Inside-polygon() algorithm for curvilinear contours. Design and Implementation

Fernando Luis Cacciola Carballal
May, 1999
cacciola@sminter.com.ar

Abstract

This paper describes an algorithm to determine whether a point is inside or outside a curvilinear polygon, based on the well known algorithm consisting of counting the intersections of an horizontal ray with the polygon.

Curvilinear polygons, as analytic resources, are not common in computer graphics, but they do arise on some domains. Normally, parametric curves, such as cubic splines or beziers, are used to model curves in graphic applications. But parametric curves cannot be easily manipulated in analytic computations. For example, obtaining a curve parallel to another is not a simple task, and is not even possible under all circumstances.
For this reason, some computational geometry applications use *non-parametric polyline curves*, formed by straight-line segments and circular arcs. The straightforward geometry of lines and circumferences makes it easy to develop complex algorithms to work with non-parametric curves.
But despite the fact that this curves are easy to treat analytically, they lack the parametric benefits of classical curves; and for this reason, it is very hard to find in the literature any work about them.

If we are to represent curvilinear polygons using patches of straight-line segments and circular arcs, named polyline curvilinear polygons, one of the fundamental operations that we need to implement is the point-in-polygon test.
Haven't been able to find any such algorithm, the author adapted the classical method used with straight-line polygons, and extended it for use with polyline curves.

Part I. Design.

General Method:

Let **C** be a polyline curvilinear polygon, made up of an ordered and oriented sequence of straight-line segments and circular arcs.
Let **p** be a 2D point.
The following method can be used to determine whether **p** is inside or outside **C**.

1. Let **L** be a straight-line segment starting at **p**, and ending at $q=(MaxX,p.y)$
   L is a ray originating at **p** and extending to the right, beyond the rightest coordinate of **C**.

2. Count the number of ***switching intersections*** between **L** and **C**.

3. If the number of switching intersections is odd, **p** is inside; otherwise, **p** is outside **C**.

Switching intersections:

Definition 1: A *switching intersection* is an intersection between **L** and **C**, s.t. **L** passes from the interior of **C** to its exterior, or vice versa.
Definition 2: A *non-switching intersection* is an intersection between **L** and **C**, s.t. **L** remains in the interior, or exterior of **C**.

A *switching intersection* implies that **L crosses C**.
A *non-switching intersection* implies that **L only-touches C**, without crossing it.
We found non-switching intersections at some vertices, and at second-order intersections on circular arcs.

If two intersections occur at the same point, they represent a single second-order intersection.
Circular edges report second-order intersection when the crossing line is tangent to the circumference of the arc.
Straight-edges report only first order intersections.

It should not consider the starting point of the edges.
It should be able to report up to 2 intersections in the case of a circular edge.
Each intersection must have an 'order' associated with it. If 2 intersections with the same edge occur at the same point, only 1 intersection, with order 2, should be reported.
It should not report continuous intersections. For example, two collinear straight-line segments do not intersect.
It does not have to report the coordinates of the intersection; only its existence and information indicating if they occur at the ending point of the edge.

Intersections at the vertices.

Statement 1: A first-order intersection not at the ending point of an edge is a switching intersection.
Statement 2: A second-order intersection not at the ending point of an edge is a non-switching intersection.

A problem arises at the vertices, because there is a parametric discontinuity at those points.
Whenever an intersection occurs at the ending point of an edge (a vertex), we need to determine if **L** is crossing **C**, or just touching it:

Analysis of straight-only edges:

Consider a polygon made up of straight edges.
Consider a ray, **L**, intersecting an edge at its ending point. The ending point of an edge is a vertex of the polygon.
Let **Curr** be the intersected edge, and let **v** be its ending point (a vertex).
Assume **Curr** is not collinear with **L**; if **L** intersects **Curr** at **v** then **Curr** is above or below **L**.
Now consider the next edge, **Next**, and assume it's not collinear with **L**. Then **Next** is also above or below **L**.

It can be seen that if both **Curr** and **Next** lie on the same side of **L** (above or below), then **L** is not crossing the polygon, but touching it at **v**. Otherwise, if **Curr** and **Next** lie on opposite sides of **L**, then **L** crosses **C**.

Statement 1 :A straight edge cannot cross **L** more than once.
Statement 2: Being **v** the intersection of **L** and a straight edge **E**, if **v** is an extreme (starting/ending point) of the edge, **E** is entirely *Above, Below or Over* **L.**

Definition 2: **S'(E)** is an *open segment* of **L** which is under the cone of influence of an edge. That is, the locus of points closer to that edge than to any other.

Statement 3: For a straight edge **E**, **S'(E)** is entirely *Inside, Outside or Over* **C**.

Definition 3: **PosL'(E),** is the position of a straight edge **E** w.r.t **L.**
It can be *Above, Over or Below.*
Above and *Below* are opposite positions.
Over is not opposite to any other position.

Definition 4: **PosS'(E),** is the position of **S'(E)** w.r.t **C.**
It can be *Inside, Over or Outside.*
Inside and *Outside* are opposite positions.
Over is not opposite to any other position.

PosS'(E) is directly defined by **PosL'(E);** that is, the position of a segment of **L** w.r.t. **C**, is defined by the position of the edge **E** w.r.t **L.**

In plain words, the above definitions and statements help in observing how the horizontal ray is getting in and out the polygon. By observing how the edges relate to **L** (above, below or over), as we move along the polygon, we are observing how **L** relates to **C** (inside, outside or over).

Statement 4: If **PosS'(Curr)** is *opposite* to **PosS'(Next)**, the vertex intersection is a switching intersection.
If **PosS'(Curr)** is *equal* to **PosS'(Next)**, the vertex intersection is a non-switching intersection.

Clearly, if **PosS'(Curr)** is opposite to **PosS'(Next)**, then the open segment of **L** under, or above **Curr** is in opposite position w.r.t the open segment above, or under **Next**. Thus, **L** crosses **C** at the intersection.
Similarly, if **PosS'(Curr)** is equal to **PosS'(Next)** , both open segments of **L** are in the *interior* or the *exterior* of **C**, and **L** touches **C**, but doesn't crosses it.

If **PosS'(Next)** is *Over*, the situation is undefined. We cannot determine how **L** relates to **C** at this point. We need to look forward for the first edge whose **PosS'(edge)** is not *Over*.
In the following, **Next** is the first edge not over **L**.

Analysis of circular edges.

Statement 4 is based on statement 1; with circular arcs, we need other statements to extend the concept behind statement 4.

Statement 5: A circular arc may cross **L** at most twice.

Suppose that **Curr** is an arc. Since **L** intersects the vertex **v**, and the ending point of **Curr** is **v**, one of the possible intersections is at **v**.
According to statement 5, only one more intersection might be found for this edge.
This intersection cannot occur at the starting point of **Curr**, because we don't consider intersections at the starting point of edges.
Therefore, the following is true:

Statement 6: If there is only 1 intersection between **L** and an edge, that edge is entirely *Above, Below o Over* **L**.
Statement 7: If there are 2 intersections between **L** and an edge, that edge can be conceptually splited in two **portions, each of one is** entirely located at opposite sides of **L**.
This definition is based on the constraint imposed on the intersection routine: second order intersections are reported as 1 single intersection.

We denote a general edge portion as **edge***.
The first portion, noted **edge>**, is the sub arc going from the starting point to the first intersection.
The last portion, noted **<edge**, is the sub arc going from the first intersection to the ending point.
(The notation **edge*** is used when there is no need to specify first or last portion).

If there is only 1 intersection, being **E** a straight or circular edge, **E***, **E>** and **<E** denote the entire edge. (**E≡E*,<E,E>**)

Statements 6 and 7 are a generalization of Statement 1 for straight or circular edges.

Definition 5: **S(E*)** is the open segment of **L** which is under the cone of influence of an edge portion. That is, the locus of points closer to that edge portion than to any other edge. This definition does not compare different portions of the same edge.

Definition 6: **PosL(E*)**, is the position of an edge portion **E*** w.r.t **L**.
It can be *Above, Over or Below*.
Above and *Below* are opposite positions.
Over is not opposite to any other position.

Definition 7: **PosS(E)**, is the position of **S(E)** w.r.t **C**.
It can be *Inside, Over or Outside.*
Inside and *Outside* are opposite positions.
Over is not opposite to any other position.

Like **PosS'(E)**, **PosS(E*)** is directly defined by **PosL(E*)**.
The trick here is that **E*** is not the same as **E**; it is a portion of **E** which is entirely at one side of **L**.

Statement 8: If **PosS(Curr>)** is *opposite* to **PosS(<Next)**, the vertex intersection is a switching intersection.
If **PosS(Curr>)** is *equal* to **PosS(<Next)**, the vertex intersection is a non-switching intersection.

The treatment of the possible values for **PosS(Curr>)** and **PosS(<Next)** remains as stated in the first analysis.

Algorithm.

In order to turn these definitions into an algorithm, we need to devise a way to uniquely obtain **PosS(Curr>)** and **PosS(<Next)** for an arbitrary edge.

Statement 9: Given the edge **E** which has exactly 1 intersection with **L** at **v**, where **v** is the ending point of **Curr**, **PosS(E*)** is given by (**p.y - L.y**); where **p** is the starting point of **E** when **E** is **Curr**, and its ending point for **E** being **Next.**

Statement 10: Given the edge **E** which has exactly 2 first-order intersections with **L**, being one at **v**, where **v** is the ending point of **Curr**, **PosS(E*)** is given by (**L.y-p.y**), where **p** is the starting point of **E** when **E** is **Curr**, and its ending point for **E** being **Next.**

Whenever **Curr** has intersections with **L**, we need to determine if any one of them is at its ending point.
If **Curr** intersects **L** at **v**, where **v** is the ending point of **Curr**, we are in a *Vertex Situation.*
If **Curr** intersects **L**, but not at **v**, we are in a *non Vertex Situation.*

Vertex situations:

As explained in the preceding paragraphs, we need to classify the vertex intersection. If we regard it as a switching intersection, we count it; otherwise we don't.
If we have found 2 intersections for **Curr** (they can only be first order), then 1 is directly regarded as switching and counted. This is correct since Statement 7 says that the two portions of the arc are at opposite positions w.r.t **L**, and therefore **C** is at opposite positions w.r.t **L** on each arc portion, so **L** crosses **C** and that extra intersection is switching.

Using statement 9 and 10, we can setup the procedure position() which calculates the position of the ray w.r.t the polygon on each edge. (The procedure is shown in the code)

The algorithm looks as follow:

```
Scan each edge in turn.
   If the Curr edge has any intersection with L.
      If at least 1 intersection is at a vertex
         Take the Next non-aligned nor over edge.
         Compute position(Curr) and position(Next).
         Compare positions.
         If they are not equal
            Count the intersection.
      If no intersection is at a vertex
         Count only first-order intersections.
```

In order to compute **position(Next)**, we need to find the intersections of **Next** with **L**, because the procedure needs information about how many intersections there are.
An optimization can be introduced testing whether **Next** is a straight line or circular arc. If it is a straight line, **position(Next)** can be computed directly from its ending point; if it is a circular arc, we need to test for its intersection with **L**, and apply the position() procedure.

If **PosS(<Next)** is *Over*, we have to search for the first edge not Over. This is true because:
If the edge is a straight edge, **PosS(<Next)==Over** means the edge is geometrically over **L**, so it is aligned with it.
If the edge is a circular edge, **PosS(<Next)==Over** means that **L** switches sides twice, in this case, the situation with **L** w.r.t **C** is undefined just as if **Next** were aligned with **L**.

Part II. Implementation.

Representations:

Unfortunately, since we need to represent polyline polygons formed with circular arcs, we need to use floating point coordinates.
Because of this, a technique called *bounded evaluation* will be used in order to assure robustness.
Bounded evaluation defines the *operation equal* as: `A=B iif |A-B|<e.`
The number **e**, called **epsilon**, is domain specific. For that reason, it is not a constant but a parameter passed along the different parts of the algorithm. (Some programs uses the standard epsilon value defined as the difference between two adjacent floating point representations, but I choose to leave this user defined).

We will represent the polygon as a linked list of edges. There are straight and arc edges.
A straight edge is defined by its endpoints.

An arc edge is defined by its center, radius, endpoints, endangles and a flag indicating if the angles are measured counter-clockwise or clockwise. Some applications force ccw orientation; I wanted to show how this is not a constraint of the algorithm so I provided support for arbitrary orientations.

The intersection routines:

Because the ray is an horizontal line, we can optimize the intersection routines.
Intersections at the ending points of the edges are central because they indicate a vertex situation. Therefore, the intersection routines test for this case specifically.
For straight edges, horizontal edges or edges lying completely at one side of the ray have no intersections, and this can be detected easily by comparing coordinates.
For arc edges, if the ray is tangent to the circumference, we have a second order intersection. According to the definitions and statements shown in the algorithm design, second order intersections should be counted only in vertex situations; indeed, only tangent intersections that occur at the ending point of the arc are reported. In this way, second order intersections are simply never reported, and the code is simplified because it doesn't need to account for 2 types of intersections. The arc intersection routine uses a trigonometric approach; there are others, but I choose this one because I believe it is numerically stable.
Because the ending point of an edge is the starting point of the next edge, we cannot report intersections that occur at the starting point.
Null edges, that is, edges in which the starting point and ending point are equal, or zero-radius arcs, are assumed not to appear. There are no special considerations for this in the code, but if they might arise in practice, additional tests should be added.

Endpoint intersections:

Because a vertex is a point shared by two adjacent edges, if an intersection at the ending point of one edge has been reported, the intersections at the starting point of the next edge should be discarded.
Because of the limitations of floating point calculations, we use a bounded evaluation to determine if the intersection is at the ending point of the edge. This evaluation defines that 2 points are equal if their Euclidean distance is lower than some number R.
This can be seen as using a disk of radius R centered at the endpoint of the edge, if the ray enters the disk it intersects the edge.
Similarly, for the next edge, we put the disk at its starting point, and if the ray enters the disk, it doesn't intersect the edge.
In the code, because there are specific comparisons arranged so that fewer calculations are actually performed, this disk is conceptually applied in each stage.
What needs to be assured is that if an intersection is reported at the endpoint of an edge, it is not reported again at the starting point of the next edge. To assure this, the code is designed so that starting and ending point tests are mutually exclusive.

The code:

The following code is C++ code. It does not use the language extensively in order to keep it generic. For this reason, the edge structure is a super structure with all the information for both straight and arc edges; this is not the way you would do this in C++, but it fits the purpose of this paper.
The code is self-contained, so general functions and definitions have been included.
I've chosen to improve in clarity rather than in efficiency, so the code is dissected in a lot of functions, including C++ inline functions for small calculations. The C++ inline facility let this code be fast even if it appears to have too many function calls.

```
////////////////////////////////////////////////////////////////////////////////////////
// Algorithm polygon_into() for curvilinear polyline polygons.
// 1999, Fernando Luis Cacciola Carballal.
//
//----------------------------------------------------------------------------------------
// Representations.
//
// point: 2D floating-point coordinates.
// edge: straight and arc edge.
struct point { point ( double _x = 0 , double _y = 0 ) : x ( _x ) , y ( _y ) {}
               double x , y ;
             } ;
const int cStraight = 0 ;
const int cArc      = 1 ;
struct edge
{
       edge ( void ) { ZeroMemory ( this , sizeof ( edge ) ) ; }
       int    mType ;
       point  mSP , mEP , mC ;
       double mR , mSA , mEA ;
       bool   mCCW ;
       edge * mNext ;
} ;
//----------------------------------------------------------------------------------------
```

```
//-----------------------------------------------------------------------------------------------
// Auxiliary functions.
inline bool equal ( double a , double b , double epsilon )
  return ( a > b ? ( a - b ) < epsilon : ( b - a ) < epsilon ) ; }
inline bool fitClosedInterval ( double l , double n , double h )
{ return ( l <= n ) && ( n <= h ) ; }
inline bool fitOpenInterval ( double l , double n , double h )
{ return ( l < n ) && ( n < h ) ; }
inline double abs ( double n )
{ return n >= 0 ? n : - n ; }
inline double pow2 ( double n )
{ return n * n ; }
inline double distance ( const point & a , const point & b )
{ return sqrt ( pow2 ( b.x - a.x )  + pow2 ( b.y - a.y ) ) ; }
inline bool equal ( const point & a , const point & b , double epsilon )
{ return distance ( a , b ) < epsilon ; }
//-----------------------------------------------------------------------------------------------


//-----------------------------------------------------------------------------------------------
// function position:
//
// Returns the position of the edge with respect to the ray at 'y'.
// It could be Below(-1), Over(0) or Above(1).
//
// 'aEdge': the edge to be considered.
// 'y': y-coordinate of ray.
// 'aExtraIsecs': Needed if the edge is an arc. See algorithm description.
// 'aCurrOrNext': Indicates if the edge is the curr or next edge.
// 'aEpsilon': Aithmetic epsilon. (1e-5 is usually a good value)
//
const int cBelow   = - 1 ; // return value from position()
const int cOver    = 0   ; // return value from position()
const int cAbove   = + 1 ; // return value from position()
const int cCurrent = 0   ; // indicates current edge.
const int cNext    = 1   ; // indicates next edge.

inline int position ( const edge & aEdge , double y , int aExtraIsecs ,  int aCurrOrNext , double aEpsilon )
{
   double ey = ( aCurrOrNext == cCurrent ? aEdge.mSP.y : aEdge.mEP.y ) ;
   double d = ( aExtraIsecs ? y - ey : ey - y ) ;
   return equal ( d , 0.0 , aEpsilon ) ? cOver : d > 0 ? cAbove : cBelow ;
}
//-----------------------------------------------------------------------------------------------


////////////////////////////////////////////////////////////////////////////////////////////////
// function IntersectStraightEdge
//
// Test is the straight edge intersects the horizontal ray to the right of 'aP'
//
// If aEdge is incident upon the ray, report 1 vertex intersection.
// Otherwise, check if aEdge straddles the ray. If does, compute and test intersection.
//
static int IntersectStraightEdge ( const edge &  aEdge   ,
                                   const point & aP      ,
                                   double        aDiskR  ,
                                   bool &        rVertex
                                 )

   int rIsecs = 0 ;
   rVertex = false ;

   bool lHorizontal = equal ( aEdge.mSP.y , aEdge.mEP.y , aDiskR ) ;

   // If the edge is horizontal we skip it, because it dosen't intersect the
   // ray (is parallel or coincident with it).
   if ( ! lHorizontal )
   {
        double lLx = aP.x - aDiskR ;
        bool lToRay = equal ( aEdge.mEP.y , aP.y , aDiskR ) ;

        // If the edge is incidient upon the ray, this is a vertex intersection.
        if ( lToRay && aEdge.mEP.x > lLx )
        {
             rIsecs  = 1    ;
             rVertex = true ;
        }
        else
        {
             // See if the edge's ending points are on oposite sides of the ray.
             bool lFAbove = aEdge.mSP.y > aP.y ;
```

```
            bool lTAbove = aEdge.mEP.y > aP.y ;

            if ( lFAbove != lTAbove )
            {
                 // If the edge straddles the ray, check actual intersection.
                 double dx  = aEdge.mEP.x - aEdge.mSP.x ;
                 double dy  = aEdge.mEP.y - aEdge.mSP.y ;
                 double d   = ( aEdge.mEP.y - aP.y ) * dx / dy ;
                 double lIx = aEdge.mEP.x - d ;

                 if ( lIx > lLx )
                 {
                      point lI ( lIx , aP.y ) ;
                      if ( ! equal ( lI , aEdge.mSP , aDiskR ) )
                      {
                           rIsecs  = 1     ;
                           rVertex = false ;
                      }
                 }
            }
       }
   }
   return rIsecs ;
}
//
////////////////////////////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////////////////////////////
// function IntersectArcEdge
//
// Test if the arc edge intersects the horizontal ray to the right of 'aP'
// The following cases are specially tested and considered:
//    The ray is tangent to the circunference:
//       A vertex intersection is reported if the ray starts from the left of the center,
//       and the arc ends at 90 or 270 deg. (that is, the arc is incident upon the ray).
//    The ray is inside the circunference: 2 intersections might exist.
//       If the arc is incident upon the ray, report 1 vertex intersection.
//       Otherwise, report intersections if they are inside the arc (testing angles), and
//       the ray starts from the left of the intersection x coordinate.


//------------------------------------------------------------------------------------------------
// function fitAngleInterval
//
// Test is 'aAngle' is in the interval ['aStartA','aEndA'].
// Angles are asummed to be normalized: inside [0,2PI).
// 'aCCW' indicates if the angles are measured counter-clockwise or clockwise.
//
inline bool fitAngleInterval ( double aStartA , double aAngle , double aEndA , bool aCCW )
{
    bool rIs = false ;
    if ( aCCW )
    {
         if ( aEndA > aStartA )
         {
              if ( fitClosedInterval ( aStartA , aAngle , aEndA ) ) rIs = true ;
         }
         else
         {
              if ( ! fitOpenInterval ( aEndA , aAngle , aStartA ) ) rIs = true ;
         }
    }
    else
    {
         if ( aEndA > aStartA )
         {
              if ( ! fitOpenInterval ( aStartA , aAngle , aEndA ) ) rIs = true ;
         }
         else
         {
              if ( fitClosedInterval ( aEndA , aAngle , aStartA ) ) rIs = true ;
         }
    }
    return rIs ;
}
//------------------------------------------------------------------------------------------------

const double cPi          = 3.14159265358979323846 ;
const double cHalfPi      = cPi / 2.0 ;
const double cThreeHalfPi = 3.0 * cPi / 2.0 ;
const double cTwoPi       = 2.0 * cPi ;
```

```
ic int IntersectArcEdge ( const edge  & aArc    ,
                          const point & aP      ,
                          double        aDiskR  ,
                          bool &        rVertex
                        )

int rIsecs = 0 ;
rVertex = false ;

double lDy       = aP.y - aArc.mC.y ;
double lDyAbs    = abs ( lDy ) ;
bool   lTangent  = equal ( lDyAbs , aArc.mR , aDiskR ) ;
bool   lStraddle = lDyAbs < aArc.mR ;

Note: lTangent might be true in near-tangent conditions, but lStraddle could also
be true; indeed, lTangent should be tested first.

double lLx = aP.x - aDiskR ;
// The ray is tangent to the circunferece.
if ( lTangent && lLx < aArc.mC.x )
{
     bool lEndV = equal ( aArc.mEP.x , aArc.mC.x , aDiskR ) ;
     bool lEndAt90  = aArc.mEP.y > aArc.mC.y && lEndV ;
     bool lEndAt270 = aArc.mEP.y < aArc.mC.y && lEndV ;

     // If the ray is above the center and the arc ends at 90 deg,
     // or the ray is below and the arc ends at 270 deg,
     // there is 1 vertex isec, otherwise, tangent intersections are not counted.
     if ( ( lDy > 0 && lEndAt90  ) || ( lDy < 0 && lEndAt270 ) )
     {
          rIsecs ++ ;
          rVertex = true ;
     }
}
else if ( lStraddle )
{
     // Get intersection angles (symmetry around the Y center line).
     double lAlfa = asin ( lDyAbs / aArc.mR ) ;  // 0-radius arcs are not supported!
     double lBeta = lAlfa + cPi ;

     // Get angles in 3rd and 4th quadrants if the ray is below the center.
     if ( lDy < 0 )
     {
          lAlfa = cTwoPi - lAlfa ;
          lBeta = cTwoPi - lBeta ;
     }

     // Compute the x coordinates of the intersections.
     double lX0 = aArc.mC.x + cos ( lAlfa ) * aArc.mR ;
     double lX1 = aArc.mC.x + cos ( lBeta ) * aArc.mR ;

     point lI0 ( lX0 , aP.y ) , lI1 ( lX1 , aP.y ) ;

     // Test for a vertex situation.
     if (     ( equal ( aArc.mEP , lI0 , aDiskR ) && lX0 > lLx )
          || ( equal ( aArc.mEP , lI1 , aDiskR ) && lX1 > lLx )
        )
     {
          rIsecs  = 1    ;
          rVertex = true ;
     }
     else
     {

          // An intersection is valid if:
          //  1.Is along the ray.
          //  2.Is not at the stating point of the edge.
          //  3.Is between the arc sweep.
          if (    lX0 > lLx
               && ! equal ( aArc.mSP , lI0 , aDiskR )
               && fitAngleInterval ( aArc.mSA , lAlfa , aArc.mEA , aArc.mCCW )
             )
               rIsecs ++ ;

          if (    lX1 > lLx
               && ! equal ( aArc.mSP , lI1 , aDiskR )
               && fitAngleInterval  ( aArc.mSA , lBeta , aArc.mEA , aArc.mCCW )

             )
               rIsecs ++ ;
     }
}

return rIsecs ;
```

```
}
//
////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////
// Finds the intersections of the ray with the current edge.
// Uses the fact that the ray is an horizontal straight line.
//
int IntersectEdge ( const  edge  & aEdge    ,
                    const  point & aP       ,
                    double         aDiskR   ,
                    bool &         rVertex
                  )
{
  if ( aEdge.mType == cArc )
       return IntersectArcEdge      ( aEdge , aP , aDiskR , rVertex ) ;
  else
       return IntersectStraightEdge ( aEdge , aP , aDiskR , rVertex ) ;
}
//
////////////////////////////////////////////////////////////////////////////////////////////////


//------------------------------------------------------------------------------------------------
// Test if the point is inside the curvilinear polygon.
//
// Algorithm:
// Count the number of switching intersections with an horizontal ray started at 'aP', and
// extending to the right.
// An intersection is considered switching if it crosses the polygon.
// An intersection is not considered switching if it is tangent to the polygon, and doesn´t
// crosses it.
//
bool into_polygon ( const edge * aPoly , const point & aP )
{
   bool rInto ;

   int  lCount = 0 ;
   bool lVertex    ;

   double lDiskR = 0.7  ; // Domain spacific. Assummes that the smaller edge is > 7 in length.

   //
   // The key point is to clasify intersections as 'switching' and 'not switching'.
   //
   // If the intersection is not at a polygon's vertex, then it is switching (counted).
   //
   // If it is at a polygon's vertex (vertex situation), then
   //    Determines the position of the edge with respect to the ray: Above,Over or Below.
   //    Determines the position of the next edge with respect to the ray.
   //    If positions are different, the vertex intersection is switching, otherwise not.
   //
   const edge * lCurr = aPoly , * lNext ;
   do
   {
       lNext = lCurr -> mNext ;

       // Get intersections.
       int lIsecs = IntersectEdge ( *lCurr , aP , lDiskR , lVertex ) ;

       //-------------------------------------------------------------------------------
       // Consider intersections.
       if ( lIsecs )
       {
            if ( lVertex  )
            {
                 // If it is a vertex intersection, compare positions of curr and
                 // next edge.

                 // position() returns the position(Above,Over or Below) of the current
                 // edge w.r.t. the ray.
                 // 'lExtraIsecs' is needed if the edge is an arc.
                 int lExtraIsecs = lIsecs - 1 ;
                 int lCurrPos = position ( *lCurr , aP.y , lExtraIsecs , cCurrent , lDiskR ) ;

                 // Find position(Next).
                 // Skips edges aligned with ray (position(Next)==Over).
                 int lNextPos ;
                 for ( bool lNextFound = false ; lNext && ! lNextFound ; )
                 {
                       // position() needs 'NextExtraIsecs'.
                       // If next is a line, NextExtraIsesc is 0, but if it is
                       // an arc, we need to compute this value.
                       // In this, case, we cache the intersections info found
```

```
                    // for the next edge, so it can be used in the next loop
                    // when next becomes curr.
                    int lNextExtraIsecs = 0 ;
                    if ( lNext -> mType == cArc )
                    {
                         int lIsecs = IntersectArcEdge ( * lNext  ,
                                                         aP       ,
                                                         lDiskR   ,
                                                         lVertex
                                                       ) ;
                         lNextExtraIsecs = ( lVertex ? lIsecs - 1 : lIsecs ) ;

                    }
                    lNextPos = position ( *lNext , aP.y , lNextExtraIsecs , cNext , lDiskR )

                    // Consider Next valid only if it is not aligned with ray.
                    if ( lNextPos != cOver )
                         lNextFound = true ;
                    if ( ! lNextFound )
                         lNext = lNext -> mNext ;
               }

               // If curr and next are on different positions w.r.t. the ray, we
               // have a switching intersection.
               if ( lCurrPos != lNextPos )
                    lCount ++ ;

               // If curr is an arc, there might have been a extra intersection
               // not at the vertex, we need to count that.
               if ( lExtraIsecs )
                    lCount += lExtraIsecs ;

               // If the next edge is a straight edge, we don't need to
               // consider it again.
               if ( lNext -> mType == cStraight )
                    lNext = lNext -> mNext ;

          }
          else
          {
             // If not a vertex inersection, count first order intersections only.
               lCount += lIsecs ;
          }
     }

     lCurr = lNext ;
   }
   while ( lCurr ) ;

   // An Odd number of switching intersections means the point is inside.
   rInto = ( ( lCount % 2 ) == 1 ) ;

   return rInto ;
}
//
////////////////////////////////////////////////////////////////////////////////////////////
```

References:

[1] Eric Haines, Point in Polygon Strategies, Graphics Gems IV, Academic Press, 1994.
[2] Kevin Weiler, An Incremental Angle Point in Polygon Test, Graphics Gems IV, Academic Press, 1994.
[3] Joseph O'Rourke: Computational Geometry in C.
[4] Joseph O'Rourke: Comp.Graphics.Algorithms, FAQ: 2D Polygon Computations.
[5] Cesar N. Gon: GeoLab project, Polygon methods.
[6] Martin Held: On the Computational Geometry of Pocket Machining, LNCS 500.