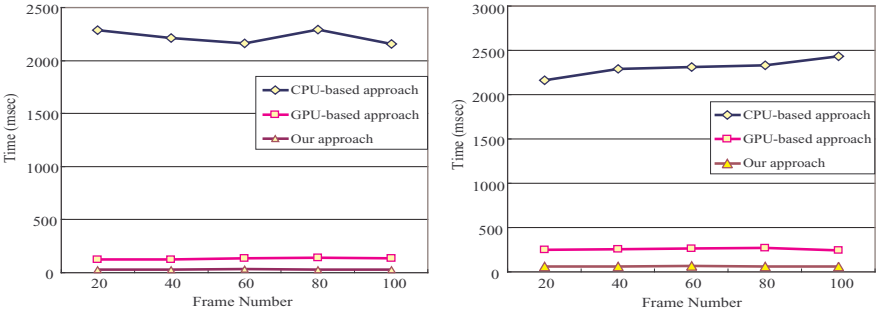
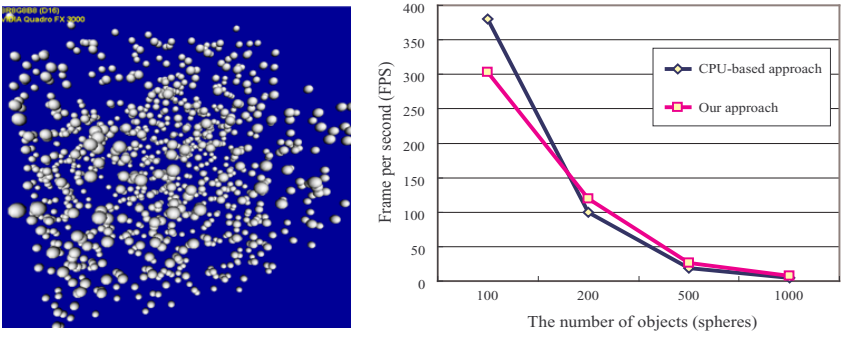


Fig. 4. The comparison result of the ray-triangle intersection testing: (a) static vs. static, (b) static vs. dynamic



(a) (b)

Fig. 5. Dynamic sphere-sphere collision testing: (a) snapshot of collision testing, (b) the comparison result according to the number of objects

Table 3. Performance comparison for ray-triangle intersection

Model	**Vertices**	**Triangles**	**CPU-based** (msec)	**Ours** (msec)
Bunny	35,947	69,451	488	4
Venus	50,002	100,002	695	5
Rabbit	67,039	134,073	925	8
Dragon	437,645	871,414	5.887	47
Happy buddha	543,644	1,085,634	7,290	53
Blade	882,954	1,765,388	11,220	65

5.2 Analysis and Limitations

Our hardware provides good performance of collision detection for large triangulated meshes. The overall benefit of our approach is due to two reasons:

- **Data reusability:** We exploit the *transformer* in the proposed hardware to avoid the transmission bottleneck due to the transformation in the CPU. As a result, we have observed 30 - 70 times improvement in ray-triangle intersection computation over prior methods based on CPU and GPU.
- **Runtime performance:** We use the high-speed processing power of the proposed hardware. We also utilize *instruction pipelining* to improve the throughput of the collision detection engine. Moreover, our current hardware implementation involves no hierarchy computation or update.

Based on these two reasons, we obtain considerable speedups over prior methods. Moreover, we are able to perform various collision queries at almost interactive frame rates.

Limitations: plusOur approach has a few limitations. Our hardware architecture includes the component of *acceleration structures*, such as kd-tree, grids and BVHs in Fig. 3. However, we could not implement this component due to the hardware resource

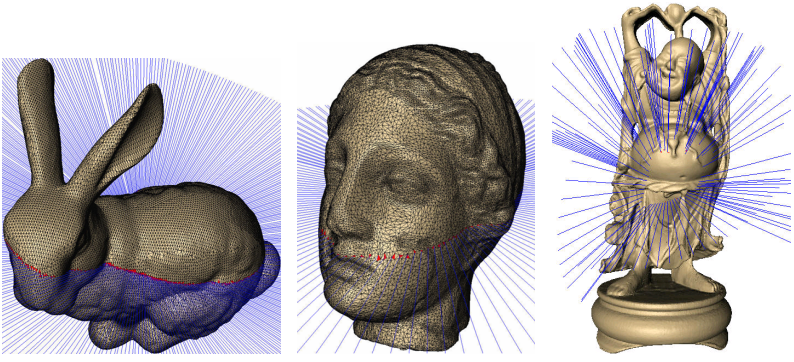


Fig. 6. Intersection trajectories of bunny, venus and happy buddha models. (The blue lines are random rays and the red triangles are intersected triangles with rays.)

limit. So, our current implementation does not support hierarchical collision detection. However, if traversal of acceleration structures is performed in CPU, we can solve this problem easily.

6 Conclusion

We present the dedicated hardware architecture to perform collision queries. We evaluate the hardware architecture for ray-triangle and sphere-sphere collision detection under the three configurations.

We have used our hardware to perform different collision queries (ray-triangle intersection, sphere-sphere intersection) in complex and dynamically moving models. The result is a hardware-accelerated ray-triangle intersection engine that is capable of outperforming a 2.8 GHz Xeon processor, running a well-known high performance software ray-triangle intersection algorithm, by up to a factor of seventy. In addition, we demonstrate that the proposed approach could prove to be faster than current GPU-based algorithms as well as CPU based algorithms for ray-triangle intersection.

References

1. N. Atay, J. W. Lockwood, and B. Bayazit, A Collision Detection Chip on Reconfigurable Hardware, In *Proceedings of Pacific Conference on Computer Graphics and Applications (Pacific Graphics)*, Oct. 2005.
2. D. Badouel, *An Efficient Ray-Polygon Intersection*, Graphics Gems I, pp. 390-394, 1990.
3. C. Cassagnabere, F. Rousselle, and C Renaud, Path Tracing Using AR350 Processor, In *Proceedings of the 2nd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, pp. 23-29, 2004.
4. C. Ericson, *Real-Time Collision Detection*, Morgan Kaufmann, 2004
5. S. Gottschalk, M. C. Lin, D. Manocha, OBB tree: A Hierarchical Structure for Rapid Interference Detection, In *Proceedings of ACM SIGGRAPH*, pp. 171-180, 1996.
6. Alexander Greß, Michael Guthe, and Reinhard Klein, GPU-based Collision Detection for Deformable Parameterized Surfaces, *Computer Graphics Forums (Eurographics 2006)*, Vol. 25, Num. 3, 2006.

7. P. M. Hubbard, Collision Detection for Interactive Graphics Applications, *IEEE Transactions on Visualization and Computer Graphics*, pp. 218-230, 1995
8. 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), pp. 21-36, 1998.
9. T. Möller and B. Trumbore, Fast, Minimum Storage Ray-Triangle Intersection *Journal of Graphics Tools*, pp.21-28, 1997.
10. J. Plücker, On A New Geometry Of Space, *Phil. Trans. Royal Soc. London*, 155:725-791, 1865.
11. T. J. Purcell, I. Buck, W. R. Mark, P. Hanrahan, Ray Tracing on Programmable Graphics Hardware, *ACM Transactions on Graphics*, 21(3), pp. 703-712, 2002.
12. A. Raabe, B. Bartyzel, J. K. Anlauf, and G. Zachmann, Hardware Accelerated Collision Detection-An Architecture and Simulation Result, In *Proceedings of IEEE Design Automation and Test in Europe Conference*, vol. 3, pp. 130-135, 2005.
13. Philippe C.D. Robert and Daniel Schweri, GPU-Based Ray-Triangle Intersection Testing, *Technical Report*, University of Bern, 2004.
14. Jörg Schmittler, Ingo Wald, and Philipp Slusallek, SaarCOR-A Hardware Architecture for Ray Tracing, In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 27-36, 2002.
15. G. Zachmann and G. Knittel, An architecture for hierarchical collision detection, In *Journal of WSCG'2003*, pp. 149-156, 2003.