RUHUNA JOURNAL OF SCIENCE
Vol 8: 67- 75, June 2017
eISSN: 2536-8400
DOI: http://doi.org/10.4038/rjs.v8i1.27

Short paper

A simple algorithm for calculating the area of an arbitrary polygon

K.R. Wijeweera[1, 2] and S.R. Kodituwakku[2, 3]

[1]*Department of Computer Science, Faculty of Science, University of Ruhuna, Sri Lanka*
[2]*Postgraduate Institute of Science, University of Peradeniya, Sri Lanka.*
[3]*Department of Statistics and Computer Science, Faculty of Science University of Peradeniya*

Correpondence: krw19870829@gmail.com

Received: October 28th 2016, Revised: June 15th 2017, Accepted: June 30th 2017

Abstract. Computing the area of an arbitrary polygon is a popular problem in pure mathematics. The two methods used are Shoelace Method (SM) and Orthogonal Trapezoids Method (OTM). In OTM, the polygon is partitioned into trapezoids by drawing either horizontal or vertical lines through its vertices. The area of each trapezoid is computed and the resultant areas are added up. In SM, a formula which is a generalization of Green's Theorem for the discrete case is used. The most of the available systems is based on SM. Since an algorithm for OTM is not available in literature, this paper proposes an algorithm for OTM along with efficient implementation. Conversion of a pure mathematical method into an efficient computer program is not straightforward. In order to reduce the run time, minimal computation needs to be achieved. Handling of indeterminate forms and special cases separately can support this. On the other hand, precision error should also be avoided. Salient feature of the proposed algorithm is that it successfully handles these situations achieving minimum run time. Experimental results of the proposed method are compared against that of the existing algorithm. However, the proposed algorithm suggests a way to partition a polygon into orthogonal trapezoids which is not an easy task. Additionally, the proposed algorithm uses only basic mathematical concepts while the Green's theorem uses complicated mathematical concepts. The proposed algorithm can be used when the simplicity is important than the speed.

Keywords. Computational geometry, computer graphics programming, coordinate geometry, Euclidian geometry, computer programming.

1 Introduction

A polygon is defined as the region of a plane bounded by a finite collection of line segments which forms a simple closed curve. Let v_0, v_1, v_2, …, v_{n-1} be n points in the plane. Here and throughout the paper, all index arithmetic will be mod n, conveying a cyclic ordering of the points, with v_0 following v_{n-1}, since $(n - 1) + 1 \equiv 0 \pmod n$. Let $e_0 = v_0v_1$, $e_1 = v_1v_2$, …, $e_i = v_iv_{i+1}$, …, $e_{n-1} = v_{n-1}v_0$ be n segments connecting the points. Then these segments bound a polygon if and only if

1) The intersection of each pair of segments adjacent in the cyclic ordering is the single point shared between them: $e_i \cap e_{i+1} = v_{i+1}$, for all $i = 0, 1, 2, ..., n - 1$.
2) Non adjacent segments do not intersect: $e_i \cap e_j = \emptyset$, for all $j \neq i + 1$.
3) None of three consecutive points v_i are collinear.

These segments define a curve since they are connected end to end and the curve is closed since they form a cycle and the curve is simple since non adjacent segments do not intersect. The points v_i are called the vertices of the polygon while the segments e_i are called its edges. Therefore a polygon of n vertices has n edges (O'Rourke 1998).

In the proposed algorithm, the polygon is separated into orthogonal trapezoids by drawing horizontal lines through each vertex of the polygon. The area of each trapezoid is computed and added up to find the area of the entire polygon.

2 Methodology

This section describes the proposed algorithm and its implementation. The C programming language has been used for the implementation.

2.1 Representation of the polygon

The polygon is represented using two arrays *x* and *y*. The *points* variable stores the number of vertices in the polygon. ($x[i]$, $y[i]$) denotes the coordinates of the i^{th} vertex where $i = 0, 1, \ldots, (points - 1)$.

2.2 Drawing horizontal lines through each vertex of the polygon

The polygon is separated into orthogonal trapezoids by drawing horizontal lines through each vertex of the polygon. This is done by using *findYg* function. First each y-coordinate is stored in *yg* array. Then *sortArray* function is used to sort those y-coordinates in ascending order.

2.3 The *refineYg* function

Each horizontal line drawn in section 2.2 may go through more than one vertex in the polygon. In those cases *yg* array contains duplicate values. The *refineYg* function is used to remove those duplicate values from the *yg* array. Using a *for* loop, initially the *yg* array is copied to *yh* array. After that only the distinct values are written to the *yg* array. The *ygn* variable will finally contain the number of distinct elements. Since the *yg* array is already sorted in ascending order, duplicate values are always in consecutive cells. The first element of *yg* array is set into first element of *yh* array by *yg*[*0*] = *yh*[*0*] and now *ygn* is 1. Using another *for* loop, values of *yh* are copied one by one to *yg* only if current value is not equal to the previous. The condition (*yh*[*i* - 1] != *yh*[*i*]) is used for this purpose.

2.4 The *pos* function

The *pos* function is used to decide the position of a given vertex (*x*[*v*], *y*[*v*]) with respect to a horizontal line *y* = *yg*[*u*]. If the vertex is on the horizontal line then the function returns zero. If the vertex is above the horizontal line then it returns 1 and if the vertex is below the horizontal line it returns -1.

2.5 The *findGaps* function

The *findGaps* function takes *u* as a parameter which denotes the indices of *yg* array. A horizontal line drawn in section 2.2 may intersect edges of the polygon as shown in Figure 1.

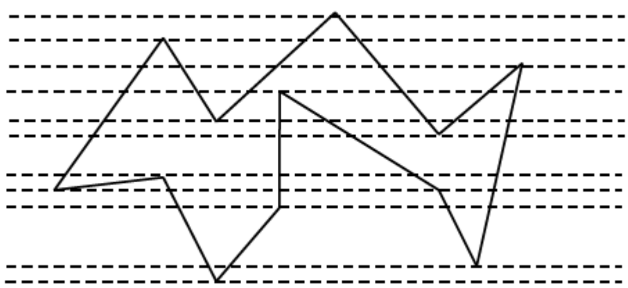

Fig. 1. Horizontal lines through vertices

Two consecutive horizontal lines bound a set of orthogonal trapezoids. In order to compute the area of them, coordinates of the end points should be calculated. For a given two consecutive horizontal lines, the intersection points of upper and lower horizontal lines differ depending on the special cases as described in this section later. Therefore x-coordinates of the intersection points corresponding to lower horizontal line are stored in *gap1* array and that of upper horizontal line are stored in *gap2* array each iteration. The *n1* and *n2* variables will contain the number of elements in *gap1* and *gap2* arrays respectively.

The purpose of first *for* loop found in *findGaps* function is to deal with intersections of each edge with the given horizontal line $y = yg[u]$. The end points of each edge are $(x[i], y[i])$ and $(x[j], y[j])$ where $i = 0, 1\ldots, (n - 1)$ and $j = (i + 1)$ % *points*. The *pi* and *pj* variables decide the positions of the end points using *pos* function.

Depending on the way edges intersect with the horizontal line, there are five possible situations as shown in Figure 2.

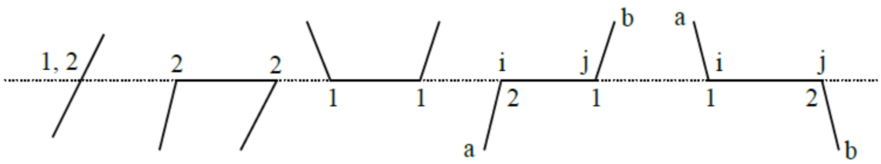

Fig. 2. Possible intersections of edges

From the left each possible situation is names as General Case, Down to Down Case, Up to Up Case, Down to Up Case, and Up to Down Case respectively. In the figure, 1 and 2 numbers denote whether that intersection point considered for *gap1* array or *gap2* array respectively. Following subsections describe how to deal with each case.

General Case

If ($pi * pj < 0$) then end points are on opposite sides of the horizontal line. The intersection point is $(xc, yg[u])$. And the equation of the edge can be written as,

$$(y - y[i]) / (x - x[i]) = (y[j] - y[i]) / (x[j] - x[i]);$$
$$x = (x[j] - x[i]) * (y - y[i]) / (y[j] - y[i]) + x[i];$$

Since the intersection point is on $y = yg[u]$ line,

$$xc = (x[j] - x[i]) * (yg[u] - y[i]) / (y[j] - y[i]) + x[i];$$

This intersection point should be included to both *gap1* and *gap2* arrays.

Down to Down Case

If ($pi = pj = 0$) then the entire edge goes through the horizontal line as in second to fifth situations in Figure 2. The values of a and b are computed by $a = i - 1$ and $b = j + 1$. They denote the indices of neighboring vertices of the end points of the edge. When $i = 0$ then $a = -1$, but actually it should be ($points$ - 1). When $j = (points - 1)$ then $b = points$, but actually it should be 0. These two special cases should be handled.

The pa and pb variables store the positions of a and b vertices respectively. If $pa * pb > 0$ then it should be either second or third case in Figure 2. If $pa < 0$, it should be definitely the second situation. In this case both the end points should be included to *gap2* array.

Up to Up Case

If $pa * pb > 0$ and $pa > 0$, it is the third situation. In this case both the end points should be included to *gap1* array.

Down to Up Case

If $pa * pb < 0$ and $pa < 0$ then it is the fourth situation. In this case i end point should be included to *gap2* array and j end point should be included to *gap1* array.

Up to Down Case

If $pa * pb < 0$ and $pa > 0$ then it is the fifth situation. In this case i end point should be included to *gap1* array and j end point should be included to *gap2* array.

Depending on the way vertices intersect with the horizontal line, there are three possible ways as shown in Figure 3.

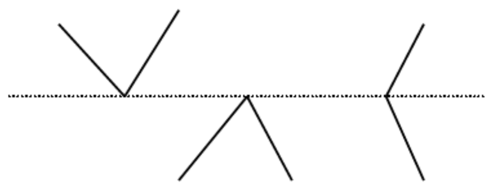

Fig. 3. Possible intersections of vertices

The first two situations are ignored since they do not affect the area between given two horizontal lines. The second *for* loop is used to deal with vertex intersections with the horizontal line. The vertices on the horizontal line are found by checking the condition $y[i] = yg[u]$. The *pa* and *pb* variables are found as in previous situation. If $(pa * pb < 0)$, it is the third situation. In this situation the intersection vertex should be included to both *gap1* and *gap2* arrays.

2.6 The *findSums* function

There are *ygn* number of horizontal lines which can be drawn through vertices of the polygon as shown in Figure 1. Each horizontal line has two arrays *gap1* and *gap2*. Inside the *for* loop, initially *n1* and *n2* are set into zero. Then *gap1* and *gap2* array values are found using *findGaps* function for each horizontal line. The *sortArray* function is used to sort those two arrays in ascending order.

2.7 The *computeSum* function

Figure 4 shows an example diagram of a horizontal line. The numbers indicate the indices of the *gap* array in *computeSum* function. The thick line segments on the horizontal line shows the intersection regions with the polygon.

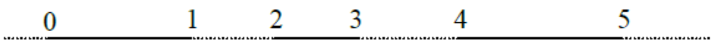

Fig. 4. Intersection regions example

The purpose of *computeSum* function is to calculate the sum of the lengths of the thick line segments and store it in $sum[index]$ element of *sum* array. The value of $sum[index]$ is set into zero initially. Then the elements of *gap* array are added or subtracted from the $sum[index]$ depending on whether their indices are even or odd respectively.

In each iteration of the *for* loop inside *findSums* function, the *computeSum* function is invoked for *gap1* and *gap2* arrays. The *sum1* and *sum2* arrays correspond to *gap1* and *gap2* arrays respectively.

2.8 The *getAreaS* function

Figure 5 shows an example of polygon parts bounded by two consecutive horizontal lines drawn through vertices of polygon. The $sum1[i]$ stores the sum of lengths of thick line segments corresponds to the lower horizontal line. And $sum2[i + 1]$ stores the sum of lengths of thick line segments corresponds to the upper horizontal line.

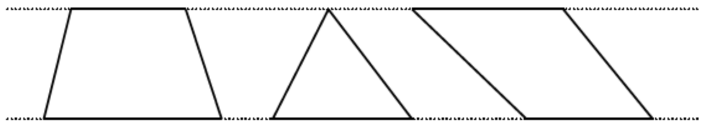

Fig. 5. Polygon parts between two horizontal lines

The *area* variable in *getAreaS* function stores the area bounded by i^{th} couple of consecutive horizontal lines. The gap between the two horizontal lines is ($yg[i + 1] - yg[i]$). Therefore the area bounded by the two horizontal lines can be computed by 0.5 * ($yg[i + 1] - yg[i]$) * ($sum1[i] + sum2[i + 1]$). This formula is similar to the area formula for a trapezoid and it can be proved as follows.

Proof: Let the gap between two horizontal lines is *h* and *N* be the number of trapezoids. Then the height, lower base and upper base of the trapezoids are (h, a_1, b_1), (h, a_2, b_2)…, and (h, a_N, b_N) respectively. Sum of areas of trapezoids,
$= (h/2) * (a_1 + b_1) + (h/2) * (a_2 + b_2) + \ldots + (h/2) * (a_N + b_N)$
$= (h/2) * [(a_1 + a_2 + \ldots + a_N) + (b_1 + b_2 + \ldots + b_N)]$

2.9 The *getArea* function

The *area* variable inside *getArea* function will store the total area of the polygon. The *for* loop accesses each area bounded between consecutive couple of horizontal lines and sum them up.

3 Results and Discussion

The algorithm was implemented using C programming language. Following hardware and software were used.

Computer: Intel(R) Celeron(R) M; processor 1.50 GHz, 896 MB of RAM;
IDE: Turbo C++; Version 3.0; Copyright(c) 1990, 1992 by Borland International, Inc;
System: Microsoft Windows XP Professional; Version 2002; Service Pack 2.

To validate the proposed algorithm, it was compared with an implementation of Shoelace Method (Wijeweera 2015, O'Rourke 1998). A set of random polygons as shown in Table 1 were used.

Table 1. Set of random polygons

Polygon	Coordinates of Vertices
1	(10, 10), (110, 210), (90, 80), (250, 60)
2	(60, 240), (10, 190), (50, 20), (110, 150), (150, 190)
3	(140, 90), (200, 240), (40, 190), (70, 10), (280, 80), (210, 130)
4	(60, 60), (300, 130), (240, 220), (100, 250), (20, 200), (40, 20), (260, 70)
5	(100, 240), (130, 80), (270, 210), (190, 20), (150, 60), (90, 10), (10, 80), (80, 100)
6	(150, 40), (50, 10), (10, 190), (120, 250), (230, 220), (130, 90), (240, 160), (240, 30), (50, 120)
7	(140, 90), (240, 60), (140, 240), (250, 170), (290, 10), (30, 10), (10, 120), (130, 220), (80, 70), (170, 140)
8	(230, 40), (30, 10), (80, 70), (10, 90), (70, 90), (150, 140), (150, 60), (190, 140), (80, 250), (210, 200), (280, 10)
9	(90, 90), (150, 160), (130, 220), (240, 70), (250, 10), (160, 110), (110, 30), (40, 10), (20, 130), (90, 250), (130, 160), (90, 190)
10	(90, 70), (60, 150), (10, 20), (60, 190), (90, 150), (140, 240), (190, 130), (230, 170), (270, 60), (190, 10), (230, 100), (140, 30), (160, 150)

The number of clock cycles to compute the area of a polygon is not measurable since the value is too small. Therefore the number of clock cycles to compute the area of the same polygon 10^8 times was measured (Kodituwakku *et al*. 2013). This was done for each polygon in Table 1 using Orthogonal Trapezoid Method (OTM) and Shoelace Method (SM). The results are shown in Table 2.

Table 2. Set of random polygons

Polygon	OTM	SLM
1	4554	204
2	5345	241
3	9914	291
4	11641	327
5	14408	361
6	20791	406
7	28416	453
8	21939	466
9	33098	490
10	41047	532

According to the results the efficiency of the proposed method is lower than the existing method.

4 Conclusion

An algorithm to computerize Orthogonal Trapezoid Method was proposed. And it was experimentally compared against Shoelace Method. The area of the polygon was computed by decomposing the polygon into a set of trapezoids. The decomposition was not a trivial task. Currently triangular polygonal meshes are used in computer graphics programming to model surfaces (Hearn and Baker 1998). The proposed decomposition technique can be to generate trapezoidal polygonal meshes.

References

Hearn D, Baker MP. 1998. *Computer Graphics, C Version*, 2nd Edition, Prentice Hall, Inc., Upper Saddle River, pp. 305-309.

Kodituwakku SR, Wijeweera KR, Chamikara MAP. 2013. An Efficient Algorithm for Line Clipping in Computer Graphics Programming, *Ceylon Journal of Science (Physical Sciences)*, Volume 17: 1-7.

O'Rourke J. 1998. *Computational Geometry in C: 2nd Edition*, Cambridge University Press 1-22.

Wijeweera KR. 2015. Finding the Area of an Arbitrary Polygon: Shoelace Formula and its Implementation in C Programming Language. Retrieved from http://www.academia.edu/9987996/Finding_the_Area_of_an_Arbitrary_Polygon_Shoelace_Formula_and_its_Implementation_in_C_Programming_Language.

Appendix

Program code is supplied as a supplementary file (Appendix, pp i-iv) linked to Wijeweera and Kodituwakku (2017). *Ruhuna Journal of Science* 8 (1): 67-75. DOI: http://doi.org/10.4038/ rjs.v8i1.27