Intersection Detection of Two Ellipsoids

Jeff Duzak

March 12, 2007

Introduction

An essential problem for 3-dimensional simulations and games is detecting collisions between objects. For example, a bowling simulation must detect when the bowling ball collides with a pin, or two pins collide with each other. Typically, 3d objects will be approximated using simple primitive objects, such as polyhedra and ellipsoids. A bowling ball is simply a sphere, and a pin can be approximated as two ellipsoids. An ellipsoid itself can be constructed by applying scaling, rotation, and translation operations to a unit sphere. Therefore, an algorithm that could quickly detect the intersection of two spheres under scaling, rotation, and translation operations would be very useful. The purpose of this research is to find such an algorithm.

The literature contains a wealth of algorithms for intersection detection. These algorithms vary with respect to the types of shapes supported, with some algorithms supporting completely arbitrary shapes, some supporting broad classes of shapes such as quadric surfaces, and some approximating complex shapes as triangle meshes. Further, the goals of the algorithms vary. Some algorithms are only interested in detecting whether or not there is an intersection. Others are concerned with computing the shape of the intersection. Others are concerned with detecting collisions over a window in time between moving objects. This report will review these algorithms, and discuss their application to our central problem.

Last, this report will describe in detail an algorithm using an adaptation of Newton's method of finding roots of a continuous function. We will discuss the implementation of this algorithm, and investigate its computational complexity.

Known Methods

There are a large number of algorithms for intersection detection of 3d objects, each useful for a particular domain of problems. A 2004 SIGGRAPH course [HA04] was entirely devoted to describing various intersection and collision detection methods. The course points out that there is a distinction between intersection detection, which checks if two objects occupy the same point in space at a static snapshot in time, and collision detection, which detects if two objects intersect at any point in a given window in time. A typical approach to collision detection is to partition a window in time into a number of static snapshots, and perform intersection detection on each of these. The number of snapshots would depend on the size and proximity of the objects and the speed at which objects move. However, the problem with this approach is that, at points in time where objects are close together and moving relatively fast, a very large number of snapshots are needed, and so the algorithm can virtually lock up. Further, in certain cases where objects just graze each other, it possible to miss collisions.

For certain limited, simple situations, it is possible to determine with certainty whether or not two objects collide in a given window of time without performing intersection detection a large number of times. For example, M. Gomez describes a simple method to detect the collision of two circles moving with constant velocities [GO99]. Such methods are very useful for 3-d games and simulations. In our example of a bowling simulation, it would be very useful to be able to, in a single calculation, determine the point in time at which the ball collides with a pin. However, these methods only cover very simple situations, such as collisions of two circles, and therefore can't be applied to the more complicated problem of the collisions of arbitrary 3-dimensional ellipsoids.

A large amount of research is devoted to detecting collisions between polyhedra. A polyhedron is composed of a number of polygon faces, each of which can be broken down into a number of triangles. Therefore, the problem reduces to intersection detection between triangles. Fujio Yamaguchi [YA85] describes how intersection detection of many simple objects, such as rays and triangles, reduces to computing the determinant(s) of one or more 4x4 matrices, and proposes to build hardware support for determinant computation as a means to detect intersections quickly.

Chazelle and Dobkin [CH87] describe an algorithm that can detect the intersection of two polyhedra in $O(\log^3 n)$ time, where n denotes the total number of vertices in the polyhedra. David Mount [MO92] describes an algorithm that can compute whether or not two 2-dimensional polygons intersect in $O(m \log^2 n)$ time, given some preprocessing on the polygons, where n again is the count of vertices, and m is the count of vertices in the simplest polygonal curve that separates the two polygons in question. It is not clear if a variation of Mount's algorithm can be applied to 3-dimensional objects.

Assuming we were to use a polyhedron method to detect intersections, we would have to approximate our ellipsoids as polyhedra. We want to detect intersections within a certain margin of error ε. Supposing that the largest radius of either ellipsoid is given by r, the value $k = \frac{r}{\epsilon}$ gives an estimate of the size of the problem. It is useful to know the complexity of a polyhedron intersection method in terms of k. Given r and ε, we can calculate approximately how small of triangles we would have to partition the ellipsoid into.

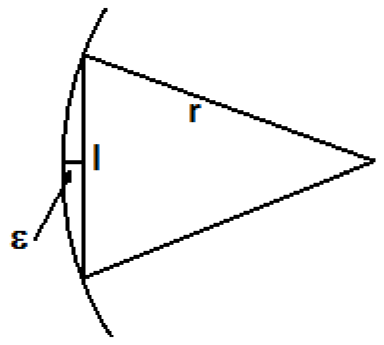

Figure 1. Approximation of an ellipsoid as a polyhedron, showing an approximate relationship between the length of the side of a face and the maximum error in the approximation

In the figure above, l denotes the length of the side of a face in the polyhedron, and, approximating the ellipsoid as a sphere, r is the radius of the sphere. The maximum distance from the polyhedron face to the ellipsoid is given by:

$$\epsilon = r - \sqrt{r^2 - \frac{1}{4}l^2}$$

The first non-zero term of the Taylor expansion for this expression gives us:

$$\epsilon \approx \frac{l^2}{8r}$$

This gives us the following expression for k:

$$k = \frac{r}{\epsilon} \approx \frac{8r^2}{l^2}$$

The surface area of the ellipsoid is proportional to r^2, and the area of a face is proportional to l^2, therefore the number of faces required is proportional to r^2 / l^2. Therefore, the number of vertices n is also proportional to r^2 / l^2, which itself is proportional to k, our measure of the size of the problem. Therefore, Chazelle and Dobkin's method can detect intersections with the desired accuracy in $O(\log^3 k)$ time.

Other research considers intersections of more general objects. A considerable amount of work has been done to compute the intersections of quadric surfaces [LE76, SA83, GO91]. A quadric surface is one defined by the equation:

$$[x \quad y \quad z \quad 1] M \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

Here, M is a 4x4 matrix. The matrix M completely describes the surface. Therefore, the matrix and the surface it describes are often referred to interchangeably. Conveniently, an arbitrary ellipsoid is a quadric surface. If a point lies on two quadric surfaces M and N, then it must satisfy the equation

$$[x \quad y \quad z \quad 1](M + \alpha N) \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

for any value of α. Certain properties of the matrix, such as its rank, can be used to deduce properties of the surface it describes. In fact, certain combinations of properties of the matrix indicate that the quadric surface is invalid, that no points satisfy the equation above. Therefore, the problem of determining whether two quadric surfaces intersect reduces to the problem of determining if $M + \alpha N$ is invalid for some α. If such an α exists, then no points satisfy the equation above. Therefore, no points lie on both quadric surfaces, and therefore the surfaces do not intersect.

The complexity of this algorithm is unclear, since it is unclear how to search for values of α that produce invalid quadric surfaces. Further, this algorithm is more concerned with determining the exact shape of an intersection between two quadric surfaces. Therefore, it is fairly heavyweight for the problem at hand, which is simply to determine whether or not two ellipsoids intersect.

Last, research by Paul Comba [CO68] allows for intersection detection of arbitrary objects. In his algorithm, an object is defined as the intersection of the sets of points which satisfy a number of continuous, differentiable inequalities $g_i(x, y, z) \leq 0$. Comba combines these inequalities into a single continuous, differentiable inequality such that any point that satisfies the inequality lies in both objects. Therefore, if such a point exists, the two objects intersect. So, the problem is reduced to that of finding the minimum value of a function. A variety of methods, such as Newton's method, can be used to find this minimum. In this way, Comba's algorithm is similar to the algorithm that is described below.

An Algorithm using Newton's Method

Here, we will describe an algorithm that was not found in the literature. First, let's define more precisely the problem of finding the intersection of two arbitrary ellipsoids. We will represent a point **p** using homogenous coordinates, which are defined as follows:

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

where x, y, and z are the customary coordinates in 3-dimesional space, and the 1 in the last row of the vector is useful in order to allow translations operations to be performed via matrix multiplication. Scaling, rotation, and translation operations can be performed by multiplying the coordinate vector by a 4x4 matrix. An arbitrary ellipsoid is formed by performing a scaling operation, followed by a rotation, followed by a translation on a unit circle. Hence, an ellipsoid is defined as:

$$E = TRS \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \forall\, x^2 + y^2 + z^2 = 1$$

where T, R, and S are the translation, rotation, and scaling matrices. Note that the matrix TRS fully defines the ellipsoid. Let us denote the ellipsoid's "local" coordinate space to mean the coordinate space in which the ellipsoid is a unit circle centered around the origin, and denote the "global" coordinate space to be the space in which all of our objects reside. Therefore, multiplying by TRS converts coordinates from the ellipsoid's local coordinate space to the global coordinate space. Further, multiplying by $S^{-1}R^{-1}T^{-1}$ converts coordinates from the global space to the ellipsoid's local space.

Now, suppose we have two ellipsoids E_1 and E_2 defined by $T_1R_1S_1$ and $T_2R_2S_2$. Define two new matrices as follows:

$$M_1 = T_1R_1S_1$$
$$M_2 = T_2R_2S_2$$

In order to convert coordinates from E_1's space to E_2's space, we first convert from E_1's space to the global coordinate space, and then from the global space to E_2's space. Therefore, multiplying by the matrix $M_2M_1^{-1}$ converts coordinates from E_1's space to E_2's space.

In order to determine if E_1 and E_2 intersect, we want to find the point on E_2 with the minimum distance to the center of E_1. If that point is within a distance of 1 from the origin in E_1's coordinate space, then the ellipsoids intersect. In order to find the point with the minimum distance to E_1, we should set up a distance function $d(u, v)$ that takes as inputs two parameters that define the location of a point on the surface of E_2. The extremes of the function happen at points for which $\nabla d(u, v) = 0$. We use Newton's method to solve for these points.

In Newton's method, given a continuous, differentiable function f(x), you solve for a root f(x)=0 by making iteratively improving guesses. Given a guess x_i, the next guess x_{i+1} is calculated as follows:

$$x_{i+1} = x_i - \frac{f(x_i)}{\frac{df}{dx}(x_i)}$$

For our problem, we take an initial guess for the point on E_2 at the minimum distance to the center of E_1, and then make subsequent guesses that we believe will bring $\nabla d(u.v)$ closer to zero. Note that it is easier to express the distance function in E_1's coordinate space, but it is easier to specify a valid point on E_2 in E_2's coordinate space. Therefore, the algorithm will convert points back and forth between E_1's and E_2's coordinate spaces. Also note that vectors can be scaled and rotated using the same scaling and rotating matrices used for points. However, translations do not affect vectors. Therefore, to convert a vector from E_2's space to E_1's space, we multiply it by $S_1^{-1}R_1^{-1}R_2S_2$. To convert a vector from E_1's space to E_2's space, we multiply by the inverse matrix.

Our iterative guesses are points on the surface of E_2. The coordinate system which we use to define these points, and with which we parameterize the distance function, is arbitrary. For example, it might make sense to use standard polar coordinates, where θ is thought of as the angle down from the north pole of the unit sphere, and ϕ is thought of as an angle of longitude. However, since the coordinate system is arbitrary, we can choose whichever one is most convenient. Further, we can change the coordinate system we use from iteration to iteration. This is particularly easy if we don't store our guess points in terms of these coordinates, but rather store them simply in Cartesian coordinates.

Recall that in E_2's local coordinate space, E_2 is a unit sphere. Our guess point lies on the surface of this sphere. Any unit vector orthogonal to the radius to our guess point is in fact the derivative of the position of that point with respect to some parameter u in some coordinate system. For example, after choosing the unit vector, we could find a polar coordinate system such that our unit vector is equal to $\frac{dp}{d\theta}$. If we find two such vectors that are themselves orthogonal, then we have $\frac{dp}{du}$ and $\frac{dp}{dv}$ for some coordinate system u, v. Further, both $\frac{d^2p}{du^2}$ and $\frac{d^2p}{dv^2}$ are equal to the unit vector from our guess point toward the center of the sphere, that is, the negative of the radius to the guess point. So, without precisely defining the coordinate system, we have found $\frac{dp}{du}$, $\frac{dp}{dv}$, $\frac{d^2p}{du^2}$, and $\frac{d^2p}{dv^2}$. We then convert these vectors to E_1's coordinate space.

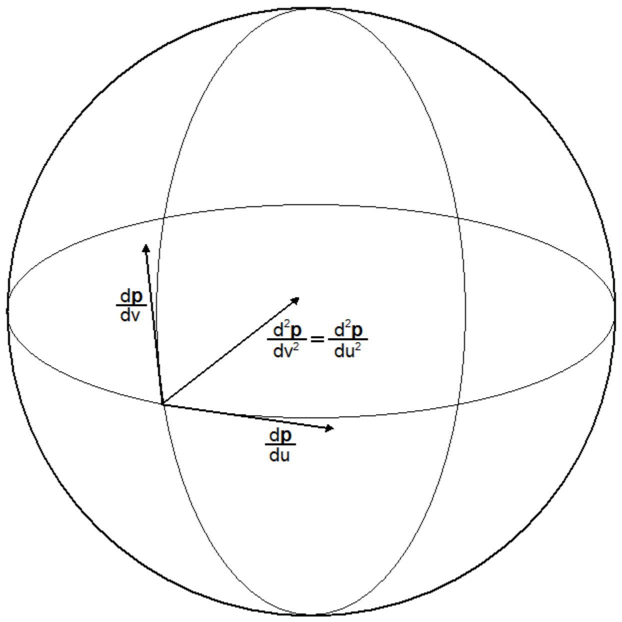

Figure 2. Any two unit vectors that are orthogonal to each other and orthogonal to the radius are equal to dp/du and dp/dv for some coordinate system (u,v).

We now have $\frac{dp}{du}$ in E_1's coordinate space, and can likewise calculate our guess point's position in E_1's coordinate space. Clearly, $\frac{dd}{du}$ is the length of the component of $\frac{dp}{du}$ that lies in the same direction as a vector from the center of E_1 to our guess point. That is, given some infinitesimal change in u, we know in what direction and at what rate our guess point will move. The component of that movement away from the center of E_1 is exactly the rate at which the distance from the center of E_1 will increase. The same argument applies to $\frac{d^2d}{du^2}$. Therefore, if N is a unit vector pointing from the center of E_1 to our guess point, then we have

$$\frac{dd}{du}=\frac{dp}{du}\cdot N$$
$$\frac{d^2d}{du^2}=\frac{d^2p}{du^2}\cdot N$$

Similarly, we calculate $\frac{dd}{dv}$ and $\frac{d^2d}{dv^2}$. Once we have calculated all four of these values, we can compute how far we should move our guess point as follows:

$$p^{'}=p-\left(\frac{\frac{dd}{du}}{\frac{d^2d}{du^2}}\right)\frac{dp}{du}-\left(\frac{\frac{dd}{dv}}{\frac{d^2d}{dv^2}}\right)\frac{dp}{dv}$$

Newton's method is known to converge quadradically. That is, if we think about the number of significant digits in the result, this number doubles with each iteration of the method. Therefore, our algorithm should run in O(log log k) time. This is significantly faster than the other methods described.

References

[CH87] Chazelle, B. and D. P. Dobkin. Intersection of convex objects in two and three dimensions. Journal of the ACM, 31:1–27, 1987.

[CO68] Comba, Paul G. A Procedure for Detecting Intersections of Three-Dimensional Objects. Journal of the ACM, Volume 15, Issue 3, July 1968. Pages 354 - 366.

[GO91] Goldman, Ronald N. and James R. Miller. Combining algebraic rigor with geometric robustness for the detection and calculation of conic sections in the intersection of two natural quadric surfaces. Proceedings of the first ACM symposium on Solid modeling foundations and CAD/CAM applications, 1991.

[GO99] Gomez, M. Simple intersection tests for games. Gamasutra, October 1999.

[HA04] Hadap, Sunil et al. Collision detection and proximity queries. ACM SIGGRAPH 2004 Course Notes.

[LE76] Levin, J. A Parametric Algorithm for Drawing Pictures of Solid Objects Composed of Quadric Surfaces. Communications of the ACM, Vol. 19, No. 10, October 1976, pp. 555-563.

[MO92] Mount, David M. Intersection detection and separators for simple polygons. Proceedings of the eighth annual symposium on Computational geometry, 1992.

[SA83] Sarraga, R. F. Algebraic Methods for Intersections of Quadric Surfaces in GMSOLID. Computer Vision, Graphics, and Image Processing, Vol. 22, No. 2, May 1983, pp. 222-238.

[YA85] Yamaguchi, Fujio. A Unified Approach to Interference Problems Using a Triangle Processor. ACM SIGGRAPH Computer Graphics, v.19 n.3, p.141-149, Jul. 1985.