Fast and Reliable Collision Culling using Graphics Hardware

Naga K. Govindaraju Ming C. Lin Dinesh Manocha

University of North Carolina at Chapel Hill
{naga, lin, dm}@cs.unc.edu
http://gamma.cs.unc.edu/RCULLIDE

Figure 1: Tree with falling leaves: In this scene, leaves fall from the tree and undergo non-rigid motion. They collide with other leaves and branches. The environment consists of more than 40K triangles and 150 leaves. Our algorithm, FAR, can compute all the collisions in about 35 msec per time step.

ABSTRACT

We present a reliable culling algorithm that enables fast and accurate collision detection between triangulated models in a complex environment. Our algorithm performs fast visibility queries on the GPUs for eliminating a subset of primitives that are not in close proximity. To overcome the accuracy problems caused by the limited viewport resolution, we compute the Minkowski sum of each primitive with a sphere and perform reliable 2.5D overlap tests between the primitives. We are able to achieve more effective collision culling as compared to prior object-space culling algorithms. We integrate our culling algorithm with CULLIDE [8] and use it to perform reliable GPU-based collision queries at interactive rates on all types of models, including non-manifold geometry, deformable models, and breaking objects.

Categories and Subject Descriptors

I.3.1 [**Hardware Architecture**]: Graphics processors; I.3.7 [**Three-Dimensional Graphics and Realism**]: Visible surface algorithms, animation, virtual reality; I.3.5 [**Computational Geometry and Object Modeling**]: Geometric algorithms

General Terms

Algorithms, performance, reliability

VRST'04, November 10-12, 2004, Hong Kong.

Keywords

Collision detection, graphics hardware, deformable models, Minkowski sums

1. INTRODUCTION

Graphics processing units (GPUs) have been increasingly used for collision and proximity computations. GPUs are well-optimized for 3-D vector and matrix operations, and complex computations on the frame-buffer pixel or image data. Different algorithms have exploited these capabilities to compute interference or overlapping regions or to cull away portions of the models that are not in close proximity. Most of these algorithms involve no preprocessing and therefore apply to both rigid and deformable models. In many cases, GPU-based algorithms can offer better runtime performance as compared to object-space algorithms.

GPU-based collision detection algorithms, however, often suffer from limited precision. This is due to the viewport resolution, sampling errors, and depth precision errors. For example, current GPUs provide a viewport resolution of $2K \times 2K$ pixels, which is equivalent to about 11 bits of fixed-precision arithmetic. The low precision and sampling errors can result in missed collisions between two objects. In contrast, object-space collision detection algorithms are able to perform more accurate interference computations using IEEE 32 or 64-bit floating arithmetic on the CPUs.

Main Results: We present a simple and efficient algorithm *FAR* for fast and reliable collision culling between triangulated models in a large environment using GPUs. We perform visibility queries to eliminate a subset of primitives that are not in close proximity, thereby reducing the number of pairwise tests that are performed for exact proximity computation. We show that the *Minkowski sum* of each primitive with a sphere provides a conservative bound for performing reliable 2.5D overlap tests using GPUs. The radius of

the sphere is a function of viewport resolution and depth buffer precision. For each geometric primitive (a collection of triangles), our algorithm computes a tight bounding offset representation. The bounding offset representation is a union of object-oriented bounding boxes (UOBB) where each OBB encloses a single triangle. Our algorithm performs visibility queries using these UOBBs on GPUs to reject primitives that are not in close proximity. Overall, our algorithm guarantees that no collisions will be missed due to limited framebuffer precision or quantization errors during rasterization.

The key advantages of our approach are:

- More reliable computations over prior GPU-based methods;
- More effective culling over existing CPU-based algorithms;
- Broad applicability to non-manifold geometry, deformable models, and breaking objects;
- Interactive performance with no preprocessing and low memory overhead.

We have combined our culling algorithm with CULLIDE [8] to perform collision detection in complex environments. We utilize the GPU for fast and reliable pruning of primitive pairs and perform exact interference tests on the CPU. We have implemented this collision culling algorithm on a Pentium IV PC with NVIDIA GeForce FX 5950 card. We are able to perform interactive collision detection between complex objects composed of tens of thousands of triangles that undergo rigid and non-rigid motion, including fracturing and deformation.

Organization: The rest of the paper is organized in the following manner. In Sec. 2, we give a brief overview of related work in collision detection. We present sufficient conditions for eliminating image-based sampling errors in Sec. 3. In Sec. 4, we present details for fast computation of bounding offset representations and our conservative culling algorithm. We describe its implementation in Sec. 5 and highlight its performance on different environments. Finally, we analyze our algorithm and describe some of its limitations in Sec. 6.

2. RELATED WORK

The problem of collision detection has been well studied for more than three decades. See recent surveys in [19] and [14] for an overview. Prior algorithms for collision detection between triangulated models can be classified into three broad categories: object-space culling, image-space intersection computation, and hybrid approaches.

Object-space culling: Most of the commonly used techniques to accelerate collision detection between two objects utilize spatial data structures, including spatial partitioning and bounding volume hierarchies. Some of the commonly used bounding volume hierarchies include sphere-trees [13, 26], AABB-trees [5, 25], OBB-trees [7, 4], k-DOP-trees [11, 17], etc. These representations are used to cull away portions of each object that are not in close proximity. Typically, these representations are built in a pre-processing stage to accelerate runtime queries. In practice, they work well for rigid objects. However, the overhead of recomputing the hierarchy on the fly for deformable models can be quite significant [2, 12].

Image-space interference computation: Several algorithms have used graphics hardware for interference and collision computations [2, 3, 9, 10, 12, 18, 23, 28, 29, 31]. These algorithms require no preprocessing; they work well on commodity GPUs. However, they have some limitations. First, they can detect a collision up to viewport resolution. The accuracy of collision detection also varies based on the relative distance between the objects, i.e. collision queries are less accurate if the objects are separated by distances greater than their average size. Second, most of these algorithms need to read back the color or depth buffer contents for further processing and readbacks can be slow on current graphics systems [18, 8]. There exists software implementations for reliable interference detection using fat edges and readback multiple depth layers [28], but they work well only on scenes with two objects and few contacts. Also, [28] does not address the issue of aliasing in depth buffer. This limitation is addressed in [27] and used for rendering image-precision silhouette edges. [27] fatten the back-facing polygons for rendering silhouette edges. The front-facing polygons are not fattened as the technique only *renders* silhouette edges. As some polygons are not fattened, the technique described in [27] may miss interferences due to limited image precision.

Hybrid methods: Hybrid algorithms combine some of the benefits of the object-space and image-space approaches. Kim et al. [16] compute the closest distance from a point to the union of convex polytopes using the GPU, refining the answer on the CPU. Govindaraju et al. [8] use occlusion queries on the GPU to cull away objects that are not colliding with others. Heidelberger et al. [10] 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. In all these cases, GPU-based techniques are used to accelerate the overall computation. However, viewport resolution governs the accuracy of these algorithms.

3. RELIABLE CULLING USING GPUS

In this section, we present our culling algorithm that performs visibility queries on GPUs and culls away primitives that are not in close proximity. We also analyze the sampling problems caused by limited viewport resolution and present a sufficient condition to perform conservative and reliable culling.

3.1 Overlap Tests

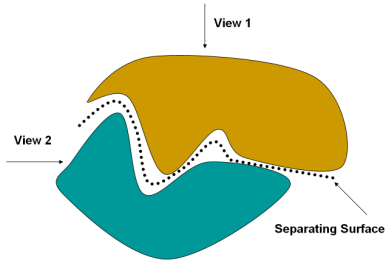


Figure 2: ***In this figure, the objects are not colliding. Using view*** 1***, we determine a separating surface with unit depth complexity along the view and conclude from the existence of such a surface that the objects are not colliding. This is a sufficient but not a necessary condition. Observe that in view*** 2***, there does not exist a separating surface with unit depth complexity but the objects are not interfering.***

Interference computation algorithms employ GPUs to perform either 2D overlap tests using color and stencil buffers or 2.5D overlap tests with additional depth information. The 2.5D overlap tests are less conservative and can be performed using occlusion queries on current graphics processors [8].

Visibility-based overlap tests: Govindaraju et al. [8] perform visibility computations to check whether two primitives, P_1 and

P_2, overlap. The approach chooses a view direction and checks whether P_1 is fully visible with respect to P_2 along that direction. If P_1 is fully visible then there exists a separating surface between P_1 and P_2. We call this the *visibility-based-overlap* (**VO**) query, which provides a sufficient condition that the two primitives do not overlap and is illustrated in Fig. 2. The **VO** query is performed efficiently on GPUs. Using three or less mutually orthogonal orthographic views, many complex objects that are in close proximity (as shown in Fig. 2) can be pruned. However, due to limited viewport and frame buffer resolution, **VO** queries can miss collisions and this problem is typical of any GPU-based interference detection algorithm. Our goal is to develop *reliable* VO queries on the GPUs for efficiently pruning complex configurations as shown in Fig 2, without missing any collisions.

CULLIDE: We now briefly describe CULLIDE [8] which performs **VO** queries between multiple objects and computes a potentially colliding set (PCS) of objects. Given n objects that are potentially colliding $P_1, ..., P_n$, Govindaraju et al. [8] describe a linear time two-pass rendering algorithm to test if an object P_i is fully visible against the remaining objects, along a view direction. The algorithm uses occlusion queries to test if an object is fully visible or not. 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 depth writes. The object P is rendered using an occlusion query. If the pixel pass count returned by occlusion query is zero, then the object P is fully visible and therefore, does not collide with S. Using this formulation, Govindaraju et al. [8] prune objects P_i that do not overlap with other objects in the environment. The algorithm begins with empty frame buffer and proceeds in two passes as follows:

In the first pass, CULLIDE rasterizes the primitives in the order $P_1, ..., P_n$ testing if they are fully visible. In this pass, if a primitive P_i is fully visible, then it does not intersect any of the objects $P_1, ..., P_{i-1}$. In the second pass, it performs the same operations but renders the primitives in the order $P_n, .., P_1$. In this pass, if a primitive P_i is fully visible, then it does not intersect 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, the primitive does not interfere with the remaining primitives and is removed from the PCS. The view directions are chosen along the world-space axes and collision culling is performed using orthographic projections.

3.2 Sampling Errors

We define the notation used in the rest of paper and the issues in performing interference detection on GPUs.

Orthographic projection: Let **A** be an axis, where $A \in \{X, Y, Z\}$ and, A_{min} and A_{max} define the lower and upper bounds on P_1 and P_2 along **A**'s direction in 3D. Let RES(**A**) define the resolution along an axis. The viewport resolution of a GPU is $\text{RES}(X) \times \text{RES}(Y)$ (e.g. $2^{11} \times 2^{11}$) and the depth buffer precision is RES(Z) (e.g. 2^{24}).

Let O be an orthographic projection with bounds $(X_{min}, X_{max}, Y_{min}, Y_{max}, Z_{min}, Z_{max})$ on the 3D primitives. The dimension of the grid along an axis in 3D is given by d_A where $d_A = \frac{A_{max} - A_{min}}{\text{RES}(A)}$. Rasterization of a primitive under orthographic projection performs linear interpolation of the vertex coordinates of each primitive and maps each point on a primitive to the 3D grid. This mapping is based on sampling of a primitive at fixed locations in the grid. When we rasterize the primitives to perform VO queries, many errors arise due to sampling. There are three types of errors:

1. **Projective and perspective aliasing errors:** These errors can result in some of the primitives not getting rasterized. This error may result in an incorrect answer to the VO query.
2. **Image sampling errors:** We can miss interferences between triangles due to sampling at the fixed locations. In this case, each triangle is sampled but the intersection set of the triangles is not sampled (see Fig. 3).
3. **Depth-buffer precision errors:** If the distance between two primitives is less than RES(**Z**), VO query may not be able to accurately compute whether one is fully visible with respect to the other.

3.3 Reliable VO Queries

We can overcome the errors described in Section 3.2 by generating a "fattened" representation T^B for each triangle T. If a triangle T interferes with a set of primitives S within a pixel, we may miss interferences because the triangle is inaccurately classified fully visible due to the following possibilities:

- **Error 1:** a fragment is not generated when rasterizing T or S.
- **Error 2:** a fragment is generated but does not sample the interfering points within the pixel.
- **Error 3:** a fragment is generated and samples the interfering points within a pixel but the precision of frame or depth buffer is not sufficient.

These errors correspond to the three types of errors discussed in Section 3.2. Our approach solves these problems using "fattened" representations of triangles that

- Generates at least two fragments for each pixel touched [1] by a triangle.
- For each pixel touched by a triangle, the depth of the corresponding two fragments bound the depth of all points of the triangle that project inside the pixel.

Using a closed fattened representation T^B for each triangle T in CULLIDE provides a sufficient condition for eliminating the sampling and precision errors. Suppose two primitives T_1 and T_2 intersect at some point within a pixel X that may or may not be sampled. Then T_1^B is not fully visible with respect to T_2^B as rasterization of T_1^B generates two fragments corresponding to X and at least one of the two fragments fails the depth test. Similarly, T_2^B is not fully visible with respect to T_1^B. Therefore, neither T_1 nor T_2 is pruned from the PCS. In the rest of the section, we formally prove that for a given orthographic view, the Minkowski sum of a bounding cube B centered at the origin with T provides a conservative fattened representation T^B and eliminates sampling or precision errors irrespective of the sampling strategy. The size of the bounding cube B is a function of the world space pixel dimensions and in practice, is very small. Therefore, P^B is a very tight fit to the original geometric primitive P.

Our algorithm does not make any assumptions about sampling the primitives within a pixel. We compute an axis-aligned bounding box B with dimension p where $p = max(2{*}d_X, 2{*}d_Y, 2{*}d_Z)$ centered at the origin. In practice, this bound may be conservative. If a GPU uses some uniform supersampling algorithm during rasterization, p can be further reduced. For example, if the GPU samples each pixel in the center, then p can be reduced by half.

Let B be an axis-aligned cube centered at the origin with dimension p. Given two primitives, P_1 and P_2, let Q be a point on their

[1] A pixel is touched by a triangle if some point of the triangle projects inside the pixel

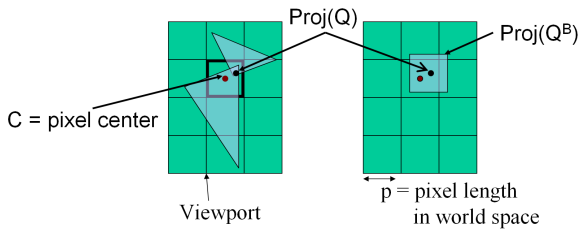


Figure 3: ***Sampling errors:*** *Q is a point on the line of intersection between two triangles in $3D$. The left figure highlights its orthographic projection in the screen space. The intersection of two triangles does not contain the center of the pixel (C) and therefore, we can miss a collision between the triangles. Q^B is the Minkowski sum of Q and an axis-aligned bounding box (B) centered at the origin with dimension p. Q^B translates B to the point Q. During rasterization, the projection of Q^B samples the center of pixel and generates at least two fragments that bound the depth of Q.*

line of intersection. We use the concept of *Minkowski sum* of a primitive P with B, ($P^B = P \oplus B$), which can be defined as: $\{\boldsymbol{p} + \boldsymbol{b} \mid \boldsymbol{p} \in P,\ \boldsymbol{b} \in B\}$. Next we show that $P \oplus B$ can be used to perform reliable VO queries. We first state two lemmas and use them to derive the main result as a theorem.

Lemma 1**:** Under orthographic transformation O, the rasterization of Minkowski sum $Q^B = (\mathrm{Q} \oplus \mathrm{B})$, where Q is a point in 3D space that projects inside a pixel X, samples X with at least two fragments bounding the depth value of Q.

Proof: Q^B is a box centered at Q and its projection covers the center of X as shown in Figure 3. As a result, Q is sampled by the rasterization hardware and two fragments that bound the depth of Q are generated. □

Lemma 2**:** Given a primitive P_1 and its Minkowski sum P_1^B = P_1 $\oplus$ B. Let X be a pixel partly or fully covered by the orthographic projection of P_1. Let us define MIN-DEPTH(P_1, X) and MAX-DEPTH(P_1, X) as the minimum and maximum depth value of the points of P_1 that project inside X, respectively. The rasterization of P_1^B generates at least two fragments whose depth values bound both MIN-DEPTH(P_1,X) and MAX-DEPTH(P_1,X) for each pixel X.

Proof: The proof follows from Lemma 1. This lemma indicates that at least two fragments are generated after rasterizing P_1^B such that their depth values provide lower and upper bounds to the depth of all points of P_1 that project inside X. This result holds irrespective of projective or perspective errors. □

Theorem 1**:** Given the Minkowski sum of two primitives with B, P_1^B and P_2^B. If P_1 and P_2 overlap, then a rasterization of their Minkowski sums under orthographic projection overlaps in the viewport.

Proof: Let P_1 and P_2 intersect at a point Q inside a pixel X. Based on Lemma 2, we can generate at least two fragments rasterizing P_1^B and P_2^B. These fragments bound all the 3D points of P_1 and P_2 that project inside X. Showing that the pairs (MIN-DEPTH(P_1, X), MAX-DEPTH(P_1,X)) and (MIN-DEPTH(P_2, X), MAX-DEPTH(P_2,X)) overlap is sufficient. This observation follows trivially as MIN-DEPTH(P_1, X) $\leq$ Depth(Q), MIN-DEPTH(P_2, X) $\leq$ Depth(Q) and MAX(P_1, X) $\geq$ Depth(Q), MAX(P_2, X) $\geq$ Depth(Q). □

3.4 Collision Culling

A corollary of Theorem 1 is that if P_1^B and P_2^B do not overlap, then P_1 and P_2 do not overlap. In practice, this test can be conservative, but it won't miss any collisions because of viewport or depth resolution. However, the Minkowski sums, P_1^B and P_2^B, are only useful when the primitives are projected along the Z-axis. To generate a view-independent bound, we compute the Minkowski sum of a primitive P with a sphere S of radius $\sqrt{3}p/2$ centered at the origin. The Minkowski sum of a primitive with a sphere is the same as the *offset* of that primitive.

4. INTERACTIVE COLLISION DETECTION

In this section, we present our reliable collision culling algorithm. We first describe a bounding offset representation for each primitive and integrate it with CULLIDE for interactive collision detection.

4.1 Bounding Offset Representations

In order to overcome sampling errors, we use a bounding offset for each primitive as implied by Theorem I. Our collision culling algorithm renders bounding offset representations to cull away primitives that are not in close proximity. Several choices are possible for computing bounding offsets and they trade-off tightness of fit with the rendering cost.

- **Exact Offsets:** The boundary of an exact offset of a triangle consists of piecewise linear and spherical surfaces. In particular, the Minkowski sum of a triangle T and a sphere S centered at the origin is the union of three edge aligned cylinders of thickness $Radius(S)$, three spheres S centered at the vertices and two triangles. The two triangles are shifted along the normal of the original triangle by the $Radius(S)$. The exact offset is the tightest fitting volume that can be rendered using graphics processors ensuring reliable interference computation. Using fragment programs, it is possible to render the exact offset representation for each triangle but can be relatively expensive.

- **Bounded Exact Offsets:** Another possibility is to tightly bound the exact offsets using three edge axis aligned bounding boxes each bounding a cylinder and a sphere. This representation is a tighter fit and replaces each triangle with three bounding boxes and two triangles, thus generating 30 vertices. In our implementation, we observed that the tight fit provides better culling but is vertex-transform limited.

- **Union of Object-oriented Bounding Boxes (UOBBs):** A tight-fitting conservative bounding representation for a primitive is a union of object-oriented bounding boxes (OBBs) where each OBB encloses a single triangle of the primitive. Given a triangle T, we compute the tightest fitting rectangle R that encloses T; one of its axes is aligned with the longest edge of the triangle. We compute the OBB for a triangle as the Minkowski sum of B and R, where B is a locally axis-aligned bounding cube of width $Diameter(S)$. The width of the OBB, along a dimension orthogonal to the plane containing R, is set equal to $\sqrt{3}p$. The bounding offset of a triangulated object is the union of OBBs of each triangle (see Fig. 4). We render this bounding offset by rendering each OBB separately and perform VO queries. In practice, this is a very tight bounding volume for an object, as compared to using a single sphere, AABB (axis-aligned bounding box) or an OBB that encloses the entire object.

Our algorithm uses UOBBs as bounding offset representations as shown in Fig. 4 for reliable collision culling. The computation of an OBB for a triangle requires 24 multiplications, 41 additions, 6 divisions and 2 comparison operations. Further optimizations such as shared edges between adjacent triangles can be used to reduce

the number of operations. Alternatively, other tight bounding volumes such as triangular prisms could be used. However, they can be expensive to compute as compared to OBBs and are more conservative for long, skinny triangles. In particular, computation of a triangular prism involves 48 multiplications, 51 additions, 9 division operations.

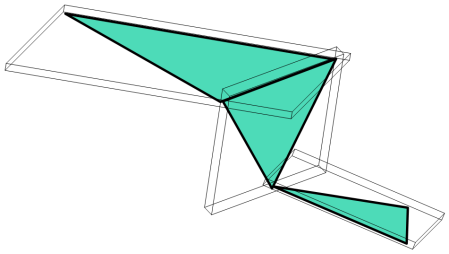


Figure 4: ***This image shows an object with three triangles and its bounding offset representation (UOBB) in wireframe. The UOBB is represented as the union of OBBs of each triangle. In practice, this bounding offset is a tight fitting bounding volume and used for culling.***

4.2 Algorithm

We have integrated our culling algorithm with CULLIDE [8] to perform reliable collision detection between objects in a complex environment. As described in section 3, CULLIDE uses **VO** queries to perform collision culling on GPUs. We extend CULLIDE to perform reliable collision culling on GPUs by using reliable **VO** queries described above. For each primitive in the PCS, we compute its bounding offset (i.e. union of OBBs) representation and use the bounding offset representations in CULLIDE to test if the primitives belong to PCS or not.

Our collision detection algorithm, FAR, proceeds in three steps. First we compute the PCS at the object level. We use sweep-and-prune [6] 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 [21].

4.2.1 Optimizations

We have implemented several optimizations in our algorithm. Bounding offset representations generate nearly twice the amount of fill in comparison to the original geometric primitives. As the offset representation for each triangle is closed, we can reduce the fill requirements for our algorithm by a factor of two by using face-culling. In our optimized algorithm, we cull front faces while rendering the offset representations with occlusion queries and we cull back faces while rendering the offset representations to the frame buffer. These operations can be performed efficiently using *back-face culling* on graphics hardware. We also reduce the number of occlusion queries in the second pass of our algorithm by testing only those primitives whose offset representations are fully visible in first pass.

The pseudo-code for our optimized algorithm is given below:

- **First pass:**

  1. *Clear the depth buffer (use orthographic projection)*
  2. *For each object P_i, $i = 1, .., n$*
     - *Disable the depth mask and set the depth function to GL_GEQUAL.*
     - *Enable back-face culling to cull front faces.*
     - *For each sub-object T_k^i in P_i*

       Render offset representation of T_k^i using an occlusion query
     - *Enable the depth mask and set the depth function to GL_LEQUAL.*
     - *Enable back-face culling to cull back faces.*
     - *For each sub-object T_k^i in P_i*

       Render offset representation of T_k^i
  3. *For each object P_i, $i = 1, .., n$*
     - *For each sub-object T_k^i in P_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 $i = n, .., 1$ and we perform occlusion queries only if the primitive is fully visible in first pass.

4.3 Localized Distance Culling

Many algorithms aim to compute all pairs of objects whose separation distance is less than a constant distance D. In this case, we modify GPU-based culling algorithms to cull away primitives whose separation distance is more than D. Given a distance d, our goal is to prune triangles further than d. We can easily modify the culling algorithm presented above to perform this query. We compute the offset of each primitive by using a sphere of radius $\frac{D}{2} + \frac{\sqrt{3}p}{2}$, rasterize these offsets and prune away a subset of primitives whose separation distance is more than D.

4.4 Accuracy

We perform reliable VO queries by rendering the bounding offsets of primitives. Theorem I guarantees that we won't miss any collisions due to the viewport resolution or sampling errors. We perform orthographic projections as opposed to perspective projections. Further, the rasterization of a primitive involves linear interpolation along all the dimensions. As a result, the rasterization of the bounding offsets guarantees that we won't miss any collision due to depth-buffer precision. If the distance between two primitives is less than the depth buffer precision, $\frac{1}{\text{RES}(Z)}$, then VO query on their offsets will always return them as overlapping. Consequently, the accuracy of the culling algorithm is governed by the accuracy of the hardware used for performing vertex transformations and mapping to the 3D grid. For example, many of the current GPUs use IEEE 32-bit floating point hardware to perform these computations.

5. IMPLEMENTATION

We have implemented FAR on a Dell precision workstation with a 2.8 GHz Xeon processor, 1 GB of main memory, and a NVIDIA GeForce FX 5950 Ultra graphics card. We use a viewport resolution of 1400×1400 to perform all the computations. We improve the rendering throughput by using vertex arrays and use GL_NV_occlusion_query to perform the visibility queries.

5.1 Benchmarks

We have tested our algorithm on three complex scenes and have compared its culling performance and accuracy with some prior

Figure 5: ***Breaking object scene: In this simulation, the bunny model falls on the dragon which eventually breaks into hundreds of pieces. FAR computes collisions among the new pieces of small objects introduced into the environment and takes*** 30 ***to*** 60 ***msec per frame.***

approaches.

Dynamically generated breaking objects: The scene consists of a dragon model initially with 112K polygons, and a bunny with 35K polygons, as shown in Fig. 5. In this simulation, the bunny falls on the dragon, causing the dragon to break into many pieces over the course of the simulation. Each piece is treated as a separate object for collision detection. Eventually hundreds of new objects are introduced into the environment. We perform collision culling to compute which object pairs are in close proximity. It takes about 35 msec towards the beginning of the simulation, and about 50 msec at the end when the number of objects in the scene is much higher.

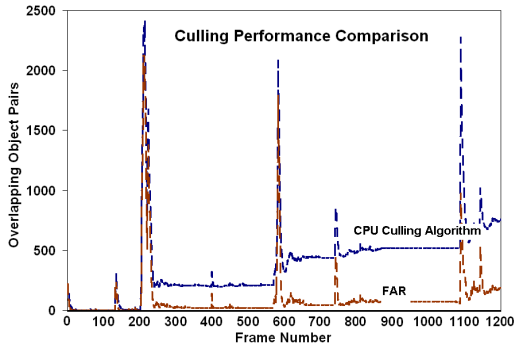


Figure 6: ***Relative culling performance on breaking objects scene: This graph highlights the improved culling performance of our algorithm as compared to a CPU-based culling algorithm (I-COLLIDE) that uses AABBs (axis-aligned bounding boxes) to cull away non-overlapping pairs. FAR reports*** 6.9 ***times fewer pairs over the entire simulation.***

We compared the culling performance of our GPU-based reliable culling algorithm with an implementation of the sweep-and-prune algorithm available in I-COLLIDE [6]. The sweep-and-prune algorithm computes an axis-aligned bounding box (AABB) for each object in the scene and checks all the AABBs for pairwise overlaps. Fig. 6 shows the comparison between the culling efficiency of AABB-based algorithm vs. FAR. Overall, FAR returns 6.9 times fewer overlapping pairs. This reduction occurs mainly because FAR uses much tighter bounding volumes, i.e. the union of OBBs for an object as compared to an AABB and is able to cull away more primitive pairs.

Interference computation between complex models: In this scene, we compute all the overlapping triangles pairs between a 68K triangles bunny that is moving with respect to another bunny, also with 68K triangles. The bunnies are deeply penetrating and the intersection boundary consists of $2,000 - 4,000$ triangle pairs. In this case, the accuracy of FAR equals that of a CPU-based algorithm using 32-bit IEEE floating point arithmetic. In contrast, CULLIDE misses a number of overlapping pairs while using a viewport resolution of $1,400 \times 1,400$. The intersection sets computed by FAR and CULLIDE are shown in Fig. 7.

Multiple objects with non-rigid motion: This scene consists of a non-rigid simulation in which leaves fall from the tree, as shown in Fig. 1. We compute collisions among the leaves of the tree and among the leaves and branches of the tree. Each leaf is represented using 156 triangles and the complete environment consists of 40K triangles. The average collision detection time is 35 msec per time step.

5.2 Comparison with Other Approaches

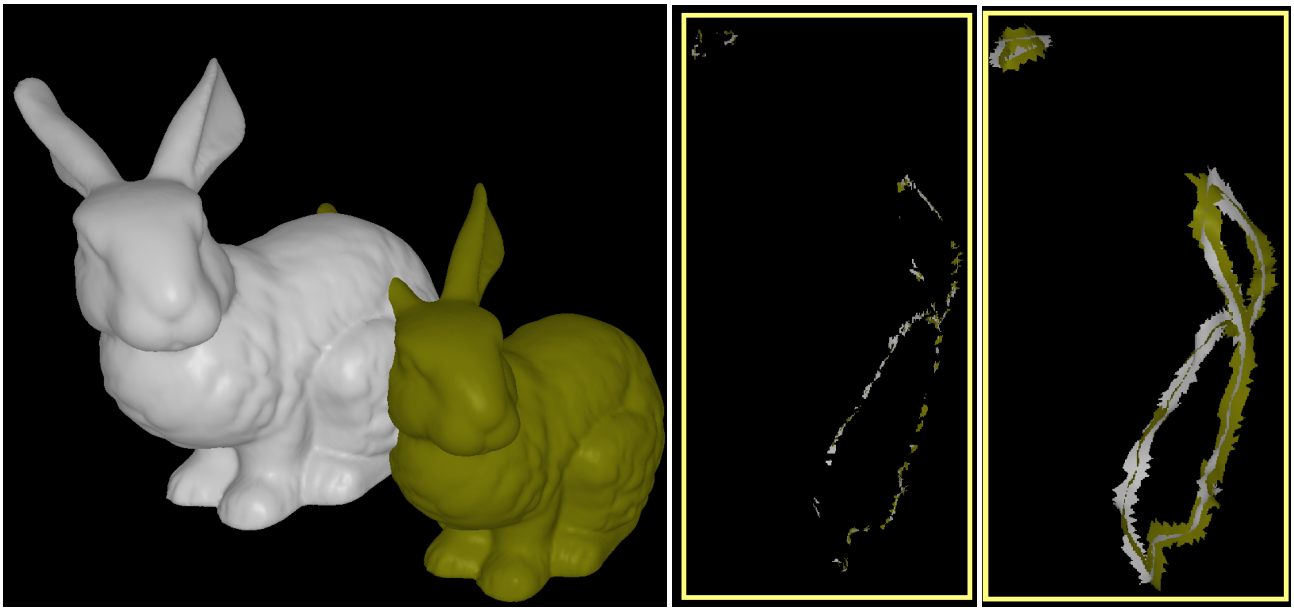
We have compared our algorithm with a CPU-based implementation SOLID [30] on the environment with dynamically generated breaking objects. SOLID is a publicly available library that uses pre-computed AABB-trees for collision culling. As the objects in our scene are dynamically generated and the topology of existing objects (eg. dragon) change, we need to dynamically compute the hierarchies. Moreover, as the hierarchies are recomputed, there is an additional overhead of allocating and deallocating memory in SOLID. Ignoring the overhead due to memory, we observed that the pre-computation of data-structures for SOLID require $100 - 176$ ms per frame. These timings do not include the pruning time. On the contrary, we are able to compute all collisions within 50ms (including UOBB computation time and pruning time).

We have also compared our algorithm with an optimized GPU implementation of CULLIDE. Our implementation runs nearly 3 times slower due to the overhead of rasterizing bounding boxes instead of triangles. However, as shown in Fig. 7, CULLIDE misses several interferences and may lead to inaccurate simulations.

6. ANALYSIS AND LIMITATIONS

Three key issues exist related to the performance of conservative collision culling algorithms: efficiency, level of culling, and precision.

Efficiency: Three factors govern the running time of our algorithm: bounding offset computation, rendering the bounding offsets and occlusion queries. The cost of computing the OBBs for each primitive is very small. The cost of rendering the OBBs on the GPUs is mainly governed by the transformations. In our current implementation, we have achieved rendering rates of $40M$ triangles per second. Finally, our algorithm uses occlusion queries to perform



(a) *Interference computation on two bunnies* (b) *Intersection curve computed by CULLIDE* (c) *Intersection curve computed by FAR*

Figure 7: ***Reliable interference computation: This image highlights the intersection set between two bunnies, each with*** 68***K triangles (shown in Fig 7(a)). Fig. 7(c) shows the output of FAR and Fig. 7(b) highlights the output of CULLIDE running at a resolution of*** 1400×1400***. CULLIDE misses many collisions due to the viewport resolution and sampling errors.***

VO queries. These queries can be fill bound for large objects. The current implementation of these queries is not optimized, yet we are able to perform 1.2 million queries per second. FAR is able to compute all the collisions between models composed of tens of thousands of primitives at interactive rates. In more complex environments (e.g. with millions of triangles), rendering and occlusion queries can become a bottleneck. However, given the growth rate of GPU performance (at a rate faster than Moore's law) and increasing bus bandwidth based on PCI-X, we expect that our algorithm can handle more complex models in the near future.

Culling: The effectiveness of most collision detection algorithms depends on how efficiently they can cull away the primitives that are not in close proximity. FAR uses union of OBBs as the underlying bounding volume and is less conservative as compared to CPU based algorithms that use AABBs or spheres to bound the primitives (see Fig. 6).

Precision: Our culling algorithm is conservative and its precision is governed by that of the VO queries. The accuracy of the culling algorithm is equivalent to that of the floating point hardware (e.g. 32-bit IEEE floating point) inside the GPUs used to perform transformations and rasterization. The precision is not governed by viewport resolution or depth-buffer precision.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a reliable GPU-based collision culling algorithm. We use bounding offsets of the primitives to perform visibility-based 2.5D queries and cull away primitives that are not in close proximity. Our new algorithm overcomes a major limitation of earlier GPU-based collision detection algorithms and is able to perform reliable interference queries. Furthermore, the culling efficiency of our algorithm is higher as compared to prior CPU-based algorithms that use AABBs or spheres for collision culling. Moreover, the culling efficiency and performance can be significantly enhanced by using Quick-CULLIDE [24]. We have demonstrated its performance in complex scenarios where objects undergo rigid and non-rigid motion.

Desirable Hardware Features: We propose a simple architecture for the graphics pipeline to accelerate the performance of our algorithm. The modified architecture requires the following characteristics:

- **Precision:** In order to obtain floating point precision, the graphics pipeline should support floating point depth buffers. However, it is important to note that the viewport resolution is mainly responsible for the sampling errors than the depth buffer precision. Therefore, floating point depth buffers alone cannot solve the sampling problem.
- **Rasterization Rules:** We set a state in which the following rasterization rules are used. These rules are used to overcome the viewport resolution problems.
  - A fragment is generated for each pixel that a triangle touches.
  - For each pixel, depth is computed at all the four corner samples of the rectangular pixel. A *depth set* function is applied onto the four depth samples and one of the four values is output as the depth of current fragment. The depth set function could either be $\{max, min\}$ and is specified as a state before rasterizing a primitive. The function *max* computes the maximum value of the four depth samples and *min* computes the minimum value of the four depth samples.

The above rules are sufficient to design an algorithm ensuring floating point precision for interference computations. In the pseudo-

code described in section 4.2.1, while rendering a primitive to the frame buffer, we set the depth set function to *min* along with the depth function *GL_LEQUAL*. This operation ensures that for each pixel touched by a primitive, we compute the minimum depth of *all* points of the primitive that project onto the pixel. While testing the fully visible status of a primitive, the depth set function is set to *max*. This operation ensures that we test if the maximum depth of all points of a primitive that project onto the pixel is fully visible or not. It is easy to see that these two operations can be used to conservatively test if a primitive interferes with another primitive or not.

Most graphics hardware implementations involve tile-based rasterization [1, 15, 20, 22]. All the pixels covered by a primitive and within a tiled region, say a region consisting of 4×4 pixels are computed before moving to the next tile. As adjacent pixels share common sample points, it is possible to design a simple architecture computing the depth at a sample point, say left corner of a pixel, and depth values at the samples covering the top and right corners of a tile. A simple hardware can be used to compute the max or min values of these fragments in a tile and output the sample depths.

The proposed implementation requires the computation of more samples than the actual number of samples in normal rasterization pipeline. However, the overhead of this computation can be minimized by using additional hardware.

In terms of future work, we would like to develop reliable and accurate GPU-based geometric algorithms for other proximity queries such as penetration and distance computation, as well as visibility and shadow computations.

Acknowledgements

Our work was supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134 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 the hardware and driver support. We are thankful to Kelly Ward for helping us with video editing and members of UNC GAMMA group for useful discussions.

8. REFERENCES

[1] Tomas Akenine-Moller and Jacob Strom. Graphics for the masses: a hardware rasterization architecture for mobile phones. *ACM Trans. Graph.*, 22(3):801–808, 2003.

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

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

[4] G. Barequet, B. Chazelle, L. Guibas, J. Mitchell, and A. Tal. Boxtree: A hierarchical representation of surfaces in 3D. In *Proc. of Eurographics'96*, 1996.

[5] 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.

[6] 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.

[7] 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.

[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] Alexander Gress and Gabriel Zachmann. Object-space interference detection on programmable graphics hardware. In *SIAM Conf. on Geometric Design and Computing*, Seattle, Washington, November13–17 2003.

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

[11] M. Held, J. Klosowski, and J. S. B. Mitchell. Real-time collision detection for motion simulation within complex environments. In *Proc. ACM SIGGRAPH'96 Visual Proceedings*, page 151, 1996.

[12] 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.

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

[14] P. Jimenez, F. Thomas, and C. Torras. 3d collision detection: A survey. *Computers and Graphics*, 25(2):269–285, 2001.

[15] B. Kelleher. Pixelvision architecture. Technical Report 1998-013, Digital Systems Research Center, 1998.

[16] Y. J. Kim, M. A. Otaduy, M. C. Lin, and D. Manocha. Fast penetration depth computation for physically-based animation. *Proc. of ACM Symposium on Computer Animation*, 2002.

[17] 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.

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

[19] M. Lin and S. Gottschalk. Collision detection between geometric models: A survey. *Proc. of IMA Conference on Mathematics of Surfaces*, 1998.

[20] Joel McCormack and Robert McNamara. Tiled polygon traversal using half-plane edge functions. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 15–21. ACM Press, 2000.

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

[22] S. Morein. ATI Radeon HyperZ technology. In *ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, Hot3D Proceedings*, 2000.

[23] 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.

[24] M. Lin N. Govindaraju and D. Manocha. Quick-cullide: Fast inter- and intra-object collision culling using graphics hardware. Technical report, University of North Carolina at Chapel Hill, July 2004.

[25] 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.

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

[27] Ramesh Raskar and Michael Cohen. Image precision silhouette edges. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 135–140. ACM Press, 1999.

[28] 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.

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

[30] Freesolid: Software library for interference detection. http://www.win.tue.nl/ gino/solid/.

[31] 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.