The Arithmetic Precision of Ray-Polygon Intersection Testing

Jack Snoeyink*

Abstract

By looking for algorithms that require minimum arithmetic precision, we obtain a simple ray-polygon intersection test that more easily handles degenerate cases.

1 Introduction

The problem of determining whether a ray hits a polygon in 3-space has a standard solution: first compute the intersection point where the ray hits the plane of the polygon, then project along one coordinate to reduce to a 2-dimensional Point-in-Polygon problem. Badouel's Graphic Gems article [1] gives details using a point-in-polygon test that is restricted to convex polygons. This multistep process is quite fast, because the inner loop is simple, but it does have two disadvantages. First, degenerate cases from each step need to be handled correctly. Second, using the derived values from the initial intersection increases the arithmetic precision required, as we shall see in Section 3. Möller and Trumbore [8] have a single-step process for ray/triangle intersection that could be modified to address these two concerns for triangles, but not for polygons.

Recently, researchers in computational geometry have been evaluating the arithmetic precision of basic geometric algorithms [3, 4, 7] by studying the degrees of polynomials computed to resolve geometric tests. In this note, we consider the ray-polygon intersection test, and derive a simple, degree three algorithm that handles all degeneracies.

The next section shows that a standard point-in-polygon test is degree two. Section 3 extends this test to a ray-polygon intersection test with degree three, and shows that Badouel's intersection and projection gives degree five. Both sections emphasize coordinate-free derivation.

When implemented in floating point, the intersection/projection approach is more prone to error, but can be faster due to the projection. In graphics, speed is often more important than accuracy.

2 Preliminaries, predicates, and point-in-polygon testing

We can begin by considering a point-in-polygon test. The input is a query point q and an n-vertex polygon P, represented as a sequence $p_0, p_1, \ldots, p_n$, with $p_n = p_0$ so that we need not worry about wrap-around. The test should decide whether q is inside, outside, or on the boundary of P. An approach first published by Shimrat [9], with a bug fix by Hacker [5], is to count whether moving q to infinity crosses an even or odd number of edges of P. Haines surveys this and other point-in-polygon tests [6].

We can represent points as homogeneous column vectors, e.g., $p_i = (p_i.x, p_i.y, 1)^T$ in three dimensions and $p_i = (p_i.x, p_i.y, p_i.z, 1)^T$ in four. We can represent the 2D point at infinity along the x-axis as $\infty = (1, 0, 0)^T$. (For mathematical details, see Stolfi's work on oriented projective geometry [10]. Although I prefer to put the homogenizing coordinate first in all dimensions, in this note I have put it last to conform with OpenGL [2].)

We will use brackets to denote the determinant for three two-dimensional points:

$$[a, b, c] = \begin{vmatrix} a.x & b.x & c.x \\ a.y & b.y & c.y \\ 1 & 1 & 1 \end{vmatrix}$$
$$= (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x).$$

Bracket $[a, b, c]$ computes the signed area of the parallelogram defined by $b - a$ and $c - a$. We determine whether c is left, right or on the directed line $\overrightarrow{ab}$ by testing whether $[a, b, c]$ is positive, negative, or zero, respectively.

Now, we can count crossings between $\overrightarrow{q\infty}$ and edges of P using the code in Algorithm 1. For each

*Department of Computer Science, UNC Chapel Hill, CB 3175 Sitterson Hall, Chapel Hill, NC, USA, 27599–3175. `snoeyink@cs.unc.edu` Research partially supported by NSF grants 9988742 and 0076984.

```
PointInPolygon(q, P) classifies q as IN, ON, or OUT of P.

Procedure PointInPolygon(q, P) {
    boolean in = false;
    for (i = 0; i < n; i++)                                   /* Check each edge */
        if (([q, ∞, p_i] < 0) ≠ ([q, ∞, p_{i+1}] < 0)) {
            if ([p_i, p_{i+1}, q] == 0) return ON;            /* q is on p_i p_{i+1} */
            else if (([p_i, p_{i+1}, q] < 0) ≠ ([p_i, p_{i+1}, ∞] < 0))
                in =!in;                                      /* q∞ crosses edge */
        }
    return (in ? IN : OUT);                                   /* Odd=IN; Even=OUT */
}
```

Algorithm 1: The crossing algorithm for the point-in-polygon test

edge $\overline{p_i p_{i+1}}$ we check if the endpoints straddle the line $\overleftrightarrow{q\infty}$. If so, we check if q is on the edge or if q and ∞ are on opposite sides of the directed line $\overrightarrow{p_i p_{i+1}}$.

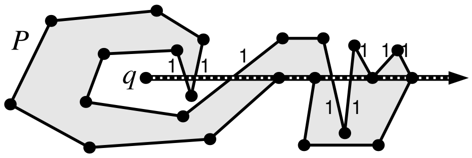

Figure 1: Handling degeneracies for point-in-polygon

Notice how the code correctly handles degenerate cases when the ray $\overrightarrow{q\infty}$ passes through a vertex or edge of P. A crossing is counted only when one endpoint is left of $\overrightarrow{q\infty}$ and the other endpoint is not, which is consistent with a slight upward perturbation of the ray $\overrightarrow{q\infty}$. Thus, eight crossings are counted in Figure 1, and q is OUT.

Expanding the bracket operator clearly shows that it is a degree two polynomial in the input variables. If we use k bit inputs, then $2k$ bits are sufficient to guarantee that the computation is performed correctly. This precision is necessary for the bracket that does not involve ∞; the other brackets actually reduce to coordinate comparisons, which are degree one polynomials.

3 Ray-polygon intersection test

Next we turn to the ray-polygon test, whose input is a query ray $\overrightarrow{qr}$, and an n-vertex polygon P whose vertices are now homogeneous four-tuples: $p_i = (p_i.x, p_i.y, p_i.z, 1)^T$. The test should decide whether the ray hits the polygon. We can move the entire ray to infinity and again count the edges crossed.

As in the previous section, we use the bracket to denote a determinant:

$$[a, b, c, d] = -\begin{vmatrix} a.x & b.x & c.x & d.x \\ a.y & b.y & c.y & d.y \\ a.z & b.z & c.z & d.z \\ 1 & 1 & 1 & 1 \end{vmatrix},$$

(We negate the sign in odd dimensions when the homogenizing coordinate is last.)

As Figure 2 illustrates, the sign of the bracket has two geometric interpretations, each corresponding to a right-hand rule. It tells whether point d is above, below, or on a plane through a, b, and c, where "above" is the direction of a right-handed screw at a that turns from b toward c. Equivalently, it tells whether a screw directed along the ray $\overrightarrow{ab}$ turns in the direction of $\overrightarrow{cd}$ or in the opposite direction. In either interpretation, the result is zero iff the four points are coplanar.

Figure 3 illustrates a little more notation for the ray-polygon intersection test. Let d be the point at infinity on the ray $\overrightarrow{qr}$; the coordinates of d are those of $q - r$ with $d.w = 0$. Choose another point at infinity not colinear with $\overline{qr}$, such as $\infty = (1, 0, 0, 0)$ if $d.y \neq 0$ or $d.z \neq 0$, and $\infty = (0, 1, 0, 0)$ otherwise. Let π be a row vector that holds the homogeneous coordinates of the plane equation for polygon P. That is, $\pi \cdot p_i = 0$. We can obtain π by finding three

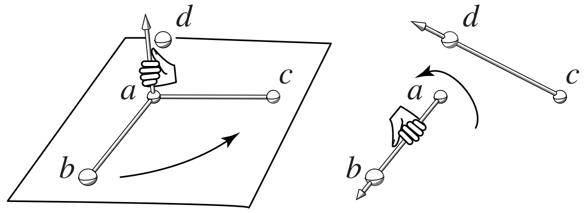

Figure 2: $[a, b, c, d]$ is positive according to a right-hand rule for planes or for lines

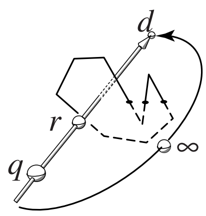

Figure 3:
Ray-polygon test

`RayPolygon`(q, r, P, π) classifies $\overrightarrow{qr}$ as HIT, GRAZE, or MISS with respect to polygon P. Row vector π is the plane containing P.

Procedure `RayPolygon`(q, r, P, π) {
    boolean hit = **false**; $d = (r - q)$; /* *Pick points at infinity* */
    $\infty = (d.y \neq 0 \,||\, d.z \neq 0)\ ?\ (1,0,0,0)^T : (0,1,0,0)^T$;
    if $((\pi \cdot q) == (\pi \cdot d))$ return MISS; /* $\overrightarrow{qr}$ *misses plane* */
    for $(i = 0;\ i < n;\ i{+}{+})$ /* *Check each edge* */
        if $(([q, r, \infty, p_i] < 0) \neq ([q, r, \infty, p_{i+1}] < 0))$ {
            if $([p_i, p_{i+1}, q, r] == 0)$ return GRAZE; /* *Hits edge* */
            else if $(([p_i, p_{i+1}, q, r] < 0) \neq ([p_i, p_{i+1}, \infty, d] < 0))$
                $hit = !hit$; /* *Count edge* */
        }
    return (hit ? HIT : MISS);
}

Algorithm 2: Ray-polygon intersection test

non-colinear points p_0, p_i, and p_j, writing the equation $\pi \cdot (X, Y, Z, W) = [p_0, p_i, p_j, (X, Y, Z, W)^T]$, and equating coefficients of X, Y, Z, and W. Thus, the coefficients of π are minors of the bracket determinant.

The ray-polygon intersection test in Algorithm 2 is a simple extension of the point-in-polygon test. It first checks if the ray $\overrightarrow{qr}$ misses the plane of the polygon. It then checks each edge $\overline{p_i p_{i+1}}$ that straddles the plane defined by q, r, ∞, and counts those for which the directed lines $\overrightarrow{qd}$ and $\overrightarrow{\infty d}$ do not have the same orientation. Degeneracies are handled as before.

In `RayPolygon`(), the bracket $[p_i, p_{i+1}, q, r]$ is a degree three polynomial, as are tests against the plane π. All other tests are degree two.

To compare this against the intersection/projection approach, we could parameterize the ray as $r(t) = q + td$ and solve for the value of t that satisfies $\pi \cdot r(t) = 0$. This $t = (\pi \cdot q)/(\pi \cdot d)$ is a rational polynomial with numerator and denominator each of degree three. The intersection point $q' = q + td$ is therefore degree four, with a degree three homogenizing coordinate. Dropping a coordinate, and applying `PointInPolygon`(q', P), we see that we must evaluate $[p_i, p_{i+1}, q']$, which is a degree five polynomial. Without dropping a coordinate the polynomial would have degree six.

Suppose that we use IEEE 754 floating point arithmetic to evaluate these tests on integer input. A careful implementation of the intersection/projection approach using rational arithmetic could guarantee that 10-bit inputs would not overflow the 53 bits of IEEE doubles. `RayPolygon`() would guarantee that 17-bit inputs would not overflow IEEE doubles. We can refine this analysis if we assume that the polygon vertices are close together, and far from the points defining the $\overrightarrow{qr}$. If the coordinates take d bits and differences between nearby points take $b < d$ bits, then the two approaches take d^2b^3 bits and db^2 bits, respectively, plus a couple of bits for rounding. Thus, if polygon edge coordinates may differ by 1000, the safe distance from the eye can be thousands for intersection/projection and billions (2^{30}) for `RayPolygon`().

When operations are implemented exactly using a multiprecision package or fixed point, then `RayPolygon`() requires less arithmetic or allows a larger domain for input. When operations are implemented in floating point, then fewer bits are lost to rounding, increasing the probability of correct decisions in the algorithm. So the real question is, "How important is correctness for a ray-polygon intersection test?" In graphics, correctness is often secondary to speed. The algorithm for `RayPolygon`() can be optimized by expanding brackets by minors to take the invariant parts of the computation outside the loop. For $[p_i, p_{p+1}, q, r]$, expansion by 2×2 minors gives the Plücker coordinates of the edges of the polygon and of the ray, and reduces the determinant to a dot product of 6-tuples. Bucketting can also be used to speed up evaluation at the cost of some preprocessing and storage. Even after optimization, however, implementations of the intersection/projection approach that calulate a floating point representation of the intersection point will use fewer multiplications because they reduce to a two-dimensional problem.

Finally, we note that if a ray hits several polygons, then to determine which is hit first we can compare the parameter values of the intersection of the ray with the plane. These values are rational numbers with numerator and denominator of degree 3, so their comparison computes a degree six polynomial. When correctness is important, then higher precision is needed for this step.

4 Conclusion

By seeking an algorithm with minimum precision using coordinate-free methods, we derived a simple, general algorithm that handles all degenerate cases. It should be considered as an alternative to the intersection/projection approach when correctness is important.

References

[1] Didier Badouel. An efficient ray-polygon intersection. In Andrew S. Glassner, editor, *Graphics Gems*, pages 390–393. Academic Press, Boston, MA, 1990.

[2] OpenGL Architecture Board. *OpenGL Reference Manual*. Addison-Wesley, 1993.

[3] J.-D. Boissonnat and F. P. Preparata. Robust plane sweep for intersecting segments. *SIAM J. Comput.*, 29(5):1401–1421, 2000.

[4] J.-D. Boissonnat and J. Snoeyink. Efficient algorithms for line and curve segment intersection using restricted predicates. *Comput. Geom. Theory Appl.*, 16(1):35–52, 2000.

[5] R. Hacker. Certification of algorithm 112: position of point relative to polygon. *Commun. ACM*, 5:606, 1962.

[6] Eric Haines. Point in polygon strategies. In Paul Heckbert, editor, *Graphics Gems IV*, pages 24–46. Academic Press, Boston, MA, 1994.

[7] Giuseppe Liotta, Franco P. Preparata, and Roberto Tamassia. Robust proximity queries: an illustration of degree-driven algorithm design. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 156–165, 1997.

[8] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[9] M. Shimrat. Algorithm 112: position of point relative to polygon. *Commun. ACM*, 5:434, 1962.

[10] J. Stolfi. *Oriented Projective Geometry: A Framework for Geometric Computations*. Academic Press, New York, NY, 1991.