An Architecture for Hierarchical Collision Detection *

Gabriel Zachmann
Computer Graphics, Informatik II
University of Bonn
email: zach@cs.uni-bonn.de

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

Abstract

We present novel algorithms for efficient hierarchical collision detection and propose a hardware architecture for a single-chip accelerator. We use a hierarchy of bounding volumes defined by k-DOPs for maximum performance. A new hierarchy traversal algorithm and an optimized triangle-triangle intersection test reduce bandwidth and computation 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.

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 try to place objects at a new position, then check for collision, and then try other positions, based on physical laws or constraints [14, 21]. This poses very high demands on collision detection performance, because they must do many collision checks per simulation cycle. 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.

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 considerations about the performance of the envisioned architecture.

2 Related Work

Hierarchical collsion detection. 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

* This is a slightly extended version of the WSCG '03 paper

(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.

Collision detection in graphics hardware. 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, 13, 16] is to render the pair of objects with an orthogonal projection and counting certain cases of overlapping intervals in the stencil buffer. This approach lends itself well to convex objects and a very special class of non-convex objects. The more general case of arbitrary concave objects cannot be solved efficiently with todays rendering hardware. Furthermore, the most general case 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 [10]. 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.

Polygon intersection tests. A number of algorithms for ray-triangle and triangle-triangle intersection have been presented in the literature [1, 3, 7, 11, 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 [11] and [1], but is also very well suited for a hardware implementation due to a very uniform control and data flow.

3 The Algorithm

3.1 DOP Trees

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 inner node stores a DOP and pointers to its children which it encloses; leaves store polygons (or other graphical primitives). 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 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

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

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

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

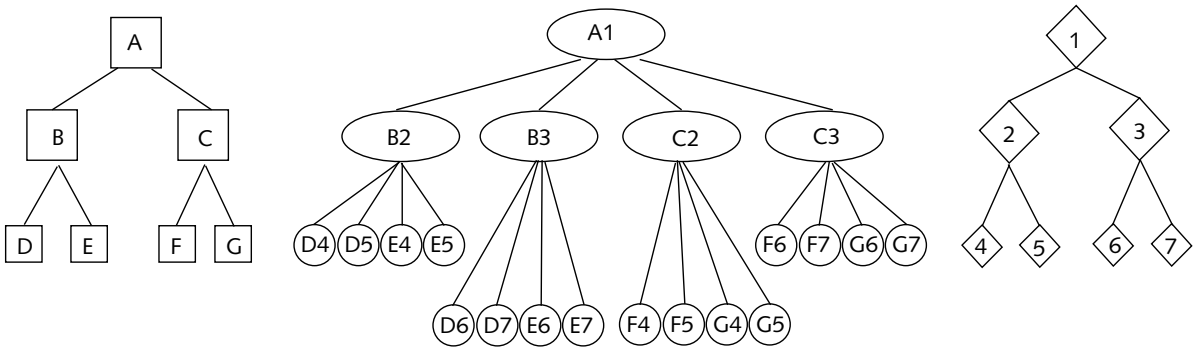

Figure 1: The simultaneous traversal of two BV hierarchies is, conceptually, equivalent to the traversal of a *BV pairs* hierarchy. Here, the right DOP tree is "tumbled" with respect to the DOP orientations of the left tree's reference frame.

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 $3k$.

3.2 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 specific to individual nodes 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.

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. 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 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.

```
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.

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 implementations 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. 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.

In a hardware implementation, we have to maintain the stack and the lists ourselves. This can be done by a stack of lists (see Figure 2). On the same stack, we keep pointers to pairs of tumbled nodes. Going down from node pair (A,B) to (A_1, A_2), we push the pointer to (A, B) onto the stack. Later, when the recursion returns to this node pair, we need to decide whether to go down into node pair (B_1, B_2) or to make a step upwards. This information can be kept in an additional bit on the stack: when the pointer is pushed onto the stack, the corresponding bit is reset; when we return to this node, we go down into the other branch and flip the bit to 1. When we return the next time, the algorithm knows to make another step upwards.

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

```
traverse(A,B,L)
transform A
transform B
init L with pairs of first level beneath roots
for all N ∈ L do
  if X and N do overlap then
    if X and N are leaves then
      check primitives enclosed by X and N
    else
      L'_X += N1, N2
    end if
  end if
end for
if A is an inner node then
  traverse(A1, A2, L'_A)
else
  traverse(A, L'_A)
end if
if B is an inner node then
  traverse(B1, B2, L'_B)
else
  traverse(B, 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 executed in parallel on both nodes A and B.

3.3 Polygon Intersection Test

In the case of collision, the traversal reaches pairs of leaves containing triangles, which have to be checked for intersection. Assume triangle A is given by vertices V^1, V^2, V^3 and triangle B is given by vertices W^1, W^2, W^3, both in their object's reference frame. Assume triangle A is part of object O_A, and B is part of O_B.

The approach in our algorithm is to check (conceptually) each edge of A against B, and vice versa. First, A's vertices are transformed into the reference frame of O_B. Assume further a 3×3 transformation M_B for triangle B such that $M_B \cdot (W^i - W^1)$ maps onto the unit triangle $(0,0,0)$, $(1,0,0)$, $(0,1,0)$. Then, we transform A by (M_B, W^1) (see Figure 3). For sake of simplicity, we will call the new vertices V^i again.

Consider each edge $\overline{PQ} := \overline{V^i V^{i+1}}$. If both P_z and $Q_z \geq 0$ or ≤ 0, then we skip this edge. Now

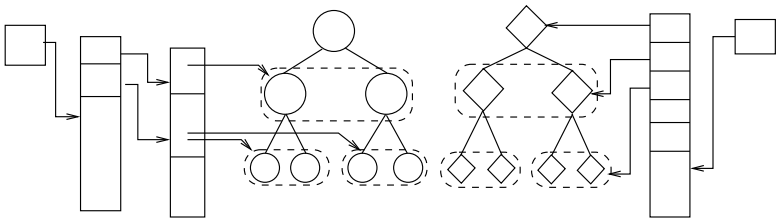

Figure 2: The improved traversal scheme can be implemented by a stack of lists. (In a hardware implementation, the stack on the right is merged into the left one.)

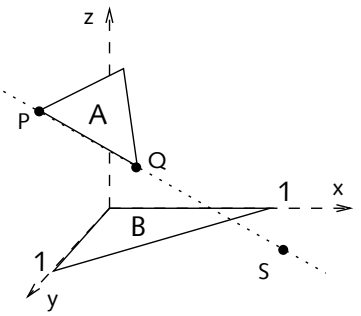

Figure 3: Using a special transformation, the intersection test can be done very efficiently.

we compute (conceptually) the intersection S of the supporting line $X = P + t\mathbf{r}, \mathbf{r} = Q - P$, with the plane $z = 0$, which is defined by $t = -\frac{P_z}{r_z}$ as $S = P - \mathbf{r}\frac{P_z}{r_z}$ (we know $r_z \neq 0$). We know that $0 \leq t \leq 1$. We also know that $S_z = 0$, so we need to compute only $S_x = P_x - r_x\frac{P_z}{r_z}$ and similarly S_y, which are, basically, the barycentric coordinates of the intersection point. Finally, we just check whether or not $S_x \geq 0 \wedge S_y \geq 0 \wedge S_x + S_y \leq 1$. If so, A and B do intersect; otherwise, we check the other edges, and, in case of no intersection, we check B against A.

In order to save the division and the vector subtraction (for $\mathbf{r}$), we reformulate the condition as follows (assuming $r_z > 0$):

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

If $r_z < 0$, then we must compare with $\leq 0, \leq 0$, and ≥ 0, respectively.

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.

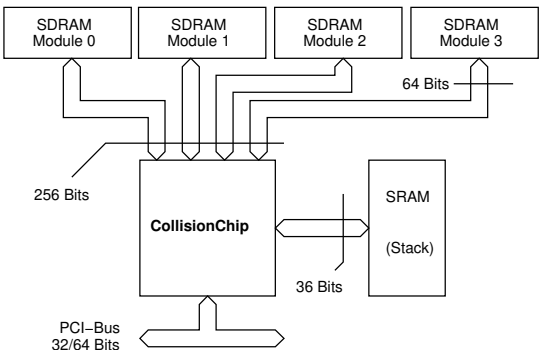

Figure 4: Schematic diagram.

4 Hardware Design

The target design is a PCI-board with one ASIC, a large on-board memory for the hierarchy, and a small SRAM 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 (see Figure 4).

4.1 The CollisionChip

Figure 5 shows all functional units of the CollisionChip. It consists of a number of large register files grouped around an arithmetic unit for floating-point dot-products (the DOTADD-unit), a Triangle Intersection Test Unit (IT-Unit), register banks connected to comparators, interfaces to the PCI-bus and to the local memories, the Stack Engine and control units as well as address generators. Although the processing of bounding volumes and triangles differ quite substantially, a common architecture was found with only low redundancy.

Although the design is geared towards high performance and the chip looks large on paper, the actual chip space will be modest. All register bits together require roughly only 100k transistors. Also note that for comparisons, IEEE floating-point operands can be treated basically like integers, which simplifies all comparators significantly. Expensive units are the DOTADD-Unit and, to a lesser degree, the IT-Unit and the four-port DOP Register File. However, units like these can be found on today's CPUs and graphics chips, so one can be confident that the CollisionChip can be built at low costs using current technology. The design was laid out for $k = 24$ and single-precision IEEE floating-point operands.

The chip will have around 450 signal pins, mostly due to the 256-bit bus, and should fit into a 600-pin package including all power pins.

The DOTADD-Unit. The DOTADD-unit is similar to transform units as found in modern graphics accelerators. Its basic function is to perform

$$d'_i = d_k \times C_{i,0} + d_m \times C_{i,1} + d_n \times C_{i,2} + c_i$$

on 32-bit floating-point numbers. The indices refer to the location in the register files. Due to the absence of data dependencies in the control flow, it can be pipelined for high clock frequency and throughput.

Processing of Bounding Volumes. Prior to the processing of two hierarchies, the matrix **C** and the coefficients **c** must be loaded into the Matrix Register File. Also, the correspondence indices **i**, **k**, **m** and **n** must be stored in the Correspondence Register File. This happens via the PCI-bus under software control, and occupies 24 lines in both register files. The software also transmits the pointers (local memory addresses) of the two root nodes as starting point to the Master Controller.

A DOP from the tumbled object is loaded from memory and sequentially stored in the DOP Register File under control of Address Generator 1. After some constant delay (again predetermined by software) a sufficiently large subset of DOP-elements **d** are or will become available in time for continuous evaluation of Equation 1. At this point in time, Address Generator 2 is triggered and the operands are fed into the DOTADD-unit. Note that for maximum performance, processing of the lines in matrix **C** occurs out-of-order, depending on the earliest availability of the required d-elements. Also note that a specific d may be used

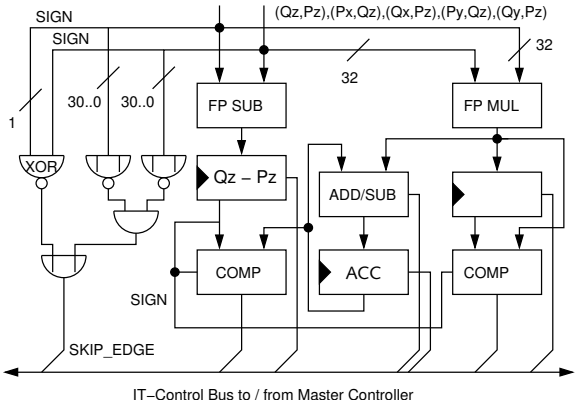

Figure 7: Intersection Test Unit for Triangles

for more than one line. The transformed DOP-elements **d′** are then stored into the Results Register Bank under control of Address Generator 3, which basically delays an index i for the duration of the pipeline delay of the DOTADD-unit. The same processing is applied to the sibling of the tumbled node.

The DOP-elements **e** of the aligned node are then loaded from memory in three transfers and stored in the registers labeled e0 ... e23 under control of the Master Controller. The bank of comparators determines overlap in parallel and signals this condition to the Master Controller. The lists of child nodes to be checked for overlap are constructed according to this condition by the Stack Engine.

Hierarchy Traversal. The novel traversal algorithm as described in Section 3.2 is implemented using a dedicated and fast external SRAM to store the lists and a suitably designed Stack Engine. Its basic task is to hand node pointers from the current list to the Master Controller and to receive child node pointers to construct new lists. Internally, it maintains a stack of list pointers and a register containing the actual level.

Processing of Triangles. As described in Sect. 3.3, testing triangle A from object O_A against triangle B from object O_B requires transforming A into the coordinate system of O_B using a rotation and a translation. These are constant for two objects, and can therefore be precomputed. The reverse test B against A may also be necessary, which requires the inverse transform. For maximum performance, all coefficients are kept on chip in the Matrix Register File in lines 24 through 29. After this first transformation, the triangle must then be transformed using the matrix stored in the leaf of the other triangle, whose coefficients are loaded

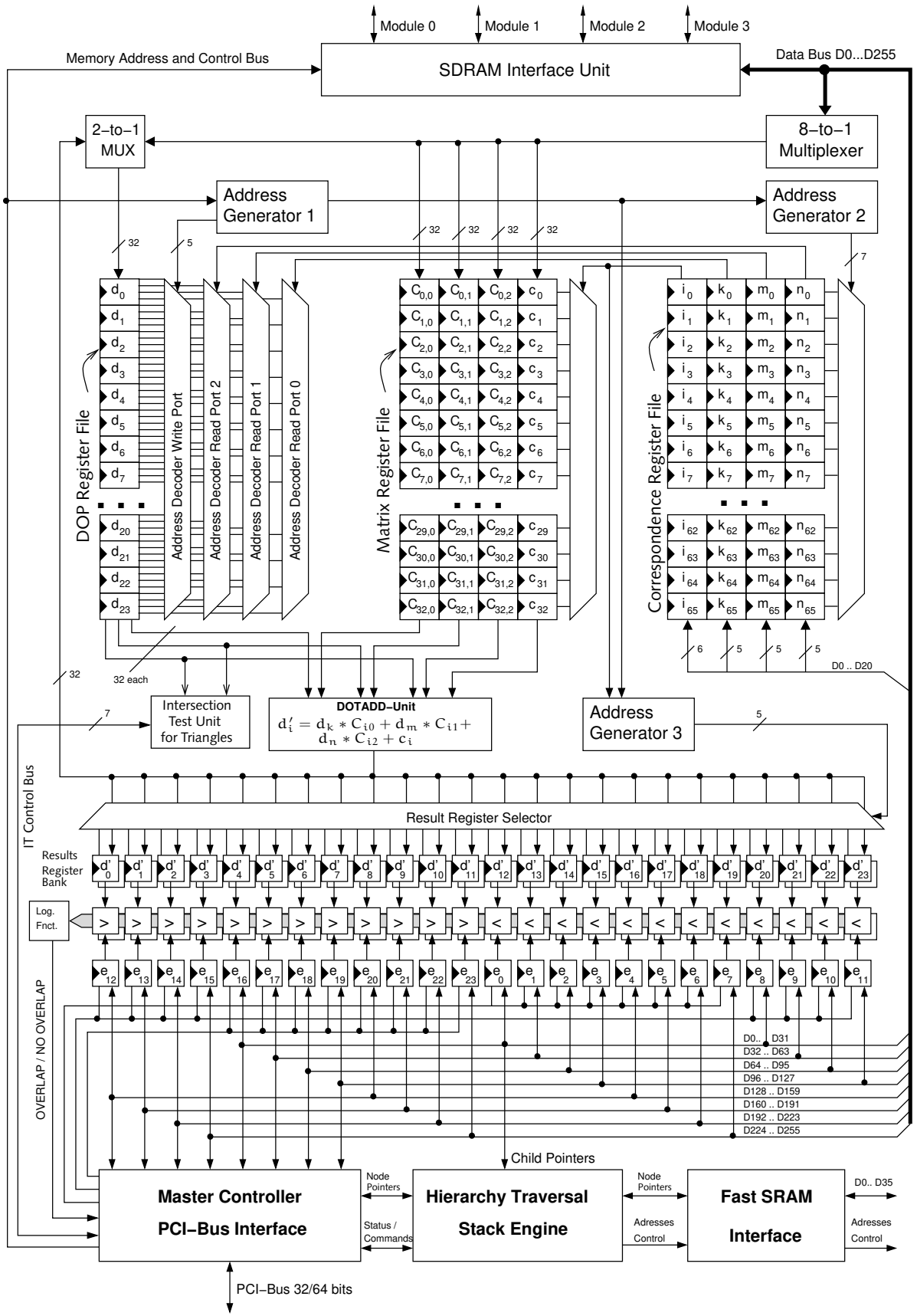

Figure 5: All functional units of the CollisionChip.

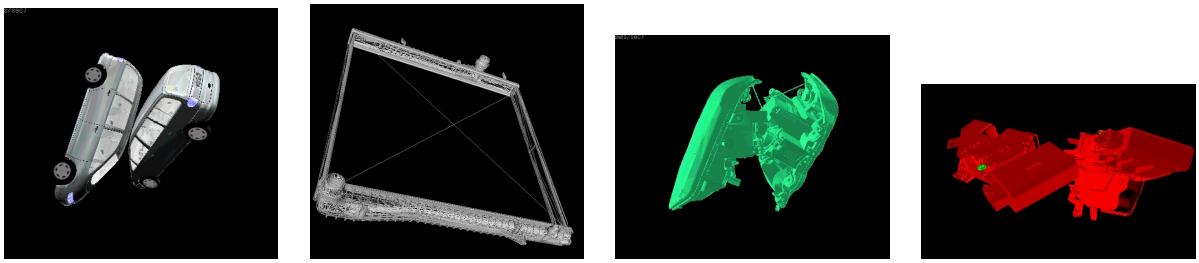

Figure 6: Some of the objects of our test suite (car body, front light, cooling filter, door lock; courtesy VW and BMW).

into lines 30 through 32 of the Matrix Register File. The Correspondence Register File has 27 lines dedicated to these transforms, and so they can be performed using the DOTADD-unit by properly setting up the Address Generators.

The concrete sequence of operations is as follows. The vertices of A are loaded into the DOP Registers $d_0 \ldots d_8$, transformed and written back into DOP Registers $d_9 \ldots d_{17}$. Meanwhile, the coefficients from the other leaf are loaded from memory. The second transformation of A leaves the vertices in registers $d_0 \ldots d_8$ again.

Further processing according to Section 3.3 and Equation 4 is then done in a separate unit called IT-Unit. (see Figure 7) This unit can have low-performance and thus low-cost arithmetic units, since triangle tests are not performed very frequently (see Table 1). This unit is controlled by the Master Controller via a 7-bit bus and supplied with operands from the DOP Register File using indices from the Correspondence Register File. Five pairs of operands need to be read per edge, which requires additional 15 entries for a total of 66 lines in the Correspondence Register File. The accumulation circuitry consisting of the ADD/SUB unit and the register ACC compute the left side of the third condition in Equation 4. The combinational circuitry to the left determines whether the edge cuts the $z = 0$ plane at all. Note that the labels in Figure 7 refer to only one edge, and that processing may have to be repeated for the other two edges.

An intersection is reported to the software; otherwise, the above procedure is repeated with operands reversed.

5 Performance Estimations

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. We also don't consider triangle-triangle-tests here since they don't occur very often.

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 build 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.

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 1 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 four cycles later. The load of the second node pair has been initiated

		average numbers					worst-case numbers				
Object	num pgons	aligned nodes visited[1]	tumbled nodes visited[2]	pgon checks	time in HW	time in SW	aligned nodes visited[1]	tumbled nodes visited[2]	time in HW	time in SW	speedup avg / worst-case
Filter	19 326	12 474	240	1 660	153	14 936	542 888	3 553	5 638	717 164	98 / 127
Frontlight	30 075	389	73	68	15	524	7 065	937	207	9 226	36 / 44
Lock	62 023	279	81	9	15	401	4 854	877	178	6 804	27 / 38
Car body	60 755	259	66	55	12	383	3 076	538	110	5 390	31 / 49
Buddha	125 000	159	50	7	9	240	3 345	301	77	4 074	26 / 53

Table 1: This table shows the performance of our hardware architecture for a number of objects that are placed in close proximity. All times are in microseconds. The average numbers have been obtained by rotating one of the objects relative to the other. The worst-case numbers are the respective maxima observed during that rotation. The collision detection times have been calculated with Equation 5, assuming 3.76 nsec/cycle (α = num. aligned nodes / num. tumbled nodes, τ = num. tumbled nodes / 2). Columns marked by (1) count multiple visits of aligned nodes, while those marked by (2) count the number of unique tumbled nodes (which are, unlike traditional traversal schemes, visited only once). The software performance has been measured using the traditional recursive hierarchy traversal.

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 two cycles on average due to bank interleaving, giving a total read time of 10 cycles for a 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.

This estimation is compared to a software implementation on a PC with a Pentium-III CPU running at 1GHz. The results are summarized in Table 1.

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.

As can be seen in Table 1, the speedup ranges between about 25 and 125 for our benchmarks. It is generally higher in worst-case scenarios, which is an important result, because interactivity is limited most severely by these cases. Thus a chip design is very well justified.

A good part of the speedup can be attributed to our novel hierarchy traversal scheme, which can be applied to all 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.

References

[1] J. Arenberg, *Re: Ray/Triangle Intersection with Barycentric Coordinates*, Ray Tracing News, 1 (1988). `http://www1.acm.org/pubs/tog/resources/RTNews/html/rtnews5b.html`.

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

[3] D. Badouel, *An Efficient Ray-Polygon Intersection*, in Graphics Gems, A. S. Glassner, ed., Academic Press, San Diego, 1990, pp. 390–393. includes code.

[4] J. Eckstein and E. 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), Plzen, Czech Republic, Feb. 1999, University of West Bohemia, pp. 71–78.

[5] S. Gottschalk, M. Lin, and D. Manocha, *OBBTree: A Hierarchical Structure for Rapid Interference Detection*, in SIGGRAPH 96 Conference Proceedings, H. Rushmeier, ed., ACM SIGGRAPH, Addison Wesley, Aug. 1996, pp. 171–180. 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 (1995), pp. 218–230. ISSN 1077-2626.

[7] R. Jones, *Intersecting a Ray and a Triangle with Plücker Coordinates*, Ray Tracing News, 13 (2000). `http://www1.acm.org/pubs/tog/resources/RTNews/html/rtnv13n1.html`.

[8] T. L. Kay and J. T. Kajiya, *Ray Tracing Complex Scenes*, in Computer Graphics (SIGGRAPH '86 Proceedings), D. C. Evans and R. J. Athay, eds., vol. 20, Aug. 1986, pp. 269–278.

[9] J. T. Klosowski, M. Held, J. S. Mitchell, H. Sowrizal, and K. Zikan, *Efficient Collision Detection Using Bounding Volume Hierarchies of* k-*DOPs*, IEEE Transactions on Visualization and Computer Graphics, 4 (1998), pp. 21–36.

[10] 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.

[11] T. Möller, *A fast triangle-triangle intersection test*, Journal of Graphics Tools, 2 (1997), pp. 25–30.

[12] T. Möller and B. Trumbore, *Fast, Minimum Storage Ray-Triangle Intersection*, Journal of Graphics Tools, 2 (1997). ISSN 1086-7651.

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

[14] J. Sauer and E. Schömer, *A Constraint-Based Approach to Rigid Body Dynamics for Virtual Reality Applications*, in Proc. VRST '98, Taipei, Taiwan, Nov. 1998, ACM, pp. 153–161.

[15] C. Schlick and G. Subrenat, *Ray Intersection of Tessellated Surfaces: Quadrangles versus Triangles*, in Graphics Gems V, A. Paeth, ed., Academic Press, San Diego, 1995, pp. 232–241.

[16] M. Shinya and M.-C. Forgue, *Interference detection through rasterization*, The Journal of Visualization and Computer Animation, 2 (1991), pp. 132–134. ISSN 1049-8907.

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

[18] D. Voorhies and D. Kirk, *Ray-Triangle Intersection Using Binary Recursive Subdivision*, in Graphics Gems II, J. Arvo, ed., Academic Press, San Diego, 1991, pp. 257–263.

[19] G. Zachmann, *Rapid Collision Detection by Dynamically Aligned DOP-Trees*, in Proc. of IEEE Virtual Reality Annual International Symposium; VRAIS '98, Atlanta, Georgia, Mar. 1998, pp. 90–97.

[20] ——, *Optimizing the Collision Detection Pipeline*, in Proc. of the First International Game Technology Conference (GTEC), Jan. 2001.

[21] G. Zachmann and A. Rettig, *Natural and Robust Interaction in Virtual Assembly Simulation*, in Eighth ISPE International Conference on Concurrent Engineering: Research and Applications (ISPE/CE2001), West Coast Anaheim Hotel, California, USA, July 2001.