Hardware Accelerated Collision Detection - An Architecture and Simulation Results

Andreas Raabe, Blazej Bartyzel, Joachim K. Anlauf, Gabriel Zachmann

Hardware Accelerated Collision Detection — An Architecture and Simulation Results

Andreas Raabe, Blazej Bartyzel, Joachim K. Anlauf
Technical Computer Science
Bonn University, Germany
{raabe, bartyzel, anlauf}@cs.uni-bonn.de

Gabriel Zachmann
Computer Graphics
Bonn University, Germany
zach@cs.uni-bonn.de

Abstract

We present a hardware architecture for a single-chip acceleration of an efficient hierarchical collision detection algorithm as well as simulation results for collision queries using this architecture. The architecture consists of two main stages, one for traversing simultaneously a hierarchy of discretely oriented polytopes, and one for intersecting triangles. Within each stage, the architecture is deeply pipelined and parallelized. For the first stage, we compare and evaluate different traversal schemes for bounding volume hierarchies.

A simulation in VHDL shows that a hardware implementation can offer a speed-up over a software implementation by orders of magnitude. Thus, real-time collision detection of complex objects at rates required by force-feedback and physically-based simulations can be achieved.

1. Introduction

Collision detection is a fundamental task in areas like animation systems, virtual reality, games, physically-based simulation, automatic path finding, virtual assembly simulation, and medical training and planning systems.

In many of these systems, collision avoidance is the ultimate goal. Most approaches today are reactive, i.e., they first place objects at a new trial position, then they check for collisions, and then compute new forces or positions, based on physical laws, so as to remove any collisions.

This approach poses very high demands on collision detection, because it must perform many collision checks per simulation cycle. A particularly demanding application is force-feedback, where updates of about 1000Hz must be done in order to achieve stable force computations. 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.

In this paper we present a new efficient architecture for hierarchical collision detection of two rigid objects using high-end ASIC technology.

We also present simulation results concerning its speed and size, which show that an implementation in dedicated hardware can speed up applications by at least an order of magnitude.

2. Related Work

Considerable work has been done on hierarchical collision detection in software [2,3,6,9,10]. Some of the bounding volumes (BVs) utilized are spheres, axis-aligned bounding boxes (AABB), oriented bounding boxes (OBB), and discretely oriented polytopes (DOP).

The first publications of work on dedicated hardware for collision detection was presented in [11, 12]. They focused on a space-efficient implementation of the algorithms, while we aim at maximum performance in this paper. In addition, they presented only a functional simulation, while we present a full VHDL implementation.

All other hardware-related research so far has tried to utilize existing graphics accelerator boards (GPU) [1,4,5,7,8]. While earlier approaches, such as [8], can basically handle only convex objects, later algorithms, such as [1, 7], have extended these to more general cases such as unions of convex objects or closed objects. The general class of "polygon soups" can be handled by [4], but they use a hybrid approach where the graphics hardware only identified potentially colliding sets.

All of the approaches using graphics hardware have the disadvantage that they either compete with the rendering process for the same hardware resource, or an additional graphics board must be spent for collision detection. The former slows down the overall frame rate considerably, while the latter would be a tremendous waste, since most of the resources of the hardware would not be utilized at all. Furthermore, most of these approaches work in image space, which reduces precision significantly.

3. The Algorithm

3.1. Hierarchical Collision Detection and Bounding Volumes

In this paper we concentrate on hierarchical collision detection. This avoids checking every triangle of an object A for collision with all triangles of object B by hierarchically grouping triangles (or other graphical primi-

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)
1530-1591/05 $ 20.00 IEEE

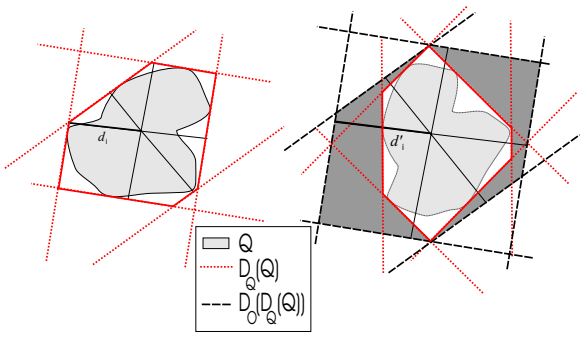


Figure 1. Our DOP overlap test gains its speed by transforming DOP Q into O's reference frame $F(O)$. The tightness loss is shown in dark grey. Obviously, each d' is determined by exactly three original d's.

tives). This yields a so-called *bounding volume hierarchy* (BVH), where each leave corresponds to one triangle and inner nodes correspond to groups of triangles. In order to achieve a feasible hardware design, we use a binary tree here, but n-ary trees could be considered as well. Each node of the tree stores a bounding volume (BV) that encloses all triangles in its group. Note that only leaves explicitely store any triangles.

In this work, we use k-DOPs as BVs because they were proven to yield very fast collision queries by extensive benchmarking in software [10], and are likely to perform well in hardware.

If two objects are checked for intersection, both hierarchies are traversed starting at both roots. If their BVs intersect, then the next level of BVs is checked. Since two objects will usually intersect only locally in a very small number of primitives, this yields a significant speed-up in the average case. In practical cases, the complexity is in $O(\log n)$ (n = number of primitives) because only a small diagonal "slice" of constant width down the BVH needs to be visited.

3.2. k-DOPs

As motivated above, we use k-DOPs as BVs. A k-DOP consists of k distances d_j along pairwise linearly independent vectors $\mathbf{B}_j$. Each of these vectors forms the normal of a halfspace. These vectors are chosen such that they form $k/2$ pairs, each of which defines a so-called *slab* [10]. The intersection of these slabs forms the BV:

$$D = \bigcap_{j=1,\ldots,k} H_j, \quad H_j : \mathbf{B}_j\mathbf{x} - d_j \le 0 \tag{1}$$

The orientation matrix B is fixed and equal for all objects. This yields a very space-efficient description for every k-DOP: only the k numbers d_j need to be stored. To avoid rounding errors we use single-precision floating-point numbers.

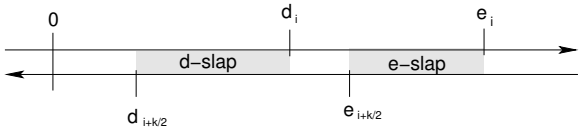


Figure 2. Two objects overlap, iff their projections intersect on every axis.

Each object O has its own reference frame (RF) $F(O)$ that describes its rotation R_O and translation T_O with respect to world coordinates. When an object moves, only R_O and T_O have to be updated. So checking two DOPs for intersection requires transformation of one of them into the RF of the other.

Assume O and Q to be two objects. Let $D_Q(Q)$ denote the minimum DOP which bounds Q with respect to the orientation matrix $\mathbf{B}$ in Q's own reference frame $F(Q)$. Since calculating $D_O(Q)$ is prohibitively expensive, we calculate $D_O(D_Q(Q))$, which is the minimum DOP in $F(O)$ bounding $D_Q(Q)$. Naturally, this incurs a loss of the BVs tightness to the underlying geometric structure.

Assume $\mathbf{M}$ to be the rotation and $\mathbf{o}$ the translation which transforms $F(Q)$ into $F(O)$. Then, we need to find distances d'_i which bound $\mathbf{M} \cdot D_Q(Q) + \mathbf{o}$ minimally.

Applying $\mathbf{M}$ and $\mathbf{o}$ to (1) yields

$$h_j : \mathbf{b}_j\mathbf{x} - d_j + \mathbf{b}_j\mathbf{o} \le 0, \quad \text{where } \mathbf{b}_j = \mathbf{B}_j\mathbf{M}^{-1} \tag{2}$$

Assume $D_O(D_Q(Q))$ is the intersection of

$$H_i : \mathbf{B}_i\mathbf{x} - d'_i \le 0, \quad i = 1,\ldots,k \tag{3}$$

Then, each d'_i corresponds to exactly one vertex of $D_Q(Q)$ and therefore to three d_j (see Fig. 1). These correspondences are the same for all nodes in an object's DOP hierarchy. So they can be determined at startup.

Let j_l, $0 \le l \le 2$, be the indices corresponding to d_i. Then,

$$\begin{pmatrix}\mathbf{b}_{j_0}\\ \mathbf{b}_{j_1}\\ \mathbf{b}_{j_2}\end{pmatrix}\mathbf{x} - \begin{pmatrix}\mathbf{d}_{j_0}\\ \mathbf{d}_{j_1}\\ \mathbf{d}_{j_2}\end{pmatrix} + \begin{pmatrix}\mathbf{b}_{j_0}\\ \mathbf{b}_{j_1}\\ \mathbf{b}_{j_2}\end{pmatrix}\mathbf{o} = \mathbf{0} \tag{4}$$

$$\mathbf{b}_i\mathbf{x} - d'_i = 0 \tag{5}$$

Equating (4) and (5) yields

$$d'_i = C_{ij_1}d_{j_1} + C_{ij_2}d_{j_2} + C_{ij_3}d_{j_3} + c_i \tag{6}$$

where C and c are chosen to be $C_{ij} := B_i\begin{pmatrix}\mathbf{b}_{j_0}\\ \mathbf{b}_{j_1}\\ \mathbf{b}_{j_2}\end{pmatrix}^{-1}$ and $c_i := b_i o$. Both are the same for all nodes in an object's DOP hierarchy; thus, they can be calculated at startup.

Checking $D_O(D_Q(Q))$ and $D_O(O)$ for intersection amounts to projecting them on the k axes given by B. They overlap if and only if there is no axis on which they are non-intersecting. Since there are always two antiparallel axes, we need to take that into account (see Fig. 2).

If **e** denotes the k-vector describing $D_O(O)$, then this test can be expressed as

$$\text{overlap} \Leftrightarrow \nexists i \in [1, \frac{k}{2}] : e_{i+\frac{k}{2}} > -d'_i \vee d'_{i+\frac{k}{2}} > -e_i \qquad (7)$$

3.3. Triangle Intersection

Once the BVH traversal reaches two leaves, we need to test the enclosed triangles for intersection. Here, we utilize the same algorithm that was already proposed in [11]. It transforms both triangles so that one of them becomes the "unit" triangle. That way, the checks to be performed on the other triangle become very simple and standardized.

For sake of reference, we just give one of the criteria for a non-intersection of one edge:

$$a < 0 \ \vee \ b < 0 \ \vee \ a + b > r_z \qquad (8)$$

where

$$\begin{aligned} a &= P_x Q_z - Q_x P_z \\ b &= P_y Q_z - Q_y P_z \end{aligned} \qquad (9)$$

with $\overline{PQ}$ being an edge of the triangle, and $\mathbf{r} = Q - P$. The other criterion is very similar.

4. The Architecture

To achieve maximum possible speed we assume to be using high-end hardware components: our target ASIC technology is a NEC UX5 CB-130 in 0.095μm-copper-technology. With a maximum of 61 gates in a row it can establish up to 800 MHz clockrate, which is our target frequency for the simulations (Sect. 5). Furthermore, we assume DDR2 memory modules.

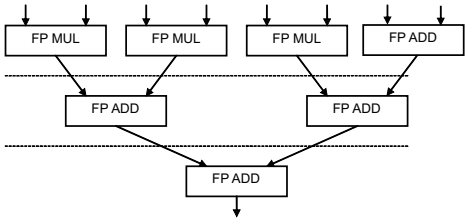
As BVs we chose to use 24-DOPs because extensive software benchmarking has shown 24 axes to be a good compromise of tightness and effort.

4.1. Design of the DOP Intersection Test

Our DOP intersection test is the combination of criterion (7) with (6), which amounts to the function

$$\widetilde{d}'_i = d_k C_{i0} + d_m C_{i1} + d_n C_{i2} + c_i + e_{i+\frac{k}{2}} \qquad (10)$$

This can be implemented as the following three-staged macro-pipeline, which we call DOPTRF unit (because it computes basically one row of the DOP transformation, i.e., Equation 6, plus part of the test, i.e., Equation 7):



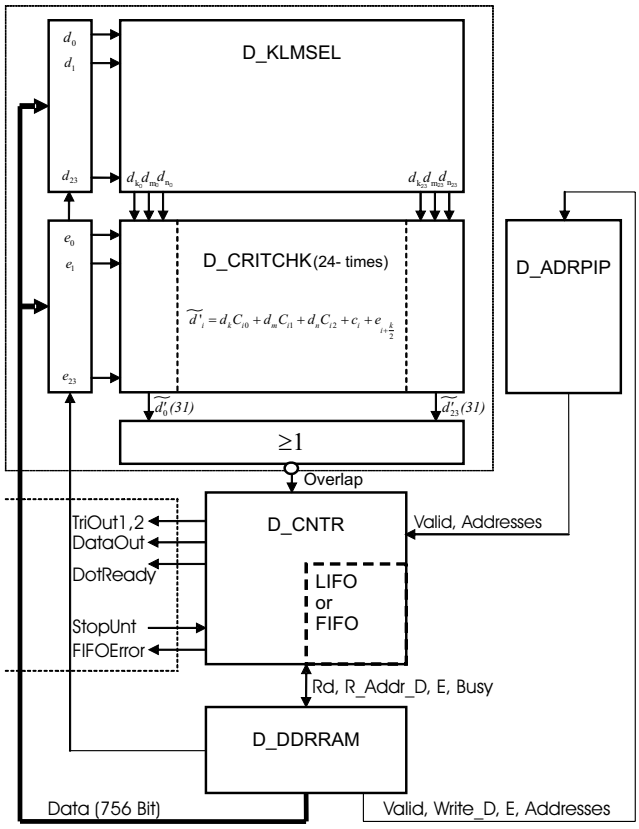


Figure 3. Architecture for DOP intersection testing.

The macro-pipeline stages were refined furthermore resulting in a pipeline of 15 stages and therefore an initialization delay of 15 clock cycles.

We use 24 DOPTRF units in parallel, and their results are NOR-reduced to check the criterion. To fill the pipeline we use a 756-bit wide bus from the DDR2-RAM. A hyper-multiplexer (D_KLMSEL) routes the correct inputs of the DOP to the DOPTRF units, which will then be transformed into the reference frame of the other DOP (see Fig. 3). A D_CNTR unit controls which DOP pair will be processed next (see Section 4.3).

4.2. Design of the Triangle Intersection Test

Using homogeneous coordinates, the affine transformations needed for the triangle intersection test (see Section 3.3) can be represented as 3×4 matrices.

The T_CHECK unit that performs the intersection test gets as input one triangle $V_A^i = [x_i y_i z_i 1], 1 \le i \le 2$, the precomputed matrix $M_B := [m_{ij}]$, and $M_{AB} := [b_{ij}]$ that transforms O's reference frame into Q's.

Calculating $\widetilde{V_A^i} = M_B \times M_{AB} \times V_A^i$ is done in the first two of 5 macro pipeline stages. These will be detailed in the following.

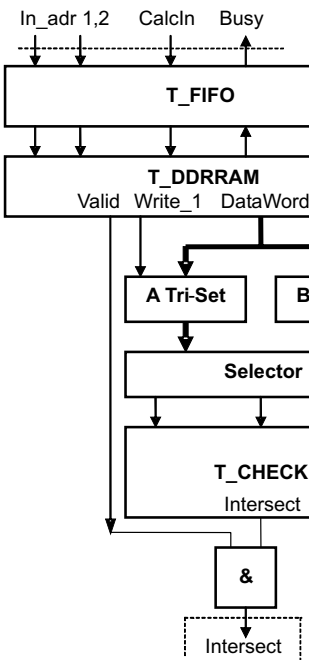


Figure 4. Architecture for Triangle Intersection

1st macro stage: The calculation of $M_b = [b'_{ij}] = M_B \times M_{AB}$ is split into three substages (marked with different brackets):

$$b'_{ij} = \{[(b_{i1}m_{1j}) + (b_{i2}m_{2j})] + (b_{i3}m_{3j})\} \quad (11)$$
$$b'_{i4} = \{[(b_{i1}m_{14}) + (b_{i2}m_{24})] + [(b_{i3}m_{34}) + (b_{i4})]\} \quad (12)$$
$$i, j = 1, \ldots, 3$$

The forth row of the resulting matrix is always [001]. These substages are further divided into microstages to gain maximum clock frequency. With this refinement, the first macro stage consumes 36 multiplication and 24 addition floating point units and takes 15 clock cycles delay to produce the first result.

2nd macro stage: Calculating $\widetilde{V_A^i} = M_b \times V_A^i$ works very similarly. The resulting substages are:

$$\widetilde{V^i} = \{[(b'_{i1}V_x^i) + (b'_{i2}V_y^i)] + [(b'_{i3}V_z^i) + (b'_{i4})]\} \quad (13)$$

Dividing this into microstages yields 27 multiplications, 27 additions, and another 15 clock cycles delay.

3rd macro stage: Before checking the edges of $\widetilde{T_A}$ for intersection with the unit triangle, we need to calculate a and b according to Eq. 9 for all three edges. Therefore, we need to calculate P_xQ_z, Q_xP_z, P_yQ_z, and Q_yP_z first. Additionally, we calculate r_z. This takes 12 multiplications, 3 additions, and 8 clock cycles. Calculating a and b from these terms takes another 6 additions and 4 clock cycles.

4th macro stage: For all three edges we now need to calculate $a + b - r_z$. After division into microstages this consumes 6 additions and 8 cycles.

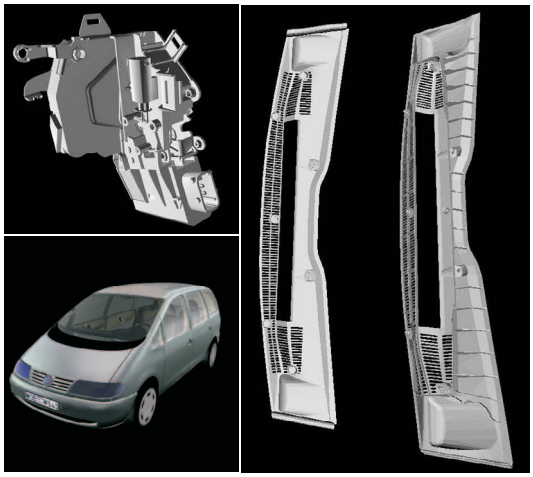


Figure 5. For benchmarking and testing, a number of different test objects with several polygon complexities were used.

5th macro stage: Here, we check the signs of a, b, and $a + b - r_z$ for all edges. This needs one clock cycle.

Overall pipeline: Putting all stages together, we get a pipeline with 52 clock cycles delay.

To fill the pipeline with data we need to buffer the triangle addresses, look them up in the DDR2_RAM which contains vertex data and the transformation matrices, and divide the data into two sets (because all edges of T_A have to be checked against T_B and vice versa). The whole TRI_UNIT is presented in Fig. 4.

4.3. Control

When the two object hierarchies are traversed symmetrically, each intersecting DOP pair results in 4 child pairs to be checked for intersection. However, other ways of traversing both BVHs can be more efficient. So far, we have compared those two possibilities that allow a small control unit and reduce the number of necessary memory accesses and transformations.

Traversing two BV hierarchies simultaneously and symmetrically basically amounts to traversing a 4-ary tree of BV pairs.

Let us denote the BVs in one tree by A, B, C, etc., and the BVs in the other tree by 1, 2, 3, etc.

Using a FIFO for determining the next DOP pair to be tested for intersection results in a plain breadth-first search on the collision tree. The processed DOP-sequence is
A1 - B2 B3 C2 C3 - D4 D5 E4 E5 - D6 D7 E6 E7 - F4 F5 G4 G5 - F6 F7 G6 G7

This can easily be optimized so that between testing two nodes of the same depth only one DOP needs to be fetched from memory. The resulting sequence is
A1 - B2 B3 C3 C2 - D4 D5 E5 E4 - E6 E7 D7 D6 - F6 F7 G7 G6 - G4 G5 F5 F4

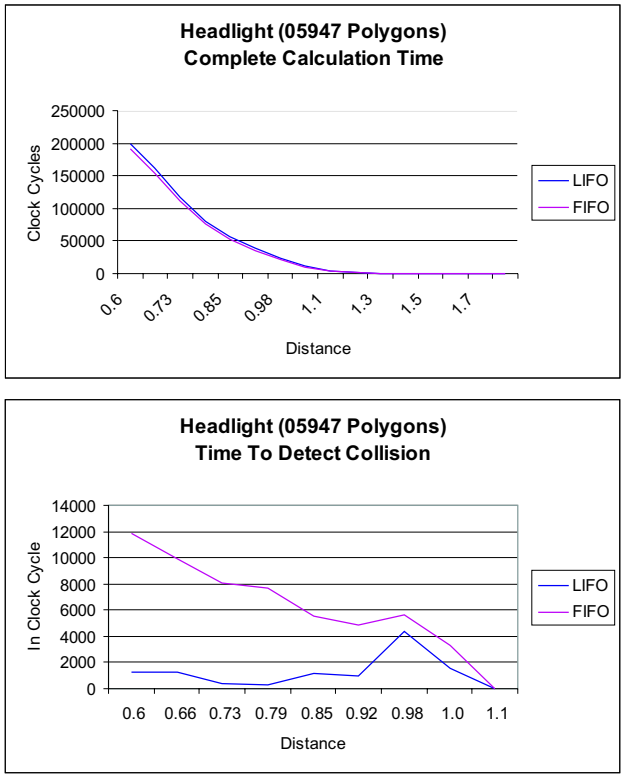


Figure 6. Top: Comparing LIFO and FIFO controlled intersection testing shows that finding all intersecting primitives takes equally long. Bottom: Our comparison shows that LIFO control is far superior to FIFO control when finding the first intersecting primitive.

Using a LIFO the sequence depends on the length of the pipeline. If we assume the pipeline to be of length one (for simplicity of presentation) we receive a plain depth first search.

A1 - B2 - D4 D5 E5 E4 - **B3** - E6 E7 D7 D6 - **C2** - G4 G5 F5 F4 - **C3** - F6 F7 G7 G6

If we push and pop nodes pairwise, we can easily reduce the number of necessary memory accesses as we did before.

Note that our DOP transformation pipeline has 15 stages. Thus, the traversal is not plain depth-first in the strictest sense. Instead, it proceeds along several paths in a depth-first manner.

5. Simulation Results

For benchmarking our architecture we used three different objects each of which in several polygon complexities (see Fig. 5). For each of them, two copies are placed at different distances from each other and with different rotations. For each constellation, the collision detection query time is determined.

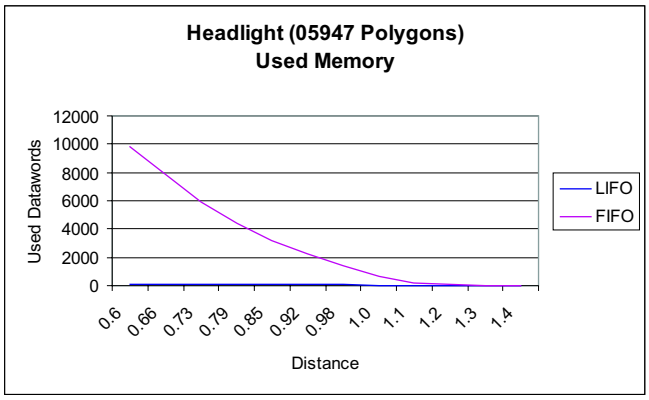


Figure 7. Although our hardware architecture implements a traversal scheme (LIFO') that is not quite a LIFO, it uses still only very few memory, compared to a FIFO.

Comparing the performance of LIFO and FIFO control, we see that finding all intersecting primitives takes equally long (see Fig. 6a).

Since using a LIFO corresponds to depth-first search on the collision tree, finding the first collision is usually much faster than using a FIFO. Our simulation results verify this (see Fig. 6b).

Another disadvantage of using a breadth-first search is the size needed for the memory structure. In the worst case, when all nodes need to be checked for intersection we must store them all in the FIFO before any leaves are checked and the queue size reduces.

With a strict depth-first traversal, the LIFO would need to be only as large as the depth of the BVH. However, as explained in Section 4.3, our traversal is not strictly depth-first. Fortunately, memory usage of the LIFO in our design seems to behave just as well (see Fig. 7).

Fig. 8 shows that our collision detection architecture, when implemented on a NEC CMOS ASIC (CB-130 UX5, 800 MHz), is up to 1000 times faster than the software implementation in determining all intersecting triangles of two objects. The software times were obtained on a 1GHz Pentium 3.

As mentioned above, a recent trend in computer graphics is to implement computationally intensive algorithms on the GPU. However, according to [5] 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.

6. Conclusion and Future Work

In this paper, we have presented, to our knowledge, the first simulation in VHDL of a hardware architecture for collision detection.

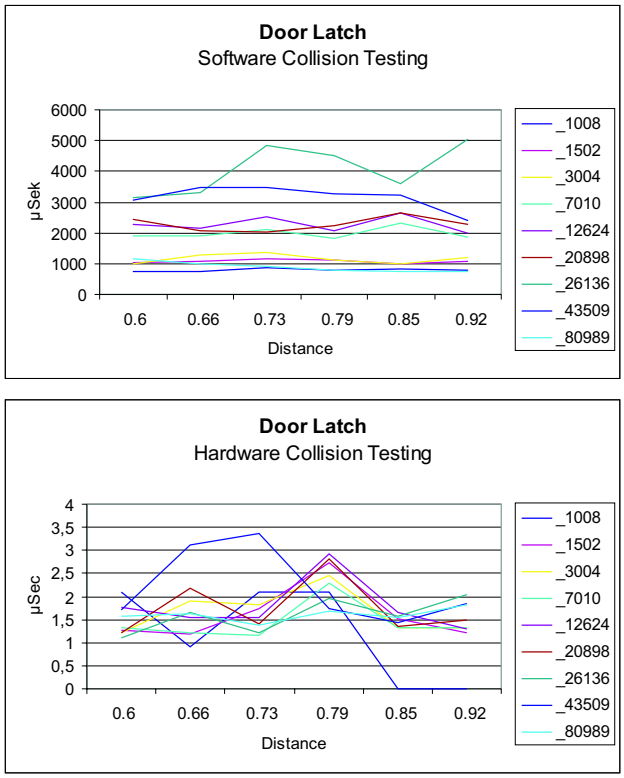


Figure 8. Comparison of software with hardware implementation at various distances and polygon complexities. Our collision detection architecture is up to 1000 times faster in determining all intersecting triangles of two objects than the software implementation running on a 1GHz Pentium 3.

We showed that hardware acceleration can be an effective way to speed up hierarchical collision detection. Using several pipelining and parallelization techniques, we achieved a speed-up of factor 1000 in VHDL simulations compared to a software implementation.

We also showed that, for simultaneous traversal of BVHs, a LIFO-controlled pipeline is far more space efficient than a FIFO-controlled one, without loss of processing speed.

Since the present paper is one of the first to look at hardware acceleration of collision detection, we believe there are many avenues for further research.

An important concern is the reduction of the bandwidth in order to make the bus from chip to memory smaller. Currently, we are investigating discretization and compression of the BVs. Furthermore, we will evaluate different kinds of primitives and BVs.

Another important issue is collision detection of deformable objects. It remains an open problem exactly which algorithms and data structures are best suited for hardware implementation.

Since we use a pipelined dataflow architecture, advanced pipelining techniques like speculative execution could be applied to reduce the number of pipeline stalls.

References

[1] G. Baciu, W. S.-K. Wong, and H. Sun. RECODE: an image-based collision detection algorithm. *The Journal of Visualization and Computer Animation*, 10(4):181–192, October - December 1999. ISSN 1049-8907.

[2] J. Eckstein and E. Schömer. Dynamic collision detection in virtual reality applications. In *WSCG'99*, pages 71–78, Plzen, Czech Republic, Feb. 1999. University of West Bohemia.

[3] S. Gottschalk, M. Lin, and D. Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. In H. Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, pages 171–180. ACM SIGGRAPH, Addison Wesley, Aug. 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[4] N. K. Govindaraju, S. Redon, M. C. Lin, and D. Manocha. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In *Graphics Hardware 2003*, pages 25–32, July 2003.

[5] A. Gress and G. Zachmann. Object-space interference detection on programmable graphics hardware. In *SIAM Conf. on Geometric Design and Computing*, Seattle, Washington, Nov.13–17 2003.

[6] P. M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, Sept. 1995. ISSN 1077-2626.

[7] D. Knott and D. K. Pai. CInDeR: Collision and interference detection in real-time using graphics hardware. In *Proc. of Graphics Interface*, Halifax, Nova Scotia,Canada, June11–13 2003.

[8] K. Myszkowski, O. G. Okunev, and T. L. Kunii. Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer*, 11(9):497–512, 1995. ISSN 0178-2789.

[9] G. J. A. van den Bergen. *Collision Detection in Interactive 3D Computer Animation*. PhD dissertation, Eindhoven University of Technology, 1999.

[10] G. Zachmann. Rapid collision detection by dynamically aligned DOP-trees. In *Proc. of IEEE Virtual Reality Annual International Symposium; VRAIS '98*, pages 90–97, Atlanta, Georgia, Mar. 1998.

[11] G. Zachmann and G. Knittel. An architecture for hierarchical collision detection. In *Journal of WSCG '2003*, pages 149–156, University of West Bohemia, Plzen, Czech Republic, Feb.3–7 2003.

[12] G. Zachmann and G. Knittel. High-performance collision detection hardware. Technical Report CG-2003-3, University Bonn, Informatikk II, Bonn, Germany, Aug. 2003.