FASTER LINE SEGMENT INTERSECTION

Franklin Antonio
QUALCOMM, Incorporated
Del Mar, California

Problem

Given two line segments in 2-D space, *rapidly* determine whether they intersect or not, and determine the point of intersection.

Rapid calculation of line segment intersections is important because this function is often a primitive called many thousands of times in the inner loops of other algorithms. An algorithm is presented here that uses approximately half as many operations as the intersection algorithm presented in Graphic Gems II (Prasad, 1991) and has tested faster on a variety of computers.

Algorithm

To develop the algorithm, it is convenient to use vector representation. Consider line segment $L12$ defined by two endpoints $P1$ and $P2$, and $L34$ defined by endpoints $P3$ and $P4$, as shown in Fig. 1.

Represent each point as a 2-D vector, i.e., $\mathbf{P}1 = (x1, y1)$, etc. Then a point $\mathbf{P}$ anywhere on the line $L12$ can be represented parametrically by a linear combination of $\mathbf{P}1$ and $\mathbf{P}2$ as follows, where α is in the interval [0, 1] when representing a point on the line *segment* $L12$:

$$\mathbf{P} = \alpha\mathbf{P}1 + (1 - \alpha)\mathbf{P}2. \tag{1}$$

This can be rewritten in a more convenient form:

$$\mathbf{P} = \mathbf{P}1 + \alpha(\mathbf{P}2 - \mathbf{P}1) \tag{2}$$

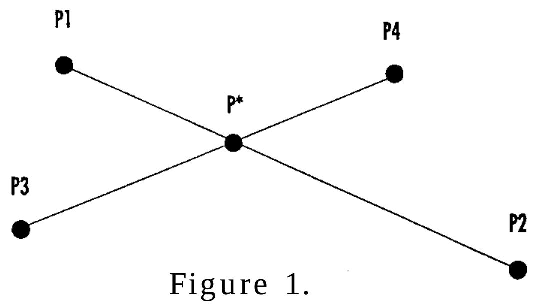

Figure 1.

In particular, we can locate the intersection point P^* by computing α and β by solution of the following linear system of equations. If the resulting a and b are in the interval [0, 1], then the line segments $L12$ and $L34$ intersect.

$$\mathbf{P}^* = \mathbf{P}1 + \alpha(\mathbf{P}2 - \mathbf{P}1), \tag{3a}$$

$$\mathbf{P}^* = \mathbf{P}3 + \beta(\mathbf{P}4 - \mathbf{P}3), \tag{3b}$$

Subtracting these equations yields

$$0 = (\mathbf{P}1 - \mathbf{P}3) + \alpha(\mathbf{P}2 - \mathbf{P}1) + \beta(\mathbf{P}3 - \mathbf{P}4) \tag{4}$$

Giving names to some intermediate values makes the equations easier to read:

$$\mathbf{A} = \mathbf{P}2 - \mathbf{P}1$$

$$\mathbf{B} = \mathbf{P}3 - \mathbf{P}4$$

$$\mathbf{C} = \mathbf{P}1 - \mathbf{P}3 \tag{5}$$

The solution of (4) for α and β is now

$$\alpha = \frac{ByCx - BxCy}{AyBx - AxBy}, \tag{6a}$$

$$\beta = \frac{AxCy - AyCx}{AyBx - AxBy}. \tag{6b}$$

Noting that the denominators of these expressions are the same, computing (5) and (6) requires nine adds and six multiplies in the worst case. (Prasad, 1991, required 14 adds and 12 multiplies in the worst case.) We avoid the division operation because we don't need α and β explicitly; we only need test them against the range [0, 1]. This is a little tricky, because the denominator may be either positive or negative. The division-avoiding test works like this:

```
if denominator > 0
        then if numerator < 0 or numerator > denominator
                then segments do not intersect
        else if numerator > 0 or numerator < denominator
                then segments do not intersect
```

Also note that (6a) can be computed and tested prior to computing (6b). If α is outside of [0, 1], there is no need to compute (6b). Finally, note that the case where denominator = 0 corresponds to collinear line segments.

Timing Measurements/Further Optimizations

The algorithm above was timed against Prasad (1991), using random data, on a variety of computers (both RISCs and CISCs). It was found to execute 40% faster than Prasad's intersection test.

Because some tests can be performed on the numerator of (6a) and (6b) after knowing only the *sign* of the denominator, but without knowledge of its value, it is tempting to use a form of the fast-sign-of-cross-product algorithm (Ritter, 1991) on the denominator, after which the numerator can be tested against zero, eliminating some cases prior to *any* multiplications. In practice, this was found to slow the algorithm, as in this arrangement of the intersection algorithm the tests in question did not eliminate a large enough fraction of the test cases to pay back for the execution time of the many sign tests in Ritter's algorithm.

An additional speed increase *was* gained by testing to see if the bounding boxes of the two line segments intersect before testing whether the line segments intersect. This decreases the number of arithmetic operations when the boxes do not intersect, but involves an additional overhead of several comparisons in every case. Therefore, there is a risk that this might actually slow down the average performance of the algorithm. However, the bounding box test was found to generate an additional 20% speed improvement, which was surprisingly consistent across different types of computer. Therefore, the C implementation includes a bounding box test.

Implementation Notes

The C implementation includes calculation of the intersection point coordinates in the case where the segments are found to intersect. This is accomplished via Eq. (3a).

The C implementation follows the same calling conventions as Prasad (1991) and produces identical results.

The C implementation uses integer arithmetic. Therefore, as Prasad warns, care should be taken to avoid overflow. Calculation of the intersection point involves operations that are cubic in the input coordinates, so limitation to input coordinates in the range [0, 1023], or other similar-sized range, will avoid overflow on 32-bit computers. When line segments do *not* intersect, input coordinates in the range [0,16383] can be handled.

See also G2, 7; G3, D.4.