Comparison of Collision Detection Algorithms

By:	W Anthony Young
	20161423
Date:	July 30th, 2004
For:	Prof. E. Chan
Subject:	CS 741 - Spring 2004

1) Introduction

Collision detection is the processing of the bounds of two objects to determine if they intersect at any time, t. Due to the complexity of processing 3D shapes inside a computer, collision detection is a difficult problem to solve in constant or linear time. However, this difficulty does not deter programmers from using collision detection algorithms to solve many real world problems. The three major application areas for collision detection algorithms are safety, research and entertainment.

One application of collision detection to safety is to aid collision avoidance in maritime, air and land vehicle navigation [2,4]. These systems must be able to interpolate when a collision between two vehicles is likely to take place. Often, transponders can be used to represent an object's position in space. Then, by extrapolation of object geometries, position, acceleration and velocity, a system is capable of determining if a crash will occur. It can then notify the vehicle navigators of the expected time and location of a crash. Navigators can then take measures to correct their course to avoid collisions.

A second application of collision detection to safety is to automatically notify emergency response crews of vehicle collisions. A tracking system would be able to notify ambulance and fire crews of the location of an accident when it occurs. The tracking system would also be able to report the speed at which vehicles were traveling, the angle of impact, etc. This information would be available to emergency crews in real time to help save lives. That way, if the people involved in the accident were unable to call for help, crews could be dispatched before a passer-by was able to get to a phone.

The major application of collision detection to research is through the use of interactive simulation and modeling. When objects are in motion, they may collide. Scientists often want to study the effects of these collisions on the objects involved in order to determine how they behave before, during and after collisions occur. To make this study possible, an algorithm must be able to precisely detect when and where collisions would take place. That way, more accurate models may be built and more accurate simulation results may be obtained.

One application of collision detection to entertainment is through graphics rendering for game engines. When a user is playing a game, they expect the characters and objects to behave as though they were in the real world. For this to happen, objects must be able to detect when they collide with each other. As such, collision detection algorithms are often a necessity in computer gaming.

A second application of collision detection to entertainment is to scene animation [7]. When companies such as Dreamworks are creating a movie, they want character and object movements within a scene to appear real. As such, animation software must be able to detect when objects collide with each other in order to make, for example, a ball bounding off a wall appear realistic. Animation relies quite heavily on collision detection to properly model clothing, objects falling or breaking, etc. As with computer games, viewers expect realistic things to happen in response to some collision, and this requires accurate collision detection.

Although the three main application areas of collision detection are very different, they are all areas that are very important to our current value system. North Americans quite highly value safety, entertainment, and research (the foundation of science and technology). As such, all three areas of application are very important. The main reason for advancement in the development of collision detection algorithms is a longing for better realism.

2) Collision Detection Algorithms

Now that some introduction has been given to the motivation for accurate collision detection, some collision detection algorithms will be presented and evaluated. This paper will discuss four different algorithms: bounding box, bounding sphere, binary space partitioning tree, and space-time bounds (Hubbard).

2.1) The Bounding Box Algorithm

One method of collision detection makes use of bounding boxes. An object's bounding box is simply a cube of minimal volume that encloses the entire object [6]. This cube is fairly simple to build as the maximum and minimum coordinates of an object can be obtained and used in the cube's construction. Then, intersections between bounding boxes of various objects in the scene can be tested using simple algebra. If the two are found to intersect, a collision is reported.

The intersection testing mechanism for two bounding boxes is quite simple. Each bounding box has six faces. In order for two boxes to intersect, at least one of the faces of each box must intersect. Therefore, simple algebra can be employed to tell if any of the 36 combinations of two object's faces intersect.

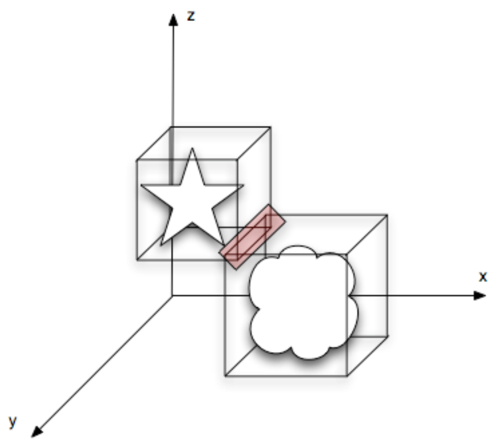


Figure 1: Example Bounding Boxes

Figure 1 depicts two possible bounding boxes for objects in 3D space. The shaded area is the area of collision between the two. In this example, a collision would be reported because two faces of the cubes cross. However, it is clear from figure 1 that the two objects do not actually intersect. Although this algorithm is not very accurate, it does have practical uses. For example, this algorithm could be used in a situation where objects are approximately square, or where quick collision detection is required and accuracy is not.

The space complexity of the bounding box algorithm is quite good. This is because bounding boxes must only store 8 3D points representing the coordinates of the cube faces. Thus, space complexity for bounding boxes is O(8) and Ω(8). The time complexity of this algorithm is, in contrast, quite poor. This is because each face of each object must be tested against each face of each other object. For this reason, the bounding box algorithm experiences a time complexity of $O((6n)^2) = O(n^2)$ (where n is the number of objects in the scene).

The bounding box algorithm provides very coarse collision detection due to the highly conservative representation of each object as a cube. As well, the algorithm executes very slowly when there are many objects in the scene. This is because it must test face intersections between all possible combinations of faces. However, bounding box collision detection is very easy to implement, and does not require large amounts of memory to store geometry and other information. Thus, when collision detection is required with very little overhead, the bounding box algorithm is a possible solution.

2.2) The Bounding Sphere Algorithm

Bounding spheres can be used for collision detection in a manner similar to bounding boxes [3]. Collision detection is in fact easier to perform with a bounding sphere as opposed to a bounding box. This is because each object only has to store one equation to represent its bounding sphere. Then, simple algebra can test for intersections between each sphere in the scene.

Initially, each object must have a bounding sphere generated for it. This requires the algorithm to decide on a center point for the object that will minimize the sphere radius. This can be tough as the chosen point must minimize radius in all directions. Further, as with bounding boxes, bounding spheres are very conservative estimates of an object's volume and thus, still only provide very coarse collision detection.

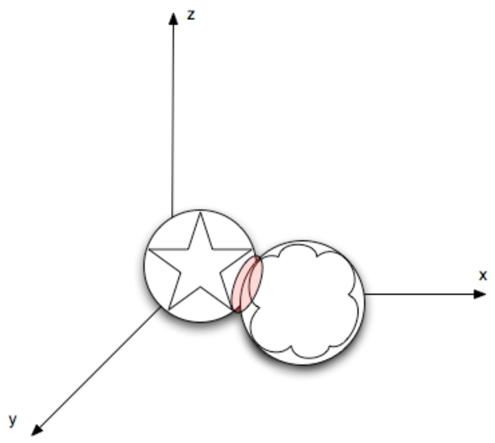


Figure 2: Example Bounding Spheres

Figure 2 shows an example intersection between two bounding spheres. As can be seen from figure 2, intersection of the two objects is reported because the two bounding spheres intersect at the location of the shaded ellipse. Similar to the bounding box algorithm, many false collisions (positives) may be reported between two bounding spheres where the enclosed objects might not actually intersect.

The space complexity of the bounding sphere algorithm is very good, and is actually better than that of the bounding box algorithm. This is because each object must only store two values: a radius and a center. These two values are enough to represent a sphere. Alternatively, the algorithm could store the equation of the bounding sphere while still keeping space complexity low. Therefore, the bounding sphere algorithm experiences a space complexity of $O(2)$ and $\Omega(2)$ (nb: the space required to store the values for an equation is the same as the space required to store the radius and center). The time complexity of the bounding sphere algorithm is still $O(n^2)$ (where n is the number of objects in the scene), which is quite poor. This is because, in the worst case, each sphere would need to be tested for intersection with each other sphere in the scene. However, fewer comparisons would be required than with bounding boxes. This is because only one intersection test must be performed per pair of objects instead of 36.

As with the bounding box algorithm, the bounding sphere algorithm implements very coarse collision detection. This is due to the fairly conservative representation of the object as a sphere. However, representing an object as a sphere is less conservative than representing it as a cube (except in special cases). As well, the bounding sphere algorithm executes quite slowly when there are many objects in the scene. However, significantly fewer calculations are performed to check for intersections between spheres than to check for intersections between cubes. This algorithm is even easier to implement than the bounding box algorithm due to the reduced number of calculations. Also, less space is consumed to store sphere objects than to store cube objects due to the smaller number of values required to represent a sphere.

2.3) The Binary Space Partitioning Tree Algorithm

Binary space partitioning (BSP) trees are used to break a 3D object into pieces for easy comparison [1]. To construct a BSP tree, the space occupied by an object is recursively broken into smaller pieces and inserted into a tree (quadtree for 2D objects, octree for 3D objects, etc). Traversing the various levels of two BSP trees can then check for intersections between the spaces two objects occupy. Each BSP tree contains a set of nodes. Internal nodes represent a certain portion of the space that an object resides in, and contains the portion of the object that resides in that space. The external (or leaf) nodes of the tree hold the polygons that make up the object. In this manner, intersections can be tested at a very fine grain with the individual polygons themselves.

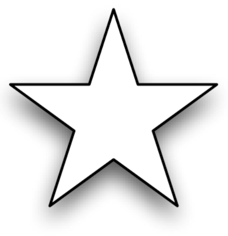
When constructing a BSP tree, the object model is refined by recursively defining the model at tree level i+1 to contain a subset of the object's space and geometry stored at tree level i. The algorithm stops refining the model when it reaches one of a few cases:

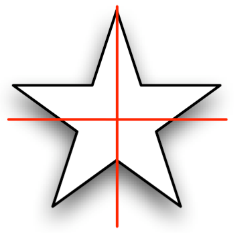
Case 1: The pre-defined minimum physical size for the section of space that is being modeled has been reached. For example, when the size of the section of space reaches 1 pixel cubed, ten pixels cubed, etc. This allows the implementer to assign a physical scale of accuracy to their collision detection.

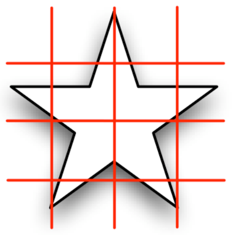
Case 2: A maximum tree depth has been reached. For example, the BSP tree reaches 10 levels, 20 levels, etc. This allows the implementer to ensure a maximum processing time for their collision detection.

Case 3: Each polygon used to define an object has been placed in a leaf node. For example, each leaf node in the BSP tree contains exactly one polygon from the object geometry. This allows the implementer to ensure that a small number of calculations need to be performed at the leaf level in order to detect a collision.

Etc: The exit case for constructing a BSP tree depends on the implementer. However, the above exit cases are the ones most frequently used when creating BSP trees.


3a. Level 0 (Root) of a BSP Tree


3b. Level 1 of a BSP Tree


3c. Level 2 of a BSP Tree

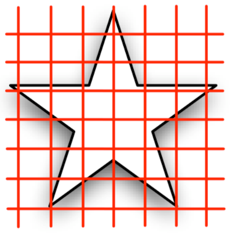
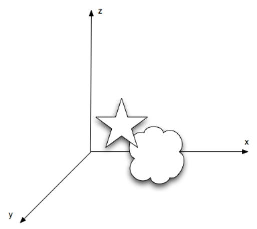

3d. Level 3 of a BSP Tree

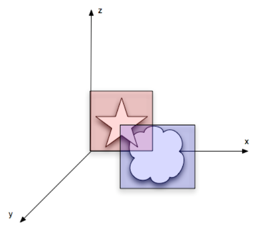
Figure 3: Partitioning of a 2D Shape for a BSP Tree

Figure 3 shows the gradual partitioning of a 2D object (in this case a star) into sections to be put into a BSP tree. As can be seen from 3a, the root level of the tree contains the entire object enclosed in a bounding box. Level 1 contains the object split into four quadrants as seen in 3b, etc. In the example, this partitioning continues to level 3, providing a tree depth of 4. (nb.: this tree is a quadtree as each node has four children. A 3D object would be stored in an octree).

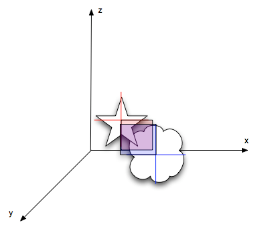
Once two BSP trees are constructed, they can be tested for collisions. Collision detection with BSP trees proceeds recursively by checking if successive levels of the trees intersect. If the two intersecting tree spaces being tested contain polygons, and the intersecting spaces are in internal nodes, the algorithm assumes an intersection occurs and processes the next level. If the two intersecting tree spaces being tested contain polygons, and the intersecting spaces are in leaf nodes, the algorithm tests the actual polygons for intersection and returns the result. If one of the tree sections being tested does not contain polygons, the algorithm can assume that no intersection occurs between the two objects.



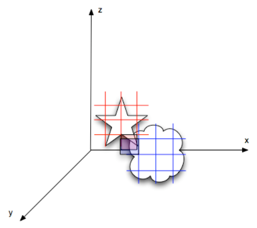
4a. Intersection Test Between Two Objects in 3D



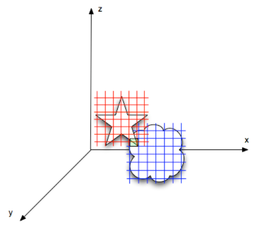
4b. Intersecting Root Level of Two BSP Trees



4c. Intersecting 1st Level of Two BSP Trees



4d. Intersecting 2nd Level of Two BSP Trees



4e. Intersecting 3rd Level of Two BSP Trees

Figure 4: Collision Detection Using BSP Trees

Figure 4 illustrates an example collision test between two objects with overlapping root levels. In this case, intersection testing proceeds to level 3 where an intersection between polygons is found in the shaded portion of the image shown in 4e.

With the BSP tree algorithm, each object must store a BSP tree. The leaves of this tree contain polygons with geometries stored as integer coordinates in X-space (i.e.: 2-space, 3-space, etc). Therefore, the space required by each object depends on the height, h, of the BSP tree that stores it, and the number of polygons, n, that make up the object (nb: assume objects are composed of convex triangles as these are the most common polygons used to model objects). For this reason, the amount of space consumed by each object in memory is $O(X^h + 3n)$ and $\Omega(X^h + 3n)$.

Using the BSP tree algorithm, each object must still be tested for collision with each other object in the scene. And, as with bounding boxes, face intersections must be tested for each object space. Thus, to determine if further processing is required, $O(n^2)$ work must be done to find intersecting objects at the coarsest level. Then, if an intersection is found between two objects, further testing must be performed on the various levels of their BSP trees. Thus, an additional O(h) tests must be performed between spaces in the BSP trees. Therefore, the final time complexity of this algorithm is, in the worst case, $O(n^2 + m*h)$ if m intersections are found, and, in the best case, $\Omega(n^2)$ if no intersections are found, assuming that each BSP tree has the same height, h.

The BSP tree collision detection algorithm is quite complex to implement. This is due to the complexity involved in constructing BSP trees, and the number of intersections that must be tested. As well, this algorithm is still very slow to process scenes with large numbers of objects, and consumes significantly more space than both the bounding box algorithm and the bounding sphere algorithm. However, collision detection with BSP trees does provide fine-grain collision detection, as opposed to the very coarse grain detection provided by the other two.

2.4) Space-Time Bounds

Hubbard, et al, proposed an algorithm that makes use of “sphere trees” and “space-time bounds” to iteratively refine collision detection accuracy proportional to the amount of processing time an application implementing the algorithm can spare [4]. Hubbard’s algorithm works in two phases: the broad phase constructs space-time bounds and does coarse grain comparisons; the narrow phase refines detection using sphere trees to determine if a collision detected in the broad phase really does occur.

The broad phase of Hubbard's algorithm constructs space-time bounds (4D structures that give a conservative estimate of an object's position over time) and tests them to find intersections at time t_i between object O_x and O_y. As well, the algorithm looks only for the earliest time when an intersection between any two objects occurs, and can stop processing any collision detection until the scene reaches that time. Once the scene advances to time t_i, the algorithm reruns the broad phase to determine if an intersection still occurs between bounds. If it does, the intersecting objects are referred to the narrow phase for processing. If it does not, the broad phase finds the next intersection and continues.

The narrow phase of Hubbard's algorithm seeks to iteratively refine intersection testing accuracy using better approximations to objects as stored in lower sphere tree levels. Sphere trees are simple tree structures that store a series of overlapping spheres that completely enclose an object. Spheres in each successive level of a sphere tree better approximates the object. Thus, the algorithm can recursively check for intersections between two objects while only processing small parts of the sphere trees. Each time the narrow phase finishes processing a level of two object's sphere trees, it allows the application to interrupt and stop processing. This allows the application to receive detection accuracy proportional to the amount of processing time that it can spare. The narrow phase will exit on one of three cases:

Case 1: The system interrupts the narrow phase after it processes a given level, i.

The returned result of detection is the result currently obtained after processing level i of the object's sphere trees. This may lead to the report of a false positive (i.e.: a collision that exists at level i of a tree, but not at level i+1, i+2, etc).

Case 2: The narrow phase finds no intersection at a sphere tree level. The result of detection is, intuitively, false.

Case 3: The narrow phase reaches the leaf level of the sphere trees and still detects a collision. Thus, the result of detection is, intuitively, true.

•Start System
–Construct Sphere Trees
–Run scene
•Broad Phase on Frame 1
–Calculates space-time bounds
–Finds intersection - O_1 and O_2 at $t_i = 5$
•Run scene to Frame 5
•Broad Phase on Frame 5
–Detects no collision on bounding spheres
–Calculates space-time bounds
–Finds intersection - O_2 and O_3 at $t_i = 7$
•Run scene to Frame 7
•Broad Phase on Frame 7
–Detects collision on bounding spheres
•Narrow Phase on Frame 7 for O_2 & O_3
–Detects collision on level 1 spheres
–Detects collision on level 2 spheres
–Detects no collision on level 3 spheres
•Exit case 2
•Broad Phase on Frame 8
–Calculates space-time bounds
–Finds intersection - O_3 and O_4 at $t_i = 10$
•Run scene to Frame 10
•Broad Phase on Frame 10
–Detects collision on bounding spheres
•Narrow Phase on Frame 10 for O_3 and O_4
–Detects collision on level 1 spheres
–Detects collision on level 2 spheres
–Detects collision on level 3 spheres
–No more levels - collision detected
•Exit case 3
•Broad Phase on Frame 11
–Calculates space-time bounds
–Finds intersection - O_4 and O_5 at $t_i = 12$
•Run scene to Frame 12
•Broad Phase on Frame 12
–Detects collision on bounding spheres
•Narrow Phase on Frame 12 for O_4 and O_5
–Detects collision on level 1 spheres
–Detects collision on level 2 spheres
–Detects collision on level 3 spheres
–Interrupted - collision currently detected
•Exit case 1

Figure 5: Example Sequence of Events for Hubbard's Algorithm

Figure 5 shows an example sequence of events in an application implementing Hubbard's algorithm. This example sequence shows the use of all three exit cases (at frames 7, 10 and 12) for the narrow phase, and how narrow phase detection can be called off (at frame 5) if broad phase detection no longer finds a collision.

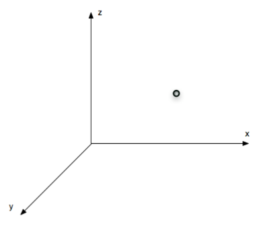
$$\boxed{|\, a(t) \,| \leq M \text{ until } t = t_j}$$

Inequality 1: Bounding of an Object's Acceleration

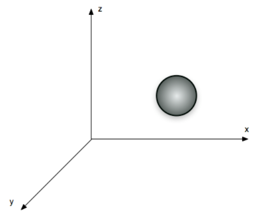
$$\boxed{|\, p_t - [p_0 + v(0)t] \,| \leq (M / 2)t^2}$$

Inequality 2: Bounding of an Object's Position

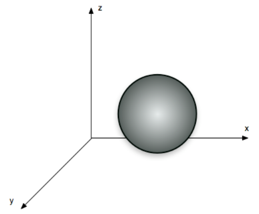
As mentioned above, Hubbard's algorithm makes use of space-time bounds. Space-time bounds are 4D structures that represent a conservative estimate of the possible 3D location of an object over time, the fourth dimension. The application providing the object data thus knows the object's position, p = (x,y,z), its velocity vector, v(t) = (x,y,z), its acceleration vector, a(t) = (x,y,z), and a value M, such that inequality 1 holds. When looking at inequality 1, notice that the magnitude of an object's acceleration in any direction cannot be greater than the value M. This leads to the observation stated in inequality 2. When looking at inequality 2, notice that the affect of an object's acceleration over time cannot put its position more than $(M / 2)t^2$ away from the position it is supposed to be in. Thus, the object's position is inside a sphere of radius $(M / 2)t^2$ centered at the point $p_0 + v(0)t$. This is because the object's acceleration over time could not have increased or decreased the object's velocity enough to move it more than $(M / 2)t^2$ off of the current calculated position.



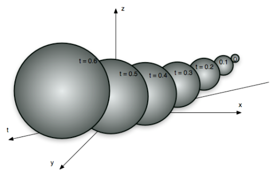
6a) An Object's Possible Position at t = 0



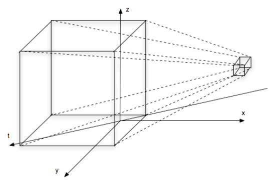
6b) An Object's Possible Position at t = 0.1



6c) An Object's Possible Position at t = 0.2



6d) An Object's Possible Position Between $0 \leq t \leq 0.6$



6e) A Possible Enclosing Hypertrapezoid for an Object's Position Between $0 \leq t \leq 0.6$

Figure 6: Bounding an Object's Position Over Time

Figures 6a – 6d show the bounding spheres of size $(M / 2)t^2$ that bound an object's position over time $0 \leq t \leq 0.6$. Figure 6e shows a hypertrapezoid that would contain, conservatively, all bounding spheres of size $(M / 2)t^2$ for an object between $0 \leq t \leq 0.6$. Thus, the object is guaranteed to be within this hypertrapezoid from $0 \leq t \leq 0.6$. This approximation obviously results in large amounts of conservatism since the enclosing hypertrapezoid bounds areas that an object could never occupy. If the application knows that an object's acceleration is limited to a certain directional vector, d(t) = (x,y,z), the algorithm can generate a "cutting plane". Vector d(t) says that an object will never head in a certain direction, and a cutting plane allows the algorithm to remove the space (behind d(t)) from a hypertrapezoid that an object will never occupy. Thus, a cutting plane allows the algorithm to reduce the size of the enclosing hypertrapezoid and make more accurate broad phase collision detections. However, the application must also provide the algorithm with the vector, d(t).

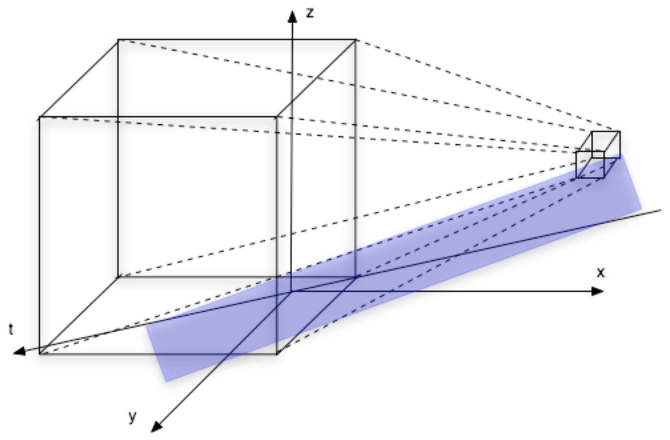


Figure 7: A Complete Space-Time Bound

A complete space-time bound consists of an application provided M, d(t), a calculated hypertrapezoid, T, and a calculated cutting plane, P. Figure 7 provides a graphic representation of a possible space-time bound for an object. The cubes represent T, and the blue plane represents P.

Space-time bounds are interesting structures, but they serve a very important purpose. If two space-time bounds intersect, a very coarse level collision has been detected. The intersection of two space-time bounds, B_1 and B_2, can happen in three ways:

Case 1: A face, f_1, of T_1, intersects a face, f_2, of T_2. In this case, two faces of two hypertrapezoids have intersected.

Case 2: A face, f, of T_1 intersects a cutting plane, P, of T_2. In this case, a face has intersected a cutting plane. Since an object can never be behind a cutting plane, the algorithm only cares about the part of f that is in front of P. In order to get in front of P, f must intersect another face. Thus, case 2 has been reduced to case 1.

Case 3: A cutting plane, P_1, of T_1 intersects a cutting plane, P_2, of T_2. In this case, two cutting planes have intersected. As in case 2, an object can never be behind a cutting plane. Thus, the algorithm is only concerned with the part of P_1 that is in front of P_2, and vice versa. As in case 2, to get in front of a cutting plane, a face from T_1 must intersect a face from T_2. Thus, case 3 has been reduced to case 1.

Three intersection cases have been reduced to one important case. So, the algorithm is only concerned with the intersection of two faces.

Each face, f, of a hypertrapezoid, T, is "normal" to one of the planes of the coordinate system

Axiom 1

Each face, f, is included in a face set, F_a
$F_a = \{ f \mid f \text{ is normal to the a-t plane}\}$, a in {x, y, z}

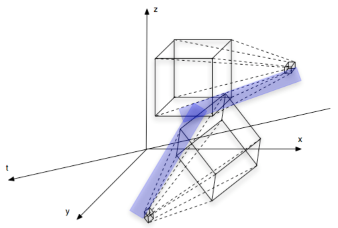
Axiom 2

Each intersection takes place between two faces in the same face set, F_a

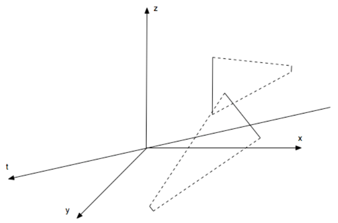
Axiom 3

Finding intersections between faces relies on axioms 1 to 3 listed above. Axioms have already been proved by Hubbard and will not be proven here. Axioms 1 to 3 provide an important optimization: in order to find intersections between hypertrapezoid faces, the algorithm must only test for intersections between faces within a normalized face set. Thus, the algorithm only has to test for intersections between all faces in F_x, then F_y, and finally F_z. This significantly reduces the number of intersection tests that need to be performed.

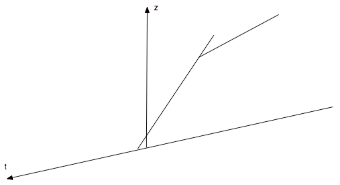
Intersections between faces can be found using one of two methods: projection or subdivision. With projection, hypertrapezoid faces are projected onto the a-t plane and tested for intersections with other faces in the same face set. With subdivision, cross-sections of the hypertrapezoid are tested for intersection recursively until one is found, or until the algorithm reaches some predefined Δt.



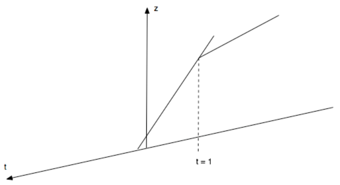
8a) Space-Time Bounds that Might Intersect



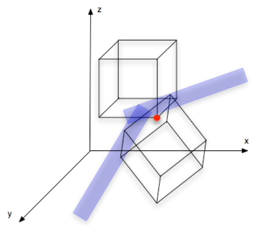
8b) Faces of the Hypertrapezoid



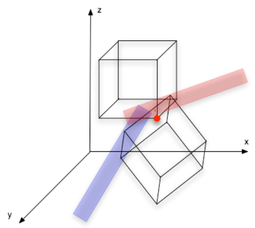
8c) Faces Projected onto t-z Plane



8d) Intersection Found at t = 1



8e) Intersection of Cube Cross-Sections



8f) Intersection of Cube Cross-Sections Below the Cutting Plane

Figure 8: Intersection Testing With Projection

The projection method of intersection testing follows a set of four steps:

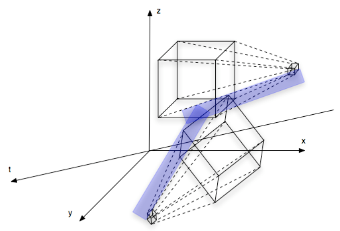
Step 1: Project each face in a face set, F_a, onto the a-t plane. Figure 8b shows two faces in the F_z set that are about to be tested for intersection (nb: these two faces belong to the hypertrapezoids in figure 8a). The two faces are projected onto the z-t plane in figure 8c. As 8c shows, faces appear as 2D lines in the plane. Intersection of these lines is necessary but not sufficient for an intersection of two faces to take place. This is because the hypertrapezoids must cross paths at time t_i (necessary condition). But, the algorithm must also check if they occupy the same space at time t_i (sufficient condition).

Step 2: Find intersections between 2D line segments. Using trivial mathematics and linear algebra, the algorithm notes an intersection between two line segments. The two projected faces shown in figure 8c are drawn with an intersection point in figure 8d. This intersection point, I, is placed in an intersection set. I contains the faces that intersected as well as the point, t, on the t axis where they crossed.

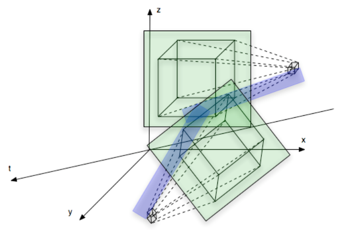
Step 3: Check for intersections, at time t, of cube cross-sections of the hypertrapezoids that the faces in I belong to. Using trivial mathematics and linear algebra, the algorithm tests the cube cross-sections of the two hypertrapezoids for intersection. If the two cubes intersect each other, the algorithm has detected a coarse collision at time t, and keeps this intersection in the intersection set. If the cubes do not collide, the intersection is removed from the intersection set. The cube cross-sections of the hypertrapezoids shown in figure 8a are tested for intersection at time t=1 as shown in figure 8e.

<u>Step 4</u>: Determine if the point of intersection is behind a cutting plane for either hypertrapezoid. Using trivial mathematics and linear algebra, the algorithm can check if the intersection of the two hypertrapezoids takes place behind either of the cutting planes. If it does, as in figure 8f, the intersection is removed from the intersection set. If not, the intersection is retained.

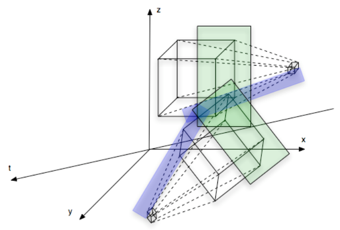
Once the algorithm has found all intersections between space-time bounds, processing can begin with the earliest intersection in the intersection set.



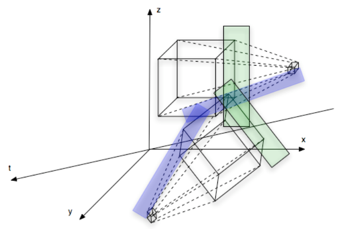
9a) Space-Time Bounds that Might Intersect



9b) 1st Subdivision of Bounds



9c) 2nd Subdivision of Bounds

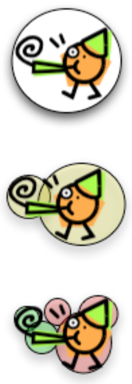


9d) 3rd Subdivision of Bounds (Δt Reached)

Figure 9: Intersection Testing With Subdivision

The subdivision method recursively divides the (possibly) intersecting hypertrapezoids in half seeking the point in time when they collide. Intersection testing involves checking for intersections between any pair of faces belonging to the two hypertrapezoids currently being processed. If there is an intersection, the hypertrapezoid is cut in half and recursively tested again. The exit case is the time, t, at which the two faces intersect, or when the size of a time slice is smaller than some threshold, Δt, provided by the application. Figure 9 shows a possible set of subdivisions. Figure 9a shows the original hypertrapezoids. 9b – 9d highlight the section of the two hypertrapezoids that was found to intersect. The exit case is reached in 9d when the size of the time slice reaches Δt.

The projection method of intersection testing is more difficult to implement than the subdivision method. However, subdivision is prone to suffer false positives due to a bad Δt value, and does not save future processing by discarding intersections behind the cutting planes. Also, the projection method executed faster during empirical testing. Tests consisted of 200 sets of randomly distributed hypertrapezoids of varying parameters (i.e.: geometry, velocity, etc.). Thus, Hubbard's algorithm implements the projection method of intersection testing to find intersections between space-time bounds.


10a) Root Level (Level 0) of a Sphere Tree

10b) Level 1 of a Sphere Tree

10c) Level 2 of a Sphere Tree

Figure 10: Possible Sphere Tree for an Object

As mentioned above, sphere trees are used for fine-grain intersection testing in the narrow phase. A sphere tree is a constant refinement of a bounding sphere for an object.

In a sphere tree, an object is represented as a set of overlapping spheres that contain the object's geometry. The overlapping spheres are constantly refined into larger sets of smaller spheres until the application's requested accuracy level is attained. So, the spheres at level i+1 of the sphere tree approximate the object more accurately than the spheres at level i. Spheres are used because of the ease with which they may be compared. Figure 10 shows a possible sphere tree for an object. 10a shows the bounding sphere for the object that is stored at the root level of the tree (i.e.: level 0). 10b shows level 1 of the sphere tree, etc. The spheres in 10b more closely approximate the object than the bounding sphere in 10a. The spheres in 10c more closely approximate the object than the spheres in 10b. This object has a sphere tree with three levels. Accuracy of a sphere tree, or "tightness of fit", is important to ensure that the algorithm performs the most accurate collision detection possible at each level of processing. Since the application can interrupt the narrow phase at any time, the algorithm aims to have the most accurate collision detection possible after each iteration.

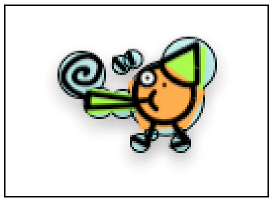
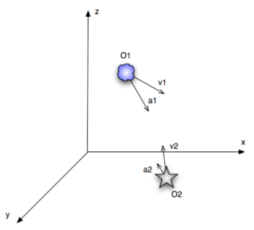
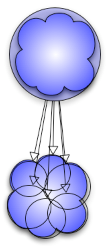


Figure 11: First Iteration of the Medial-Axis Surface Algorithm

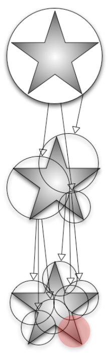
Construction of a sphere tree is a form of multi-resolution modeling. Multi-resolution modeling is a complex task that is tough to automate efficiently. The medial-axis surface method is one method used to construct sphere trees. A medial-axis surface corresponds to the "skeleton" of an object. For 3D objects, this surface is very difficult to build. Thus, the algorithm works in reverse in order to generate a sphere tree of tightest fit. The algorithm starts by covering the object with very tightly fitting spheres first, as shown in figure 11. These spheres cover the maximum volume of the object while minimizing the wasted space in each sphere. They also make up the leaf level of the sphere tree. Then, adjacent spheres are combined into larger spheres and inserted into the sphere tree one level above. The algorithm exits when all spheres have been combined into one bounding sphere at the root of the tree. In this manner, the sphere tree is effectively constructed bottom-up. Unfortunately, this algorithm takes a large amount of processing time. In one test, the algorithm took 12.4 minutes to generate the sphere tree for an object composed of 626 triangles. However, this processing must only be done once to construct the sphere tree, as the tree can be reused at each iteration of the narrow phase.



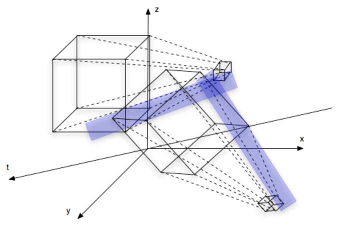
12a) A Sample Scene



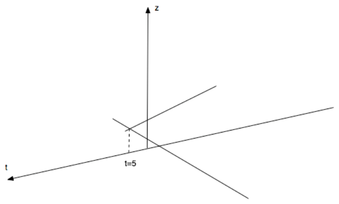
12b) A Sphere Tree for O_1



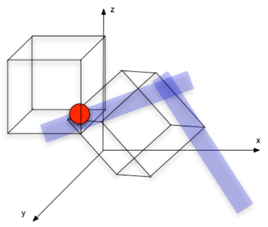
12c) A Sphere Tree for O_2



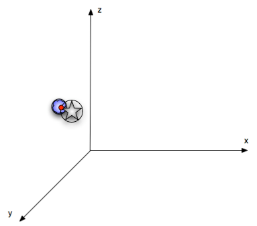
12d) Initial Space-Time Bounds for O_1 and O_2



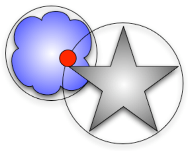
12e) Initial Intersection for O_1 and O_2 is at t_i=5



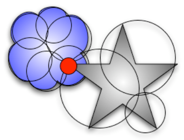
12f) Initial Intersection Above Cut Planes



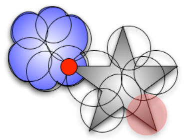
12g) Scene at $t_i = 5$



12h) Intersecting Root Level Sphere Trees



12i) Intersecting Level 1 Sphere Trees



12j) Intersecting Level 2 Sphere Trees

Figure 12: Processing of a Scene

Now that all portions of Hubbard's algorithm have been discussed, perhaps a brief example scene will aid the reader in understanding how the various pieces fit together. When the algorithm starts, the object models and values of M and d(t) are taken as input. From this information, the system generates a sphere tree for each object. The scene is then advanced to the first frame, and the broad phase is run. The broad phase calculates the hypertrapezoid and cutting plane for each object. Then, the time, t_i, of the first intersection is calculated and the scene is run to that frame. Next, broad and narrow phase testing are run as outlined above. Figure 12 presents a possible scene processing with Hubbard's algorithm. Figure 12a shows the initial objects with their velocities and accelerations. 12b and 12c show the constructed sphere trees for the two objects. 12d shows the set of initial space-time bounds for the two objects, and 12e shows the projection of the two intersecting faces onto the z-t plane. An intersection is noted between O_1 and O_2 at time t_i=5. 12f shows that the intersection between the cube cross-sections is above the cut planes, and the intersection is retained. The scene is then advanced to time t=5. The space-time bounds are recalculated and found to intersect. The objects are then referred to the narrow phase for further processing. Figure 12g shows the positions of the two objects at time t=5. 12h shows an intersection between the root level of the sphere trees. 12i shows an intersection between level 1 of the sphere trees. 12j shows an intersection between level 2 of the sphere trees. Since no more levels exist in the object's sphere trees, a collision is reported to the application at time 5 between object O_1 and O_2. Note that the application could have interrupted processing at any stage and a collision would still be reported as one was detected at each level of processing. Also, the narrow phase could have exited if no intersection was found at a lower level of the sphere tree. Since there were no lower levels to process, a collision was reported.

	All cases	
Level	Number	Mean Speedup
1	50042	1049.3
2	5079	581.6
3	2180	243.2
4	915	112.8

5(exact)	483	2.7

Figure 13: Mean Speedup Experienced by Hubbard's Algorithm

Hubbard's algorithm was empirically tested against Turk's method of collision detection. Turk's method tests for intersections between spheres stored in a BSP tree without the use of space-time bounds. The empirical tests showed that if collisions are found early in a scene (i.e.: before 0.25 seconds have elapsed), Turk's method detects them faster. Otherwise, Hubbard's algorithm experiences a detection speed boost of approximately 10 times over Turk's algorithm. Figure 13 shows the mean speedup experienced by Hubbard's algorithm, in percent speed increase, for all collision detections that reach level x processing of the sphere or BSP trees. The reduction in mean speedup at lower levels of tree processing is not addressed by Hubbard in "Collision detection for interactive graphics applications". Also, accuracy of detection is not discussed or compared.

Hubbard's algorithm must store a hypertrapezoid (16 4D points), a sphere tree (X^h links and X^h spheres, where X is the maximum fan-out at any sphere tree level), a set of bounding values (M and d(t)), and the object's description (p(t), v(t), a(t)). Then, the space required to store an object is $O(2(X^h) + 23)$ and $\Omega(2(X^h) + 23)$.

During the broad phase, Hubbard's algorithm must compare each face to each other face in a given face set. In the best case, the first test will yield an intersection between two faces in the very next frame. This means that the broad phase needs to execute $\Omega(1)$ operations. However, in the worst case, the broad phase must do $O(3(2n)^2) = O(n^2)$ comparisons (where n is the number of objects in the scene). During the narrow phase, Hubbard's algorithm must compare spheres in the sphere tree for two colliding objects. In the best case, only one comparison is required to tell that no intersection has taken place. Thus, the algorithm requires only $\Omega(1)$ comparisons. In the worst case, the entire sphere tree must be traversed in order to find or discount an intersection. Thus, the algorithm requires O(h) comparisons (where h is the height of the sphere tree), again assuming that each sphere tree has the same height. So, in the best case, the algorithm has to perform only $\Omega(2)$ comparisons for each run of the collision detection algorithm. But, in the worst case, the algorithm must perform $O(n^2 + h)$ comparisons.

Hubbard's algorithm allows average-case real-time collision detection due to its ability to avoid processing during intermediate frames. It also experiences very fine grain collision detection with the use of sphere trees. Hubbard's algorithm is empirically faster than Turk's, which was previously thought to provide the fastest and most accurate collision detection. However, Hubbard's algorithm may experience false positives if applications interrupt processing too early in the narrow phase, constructing sphere trees is very slow.

3) Conclusion

The bounding box algorithm is efficient in space but not in computation time. Also, the bounding box algorithm provides very inaccurate collision detection and cannot run in real-time. In contrast, the bounding sphere algorithm is more efficient in space and time than the bounding box algorithm. Also, the bounding sphere algorithm slightly improves the accuracy of detection, but still does not run in real-time.

The BSP tree algorithm is not very efficient in space or time, and does not run in real-time. However, the BSP tree algorithm is able to provide quite accurate collision detection. In contrast, Hubbard's algorithm provides very accurate collision detection while being relatively efficient in space consumption. However, Hubbard's algorithm is only real-time in the average case. When the number of objects in the scene increases, Hubbard's algorithm slows.

The "best" collision detection algorithm depends on the application. Bounding box and bounding sphere algorithms are useful for scenes containing objects of that approximate shape. Also, for the faster detection required by primitive games or primitive scene generation, these algorithms might be ideal. BSP tree algorithms are useful in a scene that contains objects that are not known to the system ahead of time (i.e.: auto collision, etc). This is because the BSP trees can be constructed on-the-fly if necessary as the objects are being processed. Hubbard's algorithm performs best for most real-world applications where objects are known to the system ahead of time (i.e.: games, simulations, animation, Air Traffic Control, etc.). This is because object models can be used to quickly and very accurately detect collisions on a fine grain scale.

Collision detection is a difficult problem for a computer to solve efficiently. Many different strategies exist to perform accurate detection or detection in near real-time. Detection algorithms have not progressed significantly over recent years. This is mainly due to the fact that most researchers are now using image processing techniques, instead of object models, for collision detection.

References

1. Cornell. Binary Space Partitioning Trees FAQ. Accessed June 19th, 2004 from http://www.faqs.org/faqs/graphics/bsptree-faq/
2. De Micheli, E., Lorusso, A. An approach to obstacle detection and steering control from optical flow. In Proceedings of the 1996 IEEE Intelligent Vehicles Symposium, September 1996.
3. Guibas, L. J., Kim, D., Shin, S. Fast collision detection among multiple moving spheres. In IEEE Transactions on Visualization and Computer Graphics 4(3), Sept 1998.
4. Hubbard, P. M. Collision detection for interactive graphics applications. In IEEE Transactions on Visualization and Computer Graphics, September 1995.
5. Smith, A.A., Teal, M.K. Identification and tracking of maritime objects in near-infrared image sequences for collision avoidance. In Proceedings of Seventh International Conference on Image Processing and its Applications, July 1999.
6. Suri, S., Zhou, Y. Analysis of a bounding box heuristic for object intersection. Journal of the ACM (JACK) 46(6), November 1999.
7. Yuen M. M. F., Zhang, D. Collision Detection for Clothed Human Animation. In Proceedings of Pacific Graphics 2000, October 2000.