Algorithm 7

Distance between Lines and Segments with their Closest Point of Approach

by Dan Sunday

We considered the distance from a point to a line in Algorithm 2 about the Distance of a Point to a Line, Ray or Segment. We now consider the minimum distance between lines. One sometimes has to compute the distance separating two geometric objects; for example, in collision avoidance algorithms. These objects could be polygons (in 2D) or polyhedra (in 3D which are composed of linear edges and faces. The Euclidean distance between any two geometric objects is defined as the minimum distance between any two of their points. That is, for two geometric sets $\mathbf{G}_1$ and $\mathbf{G}_2$ (in any n-dimensional space), the distance between them is defined as:

$$d(\mathbf{G}_1,\mathbf{G}_2) = \min_{P \in \mathbf{G}_1,\, Q \in \mathbf{G}_2} d(P,Q)$$

In this month's algorithm we show how to efficiently compute this distance between lines, rays and segments (in any dimension). We also show how to find the two points, P and Q, where this minimum occurs; that is, where $d(\mathbf{G}_1,\mathbf{G}_2) = d(P,Q)$. It is not true that these points always exist for arbitrary geometric sets. But, they exist for many well-behaved geometric objects such as lines, planes, polygons, polyhedra, and closed & bounded (i.e.: "compact") objects.

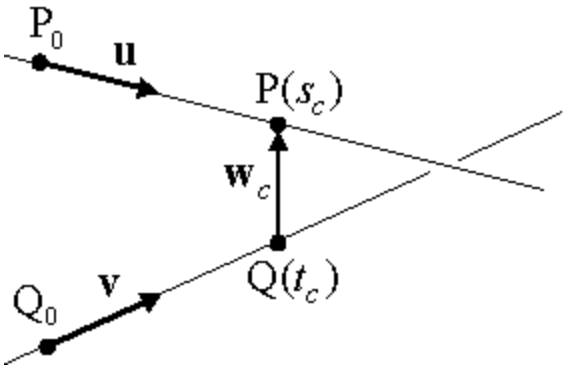
Additionally, we show how to compute the closest point of approach (CPA) between two points that are dynamically moving in a straight line. This is also important an important computation in collision detection, for example to determine how close two planes or two ships, represented as point "tracks", will get as they pass each other.

We use the parametric equation to represent lines, rays and segments, as described in the Algorithm 2 section on Lines, which also shows how to convert between this and other representations.

Distance between Lines

We first consider two infinite lines $\mathbf{L}_1$: $P(s) = P_0 + s\,(P_1-P_0) = P_0 + s\mathbf{u}$ and $\mathbf{L}_2$: $Q(t) = Q_0 + t\,(Q_1-Q_0) = Q_0 + t\mathbf{v}$. Let $\mathbf{w}(s,t) = P(s)-Q(t)$ be a vector between points on the two lines. We want to find the $\mathbf{w}(s,t)$ that has a minimum length for all s and t. This can be computed using calculus [Eberly, 2001]. Here, we use a more geometric approach, and end up with the same result as he does. A similar geometric approach was used by [Teller, 2000], but he uses a cross product which restricts his method to 3D space whereas the method we use works in any dimension. Also, the solution given here and by Eberly are faster than Teller's which computes intermediate planes and gets their intersections with the two lines.

In any n-dimensional space, the two lines $\mathbf{L}_1$ and $\mathbf{L}_2$ are closest at unique points $P(s_c)$ and $Q(t_c)$ for which $\mathbf{w}(s_c,t_c)$ attains its minimum length. Also, if $\mathbf{L}_1$ and $\mathbf{L}_2$ are not parallel, then the line segment $P(s_c)Q(t_c)$ joining the closest points is uniquely perpendicular to both lines at the same time. No other segment between $\mathbf{L}_1$ and $\mathbf{L}_2$ has this property. That is, the vector $\mathbf{w}_c = \mathbf{w}(s_c,t_c)$ is uniquely perpendicular to the line direction vectors $\mathbf{u}$ and $\mathbf{v}$, and this is equivalent to it satisfying the two equations: $\mathbf{u} \cdot \mathbf{w}_c = 0$ and $\mathbf{v} \cdot \mathbf{w}_c = 0$.



We can solve these two equations by substituting $\mathbf{w}_c = P(s_c)-Q(t_c) = \mathbf{w}_0 + s_c\mathbf{u} - t_c\mathbf{v}$, where $\mathbf{w}_0 = P_0-Q_0$, into each one to get two simultaneous linear equations:

$$(\mathbf{u}\cdot\mathbf{u})s_c - (\mathbf{u}\cdot\mathbf{v})t_c = -\mathbf{u}\cdot\mathbf{w}_0$$
$$(\mathbf{v}\cdot\mathbf{u})s_c - (\mathbf{v}\cdot\mathbf{v})t_c = -\mathbf{v}\cdot\mathbf{w}_0$$

Then, letting $a = \mathbf{u}\cdot\mathbf{u}$, $b = \mathbf{u}\cdot\mathbf{v}$, $c = \mathbf{v}\cdot\mathbf{v}$, $d = \mathbf{u}\cdot\mathbf{w}_0$, and $e = \mathbf{v}\cdot\mathbf{w}_0$, we solve for s_c and t_c as:

$$s_c = \frac{be-cd}{ac-b^2} \quad \text{and} \quad t_c = \frac{ae-bd}{ac-b^2}$$

whenever the denominator $ac-b^2$ is nonzero. Note that $ac-b^2 = |\mathbf{u}|^2|\mathbf{v}|^2-(|\mathbf{u}||\mathbf{v}|\cos q)^2 = (|\mathbf{u}||\mathbf{v}|\sin q)^2 >=0$ is always nonnegative. When $ac-b^2 = 0$, the two equations are dependant, the two lines are parallel, and the distance between the lines is constant. We can solve for this parallel distance of separation by fixing the value of one parameter and using either equation to solve for the other. Selecting $s_c = 0$, we get $t_c = d / b = e / c$.

Having solved for s_c and t_c, we have the points $P(s_c)$ and $Q(t_c)$ where the two lines $\mathbf{L}_1$ and $\mathbf{L}_2$ are closest. Then the distance between them is given by:

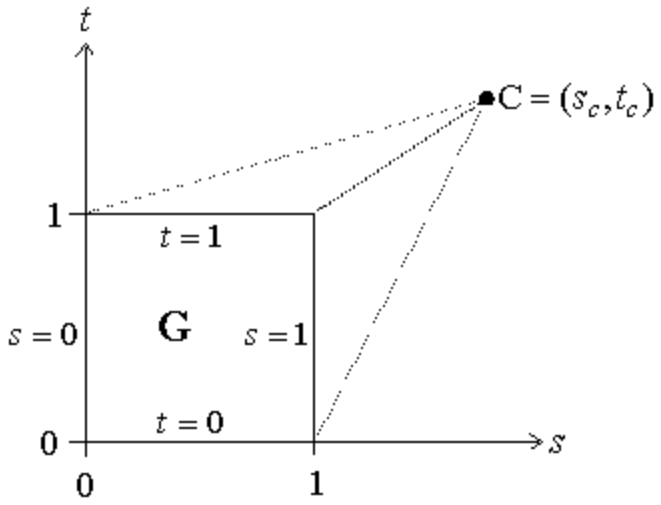
$$d(\mathbf{L}_1,\mathbf{L}_2) = \left|P(s_c) - Q(t_c)\right| = \left|(P_0 - Q_0) + \frac{(be-cd)\mathbf{u} - (ae-bd)\mathbf{v}}{ac-b^2}\right|$$

Distance between Segments and Rays

The distance between segments and rays may not be the same as the distance between their extended lines. The closest points on the extended infinite line may be outside the range of the segment or ray which is a restricted subset of the line. We represent a segment $\mathbf{S}_1$ (between endpoints P_0 and P_1) as the points on $\mathbf{L}_1$: $P(s) = P_0 + s\,(P_1-P_0) = P_0 + s\mathbf{u}$ with $0<=s<=1$. Also, a positive ray $\mathbf{R}_1$ (starting from P_0) is given by the points $P(s)$ with $s>=0$. Similarly, the segment $\mathbf{S}_2$ on $\mathbf{L}_2$ from Q_0 to Q_1 is given by the points $Q(t)$ with $0<=t<=1$.

The first step in computing a distance involving segments and/or rays is to get the closest points for the lines they lie on. So, we first compute s_c and t_c for $\mathbf{L}_1$ and $\mathbf{L}_2$, and if these are in the range of the involved segment or ray, then they are also the closest points for them. But if they lie outside the range, then they are not and we have to determine new points that minimize $\mathbf{w}(s,t)=P(s)-Q(t)$ over the ranges of interest. To do this, we first note that minimizing the length of $\mathbf{w}$ is the same as minimizing $|\mathbf{w}|^2 = \mathbf{w} \cdot \mathbf{w} = (\mathbf{w}_0 + s\mathbf{u} - t\mathbf{v}) \cdot (\mathbf{w}_0 + s\mathbf{u} - t\mathbf{v})$ which is a quadratic function of s and t. In fact, it defines a parabaloid over the (s,t)-plane with a minimum at $C=(s_c,t_c)$, and which is strictly increasing along rays in the (s,t)-plane that start from C and go in any direction. However, when segments and/or rays are involved, we need the minimum over a subregion $\mathbf{G}$ of the (s,t)-plane, and the global minimum at C may lie outside of $\mathbf{G}$. In these cases, the minimum always occurs on the boundary of $\mathbf{G}$, and in particular, on the part of $\mathbf{G}$'s boundary that is visible to C (that is, a line from C to the boundary point is exterior to $\mathbf{G}$). We say that C can "see" points on this visible boundary of $\mathbf{G}$.

To be specific, suppose that we want the minimum distance between two finite segments $\mathbf{S}_1$ and $\mathbf{S}_2$. Then, we have that $\mathbf{G} = \{(s,t) \mid 0<=s<=1 \text{ and } 0<=t<=1\} = [0,1]\times[0,1]$ is the unit square as shown in the diagram. The four edges of the square are given by $s=0$, $s=1$, $t=0$, and $t=1$. If $C=(s_c,t_c)$ is outside $\mathbf{G}$, then it can see at most two edges of $\mathbf{G}$. If $s_c<0$, C can see the s=0 edge; if $s_c>1$, C can see the s=1 edge; and similarly for t_c. Clearly, if C is not in $\mathbf{G}$, then at least 1 and at most 2 of these inequalities are true, and they determine which edges of $\mathbf{G}$ are candidates for a minimum of $|\mathbf{w}|^2$.



For each candidate edge, we use basic calculus to compute where the minimum occurs on that edge, either in its interior or at an endpoint. Consider the edge s=0, along which $|\mathbf{w}|^2 = (\mathbf{w}_0 - t\mathbf{v}) \cdot (\mathbf{w}_0 - t\mathbf{v})$. Taking the derivative with t we get a minimum when:

$$0 = \frac{d}{dt}|\mathbf{w}|^2 = -2\mathbf{v}\cdot(\mathbf{w}_0 - t\,\mathbf{v})$$

which gives a minimum on the edge at $(0,t_0)$ where:

$$t_0 = \frac{\mathbf{v}\cdot\mathbf{w}_0}{\mathbf{v}\cdot\mathbf{v}}$$

If $0<=t_0<=1$, then this will be the minimum of $|\mathbf{w}|^2$ on **G**, and P(0) and Q(t_0) are the two closest points of the two segments. However, if t_0 is outside the edge, then an endpoint of the edge, (0,0) or (0,1), is the minimum; plus, we will have to check a second visible edge in case the true minimum is on it. The other edges are treated in a similar manner.

Putting it all together by testing all candidate edges, we end up with a relatively simple algorithm that only has a few cases to check. An analogous approach is given by [Eberly, 2001], but it has more cases which makes it more complicated to implement.

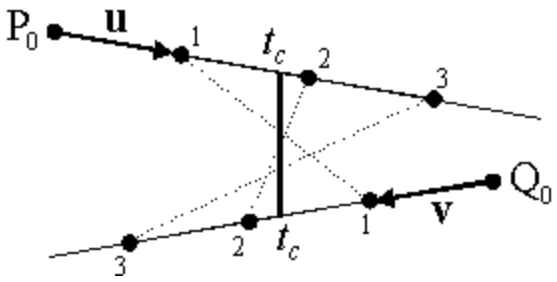
Other distance algorithms, such as line-to-ray or ray-to-segment, are similar in spirit, but have fewer boundary tests to make than the segment-to-segment distance algorithm.

Closest Point of Approach (CPA)

The "Closest Point of Approach" refers to the positions at which two dynamically moving objects reach their closest possible distance. This is an important calculation for collision avoidance. In many cases

of interest, the objects, referred to as "tracks", are points moving in two fixed directions at fixed speeds. That means that the two points are moving along two lines in space. However, their closest distance is not the same as the closest distance between the lines since the distance between the points must be computed at the same moment in time. So, even in 2D with two lines that intersect, points moving along these lines may remain far apart. But if one of the tracks is stationary, then the CPA of another moving track is at the base of the perpendicular from the first track to the second's line of motion.

Consider two dynamically changing points whose positions at time t are P(t) and Q(t). Let their positions at time t=0 be P_0 and Q_0; and let their velocity vectors per unit of time be **u** and **v**. Then, the equations of motion for these two points are $P(t) = P_0 + t\mathbf{u}$ and $Q(t) = Q_0 + t\mathbf{v}$ which are the familiar parametric equations for the lines. However, the two equations are coupled by having a common parameter t. So, at time t, the distance between them is $d(t) = |P(t)-Q(t)| = |\mathbf{w}(t)|$ where $\mathbf{w}(t) = \mathbf{w}_0 + t(\mathbf{u}-\mathbf{v})$ with $\mathbf{w}_0 = P_0-Q_0$.



Now, d(t) is a minimum when $D(t) = d(t)^2$ is a minimum, and we can compute:

$$D(t) = \mathbf{w}(t)\cdot\mathbf{w}(t) = (\mathbf{u}-\mathbf{v})\cdot(\mathbf{u}-\mathbf{v})t^2 + 2\,\mathbf{w}_0\cdot(\mathbf{u}-\mathbf{v})t + \mathbf{w}_0\cdot\mathbf{w}_0$$

which has a minimum when

$$0 = \frac{d}{dt}D(t) = 2t[(\mathbf{u}-\mathbf{v})\cdot(\mathbf{u}-\mathbf{v})] + 2\,\mathbf{w}_0\cdot(\mathbf{u}-\mathbf{v})$$

which can be solved to get the time of CPA to be:

$$t_c = \frac{-\mathbf{w}_0\cdot(\mathbf{u}-\mathbf{v})}{|\mathbf{u}-\mathbf{v}|^2}$$

whenever |**u**-**v**| is nonzero. If |**u**-**v**|=0, then the two points are traveling in the same direction at the same speed, and will always remain the same distance apart, so one can use $t_c = 0$. In both cases we have that:

$$d_{CPA}\left(P(t), Q(t)\right) = \left|P(t_c) - Q(t_c)\right|$$

Note that when $t_c < 0$, then the CPA has already occurred in the past, and the two points are getting

further apart as they move on in time.

Implementations

Here are some sample "C++" implementations of these algorithms.

```
// Copyright 2001, softSurfer (www.softsurfer.com)
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// SoftSurfer makes no warranty for this code, and cannot be held
// liable for any real or imagined damage resulting from its use.
// Users of this code must verify correctness for their application.

// Assume that classes are already given for the objects:
//    Point and Vector with
//        coordinates {float x, y, z;}
//        operators for:
//            Point  = Point ± Vector
//            Vector = Point - Point
//            Vector = Vector ± Vector
//            Vector = Scalar * Vector
//    Line and Segment with defining points {Point P0, P1;}
//    Track with initial position and velocity vector
//            {Point P0; Vector v;}
//===================================================================

#define SMALL_NUM  0.00000001 // anything that avoids division
overflow
// dot product (3D) which allows vector operations in arguments
#define dot(u,v)   ((u).x * (v).x + (u).y * (v).y + (u).z * (v).z)
#define norm(v)    sqrt(dot(v,v))  // norm = length of vector
#define d(u,v)     norm(u-v)       // distance = norm of difference
#define abs(x)     ((x) >= 0 ? (x) : -(x))   // absolute value

// dist3D_Line_to_Line():
//    Input:  two 3D lines L1 and L2
//    Return: the shortest distance between L1 and L2
float
dist3D_Line_to_Line( Line L1, Line L2)
{
    Vector   u = L1.P1 - L1.P0;
    Vector   v = L2.P1 - L2.P0;
    Vector   w = L1.P0 - L2.P0;
    float    a = dot(u,u);         // always >= 0
    float    b = dot(u,v);
    float    c = dot(v,v);         // always >= 0
    float    d = dot(u,w);
    float    e = dot(v,w);
```

```
    float    D = a*c - b*b;        // always >= 0
    float    sc, tc;

    // compute the line parameters of the two closest points
    if (D < SMALL_NUM) {          // the lines are almost parallel
        sc = 0.0;
        tc = (b>c ? d/b : e/c);   // use the largest denominator
    }
    else {
        sc = (b*e - c*d) / D;
        tc = (a*e - b*d) / D;
    }

    // get the difference of the two closest points
    Vector   dP = w + (sc * u) - (tc * v);  // = L1(sc) - L2(tc)

    return norm(dP);   // return the closest distance
}
//===================================================================

// dist3D_Segment_to_Segment():
//    Input:  two 3D line segments S1 and S2
//    Return: the shortest distance between S1 and S2
float
dist3D_Segment_to_Segment( Segment S1, Segment S2)
{
    Vector   u = S1.P1 - S1.P0;
    Vector   v = S2.P1 - S2.P0;
    Vector   w = S1.P0 - S2.P0;
    float    a = dot(u,u);         // always >= 0
    float    b = dot(u,v);
    float    c = dot(v,v);         // always >= 0
    float    d = dot(u,w);
    float    e = dot(v,w);
    float    D = a*c - b*b;        // always >= 0
    float    sc, sN, sD = D;       // sc = sN / sD, default sD = D >= 0
    float    tc, tN, tD = D;       // tc = tN / tD, default tD = D >= 0

    // compute the line parameters of the two closest points
    if (D < SMALL_NUM) { // the lines are almost parallel
        sN = 0.0;        // force using point P0 on segment S1
        sD = 1.0;        // to prevent possible division by 0.0 later
        tN = e;
        tD = c;
    }
    else {                 // get the closest points on the infinite
lines
        sN = (b*e - c*d);
        tN = (a*e - b*d);
        if (sN < 0.0) {       // sc < 0 => the s=0 edge is visible
```

```
            sN = 0.0;
            tN = e;
            tD = c;
        }
        else if (sN > sD) {  // sc > 1 => the s=1 edge is visible
            sN = sD;
            tN = e + b;
            tD = c;
        }
    }

    if (tN < 0.0) {            // tc < 0 => the t=0 edge is visible
        tN = 0.0;
        // recompute sc for this edge
        if (-d < 0.0)
            sN = 0.0;
        else if (-d > a)
            sN = sD;
        else {
            sN = -d;
            sD = a;
        }
    }
    else if (tN > tD) {      // tc > 1 => the t=1 edge is visible
        tN = tD;
        // recompute sc for this edge
        if ((-d + b) < 0.0)
            sN = 0;
        else if ((-d + b) > a)
            sN = sD;
        else {
            sN = (-d + b);
            sD = a;
        }
    }
    // finally do the division to get sc and tc
    sc = (abs(sN) < SMALL_NUM ? 0.0 : sN / sD);
    tc = (abs(tN) < SMALL_NUM ? 0.0 : tN / tD);

    // get the difference of the two closest points
    Vector   dP = w + (sc * u) - (tc * v);  // = S1(sc) - S2(tc)

    return norm(dP);   // return the closest distance
}
//===================================================================

// cpa_time(): compute the time of CPA for two tracks
//    Input:  two tracks Tr1 and Tr2
//    Return: the time at which the two tracks are closest
float
```

```
cpa_time( Track Tr1, Track Tr2 )
{
    Vector    dv = Tr1.v - Tr2.v;

    float     dv2 = dot(dv,dv);
    if (dv2 < SMALL_NUM)       // the tracks are almost parallel
        return 0.0;             // any time is ok.  Use time 0.

    Vector    w0 = Tr1.P0 - Tr2.P0;
    float     cpatime = -dot(w0,dv) / dv2;

    return cpatime;             // time of CPA
}
//===================================================================

// cpa_distance(): compute the distance at CPA for two tracks
//    Input:  two tracks Tr1 and Tr2
//    Return: the distance for which the two tracks are closest
float
cpa_distance( Track Tr1, Track Tr2 )
{
    float     ctime = cpa_time( Tr1, Tr2);
    Point     P1 = Tr1.P0 + (ctime * Tr1.v);
    Point     P2 = Tr2.P0 + (ctime * Tr2.v);

    return d(P1,P2);            // distance at CPA
}
//===================================================================
```

References

David Eberly, 3D Game Engine Design, Section 2.6 "Distance Methods" (2001)

Seth Teller, line_line_closest_points3d() (2000) cited in the Graphics Algorithms FAQ Subject 5.18 (2001)