Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS (2009)
H. Prautzsch, A. Schmitt, J. Bender, M. Teschner (Editors)

GPU Accelerated Tandem Traversal of Blocked Bounding Volume Hierarchy Collision Detection for Multibody Dynamics

J. Damkjær [1] and K. Erleben[1]

[1]Department of Computer Science, University of Copenhagen, Denmark

Abstract
The performance bottleneck of physics based animation is often the collision detection. It is well known by practitioners that the collision detection may consume more than half of the simulation time. In this work, we will introduce a novel approach for collision detection using bounding volume hierarchies. Our approach makes it possible to perform non-convex object versus non-convex object collision on the GPU, using tandem traversals of bounding volume hierarchies. Prior work only supports single traversals on GPUs. We introduce a blocked hierarchy data structure, using imaginary nodes and a simultaneous descend in the tandem traversal. The data structure design and traversal are highly specialized for exploiting the parallel threads in the NVIDIA GPUs. As proof-of-concept we demonstrate a GPU implementation for a multibody dynamics simulation, showing an approximate speedup factor of up to 8 compared to a CPU implementation.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.5]: Physically based modeling—Computer Graphics [I.3.7]: Animation—

Keywords: Physics based Animation, Collision Detection, Computer Animation, Bounding Volume Hierarchies, Graphics Hardware

1. Hierarchies for GPUs

Collision detection is an important part of physics based animation and a large research area. Two introductions and surveys to collision detection are [Eri05, vdB03]. It is a well known fact that collision detection is a major performance bottleneck in physics based animation and simulation [HTG04,RZ08]. This has motivated our work. We focus on general exact methods that can be used for non-convex objects and triangle soups. Thus, approximations and non world-space methods such as [GRLM03, KP03, JH08] are not applicable.

A specific method for doing exact collision detection between two general objects is bounding volume hierarchies [Eri05]. Here objects are subdivided into increasingly smaller sub-objects in a hierarchic fashion, until single triangles are reached. In this paper, we will approach the performance bottleneck, by utilizing modern GPU hardware for handling the collision detection. We present a method in

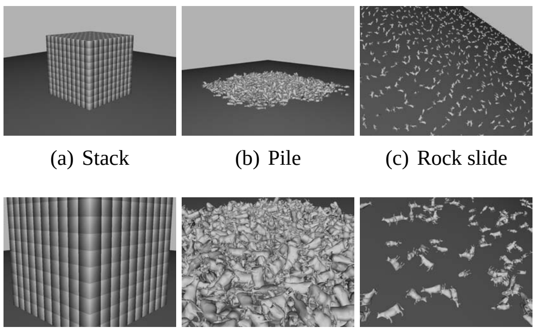

(a) Stack (b) Pile (c) Rock slide
(d) Close up of stack (e) Close up of pile (f) Close up of rock slide

Figure 1: *Examples of our GPU based bounding volume hierarchy collision detection method used in three different configurations.*

DOI: 10.2312/PE/vriphys/vriphys09/115-124

which the collision detection for multiple objects are handled in parallel, and where the parallel threads support each other in fetching the bounding volumes from memory.

Much work has been done on bounding volume hierarchies in the field of Computer Graphics. Examples are: approximating sphere hierarchies and time-critical collision detection [Hub95], OBB hierarchies [GLM96], k-DOPs [KHM*98]. Bounded deformation for fast updating of bounded sphere threes [JP04], chromatic decomposition to check for collisions between non-adjacent primitives using a linear-time culling algorithm [GKJ*05], filtering technique for continuous collision detection to avoid redundant testing [WB06], and continuous collision detection [TCYM08]. However, much of this work is not directly applicable for modern GPU based implementations.

In general, there are many works using the GPU for collision detection. Baciu and Wong [BW02] dynamically generates a hierarchy of cloth bounding boxes, in order to perform object-level culling and image-based intersection tests using conventional graphics hardware support. The method can handle 50K element cloth meshes at interactive rates. [KOLM02] computes the minimal translation distance to separate two polyhedrons using a combination of object-space and image-space techniques. Results are computed within 0.01-1 seconds for intersecting objects of 1-2K elements. In [GRLM03] an image-spaced method is presented, which computes a potentially colliding set using visibility queries. The method only checks for overlapping objects, has no support for self-collision, and its accuracy is limited by image-space resolution. The method can handle 285K triangles at 35 milliseconds per frame, on a GeForce FX Ultra 5800 GPU. Later, the accuracy problems were improved in [GLM05a] and extended to include self-collisions in [GLM05b]. In [GZ03] a shader programming approach is taken to bounding volume hierarchy collision detection. However, the GPU results are reported to be only slightly faster than a CPU counter part. Our results indicate close to an order of magnitude improvement in the best cases. An image space method for convex object versus non-convex object testing using six view volumes were presented in [FWG04]. A real-time voxelization technique was applied in [CWZ*04] similar to the layered depth image method by [HTG04]. The voxelization method is capable of dealing with 1.8M triangle objects within 100ms and the accuracy is limited by voxel resolution. Both approaches are restricted to objects with closed watertight surfaces. Geometry images were proposed in [BV05], where bounding spheres are stored in a perfectly balanced mip-map like hierarchical data structure. The method is shown to work with 60M vertex objects at very fast rates. However, the work is limited to zero genus objects, and not general triangle soups as we consider in our work. k-DOPs bounding volume hierarchies were hardware accelerated in [RBAZ05] using VHDL and not GPUs. Geometry images were revisited in [ZH07] tailored specific for regular pieces of cloth.

Havok and NVIDIA showed with Havok Fx that a large part of a physics engine could be implemented on GPU [Hav09]. Others have done complete particle based physics simulations on the GPU using shaders [Har07]. Implementation of broad-phase collision detection has been done [Gra07]. Recently, PhysX and Bullet have emerged [NVI09, bul09]. Bullet supports non-convex object versus non-convex object collision detection using the non-threaded CPU-based collision detection library GIMPACT [gim09]. For real-time interactive computer games Bullet recommends, due to performance considerations, to use approximating compounds for non-convex polygonal objects instead of BVH based collision detection. Similarily, Open Dynamics Engine has replaced the OPCODE library with GIMPACT [ode09]. We can only speculate about the internal workings of PhysX. However, there is support for triangle mesh terrains, although we have not been able to find any triangle mesh versus triangle mesh collision detection feature in PhysX.

In [TS05] a new data layout of a bounding volume hierarchy was presented. Using escape indices allowed one to encode a static descend rule for single traversals. The static descend rule made it possible to use the data layout for ray casting on a GPU. The ideas are now common place for primitives against nonconvex object collision testing queries, but can not support nonconvex object versus nonconvex object collision testing as our method can. Furthermore, our work supports completely general dynamic descend rules. A recent GDC presentation was made on a History flag method for bounding volume hierarchy collision detection on GPUs [Har09]. From the presentation, it is not clear whether the work only considers single traversals or is applicable to tandem traversals as our work is, nor are any performance measurements given for us to compare with. It seems that the History flags method uses AABBs and stores 4 bits per level during traversal, besides all nodes on the path to the root. No details are given about how collision results are collected nor details on load balancing. Our method has no need for any run time book keeping, and our blocking should provide us with better data locality than the non blocked History flag method. Further, as far as we know no open source projects or software libraries are currently available, supporting parallel or distributed bounding volume hierarchy collision detection. Thus, existing work addressing object-space collision detection on GPU hardware for general large scale triangle soups is sparse. Observe that in our work we address scenes of triangle soups of up to a total of 60M triangles.

In 2007 NVIDIA introduced Compute Unified Device Architecture (CUDA) [CUD09]. CUDA allows programmers to use the GPU as an extra processor, and do development in the high-level language C. Thus, CUDA has made it easy to use the GPU for offloading highly parallel computations from the CPU. Programmers are no longer required to make a creative mapping between their problem and textures, and colors. Programmers using CUDA get the possibility to do scatter operations. This is in contrast to older shader programs which only allow gather operations. In a way, CUDA offers a clean break away from the classical way of GPU programming by writing shaders. However, a bundle of new problems emerge, e.g. the programmer has to control each processor and the memory used. ATI has a technology similar to CUDA, called AMD Stream SDK. This uses a combination of high and low level programming language. Recently, Open Computing Language was specified [Gro09]. In this paper we focus on NVIDIA hardware due to the possibility to develop in C.

One constraint with the modern NVIDIA GPUs is that they do not have cache for all their different types of memory [NVI08]. Requesting data from memory therefore becomes expensive. There are some forms of cache: the texture cache which caches nearby pixels when accessing the texture memory, and the constant cache which is a small cache for the constant memory. Another constraint is, that there is a limited amount of registers available for doing the computations on the GPU. A general stack-based or queue-based approach for doing bounding volume hierarchy collision detection may therefore be slow, since the stack or queue would have to be stored in the GPUs global memory. In this paper, we introduce a method for performing bounding volume hierarchy collision detection on NVIDIA GPUs using CUDA, where the threads aid each other in retrieving data from memory.

We will first present our method. Following our presentation, we compare performance of our GPU implementation to a CPU implementation. Finally, we conclude and make suggestions for future work.

2. The Blocked Hierarchy and Simultaneous Descend Method

Each object is represented by a bounding volume hierarchy, and a tandem traversal scheme is used to find collisions between a pair of objects. When testing for collisions the traversal descends either of the two hierarchies, adding new collision pairs to either a stack or a queue. The traversal may descend both hierarchies at once, thereby adding more new collision pairs at a single instant.

We will introduce a method for bounding volume hierarchy collision detection descending simultaneously in both hierarchies. Furthermore we will utilize NVIDIA GPUs. We will start by explaining the hierarchy layouts and the bounding volumes. Hereafter we will discuss descend problems followed by hardware specific and memory discussions.

Use Blocked Data: A bounding volume hierarchy contains bounding volumes at each node, encapsulating the triangles contained in the sub-hierarchy of the node. A bounding volume can e.g. be a sphere, an axis-aligned bounding box (AABB), an oriented bounding box (OBB) etc [Eri05]. We have chosen to use AABBs for three main reasons. First of all, the AABB has a smaller memory footprint than e.g. an OBB. Second an AABB has better pruning capabilities than a sphere [Got00]. Third, using AABBs allow us to handle deformable objects in the future. This is not the focus of this paper and will therefore not be discussed further. Notice it is straight forward to change our method to handle either OBBs or spheres.

We store all nodes in a hierarchy in one contiguous array. Each node is identified uniquely by an index to the array entry holding the node. In our blocked hierarchy each node has an index to a *block* of children. Currently we use a block size of four children since this fits well with the number of scalar processors and threads running in a multi processor. Thus, a block must always have four consecutive nodes. The advantage is that a block of nodes can be identified by a single index to the first node in the block. To ensure that all blocks are filled up with exactly four nodes we may have to add *imaginary* nodes when constructing a hierarchy. The imaginary nodes serve no other purpose than ensuring a nice mapping of the memory. However, the usage of imaginary nodes should be minimized if possible. Not only does the hierarchy require more space, but the imaginary nodes are also processed during the execution of a collision query although as pass-through. A block of nodes can all be leaf nodes in which case we call it a leaf block, or the block can contain a mix of internal nodes and leaf nodes in which case we call it a partial leaf block. If all nodes are internal nodes we call it an internal block.

A collision detection query consists of several iterations. In each iteration both bounding volume hierarchies are descended, collision pairs are tested for overlap, possible colliding triangles are placed in a separate triangle pair array, and new collision pairs are prepared for a future iteration. This will continue as long as overlapping bounding volumes are found.

When finding a collision between two bounding volumes, new collision pairs will have to be added to a collision pair array for a future iteration. Since each node has four children, 16 new collision pairs will have to be added, because all four nodes from one hierarchy must be tested for overlap against all four nodes from the other hierarchy. However, it is not necessary to add all 16 collision pairs to the collision pair array, since the four nodes in a block lies consecutively in memory. All 16 collision pairs are therefore implicitly given from the indexes to the two blocks, see Figure 2. To process a collision pair test, one must fetch the two corresponding blocks, one from each object hierarchy, resulting in a total of 8 bounding volumes.

Avoid Descend Problems: One problem may arise during a descend when comparing an internal block with a leaf block. In this case it is not possible to continue descending both hierarchies. Either we can choose to add the collision pairs to another new array, which will be handled by another

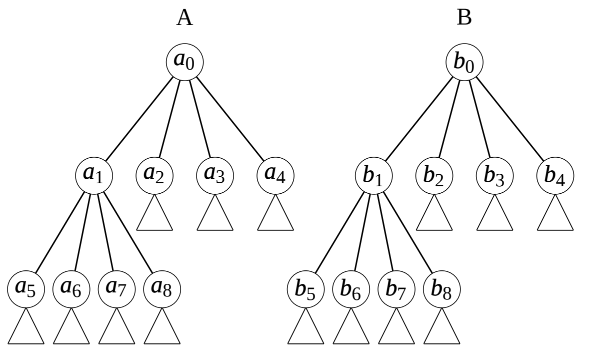


Figure 2: *Two bounding volume hierarchies, A and B. Triangles denote arbitrary sub-hierarchies. If a collision is found between the AABBs of the nodes a_1 and b_1 then it is only necessary to add the collision pair (a_5 , b_5) to the collision pair array. When the collision pair (a_5 , b_5) is retrieved all 16 collision pairs (a_5 , b_5), (a_5 , b_6), (a_5 , b_7), . . . , and (a_8 , b_8) will be given implicitly due to the contiguous memory of blocks.*

kernel optimized for handling a single leaf node versus an internal block, or we can choose to continue to add new collision pairs to the collision pair array, thereby not utilizing all 16 threads. Both solutions have disadvantages. In the first solution, adding the nodes to another array means that we will have to run yet another kernel, either within the same iteration, or in another set of iterations when all bounding volume collision pairs have been found. This will take up both time and memory, and add complexity to the method. In the second solution, adding the nodes to the existing collision pair array means that it is possible to add the same collision pair more than once. Since we only add a single of the 16 collision pairs to the collision pair array, we can not add an internal block and a single leaf node. But we can add an internal block and a leaf block, suggesting that we descend in one of the hierarchies but not in the other. We achieve this by adding a collision pair containing the child index of the descended hierarchy, and current index to the leaf block from the other hierarchy. In this case, we must take care, not to add the same collision pair more than once. This can happen if one bounding volume node in one of the hierarchies collide with two leaf nodes in the other hierarchy, see Figure 3. We have chosen to use the second solution although it adds a small amount of complexity to the kernel, it is still simpler than writing a new kernel, thereby we achieve a more simple overall method. As a result of using the second solution, some of the threads will perform overlap test that have already been done in a previous iteration. Hence, some of the 16 threads will only be used for fetching data from memory, since their overlap tests are superfluous.

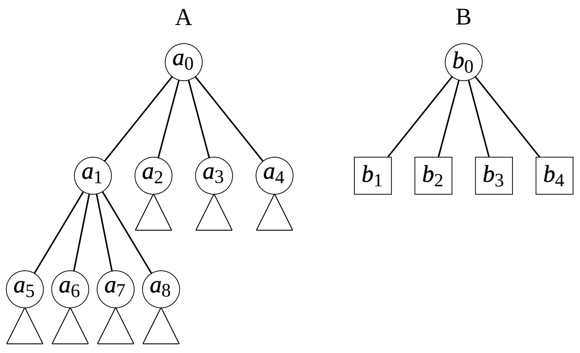


Figure 3: *Squares denote leaf nodes. Assume a collision between a_1 and b_1 and a collision between a_1 and b_2. Since we only can add blocks to the collision pair array, the collision pair (a_5 , b_1) will be added twice, unless this cases is handled properly.*

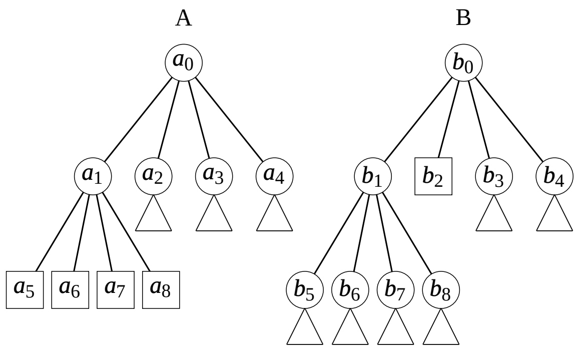


Figure 4: *Assume a collision between a_1 and b_1 and a collision between a_1 and b_2. This will add the collision pairs (a_5 , b_5) and (a_5 , b_1). If, in the next iteration, a collision between a_5 and b_1 is found then the collision pair (a_5 , b_5) will be added again.*

If a bounding volume node has a partial leaf block as a child. Then it is possible to add the same collision pair to the collision pair array more than once. The problem is shown in Figure 4. A solution to this problem is to restrict the hierarchies to disallow nodes from having partial leaf blocks as children. That is, a node can only have internal blocks or leaf blocks as children. This restriction does not have any impact on the method. Except if imaginary nodes are used more frequently by the hierarchy construction method to compensate for the lack of partial leaf blocks.

The GPU Overview: An overview of the blocked AABB GPU collision detection method is given here. A collision pair, containing indexes to bounding volume blocks in the two hierarchies together with an object pair index, is read from memory. Using the object pair index a transformation is read, allowing us to transform one of the AABBs from one object into the coordinate system of the AABB from the other object. The 16 threads running on one multiprocessor will read 4 nodes from each hierarchy. The work is shared such that each thread reads half of an AABB bounding volume node. Each of the 16 threads will then perform a separating axis test for its designated pairs of AABBs. If two boxes collide then a collision flag is written back to a *collision flag array*, or in case of a leaf/leaf overlap occurs, a leaf overlap flag is written back to *leaf flag array*. Here we ensure that if a bounding volume node collides with one or more leaf nodes, then only one flag is written to memory. This was described in detail earlier.

We perform a prefix sum scan on the collision flag array, thereby getting consecutive numbers for the collision pairs. A collecting kernel can then be issued retrieving the collision pairs, and copying them into a consecutive array, ready for the next iteration. The same is done simultaneously for overlapping leaf nodes. The array of consecutive leaf nodes are then added to an array of overlapping leaf nodes that later will be tested for overlapping triangles. When this is done the current iteration ends, and if there were found colliding bounding volume nodes, a new iteration will be started. When there are no more collision pairs a kernel is run to test the triangle pairs for overlap. This kernel performs a separating axis test on a pair of triangles. In order to avoid a lot of memory allocation, on the GPU the memory will be preallocated at the beginning of each collision query, based on the largest amount of memory used in the previous query. Pseudo-code for the complete method can be found in Listing 5.

Keep Data Alignment: The memory footprint of a single AABB bounding volume node is a position, an extend, an index to a block of children or an index to a triangle, and info about whether the node is an internal node, a leaf node, or an imaginary node. The position takes up 3 floats. The extend also takes up 3 floats. The child index or triangle index can be stored in the same integer since it is not possible to have both a child and be a leaf node. The node type info can be stored in 2 booleans. If we store the 2 booleans in an integer, we can store the complete bounding volume node in 6 floats and 2 integers, and if we store the 2 integers in floats, we can store an AABB bounding volume node in 2 of the float4 types of CUDA.

By reducing the maximum value of the children block and triangle indexes, we can store the index and the two booleans in a single integer. This will enable us to store the bounding volume in 7 floats, or a float4 and a float3. Thereby we save 4 bytes per bounding volume node. However, our experience indicate, that a float3 takes significantly more time to retrieve from memory than a float4. We will therefore use the extra 4 bytes in each bounding volume node. A half-warp is a small batch of threads running simultaneously. On current NVIDIA GPUs 16 threads corresponds to a half-warp. Each of these 16 threads can share the burden of retrieving the 8 bounding volumes from memory, by only retrieving a single float4 each.

Create Consecutive Results: On GPUs there are no ways of maintaining a dynamic data structure like e.g. a C++ STL vector, where we can add elements as they are found. Since the tandem traversal finds an unknown number of collisions in each iteration, we could just allocate enough memory for the possibility that all collision tests yield an overlap, i.e for each thread we have one memory location large enough to contain a collision pair. A thread could then write into its own designated location. At the same time, we need an array where we can designate whether a collision is found or not in a given thread. The second array, the collision flag array, will be used in another pass to find a unique consecutive index for each collision pair in order to do a collection operation. A similar array, leaf flag array, is needed for the triangles. All of this requires a large amount of memory for each collision pair, where potentially no new collisions are found. We will therefore try to minimize the amount of memory at the cost of a few extra memory look ups. Instead of writing the collision pair into a designated position as described above, we will just indicate in the collision flag array whether a collision is found. When we later do the collection routine, we will find the exact collision pair. This pair is well known since a thread is working on a specific collision pair. If a collision is found, the collision pair to add for next iteration is based on whether one of the bounding volume nodes is a leaf or not.

3. Comparison of GPU and CPU

A naive implementation of a stack-based traversal would need 32 integers to maintain a stack of 16 elements. The shared memory on NVIDIA 8- and 9-series GPUS consist of 16384 bytes, where some of these are used by the multiprocessor. Therefore, if the stack was to be maintained in the shared memory on the GPU then there would be less than 128 threads per multiprocessor. This low number of threads do not exploit the massive parallelism of the GPUs. Thus the stack would have to be maintained in the GPUs global memory, significantly slowing down the performance due to data fetching. Therefore, a GPU version of a naive stack-based implementation is not feasible to compare with.

To our knowledge, no comparable GPU methods exist. Therefore we compare with an accepted standard CPU method, Rapid [GLM96]. Rapid can handle the exact same types of triangle meshes as our method, though Rapid uses OBBs where we use AABBs. This should give Rapid better pruning capabilities. Both our method and Rapid have been integrated into the same simulator. The simulator is a constraint based simulator, using a velocity-based

```
algorithm GPU collision detection ( )
  allocate memory
  copy collision pairs to GPU
  while collision pairs
    run collision kernel
    prefix sum scan on collision flag array
    retrieve collision pairs and copy into new collision pair array
    prefix sum scan on leaf flag array
    retrieve triangle pairs and copy into potential triangle pair array
  end while
  test triangles in potential triangle array for overlap
end algorithm
```

Figure 5: *Pseudo code illustrating the complete GPU collision detection method.*

complementarity formulation and a projected Gauss-Seidel solver [ESHD05]. For both implementations, the time to perform the collision detection is measured. The measured time for collision detection is only for finding pairs of possibly overlapping triangles. Finding the contact points is not part of our method. The computer used for the tests has an Intel Core-2, Quad CPU Q6600 running at 2.40GHz, and an NVIDIA 9800GX2 graphic cards. It should be noted, that even though both the CPU and the GPU have more than one processor core, 4 for the CPU and 2 for the GPU. For simplicity and fairness, only a single core is used on both.

Performance of the collision detection pipeline is highly dependent on the type of configuration that is being simulated, the number of objects and the geometric complexity of the objects. Thus, we have designed a test-bed that tries to capture all the performance dependencies on these parameters. Our test-bed includes more than 1350 test simulations taking roughly 2 days in total for one person to conduct.

In [Erl07], a taxonomy of rigid body configuration types are defined having different contact statistics. From this taxonomy we have extracted the three configuration types which we believe span the extreme behaviours, when considering bounding volume hierarchy collision detection.

A structured stack is very compact, having many objects in close proximity with large areas of contact. The conjecture is that many collision queries will be performed, and all queries will reach many leaf-leaf tests. Our expectation is that this type of configuration will benefit the most from a GPU acceleration. An unstructured stack, like a random pile, will have voids in between objects, thereby reducing the contact areas between objects, compared to a structured stack. Although objects are in close proximity, we expect queries, that are able to prune many unnecessary leaf-leaf tests. Our expectation is that this type of configuration yields lesser performance speedup than the structured stack. Where stacking is highly static and very contact intensive, a dynamic configuration such as a rock slide will have many moving objects. Many possibly colliding object pairs are separated, and a collision query would quickly be halted. A few objects would be impacting, for instance with a support plane. The contacts will be nearly point-wise indicating that the tandem traversal will descend to a few number of leaf-leaf tests. It is our hypothesis that this type of configuration benefits from the pruning capabilities of the hierarchies, and as such will benefit the least from a GPU acceleration. We expect all configurations to benefit more from GPU acceleration when the number of objects is increased, or when the geometric complexity increases. When increasing the number of objects we expect the number of pair-wise collision queries to increase, and when increasing the geometric complexity we expect more leaf-leaf tests to be performed.

The contact area is estimated by computing the ratio of the number of triangle overlaps divided by the total number of triangles in each test configuration. The results are shown in Figure 6. This validates our hypotheses that a structured stack has large contact areas where pile has lesser contact area and rock slide has almost no contact area.

We have not spend time on constructing optimized hierarchies. As a result, roughly one third of our hierarchies consists of imaginary nodes. We expect this will give us performance penalties. It is expected that we will perform more collision overlap tests than Rapid since OBBs have better pruning capabilities than AABBs. This will result in a performance penalty for our method.

The rock slide contains a number of cows rolling down a tilted ground object. The test is run with five different numbers of cows, 500, 1000, 1500, 2000, and 2500. For each run, the cows are subdivided zero, one or two times which gives 1500, 6000, and 24000 triangles. The cows have rolled for 2000 frames. When cows fell out of the bottom, they were inserted at the top, to maintain the same amount of objects in the scene. The timing is an average of the 2000 frames. The unstructured stack contains a number of cows lying in a pile. The test is run with five different pile sizes, with 216, 343, 512, 729 and 1000 cows. For each pile size, the cows are subdivided zero, one or two times which gives 1500, 6000,

	Triangles per Object		
Objects	1500	6000	24000
1500	0.0240	0.0009	0.0004
2000	0.0290	0.0010	0.0004
2500	0.0038	0.0150	0.0007

(a) Rock Slide

	Triangles per Object		
Objects	1500	6000	24000
512	0.0747	0.0191	0.009
729	0.0871	0.0221	0.005
1000	0.0566	0.0134	0.035

(b) Pile

	Triangles per Object		
Objects	12	48	192
512	6.5990	6.8112	6.9840
729	6.5990	6.8112	6.9840
1000	6.5990	6.8112	6.9840

(c) Stack

Figure 6: *The amount of contact area is estimated as the number of colliding triangles divided by the total number of triangles. This allows us to compare differences in the amount of contact area for different configuration types. Observe that the amount of contact area is largest for the stack configuration.*

and 24000 triangles. Each run was repeated ten times, and the result is an average. The structured stack contains a number of boxes which are stacked closely together in a 3D grid. The test is run with five different grid sizes, from six to ten, which gives from 216 to 1000 boxes. For each grid size, the boxes are subdivided zero, one, and two times which gives from 12 to 192 triangles. Each run was repeated ten times, and the result is an average. The reason this test is run with boxes and not cows as the other test setups is that boxes are easier to stack than cows.

The amount of memory needed on the GPU depends on the number of overlaps found. Therefore, in the rock slide configuration with the lowest amount of overlaps, more objects can be added to the simulation. Fewer objects can be used in the unstructured pile, and the fewest amount can be used in the structured stack. Thus, maximum triangle counts of 60M, 24M and 19K (for the rock slide, pile and structured stack respectively) are the largest possible configuration bounds by the hardware we used.

The test results are given for the running time of Rapid, CUDA with copying data to and back from the GPU, and CUDA only. Complete summarizing graphs of all 1350 test results can be seen in Figure 7. For detailed viewing we have shown a few selected test runs in Figure 8.

As the results show, our CUDA method is faster than Rapid in all configurations except for the rock slide with 500 objects. The speedup becomes more noticeable when there are many objects in the scene. When using more than 500 objects, rock slide has generally the lowest speedup, at up to twice that of Rapid. The unstructured stack with more collisions have a larger speedup at about 3 to 7 times. The largest speedup is in the structured stack with many collisions at 5 to 8 times that of Rapid. Notice, that even with copying data to the GPU and back again, our CUDA method is still faster than Rapid. The structured stack benefited the most from our method, as expected. However, we would have expected a larger difference between the stack and pile. This may be caused by either the many imaginary nodes in our hierarchies, the worse pruning capabilities of the AABBs over the OBBs or a combination of both.

4. Conclusion

We have introduced, implemented and tested a method for doing bounding volume hierarchy collision detection on NVIDIA graphics hardware. Our method utilize consecutive threads to fetch bounding volume nodes from memory thereby lowering the amount of data read by a single thread. The presented method uses AABBs but is not restricted to this type of bounding volumes.

In almost all test configurations, our method was faster than the CPU reference, Rapid. We believe this shows that there is much potential in using the GPU for collision detection.

Our method has some potential areas for future improvement. First, it would be interesting to try to use OBBs instead of AABBs. Second, constructing better hierarchies with few or zero imaginary nodes should improve the performance since fewer unnecessary overlaps need to be performed. Third, the collision flag array and the leaf flag array both could be created as binary arrays since the only information stored in them is whether a collision or triangle overlap is found or not. This should minimize the amount of memory used, making it possible to handle larger scenes.

5. Acknowledgements

The authors would like to thank NVIDIA for their donation of graphics hardware, and the eScience Center, Faculty of Science, University of Copenhagen for their donation of computer hardware.

References

[bul09] Bullet physics library. web, July 2009. Open source project, `http://www.bulletphysics.com/`.

[BV05] BENEŠ B., VILLANUEVA N. G.: Gi-collide: collision detection with geometry images. In *SCCG '05: Proceedings of the 21st spring conference on Computer graphics* (New York, NY, USA, 2005), ACM, pp. 95–102.

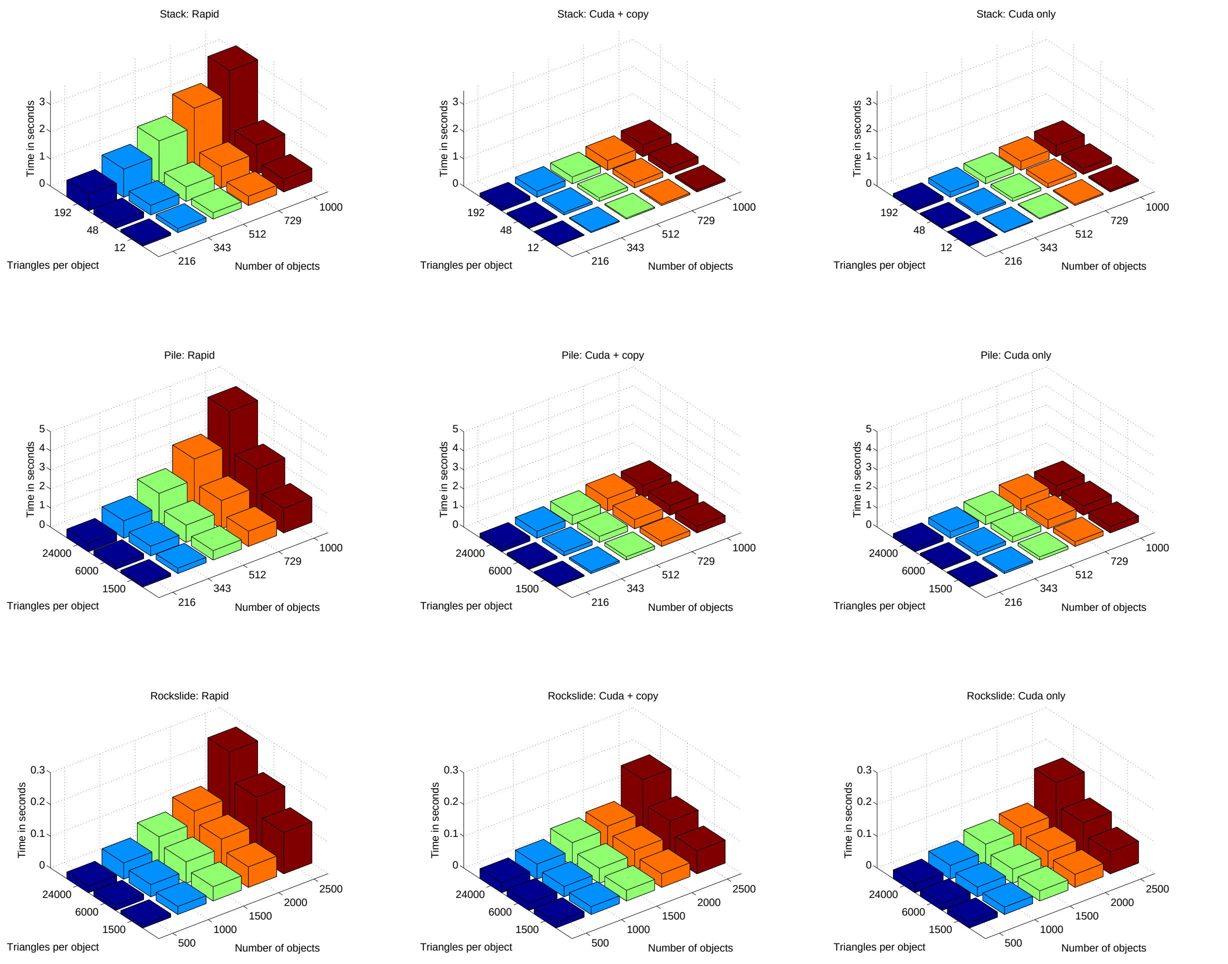


Figure 7: *Results of the test runs. Structured stack speedup of 5 to roughly 8 and unstructured stack speedup of 3-7. The rock slide speedup is only up to about 2. Notice that Rapid is a little faster for 500 objects in the rock slide configuration. This is due to the better pruning capabilities of the OBBs.*

[BW02] BACIU G., WONG W. S.-K.: Hardware-assisted self-collision for deformable surfaces. In *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology* (New York, NY, USA, 2002), ACM, pp. 129–136.

[CUD09] CUDA: Compute unified device architecture. web, July 2009. http://www.nvidia.com/object/cuda_home.html.

[CWZ*04] CHEN W., WAN H., ZHANG H., BAO H., PENG Q.: Interactive collision detection for complex and deformable models using programmable graphics hardware. In *VRST '04: Proceedings of the ACM symposium on Virtual reality software and technology* (New York, NY, USA, 2004), ACM, pp. 10–15.

[Eri05] ERICSON C.: *Real-Time Collision Detection*. Morgan Kaufmann, 2005.

[Erl07] ERLEBEN K.: Velocity-based shock propagation for multibody dynamics animation. *ACM Trans. Graph. 26*, 2 (2007), 12.

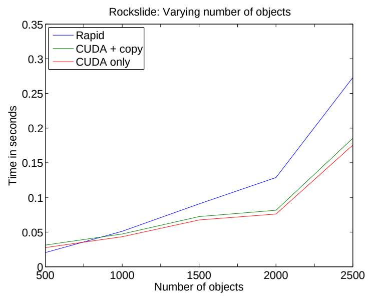
[ESHD05] ERLEBEN K., SPORRING J., HENRIKSEN K., DOHLMANN H.: *Physics-Based Animation*. Charles River Media, 2005.

[FWG04] FAN Z., WAN H., GAO S.: Simple and rapid collision detection using multiple viewing volumes. In *VRCAI '04: Proceedings of the 2004 ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry* (New York, NY, USA, 2004), ACM, pp. 95–99.

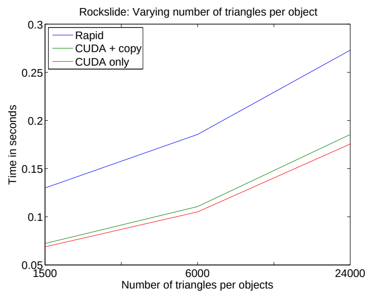
[gim09] Gimpact. web, July 2009. Open source project, http://gimpact.sourceforge.net/.

[GKJ*05] GOVINDARAJU N. K., KNOTT D., JAIN N., KABUL I., TAMSTORF R., GAYLE R., LIN M. C., MANOCHA D.: Interactive collision detection between deformable models using chromatic decomposition. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers* (New York, NY, USA, 2005), ACM, pp. 991–999.

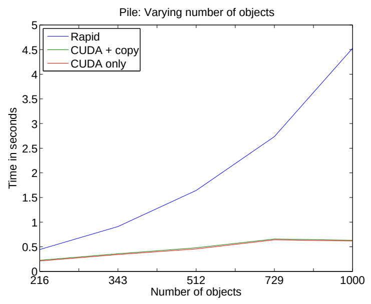
[GLM96] GOTTSCHALK S., LIN M. C., MANOCHA D.: Obbtree: a hierarchical structure for rapid interference detection. In



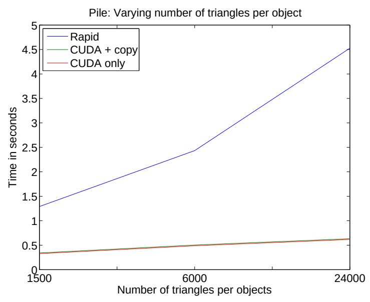
(a)



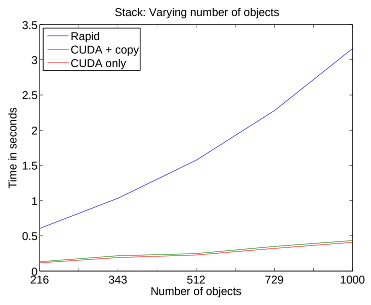
(b)



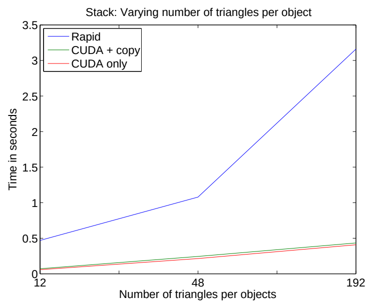
(c)



(d)



(e)



(f)

Figure 8: *Selected timings for the three configuration types. The plots in the left column are for the test cases using the largest number of subdivisions and the plots in the right column are for the test cases with the largest amount of objects.*

SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques (New York, NY, USA, 1996), ACM, pp. 171–180.

[GLM05a] GOVINDARAJU N. K., LIN M. C., MANOCHA D.: Fast and reliable collision detection using graphics processors. In *SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry* (New York, NY, USA, 2005), ACM, pp. 384–385.

[GLM05b] GOVINDARAJU N. K., LIN M. C., MANOCHA D.: Quick-cullide: Fast inter- and intra-object collision culling using graphics hardware. In *VR '05: Proceedings of the 2005 IEEE Conference 2005 on Virtual Reality* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 59–66, 319.

[Got00] GOTTSCHALK S.: *Collision Queries using Oriented Bounding Boxes*. PhD thesis, Department of Computer Science, University of N. Carolina, Chapel Hill, 2000.

[Gra07] GRAND S. L.: *GPU Gems 3*. Addison-Wesley Professional, 2007, ch. 32: Broad-Phase Collision Detection with CUDA.

[GRLM03] GOVINDARAJU N. K., REDON S., LIN M. C., MANOCHA D.: Cullide: interactive collision detection between complex models in large environments using graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 25–32.

[Gro09] GROUP K.: Open computing language. web, July 2009. http://www.khronos.org/opencl/.

[GZ03] GRESS A., ZACHMANN G.: Object-space interference detection on programmable graphics hardware. In *SIAM Conf. on Geometric Design and Computing* (Seattle, Washington, Nov.13–17 2003), Lucian M. L., Neamtu M., (Eds.), Nashboro Press, pp. 311–328.

[Har07] HARADA T.: *GPU Gems 3*. Addison-Wesley Professional, 2007, ch. 29: Real-Time Rigid Body Simulation on GPUs.

[Har09] HARADA T.: Parallizing the physics pipeline, physics simulations on the gpu. web, March 2009. `http://www.gdconf.com/`.

[Hav09] HAVOKFX: Physics simulation on nvidia gpus. web, July 2009. `http://www.havok.com`.

[HTG04] HEIDELBERGER B., TESCHNER M., GROSS M.: Detection of collisions and self-collisions using image-space techniques. In *Twelfth International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (Winter School on Computer Graphics)* (February 2004), pp. 145–152.

[Hub95] HUBBARD P. M.: Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics 1*, 3 (1995), 218–230.

[JH08] JANG H., HAN J.: Fast collision detection using the a-buffer. *The Visual Computer 24*, 7 (2008), 659–667.

[JP04] JAMES D. L., PAI D. K.: Bd-tree: output-sensitive collision detection for reduced deformable models. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), ACM, pp. 393–398.

[KHM*98] KLOSOWSKI J. T., HELD M., MITCHELL J. S. B., SOWIZRAL H., ZIKAN K.: Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics 4*, 1 (1998), 21–36.

[KOLM02] KIM Y. J., OTADUY M. A., LIN M. C., MANOCHA D.: Fast penetration depth computation for physically-based animation. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation* (New York, NY, USA, 2002), ACM, pp. 23–31.

[KP03] KNOTT D., PAI D. K.: CInDeR collision and interference detection in real-time using graphics hardware. In *Proceedings of Graphics Interface* (2003), pp. 73–80.

[NVI08] NVIDIA: *NVIDIA CUDA Programming Guide*, version 2.0 ed., 2008. `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`.

[NVI09] NVIDIA: Physx. web, July 2009. `http://www.nvidia.com/object/nvidia_physx.html`.

[ode09] Open dynamics engine. web, July 2009. Open source project, `http://www.ode.org`.

[RBAZ05] RAABE A., BARTYZEL B., ANLAUF J. K., ZACHMANN G.: Hardware accelerated collision detection - an architecture and simulation results. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 130–135.

[RZ08] RAABE A., ZAVELBERG F.: Defying the Memory Bottleneck in Hardware Accelerated Collision Detection. In *WSCG '2008* (University of West Bohemia, Plzen, Czech Republic, 2008).

[TCYM08] TANG M., CURTIS S., YOON S.-E., MANOCHA D.: Interactive continuous collision detection between deformable models using connectivity-based culling. In *SPM '08: Proceedings of the 2008 ACM symposium on Solid and physical modeling* (New York, NY, USA, 2008), ACM, pp. 25–36.

[TS05] THRANE N., SIMONSEN L. O.: *A Comparison of Accelerating Structures for GPU Assisted Ray Tracing*. Master's thesis, Department of Computer Science, University of Aarhus, 2005.

[vdB03] VAN DEN BERGEN G.: *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann, 2003.

[WB06] WONG W. S.-K., BACIU G.: A randomized marking scheme for continuous collision detection in simulation of deformable surfaces. In *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications* (New York, NY, USA, 2006), ACM, pp. 181–188.

[ZH07] ZINK N., HARDY A.: Cloth simulation and collision detection using geometry images. In *AFRIGRAPH '07: Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa* (New York, NY, USA, 2007), ACM, pp. 187–195.