SIACG 2002 - 1st Ibero-American Symposium on Computer Graphics
1-5 July 2002, Guimarães - Portugal

Algorithms for Point-Polygon Collision Detection in 2D

Juan J. Jiménez Delgado Rafael J. Segura Sánchez Francisco R. Feito Higueruela
Departamento de Informática. Escuela Politécnica Superior. Universidad de Jaén
Av. de Madrid, 35. 23071 Jaén. Spain
{juanjo,rsegura,ffeito}@ujaen.es

Abstract
In this article, several algorithms for the collision detection between a point and a polygon on a plane are presented. Coverings by triangles and barycentric coordinates are used. Temporal and geometric coherence are used for reducing their performance times. A study of times has been carried out; it shows that these simple and robust algorithms are efficient. Also, these algorithms can be used for non convex figures and they have the advantage of being easily extended to 3D.

Keywords
Collision detection, inclusion test, animation, simulation, coherence.

1. INTRODUCTION

The problem of collision detection among objects in motion is essential in several application fields, such as in simulations of the physical world, robotics, animation, manufacture, navigation by virtual worlds, etc. Apart from giving to scenes a more realistic appearance, it is necessary that the objects belonging to it, interact, so as objects do not collide, and if they do, a suitable response is obtained (objects change their trajectory or their shape, etc.) So, the development of efficient algorithms is necessary for the collision detection among objects, with the aim of improving the bottleneck that implies the checking of all pairs of features among objects of the scene, which have a performance time of O (n m p), being *n* and *m*, the number of features of objects, and *p*, the number of movements or frames in the simulation.

In previous works, a characterization of the collision detection problem and the strategies used to solve it were carried out [Jiménez02]. Other authors have also made a revision of this problem [JimTho01].

In this work, several algorithms for the collision detection between a point and a polygon on a plane are presented. The algorithms displayed try to determine if, given a point in motion, it gets in a polygon or not. We will try to reduce the number of features to deal with on each movement of objects by using the temporal coherence in their movement, as well as their geometric coherence.

Gradually we will show different algorithms, firstly for the inclusion of points in polygons (static collision detection), and then for the collision detection of points in motion in polygons (collision detection in the strict sense).

We will use a covering of the polygon by triangles with a common point on its barycenter and barycentric coordinates in order to determine the point inclusion in the polygon [Badouel90]. We have developed algorithms for different types of polygons : convex and non-convex ones, and within these last ones, algorithms for polygons with a covering of positive triangles.

The advantage of these algorithms is that they can be used for non-convex figures, that can be extended to 3D, and have showed they are robust and efficient regarding operations carried out by using a covering of triangles and operations with signs [Segura01].

Firstly, we will establish the working basis of the algorithms we propose, and then, we will present algorithms for the inclusion of a point in a polygon and the point-polygon collision detection. We will see how these methods have been implemented and a study of times that show whether they are effective or not will be carried out. Finally, possible extensions of algorithms and the work that is being carried out by the authors will be shown in the conclusion.

2. PREVIOUS DEFINITIONS

Definition 1 :Let x be a real number. We define the *sign function, sign (x)*, as follows :

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x>0 \\ 0 & \text{if } x=0 \\ -1 & \text{if } x<0 \end{cases}$$

Definition 2 : Let three points be A,B,C $\in R^n$, for n=2, the coordinates of which on the plane are A (x_A,y_A), B (x_B,y_B), C (x_C,y_C), the *signed area* of these three points is defined as :

$$|ABC| = \tfrac{1}{2} * \begin{vmatrix} x_A & x_B & x_C \\ y_A & y_B & y_C \\ 1 & 1 & 1 \end{vmatrix}$$

The signed area of these three points can be negative, depending on the points order. For counter-clockwise ordering can be easily demonstrated that the signed area is positive, and zero if the three points are aligned.

Definition 3 : A triangle T is *positive* if sign (|T|)=1, and *negative* if sign (|T|)= -1.

Theorem : Let points be A,B,C $\in R^2$, and let's suppose they are not collinear. Let another point be P $\in R^2$. Then, there are s,t,u $\in$ R, so that s + t + u = 1 and P = sA + tB + uC, that mean :

$$s = |BCP| / |ABC|$$
$$t = |CAP| / |ABC|$$
$$u = |ABP| / |ABC|$$

Numbers *s,t,u* defined in the above theorem are the barycentric coordinates of P with regard to points A, B and C.

Lemma : Let a point be P $\in R^2$ with barycentric coordinates *(s, t, u)* in relation to points A, B, C $\in R^n$, it is said that point P is inside the triangle defined by points A, B, C, if and only if [Badouel90] :

$$0 \leq s, t, u \leq 1$$

Corollary : Likewise, we define point P is outside the triangle A, B, C, if and only if :

$$s \leq 0 \vee t \leq 0 \vee u \leq 0$$

Geometric interpretation of barycentric coordinates of a point with regard to three vertices : If barycentric coordinates of a point P in relation to *S,T,U* are *s,t,u*, then a point with *sign(s)=+1* will be placed on the same side as S, as for the infinite line that goes through U and T. If *sign(s)=-1*, it will be on the opposite side ; if *sign(s)=0*, it will be on the line (Fig.1)

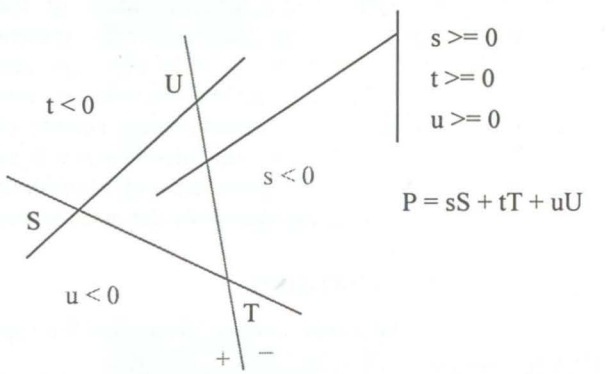


Fig. 1. Geometric interpretation of barycentric coordinates of a point with regard to a triangle

The algorithms displayed here use a covering of the objects by triangles with origin on the polygons barycenter. That is, different triangles appear with a common vertex (the polygon barycenter) and each of the n edges of the polygon. This covering uses a *O(n)* time and it is carried out as a pre-processing when constructing the figures. This generator system is valid for any kind of 2D polygon (and its extension to polyhedral solids in 3D), being either manifold or non-manifold, with or without holes, concave or convex.

3. POINT-POLYGON INCLUSION ALGORITHMS

A previous step before obtaining a collision detection algorithm consists in revising the point-polygon inclusion algorithms that represent the static collision detection (objects that do not move). This problem can be solved in different ways [Haine94]. The crossings tests [Lasszlo96], winding number [Preparata85], signed area [Hoffmann89] and [Feito95] algorithms, are some of the most representative, being the crossings test algorithm the fastest one.

We will develop algorithms based on a covering by triangles and barycentric coordinates. So, we will compare them with typical inclusion algorithms (crossings test and signed area ones [Feito95]) and with those of collision detection of section 4, in order to see how making use of coherence, algorithms are faster.

3.1. Inclusion Test of a Point on a Triangle

Once we have seen the above definitions, in order to determine whether a point is included in a triangle or not, we have only to check if barycentric coordinates of the point with regard to coordinates that form a triangle, are all in the interval [0,1], or what is the same, we have only to check if any of the barycentric coordinates is negative.

The calculations necessary to compute the barycentric coordinates of the point can be accelerated by calculating only two of the three barycentric coordinates, as s + t + u = 1, u = 1 - s- t. We can also accelerate the determinant calculation for a triangle area by the next formula which only implies 5 subtractions, 2 multiplications and 1 division :

$$|ABC| = [(x_B - x_A)(y_C - y_A) - (x_C - x_A)(y_B - y_A)] /2$$

We can obviate the division by 2 in numerator and denominator, as the area of two triangles is divided for all the barycentric coordinates :

$$s = |BCP| / |ABC| =$$
$$\det(BCP)/2 / \det(ABC)/2 = \det(BCP) / \det(ABC)$$

The point-triangle inclusion algorithm (Algorithm 1) is presented in the appendix. The receiver object is a triangle and it returns whether a point is inside or outside the triangle. Besides, if the point is inside, it returns either if it is on a vertex (giving information about it), or on an outside or inside edge, if it occurs.

The performance time of this algorithm is good enough compared with [Feito95] algorithm, as it can determine if a point is outside the triangle in near half the time used by the algorithm in [Feito95].

3.2. Inclusion Test of a Point on a Convex Polygon

In this case, the covering is formed by triangles that do not overlap (disjoint). In order to check the inclusion of a point on a polygon, the inclusion of the point on each triangle is verified sequentially. The point is considered to be inside the polygon if the inclusion test is positive in any of the triangles ; it is external to the polygon if it is outside all the triangles.

We can accelerate the test by considering only the barycentric coordinate *s* relative to the common vertex S, to all triangles of the covering. The point is inside the polygon if and only if the sign of all coordinates *s* with regard to the triangles of the covering is positive or zero;

it is outside if at least one of the signs of coordinates s is negative (Algorithm 2).

Once more, we can see a remarkable improvement as for algorithm in [Feito95], as when we find a negative coordinate s , we establish the point is outside and do not go on with the rest of covering triangles. In relation to the crossings test algorithm, the algorithm presented here, has a very similar behaviour, although with slightly higher times.

3.3. Inclusion Test of a Point on a Non-convex Polygon

In this case we determine the inclusion of the point on any of triangles of the covering according to [Feito95] algorithm, but then we use our inclusion test based on barycentric coordinates, as seen in the previous section (Algorithm 3).

4. ALGORITHMS OF POINT-POLYGON COLLISION DETECTION

Then we will see algorithms that determine whether there is collision between a point and a polygon or not. For that, we start a simulation on which the point moves round the scene and we try to determine whether the point enters the polygon or not. In order to accelerate the calculations, we make use of the temporal and geometric coherence.

4.1. Collision Detection Test between a Point and a Convex Polygon

If we deal with a point in motion and a convex polygon, we can determine whether there is collision or not at a certain time, making use of the information in the previous instants of time. Firstly, we determine whether the point is inside the polygon or not by the static test. For this, we have calculated the s barycentric coordinate of the point as for all triangles of the covering. If the point is outside the polygon, it will have one or more s negative coordinates. In fact, we are considering the zones that determine the sides of the polygon (and not the sides of the triangles of the covering that do not belong to the border) (Fig.2).

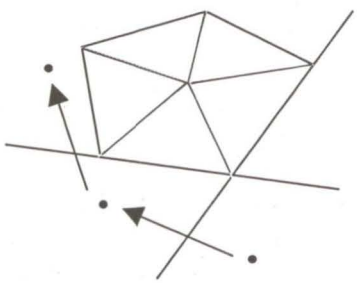


Fig. 2. Sign change of s barycentric coordinate

We choose one of the triangles with s negative coordinate. Whenever the point moves, we will only calculate the value of that coordinate with regard to that triangle. When the sign changes or it becomes zero, we will again calculate whether it is inside the polygon or not by the static convex point-polygon inclusion test; and we start again. We can make good use of temporal coherence if the first edges we consider in the static inclusion test are close to the studied edge. Then, we modify the algorithm of convex point-polygon inclusion test, so that it starts from the present edge and it returns the following edge with s negative barycentric coordinate. (See Algorithm 4).

4.2. Collision Detection Test between a Point and a Polygon with Positive Covering

In case we had a non-convex polygon, but with positive covering (all triangles of covering had a positive sign), we could not use this algorithm, but we could use the following one, which obviously is valid for convex polygons.

The algorithm basis is as follows: in case of having a positive covering, we can divide the space into zones, making use of the barycentric coordinates of a point. These zones determine whether a point is inside the corresponding triangle or not, and whether it is inside the complete polygon or not (Fig.3).

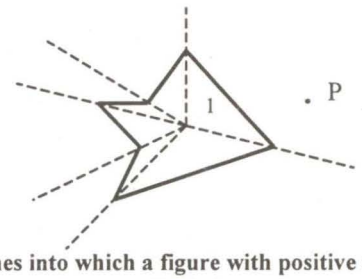


Fig. 3. Zones into which a figure with positive covering is divided

In figure 3, the point is on a zone where the signs of barycentric coordinates with regard to triangle 1 are (-1, +1, +1) that is, the point is outside the external side of the triangle and inside the inner sides of it.

By making use of temporal coherence, the point will move a little from one frame to another. So, the most probable is that the point stays in the same zone, and then it will continue outside; therefore, we will not have to calculate the barycentric coordinates in relation to the rest of triangles of the covering in order to determine if the point is outside. Let's imagine the sign of barycentric coordinates changes. If the sign of the first coordinate (s) is either positive or zero, it shows us the point is inside the triangle and the polygon. If it is one of the other two coordinates (t or u) which changes its sign, it shows us it has moved and changed its zone; we check it and consider the new zone from now (See Algorithm 6). The time used for the algorithm is $O(n)$ to determine the zone where the point is initially (See Algorithm 5) and $O(k)$ (k is a constant) on each movement, as a consequence of coherence.

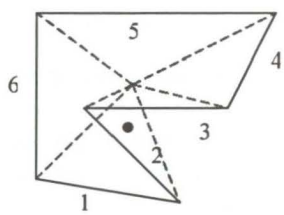
4.3. Collision Detection Test between a Point and a non Convex Polygon

We could use the static collision detection test on each movement of the point, but we would not make good use of coherence. In this case we propose an algorithm which does not need to calculate all the barycentric coordinates of a point with regard to all triangles of the covering and uses the coherence.

Let's suppose we are calculating if a point is inside one of the covering triangles. We calculate its coordinate *s* ; if it is positive, we go on with coordinate *t*. If *t* is also positive, we calculate coordinate *u* (since the point is outside the triangle if s<0 or t<0 or u>0)

As we have n triangles in the covering, we can extend it to all of them simultaneously, that is, firstly we calculate all the coordinates *s* with regard to all triangles, then we only calculate those coordinates *t* having *sign(t)=1*, and then coordinates *u* with *sign(t)=1*. So, we only calculate all coordinates *(s,t,u)* for those triangles where the point is, and multiply it by the triangle sign.

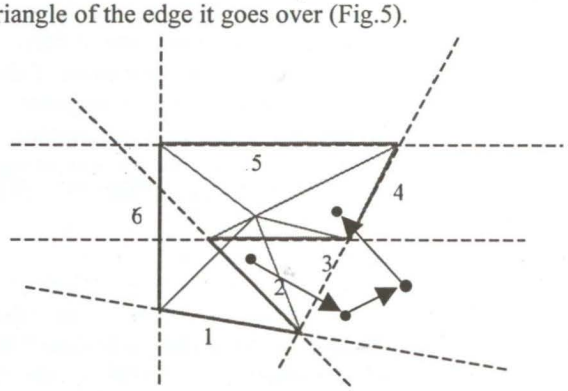
We can do it by a mask of bits. The bits, from left to right, represent each triangle of the covering. Firstly, all bits are 0, showing the point is not inside any triangle. We calculate the first coordinate *s* for all triangles and change to 1 the bit of the mask for those with *sign(s)>=0*. Then we calculate coordinate *t* only for those triangles with bit=1 in their mask. If *sign(t)=-1*, we change the bit to 0; if not, we do not change it. We do the same with coordinate *u*. In the end, we find a bit equal to 1 in the positions corresponding to the triangles where the point is (Fig.4). We only multiply by the sign of each triangle as it is made in the static detection test, in order to determine whether the point is inside the polygon or not.



Order	123456
Sign	+-++++
coord. s	110111
coord. t	11-000
coord. u	11----
Result:	inside 1,2
	outside Pol

Fig. 4. Masks of bits of a point

Up to now, we have not used coherence. We can see what happens if a point is outside the polygon and enters it. There is a variation in the sign of coordinate *s* as for the triangle of the edge it goes over (Fig.5).



Order	123456	123456	123456	123456
Sign	+-++++	+-++++	+-++++	+-++++
coord. s	110111	110011	110011	111111
coord. t	11-000	10--01	------	100100
coord. u	11----	1----0		0--100
state	outside	outside	equal	inside

Fig. 5. Calculation of masks of bits of a point in motion

So, we must keep these sign changes. If no change occurs, it is not necessary to calculate coordinates *t* and *u*. When it does, the algorithm will behave as it has been described in the previous paragraph. This implies that most of the time, movements occur in the same zone and we have only to calculate one of the three barycentric coordinates (See Algorithm 7).

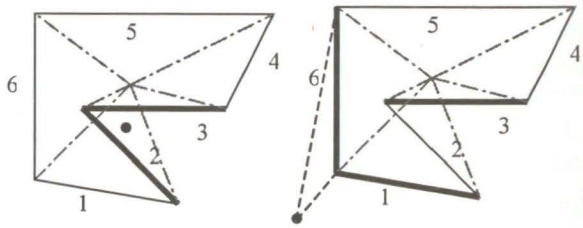
4.3.1. Algorithm improvements

With this algorithm we have to calculate one of the barycentric coordinates for all triangles of the covering whenever we move the point. We can accelerate the calculations if we succeed in reducing the number of triangles involved in the calculation of each movement. Our reasoning is as follows:

Firstly, whether the point is inside or outside the polygon is determined ; for that, the inclusion test is used. If the point is outside, edges that the point "sees" from its position are calculated, that is, the triangles edges of the covering, having the point a negative area in relation with them, are calculated.

$|PA_iA_{i\oplus1}| = BarycentricCoordS(P, A_iA_{i\oplus1}) * sign(T_i)$

This is kept in a mask of bits on which value 0 tells us that the edge "is not seen", and value 1 that the edge "is seen" (Fig.6).



Edges	123456	123456
Mask	011000	101001

Fig. 6. Mask of bits of edges a point can see

Then, the point moves and only the triangle area formed by the point with the edges the point saw a moment ago is checked (those which have 1 in the mask of bits). If a change occurs (that is, the point does not see an edge), we have to verify with the static inclusion test whether the point is inside the polygon or not, and then to calculate again the edges "seen" by the point from its new position (See Algorithm 8).

5. IMPLEMENTATION DETAILS

We have used an object-oriented approach in the implementation of classes which represent a Polygon. We have the *Polygon Class*, as a superclass, and *Convex-Polygon* and *non-Convex-Polygon classes*, which represent a convex and non-convex Polygon respectively. We have showed a *Positive-Polygon Class*, which is a particular case with a positive covering (e.g. star-shaped figures). This class has not been implemented, but has been included in this article in order to aid the algorithms comprehension. Actually, the *Polygon class*, detects the type of covering and then it works by choosing a method or another one.

We have initially carried out the covering of polygons, keeping the sign of triangles. For a higher efficiency, barycentric coordinates are calculated when it is only necessary and sometimes, previous calculations are used.

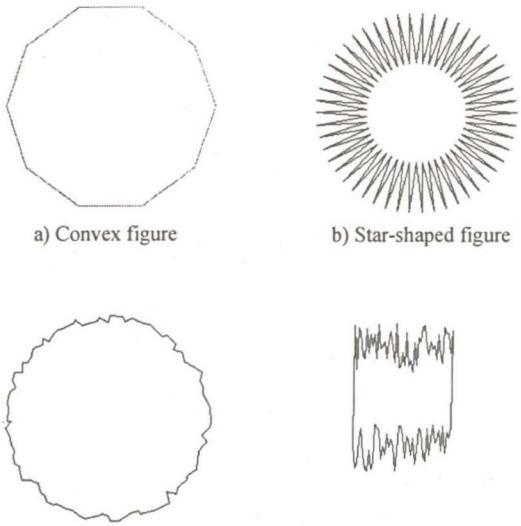
The calculation of sign function and involved determinants have been optimized.

It has been used the programming language C++ and the OpenGL graphic library on a Linux system.

6. TEMPORAL STUDY

We have compared the inclusion algorithms obtained in this work with the *crossings test* inclusion Algorithm [Haines94] and with *signed area* one [Feito95]. At the same time, we have compared these static algorithms with those of collision detection developed in order to show that these offer better times than their static versions, because of the use of coherence in movement.

Several tests have been made with different types of polygons. First, for convex polygons (Fig.7a); then, for non-convex polygons, and within them, star-shaped ones (with positive covering) (Fig.7.b) ; contour polygons (e.g. maps) (Fig.7c) ; and completely irregular and random polygons (Fig.7.d).



a) Convex figure b) Star-shaped figure

c) Contour Figure d) Non-convex figure (Irregular)

Fig.7. Different types of Polygons used in tests

For all types of polygons, the number of vertices has changed gradually from 3 to 1,000 ; it has not occurred with contour polygons (maps), on which vertices are between 10 and 1,000.

In all tests a point has been moved round a polygon, the nearest to its contour ; it has surrounded it completely (it has returned to its initial position). This movement has been formed by 90,000 positions in all cases.

The results obtained can be seen in the following figures, on which a logarithmic scale has been used, for both axes (x,y).

For convex polygons we have applied the crossings-test inclusion test, signed area one, barycentric coordinates and barycentric coordinates applied to positive covering; we have also applied collision detection tests for convex polygons, polygons with positive covering and non-convex polygons ones (Fig.8).

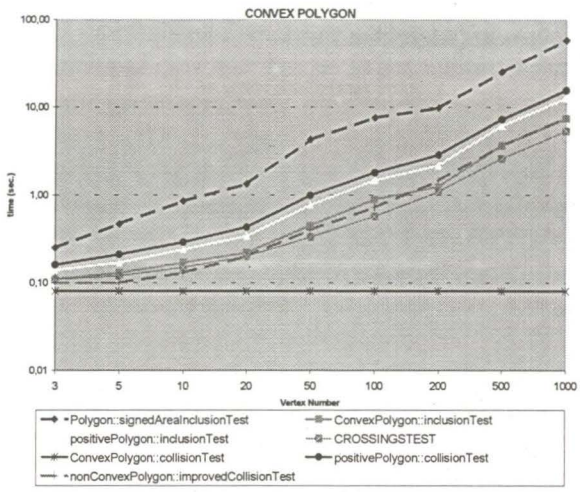


Fig. 8. Times graph for convex polygons

Undoubtedly, the collision detection algorithm for convex figures (Algorithm 4) is the fastest method, because after the first point calculation (pre-processing), it gets a lineal time. The optimized algorithm for non-convex figures (Algorithm 8) gets better times than the crossings test algorithm for figures with less than 20 vertices; for figures with more vertices, it approaches quite well, although with worse times.

Moreover, we can see that in general, algorithms for positive coverings (Algorithm 6) are slower than the others for this case, and signed area inclusion algorithm [Feito95] is the one that obtains the worse time.

For the case of non-convex polygons, we start seeing the results for star-shaped polygons (Fig.9).

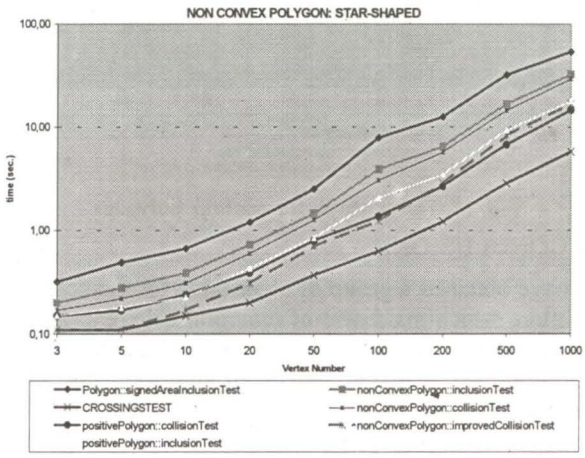


Fig. 9. Times graph for star-shaped polygons

Crossings test algorithm is the fastest one, followed by the collision detection optimized algorithm for non-convex figures (Algorithm 8), and figures with less than 150 vertices. For figures with more than 150 vertices, the collision detection algorithm for figures with positive covering (Algorithm 6), is faster than the last one.

For non-convex and completely irregular polygons (Fig.10), we can see that after the crossings test algorithm, the best one is the improved algorithm for non-convex polygons (Algorithm 8), followed by the

non-convex collision detection algorithm without any improvement (Algorithm 7).

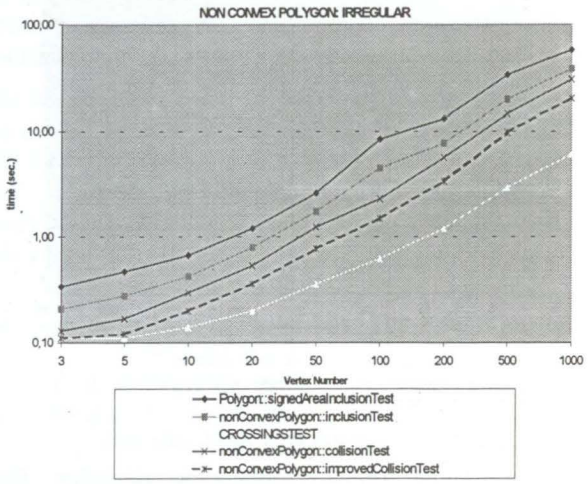


Fig. 10. Times graph for irregular polygons

In the case of contour non-convex polygons (Fig.11), we notice the improvement of the optimized collision detection algorithm (Algorithm 8) as for the algorithm without any improvement (Algorithm 7).

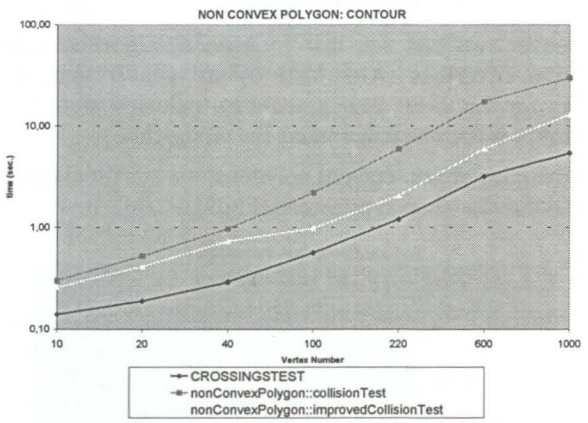


Fig. 11. Times graph for contour polygons

7. CONCLUSIONS

We have obtained a group of algorithms for the collision detection, which make use of temporal coherence. They have proved to be faster if they use this characteristic. The improved collision detection algorithm for non-convex polygons (Algorithm 8) is efficient in most situations, being only excelled by the crossings test algorithm in the case of non-convex polygons, and although times are higher, they are quite near.

Other algorithms which are efficient in certain situations have been developed : the collision detection algorithm for convex polygons (Algorithm 4) with a lineal time (after the first point calculation) excels all algorithms with a logarithmic behaviour. Another algorithm developed for polygons with positive covering (Algorithm 6) shows better for polygons with a high number of vertices.

The collision detection algorithm for convex polygons can be used in a bounding volumes hierarchy in a different detail level for non-convex polygons, so that we exclude pairs of objects in a simulation by using these bounding volumes. In this way, we can have a first level bounding volume, e.g. a rectangle, a second level one, the convex hull, a third one, a covering of positive triangles, and a forth one, the polygon itself.

In general, these algorithms, although they are not better in 2D, because of their possibility of extension to 3D together with their robustness, relative simplicity and their usefulness for any type of polygons, are suitable for several applications where time is very important.

Nowadays, we are working on the improvement of algorithms presented in this article, as for the masks of bits control (concerning their hardware implementation). We are also developing algorithms for the collision detection among polygons and their extension to 3D for complex polyhedral figures.

8. ACKNOWLEDGEMENTS

This work has been partially granted by the Ministry of Science and Technology of Spain and the European Union by means of the ERDF funds, under the research project TIC2001-2003-C03-03.

9. REFERENCES

[Badouel90] Badouel, F. An efficient Ray-Polygon intersection. Graphics Gems. Academic Press, 390-393, 1990

[Feito95] Feito, F; Torres, J.C.; Ureña, L.A. Orientation, Simplicity and Inclusion Test for Planar Polygons. Computer & Graphics, Vol. 19, N 4, 1995

[Haines94] Haines, E. Point in Polygon Strategies. Graphics Gems IV. Academic Press, 1994.

[Hoffmann89] Hoffmann, C. Geometric and Solid Modelling. An Introduction. Morgan Kaufmann Publishers, 1989.

[JimTho01] Jiménez, P.; Thomas, F.; Torras, C. 3D collision detection: a survey. Computer & Graphics 25 (2001) 269-285

[Jiménez02] Jiménez, J.J.; Segura, R.J.; Feito, F.R. Tutorial sobre Detección de Colisiones en Informática Gráfica. Accepted for its publication in Novatica, Spain, 2002

[Laszlo96] Laszlo, M. Computational Geometry and Computer Graphics in C++. Prentice Hall, 1996

[Preparata85] Preparata, F.; Shamos, M. Computational Geometry. An introduction. Springer-Verlag, 1985

[Segura01] Segura, R.J. Modelado de sólidos mediante recubrimientos simpliciales. Ph. D. Thesis. Depto. Lenguajes y Sistemas Informáticos. U.Granada, 2001

10. APPENDIX: ALGORITHMS

```
int triangle::inclusionTest(point p) {
        s0 = sign(BarycentricB0(p))
        if (s0<0) return OUT
        s1 = sign(BarycentricB1(p))
        if (s1<0) return OUT
        s2 = sign(BarycentricB2(p))
        if (s2<0) return OUT
        if (s0==0) {
                if (s1==0)
                        return VERTEX_V2;
                else    if (s2==0)
                                return VERTEX_V1;
                        else
                                return EDGE_EXTERNAL;
        } else {
                if (s1==0) {
                        if (s2==0)
                                return VERTEX_V0;
                        else
                                return EDGE_LEFT;
                } else
                        if (s2==0)
                                return EDGE_RIGHT;
        }
        return IN;
}
```

Algorithm 1. Point-Triangle inclusion test.

```
int convexPolygon::inclusionTest(point p) {
        exit = FALSE
        i = 0
        while (i < triangleNumber AND exit==FALSE) {
                s = Triangle[i]->sign(BarycentricB0(p))
                if (s<0) exit = TRUE
                i++
        }
        if (exit==TRUE) return OUT
        else return IN
}
```

Algorithm 2. Point-convex Polygon inclusion test.

```
int nonConvexPolygon::inclusionTest(point p) {
        exit = FALSE
        sum = 0
        i = 0
        while (i < triangleNumber AND exit==FALSE) {
                is_in = Triangle[i]->inclusionTest(p)
                if (is_in==EDGE_EXTERNAL OR is_in==VERTEX_V1 OR is_in==VERTEX_V2)
                        exit = TRUE
                else
                        if (is_in==IN)
                                sum += 2*Triangle[i]->sign()
                        else
                                if (is_in==EDGE_RIGHT || is_in==EDGE_LEFT)
                                        sum += Triangle[i]->sign()
                i++
        }
        if (exit==TRUE OR sum==2) return IN
        else return OUT
}
```

Algorithm 3. Point-non convex Polygon inclusion test.

```
//static inclusion test
is_in = inclusionTest(point, edge)
while (simulation goes on) {
        move point
        //collision detection test
        res = collisionTest(point, edge)
        print res
}

int convexPolygon::collisionTest(point p, int edge) {
        s = Triangle[edge]-> sign(BarycentricB0(p))
        if (s>=0) {
                is_in = inclusionTest(p, edge)
                // it returns the following zone related to an edge where the point is
                if (is_in == IN) return IN
        }
        return OUT
}
```

Algorithm 4. Point-convex Polygon collision detection test.

```
int positivePolygon::inclusionTest(point p, int edge) {
        edge = -1
        exit = FALSE
        i = 0
        while (i < triangleNumber AND exit == FALSE) {
                s0 = Triangle[i]->sign(BarycentricB0(p))
                if (s0<0)
                        s1 = Triangle[i]->sign(BarycentricB1(p))
                        s2 = Triangle[i]->sign(BarycentricB2(p))
                        if (s1>=0 AND s2>=0)
                                edge = i
                                exit = TRUE
                i++
        }
        if (edge==-1) return IN, edge
        else return OUT, edge
}
```

Algorithm 5. Point-Polygon with positive covering inclusion test.

```
//static inclusion test
is_in = inclusionTest(point,edge)
while (simulation goes on) {
        move point
        //collision detection test
        resTmp = collisionTest(point, edge)
        if (resTmp<>EQUAL_STATE)
                res = resTmp
        print res
}

int positivePolygon::collisionTest(point p, int edge) {
        s0 = Triangle[edge]->sign(BarycentricB0(p))
        s1 = Triangle[edge]->sign(BarycentricB1(p))
        s2 = Triangle[edge]->sign(BarycentricB2(p))
        if (s0<0 AND s1>=0 AND s2>=0)
                return EQUAL_STATE, edge
        if (s0>=0 AND s1>=0 AND s2>=0)
                return IN, edge
        i=0;
        while (i<triangleNumber-1) {
                if (s1<0) index = (edge+i+1) % triangleNumber
                if (s2<0) index = (edge-i-1) % triangleNumber
                s0' = Triangle[index]->sign(BarycentricB0(p))
                s1' = Triangle[index]->sign(BarycentricB1(p))
                s2' = Triangle[index]->sign(BarycentricB2(p))
                if (s0'<0 AND s1'>=0 AND s2'>=0)
                        return OUT, index
                if (s0'>=0 AND s1'>=0 AND s2'>=0)
                        return IN, index
                i++
        }
}
```

Algorithm 6. Point-Polygon with positive covering collision detection test.

```
is_in = inclusionTest(point) //static inclusion test
while (simulation goes on) {
        move point
        resTmp = collisionTest(point, mask) //collision detection test
        if (resTmp<>EQUAL_STATE)
                res = resTmp
        print res
}
int nonConvexPolygon::collisionTest(point *p, int *mask) {
        maskB0 = maskB1 = maskB2 = 0
        sum = 0 ; i = 0
        while (i<triangleNumber) {
                s = Triangle[i]->sign(BarycentricB0(p))
                if (s>=0) maskB0[i] = 1
                i++
        }
        if (maskB0 == mask) return EQUAL_STATE
        mask = maskB0
        i=0
        while (i<triangleNumber) {
                if (maskB0[i] > 0)
                        s = Triangle[i]->sign(BarycentricB1(p))
                        if (s==0) sum+=Triangle[i]->sign()
                        if (s>=0) maskB1[i] = 1
                i++
        }
        i=0
        while (i<triangleNumber) {
                if (maskB1[i] > 0)
                        s = Triangle[i]->sign(BarycentricB2(p))
                        if (s==0) sum+=Triangle[i]->sign()
                        if (s>=0) maskB2[i] = 1
                i++
        }
        maskPos = maskB2 & maskSign // mask of bits of triangles covering signs
        maskNeg = maskB2 & (~maskSign)
        i=0
        while (i<triangleNumber) {
                if (maskPos[i] > 0) sum += 2
                if (maskNeg[i] > 0) sum -= 2
                i++
        }
        if (sum==2) return IN
        else return OUT
}
```

Algorithm 7. Point-non convex Polygon collision detection test.

```
is_in = inclusionTest(point) //static inclusion test
while (simulation goes on) {
        move point
        resTmp = improvedCollisionTest(point, mask) //collision detection test
        if (resTmp<>EQUAL_STATE)
                res = resTmp
        print res
}
int nonConvexPolygon::improvedCollisionTest(point p, int *mask) {
        exit = FALSE ; i = 0
        while (i < triangleNumber AND exit == FALSE) {
                if (mask[i] == 0)
                        s = Triangle[i]->sign(TriangleB0(p))
                        if (s>=0)
                                exit = TRUE
                i++
        }
        if (exit==FALSE) return EQUAL_STATE
        else
                if (inclusionTest(p)==IN) {  //static inclusion test
                        mask = 0
                        return IN, mask
                } else {
                        collisionTestGetMask(p,mask) //bit-mask obtaining for collision test
                        return OUT, mask
                }
}
```

Algorithm 8. Point-non convex Polygon improved collision detection test.