Collision Detection and Proximity Queries
SIGGRAPH 2004 Course

Course Category

Animation – physical simulation

Course Organizers

Sunil Hadap and David Eberle
PDI/DreamWorks

Length

Half-Day

Summary Statement

This course will give an authoritative overview of various collision detection techniques. The proponents and expert practitioners from academia and industry will cover widely accepted and proved methodologies in detail. Additionally, advanced or nascent exciting topics will be touched upon. The methods discussed will be tied to their most appropriate applications and compared.

Speakers

Sunil Hadap
R&D Staff - Dynamics, PDI/DreamWorks

Dave Eberle
R&D Staff - Dynamics, PDI/DreamWorks

Pascal Volino
Senior Research Assistant, MIRALab, University of Geneva

Ming C. Lin
Associate Professor of Computer Science, UNC, Chapel Hill

Stephane Redon
Research Associate - Computer Science, UNC, Chapel Hill

Christer Ericson
Senior Principal Programmer, Sony Computer Entertainment America

Course Abstract

This course will primarily cover widely accepted and proved methodologies in collision detection. In addition more advanced or recent topics such as continuous collision detection, ADFs, and using graphics hardware will be introduced. When appropriate the methods discussed will be tied to familiar applications such as rigid body and cloth simulation, and will be compared. The course is a good overview for those developing applications in physically based modeling, VR, haptics, and robotics.

An essential task of most collision detection schemes involves determining whether two geometric primitives are intersecting. Higher level concepts such as the separating axis theorem and ray intersection will be covered. General strategies for efficient implementation of these tests will be discussed and concise references to specific tests will be provided.

Algorithms devised to reduce the number of expensive primitive tests will be covered. Bounding volume hierarchies for deformable and rigid geometry will be described and compared. The sweep and prune algorithm used to reduce the number of object pairs that need to undergo further testing will also be discussed.

Instead of determining whether two objects intersect, another strategy is to determine the proximity between them. The GJK algorithm will be discussed as an efficient method of performing this task on convex geometry. More general feature tracking methods will also be discussed.

A common problem that shows up in many applications with collision detection is that of temporal aliasing. If objects are moving too fast between collision detection calls, many techniques will fail to report a collision. Continuous methods offer a solution to this problem. In addition to being more robust they have the ability to provide very accurate contact information, which is essential to many simulation applications. Continuous techniques for deforming and rigid geometry will be discussed, along with strategies for their efficient implementation.

Collision detection techniques that provide a measure of penetration depth are useful in a variety of applications. Adaptively sampled distance fields provide a means to quickly determine penetration depth and direction for collision response. Techniques of how to build ADFs will be discussed along with their suitability in applications. Recent advances in GPU based collision computation will be introduced.

Prerequisites

Elementary geometry, introduction to data structures, linear algebra, and penchant for collision detection.

Level of difficulty

Intermediate-advanced

The course covers intermediate to advanced level topics. Thus depending on the exposure to the problem of collision detection, attendees will benefit from different aspects of the course. Some background in physically based simulation, VR, haptics or robotics is expected in order to fully appreciate the course.

Intended audience

The course will serves as a boot-camp for beginners in the field. The experienced developers and researchers in physically based modeling, VR, haptics, and robotics will find the course a comprehensive overview. Further, they will find it a good forum with a comparative study of numerous collision detection techniques and their most appropriate applications. The target audience also includes effects developers, technical directors and other researchers having interest in spatial data structures.

Course Structure

Session I – Introduction

- **Overview [Sunil Hadap 25 min]**

  This section will cover the breadth of the course. It will give a broad overview of the various techniques, their comparison and discusses which technique is best suited in what situation. The overview will put all the disparate set of topics discussed subsequently into perspective. Further, for the sake of completeness, the overview will also touch upon any topic that is not discussed in greater details.

- **Primitive Tests [Dave Eberle 25 min]**

  In this section, higher level concepts such as the separating axis theorem and few typical primitive tests such as ray-primitive intersection will be introduced. We will not attempt to discuss specific tests in detail as there are too many to cover. References to precise and optimal tests will be provided for completeness and convenience.

Session II - Broadphase and Midphase Optimizations

- **Collision Algorithms for Deformable Models [Pascal Volino 40 min]**

  This section will first introduce the concept of bounding volume hierarchies. Then it will cover topics such as Sphere Trees and AABB Trees. Choice of AABBs for deformable models will be discussed along with an introduction of surface hierarchies. Specific examples of application of these techniques to cloth simulation will be discussed. More advanced aspects such as "Self-Collision Detection" and k-DOPs will be touched upon.

- **Collision Algorithms for Rigid Bodies [Ming C. Lin 25 min]**

  This section primarily compares the optimization techniques previously introduced, specifically for rigidbodies. Then it will cover the optimizations developed specifically for rigid-body motion such as Broadphase Sweep and Prune, and OBB Trees. Session III - Algorithms for Rigid Convex Objects

- **Feature Tracking [Ming C. Lin 15 min]**

  This section will introduce an efficient method for tracking the proximity of convex rigid objects using Voronoi regions.

- **GJK [Christer Ericson 25 min]**

  This section will describe the Gilbert-Johnson-Keerthi (GJK) algorithm.

GJK is one of the most efficient algorithms for finding the smallest distance (and closest points) between the convex hulls of two point sets, and in a generalized version, between arbitrary closed convex bodies.

Session IV - Advanced Topics

This session will cover techniques which are more recent in either their development or acceptance. The continuous collision methods resolve the problem of temporal aliasing. ADFs have merit in that they provide penetration depth information crucial to some collision response models. The exciting advent of GPU computation for collision computation will be covered in the closing topic.

- **Continuous Collision Detection for Deforming Geometry [Dave Eberle 10 min]**

- **Continuous Collision Detection for Rigid Geometry [Stephane Redon 25 min]**

- **Adaptively Sampled Distance Fields [Sunil Hadap 10 min]**

- **Collision Detection using GPU [Ming C. Lin 10 min]**

Course History

There are a number of great SIGGRAPH courses on physically based modeling, rigid-body simulation, deformable models, virtual reality and haptics. However, to the best of our knowledge, no SIGGRAPH course has exclusively covered the important topic of collision detection and proximity queries so far. This is the primary motivation for organizing the course. Our dream is for the course to serve as boot camp for aspiring researchers interested in physically based modeling. Experts may find this course a handy reference to established as well as recent techniques.

List of courses from SIGGRAPH 2003 that are closely related

- Physically Based Modeling, David Baraff and Andrew Witkin
- Geometric Data Structures for Computer Graphics, Gabriel Zachmann and Elmar Langetepe
- Clothing Simulation and Animation, Hyeong-Seok Ko and David Breen
- Photorealistic Hair Modeling Animation and Rendering, Nadia Magnenat-Thalmann

These courses, through their limited coverage, clearly demonstrate the importance of collision detection. Hence we feel a dedicated course on the topic is of great complementary value. An attendee of these courses would immensely benefit from level of detail in our course.

Speakers

Sunil Hadap
R&D Staff - Dynamics, PDI/DreamWorks
Sunil Hadap is a member of the R&D Staff at PDI/DreamWorks, developing next generation special effects tools for use in upcoming film productions. His research interests include a wide range of physically based modeling aspects such as clothes, fluids, rigid body dynamics and deformable models. Sunil Hadap has completed his PhD in Computer Science from MIRALab, University of Geneva under the guidance of Prof. Nadia Magnenat-Thalmann. His PhD thesis is on Hair Simulation. During his doctoral research, he pioneered techniques for hairstyling and hair animation. The hairstyling technique, hair-as-streamlines-of-fluid-flow, was published in Eurographics Workshop on Computer Animation and Simulation, 2000. Further, an elaborate method for single hair stiffness dynamics was proposed along with hair-hair interaction by considering hair as a continuum. The results were published in Eurographics 2001. Naturally, collision detection and response is the topic of his expertise and research interest. At SIGGRAPH 2003 he was the presenter for the course - Photorealistic Hair Modeling, Animation and Rendering.

Dave Eberle
R&D Staff - Dynamics, PDI/DreamWorks
Dave Eberle is a part of the dynamics R&D group at PDI/Dreamworks developing a new suite of character dynamics tools. Prior to this position he worked on the Havok SDK team and at Disney/TSL on the dynamics team. His work was presented as part of a SIGGRAPH sketch in 2002 about the making of the film “Reign of Fire.” In a sketch entitled “A Procedural Approach to Modeling Impact Damage” presented at SIGGRAPH 2003 he presented a novel approach to generating fracture effects using the Havok rigid body SDK. He graduated from Texas A&M with a Master’s in Scientific Computation. His interest in collision detection stems from his thesis work on rigid body simulation with Prof. Donald House of the Visualization Lab. Further, during is experience at Disney and Havok he has implemented and mastered many of the techniques discussed in this course.

Pascal Volino
Senior Research Assistant, MIRALab, University of Geneva
Pascal Volino is a senior research assistant at MIRALab, University of Geneva. He is one of the early pioneers of collision detection and response methodologies for deformable objects. He continues to work on new models for cloth animation, involving versatile models for efficient simulations on situations involving high deformation, wrinkling and multilayer garments. His research is particularly focused on data structure, efficient collision detection, robust simulation and interactive cloth manipulation. His work contributes to several European projects, involving creation and simulation of virtual garments.

Ming C. Lin
Associate Professor, Department of Computer Science, UNC, Chapel Hill
Ming C. Lin received her Ph.D. from the University of California, Berkeley and is currently an associate professor in the Department of Computer Science at the University of North Carolina, Chapel Hill. Her research interests include real-time 3D graphics, applied computational geometry, physically based modeling, and robotics. She has received several honors and awards, including NSF Young Faculty Career Award and Hettleman Award for Scholarly Achievements. She has served as a program committee member for many leading conferences on virtual reality, computer graphics, robotics and computational geometry. She was the general chair and/or the program chair of the First ACM Workshop on Applied Computational Geometry, 1999 ACM Symposium on Solid Modeling and Applications, the Workshop on Intelligent Human Augmentation and Virtual Environments 2003, ACM SIGGRAPH/EG Symposium on Computer Animation 2003, and ACM Workshop on General-Purpose Computing on GPUs 2004. She also serves on the Steering Committee of ACM SIGGRAPH/EG Symposium on Computer Animation. She is an associated editor and a guest editor of several journals and magazines, including IEEE TVCG, International Journal on Computational Geometry and Applications, special issues on HAPTIC RENDERING and TOUCH-ENABLED INTERFACES of IEEE CG&A, and ACM Computing Reviews. She is the project leader of nine public domain collision detection systems: I-Collide, RAPID, VCollide, PQP, SWIFT, SWIFT++, PIVOT, DEEP and CULLIDE. She has also given several invited lectures at conferences, including SIGGRAPH, GDC, Solid Modeling, etc.

Stephane Redon
Research Associate, Department of Computer Science, UNC, Chapel Hill
Stephane Redon is currently a postdoctoral research associate at the University of North Carolina at Chapel Hill, working with Ming C. Lin in the GAMMA project. He graduated from Ecole Polytechnique, France, in 1998, and received his MS in 1999 at Pierre and Marie Curie University, Paris, France. He received a PhD in Computer Science in 2002 at INRIA-Rocquencourt, France, while working with Sabine Coquillart and Abderrahmane Kheddar on robust interactive simulation of rigid body systems and its applications to virtual prototyping and animation. His research interests include the design of robust real-time virtual environments, haptics, and physically-based modeling. His current research focuses on continuous collision detection and its applications, as well as scalable algorithms for complex dynamics systems.

Christer Ericson
Senior Principal Programmer, Sony Computer Entertainment America
Christer Ericson is a senior principal programmer at Sony Computer Entertainment America in Santa Monica, where he is the tools and technology group lead for internal development. He received his Masters degree in computer science from Umeå University, Sweden, where he lectured for several years before moving to the US in 1996. He is currently finishing up a book on real-time collision detection to be published by Morgan Kaufmann.

Course Organizers Contact

Sunil Hadap
shadap@pdi.com
Phone : +1-650-5629234

Dave Eberle
deberle@pdi.com
Phone : +1-650-5629093

PDI/DreamWorks
1800 Seaport Blvd.
Redwood City, CA 94063, USA

Primitive Tests for Collision Detection

David Eberle
PDI/Dreamworks R&D

A broad level taxonomy of the essential low level algorithms used in performing collision detection is presented. This section is intended for a beginners in the field of collision detection and can probably be skimmed by experienced readers. For many applications the techniques discussed in this chapter are enhanced by ones presented later in the course. Algorithms and code for precise tests are provided in the references as a convenience for the reader.

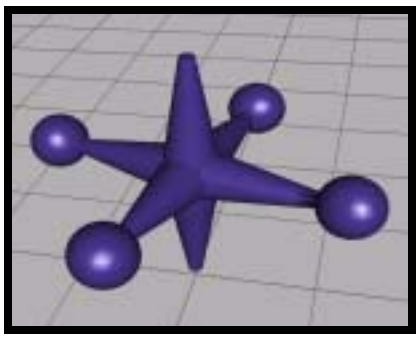
1 What is Collision Detection?

At a first glance the answer to question above seems quite trivial. Given two objects of given representations, which may not be the same, determine if they collide. As a coding practitioner one typically treats time discretely which makes this answer ambiguous. Is the answer referring to a future tense, past tense, or present tense state of the objects? Also how do we define collision? Are we asking if the objects intersect at one of the previous three locations in time? When considering collisions with respect to our everyday perception a more concise answer would be the task of determining over a given time interval whether any points of the two objects occupy the same location in space simultaneously.

Later in these course notes we will see that providing an algorithm to perform this precise task can be non-trivial. Due to the complexity associated with answering that question, practitioners over the years have come up with a number of techniques which provide answers to alternative questions which have served a variety of applications quite well. In this course we will attempt to cover the major categories of these methods.

2 What are Primitives?

The objects one can describe using modern computer graphics techniques can have seemingly limitless complexity. Complex objects are typically a combination of simpler object representations called primitives. Though it is a simple object, the model jack shown below can be represented as a union of implicit cones and spheres, a triangle mesh, or by a collection of parametric patches to name a few. Clearly a parametric patch is more complex than a collection of triangles making the term primitive a relative one. Typically the term primitive is reserved to describe a basic shape representation that is associated with simple and fast algorithms.

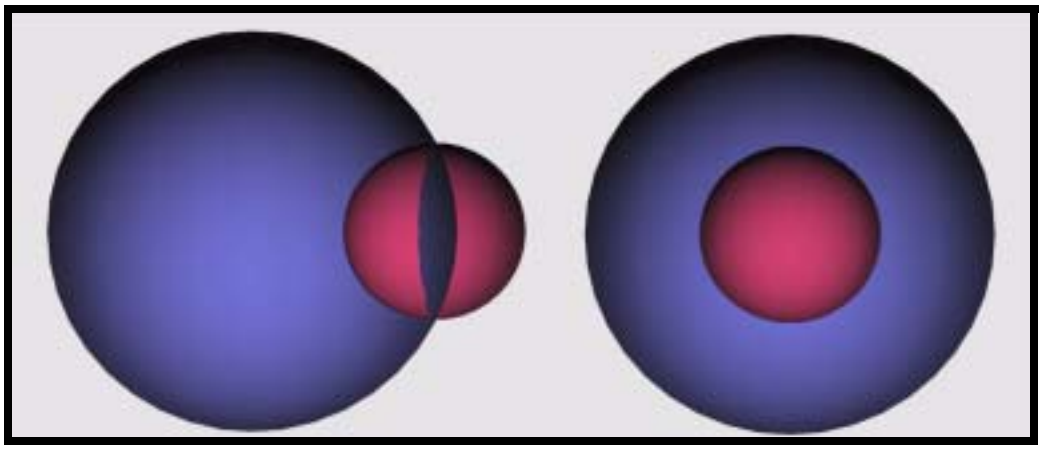


The techniques for collision detection between primitives vary widely between the representation of our objects and the information needed by the application. Consider the application of a game involving flying spaceships. If we make the decision to explode the ships in the event of a collision, then an intersection test that returns a simple boolean may suffice. However if we are dealing with a car racing application and the cars are not old Ford Pintos, we probably want the vehicles to bounce off each other in some physically plausible fashion. To provide this functionality we will need information about the point of collision between the objects so a boolean intersection test is no longer sufficient.

The factor of speed many times determines the object representation and what kind of collision detection tests can be afforded by a given application. Balancing the priorities of speed, accuracy and functionality is a daunting, but necessary task. In an attempt to aid the reader in this task the remainder of this chapter will focus on describing a variety of techniques while exposing their strengths and weaknesses with respect to each other and certain application domains.

3 Static Intersection Tests

In computer graphics the term intersection test, usually refers to the task of determining if two geometric primitives intersect at a specified discrete instant in time. Consider the case of determining whether two implicitly defined spheres intersect. This can easily be done by checking the distance between the centers against sum of the radii. Now consider the same test with each sphere represented as a triangle mesh. Suppose all pairs of triangles between the two objects are checked for intersection. If no pairs intersect do the spheres not intersect? The answer is no. If the surface geometry intersects as in the image on the left the two tests will report the same answer. Consider the case on the right where



one sphere is entirely embedded in the other. No triangles intersect so the second test returns false while the first test returns true, because the second representation omits the interior region of the objects. In practice all triangle pairs are not tested when the object is represented as a mesh. Methods to prune the number of primitive tests are covered in other portions of this course and are omitted here. In the first test the sphere is the primitive and in the second the primitives are the triangles of the objects. Clearly if you have an application that uses or can approximate objects by spheres using an implicit representation and the first test is desirable. Spheres can be too simple for many applications however, but this demonstrates how the representation of our objects can dictate the testing strategy.

Other implicitly defined shapes, such as cones, cylinders, planes, and boxes are commonly used a primitives for collision detection. There are too many specific intersection tests to cover so the reader is deferred to the references to find more information about a specific pair wise primitive intersection test.

Since many intersection tests make use of the separating axis theorem covering it seems more prudent. The separating axis theorem can be stated as follows:

> *For any two arbitrary convex, disjoint, polyhedra **A** and **B**, there exists a separating axis where the projections of the polyhedra, which form intervals on the axis, are also disjoint. If **A** and **B** are disjoint, they can be separated by an axis that is orthogonal to plane formed by one of the following:*
>
> 1. *A face normal of polyhedron **A**.*
> 2. *A face normal of polyhedron **B**.*
> 3. *A normal formed by the cross product of a pair of edges with one from **A** and the other from **B**.*

By using the separating axis test between two convex polyhedra we can determine the boolean state of whether they overlap/intersect. Taking the example of two triangles one would need to test the eleven separating axes. The first two being from the triangle normals and the other nine from the unit vectors formed by the pair wise edge cross products. If the projected geometry intervals overlap on all of these axes, then the triangles must intersect. When dealing with more complex polyhedra it can be advantageous to cache the last valid separating axis and see if it results in an immediate rejection in later queries.

Many static intersection tests only return a boolean status of the intersection. As a result they typically lack information that could be used to resolve the intersecting state. Using a simulation application's perspective, the objects start in a valid disjoint state and the future state is deemed invalid by the presence of an intersection. Historically one strategy for a simulator has been to revert the state to a valid one and step forward with a smaller step. If an intersection is found this process happens again reducing the time step further. If no intersection is found then another step is taken forward. The stopping criteria for this can also vary. The application may choose to bisect time until Δt is below some threshold. Other tests may be employed to query the distance between the objects and use this as the stopping criteria [BA89]. This can be done more efficiently by assuming the collision occurred on the last valid separating axis. The accuracy, robustness and efficiency of this type of usage is highly questionable however. If the geometry is moving fast or if your simulator has many colliding objects these strategies can often break down in practice or require additional algorithms to make things tractable [MI00,RKC02].

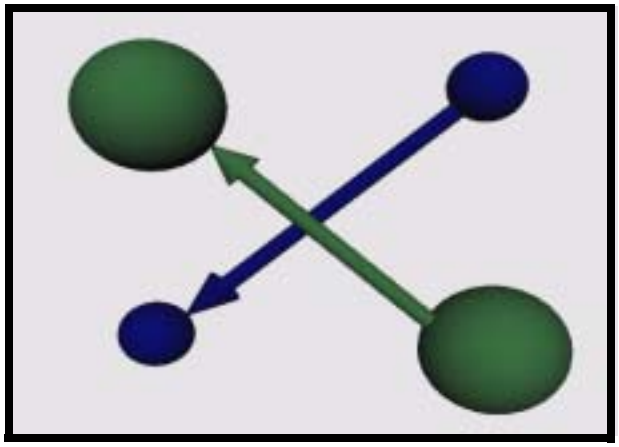
Just because static intersections generally do not provide geometric data about the collision event to separate the objects, this does not make them less important. They provide a cheap computational means for determining whether two objects do not intersect. If the objects intersect then other techniques can be employed to gather the information needed for a some kind of response. Therefore it is best to look at static intersections as one type of test in a collection of tests used to handle the problem of collision detection.

4 Moving Intersection Tests and Continuous Collision Detection

Many collision detection techniques examine the state of the geometry at discrete instances in time. If the objects are moving towards each other too fast relative to their original distance and size the geometry may not intersect at the sampled times and the collision events will be missed. This problem is referred to as temporal aliasing. In many applications this problem is ignored because it is of no use responding to collisions the user did not even see. Other applications bound the speed at which objects can move or sample for intersection more frequently to alleviate this problem.

One way to test if the objects may have intersected would be to extrude the geometry over the relative motion of the objects and perform a static test on it. In many cases this can get quite complex when the motion of the objects is non-linear. Furthermore this would at most tell the application that a collision event occurred but not when or at what points on the objects it occurred. Also in many tests of moving primitives one object is held fixed while the other object is given the relative motion between the two objects. Unfortunately many of these tests only report an boolean result and do not report the first time and point of contact. A collection of moving primitive tests that fall into this category can be found at [HM04].

In many situations there is a way to precisely determine if two objects intersected over a continuous interval of time. These methods are generally more complex, but they have the added advantage of providing the time and point of collision between the primitives. As a basic example lets revisit implicit sphere-sphere intersection.



Define the center of the first sphere as a linear function of it's beginning and end center positions.

$$c_1(t) = c_{begin} + t \cdot (c_{end} - c_{begin})$$

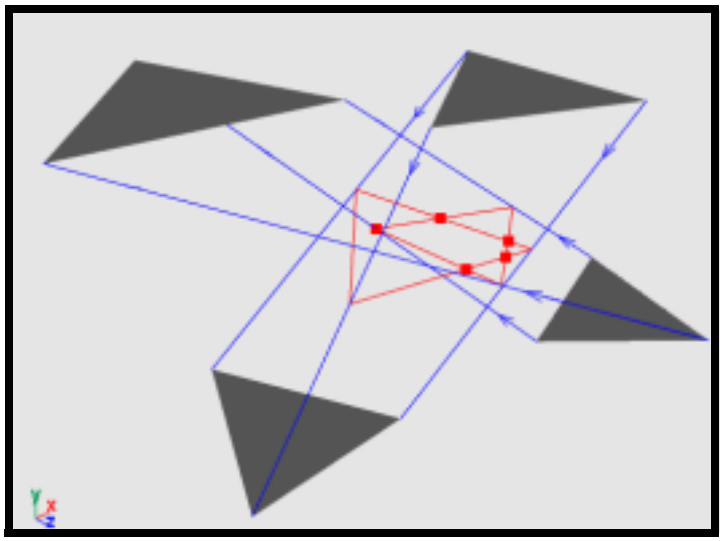
Then define the location of the center of he second sphere as $c_2(t)$ in an analogous fashion. In the static intersection test the state of the following inequality was a sufficient test.

$$\|c_2 - c_1\| - (r_2 + r_1) \leq 0$$

In the dynamic test this equation is revised to the following equality which yields a quadratic polynomial $(c_2(t) - c_2(t))^2 - (r_2 + r_1)^2 = 0$.

The roots can be solved for analytically and the smallest root $t \subset [0,1]$ gives is the time when the spheres are in contact at a single point. To find the point one advances the spheres to the position at time t and determines where the line between the two centers intersects the surfaces. Similar techniques yielding the same information have been developed for polygonal geometry under deforming and rigid motions [MW88,PR97,RKC02].

These techniques are have been used in cloth and rigid body simulation in recent years with much success and are covered in future chapters of these course notes. To wet the appetite of the reader an image of the temporal collision detection between two deforming and moving triangles is presented.



While it appears that these techniques are the exact answer to the more precise query of what it means to perform collision detection, they do have a drawback. By changing the inequality of the static test to an equality in the dynamic test something has been lost. If the spheres start in a state of penetration the dynamic test will not report this as a collision. the dynamic test now only looks for the instances in time when the surfaces are in touching contact.

At first this may not seem like a concern. One can make the restriction to the user than the geometry always start in a non-penetrating state. In practice this restriction has proven to be an unreasonable one because users can choose to ignore it and blame the programmer when things turn out wrong. The programmer would also need to be very confident about their collision response strategy. If a collision is not resolved properly it is not likely to be detected at the next query. So while these methods are very robust and can provide a lot of information other techniques lack, in the opinion of the author they are by no means the final solution to the collision detection needs of the reader. They do offer a solution to the problem of temporal aliasing and can be coupled with proximity queries and/or penetration depth techniques to create a robust collision detection system such as the one described in [BFA02].

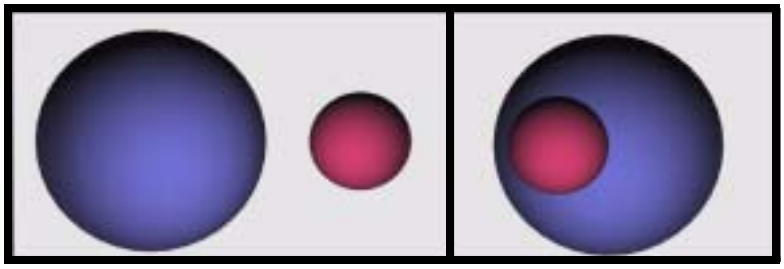
5 Proximity Queries and Penetration Depth

Instead of querying for intersection some applications choose to determine collision by keeping track of the minimum distance between objects. If the minimum distance is below a threshold then a collision is reported at the point of this minimum distance between the two objects.

To ensure that this point of minimum distance is uniquely defined these testing methods are typically used for testing between convex polyhedra. Distances are computed between edge pairs and vertex-faces pairs of the polyhedra. Algorithms such as [MI97] have been developed to the number of distance computations between these primitive features and are discussed in a later portion of this course.

In the context of some simulations, the motion of the objects is typically bounded so the objects cannot move into a state that puts them closer than some minimum distance. While this technique has the advantage of never letting the objects penetrate, it usually means reducing the time step which can potentially slow down the speed of the simulator.

Another popular algorithm for performing proximity queries between convex polyhedra is the GJK algorithm which is also covered in further detail in a later chapter. This algorithm can be extended to give a negative signed minimum distance when the objects are penetrating. Details of the penetration depth algorithm can be found in [VD03]. Adding this functionality removes the need for one to strictly bound the time step of a simulation. Applications can let the objects penetrate and then use penetration depth information as a means to determine the response to a collision. While penetration depth can provide information for a collision response, it is not necessarily always and accurate one. Consider small sphere moving from right to left in the picture below.



If the direction of minimum distance is used to determine the direction in which to displace the objects, the smaller sphere would come out the wrong side of the larger one. In some simulation applications one can use the velocities to in an attempt and alleviate this problem, but if the objects are given in a penetrating state, the response will be completely non-physical.

A more general way to handle penetration depth is by using distance fields. This technique does not impose a convexity restriction on the geometry and is discussed in the advanced topics portion of this course. Simulation applications such as [GBR03] and HA03] have used distance fields to compute signed minimum distance of mesh primitives for robust collision detection and response.

6 Ray-Primitive Tests and Regular Height Fields

Until this point primitives have been presented as shapes or elemental components of our model geometry such as edges and faces. A different type of primitive called a ray is now introduced. The ray $r(t)$ is defined by the following equation:

$$r(t) = o + t \cdot d \quad t \subset [-\infty, \infty]$$

The point o is defined as the origin of the ray and d is a unit vector direction. If a ray intersects an object at a time/parameter $t < 0$ then the intersection point is behind the origin. Typically we are only concerned with intersections $t \subset [0, \infty]$.

In many situations it is enough to know if a ray from a point intersects an object. The common game application of a player shooting something is readily handled by this type of query. In [HM99] the usage of a ray test method to keep a moving car above a terrain is discussed. Testing rays from points on the four wheels against the terrain is more efficient than doing static intersection tests between the actual geometry of the car and terrain. Furthermore the ray tests provide points of intersection which can be used to compute the distance between the wheel and the surface where as the static intersection tests would not provide this information. Using the distances and the signs of the parameter t at the point of collision one can adjust the position of the wheels to keep them above the terrain surface.

Terrain models are often described by a special type of data structure called a height field. A regular height field is a two dimensional array of altitude values spaced at regular intervals. One can think of it as a piece of graph paper with a scalar value at the grid points giving the height of the surface. This representation has the limitation that the terrain cannot have any overhangs since there can only be one height value at any point on the grid. The basic ray test essentially works like a Bressenham line algorithm in 2D. The height of the ray at intersection with the cell boundaries is easily computed and the height at the four corners of the cell are known. The ray intersects geometry in the cell only if

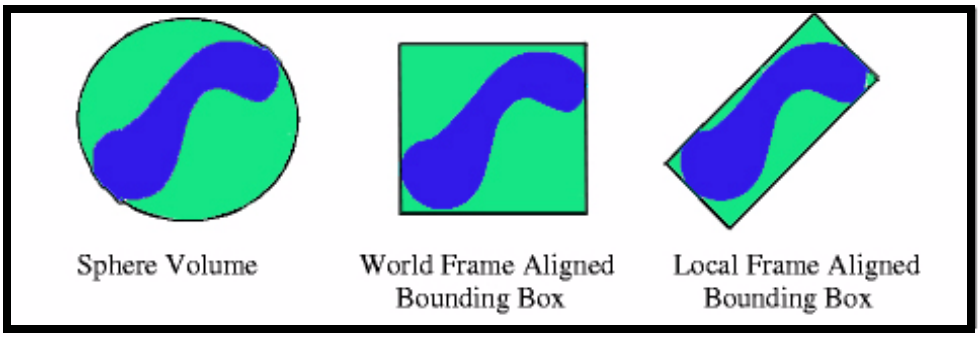
$$min(ray_{in}, ray_{out}) \leq max(h_{i,j}, h_{i,j+1}, h_{i+1,j}, h_{i+1,j+1})$$

where ray_{in} and ray_{out} are the heights of the ray at the intersection points of the 2D cell and the h values on the right hand side are the heights at the corners of the grid cell the ray travels through. Further details of the algorithm such as finding the intersection with the geometry in the cell can be found in [MU88]. The author in [MU88] readily admits the technique presented is not the most efficient traversal algorithm known, but the associated representation of the height field has lower memory requirements than the known competing techniques. Coupled with ray intersection, height fields are another valuable primitive representation for applications needing collision detection with landscapes.

Ray-primitive intersections have been widely used and published in the context of ray-tracing for rendering images [GL89]. A large collection of code and pseudo-code can be found in these references[HM99,ES02,VD03]. Ray tests for moving primitives have also been devised and are available in [HM04].

7 Primitives as Bounding Volumes

In many algorithms simple primitives are used as bounding volumes for more complex objects. Below are a few examples of common bounding volumes fit around an arbitrary object. The primary goal of using the bounding volume is to get a fast rejection test. If the bounding volumes of two objects do not overlap it is known that the objects themselves do not intersect. If the bounding volumes do overlap then further testing is required. This is a category of usage where the cheap static intersection test that returns a boolean becomes an important asset.



Typically the more complex objects are broken down into a hierarchy of bounding volumes. This strategy is covered in a future section of the course. At the bottom of such a hierarchy a primitive of one type bounds another. An example of this would be using axis aligned bounding boxs, AABBs, to bound triangles. The overlap test between two AABBs is far less expensive than the triangle-triangle intersection test.

When choosing a primitive to be used as a bounding volume the following three things must be considered.

1. How well does the bounding volume fit the underlying geometry?

2. What is the cost in updating the bounding volume if the object is moving?

3. What is the cost of the bounding volume intersection test.

Looking at the figure above and considering the first question, one notices that the sphere and axis aligned bounding box, AABB, have a lot of empty space not occupied by the geometry inside. This means that many bounding volumes will overlap more often when the geometry inside does not overlap thereby increasing the number of unnecessary tests of the geometry inside. The number of bounding volume intersection tests may also increase because they occupy more space.

The second question is a concern when the objects are moving. As the objects move the bounding volumes need to be updated. In the case of the sphere this is trivial. The bounding volume translates the same amount as the object inside. In the AABB case, the minimum and maximum extents of the geometry with respect to the world coordinate frame

must be updated. In the case of the box oriented to the object, commonly referred to as the oriented bounding box or OBB, the update depends on the motion of the object. If the motion is rigid, rotation and translation only, then the transformation must be applied to the bounding volume as well. If the geometry is deforming then a far more expensive update is needed. Creating and updating OBBs is covered in [HM99,GO96].

The third question also factors into our decision of which bounding volume to use. The sphere and AABB have simple overlap tests compared with the OBB. If we end up doing too many tests the added cost of using the OBB may become prohibitive. However this is generally not likely because it is a better candidate with respect to the fit criteria.

It should not be a surprise that there are compromises to be made between the quality of the fit, cost of updating, and cost of the intersecting a given bounding volume. Researchers and practitioners have tried to find an optimal balance between certain types of bounding volumes [VD97,GO96] but this is an ongoing topic of debate and research.

8 General Strategies in Algorithm Design

When designing an intersection test between two primitives there are some general rules of thumb ones should follow. A more thorough list is provided in [HM99].

1. Perform simple tests first that may lead to an early exit of the algorithm to avoid more expensive computations. A simple example in a static intersection test of two triangles would be to first check if all the points of one triangle have a non-zero distance of the same sign to the infinite plane of the other triangle. If this is the case we can exit knowing the triangles do not intersect before checking for edge-face intersections.

2. If possible try to cache results from previous tests. One example is caching the last separating axis from one check to the next. Other geometric elements such as face normals may able to be cached to speed up computations.

3. Try to reduce the dimension of the problem. For example the determination of whether a point is inside a triangle in 3D can be cast into a 2D version by projecting the vertices onto one of the standard planes. Having said this, efficiency still needs to be tested as the math one saves may not improve performance if more branching is required by the lower dimensional test.

In many applications the number of primitive types being tested against one another is quite small. In these situations specific optimized tests can be coded. However if an application has many primitive types and must be able to perform tests between arbitrary pairs this strategy results in a code explosion. It is then probably best to classify primitives/ objects into general classes and use more general techniques. An example would be the use of the GJK algorithm which is discussed later in the course. The algorithm works on general convex polytopes and convex implicit primitives such as boxes, spheres, cones, and cylinders. It is then possible to attempt a general API that would handle queries between all of these primitive types.

9 Summary

In this chapter we covered many types of queries that can be used to aid the task of determining whether objects collide. It has been attempted to convey the type of information each category of tests provides to help the reader make an informed choice for their application. It has been illustrated that what constitutes a collision will vary by application and the category of tests employed.

References and Acknowledgements

Many people over the years have contributed and published algorithms for various intersection and distance computations. In recent years many of these tests have been grouped together in a smaller collection of sources. For the sake of the author's sanity at this hour and convenience to the reader it was decided to reference many of these collective works instead of the original individual publications. The author wishes to thank the authors of the original papers and collective works for their efforts. The author also would like to thank the following people for their suggestions and proof reading of these notes: Ramón Montoya Vozmediano, Carlos Gonzalez-Ochoa and Manuel Kraemer.

[BA89] D. Baraff, Analytical methods for dynamic simulation of non-penetrating rigid bodies. Computer Graphics 23(3), pp. 223-232, 1989.

[BFA02] R. Bridson, R. Fedkiw and J. Anderson, Robust Treatment of Collisions, Contact and Friction for Cloth Animation, SIGGRAPH Proceedings, pp. 596-597, 2002.

[EB00] D. Eberly, 3D Game Engine Design: A Practical Approach to Real-time Computer Graphics, Morgan Kaufmann Publishing, 2000.

[ES02] D. Eberly and P. Schneider, Geometric Tools for Computer Graphics, Morgan Kaufmann Publishing, 2002.

[GBR03] E. Guendelman, R. Bridson, and R. Fedkiw, Nonconvex Rigid Bodies with Stacking, SIGGRAPH Proceedings, pp. 871-878, 2003.

[GO99] M. Gomez, Simple Intersection Tests for Games, Gamasutra, October 1999.

[GO96] S. Gottschalk, M.C. Lin and D. Manocha. OBBTree: A hierarchical structure for for rapid interference detection, ACM SIGGRAPH Proceedings, pp. 171-180, 1996.

[GL89] A. Glassner, An Introduction to Ray Tracing, Academic Press, 1989.

[HA03] S. Hadap, Hair Simulation, Ph.D. Dissertation, CUI, University of Geneva, 2003.

[HE97] M. Held, ERIT - A collection of robust and reliable intersection tests, Journal of Graphics Tools, 2(4), pp. 25-44, 1997.

[HM99] E. Haines and T. Möller, Real-time Rendering 1st Edition, A.K. Peters, pp. 289-361, 1999.

[HM04] E. Haines and T. Möller, Real-time Rendering web page of intersection tests, http://www.realtimerendering.com/int/.

[MI97] B. Mirtich, V-Clip Fast and Robust Polyhedral Collision Detection, ACM Transactions on Graphics, 17(3), pp. 177-208, 1997.

[MI00] B. Mirtich, Timewarp Rigid Body Simulation, ACM SIGGRAGPH Proceedings, pp. 193-200, 2000.

[MU88] F. Kenton Musgrave, Grid tracing: Fast ray tracing for height fields. Technical Report YALEU/DCS/RR-639, Yale University Dept. of Computer Science, 1988.

[MW88] M. Moore, and J. Wilhelms, Collision Detection and Response for Computer Animation, SIGGRAPH Proceedings, pp. 289-297, 1988.

[PR97] X. Provot, Collision and self-collision in cloth models dedicated to design garments, Graphics Interface, pp. 177-189, 1997.

[RKC02] S. Redon, A. Kheddar, and S. Coquillart, Fast continuous collision detection between rigid bodies, In Proc. EUROGRAPHICS, 2002.

[VD97] G. Van Den Bergen, Efficient Collision Detection of Complex Deformable Models using AABB Trees, Journal of Graphics Tools, 2(4), pp. 1-14, 1997.

[VD03] G. Van Den Bergen, Collision Detection in 3D Interactive Environments, Morgan Kaufmann Publishing, 2003.

Collision Detection for Deformable Objects

Pascal Volino
MIRALab, CUI, University of Geneva

Deformable objects cover a large range of applications in computer graphics and simulation, ranking from modeling techniques of curved shapes to mechanical simulation of cloth or soft volumes. Efficient collision detection is used in all these applications for ensuring consistent design and simulation.

Cloth simulation is a typical application that requires collision detection on deformable objects.

1. Collision Detection Strategies for Deformable Objects

Unlike rigid bodies, deformable objects have evolving shapes. In most cases, this implies that their surface are curved. Adequate modeling techniques are needed to describe these objects. Among the most popular, polygonal meshes and implicit surfaces (for example metaballs) are used. Whereas surfaces are usually described as polygonal meshes (usually flat polygons such as triangles or quadrangles, but possibly also curved patches such as Bezier patches or subdivision surfaces), volumes are most of the time also described by their bounding surfaces. Collision detection is usually performed on these surfaces.

The usual complexity of collision detection processes result from the large number of primitives that describe these surfaces. Most of collision detection applications need to compute which polygons of large meshes do actually collide. In most of the cases also, these meshes are animated (through user interaction or simulation processes) and collision detection has to be involved at each steps of these animations for ensuring immediate and continuous feedback to the animation control.

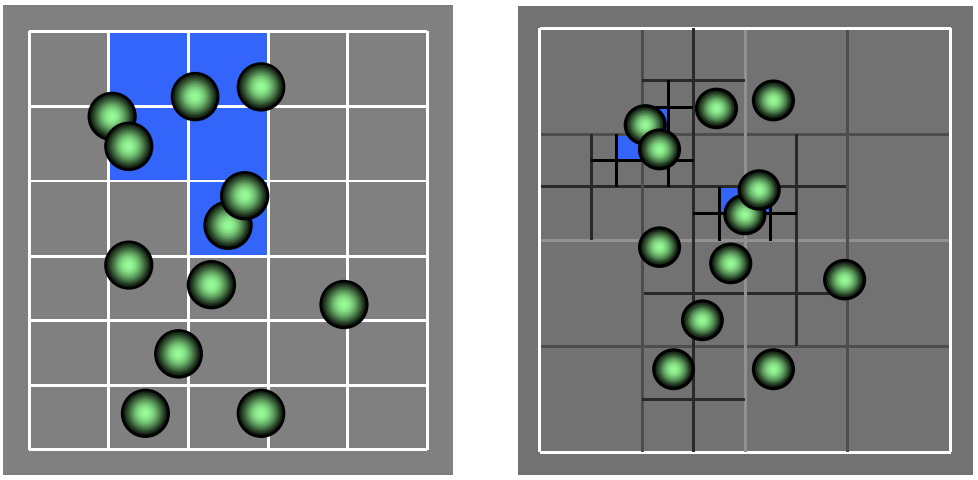
This creates a particular context that collision detection has to take advantage for optimal performance.

1.1. Hierarchy Structure

Most of efficient collision detection algorithms take advantage of a hierarchical decomposition of the complex scheme. This allows to avoid the quadratic time of testing extensively collisions between all possible couples of primitives.

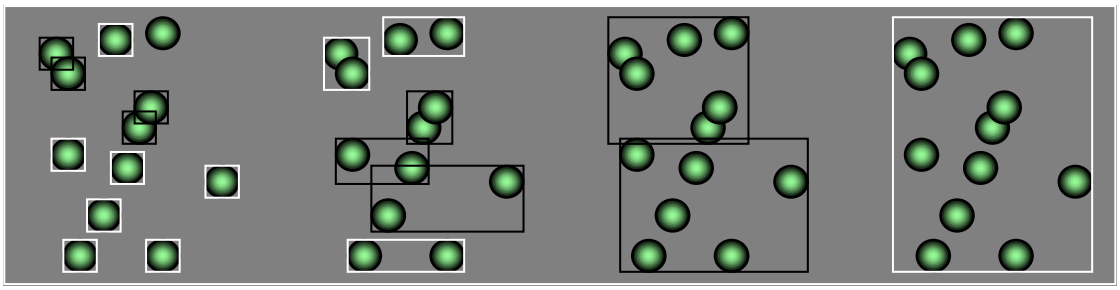
There are two major ways of constructing such hierarchies:

* *Space subdivision schemes, where the space is divided in a hierarchical structure. These are typically octree methods. Using such structure, a reduced number geographical neighbors of a given primitive are found in* ***log(n)*** *time (the depth of a hierarchy separating geographically* ***n*** *primitives) and tested for collisions against it.*



Space subdivision methods rely on a partition of space into regions containing primitives.

* *Object subdivision schemes, where the primitives of the object are grouped in a hierarchical structure. These are typically methods based on bounding volume hierarchies. Using such structure, large bunches of primitives may be discarded in* ***log(n)*** *time (the depth of a well-constructed hierarchy tree of* ***n*** *primitives) through simple techniques such as bounding-volume evaluations.*

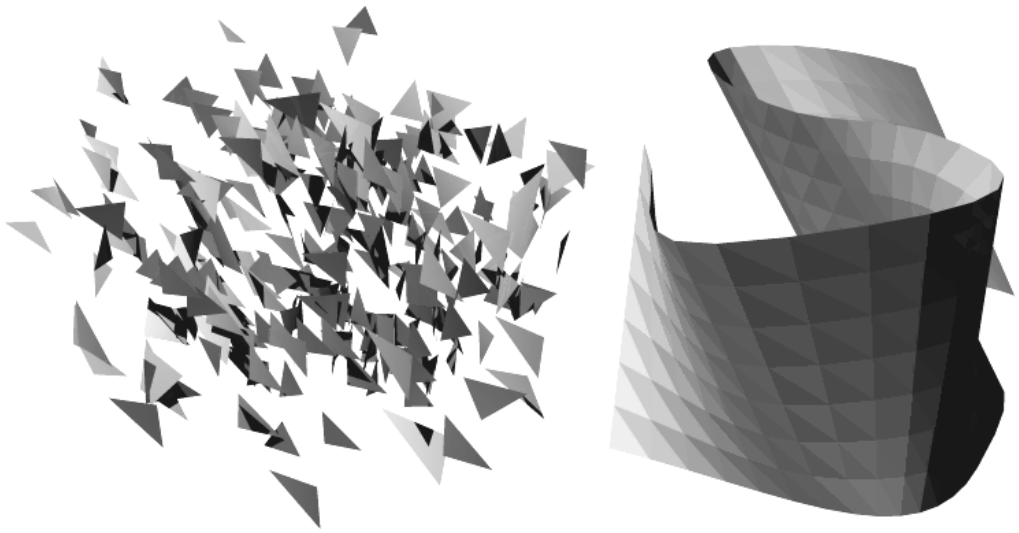


Object subdivision methods rely on a subdivision of the scene into groups of primitives.

Other methods, such as voxel methods, projection methods or render overlap methods, usually belong to non-hierarchical space subdivision schemes.

Any of these methods may be used in the context of deformable objects. However, maximum efficiency is obtained by taking advantage of all invariants that could reduce the amount of time spent in computation. In the context of deformable surfaces which use extensively discretized surface animations (polygonal meshes or patches animated through motion of their control points), the major invariant to take advantage of is *the local invariance of the mesh topology*.

Unlike polygon soups, where the primitives are totally independent one from another, the primitives of a polygonal mesh maintain a constant adjacency structure which defines, in a local state, some constant geographic properties between these primitives. Hence, adjacent elements of a polygonal mesh have similar positions whatever the motion of the surface, during all the animation.



A polygon soup, and a polygonal mesh which maintains local position constancy.

Considering that the topology of the mesh defines a constant native geographical neighborhood of the primitives of the mesh (which are usually non-colliding primitives), a good idea is to define the collision detection hierarchy based on this criteria. On the other hand, any topologically unrelated primitives ("far away" from each other on the mesh topology) which come to be geographically close to each other are very likely to be colliding.

This is why object-based hierarchies are very likely to be the best paradigm for defining an efficient collision detection algorithm for animated meshes. Compared to space subdivision methods, the main benefits are the following:

* *In most cases, they can work on a constant hierarchy structure, which does not need to be updated along the animation.*

* *The topology of the hierarchy reflects the adjacency of the surface regions described in it. Adjacency information may thus be used in various optimizations, such as the approach for optimizing self-collision detection, as discussed later.*

2. Bounding Volume Hierarchies

Hierarchical collision detection heavily relies on bounding volumes. Performance depends on:

* *How tight they enclose the object part corresponding to their hierarchy level, for avoiding at best false positives in the collision detection tests.*

* *How efficiently they can be computed from a primitive.*

* *How efficiently they can be combined, for propagation up in the hierarchy.*

* *How efficiently they can be updated if the object is animated.*

* *How efficiently a collision test can be performed between two volumes.*

The adequate bounding volume is chosen so as to offer the best compromise between these factors, and this is quite dependent on the simulation context (kind and number of object primitives, kind of animation...).

Among popular choices,

* *Bounding spheres or ellipsoids.*

* *Bounding boxes or Discrete Orientation Polytopes.*

2.1. Choosing the Suitable Bounding Volume Scheme

The major choice is to be set between three types of bounding volumes:

* *Axis-independent volumes, such as spheres.*

* *Axis-aligned volumes which are defined in absolute world coordinates.*

* *Object-oriented volumes which are defined in local object coordinates.*

This choice is particularly important when working with animated objects. In that context, the most important issue is to reduce the time taken for updating the bounding volume hierarchy for each step of the animation.

2.1.1. Object-Oriented Volumes

The first idea is to attempt to skip this recomputation whenever possible. This is possible when working with rigid objects, which only have rigid motion in the scene (translation and rotation). In this case, rather than defining the volumes in world coordinates, attaching them to object coordinates removes the need of updating them.

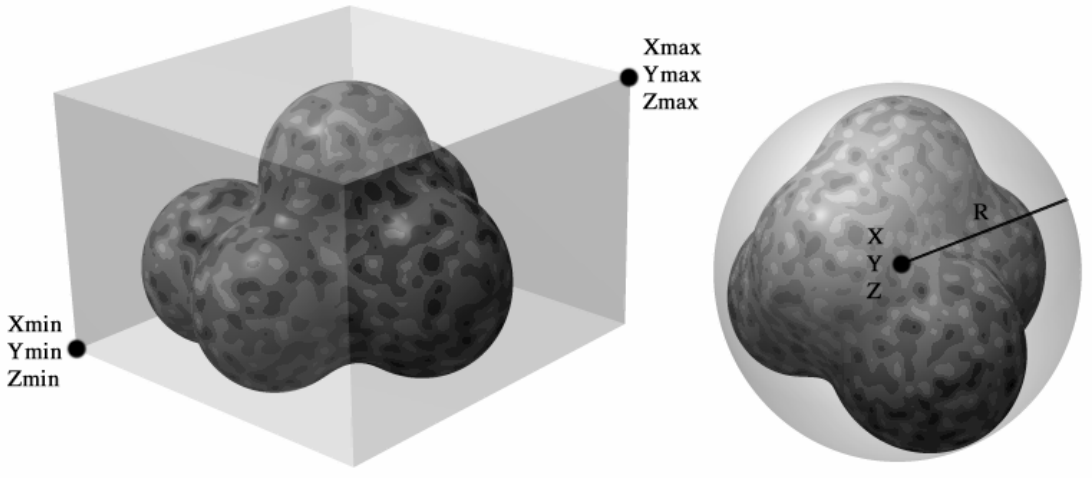
Another motivation for using axis-oriented volume is the optimization of the bounding tightness. Hence, is the object is flat or elongated, there are large benefits in using an adequately aligned volume that fits the shape tightly.

The difficulty is however moved to another place in the collision detection process: Combining and comparing bounding volumes defined in different coordinate systems. While the combination process is usually performed once for all in the hierarchy describing rigid objects, testing for collisions between bounding volumes require coordinate transformation computations that may take significant computation resources. Some schemes also expand the volumes in order to compensate the change of axis rotation, at the expense of bounding tightness. This difficulty can also be avoided by using axis-independent volumes (spheres), but these are very inefficient in bounding tightness (specially for flat or elongated objects).

2.1.2. Axis-Aligned Volumes

The other idea is to make perform extensively the recomputation of the bounding volume hierarchy at each position change, with operations on bounding volumes made as simple and efficient as possible. This is actually the best approach for deformable objects, as there is no way to efficiently exploit shape invariance.

The most popular choice for this are axis-aligned bounding boxes. They are essentially defined by the minimum and maximum coordinates of the enclosed objects, which can be computed very easily. Combining bounding boxes is also trivial, and collision test between two boxes is simply evaluated by testing min-max overlap along all coordinates.



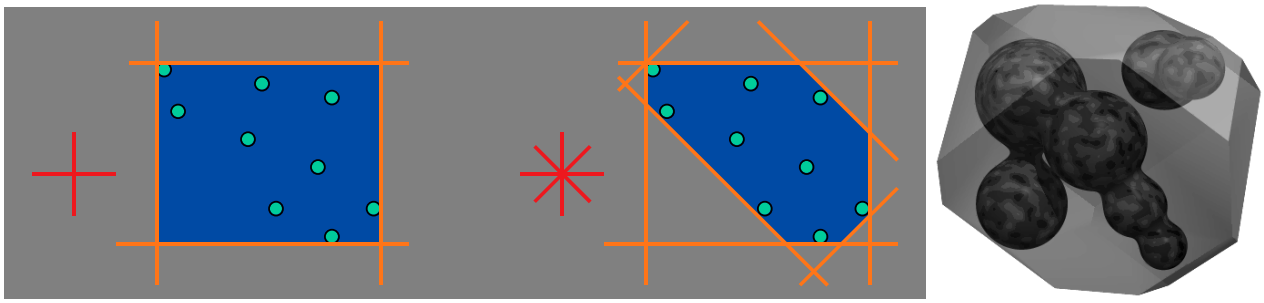
A bounding box and a bounding sphere.

One of the biggest issues with axis-aligned bounding boxes is that they do not always offer good bounding tightness, particularly when objects are flat or elongated along a diagonal direction. Of

course, reverting to their object-oriented variant would void the benefits of fast collision tests. The other solution is to use a generalization of bounding boxes along more directions than the native axes alone.

2.2. Beyond Bounding Boxes: Discrete Orientation Polytopes

Bounding boxes are based on min-max interval computations on the directions defined by the natural world coordinate axes. While this is sufficient for defining bounding volumes, why not adding more directions? It's like "cutting away" a bounding volume along particular directions so as to obtain tighter bounding volumes.



Discrete Orientation Polytopes are generalizations of bounding boxes along arbitrary directions.

2.2.1. Discrete Orientation Polytope Mathematics

Given a set of directions **Di**, a Discrete Orientation Polytope (DOP) a set of vertices **Vk** is defined by two vectors **Mi** and **Ni** such as:

$\mathbf{Mi} = \min_k(\mathbf{Di} \cdot \mathbf{Vk})$ and $\mathbf{Ni} = \max_k(\mathbf{Di} \cdot \mathbf{Vk})$

The union of two DOPs is computed as follows:

$\mathbf{Mi} = \min(\mathbf{Mi_1}, \mathbf{Mi_2})$ and $\mathbf{Ni} = \max(\mathbf{Ni_1}, \mathbf{Ni_2})$

Two DOPs do not intersect if the following condition is met for at least one value of **i**:

$\mathbf{Mi_1} > \mathbf{Ni_2}$ or $\mathbf{Mi_2} > \mathbf{Ni_1}$

2.2.2. Adequate Space Direction Sampling

The adequate set of directions to be considered should describe a sampling of the direction space as uniform as possible. Of course, this sampling is point-symmetric, since a direction vector also represents its opposite direction.

In 3D, the easiest approach is to construct a set of directions starting from the cube (standard bounding box, which is a DOP of 6 directions):

D0=(1,0,0) **D1**=(0,1,0) **D3**=(0,0,1)

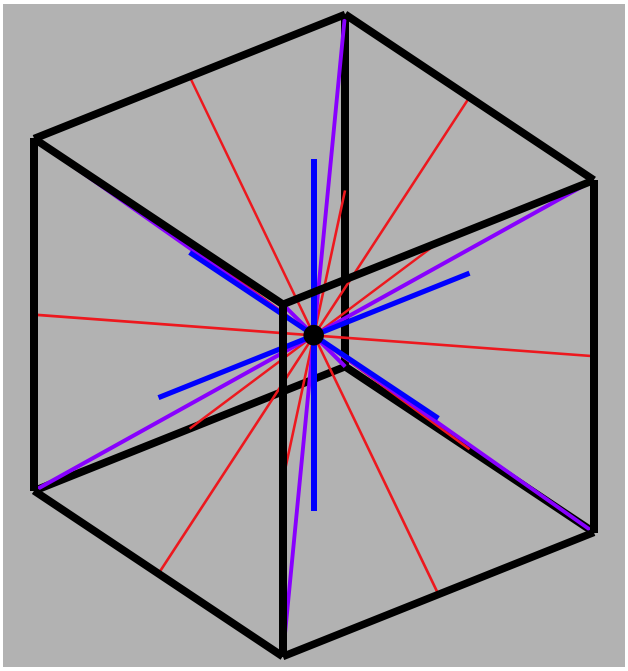
... and add the cube diagonals (directions to its vertices):

D4=(√3,√3,√3) **D5**=(√3,-√3,-√3) **D6**=(-√3,√3,-√3) **D7**=(-√3,-√3,√3)

... for obtaining a DOP describing 14 directions. 12 additional directions can be added by using the square diagonals (directions to its edge centers):

D8=(0,√2,√2) **D9**=(0,√2,-√2) **D8**=(√2,0,√2) **D9**=(-√2,0,√2) **D10**=(√2,√2,0) **D11**=(√2,-√2,0)

Note that for this set, it is not actually necessary to normalize the direction vectors, relieving the necessity of performing multiplications for computing a DOP enclosing a set of vertices. However, normalized direction vectors offer good evaluation of the size of the DOP through the min-max interval width in each direction, as well as a simple expansion scheme for distance-based collision detection.



The 26 direction set defined by a cube.

When more directions are needed, it is also possible to construct a set from a dodecahedron (12 directions), adding to it directions to its vertices (20 additional directions) and to its edge centers (30 additional directions). The benefit of this set is a better distribution of directions. However, multiplications cannot be avoided for computing the DOP of a set of vertices.

Ultimately, with a huge number of directions, DOP tend to get the shape of the convex hull of the enclosed objects.

The choice to be made is actually to find the optimal number of directions defining the DOP. In one hand, the more directions, the tighter the DOP fits to the convex hull of the object (or to the object itself if it is convex). In the other hand, the more directions, the more computation is needed for computing the DOPs and evaluating their intersections. The best choice is actually dependent on the shape of the objects (flat or elongated objects will get more benefits from tight DOPs than round or concave ones), and also the extra cost of performing explicit collision detection on object marked as colliding by rough quickly-evaluated DOPs (for instance, it takes much more time evaluating

collisions between curved patches than flat polygons). The only way of finding the best compromise is to carry out experimental benchmarks on various examples typical of the wanted simulation context.

3. Collision Detection on Polygonal Meshes

Polygonal meshes are the most popular way of describing deformable objects. They may either represent the surface object itself (for example, cloth), or the boundary of a volume object. Collision detection is usually carried out by finding which of the polygons of the meshes are actually colliding.

Objects animated and simulated as polygonal meshes.

Given the considerations discussed above, the typical best choices for detecting collisions on animated polygonal meshes are the follows:

* *Use of a hierarchical bounding volume scheme constructed on the objects.*

* *Use of efficient axis-aligned bounding volumes, such as DOPs.*

Choosing a constant hierarchy built on the mesh seems to be quite a good assumption, as for typical objects, the distance on the mesh is quite a good evaluation of the native distance that should separate features of the mesh, and bounding volumes enclosing a small region of the mesh are likely to remain small for any usual deformation.

1.2. Building Mesh Hierarchies

The performance of the collision detection algorithm is highly dependent on how well-constructed the hierarchy tree is.

The tree should be well conditioned, meaning:

* *Each node should have at maximum O(1) children.*

* *The tree should be balanced, so the tree should have O(log n) maximum depth.*

Another essential quality of the tree is to offer minimal bounding volumes for each tree node. This means that the surface elements represented by a tree node should have maximum vinicity. Thus, it is

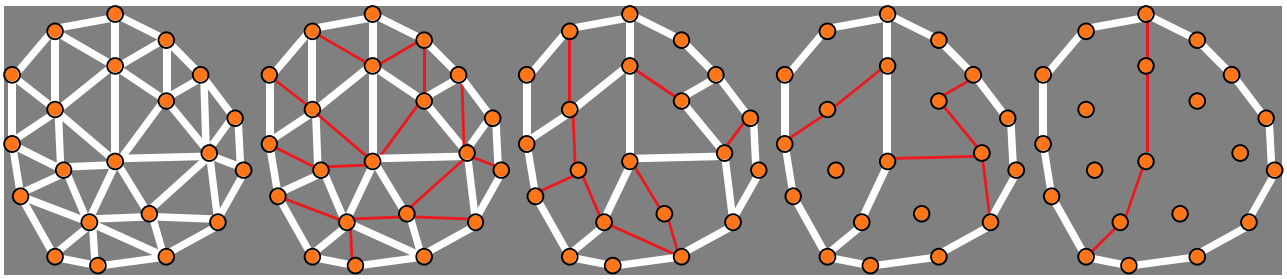
important to build the hierarchy consistently to the topology of the mesh, which is a good evaluation on the relative positioning of the mesh elements.

One efficient solution is to build the hierarchy tree using an ascending process. First the leaf nodes of the tree are constructed, corresponding to all the individual polygons of the surface. Then, an upper layer of the hierarchy is built by grouping two or three nodes in a common parent node. The tree is built level by level until only one element remains, which is the root node of the tree.

The grouping can be performed by a region-merging algorithm. Initially, each surface polygon is assigned a unique region ID, which identifies a group of polygons. During the grouping process, candidate edges that separate two different groups (two polygons having different ID) are considered. One of these edges is then selected, and all the polygons of one group (usually the smallest) are assigned the ID of the polygons of the other group. This algorithm generates groups of connected regions, within each of which all the polygons are connected by at least one edge. A counter should also be included in each group, to keep track of the number of subgroups that have been merged in the group. It can be used to limit the number children a group has. This algorithm is in fact very close to automatic labyrinth-generation algorithms which are based on the same region-merging scheme.

It is important to determine efficiently which nodes to group in order to create the parent node. First, only adjacent surface regions should be connected. Secondly, the regions generated should be as "well shaped" as possible, i.e. closer to a disk than a elongated or sprawling branched structure. The resulting bounding boxes will therefore be as compact as possible, whatever the result of any reasonable 3D deformation of the surface.

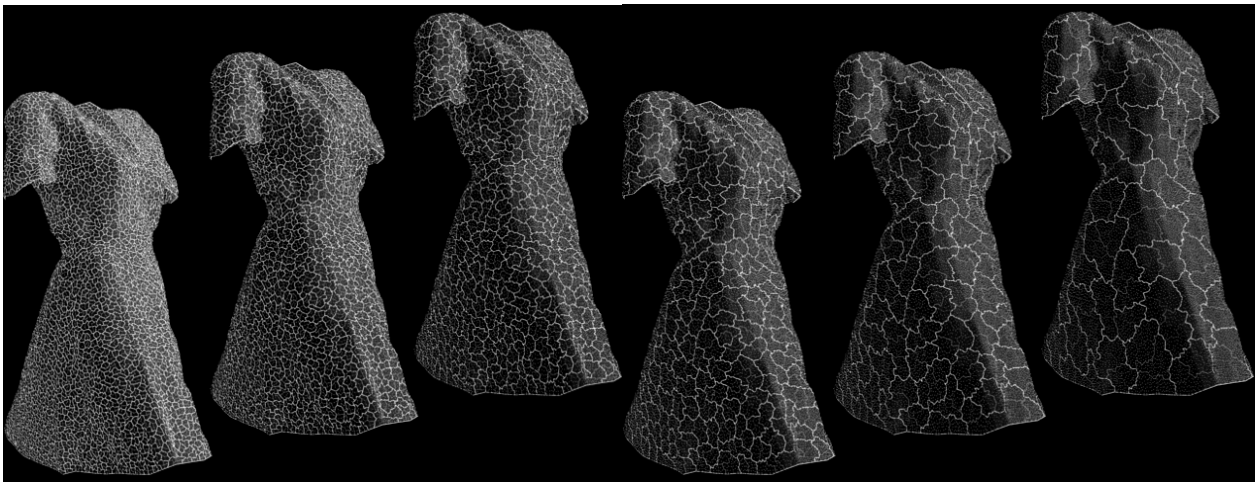
A good way to characterize "well-shaped" polygons is to compare its contour length to its surface area. The algorithm can construct groups by maximizing the "shape factor" **sqrt(SurfaceArea) / ContourLength**. At a given hierarchy level, this ratio is computed for any group that can be generated by the removal of an edge. The potential groups are then sorted by this criteria, and new groups are then constructed wherever possible, according to this sorting. Between two hierarchy levels, the algorithm first tries to merge groups into pairs, and then merges the remaining groups into these new groups.



Building a mesh hierarchy

Though this proposed grouping process does not necessarily yield hierarchy regions that globally optimize the shape factors, the results obtained are however quite acceptable for our application. More precisely, the self-collision detection algorithm remains efficient as long as the bounding volumes of non-adjacent hierarchy groups (that do not share at least a common vertex) do not intersect. Using the proposed algorithm, this feature is verified almost everywhere in usual polygonal meshes.

This algorithm should be implemented to build a hierarchy on any polygonal mesh that will be involved in collision detection. This computation should only be performed as preprocessing, and does not have to be performed for each collision detection when the surfaces have moved.



Hierarchisation of a 50 000 triangle mesh: Levels 5 to 10.

3. Self-Collision Detection

There are very few algorithmic differences between self-collision detection within one object and collision detection between two separate objects. Ultimately, a hierarchical algorithm will end up splitting a single object into separate pieces, and perform usual collision detection between these pieces.

There is actually a major performance issue related to self-collision detection, related to adjacent elements actually seen as "colliding" by usual bounding-volume tests.

3.1. Why Self-Collision Detection is so Inefficient

Self-collision detection pertains to collisions between elements of the same surface. Of course, neighboring elements of the same surface are naturally in contact to each other. Any collision detection algorithm is designed to detect geometric contact between elements, and thus will be misled by these adjacent elements, and will consider them as colliding elements.

The number of adjacencies is usually proportional to the total number of elements in one surface. In a triangular mesh, the adjacency number is roughly 1.5 times the total number of triangles for common-edge adjacencies and 6 times the total number of triangles for common-vertex adjacencies.

The time spent detecting collisions is proportional to the number of colliding elements multiplied by the logarithm of the total number of elements. Typically, the number of self-colliding elements is very small compared to the total number of elements and often null if no collisions occur. Thus, “detecting” all the adjacencies as if they were collisions is a great waste of time, particularly in situations with very few collisions.

For efficiency, an algorithm should be designed to ignore all collisions that in fact are only element adjacencies.

3.2. Optimizing Self-Collision Detection with Curvature Evaluation

3.2.1. The Curvature Criteria

A flat surface cannot have self-collisions. On the other hand, if a surface has self-collisions, it must be bent enough to form a loop.

Is there a way to formalize this intuition? Curvature appears to be the key for achieving efficiency in self-collision detection.

When there are self-collisions on a surface, at any geometrical intersection point, the surface appears twice. For obtaining such configuration, the surface has to be bent enough to form a loop.

Self-collisions only occur if the surface is bent enough for creating a loop.

This condition cannot be met if there exists a direction for which the orthogonal projection of the surface does not exhibit folds. If the surface is continuous, this means that all the normals of the surface have a dot product of constant sign with that direction vector.

We also have to consider self-collisions that may occur because of the shape of the surface boundary. This may also exhibit loops even if the surface is almost flat, when the surface contour itself exhibits a loop causing self-intersection. An additional test has therefore to be done on the surface contour. Having found a projection direction for the surface curvature, a sufficient criteria for non-intersection is that the projection of the surface contour along this direction should not self-intersect.

Thus, a surface does not have self-collisions if the following criteria are met:

* Let **S** be a continuous surface in Euclidean space, delimited by one contour **C**.

if

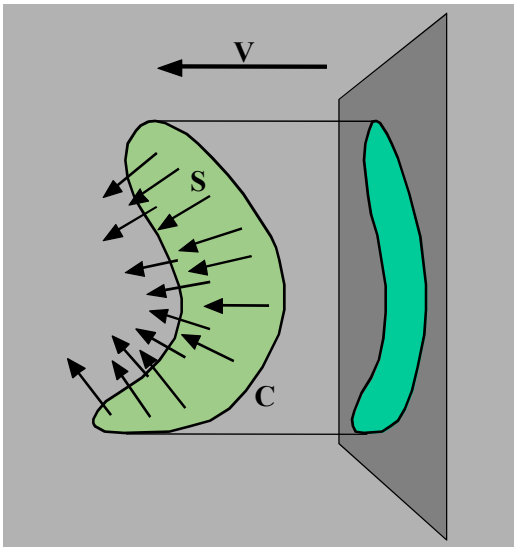
- There exists a vector **V** for which **N.V > 0** at (almost) every point of **S** (**N** being the normal vector of the surface at the considered point)

and

- The projection of **C** on a plane orthogonal to **V** along the direction of **V** has no self-intersections

then

- There are no self-collisions on the surface **S**.



Such criteria allow us to efficiently discard, from the detection process, “almost flat” surface regions that will not exhibit internal self-collisions. The union of two adjacent surface regions may also be “almost flat” so we need not detect collisions between these two regions. In particular, adjacent elements need not be checked for collisions. Implemented efficiently, this criterion will allow us to deal with the major cause of inefficiency of self-collision detection.

3.2.2. Modifying the Hierarchical Collision Detection Algorithm

As discussed, the general hierarchical collision detection algorithm works with bounding boxes. There are two main processes:

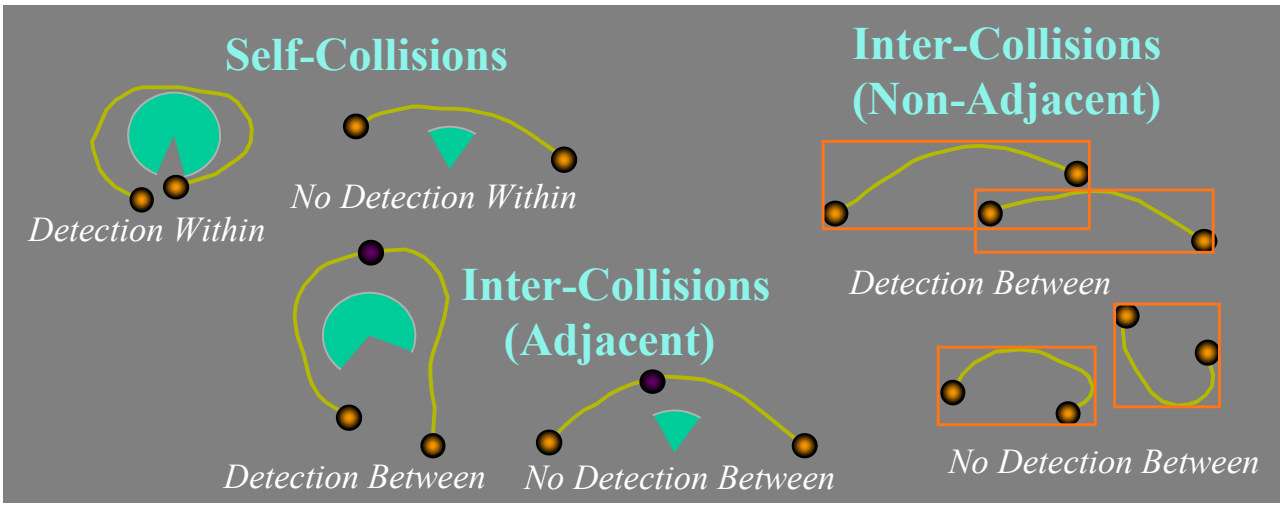
* *For detecting collisions within a surface region, collisions are detected within and between all the children of its corresponding hierarchy tree node.*
* *For detecting collisions between two surface regions, collisions are detected between the respective children nodes.*

Collisions between two nodes may be efficiently detected using a bounding-box evaluation: if the bounding boxes do not intersect, there are no intersections. However, in the standard hierarchical algorithm, there are no bounding box techniques for self-collision detection within one node. Replacing the bounding box test by a curvature test can overcome this limitation.

Curvature Optimization

For the self-collision stage, how to integrate curvature considerations is clear: Collisions should be detected within one node only if the corresponding surface does not match the curvature criteria.

When detecting collisions between two nodes, we should consider two cases: Dealing with two adjacent surfaces, collisions should be detected between the nodes only if the corresponding surface union does not verify the curvature criteria (obviously, the bounding boxes will always intersect, so the curvature criteria acts as a good replacement test). Dealing with non adjacent surfaces, the standard bounding box criteria should be used.



Using curvature or bounding volumes: The different cases.

Thus, the curvature criteria is incorporated into the hierarchical algorithm by replacing the bounding volume test by a curvature test within surface regions or between adjacent surface regions.

Given a curved surface, we need to determine the existence of, and find, a compatible direction vector that has positive dot product with all normal vectors of the surface. In our discrete model of the surface, that means that this vector should have positive dot product with the normals of all the surface polygons.

Similar to what was done for the bounding boxes, “direction boxes” that contain the allowable directions will be propagated up to the hierarchical parents.

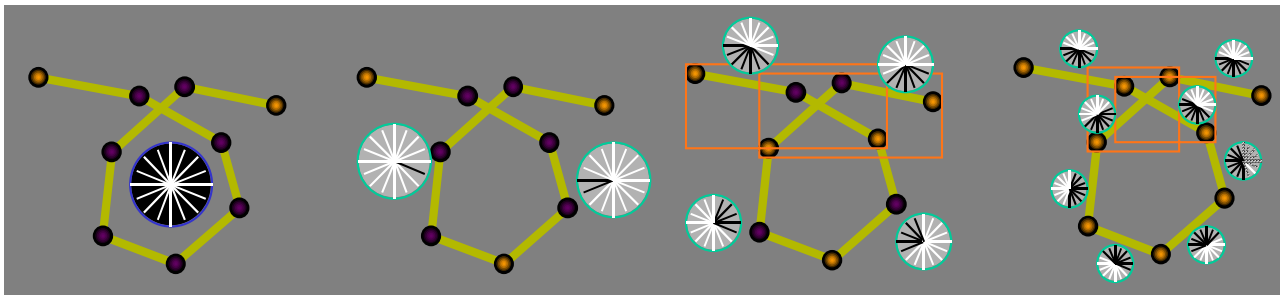
Assimilating the direction space to a sphere, the allowable direction box of a flat mesh polygon is a half sphere. When we consider two polygons, the globally allowable direction is the common part of the corresponding two sphere halves. This gives us the mechanism that should be propagated upwards in the tree hierarchy. The resulting direction box at the root of the tree may be empty, and in this case no suitable direction can be found for the whole surface. If not empty, any vector within the direction box is suitable.

Unlike volumes boxes that can be represented efficiently using space coordinates, there is no easy way to describe our direction boxes exactly. One solution is to use "direction cones" defined by a direction and an aperture angle. The bounding tightness is however very limited and combining two cones is not an efficient operation.

That difficulty can also be addressed by building a discrete set of direction vectors that will represent our direction space. The same set of sample directions can be used as the ones used for defining the DOPs used as bounding volumes: Using the 26 directions toward the face centers, vertices and edge centers of a cube, the angle accuracy is around 25°. The more directions used, the more accurate are the direction evaluations, at the expanse of computation time.

During the update process of the bounding volumes in the hierarchy tree, the direction boxes are updated as well. For each polygon of the mesh is computed the set of direction from the sample set that have positive positive dot product with the normal of the polygon. The result is stored in an array of boolean. Propagation up in the tree is done by combining these arrays with element-wise boolean AND operations. Then for any collision test, the low curvature criteria is met if TRUE elements remains in

the array, and the corresponding directions are the directions that meet the dot product requirements for all the corresponding surface region.



Combining curvature boxes and bounding volumes during the detection process.

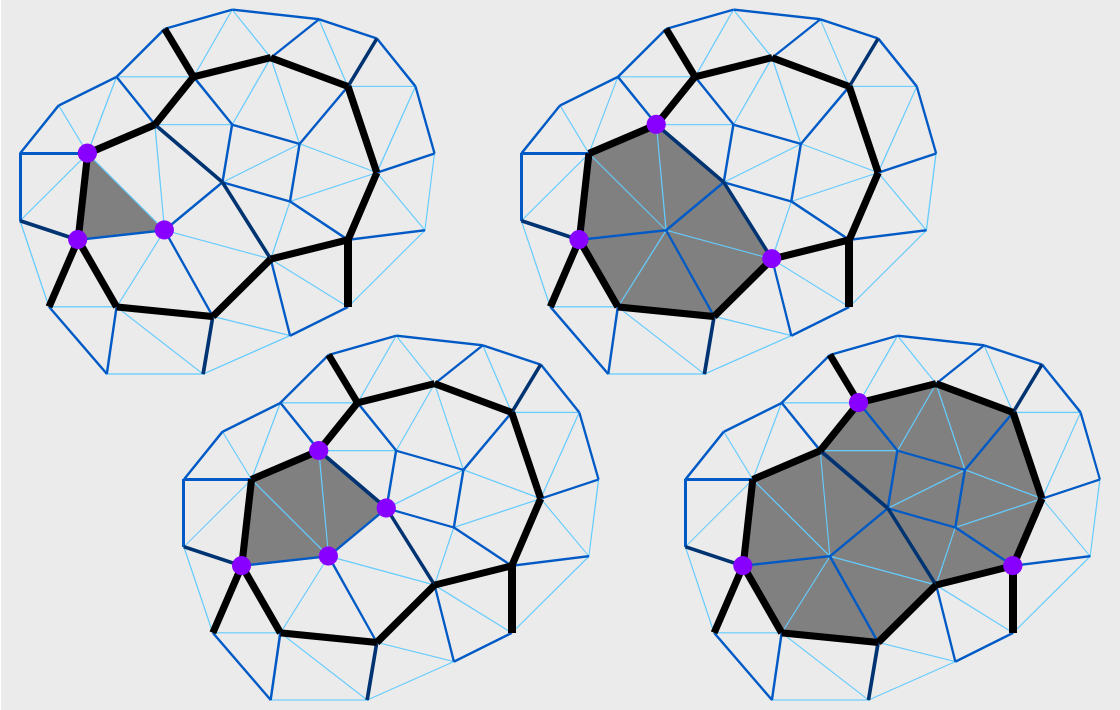
Adjacency Tests

Another major problem is the adjacency test: Given two arbitrary nodes in the hierarchy, are the corresponding surfaces adjacent? (Do they have at least one common vertex?)

This represents the major difficulty of our algorithm. This adjacency should be detected efficiently (the computation complexity should be O(log n)), and the extra storage in the tree data structure should be reasonable (the extra information for each node should be O(1)).

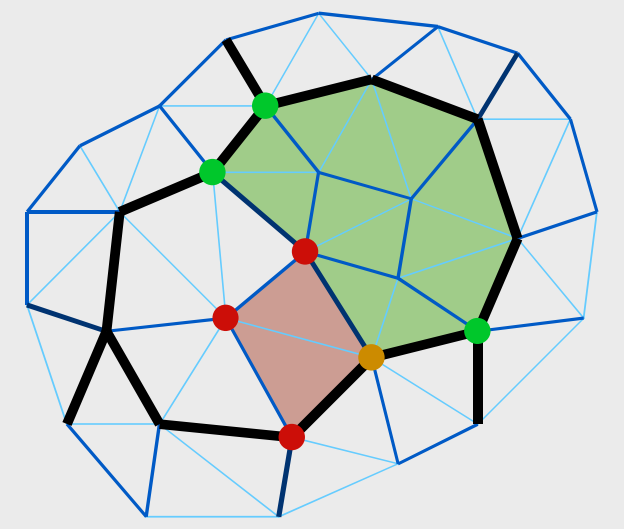
For each hierarchical node, a list of vertices should be defined, these being the vertices surrounding the corresponding surface region. Several circular lists should be considered when dealing with non-connected surfaces or surfaces having holes. Obviously, if two nodes are adjacent, they have at least one of these vertices in common.

Not all the boundary vertices should be stored, but only the vertices that separate two different surface regions among those of the highest hierarchy level that also do not include the region being considered. “Outside” is considered as a particular surface region. Whatever the node level and the total number of polygons surrounding this surface region, the number of stored vertices is approximately constant, and for usual meshes hardly exceeds six. As this adjacency information only depends on the mesh topology, this stage is usually performed once in the preprocessing stage.



Storing region boundary vertices in the hierarchy tree: The number remains roughly constant whatever the level in the hierarchy.

Adjacency testing is then performed easily: Two hierarchy nodes are adjacent if and only if they have at least one common vertex among those stored for these two nodes. This test is performed in O(1).



Two adjacent (point-connected) surface regions of the hierarchy have at least one common stored vertex.

3.2.3. Self-Collisions on Contours

The boundary shape of the surface may also be the cause of self-collisions. In most cases, this happens when an elongated strip is fold so as to produce a cone-shaped surface. Such collisions are also very

likely to happen around sharp concavities of the surface contour, where minimal folds can also produce self-collisions.

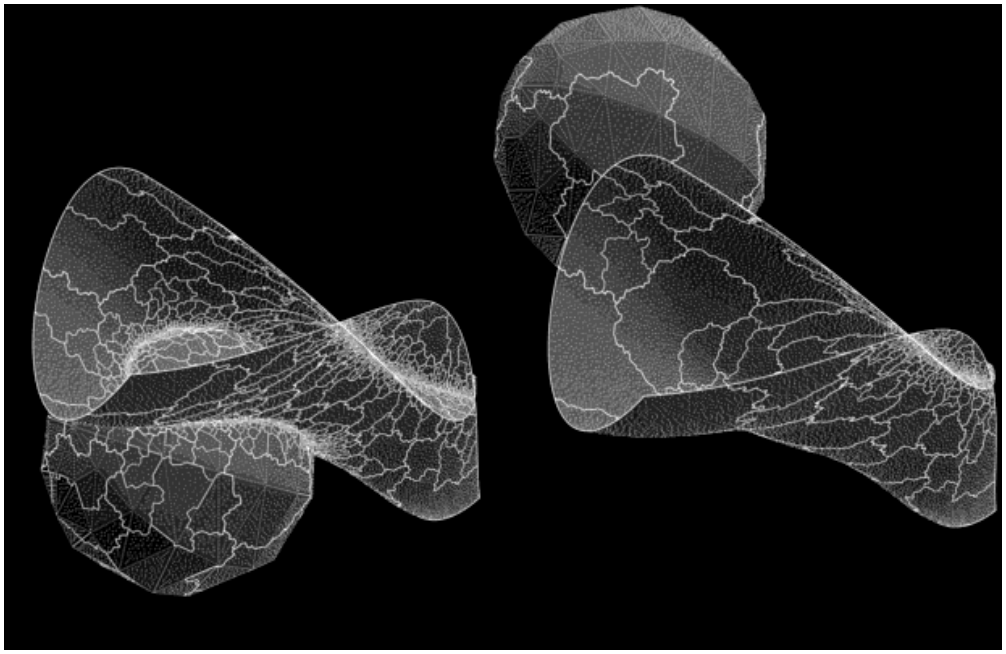
The most systematic solution would be to construct, for any surface region fulfilling the surface normal criteria, the 2D projection of the surface boundary on a plane orthogonal to one of the found directions. Despite flatness, collision detection should be carried out within or between children containing boundary intersections. This 2D collision detection process can also be based on a similar kind of curvature optimization. This would however require a significant amount of extra computation, along the data required for managing a contour-based hierarchies.

Practical test have however shown that these tests are rarely significant in most "real-world" situations involving for example garment simulation or deformation of soft objects. These tests may however improve detection of long objects with large curvature deformations (simulation of long ribbons) or surfaces with complex non-convex contours (cuts, holes).

These marginal situations may be addressed using various low-cost approximate techniques. The simplest method is to "expand" the direction boxes with a certain angle for mesh elements that are located on the boundary of the surface. This angle may also be increased for elements adjacent to non-convex boundary locations. The larger the angle, the most systematic the collision detection is, at the expanse of computation time.

3.3. Efficiency

The main interest of this algorithm is its efficiency in detecting self-collisions: Hierarchy regions that are not curved enough to contain self-collisions are efficiently omitted from the detection process. The following figure shows the regions considered when performing collision detection between two objects, as well as self-collision detection within these objects also. As a result, the algorithm efficiently focuses on the intersecting parts of the surfaces.



Collision detection is focused on the colliding parts of the surface.

The execution time required for performing collision detection is subdivided as follows:

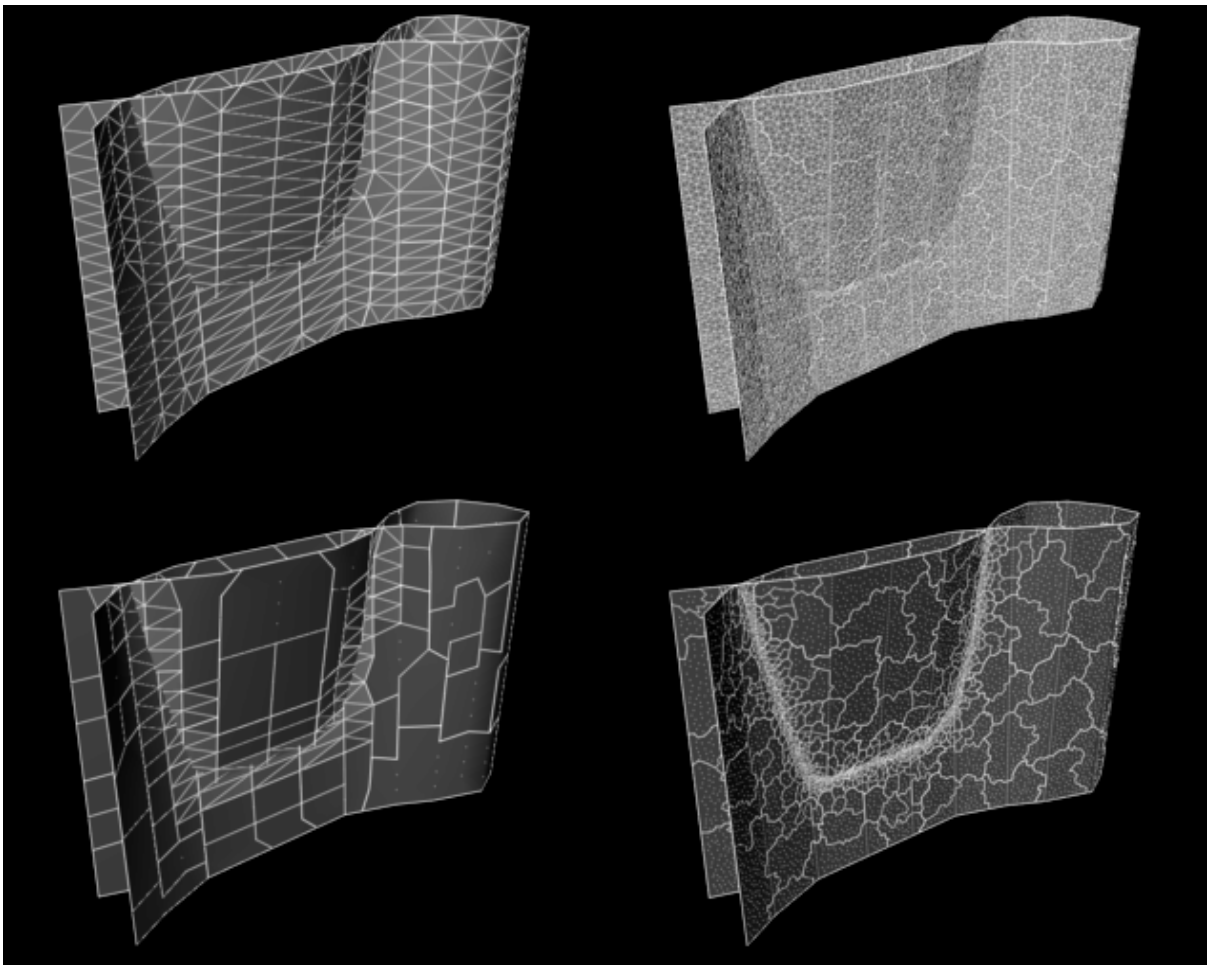
** Update of the bounding volumes of the hierarchy tree.*

** Update of the direction boxes of the hierarchy tree, if self-collision detection is required.*

** Running the collision detection algorithm on the hierarchy tree.*

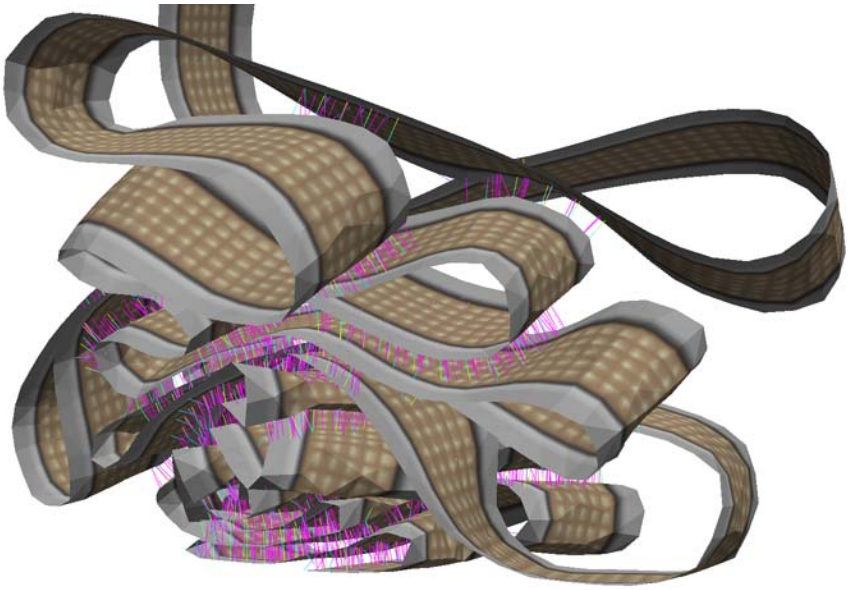
The time required for the two first steps is proportional to the number of mesh elements. The involved computations are quite reduced for mesh elements (evaluation of min-max of linear combination of vertex coordinates and normal computation of polygons usually also required for other purposes (mechanics, rendering...), and dot product with a set of directions). Their propagation along the hierarchy tree is also trivial (min-max interval merges, boolean operations). These linear-time computations only add a small overhead to the global simulation process.

The proposed scheme also behaves very well for highly discretized surfaces, as extra discretization usually yields flatter surfaces relatively to the size of the mesh elements. Hence, unless being near a collision area, the area of the surface regions considered during the detection process always have similar sizes whatever their discretization.



During the detection process, surface regions considered for the detection are similar in size and number whatever the discretization.

The major benefit of the curvature-based hierarchical collision is that full performance of hierarchical bounding-volume collision detection is preserved for self-collision detection, as the computation is not crippled by detecting all the "colliding" adjacent mesh elements of the surface.



Cloth simulation is heavily relying on self-collision detection.

Bibliography

R. BRIDSON, R. FEDKIW, J. ANDERSON, Robust Treatment of Collisions, Contact and Friction for Cloth Animation, ACM Transaction on Graphics, Proceedings of ACM SIGGRAPH, 21(3), pp 594-603, 2002.

J.D. COHEN, M.C. LIN, D. MANOCHA, M.K. PONAMGI, I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments, Symp. of Interactive 3D Graphics proc., pp 189-196, 1995.

K. FUJIMURA, H. TORIYA, K. YAMAGUSHI, T.L. KUNII, Octree Algorithms for Solid Modeling, Computer Graphics, Theory and Applications, Proceedings of InterGraphics'83, Springer-Verlag, pp 96-110, 1983.

F. GANOVELLI, J. DINGLIANA, C. O'SULLIVAN, Buckettree: Improving Collision Detection Between Deformable Objects, Proceedings of Spring Conference on Computer Graphics, 2000.

S. GOTTSCHALK, M.C. LIN, D. MANOCHA, OBB-Tree: A Hierarchical Structure for Rapid Interference Detection, Computer Graphics, Proceedings of ACM SIGGRAPH, Addison-Wesley, pp 171-180, 1996.

M. HELD, J.T. KLOSOWSKI, J.S.B. MITCHELL, Evaluation of Collision Detection Methods for Virtual Reality Fly-Throughs, Proceedings of the 7th Canadian Conference on Computational Geometry, 1995.

P.M. HUBBARD, Approximating Polyhedra with Spheres for Time-Critical Collision Detection, ACM Transactions on Graphics, 15(3) pp 179-210, 1996.

P. JIMENEZ, F. THOMAS, C. TORRAS, 3D Collision Detection: A Survey, Computer and Graphics, 25(2), pp 269-285, 2001.

J.T. KLOSOWSKI, M. HELD, J.S.B. MITCHELL, Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs, IEEE transactions on Visualization and Computer Graphics, 4(1), 1997.

T. LARSSON, T. AKENINE-MOLLER, Collision Detection for Continuously Deforming Bodies, Proceedings of Eurographics, pp 325-333, 2001.

M.C. LIN, J.F. CANNY, Efficient Collision Detection for Animation, Proceedings of the Eurographics Workshop on Animation and Simulation, 1992.

M.C. LIN, S. GOTTSCHALK, Collision Detection Between Geometric Models: A Survey, Proceedings of the IMA Conference on Mathematics of Surfaces, 1998.

J. LOMBARDO, M.P. CANI, F. NEYRET, Real-Time Collision Detection for Virtual Surgery, Proceedings of Computer Animation, IEEE Press, pp 82-91, 1999.

J. MEZGER, S. KIMMERLE, O. ETZMUSS, Hierarchical Techniques in Collision Detection for Cloth Animation, Journal of WSCG, 11(2), pp 322-329, 2003.

I.J. PALMER, R.L. GRIMSDALE, Collision Detection for Animation using Sphere-Trees, Computer Graphics Forum, 14, pp 105-116, 1995.

X. PROVOT, Collision and Self-Collision Handling in Cloth Models Dedicated to Design Garments, Proceedings of Graphics Interface, pp 177-189, 1997.

A. SMITH, Y. KITAMURA, H. TAKEMURA, F. KISHINO, A Simple and Efficient Method for Accurate Collision Detection among Deformable Polyhedra, Proceedings of IEEE Virtual Reality Annual International Symposium, pp 136-145, 1995.

G. VANDENBERGEN, Efficient Collision Detection of Complex Deformable Models using AABB Trees, Journal of Graphics Tools, 2(4), pp 1-14, 1997.

P. VOLINO, N. MAGNENAT-THALMANN, Efficient Self-Collision Detection on Smoothly Discretised Surface Animation Using Geometrical Shape Regularity, Computer Graphics Forum (Eurographics'94 proceedings), Blackwell Publishers, 13(3), pp 155-166, 1994.

R.C. WEBB, M.A. GIGANTE, Using Dynamic Bounding Volume Hierarchies to improve Efficiency of Rigid Body Simulations, Communicating with Virtual Worlds, Proceedings of CGI'92, pp 825-841, 1992.

G. ZACHMANN, Rapid Collision Detection by Dynamically Aligned DOP-Trees, Proceedings of IEEE Virtual Reality Annual International Symposium, pp 90-97, 1998.

OBBTree: A Hierarchical Structure for Rapid Interference Detection

S. Gottschalk M. C. Lin* D. Manocha
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
{gottscha,lin,manocha}@cs.unc.edu
http://www.cs.unc.edu/~geom/OBB/OBBT.html

Abstract: We present a data structure and an algorithm for efficient and exact interference detection amongst complex models undergoing rigid motion. The algorithm is applicable to all general polygonal models. It pre-computes a hierarchical representation of models using tight-fitting oriented bounding box trees (OBBTrees). At runtime, the algorithm traverses two such trees and tests for overlaps between oriented bounding boxes based on a separating axis theorem, which takes less than 200 operations in practice. It has been implemented and we compare its performance with other hierarchical data structures. In particular, it can robustly and accurately detect all the contacts between large complex geometries composed of hundreds of thousands of polygons at interactive rates.

CR Categories and Subject Descriptors: I.3.5 **[Computer Graphics]:** Computational Geometry and Object Modeling
Additional Key Words and Phrases: hierarchical data structure, collision detection, shape approximation, contacts, physically-based modeling, virtual prototyping.

1 Introduction

The problems of interference detection between two or more geometric models in static and dynamic environments are fundamental in computer graphics. They are also considered important in computational geometry, solid modeling, robotics, molecular modeling, manufacturing and computer-simulated environments. Generally speaking, we are interested in very efficient and, in many cases, real-time algorithms for applications with the following characterizations:

1. **Model Complexity:** The input models are composed of many hundreds of thousands of polygons.

2. **Unstructured Representation:** The input models are represented as collections of polygons with no topology information. Such models are also known as 'polygon soups' and their boundaries may have cracks, T-joints, or may have non-manifold geometry. No robust techniques are known for cleaning such models.

3. **Close Proximity:** In the actual applications, the models can come in close proximity of each other and can have multiple contacts.

4. **Accurate Contact Determination:** The applications need to know accurate contacts between the models (up to the resolution of the models and machine precision).

Many applications, like dynamic simulation, physically-based modeling, tolerance checking for virtual prototyping, and simulation-based design of large CAD models, have all these four characterizations. Currently, fast interference detection for such applications is a major bottleneck.

Main Contribution: We present efficient algorithms for accurate interference detection for such applications. They make no assumptions about model representation or the motion. The algorithms compute a hierarchical representation using *oriented bounding boxes (OBBs)*. An OBB is a rectangular bounding box at an arbitrary orientation in 3-space. The resulting hierarchical structure is referred to as an OBBTree. The idea of using OBBs is not new and many researchers have used them extensively to speed up ray tracing and interference detection computations. Our major contributions are:

1. New efficient algorithms for hierarchical representation of large models using tight-fitting OBBs.

2. Use of a 'separating axis' theorem to check two OBBs in space (with arbitrary orientation) for overlap. Based on this theorem, we can test two OBBs for overlap in about 100 operations on average. This test is about one order of magnitude faster compared to earlier algorithms for checking overlap between boxes.

3. Comparison with other hierarchical representations based on sphere trees and *axis-aligned bounding boxes (AABBs)*. We show that for many close proximity situations, OBBs are asymptotically much faster.

4. Robust and interactive implementation and demonstration. We have applied it to compute all contacts between very complex geometries at interactive rates.

The rest of the paper is organized in the following manner: We provide a comprehensive survey of interference detection methods in Section 2. A brief overview of the algorithm is given in Section 3. We describe algorithms for efficient computation of OBBTrees in Section 4. Section 5 presents the separating-axis theorem and shows how it can be used to compute overlaps between two OBBs very efficiently. We compare its performance with hierarchical representations composed of spheres and AABBs in Section 6. Section 7 discusses the implementation and performance of the algorithms on complex models. In Section 8, we discussion possible future extensions.

*Also with U.S. Army Research Office

2 Previous Work

Interference and collision detection problems have been extensively studied in the literature. The simplest algorithms for collision detection are based on using bounding volumes and spatial decomposition techniques in a hierarchical manner. Typical examples of bounding volumes include axis-aligned boxes (of which cubes are a special case) and spheres, and they are chosen for to the simplicity of finding collision between two such volumes. Hierarchical structures used for collision detection include cone trees, k-d trees and octrees [31], sphere trees [20, 28], R-trees and their variants [5], trees based on S-bounds [7] etc. Other spatial representations are based on BSP's [24] and its extensions to multi-space partitions [34], spatial representations based on space-time bounds or four-dimensional testing [1, 6, 8, 20] and many more. All of these hierarchical methods do very well in performing "rejection tests", whenever two objects are far apart. However, when the two objects are in close proximity and can have multiple contacts, these algorithms either use subdivision techniques or check very large number of bounding volume pairs for potential contacts. In such cases, their performance slows down considerably and they become a major bottleneck in the simulation, as stated in [17].

In computational geometry, many theoretically efficient algorithms have been proposed for polyhedral objects. Most of them are either restricted to static environments, convex objects, or only polyhedral objects undergoing rigid motion [9]. However, their practical utility is not clear as many of them have not been implemented in practice. Other approaches are based on linear programming and computing closest pairs for convex polytopes [3, 10, 14, 21, 23, 33] and based on line-stabbing and convex differences for general polyhedral models [18, 26, 29]. Algorithms utilizing spatial and temporal coherence have been shown to be effective for large environments represented as union of convex polytopes [10, 21]. However, these algorithms and systems are restrictive in terms of application to general polygonal models with unstructured representations. Algorithms based on interval arithmetic and bounds on functions have been described in [12, 13, 19]. They are able to find all the contacts accurately. However, their practical utility is not clear at the moment. They are currently restricted to objects whose motion can be expressed as a closed form function of time, which is rarely the case in most applications. Furthermore, their performance is too slow for interactive applications.

OBBs have been extensively used to speed up ray-tracing and other interference computations [2]. In terms of application to large models, two main issues arise: how can we compute a tight-fitting OBB enclosing a model and how quickly can we test two such boxes for overlap? For polygonal models, the minimal volume enclosing bounding box can be computed in $O(n^3)$ time, where n is the number of vertices [25]. However, it is practical for only small models. Simple incremental algorithms of linear time complexity are known for computing a minimal enclosing ellipsoid for a set of points [36]. The axes of the minimal ellipsoid can be used to compute a tight-fitting OBB. However, the constant factor in front of the linear term for this algorithm is very high (almost 3×10^5) and thereby making it almost impractical to use for large models. As for ray-tracing, algorithms using structure editors [30] and modeling hierarchies [35] have been used to construct hierarchies of OBBs. However, they cannot be directly applied to compute tight-fitting OBBs for large unstructured models.

A simple algorithm for finding the overlap status of two OBBs tests all edges of one box for intersection with any of the faces of the other box, and vice-versa. Since OBBs are convex polytopes, algorithms based on linear programming [27] and closest features computation [14, 21] can be used as well. A general purpose interference detection test between OBBs and convex polyhedron is presented in [16]. Overall, efficient algorithms were not known for computing hierarchies of tight-fitting OBBs for large unstructured models, nor were efficient algorithms known for rapidly checking the overlap status of two such OBBTrees.

3 Hierarchical Methods & Cost Equation

In this section, we present a framework for evaluating hierarchical data structures for interference detection and give a brief overview of OBBTrees. The basic cost function was taken from [35], who used it for analyzing hierarchical methods for ray tracing. Given two large models and their hierarchical representation, the total cost function for interference detection can be formulated as the following cost equation:

$$T = N_v \times C_v + N_p \times C_p\,, \tag{1}$$

where

T: total cost function for interference detection,
N_v: number of bounding volume pair overlap tests
C_v: cost of testing a pair of bounding volumes for overlap,
N_p: is the number primitive pairs tested for interference,
C_p: cost of testing a pair of primitives for interference.

Given this cost function, various hierarchical data structures are characterized by:

Choice of Bounding Volume: The choice is governed by two conflicting constraints:

1. It should fit the original model as tightly as possible (to lower N_v and N_p).
2. Testing two such volumes for overlap should be as fast as possible (to lower C_v).

Simple primitives like spheres and AABBs do very well with respect to the second constraint. But they cannot fit some primitives like long-thin oriented polygons tightly. On the other hand, minimal ellipsoids and OBBs provide tight fits, but checking for overlap between them is relatively expensive.

Hierarchical Decomposition: Given a large model, the tree of bounding volumes may be constructed bottom-up or top-down. Furthermore, different techniques are known for decomposing or partitioning a bounding volume into two or more sub-volumes. The leaf-nodes may correspond to different primitives. For general polyhedral models, they may be represented as collection of few triangles or convex polytopes. The decomposition also affects the values of N_v and N_p in (1).

It is clear that no hierarchical representation gives the best performance all the times. Furthermore, given two models, the total cost of interference detection varies considerably with relative placement of the models. In particular, when two models are far apart, hierarchical representations based on spheres and AABBs work well in practice. However, when two models are in close proximity with multiple number of closest features, the number of pair-wise bounding

volume tests, N_v increases, sometimes also leading to an increase in the number pair-wise primitive contact tests, N_p.

For a given model, N_v and N_p for OBBTreestend to be smaller as compared to those of trees using spheres or AABBs as bounding volumes. At the same time, the best known earlier algorithms for finding contact status of two OBBs were almost two orders of magnitude slower than checking two spheres or two AABBs for overlap. We present efficient algorithms for computing tight fitting OBBs given a set of polygons, for constructing a hierarchy of OBBs, and for testing two OBBs for contact. Our algorithms are able to compute tight-fitting hierarchies effectively and the overlap test between two OBBs is one order of magnitude faster than best known earlier methods. Given sufficiently large models, our interference detection algorithm based on OBBTrees much faster as compared to using sphere trees or AABBs.

4 Building an OBBTree

In this section we describe algorithms for building an OBBTree. The tree construction has two components: first is the placement of a tight fitting OBB around a collection of polygons, and second is the grouping of nested OBB's into a tree hierarchy.

We want to approximate the collection of polygons with an OBB of similar dimensions and orientation. We triangulate all polygons composed of more than three edges. The OBB computation algorithm makes use of first and second order statistics summarizing the vertex coordinates. They are the mean, μ, and the covariance matrix, $\mathbf{C}$, respectively [11]. If the vertices of the i'th triangle are the points $\mathbf{p}^i$, $\mathbf{q}^i$, and $\mathbf{r}^i$, then the mean and covariance matrix can be expressed in vector notation as:

$$\mu = \frac{1}{3n}\sum_{i=0}^{n}(\mathbf{p}^i + \mathbf{q}^i + \mathbf{r}^i),$$

$$\mathbf{C}_{jk} = \frac{1}{3n}\sum_{i=0}^{n}(\overline{\mathbf{p}}_j^i\overline{\mathbf{p}}_k^i + \overline{\mathbf{q}}_j^i\overline{\mathbf{q}}_k^i + \overline{\mathbf{r}}_j^i\overline{\mathbf{r}}_k^i), \qquad 1 \le j,k \le 3$$

where n is the number of triangles, $\overline{\mathbf{p}}^i = \mathbf{p}^i - \mu$, $\overline{\mathbf{q}}^i = \mathbf{q}^i - \mu$, and $\overline{\mathbf{r}}^i = \mathbf{r}^i - \mu$. Each of them is a 3×1 vector, e.g. $\overline{\mathbf{p}}^i = (\overline{\mathbf{p}}_1^i, \overline{\mathbf{p}}_2^i, \overline{\mathbf{p}}_3^i)^T$ and $\mathbf{C}_{jk}$ are the elements of the 3 by 3 covariance matrix.

The eigenvectors of a symmetric matrix, such as $\mathbf{C}$, are mutually orthogonal. After normalizing them, they are used as a basis. We find the extremal vertices along each axis of this basis, and size the bounding box, oriented with the basis vectors, to bound those extremal vertices. Two of the three eigenvectors of the covariance matrix are the axes of maximum and of minimum variance, so they will tend to align the box with the geometry of a tube or a flat surface patch.

The basic failing of the above approach is that vertices on the interior of the model, which ought not influence the selection of a bounding box placement, can have an arbitrary impact on the eigenvectors. For example, a small but very dense planar patch of vertices in the interior of the model can cause the bounding box to align with it.

We improve the algorithm by using the convex hull of the vertices of the triangles. The convex hull is the smallest convex set containing all the points and efficient algorithms of $O(n \lg n)$ complexity and their robust implementations

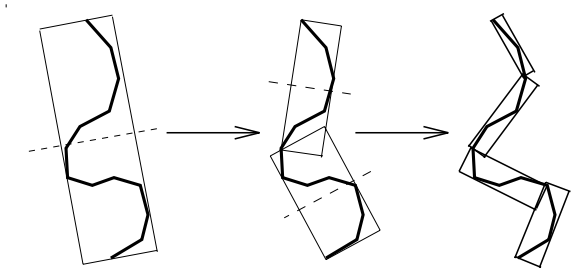


Figure 1: Building the OBBTree: recursively partition the bounded polygons and bound the resulting groups.

are available as public domain packages [4]. This is an improvement, but still suffers from a similar sampling problem: a small but very dense collection of nearly collinear vertices on the convex hull can cause the bounding box to align with that collection.

One solution is to sample the surface of the convex hull densely, taking the mean and covariance of the sample points. The uniform sampling of the convex hull surface normalizes for triangle size and distribution.

One can sample the convex hull "infinitely densely" by integrating over the surface of each triangle, and allowing each differential patch to contribute to the covariance matrix. The resulting integral has a closed form solution. Let the area of the i'th triangle in the convex hull be denoted by

$$A^i = \frac{1}{2}|(\mathbf{p}^i - \mathbf{q}^i) \times (\mathbf{p}^i - \mathbf{r}^i)|$$

Let the surface area of the entire convex hull be denoted by

$$A^H = \sum_i A^i$$

Let the centroid of the i'th convex hull triangle be denoted by

$$\mathbf{c}^i = (\mathbf{p}^i + \mathbf{q}^i + \mathbf{r}^i)/3$$

Let the centroid of the convex hull, which is a weighted average of the triangle centroids (the weights are the areas of the triangles), be denoted by

$$\mathbf{c}^H = \frac{\sum_i A^i\mathbf{c}^i}{\sum_i A^i} = \frac{\sum_i A^i\mathbf{c}^i}{A^H}$$

The elements of the covariance matrix $\mathbf{C}$ have the following closed-form,

$$\mathbf{C}_{jk} = \sum_{i=1}^{n}\frac{A^i}{12A^H}(9\mathbf{c}_j^i\mathbf{c}_k^i + \mathbf{p}_j^i\mathbf{p}_k^i + \mathbf{q}_j^i\mathbf{q}_k^i + \mathbf{r}_j^i\mathbf{r}_k^i) - \mathbf{c}_j^H\mathbf{c}_k^H$$

Given an algorithm to compute tight-fitting OBBs around a group of polygons, we need to represent them hierarchically. Most methods for building hierarchies fall into two categories: bottom-up and top-down. Bottom-up methods begin with a bounding volume for each polygon and merge volumes into larger volumes until the tree is complete. Top-down methods begin with a group of all polygons, and recursively subdivide until all leaf nodes are indivisible. In our current implementation, we have used a simple top-down approach.

Our subdivision rule is to split the longest axis of a box with a plane orthogonal to one of its axes, partitioning the polygons according to which side of the plane their center point lies on (a 2-D analog is shown in Figure 1). The subdivision coordinate along that axis was chosen to be that of the mean point, μ, of the vertices. If the longest axis cannot not be subdivided, the second longest axis is chosen. Otherwise, the shortest one is used. If the group of polygons cannot be partitioned along any axis by this criterion, then the group is considered indivisible.

If we choose the partition coordinate based on where the median center point lies, then we obtain balanced trees. This arguably results in optimal worst-case hierarchies for collision detection. It is, however, extremely difficult to evaluate average-case behavior, as performance of collision detection algorithms is sensitive to specific scenarios, and no single algorithm performs optimally in all cases.

Given a model with n triangles, the overall time to build the tree is $O(n \lg^2 n)$ if we use convex hulls, and $O(n \lg n)$ if we don't. The recursion is similar to that of quicksort. Fitting a box to a group of n triangles and partitioning them into two subgroups takes $O(n \lg n)$ with a convex hull and $O(n)$ without it. Applying the process recursively creates a tree with leaf nodes $O(\lg n)$ levels deep.

5 Fast Overlap Test for OBBs

Given OBBTrees of two objects, the interference algorithm typically spends most of its time testing pairs of OBBs for overlap. A simple algorithm for testing the overlap status for two OBB's performs 144 edge-face tests. In practice, it is an expensive test. Other algorithms based on linear programming and closest features computation exist. In this section, we present a new algorithm to test such boxes for overlap.

One trivial test for disjointness is to project the boxes onto some axis (not necessarily a coordinate axis) in space. This is an 'axial projection.' Under this projection, each box forms an interval on the axis. If the intervals don't overlap, then the axis is called a 'separating axis' for the boxes, and the boxes must then be disjoint. If the intervals do overlap, then the boxes may or may not be disjoint – further tests may be required.

How many such tests are sufficient to determine the contact status of two OBBs? We know that two disjoint convex polytopes in 3-space can always be separated by a plane which is parallel to a face of either polytope, or parallel to an edge from each polytope. A consequence of this is that two convex polytopes are disjoint iff there exists a separating axis orthogonal to a face of either polytope or orthogonal to an edge from each polytope. A proof of this basic theorem is given in [15]. Each box has 3 unique face orientations, and 3 unique edge directions. This leads to 15 potential separating axes to test (3 faces from one box, 3 faces from the other box, and 9 pairwise combinations of edges). If the polytopes are disjoint, then a separating axis exists, and one of the 15 axes mentioned above will be a separating axis. If the polytopes are overlapping, then clearly no separating axis exists. So, testing the 15 given axes is a sufficient test for determining overlap status of two OBBs.

To perform the test, our strategy is to project the centers of the boxes onto the axis, and also to compute the radii of the intervals. If the distance between the box centers as projected onto the axis is greater than the sum of the radii, then the intervals (and the boxes as well) are disjoint. This is shown in 2D in Fig. 2.

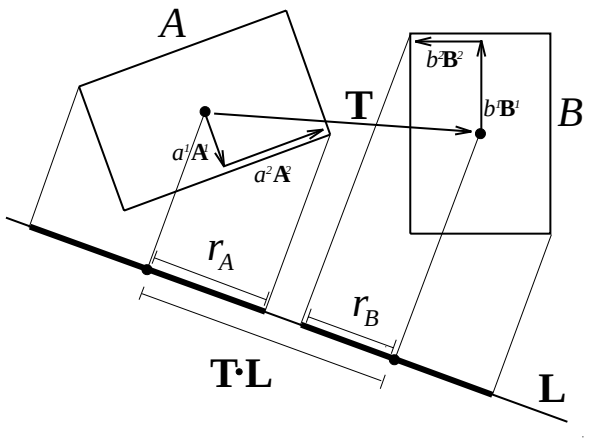


Figure 2: $\mathbf{L}$ is a separating axis for OBBs A and B because A and B become disjoint intervals under projection onto $\mathbf{L}$.

We assume we are given two OBBs, A and B, with B placed relative to A by rotation $\mathbf{R}$ and translation $\mathbf{T}$. The half-dimensions (or 'radii') of A and B are a_i and b_i, where $i = 1, 2, 3$. We will denote the axes of A and B as the unit vectors $\mathbf{A}^i$ and $\mathbf{B}^i$, for $i = 1, 2, 3$. These will be referred to as the 6 box axes. Note that if we use the box axes of A as a basis, then the three columns of $\mathbf{R}$ are the same as the three $\mathbf{B}^i$ vectors.

The centers of each box projects onto the midpoint of its interval. By projecting the box radii onto the axis, and summing the length of their images, we obtain the radius of the interval. If the axis is parallel to the unit vector $\mathbf{L}$, then the radius of box A's interval is

$$r_A = \sum_i |a_i \mathbf{A}^i \cdot \mathbf{L}|$$

A similar expression is used for r_B.

The placement of the axis is immaterial, so we assume it passes through the center of box A. The distance between the midpoints of the intervals is $|\mathbf{T} \cdot \mathbf{L}|$. So, the intervals are disjoint iff

$$|\mathbf{T} \cdot \mathbf{L}| > \sum_i |a_i \mathbf{A}^i \cdot \mathbf{L}| + \sum_i |b_i \mathbf{B}^i \cdot \mathbf{L}|$$

This simplifies when $\mathbf{L}$ is a box axis or cross product of box axes. For example, consider $\mathbf{L} = \mathbf{A}^1 \times \mathbf{B}^2$. The second term in the first summation is

$$\begin{aligned} |a_2 \mathbf{A}^2 \cdot (\mathbf{A}^1 \times \mathbf{B}^2)| &= |a_2 \mathbf{B}^2 \cdot (\mathbf{A}^2 \times \mathbf{A}^1)| \\ &= |a_2 \mathbf{B}^2 \cdot \mathbf{A}^3| \\ &= |a_2 \mathbf{B}^2_3| \\ &= a_2 |\mathbf{R}_{32}| \end{aligned}$$

The last step is due to the fact that the columns of the rotation matrix are also the axes of the frame of B. The original term consisted of a dot product and cross product, but reduced to a multiplication and an absolute value. Some terms reduce to zero and are eliminated. After simplifying all the terms, this axis test looks like:

$$|\mathbf{T}_3 \mathbf{R}_{22} - \mathbf{T}_2 \mathbf{R}_{32}| > a_2 |\mathbf{R}_{32}| + a_3 |\mathbf{R}_{22}| + b_1 |\mathbf{R}_{13}| + b_3 |\mathbf{R}_{11}|$$

All 15 axis tests simplify in similar fashion. Among all the tests, the absolute value of each element of $\mathbf{R}$ is used four times, so those expressions can be computed once before beginning the axis tests. The operation tally for all 15 axis tests are shown in Table 1. If any one of the expressions is satisfied, the boxes are known to be disjoint, and the remainder of the 15 axis tests are unnecessary. This permits early exit from the series of tests, so 200 operations is the absolute *worst case*, but often much fewer are needed.

Degenerate OBBs: When an OBB bounds only a single polygon, it will have zero thickness and become a rectangle. In cases where a box extent is known to be zero, the expressions for the tests can be further simplified. The operation counts for overlap tests are given in Table 1, including when one or both boxes degenerate into a rectangle. Further reductions are possible when a box degenerates to a line segment. Nine multiplies and ten additions are eliminated for every zero thickness.

OBBs with infinite extents: Also, when one or more extents are known to be infinite, as for a fat ray or plane, certain axis tests require a straight-forward modification. For the axis test given above, if a_2 is infinite, then the inequality cannot possibly be satisfied unless $\mathbf{R}_{32}$ is zero, in which case the test proceeds as normal but with the $a_2|\mathbf{R}_{32}|$ term removed. So the test becomes,

$$\begin{aligned} \mathbf{R}_{32} &= 0 \ \textbf{and} \\ |\mathbf{T}_3\mathbf{R}_{22} - \mathbf{T}_2\mathbf{R}_{32}| &> a_3|\mathbf{R}_{22}| + b_1|\mathbf{R}_{13}| + b_3|\mathbf{R}_{11}| \end{aligned}$$

In general, we can expect that $\mathbf{R}_{32}$ will not be zero, and using a short-circuit **and** will cause the more expensive inequality test to be skipped.

Operation	Box-Box	Box-Rect	Rect-Rect
compare	15	15	15
add/sub	60	50	40
mult	81	72	63
abs	24	24	24

Table 1: Operation Counts for Overlap Tests

Comparisons: We have implemented the algorithm and compared its performance with other box overlap algorithms. The latter include an efficient implementation of closest features computation between convex polytopes [14] and a fast implementation of linear programming based on Seidel's algorithm [33]. Note that the last two implementations have been optimized for general convex polytopes, but not for boxes. All these algorithms are much faster than performing 144 edge-face intersections. We report the average time for checking overlap between two OBBs in Table 2. All the timings are in microseconds, computed on a HP 735/125 .

Sep. Axis Algorithm	Closest Features	Linear Programming
$5-7$ us	$45-105$ us	$180-230$ us

Table 2: Performance of Box Overlap Algorithms

6 OBB's vs. other Volumes

The primary motivation for using OBBs is that, by virtue of their variable orientation, they can bound geometry more tightly than AABBTrees and sphere trees. Therefore, we reason that, all else being the same, fewer levels of an OBBTree need to be be traversed to process a collision query for objects in close proximity. In this section we present an analysis of asymptotic performance of OBBTrees versus AABBTrees and sphere trees, and an experiment which supports our analysis.

In Fig. 9(at the end), we show the different levels of hierarchies for AABBTrees and OBBTrees while approximating a torus. The number of bounding volumes in each tree at each level is the same. The ϵ for OBBTrees is much smaller as compared to ϵ for the AABBTrees.

First, we define *tightness*, *diameter*, and *aspect ratio* of a bounding volume with respect to the geometry it covers. The *tightness*, τ, of a bounding volume, B, with respect to the geometry it covers, G, is B's Hausdorff distance from G. Formally, thinking of B and G as closed point sets, this is

$$\tau = \max_{b \in B} \min_{g \in G} \mathsf{dist}(b, g)$$

The diameter, d, of a bounding volume with respect to the bounded geometry is the maximum distance among all pairs of enclosed points on the bounded geometry,

$$d = \max_{g,h \in G} \mathsf{dist}(g, h)$$

The *aspect ratio*, ρ, of a bounding volume with respect to bounded geometry is $\rho = \tau/d$.

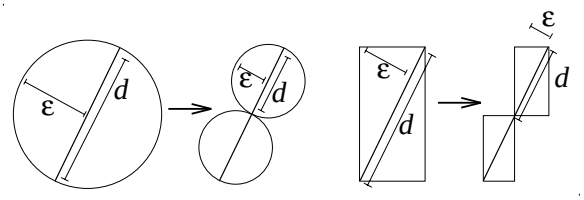


Figure 3: Aspect ratios of parent volumes are similar to those of children when bounding nearly flat geometry.

We argue that when bounded surfaces have low curvature, AABBTrees and sphere trees form fixed aspect ratio hierarchies, in the sense that the aspect ratio of a node in the hierarchy will have an aspect ratio similar to its children. This is illustrated in Fig. 3 for plane curves. If the bounded geometry is nearly flat, then the children will have shapes similar to the parents, but smaller. In Fig 3 for both spheres and AABBs, d and τ are halved as we go from parents to children, so $\rho = d/\tau$ is approximately the same for both parent and child. For fixed aspect ratio hierarchies, τ has linear dependence on d.

Note that the aspect ratio for AABBs is very dependent on the specific orientation of the bounded geometry – if the geometry is conveniently aligned, the aspect ratio can be close to 0, whereas if it is inconveniently aligned, ρ can be close to 1. But whatever the value, an AABB enclosing nearly flat geometry will have approximately the same ρ as its children.

Since an OBB aligns itself with the geometry, the aspect ratio of an OBB does not depend on the geometry's orientation in model space. Rather, it depends more on the local curvature of the geometry. For the sake of analysis, we are assuming nearly flat geometry. Suppose the bounded geometry has low constant curvature, as on the surface of a large sphere. In Fig. 4 we show a plane curve of fixed radius of curvature r and bounded by an OBB. We have $d = 2r\sin\theta$, and $\tau = r - r\cos\theta$. Using the small angle

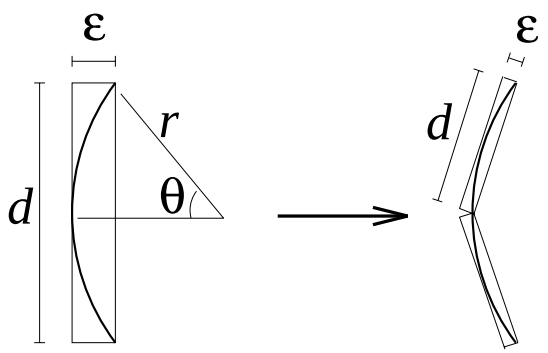


Figure 4: OBBs: Aspect ratio of children are half that of parent when bounding surfaces of low constant curvature when bounding nearly flat geometry.

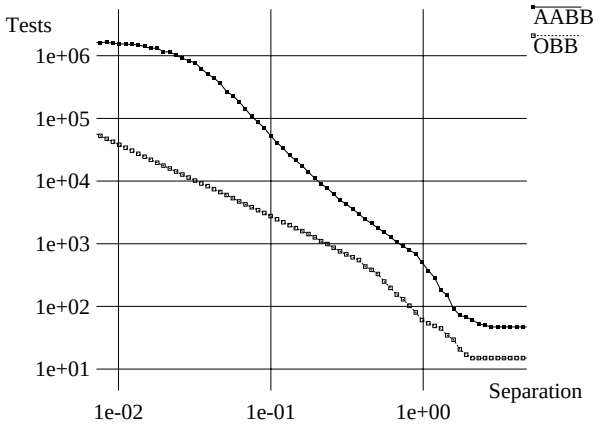


Figure 5: AABBs (upper curve) and OBBs (lower curve) for parallel close proximity (log-log plot)

approximation and eliminating θ, we obtain $\tau = d^2/8r$. So τ has quadratic dependence on d. When d is halved, τ is quartered, and the aspect ratio is halved.

We conclude that when bounding low curvature surfaces, AABBTrees and spheres trees have τ with linear dependence on d, whereas OBBTrees have τ with quadratic dependence on d. We have illustrated this for plane curves in the figures, but the relationships hold for surfaces in three space as well.

Suppose we use N same-sized bounding volumes to cover a surface patch with area A and require each volume to cover $O(A/N)$ surface area (for simplicity we are ignoring packing inefficiencies). Therefore, for these volumes, $d = O(\sqrt{A/N})$. For AABBs and spheres, τ depends linearly on d, so $\tau = O(\sqrt{A/N})$. For OBBs, quadratic dependence on d gives us OBBs, $\tau = O(A/N)$. So, to cover a surface patch with volumes to a given tightness, if OBBs require $O(m)$ bounding volumes, AABBs and spheres would require $O(m^2)$ bounding volumes.

Most contact scenarios do not require traversing both trees to all nodes of a given depth, but this does happen when two surfaces come into *parallel close proximity* to one another, in which every point on each surface is close to some point on the other surface. This is most common in virtual prototyping and tolerance analysis applications, in which fitted machine parts are tested for mechanical consistency. Also, dynamic simulations often generate paths in which one object comes to rest against another. It should be also be noted that when two smooth, highly tessellated surfaces come into near contact with each other, the region of near contact locally resembles a parallel close proximity scenario in miniature, and, for sufficiently tessellated models, the expense of processing that region can dominate the overall collision query. So, while it may seem like a very special case, parallel close proximity is an abstract situation which deserves consideration when designing collision and evaluating collision detection algorithms.

Experiments: We performed two experiments to support our analysis. For the first, we generated two concentric spheres consisting of $32,000$ triangles each. The smaller sphere had radius 1, while the larger had radius $1+\epsilon$. We performed collision queries with both OBBTrees and AABBTrees. The AABBTrees were created using the same process as for OBBTrees, except that instead of using the eigenvectors of the covariance matrix to determine the box orientations, we used the identity matrix.

The number of bounding box overlap tests required to process the collision query are shown in Fig. 5 for both tree types, and for a range of ϵ values. The graph is a log-log plot. The upper curve is for AABBTrees, and the lower, OBBTrees. The slopes of the the linear portions the upper curve and lower curves are approximately -2 and -1, as expected from the analysis. The differing slopes of these curves imply that OBBTrees require asymptotically fewer box tests as a function of ϵ than AABBTrees in our experiment.

Notice that the curve for AABBTrees levels off for the lowest values of ϵ. For sufficiently small values of ϵ, even the lowest levels of the AABBTree hierarchies are inadequate for separating the two surfaces – all nodes of both are visited, and the collision query must resort to testing the triangles. Decreasing ϵ even further cannot result in more work, because the tree does not extend further than the depth previously reached. The curve for the OBBTrees will also level off for some sufficiently small value of ϵ, which is not shown in the graph. Furthermore, since both trees are binary and therefore have the same number of nodes, the OBBTree curve will level off at the same height in the graph as the AABBTree curve.

For the second experiment, two same-size spheres were placed next to each other, separated by a distance of ϵ. We call this scenario *point close proximity*, where two nonparallel surfaces patches come close to touching at a point. We can think of the surfaces in the neighborhood of the closest points as being in parallel close proximity – but this approximation applies only locally. We have not been able to analytically characterize the performance, so we rely instead on empirical evidence to claim that for this scenario OBBTrees require asymptotically fewer bounding box overlap tests as a function of ϵ than AABBTrees. The results are shown in Fig. 6. This is also a log-log plot, and the increasing gap between the upper and lower curves show the asymptotic difference in the number of tests as ϵ decreases. Again, we see the leveling off for small values of ϵ.

Analysis: A general analysis of the performance of collision detection algorithms which use bounding volume hierarchies is extremely difficult because performance is so situation specific. We assumed that the geometry being bounded had locally low curvature and was finely tessellated. This enabled the formulation of simple relationships

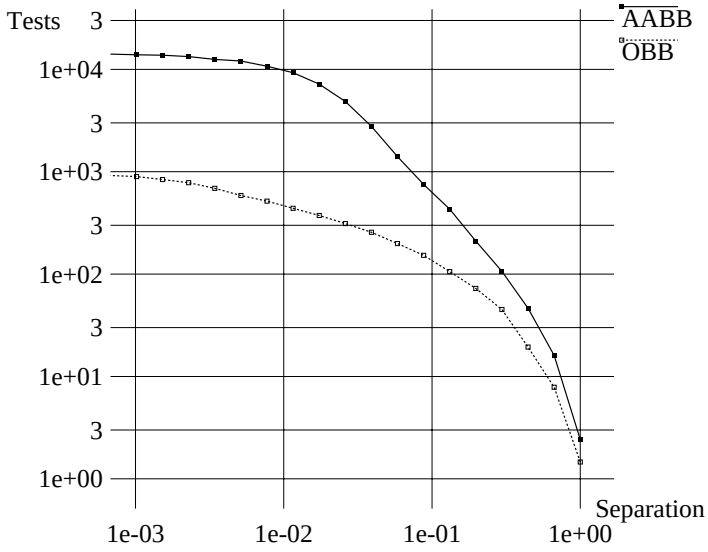


Figure 6: AABBs (upper curve) and OBBs (lower curve) for point close proximity. (log-log plot)

between τ and d. We also assumed that the packing efficiency of bounding volumes was perfect so as to formulate the relationships between d and the area of the surface covered. We believe that the inaccuracies of these assumptions account for the deviations from theory exhibited in the graph of Fig. 5.

For surface patches with high curvature everywhere, such as a 3D fractal, we may not expect to see asymptotic performance advantages for OBBs. Similarly, a coarse tessellation of a surface will place a natural limit on the number, N, the number of volumes used to approximate the surface. For a coarse tessellation, OBB-, sphere-, and AABBTrees may have to traverse their entire hierarchies for sufficiently close proximity scenarios, thus requiring approximately the same number of bounding volume overlap tests. Furthermore, for scenarios in which parallel close proximity does not occur, we don't expect the quadratic convergence property of OBBs to be of use, and again don't expect to see superior asymptotic performance.

7 Implementation and Performance

The software for the collision detection library was written in C++. The primary data structure for an OBB is a "box" class whose members contain a rotation matrix and translation vector, defining its placement relative to its parent, pointers to its parent and two children, the three box dimensions, and an object which holds a list of the triangles the box contains. The overall data structure for the box occupies 168 bytes. The tree formed from boxes as nodes, and the triangle list class, are the only compound data structures used.

An OBBTree of n triangles contains n leaf boxes and $n-1$ internal node boxes. In terms of memory requirements, there are approximately two boxes per triangle. The triangle itself requires 9 double precision numbers plus an integer for identification, totaling 76 bytes (based on 64-bit IEEE arithmetic). The memory requirement therefore totals 412 bytes per triangle in the model. This estimate does not include whatever overhead may exist in the dynamic memory allocation mechanism of the runtime environment. Using quaternions instead of rotation matrices (to represent box orientations), results in substantial space savings, but need 13 more operations per OBB overlap test. Single precision arithmetic can also be used to save memory.

7.1 Robustness and Accuracy

The algorithm and the implementations are applicable to all unstructured polygonal models. The polygons are permitted to be degenerate, with two or even one unique vertex, have no connectivity restrictions. The algorithm requires no adjacency information. This degree of robustness gives the system wider applicability. For example, space curves can be approximated by degenerate triangles as line segments – the system will correctly find intersections of those curves with other curves or surfaces.

The OBB overlap test is very robust as compared to other OBB overlap algorithms. It does not need to check for non-generic conditions such as parallel faces or edges; these are not special cases for the test and do not need to be handled separately. As a series of comparisons between linear combinations, the test is numerically stable: there are no divisions, square roots, or other functions to threaten domain errors or create conditioning problems. The use of an error margin, ϵ, guards against missing intersections due to arithmetic error. Its value can be set by the user.

Since the flow of control for the overlap test is simple and the number of operations required is small, the overlap test is a good candidate for microcoding or implemented in assembly. The test could also be easily implemented in hardware. Since most of the collision query time is spent in the overlap tests, any such optimization will significantly improve overall running time.

The Qhull package [4] is optionally used for computing the OBB orientation. It has been found to be quite robust. If we do use Qhull, we have to ensure that the input to Qhull spans 3 dimensions. If the input is rank deficient, our current implementation skips the use of Qhull, and uses all the triangles in the group. A more complete solution would be to project the input onto a lower dimensional space, and compute the convex hull of the projection (Qhull works on input of arbitrary specified dimension, but the input must be full rank).

There is the issue of propagation of errors as we descend the hierarchies, performing overlap tests. When we test two boxes or two triangles, their placement relative to one another is the result of a series of transformations, one for each level of each hierarchy we have traversed. We have not found errors due to the cascading of transformation matrices, but it is a theoretical source of errors we are aware of.

7.2 Performance

Our interference detection algorithm has been applied to two complex synthetic environments to demonstrate its efficiency (as highlighted in Table 3). These figures are for an SGI Reality Engine (90 MHz R8000 CPU, 512 MB).

A simple dynamics engine exercised the collision detection system. At each time step, the contact polygons were found by the collision detection algorithm, an impulse was applied to the object at each contact before advancing the clock.

In the first scenario, the pipes model was used as both the environment and the dynamic object, as shown in Fig. 8. Both object and environment contain 140,000 polygons.

Scenario	Pipes	Torus
Environ Size	143690 pgns	98000 pgns
Object Size	143690 pgns	20000 pgns
Num of Steps	4008	1298
Num of Contacts	23905	2266
Num of Box-Box Tests	1704187	1055559
Num of Tri-Tri Tests	71589	7069
Time	16.9 secs	8.9 secs
Ave. Int. Detec. Time	**4.2 msecs**	**6.9 msecs**
Ave. Time per Box Test	7.9 usecs	7.3 usecs
Ave. Contacts per Step	6.0	1.7

Table 3: *Timings for simulations*

The object is 15 times smaller in size than the environment. We simulated a gravitational field directed toward the center of the large cube of pipes, and permitted the smaller cube to fall inward, tumbling and bouncing. Its path contained 4008 discrete positions, and required 16.9 seconds to determine all 23905 contacts along the path. This is a challenging scenario because the smaller object is entirely embedded within the larger model. The models contain long thin triangles in the straight segments of the pipes, which cannot be efficiently approximated by sphere trees, octrees, and AABBTrees, in general. It has no obvious groups or clusters, which are typically used by spatial partitioning algorithms like BSP's.

The other scenario has a complex wrinkled torus encircling a stalagmite in a dimpled, toothed landscape. Different steps from this simulation are shown in Fig. 10. The spikes in the landscape prevent large bounding boxes from touching the floor of the landscape, while the dimples provide numerous shallow concavities into which an object can enter. Likewise, the wrinkles and the twisting of the torus makes it impractical to decompose into convex polytopes, and difficult to efficiently apply bounding volumes. The wrinkled torus and the environment are also smooth enough to come into parallel close proximity, increasing the number of bounding volume overlap tests. Notice that the average number of box tests per step for the torus scenario is almost twice that of the pipes, even though the number of contacts is much lower.

We have also applied our algorithm to detect collision between a moving torpedo on a pivot model (as shown in Fig. 7). These are parts of a torpedo storage and handling room of a submarine. The torpedo model is 4780 triangles. The pivot structure has 44921 triangles. There are multiple contacts along the length of the torpedo as it rests among the rollers. A typical collision query time for the scenario shown in Fig. 7 is 100 ms on a 200MHz R4400 CPU, 2GB SGI Reality Engine.

7.3 Comparison with Other Approaches

A number of hierarchical structures are known in the literature for interference detection. Most of them are based on spheres or AABBs. They have been applied to a number of complex environments. However, there are *no standard benchmarks* available to compare different algorithms and implementations. As a result, it is non-trivial to compare two algorithms and their implementations. More recently, [18] have compared different algorithms (based on line-stabbing and AABBs) on models composed of tens of thousands of polygons. On an SGI $Indigo^2$ Extreme, the algorithms with the best performance are able to compute all the contacts between the models in about $1/7 - 1/5$ of a second. Just based on the model complexity, we are able to handle models composed of hundreds of thousands of polygons (with multiple parallel contacts) in about $1/25 - 1/75$ of a second. We also compared our algorithm with an implementation of sphere tree based on the algorithm presented in [28]. A very preliminary comparison indicates one order of magnitude improvement. More comparisons and experiments are planned in the near future.

7.4 RAPID and benchmarks

Our implementation of our algorithms is available as a software package called RAPID (Rapid and Accurate Polygon Interference Detection). It can be obtained from: **http://www.cs.unc.edu/~geom/OBB/OBBT.html**.

Most of the models shown in this paper are also available, as well as precomputed motion sequences.

Overall, we find that given two large models in close proximity, with C_v, N_v, and N_p from the cost equation (1):

- C_v for OBBTrees is one-order of magnitude slower than that for sphere trees or AABBs.
- N_v for OBBTrees is asymptotically lower than that for sphere trees or AABBs. Likewise, N_p for OBBTrees is asymptotically lower.

Thus, given sufficiently large models in sufficiently close proximity, using OBBTrees require less work to process a collision query than using AABBTrees or sphere trees.

8 Extensions and Future Work

In the previous sections, we described the algorithm for interference detection between two polygonal models undergoing rigid motion. Some of the future work includes its specialization and extension to other applications. These include ray-tracing, interference detection between curved surfaces, view frustum culling and deformable models. As far as curve and surface intersections are concerned, current approaches are based on algebraic methods, subdivision methods and interval arithmetic [32]. Algebraic methods are restricted to low degree intersections. For high degree curve intersections, algorithms based on interval arithmetic have been found to be the fastest [32]. Such algorithms compute a decomposition of the curve in terms of AABBs. It will be worthwhile to try OBBs. This would involve subdividing the curve, computing tight-fitting OBBs for each segment, and checking them for overlaps.

In terms of view frustum culling, most applications use hierarchies based on AABBs. Rather, we may enclose the object using an OBBTree and test for overlap with the view frustum. The overlap test presented in Section 5 can be easily extended to test for overlap between an OBB and a view frustum.

Libraries and Benchmarks: There is great need to develop a set of libraries and benchmarks to compare different algorithms. This would involve different models as well as scenarios.

9 Conclusion

In this paper, we have presented a hierarchical data structure for rapid and exact interference detection between polygonal models. The algorithm is general-purpose and makes no assumptions about the input model. We have presented new algorithms for efficient construction of tight-fitting OBBTrees and overlap detection between two OBBs based on a

new separating axis theorem. We have compared its performance with other hierarchies of spheres and AABBs and find it asymptotically faster for close proximity situations. The algorithm has been implemented and is able to detect all contacts between complex geometries (composed of a few hundred thousand polygons) at interactive rates.

10 Acknowledgements

Thanks to Greg Angelini, Jim Boudreaux, and Ken Fast at Electric Boat for the model of torpedo storage and handling room. This work was supported in part by a Sloan foundation fellowship, ARO Contract P-34982-MA, NSF grant CCR-9319957, NSF grant CCR-9625217, ONR contract N00014-94-1-0738, ARPA contract DABT63-93-C-0048, NSF/ARPA Science and Technology Center for Computer Graphics & Scientific Visualization NSF Prime contract No. 8920219 and a grant from Ford Motor company.

References

[1] A.Garica-Alonso, N.Serrano, and J.Flaquer. Solving the collision detection problem. *IEEE Computer Graphics and Applications*, 13(3):36–43, 1994.

[2] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. In *An Introduction to Ray Tracing*, pages 201–262, 1989.

[3] D. Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. *ACM Computer Graphics*, 24(4):19–28, 1990.

[4] B. Barber, D. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hull. Technical Report GCG53, The Geometry Center, MN, 1993.

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

[6] S. Cameron. Collision detection by four-dimensional intersection testing. *Proceedings of International Conference on Robotics and Automation*, pages 291–302, 1990.

[7] S. Cameron. Approximation hierarchies and s-bounds. In *Proceedings. Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 129–137, Austin, TX, 1991.

[8] J. F. Canny. Collision detection for moving polyhedra. *IEEE Trans. PAMI*, 8:200–209, 1986.

[9] B. Chazelle and D. P. Dobkin. Intersection of convex objects in two and three dimensions. *J. ACM*, 34:1–27, 1987.

[10] 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.

[11] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, 1973.

[12] Tom Duff. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. *ACM Computer Graphics*, 26(2):131–139, 1992.

[13] J. Snyder et. al. Interval methods for multi-point collisions between time dependent curved surfaces. In *Proceedings of ACM Siggraph*, pages 321–334, 1993.

[14] 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.

[15] S. Gottschalk. Separating axis theorem. Technical Report TR96-024, Department of Computer Science, UNC Chapel Hill, 1996.

[16] N. Greene. Detecting intersection of a rectangular solid and a convex polyhedron. In *Graphics Gems IV*, pages 74–82. Academic Press, 1994.

[17] J. K. Hahn. Realistic animation of rigid bodies. *Computer Graphics*, 22(4):pp. 299–308, 1988.

[18] M. Held, J.T. Klosowski, and J.S.B. Mitchell. Evaluation of collision detection methods for virtual reality fly-throughs. In *Canadian Conference on Computational Geometry*, 1995.

[19] B. V. Herzen, A. H. Barr, and H. R. Zatz. Geometric collisions for time-dependent parametric surfaces. *Computer Graphics*, 24(4):39–48, 1990.

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

[21] 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.

[22] M.C. Lin and Dinesh Manocha. Fast interference detection between geometric models. *The Visual Computer*, 11(10):542–561, 1995.

[23] M. Moore and J. Wilhelms. Collision detection and response for computer animation. *Computer Graphics*, 22(4):289–298, 1988.

[24] B. Naylor, J. Amanatides, and W. Thibault. Merging bsp trees yield polyhedral modeling results. In *Proc. of ACM Siggraph*, pages 115–124, 1990.

[25] J. O'Rourke. Finding minimal enclosing boxes. *Internat. J. Comput. Inform. Sci.*, 14:183–199, 1985.

[26] M. Ponamgi, D. Manocha, and M. Lin. Incremental algorithms for collision detection between general solid models. In *Proc. of ACM/Siggraph Symposium on Solid Modeling*, pages 293–304, 1995.

[27] F.P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.

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

[29] A. Rappoport. The extended convex differences tree (ecdt) representation for n-dimensional polyhedra. *International Journal of Computational Geometry and Applications*, 1(3):227–41, 1991.

[30] S. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *Proc. of ACM Siggraph*, pages 110–116, 1980.

[31] H. Samet. *Spatial Data Structures: Quadtree, Octrees and Other Hierarchical Methods*. Addison Wesley, 1989.

[32] T.W. Sederberg and S.R. Parry. Comparison of three curve intersection algorithms. *Computer-Aided Design*, 18(1):58–63, 1986.

[33] R. Seidel. Linear programming and convex hulls made easy. In *Proc. 6th Ann. ACM Conf. on Computational Geometry*, pages 211–215, Berkeley, California, 1990.

[34] W.Bouma and G.Vanecek. Collision detection and analysis in a physically based simulation. *Proceedings Eurographics workshop on animation and simulation*, pages 191–203, 1991.

[35] H. Weghorst, G. Hooper, and D. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, pages 52–69, 1984.

[36] E. Welzl. Smallest enclosing disks (balls and ellipsoids). Technical Report B 91-09, Fachbereich Mathematik, Freie Universitat, Berlin, 1991.

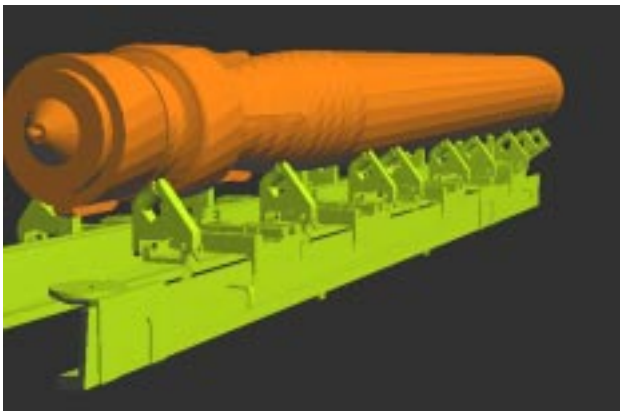


Figure 7: Interactive Interference Detection for a Torpedo (shown in yellow) on a Pivot Structure (shown in green) – Torpedo has 4780 triangles; Pivot has 44921 triangles; Average time to perform collision query: 100 msec on SGI Reality Engine with 200MHz R4400 CPU

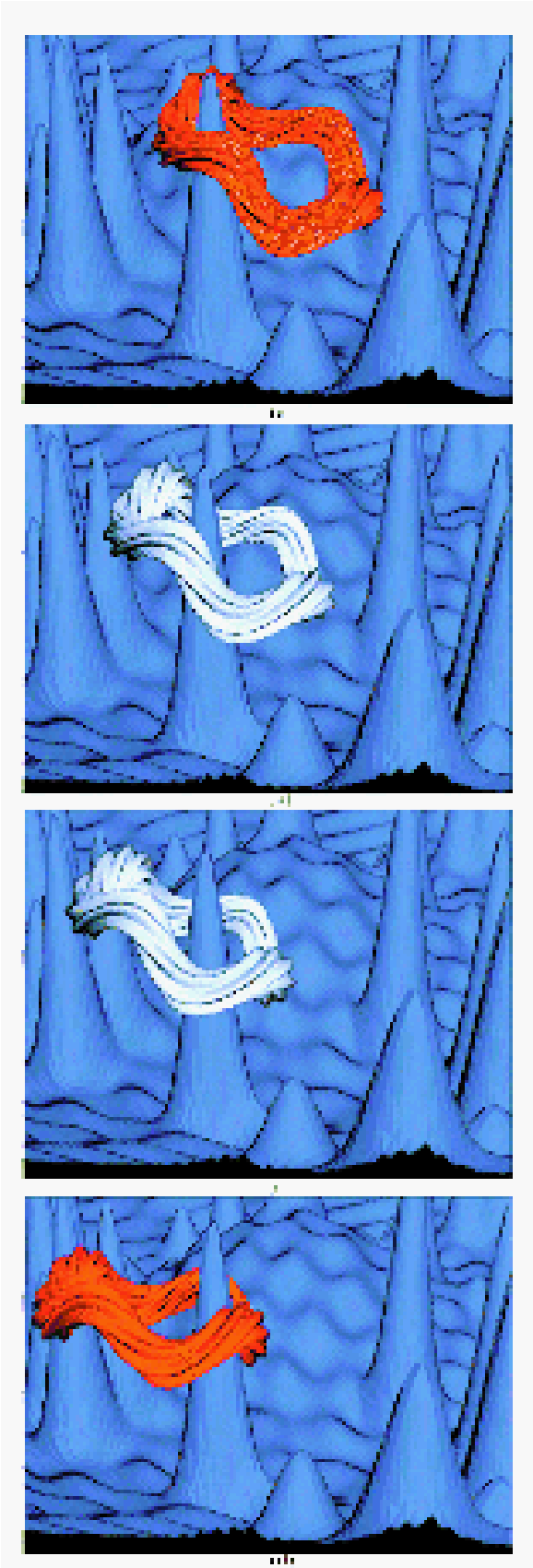


Figure 10: Interactive Interference Detection for a Complex Torus – Torus has 20000 polygons; Environment has 98000 polygons; Average time to perform collision query: 6.9 msec on SGI Reality Engine with 90MHz R8000 CPU

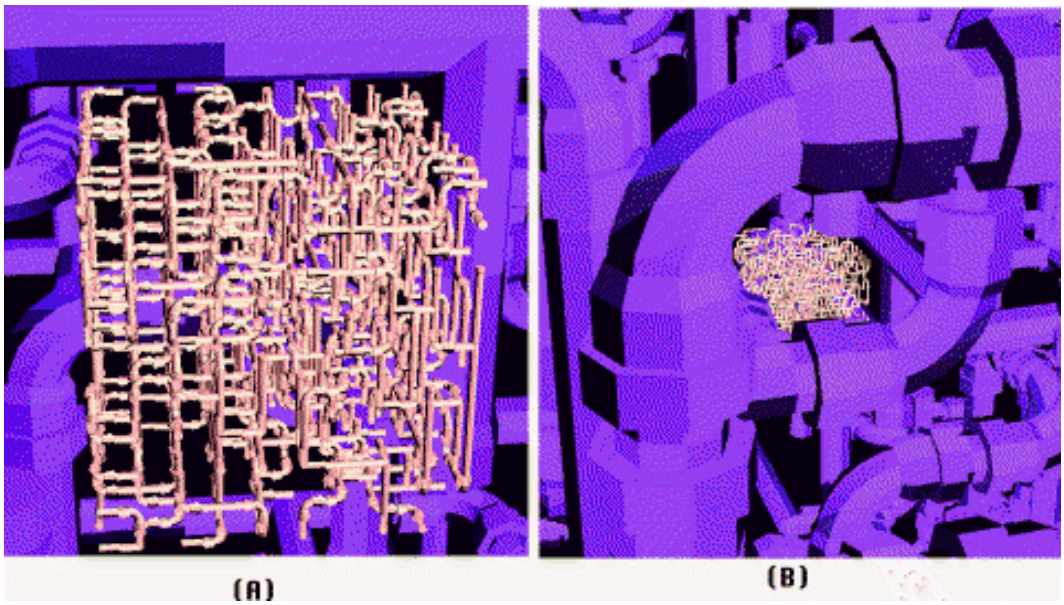


Figure 8: Interactive Interference Detection on Complex Interweaving Pipeline: $140,000$ polygons each; Average time to perform collision query: 4.2 msec on SGI Reality Engine with 90MHz R8000 CPU

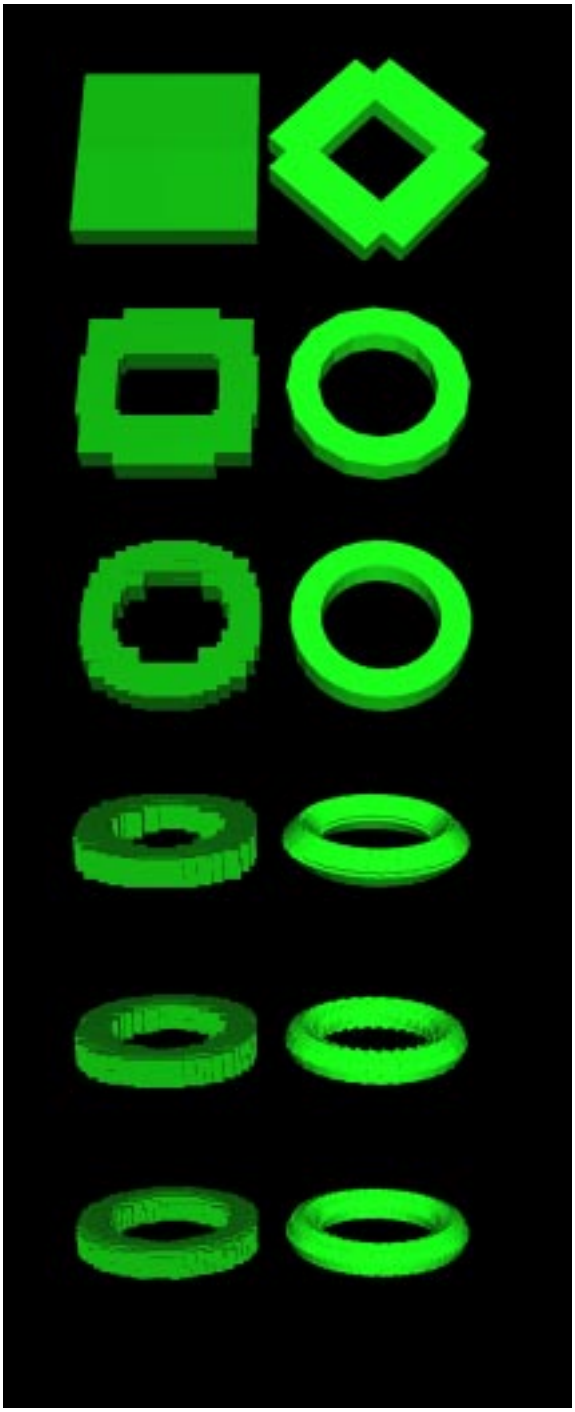


Figure 9: AABBs vs. OBBs: Approximation of a Torus – This shows OBBs converging to the shape of a torus more rapidly than AABBs.

I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments

Jonathan D. Cohen Ming C. Lin * Dinesh Manocha Madhav K. Ponamgi
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
{cohenj,lin,manocha,ponamgi}@cs.unc.edu

ABSTRACT:
We present an exact and interactive collision detection system, I-COLLIDE, for large-scale environments. Such environments are characterized by the number of objects undergoing rigid motion and the complexity of the models. The algorithm does not assume the objects' motions can be expressed as a closed form function of time. The collision detection system is general and can be easily interfaced with a variety of applications. The algorithm uses a two-level approach based on pruning multiple-object pairs using bounding boxes and performing exact collision detection between selected pairs of polyhedral models. We demonstrate the performance of the system in walkthrough and simulation environments consisting of a large number of moving objects. In particular, the system takes less than 1/20 of a second to determine all the collisions and contacts in an environment consisting of more than a 1000 moving polytopes, each consisting of more than 50 faces on an HP-9000/750.

1 INTRODUCTION

Collision detection is a fundamental problem in computer animation, physically-based modeling, computer simulated environments and robotics. In these applications, an object's motion is constrained by collisions with other objects and by other dynamic constraints. The problem has been well studied in the literature. However, no good general collision detection algorithms and systems are known for interactive large-scale environments.

A large-scale virtual environment, like a walkthrough, creates a computer-generated world, filled with real and virtual objects. Such an environment should give the user a feeling of presence, which includes making the images of both the user and the surrounding objects feel solid. For example, the objects should not pass through each other, and things should move as expected when pushed, pulled or grasped. Such actions require accurate collision detection. However, there may be hundreds, even thousands of objects in the virtual world, so a brute-force approach that tests all possible pairs for collisions is not acceptable. Efficiency is critical in a virtual environment, otherwise its interactive nature is lost [24]. A fast and interactive collision detection algorithm is a fundamental component of a complex virtual environment.

The objective of collision detection is to report all geometric contacts between objects. If we know the positions and orientations of the objects in advance, we can solve collision detection as a function of time. However, this is not the case in virtual environments or other interactive applications. In fact, in a walkthrough environment, we usually do not have any information regarding the maximum velocity or acceleration, because the user may move with abrupt changes in direction and speed. Due to these unconstrained variables, collision detection is currently considered to be one of the major bottlenecks in building interactive simulated environments [20].

Main Contribution: We present a collision detection algorithm and system for interactive and exact collision detection in complex environments. In contrast to the previous work, we show that accurate, interactive performance can be attained in most environments if we use coherence to speed up pairwise interference tests and to reduce the actual number of these tests we perform. We are able to successfully trim the $O(n^2)$ possible interactions of n simultaneously moving objects to $O(n+m)$ where m is the number of objects *very close* to each other. In particular, two objects are very close, if their axis-aligned bounding boxes overlap. Our approach is flexible enough to handle dense environments without making assumptions about object velocity or acceleration. The system has been successfully applied to architectural walkthroughs and simulated environments and works well in practice.

The rest of the paper is organized as follows. In Section 2, we review some of the previous work in collision detection. Section 3 defines the concept of coherence and describes an exact pairwise collision detection algorithm which applies it. We describe our algorithm for collision detection between multiple objects in Section 4 and discuss its implementation in Sections 5 and 6. Section 7 presents our experimental results on walkthrough environments and simulations.

*Currently at NC A & T State University, Greensboro. Approved by ARPA for public release; distribution unlimited

2 PREVIOUS WORK

The problem of collision detection has been extensively studied in robotics, computational geometry, and computer graphics. The goal in robotics has been the planning of collision-free paths between obstacles [15]. This differs from virtual environments and physically-based simulations, where the motion is subject to dynamic constraints or external forces and cannot typically be expressed as a closed form function of time [1, 3, 11, 18, 20, 21].

At the same time, the emphasis in the computational geometry has been on theoretically efficient intersection detection algorithms [22]. Most of them are restricted to a static instance of the problem and are non-trivial to implement. For convex 3-polytopes [1] linear time algorithms based on linear programming and tracking closest points [10] have been proposed. More recently, temporal and geometric coherence have been used to devise algorithms based on checking local features of pairs of convex 3-polytopes [3, 17]. Alonso et al.[1] use bounding boxes and spatial partitioning to test all $O(n^2)$ pairs of arbitrary polyhedral objects.

Different methods have been proposed to overcome the bottleneck of $O(n^2)$ pairwise tests in an environment of n bodies. The simplest of these are based on spatial subdivision. The space is divided into cells of equal volume, and at each instance the objects are assigned to one or more cells. Collisions are checked between all object pairs belonging to a particular cell. This approach works well for sparse environments in which the objects are uniformly distributed through the space. Another approach operates directly on four-dimensional volumes swept out by object motion over time [4, 14].

None of these algorithms adequately address the issue of collision detection in a virtual environment which requires performance at interactive rates for thousands of pairwise tests. Hubbard has proposed a solution to address this problem by trading accuracy for speed [14]. In an early extension of their work, Lin and Canny [16] proposed a scheduling scheme to handle multiple moving objects. Dworkin and Zeltzer extended this work for a sparse model [7].

3 BACKGROUND

In this section, we highlight the importance of coherence in dynamic environments. We briefly review the algorithm for exact pairwise collision detection and present our multi-body collision detection scheme, both of which exploit coherence to achieve efficiency.

3.1 Temporal and Geometric Coherence

Temporal coherence is the property that the application state does not change significantly between time steps, or frames. The objects move only slightly from frame to frame. This slight movement of the objects translates into geometric coherence, because their geometry, defined by the vertex coordinates, changes minimally between frames. The underlying *assumption* is that the *time steps are small enough* that the objects to do not travel large distances between frames.

[1] We shall refer to a bounded d-dimensional polyhedral set as a convex d-polytope, or briefly polytope. In common parlance, "polyhedron" is used to denote the union of the boundary and of the interior in E^3.

3.2 Pairwise Collision Detection for Convex Polytopes

We briefly review the Lin-Canny collision detection algorithm which tracks closest points between pairs of convex polytopes [16, 17]. This algorithm is used at the lowest level of collision detection to determine the exact contact status between convex polytopes. The method maintains a pair of closest features for each convex polytope pair and calculates the Euclidean distance between the features to detect collisions. This approach can be used in a static environment, but is especially well-suited for dynamic environments in which objects move in a sequence of small, discrete steps.

The method takes advantage of coherence: the closest features change infrequently as the polytopes move along finely discretized paths. The algorithm runs in *expected constant time* if the polytopes are not moving swiftly. Even when a closest feature pair is changing rapidly, the algorithm takes only slightly longer (the running time is proportional to the number of feature pairs traversed, which is a function of the relative motion the polytopes undergo). The method for finding closest feature pairs is based on Voronoi regions. The algorithm starts with a candidate pair of features, one from each polytope, and checks whether the closest points lie on these features. Since the polytopes and their faces are convex, this is a local test involving only the neighboring features of the current candidate features. If either feature fails the test, the algorithm steps to a neighboring feature of one or both candidates, and tries again. With some simple preprocessing, the algorithm can guarantee that every feature has a constant number of neighboring features.

3.3 Penetration Detection for Convex Polytopes

The core of the collision detection algorithm is built using the properties of Voronoi regions of convex polytopes. The Voronoi regions form a partition of space outside the polytope. When polytopes interpenetrate, some features may not fall into any Voronoi regions. This can at times lead to cycling of feature pairs. To circumvent this problem, we partition the *interior space* of the convex polytopes. The partitioning does not have to form the exact internal Voronoi regions, because we are not interested in knowing the closest features between two interpenetrating polytopes, but only detecting such a case. So instead we use pseudo-Voronoi regions, obtained by joining each vertex of the polytope with the centroid of the polytope [21].

Given a partition of the exterior and the interior of the polytope, we walk from the external Voronoi regions into the pseudo-internal Voronoi regions when necessary. If either of the closest features falls into a pseudo-Voronoi region at the end of the walk, we know the objects are interpenetrating. Ensuring convergence as we walk through pseudo-internal Voronoi regions requires special case analysis and will be omitted here.

3.4 Extension to Non-Convex Objects

We extend the collision detection algorithm for convex polytopes to handle non-convex objects, such as articu-

lated bodies, by using a hierarchical representation. In the hierarchical representation, the internal nodes can be convex or non-convex sub-parts, but *all* the leaf nodes are convex polytopes or features [21].

Beginning with the leaf nodes, we construct either a convex hull or other bounding volume and work up the tree, level by level, to the root. The bounding volume associated with each node is the bounding volume of the union of its children; the root's bounding volume encloses the whole hierarchy. For instance, a hand may have individual joints in the leaves, fingers in the internal nodes, and the entire hand in the root.

We test for collision between a pair of these hierarchical trees recursively. The collision detection algorithm first tests for collision between the two parent nodes. If there is no collision between the two parents, the algorithm returns the closest feature pair of their bounding volumes. If there is a collision, the algorithm expands their children and recursively proceeds down the tree to determine if a collision actually occurs. More details are given in [21].

4 MULTIPLE-OBJECT COLLISION DETECTION

Large-scale environments consist of stationary as well as moving objects. Let there be N moving objects and M stationary objects. Each of the N moving objects can collide with the other moving objects, as well as with the stationary ones. Keeping track of $\binom{N}{2} + NM$ pairs of objects at every time step can become time consuming as N and M get large. To achieve interactive rates, we must reduce this number before performing pairwise collision tests. The overall architecture of the multiple object collision detection algorithm is shown in Fig. 1.

Sorting is the key to our pruning approach. Each object is surrounded by a 3-dimensional bounding volume. We sort these bounding volumes in 3-space to determine which pairs are overlapping. We only need to perform exact pairwise collision tests on these remaining pairs.

However, it is not intuitively obvious how to sort objects in 3-space. We use a *dimension reduction* approach. If two bodies collide in a 3-dimensional space, their orthogonal projections onto the xy, yz, and xz-planes and x, y, and z-axes must overlap. Based on this observation, we choose axis-aligned bounding boxes as our bounding volumes. We efficiently project these bounding boxes onto a lower dimension, and perform our sort on these lower-dimensional structures.

This approach is quite different from the typical space partitioning approaches used to reduce the number of pairs. A space partitioning approach puts considerable effort into choosing good partition sizes. But there is *no* partition size that prunes out object pairs as ideally as testing for bounding box overlaps. Partitioning schemes may work well for environments where N is small compared to M, but object sorting works well whether N is small or large.

4.1 Bounding Volumes

Many collision detection algorithms have used bounding boxes, spheres, ellipses, etc. to rule out collisions between objects which are far apart. We use bounding box overlaps to trigger the *exact collision detection* algorithm.

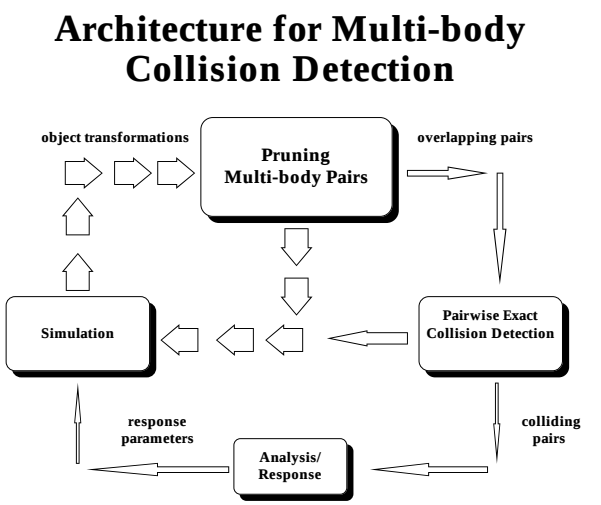


Figure 1: Architecture for Multiple Body Collision Detection Algorithm

We have considered two types of axis-aligned bounding boxes: fixed-size bounding cubes (fixed cubes) and dynamically-resized rectangular bounding boxes (dynamic boxes).

• Fixed-Size Bounding Cubes:
We compute the size of the fixed cube to be large enough to contain the object at *any* orientation. We define this axis-aligned cube by a *center* and a *radius*. Fixed cubes are easy to recompute as objects move, making them well-suited to dynamic environments. If an object is nearly spherical the fixed cube fits it well.

As preprocessing steps we calculate the center and radius of the fixed cube. At each time step as the object moves, we recompute the cube as follows:

1. Transform the center using one vector-matrix multiplication.
2. Compute the minimum and maximum x, y, and z-coordinates by subtracting and adding the radius from the coordinates of the center.

Step 1 involves only one vector-matrix multiplication. Step 2 needs six arithmetic operations (3 additions and 3 subtractions).

• Dynamically Rectangular Bounding Boxes:
We compute the size of the rectangular bounding box to be the tightest axis-aligned box containing the object at a *particular* orientation. It is defined by its minimum and maximum x, y, and z-coordinates (for a convex object, these must correspond to coordinates of up to 6 of its vertices). As an object moves, we must recompute its minima and maxima, taking into account the object's orientation. For oblong objects rectangular boxes fit better than cubes, resulting in fewer overlaps. This is advantageous as long as few of the objects are moving, as in a

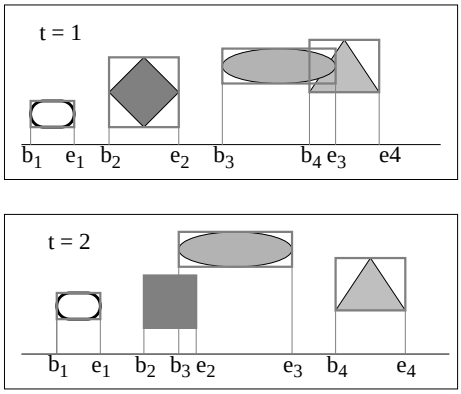


Figure 2: Bounding Box Behavior

walkthrough environment. In such an environment, the savings gained by the reduced number of pairwise collision detection tests outweigh the cost of computing the dynamically-resized boxes.

As a precomputation, we compute each object's initial minima and maxima along each axis. It is assumed that the objects are convex. For non-convex polyhedral models, the following algorithm is applied to their convex hulls. As an object moves, we recompute its minima and maxima at each time step as follows:

1. Check to see if the current minimum (or maximum) vertex for the x, y, or z-coordinate still has the smallest (or largest) value in comparison to its neighboring vertices. If so we are finished.

2. Update the vertex for that extremum by replacing it with the neighboring vertex with the smallest (or largest) value of all neighboring vertices. Repeat the entire process as necessary.

This algorithm recomputes the bounding boxes at an expected constant rate. Once again, we are exploiting the temporal and geometric coherence, in addition to the locality of convex polytopes.

We do not transform all the vertices as the objects undergo motion. As we are updating the bounding boxes new positions are computed for current vertices using matrix-vector multiplications. We can optimize this approach by realizing that we are only interested in *one* coordinate value of each extremal vertex, say the x coordinate while updating the minimum or maximum value along the x-axis. Therefore, there is no need to transform the other than coordinates in order to compare neighboring vertices. This reduces the number of arithmetic operations by two-thirds.

4.2 One-Dimensional Sweep and Prune

The one-dimensional sweep and prune algorithm begins by projecting each three-dimensional bounding box onto the x, y, and z axes. Because the bounding boxes are axis-aligned, projecting them onto the coordinate axes results in intervals (see Fig. 2). We are interested in overlaps among these intervals, because a pair of bounding boxes can overlap *if and only if* their intervals overlap in all three dimensions.

We construct three lists, one for each dimension. Each list contains the values of the endpoints of the intervals corresponding to that dimension. By sorting these lists, we can determine which intervals overlap. In the general case, such a sort would take $O(n \log n)$ time, where n is the number of objects. We can reduce this time bound by keeping the sorted lists from the previous frame, changing only the values of the interval endpoints. In environments where the objects make relatively small movements between frames, the lists will be nearly sorted, so we can sort in expected $O(n)$ time, as shown in [19, 3]. *Insertion sort* works well for previously sorted lists.

In addition to sorting, we need to keep track of changes in overlap status of interval pairs (i.e. from overlapping in the last time step to non-overlapping in the current time step, and vice-versa). This can be done in $O(n + e_x + e_y + e_z)$ time, where $e_x, e_y,$ and e_z are the number of exchanges along the $x, y,$ and z-axes. This also runs in expected linear time due to coherence, but in the worst case $e_x, e_y,$ and e_z can each be $O(n^2)$ with an extremely small constant.

Our method is suitable for dynamic environments where coherence is preserved. In computational geometry literature several algorithms exist that solve the static version of determining 3-D bounding box overlaps in $O(n \log^2 n + s)$ time, where s is the number of pairwise overlaps [12, 13]. We have reduced this to $O(n + s)$ by using coherence.

4.3 Two-Dimensional Intersection Tests

The two-dimensional intersection algorithm begins by projecting each three-dimensional axis-aligned bounding box onto any two of the x-y, x-z, and y-z planes. Each of these projections is a rectangle in 2-space. Typically there are fewer overlaps of these 2-D rectangles than of the 1-D intervals used by the sweep and prune technique. This results in fewer swaps as the objects move. In situations where the projections onto one-dimension result in densely clustered intervals, the two-dimensional technique is more efficient. The interval tree is a common data structure for performing such two-dimensional range queries [22].

Each query of an interval intersection takes $O(\log n + k)$ time where k is the number of reported intersections and n is the number of intervals. Therefore, reporting intersections among n rectangles can be done in $O(n \log n + K)$ where K is the total number of intersecting rectangles [8].

4.4 Alternatives to Dimension Reduction

There are many different methods for reducing the number of pairwise tests, such as binary space partitioning (BSP) trees [23], octrees, etc.

Several practical and efficient algorithms are based on uniform space division. Divide space into unit cells (or volumes) and place each object in some cell(s). To check for collisions, examine the cell(s) occupied by each object to verify if the cell(s) is(are) shared by other objects. Choosing a near-optimal cell size is difficult, and failing to do so results in large memory usage and computational inefficiency.

5 IMPLEMENTATION

In this section we describe the implementation details of I-COLLIDE based on the Sweep and Prune algorithm, the exact collision detection algorithm, the multi-body simulation, and their applications to walkthrough and simulations.

5.1 Sweep and Prune

As described earlier, the Sweep and Prune algorithm reduces the number of pairwise collision tests by eliminating polytope pairs that are far apart. It involves three steps: calculating bounding boxes, sorting the minimum and maximum coordinates of the bounding boxes as the algorithm sweeps through each list, and determining which bounding boxes overlap. As it turns out, we do the second and third steps simultaneously.

Each bounding box consists of a minimum and a maximum coordinate value for each dimension: x, y, and z. These minima and maxima are maintained in three separate lists, one for each dimension. We sort each list of coordinate values using insertion sort, while maintaining an overlap status for each bounding box pair. The overlap status consists of a boolean flag for each dimension. Whenever all three of these flags are set, the bounding boxes of the polytope pair overlap. These flags are only modified when insertion sort performs a swap. We decide whether or not to toggle a flag based on whether the coordinate values both refer to bounding box minima, both refer to bounding box maxima, or one refers to a bounding box minimum and the other a maximum.

When a flag is toggled, the overlap status indicates one of three situations:

1. All three dimensions of this bounding box pair now overlap. In this case, we add the corresponding polytope pair to a list of active pairs.

2. This bounding box pair overlapped at the previous time step. In this case, we remove the corresponding polytope pair from the active list.

3. This bounding box pair did not overlap at the previous time step and does not overlap at the current time step. In this case, we do nothing.

When sorting is completed for this time step, the active pair list contains all the polytope pairs whose bounding boxes currently overlap. We pass this active pair list to the exact collision detection routine to find the closest features of all these polytope pairs and determine which, if any, of them are colliding.

5.2 Exact collision detection

The collision detection routine processes each polytope pair in the active list. The first time a polytope pair is considered, we select a random feature from each polytope; otherwise, we use the previous closest feature pair as a starting point. This previous closest feature pair may not be a good guess when the polytope pair has just become active. Dworkin and Zeltzer [7] suggest precomputing a lookup table for each polytope to help find better starting guesses.

5.3 Multi-body Simulation

The multi-body simulation is an application we developed to test the I-COLLIDE system. It represents a general, non-restricted environment in which objects move in an arbitrary fashion resulting in collisions with simple impulse responses.

While we can load any convex polytopes into the simulation, we typically use those generated by the tessellation of random points on a sphere. Unless the number of vertices is large, the resulting polytopes are not spherical in appearance; they range from oblong to fat. The simulation parameters of the polytopes were their number, their complexity measured as the number of faces, their rotational velocity, their translational velocity, the density of their environment measured as the ratio of polytope volume to environment volume, and the bounding volume method used for the Sweep and Prune (fixed-size or dynamically-resized boxes).

The simulation begins by placing the polytopes at random positions and orientations. At each time step, the positions and orientations are updated using the translational and rotational velocities (since the detection routines make no use of pre-defined path, the polytopes' paths could just as easily be randomized at each time step). The simulation then calls the I-COLLIDE system and receives a list of colliding polytope pairs. It exchanges the translational velocities of these pairs to simulate an elastic reaction. Objects also rebound off the walls of the constraining volume.

We use this simulation to test the functionality and speed of the detection algorithm. In addition, we are able to visually display some of the key features. For example, the bounding boxes of the polytopes can be rendered at each time step. When the bounding boxes of a polytope pair overlap, we can render a line connecting the closest features of this polytope. It is also possible to show all pairs of closest features at each time step. These visual aids have proven to be useful in indicating actual collisions and additional geometric information for algorithmic study and analysis. See Frame 1 at the end for an example of the simulation.

5.4 Walkthrough

The walkthrough is a head-mounted display application that involves a large number of polytopes depicting a realistic scene. The integration of our library into such an environment demonstrates that an interactive environment can use our collision detection library without affecting the application's real-time performance.

The walkthrough creates a virtual environment (our video shows a kitchen and a porch). The user travels through this environment, interacting with the polytopes: picking up virtual objects, changing their scale, and moving them around. Whenever the user's hand collides with the polytopes in the environment, the walkthrough provides feedback by making colliding bodies appear red.

We have incorporated the collision detection library routines into the walkthrough application. The scene is composed of polytopes, most of which are stationary. The user's hand, composed of several convex polytopes, moves through this complex environment, modifying other polytopes in the environment. Frames 2-4 show a sequence

of shots from a kitchen walkthrough environment. The pictures show images as seen by the left eye. Frames 5-6 show the user in a porch walkthrough.

6 SYSTEM ISSUES

To use I-COLLIDE, the application first loads a library of polytopes. The file format we use is fairly simple. It is straightforward to convert polytope data from some other format (perhaps the output of some 3D modelling package) to this minimal format for I-COLLIDE. After loading the polytopes, the application then chooses some polytope pairs to activate for collision detection. This set of active pairs is fully configurable between collision passes. Inside the application loop, the application informs I-COLLIDE of the world transformation for each polytope as it moves around. At any point, the application may call the collision test routine. I-COLLIDE returns a list of all the colliding pairs, including a pair of colliding features for each. The application then responds to these collisions in some appropriate way.

6.1 Space Issues

For each pair of objects, I-COLLIDE maintains a structure that contains the bounding box overlap status and the closest feature pair between the objects. These structures conceptually form an upper-triangular $O(n^2)$ matrix. We access an entry in O(1) time by using the object id numbers as (row, column) entries. If only a few pairs of objects are interacting, then the $O(n^2)$ can be reduced at the expense of slightly larger access time. For example, we can traverse a sparse matrix list to access an entry.

6.2 Geometric Robustness

In practice there are several types of degeneracies or errors that can occur in the convex polytope models: duplicate vertices, extraneous vertices, backfacing polygons, tracking error, non-planar faces, non-convex faces, non-convex polytopes, disconnected faces, etc. We have written a pre-processor to scan for common degeneracies and correct them when possible.

6.3 Numerical Issues

Numerical robustness is an important issue in the exact collision detection code. There are many special case geometrical tests in this module, and it is difficult to ensure that the algorithm will not get into a cycle due to degenerate overlap. We deal with this by performing all of our feature tests to some tolerance. Without such a tolerance, floating point errors might allow some of the feature tests to cycle infinitely. We have not observed this in practice so far, and have been careful to make the tests stable in the presence of small errors.

The multi-body sweep and prune code is also designed to resist small numerical errors. The bounding box of each polytope is extended by a small epsilon [2] in each direction. In addition to insulating the overlap tests from errors, this precaution also helps give the exact collision detection test a chance of being activated before the objects are actually penetrating.

[2] This quantity is a function of velocity between the object pairs.

6.4 Generality

While the multi-body pruning code works well with the exact collision detection routine, it functions independently of the underlying collision detection routine. This second level collision routine might or might not be exact, and it certainly need not be limited to handling convex polytopes.

7 PERFORMANCE ANALYSIS

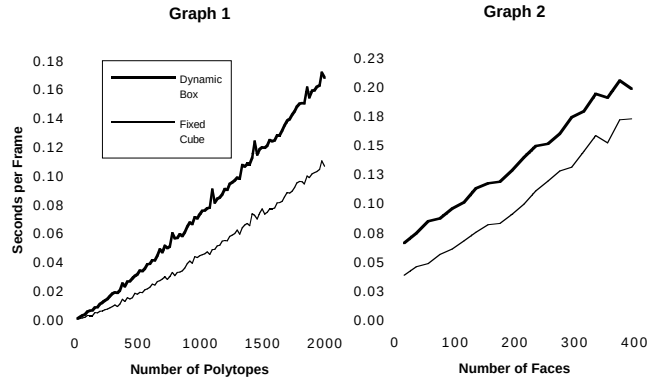
We measured the performance of the collision detection algorithm using the multi-body simulation as a benchmark. We profiled the entire application and tabulated the CPU time of only the relevant detection routines. All of these tests were run on an HP-9000/750. The main routines involved in collision detection are those that update the bounding boxes, sort the bounding boxes, and perform exact collision detection on overlapping bounding boxes. As described in the implementation section we use two different types of bounding boxes. Using fixed cubes as bounding boxes resulted in low collision time for the parameter ranges we tested.

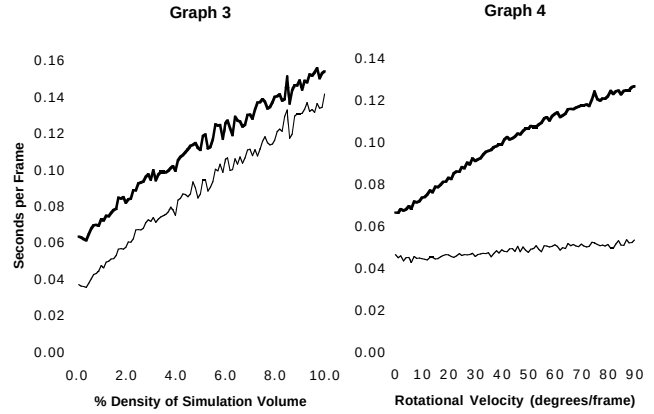
In each of the first four graphs, we plot two lines. The bold line displays the performance of using dynamically-resized bounding boxes whereas the other line shows the performance of using fixed-size cubes. All five graphs refer to "seconds per frame", where a frame is one step of the simulation, involving one iteration of collision detection without rendering time. Each graph was produced with the following parameters, by holding all but one constant.

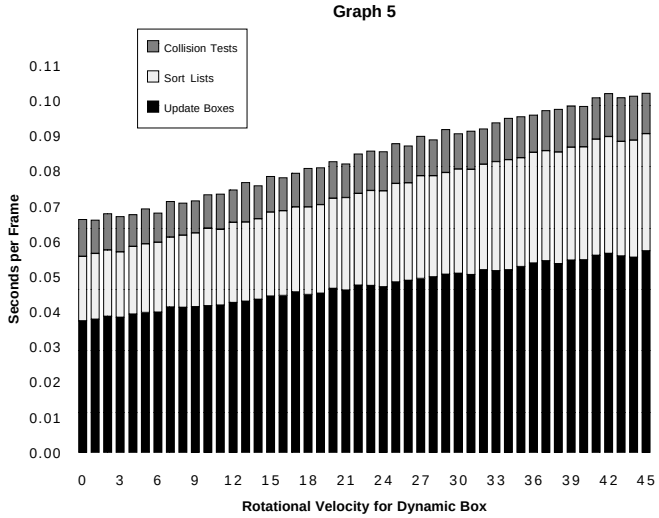
- *Number of polytopes.* The default value is a 1000 polytopes.
- *Complexity of polytopes*, which we define as the number of faces. The default value is 36 faces.
- *Rotational velocity*, which we define as the number of degrees the object rotates about an axis passing through its centroid. The default value is 10 degrees.
- *Translational velocity*, which we define in relation to the object's size. We estimate a radius for the object, and define the velocity as the percentage of its radius the object travels each frame. The default value is 10%.
- *Density*, which we define as the percentage of the environment volume the polytopes occupy. The default value is 1.0%.

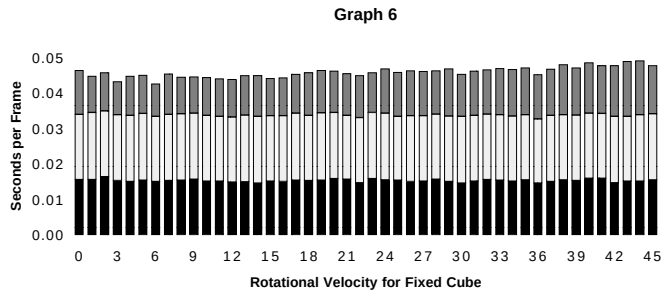
In the graphs, the timing results do not include computing each polytope's transformation matrix, rendering times, and of course any minor initialization cost. We ignored these costs, because we wanted to measure the cost of collision detection alone.

Graph 1 shows how the number of seconds per frame scales with an increasing number of polytopes. We took 100 uniformly sampled data points from 20 to 2000 polytopes. The fixed and dynamic bounding box methods scale nearly linearly with a small higher-order term. The dynamic bounding box method results in a slightly larger non-linear term because the resizing of bounding boxes









causes more swaps during sorting. This is explained further in our discussion of Graph 5. The seconds per frame numbers in Graph 1 compare very favorably with the work of Dworkin and Zeltzer [7] as well as those of Hubbard [14]. For a 1000 polytopes in our simulation, our collision time results in **23 frames per second** using the fixed bounding cubes.

Graph 2 shows how the number faces affects the collision time. We took 20 uniformly sampled data points. For the dynamic bounding box method, increasing the model complexity increases the time to update the bounding boxes because finding the minimum and maximum values requires walking a longer path around the polytope. Surprisingly, the time to sort the bounding boxes decreases with number of faces, because the polytopes become more spherical and fat. As the polytopes become more spherical and fat, the bounding box dimensions change less as the polytopes rotate, so fewer swaps are need in the sweeping step. For the fixed bounding cube, the time to update the bounding boxes and to sort them is almost constant.

Graph 3 shows the effect of changes in the density of the simulation volume. For both bounding box methods, increasing the density of polytope volume to simulation volume results in a larger sort time and more collisions. The number of collisions scales linearly with the density of the simulation volume. As the graph shows, the overall collision time scales well with the increases in density.

Graphs 4 through 6 show the effect of rotational velocity on the overall collision time. The slope of the line for the dynamic bounding box method is much larger than that of the fixed cube method. There are two reasons for this difference. The first reason is that the increase in rotational velocity increases the time required to update the dynamic bounding boxes. When we walk from the old maxima and minima to find the new ones, we need to traverse more features.

The second reason is the larger number of swapped minima and maxima in the three sorted lists. Although the three-dimensional volume of the simulation is fairly sparse, each one-dimensional view of this volume is much more dense, with many bounding box intervals overlapping. As the boxes grow and shrink, they cause many swaps in these one-dimensional lists. And as the rotational velocity increases, the boxes change size more rapidly.

Graph 6 clearly shows the advantages of the static box method. Both the update bounding box time and sort lists time are *almost* constant as the rotational velocity increases.

All of our tests show *exact* collision detection in demanding environments can be achieved without incurring expensive time penalties. The architectural walkthrough models showed no perceptible performance degradation when collision detection was added (as in Frame 2 to 5).

8 CONCLUSION

Collision detection has been considered a major bottleneck in computer-simulated environments. By making use of geometric and temporal coherence, our algorithm and system detects collisions more efficiently and effectively than earlier algorithms. Under many circumstances our system produces collision frame rates over 20 hertz

for environments with over a 1000 moving complex polytopes. Our walkthrough experiments showed no degradation of frame rates when collision detection was added. We are currently working on incorporating general polyhedral and spline models into our system and extending these algorithms to deformable models.

9 ACKNOWLEDGEMENTS

We are grateful to John Canny and David Baraff for productive discussions and to Brian Mirtich for his help in implementation of the convex polytope pair algorithm. The kitchen and porch models used in the walkthrough applications were designed by the UNC Walkthrough group, headed by Fred Brooks. This work was supported in part by DARPA ISTO order A410, NSF grant MIP-9306208, NSF grant CCR-9319957, ARPA contract DABT63-93-C-0048, ONR contract N00014-94-1-0738 and NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization, NSF Prime Contract 8920219.

References

[1] A.Garica-Alonso, N.Serrano, and J.Flaquer. Solving the collision detection problem. *IEEE Computer Graphics and Applications*, 13(3):36–43, 1994.

[2] D. Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. *ACM Computer Graphics*, 24(4):19–28, 1990.

[3] D. Baraff. *Dynamic simulation of non-penetrating rigid body simulation.* PhD thesis, Cornell University, 1992.

[4] S. Cameron. Collision detection by four-dimensional intersection testing. *Proceedings of International Conference on Robotics and Automation*, pages pp. 291–302, 1990.

[5] S. Cameron. Approximation hierarchies and s-bounds. In *Proceedings. Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 129–137, Austin, TX, 1991.

[6] J. Cohen, M. Lin, D. Manocha, and K. Ponamgi. Interactive and exact collision detection for large-scaled environments. Technical Report TR94-005, Department of Computer Science, University of North Carolina, 1994.

[7] P. Dworkin and D. Zeltzer. A new model for efficient dynamics simulation. *Proceedings Eurographics workshop on animation and simulation*, pages 175–184, 1993.

[8] H. Edelsbrunner. A new approach to rectangle intersections, Part I. *Internat. J. Comput. Math.*, 13:209–219, 1983.

[9] J. Snyder et. al. Interval methods for multi-point collisions between time dependent curved surfaces. In *Proceedings of ACM Siggraph*, pages 321–334, 1993.

[10] 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:pp. 193–203, 1988.

[11] J. K. Hahn. Realistic animation of rigid bodies. *Computer Graphics*, 22(4):pp. 299–308, 1988.

[12] 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.

[13] H.Six and D.Wood. Counting and reporting intersections of D-ranges. *IEEE Transactions on Computers*, pages 46–55, 1982.

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

[15] J.C. Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[16] M. Lin and J. Canny. Efficient collision detection for animation. In *Proceedings of the Third Eurographics Workshop on Animation and Simulation*, Cambridge, England, 1991.

[17] 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.

[18] M. Moore and J. Wilhelms. Collision detection and response for computer animation. *Computer Graphics*, 22(4):289–298, 1988.

[19] M.Shamos and D.Hoey. Geometric intersection problems. *Proc. 17th An. IEEE Symp. Found. on Comput. Science*, pages 208–215, 1976.

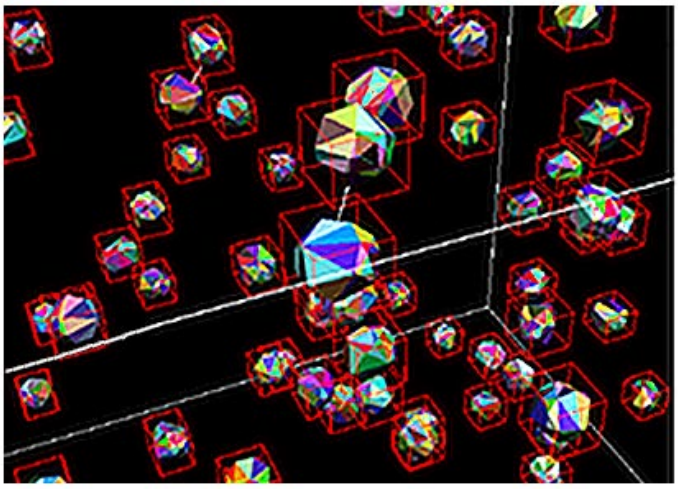
[20] A. Pentland. Computational complexity versus simulated environment. *Computer Graphics*, 22(2):185–192, 1990.

[21] M. Ponamgi, D. Manocha, and M. Lin. Incremental algorithms for collision detection between solid models. Technical Report TR94-061, Department of Computer Science, University of North Carolina, Chapel Hill, 1994.

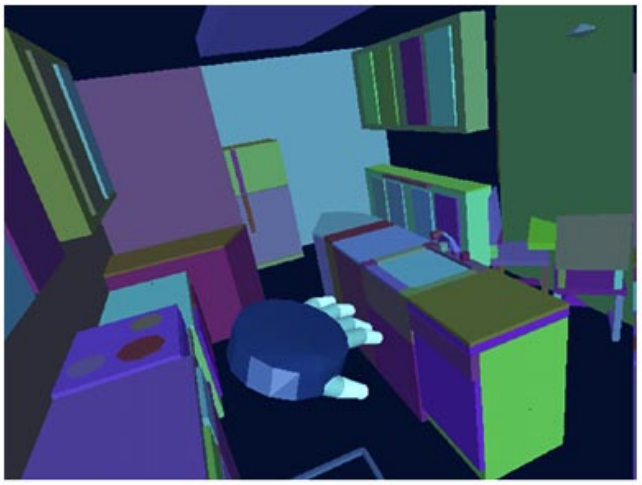
[22] F.P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, New York, 1985.

[23] W.Thibault and B.Naylor. Set operations on polyhedra using binary space partitioning trees. *ACM Computer Graphics*, 4, 1987.

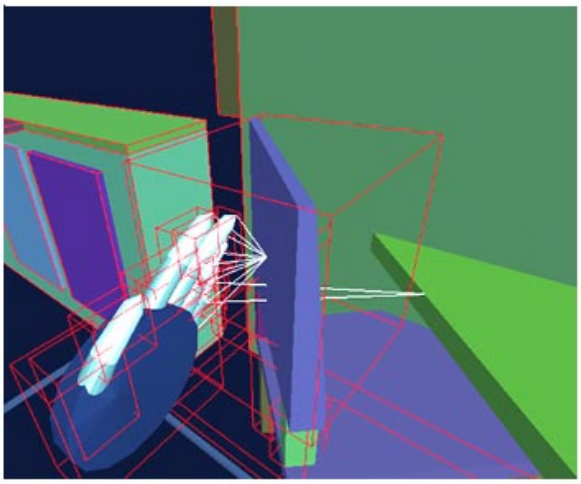
[24] D. Zeltzer. Autonomy, interaction and presence. *Presence*, 1(1):127, 1992.



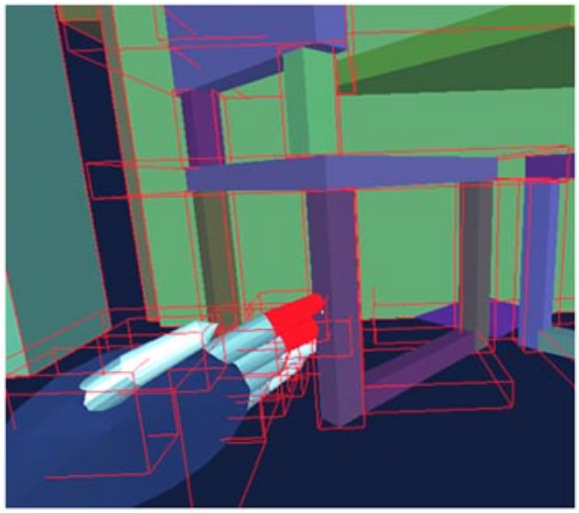
Frame 1: 100 polytopes, 1% density, 56 faces.
Pair of bounding boxes overlapping.



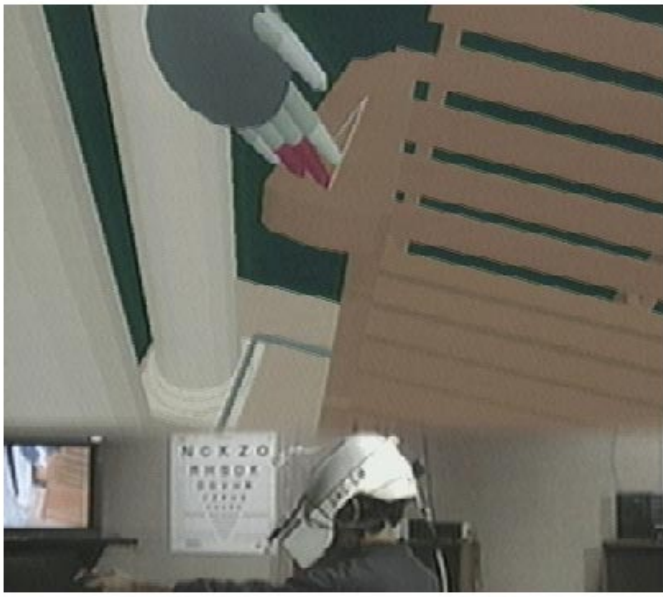
Frame 2: A multi-polytope hand moves through
a kitchen walkthrough environment.



Frame 3: When bounding boxes overlap,
closest feature pairs appear.



Frame 4: Red polytopes indicate collisions.



Frames 5 and 6: The hand touches a swing in a porch walkthrough.

SWIFT: Accelerated Proximity Queries Using Multi-Level Voronoi Marching

Stephen A. Ehmann **Ming C. Lin**

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
{ehmann,lin}@cs.unc.edu
http://www.cs.unc.edu/∼geom/SWIFT/

Abstract

We present an accelerated proximity query algorithm between moving convex polyhedra. The algorithm combines Voronoi-based feature tracking with a multi-level-of-detail representation, in order to adapt to the variation in levels of coherence and speed up the computation. It provides a progressive refinement framework for collision detection and distance queries. We have implemented our algorithm and have observed significant performance improvements in our experiments, especially on scenarios where the motion coherence is low.

Keywords: Collision detection, level-of-details, Voronoi diagrams.

1 Introduction

Proximity queries, i.e. distance[1] computations and the closely related collision detection problems, are ubiquitous in robotics, design automation, manufacturing, assembly and virtual prototyping. The set of tasks include motion planning, sensor-based manipulation, assembly and disassembly, dynamic simulation, maintainability study, simulation-based design, tolerance verification, and ergonomics analysis.

Proximity queries have been extensively studied in robotics and several specialized algorithms have been proposed for convex polyhedra as well as hierarchical approaches for general geometric models. In this paper, we present a novel algorithm that precomputes a hierarchy composed of a series of bounded error levels-of-detail (LODs) and uses them to accelerate proximity queries. Algorithms to generate LODs for polygonal models have been widely used for rendering acceleration, animation and simulation applications [12, 22, 24]. One of our goals is to take advantage of multiresolution representations commonly used in rendering applications and use them for proximity queries as well.

[1]Distance is commonly defined as the Euclidean separation distance by many applications, and we adopt the same definition in this paper.

Given a convex polyhedron, our algorithm precomputes a bounded error level-of-detail (LOD) hierarchy. It constructs a series of bounding volumes that enclose the original polyhedron. Then, it establishes parent-child relationships between the features of adjacent levels. At runtime, the hierarchy is traversed according to the type of query being performed. Within each level, the surface of an object is "marched" across using a modified Lin-Canny [16] closest feature tracking algorithm based on Voronoi regions. We refer to such a technique as "Voronoi marching" in this paper. Spatial locality and temporal coherence is captured by both the Voronoi regions and the hierarchy of LODs.

The algorithm has been implemented and analyzed using various benchmarks. Experiments were performed using various shapes of objects and various multiresolution options. Furthermore, it is compared against a straightforward directional lookup table scheme we devised to determine overall effectiveness. We observe speedups for each type of query over a simple surface traversal.

1.1 Main Results

In this paper, we present an accelerated proximity query algorithm between moving convex polyhedra which exploits multiresolution representations. Our main contributions are:

- An accelerated proximity query algorithm between convex polyhedra using multi-level Voronoi marching. The substantial performance improvements are mainly due to the use of a multiresolution representation and a faster Voronoi marching algorithm.
- A progressive refinement algorithmic framework for proximity computation. For many applications including motion planning and tolerance verification, a relatively inexpensive *approximate* distance computation with bounded error is sufficient. Our framework allows applications to progressively refine the distance estimate to suit their needs, while minimizing overall computation cost.
- A better understanding of the issues involved in designing suitable level-of-detail representations for proximity queries. These issues include level of coherence, object aspect ratios, and contact scenarios.

1.2 Organization

The rest of the paper is organized as follows. Section 2 gives a brief survey of related work. Section 3 presents an overview of our approach. The design and computation of the multi-level-of-detail representation is described in Section 4 along with our lookup table scheme. Next, a proximity query algorithm using the multiresolution representation is described in Section 5 along with other ways to accelerate queries. Section 6 describes our prototype implementation and shows the performance of our system, SWIFT. Finally, we conclude with future research directions in Section 7.

2 Previous Work

Distance computation and intersection (collision) detection problems have been fundamental subjects of study in robotics, computational geometry, simulation, and physical-based modeling. There is a wealth of literature on both analyzing the theoretical complexity of proximity queries and on designing algorithmic solutions to achieve interactive performance. We will limit the scope of the discussion in this paper to

rigid convex polyhedra, although some of the techniques may be applicable to other domains and model representation as well.

2.1 Proximity Queries for Convex Polyhedra

Most of the earlier work has focused on algorithms for convex polyhedra. A number of algorithms with good asymptotic performance have been proposed in the computational geometry literature [6]. Using hierarchical representations, an $O(log^2 n)$ algorithm is given in [4] for the convex polyhedral overlap problem, where n is the number of vertices. This elegant approach is difficult to implement robustly in 3D, however.

Good theoretical and practical approaches based on the linear complexity of the linear programming problem are known [18, 23]. Minkowski difference and convex optimization techniques are used in [9] to compute the distance between convex polyhedra.

Erickson, et al [7] recently proposed a new class of kinetic data structures for collision detection between convex polyhedra. This class of hierarchical representations has only been analyzed for the 2D case however.

In applications involving rigid motion, geometric coherence has been exploited to design algorithms for convex polyhedra based on either traversing features using locality or convex optimization [2, 5, 16, 15, 19]. These algorithms exploit the spatial and temporal coherence between successive queries and work well in practice.

2.2 Hierarchical Representations

Bounding volume hierarchies are presently regarded as one of the most general methods for performing proximity queries between general polyhedra. Specifically, sphere trees, cone trees, axis-aligned box trees, oriented box trees, k-d trees and octrees, trees based on S-bounds, and k-dops have been used for fast intersection queries for general polyhedra, as well as polygon soups [13, 20, 10, 21, 1, 14]. For most scenarios, these hierarchies excel at intersection detection but do not do so well when it comes to distance computation.

2.3 Multiresolution Techniques

Multiresolution modeling techniques, such as model simplification, have been proposed to extract the shape of the underlying geometry [25]. A recent survey on polygonal model simplification is available [17]. The main idea behind using a multiresolution hierarchy is to compute and utilize a correspondence between the original model and a simplified one. We will discuss the use of a pair of simplification algorithms due to Dobkin and Kirkpatrick [4] and to Garland and Heckbert [8]. Cohen [3] has presented algorithms based on successive mappings and appearance preserving simplification.

For proximity query, Guibas, et al. [11] proposed an elegant approach that exploits both coherence of motion and hierarchical representation for faster distance computation. Our approach differs from their *H-Walk* algorithm in that our algorithm can easily compute an approximated distance with a guaranteed error tolerance without always descending and/or ascending the entire hierarchy.

3 Algorithm Overview

Our algorithm operates on orientable 2-manifolds that are closed and represented using triangles. The polyhedra must be convex and undergo rigid motion. We will use the terms "polyhedron" and "object" interchangeably.

Multiresolution techniques typically consist of two main components. They involve the precomputation of a hierarchical representation upon which subsequent queries are performed. We wish to compute a multiresolution representation that supports proximity queries.

In the preprocessing stage, we compute a level-of-detail representation for each object. This hierarchy is organized as a sequence of convex polyhedra P_0, P_1, P_2, $\cdots$, P_k where P_0 is the input polyhedron. Each successive polyhedron is composed of fewer features and has the property that it bounds the original object. Furthermore, a correspondence is established between successive levels in each direction of the sequence. More details of the actual construction of this representation are given in Section 4.

A query using this hierarchy is performed in much the same way for each type of proximity query. Basically, as long as certain levels of two objects are intersecting, then the hierarchy is refined. When two levels are found to be disjoint, then it may be possible to end the query at a subsequent point of refinement. Furthermore, for queries other than intersection, a tolerance may be provided which specifies how close objects must be in order to answer the query. In Section 5 the hierarchical query is described in more detail.

4 Hierarchical Representations

The multi-level-of-detail representation of each object that is constructed in the algorithm's preprocessing stage must have certain properties in order to be useful and efficient for performing proximity queries. Next, we describe desirable characteristics for the representation, discuss the steps involved in the creation of the hierarchy, and touch upon various tradeoffs along the way. Then, we discuss two hierarchy creation methods we implemented. We conclude with a description of our lookup table scheme.

4.1 Terminology

Recall that the hierarchy is organized as a sequence of convex polyhedra P_0, P_1, P_2, $\cdots$, P_k where P_0 is the input polyhedron. We will call P_0 the finest level and P_k the coarsest level. We will call the level P_{i-1} the "child" of the level P_i and the level P_{i+1} the "parent" of P_i. We use the convention that the finest object is at the bottom of the hierarchy so the term *moving up* the hierarchy means moving to a coarser level. Moving to a finer level is termed *moving down* the hierarchy.

The maximum deviation of P_i from P_0 is represented by ϵ_i. The computed distance between certain levels of two objects that are found to be disjoint is δ and the maximum error associated with the distance is ϵ. The distance tolerance given by the application is δ_{max} and the error tolerance is ϵ_{max}.

4.2 Desired Features

To design a multi-level representation for accelerating distance computation while preserving locality and coherence, our goal is to produce a hierarchy that provides the following characteristics:

- **Simplified Combinatorial Complexity:** Each level of representation in the hierarchy should have less combinatorial complexity than its child and higher combinatorial complexity than its parent.

Ideally we would like to create $O(\log n)$ levels, where n is the number of vertices in the original polyhedron P_0. This is possible if a constant fraction of features are eliminated for each level. In addition, it is wise to stop the hierarchy construction when a certain number of levels or features has been reached.

- **Bounding Volumes:** If we wish to have a mechanism to compute approximate distances with bounded error tolerances ϵ_{max}, then we can make use of the levels P_i of the hierarchy which bound the original polyhedron P_0 with a global surface deviation $\epsilon_i \leq \epsilon_{max}$. Query performance can be further improved by aiming to create a hierarchy where the bounding volumes are kept as small as possible.

- **Local Correspondence:** For each level of the hierarchy P_i for $i > 0$ that bounds P_0, there must also be spatial coherence between it and its adjacent levels. This implies that a feature on polyhedron P_i will have a corresponding feature on P_{i+1} and on P_{i-1} which are proximate in position and orientation. These are called the parent and child features respectively.

4.3 Construction

The levels of the hierarchy are constructed in order starting with P_1. In particular, when constructing P_i, P_{i-1} provides the topological and geometric information required to reduce the number of features while P_0 provides geometric information to satisfy the bounding criterion. The process of constructing a level in the hierarchy is given by the following steps to create P_i:

1. Create P_i as a high quality simplification of P_{i-1}.
2. Make P_i convex by computing its convex hull.
3. Compute the center of mass of P_i and translate P_i such that its center of mass coincides with P_{i-1}'s center of mass.
4. Scale P_i from its center of mass so that it barely bounds P_0.
5. Compute the maximum deviation ϵ_i of P_i from P_0. This is the same as computing the Hausdorff distance between P_i and P_0.
6. Assign the feature correspondences from P_i to P_{i-1} and from P_{i-1} to P_i.

For the first step, any simplification algorithm may be used as long as the topology is maintained. Any convex hull algorithm may be used for the second step.

The computation of the center of mass is straightforward for a closed polyhedron. To scale P_i in Step 4, the maximum scaling required over all of the faces of P_i is found and applied. The scaling required for a face is computed by finding the extremal vertex on P_0 in the direction of the (outward pointing) face normal and computing the scaling factor required to cause the vertex to coincide with the scaled face's supporting plane.

The Hausdorff distance (ϵ_i) can be computed in this context by computing the maximum deviation over all the vertices of P_i. The deviation of a vertex of P_i is computed by computing its distance from P_0. The vertices are the only points on P_i that have to be checked because they represent local maxima of the deviation function over P_i.

To assign children (features of P_{i-1}) to the features of P_i, the nearest feature of P_{i-1} is found for each vertex of P_i and the nearest vertex is selected from this nearest feature. Each neighbor of each vertex of P_i (including the vertex) is assigned the nearest vertex as its child feature. To assign parents (features of P_i) to the features of P_{i-1}, the extremal vertex of P_i is found for each face of P_{i-1}. Each face as well as its edges and vertices are assigned the extremal vertex as their parent. These feature correspondence schemes were chosen for their spatial coherence. Of course, there are other schemes that may work better but it is unlikely that a sizeable overall improvement would be gained.

4.4 Dobkin-Kirkpatrick Hierarchy

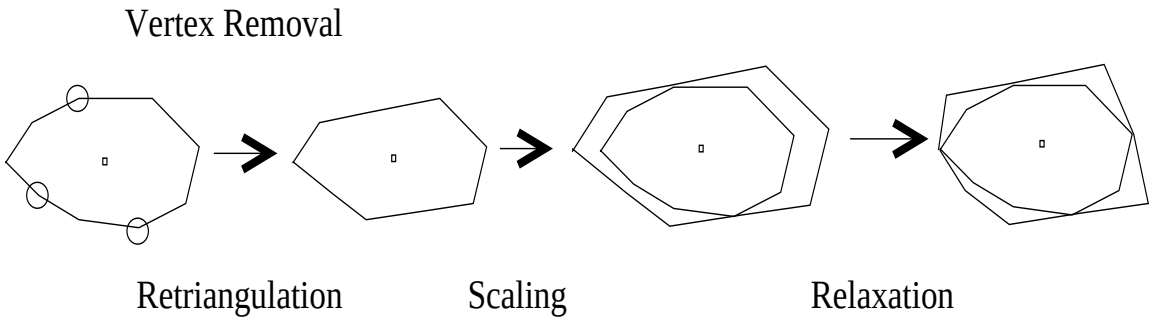


Figure 1. Dobkin-Kirkpatrick Hierarchy Construction

The first method that we implemented is based on the Dobkin-Kirkpatrick hierarchy [4]. It is a specialized algorithm that is the same as the steps given above but with a different scaling procedure and different parent and child feature assignments. An illustration of the process is shown in Figure 1.

4.4.1 Simplifying

The Dobkin-Kirkpatrick simplification does not specify which vertices should be removed but simply that an independent set is be removed at each step. If *any* independent set is removed, very large bounding volumes (Step 4) can result. Therefore, we try to choose the independent set in an intelligent manner. This is done by assigning an importance value to each vertex. A higher importance means that the vertex is more important and should not be removed. It also indicates that the geometry is not very flat near the vertex. Vertex v_{ij} of object P_i is assigned an importance value

$$I_{ij} = \sum_{k=1}^{V} (1 - \cos\theta_{ijk})$$

where V is the valence of the vertex and θ_{ijk} is the dihedral angle of the kth edge of the vertex. Thus, to create the independent set, vertices are added to it in order of increasing importance. As each vertex is added to the set, its neighbors are marked and not allowed to be subsequently added, otherwise the independent set criterion would be violated. These vertices are removed and the holes are re-triangulated in a convex manner. This can be done by taking the outer convex hull of the vertices neighboring the removed vertex. Levels are created until a tetrahedron is reached, until a minimum number of triangles is reached, or until the center of mass of the original object falls outside of the simplified object we are trying to create.

Even with this scheme, we found that excessively large bounding volumes occurred (Step 4). The reason stems from the choice of vertices that are placed in the independent set. We found that if we add as many vertices as possible to the independent set, we still add vertices with high importance because most of the low importance vertices have neighbors that are added. This may cause the volumes to become large when vertices which have sharp peaks are removed because the faces that fill a hole will have to be scaled by a large amount in order to bound a high importance (sharp) vertex. To reduce the effects of this problem we decided to only add vertices to the independent set that meet the criterion $I_{ij} \leq 1$. In other words, we stop the independent set creation when there are no more vertices whose importance values are less than or equal to one. This causes fewer vertices to be removed per level but keeps the sizes of the bounding polyhedra in check.

4.4.2 Bounding

The next step we discuss is Step 4 which involves scaling the sequence of polyhedra so that they bound the finest object. A scaling factor is computed for each P_i for $i > 0$ such that P_i bounds the finest polyhedron. Each level is then scaled by the amount computed for it. The scaling is done from the center of mass of the finest polyhedron.

We found that due to the structure of the simplification, we can add a relaxation process at this point. The relaxation involves trying to move the vertices back toward the center of mass while maintaining convexity and boundedness of P_0. Note that in both the scaling and the relaxation, the vertices move along rays from the center of mass through their original positions. The holes caused by vertex removal form regions which we call *center of mass regions*. By keeping track of the vertices that belong to certain center of mass regions, the scaling and relaxation computations can be performed more efficiently. Then, the maximum surface deviation is computed as ϵ_i.

4.4.3 Assigning Feature Correspondence

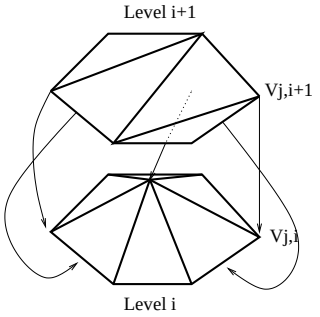


Figure 2. Dobkin-Kirkpatrick Level Correspondence

Each vertex, edge, and face on P_i for $i > 0$ is assigned a child pointer which points to a feature on P_{i-1}. The assignment can be done in a variety of ways. Figure 2 shows some of the pointers based on our assignment scheme. We assign the vertices at the coarser level to have as children their corresponding vertices at the finer level. Edges and faces at the coarser level can be of two varieties: *kept* or *removed*. Kept features are ones that are not destroyed by the vertex removals. They include edges and faces that have none of their vertices in the independent set. Removed edges and faces are ones that have one of their vertices in the independent set. They are replaced with *new* features at the coarser level. The kept features are assigned children that correspond to themselves at the finer level. For simplicity, we choose

to assign to the new features, the vertex that was removed from the finer level in order to create them. There are other schemes, which may perform better, that can be used to assign the children in this latter case. This assignment is actually done during the building of the hierarchy.

4.5 QSlim Hierarchy

The drawback of the Dobkin-Kirkpatrick hierarchy is that many levels are created which causes a slow-down for the query algorithm. The problem is that the decimation rate is not high enough. That lead us to consider this type of hierarchy which relies on a more general simplification algorithm which is able to achieve high decimation rates (arbitrary) while at the same time maintaining small bounding volumes.

The second method that we implemented is based on the publicly available QSlim package. We used the QSlim system which is an implementation of Garland and Heckbert's quadric error metric simplification algorithm [8] for the first step of the hierarchy creation process. It allows an arbitrary face target for each step of simplification allowing us to choose the decimation rate. We use the term "decimation rate" to mean the fraction of triangles left when a coarser level is created from a finer one. For example, if a level has 1000 triangles and a decimation rate of 0.25 is applied, then the newly created (coarser) level has 250 triangles. For the second step, we used the QHull convex hull library that is also publicly available. All the other steps remain the same.

There is a triangle count cutoff that determines when to stop building the hierarchy. In addition, there is a constant factor that determines what the decimation rate is across levels. These parameters affect the query performance and are discussed later.

The quadric error metric method employed by QSlim is very desirable for keeping the volumes of the bounding polyhedra small since only flat areas on a fine model are refined. This also has the effect of causing the tesselation density per solid angle of the simplified convex polyhedra to become more or less constant.

4.6 Directional Lookup Table

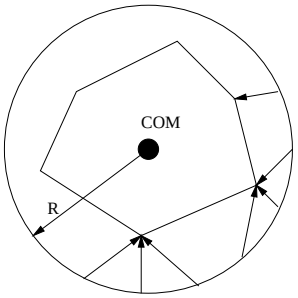


Figure 3. Lookup Table Construction

For performance comparison, we also designed and implemented a directional lookup table scheme for initializing the Voronoi marching. The lookup table is based upon direction from the center of mass (COM) of a polyhedron. Vertices are stored in the table. Figure 3 shows the idea of the construction in two dimensions.

The parameters used to construct the table are resolution and a bounding sphere radius factor. The resolution of the lookup table determines how close together the samples are. The table is based on spherical coordinates and has an entry for each angle increment (determined by the resolution) in the

altitude and azimuth directions excluding the north and south poles. There are two additional entries for the poles. The polyhedron for which the table is built has a "radius" which is defined as the maximum distance of any of its vertices from its center of mass. In other words, a sphere centered at the center of mass with radius equal to the polyhedron's radius would bound the polyhedron at any orientation. The bounding sphere radius factor is multiplied by the polyhedron's radius to construct another (larger) bounding sphere. The radius factor must be greater than or equal to 1. Sample points are generated on the new bounding sphere in the directions prescribed by the lookup table. For each of these points, the nearest vertex on the polyhedron to the point is stored at the corresponding location in the table. Using this table for query purposes is discussed in Section 5.3.

5 Proximity Query

Proximity queries can be in the form of intersection detection, error-bounded approximate distance computation, exact distance computation, or contact determination. The hierarchy is queried in much the same way for all four of these query types. The basis is a multiresolution extension on the algorithm proposed by Lin and Canny [16]. First we describe the marching within a level and then show how to answer a query by marching across levels of a multiresolution hierarchy for each of the query types.

5.1 Voronoi Marching Within a Level

We have implemented an algorithm which marches across the surface of a convex polyhedron using Voronoi regions. It is basically the same algorithm as the original *Lin-Canny* [16] algorithm and the improved version, *V-Clip* [19], in the sense that it has the same high level behavior and computes the same results. We do not give a full description here but rather a quick sketch.

The original algorithm is based on traversing the external Voronoi regions induced by the features of each convex polyhedron. The invariant is that at each step, either the inter-feature distance is reduced or the dimensionality of one or both of the features decreases by one, i.e. a move from a face to an edge or from an edge to a vertex. We employ the notion of states with the same definition as in *V-Clip*. The state transitions are a bit different because we sometimes change features on both objects in one step. For example, this happens from the edge-edge state to the vertex-vertex state. Traditionally, the feature whose Voronoi plane is found to be violated *first* is the one that is updated to. There are other methods for updating to a closer feature. For instance, we found that performance improves when we update to the feature whose plane is *most* violated which means that we have to check all the planes. This gives a higher cost to each local decision but yields a global benefit. We are currently investigating benefits from better choices of the next feature and other low level optimizations. A comparison of our low level marching implementation is compared against *V-Clip* in Section 6.

5.2 Multi-Level Voronoi Marching

In the case of intersection detection, the search normally starts at a pair of features related to the closest features from the previous query so that coherence may be exploited. Like in Lin-Canny, the distance between two convex polyhedra is minimized by marching across their surfaces. If an intersection is detected, a finer level of the hierarchy is traversed to. If at *any* level, we reach a minimum in the distance function and the objects are disjoint, then this is reported. With no additional cost, an approximate distance can be provided as δ with the error ϵ, when the objects are disjoint. The distance between the

levels of the two objects is δ and $\epsilon = \epsilon_i + \epsilon_j$ is the associated error equal to the sum of the two level deviations. The pair of closest features are used to find the features to be used for the next query. This is done by traversing their parent pointers, using the deviations ϵ_i, and keeping track of the distance δ that is decreased by the differences in deviation. If an intersection has been detected and P_0 has been reached for both objects, then intersection is reported. The pair of intersecting features is stored for the next query.

For error-bounded approximate distance computation, the application can provide a distance tolerance δ_{max} and an error tolerance ϵ_{max}. If the levels of the two objects intersect all the way to the finest levels of both objects, then intersection is reported and the pair of features is stored. Otherwise, two levels were found to be disjoint during the refinement. The disjoint features are used as in the intersection case to determine the initial features to be used for the next query. If at any point $\delta > \delta_{max}$ the query is terminated and nothing is reported. If at any point, $\epsilon \leq \epsilon_{max}$ then the error tolerance has been met and δ and ϵ are reported. This means that the exact distance is in the range $[\delta, \delta + \epsilon]$.

To compute exact distance, the same method is followed but only the distance tolerance δ_{max} is specified. As long as $\delta \leq \delta_{max}$, the distance is refined until the finest levels are reached at which point δ is reported. Finally, contact determination (closest points) queries can be handled the same as exact distance ones.

For the Dobkin-Kirkpatrick hierarchy, we found that there are many levels created. We tried three different refinement methods: refine one level on one object, refine one level on both objects, and refine two levels on both objects. These are called "Single", "Single-Single", and "Double-Double" respectively in Section 6 where we compare the performance of each of the methods.

5.3 Directional Lookup Table

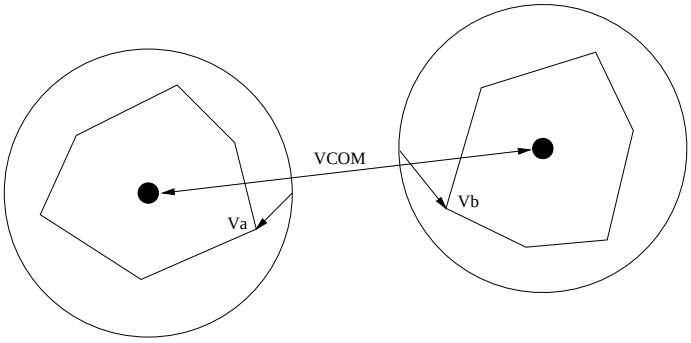


Figure 4. Lookup Table Initialization of a Query

When using the directional lookup table to help answer a query, the features from the previous query are not used. Therefore, the query time should be motion independent. Shown in Figure 4 is a two dimensional representation of the initialization of a query. First the vector defined by the center of mass of the two objects being tested (VCOM) is computed. The vector is transformed to the local coordinate frame of the objects and is used to find the table location that is the nearest to the direction given by the vector. The vertices at the corresponding locations in each object's table (V_a and V_b) are used as the feature pair to start marching from. When the lookup table is used, there is no hierarchy being used. The performance of the query depends solely on how the tables are built for each object. In the next section we see how various settings affect the performance of the lookup tables.

6 Experimental Results

The models used to test our query algorithm were empirically created by taking the convex hull of randomly sampled points on objects of various aspect ratio. Three different ellipsoidal models were created, each with 2000 triangles. Their aspect ratios are (1,1,1), (1,1,0.1), and (1,0.1,0.1). They are referred to as "fat", "plate", and "long" respectively. Each of the models was normalized to have a bounding sphere of radius r_0.

Following the performance benchmarking algorithm due to Mirtich [19] and used in [11], we created various scenarios. The algorithm has one object remain fixed while another orbits about it in an elliptical trajectory. An orbit radius is defined as the distance between the centers of the models when they are closest in the orbit. Three different pair combinations were used: fat-fat, plate-plate, and long-long. The orbit radius was set to be either $2r_0$ (small) or $3r_0$ (large). Thus, the models either just touch or are always separated by a distance of at least r_0.

Our implementation is called SWIFT (Speedy Walking via Improved Feature Testing). We obtained empirical observations for it by running programs on an SGI Reality Monster using a single 300 MHz MIPS R12000 CPU. Timings were done by using a highly accurate free-running hardware clock.

6.1 Marching Within a Level

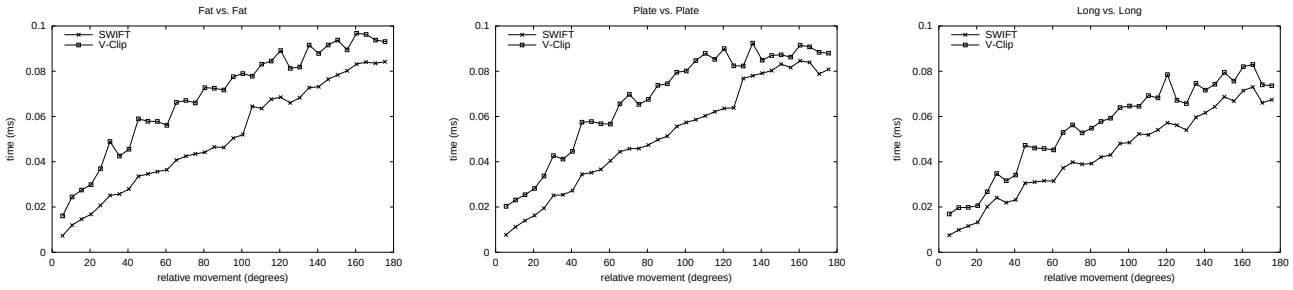


Figure 5. Performance of Marching Within a Level: *SWIFT* vs. *V-Clip*

Our implementation of the algorithm that marches within a level was compared to *V-Clip* [19] and found to be faster[2]. Figure 5 shows timings for different object pairs using the orbiting algorithm with an orbit radius of $2r_0$. No hierarchical or lookup table enhancements were used in the comparison. All future experimental results that are presented are based solely on the SWIFT implementation. Next, we add the directional lookup table and observe its performance.

6.2 Directional Lookup Table

In Figure 6 we show the results of using a directional lookup table (LUT) with an orbit radius of $2r_0$. The first parameter given in the graph legends is the bounding sphere radius factor of the lookup table and the second is the resolution in degrees. For the fat pair of objects, the resolution of the table is the only thing that matters. Furthermore, notice that the performance of the lookup table for the fat pair is much better than for the other two pairs because it (the LUT) is based on a sphere and the fat objects are very

[2]In the preliminary version of this report, available in the Electronic Proceedings of IEEE IROS 2000, we stated that our implementation was nearly twice as fast. However, we have since gathered more data and found that to be an inaccurate assessment.

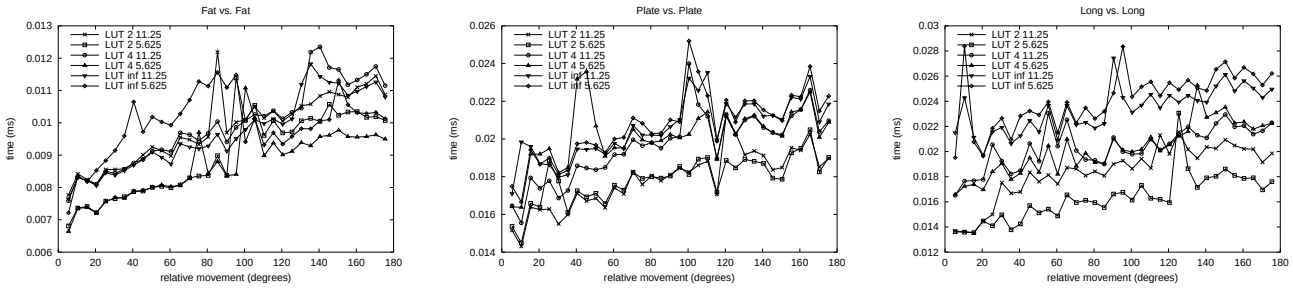


Figure 6. Performance of Various Lookup Tables

nearly spheres. For the long and plate pairs, a radius factor of 2 seems to be best. It should be stated that this may be because of the scenario we are using. The sizes of the lookup tables of various resolutions are: 8 KB for 5.625 degrees and 2 KB for 11.25 degrees. We will use the "LUT 2 5.625" performance data for the overall comparison.

6.3 Multiresolution Hierarchy

The level-of-detail hierarchy we have implemented can also be used to achieve speedups for both intersection detection and exact distance computation. The titles on the graphs in the following figures reflect the object pairs that were used as well as whether the orbit was small or large.

6.3.1 Dobkin-Kirpatrick Hierarchy

Shown in Figure 7 is the performance of various intersection detection queries using the Dobkin-Kirkpatrick hierarchy. Similarly, Figure 8 shows exact distance computation performance. In each entry of the legends of the graphs is the number of triangles that the hierarchy was cutoff at, the number of levels created (in parentheses) and the inter-level marching method used (as introduced in Section 5).

There is not a lot of difference among the various settings for intersection detection since most of the time is spent traversing the coarsest level. However, the settings which produce a coarser coarsest level do the best. Since all the levels are traversed most of the time in the case of exact distance, it makes sense the the methods to do the best are the ones that employ double descent on the levels of both objects. We will use the "DK Hier 100 Double-Double" performance data for the overall comparison.

6.3.2 QSlim Hierarchy

Shown in Figure 9 is the performance of various intersection detection queries using the QSlim hierarchy. Similarly, Figure 10 shows exact distance computation performance. In each entry of the legends of the graphs is the decimation rate (Section 4.5) and the number of levels created (in parentheses).

For the intersection detection query, the 0.125 hierarchy is anomalous because its coarsest level consists of 250 triangles while the other hierarchies have fewer. This results in a slower query because for intersection detection, most of the time is spent traversing the coarsest level. The coarsest hierarchy does the best on the intersection detection tests because of the reason previously stated. For the exact distance tests, it seems that having more levels is not as good as just having two. Even having three is noticeably slower. Also, it still seems like a coarser hierarchy does better for this type of query as well. We will use the "QS Hier 0.03125" performance data for the overall comparison.

6.4 Overall Performance Comparison

Figure 11 gives the overall performance comparison of intersection detection using the best results from each category along with the performance of SWIFT when there is no hierarchy or lookup table (Nothing). Similarly, Figure 12 gives the overall performance comparison for exact distance computation.

For intersection detection, the hierarchies do well because of their simple coarsest levels but they do not perform as well when it comes to exact distance computation because of the overhead associated with marching on the finest level to the true minimum of the distance function. The QSlim hierarchy always outperforms the Dobkin-Kirkpatrick hierarchy because there are too many levels to traverse in the latter.

6.5 Summary

The reason for the difference in the performance of the two query types is due to the fact that in exact distance computation, we cannot stop once the models are found to be disjoint like we can in intersection detection. Approximate distance computation is highly dependent upon the application so we did not evaluate its performance but expect the majority of cases in which it is used to resemble the intersection detection query. It can be shown that its cost lies in between the cost of intersection detection and exact distance computation.

It can be seen from the profiles of our timing curves that our multi-level Voronoi marching algorithm is able to keep the impact of the relative motion of the objects under control. It is also apparent that a simple lookup table scheme like the one we have presented suffices to perform in the same kind of manner. It is possible that if the objects are composed of more triangles (10,000 or 100,000), then a hierarchy of three or four levels may win out over the simple lookup table and the two level hierarchies.

7 Conclusion

We have presented a new algorithm for accelerating proximity queries between convex polyhedra using level-of-detail representations. We described the construction of the required hierarchical data structures and discussed various design issues involved. The runtime algorithm along with its issues were addressed and the performance presented.

Our implementation has been shown to have good performance. The source code is available to the public and provides a complete and entire proximity query library, with the two types of hierarchies presented. It is available at

http://www.cs.unc.edu/∼geom/SWIFT/

We have shown that a hierarchical representation integrated with Voronoi marching offers a progressive refinement framework for exact and approximate distance computation, which optimizes performance while meeting the application's need for bounded error computation. We have also shown that a simple directional lookup table can be used very effectively.

This research is the first step toward the design of distance query algorithms using multiresolution representations. There are many potential directions for future research. There is a need to develop a suitable metric for measuring or quantifying the optimality of the hierarchies, coherence levels, and level relationships. On the theoretical side, there is the interesting problem of proving a better bound on the computation of the distance between two convex polyhedra. The extension of this framework to non-convex objects and deformable models remains a very challenging problem.

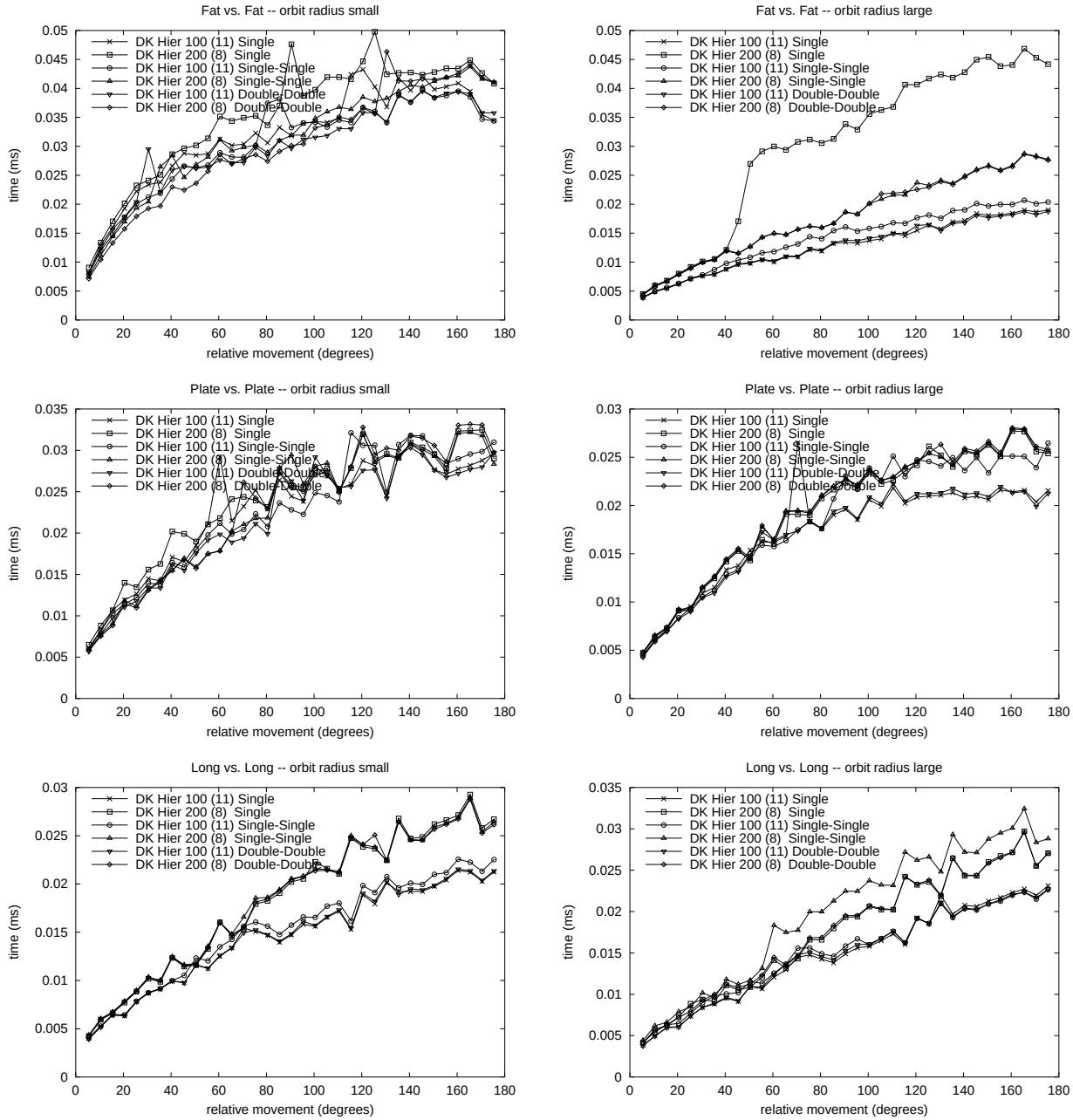


Figure 7. Performance of the Dobkin-Kirpatrick Hierarchy for Intersection Detection with Various Parameter Settings

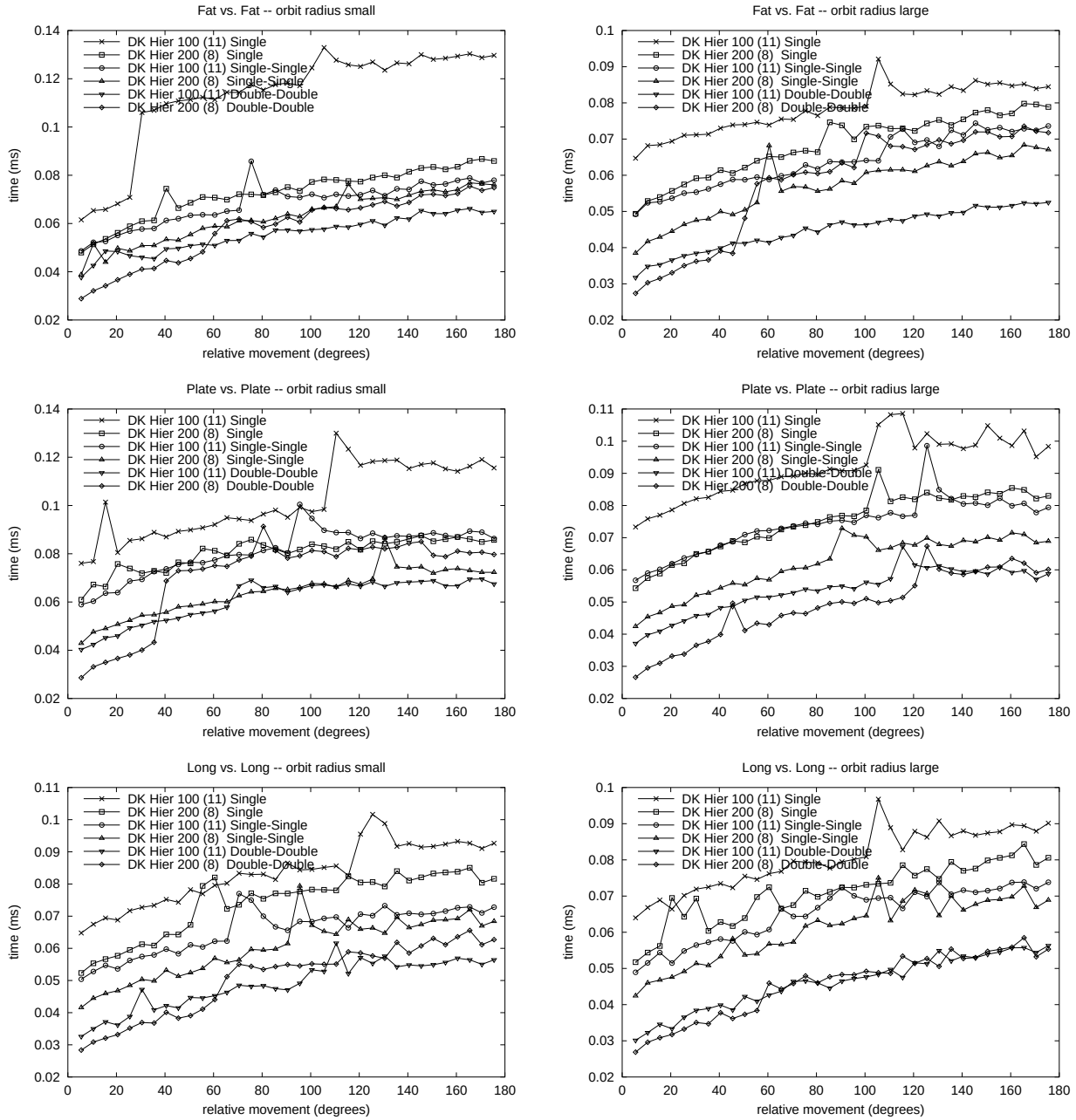


Figure 8. Performance of the Dobkin-Kirpatrick Hierarchy for Exact Distance Computation with Various Parameter Settings

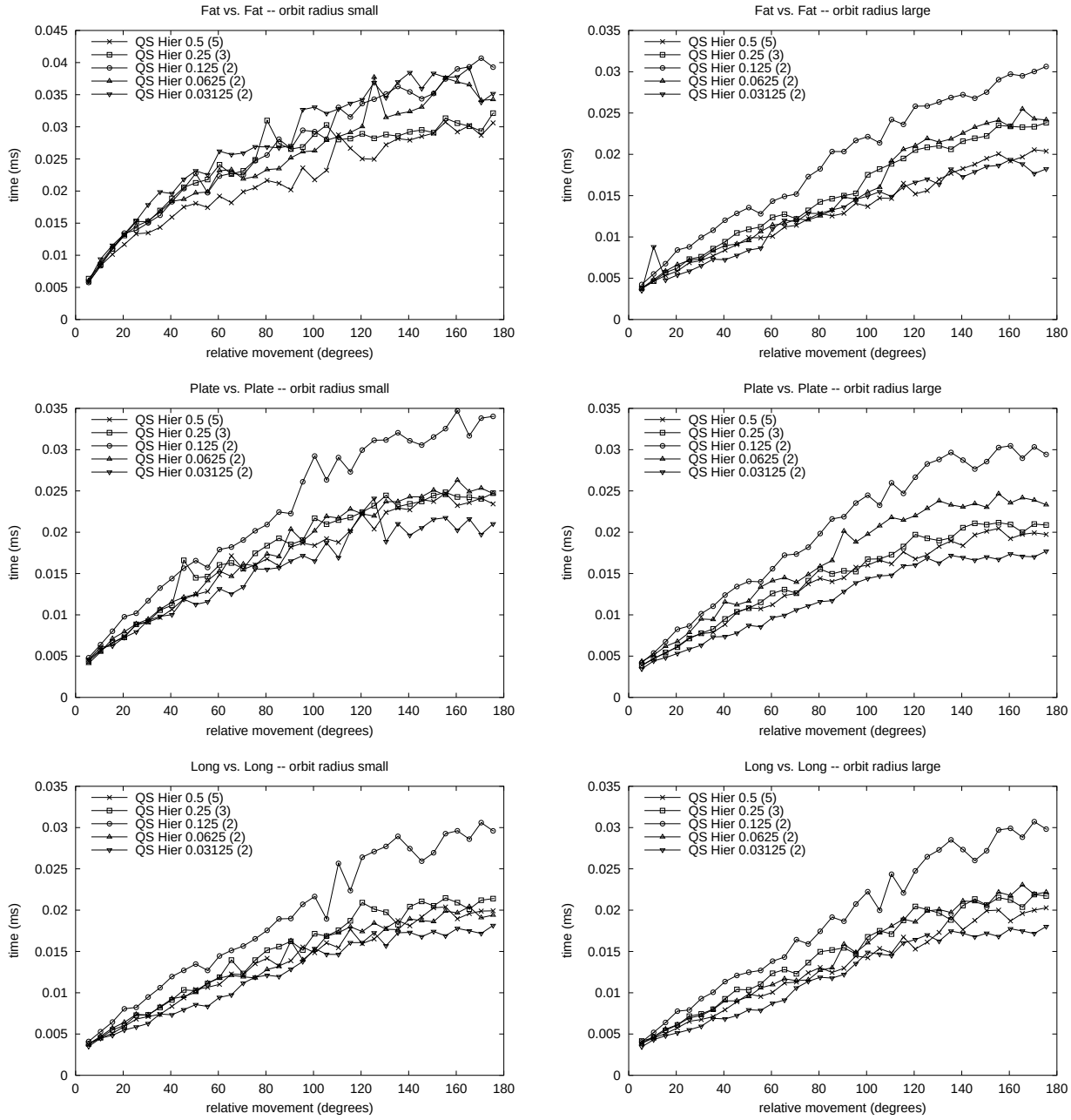


Figure 9. Performance of the QSlim Hierarchy for Intersection Detection with Various Parameter Settings

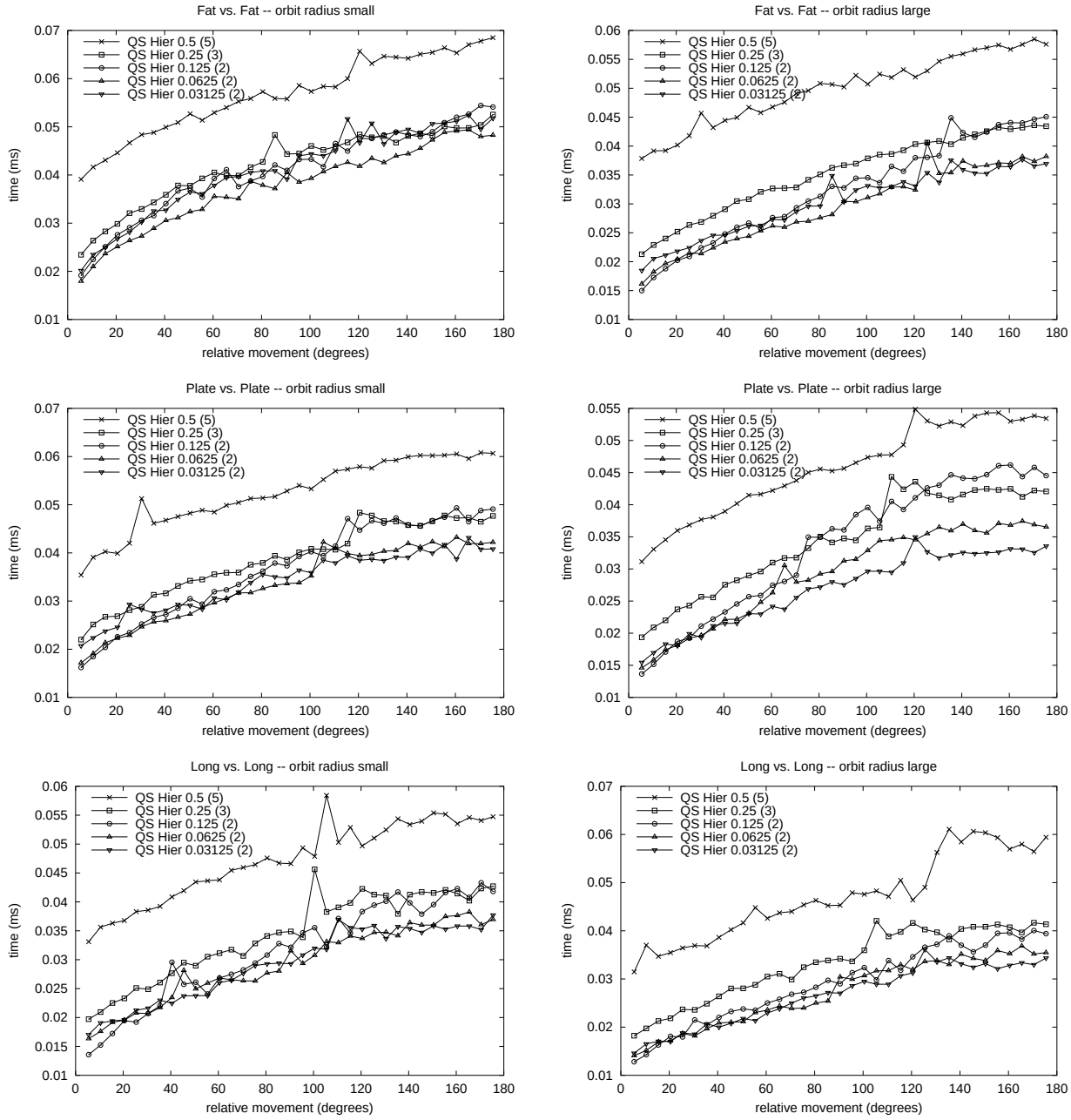


Figure 10. Performance of the QSlim Hierarchy for Exact Distance Computation with Various Parameter Settings

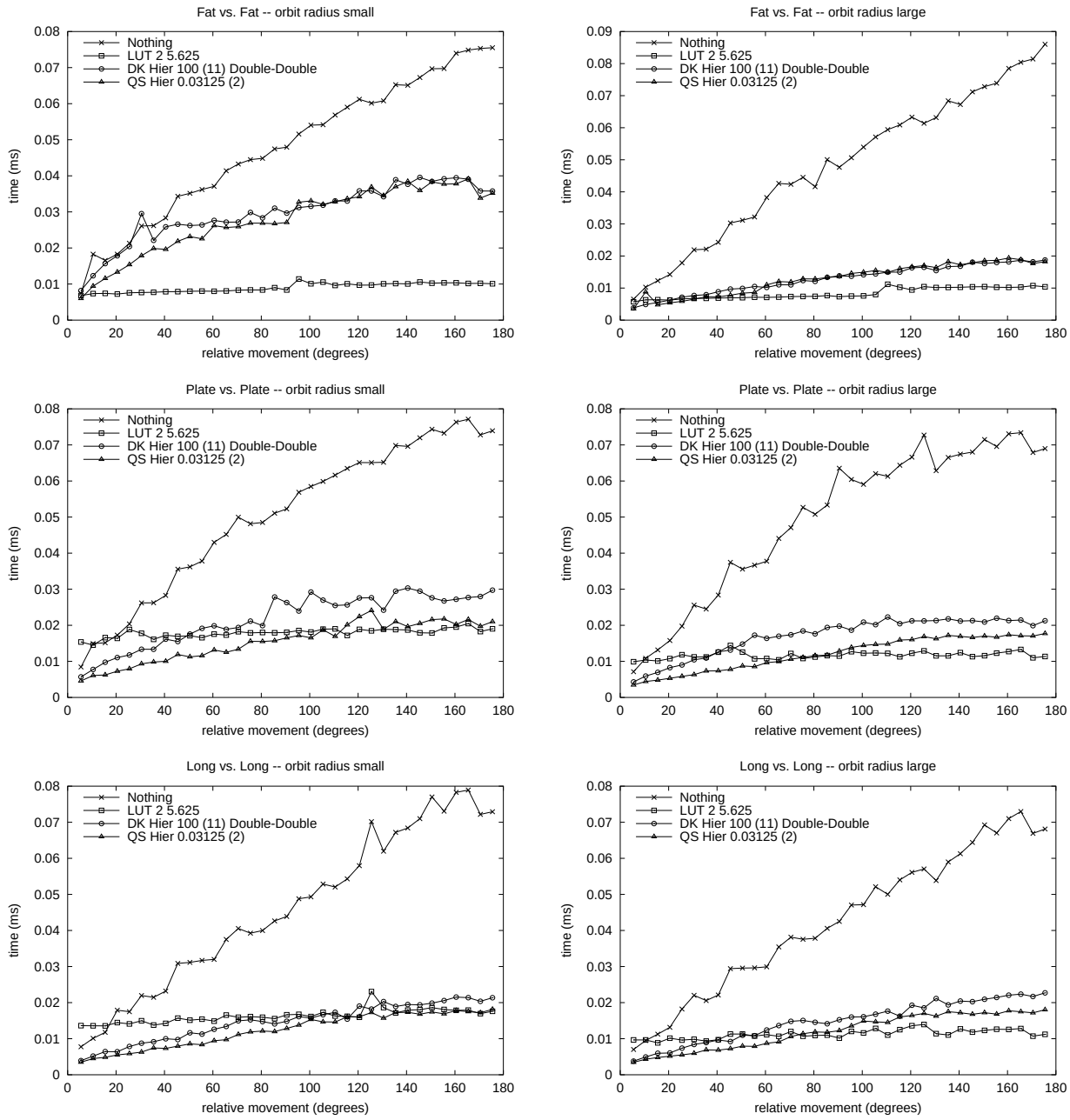


Figure 11. Performance of Best Methods for Intersection Detection

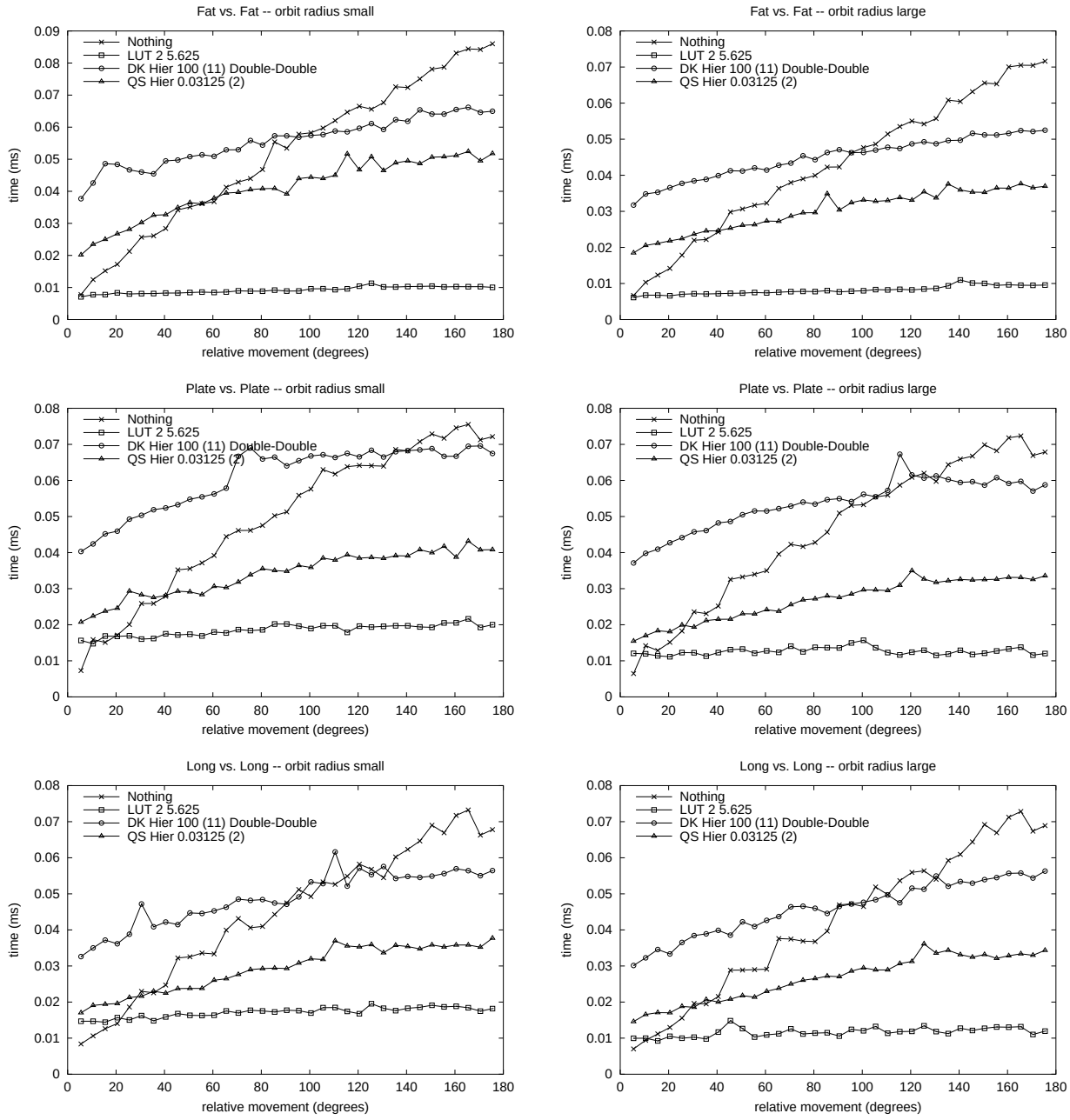


Figure 12. Performance of Best Methods for Exact Distance Computation

References

[1] S. Cameron. Approximation hierarchies and s-bounds. In *Proceedings. Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 129–137, Austin, TX, 1991.

[2] S. Cameron. Enhancing GJK: Computing minimum and penetration distance between convex polyhedra. *Proceedings of International Conference on Robotics and Automation*, pages 3112–3117, 1997.

[3] J. Cohen. Appearance-Preserving Simplification of Polygonal Models. *Ph.D. Dissertation*. University of North Carolina at Chapel Hill. 1998.

[4] 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.

[5] P. Dworkin and D. Zeltzer. A new model for efficient dynamics simulation. *Proceedings Eurographics workshop on animation and simulation*, pages 175–184, 1993.

[6] H. Edelsbrunner. Computing the extreme distances between two convex polygons. *J. Algorithms*, 6:213–224, 1985.

[7] J. Erikson, L. Guibas, J. Stolfi, and L. Zhang. Separation-sensitive collision detection for convex objects. *Proc. of SODA*, 1999.

[8] M. Garland and P. Heckbert. Surface simplification using quadric error bounds. *Proc. of ACM SIGGRAPH*, pages 209–216, 1997.

[9] 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.

[10] 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.

[11] L. Guibas, D. Hsu, and L. Zhang. *H-Walk*: Hierarchical distance computation for moving convex bodies. *Proc. of ACM Symposium on Computational Geometry*, 1999.

[12] G. Hirota, R. Maheshwari and M. Lin. Fast volume-preserving free-form deformation using multi-level optimization. *Proc. Symposium on Solid Modeling and Applications*, 1999.

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

[14] J. Klosowski, M. Held, Joseph S. B. Mitchell, K. Zikan, and H. Sowizral. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Trans. Visualizat. Comput. Graph.*, 4(1):21–36, 1998.

[15] 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.

[16] M.C. Lin and John F. Canny. Efficient algorithms for incremental distance computation. In *IEEE Conference on Robotics and Automation*, pages 1008–1014, 1991.

[17] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygon environments. In *Proc. of ACM SIGGRAPH*, 1997.

[18] N. Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. *SIAM J. Computing*, 12:pp. 759–776, 1983.

[19] Brian Mirtich. V-Clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 17(3):177–208, July 1998.

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

[21] H. Samet. *Spatial Data Structures: Quadtree, Octrees and Other Hierarchical Methods*. Addison Wesley, 1989.

[22] P. Schröder and D. Zorin. Subdivision for modeling and animation. *ACM SIGGRAPH Course Notes*, 1998.

[23] R. Seidel. Linear programming and convex hulls made easy. In *Proc. 6th Ann. ACM Conf. on Computational Geometry*, pages 211–215, Berkeley, California, 1990.

[24] E. Stollnitz, T. Derose, and D. Salesin. *Wavelets for Computer Graphics*. Morgan Kaufmann Publishers, 1996.

[25] Tiow-Seng Tan, Ket-Fah Chong, and Kok-Lim Low. Computing bounding volume hierarchies using model simplification. In *ACM Symposium on Interactive 3D Graphics*, pages 63–70, 1999.

The Gilbert-Johnson-Keerthi algorithm

Christer Ericson
Sony Computer Entertainment America

1 Introduction

One of the most effective methods for determining intersection between two polyhedra is an iterative algorithm due to Gilbert, Johnson, and Keerthi, commonly referred to as the *GJK algorithm* [Gilbert88]. As originally described, GJK is a simplex-based descent algorithm that, given two sets of vertices as inputs, finds the Euclidean distance (and closest points) between the convex hulls of these sets. In a generalized form, the GJK algorithm can also be applied to arbitrary convex point sets, not just polyhedra [Gilbert90]. While the examples presented here are in terms of polytopes, the described algorithm is the generalized version and it applies to non-polygonal convex sets in a straightforward way. For more detail on the GJK algorithm than presented here, please consult [Ericson] or [Bergen03].

2 Preliminaries

While the original presentation of the GJK algorithm is quite technical, neither the algorithm itself nor its implementation are very complicated in practice. However, understanding of the algorithm does require the introduction of some concepts from convex analysis and the theory of convex sets on which the algorithm relies.

An important point is that the GJK algorithm does not actually operate on the two input objects per se, but on the *Minkowski difference* between the objects. This transformation of the problem reduces the problem from finding the distance between two convex sets to that of finding the distance between the origin and a single convex set. The GJK algorithm searches the Minkowski difference object iteratively, a subvolume at a time, each such volume being a *simplex*. The Minkowski difference is not explicitly computed but sampled through a *support mapping* function on demand. The concepts of Minkowski

difference, simplices, and support mapping are described in more detail in the following three sections.

2.1 Minkowski sums and differences

Two operations on convex sets important for the understanding of the GJK algorithm are the *Minkowski sum* and the *Minkowski difference* of point sets. Let A and B be two point sets and let $\mathbf{a}$ and $\mathbf{b}$ be the position vectors corresponding to pairs of points in A and B. The Minkowski sum, $A \oplus B$ is then defined as the set:

$$A \oplus B = \{\mathbf{a} + \mathbf{b} : \mathbf{a} \in A, \mathbf{b} \in B\}$$

where $\mathbf{a} + \mathbf{b}$ is the vector sum of the position vectors $\mathbf{a}$ and $\mathbf{b}$. Visually the Minkowski sum can be seen as the region swept by A translated to every point in B (or vice versa). An illustration of the Minkowski sum is given in Figure 1.

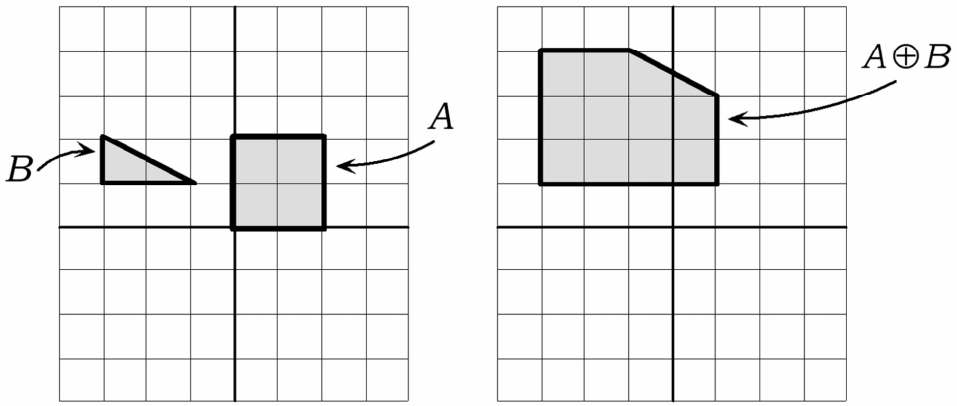


Figure 1: The Minkowski sum of a square A and a triangle B.

The Minkowski difference of two point sets A and B is defined analogously to the Minkowski sum:

$$A \ominus B = \{\mathbf{a} - \mathbf{b} : \mathbf{a} \in A, \mathbf{b} \in B\}$$

Geometrically, the Minkowski difference is obtained by adding A to the reflection of B about the origin, that is, $A \ominus B = A \oplus (-B)$ (Figure 2). For this reason, both terms are often simply referred to as the Minkowski sum. For two convex polygons, P and Q, the Minkowski sum $R = P \oplus Q$ has the properties that R is a convex polygon and the vertices of R are sums of vertices of P and Q. The Minkowski sum of two convex polyhedra is a convex polyhedron, with corresponding properties.

The Minkowski difference is important from a collision detection perspective as two point sets A and B collide (that is, have one or more points in common)

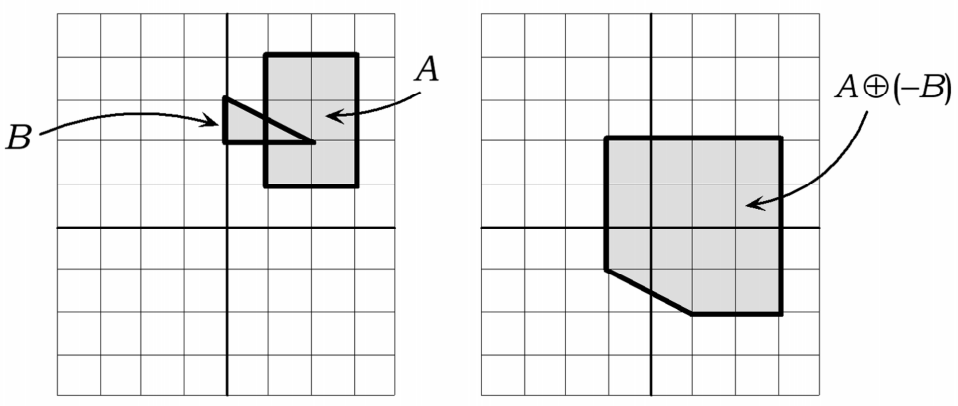


Figure 2: Since rectangle A and triangle B intersect, the origin must be contained in their Minkowski difference.

if and only if their Minkowski difference C, $C = A \ominus B$, contains the origin (c.f. Figure 2). In fact, it is possible to establish an even stronger result: computing the minimum distance between A and B is equivalent to computing the minimum distance between C and the origin. This follows since:

$$\begin{aligned} distance(A, B) &= \min\{\|\mathbf{a} - \mathbf{b}\| : \mathbf{a} \in A, \mathbf{b} \in B\} \\ &= \min\{\|\mathbf{c}\| : \mathbf{c} \in A \ominus B\} \end{aligned}$$

Note that the Minkowski difference of two convex sets is also a convex set, so its point of minimum norm is unique.

2.2 Simplices

A *d-simplex* is the convex hull of $d+1$ affinely independent points in d-dimensional space. A *simplex* (plural *simplices*) is a d-simplex, for some given d. For example, the 0-simplex is a point, the 1-simplex is a line segment, the 2-simplex is a triangle and the 3-simplex is a tetrahedron (Figure 3). A simplex has the property that removing a point from its defining set reduces the dimensionality of the simplex by one.

2.3 Supporting points and support mappings

For a general convex set C (thus not necessarily a polytope) a point from the set most distant along a given direction is called a *supporting point* of C. More specifically, P is a supporting point of C if, for a given direction $\mathbf{d}$ it holds that $\mathbf{d} \cdot P = \max\{\mathbf{d} \cdot V : V \in C\}$; that is, P is a point for which $\mathbf{d} \cdot P$ is maximal. Figure 4 illustrates the supporting points for two different convex

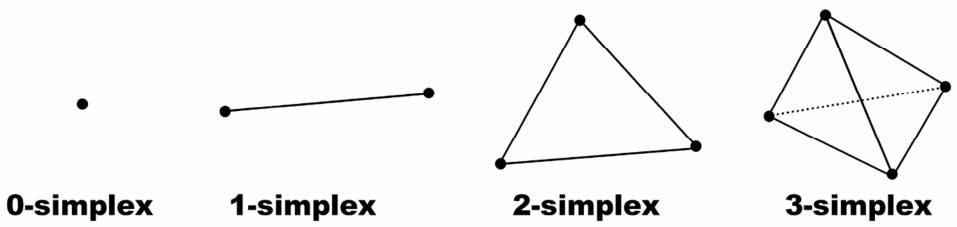


Figure 3: Simplices of dimension 0 through 3: a point, a line segment, a triangle, and a tetrahedron.

sets. Supporting points are sometimes called *extreme points*. They are not necessarily unique. For a polytope, one of its vertices can always be selected as a supporting point for a given direction. When a supporting point is a vertex, the point is commonly called a *supporting vertex*.

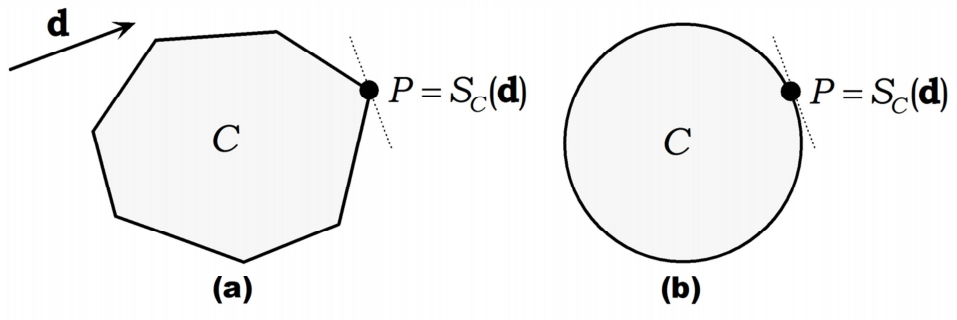


Figure 4: (a) A supporting vertex P of polygon C with respect to the direction $\mathbf{d}$. (b) A supporting point P of circle C with respect to the direction $\mathbf{d}$. In both cases, P is given by the support mapping function $S_C(\mathbf{d})$.

A *support mapping* is a function, $S_C(\mathbf{d})$, associated with a convex set C that maps the direction $\mathbf{d}$ into a supporting point of C. For simple convex shapes, such as spheres, boxes, cones, and cylinders, support mappings can be given in closed form. For example, for a sphere C centered at O and with a radius of r, the support mapping is given by $S_C(\mathbf{d}) = O + r\,\mathbf{d}/\,\|\mathbf{d}\|$ (c.f. Figure 4(b)). Convex shapes of higher complexity require the support mapping function to determine a supporting point using numerical methods.

For a polytope of n vertices, a supporting vertex is trivially found in $O(n)$ time by searching over all vertices. Assuming a data structure listing all adjacent vertex neighbors for each vertex, an extreme vertex can be found through a simple hill-climbing algorithm, greedily visiting more and more extreme vertices until no vertex more extreme can be found. This approach is very efficient as it only explores a small corridor of vertices as it moves towards the extreme vertex. For larger polyhedra, the hill-climbing can be sped up by adding one or more

artificial neighbors to the adjacency list for a vertex. Through precomputation of a hierarchical representation of the vertices, it is possible to locate a supporting point in $O(\log n)$ time. These acceleration schemes are described in more detail in [Ericson].

3 The Gilbert-Johnson-Keerthi algorithm

As mentioned earlier, the GJK algorithm effectively determines intersection between polyhedra by computing the Euclidean distance between them. The algorithm is based on the fact that the separation distance between two polyhedra A and B is equivalent to the shortest distance between their Minkowski difference C, $C = A \ominus B$, and the origin (Figure 5). Thus, the problem is reduced to that of finding the point on C closest to the origin. At the outset, this does not seem like much of an improvement as the Minkowski difference is non-trivial to compute explicitly. However, a key point of the GJK algorithm is that it does not explicitly compute the Minkowski difference C. It only samples the Minkowski difference point set using a support mapping of $C = A \ominus B$. Since the support mapping function is the maximum over a linear function, the support mapping for C, $s_{A\ominus B}(\mathbf{d})$, can be expressed in terms of the support mappings for A and B as $s_{A\ominus B}(\mathbf{d}) = s_A(\mathbf{d}) - s_B(-\mathbf{d})$. Thus, points from the Minkowski difference can be computed, on demand, from supporting points of the individual polyhedra A and B.

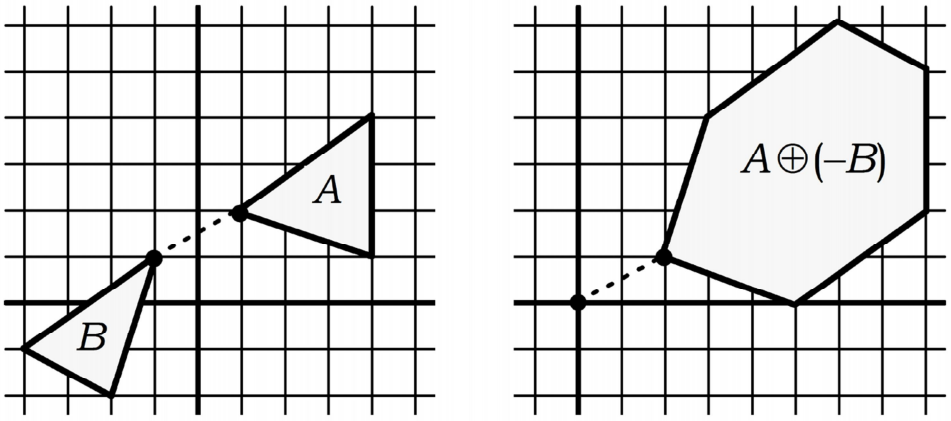


Figure 5: The distance between A and B is equivalent to the distance between their Minkowski difference and the origin.

To search the Minkowski difference C for the point closest the origin, the GJK algorithm utilizes a result known as *Carathéodory's theorem* [Rockafellar96]. The theorem says that for a convex body H of $\mathbf{R}^d$, each point of H can be expressed as the convex combination of no more than $d+1$ points from H. This allows the GJK algorithm to search the volume of the Minkowski difference C

by maintaining a set Q of up to $d+1$ points from C at each iteration. The convex hull of Q forms a simplex inside C. If the origin is contained in the current simplex, A and B are intersecting, and the algorithm stops. Otherwise, the set Q is updated to form a new simplex, guaranteed to contain points closer to the origin than the current simplex. Eventually this process must terminate with a Q containing the closest point to the origin. In the non-intersecting case, the smallest distance between A and B is realized by the point of minimum norm in $CH(Q)$ (the convex hull of Q). The following step-by-step description of the GJK algorithm specifies in more detail how the set Q is updated:

1. Initialize the simplex set Q to one or more points (up to $d+1$ points, where d is the dimension) from the Minkowski difference of A and B.
2. Compute the point P of minimum norm in $CH(Q)$.
3. If P is the origin itself, the origin is clearly contained in the Minkowski difference of A and B. Stop and return A and B as intersecting.
4. Reduce Q to the smallest subset Q' of Q such that $P \in CH(Q')$. That is, remove any points from Q not determining the subsimplex of Q in which P lies.
5. Let $V = s_{A\ominus B}(-P) = s_A(-P) - s_B(P)$ be a supporting point in direction $-P$.
6. If V is no more extremal in direction $-P$ than P itself, stop and return A and B as not intersecting. The length of the vector from the origin to P is the separation distance of A and B.
7. Add V to Q and go to 2.

Figure 6 illustrates how GJK iteratively finds the point on a polygon closest to the origin O for a two-dimensional problem. The algorithm arbitrarily starts with the vertex A as the initial simplex set Q, $Q = \{A\}$. For a single-vertex simplex, the vertex itself is the closest point to the origin. Searching for the supporting vertex in direction $-A$ results in B. B is added to the simplex set, giving $Q = \{A, B\}$. The point on $CH(Q)$ closest to the origin is C. Since both A and B are needed to express C as a convex combination, both are kept in Q. D is the supporting vertex in direction $-C$ and it is added to Q, giving $Q = \{A, B, D\}$. The closest point on $CH(Q)$ to the origin is now E. Since only B and D are needed to express E as a convex combination of vertices in Q, Q is updated to $Q = \{B, D\}$. The supporting vertex in direction $-E$ is F, which is added to Q. The point on $CH(Q)$ closest to the origin is now G. D and F is the smallest set of vertices in Q needed to express G as a convex combination, so Q is updated to $Q = \{D, F\}$. At this point, since no vertex is closer to the origin in direction $-G$ than G itself, G must be the closest point to the origin, and the algorithm terminates.

Note that the GJK algorithm trivially deals with spherically extended polyhedra, simply by comparing the computed distance between the inner polyhedral

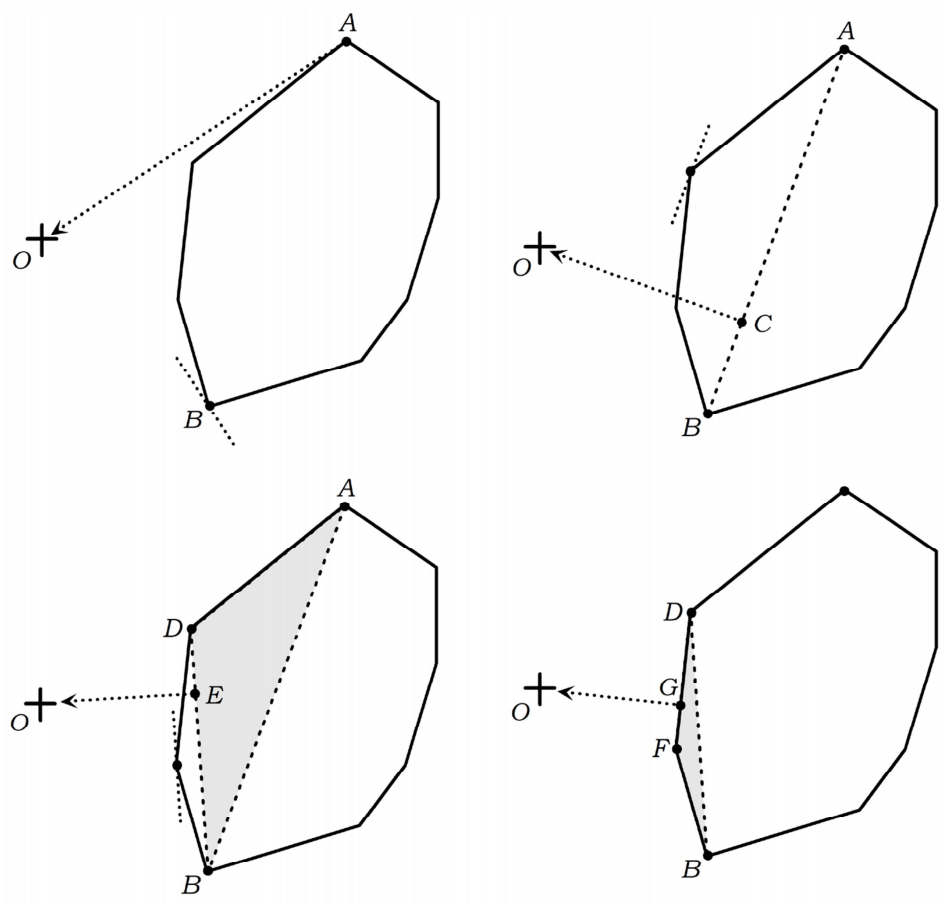


Figure 6: GJK finding the point on a polygon closest to the origin.

structures with the sum of the radii of the spherical extensions. Note also that if the GJK algorithm is applied to the vertex sets of nonconvex polyhedra, it will compute the smallest distance between the convex hulls of these nonconvex polyhedra.

While presented here as a method operating on polyhedra only, the GJK algorithm can, in fact, be applied to arbitrary convex bodies. Since the input bodies are only ever sampled through their support mappings, all that is required to allow the GJK algorithm to work with general convex objects is to supply appropriate support mappings.

The GJK algorithm terminates with the separation distance in a finite number of steps for polyhedra. However, it only asymptotically converges to the separation distance for arbitrary convex bodies. Therefore, a suitable tolerance should be added to the termination condition to allow the algorithm to terminate correctly when operating on non-polyhedral objects.

On termination with non-intersection, in addition to returning the separation distance, it is possible to have GJK compute the closest points between the input objects, computed from the points of objects A and B that formed the points in the last copy of simplex set Q. The steps of this computation are beyond the scope of these notes, see [Ericson] for full details.

3.1 Finding the point of minimum norm in a simplex

It remains to describe how to determine the point P of minimum norm in $CH(Q)$ for a simplex set $Q = \{Q_1, Q_2, \ldots, Q_k\}$, $1 \leq k \leq 4$. In the original presentation of GJK, this is done through a single procedure called the distance subalgorithm. The distance subalgorithm reduces the problem to considering all subsets of Q separately. For example, for $k = 4$ there are 15 subsets corresponding to 4 vertices (Q_1, Q_2, Q_3, Q_4), 6 edges (Q_1Q_2, Q_1Q_3, Q_1Q_4, Q_2Q_3, Q_2Q_4, Q_3Q_4), 4 faces ($Q_1Q_2Q_3$, $Q_1Q_2Q_4$, $Q_1Q_3Q_4$, $Q_2Q_3Q_4$), and the interior of the simplex ($Q_1Q_2Q_3Q_4$). The subsets are searched one by one, in order of increasing size. The search stops when the origin is contained in the *Voronoi region* of the feature (vertex, edge, or face) specified by the examined subset (or, for the subset corresponding to the interior of the simplex, when the origin is inside the simplex). Once a feature has been located, the point of minimum norm on this feature is given by the orthogonal projection of the origin onto the feature.

The Voronoi region for a feature F is the region of space containing points that lie closer to (or as close to) F than to any other feature. Figure 7 illustrates the Voronoi regions determined by the features of a triangle.

Consider again the case of $Q = \{Q_1, Q_2, Q_3, Q_4\}$. An arbitrary point P (specifically the origin) lies in the Voronoi region for, say, vertex Q_1 if and only if the following inequalities are satisfied:

$$\begin{aligned}(P - Q_1) \cdot (Q_2 - Q_1) &\leq 0 \\ (P - Q_1) \cdot (Q_3 - Q_1) &\leq 0 \\ (P - Q_1) \cdot (Q_4 - Q_1) &\leq 0\end{aligned}$$

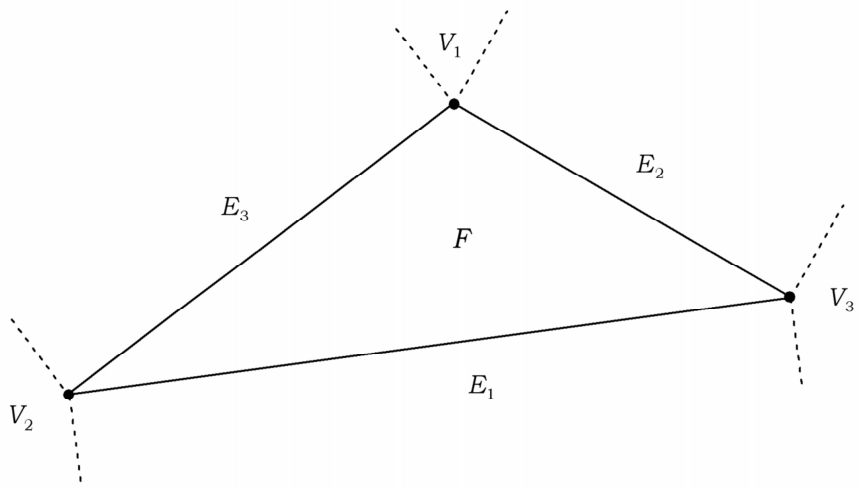


Figure 7: A triangle divides its supporting plane into 7 Voronoi feature regions: 1 face region (F), 3 edge regions (E_1, E_2, E_3), and 3 vertex regions (V_1, V_2, V_3).

P lies in the Voronoi region associated with edge Q_1Q_2 if and only if the following inequalities are satisfied:

$$\begin{gathered}(P - Q_1) \cdot (Q_2 - Q_1) \geq 0 \\ (P - Q_2) \cdot (Q_1 - Q_2) \geq 0 \\ (P - Q_1) \cdot ((Q_2 - Q_1) \times \mathbf{n}_{123}) \geq 0 \\ (P - Q_1) \cdot (\mathbf{n}_{142} \times (Q_2 - Q_1)) \geq 0\end{gathered}$$

where:

$$\begin{aligned}\mathbf{n}_{123} &= (Q_2 - Q_1) \times (Q_3 - Q_1) \\ \mathbf{n}_{142} &= (Q_4 - Q_1) \times (Q_2 - Q_1)\end{aligned}$$

If P does not lie in a vertex or edge Voronoi region, face regions are tested by checking if P and the remaining point from Q set lie on opposite sides of the plane through the three chosen points from Q to form the face. For example, P lies in the Voronoi region of $Q_1Q_2Q_3$ if and only if the following inequality holds:

$$((P - Q_1) \cdot \mathbf{n}_{123})((Q_4 - Q_1) \cdot \mathbf{n}_{123}) < 0$$

where again:

$$\mathbf{n}_{123} = (Q_2 - Q_1) \times (Q_3 - Q_1)$$

Analogous sets of inequalities can be defined for testing containment in the remaining vertex, edge, and face Voronoi regions. Note that most of the computed quantities are shared between different Voronoi region tests and need not be recomputed, resulting in an efficient test overall. Simplex sets of fewer than four points are handled in a corresponding way.

4 GJK for moving objects

While the GJK algorithm is usually presented and thought of as operating on two convex polyhedra, it is more general than so. Given two point sets, it computes the minimum distance vector between the convex hulls of the point sets (an easy way of seeing this is to note that the support mapping function never returns a point interior to the hull). This is an important distinction as it allows GJK to be used to determine collisions between convex objects under linear translational motion in a straightforward manner.

One approach to dealing with moving polyhedra is presented in [Xavier97]. Consider two polyhedra P and Q, with movements given by the vectors $\mathbf{t}_1$ and $\mathbf{t}_2$, respectively. To simplify the collision test, the problem is recast so Q is stationary. The relative movement of P (with respect to Q) is now given by $\mathbf{t} = \mathbf{t}_1 - \mathbf{t}_2$. Let V_i be the vertices of P in its initial position. $V_i + \mathbf{t}$ describes the location of the vertices of P at the end of its translational motion.

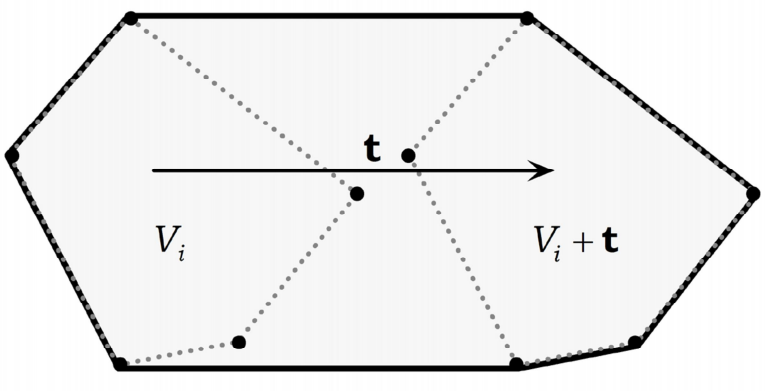


Figure 8: For a convex polyhedron under a translational movement $\mathbf{t}$, the convex hull of the vertices V_i at the start and the vertices $V_i + \mathbf{t}$ at the end of motion corresponds to the swept hull for the polyhedron.

It is not hard to see that as P moves from start to end over its range of motion, the volume swept out by P corresponds to the convex hull of its vertices at their initial and final positions (Figure 8). Determining if P collides with Q during its translational motion is therefore as simple as passing GJK the vertices of P at both start and end of P's motion (since this convex hull is what GJK effectively computes, given these two point sets). A drawback with this solution is that doubling the number of vertices for P increases the time to find a supporting vertex. A better approach, which does not suffer from this problem, is to consider the movement vector $\mathbf{t}$ of P with respect to $\mathbf{d}$, the vector for which an extreme vertex is sought. From the definition of the extreme vertex it is easy to see that when $\mathbf{t}$ is pointing away from $\mathbf{d}$, none of the vertices $V_i + \mathbf{t}$ can be more extreme than the vertices V_i. Thus, when $\mathbf{d} \cdot \mathbf{t} \leq 0$, an extreme vertex is guaranteed to be found amongst the vertices V_i (Figure 9(a)). Similarly, when

$\mathbf{d} \cdot \mathbf{t} > 0$, only the vertices $V_i + \mathbf{t}$ need to be considered for locating an extreme vertex (Figure 9(b)).

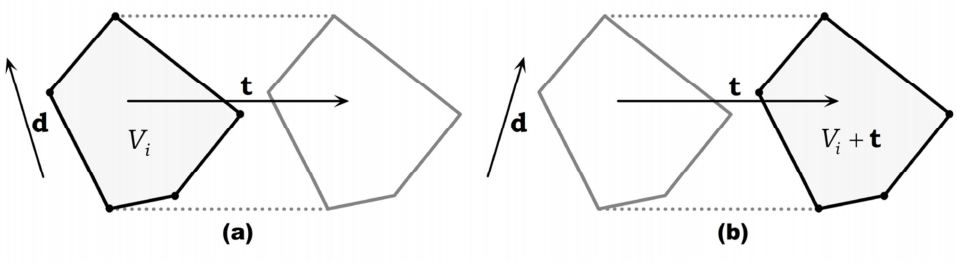


Figure 9: (a) When $\mathbf{d} \cdot \mathbf{t} \leq 0$, the supporting vertex is found amongst the original vertices V_i and the vertices $V_i + \mathbf{t}$ do not have to be tested. (b) Similarly, when $\mathbf{d} \cdot \mathbf{t} > 0$, only the vertices $V_i + \mathbf{t}$ have to be considered.

The second of two presented approaches is effectively implemented by changing the support mapping function such that the motion vector is added to the vertices during hill-climbing.

A drawback is that both presented methods only provide interference detection. However, interval halving can be effectively used to get the time of collision. Using an interval halving approach, the simplices from the previous iteration are ideal candidates for starting the next iteration. Alternatively, the time of collision can be obtained iteratively using a root finder such as Brent's method, as described in [Vlack01].

References

[Bergen03] van den Bergen, Gino. *Collision detection in interactive 3d environments.* Morgan Kaufmann Publishers, 2003.

[Ericson] Ericson, Christer. *Real-Time collision detection.* Morgan Kaufmann Publishers. Forthcoming.

[Gilbert88] Gilbert, Elmer. Daniel Johnson, S. Sathiya Keerthi. "A fast procedure for computing the distance between complex objects in three dimensional space." *IEEE Journal of Robotics and Automation*, vol.4, no. 2, pp. 193-203, 1988.

[Gilbert90] Gilbert, Elmer. Chek-Peng Foo. "Computing the Distance Between General Convex Objects in Three-Dimensional Space." *IEEE Transactions on Robotics and Automation*, vol. 6, no. 1, pp. 53-61, 1990.

[Rockafellar96] Rockafellar, R. Tyrell. *Convex Analysis.* Princeton University Press, 1996.

[Vlack01] Vlack, Kevin. Susumu Tachi. "Fast and accurate spacio-temporal intersection detection with the GJK algorithm." *Proceedings of the International Conference on Artificial Reality and Telexistence (ICAT2001)*, pp. 79-84, 2001. `http://vrsj.t.u-tokyo.ac.jp/ic-at/papers/01079.pdf`

[Xavier97] Xavier, Patrick. "Fast swept-volume distance for robust collision detection." *Proceedings of the 1997 IEEE International Conference on Robotics and Automation*, Albuquerque, NM, April 1997.

Continuous Collision Detection for Deforming Triangle Meshes

David Eberle
PDI/Dreamworks R&D

In this section the concept of continuous collision detection is revisited. An algorithm for performing continuous collision detection between deforming triangles is then presented. Implementation details and limitations of the algorithm are then discussed.

1 Introduction

Most of the collision detection strategies presented up to this point all examine the state of the geometry at discrete time intervals. When the objects are moving too fast with respect to each other either due to their relative speed and size, these algorithms can miss collision events entirely.

Some applications can afford to either ignore this problem or reduce the time step accordingly. Those that modify the time step can have issues with handling large numbers of objects. Using the technique of back tracking or bisecting the time interval typically involves the rolling back and forth the state of many objects to handle the collision events between a small number of objects. Other techniques let the object penetrate and then try to correct the penetration over future time steps. While the later approach scales better, the accuracy of the collision time and point of contact is questionable for both.

In recent years the alternative of handling collision detection in a continuous way has been gaining popularity [BFA02,PRO97]. These techniques do not suffer from the temporal aliasing problem and can report the exact time and location of contact. The idea is far from recent however and has been published in the graphics community more than a decade ago [MW88]. These methods were once considered too expensive for most applications and did not receive much attention in the past. Modern CPU speeds, further development, and a need for a more robust solution to the problems of collision detection and the problem of contact point determination have made these methods more attractive in recent years.

The remainder of this chapter will focus on the problem of continuous collision detection between deforming triangles. The algorithm is not novel but does require a lot of effort to achieve a robust implementation.

2 Vertex Motion

To perform continuous collision detection one must first characterize the motion of the geometry. Over a time interval t we assume that each vertex has a linear trajectory that can be described in terms of it's beginning and end positions. The limitations due to this assumption will be discussed later. Under this assumption one can write the following

equation for the motion of a point *p*. Here p_0 is the beginning position of the vertex and p_1 is the end position of the vertex.

$$p(t) = p_0 + t \cdot (p_1 - p_0) \quad t \subset [0,1] \qquad \textbf{E.1}$$

To determine the temporal collision between two triangles, six vertex-face tests and nine edge-edge tests must be performed.

2 Vertex-Face Test

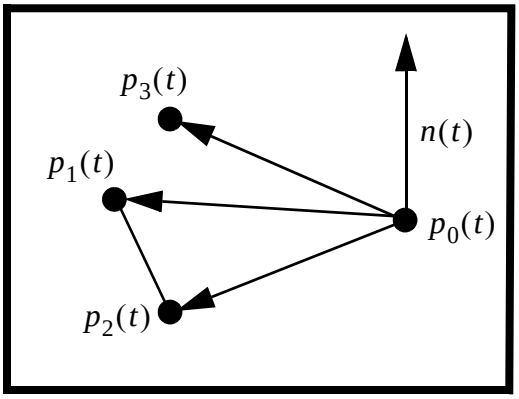


Diagram of the vertex-face test

To determine whether a vertex collides with a deforming triangle over the time interval, first check to see if it crosses the moving infinite plane that the triangle lies in. Each vertex has its motion described by **E.1**. The normal of the plane can be defined as a quadratic function in *t* by taking the cross product of the two triangle edges defined in terms of the point trajectories.

$$n(t) = (p_1(t) - p_0(t)) \times (p_2(t) - p_0(t)) \qquad \textbf{E.2}$$

When $p_3(t)$ crosses the infinite moving plane the vector between $p_3(t)$ and any point on the triangle will be perpendicular to $n(t)$. Then arbitrarily select $p_0(t)$ as the point on the triangle and arrive at the following the cubic equation in *t*.

$$n(t) \cdot (p_3(t) - p_0(t)) = 0 \qquad \textbf{E.3}$$

Cubic equations have an analytical solution which provides an advantage over the technique in [MW88], which requires a numerical root solver. However the algorithm work is not finished with this approach. At this point only the times that the point crosses the infinite plane are known. For each time $t \subset [0,1]$ reported from the root solver, one must compute the positions of the points and test whether the vertex is inside the boundary of the triangle. A number of techniques are available to perform this test and it is left to the reader to decide which is most efficient for their purposes.

3 Edge-Edge Test

The edge-edge test is derived in a similar fashion. Define the edges in terms of the vertex trajectories and take their cross product to express the normal as a quadratic in t. Then define another vector between two vertices, one from each edge and take the dot product of this vector with the equation for the normal to arrive at another cubic equation in t. Once again for each time $t \subset [0,1]$ reported from the root solver, test to see if the point of intersection at the given time between the infinite lines is within the boundary of both line segments. Once again the choice of this test is left to the reader.

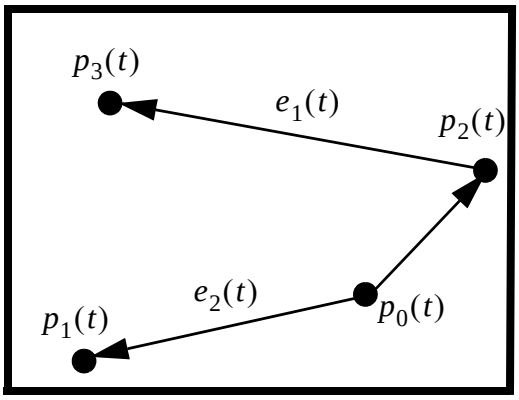


Diagram of the Edge-Edge test

4 Implementation details

The amount of numerical computation for these tests is exceedingly high when compared to static intersection tests. Numerical error accumulation is thus a significant problem and double precision is strongly recommended. The root solver used must also be very robust. It must be able to handle polynomials from degree one to three.

Determining whether a polynomial coefficient should be treated as a zero should be handled in a relative fashion by finding the largest absolute value of the coefficients and testing the quotient of the coefficients and this maximum against the tolerance value. Once the solver returns the values they are put back into the polynomial and the result is compared against a solution tolerance. If the result is less that the solution tolerance then the tolerance on the coefficients is modified and the root solver is called again. The coefficient is modified in an alternating increasing and decreasing fashion until a solution is found.

If the goal of the test is to report all collisions within some tolerance of the first collision, t_{first}, some optimization can be done. If t_{first} is passed to the tests then geometric boundary tests for all $t > t_{first} + \varepsilon$ can be avoided.

5 Usage, Advantages and Limitations

Testing all triangles of a mesh against another is still a prohibitive prospect when dealing with this algorithm. To prune tests an AABB tree can be used, where the leaf nodes contain the maximum and minimum coordinates of the triangle over the time interval. A robust implementation of this test provides very accurate contact data for a collision

response method. The exact time, point of contact, and collision normal can be provided by the information in this test.

In practice however a collision detection algorithm's value is tightly coupled to the collision response. In our case if collision response does not do perform it's task properly, the collision will not be reported again. For example in the vertex-face case if the vertex is on the wrong side of the plane after the response another application of the temporal collision will be of no use in correcting this problem. Therefore it may be wise to couple this technique with another that detects provides proximity or penetration depth information.

Another concern that must be addressed is the original assumption of linear vertex motion. In cloth simulations using a first order integrator the motion over the time interval is a linear trajectory and there is no problem. However if we try to apply this method to rigid body motions which have a non-linear angular component then this assumption is an invalid one. The techniques described in the next section address this problem. Performing triangle vs. triangle tests will produce multiple edge-edge contacts. This can be dealt with in many ways, but it is a notable concern when considering a collision response algorithm.

Experience has shown that the algorithm can perform hundreds of queries and compute all of the contact data in real-time. Coupled with other pruning techniques continuous methods can compete with and complement other collision detection strategies. The pseudo-code given below is by no means a guide to the most efficient implementation, but it demonstrates the logic of the test. Efficient retrieval and storage of the collision data is an issue left to the reader.

A.1 Pseudo-code

```
bool vertexTriangleCollide(beginPositions, velocities, tfirst,collisionData) {
    bool collided = false;
    Real tFirstVFCollision;
    computeVFPolynomialCoefficients(coeff[4],vertexBeginPositions,
                                                    vertexVelocities);
    rootSolve(coeff[4],roots[4],numRoots, coeffTol,solutionTol);
    for each root t in [0,1] {
        if (t < tfirst + eps) {
            computeVertexPositions(vertexPosAtT[4],vertexBeginPositions,
                                        vertexVelocities,t);
            if  (pointInTriangle(vertexPosAtT[4])) {
                collided = true;
                tFirstVFCollision = t;
                if (tFirstVFCollision < tfirst) {
                   tfirst = tFirstVFCollision;
                }
                storeCollisionData(collisionData);
            }
        }
    }
    return collided;
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////
bool edgeEdgeCollide(beginPositions, velocities, tfirst,collisionData) {
       bool collided = false;
       Real  tFirstEECollision;
       computeEEPolynomialCoefficients(coeff[4],vertexBeginPositions,
                                                        vertexVelocities);
       rootSolve(coeff[4],roots[4],numRoots, coeffTol,solutionTol);
       for each root t in [0,1] {
           if (t < tfirst + eps) {
              computeVertexPositions(vertexPosAtT[4],vertexBeginPositions,
                                          vertexVelocities,t);
              if  (edgeEdgeOverlap(vertexPosAtT[4])) {
                  collided = true;
                  tFirstEECollision = t;
                  if (tFirstEECollision < tfirst) {
                     tfirst = tFirstEECollision;
                  }
                  storeCollisionData(collisionData);
              }
           }
     }
     return collided;
 }

////////////////////////////////////////////////////////////////////////////////////////////////
bool triTriCollide(vertexArrayBegin, vertexArrayEnd,tfirst){
      bool collided =  false;
      computeVertexVelocities(vertexArrayBegin,vertexArrayEnd,vertexVelocities);

      for each vertex-face pair {
         Stuff vertexBegin and vertexVel with the correct data
         if (vertexTriangleCollide(vertexBegin[4], vertexVel[4],
                                      tfirst,collisionData) {
             collided = true;
         }
     }
     for each edge-edge pair {
         Stuff vertexBegin and vertexVel with the correct data
         if (edgeEdgeCollide(verticesBegin[4] vertexVel[4],
                                   tfirst,collisionData) {
             collided = true;
         }
     }
     // Consider collisionData as a static array of 15 collision data objects
     // If this triTriCollide returns true, sweep the array to get the results
     for each collisionData with t < tfirst invalidate the results
     return collided;
  }
```

A.2 Polynomial coefficients

If we let $n(t)_{ee} = (p_3(t) - p_2(t)) \times (p_1(t) - p_0(t))$ and the vector between the edges to be $(p_3(t) - p_1(t))$ then the cubic polynomial coefficients for the edge-edge test are the same as for the vertex-face test defined by **E.3.** These polynomial coefficients are provided as a cut and past convenience to the reader and are in no way in the most optimal form. The serious practitioner is encouraged to derive the terms by hand and optimize the computations. The coefficients are for the cubic polynomial of the following form.

$$At^3 + Bt^2 + Ct + D = 0$$

The "b" in the terms indicates that it is a component of the beginning vertex position. The coordinate and vertex index make up the other components of these terms. Each vertex has a velocity term defined by the difference between it's end and begin positions. For example v1z is the z coordinate of the velocity of vertex one and is defined by ez1-bz1. While a simulator may take any arbitrary time step, time has been parameterized to the interval [0,1] for the purposes of collision detection.

```
A =((v3z - v2z) * v1y + (- v3z + v1z) * v2y +
   (v2z - v1z) * v3y) * v0x + ((v2z - v3z) * v0y +
   (- v0z + v3z) * v2y + (v0z - v2z) * v3y) * v1x +
   ((- v1z + v3z) * v0y + (- v3z + v0z) * v1y +
   (- v0z + v1z) * v3y) * v2x + ((v1z - v2z) * v0y +
   (- v0z + v2z) * v1y + (- v1z + v0z) * v2y) * v3x;
```

```
B =((bz3 - bz2) * v1y + (bz1 - bz3) * v2y +
   (- bz1 + bz2) * v3y + (by2 - by3) * v1z +
   (by3 - by1) * v2z + (- by2 + by1) * v3z) * v0x +
   ((- bz3 + bz2) * v0y + (- bz0 + bz3) * v2y +
   (bz0 - bz2) * v3y + (by3 - by2) * v0z + (- by3 +
   by0) * v2z + (by2 - by0) * v3z) * v1x + ((bz3 - bz1) *
   v0y + (- bz3 + bz0) * v1y + (- bz0 + bz1) * v3y +
   (- by3 + by1) * v0z + (by3 - by0) * v1z + (by0 - by1) *
   v3z) * v2x + ((bz1 - bz2) * v0y + (bz2 - bz0) * v1y +
   bz0 - bz1) * v2y + (by2 - by1) * v0z + (by0 - by2) *
   v1z + (- by0 + by1) * v2z) * v3x + ((- bx2 + bx3) *
   v1z + (bx1 - bx3) * v2z + (- bx1 + bx2) * v3z) * v0y +
   ((bx2 - bx3) * v0z + (- bx0 + bx3) * v2z + (- bx2 +
   bx0) * v3z) * v1y + ((- bx1 + bx3) * v0z + (- bx3 +
   bx0) * v1z + (- bx0 + bx1) * v3z) * v2y +
   ((bx1 - bx2) * v0z + (bx2 - bx0) * v1z +
   (- bx1 + bx0) * v2z) * v3y;
```

```
C =(- by1 * bz2 + bz1 * by2 - bz1 * by3 + by1 * bz3 +
   bz2 * by3 - by2 * bz3) * v0x + (- bz0 * by2 +
   bz0 * by3 + by0 * bz2 - by0 * bz3 + by2 * bz3 –
   bz2 * by3) * v1x + (- bz1 * by0 - bz0 * by3 –
   by1 * bz3 + by1 * bz0 + by0 * bz3 + bz1 * by3) * v2x +
   (- by0 * bz2 - by1 * bz0 + bz1 * by0 + by1 * bz2 –
   bz1 * by2 + bz0 * by2) * v3x + (bx2 * bz3 - bx1 * bz3 +
   bx1 * bz2 + bz1 * bx3 - bz2 * bx3 - bz1 * bx2) * v0y +
   (- bx0 * bz2 + bz0 * bx2 - bx2 * bz3 + bx0 * bz3 +
   bz2 * bx3 - bz0 * bx3) * v1y + (bz0 * bx3 - bx0 * bz3 +
   bz1 * bx0 + bx1 * bz3 - bz1 * bx3 - bx1 * bz0) * v2y +
   (- bx1 * bz2 + bx1 * bz0 + bx0 * bz2 + bz1 * bx2 –
   bz1 * bx0 - bz0 * bx2) * v3y + (by1 * bx2 - by1 * bx3 –
   bx1 * by2 + bx1 * by3 - bx2 * by3 + by2 * bx3) * v0z +
   (by0 * bx3 - by2 * bx3 - bx0 * by3 + bx2 * by3 +
   bx0 * by2 - by0 * bx2) * v1z + (bx1 * by0 - bx1 * by3 +
   by1 * bx3 - by1 * bx0 + bx0 * by3 - by0 * bx3) * v2z +
   (- bx1 * by0 + bx1 * by2 - by1 * bx2 + by1 * bx0 –
   bx0 * by2 + by0 * bx2) * v3z;

D = by1 * bx2 * bz0 - bz1 * bx0 * by3 - bx1 * bz2 * by3 –
    by1 * bx2 * bz3 + bx1 * by2 * bz3 + bz1 * bx2 * by3 –
    by1 * bz0 * bx3 - bx0 * by2 * bz3 + bz1 * by2 * bx0 –
    bx1 * by2 * bz0 + by1 * bz2 * bx3 - bx1 * by0 * bz3 –
    by0 * bz2 * bx3 + bx1 * bz2 * by0 + bz0 * by2 * bx3 –
    bz0 * bx2 * by3 - by1 * bz2 * bx0 - bz1 * bx2 * by0 +
    bx0 * bz2 * by3 - bz1 * by2 * bx3 + by0 * bx2 * bz3 +
    bz1 * by0 * bx3 + bx1 * bz0 * by3 + by1 * bx0 * bz3;
```

References

[BFA02] R. Bridson, R. Fedkiw and J. Anderson, Robust Treatment of Collisions, Contact and Friction for Cloth Animation, SIGGRAPH Proceedings, (2002), pp. 596-597.

[MW88] M. Moore, and J. Wilhelms, Collision Detection and Response for Computer Animation, SIGGRAPH Proceedings, (1988), pp. 289-297.

[PRO97] X. Provot, Collision and self-collision in cloth models dedicated to design garments, Graphics Interface, (1997), pp. 177-189.

Continuous Collision Detection for Rigid and Articulated Bodies

Stephane Redon
Department of Computer Science
University of North Carolina at Chapel Hill

In these notes, we present an overview of some recent work on continuous collision detection methods, which guarantee consistent simulations by computing the time of first contact and the contact state for colliding objects. We describe techniques to perform continuous collision detection for rigid and articulated bodies. The time-parameterized equations for continuous collision detection between rigid triangle primitives are presented and methods are described to solve them efficiently. Continuous overlap tests between hierarchies of bounding volumes, which help achieve efficient collision detection for complex models, are presented as well. Some basic template data structures are introduced to allow the reader to easily start implementing the methods described in these notes. An appendix provides some code bits to help the reader start his or her own implementation.

1 Introduction

Collision detection methods can roughly be split into two categories. Most well-known collision detection methods are *discrete*: they sample the objects' trajectories at discrete times and report *interpenetrations* only. Discrete collision detection methods, whereas generally simpler to implement and used very frequently in dynamics simulators, may cause various problems. Besides the lack of physical realism resulting from the penetration, these methods can miss collisions when objects are too thin or too fast. Whereas an adaptive timestep and predictive methods can be used to correct this problem in *offline* applications, it may not be adapted in interactive applications when a relatively high and constant framerate is required.

A second problem due to the discretization of trajectories is that *backtracking* methods must be used to compute the time of first contact, which is often required in constraint-based dynamics simulation methods [FMM77, MW88, Bar90]. These backtracking methods consist in looking for the time of first contact through a recursive method. Assume that the current time interval is $[t_n, t_{n+1}]$. Essentially, one time of first contact t_e is *estimated* in this interval (for example, by taking the intermediate instant $\frac{t_n+t_{n+1}}{2}$, or by a linear or quadratic interpolation method depending on objects' velocities and and accelerations). Objects' positions are then computed at this instant and an interpenetration detection is performed again. Depending on whether the objects interpenetrate or not, the algorithm decides that the first time of collision is in $[t_n, t_e]$ or $[t_e, t_{n+1}]$, respectively, and loops on this new interval. The process stops when the amount of interpenetration is smaller than a predetermined threshold.

Such backtracking methods can have a high computational cost (which may be difficult to predict) when objects are complex or when they have interpenetrated much. Besides, since backtracking is only performed when an interpenetration has been detected, non-connected objects, or even non-

convex objects, can enter a configuration from which they could not get out (consider, for example, the case of two torii which, at one frame, would not be interlocking and, at the next one, would be).

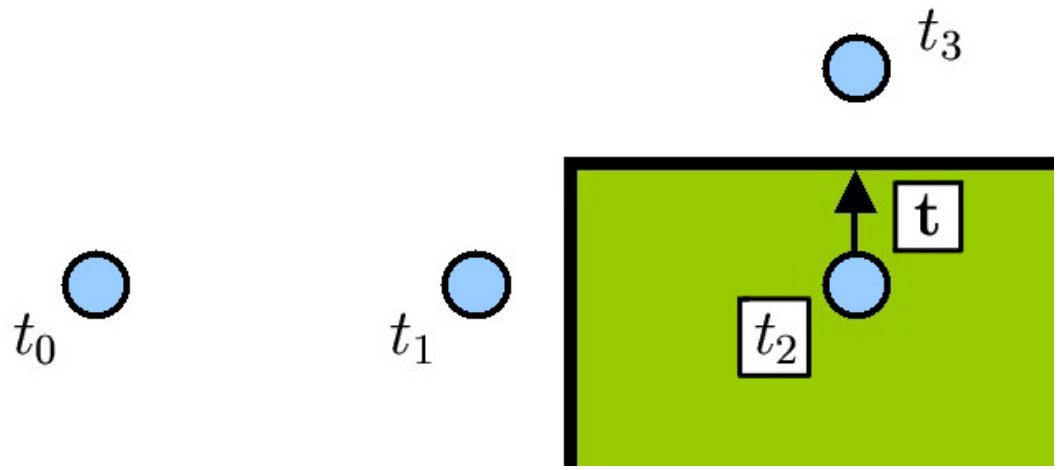


Figure 1: Allowing interpenetration between objects can lead to incoherences in the simulation. In this example, the object that penetrated at time t_2 pops out from the wrong side of the obstacle.

One way to avoid using backtracking methods is to *allow* interpenetrations between objects, which can be difficult to justify from a physical point of view, and to determine the amount of interpenetration. This problem, however, is extremely difficult for general (non-convex) objects, and the best present results do not take the trajectory of the object into account when determining this interpenetration amount [KOLM02]. Thus, in the case depicted in Figure 1, the mobile point is inside the obstacle at time t_2, but at this time the smallest interpenetration is represented by the vector $\mathbf{t}$, which leads to take out the mobile point by the top of the obstacle as time t_3. Let us note, however, that the amount of interpenetration can be used to speed up backtracking methods by leading to a better estimation of the time of first contact t_e.

Finally, the interpenetration of objects can be a cause of instability in the dynamics simulation. What is not a problem in a purely virtual simulation (*i.e.* non-interactive) has much more importance when a haptic device is used to interact with the virtual environment. For example, if one refers to the classic benchmark used to evaluate interaction methods, the *peg-in-a-hole* test depicted in Figure 2, it is clear that a first interpenetration of the peg on one side of the hole at time t (position P_t) creates a large force to remove the interpenetration. This force often leads to a greater interpenetration on the opposite side of the hole at the next instant $t + 1$ (position P_{t+1}), which creates a greater reaction force than the previous. This oscillation, by amplifying itself, leads to an unstable simulation [GMELM00].

In this part of the course, we present an overview of some recent work on *continuous* collision detection methods, which guarantee consistent simulations by computing the time of first contact and the contact state for colliding objects. We describe techniques to perform continuous collision detection for rigid and articulated bodies. The time-parameterized equations for continuous collision detection between rigid triangle primitives are presented and methods are described to solve them efficiently. Continuous overlap tests between hierarchies of bounding volumes, which help achieve

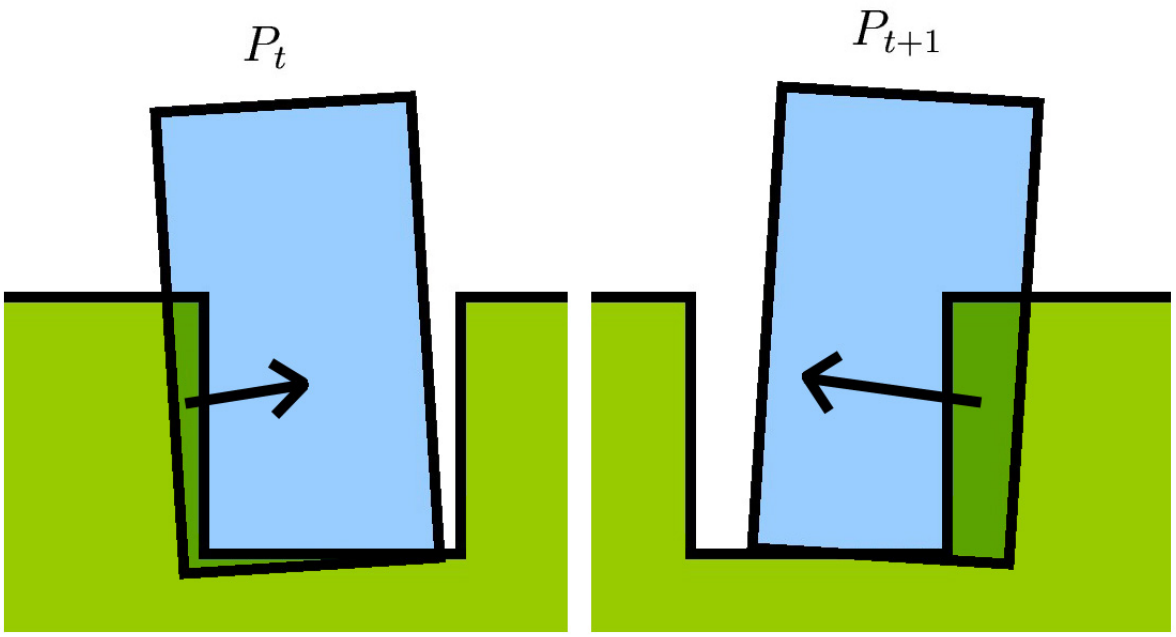


Figure 2: Interpenetrations between objects may yield unstable simulations.

efficient collision detection for complex models, are presented as well. Some basic template data structures are introduced to allow the reader to easily start implementing the methods described in these notes. Figure 3 shows an example of the benefit of using a continuous collision detection method for an articulated body. The upper half of the figure shows two successive configurations of a Puma robot which do not penetrate the environment. The lower half shows a linear interpolation of the configurations and an intermediate configuration corresponding to the first time of contact between the robot and the environment.

2 Arbitrary in-between motions

Most of the time, the actual motion of the objects is not available. Two major reasons are:

Sampling interface When one or more of the objects are acted upon trough a user interface (e.g. a simple 2d mouse, a joystick, or a full-body motion capture system), the user actions are sent at discrete times only. As a consequence, the user actions between these discrete instants are lost.

Discretized dynamics formulation When the objects take part in a dynamics simulation, the dynamics equations have to be discretized in order to be solved (e.g. through an Euler or Runge-Kutta scheme). The positions, velocities and accelerations of the objects are computed at discrete times only[1].

[1]Recall moreover that the discretization includes approximations, so that even the positions, velocities and accelerations computed at discrete times are approximations.

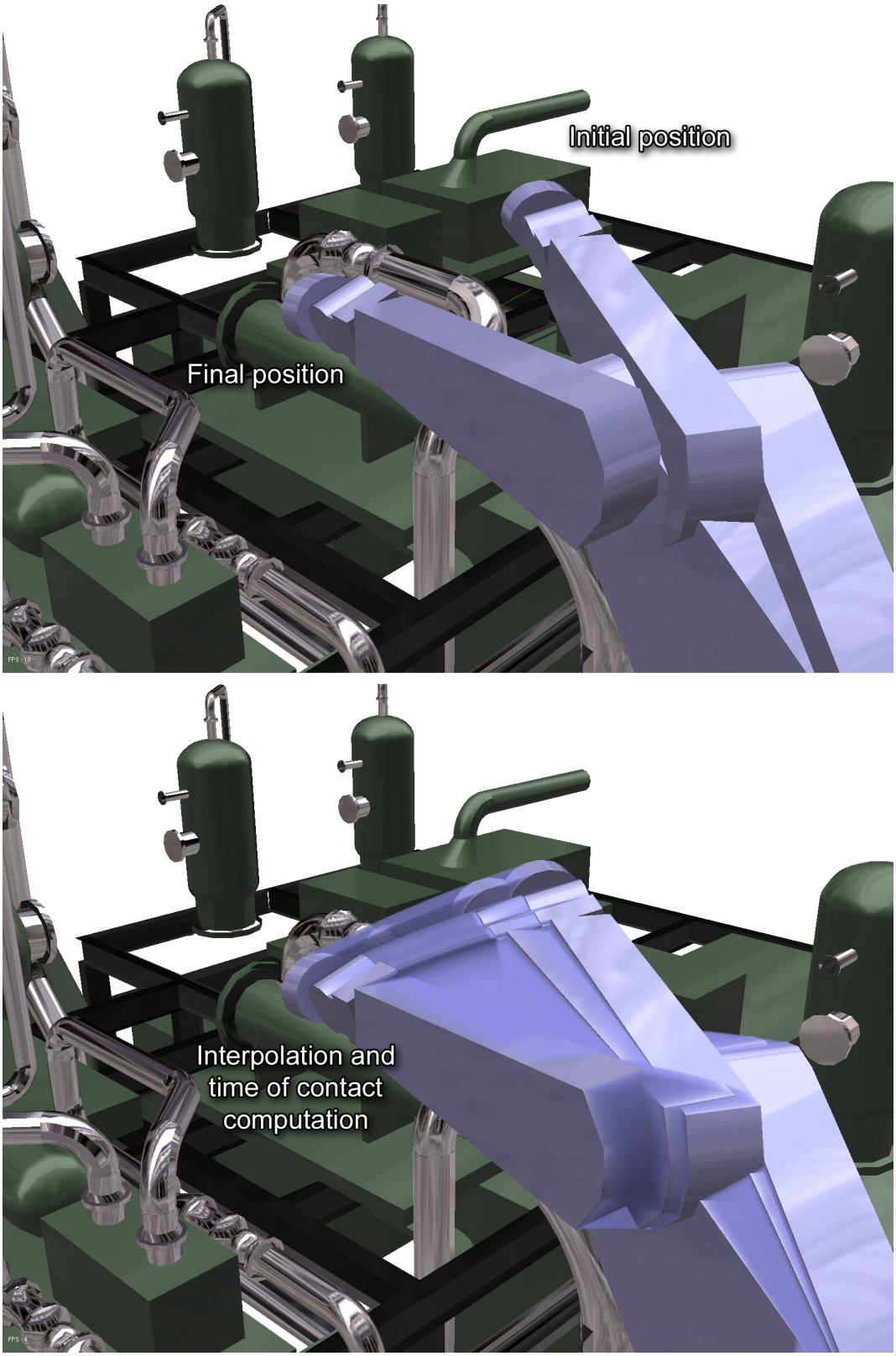


Figure 3: Benefit of using a continuous collision detection method for an articulated body. The upper half of the figure shows two successive configurations of a Puma robot which do not penetrate the environment. The lower half shows a linear interpolation of the configurations and an intermediate configuration corresponding to the first time of contact between the robot and the environment.

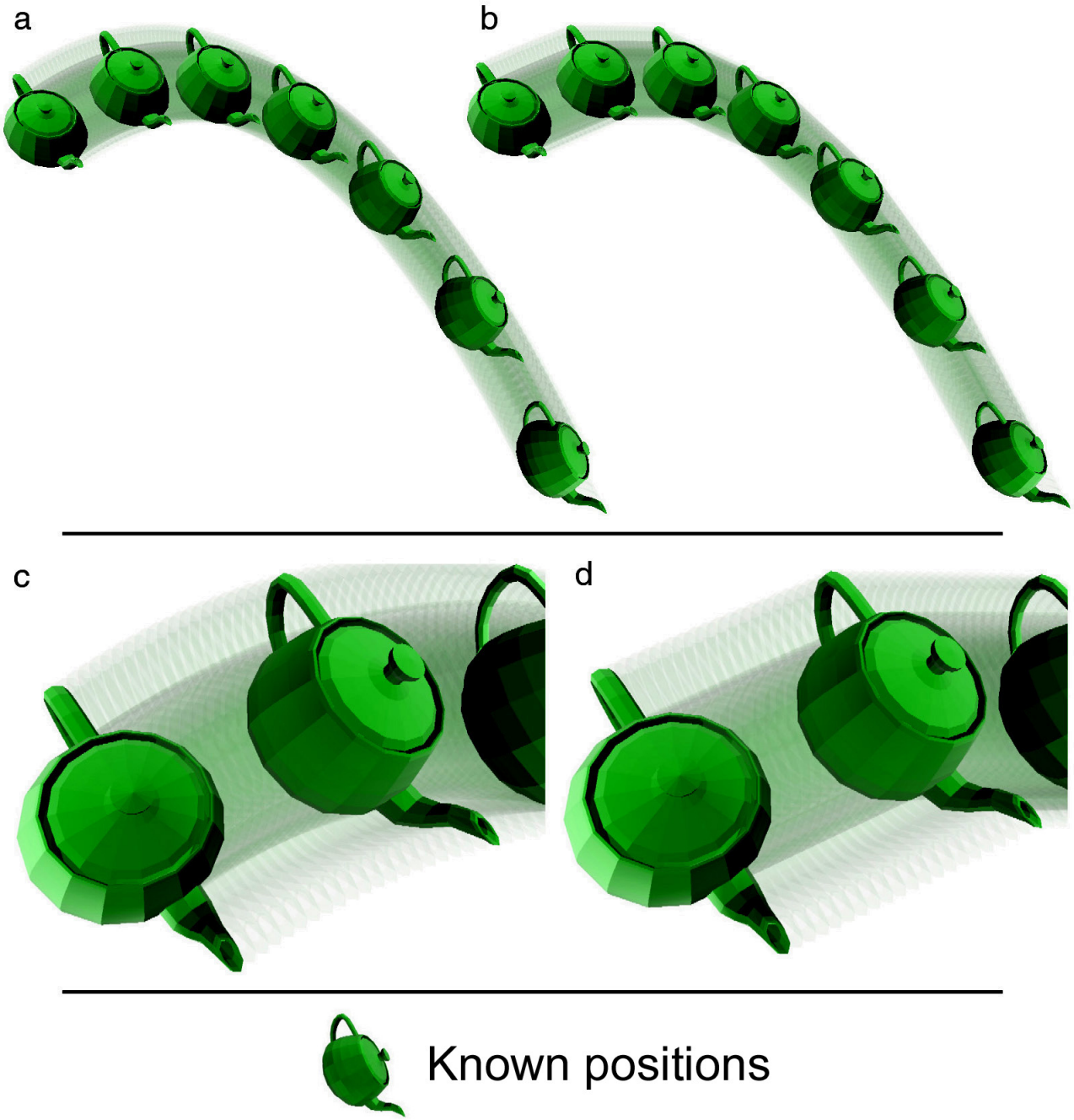


Figure 4: Use of an arbitrary in-between motion to interpolate the known successive positions of the teapot.

In order to prevent any interpenetration of the objects, we thus need to provide a continuous motion with which we will perform collision detection. Precisely, we are going to use an *arbitrary in-between motion*, which must satisfy several requirements:

Interpolation The in-between motion must at least interpolate positions. Higher order interpolations can be used depending on the application.

Continuity The interpolation must be at least $\mathcal{C}^0$. The motions we are going to use in these notes will actually be $\mathcal{C}^\infty$.

Rigidity The in-between motion needs to preserve the rigidity of the links. For consistency reasons, we cannot use a straight segment interpolation for object vertices when the object rotates.

Application-dependent constraints Depending on the application, some supplementary constraints might have to be satisfied by the in-between motion. In robotics applications, for example, some links might have a pre-defined special type of motion (e.g. screw motion). The arbitrary in-between motion chosen for the application needs to be able to produce these constrained motions.

Provided these requirements are satisfied, we can *arbitrarily* choose an in-between motion. The goal is to determine an arbitrary in-between motion which makes it efficient to perform the various steps in the continuous collision detection algorithm.

To better visualize the way the arbitrary motion is used, let's consider the case of an interactive dynamics simulator which computes the position of a teapot and render a new frame at a fixed rate of 30 frames per second, for example. Between two frames, the motion of the teapot is not visible by the user and is replaced by an arbitrary in-between motion. Since the real positions are respected at the successive discrete instants, only the *local* motion of the teapot is modified, and its *global* motion is preserved, as shown in Figure 4. In this example, the position of the teapot is known at seven discrete instants t_0, ..., t_6. In the left part ((a), zoomed in (c)), the real motion of the teapot between these instants is represented in transparency. In the right part ((b), zoomed in (d)), the known positions are preserved, but the real motion of the object between two successive known positions has been replaced by an arbitrary in-between motion, represented in transparency. The general aspect of the teapot trajectory is preserved.

It should be clear, now, that using an arbitrary in-between motion is more or less equivalent to the underlying principle of every integration scheme used to solve the dynamics differential equations: discretizing the differential equations amounts to make finite-order assumptions on velocities and accelerations between successive discrete instants.

Let us now formalize the constraints imposed on the arbitrary in-between motion. To do this, let $\mathbf{P}_R(t)$ denote the 4×4 matrix describing the *real* position of the object during the time interval $[t_n, t_{n+1}]$. Recall that this matrix allows us to compute the real (homogeneous) coordinates $\mathbf{x}_R(t)$ of a point of the object in the global frame from its (homogeneous) coordinates $\mathbf{x}_o$ in the local frame of the object:

$$\mathbf{x}_R(t) = \mathbf{P}_R(t)\mathbf{x}_o. \tag{1}$$

Vectors $\mathbf{x}_R(t)$ and $\mathbf{x}_o$ are homogeneous vectors in $\mathbb{R}^4$, for which the last coordinate is the real number 1.

Let's denote now the object's position matrix when using the *arbitrary* motion, during the same time interval $[t_n, t_{n+1}]$ by $\mathbf{P}_A(t)$. Similarly, the arbitrary coordinates $\mathbf{x}_A(t)$ of a point of the object

in the global frame are obtained from its coordinates in the local frame of the object:

$$\mathbf{x}_A(t) = \mathbf{P}_A(t)\mathbf{x}_o \tag{2}$$

The three constraints can be formalized simply:

- the interpolation constraint merely imposes that $\mathbf{P}_A(t_n) = \mathbf{P}_R(t_n)$, as well as $\mathbf{P}_A(t_{n+1}) = \mathbf{P}_R(t_{n+1})$,
- the continuity constraint imposes that the function $t \mapsto \mathbf{P}_A(t)$ is continuous on the interval $[t_n, t_{n+1}]$, that is, that the components of $\mathbf{R}_A(t)$ and $\mathbf{T}_A(t)$ are continuous functions of time on this interval.
- the rigidity constraint imposes that the matrix $\mathbf{P}_A(t)$ is a *position* matrix at every time t between t_n and t_{n+1}. In other words, it must not include deformation terms (scaling terms, for example), and must be the combination of a rotation matrix and a translation vector, according to the classic form of a homogeneous position matrix:

$$\mathbf{P}_A(t) = \begin{pmatrix} \mathbf{R}_A(t) & \mathbf{T}_A(t) \\ \mathbf{0} & 1 \end{pmatrix}, \tag{3}$$

Of course, a fourth constraint is implicit: the arbitrary in-between motion should be close to the real motion. It would thus be desirable to define a measure allowing to evaluate the difference between the real object motion and the arbitrary one, used to detect collisions between the successive discrete instants. However, we noticed that this real motion is rarely available. For this reason, we suggest using the motion defined by the position and velocity determined by the dynamics calculator as a reference. Thus, we saw that for each successive discrete instant the dynamics calculator computes objects' positions and velocities from those of the previous discrete instant and possibly, for higher-order integration schemes (as the fourth-order Runge-Kutta integration scheme), of those established to certain intermediate instants. It is possible, as is done in a way by the dynamics calculator, to assume that the object's rotational and translation velocities are constant during the timestep. A natural arbitrary motion is then a motion whose velocities remain close to these reference velocities.

Although it is possible to define this last constraint more rigorously, for example by defining the maximal error on the position of points of the object resulting from the use of an arbitrary motion, we noticed empirically that the arbitrary motions described in these notes are always sufficiently natural so that the user is not surprised by the position of the object that he manipulates at the time of the collision (as one can expect when examining Figure 4).

Let's note that replacing the objects' motions by arbitrary ones between two successive discrete instants t_n and t_{n+1} has a consequence on the simulation only if a collision occurs between these two instants. If no collision is detected during the in-between time interval, the objects are placed to the final positions computed by the dynamics calculator. However, if a collision between two objects is detected at time t_c, it is necessary to use the arbitrary motions to compute the positions of *all* the objects at time t_c, since these motions have been used for the detection of collisions.

Otherwise, some interpenetrations could occur, as shown in Figure 5. In (a), a collision has been detected at time t_c while using the arbitrary in-between motion. In (b), the dynamics calculator has been used to compute the real position of the object at time t_c, which results in an interpenetration. For the same reason, when a collision is detected, the arbitrary motion must also be used for objects that did not enter in contact with another object. To compute their real positions at time t_c using the dynamics calculator could induce interpenetrations, since these positions have not been used for the detection of collisions.

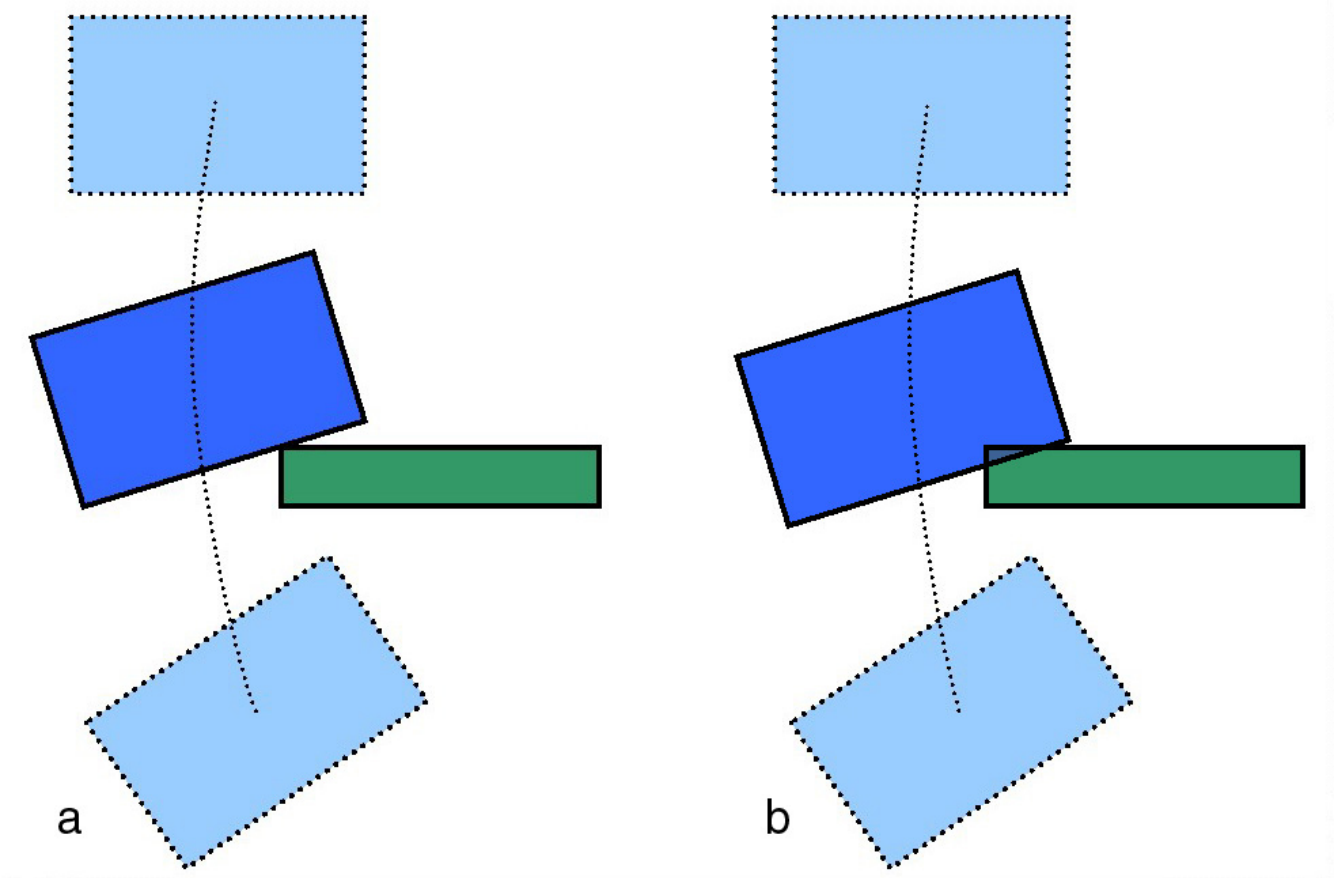


Figure 5: To avoid interpenetrations, it is necessary to compute the objects' positions at the instant of collision from the in-between motion used for the detection of collisions, and not from the interpolating motion computed by the dynamics equations.

Consequently, the use of an arbitrary in-between motion to detect collisions perturbs the course of the simulation. It is indeed very unlikely that the real object motion and the arbitrary in-between motion would lead to detect collisions at the same instants and at the same places. It is not even guaranteed that a collision which occurs between two objects when one of the two motions (real or arbitrary) is used would also occur when the other motion is used. This is the price we have to pay to perform continuous collision detection when the actual object motion is not known. This allows, however, to continuously detect collisions very efficiently, while preserving the benefits of a continuous method that would use the real object motion. Indeed, with this method, objects are permanently in a consistent state: no interpenetration is possible and no collision can be missed.

Besides, for some more rigorous physical applications, it should be sufficient to reduce the length of the timestep, thanks to the continuity and interpolation constraints imposed on the arbitrary motion. Indeed, the instants between which the motion of the object is replaced are not necessarily instants to which objects are displayed. Let's assume, for example, that the object moves very fast on the time interval $[t_n, t_{n+1}]$. In order to reduce the error between the real object motion and the arbitrary one used for continuous collision detection, it can be preferred to divide the time interval in two smaller intervals $[t_n, t_i]$ and $[t_i, t_{n+1}]$, where $t_i = \frac{t_n+t_{n+1}}{2}$. At the intermediate time t_i, the position of the object is evaluated by the dynamics simulator and is used to replace the real object motion by *two* successive arbitrary motions, on $[t_n, t_i]$ first then on $[t_i, t_{n+1}]$. This ensures that the intermediate position to the time t_i, at least, is the one computed by the dynamics simulator.

Thus, the error created by the use of an arbitrary in-between motion relies upon the same approximation principle than the one which prevails when discretizing the dynamics equations. Overall, we believe that this approximation problem is largely compensated by the benefits provided by a continuous collision detection method.

To conclude this section, let's summarize the (simple) idea behind the use of an arbitrary in-between motion: *since the real object motion between any two successive discrete instants cannot be used to continuously detect collisions, it is replaced by an arbitrarily fixed in-between motion, which must satisfy three constraints: this arbitrary motion must* interpolate *in a* continuous *and* rigid *way the object's configurations between successive discrete instants. Among the arbitrary motions which satisfy these constraints, we choose one which allows us to perform the various steps of the continuous collision detection algorithm very efficiently.*

2.1 Rigid bodies

Let us now describe two possible arbitrary in-between motions for rigid bodies. Again, recall that we want to choose a simple motion.

2.1.1 Constant-velocity translation and rotation

One possibility is to assume that the rigid motion over the timestep is a constant-velocity one, composed of a translation along a fixed direction, and a rotation along a fixed (potentially distinct) direction.

Let the 3-dimensional vector $\mathbf{c}^0$ and the 3×3 matrix $\mathbf{R}^0$ denote the position and orientation of the rigid body in the world frame at the beginning of the (normalized) time interval $[0, 1]$. Let $\mathbf{s}$ denote the total translation during the timestep, and let ω and $\mathbf{u}$ respectively denote the total rotation angle and the rotation axis. For a given timestep, $\mathbf{c}^0$, $\mathbf{R}^0$, ω, $\mathbf{u}$ and $\mathbf{s}$ are constants.

The position of the rigid body at a given time t in $[0, 1]$ is thus:

$$\mathbf{T}(t) = \mathbf{c}^0 + t\mathbf{s}, \tag{4}$$

The orientation of the rigid body is:

$$\mathbf{R}(t) = \cos(\omega t).\mathbf{A} + \sin(\omega t).\mathbf{B} + \mathbf{C}, \tag{5}$$

where $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are 3×3 constant matrices which are computed at the beginning of the time step:

$$\begin{aligned} \mathbf{A} &= \mathbf{R}^0 - \mathbf{u}.\mathbf{u}^T.\mathbf{R}^0 \\ \mathbf{B} &= \mathbf{u}^*.\mathbf{R}^0 \\ \mathbf{C} &= \mathbf{u}.\mathbf{u}^T.\mathbf{R}^0 \end{aligned} \tag{6}$$

where $\mathbf{u}^*$ denotes the 3×3 matrix such as $\mathbf{u}^*\mathbf{x} = \mathbf{u} \times \mathbf{x}$ for every three-dimensional vector $\mathbf{x}$. If $\mathbf{u} = (u^x, u^y, u^z)^T$, then:

$$\mathbf{u}^* = \begin{pmatrix} 0 & -u^z & u^y \\ u^z & 0 & -u^x \\ -u^y & u^x & 0 \end{pmatrix} \tag{7}$$

Consequently, the motion of the rigid body is described by the following 4×4 homogeneous matrix:

$$\mathbf{P}(t) = \begin{pmatrix} \mathbf{R}(t) & \mathbf{T}(t) \\ \mathbf{0} & 1 \end{pmatrix}, \tag{8}$$

in the world frame.

The motion parameters $\mathbf{s}$, $\mathbf{u}$ and ω are easy to compute. Assume $\mathbf{c}^0$ and $\mathbf{c}^1$ (resp. $\mathbf{R}^0$ and $\mathbf{R}^1$) are the initial and final positions (resp. orientations) of the rigid body in the world frame. Then $\mathbf{s} = \mathbf{c}^1 - \mathbf{c}^0$, and $(\mathbf{u}, \omega)$ is the rotation extracted from the rotation matrix $\mathbf{R}^1(\mathbf{R}^0)^T$. A code bit which performs this extraction is provided in the appendix.

2.1.2 Screw motions

An even simpler motion can be used, for which the rotation axis and the translation have the same direction. Such a motion is called a *screw motion*.

Precisely, a screw motion $\mathcal{V}(\omega, s, \mathbf{O}, \mathbf{u})$ is the commutative composition of a rotation and a translation along the same axis. The real parameters ω and s (now a real number) respectively denote the total amount of rotation and the total amount of translation in the transformation, $\mathbf{O}$ is a point on the the screw motion axis, and $\mathbf{u}$ is a unit vector describing the axis orientation. Note that the total translation is now $\mathbf{s} = s.\mathbf{u}$. A screw motion is depicted in Figure 6. In this example, the screw motion transforms the point $\mathbf{A}$ into $\mathbf{A}'$. Depending on whether the rotation or the translation is applied first to the point $\mathbf{A}$, the intermediate point is respectively $\mathbf{A}_1$ or $\mathbf{A}_2$.

The benefit of using screw motions comes from the fact that they allow us to interpolate any two rigid positions with less degrees of freedom, and thus reduce the computational cost of evaluating the motion matrix. Whatever the object positions at times t_n and t_{n+1}, Chasles' theorem states that there exists an unique screw motion which transforms the initial position (*i.e.* at time t_n) into the final position (*i.e.* at time t_{n+1}) (when $\mathbf{O}$ on the screw motion axis is fixed, and when ω is required to be positive [Cha1831]). A code bit provided in the appendix shows how the equivalent screw motion can be obtained directly from the object's translational and rotational velocities.

In theory, using a screw motion to interpolate two successive positions could lead to a non natural in-between motion. In Figure 7, the real object motion (on the left) has been replaced by the equivalent screw motion with positive angle (on the right). For applications which require a very large rotation angle over the time interval $[0, 1]$, it might be advisable to subdivide the time interval into several smaller ones.

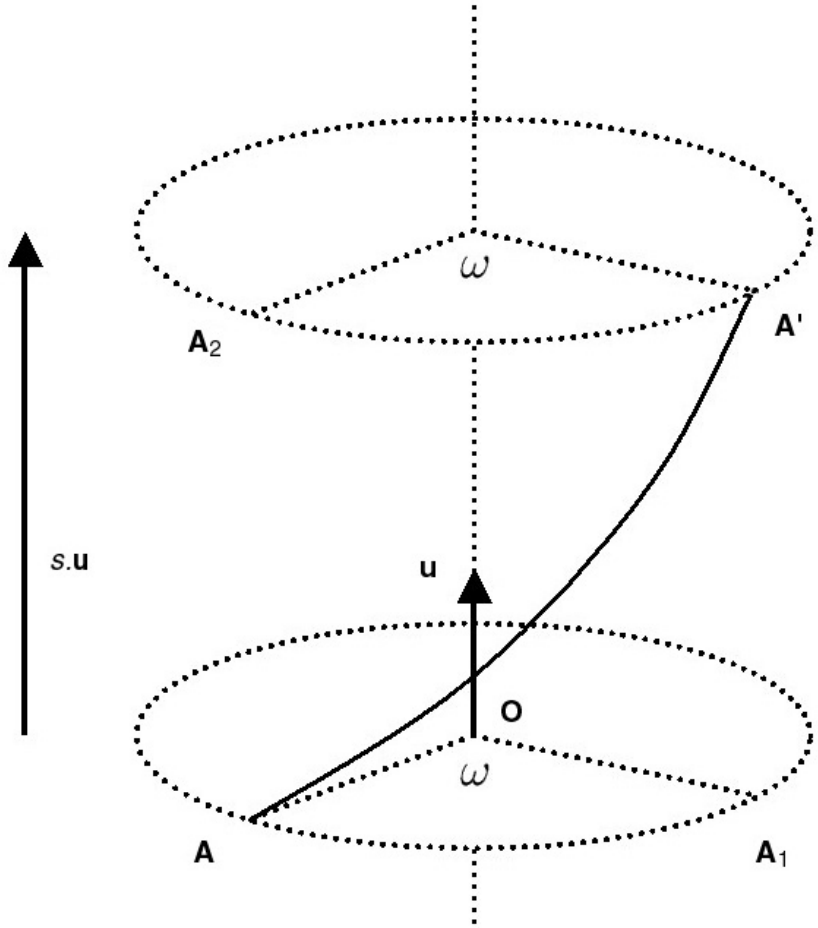


Figure 6: A screw motion is the commutative composition of a rotation and a translation of same axes.

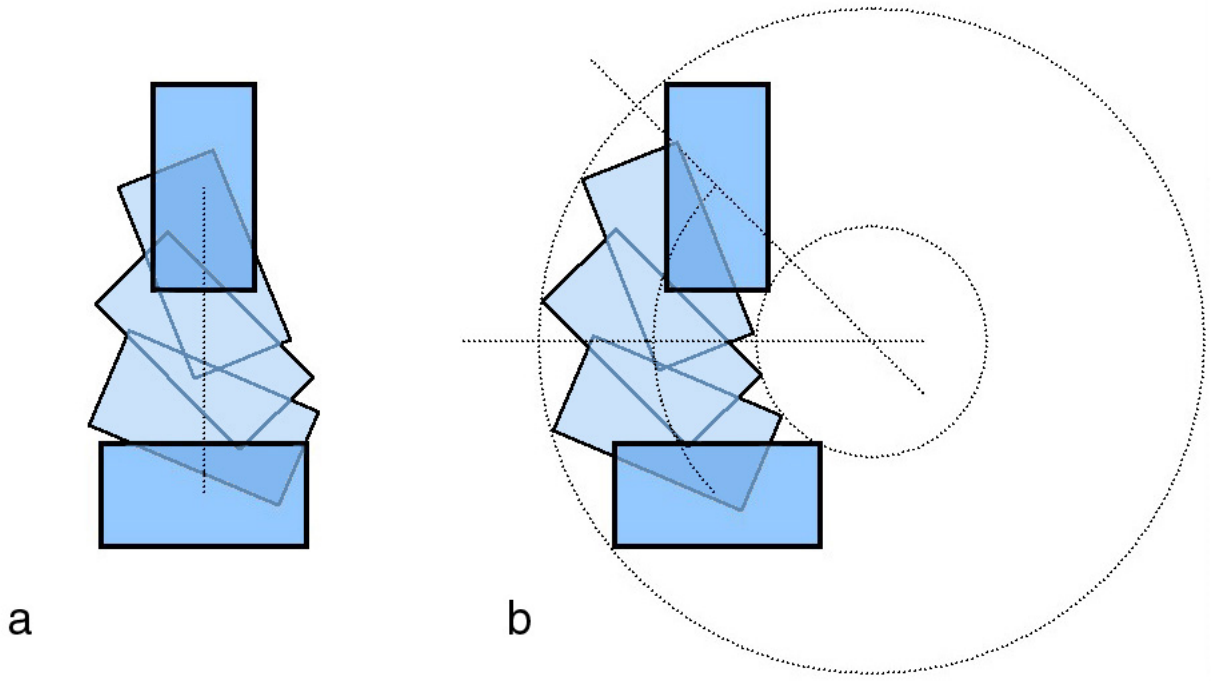


Figure 7: Using a screw motion to replace the real object motion. a: the real object motion is a pure translation at constant velocity (from top to bottom) combined to a rotation at constant velocity around the object's center of mass. b: the real object motion has been replaced by the equivalent (and unique) screw motion with positive angle. For applications which require a very large rotation angle over the time interval $[0, 1]$, it might be advisable to subdivide the time interval into several smaller ones.

We can now build a general class of screw motion-based arbitrary in-between motions. Assume, without loss of generality, that the current time interval is the interval $[0, 1]$. In order to get a rigid and continuous motion that interpolates the initial and final positions, it is sufficient to make the parameters $\omega$and s vary continuously. This can be achieved by choosing two functions $a : \mathbb{R}^2 \times [0, 1] \to \mathbb{R}$ and $b : \mathbb{R}^2 \times [0, 1] \to \mathbb{R}$ such as, for all pair (ω, s) in $\mathbb{R}^2$, the functions

$$\begin{aligned} a_{\omega,s} &: \left\{ \begin{array}{c} [0,1] \to \mathbb{R} \\ t \mapsto \omega(t) = a(\omega, s, t) \end{array} \right. \\ b_{\omega,s} &: \left\{ \begin{array}{c} [0,1] \to \mathbb{R} \\ t \mapsto s(t) = b(\omega, s, t) \end{array} \right. \end{aligned} \tag{9}$$

are C^1, monotonous, and respect the interpolation constraint, *i.e.* $a_{\omega,s}(0) = b_{\omega,s}(0) = 0$, and $a_{\omega,s}(1) = \omega$ and $b_{\omega,s}(1) = s$.

The class of screw motion-based arbitrary in-between motions has the form:

$$\mathcal{M} : \left\{ \begin{array}{c} [0,1] \times \mathbb{R}^3 \to \mathbb{R}^3 \\ (t, A) \mapsto A(t) = \mathcal{V}(a_{\omega,s}(t), b_{\omega,s}(t), O, \vec{u})(A_0) \end{array} \right. \tag{10}$$

where A_0 is a point of the object at time 0 and $A(t)$ the same point during the arbitrary in-between motion. It is worth noticing that the two functions a and b depend on the screw motion parameters only, and not on the object shape or part. This guarantees that all points of the object have the same rigid motion. Besides, thanks to the conditions imposed on the functions $a_{\omega,s}$ and $b_{\omega,s}$, arbitrary motions of form (10) are truly rigid, continuous and interpolating.

A motion in the class (10) can be expressed simply in matrix form. Define first a *screw motion frame* as a frame in which the Oz axis is the screw motion axis. Because of axial symmetry, there exists an infinity of such frames, and it is sufficient to choose one of them. In one of these frames, the screw motion can be expressed simply:

$$\mathbf{V}(t) = \begin{pmatrix} \cos(a_{\omega,s}(t)) & -\sin(a_{\omega,s}(t)) & 0 & 0 \\ \sin(a_{\omega,s}(t)) & \cos(a_{\omega,s}(t)) & 0 & 0 \\ 0 & 0 & 1 & b_{\omega,s}(t) \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{11}$$

for $t \in [0, 1]$. In the global frame, the screw motion is then:

$$\mathbf{S}(t) = \mathbf{P}_V^{-1} \mathbf{V}(t) \mathbf{P}_V \tag{12}$$

where $\mathbf{V}(t)$ is the screw motion with Oz axis, $\mathbf{P}_V$ is the transformation matrix from the global frame to the screw motion frame, and $\mathbf{P}_V^{-1}$ is the inverse of $\mathbf{P}_V$.

Thanks to the expression of the screw motion in the global frame (12), it is possible to get the coordinates of any object point $\mathbf{x}(t)$ during the arbitrary in-between motion:

$$\mathbf{x}(t) = \mathbf{P}(t)\mathbf{x}_o = \mathbf{P}_V^{-1} \mathbf{V}(t) \mathbf{P}_V \mathbf{P}_0 \mathbf{x}_o \tag{13}$$

where $\mathbf{x}_o$ denotes the point coordinates in the object frame, and $\mathbf{P}_0$ is the objects's position matrix at time 0. The object's position matrix during the arbitrary motion is $\mathbf{P}(t)$.

2.2 Articulated bodies

An articulated body is defined as a set of rigid bodies, or *links*, connected by bilateral constraints. Assume there is no loop in the articulated body. It is simple to define an arbitrary in-between motion:

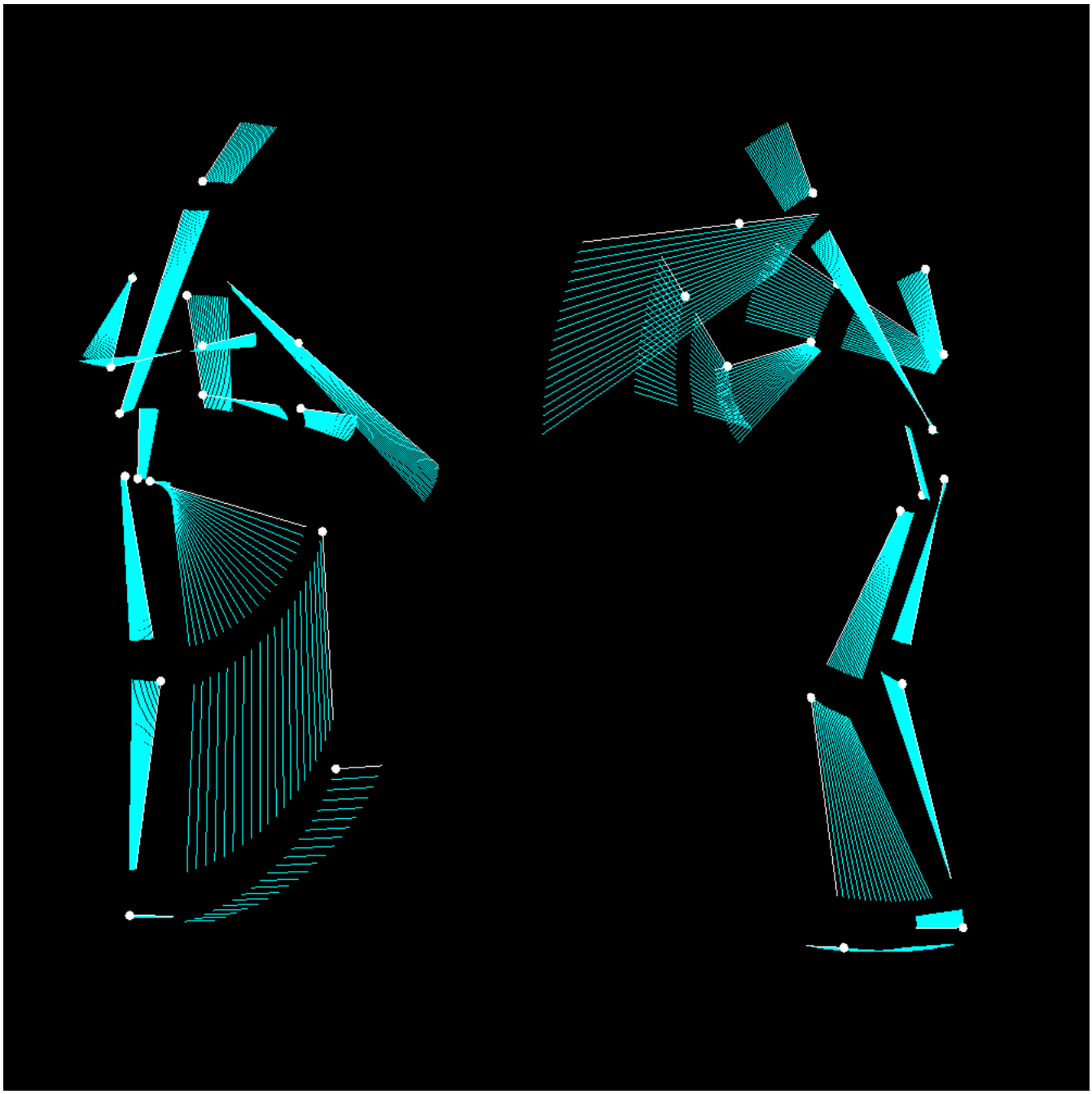


Figure 8: An arbitrary in-between motion for an articulated model. The motion of each link has a constant translational and rotational velocity in the reference frame of its parent. The rigidity of each link is preserved.

it suffices to express the motion of each link in the reference frame of its parent link, and not in the world frame. The motion of the root link of the articulated model is still expressed in the world frame.

Assume, for the sake of simplicity of notation, that the parent of link i is $i-1$. The index denoting the world frame is 0. Let $\mathbf{P}_i^{i-1}(t)$ denote the position matrix of link i in the reference frame of its parent link $i-1$. Then the matrix

$$\mathbf{P}_i^0(t) = \mathbf{P}_1^0(t).\mathbf{P}_2^1(t)...\mathbf{P}_i^{i-1}(t) \tag{14}$$

describes the motion of link i in the world frame.

Figure 8 shows an interpolating motion for an articulated model, where the motion of each link has a constant translational and rotational velocity in the reference frame of its parent. Note that the rigidity of each link is preserved.

3 Interval arithmetic

A simple way to robustly perform the computations involved in the various steps of a continuous collision detection algorithm is to use interval arithmetic.

Interval arithmetic consists in computing with intervals instead of numbers. Several good introductions to interval arithmetic can be found for example in [Moo62, Sny92, Kea96]. As is well known, the definition of a closed real interval $[a, b]$ is:

$$I = [a, b] = \{x \in \mathbb{R},\ a \leqslant x \leqslant b\} \tag{15}$$

This definition can be generalized to vectors. A vector interval is simply a vector whose components are intervals:

$$\begin{aligned} I_n &= [a_1, b_1] \times ... \times [a_n, b_n] \\ &= \{\mathbf{x} = (x_1,...,x_n) \in \mathbb{R}^n,\ a_i \leqslant x_i \leqslant b_i \quad \forall i,\quad 1 \leqslant i \leqslant n\} \end{aligned} \tag{16}$$

In $\mathbb{IR}^3$, for example, a simple alternate notation can be:

$$\begin{pmatrix} [x_l, x_u] \\ [y_l, y_u] \\ [z_l, z_u] \end{pmatrix} \tag{17}$$

The set of intervals is denoted $\mathbb{IR}$, while the set of vector intervals is denoted $\mathbb{IR}^n$.

Basic operations can be transposed to intervals:

$$\begin{aligned} [a, b] + [c, d] &= [a + c, b + d] \\ [a, b] - [c, d] &= [a - d, b - c] \\ [a, b] \times [c, d] &= [min(ac, ad, bc, bd), max(ac, ad, bc, bd)] \\ 1/\,[a, b] &= [1/b, 1/a] \quad \text{if } a > 0 \text{ or } b < 0 \\ [a, b]\,/\,[c, d] &= [a, b] \times (1/\,[c, d]) \quad \text{if } c > 0 \text{ or } d < 0 \\ [a, b] &\leqslant [c, d] \quad \text{if } b \leqslant c \end{aligned} \tag{18}$$

Elementary operations in $\mathbb{IR}^n$ are performed component-wise. Operations between real numbers and real intervals can be performed by identifying $\mathbb{R}$ and the set of "point" intervals $\{[x, x], x \in \mathbb{R}\}$.

Interval arithmetic can be used to bound a function over an interval very easily, provided the analytic expression of the function is known, and provided we can easily bound the sub-expressions in the function.

An example will make this clear. Assume we want to bound the function $t \mapsto \sqrt{3}\cos(t) + \sin(t)$ over the time interval $[0, \pi/2]$. This function is very similar to the ones we obtain when we plug the arbitrary in-between motions described above in the continuous collision detection equations.

Being able to bound the sine and cosine sub-expressions is all that is required to bound this function. We know that:

$$t \in \left[0, \frac{\pi}{2}\right] \Rightarrow \begin{cases} \cos(t) \in [0,1] \\ \sin(t) \in [0,1] \end{cases}$$

Note that this is not deduced from the elementary interval operations, but has to be known. This is what is meant by "we can bound easily the sub-expressions in the function". From now on, however, we only need to use the elementary interval operations to provide some bounds on the function. Since, by definition,

$$\sqrt{3} \in \left[\sqrt{3}, \sqrt{3}\right],$$

and

$$\cos(t) \in [0,1], \forall t \in \left[0, \frac{\pi}{2}\right],$$

we determine that

$$\sqrt{3}\cos(t) \in \left[\sqrt{3}, \sqrt{3}\right] \times [0,1] = \left[0, \sqrt{3}\right], \forall t \in \left[0, \frac{\pi}{2}\right],$$

by performing a simple interval multiplication.

Similarly, using the interval addition, we know that

$$\sqrt{3}\cos(t) + \sin(t) \in \left[0, \sqrt{3}\right] + [0,1] = \left[0, \sqrt{3}+1\right], \forall t \in \left[0, \frac{\pi}{2}\right],$$

and we have thus bounded the function.

Note that the bounds we have obtained are not exact, since the tightest bounding interval is actually $[1,2]$. In this example, the reason for the looseness of the bounds is that the sine function is increasing while the cosine function is decreasing. Provided we know exact bounds on these sub-expressions, however, it can be shown that the bounds on the function tend to be exact when the size of the time interval tends towards zero.

Exact bounds on the sub-expressions we encounter in these notes are actually very easy to obtain. For example, since the cosine function is decreasing over $[0, \pi/2]$, we know that

$$a, b \in \left[0, \frac{\pi}{2}\right], a < b \Rightarrow \cos(t) \in [\cos(b), \cos(a)], \forall t \in [a, b].$$

The power of interval arithmetic for our purpose comes from the fact that efficient interval operations can be simply implemented. Some code bits are provided in the appendix. They describe basic data structures to represent intervals, three-dimensional vector intervals, and 3×3 interval matrices, respectively `cInterval`, `cIAVector3` and `cIAMatrix33`. They should constitute a good starting point to implement interval computations. For example, denoting `cosBounds` and `sinBounds` the intervals respectively bounding the cosine and sine functions on the time interval $[0, \pi/2]$, the interval bound previously computed is simply

```
cInterval bounds=cInterval(sqrt(3))*cosBounds+sinBounds; // bounds=[0,sqrt(3)+1]
```

We can now describe how to perform elementary continuous collision detection and continuous overlap tests between bounding volumes.

4 Elementary continuous collision detection

Continuous collision detection methods for polyhedral objects must only detect three types of contact. Indeed, all contacts between two polyhedral objects A and B include at least one of these three *elementary* contact types:

- an edge of A contacts an edge of B.
- a vertex of A contacts a face of B.
- a face of A contacts a point of B.

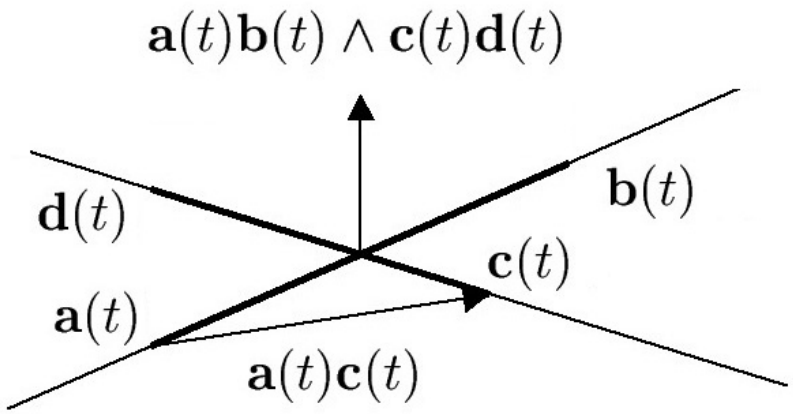


Figure 9: Collision detection between two edges.

These contact types are easily expressed geometrically. For the edge/edge case, it suffices to detect a collision between the lines containing the edges. If $\mathbf{a}(t)\mathbf{b}(t)$ is the first edge and $\mathbf{c}(t)\mathbf{d}(t)$ is the second edge, then the lines intersect when:

$$\mathbf{a}(t)\mathbf{c}(t) \cdot (\mathbf{a}(t)\mathbf{b}(t) \wedge \mathbf{c}(t)\mathbf{d}(t)) = 0, \tag{19}$$

i.e. when the vector $\mathbf{a}(t)\mathbf{c}(t)$ is in the plane defined by the two edges (*cf.* Figure 9). Once an intersection has been detected at some instant between the two lines, we check whether it belongs to the edges or, equivalently, if the edges intersect *at that time* (and not only the supporting lines. This can be robustly performed thanks to a discrete edge/edge proximity test (in general, due to finite precision computations, the edges do not *exactly* touch at the collision time). We then keep the earliest valid collision. The contact time is the earliest valid collision time. The contact position is the position of the vertex at that time, and the contact normal is the (normalized) cross-product of the edges at that time.

For the vertex/face and face/vertex, a collision is first detected between the point and the plane containing the face. If $\mathbf{a}(t)$ is the point and $\mathbf{b}(t)\mathbf{c}(t)\mathbf{d}(t)$ is the triangle, a collision occurs when:

$$\mathbf{a}(t)\mathbf{b}(t) \cdot (\mathbf{b}(t)\mathbf{c}(t) \wedge \mathbf{b}(t)\mathbf{d}(t)) = 0, \tag{20}$$

that is when the vector $\mathbf{a}(t)\mathbf{b}(t)$ is in the vector plane defined by the face normal $\mathbf{b}(t)\mathbf{c}(t) \wedge \mathbf{b}(t)\mathbf{d}(t)$ (*cf.* Figure 10). When such a collision is detected, we check whether the point belongs to the face

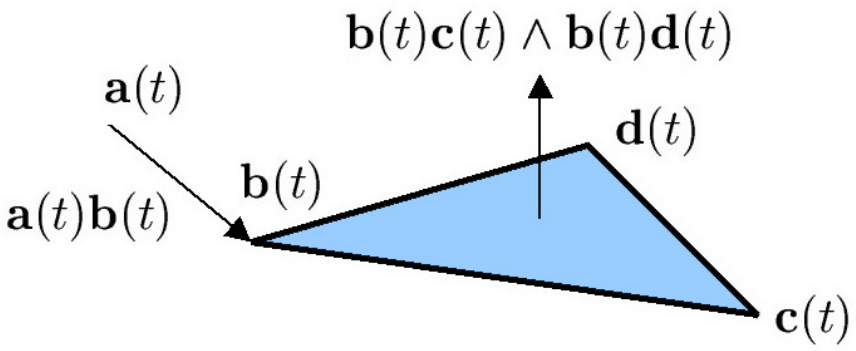


Figure 10: Collision detection between a point and a face.

at that time. This can be robustly performed thanks to a vertex/triangle proximity test (in general, due to finite precision computations, the vertex is not *exactly* in the plane at the collision time). We then keep the earliest valid collision. The contact time is the earliest valid collision time. The contact position is the position of the vertex at that time, and the contact normal is the normal to the triangle at that time.

In practice, interval arithmetic can be used to solve equations (19) and (20). Formally, these equations have the form

$$f(t) = 0, t \in [0, 1],$$

and we want to determine the smallest root t_c. Assume we are able to bound the function f over the time interval $[0, 1]$. If these bounds do not contain zero, meaning that the function is strictly positive or strictly negative over the time interval $[0, 1]$, then f cannot have any root in $[0, 1]$.

However, if these bounds *do* contain zero, then the function f *might* have a root in $[0, 1]$ (might only, if the bounds are not tight or if the function is not continuous[2]). In this case, we refine the time interval and repeat the process: we bound the function f on the time intervals $[0, 1/2]$ and $[1/2, 1]$, and we examine these bounds (first $[0, 1/2]$ and then $[1/2, 1]$, since we are looking for the *earliest* collision). This process is recursively performed until the examined bounds do not contain zero (meaning that the function does not have any root on the time sub-interval), or until the size of the examined time sub-interval is smaller than a user-defined threshold (which characterizes the temporal precision of the collision detection).

The C++ code for this interval recursive root-finding method is:

```
bool computeCollisionTime(cInterval I, double &tc) {

    // I is the time interval currently examined.
    // Initially, I=[0,1].
    //
    // tc is the time of earliest collision.
    //
    // The function returns true if and only if a collision has been found

    cInterval boundsF=boundFunctionF(I); // bound f over I
```

[2]Of course, the functions involved in these notes are all continuous.

```
    if (boundsF.i[0]>0) return false; // the bounds do not contain 0
    if (boundsF.i[1]<0) return false; // the bounds do not contain 0

    // from here, the bounds contain 0: there might be a collision

    if ((I.i[1]-I.i[0])<timeThreshold) { // sufficient precision

        bool valid=checkRootValidityF(I); // check the validity of the root
        if (valid) tc=I.i[0]; // return a conservative collision time
        return valid;

    }

    // insufficient time precision, refine the time interval

    double m=0.5*(I.i[0]+I.i[1]); // mid-time
    bool rootFound=computeCollisionTime(cInterval(I.i[0],m),tc);
    if (rootFound) return true;

    // no root of the first time sub-interval, check the second one

    return computeCollisionTime(cInterval(m,I.i[1]),tc);

}
```

The bounds on the function f are computed using interval arithmetic, as explained in Section 3. Assume, for example, that we want to bound the function in the edge/edge continuous collision detection equation (19) over the time interval I. Assume, first, that we know some bounds on the coordinates of the vertices involved in the test. Precisely, let `boundsA`, `boundsB`, `boundsC` and `boundsD` denote the three-dimensional interval vectors (`cIAVector3` objects) which bound the coordinates of $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ and $\mathbf{d}$ over the time interval I, respectively. The bounds `boundsF` on the function f are easily determined:

```
cInterval boundsF=(boundsC-boundsA)|((boundsB-boundsA)^(boundsD-boundsC));
```

As can been seen, the computation of the bounds is simply the interval counterpart of the evaluation of the function involved in the edge/edge test.

The bounds on the coordinates of the vertices are computed in a similar way. Assume for example that the vertex $\mathbf{a}$ belongs to a rigid body which moves according to the linear interpolation characterized by equations (4) and (5). The coordinates of the position vector $\mathbf{T}(t)$ are linear functions of time, and we can easily determine bounds for them, over any time interval I. Similarly, the components of the orientation matrix $\mathbf{R}(t)$ are simple trigonometric functions, which can be easily bounded (these functions are actually very similar to the one given in example in Section 3). Let `aLocal` denote the coordinates of $\mathbf{a}$ in the local frame of the rigid body. The variable `aLocal` is a *point* vector interval stored in a `cIAVector3` object[3]:

[3]For optimization purposes, a `cVector3` class can be implemented to contain point interval vectors, as well as a multiplication between a `cIAMatrix33` object and a `cVector3` object. This special multiplication can perform more

```
cIAVector3 aLocal(xA,xA,yA,yA,zA,zA);
```

Denoting by `boundsT` the `cIAVector3` object which bounds the coordinates of $\mathbf{T}(t)$, and by `boundsR` the `cIAMatrix33` object which bounds the components of $\mathbf{R}(t)$, over the time interval I, the bounds `boundsA` on $\mathbf{a}$ are simply:

```
cIAvector3 boundsA=boundsR*aLocal+boundsT;
```

For articulated bodies, the bounds are computed in a similar way. Assume we have a `cIAMatrix44` class which can contain 4×4 interval matrices. The `cIAMatrix44` objects are designed to contain 4×4 interval homogeneous position matrices. Assume also that right-multiplying them by a `cIAVector3` object produces the interval counterpart of the expected real multiplication, which applies a rotation and a translation to a vector[4].

Let the `cIAMatrix44` objects `boundsP1`, `boundsP2`, ..., `boundsPi` respectively denote the bounds on the position matrices $\mathbf{P}_1^0(t)$, $\mathbf{P}_2^1(t)$, ..., $\mathbf{P}_i^{i-1}(t)$ over the time interval I. Assume `aLocal` is a point interval vector which contains the coordinates of the vertex $\mathbf{a}$ in the local frame of link i. The bounds `boundsA` on the coordinates of $\mathbf{a}$ in the world frame, over the time interval I, are:

```
cIAVector3 boundsA=boundsP1*boundsP2*...*boundsPi*aLocal;
```

Note that, for efficiency purposes, it is preferable to perform the multiplications from right to left, so that only matrix-vector multiplications have to be computed. In general, in order to further reduce the complexity of the evaluation of interval position matrices, a *simultaneous resolution* scheme can be used [RKLM04].

It should now be clear why the choice of the arbitrary in-between motion has a huge impact on the overall efficiency of the continuous collision detection algorithm. The arbitrary in-between motion is going to be evaluated several times whenever some bounds on a continuous collision detection function are needed. If acceptable in the application, it can be advised, for example, to use an in-between motion which reduces the collision detection equations to polynomial equations [Can86, RKC00].

5 Continuous overlap tests between bounding volumes

In order to avoid performing all possible elementary tests for any object pair, many collision detection algorithms rely on *bounding-volume hierarchies*. Basically, if two objects are enclosed in bounding volumes which don't overlap, then it is known for sure that they don't collide. Hierarchies of bounding volumes are used to recursively perform such overlap tests which can conservatively cull away large parts of the objects when testing for a collision. Typical bounding volumes used for collision detection include spheres [Qui94, Hub95, RKK97, BS02], axis-aligned bounding boxes (AABBs) [VDB98], oriented bounding boxes (OBBs) [Got99, GLM96], and k-dops [KHMSZ98].

efficiently than the regular interval matrix vector multiplication, since fewer branching operations are required. For clarity, this is not actually described in these notes.

[4]These 4×4 interval matrices are introduced to simplify the expression of the computation of `boundsA`, but the equivalent interval operations can be performed using `cIAMatrix33` and `cIAVector3` objects.

Since we want to perform continuous collision detection between objects, we need to design a continuous overlap test between two bounding volumes. Precisely, we need to determine whether two bounding volumes will overlap during the timestep.

What makes the task easier is that it is not necessary to perform an *exact* test. We need to be sure that we do not miss an overlap between two bounding volumes during the timestep, but it is acceptable to declare that two bounding volumes overlap when they don't. The error will be captured later by smaller bounding volumes in the hierarchy, or ultimately by the elementary continuous collision detection tests. Such a test is called *conservative*.

In the following, we describe continuous overlap tests between spheres, axis-aligned bounding boxes, and oriented bounding boxes.

5.1 Spheres

Assume spheres are used as bounding volumes. Let $\mathbf{c}_1$ and $\mathbf{c}_2$ denote the centers of the spheres, and let r_1 and r_2 denote the radii of the spheres.

The spheres overlap if and only if the distance between their centers is smaller than the sum of their radii:

$$||\mathbf{c}_1\mathbf{c}_2|| \leqslant r_1 + r_2,$$

or, equivalently, if and only if

$$(\mathbf{c}_2 - \mathbf{c}_1)^2 \leqslant (r_1 + r_2)^2 . \tag{21}$$

Using interval arithmetic, we can design a conservative test to bound the left member of inequality (21) on any time interval I. If the lower bound of the left member is greater than the right member, then we know for sure that the distance between the centers is greater than the sum of the radii during the whole time interval I, in which case it is safe to declare that the spheres won't overlap during this time interval.

If the lower bound of the left member is smaller than the right member, however, there might be an overlap during the time interval, and we conservatively declare so.

The bounds on the left member are obtained as previously, by first bounding the coordinates of the center of the spheres, and then performing the interval counterpart of the function in the left member: assuming the bounds on the centers are two `cIAVector3` objects, `boundsC1` and `boundsC2`, some bounds `boundsLeft` on the left member are:

```
cInterval boundsLeft=(boundsC2-boundsC1)|(boundsC2-boundsC1);
```

5.2 Axis-aligned bounding boxes

Axis-aligned bounding volumes are typically recomputed at the beginning of each time step in discrete methods. Assuming these boxes are attached to the bodies during the timestep, they simply become *oriented* bounding boxes, as they loose their axis-aligned characteristic while the objects move. Consequently, the appropriate continuous overlap test in that case is the one between two oriented bounding boxes, described in the next section.

However, it is simple to obtain axis-aligned boxes which bound an object during a whole time interval. Indeed, *any three-dimensional vector interval is actually an axis-aligned bounding box.* Assume we determine bounds on a vertex motion during a time interval. By definition, since the bounds are computed coordinate per coordinate, we have actually obtained an axis-aligned box which

bounds the moving vertex during the whole time interval. We can thus easily determine AABBs which bound the moving object during the whole time interval.

Note that the AABBs can be obtained from a simplified version of the object geometry, provided this simplified version contains the original object. Assume, for example, that the object is included in a sphere. Using interval arithmetic, the bounds on the coordinates of the center of the sphere can be obtained easily. These bounds are in fact an AABB which encloses the moving center of the sphere during the time interval. Enlarging this AABB by the radius of the sphere results in an AABB which contains the sphere, and thus the object, during the whole time interval.

More generally, assume the object is enclosed in the convex hull of a set of points. An AABB can be obtained for each of these points using interval arithmetic. An AABB which contains all these AABBs is guaranteed to contain the object during the whole time interval.

When these dynamically generated AABBs have been computed, the traditional discrete AABB/AABB test can be used to conservatively determine whether the objects are going to overlap or not during the time interval.

5.3 Oriented bounding boxes

Let us now proceed to the case of oriented bounding boxes (OBBs). For a rigid object, a hierarchy of OBBs can be computed offline. Similarly, for an articulated model composed of rigid links, a hierarchy of OBBs can be computed offline for each link.

The goal is thus to (conservatively) determine whether the boxes are going to overlap during the time interval.

A well-known overlap test for oriented bounding boxes is the one which relies upon the separating axis theorem [GLM96]. Lets assume that the first OBB is described by three axes $\mathbf{e}_1$, $\mathbf{e}_2$ and $\mathbf{e}_3$, a center $\mathbf{T}_A$, and its half-sizes along its axes a_1, a_2 and a_3. Similarly, assume the second OBB is described by its axes $\mathbf{f}_1$, $\mathbf{f}_2$ and $\mathbf{f}_3$, its center $\mathbf{T}_B$, and its half-sizes along its axes b_1, b_2 and b_3. The separating axis theorem states that two static OBBs overlap if and only if all of fifteen separating axis tests fail. A separating test is simple: an axis $\mathbf{a}$ separates the OBBs if and only if:

$$|\mathbf{a} \cdot \mathbf{T}_A\mathbf{T}_B| > \sum_{i=1}^{3} a_i|\mathbf{a} \cdot \mathbf{e}_i| + \sum_{i=1}^{3} b_i|\mathbf{a} \cdot \mathbf{f}_i|. \tag{22}$$

This test is performed for fifteen axes at most. The sufficient set of fifteen axes is:

$$\{\mathbf{e}_i,\ \mathbf{f}_j,\ \mathbf{e}_i \wedge \mathbf{f}_j,\ 1 \leqslant i \leqslant 3,\ 1 \leqslant j \leqslant 3\} \tag{23}$$

Intuitively, the left member of inequality (22) is the projected distance between the two centers of the boxes in the direction of $\mathbf{a}$, while the right member is the sum of the projected radiuses of the boxes, in the same direction (cf. Figure 11).

We can extend the discrete OBB/OBB overlap test to the continuous domain using interval arithmetic [RKC02b]. Since each member of inequality (22) is a function of time depending on the specific arbitrary in-between motion, we can use interval arithmetic to bound both members over a time interval I. When the lower bound on the left member is larger than the upper bound on the right member, the axis $\mathbf{a}$ separates the boxes during the entire time interval I, and the pair of boxes is discarded, since we know for sure that the boxes will not overlap during the time interval.

As before, once the bounds on the elements involved in the test have been computed, the bounds on the two members are determined easily. Denoting by `boundsA`, `boundsE1`, `boundsE2`, `boundsE3`, `boundsF1`, `boundsF2`, `boundsF3`, `boundsTA`, and `boundsTB` the `cIAVector3` objects which contain the bounds on $\mathbf{a}$, $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$, $\mathbf{f}_1$, $\mathbf{f}_2$, $\mathbf{f}_3$, $\mathbf{T}_A$, and $\mathbf{T}_B$, respectively, the lower bound on the left member is:

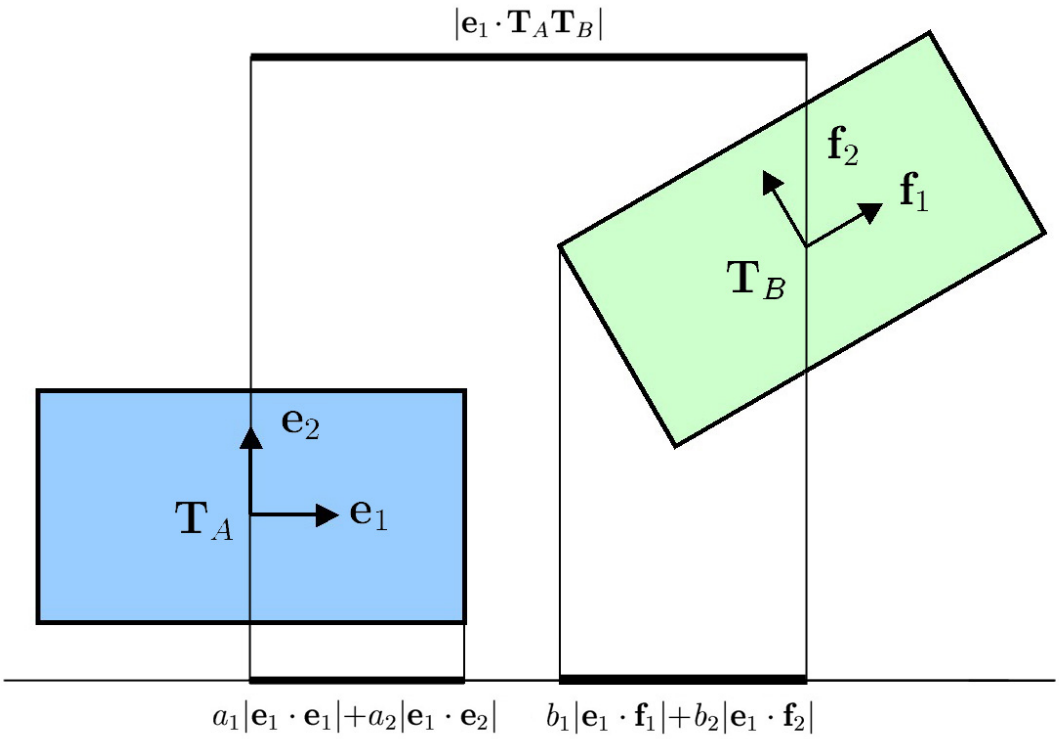


Figure 11: The axis $\mathbf{e}_1$ separates the two oriented bounding boxes since, in the axis direction, the projected distance between the centers of the boxes $|\mathbf{e}_1 \cdot \mathbf{T}_A\mathbf{T}_B|$ is larger than the sum of the projected radii of the boxes, $(a_1|\mathbf{e}_1 \cdot \mathbf{e}_1| + a_2|\mathbf{e}_1 \cdot \mathbf{e}_2|) + (b_1|\mathbf{e}_1 \cdot \mathbf{f}_1| + b_2|\mathbf{e}_1 \cdot \mathbf{f}_2|)$.

```
double lBoundLeft=(boundsA|(boundsTB-boundsTA)).getAbsLower();
```

where the function `getAbsLower()` returns the lower bound on the absolute value of an interval. Similarly, the upper bound on the right member is:

```
double uBoundRight=a1*(boundsA|boundsE1).getAbsUpper()+
                   a2*(boundsA|boundsE2).getAbsUpper()+
                   a3*(boundsA|boundsE3).getAbsUpper()+
                   b1*(boundsA|boundsF1).getAbsUpper()+
                   b2*(boundsA|boundsF2).getAbsUpper()+
                   b3*(boundsA|boundsF3).getAbsUpper();
```

where the function `getAbsUpper()` returns the upper bound on the absolute value of an interval. The functions `getAbsLower()` and `getAbsUpper()` are provided in the appendix.

Note that, due to the special form of the axis $\mathbf{a}$, this last computation can actually be simplified, as in the original discrete test (cf. [GLM96]). For example, when $\mathbf{a} = \mathbf{e}_1$, we know that

$$\mathbf{a}.\mathbf{e}_1 = 1$$

and

$$\mathbf{a}.\mathbf{e}_2 = \mathbf{a}.\mathbf{e}_2 = 0.$$

Since this holds at all times, it is not necessary to actually compute the first three interval dot products in this case (this would produce conservative but loose bounds).

Recall also that, since the axes of the boxes are vectors and not vertices, the translation component must not be included when computing the bounds on their coordinates. Assume, for example, that the axis $\mathbf{e}_1$ belongs to a box attached to a rigid body which moves according to the linear interpolation characterized by equations (4) and (5). Let `eLocal` denote the `cIAVector3` object which contains the components of the axis $\mathbf{e}_1$ in the local frame of the rigid body, and let `boundsR` denote the `cIAMatrix33` object which contain the bounds on the orientation matrix $\mathbf{R}(t)$. The `cIAVector3` object `boundsE1` is computed as follow:

```
cIAVector3 boundsE=boundsR*eLocal;
```

5.4 Remarks

Again, we have used interval arithmetic to perform the continuous tests. As opposed to what happens with the elementary tests, though, the interval computations which occur during the continuous overlap tests between bounding volumes are generally performed once only, over the full time interval $[0, 1]$: the bounds on the functions involved in the tests are computed once and for all on the time interval $[0, 1]$ and these bounds are used to conservatively determine the overlap status of the bounding volumes during this time interval. This comes from the fact that we do not really need to know *when* the bounding volumes will begin to overlap (although that might be a useful information), but only *if* they are going to overlap during the given time interval.

However, we have noted in Section 3 that the bounds obtained using interval arithmetic are generally not tight. This is especially the case when the total amount of rotation is very large over

one single time step[5] In order to reduce the conservativeness of the test, which would lead to declare that the bounding volumes overlap too often and would make us loose the benefit of using bounding-volume hierarchies, it is best to subdivide the time interval one or several times when the total amount of rotation is very large. The cost of replacing the single test by several tests on smaller time sub-intervals is usually compensated by the early culling, which prevents the need to unnecessarily go further down the hierarchies of bounding volumes.

For articulated bodies, an intermediate culling step can be added in order to prevent the increased conservativeness of interval arithmetic when the depth of the articulated model increases [RKLM04].

6 Conclusion

These notes have presented an overview of some recent work on continuous collision detection methods, which guarantee consistent simulations by computing the time of first contact and the contact state for colliding objects. We have described techniques to perform continuous collision detection for rigid and articulated bodies. In the following pages, an appendix provides some code bits which, although not optimized, should help the reader start his or her own implementation of continuous collision detection.

[5]Interval arithmetic produces tight bounds when the motion is a pure linear translation thanks to the monotony of the functions involved.

Appendix

This appendix provides some code bits to help the reader start his or her own implementation. The code bits are most certainly not fully optimized, but should constitute a good starting point.

Conversion from a rotation matrix to an angle/axis pair

Here is a code bit which converts a 3×3 rotation matrix to an angle/axis pair. It uses an intermediate conversion to a quaternion. Some measures are taken in the code to help prevent round-off errors.

```
// input

double mat[3][3]; // the rotation matrix

// output

double rotAngle; // the rotation angle
double rotAxis[3]; // the rotation axis

// conversion to a quaternion

double x,y,z,w;
double w2=0.25*(1.0+mat[0][0]+mat[1][1]+mat[2][2]);

if (w2>1e-6) {

    w=sqrt(w2);
    x=(mat[2][1]-mat[1][2])/(4.0*w);
    y=(mat[0][2]-mat[2][0])/(4.0*w);
    z=(mat[1][0]-mat[0][1])/(4.0*w);

}
else {

    w=0.0;
    double x2=0.5*(mat[1][1]+mat[2][2]);

    if (x2>1e-6) {

        x=sqrt(x2);
        y=mat[1][0]/(2.0*x);
        z=mat[2][0]/(2.0*x);

    }
    else {

        x=0.0;
        double y2=0.5*(1.0-mat[2][2]);
```

```
        if (y2>1e-6) {

            y=sqrt(y2);
            z=mat[2][1][0]/(2.0*y);

        }
        else {

            y=0.0;
            z=1.0;

        }
    }
}

// normalize quaternion

double invnormq=1.0/sqrt(w*w+x*x+y*y+z*z);
w*=invnormq;
x*=invnormq;
y*=invnormq;
z*=invnormq;

// conversion to angle/axis form

if (w>1.0) w=1.0;

rotAngle=2.0*acos(w);

if (rotAngle<1e-6) {

    rotAngle=0.0;
    rotAxis[0]=1.0;
    rotAxis[1]=0.0;
    rotAxis[2]=0.0;

}
else {

    double invnorm=1.0/sqrt(x*x+y*y+z*z);
    rotAxis[0]=x*invnorm;
    rotAxis[1]=y*invnorm;
    rotAxis[2]=z*invnorm;

}
```

Conversion from object velocities to a screw motion

An equivalent screw motion can be obtained from the object's translational velocity **s** and the object's rotational velocities described by the rotation angle ω (in radians) and the rotation axis **u**. Essentially, the method consists in decomposing the translation into two terms, one which is parallel to the rotation axis, and one which is orthogonal to it. The orthogonal component is then suppressed by translating the rotation axis. This method is implemented in the following code bit.

```
// input

double s[3]; // the object translational velocity
double rotAngle; // the rotation angle
double rotAxis[3]; // the rotation axis

// output

double sScrew; // the amount of translation in the screw motion
double o[3]; // a point on the screw axis
double w; // the screw angle
double u[3]; // the screw axis

w=rotAngle;
if ((w>=-1e-6)&&(w<=1e-6)) w=0;

if (w==0) { // no rotation

    sScrew=sqrt(s[0]*s[0]+s[1]*s[1]+s[2]*s[2]);
    if ((sScrew>=-1e-6)&&(sScrew<=1e-6)) sScrew=0;

    if (sScrew==0) { // no motion

        o[0]=o[1]=o[2]=0.0;
        u[0]=u[1]=u[2]=0.0;

   }
   else {

        u[0]=s[0]/sScrew;
        u[1]=s[1]/sScrew;
        u[2]=s[2]/sScrew;
        o[0]=o[1]=o[2]=0.0;

   }

}
else { // rotation

    // compute the screw axis

    u[0]=rotAxis[0];
```

```
        u[1]=rotAxis[1];
        u[2]=rotAxis[2];

        // decompose the translation

        sScrew=s[0]*u[0]+s[1]*u[1]+s[2]*u[2]; // component along the screw axis

        double n1[3]; // component orthogonal to u
        n1[0]=s[0]-sScrew*u[0];
        n1[1]=s[1]-sScrew*u[1];
        n1[2]=s[2]-sScrew*u[2];

        double n1Norm2=n1[0]*n1[0]+n1[1]*n1[1]+n1[2]*n1[2];

        if (n1Norm2>1e-6) {

            n1Norm2=sqrt(n1Norm2);
            n1[0]/=n1Norm2;
            n1[1]/=n1Norm2;
            n1[2]/=n1Norm2;

            double n1Orth[3]; // n1Orth=u^n1
            n1Orth[0]=u[1]*n1[2]-u[2]*n1[1];
            n1Orth[1]=u[2]*n1[0]-u[0]*n1[2];
            n1Orth[2]=u[0]*n1[1]-u[1]*n1[0];

            double n2x=cos(0.5*w);
            double n2y=sin(0.5*w);

            double sn1=s[0]*n1[0]+s[1]*n1[1]+s[2]*n1[2];

            o[0]=0.5*sn1*(n1[0]+n2x/n2y*n1orth[0]);
            o[1]=0.5*sn1*(n1[1]+n2x/n2y*n1orth[1]);
            o[2]=0.5*sn1*(n1[2]+n2x/n2y*n1orth[2]);

        }

        else o[0]=o[1]=o[2]=0.0;

    }
```

In order to compute the transformation matrix $\mathbf{P}_V$ from the global frame to a local frame of the screw motion, the following code bit can be used. In the local frame, the screw axis is Oz. Positive translation is along z+. So the whole computation does this: align the Oz axis along the screw axis (thanks to $\mathbf{u}$), and then find two other orthogonal axes to complete the basis.

```
    // input

    double o[3]; // a point on the screw axis
```

```
    double u[3]; // the screw axis

    // output

    double R[3][3]; // the orientation component
    double T[3]; // the translation component

    if ((u[0]>0.1)||(u[0]<-0.1)||(u[1]>0.1)||(u[1]<-0.1)) {

        double n=sqrt(u[0]*u[0]+u[1]*u[1]);
        double invn=1.0/n;
        R[0][0]=-u[1]*invn;
        R[0][1]=u[0]*invn;
        R[0][2]=0.0;
        R[1][0]=-u[0]*invn*u[2];
        R[1][1]=-u[1]*invn*u[2];
        R[1][2]=n;
        R[2][0]=u[0];
        R[2][1]=u[1];
        R[2][2]=u[2];
        T[0]=o[0]*u[1]*invn-o[1]*u[0]*invn;
        T[1]=-(o[0]*R[1][0]+o[1]*R[1][1]+o[2]*n);
        T[2]=-(o[0]*u[0]+o[1]*u[1]+o[2]*u[2]);

    }
    else {

        double sgnu2=(u[2]>=0?1:-1);
        R[0][0]=1;
        R[1][1]=sgnu2;
        R[2][2]=sgnu2;
        R[1][0]=R[2][0]=R[0][1]=R[2][1]=R[0][2]=R[1][2]=0.0;
        T[0]=-o[0];
        T[1]=-sgnu2*o[1];
        T[2]=-sgnu2*o[2];

    }
```

Interval arithmetic structures

Here are some basic structures to perform interval arithmetic computations. For clarity, these structures are not optimized (e.g. branching can be reduced in the interval multiplication). Similarly, no measure is taken to make sure the computations are effectively conservative (i.e. switching the rounding mode [Sny92]).

We first design an interval, which handles some basic interval operations.

```
class cInterval {
```

```
public:

    double i[2];

    cInterval() { ; }
    cInterval(double v) { i[0]=i[1]=v; }
    cInterval (double ll, double rr) { i[0]=ll;i[1]=rr; }

    cInterval   operator+(const cInterval &in) {

        return cInterval(i[0]+in.i[0],i[1]+in.i[1]);

    }
    cInterval   operator-(const cInterval &in) {

        return cInterval(i[0]-in.i[1],i[1]-in.i[0]);

    }
    cInterval&  operator+=(const cInterval &in) {

        i[0]+=in.i[0];i[1]+=in.i[1];return *this;

    }
    cInterval&  operator-=(const cInterval &in) {

        i[0]-=in.i[1];i[1]-=in.i[0];return *this;

    }
    cInterval   operator*(const cInterval &in) {

        register double temp,vmin,vmax;
        vmin=vmax=i[0]*in.i[0];

        temp=i[0]*in.i[1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
        temp=i[1]*in.i[0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
        temp=i[1]*in.i[1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
        return cInterval(vmin,vmax);

    }

    cInterval   operator/(const cInterval &in) {

        // assumes that the interval does not contain 0

        return *this*cInterval(1.0/in.i[1],1.0/in.i[0]);

    }

    double      getAbsLower() {
```

```
        // returns the lower bound on the absolute value of the interval

        if (i[0]>=0) return i[0];
        if (i[1]>=0) return 0;
        return -i[1];

    }

    double      getAbsUpper() {

        // returns the upper bound on the absolute value of the interval

        if (i[0]+i[1]>=0) return i[1];
        return -i[0];

    }

};
```

Using this interval, we can now easily build an interval vector class. Note that we use a dual representation to facilitate the access to the interval components of the vectors. In particular, two essential interval operations, the interval dot product and the interval cross product, are much easier to code using the interval class.

```
class cIAVector3 {

public:

    // we use a dual representation to store the interval components

    union {
        struct {
            cInterval x,y,z; // one interval per coordinate
        };
        struct {
            double  v[3][2]; // array version
        };
    };


    cIAVector3() {}
    cIAVector3(double xl, double xu, double yl, double yu, double zl, double zu) {

        v[0][0]=xl;v[0][1]=xu;
        v[1][0]=yl;v[1][1]=yu;
        v[2][0]=zl;v[2][1]=zu;
```

```
}
cIAVector3(cInterval &nx, cInterval &ny, cInterval &nz) { x=nx;y=ny;z=nz; }
cIAVector3(cVector3 &u) {

    v[0][0]=v[0][1]=u.v[0];v[1][0]=v[1][1]=u.v[1];v[2][0]=v[2][1]=u.v[2];

}

// operators

cIAVector3      operator+(const cIAVector3 &u) const {

    return cIAVector3(v[0][0]+u.v[0][0],v[0][1]+u.v[0][1],
                      v[1][0]+u.v[1][0],v[1][1]+u.v[1][1],
                      v[2][0]+u.v[2][0],v[2][1]+u.v[2][1]);

}

cIAVector3&     operator+=(const cIAVector3 &u) {

    v[0][0]+=u.v[0][0];v[0][1]+=u.v[0][1];
    v[1][0]+=u.v[1][0];v[1][1]+=u.v[1][1];
    v[2][0]+=u.v[2][0];v[2][1]+=u.v[2][1];
    return *this;

}

cIAVector3      operator-(const cIAVector3 &u) const {

    return cIAVector3(v[0][0]-u.v[0][1],v[0][1]-u.v[0][0],
                      v[1][0]-u.v[1][1],v[1][1]-u.v[1][0],
                      v[2][0]-u.v[2][1],v[2][1]-u.v[2][0]);

}

cIAVector3&     operator-=(const cIAVector3 &u) {

    v[0][0]-=u.v[0][1];v[0][1]-=u.v[0][0];
    v[1][0]-=u.v[1][1];v[1][1]-=u.v[1][0];
    v[2][0]-=u.v[2][1];v[2][1]-=u.v[2][0];
    return *this;

}

cIAVector3&     operator=(const cVector3 &u) {

    v[0][0]=v[0][1]=u.v[0];v[1][0]=v[1][1]=u.v[1];v[2][0]=v[2][1]=u.v[2];
    return *this;

}
```

```
    cInterval       operator|(const cIAVector3 &u) {

        // interval dot product

        return  x*u.x+y*u.y+z*u.z;

    }
    cIAVector3      operator^(const cIAVector3 &u) {

        // interval cross product

        return cIAVector3(y*u.z-z*u.y,z*u.x-x*u.z,x*u.y-y*u.x);

    }

};
```

Finally, we can build an interval matrix class. Some useful accessors are included. For convenience, we also provide a non-optimized matrix-vector and matrix-matrix interval multiplication (as in the basic interval multiplication, branching can be reduced).

```
cIAVector3  cIAMatrix33::operator*(cIAVector3& v) {

    // Interval method : r=m*v

    double xl,xu,yl,yu,zl,zu;
    register double temp,vmin,vmax;

    // r.v[0]

    vmin=vmax=m[0][0][0]*v.v[0][0];
    temp=m[0][0][0]*v.v[0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[0][0][1]*v.v[0][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[0][0][1]*v.v[0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    xl=vmin;xu=vmax;

    vmin=vmax=m[0][1][0]*v.v[1][0];
    temp=m[0][1][0]*v.v[1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[0][1][1]*v.v[1][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[0][1][1]*v.v[1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    xl+=vmin;xu+=vmax;

    vmin=vmax=m[0][2][0]*v.v[2][0];
    temp=m[0][2][0]*v.v[2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[0][2][1]*v.v[2][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[0][2][1]*v.v[2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    xl+=vmin;xu+=vmax;
```

```
    // r.v[1]

    vmin=vmax=m[1][0][0]*v.v[0][0];
    temp=m[1][0][0]*v.v[0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[1][0][1]*v.v[0][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[1][0][1]*v.v[0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    yl=vmin;yu=vmax;

    vmin=vmax=m[1][1][0]*v.v[1][0];
    temp=m[1][1][0]*v.v[1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[1][1][1]*v.v[1][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[1][1][1]*v.v[1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    yl+=vmin;yu+=vmax;

    vmin=vmax=m[1][2][0]*v.v[2][0];
    temp=m[1][2][0]*v.v[2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[1][2][1]*v.v[2][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[1][2][1]*v.v[2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    yl+=vmin;yu+=vmax;

    // r.v[2]

    vmin=vmax=m[2][0][0]*v.v[0][0];
    temp=m[2][0][0]*v.v[0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[2][0][1]*v.v[0][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[2][0][1]*v.v[0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    zl=vmin;zu=vmax;

    vmin=vmax=m[2][1][0]*v.v[1][0];
    temp=m[2][1][0]*v.v[1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[2][1][1]*v.v[1][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[2][1][1]*v.v[1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    zl+=vmin;zu+=vmax;

    vmin=vmax=m[2][2][0]*v.v[2][0];
    temp=m[2][2][0]*v.v[2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[2][2][1]*v.v[2][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[2][2][1]*v.v[2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    zl+=vmin;zu+=vmax;

    return cIAVector3(xl,xu,yl,yu,zl,zu);

}

cIAMatrix33 cIAMatrix33::operator*(cIAMatrix33& mat) {


    // Interval method : res=m*mat

    register double temp,vmin,vmax;
```

```
double res[3][3][2];

// res[0][0]

vmin=vmax=m[0][0][0]*mat.m[0][0][0];
temp=m[0][0][0]*mat.m[0][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][0][1]*mat.m[0][0][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][0][1]*mat.m[0][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[0][0][0]=vmin;res[0][0][1]=vmax;

vmin=vmax=m[0][1][0]*mat.m[1][0][0];
temp=m[0][1][0]*mat.m[1][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][1][1]*mat.m[1][0][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][1][1]*mat.m[1][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[0][0][0]+=vmin;res[0][0][1]+=vmax;

vmin=vmax=m[0][2][0]*mat.m[2][0][0];
temp=m[0][2][0]*mat.m[2][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][2][1]*mat.m[2][0][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][2][1]*mat.m[2][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[0][0][0]+=vmin;res[0][0][1]+=vmax;

// res[1][0]

vmin=vmax=m[1][0][0]*mat.m[0][0][0];
temp=m[1][0][0]*mat.m[0][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[1][0][1]*mat.m[0][0][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[1][0][1]*mat.m[0][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[1][0][0]=vmin;res[1][0][1]=vmax;

vmin=vmax=m[1][1][0]*mat.m[1][0][0];
temp=m[1][1][0]*mat.m[1][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[1][1][1]*mat.m[1][0][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[1][1][1]*mat.m[1][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[1][0][0]+=vmin;res[1][0][1]+=vmax;

vmin=vmax=m[1][2][0]*mat.m[2][0][0];
temp=m[1][2][0]*mat.m[2][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[1][2][1]*mat.m[2][0][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[1][2][1]*mat.m[2][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[1][0][0]+=vmin;res[1][0][1]+=vmax;

// res[2][0]

vmin=vmax=m[2][0][0]*mat.m[0][0][0];
temp=m[2][0][0]*mat.m[0][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[2][0][1]*mat.m[0][0][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[2][0][1]*mat.m[0][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[2][0][0]=vmin;res[2][0][1]=vmax;

vmin=vmax=m[2][1][0]*mat.m[1][0][0];
```

```
temp=m[2][1][0]*mat.m[1][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[2][1][1]*mat.m[1][0][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[2][1][1]*mat.m[1][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[2][0][0]+=vmin;res[2][0][1]+=vmax;

vmin=vmax=m[2][2][0]*mat.m[2][0][0];
temp=m[2][2][0]*mat.m[2][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[2][2][1]*mat.m[2][0][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[2][2][1]*mat.m[2][0][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[2][0][0]+=vmin;res[2][0][1]+=vmax;

// res[0][1]

vmin=vmax=m[0][0][0]*mat.m[0][1][0];
temp=m[0][0][0]*mat.m[0][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][0][1]*mat.m[0][1][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][0][1]*mat.m[0][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[0][1][0]=vmin;res[0][1][1]=vmax;

vmin=vmax=m[0][1][0]*mat.m[1][1][0];
temp=m[0][1][0]*mat.m[1][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][1][1]*mat.m[1][1][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][1][1]*mat.m[1][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[0][1][0]+=vmin;res[0][1][1]+=vmax;

vmin=vmax=m[0][2][0]*mat.m[2][1][0];
temp=m[0][2][0]*mat.m[2][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][2][1]*mat.m[2][1][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][2][1]*mat.m[2][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[0][1][0]+=vmin;res[0][1][1]+=vmax;

// res[1][1]

vmin=vmax=m[1][0][0]*mat.m[0][1][0];
temp=m[1][0][0]*mat.m[0][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[1][0][1]*mat.m[0][1][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[1][0][1]*mat.m[0][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[1][1][0]=vmin;res[1][1][1]=vmax;

vmin=vmax=m[1][1][0]*mat.m[1][1][0];
temp=m[1][1][0]*mat.m[1][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[1][1][1]*mat.m[1][1][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[1][1][1]*mat.m[1][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[1][1][0]+=vmin;res[1][1][1]+=vmax;

vmin=vmax=m[1][2][0]*mat.m[2][1][0];
temp=m[1][2][0]*mat.m[2][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[1][2][1]*mat.m[2][1][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[1][2][1]*mat.m[2][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[1][1][0]+=vmin;res[1][1][1]+=vmax;
```

```
// res[2][1]

vmin=vmax=m[2][0][0]*mat.m[0][1][0];
temp=m[2][0][0]*mat.m[0][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[2][0][1]*mat.m[0][1][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[2][0][1]*mat.m[0][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[2][1][0]=vmin;res[2][1][1]=vmax;

vmin=vmax=m[2][1][0]*mat.m[1][1][0];
temp=m[2][1][0]*mat.m[1][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[2][1][1]*mat.m[1][1][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[2][1][1]*mat.m[1][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[2][1][0]+=vmin;res[2][1][1]+=vmax;

vmin=vmax=m[2][2][0]*mat.m[2][1][0];
temp=m[2][2][0]*mat.m[2][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[2][2][1]*mat.m[2][1][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[2][2][1]*mat.m[2][1][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[2][1][0]+=vmin;res[2][1][1]+=vmax;

// res[0][2]

vmin=vmax=m[0][0][0]*mat.m[0][2][0];
temp=m[0][0][0]*mat.m[0][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][0][1]*mat.m[0][2][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][0][1]*mat.m[0][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[0][2][0]=vmin;res[0][2][1]=vmax;

vmin=vmax=m[0][1][0]*mat.m[1][2][0];
temp=m[0][1][0]*mat.m[1][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][1][1]*mat.m[1][2][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][1][1]*mat.m[1][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[0][2][0]+=vmin;res[0][2][1]+=vmax;

vmin=vmax=m[0][2][0]*mat.m[2][2][0];
temp=m[0][2][0]*mat.m[2][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][2][1]*mat.m[2][2][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[0][2][1]*mat.m[2][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[0][2][0]+=vmin;res[0][2][1]+=vmax;

// res[1][2]

vmin=vmax=m[1][0][0]*mat.m[0][2][0];
temp=m[1][0][0]*mat.m[0][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[1][0][1]*mat.m[0][2][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[1][0][1]*mat.m[0][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
res[1][2][0]=vmin;res[1][2][1]=vmax;

vmin=vmax=m[1][1][0]*mat.m[1][2][0];
temp=m[1][1][0]*mat.m[1][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
temp=m[1][1][1]*mat.m[1][2][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
```

```
    temp=m[1][1][1]*mat.m[1][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    res[1][2][0]+=vmin;res[1][2][1]+=vmax;

    vmin=vmax=m[1][2][0]*mat.m[2][2][0];
    temp=m[1][2][0]*mat.m[2][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[1][2][1]*mat.m[2][2][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[1][2][1]*mat.m[2][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    res[1][2][0]+=vmin;res[1][2][1]+=vmax;

    // res[2][2]

    vmin=vmax=m[2][0][0]*mat.m[0][2][0];
    temp=m[2][0][0]*mat.m[0][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[2][0][1]*mat.m[0][2][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[2][0][1]*mat.m[0][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    res[2][2][0]=vmin;res[2][2][1]=vmax;

    vmin=vmax=m[2][1][0]*mat.m[1][2][0];
    temp=m[2][1][0]*mat.m[1][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[2][1][1]*mat.m[1][2][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[2][1][1]*mat.m[1][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    res[2][2][0]+=vmin;res[2][2][1]+=vmax;

    vmin=vmax=m[2][2][0]*mat.m[2][2][0];
    temp=m[2][2][0]*mat.m[2][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[2][2][1]*mat.m[2][2][0];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    temp=m[2][2][1]*mat.m[2][2][1];if (temp<vmin) vmin=temp; else if (temp>vmax) vmax=temp;
    res[2][2][0]+=vmin;res[2][2][1]+=vmax;

    return cIAMatrix33(res);

}
```

References

[AP97] M. Anitescu and F. A. Potra. Formulating Dynamic Multi-Rigid-Body Contact Problems with Friction as Solvable Linear Complementarity Problems. Nonlinear Dynam. 14 (1997), no. 3, 231–247.

[APS99] M. Anitescu, F. A. Potra and D. E. Stewart. Time-stepping for Three-dimensional Rigid Body Dynamics. Computational Modeling of Contact and Friction. Comput. Methods Appl. Mech. Engrg. 177 (1999), no. 3-4, 183–197.

[Bar90] D. Baraff. Curved Surfaces and Coherence for Non-penetrating Rigid Body Simulations. In Computer Graphics (Proc.SIGGRAPH), volume 24, pages 19-28. ACM, August 1990. 1

[Bar94] D. Baraff. Fast Contact Force Computation for Nonpenetrating Rigid Bodies. In SIGGRAPH 94 Conference Proceedings, Annual Conference Series, pp 23-34. ACM SIGGRAPH, Addison Wesley, 1994.

[BS02] G. Bradshaw and C. O'Sullivan. Sphere-Tree Construction using Dynamic Medial Axis Approximation. In Proceedings of ACM Symposium on Computer Animation 2002. 20

[BW97] D. Baraff and A. Witkin. Partitioned Dynamics, Technical Report CMU-RI-TR-97-33, Robotics Institute, Carnegie Mellon University, 1997.

[Cam90] S. A. Cameron.collision detection by four-dimensional intersection testing. IEEE Trans. Robotics and Automation. 6, 3 (June 1990), pp 291-302.

[Can86] J. F. Canny. *collision detection for moving polyhedra.* IEEE Trans. Patt. Anal. Mach. Intell. 8,2 (March 1986), pp 200-209. 20

[Cha1831] M. Chasles. Note sur les Propriétés Générales du Système de Deux Corps Semblables Entre Eux, Placés d'une Manière Quelquonque Dans l'Espace; et sur le Déplacement Fini, ou Infiniment Petit d'un Corps Solide Libre. Bulletin des Sciences Mathematiques de Ferussac, XIV, pp. 321-336. 1831. 10

[CSB95] J. Colgate, M. Stanley, and J. Brown. Issues in the haptic display of tool use. In Int. Conf. on Intelligent Robots and Systems, (Pittsburgh), August 1995.

[Duf92] T. Duff. *Interval Arithmetic and Recursive Subdivision for Implicit Functions and Constructive Solid Geometry.* Computer Graphics, 26(2), July 1992, pp. 131-138.

[FMM77] Forsythe, G.E., Malcolm, M.A., and Moler, C.B., Computer Methods for Mathematical Computations, Prentice Hall, Inc., Englewood Cliffs, 1977. 1

[GRLM03] N. Govindaraju, S. Redon, Ming C. Lin and Dinesh Manocha. CULLIDE: Interactive Collision Detection between Complex Models in Large Environments using Graphics Hardware. ACM SIGGRAPH/ Eurographics Graphics Hardware Proceedings, 2003.

[Got99] S. Gottschalk. *collision queries using oriented bounding boxes.* PhD Thesis. 1999. 20

[GLM96] S. Gottschalk, M. C. Lin, and D. Manocha. OBB-Tree: A Hierarchical Structure for Rapid Interference Detection. In SIGGRAPH 96 Conference Proceedings, Annual Conference Series. ACM SIGGRAPH, Addison Wesley, August 1996. 20, 22, 24

[GMELM00] A. Gregory, A. Mascarenhas, S. Ehmann, M. Lin and D. Manocha. *Six degree-of-freedom haptic display of polygonal models.*In Proc. IEEE Visualization, 2000. 2

[Har99] Michael Hardt. Multibody Dynamical Algorithms, Numerical Optimal Control, with Detailed Studies in the Control of Jet Engine Compressors and Biped Walking Department of Electrical and Computer Engineering (Intelligent Systems, Robotics, and Control) University of California San Diego, June 1999.

[Hub95] P. M. Hubbard. *collision detection for interactive graphics applications.* Ph.D. Thesis, April 1995. 20

[JTT01] P. Jiménez, F. Thomas and C. Torras. 3D collision detection: a survey. Computers and Graphics, 25 (2), pp. 269-285, (April 2001), Pergamon Press / Elsevier Science.

[Kea96] R. B. Kearfott. Interval Computations: Introduction, Uses, and Resources, Euromath Bulletin 2 (1), pp. 95-112 (1996). 15

[KOLM02] Young J. Kim, Miguel A. Otaduy, Ming C. Lin, Dinesh Manocha. Fast Penetration Depth Computation for Physically-based Animation. ACM Symposium on Computer Animation, July 21-22, 2002. 2

[KHMSZ98] J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral and K. Zikan. Efficient collision Detection Using Bounding Volume Hierarchies of k-DOPs. IEEE Transactions on Visualization and Computer Graphics, March 1998, Volume 4, Number 1. 20

[LKCGC01] A. Lécuyer, A. Kheddar, S. Coquillart, L. Graux, and P. Coiffet, A Haptic Prototype for the Simulations of Aeronautics Mounting/Unmounting Operations. IEEE Int. Workshop on Robot-Human Interactive Communication, Bordeaux and Paris, France, 2001.

[LG98] M. C. Lin and S. Gottschalk. Collision detection between geometric models: a survey. In IMA Conference on Mathematics of Surfaces (San Diego (CA), May 1998), vol. 1, pp. 602– 608.

[Lot84] P. Lötstedt. Numerical simulation of time-dependent contact friction problems in rigid body mechanics. SIAM Journal of Scientific Statistical Computing, vol. 5, no. 2, pp. 370- 393, 1984.

[Moo62] R. E. Moore. Interval analysis and automatic error analysis in digital computation. PhD Thesis, Stanford University, October 1962. 15

[MW88] M. Moore and J. Wilhelms. collision Detection and Response for Computer Animation. In Computers Graphics (Proceedings of SIGGRAPH 88), Annual Conference Series, pp 289-298. ACM SIGGRAPH, August 1988. 1

[ODGK03] O'Sullivan ,C. Dingliana, J., Giang, T. Kaiser. Evaluating the Visual Fidelity of Physically Based Animations. M.K. ACM Transactions on Graphics. 22(3), Proceedings of SIGGRAPH 2003, July 2003.

[Qui94] S. Quinlan. Efficient distance computation between non-convex objects. In Proceedings of International Conference on Robotics and Automation, pp 3324-3329, 1994. 20

[Red04] S. Redon. Fast Continuous Collision Detection and Handling for Desktop Virtual Prototyping. To appear in Virtual Reality Journal (Springer Verlag).

[RKC00] S. Redon, A. Kheddar and S. Coquillart. An Algebraic Solution to the Problem of collision Detection for Rigid Polyhedral Objects. In Proceedings of IEEE International Conference on Robotics and Automation, pp 3733-3738, April 2000. 20

[RKC01] S. Redon, A. Kheddar and S. Coquillart. CONTACT: arbitrary in-between motions for continuous collision detection. In Proceedings of IEEE ROMAN'2001, Sep. 2001.

[RKC02a] S. Redon, A. Kheddar and S. Coquillart. Gauss' least constraint principle and rigid body simulations. In Proceedings of IEEE International Conference on Robotics and Automation, May 2002.

[RKC02b] S. Redon, A. Kheddar and S. Coquillart. Fast Continuous collision Detection between Rigid Bodies. In Proceedings of Eurographics 2002. September 2002. 22

[RKC02c] S. Redon, A. Kheddar and S. Coquillart. Hierarchical Back-Face Culling for collision Detection. In proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems. September 2002.

[RKLM04] S. Redon, Young J. Kim, Ming C. Lin and Dinesh Manocha. Fast Continuous Collision Detection for Articulated Models. In Proceedings of IEEE Solid Modeling 2004. 20, 25

[RKLMT04] S. Redon, Young J. Kim, Ming C. Lin, Dinesh Manocha and Jim Templeman. Interactive and Continuous Collision Detection for Avatars in Virtual Environment. In Proceedings of IEEE International Conference on Virtual Reality 2004.

[RKK97] D. C. Ruspini, K. Kolarov and O. Khatib. The Haptic Display of Complex Graphical Environments. Computer Graphics Proceedings, SIGGRAPH 97 pp 345-52 20

[SIS96] Yair Shapira, Moshe Israeli, Avram Sidi. Towards Automatic Multigrid Algorithms for SPD, Nonsymmetric and Indefinite Problems. Journal on Scientific Computing Volume 17, Number 2 pp. 439-453, 1996.

[Sny92] J. Snyder. Interval analysis for Computer Graphics. Computer Graphics, 26(2),pages 121-130, July 1992. 15, 30

[SWFCB93] J. Snyder, A. Woodbury, K. Fleischer, B. Currin, and A. Barr, Interval Methods for Multi-point collisions between Time-Dependent Curved Surfaces. Computer Graphics, 27(2), pp. 321-334, Aug. 1993.

[ST96] D. E. Stewart and J. C. Trinkle. An Implicit Time-Stepping Scheme for Rigid Body Dynamics with Inelastic collisions and Coulomb Friction. International Journal of Numerical Methods in Engineering, 39:2673-2691, 1996.

[TSHBS03] N. Tarrin, S. Coquillart, S. Hasegawa, L. Bouguila, M. Sato. The Stringed Haptic Workbench: a New Haptic Workbench Solution. In proceedings of Eurographics, September 2003.

[VDB98] G. Van den Bergen. *Efficient collision detection of complex deformable models using AABB trees.* Journal of Graphics Tools, 2(4):1-14, 1997. 20

[VHBZ90] Von Herzen, B., A.H. Barr, and H.R. Zatz, Geometric collisions for Time-Dependent Parametric Surfaces. Computer Graphics, 24(4), August 1990, pp. 39-48.

Modeling Dynamic Hair as a Continuum

Sunil Hadap*
R&D Staff, PDI/DreamWorks

Nadia Magnenat-Thalmann†
MIRALab, University of Geneva

In computer graphics, there are numerous novel models developed for animation of synthetic and natural objects, animals, and virtual humans. Many of the models do not reflect the physical reality, but the mere visual resemblance. For example, a digital actor may not walk in accordance with the accurate dynamics of the body, rather she will follow the footsteps "key-framed" by the animator. In this particular case, the underlying animation model may very well be complex. However, it does not reflect the reality. One of the highlights of such an approach is – it leaves the animator with a complete control of the result. On the other hand, when it comes to animation of fluids, explosions, solid fracture and hair, computer graphics has opted for "direct numerical simulation". These models tend to be more and more physically based. Here the animators are not given the explicit control over the result. The control is only cursory and by means of setting up boundary conditions and defining external force fields. However, the models being accurate, they produce very convincing results. Our approach to hair animation is in the same spirit. For us, the cloth simulation system at MIRALab – University of Geneva and its approach is very inspirational to this regard.

We have discussed the state-of-the-art in hair simulation in [Hadap 2003; Magnenat-Thalmann et al. 2000]. We have identified the difficulties in hair dynamics along with previous attempts, their advantages and limitations. To start with, we take only a quick recap. In this chapter, our focus is mainly the dynamics of long styled hair, as against fur. In this regard, the explicit hair models (or the wisp models) are most effective. In these models, each and every hair (or wisp) is explicitly considered for the dynamics. This makes these models intuitive and close to reality. However, the shape intricacies, the thin geometry and the relatively high stiffness along with the high degree of damping make dynamics of individual hair strand difficult. Further more, the shear number of hair strands demands very careful balance between adequate details in the elastic models and the numerical complexity. Anjyo *et al* [Anjyo et al. 1992] pioneered and Lee *et al* [Lee and Ko 2001] developed on their work in which they diligently model hair inertial and stiffness dynamics as projective two dimensional cantilever dynamics. In our opinion there is plenty of scope for more complex models considering the current computing power. On the other hand, Rosenblum *et al* [Rosenblum et al. 1991] used a mass-spring-hinge model to control the position and the orientation of the hair strand. Unlike the cantilever models, the spring-mass-hinge models are truly three dimensional. However, they give rise to stiff differential equations of motion. They are also inappropriate to model the dynamics of non-straight neutral shape of hair along with the twisting motion. Many of these attempts used approximate collision detection. They replace the head and body geometry by simple analytic shapes such as ellipsoids. We strongly feel that recent developments in real-time computer graphics with regard to collision detection and response certainly facilitate accurate hair-body collision handling. Appreciably, none of the previous attempts considered hair-hair and hair-air interactions - until very recently. Work by Plante *et al* [Plante et al. 2001] and Chang *et al* [Chang et al. 2002] addressed the problem by considering only interaction between wisps. In these novel propositions, the configuration of the wisps remains rather constant and hair does not break away from one wisp and join another.

In recent years the computing power has grown many times. Supercomputing power of the past is becoming increasingly available to the animator's workstations. There is a need to develop new hair dynamics models in light of current and future computing advances. In this chapter, we hope to have developed enough "food for computing" by attempting hair-hair and hair-air interaction with elaborate elastic dynamics of individual hair stand. While making a paradigm shift, to model hair-hair and hair-air interactions, we propose to consider hair as continuum. Subsequently, we treat the hair-hair interaction dynamics and hair-air interaction dynamics to be fluid dynamics. This proves to be a strong as well as viable approach for an otherwise very complex phenomenon. We use smoothed particle hydrodynamics (SPH) as the numerical model. However, for the realization of the shape memory and the rendering, we still need to retain the notion of individual hair strand. In that regard, we develop an elaborate stiffness and inertial dynamics of the individual hair strand. We treat it as a serial rigid multi-body chain. This being a reduced coordinate formulation, the stiffness dynamics is numerically stable and fast. Finally, we unify the continuum interaction dynamics and the stiffness dynamics to realize a strong hair animation framework.

The outline of the chapter is as follows. In the next section, we develop the basic continuum hair model. Realizing the need to retain the individual character of hair, Section 2 gives a detailed model of stiffness dynamics for single

*hadap@acm.org, This work is done at MIRALab, University of Geneva towards completion of PhD
†thalmann@miralab.unige.ch

hair. Section 3 explains the integration of two seemingly disparate approaches, hair volume as a continuum and dynamics of an individual hair. Section 4 demonstrates how hair-body interactions can be modeled in an unified way as the fluid boundary condition. We also discuss various details about collision detection and response in the section. To add hair-air interaction to the model, Section 5 extends the idea of hair as a continuum to a mixture of hair and air. We outline our scheme for the numerical integration in Section 6. Section 7 address some of the implementation issues that make proposed hair dynamics viable. Finally we show the results that demonstrate the effectiveness of the developed hair dynamics model in animating long hair.

1 Hair as a Continuum

Hair has many properties similar to fluid flow. These similarities were identified and exploited in the chapter – "Modeling Hair Shape as Streamlines of Fluid Flow" for the effective static hair shape modeling. The hair is modeled as streamlines of well setup ideal flow. Unfortunately, in this novel approach, no analogy could be developed between fluid flow and the dynamics of hair. We take inspiration from this approach and in this section explore the possibility of modeling complex hair dynamics as fluid dynamics. We consider if and how we can extend the idea of hair being streamlines of fluid flow by associating shape memory to streamlines.

Figure 1: Hair as a Continuum

Hair-hair interaction is very important and the most difficult problem in achieving visually pleasing hair animation. Only recently, Chang *et al* [Chang et al. 2002] and Plante *et al* [Plante et al. 2001] developed hair-hair interaction models based on wisps. They carried out explicit collision detection and response between the wisps of hair. Although these methods are clever and effective, they have the limitation that hair cannot break away from one wisp and join another. We would like to model interactions on hair-hair basis. There are many advances in collision detection and response as compiled by Lin *et al* [Lin and Gottschalk 1998]. However, they are simply unsuitable for the problem at hand because of shear number complexity of hair. This problem warrants to take a radical approach – consider hair as a continuum, see Figure 1. Let us start the discussion by defining the continuum. In the continuum the physical properties of the medium such as pressure, density and temperature are defined at each and every point in the specified region. Fluid dynamics regards liquids and gases as a continuum and even elastic theory regards solids as such, ignoring the fact that they are still composed of individual molecules. Indeed, the assumption is quite realistic at a certain length scale of the observation but at smaller length scales the assumption may not be reasonable. While considering hair as a continuum, it can be argued that hair-hair spacing is not at all comparable to inter molecular distances. However, individual hair-hair interaction is of no interest to us apart from its end effect. Hence, we treat the size of individual hair and hair-hair distance much smaller than the overall volume of hair, justifying the continuum assumption. Panton [Panton 1995] gives an interesting discussion on the continuum assumption. As we develop the model further, it will be apparent that the above assumption is not just about approximating the complex hair-hair interaction. An individual hair is surrounded by air. As it moves, it generates a boundary layer of the air. The boundary layer influences many other hair strands in motion. This aerodynamic form of friction is comparable to mere hair-hair contact friction. In addition, there are electrostatic forces to take part in the dynamics. It is not feasible to model these complex multiple forms of interactions accurately. This inspires us to consider interaction of individual hair strand with the other surrounding strands, in a macroscopic manner, through the continuum assumption. That way, we hope to have a sound model for an otherwise very complex phenomenon.

As we start considering hair as a continuum, we define the properties of such a medium, namely the hair medium. There are two possibilities – hair medium could be considered as a solid or a liquid. This depends on how it behaves under shearing forces. Under shearing stresses, the solids deform till they generate counter stresses. If the shearing

stresses are removed, the solids exhibit ability of retaining their original shape. The liquids are not able to withstand any shearing stresses. Under the influence of the shearing stresses they continue to deform indefinitely and they don't have any shape memory. In case of hair, if we apply a lateral shearing motion it acts like a liquid. At the same time, length wise, it acts as a solid. They even have bending rigidity. Thus, there is a duality in the behavior of hair as a continuum.

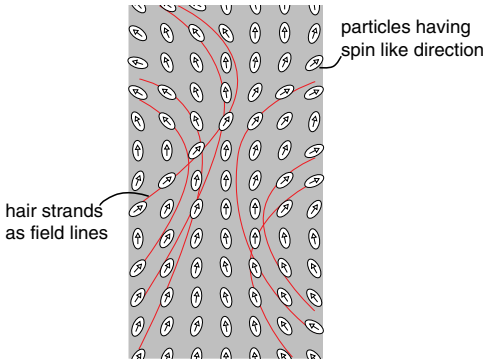


Figure 2: Hair as Field Lines of Oriented Molecules

Interestingly, there are certain liquids that too exhibit this duality – liquid crystals. Liquid crystal molecules have certain preferred orientations as shown in Figure 2, which form a field. The molecules can no more freely demonstrate the fluid like motion because of their preferred directions. Inspired from liquid crystals, we tried to develop a hair dynamics model based on the assumption that each particle of the hair medium will have a spin like pseudo direction, which defines a field. Then the field lines would be synonymous to individual hair strands. The hair dynamics can be formulated based on the anisotropy due to particle orientations. Further, we would have associated some deformation energy to the field lines to simulate the shape memory associated with the hair strand, discussed earlier in this section. However, we realized that this kind of dynamics mimics hair dynamics only instantaneously. Although hair can get sheared laterally, this cannot happen indefinitely. Soon the global, lengthwise effects would come into effect to restrict the lateral motion. Secondly, as the particles of the hair medium move, they form new field lines hence new hairs. This leads to the problem of frame coherency in the model. This would be visually quite disturbing in successive frames of animation. Even from mere rendering point of view, we cannot treat hair solely as a continuum, unless the viewpoint is far enough and individual hair movement is not perceived. Thus, we have to retain the individual character of hair as well, while considering hair as a continuum. Finally, we realize the model by splitting hair dynamics into two parts:

- Hair-hair, hair-body and hair-air interactions, which are modeled using continuum dynamics, and more precisely fluid dynamics
- Individual hair geometry and stiffness, which is modeled using the dynamics of an elastic fiber

Interestingly, this approach even addresses the solid-liquid duality effectively. The model can be visualized as a bunch of hair strands immersed in a fluid. The hair strands are kinematically linked to fluid particles in their vicinity. The individual hair has its own stiffness dynamics and it interacts with the environment through the kinematical link with the fluid. The stiffness dynamics of an individual hair is quite straight forward, which is developed in the next section.

In order to develop the continuum model further, let us start identifying various physical quantities involved in fluid dynamics. Density, pressure and temperature are the basic constituents of fluid dynamics. The density of the hair medium is not precisely the density of individual hair. It is rather associated with the number density of hair in an elemental volume. In Figure 1, observe that the density of the hair medium is less when the number density of the hair is less. The density of the hair medium is thus defined as the mass of the hair per unit occupied volume and is denoted as ρ. The notion of density of the hair medium enables us to express the conservation of mass (it is rather conservation of the number of hair strands) in terms of the continuity equation [Panton 1995]

$$\frac{1}{\rho}\frac{d\rho}{dt} = -\nabla \cdot \vec{v} \tag{1}$$

where, $\vec{v}$ is the local velocity of the medium. Note that the fluid dynamics equations are in the Langrangian form, unlike more popular Eulerain form. This is more explained in section 3. The continuity equation states that the relative rate of change of density ($\frac{1}{\rho}\frac{d\rho}{dt}$), at any point in the medium, is equal to the negative gradient of the velocity field at that point ($-\nabla \cdot \vec{v}$). This is the total outflux of the medium at that point. The physical interpretation of the continuity equation in our case is that, as the hair strands start moving apart, their number density, and hence the density of the hair medium drops and vice a versa.

The pressure and the viscosity in the hair medium represent all the forces due to various forms of interactions of hair strands as described previously. If we try to compress a bunch of hair, it develops a pressure such that the hair strands will tend to move apart. The viscosity would account for various forms of interactions such as hair-hair, hair-body and hair-air. These are captured in the form of the momentum equation [Panton 1995] of fluid.

$$\rho\frac{d\vec{v}}{dt} = \nu\nabla \cdot (\nabla\vec{v}) - \nabla p + F_{bd} \tag{2}$$

The acceleration of the fluid particles $\frac{d\vec{v}}{dt}$ with the spatial pressure variation $-\nabla p$ would be such that it will tend to even out the pressure differences and as the fluid particles move, there will be always resistance $\nu\nabla \cdot (\nabla\vec{v})$ in the form of the friction. The body forces F_{bd}, *i.e.* the inertial forces and gravitational influence are also accounted for in the equation.

Temperature considerably affects the properties of hair. However, we do not have to consider it in the dynamics. We treat the hair dynamics as an isothermal process unless we are trying to simulate a scenario of hair being dried with a hair dryer. Secondly, the temperature is associated with the internal energy of the fluid, which is due to the continuous random motion of the fluid molecules. At the length scale of our model *i.e.* treating hair as a continuum, there is no such internal energy associated with the hair medium. Subsequently, we drop the energy equation of fluid, which is associated with the temperature and the internal energy.

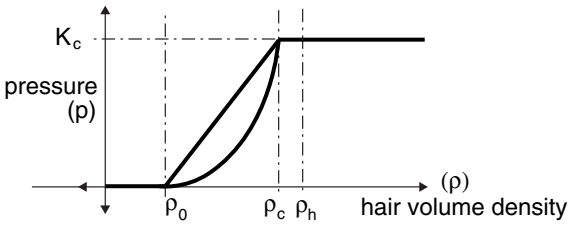


Figure 3: Equation of State

The equation of state (EOS) [Panton 1995] binds together all the fluid equations. It gives a relation between density, pressure and temperature. In our case of hair-hair interaction, the EOS plays a central role along with the viscous fluid forces. The medium we are modeling is not a real medium such as gas or liquid. Hence, we are free to "design" EOS to suit our needs. The following equation is our proposition:

$$p = \begin{cases} 0 & \text{if } \rho < \rho_0, \\ K_c(\frac{\rho-\rho_0}{\rho_c-\rho_0})^n & \text{if } \rho_0 \le \rho < \rho_c, \\ K_c & \text{if } \rho_c < \rho \end{cases} \tag{3}$$

We define the hair rest density ρ_0 as a density below which statistically there is no hair-hair collisions. In addition, we define hair close packing density as ρ_c that represents the state of the hair medium in which hair strands are packed to the maximum extent. This density is slightly lower than the physical density of hair, ρ_h. Figure 3 illustrates the relation between the density and the pressure of the hair medium. In the proposed pressure/density relationship, notice that there is no pressure built up below the hair rest density ρ_0. As one starts squeezing the hair volume, pressure starts building up. As a consequence, the hair strands are forced apart. At the hair compaction density ρ_c, the pressure is maximum. K_c is the interaction constant of the hair volume. The power n refers to the ability of

hair volume to get compressed. If the hair is well aligned, the power will be high. In this case, as we compress the hair volume, suddenly the hair strands start to form close packing and build the pressure quickly. On the contrary, if hair is wavy and not very well aligned, the pressure build up is not abrupt. This will lead to power n towards one.

Instead of modeling the collisions of individual hair strand with the body, we model them, in a unified way, as a boundary condition of the fluid flow. There are two forms of the fluid boundary conditions a) flow tangency condition - the fluid flow normal to the obstacle boundary is zero. b) flow slip condition - the boundary exerts a viscous pressure proportional to the tangential flow velocity. The formulation of the flow boundary condition is deferred to Section 3, where we will introduce the numerical fluid model. It will be apparent that although we model the hair-body interactions as fluid boundary condition, after discretization, the model directly falls under traditional collision detection and response techniques.

Having developed the groundwork for hair-hair and hair-body interactions, in the next section we develop an elaborate stiffness dynamics of the individual hair strand.

2 Single Hair Dynamics

In the previous section we discussed how we could think of hair-hair interaction as fluid forces by considering hair volume as a continuum. However, for the reasons explained there, we still need to retain the individual character of a hair strand. The stiffness dynamics of an individual hair is discussed in this section.

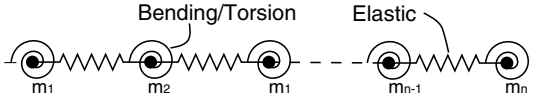


Figure 4: Hair Strand as an Oriented Particle System

In the case of single hair dynamics, as discussed in [Hadap 2003; Magnenat-Thalmann et al. 2000], there are two approaches so far – cantilever dynamics and mass-spring-hinge dynamics. We have seen that the cantilever dynamics is not truly three dimensional. Thus, we sincerely feel that it has limited potential in the light of current computational power. We discuss the spring-mass-hinge model and highlight its limitations, which leads to the development of our model. In a very straightforward manner, one models hair as a set of particles connected by tensile, bending and torsional springs [Daldegan et al. 1993; Rosenblum et al. 1991], as shown in Figure 4. If the hair strand is approximated by a set of n particles, then the system has $6n$ degrees of freedoms (DOFs) attributed to three translations, two bendings and one twist per particle. Treating each particle as a point mass, we can setup a set of governing differential equations of motion and try integrating them. Unfortunately this is not a viable solution. Hair is one of the many interesting materials in nature. It has remarkably high Elastic Modulus of 2-6GPa. Moreover, being very small in diameter, it has very large tensile strength as compared to its bending and torsional rigidity. This proves to be more problematic in terms of the numerics. We are forced to choose very small time steps due to the stiff equations corresponding to the tensile mode of motion, in which we are hardly interested. In fact, the hair fiber hardly stretches by its own weight and body forces. It just bends and twists.

Hence, it is better to choose one of the following two possibilities. Constrain the differential motion of the particles that amounts to the stretching using *constrained dynamics* [Baraff 1996]. Alternatively, reformulate the problem altogether to remove the DOFs associated with the stretching, namely a *reduced coordinate formulation* [Featherstone 1987]. Both methods are equally efficient, being linear time. Parameterizing the system DOFs by an exact number of generalized coordinates may be extremely hard or even impossible for the systems having complex topology. In this case, a constrained method is preferred for its generality and modularity in modeling complex dynamical systems. However, for the problem at hand, the reduced coordinate formulation is a better method for the following reasons:

- Reduced coordinates are preferred when in our case the $3n$ DOFs remaining in the system are comparable to the $3n$ DOFs removed by the elastic constraints.

- The system has fixed and simple topology where each object is connected to maximum of two neighbors. We can take advantage of the simplicity and symbolically reduce the most of the computations.

- Reduced coordinate formulation directly and accurately facilitates the parametric definition of bending and torsional stiffness dynamics. The geometry of spring-mass-hinge system is one dimensional and the masses are point masses. Thus it is difficult to accurately formulate the bending and torsional dynamics as one can not effectively resolve the orientations that facilitate definition of bending and torsion in three dimensions.

Subsequently we model an individual hair strand as a *serial rigid multi-body chain*.

2.1 Hair as Serial Rigid Multi-body Chain

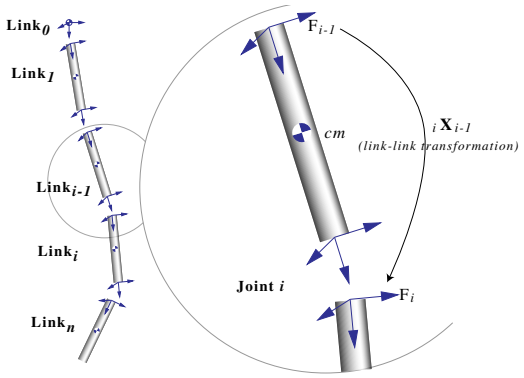


Figure 5: Hair Strand as Rigid multi-body Serial Chain

The first step is to clearly define the serial rigid multi-body system that approximates the motion of individual hair strand. We divide the strand into n segments of equal length as shown in Figure 5. The advantages of defining segments of equal length will be made clear, subsequently. The n segments are labeled as $link_1$ to $link_n$. Each link is connected to two adjacent links by a three DOF spherical joint forming a single un-branched open-loop kinematic chain. The joint between $link_{i-1}$ and $link_i$ is labeled $joint_i$. The position where the hair is rooted to scalp is synonymous to $link_0$ and the joint between head and hair strand is $joint_1$.

Further, we introduce n coordinate frames $\mathfrak{F}_i$, each attached to the corresponding $link_i$. The coordinate frame $\mathfrak{F}_i$ moves with the $link_i$. The placement of coordinate system is largely irrelevant to the mathematical formulations, but they do have an important bearing on efficiency of computations, which is discussed subsequently. Having introduced the link coordinates, we introduce the *spatial transformations* that enable us to transform *spatial entities* defined in the coordinate frame of one link, in terms of the coordinate frame of the adjacent link. ${}_i\hat{\mathbf{X}}_{i-1}$ is an adjacent-link coordinate *spatial transformation* which operates on a *spatial vector* represented in coordinate frame $\mathfrak{F}_{i-1}$ and produces a representation of the same spatial vector in coordinate frame $\mathfrak{F}_i$. For comprehensive discussion on spatial vector algebra and it's application to rigid body dynamics along with the peculiar notations, we refer to pioneering work by Featherstone [Featherstone 1987].

We use the notations introduced by Featherstone. Small case letters such as l denote scalars and bold face letters such as $\mathbf{v}$ denote cartesian vectors. Spatial 6×1 vectors and spatial 6×6 matrices are denoted by bold face small and capital letters, respectively, having a hat on top, e.g. $\hat{\mathbf{v}}$. Subscript on an entity denotes the associated link, e.g. $\hat{\mathbf{v}}_i$ denote the spatial velocity of the i^{th} link. Entities with dash, e.g. $\hat{\mathbf{I}}'$, denote that they are defined in the local coordinate frame. The spatial transpose operator is denoted by a superscript S, e.g. $\hat{\mathbf{X}}^S$.

Figure 5 illustrates the definition of a hair strand as a serial multi-body rigid chain. The spatial transformation ${}_i\hat{\mathbf{X}}_{i-1}$ is composed of a pure translation, which is constant as the length of the segment is constant, and a pure orientation which is variable. We use a unit quaternion $\mathbf{q}_i$ to describe the orientation of each link with respect to the previous link. Then, we augment the components of n quaternions, one per joint, to form $\mathbf{q} \in \Re^{4n}$, the system state vector. Note that, additional n unit quaternion constraints, *i.e.* $|\mathbf{q}_i| = 1$, make system have $3n$ coordinates. Thus system is optimally represented to have $3n$ DOFs. Moreover, the angular velocity across the spherical joint is described by conventional 3×1 angular velocity vector $\mathbf{w}_i$. These form $\mathbf{w} \in \Re^{3n}$, the derivative state vector of the system. The spatial motion of the rigid body, $link_i$ in our case, is fully characterized by its 6×1 *spatial velocity* $\hat{\mathbf{v}}_i$, 6×1 *spatial acceleration* $\hat{\mathbf{a}}_i$ and 6×6 *spatial inertia tensor* $\hat{\mathbf{I}}_i$. In the next subsections, we will formulate the spatial dynamics of serial rigid multi-body chain in terms of the system state variables $\mathbf{q}$ and $\mathbf{w}$ and their respective derivatives $\dot{\mathbf{q}}$ and $\dot{\mathbf{w}}$, using the physical quantities $\hat{\mathbf{v}}_i$, $\hat{\mathbf{a}}_i$ and $\hat{\mathbf{I}}_i$ for dynamics.

2.2 Kinematics of Hair Strand

A 6×3 motion sub-space $\hat{\mathbf{S}}$ relates the angular velocity $\mathbf{w}_i$ to spatial velocity across the joint, which is the only allowed motion by the spherical joint. Since the position of the link in its own coordinate frame remains fixed, we can express the motion sub-space $\hat{\mathbf{S}}$ as a constant matrix.

$$\hat{\mathbf{S}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4}$$

Subsequently, the velocity and acceleration across the spherical joint are given by the following equations:

$$\begin{aligned} \hat{\mathbf{v}}_i &= {}_i\hat{\mathbf{X}}_{i-1}\hat{\mathbf{v}}_{i-1} + \hat{\mathbf{S}}\mathbf{w}_i \\ \hat{\mathbf{a}}_i &= {}_i\hat{\mathbf{X}}_{i-1}\hat{\mathbf{a}}_{i-1} + \hat{\mathbf{v}}_i \hat{\times}\ \hat{\mathbf{S}}\mathbf{w}_i + \hat{\mathbf{S}}\dot{\mathbf{w}}_i \end{aligned} \tag{5}$$

Given joint angular velocities $\mathbf{w}_i$ and joint angular accelerations $\dot{\mathbf{w}}_i$, Equations 4 and 5 enable us to recursively compute the link velocities $\hat{\mathbf{v}}_i$ and the link accelerations $\hat{\mathbf{a}}_i$, with $\hat{\mathbf{v}}_0$ and $\hat{\mathbf{a}}_0$ as a starting point. In our case $\hat{\mathbf{v}}_0$ and $\hat{\mathbf{a}}_0$ are the spatial velocity and the spatial acceleration of hair root, *i.e.* the scalp. We need to successively integrate the derivative vectors of the system *i.e.* integrating $\dot{\mathbf{w}}_i$ into $\mathbf{w}_i$ and $\mathbf{w}_i$ into $\mathbf{q}_i$. One can notice that the angular velocity $\mathbf{w}_i$ can not be integrated directly into joint variables $\mathbf{q}_i$. However, the following equation relates the joint variable rates $\dot{\mathbf{q}}_i$ expressed as quaternions to the angular velocities $\mathbf{w}_i$

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = 1/2 \begin{bmatrix} -q_1 & -q_2 & -q_3 \\ q_0 & -q_3 & q_2 \\ q_3 & q_0 & -q_1 \\ -q_2 & q_1 & q_0 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} \tag{6}$$

$$q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1 \tag{7}$$

Next step is to identify various external forces acting on the links, which induce the joint angular accelerations and make hair strand bend and move.

2.3 Forward Dynamics of Hair Strand

Before we discuss the dynamics of single hair strand, we tabulate the physical properties of a typical human hair strand. Especially note the formulas for the moment of area, the polar moment of area, bending spring constant and torsional spring constant. The detailed discussion on the hair properties is covered in [Hadap 2003].

Number of hair strands on human scalp	90-120 thousand, we use 20-40 thousand for animation purpose which are "data-amplified" to around 50-60 thousand for rendering purpose
Typical diameter ($D = 2R$)	60-100 μm
Cross section	Circular to elliptical, we assume circular
Typical distance between hair on scalp	1mm
Linear density of hair strand	30-100 μgm/cm
Density of hair material	1.2 gm/cm^3
Elastic modulus (E)	2-6 GPs
Moment of area (I)	$\pi R^4/4$
Polar moment of area (I_p)	$\pi R^4/2$
Equivalent bending spring constant (K_b)	EI/l, where l is length of the hair segment
Equivalent torsional spring constant (K_t)	$\frac{1}{2(1+\nu)}\frac{EI_p}{l}$, where ν is Poisson ratio

Table 1: Typical Physical Properties of Human Hair

A number of forces act on each link apart from the gravitational influence $\hat{\mathbf{g}}$. The explicit definition of the point of action of the spatial force on link is irrelevant as it is embedded in the definition of the spatial force, thus resulting in a very compact representation.

- The gravitational influence is accommodated by giving the base of the zeroth link representing the root a fictitious additional negative gravitational acceleration, *i.e.* by subtracting $\hat{\mathbf{g}}$ from $\hat{\mathbf{a}}_0$.
- The inertial dynamics plays a pivotal role in the case of dynamics of hair strand, even though the hair strand is thin in geometry. Inertia of the individual segment is indeed small as compared to its stiffness. However, it is the first and second moments of inertia that govern the dynamics. The serial rigid-multibody chain formulation facilitates us to accurately account for the inertial dynamics of the hair strand. The *spatial momentum* of each link is composed of the spatial velocity $\hat{\mathbf{v}}_i$ and the *spatial inertia* $\hat{\mathbf{I}}_i$, a 6x6 matrix. Since the position of the link in its own coordinate frame remains fixed, we can express the spatial inertia $\hat{\mathbf{I}}_i$ as a constant. Further, by proper choice of coordinate system, $\hat{\mathbf{I}}_i$ assumes a rather simple form.

$$\hat{\mathbf{I}} = \begin{bmatrix} \mathbf{H}^T & \mathbf{M} \\ \mathbf{I} & \mathbf{H} \end{bmatrix} \tag{8}$$

where $\mathbf{M}$, $\mathbf{H}$ and $\mathbf{I}$ are the 3x3 matrix representations of zeroth, first and second moments of mass of the link around the origin of its own frame as described above. Notice that although the mass of the individual link is small and subsequently the mass matrix $\mathbf{M}$ tend to be singular, due to well conditioned $\mathbf{H}$ and $\mathbf{I}$, $\hat{\mathbf{I}}$ is not singular. Table 1 gives the expressions for moment of area and polar moment of area of the cylindrical hair segment, which enables us to formulate the second moment of inertia $\mathbf{I}$.

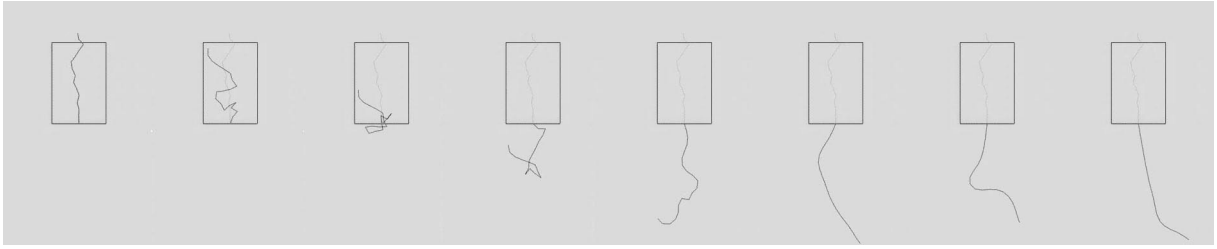


Figure 6: Free-fall of Hair Strand – No Stiffness

Figure 6 illustrates the motion of free falling hair strand without stiffness. Thus the motion is solely governed by the gravity and inertial dynamics of links. This motion is similar to that of a chain. Needless to state that the elongation constraint is always maintained as the system does not have any corresponding DOF in the definition. We defer the details of the algorithm that computes the motion till the next section, where we will have defined all the forms of forces acting on the hair strand.

- In order to account for the bending and torsional rigidity of the hair strand, the $joint_i$ exerts an actuator force $\mathbf{Q}_i^a \in \Re^3$ on both $link_{i-1}$ and $link_i$ in opposite directions. The actuator force is not a spatial force but rather a force expressed in joint motion space. The joint actuator force is a function of joint variables $\mathbf{q}_i$ incorporating the bending and torsional stiffness constants. To realize the joint actuator force, we uniquely decompose the joint variable $\mathbf{q}_i$ into a pure bending component θ_i^b around the axis $\mathbf{b}_i$ followed by a pure twist component θ_i^t around link axis. We would like to highlight that this unique decomposition is only possible due to the accurate representation of the orientation of the adjacent links via joint variable $\mathbf{q}_i$. Similarly, the neutral hair strand shape defines a set of neutral orientations $\mathbf{q}_i^0$ which are decomposed into the neutral bending component θ_i^{b0} around axis $\mathbf{b}_i^0$ and the pure twist component θ_i^{t0}. From θ_i^b, $\mathbf{b}_i$, θ_i^{b0}, $\mathbf{b}_i^0$, θ_i^t and θ_i^{t0}, given equivalent bending spring constant and equivalent twist spring constant listed in Table 1 along with respective damping constants, one can formulate the actuator force $\mathbf{Q}_i^a$. The details of the formulation needs the discussion on how to represent quantities in the joint space, instead we refer to discussion in [Featherstone 1987].

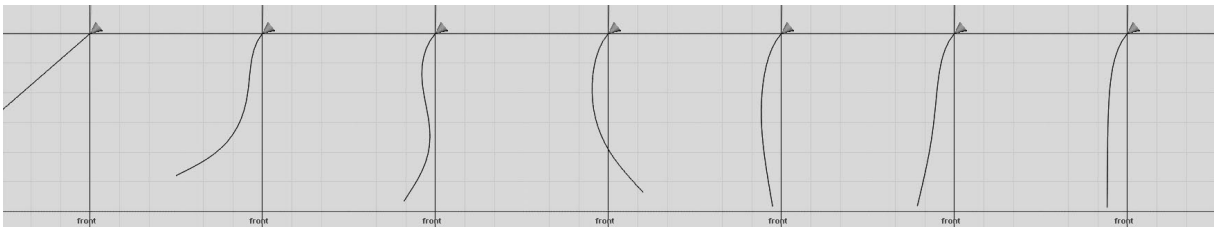


Figure 7: Shape-memory of Hair Strand

Figure 7 illustrates the stiffness dynamics of a hair strand. Under cantilever action, the hair strand bends in the direction of the gravity. Due to (primarily) bending stiffness, it tries to retain its neutral shape – straight line in this case.

- Force $\hat{\mathbf{f}}_{ci}$ is the interaction spatial force (aggregate of line force and torque) on $link_i$ coming from the kinematic link with the hair medium as discussed in Section 1. The actual form of $\hat{\mathbf{f}}_{ci}$ is given in Sections 3, 4 and 5. This force accounts for all the interaction effects such as hair-hair collision, hair-body collision and hair-air drag.

Given the set of forces acting on the system, we now need to calculate the motion of the hair strand. This evolves calculation of the induced joint angular accelerations $\dot{\mathbf{w}}_i$ followed by the integration. This is a forward dynamics problem involving a rigid multi-body system. We use Articulated-Body Method to solve the hair strand forward dynamics. This method has a computational complexity of $O(n)$. The detailed discussion of this algorithm is beyond the scope of this chapter. It is comprehensively covered in [Featherstone 1987; Mirtich 1996]. In the next section we give a brief outline of the method.

2.4 Articulated-Body Forward Dynamics Algorithm

We use Articulated-Body method to solve the hair strand dynamics stated in the previous section. This method has a computational complexity if $O(n)$ as compared with $O(n^3)$ methods such as Composite-Rigid-Body method. For the detailed discussion of these algorithm refer to [Featherstone 1987].

A collection of rigid bodies connected by joints is called an articulated body. To define an articulated body inertia, we single out a particular member of the articulated body, called the handle, and define the articulated inertia as a relationship between a test force $\hat{\mathbf{f}}$ applied to the handle and the acceleration $\hat{\mathbf{a}}$ of the handle according to

$$\hat{\mathbf{f}} = \hat{\mathbf{I}}^A \hat{\mathbf{a}} + \hat{\mathbf{p}} \tag{9}$$

$\hat{\mathbf{I}}^A$ is the articulated-body inertia and $\hat{\mathbf{p}}$ is the bias force, which is the value of the test force that must be applied to the handle in order to give it zero acceleration. Then, the basic idea of the articulated-body algorithm is to treat

n-joint multi-body system as a one-joint system whose only moving link is in fact the handle of an articulated body comprising all the remaining links. Then we find the acceleration of the first joint solving the forward dynamics of a single joint robot, which is relatively simple. Having solved for acceleration of joint 1, we can treat link 1 as the (moving) base of an $n-1$ joint robot and repeat the process for joint 2, and so on.

The algorithm is a 3 step process.

Common sub-expressions

$$\hat{\mathbf{c}}_i = \hat{\mathbf{v}}_i \hat{\times} \hat{\mathbf{S}}\mathbf{w}_i \tag{10}$$

$$\hat{\mathbf{h}}_i = \hat{\mathbf{I}}_i^A \hat{\mathbf{S}} \tag{11}$$

$$d_i = \hat{\mathbf{S}}^S \hat{\mathbf{h}}_i \tag{12}$$

$$u_i = \hat{\mathbf{S}} Q_i^a - \hat{\mathbf{h}}_i^S \hat{\mathbf{c}}_i - \hat{\mathbf{S}}^S \hat{\mathbf{p}}_i \tag{13}$$

Step 1

$$\hat{\mathbf{v}}_i = {}_i\hat{\mathbf{X}}_{i-1} \hat{\mathbf{v}}_{i-1} + \hat{\mathbf{S}}\mathbf{w}_i \tag{14}$$

Step 2

$$\hat{\mathbf{p}}_i^v = \hat{\mathbf{v}}_i \hat{\times} \hat{\mathbf{I}}_i \hat{\mathbf{v}}_i \tag{15}$$

$$\hat{\mathbf{I}}_i^A = \hat{\mathbf{I}}_i +_i \hat{\mathbf{X}}_{i+1} (\hat{\mathbf{I}}_{i+1}^A - \frac{\hat{\mathbf{h}}_{i+1} \hat{\mathbf{h}}_{i+1}^S}{d_{i+1}})_{i+1} \hat{\mathbf{X}}_i, \quad (\hat{\mathbf{I}}_n^A = \hat{\mathbf{I}}_n) \tag{16}$$

$$\hat{\mathbf{p}}_i = \hat{\mathbf{p}}_i^v +_i \hat{\mathbf{X}}_{i+1} (\hat{\mathbf{p}}_{i+1} + \hat{\mathbf{I}}_{i+1}^A \hat{\mathbf{c}}_{i+1} + \frac{u_{i+1}}{d_{i+1}} \hat{\mathbf{h}}_{i+1}), \quad (\hat{\mathbf{p}}_n = \hat{\mathbf{p}}_n^v) \tag{17}$$

Step 3

$$\dot{\mathbf{w}}_i = \frac{u_i - \hat{\mathbf{h}}_i^S \, {}_i\hat{\mathbf{X}}_{i-1} \hat{\mathbf{a}}_{i-1}}{d_i} \tag{18}$$

$$\hat{\mathbf{a}}_i = {}_i\hat{\mathbf{X}}_{i-1} \hat{\mathbf{a}}_{i-1} + \hat{\mathbf{c}}_i + \hat{\mathbf{S}} \dot{\mathbf{w}}_i \tag{19}$$

§1 Start from $\hat{\mathbf{v}}_0$, the velocity of the base *i.e.* that of the hair strand root. Using current value of joint angular velocities $\mathbf{w}_i, i = 1 \ldots n$, compute all the link velocities $\hat{\mathbf{v}}_i, i = 1 \ldots n$ from $\hat{\mathbf{v}}_{i-1}$, using equation 14.

§2 Given link spatial inertias $\hat{\mathbf{I}}_i, i = 1 \ldots n$, start from the last link's articulated-body inertia $\hat{\mathbf{I}}_n^A = \hat{\mathbf{I}}_n$ and bias force $\hat{\mathbf{p}}_n = \hat{\mathbf{v}}_n \hat{\times} \hat{\mathbf{I}}_n \hat{\mathbf{v}}_n$. Compute all the articulated-body inertias $\hat{\mathbf{I}}_i^A, i = n-1 \ldots 1$ and the bias forces $\hat{\mathbf{p}}_i, i = n-1 \ldots 1$ from $\hat{\mathbf{I}}_{i+1}^A$ and $\hat{\mathbf{p}}_{i+1}$, using equations in Step 2.

§3 Once we know all the articulated-body inertias and bias forces we start from the link 1. Given all the external forces acting on links (see Section 2.3), we compute the joint angular accelerations $\dot{\mathbf{w}}_i, i = 1 \ldots n$ using equations in Step 3. We update the link acceleration $\hat{\mathbf{a}}_i$ before we move on to the next link.

The time evolution of the hair strand shape is broken into discrete steps in time. At each time step, we evaluate the joint angular accelerations $\dot{\mathbf{w}}_i, i = 1 \ldots n$ followed by the numerical integration. The details of numerical integration are covered in Section 6 after the details of all the forces acting on the hair strand are covered in the subsequent sections.

3 Fluid Hair Model

In the previous section we described the precise dynamics of individual hair strand. We considered bending and torsional stiffness of hair along with body forces *viz.* inertia and gravitational influence. In this section, we develop on the proposed continuum model for hair-hair interactions. As discussed in Section 1, the density and the pressure of the hair medium form the basic constituents of the fluid-hair model. The continuity equation (Eq. 1), the momentum equation (Eq. 2) and the equation of state (Eq. 3) capture the overall dynamics of hair-hair interaction. Establishing

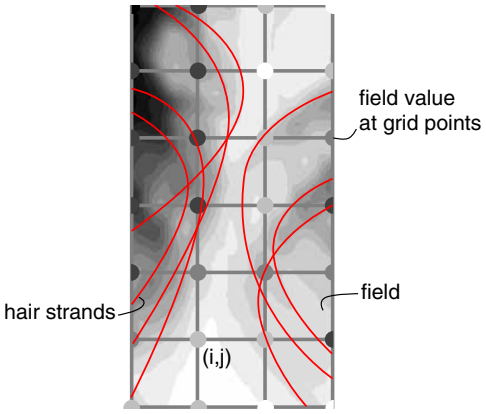


Figure 8: Fluid Dynamics – Eulerian viewpoint

the kinematical link between the dynamics of the individual hair strand and the dynamics of interactions is a crucial part of the algorithm, which is addressed in this section.

The conventional fluid dynamics formulation uses Eulerian viewpoint. One way to think of Eulerian method is to think of an observer watching the fluid properties such as density, temperature and pressure change at a certain fixed point in space, as fluid passes through this point. In the numerical simulations, the space is discretised using a rectangular grid or a triangular mesh to define these few observation points for computations, as shown in Figure 8. Hence using the Eulerian viewpoint, we will ultimately get fluid forces acting at this fixed set of points. We would like to transfer the fluid force at each of these points onto the individual hair, which is in the vicinity of the point. There is no trivial correlation between the grid points and the hair strands, unless they coincide. Also the hair strand will be in the vicinity of new set of grid points every time it moves. This makes it difficult to formulate the kinematical link between the two. There are methods such as the particle-in-cell method introduced by Hockney *et al* [Hockney and Eastwood 1988], which try to do the same. However, we opted for the other, less popular but effective, Langrangian formulation of fluid dynamics. We explain the benefits subsequently.

In Langrangian formulation, the physical properties are expressed as if the observer is moving with the fluid particle. *Smoothed Particle Hydrodynamics* (SPH), invented by Monaghan [Monaghan 1992], is one of the Langrangian numerical methods, that utilizes space discretisation via number of discrete points that move with the fluid flow. One of the first applications of SPH in computer animation was done by Gascuel *et al* [Gascuel et al. 1996]. For a good overview of SPH, we refer to [Morris 1995].

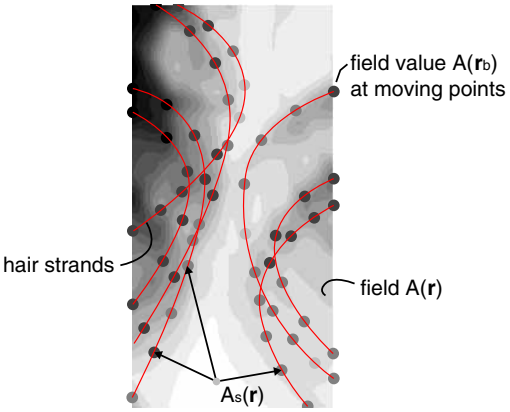


Figure 9: Fluid Dynamics – Langrangian viewpoint

Figure 9 illustrates the concept of smoothed particles. The physical properties are expressed at the center of each of these smoothed particles. Then the physical property at any point in the medium is defined as a weighted sum of the properties of all the particles.

$$A_s(\mathbf{r}) = \sum_b A_b \frac{m_b}{\rho_b} W(\mathbf{r} - \mathbf{r}_b, h) \tag{20}$$

The summation interpolant $A_s(\mathbf{r})$ can be thought of as the smoothed version of the original property function $A(\mathbf{r})$. The field quantities at particle b are denoted by a subscript b. Thus, the mass associated with particle b is m_b and density at the centre of the particle b is ρ_b, and the property itself is A_b. We see that the quantity $\frac{m_b}{\rho_b}$ is the inverse of the number density (*i.e.* the specific volume) and is, in some sense, a volume element. The function W is the weight function referred as interpolating kernel in SPH. Details of the interpolating function are covered subsequently.

To exemplify, the smoothed version of density at any point of medium is

$$\rho(\mathbf{r}) = \sum_b m_b W(\mathbf{r} - \mathbf{r}_b, h) \tag{21}$$

Figure 9 illustrates how density is recorded onto each particle, denoted by varying degree of gray scale values of the dots. The field is defined at each and every point in the region by weighted sum of the field values of the surrounding particles, which is denoted by the continuous gray tones in the region.

Similarly, it is possible to obtain an estimate of the gradient of the field, provided W is differentiable, simply by differentiating the summation interpolant

$$\nabla A_s(\mathbf{r}) = \sum_b A_b \frac{m_b}{\rho_b} \nabla W(\mathbf{r} - \mathbf{r}_b, h) \tag{22}$$

The interpolating kernel $W(\mathbf{r} - \mathbf{r}', h)$ has the following properties

$$\int W(\mathbf{r} - \mathbf{r}', h) d\mathbf{r}' = 1 \tag{23}$$

$$\lim_{h \to 0} W(\mathbf{r} - \mathbf{r}', h) = \delta(\mathbf{r} - \mathbf{r}') \tag{24}$$

The choice of the kernel is not important in theory as long as it satisfies the above kernel properties. However, for practical purposes we need to choose a kernel, which is simple to evaluate and has compact support. The *smoothing length* h defines the extent of the kernel. We use the cubic spline interpolating kernel.

$$W(\mathbf{r}, h) = \frac{\sigma}{h^\nu} \begin{cases} (1 - \frac{3}{2}s^2 + \frac{3}{4}s^3) & \text{if } 0 \le s \le 1, \\ \frac{1}{4}(2 - s)^3 & \text{if } 1 \le s \le 2, \\ 0 & \text{otherwise} \end{cases} \tag{25}$$

Where $s = |\mathbf{r}|/h$, ν is the number of dimensions and σ is the normalization constant with values $\frac{2}{3}$, $\frac{10}{7\pi}$, or $\frac{1}{\pi}$ in one, two, or three dimensions, respectively. We can see that the kernel has a compact support, *i.e.* its interactions are exactly zero at distances $|\mathbf{r}| > 2h$. Evaluating the field at each point involves computation of the contribution due to all the particles in the region. The compact support, i.e. the kernel has zero value outside the smoothing length, drastically reduces the computational overhead as we need to consider only the neighboring particles within the smoothing length in order to evaluate the function at a point. Figure 10 illustrates a typical kernel having a compact support. We keep the smoothing length h constant throughout the simulation to facilitate a speedy search of neighborhood of the particles. The nearest neighbor problem is well known in computer graphics. Section 7 gives the strategy for the linear time neighbor search.

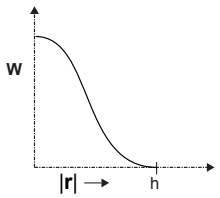


Figure 10: SPH Kernel having Compact Support

There is no underlying grid structure in the SPH method, which makes the scheme suitable for our purpose. We are free to choose the initial positions of the smoothed particles as long as their distribution reflects the local density depicted by Equation 21. Eventually the particles will move with the fluid flow. In order to establish the kinematical link between the individual hair dynamics and the dynamics of interactions, we place the smoothed particles directly onto the hair strands as illustrated in the Figure 9. We keep the number of smoothed particles per hair segment constant, just as we have kept the hair segment length constant, for the reasons of computational simplicity. As the smoothed particles are glued to the hair strand, they can no longer move freely with the fluid flow. They just exert forces arising from the fluid dynamics onto the corresponding hair segment and move with the hair segment (in the figure, the hair strand is not discretised to show the segments). Thus, we have incorporated both, the elastic dynamics of individual hair and the dynamics of interactions into hair dynamics.

Apart from providing the direct kinematical link, the SPH method has other numerical merits when compared to a grid-based scheme:

- As there is no need for a grid structure, we are not defining a region of interest to which the dynamics must confine to. This is very useful considering the fact that, in animation the character will move a lot and the hair should follow it.
- No memory is wasted in defining the field in the region where there is no hair activity, which is not true in the case of grid-based fluid dynamics.
- As the smoothed particles move with the flow carrying the field information, they optimally represent the fluctuations of the field. In the case of grid-based scheme, it is necessary to opt for tedious adaptive grid techniques to achieve similar computational resolution, within given memory footprint.

In the rest of the section, we discuss the SPH versions of the fluid dynamics equations. Each smoothed particle has a constant mass m_b. The mass is equal to the mass of the respective hair segment divided by the number of smoothed particles on that segment. Each particle carries a variable density ρ_b, variable pressure p_b and has velocity $\mathbf{v}_b$. The velocity $\mathbf{v}_b$ is actually the Cartesian velocity of the point on the hair segment where the particle is located, expressed in the global coordinates. $\mathbf{r}_b$ is the global position of the particle, *i.e.* the location of the particle on the hair strand in the global coordinates. Once, initially, we have placed the particles on the hair strands, we compute the particle densities using Equation 21. Indeed, the number density of hair at a location reflects the local density, which is consistent with the definition of the density of the hair medium given in Section 1.

For brevity, we introduce the notation $W_{ab} = W(\mathbf{r}_a - \mathbf{r}_b, h)$. Similarly, let $\nabla_a W_{ab}$ denote the gradient of W_{ab} with respect to $\mathbf{r}_a$ (the coordinates of particle a). The quantities such as $\mathbf{v}_a - \mathbf{v}_b$ shall be written as $\mathbf{v}_{ab}$.

The density of each particle can be always found from Equation 21, but this equation requires an extra loop over all the particles, which means the heavy processing of nearest neighbour finding, before it can be used in the calculations. A better formula is obtained from the smoothed version of the continuity equation, Equation 1.

$$\frac{d\rho_i}{dt} = \rho_i \sum_{j=1}^{N} \frac{m_j}{\rho_j} \mathbf{v}_{ij} \cdot \nabla_i W_{ij} \tag{26}$$

Using this formula, we now can update the particle density without going through the particles just by integrating the above equation. However, we would have to correct the densities from time to time using Equation 21, to avoid the density being drifted due to numerical inaccuracies.

The smoothed version of the momentum equation, Equation 2, without the body forces, is as follows

$$\frac{d\mathbf{v}_i}{dt} = -\sum_{j=1}^{N} m_j \left(\frac{p_j}{\rho_j^2} + \frac{p_i}{\rho_i^2} + \prod_{ij}\right) \nabla_i W_{ij} \tag{27}$$

The reason for dropping the body force F_{bd} is that, the comprehensive inertial and gravitational effects are already incorporated in the stiffness dynamics of the individual strand. Otherwise, we would be duplicating them.

As the particles are glued to the respective hair segment, they cannot freely attain the acceleration $\frac{d\mathbf{v}_i}{dt}$ given by the momentum equation. Instead, we convert the acceleration into a force by multiplying both the sides of Equation 27 with the mass of the particle m_i. Thus, instead of particle accelerating according to the governing equations of motion, they merely apply forces, arising from the fluid dynamics, onto the hair strand. In the previous section, we

referred this total of all the fluid forces due to each particle on the segment as the interaction force $\hat{\mathbf{f}}_{ci}$. Although, we need to convert the Cartesian form of the force into the spatial force in order to incorporate it in the spatial dynamics – this is straightforward.

In Equation 27, $\prod_{ij}$ is the viscous pressure, which accounts for the frictional interaction between the hair strands. We are free to design it to suit our purpose, as it is completely artificial, taking inputs from the artificial viscosity form for SPH proposed by [Morris 1995], we set it to

$$\begin{aligned} \prod_{ij} &= \begin{cases} \frac{-c\mu_{ij}}{\bar{\rho}_{ij}} & \text{if } \mu_{ij} < 0 \\ 0 & \text{if } \mu_{ij} \geq 0 \end{cases} \\ \mu_{ij} &= h\frac{\mathbf{v}_{ij} \cdot \mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^2 + h^2/100} \\ \bar{\rho}_{ij} &= (\rho_i + \rho_j)/2 \end{aligned} \tag{28}$$

Here, the constant c is the speed of sound in the medium. However, in our case, it is just an animation parameter. We are free to set this to an appropriate value that obtains satisfactory visual results. The term incorporates both bulk and shear viscosity, and in totality accounts for all the dissipative interactions amongst the hair strands due to the friction and the boundary layer around the hair strands.

At each step of the integration, first we obtain the density at each particle ρ_i using Equation 26. To correct numerical errors from time to time, we use Equation 21. The only unknown quantity so far is the pressure at each particle p_i. Once we know the particle densities ρ_i, the equation of the state (Equation 3), directly gives the unknown pressure. This is the central theme of the algorithm. Subsequently, we compute the fluid forces acting on each particle using the momentum equation (Equation 27). We know now the interaction forces $\hat{\mathbf{f}}_{ci}$ for each hair segment and we are ready to integrate the equation of the motion for individual hair strand, which is covered in Section 6.

The complete validation of the model remains to be done through systematic virtual experiments, backed by empirical study. We believe that the model has good scientific potential – we leave this aspect as a future work. However, the model in the existing form is adequate to capture the hair-hair interactions for the animation purpose.

4 Hair-body Interactions as Fluid Boundary Condition

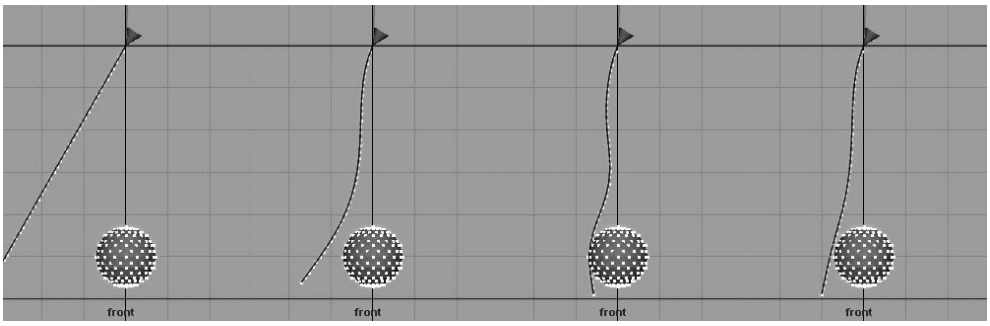


Figure 11: Hair-obstacle Collision – Interaction with Boundary Particles

As discussed in Section 1, we model hair-body interactions as the fluid boundary condition. It is quite straightforward to model solid boundaries, either stationary or in motion, using special boundary particles. We place the boundary particles along the geometry as shown in Figure 11. The boundary particles do not contribute to the density of the fluid and they are inert to the forces coming from the fluid particles. However, they exert a boundary force onto the neighboring fluid particles. A typical form of boundary force is as follows and is given by Morris [Morris 1995]. Each boundary particle has an outward pointing unit normal $\mathbf{n}$ and exerts a force

$$\begin{aligned} \mathbf{f}_n &= K_n \;\; f_1(\triangle\mathbf{r} \cdot \mathbf{n}) \;\; P(\triangle\mathbf{r} \cdot \mathbf{t}_r) \;\; \mathbf{n} \\ \mathbf{f}_t &= -K_f \;\; |\mathbf{f}_n| / K_n \;\; (\triangle\mathbf{v} \cdot \mathbf{t}_v) \;\; \mathbf{t}_v \end{aligned} \tag{29}$$

where, $\triangle\mathbf{r}$ is the position vector from the boundary particle to the colliding fluid particle. The tangent $\mathbf{t}_r$ is the unit projection of the position vector $\triangle\mathbf{r}$ onto the tangent plane at the position of the boundary particle. Similarly, the

tangent $\mathbf{t}_v$ is the unit projection of the relative approach velocity of the fluid particle $\triangle\mathbf{v}$, onto the tangent plane at the position of the boundary particle. Function f_1 is any suitable unit function, which will repel the flow particle away. P is Hamming window, which spreads out the effect of the boundary particle to neighbouring points in the boundary. That way, we have a continuous boundary defined by discrete set of boundary particles. The coefficient of friction K_f determines the extent of the tangential flow slip force $\mathbf{f}_t$, whereas coifficient K_n determines the extent of collision force $\mathbf{f}_n$. Figure 11 demonstrates that the boundary particles are quite effective in achieving collision response. As the boundary particle method is within the framework of SPH, one need not use exclusive collision detection and response for the purpose. However, this method have a significant drawback. The method works effectively only if the boundary particles are placed uniformly on the obstacle geometry. Secondly, if the separation between the boundary particles is large, hair will slip into the boundary though the separation. These drawbacks demand extra work by animators to define a clean uniform geometry as the boundary particles are placed on the vertices of the geometry. We developed a more detailed collision detection and response technique, although the theme remains the same – penalize the smoothed particles approaching the boundary with a collision force.

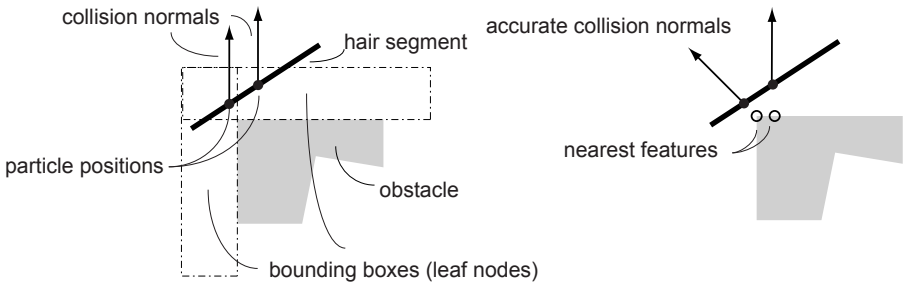


Figure 12: Inaccurate vs Accurate Collision Normal and Nearest Feature

Computer graphics has extensive methods for collision detection. For the state-of-the-art in collision detection, we refer to the nice overview by Lin *et al* [Lin and Gottschalk 1998]. Choosing a right collision detection strategy for hair simulation required a lot of deliberation. The large number of hair strands in the simulation (typically 5,000 to 25,000) pose the challenge. We need to detect the collision of the very large number of smoothed particles with large number of mesh polygons. Many popular collision detection techniques such as AABB Tree and OBB Tree methods are local in nature. They only may detect the proximity of the particle with a mesh polygon and particularly fail to give exact collision normal. One can assign the normal of the colliding mesh polygon as the collision normal, as shown in Figure 12a. However, the collision normal should point away from the nearest feature of the colliding polygon, as shown in Figure 12b. The accurate collision normal is required for effective computation of the collision force and particularly the frictional force. The closes feature tracking method [Cohen et al. 1995] gives the accurate nearest feature of mesh, be it a polygon, an edge or a vertex. However, this method is quite slow for our purpose.

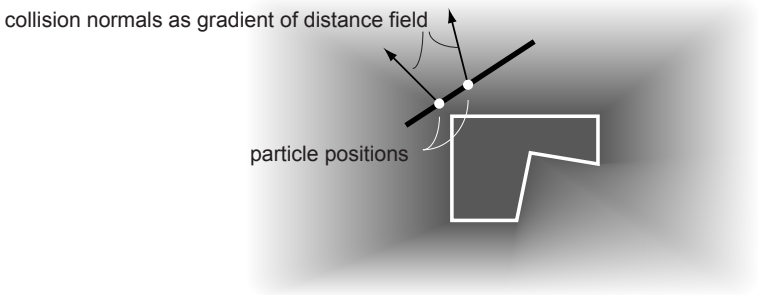


Figure 13: Collision Detection using Distance Field

We use novel *adaptively sampled distance field* (ADF) method by Frisken *et al* [Frisken et al. 2000]. Distance field is the scalar field in a region surrounding the obstacle. Figure 13 illustrates the distance field around the obstacle. The value of the distance field at any point in the region defines the nearest distance from that point to the obstacle geometry. We would like to highlight a property of the distance field – the gradient of the distance field at any point always points away from the obstacle. Whereas, the scalar field value directly determines how close is the point from

the obstacle. As shown in the figure, once we determine that the particles are colliding by examining the distance field at the particle positions, the collision normals can be assigned to the gradients of the distance field at those positions. ADF encodes the distance field in a very memory efficient manner. Further, evaluating ADF at arbitrary point along with the gradient is very fast. Constructing the ADF is computationally intensive, which we do it for each time step. However, the overall method of determining collisions of smoothed particles along with collision normals is very fast and accurate using the ADF. For the numerous details on constructing ADF, we refer to [Frisken et al. 2000].

For the collision response, we use simple *penalty method*. The collision force is similar to Equation 29 and is given by the following equation

$$\begin{aligned} \mathbf{f}_n &= K_n \ f_1(r) \ \mathbf{n} \\ \mathbf{f}_t &= -K_f \ |\mathbf{f}_n| / K_n \ (\triangle \mathbf{v} \cdot \mathbf{t}_v) \ \mathbf{t}_v \end{aligned} \tag{30}$$

where, r is the nearest distance from the fluid particle to the boundary, which is directly read from the ADF. $\mathbf{n}$ is the unit collision normal defined as the gradient of the distance field at the fluid particle position. Whereas $\mathbf{t}$ is unit projection of approach velocity of the fluid particle $\triangle \mathbf{v}$ onto the tangent plane. After experimentation with various forms of the penalty function f_1, we found Perlin's *gain* function [Perlin 1985] to be most suitable. The following figure illustrates the gain function

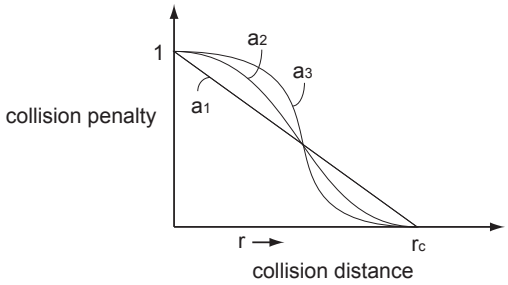


Figure 14: Penalty Function as Perlin's gain function

The collision penalty is zero for r above the collision distance r_c, whereas for $r < r_c$ it increases to one. One can adjust how fast the penalty increases with the decrease in r by varying the gain parameter a.

5 Hair-air, a mixture

We considered hair-hair and hair-body interactions in Sections 3 and 4. In this section, we address the hair-air interactions. Hair-air interactions are important for the following reasons:

- We would like to animate hair blown by wind.
- As the mass of an individual hair strand is very small compared to the skin friction drag created by its surface, the air drag is quite significant. Thus, most of the damping in hair dynamics comes from the air drag. The internal damping pertaining to dissipation in the deformation is quite negligible as compared to the air drag.
- Air plays a major role in hair-hair interaction. As a hair strand moves through air, it generates a boundary layer, which influences the neighboring hair strand even if the physical contact is minimum.
- Most importantly, hair volume affects the air field. Hair volume is not completely porous. Thus it acts as a partial obstacle to the wind field, to alter it.

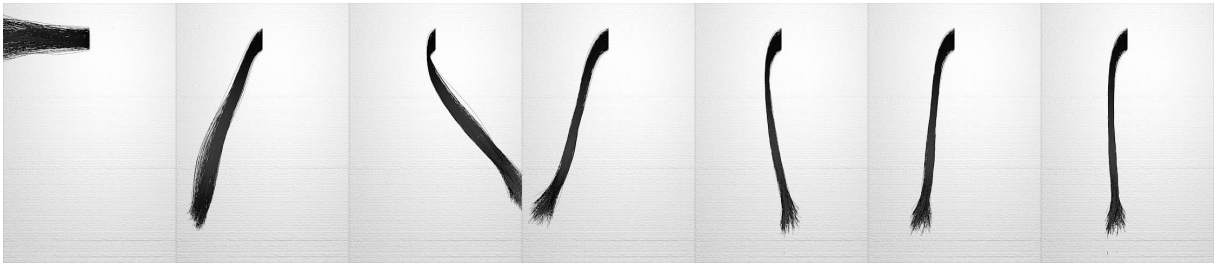


Figure 15: Cantilever of Hair – Moderate Air Drag

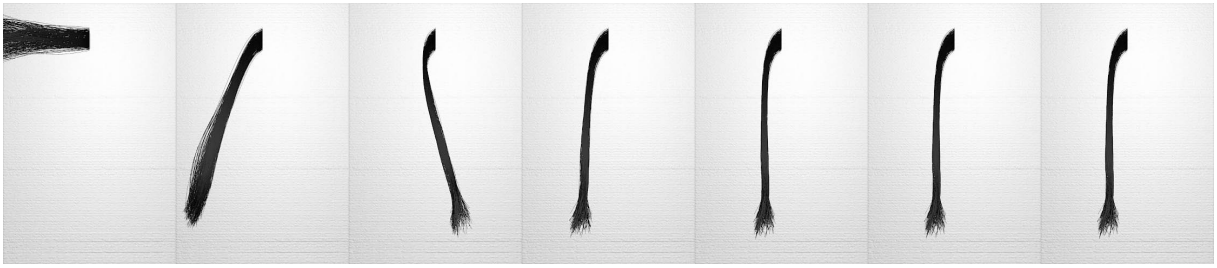


Figure 16: Cantilever of Hair – High Air Drag

Initially we thought of a very simple model for adding wind effects in the hair animation. There are significant advances in computer graphics for modeling turbulent gaseous fields [Stam 1997]. These models are mostly empirical. We incorporate the effect of the field by adding extra force to each of the smoothed particles $\hat{\mathbf{f}}_{di} = \mu(\hat{\mathbf{v}_{wi}} - \hat{\mathbf{v}}_i)$ in addition to the interaction force $\hat{\mathbf{f}}_{ci}$. Here, $\hat{\mathbf{v}_{wi}}$ is the local wind velocity at particle i, expressed as the spatial vector and $\hat{\mathbf{v}}_i$ is the velocity of the particle i. μ is the drag coefficient, which is an animation parameter. Figures 15 and 16 illustrate how variation in the drag coefficient affects the motion of hair. For that purpose, we let the bunch of hair fall under gravity undergoing cantilever action. The two examples illustrate the successive frames of the animations corresponding to each cycle of the oscillation. Observe that in the first example, the bunch of hair makes two oscillations before coming to rest, whereas in the second example it makes hardly one oscillation. The air drag coefficient μ is double in the second example. In both the examples, hair exhibit high degree of damping as seen in real-life.

However, this strategy is a passive one. As mentioned early in the section, hair volume should also affect the wind field for more realistic animations, as it is not completely porous. Subsequently, we extend the hair continuum model. We postulate that the hair medium is a mixture of hair material and air. There are many ways to model a mixture; we model it in a very straightforward way. Let the two fluids, hair medium and air, have their own fluid dynamics. We just link the two by adding extra drag force to the momentum equation. Thus, the SPH forms of the equations for the hair-air mixture are:

$$\frac{d\rho_i^h}{dt} = \sum_{j=1}^{N} m_j^h (\mathbf{v}_i^h - \mathbf{v}_j^h) W_{ij} \quad \text{hair continuity} \tag{31}$$

$$\frac{d\rho_k^w}{dt} = \sum_{l=1}^{M} m_l^w (\mathbf{v}_k^w - \mathbf{v}_l^w) W_{kl} \quad \text{air continuity} \tag{32}$$

$$\frac{d\mathbf{v}_i^h}{dt} = \frac{\mu(\mathbf{v}^a(\mathbf{r}_i^h) - \mathbf{v}_i^h)}{m_i} - \sum_{j=1}^{N} m_j^h \left(\frac{p_j^h}{{\rho^h}_j^2} + \frac{p_i^h}{{\rho^h}_i^2} + \prod_{ij}^{h}\right) \nabla_i W_{ij} \tag{33}$$

$$\frac{d\mathbf{v}_k^a}{dt} = \frac{\mu(\mathbf{v}^h(\mathbf{r}_k^a) - \mathbf{v}_k^a)}{m_k} - \sum_{l=1}^{M} m_l^a \left(\frac{p_l^a}{{\rho^a}_l^2} + \frac{p_k^a}{{\rho^a}_k^2} + \prod_{kl}^{a}\right) \nabla_k W_{kl} \tag{34}$$

Observe that there are two different sets of smoothed particles corresponding to two constituents of the mixture. $\mathbf{v}^a(\mathbf{r}_i^h)$ is air velocity experienced by the hair particle i, *i.e.* the velocity estimate of air at point $\mathbf{r}_i^h$ and vice versa for the air particle. Thus there is a coupling by the drag coefficient μ between the two fluid dynamics. For zero drag coefficient, hair and air will move without affecting each other. This model, although computationally expensive, results in very good animation of hair blown by wind – discussed in Section 8.

6 Numerical Integration

We assume that the reader is already conversant with numerical integration issues. For an extensive discussion on numerical methods we refer to standard text book – "Numerical Recipes in C: The Art of Scientific Computing" [Press et al. 1993]. The sole purpose of developing hair stiffness dynamics as dynamics of serial rigid multi-body chain is to have dynamical equations which are non-stiff. Thus it is quite sufficient to use an explicit numerical integration scheme. We use fifth order Runge-Kutta integration method with adaptive time stepping via error control [Press et al. 1993] for the purpose. The higher order integration scheme is not only more accurate, it has even better stability. So the user can choose large time steps which compensate for the extra computational overheads involved in higher order schemes. As the animator is readily given visual feedback, she can immediately judge the discrepancy in the results due to instability. She thus can choose appropriate constant time step. The more technically oriented animator, can have a more detailed control over the time step by setting appropriate minimum and maximum bounds on the time step. The adaptive integrator will choose the time step in the user specified range after analyzing the error estimate arising from the previous time step. Figure 17 illustrates the typical instability in the hair animation.

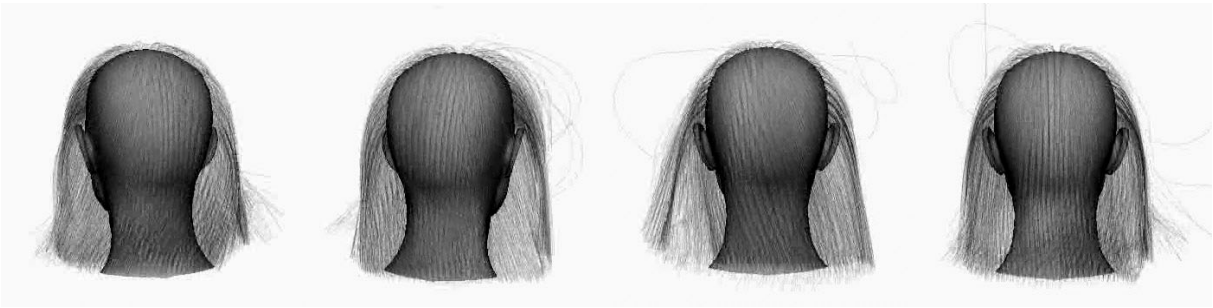


Figure 17: Instability in Hair Dynamics

Interestingly, we have observed that in the current implementation, where we use floating point precision, the user can not set the hair segment length l less than one centimeter for earth's gravity of 980cm/sec^2, no matter how small the time step is. We suspect this is related to numerical precision rather than the stability region of the dynamics.

However, the choice of explicit integration method has a major drawback. These methods have very narrow stability region. To achieve non-straight neutral shape of the hair we need to assign considerably high bending and torsional rigidities. Moreover, hair motion is highly damped demanding high degree of air-drag. Using the explicit integration methods, the hair simulation is almost always operating near bounds of the stability region. As a result, we are able to simulate straight to wavy hair, even though the model is able to accommodate the case of very curly hair.

We would like to use implicit integration methods, which we leave it as future work. In the case of chosen stiffness dynamics model, it is non-trivial to formulate the implicit integration schemes. It is not possible to express the Jaccobian of the stiffness dynamics using Featherstone's algorithm. We need to investigate alternative implicit methods that use only approximate estimation of the Jaccobian. We list this drawback in detail in the concluding section along with the future research possibilities.

Another source of instability that might occur is due to usage of relatively crude penalty method for the collision response. If hair moving with high velocity collides the boundary, one needs to apply large penalty force to avoid the penetration. This would demand smaller time steps to avoid instability. However, we would like to point out that the collision avoidance is part of the fluid boundary condition. Thus as the hair strand approaches the boundary, hair-hair interaction "makes the strand aware of the boundary" due to the fluid dynamic forces. Thus the hair-body collision is never a hard collision. In future we would like to replace the collision response by a more elaborate method – impulse dynamics. We full heartedly refer to the pioneering work by Brian Mirtich [Mirtich 1996], where he integrated the rigid-body impulse dynamics into the rigid multi-body dynamics framework.

7 Implementation Issues

The developed models for individual hair dynamics and hair-hair, hair-body, hair-air interactions are quite elaborate. Naturally, they are computationally intensive. Thus, a meticulous implementation is desired to make the methodology viable even with today's ever growing desktop computing power. In this section we discuss some of the implementation issues.

7.1 Data-parallel Implementation of Single Hair Dynamics

Modeling a hair strand as a rigid multi-body serial chain accurately captures all the relevant modes of motion and stiffness dynamics. Formulating only the exact number of relevant DOFs, *i.e.* bending and torsion, we have removed the source of the stiff equations of motion associated with the high tensile rigidity of the hair strand. Hence, we obtain an advantage in terms of possible large simulation time steps, even though the dynamics calculations are a bit involved.

We keep the length of the hair segment per hair strand constant. We also align the hair segment's local coordinate system to the principal inertial axis. That way the 6x6 spatial inertia tensor takes a simple form with many zeros and is constant. Length being constant, the only variable part in the coordinate transformation, from one link to another, is rotation. Exploiting the special multi-body configuration of hair, we have symbolically reduced most of the Articulated Rigid Body Dynamics calculations and have fine-tuned it to be most efficient. The time complexity of algorithm is linear. This puts no restriction on the number of rigid segments we can have per hair strand.

Finally, we parallelize the task of the hair strand computation. We exploit four-way parallel Single Instruction Multiple Data (SIMD) capability of Pentium III processors. We sort the hair strands by their number of hair segments. Then we club four hair strands, equal in number of segments, as far as possible or we trim a few. We then can compute four hair strands at a time on a single processor. Additionally, we assume two to four CPUs are available to use. Using these strategies, we are able to simulate 10,000 hair strands, having 30 segments on an average, in less than 2 minutes for each frame.

7.2 Efficient implementation of Smoothed Particle Hydrodynamics

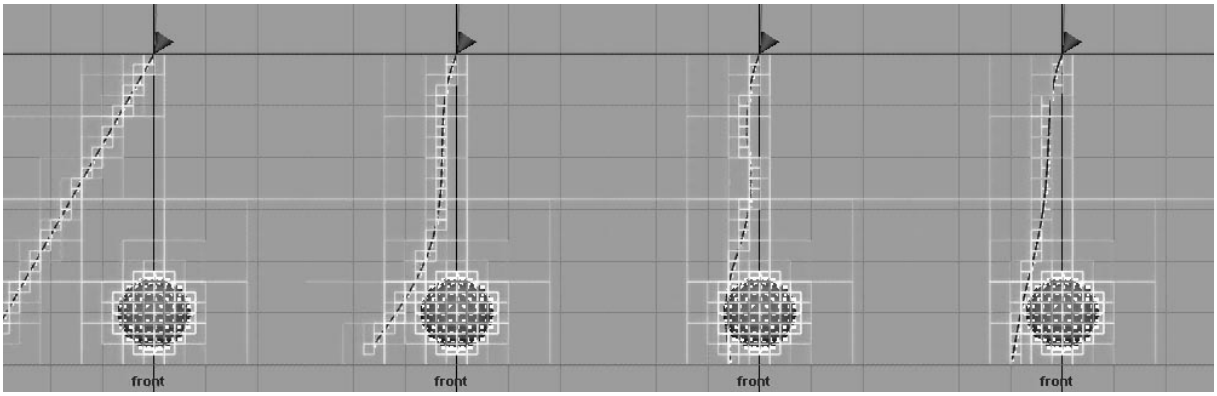


Figure 18: Hair-obstacle Collision – Use of Octree to Track Particle Interactions

The smoothed particle's kernel has compact support. The particle influences only its near neighbours, more precisely the ones that are in the circle of smoothing length. Thus the time complexity of the fluid computation is O(kn), where n is the total number of particles and k is the typical number of particles coming under influence of one particle. We still have to ensure that we use an efficient algorithm to locate the neighbours. There are many strategies for collision detection and neighbour search. For a detailed survey, refer to Lin and Gottschalk [Lin and Gottschalk 1998]. However, smoothed particles being point geometries, we use the Octree space partitioning. Vemuri *et al* [Vemuri et al. 1998] used the Octree for granular flow which is very similar to our application.

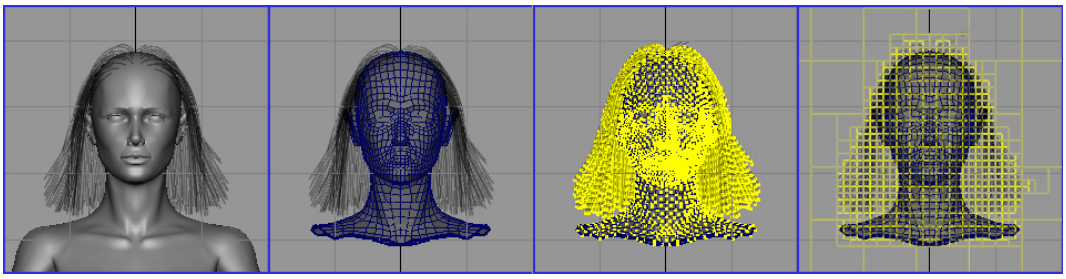


Figure 19: Approximate geometry, Smoothed particles, Octree

8 Results

We report three short animations using the described methodology. They are in increasing order of scene complexity. However, they utilize the same underlying models discussed so far. The simplest of the animations highlight a multitude of the dynamics in minute detail and the more complex ones illustrate the effectiveness of the methodology in animating real life hair animations. In the end we discuss animating hair for a dance sequence.

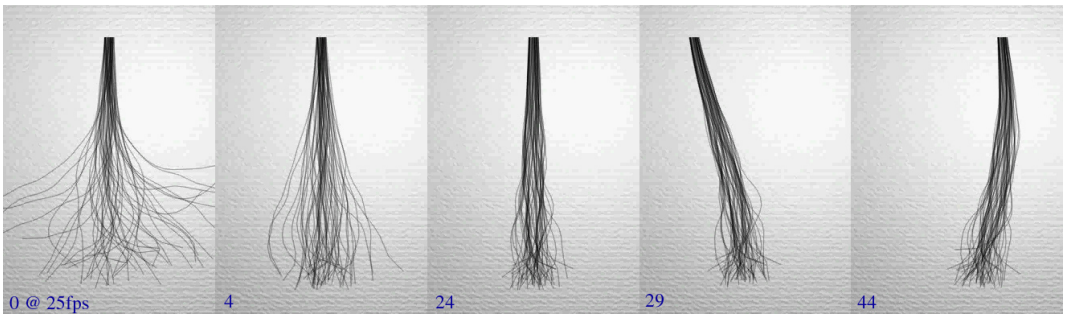


Figure 20: Starting from the initial spread, individual hair strands collapse under gravity. As they get close, the pressure built up in the "hair fluid" retains the volume (frame 24). In the subsequent frames, the body forces and hair-air interaction is prominent

In the first animation, from the initial spread, individual hair strands collapse under gravity. Hair strands have their shape memory working against gravity. Otherwise they would have straightened up at frame 24. Also, as the hair strands get close, the pressure builds up due to increase in the number density in the "hair fluid", which further retains the volume, throughout the animation, by keeping individual hair apart. The inertial forces and the influence of air are evident in the oscillatory motion of hair. The air drag is most effective towards the tip of hair strands. Observe the differential motion between the tips. Hair strands on the periphery experience more air drag than the interior ones. This is only possible due to the fluid-hair mixture model; the movement of hair does set air in motion like a porous obstacle.

Figure 21: Free fall of hair – Hair volume, modeled as fluid, falls freely under gravity. However, the individual hair's length constraint quickly restricts the free falling motion to give it a bounce. At the same time, "hair fluid" collides with the body and bursts away sidewise.

The second animation scenario is to illustrate the "fluid" motion of hair without loosing the character of individual hair. The hair volume starts falling freely under gravity. Quickly, the individual hair's length constraint and stiffness restricts the free falling motion to give it a bounce, towards the end of the free fall (frame 53). At the same time, "hair fluid" collides with the body and bursts away sidewise (frame 70). The air interaction gives an overall damping. Observe that the hair quickly settles down, even after the sudden jerk in the motion, due to air drag and hair friction with the body.

Figure 22: Hair blown by wind – The "fluid hair" model is extended to "hair-air mixture". Indeed, the complex hair-hair, hair-air and hair-body interactions are modeled under a single framework.

The third animation in Figure 22 exclusively illustrates the effectiveness of the model in animating hair blown by wind. Needless to say that there is an influence of airfield on individual hair. More importantly, body and hair volume acts as a full and partial obstacle to air altering its flow.

Figure 23: Dance Sequence Demonstrating Hair Animation by Nedjma Kadi and Sunil Hadap

The animation methodologies are implemented in a Maya plugin – MIRAHairSimulation. We next present a representative animation sequence which is the result of typical usage of MIRAHairSimulation by animators at MIRALab, University of Geneva.

Figure 23 is a dance sequence of around 1 minute. The dance is motion captured using Vicon8 optical motion tracking system. The animators used 3ds max for setting up the body deformations and the Fashionizer, MIRALab's flagship cloth simulation system for achieving the cloth animation. The resulting animated mesh sequence was imported into Maya for adding hair animation. The process of setting up hair animation and computing hair simulation including hair rendering towards the satisfactory results took around 2 weeks. The dynamic hairstyle has around 8,000 hair clumps and the total number of polygons for the dress and the body is around 20,000. It took on an average 243 seconds for computing one frame of the animation, whereas the rendering of the sequence took on an average 370 seconds per PAL frame. We used RenderMan for rendering of the dance sequence. The simulation was computed on a workstation having Intel Xeon 2.2MHz processor with 2GB RAM.

9 Summary

We have developed a powerful hair dynamics model.

- Stiffness Dynamics – We have given an elaborate model for the stiffness and inertial dynamics of an individual hair strand. We treat the hair strand as a serial rigid multi-body system. This reduced coordinate formulation gives very accurate and effective representation for the dynamics of non-straight (wavy) hair by providing precise parametric definition of the bending and the torsion in three dimensions. The formulation also partly eliminates the stiff numerical equations enabling large time-steps, thus faster simulations.

- Interaction Dynamics – The hair-hair, the hair-air interactions and the accurate hair-body collisions were one of the few unsolved problems in computer graphics – until recently. We have exclusively addressed this problem by making a paradigm shift and treating hair as a continuum. We model the hair-hair, the hair-body and the hair-air interactions in a unified way using fluid dynamics. The continuum assumption proves to be a very strong model for otherwise very complex interaction phenomena.

10 Limitations and Future Work

- We have successfully attempted to capture the detailed dynamics of straight to wavy hair typically found in moderately complex hairstyles. However the problem of animating complex hairstyles, i.e. the hairstyles involving curly hair or the hairstyles having intricate geometry, still eludes us. By adopting the reduced coordinate formulation, we hoped to have completely eliminated the stiff differential equations. However, we learned that very high bending and torsional rigidity is required to firmly maintain the intricate geometric definition of the complex hairstyle, under gravity. At the same time the hair motion is highly damped. Both these problems clearly put the inherent stability of the implicit integration methods in high demand.

  Unfortunately, it is not possible to express the Jaccobian of the system using the articulated rigid body dynamics that we have used. This is due to the iterative nature of the inertial dynamics formulation. In future we would like to explore the possibility of expressing the approximate Jaccobian that corresponds to the stiff part of the differential equations and use implicit integration methods based on that.

  Another possibility is to follow the constrained dynamics. The constrained dynamics has the best possibilities of using implicit integration methods. These methods may also provide better collision response possibilities. However, the constrained dynamics methods severely suffer from the numerical inaccuracies in terms of drift in the constraints. One needs to use sophisticated numerical methods to avoid the problem associated with the drift.

  The original idea of using the spring-mass-hinge systems still remains to be one of the strong possibilities we would like to explore. The current advances in the implicit integration methods fade away the limitations of these methods of being "stiff" in nature due to elongation constraints that are expressed as stiff springs. However, we would have to investigate the ways of expressing the stiffness dynamics of bending and torsion, in the framework of the spring-mass-hinge system, which in our primary opinion is complex.

- We have implemented the collision response which is essentially the penalty method. This method does not have good stability characteristics. We have barely managed to handle various hair-body collision situations arising in real-life complex motions such as the dance sequence.

  We would like to explore the Brian Mirtich's work of impulsed based collision response [Mirtich 1996], which is unconditionally stable, although it is numerically expensive.

References

Anjyo, K., Usami, Y., and Kurihara, T. 1992. A simple method for extracting the natural beauty of hair. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques (SIGGRAPH)*, ACM SIGGRAPH.

Baraff, D. 1996. Linear-time dynamics using lagrange multipliers. *Proceedings of SIGGRAPH 96* (August), 137–146.

Chang, J., Jin, J., and Yu, Y. 2002. A practical model for mutual hair inteactions. In *Proceedings of Symposium on Computer Animation*, ACM SIGGRAPH, San Antonio, USA.

Cohen, J., Lin, M., Manocha, D., and Ponamgi, K. 1995. I-collide: An interactive and exact collision detection system for large-scaled environments. In *Proceedings of ACM Symposium on Interactive 3D Graphics*, 189–196.

Daldegan, A., Magnenat-Thalmann, N., Kurihara, T., and Thalmann, D. 1993. An integrated system for modeling, animating and rendering hair. *Computer Graphics Forum, Proceedings of Eurographics 12*, 3, 211–221.

Featherstone, R. 1987. *Robot Dynamics Algorithms*. Kluwer Academic Publishers.

Frisken, S., Perry, R., Rockwood, A., and Jones, T. 2000. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of ACM SIGGRAPH*, 249–254.

Gascuel, J., Cani, M., Desbrun, M., Leroy, E., and Mirgon, C. 1996. Smoothed particles: A new paradigm for animating highly deformable bodies. In *6th Eurographics Workshop on Animation and Simulation'96*, Paris.

Hadap, S. 2003. *Hair Simulation*. PhD thesis, MIRALab, CUI, University of Geneva. No 3416, Science Faculty.

Hockney, R. W., and Eastwood, J. W. 1988. *Computer Simulation Using Particles*. Adam Hilger, March. ISBN: 0852743920.

Lee, D.-W., and Ko, H.-S. 2001. Natural hairstyle modeling and animation. *Graphical Models 63*, 2 (March), 67–85.

Lin, M., and Gottschalk, S. 1998. Collision detection between geometric models: A survey. In *Proceedings of IMA Conference on Mathematics of Surfaces*.

Magnenat-Thalmann, N., Hadap, S., and Kalra, P. 2000. State of the art in hair simulation. In *Proceedings of International Workshop on Human Modeling and Animation*, Korea Computer Graphics Society, Seoul, Korea, 3–9.

Mirtich, B. 1996. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California, Berkeley.

Monaghan, J. J. 1992. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics 30*, 543–574.

Morris, J. P. 1995. An overview of the method of smoothed particle hydrodynamics. AGTM Preprints, University of Kaiserslautern.

Panton, R. L. 1995. *Incompressible Flow*, 2nd edition ed. John Wiley, December.

Perlin, K. 1985. An image synthesizer. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques (SIGGRAPH)*, ACM SIGGRAPH, 287–296.

Plante, E., Cani, M.-P., and Poulin, P. 2001. A layered wisp model for simulating interactions inside long hair. In *Proceedings of Eurographics Workshop, Computer Animation and Simulation*, EUROGRAPHICS, Manchester, UK.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. 1993. *Numerical Recipes in C: The Art of Scientific Computing*, 2nd edition ed. Cambridge Univ Press.

Rosenblum, R., Carlson, W., and Tripp, E. 1991. Simulating the structure and dynamics of human hair: Modeling, rendering and animation. *Journal of Visualzation and Computer Animation 2* (June), 141–148. John Wiley.

Stam, J. 1997. Stochastic dynamics: Simulating the effects of turbulence on flexible structures. *Computer Graphics Forum 16*, 3 (August), 159–164.

Vemuri, B. C., Chen, L., Vu-Quoc, L., Zhang, X., and Walton, O. 1998. Efficient and accurate collision detection for granular flow simulation. *Graphical Models and Image Processing 60*, 5 (November), 403–422.

Fast and Reliable Collision Culling using Graphics Hardware

Naga Govindaraju Ming Lin Dinesh Manocha
Department of Computer Science
University of North Carolina at Chapel Hill

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

Keywords: interference detection, graphics hardware, sampling, interactive computer graphics.

1 Introduction

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

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

Main Results: We present a novel algorithm for fast and reliable collision culling between triangulated models in a large environment using GPUs. We perform visibility queries to eliminate a subset of primitives that are not in close proximity, thereby reducing the number of pairwise tests that are performed for exact proximity computation. We show that the *Minkowski sum* of each primitive with a sphere provides a conservative bound for 2.5D overlap tests. We compute a bounding offset approximation for each primitive based on the Minkowski sum; the radius of the bounding offset is a function of the viewpoint and depth-buffer resolutions. We render the bounding offsets using orthographic projections. Our algorithm guarantees that no collisions will be missed due to limited frame-buffer precision or quantization errors during rasterization. The key advantages of our approach are:

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

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

2 Related Work

The problem of collision detection has been well studied for more than three decades. See recent surveys in [Lin and Gottschalk 1998] and [Jimenez et al. 2001] for an overview. Prior algorithms for collision detection between triangulated models can be classified into three broad categories: object-space culling, image-space intersection computation, and hybrid approaches.

Object-space culling: Most of the commonly used techniques to accelerate collision detection between two objects utilize spatial data structures, including spatial partitioning and bounding volume hierarchies. These representations are used to cull away portions of each object that are not in close proximity. Typically, these representations are built in a pre-processing stage to accelerate runtime queries. In practice, they work well for rigid objects. However, the overhead of recomputing the hierarchy on the fly for deformable models can be quite significant [Baciu and Wong 2002; Hoff et al. 2001].

Image-space interference computation: Several algorithms have used graphics hardware for interference and collision computations [Baciu et al. 1998; Baciu and Wong 2002; Heidelberger et al. 2003; Hoff et al. 2001; Knott and Pai 2003; Myszkowski et al. 1995; Rossignac et al. 1992; Shinya and Forgue 1991; Vassilev et al. 2001]. These algorithms require no preprocessing; they work well on commodity GPUs. However, they have some limitations. First, they can detect a collision up to viewport resolution. The accuracy of collision detection also varies based on the relative distance between the objects, i.e. collision queries are less accurate if the objects are separated by distances greater than their average size. Second, most of these algorithms need to read back the color or depth buffer contents for further processing and readbacks can be slow on current graphics systems [Knott and Pai 2003; Govindaraju et al. 2003].

Hybrid methods: Hybrid algorithms combine some of the benefits of the object-space and image-space approaches. Kim et al. [2002] compute the closest distance from a point to the union of convex polytopes using the GPU, refining the answer on the CPU. Govindaraju et al. [2003] use occlusion queries on the GPU to cull away objects that are not colliding with others. Heidelberger et al. [2003] compute layer depth images (LDIs) on the GPU, use the LDIs for explicit computation of the intersection volumes between two closed objects, and perform vertex-in-volume tests. In all these cases, GPU-based techniques are used to accelerate the overall computation. However, viewport resolution governs the accuracy of these algorithms.

3 Reliable Culling using GPUs

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

perform conservative and reliable culling.

3.1 Overlap tests and Sampling Errors

Interference computation algorithms employ GPUs to perform either 2D overlap tests using color and stencil buffers or 2.5D overlap tests with additional depth information. The 2.5D overlap tests are less conservative and can be performed using occlusion queries on current graphics processors [Knott and Pai 2003; Govindaraju et al. 2003].

Visibility-based overlap tests: Govindaraju et al. [Govindaraju et al. 2003] describe the use of visibility computations to check whether two primitives, P_1 and P_2, overlap. The approach chooses a view direction and checks whether P_1 is fully visible with respect to P_2 along that direction. If P_1 is fully visible then there exists a separating surface between P_1 and P_2. We call this the *visibility-based-overlap* (**VO**) query, which provides a sufficient condition that the two primitives do not overlap and is illustrated in figure 1. The **VO** query is performed efficiently on GPUs. Using three or less than three mutually orthogonal orthographic views, many complex objects that are in close proximity (as shown in figure 1) can be pruned. However, due to limited viewport and frame buffer resolution, **VO** queries can miss collisions and is typical of any GPU-based interference detection algorithm. Our goal is to use *reliable* VO queries on the GPUs for pruning complex configurations as in Fig 1 efficiently, without missing any collisions.

CULLIDE: We now briefly describe CULLIDE [Govindaraju et al. 2003] which performs **VO** queries between multiple objects and computes a potentially colliding set (PCS) of objects. Given n objects that are potentially colliding $O_1, ..., O_n$, Govindaraju et al. describe a linear time two-pass rendering algorithm to test if an object O_i is fully visible against remaining objects, along a view direction. The algorithm uses occlusion queries to test if an object is fully visible or not. To test if an object O is fully visible against a set of objects S, we first render S into the frame buffer. We then set the depth function to GL_GEQUAL and disable depth writes. The object O is rendered using an occlusion query. If the pixel pass count returned by occlusion query is zero, then the object O is fully visible. Govindaraju et al. [Govindaraju et al. 2003] also extend their algorithm to prune sub-objects of an object that do not overlap with other objects in the environment. The pseudo-code to compute PCS of sub-objects is described below

- **First pass:**
  1. Clear the depth buffer (use orthographic projection)
  2. For each object O_i, $i = 1, .., n$
     - Disable the depth mask and set the depth function to GL_GEQUAL.
     - For each sub-object T_k^i in O_i
       Render T_k^i using an occlusion query
     - Enable the depth mask and set the depth function to GL_LEQUAL.
     - For each sub-object T_k^i in O_i
       Render T_k^i
  3. For each object O_i, $i = 1, .., n$
     - For each sub-object T_k^i in O_i
       Test if T_k^i is not visible with respect to the depth buffer. If it is not visible, set a tag to note it as fully visible.
- **Second pass:**

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

The view directions are chosen along the world-space axes and collision culling is performed using orthographic projections. We demonstrate accurate collision culling using reliable **VO** queries in CULLIDE.

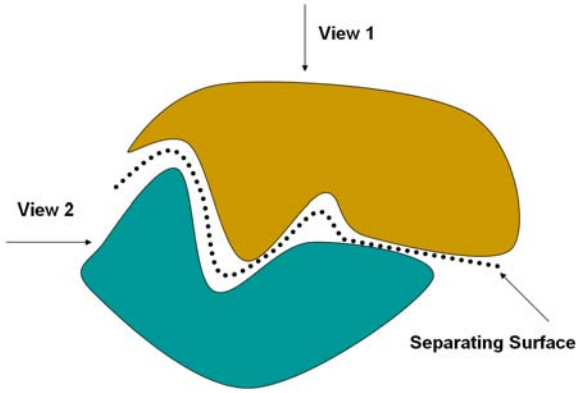


Figure 1: *In this figure, the objects are not colliding. Using view* 1*, we determine a separating surface with unit depth complexity along the view and conclude from the existence of such a surface that the objects are not colliding. This is a sufficient but not a necessary condition. Observe that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

3.2 Sampling Issues and Notation

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

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

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

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

3.3 Reliable VO Queries

To overcome the problems due to viewport and depth-precision resolution, we compute a bounding offset for each primitive. Instead of rasterizing the original primitives, we rasterize these bounding offsets and use them to perform VO queries between P_1 and P_2. Moreover, we will show that our queries are conservative irrespective of the viewport resolution and depth-buffer precision.

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

Let B be an axis-aligned cube centered at the origin with dimension p. Given two primitives, P_1 and P_2, let Q be a point on their line of intersection. We use the concept of *Minkowski sum* of

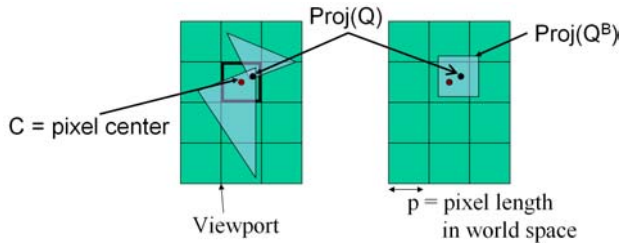


Figure 2: *Sampling errors: Q is a point on the line of intersection between two triangles in $3D$. The left figure highlights its orthographic projection in the screen space. The intersection of two triangles does not contain the center of the pixel (C) and therefore, we can miss a collision between the triangles. Q^B is the Minkowski sum of Q and an axis-aligned bounding box (B) centered at the origin with dimension p. Q^B translates B to the point Q. During rasterization, the projection of Q^B samples the center of pixel and generates at least two fragments that bound the depth of Q.*

a primitive P with B, ($P^B = P \oplus B$), which can be defined as: $\{\boldsymbol{p} + \boldsymbol{b} \mid \boldsymbol{p} \in P,\ \boldsymbol{b} \in B\}$. Next we show that $P \oplus B$ can be used to perform reliable VO queries. We first state two lemmas without proof and use them to derive the main result as a theorem.

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

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

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

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

3.4 Collision culling and bounding offsets

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

The boundary of an exact offset of a triangle consists of piecewise linear and spherical surfaces. Instead of using the exact offset, we compute a bounding offset for each triangle that is cheaper to compute and render. We bound the offset of a triangle by using a single OBB (oriented bounding box). Given a triangle T, we compute the tightest fitting rectangle R that encloses T; one of its axes is aligned with the longest edge of the triangle. We compute the OBB for a triangle as the Minkowski sum of B and R. The width of the OBB, along a dimension orthogonal to the plane containing R, is set equal to $\sqrt{3}p$. The bounding offset of a triangulated object is the union of OBBs of each triangle (see Fig. 3). We will render

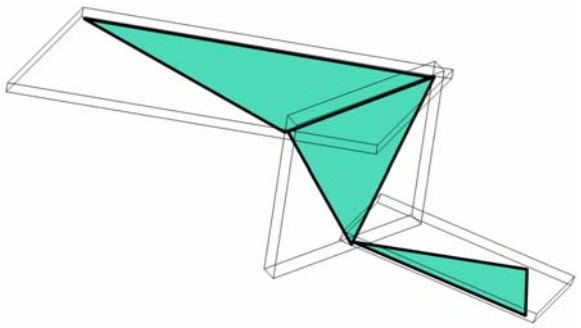


Figure 3: *This image shows an object with three triangles and its bounding offset representation in wireframe. The bounding offset is represented as the union of OBBs of each triangle. In practice, this bounding offset is a very tight fitting bounding volume.*

this bounding offset by rendering each OBB separately and perform VO queries. In practice, this is a very tight bounding volume for an object, as compared to using a single sphere, AABB (axis-aligned bounding box) or an OBB that encloses the object.

3.5 Accuracy

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

4 Implementation and Performance

We have integrated our culling algorithm with CULLIDE [Govindaraju et al. 2003] to perform reliable collision detection between objects in a complex environment. As described in section 3, CULLIDE uses **VO** queries to perform collision culling on GPUs. We extend CULLIDE to perform reliable collision culling on GPUs by using reliable **VO** queries described above. For each primitive in the PCS, we compute its bounding offset (i.e. union of OBBs) representation and use the bounding offset representations in CULLIDE to test if the primitives belong to PCS or not.

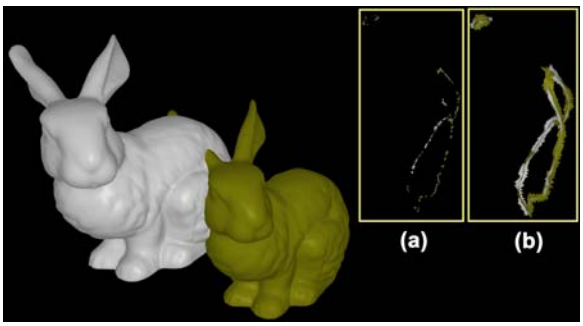


Figure 4: *Reliable interference computation: This image highlights the intersection set between two bunnies, each with 68K triangles. The top right image (b) shows the output of FAR and the top left image (a) highlights the output of CULLIDE running at a resolution of 1400×1400. CULLIDE misses many collisions due to the viewport resolution and sampling errors.*

Our collision detection algorithm, FAR, proceeds in three steps. First we compute the PCS at the object level. We use sweep-and-prune [Cohen et al. 1995] on the PCS to compute the overlapping pairs at the object level. Next we compute the PCS at the sub-

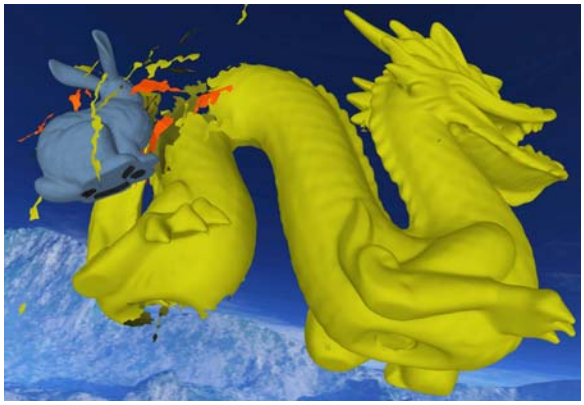


Figure 5: *Breaking object scene: In this simulation, the bunny model falls on the dragon which eventually breaks into hundreds of pieces. FAR computes collisions among the new pieces of small objects introduced into the environment and takes* 30 *to* 60 *msec per frame.*

object level and the overlapping pairs. Finally, we perform exact interference tests between the triangles on the CPU [Moller 1997].

We have implemented several optimizations in our system. Rendering bounding offset representations requires nearly twice the amount of fill when compared to that generated by original primitives. As the offset representations for each triangle is closed, we can reduce the fill requirements for our algorithm by a factor of 2 using face-culling. In our optimized algorithm, we cull front faces while rendering offset representations with occlusion queries and we cull back faces while rendering offset representations to the frame buffer. These operations can be performed efficiently using *back face-culling* on graphics hardware. We also reduce the number of occlusion queries in the second pass of our algorithm by testing only those primitives whose offset representations are fully visible in first pass.

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

- **First pass:**

  1. Clear the depth buffer (use orthographic projection)
  2. For each object O_i, $i = 1, .., n$
     - Disable the depth mask and set the depth function to GL_GEQUAL.
     - Enable back face-culling to cull front faces.
     - For each sub-object T_k^i in O_i
       Render offset representation of T_k^i using an occlusion query
     - Enable the depth mask and set the depth function to GL_LEQUAL.
     - Enable back face-culling to cull back faces.
     - For each sub-object T_k^i in O_i
       Render offset representation of T_k^i
  3. For each object O_i, $i = 1, .., n$
     - For each sub-object T_k^i in O_i
       Test if T_k^i is not visible with respect to the depth buffer. If it is not visible, set a tag to note it as fully visible.

- **Second pass:**

  Same as First pass, except that the two "For each object" loops are run with $i = n, .., 1$ and we perform occlusion queries only if the primitive is fully visible in first pass.

We have implemented FAR on a Dell precision workstation with a 2.8GHz Xeon processor, 1 GB of main memory, and a NVIDIA GeForce FX 5950 Ultra graphics card. We use a viewport resolution of 1400×1400 to perform all the computations. We improve the rendering throughput by using vertex arrays and use GL_NV_occlusion_query to perform the visibility queries. We have tested our algorithm on three complex scenes and have compared its culling performance and accuracy with some prior approaches.

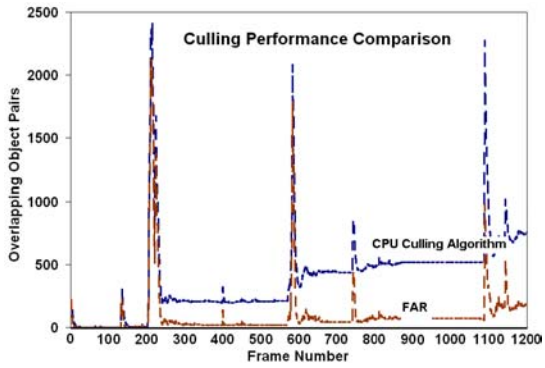


Figure 6: *Relative culling performance on breaking objects scene: This graph highlights the improved culling performance of our algorithm as compared to a CPU-based (object-space) culling algorithm that uses AABBs (axis-aligned bounding boxes) to cull away non-overlapping pairs. FAR reports* 6.9 *times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

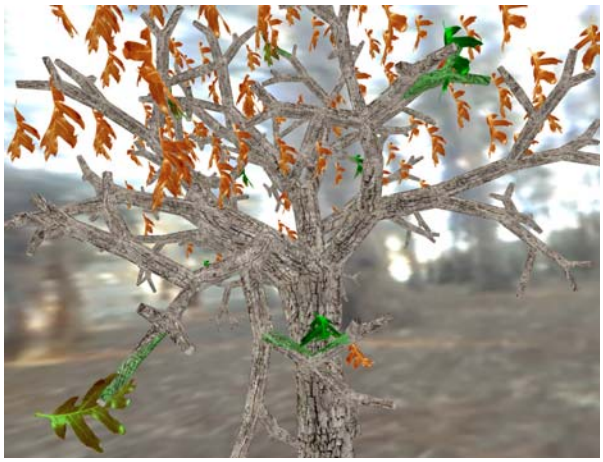


Figure 7: *Tree with falling leaves: In this scene, leaves fall from the tree and undergo non-rigid motion. They collide with other leaves and branches. The environment consists of more than* 40*K triangles and* 150 *leaves. FAR can compute all the collisions in about* 35 *msec per time step.*

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

We compared the culling performance of our GPU-based reliable culling algorithm with an implementation of the sweep-and-prune algorithm available in I-COLLIDE [Cohen et al. 1995]. The sweep-and-prune algorithm computes an axis-aligned bounding box (AABB) for each object in the scene and checks all the AABBs for pairwise overlaps. Fig. 6 shows the comparison between the culling efficiency of AABB-based algorithm vs. FAR. Overall, FAR returns 6.9 times fewer overlapping pairs. This reduction occurs mainly because FAR uses much tighter bounding volumes, i.e. the union of OBBs for an object as compared to an AABB and is able to cull away more primitive pairs.

Interference computation between complex models: In this scene, we compute all the overlapping triangles pairs between a 68K triangles bunny that is moving with respect to another bunny, also with 68K triangles. The bunnies are deeply penetrating and the intersection boundary consists of $2,000 - 4,000$ triangle pairs. In

this case, the accuracy of FAR equals that of a CPU-based algorithm using 32-bit IEEE floating point arithmetic. In contrast, CULLIDE misses a number of overlapping pairs while using a viewport resolution of $1,400 \times 1,400$. The intersection sets computed by FAR and CULLIDE are shown in Fig. 4.

Multiple objects with non-rigid motion: This scene consists of a non-rigid simulation in which leaves fall from the tree, as shown in Fig. 7. We compute collisions among the leaves of the tree and among the leaves and branches of the tree. Each leaf is represented using 156 triangles and the complete environment consists of 40K triangles. The average collision detection time is 35 msec per time step.

5 Applications

We have also applied *reliable* **VO** queries for improving the accuracy of other image-based algorithms on GPU.

- **Shadow Volumes:** Recently, Lloyd et al. [Lloyd 2004] propose a clamping algorithm on GPUs for reducing the fill requirements of shadow volumes. The clamping algorithm uses **VO** queries on GPUs for testing if portions of shadow volumes are fully visible or not. Due to image-precision errors, certain portions can be inaccurately classified as fully visible. Using reliable **VO** queries, we are able to clamp shadow volumes upto object-precision.

- **Localized Distance Culling** Many algorithms aim to compute all pairs of objects whose separation distance is less than a constant distance D. In this case, we modify GPU-based culling algorithms to cull away primitives whose separation distance is more than D. We can easily modify the culling algorithm presented above to perform this query. We compute the offset of each primitive by using a sphere of radius $\frac{D}{2} + \frac{\sqrt{3}p}{2}$, rasterize these offsets and prune away a subset of primitives whose separation distance is more than D. Note that CULLIDE only detects interfering triangles whereas our approach extends it to perform localized distance culling as well as accurate collisions computations.

- **Local Distance Fields:** Localized distance culling also enables us for computing fast local distance fields. Only the primitives that are not pruned using our approach contribute to the local distance field.

- **Continuous Collision Detection:** Recently, a system **Avatar** [Redon et al. 2004] uses CULLIDE to perform continuous collision detection on GPUs for virtual environments.

6 Analysis and Limitations

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

Efficiency: Three factors govern the running time of our algorithm: bounding offset computation, rendering the bounding offsets and occlusion queries. The cost of computing the OBBs for each primitive is very small. The cost of rendering the OBBs on the GPUs is mainly governed by the transformations. In our current implementation, we have achieved rendering rates of $40M$ triangles per second. Finally, our algorithm uses occlusion queries to perform VO queries. These queries can be fill bound for large objects. The current implementation of these queries is not optimized, yet we are able to perform 1.2 million queries per second. FAR is able to compute all the collisions between models composed of tens of thousands of primitives at interactive rates. In more complex environments (e.g. with millions of triangles), rendering and occlusion queries can become a bottleneck. However, given the growth rate of GPU performance (at a rate faster than Moore's law) and increasing bus bandwidth based on PCI-X, we expect that our algorithm can handle more complex models in the near future.

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

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

7 Conclusion and Future Work

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

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

References

BACIU, G., AND WONG, S. 2002. Image-based techniques in a hybrid collision detector. *IEEE Trans. on Visualization and Computer Graphics*.

BACIU, G., WONG, S., AND SUN, H. 1998. Recode: An image-based collision detection algorithm. *Proc. of Pacific Graphics*, 497–512.

COHEN, J., LIN, M., MANOCHA, D., AND PONAMGI, M. 1995. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proc. of ACM Interactive 3D Graphics Conference*, 189–196.

GOVINDARAJU, N., REDON, S., LIN, M., AND MANOCHA, D. 2003. Cullide: Interactive collision detection between complex models in large environments using graphics hardware. *Proc. of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 25–32.

HEIDELBERGER, B., TESCHNER, M., AND GROSS, M. 2003. Real-time volumetic intersections of deforming objects. *Proc. of Vision, Modeling and Visualization*.

HOFF, K., ZAFERAKIS, A., LIN, M., AND MANOCHA, D. 2001. Fast and simple 2d geometric proximity queries using graphics hardware. *Proc. of ACM Symposium on Interactive 3D Graphics*, 145–148.

JIMENEZ, P., THOMAS, F., AND TORRAS, C. 2001. 3d collision detection: A survey. *Computers and Graphics 25*, 2, 269–285.

KIM, Y., OTADUY, M., LIN, M., AND MANOCHA, D. 2002. Fast penetration depth computation for physically-based animation. *Proc. of ACM Symposium on Computer Animation*.

KNOTT, D., AND PAI, D. 2003. Cinder: Collision and interference detection in real-time using graphics hardware. *Proc. of Graphics Interface*, 73–80.

LIN, M., AND GOTTSCHALK, S. 1998. Collision detection between geometric models: A survey. *Proc. of IMA Conference on Mathematics of Surfaces*.

LLOYD, B., WENDT, J., GOVINDARAJU, G., AND MANOCHA D. 2004. CC Shadow Volumes. *UNC Technical Report*.

MOLLER, T. 1997. A fast triangle-triangle intersection test. *Journal of Graphics Tools*.

MYSZKOWSKI, K., OKUNEV, O. G., AND KUNII, T. L. 1995. Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer 11*, 9, 497–512.

REDON, S., KIM, Y., LIN, M., MANOCHA, D., AND TEMPLEMAN, J. 2004. Interactive and continuous collision detection for avatars in virtual environments. In *Proc. of IEEE VR Conference*.

ROSSIGNAC, J., MEGAHED, A., AND SCHNEIDER, B. 1992. Interactive inspection of solids: cross-sections and interferences. In *Proceedings of ACM Siggraph*, 353–60.

SHINYA, M., AND FORGUE, M. C. 1991. Interference detection through rasterization. *The Journal of Visualization and Computer Animation 2*, 4, 131–134.

VASSILEV, T., SPANLANG, B., AND CHRYSANTHOU, Y. 2001. Fast cloth animation on walking avatars. *Computer Graphics Forum (Proc. of Eurographics'01) 20*, 3, 260–267.

Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware

Kenneth E. Hoff III, Tim Culver, John Keyser, Ming Lin, Dinesh Manocha

University of North Carolina at Chapel Hill
Department of Computer Science

Abstract: We present a new approach for computing generalized 2D and 3D Voronoi diagrams using interpolation-based polygon rasterization hardware. We compute a discrete Voronoi diagram by rendering a three dimensional distance mesh for each Voronoi site. The polygonal mesh is a bounded-error approximation of a (possibly) non-linear function of the distance between a site and a 2D planar grid of sample points. For each sample point, we compute the closest site and the distance to that site using polygon scan-conversion and the Z-buffer depth comparison. We construct distance meshes for points, line segments, polygons, polyhedra, curves, and curved surfaces in 2D and 3D. We generalize to weighted and farthest-site Voronoi diagrams, and present efficient techniques for computing the Voronoi boundaries, Voronoi neighbors, and the Delaunay triangulation of points. We also show how to adaptively refine the solution through a simple windowing operation. The algorithm has been implemented on SGI workstations and PCs using OpenGL, and applied to complex datasets. We demonstrate the application of our algorithm to fast motion planning in static and dynamic environments, selection in complex user-interfaces, and creation of dynamic mosaic effects.

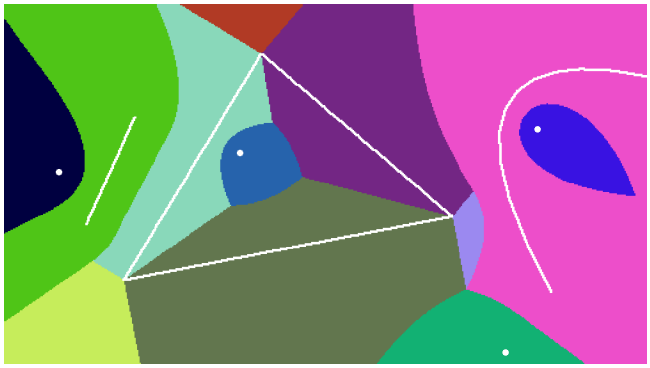
CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling; I.3.3 [Computer Graphics]: Picture/Image Generation.

Additional Key Words: Voronoi diagrams, graphics hardware, polygon rasterization, interpolation, motion planning, proximity query, medial axis, OpenGL, framebuffer techniques.

1 INTRODUCTION

Given a set of primitives, called Voronoi sites, a Voronoi diagram partitions space into regions, where each region consists of all points that are closer to one site than to any other. Voronoi diagrams have been used in a number of applications including visualization of medical datasets, proximity queries, spatial data manipulation, shape analysis, computer animation, robot motion planning, modeling spatial structures and processes, pattern recognition, and locational optimization. The concept of Voronoi diagrams has been around for at least four centuries, and since the 1970s, algorithms for computing Voronoi diagrams of geometric primitives have been developed in computational geometry and related areas.

e-mail: {hoff,culver,keyser,lin,dm}@cs.unc.edu
WWW: http://www.cs.unc.edu/~geom/voronoi/



Cover Plate: Discrete approximation of the generalized Voronoi diagram of four points, a line, a triangle, and one cubic Bézier curve computed interactively on a PC.

Good theoretical and practical algorithms are known for computing ordinary Voronoi diagrams of points in any dimension. Ordinary Voronoi diagrams can be generalized in many different ways by using different distance functions and site shapes. A common generalization is to compute the diagram for higher-order sites, such as lines and curves. This greatly increases the complexity since the boundaries of the diagram are composed of high-degree algebraic curves and surfaces, and their intersections; the boundaries of an ordinary point Voronoi diagram are linear. No practically efficient and numerically robust algorithms are known for constructing a topologically consistent, continuous representation of generalized Voronoi diagrams.

Given the practical complexity of computing an exact generalized Voronoi diagram, many authors have proposed approximate algorithms. Interesting approaches include computing the Voronoi diagram of a point-sampling of the sites, adaptively subdividing space to locate the Voronoi boundary, and point-sampling the space to form a volumetric representation of the diagram. In practice, these previous algorithms take considerable time and memory on large numbers of input sites, or are restricted in generality.

Main Contributions: In this paper, we present an approach that computes discrete approximations of generalized Voronoi diagrams to an arbitrary resolution using polygon rasterization hardware. Our contributions include:

1. Efficient methods to approximate the distance function, with bounded error, for points, lines, polygons, polyhedra, curves, and curved surfaces using a polygonal mesh that is linearly interpolated by graphics hardware.
2. Efficient algorithms to find Voronoi boundaries and neighbors, and to construct Delaunay triangulations.

3. Techniques to construct weighted and farthest-site generalized Voronoi diagrams in 2D and 3D.

4. Demonstration of the effectiveness of our approach to the following applications:

   - Fast motion planning in static and dynamic environments
   - Selection in complex user-interfaces
   - Generation of dynamic mosaics

The resulting techniques have been effectively implemented on PCs and high-end SGI workstations using the OpenGL graphics library. A 2D example computed in real-time is shown in the cover plate. Our techniques improve upon the state of the art in following ways:

- **Generality**: We make no assumption with respect to input primitives. We only need to mesh the distance function of a site over a grid of point samples.
- **Efficiency**: We show that our approach is quite fast. Its speed arises from using coarse polygonal approximations of the distance functions while still maintaining a specified error bound, using polygon rasterization hardware to reconstruct the distance values, and using the Z-buffer depth comparison to perform distance comparisons. We demonstrate the 2D approach on models composed of nearly 100K triangles in a real-time motion planning application through a complex dynamic scene. We derive efficient meshing strategies for polygonal models in 3D, and show the results of a prototype implementation that demonstrates its potential.
- **Tight Bounds on Accuracy**: Although our approach produces a discretized Voronoi diagram, all sources of error are enumerated and techniques are given to produce output within any specified tolerance.
- **Ease of Implementation**: The approach can be easily implemented on current graphics systems. The special cases are limited and the problem reduces to simply meshing a distance function for any new site.

2 RELATED WORK

The concept of Voronoi diagrams has been around for at least four centuries. In his treatment of cosmic fragmentation in *Le Monde de Mr. Descartes, ou Le Traite de la Lumière*, published in 1644, Descartes uses Voronoi-like diagrams to show the disposition of matter in the solar system and its environment. The first presentations of this concept appeared in the work of [Diric50] and [Voron08]. Algorithms for computing Voronoi diagrams have been appearing since the 1970s. See the surveys by [Auren91] and [Okabe92] on various algorithms, applications, and generalizations of Voronoi diagrams.

2.1 Voronoi Diagrams of Points

Among the algorithms known for computing Voronoi diagrams of points in 2D, 3D, and higher dimensions are the divide-and-conquer algorithm proposed by [Shamo75] and Fortune's sweepline algorithm [Fortu86]. Numerically robust algorithms for constructing topologically consistent Voronoi diagrams have been proposed by [Inaga92, Sugih94]. A number of implementations in exact and floating-point arithmetic are also available.

2.2 Generalized Voronoi Diagrams

Algorithms have been proposed for constructing Voronoi diagrams of higher order sites. Two broad approaches based on incremental and divide-and-conquer techniques have been summarized in [Okabe92]. The set of algorithms includes divide-and-conquer algorithms for polygons [Lee82, Held97], an incremental algorithm for polyhedra [Milen93b], and 3D tracing for polyhedral models [Milen93, Sherb95, Culve99]. Curved sites and CSG objects are handled in [Chian92, Dutta93, Hoffm94]. In all these cases, the computation of generalized Voronoi diagrams involves representing and manipulating high-degree algebraic curves and surfaces and their intersections. As a result, no efficient and numerically robust algorithms are known for computing them.

2.3 Approximate Voronoi Diagrams

Many authors compute approximations of generalized Voronoi diagrams based on the Voronoi diagram of a point-sampling of the sites [e.g. Sheeh95]. However, deriving any error bounds on the output of such an approach is difficult, and the overall complexity is not well understood.

[Vleug95] and [Vleug96] have presented an approach that adaptively subdivides space into regular cells and computes the Voronoi diagram up to a given precision. [Laven92] uses an octree representation of objects and performs spatial decomposition to compute the approximation. [Teich97] computes a polygonal approximation of Voronoi diagrams by subdividing the space into tetrahedral cells. All these algorithms take considerable time and memory for large models composed of tens of thousands of triangles, and cannot easily be extended to directly handle dynamic environments.

The idea of using polygon rasterizing hardware and rendering of cones to construct 2D Voronoi diagrams of points is suggested in [Haebe90] and in the OpenGL 1.1 Programming Guide [Woo97].

2.4 Graphics Hardware

Polygon rasterization graphics hardware has been used for a number of geometric computations, such as visualization of constructive solid geometry models [Rossi86, Goldf89] and interactive inspection of solids, including cross-sections and interferences [Rossi92]. Algorithms for real-time motion planning using raster graphics hardware have been proposed by [Lengy90].

3 OVERVIEW

In this section, we present the basic concepts important to our approach. We give a formal definition of generalized Voronoi diagrams and present a simple brute-force strategy for computing a discrete approximation. We then show how we may greatly accelerate this using graphics hardware.

3.1 Generalized Voronoi Diagrams

The set of input sites is denoted as $A_1, A_2, \ldots, A_k$. For any point p in the space, $dist(p, A_i)$ denotes the distance from the point p to the site A_i. The dominance region of A_i over A_j is defined by

$$Dom(A_i, A_j) = \{\ p \mid dist(p, A_i) \leq dist(p, A_j)\ \}$$

For a site A_i, the Voronoi region for A_i is defined by

$$V(A_i) = \cap_{j \neq i} Dom(A_i, A_j)$$

The partition of space into $V(A_1)$, $V(A_2)$, …, $V(A_k)$ is called the *generalized Voronoi diagram*. The (ordinary) Voronoi diagram corresponds to the case when each A_i is an individual point. The boundaries of the regions $V(A_i)$ are called *Voronoi boundaries*. For primitives such as points, lines, polygons, and splines, the Voronoi boundaries are portions of algebraic curves or surfaces.

3.2 Discrete Voronoi Diagrams

Perhaps the simplest way to compute a discrete Voronoi diagram is to uniformly point-sample the space containing Voronoi sites. For each sample point, we find the closest site and its distance. Associating each point in space with its closest sample point induces a uniform subdivision into rectangular cells. For any point, we know the distance to the closest site to within the maximum distance between a point in space and a sample point, i.e. half the diagonal length of a cell.

A simple brute-force approach to find the closest sites is to iterate through all sample points, computing distances to all sites and recording the closest site and distance. The algorithm can be rearranged to iterate through the sites: for each site, compute distances to all sample points and update the current closest site and distance. The second arrangement is amenable to an implementation in graphics hardware.

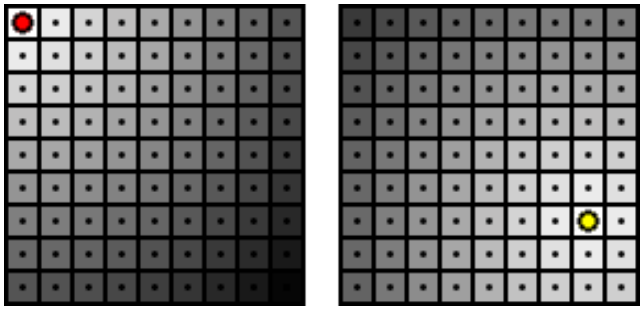


Figure 1: Image of the sampled distance functions for two point sites. Uniform point sampling induces a rectangular cell subdivision of space.

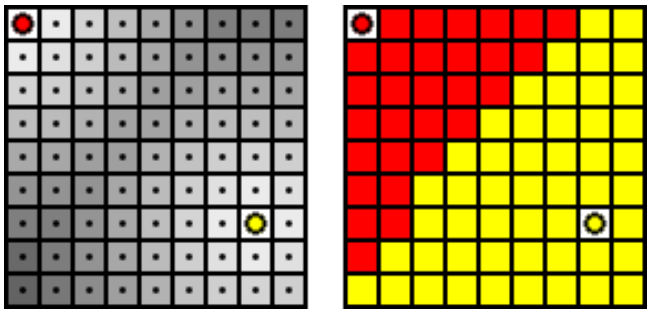


Figure 2: The two distance images are composited through a distance comparison operation. The current closest site and the distance to each site is updated based on the lesser distance value. The resulting Voronoi diagram is composed of a distance image (left) and an closest-site ID image (right).

3.3 Polygon Rasterization Hardware

Our approach makes use of standard Z-buffered raster graphics hardware for rendering polygons. The frame buffer stores the attributes (intensity or shade) of each pixel in the image space; the Z-buffer, or depth buffer, stores the z-coordinate, or depth, of every visible pixel. Given only the vertices of a triangle, the rasterization hardware uses linear interpolation to compute depth values across the triangle's surface. All raster samples covered by a triangle have an interpolated *z*-value.

3.4 Our Approach

A key concept for our approach is that of the *distance function* for a site, which gives, for any point, the distance to that site. The main idea of our approach is to render a polygonal mesh approximation to each site's distance function. Each site is assigned a unique color ID, and the corresponding distance mesh is rendered in that color using a parallel projection. We make use of two components of the graphics hardware: linear interpolation across polygons and the Z-buffer depth comparison operation. When rendering a polygonal distance mesh, the polygon rasterization reconstructs all distances across the mesh. The Z-buffer depth test compares the new depth value to the previously stored value. If the new value is less, the Z-buffer records the new distance, and the color buffer records the site's ID. In this way, each pixel in the frame buffer will have a color corresponding to the site to which it is closest, and the depth-buffer will have the distance to that site. In order to maintain an accurate Voronoi diagram, we bound the error of the mesh to be smaller than the distance between two sample points.

Our approach is inspired by an interesting sidenote in the OpenGL 1.1 Programming Guide [Woo97]. In the Section "Now That You Know" on "Dirichlet Domains", the authors briefly discuss a simple method to construct discretized 2D Voronoi diagrams for points using OpenGL graphics hardware. The authors mention the use of cones for Voronoi diagrams of points in 2D, but warn that the technique "might require thousands of polygons." We show that we can render cones using fewer than 100 triangles for a 1K×1K resolution grid and achieve the same level of accuracy. In addition, we generalize this approach to higher-order sites in both two and three dimensions.

4 THE DISTANCE FUNCTIONS

For both 2D and 3D, our discrete Voronoi diagram computation has been reduced to finding a 3D polygonal mesh approximation to the distance function of a Voronoi site over a planar 2D rectangular grid of point samples. The error in the approximation must be bounded so that by rendering this mesh using graphics hardware, we can efficiently and accurately compute the distances between the site and all of the point samples.

In this section, we describe the distance functions associated with various sites, and provide efficient methods for meshing these functions within a specified error tolerance.

4.1 2D Voronoi Diagrams

Denote the distance from a site A to each pixel location (x,y) by $dist(A,(x,y))$. The *distance function* of A is given by $d(x,y)=dist(A,(x,y))$. Meshing this function corresponds to approximating the graph of $d(x,y)$ with a polygonal model.

The three basic types of 2D sites are points, lines, and polygons. Their corresponding distance functions are shown in the table. In this section, we present algorithms for computing distance meshes for each of them.

2D site	Shape of Distance Function	Figure
Point	Right circular cone	3a
Line segment	"Tent"	3b
Polygon	Cones and tents	5

Table 1: Shape of Distance Functions for 2D Sites

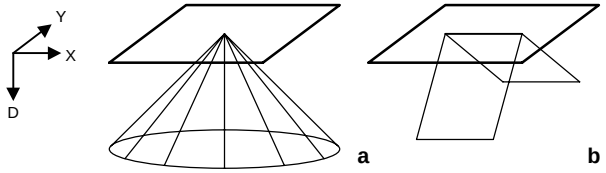


Figure 3: The distance meshes used for a point (left) and a line segment (right). The XY-plane containing the site is shown above each mesh.

4.1.1 Points in 2D

The distance function for a point in the plane is a right circular cone. We approximate cones as a triangle fan proceeding radially outward from the apex (Figures 3a and 4-left). A point's Voronoi region can potentially extend to any portion of the region of interest, and thus the radius at the cone's base must be of size $M\sqrt{2}$ if the scene is contained in an $M\times M$ square.The mesh's radial lines lie on the cone. The maximum error in distance occurs at the cone base between adjacent vertices. Because the cone is right circular, the error in approximating the circular base as viewed from above is equal to the error in distance.

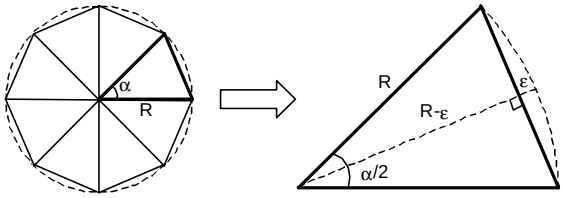


Figure 4: A single triangle of the meshed point distance function cone. α is the angle we wish to maximize, R is the radius of the cone (max dist between site and sample pt), and ε is the max error.

From this formulation (see Figure 4), we compute the maximum angle as:

$$\cos(\alpha/2) = \frac{R-\varepsilon}{R} \;\rightarrow\; \alpha = 2\cos^{-1}\left(\frac{R-\varepsilon}{R}\right)$$

For example, for a maximum distance error of no more than one pixel's width, a cone mesh for a 512×512 grid will require only 60 triangles. A 1024×1024 grid will require 85 triangles.

4.1.2 Line Segments in 2D

The distance function for a line segment is composed of three parts: one for the segment itself and one for each endpoint. The endpoints are treated the same way as points. The distance function for the line segment (excluding the endpoints) is just a "tent" (Figure 3b); its distance mesh is composed of two quadrilaterals. These represent the distance function exactly, so there is no error in the distance mesh representation. The only error for the line segment is in the cone mesh for the endpoint distance functions, as described in the previous section.

4.1.3 Polygons and Per-feature Voronoi Diagrams

It is often useful to consider sites as a collection of features, rather than as a single entity. For example, a line segment would be considered as three features: the two endpoints and the linear edge between them. By rendering the distance meshes for different features in different colors, we obtain a discrete approximation of a *per-feature Voronoi diagram*. Such diagrams are useful in several contexts: for example, the computation of a medial axis of a polygon. A picture of a per-feature Voronoi diagram for a polygon is given in Figure 5-left.

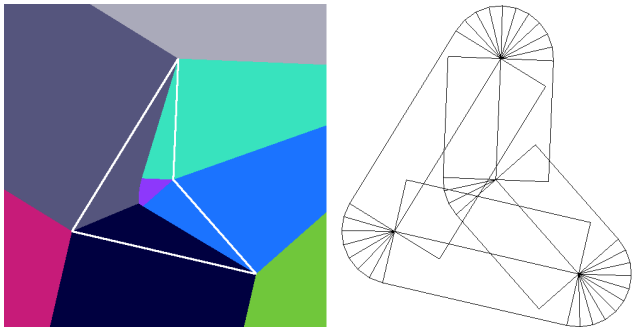


Figure 5: The per-feature Voronoi diagram of a quadrilateral (left). The corresponding distance mesh (right).

Polygons are rendered as a series of linear segments connected at the vertices. Each edge and vertex is a feature. For the vertices, rendering a triangle fan connecting two adjacent edges, rather than a full point distance mesh cone, saves on the total number of triangles computed and ensures that the distance meshes for adjacent features join smoothly. See Figure 5-right for an illustration.

4.2 3D Voronoi Diagrams

Our algorithm computes a 3D discrete Voronoi diagram slice-by-slice. Each slice is parallel to the (x,y)-plane and is computed independently.

Consider the slice $z=z_0$. To construct the intersection of the Voronoi diagram with this slice, consider the distance function for a site A, restricted to the slice. Denote the restricted distance function by $dist(x,y)=dist(A,(x,y,z_0))$. In this section, we describe $dist(x,y)$ for polygon, line segment, and point sites. As in the 2D case, computing the discrete Voronoi diagram is a matter of meshing the distance function $d=dist(x,y)$ for each site and rendering these meshes.

The distance meshes we give for the 3D problem are for a per-feature Voronoi diagram. Thus, a detached triangle site is treated as seven features: a polygon, three line segments, and three points. As in 2D per-feature diagrams, some features have a restricted region of influence.

3D site	Shape of distance function	Figure
Polygon	Plane	6
Line segment	Elliptical cone	7
Point	1 sheet of a hyperboloid of 2 sheets	8

Table 2: Shape of Distance Functions for 3D Sites

4.2.1 Polygons in 3D

The influence of this site in 3D is confined to the region formed by sweeping the polygon orthogonally through space, since points outside this region are considered to be closer to an edge or vertex of the polygon. In the slice, this region is a polygon, and $dist(x,y)$ is linear within this region, as illustrated in Figure 6. The distance to the site is computed at the vertices of the region, and a distance mesh composed of a single polygon is rendered. No meshing error

is incurred. If the polygon intersects the slice, the intersection is computed and the polygon is decomposed into two sub-polygons. Each sub-polygon is treated as above.

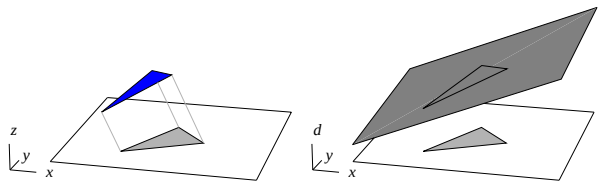


Figure 6: A polygonal site and its region of influence in a slice (left). The corresponding linear distance function (right).

4.2.2 Line Segments in 3D

The graph of the distance function for a line segment site is an elliptical cone (Figure 7). The apex of the cone lies at the intersection of the segment's line with the slice, and the cone's eccentricity is determined by the relative angle of the line and the slice. The 3D region of influence of a line segment lies between two parallel planes through the endpoints, since a point outside these planes is closer to one of the endpoints than to the segment.

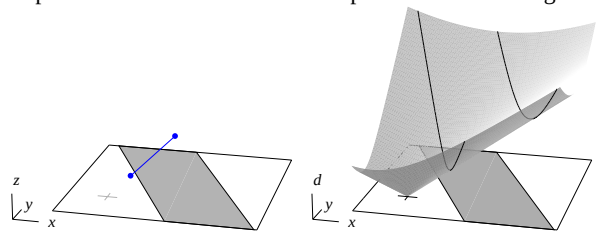


Figure 7: A line-segment site and its region of influence in a slice (left). The corresponding conical distance function (right).

4.2.3 Points in 3D

The distance function for a point site is shown in Figure 8. Its graph is one sheet of a hyperboloid of revolution of two sheets. If the point lies in the slice, the distance function is a cone rather than a hyperboloid. The region of influence for a single point is the entire slice.

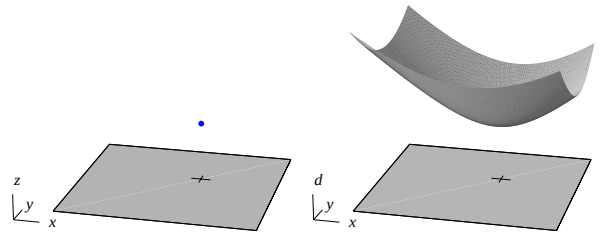


Figure 8: A point site and its region of influence in a slice (left). The corresponding hyperbolic distance function (right).

4.2.4 Meshes for Line Segments and Points in 3D

The construction of bounded-error meshes for the line-segment and point distance functions is detailed in [Hoff99]. The method attempts to minimize the complexity of the mesh by committing the maximum allowable error ε in each mesh cell. The structure of the mesh depends only on the *resolution* of the Voronoi diagram, defined by the ratio of the diameter *M* of the model to the maximum meshing error ε. The mesh structure is precomputed; during the Voronoi diagram construction, the mesh is constructed using table-lookup. Examples of the meshes produced by this method are shown in Figure 9.

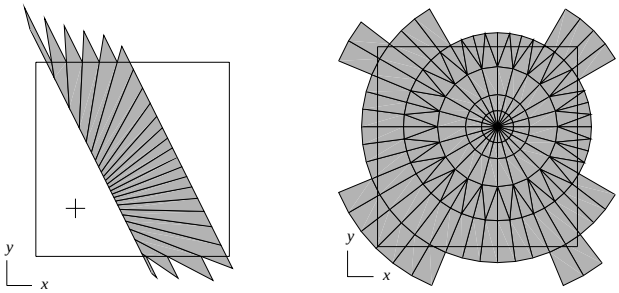


Figure 9: A bounded-error distance mesh for the line-segment site (left) and the point site (right).

4.3 Generalization to Curved Sites

The exact distance function for a curved site can be rather complicated, and for splines or algebraic curves is a high-degree algebraic function. We simplify this by creating a linear tessellation of the curved site, and then meshing the distance function of this approximation. We can use algorithms such as in [Filip87] and [Kumar96] to obtain bounded-error tessellations.

Figure 10 shows the mesh for a Bézier curve. Since the mesh for a linear segment is exact, the distance error for any of the linear segments is just the error in the deviation of the line from the original curve. The endpoints of the curve must be treated as points, just as for the line segment. The distance mesh for the "joints" between linear segments is a portion of the radial mesh of triangles. An overall maximum error bound of ε can be obtained for the entire curve by:

- tessellating the curve into linear segments with maximum error bound of ε;
- rendering the distance mesh for the linear segments; and
- treating the endpoints and joints as points, and rendering each point distance mesh with maximum error bound of ε.

This approach generalizes to 3D surfaces, which can be tesselated into a polygonal mesh. The error is bounded in a similar way.

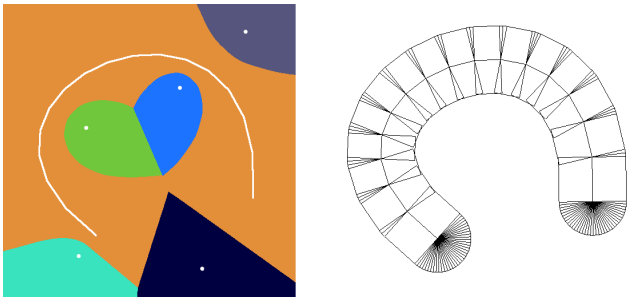


Figure 10: The Voronoi diagram of a Bézier curve and 5 points (left). The distance mesh for the Bézier curve that has been tessellated into 16 segments (right).

4.4 Weighted and Farthest-site Diagrams

In a *weighted* Voronoi diagram, the distance functions are additively or multiplicatively weighted [Okabe92]. Translation of a distance mesh along the distance axis accounts for additive

weights. Linear scaling along the distance axis accounts for multiplicative weights. In 2D, this is equivalent to changing the angle of the cone or tent. Scaling the distance mesh also scales the meshing error.

In a *farthest-site* Voronoi diagram, the farthest site from each point is found. Unlike in the nearest-site diagram, the distance function monotonically decreases as we move away from the site. We obtain the proper distance relationships by negating the distance functions. In practice, however, we need only reverse the depth-test (less-than to greater-than) and change the depth initialization from ∞ to 0.

Figure 11: Standard nearest-site Voronoi regions (left). Farthest regions for the same sites (middle). Weighted regions (right). Weights: line, 2; dark point, 1; light point, 0.5.

5 BOUNDARIES AND NEIGHBORS

A continuous Voronoi diagram representation usually specifies the *Voronoi boundaries* that separate the set of Voronoi regions. In our discrete representation, we must search for the boundaries using approaches similar to iso-surface extraction and root-finding techniques [Bloom97]. However, instead of trying to bracket zero-crossings between sample points where iso-surface functions evaluate to values of opposite sign, we simply find the boundaries in the space between pixel samples of different color. Using the same approaches, we can either point-sample the boundary or compute an approximate mesh representation. In order to increase the precision, we must either use a higher overall resolution or adaptively refine.

One approach is to examine each pair of adjacent cells in 2D or 3D. If the colors are different, the location between the samples is marked as a point on the Voronoi boundary. The operation is very simple and can be accelerated through image operations in graphics hardware.

Another approach is based on a *continuation* method that starts at a point known to be on the boundary and walks along the boundary until all boundary points have been found [Bloom97]. Since we only compare locations near known boundaries, it is output sensitive. The correctness of the continuation method depends on whether the Voronoi boundaries are connected. The boundaries of a generalized Voronoi diagram of a collection of convex sites are always connected, so the method is correct for inputs consisting of point, line-segment, or convex polygonal sites. The method may fail in the presence of curves, curved surfaces, or concave sites where the generalized Voronoi diagram may have isolated components.

In this approach, at least one boundary point must be known as a "seed" value. Assuming convex sites, some Voronoi boundary passes through the edge of the bounded region in which we are computing the diagram, so the method begins by examining every window border pixel. When all Voronoi boundaries are connected only one seed point is needed since all others can be reached from that first point. Starting from a seed point, we recursively check all neighbors that are a different color from the current pixel's. All visited pixels are marked and avoided in the recursion.

This algorithm also finds the *Voronoi neighbors*–pairs of sites that share a Voronoi boundary. This concept is useful in a wide variety of applications, including computing the dual of the ordinary Voronoi diagram–the Delaunay triangulation. The boundary finding algorithms find pairs of adjacent pixels with different colors. The sites corresponding to those two colors are reported to be Voronoi neighbors. Connecting Voronoi neighbors with line segments constructs the Delaunay triangulation.

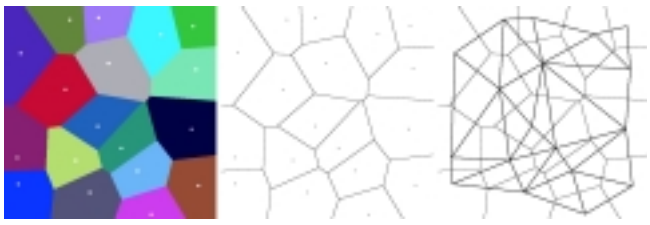


Figure 12: Voronoi diagram of set of 2D points (Left). Boundaries found with continuation-based approach (middle). Delaunay triangulation by connecting neighboring sites (right).

6 SOURCES OF ERROR

In this section we analyze all sources of error in our approach, and discuss how to reduce this error. We consider two broad categories: error in distance approximation and combinatorial error.

6.1 Distance Error

Distance error is the error in the distance computed from a pixel to a site. There are three sources of distance error:

- *Meshing error*, from approximating the true distance function by the distance mesh. We discussed how to bound this error in Section 4.
- *Tessellation error*, from tessellating a curved site into a number of linear sites. The tessellation algorithms presented in [Filip87, Kumar96] give tight bounds. Tessellation error is reduced by using a finer approximation to the site.
- *Hardware precision error*, from the use of fixed-precision arithmetic (integer or floating-point) during rasterization. Hardware precision error cannot be removed without resorting to multiple-precision arithmetic, but hardware error is usually negligible compared to meshing error.

These errors are additive–i.e. the error from one source is not magnified by the other sources. The total distance error is at most the sum of the errors from these three sources.

6.2 Combinatorial Error

Combinatorial error refers to qualitative error as opposed to quantitative. For example, a pixel is assigned the wrong color, or the algorithm reports an incorrect pair of Voronoi neighbors. There are three sources that contribute to combinatorial error:

- *Distance error*, as described in the previous section. With significant distance error, depth comparison at a pixel may make a farther site appear closer, causing the pixel to be colored incorrectly.

- *Resolution error*, a result of discrete sampling. If this sampling is too coarse, we may miss some Voronoi regions or find spurious neighbors. Handling resolution error is described below.
- *Z-buffer precision error*, the limitations of the number of bits of precision provided by the Z-buffer. Current graphics systems have 24 bits or 32 bits of precision for each pixel in the Z-buffer, which is more than the 23 bits provided in standard floating-point. If the distances between two pixels cannot be determined within that precision, the Z-buffer cannot accurately choose the correct color. This effect is small when compared to the other two, but can be significant at very high resolutions with very little distance error. A higher-precision Z-buffer can be simulated in software at a significant loss in efficiency.

Adaptive resolution allows us to "zoom in" on a region of interest, reducing potential resolution error. This involves identifying a window of interest and applying the appropriate linear transformation for zooming into that region. Figure 13 shows an example. Note that when zooming in, sites outside of the viewing region can still have Voronoi regions inside the region. Thus, the "maximum distance to a site" must be adjusted appropriately when computing the distance error bounds.

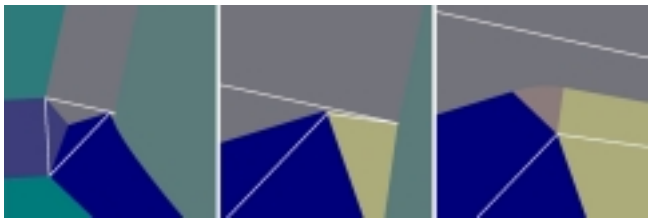


Figure 13: Adaptive resolution allows us to zoom in on features that could otherwise be missed.

Resolution error can cause a number of combinatorial problems, such as missing the entire Voronoi region of a site. One such example is shown in Figure 14 (left two images). When no cell has the color of a particular site, we can separately render the site itself, computing the pixels covering that site. By zooming around those pixels, we will find pixels in the Voronoi region of that site. The same technique can be applied to cells in 3D. Another problem arising from resolution error is incorrectly finding Voronoi neighbors (shown in Figure 14 – right two images). This problem (when due solely to resolution error) can be alleviated by adaptively zooming in on all boundary pixels.

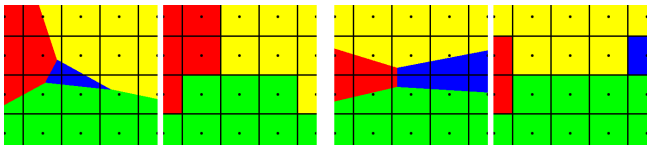


Figure 14: Problems caused by resolution error. An entire region in the center will be missed since it does not hit any pixel centers (left two images). The left and right regions, which should meet, become disconnected after rasterization (right two images).

6.3 Error Bounds

Distance error occasionally causes a pixel to be colored incorrectly. However, in a certain sense, the pixel is "almost" the right color. Assume that there is no Z-buffer precision error, and that we can bound the maximum distance error by ε, as described earlier. For a pixel P colored with the ID of site A and with a computed depth buffer value of D, we know that:

$$D - \varepsilon \le dist(P,A) \le D + \varepsilon$$

Furthermore, we know that for any other site B,

$$D - \varepsilon \le dist(P,B)$$

From this information, we easily determine that

$$dist(P,A) \le dist(P,B) + 2\varepsilon$$

where $dist(X,Y)$ means the distance from the center of pixel X to site Y. That is, if a pixel is colored with the ID of A, then site A is no more than 2ε farther from the pixel center than any other site. The same bound holds in 3D.

7 APPLICATIONS

There are many applications that benefit from fast computation of a discrete Voronoi diagram, an approximation to the distance function, or both. We describe three that we have implemented.

7.1 Motion Planning

Motion planning is a fundamental problem in robotics and computational geometry, with applications to the animation of digital actors, maintainability studies in virtual prototyping, and robot-assisted medical surgery. The classic Piano Mover's problem involves finding a collision-free path for a robot moving from one location (and orientation) to another in an environment filled with obstacles. Numerous approaches to this problem have been proposed, some of which are based on generalized Voronoi diagrams [Latom91]. The underlying idea is to treat the obstacles as sites. The Voronoi boundaries then provide paths of maximal clearance between the obstacles. Due to the practical complexity of computing generalized Voronoi diagrams, the applications of such planners have been limited to environments composed of a few simple obstacles.

Our discrete Voronoi computation algorithm can be applied to motion planning in both static and dynamic environments. The Voronoi algorithm computes the approximate distance to the nearest obstacle. The basic approach we implemented is based on the potential field method, which repels a robot away from the obstacles and towards the goal using a carefully designed artificial potential function. Other Voronoi diagram or distance-based approaches are also possible. The details of our motion planning algorithm are provided in [Hoff99].

We demonstrate our planner's effectiveness in a complex environment: the interior of a house, composed of over 100,000 triangles. We use the x- and y-components of the polygons to give the 2D input primitives for our algorithm. The robot has three degrees of freedom: x- and y-translation along the ground and rotation about the z-axis. Color plate 2 and the video show a sequence of piano motions automatically generated by our motion planner in a static environment. Color plate 2 also shows an image of the distance function for the house. We also apply our planner to environments with moving obstacles. Our video demonstrates the movement of a music stand through a house filled with moving furniture. The entire potential field and the motion planning sequence are computed in real time.

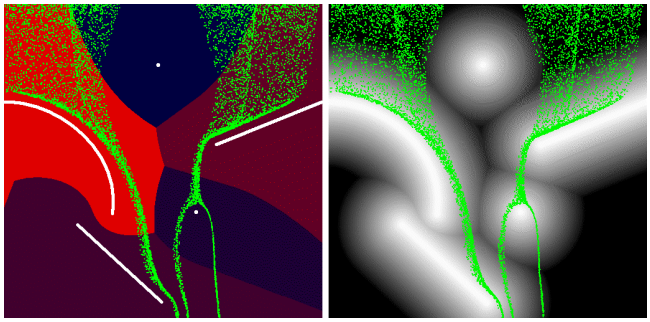


Figure 15: Motion planning of falling particles. Sites are avoided by using the Voronoi diagram's distance buffer (right) to create a potential field. This same principle is used in the rigid-body planner.

7.2 Selection in Complex User Interfaces

Complex 2D user interfaces sometimes require quick determination of the object nearest to the cursor. The Voronoi diagram of the interface can be used as a nearest-object lookup table indexed by sample points. Given the cursor position, it is simple to find the nearest sample point, and thus the nearest object. In some interfaces it may be desirable to know the distance to the selected object as well. We used this technique in our 2D implementation to allow the user to interactively move sites with the mouse.

7.3 Mosaics

We can use our approach for generating Voronoi diagrams to create an interesting artistic effect called mosaicing. A mosaic is a tiled image, where each tile has a single color. The Voronoi diagram of a point set can be used as a tiling [Haebe90]. Each Voronoi tile is colored with a color taken locally from the image. In our implementation, each tile is colored by the image pixel closest to the point site (see color plate 1). Our algorithm can perform this operation very quickly, allowing dynamic mosaics in which the mosaic tiling, the source image, or both may change in real time.

By randomly distributing point sites across an image, we obtain an effect similar to many mosaic filter effects seen in image editing programs. By clustering point sites around areas of higher detail, we obtain a classic tiling seen in many real-life mosaics where smaller tiles are used in areas of greater detail.

8 IMPLEMENTATION

For the 2D case, we implemented a complete interactive system incorporating all of the features and applications described here. Example output is shown throughout the paper. The video demonstrates interactive computation of more complex diagrams. In 3D, we show results from a prototype system that uses a simpler distance meshing strategy (see color plate 3 and the video for example output).

We implemented the 2D and 3D systems in C++ using the OpenGL graphics library and the GLUT toolkit. Any graphics API specification that uses a standard Z-buffered interpolation-based raster graphics system is sufficient to support the Voronoi diagram computation. Motion planning and the basic operations of boundary and neighbor finding require reading back of the color and depth buffers. Our system runs, without source modification, on both an MS-Windows-based PC and a high-end SGI Onyx2 with InfiniteReality Graphics. Surprisingly, the performance on a 400 Mhz Intel Pentium II PC with an Intergraph Intense 3D Pro 3410-T graphics accelerator was comparable to the SGI performance. In fact, in boundary finding, neighbor finding, and particle motion planning applications, the performance exceeded the high-end SGI. This was mainly due to intense buffer readback requirements. Each distance mesh must cover every pixel, so performance is bounded by the graphics hardware's pixel fill-rate. For large numbers of input sites, therefore, the SGI outperforms the PC.

When the distance-error tolerance is relaxed, the amount of geometry rendered for each site can be reduced, slightly improving performance. However, the biggest gains are achieved by reducing the number of pixels filled. In many practical cases, we can increase the performance significantly by bounding the site distance functions to a maximum distance. This allows reduction of the size of the distance meshes drawn so that only a portion of the screen is covered for each site. We exploit this observation to obtain interactive rates in the 1,000-point example shown in color plate 1, in the 10,000-point example shown in the video, and in the general case for the computation of the potential field used in the motion-planner. For closed higher-order primitives, such as polygons, we can further increase performance by restricting the distance function to only the inside or outside regions. This is useful in computing potential fields and medial axes.

9 CONCLUSIONS AND FUTURE WORK

We have presented a method for rapid computation of generalized discrete Voronoi diagrams in two and three dimensions using graphics hardware. We have presented techniques for creating a mesh of the distance function for each site with bounded error, and described how this distance mesh allows us to compute the Voronoi diagram rapidly. We have analyzed various sources of error, as well as how to bound or reduce those errors. Finally, we have demonstrated a few applications using our approach.

In the future, we would like to extend this work in the following ways: generalizations of distance functions and site geometry, further applications, other distance meshing strategies, and more acceleration techniques for the 3D Voronoi volume computation.

ACKNOWLEDGEMENTS

Supported in part by ARO Contract DAAH04-96-1-0257 and DAAG55-98-1-0322, NSF Career Award CCR-9625217, NSF grants EIA-9806027 and DMI-9900157, NIH Research Resource Award 2P41RR02170-13, ONR Young Investigator Award and Intel. We would also like to thank Sarah Hoff for extensive help with editing and color plates, Chris Weigle for suggesting mosaicing using 2D point Voronoi diagrams, the UNC-walkthrough group for the house model, and the reviewers for their helpful comments.

REFERENCES

[Auren91] F. Aurenhammer. *Voronoi Diagrams: A Survey of a Fundamental Geometric Data Structure*. ACM Computing Surveys, 23:345–405, 1991.

[Bloom97] J. Bloomenthal, C. Bajaj, J. Blinn, M-P. Cani-Gascuel, A. Rockwood, B. Wyvill, and G. Wyvill. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers, Inc. San Francisco, CA. 1997.

[Chian92] C–S. Chiang. *The Euclidean Distance Transform*. Ph. D. thesis, Dept. Comp. Sci., Purdue Univ., West Lafayette, IN, August 1992. Report CSD-TR 92–050.

[Culve99] T. Culver, J. Keyser, and D. Manocha. *Accurate Computation of the Medial Axis of a Polyhedron*. Proc. of the Fifth Symp. on Solid Modeling and Applications. 1999.

[Dutta93] D. Dutta and C.M. Hoffmann. *On the Skeleton of Simple CSG Objects*. Journal of Mechanical Design, ASME Transactions, 115(1):87–94, 1993.

[Diric50] G.L. Dirichlet. *Uber die Reduktion der Positiven Quadratischen Formen mit Drei Unbestimmten Ganzen Zahlen*. J. Reine Angew. Math., 40:209–27, 1850.

[Filip87] D. Filip and R. Goldman. *Conversion from Bézier-rectangles to Bézier-triangles*. CAD, 19:25–27, 1987.

[Fortu86] S. Fortune. *A Sweepline Algorithm for Voronoi Diagrams*. In Proc. 2nd Annual ACM Symp. on Comp. Geom., pages 313–322, 1986.

[Goldf89] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs. *Near Real-time CSG Rendering Using Tree Normalization and Geometric Pruning*. IEEE Computer Graphics and Applications, 9(3):20–28, May 1989.

[Haebe90] P. Haeberli. *Paint by Numbers: Abstract Image Representation*. Computer Graphics (SIGGRAPH '90 Proc). vol. 25. pgs 207-214.

[Held97] M. Held. *Voronoi Diagrams and Offset Curves of Curvilinear Polygons*. Computer-Aided Design, 1997.

[Hoff99] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. *Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware*. Technical Report TR99-011, Dept. of Comp. Sci., University of North Carolina at Chapel Hill, 1999.

[Hoffm94] C.M. Hoffmann. *How to Construct the Skeleton of CSG Objects*. In A. Bowyer and J. Davenport, editors. Proc. of the Fourth IMA Conference, The Mathematics of Surfaces, University of Bath, UK, Sept. 1990. Oxford University Press, New York, 1994.

[Inaga92] H. Inagaki, K. Sugihara, and N. Sugie. *Numerically Robust Incremental Algorithm for Constructing Three-dimensional Voronoi Diagrams*. In Proc. 4th Canad. Conf. Comp. Geom., pgs 334-339, 1992.

[Kumar96] S. Kumar, D. Manocha, and A. Lastra. *Interactive Display of Large NURBS Models*. IEEE Trans. on Vis. and Computer Graphics. vol 2, no 4, pgs 323-336, Dec 1996.

[Latom91] J.C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[Laven92] D. Lavender, A. Bowyer, J. Davenport, A. Wallis, and J. Woodwark. *Voronoi Diagrams of Set-theoretic Solid Models*. IEEE Computer Graphics and Applications, 12(5):69–77, Sept 1992.

[Lee82] D.T. Lee. *Medial Axis Transformation of a Planar Shape*. IEEE Trans. Pattern Anal. Mach. Intell., PAMI–4:363–369, 1982.

[Lengy90] J. Lengyel, M. Reichert, B.R. Donald, and D.P. Greenberg. *Real-time Robot Motion Planning Using Rasterizing Computer Graphics Hardware*. Computer Graphics (SIGGRAPH '90 Proc.), vol. 24, pgs 327–335, Aug 1990.

[Milen93] V. Milenkovic. *Robust Construction of the Voronoi Diagram of a Polyhedron*. In Proc. 5th Canadian. Conference on Comp. Geom., pgs 473-478, 1993.

[Milen93b] V. Milenkovic. *Robust Polygon Modeling*. Computer Aided Design, 25(9), 1993. (special issue on Uncertainties in Geometric Design).

[Okabe92] A. Okabe, B. Boots, and K. Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, Chichester, UK, 1992.

[Rossi92] J. Rossignac, A. Megahed, and B. Schneider. *Interactive Inspection of Solids: Cross-sections and Interferences*. Computer Graphics (SIGGRAPH '92 Proc.), vol. 26, pgs 353–360, July 1992.

[Rossi86] J.R. Rossignac and A.A.G. Requicha. *Depth-buffering Display Techniques for Constructive Solid Geometry*. IEEE Computer Graphics and Applications, 6(9):29–39, 1986.

[Sheeh95] D.J. Sheehy, C.G. Armstrong, and D.J. Robinson. *Computing the Medial Surface of a Solid from a Domain Delaunay Triangulation*. In Proc. ACM/IEEE Symp. on Solid Modeling and Applications, May 1995.

[Shamo75] M.I. Shamos and D.Hoey. *Closest-point Problems*. In Proc. 16th Annual IEEE Symposium on Foundations of Comp. Sci., pages 151–162, 1975.

[Sugih94] K. Sugihara and M. Iri. *A Robust Topology-oriented Incremental Algorithm for Voronoi Diagrams*. International Journal of Comp. Geom. Appl., 4:179–228, 1994.

[Sherb95] E.C. Sherbrooke, N.M. Patrikalakis, and E. Brisson. *Computation of the Medial Axis Transform of 3D Polyhedra*. In Solid Modeling, pages 187–199. ACM, 1995.

[Teich97] M. Teichmann and S. Teller. *Polygonal Approximation of Voronoi Diagrams of a Set of Triangles in Three Dimensions*. Tech Rep 766, Lab of Comp. Sci., MIT, 1997.

[Vleug95] J. Vleugels and M. Overmars. *Approximating Generalized Voronoi Diagrams in Any Dimension*. Technical Report UU-CS-1995-14, Dept. of Comp. Sci., Utrecht University, 1995.

[Vleug96] J. Vleugels, V. Ferrucci, M. Overmars, and A. Rao. *Hunting Voronoi Vertices*. Comp. Geom. Theory Appl., 6:329–354, 1996.

[Voron08] G.M. Voronoi. *Nouvelles Applications des Paramètres Continus à la Théorie des Formes Quadratiques. Deuxième Mémoire: Recherches sur les Parallélloèdres Primitifs*. J. Reine Angew. Math., 134:198–287, 1908.

[Woo97] M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide*, Second Edition. Addison Wesley, 1997.

Plate 1: We can use the Voronoi diagram to create real-time mosaic effects. Left: Voronoi diagram computed for 1000 2D point sites at a resolution of 512x512. Center: Same diagram using the point locations to index into a 256x256 face texture to obtain the region colors. Right: The final effect at 512x512 with points removed. In the video, we show this same effect in real-time with 10,000 points.

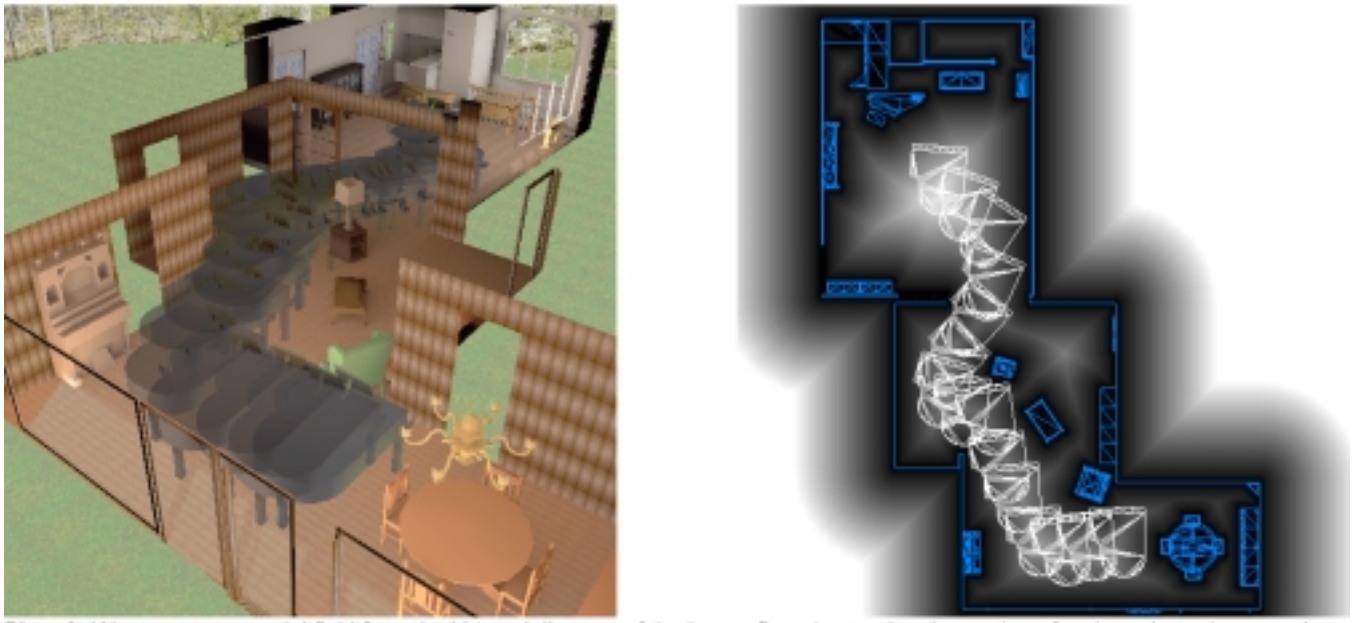


Plate 2: We create a potential field from the Voronoi diagram of the house floorplan to plan the motion of a piano through a complex static scene in real-time. Left: The piano motion sequence. Right: Motion sequence overlaying the floorplan distance function. The video demonstrates navigation through a dynamic scene in real-time.

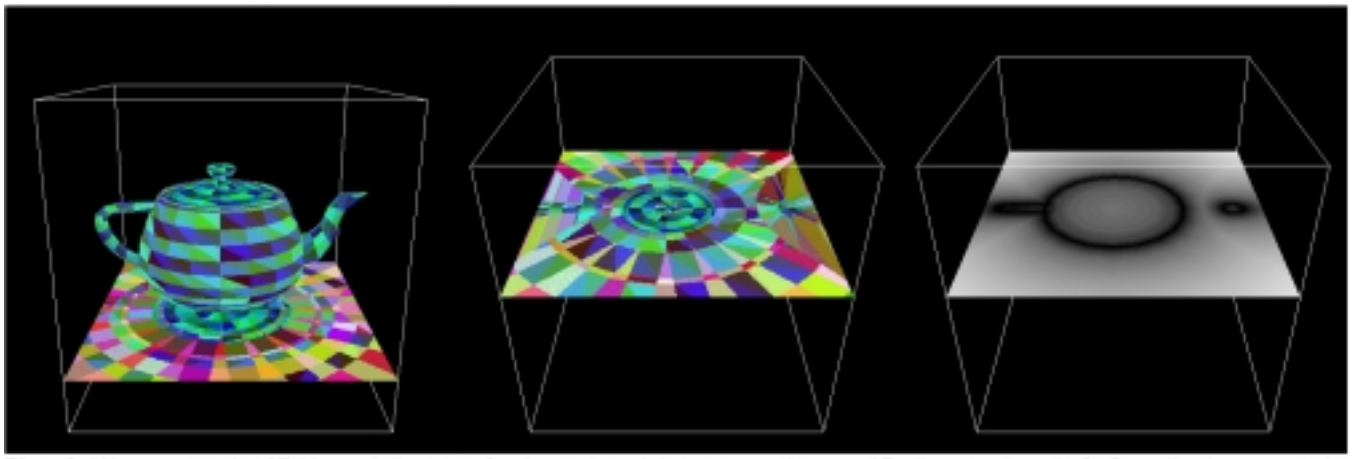


Plate 3: We compute the 3D Voronoi diagram of polygonal models by computing one 2D slice at a time. Left: One slice is computed just below the teapot. Center: Complete slice halfway through. Right: Distance image. The colors indicate the Voronoi regions for face edge, and vertex sites. Only the face site colors are shown on the model.

Fast 3D Geometric Proximity Queries between Rigid and Deformable Models Using Graphics Hardware Acceleration

Kenneth E.Hoff III, Andrew Zaferakis, Ming Lin, Dinesh Manocha

The University of North Carolina at Chapel Hill
Department of Computer Science
{hoff,andrewz,lin,dm}@cs.unc.edu

Abstract

We present an approach for computing generalized proximity information between arbitrary polygonal models using graphics hardware acceleration. Our algorithm combines object-space localization, multi-pass rendering techniques, and accelerated distance field computation to perform complex proximity queries at interactive rates. It is applicable to any closed, possibly non-convex, polygonal object and requires no precomputation, making it suitable for both rigid and dynamically deformable geometry of relatively high complexity. The proximity queries include, not only collision detection, but also the computation of intersections, minimum separation distance, closest points, penetration depth and direction, and contact points and normals. The load is balanced between CPU and graphics subsystems through a hybrid geometry and image-based approach. Geometric object-space techniques coarsely localize potential interactions between two objects, and image-space techniques accelerated with graphics hardware provide the low-level proximity information. We have implemented our system using the OpenGL graphics library and have tested it on various hardware configurations with a wide range of object complexities and contact scenarios. In all cases, interactive frame rates are achieved. In addition, our algorithm's performance is heavily based on the graphics hardware computational power growth curve which has exceeded the expectations of Moore's Law for general CPU power growth.

1. Introduction

Many applications of computer graphics or computer simulated environments require spatial or proximity relationships between objects. In particular, dynamic simulation, haptic rendering, surgical simulation, robot motion planning, virtual prototyping, and computer games often need to perform different proximity queries at interactive rates. The set of queries include collision detection, intersection, closest point computation, minimum separation distance, penetration depth, and contact points and normals. Algorithms to perform different queries have been well studied in computer graphics, virtual environments, robotics and computational geometry. Most of the current approaches involve considerable pre-processing and therefore are not fast enough for deformable models. Furthermore, no good algorithms are known for penetration depth computation between general, non-convex models.

We present a novel approach to perform all the proximity queries between rigid and deformable models using graphics hardware acceleration. Our algorithm localizes potential interactions using object-space techniques, point-samples the region, and then uses polygon rasterization hardware to compute object intersections, closest points, and the distance field and its gradients.

The main features of our approach include a unified framework for all proximity queries, applicability to non-convex polygonal models, computational efficiency allowing interactive queries on current PCs, robustness in terms of not dealing with any special-cases or degeneracies, and portability across various CPU/graphics combinations. A user-specified error threshold for pixel point sampling density and distance approximation governs the accuracy of the overall approach. Some of the novel features of our approach include:

- Improved and efficient construction of distance meshes used to compute 3D Voronoi diagrams accelerated with graphics hardware.
- Site culling algorithms and distance mesh culling for increased performance of Voronoi computation.

- Improved graphics hardware acceleration of computing the intersection between two, possibly non-convex, polygonal objects, over an entire volume.
- Improved algorithm for computing 3D image-space intersections that handles both inter-object and self-collisions.
- Computation of the gradients of the distance field using graphics hardware.

We have implemented our algorithm on various hardware configurations, and demonstrate its performance to compute different queries between rigid and dynamically deforming polygonal objects. Our approach is well suited for computing proximity query information needed for collision responses between dynamic deformable models. The use of graphics hardware allows us to perform different queries at interactive rates on complex deformable models. Moreover, it is relatively simple to implement all these queries in a robust manner. Over the last decade, the graphics processors (GPUs) processing power has been progressing at a rate faster than the CPUs and this will result in handling even more complex scenarios at interactive rates.

2. Related Work

Algorithms for computing collisions, intersections, and minimum separation distances have been extensively researched. Many are restricted to convex objects [Cameron 97, Ehmann01, Gilbert88, Lin91, Mirtich98] or are based on hierarchical bounding-volume or spatial data structures that require considerable precomputation and are best suited for rigid geometry [Hubbard93, Quinlan94, Gottschalk96, Johnson98, Klosowski98]. Some algorithms handle dynamically deforming geometry by assuming that motion is expressed as a closed form function of time [Snyder93] or by using very specialized algorithms [Baraff92]. In our approach, we emphasize the handling of non-convex, dynamically deformable objects with no precomputation or knowledge of object motions. In addition, we obtain computational complexity that grows linearly with geometric complexity for a fixed error tolerance and contact scenario.

As compared to collision detection and separation distance computation, there is relatively little work on penetration depth computation. Penetration depth is typically defined as the minimum translational distance needed to separate two objects. We define it with respect to a point as the minimum translational distance and direction needed to separate a penetrating point from an object's interior. Dobkin et al. have presented an algorithm to compute the intersection depth of convex polytopes, though no practical implementation is known [Dobkin93]. Cameron has presented a practical algorithm that computes an approximate depth for convex polytopes [Cameron97]. No practical algorithms are known for general, non-convex polyhedra.

Our algorithm relies on the computation of discretized distance fields and graphics hardware-accelerated geometric computation. Distance fields - scalar fields that specify minimum distance to a shape for all points in the field - have been used for many applications in graphics, robotics and manufacturing [Frisken00, Fisher01]. Common algorithms for distance field computation are based on level sets [Sethian96] or adaptive techniques [Frisken00]. However, they either require static geometry, extensive preprocessing, or lack tight error bounds. Graphics hardware has been used to accelerate a number of geometric computations, such as visualization of constructive solid geometry models [Goldfeather89], cross-sections and interferences [Rossignac92], and computation of the Minkowki sum [Kaul92]. However, these only compute intersections, not distance-related queries. Algorithms also exist for motion planning using graphics hardware acceleration and distance fields [Kimmel98, Lengyel90, Pisula00]. More recently, an algorithm has been proposed to compute generalized Voronoi diagrams and distance fields using graphics hardware [Hoff99]. Its application to motion planning was presented in [Pisula00]. Also, proximity queries accelerated using graphics hardware was presented in [Hoff01], but it was restricted to 2D and its extension to 3D was not obvious.

2.1 Voronoi and Distance field Computation

In [Hoff99], they present an algorithm for computing approximate 2D and 3D generalized Voronoi diagrams for polygonal objects with a variety of distance metrics. The representation is in the form of a discretized regular grid of sample points (images) across a 2D slice. A 3D Voronoi diagram is composed of a sequence of these slices across the volume to form a regular 3D grid. At each grid point, the ID of the nearest site and its associated distance is stored. They accelerate a brute-force algorithm using graphics hardware.

Instead of relying on a distance evaluation between a point and a Voronoi site, a polygonal distance mesh is constructed so that when rendered it computes the correct distance value as the Z-coordinate. If these distance meshes are rendered for each site with Z-buffer visibility enabled, the correct comparisons and updates will also be performed. This reduces the problem to finding a polygonal mesh approximation of a 2D slice of the distance function. In 3D, the distance mesh must approximate a 2D slice of the 3D domain.

Their 3D implementation simply used a coarse regular grid with direct distance evaluations at each grid point. This often required over-meshing, inefficient direct distance evaluations at grid points, and did not take advantage of the inherent symmetry in the functions being approximated. In addition, this approximation did not provide a tight bound on the approximation and the computation times were on the order of minutes to hours for high resolution Voronoi diagrams of complex models.

We extend the 3D distance mesh ideas and formulate a very fast and efficient bounded error approximation without requiring any lookup tables or complex data structures. In addition, we present methods for greatly accelerating the distance evaluations through culling techniques.

2.2 2D Proximity Queries using Graphics HW

In [Hoff01], they presented an approach using the graphics hardware based Voronoi computation for performing more general proximity queries, such as those needed in computing collision responses in a dynamics simulation. This paper focused on the interactions between 2D, possibly non-convex, polygonal objects only, but illustrated the potential for having a unified framework for a wide range of proximity queries. Many of the queries supported are particularly difficult for object-space algorithms, such as computing intersections, penetrating points, and penetration depths and directions. They used image-space techniques for performing these queries that were accelerated using graphics hardware. The core operations were based on queries into the Voronoi diagram. They presented a pipeline that allowed load balancing between CPU and graphics subsystems by first incorporating an object-space geometric localization phase to restrict the area over which the image-space phase must be performed.

Through improvements in the Voronoi diagram computation, we have extended this work into 3D. Many additional optimizations were necessary to make this run well in practice, including: faster and efficient distance meshing with bounded error, conservative Voronoi site culling, and making the queries symmetric (query A w.r.t. B is the same as B w.r.t. A). In addition, we constructed a specialized algorithm for computing 3D intersections efficiently. Previously in [Hoff01] for 2D, they relied on pixel overwrite to find intersection points. For 3D, we used a parity based strategy similar to operations used in graphics hardware-accelerated visualization of CSG operations and shadow volumes.

3. Overview of Our Approach

Given a collection of closed 3D polygonal objects, we perform coarse geometric localization to find rectangular regions of space (axis-aligned bounding boxes) that contain either potential intersections or closest feature pairs between objects. We uniformly point-sample these regions and use polygon rasterization hardware to compute object intersections, closest points, and the distance field. The distance field and its gradient vector field provide the distance and direction to the nearest feature for each point in the localized region, which gives the contact normals, minimum separation distances, or penetration depths. Our core algorithm computes the proximity information between two 3D, possibly non-convex, polygonal objects. Higher-order curved surfaces are tessellated into polygons with bounded distance deviation error. In our hybrid approach, there are two top-level operations:

(1) Geometric object-space operations to coarsely localize potential intersection regions or closest features

(2) Image-space operations using graphics hardware to compute the proximity information in the localized regions

Most of our improvements center around Voronoi and distance field computation since it is by far the most costly operation and is the most demanding of the graphics hardware. Load balancing between CPU and graphics subsystems is achieved by varying the coarseness of the object-space localization and by using object-space culling strategies. Tighter localized regions result in fewer pixels and a smaller bound on the maximum distance needed for Voronoi computation, thus reducing the fill and geometry loads on the graphics pipeline. We can also balance the load between these two main stages of the graphics pipeline by shifting the distance error tolerance in the Voronoi computation between fill and geometry: increasing the pixel resolution decreases the distance mesh resolution and vice versa. The main parts of the proximity query pipeline are shown in the following figure:

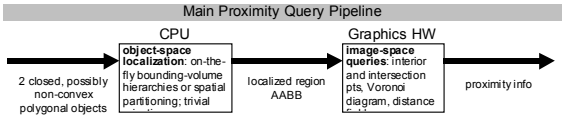


Figure 1: The proximity query pipeline is composed of two main stages: geometric localization and image-space queries. The most complex queries are performed by graphics hardware. Each stage can be varied to balance the load between CPU and graphics subsystems.

4. Object-space Geometric Localization

The image-based queries operate on a uniform 3D grid of sample points in regions of space containing potential interactions. The graphics hardware pixel framebuffer is used as a 2D slice of the grid and the proximity queries become pixel operations, therefore the performance varies dramatically with the pixel resolution. To avoid excessive load, a geometric localization step is used to localize regions of potential interaction or as a trivial rejection stage. This hybrid geometry/image-based approach helps balance the load between the CPU and graphics subsystems, giving us portability between different workstations with varying performance characteristics. More sophisticated geometric techniques, to tightly localize potential intersections or closest feature pairs, dramatically reduce the graphics pipeline overhead, but increases the CPU usage and the complexity of the algorithm. We use coarse fixed-height bounding-volume hierarchies to achieve this balance between speed and complexity, and between CPU and graphics usage.

There are many general and efficient algorithms available for localizing geometry based on bounding-volume hierarchies [Gottschalk96, Hubbard93, Johnson98, Quinlan94]. However, for exact collision detection these algorithms typically perform well only on static geometry where the hierarchy can be precomputed. In order to handle dynamic deformable geometry with no precomputation, we use coarse levels for efficient trivial rejection and obtain reasonable geometric localization. In addition, we perform lazy evaluation of relevant portions of the hierarchies while performing the collision or distance query. A subtree rooted at a particular node is only computed if its children need to be visited during the query traversal. The trees are destroyed

after every proximity query, and no actual tree data structures are required since the child nodes are recursively passed to the query routine. A maximum height of each object tree is used to balance the CPU and graphics load. Similar algorithms can be constructed using spatial partitioning rather than bounding-volume hierarchies. Since the resulting localized region needs to be rectangular (an axis-aligned cube) to allow simple use of the graphics hardware, we use a dynamically constructed AABB-tree. With a fixed number (depth of the tree) of linear passes over the geometry we obtain reasonable localization.

The typical proximity query is between two objects at a time. However, it is possible to perform many simultaneous queries for all objects in an N-body simulation. We could perform the proximity queries for all objects with one image-space query by using a localized region that encloses the entire scene. This may be more efficient in cases when the objects are densely packed with many complex contacts throughout the space containing the objects. For example, in a dense rigid body simulation where many objects are interacting simultaneously (e.g. an asteroid field), a single image-space query over the entire space may be more appropriate (localized region is the world bounding box). In addition, as the computational power of graphics systems continues to overtake the general CPU power, coarser and simpler localization will be favored.

The geometric localization step may often result in multiple disconnected regions on each object. In these cases, the proximity query must be repeated for each localized region. Geometric localization for intersecting and nearest features can be found by using existing bounding-volume or spatial partitioning approaches that act on object boundaries, but finding localized regions around volume intersections requires a specialized algorithm. At each step of refinement, the parent bounding box must fully contain the volume intersection. This can be accomplished by first starting with the intersection of the top-level object bounding boxes. This intersection box will surely contain the intersection volume. Now we can refine this localization by computing the bounding box of the portion of each object that lies in the current box. We then repeat the process on the intersection of these two boxes which is also guaranteed to contain the intersection volume.

5. Image-space Proximity Queries

The proximity queries are simplified using uniform point sampling inside an axis-aligned bounding box (localized region) and accelerated with graphics hardware. This image-space approach helps decouple the objects' geometric complexity from the computational complexity for a specified error tolerance. We point-sample the space containing the geometry within the localized regions with a uniform rectangular 3D grid and perform the queries on this volumetric representation using graphics hardware acceleration. The image-based queries include computing intersections between objects, computing the distance field of an object boundary, and computing the gradient of the distance field. Variations of these basic operations are used to perform the remaining queries. The basic pipeline is shown in Figure 2.

The 3D image-space queries avoid excessive data handling when processing the entire volume of the localized region. Each query must be performed over the uniform 3D grid, one 2D slice at a time. The application query information is sent to the application as it is processed slice-by-slice to avoid processing and storing the entire 3D image. In addition, many of the queries have been made symmetric to avoid a second pass as needed in the previous work.

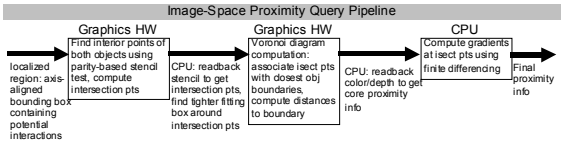


Figure 2: The most computationally intensive tasks are performed by the graphics hardware. These stages are also the most difficult for geometric object-space approaches. We accelerate simple brute-force image-space solutions using graphics hardware to obtain interactive performance on complex models with no precomputation

5.1 Intersections

We compute intersection points on a 2D slice by performing a parity test, as is often used in shadow volumes and CSG rendering, using graphics hardware stencil operations [Crow77, Rossignac92]. In order to find intersections, we must first be able to identify sample points that are inside the object. The set of sample points that are inside both objects form the intersection points between the objects. We then describe another generalized strategy that can handle intersections between multiple objects simultaneously along with the more complex self-intersections.

For any closed object, we can determine if a point is inside the object by shooting a ray from the point in any direction and counting the number of times the object's surface is crossed. If the count is even, the point is outside of the object; if odd, the point is inside. We can simultaneously determine which sample points on a 2D planar slice are interior points by projecting all of the geometry on one side of the plane onto the plane and counting the number of times pixels are overwritten. This computation is performed using the graphics hardware through an odd-even parity test for rendered geometry clipped by the plane and projected onto the plane. Each time a pixel is overwritten the parity bit is flipped. Pixels whose stencil bit is set to 1 represent points on the slice that are inside the object. Initially the stencil buffer is initialized to 0.

5.1.1 Incremental Update and Bucket Sorting

For a single slice, this computation requires rendering all of the geometry on one side of the plane (clipped by the plane). However, this is inefficient for evaluating interior points on many slices swept through our 3D localization box. We improve efficiency by performing a plane sweep and updating the stencil buffer incrementally. For each slice, we

only render the geometry between the current slice and the previous slice.

This incremental update improves the running time dramatically since on average the entire model is only drawn once! As opposed to the single slice approach where the entire model to one side of the slice had to be drawn for each successive slice. We can obtain even greater performance by first sorting the geometric primitives along the Z-axis by their minimum Z-values. A general sort would require O(n log n) time complexity. We obtain expected O(n) complexity by performing a bucket sort by using the slab positions as the buckets. With one pass through the geometry, we can assign each primitive to a bucket by its minimum Z-value. We maintain a list of currently active geometry for each slab. For each subsequent slab we add geometry to the list from the associated bucket. Geometry is removed from the list by checking if the old primitives' max Z-value is less than the current slab NearZ (swept past the primitives). This also dramatically improves performance with very little extra complexity or data. We avoid having to search for geometry that intersects the current slab. In addition, there is no need to add geometry to the buckets that lies outside of the XY min/max box. In practice, very little geometry has to be processed for the interior computation.

In order to find the intersection between two objects, we compute the interior of both objects inside of the localized region one slice at a time. The interior of both objects is encoded in a different bit of the stencil buffer. The set of points with both bits set are intersection points since they are interior to both objects. To actually extract these points, we must read the stencil image and search for the pixels with the appropriate value (a value of 3 from the 1st and 2nd least significant bits being set). These points must then be transformed from pixel-space into object-space.

5.1.2 Complexity and Error Analysis

Our new algorithm for intersection computation for 3D non-convex objects is simpler as compared to the 2D intersection computation algorithm presented in [Hoff01]. The major weakness of finding overwritten pixels between two non-convex polygons, was that they had to be triangulated in order to be rendered. This was the dominant part of the intersection computation since it was worst case O(n log n) rather than O(n). However, the expected running time of most triangulation algorithms is usually close to linear. In 3D, we only require the O(n) complexity where n is the number of primitives. The actual running time varies most dramatically with the ratio of the size of the localized region over the error tolerance, and is largely independent of the geometric complexity. More complex forms of contact do not result in increased running times unless the size of the localized region is increased dramatically or the error tolerance is reduced. These cases are difficult to analyze since they vary dramatically with the object configurations. More sophisticated geometric localization will reduce performance variations.

The complexity of rendering objects grows linearly with respect to the number of primitives for a fixed pixel resolution. Computing intersections geometrically between two polygon boundaries is worst case $O(n^2)$ since all primitives could intersect each other. The complexity of our algorithm is O(n) where n is the number of primitives. The hierarchical geometric localization step is also O(n) since the maximum depth of the tree is held constant. This tree depth balances the load between the CPU and graphics subsystems.

Similarly to the 2D case, the error in the interior and intersection computation is related to the pixel error in scan-conversion. The actual interior regions will never be off by more than half of the length of the diagonal of a pixel's rectangular cell (the error tolerance). The error tolerance has a dramatic effect on the number of pixels that have to be processed. When reduced error tolerances are required, better geometric object-space localization must be employed to reduce the load on the graphics subsystem. Furthermore, we can also balance the loads between geometry and fill stages of the graphics pipeline by trading off error in the pixel resolution and the distance mesh granularity.

Incorrect intersection parity resulting from pixel sample points lying exactly on tangent points to the object surface are avoided through correct minimum-based triangle rasterization as described in [Rossignac92]: either the crossing will be counted twice or not at all.

5.1.3 Multiple Objects and Self-Collisions

We can modify the intersection routine to handle self-collisions and multiple objects with very little modification to the previous algorithm. The modification adds the complexity of having to distinguish front and back faces for polygons in each slab for a parallel projection and has the slight restriction of only handling the intersection of at most 255 simultaneous volumes (limit of 8-bit stencil buffer).

Instead of finding the interior of both objects separately and then finding their common intersection, we can simply finding the intersection directly using the geometry from both objects simultaneously using the classic parity test used in the shadow volume algorithm. Since we want to know if a point is inside two volumes simultaneously, a ray emanating from a query point must have exited at least two more volumes than it has entered.

Instead of simply flipping a bit each time a boundary is crossed (front or back facing), starting with a stencil counter initialized to zero, we increment the counter each time a volume is exited (a back face is rendered) and decrement the counter whenever a volume is entered (front face is rendered). The counter will indicate the number of objects containing the point. We are interested in the intersection points, so the counter must at least be 2. We simply modify our existing approach of rendering slabs to perform this count instead. We must classify all object faces for each slab as front or back facing with respect to a parallel projection. Since all object triangles are handled together, we can handle more than 2 objects and we can also find self-intersections of

a single object. Stencil counts of 2 or greater indicates a point that is in the intersection of at least one pair of objects or an object with itself.

5.2 Distance Field Computation

We use the algorithm presented in [Hoff99] for constructing generalized Voronoi diagrams using graphics hardware for 3D polygonal objects. This approach computes an image-based representation of the Voronoi diagram in both the color and the depth buffers for one 2D slice of the 3D volume at a time. A pixel's color identifies the polygon feature (vertex or edge) that is closest to the slice pixel's sample points; its depth value corresponds to the distance to the nearest feature. The depth buffer is an image-based representation of the distance field of the object boundaries. The distance field is computed by rendering 3D bounded-error polygonal mesh approximations of a 2D planar slice of the distance function where the depth of the rendered mesh at a particular pixel location corresponds to the distance to the nearest geometric feature.

The goal in constructing a distance mesh is to find a piecewise linear approximation across a 2D planar domain of a Voronoi site's 3D scalar distance function. The distance to a site from a point (x,y,z) is defined as $D(x,y,z)$. The function we are interested in approximating is for a 2D planar slice $z=Z_{slice}$. So we wish approximate the 2D scalar function $D(x,y, Z_{slice})$, where Z_{slice} is a constant for any particular slice, such that the approximation D' and actual distance function D never differ by more than the user-specified distance error. In addition, the domain across the slice is restricted to a 2D window and the range of the function is restricted to $z \in [0,MaxDist]$. The shape of the distance mesh for a 3D point is one sheet of a hyperboloid of two sheets; for a line, an elliptical cone; and for a plane, another plane.

In [Hoff99], distance meshes were constructed using lookup tables. We construct error-bounded polygonal mesh approximations of a 2D planar slice of a primitives distance function at run-time with no precomputation at faster rates than the algorithm based on lookup tables. We solve for the mesh stepsizes needed to maintain the desired error threshold while taking advantage of symmetry. We attempt to actual obtain the desired error to make the meshes as coarse as possible for rendering efficiency. In addition, we only construct geometry that lies within the slice window.

For computing distance fields for proximity queries, we obtain higher performance than the generalized Voronoi diagram computation because of the localized regions. In the case of computing distance fields for proximity queries, the localized regions always contain the geometry that is in potential contact or that contains the closest features. The farthest away points on two objects can be is in opposite corners of the localized region box. So the maximum distance we need to construct distance meshes for is half of the diagonal length of the box. Reducing the maximum distance results in the greatest speedups in Voronoi computation since it reduces geometry and fill by reducing the overall extent of the distance meshes, and the smaller bound allows the objects to be easily culled if they are too far from the localized region thus avoiding distance mesh construction completely. In addition, the distance mesh generation routines attempt to minimize the number of primitives drawn by constructing a mesh that is as coarse as possible while staying within the specified error bound (the error bound is tight, this deviation can actually be measured for various places in the mesh approximation) and by only generating primitives that are inside the localized region bounding box. In addition, in many proximity queries we can further reduce the maximum distance needed when we only want intersection or closest points near the boundaries of the object.

5.3 Gradient of the Distance Field

We compute the gradient of the distance field at pixel locations by using central differences in all three principal axis directions. In practice, this simple approach gives reasonable results even with the distance error and lack of C^1 or higher continuity in the polygonal distance mesh approximations used to compute the distance field. Gradients are computed in software for selected points after reading back the distance values. If the entire gradient field is desired, we could accelerate the computation using multi-pass rendering or pixel shading operations.

The most difficult problem in computing the gradient is in handling discontinuities and boundaries in the distance field. There are three types of discontinuities that occur: across Voronoi boundaries, across Voronoi sites, and at the boundaries of the grid. In each case, the support of the finite differencing "kernel" has to cross a discontinuity and gives an incorrect gradient. A more robust method is shown in the fast marching methods in [Sethian96]. He solves for a distance value at an unknown point using an implicit method based on the fact that at least one adjacent distance value must be known and does not cross a discontinuity, and that for the nearest Euclidean distance metric the magnitude of the gradient must be 1 everywhere (except at the discontinuities). We use the same method by just using the one-sided difference in each direction that will result in a gradient whose magnitude is 1 (choose the adjacent value in each direction that has the maximum difference). Adjacent distance values that cross a discontinuity will not be chosen. An alternative, but slightly more complex, strategy is to compute the gradients of the continuous distance meshes directly.

By directly encoding gradients at distance mesh vertices, we can use the linear interpolation of polygon rasterization to compute gradients at all pixels. Since we would be linearly interpolating a gradient, this gives us a higher order interpolation than central differencing of adjacent distance values. This is comparable to the difference between Gouraud and Phong interpolation (the first linearly interpolates color values across a polygon, the second linearly interpolates the surface normal for per-pixel lighting

calculations). In addition, the gradient is much simpler if computed only for a single site at a time during distance mesh construction. We need only provide the direction to the nearest point on the site at each distance mesh vertex. The main difficulty with this approach is in the encoding of the gradient for rapid computation by graphics hardware.

This approach as some difficulties due to limitations of graphics hardware framebuffer precision. There are a number of ways we can interpolate the gradient information. The simplest is to encode the signed normalized components into the 8-bit RGB color values at each vertex (using hardware scale and bias operations for sign). The linear interpolation would give the correct results to 8-bits of precision. This approach introduces quantization error when encoding and additional error during interpolation. Using 3D texture coordinates, high precision encoding and interpolation is obtained. However, the resulting per-pixel texture coordinates are still quantized to low precision RGB values in the framebuffer. The texture-mapping function would simply be the identity mapping. We are interpolating (s,t,r) gradient values and we want those values directly at each pixel. The graphics hardware does not allow higher precision intermediate results for multi-pass operations. However, the texture-mapping method has the advantage of only introducing significant error at the final stage; encoding and interpolation are done at floating point precision. Also, the signs will be correctly handled without any additional scaling or biasing. However, we also have no simple way of performing the identity map. We must use a 1D texture that maps [-1,1] to [0,255], but this can only be applied to one texture component at a time. This would require three passes in order to transform (s,t,r) into RGB values. A less efficient approach would involve the use of a 3D texture map. Current pixel-level programmable graphics hardware may provide a simpler and more efficient way to handle this mapping.

5.4 Other Proximity Queries

We use the basic operations of computing interior points, intersections, the distance field, and the gradient of the distance field to perform the other proximity queries mentioned in section 1.

Penetration Depth and Direction: For a point on object A that is penetrating object B, we define the penetration depth and direction for the point as the distance and direction to the nearest feature on B. This information is provided directly from the distance field and its gradient computed at the penetrating point. Penetrating points are found using the intersection operation. Intersection points are associated with each object based on the Voronoi diagram's color buffer that indicates the closest object to each point. Contact points and Normals are computed in the same way. Approximate contact points result from the objects slightly penetrating each other.

Closest Point: We find the point on object A that is closest to object B by first geometrically localizing potential closest feature regions (one bounding box on each object) using some hierarchical approach. We then compute the distance field of object B and the interior points of A in A's localized region (gives us minimum distance to B for all points in A in A's localized region). We then search these points to find the one with the smallest distance value. This point will be the point on A that is closest to B. This process has to be repeated for B with respect to A. This requires two passes, but the interior points and the distance field needs to only be computed once for each object.

Separation Distance and Direction: We find the minimum separation distance and direction between two objects A and B by first computing the closest point on A to B and vice versa. Ideally, we find the closest point on B to A from the distance value and gradient at the closest point on A to B, but the amplification of errors over the greater distance may cause problems. The distance between these two closest points is the separation distance and the line segment between them gives the separation direction.

6 Performance

We tested the system performance in both rigid and deformable body dynamic simulations on a several different hardware configurations. In the rigid body cases, we measured the performance of the system in computing proximity query information needed for computing a penalty-based collision response. In these cases, only shallow penetration is allowed since the objects bounce off of each other. For the deformable cases, we perform only the proximity queries without collision response to show the worst case of computing proximity information for many deep simultaneous contact scenarios with dynamically deforming geometry. We tried to choose three hardware configurations that would reflect variations in balance between CPU and graphics computational power:

(1) Pentium-4 1.8Ghz with GeForce3 Ti500 graphics: fast CPU and fast graphics

(2) 1 graphics pipe and 1 300Mhz MIPS R12000 processor of an SGI Reality Monster with InfiniteReality2 graphics: slower CPU and fast graphics

(3) PentiumIII-750Mhz laptop with ATI Rage Pro LT: fast CPU and slow graphics.

Because of the ability to balance the load between the CPU and graphics subsystems and between stages of the graphics pipeline, we are able to achieve interactive performance on all configurations. In most cases, we only needed very simple one-level geometric localization (intersection of top-level axis-aligned bounding boxes). Most of the balancing was between stages of the graphics pipeline (much of the geometry stage on older graphics systems was performed on the CPU: before hardware T&L). We also show the effects of the varying the distance threshold on system performance.

For performance evaluation, we implemented a rigid body simulator with collision response and a variety of deformable simulations without collision responses to allow

more complex contact scenarios. The test scenarios vary from simple convex objects composed of around 2 thousand triangles with simple contact regions to non-convex objects with nearly 10,000 triangles with multiple complex overlapping and interlocking contact regions. The average query times are shown in Table 1. It is important to note that the query time is not growing because of the increase in geometric complexity, but rather because our more complex models are in more complex contact configurations.

The performance of our image-space query system depends more on the contact configuration than on the complexity of the objects. The distance error tolerance determines the point sample density across the contact volume. The density and the volume of the localized regions and the contact regions determine the number of pixels that have to be processed. If an insufficient level of geometric localization is used, the number of pixels to process may increase dramatically. The user must decide the appropriate amount of localization to properly balance the CPU/graphics load. In addition, the performance can be varied dramatically by the user-specified distance error tolerance. In Table 2, we show the effects on performance with a varying error tolerance.

Average Total Per-frame Proximity Query Times					
Demo	**#Tris**	**Isect Pts**	**GeForce3**	**IR2**	**Rage Pro**
Cylinders	2000	513	12ms	61ms	45ms
Tori	5000	1412	71	262	257
Heart	8000	317	149	329	434
Rigid	15000	2537	313	1001	966

Table 1: Performance timings for dynamics simulations. The number of triangles, average number of intersection points, and average time to run proximity queries per frame is reported for error tolerance d (see Table2). Timing data was gather from three machines, a Pentium4 1.8GHz desktop with a 64Mb GeForce3, a SGI 300MHz R12000 with InfiniteReality2 graphics, and a Pentium-III 750Mhz laptop with ATI Rage Pro LT graphics.

Effects of Error Tolerance on Performance				
Error	**Isect Pts/Frame**	**GeForce3**	**IR2**	**Rage Pro LT**
***d*/4**	89605	548ms	1701ms	2846ms
***d*/2**	11238	169	578	689
d	1413	71	262	257
2*d*	177	32	189	103
4*d*	22	15	56	40

Table 2: The effect on performance when changing the distance error tolerance *d*. The average number of intersection points per frame is also reported. We used proximity queries on the deformable tori demo. The error determines the number of pixels used in the image-based operations. Systems with low graphics performance are more directly affected by the choice of *d*; however, as the error is increased there is less dependence on graphics performance and the faster laptop CPU overtakes the InfiniteReality2 system.

Although we focused most of our efforts on handling deformable body proximity queries, our system is also applicable to rigid body queries. We use a penalty-based collision response that acts on individual point samples that approximate our object. These point samples arise from our image-space proximity queries. Particles are allowed to penetrate objects in penalty-based collision response computation. When a penetration is detected, a spring based restoring force, whose magnitude is proportional to penetration depth, is then applied to the particle until it has separated from the object. The measure of penetration is notoriously expensive to compute and limits the use of penalty-based techniques to mostly models decomposable into convex primitives. The generality and computational efficiency provided by our proximity query algorithms alleviates this problem.

7 Conclusion and Future Work

We have presented a hybrid geometry- and image-based algorithm for computing geometric proximity queries between two non-convex closed 3D polygonal objects using graphics hardware. This approach has a number of advantages over previous approaches. The unified framework allows us to compute all the queries, including penetration depth and direction and contact normals. Furthermore, it involves no precomputation and handles non-convex objects; as a result, it is also applicable to dynamic or deformable geometric primitives. In practice, we have found the algorithm to be simple to implement (as compared to similarly robust geometric algorithms), quite robust, fast (considering the complexity of the queries), and very flexible. We have developed an interactive system that shows proximity queries computed between 3D dynamic deformable objects to illustrate the effectiveness of our approach.

8 Acknowledgements

This research has been supported in part by ARO Contract DAAG55-98-1-0322, DOE ASCII Grant, NSF Grants NSG-9876914, NSF DMI-9900157 and NSF IIS-982167, ONR Contracts N00014-01-1-0067 and N00014-01-1-0496 and Intel

References

[Baraff92] D. Baraff, *Dynamic Simulation of Non-Penetrating Rigid Bodies*. Ph.D. Thesis, Dep of Comp. Sci., Cornell University, March 1992

[Cameron97] S. Cameron, *Enhancing GJK: Computing Minimum and Penetration Distance between Convex Polyhedra*. International Conference on Robotics and Automation, 3112-3117, 1997

[Crow77] F. Crow, *Shadow Algorithms for Computer Graphics*. SIGGRAPH 77.

[Dobkin93] D. Dobkin, J. Hershberger, D. Kirkpatrick, S. Suri, *Computing the Intersection Depth of Polyhedra. Algorithmica*, 9(6), 518-533, 1993

[Ehmann01] S. Ehmann and M. Lin. *Accurate and Fast Proximity Queries between Polyhedra Using Surface Decomposition*. Eurographics 2001

[Fisher01] S. Fisher and M. Lin. *Fast Penetration Depth Estimation for Elastic Bodies Using Deformed Distance Fields*. Proc. Intl. Conf. on Intelligent Robots and Systems, 2001

[Frisken00] S. Frisken, R. N. Perry, A. P. Rockwood, T. R. Jones, *Adaptively Sampled Distance Fields: A General Representation of Shapes for Computer Graphics*. SIGGRAPH00, 249-254, July 2000

[Gilbert88] E. G. Gilbert, D. W. Johnson, S.S. Keerthi. *A Fast Procedure for Computing the Distance Between Objects in Three-Dimensional Space*. IEEE J. Robotics and Automation, RA(4): 193-203, 1988

[Goldfeather89] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs. *Near Real-time CSG Rendering Using Tree Normalization and Geometric*

Pruning. IEEE Computer Graphics and Applications, 9(3):20-28, May 1989

[Gottschalk96] S. Gottschalk, M. C. Lin, D. Manocha, *OBB-Tree: A Hierarchical Structure for Rapid Interference Detection*. SIGGRAPH 96, 171-180, 1996

[Hoff99] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. *Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware*. SIGGRAPH 99, 277-285, 1999

[Hoff01] K. Hoff, A. Zaferakis, M. Lin, and D. Manocha. *Fast and Simple 2D Geometry Proximity Queries Using Graphics Hardware*. ACM Symposium on Interactive 3D Graphics, 2001

[Hubbard93] P. M. Hubbard, *Interactive Collision Detection*. IEEE Symposium on Research Frontiers in Virtual Reality. 24-31, 1993

[Kaul92] A. Kaul and J. Rossignac, *Solid-interpolating Deformations: Construction and Animation of PIPs*, Computer and Graphics, vol 16, 107-116, 1992

[Kimmel98] R. Kimmel, N. Kiryati, A. Bruckstein, *Multi-Valued Distance Maps for Motion Planning on Surfaces with Moving Obstacles*. IEEE Transactions on Robotics and Automation, vol 14: 427-438, 1998

[Klosowski98] J. Klosowski, M. Held, J. Mitchell, K. Zikan, H. Sowizral. *Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs*. IEEE Trans. Vis. Comp. Graph, 4(1):21-36, 1998

[Johnson98] D. Johnson, E. Cohen, *A Framework for Efficient Minimum Distance Computation*, IEEE Conf. On Robotics and Animation, 3678-3683, 1998

[Lengyel90] J. Lengyel, M. Reichert, B.R. Donald, and D.P. Greenberg. *Real-time Robot Motion Planning Using Rasterizing Computer Graphics Hardware*. Computer Graphics (SIGGRAPH 90 Proc.), vol. 24, pgs 327-335, Aug 1990

[Lin91] M. Lin, J. Canny. *Efficient Algorithms for Incremental Distance Computation*. IEEE Transactions on Robotics and Automation, 1991

[Mirtich96] B. Mirtich, *Impulse-Based Dynamic Simulation of Rigid Body Systems*. Ph.D. Thesis, University of California, Berkeley, Dec 1996

[Mirtich98] B. Mirtich, *V-Clip: Fast and Robust Polyhedral Collision Detection*. ACM Trans. on Graph, 17(3):177-208, 1998

[Pisula00] C. Pisula, K. Hoff, M. Lin, and D. Manocha. *Randomized Path Planning for a Rigid Body Based on Hardware Accelerated Voronoi Sampling*. Proc. of Workshop on Algorithmic Foundations of Robotics, 2000

[Quinlan94] S. Quinlan, *Efficient Distance Computation between Non-Convex Objects*. International Conf. on Robotics and Automation, 3324-3329, 1994

[Rossignac92] J. Rossignac, A. Megahed, and B. Schneider. *Interactive Inspection of Solids: Cross-sections and Interferences*. SIGGRAPH 92, 26, 353-360, July 1992

[Sethian96] J. Sethian, *Level Set Methods*, Cambridge University Press, 1996

[Snyder93] J. Snyder, A. Woodbury, K. Fleischer, B. Currin, A. Barr, *Interval Methods for Multi-Point Collisions Between Time Dependent Curved Surfaces*. ACM Computer Graphics, 321-334, 1993

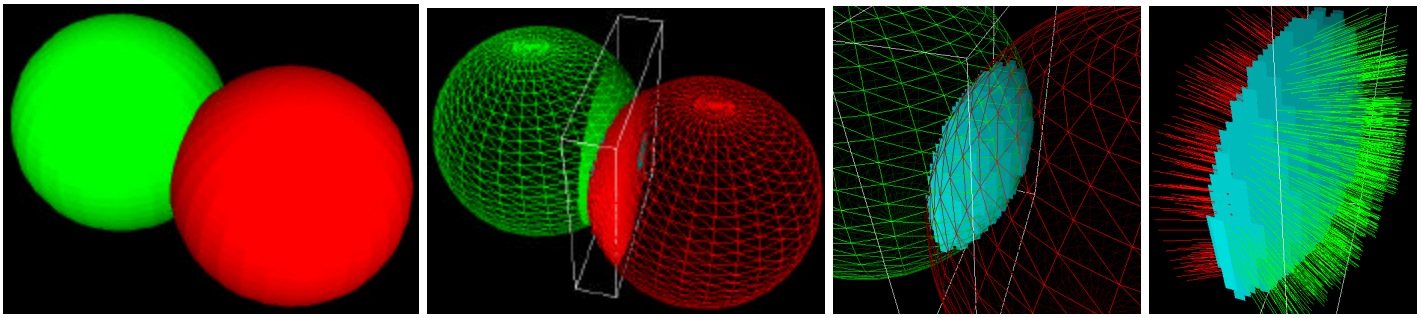


Plate 1 hybrid proximity query pipeline: Given two closed polygonal objects, a coarse object-space geometric localization step is performed to find an axis-aligned bounding box that contains a potential interaction (2). Inside the localized region, the lower-level image-space queries are performed. First the interior of each object is indentified using an incremental stencil parity test for a series of 2D slices across the volume (2). The set of point that are determined to lie in the interior of both objects form the intersection points between the objects (3). Then, the Voronoi diagram is computed inside a tighter region around the intersection points at the same resolution as the intersection resolution. The Voronoi diagram serves two purposes: associates intersection points with their closest object boundaries, and provides the distance field. The distance value at an intersection point gives the penetration depth, and the gradient gives the penetration direction.

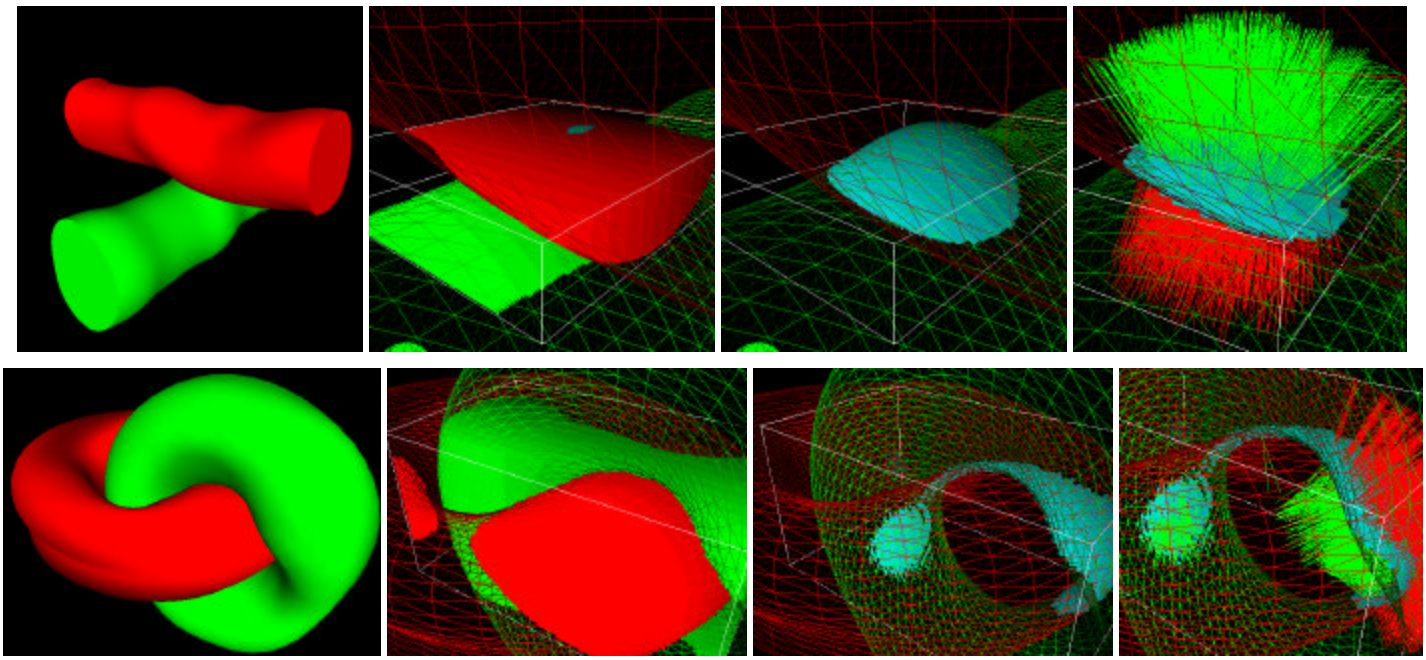


Plate 2 real-time dynamic deformable proximity queries: The same proximity query pipeline can be applied to dynamic deformable models where every vertex is assumed to change for every frame. The complex contacts between non-convex objects can result in disconnected intersection regions. Each cylinder model is composed of 2000 triangles and the average query time is 12ms for an average of 513 intersection points per query. The tori are composed of 5000 triangles and the query time is 71ms for 1412 intersection points. Each simulation performed at interactive rates on a Pentium4 1.8GHz desktop with a 64Mb GeForce3.

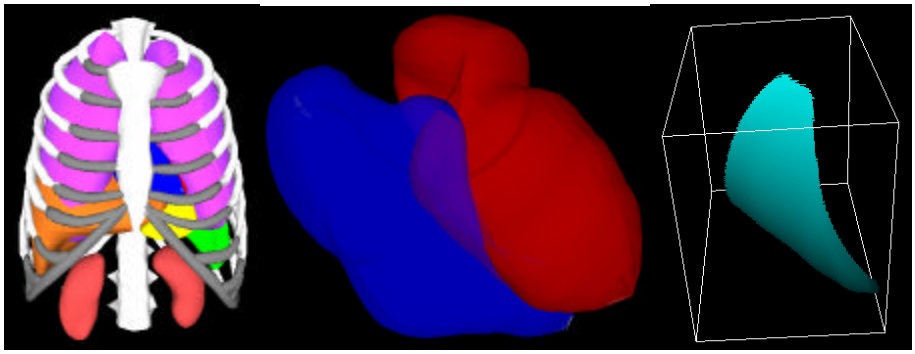


Plate 3 proximity queries on body heartbeat simulation: The proximity queries are used for path verification of the organs during a precomputed breathing simulation. Here we can see that the two ventricles are actually intersecting. The heart is composed of 8000 triangles and the average query time is 149ms for an average of 317 intersection points. This simulation performed at interactive rates on a Pentium4 1.8GHz desktop with a 64Mb GeForce3.

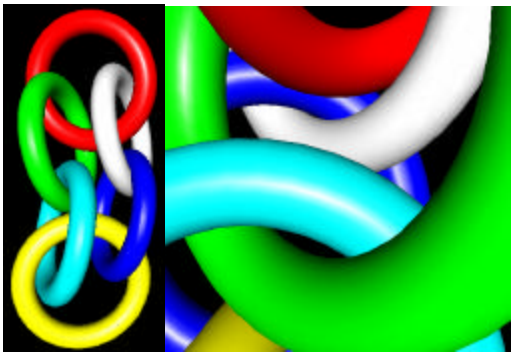


Plate 4 multiple complex contact scenario in an interactive rigid body simulation: Collision responses are computed using a penalty-based method that requires penetration depth computation. Each ring is composed of 2500 triangles, average query time is 313ms for 2537 intersection points.

Fast and Simple 2D Geometric Proximity Queries Using Graphics Hardware

Kenneth E. Hoff III, Andrew Zaferakis, Ming Lin, Dinesh Manocha

Department of Computer Science
University of North Carolina at Chapel Hill
http://www.cs.unc.edu/~geom/PIVOT/

ABSTRACT

We present a new approach for computing generalized proximity information of arbitrary 2D objects using graphics hardware. Using multi-pass rendering techniques and accelerated distance computation, our algorithm performs proximity queries not only for detecting collisions, but also for computing intersections, separation distance, penetration depth, and contact points and normals. Our hybrid geometry and image-based approach balances computation between the CPU and graphics subsystems. Geometric object-space techniques coarsely localize potential intersection regions or closest features between two objects, and image-space techniques compute the low-level proximity information in these regions. Most of the proximity information is derived from a distance field computed using graphics hardware. We demonstrate the performance in collision response computation for rigid and deformable body dynamics simulations. Our approach provides proximity information at interactive rates for a variety of simulation strategies for both backtracking and penalty-based collision responses.

Keywords: Proximity queries, collision detection, penetration depth, graphics hardware acceleration, multi-pass techniques.

1 INTRODUCTION

Many applications of computer graphics or computer simulated environments require spatial or proximity relationships between objects. In particular, dynamic simulation, haptic rendering, surgical simulation, robot motion planning, virtual prototyping, and computer games often require many different proximity queries simultaneously at interactive rates. We focus on interactive computation of the following proximity queries between 2D objects: collision detection, intersection, minimum separation distance, penetration depth, and contact points and normals.

Algorithms for determining collisions, intersections, and minimum separation distances have been extensively researched. Many are restricted to convex objects [2,4,6,16] or are based on hierarchical bounding-volume or spatial data structures that require considerable precomputation and are best suited for rigid geometry [8,12,14,19]. Some algorithms handle dynamically deforming geometry by either having prior knowledge of motion trajectories [22] or by using very specialized algorithms [1]. In our approach, we emphasize the handling of non-convex, dynamically deformable objects with no precomputation or knowledge of object motions.

Penetration depth is typically defined as the minimum translational distance needed to separate two objects. We define it with respect to a point as the minimum translational distance and direction needed to separate a penetrating point from an object's interior. This information is useful for penalty-based collision response computation. Dobkin et al. have presented an algorithm to compute the intersection depth of convex polytopes, though no practical implementation is known [3]. In general, no robust and efficient algorithms are known for computing the penetration depth and direction for general, non-convex primitives.

Our algorithm relies on the computation of discretized distance fields and graphics hardware-accelerated geometric computation. Distance fields – scalar fields that specify minimum distance to a shape for all points in the field – have been used for many applications in graphics, robotics and manufacturing [5,9]. Common algorithms for distance field computation are based on level sets [21] or adaptive techniques [5]. However, they either require static geometry, extensive preprocessing, or lack tight error bounds. Graphics hardware has been used to accelerate a number of geometric computations, such as visualization of constructive solid geometry models [7] and cross-sections and interferences [20]. However, these only compute intersections, not distance-related queries. Algorithms also exist for motion planning using graphics hardware acceleration and distance fields [11, 13,15,18]. More recently, an algorithm has been proposed to compute generalized Voronoi diagrams and distance fields using graphics hardware [10]. Its application to motion planning was presented in [11,18].

Our algorithm combines coarse traditional hierarchical approaches and multi-pass rendering techniques with the graphics hardware-accelerated distance field computation presented in [10]. The main features of our approach include a unified framework for all proximity queries, generality to non-convex polygons, no required precomputation or complex data structures, computational efficiency allowing interactive queries on current PCs, robustness requiring no special-case handling of degeneracies, portability across various CPU/graphics combinations, and error-bounds on approximations. We have implemented our algorithm on PC and SGI platforms, and demonstrated its performance in computing collision responses in both rigid- and deformable-body dynamic simulations. Our current algorithm and implementation focuses on 2D polygonal objects, but the basic design principles extend to 3D and are the focus of our current work.

2 OUR APPROACH

While algorithms exist for performing some of the proximity queries in both 2D and 3D, none meet all of our requirements even

in 2D. Our first step in developing a general unified approach that is efficient and robust in practice focuses on the general 2D proximity problem. Given a collection of 2D objects, we perform coarse geometric localization to find rectangular regions of space that contain either potential intersections or closest feature pairs between objects. We uniformly point-sample these regions and use polygon rasterization hardware to compute object intersections, closest points, and the distance field. The distance field and its gradient vector field provide the distance and direction to the nearest feature for each point in the localized region, which gives the contact normals, minimum separation distances, or penetration depths. Our core algorithm computes the proximity information between two 2D, simple, possibly non-convex polygons. Higher-order primitives are tessellated into polygons with bounded distance deviation error. In our hybrid approach, there are two top-level operations: (1) geometric object-space operations to coarsely localize potential intersection regions or closest features, and (2) image-based operations using graphics hardware to compute the proximity information in the localized regions.

2.1 Geometric Localization

The image-based queries operate on a uniform grid of sample points in regions of space containing potential interactions. The graphics hardware pixel framebuffer is used as the grid and the queries become pixel operations, therefore the performance varies dramatically with the pixel resolution. To avoid excessive load, a geometric localization step is used to window regions of potential interaction or as a trivial rejection stage. This hybrid geometry/image-based approach helps balance the load between the CPU and graphics subsystems, giving us portability between different workstations with varying performance characteristics. Using more sophisticated geometric techniques to tightly localize potential intersections or closest feature pairs dramatically reduces the graphics pipeline overhead, but increases the CPU usage and the complexity of the algorithm. We use coarse bounding-volume hierarchies to achieve this balance between speed and complexity, and CPU and graphics usage.

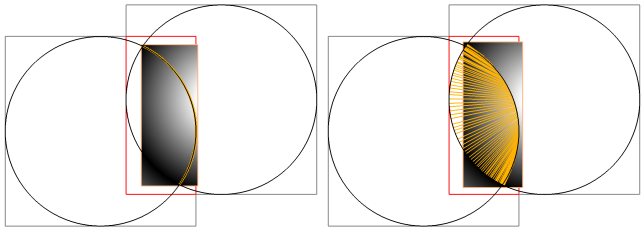


Figure 1: Points on the boundary of left circle intersecting the volume of the right circle, a tight-fitting bounding box around these penetrating points, and the distance field of the right circle computed in this bounding region (left). Gradient vectors at the penetrating points computed using central differencing in the distance field. The lengths represent the distance to the boundary (right). The top-level bounding boxes and their intersection used for computing the intersection points are also shown.

There are many general and efficient algorithms available for localizing geometry based on bounding-volume hierarchies [8,12,14,19]. However, for exact intersection testing these algorithms typically perform well only on static geometry where the hierarchy can be precomputed. In order to handle dynamic deformable geometry with no precomputation, we use coarse levels for efficient trivial rejection and to obtain reasonable geometric localization. In addition, we perform lazy evaluation of relevant portions of the hierarchies while performing the collision or distance query. A subtree rooted at a particular node is only computed if its children need to be visited during the query traversal. The trees are destroyed after every proximity query, and no actual tree data structures are required since the child nodes are recursively passed to the query routine. A maximum height of each object tree is used to balance the CPU and graphics load.

2.2 Image-based Proximity Queries

The proximity queries are simplified using uniform point sampling and accelerated with graphics hardware. This image-based approach helps decouple the objects' geometric complexity from the computational complexity for a specified error tolerance. The geometric localization step improves the performance since large areas of space and portions of the objects can be rejected from the query computation. We point-sample the geometry and the space around the geometry within the localized regions with a uniform rectangular grid and perform the queries on this volumetric representation using graphics hardware acceleration. The image-based queries include computing intersections between objects, computing the distance field of an object boundary, and computing the gradient of the distance field. Variations of these basic operations are used to perform the remaining queries.

2.2.1 Intersections

There are three types of intersections possible between two polygonal objects: boundary-boundary, boundary-volume, and volume-volume (boundary-volume is shown in Figure 1). We render both objects within a localized region using the graphics hardware and treat overwritten pixel sample points as the intersection points. The type of intersection determines whether the boundary or the interior of the object is rendered. Several strategies are given for detecting overwritten pixels (Table 1).

Multi-pass operations for finding object intersections

Buffer	Clear val	Render B	Render A	Intersection
Stencil	0	increment by 1	for all pixels==1, incr by 1	stencil value: 2
Color: blend ops	0,0,0	set color to 128,128,128	in color 127,127,127 with additive blend	color = 255,255,255
Color: logic ops	0,0,0	set color to 127,127,127	in color 128,128,128	color = 255,255,255
Color and Depth	0,0,0 and 1	depth = 0 depth func = always pass	depth = 0 depth func = equals Color = 255,255,255	color = 255,255,255

Table 1: OpenGL multi-pass rendering options for finding the overwritten pixels. The basic ops: a buffer is cleared; object B is rendered setting buffer values of all covered pixels; object A is rendered changing buffer values of pixels covered by A and B; intersection points are represented by pixels whose buffer values are set in the last pass. Each approach varies in performance, in the resulting buffer state, and in the sophistication needed in the underlying hardware implementation.

The error in the intersection calculation is governed by the pixel resolution. Given a distance error bound d, we choose a resolution so that no point in the rectangular region can be farther than d from a pixel sample point (d is the half diagonal length of a pixel grid cell). These error bounds hold for filled polygons, since all pixels in the interior of the polygon will be rasterized. Line segment rasterization does not guarantee that all pixels within d distance of the line will be set, so we draw an offset polygon surrounding the line segment that is d distance away from the line segment using the bounded-error distance mesh presented in [10].

The intersection operation requires clearing a buffer, rendering the objects into the potential intersection region, reading the buffer containing the intersection information, and searching through the image to find the intersection pixels. We avoid the full-screen clear by drawing a polygon the size of the localized region. Hardware min/max or histogram queries eliminate read back and the per-pixel search when no intersections have occurred, but

these operations may not be available on some platforms. In this case, the coarse bounding-volume hierarchy is used to reject object pairs. When the image operations dominate the query time, performance can be improved by increasing the error tolerance or by improving the geometric localization step by traversing deeper levels in the hierarchy. The running time of these image operations is largely independent of the object complexity, thus becoming negligible for complex objects.

The complexity of object rasterization grows linearly with respect to the number of vertices. Computing intersections geometrically between two polygon boundaries is worst case $O(n^2)$ since all edges could intersect. The complexity of our algorithm is O(n) where n is the number of vertices. The hierarchical geometric localization step is also O(n) since the maximum depth of the tree is held constant.

2.2.2 Distance Field

We use a variation of the algorithm described in [10] for constructing generalized Voronoi diagrams using graphics hardware for 2D polygonal objects. This approach computes an image-based representation of the Voronoi diagram in both the color and the depth buffers. A pixel's color identifies the polygon feature (vertex or edge) that is closest to that pixel's sample point; its depth value corresponds to the distance to the nearest feature. The depth buffer is an image-based representation of the distance field of the polygon boundary. The distance field is computed by rendering 3D bounded-error polygonal mesh approximations of the distance function where the depth of the rendered mesh at a particular pixel location corresponds to the distance to the nearest 2D polygon feature. Distance values at arbitrary points are bilinearly interpolated from the four nearest pixel distance values.

The algorithm by Hoff et al. only gives unsigned distance [10]. We need signed distances to avoid problems when computing the gradient near an object boundary for computing surface contact normals. We extend this algorithm to compute signed distances by distinguishing the inside and outside regions of the object using any of the available buffers to encode the "negative" interior of the object. We simply render the polygon, setting values in a pixel buffer. For each distance value, we also have a sign value that is read from this other buffer. Several possibilities include: setting the stencil buffer to 1; setting the color buffer to white; setting the most significant bit of the color ID in the Voronoi computation. For arbitrary points, the sign value can also be bilinearly reconstructed between 0 and 1. Values less than or equal to 0.5 can be positive and values greater than 0.5 can be negative.

Distance field computation requires clearing the depth buffer, rendering the objects' distance mesh, reading back the depth buffer, and rendering and reading of sign values. This is often more efficient than computing the intersection since we only need distance values at the intersection points (or at closest feature or penetrating points). In fact, we may not even need to read back the entire buffer since we could read just the individual pixel locations that we are querying (Figure 1).

2.2.3 Gradient of the Distance Field

We compute the gradient of the distance field at pixel locations by using central differences. For an arbitrary point, we compute the gradient as the bilinear interpolation of the gradients at the four surrounding pixel locations. In practice, this gives reasonable results even with the error and lack of C^1 or higher continuity in the polygonal distance mesh approximations used to compute the distance field (Figure 1). Gradients are computed in software for selected points after reading back the distance values. If the entire gradient field is desired, we could accelerate the computation using multi-pass rendering. For the *x* component of the gradient, we could subtractively blend the distance image shifted two pixels to the left with the original distance image. For the *y* component, we blend with the image shifted two pixels down. The division by 2 is performed by a multiplicative blend of 0.5. Unfortunately, subtractive blending is currently not available on all platforms even though it has been accepted into most graphics APIs, and the limited precision of pixel arithmetic may cause noticeable errors.

2.2.4 Other Proximity Queries

Given the basic operations of computing intersections, distance fields, and gradient of the distance field, we can perform the other proximity queries mentioned in section 1.

Penetration Depth and Direction: For a point on object A that is penetrating object B, we define the penetration depth and direction for the point as the distance and direction to the nearest feature on B. This is given by the distance field and its gradient computed at the penetrating point. Penetrating points are found using the intersection operation.

Contact Points and Normals: Ideally, the contact points are simply the intersections of the object boundaries; however, we often need the set of points that are almost in contact. For a given contact distance threshold *d*, we find all boundary points that are within *d* distance of each other. The basic approach is slightly modified to efficiently handle this query. First of all, in the geometric localization stage we find the potential intersection between the two polygons that are slightly thicker. This is handled by enlarging all bounding boxes by *d* in each. We then find the intersection between the boundaries by drawing the objects' boundary line segments with an enlarged offset of *d* (using the distance mesh from [10]) and finding intersecting pixels. Normals at each contact point are computed from the gradient of the signed distance field (signed distance to avoid distance discontinuity near the object boundary).

Closest Point: We find the point on object A that is closest to object B by rendering the boundary of object A in the localized region of A containing its closest feature, rendering the distance field of B in this same region, and then searching the boundary points of A and finding the point that is closest to B.

Separation Distance and Direction: We find the minimum separation distance and direction between two objects A and B by first computing the closest point on A to B and vice versa. Ideally, we find the closest point on B to A from the distance value and gradient at the closest point on A to B, but the amplification of errors over the greater distance may cause problems. The distance between these two closest points is the separation distance and the line segment between them gives the separation direction.

3 PERFORMANCE

We demonstrate the effectiveness of our proximity queries in computing collision responses for interactive dynamic simulations of rigid and deformable objects. We compute the collision response for a particle and use a collection of particle responses to extend to rigid and deformable bodies [1,19]. We implemented collision responses for simulations with and without penetration constraints. In constrained simulation, penetration is avoided through a backtracking algorithm that finds the state of all objects "just before" a collision. A bisection search in time is performed between the last non-collision state and the collision state for all

objects in the scene, and the collision response is then computed for the objects that are in close contact. In unconstrained simulation, penetration is allowed, but a spring-based restoring penalty force proportional to the penetration depth is applied to the object until separation occurs. Collision responses between object pairs are handled locally without requiring global update of the entire system.

Each collision response requires different proximity information. Constrained simulation requires points of *close* contact and the contact normals. Unconstrained simulation requires points of penetration and their penetration depths. The effectiveness of our approach is most clearly shown in the unconstrained, penalty-based approach because of the difficulty in computing penetration depth. Earlier algorithms give only coarse approximations without error bounds or are only restricted to convex objects.

We tested the system in several different contact scenarios. In each simulation, the user provides the initial position, orientation, and velocity of a collection of objects, and the appropriate collision responses are computed as the simulation advances. See the colorplate for descriptions of the simulations. Each simulation is performed on rigid-bodies except for **wavy**, but the same proximity query algorithm was used on all simulations. More deformable bodies were not shown because of the difficulty in developing effective deformable simulations. In table 2, we show the average total per-frame proximity query times. **Wavy** requires more time because there are large areas of continuous close contact as the shapes conform to each other when colliding. In table 3 we show the effects of the distance error on performance.

Average Total Per-frame Proximity Query Times					
Demo	**Objects**	**Lines**	**GeForce2**	**InfiniteReality2**	**ATI Rage Pro LT**
Map	6	719	0.281ms	0.901ms	0.434ms
Gears	13	391	0.015	0.026	0.064
Links	15	440	0.020	0.052	0.038
Cars	18	266	0.007	0.026	0.015
Wavy	2	200	1.030	2.360	2.990

Table 2: Performance timings for dynamics simulations. The number of objects, number of line segments, and the average total time in milliseconds to run proximity queries on all objects in the scene per frame is reported. Timing data was gathered from three machines: a Pentium-III 933MHz desktop with a 64Mb GeForce2, a SGI 300MHz R12000 with InfiniteReality2 graphics, and a Pentium-III 750Mhz laptop with ATI Rage Pro LT graphics.

Effects of Error Tolerance on Performance of Wavy			
Error	**GeForce2**	**InfiniteReality2**	**ATI Rage Pro LT**
d/4	0.710ms	1.270ms	5.560ms
d/2	0.315	1.000	1.850
d	0.211	0.930	0.895
2d	0.176	0.879	0.631
4d	0.165	0.876	0.535

Table 3: The effect on performance when changing the distance error tolerance *d*. We used proximity queries on the **wavy** demo with no collision response. The error determines the number of pixels used in the image-based operations. Systems with low graphics performance are more directly affected by the choice of *d* (see ATI Rage Pro LT); however, as the error is increased there is less dependence on graphics performance and the faster laptop CPU overtakes the InfiniteReality2 system.

4 CONCLUSION

We have presented a hybrid geometry- and image-based algorithm for computing geometric proximity queries between two arbitrary 2D objects using graphics hardware. This approach has a number of advantages over previous approaches since the unified framework allows us to compute all the queries, including penetration depth and contact normals. Furthermore, it involves no precomputation and handles non-convex polygons; as a result, it is also applicable to dynamic or deformable geometric primitives. In practice, we have found the algorithm to be simple to implement, quite robust, fast (considering the complexity of the queries), and very flexible. We have developed an interactive 2D dynamic simulation system for rigid and deformable objects to illustrate the effectiveness of our approach. We are currently extending this framework to 3D for interactive proximity queries on complex, dynamic geometry.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful suggestions. This work was supported by ARO DAAG55-98-1-0322, DOE ASCII Grant, NSF NSG-9876914, NSF DMI-9900157, NSF IIS-982167, ONR Young Investigator Award, and Intel.

REFERENCES

[1] D. Baraff, *Dynamic Simulation of Non-Penetrating Rigid Bodies*. Ph.D. Thesis, Dep of Comp. Sci., Cornell University, March 1992

[2] S. Cameron, *Enhancing GJK: Computing Minimum and Penetration Distance between Convex Polyhedra*. International Conference on Robotics and Automation, 3112-3117, 1997

[3] D. Dobkin, J. Hershberger, D. Kirkpatrick, S. Suri, *Computing the Intersection Depth of Polyhedra*. Algorithmica, 9(6), 518-533, 1993

[4] S. Ehmann and M. Lin. *Accelerated Proximity Queries Between Convex Polyhedra By Multi-Level Voronoi Marching*. Proc. International Conf. on Intelligent Robots and Systems, 2000

[5] S. Frisken, R. N. Perry, A. P. Rockwood, T. R. Jones, *Adaptively Sampled Distance Fields: A General Representation of Shapes for Computer Graphics*. SIGGRAPH 00, 249-254, July 2000

[6] E. G. Gilbert, D. W. Johnson, S.S. Keerthi. *A Fast Procedure for Computing the Distance Between Objects in Three-Dimensional Space*. IEEE J. Robotics and Automation, RA(4): 193-203, 1988

[7] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs. *Near Real-time CSG Rendering Using Tree Normalization and Geometric Pruning*. IEEE Computer Graphics and Applications, 9(3):20–28, May 1989

[8] S. Gottschalk, M. C. Lin, D. Manocha, *OBB-Tree: A Hierarchical Structure for Rapid Interference Detection*. SIGGRAPH 96, 171-180, 1996

[9] G. Hirota, S. Fisher, M. Lin. Simulation of Non-penetrating Elastic Bodies Using Distance Fields. University of North Carolina at Chapel Hill Technical Report: TR00-018. Spring 2000

[10] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. *Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware*. SIGGRAPH 99, 277-285, 1999

[11] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. *Interactive Motion Planning Using Hardware-Accelerated Computation of Generalized Voronoi Diagrams*. Proc. of IEEE International Conf. on Robotics and Automation, 2000

[12] P. M. Hubbard, *Interactive Collision Detection.* IEEE Symposium on Research Frontiers in Virtual Reality. 24-31, 1993

[13] R. Kimmel, N. Kiryati, A. Bruckstein, *Multi-Valued Distance Maps for Motion Planning on Surfaces with Moving Obstacles.* IEEE Transactions on Robotics and Automation, vol 14: 427-438, 1998

[14] D. Johnson, E. Cohen, *A Framework for Efficient Minimum Distance Computation*, IEEE Conf. On Robotics and Animation, 3678-3683, 1998

[15] J. Lengyel, M. Reichert, B.R. Donald, and D.P. Greenberg. *Real-time Robot Motion Planning Using Rasterizing Computer Graphics Hardware*. Computer Graphics (SIGGRAPH 90 Proc.), vol. 24, pgs 327–335, Aug 1990

[16] M. Lin, J. Canny. *Efficient Algorithms for Incremental Distance Computation*. IEEE Transactions on Robotics and Automation, 1991

[17] B. Mirtich, *Impulse-Based Dynamic Simulation of Rigid Body Systems*. Ph.D. Thesis, University of California, Berkeley, Dec 1996

[18] C. Pisula, K. Hoff, M. Lin, and D. Manocha. *Randomized Path Planning for a Rigid Body Based on Hardware Accelerated Voronoi Sampling*. Proc. of Workshop on Algorithmic Foundations of Robotics, 2000

[19] S. Quinlan, *Efficient Distance Computation between Non-Convex Objects*. International Conf. on Robotica and Automation, 3324-3329, 1994

[20] J. Rossignac, A. Megahed, and B. Schneider. *Interactive Inspection of Solids: Cross-sections and Interferences*. SIGGRAPH 92, 26, 353–360, July 1992.

[21] J. Sethian, *Level Set Methods*, Cambridge University Press, 1996

[22] J. Snyder, A. Woodbury, K. Fleischer, B. Currin, A. Barr, *Interval Methods for Multi-Point Collisions Between Time Dependent Curved Surfaces*. ACM Computer Graphics, 321-334, 1993

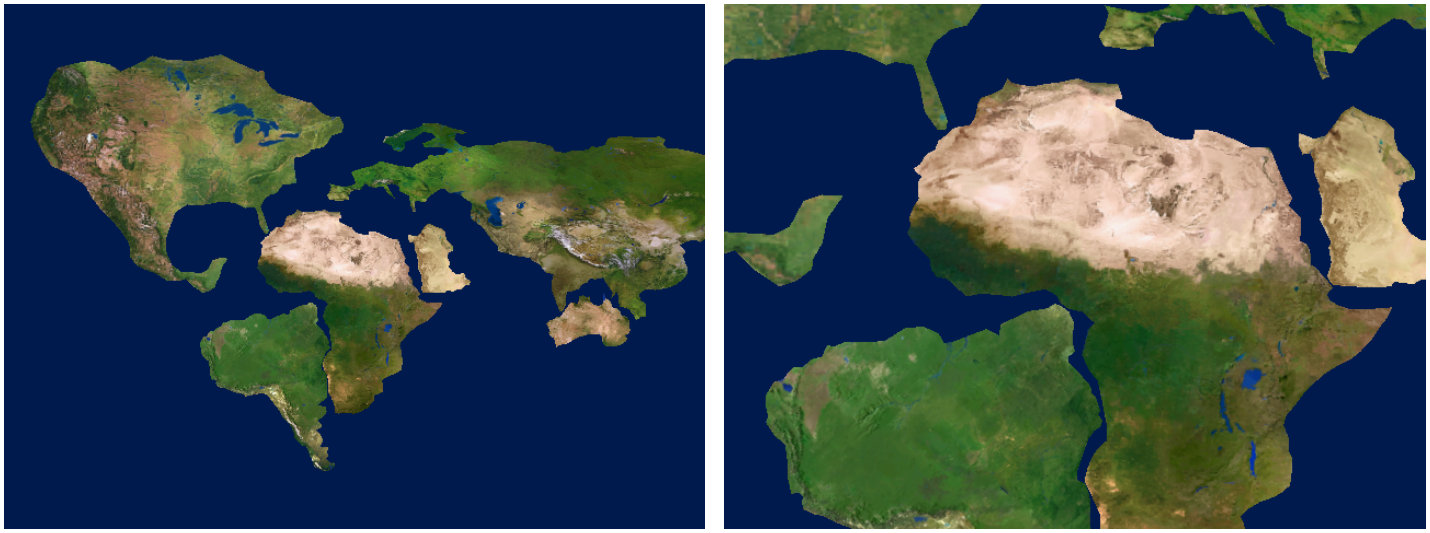


PLATE 1: **Map** (large non-convex objects, frequent simultaneous close contact). Our approach computes proximity query information needed for penalty-based collision response between complex non-convex objects. For each penetrating point, we compute a minimal penetration depth and direction and apply a penalty force to resolve the collision. Intersections between the top-level axis-aligned bounding boxes were used as potential intersection regions. A coarse hierarchical search with oriented bounding boxes would find smaller potential intersection regions, thus improving performance. Even with this simplified search, we achieved interactive performance on several difference machines with widely varying CPU/graphics combinations (see Table 2).

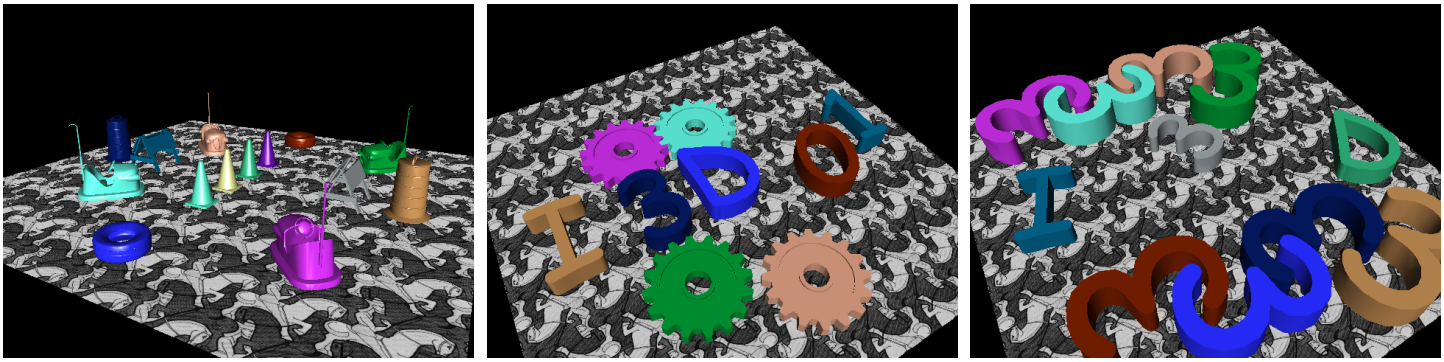


PLATE 2: **Cars** (convex objects, less frequent contact), **Gears** (non-convex, less frequent interlocking contact), and **Links** (non-convex objects, frequent simultaneous interlocking contacts) demos. Collision responses in some specialized 3D scenes, such as those whose objects collide only in the 2D plane, can be computed using our approach. The 2D projection of each object onto the plane is used for the dynamics simulation. The left-image shows our method applied to a standard non-penetrating backtracking collision response method where contact points and normals are computed. All of the other simulations use the penalty-based collision response based on penetrating points and their penetration depths and directions. The right two images show collision responses between complex interlocking non-convex objects which are easily handled without specialized techniques such as convex decomposition.

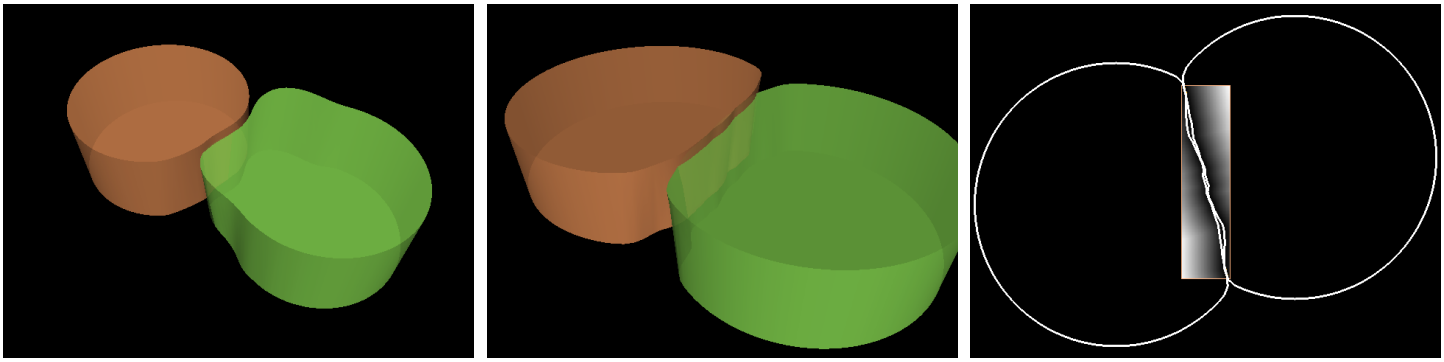


PLATE 3: **Wavy** (large deformable-bodies, continuous contact). Other important characteristics of our proximity query algorithm include not requiring any precomputation or complex data structures. Here we show proximity information being used for collision response between dynamically deformable bodies. The left image shows a wave propagating through the right object and hitting the left object. The collision response causes the object to become indented and creates a reaction wave in the left object. Many dynamics simulations resolve collisions by backing up the simulation to a moment before contact, we use a penalty-based method that applies a force at each penetrating point based on the amount of penetration. The center image shows the penalty-based response between two objects that were initially overlapping by a large amount. The right image shows the distance field around the contact area.

Primitive Tests for Collision Detection

Dave Eberle
PDI/Dreamworks R&D

Contents

- What is collision detection?
- What do we call a primitive?
- Overview of various techniques we have at our disposal to determine if two primitives collide?
- What are the tradeoffs between these techniques?

What is collision detection?

- Given two objects determine if they collide? – Trivial!

What is collision detection?

- Given two objects determine if they collide? – Trivial!
- Do the objects share a common point in space at the same time?

What is collision detection on a computer?

- Time is typically treated in a discrete fashion.

What is collision detection on a computer?

- Time is typically treated in a discrete fashion.
- Leads to alternate tests to answer questions for domains specific applications.

Application influence

- Spaceship game flying through asteroids.
- Collision results in spaceship exploding.
- Test that provides a boolean answer is sufficient.

Application influence

- Car racing game.
- Collision results vehicle bouncing and spinning in a physically plausible way .
- Test that provides a boolean result is insufficient.
- Needs some information about the point of contact.

Application influence

- Car racing game.
- Collision results vehicle bouncing and spinning in a physically plausible way .
- Test that provides a boolean result is insufficient.
- Needs some information about the point of contact.
- Unless you are racing a Ford Pinto ☺

Static Intersection Tests

- Examine state of geometry at an instant in time.
- Are typically very fast.
- Many times return only a boolean result.
- Too many specific tests to cover.
- Separating axis theorem is a common foundation for many tests.

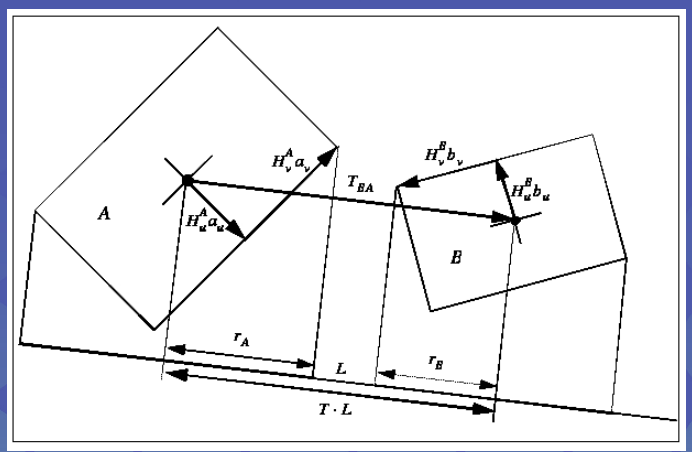
Separating Axis Theorem

For any two arbitrary convex, disjoint, polyhedra **A** and **B**, there exists *a separating axis where the projections of the polyhedra, which form intervals on the axis, are also disjoint. If* ***A*** *and* ***B*** *are disjoint, they can be separated by an axis that is orthogonal to plane formed by one of the following:*

1. *A face normal of polyhedron* ***A****.*
2. *A face normal of polyhedron* ***B****.*
3. *A normal formed by the cross product of a pair of edges with one from* ***A*** *and the other from* ***B****.*

Dave Eberle

Separating Axis Theorem applied to oriented boxes.



Dave Eberle

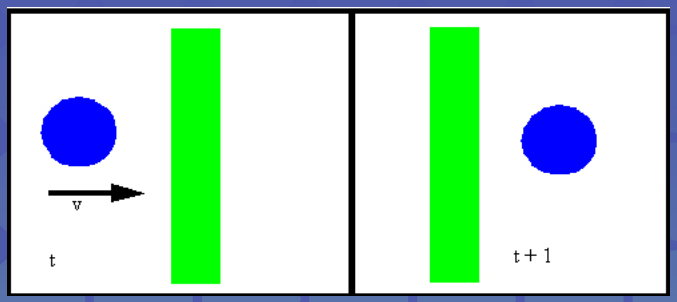
Static Intersection Tests
..continued

- Generally do not provide data for response.
- Can provide a quick rejection of intersection.
- Useful as basic components of more complex collision detection algorithms.

Dave Eberle

Continuous and Moving Primitive Intersection Tests

- Temporal Aliasing Problem:
  Static tests can miss collisions if the objects are too small or moving too fast.



Dave Eberle

Continuous and Moving Primitive Intersection Tests

- Extrude geometry and perform static test?
  - Typically results in only a boolean result.
- Reformulate test and treat time in a continuous fashion.

Continuous Collision Tests

- Report first time of contact.
- Can provide accurate collision data.
- Assume objects are initially disjoint.
- Add robustness to many strategies by overcoming the temporal aliasing problem.
- Are generally more expensive than static intersection tests.

Proximity Queries

- Track and bound distance between primitives or features of objects.
- Collision reported when distance is below small threshold.
- Typically restricted to convex geometry.
- Can be used to prevent objects from ever penetrating.
- Can slow down simulation applications with many objects or collision events.

Proximity Queries – Penetration Depth

- Can report signed distance between primitives/objects.
- Objects can be allowed to penetrate.
- Response is not always the most accurate.
- Can be used to avoid bottleneck in simulation of large numbers of objects caused by other methods.

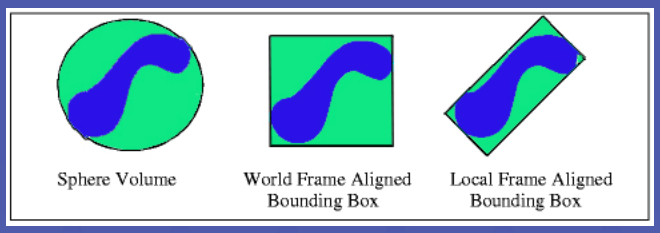
Ray-Primitive Tests

- Simpler than object/object intersection tests in many cases.
- Often good enough for a variety of applications: Shooting in games, cars driving over terrain.
- Abundance of literature from ray-tracing for rendering purposes.
- Typically provides a point of intersection if the ray hits a primitive.

Regular Height Fields

- A primitive representation useful for representing landscapes.
- Offers fast ray intersection methods.
- Are memory efficient.

Primitives as Bounding Volumes



- Use primitives to bound complex objects.
- Use simpler primitives to bound more complex primitives. Example: Box primitive bounding a triangle.
- Use static intersection tests to try to obtain a fast rejection.

Dave Eberle

Trade offs in choice of BV primitives

- How well does the bounding volume fit the underlying geometry?
- What is the cost in updating the bounding volume if the object is moving?
- What is the cost of the bounding volume intersection test.

Dave Eberle

General Strategies in Algorithm Design

- Perform simple tests first that may lead to an early exit.
- If possible try to cache results from previous tests.
- Try to reduce the dimension of the problem.

Summary

- Application dictates the complexity of what is needed.
- Different categories of tests provide different information.
- Overview of strengths and weaknesses of methods in each category.

Collision Detection for Deformable Objects

SIGGRAPH2004

Pascal Volino
MIRALab, Univ. of Geneva

Deformable Objects

SIGGRAPH2004

- Cloth
- Soft volumes
- …

- Mostly animated by mechanical simulation
- In most cases: Collision detection on deformable surfaces

Detection Strategies for Deformable Objects

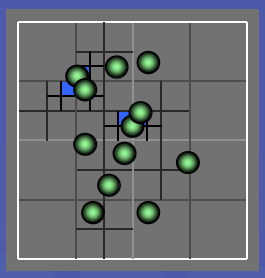
- Usually, collision detection between or within discretized curved surfaces
  - Polygonal meshes of flat primitives
    - Triangles, quadrangles…
  - Curved primitives
    - Bezier patches, subdivision surfaces
- Problem: Managing the complexity of large numbers of primitives within the objects

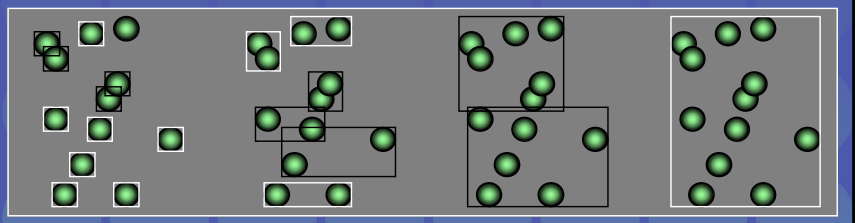
Detection Strategies for Deformable Objects

- Hierarchies
  - Reduction of combinatory tests through groupings
- Two main possibilities
  - Space subdivision hierarchies
    - Grouping primitives through their proximities
  - Object subdivision hierarchies
    - Grouping primitives through their adjacencies

Detection Strategies for Deformable Objects

- Space Subdivision Hierarchies
  - Voxel grids, Octrees
- Object Subdivision Hierarchies
  - Bounding Volume Hierarchies

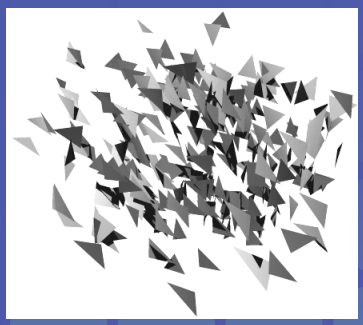




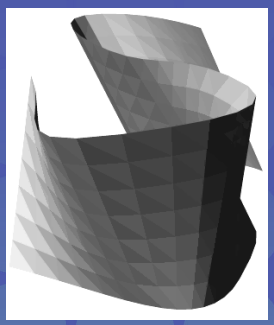
Pascal Volino

Detection Strategies for Deformable Objects

  - Usually, deformable surfaces
- Take advantage of local position consistency between elements



Polygon Soup



Polygonal Mesh

Pascal Volino

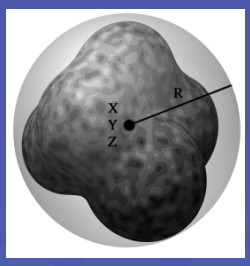
Detection Strategies for Deformable Objects

- Constant topological invariance between surface elements
- Object hierarchies seem more appropriate
  - Real collisions occur when topologically distant elements are geometrically close
  - Consistency: Nonsystematic update of the hierarchy structure
    - Rare or inexistent updates

Bounding Volumes

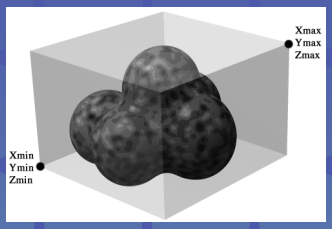
- Qualities
  - Tightness (avoid false positives)
  - Time of computation from primitives
  - Time of union combinations
  - Time of update on object motion
  - Time of intersection test
- Contextual choice of bounding volumes

Bounding Volumes



- Popular choices
  - Bounding spheres or ellipsoids
  - Bounding boxes or Discrete Orientation Polytopes

- Two schemes
  - Object-oriented volumes
  - Axis-oriented volumes



Bounding Volumes

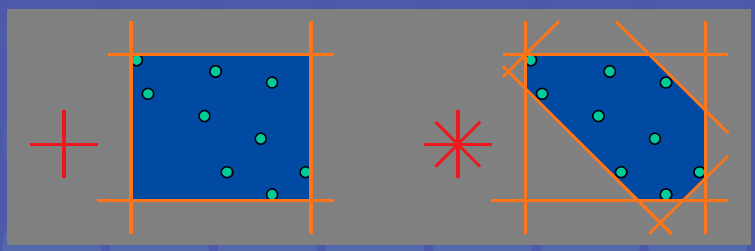
- Object-Oriented Volumes
  - Avoid recomputation of the hierarchy in case of rigid motion of the whole object
  - Optimal axis orientation
    - Flat or elongated objects
  - Problem: Combining and testing collisions between volumes of different axes
    - Avoided by the use of axis-independent volumes (spheres), which are however inefficient
  - Adapted for rigid objects of complex shape
    - Robotics

Bounding Volumes

- Axis-Oriented Volumes
  - Simple and fast computation
  - Problem: Not optimally bounding
    - Flat or elongated objects
  - Adapted when no rigid-motion consistency is found within the object
    - Deformable objects
  - Popular choice: Bounding box

Bounding Volumes

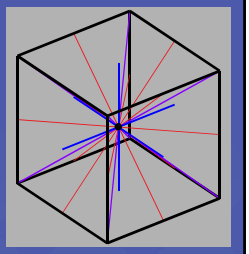
- Improving the bounding box
  - Generalizing the box with more directions

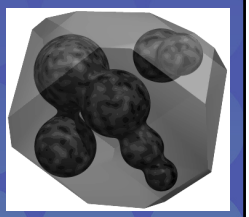


- Discrete Orientation Polytopes

Bounding Volumes

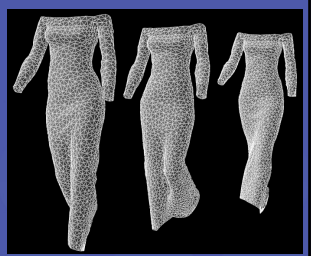
- Discrete Orientation Polytopes
  - Arbitrary number of directions
  - Popular choice: Directions toward the face, edge and vertices of a cube
    - The more directions: The tighter the volume, but also the more computations needed
    - Infinite directions: Convex hull of the object
  - Optimal choice of directions





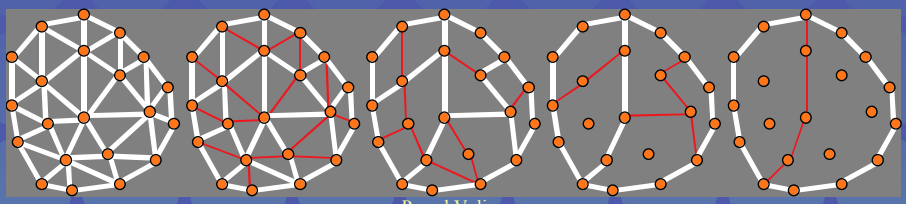
Collision Detection on Polygonal Meshes

- Choices
  - Bounding volume hierarchies
    - Defined on the topology of the objects
    - Marginally or not updated
  - Axis-aligned bounding volumes
    - DOPs for bounding flat surfaces with correct tightness
    - Updated along object motion and deformation



Collision Detection on Polygonal Meshes

- Building a suitable hierarchy on the mesh
  - Build strategies:
    - Root-Leaf approach: Subdividing groups of mesh elements
    - Leaf-root approach: Joining groups of mesh elements

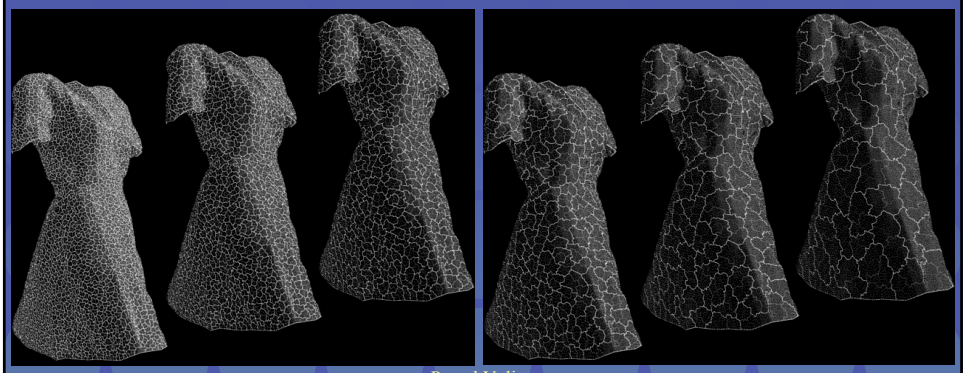


Collision Detection on Polygonal Meshes

- Building a suitable hierarchy on the mesh
  - Structural quality:
    - O(1) children for each node
    - O(log n) maximal tree depth
  - Optimal shape of surface regions
    - As "round" as possible
    - Minimize size of bounding volume
    - Maximize Sqrt(SurfaceArea)/ContourLength

Collision Detection on Polygonal Meshes

- Mesh Hierarchies
  - Multiresolution model of the surface

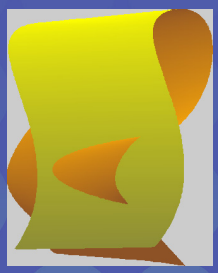


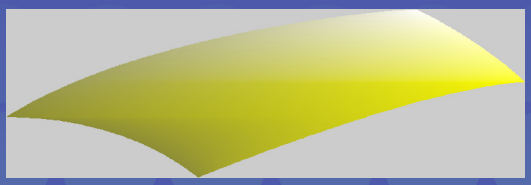
Self-Collision Detection

  - Almost same algorithms, but…
- Self-collision detection is inherently inefficient
  - How to make difference between adjacent and actually colliding elements?
  - Standard algorithms will waste time detecting all adjacencies

Self-Collision Detection

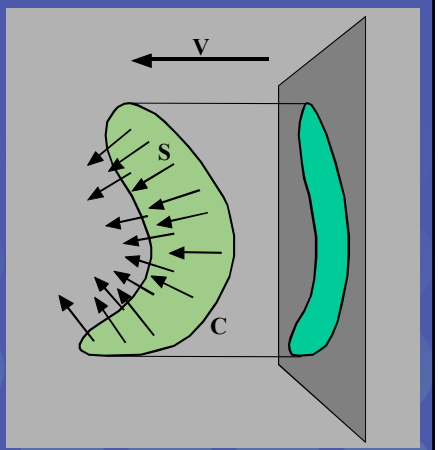
- The curvature criteria
  - There are no self-collisions within surfaces that are not curved enough to form a loop





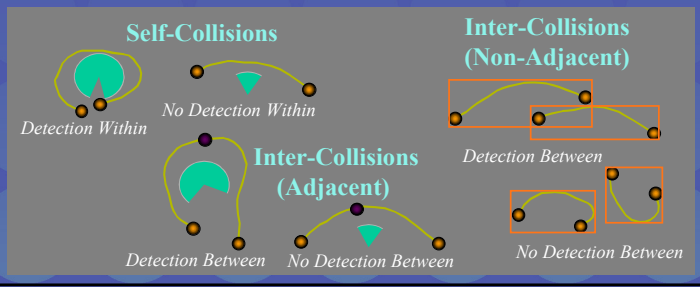
Self-Collision Detection

- The curvature criteria
  - If there exists a direction with positive dot product to the surface normal all over a surface region
  - And if the projection of the surface contour on a plane orthogonal to that direction does not have any self-collisions
  - Then there is no self-collisions within that surface region



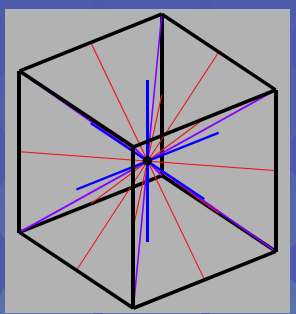
Self-Collision Detection

- Integrating curvature tests in the hierarchical algorithm
  - Replacing bounding volume by curvature tests
    - For self-collision detection within one region
    - For collision detection between two adjacent regions



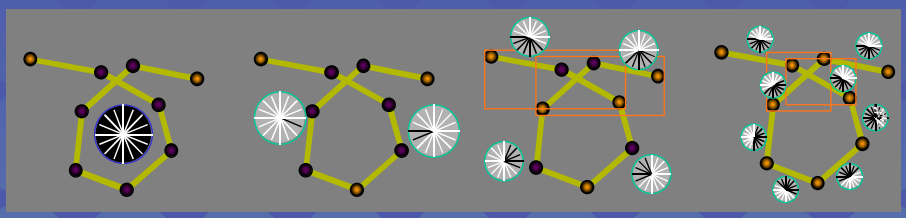
Self-Collision Detection

- Adding curvature information in the hierarchy
  - Direction cones
    - Defined by a direction and an angle
  - Direction samples
    - Which directions of a predefined direction set has positive dot product
      - Customizable accuracy
      - Very fast propagation up in the hierarchy



Self-Collision Detection

- Combining bounding volume and direction tests for self-collision detection
  - Quickly discarding flat surface areas from self-collision tests

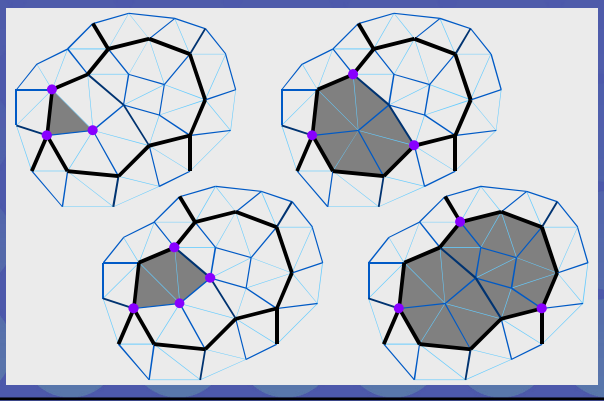


Self-Collision Detection

- Adjacency tests
  - Two surface regions sharing at least one vertex
  - Between any node of the hierarchy
  - Using an adequate numbering scheme in case of regular meshes (quadtree, subdivision patches)
  - Using well-selected stored vertices from the surface boundary

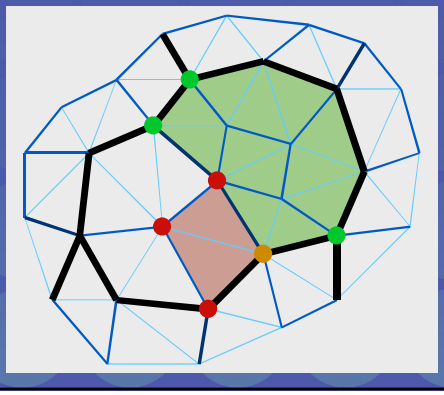
Self-Collision Detection

- Storing O(1) vertices in each node
  - Only those separating two other adjacent regions of same level



Self-Collision Detection

- O(1) adjacency check
  - Adjacent nodes have at least one common stored vertex



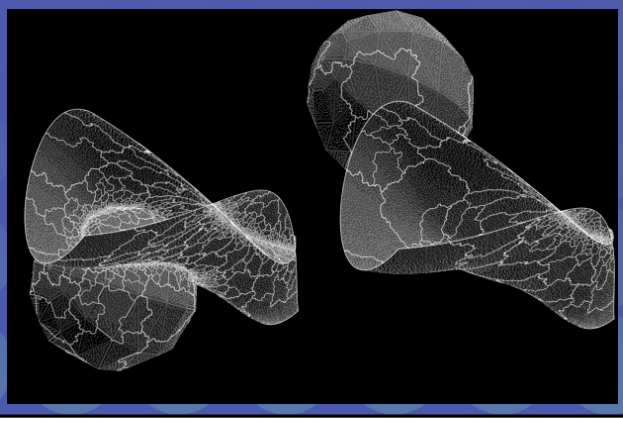
Self-Collision Detection

- Self-collisions on contours
  - May cause self-collisions within surfaces despite fulfilling low-curvature criteria
  - Likely to occur in highly deformed elongated surfaces (bended in cone patterns) or concave shapes
  - Situations unlikely to occur in real-life simulations

Collision Detection Proposed Algorithm

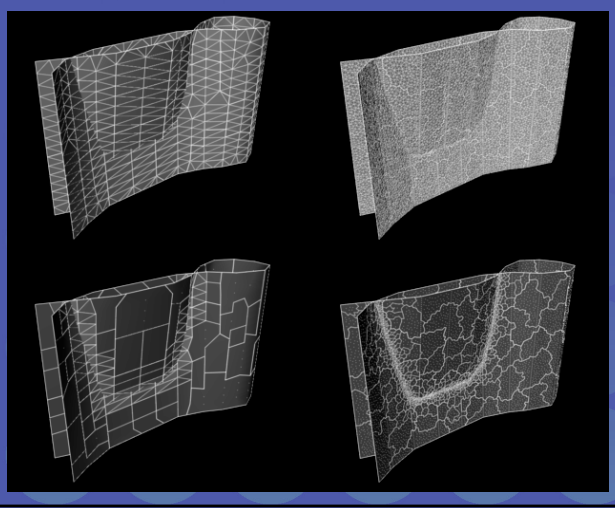
- Preprocessing
  - Construct the hierarchy tree
- Processing for each frame
  - Update the bounding volumes
  - Update curvature info if self-collision needed
  - Browse the hierarchy for collisions
    - Using bounding volume tests
    - Using curvature tests for self-collisions or adjacencies

Detection Efficiency



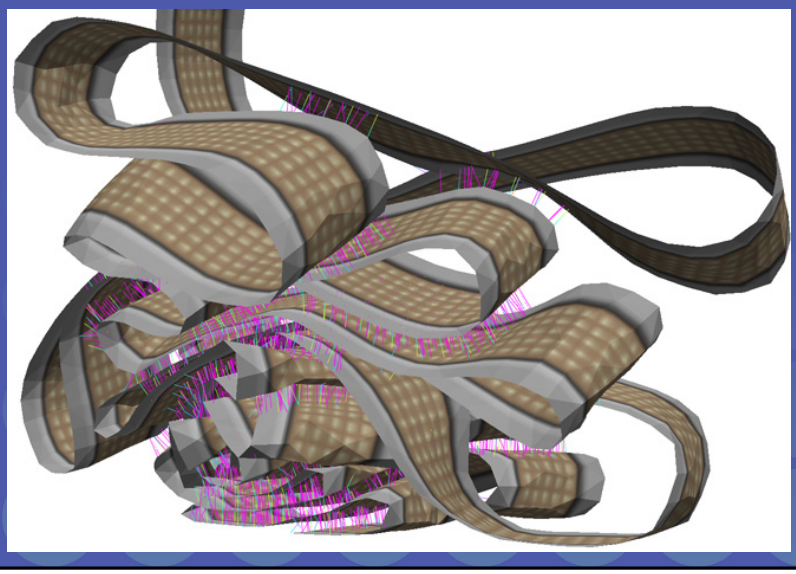
- Curvature tests restore native efficiency of hierarchical collision detection in the case of self-collisions

Detection Efficiency



- Detection time not too affected by discretization

Detection Efficiency



Fast Collision Detection Between General Polyhedral Models

Ming C. Lin
lin@cs.unc.edu
http://www.cs.unc.edu/~lin
http://gamma.cs.unc.edu/
University of North Carolina at Chapel Hill

Methods for General Models

- Decompose into convex pieces, and take minimum over all pairs of pieces:
  - Optimal (minimal) model decomposition is NP-hard.
  - Approximation algorithms exist for closed solids, but what about a list of triangles?
- Collection of triangles/polygons:
  - n*m pairs of triangles - brute force expensive
  - Hierarchical representations used to accelerate minimum finding

Hierarchical Representations

- Two Common Types:
  - Bounding volume hierarchies – trees of spheres, ellipses, cubes, axis-aligned bounding boxes (AABBs), oriented bounding boxes (OBBs), K-dop, SSV, etc.
  - Spatial decomposition - BSP, K-d trees, octrees, MSP tree, R-trees, grids/cells, space-time bounds, etc.
- Do very well in “rejection tests”, when objects are far apart
- Performance may slow down, when the two objects are in close proximity and can have multiple contacts

BVH vs. Spatial Partitioning

BVH:

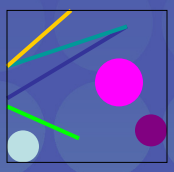
- Object centric

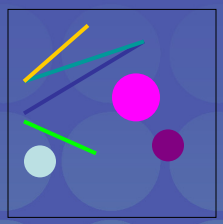
- Spatial redundancy

SP:

- Space centric

- Object redundancy





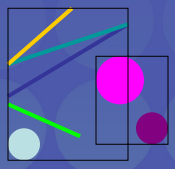
BVH vs. Spatial Partitioning

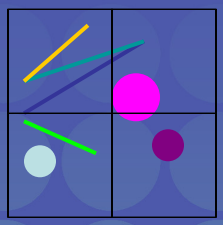
BVH:

- Object centric
- Spatial redundancy

SP:

- Space centric
- Object redundancy





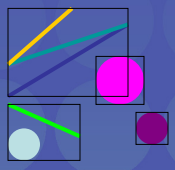
BVH vs. Spatial Partitioning

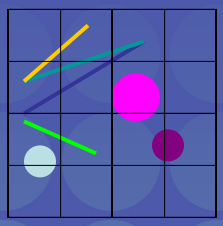
BVH:

- Object centric
- Spatial redundancy

SP:

- Space centric
- Object redundancy





BVH vs. Spatial Partitioning

BVH:

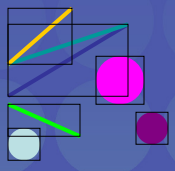
- Object centric

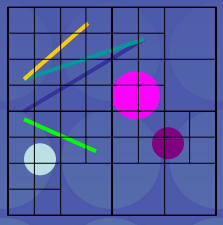
- Spatial redundancy

SP:

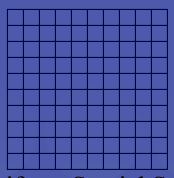
- Space centric

- Object redundancy

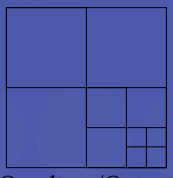




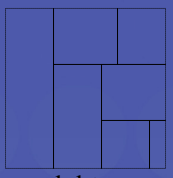
Spatial Data Structures & Subdivision



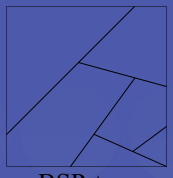
Uniform Spatial Sub



Quadtree/Octree



kd-tree



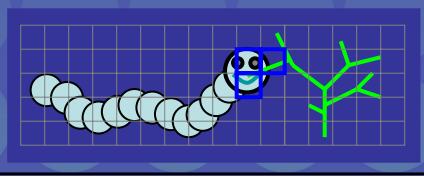
BSP-tree

- Many others……

  (see the lecture notes)

Uniform Spatial Subdivision

- Decompose the objects (the entire simulated environment) into identical cells arranged in a fixed, regular grids (equal size boxes or voxels)
- To represent an object, only need to decide which cells are occupied. To perform collision detection, check if any cell is occupied by two object
- Storage: to represent an object at resolution of *n* voxels per dimension requires upto n^3 cells
- Accuracy: solids can only be "approximated"

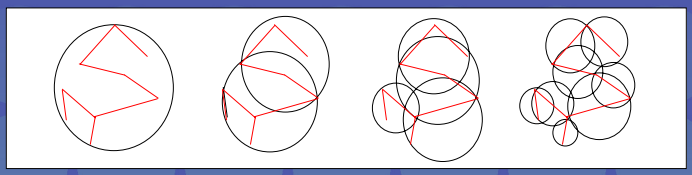


Octrees

- *Quadtree* is derived by subdividing a 2D-plane in both dimensions to form quadrants
- Octrees are a 3D-extension of quadtree
- Use divide-and-conquer
- Reduce storage requirements (in comparison to grids/voxels)

Bounding Volume Hierarchies

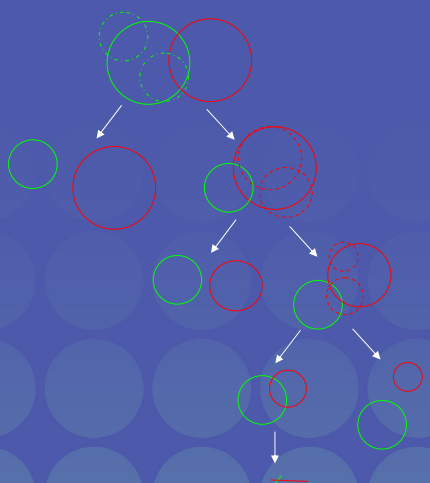
- Model Hierarchy:
  - each node has a simple volume that bounds a set of triangles
  - children contain volumes that each bound a different portion of the parent's triangles
  - The leaves of the hierarchy usually contain individual triangles
- A binary bounding volume hierarchy:



Type of Bounding Volumes

- Spheres
- Ellipsoids
- Axis-Aligned Bounding Boxes (AABB)
- Oriented Bounding Boxes (OBBs)
- Convex Hulls
- k-Discrete Orientation Polytopes (k-dop)
- Spherical Shells
- Swept-Sphere Volumes (SSVs)
  - Point Swetp Spheres (PSS)
  - Line Swept Spheres (LSS)
  - Rectangle Swept Spheres (RSS)
  - Triangle Swept Spheres (TSS)

BVH-Based Collision Detection



Collision Detection using BVH

1. Check for collision between two parent nodes (starting from the roots of two given trees)
2. If there is no interference between two parents,
3. Then stop and report “no collision”
4. Else All children of one parent node are checked against all children of the other node
5. If there is a collision between the children
6. Then If at leave nodes
7. Then report “collision”
8. Else go to Step 4
9. Else stop and report “no collision”

Evaluating Bounding Volume Hierarchies

Cost Function:

$$F = N_u \times C_u + N_{bv} \times C_{bv} + N_p \times C_p$$

F: total cost function for interference detection

N_u: no. of bounding volumes updated

C_u: cost of updating a bounding volume,

N_{bv}: no. of bounding volume pair overlap tests

C_{bv}: cost of overlap test between 2 BVs

N_p: no. of primitive pairs tested for interference

C_p: cost of testing 2 primitives for interference

Designing Bounding Volume Hierarchies

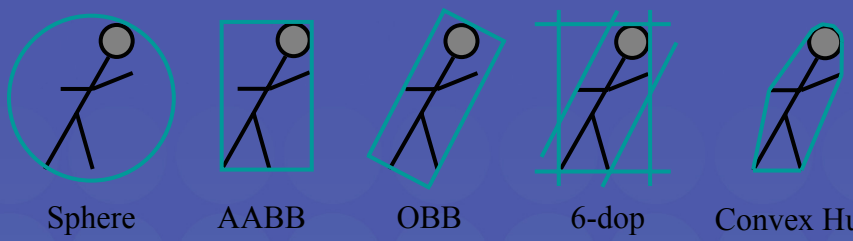
The choice governed by these constraints:

- It should fit the original model as tightly as possible (to lower N_{bv} and N_p)
- Testing two such volumes for overlap should be as fast as possible (to lower C_{bv})
- It should require the BV updates as infrequently as possible (to lower N_u)

Observations

- Simple primitives (spheres, AABBs, etc.) do very well with respect to the second constraint. But they cannot fit some long skinny primitives tightly.
- More complex primitives (minimal ellipsoids, OBBs, etc.) provide tight fits, but checking for overlap between them is relatively expensive.
- Cost of BV updates needs to be considered.

Trade-off in Choosing BV's



Building Hierarchies

- Choices of Bounding Volumes
  - cost function & constraints
- Top-Down vs. Bottum-up
  - speed vs. fitting
- Depth vs. breadth
  - branching factors
- Splitting factors
  - where & how

Sphere-Trees

- A *sphere-tree* is a hierarchy of sets of spheres, used to approximate an object
- Advantages:
  - Simplicity in checking overlaps between two bounding spheres
  - Invariant to rotations and can apply the same transformation to the centers, if objects are rigid
- Shortcomings:
  - Not always the best approximation (esp bad for long, skinny objects)
  - Lack of good methods on building sphere-trees

Methods for Building Sphere-Trees

- "Tile" the triangles and build the tree bottom-up
- Covering each vertex with a sphere and group them together
- Start with an octree and "tweak"
- Compute the medial axis and use it as a skeleton for multi-res sphere-covering
- Others……

k-DOP's

- *k*-dop: *k*-discrete orientation polytope a convex polytope whose facets are determined by half-spaces whose outward normals come from a small fixed set of *k* orientations
- For example:
  - In 2D, an 8-dop is determined by the orientation at +/- {45,90,135,180} degrees
  - In 3D, an AABB is a 6-dop with orientation vectors determined by the +/-coordinate axes.

Choices of k-dops in 3D

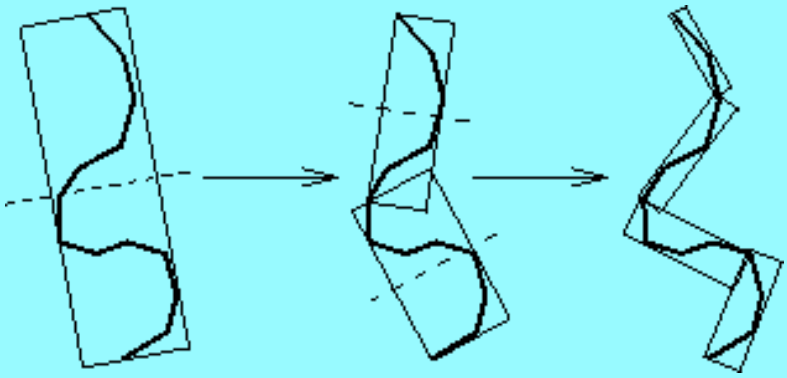
- 6-dop: defined by coordinate axes
- 14-dop: defined by the vectors (1,0,0), (0,1,0), (0,0,1), (1,1,1), (1,-1,1), (1,1,-1) and (1,-1,-1)
- 18-dop: defined by the vectors (1,0,0), (0,1,0), (0,0,1), (1,1,0), (1,0,1), (0,1,1), (1,-1,0), (1,0,-1) and (0,1,-1)
- 26-dop: defined by the vectors (1,0,0), (0,1,0), (0,0,1), (1,1,1), (1,-1,1), (1,1,-1), (1,-1,-1), (1,1,0), (1,0,1), (0,1,1), (1,-1,0), (1,0,-1) and (0,1,-1)

Building Trees of k-dops

The major issue is updating the k-dops:

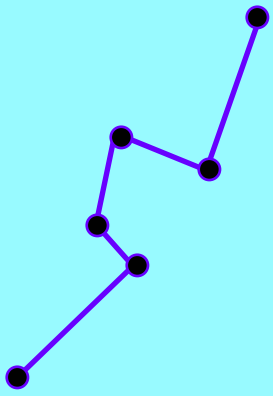
- Use Hill Climbing (as proposed in I-Collide) to update the min/max along each $k/2$ directions by comparing with the neighboring vertices
- But, the object may not be convex…… Use the approximation (convex hull vs. another k-dop)

Building an OBBTree



Recursive top-down construction:
partition and refit

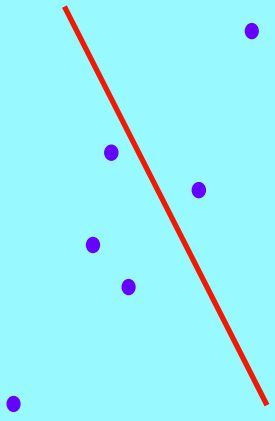
Building an OBB Tree



Given some polygons,
consider their vertices...

Building an OBB Tree

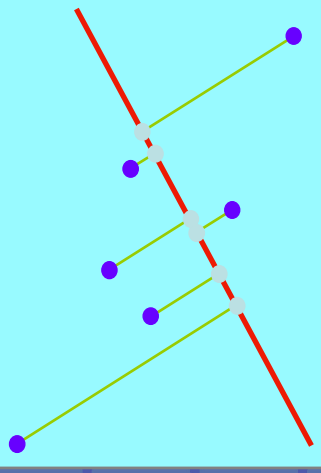
… and an arbitrary line



Building an OBB Tree

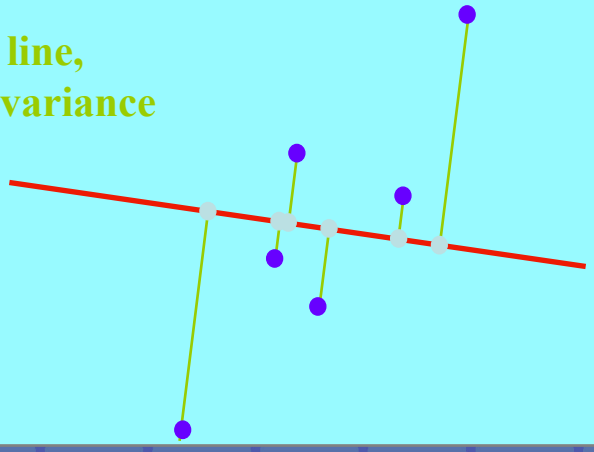
Project onto the line

Consider variance of distribution on the line



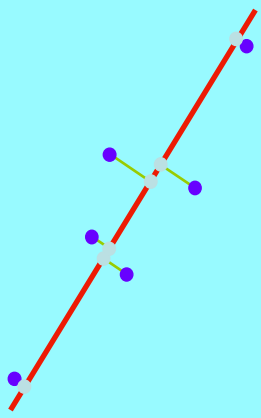
Building an OBB Tree

Different line,
different variance



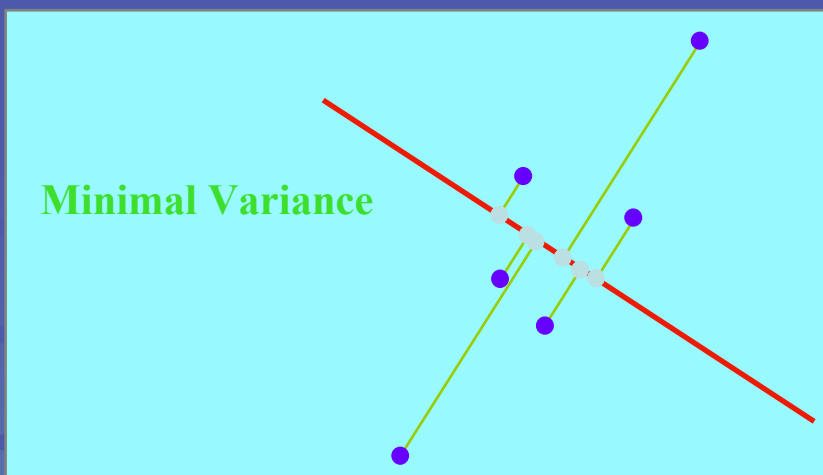
Building an OBB Tree

Maximum Variance



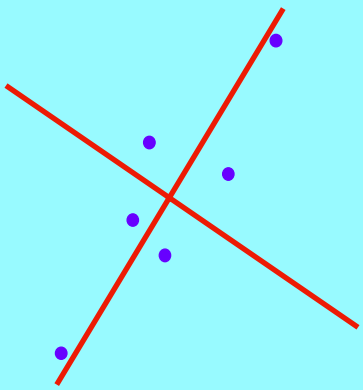
Building an OBB Tree

Minimal Variance



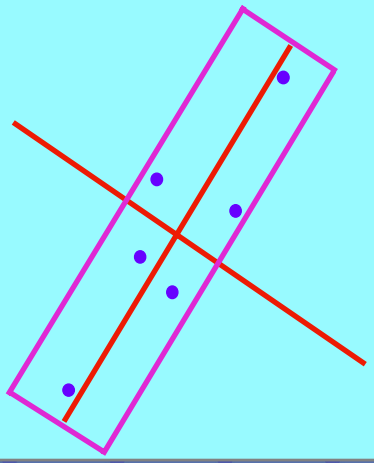
Building an OBB Tree

Given by eigenvectors
of covariance matrix
of coordinates
of original points



Building an OBB Tree

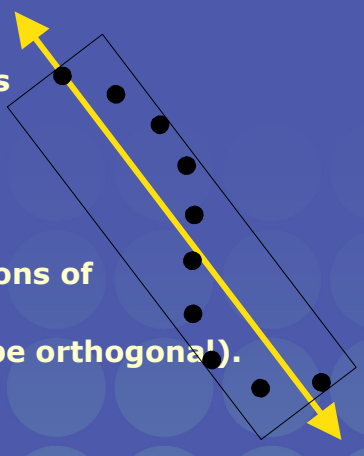
Choose bounding box oriented this way



Building an OBB Tree: Fitting

Covariance matrix of point coordinates describes statistical spread of cloud.

OBB is aligned with directions of greatest and least spread (which are guaranteed to be orthogonal).



Fitting OBBs

- Let the vertices of the i'th triangle be the points a^i, b^i, and c^i, then the mean μ and covariance matrix C can be expressed in vector notation as:

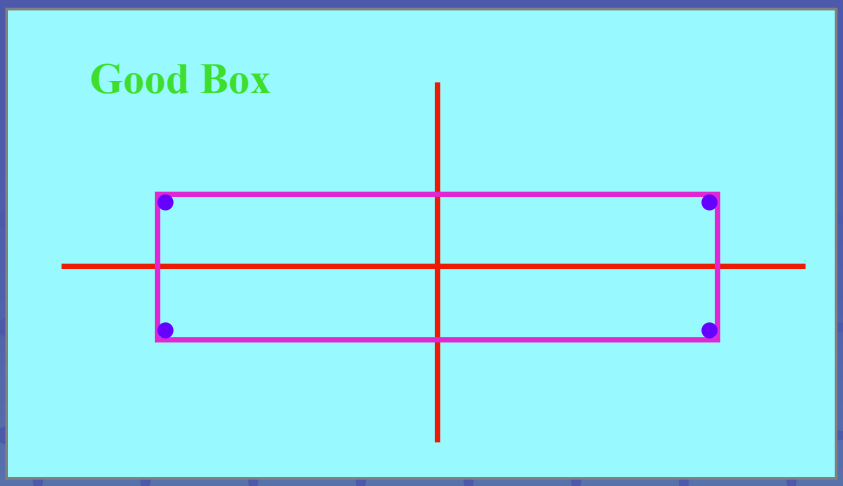
$$\mu = \frac{1}{3n}\sum_{i=0}^{n}(\mathbf{a}^i + \mathbf{b}^i + \mathbf{c}^i),$$

$$\mathbf{C}_{jk} = \frac{1}{3n}\sum_{i=0}^{n}(\overline{\mathbf{a}}_j^i\overline{\mathbf{a}}_k^i + \overline{\mathbf{b}}_j^i\overline{\mathbf{b}}_k^i + \overline{\mathbf{c}}_j^i\overline{\mathbf{c}}_k^i), \qquad 1 \le j,k \le 3$$

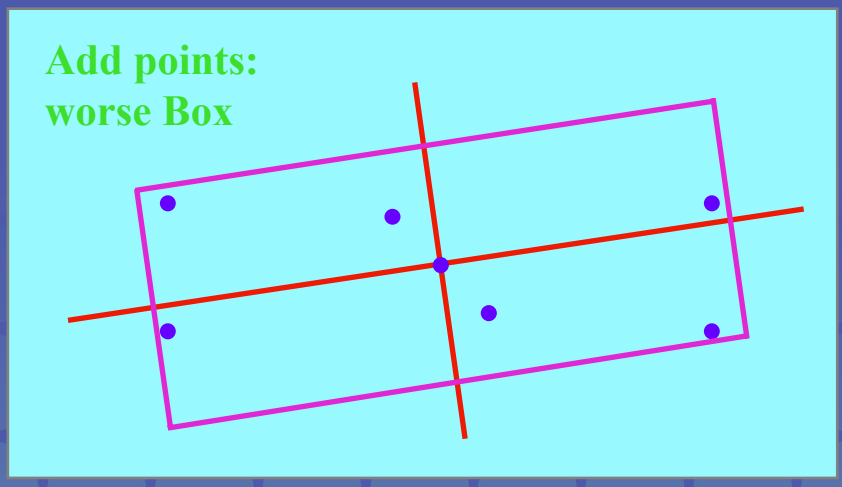
where n is the number of triangles, and

$$\overline{\mathbf{a}}^i = \mathbf{a}^i - \mu, \quad \overline{\mathbf{b}}^i = \mathbf{b}^i - \mu, \quad \overline{\mathbf{c}}^i = \mathbf{c}^i - \mu.$$

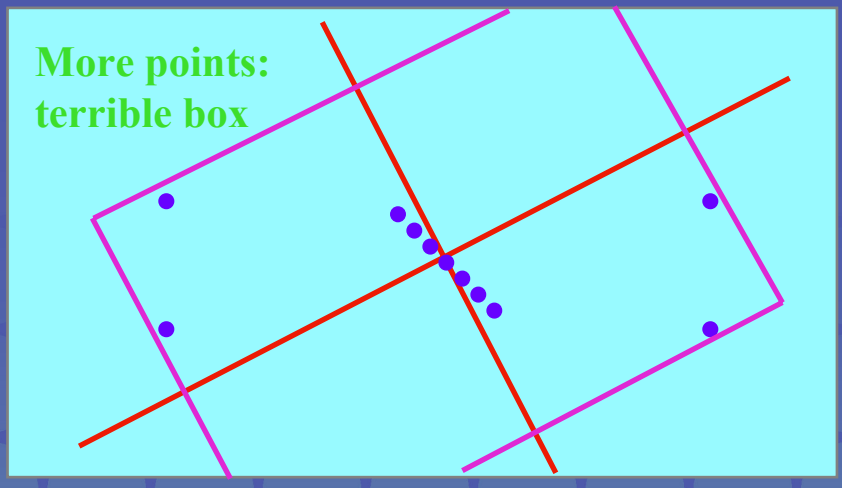
Building an OBB Tree



Building an OBB Tree

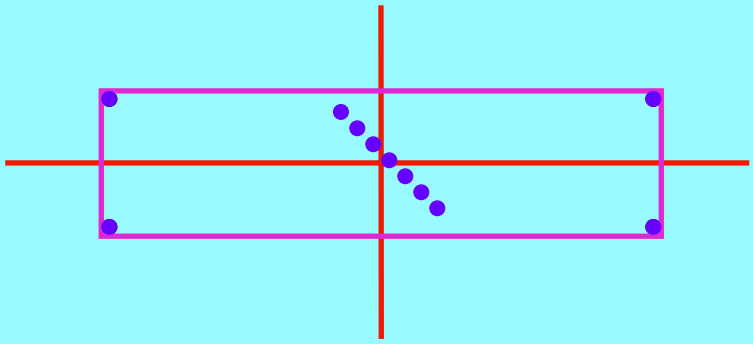


Building an OBB Tree



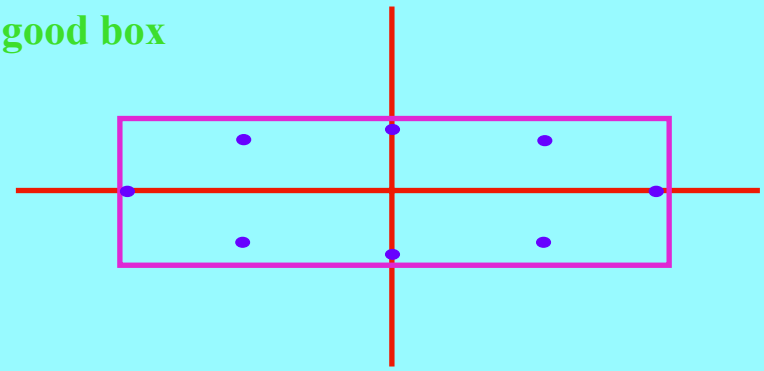
Building an OBB Tree

Compute with extremal points only



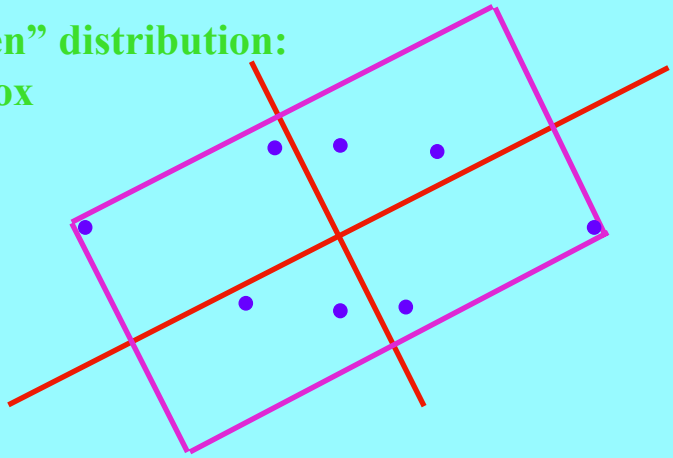
Building an OBB Tree

"Even" distribution:
good box



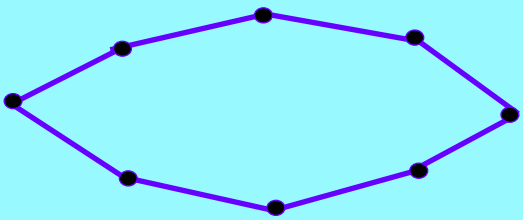
Building an OBB Tree

"Uneven" distribution:
bad box

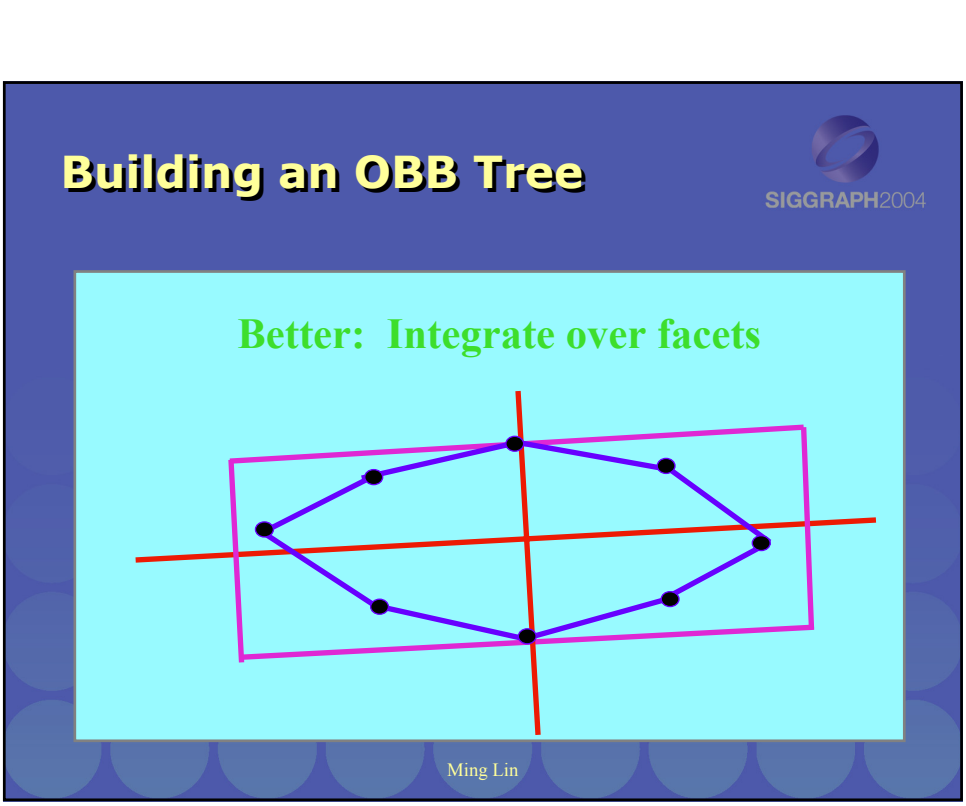


Building an OBB Tree

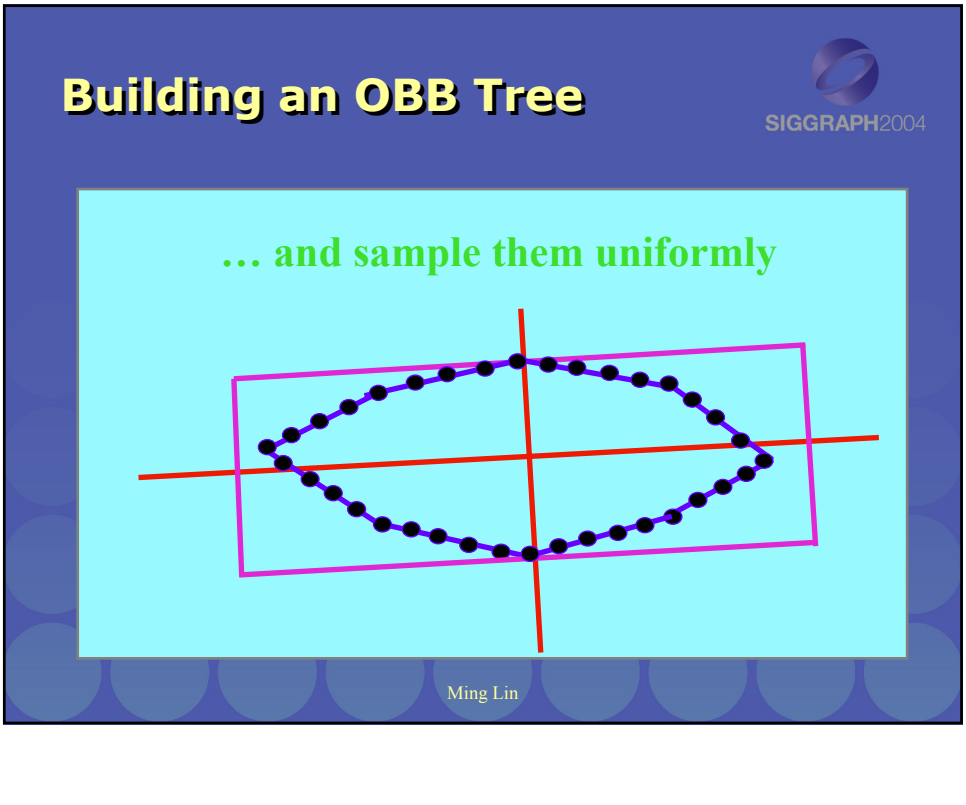
Fix: Compute facets of convex hull...



Building an OBB Tree



Building an OBB Tree



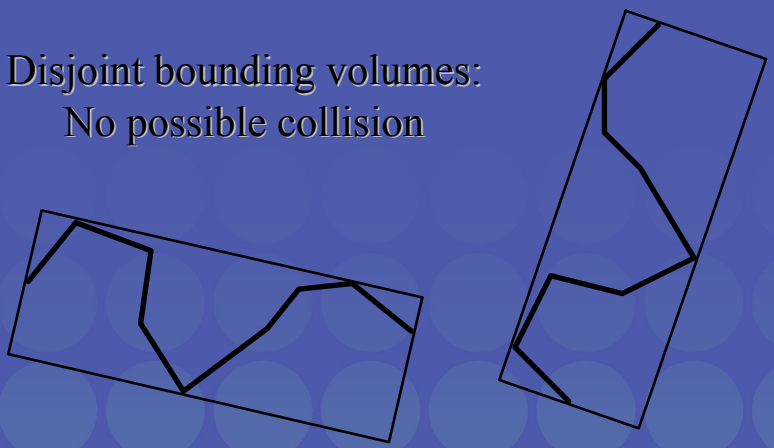
Building an OBB Tree: Summary

OBB Fitting algorithm:

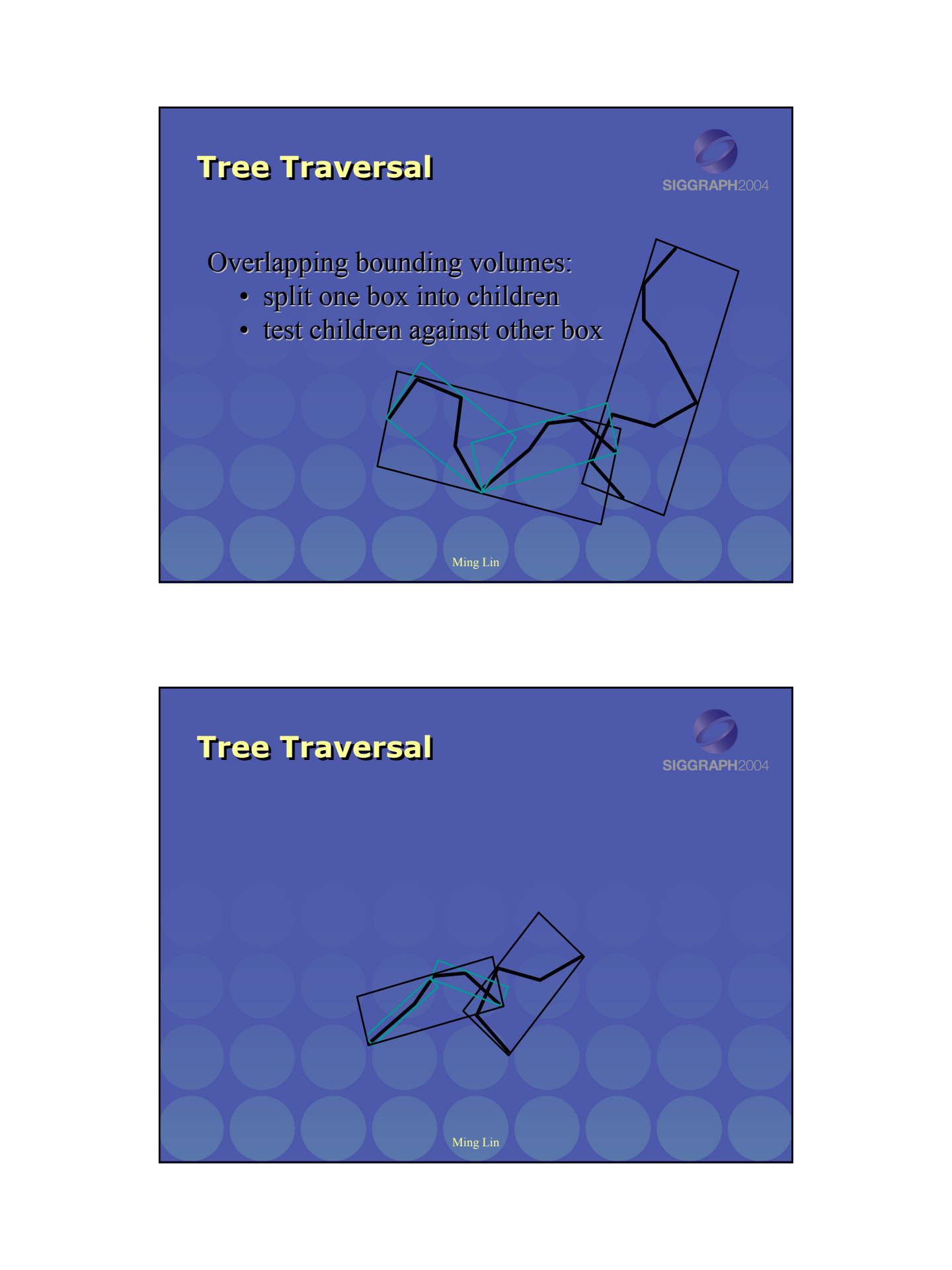
- covariance-based
- use of convex hull
- not foiled by extreme distributions
- O(n log n) fitting time for single BV
- O(n $\log^2$ n) fitting time for entire tree

Tree Traversal

Disjoint bounding volumes:
No possible collision



Tree Traversal



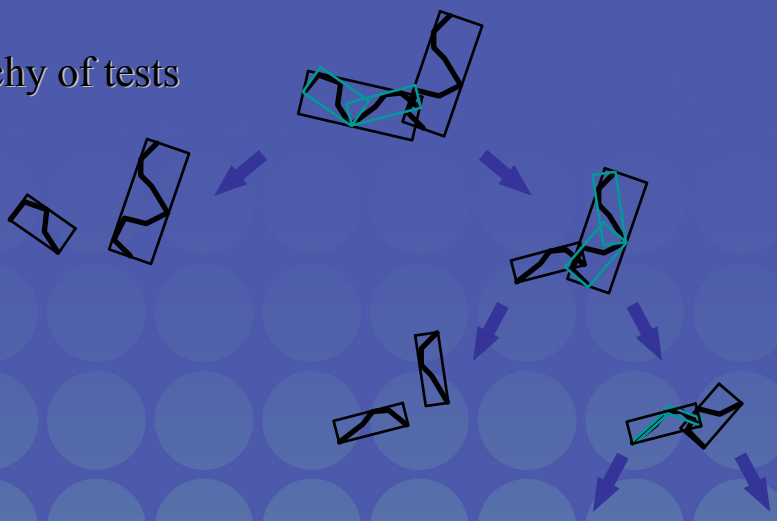
Overlapping bounding volumes:

- split one box into children
- test children against other box

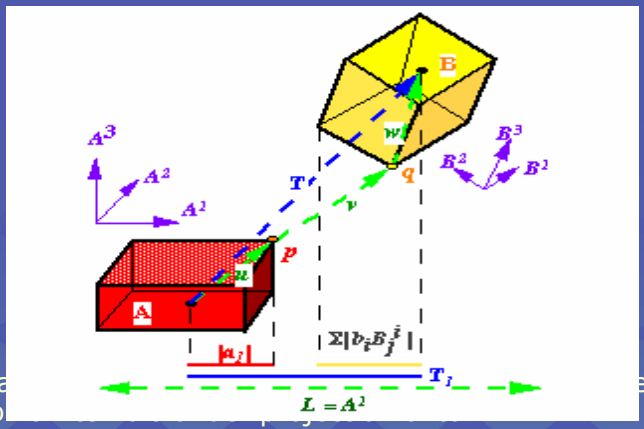
Tree Traversal

Tree Traversal

Hierarchy of tests



Separating Axis Theorem



- L is a … ecome disjo…

Separating Axis Theorem

Two polytopes A and B are disjoint iff there exists a separating axis which is:

perpendicular to a face from either

or

perpedicular to an edge from each

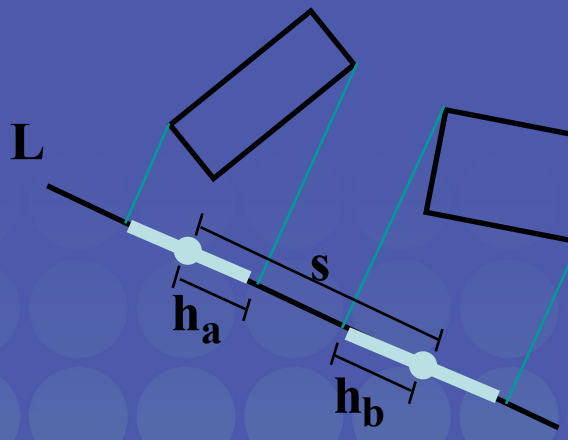
Implications of Theorem

- Given two generic polytopes, each with E edges and F faces, number of candidate axes to test is:

$$2F + E^2$$

- OBBs have only E = 3 distinct edge directions, and only F = 3 distinct face normals. OBBs need at most 15 axis tests.
- Because edge directions and normals each form orthogonal frames, the axis tests are rather simple.

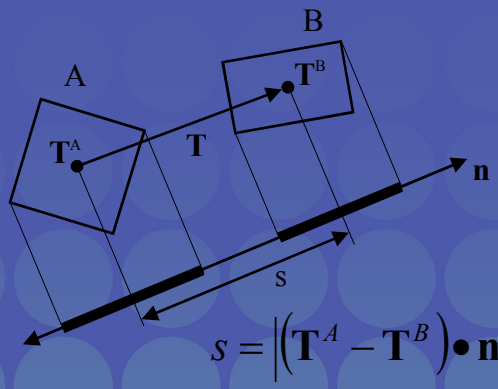
OBB Overlap Test: An Axis Test



L is a separating axis iff: $s > h_a + h_b$

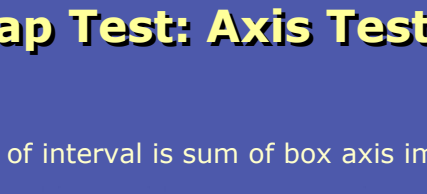
OBB Overlap Test: Axis Test Details

Box centers project to interval midpoints, so midpoint separation is length of vector **T**'s image.

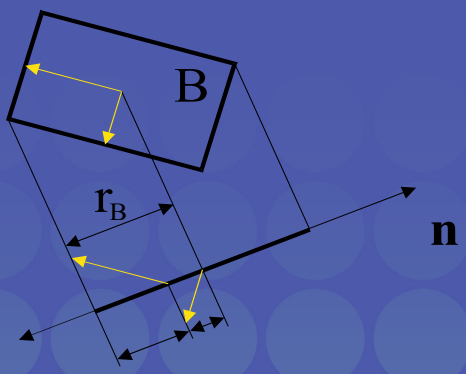


$$s = \left| \left(\mathbf{T}^A - \mathbf{T}^B \right) \bullet \mathbf{n} \right|$$

OBB Overlap Test: Axis Test Details



- Half-length of interval is sum of box axis images.



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

OBB Overlap Test

- Typical axis test for 3-space.

```
s = fabs(T2 * R11  -  T1 * R21);
ha = a1 * Rf21  +  a2 * Rf11;
hb = b0 * Rf02  +  b2 * Rf00;
if (s > (ha + hb)) return 0;
```

- Up to 15 tests required.

OBB Overlap Test

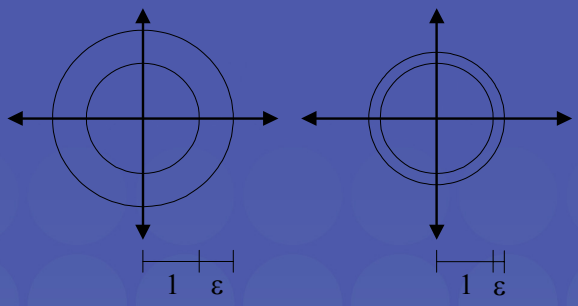
- Strengths of this overlap test:
  - 89 to 252 arithmetic operations per box overlap test
  - Simple guard against arithmetic error
  - No special cases for parallel/coincident faces, edges, or vertices
  - No special cases for degenerate boxes
  - No conditioning problems
  - Good candidate for micro-coding

OBB Overlap Tests: Comparison

Test Method	Speed(us)
Separating Axis	6.26
GJK	66.30
LP	217.00

Benchmarks performed on SGI Max Impact, 250 MHz MIPS R4400 CPU, MIPS R4000 FPU

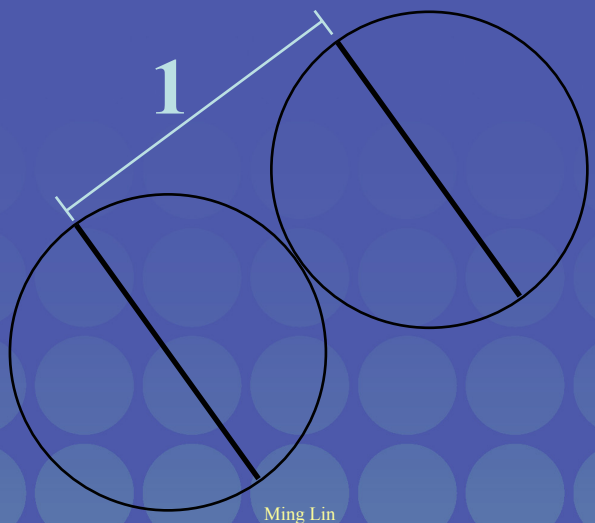
Parallel Close Proximity



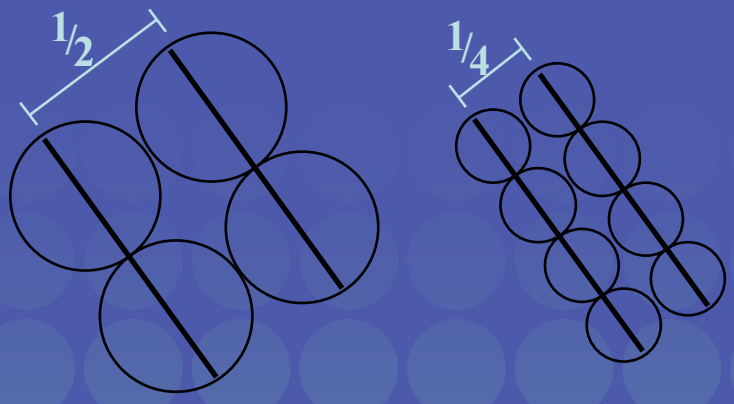
Two models are in *parallel close proximity* when every point on each model is a given fixed distance (ε) from the other model.

Q: How does the number of BV tests increase as the gap size decreases?

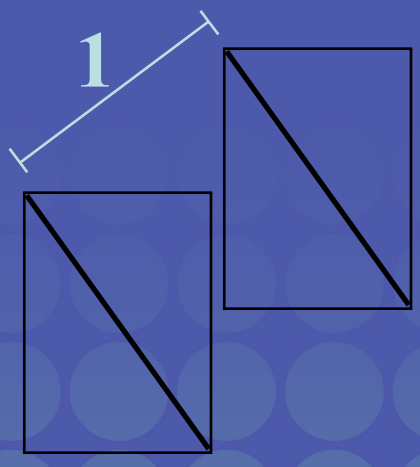
Parallel Close Proximity: Convergence



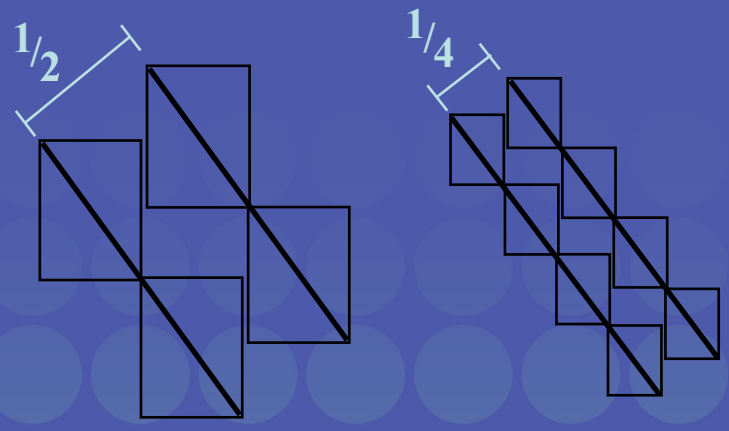
Parallel Close Proximity: Convergence



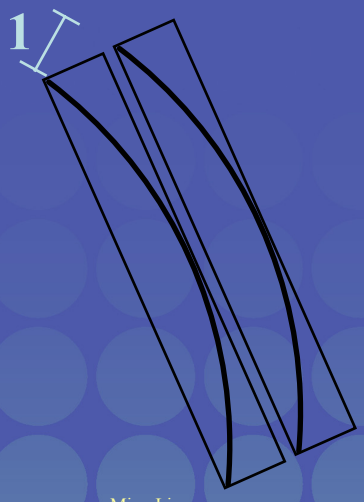
Parallel Close Proximity: Convergence



Parallel Close Proximity: Convergence

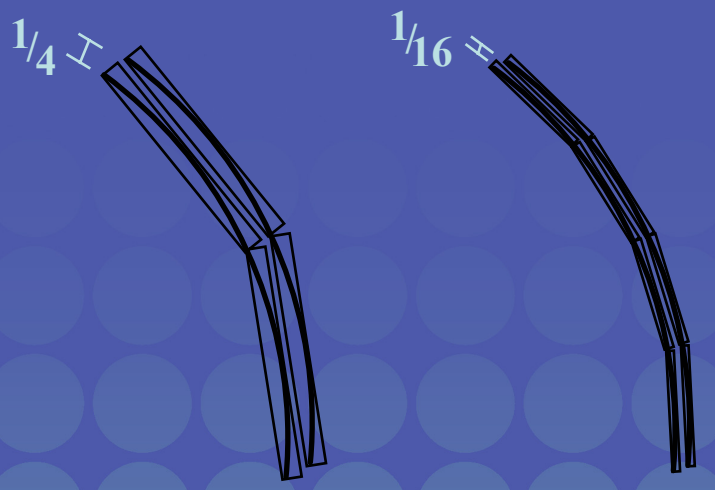


Parallel Close Proximity: Convergence



Parallel Close Proximity: Convergence

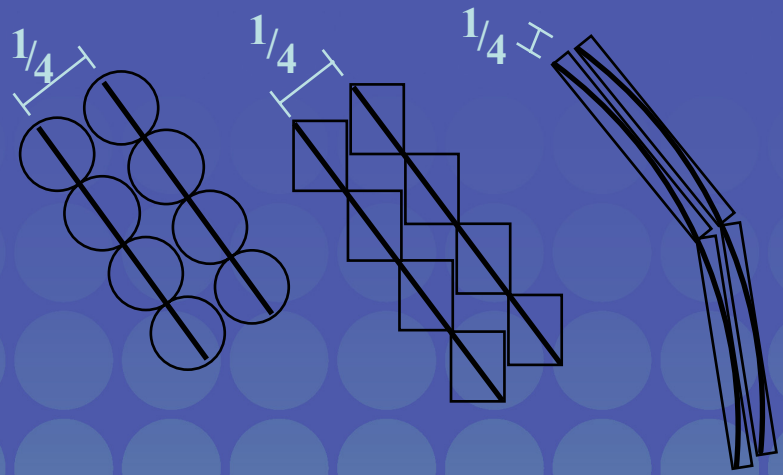
SIGGRAPH2004



Ming Lin

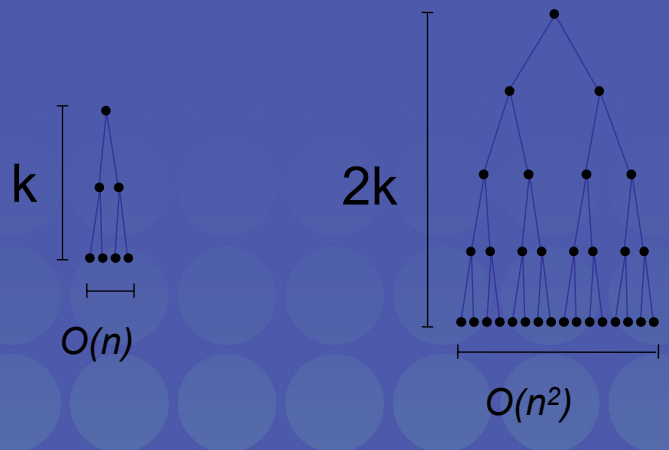
Parallel Close Proximity: Convergence

SIGGRAPH2004



Ming Lin

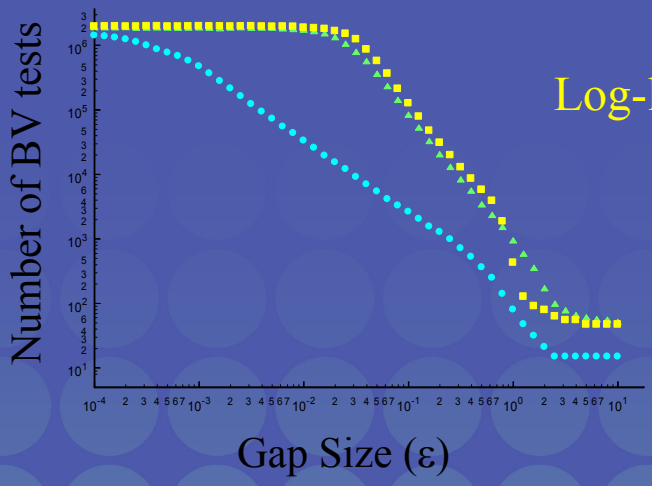
Performance: Overlap Tests



OBBs

Spheres & AABBs

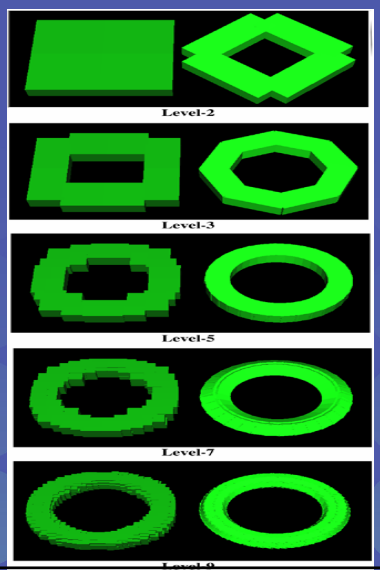
Parallel Close Proximity: Experiment



Log-log plot

OBBs asymptotically outperform AABBs and spheres

Example: AABB's vs. OBB's

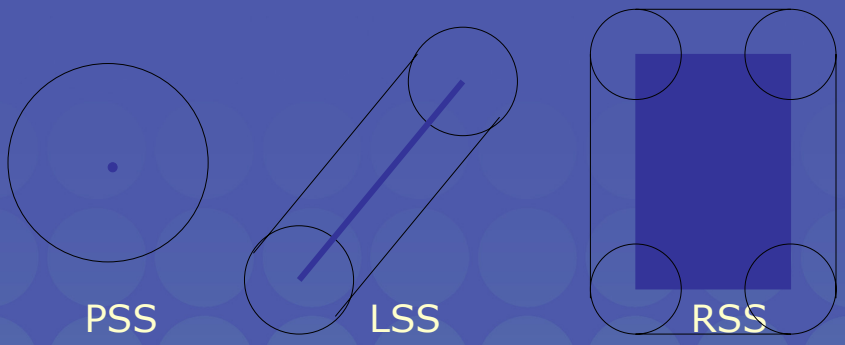


Approximation of a Torus

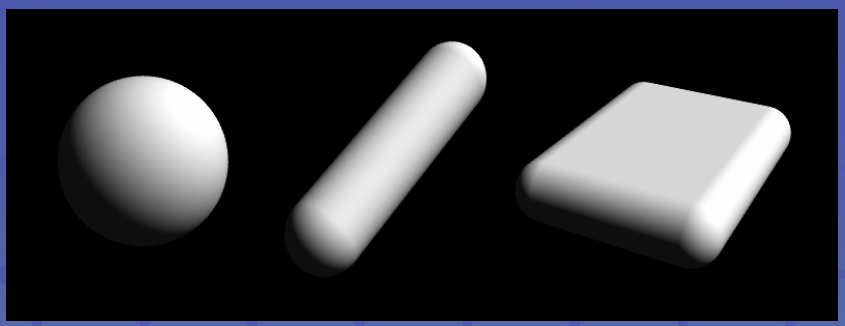
Implementation: RAPID

- Available at: http://www.cs.unc.edu/~geom/OBB
- Part of V-COLLIDE: http://www.cs.unc.edu/~geom/V_COLLIDE
- Thousands of users have ftp'ed the code
- Used for virtual prototyping, dynamic simulation, robotics & computer animation

Hybrid Hierarchy of Swept Sphere Volumes



Swept Sphere Volumes (S-topes)



SSV Fitting

- Use OBB's code based upon *Principle Component Analysis*
- For PSS, use the largest dimension as the radius
- For LSS, use the two largest dimensions as the length and radius
- For RSS, use all three dimensions

Overlap Test

- One routine that can perform overlap tests between all possible combination of CORE primitives of SSV(s).
- The routine is a specialized test based on Voronoi regions and OBB overlap test.
- It is faster than GJK.

Hybrid BVH's Based on SSVs

- Use a simpler BV when it prunes search equally well - benefit from lower cost of BV overlap tests
- Overlap test (based on Lin-Canny & OBB overlap test) between all pairs of BV's in a BV family is unified
- Complications
  - deciding which BV to use either *dynamically* or *statically*

PQP: Implementation

- Library written in C++
- Good for any proximity query
- 5-20x speed-up in *distance computation* over prior methods
- Available at *http://www.cs.unc.edu/~geom/SSV/*

References

- *OBB-Tree: A Hierarchical Structure for Rapid Interference Detection*, by S. Gottschalk, M. Lin and D. Manocha, Proc. of ACM Siggraph, 1996.
- *Rapid and Accurate Contact Determination between Spline Models using ShellTrees*, by S. Krishnan, M. Gopi, M. Lin, D. Manocha and A. Pattekar, Proc. of Eurographics 1998.
- *Fast Proximity Queries with Swept Sphere Volumes*, by Eric Larsen, Stefan Gottschalk, Ming C. Lin, Dinesh Manocha, Technical report TR99-018, UNC-CH, CS Dept, 1999. (Part of the paper in Proc. of IEEE ICRA'2000)

Fast Proximity Queries Between Convex Polyhedra

Ming C. Lin
lin@cs.unc.edu
http://www.cs.unc.edu/~lin
http://gamma.cs.unc.edu/
University of North Carolina at Chapel Hill

Voronoi Diagrams

- Given a set *S* of *n* points in R^2, for each point p_i in *S*, there is the set of points *(x, y)* in the plane that are closer to p_i than any other point in *S*, called Voronoi polygons. The collection of *n* Voronoi polygons given the *n* points in the set *S* is the *"Voronoi diagram"*, ***Vor(S)***, of the point set *S*.

Intuition: **To partition the plane into regions, each of these is the set of points that are closer to a point p_i in *S* than any other. The partition is based on the set of closest points, e.g. bisectors that have 2 or 3 closest points.**

Generalized Voronoi Diagrams

- The extension of the Voronoi diagram to higher dimensional features (such as edges and facets, instead of points)**;** i.e. the set of points closest to a *feature*, e.g. that of a polyhedron.

- FACTS:
  - **In general, the generalized Voronoi diagram has quadratic surface boundaries in it.**
  - **If the polyhedron is convex, then its generalized Voronoi diagram has planar boundaries.**

Voronoi Regions

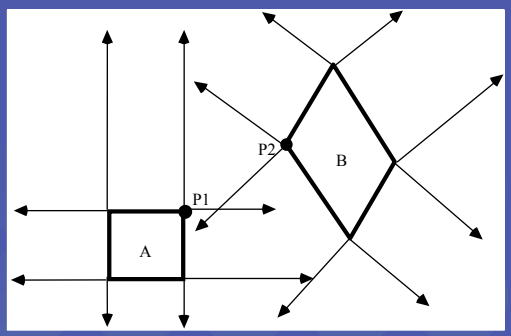
- A *Voronoi region* **associated with a** *feature* **is a set of points that are closer to that feature than any other.**
- FACTS:
  - **The Voronoi regions form a partition of space outside of the polyhedron according to the closest feature.**
  - **The collection of Voronoi regions of each polyhedron is the generalized Voronoi diagram of the polyhedron.**
  - **The generalized Voronoi diagram of a convex polyhedron has linear size and consists of polyhedral regions. And, all Voronoi regions are convex.**

Voronoi Marching

Basic Ideas:

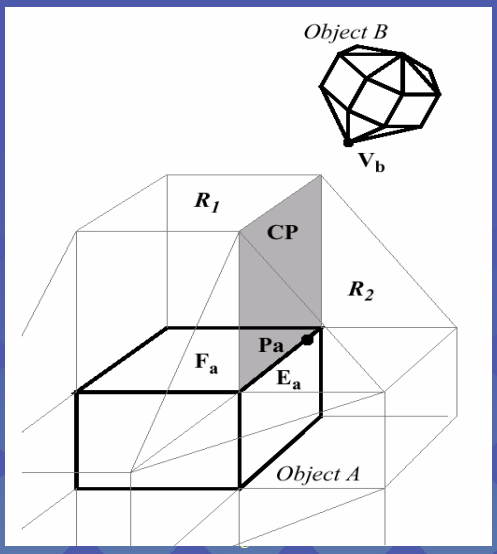
- Coherence: **local geometry does not change much, when computations repetitively performed over successive small time intervals**
- Locality: **to "*track*" the pair of closest features between 2 moving convex polygons(polyhedra) w/ Voronoi regions**
- Performance: **expected** constant **running time, independent of the geometric complexity**

Simple 2D Example



Objects A & B and their Voronoi regions: P1 and P2 are the pair of closest points between A and B. Note P1 and P2 lie within the Voronoi regions of each other.

Basic Idea for Voronoi Marching



Large, Dynamic Environments

- For dynamic simulation where the velocity and acceleration of all objects are known at each step, use the scheduling scheme (implemented as heap) to prioritize "critical events" to be processed.
- Each object pair is tagged with the estimated time to next collision. Then, each pair of objects is processed accordingly. The heap is updated when a collision occurs.

Scheduling Scheme

- a_{max}: an upper bound on relative acceleration between any two *points* on any pair of objects.
- a_{lin}: relative absolute linear
- α: relative rotational accelerations
- ω: relative rotational velocities
- r: vector difference btw CoM of two bodies
- d: initial separation for two given objects

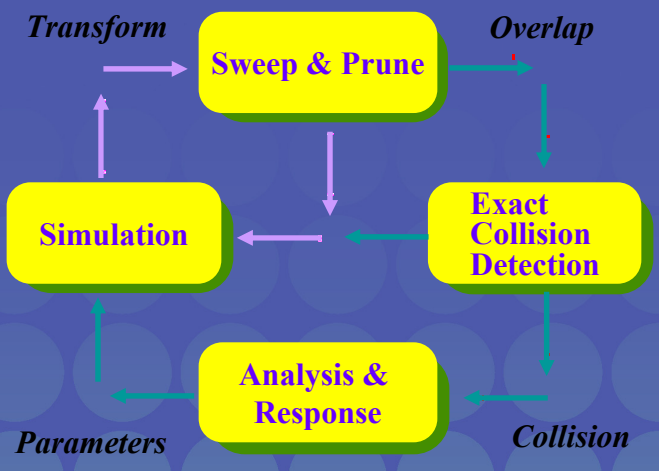
$$a_{max} = | a_{lin} + \alpha \times r + \omega \times \omega \times r |$$

$$v_i = | v_{lin} + \omega \times r |$$

- Estimated Time to collision:

$$t_c = \{ (v_i^2 + 2 a_{max} d)^{1/2} - v_i \} / a_{max}$$

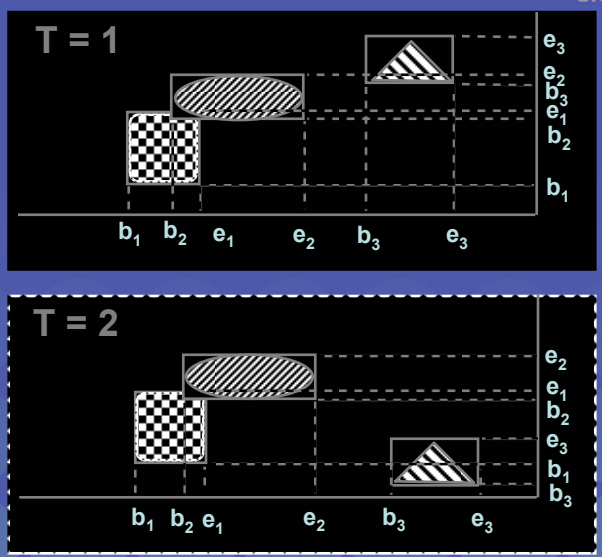
Collide System Architecture



Sweep and Prune

- Compute the axis-aligned bounding box (fixed vs. dynamic) for each object
- Dimension Reduction by projecting boxes onto each *x, y, z*- axis
- Sort the endpoints and find overlapping intervals
- Possible collision -- only if projected intervals overlap in all 3 dimensions

Sweep & Prune



Updating Bounding Boxes

- Coherence **(greedy algorithm)**
- Convexity properties **(geometric properties of convex polytopes)**
- Nearly constant time**, if the motion is relatively "small"**

Use of Sorting Methods

- Initial sort -- quick sort runs in $O(m \log m)$ just as in any ordinary situation
- Updating -- insertion sort runs in $O(m)$ due to coherence. We sort an almost sorted list from last stimulation step. In fact, we look for "swap" of positions in all 3 dimension.

Implementation Issues

- Collision matrix -- basically *adjacency matrix*
- Enlarge bounding volumes with some tolerance threshold
- Quick start polyhedral collision test -- using bucket sort & look-up table

References

- *Collision Detection between Geometric Models: A Survey*, by M. Lin and S. Gottschalk, Proc. of IMA Conference on Mathematics of Surfaces 1998.
- *I-COLLIDE: Interactive and Exact Collision Detection for Large-Scale Environments*, by Cohen, Lin, Manocha & Ponamgi, Proc. of ACM Symposium on Interactive 3D Graphics, 1995.
(More details in Chapter 3 of M. Lin's Thesis)

The Gilbert-Johnson-Keerthi (GJK) Algorithm

Christer Ericson
Sony Computer Entertainment America

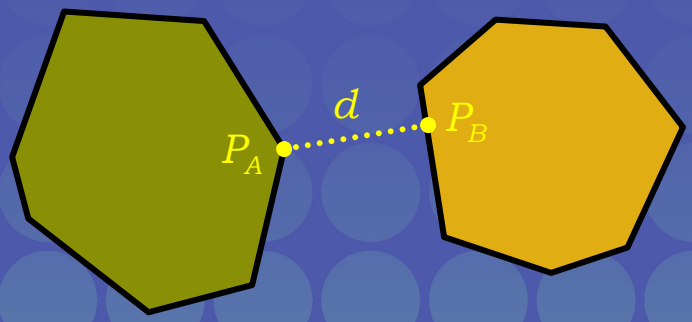
christer_ericson@playstation.sony.com

Talk outline

- What is the GJK algorithm
- Terminology
- "Simplified" version of the algorithm
  - One object is a point at the origin
  - Example illustrating algorithm
- The distance subalgorithm
- GJK for two objects
  - One no longer necessarily a point at the origin
- GJK for moving objects

GJK solves proximity queries

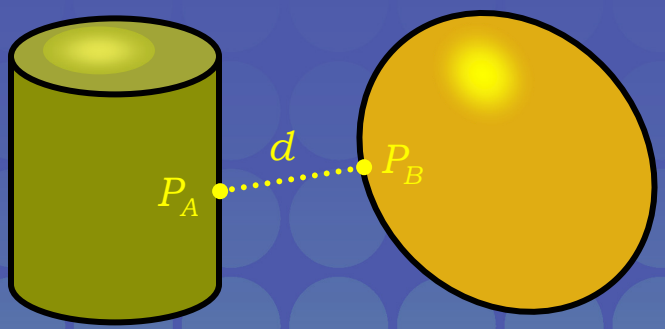
- Given two convex polyhedra
  - Computes distance *d*
  - Can also return closest pair of points P_A, P_B



Christer Ericson

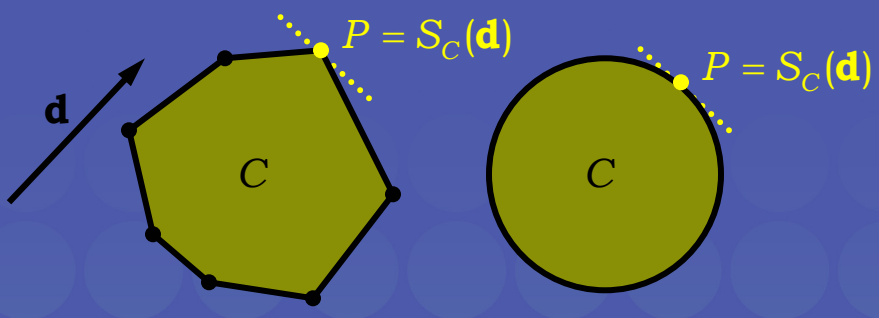
GJK solves proximity queries

- Generalized for arbitrary convex objects
  - As long as they can be described in terms of a *support mapping* function



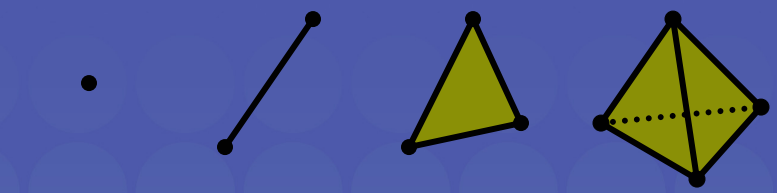
Christer Ericson

Terminology 1(3)



Supporting (or extreme) point P for direction $\mathbf{d}$ returned by support mapping function $S_C(\mathbf{d})$

Terminology 2(3)



0-simplex 1-simplex 2-simplex 3-simplex

simplex

Terminology 3(3)

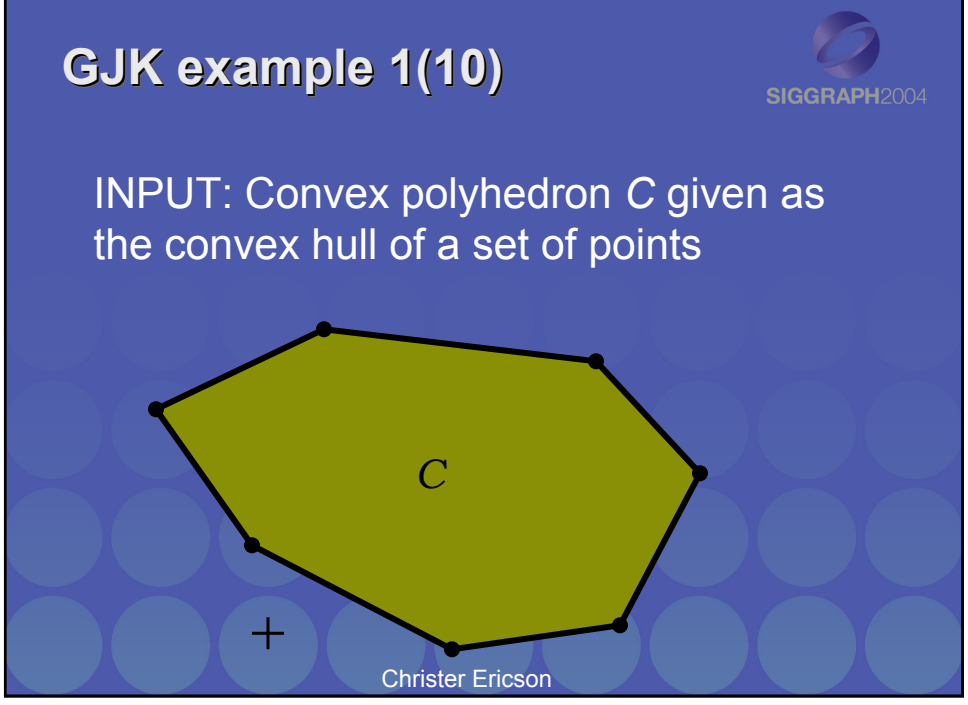
Point set *C*

Convex hull, CH(*C*)

The GJK algorithm

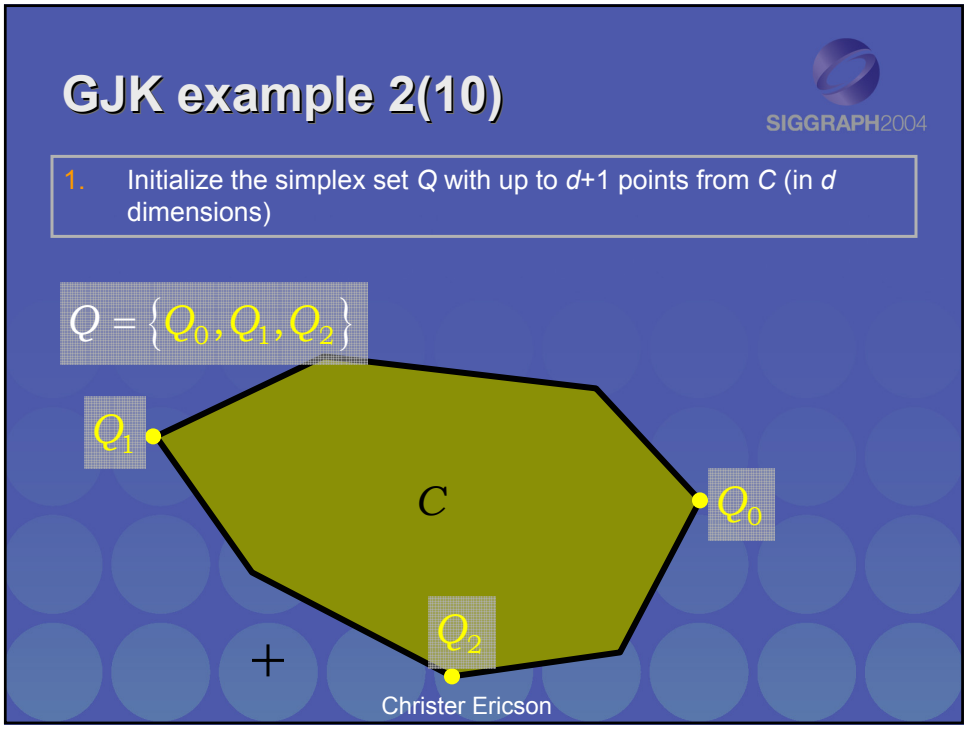
1. Initialize the simplex set *Q* with up to *d*+1 points from *C* (in *d* dimensions)
2. Compute point *P* of minimum norm in CH(*Q*)
3. If *P* is the origin, exit; return 0
4. Reduce *Q* to the smallest subset *Q*' of *Q*, such that *P* in CH(*Q*')
5. Let $V=S_C(-P)$ be a supporting point in direction $-P$
6. If *V* no more extreme in direction $-P$ than *P* itself, exit; return $||P||$
7. Add *V* to *Q*. Go to step 2

GJK example 1(10)



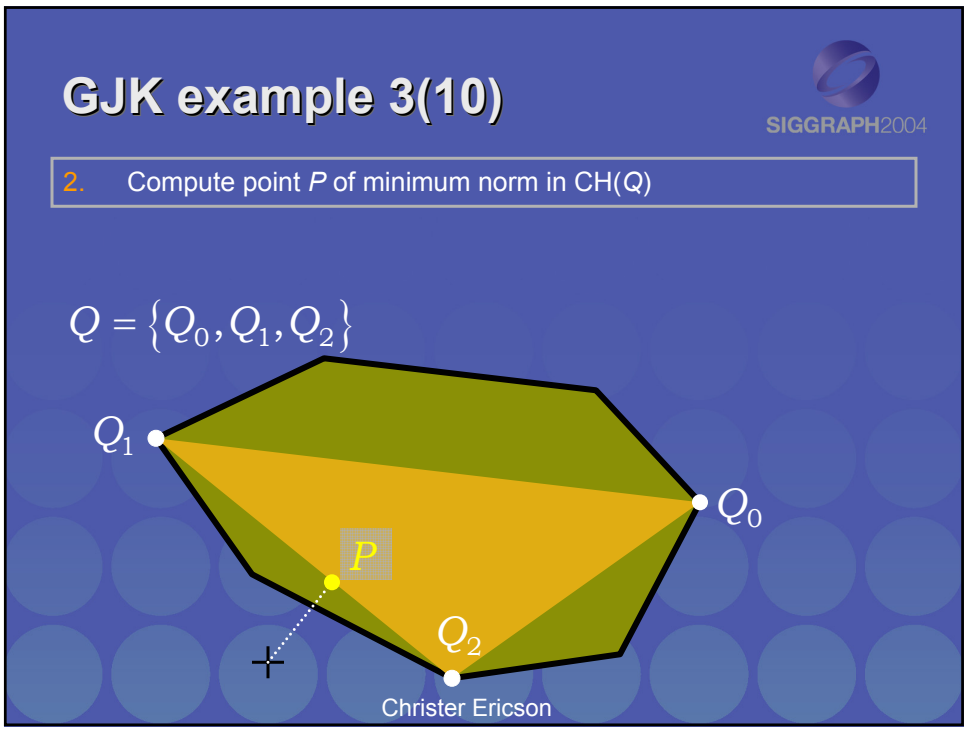
INPUT: Convex polyhedron *C* given as the convex hull of a set of points

GJK example 2(10)

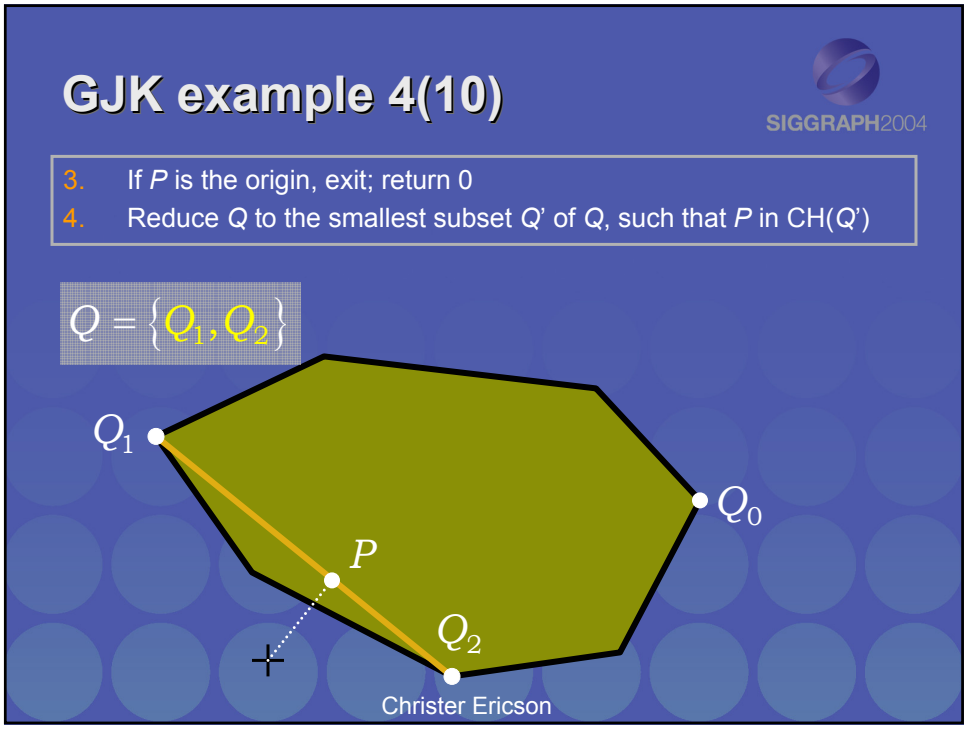


1. Initialize the simplex set *Q* with up to *d*+1 points from *C* (in *d* dimensions)

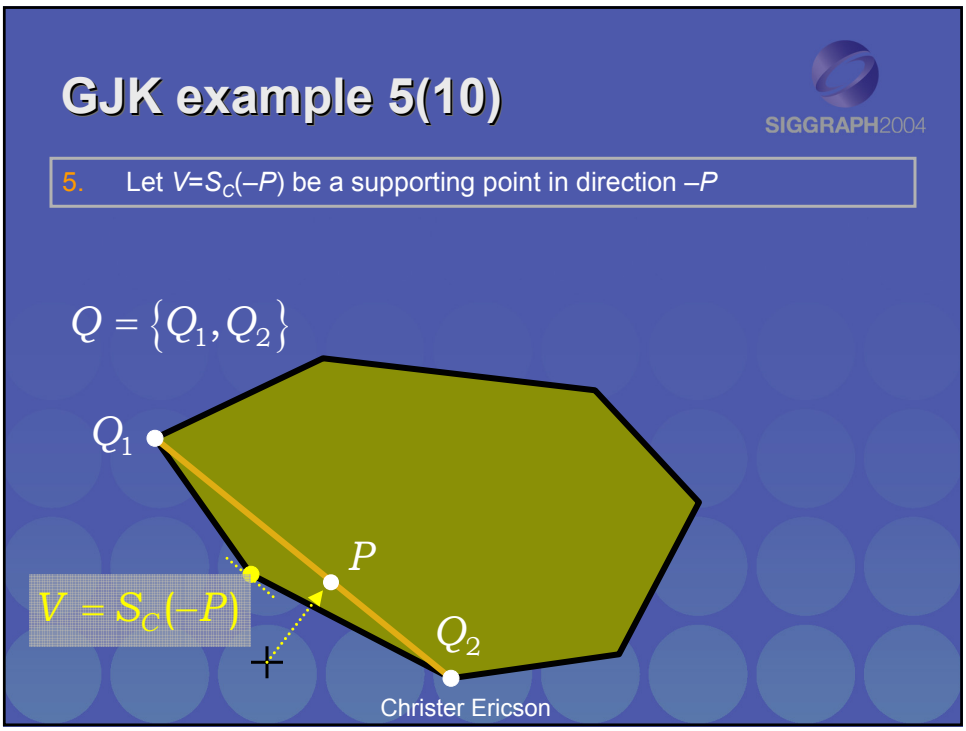
GJK example 3(10)



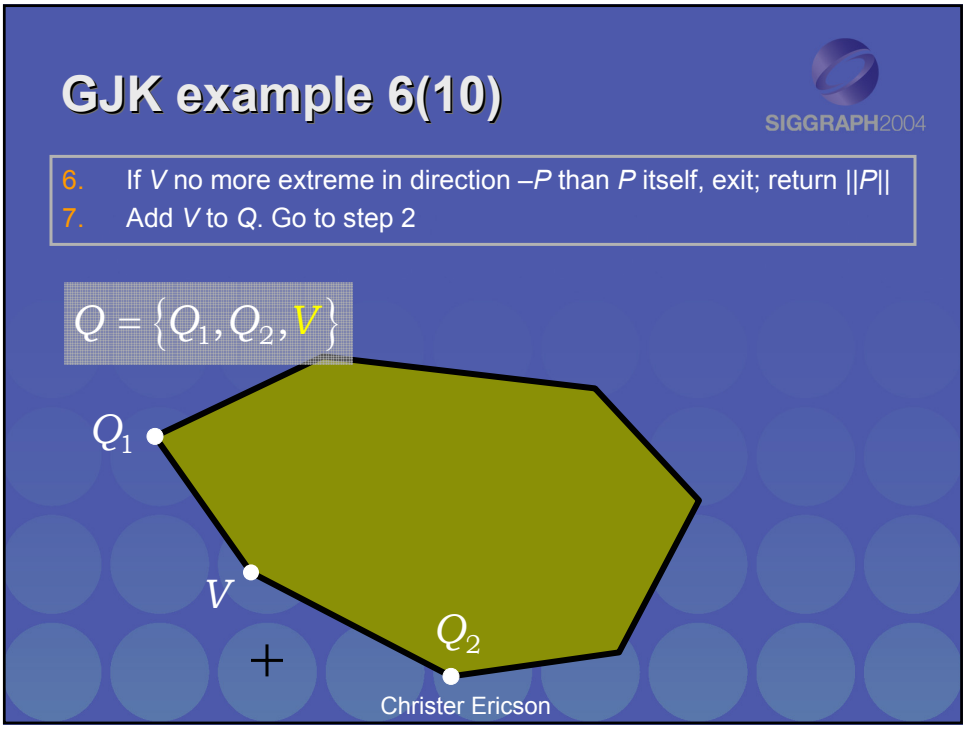
GJK example 4(10)



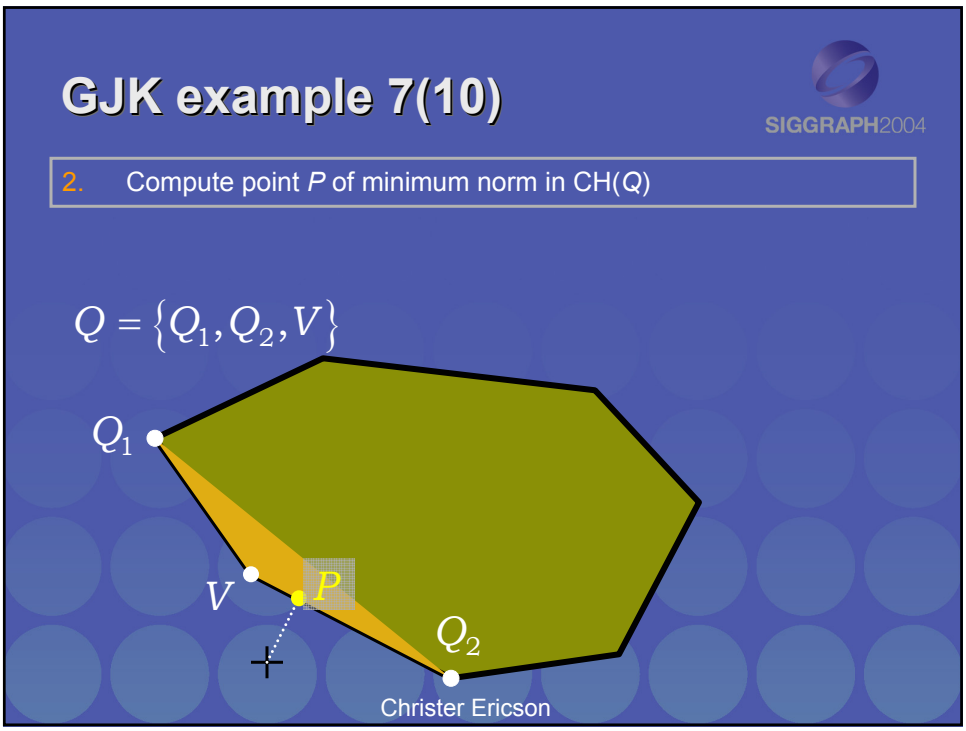
GJK example 5(10)



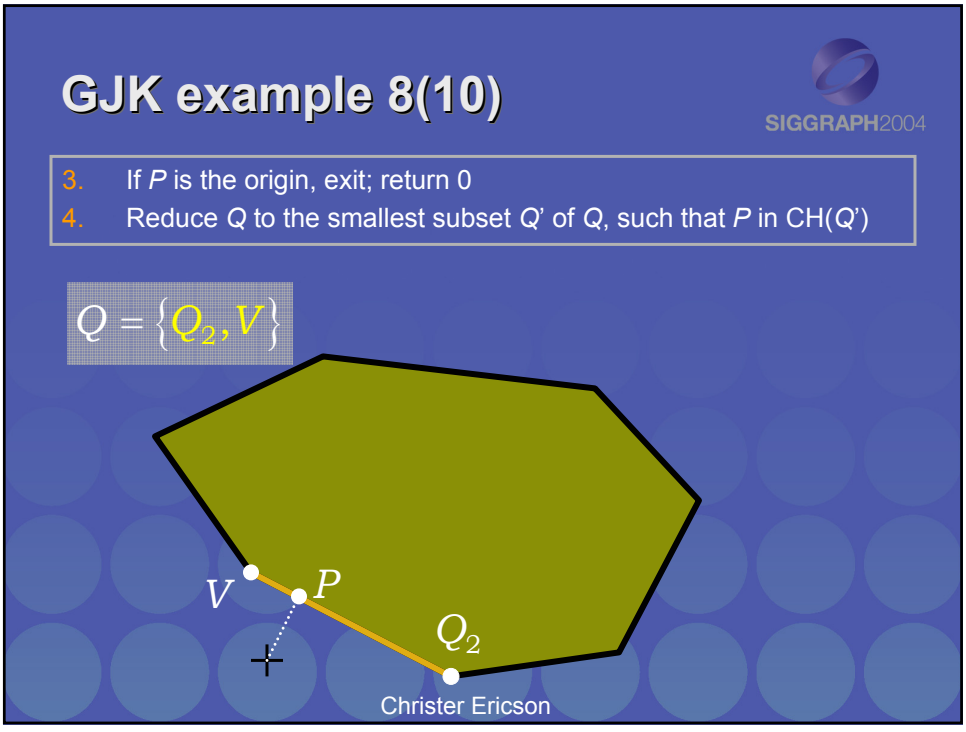
GJK example 6(10)



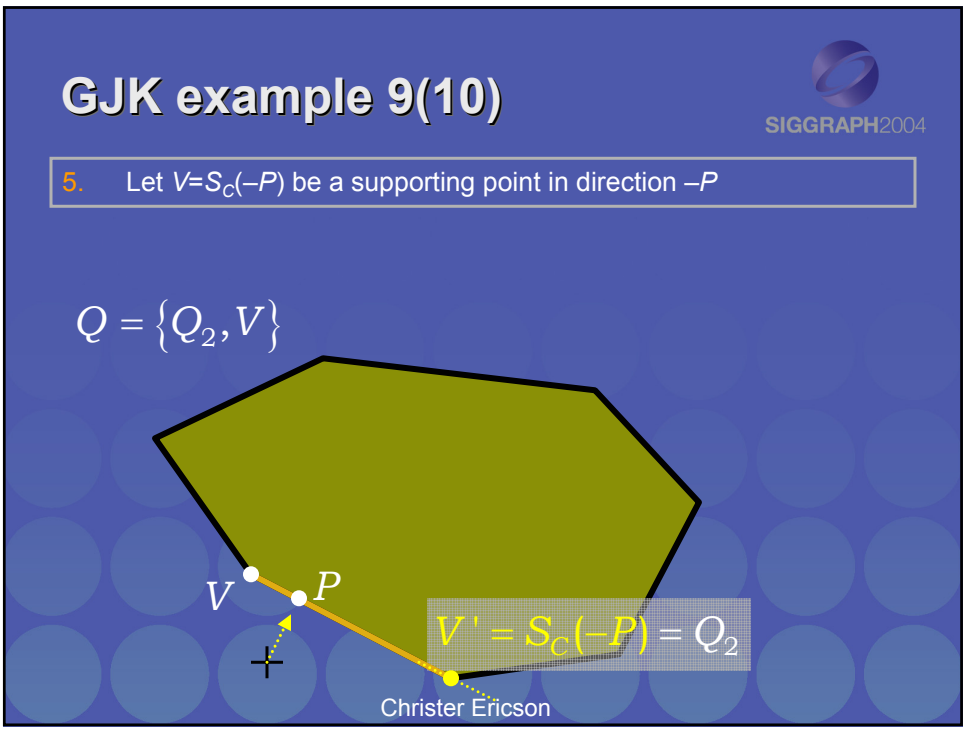
GJK example 7(10)



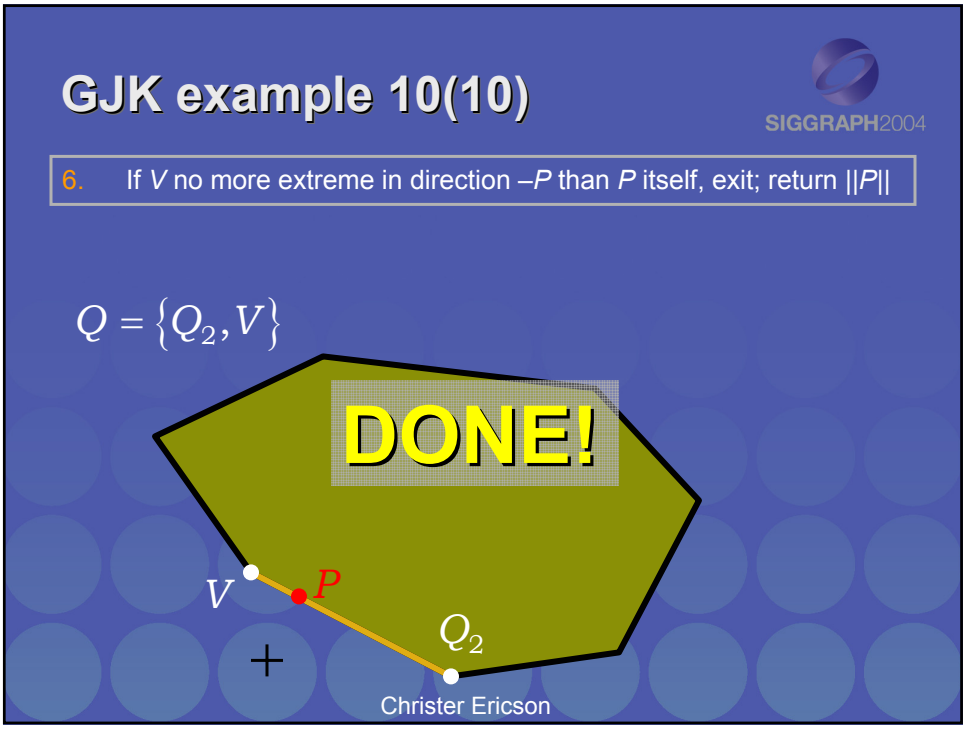
GJK example 8(10)



GJK example 9(10)



GJK example 10(10)



Distance subalgorithm 1(2)

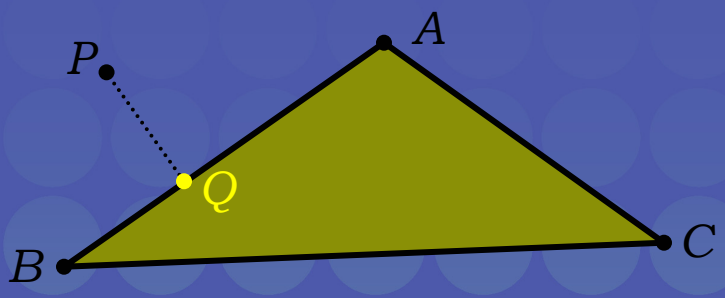
- Approach #1: Solve algebraically
  - Used in original GJK paper
  - Johnson's distance subalgorithm
    - Searches all simplex subsets
    - Solves system of linear equations for each subset
    - Recursive formulation
    - From era when math operations were expensive
    - Robustness problems
  - See e.g. Gino van den Bergen's book

Distance subalgorithm 2(2)

- Approach #2: Solve geometrically
  - Mathematically equivalent
    - But more intuitive
    - Therefore easier to make robust
  - Use straightforward primitives:
    - `ClosestPointOnEdgeToPoint()`
    - `ClosestPointOnTriangleToPoint()`
    - `ClosestPointOnTetrahedronToPoint()`
  - Second function outlined here
    - The approach generalizes

Closest point on triangle

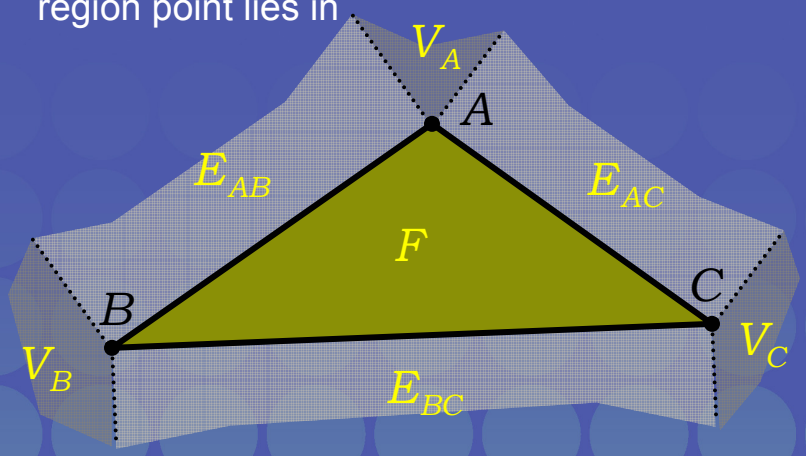
- ClosestPointOnTriangleToPoint()
  - Finds point on triangle closest to a given point



Christer Ericson

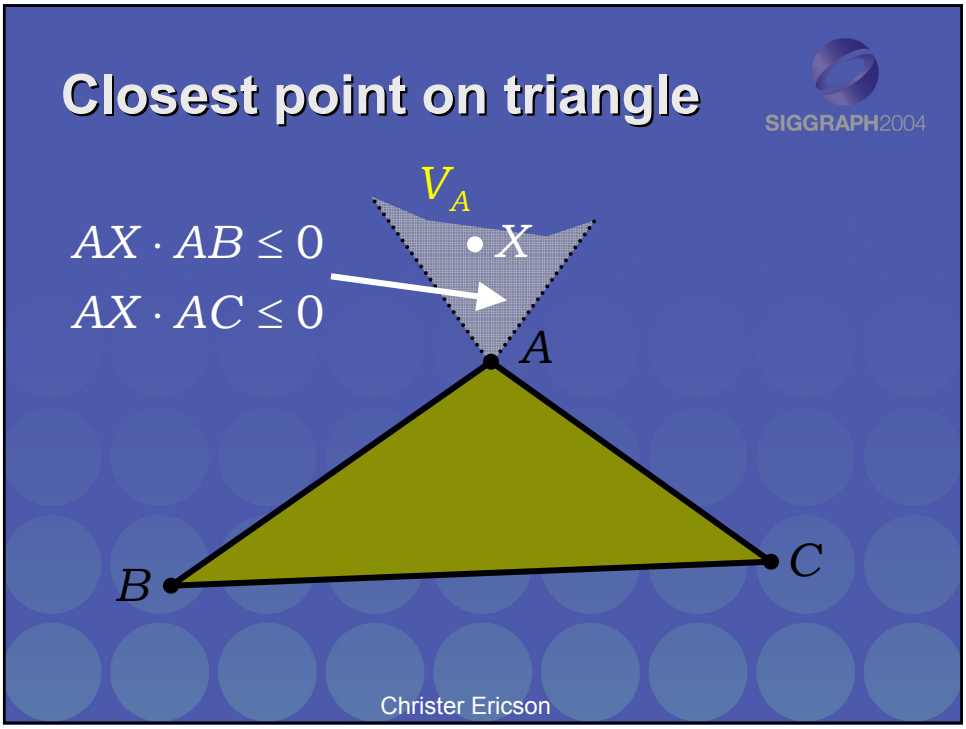
Closest point on triangle

- Separate cases based on which feature Voronoi region point lies in

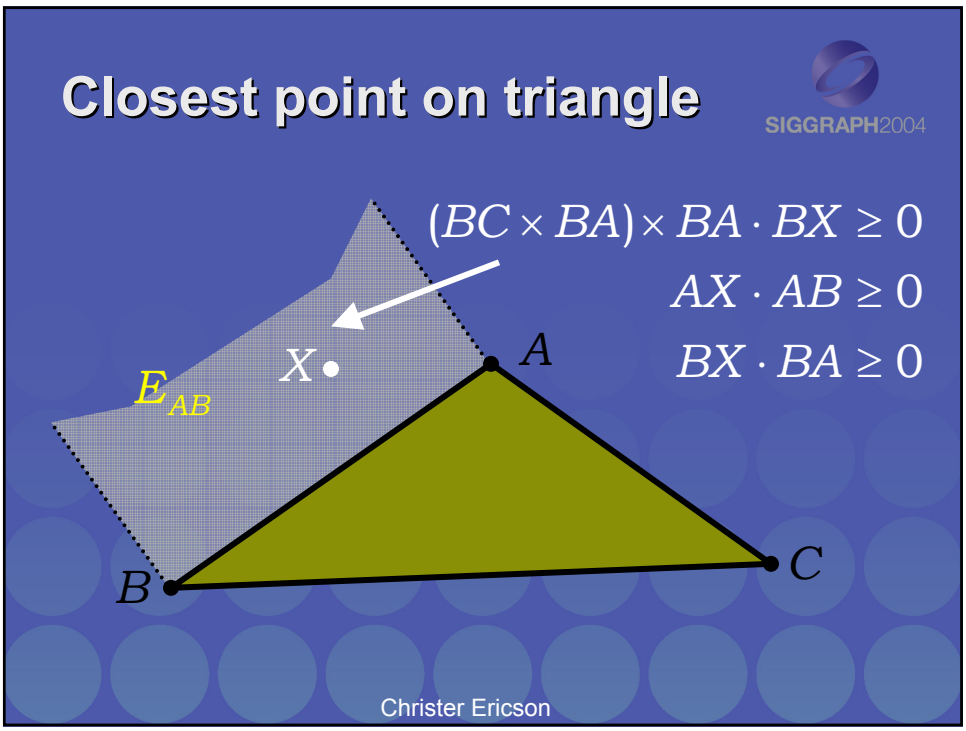


Christer Ericson

Closest point on triangle



Closest point on triangle

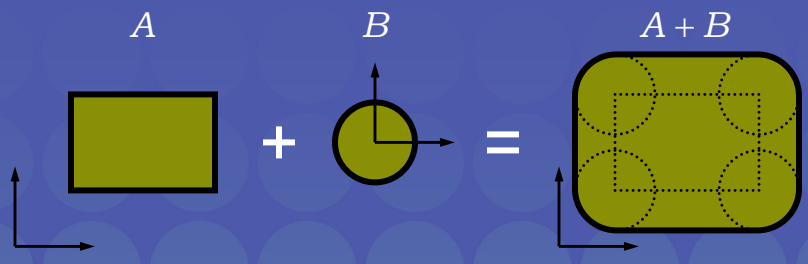


GJK for two objects

- What about two polyhedra, *A* and *B*?
- Reduce problem into the one solved
  - No change to the algorithm!
  - Relies on the properties of the **Minkowski difference** of *A* and *B*
- Not enough time to go into full detail
  - Just a brief description

Minkowski sum & difference

- Minkowski sum
  - The sweeping of one convex object with another



- Defined as:
  - $A + B = \{\mathbf{a} + \mathbf{b} : \mathbf{a} \in A, \mathbf{b} \in B\}$

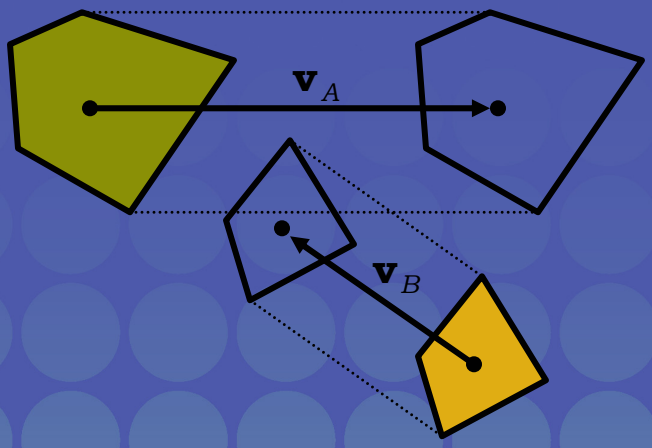
Minkowski sum & difference

- Minkowski difference, defined as:
  - $A - B = \{\mathbf{a} - \mathbf{b} : \mathbf{a} \in A, \mathbf{b} \in B\}$
    $= A + (-B)$
- Can write distance between two objects as:
  - $\text{distance}(A, B) = \min\{\|\mathbf{a} - \mathbf{b}\| : \mathbf{a} \in A, \mathbf{b} \in B\}$
    $= \min\{\|\mathbf{c}\| : \mathbf{c} \in A - B\}$
- *A* and *B* intersecting iff *A–B* contains the origin*!*
  - Distance between *A* and *B* given by point of minimum norm in *A–B!*

The generalization

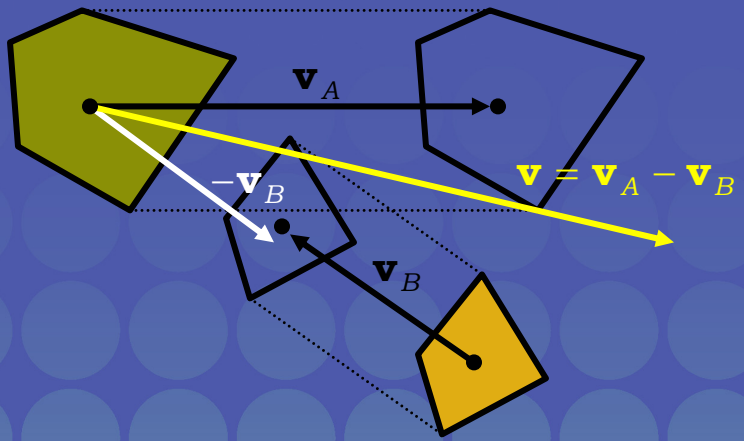
- *A* and *B* intersecting iff *A–B* contains the origin*!*
  - Distance between *A* and *B* given by point of minimum norm in *A–B!*
- So use previous procedure on *A–B!*
- Only change needed: computing $S_C(\mathbf{d}) = S_{A-B}(\mathbf{d})$
- Support mapping separable, so can form it by computing support mapping for *A* and *B* separately*!*
  - $S_C(\mathbf{d}) = S_{A-B}(\mathbf{d}) = S_A(\mathbf{d}) - S_B(-\mathbf{d})$

GJK for moving objects



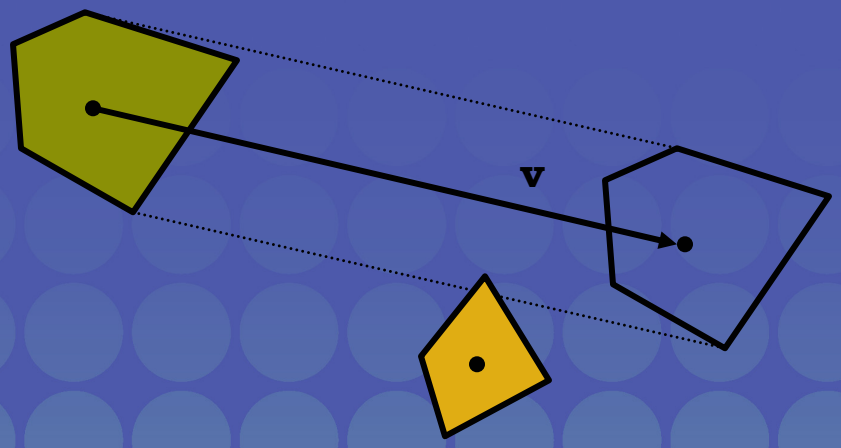
Christer Ericson

Transform the problem…



Christer Ericson

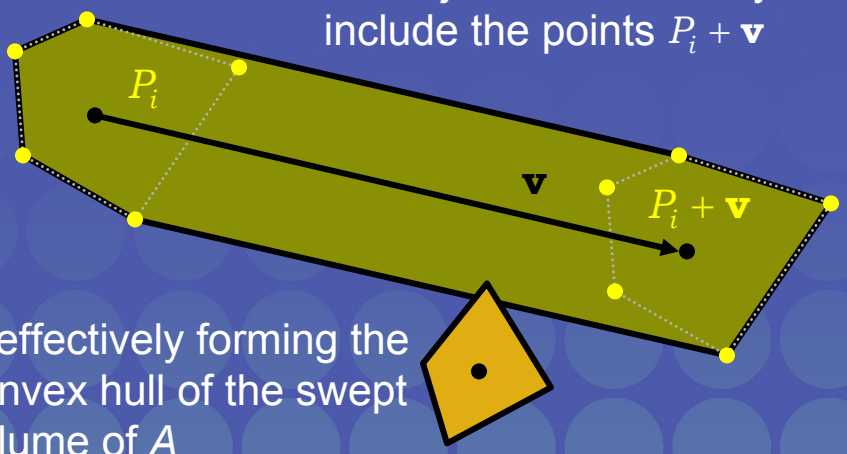
…into moving vs stationary



Christer Ericson

Alt #1: Point duplication

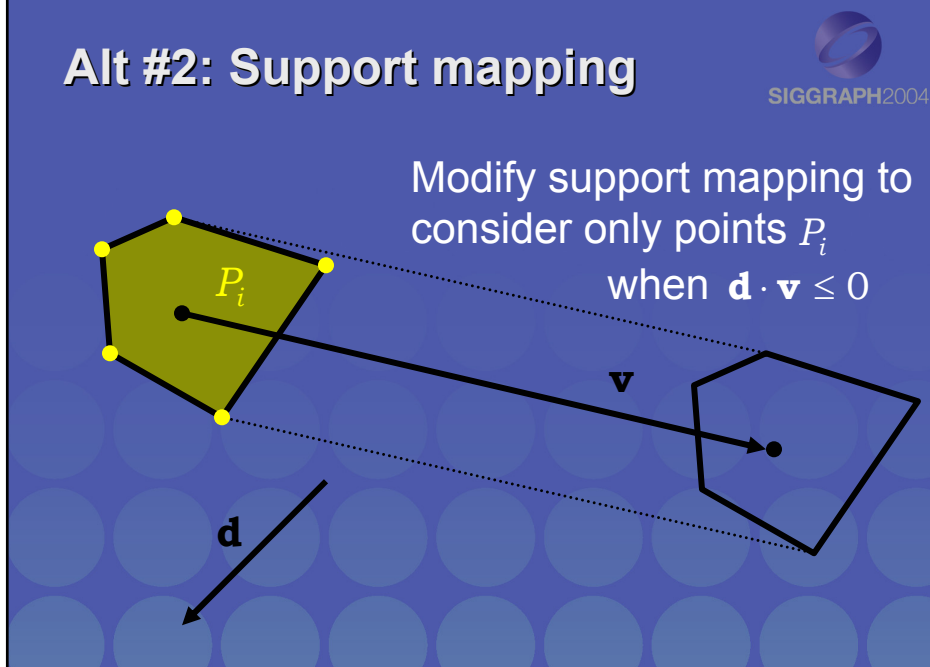
Let object *A* additionally include the points $P_i + \mathbf{v}$



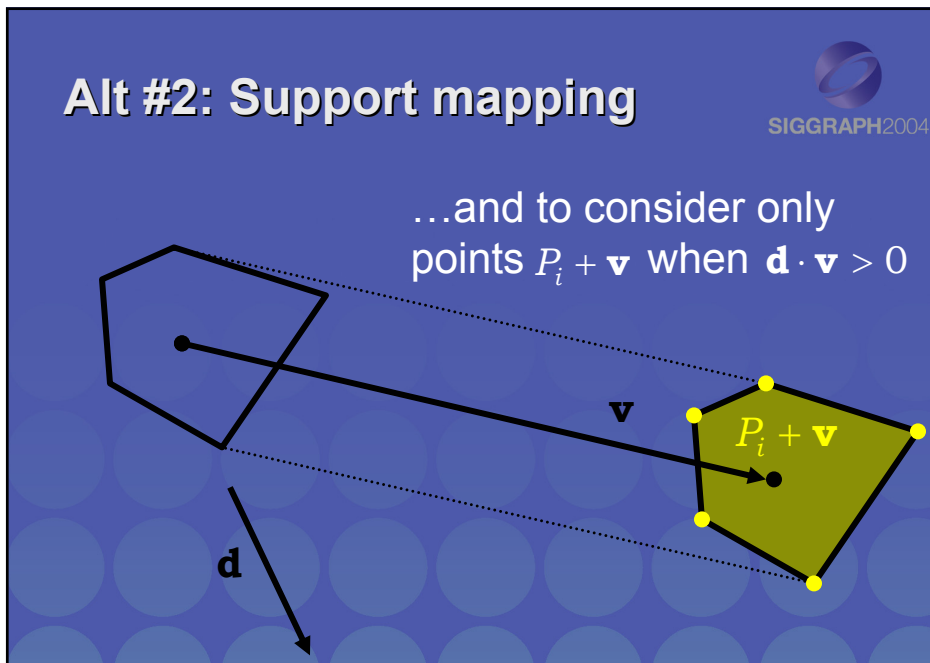
…effectively forming the convex hull of the swept volume of *A*

Christer Ericson

Alt #2: Support mapping



Alt #2: Support mapping



GJK for moving objects

- Presented solution
  - Gives only Boolean interference detection result
- Interval halving over **v** gives time of collision
  - Using simplices from previous iteration to start next iteration speeds up processing drastically
- Overall, always starting with the simplices from the previous iteration makes GJK…
  - Incremental
  - Very fast

References

- Ericson, Christer. *Real-time collision detection*. Morgan Kaufmann, forthcoming. http://www.realtimecollisiondetection.com/
- van den Bergen, Gino. *Collision detection in interactive 3D environments*. Morgan Kaufmann, 2003.
- Gilbert, Elmer. Daniel Johnson, S. Sathiya Keerthi. “A fast procedure for computing the distance between complex objects in three dimensional space.” *IEEE Journal of Robotics and Automation*, vol.4, no. 2, pp. 193-203, 1988.
- Gilbert, Elmer. Chek-Peng Foo. “Computing the Distance Between General Convex Objects in Three-Dimensional Space.” *Proceedings IEEE International Conference on Robotics and Automation*, pp. 53-61, 1990.
- Xavier Patrick. “Fast swept-volume distance for robust collision detection.” *Proc of the 1997 IEEE International Conference on Robotics and Automation*, April 1997, Albuquerque, New Mexico, USA.
- Ruspini, Diego. *gilbert.c*, a C version of the original Fortran implementation of the GJK algorithm. ftp://labrea.stanford.edu/cs/robotics/sean/distance/gilbert.c