Computer Graphics Technical Reports

CG-2003-3

High-Performance Collision Detection Hardware

Günter Knittel
WSI/GRIS, University of Tübingen. knittel@gris.uni-tuebingen.de

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

Institut für Informatik II
Universität Bonn
D-53117 Bonn, Germany

ISSN 1610-8892

High-Performance Collision Detection Hardware

Günter Knittel
WSI/GRIS
University of Tübingen
knittel@gris.uni-tuebingen.de

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

August 25, 2003

Abstract

We present a novel hardware architecture for a single-chip collision detection accelerator and algorithms for efficient hierarchical collision. We use a hierarchy of k-DOPs for maximum performance. A new hierarchy traversal algorithm and an optimized triangle-triangle intersection test reduce bandwidth and computational costs. The resulting hardware architecture can process two object hierarchies and identify intersecting triangles autonomously at high speed. Real-time collision detection of complex objects at rates required by force-feedback and physically-based simulations can be achieved even in worst-case configurations.

Keywords: graphics hardware, computer animation, virtual reality, hierarchical algorithms, triangle intersection.

1 Introduction

Collision detection is an elementary 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 or collision handling is the ultimate goal. Since algorithms for computing the exact time of collision are still too slow or too restrictive, most approaches are "reactive" in that they first place objects at a new position, and then handle collisions based on physical laws or constraints. This poses very high demands on collision detection performance, because usually the exact contact point(s) must be found by an iteration involving many collision checks per frame. Another very demanding application is rendering force-feedback, where collisions of an (invisible) surface contact object must be checked at about 1000Hz in order to achieve stable force computations.

It has been reported by many researchers that collision detection is still the major time-consuming step in many simulation or visualization applications [14]. Since collision detection is such a fundamental task, it would be highly desirable to have hardware acceleration available just like 3D graphics accelerators. Using specialized hardware, general-purpose processors can be freed from computing collisions. This will enable even low-end single-processor PCs and game consoles to do real-time collision detection in very complex scenarios at an affordable price.

In this paper, we propose an architecture which implements hierarchical collision detection for rigid objects in hardware. We have concentrated on hierarchical algorithms, because they have offered the best performance for so-called "polygon soups". Such a collision detection hardware will comprise the last stage of a collision detection pipeline [20]. This is where the bulk of the work is done in typical scenarios involving a modest number of objects with large polygon counts. We assume the hierarchies have already been computed. This is not a time-critical task, and can be done in software when the application loads objects at startup time.

In addition, we present a new traversal scheme that significantly reduces the computational costs and the memory bandwidth.

The next section describes related work, while Section 3 describes novel algorithms that are suitable for hardware implementation. Section 4 describes the hardware design in detail. Finally, Section 5 presents some benchmarks and simulation about the performance of the envisioned architecture.

2 Related Work

Considerable work has been done on hierarchical collision detection in software [4–6, 17, 19]. Some of the bounding volumes (BVs) utilized are spheres, axis-aligned bounding boxes (AABB), oriented bounding boxes (OBB), and discretely oriented polytopes (DOP). However, all traversal schemes proposed so far are inefficient in that they possibly visit the same nodes many times.

There is virtually no literature about the design of hardware architectures dedicated to collision detection. All research so far has tried to utilize existing graphics hardware. The approach taken by [2, 10, 13, 16] is to render the pair of objects with an orthogonal projection and count certain cases of overlapping intervals in the stencil buffer. This approach lends itself well to convex objects. Most of these algorithms cannot handle non-convex objects, and the most general class of "polygon soups" (which comprises non-closed objects, in particular) cannot be handled by this approach at all.

Another approach of utilizing the graphics hardware is to define a viewing volume (frustum or box) around one of the objects (the query object) and render the scene against that volume [11]. This is facilitated by OpenGL which can feed back the faces actually being rendered. This approach can be efficient for specific applications. However, it is not an accurate collision detection, unless the query object has the same shape as one of the two possible viewing volumes.

All of the approaches using graphics hardware have the disadvantage that they either compete with the rendering module for the graphics pipe, 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 overkill, since most of the resources of the hardware would not be made use of. Furthermore, these approaches work in image space, which reduces precision significantly.

A number of algorithms for ray-triangle and triangle-triangle intersection have been presented in the literature [1, 3, 7, 12, 15, 18]. Most of them compute either the barycentric coordinates or a number of 4×4 determinants. We propose a very efficient algorithm for checking intersection of triangles that does not need any division. Our new algorithm not only uses less multiplications and additions than [12] and [1], but is also very well suited for a hardware implementation due to a very uniform control and data flow.

3 The Algorithm

On this section we will describe our new traversal algorithm and then briefly recall the calclations for the overlap test of DOPs and the intersection test of triangles.

3.1 Hierarchy Traversal

The general, traditional scheme for hierarchical collision detection is a simultaneous, recursive traversal of two BV hierarchies (see Algorithm 1). However, this procedure incurs several penalties:

1. Nodes in both trees are usually visited several times; this is a general problem of all hierarchical collision detection algorithms (see Figure 1).
2. If the nodes have to be transformed (or other computations per node have to be performed), then this will be done several times for the same node.

The second penalty is a consequence of the first one; it could be alleviated by storing the result of the node transformation back into the node. Unfortunately, this has other disadvantages: first, the BV hierarchy occupies more memory (in the case of DOP trees, this would increase the memory usage by a factor 2); second, more importantly, the algorithm would no longer be thread-safe, so that multiple pairs of objects could no longer be checked in parallel. One could also precompute the transformation of a node's children during traversal and store the results on the stack. However, this reduces the number of node transformations only by a factor 2.

In contrast, our novel traversal scheme reduces the number of nodes visited, transfer volume from memory, and number of node transformations dramatically. Our traversal scheme only needs an additional small stack.

The idea is to avoid simultaneous traversal of two BV hierarchies. Instead, we traverse only one hierarchy and compare each node of that one with a list of nodes from the other hierarchy (see Figure 2). Let us call nodes that need to be transformed *tumbled nodes*, the other ones *aligned nodes* (see Figure 1). Assume that we are visiting a tumbled node A, and that a list $\mathcal{L}$ contains all

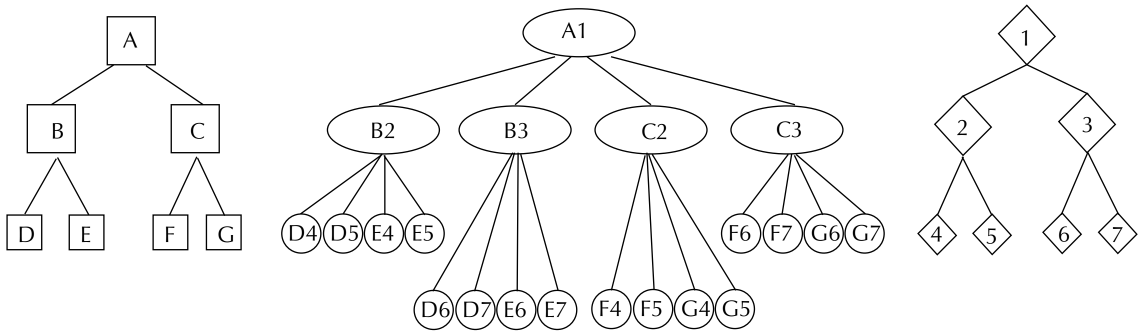

Figure 1: The simultaneous traversal of two BV hierarchies is, conceptually, equivalent to the traversal of a *BV pairs* hierarchy. Here, the right DOP tree is "tumbled" with respect to the DOP orientations of the left tree's reference frame.

traverse(A,B)
if A and B do not overlap then
  return
end if
if A and B are leaves then
  return intersection of primitives encl. by A and B
else
  for all children A[i] and B[j] do
    traverse(A[i],B[j])
  end for
end if

Algorithm 1: The traditional traversal scheme. A[i] and B[j] are the child nodes of A and B, resp. For sake of clarity, the "mixed" cases (one node is a leaf, the other is not) are omitted.

traverse(A,B,$\mathcal{L}$)
transform A
transform B
for all $N \in \mathcal{L}$ do
  if X and N do overlap then
    if X and N are leaves then
      check primitives enclosed by X and N
    else
      $\mathcal{L}'_X + = N_1, N_2$
    end if
  end if
end for
if A is an inner node then
  traverse(A_1, A_2,$\mathcal{L}'_A$)
else
  traverse(A,$\mathcal{L}'_A$)
end if
if B is an inner node then
  traverse(B_1, B_2,$\mathcal{L}'_B$)
else
  traverse(B,$\mathcal{L}'_B$)
end if

Algorithm 2: The new algorithm scheme for hierarchical collision detection that transforms each tumbled DOP only once, and that reduces the number of multiple visits of nodes by a factor 2. Operations involving node "X" are performed for both nodes A and B. They can be executed in parallel.

aligned nodes with which A needs to be checked for overlap. So we check all pairs $(A, \mathcal{L}^i)$; whenever such a pair overlaps, we append the two children $\mathcal{L}^i_j, j \in \{1, 2\}$, to a new list $\mathcal{L}'$. After $\mathcal{L}$ has been completely processed, $\mathcal{L}'$ contains all aligned nodes that need to be checked with A_1 and A_2, the two children of A. It is obvious that with this traversal we visit each tumbled node only once, and thus we transform the DOP stored with it exactly once.

This scheme works for all kinds of hierarchical collision detection, not just DOP trees. Depending on how much work per node-node overlap test can be factored out into one of the two nodes, the benefit of our new method can be dramatic.

For example, considering Figure 1, a possible sequence of pairs of nodes could be: A1 B2 D4 E4 D5 E5 C2 F4 G4 F5 G5 B3 C3. This means, that with the classical traversal the sequence of node transformations is: 1 2 4 4 5 5 2 4 4 5 5. In contrast, with our new traversal scheme, this sequence of visited node pairs is: A1 B2 C2 D4 E4 F4 G4 D5 E5 F5 G5 B3 C3, and the sequence of node transformations is: 1 2 4 5 3.

A hardware implementation allows us to improve the algorithm further by performing DOP overlap tests in parallel. We can exploit the fact that if two nodes A, B overlap, then we always need to check *all* children pairs (A_i, B_j). Consequently, instead of storing pointers to all children in the list $\mathcal{L}'$, we store only one pointer for each pair of siblings. By the nature of the binary tree, performing two overlap tests in parallel yields the greatest cost/performance benefit. To this end, we load a sibling pair of tumbled DOPs (A, B), transform them sequentially, and compare the two in parallel with each DOP from $\mathcal{L}$. This results in two new lists, one for child pair (A_1, A_2) and one for (B_1, B_2). In the sequential version described in the previous paragraph, we produced these two lists at very different times during the traversal, and we processed each of them twice; now, we produce those two lists simultaneously, and then we process each of them only once. [1] The benefit of this is that the time needed for overlap tests and the number of times an axis-aligned DOP needs to be transferred from memory is cut by a factor of two.

The pseudo-code in Algorithm 2 summarizes this new algorithm scheme. The traversal starts with list $\mathcal{L}$ initialized to the 4 pairs of first the level beneath the roots. Note that, for clarity, we have omitted the "mixed" cases. Note also that the last call of `traverse` is actually a call of an overloaded version, which has only slight differences from the algorithm shown here.

3.2 DOP overlap test

The basic operation of any hierarchical collision detection algorithm is the overlap check of two nodes from different objects. In this section, we briefly recall the calculations necessary for collision detection using DOP trees. The derivation of the following formulas can be found in [19].

DOPs are bounding volumes that are a generalization of axis-aligned bounding boxes. They have been introduced into computer graphics by [8]. DOP trees are a hierarchical representation of objects [9, 19]. Each node stores a DOP and pointers to its children which it encloses; leaves store pointers to polygons instead of children. A DOP is described by k numbers (hence *k-DOP*), usually represented by a vector of k floats. Extensive benchmarks have shown $k = 24$ to be optimal.

Given two objects O_A and O_B, and two DOPs $\mathbf{d}, \mathbf{e} \in \mathbb{R}^k$ from O_A and O_B's DOP trees, resp., the overlap test, as presented in [19], proceeds in two steps: first, DOP $\mathbf{d}$ from O_A's hierarchy is transformed into $\mathbf{d}'$ in the coordinate frame of O_B by

$$\mathbf{d}' = \mathbf{C} \times \mathbf{d} + \mathbf{c} \quad , \tag{1}$$

where

$$\mathbf{C} = \begin{pmatrix} \cdots & c_{0,0} & \cdots & c_{0,1} & \cdots & c_{0,2} & \cdots \\ \vdots & & & & & & \\ \cdots & c_{k-1,0} & \cdots & c_{k-1,1} & \cdots & c_{k-1,2} & \cdots \end{pmatrix}$$

where in matrix $\mathbf{C}$ exactly three entries per row are non-zero. Second, $\mathbf{d}'$ is compared componentwise with DOP $\mathbf{e}$ according to

$$\begin{aligned} &\exists i \le \tfrac{k}{2} : \; d'_i > e_{\frac{k}{2}+i} \;\vee\; e_i > d'_{\frac{k}{2}+i} \quad \Leftrightarrow \\ &\mathbf{d} \text{ and } \mathbf{e} \text{ do not overlap} \end{aligned} \tag{2}$$

where $d'_i < d'_{\frac{k}{2}+i}$ define a slab (analogously for all DOPs).

Matrix $\mathbf{C}$ and vector $\mathbf{c}$ depend only on the position of the two objects relative to each other. They are computed during the set-up by the software API of the collision detection hardware.

Since the $k \times k$-matrix $\mathbf{C}$ in Equation 1 has exactly 3 coefficients per row that are not 0, we can compute $\mathbf{d}'$ more efficiently by

$$d'_i = \mathbf{C}_i \begin{pmatrix} d_{j_{i,0}} \\ d_{j_{i,1}} \\ d_{j_{i,2}} \end{pmatrix} + c_i \tag{3}$$

where correspondence j stores the place of those coefficients which are not zero. So, by introducing a $k \times 3$ correspondence matrix j, we can reduce the size of the transformation matrix $\mathbf{C}$ to $k \times 3$. Consequently, the number of multiplications is $3k$.

3.3 Polygon intersection test

When the traversal reaches pairs of leaves (which containing triangles), a triangle-triangle intersection test has to be performed. In the following, we will briefly recall the calculations involved, which have already been described elsewhere [21].

The approach of our algorithm is to precompute a matrix M_B that transforms B into the unit triangle, and then check (conceptually) each edge of $A' = A \cdot M \cdot M_B$ against that unit triangle (and vice versa), where M is the transformation from O_A into O_B.

[1] This scheme can be generalized straight-forward to process 2^m tumbled nodes simultaneously.

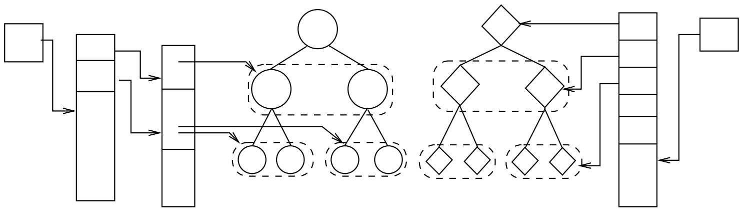

Figure 2: The improved traversal scheme can be implemented by a stack of lists. (In a hardware implementation, the stack on the right is merged into the left one.)

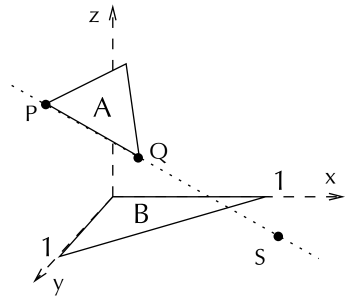

Figure 3: Using a special transformation, the intersection test can be done very efficiently.

This basically amounts to the following calculations and tests per edge $\overline{PQ}$ of A':

$$\begin{aligned} &P_x Q_z \geq Q_x P_z \qquad \wedge \\ &P_y Q_z \geq Q_y P_z \qquad \wedge \\ &P_x Q_z - Q_x P_z + P_y Q_z - Q_y P_z \leq Q_z - P_z \end{aligned} \tag{4}$$

The algorithm gains its special efficiency because we can precompute the matrices M_A and M_B (they can be obtained from a simple linear equation system), and because we do not need to compute the exact intersection point.

In our case of collision detection using DOP trees, we can store these matrices in the leaves instead of the DOPs. We do not need to check pairs of leaf DOPs, because the immediate check of triangles is faster. Storing the triangle matrix M_B and 3 vertices needs $3 \times 4 + 3 \times 3 = 21$ floats, which fit well into the nodes of a 24-DOP tree.

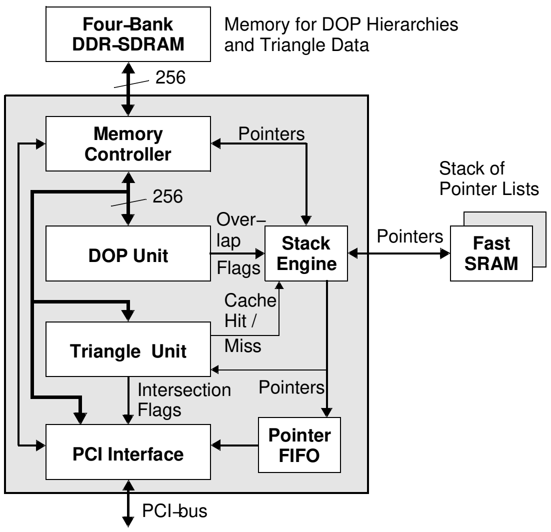

Figure 4: Block diagram of the CollisionChip and external Memory Systems.

4 Hardware Architecture

The target design is a PCI-board with one ASIC (or FPGA), a large on-board memory for the hierarchies, and two SRAM devices as dedicated stack memory. Crucial for the performance is the bandwidth towards the local memory, and so a four-bank SDRAM configuration with a 256-bit bus was chosen. The proposed circuitry is a high-performance, massively parallel implementation of the algorithms described above. Figure 4 shows as major functional units Memory Controller, Stack Engine, DOP Unit, Triangle Unit and PCI Interface. The Stack Engine performs the novel traversal algorithm as described in Section 3 by creating and processing lists of node pointers. It maintains these lists in two external fast SRAMs. Pointers are passed to the Memory Controller, which fetches the corresponding DOP-coefficients or triangle data from the SDRAM. The DOP-Unit performs the overlap test, while the Triangle Unit tests for triangle intersections. The identifiers of intersecting triangles are delivered to the software via a Pointer FIFO and the PCI Interface. We will explain all components in greater detail in the following sections.

4.1 DOP Unit

The DOP Unit performs the DOP transformation for "tumbled" nodes as described in Section 3, and the overlap test with all the "aligned" nodes in the current list. A schematic drawing is shown in Fig-

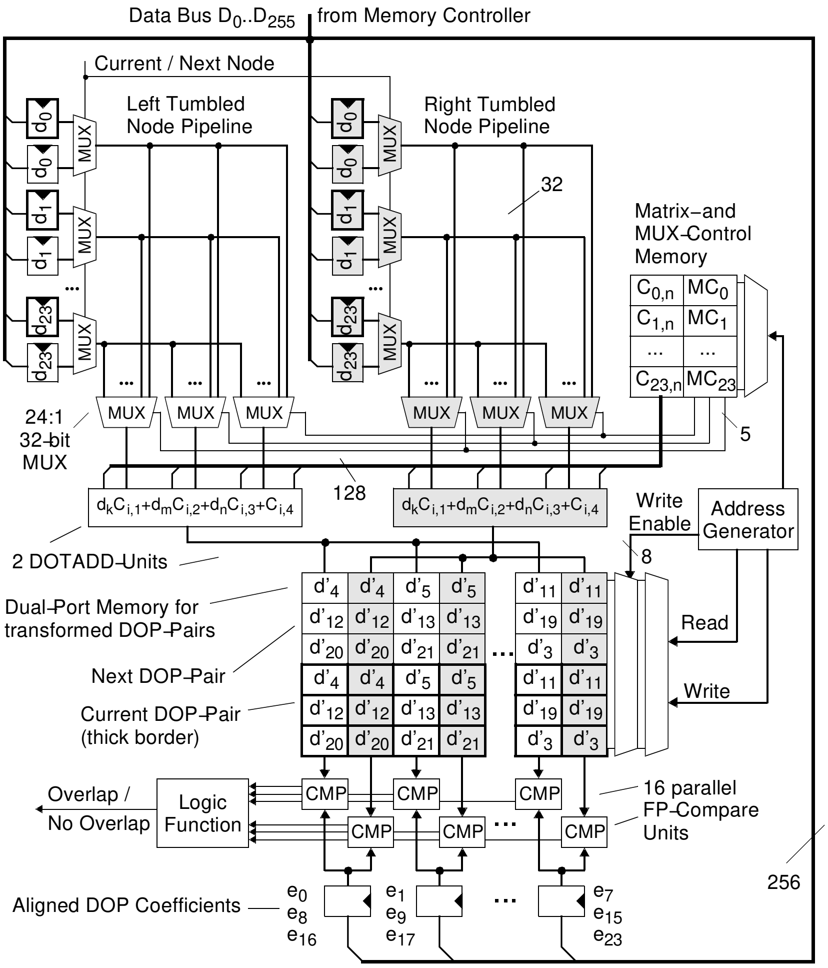

Figure 5: The DOP Unit checks pairs of DOPs for overlap. It has a throughput of two pairs every 4 cycles.

ure 5. As explained above, the algorithm always processes pairs (siblings) of tumbled nodes. Thus, there is one processing pipeline for the right (gray units) and one for the left sibling.

Prior to the processing of two hierarchies, the matrix **C** and a set of multiplexer control bits (MC) must be loaded into the on-chip Matrix- and MUX-Control Memory (MMM).

Once a pair of DOPs has been loaded into the two vertical register sets, transformation starts, under control of an Address Generator. This unit cycles through the MMM, which causes the proper sets of matrix elements and DOP coefficients to appear at the inputs of two so-called DOTADD-units. These pipelined units basically compute one dot product. They produce one DOP coefficient d' per clock, which are then written pairwise into a dual-port memory, which stores 8 pairs in one entry.

Once a tumbled DOP pair has been transformed, overlap check against all the aligned DOPs in the current list will be started. Since the on-chip data bus is 256 bits wide, the aligned node coefficients e will appear in bundles of eight. Each bundle is compared to the corresponding set of coefficients d' (note the dual-port memory layout), using a bank of 16 floating-point compare units. The overlap flags are then passed to the Stack Engine. Given the maximum overall data transfer rate, an overlap test of one aligned DOP against two tumbled DOPs takes a minimum of 4 clock cycles.

Once a tumbled DOP pair has been loaded from memory, however, the memory would be idle during the transformation. We exploit this by prefetching the two child DOPs of the left parent DOP, and

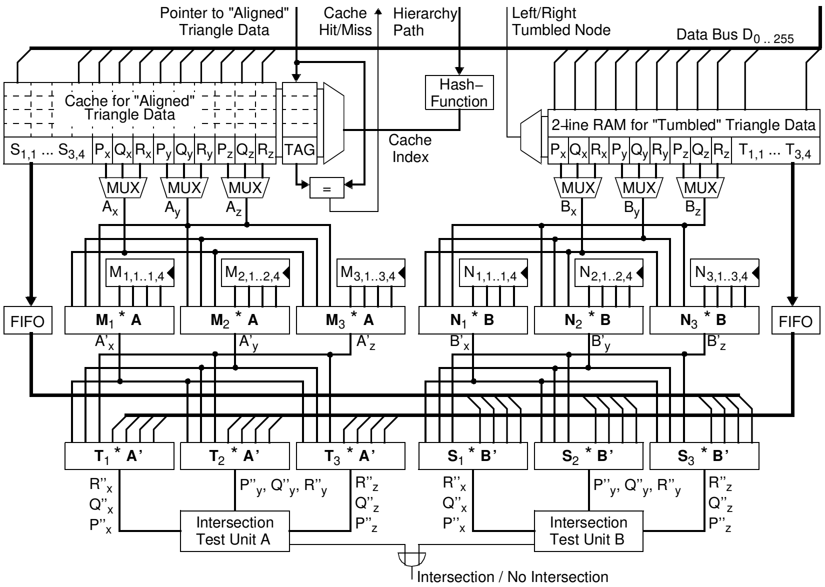

Figure 6: The first stage of the Triangle Unit transforms pairs of triangles. Only on the left side a small cache of about 11 kB is needed.

by transforming these coefficients speculatively during processing of the following list. Thus, the number of input registers holding d must be doubled as well as the memory capacity to store d'. In Figure 5, the two sets of storage elements are distinguished by thick or thin borders. It should also be clear why a dual-port memory is needed for the coefficients d': while a set of 16 coefficients is read for overlap test, two transformed coefficients of the children are written.

The concept of hiding the DOP transformation behind the overlap tests is used whenever possible. This means that the children of the left parent in the tumbled hierarchy are loaded and transformed as soon as the required resources are available. If the lists are long enough, processing of tumbled DOPs is hidden except for the memory read. Only in case there are no overlaps in a given list (and so no new lists are generated), an initial latency will occur. This initial latency is estimated to be 64 clock cycles, starting from the point when the node pointer is fetched from the SRAM and ending with two transformed DOPs being present in the dual-port memory.

Unfortunately, an efficient cache architecture for the DOP unit has yet to be found. All our simulations revealed a very low spatial and temporal coherence in the DOP access pattern. Hit rates above 80% can only be achieved with large cache sizes (>256KBytes), independent of index function or cache associativity. Likewise, using Page Interleaving and Page Mode Accesses do not improve the perfomance significantly.

4.2 Stack Engine

The Stack Engine processes a list by sequentially reading the DOP pointers in the list from the SRAM, and passing each pointer to the Memory Controller. This unit returns, along with the DOP pair, two child pointers to the Stack Engine. In case the parents are tumbled DOPs, these two pointers form the heads of two new lists and are written into the two SRAM chips. In case the parents are aligned DOPs, the Stack Engine evaluates the overlap flags and appends each pointer to the corresponding list, or discards it accordingly. When the list is done, the Stack Engine recurses on the next lower level. By default, the "left" branch is taken. If the left list is empty, the right list is used. If this list is empty too, the Stack Engine steps up one level in the hierarchy.

Internally, the Stack Engine maintains a stack of list pointers, one for each hierarchy level and branch, and a register containing the actual hierarchy level λ. Reads refer to level λ, writes to $\lambda + 1$. Since the lists are of varying length, but written contiguously into the SRAMs, there are additional marker registers defining the start of each list. Each SRAM chip is dedicated to either the left or the right list.

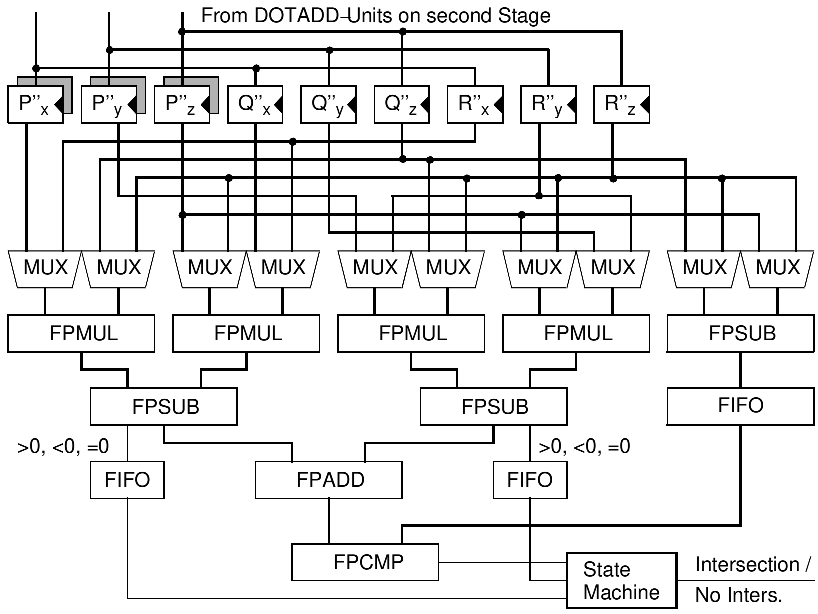

Figure 7: Second stage of the Triangle Unit performing the edge-triangle tests. The throughput together with the first stage is one pair every 3 cycles.

In case overlapping DOPs are leaves, the Stack Engine passes the pointers to the triangle data to the Triangle Unit. If the corresponding triangle data is not already available in this unit, list processing is interrupted in order to retreive triangle data from SDRAM. After this, the Triangle Unit is triggered, while list processing resumes.

4.3 Triangle Unit

A block diagram of the Triangle Unit is shown in Figure 6. This unit performs the optimized intersection test from Section 3 in a parallel, pipelined way. If all input data are available, this unit can complete an intersection test of two triangles every 3 clocks.

For high performance, there are storage elements for triangle data (i.e., vertices and triangle matrices) in the input stage of this unit. Since the algorithm always processes the complete list for a given tumbled node pair, we only need a two-entry SRAM for the associated triangle data. This SRAM is loaded upon the first overlap of leaves within a given list.

On the aligned side, we have found a cache for triangle data to be very useful. We assume a direct mapped cache with 128 lines, each holding 21 floats (three vertices plus a 3×4 matrix) and a tag for a total size of 11264 bytes. For most practical cases, hit ratio is above 75%. Both memory systems deliver 1 vertex per clock.

The next stage consists of 6 DOTADD-units, which transform each triangle into the coordinate frame of the object containing the other triangle. The transformation matrices M and $N := M^{-1}$ are constant over the whole collision test, and are pre-loaded by software into registers. This stage computes two transformed vertices per clock.

After this first transformation, each triangle must then be transformed using the matrix stored in the leaf of the other triangle. These matrices have travelled through FIFOs to arrive in time at the second row of DOTADD-units. Each FIFO needs $16 \times 12 \times 4 = 768$ bytes. The second row of DOTADD-units also computes one vertex per triangle per clock.

The final edge/triangle intersection tests are carried out in a separate functional unit called Intersection Test Unit (ITU). This unit must be able to accept one vertex per clock, from which with two clocks latency one edge per clock is generated. To this end it has a set of input registers, of which three are double buffered to keep up with the stream of incoming vertices. A block diagram of this unit is shown in Figure 7. Since triangles must be tested mutually, there are two ITUs on the CollisionChip.

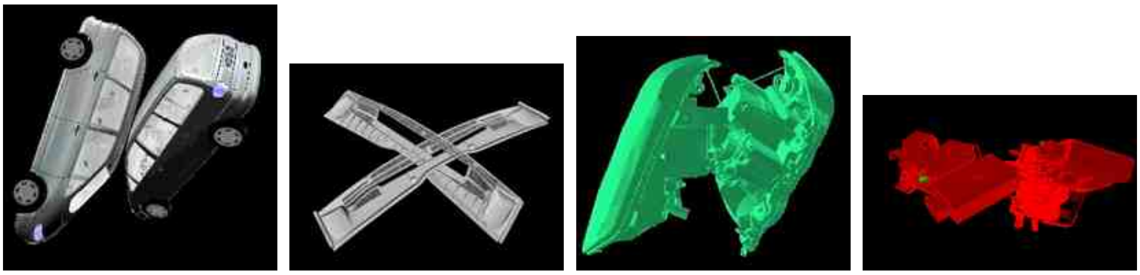

Figure 8: Some of the objects of our test suite (car body, cover, front light, door lock; courtesy VW and BMW).

The first stage of arithmetic units compute the products and the right side of the third term of ??. The multiplexers are used to funnel the proper operands for the current edge to the units. The subtractors on the second stage compute the first two terms of ??. The final adder computes the left side of the first term, which is compared to the right side in the last stage. All operands or flags computed in earlier stages travel through FIFO-memories to appear at the right time.

As a result, the Triangle Unit can complete an intersection test between two triangles every 3 clock cycles.

Intersections are reported to the software via the PCI Interface, along with the associated pointers from the Pointer FIFO.

4.4 Memory Controller

The Memory Controller facilitates access of the software to the SDRAM via the PCI Interface. Most importantly, though, the Memory Controller receives a stream of node pointers from the Stack Engine and retrieves the associated DOPs or triangle data from memory. It has an input register large enough to store the data of a complete node pair.

4.5 Multi-Bank SDRAM

The memory is built from four standard 64-bit modules placed in parallel for a 256-bit data bus. Our performance model assumes 133MHz DDR-SDRAM chips with a 2-2-2 access characteristic (2 cycles each for the precharge time, RAS-CAS-delay, and CAS-latency). Node data (DOPs as well as triangle vertices and matrices) are aligned on 128-byte boundaries. Node pairs are always stored contiguously in 256 bytes, which can be loaded in a double-data-rate burst transfer of length eight. Accordingly, the CollisionChip is assumed to be clocked at 266MHz. We assume further 8 memory chips per module with a page size of 1KBits, for a total page size across all modules of 4KBytes. Thus, sixteen node pairs can be packed in one page. Also, each SDRAM chip is assumed to have four internal banks. Accordingly, we use Page Interleaving and Page Mode accesses. Finally, we assume a page miss access to a node pair to consume 20 clock cycles, a page hit to consume only 8 cycles.

5 Performance

We have conducted performance comparisons between the traditional and our new algorithms in software, and between software and hardware (both using the new algorithms). The hardware performance estimation was obtained from a functional simulation written in C++, using the timing parameters given in Section 4. In all cases, we have used a suite of test objects, mostly from the automotive industry. Four objects of it can be seen in Figure 8. Since the number of nodes to be visited varies greatly with the position of the two objects relative to each other, we give average collision detection times for two identical objects in many different relative positions. Equally important, however, are worst-case timings since interactivity is limited most severely in these cases. All software timings have been obtained on a Pentium 4 with 1.8GHz under Linux using g++ 3.0.4.

5.1 New vs. old traversal

In this section, we will present some measurements comparing the new scheme with the old scheme for the DOP hierarchy using the DOP overlap test and construction algorithm presented earlier in [19]. Only the traversal is different be-

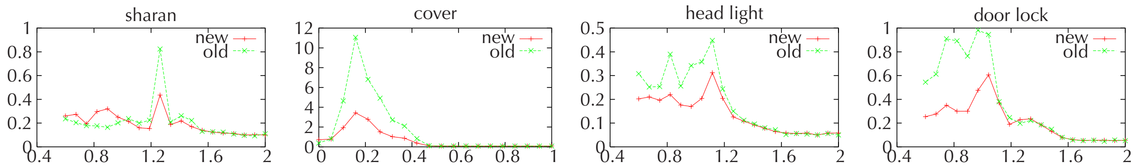

Figure 9: The performance gain of the new traversal scheme. The x axis shows the distance between the two objects, and the y axis shows the average collision detection time in milliseconds.

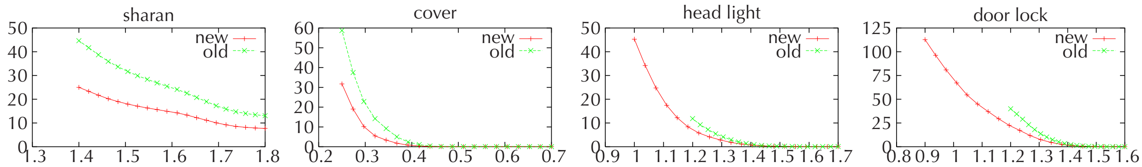

Figure 10: In contrast to Figure 9, here the algorithm returned all pairs of intersecting polygons (if any). The meaning of the axes is the same. Note that here, the plots show only the "interesting" range of distance from close proximity to lightly penetrating.

tween the two versions, all other parts of the software remain the same.

As mentioned in Section 3, our new traversal scheme can be applied to virtually all BV hierarchies.

Figures 9 and 10 show the comparison of the two traversal schemes for some of the objects of our test suite. The polygon counts per object are 28167, 10306, 30075, and 26136 polygons for the car body, filter, head light, and door lock, resp. Times are averaged over many different orientations. In Figure 9, the algorithm returned only the first witness, while in Figure 10, the algorithm returned all witnesses (i.e., pairs of intersecting polygons). Apparently, in that case, the gains are not as dramatic as in the first case, which we attribute to the fact that most of the time the CPU is waiting for the memory. If one compares the maximal collision time, quite similar plots are obtained. Overall, our new traversal scheme is almost always significantly faster than the traditional scheme.

From a hardware point of view, other characteristic numbers of the algorithm are more interesting. (The overall performance of the hardware will be presented below).

Figure 11 shows the average number of nodes visited during the traversal; visits on both trees are accounted for, and multiple visits are counted multiple times, i.e., this number is equivalent to the number of memory transfers that must be done (assuming there would be no cache). This visit count was averaged over a large number of different orientations for a certain distance between the two objects. The distance was chosen where the first peak occurred in Figure 9.

Even more dramatic is the difference in the number of DOP transformations that are performed during one traversal (see Figure 12). This is, of course, due to the fact that in the traditional scheme tumbled nodes are visited multiple times. The number of polygon intersection tests and the number overlap tests is more or less the same for both the old and the new traversal scheme.

5.2 Hardware performance

Processing of a given list involves reading and transforming two tumbled nodes, and reading and comparing the appropriate number of aligned node pairs. We assume that throughput is limited by transformation performance and memory bandwidth; the stack engine is assumed to be always fast enough. Further assumptions are as follows: nodes are defined by 24 single-precision floating-point numbers plus auxiliary data placed in memory on 128-byte boundaries. The memory is built from DDR-SDRAM chips with a 2-2-2 access characteristic (2 cycles each for the precharge time, RAS-CAS-delay, and CAS-latency). The CollisionChip is assumed to run at the data burst frequency, e.g. 266MHz for PC133 memory chips. A cycle of the CollisionChip equals one half of a memory cycle. The SDRAM Interface can buffer an entire node pair (256 bytes) and thus allows a burst length of eight to be used. In the following, cycles refer to chip cycles. Then, a random access to a node pair takes 16 cycles to complete.

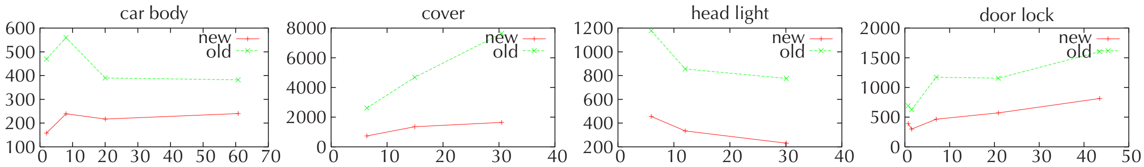

Figure 11: The total number of nodes visited on average during traversal (y-axis), i.e., the amount of traffic from memory. On the x-axis is the number of polygons for one object in 1000.

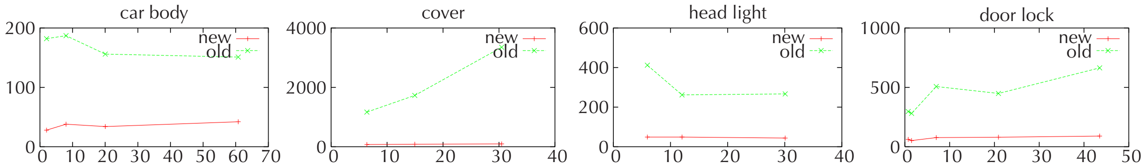

Figure 12: The total number of DOP transformation on average during a traversal (y-axis). X-axis = number of polygons for one object in 1000.

The first d-parameter of a tumbled node can be written in the DOP Register File 10 cycles after the (random) read was initiated. On average, continuous evaluation of Equation 3 can start after additional 12 cycles, when the first half of all d's are available. The DOTADD-unit is assumed to have 6 pipeline stages. The first result will be clocked into the Results Register Bank after a total of 28 cycles, the last one after 52 cycles.

Last access to the DOP Register File for the processing of the first tumbled node sibling occurs in cycle 46. The other sibling can then be transferred sequentially from the SDRAM Interface Unit into the DOP Register File and processed in the same way. The transformed sibling will be ready in the Results Register Bank after 88 cycles.

By that time, the first pair of aligned nodes in the list has been fetched from memory, with one of the nodes being present in "e"-register bank. The other node will be processed 4 cycles later. The load of the second node pair has been initiated such that processing can continue uninterrupted throughout cycle 100.

For all further memory reads, since we assume page faults for practically all memory reads, a delay will occur between read cycles. On memory chips with four internal banks, this delay will be 2 cycles on average, due to bank interleaving, giving a total read time of 10 cycles per node pair.

Thus, the performance can be estimated as

$$T_L = 100 + (\alpha - 2) * 10,$$

where T_L is the number of cycles needed to process a list, and α is the number of aligned node pairs in the list. If for a given collision test for two objects there are τ lists to process, each with α node pairs on average, the total performance can be characterized as

$$T_T = (100 + (\alpha - 2) * 10) * \tau \qquad (5)$$

The number of lists τ is given by the number of visited tumbled node pairs.

Based one these cycle times, we have implemented a functional simulation of the hardware in C++, which was assumed to run at 266 MHz. Then, exactly the same benchmark procedure as for the software has been run with this simulation. The software implementation was running on a Pentium-4 CPU at 1.8 GHz. The speedup for worst-case object configurations is shown in Figure 13.

6 Conclusions and Future Work

In this paper, we have presented novel algorithms and a hardware architecture for performing hierarchical collision detection. It is arguably the first special-purpose hardware architecture dedicated to this task. We lay special emphasis on the fact that this architecture is suitable for "polygon soups" in general, as opposed to previously reported methods utilizing graphics hardware.

The speedup of a 266 MHz collision detection chip over a 1.8 GHz software solution is around 30 on average. It is generally higher in worst-case scenarios, which is an important result, because interactivity and stability is limited most severely by these cases. Thus a chip design is very well justified.

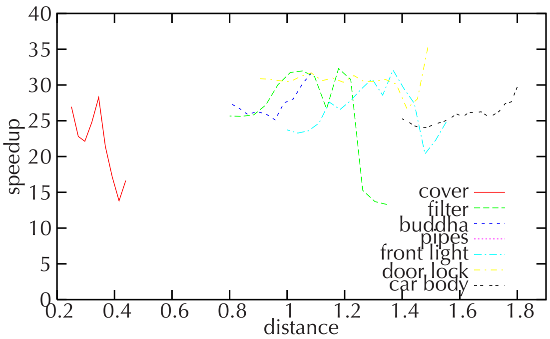

Figure 13: Speedup of our hardware architecture over our software implementation. Note that the software was running on a 1.8 GHz Pentium 4, while our hardware architecture was assumed to run only at 266 MHz.

A good part of the speedup can be attributed to our novel hierarchy traversal scheme, which can be applied to most kinds of bounding volume hierarchies.

Our near-term goal will be to implement a VHDL model of the CollisionChip, identify potential bottlenecks, and further optimize the architecture towards even higher processing speeds. Our long-term goal will be to integrate this project into an industrial virtual prototyping application.

We will also look into the open issue whether or not hierarchical algorithms are best suited for a hardware implementation, because of their bad memory coherence. Therefore, we will investigate non-hierarchical data structures, which also might offer the benefit of making deformable collision detection possible.

Acknowledgements

This work is partially supported by the DFG program "Aktionsplan Informatik" under grant ZA292/1-1.

References

[1] Jeff Arenberg. Re: Ray/triangle intersection with barycentric coordinates. *Ray Tracing News*, 1(11), 1988. `http://www.acm.org/pubs/tog/resources/RTNews/html/rtnews5b.html`.

[2] George Baciu, Wingo Sai-Keung Wong, and Hanqiu 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.

[3] Didier Badouel. An efficient ray-polygon intersection. In *Graphics Gems*, Andrew S. Glassner, Ed., pages 390–393. Academic Press, San Diego, 1990. includes code.

[4] Jens Eckstein and Elmar Schömer. Dynamic collision detection in virtual reality applications. In *Proc. The 7-th Int'l Conf. in Central Europe on Comp. Graphics, Vis. and Interactive Digital Media '99 (WSCG'99)*, pages 71–78, Plzen, Czech Republic, February 1999. University of West Bohemia.

[5] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. In *SIGGRAPH 96 Conference Proceedings*, Holly Rushmeier, Ed., pages 171–180. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

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

[7] Ray Jones. Intersecting a ray and a triangle with Plücker coordinates. *Ray Tracing News*, 13(13), 2000. `http://www.acm.org/pubs/tog/resources/RTNews/html/rtnv13n1.html`.

[8] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, David C. Evans and Russell J. Athay, Eds., vol. 20, pages 269–278, August 1986.

[9] James T. Klosowski, Martin Held, Jospeh S. B. Mitchell, Henry Sowrizal, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, January 1998.

[10] Dave Knott and Dinesh 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.

[11] J.-C. Lombardo, M.-P. Cani, and F. Neyret. Real-time collision detection for virtual surgery. In *Proc. of Computer Animation*, Geneva, Switzerland, May26-28 1999.

[12] Tomas Möler. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30, 1997. ISSN 1086-7651.

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

[14] Jörg Sauer and Elmar Schömer. A constraint-based approach to rigid body dynamics for virtual reality applications. In *Proc. VRST '98*, pages 153–161, Taipei, Taiwan, November 1998. ACM.

[15] Christophe Schlick and Gilles Subrenat. Ray intersection of tessellated surfaces: Quadrangles versus triangles. In *Graphics Gems V*, Alan Paeth, Ed., pages 232–241. Academic Press, San Diego, 1995.

[16] M. Shinya and M.-C. Forgue. Interference detection through rasterization. *The Journal of Visualization and Computer Animation*, 2(4):132–134, October–December 1991.

[17] Gino Johannes Apolonia van den Bergen. *Collision Detection in Interactive 3D Computer Animation*. PhD dissertation, Eindhoven University of Technology, 1999.

[18] Douglas Voorhies and David Kirk. Ray-triangle intersection using binary recursive subdivision. In *Graphics Gems II*, James Arvo, Ed., pages 257–263. Academic Press, San Diego, 1991.

[19] Gabriel 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, March 1998.

[20] Gabriel Zachmann. Optimizing the collision detection pipeline. In *Proc. of the First International Game Technology Conference (GTEC)*, January 2001.

[21] Gabriel Zachmann and Günter Knittel. An architecture for hierarchical collision detection. In *Journal of WSCG '2003*, pages 149–156, University of West Bohemia, Plzen, Czech Republic, February3–7 2003. `http://www.gabrielzachmann.org/`.