2007-01-12

System for Collision Detection Between Deformable Models Built on Axis Aligned Bounding Boxes and GPU Based Culling

David Owen Tuft
Brigham Young University - Provo

A SYSTEM FOR COLLISION DETECTION BETWEEN DEFORMABLE MODELS BUILT ON AXIS ALIGNED BOUNDING BOXES AND GPU BASED CULLING

by

David O. Tuft

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

April 2007

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

David O. Tuft

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Parris K. Egbert, Chair
Date	Bryan S. Morse
Date	Eric G. Mercer

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of David O. Tuft in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Parris K. Egbert
Chair, Graduate Committee

Accepted for the Department

Parris K. Egbert
Graduate Coordinator

Accepted for the College

Thomas W. Sederberg
Associate Dean,
College of Physical and Mathematical Sciences

ABSTRACT

A SYSTEM FOR COLLISION DETECTION BETWEEN DEFORMABLE MODELS BUILT ON AXIS ALIGNED BOUNDING BOXES AND GPU BASED CULLING

David O. Tuft

Department of Computer Science

Master of Science

Collision detection between deforming models is a difficult problem for collision detection systems to handle. This problem is even more difficult when deformations are unconstrained, objects are in close proximity to one another, and when the entity count is high. We propose a method to perform collision detection between multiple deforming objects with unconstrained deformations that will give good results in close proximities. Currently no systems exist that achieve good performance on both unconstrained triangle level deformations and deformations that preserve edge connectivity.

We propose a new system built as a combination of Graphics Processing Unit (GPU) based culling and Axis Aligned Bounding Box (AABB) based culling. Techniques for performing hierarchy-less GPU-based culling are given. We then discuss how and when to switch between GPU-based culling and AABB based techniques.

ACKNOWLEDGMENTS

I would like to thank Parris Egbert for his advice, encouragement, and support as I finished my thesis long distance in North Carolina. I would also like to thank my committee for working with me.

I would also like to thank my dear parents and wife for support and encouragement.

Contents

Acknowledgments **vi**

List of Tables **ix**

List of Figures **xii**

1 Introduction **1**

1.1 Axis-Aligned Bounding Boxes 2

1.2 GPU Based Culling 2

1.3 Summary of Contributions 3

1.4 Thesis Organization 4

2 Related Work **5**

2.1 Collision Detection 5

2.2 Rigid Body Collision Detection 6

2.3 Bounding Volumes and Hierarchies 8

2.4 Deformable Models 8

3 Background and Problem Analysis **11**

3.1 Axis-Aligned Bounding Boxes 11

3.2 GPU-Based Culling 12

3.2.1 GPU Features 13

3.2.2 GPU-Based Culling Algorithm 15

3.3 Analysis 18

3.3.1 Geometric Coherency 19

3.3.2 Geometric Proximity 21

3.3.3 Deformable Model Collision Detection Bottlenecks 22

4 The Combined AABB and GPU Culling System 25

4.1 Overview 25

4.2 GPU-based culling 25

4.2.1 Determining The Proper View Configurations 26

4.2.2 Object Sort 30

4.2.3 Triangles Per Query Determination 32

4.3 A Framework built on GPU and AABB Culling 34

4.3.1 Framework Analysis 36

4.4 Deciding Which Culling to Use 39

4.4.1 Mode 1 40

4.4.2 Mode 2 41

4.4.3 System 41

5 Results 43

5.1 Hierarchy-Less Collision Detection 43

5.1.1 Object Sort 44

5.1.2 View Configurations 44

5.1.3 Triangles Per Occlusion Query 44

5.1.4 Final Results 45

5.2 Combined GPU and AABB System 45

5.2.1 Data Set Results 46

6 Conclusions 57

Bibliography 61

List of Tables

3.1 AABB Hierarchy Bottlenecks 22

4.1 Triangles per Occlusion Query Variables 34

4.2 AABB and GPU Cull Framework 36

4.3 Collision Detection Times 37

4.4 Decision System Variables 39

4.5 Decision System Functions 39

List of Figures

2.1 Worst-case Triangle Test 5
2.2 Simple Hierarchy 6
2.3 Sweep and Prune 7
2.4 BART Ray Tracing Benchmark 10
3.1 AABBs Edge Connectivity 12
3.2 AABBs No Edge Connectivity 13
3.3 AABB Overlap 14
3.4 Simplified GPU-Pipeline 15
3.5 Depth Buffer 16
3.6 Occlusion Query 17
3.7 Depth Complexity 18
3.8 Collision Detection Coherency 19
3.9 Random Triangle Movement 20
3.10 Two Objects 21
3.11 Collision Detection Proximity 22
4.1 Two Bunnies Colliding 26
4.2 Time For Different View Configurations 27
4.3 Two Sheets in Close Proximity 28
4.4 Time For Different View Configurations. 29
4.5 Effect of Sorting Objects before the GPU Cull 31
4.6 Times to Perform Occlusion Queries 32
4.7 AABB GPU-Cull Framework 35
4.8 Triangle Intersections Needed 38
5.1 Visualization of Data Sets 48

5.2 Results for Sorting Objects 49
5.3 View Configuration Results 50
5.4 Results of Triangles Per Occlusion Query 51
5.5 Final Hierarchy-less Results 52
5.6 Results for Data Set A, Two Bunnies 53
5.7 Results for Data Set B, Two Waving Sheets 53
5.8 Results for Data Set C, Sin Wave Sheets 54
5.9 Results for Data Set D, BART Animation 54
5.10 Results for Data Set E, Exploding dragons, Over 25 Frames 55
5.11 Results for Data Set E, Exploding dragons, Over 50 Frames 55
5.12 Results for Data Set E, Exploding dragons, Over 100 Frames 56
5.13 Results for Data Set E, Exploding dragons, Over 200 Frames 56

Chapter 1

Introduction

Collision detection is the algorithmic process of determining if two solids intersect. Collision detection adds realism to simulations, interactive games, and movies. The most difficult types of collision detections are those in which the objects are in close proximity to one another and those in which the objects are deformable. Deforming models are extremely important in the simulation and animation industry. Film and animation companies have begun integrating more complicated simulations of fluids and other materials into scenes. Collision detection is an essential part of any realistic simulation. If an animated car crashes into a building, the simulation to bend and tear the frame of the car is dependant upon where the deforming pieces of the car collide.

Collision detection is a well-studied problem with most methods using hierarchical representations and bounding primitives to reduce the complexity of the intersection tests. However, current systems do not work well on all types of collision detection. Many of them are geared towards special cases and do not attempt to handle all types of deformable models. To solve this problem, we approach deformable model collision detection in terms of Axis-Aligned Bounding Boxes (AABB)s and Graphics Processing Unit (GPU) based culling. We combine AABBs and a GPU based Culling technique in order to perform collision detection on a wide array of model types, ranging from rigid bodies to simulations that require hierarchy-less collision detection.

1.1 Axis-Aligned Bounding Boxes

AABBs are a simple and elegant approach to collision detection and have been in use for a long time. Fast AABB updates are a powerful approach to deformable model collision detection [1]. However, there are two setbacks to the AABB approach. First, AABBs do not bound primitives tightly. This can result in extra primitive-primitive intersections. Second, AABB updates can cause the AABB hierarchy to become inefficient. Our system uses GPU culling in conjunction with AABBs in order to compensate for weaknesses inherent in each system.

1.2 GPU Based Culling

A key part of modern collision detection systems is culling, which is the act of pruning the set of objects that must be tested for collisions. Spatial bounding and partitioning schemes such as grids, bounding volume hierarchies, and space partitioning trees can be used for culling objects. Successful culling schemes reduce the overall time required for collision detection systems.

GPU-based culling techniques are an alternative approach to hierarchy-based collision-detection techniques. GPU-based culling techniques such as Cullide [2] render objects using occlusion queries and trivially cull objects that are determined not to intersect. By using the occlusion query capability of the GPU in this fashion, some of the non-colliding objects in close proximity to one another can be quickly culled from the collision detection set. This approach has many shortcomings. Rendering visibility queries has a high overhead. Also, the amount of visibility query throughput on modern graphics cards has not increased while the amount of geometry possible to render per query has gone up. This makes a naive implementation of one occlusion query per triangle inefficient. GPU culling is also very sensitive to views from which the objects are rendered. Using a view with high depth complexity will dramatically decrease the system's efficiency. Moreover, the order of rendering makes a difference in the effectiveness of the GPU cull. Since it is impractical to require the end user to specify the order in which geometry is rendered, GPU culling can perform well below optimal, based on these variables.

1.3 Summary of Contributions

This thesis approaches deformable model collision detection with AABB-based culling and GPU-based culling. We present algorithms and metrics for determining an optimal GPU cull. This improved GPU cull is used for hierarchy-less collision detection and for cases where AABB-based culling breaks down. Our analysis shows that AABBs have difficulties in two major areas:

1. AABBs can become become primitive intersection bound.
2. AABBs can become AABB intersection bound.

The vast majority of collision detection systems use triangles as the basic primitive. Fast triangle-triangle intersection code has been made available [3]. In many cases the triangle-triangle intersections are very fast. However, when many triangles are in close proximity, the triangle-triangle tests can become a bottleneck as AABBs do not tightly bound triangles.

Unconstrained triangle deformations are another major issue for AABB-based techniques and cause AABBs to become AABB intersection bound. Most AABB-based techniques require hierarchies to be rebuilt, or constrain the geometry to preserve edge connectivity.

While GPU based culling techniques are the best hierarchy-less collision detection techniques, there are other overhead issues associated with them.

1. The view direction from which objects are rendered affects cull efficiency.
2. The order in which primitives are rendered affects cull efficiency.
3. GPU culling does not always cull sufficiently to justify the cost.
4. Visibility hardware is geared towards multiple triangles per query.

This thesis has four main contributions. The first contribution consists of three mechanisms which make GPU-based culling more efficient. Mechanisms are provided for picking optimal view directions, sorting objects to give better culling,

and determining the number of triangles to render per visibility query. The second contribution is our framework to perform combined GPU-based and AABB-based culling and the analysis of what limits this technique. The third contribution is our technique for rebuilding the hierarchy in one thread while performing GPU-based culling in another thread. The final contribution is our algorithm for performing collision detection based on timing metrics. Our simulations have shown that this system performs as well or better than traditional AABB-based collision detection or GPU-based culling on a wide variety of models.

1.4 Thesis Organization

Chapter 2 lays the ground work by discussing previous work in collision detection. Chapter 3 gives a more detailed background on the specific types of collision detection that will be used in this thesis. This chapter also gives an analysis of deformable model collision detection. The analysis frames the problem of deformable model collision detection in terms of geometric proximity or lack of geometric coherency. Chapter 4 outlines our contribution to GPU based culling as well as our systems for performing collision detection. Chapter 5 discusses the results, and Chapter 6 provides conclusions.

Chapter 2

Related Work

2.1 Collision Detection

Testing a single triangle for collision against a set of triangles is inherently an $O(N)$ operation where N is the number of triangles in the set. Figure 2.1 represents a worst-case scenario for collision detection. In this figure all pairwise triangles are colliding. In this case it is impossible to solve collision detection in less than $O(N)$ time. In a typical scene, most triangles are not colliding. Collision-detection algorithms leverage off of this to reduce the complexity of collision detection. Figure 2.2 shows a simple hierarchy. In this hierarchy, the number of collision detection tests for the triangle in blue is reduced from $O(N)$ to $O(\log N)$.

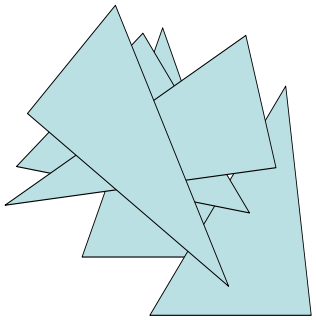

Figure 2.1: **Worst-case Triangle Test** *This example represents a worst-case scenario for collision detection. This example is in 2D. In this case all triangles are colliding.*

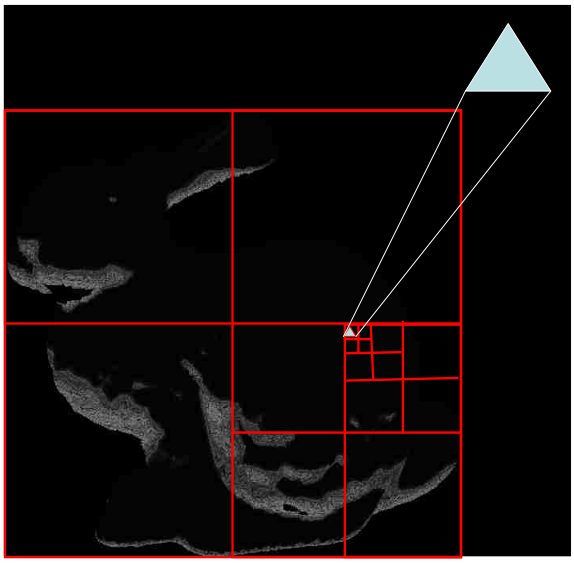

Figure 2.2: **Simple Hierarchy** *Hierarchies reduce the number of intersection tests required in most cases.*

2.2 Rigid Body Collision Detection

Collision detection developed simultaneously with the simulation field. Collision detection is rooted in mathematical science. Mathematic algorithms for determining if two geometric primitives intersect have been around for a long time. Garcia-Alonso et al. computed collision detection by first computing collisions between bounding volumes. For those objects whose bounding volumes collided, voxel grids were tested for collision detection. Finally facets within colliding grids were checked. In this way they were able to cull collisions [4].

Moller et al. presented a fast triangle to triangle intersection algorithm. This algorithm has been used in a large number of collision detection systems for the triangle intersection test [3]. This algorithm is the de facto technique for performing collision detection between triangles. The code supplied by the authors has been used as the basic primitive collision detection intersection test in many systems [5, 6].

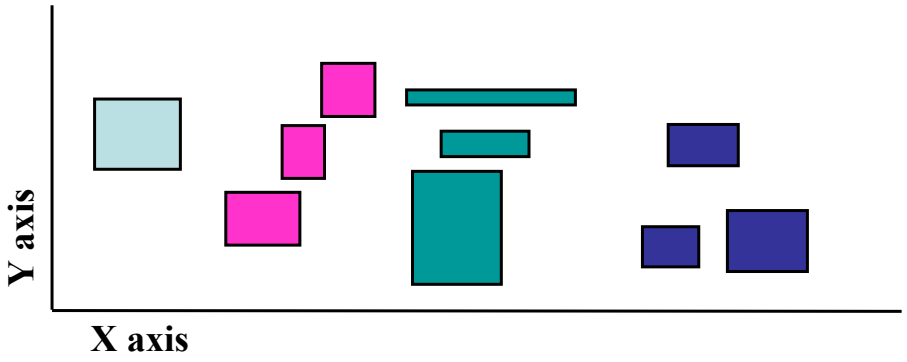

Figure 2.3: **Sweep and Prune** *Sweep and prune groups objects along an axis based on minimum and maximum extents.*

I-Collide is a collision detection system for large-scale environments with objects undergoing rigid body motion [7]. I-Collide introduced some new techniques for skipping groups of intersection tests. The sweep and prune algorithm is introduced as a technique that iterates across primitives or objects sorted with respect to minimal values in an axis direction. The algorithm compares the minimum and maximum extents along an axis to determine which objects can be pruned from the set of objects that potentially collide. This algorithm allows for the set of objects that must be tested against each other to be greatly reduced. Figure 2.3 shows how the minimum and maximum extents of bounding volumes can be grouped into smaller sets.

Lin et al. describe collision detection for polygonal objects, spline or algebraic surfaces, CSG models, and deformable bodies. This paper focuses on how the model representation leads to different types of collision detection [5].

Jimenez et al. split collision detection into four categories based on the locations and the trajectories of moving objects. Knowing the category in which a potential collision falls can help in reducing the collision detection costs [8].

2.3 Bounding Volumes and Hierarchies

Axis Aligned Bounding Boxes (AABBs) have been commonly used in graphics [9]. AABBs are boxes where the edges of the box are perpendicular to the coordinate axis. AABBs are stored as two points: a minimum and a maximum. In 3D this amounts to 6 floating-point numbers or 24 bytes. Bounding boxes can be fit around primitives by simply determining the minimum and maximum vertices of the primitives. A major benefit of AABBs is the trivial overlap tests for detecting collisions between AABBs. Figure 2.2 represents a hierarchy of AABBs.

AABBs are often used in collision-detection schemes. Van den Bergen describes a system in which a hierarchy of AABBs is used [10]. This hierarchy is transformed with the geometry for any affine transformation. Transforming the AABB hierarchy converts the hierarchy to an Object-Oriented Bounding Volume (OOBV), which may or may not be axis aligned. This eliminates the need to rebuild or update the hierarchy for affine transformations but increases the complexity of the intersection tests as OOBV intersections tests are more complicated operations. In the case of deformations, Van Den Bergen updates the bounding volume from the leaf node up. This has been shown to be a fast operation on modern CPUs. If the deformations are minimal, this technique works well. If the deformations are large and the object components do not maintain connectivity, this can become much more compute intensive.

2.4 Deformable Models

Deformable model collision detection complicates the collision detection process. Deformable models are those models in which the underlying primitives move from frame to frame. Deformable models range from models that change very little during a scene, and are more easily dealt with, to models that under go incoherent and random transformation at the triangle level.

Because of the movement that triangles can undergo, bounding volumes and hierarchies become useless without some form of rebuild or update. Most deformable model collision detection systems constrain the types of deformations an object can

experience, and then leverage heavily off of current non-deformable collision detection work [11, 12, 1]. Figure 2.4 shows an animation that is part of the Benchmark for Animated Ray Tracing (BART). Key frames 1–6 show triangle movement that appears random. The type of animation in frames 1–6 is the most difficult type of animation for collision detection algorithms. Many triangles are colliding. Triangles are in close proximities and triangles are overlapping. A final complication is that triangles are moving and deforming. Frames 1-6 of the BART animation require expensive hierarchy rebuilds or hierarchy-less collision detection.

Frames 7–9 are much simpler cases to deal with. Edge connectivity is preserved for the majority of triangles. Because there is high coherency between frames, and most triangles retain connectivity, hierarchy-based methods often outperform hierarchy-less methods in examples such as this. The BART benchmark shows that multiple techniques can be used even during the course of one animation to take advantage of the type of deformations occurring.

This chapter has discussed the basics of collision detection and given an introduction to rigid bodies and deformable models. Because the main thrust of this thesis is deformable model collision detection, we next give more details about the current state of this field. Following that discussion, we will describe the challenges in deformable-model collision detection, then present our solution to these challenges.

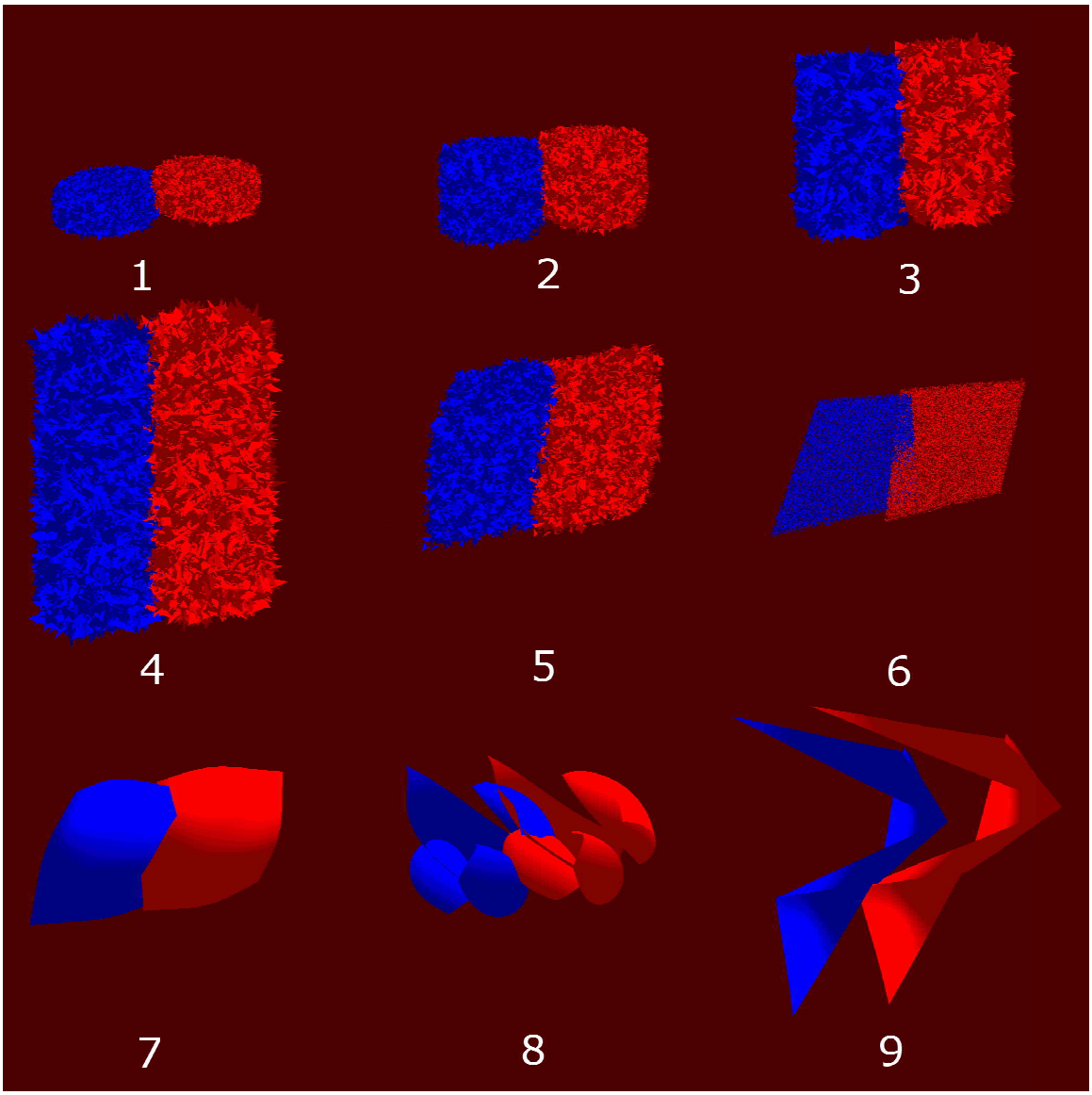

Figure 2.4: **BART Ray Tracing Benchmark** *A highly deformable animation from the BART Ray tracing benchmark.*

Chapter 3

Background and Problem Analysis

3.1 Axis-Aligned Bounding Boxes

As has been mentioned earlier, AABBs are frequently used in collision detection and ray tracing. Larsson and Akenine-Moller showed that AABBs can be used to perform collision detection on deforming models. They first compute an AABB hierarchy as a binary tree, quadtree, or octree [11]. Their method builds the tree bottom-up and preserves connectivity. Their algorithm requires that the deforming bodies preserve edge connectivity. This technique uses a mixture of top-down AABB updates and bottom-up AABB updates. The tree is updated top-down for the first N levels. Next, collision detection is performed on the first N levels of all object hierarchies in the scene. If there are intersections at level N of the tree, the rest of the tree is updated in a bottom-up fashion to guarantee accuracy. This technique is able to cull portions of the scene that do not intersect without doing a full bottom-up update. However, the authors state that this can sometimes be more expensive than updating the entire hierarchy in a bottom-up fashion.

AABB updates are efficient for many types of deforming models. Figure 3.1 shows a deformation where edge connectivity is preserved. While the new bounding structure might not be the optimal AABB hierarchy, it is still efficient. Figure 3.2 shows a scenario where AABBs are fit around geometry but the triangles separate. In this case the updated hierarchy is poor. The case where edge connectivity is lost is common in any type of deformation involving breaking, tearing, or exploding.

In this case it is better to rebuild the hierarchy which takes $O(\log N)$ time

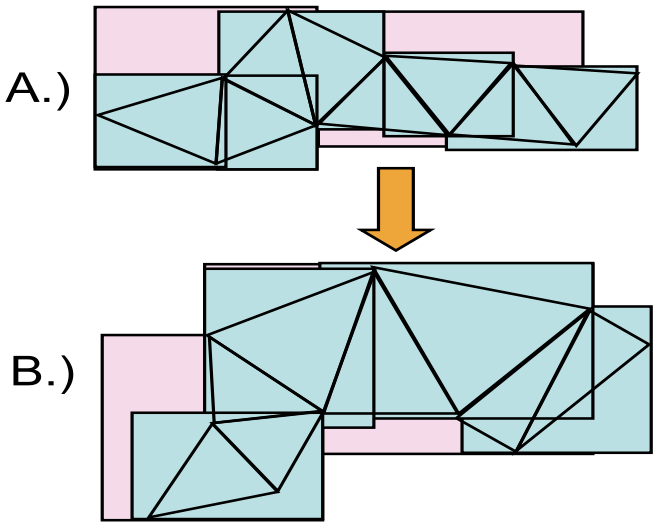

Figure 3.1: **AABBs Edge Connectivity** *When edge connectivity is preserved AABBs are efficient bounding primitives.*

rather than use an inefficient hierarchy. However, this is an expensive operation. Because of the inability to handle highly deformable models, AABBs are not a viable solution for all types of deforming models.

Another disadvantage of AABBs is that they are less efficient when objects are in close proximity. This is due to the shape of AABBs, which do not bound objects as efficiently as other primitives, such as OOBBs. Figure 3.3 shows a case where the primitives' bounding boxes intersect but the primitives do not. There are other techniques that are more efficient at culling triangle - triangle intersections than AABBs. The next section explains one such technique.

3.2 GPU-Based Culling

GPU-based culling reduces the set of objects that must be tested for collision detection. This is done by running visibility queries on the graphics card between individual triangles and the rest of the triangles in the set. These algorithms are multi-pass. Each pass consists of multiple renders and state changes on the GPU.

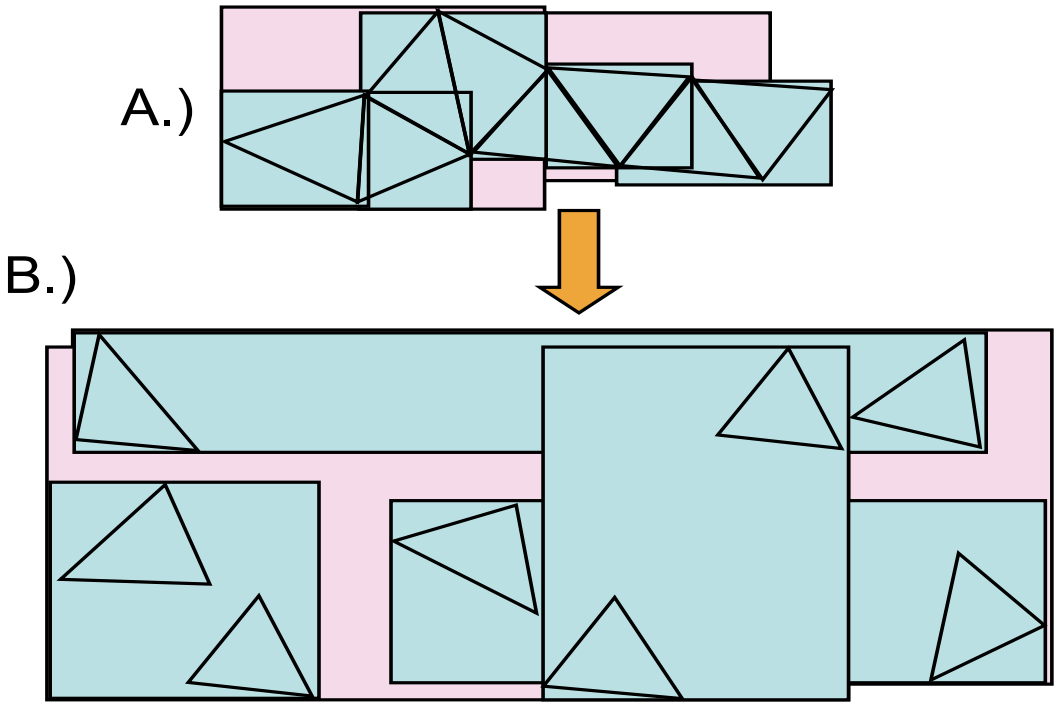

Figure 3.2: **AABBs No Edge Connectivity** *In this example of AABBs without edge connectivity the AABBs are updated but do not bound efficiently*

Because the culling is at the triangle level, the culling is more efficient than AABB-based culling. The theoretical complexity of GPU-based culling is $O(N)$.

3.2.1 GPU Features

Because of the parallel nature of graphics, GPUs have been progressing at a rate faster than Moore's law. While GPUs were not designed with the intent of performing collision detection, some of the key features of GPUs can be harnessed to perform collision detection. The GPU is special purpose hardware designed for rendering objects very quickly. Taking advantage of this special purpose hardware requires finding algorithms that are similar in nature to the algorithms for which this special purpose hardware was designed.

Figure 3.4 shows some of the key features of a modern GPU pipeline. At the

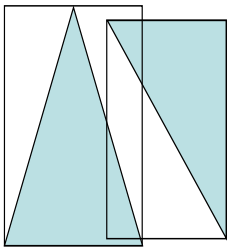

Figure 3.3: **AABB Overlap** *Many times AABBs collide when the primitives they bound do not*

top of the figure are the vertex processors. These processors are Multiple Instruction Multiple Data (MIMD) units and operate on the data associated with each vertex. The rasterization engine breaks the data up into fragments. A fragment is a generalization of a pixel. Pixels denote one dot on a screen. A fragment can be an element in a color buffer, a texel in a texture map, or a depth value in a depth buffer. The fragments are then fed into the fragment processors. These processors operate on all the data associated with a fragment. The depth and color information is passed to the fixed function Raster Operators (ROPs). The ROP units perform fixed function operations between the fragments and the depth buffer. Figure 3.5 represents the depth buffer. This 2D array of values is used in Z-Buffer rendering to determine the pixels nearest the viewer. Each pixel in the grid has a 24-bit value associated with it. This 24 bit value is scaled between 0 and 1. The objects nearest to the viewer are represented by 0. the objects farthest from the viewer are represented by 1. Incoming fragments are compared against the depth buffer. If they pass the depth test, the depth buffer and color buffer are overwritten.

The ROP units compare the fragments of individual triangles against the depth buffer to determine visibility. The ROP units are very specialized and highly efficient. These units are able to compare a great number of incoming pixels against the depth buffer. This is a vital part of Z-buffer rendering.

Another option available on graphics cards is the occlusion query. Figure 3.6

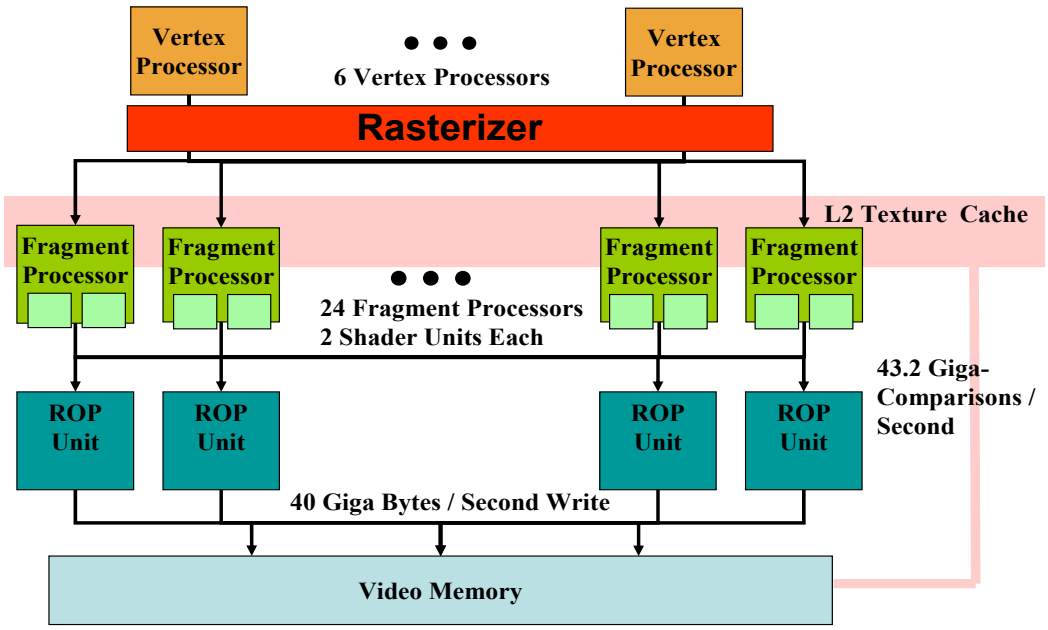

Figure 3.4: **Simplified GPU-Pipeline** *In this abstraction of a GPU-pipeline, the Vertex Processors transform triangles which are then rasterized into fragments. Fragments are shaded and fed into the ROP units. Each ROP unit performs fixed function updates to the depth, stencil, and color buffers.*

shows a buffer with a triangle rasterized into it. Algorithm 1 shows the code for rendering with occlusion queries. When the code in Algorithm 1 is run, Figure 3.6 is generated and 15 pixels are returned from the occlusion query.

3.2.2 GPU-Based Culling Algorithm

With the advent of programmable graphics hardware many techniques have been developed that entail rendering objects to the depth buffer and then using the information in the depth buffer or color buffer to do screen space analysis of the scene. Govindaraju et al. were the first to propose using the GPU to cull objects that did not collide [2]. Their system, known as Cullide, makes use of the depth buffer but

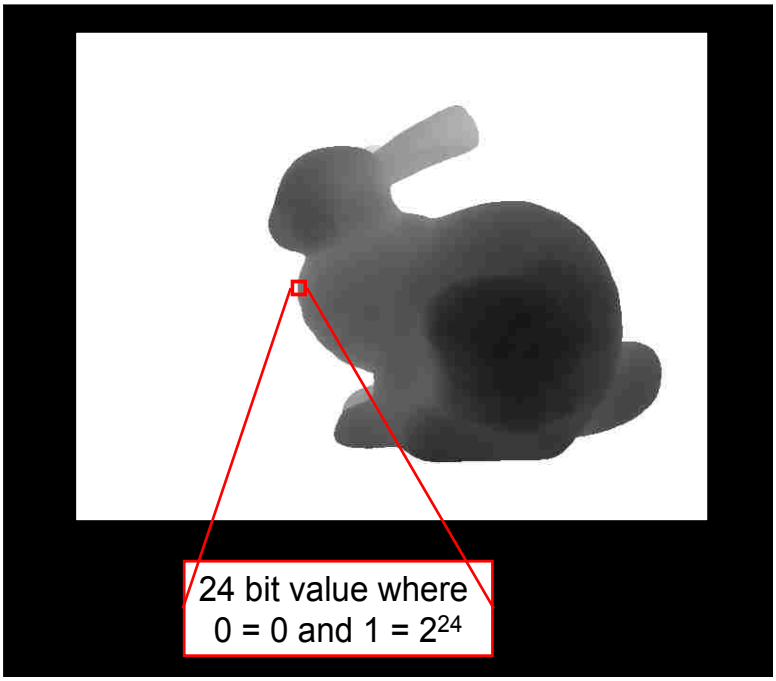

Figure 3.5: **Depth Buffer** *The depth buffer is 2D array of 24 bit values representing the values between 0 and 1.*

does not require an expensive depth buffer read. Cullide operates at the triangle level, and does not require a hierarchy. It has been shown to be useful on both objects in close proximity one to another and on deforming models [2].

GPU-Based culling renders objects to a depth buffer in any order and then renders the same objects in the opposite order. GPU-Based Culling relies on hardware occlusion queries to determine if pixels were rasterized. When no pixels are rasterized for a particular object in both passes, that object can safely be removed from the list of potentially colliding objects. The end result is a much smaller set of objects on which to perform collision detection.

A high level overview of GPU culling is as follows. Two flags are stored for each object. These flags are: First pass Fully Visible (FFV) and Second pass Fully Visible (SFV). The two flags are initialized to false. The culling is done by using a multi-pass approach consisting of two renders per object per pass. The first render is used to check the object against the depth buffer. The second render renders the

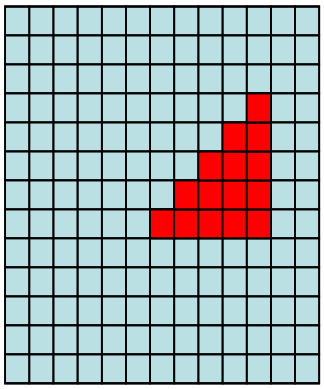

Figure 3.6: **Occlusion Query** *An occlusion query returns the number of pixels rasterized to the screen. The occlusion query for the triangle returns the number 15.*

Algorithm 1 Occlusion Query Code

```
glBeginOcclusionQueryNV(_occ_queries[0]); // (1) Begin Occlusion Query
  glBegin(GL_TRIANGLES); // (2) Draw Primitives
    glVertex2i(6, 5);
    glVertex2i(10, 5);
    glVertex2i(10, 9);
  glEnd();
glEndOcclusionQueryNV();// (3) End Occlusion Query
glGetOcclusionQueryuivNV(_occ_queries[inner],GL_PIXEL_COUNT_NV,
&pixelCount);// (4) Read back results of Occlusion Query
```

object into the depth buffer for consequent objects to be checked against. The second pass is the same, but the order of the objects rendered is reversed. After these two passes all objects with both the FFV and SFV flags set to true are removed from the PCS. This process is repeated from multiple views to further reduce the PCS. In practice we use one to three axis-aligned views.

GPU-based culling operates at the triangle level. It therefore does not require a hierarchy. GPU-based culling can also cull objects in close proximity as it culls against individual primitives. GPU-based culling is not without penalties. The GPU-cull has the overhead of the render calls for all triangles in the scene. Each time the GPU-cull is called it could render triangles multiple times. Current NVIDIA graphics

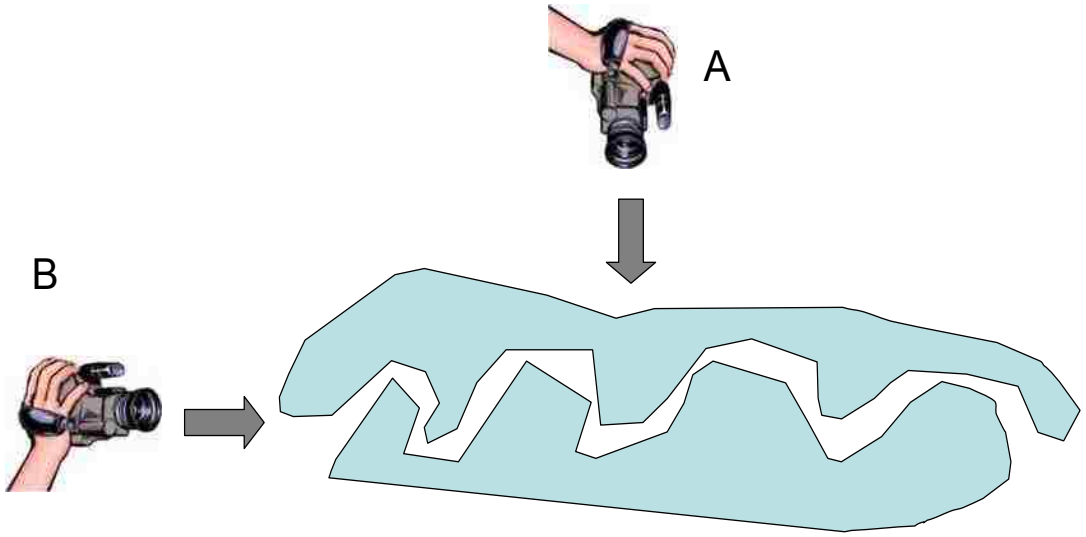

Figure 3.7: **Depth Complexity** *View A has a lower depth complexity than View B.*

cards can perform 1.3 million occlusion queries per second. This makes the cost of an occlusion query under a microsecond. This is fast, but it is still a penalty that must be paid per visibility query. It can also be difficult to determine the best view direction. Figure 3.7 demonstrates the importance of finding a view with low depth complexity. If view A is chosen, the depth complexity will be low, and GPU culling will cull the set to 0 in 1 pass. If view B is chosen, GPU culling will take 8 passes to cull all triangles.

3.3 Analysis

In this section we analyze collision detection in terms of geometric coherency of the primitives and geometric proximity of the primitives. These are the two most important factors in collision detection. This analysis will outline the factors that go into determining the correct collision detection methods. Section 3.3.1 describes coherency. Section 3.3.2 describes proximity. Section 3.3.3 outlines our algorithm for determining the correct collision detection technique based on coherence and proximity. In Chapter 4 we present our collision detection system and improved version of

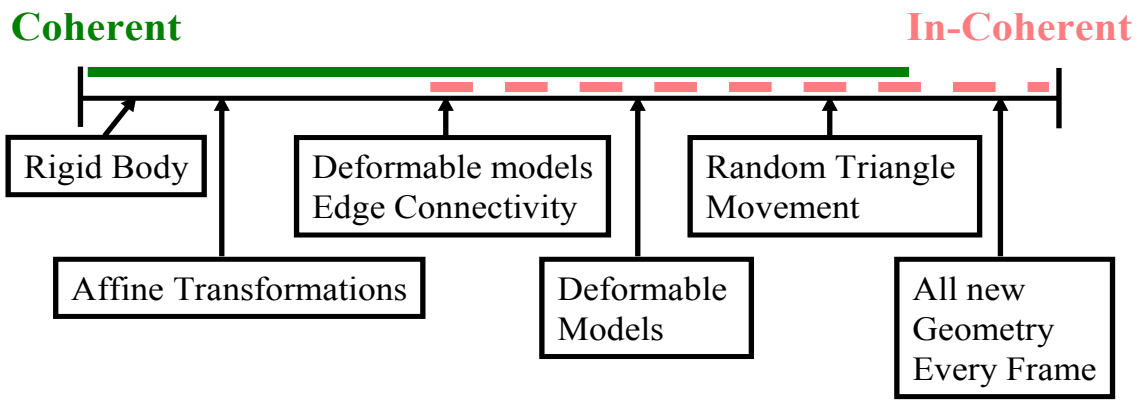

Figure 3.8: **Collision Detection Coherency** *Most collision detection schemes take advantage of coherency between frames. As Coherency decreases, collision detection schemes must focus on hierarchy rebuilding, hierarchy updates, and hierarchy-less collision detection schemes.*

GPU culling as the contribution of this thesis.

3.3.1 Geometric Coherency

We use the term "geometric coherency" to describe the amount of change that primitives undergo from one frame to the next with respect to the neighboring primitives. If all the primitives move together, the coherency is high. When triangles move away from one another the coherency is very low. Figure 3.8 outlines the different types of model coherency. This scale ranges from extremely coherent to incoherent. The most coherent are those bodies that do not change from frame to frame. Models transformed by affine transformations are slightly less coherent. These groups are rigid bodies and are efficiently handled by rigid body collision detection techniques. The remaining types of models are all deformable models. The deformable models group consists of models in which edge connectivity is preserved. The next group contains deformable models without edge connectivity. The random triangle movement group contains objects that deform similar to explosion simulations. The last group contains models where new geometry is created every frame. An example

that would fit into this group is the surface of a fluid simulation.

When object deformations are minimal, AABB hierarchies can be updated and efficient collision culling can be performed. Using hierarchy updates when leaf nodes of the AABB separate can lead to slow collision detection. Figure 3.9 is a graph for a simulation where two objects explode into each other. As these objects explode the AABB nodes stretch to bound primitives that are not in close proximity one to another. As the simulation progresses, collision detection grows exponentially. This is similar to the image in Figure 3.2. In the worst case, new geometry is constructed every frame. In this case there are two options: build a new hierarchy each frame or use a hierarchy-less technique such as GPU culling.

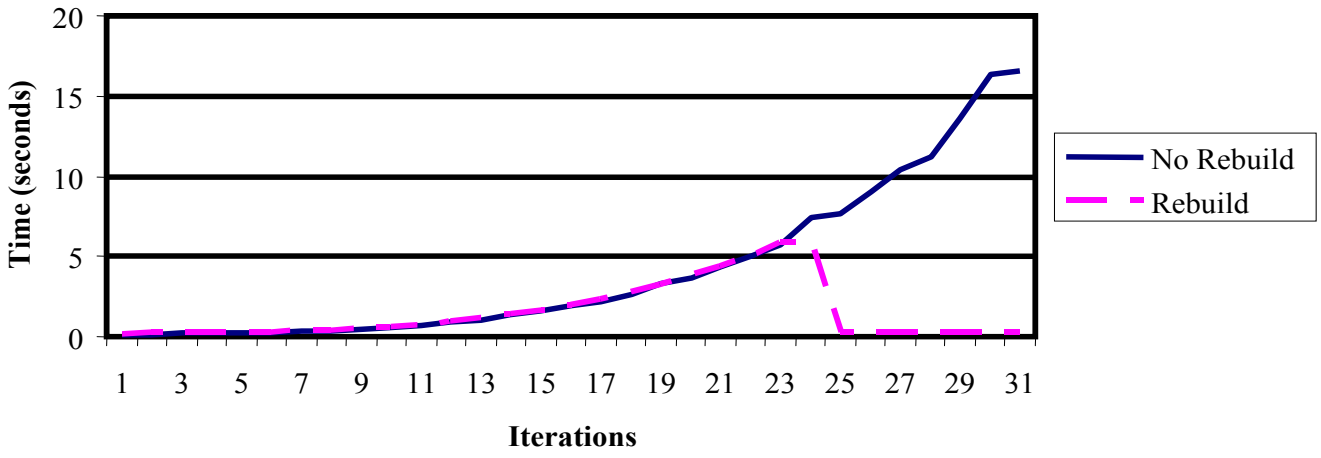

Figure 3.9: **Random Triangle Movement** *This graph represents the data for AABB based collision detection on a data set consisting of triangles moving in random directions. As the triangles at the root level separate, the run time rises exponentially. The plot represented by the dotted line shows the effects of rebuilding the hierarchy. On frame 24 the hierarchy is rebuilt and collision detection times become fast again. The rebuild time was not included in the plot.*

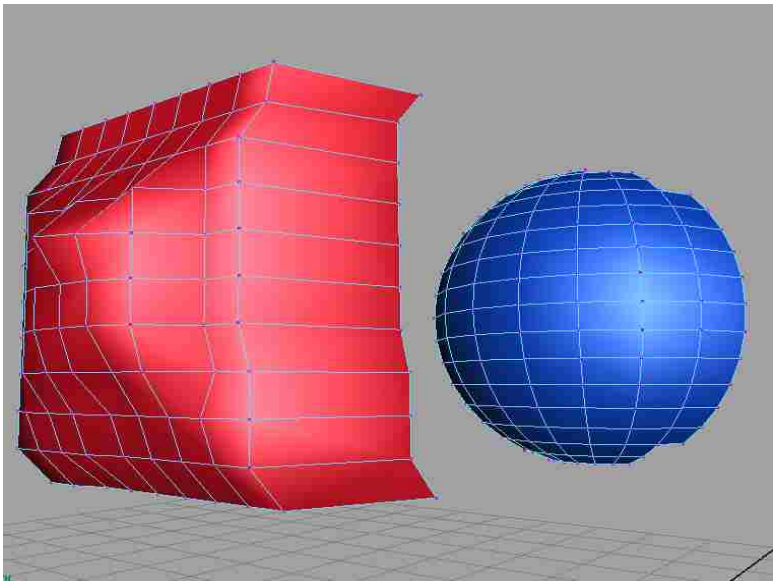

Figure 3.10: **Two Objects** *Two objects that are not in close proximity one to another.*

3.3.2 Geometric Proximity

We can describe both GPU culling and AABB-Hierarchy based collision detection as culling algorithms. It is obvious that the GPU culling algorithm removes triangles from the set of those that must be tested for collisions. AABB hierarchy based collision can also be thought of as a culling technique. Every time an AABB fails an intersection test with another AABB, all of the primitives and AABBs below that AABB are removed from the set of primitives that must be checked.

AABB based collision detection begins to suffer when objects are in close proximity. Figure 3.10 shows two objects that can be culled at the root level by AABBs. If these two objects are moved into very close proximity, the AABB culling is lessened and many triangle to triangle intersection tests must be performed. Figure 3.11 shows a plot of the number of intersection tests that must be performed as objects move into close proximity. In this figure the model has many overlapping triangles and collision detection becomes very difficult.

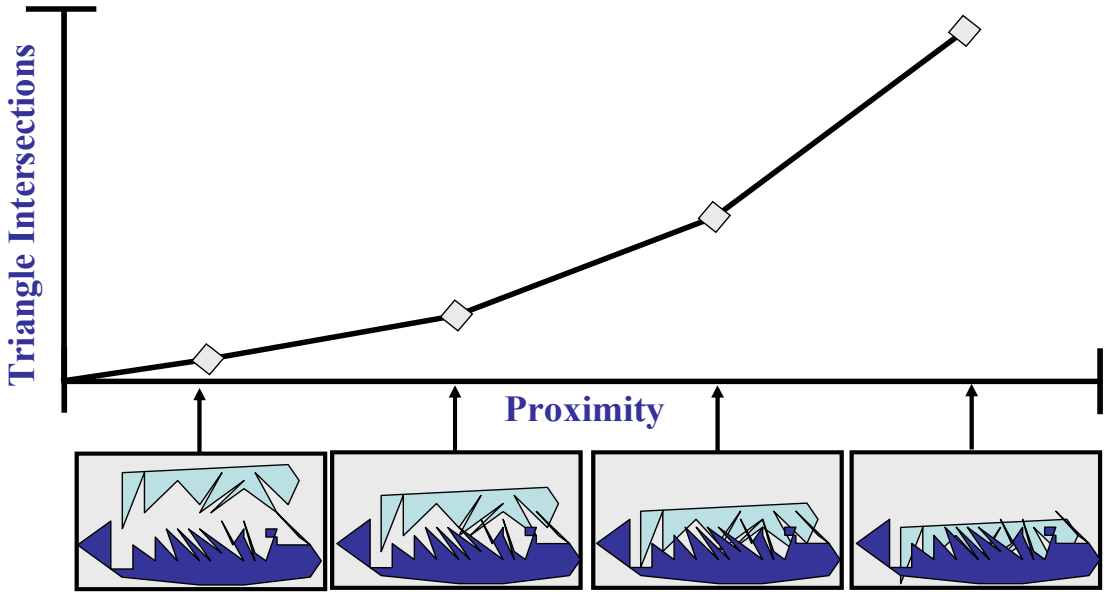

Figure 3.11: **Collision Detection Proximity** *As Objects come into closer proximity triangle triangle intersections increase.*

3.3.3 Deformable Model Collision Detection Bottlenecks

Table 3.1 outlines the two bottlenecks that limit AABB based collision detection. These bottlenecks are directly related to the problems discussed in Section 3.3.1 and Section 3.3.2. The primitive intersection bottle neck occurs when objects are in close proximity one to another. The AABB intersection bottleneck results from a poorly formed hierarchy. This can happen when AABBs are updated to fit around triangles and subtrees that are highly deformable.

Table 3.1: AABB Hierarchy Bottlenecks

1.	Primitive intersection bottle neck.
	a. Optimized GPU cull.
	b. hybrid AABB and GPU cull framework.
2.	AABB Intersection bottle neck.
	a. AABB rebuild in parallel with GPU cull.
	b. Hierarchy-less GPU cull.

Both of these bottlenecks have solutions. When primitive intersections are the bottleneck, it can be faster to perform the GPU cull and then rebuild a smaller hierarchy based on the non-culled subset of the geometry. Another option is to use a hybrid approach where AABBs are culled with the AABB hierarchy, but primitives are then culled with the GPU. When the bottleneck is the AABB tests, there are again two options. The first solution is to rebuild the hierarchy in a thread while performing GPU based culling in another. The second is to perform hierarchy-less collision detection by strictly using the GPU. These techniques are addressed in the following chapter.

Chapter 4

The Combined AABB and GPU Culling System

4.1 Overview

This chapter explains our combined collision detection system. We first discuss GPU culling and the three techniques we have developed to optimize the GPU cull. This is presented in Section 4.2. In Section 4.3, we explain our AABB based framework and perform an analysis on the needed conditions for it to be beneficial. Finally, in Section 4.4, we describe the combined AABB and GPU culling system and give an algorithm for selecting how and when to switch between them as an animation progresses.

4.2 GPU-based culling

Our GPU culling technique extends the work done in [2], [13], and [14]. Our work implements these ideas and adds to them metrics that dictate how to use GPU-based culling. In the following sections we show that GPU culling can be used in many different ways with different performance implications. After the GPU cull is finished, an AABB hierarchy must be built to perform the final collision detection. Because of this, the GPU-based culling technique is a tradeoff between the expensive AABB hierarchy rebuild of the culled set and the time to cull the set. Throughout this section we report the time to perform GPU culling as the summation of all the steps to perform collision detection. This consists of multiple cull passes plus the time to build a hierarchy and enact the final AABB collision detection. The goal of this part of the work is to optimize the GPU cull. We do this using three techniques.

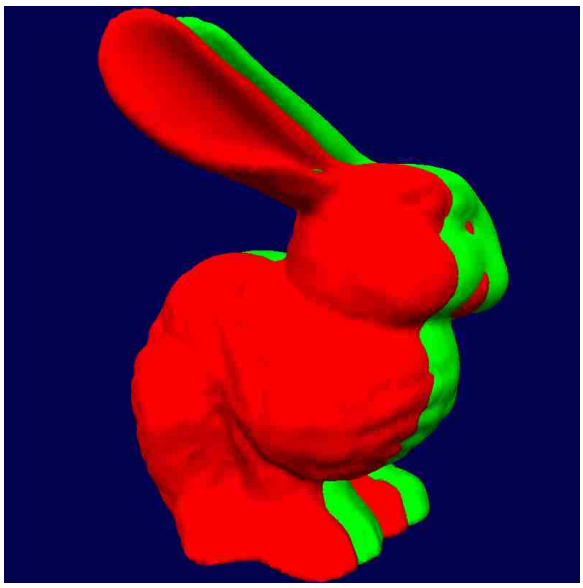

Figure 4.1: **Two Bunnies Colliding** *A visualization of a data set consisting of two bunnies overlapping.*

1. Determine the view configuration that minimizes the GPU cull time.

2. Sorting the objects to maximize the number of primitives culled.

3. Determining the proper number of triangles to include in each occlusion query.

The following sections discuss our approaches to these optimizations.

4.2.1 Determining The Proper View Configurations

GPU Culling is an $O(n)$ operation where n is the number of primitives in the scene. Choosing the best view for a given scene will increase culling and, as a consequence, decrease the number of primitives rendered on consequent cull passes. Rendering multiple cull passes can become very expensive when culling is not efficient. Figure 4.2 and Figure 4.4 show the total time needed to perform collision detection using different cull directions for two data sets. All combinations of axis-aligned view directions are iterated. Figure 4.2 corresponds to the screen shot in Figure 4.1. In this scene the depth complexity is very similar for all three views. In such situations

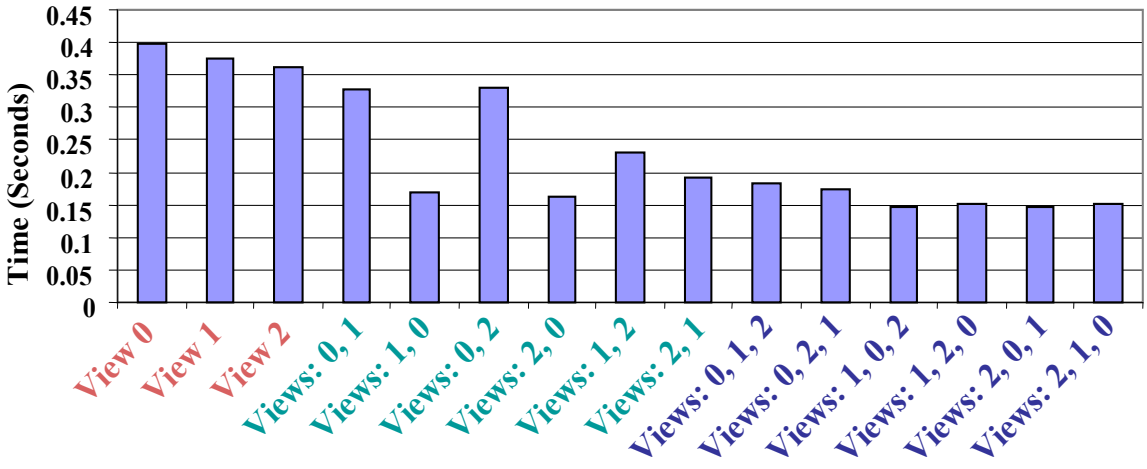

Figure 4.2: **Time For Different View Configurations** *Times given for different view configurations of GPU culling for the data set in Figure 4.1. The views are labelled 0, 1, and 2. These views are axis aligned corresponding to one of the X, Y, or Z axis, respectively.*

it is more important to render from all views than to find the best order. However, as the graph demonstrates, there is still an optimal ordering of the three view directions. Figure 4.4 gives the timings for various view configurations for the data set in Figure 4.3. This data set has a very high depth disparity in the X direction and in the Y direction. The Z direction has very low depth complexity and is, therefore, the optimal choice with respect to culling. The configurations that render view Z first are considerably faster than any others. We can see from Figure 4.2 and 4.4 that choosing the best view direction in many cases will result in a 10 to 100 percent increase in speed.

Our goal is to find the fastest configuration of views and to detect when different view configurations become faster. This requires a balance between exploration and coherency. If the amount of exploration is too low, our technique becomes stuck in less-optimal view configurations and more optimal view configurations are never found. With too much exploration our algorithm would tend towards an average of

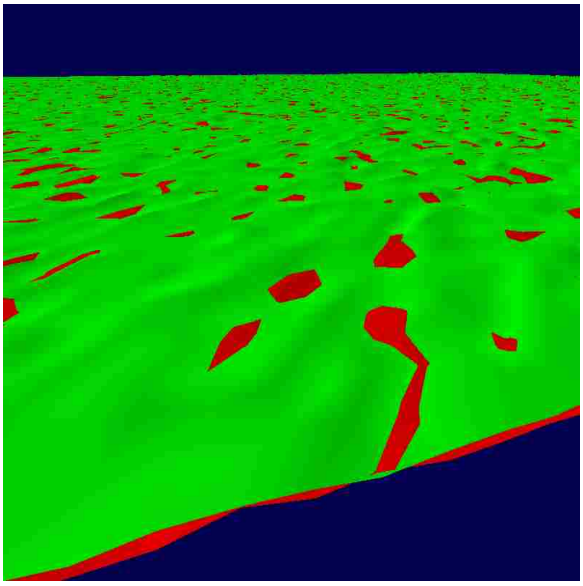

Figure 4.3: **Two Sheets in Close Proximity** *A visualization of a data set consisting of two sheets of cloth in close proximity.*

all configurations. In order to find the correct balance between exploration and coherency, our metrics take into account the percent culled per frame for the three view configurations and uses a decay principle to keep highly rated views, thus limiting exploration. Our technique explores one view configuration per frame. The decision consists of a few multiplies and compares, and is therefore irrelevant to overall time. Our view configuration technique is outlined in the following sections.

Percent Culled

Our primary indicator of a view direction's efficiency is the percent culled by that view. Using the percent culled as the primary indicator guarantees that our method will perform well when one view direction has a much lower depth disparity. In most cases the views that cull the best tend to result in the fastest overall frame time. This seems to be an obvious truth but there are some very rare situations where it does not hold. This is because of the many factors that go into the GPU cull. The orthographic projection changes every time the set of objects to be culled

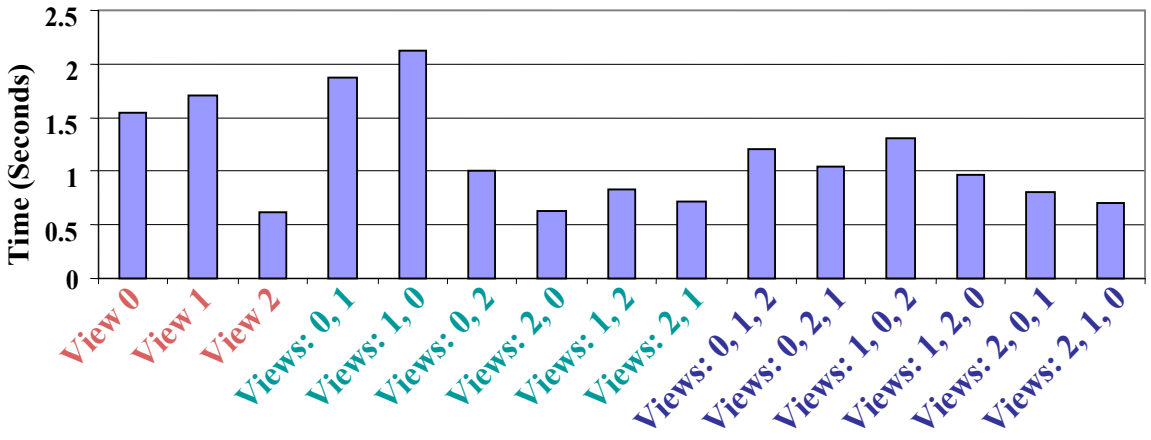

Figure 4.4: **Time For Different View Configurations.** *Times given for different view configurations of GPU culling for the data set in Figure 4.3. The views are labelled 0, 1, and 2. These views are axis aligned corresponding to the X, Y, or Z axis, respectively.*

changes. It is possible for one view to cull less, but cull the correct triangles that would cause increased resolution in following cull passes. It is also possible for a less efficient cull direction to cull triangles that allow for a much more efficient cull pass from subsequent cull passes. Because these cases are not common our technique does not directly optimize for them. Instead, our technique is biased towards view directions that have culled well on previous frames.

Decision Mechanism

In order to perform exploration on every cull pass, we store the percentage culled for every frame and for every cull pass. This can be represented as a matrix. The columns represent the cull passes, while the rows represent the view number. Our selection algorithm iterates over the matrix and picks the highest percent culled per view for each cull pass.

The view configuration matrix is a three by three matrix. The columns represent the X, Y, and Z view directions. The rows represent the cull passes. We hard

Algorithm 2 View Configuration Matrix Update

```
(1) The selected views for the 3 iterations are updated with Equation 4.1
(2) View options for cull pass 1 are increased by options for cull pass 2
and cull pass 3
(3) All view options for all iterations decay with Equation 4.2
```

code to 3 cull passes as this has shown to be a an efficient number. Algorithm 2 gives the three steps to update the view configuration matrix. We first update the views chosen for each of the cull passes.

$$view\_selected = \frac{view\_selected + update\_value * 3}{4} \tag{4.1}$$

The *update_value* is the percentage of the remaining triangles culled. If the set was completely culled on a previous iteration, then this value defaults to 0. This has a very heavy weight as the success of the previous frame is a very good indication of consequent frames. Next, we update the first row with the values of the second and third rows. In many cases the first cull pass becomes less efficient while the second and third cull passes become more efficient. Step (2) is an exploration term. When a cull pass is successful on the second or third pass, Step (2) will update the values for Cull Pass 1. Finally, each value in the matrix is multiplied by the $DECAY\_RATE$ variable.

$$view = view * DECAY\_RATE \tag{4.2}$$

Causing the variables to decay helps the efficiency of both Steps (1) and (2). As the animation progresses, the cull efficiency of a view tends to change. Causing the values to decay keeps the algorithm from exploring views that are no longer guaranteed to be efficient.

4.2.2 Object Sort

GPU culling is a rendering-based algorithm. As such, shortcuts are taken to limit the amount of rendering that must be performed. One optimization entails storing flags denoting the passes in which objects are visible. For the details of

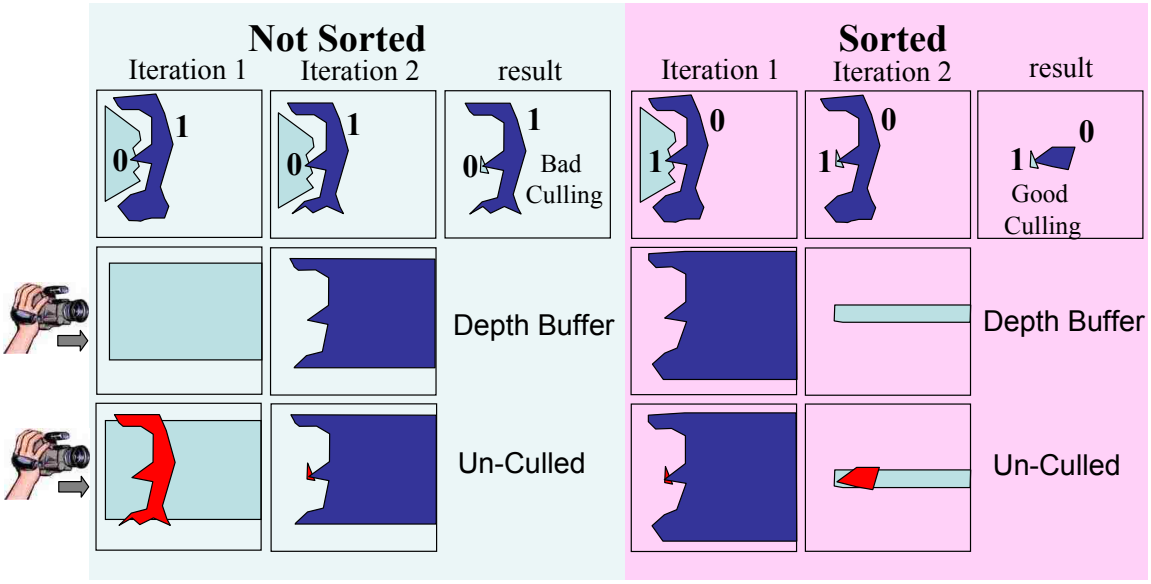

Figure 4.5: **Effect of Sorting Objects before the GPU Cull** *In the left side of the figure an unsorted set of objects is culled. On the right side of the figure, the same set of objects is sorted by the distance from the camera. We show the two iterations of a GPU cull. First, object* 1 *is compared against the depth buffer containing object* 0*. Next, object* 1 *is culled against the portion of object* 0 *that remains un-culled.*

this algorithm, see Quick-Cullide [14]. Without this optimization, objects must be rendered four times per GPU Cull. However, the optimization from Quick-Cullide does not work flawlessly. Figure 4.5 shows an example where the Quick-Cullide optimization is almost ineffective. In this case, the first iteration does not cull well, as shown on the left side of Figure 4.5. When the order of the objects is reversed the cull is much more successful. In this example, two objects are shown. However, this same problem occurs with any number of unsorted objects.

In our experience, sorting the objects along the camera's view direction tends to give good results. As shown on the right side of Figure 4.5, we are able to obtain significant speedups.

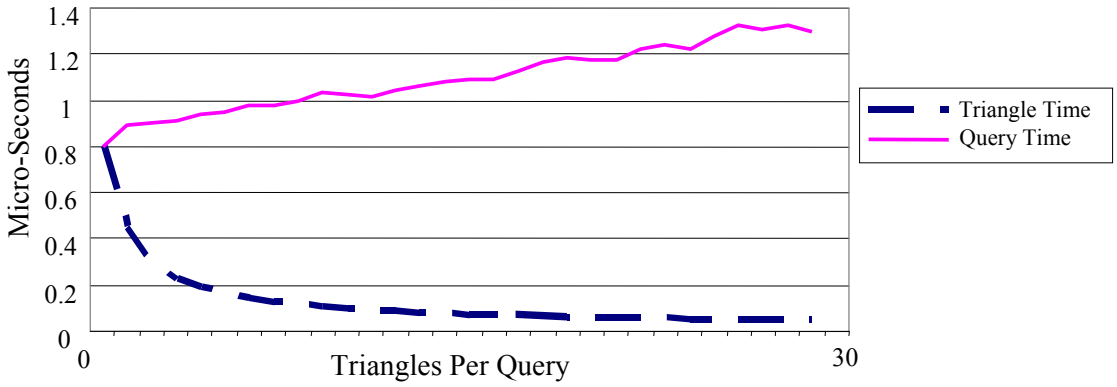

Figure 4.6: **Times to Perform Occlusion Queries** *This graph shows the time to perform occlusion queries when multiple triangles are rendered per query. Graph was plotted with an NVIDIA 6800 graphics cards using the 78.01 driver version. Triangles are rendered as vertex buffer objects to maintain highest rendering throughput.*

4.2.3 Triangles Per Query Determination

We have benchmarked occlusion query performance over a wide range of graphics cards. Our findings show that occlusion query capabilities have increased very little over the past few years. Current cards can perform about 1.3 million occlusion queries per second. Occlusion queries were first implemented in commodity hardware with the NVIDIA Geforce fx series. As newer cards have been introduced, occlusion query performance has changed very little. Figure 4.6 shows the plots of time taken per occlusion query as well as per triangle. The plot representing time per query changes very little as the geometry is increased. The time per triangle decreases exponentially. This shows that a naive implementation of one occlusion query per triangle is occlusion query bound. In some cases this is still the best course of action becuase increasing the geometry rendered per occlusion query yields lower culling. However, as Figure 4.6 shows, significant speedup can be obtained by rendering more than one triangle per occlusion query.

As with our view determination mechanism, our triangles per query mechanism relies on exploration and coherence. We store a predicted cull amount for each level

of triangles per query. After each cull the predicted cull amount is updated based on the culling results.

$$PX_{cnt} = \frac{PX_{cnt} + X_{cnt} * 3}{4} \tag{4.3}$$

is the update equation. Table 4.1 defines the variables used in this section.

Triangles per query cannot simply be updated based on efficiency in culling. The most effective GPU cull is almost always based on one triangle per query. However, using more triangles per query is faster and can sometimes make up for a less efficient cull. In other words, there is a tradeoff between the amount culled and the time taken to cull. In

$$N = \frac{XO}{P} + UR \tag{4.4}$$

we quantize the total time taken as the time per occlusion query multiplied by the number of queries divided by the number of triangles per query plus the time to rebuild the hierarchy as a function of un-culled triangles. The AABB collision detection test is very fast after the hierarchy is built, making it negligible. We also define the culled triangles and un-culled triangles in terms of each other in

$$T = X + U \tag{4.5}$$

Solving for for U gives us

$$U = T - X \tag{4.6}$$

We then substitute this into Equation 4.4 resulting in

$$N = \frac{XO}{P} + RT - RX \tag{4.7}$$

We can solve for the number of primitives that must be culled to justify P.

$$X = \frac{N - RT}{\frac{O}{P} - R} \tag{4.8}$$

After we have determined how many triangles must be culled for both an increased and decreased number of triangles per query, we can update our indicator variables:

$$X\_inc\_fit = PX_{inc} - X_{inc} \tag{4.9}$$

Table 4.1: Triangles per Occlusion Query Variables

Variable Name	Definition
X	The number of primitives that must be culled to justify P.
U	The number of primitives that will be left un-culled.
T	The total number of primitives.
O	The cost of an occlusion query.
P	The number of triangles rendered per occlusion query.
R	The equation representing the rebuild cost.
N	Calculated cost of occlusion queries plus rebuild
PX	Estimated culling per triangle count.
X_inc_fit	Indicator of performance when triangles per query are increased
X_dec_fit	Indicator of performance when triangles per query are decreased

Algorithm 3 Steps to Calculate Triangles per Occlusion Query

(1) Update predicted X for selected P with Equation 4.3.
(2) X_{inc} is calculated from Equation 4.8 by substituting $P+1$ for P.
(3) X_{dec} is calculated from Equation 4.8 by substituting $P-1$ for P.
(4) X_inc_fit is calculated with Equation 4.9.
(5) X_dec_fit is calculated with Equation 4.10.
(6) Increase P if $X\_inc\_fit > X\_dec\_fit$ and $X\_inc\_fit > 0$.
(7) Decrease P if $X\_dec\_fit > X\_inc\_fit$ and $X\_dec\_fit > 0$.

and

$$X\_dec\_fit = PX_{dec} - X_{dec} \tag{4.10}$$

perform this update. These two variables indicate how many more triangles will be culled by increasing or decreasing the triangles per query. Algorithm 3 shows the complete steps our triangle per query mechanism takes.

4.3 A Framework built on GPU and AABB Culling

Figure 4.7 outlines the steps to combine GPU-based culling and AABB-based culling. The general idea is to perform the AABB cull down to the triangle level but skip the triangle intersection tests. When a triangle intersection is reached, the triangle is marked to be tested later with the GPU cull. In the next phase the marked triangles are culled with the GPU. These results are then percolated up the tree and final collision detection takes places using the culled tree.

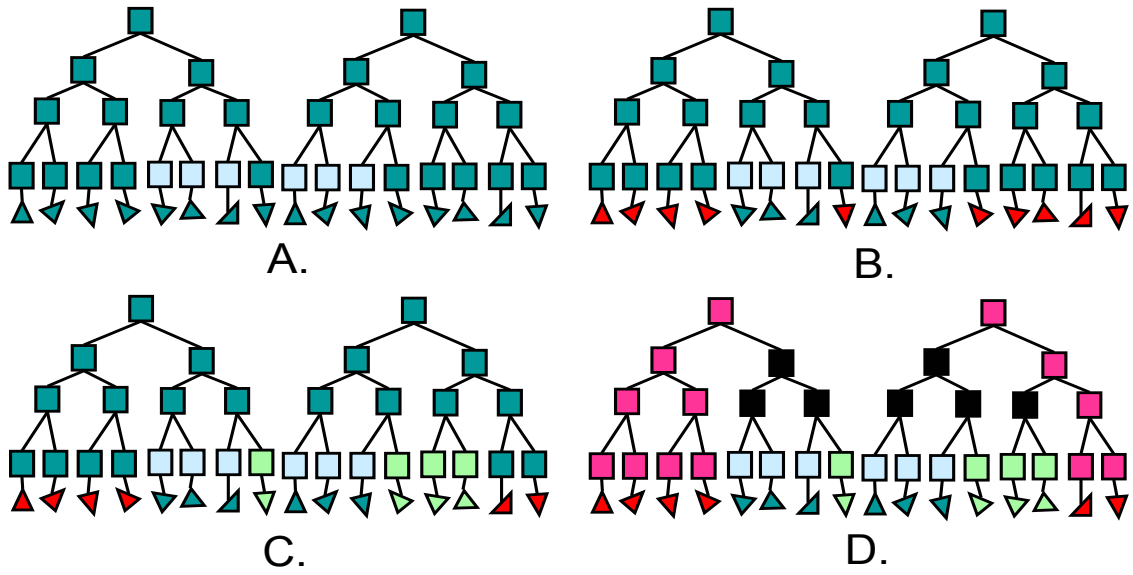

Figure 4.7: **AABB GPU-Cull Framework** *A. AABBs are initially used to cull down to the Leaf nodes. B. Un-culled triangles are then culled with the GPU. C. The results of both culls are then percolated back up the tree. D. Final collision then takes place using the marked tree.*

This technique is useful when triangle intersections become the bottleneck. This can happen in two ways:

1. Objects deform in close proximity one to another with the result being that many triangles are in very close proximity.

2. Triangle-Triangle intersections speeds are sufficiently slow.

Case 1 can be restated as a problem of run-time complexity. When triangles are distributed uniformly, a triangle will only be tested against the following: $N \log N$ AABBs, 1 triangle, and possibly a few additional triangles that share edges with the colliding triangles. When triangles deform by bunching up or overlapping, the number of triangle intersections becomes much larger. In case 2 above, triangle-triangle intersection speeds may vary significantly from one architecture to another. Also, the triangle-triangle intersection test given by Moller [3] is not constant. The

Table 4.2: AABB and GPU Cull Framework

Variable Name	Definition
A_1	Number of Initial AABB to AABB collisions.
A_2	Number of Final AABB to AABB collisions.
TRI_1	Number of Triangle to triangle collisions after AABB cull.
TRI_R	Number of Triangles to render for GPU cull after AABB cull.
TRI_2	Number of Triangle to triangle collisions after AABB cull and GPU cull.
A_I	AABB intersection time.
T_I	Triangle intersection time.
P_C	Percent of triangles culled.
$Pure\_AABB$	Time taken to perform AABB collision detection.
$Hybrid$	Time to perform our hybrid approach.

test performs many calculations with increasing complexity to determine if triangles do not intersect. This causes a large variance in the time per triangle test.

4.3.1 Framework Analysis

In this section we perform analyses of our framework in order to determine circumstances where our framework is more efficient than a purely AABB based approach to collision detection. The variables used in this section are defined in Table 4.2.

$$Pure\_AABB = A_1 * A_I + TRI_1 * T_I \tag{4.11}$$

breaks the time to perform AABB based collision up into the AABB intersection cost and the triangle intersection cost.

$$Hybrid = A_1 * A_I + TRI_R * O + A_2 * A_I + TRI_2 * T_I \tag{4.12}$$

gives the time to perform our hybrid method. TRI_1 and TRI_R represent the number of triangles that must be checked for collision detection on the CPU and GPU respectively. Both methods have the same AABB test denoted by $A_1 * A_I$. We set the previous two equations equal to each other in

$$TRI_1 * T_I = TRI_R * O + A_2 * A_I + TRI_2 * T_I \tag{4.13}$$

Table 4.3: Collision Detection Times

Type	Time
Triangle Intersection Test	100 nanoseconds
Occlusion Query	1000 nanoseconds

This tells us that the GPU cull plus the un-culled triangle intersections plus the final AABB test must be smaller than the triangle tests in order for the hybrid method to be faster. In order to get an upper bound, we drop the final AABB test and final CPU triangle tests giving us

$$TRI_1 * T_I > TRI_R * O \tag{4.14}$$

In

$$TRI_1 = O/T_I \tag{4.15}$$

we set TRI_R equal to 1, and solve for TRI_1. This equation tells us that the ratio of O/T_I is the number of intersections that the CPU-only test would have had to have performed in order for our hybrid approach to be as fast as a pure CPU approach. Table 4.3 gives the times for primitive intersections. These timings were benchmarked on a 1.6Ghz Intel Centrino. These times represent the average time over millions of tests to ensure accuracy. With an occlusion query time of 1 microsecond and a triangle intersection test time of 100 nanoseconds there must be 10 times more intersections than GPU culls. If the GPU is not able to cull all triangles then we must look at the number of intersections required as a function of percentage culled. We start with Equation 4.13. We represent TRI_2 as a percentage of TRI_1. P_C is a value between 0 and 1 and represents the percentage of the triangles that were not culled by the GPU. We drop the final AABB test term A_2 in order to show a lower bound. An alternative would be to bound A_2 in terms of $tri_1 * P_C$. However the correct equation to do this would be dependant upon the models of the scene and their respective configurations. This gives us

$$TRI_1 * T_I > TRI_R * O + P_C * TRI_1 * T_I \tag{4.16}$$

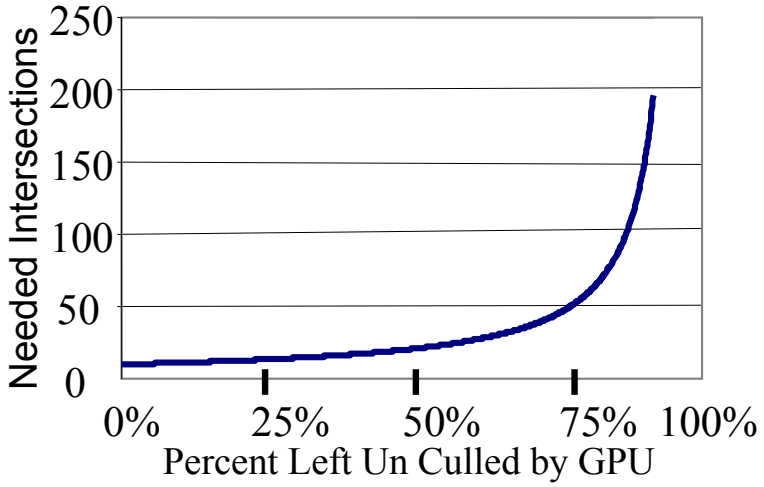

Figure 4.8: **Triangle Intersections Needed** *Triangle intersections needed as a function of percentage culled by GPU. Equation 4.17 is evaluated as* P_C *goes from* 0 *percent to* 100 *percent.*

Next, we set TRI_R to 1 and solve for TRI_1. Our result is

$$TRI_1 = \frac{O}{T_I - P_C * T_I} \tag{4.17}$$

This is still a lower bound as the occlusion query cost would tend to be higher. GPU culling can require as many as two occlusion queries per Triangle. In Figure 4.8 we iterate from all culled to none culled. The graph shows the number of triangle intersections that must be culled in order for GPU culling to be faster. As we approach 100 percent un-culled, the number of triangle intersections needed becomes infinite.

In practice we found a few examples where occlusion queries are fast enough to use our hybrid method. When objects begin deforming and overlapping enough to create many collisions per triangle, the GPU culling tends to be less efficient. Equation 4.17 shows that decreasing the time taken to perform occlusion queries would decrease the number of triangle intersections left un-culled by the AABB cull pass. For architectures where occlusion queries are as fast or faster than the average triangle triangle intersection code, this system would be a win. The other option is to use this only where there are an order of magnitude more triangle - triangle

Table 4.4: Decision System Variables

Variable Name	Definition
$AABB_T$	AABB Collision Detection Time.
GPU_T	GPU Collision Detection Time.
$AABB\_Rebuild\_Average$	Average cost of AABB rebuilds over time.

Table 4.5: Decision System Functions

Function	Description
$Perform\_AABB$	Perform AABB based collision detection.
$Perform\_GPU_1$	Perform GPU based collision detection with out metrics.
$Perform\_GPU_2$	Perform GPU based collision detection with metrics.
$Mode1$	Switch to Mode 1.
$Mode2$	Switch to Mode 2.
$Rebuild\_In\_Tread$	Rebuild AABB hierarchy in a thread.
$explore$	Return probability of Mode 2.
$inc\_explore$	Increase explore likelihood.
$dec\_explore$	Decrease explore likelihood.

intersections than triangles. However, this is an unlikely situation.

4.4 Deciding Which Culling to Use

This section outlines how we determine the type of culling to be used. We describe two different culling modes, labelled Mode 1 and Mode 2. Mode 1 is primarily an AABB-based technique that uses the GPU during rebuilds. Mode 2 is GPU-based culling. In cases where the models preserve edge connectivity, and are not in close proximity, Mode 1 is the fastest technique. When objects become extremely deformable or in very close proximities, Mode 2 is fastest. We explain the two different modes, Mode 1 and Mode 2, and then the decision algorithm in the next sections. Tables 4.4 and 4.5 describe the variables and functions used.

The GPU AABB combined framework could one day be used in place of Mode 1. This technique has shown to be a very good way of removing the triangles that are not in close proximity with AABBs, and then culling more of the triangle in close proximity with the GPU.

4.4.1 Mode 1

Algorithm 4 Algorithm for Mode 1.

```
AABB_T = Perform_AABB(); // (1) Sample the AABB time.
GPU_T = Perform_GPU_1(); // (2) Sample the GPU time.
while(animation); // (3) Iterate through animation sequence
   if (AABB_T < GPU_T); // (4) AABB time is faster perform AABB based
     Perform_AABB();
   else if (AABB_T > GPU_T); // (5) GPU cull time is faster rebuild and test.
     Rebuild_In_Tread() AND Perform_GPU_1();
     AABB_T = Perform_AABB();
     GPU_T = Perform_GPU_1();
     if (AABB_T > GPU_T); // (6) If AABB is still faster than GPU move to mode 2
       Mode2()
     if (AABB_Rebuild_Average > GPU_T); // (7) If AABB average build time is
greater than GPU time, move to mode 2
       Mode2()
     if (explore()); // (8) use explore function to test Mode 2
       Mode2()
```

Mode 1 is primarily a CPU-based approach. Algorithm 4 gives the algorithm for Mode 1. We store the time taken to perform AABB based collision detection in $AABB_T$ and the time taken to perform GPU-based, hierarchy-less collision detection in GPU_T. When GPU_T becomes larger than $AABB_T$, we rebuild the AABB hierarchy in a thread and perform GPU based collision detection. After the AABB rebuild thread has finished, we again time the AABB based collision detection and GPU based culling. If GPU based culling is still faster, we switch to Mode 2. We also have a probability based explore function that switches to Mode 2. The rate of exploration is updated in Mode 2.

Our improved GPU cull is not used in Mode 1. The improved GPU cull is only useful when it is called enough to offset the cost of exploration.

4.4.2 Mode 2

Mode 2 is the improved GPU Cull outlined in Section 4.2. Algorithm 5 gives the steps performed in Mode 2. As mentioned earlier Mode 2 sets the exploration probability variable. The exploration variable is only used by Mode 1 to switch to Mode 2. When the GPU collision detection becomes slower than AABB based, we decrease exploration probability and return to Mode 1.

Algorithm 5 Algorithm for Mode 2.

```
while(animation); // (1) Iterate through animation sequence
   inc_explore ; // (2) Increase exploration.
   GPU_T = Perform_GPU_2() ; // (3) Perform GPU based collision detection.
   if (AABB_T < GPU_T ); // (4) If AABB time is faster, perform AABB based
     dec_explore();// (5) Decrease exploration at rate of GPU_T / AABB_T
     Mode1();
```

4.4.3 System

Our System starts in Mode 1 where both AABB-based and GPU-based collision detection are timed. For slightly deformable models, rigid bodies, and models that are not in close proximity, the AABB-based timing will be faster. In such cases our system will continue using Mode 1 unless the probability-based explore function switches to Mode 2. When Mode 2 is consistently slower than Mode 1 the explore function is decreased. For those models where the GPU is faster, Mode 2 is used until the AABB time is faster than the time for GPU based culling. Because the exploration is initially high, our GPU cull will have a few frames to try different view configurations, and triangles per query. Allowing Mode 2 to return to Mode 1 helps on benchmarks like the BART benchmark where the animation is initially incoherent but becomes more coherent.

In the following chapter, we show results for hierarchy-less collision detection and our system made up of Mode 1 and Mode 2.

Chapter 5

Results

The collision-detection system obtains good results on a wide range of models. Figure 5.3 shows the 5 data sets used to test our system. We refer to these data sets as A, B, C, D, and E. Data set A is an animation of two bunnies rotating. The bunnies are both rigid bodies, with affine transformations performed on them. Data set B is an animation of two pieces of cloth waving, deforming, and rotating. Data set C is an animation of two sheets deformed by sine waves. The animation rotates the two sheets of cloth. Data set D is the BART animation from Figure 2.4. Data set E is an animation of a dragon exploding. We use data sets A, B, and C in Section 5.1 for our timings on hierarchy-less collision detection. These three data sets show the effects of our three GPU cull optimizations. All data sets are used in Section 5.2 to give results for our collision detection system.

5.1 Hierarchy-Less Collision Detection

In the next three sections we present the results of sorting, picking the optimal view configurations and determining the optimal number of triangles per query. We used the data sets A, B, and C. Figure 5.2, Figure 5.3, and Figure 5.4 follow the same format. On the left side of the figure are three graphs showing the time taken for collision detection for each data set (in seconds). Bar graphs on the right of the figures represent the total cost of the methods for all three data sets. The horizontal axis represents time. All times are given in seconds.

5.1.1 Object Sort

Figure 5.2 shows the results for sorting the objects before rendering them. On data set A, the sort is only slightly faster. In some cases the sort can even result in a slow down. Sorting at the primitive level would guarantee an optimal render order, however, this would be expensive. Set B and C show a 10 and 25 percent increase respectively. These data sets have a low depth complexity for one view configuration. This makes sorting very important.

5.1.2 View Configurations

As a baseline, we time three view configurations. The goal of our view configurations mechanism is to be as fast or faster than any other view configuration. Figure 5.3 shows that our technique is nearly as fast or faster than the best hard coded view configuration on all three data sets. For data set A, the view configurations do little to impact performance. However, there is still a six percent difference between different view configurations. On this data set our technique averages out to be about as fast as the *View Z,X,Y*. On data set B *View Z,X,Y* is the best. Therefore, our technique explores until it finds *View Z,X,Y* and then continues using it. Data set C shows our algorithms working optimally. As the animation rotates, different view configurations become optimal. Our technique moves between these views for a speedup over all other hard coded views. Finding the correct view in this case can amount to a three times speedup.

When our sorting technique is coupled with our view configuration mechanism, the two techniques mutually help one another. The sorting mechanism benefits because the optimal view direction is less prone to noise. The view configuration mechanism benefits because good views are rated higher.

5.1.3 Triangles Per Occlusion Query

Figure 5.4 gives our results for the triangles per query mechanism. As mentioned earlier, GPU based culling can benefit greatly from rendering multiple triangles per occlusion query. For data set A, the optimal number stays between two and three.

Our technique moves between two and three and then eventually settles on two. Data set B has so many collisions that one triangle per query is optimal. Our technique finds this and continues with one triangle per query. For data set C, our technique explores up to 15 triangles per query. However, to make the graph readable only 1, 2, and 3 triangles per query are shown.

5.1.4 Final Results

Figure 5.5 gives our final results for hierarchy-less collision detection. The different data sets each benefit from different combinations of our three mechanisms. Data set A benefits most from the sorting and the triangles per query. Data set B benefits from view selection and sorting. Data set C benefits from all three techniques. Performance increase varies from 10 percent faster to an order of magnitude faster.

Data set C exemplifies the strengths of our improved GPU cull. The naive GPU cull consisting of randomly choosing a view configuration, using one triangle per occlusion query, and not sorting, is slower than AABB based collision detection. With our approach, data set C shows a large speedup. The triangles in this data set are in close enough proximity to require many triangle-triangle tests with the AABB method. With the correct views, triangles per query, and sort, the GPU based approach is able to cull many triangle-triangle intersections.

5.2 Combined GPU and AABB System

Our system couples our GPU collision detection with AABBs. In order to be successful, our technique must sample between Mode 1 and Mode 2 frequently enough to change when one technique becomes faster. However, over-sampling tends towards an average of the two modes. Mode 1 performs optimally on slightly deforming models and rigid body collision detection. Mode 2 performs best on objects that are either in very close proximity one to another, or in which constant rebuilding becomes slower than performing hierarchy-less collision detection.

5.2.1 Data Set Results

In the following sections we give results and explanations for our method, AABBs, and a naive GPU only technique. Our AABB based technique rebuilds after the collision detection time becomes more expensive than three fourths of a rebuild. This is not always the fastest approach to rebuilding, but determining when to perform a rebuild is an open-ended problem.

Data Set A

Data set A, represented by Figure 5.6, is a rigid body simulation. Throughout the simulation none of the objects come into close enough proximity to warrant the switch to Mode 2. This animation quickly finds Mode 1. The spikes in the graph represent unsuccessful exploration into Mode 2.

Data Set B

Figure 5.7 represents two sheets deforming into each other. This is another technique where Mode 1 is fastest throughout. This is because of the large number of collisions. The triangles that the GPU is unable to cull must be built into a hierarchy and tested.

Data Set C

Figure 5.8 gives the results for data set C. For this data set, GPU-based culling is the fastest approach. During the first few iterations, the animation moves between Mode 1 and Mode 2. Once the appropriate views and triangles per query are found, Mode 2 is considerably faster. The spikes in the graph represent places where Mode 2 becomes slower. As the GPU cull mechanisms find new view configurations, the system becomes more confident in Mode 2.

Data Set D

In previous sections we mentioned the BART benchmark as a very difficult case to handle. Figure 2.4 shows the animations of data set D. During the first half

of the animation, the triangles move in a random fashion. During the last half, most of the edge connectivity is preserved. Figure 5.9 shows our results on this data set. During the first half of the animation, frames must be rebuilt often. The cost of rebuilds is still too low to warrant Mode 2. In the more coherent second half of the animation, the hierarchy does not need to be rebuilt as often.

Data Set E

Data set E is an example of an object deforming too fast for a hierarchy to be rebuilt. We created four scenarios for this object. The timing results of these four scenarios are shown in Figures 5.10 through 5.13. In each successive figure, the models deform more slowly. In the first scenario shown in Figure 5.10 Mode 2 is fastest. This is due to the expensive and frequent rebuild costs. In the next two scenarios, shown in Figures 5.11 and 5.12, Mode 2 is initially optimal, but as the sequence progress, Mode 1 becomes faster. In the final scenario, shown in Figure 5.13, Mode 1 is fastest throughout.

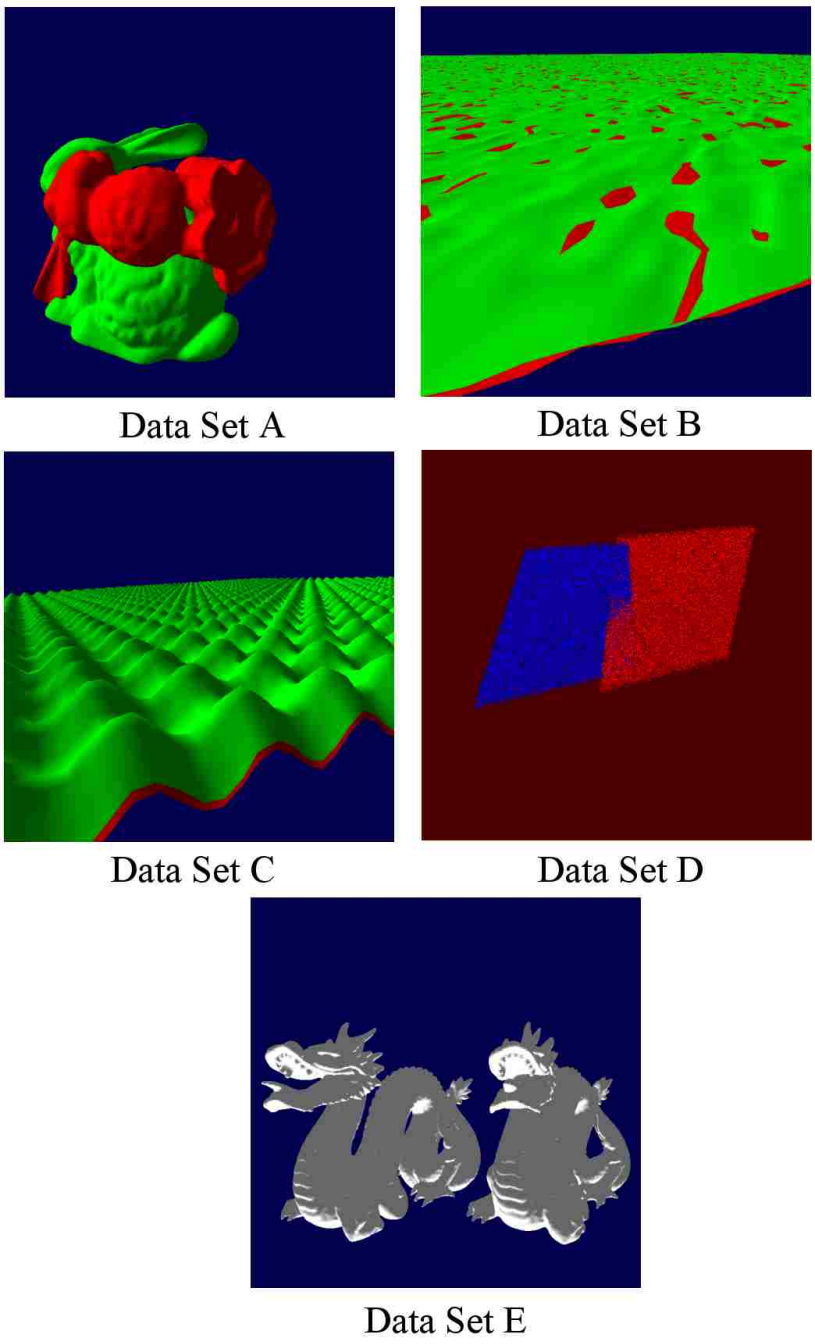

Figure 5.1: **Visualization of Data Sets** *This figure is a rendering of the data sets used.*

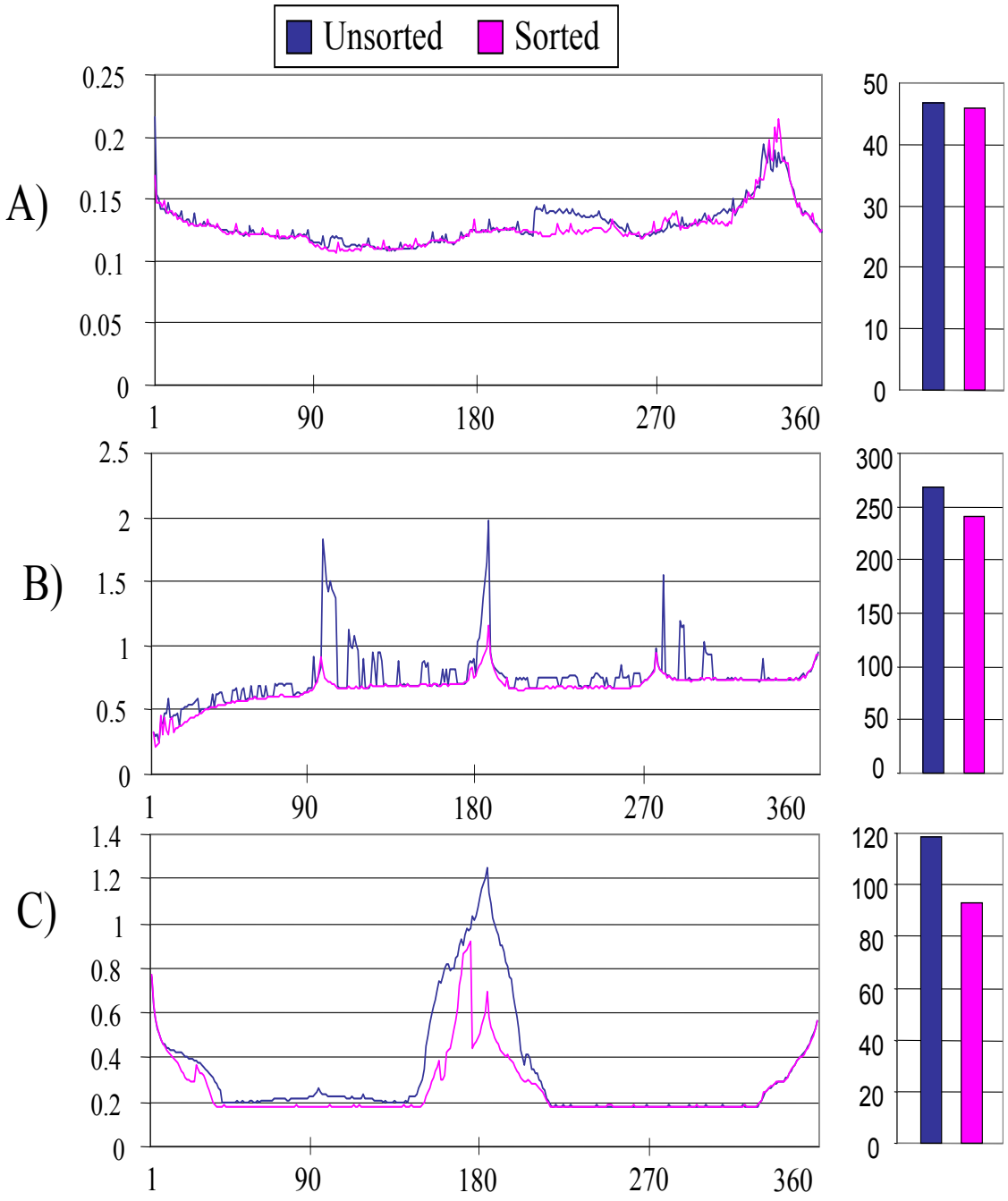

Figure 5.2: **Results for Sorting Objects** *This graph shows sorted and unsorted results. This graph shows that sorting is almost always faster.*

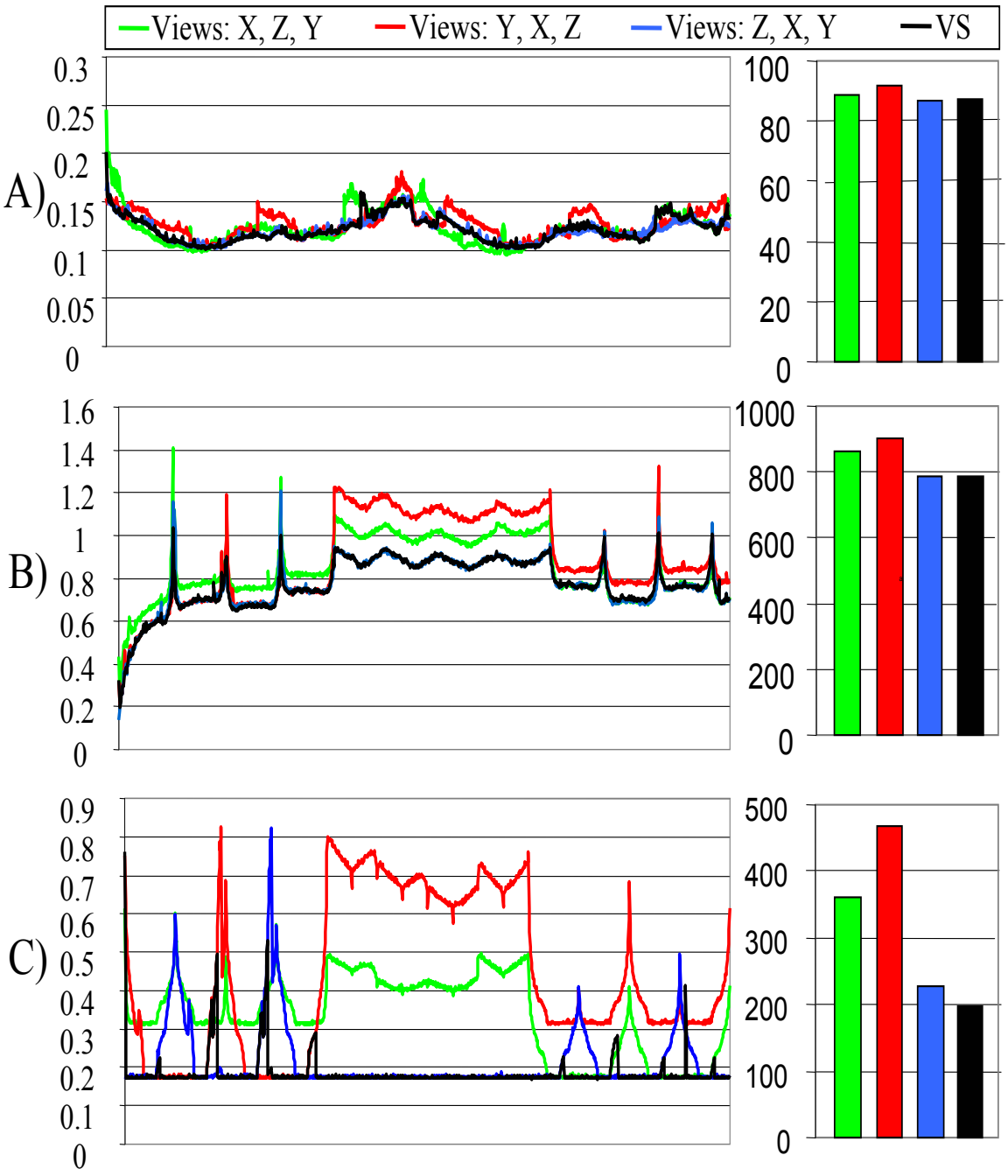

Figure 5.3: **View Configuration Results** *This graph shows 4 different view configurations. the first three are different combinations of the the axis aligned views X, Y, and Z. VS represent our view selection technique.*

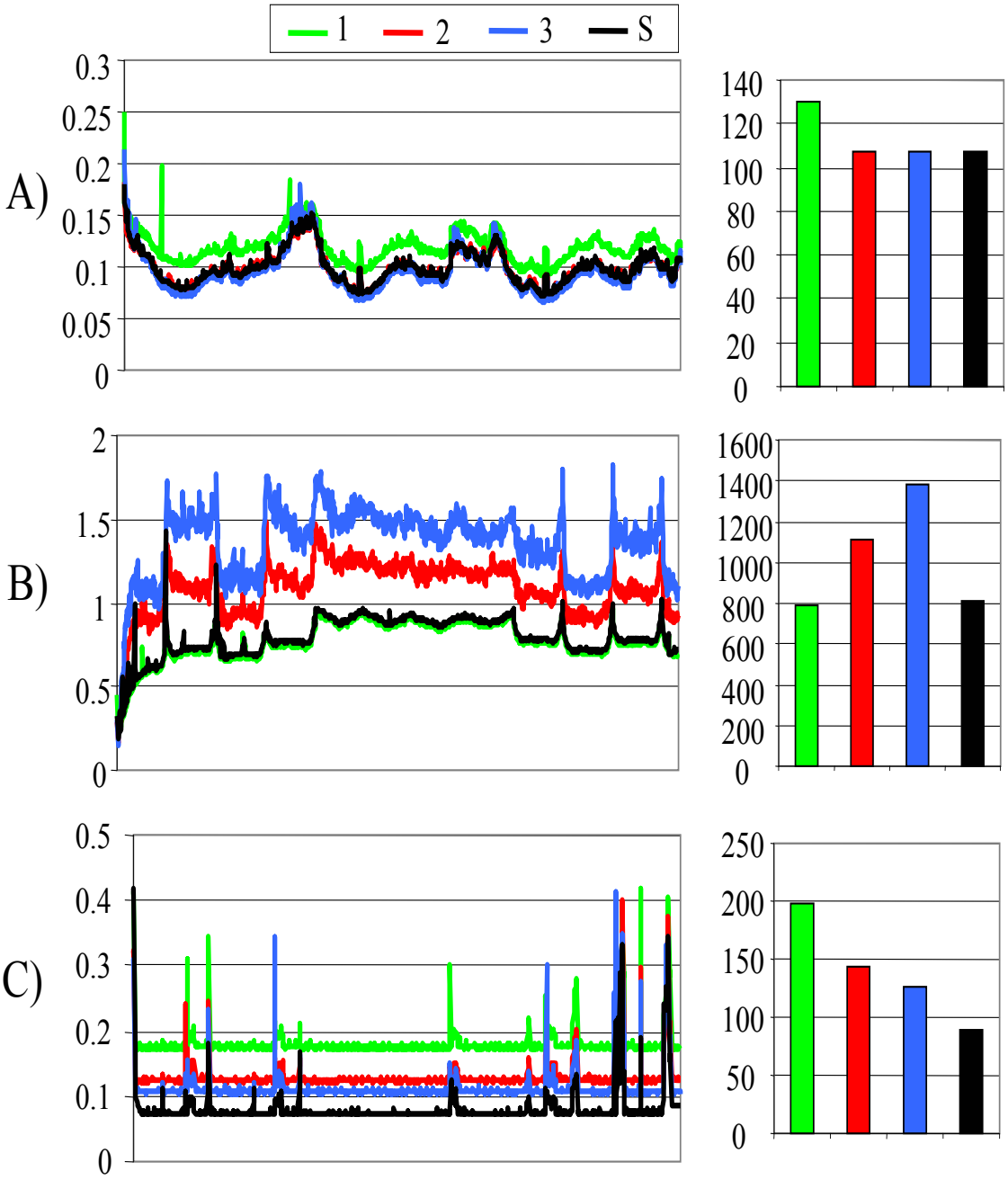

Figure 5.4: **Results of Triangles Per Occlusion Query** *This graph shows the overall collision detection time for 1, 2, and 3 triangles per query. Our technique is denoted with the letter S.*

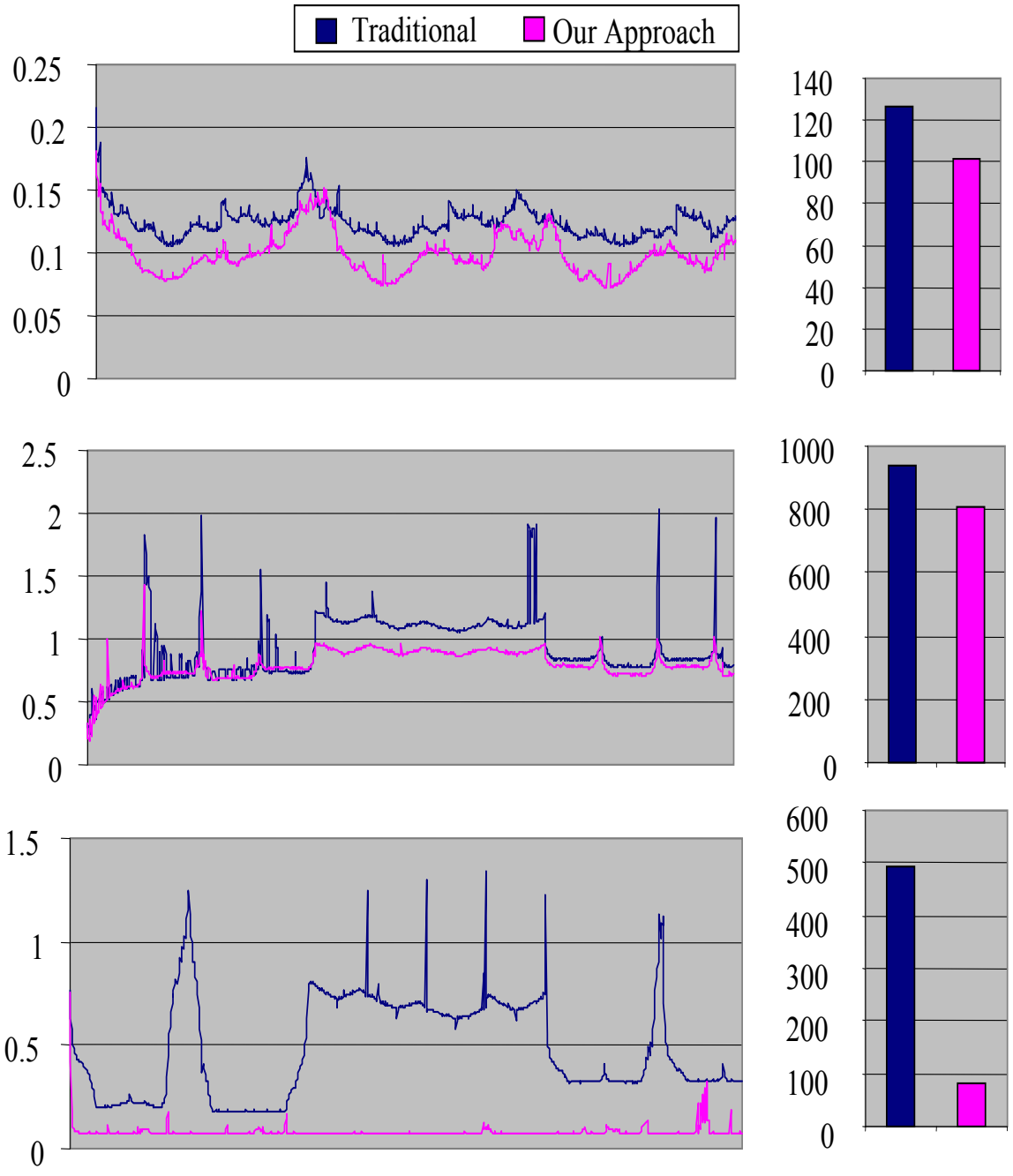

Figure 5.5: **Final Hierarchy-less Results** *This graph shows our final results against a naive GPU cull.*

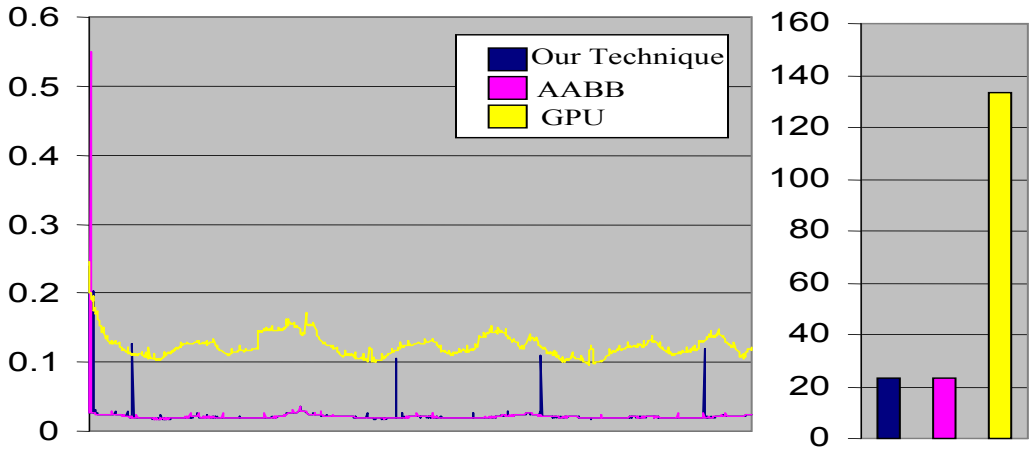

Figure 5.6: **Results for Data Set A, Two Bunnies**

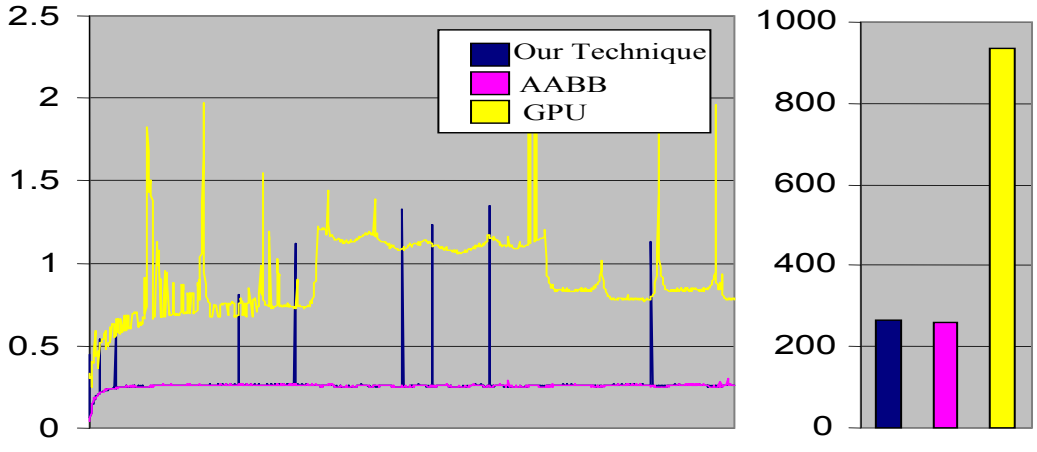

Figure 5.7: **Results for Data Set B, Two Waving Sheets**

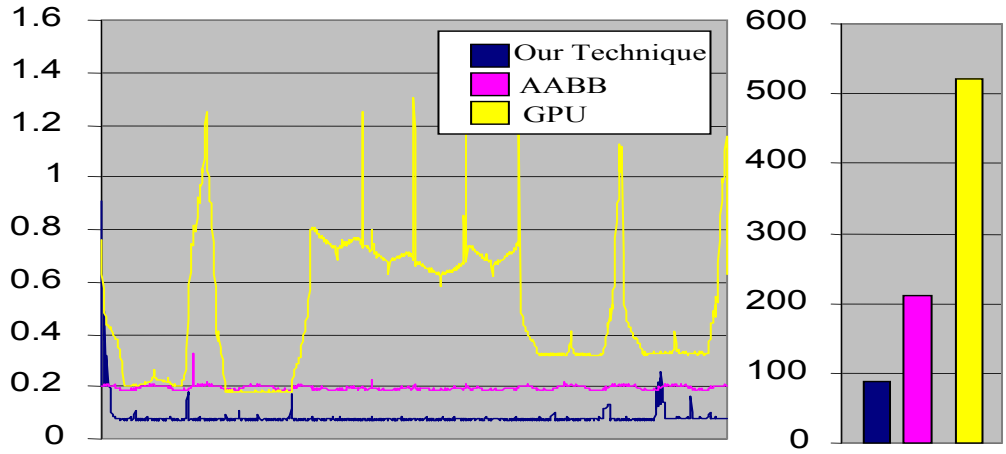

Figure 5.8: **Results for Data Set C, Sin Wave Sheets**

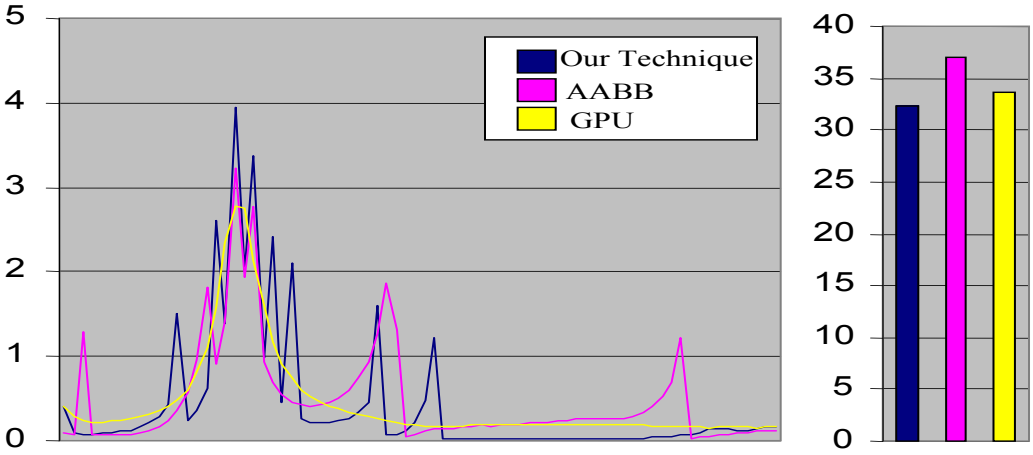

Figure 5.9: **Results for Data Set D, BART Animation**

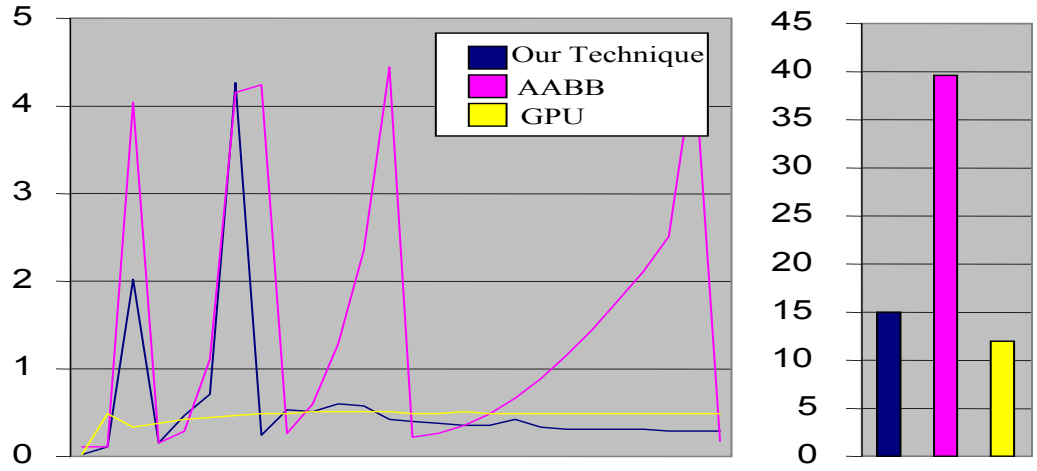

Figure 5.10: **Results for Data Set E, Exploding dragons, Over 25 Frames** *The animation is simulated with a step size of 8 over 25 frames.*

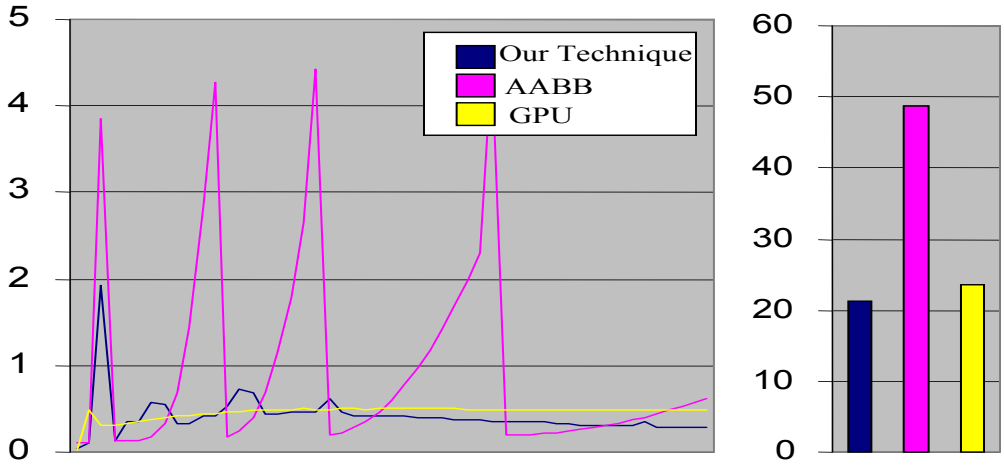

Figure 5.11: **Results for Data Set E, Exploding dragons, Over 50 Frames** *The animation is simulated with a step size of 4 over 50 frames.*

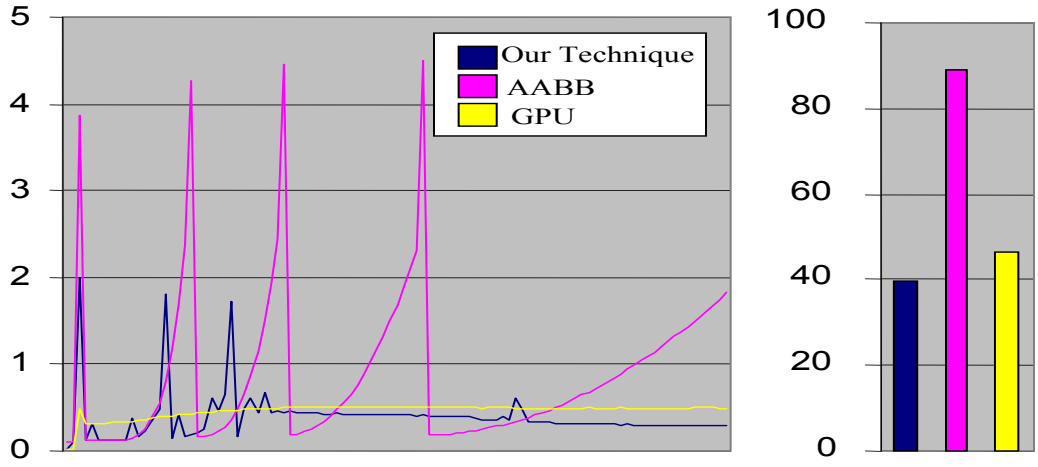

Figure 5.12: **Results for Data Set E, Exploding dragons, Over 100 Frames**
The animation is simulated with a step size of 2 over 100 frames.

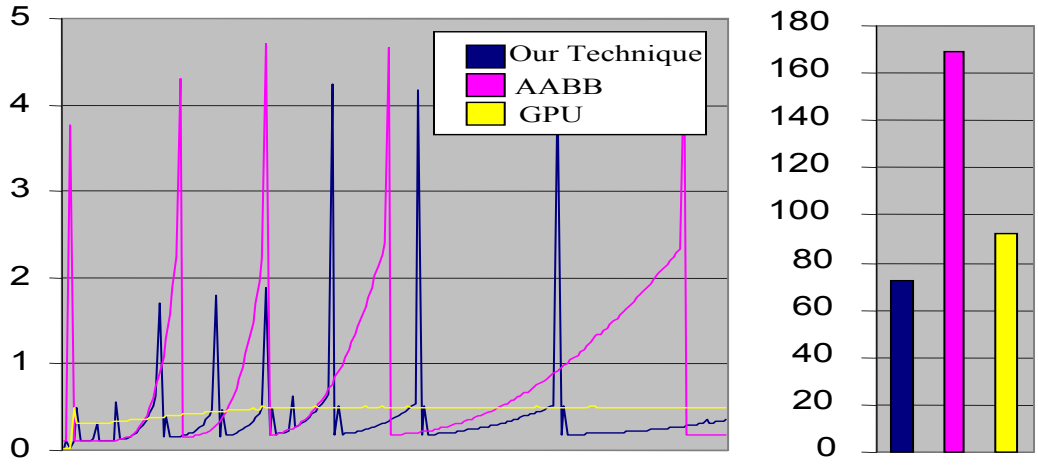

Figure 5.13: **Results for Data Set E, Exploding dragons, Over 200 Frames**
The animation is simulated with a step size of 1 over 200 frames.

Chapter 6

Conclusions

We have created a system for performing collision detection on deformable models. This system is built upon AABBs and GPU-based culling. Our system switches between AABBs and GPU-based culling using timing and coherency metrics. For systems where AABB-based collision detection is the optimal solution we perform GPU-based culling in parallel with hierarchy rebuilds. The rate of rebuild is determined by comparing GPU-based culling timings and AABB timings. When our system finds a scenario that is best handled by GPU culling, our advanced GPU culling technique is invoked.

Our GPU Culling is an extension of current GPU culling systems. Our system determines view direction orderings by rating views based on culling efficiency and coherency. Our algorithm also sorts the objects. Finally our GPU cull calculates the number of triangles to render using coherency based metrics that predict the collision detection time for either increasing or decreasing the number of triangles per query.

Our system performs well on many different types of deformable models and handles rigid bodies well. By switching between Mode 1 and Mode 2 we are able to choose the best technique based on past experience. The cost of exploration is distributed across the entire simulation and becomes transparent.

Our results show that GPU culling on current hardware is most useful in two cases. The first case is when objects deform so fast that the time to rebuild the hierarchy over the frames becomes very large. The second case is that in which nearly all triangles are in close proximity, with few actual collisions. Using our form of GPU based culling, the highest-rated view directions, sorting, and triangles per query are

computed. This can result in an order of magnitude speedup in some cases.

In some simulations models deform enough to warrant hierarchy rebuilds but do not warrant switching entirely to GPU culling. In this case our technique rebuilds the hierarchy in a thread while performing GPU based culling in the foreground.

Our techniques and equations for determining when a combined approach can be faster than AABB based culling alone could be used as a guideline by Independent Hardware Vendor (IHV)s to set and achieve occlusion query throughput goals.

Bibliography

[1] T. Larsson and T. Akenine-Mller, "Strategies for bounding volume hierarchy updates for ray tracing of deformable models," *Tech. Report*. [Online]. Available: citeseer.ist.psu.edu/larsson03strategies.html

[2] 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*, pp. 25–32, 2003.

[3] T. Moller, "A fast triangle-triangle intersection test," *JGTOOLS: Journal of Graphics Tools*, vol. 2, 1997. [Online]. Available: citeseer.ist.psu.edu/moller97fast.html

[4] J. F. Alejandro Garcia-Alonso, Nicols Serrano, "Solving the collision detection problem," *IEEE Computer Graphics and Applications*, vol. 14, no. 3, pp. 21–36, 5 1994.

[5] S. G. M.C. Lin, "Collision detection between geometric models: A survey," *Proceedings of IMA, Conference of Mathematics of Surfaces*, pp. 602–608, 1998. [Online]. Available: citeseer.ist.psu.edu/lin98collision.html

[6] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan, "Efficient collision detection using bounding volume hierarchies of k-DOPs," *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 21–36, 1998. [Online]. Available: citeseer.ist.psu.edu/klosowski96efficient.html

[7] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi, "I-collide: An interactive and exact collision detection system for large-scale environments," in *Proc. ACM Interactive 3D Graphics Conf.*, 1995, pp. 189–196.

[8] P. Jimnez, F. Thomas, and C. Torras, "3D Collision Detection: A Survey," *Computers and Graphics*, vol. 25, no. 2, pp. 269–285, Apr. 2001. [Online]. Available: citeseer.ist.psu.edu/431815.html

[9] S. Gottschalk, M. C. Lin, and D. Manocha, "OBBTree: A hierarchical structure for rapid interference detection," *Computer Graphics*, vol. 30, no. Annual Conference Series, pp. 171–180, 1996. [Online]. Available: citeseer.ist.psu.edu/gottschalk96obbtree.html

[10] G. van den Bergen, "Efficient collision detection of complex deformable models using AABB trees," *Journal of Graphics Tools: JGT*, vol. 2, no. 4, pp. 1–14, 1997. [Online]. Available: citeseer.ist.psu.edu/vandenbergen98efficient.html

[11] T. Larsson and T. Akenine-Mller, "Collision detection for continuously deforming bodies," *Eurographics*, pp. 325–333, 2001, 2001. [Online]. Available: citeseer.ist.psu.edu/larsson01collision.html

[12] D. JAMES and D. PAI, "Bd-tree: Output-sensitive collision detection for reduced deformable models," *ACM Transactions on Graphics (SIGGRAPH 2004)*, vol. 23, pp. 393–398, Aug 2004. [Online]. Available: citeseer.ist.psu.edu/656788.html

[13] N. Govindaraju, M. Lin, and D. Manocha, "Quick-cullide: Fast inter- and intra-object collision culling using graphics hardware," *Proceedings of IEEE Virtual Reality Conference*, pp. 59 – 66, 319, 2005.

[14] ——, "Fast and reliable collision culling using graphics hardware," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 2, pp. 143–154, Mar/Apr, 2006.

[15] S. Aharon and C. Lenglet, "Collision detection algorithm for deformable objects using opengl," in *Proceedings of the 5th International Conference on*

Medical Image Computing and Computer-Assisted Intervention-Part II, 2002, pp. 211–218. [Online]. Available: citeseer.ist.psu.edu/aharon02collision.html

[16] F. Ganovelli, J. Dingliana, and C. O'Sullivan, "Buckettree: Improving collision detection between deformable objects," in *Spring Conference in Computer Graphics (SCCG2000)*, 2000, pp. 156–163. [Online]. Available: citeseer.ist.psu.edu/ganovelli00buckettree.html

[17] W. S. W. George Baciu, "Image-based techniques in a hybrid collision detector," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 2, pp. 254–271, Apr-Jun 2003.

[18] M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets, and M.Gross, "Optimized spatial hashing for collision detection of deformable objects," *Proceedings of Vision, Modeling, Visualization VMV'03*, pp. 47–54, 2003. [Online]. Available: citeseer.ist.psu.edu/teschner03optimized.html

[19] A. Smith, Y. Kitamura, H. Takemura, and F. Kishino, "A simple and efficient method for accurate collision detection among deformable objects in arbitrary motion," *Proc. of the IEEE Virtual Reality Annual International Symposium*, pp. 136–145, 1995. [Online]. Available: citeseer.ist.psu.edu/smith95simple.html

[20] K. E. H. III, A. Zaferakis, M. C. Lin, and D. Manocha, "Fast and simple 2d geometric proximity queries using graphics hardware," in *Symposium on Interactive 3D Graphics*, 2001, pp. 145–148. [Online]. Available: citeseer.ist.psu.edu/hoff01fast.html

[21] C. W., W. H., Z. H., B. H., and P. Q., "Interactive collision detection for complex and deformable models using programmable graphics hardware," in *Proceedings of the ACM symposium on Virtual reality software and technology*, 2004, pp. 10–15.