

BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-92-M9

“Interactive Collision Detection”

by
Stephan J. Zachwieja

Interactive Collision Detection

Stephan J. Zachwieja

Department of Computer Science
Brown University

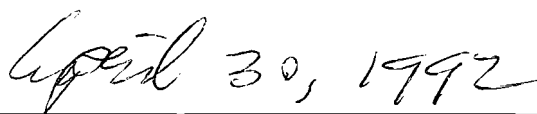
Submitted in partial fulfillment of the requirements
for the Degree of Master of Science in the
Department of Computer Science at Brown University

27 April 1992

This research project by Stephan J. Zachwieja is accepted in its present form by the Department of Computer Science at Brown University in partial fulfillment of the requirements for the Degree of Master of Science.

A handwritten signature in cursive script, appearing to read "Andries van Dam", written above a horizontal line.

Professor Andries van Dam
Advisor

A handwritten date "April 30, 1992" in cursive script, written above a horizontal line.

Date

Interactive Collision Detection

Stephan J. Zachwieja

Brown University, Providence, RI 02912
Digital Equipment Corporation, Rocky Hill, CT 06067

Abstract

When simulating motion, it is considered both undesirable and visually disturbing for objects to inter-penetrate one another. Such an effect jars one's sense of reality. In order to avoid this we add non-penetration constraints to those objects which we do not want to inter-penetrate.

The two problems involved in simulating non-penetration constraints between objects are (1) *collision detection* - detecting and describing contact between pairs of objects, and 2) *collision response* - calculating the forces present between colliding objects. This paper discusses the first problem of detecting collisions and presents an algorithm for detecting and describing collisions in an efficient manner. This algorithm is both hierarchical and adaptive and works for arbitrary closed polyhedra. Also discussed are a number of heuristics that can be used to help speed up collision detection.

1 Introduction

Traditional computer graphics and modeling systems look at objects as little more than a collection of geometric shapes. Objects are often devoid of physical properties such as mass and velocity. A user may move an object from point p_0 to point p_1 , but the change is kinematic. The object moves instantaneously, or the movement is interpolated over time, but the object never has a velocity. As faster processors and more capable graphics engines become available, users become more ambitious, scenes become more complex, and more objects are set into motion. Each object has its own point p_0 and its own point p_1 , and each object is told how it should move between the two points. However, each object generally only knows about itself. That is, object A knows where it is and where it is going, but object A has no knowledge of what object B might be doing. This means any interaction between object A and object B must be addressed by the user.

A simple collision between two objects such as a ball and the ground might be easy for a user to simulate. The ball accelerates downward toward the ground. The user calculates the time when the ball hits the ground and then negates the velocity of the ball causing it to bounce, except the object has no velocity as such; it moves kinematically. The user may also want to delay the bounce, and decrease the “velocity” to make the collision look less elastic and more realistic. Maybe the ball has some horizontal motion and should now spin as a result of friction acting on the ball when it hits the ground. The user has to determine how much spin. Just a little more; no, too much - and this is supposed to be a simple collision. The amount of work involved goes up as a function of the square of the number of objects in the scene. Each collision needs to be tweaked to perfection, and the user is forced to spend many hours watching the simulation over and over again.

Recent work has focused on the simulation of rigid bodies using Newtonian dynamics. Objects have velocity and acceleration as well as mass, moments of inertia and coefficients of friction and elasticity. Users no longer have to guess at how an object will behave when a gravitational force is applied to an object, or when a torque is applied to make an object rotate. In order for the dynamic simulation to be accurate, the interaction between objects must also be modeled. That is, when object A hits object B , how do the two objects respond? This problem is most easily solved when it is

broken into two parts. The first part is that of detecting interaction between objects. The second part is determining how to respond to the interaction. This paper discusses the first of the two problems - detecting object interaction, or collision detection.

The test for object interaction is actually a test for object intersection. Detecting that there is an intersection between two objects is relatively simple. If the objects are closed polyhedra, then simple ray intersection techniques are used. The two difficult aspects of detecting collisions are 1) calculating and describing the *initial* point of contact so that a proper response may be effected, and 2) making the calculation of that contact point efficient enough for the algorithm to be used in interactive simulations. The collision detection problem is inherently an $O(n^2)$ algorithm. Every edge of every object needs to be tested against every face of every other object, and vice versa. A naive approach can result in an unusable implementation. Special care needs to be taken to avoid unnecessary tests and duplication of effort. As usual, there are tradeoffs, and one needs to weigh the assumptions made and shortcuts taken against the loss in realism and gains in interactive speed.

2 Previous Work

There are many strong algorithms for collision detection and collision response. Their origins are mostly rooted in robotics, computational geometry, and manufacturing, although there has been a recent flurry of algorithms for doing dynamic simulations in computer graphics. Most of the algorithms are presented in the form of a dedicated simulator. That is, the simulator does one specific task; it detects collisions and responds to them. Each algorithm makes various assumptions about the motion or surface of the objects being simulated.

2.1 Motion Planning

Methods for detecting collisions are presented by Canny [4] and Gilbert, et al. [8]. Both methods address the general case of intersections between multiple polyhedra and limit movement in the system to a single object in an otherwise static world.

Canny [4] presents an algorithm for detecting the *exact* point of collision between stationary obstacles and an object that is moving with constant linear and angular velocity. While this algorithm is designed specifically for dealing with motion and path planning problems, it also is suited to the generic collision detection problem because it is extensible to multiple moving objects. Specifically, a moving object is tested against a set of obstacles to determine if they intersect. If they do intersect then the *exact* point of intersection is derived using a root-finding procedure on equations of motion which are represented using quaternions. Since rotation of a vector by a quaternion requires only addition and multiplication, the constraints on movement will be algebraic. This algebraic form of the constraints greatly simplifies computation of collision points and allows the calculation of an exact collision point using a root-finding procedure.

The best feature of Canny's algorithm is that it finds an exact point of collision. No bounds for the initial intersection test are given, but it seems clear from the presentation that the algorithm is $O(n^2)$ in the number of faces and edges in the polyhedra being tested. Given that two polyhedra intersect, the algorithm proceeds to search for roots to a twelfth order polynomial using an iterative search technique. The convergence can be very slow, and each iteration increases the amount of floating point error. This tends to make the *exact* collision points somewhat less exact.

Gilbert, et. al. [8] present a fast procedure for computing Euclidean distance between convex polytopes. Their algorithm measures distance between two objects as the distance between the two closest points in each of the objects. A collision between objects is reported when the distance between two objects is less than or equal to zero. Their paper makes no mention whatsoever about edge detection and is limited to problems where only one object is moving through an otherwise static world. Some effort is made to describe the collision by using the negative distance as a measure of collision depth, but nothing more is offered that would provide enough information to accurately respond to a collision.

The attraction in the algorithm presented by Gilbert, et. al., is the possibility of using the convex hull of an object as the bounding box. Both spheres and rectilinear bounding boxes tend to encompass a large amount of empty space which can result in much unnecessary work. However, it is not clear whether the algorithm is fast enough to remove simple bounding box tests

as a means of trivially rejecting non-colliding objects. No explicit mention is made of the bounds on the running time for this algorithm, but it is said to have a computational cost that is approximately linear in the number of points defining the polytopes. The linear cost is achieved by caching some initial computations on the static objects as a function of a known path the single moving object will take. In short, the algorithm is designed to look at a static scene and determine where collisions may occur so they may be avoided, as opposed to the more interesting case of a dynamic environment where collisions are detected and then responded to in a physically correct manner.

2.2 Interference Detection

A method for interference detection in manufacturing and industrial environments is presented by Boyse [3]. As factories become increasingly automated the use of machinery and robots also increases. Such machinery requires careful orchestration to work properly with other machinery. When designing components for product manufacturing and testing facilities it is important to take into consideration possible interference between moving objects. Unfortunately, a two-dimensional drafting medium does not always show interference among three-dimensional objects, especially when one or more of the objects is moving. If interference is not detected at the design phase then it may not be caught until a prototype is built or, worse yet, when production begins, which means lost time and money.

Boyse defines an algorithm for both static and dynamic interference checking. The static interference checking is much the same as every other algorithm discussed in this paper. That is, objects are represented as a series of planar faces defined by edges connecting vertices. The edges of one object are tested against the faces of another object using standard ray intersection techniques. Boyse then goes on to discuss techniques for dynamic interference checking. Specifically, an algorithm is offered for detecting collisions between a moving edge and a stationary face. Two types of collisions are defined. The first is when an endpoint of the edge pierces the face. The second involves the edge sweeping across and intersecting the boundary or edge of the face. The first is solved using trigonometric functions to determine if an endpoint of the moving edge pierces the face after rotating through some

angle θ . The second involves sweeping the edge over time, generating a hyperboloid of revolution and then testing for intersections between the surface and each of the edges in the face by solving quadratics.

Boyse's algorithm is limited to tests where only one object is moving. The motion of a given point or edge being tested can be made relative to the opposing face effectively making the face stationary, but the surface swept by the edge will no longer be quadratic. Additional translational and rotational components yield higher order polynomials which cannot be solved analytically, thus requiring the use of a root-finding procedure.

2.3 Implicit Representation

Any object or surface can be modeled with a sufficient number of polygons. However, sometimes it is more convenient to represent object using an implicit representation - for example, a sphere. Some objects may be represented by bicubic patches and still others are better defined by higher-level parametric functions of u and v , as in $f(u, v)$. Existing algorithms are not guaranteed to find the earliest collision unless the functions used to define the surfaces are restricted. Von Herzen, et. al. [18] present such an algorithm, limiting functions to those having computable bounds on their regional rates of change. These bounds on the rates of change are called Lipschitz values. Additionally, Von Herzen, et. al., use Jacobian bounding boxes to reduce the number of computations required to detect interference. A constraint is also placed on the maximum velocity of any point on the surface of an object. Baraff [2] presents a paper on simulating non-penetration constraints for curved surfaces. Surfaces are limited to those that are twice-differentiable without boundary. The bulk of Baraff's paper focuses on the forces present between contacting curved surfaces rather than the detection of contact between such surfaces.

2.4 Polyhedral Representations

Hahn [9], Moore and Wilhelms [10] and Baraff [1] discuss methods for modeling contact and the forces required to prevent inter-penetration between polyhedral objects. All three of the papers concentrate on the response be-

tween colliding or contacting objects.

Hahn's paper specifically mentions testing of edges against faces and a hierarchical algorithm involving bounding boxes, both which have been presented before and are incorporated into the UGA collision detection algorithm. Hahn also discusses the various types of collisions but leaves out some of the most difficult types including collisions between coplanar faces and collisions between a face and an edge parallel to that face. Hahn uses a standard backtracking technique for finding an exact point and time of collision. The backtracking technique only takes into account linear velocity. It does not account for linear acceleration, or angular components. The bulk of Hahn's paper is dedicated to collision response, including modeling the forces necessary to prevent inter-penetration and the building of graphs to model simultaneous contact between objects. This allows modeling of force propagation through a series of objects such as in the break in a game of billiards.

Moore and Wilhelms [10] focus their paper on collision response. Early portions of the paper discuss collision detection, but all discussion is limited to intersections between convex polyhedra. Equations are offered for solving vertex face intersections resulting in fifth order polynomials. These cannot be solved analytically and a binary search is used to converge on an exact time and point of collision. The remainder of the paper focuses on collision response. This includes solutions using springs and analytical solutions using large sparse matrices, the later of which can be used to respond to collisions of an arbitrary number of objects in simultaneous contact. This use of sparse matrices is the basis for collision response within UGA. However the UGA implementation is hardcoded for collisions between only two objects.

The Baraff [1] paper is dedicated almost entirely to collision response with little mention of collision detection. There is, however, a useful discussion on restricting contact points. Specifically, if an edge is in contact and parallel to a face, the contact can be modeled by just looking at the endpoints of the edge. Similarly when there is a coplanar collision between two faces, the collision can be properly modeled by applying forces only to those points which are endpoints of the edges on the faces, and those points where the edges of the two opposing faces cross. It is not necessary to integrate the collision force across the entire region of contact.

2.5 Previous Work Summary

Most of the papers discussed above, concentrate on collision *response*. That is, there is an emphasis on producing an accurate and realistic physical response between contacting objects. Each of the implementations exists within a dedicated simulator, thus limiting their general usefulness in a larger integrated graphics simulation environment. Some limit motion to a single object through an otherwise static scene, and none allow interaction by the user after the simulation begins. Simulation times are given in minutes. However, users want to see the simulation as it happens, and they want to see smooth continuous motion. They want to see multiple moving objects, and they want to be able to change the motion of object on a whim while the simulation is in progress. The algorithm within this environment should not only produce a correct physical simulation, but it should be efficient enough for it to be interactive and useful as a tool for creating complex dynamic simulations. Furthermore, the entire package should be integrated into a complete graphics modeling environment. Toward that end, my focus was on creating an efficient, practical, and useful algorithm that would integrate into the rich graphics modeling environment that is UGA.

3 Collision Detection within UGA

My goal for collision detection within UGA was to implement a general purpose collision detection algorithm. That is, an algorithm that would fit into a general purpose simulator. Most of the previous work in collision detection and response involves a dedicated simulator. Such simulators have knowledge about object shapes and collision dynamics, but little else. The UGA environment encompasses a large number of different packages that allow multiple simulation processes to run concurrently affecting the same set of objects. It was important that my collision detection algorithm fit into this framework. Another goal was to make the collision detection algorithm fast enough for the user to interact with it. If the simulation is too slow and motion is jerky, then it is difficult for the user to interact with the objects. Algorithms involving dedicated simulators don't give users the privilege of interaction. The objects are defined and set into motion, and the simulation runs to completion - by itself. When two objects intersect the simulation

starts backtracking to find the initial point of intersection. Typically this involves moving the intersecting objects back in time until just before initial contact is made. Existing algorithms use a number of binary search techniques, moving the objects to their positions at a previous time, and then checking to see if they intersect at that time. This means another complete intersection test, and the transformation of the entire object at each iteration in the binary search.

In contrast, the UGA collision detection algorithm does extent checking on all faces and edges to reduce the number of complete edge and face tests. When testing a face against the edges of another object, the face is tested first against the extent of the other object. This can eliminate all the edge tests for the face with one simple extent check. The extent checks are essentially free. Furthermore, if the time step is sufficiently small, then only two or three faces of the first object will intersect the opposing object making the lower bound $O(n)$ in the number of edges in the opposing object. Furthermore, the UGA collision detection algorithm does not transform the entire object at each iteration of the binary search. Only those components which are known to be intersecting are moved backward in time, and additional components may be eliminated from the search at each subsequent iteration. In practice, this makes the time complexity for intersecting objects near constant.

At the higher level, the collision detection algorithm does pairwise testing of every object being simulated. Tests begin simply using rectilinear extents. If the extents of two objects intersect then additional tests are performed. All the faces of one object are tested against the edges of the opposing object and vice versa. All edges and faces found to be intersecting are then placed on a list. The list is then processed using a backtracking technique to find the initial point or points of contact. All points of contact are then coalesced into a single point of contact and passed along with information describing the plane of collision which is used to respond to the collision.

3.1 Discrete Simulation

The simulation process models time discretely. That is, the user defines the interval or granularity at which the system will be updated. If an object is moving at a speed of two, that means it is moving two units of distance per one unit of time. If the user defines the interval to be 0.1, then the system is

updated ten times per unit of time and the object moves 0.2 units of distance during each interval or *time step*.

Modeling time discretely creates aliasing problems. If the system is not updated at a high enough frequency some events may be lost. For instance, figure 1 shows two balls of radius one; the distance between the two balls is also one. If the ball on the left moves two units to the right in a single time step, and the ball on the right moves two units to left in that same time step, the balls will exchange positions and will not be penetrating at the end of the time step. No collision will be detected. This can be overcome by extruding the moving object volumes and then testing the volumes for a collision, but this can be very expensive. The alternative is to increase the sampling rate by reducing the time interval. This effectively limits the displacement of objects within a time step thus reducing the possibility that two objects will pass completely through each other in a single time step.

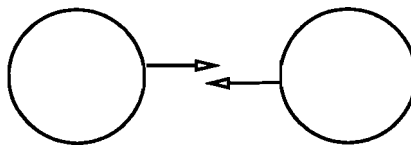


Figure 1: two balls passing through each other

Another problem with discrete time modeling is that collisions are not detected until the end of the time step in which they occur. Consider a simulation proceeding from time 0.0 to time 1.0 with a collision occurring at time 0.75. If the interval is 0.1, then the simulation will run from time 0.0 to time 0.7 with no collisions detected. The next time step occurs at time 0.8. By this time the objects that would otherwise collide and bounce off each other at time 0.75 are now penetrating each other. If the collision is to be modeled properly then it is necessary to move backward in time and find the point of initial contact. This process is called *backtracking*.

3.2 Objects and Controllers

The implementation of collision detection is based on the notion of *objects* and *controllers* within the UGA architecture. Objects are a combination of data structures and methods. The data structures contain state information for some small portion of the system, such as graphical objects, logical tools, and physical devices. Methods are procedures having inherent knowledge about the objects to which they belong, allowing them to manage, maintain and retrieve information on objects. A controller is a special class of object that observes other objects and modifies those objects as a function of observed interaction between those objects. For instance, an electron passing between two oppositely charged plates will turn toward the positive plate. The electron object knows only about the electron object. The plate objects know only about charged plates. It is the controller that knows how electrons interact with charged plates and observes all three objects to make sure that the interaction between the electron and the charged plates is correct.

There are two controller objects within UGA for detecting and responding to collisions. The collision detection controller, or detection object, observes other objects and reports when objects have collided. The collision response controller, or response object, applies forces to objects based on the information returned by the detection object. At each time step in a simulation the response object is queried once for each object it is observing. If it is asked about object *A*, it then asks the detection object if object *A* has collided with any other object. If object *A* has collided with another object, then the detection object reports that collision to the response object. The response object, knowing how colliding objects interact with one another, then applies changes to object *A* to effect the correct response. Note that no changes are applied to the object with which object *A* collided. If object *A* collided with object *B*, changes to object *B* will be applied at such time that the response object is explicitly asked about object *B*.

3.3 Object Representation

The current implementation of collision detection within UGA is limited to tests between closed polyhedra. The polyhedra can be both regular and non-regular and need not be convex. The boundary representation for all such

polyhedra is a triangle mesh composed of information on individual vertices, edges and faces. Additional connectivity information is available as are face and vertex normals for the polyhedra. Extent information is maintained for the object as a whole and may also be present for individual faces and edges. The collision detection object uses a lazy evaluation technique for gathering data. If an object is not involved in a collision or a near collision, then the object data will include only the object extent. When the extents of two objects intersect additional inquiries are made to retrieve the boundary representations of the objects, and additional edge and face extents are computed.

If it is determined that two objects intersect, then physical data is added to the object as well. Specifically, the detection object makes inquiries about velocities and accelerations so that it may begin backtracking to determine the initial time and point of intersection. Additional physical data, such as mass, moments of inertia, and coefficients of friction and elasticity are available but are used only by the response object.

3.4 Intersection Detection

The first step in detecting a collision is determining whether two objects intersect. At each time step the response object makes multiple inquiries of the detection object as to which objects have collided with which other objects. When queried about a particular object A , and a set of objects B , the detection object makes inquiries of object A and each object in the set B to obtain their rectilinear extents. The extent of object A is tested against the extent of each object in the set B . If the extents of two objects do not intersect then the objects themselves cannot intersect, and that pair of objects is trivially rejected as not intersecting.

If the extents of any two objects intersect then it is necessary to do additional testing between the boundary representations of those objects. The following set of necessary and sufficient conditions are defined for detecting intersections between two objects, A and B , where both objects are closed polyhedra represented by a triangle mesh as described in section 3.3.

1. a vertex of A lies inside B , or
2. a vertex of B lies inside A , or

3. an edge of A intersects a face of B , or
4. an edge of B intersects a face of A .

An informal proof of these conditions is discussed in [4]. Conditions 3 and 4 both require $O(nm)$ ray intersection tests where n and m are the number of faces and edges in each object. Conditions 1 and 2 require an additional $O(nm)$ ray intersection tests, where n is the number of faces and m is the number of vertices, to perform an odd-even rule test to determine if each vertex is inside or outside of an object. The number of vertex tests can be significantly reduced if one assumes the time step is sufficiently small such that no vertex passes so far into an object that it has no connecting edges which intersect a face of the object. Conversely, this means that all vertices inside of an object have at least one connected edge that intersects a face of that same object.

If conditions 3 and 4 are tested for first, and a list of edges is constructed, then conditions 1 and 2 can be eliminated. This is true because every vertex on the inside of the object will by definition be connected to an edge that intersects a face of the opposing object. Since all such edges are kept in a list, all vertices inside of an object can be obtained from that list without additional testing.

For instance, figure 2.a shows a cube penetrating the surface of a plane. There are two vertices beneath the plane. Both vertices can be found by looking at the endpoints of the two edges piercing the plane. Figure 2.b shows a cube with beveled edges penetrating the same plane. Four of the vertices of the cube are beneath the plane. The two lower vertices are not connected to any edge that intersects the face of the plane. This conflicts with the above assumption requiring all vertices inside of an object have at least one connected edge intersecting a face of that same object.

When testing for intersections between the boundary representation of two objects all faces of the first object are tested against all edges of the second object, and vice versa. Initial testing is done between face and edge extents. If the extent of a face does not intersect the opposing object extent, then there is no need to test that face against every edge in the opposing object. All faces and edges that are not eliminated by the extent testing are tested pairwise using a ray intersection technique. Each intersecting edge and face pair is recorded and added to a list of intersections. The vertex

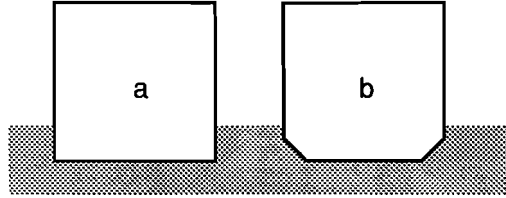


Figure 2:

of the intersecting edge that is behind the face is also placed on the list of intersections. Recording both the edge and face pair as well as the vertex provides enough information to determine how the intersection developed. That is, how did the edge first come in contact with the face? There are two possibilities. Either an endpoint of the edge pierced the face, or the edge crossed one of the edges that defines the boundary of the face.

3.5 Micro Backtracking

If two objects intersect, then two lists are constructed during the initial intersection testing that describe all the points of intersection. The first list contains all pairs of intersecting faces and edges. The second list is derived from the first and contains those vertices on the edges in the first list that lie behind the faces those edges intersect. There is a one-to-one correspondence between the items on each of the lists. The lists represent many points of contact, but the response object is only interested in the initial point of contact.

Like many of the papers discussed in section 2, the UGA collision detection implementation uses a backtracking technique to find the initial time and point of intersection. Specifically, the detection object moves backward in time and then checks to see if the two objects being tested are still intersecting. If the detection object moves back in time, and the objects are no longer intersecting, then the time chosen is too far back and the detection objects moves forward in time. The detection object continues this process moving backward and forward as necessary until it converges on the initial

time and point of intersection. Most algorithms transform both objects involved at each intermediate time step in the backtracking process. The UGA collision detection algorithm uses the lists of faces, edges and vertices that were constructed in the initial intersection tests to isolate those components of the objects that are involved in the collision. Transforming two faces individually may result in duplication of effort because the two faces may share vertices. If it is determined which vertices are involved in the testing, then those vertices are transformed once and only once at each step in the backtracking process.

At each step in the backtracking process additional faces, edges, and vertices are eliminated from testing. This reduces the amount of work which will have to be done for future iterations in the backtracking process. For instance, edge *A* and face *A* first intersect each other at time 0.12, and edge *B* and face *B* first intersect each other at time 0.13. Both intersections occur in the current time step which goes from 0.0 to 1.0. The initial binary search begins at time 0.5. The initial intersection time for both edge face pairs occurred before that time. The second iteration in the binary search is done at time 0.25. Again both intersections occurred before that time and the third iteration is done at time 0.125. The intersection between edge *A* and face *A* occurs before the time 0.125 and the intersection between edge *B* and face *B* occurs after.

At this point there are two things to note. First, the goal is to find the first point of intersection. This means when doing backtracking the precedence for the search is to test for those intersections that occurred earliest in time. So when given the choice above, the next iteration in the binary search sets the time to 0.0625 so that the search converges on the intersection between edge *A* and face *A*. The second thing to note is that the faces, edges, and vertices in the list are only potential intersection points. That is, following edge *A* and face *A* backward in time may produce no valid intersection point. It may in fact be determined that edge *A* and face *A* have an initial intersection time of 0.12, but it may also be the case that the face was penetrated from the rear which is an uninteresting case (see section 3.6.1 for an explanation). This means work on edge *B* and face *B* can only be suspended, not abandoned. Each time the backtracking process moves backward in time, all of the edges and faces going backward in time are placed on a list, all of the edges and faces forward in time are placed on a second

list which is in turn pushed on a stack structure. The stack can be accessed later if the current backtracking path fails to yield an intersection point. If edge A and face A yield a valid intersection point, then all lists on the stack are popped off the stack and discarded, because it is clear that the unfinished searches remaining on the the stack will converge on a time that is later than the intersection between edge A and face A . However, it is not really known in advance that one intersection occurs at time 0.12 and the other at time 0.13. When the backtracking process splits the components at time 0.125, it may be the case that the intersection of edge A and face A occurs between time 0.125 and time $0.125 - \epsilon$ and that the intersection between edge B and face B occurs between time 0.125 and time $0.125 + \epsilon$. If two intersections occur within some epsilon of each other, then those intersections are treated as a single intersection, and both intersections are needed. This means when an intersection is found at some time, the contents of the stack can only be discarded when the time for the last iteration on the top of the stack is at least one epsilon greater than the intersection time.

Micro-backtracking takes place for all edge and face pairs as well as the list of vertices derived from the list of edge and face pairs. Sections 3.7 and 3.6 discuss this in greater detail. The discussion is treated outside the scope of micro-backtracking to make it easier to understand. All such binary searches are done in parallel using lists and stacks to reduce the number of transformations and eliminate unnecessary tests.

3.6 Vertex Detection

As mentioned in section 3.4, a list of vertices is constructed during the initial intersection test. Each vertex in the list is an endpoint of an edge intersecting a face of the opposing object. Each vertex has potentially intersected a face of the opposing object. The face that the vertex intersects may not be the same face the corresponding edge intersects. For this reason each vertex in the list is tested against every face of the opposing object.

3.6.1 Backface Culling

Given a vertex and a face, it is necessary to determine if the vertex passed through the face in the previous time step. This is done by first determining

if the vertex has passed through the plane defined by the face. A vector is constructed from the vertex to a point on the plane, and then the scalar dot product of that vector and a normal to the plane is taken.

If the scalar dot product is negative, then the vertex is in front of the face and the vertex face pair can be trivially rejected. Specifically, if the time step is small relative to the size and velocity of the objects being tested then movement within that time step is essentially linear. This means that in order for the vertex to intersect the face and finish in front of the face it would have to have passed through the face from behind. Since all of the objects are closed polyhedra, the only way the vertex could pass through the face from behind is if it was inside the object just before intersecting the face from behind. Furthermore, the only way the vertex could be inside the object is if it first passed through the front of some other face of the object to get inside. Since the goal is to find the first intersection or set of intersections, all intersections where the vertex passes through a face from behind can be ignored.

3.6.2 Binary Search

If the scalar dot product is positive, then the vertex is behind the face and backtracking begins using a binary search. At each iteration of the binary search a vector is constructed from the vertex to the plane defined by the face. The vertex is known to be behind the face at the end of the current time step so the binary search begins at the beginning of the time step. In the first iteration, if the scalar dot product of the constructed vector and the normal to the face is positive, then the vertex is behind the plane at the beginning of the time step as well. Since motion is assumed to be linear within a given time step this means the vertex does not cross the plane within the time step and therefore cannot intersect the face. The binary search can then be terminated after only one iteration.

If the scalar dot product of the constructed vector and normal to the face is negative then the vertex has moved across the plane in front of the face. This means the vertex has crossed the plane in the time step and the binary search continues to determine the initial time and point that it crosses the plane. At each iteration a new vector is constructed. Each time the scalar dot product with the normal to the face is negative the point is in front of

the plane and the search interval is moved forward in time toward the end of the time step. Each time the scalar dot product is positive the search interval is moved backward in time toward the beginning of the time step. This process continues until a point is converged on.

In practice it is impossible to determine the exact point of collision, and the binary search is stopped at such time that some level of precision has been reached and the last iteration placed the vertex in front of the plane defining the face. It is important to continue the binary search if the point is still behind the face, even after reaching the desired level of precision. If the binary search is terminated with the point still behind the face then the objects are still intersecting and the collision cannot be properly resolved.

Once the point at which the vertex crosses the plane has been calculated it is necessary to determine if the point lies within the face itself. Since all of the faces are triangles, a standard technique is used for determining whether the point is in the interior of the face. If the point is inside the face then a record of the intersection is made and inserted into a list of collisions which will be used to describe the final point of collision.

3.7 Edge Detection

After doing the initial object intersection test as described above in section 3.4, there exists a list of edges and the faces they intersect. Given an edge and the face it intersects from that list, one of two things must have transpired for the intersection to have occurred. Either an endpoint of the edge pierced the face, or the edge crossed one of the edges defining the boundary of the face. If an endpoint of the edge pierced the face it will be detected during the vertex detection phase as described in section 3.6. This means only the latter case needs to be tested. Once again a binary search is used to find the edge of the face, if any, that the intersecting edge has crossed to enter the face.

3.7.1 More Binary Search

As mentioned before, the time step is assumed to be sufficiently small such that the motion is linear within a time step. The list of edge and face

intersections was constructed at the end of the time step, so the binary search to determine the initial point of contact starts at the beginning of the time step. If an edge and face still intersect at the beginning of the time step then the edge never crosses the boundary of the face. If the edge no longer intersects the face then the edge has crossed one of the edges on the boundary of the face. The edge that was crossed is recorded and the binary search continues until convergence on the point where the intersecting edge crossed the boundary of the face.

3.7.2 Polygon Traversal

Because the boundary representation of the objects is defined using triangles and not arbitrary polygons, each face of an object may actually be composed of multiple triangles. The list of edge and face intersections are really edge and triangle intersections. Some edges of these triangle are interior to a larger polygon that defines a face. Such is the case for the diagonal edge across the face of a cube. Given an edge and the triangle it intersects, it is necessary to determine if that edge crosses the edge boundary of that triangle. If the edge crosses the boundary of the triangle and the edge defining that boundary is an interior edge to a larger polygon defining the face, then the search continues backward in time with the triangle on the other side of that edge. In this manner it can be determined which edge of the larger polygon defining the face is crossed rather than one of the interior edges of a single triangle in the face.

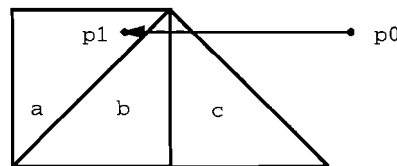


Figure 3: one face, but three triangles

Figure 3 shows the top face of a trapezoidal cylinder defined by the three

triangles a , b and c . In the same figure there is a point, p_1 showing where an edge normal to the face intersects the triangle a . At the beginning of the time step, the edge normal to the face was at the point p_0 . Since linear motion is assumed, the edge would have to move across triangles c and b before entering triangle a . As the binary search continues backward in time, the edge normal to the face crosses the edge separating triangles a and b . If the triangle on the other side of the edge is coplanar with the triangle currently being traversed, triangle a , then the search continues through that adjacent coplanar triangle, b . The search then moves across triangle b and then into triangle c . Triangle c is traversed until the edge normal to the face exits at the rightmost edge of triangle c . The triangle on the other side of the rightmost edge in triangle c is not coplanar with triangle c and the traversal is complete. This is the initial point of contact which is recorded in a list of collisions used to describe the final point of collision.

3.7.3 Interior Edges

For the same reasons as described in the section on polygon traversal, 3.7.2, if there is an edge in the list of edges and faces which is an interior edge, then that edge can be ignored.

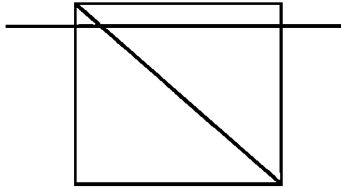


Figure 4: edge crossing an interior edge of a face

In the case where an edge falls onto a face and is coplanar with that face producing a series of intersections all at the same time, it is necessary to eliminate intersections with interior edges. Consider figure 4 showing the top face of a cube with an edge lying across the face. The edge intersects three other edges producing three separate intersections. Clearly the resulting

collision force between the edge and the face should be halfway between the rightmost and leftmost intersection point. If an averaging technique is used, the collision point would be weighted toward one end because of the extra intersection with the interior edge making the collision response incorrect. By eliminating this type of intersection early, the job becomes easier in the later stages when there may be a large list of collision points to sort through in an effort to find a single point and a collision plane that accurately describe the contact between two objects.

3.8 Data Reduction

The response object expects one collision event to be returned for each pair of colliding objects. If object *A* collides with objects *B* and *C* then two events are returned for object *A*, and one event is returned for each of objects *B* and *C*. There are several different types of contact that can occur between colliding objects, and it may be the case that a pair of objects collide at multiple points simultaneously. Each type of contact has its own footprint and generates a different set of contact points. For instance, a collision between a vertex and a face generates one contact point. Contact between an edge and a face may generate several collinear points. A collision between two faces generates a coplanar set of contact points lying on the convex hull of the intersection between the two faces. If there is a single contact point, then the job is easy and that single point is passed straight through to the response object. If the list contains two unique contact points then the two points are averaged together and the result is passed to the response object. Collisions involving three or more points of contact require additional work.

The UGA collision detection algorithm looks for two things in a large set of contact points. The first is to see if the points are collinear. The second is to see if the points are coplanar. If a set of points is collinear then the two endpoints or extrema on the line are averaged together to yield a single point of collision. All other intermediate points are discarded. If a set of points is coplanar, but not collinear, then all of the points are averaged together to yield a single point of contact. A better algorithm for a set of non-linear coplanar points would be to determine the center of the convex hull and use that as the contact point. However, one of the main goals of the collision detection algorithm is that it be interactive. Taking the average is much

faster.

The last type of multi-contact collision set is when there are four or more non-coplanar points. This only happens when non-convex objects intersect. In this case all of the points are simply averaged together. A better algorithm would be to divide the points into sets of coplanar points as a function of which edges and faces they lie on. However, such an algorithm would have an exponential time complexity. Taking the average produces reasonable results for most such intersections. Once again the need for speed wins out.

After reducing the data to a single collision point, and constructing the plane of collision about that point, the collision is reported to the response object and collision detection for the particular object pair is complete.

3.9 Restrictions

The current detection algorithm makes some assumptions about objects and their motion. The following restrictions apply to the collision detection implementation within UGA.

- The boundary representation of all objects must be polyhedra composed of triangle faces. The current collision detection implementation does all of its work on triangles faces, edges and vertices. This is the representation used in the modeler within UGA.
- No pair of objects may be intersecting at the beginning of a time step. If objects are intersecting at the beginning of a time step their intersection will not be detected until the end of the time step. The detection object then works backward toward the beginning of the time step trying to *undo* the intersection. If a pair of objects intersects at both the beginning and the end of a time step, the initial point of intersection cannot be determined because it does not occur within the current time step.
- All polyhedra must be closed. The implementation assumes that this is true and makes decisions based on this assumption. If objects are not closed then some intersections and hence some collisions may be missed as a result of the back-face culling optimizations.

- The time step must be sufficiently small so that all vertices inside an opposing object are connected to at least one edge that intersects a face of that opposing object (see section 3.4).

4 Future Work

The implementation of collision detection within UGA is complete for both convex and non-convex polyhedra. Extensive testing has been done using convex polyhedra. Testing of non-convex polyhedra has not been as rigorous. The algorithm should work well for most situations. However, there are a number of improvements that could be made to make the algorithm more robust, more realistic and more interactive in terms of speed.

4.1 Macro Backtracking

As mentioned in section 3.1, the simulation of time is discrete. That is the system is updated at some discrete interval of time. The detection object is called once per time step for each pair of objects in the system. If multiple collisions are detected in a single time step, then only the earliest is reported. The later collisions must be ignored because the first collision causes a discontinuity in the system. Any work done after that discontinuity is invalid.

For instance, objects A , B , and C are being simulated from time 0.0 to time 1.0 at an interval of 0.1. At the end of each time step, object A is tested against objects B and C , and object B is tested against object C . At times 0.1 and 0.2 there are no intersections and, therefore, no collisions. At time 0.3 inquiries are made for the positions of all three objects. Objects B and C are detected as intersecting and backtracking determines the initial point of contact to be at time 0.25. The test done between object A and B is now invalid because the inquiry for the position of object B at time 0.3 did not take into account the collision with object C . The test between object A and object C is also invalid. The response object could integrate the velocities and accelerations for objects B and C over the remainder of the time step, and then call the detection object again. However, the response object does not know how to integrate velocities and accelerations. The response object only knows how to respond to collisions. In the current implementation,

if there are additional events that occur between time 0.25 and 0.30 as a result of the collision at time 0.25, then they are discarded and lost. This includes a second collision that might occur between objects *B* and *C*. The end result can be incorrect response and in some case unrecoverable penetration of objects.

The solution to this problem is macro-backtracking - backtracking of the entire simulation to handle a discontinuity in the system. Specifically, if a simulation goes from time 0.0 to time 1.0 and an event changes the system at time 0.5, then that event should be processed at the time it occurs, not at the end of the time step. The simulation should then be restarted at the time of the event. Such support would need to be added at the system level, so that it may be used by all controllers in a consistent and uniform manner. Note also that special care needs to be taken to avoid needless resimulation of objects that would not otherwise be affected by the event.

4.2 Contact Collisions

The UGA collision detection algorithm is capable of detecting contact collisions, but the collision response model does not handle such collisions. For instance, if a block is lying on a table with downward acceleration due to gravity the block collides with the table at the beginning of the time step. The collision will not be detected until the end of the time step. By that time the block has fallen through the table. At the beginning of the time step the objects have no velocity and therefore no momentum. The response algorithm uses momentum transfer and kinematic changes to effect responses; no forces are involved. In this particular case the block is translated back to the top of the table based on the time of initial contact and the depth of penetration. The correct response would be for the table to apply a normal force to the block so that the effects of gravity are negated.

4.3 Multiple simultaneous collisions

Multiple simultaneous collisions are detected by the current UGA collision detection algorithm. Such collisions are passed as a list back to the response object. However, the collision response object does not handle the multiple

collisions. Specifically, it looks for the earliest collision in the list and responds to that single collision and no others. The detection object always returns the earliest collision. When the detection object returns multiple collisions, they all occur at the same time. This means that the single collision chosen by the response object is a function of the list ordering, which is essentially random based on the order in which inquiries are made in the user script.

A simple enhancement to the response algorithm would be to process the entire list of collisions returned by the detection object, and sum the responses. The sum would then be applied to the object of which is being inquired. This would handle most instances of multiple simultaneous collisions. However, this would not handle propagation of collisions through a series of contacting objects. In order to handle propagation forces a model using dependency graphs would have to be constructed to apply forces properly to all objects involved. This is discussed in the papers by Baraff [1] and Hahn [9].

4.4 Interframe Coherence

Collision detection in a simulation environment is a series of geometric intersection problems - one at each time step. Each problem is very similar to the one before. However, most collision detection algorithms restrict themselves to static configurations. Information computed in a previous time step is ignored at the current or next time step. If a collision detection algorithm can structure itself to take advantage of the geometric coherence between time steps, the running time may be substantially reduced.

Little work has been done in the area of coherence. This is true not only for dynamic simulations but in robotics and computational geometry as well. I had investigated a number of methods including spatial subdivision using dynamic d -rectangle trees [13] [6] [7] to reduce the number of object tests to $O(n \log n)$, and the use of BSP trees to reduce the number of edge and face intersections [16] [15] [11]. However, the caching mechanism within UGA makes it difficult to create a persistent dynamic data structure. If a data structure is created and cached at time t and then a change is made to that structure at time $t + 1.0$, then the entire data structure needs to be copied before it can be changed. In this case the amount of work required to change

current system exceeded the amount of available time.

4.5 Implicit Representations

The current collision detection algorithm expects a triangle mesh boundary representation. Spheres, quadrics and other implicitly defined surfaces need to be tessellated if they are to interact with other objects. The tessellated approximations will yield approximate results which can vary as a function of the tessellation granularity. It would not be unreasonable to have several detection objects each capable of dealing with collisions between certain classes of objects. The only detection object in UGA today detects collisions between two closed polyhedra. The next two obvious choices for detection objects would detect collisions between two parametric surfaces and between a parametric surface and a closed polyhedra.

5 Conclusion

This paper presents an algorithm for detection collision between closed polyhedral objects. While the algorithm is $O(n^2)$ in the number of edges and faces being simulated, heuristics are presented for reducing the time complexity to a near constant time for non-intersecting objects. Additionally, the computational cost of tests between intersecting objects is significantly reduced using micro-backtracking techniques. All of this work has been incorporated into UGA.

References

- [1] DAVID BARAFF, "Analytical Methods for Dynamic Simulation of Non-penetrating Rigid Bodies", Computer Graphics, Volume 23, Number 3, July 1989, pp. 223-231
- [2] DAVID BARAFF, "Curved Surfaces and Coherence for Non-penetrating Rigid Body Simulation", Computer Graphics, Volume 24, Number 4, August 1990, pp. 19-28

- [3] JOHN W. BOYSE, "Interference Detection Among Solids and Surfaces", Communications of the ACM, Volume 22, Number 1, January 1979, pp. 3-9,
- [4] JOHN CANNY, "Collision Detection for Moving Polyhedra", IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume 8, Number 2, March 1986, pp. 200-209.
- [5] R.K. CULLEY AND K.G. KEMPF, "A Collision Detection Algorithm Based on Velocity and Distance Bounds" Proceedings 1986 IEEE International Conference on Robotics Automation, Volume 2, pp 1064-1069.
- [6] HERBERT EDELSBRUNNER, "A New Approach to Rectangle Intersections, Part I", International Journal of Computer Mathematics, Volume 13, 1983, pp. 209-219.
- [7] HERBERT EDELSBRUNNER, "A New Approach to Rectangle Intersections, Part II", International Journal of Computer Mathematics, Volume 13, 1983, pp. 221-229.
- [8] ELMER GILBERT, DANIEL JOHNSON AND S. SATHIYA KEERTHI, "A Fast Procedure for Computing the Distance Between Objects in Three Dimensional Space", IEEE Journal of Robotics and Automation, Volume 4, No. 2, April 1988, pp. 193-203.
- [9] JAMES K. HAHN, "Realistic Animation of Rigid Bodies", Computer Graphics, Volume 22, Number 4, August 1988, pp. 299-308.
- [10] MATTHEW MOORE AND JANE WILHELMS, "Collision Detection and Response for Computer Animation", Computer Graphics, Volume 22, Number 4, August 1988, pp. 289-298.
- [11] BRUCE F. NAYLOR, JOHN AMANATIDES, AND WILLIAM C. THIBAUT, "Merging BSP Trees Yields Polyhedral Set Operations" Computer Graphics, Volume 22, Number 4, August 1988, pp. 289-298.
- [12] JOHN C. PLATT AND ALAN H. BARR, "Constraint Methods for Flexible Models", Computer Graphics, Volume 24, Number 4, August 1990, pp. 115-124.

- [13] HANS W. SIX AND DERICK WOOD "Counting and Reporting Intersections of d -Ranges" IEEE Transactions on Computers, Volume C-31, Number 3, March 1982, pp. 181-187.
- [14] DEMETRI TERZOPOULUS, JOHN C. PLATT, AND ALAN H. BARR, "Elastically Deformable Models", Computer Graphics, Volume 21, Number 4, July 1987, pp. 205-214.
- [15] WILLIAM C. THIBAUT AND BRUCE F. NAYLOR, "Set Operations on Polyhedra Using Binary Space Partitioning Trees", Computer Graphics, Volume 21, Number 4, July 1987, pp. 153-162.
- [16] ROBERT B. TILOVE "A Null-Object Detection Algorithm for Constructive Solid Geometry", Communications of the ACM, Volume 27, Number 7, July 1984, pp. 684-694.
- [17] TETSUYA UCHICKI, TOSHIKA OHASHI, AND MARIO TOKORO, "Collision Detection in Motion Simulation", Computers & Graphics, Volume 7, Number 3-4, 1983, pp. 285-293.
- [18] BRIAN VON HERZEN, ALAN H. BARR, AND HAROLD R. ZATZ, "Geometric Collisions for Time-Dependent Parametric Surfaces", Computer Graphics, Volume 24, Number 4, August 1990, pp. 39-48.