EUROGRAPHICS '0x / N.N. and N.N.
(Editors)

Volume 0 (1981), Number 0

gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries

C. Lauterbach and Q. Mo and D. Manocha

University of North Carolina at Chapel Hill

Abstract

We present novel parallel algorithms for collision detection and separation distance computation for rigid and deformable models that exploit the computational capabilities of many-core GPUs. Our approach uses thread and data parallelism to perform fast hierarchy construction, updating, and traversal using tight-fitting bounding volumes such as oriented bounding boxes (OBB) and rectangular swept spheres (RSS). We also describe efficient algorithms to compute a linear bounding volume hierarchy (LBVH) and update them using refitting methods. Moreover, we show that tight-fitting bounding volume hierarchies offer improved performance on GPU-like throughput architectures. We use our algorithms to perform discrete and continuous collision detection including self-collisions, as well as separation distance computation between non-overlapping models. In practice, our approach (gProximity) can perform these queries in a few milliseconds on a PC with NVIDIA GTX 285 card on models composed of tens or hundreds of thousands of triangles used in cloth simulation, surgical simulation, virtual prototyping and N-body simulation. Moreover, we observe more than an order of magnitude performance improvement over prior GPU-based algorithms.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric Algorithms, languages, and systems I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Object hierarchies

1. Introduction

Geometric proximity queries including collision detection and separation distance computation are widely used in computer graphics, physically-based modeling, virtual reality, haptics and robotics. In order to perform accurate simulations, it is important to ensure that there are no collisions or penetrations between the objects or primitives. Moreover, distance queries are used to estimate the time of collision and the response force in dynamics simulation. Many applications, such as haptic rendering, need to perform these queries at interactive rates (e.g. 1 KHz).

There is extensive literature on proximity queries. Some of the widely used algorithms use bounding volume hierarchies (BVHs) to accelerate different queries. These algorithms can be classified based on the choice of the bounding volumes. Furthermore, these hierarchies can be reconstructed or updated using incremental methods for deformable models.

In this paper, we address the problem of exploiting multiple cores on commodity GPUs or many-core processors for fast proximity queries. Many algorithms have been proposed to perform fast collision queries on current GPUs using depth or stencil buffer tests [HTG03, KP03, GRLM03] or compute distance fields by rendering the distance functions [SOM04, SGG*06, SGGM06, MRS08]. Essentially, these algorithms exploit the parallel rasterization capabilities of a GPU and the total work performed is relatively high as compared to CPU-based algorithms that use BVHs. Furthermore, many prior GPU-based approaches perform these queries at a discrete or image-space resolution, which may not provide sufficient accuracy for physically-based simulation.

Main Results: We present novel GPU-based algorithms to efficiently perform collision and separation distance queries between rigid and deformable models. We treat GPUs as throughput many-core processors that can support thousands of concurrent threads and use them for parallel BVH hierarchy construction, hierarchy update and hierarchy traversal. Our formulation takes into account some of the architec-

tural characteristics of current GPUs and is based on explicit balancing of work units coupled with very lightweight synchronization between the cores (see Section 2). This makes it possible to apply our approach to very fine-grained workloads such as overlap tests in collision and distance queries (see Section 3).

We show that throughput GPU-architectures can be exploited to use tighter fitting bounding volumes, such as oriented bounding boxes (OBBs) and rectangular swept spheres (RSS) with a very small overhead. The higher culling efficiency of these tight fitting bounding volumes results in fewer false positives in terms of primitive or elementary tests. This is in contrast to the development in CPU-based algorithms that tend to use simple bounding volumes such as AABBs for collision and distance computations. In order to improve the performance of distance queries, we also present efficient methods to compute linear bounding volume hierarchies (LBVH) of RSS volumes. The parallel traversal of the LBVHs is improved by using a shallower hierarchy with higher branching factor, as opposed to a binary tree (see Section 4). The shallow BVH reduces the data dependency between different steps of the hierarchical distance computation and improves the overall performance. Moreover, our hierarchy update and refitting algorithms are an order of magnitude faster than previous CPU implementations. We have implemented our approach on a PC with NVIDIA GTX 285 GPU and evaluated its on several benchmarks with 40K to 250K triangles used in cloth simulation, virtual prototyping, surgical simulation, and N-body simulation (see Section 5). We observe up to an order of magnitude performance improvement over prior GPU-based algorithms that perform these queries at object-space resolution (see Section 6). We also compare the performance of our algorithm with prior CPU-based single-core and multi-core algorithms that use BVHs and highlight the speedups.

2. Background

We first briefly summarize previous work on collision and distance queries. Next, we provide some background on current GPU architectures.

2.1. Collision detection and distance queries

Hierarchical techniques based on BVHs have been widely used to accelerate collision and distance queries. These include sphere trees, AABB trees, OBBTrees, K-DOP trees, etc. [Eri04]. Most recent improvements to these queries has either come from improved culling techniques [CTM08, TCYM08] or parallelism [KHeY08]. In terms of separation distance queries, different algorithms based on bounding volume hierarchies have been proposed [vdB97, Qui94, LGLM00, JC04]. However, there is less work on using hierarchical collision on GPUs. Instead, one approach has been to use the GPU for broad-phase collision only [LG07]. Alternatively, many GPU-based collision checking algorithms exploit the rasterization capabilities of GPUs by using depth or stencil buffer tests at image-space resolution [HTG03, KP03, GRLM03]. In order to handle complex models, the hierarchical traversals are performed on the CPUs [GKJ*05, SGG*06] and this results in additional CPU-GPU data transfer overhead. A similar approach combines both multi-core CPUs and GPUs [KHH*09]. The rasterization capabilities of GPUs have been used for fast distance field computation [SOM04, MRS08] and they can also be used to compute separation or penetration distances [SGG*06].

2.2. GPU architectures

In recent years, the focus in processor architectures has shifted from increasing clock rate to increasing parallelism. Some of the most successful instances of throughput architectures are GPUs, which have become very general processors with a strong focus on achieving performance through high parallelism. Current high-end GPUs have a theoretical peak speed of up to a few Tera-FLOPs, thus far outpacing current CPU architectures. Specifically, there are several features that distinguish GPU architectures from the multi-core CPU systems and also make it harder to achieve peak performance. First, the GPUs usually have a high number of independent cores (e.g. the current generation NVIDIA GTX 280 has 30 cores) and each of the individual cores is a vector processor capable of performing the same operation on several data elements simultaneously (e.g. 32 or 64 in current GPUs). Second, GPUs do not provide a general cache hierarchy for all memory accesses like CPUs. Instead, each core can handle several separate tasks in parallel and switch between them in hardware when one of them is waiting for a memory operation to complete. This hardware multithreading approach is thus designed to hide the latency of memory accesses to perform some other work in the meantime. However, both of these characteristics imply that – unlike CPUs – achieving high performance in a GPU-based algorithms depends on two things: (1) Providing a sufficient number of parallel tasks so that all the cores are utilized; (2) Providing several times that number of tasks just so that each core has enough work to perform while waiting for data from relatively slow memory accesses.

There is considerable literature in parallel computing on the use of work queues for load balancing, including locking and non-locking shared queues such as work stealing approaches [ABP98]. These techniques map very well to hierarchical and recursive operations and have been employed extensively in parallel systems. However, they have not been used on GPUs as the overhead of performing communication between the cores through main memory can be rather high. Previous techniques for GPU work queues used explicit compaction methods between kernel calls [ZHWG08, LGS*09]. Other methods have explored lock-free queues and work stealing to parallelize octree construction [CT08]

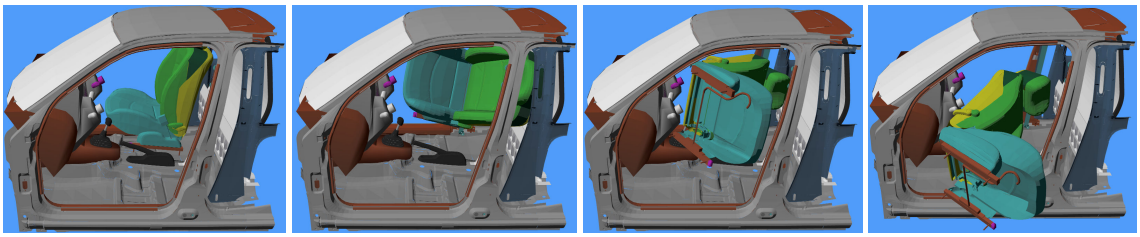

Figure 1: ***CAD/CAM benchmark:*** *This sequence shows the collision-free path computed by a planner for simulating the removal of a car seat for virtual prototyping [ZHKM08]. The underlying planner performs distance queries repeatedly to compute such a path. Both objects are in close proximity configuration, which is a challenging scenario for separation distance computation. The models consist of 245k triangles with an average query time of 55ms using our approach. It is more than 12 times faster than prior CPU-based algorithms.*

Figure 2: ***Surgical Simulation:*** *Simulating the motion of a deformable catheter in the arteries for surgical simulation [MLM08]. The catheter deforms as it passes through the network of arteries and we use distance queries to perform the simulation. The model has around 12K triangles and the average distance query takes about 28 ms.*

and found work stealing to perform best. Overall, the overhead of these methods makes them efficient only for applications with relatively high computational intensity and coarse-grained parallelism such as hierarchy construction. There are known parallel algorithms to accelerate hierarchical traversals [RK87, KG94] and they are also applied to parallel collision detection [KSTK95, GW07]. Parallel simulations were also investigated in [HGS*07]. However, most of these methods either perform fine-grained communication between the processors or are not suited for GPU-like architectures.

3. Hierarchy operations on GPU architectures

In this section, we present an overview of our approach for performing hierarchical operations on GPUs.

3.1. Bounding Volume Hierarchies

BVHs have been widely used for accelerating collision and distance queries. These include discrete as well as continuous collision detection, including self-collisions. These hierarchies can be characterized by the choice of the BV. The simplest hierarchies use simple BVs such as spheres or AABBs, which have a lower storage overhead, lower cost for updating the hierarchy and performing overlap tests. Other BVHs use tight fitting BVs such as K-DOPs, OBBs and RSS. These BVHs have a higher storage overhead and increased cost of overlap test. However, their culling efficiency is much higher than the BVHs based on simple BVs, i.e. they result in fewer false positives. In case of rigid models, the BVHs are computed once and are traversed at run-time to resolve the query. In this case, some of the fastest collision algorithms use tight filling BVs such as K-DOPs [KHM*98] and OBBs [GLM96] and the fastest separation distance computation algorithms are based on RSS [LGLM00]. However, for deformable models, the cost of reconstructing or updating the hierarchy at each step of the simulation can be high. Therefore, most CPU-based algorithms for deformable models use AABB or sphere hierarchies (e.g. [CTM08].) However, these simple BVs can result in a high number of false positives and the resulting algorithms perform a high number of elementary tests [CTM08].

3.2. Challenges

In case of hierarchical collision detection and distance queries, the major challenge is that work is generated dynamically as the algorithm progresses and the computational load on different cores can change significantly. Thus, any parallel hierarchy-based algorithm needs to address the problem of load balancing and work distribution in order to maintain availability of parallelism for all cores. On multi-threaded CPU architectures, prior approaches have used work queues and work stealing for operations with hierarchies and recursion with similar properties [ABP98]. However, these techniques do not currently work well on GPUs for multiple reasons. Primarily, they are based on the assumption that low-latency communication between cores is possible in order to manage concurrent access to shared data structures. Unfortunately, this is only possible in a very restricted sense on current GPUs. The main barrier to communication is the latency and lack of a memory consistency model in the global GPU memory shared by the cores, i.e. different cores are not guaranteed to see memory writes from other cores or may not even see them in the same order they were written. Even though newer GPU architectures provide atomic operations such as compare-and-swap (CAS) that could be used for locking operations, the remaining problem is that previous writes to the memory protected by the

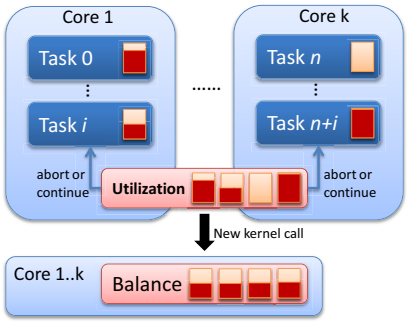

Figure 3: ***Load balancing for hierarchy computation and traversal:*** *In our approach, each task keeps its own local work queue in local memory and can generate new work units (such as intersection operation) without coordinating with others. After processing a work unit, each task is either able to run further or has an empty or completely full work queue and wants to abort.*

lock may not have been executed yet, thus preventing implementation of work queues or other structures shared by all cores. Even if memory consistency was not a problem, busy waiting such as by spinning on a lock variable is relatively inefficient on an architecture with high memory latency and hardware multi-threaded execution can also lead to priority inversion and prevent other threads on the same core from performing useful work. As a result, one of the major challenges in terms of hierarchical traversal is to balance the load evenly among multiple cores on the GPUs.

3.3. Lightweight work balancing

In the context of this paper, hierarchy operations include workloads such as testing a pair of nodes for intersection or computing a separation distance. As a result of that test, new pairs of nodes may have to be tested afterward. In the further discussion, we refer to the specific test to be done (e.g. references to two nodes) as a *work unit*, the code run to perform tests as a *kernel* and an instance of the kernel executed on a core as a *task*. In order to efficiently parallelize the hierarchy operation, we describe a novel approach that distributes these work units between GPU cores and threads efficiently. The main goal of our technique is to minimize the amount of synchronization overhead, while performing actual work. In our approach, we launch a number of parallel tasks that run on separate cores. Every task keeps its own work queue either in the processor-local memory or global memory, depending on its size constraints. These queues are only accessible locally; thus, no synchronization is necessary between cores when reading or writing to queues. Work kernels can remove an element from the queue and then create new ones as well. In order to use the vector processors, we also provide implementations for data-parallel access to the queues such that the kernel may work on multiple work units in parallel. For example, on a 32-wide vector unit, at each step up to 32 work elements are dequeued, processed and then some number of new units is pushed back onto the queue. To perform parallel queue operations, there are two options: first, data parallel prefix sum operations can provide a way to compute an offset to store each new element in the queue. Second, some GPUs also provide atomic operations on memory and thus it is also possible to just atomically increase a queue pointer to store elements. We have found the latter option to be faster on current hardware.

The main step that ensures that all cores have work is the balancing step (also see Fig. 3). Synchronization is only performed on one global counter that holds the number of cores that cannot do further work (i.e. the queue is empty or full). Whenever a task reaches that state, it atomically increases the variable and terminates the kernel. After executing a work unit, each task tests the current value of the variable and compares it against a user-specified threshold of idle tasks. If higher, then it terminates and writes back its local queue to the global memory; otherwise, it continues. After all cores have either finished or aborted, we run a kernel that examines the work queues from all the cores, then assigns a roughly equal number of work units to each. If there are unprocessed work units left, then the work kernel for the current hierarchy operation is called again and the process repeats. Note that the number of actual tasks is in fact larger than the total number of GPU cores, to allow for hardware multi-threading. Therefore, even if some percentage of tasks has been stopped, the core it was scheduled on may not be idle but just processing another task. In practice, we have found a threshold of 50% idle tasks for balancing works well on current GPU architectures.

4. Collision and distance queries

In this section, we present our novel parallel algorithms to perform collision and distance queries.

4.1. Hierarchy traversal for collision detection

We use BVHs to check for collisions between two disjoint objects (inter-object collisions) as well as self-collisions for deformable objects (intra-object collisions). We assume that each object is composed of triangles and we do not make any assumptions about their connectivity. As a special case, self-intersection involves checking whether any of the non-adjacent triangles of an object intersect each other, as is needed in cloth simulation or surgical simulation. Since many triangles can be small with large aspect ratios, overlaps can be missed if discrete collision checking is used. Therefore, *continuous* collision detection algorithms are used to check whether there is a collision between the discrete time instances, and is more expensive than discrete collision checking as the elementary tests involve finding roots of cubic polynomials. The BVH is built on top of each object's triangle or the swept volume between the

time instances. During each step of the simulation, we use the hierarchies to compute the potentially colliding pairs and only perform triangle-triangle overlap tests on those candidate pairs.

4.1.1. Simultaneous hierarchy traversal

The traversal algorithm starts with the two BVH root nodes and tests the BV for overlap in a recursive manner. If the BVs overlap, then all possible pairings of their children are recursively tested for intersection. If both of the nodes are leafs, then the two corresponding triangles are tested for exact overlap. If only one of the two nodes is a leaf, then it is tested against the children of the other node. This can be seen as traversing a tree of possible bounding volume node pairs, also called the bounding volume test tree [LGLM00] (BVTT.) Implicitly, our algorithm is performing a parallel traversal of a BVTT.

The main work units are pairs of BVH nodes and for a binary tree each intersection test can generate up to four new pairs. All intersection tests between the node of the hierarchy can be performed independently. The intersection kernel can be run using the vector units to process several intersections in parallel and push the resulting new intersection pairs on the work queue or in a separate result queue for actual triangle pairs. After this traversal, the overall list of triangle pairs is then used as input for an intersection test kernel that tests for actual overlap. Because all potential intersections can be performed in parallel, this step is simple to implement by starting enough tasks for all the pairs.

4.1.2. BVTT traversal

One problem with this approach is that there is a lack of available parallelism while testing the higher levels of the hierarchy that can reduce the overall performance of the algorithm. However, it is possible to exploit temporal coherence in the traversal and drastically increase the level of parallelism. In many interactive applications there is considerable temporal or spatial coherence between successive time steps. This can be exploited for front tracking [KHM*98, EL01]. We keep track of the front in the BVTT which consists of all the intersecting leaf node pairs of the BVTT as well as every non-intersecting node pair for which a sibling overlaps. This simple list of node pairs can be generated during BVH traversal. For the next frame, we use this list as input for the kernel and use the same pairs as the starting work units. The traversal kernel for processing this front is a modified version of the standard traversal algorithm: for each initial node, the intersection test is performed again with the updated BVs. If the pair had an overlap during the last frame and intersects again, then the triangle pair is written to the intersection queue. If the pair did not intersect last frame, but does now, then new work units are created as in the normal traversal. Finally, if the pair does not intersect, then the kernel loads the parents of both the nodes and tests them for overlap. If they still intersect, then we have at least one sibling that must still overlap and the node pair is kept in the front. Otherwise, the front is moved upwards by adding the pair of parent nodes to the work queue for the next step. However, since all the siblings belong to the front by definition, the parent pair would be added to the list multiple times, which need to be avoided. Instead, we check whether the pair is the leftmost child in the BVTT and only add the parent pair if that is the case, and thereby avoid duplicates.

4.2. Hierarchy traversal for distance queries

The distance computation algorithms also traverse the BVHs but in a different manner than a collision query. When traversing the BVH for a distance query between a pair of objects, the distance between the two root nodes of the respective BVHs is computed and stored as the initial value of the minimum distance. Based on this minimum distance value, which is updated throughout the traversal, a decision is made to descend one or both of the BVHs if the distance between the nodes at the higher levels is found to be smaller than the stored minimum distance. In this case the global bound on the minimum distance is updated. If, on the contrary, higher level nodes are found to be farther apart than the current minimum distance bound, there is no need to descend further down to those nodes. Unlike collision detection, it is not possible to prune all the branches at any given level of the hierarchy since there will be at least one path for which the traversal needs to go down to the leaf level of each BVH, and the final result of the minimum distance is computed based on the triangle pairs at the leaf level. Moreover, the decision to descend further down the hierarchy depends on the minimum value among all the previous distance computation results, which makes the algorithm highly dependent on the relative configuration of the objects between the steps. Our traversal algorithm addresses those two issues and makes it amenable for parallel computation.

In order to increase the level of parallelism during hierarchy traversal, we replace the binary BVH with a tree with a larger branching factor. In this case, each node within the hierarchy has more than two children. These hierarchies offer some additional benefits in terms of parallel tree traversal. Our first observation is that there is no early exit that enables skipping of the lower level nodes in the hierarchy during the distance query traversal. During each parallel traversal step the algorithm will descend at most one level. As a result, it helps to have a shallower hierarchy rather than a deep hierarchy so that traversing to the bottom level will have a shorter path.

Another observation is that the distance queries require maintaining a minimum distance value computed so far during the traversal. This is used to avoid descending paths whose parent nodes are farther apart than the current minimum distance. The only way to keep this minimum distance

value most up-to-date is to either update it atomically whenever a distance is computed between a pair of nodes, or to periodically examine all the distances computed since the last query and compute the minimum. The first method can be very expensive on GPU-like parallel architecture, and will end up serializing the operations in the worst case. The second method requires periodic synchronization followed by a reduction, so in order for it to work efficiently we need to have enough computation load between the synchronization points and these computations should be useful work towards the final result. By having larger branching factors within the hierarchy more pair-wise distance computations can be carried out in one parallel step, before the results are compared to the stored minimum distance all at once and the stored value gets updated. In this manner we also maximize the pruning efficiency at each step of the traversal by having many pairs of nodes evaluated at each level and obly preserving the closest pairs. One thing to note is that by using branching factor larger than two seems to be contradicting previous approaches that used binary trees to get the best distance query results. This can be explained by the fact that most work on BVHs has been on collision detection, object intersection, or other similar scenarios that have the possibility of performing an early exit in the hierarchy. However, the separation distance queries do not share this property. If we have higher memory resources on the parallel architecture, we may even use higher branching factors. However, on current GPUs where the number of simultaneous execution threads is limited by shared memory resources, the choice of branching factors becomes a trade-off between the memory cost and the pruning efficiency benefit brought by wide branches. There exists a theoretical optimal setting based on the parameters of a parallel architecture, and we report empirical results that 8-way trees give us best performance after experimenting with a range of branching factors in our implementation.

4.3. Hierarchy construction and refitting

For both distance and collision queries we do not only need an efficient way to traverse the hierarchy, but we also need to construct or refit the hierarchy as the model undergoes deformation. Traditional construction methods use a divide-and-conquer, top-down approach where a set of objects is recursively split into smaller groups. In order to provide maximum culling for our applications, this is performed until the leaf nodes in the hierarchy refer to only one object. Our hierarchy construction is built on a parallel algorithm for GPUs described in [LGS*09]. However, that algorithm was designed to build a hierarchy of axis-aligned bounding boxes (AABBs) for ray tracing, and we use tighter-fitting boxes for collision and distance tests.

We separate the construction algorithm into two phases, a splitting phase followed by a fitting phase. In the splitting phase all the underlying primitives are grouped into nodes that are placed in the hierarchy. The bounding volumes that are associated with each node, however, are not constructed in this phase. We use the fast and approximate LBVH construction method from [LGS*09] to perform the grouping and organize the nodes in the splitting phase. After the hierarchy structure is computed in terms of all the splits, we fit our BVs around the primitives within each node to complete the hierarchy construction. Because the nodes are already organized in the splitting phase, the fitting phase can be performed across all the nodes in parallel. Note that the LBVH algorithm is highly parallelizable, since the main computation is performed using an efficient radix sort.

Refitting of bounding boxes is performed by using a bottom-up traversal of the tree. Starting at the leafs, the bounding boxes are computed from the actual geometric primitives, and that information is propagated upwards while merging the volumes from all the node's children. To perform this computation in parallel, we store the tree in an array arranged by the levels, which can be obtained directly from the LBVH construction. Next, we process all the nodes at each level of the tree in parallel across all the GPU cores and on each vector unit. We call the refitting kernel once per level, but it would also be possible to use just one kernel call using a global barrier between the levels and ensure memory consistency.

4.4. Tight fitting bounding volumes

The tight fitting bounding volumes have been shown to provide much higher culling efficiency during distance and collision queries [GLM96, KHM*98, LGLM00]. However, most recent CPU-based algorithms for deformable models tend to use simple BVs such as AABBs because of compact storage and lower cost of refitting or hierarchy computation. However, this trade-off may not hold on GPU architectures, due much higher ratio of computational power to memory latency.

In order to perform collision queries, we use OBBs (and compare against AABBs) in our implementation. We use the fitting method based on principal component analysis as well as the separating axis overlap test [GLM96]. OBBs are merged in the refitting process. Even though these operations involve using about $1-2$ orders of magnitude more instructions than AABB tree updates, the overall parallel computation and higher culling efficiency of OBBs results in improved overall performance.

For distance queries we adopt rectangular swept spheres (RSS) as the building block of our BVHs [LGLM00]. RSS volumes are constructed and represented by taking the Minkowski sum of a sphere with certain radius and a arbitrarily oriented rectangle. Similar to OBBs, RSS are able to bound the underlying primitives more tightly than simpler BVs such as spheres or AABBs. In addition, RSS has the superior property of having an elegant and robust way of

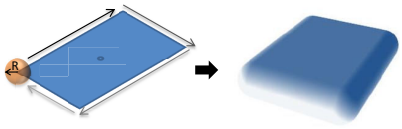

Figure 4: *Rectangular swept spheres (RSS) are constructed by augmenting an oriented box with a sphere around the borders.*

Model	Tris	Build		Refit	
Collision		AABB	OBB	AABB	OBB
Flamenco	49k	22	27	1.73	4.6
Princess	40k	20	24	1.4	3.9
Sphere/Cloth	92k	31	39	2.5	7.9
Balls	146k	56	68	3.2	11.5

Figure 6: ***Parallel hierarchy computations:*** *The benchmark scenes used in this paper, and timings (in ms) for our construction and refitting algorithm both for AABB and OBB hierarchies. Note that the overhead for creating and refitting a hierarchy with more complex bounding volumes is relatively low.*

computing pair-wise distance by calculating the distance between the embedded rectangles and then subtracting the sum of the sphere radius, as illustrated in Fig. 4.

The LBVH algorithm was originally designed for AABBs [LGS*09]. We found it more useful for a RSS hierarchy to apply one transformation step before using LBVH: In the original LBVH method, the Morton code is derived directly from the primitives coordinates in the world space, while for RSSs we first fit an RSS around all the primitives to find out the optimal orientation of the coordinates, and then the Morton code is computed from primitive positions transformed into this oriented space. This way the hierarchy construction start from a closer-to-optimal orientation and conforms better to the set of underlying primitives. As compared to AABBs, it is much easier to apply rigid transformation on RSSs, therefore the constructed hierarchy does not need to be updated unless the objects deform.

5. Results and analysis

We have implemented our approach using a Intel Core2 Duo system at 2.83 GHz on 4 cores. We use CUDA on a NVIDIA GTX 285 GPU that has a total of 30 processing cores and 1 GB of memory. We use a standard algorithm for discrete triangle-triangle intersection and solve the cubic equation [Pro97] for each of the 15 elementary vertex/face and edge/edge tests to compute the first time of contact for continuous triangles.

We use several commonly benchmark scenes for collision and distance queries and compare their performance with prior methods (see Fig. 5, Fig.1, and Fig.2). These models range from 12k to 245k triangles each and can have multi-

Model	Discrete		Continuous	
	AABB	OBB	AABB	OBB
Flamenco	27	22	37	34
Princess	17	22	26	29
Sphere/Cloth	28	36	42	38
Balls	78	70	91	74

Figure 7: ***Performance results:*** *This table highlights the performance of our collision detection algorithm, which checks for inter-object and intra-object collisions. All the numbers are in milli-sec. and include the time for refitting, front-based traversal and pairwise triangle intersections.*

Model	Triangles	Distance	Refitting	Construction
Catheter	12k	40	3	30
Letters	5.5k	25	2	25
Car/Seat	245k	55	7	164

Figure 8: ***Distance query results:*** *Results for our distance query algorithm. All numbers in milliseconds.*

ple triangle pairs in close proximity. For example, the Flamenco model has several cloth layers very close together, representing a hard case for culling for collision detection algorithms. Figure 6 summarizes the results for our parallel GPU construction and refitting algorithms. We also compare the timings for building a AABB BVH to show the overhead of OBBs. Since refitting cost is about an order of magnitude lower than construction, it is a good choice for handling deformable models and animation sequences with no topological changes.

The collision detection algorithm's performance is summarized in Fig. 7. We provide results for discrete as well as continuous versions using either AABB or OBB bounding volumes. In addition, we show the impact of exploiting temporal coherence by using our front-based traversal implementation of the same algorithms. All the numbers are averaged over the whole animation. Note that AABBs are slightly faster on most benchmarks for discrete collision detection where pairwise intersection tests are relatively cheap. However, for continuous collisions where pairwise tests are far more expensive, OBBs provide better performance despite their higher cost for maintenance and traversal. The front-based traversal generally results in a speedup compared to full traversal, although the result is model-dependent.

The performance of distance query is listed in Fig. 8. We also list the time needed for refitting (between each frame) and initial RSS construction (first frame only). For all queries, we used a tree with branching factor 8 which proved the fastest for our tests. Note that on the smaller models our algorithm is noticeably slower. We ascribe this to the fact that for only a few thousand triangles it is unlikely that the traversal for separation distance can provide sufficient parallel work units to come close to utilizing all GPU cores. On the Car/Seat benchmark, the complexity of the query is much higher and many more paths in the hierarchy can be evaluated in parallel.

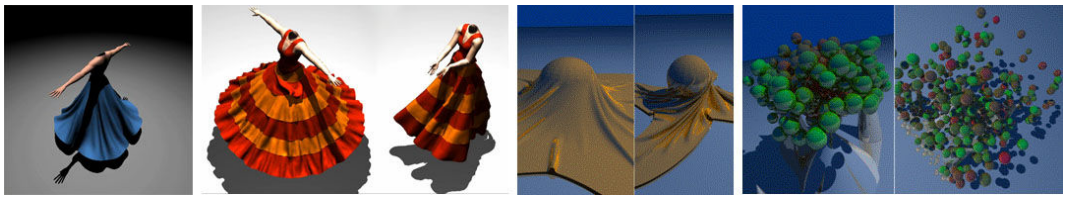

Figure 5: ***Benchmarks:*** *The benchmark models used for collision detection in this order: Princess cloth simulation (40K triangles); Flamenco cloth simulation (49K triangles), Cloth dropping on sphere (92K) and n-body simulation (146K). Our algorithm can perform interactive continuous self-collisions using OBB hierarchies and pairwise elementary tests on all of these models in tens of milliseconds, almost* $7-12$ *faster than prior methods.*

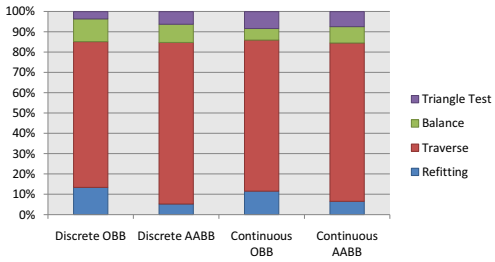

Figure 9: ***Split-up of timings:*** *The fraction of time spent in the parts of the algorithm differ based on whether continuous or discrete collision detection is performed and the choice of the BV. In general, the use of OBBs results in more time spent in refitting and traversing, but less in intersection tests due to higher culling efficiency.*

We also highlight a breakdown of timings within the collision detection algorithm, i.e. refitting, traversing, balancing and triangle intersection (see Fig. 9), in the Flamenco model. The results show that a large part of the time is spent in the traversal part while intersection tests account for a smaller percentage, mostly due to the fact that it runs with optimal parallel utilization given that all intersections can be performed independently. Refitting takes a relatively constant fraction of overall time. For distance queries, only a few pairwise distance queries are performed and thus almost all of the time is spent in the hierarchy traversal.

6. Analysis and comparison

6.1. Analysis

The performance results show interesting implications for the choice of BVs for collision detection on GPUs. Current CPU approaches use AABBs since they provide very fast intersection and refit operations and are relatively compact in memory. Our results show that for discrete collision detection AABBs can provide improved performance over OBBs in many cases. However, for continuous collision detection this situation is reversed and OBBs provide faster performance. We have found that OBB intersection and refitting benefits from having much higher *compute density* that is a good match to the high computational power of GPUs. In particular, OBBs use 2.5 times the memory of AABBs, but operations such as computing an OBB from triangles or intersecting two OBBs take about two orders of magnitude more instructions. For example, AABB-AABB intersection needs just 6 comparison operations and needs to load 12 coordinates to do so. In contrast, OBB-OBB intersection test loads 30 coordinates, but then needs to perform 15 separating axis tests, resulting in hundreds of operations. This will be slower on a low-latency CPU architecture with less compute power, but fast on a GPU architecture where high-latency memory accesses are expensive and computational power is high. Further, these separating axis tests can be performed in parallel. Similarly, RSS bounding volumes for distance queries also have more computationally intensive operations and thus work well on a GPU architecture.

Limitations: Our approach has some limitations. Firstly, getting high performance or speedup for collision detection and distance queries depends on having a relatively large front in the BVTT traversal tree. Unlike self-collision, inter-object collision between two different hierarchies, e.g. deformable models, may exhibit a much smaller front unless the objects are very close such that there are many BVs overlaps. For very small models, available parallelism may be limited. In this case, handling multiple object queries in parallel will be a better solution to exploit the capabilities of a GPU. In general, even though our approach tries to implement very lightweight synchronization, we are still limited by inherent memory latency of GPUs for communication. To achieve better scaling, our approach could benefit from having a low-latency communication channel between the cores on GPU architectures. This would allow the implementation of algorithms such as work stealing during kernel execution, which based on our experiments can already be implemented on current GPU architectures, but are very slow. Future GPU architectures that have coherent caching could also improve performance significantly.

6.2. Comparison

Other approaches for work distributions on GPUs have explored lock-free work queues and work stealing approaches [CT08]. We have found that these methods work well as long

Model	Balancing		Work stealing	
	AABB	OBB	AABB	OBB
Flamenco	40	38	124	81
Princess	27	34	105	116
Sphere/Cloth	47	43	208	124
Balls	99	81	337	178

Figure 10: ***Balancing vs. work stealing:*** *Results for continuous collision detection both for our work balancing and ABP work stealing [ABP98] such as used in previous work [CT08]. Although more efficient than global queues, the overhead from memory latency still dominates the traversal operation in work stealing. Note that for the more complex models OBBs provide much higher performance due to the reduction in traversal steps.*

as parallel tasks are relatively heavy-weight, such as the octree construction in [CT08], and our results for BVH construction are that work stealing provides roughly equivalent or slightly faster execution time compared to work balancing. However, for our collision and distance queries each operation is far more fine-grained and even efficient work queue methods have overhead that dominates the overall computation as shown in Fig. 10.

Collision detection: Some of the fastest CPU-based algorithms for continuous self-collision use feature-based hierarchies and other culling methods (e.g. normal cone tests) to reduce the number of pairwise intersection tests. As an example, the representative triangle algorithm [CTM08] takes about 200ms per frame on a single core for continuous self-collision detection on the Flamenco model, as compared to our algorithm that takes about 35ms to perform a query. Note that these culling approaches are orthogonal to our work and could be integrated into our framework as well. Some recent algorithms have implemented the hierarchical CCD test on multi-core CPU systems [KHeY08] and performed continuous self-collision on the cloth/sphere benchmark in 53ms (vs. our 34ms) using an 8-core Xeon system. A more recent hybrid version running on 4 CPU cores and 2 GPUs improves timings to only 23ms, but uses more computational resources and runs only pairwise intersection tests on the GPU [KHH*09].

Several previous approaches have been proposed to perform self-collision on GPUs using rasterization algorithms [HTG03, KP03, GRLM03]. Govindaraju *et al.* [GKJ*05] used occlusion queries along with CPU-based overlap tests to perform continuous self-collisions for cloth simulation and were able to handle a 13K triangle version of the 40K Princess model at about 500ms. In contrast, our approach tests the same model with three times the complexity thirty times faster (though, disregarding scaling issues, the GPU used here has roughly 10-20 times peak performance.) Similarly, Sud *et al* [SGG*06] perform self-collisions by using discrete Voronoi diagrams generated by rasterization. Their approach took 800ms (1̃50ms scaled by FLOPs) on a 15K version of the Cloth/Ball scene we use, whereas our approach is over 25x faster on the full 92K model. Note that it is hard to compare performance across GPU generations. In addition, previous approaches relied heavily on occlusion query performance and CPU-GPU bandwidth which has not scaled with the computational power of many-core GPUs.

Distance queries: The fastest CPU approaches to distances are based on RSS hierarchies. We compare our performance against a single-core implementation of the widely-used PQP [LGLM00] running on our benchmark system. For the Car/Seat benchmark, the CPU-based query time is about 1.2s (on a single core) not including build and refit vs. 55ms for our GPU implementation. Additionally, refitting and rebuilding the RSS hierarchy is drastically slower and for complex models may dominate the overall simulation cost. On GPUs, rasterization-based approaches to distances queries have also been demonstrated. Sud *et al* [SGG*06] perform object-precision distance queries: on the same model at higher resolution (92k vs. 15k triangles) our approach performs roughly 27x faster (5x scaled by FLOP/s). Alternate approaches that compute image-space or voxel-space distance fields, such as [MRS08] provide roughly equivalent performance to ours, but may not work on complex scenes such as the car-seat benchmark due to lack of precision and when object-space accuracy is required.

7. Conclusion and future work

We have presented new GPU algorithms for discrete and continuous collision detection. We have also introduced a parallel front-based traversal method for collision detection that greatly reduces the bottlenecks in collision traversal. Our results show that we can compute and use tight-fitting bounding volumes interactively and perform continuous collision detection an order of magnitude faster than previous GPU approaches and competitively or faster than current multi-core CPU methods. We also provided a fast distance query solution that increase the speed of existing exact methods significantly despite the seemingly lack of inherent parallelism of the problem. We have shown the advantages of wide-branching tree structures over binary trees in parallel distance queries, and have worked around the high degree of data dependency by reformulating both the construction and traversal algorithm to extract enough parallelism.

There are many avenues for future work. Algorithms that use hierarchies and operations similar to those of proximity queries include n-body simulation, point-based modeling algorithms and many more. Enabling their efficient implementation on GPUs is important for high-performance implementations. Other variations of proximity queries similar to separation distance queries are also widely used and could be implemented similarly. For self-collision, techniques have been proposed to reduce the amount of edge/edge and vertex/face intersection tests significantly [CTM08]. Our approach is mostly orthogonal, so it would be interesting to investigate implementing these methods in our framework.

Acknowledgments

We would like to thank Liangjun Zhang and Will Moss for their help and E. Ferre from Kineo CAM for the car model. This research was supported in part by ARO Contract W911NF-04-1-0088, NSF awards 0636208, 0917040 and 0904990, DARPA/RDECOM Contract WR91CRB-08-C-0137, and Intel.

References

[ABP98] ARORA N. S., BLUMOFE R. D., PLAXTON C. G.: Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures* (New York, NY, USA, 1998), ACM, pp. 119–129.

[CT08] CEDERMAN D., TSIGAS P.: On sorting and load balancing on gpus. *SIGARCH Comput. Archit. News 36*, 5 (2008), 11–18.

[CTM08] CURTIS S., TAMSTORF R., MANOCHA D.: Fast collision detection for deformable models using representative-triangles. *Proc. of ACM Symposium on Interactive 3D Graphics and Games* (2008).

[EL01] EHMANN S., LIN M.: Accurate and fast proximity queries between polyhedra using surface decomposition. In *Proc. of Eurographics* (2001).

[Eri04] ERICSON C.: *Real-Time Collision Detection*. Morgan Kaufmann, 2004.

[GKJ*05] GOVINDARAJU N., KNOTT D., JAIN N., KABAL I., TAMSTORF R., GAYLE R., LIN M., MANOCHA D.: Collision detection between deformable models using chromatic decomposition. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH) 24*, 3 (2005), 991–999.

[GLM96] GOTTSCHALK S., LIN M., MANOCHA D.: OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96* (1996), 171–180.

[GRLM03] GOVINDARAJU N., REDON S., LIN M., MANOCHA D.: CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. *Proc. ACM SIGGRAPH/EG Workshop on Graphics Hardware* (2003), 25–32.

[GW07] GRINBERG I., WISEMAN Y.: Scalable parallel collision detection simulation. *Proc. of Signal and Image Processing* (2007).

[HGS*07] HUGHES C. J., GRZESZCZUK R., SIFAKIS E., KIM D., KUMAR S., SELLE A., CHHUGANI J., HOLLIMAN M. J., CHEN Y.-K.: Physical simulation for animation and visual effects: parallelization and characterization for chip multiprocessors. In *ISCA* (2007), pp. 220–231.

[HTG03] HEIDELBERGER B., TESCHNER M., GROSS M.: Real-time volumetric intersections of deforming objects. *Proc. of Vision, Modeling and Visualization* (2003), 461–468.

[JC04] JOHNSON D. E., COHEN E.: Unified distance queries in a heterogeneous model environment. In *ASME DETC* (2004).

[KG94] KUMAR V., GRAMA A. Y.: Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing 22* (1994), 60–79.

[KHeY08] KIM D.-S., HEO J.-P., EUI YOON S.: *PCCD: Parallel Continuous Collision Detection*. Tech. Rep. CS-TR-2008-298, Dept. of CS, KAIST, 2008.

[KHH*09] KIM D.-S., HEO J.-P., HUH J., KIM J., EUI YOON S.: HPCCD: hybrid parallel continuous collision detection. In *Computer Graphics Forum (Proc. Pacific Graphics)* (2009).

[KHM*98] KLOSOWSKI J., HELD M., MITCHELL J., SOWIZRAL H., ZIKAN K.: Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. on Visualization and Computer Graphics 4*, 1 (1998), 21–37.

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

[KSTK95] KITAMURA Y., SMITH A., TAKEMURA H., KISHINO F.: Parallel algorithms for real-time colliding face detection. *IEEE RO-MAN'95 TOKYO* (Jul 1995), 211–218.

[LG07] LE GRAND S.: Broad-phase collision detection with cuda. *GPU Gems 3* (August 2007).

[LGLM00] LARSEN E., GOTTSCHALK S., LIN M., MANOCHA D.: Distance queries with rectangular swept sphere volumes. *Proc. of IEEE Int. Conference on Robotics and Automation* (2000), 3719–3726.

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast bvh construction on gpus. In *Proc. Eurographics '09* (2009).

[MLM08] MOSS W., LIN M., MANOCHA D.: Constraint-based motion synthesis for deformable models. *Computer Animation and Virtual World*, September (2008). Cover Image, Special Issue (Best of Computer Animationand Social Agents).

[MRS08] MORVAN T., REIMERS M., SAMSET E.: High performance gpu-based proximity queries using distance fields. *Computer Graphics Forum 27*, 8 (Dec. 2008), 2040–2052.

[Pro97] PROVOT X.: Collision and self-collision handling in cloth model dedicated to design garment. *Graphics Interface* (1997), 177–189.

[Qui94] QUINLAN S.: Efficient distance computation between non-convex objects. pp. 3324–3329.

[RK87] RAO V. N., KUMAR V.: Parallel depth-first search, part i: Implementation. *International Journal of Parallel Programming 16* (1987), 6–479.

[SGG*06] SUD A., GOVINDARAJU N., GAYLE R., KABUL I., MANOCHA D.: Fast proximity computation among deformable models using discrete voronoi diagrams. *Proc. of ACM SIGGRAPH* (2006), 1144–1153.

[SGGM06] SUD A., GOVINDARAJU N., GAYLE R., MANOCHA D.: Interactive 3d distance field computation using linear factorization. In *Proc. ACM Symposium on Interactive 3D Graphics and Games* (2006), pp. 117–124.

[SOM04] SUD A., OTADUY M. A., MANOCHA D.: DiFi: Fast 3D distance field computation using graphics hardware. *Computer Graphics Forum (Proc. Eurographics) 23*, 3 (2004), 557–566.

[TCYM08] TANG M., CURTIS S., YOON S.-E., MANOCHA D.: Interactive continuous collision detection between deformable models using connectivity-based culling. In *Proc. ACM SPM '08* (2008), pp. 25–36.

[vdB97] VAN DEN BERGEN G.: Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools 2*, 4 (1997), 1–14.

[ZHKM08] ZHANG L., HUANG X., KIM Y., MANOCHA D.: D-plan: Efficient collision-free path computation for part removal and disassembly, 2008.

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time kd-tree construction on graphics hardware. In *Proc. SIGGRAPH Asia* (2008).