A Representation and Algorithm for Exact Computation of Cascaded Polygon Intersections with Fixed Storage Requirements

Clifford A. Shaffer and Charles D. Feustel

TR 91-29

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

September 25, 1991

A REPRESENTATION AND ALGORITHM FOR EXACT COMPUTATION OF CASCADED POLYGON INTERSECTIONS WITH FIXED STORAGE REQUIREMENTS

Clifford A. Shaffer*
Charles D. Feustel**

*Department of Computer Science and
**Department of Mathematics
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

ABSTRACT

Given a collection of polygons (or a collection of sets of polygons) with vertex points specified to some fixed-point precision, a technique is presented for accurately representing the intersection points formed by elements in the collection. In particular, we recognize that in 2D, the result of a series of intersections between polygons must be composed of line segments that are part of the line segments making up the original polygons. While our techniques greatly improve the accuracy of floating point representations for intersection points, we also provide algorithms based on rational arithmetic that compute these intersection points exactly. The storage required to represent an intersection vertex requires a fixed amount of storage slightly greater than four times the number of bits in the input resolution. Furthermore, no intermediate quantity requires resolution slightly greater than four times the resolution resolution of input vertex values, so implementation on existing computers at practical resolution can easily be done. Cascaded intersection operations do not require ever greater amounts of storage for vertex points, as would normally be required by a rational arithmetic approach. We also prove that a similar approach is not possible for any reasonable set of rotations.

September 25, 1991

1. INTRODUCTION

Computing the union or intersection of two or more objects is an important operation in computational geometry. It is of fundamental importance in Geographic Information Systems (GIS) and CAD applications, and is also used in Computer Graphics animation and object space hidden surface removal algorithms [Warn69, Weil77]. In this paper, we treat the problem of taking the intersection of polygons represented by a series of line segments defining the boundary (i.e., the traditional boundary representation in 2D). In particular, we wish to take two sets of polygons, and return the set of polygons describing the area contained in both input sets. Furthermore, we wish to take this result, and then produce the intersection with a third set of polygons (itself possibly the result of an intersection operation). This type of operation would be used in CAD systems based on constructive solid geometry (whether half-plane or primitive polygons/polyhedra) and in GIS for analysis of the interactions between several map overlays.

While many intersection algorithms exist, nearly all implementations have major shortcomings with respect to the accuracy of the result (see [Hoff89a, Hoff89b] for a detailed discussion of the potential pitfalls and several suggested solutions). In particular, simply using floating point values to represent input vertex coordinates and computed intersection point coordinates leads to serious and well known problems of numeric stability. At each step, either increased precision is required, or else the accuracy of the result is reduced. Reduced accuracy leads to topological inconsistencies such as query points that are determined to fall within each of a pair of input polygons, but not within their intersection. A series of cascaded intersections (i.e., the intersection of polygons A and B intersected with C, intersected with D, ...) further increases the problems since typically either the accuracy becomes arbitrarily small, or the precision required to represent a vertex becomes unbounded.

One approach to dealing with the problem is to conceptualize the vertex points and resulting line segments as a fuzzy specification with a region of uncertainty (see [Sega90] and related work).

The size of the uncertainty region grows rapidly with more intersections. Another approach is to use rational arithmetic of infinite precision (for example, [Fran86]). When an intersection takes place, the size of the numerator and denominator of the fraction representing the coordinate of the intersection point grow as necessary for exact representation. If input resolution required D bits, then each intersection calculation can require the size of each vertex to grow by D bits.

We present a technique for representing the intersections between line segments with vertices of a fixed resolution. We recognize that the output polygon from a series of intersection operations must be composed of pieces of the line segments that make up the input polygons. By keeping track of the necessary input line segments (in their original form) along with descriptions of intersection points along these line segments in parametric form, we greatly increase the accuracy of the resulting intersection polygon when floating point numbers are used to represent the parameter values. Furthermore, if rational arithmetic is used, we can limit the size of the representation for such intersection points to be less than four times the size of the input vertex points. This approach is similar to the method for an exact rational arithmetic representation presented in [Hoff89a]. However, our method, based on fixed precision vertex points rather than a fixed set of lines, is simple to implement and quite efficient. This paper provides a precise implementation of the method. Our empirical timings of the method compared favorably to a floating point representation for the parameters. Finally our method, unlike that of [Hoff89a], is closed under translation of vertex points by integer units of the minimum resolution.

Note that we do not deal in any way with accuracy of the input data. Particularly in GIS, there is a serious debate on how to manage error when input data is not entirely correct, yet sufficiently well correlated to result in tiny "sliver" polygons (see [Good89]). Our output will be no more "correct" than the input data. However, our approach allows us to avoid creating any additional errors due to numeric operations.

2. METHOD

We restrict input vertex values to be fixed precision numbers. In particular, we assume for the remainder of this paper that the vertex coordinates are integers in the range -16383 to 16383 – i.e., 14 value bits and one sign bit. If necessary we could consider the input as the result of a "polygon normalization" operation [Mile88] where some polygon whose vertices with real number coordinates have been modified to have integer coordinates in such a way as to preserve appropriate topology. We prefer to view the input as coming from a user working at a raster-based system, whether in a CAD or GIS environment. Our suggested resolution is far greater than the resolution of modern computer monitors, and also higher than any reasonable expectation for manual precision from an input device such as a cartographic digitizing table. The stored precision could be greater if necessary, but by selecting 15 bit precision we simplify the implementation of our rational arithmetic approach.

A polygon is represented in our system as a collection of consecutive line subsegments (as opposed to the more typical view of a collection of connected vertices). A *line subsegment* is some portion of a line segment that is represented by (i) two endpoints with integer coordinates whose precision is 15 bits and (ii) two parameter values. This line segment specified by the vertex points will be referred to as the *a priori* line segment. The parameter values specify the extent of the line subsegment that actually makes up some part of the polygon boundary. These parameters represent positions some fraction of the distance from the first to the second a priori vertex points. For polygons that have not undergone any intersections, each boundary subsegment will be the complete a priori line segment. Thus, the parameter values of such subsegments will be 0 and 1 (specifying the beginning and the end of the a priori line segment, respectively). In addition, one could use a direction flag to indicate whether this particular line subsegment goes from vertex 1 to vertex 2, or vice versa. This would allow for sharing of a priori line segment objects among a set of polygons. This approach has some similarity to that used in [Nels86] for representing linear feature data in a quadtree to allow for recombination of data that has be split by an intersection operation.

The combination of a priori line segments and associated parameters for computed polygon boundary subsegments is the key to exact precision calculations within bounded space. This approach requires that the parameter values merely be representable and comparable. Most importantly, we need not represent the multiplication of two such parameters, with the attendant loss of accuracy (or increase in precision). If we choose to use floating point numbers for parameter representation, the result will always be consistent so long as all possible parameter values are distinguishable. This depends on the resolution both of the vertex points and the floating point number.

Exactness can be guaranteed through the use of rational arithmetic. Here, parameters are described by two 32 bit quantities, representing the numerator and denominator of the parameter. The denominator will always be a positive 32 bit integer. For parameters describing points on the line segment, the numerator will be between 0 and the value of the denominator (i.e., the parameter is between 0 and 1). For computed results, the numerator can be outside this range when the point of intersection for the two line segments does not actually fall on the line segments.

Our intersection routines consist of calculating the parametric values of the intersection point between two a priori line segments (i.e., one set of numerator and denominator for each line segment). Given these parameter values, we now need only decide if the new intersection point lies to the left or the right of the old one along a subsegment to determine the intersection polygon's new boundary line.

The parameter value for the intersection point between two line subsegments is calculated as follows. First, compute the intersection of the a priori line segments, taken from the structure definitions of the subsegments. The intersection routine returns TRUE if the line segments intersect and are not parallel, and FALSE otherwise. A structure is also returned that describes the intersection location by its parameter values along each of the two intersecting a priori line segments, and information about the direction of crossing for each parameter – i.e, whether the line intersection is going from in to out, or out to in.

We now describe how to calculate the parameter values for the intersection (if there is one). First we check bounding boxes for a quick non-intersection test. Assuming the bounding boxes intersect, we generate the equation for the infinite line from the endpoints for one a priori line segment. Substituting the two endpoints from the other line into this equation results in two values each of which is negative, positive, or zero. If the signs of the two values are the same, then the two lines do not intersect (i.e., both endpoints of the second line segment are to one side of the first line segment). If the signs are different, then we repeat the process with the roles of the two lines reversed. If the result of this second test gives two values with different signs, then we know that there is an intersection between the a priori line segments. This is stronger than saying that the a priori lines extended to infinity have an intersection. If the lines extended to infinity cross, but the line segments themselves do not cross, then one pair of intersection parameters will have different signs, but the other pair will have identical signs.

The equations that we use for the substitution of the endpoints of line 1 into the equation for line 2 are given below. This will be important later when we explain how to calculate the actual intersection point. We use the following line equation:

$$(Y - Y_1)(X_2 - X_1) - (X - X_1)(Y_2 - Y_1) = 0$$

when substituting point (X, Y) into the line with endpoints (X_1, Y_1) to (X_2, Y_2). We evaluate the equation four times, first substituting the endpoints of line segment Q into the equation for line segment P, then substituting the endpoints of line segment P into the equation for line segment Q. This is done by calculating the four quantities a, b, c and d defined as follows:

$$\begin{aligned}
a &= (Q_{1y} - P_{1y})(P_{2x} - P_{1x}) - (Q_{1x} - P_{1x})(P_{2y} - P_{1y})\\
b &= (Q_{2y} - P_{1y})(P_{2x} - P_{1x}) - (Q_{2x} - P_{1x})(P_{2y} - P_{1y})\\
c &= (P_{1y} - Q_{1y})(Q_{2x} - Q_{1x}) - (P_{1x} - Q_{1x})(Q_{2y} - Q_{1y})\\
d &= (P_{2y} - Q_{1y})(Q_{2x} - Q_{1x}) - (P_{2x} - Q_{1x})(Q_{2y} - Q_{1y})
\end{aligned}$$

Once we know that there is a proper intersection between the two a priori line segments, we must calculate the two intersection parameters, each defined by their numerator and denominator. To do this, we need to solve the following vector equation for parametric variables s and t:

$$(1-t)P_1 + t * P_2 = (1-s)Q_1 + s * Q_2$$

which can be rewritten as

$$t(P_2 - P_1) + s(Q_1 - Q_2) = Q_1 - P_1.$$

We solve this equation by calculating three determinants. We define $s = \text{sdet}/\text{det}$ and $t = \text{tdet}/\text{det}$ where det, sdet, and tdet are defined as:

$$\begin{aligned}
\text{det} &= (P_{2x} - P_{1x})(Q_{1y} - Q_{2y}) - (P_{2y} - P_{1y})(Q_{1x} - Q_{2x})\\
\text{sdet} &= (P_{2x} - P_{1x})(Q_{1y} - P_{1y}) - (P_{2y} - P_{1y})(Q_{1x} - P_{1x})\\
\text{tdet} &= (Q_{1x} - P_{1x})(Q_{1y} - Q_{2y}) - (Q_{1y} - P_{1y})(Q_{1x} - Q_{2x})
\end{aligned}$$

By suitable rearrangement, we find that $\text{det} = a - b$, $\text{sdet} = a$ and $\text{tdet} = -c$. Thus, our work to check if there is a proper intersection between the two line segments by substituting the endpoints into line equations provides most of the calculation required to determine the parameters of the intersection point. Note that each parameter (s and t) is actually stored as a numerator and denominator of a fraction. It should be easy to see from the above equations that if the initial endpoints for lines P and Q require n bits for their representation, than these numerators and denominators will each require at most $2n + 1$ bits plus a sign bit. The Algorithm in the appendix shows Pascal-like pseudo-code for calculating the intersection point between two line subsegments.

Given the parameter definition for an intersection point, we must also be able to calculate the relative position of two such intersection points along a line segment. In other words, we must determine if one intersection point is to the left or to the right of another intersection point along the line segment. This is used in the polygon set operation algorithms to do their work of deciding which parts of the a priori line segments will actually be in the answer. This calculation is straightforward since doing so is equivalent to deciding which of the fractions $\frac{N_1}{D_1}$ and $\frac{N_2}{D_2}$ is greater by comparing the two products $N_1 * D_2$ and $N_2 * D_1$. This requires a (temporary) further doubling of the resolution, to $4n + 4$ bits, including the sign bit. For our implementation, we allow original vertex points to be 15 bit unsigned integers. The numerator and denominator of the parameters s and t are each 32 bit signed integers. The intermediate calculations for comparing the magnitude of the parameters require multiplication of two 32 bit quantities, which can easily be done in software if 64 bit integers are not provided by the compiler.

3. ROTATIONS

Ideally we would like to extend our algorithm for exact intersections to include rotations. So that all computations can be made exactly, rotation matrices should have rational entries, and the numerators and denominators of these entries should have a fixed bounded magnitude. We also require a "reasonable" number of distinct rotations – while the four rotations provided by multiples of 90° meet our requirements for rational entries in the rotation matrix, clearly four rotations are not enough!

Since we want the operations of rotation and intersection to commute, the set of rotations should be closed under matrix multiplication. We show below that one cannot satisfy these constraints, first offering a proof for 2D then extending it to 3D.

We suppose that G is a finite set of 2×2 rotation matrices closed under multiplication, and each element of G has rational coefficients. From a theorem of Da Vinci [Wey52], any finite semigroup (having n elements) of direct isometries (i.e., distance preserving transformations) of the plane is a cyclic group of rotations. In particular, any finite semigroup of rotation matrices is a cyclic group; i.e., the elements have inverses and there is an identity rotation. Note that the set must be finite because of the limitation on size of the numerator and denominator of the rotation matrix entries. There is a natural isomorphism f of the set of rotation matrices onto the set of complex numbers with modulus 1 which carries the matrix

$$\begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$$

to $\cos\alpha + i\sin\alpha$. Since the group G is now seen to be a finite cyclic group of order n, if $x \in G$ and $f(x) = \cos\alpha + i\sin\alpha$, there is an integer k such that $n\alpha = 2k\pi$.

It follows that the set G is generated by a rotation of $\frac{2\pi}{n}$ radians. Visually, imagine a vertical line segment that can be subjected to n rotations, forming n equal size pie slices. Thus $f(G)$ is the set of roots of $x^n - 1 = 0$ (this should be familiar to anyone who has studied Fast Fourier Transforms). Let ζ be a root of $x^n - 1 = 0$. Then $\overline{\zeta}$ (i.e., the complex conjugate of ζ) is also a root, so ζ and $\overline{\zeta}$ are algebraic integers. $(\zeta + \overline{\zeta}) = 2\mathrm{Re}(\zeta)$ is an algebraic integer and rational. This forces $\zeta + \overline{\zeta}$ to be an integer [Lang71]. $|\zeta + \overline{\zeta}| \leq |\zeta| + |\overline{\zeta}| = 2$. Thus, Re (ζ) can only be -1, $-\frac{1}{2}$, 0, $\frac{1}{2}$, or 1. We note that Re $(\zeta) = \frac{1}{2}$ implies $\mathrm{Im}(\zeta) = \sqrt{1 - \left(\frac{1}{2}\right)^2} = \frac{\sqrt{3}}{2}$ which is irrational, and conclude that the only possible rotations are by multiples of 90°.

In 3D, a finite set of matrices, each of which has determinate one, which is closed under multiplication, is a group. The finite subgroups of the orthogonal group, which consists of matrices such that $AA^T = I$, are well known (see any book on crystallography). Those that have rational coordinates have order less than or equal to 24. In other words, the largest finite subgroup is the set of rotations which carry a cube to itself [Yale68].

From the foregoing proofs, we conclude that approximations must be made. One can either (1) take an exact rotation matrix, perform the rotation, and then truncate the result (the approach suggested in [Hoff89a]) or (2) approximate each rotation matrix. Using floating point numbers is effectively equivalent to (1). We see that in (2) the product of matrices approximating rotations of $\alpha°$ and $\beta°$ is only an approximation to the matrix approximating a rotation of $(\alpha + \beta)°$. In both cases, exactness is lost; however, in (2) the linear transformation representing a rotation has determinate near one and is thus a topological map, i.e., a continuous one-to-one map which distorts objects slightly, but does not displace vertices across lines.

We can extend the methods previously described to support some operations involving rotations. Specifically, we can do the following requiring only a fixed increase in storage. We can rotate an object by some specified angle (actually, we then do an operation that is *almost* a rotation by that angle, and is topologically consistent). By using the standard technique of storing the original polygon and its rotation angle, we can do an arbitrary number of rotations on a polygon, preserving all topological properties without a growth in storage requirements. We can take the intersection of two rotated objects (if the rotations are the same), and it will identical to the rotation of the intersection of the original objects. We can perform point in polygon on a polygon with some query point (which is the intersection point of two a priori lines) and get the same answer as we would get if we rotated the polygon and the query point, and then performed the query.

4. CONCLUSIONS

We performed an experiment to compare the time required to use our exact intersection representation with the floating point representation of parameters. Note that with 15 bit vertices and 64 bit floating point values, we were unable to find any case where the floating point representation produces incorrect results. We took the intersection of a large number of quadrilaterals generated at random such that they all share a common central region so that the intersection would not be empty. We performed the test both on a DECstation 5000 (for 10,000 polygons) and a MacII (for 1,000 polygons) so as to compare the performance on different architectures. The program was implemented using the C programming language. On both machines we found that both versions of the program ran in roughly the same time, with the floating point version being perhaps

20% faster than the integer version. However, neither implementation was optimized. Code tuning could easily change the relative timings. Thus, the most that we can conclude from this experiment is that use of the accurate intersection approach is not significantly slower than the floating point approach.

We have presented the details for representing one or more collections of polygons so as to accurately compute their intersection. This representation is quite easy to implement, and does not have unreasonable space requirements. It is quite efficient as compared to straightforward floating point calculations. We also demonstrated that a generalized scheme for accurate computation with rotations is not possible.

The 2D version of this method as described in this paper can be used directly for GIS using a vector model for representing regions, 2D CAD systems, and object-space hidden surface removal algorithms (i.e., such as Warnock's algorithm [Warn69] or Weiler and Atherton's algorithm [Weil77]. Our method should be fairly easy to convert to 3D. A related approach based on a fixed set of planes is presented in [Hoff89a].

5. ACKNOWLEDGEMENTS

We wish to express our thanks to Daniel Farkas, who provided the proofs in Section 3.

6. REFERENCES

1. [Fran86] W.R. Franklin, P.Y.F. Wu and S. Samaddar, Prolog and geometry objects, *IEEE Computer Graphics and Applications 6*, 11(November 1986), 46-55.

2. [Good89] M. Goodchild and S. Gopal, Eds., *Accuracy of Spatial Databases*, Taylor & Francis, London, 1989.

3. [Hoff89a] C.M. Hoffman, *Geometric & Solid Modeling: An Introduction*, Morgan Kaufmann, San Mateo, CA, 1989.

4. [Hoff89b] C.M. Hoffman, The problems of accuracy and robustness in geometric computation, *IEEE Computer 22*, 3(March 1989), 31-41.

5. [Lang71] S. Lang, *Algebra*, Addison Wesley, 1971, p. 240.

6. [Mile88] V. Milenkovic, Verifiable implementations of geometric algorithms using finite precision arithmetic, *Artificial Intelligence 37*, (1988), 377-401.

7. [Nels86] R.C. Nelson and H. Samet, A consistent hierarchical representation for vector data, *Computer Graphics 20*, 4(August 1986), 197-206.

8. [Sega90] M. Segal, Using tolerances to guarantee valid polyhedral modeling results, *Computer Graphics 24*, 4(August 1990), 105-114.

9. [Warn69] J.E. Warnock, A hidden line algorithm for halftone picture representation, TR 4-15, Computer Science Department, University of Utah, Salt Lake City, Utah, 1969.

10. [Weil77] K. Weiler and P. Atherton, Hidden surface removal using polygon area sorting, *Computer Graphics 11*, 2(Summer 1977), 214-222.

11. [Weyl52] H. Weyl, *Symmetry*, Princeton University Press, 1952, p. 65.

12. [Yale68] P.B. Yale, *Geometry and Symmetry*, Holden-Day, 1968, p. 107.

7. APPENDIX

The algorithms presented in this appendix are written in PASCAL, augmented with a **return** construct. The following subroutines are used in the pseudocode without formal definition.

maxmin(a, b, *maxval*, *minval*) returns in *maxval* the greater of values a and b, and returns in *minval* the lesser of values a and b.

sign(x) returns 1 if x is positive, 0 if x is 0, and -1 if x is negative.

```
type
  DBLINT = array [0..1] of integer; { 64 bit integer }
  POINTRANGE = -16383 .. 16383; { Vertex value range }
  ENDPOINT = record { A priori segment vertex }
    x, y : POINTRANGE
  end;
  SEGMENT = record { A priori line segment }
    first, last : ENDPOINT
  end;
  PARAM = record { Intersection parameter along SEGMENT }
    num, denom : integer
  end;
  SUBSEGMENT = record { A computed subsegment }
    apriori : SEGMENT;
    direction : boolean;
    ParOne, ParTwo : PARAM
  end;
```

```
    INTPOINT = record { An intersection point descriptor }
      seg : array [0..1] of SEGMENT;
      par : array [0..1] of PARAM;
      a, b, c, d : integer
    end;

  function SubSegmentIntersect(ss1, ss2 : ↑SUBSEGMENT; var ipt : ↑INTPOINT) : boolean;
    { Calculate and return in ipt the intersection point for two subsegments. This function calls
      SegmentIntersect to check for intersection of the a priori line segments, and then checks to
      see if the intersection point actually falls within the parameters defining the endpoints of the
      subsegments. }
  begin { We assume that lines always go from the first endpoint to the second }
    if (not SegmentIntersect(ss1↑apriori, ss2↑apriori, ipt)) then return(FALSE);
    if (LessThan(ipt↑par[0], ss1↑ParOne) or GreaterThan(ipt↑par[0], ss1↑ParTwo)) then
      return(FALSE)
    if (LessThan(ipt↑par[1], ss2↑ParOne) or GreaterThan(ipt↑par[1], ss2↑ParTwo)) then
      return(FALSE)
    return(TRUE)
  end;

  function SegmentIntersect(p, q : ↑SEGMENT; var ipt : ↑INTPOINT) : boolean;
    { Return in ipt the descriptor for the intersection of a priori line segments p and q if there is such
      an intersection. The function returns TRUE iff there is an intersection. }
  var
    max1, min1, max2, min2 : integer; { For computing bounding boxes }
    a, b, c, d : integer; { See equations in Section 2 }
  begin
    { Check if bounding boxes for the segments intersect }
    maxmin(p↑first.x, p↑last.x, max1, min1);
    maxmin(q↑first.x, q↑last.x, max2, min2);
    if ((max1 < min2) or (min1 > max2)) then return(FALSE);
    maxmin(p↑first.y, p↑last.y, max1, min1);
    maxmin(q↑first.y, q↑last.y, max2, min2);
    if ((max1 < min2) or (min1 > max2)) then return(FALSE);

    { Solve vector equation $(1 - t) * p1 + t * p2 = (1 - s) * q1 + s * q2.$ }
    a := (q↑first.y − p↑first.y) ∗ (p↑last.x − p↑first.x) − q↑first.x − p↑first.y) ∗ (p↑last.y − p↑first.y);
    b := (q↑last.y − p↑first.y) ∗ (p↑last.y − p↑first.y) − (q↑last.x − p↑first.x) ∗ (p↑last.y − p↑first.y);
    if (sign(a) = sign(b)) then return(FALSE); { points both on same side or on line }

    c := (p↑first.y − q↑first.y) ∗ (q↑last.x − q↑first.x) − (p↑first.x − q↑first.x) ∗ (q↑last.y − q↑first.y);
    d := (p↑last.y − q↑first.y) ∗ (q↑last.x − q↑first.x) − (p↑last.x − q↑first.x) ∗ (q↑last.y − q↑first.y);
    if (sign(c) = sign(d)) then return(FALSE); { points both on same side }
    ipt↑seg[0] = p;    ipt↑seg[1] = q;

    { Used to determine directions of crossing }
    ipt↑a := a;   ipt↑b := b;   ipt↑c := c;   ipt↑d := d;

    if ((a − b) > 0) then
    begin
    ipt↑par[0].denom = a − b;
    ipt↑par[0].num = −c;
    ipt↑par[1].denom = a − b;
    ipt↑par[1].num = a
    endelse { at this point, (a − b) must be less than 0 }
    begin
    ipt↑par[0].denom = b − a;
    ipt↑par[0].num = c;
    ipt↑par[1].denom = b − a;
    ipt↑par[1].num = −a
    end
end;

function GreaterThan(Par1, Par2 : PARAM) : boolean;
{ Return TRUE iff line subsegment parameter Par1 is greater than Par2. }
begin
  return(LessThan(Par2, Par1))
end;

function LessThan(Par1, Par2 : PARAM) : boolean;
{ Return TRUE iff line subsegment parameter Par1 is less than Par2. }
var Res1, Res2 : DBLINT;
begin
  { Assume that all components are non-negative since this code is used with proper intersections
  of line segments. }
  mult(Par1.num, Par2.denom, Res1);
  mult(Par2.num, Par1.denom, Res2);
  if (Res1[1] < Res2[1]) then return(TRUE)
  else if (Res1[1] = Res2[1]) then
    if (Res1[0] < Res2[0]) then return(TRUE);
  return(FALSE)
end;

procedure mult(long1, long2 : integer; var Result : DBLINT);
{ Multiply two 32 bit integers, returning the result into array Result. Input values long1 and long2
  are positive. }
begin
  Result[0] = (long1 mod 2^16) * (long2 mod 2^16);
  Result[1] = (long1 div 2^16) * (long2 div 2^16);
  temp = (long1 mod 2^16) * (long2 div 2^16);
  temp = temp + (long1 div 2^16) * (long2 mod 2^16);
  Result[1] = Result[1] + (temp div 2^16);
  Result[0] = Result[0] + ((temp mod 2^16) * 2^16)
end;
```