Collision Queries using Oriented Bounding Boxes

by

Stefan Gottschalk

A Dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

2000

Approved by:

Dinesh Manocha, Advisor

Ming C. Lin, Advisor

Frederick P. Brooks, Jr., Reader

Copyright © 2000
Stefan Gottschalk
All rights reserved

Abstract

Collision Queries using Oriented Bounding Boxes

Under the direction of Dinesh Manocha and Ming Lin

Bounding volume hierarchies (BVHs) have been used widely in collision detection algorithms. The most commonly used bounding volume types are axis-aligned bounding boxes (AABBs) and spheres, which have simple representations, compact storage, and are easy to implement. This dissertation explores the use of oriented bounding boxes (OBBs), which may be aligned with the underlying geometry to fit more tightly. We believe that OBBs historically have been used less often because previously known methods for testing OBBs for overlap were relatively expensive, good methods for automatic construction of trees of OBBs were not known, and the benefits of using OBBs were not well understood. In this dissertation we present methods for building good trees of OBBs and demonstrate their use in efficient collision detection. In particular, we present a new OBB overlap test which is more efficient than previously known methods. We also examine some of the trade-offs of using OBBs by analyzing benchmark results comparing the performances of OBBs, AABBs and spheres, and we show that OBBs can significantly outperform the latter two bounding volumes for important classes of inputs. We also present two new tools, the bounding volume test tree (BVTT) and the contact pair matrix (CPM), for analyzing collision queries. Finally, we describe the design and implementation of a software system that permits the fair comparison of algorithmic variations on BVH-based collision detection.

Contents

List of Tables ix

List of Figures x

1 Introduction 1

1.1 Applications of Collision Queries 1
1.2 Model Representations 3
1.3 Bounding Volume Hierarchies 4
1.4 Oriented Bounding Box Trees (OBBTrees) 6
1.5 Cost of Collision Query 7
1.6 Characterization of Proximity 8
1.7 Thesis 9
1.8 Results 10
1.9 Summary of Chapters 11

2 Bounding Volume Hierarchies and Collision Queries 13

2.1 Example: Collision Detection with Sphere Trees 13
2.1.1 Bounding Volume Hierarchies 14
2.1.2 A Collision Query 16
2.2 Types of Bounding Volumes 20
2.2.1 Bounding Spheres 20
2.2.2 Axis-Aligned Bounding Boxes 20
2.2.3 Oriented Bounding Boxes 21
2.3 Updating Bounding Volumes 21
2.3.1 Coordinate Systems 21
2.3.2 Aligned and Non-Aligned Updates 22
2.3.3 Choice of Update Space 23
2.3.4 Update Caching 23

2.4 Tree Traversal Rules 24
2.5 The Bounding Volume Test Tree 24
2.5.1 Invariants Applicable to Binary BVTTs 26
2.5.2 Limit on Speedup due to Trivial Rejection Tests 28
2.5.3 Limit on Speedup for Temporal Coherence Scheme 29
2.6 Contact Pair Matrix 29
2.6.1 Correctness of Tandem Tree Traversal 30
2.6.2 Ideal BVs and Lower Bounds on BV Tests 33
2.6.3 Lower Bounds on BV Tests 34
2.6.4 Summary of CPM Result 36
2.7 Summary of Chapter 37

3 Fitting an OBB 38
3.1 Building a Bounding Volume Hierarchy 38
3.1.1 Top-Down 39
3.1.2 Bottom-Up 40
3.1.3 Incremental Insertion 40
3.2 Computing the OBB Parameters 40
3.3 Covariance-Based Methods 41
3.3.1 Distribution of Vertices 42
3.3.2 Distribution of Extremal Vertices 43
3.3.3 Distribution of Triangles (or Line Segments) 44
3.3.4 Distribution of Convex Hull Boundary 45
3.3.5 Degenerate Convex Hulls 46
3.3.6 Summary of Covariance Method 47
3.4 Formulas for Covariance Matrices 48
3.4.1 Covariance Matrix of Points 48
3.4.2 Covariance Matrix of Line Segments 49
3.4.3 Covariance Matrix of Triangles 53
3.5 Tightness of OBBs 57
3.5.1 Non-Optimality of OBBs 57
3.5.2 Worst-Case Input for Covariance-Fitted OBBs 58
3.6 Summary of Chapter 62

4 OBB Overlap Test **64**
4.1 Previous Methods 64
4.1.1 Linear Programming 65
4.1.2 Gilbert-Johnson-Keerthi Distance Computation 65
4.1.3 Lin-Canny Distance Computation 66
4.1.4 Greene's Box-Polytope Test 66
4.1.5 Exhaustive Edge-Face Testing 67
4.1.6 Summary of Previous Methods 67
4.2 Separating Axis Theorem 67
4.2.1 Minkowski Difference of Two OBBs 68
4.2.2 Minkowski Difference and Separating Axes 70
4.2.3 Combinatoric Structure of Separating Axis Theorem 71
4.3 Implementation for OBBs 72
4.3.1 Posing the Problem 72
4.3.2 Simple Formulation 73
4.3.3 Optimization: Interval Widths and Separations 73
4.3.4 Optimization: Structure of Candidate Vector 76
4.3.5 Optimization: Choice of Coordinate System 78
4.3.6 Optimization: Removing Common Subexpressions 80
4.4 Special Case OBBs 80
4.5 Robustness Issues 81
4.6 Program Code 84
4.7 Summary of Chapter 86

5 Performance and Analysis of Collision Queries **88**
5.1 Query Cost Equation 88
5.2 Speed of Fundamental Tests 89
5.2.1 Comparison of OBB Overlap Tests 89
5.2.2 Comparison of Overlap Tests for BV Types 90
5.3 Collision Query Benchmarks and Analysis 91
5.3.1 Sliding Sphere Benchmark 91
5.3.2 Parallel Close Proximity 97
5.3.3 Transverse Contact 107
5.4 Summary of Chapter 112

6 **Previous Work** **113**
6.1 Problem Domain Classification 113
6.1.1 Model Representations 113
6.1.2 Proximity Queries 116
6.1.3 Simulation Environments 118
6.2 Collision Detection for Polygonal Models 119
6.2.1 Convex Polytopes 119
6.2.2 Bounding Volume Hierarchies 123
6.2.3 Spatial Partitionings for Polygon Soups 124
6.3 N-Body Processing 126
6.3.1 Scheduling Scheme 126
6.3.2 Sorting-Based Sweep and Prune 126
6.3.3 Interval Tree for 2D Intersection Tests 126
6.3.4 Uniform Spatial Subdivision 127
6.4 Public Domain Software Packages 127
6.4.1 I-COLLIDE Collision Detection System 127
6.4.2 RAPID Interference Detection System 127
6.4.3 V-COLLIDE Collision Detection System 128
6.4.4 Distance Computation between Convex Polytopes 128
6.4.5 SOLID Interference Detection System 128
6.4.6 V-Clip Collision Detection System 128

7 **Implementation of Framework for Proximity Queries** **130**
7.1 Introduction to FPQ 131
7.2 Bounding Volume Hierarchy Data Structure 133
7.2.1 Writing Bounding Volume Hierarchies to Disk 137
7.2.2 Optimizing Memory for Embedding 138
7.3 Structure of the Query Procedure 140
7.3.1 Unification with Distance and Span Queries 142
7.3.2 Descent Rules 143
7.3.3 Fixed Branching Factor 144
7.3.4 Updating Bounding Volumes 145
7.3.5 Implementing Update Caching 147
7.4 Bounding Volume Abstractions 147
7.4.1 The Case Against Virtual Functions 148

7.4.2 Bounding Volume Representation 149
7.4.3 Bounding Volume Access Routines 149
7.5 Building the Bounding Volume Hierarchy 151
7.5.1 Loading the Model 151
7.5.2 Building the Tree 151
7.6 Summary of Chapter 152

8 Future Work 153
8.0.1 Improved Proximity Characterization 153
8.0.2 Analysis of BVH-Based Distance Queries 154
8.0.3 Penetration Distance Queries for Polygon Soups 154
8.0.4 Hausdorff Distance Queries for Polygon Soups 155
8.0.5 Hausdorff-Driven Greedy Bottom-Up OBBTree Construction . 155
8.0.6 Lazy Evaluation and Curved Surfaces 155
8.0.7 Run-Time Adaptive Hierarchies 156
8.0.8 Deformable Models 156

A Proximity Queries 158
A.1 Types of Queries 158
A.2 Queries and Minkowski Differences 160
A.2.1 Collision Queries and Minkowski Space 162
A.2.2 Separation Distance Queries and Minkowski Space 163
A.2.3 Spanning Distance Queries and Minkowski Space 164
A.2.4 Penetration Distance Queries and Minkowski Space 164
A.2.5 Hausdorff Distance Queries and Minkowski Space 165
A.3 Accelerating Collision, Separation and Span Queries 166
A.3.1 Collision Queries 166
A.3.2 Separation Distance Queries 168
A.3.3 Spanning Distance Queries 169
A.3.4 Penetration Distance Queries 169
A.3.5 Hausdorff Distance Queries 170

Bibliography 171

List of Tables

4.1 A summary of the costs of various steps in the OBB overlap test based on separating axes. If one of the boxes is already expressed in terms of the coordinate system of the other, then the first 63 arithmetic operations can be skipped. If one is not concerned about guarding against arithmetic error, the second 9 operations can be skipped. If the first axis test proves the OBBs disjoint, then only 89 arithmetic operations are performed (including transform and error guard steps). If all the tests are executed, then 252 arithmetic operations are performed. 87

5.1 Timing of OBB overlap tests. The benchmarks were run on an SGI Maximum Impact with a 250 MHz MIPS R4400 CPU, MIPS r4000 FPU, 192MB RAM, 2MB secondary cache. 90

5.2 Timing of overlap tests for different BV types. The benchmarks were run on an SGI Maximum Impact with a 250 MHz MIPS R4400 CPU, MIPS r4000 FPU, 192MB RAM, 2MB secondary cache. 90

6.1 Five proximity measures. 116

6.2 Some of the basic approaches for computing intersection or contact status of convex polyhedra. 119

6.3 Bounding volume hierarchy types for polygon soups. 123

6.4 Spatial partitionings for polygon soups. 124

List of Figures

1.1 Collision queries can be used on this polygonal engine model to verify collision-free paths of mechanical components. Model courtesy of Engineering Animation, Incorporated. 2
1.2 A bounding volume hierarchy of spheres. The underlying geometry depicted here is a collection of line segments. A top-level sphere encloses the entire model. The next level down, two spheres each cover half the segments. At each stage, the primitives covered by a BV are also covered by the union of that BV's children, and leaf nodes in the hierarchy each cover a single primitive. Notice that child BVs can extend beyond the space of the parent, and that sibling BVs can overlap each other. 4
1.3 Different types of bounding volumes have different characteristics. Simple shapes like spheres and axis-aligned bounding boxes (AABBs) do not tend to fit their underlying geometry as tightly as do more complex shapes such as oriented bounding boxes and convex hulls. Simple shapes tend to have fast overlap tests, but require many such tests to perform a collision query. Complex shapes have slower overlap tests, but require fewer of them to perform a query. 5
1.4 Here we show four successive levels of a hierarchy of OBBs covering a collection of line segments. Due to their tighter fit, OBBs tend to require fewer overlap tests to complete a collision query than do spheres or AABBs. 7
1.5 Parallel Close Proximity and Transverse Contact situations. Parallel close proximity is characterized by the size of the gap between two parallel surfaces. Transverse contact is characterized by the angle at which two intersecting surfaces meet. 8

2.1 A model and a symbolic presentation of its bounding volume hierarchy. Part (a) is the model, consisting of four primitives (in this 2D example, the primitives are line segments). Part (b) shows a recursive partitioning of the primitives, while part (c) assigns a name to each partition. Part (d) assigns a name to the bounding volumes of the partitions, and this is the symbolic presentation of the bounding volume hierarchy of model A. 14
2.2 Part (a) shows the top-level bounding volume, named V_A, which is the root node of the bounding volume hierarchy. Part (b) shows the next level of the BVH, consisting of the BVs V_{A_0} and V_{A_1}. Part (c) shows the leaf nodes of the BVH. 15
2.3 Model B, consisting of 5 primitives. Part (a) shows the model graphically, while parts (b) through (d) show the recursive partitioning, the names for those partitions, and the names for the bounding volumes for those partitions, respectively. Notice that this partitioning has a 3-way branch. 15
2.4 Graphical display of the BVH for model B. Part (a) shows the top level node, part (b) shows the three second level nodes, and part (c) shows the leaf nodes. Notice that node V_{B_1} is a second level leaf node. . . . 16
2.5 The models juxtaposed – bounded apart by their top level spheres. . 16
2.6 A scenario in which the models do not touch, but the top-level BVs do not bound the models apart. Part (a) shows the models and the overlapping top level BVs, V_A and V_B. Part (b) shows the children of V_A being tested against V_B, and since there are no overlaps, the models are known to be disjoint. Part (c) shows an alternative way to show that the models are disjoint: testing the children of V_B against V_A finds no overlaps. 17
2.7 The models touch, and therefore the top level bounding volumes must also touch. 18
2.8 For each sub-problem, descending model B yields three more, for a total of six sub-problems. In sub-problem (b3), the bounding spheres are just barely disjoint. 19
2.9 Children of V_{B_0} do not touch V_{A_0}. 19

2.10 The model geometry is specified in terms of the model coordinate system. Queries must specify the placement of the models in the common world coordinate system. 22
2.11 An aligned type, such as AABBs, cannot be represented after arbitrary rotation. To maintain a conservative bound, we can either reexamine the geometry to produce a new tightly-fitted aligned box, or with less computation we can cover the rotated box by an aligned one. 23
2.12 With an aligned type such as AABBs, Using world space updates can mean testing loose BVs, whereas using a model space update guarantees that one of the BVs will be tight. In this particular example, the BVs overlap when using world space updates, but not when using either of the model space updates. 24
2.13 Pictorial representation of the bounding volume test tree. 25
2.14 Symbolic representation of the bounding volume test tree. Circles represent tests between BVs, and triangles represent tests between polygons. An "X" indicates the items were touching, otherwise they were not touching. 25
2.15 The numbers of different types of nodes in any binary BVTT satisfies the equation $1 = D - I + 2S + 2K$, where D and I count disjoint and intersecting BVs, respectively, S counts the number of separated primitives tested, and K counts the number of contacts found. . . . 26
2.16 A series of substitutions can grow any realizable BVTT from an initial test between disjoint BVs. We prove that all BVTTs satisfy the constraint $1 = D - I + 2S + 2K$ by inducting on the number of substitutions. 27
2.17 The contact pair matrix (CPM) cross indexes all the primitives of two models, showing which pairs are in contact with an "X". 30
2.18 Testing two BVs corresponds to examining a sub-block of the contact pair matrix. If the BV test yields disjoint, then the sub-block is known to be empty of contacts. The converse is not true, since the BV tests are conservative – the BV tests can yield intersection even though the sub-block contains no contacts. 31
2.19 This is the contact pair matrix for the sphere tree example. The single contact between a_4 and b_3 is shown. 32
2.20 This is the sequence of subdivisions for the sphere tree example. Each stage of the subdivision is accompanied by the corresponding BVTT. 32

2.21 The most favorable arrangement of $k > 0$ contacts among n^2 cells is as a block which can be isolated in the minimal number of subdivisions, $\log_2(n^2/k)$. Once the block is isolated, $k-1$ additional subdivisions are required to isolate all k contacts. The profile of the BVTT is a long thin stalk $\log_2(n^2/k)$ levels long, and terminating in a fully branched subtree of k leaf nodes. 35
2.22 The least favorable arrangement of $k > 0$ contacts among n^2 cells is distributed such that the topmost $k-1$ subdivisions have a contact on each side. After partitioning the CPM into k blocks containing one contact each, each block requires $\log_2(n^2/k)$ more subdivisions to locate the contact. The profile of the BVTT is a fully branched subtree with k stalks hanging from its leaves, each stalk being $\log_2(n^2/k)$ levels long. 36

3.1 Building a bounding volume hierarchy with the top-down split-and-fit technique. The primitives are partitioned according to which side of a dividing plane they lie on, and the two new groups are each fitted with their own bounding volumes. Our choice of dividing plane is one which is orthogonal to the direction of greatest spread, and which passes through the centroid of the group of primitives. 39
3.2 Collection of points with direction of maximal spread and OBB . . . 42
3.3 Interior vertices affect alignment. 43
3.4 Sampling along boundary affects alignment. 44
3.5 Integrating across the primitives (here, line segments) gets better results. 44
3.6 Small outliers expand the box without significantly realigning it. . . . 45
3.7 Using the convex hull boundary gives good fit. 46
3.8 Demonstration that covariance-aligned OBBs are suboptimal. P_1 is OBB for M_1. Expanding M_1 slightly, but still staying within P_1, obtains a new model, M_2, whose covariance-aligned OBB is P_2. Since P_1 and P_2 both cover M_1 and M_2, one of them must be suboptimal. . . . 58
3.9 Because a square has the same statistical spread in all directions, any orientation may be chosen for the covariance-aligned OBB, and the worst choice has twice the area of the optimal choice. 59

3.10 Tightness of covariance-fitted 2D OBBs for 10,000 randomly generated point sets. Each used the covariance method to fit an OBB to a set of four points randomly chosen in the unit square. The area of the covariance-fitted OBB was compared to the minimum possible OBB. 10,000 trials were conducted. This plot shows that most of the covariance-fitted 2D OBBs are within 10% of the minimum area. . . 60
3.11 Relationship between eccentricity and volume optimality. The plot suggests that if the eccentricity of the planar input figure exceeds 10, then the covariance-fitted OBB will have within a factor 1.3 of minimum possible area. 61

4.1 The Gauss map $G(A)$ of an OBB A is three mutually orthogonal great circles. Since the Gauss map of the minkowski difference $G(A \ominus B)$ of two polytopes A and B is the superposition of their individual Gauss maps, $G(A \ominus B)$ consists of six great circles, of which the first three are mutually orthogonal, and the second three are mutually orthogonal. This can be used to prove that in general the polytope $A \ominus B$ has 30 faces, 60 edges, and 32 vertices. 68
4.2 If A and B are OBBs, then $P = A \ominus B$ is the intersection of 15 slabs. A given slab contains the origin if and only if the axis perpendicular to the slab is a separating axis. In fact, the distance of the origin O from the slab equals the length of the gap between the images of A and B under axial projection onto the separating axis. 70
4.3 An axis test determines whether the images of A and B are disjoint under axial projection. Fifteen such tests are sufficient to determine whether two OBBs overlap. 72
4.4 The box centers project onto the midpoints of their intervals. The separation of the midpoints equals the length of the projection of the vector joining the box centers: $s = \mathbf{T} \cdot \mathbf{n}/|\mathbf{n}|$. 73
4.5 The half-width (radius) of an interval equals the sum of the lengths of the projections of the box axes. For 2D OBBs, as depicted here, we have only 2 box axes, but for 3D OBBs we would have 3 box axes. . . 74
4.6 Two intervals are disjoint if and only if the separation of their midpoints is greater than the sum of their half-widths (radii). The relevant comparison is $s > r_A + r_B$. 74

5.1 Polygon count, geometric complexity, and geometric proximity all contribute to the work of processing a query. 91

5.2 Schematic of simple benchmark: one stationary and one moving tessellated sphere. 91

5.3 A sphere of 90 triangles, tessellated by latitude and longitude. The spheres used in the benchmark each have 40,000 triangles. 92

5.4 Number of contacts found as sphere moves. This shows that the number of contacts is proportional to the length of the intersection curve, plus a random component induced by the random orientations of the sphere models. 93

5.5 Number of BVs tested as sphere moves. This distinguishes OBB from AABBs and spheres, showing that OBBs require many fewer overlap tests than the others, especially where the models first make contact. This means that OBBs have more pruning power than spheres and AABBs. 94

5.6 Number of primitives tested as sphere moves. Because of the superior pruning power of OBBs, we descend less often to the leaf nodes of the bounding volume hierarchies, and therefore test fewer primitives. . . 95

5.7 Ratio of AABB tests and sphere tests over OBB tests. This shows how much greater pruning power OBBs have over AABBs and spheres. It shows that OBBs require approximately 30 times fewer BV tests than spheres just prior to contact, and 25 times fewer than AABBs. Where there is contact, OBBs appear to have 2 to 9 times the pruning power of the other two, depending on where the sphere models are. 96

5.8 *Contact cost* as a function of position of smaller sphere. The contact cost is the number of BV tests divided by the number of contacts found. This normalizes for the size of the query's output, measuring the efficiency of the query. 97

5.9 Cutaway view of parallel close proximity configuration, with coordinate axes included for reference. 98

5.10 Schematic of Parallel Close Proximity experiment. 99

5.11 Parallel Close Proximity – Number of bounding volume tests versus surface separation (log-log plot). 100

5.12 Schematic diagram of a typical plot of number of BV tests versus surface separation, for Parallel Close Proximity situations. 100

5.13 Lower ledge in Parallel Close Proximity plot is created by clearance afforded by early levels in the BVHs. 101
5.14 Footprint and profile of spherical BVs. The same measures are valid for other BV types. 103
5.15 Diameter and thickness for OBBs are related by curvature of bounded geometry, which is the basis for their quadratic convergence. 103
5.16 OBBs can be seen to converge more rapidly than AABBs and spheres. The aspect ratio of AABB and sphere children resemble that of their parents, while the aspect ratio of OBB children tend to be greater than their parents. 104
5.17 Average thickness of OBBs on a given level of the BVH (log plot). . . 106
5.18 Angle at which surfaces meet as a function of position of moving sphere.107
5.19 Sine of the surface intersection angle at each position of the moving sphere. 108
5.20 Transverse contact situation. 109
5.21 Dependence of contacts on tessellation of maximally transverse spheres (reference line has slope 1/2). 109
5.22 Dependence of BV tests on tessellation of maximally transverse spheres (reference lines have slope 1/2). 110
5.23 Dependence of contact cost on tessellation of maximally transverse spheres (reference lines are level). 111

6.1 A Taxonomy of 3D Model Representations 114

7.1 BVH memory diagram – the vertex array is optional. 134
7.2 The basic structures, implemented in C++. 135
7.3 A model and its various bounding spheres. 135
7.4 The model's bounding sphere hierarchy. 136
7.5 The model's node and face arrays. 136
7.6 Access routines for node operations. 138
7.7 Recursive and iterative forms of collision query routine. 141
7.8 Collision, distance, and span queries have very similar structure. . . . 142
7.9 Common procedure for all queries. 143
7.10 A version of `DescendA` which descends the largest diameter BV. . . . 144
7.11 We use a loop for variable branching factors, but unroll them for binary trees. 144

7.12 `TestBV` performs updates prior to bounding volume testing. 145
7.13 `TestBV()` with update caching enabled. 146
7.14 The FPQ_BVReps structure is defined according to the bounding volume type. 149
7.15 Metrics for bounding volumes. 149
7.16 Proximity query functions for bounding volumes. 150
7.17 Procedure for updating a bounding volume. The bounding volume A gets transformed by rotation $\mathbf{R}$ and translation $\mathbf{T}$, with the result being assigned to Z. 150
7.18 Procedure for fitting a BV to a collection of faces. F is a list of n faces. The resulting BV is placed in A. 150
7.19 This is the interface for creating a model. `BeginModel()` is called first, followed by many calls of `AddFace()`, and ending with `EndModel()`. Each call of `AddFace()` adds a face to a growing list. The hierarchy is built when `EndModel()` is called. 151

A.1 The Minkowski sum of two line segments is a parallelogram: the shape swept out by using one as a “brush” and the other as the brush’s path. 161
A.2 The Minkowski difference of two shapes is the Minkowski sum of the first shape with the inverse of the second shape, where inverse means reflecting through the origin. 162
A.3 The Minkowski difference of two models depends on their shapes and their relative placement. Model A has 4 components, and model B has 5 components, and the Minkowski difference has $4 \times 5 = 20$ components. 162
A.4 The Minkowski difference of intersecting models touches the origin. The components of the Minkowski difference covering the origin identify all the primitive pairs which touch. 163
A.5 The point of the Minowski difference closest to the origin corresponds to the points of closest approach of the two point sets, indicated by the gray arrow. 164
A.6 The point in the Minowski difference furthest from the origin corresponds to the points of greatest separation of the two point sets, indicated by the circles. 165
A.7 The penetration distance corresponds to the closest point outside the Minkowski difference to the origin, as indicated by the circles. 166

A.8 The Hausdorff distance of A from B corresponds to the last point of A covered as B is progressively expanded. This last covered point is the point on A which is most distant from B, and this distance is the Hausdorff distance. 167

Chapter 1

Introduction

This chapter describes the context of the work, presents the thesis statement, and provides an overview of the dissertation. Section 1.1 describes collision queries in detail and discusses their applications. Section 1.2 discusses different model representations. Section 1.3 introduces bounding volume hierarchies and describes how they can be used to perform collision queries. Section 1.4 describes oriented bounding boxes. Section 1.5 discusses the factors contributing to the cost of a collision query. Section 1.6 discusses how we characterize proximity. Section 1.7 presents our thesis statement. Section 1.8 lists our results. Section 1.9 gives a summary of each of the chapters.

1.1 Applications of Collision Queries

We define an *object* or a *model* to be a given subset of 3-dimensional space. The shape of this subset might be represented by a collection of polgyons, or by primitive solid shapes, or by curved surfaces, as described in Section 1.2.

A *collision query* determines the intersection between given objects, and is used in computer-aided design and manufacturing, animation systems, and physically-based modeling. For example, a virtual prototyping application may run the components of an engine model – such as the one shown in Figure 1.1 – through their prescribed motions and use collision queries to verify that fitted parts do not interpenetrate at any step.

A *distance query* computes the distance between two objects. An architectural design system may perform distance queries to verify that certain functional components – for example, an electrical conduit and a fuel line network – are separated

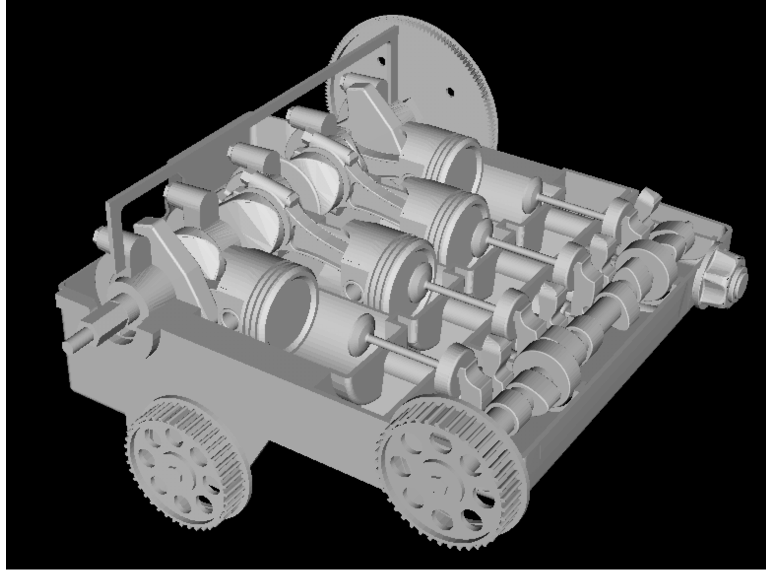


Figure 1.1: Collision queries can be used on this polygonal engine model to verify collision-free paths of mechanical components. Model courtesy of Engineering Animation, Incorporated.

by some minimum distance at all points. Path-planning systems, which automatically find viable paths for assembly and disassembly of multiple parts, or which guide robots through obstacles in an environment, often require collision detection or distance computation as subroutines of their own higher-level algorithms.

For many of these applications, collision and distance queries are computational bottlenecks. For instance, architectural designs can use models consisting of millions of polygons, and the time required to perform collision or distance queries can hinder interactivity. Similarly, applications using force-feedback devices may need to compute all contacts between complex 3D models in less than one millisecond. Reducing the execution time of these queries would lead to greater performance and greater usability of these tools.

Distance and collision queries are members of a larger class, called *proximity queries*, which yield information regarding the relative placement of models. Other examples of proximity queries are spanning distance (the distance between the most separated points between the models), the penetration distance (the smallest translation required to separate two models), and the Hausdorff distance (the greatest distance of any of the points of one model from all the points of the other model). More details on these query types are given in Appendix A.

Furthermore, there are queries applicable to dynamic scenarios, such as finding

when the next contact between two moving bodies will occur, or finding a conservative bound on the time of next contact. Some applications may need extensions to these queries which work for collections of models, rather than just pairs of models.

There is a great variety of queries with different uses for different applications, and there is a great variety of algorithms for implementing them. In our survey of the previous work in Chapter 6 we will discuss some of them in more detail. In this dissertation, the algorithms we present will address collision queries for pairs of polygonal models.

1.2 Model Representations

Geometric models used in computer graphics, CAD/CAM, and robotics can be divided into two broad categories: polygonal and non-polygonal. *Polygonal representations* are collections of polygons, which may be unstructured, arranged as meshes, or form convex polytopes. Non-polygonal representations include parametric surfaces, implicit surfaces, and constructive solid geometry (CSG).

Polygon soups are arbitrary collections of polygons in 3-space, not necessarily connected. Unlike implicit and parametric representations, polygons cannot represent curved surfaces exactly. Instead, smooth curved surfaces are approximated by tessellating the surface to any desired precision. Within this limitation, polygons can approximate any of the shapes achievable by implicit or parametric surfaces. There is hardware support for rendering polygonal models in many available computer systems, which makes polygonal representations an attractive choice for visualization applications.

Another important class of polygonal models are the convex polytopes. They have special properties which are exploited by many collision and distance computation algorithms, particularly for scenarios exhibiting temporal coherence. Some of these algorithms permit models to be expressed as collections of convex polytopes.

In this dissertation, we are concerned exclusively with general polygonal models, such as polygon soups. Consequently, the algorithms we present are applicable to any polygonal model, convex or otherwise, and regardless of whether it forms a clean manifold or whether it contains topological information.

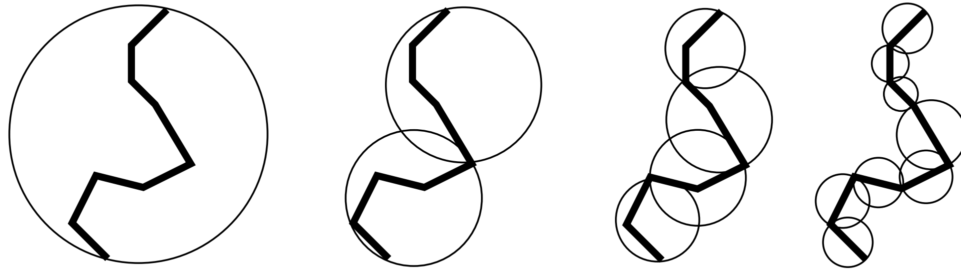


Figure 1.2: A bounding volume hierarchy of spheres. The underlying geometry depicted here is a collection of line segments. A top-level sphere encloses the entire model. The next level down, two spheres each cover half the segments. At each stage, the primitives covered by a BV are also covered by the union of that BV's children, and leaf nodes in the hierarchy each cover a single primitive. Notice that child BVs can extend beyond the space of the parent, and that sibling BVs can overlap each other.

1.3 Bounding Volume Hierarchies

Bounding volume hierarchies (BVHs) are one of the simplest and most widely used data structures for performing collision detection on complex models. The methods based on BVHs are easy to understand and implement. Moreover, they need not rely on any topological properties of the models, and consequently are applicable to polygon soups.

A *bounding volume hierarchy* is a tree of bounding volumes, such as spheres or boxes, whose collective leaf nodes spatially enclose all the model geometry, and in which each parent spatially encloses all the geometry covered by its descendent leaf nodes. Generally, each bounding volume in the hierarchy is made as small as possible – in terms of volume, surface area, diameter, or other appropriate size measure – while still covering its underlying geometry. A depiction of a hierarchy of spheres is shown in Figure 1.2.

Another class of spatial data structures used for collision detection are spatial subdivisions. Spatial subdivisions are a recursive partitioning of the embedding space, whereas bounding volume hierarchies are based on a recursive partitioning of the primitives of an object. Examples of spatial subdivisions are octrees, binary spatial partition trees (BSP trees), and k-d trees. In spatial subdivisions, sibling regions never overlap, never extend beyond the space of the parent, and their union always covers the parent space – these properties distinguish spatial subdivisions from bounding volume hierarchies.

Bounding volume hierarchies and spatial subdivisions have been used for a variety

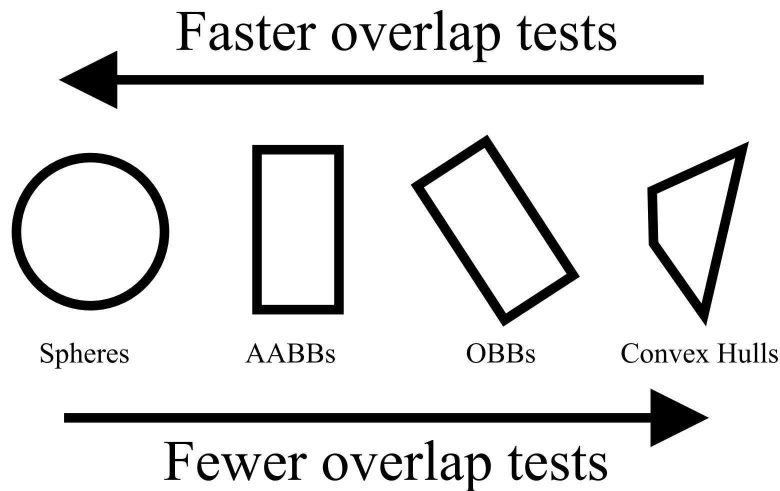


Figure 1.3: Different types of bounding volumes have different characteristics. Simple shapes like spheres and axis-aligned bounding boxes (AABBs) do not tend to fit their underlying geometry as tightly as do more complex shapes such as oriented bounding boxes and convex hulls. Simple shapes tend to have fast overlap tests, but require many such tests to perform a collision query. Complex shapes have slower overlap tests, but require fewer of them to perform a query.

of other geometric or graphics applications, such as occlusion culling, view frustum culling, backface culling, ray tracing, and hierarchical radiosity. In all of these applications, the hierarchical nature of the data structure is used in a divide-and-conquer approach to the problem. In this dissertation we restrict ourselves to collision query algorithms based on bounding volume hierarchies.

A BVH-based collision query proceeds by recursively testing bounding volumes of the models for overlap. For each such test, if they do overlap, then the children of the BVs are tested pairwise for overlap. If they do not overlap, then that recursion branch terminates. If two leaf nodes are found to overlap, then the polygons they enclose are tested pairwise, and the results of these tests are added to the query output.

There are a variety of algorithmic variations on the basic method. For example, one may use different BV types: spheres, axis-aligned bounding boxes (AABBs), arbitrarily oriented bounding boxes (OBBs), ellipsoids, convex hulls, fixed-direction hulls (also known as k-sided discrete orientation polytopes, or k-DOPS), cylinders, spherical shells, and others. Indeed, any shape may be used – it need not be convex or even connected – provided two operations are defined for it: First, how to fit the

shape to a collection of polygons, and second, how to test two such shapes for overlap. In general, more complex shapes fit more tightly to the underlying geometry, enabling the query to complete using fewer overlap tests, although each individual overlap test usually takes longer. The trade-off is between the number of tests and the speed of each test. A depiction of this trade-off is shown in Figure 1.3.

Another algorithmic choice is how to determine the sequence of BV overlap tests. One may proceed with a depth-first recursion, as described in the previous paragraph, or one may proceed with a breadth-first traversal of the tree of tests. Also, when two BVs are found to overlap, we can test the children of both pairwise, or we may choose to test one BV against the children of the other according to some criterion, such as their volumes, surface areas, or diameters. These choices may be termed *traversal rules*. Different rules may have different performance consequences, but all valid rules yield a correctly functioning algorithm.

Yet another algorithmic choice is how to construct the bounding volume hierarchy: top-down, bottom-up, or incremental insertion. A top-down method fits a BV to the entire model, partitions the primitives into two groups, and recursively applies the operation to each group. A bottom-up method fits BVs to each individual primitive, and then iteratively combines groups to form larger groups, fitting a new BV to each new group when it is formed. An incremental insertion technique builds a tree by inserting primitives into the tree structure one primitive at a time, adjusting the bounding volume hierarchy with each insertion. Each of these approaches has many variations. It is not yet known what methods produce trees enabling the most efficient collision detection.

1.4 Oriented Bounding Box Trees (OBBTrees)

An *oriented bounding box* (OBB) is an arbitrarily oriented rectanguloid. An *axis-aligned bounding box* (AABB) is a rectanguloid whose faces are aligned with the coordinate axes of its parent coordinate system. Whereas an AABB can be represented with just minimum and maximum extents along each axis, an OBB representation must encode not only position and widths, but also orientation. The advantage OBBs have over AABBs as bounding volumes is that they can bound their enclosed geometry more tightly. A 2D schematic representation of a hierarchy of OBBs is shown in Figure 1.4.

In this dissertation, we explore the general choice of BV type and its impact on

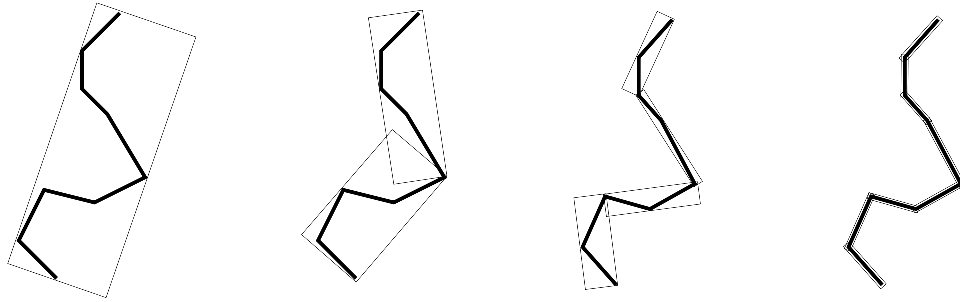


Figure 1.4: Here we show four successive levels of a hierarchy of OBBs covering a collection of line segments. Due to their tighter fit, OBBs tend to require fewer overlap tests to complete a collision query than do spheres or AABBs.

performance. In particular, we characterize the situations in which OBBs outperform the two most commonly used shapes, AABBs and spheres.

1.5 Cost of Collision Query

The fundamental operations of a collision query are the BV overlap test and the primitive overlap test. The time required to perform a collision query can be approximated as

$$T = N_v T_v + N_p T_p$$

where N_v and N_p are the number of overlap tests for bounding volumes and primitives, respectively, and T_v and T_p are the average times required to perform each such test. Queries using simple BVs, such as spheres and AABBs, exhibit large N_v and small T_v, while those using more complex BVs such as convex hulls or OBBs have smaller N_v and larger T_v. Also, using simple BVs tends to increase N_p, since the leaf nodes of simple BVs are less likely to bound the models apart than those of complex (and therefore tighter) BVs. It is not immediately obvious from this equation, given the opposite tendencies of T_v and N_v, what is the best choice of BV type. In fact, we believe that no single BV type is the best choice for all occasions – different applications may perform best with different BV types.

Several factors contribute to the cost of a collision query. Models with many polygons tend to be more expensive to query than models with fewer polygons. Models with more complex geometric shape (such as a tree) tend to be more expensive than ones with simple shape (such as a bowling ball), even if the polygon counts are the same. Collision queries are more expensive when the models are closer together than

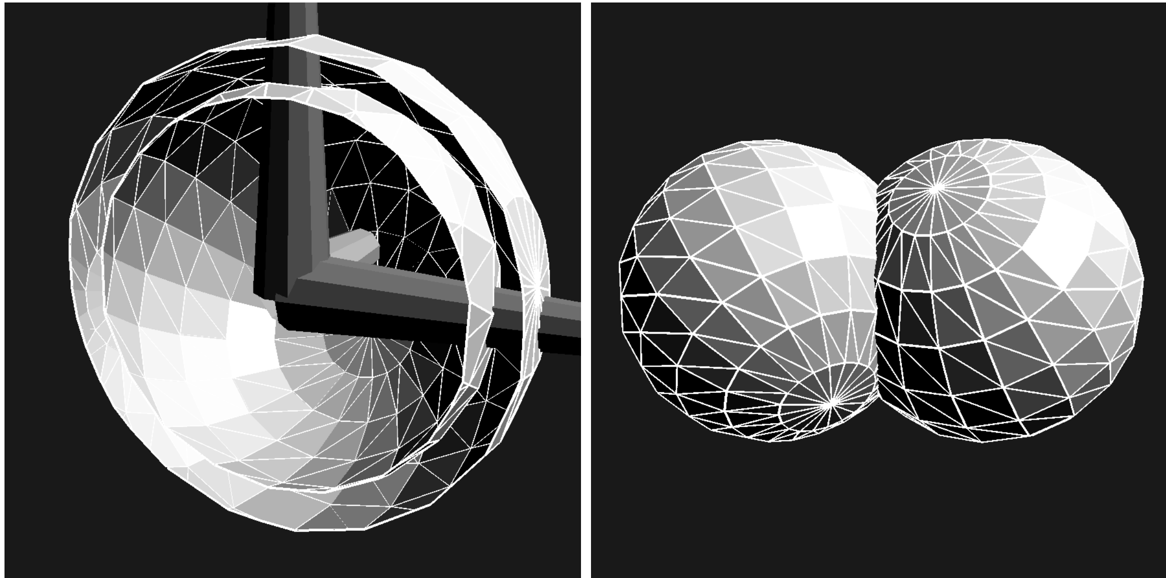


Figure 1.5: Parallel Close Proximity and Transverse Contact situations. Parallel close proximity is characterized by the size of the gap between two parallel surfaces. Transverse contact is characterized by the angle at which two intersecting surfaces meet.

when they are farther apart. Furthermore, collision queries between well-separated models can be resolved in a constant number of BV tests, regardless of the number of polygons – so certain spatial arrangements can make polygon counts irrelevant to the cost of a collision query. Finally, collision queries that yield a list of all touching polygons will tend to be more expensive as the number of contacts increases.

Thus the cost of a collision query depends not only on the size of the input (number of polygons in the two models) and the size of the output (the number of touching primitive pairs found), but also on the nature and degree of the proximity of the models. Input and output sizes are easily quantified. However, proximity is not so easily quantified, since it should capture the shape complexity of the models and how those shapes spatially interact.

1.6 Characterization of Proximity

Rather than attempt to characterize proximity in general, we examine two special cases: *parallel close proximity*, and *transverse contact*. Two objects are said to be in parallel close proximity when every point on one object is a constant distance from

the other object, and vice-versa. This type of proximity is represented by a single number: the separation distance (also called *gap size*). For example, two concentric spheres are in parallel close proximity, and the difference between their radii is the gap size.

Two shapes are in transverse contact if, at every contact point, their surfaces meet at a given angle. This intersection angle characterizes the transverse contact. For example, two overlapping spheres are in transverse contact – assuming one does not completely enclose the other – because their surfaces meet at the same angle everywhere along their intersection curve.

We will characterize the performance of BVH-based collision queries for these special cases of proximity. For instance, as the gap size between concentric sphere models is reduced, the number of BV tests required to perform a collision query increases because the BVHs of the models must be descended to ever-increasing depth in order to bound the models apart. Furthermore, in such situations, the number of BV tests is insensitive to the number of polygons in the models, provided they have been tessellated finely enough to allow the leaf nodes of their BVHs to bound the models apart.

For transverse contact, we find that increasing the tessellation of the models increases the number of contact pairs found, and also proportionally increases the number of BV tests required to find those contact pairs.

1.7 Thesis

The central idea of this dissertation is that by virtue of their tighter fit,

> **trees of oriented bounding boxes can yield superior performance for collision detection in parallel close proximity.**

Due to their tighter fit, collision queries using OBBs require fewer BV overlap tests than when using the simpler BV types. The differences in the pruning power of OBBs and the simpler BV types become most significant when large portions of each surface are very close to the other surface – such as when the surfaces are parallel and separated by a small gap.

1.8 Results

The specific new results of this dissertation are summarized as follows:

Efficient OBB Overlap Test: We propose an efficient overlap test for two OBBs, which completes in 89 operations in the best case and 252 operations in the worst case. This is an order of magnitude faster than the prior state of the art. It does not require special treatment of non-generic inputs, such as parallel faces or edges. Furthermore, it works correctly for boxes with zero widths, and advance knowledge of such inputs permits further simplification of the test. The test uses no complex data structures and has simple straight-line flow of control with possible early exits, which makes it a good candidate for microcoding. The OBB overlap test is described in detail in Chapter 4.

Fitting an OBB to a Set of Polygons: We present an $O(n \log n)$ method for finding a well-fitted OBB for a given set of polygons. Our approach is to characterize the surface of the convex hull of the polygons statistically, taking the principal directions of the covariance matrix of this distribution as the alignment for the OBB. Our OBB fitting method is described in detail in Chapter 3.

Analysis of OBBs for Collision Detection: We analyze performance of OBBs relative to spheres and axis-aligned bounding boxes (AABBs) in collision queries for parallel close proximity and transverse contact. We show that for parallel close proximity situations, OBBs asymptotically outperform AABBs and spheres as the gap size is reduced because of their quadratic convergence property. We also show that for transverse contact situations, the performance of BVH-based methods are asymptotically optimal as a function of model tessellation density, regardless of which BV type is used. This analysis is presented in Chapter 5.

Analysis Tools for Collision Queries: We describe two new tools for analyzing collision queries: the bounding volume test tree (BVTT) and the contact pair matrix (CPM). The BVTT is an organization of the overlap tests into a tree structure, where each node is labeled according to the kind of test (whether between BVs or between polygons) and its outcome (overlapping or disjoint). We show that the numbers of the different types of nodes in the BVTT for any query satisfy an invariant, and this fact is used to obtain bounds on the

speedup obtainable by certain acceleration techniques. The CPM is another way of organizing the tests, and is used to obtain a lower bound on the number of BV tests required to locate all k contacts among two models of n polygons each. Both these tools are used to obtain useful results about collision queries in this dissertation. These tools are presented in Chapter 2.

Accurate and Efficient Collision Detection Package A collision detection package called Robust and Accurate Polygon Interference Detection (RAPID) has been developed using a subset of the OBB algorithms described in this dissertation, and is presently in use by several researchers and businesses.

Experimental Framework for Proximity Queries The software system design employed in RAPID has been generalized to permit selection of bounding volume types, traversal rules, and other algorithmic variations, and implemented in a working system called Framework for Proximity Queries (FPQ). This system was used to gather all the performance data presented in this dissertation. This framework is a good testbed for incorporating new BV types and new proximity query algorithms based on bounding volume hierarchies. The implementation of FPQ is described in detail in Chapter 7.

1.9 Summary of Chapters

The rest of the dissertation is organized as follows:

Chapter 2: An overview of collision queries and bounding volume hierarchies. We present new tools to aid our intuition and our analysis – the bounding volume test tree (BVTT) and the contact pair matrix (CPM) – which are used to deduce bounds on performance under certain general assumptions. We review various BV types in detail, and discuss some elementary issues such as transforming BVs, and caching BV computations.

Chapter 3: How to build a bounding volume hierarchy of OBBs. We discuss the three major approaches to building bounding volume hierarchies: top-down, bottom-up, and incremental insertion. All three approaches require a method for fitting the bounding volume to a collection of polygons. For OBBs, the most difficult step in fitting to a set of polygons is choosing a suitable orientation. This chapter discusses how and why we use the principal directions of the surface

of the convex hull of the polygon vertices to choose the orientation. This chapter also discusses the optimality of OBBs computed by this method.

Chapter 4: The OBB overlap test. We state and prove the "separating axis theorem," apply it to OBBs, and then present an implementation of the test based on this theorem. The implementation is developed as an initial simple formulation, to which we then apply a series of arithmetic optimizations.

Chapter 5: Performance comparisons among types of BVs. We examine the speed of different overlap test methods for OBBs, concluding that our overlap test for OBBs is an order of magnitude faster than the prior state of the art. We also analyze the results of performance benchmarks, testing the pruning power of trees of spheres, AABBs, and OBBs under different circumstances. The analysis concludes that OBBs perform asymptotically less work than the other two types as a function of gap size in parallel-close-proximity situations. Also, we show that using BVHs of any of the three BV types perform within a constant factor of optimal execution time as a function of tessellation density in transverse-contact situations.

Chapter 6: A review of previous work on collision detection in the fields of CAD/CAM, robotics, computational geometry, and computer graphics. We categorize the previous work according to their different model representations, different types of queries, and the methods used to implement them.

Chapter 7: Design and implementation of a framework for proximity queries. This framework decomposes the proximity query problem into compatible abstract components, permitting the user to select BV type, traversal rules, arithmetic precision, tree construction rules, and other algorithmic variations. Also, it can be configured to perform collision detection, separation distance computation, or spanning distance computation. We used this system to compare the performance of different BV types for collision queries.

Chapter 8: Future work. We discuss promising research directions.

Chapter 2

Bounding Volume Hierarchies and Collision Queries

This chapter explains in detail how bounding volume hierarchies (BVHs) can be used to perform collision queries, and presents two new analysis tools: the *bounding volume test tree* (BVTT) and the *contact pair matrix* (CPM). The BVTT is an organization of the BV tests and primitive tests as a tree structure, and is subject to an invariant which we will use to establish upper bounds on the effectiveness of certain query acceleration techniques. The CPM is a different organization of those tests which will enable us to establish lower bounds on the number of BV tests required to perform a collision query.

Section 2.1 presents an example of a collision query using sphere-trees while introducing some useful notation and terminology. Section 2.2 describes the specifics of various BV types. Section 2.3 discusses issues regarding transformation of BVs. Section 2.4 discusses different rules for deciding how two bounding volume hierarchies are to be traversed. Section 2.5 describes a query in terms of its bounding volume test tree (BVTT), and uses it to deduce bounds on speedups achievable by some proposed acceleration techniques. Section 2.6 presents the contact pair matrix (CPM) and uses it to deduce lower bounds on performance of BVH-based collision queries.

2.1 Example: Collision Detection with Sphere Trees

We treat two models that reside in the plane, and are composed of line segments, but the methods described apply equally well to polygonal models embedded in 3-space. In general, we will not require our models to be connected or to possess any particular

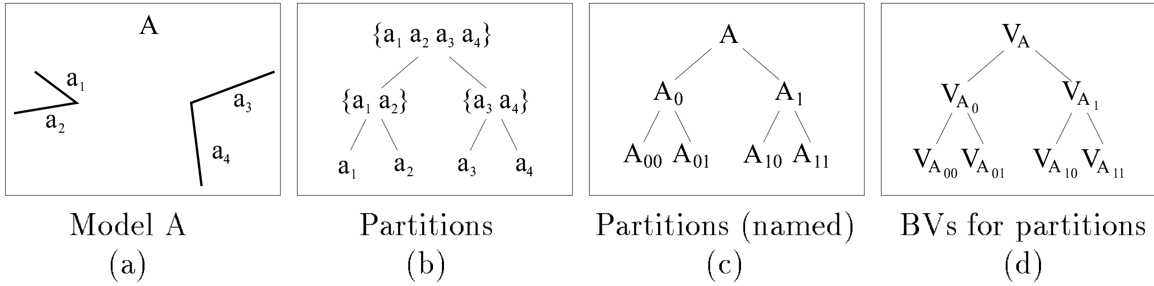


Figure 2.1: A model and a symbolic presentation of its bounding volume hierarchy. Part (a) is the model, consisting of four primitives (in this 2D example, the primitives are line segments). Part (b) shows a recursive partitioning of the primitives, while part (c) assigns a name to each partition. Part (d) assigns a name to the bounding volumes of the partitions, and this is the symbolic presentation of the bounding volume hierarchy of model A.

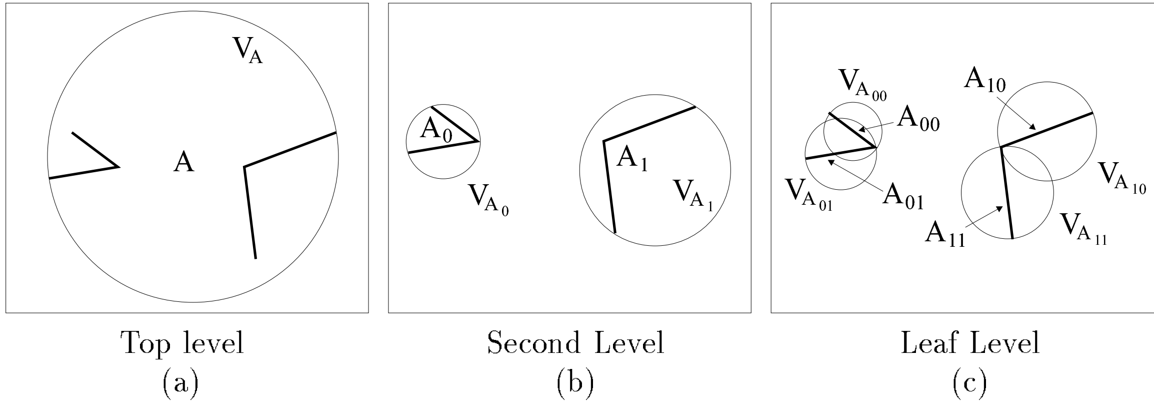
topology.

2.1.1 Bounding Volume Hierarchies

Consider the model consisting of four line segments named a_1 through a_4, as shown in Figure 2.1(a). Notice that the model is not completely connected; it has two disconnected pieces. The model can be recursively partitioned into a hierarchy, shown in Figure 2.1(b), such that the top-level node is the entire model, the leaf nodes are the individual primitive elements (line segments, in this case), and each node equals the union of its children. The example shown in Figure 2.1(b) is just one of many ways to partition the set of primitives.

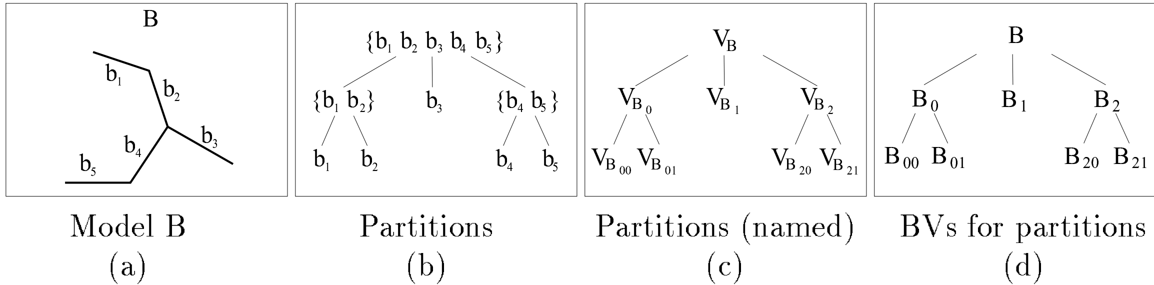
These nodes can be labeled as shown in Figure 2.1(c). The top-level node is the symbol for the whole model, the name of any first child is the name of the parent with a "0" subscript appended, and the name of any second child has a "1" subscript appended. Thus, the sequence of "0"s and "1"s describes a path of "left" and "right" choices, starting at the root, which leads to the given node. This naming convention extends to any number of children, using the digits "2," "3,", etc. These nodes are named subsets of the original model.

The sets A, A_0, A_1, A_{00}, A_{01}, and so forth can be assigned bounding spheres. It is not necessary for the assigned sphere to be of minimal size, but smaller spheres generally yield better collision query execution times. The bounding volume for a point set A_x is given the symbolic name V_{A_x}. Thus the bounding spheres form their own hierarchy, shown schematically in Figure 2.1(d). Graphically, we can draw the



Top level (a) Second Level (b) Leaf Level (c)

Figure 2.2: Part (a) shows the top-level bounding volume, named V_A, which is the root node of the bounding volume hierarchy. Part (b) shows the next level of the BVH, consisting of the BVs V_{A_0} and V_{A_1}. Part (c) shows the leaf nodes of the BVH.



Model B (a) Partitions (b) Partitions (named) (c) BVs for partitions (d)

Figure 2.3: Model B, consisting of 5 primitives. Part (a) shows the model graphically, while parts (b) through (d) show the recursive partitioning, the names for those partitions, and the names for the bounding volumes for those partitions, respectively. Notice that this partitioning has a 3-way branch.

original model A along with each level of the bounding volume hierarchy. Figure 2.2(a) shows A with V_A, the sphere covering the entire model. Figure 2.2(b) shows A_0 and A_1 with their bounding spheres. Finally, Figure 2.2(c) shows the leaf nodes of the hierarchy and the model primitives they enclose.

Now consider model B, shown in Figure 2.3(a). The partitioning for model B, shown in Figure 2.3(b), includes a three-way branch. BVHs are not required to be binary trees, although this restriction simplifies implementation. The partitions are given names in manner similar to that of model A, shown in Figure 2.3(c), and the structure of the bounding volume hierarchy is shown in Figure 2.3(d). The levels of the bounding volume hierarchy are shown graphically in Figure 2.4. Note that the subset B_1 and its bounding volume are second level nodes as well as leaf nodes in their respective hierarchies.

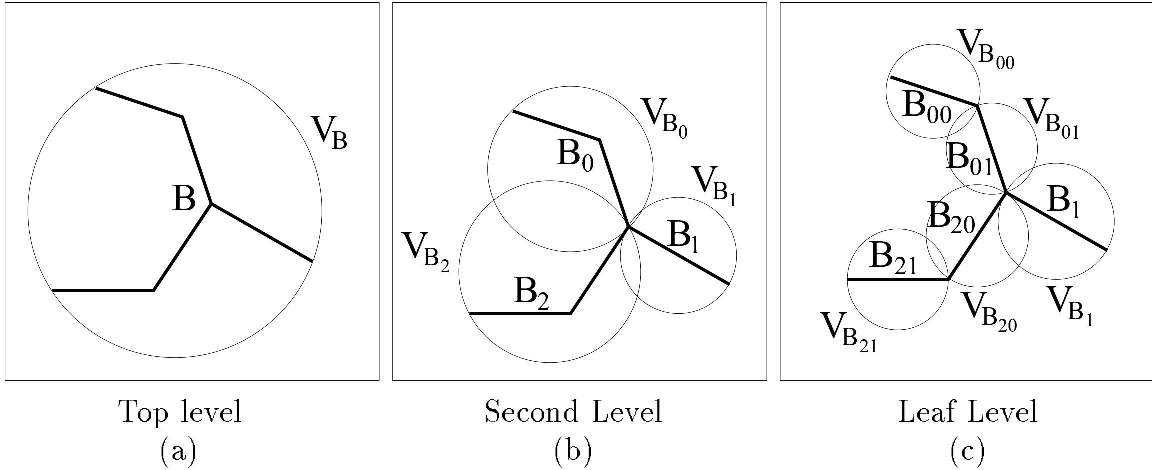


Figure 2.4: Graphical display of the BVH for model B. Part (a) shows the top level node, part (b) shows the three second level nodes, and part (c) shows the leaf nodes. Notice that node V_{B_1} is a second level leaf node.

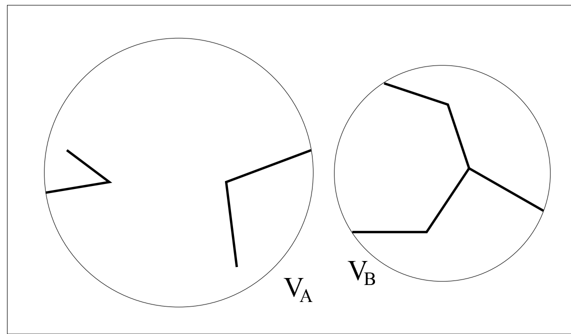


Figure 2.5: The models juxtaposed – bounded apart by their top level spheres.

2.1.2 A Collision Query

Given the two models and their placements in the world space, the simplest approach to perform a collision query is to test each of the four line segments in A against each of the five line segments in B, requiring 20 pairwise primitive overlap tests. While this approach is reasonable for relatively small models such as these, we cannot perform exhaustive pairwise testing on models which have millions of primitives. Instead, we perform a trivial (but conservative) test by checking the top-level bounding volumes for overlap. In the case shown in Figure 2.5, the bounding volumes V_A and V_B do not touch, and consequently the enclosed models cannot possibly be in contact. Technically, we say that *V_A and V_B bound the models A and B apart.*

If the models are a little closer together, as shown in Figure 2.6(a), the top-level spheres touch and more work is needed. There are several choices for our next step,

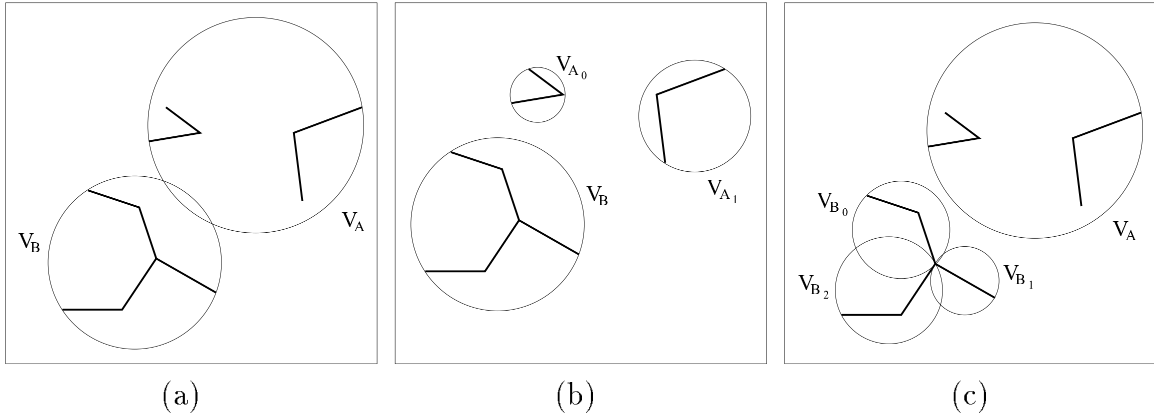


Figure 2.6: A scenario in which the models do not touch, but the top-level BVs do not bound the models apart. Part (a) shows the models and the overlapping top level BVs, V_A and V_B. Part (b) shows the children of V_A being tested against V_B, and since there are no overlaps, the models are known to be disjoint. Part (c) shows an alternative way to show that the models are disjoint: testing the children of V_B against V_A finds no overlaps.

but in this example, we will choose to test the children of V_A (which are V_{A_0} and V_{A_1}) against V_B, as is shown in Figure 2.6(b). In our terminology, we say that we *descend* A, to move from a comparison between the pair (V_A, V_B) to two comparisons, between the pair (V_{A_0}, V_B), and between the pair (V_{A_1}, V_B). Since V_{A_0} doesn't touch V_B, the point set A_0 doesn't touch the point set B, and likewise for A_1 and B. Since A_0 and A_1 together make up the entire model A, there is no contact between A and B. We could have chosen to test the children of V_B (note that there are three of them) against A, as is shown in Figure 2.6(c), with the same result that A and B are disjoint.

When the models are actually in contact, as shown in Figure 2.7(a), we will not be able to use BV tests to bound the models entirely apart. However, the BV tests still enable us to eliminate whole groups of potential contacts from consideration. In Figure 2.7(a), the top-level bounding spheres overlap, so we descend A. The two new problems are shown in Figure 2.7 parts (b) and (c). We have just divided the overall collision detection problem (between A and B) into two smaller problems: the first is between A_0 and B, and the second is between A_1 and B. If we call $\text{cont}(A, B)$ the set of contacts between A and B, then

$$\text{cont}(A, B) = \text{cont}(A_0, B) \bigcup \text{cont}(A_1, B)$$

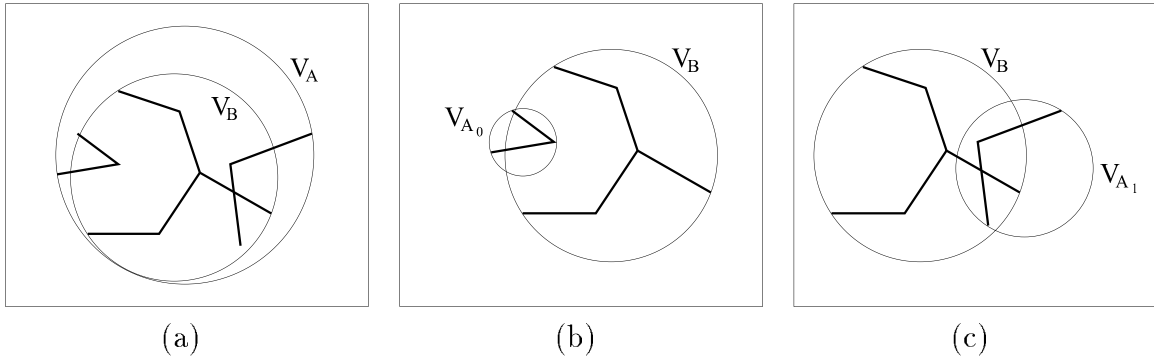


Figure 2.7: The models touch, and therefore the top level bounding volumes must also touch.

From this point, we could solve the two sub-problems independently, and merge their results for the final answer. This formula can applied recursively, and is the basis for the divide-and-conquer approach used in BVH-based collision detection.

In both sub-problems, we have overlaps between the BVs, so with each pair we descend B, to test the children of V_B against the children of V_A. Since V_B has three children, each one of our sub-problems leads to three new sub-problems, which are shown in Figure 2.8. Parts (a1), (a2), and (a3) show the sub-problems arising from the (V_{A_0}, V_B) pair, and parts (b1), (b2), and (b3) show the sub-problems arising from the (V_{A_1}, V_B) pair. Three of these six sub-problems have disjoint bounding spheres, and therefore contribute no contacts to the global solution. The other three sub-problems, parts (a1), (a3), and (b2), have overlapping bounding spheres, and require more work.

Part (a1) concerns the pair (V_{A_0}, V_{B_0}). Descending B from here yields two more sub-problems, each of which has disjoint bounding spheres, as shown in Figure 2.9(a). Likewise, the pair (V_{A_0}, V_{B_2}) from part (a3) results in two sub-problems when we descend B, and from Figure 2.9(b) we see that the subsets are bounded apart by their respective bounding volumes. For part (b2), from the pair (V_{A_1}, V_{B_1}) we cannot descend B because V_{B_1} has no children – it is a leaf node of the BVH. Therefore we choose to descend A, yielding the two sub-problem pairs $(V_{A_{10}}, V_{B_1})$ and $(V_{A_{11}}, V_{B_1})$, shown separately in Figures 2.9 parts (c) and (d). In each case, the bounding volumes overlap, and we cannot descend any further because both bounding volumes are leaf nodes of their respective BVHs. All that remains is to test the actual line segments for overlap. In the case of the pair $(V_{A_{10}}, V_{B_1})$, the segments are disjoint, and in the case of the pair $(V_{A_{11}}, V_{B_1})$, the segments touch, yielding the only contact between A and B.

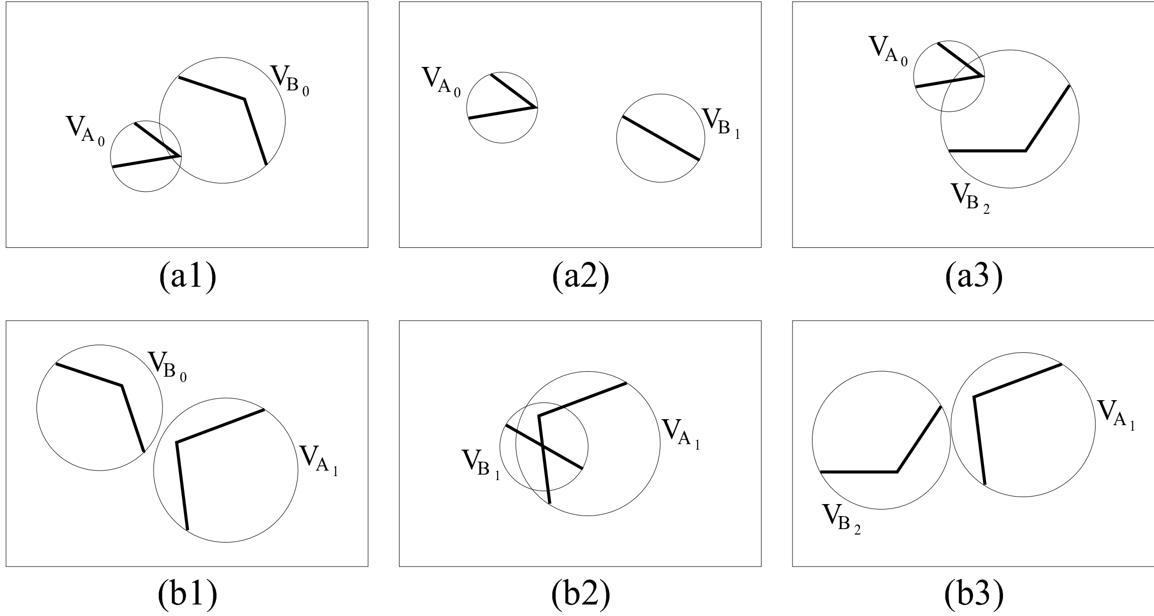


Figure 2.8: For each sub-problem, descending model B yields three more, for a total of six sub-problems. In sub-problem (b3), the bounding spheres are just barely disjoint.

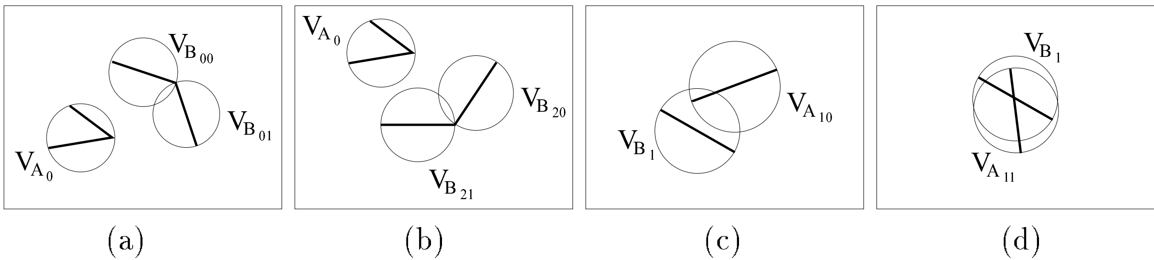


Figure 2.9: Children of V_{B_0} do not touch V_{A_0}.

2.2 Types of Bounding Volumes

There are many types of bounding volumes: spheres, oriented bounding boxes, axis-aligned bounding boxes, convex hulls, ellipsoids, prisms, cubes, and many more. In order to use a BV as part of a collision detection system, we must be able to do two things: fit a BV to any given collection of primitives, and determine whether two given BVs overlap. The two most commonly used BV types are spheres and axis-aligned bounding boxes (AABBs). In this dissertation we mainly use oriented bounding boxes and compare their performance with spheres and AABBs.

Every BV is parameterized by several variables which represent the BV and are used by the algorithms that determine overlap.

2.2.1 Bounding Spheres

A reasonable representation for a sphere is the center point $\mathbf{c}$ and radius r. They specify the region

$$R = \{(x, y, z)^T \quad | \quad (x - \mathbf{c}_x)^2 + (y - \mathbf{c}_y)^2 + (z - \mathbf{c}_z)^2 < r^2\}$$

This requires four parameters.

2.2.2 Axis-Aligned Bounding Boxes

An axis-aligned bounding box (AABB) can be specified several way: as min and max values along each axis: $l_x, l_y, l_z, u_x, u_y, u_z$ which specify the region

$$R = \{(x, y, z) \quad | \quad l_x \leq x \leq u_x,\ l_y \leq y \leq u_y,\ l_z \leq z \leq u_z\}$$

These are lower and upper bounds on the coordinates of the points in the region.

Alternatively, it can be specified as a corner point $\mathbf{c}$ and edge lengths d_x, d_y, and d_z. This is the region

$$R = \{(x, y, z) \quad | \quad \mathbf{c}_x \leq x \leq \mathbf{c}_x + d_x,\ \mathbf{c}_y \leq y \leq \mathbf{c}_y + d_y,\ \mathbf{c}_z \leq z \leq \mathbf{c}_z + d_z\}$$

Or, it could also be specified as a center point $\mathbf{c}$ and edge half-lengths along each axis, r_x, r_y, and r_z. This is the region,

$$R = \{(x, y, z) \quad | \quad |\mathbf{c}_x - x| \leq r_x,\ |\mathbf{c}_y - y| \leq r_y,\ |\mathbf{c}_z - z| \leq r_z\}$$

In all cases, we require six parameters to represent an axis-aligned bounding box.

2.2.3 Oriented Bounding Boxes

An oriented bounding box is a rectanguloid shape, like AABBs, except that it can have arbitrary orientation. In this dissertation, our representation for OBBs will be as a center point $\mathbf{c}$, edge half-lengths, r_1, r_2, and r_3, and an orientation specified as three mutually orthogonal unit vectors $\mathbf{v}^1, \mathbf{v}^2$, and $\mathbf{v}^3$. These vectors are the columns of a 3×3 rotation matrix. The region specified is

$$R = \{\mathbf{c} + ar_1\mathbf{v}^1 + br_2\mathbf{v}^2 + cr_3\mathbf{v}^3 \quad | \quad a, b, c \in [-1, 1]\}$$

This requires fifteen parameters. If the orientation of the OBB is specified instead as a quaternion, we require only ten parameters. Using Euler angles would require only nine parameters total, but that representation is costly, as it requires trigonometric functions to recover the directions of the box axes, which are needed by the fast OBB overlap test.

2.3 Updating Bounding Volumes

BVH-based collision queries employ BV overlap tests, and these tests usually assume that the BVs are specified with respect to the same coordinate system. The BVs are originally built with respect to their local model spaces, and the the BVs of the two models must be transformed into a common coordinate system prior to the BV test. This transformation is called the *BV update*.

2.3.1 Coordinate Systems

There are usually two kinds of coordinate systems to consider when performing a proximity query: *world space* and model space. For polygonal models, the vertices of the polygons are specified with respect to its model coordinate system. When we perform a query between two models, we must specify the placement of the two models in a common coordinate system, which we call the world space. These three coordinate systems are shown in Figure 2.10. The arrows in this diagram are labeled with $[\mathbf{R1}, \mathbf{T1}]$ and $[\mathbf{R2}, \mathbf{T2}]$; these represent the rotation and translation components of the rigid body transformation which takes a point from the model coordinate

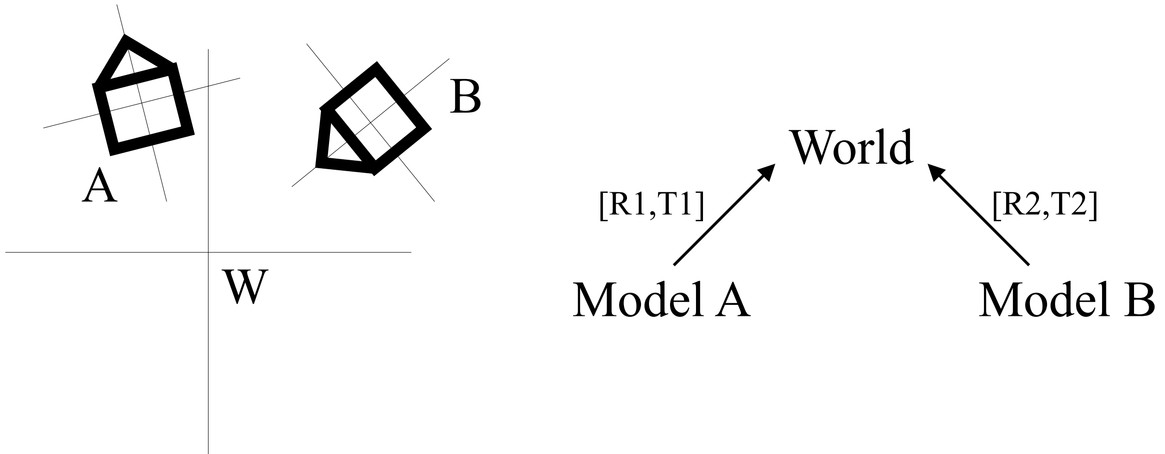


Figure 2.10: The model geometry is specified in terms of the model coordinate system. Queries must specify the placement of the models in the common world coordinate system.

system to the world coordinate system. When the bounding volume hierarchies are built, the bounding volumes are also usually specified with respect to the model coordinate system.

2.3.2 Aligned and Non-Aligned Updates

With regard to updates, there are two types of BVs: aligned and non-aligned. Arbitrary rotations of non-aligned types, such as spheres and oriented bounding boxes, can still be represented precisely, but aligned types, such as axis-aligned bounding boxes, have an orientation restriction. A 45 degree turn about the Z-axis of an AABB yields a box which is no longer axis-aligned, and this volume cannot be exactly covered by any AABB. Following an update, we must guarantee that the new AABB encloses the underlying geometry. There are essentially two approaches to this problem, as shown in Figure 2.11: the first re-examines the geometry to determine an optimal AABB specified with respect to the new coordinate system, and the second approach wraps the transformed AABB with a new AABB. The first approach requires more time, although pre-processing the geometry with a suitable search structure can reduce that time for an increase in memory overhead. The second approach typically yields a looser bounding volume, but the computation can be performed in constant time and requires no extra memory. In this dissertation we have exclusively used the second method whenever we used AABBs.

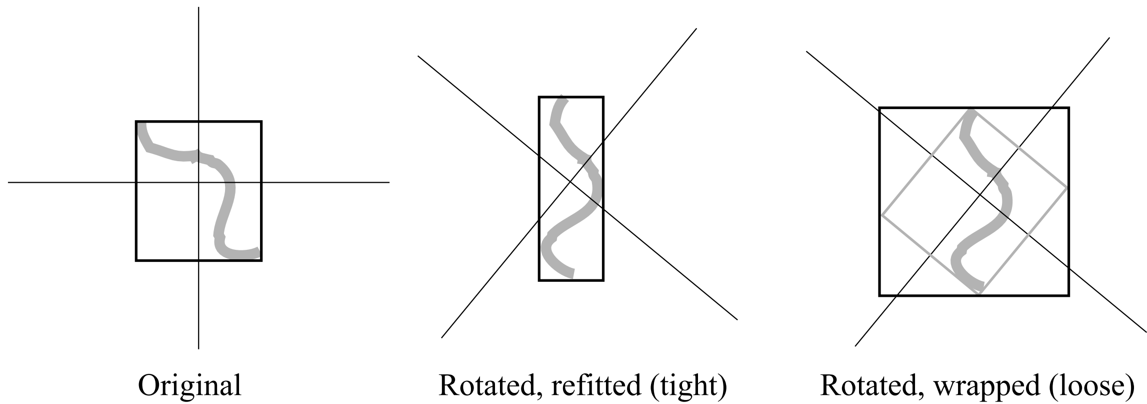


Figure 2.11: An aligned type, such as AABBs, cannot be represented after arbitrary rotation. To maintain a conservative bound, we can either reexamine the geometry to produce a new tightly-fitted aligned box, or with less computation we can cover the rotated box by an aligned one.

2.3.3 Choice of Update Space

Two BVs can be tested for overlap once they are expressed in a common coordinate system. There are typically two approaches to getting the two BVs into a common coordinate system: either transform both BVs into the world space (which we call *world space updates*), or transform one of the BVs into the model space of the other (which we call *model space updates*). Both approaches yield correct results, but each has performance advantages and disadvantages.

When using the world space update strategy, BVs from both models must be transformed, whereas when using the model space update strategy, only one needs to be transformed. Hence, the model space update strategy performs half as many update transformations. In addition to performing less work, the model space update strategy tests potentially tighter BVs, as well, as is shown in Figure 2.12.

2.3.4 Update Caching

A BV is updated just prior to performing the BV test, but the update information is typically thrown away after the BV test is over. Since a given BV is often involved in multiple tests during the course of a query, we may choose to store the updated information to avoid repeating the transformation for later tests. The easiest way to implement this is with additional storage in the BV node structure which holds the updated representation parameters, as well as a flag to indicate whether the cached updated parameters are current. Since the BV representation is typically

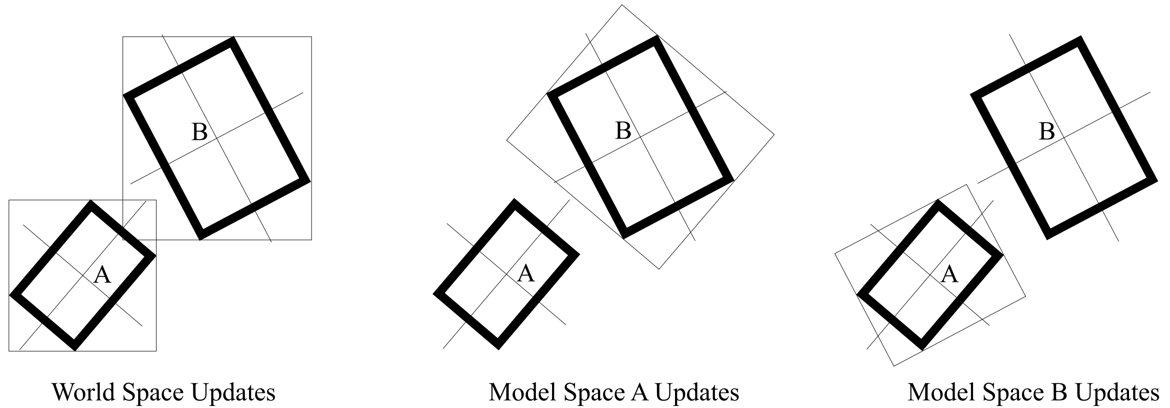


Figure 2.12: With an aligned type such as AABBs, Using world space updates can mean testing loose BVs, whereas using a model space update guarantees that one of the BVs will be tight. In this particular example, the BVs overlap when using world space updates, but not when using either of the model space updates.

the majority of the memory footprint of a BV node, the additional representation for updated parameters increases the storage requirements for the node by 50% to almost 100%, depending on the BV type.

2.4 Tree Traversal Rules

Traversal rules decide which tests to perform next, after finding two overlapping bounding volumes. In the sphere tree example of Section 2.1 we used the "descend largest radius" rule, which means we descend to the children of whichever BV has the largest radius. Two other similar rules are "descend largest volume", "descend largest surface area". Of course, any imaginable rule can be used, such as "descend smallest" using any of these metrics, or "descend model A" or "descend model B". More complex rules can be invented which choose based on some function of the traversal history. Throughout this dissertation, we use the rule "descend largest diameter".

2.5 The Bounding Volume Test Tree

The overlap tests between bounding volumes proceed recursively: after testing two bounding volumes, we then select the children of one of them to test against the other. In this way, a test between a pair of overlapping bounding volumes leads to two new

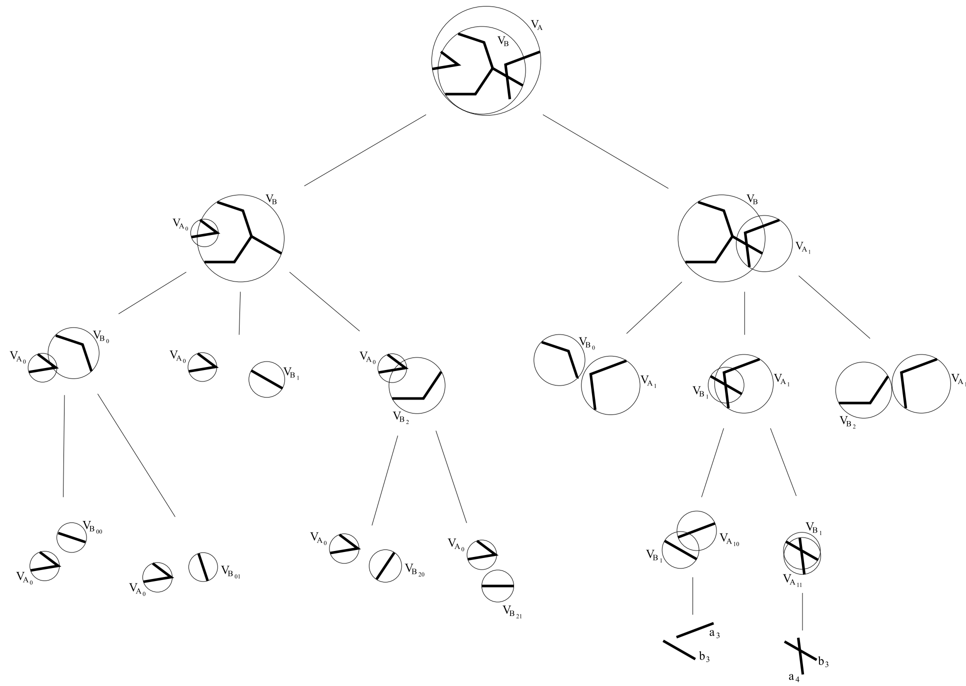


Figure 2.13: Pictorial representation of the bounding volume test tree.

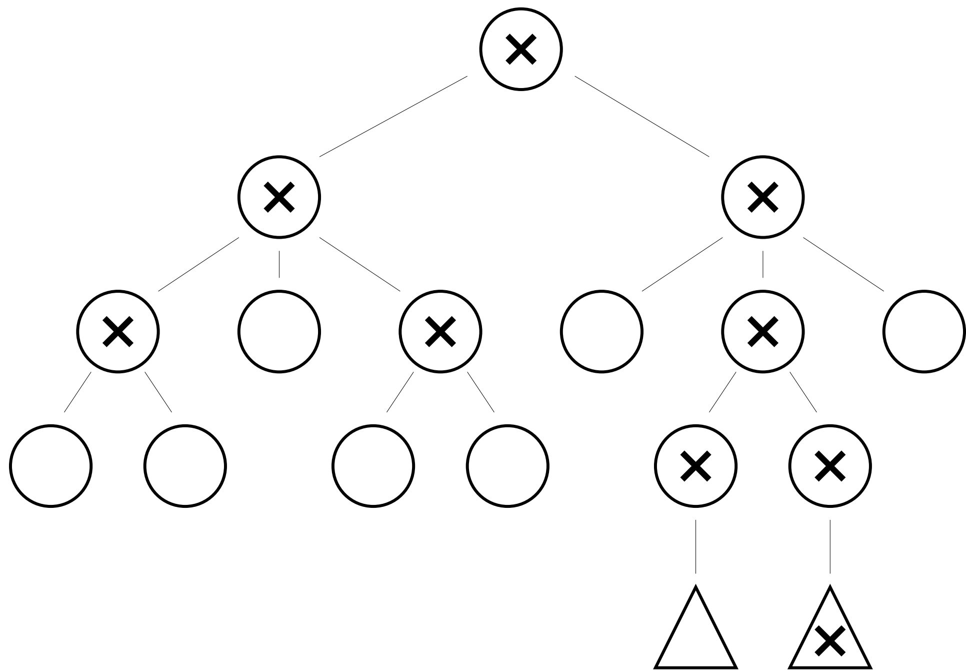


Figure 2.14: Symbolic representation of the bounding volume test tree. Circles represent tests between BVs, and triangles represent tests between polygons. An “X” indicates the items were touching, otherwise they were not touching.

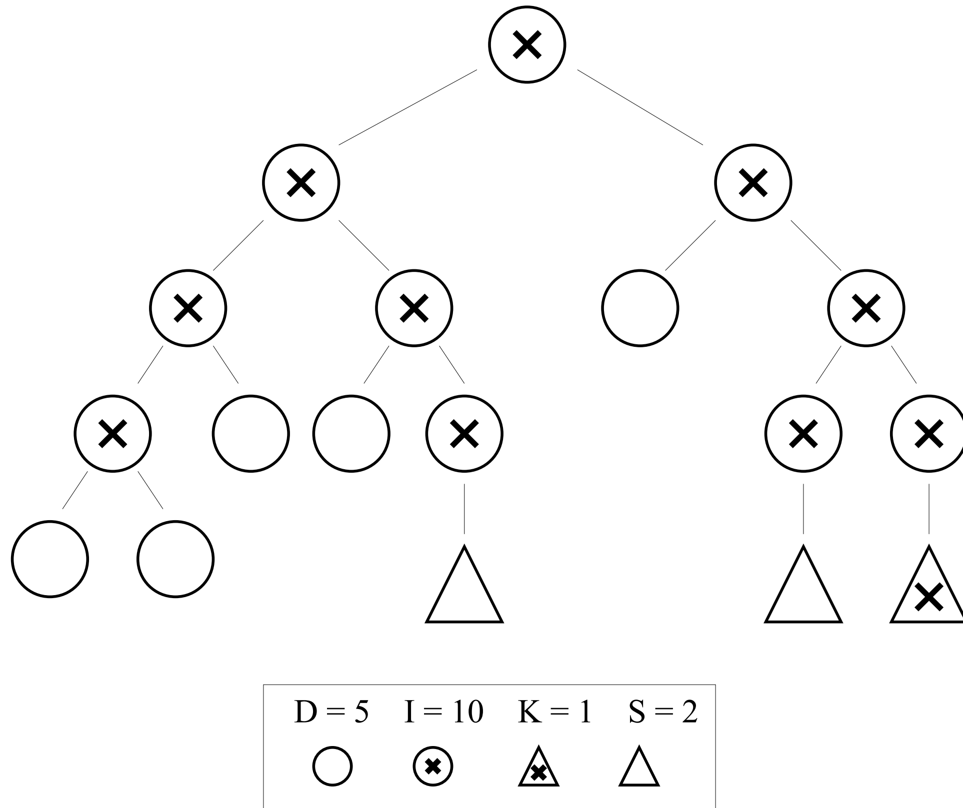


Figure 2.15: The numbers of different types of nodes in any binary BVTT satisfies the equation $1 = D - I + 2S + 2K$, where D and I count disjoint and intersecting BVs, respectively, S counts the number of separated primitives tested, and K counts the number of contacts found.

pairs to test. This sequence of tests can itself be organized into a tree, which is shown pictorially in Figure 2.13 and symbolically in Figure 2.14. The trees represent the same sequence of tests as in the sphere tree example from Section 2.1. A circle represents a BV pair test, and the "X" indicates it yielded intersection. A triangle represents a test between primitives, and an "X" indicates it yielded "contact".

2.5.1 Invariants Applicable to Binary BVTTs

If a BVTT is binary, then it satisfies an invariant which relates to the number of BV tests and primitive tests and their outcomes.

Consider the binary BVTT portrayed graphically in Figure 2.15. We can count $I = 10$ tests among intersecting BVs, and $D = 5$ tests among disjoint BVs. Furthermore, there was $K = 1$ contact found among the primitives, and there were $S = 2$ tests among separated primitives. Thus we have two kinds of tests, each with two

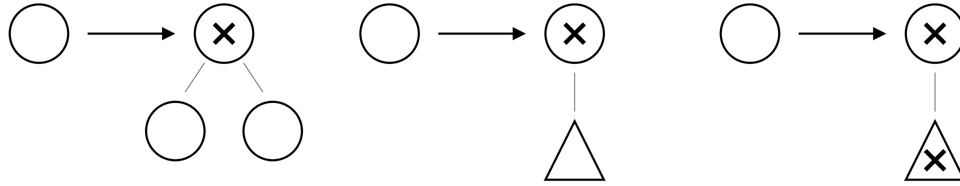


Figure 2.16: A series of substitutions can grow any realizable BVTT from an initial test between disjoint BVs. We prove that all BVTTs satisfy the constraint $1 = D - I + 2S + 2K$ by inducting on the number of substitutions.

possible outcomes. We will refer to these as *I-tests, D-tests, K-tests*, and *S-tests*, respectively. The counts of these four kinds of tests occurring in the BVTT satisfy the constraint

$$1 = D - I + 2S + 2K.$$

We will show that the counts for the various tests for any possible BVTT must also satisfy this constraint. We begin with a single root node representing disjoint BVs – this is the BVTT resulting from widely separated models, which is resolved with a single top-level test. The counts for this BVTT are given in the 4-tuple $(D, I, S, K) = (1, 0, 0, 0)$, and it satisfies the constraint.

We now induct on the number of transformations applied to this initial trivial tree. After each step, we have a valid BVTT, and all valid BVTTs can be generated after a suitable number of steps. A single D-test test can be subjected to one of three transformations, shown in Figure 2.16. It can be replaced by an I-test followed by two D-tests, or by an I-test followed by an S-test, or by an I-test followed by a K-test. The first transformation results in a net addition of an I-test and a D-test, changing the 4-tuple by $(+1, +1, 0, 0)$. The second transformation eliminates one D-test and adds one I-test and one S-test, changing the 4-tuple by $(-1, +1, +1, 0)$. The third transformation eliminates one D-test and adds one I-test and one K-test, changing the 4-tuple by $(-1, +1, 0, +1)$. If the tree's 4-tuple satisfies the constraint before substitution, then it will satisfy the constraint after any of these substitutions. By applying a series of these substitutions to the initial D-test, we can build any feasible BV tree, and therefore any feasible BVTT satisfies the constraint.

If two models are sufficiently separated so as to be bounded apart only with BV tests, then $K = S = 0$ and the constraint simplifies to

$$D = I + 1.$$

This means that almost exactly half the BV tests result in disjoint and half in intersection, if D and I are large. Even in the event that K and S are large, they are small compared to I and D in most cases, and so I and D occur with an almost 1:1 ratio.

2.5.2 Limit on Speedup due to Trivial Rejection Tests

Suppose we propose to optimize the rejection of disjoint BVs by using layered bounding volumes: a complex BV bounding the geometry, and then a simpler BV around the complex BV. The rationale is that we might accelerate a query by testing first with spheres, for example, and then with OBBs only if necessary. If using spheres first enables us to skip many of the OBB tests, then we may have a net gain in speed. With our understanding of the BVTT's structure, we can deduce limits on the speedup attainable by this optimization technique.

Consider the structure of the BVTT for a query which uses OBBs alone. Assume there will be approximately as many I-tests as there are D-tests. Suppose that the sphere test costs no time and catches all the cases where the OBBs are disjoint (which is optimistic both in cost and success rate). This implies that the expense of the D-tests drops to nothing, and the expense of the I-tests remains the same. Note that the number of tests has not changed. The speedup is

$$s = \frac{I\ T_I + D\ T_D}{I\ T_I} = 1 + \frac{D\ T_D}{I\ T_I} \approx 1 + T_D/T_I$$

where D and I are number of D-tests and I-tests, respectively, and T_D and T_I are the typical execution times of the complex BV tests. The final simplification of the speedup formula is a result of the assumption $D \approx I$.

For the efficient OBB test we describe in Chapter 4, T_D is less than T_I – it usually takes slightly less time to decide that two OBBs are disjoint than to decide that two OBBs overlap. This is because the OBB test uses a series of trivial tests, any one of which could determine disjointedness. Consequently, $T_D/T_I < 1$, so the speedup under these ideal conditions is less than two.

Another test method, the Gilbert-Johnson-Keerthi (GJK) algorithm [GJK88] applied to OBBs, has the property that overlapping BVs take slightly less time to process than disjoint BVs. Consequently, $T_D/T_I > 1$, and our upper bound on the speedup is slightly greater than 2.

The assumption that $K = S = 0$ was optimistic. If our BVTT has $K > 0$ or $S > 0$, then the speedup gained is even less, because these tests are not simplified by

the proposed acceleration technique.

2.5.3 Limit on Speedup for Temporal Coherence Scheme

We show another application of the BVTT invariant by deducing an upper bound on the speedup attainable by a certain acceleration technique which evaluates a subset of the nodes in the BVTT. This technique speeds up collision queries in dynamic scenes where there is significant frame-to-frame coherence in the model placements.

The idea is to maintain a list of the leaves of the BVTT generated during a query to serve as a collective *witness* to the separation or contact of two models. Under the coherence assumption, the models would be in similar relative configuration during the next frame, and consequently the BVTT generated for that frame would be similar to the one generated the previous frame. Rather than generating the current frame's BVTT from the root, we can repeat the tests of the leaf nodes of the previous frame's BVTT, but using the models' current positions. If those tests have the same results that they had on the previous frame, then we know that the contact status of the models has not changed.

We will not go into the details, but say simply that the best case for this acceleration technique is one in which the test outcomes for the current frame are identical to those of the previous frame, and no primitives were tested (so $K = S = 0$). In this case, this technique has saved us from performing the tests of the internal BVTT nodes. The speedup so attained is

$$s = \frac{I\ T_I + D\ T_D}{D\ T_D} = 1 + \frac{I\ T_I}{D\ T_D} \approx 1 + T_I/T_D$$

For spheres, $T_I \approx T_D$ since the overlap test has no early exits, and we would expect no better than a factor of two speedup. As stated in the previous section, for our fastest OBB overlap test we have T_I slightly greater than T_D, so this could yield a slightly more significant speedup.

2.6 Contact Pair Matrix

Between a model A of m polygons and a model B of n polygons, there are potentially mn contact pairs, as shown in the matrix in Figure 2.17. This is the contact pair matrix (CPM). One can compare each polygon in A to each polygon in B, testing for overlap – essentially visiting every element in the matrix and filling in a "–" or an "X"

		B						
		b_1	b_2	b_3	b_4	------		b_n
	a_1	-	X	-	-	-	-	-
	a_2	-	-	-	X	-	-	-
	a_3	-	-	X	-	-	-	-
A	a_4	-	-	-	-	X	-	X
	¦	-	-	-	-	-	-	-
	¦	-	X	-	-	-	-	-
	a_m	-	-	-	X	-	-	-

Figure 2.17: The contact pair matrix (CPM) cross indexes all the primitives of two models, showing which pairs are in contact with an "X".

for "separated" and "contact", respectively. Generally, we expect there to be many fewer than mn contact pairs. Thus, we expect the contact pair matrix to be very sparsely populated with "X"s. Of course, stepping through the matrix and testing each pair sequentially is $O(mn)$ work. BVH-based collision detection algorithms use BV overlap tests to verify that entire blocks are empty of contacts.

We will use the contact pair matrix to show that BVH-based collision algorithms find all possible contacts regardless of what traversal rules are used. We will also use it to prove lower bounds on how many BV tests are required to locate a given number of contacts.

2.6.1 Correctness of Tandem Tree Traversal

From our example in Section 2.1, at each recursive step we chose to test one BV (of either A or B) against the children of the other according to the "descend largest diameter" traversal rule. Sometimes we descend one BVH, and sometimes we descend the other, depending on the conditions at each step. In this manner, we gradually descend both trees in tandem. In this section we show that a collision query is a progressive partitioning of the contact pair matrix, that the traversal rule determines the choice of partitions, and that contacts cannot be missed regardless of the choices made.

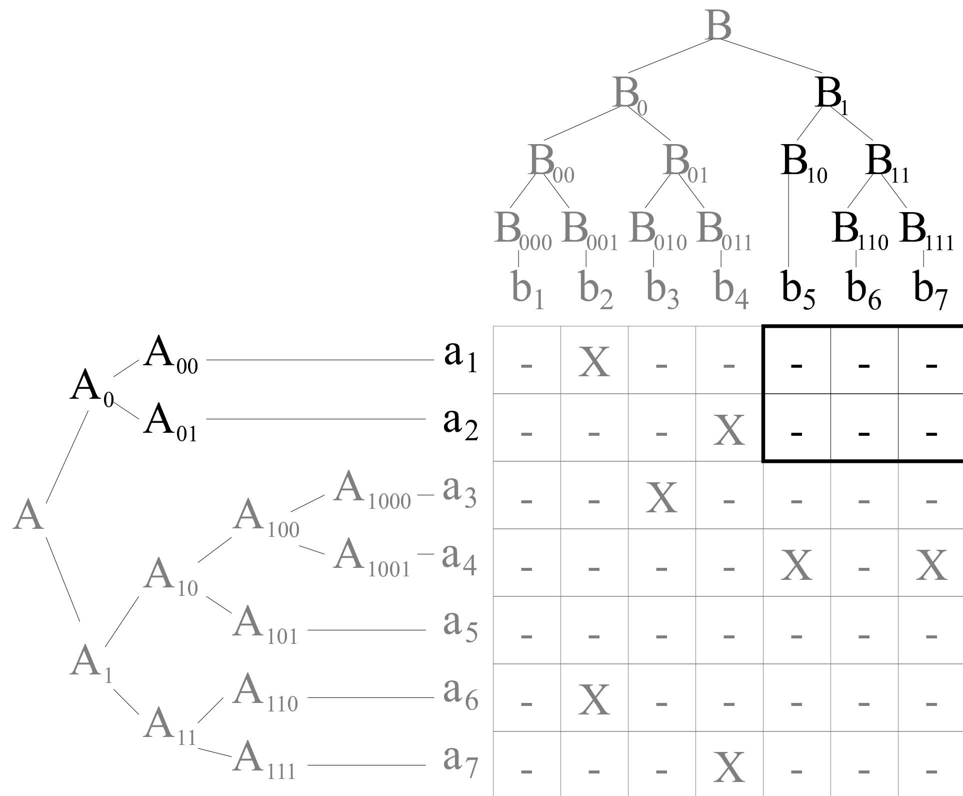


Figure 2.18: Testing two BVs corresponds to examining a sub-block of the contact pair matrix. If the BV test yields disjoint, then the sub-block is known to be empty of contacts. The converse is not true, since the BV tests are conservative – the BV tests can yield intersection even though the sub-block contains no contacts.

Consider that each model possesses a BVH, and let us assume that the primitives have the same ordering as the BVH leaf nodes as visited by a standard pre-order traversal of the BVH. This is shown in Figure 2.18. Now suppose that we have tested V_{A_0} against V_{B_1} and found them to be disjoint. This means that none of the primitives covered by V_{A_0} can be touching any of the primitives covered by V_{B_1}, which implies that the corresponding 2×3 sub-block of the CPM must be filled with "–"s. The converse does not hold: even if the block is empty of contacts, the corresponding test between the BVs may yield overlap.

Recall the sequence of tests from our sphere tree example from Section 2.1 in terms of the matrix. Model A has 4 primitives, and model B has 5, so the matrix has 4 rows and 5 columns, as shown in Figure 2.19. The one contact is recorded within the matrix as being between a_4 and b_3, as shown graphically in Figure 2.7(a). The series of tests are reflected in the repeated subdivisions of the matrix, as shown in Figure 2.20. The first test is between V_A and V_B, which is the entire matrix, as

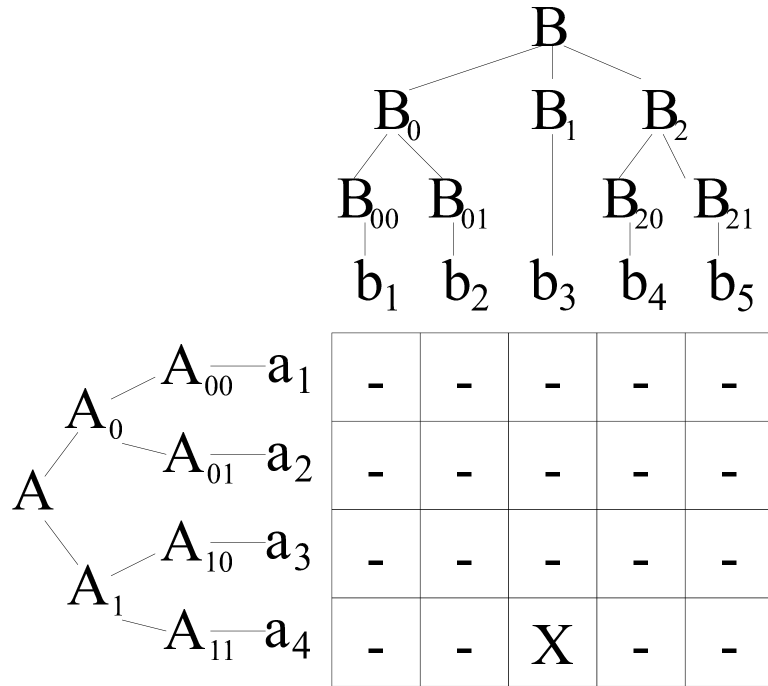


Figure 2.19: This is the contact pair matrix for the sphere tree example. The single contact between a_4 and b_3 is shown.

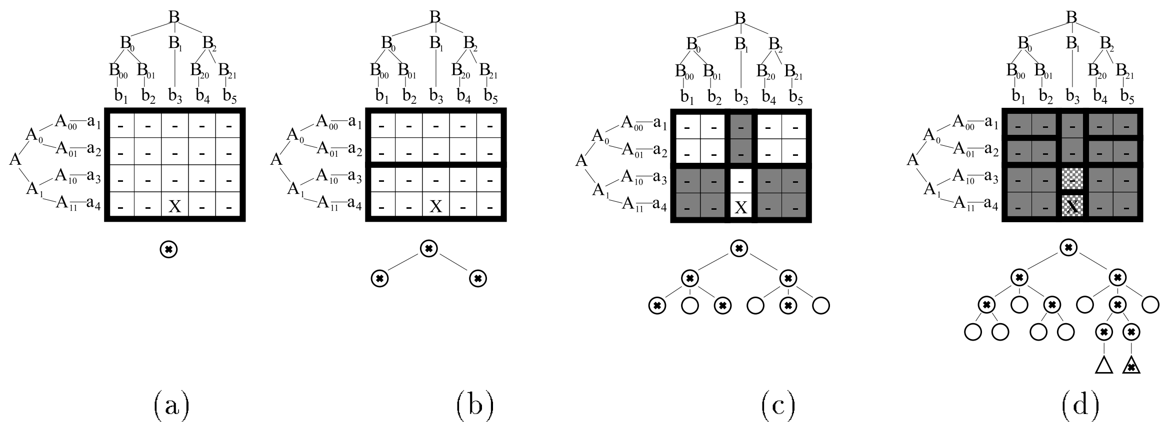


Figure 2.20: This is the sequence of subdivisions for the sphere tree example. Each stage of the subdivision is accompanied by the corresponding BVTT.

shown in Figure 2.20(a). These BVs are overlapping, and the test leads to two new pairs, (V_{A_0}, V_B), and (V_{A_1}, V_B), shown in Figure 2.20(b) as two blocks which together cover the entire matrix. Each of these tests yields intersection, and leads to three new tests between BVs, shown in Figure 2.20(c). The three blocks which are shaded are for the pairs found to be disjoint, implying that those blocks will not be further subdivided. The upper left and upper right blocks each yield two new tests between BVs, all of which are found to be disjoint. The lower middle block yields two new tests which are both found to be overlapping. This next generation of tests is shown in Figure 2.20(d). The tests between disjoint BV pairs are shaded solid gray. The last two tests are between BVH leaf nodes, and so cannot lead to further BV tests. Instead, these lead to tests between primitives, which is indicated by filling the cells with a checker pattern. Each block is labeled "X" if the primitives were in contact, and "–" if they were separate.

The series of BV overlap tests is a progressive (depth-first) partitioning of the matrix. It should be evident that no contacts can be missed, regardless of the choices we make during tandem tree traversal, since no partitioning can divide the matrix in such a way as to exclude any cell. Furthermore, there is no danger of prematurely terminating a recursion branch that contains a contact, since the test corresponding to a block containing a contact must result in an intersection.

2.6.2 Ideal BVs and Lower Bounds on BV Tests

In this section we derive a lower bound on the number of BV tests required to locate $k > 0$ contacts between two models of n primitives each. To help prove this bound, we introduce the concept of "ideal bounding volumes."

An ideal BV is a hypothetical bounding volume that covers its geometry so tightly that the BV overlap test yields disjoint if and only if the covered geometry is disjoint. The ideal BV outperforms all other types in its ability to prune the BVTT. Consequently, a lower bound on the number of tests between ideal BVs would also be a lower bound on the number of tests required between ordinary BVs, such as spheres or OBBs.

2.6.2.1 Ideal BV tests required to locate a single contact

Suppose the CPM contained exactly one contact, and that the BVHs for the models were perfectly balanced binary trees of ideal BVs. Perfectly balanced binary trees

means that the models each have $n = 2^m$ primitives, with m a positive integer. The CPM has n^2 cells, and the number of BV tests required to locate that one contact is $2\log_2(n^2) + 1$.

This is proved as follows. We start with the entire CPM, containing n^2 cells, one of which is a contact. The first subdivision yields two blocks, each containing $n^2/2$ cells. One of the blocks contains the contact, and the other does not. Each of these blocks represents a BV test; the one containing the contact yields intersecting, and the other yields disjoint, because the BVs are ideal. The block containing the contact is further subdivided into two blocks containing $n^2/4$ cells each. The series of tests and subdivisions continue until a subdivision yields two blocks containing one cell each, only one of which has the contact. The number of subdivisions is $s = \log_2 n^2$. Every subdivision leads to two blocks and their associated tests. This accounts for every test except the very first, which preceded the first subdivision. Thus, there are $2s + 1 = 2\log_2 n^2 + 1$ BV tests to locate the single contact in a block containing n^2 cells. This result can also be applied to sub-blocks containing a single contact.

2.6.2.2 Ideal BV tests required to locate n^2 contacts

If the CPM were entirely filled with contacts, then we would require $2n^2 - 1$ BV tests to identify all the contacts. This is most easily seen by drawing out the BVTT. Since the BVTT must have a leaf node for every contact, which is n^2, it must have $n^2 - 1$ internal nodes, each of which corresponds to a subdivision. The number of BV tests is therefore $2s + 1 = 2n^2 - 1$. In the case of the fully-filled CPM, the results would be the same using whether ideal BVs or real BVs. This result can also be applied to fully-filled sub-blocks of the CPM.

2.6.3 Lower Bounds on BV Tests

We will use ideal BVs and the results of the previous subsection to deduce lower and upper bounds on the number of ideal BV tests required to locate k contacts in a CPM containing n^2 cells. The lower bound on the number of ideal BV tests is also a lower bound on the number of ordinary BV tests.

Suppose we have two models, each with n primitives. Assume that the BVHs for these models are perfectly balanced using ideal BVs. We will use the CPM to show that if the models are touching and there are exactly $k > 0$ contacts, then the number

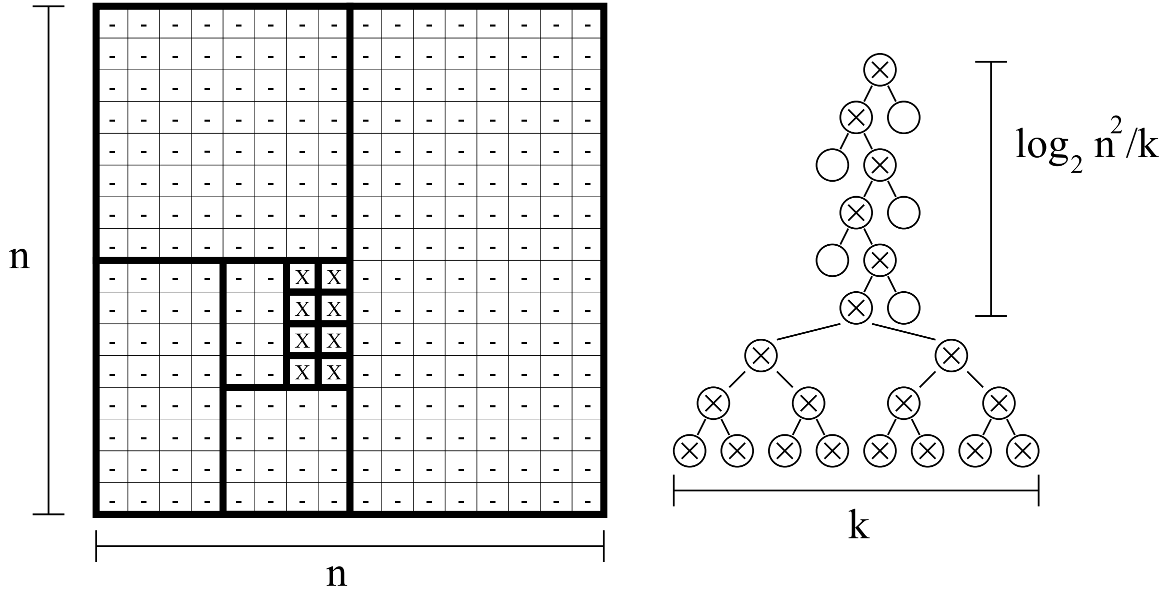


Figure 2.21: The most favorable arrangement of $k > 0$ contacts among n^2 cells is as a block which can be isolated in the minimal number of subdivisions, $\log_2(n^2/k)$. Once the block is isolated, $k - 1$ additional subdivisions are required to isolate all k contacts. The profile of the BVTT is a long thin stalk $\log_2(n^2/k)$ levels long, and terminating in a fully branched subtree of k leaf nodes.

of BV tests required to identify all k contacts is at least

$$v_l(n, k) = 2\log_2(n^2/k) + 2k - 1$$

and at most

$$v_u(n, k) = 2k\log_2(n^2/k) + 2k - 1$$

The lower bound on the number of ideal BV tests, $v_l(n, k)$ will also establish a lower bound on the number of real BV tests required.

The lower bound assumes a favorable arrangement of the contacts within the CPM. The most favorable arrangement is where they exactly fill one sub-block of the CPM corresponding to the test between two BVs. This sub-block would have $k > 0$ cells. There would be n^2/k other sub-blocks of the same size, all devoid of contacts. The minimum number of subdivisions required to locate the filled sub-block among the other n^2/k sub-blocks is $\log_2(n^2/k)$. Having isolated this sub-block of k cells, we require $k - 1$ additional subdivisions to identify each of the contacts. This adds up to $s = \log_2(n^2/k) + k - 1$ subdivisions, which corresponds to $2s + 1 = 2\log_2(n^2/k) + 2k - 1$ BV tests.

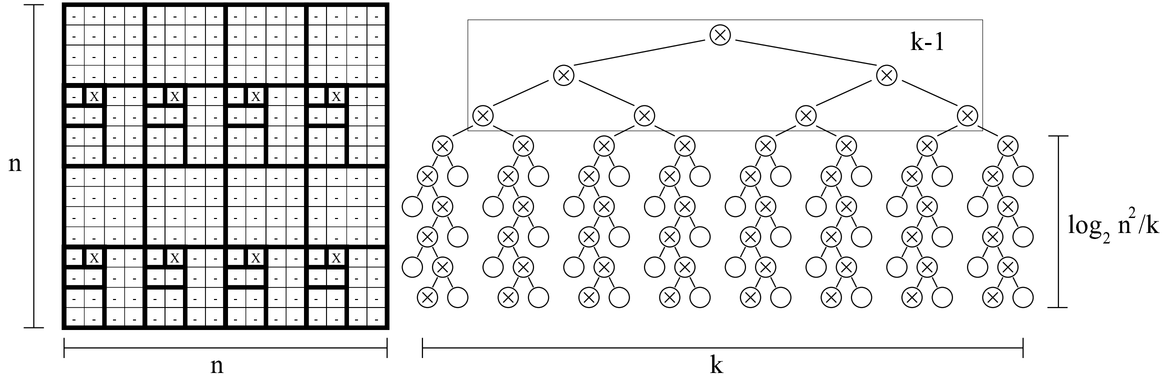


Figure 2.22: The least favorable arrangement of $k > 0$ contacts among n^2 cells is distributed such that the topmost $k-1$ subdivisions have a contact on each side. After partitioning the CPM into k blocks containing one contact each, each block requires $\log_2(n^2/k)$ more subdivisions to locate the contact. The profile of the BVTT is a fully branched subtree with k stalks hanging from its leaves, each stalk being $\log_2(n^2/k)$ levels long.

The upper bound assumes a least favorable arrangement where each of the $k > 0$ contacts sits in its own sub-block of n^2/k cells. The topmost $k-1$ subdivisions could partition the CPM into these k cells, and for each we require $\log_2(n^2/k)$ additional subdivisions to locate the contact. This is a total of $s = k - 1 + k\log_2(n^2/k)$ subdivisions, corresponding to $2s + 1 = 2k\log(n^2/k) + 2k - 1$ BV tests.

The v_u BV tests represent worst case performance for ideal BVs, and therefore represent a lower bound on the worst case performance for ordinary BVs. The v_l represent best case performance for ideal BVs, and therefore represent a lower bound on performance for all BVs. Because the assumptions of ideal BVs and balanced BV trees were optimistic, no combination of BV type, tree building strategy, and tree descent mechanism can guarantee better performance than v_l using binary BVHs.

2.6.4 Summary of CPM Result

We introduced the contact pair matrix as a tool for analyzing the tandem tree traversal process. The most important result we obtained was that the number of BV tests required to find $k > 0$ contacts between two models of n primitives each would be at least $v_l(n,k) = \log_2(n^2/k) + k - 1$. We did this by postulating the existence of an "ideal bounding volume" which outperformed all real BVs, and determined that this was the lower bound on the number of ideal BV tests required.

2.7 Summary of Chapter

The primary results of this chapter are in Sections 2.5 and 2.6, where we examine the structure of the bounding volume test tree (BVTT) and the contact pair matrix (CPM), respectively. These analytic tools help illustrate and reason about the query process. Using the BVTT we derive upper bounds on the speedup attainable by optimization techniques such as trivial rejection tests for overlap (Section 2.5.2) and temporal coherence in tree traversal (Section 2.5.3). Furthermore, we use the CPM to derive a lower bound on the number of BV tests required to locate the k contacts among two touching models of n primitives each (Section 2.6.3).

Chapter 3

Fitting an OBB

The principal of this chapter is a method for fitting an OBB to a collection of polygons in space. Our technique examines the statistical spread of the polygons to determine a suitable orientation for the box. We describe inputs which cause a simple application of the technique to produce arbitrarily bad orientations, and we address this issue by using convex hulls to lessen the impact of irregular triangle distributions.

Section 3.1 discusses in abstract terms the three common approaches to building BVHs: top-down, bottom-up, and incremental insertion. Section 3.2 explains how the size and position of an OBB is determined once its orientation has been chosen. Section 3.3 discussess the specifics of how to choose a suitable orientation for an OBB. Section 3.4 presents the formulas for the covariance matrix required in section Section 3.3. Section 3.5 discusses the optimality of OBBs produced by this method.

3.1 Building a Bounding Volume Hierarchy

There are three basic approaches to building a bounding volume hierarchy (BVH): top-down, bottom-up, and incremental insertion. Each of these methods have been explored in the literature, but no consensus has been reached concerning which methods produce superior trees. In Chapter 5, where we present results of benchmark tests, all the trees used were produced using a top-down approach.

All of the methods require some means of fitting a BV to a collection of polygons. The method for fitting AABBs is trivial, and minimal volume spheres are discussed elsewhere in the literature. The a range of methods for fitting OBBs is discussed in Section 3.3.

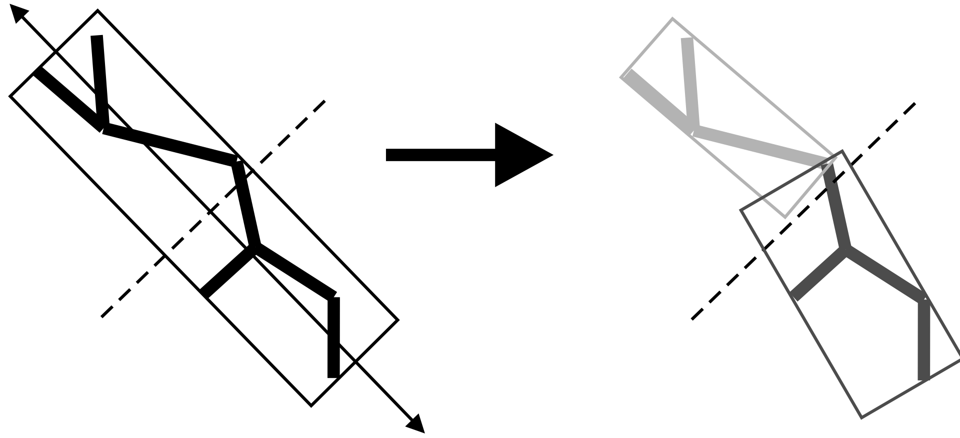


Figure 3.1: Building a bounding volume hierarchy with the top-down split-and-fit technique. The primitives are partitioned according to which side of a dividing plane they lie on, and the two new groups are each fitted with their own bounding volumes. Our choice of dividing plane is one which is orthogonal to the direction of greatest spread, and which passes through the centroid of the group of primitives.

3.1.1 Top-Down

Top-down approaches are probably the most commonly used in practice. They are easily implemented as a recursive application of "fit-and-split" operations: given a collection of polygons, fit a BV to them, and then partition the collection into two (or more) groups. To each group, fit a new BV, and partition again. There are many variations on the method, mainly due to choices of how to partition the polygons. Assuming reasonably balanced partitions, the BVH has depth $O(\log_2 n)$ for n polygons. One common approach to partitioning is to choose a suitable dividing plane, and to partition the polygons according to which side of the plane their centroids fall.

Unless some search structure has been applied to the geometry beforehand, the partitioning process is $O(k)$ for k polygons. If the fitting process is also $O(k)$ for k polygons, then the tree building process has the same recurrence relation as quicksort – expected $O(n \log n)$ if there is reasonable balance, but $O(n^2)$ in the worst case. If the fitting process is $O(n \log n)$ – such as when constructing the convex hull – then the top-down process will take $O(n \log^2 n)$ time.

The approach we used in our benchmark implementations was to choose an axis of greatest spread – determined by the covariance of the surfaces of the polygons (see Sections 3.3 and 3.4.3) – and then to choose a dividing plane orthogonal to that axis which also passed through the centroid of the geometry to be partitioned. One step of this recursive process is shown in Figure 3.1.

3.1.2 Bottom-Up

Another way to build a tree is to fit every polygon with its own BV, and then to perform successive merges. This is the "knit-and-fit" approach to BVH construction. Each merge creates a parent node from two smaller ones, and the tree is complete when all the nodes have been gathered into the hierarchy. It requires $n-1$ merges for n polygons. After each merge, a new bounding volume must be created for the group of k polygons. This new bounding volume can be created in $O(1)$ time if it bounds the BVs of the two child groups, in $O(k)$ time if it examines each of the k polygons, or perhaps in $O(k \log k)$ time if it performs a sophisticated operation such as finding extremal points among the polygons. The total BVH build time would be $O(n)$, $O(n \log n)$, and $O(n \log^2 n)$ using these three fitting times, respectively. This assumes $O(1)$ time to select the next pair to merge.

Sometimes octree subdivision is regarded as a bottom-up approach, but octree subdivision can also be viewed equally well as a top-down approach. The "splits" or "knits", depending on one's viewpoint, are predetermined by the placement of the top-level node of the octree.

3.1.3 Incremental Insertion

Incremental insertion adds primitives to the BVH one at a time until all the primitives have been incorporated into the tree. The insertion can proceed as a traversal from the root down to a leaf, making branching decisions *en route*, and enlarging BVs as necessary as it descends. Alternatively, the new primitive can be merged to some nearby leaf node, and the addition causes a wave of enlargements (if necessary) to propagate upward toward the root.

3.2 Computing the OBB Parameters

Our basic approach to fitting an OBB to a model (or to a portion of a model) is first to choose an orientation for the OBB, and then to choose a center and minimal edge lengths that enable it to cover the model. The difficult part is choosing an orientation – the rest follows easily, as we show.

Suppose we have selected an orientation for our OBB. Let the three vectors $\mathbf{v}^1$, $\mathbf{v}^2$, and $\mathbf{v}^3$ be aligned with the face normals of the OBB. Also, let $\mathbf{p}^k$ be the kth vertex of the model being fitted (k ranges from 1 to n).

Now, project all the vertices of the model, $\mathbf{p}^k$, onto each of the vectors. The upper and lower extremes along each axis is given by

$$\begin{aligned}
u^1 &= \max_k(\mathbf{v}^1 \cdot \mathbf{p}^k) \\
u^2 &= \max_k(\mathbf{v}^2 \cdot \mathbf{p}^k) \\
u^3 &= \max_k(\mathbf{v}^3 \cdot \mathbf{p}^k) \\
l^1 &= \min_k(\mathbf{v}^1 \cdot \mathbf{p}^k) \\
l^2 &= \min_k(\mathbf{v}^2 \cdot \mathbf{p}^k) \\
l^3 &= \min_k(\mathbf{v}^3 \cdot \mathbf{p}^k)
\end{aligned}$$

The ith axis of the OBB is aligned with $\mathbf{v}^i$, and the OBB's width along this axis will be given by $u^i - l^i$ (but note that our preferred representation stores the half-widths). Finally, the center point $\mathbf{c}$ for the OBB is given by

$$\mathbf{c} = \frac{1}{2}(l^1 + u^1)\mathbf{v}^1 + \frac{1}{2}(l^2 + u^2)\mathbf{v}^2 + \frac{1}{2}(l^3 + u^3)\mathbf{v}^3$$

Observe that $\mathbf{c}$ projects to the midpoint of the range along each axis.

3.3 Covariance-Based Methods

In this section we describe how to choose an orientation for the OBB. We want the OBB to be aligned with the model. If the model is long and thin, like a pencil, then we want an axis of the OBB to be aligned with the long axis of the pencil. If the model is especially thin along just one direction, such as a dinner plate, then we want an axis of the OBB to be aligned normal to the plane of the plate. In a sense, the pencil model is "line-like", and the plate model is "plane-like". We want the OBB to be aligned with the best fit line to the former, or to the best fit plane of the latter. We can achieve this by examining the principal components of the statistical distribution of the geometry.

The shape of a cloud of points – such as the vertices of a model – can be approximately described by a covariance matrix, $\mathbf{C}$, and its location by a mean point or centroid, $\mathbf{m}$. For points in n-dimensional space, these are a real symmetric $n \times n$ matrix, and an n-vector, respectively. These quantities describe the cloud of points in the same manner as a normal Gaussian curve describes the distribution of a set of points on the real number line. The covariance matrix and centroid are just a gen-

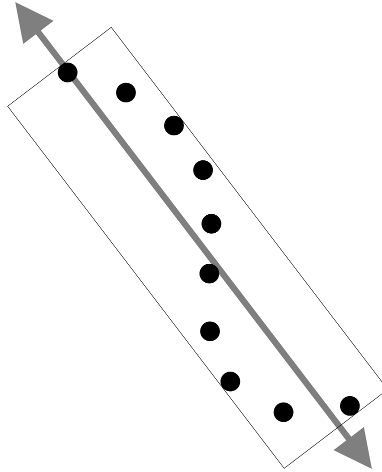


Figure 3.2: Collection of points with direction of maximal spread and OBB

eralization into higher dimensions of the familiar bell-shaped curve. The statistical spread in the direction $\mathbf{v}$ is given by $\mathbf{v}^T\mathbf{C}\mathbf{v}$. That is, the variance of the images of the points under axial projection onto $\mathbf{v}$ is $\mathbf{v}^T\mathbf{C}\mathbf{v}$. The directions which maximize or minimize this quantity are the eigenvectors of $\mathbf{C}$, which are mutually orthogonal because $\mathbf{C}$ is real and symmetric.

So, our method is to compute a covariance matrix, compute the eigenvectors of that matrix, examine the extents of the model vertices along these directions, and to determine the center point and thicknesses of the OBB accordingly. However, there are subtleties concerning *what* we compute the covariance of, and *how* we compute it.

In the remainder of this section, our figures will illustrate our methods as applied to planar models composed of points and line segments, but the methods apply equally well to points, line segments, and triangles in 3-space, which is our intended application. We should point out that these methods can be extended to all simplices embedded in a finite-dimensional Euclidean space.

3.3.1 Distribution of Vertices

Figure 3.2 shows 10 points, with their direction of maximal spread given by the large gray arrow. The arrow determines the orientation of the covariance-fitted OBB, which is shown as a tilted rectangle.

Defining the OBB orientation based on the direction of maximal spread of the vertices works well with models whose vertices are a fairly uniform sampling of the

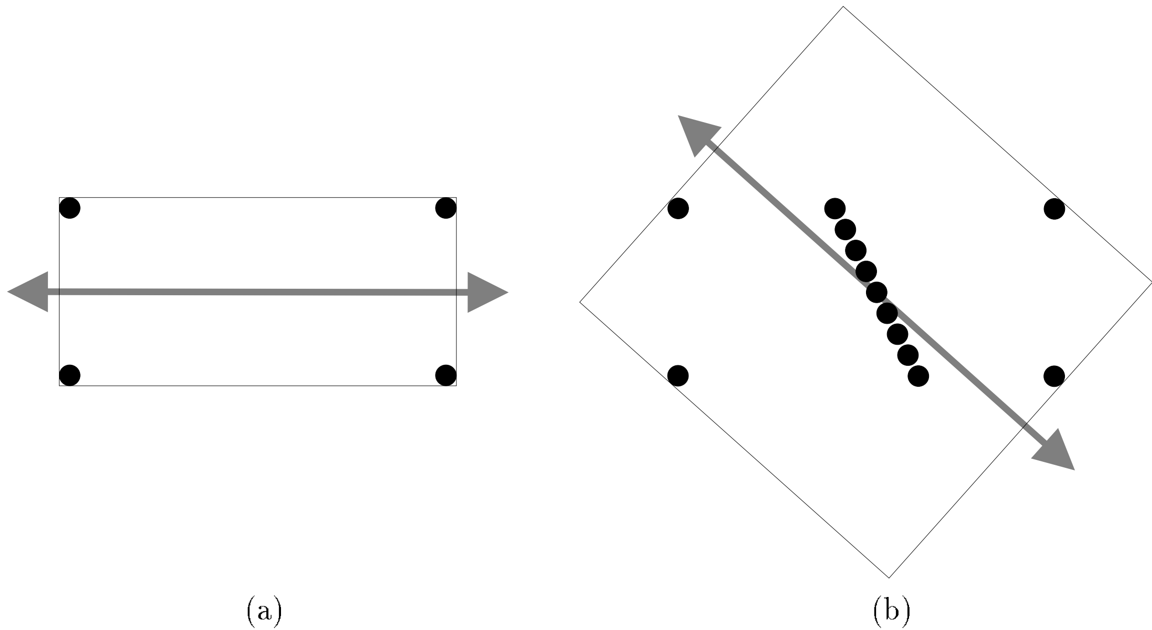


Figure 3.3: Interior vertices affect alignment.

model surface, such as surface patches which have been nicely tessellated. However, there are inputs for which this approach produces a very poorly fitted OBB.

Figure 3.3(a) shows four vertices in a rectangle. The direction of maximal spread is aligned with the rectangle, so the OBB fits perfectly. In figure 3.3(b), we have added several points to the interior of the original rectangle. This influences the distribution and alters the direction of maximal spread, causing the OBB to have poor fit. It can be shown that with a sufficient number of points on the interior of the rectangle, the OBB can be made to have *any* orientation, regardless of the shape formed by the extremal points, so long as the hull is not degenerate and has a finite number of vertices.

3.3.2 Distribution of Extremal Vertices

The solution to the problem described above is to consider only the extremal points when computing direction of maximal spread. By doing so, the interior points which caused the misalignment in the previous example are eliminated. However, even point sets consisting entirely of extremal points can produce *arbitrarily bad* orientations of the OBBs.

The two examples shown in Figure 3.4 are different samplings of the same shape: a rectangle with rounded corners. In Figure 3.4 (a), the sampling is uniform, where

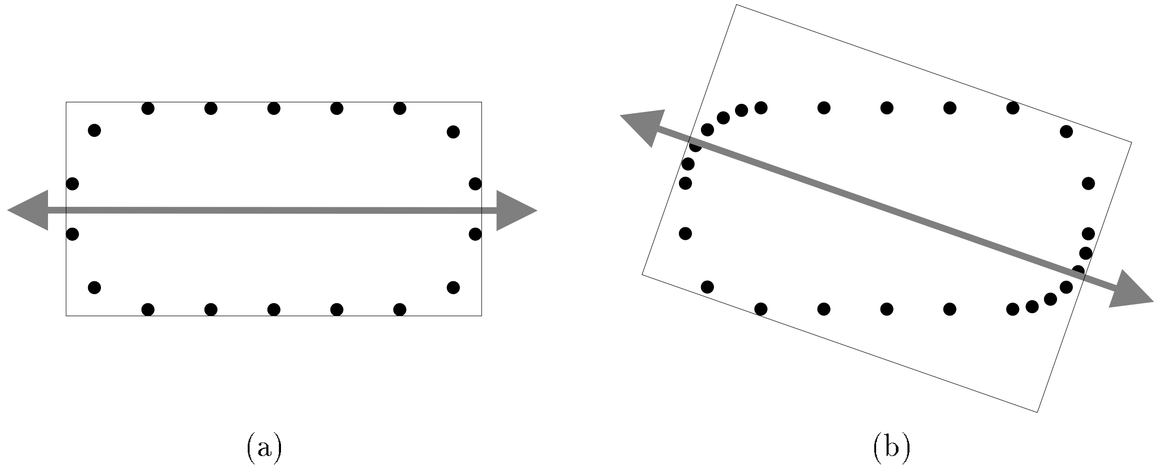


Figure 3.4: Sampling along boundary affects alignment.

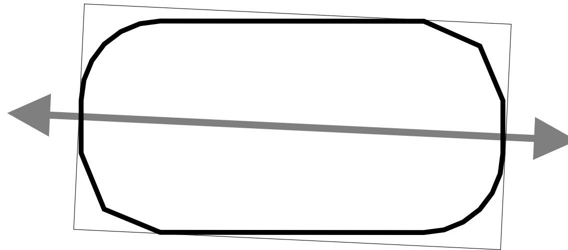


Figure 3.5: Integrating across the primitives (here, line segments) gets better results.

the sample points are roughly evenly spaced. Noting the symmetry of the figure, one can reason that the direction of maximal spread would be the horizontal arrow shown.

In Figure 3.4 (b), the upper left and lower right corners are more densely sampled. These denser clusters bias the direction of maximal spread, again causing the OBB to have a poor fit. As in the previous subsection, sufficiently dense samplings in just the right places can bias the maximal direction without bound, leading to an arbitrarily bad orientation for the OBB.

3.3.3 Distribution of Triangles (or Line Segments)

It appears that we cannot reliably use just the model vertices, since irregularities in the tessellation can cause a very bad fit. We must abandon points altogether, and consider the distribution of the infinite collection of points which are the primitive elements themselves. That is, the points of the 2D shape in Figure 3.5 is a subset of the plane, and the distribution of this subset will yield a fit which is less sensitive to

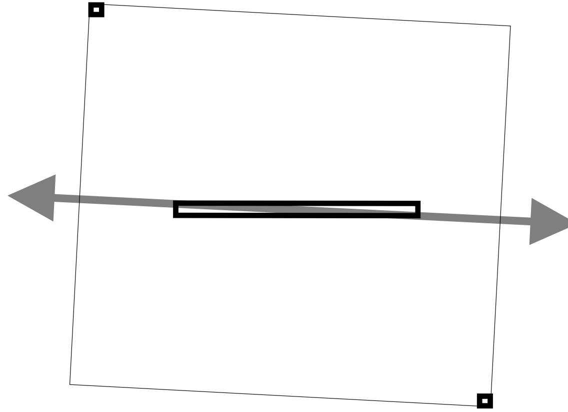


Figure 3.6: Small outliers expand the box without significantly realigning it.

the sampling density.

Even though the upper left and lower right rounded corners have more vertices, there is still approximately the same amount of line segment material to contribute to the covariance matrix. That is, some corners have a few long line segments, and others have many short ones, but the overall length is similar. Notice, however, that the figure is not perfectly symmetric, and that there will still be some small misalignment.

3.3.4 Distribution of Convex Hull Boundary

The difficulty with the last proposed method is that contributions are proportional to the length of the line segments, and certain inputs can contain very small line segments which do not significantly affect the orientation of the OBB, but which nevertheless require the OBB to be expanded to include them, such as in Figure 3.6. We see in this figure that the direction of maximal spread is mostly along the long axis of the center rectangle, and slightly turned due to the two small squares. Because the squares are so small, they do not significantly affect the direction of maximal spread – most of the contributions to the covariance matrix comes from the center rectangle. Unfortunately, although the squares may be insignificant contributors to the statistical distribution, the OBB must be made large enough to include them, making for a very large box with a bad fit.

The question is, how do we automatically recognize when small features, like the squares, are important enough to determine the alignment of the OBB? Our solution is to use the boundary of the convex hull of the shape, rather than the shape itself.

Figure 3.7 shows the original shape drawn in thin black line. The thick black line

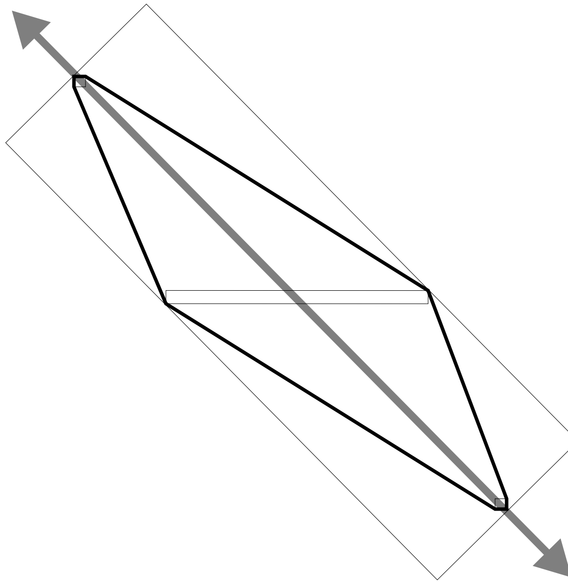


Figure 3.7: Using the convex hull boundary gives good fit.

segments are the convex hull boundary. The gray arrow is the direction of maximal spread of the line segments forming the convex hull boundary. The resulting OBB is drawn in thin black line, enclosing the convex hull. The use of the convex hull makes the covariance-based fit much more reliable in the face of unusual polygon distributions.

3.3.5 Degenerate Convex Hulls

Although the convex hull approach is very resistant to misalignment caused by dense clusters or tiny outliers of model geometry, it is still vulnerable to extremely degenerate geometry: a model whose vertices are collinear. Even if the model is embedded in 3-space, such a degenerate model is said to have a low intrinsic dimension (in this case, 1). The elements of the covariance matrix are ultimately computed as integrals over the boundary of the hull. The convex hull of a collinear model just a line segment which has zero surface area, and consequently all integrals over the surface would be zero, yielding a covariance matrix containing only zeroes. Such a matrix implies a distribution with zero spread in all directions: a single point. This is an error arising from the incorrect assumption that the model vertices are in sufficiently general position.

This weakness is easily corrected: convex hull algorithms can be augmented to report when the inputs have low intrinsic dimension (such as planar or collinear

inputs). Also, the covariance matrix of the vertices of the triangles yields a good estimate of intrinsic dimension: the intrinsic dimension is the number of nonzero eigenvalues. An extremely small eigenvalue indicates a "nearly degenerate" model.

If we are concerned that our inputs may sometimes be so degenerate, we can explicitly check the intrinsic dimension. When the points are collinear, the best-fit OBB is aligned with the line and fits it well. Otherwise, we proceed as normal, guaranteed of a meaningful covariance matrix based on the integration of a convex hull boundary.

3.3.6 Summary of Covariance Method

The use of vertices alone works well when the model is a uniformly tessellated mesh. We took an adversarial approach and identified a specific weakness in the method – that internal vertices could cause misalignment – so we refined the technique to use only the extremal vertices. Again, we found a weakness with the use of extremal vertices, so we used the whole primitives (be they triangles in 3D or line segments in 2D). This works well, but it also has weaknesses – small outliers expand the box without suitably realignining it – so we started used the facets of convex hulls. Using convex hulls is the least prone to poor fits , but it is also the most expensive.

If we choose to use the n primitives directly, we can compute the covariance matrix in $O(n)$ time, after which a constant amount of time is required to compute the eigenvectors, and then another $O(n)$ time is required to find the center point $\mathbf{c}$ and extents of the OBB. If we use the convex hull approach, then we spend $O(n \log n)$ time computing the convex hull, then $O(k)$ time computing the covariance matrix (where k is the number of facets in the hull), and constant time finding the eigenvectors, and finally $O(n)$ time for the center point and extents. The methods are summarized in the list below, according to the elements used to compute the covariance matrix,

Vertex points: $O(n)$ work, good for simple models whose vertices are a uniform sampling of the surface

Triangle facets: $O(n)$ work, good for simple models – the tessellation need not be uniform, but model should not have much interior geometry.

Convex hull facets: $O(n \log n)$ work, good for general inputs

3.4 Formulas for Covariance Matrices

The various methods presented in the previous section require the computation of a covariance matrix for the spatial distribution of various elements, depending on the method being used. The simplest requires the covariance matrix of the vertices of the primitive elements. Others require the covariance matrix of line segments or of triangles (depending on whether we are in 2D or 3D space) belonging to the model itself or of the convex hull (depending on choice of method). Consequently, we need to derive the formulas for the covariance matrix of points, lines, and triangles in 2- and 3-space. The formulas we present for the elements of the covariance matrix are valid for all dimensions.

3.4.1 Covariance Matrix of Points

Suppose we are given a collection of n points, $\mathbf{p}^1, \mathbf{p}^2, \ldots, \mathbf{p}^n$, to which to fit a box. These points form a cloud in 3 space, and have some statistical distribution, characterized by the mean, $\mathbf{m}$, and covariance matrix, $\mathbf{C}$. The mean describes the "center of mass" of the cloud. The covariance matrix contains information about how the cloud is approximately spread out, i.e. whether it is ball-like, plane-like, or line-like.

More precisely, the eigenvectors of the covariance matrix give the directions along which the cloud has maximum and minimum statistical spread. The i, jth element of the covariance matrix $\mathbf{C}$ is defined as

$$\mathbf{C}_{ij} = \mathrm{Cov}(\mathbf{x}_i, \mathbf{x}_j) = \mathrm{E}[(\mathbf{x}_i - \mathbf{m}_i)(\mathbf{x}_j - \mathbf{m}_j)]$$

where $\mathbf{x}$ is a discrete random vector chosen from among the n given points, $\mathbf{x}_i$ is its ith coordinate, and $\mathrm{E}[\cdot]$ is the expectation operator. For notational convenience, we use $\mathbf{m} = \mathrm{E}[\mathbf{x}]$.

A well-known expression for covariance is

$$\mathbf{C}_{ij} = \mathrm{Cov}(\mathbf{x}_i, \mathbf{x}_j) = \mathrm{E}[\mathbf{x}_i\mathbf{x}_j] - \mathbf{m}_i\mathbf{m}_j$$

For the n points in our set,

$$\mathbf{C}_{ij} = \mathrm{Cov}(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{n}\sum_{k=1}^{n}[\mathbf{p}_i^k\mathbf{p}_j^k] - \mathbf{m}_i\mathbf{m}_j$$

where

$$\mathbf{m}_i = \mathrm{E}[\mathbf{x}_i] = \frac{1}{n}\sum_{k=1}^{n} \mathbf{p}_i^k$$

An alternative form for the covariance is

$$\mathbf{C}_{ij} = \mathrm{Cov}(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{n}\sum_{k=1}^{n}(\mathbf{p}_i'^k \mathbf{p}_j'^k)$$

where $\mathbf{p}'^k = \mathbf{p}^k - \mathbf{m}$.

These formulas are valid for the n points embedded in any finite dimensional space.

3.4.2 Covariance Matrix of Line Segments

Given a collection of n line segments, we need to compute the covariance matrix for their distribution. The kth line segment will be denoted P^k, and is given by the endpoints $\mathbf{p}^k$ and $\mathbf{q}^k$. Its length will be denoted by L^k, and its midpoint by $\mathbf{m}^k$. The goal of this section is to express the covariance matrix as a closed-form expression of the coordinates of the endpoints. Quantities such as length and midpoint become convenient computationally and conceptually as we move from the definition of the covariance matrix to its expression in closed form.

In the previous section, $\mathbf{x}$ was a random vector which could take on the value of any of the n input points with equal probability. In this section, $\mathbf{x}$ is a random vector which takes on the value of a point on any of the input line segments with equal probability. In this case, "equal probability" means that we are randomly sampling points on the line segments with equal density, implying that the likelihood of selecting a given line is proportional to its length – please note that we are not literally sampling the line segments, but rather we are using probability theory to compute what the statistical measures would be if we were to conduct such a sampling.

We begin with the definition of covariance of a random variable $\mathbf{x}$,

$$\mathbf{C}_{ij} = (\mathrm{E}[\mathbf{x}_i - \mathrm{E}[\mathbf{x}_i]])(\mathrm{E}[\mathbf{x}_j - \mathrm{E}[\mathbf{x}_j]]) = \mathrm{E}[\mathbf{x}_i\mathbf{x}_j] - \mathrm{E}[\mathbf{x}_i]\mathrm{E}[\mathbf{x}_j]$$

Our task is to evaluate this for a collection of line segments.

The covariance is defined in terms of $\mathrm{E}[\mathbf{x}_i\mathbf{x}_j]$, $\mathrm{E}[\mathbf{x}_i]$, and $\mathrm{E}[\mathbf{x}_j]$. From the definition

of expectation of a continuous random vector, these quantities are:

$$\begin{aligned} \mathrm{E}[\mathbf{x}_i] &= \frac{\int_M \mathbf{x}_i dS}{\int_M dS} \\ \mathrm{E}[\mathbf{x}_j] &= \frac{\int_M \mathbf{x}_j dS}{\int_M dS} \\ \mathrm{E}[\mathbf{x}_i \mathbf{x}_j] &= \frac{\int_M \mathbf{x}_i \mathbf{x}_j dS}{\int_M dS} \end{aligned}$$

We will define L^M to be the sum of the lengths of all the line segments in the model, and $\mathbf{m}^M$ to be the centroid (or center of mass, or mean point) of the model as a whole:

$$L^M = \sum_k L^k$$

and

$$\mathbf{m}^M = \mathrm{E}[\mathbf{x}]$$

Since the integration domain is the model, M, which is just a collection of line segments, an integration over M can be expressed as a sum of integrals over the individual segments P^k. For all the integrals we write,

$$\int_M f dS = \sum_k \int_{P^k} f dS$$

where dS is the appropriate infinitesimal arc length element of the integration.

So, the three integrals become

$$\begin{aligned} \mathrm{E}[\mathbf{x}_i] &= \frac{\sum_k \int_{P^k} \mathbf{x}_i^k dS}{\sum_k \int_{P^k} dS} \\ \mathrm{E}[\mathbf{x}_j] &= \frac{\sum_k \int_{P^k} \mathbf{x}_j^k dS}{\sum_k \int_{P^k} dS} \\ \mathrm{E}[\mathbf{x}_i \mathbf{x}_j] &= \frac{\sum_k \int_{P^k} \mathbf{x}_i^k \mathbf{x}_j^k dS}{\sum_k \int_{P^k} dS} \end{aligned}$$

Recall that the kth line segment has endpoints $\mathbf{p}^k$ and $\mathbf{q}^k$. In order to integrate over the line segment we need to parameterize it. We will use

$$\mathbf{x}^k(s) = \mathbf{p}^k + s(\mathbf{q}^k - \mathbf{p}^k) = \mathbf{p}^k + s\mathbf{v}^k, \qquad s \in [0,1]$$

and

$$P^k = \{\mathbf{x}^k(s) \mid 0 \leq s \leq 1\}$$

where we will sometimes use $\mathbf{v}^k = \mathbf{q}^k - \mathbf{p}^k$ to shorten some of the formulas and derivations. With this parameterization, the integral of any function f over the kth line segment is

$$\int_{P^k} f dS = L^k \int_0^1 \mathbf{f} ds$$

The integral of the constant function 1 over a line segment is just the length of the line segment

$$\int_{P^k} dS = L^k \int_0^1 ds = L^k(1-0) = L^k$$

This simplifies the above equations,

$$\begin{aligned}
\mathrm{E}[\mathbf{x}_i] &= \frac{\sum_k \int_{P^k} \mathbf{x}_i^k dS}{\sum_k L^k} = \frac{1}{L^M} \sum_k \int_{P^k} \mathbf{x}_i^k dS \\
\mathrm{E}[\mathbf{x}_j] &= \frac{\sum_k \int_{P^k} \mathbf{x}_j^k dS}{\sum_k L^k} = \frac{1}{L^M} \sum_k \int_{P^k} \mathbf{x}_j^k dS \\
\mathrm{E}[\mathbf{x}_i \mathbf{x}_j] &= \frac{\sum_k \int_{P^k} \mathbf{x}_i^k \mathbf{x}_j^k dS}{\sum_k L^k} = \frac{1}{L^M} \sum_k \int_{P^k} \mathbf{x}_i^k \mathbf{x}_j^k dS
\end{aligned}$$

where L^M is the sum of all the segment lengths.

Now consider the integral $\int_{P^k} \mathbf{x}_i^k dS$. Using the parameterization for the kth line, we have

$$\begin{aligned}
\int_{P^k} \mathbf{x}_i^k dS &= L^k \int_0^1 \mathbf{x}_i^k(s) ds \\
&= L^k \int_0^1 \mathbf{p}_i^k + s\mathbf{v}_i^k ds \\
&= L^k \left[s\mathbf{p}_i^k + \frac{1}{2} s^2 \mathbf{v}_i^k \right]_0^1 \\
&= L^k \left[\mathbf{p}_i^k + \frac{1}{2} \mathbf{v}_i^k \right] \\
&= L^k \left[\mathbf{p}_i^k + \frac{1}{2} (\mathbf{q}_i^k - \mathbf{p}_i^k) \right] \\
&= L^k \frac{1}{2} (\mathbf{p}_i^k + \mathbf{q}_i^k) \\
&= L^k \mathbf{m}_i^k
\end{aligned}$$

The result of the integral is the midpoint of the line segment multiplied by its length. Of course, we can substitute the variable j for i to get a similar solution for $\int_{P^k} \mathbf{x}_j^k dS$. Remember that we are summing this integral over all the line segments, so

$$\mathbf{m}_i^M = \mathrm{E}[\mathbf{x}_i] = \frac{1}{L^M} \sum_k L^k \mathbf{m}_i^k$$

The integral $\int_{P^k} \mathbf{x}_i^k \mathbf{x}_j^k dS$ is more complex, but still relatively simple to evaluate. We have

$$\begin{aligned}
\int_{P^k} \mathbf{x}_i^k \mathbf{x}_j^k dS &= L^k \int_0^1 \mathbf{x}_i^k(s)\mathbf{x}_j^k(s) ds \\
&= L^k \int_0^1 \left(\mathbf{p}_i^k + s\mathbf{v}_i^k\right)\left(\mathbf{p}_j^k + s\mathbf{v}_j^k\right) ds \\
&= L^k \int_0^1 \left(\mathbf{p}_i^k\mathbf{p}_j^k + s\mathbf{p}_i^k\mathbf{v}_j^k + s\mathbf{v}_i^k\mathbf{p}_j^k + s^2\mathbf{v}_i^k\mathbf{v}_j^k\right) ds \\
&= L^k \left[s\mathbf{p}_i^k\mathbf{p}_j^k + \frac{1}{2}s^2\mathbf{p}_i^k\mathbf{v}_j^k + \frac{1}{2}s^2\mathbf{v}_i^k\mathbf{p}_j^k + \frac{1}{3}s^3\mathbf{v}_i^k\mathbf{v}_j^k\right]_0^1 \\
&= L^k \left(\mathbf{p}_i^k\mathbf{p}_j^k + \frac{1}{2}\mathbf{p}_i^k\mathbf{v}_j^k + \frac{1}{2}\mathbf{v}_i^k\mathbf{p}_j^k + \frac{1}{3}\mathbf{v}_i^k\mathbf{v}_j^k\right) \\
&= L^k \left(\frac{1}{3}\mathbf{p}_i^k\mathbf{p}_j^k + \frac{1}{6}\mathbf{p}_i^k\mathbf{q}_j^k + \frac{1}{6}\mathbf{q}_i^k\mathbf{p}_j^k + \frac{1}{3}\mathbf{q}_i^k\mathbf{q}_j^k\right) \\
&= \frac{L^k}{6}\left((\mathbf{p}_i^k + \mathbf{q}_i^k)(\mathbf{p}_j^k + \mathbf{q}_j^k) + \mathbf{p}_i^k\mathbf{p}_j^k + \mathbf{q}_i^k\mathbf{q}_j^k\right) \\
&= \frac{L^k}{6}\left(4\mathbf{m}_i^k\mathbf{m}_j^k + \mathbf{p}_i^k\mathbf{p}_j^k + \mathbf{q}_i^k\mathbf{q}_j^k\right)
\end{aligned}$$

Having evaluated the integrals for $\mathrm{E}[\mathbf{x}_i]$, $\mathrm{E}[\mathbf{x}_j]$, and $\mathrm{E}[\mathbf{x}_i\mathbf{x}_j]$, we can now finish our derivation of the formulas for the elements of the covariance matrix.

$$\begin{aligned}
\mathbf{C}_{ij} &= \mathrm{E}[\mathbf{x}_i\mathbf{x}_j] - \mathrm{E}[\mathbf{x}_i]\mathrm{E}[\mathbf{x}_j] \\
&= \frac{1}{6L^M}\left(\sum_k L^k(4\mathbf{m}_i^k\mathbf{m}_j^k + \mathbf{p}_i^k\mathbf{p}_j^k + \mathbf{q}_i^k\mathbf{q}_j^k)\right) - \mathbf{m}_i^M\mathbf{m}_j^M
\end{aligned}$$

This formulation is fairly efficient, and can be performed with a single pass through the list of line segments. A two-pass formulation which is less subject to arithmetic

error generated by finite-precision machine arithmetic is,

$$\mathbf{C}_{ij} = \frac{1}{6L^M}\left(\sum_k L^k(4\mathbf{m}_i'^k\mathbf{m}_j'^k + \mathbf{p}_i'^k\mathbf{p}_j'^k + \mathbf{q}_i'^k\mathbf{q}_j'^k)\right)$$

where $\mathbf{p}'^k = \mathbf{p}^k - \mathbf{m}^M$, $\mathbf{q}'^k = \mathbf{q}^k - \mathbf{m}^M$, and $\mathbf{m}'^k = \frac{1}{2}(\mathbf{p}'^k + \mathbf{q}'^k) = \mathbf{m}^k - \mathbf{m}^M$.

3.4.3 Covariance Matrix of Triangles

The formulas for the covariance matrix of triangles are similar to those for line segments. Whereas in the previous subsection $\mathbf{x}$ was a random point on a line in the model, here it is a random point on a triangle. Again, we will use superscript k to refer to the kth triangle, and superscript M when referring to the model, which is the entire set of triangles. We will use $\mathbf{m}^k$ to denote the mean point of the kth triangle, $\mathbf{m}^M$ for the mean point of the entire model, A^k for the surface area of the kth triangle, and A^M for the surface area of the entire model (the sum of the A^ks). We will use P^k to refer to the kth triangle as a domain of integration , and the vertices of the kth triangle are given by $\mathbf{p}^k$, $\mathbf{q}^k$, and $\mathbf{r}^k$.

Our parameterization of the kth triangle is

$$\mathbf{x}^k(s,t) = \mathbf{p}^k + s\mathbf{u}^k + t\mathbf{v}^k, \qquad \text{with} \quad s \in [0,1], \quad t \in [0, 1-s]$$

where $\mathbf{u}^k = \mathbf{q}^k - \mathbf{p}^k$ and $\mathbf{v}^k = \mathbf{r}^k - \mathbf{p}^k$.

Integrating a function $f(s,t)$ over the surface of the kth triangle is given by

$$\int_{P^k} f dA = |\mathbf{u} \times \mathbf{v}| \int_0^1 \int_0^{1-s} f(s,t)\ dtds$$

As before, the covariance matrix is given by

$$\mathbf{C}_{ij} = \mathrm{E}[\mathbf{x}_i\mathbf{x}_j] - \mathrm{E}[\mathbf{x}_i]\mathrm{E}[\mathbf{x}_j]$$

and we apply the same definitions for expectation,

$$\mathrm{E}[\mathbf{x}_i] = \frac{\int_M \mathbf{x}_i dA}{\int_M dA}$$

$$\mathrm{E}[\mathbf{x}_j] = \frac{\int_M \mathbf{x}_j dA}{\int_M dA}$$

$$\mathrm{E}[\mathbf{x}_i\mathbf{x}_j] = \frac{\int_M \mathbf{x}_i\mathbf{x}_j dA}{\int_M dA}$$

All these integrals over M can be viewed as the sum of the integrals over each of the triangles. For instance, the denominator in the above equations is

$$\begin{aligned}
\int_M dA &= \sum_k \int_{P^k} dA \\
&= \sum_k |\mathbf{u}^k \times \mathbf{v}^k| \int_0^1 \int_0^{1-s} dt ds \\
&= \sum_k |\mathbf{u}^k \times \mathbf{v}^k| \int_0^1 (1-s) ds \\
&= \sum_k |\mathbf{u}^k \times \mathbf{v}^k| \frac{1}{2} \\
&= \sum_k A^k \\
&= A^M
\end{aligned}$$

where $A^k = \frac{1}{2}|\mathbf{u}^k \times \mathbf{v}^k|$ is the area of the kth triangle, and A^M is the area of all the triangles in the model M. This simplifies the expectation equations to

$$\begin{aligned}
\mathrm{E}[\mathbf{x}_i] &= \frac{1}{A^M} \int_M \mathbf{x}_i dA \\
\mathrm{E}[\mathbf{x}_j] &= \frac{1}{A^M} \int_M \mathbf{x}_j dA \\
\mathrm{E}[\mathbf{x}_i\mathbf{x}_j] &= \frac{1}{A^M} \int_M \mathbf{x}_i\mathbf{x}_j dA
\end{aligned}$$

The first integral evaluates to

$$\begin{aligned}
\mathrm{E}[\mathbf{x}_i] &= \frac{1}{A^M} \int_M \mathbf{x}_i dA \\
&= \frac{1}{A^M} \sum_k \int_{P^k} \mathbf{x}_i^k dA
\end{aligned}$$

and

$$\int_{P^k} \mathbf{x}_i^k dA = |\mathbf{u}^k \times \mathbf{v}^k| \int_0^1 \int_0^{1-s} \mathbf{x}_i^k dt ds$$

$$
\begin{aligned}
&= |\mathbf{u}^k \times \mathbf{v}^k| \int_0^1 \int_0^{1-s} \mathbf{p}^k + s\mathbf{u}^k + t\mathbf{v}^k dt ds \\
&= |\mathbf{u}^k \times \mathbf{v}^k| \int_0^1 (1-s)\mathbf{p}^k + (1-s)s\mathbf{u}^k + \frac{1}{2}(1-s)^2\mathbf{v}^k ds \\
&= |\mathbf{u}^k \times \mathbf{v}^k| \left(\frac{1}{2}\mathbf{p}^k + \frac{1}{6}\mathbf{u}^k + \frac{1}{6}\mathbf{v}^k\right) \\
&= |\mathbf{u}^k \times \mathbf{v}^k| \left(\frac{1}{2}\mathbf{p}^k + \frac{1}{6}(\mathbf{q}^k - \mathbf{p}^k) + \frac{1}{6}(\mathbf{r}^k - \mathbf{p}^k)\right) \\
&= |\mathbf{u}^k \times \mathbf{v}^k| \frac{1}{6}(\mathbf{p}^k + \mathbf{q}^k + \mathbf{r}^k) \\
&= \left(\frac{1}{2}|\mathbf{u}^k \times \mathbf{v}^k|\right)\left(\frac{1}{3}(\mathbf{p}^k + \mathbf{q}^k + \mathbf{r}^k)\right) \\
&= A^k \mathbf{m}_i^k
\end{aligned}
$$

So the first expectation formula evaluates to

$$
\begin{aligned}
\mathrm{E}[\mathbf{x}_i] &= \frac{1}{A^M} \sum_k |\mathbf{u}^k \times \mathbf{v}^k| \int_0^1 \int_0^{1-s} \mathbf{x}_i^k dt ds \\
&= \frac{1}{A^M} \sum_k A^k \mathbf{m}_i^k
\end{aligned}
$$

Notice that this resembles the corresponding formula for line segments in the previous subsection. The second expectation formula, for $\mathrm{E}[\mathbf{x}_j]$, is similar to that for $\mathrm{E}[\mathbf{x}_i]$.

The third integral, for $\mathrm{E}[\mathbf{x}_i\mathbf{x}_j]$, is again more complex. We follow a similar development as for line segments.

$$
\begin{aligned}
\mathrm{E}[\mathbf{x}_i\mathbf{x}_j] &= \frac{1}{A^M} \int_M \mathbf{x}_i\mathbf{x}_j dA \\
&= \frac{1}{A^M} \sum_k \int_{P^k} \mathbf{x}_i^k\mathbf{x}_j^k dA
\end{aligned}
$$

and

$$
\int_{P^k} \mathbf{x}_i^k\mathbf{x}_j^k dA = |\mathbf{u}^k \times \mathbf{v}^k| \int_0^1 \int_0^{1-s} \mathbf{x}_i^k\mathbf{x}_j^k dt ds = |\mathbf{u}^k \times \mathbf{v}^k| \int_0^1 \int_0^{1-s} f(s,t) dt ds
$$

where

$$
\begin{aligned}
f &= \mathbf{x}_i^k\mathbf{x}_j^k \\
&= (\mathbf{p}_i^k + s\mathbf{u}_i^k + t\mathbf{v}_i^k)(\mathbf{p}_j^k + s\mathbf{u}_j^k + t\mathbf{v}_j^k)
\end{aligned}
$$

$$
\begin{aligned}
&= \mathbf{p}_i^k\mathbf{p}_j^k + s(\mathbf{u}_i^k\mathbf{p}_j^k + \mathbf{p}_i^k\mathbf{u}_j^k) + t(\mathbf{v}_i^k\mathbf{p}_j^k + \mathbf{p}_i^k\mathbf{v}_j^k) \\
&\quad + st(\mathbf{u}_i^k\mathbf{v}_j^k + \mathbf{v}_i^k\mathbf{u}_j^k) + s^2(\mathbf{u}_i^k\mathbf{u}_j^k) + t^2(\mathbf{v}_i^k\mathbf{v}_j^k)
\end{aligned}
$$

The integration with respect to t gives us

$$
\begin{aligned}
\int_0^{1-s} f(s,t)dt &= (1-s)\mathbf{p}_i^k\mathbf{p}_j^k + (1-s)s(\mathbf{u}_i^k\mathbf{p}_j^k + \mathbf{p}_i^k\mathbf{u}_j^k) + \frac{1}{2}(1-s)^2(\mathbf{v}_i^k\mathbf{p}_j^k + \mathbf{p}_i^k\mathbf{v}_j^k) \\
&\quad + \frac{s}{2}(1-s)^2(\mathbf{u}_i^k\mathbf{v}_j^k + \mathbf{v}_i^k\mathbf{u}_j^k) + (1-s)s^2(\mathbf{u}_i^k\mathbf{u}_j^k) + \frac{1}{3}s^3(\mathbf{v}_i^k\mathbf{v}_j^k)
\end{aligned}
$$

and integrating that with respect to s yields

$$
\begin{aligned}
\int_0^1\int_0^{1-s} f(s,t)dtds &= \frac{1}{2}\mathbf{p}_i^k\mathbf{p}_j^k + \frac{1}{6}(\mathbf{u}_i^k\mathbf{p}_j^k + \mathbf{p}_i^k\mathbf{u}_j^k) + \frac{1}{6}(\mathbf{v}_i^k\mathbf{p}_j^k + \mathbf{p}_i^k\mathbf{v}_j^k) \\
&\quad + \frac{1}{24}(\mathbf{u}_i^k\mathbf{v}_j^k + \mathbf{v}_i^k\mathbf{u}_j^k) + \frac{1}{12}(\mathbf{u}_i^k\mathbf{u}_j^k) + \frac{1}{12}(\mathbf{v}_i^k\mathbf{v}_j^k)
\end{aligned}
$$

Substituting $\mathbf{u}^k = \mathbf{q}^k - \mathbf{p}^k$ and $\mathbf{v}^k = \mathbf{r}^k - \mathbf{p}^k$, we have

$$
\begin{aligned}
\mathbf{u}_i^k\mathbf{p}_j^k + \mathbf{p}_i^k\mathbf{u}_j^k &= \mathbf{q}_i^k\mathbf{p}_j^k - 2\mathbf{p}_i^k\mathbf{p}_j^k + \mathbf{p}_i^k\mathbf{q}_j^k \\
\mathbf{v}_i^k\mathbf{p}_j^k + \mathbf{p}_i^k\mathbf{v}_j^k &= \mathbf{r}_i^k\mathbf{p}_j^k - 2\mathbf{p}_i^k\mathbf{p}_j^k + \mathbf{p}_i^k\mathbf{r}_j^k \\
\mathbf{u}_i^k\mathbf{v}_j^k &= \mathbf{q}_i^k\mathbf{r}_j^k - \mathbf{p}_i^k\mathbf{r}_j^k - \mathbf{q}_i^k\mathbf{p}_j^k + \mathbf{p}_i^k\mathbf{p}_j^k \\
\mathbf{v}_i^k\mathbf{u}_j^k &= \mathbf{r}_i^k\mathbf{q}_j^k - \mathbf{r}_i^k\mathbf{p}_j^k - \mathbf{p}_i^k\mathbf{q}_j^k + \mathbf{p}_i^k\mathbf{p}_j^k \\
\mathbf{u}_i^k\mathbf{u}_j^k &= \mathbf{q}_i^k\mathbf{q}_j^k - \mathbf{q}_i^k\mathbf{p}_j^k - \mathbf{p}_i^k\mathbf{q}_j^k + \mathbf{p}_i^k\mathbf{p}_j^k \\
\mathbf{v}_i^k\mathbf{v}_j^k &= \mathbf{r}_i^k\mathbf{r}_j^k - \mathbf{r}_i^k\mathbf{p}_j^k - \mathbf{p}_i^k\mathbf{r}_j^k + \mathbf{p}_i^k\mathbf{p}_j^k
\end{aligned}
$$

and making the substitutions, we get

$$
\begin{aligned}
\int_0^1\int_0^{1-s} f(s,t)dtds &= \frac{1}{24}\big(12\mathbf{p}_i^k\mathbf{p}_j^k + 4(\mathbf{u}_i^k\mathbf{p}_j^k + \mathbf{p}_i^k\mathbf{u}_j^k) + 4(\mathbf{v}_i^k\mathbf{p}_j^k + \mathbf{p}_i^k\mathbf{v}_j^k) \\
&\quad + 1(\mathbf{u}_i^k\mathbf{v}_j^k + \mathbf{v}_i^k\mathbf{u}_j^k) + 2(\mathbf{u}_i^k\mathbf{u}_j^k) + 2(\mathbf{v}_i^k\mathbf{v}_j^k)\big) \\
&= \frac{1}{24}\left[2\mathbf{p}_i^k\mathbf{p}_j^k + \mathbf{p}_i^k\mathbf{q}_j^k + \mathbf{p}_i^k\mathbf{r}_j^k + \mathbf{q}_i^k\mathbf{p}_j^k + 2\mathbf{q}_i^k\mathbf{q}_j^k \right. \\
&\quad \left. + \mathbf{q}_i^k\mathbf{r}_j^k + \mathbf{r}_i^k\mathbf{p}_j^k + \mathbf{r}_i^k\mathbf{q}_j^k + 2\mathbf{r}_i^k\mathbf{r}_j^k\right] \\
&= \frac{1}{24}\left[(\mathbf{p}_i^k + \mathbf{q}_i^k + \mathbf{r}_i^k)(\mathbf{p}_j^k + \mathbf{q}_j^k + \mathbf{r}_j^k) \right. \\
&\quad \left. + \mathbf{p}_i^k\mathbf{p}_j^k + \mathbf{q}_i^k\mathbf{q}_j^k + \mathbf{r}_i^k\mathbf{r}_j^k\right] \\
&= \frac{1}{24}\left[9\mathbf{m}_i^k\mathbf{m}_j^k + \mathbf{p}_i^k\mathbf{p}_j^k + \mathbf{q}_i^k\mathbf{q}_j^k + \mathbf{r}_i^k\mathbf{r}_j^k\right]
\end{aligned}
$$

where we used $3\mathbf{m} = (\mathbf{p} + \mathbf{q} + \mathbf{r})$ in the last step.

Finally, our evaluation of the expectation $\mathrm{E}[\mathbf{x}_i \mathbf{x}_j]$ is

$$\begin{aligned} \mathrm{E}[\mathbf{x}_i \mathbf{x}_j] &= \frac{1}{A^M} \sum_k \int_{P^k} \mathbf{x}_i^k \mathbf{x}_j^k dA \\ &= \frac{1}{A^M} \sum_k 2A^k \int_0^1 \int_0^{1-s} f(s,t) dt ds \\ &= \frac{1}{A^M} \sum_k \frac{2A^k}{24} \left[9\mathbf{m}_i^k \mathbf{m}_j^k + \mathbf{p}_i^k \mathbf{p}_j^k + \mathbf{q}_i^k \mathbf{q}_j^k + \mathbf{r}_i^k \mathbf{r}_j^k \right] \end{aligned}$$

Notice how remarkably similar this is to the corresponding expectation for line segments. Now that we have evaluated the expectations, the covariance formula is

$$\mathbf{C}_{ij} = \mathrm{E}[\mathbf{x}_i \mathbf{x}_j] - \mathrm{E}[\mathbf{x}_i]\mathrm{E}[\mathbf{x}_j] = \frac{1}{A^M} \sum_k \frac{1}{24} \left[9\mathbf{m}_i^k \mathbf{m}_j^k + \mathbf{p}_i^k \mathbf{p}_j^k + \mathbf{q}_i^k \mathbf{q}_j^k + \mathbf{r}_i^k \mathbf{r}_j^k \right] - \mathbf{m}_i^M \mathbf{m}_j^M$$

3.5 Tightness of OBBs

In this section we examine the optimality of covariance-fitted OBBs. First we prove that the method generally produces OBBs with non-optimal fit, and then we discuss the worst case inputs produce by this method.

3.5.1 Non-Optimality of OBBs

In this section we show that OBBs produced by the covariance method are not optimal in terms of volume, surface area, diameter (length of longest enclosed line), *or any other function of the box widths.* If the box widths along the axes are w_x, w_y, and w_z, then the volume, surface area, and diameter measures are:

$$\begin{aligned} V &= w_x w_y w_z \\ A &= 2(w_x w_y + w_y w_z + w_z w_x) \\ D &= \sqrt{w_x^2 + w_y^2 + w_z^2} \end{aligned}$$

Consider some simple convex shape, such as that shown in Figure 3.8(a). We will call the shape M_1, and the its covariance-fitted OBB is called P_1 (drawn in thin black line). Now consider modifying the convex shape as shown in Figure 3.8(b). We have expanded the convex figure by adding a single vertex which falls within P_1 (P_1

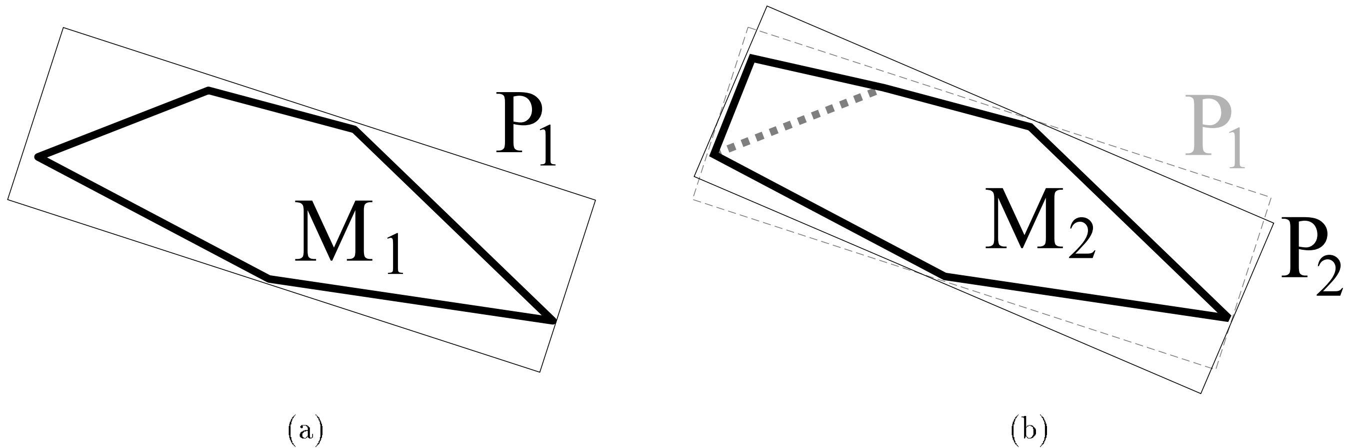


Figure 3.8: Demonstration that covariance-aligned OBBs are suboptimal. P_1 is OBB for M_1. Expanding M_1 slightly, but still staying within P_1, obtains a new model, M_2, whose covariance-aligned OBB is P_2. Since P_1 and P_2 both cover M_1 and M_2, one of them must be suboptimal.

is drawn with gray dashed lines). We will call this new convex shape M_2, and its covariance-fitted OBB is P_2.

Except in special cases, the fit of the OBB is a smooth function of the covariance matrix, so P_1 and P_2 have different orientations because they have different inputs, and therefore they have different widths as well.

Because P_2 covers M_2, and M_2 covers M_1, then P_2 covers M_1. Also, P_1 covers M_2 by construction. Therefore, P_1 and P_2 *each* cover M_2 and M_1, and because their widths are distinct, they cannot *both* be optimal in the same function of w_x, w_y, and w_z. This logic holds for any nontrivial smooth function of the box dimensions. Therefore, covariance method produces OBBs which are not necessarily optimal in volume, surface area, diameter, or any other function of the box widths.

3.5.2 Worst-Case Input for Covariance-Fitted OBBs

Although covariance-fitted OBBs are known to be non-optimal, we believe that they are always within some constant factor of optimal area (for 2D OBBs) or volume (for 3D OBBs). Here we examine the worst-case inputs for our covariance-based fitting methods.

We argue that covariance-fitted 2D OBBs have no more than twice the minimum possible area, and that squares are their worst case input. We are not able to prove this bound analytically, but we present the rationale for this conjecture, and we show

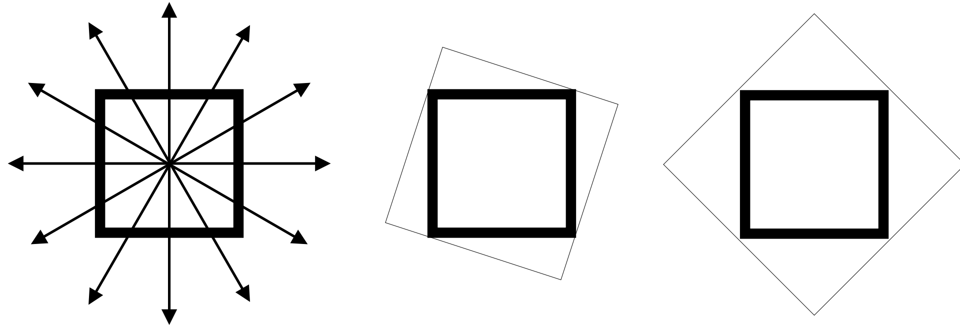


Figure 3.9: Because a square has the same statistical spread in all directions, any orientation may be chosen for the covariance-aligned OBB, and the worst choice has twice the area of the optimal choice.

the result of numerical studies which support it. On the strength of the arguments for the 2D case, we conjecture that the worst case input in 3D is a cube, and that covariance-fitted 3D OBBs have no greater than 4.63 times the minimum attainable volume.

One property of covariance is that the directions of most and least spread are perpendicular to one another – this is true for spaces of any dimension. From this it is easy to conclude, using a symmetry argument, that for any planar figure exhibiting rotational symmetry of degree 3 or more, *all* directions have equal statistical spread. In other words, equilateral triangles, squares, pentagons, and all the regular n-gons have no preferred direction. Since, for these figures, every direction has the same spread as every other, any OBB orientation could be chosen. Consequently, if given a square as input, the OBB fitting routine *could* select an orientation rotated 45 degrees with respect to the input square, which would be the superscribed OBB of *maximum* area for that input: an area twice the minimum possible. This is shown in Figure 3.9.

So, for squares, the ratio of maximum to minimum area OBB is 2 to 1. For triangles, it is less, and for pentagons, hexagons, septagons, and all the remaining n-gons, the ratio is less than two and decreases monotonically with higher n. As n approaches infinity, the n-gon resembles a circle, for which the maximum and minimum area OBB have the same area. Among all the regular n-gons, squares are the worst case inputs. We are unable to prove this statement analytically, so we performed an experiment to challenge the conjecture that squares are the worst-case input.

We wrote a program to generate random convex figures, to compute the minimum area bounding box for each one, and to compare its area to that of the covariance-

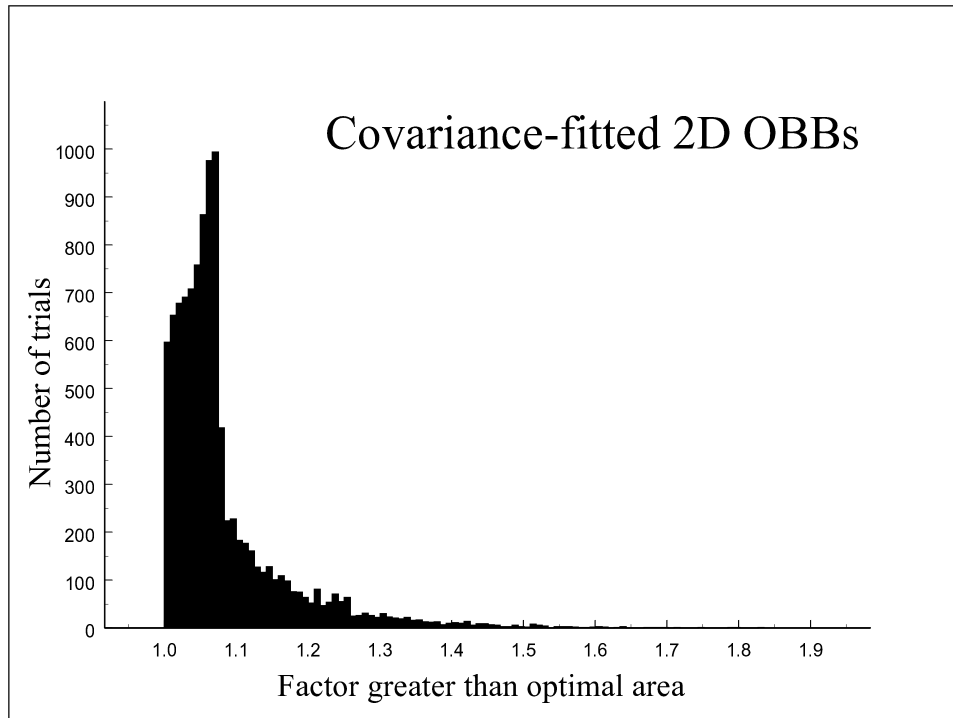


Figure 3.10: Tightness of covariance-fitted 2D OBBs for 10,000 randomly generated point sets. Each used the covariance method to fit an OBB to a set of four points randomly chosen in the unit square. The area of the covariance-fitted OBB was compared to the minimum possible OBB. 10,000 trials were conducted. This plot shows that most of the covariance-fitted 2D OBBs are within 10% of the minimum area.

aligned OBB. The convex figure was generated by computing the convex hull of four points chosen at random in the unit square. Some of the convex hulls contained four vertices, and some had three (because one of the random points was not extremal). The minimum area box was found by selecting the least volume box from among all boxes aligned with an edge of the polygon. The covariance-aligned OBB was chosen using the distribution of the edges of the convex hull boundary.

Among 10,000 test cases, all the covariance-aligned OBBs were between 1 and 2 times the area of the minimum area box. The histogram is shown in Figure 3.10. The majority of the cases yielded ratios between 1.0 and 1.05, suggesting that the covariance-aligned OBBs did quite well in minimizing volume.

We find an interesting relationship between the condition number of the covariance matrix and the area ratio: the higher the condition number, the lower the ratio is likely to be. We see that the ratios range from 1 to 2, and when the condition number is especially high, the ratios tend to be closer to 1.

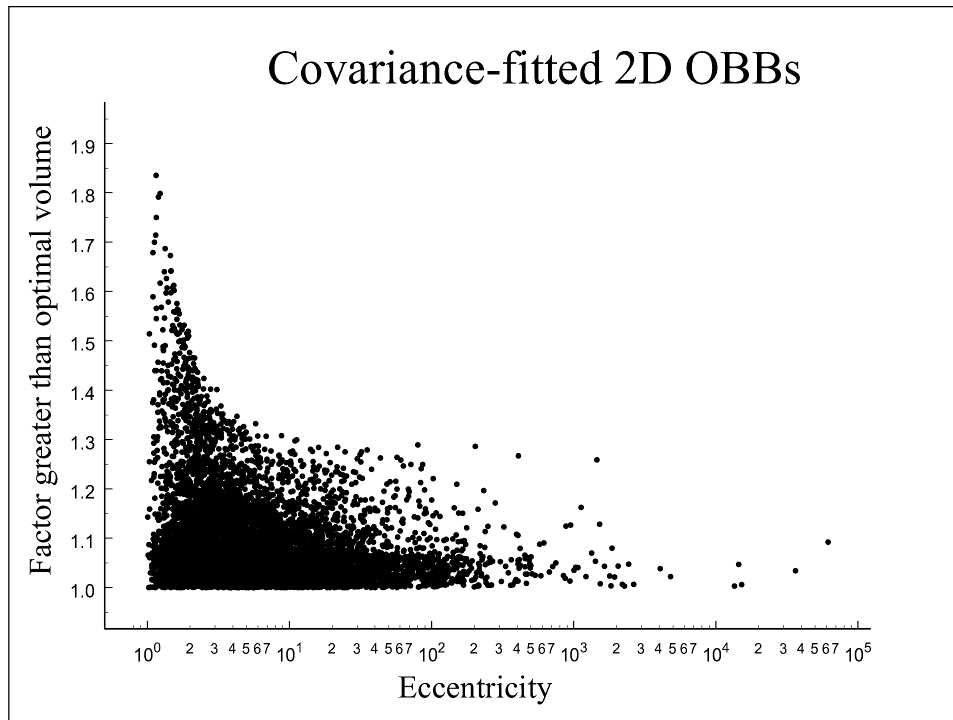


Figure 3.11: Relationship between eccentricity and volume optimality. The plot suggests that if the eccentricity of the planar input figure exceeds 10, then the covariance-fitted OBB will have within a factor 1.3 of minimum possible area.

In Figure 3.11 we show the condition number (x-axis) versus the ratio (y-axis) for the same 10,000 trials. This is a log-linear plot, so we can see the effect of a wide range of condition numbers. Very clearly we can see that condition number is related to the maximum ratio we can expect to see. For example, we would not expect to find a ratio above 1.30 associated with condition numbers above 10. The condition number of the covariance matrix is the ratio of the largest eigenvalue over the smallest eigenvalue, which is the ratio of most spread over least spread. Hence, it is a measure of eccentricity: long thin objects yield a high condition number, and fat object have a low condition number. The regular polygons all have condition number of 1. This is good news for the method of covariance-alignment. It means that the method is most effective for long thin objects, which are the objects for which volume of the OBB is most sensitive to alignment.

The 3D case is harder to test because their is no known simple implementation for computing a minimum volume OBB (there is a method devised by O'Rourke, but is difficult to implement). Our conjectured worst case input is an cube. The cube has degree four rotational symmetry along three separate axes, from which we infer that it has equal spread in every direction. This can be confirmed directly by computing the covariance matrix and observing that it is a real multiple of the identity matrix, and consequently all vectors are eigenvectors with the same eigenvalue. The worst fit covariance-aligned OBB would be found by rotating the OBB 60 degrees about the axis $(1, 1, 1)$, that is, the axis extending from the center of the cube through a corner. The resulting misaligned OBB has volume 4.63 times the volume of the cube. Our conjecture is that no covariance-aligned OBB will have worse than 4.63 times the volume of the minimum volume box, and furthermore (on the strength of our observations for the 2D case) that for more eccentric shapes, we can expect to see a tighter fit.

3.6 Summary of Chapter

The main result of this chapter is a covariance-based method for fitting an OBB to a collection polygons. The method aligns the OBB with the principal directions of the covariance matrix of the surface of the convex hull of the polygons. This technique is reasonably unbiased by unusual distributions of polygons. We proved that covariance-fitted OBBs do not necessarily produce minimal volume OBBs. We also demonstrated that 2D covariance-fitted OBBs will have within a factor of two of

minimum area, and we conjectured that 3D covariance-fitted OBBs will have within a factor of 4.63 of minimum volume. Furthermore, more elongated models will tend to be better fitted.

Chapter 4

OBB Overlap Test

This chapter presents a new algorithm and its implementation for determining whether two oriented bounding boxes (OBBs) are in contact. We believe that although some previous researchers were aware that OBBs would fit their models more tightly than spheres and AABBs, they avoided using OBBs for collision detection because known methods for testing two OBBs for overlap were not fast enough. The new OBB overlap test is several times faster than previously known tests for OBBs, and is fast enough to make OBBs a reasonable alternative to the other BV types.

Section 4.1 presents previous known methods for determining whether differently oriented bounding boxes overlap. In Section 4.2 presents a proof of the "separating axis theorem." Section 4.3 presents an implemention of the test based on the theorem as applied to OBBs. We begin with a simple formulation of the problem and apply a series of optimizations, yielding a test which completes in 89 operations in the best case, and 252 operations in the worst case. Section 4.4 presents optimizations for the special cases of OBBs with zero thicknesses or infinite extents. Section 4.5 presents a case study of a potential failure due to arithmetic error, and discusses its remedy. Section 4.7 reviews the results of the chapter.

4.1 Previous Methods

There are at least five prior approaches for determining whether two OBBs overlapped: linear programming, Gilbert-Johnson-Keerthi distance computation, Lin-Canny distance computation, Ned Greene's box-polytope intersection test, and exhaustive edge-face testing.

4.1.1 Linear Programming

Linear programming can be used to determine the contact status of any two convex polytopes. The linear programming approach poses the OBB overlap test as a feasibility problem in 4 variables with 16 constraints. The variables are the coefficients for the equation

$$f(x, y, z) = \alpha x + \beta y + \gamma z + \delta.$$

The points in space for which $f(x, y, z) = 0$ describe a plane. If all eight vertices of one box lie on an opposite side of the plane from those of the other box, then the plane is a separating plane. The 16 constraints are

$$\begin{aligned} f(x'_1, y'_1, z'_1) &< 0 \\ f(x'_2, y'_2, z'_2) &< 0 \\ &\vdots \\ f(x'_8, y'_8, z'_8) &< 0 \end{aligned}$$

and

$$\begin{aligned} f(x''_1, y''_1, z''_1) &\geq 0 \\ f(x''_2, y''_2, z''_2) &\geq 0 \\ &\vdots \\ f(x''_8, y''_8, z''_8) &\geq 0 \end{aligned}$$

where (x'_i, y'_i, z'_i) are the coordinates of the vertices of one box, and (x''_i, y''_i, z''_i) are the coordinates of the vertices of the other box. If standard LP methods can find a feasible solution, then the boxes are disjoint. If no feasible solution can be found, then they overlap.

4.1.2 Gilbert-Johnson-Keerthi Distance Computation

As with linear programming, the GJK algorithm is applicable to any pair of convex polytopes. This technique computes the distance between the two boxes: nonzero distance implies separation, while zero distance implies contact. This method constructs

a sequence of simplices whose vertices are taken from the vertices of the Minkowski difference of the boxes. Each simplex generated is closer to the origin than the previous, and a simple test determines when the closest possible simplex has been found. The process stops, trivially, when a simplex containing the origin is found. The distance of the final simplex from the origin is the distance between the OBBs.

4.1.3 Lin-Canny Distance Computation

The Lin-Canny algorithm is also applicable to all convex polytopes. This method locates the pair of closest features (a feature is a vertex, edge, or face) by walking across the surfaces of the polytopes until the two closest features are found, which can be verified by examining their external Voronoi regions. The distance between the closest features is then taken as the distance between the polytopes. This method is designed for use as a polytope distance computation step for use in scenarios exhibiting temporal coherence. It exploits temporal coherence because by tracking the pair of closest points on the polytopes, it has relatively little work to do when the polytopes move a small amount. For sufficiently small movements, the Lin-Canny algorithm performs a constant amount of work, regardless of the polytope's complexities.

4.1.4 Greene's Box-Polytope Test

Greene proposes an excellent method for determining whether a box and a convex polytope overlap. His method is based on the observation that a box and a polytope overlap if and only if (1) the images of the box and polytope intersect under all three orthographic projections (top, side, and front views), and (2) the box does not lie entirely on the outside of any of the halfspaces forming the polytope. His method begins by determining which vertex of the box is nearest to each face-plane of the polytope, computing the axis-aligned bounding box for the polytope, and finally determining for each of the three orthographic projections the vertex of the box closest to the each silhouette-line of the polytope. As applied to two boxes, the test requires evaluation of no more than thirty inequalities to determine contact status. The strength of Greene's method is that permits a series of same-aligned (but differently-placed and differently-sized) boxes to be efficiently tested against a polytope, such as a viewing frustum. Its weakness is that it requires significant computation before performing the thirty comparisons.

4.1.5 Exhaustive Edge-Face Testing

The most straightforward overlap test involves testing all the edges of one box for intersection with all the faces of the other, and vice-versa. If two boxes touch, then either one completely contains the other, or their boundaries overlap. Assuming general position, the latter case implies that a face and an edge are intersecting. Therefore, exhaustively testing all combinations will find the contact. If contact is not found, then we test whether any point in one box is entirely contained in the other box, and vice-versa. This requires 144 edge-face tests, of which each is a 3-by-3 linear system, plus two point-in-box tests. Some optimization is possible if we transform the problem so that one of the boxes is axis-aligned.

4.1.6 Summary of Previous Methods

The exhaustive face-edge testing method requires 144 inequalities, which is too many tests to be competitive with the other methods. However, it is easy to implement. It potentially suffers from robustness problems – if given a nearly parallel face and edge, the linear system will be ill-conditioned.

The linear programming method is general, applicable to pairs of convex polytopes, and consequently it is more powerful than we require.

The two distance computation methods are still the methods of choice in the problem domains for which they were designed, but they also apply to more general inputs, and they compute distance, which is more information than we require. Simpler and faster methods can be used to compute the overlap status of simple boxes.

Greene's description of his algorithm emphasizes its applicability to testing whole families of boxes against a given polytope – there is significant precomputation involved in finding which vertices are closest to which polytope face-planes, and doing the same to their images under each of three orthogonal projections. This method is ideal for testing a series of same-aligned boxes against the polytope, but is not so well-suited for a single box-box overlap test.

4.2 Separating Axis Theorem

An axis $\mathbf{n}$ is a *separating axis* of two point sets A and B if and only the images of A and B under axial projection onto $\mathbf{n}$ are disjoint. The separating axis theorem states that two polytopes A and B are disjoint if and only if there exists a separating axis

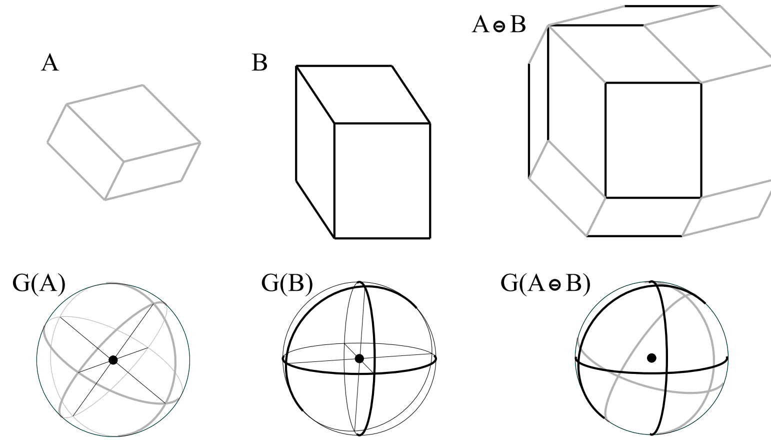


Figure 4.1: The Gauss map $G(A)$ of an OBB A is three mutually orthogonal great circles. Since the Gauss map of the minkowski difference $G(A \ominus B)$ of two polytopes A and B is the superposition of their individual Gauss maps, $G(A \ominus B)$ consists of six great circles, of which the first three are mutually orthogonal, and the second three are mutually orthogonal. This can be used to prove that in general the polytope $A \ominus B$ has 30 faces, 60 edges, and 32 vertices.

which is either perpendicular to a face of one of the polytopes or is perpendicular to an edge taken from each.

The separating axis theorem is the dual of a well-known theorem which states that two polytopes are disjoint if and only if there exists a separating plane which is parallel to a face of one of the polytopes or is parallel to an edge taken from each. Clearly, if a plane is a separating plane, then any axis perpendicular to it is a separating axis. Similarly, if an axis is a separating axis, then there exists a separating plane perpendicular to it.

If one accepts the validity of the separating plane theorem, then the separating axis theorem follows as a simple corollary. We will examine the structure of the problem, so as to gain insight into its efficient solution. We will first examine the structure of the Minkowski difference of two OBBs, and then show how this structure relates to the axes which separate the OBBs. We should point that the Minkowski difference of two OBBs is never explicitly computed during our overlap test – we only use it in this exposition to help explain the structure of the problem.

4.2.1 Minkowski Difference of Two OBBs

The *features* of a polyhedron $P \subset R^3$ are its vertices, edges, and faces. These features have 0, 1, and 2 dimensions, respectively. A feature $F \subset P$ is *extremal* in a given

direction $\mathbf{v}$ (taken as a unit vector) if

$$\mathbf{f} \cdot \mathbf{v} \geq \mathbf{p} \cdot \mathbf{v} \qquad \forall \quad \mathbf{f} \in F, \mathbf{p} \in P$$

meaning that the entire feature is at least as far in direction v as any other part of the polytope. Every point on a unit sphere can be associated with a direction in 3-dimensional space. A Gauss map of a polyhedron is a partitioning of the surface of a sphere according to this rule: the point $\mathbf{v}$ on the sphere is associated with the feature extremal in direction $\mathbf{v}$ which has largest dimension. We say that the feature F is mapped to the set of directions D whose points $\mathbf{d} inD$ are associated with F under that rule.

The Gauss map of a polyhedron always has the following structure. Vertices map to convex regions on the sphere. An edge maps to great arcs (a segment of a great circle) on the sphere composed of directions perpendicular to that edge. A face maps to exactly one point on the sphere, which is the direction of the face's normal.

The Gauss map, $G(A)$, of a box A is three mutually orthogonal great circles. Likewise for the Gauss map $G(B)$ for box B. The Gauss map $G(P)$ of their Minkowski difference $P = A \ominus B$ is the superposition of $G(A)$ and $G(B)$ and therefore consists of six great circles, of which the first three are mutually orthogonal, and the last three are mutually orthogonal. By "orthogonal great circles", we mean that they meet only at right angles on the surface of the sphere. These are shown in Figure 4.1.

If we assume that the boxes are in general position, then no three great circles meet in a point. Consequently, the number of vertices on $G(P)$ is 30. This is because there are $\binom{6}{2} = 15$ pairs of circles, and each pair produces two vertices (as antipodal pairs). The 30 vertices in $G(P)$ correspond to the 30 faces of P, and since the vertices of $G(P)$ come in antipodal pairs, the faces of P come in parallel pairs (with opposite outward normals). There is a one-to-one correspondence between the edges of any convex polytope and the arcs of its Gauss map. The Gauss map $G(P)$ has 60 arcs, since each of the six great circles is twice cut by each of the remaining five circles, therefore $A \ominus B$ has 60 edges. Euler's formula, $2 = E - F - V$, can then be used to deduce that $A \ominus B$ has 32 vertices. If the boxes are not in general position, more than two great circles may meet in a point, implying that the vertices, arcs, and regions on $G(P)$ number even less than for general position.

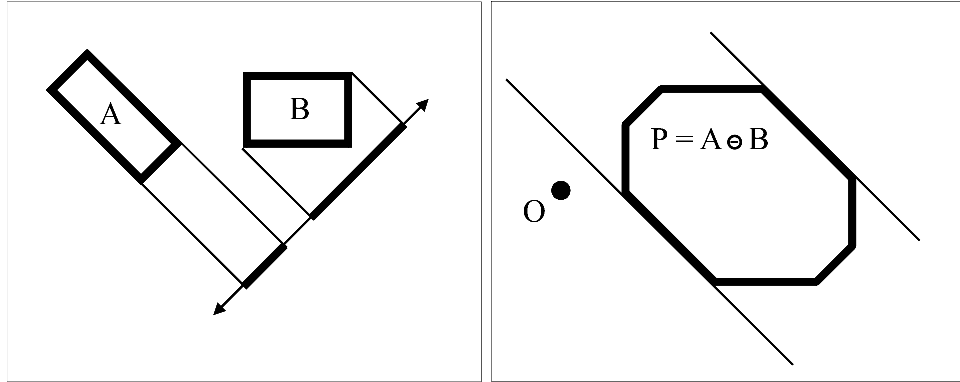


Figure 4.2: If A and B are OBBs, then $P = A \ominus B$ is the intersection of 15 slabs. A given slab contains the origin if and only if the axis perpendicular to the slab is a separating axis. In fact, the distance of the origin O from the slab equals the length of the gap between the images of A and B under axial projection onto the separating axis.

4.2.2 Minkowski Difference and Separating Axes

Any polytope can be defined as an intersection of halfspaces, each face defining a halfspace. Our polytope P, since it is composed of 15 pairs of parallel faces, is the intersection of 15 slabs, each slab being the intersection of the two halfspaces corresponding to the associated two faces. These 15 slabs have a particular organization, which we examine next.

Let the circles of $G(A)$ be colored gray, and the circles of $G(B)$ be colored black. Let the corresponding circles on $G(P)$ be gray or black, according to whether they derive from $G(A)$ or $G(B)$. Three antipodal pairs of the vertices of $G(P)$ come from crossings of gray circles, and three pairs come from crossings of black circles. Each of the remaining nine pairs come from the crossings of a gray and a black circle.

Every location on the Gaussian sphere is associated with a direction. The points at the crossings of gray circles are located at directions associated with the normals of the faces of A. Similarly, the points at the crossings of black circles are located at directions associated with the normals of the faces of B. The remaining points, the crossings of black and gray, are directions orthogonal to an edge from each of A and B – since a gray circle is the set of directions orthogonal to some edge of A, and a black circle is the set of directions orthogonal to some edge of B, those circles cross at directions which are by definition orthogonal to both. Consequently, the faces of P, whose normals are associated with the vertices of $G(P)$, are either parallel to faces of A or B, or parallel to an edge from each.

Recall that the Minkowski difference P contains the origin if and only if the A and B overlap. Since P is the intersection of 15 slabs, it contains the origin if and only if the origin lies inside all 15 slabs. Therefore, we can pose the problem as 15 origin-in-slab tests. Each slab test corresponds to an axis test for the original OBB, as shown in Figure 4.2: a slab contains the origin if and only if the axis perpendicular to the slab is a separating axis of the OBBs. Since the slabs are aligned with the faces of P, and the faces of P are either parallel to a face of an OBB or parallel to an edge taken from each OBB, this proves the separating plane theorem, and the separating axis theorem follows as a corollary.

If the boxes are not in general position, some of the 15 candidate axis directions will coincide, but they will still form a sufficient set of tests.

4.2.3 Combinatoric Structure of Separating Axis Theorem

Given two general convex polytopes, each with F faces and E edges, the separating axis theorem states that it would suffice to test $E^2 + 2F$ candidate axes to determine whether they overlap – F candidates normal to the faces of the first polytope, F candidates normal to the faces of the second, and E^2 candidates perpendicular to the edges taken pairwise. For example, the test for tetrahedra would require 44 candidates.

Furthermore, computing the image of one of the polytopes under axial projection onto to axis could require $O(V)$ time, where V is the number of vertices. A search structure could be built for the polytope to reduce this to $O(\log V)$ time. If we can assume that the number of edges and faces of the polytopes are approximately proportional to the number of vertices, then the overlap test based on separating axes would require $O(V^2 \log V)$ time, showing that the method scales very poorly with the number of vertices.

However, OBBs have special structure which makes them especially well-suited for the separating axis theorem. Although tetrahedra require 44 axis tests, OBBs only require 15. This is because there are only 3 *distinct* face normals and 3 *distinct* edge directions on an OBB (disregarding sign). Furthermore, the images of the OBBs under axial projection can be computed quite efficiently, as will be shown in Section 4.3.3. Finally, the fact that the set of face normal directions and the set of edge directions are the same, and furthermore these directions are mutually orthogonal, permit further simplifications. OBBs (including AABBs as a special case) are unique among the 3D polytopes in possessing these properties. No other solid shape permits the same

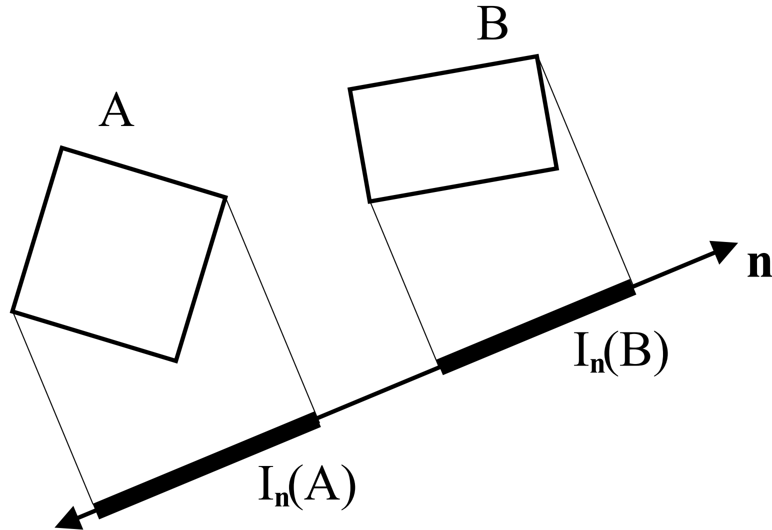


Figure 4.3: An axis test determines whether the images of A and B are disjoint under axial projection. Fifteen such tests are sufficient to determine whether two OBBs overlap.

degree of algebraic and combinatoric simplification for the separating axis theorem.

4.3 Implementation for OBBs

Recall that P is the intersection of 15 slabs. Testing whether the origin falls outside a slab of P is equivalent to testing whether A and B are disjoint under projection onto the slab's face normal. How simply can the axial projection be expressed, in terms of the basic arithmetic operations?

4.3.1 Posing the Problem

Suppose we have two arbitrarily oriented boxes, A and B, in a common coordinate system, and we wish to determine whether they are overlapping. Let the half-widths of A along its x-, y-, and z-axes be given by a_1, a_2, and a_3. Let the center of A be located at the point $\mathbf{T}^A$, and let the three axis directions of A be the vectors $\mathbf{R}_1^A$, $\mathbf{R}_2^A$, and $\mathbf{R}_3^A$, which form the columns of the rotation matrix $\mathbf{R}^A$. Let the half-widths, position, and orientation of B be similarly defined.

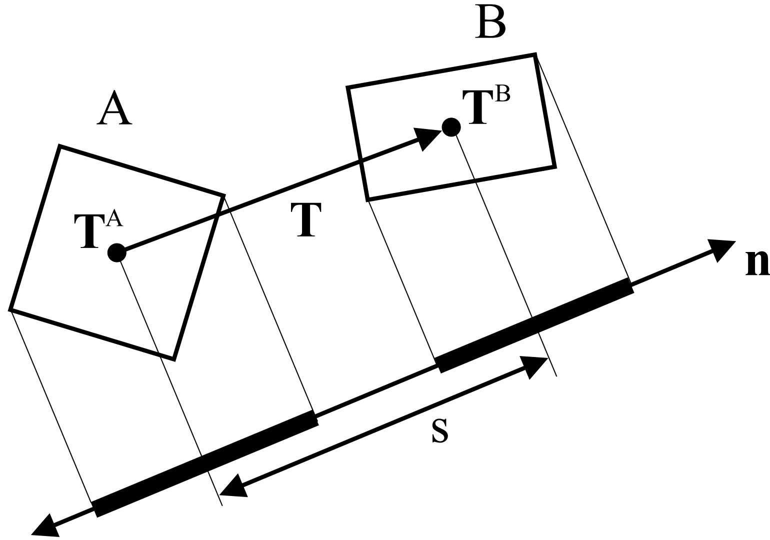


Figure 4.4: The box centers project onto the midpoints of their intervals. The separation of the midpoints equals the length of the projection of the vector joining the box centers: $s = \mathbf{T} \cdot \mathbf{n}/|\mathbf{n}|$.

4.3.2 Simple Formulation

Since A and B are closed region of genus 0 (i.e. they don't have holes), their images under axial projection are just intervals on the axis. Since they are polytopes, we can project just their vertices onto the axis, and take their maximum and minimum extents to find the endpoints of the intervals. Since there are 8 vertices for A, it takes 8 dot products plus 14 comparisons to find the extents of the interval on the axis. Likewise for B. Then there are a couple of comparisons to determine if the intervals are disjoint.

That makes 16 dot products and 30 comparisons for an axis test. Since there are 15 axes for a complete overlap test between boxes, this could amount to 225 dot products and 450 comparisons. Those 225 dot products require 675 multiplies and 550 additions. And this assumes that we have the vertices of the boxes available without any additional computation.

4.3.3 Optimization: Interval Widths and Separations

Observe that the center point of a box projects onto the midpoint of that box's interval, which means we can project the center points onto the axis to get the separation between the interval midpoints, as shown in Figure 4.4. Furthermore, we can compute the width of an interval by projecting any three mutually orthogonal edges

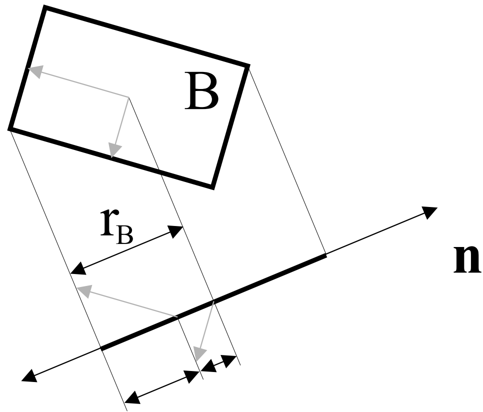


Figure 4.5: The half-width (radius) of an interval equals the sum of the lengths of the projections of the box axes. For 2D OBBs, as depicted here, we have only 2 box axes, but for 3D OBBs we would have 3 box axes.

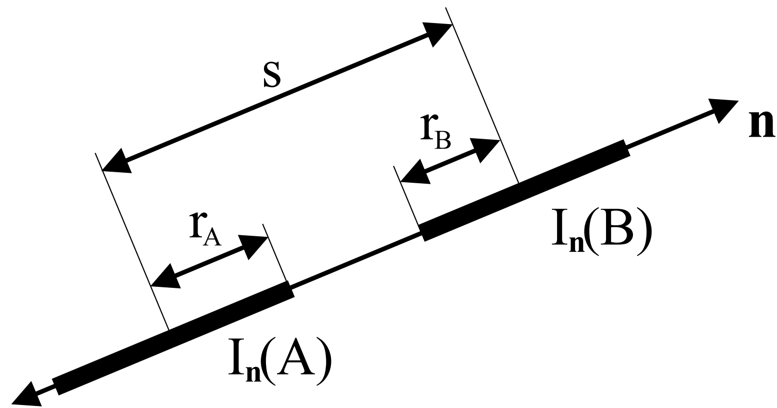


Figure 4.6: Two intervals are disjoint if and only if the separation of their midpoints is greater than the sum of their half-widths (radii). The relevant comparison is $s > r_A + r_B$.

of a box onto the axis and summing the lengths of their images. We compute the half-width of an interval by summing the lengths of the images of the box axes, as shown in Figure 4.5. Finally, note that the intervals of A and B are disjoint iff their half-lengths add up to less than the distance separating their midpoints. We will call the half-length of A its *radius* (it is like a one-dimensional disk), denoted r_A, and likewise for B, and the distance between their midpoints is s, for separation. Then the box intervals are disjoint under axial projection iff

$$s > r_A + r_B$$

as shown in Figure 4.6.

Now consider the representation for our boxes: let the center of box A be at location $\mathbf{T}^A$, with three axes given by unit vectors $\mathbf{R}_0^A$, $\mathbf{R}_1^A$, and $\mathbf{R}_2^A$, and the edge half-lengths of A be given by real numbers a_0, a_1, and a_2. Similarly, let the center of B be at $\mathbf{T}^B$, and whose axes are given as unit vectors $\mathbf{R}_0^B$, $\mathbf{R}_1^B$, and $\mathbf{R}_2^B$, and dimensions given by b_0, b_1, and b_2.

Now, given an axis through the origin with direction $\mathbf{n}$ (not necessarily unit length), we want to project these two boxes A and B onto it and decide if the images under axial projection are disjoint.

The center of A projects onto the axis at a signed distance $\mathbf{T}^A \cdot \mathbf{n}$ from the origin. Similarly for B. Thus, the separation between the images of $\mathbf{T}^A$ and $\mathbf{T}^B$ under axial projection is

$$s = |\mathbf{T}^A \cdot \mathbf{n} - \mathbf{T}^B \cdot \mathbf{n}|/|\mathbf{n}| = |(\mathbf{T}^A - \mathbf{T}^B) \cdot \mathbf{n}|/|\mathbf{n}|$$

The half-length (or radius) of the interval formed by A under axial projection is given by the sum of the images under axial projection of the half-lengths of three orthogonal edges of A. So,

$$r_A = (a_1|\mathbf{R}_1^A \cdot \mathbf{n}| + a_2|\mathbf{R}_2^A \cdot \mathbf{n}| + a_3|\mathbf{R}_3^A \cdot \mathbf{n}|)/|\mathbf{n}|$$

and likewise,

$$r_B = (b_1|\mathbf{R}_1^B \cdot \mathbf{n}| + b_2|\mathbf{R}_2^B \cdot \mathbf{n}| + b_3|\mathbf{R}_3^B \cdot \mathbf{n}|)/|\mathbf{n}|$$

The final test is whether

$$|(\mathbf{T}^A - \mathbf{T}^B) \cdot \mathbf{n}|/|\mathbf{n}| >$$
$$(a_1|\mathbf{R}_1^A \cdot \mathbf{n}| + a_2|\mathbf{R}_2^A \cdot \mathbf{n}| + a_3|\mathbf{R}_3^A \cdot \mathbf{n}|)/|\mathbf{n}| +$$

$$(b_1|\mathbf{R}_1^B \cdot \mathbf{n}| + b_2|\mathbf{R}_2^B \cdot \mathbf{n}| + b_3|\mathbf{R}_3^B \cdot \mathbf{n}|)/|\mathbf{n}|$$

but in this case, we can multiply through by $|\mathbf{n}|$. This means it does not inconvenience us if $\mathbf{n}$ is not a unit vector, so long as it is nonzero.

After eliminating the $|\mathbf{n}|$ in the denominator, this amounts to 7 dot products, 7 absolute values, 1 vector subtraction, 6 vector-scalar multiplies, and 4 scalar additions. This totals 39 multiplies, 21 additions/subtractions, and 7 absolute values for one axis test. This is 67 arithmetic operations per axis test, and with 15 axis tests, this would be 1005 arithmetic operations.

4.3.4 Optimization: Structure of Candidate Vector

The dot products with $\mathbf{n}$ has structure which can be exploited to further reduce the operation count. First, recall that the axes we want to test are either face normals of one of the boxes, or perpendicular to an edge from each box (equivalently, perpendicular to a face normal from each box). So, we have three cases: $\mathbf{n} = \mathbf{R}_i^A$, $\mathbf{n} = \mathbf{R}_i^B$, or $\mathbf{n} = \mathbf{R}_i^A \times \mathbf{R}_j^B$, with $i, j \in \{1, 2, 3\}$.

Case 1:

For the sake of example, we will choose $i = 2$, so $\mathbf{n} = \mathbf{R}_2^A$. Substituting $\mathbf{R}_i^A$ for $\mathbf{n}$ in the above axis test,

$$\begin{aligned}
&|(\mathbf{T}^A - \mathbf{T}^B) \cdot \mathbf{R}_2^A| > \\
&\quad (a_1|\mathbf{R}_1^A \cdot \mathbf{R}_2^A| + a_2|\mathbf{R}_2^A \cdot \mathbf{R}_2^A| + a_3|\mathbf{R}_3^A \cdot \mathbf{R}_2^A|) + \\
&\quad (b_1|\mathbf{R}_1^B \cdot \mathbf{R}_2^A| + b_2|\mathbf{R}_2^B \cdot \mathbf{R}_2^A| + b_3|\mathbf{R}_3^B \cdot \mathbf{R}_2^A|)
\end{aligned}$$

and noting that the axes of A are mutually perpendicular and of unit length, most of the terms for r_A vanish and we get

$$|(\mathbf{T}^A - \mathbf{T}^B) \cdot \mathbf{R}_2^A| > a_2 + (b_1|\mathbf{R}_1^B \cdot \mathbf{R}_2^A| + b_2|\mathbf{R}_2^B \cdot \mathbf{R}_2^A| + b_3|\mathbf{R}_3^B \cdot \mathbf{R}_2^A|)$$

For this case, when $\mathbf{n}$ is a face normal of A, the operation count is 4 absolute values, 15 multiplies, 6 addition/subtractions.

Case 2:

Again, we choose $i = 2$ for this example, but it should be clear how other box

axes of B will simplify. Substituting R_i^B for $\mathbf{n}$ in the above axis test,

$$\begin{aligned}
&|(\mathbf{T}^A - \mathbf{T}^B) \cdot \mathbf{R}_2^B| > \\
&\quad (a_1|\mathbf{R}_1^A \cdot \mathbf{R}_2^B| + a_2|\mathbf{R}_2^A \cdot \mathbf{R}_2^B| + a_3|\mathbf{R}_3^A \cdot \mathbf{R}_2^B|) \\
&\quad +(b_1|\mathbf{R}_1^B \cdot \mathbf{R}_2^B| + b_2|\mathbf{R}_2^B \cdot \mathbf{R}_2^B| + b_3|\mathbf{R}_3^B \cdot \mathbf{R}_2^B|)
\end{aligned}$$

and noting that the axes of B are mutually perpendicular and of unit length, then most of the terms for $\mathbf{R}_B$ vanish, as with A, and we get

$$|(\mathbf{T}^A - \mathbf{T}^B) \cdot \mathbf{R}_2^B| > (a_1|\mathbf{R}_1^A \cdot \mathbf{R}_2^B| + a_2|\mathbf{R}_2^A \cdot \mathbf{R}_2^B| + a_3|\mathbf{R}_3^A \cdot \mathbf{R}_2^B|) + b_2$$

When $\mathbf{n}$ is a face normal of B, the comparison has exactly the same form as for case 1, and the operation count is identical: 4 absolute values, 15 multiplies, 6 addition/subtractions.

Case 3:

This case is a little more complex. In this case, n is the cross product of an axis from each box. This example shows how the axis test simplifies for $\mathbf{n} = \mathbf{R}_1^A \times \mathbf{R}_2^B$. Making the substitution, we get

$$\begin{aligned}
&|(\mathbf{T}^A - \mathbf{T}^B) \cdot (\mathbf{R}_1^A \times \mathbf{R}_2^B)| > \\
&\quad (a_1|\mathbf{R}_1^A \cdot (\mathbf{R}_1^A \times \mathbf{R}_2^B)| + a_2|\mathbf{R}_2^A \cdot (\mathbf{R}_1^A \times \mathbf{R}_2^B)| + a_3|\mathbf{R}_3^A \cdot (\mathbf{R}_1^A \times \mathbf{R}_2^B)|) + \\
&\quad (b_1|\mathbf{R}_1^B \cdot (\mathbf{R}_1^A \times \mathbf{R}_2^B)| + b_2|\mathbf{R}_2^B \cdot (\mathbf{R}_1^A \times \mathbf{R}_2^B)| + b_3|\mathbf{R}_3^B \cdot (\mathbf{R}_1^A \times \mathbf{R}_2^B)|)
\end{aligned}$$

Now, we make use of the identity $\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w}) = \mathbf{w} \cdot (\mathbf{u} \times \mathbf{v}) = \mathbf{v} \cdot (\mathbf{w} \times \mathbf{u})$, to rotate the triple product into a form which takes the cross product of two vectors from the same box,

$$\begin{aligned}
&|(\mathbf{T}^A - \mathbf{T}^B) \cdot (\mathbf{R}_1^A \times \mathbf{R}_2^B)| > \\
&\quad (a_1|\mathbf{R}_2^B \cdot (\mathbf{R}_1^A \times \mathbf{R}_1^A)| + a_2|\mathbf{R}_2^B \cdot (\mathbf{R}_2^A \times \mathbf{R}_1^A)| + a_3|\mathbf{R}_2^B \cdot (\mathbf{R}_3^A \times \mathbf{R}_1^A)|) + \\
&\quad (b_1|\mathbf{R}_1^A \cdot (\mathbf{R}_2^B \times \mathbf{R}_1^B)| + b_2|\mathbf{R}_1^A \cdot (\mathbf{R}_2^B \times \mathbf{R}_2^B)| + b_3|\mathbf{R}_1^A \cdot (\mathbf{R}_2^B \times \mathbf{R}_3^B)|)
\end{aligned}$$

Six of the seven cross products simplify: those between different vectors of the same box become the remaining vector from that box, while those between the same vector vanish, to get

$$|(\mathbf{T}^A - \mathbf{T}^B) \cdot (\mathbf{R}_1^A \times \mathbf{R}_2^B)| >$$

$$(a_2| - \mathbf{R}_2^B \cdot \mathbf{R}_3^A| + a_3|\mathbf{R}_2^B \cdot \mathbf{R}_2^A|) + (b_1| - \mathbf{R}_1^A \cdot \mathbf{R}_3^B| + b_3|\mathbf{R}_1^A \cdot \mathbf{R}_1^B|)$$

We can drop the minus signs, since they occur within absolute value operations, so the final form of the test is:

$$\begin{aligned}&|(\mathbf{T}^A - \mathbf{T}^B) \cdot (\mathbf{R}_1^A \times \mathbf{R}_2^B)| > \\ &\quad (a_2|\mathbf{R}_2^B \cdot \mathbf{R}_3^A| + a_3|\mathbf{R}_2^B \cdot \mathbf{R}_2^A|) + (b_1|\mathbf{R}_1^A \cdot \mathbf{R}_3^B| + b_3|\mathbf{R}_1^A \cdot \mathbf{R}_1^B|)\end{aligned}$$

The operation count is 5 absolute values, 19 multiplies. 16 additions/subtractions. Of the 15 axis tests necessary to determine overlap status of two boxes, 3 are of case 1, 3 are of case 2, and 9 are of case 3, totalling 69 absolute values, 261 multiplies, and 180 additions/subtractions.

4.3.5 Optimization: Choice of Coordinate System

If both boxes undergo a rigid transformation, then the overlap status is unchanged, even though the coordinates of their centers and their axes may change. Such a rigid transformation can allow further algebraic simplification to achieve a net savings in the operation count.

Specifically, we select a frame of reference which places box A at the origin and aligns it with the canonical coordinate axes. Treating $\mathbf{R}^A$ as a rotation matrix, we are applying rotation $(\mathbf{R}^A)^{-1} = (\mathbf{R}^A)^T$ followed by translation $-(\mathbf{R}^A)^T\mathbf{T}^A$. After applying this to box A, we have $\mathbf{T}^A = (0,0,0)^T$, and $\mathbf{R}_1^A = (1,0,0)^T$, $\mathbf{R}_2^A = (0,1,0)^T$, and $\mathbf{R}_3^A = (0,0,1)^T$. For box B, we replace $\mathbf{R}^B$ with $(\mathbf{R}^A)^T\mathbf{R}^B$, and $\mathbf{T}^B$ with $(\mathbf{R}^A)^T(\mathbf{T}^B - \mathbf{T}^A)$, which can be done as a preprocessing step.

After having preprocessed box B as above, and applying the simplifying substitutions for A, we can see what algebraic simplifications result in each of the three cases. This coordinate transformation costs one matrix-matrix multiply, one matrix-vector multiply, and one vector-vector subtraction, which amounts to 36 multiplies, 27 additions.

case 1:

$$\begin{aligned}&|((0,0,0)^T - \mathbf{T}^B) \cdot (0,1,0)^T| > \\ &\quad a_2 + (b_1|\mathbf{R}_1^B \cdot (0,1,0)^T| + b_2|\mathbf{R}_2^B \cdot (0,1,0)^T| + b_3|\mathbf{R}_3^B \cdot (0,1,0)^T|)\end{aligned}$$

The dot product with a canonical basis vector simply selects the appropriate coordinate, so all the dot products are replaced by the appropriate elements of the vectors,

$$|\mathbf{T}_2^B| > a_2 + (b_1|\mathbf{R}_{12}^B| + b_2|\mathbf{R}_{22}^B| + b_3|\mathbf{R}_{32}^B|)$$

The total operation count is 4 absolute values, 3 multiplications, 3 additions/subtractions.

case 2:

Making the substitutions, we get

$$\begin{aligned}&|((0,0,0)^T - \mathbf{T}^B) \cdot \mathbf{R}_2^B| > \\ &\quad (a_1|(1,0,0)^T \cdot \mathbf{R}_2^B| + a_2|(0,1,0)^T \cdot \mathbf{R}_2^B| + a_3|(0,0,1)^T \cdot \mathbf{R}_2^B|) + b_2\end{aligned}$$

which becomes

$$|\mathbf{T}^B \cdot \mathbf{R}_2^B| > (a_1|\mathbf{R}_{21}^B| + a_2|\mathbf{R}_{22}^B| + a_3|\mathbf{R}_{23}^B|) + b_2$$

Here we see some asymmetry between case 1 and case 2, because our rigid transformation favored A to make its representation simpler than Bs. The total operation count is 4 absolute values, 6 multiplies, 5 additions/subtractions.

case 3:

$$\begin{aligned}&|(-\mathbf{T}^B) \cdot ((1,0,0)^T \times \mathbf{R}_2^B)| > \\ &\quad (a_2|\mathbf{R}_2^B \cdot (0,0,1)^T| + a_3|\mathbf{R}_2^B \cdot (0,1,0)^T|) + (b_1|(1,0,0)^T \cdot \mathbf{R}_3^B| + b_3|(1,0,0)^T \cdot \mathbf{R}_1^B|)\end{aligned}$$

becomes

$$|\mathbf{R}_{22}^B\mathbf{T}_3^B - \mathbf{R}_{23}^B\mathbf{T}_2^B| > (a_2|\mathbf{R}_{23}^B| + a_3|\mathbf{R}_{22}^B|) + (b_1|\mathbf{R}_{31}^B| + b_3|\mathbf{R}_{11}^B|)$$

So the total count is 5 absolute values, 6 multiplies, 4 addition/subtractions. Again, given that there are 3 of case 1, 3 of case 2, and 9 of case 3, the total operation count is now 69 absolute values, 81 multiplies, and 60 additions/subtractions. When we add in the cost of the rigid coordinate transformation, it becomes 69 absolute values, 117 multiplies, and 87 add/subtracts.

4.3.6 Optimization: Removing Common Subexpressions

When we write out all 15 axis tests, we see that we are taking the absolute value of each of the elements of $\mathbf{R}^B$ in several places. To save more operations, we can compute the absolute values of the elements of $\mathbf{R}^B$ once in advance, and reuse them. In that case, we have one absolute value in each axis test, and nine absolute values as a precomputation, for a total of 24 absolute values (saving 45 absolute values). The final operation count is 24 absolute values, 117 multiplies, 87 adds/subtracts, and 15 comparisons.

4.4 Special Case OBBs

In cases where an OBB is known to have either a thickness of zero or an infinite extent along some axis, the overlap test formulas can be simplified to save arithmetic operations. Zero thickness boxes arise when single polygons or planar collections of polygons are bounded. Infinitely long OBBs may arise in other applications, such as ray tracing of beams or "thick rays" [AK89].

In the case of a zero thickness, the zero can replace the associated half-width parameter which simplifies the formulas. For example, if box B has a zero thickness along its second axis, then the axis test

$$|(-\mathbf{T}_2^B)| > a_2 + (b_1|\mathbf{R}_{12}^B| + b_2|\mathbf{R}_{22}^B| + b_3|\mathbf{R}_{32}^B|)$$

can be simplified to

$$|(-\mathbf{T}_2^B)| > a_2 + (b_1|\mathbf{R}_{12}^B| + b_3|\mathbf{R}_{32}^B|)$$

which saves one add, one multiply, and one absolute value.

On the other hand, if the extent along the second axis is known to be infinite, then this test can be simplified to

$$(\mathbf{R}_{22}^B \neq 0) \qquad \text{or} \qquad |(-\mathbf{T}_2^B)| > a_2 + (b_1|\mathbf{R}_{12}^B| + b_3|\mathbf{R}_{32}^B|).$$

It is likely that the boxes are not perfectly aligned, and that element $\mathbf{R}_{22}^B$ is nonzero, which means that if we have a short-circuit "or" operation, the second term is never evaluated. If this is the case, the entire test simplifies to a single comparison. In the unlikely event that $\mathbf{R}_{22}^B$ is zero, then the second term is still simpler than the original – in fact it is exactly the expression for the zero thickness case.

4.5 Robustness Issues

We now examine a very subtle robustness issue. Although the mathematics of the OBB overlap test is very robust – not subject to conditioning problems or failure on nongeneric inputs such as parallel faces – it can fail in the presence of small errors caused by finite precision machine arithmetic. In this section we present a case study of a bug encountered in the original implementation of the overlap test. This test was implemented according to the formulas thus far presented.

To review, an axis test for candidate axis $\mathbf{n}$ is

$$\begin{aligned}
&|(\mathbf{T}^A - \mathbf{T}^B) \cdot \mathbf{n}|/|\mathbf{n}| > \\
&\quad (a_1|\mathbf{R}_1^A \cdot \mathbf{n}| + a_2|\mathbf{R}_2^A \cdot \mathbf{n}| + a_3|\mathbf{R}_3^A \cdot \mathbf{n}|)/|\mathbf{n}| + \\
&\quad (b_1|\mathbf{R}_1^B \cdot \mathbf{n}| + b_2|\mathbf{R}_2^B \cdot \mathbf{n}| + b_3|\mathbf{R}_3^B \cdot \mathbf{n}|)/|\mathbf{n}|
\end{aligned}$$

If the inequality is true, then the vector $\mathbf{n}$ is a separating axis, and the boxes are disjoint. However, the test becomes undefined when $\mathbf{n} = \mathbf{0}$ due to the division by zero, which is appropriate since there is no interpretation for projection onto a vector of zero length. However, one of our simplifications eliminated the $|\mathbf{n}|$ factor from the denominator of the inequality.

Recall that for any given choice of $\mathbf{n}$ as being one of the 15 candidate axes, we could employ some kind of algebraic simplification. For example, when $\mathbf{n} = \mathbf{R}_2^A \times \mathbf{R}_2^B$, then the term $a_1|\mathbf{R}_1^A \cdot \mathbf{n}|$ became

$$a_1|\mathbf{R}_1^A \cdot \mathbf{n}| = a_1|\mathbf{R}_1^A \cdot (\mathbf{R}_2^A \times \mathbf{R}_2^B)| = a_1|\mathbf{R}_2^B \cdot (\mathbf{R}_1^A \times \mathbf{R}_2^A)| = a_1|\mathbf{R}_2^B \cdot \mathbf{R}_3^A|$$

and then recognizing that we could choose a coordinate system aligned with box A to simplify this to $a_1|\mathbf{R}_{23}^B|$. All direct references to the length of $\mathbf{n}$ are no longer present.

Suppose we are given the following inputs to the the axis test: the rotation of B with respect to A is

$$R^B = \begin{pmatrix} -0.0641566 & 0 & -0.99794 \\ 0 & 1 & 0 \\ -0.99794 & 0 & 0.0641566 \end{pmatrix}$$

and its center is at

$$T^B = \begin{pmatrix} -0.147256 \\ 1.76777 \\ 1.80947 \end{pmatrix}$$

and the box dimensions are

$$a = \begin{pmatrix} 3.53553 \\ 1.76777 \\ 0 \end{pmatrix}$$

and

$$b = \begin{pmatrix} 2.33155 \\ 0.565685 \\ 0.56452 \end{pmatrix}$$

This is the circumstance in which the second axis of B is parallel to the second axis of A, and when $\mathbf{n}$ is the cross product of these two parallel vectors, it becomes the zero vector. Therefore, all the products in

$$\begin{aligned} &|(\mathbf{T}^A - \mathbf{T}^B) \cdot \mathbf{n}| > \\ &\quad (a_1|\mathbf{R}_1^A \cdot \mathbf{n}| + a_2|\mathbf{R}_2^A \cdot \mathbf{n}| + a_3|\mathbf{R}_3^A \cdot \mathbf{n}|) + \\ &\quad (b_1|\mathbf{R}_1^B \cdot \mathbf{n}| + b_2|\mathbf{R}_2^B \cdot \mathbf{n}| + b_3|\mathbf{R}_3^B \cdot \mathbf{n}|) \end{aligned}$$

yield zero and the comparison simply becomes

$$0 > 0$$

which is trivially false. The optimized expression

$$|\mathbf{T}_1^B\mathbf{R}_{32}^B - \mathbf{T}_3^B\mathbf{R}_{12}^B| > a_1|\mathbf{R}_{32}^B| + a_3|\mathbf{R}_{12}^B| + b_1|\mathbf{R}_{23}^B| + b_3|\mathbf{R}_{21}^B|$$

also evaluates to

$$0 > 0$$

since all the elements of $\mathbf{R}^B$ which are being referenced are zero.

So, when two axes are parallel, the mathematics of the test reduces to a trivial false, thus essentially removing that axis test from the battery of 15 tests. This is desirable, since the vector $\mathbf{n}$ is the zero vector in this situation, so using it as a candidate axis is not a well-defined operation.

This trivial false result is mathematically guaranteed, so one could claim that degeneracies and nongeneric input cannot foil the algorithm. However, invalid input which might arise from arithmetic error can foil the algorithm, as we show next. Consider the matrix

$$\mathbf{R}^B = \begin{pmatrix} -0.0641566 & -5.54743 \times 10^{-16} & -0.99794 \\ 1.54303 \times 10^{-17} & 1 & -2.22883 \times 10^{-16} \\ -0.99794 & 6.41346 \times 10^{-20} & 0.0641566 \end{pmatrix}$$

This matrix is not quite orthonormal, as any rotation matrix should be. This is an actual matrix which was encountered in practice as the product of three rotation matrices.

The expression

$$|\mathbf{T}_1^B \mathbf{R}_{32}^B - \mathbf{T}_3^B \mathbf{R}_{12}^B| > a_1|\mathbf{R}_{32}^B| + a_3|\mathbf{R}_{12}^B| + b_1|\mathbf{R}_{23}^B| + b_3|\mathbf{R}_{21}^B|$$

now evaluates to (using IEEE 64-bit double precision floating point)

$$1.00378137201 \times 10^{-15} > 5.28600321408 \times 10^{-16}$$

which is true and consequently the axis returns a "disjoint" result for the box overlap test, even though the boxes are significantly interpenetrating along the y-axis of box A.

The right hand side of the inequality should be larger when the boxes interpenetrate. But the nearly parallel axes cause the cross product to be very small, which in turn causes the values on either side of the inequality to be proportionally shrunk (because the optimized expressions do not divide by $|\mathbf{n}|$ to compensate for non-unitary $\mathbf{n}$). When the values on either side of the inequality are so small, the error introduced into the rotation matrix has a relatively larger influence – enough to tip the balance of the inequality.

Our solution was to add a small number – we chose 0.000001 – to the absolute values of the matrix elements in the right hand side of all the inequalities. Since the error appeared in the axes tests where only the zero elements (or near zero elements) of the matrix were being used, adding 0.000001 amounts to an enormous increase in the right-hand side. This is an enormous increase only in those situations where both sides would otherwise be nearly zero. Consequently, this has the effect of making axis tests corresponding to nearly parallel edges disproportionately conservative, which is

desirable. Note that this addition of a constant is still scale-invariant: it is being added to the components of the rotation matrix, which range between -1 and 1, regardless of the scale of the model.

4.6 Program Code

The following is C++ program code which implements the overlap test described above. It does not contain the special checks to improve performance for zero or infinite box widths. It does include the modification to make it robust against almost orthonormal "rotation" matrices. It assumes that box B is expressed with respect to box A.

```
// The following routine tests whether two boxes are overlapping,
// returning 1 if they touch, and zero if they do not.
// It assumes that B,T are rotation and translation which describes
// the second box in relation to the first.  The arrays a and b
// are the dimensions of the first and second boxes, respectively.
//
// Stefan Gottschalk, December, 1999.
// stefan@cs.unc.edu
// University of North Carolina at Chapel Hill
// Chapel Hill, NC, 27599
//

#define TESTCASE1(x) \
      (fabs(T[x]) > \
      (a[x] + b[0] * Bf[0][x] + b[1] * Bf[1][x] + b[2] * Bf[2][x]))

#define TESTCASE2(x) \
      (fabs(T[0]*B[0][x] + T[1]*B[1][x] + T[2]*B[2][x]) > \
      (b[x] + a[0] * Bf[0][x] + a[1] * Bf[1][x] + a[2] * Bf[2][x]))

#define TESTCASE3(i,j) \
      (fabs(T[(i+2)%3] * B[(i+1)%3][j] - T[(i+1)%3] * B[(i+2)%3][j]) > \
       ( a[(i+1)%3] * Bf[(i+2)%3][j] + a[(i+2)%3] * Bf[(i+1)%3][j] + \
```

```
        b[(j+1)%3] * Bf[i][(j+2)%3] + b[(j+2)%3] * Bf[i][(j+1)%3] ) )

int
obb_overlap(double B[3][3], double T[3], double a[3], double b[3])
{
  const double eps = 1e-6;  // to counteract effects of arithmetic error
  double Bf[3][3];

  Bf[0][0] = fabs(B[0][0]) + eps;
  Bf[0][1] = fabs(B[0][1]) + eps;
  Bf[0][2] = fabs(B[0][2]) + eps;
  Bf[1][0] = fabs(B[1][0]) + eps;
  Bf[1][1] = fabs(B[1][1]) + eps;
  Bf[1][2] = fabs(B[1][2]) + eps;
  Bf[2][0] = fabs(B[2][0]) + eps;
  Bf[2][1] = fabs(B[2][1]) + eps;
  Bf[2][2] = fabs(B[2][2]) + eps;

  // CASE 1: (three of them)
  if TESTCASE1(0) return 0;
  if TESTCASE1(1) return 0;
  if TESTCASE1(2) return 0;

  // CASE 2: (three of them)
  if TESTCASE2(0) return 0;
  if TESTCASE2(1) return 0;
  if TESTCASE2(2) return 0;

  // CASE 3: (nine of them)
  if TESTCASE3(0,0) return 0;
  if TESTCASE3(1,0) return 0;
  if TESTCASE3(2,0) return 0;
  if TESTCASE3(0,1) return 0;
  if TESTCASE3(1,1) return 0;
  if TESTCASE3(2,1) return 0;
```

```
  if TESTCASE3(0,2) return 0;
  if TESTCASE3(1,2) return 0;
  if TESTCASE3(2,2) return 0;

  return 1;
}
```

4.7 Summary of Chapter

We presented a new overlap test for OBBs based on the separating axis theorem, which requires as many as 15 axis tests. The separating axis theorem can apply to arbitrary convex polytopes, but it is especially well-suited for OBBs. The test uses no complex data structures and simple flow of control, which makes it a good candidate for microcoding. Parallel features and degenerate inputs need not be handled as special cases. The final tally of arithmetic operations is shown in Table 5.1. Including a coordinate transformation and operations to guard against arithmetic error, the worst case input causes the test to execute 252 arithmetic operations, while the best case input requires only 89. If one can pose the problem as one OBB expressed in the coordinate system of the other, then 63 of the operations can be omitted. In Section 5.2.1 we demonstrate that this test is approximately 10 times faster than the prior state of the art.

Test Step	*Total Ops*	*Compares*	*Add/Subs*	*Mults*	*Abs. Vals.*
Coordinate Transform	63		27	36	
Arithmetic Error Guard	9		9		
Common Absolute Values	9				9
Case 1 Test: $\mathbf{n} = \mathbf{A}^1$	8	1	3	3	1
Case 1 Test: $\mathbf{n} = \mathbf{A}^2$	8	1	3	3	1
Case 1 Test: $\mathbf{n} = \mathbf{A}^3$	8	1	3	3	1
Case 2 Test: $\mathbf{n} = \mathbf{B}^1$	13	1	5	6	1
Case 2 Test: $\mathbf{n} = \mathbf{B}^2$	13	1	5	6	1
Case 2 Test: $\mathbf{n} = \mathbf{B}^3$	13	1	5	6	1
Case 3 Test: $\mathbf{n} = \mathbf{A}^1 \times \mathbf{B}^1$	12	1	4	6	1
Case 3 Test: $\mathbf{n} = \mathbf{A}^1 \times \mathbf{B}^2$	12	1	4	6	1
Case 3 Test: $\mathbf{n} = \mathbf{A}^1 \times \mathbf{B}^3$	12	1	4	6	1
Case 3 Test: $\mathbf{n} = \mathbf{A}^2 \times \mathbf{B}^1$	12	1	4	6	1
Case 3 Test: $\mathbf{n} = \mathbf{A}^2 \times \mathbf{B}^2$	12	1	4	6	1
Case 3 Test: $\mathbf{n} = \mathbf{A}^2 \times \mathbf{B}^3$	12	1	4	6	1
Case 3 Test: $\mathbf{n} = \mathbf{A}^3 \times \mathbf{B}^1$	12	1	4	6	1
Case 3 Test: $\mathbf{n} = \mathbf{A}^3 \times \mathbf{B}^2$	12	1	4	6	1
Case 3 Test: $\mathbf{n} = \mathbf{A}^3 \times \mathbf{B}^3$	12	1	4	6	1
Best Case (exit after first test)	89	1	39	39	10
Worst Case (execute all tests)	252	15	96	117	24

Table 4.1: A summary of the costs of various steps in the OBB overlap test based on separating axes. If one of the boxes is already expressed in terms of the coordinate system of the other, then the first 63 arithmetic operations can be skipped. If one is not concerned about guarding against arithmetic error, the second 9 operations can be skipped. If the first axis test proves the OBBs disjoint, then only 89 arithmetic operations are performed (including transform and error guard steps). If all the tests are executed, then 252 arithmetic operations are performed.

Chapter 5

Performance and Analysis of Collision Queries

The results we present in this chapter are measurements of the fast OBB overlap test, and the performance characterization of various BV types in certain special case collision scenarios. We show three specific results. First, the overlap test based on separating axes is an order of magnitude faster than the prior state of the art (Section 5.2.1). Second, OBBs asymptotically outperform spheres and AABBs as we reduce the gap size in *parallel close proximity* situations (Section 5.3.2.3). Third, all three BV types exhibit asymptotically optimal performance with respect to tessellation density in transverse contact situations (Section 5.3.3.1, using the lower bounds result from Section 2.6.3).

Section 5.1 presents the *cost equation*, which expresses the cost of a query as a linear combination of its fundamental operations. Section 5.2 presents the experimental measurements of each of the BV overlap tests. Section 5.3 describes a series of benchmarks we performed, and analyzes their results concerning asymptotic performance of different bounding volumes.

5.1 Query Cost Equation

A collision query consists of a series of BV tests and possibly polygon tests. It is helpful to model the query time as a linear function of the number of such tests, as expressed in the following equation,

$$T = T_v N_v + T_p N_p$$

where T_v is the time for one BV test, N_v is the number of BV tests made during the query, T_p is the time for one polygon test, and N_p is the number of polygon tests. We assume update caching is not being performed (update caching is discussed in Section 2.3.4), and we assume that the cost of performing updates of BVs (whether one or two per BV test is being performed) is included in T_v. We ignore the query setup time, which is a constant independent of input size or output size. We have observed that setup time is negligible for all but the most trivial queries.

5.2 Speed of Fundamental Tests

5.2.1 Comparison of OBB Overlap Tests

Here we compare an implementation of our OBB overlap test with implementations of methods based on Seidel's randomized linear programming (LP) algorithm [Sei90] and the GJK [GJK88] algorithm. Michael E. Hohmeyer is the author of the randomized linear programming software. The implementation of GJK was obtained from Sean Quinlan.

The speed of each OBB overlap method is shown in Table 5.2.1. These are average times obtained from OBB test pairs taken from actual collision queries on highly tesselated sphere models. The OBB test pairs were divided into two lists: one containing the disjoint pairs, and one containing the overlapping pairs. We then measured how long it took to execute all the OBB overlap tests in each list using the three methods, and reported the average OBB overlap test times in table Table 5.2.1.

This measurement approach ensures that the OBBs being tested are representative of those encountered during collision queries, and it eliminates the BVH traversal time from the OBB overlap test measurements.

These times include the computations needed to transform one of the OBBs into the coordinate system of the other (the *update*, described in Section 2.3). For the LP and GJK methods, some additional computation was needed to convert the OBB representation into the form needed by the method: the LP algorithm requires a set of half-spaces, while the GJK algorithm requires the vertices of the OBBs.

These tests have variable execution times, depending on the specific arrangement of the input OBBs. For example, the separating axis test is a series of 15 trivial rejection tests. If the OBBs overlap, all 15 tests must be performed before we conclude the OBBs are touching. If the OBBs are disjoint, then one of the tests will have triggered

Method used	*Disjoint (μsecs)*	*Intersecting (μsecs)*
Separating axis	5.09	7.44
GJK	70.0	62.5
Linear programming	202	232

Table 5.1: Timing of OBB overlap tests. The benchmarks were run on an SGI Maximum Impact with a 250 MHz MIPS R4400 CPU, MIPS r4000 FPU, 192MB RAM, 2MB secondary cache.

Method used	*Disjoint (μsecs)*	*Intersecting (μsecs)*
Sphere	0.97	0.92
AABB	1.36	1.38
OBB (Separating Axis)	5.09	7.44

Table 5.2: Timing of overlap tests for different BV types. The benchmarks were run on an SGI Maximum Impact with a 250 MHz MIPS R4400 CPU, MIPS r4000 FPU, 192MB RAM, 2MB secondary cache.

an early exit, yield a result of "disjoint". The times are significantly different, as we can see from Table 5.2.1.

The timings show that the separating axis method is, on average, approximately 10 times faster than the GJK method, and approximately 35 times faster than the linear programming method. It should be noted that the separating axis implementation is finely tuned and optimized specifically for OBBs, while the other two methods are extremely general. One would expect the margin to narrow significantly if the other two methods received the same attention to specialization as the separating axis method. However, we do not believe that any amount of specialization will make them faster than the separating axis method.

5.2.2 Comparison of Overlap Tests for BV Types

In this section we examine the speeds of the overlap tests for OBBs, spheres, and AABBs. The timing data was obtained in exactly the same manner as for the OBB method comparisons in the previous section: by performing collision queries on tessellated sphere models, but here we used BVHs composed of OBBs, of spheres, or of AABBs, and compared the BV overlap times in each category. The times are shown in Table 6.2. As expected, the tests for the simpler BV types are faster than those for OBBs.

Lesser Query Complexity Greater

Fewer ← Polygons → More
Further apart ← Proximity → Closer Together
Lesser ← Shape Complexity → Greater

Figure 5.1: Polygon count, geometric complexity, and geometric proximity all contribute to the work of processing a query.

x=0.0 parallel surfaces
x=0.2 first contact
x=1.46 transverse contact
x=1.8 parting contact
x=2.0 well separated

Figure 5.2: Schematic of simple benchmark: one stationary and one moving tessellated sphere.

5.3 Collision Query Benchmarks and Analysis

Practitioners and researchers generally agree that many factors contribute to the work required to perform a collision query. It is evident that larger models (in terms of polygon count) are generally more expensive to query against than smaller models. It is also evident that given the same number of polygons, models which have complex geometry are more challenging than those with simple geometry – for example, a 10k polygon tessellated sphere usually is simpler than a 10k polygon system of pipes. Also, proximity queries take more work when the models are close than when they are widely separated. These tendencies are diagrammed in Figure 5.1.

These factors – polygon count, geometric complexity, and geometric proximity – contribute to the work required for a query in complex ways. One goal of this chapter is to partially untangle their dependencies upon one another and to explain their respective contributions to BVH-based collision queries for a limited class of situations.

5.3.1 Sliding Sphere Benchmark

Consider a very simple benchmark: collision queries on two tessellated spheres. The purpose of the benchmark is to demonstrate that OBBs have the greatest advantage

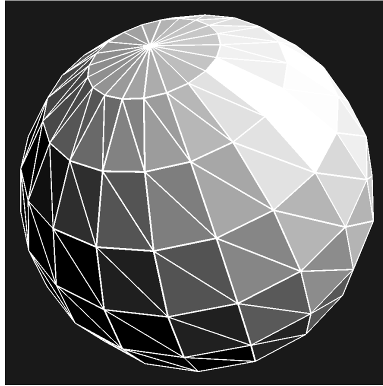


Figure 5.3: A sphere of 90 triangles, tessellated by latitude and longitude. The spheres used in the benchmark each have 40,000 triangles.

over the simpler BV types in grazing and near grazing situations. The benchmark is diagrammed in Figure 5.2. Each sphere has approximately 40k triangles, tesselated in the familiar latitude/longitude manner. This style of tessellation is shown in Figure 5.3 – although the spheres used in our benchmark have 40,000 triangles. One sphere is of unit radius and is fixed at the origin. The other sphere has radius 0.8 units. We chose spheres because they have a very simple shape, intending to introduce as few complicating factors as possible. The smaller sphere begins at the origin, and travels along the x-axis in increments of 0.002 units until its center reaches $x = 2$. At each of these 1001 positions, the two spheres are given random orientation, and a collision query is performed which finds all the contact pairs between them (a *contact pair* is a pair of touching triangles). For each query, we recorded the number of BV tests required, number of triangle tests, and number of contact pairs. In Figure 5.4 we show the plot of the number of triangle contact pairs as a function of the x-coordinate of the smaller sphere.

The general trend of the plot is to rise sharply at $x = 0.2$, peaking at approximately $x = 0.6$, descending to zero at $x = 1.8$. The two ideal spheres intersect in a circle centered around the x-axis, and whose circumference is a function of x:

$$c(x) = 2\pi r(x) = 2\pi \sqrt{(0.8)^2 - (\frac{(1^2 - (0.8)^2 - x^2}{2x})^2}$$

The dashed line in Figure 5.4 which passes through the cloud of points is the circumference function magnified by 70, demonstrating that the number of contacts tends to be proportional to the length of the intersection curve. In this case, the proportionality constant is 70, indicating that the piecewise linear curve which is the intersection

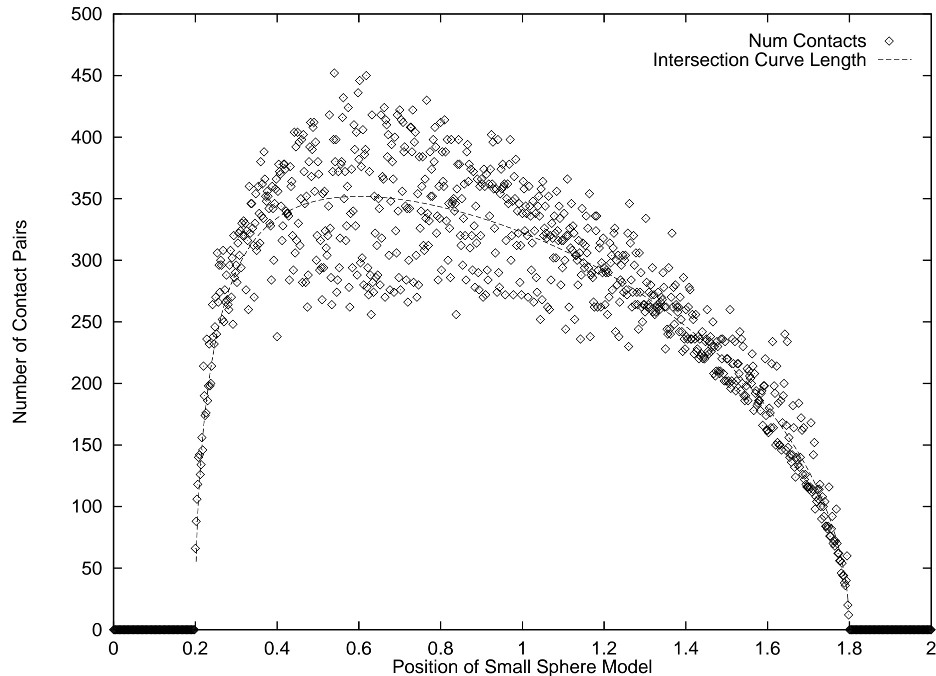


Figure 5.4: Number of contacts found as sphere moves. This shows that the number of contacts is proportional to the length of the intersection curve, plus a random component induced by the random orientations of the sphere models.

of the models contains 70 segments per unit of length. The number of contact pairs has a great deal of random variation caused by the random selection of orientations at each step.

We repeated this experiment using each bounding volume type: OBBs, AABBs, and spheres. The number of contacts for each BV type are identical, but the distributions of the number of bounding volume tests are qualitatively different, which are shown in Figure 5.5. Spheres and AABBs have a very similar rise as the models approach and touch, and both have a sharp peak at approximately the same point at x = 0.25, after which there is a sudden drop. This slope gradually begins to level out, until the parting kiss at x = 1.8. Close examination of the plot shows that these two distributions overlap a little, but the points for spheres are generally above the points for AABBs, implying that AABBs are slightly more efficient than spheres at pruning the bounding volume test tree. The OBB plot also rises sharply where the models first meet, but the rise begins later, and levels off very quickly, never forming a sharp peak. The fact that so many fewer BV tests are required by OBBs demonstrates their superior pruning power.

Figure 5.6 shows the number of faces tested. These plots have a form very similar

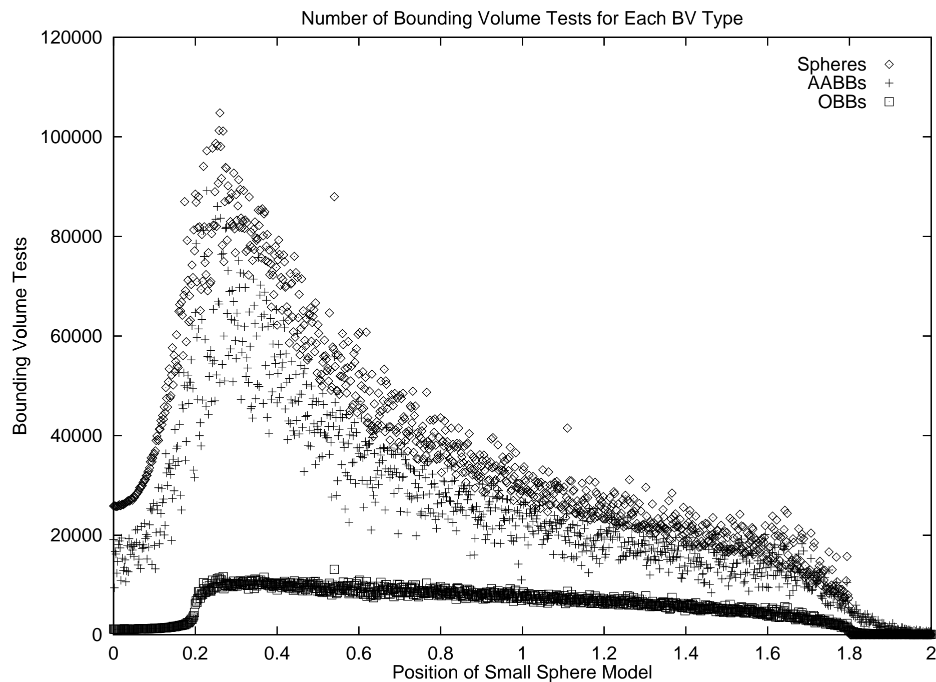


Figure 5.5: Number of BVs tested as sphere moves. This distinguishes OBB from AABBs and spheres, showing that OBBs require many fewer overlap tests than the others, especially where the models first make contact. This means that OBBs have more pruning power than spheres and AABBs.

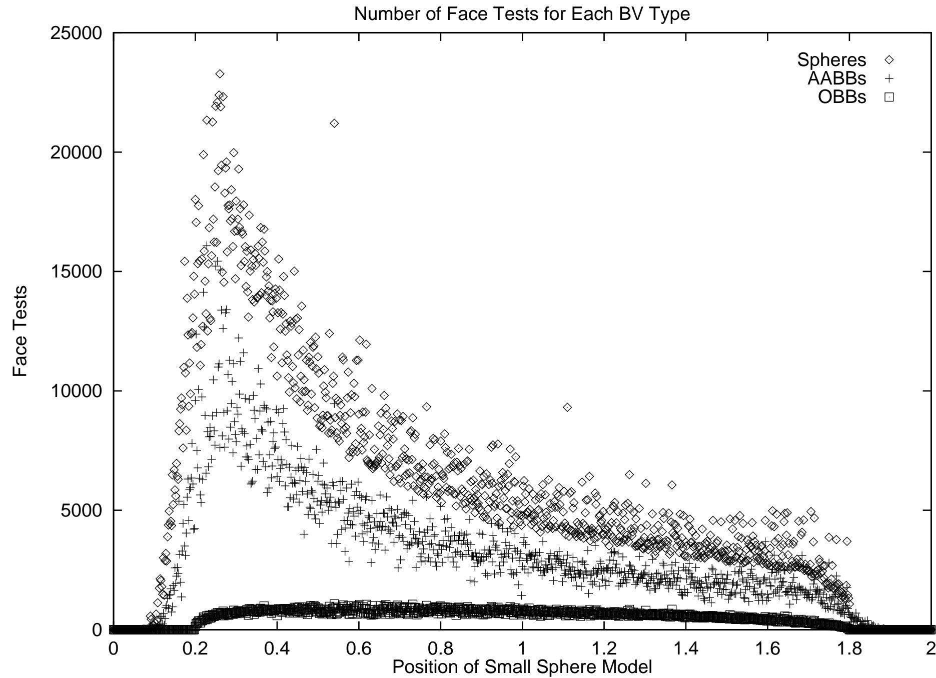


Figure 5.6: Number of primitives tested as sphere moves. Because of the superior pruning power of OBBs, we descend less often to the leaf nodes of the bounding volume hierarchies, and therefore test fewer primitives.

to their BV counterparts, although we see a better defined gap between the AABB and sphere distributions. For all BV types, the number of triangle tests is zero for very small values of x. The triangle tests become nonzero at different values of x for different BV types: first for spheres at approximately $x = 0.1$, then AABBs shortly thereafter, and finally for OBBs at just below $x = 0.2$. This shows that the superior pruning power of OBBs also means testing fewer primitives when using OBBs than when using spheres or AABBs.

Fewer BV tests are required by OBBs than by AABBs and spheres. Figure 5.7 shows the ratio of the former with each of the latter two.

OBBs have the greatest advantage just prior to contact at $x = 0.2$, and just after contact at $x = 1.8$. In the range where there is contact, the ratiosdecline to between approximately 2 and 9, and there is a very gentle bow with a low point near $x = 1.30$. This suggests that BVH-based queries are operating in qualitatively different regimes when there is contact than when there is not. The overall conclusion is that OBBs prune the BVTT better than AABBs and spheres throughout range of this experiment, and OBBs perform best (relative to the other BVs) when the tessellated spheres are very close but not touching.

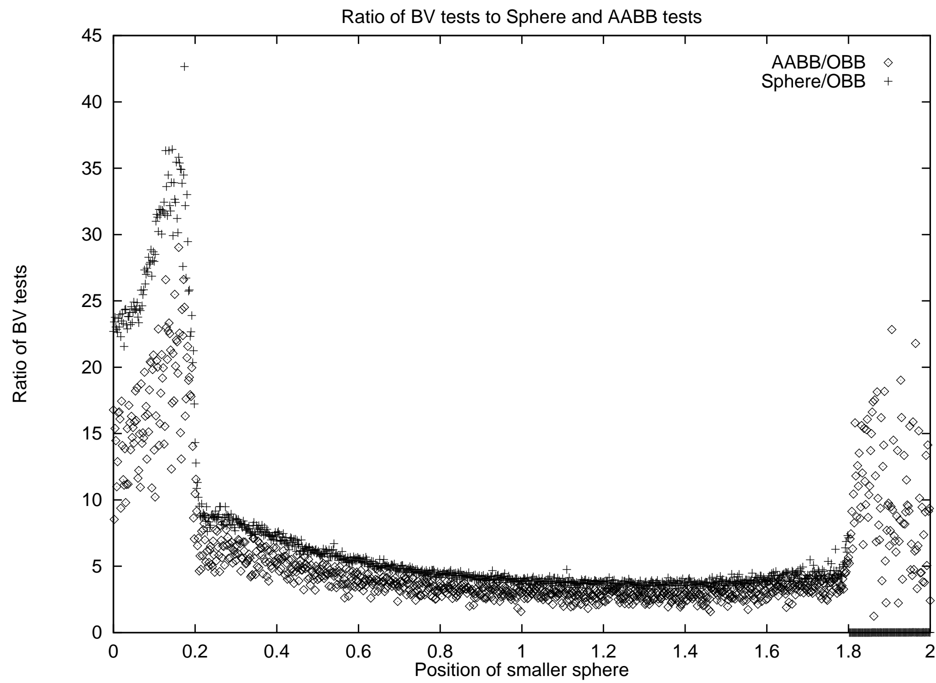


Figure 5.7: Ratio of AABB tests and sphere tests over OBB tests. This shows how much greater pruning power OBBs have over AABBs and spheres. It shows that OBBs require approximately 30 times fewer BV tests than spheres just prior to contact, and 25 times fewer than AABBs. Where there is contact, OBBs appear to have 2 to 9 times the pruning power of the other two, depending on where the sphere models are.

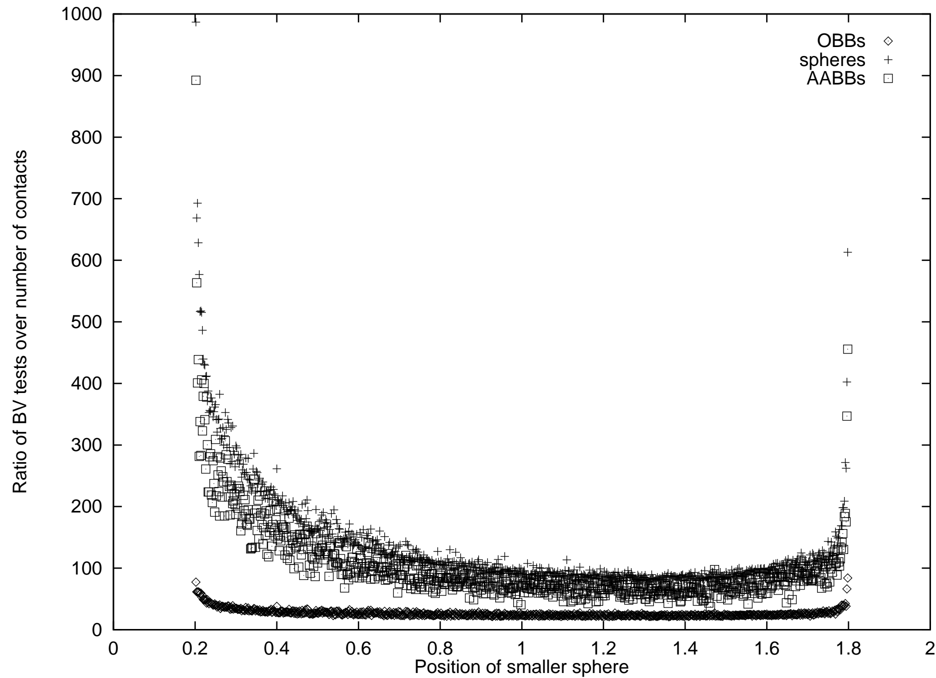


Figure 5.8: *Contact cost* as a function of position of smaller sphere. The contact cost is the number of BV tests divided by the number of contacts found. This normalizes for the size of the query's output, measuring the efficiency of the query.

We can also look at the *contact cost* for each of the BV types. The contact cost for a query is defined as the number of BV tests divided by the number of contacts found. If we think of every contact found as a unit of output, then the contact cost expresses effort per unit output, which is a measure of efficiency. Contact cost is useful because it normalizes for variations in output size. Contact cost is plotted against position of the moving sphere model in Figure 5.8. It is valid only for $0.2 < x < 1.8$, of course, since the number of contacts is zero outside that range. We see that the contact cost is highest at the ends of the range, where the two surfaces approach tangency.

5.3.2 Parallel Close Proximity

The initial configuration of concentric spheres is the simplest nontrivial one to analyze. An example of this is shown in Figure 5.9. We call this *parallel close proximity*, so named because the opposing surfaces are parallel to one another, and every point on each surface is exactly a given ϵ distance from the other surface. This definition applies to the ideal spheres which are approximated by the polygons of the models. The "close" in "parallel close proximity" does not contribute meaningfully to the

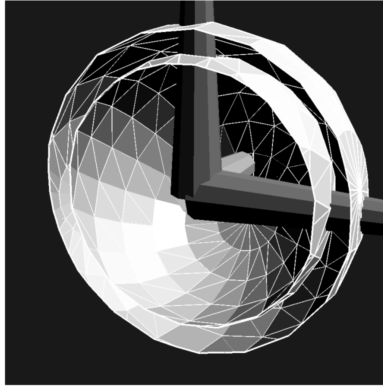


Figure 5.9: Cutaway view of parallel close proximity configuration, with coordinate axes included for reference.

definition, but this configuration is generally most interesting as ϵ diminishes – i.e. as the parallel surfaces come into closer proximity to one another.

This is one of the most challenging configurations that collision detection systems typically encounter. Modeling applications in which parts are tightly fitted are likely to produce parallel close proximities. Also, applications which simulate physical dynamical systems, invoking collision responses as appropriate, give rise to many grazing contact or near contact situations as objects bounce off or come to rest against one another. For such situations, selected portions of the models approximate parallel close proximity.

For this type of configuration, the amount of work to process a collision query is a sensitive function of the gap between the surfaces. The closer the surfaces are together, the more finely they must be approximated in order to bound them apart, requiring us to descend the BVHs of both models more deeply. Because every point on each model is close to the other model, a reduction in the gap requires a query to descend to a greater depth – across the entire tree, not just a narrow portion of it – and every additional level descended doubles the number of nodes visited.

Note that a uniform separation distance does not imply that the trees are descended to a uniform depth. Some BV types, such as AABBs, approximate surfaces better or worse depending on the surface orientation. Those portions of the tree which cover an axis-aligned section of the sphere model will need to be descended less deeply than those portions which cover a non-aligned section. However, even with a tree of nonuniform depth, adding an additional level to each existing leaf node doubles the number of nodes in the tree. Furthermore, the convergence properties of AABBs which cover more and less aligned geometry are identical, although the actual

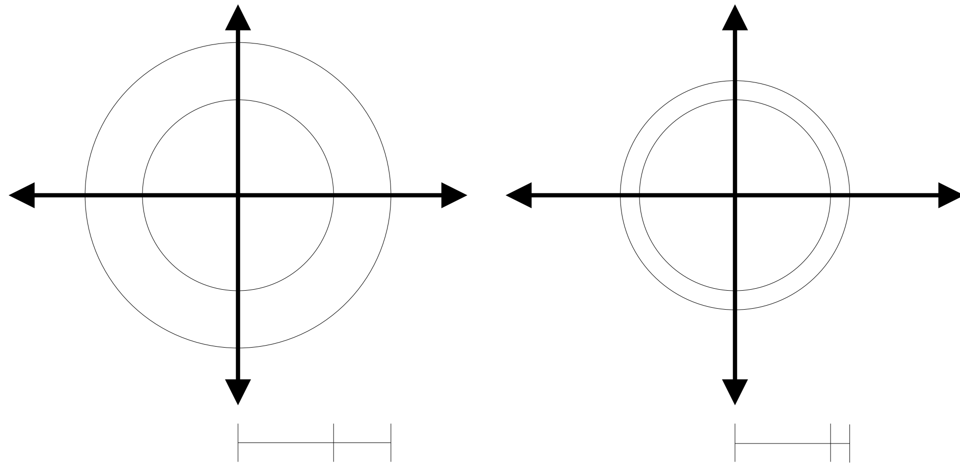


Figure 5.10: Schematic of Parallel Close Proximity experiment.

dimensions of the boxes depend on the alignment. So, the reasoning we employ in the following section applies to trees which have been descended to varying depths, as is the case with AABBs. This assertion is confirmed by the experimental results presented next.

Figure 5.10 shows a schematic diagram of an experiment involving two concentric spheres. In this experiment we placed a 40k polygon sphere of unit radius inside a 40k polygon sphere of radius $1 + \epsilon$. For each choice of ϵ, we performed 100 collision queries on these spheres. For each query, we gave the spheres random orientations. Figure 5.11 shows a plot, for each BV type, of the number of bounding volume tests over a range of surface separations, ϵ. The random orientations caused the number of bounding volume tests to have a random component, but the plot shows the mean for each surface separation of ϵ. Error bars spanning one standard deviation would be imperceptible in this plot, and so were not drawn (the standard deviations were less than 2% of the means for all parts except on the right hand ledge).

The remainder of this section is devoted to explaining why these curves are shaped as they are. Qualitatively, the plots for each type have the same four distinguishable regions, as shown in Figure 5.12. Every curve is level for extremely large and extremely small values of ϵ. These two ledges are two of the four regions of interest. In between, there is a linear downward ramp, which is the third region. Between the ramp and the ledges on either side are areas of constant slope. An interesting feature on all three plots, for which we have no explanation, is the transition from the ramp to the lower ledge, which involves a brief period of even *steeper* descent (more

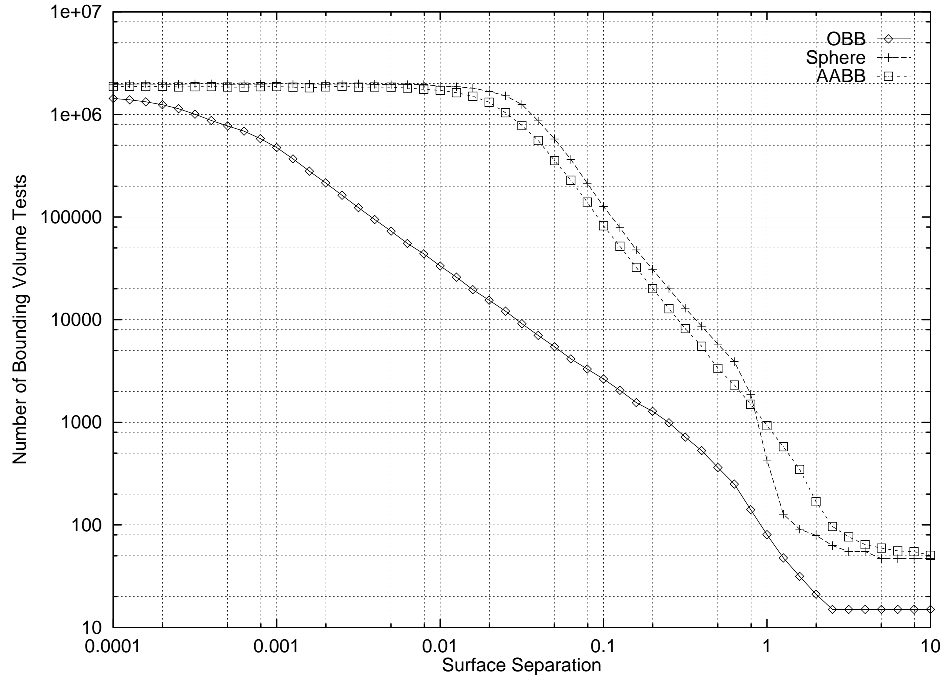


Figure 5.11: Parallel Close Proximity – Number of bounding volume tests versus surface separation (log-log plot).

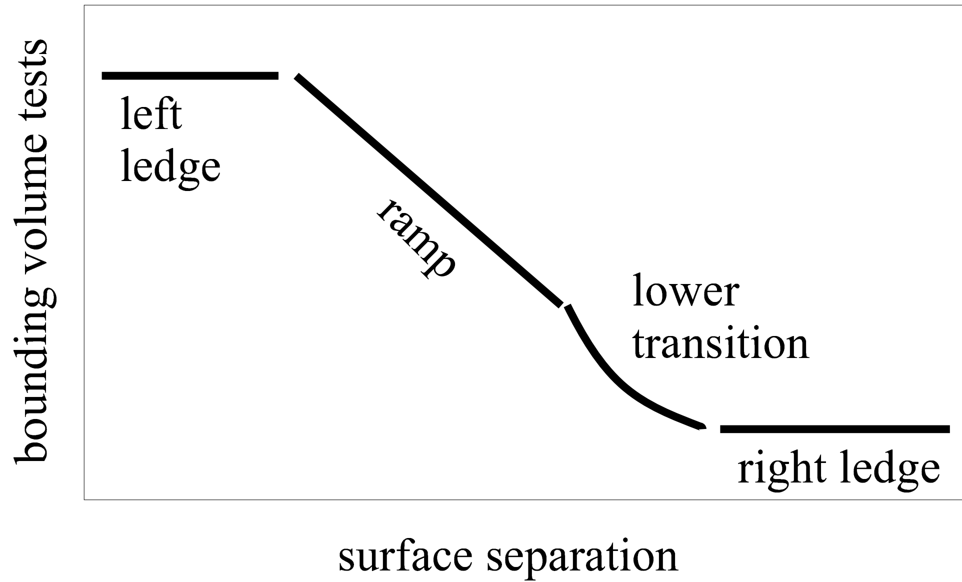


Figure 5.12: Schematic diagram of a typical plot of number of BV tests versus surface separation, for Parallel Close Proximity situations.

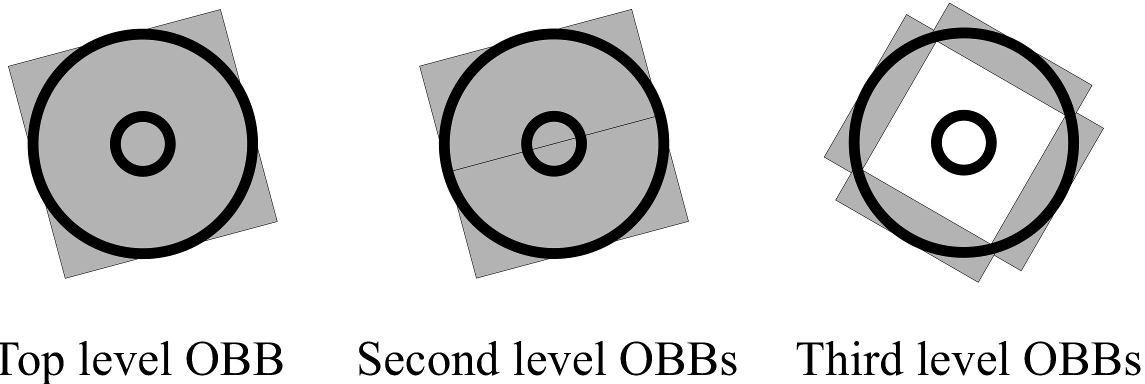


Figure 5.13: Lower ledge in Parallel Close Proximity plot is created by clearance afforded by early levels in the BVHs.

negative slope) before it levels off.

5.3.2.1 Right-Hand Ledge

The right-hand ledge, for extremely large values of ϵ, is level because there is a certain minimal depth to which we must descend the BVH of the outer sphere in order to bound it away from the point at its center. However, once this level is reached, it provides significant clearance for the object in the middle. This effect is diagrammed as a 2D example in Figure 5.13. For 3D OBBs, 15 BV tests is required to obtain clearance, no matter how small the middle sphere becomes relative to the outer one. For each of the other shapes, this minimal number of necessary tests is different.

5.3.2.2 Left-Hand Ledge

The left-hand ledge, for extremely small values of ϵ, is level because a tiny gap brings the leaf nodes of the BVHs into contact. This means that both BVHs have been descended to their leaves, and decreasing ϵ even further cannot cause more BVs to be tested, because there are no additional children to be visited. The height (the y-coordinate representing the number of BV tests) of each ledge is a function of the number of polygons in the models and the packing efficiency of the BV type. Generally speaking, increasing the tessellation will increase the height of the ledge, since that increases the depth of the BVHs. One implication of this is that for "infinitely tessellated model", the ramp would never level off on the left and the ledge would not exist. Of course, ordinary polygonal models have a finite number of polygons, and therefore have finite BVHs. However, models with curved surfaces

(such as NURBS patches) which are tessellated on-the-fly could also generate BVHs on an as-needed basis, and to whatever resolution required. Such generative models could conceivably be used in a manner in which ever decreasing ϵ leads to an ever increasing number of bounding volume tests, without limit.

5.3.2.3 Ramp

The slope of the ramp for OBBs in the log-log plot is measured to be -1.14, whereas the slopes for AABBs and spheres are almost exactly -2. This is interesting for two reasons. First, the slopes are close to small integral values, suggesting a simple mechanism behind the growth of the number of bounding volume tests as a function of ϵ. Second, by virtue of their slopes, the spheres and AABBs are in some sense identical to each other in a way which distinguishes them from the OBBs. Specifically, the growth of OBB tests is asymptotically less than the growth of sphere or AABB tests with diminishing ϵ.

The salient property distinguishing OBBs from the other BV types is that as we descend levels of their respective BVHs, OBBs converge *quadratically* to the underlying geometry, whereas both spheres and AABBs converge *linearly*. To make this statement more precise, let d be the diameter of a BV, and τ be its width (or thickness, if you prefer). The diameter of a set of points is the greatest distance between any two of them. The width of a set of points is the width of the thinnest infinite slab which encloses them. From the definitions, we can see that $d \geq \tau$ for any set of points. Spheres, by nature of their shape, have diameter equal to width. AABBs and OBBs have varying widths relative to their diameters, but generally the further we descend the BVH, the smaller both d and τ become for BVs at a given level. The distinguishing characteristic of OBBs is that τ shrinks quadratically with d as we descend levels, whereas for AABBs and spheres, d and τ shrink proportionally, on average.

When a BV covers a nearly planar patch of geometry, the axis along which τ is measured is generally transverse (if not orthogonal) to the surface. The shrinkage of τ roughly represents a shrinkage of the BVs extent normal to the surface: a reduction in its profile. The reduction of d roughly represents a reduction in the BVs footprint on the surface. The footprint and profile are shown for spheres in Figure 5.14.

As we descend the levels of their respective BVHs, τ for OBBs shrinks faster than does τ for spheres and AABBs, and consequently we do not have to descend as far to bound the models apart when using OBBs. In fact, we will show that trees of

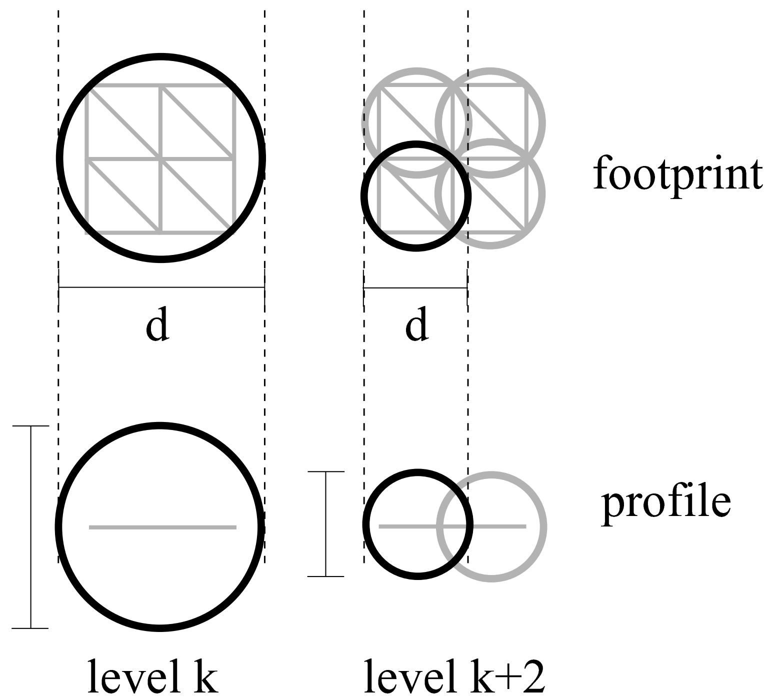


Figure 5.14: Footprint and profile of spherical BVs. The same measures are valid for other BV types.

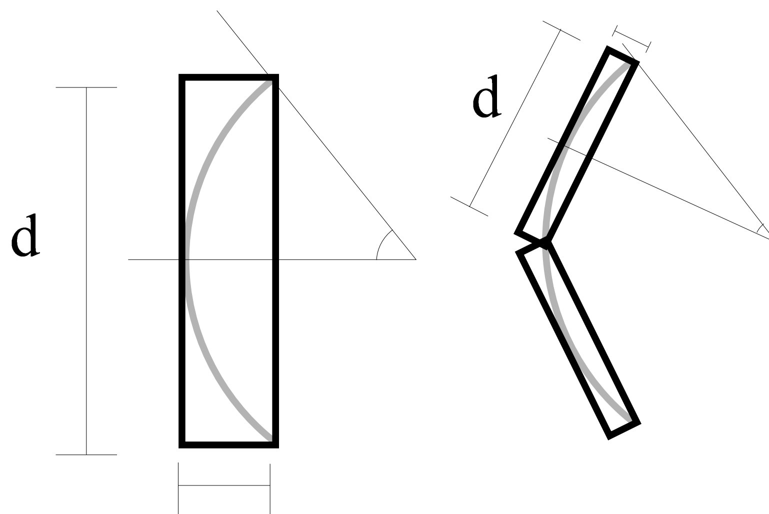


Figure 5.15: Diameter and thickness for OBBs are related by curvature of bounded geometry, which is the basis for their quadratic convergence.

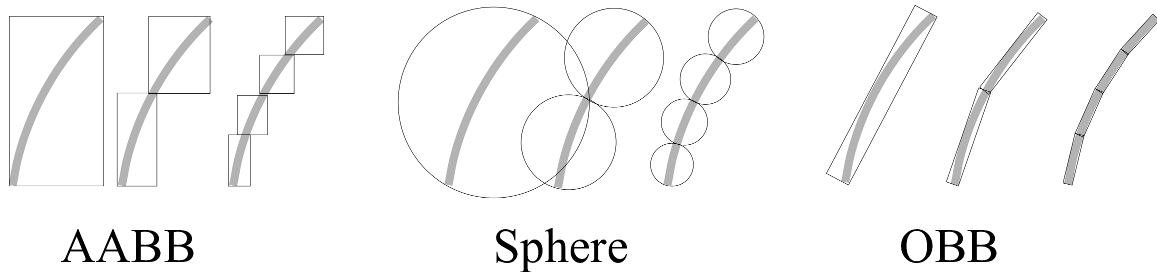


Figure 5.16: OBBs can be seen to converge more rapidly than AABBs and spheres. The aspect ratio of AABB and sphere children resemble that of their parents, while the aspect ratio of OBB children tend to be greater than their parents.

AABBs and spheres must be descended twice as deeply to bound models apart for a given gap, and that this results in an asymptotically greater growth in the number of bounding volume tests as a function of ϵ.

The convergences of AABBs, spheres, and OBBs can be seen graphically in Figure 5.16.

Consider a BV which bounds some approximately planar geometry of given area, A. Its two children together cover that same geometry, and therefore cover the same area. Assuming the BVH is reasonably balanced, each child covers approximately half the area covered by its parent, and with each additional level we descend in the BVH, the area covered by a given BV is reduced by a factor of two. The area covered by each bounding volume l levels down from the original bounding volume is $a(l) = A/2^l$. This bounds the diameter of the geometry covered by each level l bounding volume – the diameter d can be no less than $2\sqrt{A/\pi}$ (for planar geometry, a disc has $d = 2\sqrt{A/\pi}$, and for all other shapes $d > 2\sqrt{A/\pi}$). Since the diameter is proportional to the square root of the area, we have

$$d = O(\sqrt{a(l)}) = O(\sqrt{A/2^l}) = O(2^{-\frac{l}{2}})$$

This simply means that the diameter is approximately halved every two levels we descend in a binary tree, regardless of BV type. This assumed that the geometry was approximately planar, and that the subdivision was reasonably uniform (the uniformity assumption is actually violated at the poles of our spheres, where the triangles become long and thin as tessellation increases – but this violation is confined to the poles).

With spheres and AABBs, τ shrinks proportionally with d, as was depicted in Figure 5.14, so every two levels we descend the BVH causes the thickness of the

approximation to be halved. This means that halving ϵ requires us to descend the BVHs an additional two levels. Since each additional level doubles the number of BVs visited, two additional levels increases the number of BV visited by a factor of four. So a factor 2^{-k} change in ϵ requires an additional $2k$ levels, which leads to a factor 2^{2k} change in the number of BVs visited. Since $2^{2k} = (2^{-k})^{-2}$, the factor change of f in ϵ leads to a factor f^{-2} change in number of bounding volume tests, and on a log-log plot this appear as a shift downward 2 units for every unit rightward: a slope of -2.

A similar analysis attends to OBBs, with the only difference being that τ shrinks as the square of d. Consider an OBB covering a patch of geometry whose radius of curvature is large compared to the size of the OBB (in other words, relatively low curvature), as shown in Figure 5.15. If the curvature is low, then ϕ is small. τ will be proportional to $1 - \cos\phi \approx \phi^2$, and d will be proportional to ϕ. Thus, with successive levels in the hierarchy, as ϕ diminishes, τ shrinks as the square of d. So, halving d will quarter τ. Consequently, quartering ϵ requires us to descend 2 levels in a tree of OBBs, and a factor 4^{-k} change in ϵ requires an additional $2k$ levels of descent, leading to a factor 2^{2k} change in the number of BVs visited. Substituting $f = 4^{-k} = 2^{-2k}$, a factor f change in ϵ leads to a factor $2^{2k} = f^{-1}$ change in bounding volumes tested, reflected in a log-log plot as a line of slope -1. Our experimental measurements show that the slope is actually about -1.14, which we discuss later.

The implication is that for a given decrease in the gap size, we must descend twice as many additional levels of the sphere and AABB hierarchies as we would of the OBB hierarchy, and since each additional level doubles the number of BV tests, the factor two difference in additional levels translates into a square factor difference in additional work. More precisely, k additional levels in the OBB tree requires $O(2^k)$ additional work, whereas $2k$ additional levels in the sphere and AABB requires $O(2^{2k}) = O((2^k)^2)$ work. In the limit, the execution time of spheres and AABBs increases with decreasing ϵ as $T = O(e^{-2})$, and for OBBs it is $T = O(e^{-1})$.

We should note that the analysis we performed in the preceding paragraphs made use only of the convergence rates of the BVs, and not the specific shapes. Provided the models are sufficiently tessellated and barring extremely nonuniform distributions of BVs in the hierarchy, execution time for parallel close proximity situations is determined only by the convergence rate of the BV type used. If the BV has convergence

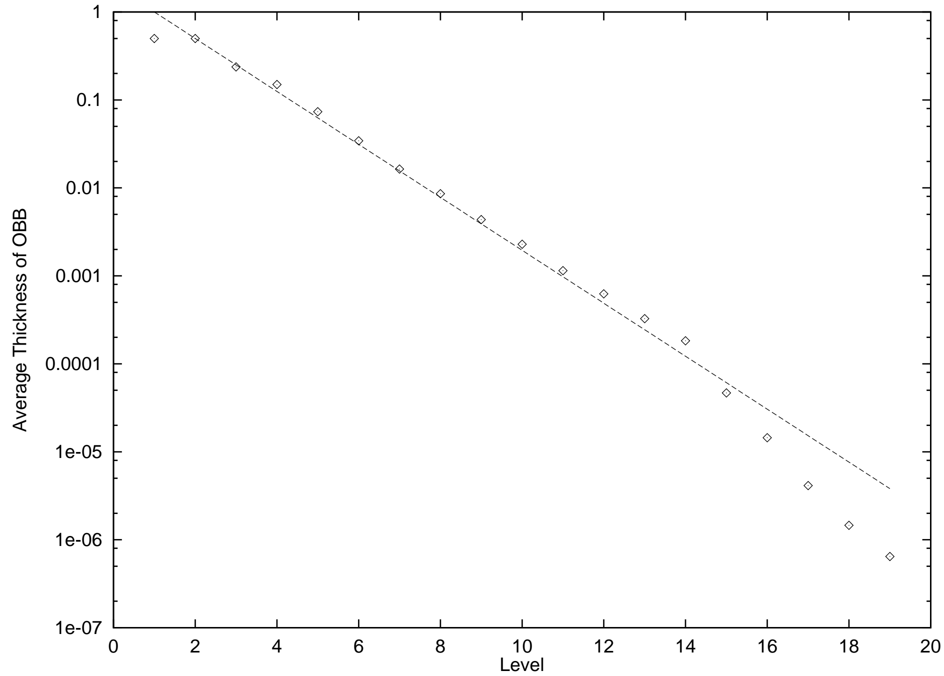


Figure 5.17: Average thickness of OBBs on a given level of the BVH (log plot).

rate r, then the execution time is

$$T = O(e^{-\frac{2}{r}})$$

The cause for the -1.14 slope for OBBs remains a mystery. We stated that that for OBBs, the thickness of the OBBs should diminish by a factor of four for every two levels descended in the BVH. Figure 5.17 shows the average thickness of the OBBs for each level. The line in the plot is provided as a reference – its slope is the rate of reduction we predicted. On average, BVs on levels 2 through 14 obey this rule closely. The BVs on the first few levels BVs cover geometry of such high curvature (high with respect to the size of the BV itself) that the rule is not followed. The last few levels are populated by increasingly large numbers of leaf nodes which cover only a single polygon, and thus have zero thickness, which skews the average. Our conclusion is that the diminishing thickness assumption is valid, and is not the cause of the -1.14 slope.

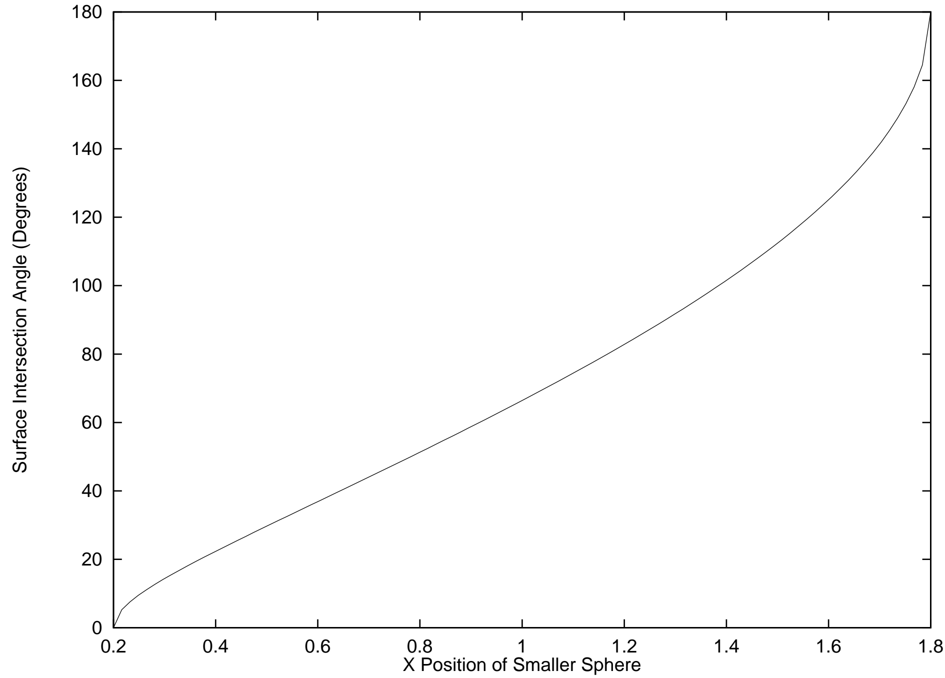


Figure 5.18: Angle at which surfaces meet as a function of position of moving sphere.

5.3.3 Transverse Contact

When the models intersect at a significant (non-grazing) angle, they are said to be in *transverse contact*. The term transverse refers to the angle at which the surfaces meet. A right angle is maximally transverse, while meeting at a tangent is minimally transverse, or perhaps said to be *not transverse*. For the moving sphere experiment, the ideal spheres meet at the same angle everywhere along the ideal intersection curve, and this angle is a function of x,

$$\theta = \cos^{-1}(r(x)) + \cos^{-1}(r(x)/0.8)$$

where r is itself a function of x as previously given,

$$r(x) = \sqrt{(0.8)^2 - (\frac{(1^2 - (0.8)^2 - x^2}{2x})^2}$$

A plot of this curve is shown in Figure 5.18. The surfaces are maximally transverse at about $x = 1.28$, which, as you may recall, is close to where the contact cost in Figure 5.8 is minimal.

The key features of transverse contact is that there are many contact pairs and

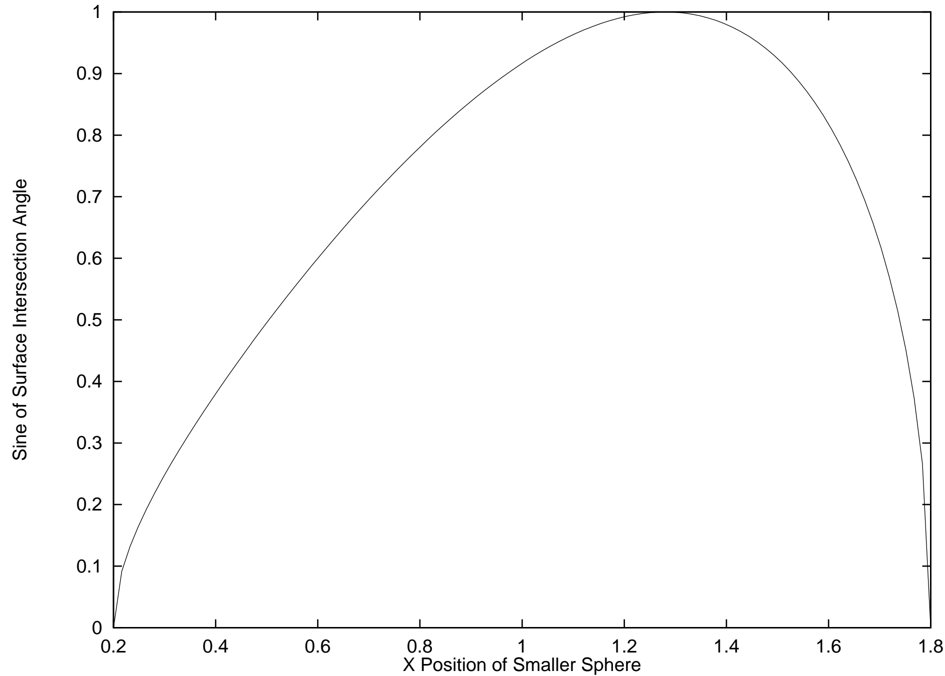


Figure 5.19: Sine of the surface intersection angle at each position of the moving sphere.

relatively few polygons which are close together. This is in direct contrast to parallel close proximity where there are zero contact pairs and many polygons which are close together. Of course, with transverse contact, polygons which are in the neighborhood of contact pairs are necessarily close, but these are few compared to the number of polygons in the entire model.

We will explore how increasing the tessellation increases the number of contact pairs and number of BV tests.

5.3.3.1 Transverse Contact: Dependence on Tessellation

Our experiment is with two maximally transverse spheres of unit radius, placed 1.41421 (approximately square root of two) units apart, such as those shown in Figure 5.20. With spheres of 80 polygons each, we performed 1000 collision queries, and recorded the average number of contacts and BV tests. Prior to each query, the spheres were given random orientations. This process was repeated for sphere models ranging from 80 polygons to almost 600k polygons, measuring the average query effort for each degree of tessellation.

The number of contacts varied as the square root of the number of polygons, shown

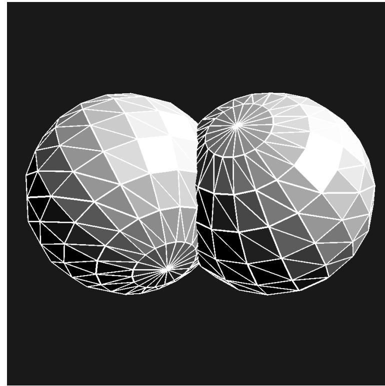


Figure 5.20: Transverse contact situation.

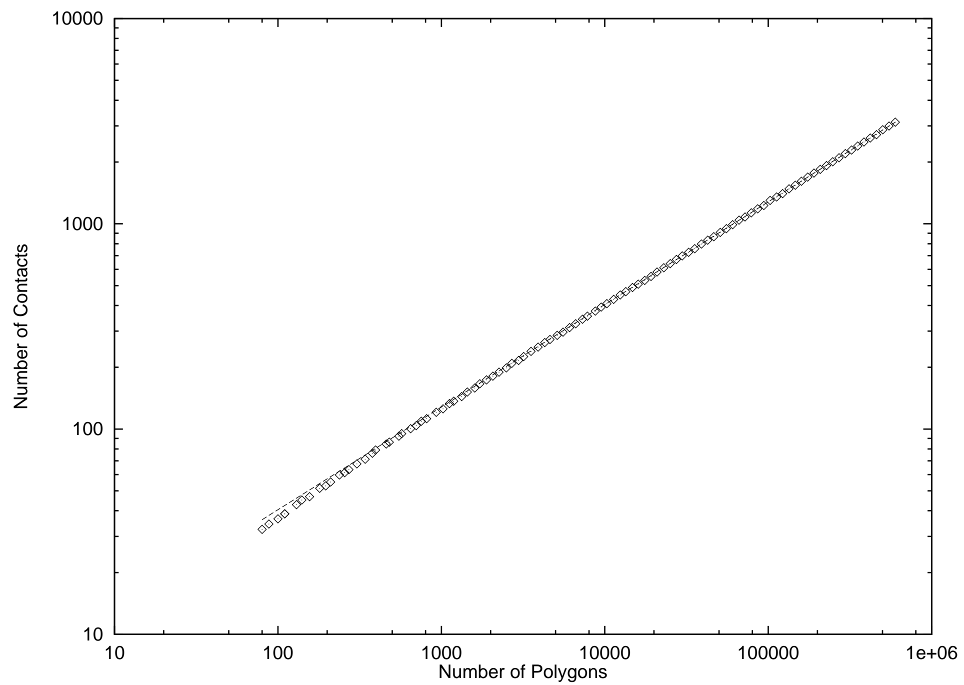


Figure 5.21: Dependence of contacts on tessellation of maximally transverse spheres (reference line has slope 1/2).

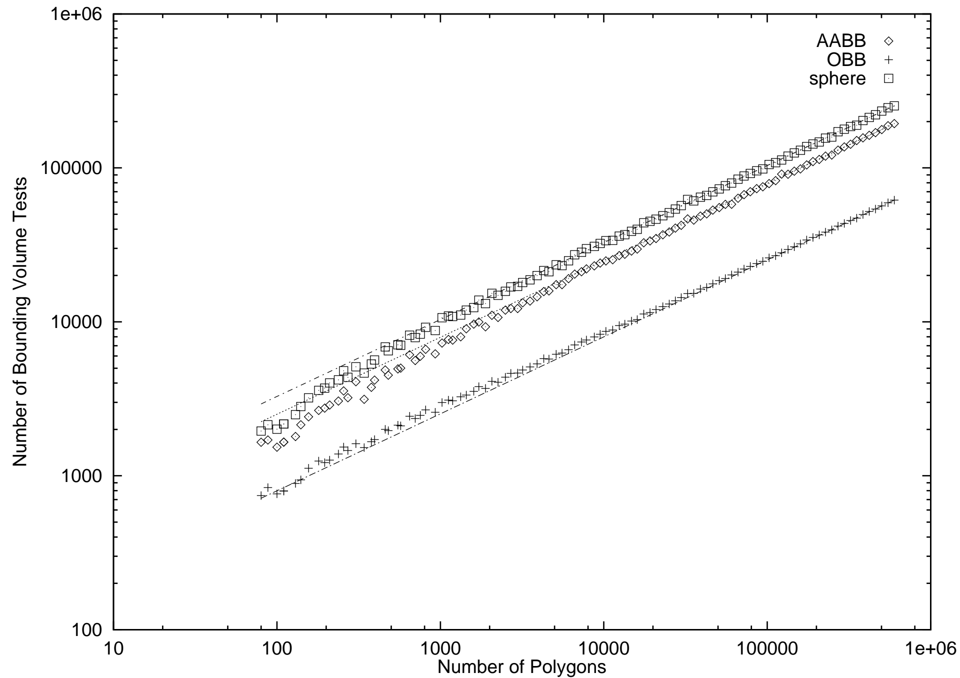


Figure 5.22: Dependence of BV tests on tessellation of maximally transverse spheres (reference lines have slope 1/2).

in Figure 5.21. This was expected, as it is well-known that a sufficiently smooth curve on a surface with n partitions will intersect $O(\sqrt{n})$ of them, provided the partitioning is sufficiently uniform. In this case, the surface is that of the models, the curve is the piecewise linear intersection of the models, and the partitions are the model facets.

The number of BV tests also appeared to grow as the square root of the number of polygons, as shown in Figure 5.22, with reference lines of slope 1/2. Although this has some important consequences, we should emphasize that it can be difficult to gauge asymptotic performance from empirical measurements, since one never knows whether the input sizes were large enough to allow the dominant terms to stand out.

As the number of polygons grow, the number of contacts and the number of BV tests grow. There is sufficient coherence among the contacts that the additional BV tests are divided among the additional contacts so as to leave the contact cost unchanged. The contact cost for each of the BV types is plotted in Figure 5.23.

Recall from Section 2.6.3 the bounds on the number of BV tests as a function of polygon count and contacts found. The lower limit was

$$v_l(n, k) = 2\log(n^2/k) + 2k - 1$$

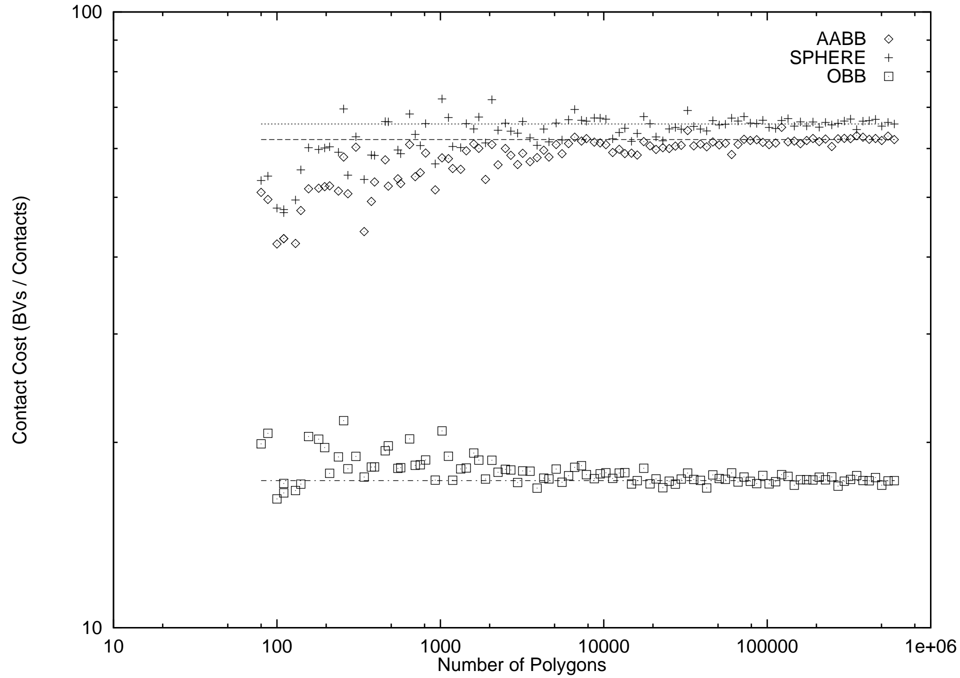


Figure 5.23: Dependence of contact cost on tessellation of maximally transverse spheres (reference lines are level).

and the upper limit was

$$v_u(n,k) = 2k\log(n^2/k) + 2k - 1$$

These bounds assumed perfectly balanced BVHs and perfect hypothetical bounding volumes (which were both optimistic assumptions). The difference between the bounds was based on favorable and unfavorable arrangements of the k contacts in the contact pair matrix, which contained n^2 cells.

In our experiment, we observed $k = O(\sqrt{n})$, as expected, and making this substitution into our bounds, we obtain

$$v_l(n,k) = O(\log(n^2/\sqrt{n})) + O(\sqrt{n}) = O(\sqrt{n})$$

and

$$v_u(n,k) = O(\sqrt{n}\log(n^2/\sqrt{n}) + O(\sqrt{n}) = O(\sqrt{n}\log n)$$

Notice that the lower bound on the number of BV tests is $O(\sqrt{n})$, which is what we observed in experiment.

We conclude that all three BV types in our benchmark performed within a con-

stant factor of optimal for this transverse contact situation, and no choice of BV or tree-building method can possibly do asymptotically better (since in Section 2.6.3 we proved that perfectly tight BVs under optimistic circumstances can do no better than this). Use of more sophisticated bounding volumes or more sophisticated tree building techniques will not improve upon this, except perhaps by constant factors. This is a strong statement, and our assumptions bear repeating. Our crucial assumption that $v = O(\sqrt{n})$ was based on experiment, and so does not carry the finality of a correct proof. Furthermore, the experiment employed uniformly tessellated smooth surfaces, so these results do not necessarily apply to more complex models with irregular tessellations. Also, this applies only to the dependence on tessellation, but not to dependence on angle or other quantities.

5.4 Summary of Chapter

In this chapter we presented and analyzed results of timing benchmarks. We showed that our new overlap test based on separating axes is 10 times faster than previously known methods. We also showed two results concerning asymptotic performance of BVH-based methods: OBBs outperform spheres and AABBs as gap size decreases in parallel close proximity situations, and all three BV types exhibit asymptotically optimal execution times with increasing tessellation density in transverse contact situations.

We should emphasize that the although BVH-based methods work for arbitrary polygonal models, these asymptotic results were based on experiments with highly tessellated smooth surfaces, and the analysis for parallel close proximity depended on the assumption that the tessellations of the surfaces were sufficiently dense and uniform.

Chapter 6

Previous Work

In this chapter, we survey the state of the art in collision detection between geometric models represented by a collection of polygons. Section 6.1 presents a classification of the collision detection problem domain. Section 6.2 surveys the algorithms available for polygonal models. Section 6.3 highlights techniques for reducing the number of pairwise intersection tests among multiple static or moving objects. Section 6.4 lists a number of public domain packages for collision detection.

6.1 Problem Domain Classification

A wide range of techniques, including hierarchical representation, algebraic formulations, spatial partitioning, analytical methods, and optimization methods, have been proposed. Algorithm design depends on the model representation, the desired query types, and the simulation environment.

6.1.1 Model Representations

There are many types of model representations used in CAD/CAM and 3D graphics. One possible taxonomy, which we adopt in this chapter, is shown in Figure 6.1. The major division among model representations is between the polygonal and the curved surface representations.

6.1.1.1 Polygonal models

Polygonal objects are the most commonly used models in computer graphics and modeling. Their simple representation and versatility, plus availability of hardware-

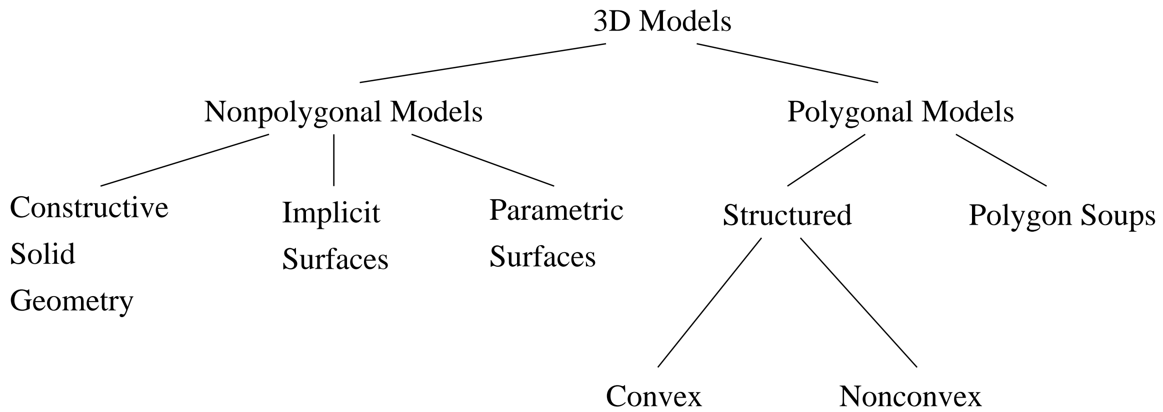


Figure 6.1: A Taxonomy of 3D Model Representations

accelerated rendering, make them an attractive choice in visualization and simulation applications.

The most general class of polygonal model is the *polygon soup*, which is a collection of polygons that are not geometrically connected and has no topology information available. Algorithms operating on polygon soups can make no assumptions about the placement of the polygons, such as them being connected in some particular way.

A *structured model* is a collection of polygons which forms a closed manifold, and for which connectivity information is available. Such a model has a well-defined inside and outside – it is a proper solid – and some geometric algorithms rely on this structure. If the object defined by the closed manifold is convex, then this additional structure can be exploited by some collision detection algorithms.

This taxonomy is not complete. For example, models composed of collections of triangle strips or meshes have structure, but do not form closed manifolds. However, most algorithms proposed apply to one of the two extremes of convex polytopes and polygons soups, and we will not examine the work that falls in between.

6.1.1.2 Constructive Solid Geometry

Constructive Solid Geometry or CSG forms objects from primitives such as blocks, spheres, cylinders, cones, and tori, by combining them with set theoretic operations such as union, intersection, and set difference [RR92, Hof89]. One strength of the CSG representation is that it enables an intuitive design process of building shapes by means of cutting (intersection and set difference) and joining (union) simple shapes to form more complex ones. The difficulty with CSG is that certain operations, such as rounding an edge or filleting a join, are difficult to describe with CSG operations.

Furthermore, an accurate boundary or surface representation, useful for rendering or interference computations, can be hard to compute from CSG representations [Hof89, KKM97].

6.1.1.3 Implicit Surfaces

Implicit surfaces are defined using implicit functions. They are defined with mappings from 3-space to the real numbers, $f : R^3 \mapsto R$, and the implicit surfaces are the loci of points where $f(x, y, z) = 0$. Such functions partition space into regions which are inside the model, $f(x, y, z) < 0$, and outside, $f(x, y, z) > 0$. While some implcit surface have infinite extent (such as the hyperboloid of one sheet, $f(x, y, z) = x^2 + y^2 - z + 1$), many implicit surfaces form closed manifolds (such as an ellipsoid $f(x, y, z) = x^2 + 2y^2 + 3z^2$), which is a desirable property for modeling objects.

If the function is polynomial in x, y, and z, then it is called *algebraic*, which includes the algebraic surfaces [Sed90], and higher-order functions [BW90]. Implicit surfaces are also often used as the primitives in CSG systems.

A special case of algebraic surfaces are the *quadrics*, which are the second-degree polynomials in x, y, and z. These can represent slabs, cones, spheres, and cylinders in a unified framework. They are widely used in a number of applications and a number of specialized algorithms have been developed for intersection computations between quadrics [FNO89, MG91, SJ91].

6.1.1.4 Parametric Surfaces

Parametric surfaces are images under mappings from some subset of the plane to space, $f : R^2 \mapsto R^3$. Unlike implicit surfaces, parametric surfaces are not generally closed manifolds. Hence, unlike CSG and implicit surfaces, they do not represent a complete solid model, but rather a description of surface boundary.

Parametric surfaces are easier to polygonalize and render as compared to the implicit surfaces, and a special class called the Non-Uniform Rational B-Spline (NURBS) has gained in popularity in CAD [LR80, Far93]. NURBS have some very nice properties which make them easier to operate on. They can also be represented using Bézier patches. It is worth noting that rational parametric surfaces (like NURBS and Bézier patches) are a proper subset of algebraic surfaces.

Contact/Intersection	$\text{cont}(P,Q) = P \cap Q$
Minimum Distance (Separation)	$d_{min}(P,Q) = \min_{p \in P} \min_{q \in Q} \lvert p - q \rvert$
Maximum Distance (Span)	$d_{max}(P,Q) = \max_{p \in P} \max_{q \in Q} \lvert p - q \rvert$
Hausdorff Distance	$d_{haus}(P,Q) = \max_{p \in P} \min_{q \in Q} \lvert p - q \rvert$
Penetration Distance	$d_{pen}(P,Q) = \min\ t \mid p + t \neq q,\ \forall (p,q) \in P \times Q$

Table 6.1: Five proximity measures.

6.1.2 Proximity Queries

Although this dissertation presents new work concerning collision detection between a pair of general polygonal models, it is useful to discuss other types of queries. Collision detection is one member of a family of proximity queries which yield information about the relative placements of objects. The five most familiar and easily defined proximity measures are collision, minimum distance (sometimes called separation distance), maximum distance (sometimes called span), Hausdorff distance, and penetration distance. The definitions of these measures are listed in Table 6.1.2.

The objects are defined as point sets P and Q which are closed and compact subsets of R^3. The proximity measure definitions are valid regardless of topology, smoothness, or other ordinary geometric properties. Thus, the definitions are very general and are applicable to more specific representations of objects, for example polygon soups, implicit surfaces, or NURBS.

6.1.2.1 Collision and Intersection

A collision query may report merely whether two objects in a given pose are touching, possibly reporting the points on each object which coincide (if they exist) as 'witnesses' to the result. Alternatively, a collision query may report a complete description of the set intersection of the objects, $P \cap Q$, yielding a null object if the given objects don't touch. Furthermore, given a description of the object's trajectories in space, the collision query might report the time of next contact (if any).

Collision queries are useful in dynamic simulations for checking whether moving objects have made contact in a given time step, in virtual prototyping applications for checking that mechanical components have not spatially interfered, and in path planning applications which seek to verify that steps along a particular path through an environment are collision-free. The collision query which yields a time of next contact can be used in physical simulations to control the size of the time step [Lin93, LM95].

6.1.2.2 Distance Measures

The remaining four proximity measures are concerned with various distances.

Some applications need to compute the minimum distance between two objects, defined as the shortest line joining the two point sets. This is useful for tolerance tests and clearance verification in virtual prototyping applications. Also, some path planning algorithms can benefit from distance computations.

The longest line anchored in the two point sets is the maximum distance. The maximum distance has not yet found extensive application. We include it here primarily because it fits neatly within our conceptual framework.

The Hausdorff distance of P from Q measures the geometric deviation of P from Q. It is the distance of the point on P which is furthest from all the points on Q. This measure is a very good candidate for driving polygonal model simplification algorithms. However, it has not seen much use, as it is currently expensive to compute.

Finally, we define the penetration distance as the minimum translation required to bring one point set out of contact with another. The query yields zero if P and Q are not touching, and some non-negative value if they are in contact.

For all four of these distance measures, there is an exact, approximate, and range form. The exact form of the query yields the answer to within greatest feasible precision. The approximate form yields an answer which is accurate to within some prespecified absolute or relative error tolerance. The range form does not yield the true distance, but only whether the true distance is less than or greater than a given distance. The approximate and range forms will generally be significantly less expensive than the exact form, for most inputs.

6.1.2.3 Remarks on Proximity Measures

Certain relations hold among the various proximity measures. For instance, it should be apparent from the proximity measure definitions that for any two point sets P and Q, we have

$$d_{min}(P, Q) \leq d_{haus}(P, Q) \leq d_{max}(P, Q)$$

Also, if two closed, compact point sets do not touch, there must be some positive separation between them,

$$\text{cont}(P, Q) = \emptyset \text{ iff } d_{min}(P, Q) > 0$$

It is not possible that both the separation (minimum distance) and the penetration distance can both be nonzero for a given pair of point sets. However, it is possible for both measures to be zero, such as when the point sets are grazing.

Appendix A gives a more detailed presentation of these different proximity measures for polygonal models, and discusses the relative computational complexity of their exact forms.

6.1.3 Simulation Environments

Special characteristics of a simulation may be considered when designing and choosing the most appropriate algorithm for collision detection. Here we examine a few common cases.

6.1.3.1 Pair Processing vs. N-Body Processing

If the problem involves only a pair of models, we call it "pair processing." If we have many different parts, we call it "n-body processing," in reference to the classic problem in celestial mechanics (many bodies moving under mutual gravitational influence).

6.1.3.2 Dynamic Scenarios and Temporal Coherence

Queries are often executed repeatedly on the same models in the same environment, as the objects rotate and translate (or possibly are subject to non-rigid transformations [HLMD96]) at successive time steps. In these dynamic environments, the geometric relationships may change only slightly in successive time steps, if the motion between steps is relatively small. Algorithms that can capitalize on this property are said to be exploit *temporal coherence*.

Temporal coherence has been exploited with convex polyhedra based on local features [Bar90, LC91, Lin93]. Local properties have been used in the earlier motion planning algorithms by [Don84, LPW79] when two objects come into contact. These algorithms exploit the spatial and temporal coherence between successive queries and work well in practice.

In order to exploit temporal coherence, some algorithms require bounds [Lin93, LM95] on the motion of the objects (e.g. objects' velocities or accelerations). Other algorithms, such as the ones based on interval arithmetic, need a closed-form expression of the motion as a function of time. Some algorithms demand no information on the motion, but need only the placements of the objects at successive time steps.

Reference	*Method or Concept*	*Precomp.*	*Temp. Coh.*	*Dist.*
[Meg83, Sei90]	Linear Programming	No	No	No
[GJK88]	GJK Minkowski Differences	No	No	Yes
[DK90]	Dobkin-Kirkpatrick Drums	Yes	No	No
[LC91, Lin93]	Lin-Canny Voronoi Regions	Yes	Yes	Yes
[Cam97]	Enhanced GJK	No	Yes	Yes

Table 6.2: Some of the basic approaches for computing intersection or contact status of convex polyhedra.

Sometimes the problem involves objects which are not in motion. For example, given a model of an entire power-plant, design engineers may be interested in performing static interference checks among components of the entire plant for tolerance verification and access clearance. In such cases, the preprocessing time cannot be amortized over multiple time steps, and therefore the preprocessing overhead takes on greater significance.

6.1.3.3 Rigid Bodies vs. Deformable Models

Models may deform over time. Assuming that the deformations between time steps are small, some algorithms may be able to exploit temporal coherence in this case as well.

6.2 Collision Detection for Polygonal Models

A wide range of methods have been used for collision detection among polygonal models. Convex polytopes have nice properties which permit efficient solution of the intersection problem. Some of these properties have also been used to exploit temporal coherence. For most non-convex polygonal models, hierarchical methods have been employed, and more recently we have seen more work on tighter-fitting bounding volumes.

6.2.1 Convex Polytopes

Most of the early work in collision detection has focused on algorithms for convex polytopes. A number of algorithms with good asymptotic performance have been proposed in the computational geometry literature. Using hierarchical representations, an $O(\log^2 n)$ algorithm is given in [DK90] for polytope-polytope overlap prob-

lem, where n is the number of vertices. This elegant approach has not been robustly implemented in 3D, however. Good theoretical and practical approaches based on the linear complexity of the linear programming problem are known [Meg83, Sei90]. Minkowski difference and convex optimization techniques are used in [GJK88] to compute the distance between convex polytopes by finding the closest points. Convexity plays an important role in all these algorithms.

Convex point sets have two important properties, which some of the algorithms exploit. First, for any two nonintersecting convex point sets in general position, there is a unique pair of closest points between them. Thus, some algorithms compute contact status by actually computing the separation distance. Second, as the points sets move, the pair of closest points move smoothly across the surfaces of the points sets. This property is very important for those algorithms which exploit temporal coherence.

The algorithms applicable to convex polytopes are especially relevant to this dissertation, since many of the bounding volumes used in collision detection algorithms are convex polytopes, such as axis-aligned bounding boxes, oriented bounding boxes, *discrete orientation polytopes* (or k-DOPs, which are polytopes whose face normals are taken from a given finite set of directions), and convex hulls. Consequently, it is important to be familiar with these techniques when designing an overlap test for a specific bounding volume type which is also a convex polytope. Some of these algorithms require some preprocessing to prepare search structures. A summary of the basic methods and their properties are given in Table 6.2.1.

6.2.1.1 Linear Programming

The problem of finding whether two convex polytopes intersect can be cast as a linear programming problem. Each polytope can be defined as an intersection of half-spaces. Our linear programming problem is to find a feasible solution to an arbitrary objective function subject to the linear constraints matching the half-spaces of the polytope. The classic LP algorithm indicates whether the intersection of all the half-spaces is empty, in which there is no feasible solution regardless of the objective function. The absence of a feasible solution indicates that there is no point contained in all the half-spaces, implying that the polytopes do not intersect. This approach is readily implemented, and requires no precomputation. If the half-spaces are not readily available, the problem can also be framed in terms of the coordinates of the vertices. In this case, if a feasible solution is found, it yields the coefficients of the plane which

separates the vertices. Thus, if a feasible solution is not found, it implies there is no separating plane, implying that the polyhedra are touching.

6.2.1.2 GJK Minkowski Differences

The Gilbert-Johnson-Keerthi (GJK) algorithm makes use of Minkowski differences of the polytopes being tested for intersection. It does not explicitly compute the Minkowski difference, which for two polytopes each with $O(n)$ vertices is a polytope of complexity $O(n^2)$. Rather, this algorithm computes a subset of its vertices on an as-needed basis.

Given two point sets $P, Q \in R^3$, the Minkowski difference $P \ominus Q$ is the point set

$$P \ominus Q = p - q | p \in P, q\ inQ$$

The point sets P and Q have nonempty intersection if and only if $P \ominus Q$ contains the origin. The GJK algorithm generates a series of simplices (polytopes of four or fewer vertices) of decreasing distance from the origin whose vertices are taken from $P \ominus Q$. The algorithm terminates when a simplex is found either to contain the origin (implying that $P \ominus Q$ contains the origin and that P and Q are in contact) or to be the same closest possible to the origin (implying that $P \ominus Q$ does not contain the origin, and that P does not touch Q).

The algorithm takes $O(n)$ time to generate each simplex, and perhaps as many as $O(n)$ simplices will be required, for $O(n^2)$ total running time, although in practice it is seen to complete much faster. The algorithm requires no precomputed search structures, and requires no structural or topological information. Its inputs are the vertices of the polytopes.

6.2.1.3 Dobkin-Kirkpatrick Drums

Dobkin and Kirkpatrick present a method for determining whether two polyhedra are touching. Each polyhedron is preprocessed into *drums* by slicing it into horizontal slabs, the cuts being made at the height of each vertex. The result for a polyhedron of n vertices is $n-1$ drums, each with a simple structure. They present an algorithm which, given a drum from each polytope, either yields an intersection point or yields a plane supporting one of the drums. The search for the polyhedron intersection begins with a test between the middle drums. Either an intersection is found immediately between the two given drums, or a plane separating the drums is found which supports

one of the drums. they prove that if the two drums do not touch, half of one of the polyhedra can be eliminated from consideration. Thus, at each step one of the inputs is reduced in size by half.

The algorithm requires a preprocessing step taking $O(n^2)$ time and space, after which repeated queries with different polyhedron poses may be executed in $O(\log^2 n)$ time each. It is possible to employ a more compact representation, requiring $O(n \log n)$ storage and precomputation, but increased access times increase the query time to $O(\log^3 n)$.

6.2.1.4 Lin-Canny Algorithm

The vertices, edges, and faces of a polytope are called its *features*. The voronoi region of a feature is that set of points in space which are closer to that feature than to any other feature. These voronoi regions can be described as intersections of half-spaces, and are easily computed given the description of a polytope.

Lin and Canny describe an algorithm by which they walk a series of candidate points across the surfaces of the polyhedra until the points arrive at the two closest features. The condition of being the closest features is easily verified using the voronoi regions: two features A and B are closest features if and only if the point on feature A closest to feature B is inside the voronoi region of feature B, and vice-versa. This verification takes $O(1)$ time for most polytopes, so the running time of the algorithm is on the order of the number of steps in the walk. The walk is directed by the verification itself: when a point is found not to be in the voronoi region tested, we walk to a neighboring feature whose voronoi region is closer to the tested point.

This algorithm exploits temporal coherence by starting with the previous query's closest points and features. For small inter-query movements, the new closest features will be found nearby, requiring very short walks. For sufficiently small motions between queries, the completion time is $O(1)$.

6.2.1.5 Enhanced GJK

Cameron later found a mathematical correspondence between the workings of the Lin-Canny algorithm and the GJK algorithm, and was able to devise a hybrid which borrowed the best attributes of each, yielding a fast algorithm which exploited temporal coherence, required no precomputation, required only the vertex coordinates for input, and was fairly robust.

Reference	*BV Hierarchy Type*
[Hub93]	spheres
	axis-aligned bounding boxes
[BCG+96]	oriented bounding boxes
[KHM+96]	discrete orientation polytopes
[KPLM98]	spherical shells

Table 6.3: Bounding volume hierarchy types for polygon soups.

6.2.2 Bounding Volume Hierarchies

A bounding volume hierarchy assigns the primitives composing a model (such as polygons) to the leaf nodes of a hierarchy of bounding volumes. In most cases, the leaf node bounding volumes cover the assigned polygons, although in some cases many leaf bounding volumes are used to collectively cover a polygon. A parent bounding volume must cover the polygons covered by its children, and consequently the root node covers the entire model. It is permissible that child bounding volumes extend beyond the scope of their parent, although for some hierarchies this never occurs, such as with axis-aligned bounding boxes and k-DOPs.

The most commonly used bounding volume types are spheres [Hub93, Qui94] and axis-aligned bounding boxes. These shapes are very simple and can be tested for overlap very quickly.

More recent work appears to focus on tighter-fitting bounding volumes, which usually require fewer bounding volume overlap tests to determine contact between models. However, these more complex shapes are also more expensive to test for overlap. Barequet et al. [BCG+96] have used arbitrarily oriented bounding boxes in hierarchical representations of surfaces for performing collision detection.

Klosowski et al. [KHM+96] have used *discrete orientation polytopes* (k-DOPs). k-DOPs are superior approximations to bounded geometry. Krishnas et. al. [KPLM98] have proposed a higher order bounding volume, designed to match curvature of the underlying 3D geometry, especially suited for Bézier patches and NURBS.

The process by which bounding volume hierarchies perform collision detection is described in detail in Chapter 2. The choice of bounding volume type is independent of the general method of traversing the hierarchies and affects only the low-level details of how a BV is fitted and how a BV overlap test is performed. As is demonstrated in Chapter 7, we can define the use of BVHs in terms of abstract BV operations.

Reference	*Partitioning*
(mitchell95)	Free Space Tetrahedralization
(Naylor90)	Binary Space Partition
(MooreWilhelms88)	Octree
(mitchell95)	k-d Tree
(Garcia-Alonzo)	3D Grid

Table 6.4: Spatial partitionings for polygon soups.

6.2.3 Spatial Partitionings for Polygon Soups

Spatial partitionings subdivide the object space into a collection of disjoint cells. These partitionings are usually hierarchical in nature. Spatial partitioning data structures are listed in Table 6.2.3 along with their associated references.

One of the disadvantages of spatial partitionings is that splitting polygons is usually unavoidable, leading to some increase in the depth of the tree data structure. Furthermore, the cells produced by the partitioning tend not to tightly cover the models, making them less suitable for determining contact status when the models are close together.

6.2.3.1 3D Grid

The simplest spatial partitioning is a volumetric 3D grid in the model space, as presented by Garcia-Alonso et al. [GASF94]. This nonhierarchical partitioning assigns each polygon to the voxels or bins which it touches during a preprocessing step. Each voxel then contains a list of all the polygons it contains or touches. When a query is performed between two models with given placements, we consider each pair of touching voxels. For each voxel pair, we test each polygon assigned to one against every polygon assigned to the other.

6.2.3.2 Binary Space Partition

Naylor et al. present a method by which binary space partitions can be used to perform the basic set theoretic operations on polyhedral objects [FKN80, Nay90]. For collision detection, we compute the set intersection of the two objects.

6.2.3.3 Octree, *k-d* Tree

Octrees and *k-d* Trees are used in a manner similar to that of bounding volume hierarchies, except for the details of tree construction and representation. During a query, both trees are traversed in tandem, and overlap tests between cells are used to eliminate potential contacts. An Octree is the familiar recursive 8-way splitting of cubical volumes. A *k-d* Tree is like a BSP Tree except that the cutting planes are restricted to alignment with the coordinate axes, yielding partitions which are rectanguloids. This flexibility leads to a potentially tighter fit to the covered geometry than Octrees afford.

6.2.3.4 Free Space Tetrahedralization

This interesting technique, presented by Held et al. [HKM95], preprocesses an environment composed of triangles by treating the free space as an object to be partitioned into tetrahedrons. The boundary of the free space is composed of the triangles making up the environment, so some of the sides of some of the tetrahedra will be environment triangles. The partitioning is designed to have a low stabbing number, meaning that a ray shot into the partitioning will strike a small number of partitions before encountering the boundary.

A query for whether a flying object has collided with the environment proceeds in two phases. First, we enumerate all the tetrahedra which the bounding box of the flying object touches. This is done by locating the tetrahedron containing a given corner of the bounding box, and then performing a breadth-first search. Second, each polygon of the flying object is tested for contact against all the environment triangles of the tetrahedra in the list.

The key to the method's efficiency is quickly finding the tetrahedron containing the specific corner of the bounding box. On the second and subsequent queries, we form a ray to the corner's present position from the previous position, and walk along the tetrahedral mesh from the previous starting tetrahedron to the new starting tetrahedron. The low stabbing number was desirable because this reduces the expected number of tetrahedra encountered during a walk of given length.

Although Held et al describe this algorithm in terms of a polyhedral object and environment, these methods are applicable to polygon soups.

6.3 N-Body Processing

For environments consisting of n (possibly moving) objects, performing $O(n^2)$ pairwise interference checks becomes a computational bottleneck when n is large. In order to eliminate some unnecessary pairwise checks and to speed up the runtime performance, several techniques have been proposed. Algorithms of complexity $O(nlog^2n+m)$ have been presented for spheres in [HSS83] and rectangular bounding boxes in [Ede83], where m corresponds to the actual number of overlaps. Some of the fastest practical algorithms for moving objects require the knowledge of maximum acceleration and velocity; others exploit the spatial arrangement to reduce the number of pairwise interference tests without assuming any knowledge of trajectories.

6.3.1 Scheduling Scheme

With bounds on velocities and accelerations we can estimate lower bounds on potential collision times. Scheduling algorithms [Lin93, LM95] maintain a queue of all object pairs that might collide, which is sorted by lower bounds on time to collision. These lower bounds on the time to collision are calculated adaptively and updated when a critical event, such as a collision, occurs. This technique has been successfully incorporated in the Impulse-Based Dynamics Simulator [MC95], reducing the frequency of the collision checks and thereby speeding up dynamics simulation.

6.3.2 Sorting-Based Sweep and Prune

More recently, Cohen et al. have presented algorithms and a system, I-COLLIDE, based on spatial and temporal coherence, for large environments composed of multiple moving objects [CLMP95]. The number of object pair interactions is reduced to only the pairs within close proximity by sorting axis-aligned bounding boxes (AABBs) surrounding the objects. It is output sensitive and its run time is linearly dependent on the number of objects in the environment instead of quadratic dependence. It uses dynamically sized AABBs, linear sweep and prune, and geometric coherence to quickly reject the object pairs that are unlikely to collide within the next time step.

6.3.3 Interval Tree for 2D Intersection Tests

We can use the interval tree [Ede83] for static query, as well as for the *rectangle intersection problem*, in which all intersections among a given set of 2D rectangles

are found. Finding the intersections of one rectangle with a given collection of n other rectangles takes takes $O(\log n + k)$ time where k is the number of reported intersections. Repeating this operation for each of the rectangles in a given set requires a total time of $O(n \log n + K)$ where K is the total number of intersecting rectangles.

6.3.4 Uniform Spatial Subdivision

We can divide the space into unit cells (or volumes) and place each object (or bounding box) in some cell(s) [Ove92]. To check for collisions, we have to examine the cell(s) occupied by each box to verify if the cell(s) is(are) shared by other objects. The memory required is $O(p^3)$ when using a $p \times p \times p$ grid. It is difficult to set a near-optimal size for each cell, and if the size of the cell is not properly chosen, the computation can be expensive or prohibitively memory intensive. For an environment where objects are of uniform size [Tur89], this is a rather ideal algorithm and especially suitable for parallelization.

6.4 Public Domain Software Packages

Most of public domain systems are applicable to polygonal models and some are also applicable to large environments composed of multiple moving objects. It is nearly impossible to compare different algorithms and systems fairly, since their performance varies, depending on the simulation environments (models, varieties of contacts, query types, motion description, etc.) and other factors. Here we only list them in the chronological order of their release and briefly describe their special characteristics.

6.4.1 I-COLLIDE Collision Detection System

http://www.cs.unc.edu/˜geom/I_COLLIDE.html.

I-COLLIDE is an interactive and exact collision-detection library for environments composed of *many* convex polyhedra or the union of convex pieces, based on the expected constant time, incremental distance computation algorithm [LC91, Lin93] and algorithms to check for collision between multiple moving objects [CLMP95].

6.4.2 RAPID Interference Detection System

http://www.cs.unc.edu/˜geom/OBB/OBBT.html.

RAPID is a robust and accurate polygon interference detection library for pairs of unstructured polygonal models. It is applicable to polygon soups – models which contain no adjacency information and obey no topological constraints. It is most suitable for close proximity configurations between highly tessellated smooth surfaces. [GLM96].

6.4.3 V-COLLIDE Collision Detection System

http://www.cs.unc.edu/~geom/V_COLLIDE.

V-COLLIDE is a collision detection library for large dynamic environments [HLC+97], and unites the n-body processing algorithm of I-COLLIDE and the pair processing algorithm of RAPID. It is designed to operate on large numbers of static or moving polygonal objects to allow dynamic addition or deletion of objects between timesteps. [Sys97]

6.4.4 Distance Computation between Convex Polytopes

http://www.comlab.ox.ac.uk/oucl/users/stephen.cameron/distances.html

This package is an enhanced and dynamic version of the distance routine of Gilbert, Johnson and Keerthi (GJK)[GJK88], which allows the tracking of the distance between a pair of convex polyhedra. It requires a list of all the edges in each convex polyhedra for best performance. Its performance is comparable to Lin-Canny convex polytope overlap test. [Cam97]

6.4.5 SOLID Interference Detection System

http://www.win.tue.nl/cs/tt/gino/solid/

SOLID is a library for interference detection of multiple three-dimensional polygonal objects (including polygon soups) undergoing rigid motion. Its performance and applicability are comparable to those of V-COLLIDE.

6.4.6 V-Clip Collision Detection System

http://www.merl.com/people/mirtich/vclip.html

The Voronoi Clip, or V-Clip, algorithm is a low-level collision detection algorithm for polyhedral objects – an improvement of the closest-feature tracking algorithm [LC91, Lin93]. It operates on a pair of convex polyhedra, or nonconvex hierarchies of them. In addition to distance computation, it can also report penetration points and estimated penetration distance between overlapping models. [Mir98].

Chapter 7

Implementation of Framework for Proximity Queries

In this chapter we describe the design and implementation of the software system we used to execute the performance benchmarks presented in Chapter 5. We describe this system for three reasons: First, the structure of this system is a superset of the RAPID system (*Robust and Accurate Polygon Interference Detection*) which is now in use in several commercial and research institutions. Second, this system was used to perform the benchmark comparisons between different bounding volume types. Third, other researchers in the field of collision and distance computation and related areas may be interested in the details of how our framework was implemented, and benefit from the discussion of the design choices.

In Section 7.1 we discuss the design goals of the system, the trade-offs present in the system design, the rationale for some of the implementation choices. The remainder of the chapter devotes one section to each major data structure or algorithmic element in the system. In Section 7.2 we discuss the data structure representing the bounding volume hierarchy. In Section 7.3, we discuss the structure of a proximity query algorithm, and show the similarities among collision, separation distance, and spanning distance queries, which can be exploited to simplify software design. In Section 7.4, we discuss abstractions for bounding volumes: a set of useful operations such as fitting to a collection of polygons, undergoing rigid transformation, or testing for overlap. The tree building and query algorithms manipulate the BVs through these abstract interfaces. In this section we also discuss the pitfalls of employing pure virtual functions in C++, which is one way to implement abstract data types. Finally, in Section 7.5 we discuss the interface for loading a model, and the implementation for building the BVH once the model is loaded.

7.1 Introduction to FPQ

Many of the ideas and algorithms described in this dissertation were implemented in a software system called *Framework for Proximity Queries*, abbreviated FPQ. This system framework permits a researcher to experiment with different choices of bounding volumes, traversal strategies, tree building algorithms, and other system design choices to find the best combination for his or her application.

There are tradeoffs among robustness, speed, memory use, and ease of use in our choices of algorithms and their implementations. FPQ allows the user to select from a palette of compatible algorithms using using compile-time switches. These switches control conditional compilation of the source code using the familiar `#if` C++ compiler directive, effectively specializing the code to the needs of the user. The resulting compiled code is compact and efficient.

The basic system requires the user to make choices regarding bounding volume type, whether to use nested coordinate systems in the hierarchies, choice of coordinate systems for updates, and traversal rules. These are the fundamental variations on the use of bounding volume hierarchies for proximity queries.

Bounding Volume Type The user chooses at compile-time whether to use spheres, AABBs, or OBBs. This choice affects memory usage, test tree pruning, and BV test speed.

Hierarchy Coordinate Systems These can be either flat or nested. Flat is almost always preferred, except in some special cases.

Update Coordinate System The user must specify whether to use the world space, a model space, or a bounding volume space in which to perform BV volume tests.

Traversal Rules The user must specify what rule to use for deciding which bounding volume's children to use for the next test. Choices implemented are "Descend largest diameter", "descend largest surface area", "descend largest volume", "descend model A", and "descend model B". This choice affects how coordinate transformations should be composed.

Several features are designed to make the library easier to debug and to use,

Built-in Textual and Graphical Diagnostics Routines are provided which execute OpenGL procedures causing a bounding volume, an entire tree, or the

result of a query to be drawn on the screen. Also, all routines update various statistics which can be examined, such as how many bounding boxes were tested during a query, or what is the memory usage of a model's BVH. A compile-time switch eliminates the code for maintaining the statistics, as well as their allocated storage.

User-Defined Namespace This enables the user to compile and link to several versions of the FPQ library, each configured to use a different set of compile-time options.

Fast Storage and Retrieval of Hierarchies The data structures which encode the bounding volume hierarchies are organized into a few large blocks of memory, which can be stored or retrieved with just a few system calls.

Iterative Tree Traversal Rather than using recursive procedure calls to perform a depth first search, we use an iterative procedure which employs a stack data structure. This grants us more control over the memory usage, and allows us to interrupt and resume the traversal, which is useful for single-stepping through the traversal, displaying results at each step.

Some of the features allow users to trade among speed, memory usage, accuracy, and ease of use.

Copy vertex coordinates vs. using pointers The user has the option of having FPQ create a local copy of the coordinates of the models' vertices, or merely keeping pointers to the user's copy of this data. Users often must keep a copy of this data for rendering or other tasks.

Floats vs Doubles The user may specify whether to use double precision or single precision floating point numbers for bounding volume parameters and arithmetic operations. Doubles are less prone to arithmetic error, while floats use less space and may execute faster.

Lazy Construction of Hierarchies The user may opt to have the trees built on an as-needed basis. Children of a node are constructed just prior to being visited. For a short sequence of queries, this can yield a significant time savings, because only those portions of the trees which get visited actually get built.

Update Caching The user may opt to have updated bounding volumes cached for later use. This is a compile-time option, which permits the cached parameter variables to be omitted from the bounding volume structure if the user chooses not to employ update caching.

Branching Factor Most of the hierarchies we consider are binary trees. In some cases, one might want three-way trees, or perhaps trees in which nodes have a variable number of children. A compile-time switch allows the user to "hard-wire" the code to assume 2-way, 3-way, or variable way trees. In the last case, a loop is employed in certain segments of code. In the first two cases, the loop can be unrolled, and the code is straight-line. By default, FPQ uses strictly binary trees.

Truncated Trees Sometimes we want every leaf node of a hierarchy to enclose just one triangle. In some cases, to save memory at the expense of speed, we may wish to enclose multiple triangles in each leaf node.

Some of the options interact in complex ways, but it is possible to abstract the problem of processing a query so that the options concern distinct portions of the source code. We begin with an explanation of the basic data structures employed by FPQ, and then explain in a top-down fashion how a query is executed. Along the way, we note how certain design choices enable the features listed above.

7.2 Bounding Volume Hierarchy Data Structure

At the heart of FPQ is the data structure which stores a bounding volume hierarchy for a model. A bounding volume hierarchy is stored in three (or four) memory blocks: the model block, the node array, the face array, and (optionally) the vertex array. This structure is shown in Figure 7.1 – the vertex array is grayed to show that it is optional.

The basic C++ `struct` types defining a model, a BV node, a face, and a vertex are shown in Figure 7.2. The model block is a simple C++ class which contains model-specific information and pointers to the aforementioned arrays. The model class "owns" the arrays – it is responsible for creating and deleting them. The vertex array, if it exists, is a simple array of vertices, of which each vertex is an array of three FPQ_REALs (An FPQ_REAL is `typedef` defined to be either a float or a double, depending upon the user's configuration of the library). The face array is an array of

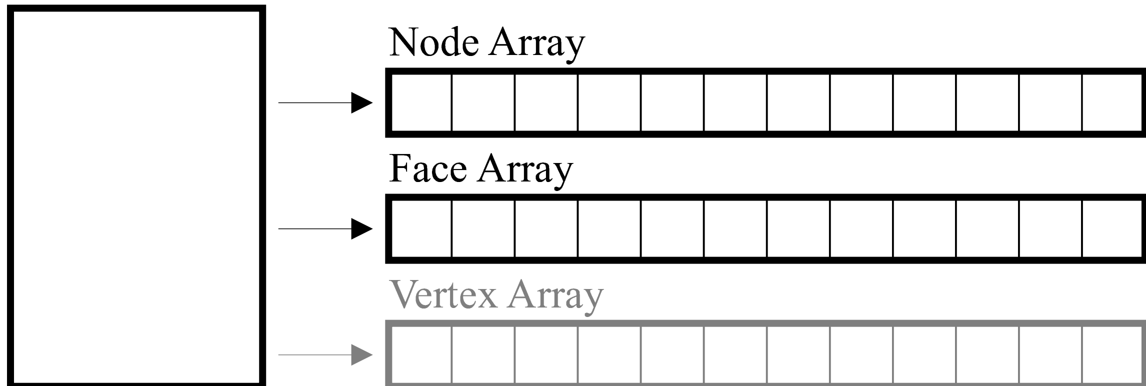


Figure 7.1: BVH memory diagram – the vertex array is optional.

faces, of which each face is an array of three pointers. Each of these pointers points to the start of an array of three FPQ_REALs – which might be the coordinates in a vertex structure in the vertex array, or might be a vertex managed by the user, depending on how the user configures FPQ. Either way, during tree construction and query processing, the vertices are treated as read-only data. The node array is a tree of nodes embedded in an array. Each node contains parameters specifying the bounding volume placement (in the FPQ_BVRep structure, described in Section 7.4), references to the children nodes, and references to the enclosed faces.

The term *reference* means something that encodes the location of a thing, such as a pointer, an array index, or something more complex. We do not mean the word in the specific technical sense of a C++ reference.

As a specific example of tree embedding, consider the simple model consisting of 5 line segments, which we used in Section 1.1. This is shown with its bounding spheres in Figure 7.3. The hierarchy is not strictly binary, and each leaf node contains exactly one face. There are 8 bounding volumes, whose organization is shown in Figure 7.4. For this structure, the node array and face array are organized as shown in Figure 7.5. The elements of the face array contain information about each face (each face is actually a line segment in Figure 7.3). Each node in the node array contains information about which nodes are its children, and which faces it covers.

Trees of any branching factor can be embedded in an array, and it is easy to ensure that sibling nodes occupy contiguous elements of the node array. This means that we need only two numbers to refer to the list of children of a node, even if there are dozens of children: one number specifies the index of the first child, and the other

```
struct FPQ_Model
{
  eArray<FPQ_Node> nodes;
  eArray<FPQ_Face> faces;
  eArray<FPQ_Vertex> vertices;
};
```
(a)

```
struct FPQ_Node
{
  FPQ_BVReps bv;
  int num_children;
  FPQ_Node *first_child;
  int num_faces;
  FPQ_Face *first_face;
};
```
(b)

```
struct FPQ_Face
{
  FPQ_REAL *v1, *v2, *v3;
};
```
(c)

```
struct FPQ_Vertex
{
  FPQ_REAL coords[3];
};
```
(d)

Figure 7.2: The basic structures, implemented in C++.

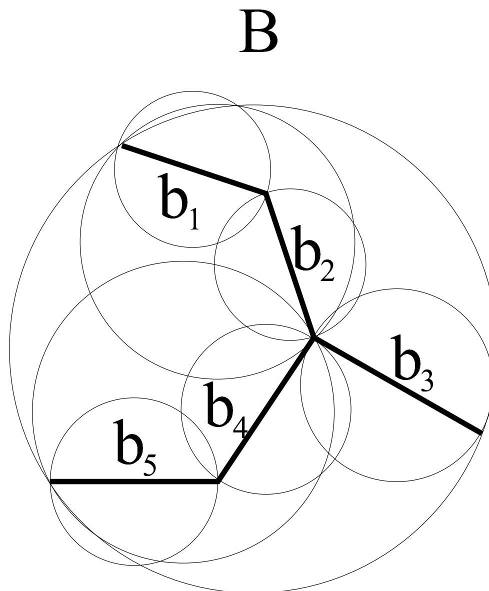


Figure 7.3: A model and its various bounding spheres.

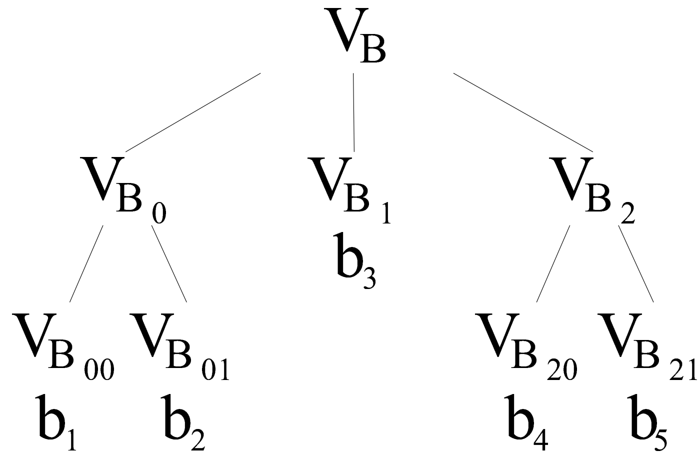


Figure 7.4: The model's bounding sphere hierarchy.

	0	1	2	3	4	5	6	7
1st child	1	4	-	6	-	-	-	-
num children	3	2	0	2	0	0	0	0
1st face	0	0	1	3	0	1	3	4
num faces	5	2	1	2	1	1	1	1
	V_B	$V_{B\,0}$	$V_{B\,1}$	$V_{B\,2}$	$V_{B\,00}$	$V_{B\,01}$	$V_{B\,20}$	$V_{B\,21}$

Node Array

0	1	2	3	4
b_1	b_2	b_3	b_4	b_5

Face Array

Figure 7.5: The model's node and face arrays.

number is a count of the children. For leaf nodes, the count is zero, and the "first child" number is not used (hence, we have "–"s in Figure 7.5). Furthermore, all the faces under any given node are stored in contiguous elements of the face array. As with the child node references, this means that we need only two numbers to specify all the faces owned by a node (whether leaf or internal): one number is the index of the first face, and the other number is a count of the faces owned. Simply put, these pairs of numbers refer to sub-arrays within the node and face arrays. We should note here that we generally do not need to access the triangles owned by an internal node.

On most modern machines, integers are four bytes, and by using 32-bit integers to index the array, we can access four billion nodes. With a binary BVH, four billion nodes can cover a model composed of two billion faces.

7.2.1 Writing Bounding Volume Hierarchies to Disk

Since the model hierarchy is organized into four large memory blocks, transferring the data to and from disk is accomplished with only four writes or reads and minimal formatting. With a dynamically allocated tree in which each node or face occupies a random segment of memory, we might write the tree to disk piecemeal, possibly with one write per node with additional information to encode the tree structure. This is our primary reason for embedding the tree in arrays.

Writing our BVH to disk takes place in four steps. First, we write the model block. Next, we write the vertex array, if it exists. Next we pass through the face array, transforming each vertex pointer into a byte offset from the start of the vertex array, after which we write the face block to disk. Finally, we pass through the node array, transforming face pointers into byte offsets from the first face element, after which we write the node array to disk. To read it from disk, we first read the model block, which tells us the length of the remaining blocks. We then read the vertex array, then the face array, and the node array. We then make a pass through the face and node arrays, transforming the byte offsets into real pointers again.

If minimizing I/O calls is paramount, the four blocks can be marshalled into a single larger block of memory, possibly with padding bytes to ensure appropriate memory alignment of the array elements (on word or paragraph boundaries, for example). After transforming the pointers to byte offsets, the entire data structure can be written to disk in a single I/O call. Similarly, it can be retrieved with a single I/O call. There is no need to "unpack" the four blocks, since they are acceptable as a nearly contiguous group, as long as the arrays are not going to be extended.

```
bool Leaf(FPQ_Model *M, FPQ_Node *N);

int NumFaces(FPQ_Model *M, FPQ_Node *N);

int NumChildren(FPQ_Model *M, FPQ_Node *N);

FPQ_Face *Face(FPQ_Model *M, FPQ_Node *N, int k);

FPQ_Node *Child(FPQ_Model *M, FPQ_Node *N, int k);
```

Figure 7.6: Access routines for node operations.

There is one detail concerning the vertex block we should address. If the FPQ is configured to point to user-managed vertices rather than it's own copy, how do we read a hierarchy from disk and have the faces refer to the correct vertex data? If we can assume that the user vertices are stored in a single array, we can transform the vertex pointers in the face array into byte offsets, as before, and then store just the model, node, and face blocks. Later, when this is loaded, the user tells us the start of his vertex array, and we turn the offsets back into real pointers. This assumes that during tree retrieval the vertex data is ordered the same way it was when we stored the tree.

7.2.2 Optimizing Memory for Embedding

FPQ embeds a tree of nodes in an array using four integers per node structure. Some optimizations are possible, which are discussed in the following subsections, and explained in Appendix A. These optimizations use different encodings for specifying what faces are owned by a leaf node, or what children belong to a parent node. We hide these details behind access routines, listed in Figure 7.6. Each of the procedures take a pointer to the node concerned, and a pointer to the model that node belongs to. The `Leaf` procedure returns true if the node is a leaf node, and false otherwise. The `NumChildren` and `NumFaces` procedures return the number of child nodes and the number of faces owned by the given node, respectively. The `Face` procedure returns a pointer to the kth face owned by the node. Similarly, the `Child` procedure returns a pointer to the kth child of the node.

With a straightforward organization of a node structure, such as shown in Figure 7.2, these access routines are very simple. Some organizations are more cryptic, encoding the information in a more compact form which must then be extracted. We redefine the access routines accordingly, and all the tree traversal algorithms are

expressed in terms of them. We next discuss two such encodings for the embedding parameters, which each use 32 bits per node.

7.2.2.1 Binary Trees with Single-Primitive Leaves

Suppose we are using only a binary tree in which every leaf node contains only one model face. If we know this at compile-time, we need not store `num_children` and `num_faces` in the node structure; they are implicit. Internal nodes have two children, and no faces. Leaf nodes have no children, and one face. For internal nodes, we still need to know who the two children are, and for leaf nodes, we still need to know what triangle the node owns. We can use a single signed integer, k, to encode this information for both types.

Observe that *either* a node is internal and has children *or* it is a leaf node and owns a face, but never both. We will let $k < 0$ indicate that the node is internal, and $k >= 0$ indicate that the node is a leaf. If the node is internal, then the first child is at position $-k$ in the node array, and the second child is at position $-k + 1$. If the node is a leaf, then the face that it owns is at position k in the face array. If k is a 32-bit integer, then this encoding can refer to 2^{31} nodes, of which half are leaf nodes, adequate for a model of up to 1 billion triangles. This was assuming we had a strict binary tree and only one triangle per leaf node. Note, this method applies equally well to strict n-way trees, in which every node has exactly n children.

7.2.2.2 Variable Branching and Multiple-Primitive Leaves

The above optimization assumed strict n-way trees and exactly 1 face per leaf node. How far can we relax these assumptions and still keep the node/face references down to 32 bits? Let us begin by setting a large (but somewhat arbitrary) upper limit on the number of triangles we expect to see in a model: 16 million triangles. This is 2^{24} triangles, whose binary tree would require $2^{25} - 1$ nodes. A tree which has internal nodes with more than 2 children would require fewer of them. So, we need to reserve at most 25 bits for the node reference in internal nodes, and 24 bits for the face reference in leaf nodes. We also need to reserve 1 bit for the leaf status flag, so we have 6 bits remaining for internal, and 7 bits remaining for leaf nodes. These remaining bits can be used for `num_children` and `num_faces`, respectively. This allows an internal node to have from 1 to 64 children, and a leaf node can own from 1 to 128 faces. These fanouts and face counts are quite adequate for most applications.

We should emphasize this trade-off, however: allocating more bits for large fanouts and face counts leaves fewer bits for referencing the first child or the first face, which limits the polygon count of the model we can process.

7.3 Structure of the Query Procedure

A basic query is a depth-first traversal through the bounding volume test tree as described in Chapter 3. Rather than using recursive procedure calls, we use a stack and a loop to implement our own recursion. This is equally efficient, and permits us to interrupt and resume the query process after any given step. The recursive and iterative forms of the collision query for strict binary trees are shown in Figure 7.7. The procedure texts are spaced so as to juxtapose corresponding elements.

In the recursive form, we first test if two bounding volumes, A and B, have collided. If they have not, then we terminate that recursion branch. If they have, then we check to see if both BVs are leaves of their respective hierarchies. If they are, then we test the faces they own using `CollTris()`, which adds contact pairs to the list in the result structure R. If they are not both leaves, then we can descend to the children of one of the BVs to perform the next generation of tests. The function `DescendA()` takes both bounding volumes, and decides whether the first (returning `true`) or the second (returning `false`) should be descended. Each branch leads to two more invocations of the recursive procedure, after which the current invocation completes.

In the iterative form, the same elements are present: the collision test between bounding volumes, the tests for leaf status, the collision checks among triangles owned by the nodes, and the decision function `DescendA()`. The variable `Q` is a standard stack (a last-in first-out queue) of BV pairs, initialized to contain only the top-level bounding volumes, passed in as A and B. We then iterate while the stack contains pairs, executing essentially the same series of operations as in the recursive form, with three main differences. First, the next BV pair to test is obtained from the stack with the `Pop` operation. Second, recursion branches are terminated with the `continue` statement, which immediately jumps to the top of the loop. And third, the pending recursion branches are scheduled by pushing pairs onto the stack. Note that the left and right recursion branches are given in reverse order from those in the recursive version, to account for the fact that the most recently pushed pair is the next one to be tested. These two forms execute precisely the same sequence of

```
Procedure
RColl(R,A,B)
{








  if (!CollBV(A,B))
    return;

  if (Leaf(A) AND Leaf(B))
  {
    CollTris(R,A,B);
    return;
  }

  if (DescendA(A,B))
  {
    RColl(Left(A),B);
    RColl(Right(A),B);
  }
  else
  {
    RColl(A,Left(B));
    RColl(A,Right(B));
  }

}
```

```
Procedure
IColl(R,A,B)
{
  Q = NULL;
  Push(Q,A,B);

  while(!Empty(Q))
  {
    Pop(Q,A,B);

    if (!CollBV(A,B))
      continue;

    if (Leaf(A) AND Leaf(B))
    {
      CollTris(R,A,B);
      continue;
    }

    if (DescendA(A,B))
    {
      Push(Q,Right(A),B);
      Push(Q,Left(A),B);
    }
    else
    {
      Push(Q,A,Right(B));
      Push(Q,A,Left(B)); }
    }
  }
}
```

Figure 7.7: Recursive and iterative forms of collision query routine.

Collision

```
Procedure
IColl(R,A,B)
{
  R.list = <empty>;
  Q = <empty>;
  Push(Q,A,B);

  while(!Empty(Q))
  {
    Pop(Q,A,B);
    if (!CollBV(A,B))
      continue;

    if (Leaf(A) AND Leaf(B))
    {
      CollTris(R.list,A,B);
      continue;
    }

    if (DescendA(A,B))
    {
      Push(Q,Right(A),B);
      Push(Q,Left(A),B);
    }
    else
    {
      Push(Q,A,Right(B));
      Push(Q,A,Left(B));
    }
  }

  return R;
}
```

Distance

```
Procedure
IDist(R,A,B)
{
  R.d = <infinity>;
  Q = <empty>;
  Push(Q,A,B);

  while(!Empty(Q))
  {
    Pop(Q,A,B);
    if (DistBV(A,B) > R.d)
      continue;

    if (Leaf(A) AND Leaf(B))
    {
      DistTris(R.d,A,B));
      continue;
    }

    if (DescendA(A,B))
    {
      Push(Q,Right(A),B);
      Push(Q,Left(A),B);
    }
    else
    {
      Push(Q,A,Right(B));
      Push(Q,A,Left(B));
    }
  }

  return R;
}
```

Span

```
Procedure
ISpan(R,A,B)
{
  d = 0;
  Q = <empty>;
  Push(Q,A,B);

  while(!Empty(Q))
  {
    Pop(Q,A,B);
    if (SpanBV(A,B) < R.d)
      continue;

    if (Leaf(A) AND Leaf(B))
    {
      SpanTris(R.d,A,B));
      continue;
    }

    if (DescendA(A,B))
    {
      Push(Q,Right(A),B);
      Push(Q,Left(A),B);
    }
    else
    {
      Push(Q,A,Right(B));
      Push(Q,A,Left(B));
    }
  }

  return R;
}
```

Figure 7.8: Collision, distance, and span queries have very similar structure.

bounding volume tests and triangle tests.

The iterative form is very useful for debugging. The body of the loop performs a single step of the query, and after each step, the next BV pair on the stack represents the pair to be tested next, which can be drawn graphically. A user can direct the application to single step through the query process, showing the next and all pending BV pairs, and other information. This can be used to inspect the progress of the query by visual verification, or to gain insight into its behavior and performance.

7.3.1 Unification with Distance and Span Queries

Distance and span queries are very similar in structure to the collision query. These are shown side-by-side using the iterative form in Figure 7.8. Collision queries add to a list of contacts in a result structure. Simple distance and span queries revise an estimate, which is held in the result structure. When all tests have been exhausted, the estimate is known to be exactly correct.

For implementation purposes, it is sometimes hazardous to maintain multiple

Any Query

```
Procedure
IAnyQuery(R,A,B)
{
  InitQuery(R,Q,A,B);

  while(!Empty(Q))
  {
    Pop(Q,A,B);

    if (TestBV(R,A,B))
      continue;

    if (Leaf(A) AND Leaf(B))
    {
      TestTris(R,A,B));
      continue;
    }

    if (DescendA(A,B))
    {
      Push(Q,Right(A),B);
      Push(Q,Left(A),B);
    }
    else
    {
      Push(Q,A,Right(B));
      Push(Q,A,Left(B));
    }
  }

  return d;
}
```

Figure 7.9: Common procedure for all queries.

nearly identical procedures. Modifications in algorithms or data structures would often require parallel revisions, and in the event that one revision was missed, it might not be obvious during debugging that the bad procedure is out of sync with the others. For this reason we create a single procedure which incorporates the common structure, hiding the differences behind functions, as shown in Figure 7.9. Depending on whether FPQ is configured to perform collision, distance, or span queries, the functions `InitQuery()`, `TestBV()`, and `TestTris()` perform the appropriate operations. In this way we have separated the details of what query is being performed from the details of how the test tree is being traversed.

7.3.2 Descent Rules

The descent policy, as described in Chapter 2, is embodied in the function `DescendA`. Depending on how FPQ is configured, this function may compare the diameters of the

```
bool
DescendA(A,B)
{
  return ( Leaf(B) OR (!Leaf(A) AND (DiameterBV(A) >= DiameterBV(B)));
};
```

Figure 7.10: A version of `DescendA` which descends the largest diameter BV.

Binary

```
if (DescendA(A,B))
{
  Push(Q,Right(A),B);
  Push(Q,Left(A),B);
}
else
{
  Push(Q,A,Right(B));
  Push(Q,A,Left(B));
}
```

Variable Children

```
if (DescendA(A,B))
{
  for (k=NumChildren(A)-1; k>=0; k--)
    Push(Q,Child(A,k),B);
}
else
{
  for (k=NumChildren(A)-1; k>=0; k--)
    Push(Q,A,Child(B,k));
}
```

Figure 7.11: We use a loop for variable branching factors, but unroll them for binary trees.

argument BVs, or it may compare their surface areas, or their volumes, or some other quantity. If one wishes to experiment with other criteria, these may be incorporated into the `DescendA()` function. It must always consider the leaf status of A and B, however, as `DescendA()` must never suggest descending to the children of a leaf node. The software, if it is configured to run most efficiently, will not include safety checks for accessing the invalid child pointers of a leaf node.

7.3.3 Fixed Branching Factor

The code presented thus far assumes a binary tree, but these data structures and algorithms also apply to trees whose nodes have a variable number of children. Using a tree with variable degree nodes requires us to use a loop in the descent section of the code, as shown in Figure 7.11. The software is designed to use conditional compilation according to compile-time switches, so that the appropriate code is used. Of course, we could use the `for` loops even on binary trees, but it is more efficient to unroll them. The functions `Left(A)` and `Right(A)` are synonymous with `Child(A,0)` and `Child(A,1)`, respectively.

Model A Space

```
TestBV(R,A,B)
{
  UpdateBV(Ba,B.nt,R.Rab,R.Tab);
  CollBV(R,A.nt,Ba);
}
```

(a)

World Space

```
TestBV(R,A,B)
{
  UpdateBV(Aw,A.nt,R.Rwa,R.Twa);
  UpdateBV(Bw,B.nt,R.Rwb,R.Twb);
  CollBV(R,Aw,Bw);
}
```

(b)

Figure 7.12: `TestBV` performs updates prior to bounding volume testing.

7.3.4 Updating Bounding Volumes

During construction of a BVH, the BVs are specified with respect to the model's local coordinate system, which is considered the BV's native coordinate system. During a collision query, the models' locations are specified with respect to a world coordinate system. As we traverse the two trees, pairs of BVs must be brought into a common coordinate system in order to perform the bounding volume overlap, distance, or span computation.

Updates in FPQ are performed in the `TestBV()` routine, seen used in Figure 7.9. This routine accepts a pair of BVs and calls the overlap, distance, or span procedure after bringing them into the same coordinate system.

If FPQ is configured to use "World Space Updates" then both BVs are updated to world space prior to testing. If FPQ is configured to use "Model Space Updates" then one of the BVs is updated to the other BVs model space prior to testing. These alternatives are shown in Figure 7.12, where FPQ has been configured to perform collision checking. In part (a), the bounding volume B from the second model is transformed into the models space of A. This transformation is determined by the rotation $\mathbf{R}_{ab}$ and translation $\mathbf{T}_{ab}$, which are retrieved from the result structure `R`. In part (b), both bounding volumes are transformed from their respective model coordinate systems into the world space. The rotation $\mathbf{R}_{wa}$ and translation $\mathbf{T}_{wa}$ transform to world from the A model space, while $\mathbf{R}_{wb}$ and $\mathbf{T}_{wb}$ transform to world from the B model space. These transformations among model coordinate systems are computed once and placed in the result structure `R` during query initialization.

World Space with Caching

```
TestBV(R,A,B)
{
  if (!IsUpdated(R,A))
  {
    UpdateBV(A.up,A.nt,R.Rwa,R.Twa);
    SetUpdated(R,A);
  }
  if (!IsUpdated(R,B))
  {
    UpdateBV(B.up,B.nt,R.Rwb,R.Twb);
    SetUpdated(R,B);
  }
  CollBV(R,A.up,B.up);
}
```

```
bool IsUpdated(R,A)
{
  return (R.id == A.id);
}
```

```
SetUpdated(R,A)
{
  A.id = R.id;
}
```

Figure 7.13: `TestBV()` with update caching enabled.

7.3.5 Implementing Update Caching

FPQ implements update caching by the simplest possible means. The node structure holds two `FPQ_BVReps` structures for its bounding volume, one called `nt` for its native model space placement, and the other called `up` for its cached updated placement. The BV proximity query always uses the `up` members of the nodes, and the `UpdateBV()` is skipped for nodes whose updated member is current. The code for `TestBV()` with caching enabled is shown in Figure 7.13. This is for world space updates; for model space updates the code is similar, but designed to update only one model.

Every query is given a unique id number, starting with zero and incremented whenever any model moves – the user is responsible for informing FPQ when this happens. This id is stored in the result structure R, and every node is also given storage it. The test `IsUpdated()` compares the query id with the id stored in the node; if they match, then the node is deemed updated and the test returns true, skipping the update. If the numbers do not match, then we perform the update, and then call `SetUpdated()`, which merely assigns the id in R to the id in the node.

The procedures `IsUpdated()` and `SetUpdated()` are very simple, and one might think that TestBV would be simpler and faster if it handled the id fields directly. We felt that if we wished to experiment with other update status storage methods, these functions would nicely facilitate implementation modifications. Furthermore, by using the `inline` function specifier we pay no performance penalty for a procedure call. We should note that if FPQ is configured *not* to perform update caching, then the `up` and `id` fields are ommitted from all data structures, and the updates in `TestBVs()` are performed unconditionally.

7.4 Bounding Volume Abstractions

Different bounding volumes require different representation parameters, as well as different operations for transforming them, testing for overlap, computing distance and span, fitting to a collection of triangles, and computing metrics such as volume, surface area, and diameter. All these details are hidden behind procedures, and all the algorithms for tree building and query processing are expressed in terms of those procedures. This section exposes some of the BV differences, and shows how those differences are hidden from the higher level algorithms.

7.4.1 The Case Against Virtual Functions

The bounding volume types are different in their specifics, such as their formulas for computing volumes, but very similar in their operations – they all need to be fitted to sets of triangles, tested for overlap, and so forth. This is textbook application for polymorphism, implemented in C++ via derived classes and virtual functions, and many veteran C++ programmers would immediately choose this approach for hiding the details of the various bounding volume types. Our experience is that in this case, the use of virtual functions can impose a significant performance penalty, as we explain in this section.

Our earlier attempts at a system which could support multiple bounding volume types employed a C++ abstract class representing a generic bounding volume. It included C++ pure virtual functions for all the BV procedures, such as fitting to a collection of triangles or computing the distance between two such BVs. Specific bounding volume types such as spheres, axis-aligned bounding boxes and oriented bounding boxes were derived from the abstract class. The high level procedures which performed queries or constructed BVH's handled generic BV objects, using their pure virtual functions to manipulate them, and C++ determined at run-time the actual BV type of the object and called the appropriate procedures.

The abstraction was effective and permitted very clean and compact design, but there were two ways in which this design hindered performance. First, because we accessed the bounding volumes through the abstract base class, the bounding volume functions could not be inlined at the calling points – for example, when calling the volume function for a BV, the compiler could not know whether the sphere volume code or the OBB volume code should be used. Second, every object of a class with virtual functions implicitly has some additional storage. In our case, this was 32 bytes. This additional storage becomes significant when millions of bounding volumes are being allocated. For example, this 32 bytes would double the storage required by an otherwise simple sphere BV type.

For this reason we have opted to employ no virtual functions in FPQ. The bounding volume type is selected at compile-time using `#define` directives, and the bounding volume operations are compiled accordingly. The bounding volume representation is known at compile-time, so most functions can be successfully inlined, and no storage is added to the bounding volume structure for virtual function support.

Sphere | AABB | OBB

```
struct FPQ_BVreps
{
  FPQ_REAL c[3];
  FPQ_REAL r;
};
```

```
struct FPQ_BVreps
{
  FPQ_REAL c[3];
  FPQ_REAL r[3];
};
```

```
struct FPQ_BVreps
{
  FPQ_REAL R[3][3];
  FPQ_REAL c[3];
  FPQ_REAL r[3];
};
```

Figure 7.14: The FPQ_BVReps structure is defined according to the bounding volume type.

```
FPQ_REAL DiameterBV(FPQ_BVReps *A);

FPQ_REAL VolumeBV(FPQ_BVReps *A);

FPQ_REAL SurfaceAreaBV(FPQ_BVReps *A);
```

Figure 7.15: Metrics for bounding volumes.

7.4.2 Bounding Volume Representation

FPQ can be configured for one of three bounding volume types: spheres, AABBs, and OBBs. The structure holding the representation parameters is called `FPQ_BVReps`, and is defined according to which bounding volume FPQ will use. The different definitions are shown in Figure 7.14. The spheres consist of a center point and a radius. The axis-aligned bounding boxes have a center point and a "radius" along each axis. The oriented bounding boxes have a center point, a 3×3 matrix defining its orientation, and then a "radius" for each local axis.

7.4.3 Bounding Volume Access Routines

Figure 7.15 lists three BV metrics defined for all volume types. The diameter of a bounding volume (or any point set) is the length of the longest line whose endpoints lie in the volume. Volume and surface area are the conventional measures of a volume. All of these are very simple arithmetic functions for all the bounding volumes listed.

Figure 7.16 lists the three proximity queries defined for any BV: collision, distance, and span. Given two bounding volumes, they return the appropriate quantity. These

```
bool      CollBV(FPQ_BVReps *A, FPQ_BVReps *B);

FPQ_REAL DistBV(FPQ_BVReps *A, FPQ_BVReps *B);

FPQ_REAL SpanBV(FPQ_BVReps *A, FPQ_BVReps *B);
```

Figure 7.16: Proximity query functions for bounding volumes.

```
void UpdateBV(FPQ_BVReps *Z, FPQ_BVReps *A, FPQ_REAL R[3][3], FPQ_REAL T[3]);
```

Figure 7.17: Procedure for updating a bounding volume. The bounding volume A gets transformed by rotation $\mathbf{R}$ and translation $\mathbf{T}$, with the result being assigned to Z.

functions are almost trivial for spheres, whereas their implementations for OBBs are considerably more intricate.

We must also have a means of transforming a BV representation from one coordinate system to another. This is called the `UpdateBV()` function, and its prototype is shown in Figure 7.17. This function transforms the BV specified by A with a rotation R and translation T, putting the result into Z. For spheres and OBBs, this process is exact. For AABBs, the transformed A may not be aligned with its new coordinate axes, and thus have no exact representation. In this case, we cover the transformed A with the smallest possible AABB, which is then stored in Z. This is the "wrapping" process for aligned BV types, described in chapter 3.

Finally, shown in Figure 7.18, there is a procedure which takes a list of faces and computes a BV which covers them, putting the result into A. This BV should be as small as possible, within reasonable effort. This procedure is used in the top-down tree building process. The options and variations for volume fitting are set by compile-time switches.

```
void FitToFacesBV(FPQ_BVReps *A, FPQ_Face *F, int n);
```

Figure 7.18: Procedure for fitting a BV to a collection of faces. F is a list of n faces. The resulting BV is placed in A.

```
void FPQ_Model::BeginModel();

void FPQ_Model::AddFace(FPQ_REAL *p1, FPQ_REAL *p2, FPQ_REAL *p3);

void FPQ_Model::EndModel();
```

Figure 7.19: This is the interface for creating a model. `BeginModel()` is called first, followed by many calls of `AddFace()`, and ending with `EndModel()`. Each call of `AddFace()` adds a face to a growing list. The hierarchy is built when `EndModel()` is called.

7.5 Building the Bounding Volume Hierarchy

The bounding volume hierarchy is constructed in two phases. There is first the loading phase, described in Section 7.5.1, and then the tree building phase, described in Section 7.5.2.

7.5.1 Loading the Model

The interface for loading the model data is similar to the OpenGL graphics API: the user executes a `BeginModel()` procedure call, followed by multiple `AddFace()` calls, and concluded with a `EndModel()` call. Their prototypes are shown in Figure 7.19. These procedures are member functions of the `FPQ_Model` class, and they apply to the model being used to call them. In each call to `AddFace()` the user should pass three pointers – each pointer points to an array of three `double`s or `float`s (depending on how FPQ is configured), representing the coordinates of the vertices of a triangular face.

If FPQ is configured to make its own copy of the vertex data, then the `AddFace()` procedure copies the three vertices into its own vertex array, and then adds a new face in the face array whose pointers point to the new vertices. If FPQ is configured to use the user-managed vertices, the vertex array is left empty, and the new face uses the addresses of the user's vertices.

7.5.2 Building the Tree

The bounding volume hierarchy is constructed when the `EndModel()` function is called, using a top-down "fit-and-split" method. Beginning with the list of faces in the face array, the BV function `FitToFacesBV()` is used to obtain a fitted BV for them. This array is then partitioned in place, yielding two new sub-arrays, on which

the "split-and-fit" routine is recursively invoked on each of these lists. A recursion branch is terminated when its face list is smaller than a specified length, in which case the BV fitted to that list is left as a leaf node owning that list.

There are many ways to approach the partitioning of a list of faces. The basic steps used by FPQ are

1. Choose a splitting axis, which may be arbitrarily directed.
2. Choose a point on that axis at which to place a splitting plane.
3. Partition the faces according to which side of the plane their centroids lie.

Within this framework, there are many variations. FPQ chooses a splitting axis which is aligned with the direction of maximum spread of the faces. The splitting plane is placed where the centroid of the set of faces projects onto the axis. In short, we use a splitting plane which cuts the "center of mass" of the triangles and which is normal to the direction of their greatest spread. Currently, FPQ creates only binary trees, although the queries are designed to use variable way trees, as well.

7.6 Summary of Chapter

In this chapter we described the implementation of a software framework for BVH-based collision and distance computation. The rationale for constructing a configurable software framework was to enable fair comparison between variations on the basic approach – such as using different BV types.

The performance of a system depends not only on the fundamental algorithms being employed, but also by the skill and time investment of the implementor. We believed that two systems which had small algorithmic differences (such as using different BV types) but which had great implementation differences (such as being written on different dates or by different people) would not yield a fair comparison. By developing a configurable software framework which encompassed a large family of algorithms using the same implementation base, we could dismiss implementation bias as a possible source of performance variation.

Chapter 8

Future Work

The following sections describe interesting research directions. All these subjects involve bounding volume hierarchies, although not necessarily trees of OBBs.

8.0.1 Improved Proximity Characterization

How to characterize proximity is a very deep issue in collision detection. As stated in Section 1.5, the cost of a collision query is not only due to the number of polygons and the number of contacts, but also due to the nature and degree of the proximity of the models.

In this work it was observed that for parallel close proximity situations, the polygon count was irrelevant, and that only the separation ϵ of the surfaces and the convergence rate r of the BV type used dictated the asymptotic running time of the collision test,

$$T = O(\epsilon^{-2/r})$$

This is a very special case of proximity. Identification and characterization of other basic types of proximity may yield a canonical set of proximity types, and it may be that the scenarios encountered in practice can be viewed as some combination of these fundamental components. With a classification system in hand and a deeper understanding of what factors contribute to the cost of a proximity query, we might be better able to predict collision query performance and to design data structures and algorithms more suitable for specific applications.

8.0.2 Analysis of BVH-Based Distance Queries

A distance query has very similar structure to a collision query. It may be possible to undertake an analysis of distance queries similar to our analysis of collision queries. Preliminary studies show some counter-intuitive performance characteristics of BVH-based distance computation, such as an increased execution time as objects get further apart. This is the opposite qualititative behavior we observe for collision detection. It would be useful to characterize the performance of BVH-based distance computation more carefully.

8.0.3 Penetration Distance Queries for Polygon Soups

Penetration distance between two overlapping point sets can be defined as the minimum required translation of one of them to make them disjoint.

$$\text{pen}(A, B) = \text{magnitude of shortest } \mathbf{v} \text{ such that} \min_{a \in A} \min_{b \in B} |\mathbf{a} - \mathbf{b} + \mathbf{v}| > 0.$$

By this definition, the notion of penetration distance is well defined for any type of model, including arbitrary polygonal models.

There are currently no known efficient methods for computing the penetration distance of two arbitrary polygonal models. An $O(n^6)$ method is known for pairs of polyhedra of n faces, but this is too expensive to be practical. It would be useful to compute penetration distance with good expected execution time, even if worst-case execution time was poor. Many applications which need penetration distance, such as kinematics simulations and haptic display systems, allow models to interpenetrate only by small amounts before adjusting their configuration – this property of the inputs might be exploitable.

In such applications, Mirtich and Canny [MC95] efficiently approximate penetration distance between polytopes by using closest features from a previous timestep when the models were disjoint. But they do not guarantee the accuracy of the estimate, and their approach only applies to polytopes and unions of polytopes. Another method is required by applications which have no temporal coherence or which need exact penetration distance.

8.0.4 Hausdorff Distance Queries for Polygon Soups

The Hausdorff distance is an asymmetric measure which yields the maximum distance of any of the points in one object from all of the points in another object,

$$\text{haus}(A, B) = \max_{a \in A} \min_{b \in B} |\mathbf{a} - \mathbf{b}|.$$

The Hausdorff distance of A from B measures the maximum distance that any portion of A strays from B. This measure is well defined for any closed and compact point set.

This quantity appears to be more difficult to compute than separation distance, but easier to compute than penetration distance (see Appendix A). It might be useful in applications which need a measure of geometric similarity, such as polygonal model simplification algorithms. It is also a necessary component for the bottom-up tree construction algorithm described in Section 8.0.5.

8.0.5 Hausdorff-Driven Greedy Bottom-Up OBBTree Construction

The Hausdorff distance of an OBB from its underlying geometry may be an appropriate figure-of-merit for the fit of an OBB. In applications where models frequently graze or nearly touch, but do not significantly interpenetrate, we might get significantly better performance if we use BVHs in which the Hausdorff figure-of-merit is minimal.

Bottom-up tree construction methods initially fit BVs to primitives, and then iteratively combine groups (fitting new BVs to newly created groups) until all the primitives have been joined together. Such methods may be superior to the top-down approaches we use in this dissertation. Greedy bottom-up construction methods can be driven by arbitrary objective functions, which are used to select the next merge pair. By using the Hausdorff metric as a guide, we may be able to construct BVHs for complex models which yield better performance for grazing and near-contact situations.

8.0.6 Lazy Evaluation and Curved Surfaces

Bounding volume hierarchies can be applied to any partitioning of any closed and compact point set. Throughout this dissertation, we discussed polygonal models,

sometimes referring to the polygons as primitives. Consider instead, a recursive partitioning of a curved surface into patches, where each patch plays the role of a primitive. BVH-based methods reduce the global problem to a series of local patch-patch intersection problems.

However, curved surfaces are not composed of a finite number of patches (unlike polygonal models), and there may be no obvious choice of how deep to build the BVH for a curved surface. The solution may be to create the bounding volume hierarchy on-the-fly during a query. This reopens the box fitting problem, which we addressed for polygons, but now applied to partitions of a curved surface.

Some of the issues to address here are: How best to partition the curve during a query, how best to fit a BV to a patch, how to decide whether to compute a patch intersection or to further subdivide, and how to compute the intersection between two patches.

8.0.7 Run-Time Adaptive Hierarchies

A run-time adaptive hierarchy is a bounding volume hierarchy which reorganizes itself periodically according to the history of queries to improve the expected execution time of future queries. Applications which exhibit significant temporal coherence or which repeatedly access a limited portion of a large model may benefit from this approach.

The idea is that during a slow-moving simulation of, for example, a piston in an engine block, the BVH of the engine block reorganizes itself to place the primitives lining the combustion chamber closer to the top, while the primitives in more remote portions of the engine block fall to deeper levels.

In such applications, there is likely to be great disparity between the frequencies with which the BVs are visited. Following the principle of Huffman encoding, it is desirable to place the frequently visited BVs and primitives closer to the root node in the hierarchy – we unbalance the tree in favor of the nodes and primitives which are most likely to be involved in future queries.

8.0.8 Deformable Models

Examples of deformable models range from articulated objects composed of a few rigid components, like a robot arm, to fully-deformable objects such as meshes used in cloth dynamics simulations. In all cases a single precomputed BVH may not suffice to enclose the models as they change shape.

Some global deformations such as nonuniform scaling or affine deformations can be accommodated by new BVH representations which can lazily deform the BVs with the models during query time. Some local changes, such as a rigid rotation of a component (such as on the robot arm), can be accommodated by adjusting a local transformation in the BVH, provided the BVH has a structure resembling the model's own hierarchical structure. However, arbitrary local deformations of the model (such as updating a vertex in the cloth dynamics mesh) may require a non-local adjustment of the BVH, and possibly affecting many nodes in the hierarchy. For such deformations, we need structures which enable us to adjust bounding volume hierarchies quickly as models change shape arbitrarily.

Appendix A

Proximity Queries

In this appendix we present a unified view of types of proximity queries: separation distance, spanning distance, Hausdorff distance, and penetration distance.

Section A.1 discusses five types of proximity queries, listing possible taxonomical categories.

Section A.2 expresses the five query measures in terms of the Minkowski difference.

Section A.3 expresses some of these queries in terms of functions of the partitions of their models, and shows the logic behind how these expressions can be simplified in light of information gained by BV tests. The expressions in terms of partitions mirror the BVH-based tandem tree traversal. We state that the penetration and Hausdorff distances cannot be expressed in terms of partitions of both models, and offer this as an explanation of why BVH-based techniques have not been applied to these measures as successfully as they have been applied to collision, separation, and spanning distances.

A.1 Types of Queries

A proximity query is any computation that produces information about the relative placement of models. Collision detection and distance computation are two examples.

There are two forms of collision query: boolean and enumerative. The boolean distance query computes whether the two sets have a point in common. The enumerative form yields some representation of the intersection set.

There are several varieties of distance measure. These are measures which yield a single real number which describes a relative distance between two sets A and B.

Separation Distance: This is the length of the shortest line joining the sets:

$$\operatorname{dist}(A, B) = \min_{a \in A} \min_{b \in B} |a - b|.$$

Hausdorff Distance: This measures the maximum deviation of one set from the other:

$$\operatorname{haus}(A, B) = \max_{a \in A} \min_{b \in B} |a - b|.$$

Note that this is an asymmetric measure.

Spanning Distance: This is the greatest distance between points of the two sets:

$$\operatorname{span}(A, B) = \max_{a \in A} \max_{b \in B} |a - b|.$$

Penetration Distance: This is the minimum distance needed to translate one set to make it disjoint from the other:

$$\operatorname{pen}(A, B) = \text{magnitude of shortest } \mathbf{v} \text{ such that} \min_{a \in A} \min_{b \in B} |\mathbf{a} - \mathbf{b} + \mathbf{v}| > 0.$$

There are at least three forms of each of these distance measures: exact, approximate, and boolean. The exact form asks for the exact measure. The approximate form yields an answer which is within some given error tolerance of the true measure – the tolerance could be specified as a relative or absolute error. The boolean form reports whether the exact measure is greater or less than a given value. The boolean and approximate forms generally complete with much less work than the exact form. The exact form is more powerful than the other two forms in the sense that the result of an exact query can be used to trivially answer an approximate or boolean query. Furthermore, the norm by which distance is defined, $|x|$, can be varied. The Euclidean norm is most common, but in principle other norms are possible, such as the l_2 (Manhattan distance) and l_∞ (largest coordinate) norms.

We can add the element of time to a 2-body query. If the trajectories of two bodies are known, then we can ask when is the next time that a particular boolean query (whether collision, separation distance, Hausdorff, span, or penetration) will become true or false. In fact, this "time-to-next-event" query can have exact, approximate, and boolean forms (the boolean form would answer whether the next event would be before or after a specified time). These queries are called *dynamic queries*, whereas the ones that do not use motion information are called *static queries*. Dynamic systems

often perform static queries at specific time steps in a simulation, so static queries can be applied to dynamic problems.

These measures, as defined above, apply only to pairs of sets. However, some applications work with many objects, and need to find the collisions among all pairs of them, or determine which pairs satisfy a particular distance relationship (are within a given distance, or penetrate a particular amount), or determine the next event among all of them. Thus all the query types listed above have associated n-body variants. These generally use techniques that eliminate object pairs from consideration to avoid executing a 2-body query on all pairs.

This description of the problem domain is based solely on the information computed by the query, and not based on the methods employed. Most algorithms described in the literature can be classified using this system. These categories are mostly independent, so most combinations are meaningful – except that only collision has an enumerative form, and it does not have exact or approximate forms.

report: boolean, exact, approximate, enumerative.

measure: separation, span, Hausdorff, penetration, collision.

multiplicity: 2-body, n-body.

temporality: static, dynamic.

representation: polygons, convex polytopes, implicit, parametric, NURBS, quadrics, unions of these, set theoretic combinations of these.

dimension: 2,3,d.

Clearly there is a great variety of queries with different uses for different applications, and there is a great variety of algorithms for implementing them.

Most of the dissertation addresses static enumerative collision queries between pairs of polygonal models in 3-space. However, there is insight to be gained by examining the similarities and differences among some of the different query types.

A.2 Queries and Minkowski Differences

There is a strong connection between Minkowski differences and most of the distance measures mentioned above.

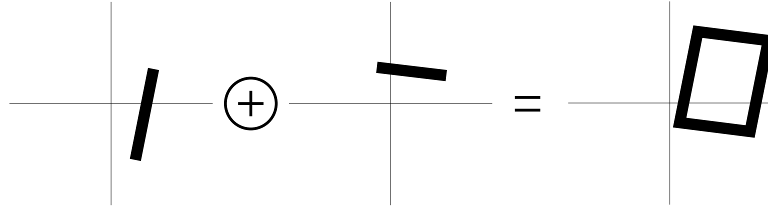


Figure A.1: The Minkowski sum of two line segments is a parallelogram: the shape swept out by using one as a "brush" and the other as the brush's path.

Given two point sets, A and B, the Minkowski difference is the set of points P such that

$$P = A \ominus B = \{\ \mathbf{a} - \mathbf{b} \mid \mathbf{a} \in A, \mathbf{b} \in B\}$$

The Minkowski difference can be understood in terms of the Minkowski sum and the reflection operation. The reflection of the set B is $-B$, and is defined as

$$-B = \{\ -\mathbf{b} \mid \mathbf{b} \in B\}$$

and the Minkowski sum is

$$A \oplus B = \{\ \mathbf{a} + \mathbf{b} \mid \mathbf{a} \in A, \mathbf{b} \in B\}$$

Given these definitions, we have

$$P = A \ominus B = A \oplus -B$$

The Minkowski sum can be viewed graphically as a "painting" operation. One shape is treated as the brush, and the other is used as the brush's path, as is shown in Figure A.1. The brush is placed at all positions such that its origin falls on a point in the path set. Seen this way, the brush set sweeps out a portion of space according to the path set. If the path is a simple line segment, the operation resembles an extrusion. To represent the Minkowski difference graphically, it is convenient to think of it as a Minkowski sum with the reflection, as shown in Figure A.2.

If two points sets are each composed of many discrete parts, then the Minkowski sum and difference are composed of the sum and differences of the parts taken pairwise. The difference of unions is the union of the differences, or more precisely, if A

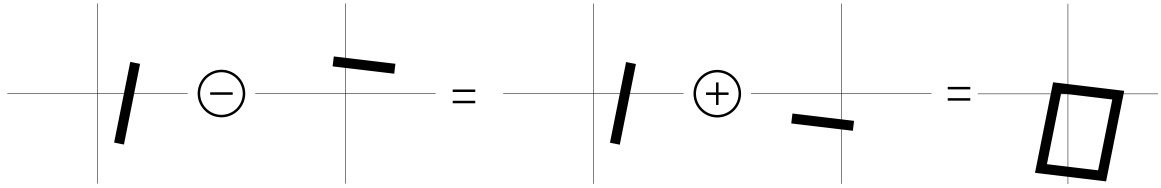


Figure A.2: The Minkowski difference of two shapes is the Minkowski sum of the first shape with the inverse of the second shape, where inverse means reflecting through the origin.

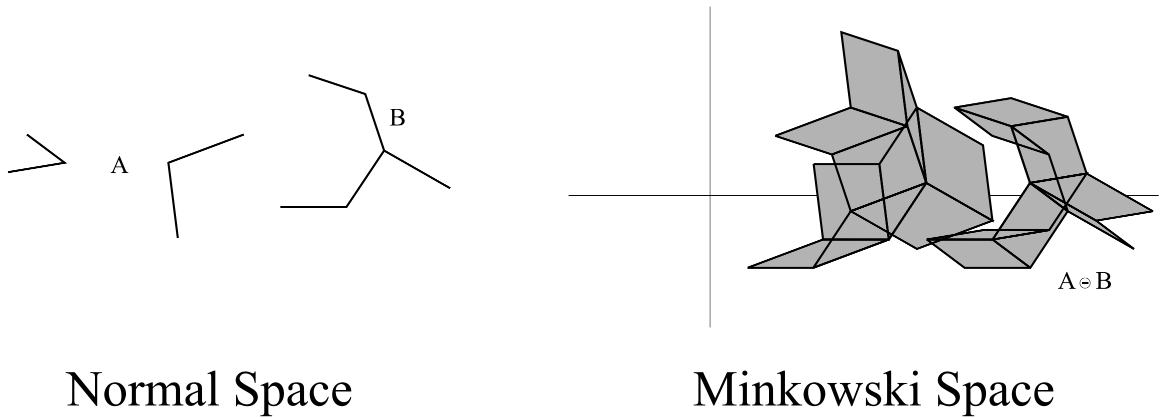


Figure A.3: The Minkowski difference of two models depends on their shapes and their relative placement. Model A has 4 components, and model B has 5 components, and the Minkowski difference has $4 \times 5 = 20$ components.

is the union of parts $a_0, a_1, \ldots, a_n$ and B is the union of $b_0, b_1, \ldots, b_m$, then

$$A \ominus B = \left(\bigcup_i a_i\right) \ominus \left(\bigcup_j b_j\right) = \bigcup_{i,j}(a_i \ominus b_j)$$

So $A \ominus B$ can be viewed as the union of mn components. Figure A.3 shows the models used in the sphere tree example of Section 2.1 in normal space and in Minkowski space. The Minkowski difference of A and B is shown as the union of the Minkowski differences of the components taken pairwise. With 4 line segments in A and five line segments in B, the Minkowski difference is a union of 20 parallelograms.

A.2.1 Collision Queries and Minkowski Space

Since,

$$P = A \ominus B = \{\, \mathbf{a} - \mathbf{b} \mid \mathbf{a} \in A, \mathbf{b} \in B\}$$

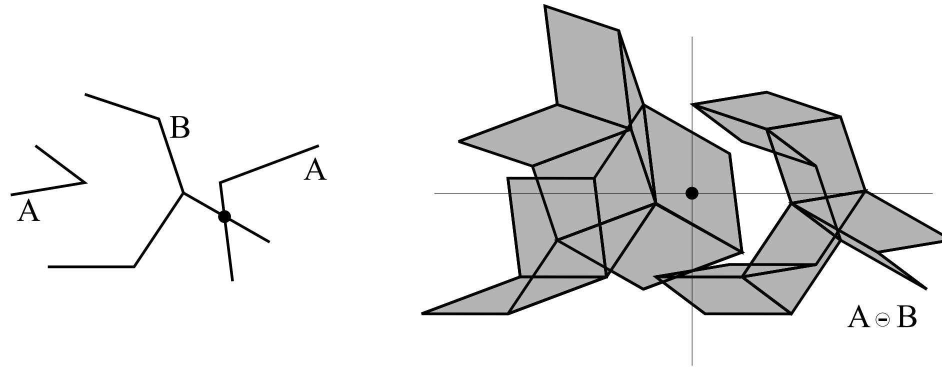


Figure A.4: The Minkowski difference of intersecting models touches the origin. The components of the Minkowski difference covering the origin identify all the primitive pairs which touch.

then P contains the origin if and only if A and B touch. If P contains the origin, then that means there exists points $\mathbf{a}$ and $\mathbf{b}$ such that $\mathbf{a} - \mathbf{b} = \mathbf{0}$, which means $\mathbf{a} = \mathbf{b}$, which means A and B have points in common. The converse is also true: if A and B have two points in common, then there exists $\mathbf{a}$ and $\mathbf{b}$ such that $\mathbf{a} = \mathbf{b}$, which means the point $\mathbf{a} - \mathbf{b}$ will be added to P, and this point is the origin.

Figure A.3 shows A and B disjoint, and the Minkowski difference does not include the origin. In Figure A.4 we show the same two models, but positioned so as to intersect. The Minkowski difference covers the origin. The parallelogram which covers the origin is the Minkowski difference of the two line segments which touch.

A.2.2 Separation Distance Queries and Minkowski Space

The distance between two point sets equals the distance of their Minkowski difference from the origin. The distance between A and B is defined as the shortest distance between all pairs of points. This is written

$$\mathrm{dist}(A, B) = \min_{a \in A, b \in B} |a - b|.$$

Now consider the distance of $P = A \ominus B$ from the origin,

$$\mathrm{dist}(\mathbf{0}, P) = \min_{p \in P} |p|$$

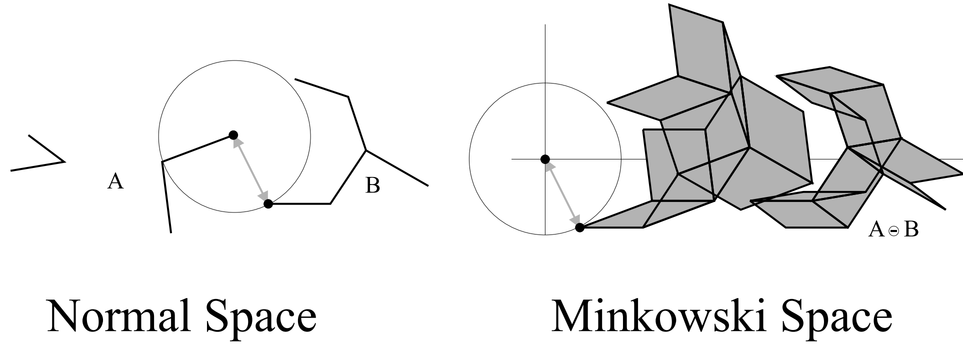


Figure A.5: The point of the Minowski difference closest to the origin corresponds to the points of closest approach of the two point sets, indicated by the gray arrow.

The point $\mathbf{p} \in P$ closest to the origin identifies the points in A and B which are closest together, as shown in Figure A.5.

A.2.3 Spanning Distance Queries and Minkowski Space

The spanning distance of two point sets corresponds to the most distant point of their Minkowski difference from the origin. This is illustrated in Figure A.6. The proof has exactly the same structure as that for separation distance.

A.2.4 Penetration Distance Queries and Minkowski Space

The penetration distance of set A and B is the shortest possible translation of one of them which makes them disjoint. This corresponds to the point in P' closest to the origin, where P' is the set complement of $P = A \ominus B$. The penetration distance is defined as

$$\text{pen}(A, B) = \text{shortest } v \text{ such that} \min_{a \in A} \min_{b \in B} |a - b + v| > 0.$$

Let us define $A+\mathbf{v}$ as being the set of points from A shifted by $\mathbf{v}$. Then the penetration distance can be written as

$$\text{pen}(A, B) = \text{shortest } v \text{ such that } \mathbf{0} \notin (A + \mathbf{v}) \ominus B$$

Since

$$(A + \mathbf{v}) = \{\mathbf{a} + \mathbf{v} \; | \mathbf{a} \in A\},$$

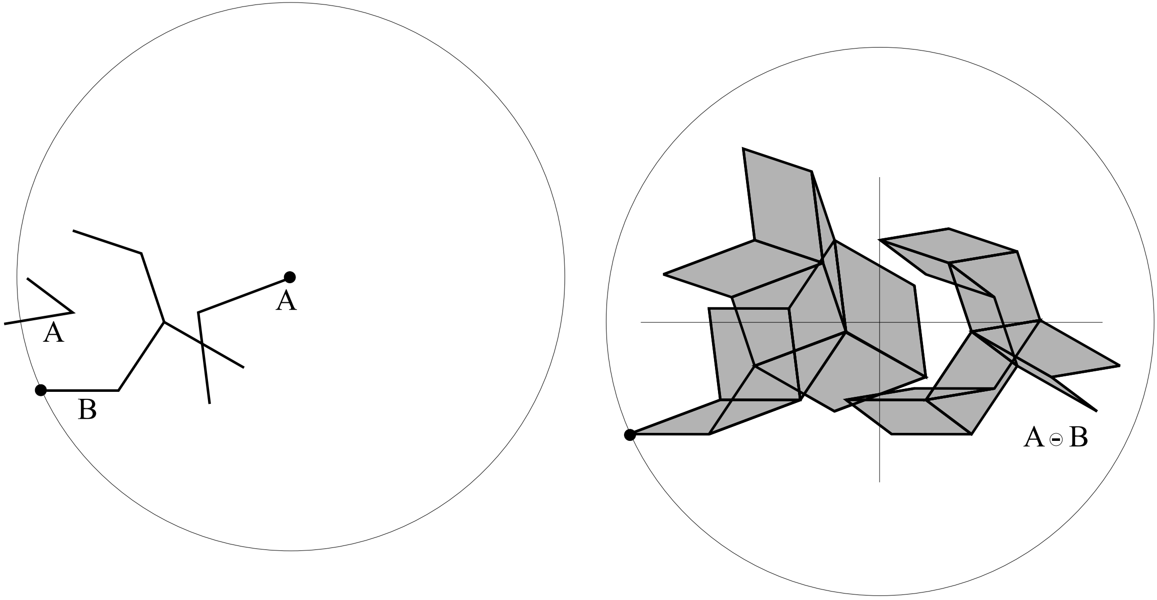


Figure A.6: The point in the Minowski difference furthest from the origin corresponds to the points of greatest separation of the two point sets, indicated by the circles.

we have

$$(A + \mathbf{v}) \ominus B = \{\mathbf{a} + \mathbf{v} - \mathbf{b} | \mathbf{a} \in A, \mathbf{b} \in B\} = (A \ominus B) + \mathbf{v}.$$

So translating A causes $A \ominus B$ to translate by the same amount in the same direction. Similar reasoning shows that translating B causes $A \ominus B$ to translate the same amount in the opposite direction. Looking at the Minkowski difference of touching models, we see that the minimum distance to translate A to separate the models is also the minimum translation of $A \ominus B$ to uncover the origin, which is the closest point outside of $A \ominus B$ to the origin.

A.2.5 Hausdorff Distance Queries and Minkowski Space

The Hausdorff distance of A from B is the maximum distance of any point on A from the model B. One way to view this is to grow B until it entirely encloses A. Growing B is done by taking the Minkowski sum of B and an origin-centered ball of radius r, denoted Ω_r. Increasing r causes the sum to "fatten", the Hausdorff distance is the smallest r such that $B \oplus \Omega_r$ entirely covers A. This view is shown in the "Normal Space" side of Figure A.8. Another way to view this is by examining the family of

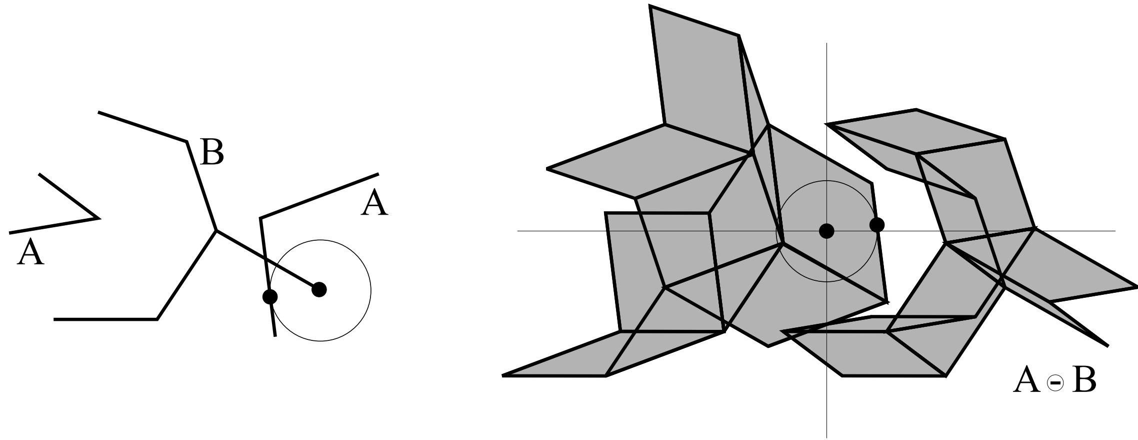


Figure A.7: The penetration distance corresponds to the closest point outside the Minkowski difference to the origin, as indicated by the circles.

sets $B \ominus \{\mathbf{a}\}$, where $\mathbf{a} \in A$, in which each member is a shifted version of B. The Hausdorff distance is the radius of the smallest origin-centered ball which touches every member of the family. This view is shown in the "Minkowski Space" side of Figure A.8.

A.3 Accelerating Collision, Separation and Span Queries

In this section we discuss how collision, separation, and span queries can be accelerated with bounding volumes. We also discuss why these BVH-based methods do not work as well for penetration and Hausdorff distance queries.

A.3.1 Collision Queries

As mentioned in Section 2.1.2, the contacts found among two models can be expressed recursively as the union of the contacts found among their partitions. That is, if a set of points A can be partitioned into subsets $P_1, P_2, \ldots, P_s$, and a set B can be

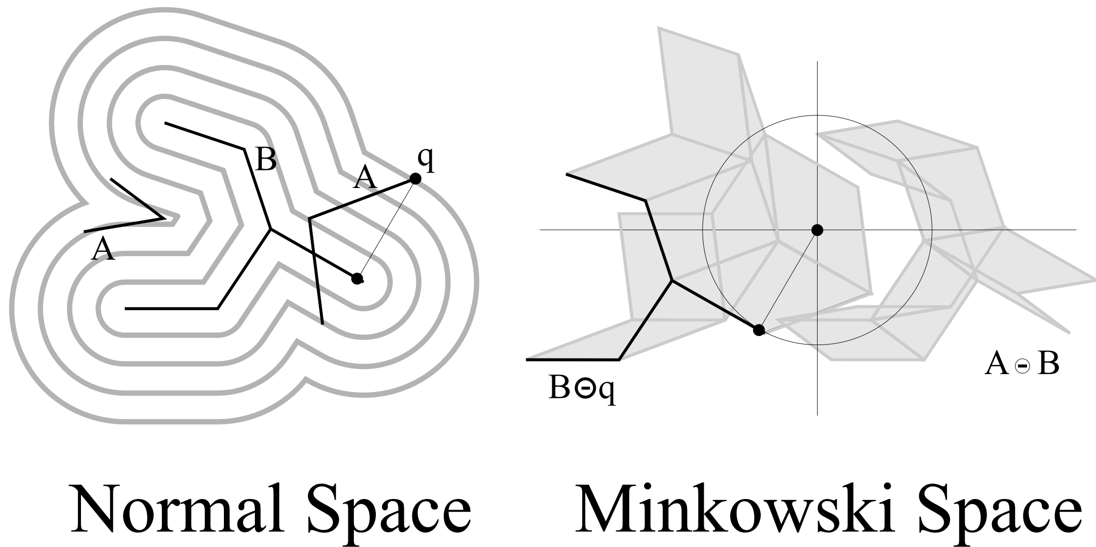


Figure A.8: The Hausdorff distance of A from B corresponds to the last point of A covered as B is progressively expanded. This last covered point is the point on A which is most distant from B, and this distance is the Hausdorff distance.

partitioned into subsets $Q_1, Q_2, \ldots, Q_t$, then

$$\text{cont}(A, B) = \bigcup_{i,j} \text{cont}(P_i, Q_j).$$

This is the basis of the divide-and-conquer approach to the problem. For polygonal models, the partitions can correspond to collections of polygons. The partitioning is performed recursively until we reach $\text{cont}(a_i, b_j)$, which is the contact between just two polygons. If model A has m polygons, and B has n polygons, then we can write an expression tree which has mn leaves, and $mn - 1$ internal nodes.

Explicitly writing out the expression tree and evaluating it would take $O(mn)$ work, which is equivalent to simply testing all the polygons pairwise. We can accelerate the evaluation by testing the bounding volumes of subsets, which amounts to a trivial test for whether contacts exist between them,

$$\text{cont}(V_A, V_B) = \emptyset \Rightarrow \text{cont}(A, B) = \emptyset.$$

Performing such tests enables us to prune the expression tree during evaluation.

The overlap test for bounding volumes is equivalent to testing whether the Minkowski difference of the bounding volumes contains the origin. The Minkowski difference of

the bounding volumes covers the Minkowski difference of the geometry they bound – so $V_A \ominus V_B$ is a bounding volume for $A \ominus B$ in Minkowski space, and the process of traversing the bounding volume test tree resembles a process of traversing a bounding volume hierarchy of $A \ominus B$, and terminating recursion branches when a Minkowski BV does not contain the origin.

A.3.2 Separation Distance Queries

An exact separation distance query also has a recursive partitioning identity, which is that the separation distance between two models equals the minimum of the separation distances among its partitions, as expressed in

$$\text{dist}(A, B) = \min_{i,j} \text{dist}(P_i, Q_j).$$

The expression tree deriving from this recursive partitioning can also be pruned, by noting that the distance between two subsets is bounded from below by the distance between their bounding volumes:

$$\text{dist}(V_A, V_B) \leq \text{dist}(A, B)$$

For example, we know that

$$\text{dist}(A, B) = \min(\text{dist}(A_0, B), \text{dist}(A_1, B))$$

If $\text{dist}(A_0, B)$ evaluates to a distance d, and we find that $\text{dist}(V_{A_1}, V_B) > d$, then that implies that

$$d < \text{dist}(V_{A_1}, V_B) \leq \text{dist}(A_1, B)$$

which means that we need not evaluate $\text{dist}(A_1, B)$ because it cannot possibly be lower than d.

The evaluation proceeds by progressively revising an upper bound on the distance between the two models. This can be imagined as a progressively shrinking ball centered at the origin of the Minkowski space. The distance between two bounding volumes V_A and V_B equals the distance of their Minkowski difference $V_A \ominus V_B$ from the origin. Since $V_A \ominus V_B$ is a superset of $A \ominus B$, whenever $V_A \ominus V_B$ does not touch the ball, we know that $A \ominus B$ cannot touch the ball, and therefore cannot be used to revise the bound downward.

A.3.3 Spanning Distance Queries

The spanning distance is in some sense a dual of the separation distance. The recursive partition expression is

$$\mathrm{span}(A, B) = \max_{i,j} \mathrm{span}(P_i, Q_j).$$

and the pruning expression is

$$\mathrm{span}(A, B) \leq \mathrm{span}(V_A, V_B)$$

The approach is otherwise exactly the same. When evaluating

$$\mathrm{span}(A, B) = \max(\mathrm{span}(A_0, B), \mathrm{span}(A_1, B))$$

if we find that $\mathrm{span}(A_0, B)$ evaluates to a distance d, and if we find that $d \geq \mathrm{span}(V_{A_1}, V_B)$, then we know that

$$d \geq \mathrm{span}(V_{A_1}, V_B) \geq \mathrm{span}(A_1, B)$$

and that $\max(\mathrm{span}(A_0, B), \mathrm{span}(A_1, B))$ simplifies to d.

The evaluation proceeds by progressively revising a lower bound on the spanning distance. This can be viewed as an ever widening "antiball" (the set complement of a ball – which covers all of space further than a given distance from the origin) in Minkowski space. The span of two bounding volumes equals their maximum distance from the origin of their Minkowski difference, and this Minkowski difference entirely covers the Minkowski difference of their bounded geometry. Therefore, if $V_A \ominus V_B$ fits does not touch the antiball (that is, it fits entirely within the radius of the antiball), then $A \ominus B$ also does not touch it, in which case it cannot be used to revise the lower bound on the spanning distance.

A.3.4 Penetration Distance Queries

The separation and spanning distances could be shown graphically as extremal points of the Minkowski differences, and their evaluation could be formulated as an expression tree involving recursive partitionings, and this tree could be pruned during evaluation by employing bounding volume measures.

The penetration distance was shown to be the closest point to the origin outside the Minkowski difference. Thus, it is neither a maximum nor a minimum distance point among the components. Furthermore, the search for a witness point in Minkowski space for the penetration distance is more easily expressed negatively than positively – that is, the components of the Minkowski difference of the two models enumerate where the witness point is not located.

This makes a straightforward formulation of the penetration distance in terms of bounding volume hierarchies very difficult.

A.3.5 Hausdorff Distance Queries

As for penetration distance, a suitable formulation for Hausdorff distance in terms of recursive partitions has not been found. It is the case that

$$\mathrm{haus}(A, B) = \bigcup_i \mathrm{haus}(P_i, B).$$

This expresses the Hausdorff distance in terms of partitions of A, but not B, which simplifies the problem somewhat, but does not enable us to break the problem all the way down to pairs of primitives: we can go as far as a primitive of A against the entire model of B.

The separation and spanning distances of two models bounds their Hausdorff distances:

$$\mathrm{dist}(A, B) \leq \mathrm{haus}(A, B) \leq \mathrm{span}(A, B)$$

This information can be used to potentially eliminate some pairs of partitions of the models from consideration.

Bibliography

[AK89] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. In *An Introduction to Ray Tracing*, pages 201–262, 1989.

[Bar90] D. Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. *ACM Computer Graphics*, 24(4):19–28, 1990.

[BCG+96] 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.

[BW90] Jules Bloomenthal and Brian Wyvill. Interactive techniques for implicit modeling. In Rich Riesenfeld and Carlo Sequin, editors, *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, volume 24, pages 109–116, March 1990.

[Cam84] Stephen Cameron. *Modelling Solids in Motion.* PhD thesis, University of Edinburgh, 1984.

[Cam85] S. Cameron. A study of the clash detection problem in robotics. *Proceedings of International Conference on Robotics and Automation*, pages 488–493, 1985.

[Cam91] S. Cameron. Approximation hierarchies and s-bounds. In *Proceedings. Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 129–137, Austin, TX, 1991.

[Cam97] S. Cameron. Enhancing gjk: Computing minimum and penetration distance between convex polyhedra. *Proceedings of International Conference on Robotics and Automation*, pages 3112–3117, 1997.

[CC86] S. Cameron and R. K. Culley. Determining the minimum translational distance between two convex polyhedra. *Proceedings of International Conference on Robotics and Automation*, pages 591–596, 1986.

[CLMP95] 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.

[DK90] D. P. Dobkin and D. G. Kirkpatrick. Determining the separation of preprocessed polyhedra – a unified approach. In *Proc. 17th Internat. Colloq. Automata Lang. Program.*, volume 443 of *Lecture Notes Comput. Sci.*, pages 400–413. Springer-Verlag, 1990.

[Don84] B. R. Donald. Motion planning with six degrees of freedom. Master's thesis, MIT Artificial Intelligence Lab., 1984. AI-TR-791.

[Ede83] H. Edelsbrunner. A new approach to rectangle intersections, Part I. *Internat. J. Comput. Math.*, 13:209–219, 1983.

[Far93] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide.* Academic Press Inc., 1993.

[FKN80] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Proc. of ACM Siggraph*, 14(3):124–133, 1980.

[FNO89] R.T. Farouki, C.A. Neff, and M. O'Connor. Automatic parsing of degenerate quadric-surface intersections. *ACM Transactions on Graphics*, 8:174–203, 1989.

[GASF94] A. Garcia-Alonso, N. Serrano, and J. Flaquer. Solving the collision detection problem. *IEEE Comput. Graph. Appl.*, 14:36–43, May 1994.

[GJK88] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between objects in three-dimensional space. *IEEE J. Robotics and Automation*, vol RA-4:193–203, 1988.

[GLM96] S. Gottschalk, M. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. In *Proc. of ACM Siggraph'96*, pages 171–180, 1996.

[HKM95] M. Held, J.T. Klosowski, and J.S.B. Mitchell. Evaluation of collision detection methods for virtual reality fly-throughs. *Canadian Conference on Computational Geometry*, 1995.

[HLC+97] T. Hudson, M. Lin, J. Cohen, S. Gottschalk, and D. Manocha. V-collide: Accelerated collision detection for vrml. In *Proc. of VRML Conference*, pages 119–125, 1997.

[HLMD96] M. Hughes, M. Lin, D. Manocha, and C. Dimattia. Efficient and accurate interference detection for polynomial deformation and soft object animation. In *Proceedings of Computer Animation*, pages 155–166, Geneva, Switzerland, 1996.

[HMPY97] C. Hu, T. Maekawa, N. Patrikalakis, and X. Ye. Robust interval algorithm for surfaces intersections. *Computer-Aided Design*, 29(9):617–627, 1997.

[Hof89] C.M. Hoffmann. *Geometric and Solid Modeling.* Morgan Kaufmann, San Mateo, California, 1989.

[HSS83] J.E. Hopcroft, J.T. Schwartz, and M. Sharir. Efficient detection of intersections among spheres. *The International Journal of Robotics Research*, 2(4):77–80, 1983.

[Hub93] P. M. Hubbard. Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*, October 1993.

[KHM+96] J. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. In *Siggraph'96 Visual Proceedings*, page 151, 1996.

[KKM97] J. Keyser, S. Krishnan, and D. Manocha. Efficient and accurate B-rep generation of low degree sculptured solids using exact arithmetic. In *Proc. Symposium on Solid Modeling and Applications*, pages 42–55, 1997.

[KPLM98] S. Krishnan, A. Pattekar, M. Lin, and D. Manocha. Spherical shell: A higher order bounding volume for fast proximity queries. In *Proc. of Third International Workshop on Algorithmic Foundations of Robotics*, pages 122–136, 1998.

[LC91] M.C. Lin and John F. Canny. Efficient algorithms for incremental distance computation. In *IEEE Conference on Robotics and Automation*, pages 1008–1014, 1991.

[Lin93] M.C. Lin. *Efficient Collision Detection for Animation and Robotics.* PhD thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, December 1993.

[LM95] M.C. Lin and Dinesh Manocha. Fast interference detection between geometric models. *The Visual Computer*, 11(10):542–561, 1995.

[LPW79] T. Lozano-Pérez and M. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Comm. ACM*, 22(10):560–570, 1979.

[LR80] J.M. Lane and R.F. Riesenfeld. A theoretical development for the computer generation and display of piecewise polynomial surfaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(1):150–159, 1980.

[MC95] B. Mirtich and J. Canny. Impulse-based simulation of rigid bodies. In *Proc. of ACM Interactive 3D Graphics*, Monterey, CA, 1995.

[Meg83] N. Megiddo. Linear-time algorithms for linear programming in r^3 and related problems. *SIAM J. Computing*, 12:pp. 759–776, 1983.

[MG91] J. Miller and R. Goldman. Combining algebraic rigor with geometric robustness for the detection and calculation of conic sections in the intersection of two quadric surfaces. *Proc. Symposium on Solid Modeling and Applications*, pages 221–233, 1991.

[Mir98] Brian Mirtich. V-Clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 17(3):177–208, July 1998.

[Nay90] B.F. Naylor. Binary space partitioning trees as an alternative representation of polytopes. *Computer-Aided Design*, 22(4), 1990.

[Ove92] M. H. Overmars. Point location in fat subdivisions. *Inform. Proc. Lett.*, 44:261–265, 1992.

[Pat93] N.M. Patrikalakis. Surface-to-surface intersections. *IEEE Computer Graphics and Applications*, 13(1):89–95, 1993.

[Qui94] S. Quinlan. Efficient distance computation between non-convex objects. In *Proceedings of International Conference on Robotics and Automation*, pages 3324–3329, 1994.

[RR92] A.A.G. Requicha and J.R. Rossignac. Solid modeling and beyond. *IEEE Computer Graphics and Applications*, pages 31–44, September 1992.

[Sed90] T.W. Sederberg. Techniques for cubic algebraic surfaces. *IEEE Computer Graphics and Applications*, pages 14–25, July 1990.

[Sei90] R. Seidel. Linear programming and convex hulls made easy. In *Proc. 6th Ann. ACM Conf. on Computational Geometry*, pages 211–215, Berkeley, California, 1990.

[SJ91] C. Shene and J. Johnstone. On the planar intersection of natural quadrics. *Proc. Symposium on Solid Modeling and Applications*, pages 234–244, 1991.

[Sys97] SOLID Interference Detection System. http://www.win.tue.nl/cs/tt/gino/solid/, 1997.

[Tur89] G. Turk. Interactive collision detection for molecular graphics. Master's thesis, Computer Science Department, University of North Carolina at Chapel Hill, 1989.

[ZSP93] J. Zhou, E.C. Sherbrooke, and N.M. Patrikalakis. Computation of stationary points of distance functions. *Engineering with Computers*, 9(4):231–246, 1993.