

CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments using Graphics Hardware

Naga K. Govindaraju, Stephane Redon, Ming C. Lin and Dinesh Manocha

Department of Computer Science, University of North Carolina at Chapel Hill, U.S.A.
<http://gamma.cs.unc.edu/CULLIDE>

Abstract

We present a novel approach for fast collision detection between multiple deformable and breakable objects in a large environment using graphics hardware. Our algorithm takes into account low bandwidth to and from the graphics cards and computes a potentially colliding set (PCS) using visibility queries. It involves no precomputation and proceeds in multiple stages: PCS computation at an object level and PCS computation at sub-object level, followed by exact collision detection. We use a linear time two-pass rendering algorithm to compute each PCS efficiently. The overall approach makes no assumption about the input primitives or the object's motion and is directly applicable to all triangulated models. It has been implemented on a PC with NVIDIA GeForce FX 5800 Ultra graphics card and applied to different environments composed of a high number of moving objects with tens of thousands of triangles. It is able to compute all the overlapping primitives between different objects up to image-space resolution in a few milliseconds.

1. Introduction

High-performance 3D graphics systems are becoming as ubiquitous as floating-point hardware. They are now a part of almost every personal computer or game console. In addition, graphics hardware is becoming more programmable and is increasingly used as a co-processor for many diverse applications. These include ray-tracing, intersection computations, simulation of dynamic phenomena, atmospheric effects and scientific computations.

In this paper, we mainly address the problem of collision detection among moving objects, either rigid or deformable, using graphics processing units (GPUs). Collision detection is an important problem in computer graphics, game development, virtual environments, robotics and engineering simulations. It is often one of the major computational bottlenecks in dynamic simulation of complex systems. Collision detection has been well studied over the last few decades. However, most of the efficient algorithms are limited to rigid bodies and require preprocessing. Although some algorithms have been proposed for deformable models, either they are limited to simple objects or closed sets, or they are designed for specialized models (eg. cloth).

Many algorithms exploiting graphics hardware capabilities have been proposed for collision queries and proximity computations^{1, 2, 8, 9, 13, 18, 20, 22, 23}. At a broad level, these algorithms can be classified into two categories: use of depth and stencil buffer techniques for computing interference and fast computation of distance fields for proximity queries. These algorithms perform image-space computations, and are applicable to rigid and deformable models. However, they have three main limitations:

- **Bandwidth Issues:** Although graphics hardware is progressing at a rate faster than Moore's Law, the bandwidth to and from the graphics cards is not increasing as fast as computational power. Furthermore, many algorithms readback the frame-buffer or depth buffer during each frame. These readbacks can be expensive on commodity graphics hardware, e.g. it takes 50 milliseconds to read back the $1K \times 1K$ depth buffer on a Dell 530 Workstation with NVIDIA GeForce 4 card.
- **Closed Objects:** Many of these algorithms are mainly restricted to closed objects, as they use graphics hardware stencil operations to perform virtual ray casting operations and determine whether a point is inside or outside.
- **Multiple Object-Pair Culling:** Most of the current algo-

gorithms are designed for a pair of objects and not intended for large environments composed of multiple moving objects.

Main Contribution: We present a novel algorithm for collision or interference detection among multiple moving objects in a large environment using graphics hardware. Given an environment composed of triangulated objects, our algorithm computes a *potentially colliding set (PCS)*. The PCS consists of objects that are either overlapping or are in close proximity. We use visibility computations to prune the number of objects in the PCS. This is based on a linear time two-pass rendering algorithm that traverses the list of objects in forward and reverse order. The visibility relationships are computed using image-space occlusion queries, which are supported on current graphics processors.

The pruning algorithm proceeds in multiple stages. Initially it compute a PCS of objects. Next it considers all *sub-objects* (i.e. bounding boxes, groups of triangles, or individual triangles) of these objects and computes a PCS of sub-objects. Finally, it uses an exact collision detection algorithm to compute the overlapping triangles. The complexity of the algorithm is a linear function of the input and output size, as well as the size of PCS after each stage. Its accuracy is governed by the image precision and depth buffer resolution. Since there are no depth-buffer readbacks, it is possible to perform the image-space occlusion queries at a higher resolution without significant degradation in performance. The additional overhead is in terms of fill-rate and not the bandwidth.

We have implemented the algorithm on a Dell 530 Workstation with NVIDIA GeForce FX 5800 Ultra graphics card and a Pentium IV processor, and have applied it to three complex environments: 100 moving deformable objects in a cube, 6 deforming torii (each composed of 20,000 polygons), and two complex breakable objects composed of 35,000 and 250,000 triangles. In each case, the algorithm can compute all the overlapping triangles between different objects in just a few milliseconds.

Advantages: As compared to earlier approaches, our algorithm offers the following benefits. It is relatively simple and makes no assumption about the input model. It can even handle “polygon soups”. It involves no precomputation or additional data structures (e.g. hierarchies). As a result, its memory overhead is low. It can easily handle deformable models and breakable objects with deforming geometry and changing topology. Our algorithm doesn’t make any assumptions about object motion or temporal coherence between successive frames. It can efficiently compute all the contacts among multiple objects or a pair of highly tessellated models at interactive rates.

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 PCS computation using visibility

queries in Section 3. We present our two-stage algorithm in Section 4. In Section 5, we describe its implementation and highlight its performance on different environments. We also analyze its accuracy and performance.

2. Previous Work

In this section, we give a brief survey of prior work on collision detection and graphics-hardware-accelerated approaches.

2.1. Collision Detection

Typically for a simulated environment consisting of multiple moving objects, collision queries consist of two phases: the “broad phase” where collision culling is performed to reduce the number of pairwise tests, and the “narrow phase” where the pairs of objects in proximity are checked for collision^{3, 10}.

Algorithms for narrow phase can be further subdivided into efficient algorithms for convex objects and general-purpose algorithms based on spatial partitioning and bounding volume hierarchies for polygonal or spline models (please see survey in^{14, 15, 16, 19}). However, these algorithms often involve precomputation and are mainly designed for rigid models.

2.2. Acceleration Using Graphics Hardware

Graphics hardware has been increasingly utilized to accelerate a number of geometric computations, including visualization of constructive solid geometry models^{5, 21}, interferences and cross-sections^{1, 2, 18, 20, 22}, distance fields and proximity queries^{7, 8}, Minkowski sums^{11, 12}, and specialized algorithms including collision detection for cloth animation²³ and virtual surgery¹⁷. All of these algorithms perform image-space computations and involve no preprocessing. As a result, they are directly applicable to rigid as well as deformable models. However, the interference detection algorithms based on depth and stencil buffers^{2, 18, 20} are limited to closed objects. The approaches based on distance field computations^{7, 8} can also perform distance and penetration computation between two objects. But, they require depth-buffer readbacks, which can be expensive on commodity graphics hardware.

3. Collision Detection Using Visibility Queries

In this section, we give an overview of our collision detection algorithm. We show how a PCS can be computed using image-space visibility queries, followed by exact collision detection between the primitives.

Given an environment composed of n objects, O_1, O_2, \dots, O_n . We assume that each object is represented as a collection of triangles. Our goal is to check

which objects overlap and also compute the overlapping triangles in each intersecting pair. In this paper, we restrict ourselves to inter-object collisions. Our algorithm makes no assumption about the motion of objects or any coherence between successive frames. In fact, the number of objects as well as the number of triangles in each object can change between successive frames.

3.1. Potentially Colliding Set (PCS)

We compute a PCS of objects that are either overlapping or are in close proximity. If an object O_i does not belong to the PCS, it implies that O_i does not collide with any object in the PCS. Based on this property, we can prune the number of object pairs that need to be checked for exact collision. This is similar to the concept of computing the potentially visible set (PVS) of primitives from a viewpoint for occlusion culling ⁴.

We perform visibility computations between the objects in image space to check whether they are potentially colliding or not. Given a set S of objects, we test the relative visibility of an object O with respect to S using an image-space visibility query. The query checks whether any part of O is occluded by S . It rasterizes all the objects belonging to S . O is considered *fully-visible* if all the fragments generated by rasterization of O have a depth value less than the corresponding pixels in the frame buffer. We do not consider self-occlusion of an object (O) in checking its visibility status. We use the following lemma to check whether O is overlapping with any object in S .

Lemma 1: *An object O does not collide with a set of objects S if O is fully-visible with respect to S .*

Proof: The proof of this lemma is quite obvious. If O is overlapping with any object in S , then some part of O is occluded by S . We also note that this property is independent of the projection plane.

The accuracy of the algorithm is governed by the underlying precision of the visibility query. Moreover, this lemma only provides a sufficient condition and not a necessary condition.

3.2. Visibility Based Pruning

We use Lemma 1 for PCS computation. Given n objects O_1, \dots, O_n , we check if O_i potentially intersects with at least one of $O_1, \dots, O_{i-1}, O_{i+1}, \dots, O_n$, $1 \leq i \leq n$. Instead of checking all possible pairs (which can be $O(n^2)$), we use the following lemma to design a linear-time algorithm to compute a conservative set.

Lemma 2: *Given n objects O_1, O_2, \dots, O_n , an object O_i does not belong to PCS if it does not intersect with $O_1, \dots, O_{i-1}, O_{i+1}, \dots, O_n$, $1 \leq i \leq n$. This test can be easily decomposed as: an object O_i does not belong to PCS if*

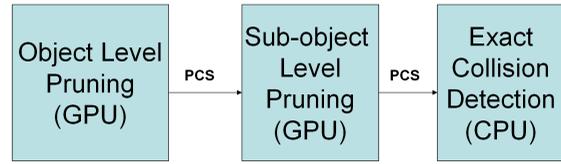


Figure 1: System Architecture: The overall pipeline of the collision detection algorithm for large environments

it does not intersect with O_1, \dots, O_{i-1} and with O_{i+1}, \dots, O_n , $1 \leq i \leq n$.

Proof: Follows trivially from the definition of PCS.

We use Lemma 2 to check if an object belongs to the PCS. Our algorithm uses a two pass rendering approach to compute the PCS. In the first pass, we check if O_i potentially intersects with at least one of the objects O_1, \dots, O_{i-1} . In the second pass, we check if it potentially intersects with one of O_{i+1}, \dots, O_n . If an object does not intersect in either of the two passes, then it does not belong to the PCS.

Each pass requires the object representation for an object to be rendered twice. We can either render all the triangles used to represent an object or a bounding box of the object. Initially, the PCS consists of all the objects in the scene. We perform these two passes to prune objects from the PCS. Furthermore, we repeat the process by using each coordinate axis as the axis of projection to further prune the PCS. We use Lemma 1 to check if an object potentially intersects with a set of objects or not.

It should be noted that our GPU based pruning algorithm is quite different from algorithms that prune PCS using 2D overlap tests. Our algorithm does not perform frame-buffer readbacks and computes a PCS that is less conservative an algorithm based on 2D overlap tests.

3.3. Localizing the Overlapping Features

Many applications need to compute the exact overlapping features (e.g. triangles) for collision response. We initially compute the PCS of objects based on the algorithm highlighted above. Instead of testing each object pair in the PCS for exact overlap, we again use the visibility formulation to identify the potentially intersecting regions among the objects in the PCS. Specifically, we use a fast global pruning algorithm to localize these regions of interest.

We decompose each object into sub-objects. A sub-object can be a bounding box, a group of k triangles (say a constant k), or a single triangle. We extend the approach discussed in Section 3.1 to sub-object level and compute the potentially intersecting regions based on the following lemma.

Lemma 3: *Given n objects O_1, O_2, \dots, O_n and each object O_i is composed of m_i sub-objects $T_1^i, T_2^i, \dots, T_{m_i}^i$, a sub-object T_k^i of O_i does not belong to the object's potentially intersecting region if it does not intersect with the sub-objects*

of $O_1, \dots, O_{i-1}, O_{i+1}, \dots, O_n$, $1 \leq i \leq n$. This test can be decomposed as, a sub-object T_k^i of O_i does not belong to the potentially intersecting region of the object if it does not intersect with the sub-objects of O_1, \dots, O_{i-1} and O_{i+1}, \dots, O_n , $1 \leq i \leq n$.

Proof: Follows trivially from Lemma 2.

In this case, we again use visibility queries to resolve the intersections among sub-objects of different objects. However, we do not check an object for self-intersections or self-occlusion and therefore, do not perform visibility queries among the sub-objects of the same parent object.

3.4. Collision Detection

Our overall algorithm performs pruning at two stages, object level and sub-object level, and eventually checks the primitives for exact collision.

- **Object Pruning:** We perform object level pruning by computing the PCS of objects. We first use AABBs of the objects to prune this set. Next we use the exact triangulated representation of the objects to further prune the PCS. If the PCS is large, we use the sweep-and-prune algorithm³ to compute potentially colliding pairs and decompose the PCS into smaller subsets.
- **Sub-Object Pruning:** We perform sub-object pruning to identify potential regions of each object in PCS that may be involved in collision detection.
- **Exact Collision Detection:** We perform exact triangle-triangle intersection tests on the CPU to check if two objects collide or not.

The architecture of the overall system is shown in Fig. 1, where the first two stages are performed using image-space visibility queries (on the GPU) and the last stage is performed on the CPU.

4. Interactive Collision Detection

In this section, we present our overall collision detection algorithm for computing all the contacts between multiple moving objects in a large environment. It uses the visibility pruning algorithm described in Section 3.2. The overall algorithm is general and applicable to all environments. We also highlight many optimizations and the visibility queries used to accelerate the performance of our algorithm.

4.1. Pruning Algorithm

We use a two-pass rendering algorithm based on the visibility formulation defined in Section 3.2 to perform linear time PCS pruning. In particular, we use Lemma 2 to compute the PCS. In the first pass, we clear the depth buffer and render the objects in the order O_1, \dots, O_n along with image space occlusion queries. In other words, for each object in O_1, \dots, O_n , we render the object and test if it is fully visible

with respect to the objects rendered prior to it. In the second pass, we clear the depth buffer and render the objects in the reverse order O_n, O_{n-1}, \dots, O_1 along with image space occlusion queries. We perform the same operations as in the first pass while rendering each object. At the end of each pass, we test if an object is fully visible or not. An object classified as fully-visible in both the passes does not belong to the PCS.

4.2. Visibility Queries

Our visibility based PCS computation algorithm is based on hardware visibility query which determines if a primitive is *fully-visible* or not. Current commodity graphics hardware supports an image-space occlusion query that checks whether a primitive is visible or not. These queries scan-convert the specified primitives and check if the depth of any pixel changes. Various implementations are provided by different hardware vendors and each implementation varies in its performance as well as functionality. Some of the well-known occlusion queries based on the OpenGL extensions include the GL_HP_occlusion_test (http://oss.sgi.com/projects/ogl-sample/registry/HP/occlusion_test.txt) and the NVIDIA OpenGL extension GL_NV_occlusion_query (http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion_query.txt). The GL_HP_occlusion_test returns a boolean answer after checking if any incoming fragment passed the depth test, whereas the GL_NV_occlusion_query returns the number of incoming fragments that passed the depth test.

We need a query that tests if all the incoming fragments of a primitive have a depth value *less* than the depth of the corresponding fragments in the frame buffer. In order to support such a query, we change the depth test to pass only if the depth of the incoming fragment is greater than or equal to the depth of the corresponding fragment in the frame buffer. With this depth comparison function, we use an image space occlusion query to test if a primitive is not visible when rendered against the depth buffer. If the primitive is classified as not visible, then each incoming fragment has a depth value less than the corresponding depth value in the frame buffer, thus providing a visibility query to test if a primitive is fully visible. Note that we need to disable the depth writes so that the change of depth function does not affect the depth buffer. We refer to these queries as *fully-visibility* queries in the rest of the paper. These queries can sometimes stall the graphics pipeline while waiting for results. We describe techniques to avoid these stalls (discussed in section 4.5).

Bandwidth Requirements: Occlusion queries can be performed at the rate of rasterization hardware and involve very low bandwidth requirements in comparison to frame buffer readbacks. If we perform n occlusion queries, we readback n integers for a total of $4n$ bytes, sent to the host CPU from the GPU. Moreover, the bandwidth requirement for n occlusion queries is independent of the resolution of the frame buffer.

4.3. Multiple Level Pruning

We extend the visibility pruning algorithm to sub-objects, to identify the potentially intersecting regions among the objects in the PCS. We use Lemma 3 to perform sub-object level pruning. We render each sub-object for every object in the PCS with a fully-visibility query. The sub-object could be a bounding box, a group of triangles, or even a single triangle.

The following is the pseudocode of the algorithm.

- **First pass:**
 1. Clear the depth buffer (use orthographic projection)
 2. For each object O_i , $i = 1, \dots, n$
 - Disable depth mask and set the depth function to `GL_EQUAL`.
 - For each sub-object T_k^i in O_i

Render T_k^i using an occlusion query
 - Enable the depth mask and set the depth function to `GL_LEQUAL`.
 - For each sub-object T_k^i in O_i

Render T_k^i
 3. For each object O_i , $i = 1, \dots, n$
 - For each sub-object T_k^i in O_i

Test if T_k^i is not visible with respect to the depth buffer. If it is not visible, set a tag to note it as fully visible.
- **Second pass:**

Same as First pass, except that the two “For each object” loops are run with $i = n, \dots, 1$.

4.4. Collision Detection

The overall collision detection algorithm performs object level pruning, sub-object level pruning, and triangle intersection tests among the objects in PCS.

4.4.1. Object level pruning

We perform object level pruning to compute the PCS of objects. Initially, all the objects belong to the PCS. We first perform pruning along each coordinate axis using the axis-aligned bounding boxes as the object’s representation for collision detection. The pruning is performed till the PCS does not change between successive iterations. We also use the object’s triangulated representation for further pruning the PCS. The size of the resulting set is expected to be small and we use all-pair bounding box overlap tests to compute the potentially intersecting pairs. If the size of this set is large, then we use sweep-and-prune technique³ to prune this set instead of performing all-pair tests.

4.4.2. Sub-Object level pruning

We perform multiple level pruning to identify the potentially intersecting triangles among the objects in the PCS. We group adjacent local triangles (say k triangles) to form a sub-object used in multi-level pruning and prune the potential regions considerably. This improves the performance of the overall algorithm because performing a fully-visible query for each single triangle in the PCS of objects can be expensive. At the next level, we consider the PCS of sub-objects and perform pruning using each triangle as a sub-object. The multiple-level sub-object pruning is performed across each axis.

4.4.3. Intersection Tests

We perform exact collision detection between the objects involved in the potentially colliding pairs by testing their potentially intersecting triangles.

4.5. Optimizations

In this section, we highlight a number of optimizations used to improve the performance of the algorithm.

- **AABBs and Orthographic Projections:** We use orthographic projection of axis-aligned bounding boxes. These could potentially provide an improvement factor of six in the rendering performance. Orthographic projection is used for its speed and simplicity. In addition, we use axis-aligned bounding boxes to prune the objects for intersection tests.
- **Visibility Query returning Z-fail:** A hardware visibility query providing z-fail (in particular, a query to test if z-fail is non-zero) would reduce the amount of rendering by a factor of two for AABBs under orthographic projections. This query allows us to update the depth buffer along with the occlusion query, thus providing a factor of two performance improvement. We take additional care in terms of ordering the view-axis perpendicular faces of the bounding boxes, and ensure that the results are not affected by possible self-occlusion, thus not affecting the query result by self-occlusion.
- **Avoid Stalls:** We utilize `GL_NV_occlusion_query` to avoid stalls in the graphics pipeline. We query the results of the occlusion tests at the end of each pass, improving the performance of our algorithm by a factor of four when compared to a system using `GL_HP_occlusion_test`.
- **Rendering Acceleration:** We use vertex arrays in video memory to improve the rendering performance by copying the object representation to the video memory. The rendering performance can be further improved by representing the objects in terms of triangle strips and using them along with vertex arrays.

5. Implementation and Performance

We have implemented our system on a Dell Precision Workstation consisting of a 2.4 GHz Pentium 4 CPU and a

GeForce FX Ultra 5800 GPU. We are able to perform around 400K image-space occlusion queries per second on this card. We have tested our system on four complex simulated environments.

- **Environment 1:** It consists of 100 deformable, open-ended cylinders moving in a unit cube with a density of 5–6%. Each object consists of 200 triangles. The average collision pruning time is around 4ms at an image-space resolution of 800×800 . A snapshot from this environment is shown in Fig. 5(a).
- **Environment 2:** It consists of six deformable torii, each composed of 20,000 triangles. The scene has an estimated density of 6–8%. The average collision pruning query time is around 8ms. A snapshot from this environment is shown in Fig. 5(b).
- **Environment 3:** It consists of two highly tessellated models: a bunny (35K triangles) and a dragon (250K triangles). In Fig. 5(c), we show a relative configuration of the two models and different colors are used to highlight the triangles that belong to the PCS. A zoomed-up view of the intersection region is shown in Fig. 5(d). It takes about 40 ms to perform each collision query.
- **Breaking Objects:** We used our collision detection algorithm to generate a real-time simulation of breaking objects. Fig. 6 highlights a sequence from our dynamic simulation with the bunny and the dragon colliding and breaking the dragon into multiple pieces due to impact. The total number of objects and the number of triangles in each piece are changing over the course of the simulation. Earlier collision detection algorithms are unable to handle such scenarios in real-time. Our collision detection algorithm takes about 35ms (on average) to compute all the overlapping triangles during each frame.

5.1. Performance Analysis

We have tested the performance of our algorithm and system on different benchmarks. Its overall performance is governed by some key parameters. These include:

- **Number of objects :** Our object level pruning algorithm exhibits linear time performance in our benchmarks. We have performed timing analysis by varying the number of deformable objects and Fig. 2 summarizes the results. In our simulations, we have observed that the pruning algorithm requires only a few iterations to converge (typically, it is two). Also, each iteration reduces the size of PCS. Therefore, the visibility based pruning algorithm traverses a smaller list of objects during subsequent iterations.
- **Triangle count per object :** The performance of our system depends upon the triangle count of the potentially intersecting objects. We have tested our system with simulations on 100 deformable objects consisting of varying triangle count. Fig. 3 summarizes the results. The graph indicates a linear relationship between the polygon count

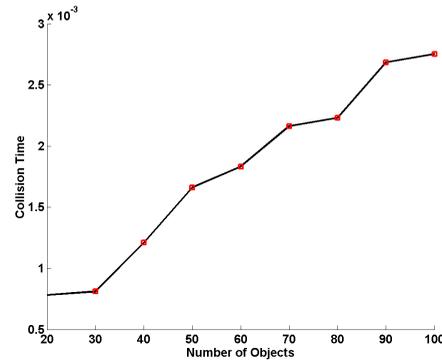


Figure 2: Number of objects v/s Average collision pruning time: This graph highlights the relationship between number of objects in the scene and the average collision pruning time (object pruning and sub-object/triangle pruning). Each object is composed of 200 triangles. The graph indicates that the collision pruning time is linear to the number of objects.

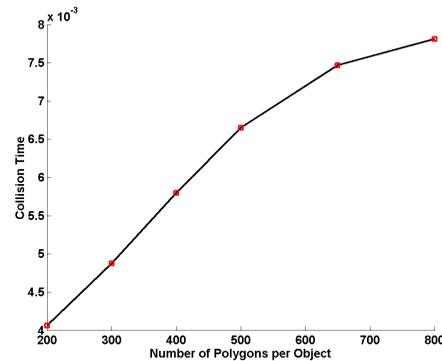


Figure 3: Polygons per object vs Average collision query time: Graph shows the linear relationship between the number of polygons per object and the average collision pruning time. This scene is composed of 100 deformable cylinders and has a density of 1–2%. The collision pruning time is averaged over 500 frames and at an image-space resolution of 800×800

and the average collision query time. Moreover, the number of polygons per object is much higher than the number of objects in the scene.

- **Accuracy and Image-Space Resolution :** The accuracy of the overall algorithm is governed by image-space resolution. Typically a higher resolution leads to higher fill-rate requirements, in terms of rendering the primitives, bounding boxes and performing occlusion queries. A lower image-space resolution can improve the query time, but can miss intersections between two objects, whose boundaries are touching tangentially or have a very small penetration. Figure 4 highlights the relationship between collision pruning time and the image-space resolution.

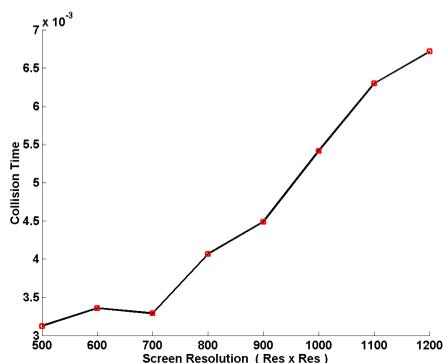


Figure 4: Image-space resolution vs Average collision query time: Graph indicating the linear relationship between screen resolution and average collision query time. The scene consists of 100 deformable cylinders and each object is composed of 200 triangles.

- **Output Size:** The performance of any collision detection algorithm varies as a function of the output size, i.e. the number of overlapping triangle pairs. In our case, the performance varies as a linear function of the size of PCS after object level pruning and sub-object level pruning as well as the number of triangle pairs that need to be tested for exact contact. In case two objects have a deep penetration, the output size can be high and therefore the size of each PCS can be high as well.
- **Rasterization optimizations:** The performance of the system is accelerated using the rasterization optimizations discussed in Section 4.5. We have used AABBs with orthographic projections for our pruning algorithms. We have used immediate mode for rendering the models and breakable objects, and used GL_NV_occlusion_query to maximize the performance.

5.2. Pruning Efficiency

Our overall approach for collision detection is based on pruning techniques. Its overall performance depends on the input complexity, the relative configuration of different objects in the scene as well as pruning efficiency of the object-level and sub-object level algorithms. Most pruning algorithms based on bounding volume hierarchies can take a long time for parallel close proximity scenarios⁶ (eg. two concentric spheres with nearly same radius). Our algorithm performs pruning at the triangle level and works well in these configurations. It should be noted that the pruning efficiency largely depends upon the choice of view direction for orthographic projection. Certain view directions may not provide sufficient pruning as a larger number of objects may be partially visible from this view. One possible solution to avoid such configurations is randomly select the directions for orthographic projection.

5.3. Comparison with Other Approaches

Collision detection is well-studied in the literature and a number of algorithms and public-domain systems are available. However, none of the earlier algorithms provide the same capabilities or features as our algorithm based on PCS computation. As a result, we have not performed any direct timing comparisons with the earlier systems. We just compare some of the features of our approach with the earlier algorithms.

Object-Space Algorithms: Algorithms based on sweep-and-prune are known for N-body collision detection³. They have been used in I-COLLIDE, V-COLLIDE, SWIFT, SOLID and other systems. However, these algorithms were designed for rigid bodies and compute a tight fitting AABB for each object using incremental methods, followed by sorting their projections of AABBs along each axis. It is not clear whether they can perform real-time collision detection on large environments composed of deformable models. On the other hand, our algorithm performs two passes on the object list to perform the PCS. We expect that our PCS based algorithm to be more conservative as compared to sweep-and-prune. Furthermore, the accuracy of our approach is governed by the image-space resolution.

A number of hierarchical algorithms have been proposed to check two highly tessellated models for overlap and some optimized systems (e.g. RAPID, QuickCD) are also available. They involve considerable preprocessing and memory overhead in generating the hierarchy and will not work well on deformable models or objects with changing topology.

Image-Space Algorithms : These include algorithms based on stencil buffer techniques as well as distance field computations. Some systems such as PIVOT are able to perform other proximity queries including distance and local penetration computation, whereas our PCS based algorithm is limited to only checking for interference. However, our algorithm only needs to readback the output of a visibility query and not the entire depth-buffer or stencil buffer. This significantly improves its performance, especially when we use higher image-space precision. Unlike earlier algorithms, our PCS-based algorithm is applicable to all triangulated 3D models (and not just closed objects), and can handle arbitrary number of objects in the environment.

5.4. Conclusions and Future Work

We have presented a novel algorithm for collision detection between multiple deformable objects in a large environment using graphics hardware. Our algorithm is applicable to all triangulated models, makes no assumption about object motion and can compute all contacts up to image-space resolution. It uses a novel, linear-time PCS computation applied iteratively to objects and sub-objects. The PCS is computed using image-space visibility queries widely available

on commodity graphics hardware. It only needs to readback the results of a query, not the frame-buffer or depth buffer.

Limitations: Our current approach has some limitations. First, it only checks for overlapping objects, and does not provide distance or penetration information. Secondly, its accuracy is governed by the image-space resolution. Finally, it currently cannot handle self-collision within each object.

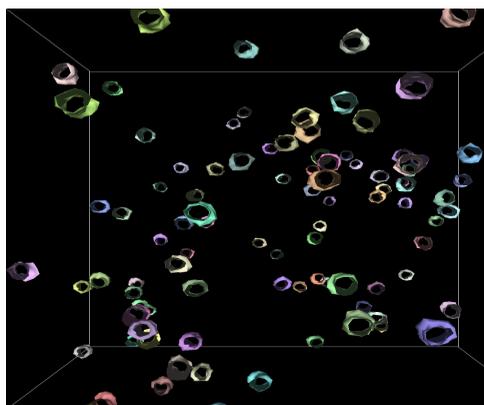
There are many avenues for future work. In addition to overcoming these limitations, approaches we would like to use our PCS based collision detection for more applications and to evaluate its impact on the accuracy of the overall simulation. We would like to further investigate use of the new programmability features of graphics hardware to design improved geometric query algorithms.

Acknowledgments

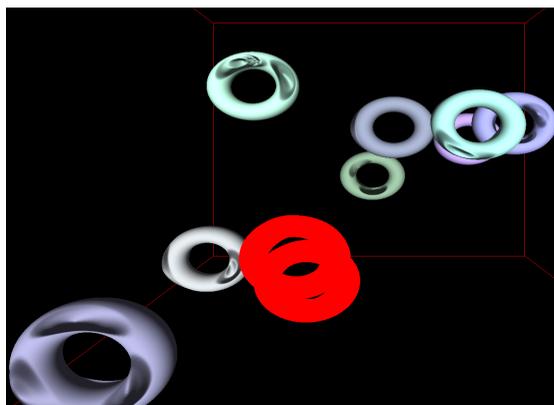
This research is supported in part by ARO Contract DAAD 19-99-1-0162, NSF awards ACI-9876914, IIS-982167, ACI-0118743, ONR Contracts N00014-01-1-0067 and N00014-01-1-0496, and Intel Corporation. The Stanford dragon and bunny are courtesy of Stanford Computer Graphics Laboratory. We would like to thank NVIDIA especially David Kirk, Steven Molnar, Paul Keller and Stephen Ehmann for their support. We would also like to thank Avneesh Sud for his support.

References

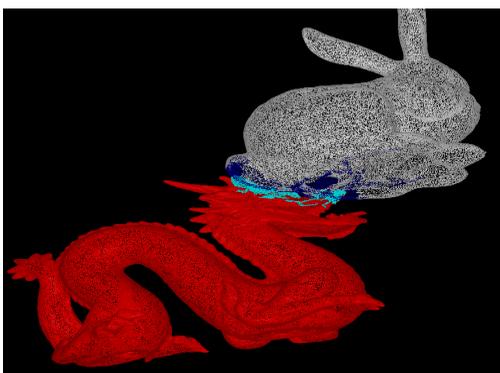
1. G. Baciú and S.K. Wong. Image-based techniques in a hybrid collision detector. *IEEE Trans. on Visualization and Computer Graphics*, 2002.
2. G. Baciú, S.K. Wong, and H. Sun. Recode: An image-based collision detection algorithm. *Proc. of Pacific Graphics*, pages 497–512, 1998.
3. J. Cohen, M. Lin, 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.
4. D. Cohen-Or, Y. Chrysanthou, and C. Silva. A survey of visibility for walkthrough applications. *SIGGRAPH Course Notes # 30*, 2001.
5. J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs. Near real-time csg rendering using tree normalization and geometric pruning. *IEEE Computer Graphics and Applications*, 9(3):20–28, 1989.
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. K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. *Proceedings of ACM SIGGRAPH*, pages 277–286, 1999.
8. 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*, 2001.
9. K. Hoff, A. Zaferakis, M. Lin, and D. Manocha. Fast 3d geometric proximity queries between rigid and deformable models using graphics hardware acceleration. Technical Report TR02-004, Department of Computer Science, University of North Carolina, 2002.
10. P. M. Hubbard. Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*, October 1993.
11. A. Kaul and J. Rossignac. Solid-interpolating deformations: construction and animation of pips. *Computer and Graphics*, 16:107–116, 1992.
12. Y. Kim, M. Lin, and D. Manocha. Deep: An incremental algorithm for penetration depth computation between convex polytopes. *Proc. of IEEE Conference on Robotics and Automation*, 2002.
13. Y. Kim, M. Otaduy, M. Lin, and D. Manocha. Fast penetration depth computation using rasterization hardware and hierarchical refinement. *Proc. of Workshop on Algorithmic Foundations of Robotics*, 2002.
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. M. Lin and S. Gottschalk. Collision detection between geometric models: A survey. *Proc. of IMA Conference on Mathematics of Surfaces*, 1998.
16. M. Lin and D. Manocha. Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*, 2003. to appear.
17. J. C. Lombardo, M.-P. Cani, and F. Neyret. Real-time collision detection for virtual surgery. *Proc. of Computer Animation*, 1999.
18. 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.
19. S. Redon, A. Kheddar, and S. Coquillart. Fast continuous collision detection between rigid bodies. *Proc. of Eurographics (Computer Graphics Forum)*, 2002.
20. 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.
21. J. Rossignac and J. Wu. Correct shading of regularized csg solids using a depth-interval buffer. In *Eurographics Workshop on Graphics Hardware*, 1990.
22. M. Shinya and M. C. Fongue. Interference detection through rasterization. *The Journal of Visualization and Computer Animation*, 2(4):131–134, 1991.
23. 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.



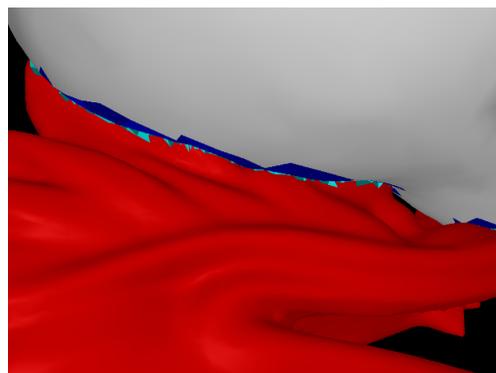
(a) Environment 1: This scene consists of 100 dynamically deforming open cylinders moving randomly in a room. Each cylinder is composed of 200 triangles.



(b) Environment 2: This scene consists of 10 dynamically deforming torii moving randomly in a room. Each torus is composed of 20000 triangles



(c) Environment 3: Wired frame of dragon and bunny rendered in the following colors - {cyan,blue} highlight triangles in the PCS, {red, white} illustrate portions not in the PCS. The dragon consists of 250K triangles and the bunny consists of 35K faces



(d) Environment 3: Zoomed view highlighting the exact intersections between the triangles in the PCS. The configuration of the two objects is the same as Fig. 5(c). The cyan and blue regions are the overlapping triangles of the dragon and bunny respectively.

Figure 5: Snapshots of simulations on three complex environments. Our collision detection algorithm takes 4,8, 40ms respectively on each benchmark to perform collision queries on a GeForce FX 5800 Ultra with an image resolution of 800×800 .

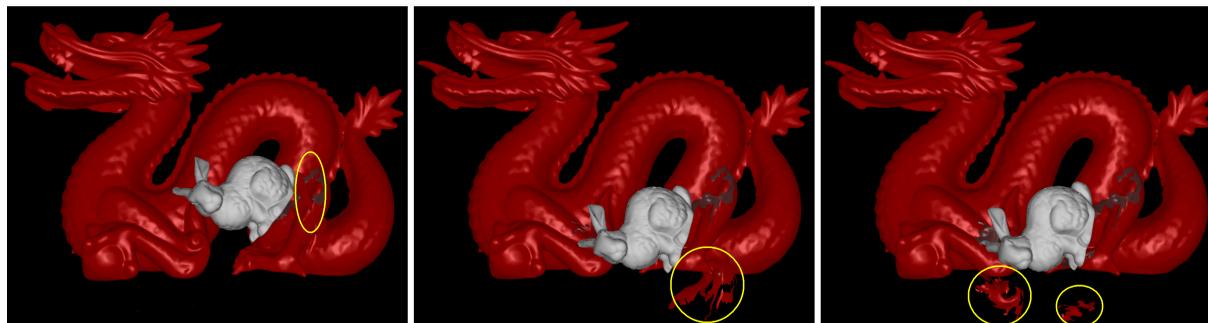


Figure 6: Environment with breakable objects: As the bunny (with 35K triangles), falls through the dragon (with 250K), the number of objects in the scene (shown with a yellow outline) and the triangle count within each object change. Our algorithm computes all the overlapping triangles during each frame. The average collision query time is 35 milliseconds per frame.