2.3

Fast, Robust Intersection of 3D Line Segments

Graham Rhodes, Applied Research Associates
grhodes@sed.ara.com

The problem of determining the intersection of two line segments comes up from time to time in game development. For example, the line/line intersection problem can be beneficial in simple collision detection. Consider two objects in three-dimensional space that are moving in time. During a time step or animation frame, each object will move from one point to another along a linear path. The simplest check to see if the objects collide during the time step would be to see how close the two linear paths come to crossing, and if they are within a certain distance of each other (in other words, less than the sum of the radii of bounding spheres of the objects), then process a collision. Other common applications for line segment intersections include navigation and motion planning (for example, when combined with an AI system), map overlay creation, and terrain/visibility estimation.

This gem describes a robust, closed form solution for computing the intersection between two infinite lines or finite-length line segments in three-dimensional space, if an intersection exists. When no intersection exists, the algorithm produces the point along each line segment that is closest to the other line, and a vector between the two nearest points.

What Makes This Algorithm Robust?

The algorithm presented here is robust for a couple of reasons. First, it does not carry any special requirements (for example, the line segments must be coplanar). Second, it has relatively few instances of tolerance checks. The basic algorithm has only two tolerance checks, and these are required mathematically rather than by heuristics.

The Problem Statement

Given two line segments in three-dimensional space, one that spans between the points $\vec{j4i} = [A_{lx} \quad Ai_y \quad A_{lz}]^r$ and $\vec{A_2} = [A^\wedge \quad A_{2y} \quad A_{2z}\backslash^T$ and one that spans between the points $B_l = [5_{lx} \quad Bi_y \quad B_{lz}\backslash^r$ and $B_2 = [B^\wedge \quad B_{2y} \quad B_{2z}]^T$, we would like to find the true point of intersection, $P=[P_X \quad P_y \quad P_Z]^T$, between the two segments, if it exists. "When

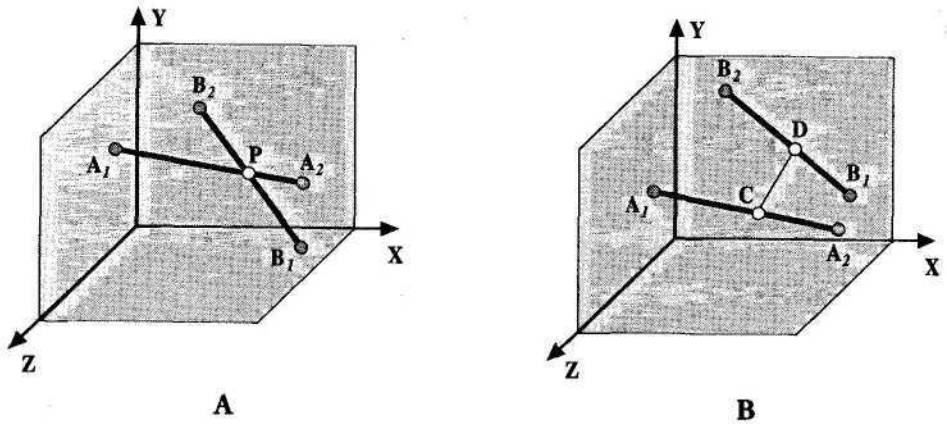


FIGURE 2.3.1 *Two line segments in three-dimensional space. A) An intersection exists. B) No intersecton.*

no intersection exists, we would like to compromise and find the point on each segment that is nearest to the other segment. Figure 2.3.1 illustrates the geometry of this situation.

The nearest points, labeled *C* and *D* respectively, can be used to find the shortest distance between the two segments. This gem focuses on finding the nearest points, which are identical to the true intersection point when an intersection exists.

Observations

Before delving into how to solve the line intersection problem, it can be useful to make a few observations. What are the challenges to solving the problem correctly?

Consider an arbitrary, infinite line in space. It is likely that the line will intersect an arbitrary plane (if the line is not parallel to the plane, then it intersects the plane); however, it is unlikely that the line will truly intersect another line (even if two three-dimensional lines are not parallel, they do not necessarily intersect). From this observation, we can see that no algorithm designed to find only true intersections will be robust, capable of finding a result for an arbitrary pair of lines or line segments, since such an algorithm will fail most of the time. The need for a robust algorithm justifies the use of an algorithm that finds the nearest points between two lines, within a real-time 3D application such as a game.

Since every student who has taken a basic planar geometry class has solved for the intersection of lines in a two-dimensional space, it is useful to consider the relationship between the three-dimensional line intersection problem and the two-dimensional intersection problem. In two-dimensional space, any two nonparallel lines truly intersect at one point. To visualize what happens in three-dimensional space, consider a plane that contains both defining points of line A, and die first defining point of line

B. Line A lies within the plane, as does the first defining point of line B. Note that die point of intersection of die two lines lies on die plane, since diat point is contained on line A. The point of intersection also lies on line B, and so two points of line B lie widiin die plane. Since two points of line B lie in the plane, the entire line lies in the plane.

The important conclusion here is diat whenever there is a true intersection of two lines, those two lines do lie widiin a common plane. Thus, any time two three-dimensional lines have a true intersection, the problem is equivalent to a two-dimensional intersection problem in die plane diat contains all four of the defining points.

NaTve Solutions

A naive, and problematic, solution to die intersection problem is to project the two segments into one of the standard coordinates planes (XY, YZ, or XZ), and then solve the problem in the plane. In terms of implementation, the primary difficulty widi diis approach is selecting an appropriate plane to project into. If neither of the line segments is parallel to any of the coordinate planes, dien the problem can be solved in any coordinate plane. However, an unacceptable amount of logic can be required when one or both segments are parallel to coordinate planes. A variation on diis approach, less naive but still problematic, is to form a plane equation from three of the four points, $\vec{A}_{ly}$ $\vec{A}_2$, $\vec{B}_{l3}$ and $\vec{B}_2$, project all four points into the plane, and solve the problem in the plane. In the rare case that there is a true intersection, this latter approach produces the correct result.

One key feature that is completely lacking from the basic two-dimensional projected intersection problem is the ability to give a direct indication as to whether a three-dimensional intersection exists. It also doesn't provide the three-dimensional nearest points. It is necessary to work backwards to produce diis vital information.

The biggest problem with either variation on the projected solution arises when the two lines pass close to one anodier, but do not actually intersect. In this case, the solution obtained in any arbitrary projection plane will not necessarily be the correct pair of nearest points. The projection will often yield completely wrong results! To visualize this situation (which is difficult to illustrate on a printed page), consider the following mind experiment. There are two line segments floating in space. Segment A is defined by die points (0, 0, 0) and (1, 0, 0), and segment B is defined by (1, 0, 1) and (1, 1, 1). When the lines are viewed from above, equivalent to projecting the lines into the XY plane, the two-dimensional intersection point is (1, 0, 0), and the three-dimensional nearest points are (1, 0, 0) and (1, 0, 1). These are the correct nearest points for the problem. However, if those two lines are viewed from different arbitrary angles, the two-dimensional intersection point will move to appear anywhere on the two line segments. Projecting the two-dimensional solution back onto the diree-dimensional lines yields an infinite number of "nearest" point pairs, which is clearly incorrect. The test code provided on the companion CD-ROM is a useful tool to see

this problem, as it allows you to rotate the view to see two line segments from different viewing angles, and displays the three-dimensional nearest points that you can compare to the intersection point seen in the viewport.

In the next section, I derive a closed-form solution to the calculation of points *C* and *D* that does not make any assumptions about where the two line segments lie in space. The solution does handle two special cases, but these cases are unavoidable even in the alternative approaches.

Derivation of Closed-Form Solution Equations

Calculating the Nearest Points on Two Infinite Lines

The equation of a line in three-dimensional space can be considered a vector function of a single scalar value, a parameter. To derive a closed-form solution to the nearest-point between two 3D lines, we first write the equation for an arbitrary point, $\vec{C} = [C_x \quad C_y \quad CJ^r$, located on the first line segment, as Equation 2.3.1.

$$C = A_1 + sL_A, \text{ where } L_A = (A_2 - 4) \tag{2.3.1}$$

Notice that Equation 2.3.1 basically says that the coordinates of any point on the first segment are equal to the coordinates of the first defining point plus an arbitrary scalar parameter *s* times a vector pointing along the line from the first defining point to the second defining point. If *s* is equal to zero, the coordinate is coincident with the first defining point, and if *s* is equal to 1, the coordinate is coincident with the second defining point. We can write a similar equation for an arbitrary point, $\vec{D} = [D_x \quad D_y \quad DJ^T$, located on the second line segment, as Equation 2.3.2:

$$\vec{D} = \vec{5}; + t\vec{L}_B, \text{ where } \vec{L}_B = (\vec{\#}, - \vec{B}) \tag{2.3.2}$$

Here, *t* is a second arbitrary scalar parameter, with the same physical meaning as *s* with respect to the second line segment. If the parameters *s* and *t* are allowed to be arbitrary, then we will be able to calculate points $\vec{C}$ and $\vec{D}$ as they apply to infinite lines rather than finite segments. For any point on *a. finite* line segment, the parameters *s* and *t* will satisfy $0 < s,t < 1$. We'll allow *s* and *t* to float arbitrarily for now, and treat the finite length segments later.

The two 3D line segments intersect if we can find values of *s* and *t* such that points $\vec{C}$ and $\vec{D}$ are coincident. For a general problem, there will rarely be an intersection, however, and we require a method for determining *s* and *t* that corresponds to the nearest points $\vec{C}$ and $\vec{D}$. The remainder of the derivation shows how to solve for these values of; and *t.*

First, subtract Equation 2.3.2 from Equation 2.3.1 to obtain the following equation for the vector between points $\vec{C}$ and $\vec{D}$:

$$C\text{-}D = -AB + sL_A - tL_B = [o \quad 0 \quad OJ^r,$$

where $\overrightarrow{AB} = \vec{B}_l - \vec{A}_l$ (2.3.3)

Here, since we would like for points $\vec{C}$ and $\vec{D}$ to be coincident, we set the vector between the points to be the zero vector. The right side of Equation 2.3.3 can then be represented by the following matrk equation:

$$\begin{bmatrix} L_{Ax} & -L_{S^*} \\ L_{Ay} & -L_{By} \\ L_{A\cdot} & -L_{\cdot z} \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} AB_x \\ AB_y \\ AB_z \end{bmatrix} \quad (2.3.4)$$

There are three rows in Equation 2.3.4, one for each coordinate direction, but only two unknowns, the scalar values *s* and *t.* This is a classic over-determined or under-constrained system. The only way there can be an exact solution is if the coefficient matrix on the left side turns out to have rank 2, in which case the three equations are equivalent to just two independent equations, leading to an exact solution for *s* and *t.* Geometrically, when there is an exact solution, the two lines have a true intersection and are coplanar. Thus, two arbitrary lines in three-dimensional space can only have a true intersection when the lines are coplanar.

The difference between the left side and right side of Equation 2.3.4 is equal to the vector representing the distance between $\vec{C}$ and $\vec{D}$. It is also the error vector of Equation 2.3.4 for any arbitrary values of *s* and *t.* We determine the nearest points by minimizing the length of this vector over all possible values of *s* and *t.*

The values of *s* and *t* that minimize the distance between $\vec{C}$ and $\vec{D}$ correspond to a linear least-squares solution to Equation 2.3.4. Geometrically, the least-squares solution produces the points $\vec{C}$ and $\vec{D}$. When we have the case of the segments being coplanar but not parallel, then the algorithm will naturally produce the true intersection point. Equation 2.3.4 can be written in the form:

$$M? = b, \text{ where } 7 = \begin{bmatrix} s & t \end{bmatrix}^T \quad (2.3.5)$$

One method for finding the least-squares solution to an over-determined system is to solve the *normal equations* instead of the original system [Golub96]. The normal equations approach is suitable for this problem, but can be problematic for general problems involving systems of linear equations. We generate the normal equations by premultiplying the left side and right side by the transpose of the coefficient matrk M. The normal equations for our problem are shown as Equation 2.3.6.

$$M^TM? = M^T\pounds, \text{ where } M^r \text{is the transpose of } M. \quad (2.3.6)$$

Equation 2.3.6 has the desired property of reducing the system to the solution of a system of two equations, exactly the number needed to solve algebraically for values of / and *t*. Let's carry through the development of the normal equations for Equation 2.3.4. Expanding according to Equation 2.3.6 , the normal equations are:

$$\begin{bmatrix} \bar{L}AX & \bar{L}_{Ay} & \bar{L}\wedge \\ r & r & r \end{bmatrix} \begin{bmatrix} L_{Ax} & -L_{Bx} \\ L_{Ay} & \sim^{L}{}_{By} \\ L_{Az} & -L_{Bz} \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} I_{Ax} & I_{Ay} & I_{Az} \\ -L_{Bx} & -L_{By} & -L_{Bz} \end{bmatrix} \begin{bmatrix} AB_X \\ AB_y \\ AB_z \end{bmatrix} \quad (2.3.7)$$

Carrying through the matrix algebra:

$$\begin{bmatrix} \vec{L}_A \cdot \vec{L}_A & -\vec{L}_A \cdot \vec{L}_B \\ \backslash -\vec{L}_A - \vec{L}_B & \vec{L}_B \cdot \vec{L}_B \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} \vec{L}_A \cdot \overrightarrow{AB} \\ -\vec{L}_S - \overrightarrow{AB}\_ \end{bmatrix} \quad (2.3.8)$$

Or, simplifying by defining a series of new scalar variables:

$$\begin{bmatrix} L_{11} & L_{12} \\ L_{12} & L_{22} \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} r_A \\ r_B \end{bmatrix} \quad (2.3.9)$$

This is a simple 2x2 system, and to complete this section we will solve it algebraically to form a closed-form solution for *s* and *t*. There are a number of ways to solve Equation 2.3.9, including Cramer's rule [O'Neil87] and Gaussian elimination [Golub96]. Cramer's rule is theoretically interesting, but expensive, requiring approximately («+!)! multiply and divide operations for a general-sized problem. Gaussian elimination is less expensive, requiring «^{3}/3 multiply and divide operations. There are other approaches to solving systems of linear equations that are significantly more reliable and often faster for much larger systems, including advanced direct solution methods such as QR factorizations for moderate-sized systems, and iterative methods for very large and sparse systems. I will derive the solution using Gaussian elimination, which is slightly less expensive than Cramer's rule for the 2x2 system. Here, we perform one row elimination step to yield an upper triangular system. The row elimination step is as follows. Modify row 2 of Equation 2.3.9 by taking the original row 2 and subtracting row 1 times L_{l2}/L_n. to yield Equation 2.3.10.

$$\begin{bmatrix} A, & L_{12} \\ L_{12} - \text{Ai } \frac{\text{Aa}}{\text{MI}} \text{ Aa} & \text{Aa } \frac{I_{12}}{\text{Ai}} \end{bmatrix} = \begin{bmatrix} r_A \\ r_n - r_A \frac{\text{Aa}}{\text{Ai}} \end{bmatrix} \quad (2.3.10)$$

Simplify Equation 2.3.10 and multiply the new row 2 by L_u to yield the upper triangular system shown in Equation 2.3.11.

$$\begin{bmatrix} A1 & \wedge .1*1 = 1 \\ o & AiA_2\text{-}A^2{}_2 \end{bmatrix} \quad (2\text{-}3.11)$$

Equation (2.3.11) immediately yields a solution for *t,*

$$t = \frac{AI\wedge B \sim LUTA}{*i1\text{-}\wedge 22 \sim\sim A\pounds} \quad (2.3.12)$$

and then, for 5,

$$s = \frac{TA \sim Lnt}{Ai} \quad (2.3.13)$$

It is important to note that Equations 2.3.12 and 2.3.13 fail in certain degenerate cases, and it is these degenerate cases that require that we use tolerances in a limited way. Equation 2.3.13 will fail if line segment A has zero length, and Equation 2.3.12 will fail if either line segment has zero length or if the line segments are parallel. These situations lead to a divide-by-zero exception. I provide more discussion later in the section titled *Special Cases.*

In terms of computational expense for the 2x2 problem, the only difference between solving for *s* and *t* using Gaussian elimination and Cramer's rule, for this case, is that the computation of *s* requires one multiply, one divide, and one subtraction for Gaussian elimination, but four multiplies, one divide, and two subtractions for Cramer's rule.

To summarize from the derivation, given line segment A from point $\vec{A}_1$ to $\vec{A}_2$ and line segment B from point $\vec{B}_1$ to $\vec{B}_2$, define the following intermediate variables:

$$L_A = (A, -4); \; l_s = (B_2 - B,) - \sim A\vec{B} = BI\text{-}AI \quad (2.3.14)$$

and

$$\bullet HI = \wedge\vec{A} \bullet \vec{L}_A; \qquad L_{22} = \vec{L}_B \bullet \vec{L}_B; \qquad Z\wedge = -\vec{L}_A \bullet \vec{L}_B$$
$$r_A = \vec{L}_A \cdot \overrightarrow{AB}; \qquad r_B = -\vec{L}_B \cdot \overrightarrow{AB} \quad (2.3.15)$$

Compute the parameters ^ and *t* that define the nearest points as,

$$t = \frac{LU\}_{,,B} \sim \wedge^2 [^A}{AiAi \sim Ai} \quad (2.3.16)$$

and

$$s = \frac{r_A - L_{12}t}{L_{11}} \quad (2.3.17)$$

The point where the first segment comes closest to the second segment is then given by:

$$\vec{C} = \vec{A}L + s\vec{L}_A \tag{2.3.18}$$

and the point where the second segment comes closest to the first segment is given by:

$$\vec{\pounds}> = \vec{\#}! + t\vec{L}_B \tag{2.3.19}$$

We can consider a point located halfway between the two nearest points to be the single point in space that is "nearest" to both lines/segments as:

$$P = (C + D)/2 \tag{2.3.20}$$

Of course, when the lines do intersect, point $\vec{P}$ will be the intersection point.

Special Cases

When we talk about the nearest points of two infinite lines in space, there are only two possible special cases. The first case occurs when one or both lines are degenerate, defined by two points that are coincident in space. This occurs when point $\vec{A}$^ is coincident with $\vec{A}_2$, or when $\vec{B}$\ is coincident with $\vec{B}_2$. We'll call this the *degenerate line special case.* The second case occurs when the two lines are parallel, called *$\<z parallel tine special case.*

It is easy to relate the degenerate line special case to the equations developed previously. Note that variable L_n, defined in Equation 2.3.15, is equal to the square of the length of line segment *A,* and L_{22} is equal to the square of the length of segment *B.* If either of these terms is zero, indicating that a line is degenerate, then the determinant of the matrix in Equation 2.3.9 is zero, and we cannot find a solution for *s* and *t.* Note that when either L_n or L_{22} is zero, then L_{12} is also zero.

One standard test to check and decide if line *A* is degenerate is the following,

```
bool line_is_degenerate = L_n < e^2 ? true : false;
```

Here, e is a small number such as perhaps 10^{-6}. It is wiser to choose a value for £ such as 10^{-6} rather than a much smaller number such as machine epsilon.

When segments *A* and *B* are both degenerate, then point $\vec{C}$ can be selected to be equal to point $\vec{A}$, and point $\vec{D}$ can be selected to be equal to point $\vec{B}$^. When segment *A* alone is degenerate, then point $\vec{C}$is equal to $\vec{A}$\> and point $\vec{D}$ is found by computing the point on segment *B* that is nearest to point $\vec{C}$. This involves computing a value for parameter *tonly,* from Equation 2.3.21.

$$-\vec{L}_B t = \overrightarrow{AB} \tag{2.3.21}$$

Equation 2.3.21 is a simplification of Equation 2.3.4 for the case where segment *A is* degenerate, and again it requires that we find a least-squares solution. The least-squares solution, shown here using normal equations without derivation, is:

$$t = \frac{-\vec{L}_B \cdot \overrightarrow{AB}}{\vec{L}_B \cdot \vec{L}_B} = \frac{r_B}{L_{22}} \qquad (2.3.22)$$

Point $\vec{D}$ can be calculated using Equation 2.3.2.

When segment B alone is degenerate, then point $\vec{D}$ is set equal to $\vec{B}$\, and point $\vec{C}$ is found by computing the point on segment A that is nearest to point $\vec{D}$. This involves computing a value for parameter *s* only, from Equation 2.3.23, which is analogous to Equation 2.3.21.

$$\vec{L}A^* = \overrightarrow{AB} \qquad (2\text{-}3.23)$$

Solving for *s* yields:

$$s = \frac{\text{-p-}}{\text{Ai}} \qquad (2.3.24)$$

Note that Equation 2.3.24 is identical to Equation 2.3.13 with fset equal to zero. Since *t* equals zero at point $\vec{B}_{lt}$ our derivation here is consistent with the derivation for nondegenerate lines.

Certainly, a nice way to handle the cases where only one segment is degenerate is to write a single subroutine that is used both when segment A alone is degenerate and when B alone is degenerate. It is possible to do this using either Equation 2.3.22 or Equation 2.3.24, as long as the variables are treated properly. The implementation provided on the companion CD-ROM uses Equation 2.3.24 for both cases, with parameters passed in such that the degenerate line is always treated as segment B, and the nondegenerate line is always treated as segment A.

(^CBj5| mm a

It is also easy to relate die parallel line special case to the equations developed previously, although it is not quite as obvious as the degenerate case. Here, we have to remember that L_{12} *is* the negative dot product of the vectors $\vec{L}_A$ and $\vec{L}_B$, and when the lines are parallel, the dot product is equal to the negative of the length of $\vec{L}_A$ times the length of $\vec{L}_B$. The determinant of the matrix in Equation 2.3-9 is given by $L_nL_22 - L_u$, and this is equal to zero when L_{12} is equal in magnitude to the length of $\vec{L}_A$ times the length of $\vec{L}_B$. Thus, when the line segments are parallel, Equation 2.3.9 is singular and we cannot solve for *s* and *t*.

In the case of infinite parallel lines, every point on line *A* is equidistant from line *B*. If it is important to find the distance between lines *A* and *B*, simply choose $\vec{C}$ to be equal to $\vec{A}_{lt}$ and then use Equations 2.3.22 and 2.3.2 to find $\vec{D}$. Then, the distance between $\vec{C}$ and $\vec{D}$ is the distance between the two segments. We'll look at how to handle finite length segments in the next section.

Coding Efficiency

For coding efficiency, you should check first for degenerate lines, and then for parallel lines. This approach eliminates the need to calculate some of the convenience variables from Equations 2.3.14 and 2.3.15 when one or both of the lines are degenerate.

Dealing with Finite Line Segments

The previous two sections treated infinite lines. This is useful; however, there are perhaps many more situations in game development when it is required to process finite line segments. So, how do we adjust the results shown previously to deal with finite-length line segments?

Line Segments that Are Not Parallel

If Equations 2.3.12 and 2.3.13 generate values of s and t that are both within the range [0,1], then we don't need to do anything at all, since the finite length line segment results happen to be identical to the infinite line results. Whenever one or both of s and rare outside of [0,1], then we have to adjust the results. For nonparallel lines, there are two possibilities: 1) s or t is outside of [0,1] and the other is inside [0,1]; and 2) both s and rare outside of [0,1]. Figure 2.3.2 illustrates these two cases.

For the case when just one of s or t is outside of [0,1], as in Figure 2.3.2a, all we need to do is:

1. Clamp the out-of-range parameter to [0,1].
2. Compute the point on the line for the new parameter. This is the nearest point for the first segment.
3. Find the point on the other line that is nearest to the new point on the first line, with the nearest point calculation performed for a finite line segment. This is the nearest point for the second segment.

In the last step, just clamp the value from Equation 2.3.22 to [0,1] before calculating the point on the other segment.

For the case when both s and t are outside of [0,1], as in Figure 2.3.2b, the situation is slightly more complicated. The process is exactly the same except that we have to make a decision about which segment to use in the previous process. For example, if we selected line segment^ in Figure 2.3.2b, step 2 would produce point $\vec{A}_2$. Then, step 3 would produce point $\vec{5}_{ls}$ the nearest point on segment B to point $\vec{A}_2$. The pair of points, $\vec{A}_2$ and $\vec{BI}$ clear jv are *not* the correct points to choose for $\vec{C}$and $\vec{D}$. Point $\vec{B}_l$ is the correct choice for D, but there is a point on segment A that is much closer to segment B than $\vec{A}_2$. In fact, the point generated by step 3 will always be the correct choice for either $\vec{C}$ or $\vec{D}$. It is the point from step 2 that is incorrect. We can compute the other nearest point by just using the result from step 3. The process for both s and t outside of [0,1] then becomes:

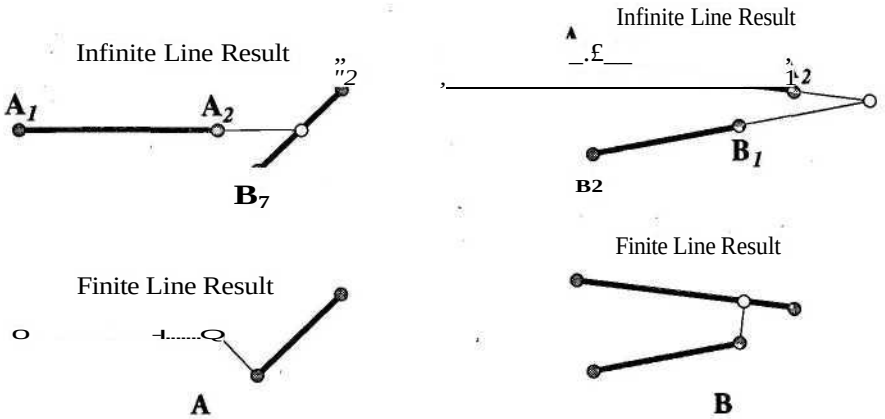


FIGURE 2.3.2 *Finite-length line segments. A) Either* sortis *outside of[0,1]. B) Both s and t are outside of[0,1].*

1. Choose a segment and clamp its out-of-range parameter to [0,1].
2. Compute the point on the line for the new parameter. This is *not guaranteed* to be the nearest point for the first segment!
3. Find the point on the other line that is nearest to the new point on the first line, with the nearest point calculation performed for a finite line segment. This is the nearest point for the second line segment.
4. Find the point on the first line segment that is nearest to the point that resulted from step 3. This is the nearest point for the first line segment.

If we select segment B in Figure 2.3.2b as our initial segment to correct, we would immediately select point 5, and step 3 would give the point between $\vec{A}_1$ and $\vec{A}_2$. In this case, step 4 is not required. The implementation provided here does not bother to check for this situation.

Line Segments that Are Parallel

There are two basic possible scenarios when the two segments are parallel, both of which are illustrated in Figure 2.3.3. First, there might be a single unique pair of nearest points, shown in Figure 2.3.3a. This always occurs when the projection of both segments into a line parallel to both do not overlap. Second, there might be a locus of possible nearest point pairs, shown in Figure 2.3.3b. Here, we could choose the two nearest points to be any pair of nearest points between the two vertical gray lines. The implementation provided on the accompanying CD-ROM selects the nearest points for finite length, overlapping parallel line segments to be halfway between the gray lines; that is, at the midpoint of the overlapping portion of each segment.

It is important to note that when the two segments are parallel, or almost parallel, the nearest points computed by this algorithm will often move erratically as the lines

FIGURE 2.3.3 *Parallel line segments. A) Unique nearest point pair. B) Locus of nearest pointpairs.*

are rotated slightly. The algorithm will not fail in this case, but the results can be confusing and problematic, as the nearest points jump back and forth between the ends of the segments. This is illustrated in Figure 2.3.4.

Shown in Figure 2.3.4a, the nearest points will stay at the far left until the lines become exactly parallel, at which point the nearest points will jump to the middle of the overlap section. Then, as the lines continue to rotate past parallel, the nearest points will jump to the far right, shown in Figure 2.3.4b. This behavior may be problematic in some game applications. It is possible to treat the behavior by using a different approach to selecting the nearest point when lines are parallel or near parallel. For example, you could implement a rule that arbitrarily selects the point nearest $\vec{A}$\ as the nearest point on segment A when the segments are parallel within, say, 5 degrees of each other. To avoid the erratic behavior at the 5-degree boundary, you would need to blend this arbitrary nearest point with an algorithmically generated nearest point between, say, 5 and 10 degrees, with the arbitrary solution being 100% at 5 degrees and 0% at 10 degrees. This solution will increase the expense of the algorithm. There are certainly other approaches, including ones that may be simpler, cheaper, and more reliable. The implementation provided on the companion CD-ROM does not attempt to manage this behavior.

FIGURE 2.3.4 *Erratic movement of nearest points for nearly parallel line segments. A) Nearestpoints at the left. B) Nearestpoints at the right.*

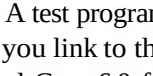
Implementation Description

The implementation includes four C-language functions, contained in the files *lineintersect_utils.h* and *lineintersect_utils.cpp.* The primary interface is the function *IntersectLineSegments,* which takes parameters defining the two line segments, and returns points $\vec{C}$, $\vec{D}$, and $\vec{P}$, as well as a vector between points $\vec{C}$and $\vec{D}$. The function

also takes a parameter indicating whether you want the line segments to be treated as infinite lines, and a tolerance parameter to be used to check the degenerate and parallel line special cases. The vector between $\vec{C}$ and $\vec{D}$ can be used outside of the implementation to determine a distance between the lines. It is important to note that the vector is not necessarily normal to either of the line segments if the lines are finite. If the lines are infinite and at least one is not degenerate, the vector will be normal to the nondegenerate line(s). The supporting functions are as follows:

- *FindNearestPointOnLineSegment* calculates the point on a line segment that is nearest to a given point in three-dimensional space.
- *FindNearestPointOjParallelLineSegments* calculates representative (and possibly unique) values for $\vec{C}$and $\vec{D}$ for the case of parallel lines/segments.
- *AdjustNearestPoints* adjusts the values of $\vec{C}$ and $\vec{D}$ from an infinite line solution to a finite length line segment solution.

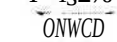
The code is documented with references to the text.



A test program is also provided, called *line_intersection_demo.* The demo requires that you link to the GLUT library for OpenGL. Project files are present for Microsoft Visual C++ 6.0 for Windows. It should not be too difficult to port this to other systems that support OpenGL and GLUT.

Opportunities to Optimize

The implementation source code was written carefully, but without any attempt to optimize for a particular processor or instruction set. There are a number of opportunities in every code to optimize the implementation for a given platform. In this case, perhaps the biggest opportunity is in the area of vectorization. There are numerous operations in this code that require a multiply or addition/subtraction operation on all three elements of a vector. These are prime opportunities to vectorize. Additionally, if you have an instruction set that supports high-level operations such as dot products, take advantage when evaluating Equation (2.3.15), for example. To truly maximize the performance, I strongly recommend that you use a professional code profiling utility to identify bottlenecks and opportunities for your target platform(s).



The text presented here and the implementation provided on the accompanying CD-ROM is rigorous, and treats every conceivable situation. The code is generally efficient, but in the case where the infinite lines intersect outside of the range of the finite segments (in other words, one or both *ofs* and *t* are outside of [0,1]), the true nearest points are not necessarily cheap to compute. In fact, the nearest point problem we've solved here is a minimization problem, and as is the case in general, the cost increases when constraints are applied to minimization problems. Beyond processor/platform-specific optimizations, it is certainly possible to remove parts of the implementation that are not required for your application. For example, if you do not need to treat finite length segments, remove everything that deals with finite length

segments. Just have the main function return a bool that is true when the nearest point is found between the finite segment endpoints, and false when the nearest point is found outside the finite segment endpoints.

Conclusions

The algorithm discussed here is rigorous and capable of handling any line intersection problem without failing. Depending on your particular use of line intersections, you may need to adjust the algorithm; for example, to manage the idiosyncrasies that arise when two finite segments are nearly parallel, or to remove the processing of finite segments when you only deal with infinite lines. I sincerely hope that some of you will benefit from this formal discussion of line and line segment intersections, along with ready-to-use source code.

References

[Golub96] Golub, Gene H., and Charles F. van Loan, *Matrix Computations, Third Edition*, The Johns Hopkins University Press, 1996.

[O'Neil87] O'Neil, Peter V., *Advanced Engineering Mathematics, Second Edition*, Wadsworth Publishing Company, 1987.