Quick-CULLIDE: Fast Inter- and Intra-Object Collision Culling Using Graphics Hardware

Naga K. Govindaraju * Ming C. Lin † Dinesh Manocha ‡

Department of Computer Science
University of North Carolina at Chapel Hill

ABSTRACT

We present a fast collision culling algorithm for performing inter- and intra-object collision detection among complex models using graphics hardware. Our algorithm is based on CULLIDE [8] and performs visibility queries on the GPUs to eliminate a subset of geometric primitives that are not in close proximity. We present an extension to CULLIDE to perform intra-object or self-collisions between complex models. Furthermore, we describe a novel visibility-based classification scheme to compute potentially-colliding and collision-free subsets of objects and primitives, which considerably improves the culling performance. We have implemented our algorithm on a PC with an NVIDIA GeForce FX 6800 Ultra graphics card and applied it to three complex simulations, each consisting of objects with tens of thousands of triangles. In practice, we are able to compute all the self-collisions for cloth simulation up to image-space precision at interactive rates.

CR Categories: I.3.1 [Computing Methodologies]: Hardware Architecture—Graphics Processors; I.3.7 [Computing Methodologies]: Three-Dimensional Graphics and Realism—Visible surface algorithms, animation, virtual reality; I.3.5 [Computing Methodologies]: Computational Geometry and Object Modeling—Geometric algorithms;

Keywords: Collision detection, physically-based simulation, virtual environment, graphics hardware, physical interaction

1 INTRODUCTION

The problem of detecting collisions between real and virtual objects is fundamental in virtual reality (VR), physically-based simulation, and virtual prototyping. As we simulate the motion of avatars and other objects in a virtual environment, it is essential to compute all inter-object and intra-object contacts at interactive rates for generating realistic motion and behavior.

Collision detection and contact determination problems have been investigated for more than three decades. Some of the commonly used algorithms are based on bounding volume hierarchies and they work well on objects undergoing rigid motion. However, these algorithms may not work well on non-rigid or deformable objects due to the overhead of updating the bounding volume hierarchy. Furthermore, it is a major challenge to compute intra-object or *self-collisions* at interactive rates for complex models with substantial amount of deformation. Such scenarios frequently arise in the context of avatar motion, cloth or hair simulation.

*e-mail: naga@cs.unc.edu
†e-mail: lin@cs.unc.edu
‡e-mail: dm@cs.unc.edu

Recently, GPU-based algorithms are increasingly used to perform collision and proximity computations [1, 2, 8, 9, 11, 15, 21, 24, 25, 27]. These algorithms exploit the rasterization capabilities of the GPUs to check for overlaps and involve no precomputation. As a result, these algorithms are applicable to deformable and non-rigid models. However, current GPU-based algorithms are either restricted to closed objects or do not check for self-collisions or unable to perform collision detection at interactive rates for interactive VR applications.

Main Contributions: We present a fast algorithm for collision culling and detection among polygon-soup models using graphics hardware. Similar to CULLIDE [8], our algorithm uses visibility queries to compute a *potentially colliding set* (PCS) of objects or primitives (as shown in Fig. 1) during each frame. Eventually, we perform exact CPU-based collision detection between the triangulated primitives in the PCS. In order to handle complex environments, we present two major extensions over CULLIDE.

- We generalize the formulation of PCS to check for both *inter-object* and *intra-object* collisions. As a result, we can automatically compute self-collisions in complex models at interactive rates.

- We improve the pruning and culling algorithm to compute *collision-free* subsets based on visibility computations. In particular, we present novel relationships between collision culling and visibility-based ordering of objects. The new culling algorithm can significantly lower the number of visibility queries and the rasterization overhead.

The overall collision detection algorithm, Quick-CULLIDE, makes no assumptions about the underlying models, has low bandwidth requirements, and performs inter-object and intra-object collision computations at image-precision. We have implemented Quick-CULLIDE on a 3.4 GHz PC with NVIDIA GeForce FX 6800 Ultra card and applied it to three complex simulations with objects composed of tens of thousands up to $250K$ triangles. Our algorithm is able to compute all the contacts up to image-space precision in tens of milliseconds.

Organization: The rest of the paper is organized as follows. We give a brief survey of prior work on collision detection and hardware accelerated computations in Section 2. We give an overview of CULLIDE in Section 3 and extend it to handle self-collisions. We present the novel culling algorithm in Section 4. In Section 5, we describe its implementation and highlight its performance on three different environments. We also analyze its accuracy and performance.

2 RELATED WORK

The problem of interference and collision detection has been studied for more than three decades and many recent surveys are available [17, 26]. These interactions require fast and reliable interference

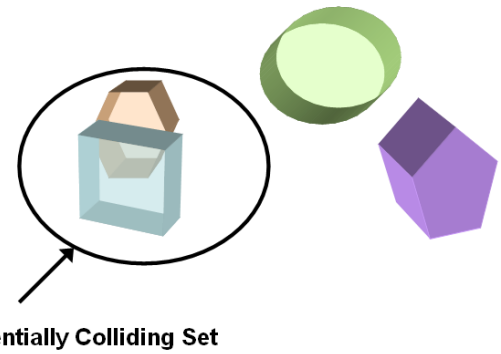


Figure 1: Potentially Colliding Set: In this viewpoint, two of the four objects are in close proximity and belong to the potentially colliding set.

2.1 Rigid Body Algorithms

Many rigid body algorithms use spatial data structures to accelerate interference computations. Common spatial data structures include spatial-partitioning structures and bounding-volume hierarchies [3, 6, 12, 14, 22, 23]. At run-time, the hierarchy is traversed, and bounding volumes are used to cull away portions of objects that are not in close proximity. Typically, these representations are built during the pre-processing stage and are used to accelerate run-time queries.

Efficient algorithms for handling large environments consisting of multiple moving objects have also been designed. These techniques reduce the number of pairwise collision checks by using spatial subdivision algorithms or checking whether the bounding boxes of the objects overlap [2, 4].

2.2 Deformable Models and Cloth Simulation

Hierarchical data structures have also been used to accelerate collision computations between objects undergoing non-rigid motion. These include fast update of hierarchies of axis-aligned bounding boxes (AABBs) [16] and reduced deformable models that can be expressed using linear superposition of precomputed displacement fields [13]. Many specialized algorithms have been proposed for collision detection in cloth simulation [5, 10, 19, 28] and they also check for self-collisions.

2.3 GPU-based Algorithms

Many image-space algorithms utilize graphics processors (GPUs) for interference and collision computations [1, 9, 11, 15, 18, 21, 24, 25, 27]. These algorithms require no pre-processing and therefore are well suited for handling non-rigid motion. Most of these algorithms perform visibility computations to compute overlapping regions between the objects. Objects are rendered from a view-point, and either 2D or 2.5D overlap tests are performed in image-space. Many of these algorithms are limited to closed objects or involve frame-buffer readbacks. Frame-buffer readbacks can be slow on current graphics systems, as they involve graphics pipeline stalls, and are limited by the bandwidth from the GPU to the CPU [8, 15].

Many hybrid algorithms [8, 9, 10, 11] combine some of the benefits of the object-space approaches along with GPU-based accelerations. Heidelberger et al. [9] compute layer depth images (LDIs) on the GPU, use the LDIs for explicit computation of the intersection volumes between two closed objects, and perform vertex-in-volume tests. Recently, the algorithm was extended to check for self-collisions between water-tight objects [10].

3 Collision Culling Using Visibility Queries

In this section, we give an overview of CULLIDE and present an extension of the culling algorithm to check for self-collisions.

3.1 CULLIDE

In this section, we briefly describe CULLIDE [8] which performs visibility computations on GPUs to eliminate geometric primitives that are not in close proximity. The algorithm is based on a sufficient condition for testing whether a separating surface of unit depth complexity exists between a geometric primitive P and a set of primitives S along the view direction. This condition can be tested efficiently based on whether P is fully visible with respect to S along the view direction and is illustrated in Fig. 2. The full-visibility test can be performed efficiently on GPUs using occlusion queries. To test if an object P is fully visible against a set of objects S, CULLIDE first renders S into the frame buffer. Next, it sets the depth function to *GL_GEQUAL* and disables the depth writes. The object P is rendered using an occlusion query. The occlusion query computes the number of pixels that pass the count and determines whether P is fully visible. The full visibility of P is a sufficient condition that P does not overlap with S.

Given n objects that are potentially colliding $P_1, ..., P_n$, CULLIDE performs the full-visibility tests and computes a potentially colliding set (PCS) of objects (as shown in Fig. 1). A linear time two-pass rendering algorithm is used to test if an object P_i is fully visible against the remaining objects, along the view direction.

The algorithm begins with an empty frame buffer and proceeds in two passes as follows:

- **First pass:** Rasterize the primitives in the order $P_1, ..., P_n$ and test if they are fully visible. In this pass, if a primitive P_i is fully visible, then it does not overlap with any of the objects $P_1, ..., P_{i-1}$.

- **Second pass:** Perform the same operations as in the first pass but the order of rendering is changed to $P_n, .., P_1$. In this pass, if a primitive P_i is fully visible, then it does not overlap with any of the objects $P_n, .., P_{i+1}$.

At the end of two passes, if a primitive is fully visible in both the passes, then the primitive does not interfere with the remaining primitives and is pruned from the PCS. The view directions are chosen along the world-space axes and collision culling is performed using orthographic projections. This culling step is performed at an object level, as well as a sub-object level to prune the PCS. Eventually the primitives in the PCS are tested for exact collision using a CPU-based triangle overlap routine. In practice, there exist two major limitations of CULLIDE:

- **Self-collisions:** The pruning performed by CULLIDE is based on the existence of a separating surface between the geometric primitives. Therefore, the PCS computed by CULLIDE could be very conservative on meshes with connected triangles.

- **Culling performance:** The extent of culling depends upon the number of objects occluded by other objects along the view direction. In the worst case, we may prune at most one object even though no object is colliding along the view direction. This affects the culling efficiency as well as performance of the overall algorithm.

We present two novel algorithms to overcome these limitations.

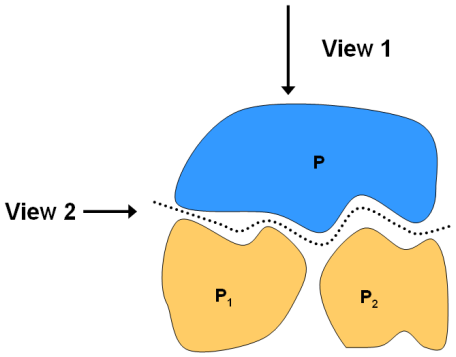


Figure 2: This figure illustrates a close-proximity scenario between an object P and a set of objects $S = \{P_1, P_2\}$. The object P is not colliding with S. Moreover, the object P is fully visible against S along view 1. Therefore, there exists a separating surface between P and S of unit depth complexity along view 1. Note that this surface's existence is a sufficient but not a necessary condition for computing that P is not colliding with S. For example, in view 2, there does not exist a separating surface with unit depth complexity but the objects are not overlapping.

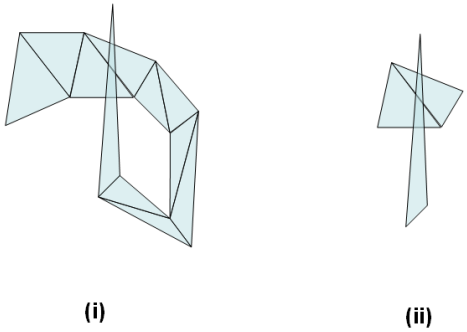


Figure 3: The left image shows an object composed of triangles with shared edges and vertices. The right image shows the self-intersecting triangles in the object. Observe that these self-intersecting triangles do not share an edge or a vertex.

3.2 Self-Collision Culling using GPUs

We extend the PCS computation algorithm in CULLIDE to compute potentially self-intersecting regions of deformable objects. Our algorithm can handle polygonal soups and does not require mesh connectivity information for self-collision pruning. We implicitly compute the connectivity information in the depth buffer while performing collision pruning.

We classify the possible contacts between the geometric primitives including contacts between neighboring primitives into two categories:

- **Touching Contacts:** These contacts occur when the primitives touch each other at a point or along an edge. Contacts computed between neighboring primitives belong to this category.
- **Penetrating Contacts:** These contacts occur when primitives penetrate each other.

The touching contacts often lead to robustness issues in collision detection. Our algorithm ignores these cases and considers only the penetrating contacts for self-collision computations. Self-collision detection algorithms do not compute the collisions between the neighboring primitives. Since we ignore touching contacts, our algorithm does not report collisions between two neighboring primitives that share boundary along a vertex or an edge.

Our self-collision culling algorithm proceeds in the following manner. First, we initialize the PCS used for self-collision culling by including all the potentially self-colliding geometric primitives in an object O. We treat each geometric primitive P as a separate object for collision culling. Next, we perform visibility queries in image space to test whether a triangle is potentially penetrating. A geometric primitive P is not potentially penetrating with a set of rasterized geometric primitives if all the fragments generated by the rasterization of P have depth values *less than or equal* to those of the corresponding pixels in the frame buffer. We use the following lemma to compute the PCS of self-colliding primitives.

Lemma 1: Given n geometric primitives $P_1, P_2, ..., P_n$, a geometric primitive P_i does not belong to the PCS of self-colliding primitives if it does not overlap with $P_1, .., P_{i-1}, P_{i+1}, ..., P_n$, $1 \leq i \leq n$. This test can be easily decomposed as follows: a geometric primitive P_i does not belong to the PCS of self-colliding primitives if it does not overlap with $P_1, .., P_{i-1}$ and with $P_{i+1}, ..., P_n$, $1 \leq i \leq n$.

Proof: Follows trivially from the definition of the PCS and our formulation of self-collisions. □

We use visibility-based computations described in the previous section to prune the PCS of self-colliding primitives efficiently. We classify a geometric primitive as fully visible for self-collision computation based on the following definition.

Definition: A geometric primitive is considered fully visible for *self-collision culling* if it is not potentially penetrating.

Our self-collision culling algorithm uses the two-pass rendering algorithm to perform object-level pruning with the above definition to compute the fully visible status of a primitive. The geometric primitives computed as potentially penetrating are then tested for exact overlap using a CPU-based triangle intersection routine.

4 QUICK-CULLIDE

In this section, we present our novel culling algorithm to compute a potentially colliding set.

4.1 Efficient Culling

The performance of our collision detection algorithm depends upon the culling efficiency obtained by our pruning algorithm. It also depends upon the number of pair-wise overlap tests performed between the geometric primitives in the PCS during the exact collision detection phase. We use the visibility information in our two-pass rendering algorithm to compute *collision-free* sets. A collision-free set is defined as a set of objects that do not collide with each other. We use these collision-free sets to:

- **improve the culling efficiency** by removing redundant visibility computations on objects in the PCS in subsequent pruning iterations,
- **improve rasterization performance** by reducing the number of rendering operations, and
- **reduce the number of pair-wise collision tests** by eliminating collision computations among objects in the collision-free sets.

We classify the objects in the PCS during each iteration of our two-pass rendering algorithm into four categories:

1. **BFV:** These objects are fully visible in both the passes and are pruned from the PCS.

2. **FFV:** These objects are fully visible only in the first pass.

3. **SFV:** Objects in SFV are fully visible only in the second pass.

4. **NFV:** These objects are not fully visible in both the passes.

The objects in each of these sets are ordered based on their rendering order in the first pass of the algorithm. Each object is associated with an index. For e.g., object O_i has index i. Also, the sets BFV, FFV, SFV, and NFV are disjoint. We now present some of the properties of these sets and use them to design an efficient collision culling algorithm.

Lemma 2: FFV and SFV are collision-free sets.
Proof: Let S denote the set FFV and be composed of objects $\{O_S^1, O_S^2, ..., O_S^m\}$. We now prove that no two objects O_S^i and O_S^j in S collide with each other. Without loss of generality, let $i < j$. Then, in the two-pass rendering algorithm, the object O_S^i is rendered prior to the object O_S^j. As the object O_S^j is fully visible with respect to O_S^i, using Lemma 1 in CULLIDE, we conclude that the two objects do not collide. Therefore, FFV is collision-free. The proof for $S = SFV$ is collision-free is similar. □

We use the following lemmas to design a better culling algorithm.

Lemma 3: For each object $O_i \in FFV$, let $S_i = \{O_j, j > i, O_j \in S\}$ where $S = SFV \cup NFV$. If an object $O_i \in FFV$ does not collide with S_i, then it does not collide with any of the objects in SFV or NFV and can be pruned from the PCS.
Proof: Follows from Lemma 1 in CULLIDE [8]. This lemma implies that if an object $O_i \in FFV$ and is fully visible in the second pass of the pruning algorithm, then it provides a sufficient condition to prune the object from the PCS. □

Lemma 4: For each object $O_i \in SFV$, let $S_i = \{O_j, j < i, O_j \in S\}$ where $S = FFV \cup NFV$. If an object $O_i \in SFV$ does not collide with S_i, then it does not collide with any of the objects in FFV or NFV and can be pruned from the PCS.
Proof: Follows from Lemma 1 in CULLIDE [8]. This lemma implies that if an object $O_i \in SFV$ and is fully visible in the first pass of the pruning algorithm, then it provides a sufficient condition to prune the object from the PCS. □

Lemma 5: Let $S_1 = FFV \cup NFV$ be a set ordered by object indices in the increasing order and $S_2 = SFV \cup NFV$ be a set ordered by object indices in the decreasing order. In the two-pass rendering algorithm, if we perform the first pass using objects in S_1 and the second pass using objects in S_2, and an object O_i is fully visible in both the passes, then it does not collide with any of the objects in FFV, SFV or NFV.
Proof: Clearly the object O_i belongs to $NFV = S_1 \cap S_2$ as it is fully visible in both the passes. It is trivial to see that the object does not collide with any of the objects in NFV. We now prove that the object does not collide with any object $O_j \in FFV$.

- If $j < i$, then O_i does not collide with O_j as O_i is fully visible in the first pass.

- If $j > i$, then O_j does not collide with O_i as $O_j \in FFV$.

Similarly, we prove that the O_i does not collide with the objects in SFV. □

Using Lemmas 3, 4, and 5, we come up with an efficient culling algorithm. We modify the first pass of CULLIDE as follows:

- For each object O_i in PCS, i=1,..,n
  - If $O_i \in SFV$ or $O_i \in NFV$, test whether the object is fully visible using an occlusion query.
  - If $O_i \in FFV$ or $O_i \in NFV$, render the object into the frame buffer.

- For each object O_i in PCS, i=1,..,n
  - If $O_i \in SFV$ or $O_i \in NFV$, and the occlusion query determines O_i as fully visible
    - If $O_i \in SFV$, then tag O_i as a member of BFV.
    - If $O_i \in NFV$, then tag O_i as a member of FFV.

Similarly, we modify the second pass as follows:

- For each object O_i in PCS, i=n,..,1
  - If $O_i \in FFV$ or $O_i \in NFV$, test whether the object is fully visible using an occlusion query.
  - If $O_i \in SFV$ or $O_i \in NFV$, render the object into the frame buffer.

- For each object O_i in PCS, i=n,..,1
  - If $O_i \in FFV$ or $O_i \in NFV$, and the occlusion query determines O_i as fully visible
    - If $O_i \in FFV$, then tag O_i as a member of BFV.
    - If $O_i \in NFV$, then tag O_i as a member of SFV.

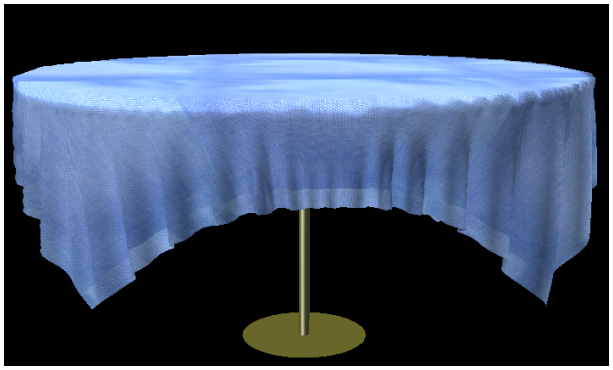
The modified algorithm computes the different sets in the following manner:

1. Objects that are fully visible in both the passes:
   - This subset of objects belonging to NFV are pruned from the PCS (based on Lemma 5).
2. Objects that are fully visible in the first pass:
   - **NFV:** These objects are removed from NFV and placed in FFV.
   - **SFV:** These objects are removed from the PCS (based on Lemma 4).
   - **FFV:** Visibility computations are not performed for these objects in this pass.
3. Objects that are fully visible in the second pass:
   - **NFV:** These objects are removed from NFV and placed in SFV.
   - **FFV:** These objects are removed from the PCS (based on Lemma 3).
   - **SFV:** Visibility computations are not performed for these objects in this pass.

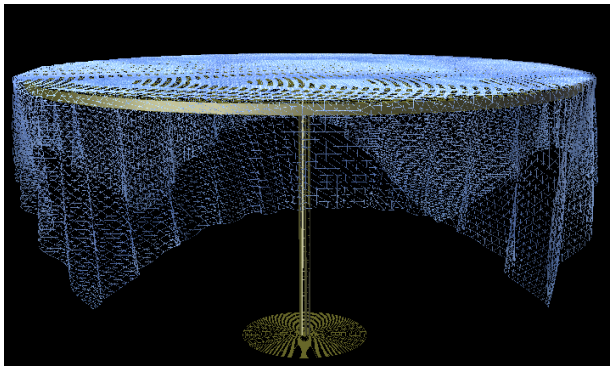
The improved culling algorithm reduces the number of rendering operations and occlusion queries each by *sizeof($FFV \cup SFV$)*, as compared to CULLIDE.

4.2 Collision Detection

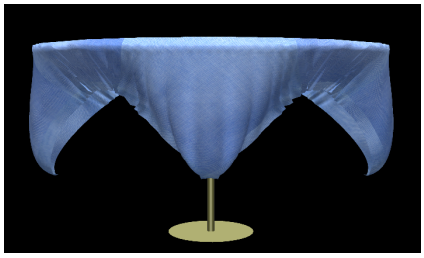
Our collision detection algorithm, Quick-CULLIDE, proceeds in three steps. First we compute the PCS at the object level using our improved culling algorithm. We use sweep-and-prune [4] on the PCS to compute the overlapping pairs at the object level. Next we compute the PCS at the sub-object level and the overlapping pairs. Finally, we perform exact interference tests between the triangles on the CPU [20]. For self-collision computations, we use the algorithm described in Section 3.2.



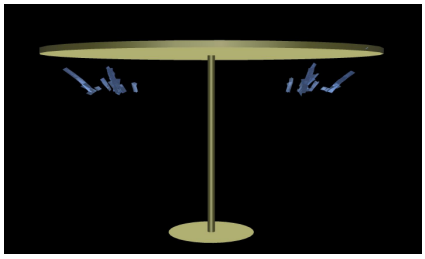
(a) Cloth simulation



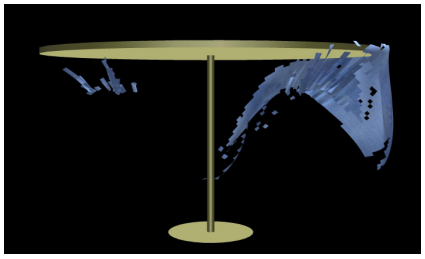
(b) Wireframe of the cloth (Fig. a)



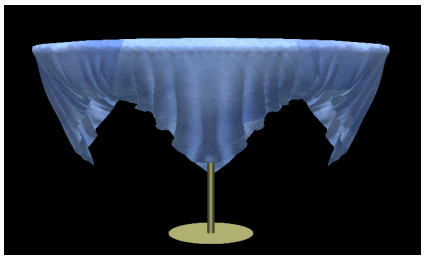
(c) One snapshot in the time sequence



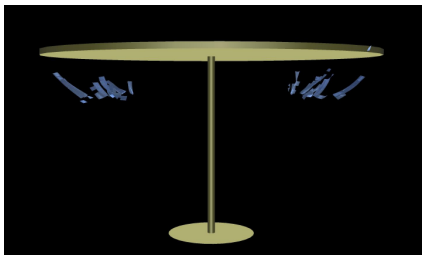
(d) PCS computed using Quick-CULLIDE in the snapshot in Fig. (c)



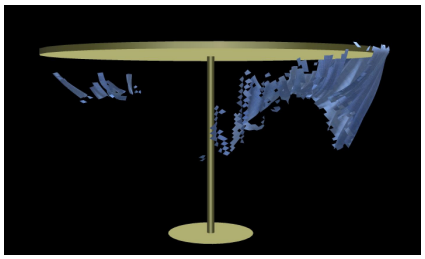
(e) PCS computed using CULLIDE-based culling algorithm applied to the snapshot in Fig. (c)



(f) Another snapshot in the time sequence



(g) PCS computed using Quick-CULLIDE in the second snapshot (Fig. f)



(h) PCS computed using CULLIDE-based culling algorithm applied to the second snapshot (Fig. f)

Figure 4: Cloth simulation: These sequence of images show various instances of a cloth piece falling over a circular table. The cloth is modeled using 20K triangles as shown in Fig. (a). A wireframe of the cloth is shown in Fig. (b). The simulation is performed using our self-collision detection algorithm Quick-CULLIDE and we demonstrate its benefit over CULLIDE. Figs. (d) & (e), and (g) & (h) show the PCS computed using (Quick-CULLIDE) & (CULLIDE), respectively, for the instances shown in Figures (c) & (f), respectively. Quick-CULLIDE is able to perform more culling and computes a PCS with size smaller by one order of magnitude in comparison to the size of the PCS computed by CULLIDE. The average collision detection time is 21 msec using Quick-CULLIDE.

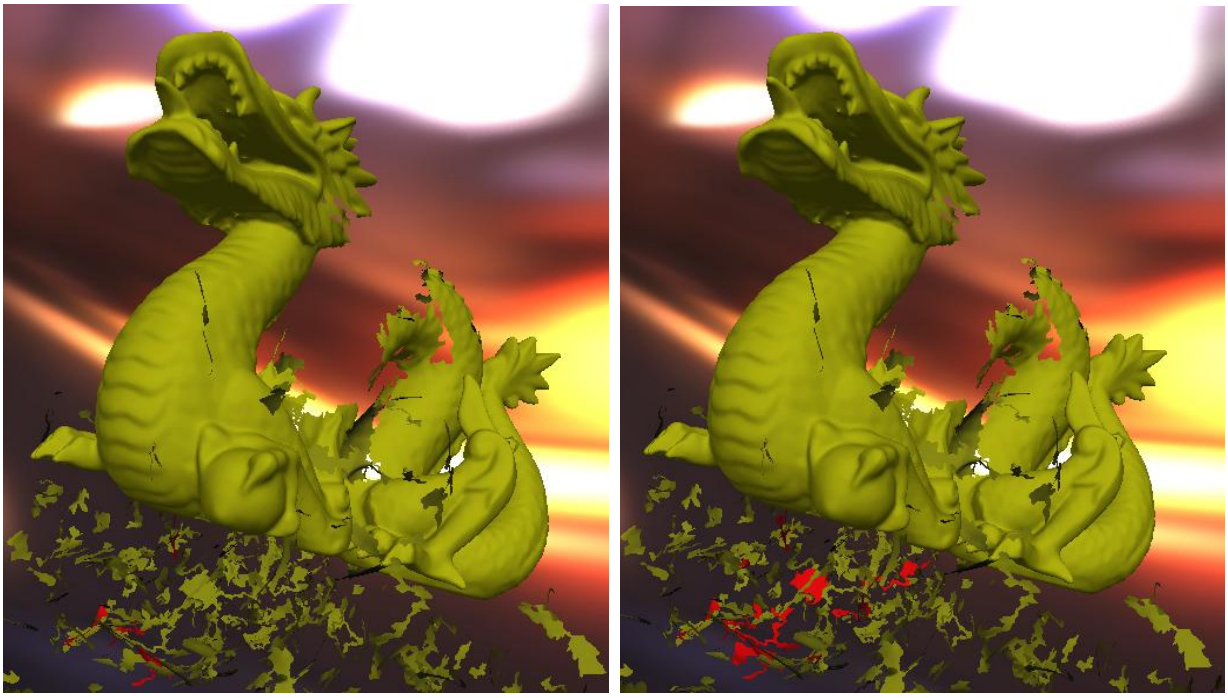
5 Implementation and Performance

We have implemented our collision detection algorithm using OpenGL on a PC with a 3.4 GHz Intel Pentium IV CPU, an NVIDIA GeForce FX 6800 Ultra GPU, and 2 GB of main memory. The PC is running on the Windows XP operating system and the data is transferred from the CPU to the GPU using an AGP 8X interface. We have performed collision culling on the GPU using an offscreen buffer with a viewport resolution of $4K \times 4K$. The rendering operations are performed efficiently using vertex arrays and we use *GL_NV_occlusion_query* for performing the visibility queries asynchronously. We have tested our system on three complex environments and compared the level of culling in Quick-CULLIDE with that in CULLIDE.

- **Cloth Simulation:** We have implemented a cloth simulator and used Quick-CULLIDE to check for inter- and intra-object collisions. The cloth is represented using a rectangular grid with 20K triangles and the simulation is performed using the verlet integration method. In this simulation, the cloth falls on a circular table and as the simulation progresses, the cloth generates several folds and wrinkles. We have performed collision culling at the triangle-level using two axis-aligned views (along X and Z axes). Fig. 4 shows a sequence of the snapshots of the cloth simulation based on time. The average collision pruning time is $20 - 22$ msec.

- **Breaking Objects:** In this environment, a bunny composed of 35K polygons repeatedly falls on a dragon composed of 250K polygons and fractures the dragon. During the course of the simulation, hundreds of new object pieces are generated into the scene as the bunny breaks the dragon. We have used three axis-aligned views for performing the collision culling. Our algorithm is able to detect all the collisions within 25 msec.

- **Objects Undergoing Non-Rigid Motion:** In this simulation, several deformable leaves fall from the tree, and collide with each other and the branches of the tree as shown in Fig. 6. The average collision detection time is nearly 25 msec per frame.



(a) PCS (shown in red) computed using Quick-CULLIDE (b) PCS (shown in red) computed using CULLIDE

Figure 5: Breaking objects simulation: In this simulation, a bunny composed of 35K triangles fractures a dragon composed of 250K triangles. As the simulation progresses, hundreds of new objects in close proximity are introduced into the scene. Our algorithm Quick-CULLIDE is able to compute all the collisions within 25ms on a PC with a NVIDIA GeForce FX 6800 Ultra GPU. Moreover, our algorithm is able to perform more pruning and computes a more compact PCS in comparison to CULLIDE. Figures (a) and (b) show the PCS - rendered in color red, computed using Quick-CULLIDE and CULLIDE, respectively.

5.1 Analysis

In this section, we analyze the culling efficiency and the performance obtained by our algorithm.

5.1.1 Culling Efficiency

We have compared the culling efficiency of our algorithm Quick-CULLIDE with an implementation of CULLIDE within the cloth simulation system. In order to compare the algorithms, we have used the same collision culling resolution and the view directions in both the algorithms. Fig. 7 shows the comparison between the culling efficiency obtained by Quick-CULLIDE vs. CULLIDE. In this simulation, Quick-CULLIDE reduces the size of the PCS, as well as the number of overlapping pairs by nearly one order of magnitude as compared to CULLIDE.

5.1.2 Performance

We have compared the performance of Quick-CULLIDE with CULLIDE on each of the three simulations. Fig. 8 shows a comparison of the performance. In this simulation, we observe that Quick-CULLIDE can compute all the overlapping pairs of triangles faster than CULLIDE. Table 2 summarizes the average performance obtained by Quick-CULLIDE and CULLIDE on each of the three benchmarks.

5.1.3 Factors

A number of factors affect the performance, as well as the culling efficiency of Quick-CULLIDE. These include:

Simulation	Average PCS size (triangles)	
	Quick-CULLIDE	CULLIDE
Cloth	210	1340
Breaking objects	1400	3600
Non-rigid motion	450	1200

Table 1: This table highlights the average PCS size computed by Quick-CULLIDE and CULLIDE on the three simulations. We observe that Quick-CULLIDE is able to prune $4 - 10$ times more non-overlapping triangles as compared to CULLIDE upto image-precision resolution.

- **Depth complexity:** The extent of culling obtained by our algorithm along a view direction depends upon the number of objects that project onto the screen-space along that view direction.

- **Order of rendering:** Our collision culling algorithm works best in scenes where the objects are rendered in a back-to-front order along the view direction. The level of culling obtained by Quick-CULLIDE is always better than CULLIDE.

- **Number of views:** The culling efficiency also depends upon the number of views, as well as the choice of the view directions used to perform collision detection. As the number of views used for collision detection increase, the pruning efficiency of CULLIDE approaches that of Quick-CULLIDE. However, as the number of views increases, the cost of collision culling increases as well.

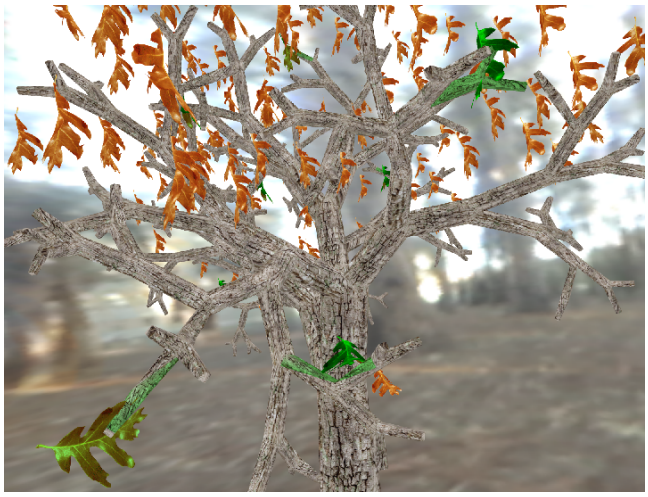


Figure 6: Simulation with non-rigid objects: In this simulation, many leaves are falling from the tree. They undergo non-rigid motion and collide with other leaves and branches of the tree. Our algorithm Quick-CULLIDE is able to compute all the collisions reliably within 25 msec on a PC with NVIDIA GeForce FX 6800 Ultra GPU.

Simulation	Average collision time (in msec)	
	Quick-CULLIDE	CULLIDE
Cloth	21	30
Breaking objects	25	35
Non-rigid motion	25	31

Table 2: This table highlights the real-time performance obtained by using Quick-CULLIDE and CULLIDE on the three simulations. We observe that Quick-CULLIDE is faster than CULLIDE in each of the three simulations.

5.2 Comparison with Other Approaches

In this section, we compare our algorithm with two other GPU-based self-collision detection algorithms [1, 9]. Our algorithm is able to compute self-collisions among general deformable models whereas [1, 9] are designed to work well for closed or water-tight models. Therefore, it is difficult to perform direct timing comparisons. Instead, we compare some of the features of our algorithm with these algorithms.

Our algorithm is able to handle self-collisions and collisions among a large number of objects, whereas [1, 9] work on pairs of objects. The GPU-based implementations of Heidelberger et al. [9], and Baciu and Wong [1] readback the framebuffer which can be expensive on current PCs. In contrast, our algorithm does not perform framebuffer readbacks. Moreover, we perform collision culling at a very high image-space resolution of $4K \times 4K$ within just a few milliseconds (< 40 msec). Most of the earlier approaches used a lower image-space resolution. Finally, our algorithm is well-suited to handle deformable, breaking, and non-rigid geometry, as well as polygon-soup models.

5.3 Limitations

Our algorithm has some limitations. Our pruning algorithm computes a PCS of geometric primitives, and does not compute the overlap information or the extent of penetration. However, a post-processing step can be added to extract the information. The precision of our self-collision culling algorithm is limited to image resolution and is lower than of object-space algorithms. Also, our self-collision culling algorithm ignores touching contacts. Our pruning algorithm achieves best performance when the objects are rendered in a back-to-front order. Furthermore, the culling efficiency depends on the relative object configurations, and the depth complexity of the scene along the view.

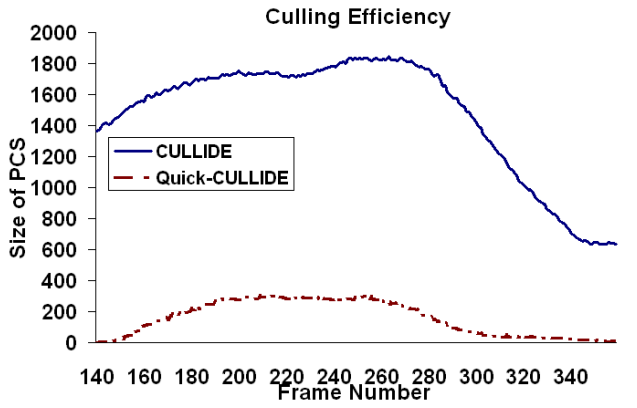


Figure 7: Comparison of pruning performance between Quick-CULLIDE and CULLIDE: This graph shows the number of triangles in the PCS computed using Quick-CULLIDE and CULLIDE respectively on the cloth simulation. We observe that Quick-CULLIDE is able to reduce the PCS size by nearly a magnitude in comparison to CULLIDE.

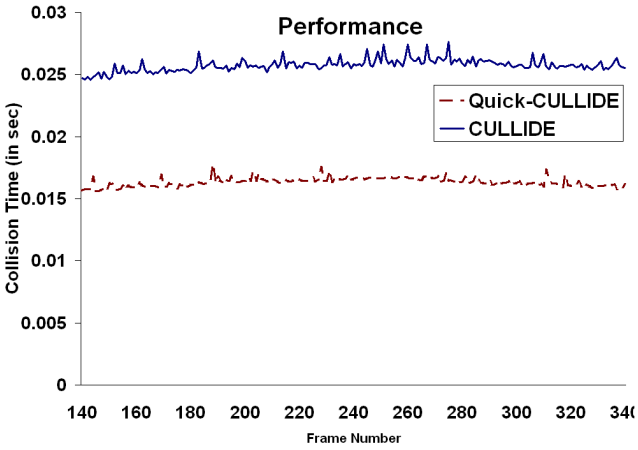


Figure 8: Performance comparison between Quick-CULLIDE and CULLIDE: This graph compares the time taken by Quick-CULLIDE vs. CULLIDE to compute all the collisions during each frame in cloth simulation.

6 Conclusions and Future Work

We have presented an efficient GPU-based collision culling algorithm for performing inter- and intra-object collision detection. Our algorithm uses a novel visibility-based classification to compute potentially colliding and collision-free subsets of geometric primitives. The preliminary comparisons of our algorithm with CULLIDE indicate up to an order of magnitude improvement in culling efficiency for some scenarios. The collision computations are performed upto image-resolution and can be improved to single precision floating-point computation using our recent techniques [7].

Many avenues exist for future research. It may be possible to

use the collision-free sets for pair-wise overlap computation. We would like to extend these algorithms for other proximity computations, including distance and penetration depth computation, as well as exploring the new programmability features of GPUs to further improve the performance of our algorithm.

ACKNOWLEDGEMENTS

Our work was supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134, 982167 and 0118743, ONR Contracts N00014-01-1-0067 and N00014-01-1-0496, DARPA Contract N61339-04-C-0043 and Intel. We would like to thank NVIDIA corporation for their hardware and driver support. We would like to acknowledge Kelly Ward for video editing and members of UNC Walkthrough and GAMMA groups for useful discussions.

REFERENCES

[1] G. Baciu and S.K. Wong. Image-based techniques in a hybrid collision detector. *IEEE Trans. on Visualization and Computer Graphics*, 2002.

[2] G. Baciu, S.K. Wong, and H. Sun. Recode: An image-based collision detection algorithm. *Proc. of Pacific Graphics*, pages 497–512, 1998.

[3] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. *Proc. SIGMOD Conf. on Management of Data*, pages 322–331, 1990.

[4] J. Cohen, M. Lin, D. Manocha, and M. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proc. of ACM Interactive 3D Graphics Conference*, pages 189–196, 1995.

[5] F. Cordier and N. Magnenat-Thalmann. Real-time animation of dressed virtual humans. *Eurographics*, 21(3), 2002.

[6] S. Gottschalk, M. Lin, and D. Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96*, pages 171–180, 1996.

[7] N. Govindaraju, M. Lin, and D. Manocha. Fast and reliable collision detection using graphics hardware. *Proc. of ACM VRST*, 2004.

[8] N. Govindaraju, S. Redon, M. Lin, and D. Manocha. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. *Proc. of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 25–32, 2003.

[9] B. Heidelberger, M. Teschner, and M. Gross. Real-time volumetic intersections of deforming objects. *Proc. of Vision, Modeling and Visualization*, 2003.

[10] B. Heidelberger, M. Teschner, and M. Gross. Detection of collisions and self-collisions using image-space techniques. *Journal of WSCG*, 12(3), 2004.

[11] K. Hoff, A. Zaferakis, M. Lin, and D. Manocha. Fast and simple 2d geometric proximity queries using graphics hardware. *Proc. of ACM Symposium on Interactive 3D Graphics*, pages 145–148, 2001.

[12] P. M. Hubbard. Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*, October 1993.

[13] D. L. James and D. K. Pai. Bd-tree: Output-sensitive collision detection for reduced deformable models. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)*, 2004.

[14] J. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. on Visualization and Computer Graphics*, 4(1):21–37, 1998.

[15] D. Knott and D. Pai. Cinder: Collision and interference detection in real-time using graphics hardware. *Proc. of Graphics Interface*, pages 73–80, 2003.

[16] Thomas Larsson and Tomas Akenine-Moller. Collision detection for continuously deforming bodies. In *Eurographics*, 2001.

[17] M. Lin and D. Manocha. Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*, 2003.

[18] J. C. Lombardo, M.-P. Cani, and F. Neyret. Real-time collision detection for virtual surgery. *Proc. of Computer Animation*, 1999.

[19] J. Mezger, S. Kimmerle, and I. Etzmuβ. Hierarchical techniques in collision detection for cloth animation. *Journal of WSCG*, 11(2):322–329, 2003.

[20] T. Moller. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2), 1997.

[21] K. Myszkowski, O. G. Okunev, and T. L. Kunii. Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer*, 11(9):497–512, 1995.

[22] M. Ponamgi, D. Manocha, and M. Lin. Incremental algorithms for collision detection between solid models. *IEEE Transactions on Visualization and Computer Graphics*, 3(1):51–67, 1997.

[23] S. Quinlan. Efficient distance computation between non-convex objects. In *Proceedings of International Conference on Robotics and Automation*, pages 3324–3329, 1994.

[24] J. Rossignac, A. Megahed, and B.D. Schneider. Interactive inspection of solids: cross-sections and interferences. In *Proceedings of ACM Siggraph*, pages 353–60, 1992.

[25] M. Shinya and M. C. Forgue. Interference detection through rasterization. *The Journal of Visualization and Computer Animation*, 2(4):131–134, 1991.

[26] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino. Collision detection for deformable objects. *Eurographics*, 2004.

[27] T. Vassilev, B. Spanlang, and Y. Chrysanthou. Fast cloth animation on walking avatars. *Computer Graphics Forum (Proc. of Eurographics'01)*, 20(3):260–267, 2001.

[28] P. Volino and N. Magnetat Thalmann. Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. *Computer Graphics Forum (EuroGraphics Proc.)*, 13(3):155–166, 1994.