Object-Space Interference Detection on Programmable Graphics Hardware

Alexander Greß Gabriel Zachmann

Abstract. We present a novel method for checking the intersection of polygonal models on graphics hardware utilizing its SIMD, occlusion query, and floating-point texture capabilities. Our algorithm simultaneously traverses a pair of bounding volume hierarchies performing all the necessary computations during this traversal, including the final triangle intersection tests, in vertex and fragment programs on the GPU. Unlike previous graphics hardware based methods, our method does all computations in object space and does not make any requirements on connectivity or topology.

§1. Introduction

Fast and exact collision detection of polygonal objects undergoing rigid motions is at the core of many simulation algorithms in computer graphics. In particular, virtual reality applications such as virtual prototyping need exact collision detection at interactive speed for very complex, arbitrary "polygon soups". It is also a fundamental problem of dynamic simulation of rigid bodies, simulation of natural interaction with objects, haptic rendering, path planning, and CAD/CAM.

Currently, the performance of graphics hardware (GPUs) is progressing faster than general-purpose CPUs. The main reason is an architecture that combines *stream processing* [11] and SIMD processing. In addition, the programmability of the GPU has increased drastically over the past few years. Overall, today a programmer can write *kernels* for all stages of the graphics pipeline that are automatically executed in parallel on an indefinite number of processing units. This has led many researchers to investigate exploitation of the GPU for other computations, such as matrix computations, ray tracing, distance field computation, etc.

xxx and xxx (eds.), pp. 1–4.
Copyright © 200x by Nashboro Press, Brentwood, TN.
ISBN 0-9728482-x-x

Many algorithms have been proposed to utilize graphics hardware for the problem of collision detection. They can be classified into techniques that make use of the depth and stencil buffer tests, and those that compute discrete distance fields. In any case, the problem is approached in *image space*, i.e., it is discretized.

However, to our knowledge, no attempts have been made to utilize the GPU while still performing all computations in *object space*. This is what we address in this paper.

Based on techniques known from traditional CPU based collision detection approaches we develop a new method that utilizes the graphics hardware for hierarchical collision detection. Our algorithm simultaneously traverses a pair of bounding volume hierarchies consisting of axis-aligned bounding boxes. All computations during this traversal, including the final triangle intersection tests, are performed in vertex and fragment programs on the GPU. The algorithm has no requirements on the shape, topology, or connectivity of the polygonal input models.

§2. Related Work

GPU based processing has become a trend over the past few years [15]. Generally, the idea is to formulate the given problem such that it can be solved by a number of rendering passes. During each of them, some of the computations are performed by rendering a number of geometric primitives, thereby updating one or more of the available buffers (stencil, color, z-buffer, etc.).

A clever way to utilize graphics hardware was presented by [13]. Based on the observation that an intersection can occur if and only if an edge of one object intersects the other one, they render edges of one object and polygons of the other. This even works for deformable geometry. Unlike many previous approaches, objects do not need to be convex. However, they must still be closed. Furthermore, it seems to work robustly only for moderate polygon counts.

A hybrid approach was proposed by [8]. Here, the graphics hardware is used only to detect potentially colliding objects, while triangle-triangle intersections are performed in the CPU. While this approach alleviates previous restrictions on object topology, its effectiveness seems to degrade dramatically when the density of the environment increases.

The approach presented by [2] can compute the penetration depth using graphics hardware, but only for convex objects.

Earlier image-based methods include [20, 18, 3, 16, 4].

Virtually all image-based collision detection methods have several drawbacks in common:

1. Their complexity is usually in $\mathcal{O}(n)$, which is much slower than the

complexity of hierarchical, software-based approaches that seem to be in $\mathcal{O}(\log n)$.

2. Rendering geometric primitives basically amounts to a discretization of the problem and thus introduces geometric errors. These errors depend more or less on the size of the viewport, the internal representation of numbers, and the number of bits per pixel in the z-buffer. The size of the viewport has significant impact on the performance.

Traditionally, rigid collision detection has been solved by simultaneously traversing a precomputed bounding volume hierarchy. A wealth of different BV hierarchies has been explored, such as sphere trees [10, 19], OBB trees [7], DOP trees [12, 22], Boxtrees [23, 1], AABB trees [21, 14], and convex hulls [5].

Another rather recent effort is to design hardware specifically for the purpose of collision detection [24, 25]. In the present paper, however, we are concerned with utilizing commodity general-purpose graphics hardware.

To our knowledge, there is no previous work tackling the problem of collision detection in object space on the GPU.

§3. Triangle Intersection Tests

Assume that two triangle meshes are to be checked for interference. Obviously, the brute-force approach is to check all triangles of the first mesh against all triangles of the second mesh for intersection. Before passing on to the hierarchical approach in the following section, we describe how to realize the straightforward solution in programmable graphics hardware.

The $m \cdot n$ triangle intersection tests required to check all m triangles of one mesh against all n triangles of another mesh obviously do not depend on each other and thus can be performed efficiently in graphics hardware by rendering a rectangle of size $m \times n$ using a fragment program that tests exactly one triangle pair at a time.

Note that on programmable graphics hardware, rendering a rectangle of size $m \times n$ corresponds to $m \cdot n$ invocations of a fragment program. Theoretically, all these fragment program invocations could be performed in parallel since the input required for each invocation may not depend upon the output of another fragment program invocation for the same rectangle primitive. Therefore, all available fragment program execution units can be utilized for such a task.

To determine whether two triangles intersect, we implemented the *SAT* method as fragment program, which checks if there is an axis such that the two triangles projected onto that axis are disjoint. This method is based on the separating axes theorem [6].[1] Each fragment, whose corresponding triangle pair does not intersect, is discarded by the fragment program. (A fragment can be discarded conditionally using the *KIL* instruction in an ARB fragment program or, equivalently, using the *clip* instruction in NVIDIA's *Cg* or Microsoft's *HLSL* high-level shading language.)

To determine the number of intersecting triangle pairs, we use the occlusion query feature of current GPUs. (Occlusion queries are OpenGL core functionality since version 1.5, and have also been available in prior OpenGL versions via various hardware-specific extensions.) The occlusion query returns the number of pixels actually written to the output buffer. Using the described fragment program, which discards all fragments corresponding to non-intersecting triangle pairs, this number exactly corresponds to the number of intersecting triangle pairs.

The input required for the intersection tests has to be stored in graphics memory. This can be done using floating-point textures available on current graphics hardware. An array of vertex positions is represented by a three-component floating-point 1D texture. As in a simple triangle soup representation, we use three such textures for each input model to store the three vertex positions of each triangle.

To be able to check two polygonal models specified in different object-spaces for interference, we need the transformation matrix from one object-space to the other one as further input. Since this matrix is constant for all $m \cdot n$ intersection tests, it can be passed to the fragment program as program parameter (i.e., as uniform variable in the high-level shading language). Using this matrix, the fragment program first transforms the three vertices of the considered triangle from the first mesh to the object-space of the second mesh and then does the intersection test, discarding the fragment if there is no intersection. However, in total each triangle of the first mesh is transformed n-times using this method.

Therefore, it is more efficient to move the transformation into the vertex program, as it is done in the following alternative solution. Instead of rendering a single rectangle primitive of size $m \times n$, we render m horizontal line primitives of length n. Using this technique, the vertex program is invoked $2m$-times (as a line primitive consists of two vertices), and the fragment program is invoked n-times per line. The vertex program transforms a triangle of the first mesh into the object-space of the second mesh and passes the transformed triangle as fragment data to the fragment pro-

[1]Note, that the graphics hardware available at the time when this work was done had no dynamic flow control logic in the fragment program units. Without dynamic flow control, the execution time of the fragment program always depends on the *total* number of instructions of the program, even for fragments that are discarded at an early stage of the program. Therefore, we use a triangle intersection test method that requires less case differentiations than alternative methods and thus can be implemented with a comparatively low total instruction count.

gram. Contrary to the first solution, we now use a vertex array instead of three textures to store the triangles of the first mesh and make the transformation matrix a program parameter of the vertex program.

§4. Hierarchical Interference Detection

Although the simple brute-force approach described in the previous section takes advantage of the parallel architecture of the GPU, it can clearly not outperform a clever hierarchical approach if large objects or scenes are to be tested. Therefore, in this section we propose a method for hierarchical interference detection on the GPU that is based on a bounding volume hierarchy.

4.1. Bounding Volume Hierarchy

As in traditional CPU-based approaches, one object is to be checked for interference with another one by simultaneously traversing their two bounding volume hierarchies. We use axis-aligned bounding boxes (AABBs) as bounding volumes since they are well-suited for our GPU based approach and, despite their simplicity, still very efficient [21, 23].

For each object, we generate an AABB tree that consists of a bounding box at each inner node and a triangle at each leaf node. This generation is done in a preprocessing step on the CPU.

During the simultaneous traversal of two AABB trees S and T, all those pairs of nodes (S_i, T_j) are to be visited that are on the same hierarchy level in the corresponding trees and for which the parent nodes overlap. Various traditional CPU-based approaches use a depth-first traversal strategy. However, this way the decision whether a pair of nodes has to be visited depends on the result of an overlap test that was performed immediately before. Therefore, this strategy is not suited for parallel execution on multiple vertex or fragment program units.

Instead, we use a breadth-first traversal scheme, i.e., when a node pair of a certain hierarchy level is visited, the node pairs of preceding hierarchy levels must have been processed already.

To be able to traverse the AABB trees efficiently, the trees have to be balanced. Furthermore, since leaf nodes will be handled differently than inner nodes by our algorithm, we require that there are no leaf nodes in the tree other than at the lowest hierarchy level, even if we construct a tree with a number of leaf nodes that is not a power of two. Therefore, we construct a tree where the lowest hierarchy level L_{max} has exactly as many nodes as there are triangles in the input model. For any other hierarchy level $L = 0, \ldots, L_{max} - 1$ the number of nodes $n(L)$ equals $\lceil \frac{n(L+1)}{2} \rceil$. This way, each inner node of the resulting AABB tree has one or two child nodes. Note, that there is at most one inner node at each hierarchy level that has just one child node.

When traversing two AABB trees simultaneously, in the following we assume that both consist of the same number of hierarchy levels. For two trees of different depths, this requirement can be achieved by adding further levels consisting of a single inner node at the top of one of the two trees.

4.2. Outline of the Algorithm

Since we traverse the tree breath-first and since at each hierarchy level only certain node pairs are to be visited, we have to store the indices of these node pairs temporarily during the traversal. For this purpose, we use a 2D buffer, which we will refer to in the following as *node pair index map*.

This buffer contains an array of index sets as follows. Let $L_j = \{i \,|\, \text{AABB}(S_i) \text{ overlap } \text{AABB}(T_j)\}$. Putting the contents of set L_j in the 2D buffer at row j, stored successively starting at the first pixel in this row, the complete buffer consists of m horizontal lines of different lengths. The lengths of all these lines (or, more precisely, their start and end points) are stored in a vertex array.

In addition, we require a second temporary 2D buffer, that we call *overlap count map*. This buffer consists of multiple levels, exactly as much as there are hierarchy levels in the AABB trees. As the node pair index map, also each level of the overlap count map consists of m horizontal lines of different lengths. The contents of the overlap count map are constructed during the AABB tree traversal at each hierarchy level L as follows.

At first, all those AABB node pairs that are to be visited at the considered level L are checked for overlap. Each such node pair corresponds to one entry in the overlap count map at level L. If the AABB overlap test of a certain node pair was positive and thus the corresponding child nodes are to be visited when processing the next hierarchy level, the number of these child nodes is written into the corresponding entry of the overlap count map. Otherwise the entry of the overlap count map is set to 0. How this is done using the GPU is described in Section 4.3.

If this step results in a map containing only 0-entries (what can be determined using an occlusion query), all AABB overlap tests have been negative, and therefore the two objects definitively do not collide.

Otherwise, the AABB tree traversal is continued as follows. Before the iteration proceeds to the next hierarchy level, the node pair index map has to be updated, as well as the vertex array containing the start and end points of the horizontal lines contained in this map. This is done in two steps. First, the vertex array is updated, as well as levels $0, \dots, L-1$ of the overlap count map. Second, the information contained in levels $0, \dots, L$ of the overlap count map is used to construct the node pair index map required as input for processing the next hierarchy level. These two steps are explained in detail in Section 4.4.

The whole process is repeated for all hierarchy levels as long as there are positive AABB overlap tests. If the last hierarchy level is reached, instead of testing AABB overlaps, triangle intersection tests are performed on the GPU for all leaf node pairs that have to be visited according to the node pair index map. Using an occlusion query, we obtain the number of intersecting triangle pairs for the considered objects. If required, the actual list of intersecting triangles can be obtained via read-back from graphics memory.

The outline of the overall algorithm is summarized in Fig. 1.

4.3. Box Overlap Tests

To perform the overlap tests of all AABB pairs corresponding to the indices contained in the node pair index map, we use a fragment program that is executed by rendering m horizontal lines, similar to the method described in Section 3. Here however, these lines have different lengths, as defined by the content of the vertex array. The fragment program uses the index i contained in the node pair index map to obtain the AABB of node S_i from a pair of 1D floating-point textures. One of these two textures contains the center point of the AABBs, while the other one contains the corresponding box extents. In addition, one of them contains also the number of child nodes of the tree node, which is 1 or 2.

This information is used in the fragment program to check whether the AABB pair overlaps by performing a *SAT lite* test [21]. This is shown in the fragment program subroutine found in Appendix A.

All fragments corresponding to non-overlapping AABB pairs are discarded by this program. Otherwise (not shown in the program), the number of children of S_i is written to level L of the overlap count map.

4.4. Generating the Node Pair Index Map

We construct the new node pair index map and the corresponding vertex array in multiple passes using the following technique.

The length of each horizontal line in the new node pair index map corresponds to the number of nodes of level $L + 1$ for whose parent nodes the AABB overlap test was positive. By construction, this number is equal to the sum of all values in level L of the overlap count map at the corresponding row.

Therefore, we can construct the vertex array by summing up these values. In analogy to the construction of 1D MIP maps on the GPU, we do this by constructing the levels $L - 1, \ldots, 0$ of the overlap count map as follows. Each level $i = L - 1, \ldots, 0$ of this map consists of $2^i \times m$ entries, each of which is calculated by summing up two values from level $i + 1$, see Fig. 2.

We store the entries of level 0 of the overlap count map, each of which corresponds to the sum of all values in the corresponding row of level L, in

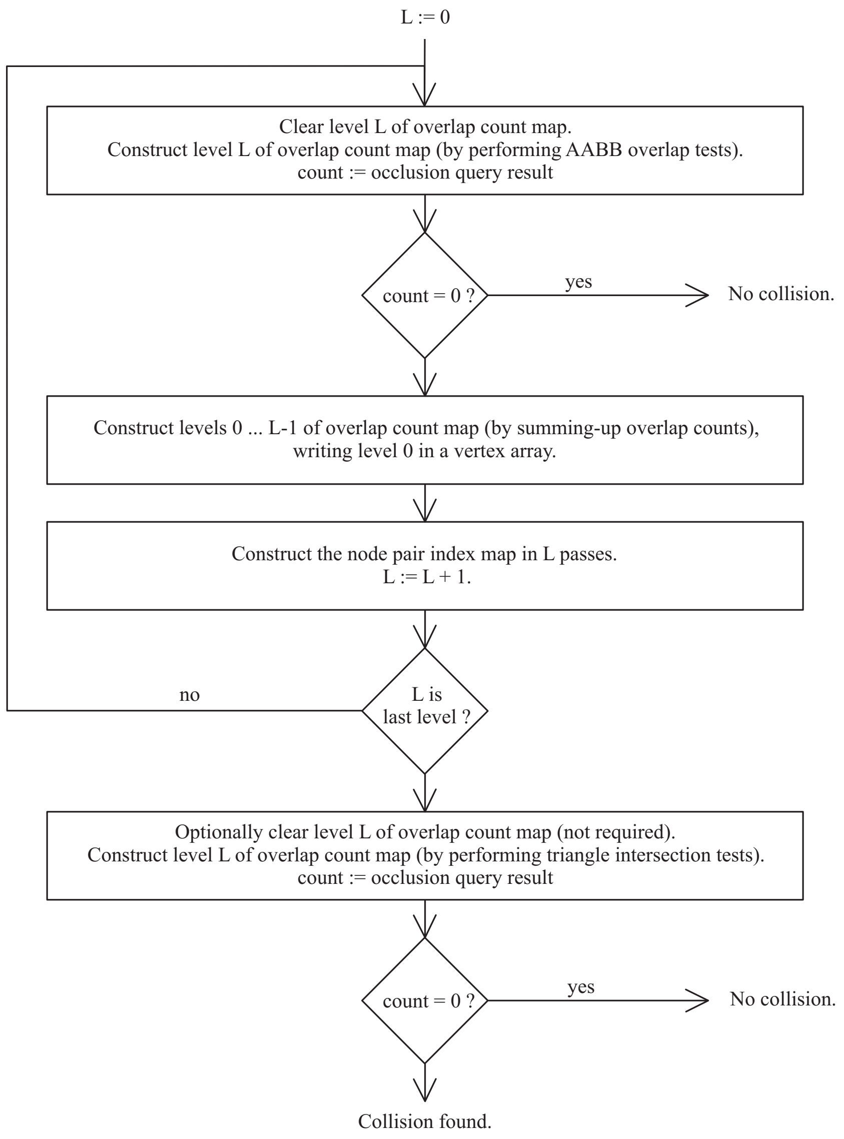

L := 0
Clear level L of overlap count map.
Construct level L of overlap count map (by performing AABB overlap tests).
count := occlusion query result
count = 0 ?
yes
No collision.
Construct levels 0 ... L-1 of overlap count map (by summing-up overlap counts), writing level 0 in a vertex array.
Construct the node pair index map in L passes.
L := L + 1.
L is last level ?
no
Optionally clear level L of overlap count map (not required).
Construct level L of overlap count map (by performing triangle intersection tests).
count := occlusion query result
count = 0 ?
yes
No collision.
Collision found.

Fig. 1. The outline of our approach.

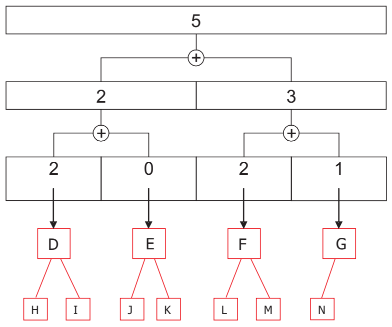


Fig. 2. Summing up the overlap counts.

a vertex array, such that they can be used as vertex program input for the AABB overlap tests at hierarchy level $L+1$. In our current implementation, this is accomplished by transferring the data from the render target texture directly to the vertex array using the `EXT_pixel_buffer_object` OpenGL extension.

Next, we construct the new node pair index map for hierarchy level $L+1$ in L passes as follows.

The basic idea is that for every row of the node pair index map the nth entry corresponds to the nth AABB tree node of level $L+1$ that is to be visited. This node can be found be traversing the AABB tree starting at the root node. Note, that the first entry of level 1 of the overlap count map contains the number of nodes of level $L+1$ to be visited that are reached from the root node via its first child node. Therefore, this value decides whether we must proceed to the first or to the second child of the root node to reach the searched node. Then, this step is repeated using levels $2,\ldots,L$ of the overlap count map.

This technique to construct the new node pair index map is realized on the GPU as follows. We need a temporary buffer of the same size as the node pair index map that we are going to construct, consisting of two components per entry. The first component, called *current node index* in the following, is used to store the indices of AABB tree nodes that are visited during the traversal. The second component, called *current child index* in the following, corresponds to n if we search the nth node of level $L+1$ to be visited that is reached from the node specified by the current node index.

At the beginning, this temporary buffer is initialized as follows: In each row of the texture, the current child index is numbered consecutively starting with 0. The current node index is initialized to 0 and thus addresses the root node. Each row of the temporary buffer is assumed to be of the corresponding length stored in the vertex array.

We use a fragment program that does the following for each pass $i = 1, \ldots, L$: First, the value from the overlap count map of level i at the index that equals $2 \cdot$*current node index* is read. Then, this value (*overlap count*) is compared against the current child index. If it is greater, the current node index is replaced by $2 \cdot$*current node index* in the temporary buffer, and the current child index remains unchanged. Otherwise, the current node index is replaced by $2 \cdot$*current node index*$+1$, and the overlap count is subtracted from the current child index, see Fig. 3.

After these L passes, the new node pair index map can be obtained based on the values in the temporary buffer and the current contents of the node pair index map as follows. For each entry of the map, we identify the actual corresponding AABB tree node by accessing the current contents of the node pair index map at the index specified by the current node index from the temporary buffer. Then we store the index of its first or second child, depending on the current child index from the temporary buffer, into the new node pair index map. This step is shown in the lower right of Fig. 3.

Instead of doing this last step in an additional render pass, it can be incorporated directly into the fragment programs used for box overlap and triangle intersection testing, such that the construction of the node pair index map requires L passes in total.

Note that the multi-pass construction technique described above is also suited for graphics hardware with dependent texture read limit (like the current ATI Radeon GPUs). On hardware without such a limit, the number of passes could be reduced by combining multiple of these passes into a single one, provided that total number of textures used in such a pass does not exceed the corresponding hardware limit.

§5. A Hybrid Approach Based on Temporal Coherence

It is clear that for a given object pair it might not be most efficient to start the hierarchy traversal at the top levels of the two hierarchies. If more than half of the box pairs of a certain level overlap, then we can save box overlap tests by starting the traversal at that level. Since the traversal itself incurs some further computational overhead, it might be more efficient to start at a certain hierarchy level even for a smaller percentage of overlapping box pairs. However, to decide which level is most efficient to start with, we would have to know the number of overlapping box pairs in advance, what is obviously not possible.

However, in a typical real-time application of collision detection objects are moving on a smooth path. Therefore, we can use a heuristic based on temporal coherence: we simply remember for a each object pair the highest hierarchy level at which more than a certain percentage of box pairs overlapped. This percentage is obtained using an occlusion query.

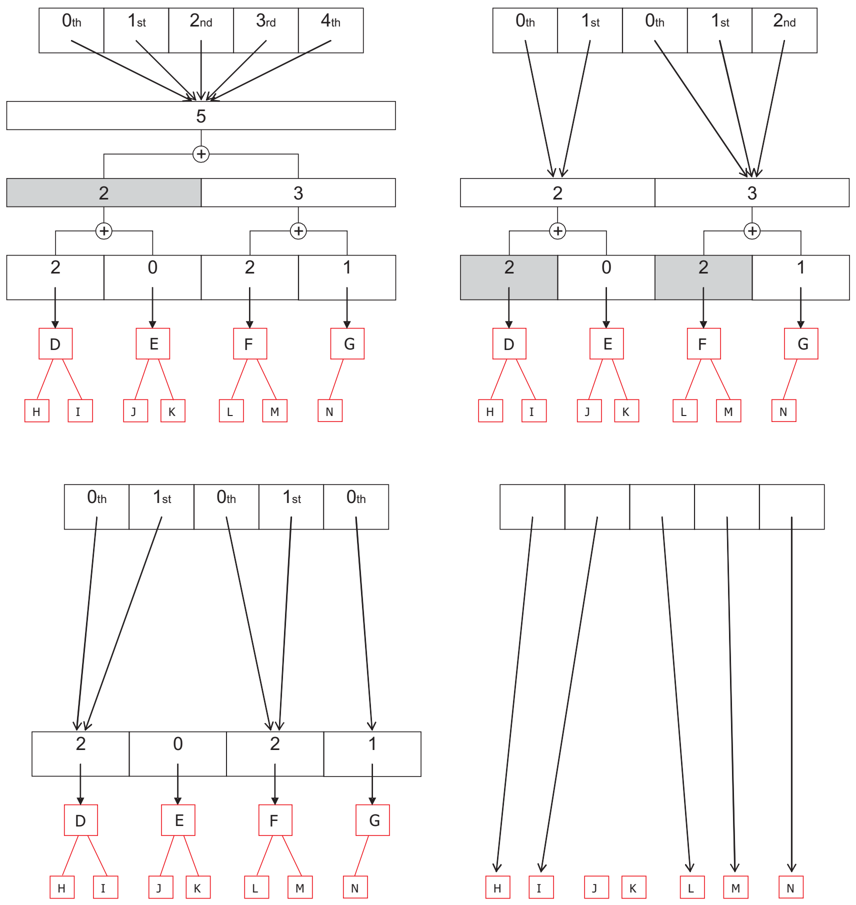


Fig. 3. Construction of the node pair index map in multiple passes. The values on gray background are the overlap counts that are compared to the current child indices shown at the top row.

For the next frame, we start the traversal of the hierarchy at exactly this level. Based on the number of overlaps we determine on this level now, we adjust the "entrance" level for the next collision check.

§6. Implementation

We implemented the method described in Section 5 in C++ using OpenGL on a NVIDIA GeForce FX 5900 GPU.

The AABB tree is stored using 32-bit floating-point textures with *rectangle* texture target (i.e. on texture lookup, the centers of the texels contained in a $m \times n$-sized texture are addressed using (u, v)-coordinates with $u = \frac{1}{2}, 1\frac{1}{2}, \ldots, m - \frac{1}{2}$ and $v = \frac{1}{2}, 1\frac{1}{2}, \ldots, m - \frac{1}{2}$).

The multi-level overlap count map and the node pair index map are both used as temporary buffers during the described algorithm. Therefore, we require a method that enables these textures to be used as targets for rendering operations. Probably the most efficient technique allowing this would be to use the upcoming `ARB_super_buffer` extension (which is currently under specification). However, since this OpenGL extension is not yet available for our target system, we had to use the *p-buffer* and *render-to-texture* WGL extensions instead.

However, due to driver limitations on the GPU we used, integer p-buffers cannot have more than 8 bits per component. As this would not be sufficient in our case, we use 16-bit half-precision floating-point p-buffers instead. (Their 11-bit mantissa is sufficient to accurately store values up to 4096, which corresponds to the maximum texture size on the used GPU and thus to our maximum index value.)

A disadvantage of using p-buffers is that each render target change causes a GL render context switch which is connected with degraded performance caused by pipeline flushes. Another disadvantage is that we cannot share occlusion queries between multiple render contexts, which would be of great benefit for delayed evaluation of the query result. Fortunately, these disadvantages are expected to disappear when the super buffer extension will be available. Furthermore, the super buffer extension will allow to render level 0 of the overlap count map directly to a vertex array instead of transferring the data from a p-buffer to the vertex array using the `EXT_pixel_buffer_object` extension (see Section 4.4).[2]

§7. Results

We tested the performance of our algorithm using a test scenario similar to that of [25]: Two identical objects are positioned at a certain distance

[2] Another alternative to transferring the data to the vertex array would be to access the texture containing level 0 of the overlap count map directly from the vertex program. However, this requires graphics hardware that allows to access textures from a vertex program, which was not available at the time when this work was done.

from each other. The distance is computed between the centers of the bounding boxes of the two objects; objects are scaled uniformly so they fit into a cube of size 2^3. One of the two objects is rotated around a fixed axis by a fixed number of small steps. In each step, the two objects are checked for collision, and the average collision detection time for a complete revolution at that distance is computed. Then the process is repeated with a slightly decreased distance of the two objects.

This test was performed on a set of CAD objects with varying complexities. To compare the performance of GPU and CPU based collision detection methods, we also implemented the AABB tree traversal approach on the CPU using an identical traversal scheme and ran the same tests on that implementation. Fig. 4 shows the results.

Note, that the timings include the determination of all intersecting triangle pairs as well as the read-back of its indices from graphics memory in case of the GPU implementation.

When comparing the performance of the individual steps of the algorithm between its GPU and CPU implementations, it turns out that in the GPU implementation the box overlap and triangle intersection tests perform up to four times faster especially at the lower hierarchy levels. However, this speed-up is reduced by the overhead of the node pair index map generation, which is larger in the GPU implementation. Overall, in general our current GPU implementation is slightly faster than the CPU implementation.

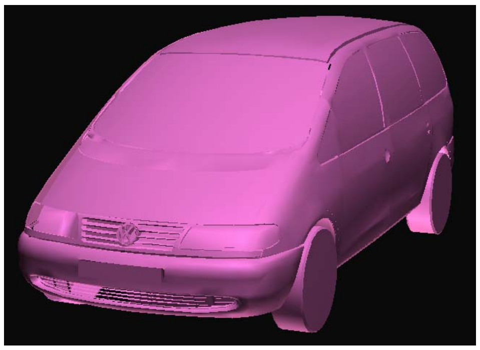
We also made some tests with a very early super buffer beta implementation on an ATI Radeon GPU, where we used the super buffer for rendering to our temporary textures as well as for directly rendering into the vertex array. However, due to some issues with this beta implementation, we cannot provide reliable results so far. But as soon as they will be resolved, we expect that the overhead of the node pair index map generation can be reduced by the use of super buffers.

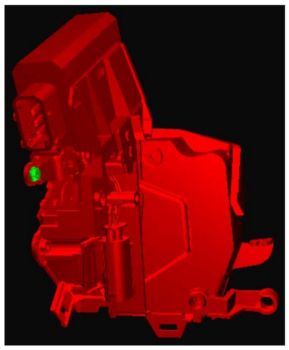
§8. Conclusion and Future Work

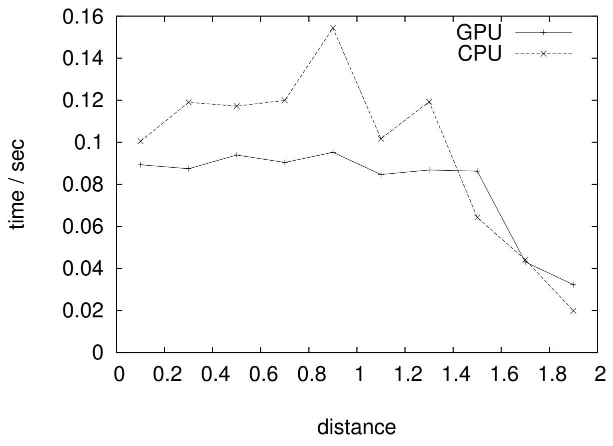
We presented a method for interference detection using programmable graphics hardware. Unlike previous GPU-based approaches, there are no requirements on shape, topology, and connectivity of the polygonal input models. Furthermore, all calculations are done in object-space rather than image-space.

Our method is a hierarchical algorithm that borrows ideas from traditional CPU based collision detection approaches. It simultaneously traverses a pair of bounding volume hierarchies consisting of axis-aligned bounding boxes. In addition, the method utilizes temporal coherence when used for collision testing over a period of time.

Although our current implementation can probably be improved by







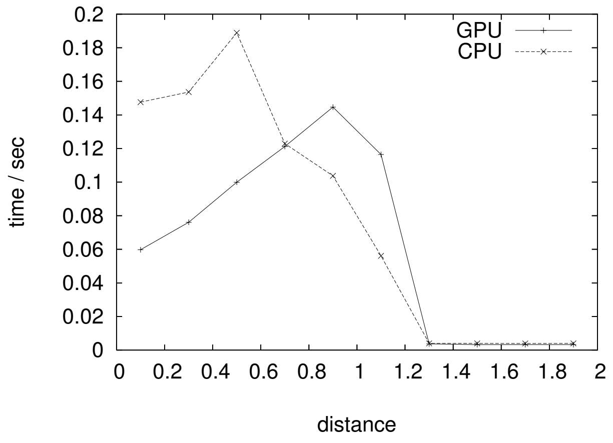


Fig. 4. The collision response times for the car are shown in the middle, and those for the door lock at the bottom. The corresponding polygonal models can be seen in the top row. (Data courtesy of VW and BMW.)

using upcoming hardware and driver features it is still able to compete with CPU based hierarchical collision detection approaches. This allows using the CPU idle time (for example while waiting for occlusion query results) for other tasks. And with the release of future GPUs, our method is potentially able to outperform the CPU based approaches clearly.

As there is no early termination in case of a determined intersecting triangle pair, the main target of this method are applications that need to determine the complete list of intersecting triangles or its number.

There are still possible enhancements and optimization opportunities we want to investigate in the future:

- One objective of future work is to enhance the algorithm for the use in complex environments where multiple polygonal models with its own object-space have to be checked against each other. It might be worthwhile to check if we can speed-up the collision detection in this scenario by starting the bounding volume traversal with processing multiple pairs of AABB trees at once, for example by accessing a matrix palette from the vertex program for handling the individual transformation matrixes.
- Our approach could easily be modified to use hierarchies based on bounding volumes other than AABBs. One future task would be to evaluate the influence of using different bounding volumes on the performance of our GPU based approach.
- Recently (after finishing the work on this paper), graphics hardware has become available that has dynamic flow control logic in the fragment program units and allows to access textures from vertex programs. Therefore, another objective of future work is to incorporate these features in our approach, for example by accessing level 0 of the overlap count map directly from a texture rather than transferring it to a vertex array first. Furthermore, we would like to explore how much the triangle intersection and AABB overlap tests can profit from dynamic flow control and if alternative triangle intersection test methods like [17, 9] are more efficient on this kind of hardware.

§A. Fragment Program Implementation

SAT lite test in high-level shading language:

```
void testBoxOverlap (float4 centerS, float3 extentS,
                     float4 centerT, float3 extentT)
{
    float3 dist;          // distance between centers
    float3 extsum;        // interval radii
    // compute difference of box centers in S space
    float3 distInS = float3(dot(centerT, matC[0]),
                            dot(centerT, matC[1]),
                            dot(centerT, matC[2]))
                   - centerS.xyz;

    // determine three potentially separating axes
    dist      = distInS;
    extsum    = extentS;
    extsum.x += dot(extentT, matAbsC[0]);
    extsum.y += dot(extentT, matAbsC[1]);
    extsum.z += dot(extentT, matAbsC[2]);

    // discard fragment if any component
    // of "extsum - abs(dist)" < 0
    clip(extsum - abs(dist));

    // determine three more potentially separating axes
    dist.x    = dot(distInS, matCInv[0]);
    dist.y    = dot(distInS, matCInv[1]);
    dist.z    = dot(distInS, matCInv[2]);
    extsum.x =  dot(extentS, matAbsCInv[0]);
    extsum.y =  dot(extentS, matAbsCInv[1]);
    extsum.z =  dot(extentS, matAbsCInv[2]);
    extsum   += extentT;

    // discard fragment if any component
    // of "extsum - abs(dist)" < 0
    clip(extsum - abs(dist));
}
```

where the uniform parameter *matC* is the transformation matrix from T's object-space to S's, *matCInv* is its inverse, and *matAbsC* and *matAbsCInv* are the matrixes obtained by taking the absolute value of each entry of *matC* and *matCInv*, respectively.

§B. References

1. Agarwal, P., M. de Berg, J. Gudmundsson, M. Hammar, and H. Haverkort, Box-trees and r-trees with near-optimal query time, Discr. Comp. Geom. **28** (2002), 291–312.

2. Agarwal, P., S. Krishnan, N. Mustafa, and S. Venkatasubramanian, Streaming geometric optimization using graphics hardware, in *Proc. of the 11th European Symposium on Algorithms*, 2003, 544–555.

3. Baciu, G. and W. S.-K. Wong, Hardware-assisted self-collision for deformable surfaces, in *Proc. of the ACM Symposium on Virtual Reality Software and Technology (VRST)*, ACM Press, 2002, 129–136.
4. Baciu, G. and W. S.-K. Wong, Image-based techniques in a hybrid collision detector, IEEE Trans. on Visualization and Comput. Graphics **9** (2003), 254–271.
5. Ehmann, S. A. and M. C. Lin, Accurate and fast proximity queries between polyhedra using convex surface decomposition, Comput. Graphics Forum **20** (2001), 500–510.
6. Gottschalk, S., *Collision queries using oriented bounding boxes*, PhD thesis, University of North Carolina at Chapel Hill, 2000.
7. Gottschalk, S., M. Lin, and D. Manocha, OBB-Tree: A hierarchical structure for rapid interference detection, in *SIGGRAPH 96 Conference Proceedings*, H. Rushmeier (ed.), Addison Wesley, 1996, 171–180.
8. Govindaraju, N., S. Redon, M. C. Lin, and D. Manocha, CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware, in *Proc. of Graphics Hardware*, San Diego, California, 2003, 25–32.
9. Held, M., ERIT: A collection of efficient and reliable intersection tests, Journal of Graphics Tools **2** (1997), 25–44.
10. Hubbard, P. M., Approximating polyhedra with spheres for time-critical collision detection, ACM Trans. on Graphics **15** (1996), 179–210.
11. Kapasi, U. J., S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, Programmable stream processors, IEEE Comput. (2003), 54–61.
12. Klosowski, J. T., M. Held, J. S. B. Mitchell, H. Sowrizal, and K. Zikan, Efficient collision detection using bounding volume hierarchies of k-DOPs, IEEE Trans. on Visualization and Comput. Graphics **4** (1998), 21–36.
13. Knott, D. and D. K. Pai, CInDeR: Collision and interference detection in real-time using graphics hardware, in *Proc. of Graphics Interface*, Halifax, Nova Scotia, Canada, 2003, 73–80.
14. Larsson, T. and T. Akenine-Möller, Collision detection for continuously deforming bodies, in *Proc. of Eurographics*, 2001, 325–333.
15. Lin, M. C. and D. Manocha, Interactive geometric computations using graphics hardware, in *SIGGRAPH 2002 Course Notes*, No. 31, 2002.
16. Lombardo, J.-C., M.-P. Cani, and F. Neyret, Real-time collision detection for virtual surgery, in *Proc. of Computer Animation*, Geneva, Switzerland, 1999, 82–90.

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

18. Myszkowski, K., O. G. Okunev, and T. L. Kunii, Fast collision detection between complex solids using rasterizing graphics hardware, Visual Comput. **11** (1995), 497–512.

19. Palmer, I. J. and R. L. Grimsdale, Collision detection for animation using sphere-trees, Comput. Graphics Forum **14** (1995), 105–116.

20. Shinya, M. and M.-C. Forgue, Interference detection through rasterization, Journal of Visualization and Comput. Animation **2** (1991), 132–134.

21. van den Bergen, G., Efficient collision detection of complex deformable models using AABB trees, Journal of Graphics Tools **2** (1997), 1–14.

22. Zachmann, G., Rapid collision detection by dynamically aligned DOP-trees, in *Proc. of IEEE Virtual Reality Annual International Symposium (VRAIS)*, Atlanta, Georgia, 1998, 90–97.

23. Zachmann, G., Minimal hierarchical collision detection, in *Proc. of ACM Symposium on Virtual Reality Software and Technology (VRST)*, Hong Kong, China, 2002, 121–128.

24. Zachmann, G. and G. Knittel, An architecture for hierarchical collision detection, in *Journal of WSCG '2003*, University of West Bohemia, Plzen, Czech Republic, 2003, 149–156.

25. Zachmann, G. and G. Knittel, High-performance collision detection hardware, Tech. Rep. CG-2003-3, University of Bonn, Bonn, Germany, 2003.

Alexander Greß and Gabriel Zachmann
University of Bonn
Bonn, Germany
`{gress|zach}@cs.uni-bonn.de`
`http://cg.cs.uni-bonn.de`