Real-time Collision Culling of a Million Bodies on Graphics Processing Units

Fuchang Liu* Takahiro Harada† Youngeun Lee‡ Young J. Kim§
Ewha Womans University, Seoul, Korea †Advanced Micro Devices, Inc.

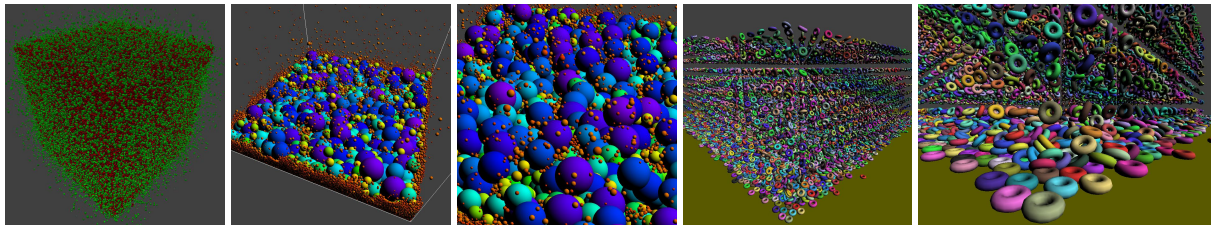


Figure 1: ***GPU-based collision culling for massive bodies.*** *N-body collision detection for 1M arbitrarily moving boxes (first image), real-time simulation of 0.3M particles of random size on GPUs (second and third images), real-time rigid-body dynamics for 16K torus models of varying sizes on GPUs (fourth and fifth images). In these challenging benchmarks, our algorithm can find all the colliding bodies at interactive rates.*

Abstract

We cull collisions between very large numbers of moving bodies using graphics processing units (GPUs). To perform massively parallel sweep-and-prune (SaP), we mitigate the great density of intervals along the axis of sweep by using principal component analysis to choose the best sweep direction, together with spatial subdivisions to further reduce the number of false positive overlaps. Our algorithm implemented entirely on GPUs using the CUDA framework can handle a million moving objects at interactive rates. As application of our algorithm, we demonstrate the real-time simulation of very large numbers of particles and rigid-body dynamics.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems

Keywords: Collision detection, Graphics hardware, Dynamics simulation

1 Introduction

Collision detection (CD) is the problem of determining the interference between objects moving in space. CD has many applications in computer graphics, computer animation, virtual reality, geometric modeling, haptics and robotics [Lin and Manocha 2003]. Due to its importance, CD has been extensively investigated and still considered as an active research area.

At a high level, CD techniques can be classified into broad-phase and narrow-phase algorithms. The former determines possible intersecting pairs among n objects; the latter decides whether a pair of objects, usually found by a broad-phase algorithm, actually intersect. This paper focuses on broad-phase CD, also called n-body collision culling.

There is increasing demand for handling very large sets of objects in different applications. In physically-based animation, particle-based simulation techniques such as the distinct-element method (DEM), smoothed particle hydrodynamics (SPH) and moving-particle semi-implicit (MPS) require millions of particles for realistic simulation of fluids and granular materials [Harada et al. 2007]. In massively multi-player online role-playing games (MMORPG), thousands of players may be logged on to a single gaming server, and interacting with tens of thousands of autonomous creatures and environmental obstacles [Zerodin 2010]. In a virtual city environment [Brown 2008], tens of thousands of objects interact with each other, and their motions need to be physically realistic. All of these applications require a very fast broad-phase CD algorithm.

The most well-known broad-phase CD algorithms include sweep and prune (SaP) [Baraff 1992; Cohen et al. 1995] and spatial subdivision [Ericson 2005]. The former is essentially a dimension-reduction approach in which objects are projected into a lower-dimension (typically a single dimension) and then overlap tests are performed by sweeping a hyperplane along the dimensional axis. The latter is a spatial hashing technique in which objects are registered into some form of grid, and local intersection tests are performed inside the grids. SaP is effective when moving objects have a high spatial coherence, which can be exploited by the sweep operation; otherwise, the efficiency substantially degrades. And subdivision is effective for objects of similar size; otherwise, a sophisticated hierarchical structure is required, which is hard to maintain for moving objects.

Main Contribution We present a fast collision culling algorithm for very large numbers of objects using graphics processing units (GPUs). It is a hybrid SaP and subdivision method which can perform collision culling efficiently without any restriction on the sizes of objects or the nature of their motion. The key idea behind our algorithm is to take advantage of the parallel nature of SaP by performing SaP simultaneously for many objects using the many blocks of threads available on modern GPUs. We also use principal component analysis (PCA) to choose the best sweep direction for parallel SaP. In addition, a novel two-level spatial subdivision technique is combined with parallel SaP to alleviate the problem of densely projected intervals, caused by the reduction in dimensionality. Our algorithm can easily handle insertion and removal of

*liufu@ewha.ac.kr
†Takahiro.Harada@amd.com
‡youngeunlee@ewhain.net
§kimy@ewha.ac.kr

objects and can also take advantage of motion coherence if it exists in the simulated environment. We have implemented our algorithm entirely on GPUs using the CUDA framework, and in practice, we can perform collision culling for a million arbitrarily moving objects at interactive rates. We assess this to be 71 times faster than state-of-art collision culling programs running on CPUs, and 212 times faster than methods that use GPUs. As application of our algorithm, we demonstrate the real-time simulation of very large numbers of particles and rigid-body dynamics completely running on GPUs using our fast collision culling algorithm.

2 Previous Work

We will briefly survey prior work related to n-body collision culling.

2.1 Geometric Intersection

Earlier CD algorithms considered the n-body collision detection in terms of geometric intersections among simply-shaped objects such as axis-aligned bounding boxes (AABBs) or spheres.

The optimal algorithm [Edelsbrunner and Maurer 1981] to find the intersections of n AABBs in 3D is based on plane sweeps and takes $O(n \log^2 n + k)$ where k is the number of objects that actually intersect. Since then, many sweep-based algorithms continue to be introduced. Notably, a sorting-based technique known as sweep and prune (SaP), due to several researchers [Baraff 1992; Lin 1993; Cohen et al. 1995], has proven to be very effective in practice, especially in an environment of high spatial coherence. The computational complexity of SaP is known to be $O(n + s)$, where s is the number of swapping operations required to maintain the sorted order of the objects. Recently, [Tracy et al. 2009] have shown that s can increase super-linearly with respect to the number of objects, and motivating a hybrid SaP and spatial subdivision to reduce the number of swaps [Ponamgi et al. 1995; Tracy et al. 2009]. [Coming and Staadt 2006; Coming and Staadt 2008] introduce a continuous version of SaP for objects that follow time-dependent paths. No parallel algorithms are currently known for sweep-based, collision culling and it is not clear whether existing methods are able to handle a million objects due to the super-linearity of the swapping problem. Moreover, [Terdiman 2007] hypothesized that parallelizing the hybrid SaP and subdivision may perform poorly. [Woulfe et al. 2007] did propose a hardware-based broad-phase CD algorithm, but this takes $O(n^2)$ time and is therefore limited to a relatively small number of AABBs.

Another class of collision culling algorithms are based on bounding volume hierarchy (BVH), which is constructed by treating individual objects as leaf nodes and constructing their bounding volumes recursively. Then, self-collision detection within the same BVH is used to find intersecting objects [Tang et al. 2009; Tang et al. 2010]. More recently, GPU-based algorithms have been proposed to construct the BVH dynamically at run-time [Lauterbach et al. 2009; Lauterbach et al. 2010]. Even though these approaches are quite efficient, they are mainly designed for narrow-phase CD and it is not clear whether these are applicable to fast broad-phase CD for massive objects.

2.2 Spatial Subdivision

In spatial subdivision, a workspace is divided into one or more grids, which localize collision detection. Different types of uniform and non-uniform grids such as the k-d tree, octree and BSP-tree have been introduced [Samet 2006] and many associated CD techniques have been proposed [Ericson 2005]. Uniform subdivision is the simplest and most suitable for GPU implementation [Grand 2008; Mazhar et al. 2009], but the effectiveness of uniform grids is severely reduced with objects of widely varying sizes.

Hierarchical subdivision methods address this problem. [Mirtich 1996] proposed a CD technique based on hierarchical spatial hash table, but is not suitable for large inhomogeneous data-sets. Structures such as recursive grids, hierarchies of uniform grids and adaptive grids [Ericson 2005] are widely used in ray-tracing to localize ray/object intersection. In particular, [Zhou et al. 2008] showed that a k-d tree can be constructed in real-time for global illumination using GPUs. However, these non-uniform grid structures perform poorly when objects undergo severe transformation [Wald 2007].

3 Sweep and Prune on GPUs

We will now describe our parallel SaP algorithm. We will begin by explaining the sequential SaP algorithm designed to run on CPUs, and then introduce our parallel algorithm for GPUs.

3.1 Sweep and Prune

Given n objects O_i in 3D, the goal of SaP is to find all overlapping pairs $\mathcal{P}$ of objects. Thus $\mathcal{P} = \{(O_i, O_j) | O_i \cap O_j \neq \emptyset, 1 \leq i \neq j \leq n\}$. Often, an object O_i is a simple volume such as an AABB or sphere that bounds more complicated geometry. We will assume that the objects are simple enough to allow us to determine whether $O_i \cap O_j \neq \emptyset$ in constant time.

SaP was originally designed to exploit the fact that a pair of AABBs overlap iff their projected intervals overlap in all three dimensions. The original SaP algorithm [Baraff 1992] can be described as follows:

1. Project the extent of each object O_i on to some principal axis, for instance, the x-axis, producing an 1D interval of extrema, $\mathcal{I}_i = [m_i, M_i]$.
2. Sort m_i and M_i for all i, and obtain a sorted list $\mathcal{L}$.
3. Sweep $\mathcal{L}$ and maintain an active list $\mathcal{A}$ as follows:
   - Add O_i to $\mathcal{A}$ when m_i is retrieved from $\mathcal{L}$ and add all O_i in $\mathcal{A}$ to the set of colliding pairs $\mathcal{P}_x$.
   - Remove O_i from $\mathcal{A}$ when M_i is retrieved from $\mathcal{L}$.
4. Repeat steps 1-3 for $y-, z-$axes, and obtain the colliding pairs $\mathcal{P}_y$ and $\mathcal{P}_z$. Report the final set of colliding pairs $\mathcal{P} = \mathcal{P}_x \cap \mathcal{P}_y \cap \mathcal{P}_z$.

If the motion of the objects is highly coherent, then step 2 in this algorithm can be implemented efficiently as an insertion sort and step 3 can be replaced by swapping operations between neighboring O_i [Cohen et al. 1995]. Note that step 4 becomes redundant for spheres, since the exact overlap test between spheres can be implemented along axial direction by comparing the distance between the centers of two spheres with the sum of their radii. Moreover, the intervals $\mathcal{I}_i$ are the diameters of the spheres positioned on their projected centers.

3.2 Parallel Sweep and Prune

The basic SaP algorithm is not suitable for very large numbers of objects in arbitrary motion, nor for parallel implementation, even though the sorting step (step 1) can be efficiently parallelized on GPUs using a radix sort [Sengupta et al. 2007]. The reasons are:

- Maintaining the active list $\mathcal{A}$ cannot be parallelized since it depends on the sweeping order.

- Steps 3 and 4 have a low arithmetic intensity which fails to amortize the costly memory accesses of current GPU architectures.
- The assumption of highly coherent motion and the associated swapping operation are undesirable when the number of objects is really large, since the number of operations can increase superlinearly.

We have designed an alternative way of performing SaP which does not require an active list. Steps 1 and 4 remain the same, but in step 2, we now sort O_i by m_i alone to obtain a sorted list $\mathcal{L}$. Then, as we sweep $\mathcal{L}$, we only test whether $m_j \in \mathcal{I}_i$ for some of the objects $O_j, i < j$, since (O_j, O_i) can be inferred from (O_i, O_j). This algorithm may be considered as a parallel version of that proposed by [Terdiman 2007].

By relying on a GPU-based radix sort for step 2, we remove the need for swaps. Then we parallelize the sweeping step 3 by independently sweeping each object O_i within its own interval $\mathcal{I}_i$. More precisely, as set out in Alg.1, we launch a GPU thread $\mathcal{T}_i$ for each object O_i that sweeps a subset of the sorted list $\mathcal{L}$ to find a local list of colliding pairs $\mathcal{P}_i = \{(O_i, O_j) | i < j, m_j \in \mathcal{I}_i\}$ until $\exists j > i, M_i < m_j$ (also see Fig. 2). To increase the arithmetic intensity of our algorithm, we merge steps 3 and 4 by performing overlap tests in all dimensions for each pair $(O_i, O_j) \in \mathcal{P}_i$ such that $O_i \cap O_j \neq \emptyset$. If we are dealing with spheres rather than AABBs, merging steps 3 and 4 is even easier because we only need to compare the centers of two spheres with the sum of their radii. Once all threads finish execution, we merge the local colliding pairs to obtain a final list $\mathcal{P} = \cup \mathcal{P}_i$. In principle, modern GPUs can execute tens of thousands of threads in parallel [NVIDIA 2010]. Note that we can easily insert and remove objects in our algorithm since it does not require any preprocessing or additional querying structures.

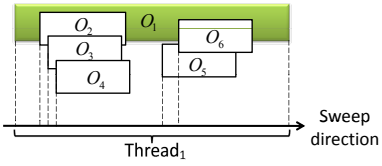


Figure 2: ***Parallel SaP.*** *O_1 is handled by thread$_1$, and the overlapped pairs $\{(O_1, O_2), (O_1, O_3), (O_1, O_4), (O_1, O_5), (O_1, O_6)\}$ are tested for collisions.*

Algorithm 1 GPU SaP Algorithm

1. Project each O_i onto an axis and obtain $\mathcal{I}_i$.
2. Sort all the O_is by m_i in parallel using a radix sort, and obtain the sorted list $\mathcal{L}$.
3. For each O_i, execute the following in parallel.
   (a) Sweep $\mathcal{L}$ in the interval $\mathcal{I}_i$ until $M_i < m_j$ for some object $O_j, i < j$.
   (b) If $m_j \in \mathcal{I}_i$ for any O_j, then check whether $O_i \cap O_j \neq \emptyset$. If so, add the pair (O_i, O_j) to $\mathcal{P}_i$.
4. Return $\mathcal{P} \equiv \cup \mathcal{P}_i$.

3.3 Workload Balancing

It is the extremely large number of computational threads available on modern GPUs that makes Alg. 1 effective. However, if objects are of dissimilar sizes, a big object may be induced in many more colliding pairs than smaller ones. This means that a thread assigned to this big object has more work to do than the smaller ones, while threads assigned to small objects become idle. This unbalances the task distribution and wastes the computing power of GPUs. We solve this problem by assigning more threads to objects likely to require more pairwise collision tests, thus allocating each thread roughly the same workload. Since the actual number of collision pairs is not known in advance, we hypothesize that this number is proportional to the size of an object. For each such object, we divide the pairwise collision tests into partitions containing roughly the same number of tests, and assign one thread to each partition. For instance, in Fig. 3, O_1 needs to test for collisions with 8 other objects, while O_4 requires a single collision test with O_5. But if we divide the tasks for O_1 into two partitions, each thread only needs to process four collision tests.

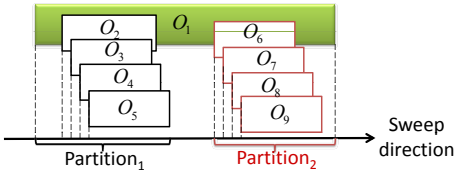


Figure 3: ***Distribution of unbalanced pairwise tests by partitioning.*** *The potentially colliding pairs for O_1 is partitioned into partition$_1$ $= \{(O_1, O_2), (O_1, O_3), (O_1, O_4), (O_1, O_5)\}$ and partition$_2$ $= \{(O_1, O_6), (O_1, O_7), (O_1, O_8), (O_1, O_9)\}$. Each of these partitions is assigned to a separate thread.*

In more detail, let us assume that an object O_i requires a set of pairwise tests $\mathcal{S}_i = \{(O_i, O_{n_1}), \cdots, (O_i, O_{n_j})\}$ We partition this set into subsets $\mathcal{S}_i = P_{i_1} \cup P_{i_2} \cup \cdots \cup P_{i_k}$, where $P_{i_k} = \{(O_i, O_{n'_1}), \cdots, (O_i, O_{n'_l})\}$ and $| \mathcal{S}_i | = \sum | P_{i_k} |$. Further, we impose the restriction that $| P_{i_k} | \leq \tau$, where τ is the maximum number of colliding pairs to be allocated to a single thread. Then, we assign a thread $\mathcal{T}_i$ to perform the tests in every partition P_i. In practice, we can determine τ from the maximum number of active threads that can be executed by GPUs; the higher this maximum, the smaller τ should be.

We also have to determine the number of GPU threads that we need to spawn before we run parallel SaP, since threads cannot be allocated dynamically on current GPUs. To do this, we first find the number of partitions for an object O_i, by using binary search to position M_i in $\mathcal{L}$ such that $m_p \leq M_i \leq m_{p+1}$. Then, $\lceil \frac{p-i}{\tau} \rceil$ is the number of partitions for O_i. The total number of partitions is the number of GPU threads that need to be spawned.

3.4 Choosing the Sweep Direction

So far, we have assumed that one of the three axes is the sweep direction. But this may give a poor result with SaP, as illustrated in Fig. 4. In this example, choosing the $x-$axis leads to an unnecessarily large number of overlap tests.

The best sweep direction is the one that allows projection to separate the data as much as possible. We use principal component analysis (PCA) [Jolliffe 2002] to find the sweep direction which maximizes the variance of objects after projection, which corresponds to the direction of the first principal component. The first principal component $\mathbf{w}_1$ of a data set $\mathbf{X}$ with a mean of zero can be expressed as:

$$\begin{aligned} \mathbf{w}_1 &= \arg\max_{\|\mathbf{w}\|=1} Var\{\mathbf{w}^T\mathbf{X}\} \\ &= \arg\max_{\|\mathbf{w}\|=1} E\{\mathbf{w}^T\mathbf{X}\mathbf{X}^T\mathbf{w}\}, \end{aligned}$$

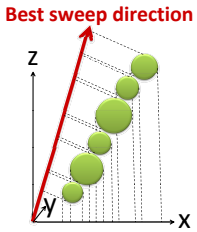


Figure 4: ***Best sweep direction.*** *Projection on to the best sweep direction, found by PCA, produces much fewer interval overlaps than projection on to, for instance, x-axis.*

where $\mathbf{w}_1$ is the eigenvector corresponding to the largest eigenvalue of the covariance matrix $\mathbf{C} = \mathbf{X}\mathbf{X}^T$.

PCA has been implemented using GPUs [Andrecut 2009]. However, this algorithm is too slow for interactive applications and does not exploit the low dimensionality of our application. We have therefore implemented PCA as follows: We first compute the mean of $\mathbf{X}$ using the reduced (up-sweep) parallel scan algorithm on GPUs [Sengupta et al. 2007]. Then the eigenvectors of $\mathbf{C} = \mathbf{X}\mathbf{X}^T$ are easy to compute: $\mathbf{X}$ is $3 \times n$ and $\mathbf{C}$ is a 3×3 symmetric matrix. Finally we use the Jacobi method [Press et al. 1992] to find the eigenvalues of the symmetric matrix $\mathbf{C}$.

3.5 Motion Coherence

The swapping in the original SaP algorithm takes advantage of motion coherence. This is highly effective when only a small number of objects are moving; however, it is not suitable for GPUs since it requires frequent memory accesses, which are more expensive than computation on GPUs. We now propose a GPU-friendly technique which is especially suitable for situations in which a relatively small proportion of the total number of objects are moving at any time.

Motion coherence is utilized by classifying the set of objects $\mathcal{O} = \{O_i\}$ at a time t into two time-dependent disjoint subsets: a set of moving objects $\mathcal{O}_m(t)$ and a set of static objects $\mathcal{O}_s(t)$, *i.e.* $\mathcal{O} = \mathcal{O}_m(t) \cup \mathcal{O}_s(t)$. Then we find the time-dependent colliding pairs $\mathcal{P}(t)$ by performing CD for $\mathcal{O}_m(t)$ and $\mathcal{O}_s(t)$ separately. This strategy is effective when $|\mathcal{O}_m(t)| \ll |\mathcal{O}_s(t)|$, since the colliding pairs of $\mathcal{O}_s(t)$ are not much different from those in $\mathcal{O}_s(t-1)$ and thus changing from $\mathcal{P}(t-1)$ to $\mathcal{P}(t)$ only requires a small update.

More specifically, we perform the following procedure to update the collision pairs $\mathcal{P}(t-1)$ to $\mathcal{P}(t)$:

1. Project the extent of $O_i \in \mathcal{O}$ and sort them by m_i as before.
2. Find the set of colliding pairs $\mathcal{P}_M(t)$ caused by the moving objects in $\mathcal{O}_m(t)$ as follows:
   (a) For all $O_i \in \mathcal{O}_s(t)$, find the set of colliding pairs $\mathcal{P}_{sm}(t)$ between the sets of static $\mathcal{O}_s(t)$ and moving objects $\mathcal{O}_m(t)$ by sweeping in parallel, so that $\mathcal{P}_{sm}(t) = \{(O_i, O_j) | O_i \cap O_j \neq \emptyset \ \wedge \ O_i \in \mathcal{O}_s(t), O_j \in \mathcal{O}_m(t) \ \wedge \ i < j\}$.
   (b) For all $O_i \in \mathcal{O}_m(t)$, find the set of colliding pairs $\mathcal{P}_{m*}(t)$ by comparing the set of moving objects $\mathcal{O}_m(t)$ against all the other objects in $\mathcal{O}$ by sweeping in parallel, so that $\mathcal{P}_{m*}(t) = \{(O_i, O_j) | O_i \cap O_j \neq \emptyset \wedge O_i \in \mathcal{O}_m(t), O_j \in \mathcal{O} \ \wedge \ i < j\}$.
   (c) $\mathcal{P}_M(t) \equiv \mathcal{P}_{sm}(t) \cup \mathcal{P}_{m*}(t)$.
3. Find the set of interfering pairs $\mathcal{P}_S(t)$ within the set of static objects in $\mathcal{O}_s(t)$ which are static after $\mathcal{P}(t-1)$. This is expressed by $\mathcal{P}_S(t) = \mathcal{P}(t-1) - \mathcal{P}'_M(t-1)$, where $\mathcal{P}'_M(t-1)$ is the set of colliding pairs $\mathcal{P}(t-1)$ at time $t-1$ even though one of the objects in each pair is in $\mathcal{O}_m(t)$ at time t; thus $\mathcal{P}'_M(t-1) \equiv \{\forall (O_i, O_j) \in \mathcal{P}(t-1) | O_i \in \mathcal{O}_m(t) \vee O_j \in \mathcal{O}_m(t)\}$.
4. Report the colliding pairs $\mathcal{P}(t) = \mathcal{P}_M(t) \cup \mathcal{P}_S(t)$.

Note that this approach does not rely on any swapping operations and is a recasting of the parallel SaP methods discussed in the previous sections; and thus it is quite effective.

4 Spatial Subdivision on GPUs

As observed by [Tracy et al. 2009], using the SaP for a very large number of objects may produce a daunting number of overlaps between projected intervals in the sweep direction, even though the actual number of interfering objects in 3D may be not excessive. This problem can persist even if the best sweep direction is chosen using the PCA technique, as explained in Sec. 3.4. To solve this problem, we now propose a hybrid approach in which SaP is combined with both workspace subdivision and cell subdivision, to cull away objects that are relatively far apart.

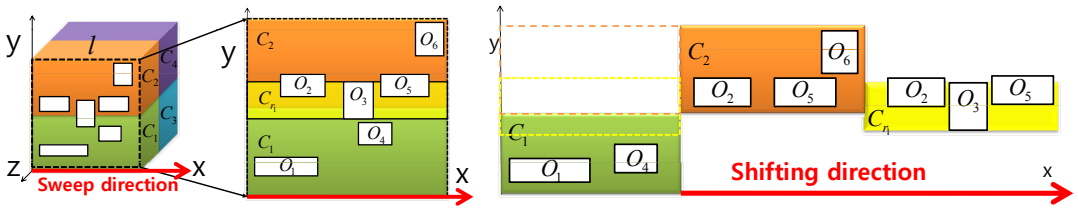
4.1 Workspace Subdivision

The first subdivision is performed before SaP is executed, to reduce the density of the object intervals projected on to the sweep axis. This is a uniform subdivision, to facilitate GPU implementation. Given a sweep direction $\mathbf{d}$ for SaP, we subdivide the 3D workspace into $m \times m$ grid cells cut by planes parallel to $\mathbf{d}$. A z cross-section of a 2×2 subdivision is illustrated in Fig.5-(a) when x is the SaP direction. In practice, we choose $m = \lceil \frac{n}{64K} \rceil$ where n is the number of objects, since we have found that the techniques described in Sec. 3 work well for up to 64K objects using PCA. Initially we put all the objects into two subsets: $\mathcal{O}_{in} = \{O_i \in \mathcal{O} | O_i$ is completely inside some cell $C_j\}$ and $\mathcal{O}_{bd} \equiv \mathcal{O} - \mathcal{O}_{in}$. Later, $\mathcal{O}_{bd}$ will be further expanded.

If there were no object crossing the boundary of any cell (*i.e.* $\mathcal{O}_{bd} = \emptyset$), we could execute SaP in parallel across rows or columns of grid cells. This requires each cell to have a unique index, as shown in Fig. 5-(a), and the index assignment can be arbitrary. Then, we can translate the interval $\mathcal{I}_i$ projected by each object $O_i \in \mathcal{O}_{in}$ by $(j-1) \times l$ along $\mathbf{d}$, where C_j is the cell to which O_i belongs and l is the size of the workspace along $\mathbf{d}$, as illustrated in Fig. 5-(b). This *shifting* technique reduces the number of overlaps in the sweep direction and increases the effectiveness of SaP.

In practice, some objects will cross the cell boundaries; *i.e.* $\mathcal{O}_{bd} \neq \emptyset$. We find the objects that may potentially collide with those in $\mathcal{O}_{bd}$, and call these the objects affected by the boundaries. To find these objects, we compute the maximum extent of the objects in $\mathcal{O}_{bd}$. The extent of a sphere is its diameter and that of an AABB is its longest dimension. Then, we estimate the regions $\mathcal{R} = \{C_{\mathcal{R}_1}, \cdots, C_{\mathcal{R}_n}\}$ affected by $\mathcal{O}_{bd}$, for instance $\mathcal{R} = C_{r_1}$ in Fig. 5, and find all the objects that overlap with $\mathcal{R}$; we add these objects to $\mathcal{O}_{bd}$. We treat each region C_{r_i} as a new cell with a unique cell index, and associate each object in $\mathcal{O}_{bd}$ with a cell C_{r_i} that overlaps that object. For instance, in Fig. 5, initially $\mathcal{O}_{bd} = \{O_3\}$, but expanded to $\mathcal{O}_{bd} = \{O_2, O_3, O_5\}$ due to C_{r_1}.

Then, we translate each $O_i \in \mathcal{O}_{bd}$ along the SaP direction, just like the objects in $\mathcal{O}_{in}$ based on its cell index r_i of C_{r_i}. Notice that some objects may be repeated if they belong to both $\mathcal{O}_{in}$ and $\mathcal{O}_{bd}$



(a) Cross-sectioned view of workspace subdivision (b) Shifting the cells to perform SaP

Figure 5: ***Workspace subdivision combined with SaP. (a)*** *2×2 workspace subdivision. $\{O_1, O_3, O_4\} \subset C_1, \{O_2, O_3, O_5, O_6\} \subset C_2, \{O_2, O_3, O_5\} \subset C_{r_1}, \{O_1, O_2, O_4, O_5, O_6\} \subset \mathcal{O}_{in}, \{O_2, O_3, O_5\} \subset \mathcal{O}_{bd}$. **(b)** x is the SaP/shifting direction. $\{O_1, O_4\}$ and $\{O_2, O_5, O_6\}$ are shifted with C_1 and C_2, respectively, since $\{O_1, O_4\} = C_1 \cap \mathcal{O}_{in}, \{O_2, O_5, O_6\} = C_2 \cap \mathcal{O}_{in}$. $\{O_2, O_3, O_5\}$ are also shifted with C_{r_1} since $\{O_2, O_3, O_5\} = C_{r_1} \cap \mathcal{O}_{bd}$.*

(*e.g.* O_2, O_5 in Fig. 5), or if they are affected by more than one cell in $\mathcal{R}$.

4.2 Cell Subdivision

Since objects within different cells in the workspace subdivision cannot collide, these objects do not need any further collision tests. This substantially reduces the computational overhead of parallel SaP. However, it is still possible that some cells can contain many objects whose projected intervals overlap, even though the associated objects do not actually intersect. To reduce the number of false positives, we employ a second, intra-cell, subdivision during SaP.

Our intra-cell subdivision is a variant of that used by [Mazhar et al. 2009], but ours is only two-dimensional because we subdivide the cells of the workspace subdivision parallel to the sweep direction. We introduce a new mapping for subcells. Using an m-bit address, where m is the logarithm of the total number of subcells and half of these bits correspond to each dimension, as shown in Fig. 6. This allows SaP to be performed only for those objects which share the same subcell. Fig.6 shows an example in which a cell is divided into 16 subcells, eliminating the need for a collision test between objects O_1 and O_4. We can find out which subcells contain an object occupies from the extrema of that object. The extremal points of O_4 in Fig. 6 are mapped to 1010 and 1111. Then, we can use a bitwise `AND` operation to check whether two objects share the same subcell. Finally, SaP parallelized for each O_i only needs to consider the objects that share the same subcell, which can substantially reduce the candidate set.

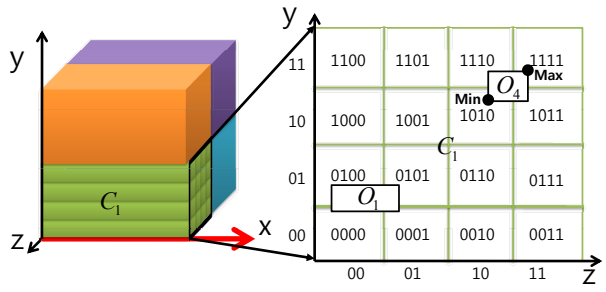


Figure 6: ***Cell subdivision.*** *Each cell is subdivided into 4×4 subcells and the right image is a cross-sectioned view of the left image for C_1.*

5 Results and Discussion

Now we present our experimental results, discuss the performance of our broad-phase CD algorithms, and provide an analysis of the algorithm.

5.1 Implementation and Benchmarking Results

We implemented our algorithm using the Visual Studio C++ and NVIDIA CUDA programming languages on a PC equipped with Intel Quad-core 2.66GHz CPU with a 2.8 GB main memory and an NVIDIA Tesla C1060 graphics card with a 4Gb memory, under Windows Vista.

To implement the parallel SaP in Alg. 1, we allocated a number of CUDA threads to each object, determined by the number of partitions of that object, as explained in Sec. 3.3. Since our algorithm is designed to deal with massive data, it is important to utilize the memory hierarchy in CUDA effectively. Initially, we store all objects as well as the final collision results in the global memory of the GPUs. Then we take advantage of the temporal coherence of the parallel SaP operations by caching the positions of some objects in the shared memory of each block[1] of threads. However, only 512 objects can be cached in each thread block because the size of the shared memory is limited and its use can also affect the number of active threads that GPUs can spawn (*i.e.* GPU occupancy).

We benchmarked the performance of our algorithm in different scenarios including random object configurations, particle simulation and rigid-body dynamics, and compared it against that of other algorithms based on both CPUs and GPUs.

Random Configurations We used a benchmark setup similar to the Bullet collision library[2], in which a set of AABBs are uniformly distributed in space and moving randomly. As we changed the number of AABBs from 16K to 960K, we measured the performance of our algorithm and that of the three broad-phase CD algorithms provided in Bullet, which are BoxPruning, ArraySaP and an AABB dynamic tree. The first two algorithms are based on SaP and the last one using a dynamic bounding volume hierarchy. All of these algorithms run on CPU. The size of the AABBs also varies from 0.5% to 8% to the size of the bounding box of the workspace. As shown in Fig. 7, our algorithm outperforms the fastest of the Bullet implementations (*i.e.* the AABB dynamic tree) by a factor of 71 times.

We also investigated the performance of our algorithm when only some of the objects are moving. The objects are one million AABBs of varying sizes, and we changed the percentage of moving objects from 5% to 25%, and observed the performance of the technique explained in Sec. 3.5. Fig.8 shows that the number of new

[1] In CUDA a thread block may contain up to 512 threads.
[2] `http://bulletphysics.org`

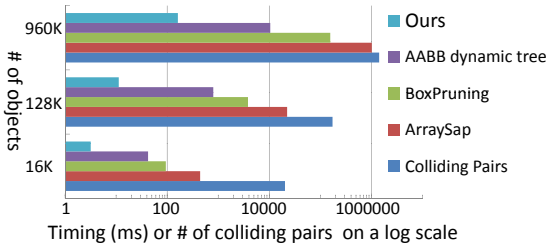


Figure 7: ***Comparison of CD performance with the different methods in Bullet CPU algorithms.*** *Our algorithm takes 3 ms, 11 ms and 161 ms for 16K, 128K, and 960K objects, respectively. Each of these results corresponds to 14, 71, and 64 times performance improvement over the fastest method in Bullet.*

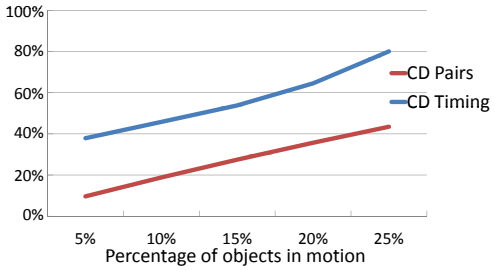


Figure 8: ***CD when only some objects are moving.*** *The CD pairs means the proportion of collisions due to moving objects. The CD timing means the computation time relative to all objects moving.*

collisions generated by moving objects, as a proportion of the total number of interferences is almost linear with computation time, which implies that as more collision pairs are introduced by moving objects, our algorithm requires more time to process them. This means that our algorithm efficiently utilizes the collision results introduced by static objects, which are cached from the previous time step.

Particle Simulation Many physically-based animations require broad-phase and narrow-phase CD algorithms. However, broad-phase CD is usually sufficient for particle simulation, since particles are often modeled as spheres. We benchmarked on large sets of particles of varying sizes. We did this by modifying an open particle simulation demo, originally from NVIDIA[NVIDIA 2010; Grand 2008]. This demonstration program comes with a GPU-based uniform subdivision collision culling algorithm. An algorithm due to [Mazhar et al. 2009] is also similar to this algorithm, but their code is not publicly available. As shown in Fig. 1, we introduced 100K and 0.3M spheres of the size varying from 0.3% to 20% of the dimension of the workspace and simulated their motions under gravity. We then measure the performance of our algorithm and that of a uniform subdivision algorithm that also runs GPUs. While CD takes up most of the computation, it is hard to decouple the collision times from the simulation times using NVIDIA's uniform subdivision method. However, for 100K and 0.3M particles, our algorithm takes 56 *ms* and 252 *ms* on average for both collision detection and particle simulation while uniform subdivision 4452 *ms* and 53464 *ms*; thus our algorithm outperforms uniform subdivision by a factor of 212 times.

Approximate Rigid-Body Dynamics Accurate rigid-body dynamics requires narrow- as well as broad-phase CD, which can be costly for massive objects. However, for real-time applications such as computer games and virtual environments, it is often acceptable to approximate the dynamics to maintain the speed of simulation. [Harada 2007] approximated a rigid model with a set of uniform spheres, and used a penalty-based approach running in parallel on GPUs. This avoids narrow-phase CD. This approach is quite fast, but many spheres are required for a reasonably good approximation of rigid geometry, and the feasible number of colliding pairs is also limited by the image-based CD technique. We have extended this approach by using spheres of arbitrary sizes and our broad-phase CD algorithm allows many more colliding pairs. We simulated 16K torus models approximated by six spheres of varying size moving under gravity. We were able to simulate the approximate rigid-body dynamics entirely running on GPUs in 18 *ms*, including collision detection, as shown in Fig.1.

5.2 Discussion

We compare the effectiveness of a sweep axis determined using PCA, explained in Sec. 3.4. The objects are spheres randomly moving inside an AABB whose aspect ratio is 1:2:3 along each dimension, and torus models whose motion is governed by rigid-body dynamics. Fig. 9 shows that the use of PCA speeds up the algorithm by a factor of more than three times. In these experiments, we do not use subdivision.

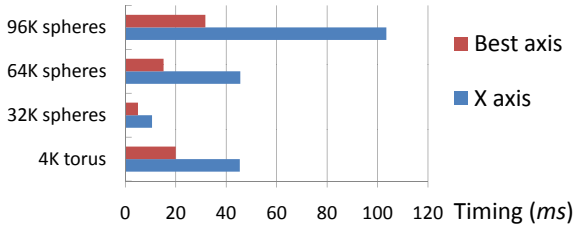


Figure 9: ***Performance of SaP using the best axis, found by PCA, and x axis.***

We use PCA for up to 64K objects, since the performance difference with and without using PCA is becoming less significant due to the overlap density of projected intervals. More precisely, for more than 64K objects, we compute the variation of projected intervals along x, y, z axes using GPUs and choose one of the principal axes as sweep direction.

The timing contributions of the different components of our CD algorithm are given in Table 1, for the random-spheres benchmark. The timings with and without using workload balancing technique explained in Sec. 3.3 are also given in Table 2.

Number of objects	64K	256K	1M
PCA	3%	-	-
Radix sort	10%	18%	6%
Subdivision	-	12%	4%
Workload balancing	4%	9%	4%
Parallel SaP	79%	54%	84%
Data preparation	4%	8%	2%

Table 1: ***Timings of the components of our collision culling algorithm.***

Workload balancing	With	Without
64K	13.71	30.05
128K	48.16	124.11
256K	241.51	467.78

Table 2: ***Timings (in ms) with and without using workload balancing for a different number of random-spheres.***

We also analyzed the worst-case computational complexity of our algorithm, under the assumption that the number of active threads on GPUs is unlimited. Given n objects, the radix sort takes $O(n)$ time, binary search for workload balancing takes $O(\log n)$ time, and the parallel SaP takes $O(\tau(\varepsilon + \zeta))$ time. Here, τ is the maximum number of pairwise collision tests in each thread, as explained in Sec. 3.3. ε is the maximum time to read data from global or shared memory, and ζ is the maximum time to write back the collision results to global memory. The finite number of actual GPU threads limits the selection of τ and we set $\tau = \frac{n}{2K}$ on Tesla C1060 where n is the number of objects and this gives good results.

There are naturally some limitations to our algorithm. Even though it is designed to increase the arithmetic intensity, it is still affected by the high memory latency in GPUs. In particular, due to the CUDA architecture, it is still expensive to read object configurations from, and write the collision results back, to global memory. Thus, if the simulation is running on an associated CPU, the readback from the GPU to the CPU may take a substantial amount of time, which depends on the bus architecture. Our algorithm is not completely automatic since it relies on some parameters such as τ and m depending on the number of active threads that GPUs can utilize.

6 Conclusions

We have presented a parallel collision culling algorithm that runs on GPUs. Its essence is to parallelize the SaP operations individually for different objects and then to allocate extra threads to objects likely to require many collision tests. Computation times are further reduced by using PCA to select the sweep direction, and two-level subdivision. Our algorithm substantially outperformed existing CPU- and GPU-based approaches in various benchmarks. In future work, we would like to use the improved caching capability that will be available on future GPUs (*e.g.* the Fermi or Larrabee [Seiler et al. 2008] architectures) to handle object data more rapidly. There are many further possibilities of the use of multi-GPUs, for instance, to perform workspace and cell subdivision simultaneously for each graphics card. We also would like to apply our culling algorithm to deformable models, character motion planning, and fracture dynamics. Finally, continuous collision detection between n-bodies based on our approach is an interesting direction to pursue.

Acknowledgements

This research was supported in part by the IT R&D program of MKE/MCST/IITA (2008-F-033-02, Development of Real-time Physics Simulation Engine for e-Entertainment) and the KRF grant (2009-0086684). We thank Erwin Coumans and Roman Ponomarev for their help on optimizing GPU codes, and Hyungon Ryu at NVIDIA Korea for providing CUDA-related suggestions. We also thank the anonymous reviewers for their comments to improve the paper presentation.

References

ANDRECUT, M. 2009. Parallel GPU implementation of iterative PCA algorithms. *Journal of Computational Biology 16*, 11.

BARAFF, D. 1992. *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD thesis, Cornell University.

BROWN, S., 2008. The scalable city project. `http://scalablecity.net`.

COHEN, J., LIN, M., MANOCHA, D., AND PONAMGI, M. 1995. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proc. of ACM Interactive 3D Graphics Conference*, 189–196.

COMING, D. S., AND STAADT, O. G. 2006. Kinetic sweep and prune for multi-body continuous motion. *Computers and Graphics 30*, 439–449.

COMING, D. S., AND STAADT, O. G. 2008. Velocity-aligned discrete oriented polytopes for dynamic collision detection. *IEEE Transactions on Visualization and Computer Graphics 14*, 1.

EDELSBRUNNER, H., AND MAURER, H. A. 1981. On the intersection of orthogonal objects. *Inf. Process. Lett. 13*, 177–181.

ERICSON, C. 2005. *Real-Time Collision Detection*. Morgan Kaufmann.

GRAND, S. L. 2008. *GPU Gems 3*. Addison-Wesley, ch. Broad-Phrase Collision Detection with CUDA, 697–721.

HARADA, T., KOSHIZUKA, S., AND KAWAGUCHI, Y. 2007. Smoothed particle hydrodynamics on GPUs. In *Computer Graphics International*.

HARADA, T. 2007. *GPU Gems3*. Addison-Wesley Pearson Education, ch. Real-time Rigid Body Simulation on GPUs.

JOLLIFFE, I. T. 2002. *Principal Component Analysis*. Springer.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH construction on GPUs. In *Eurographics*.

LAUTERBACH, C., MO, Q., AND MANOCHA, D. 2010. gProximity: Hierarchical GPU-based operations for collision and distance queries. In *Eurographics*.

LIN, M., AND MANOCHA, D. 2003. Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*.

LIN, M. C. 1993. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, University of California, Berkeley, CA.

MAZHAR, H., HEYN, T., TASORA, A., AND NEGRUT, D. 2009. Collision detection using spatial subdivision. In *Multibody Dynamics*.

MIRTICH, B. V. 1996. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California, Berkeley.

NVIDIA. 2010. *NVIDIA CUDA Best Practice Guide*.

PONAMGI, M., MANOCHA, D., AND LIN, M. 1995. Incremental algorithms for collision detection between general solid models. In *Proc. ACM Symposium on Solid Modeling and Applications*, 293–304.

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 1992. *Numerical Recipes in C*. Cambridge University Press.

SAMET, H. 2006. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph. 27*, 3, 1–15.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for GPU computing. In *Graphics Hardware*, 97–106.

TANG, M., CURTIS, S., YOON, S.-E., AND MANOCHA, D. 2009. ICCD: Interactive continuous collision detection between deformable models using connectivity-based culling. *IEEE Transactions on Visualization and Computer Graphics 15*, 544–557.

TANG, M., MANOCHA, D., AND TONG, R. 2010. MCCD: Multi-core collision detection between deformable models. *Graphical Models 72*, 2, 7–23.

TERDIMAN, P. 2007. Sweep-and-prune. `http://www.codercorner.com/SAP.pdf`.

TRACY, D. J., BUSS, S. R., AND WOODS, B. M. 2009. Efficient large-scale sweep and prune methods with AABB insertion and removal. In *Proc. IEEE Virtual Reality*, 191–198.

WALD, I. 2007. On fast construction of SAH-based bounding volume hierarchies. In *IEEE/EG Symposium on Interactive Ray Tracing*, 33–40.

WOULFE, M., DINGLIANA, J., AND MANZKE, M. 2007. Hardware accelerated broad phase collision detection for realtime simulations. In *Workshop in Virtual Reality Interactions and Physical Simulation*.

ZERODIN, 2010. Zerodin games. `http://www.zerodingames.com`.

ZHOU, K., HOU, Q., WANG, R., AND GUI, B. 2008. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics*.