Ceylon Journal of Science 45(3) 2016: 65-76
DOI: http://doi.org/10.4038/cjs.v45i3.7402

RESEARCH ARTICLE

Accurate, Simple and Efficient Triangulation of a Polygon by Ear Removal with Lowest Memory Consumption

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

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

Received: 22/10/2016; Accepted: 10/11/2016

Abstract: Polygons can conveniently represent real world objects. In automatic character recognition, shapes of individual letters are represented by polygons. In robotics, obstacles are represented using polygons. In computer graphics programming, solid objects are represented using polygons on the two dimensional screen. The polygons can be easily manipulated using known mathematical operations. That is the reason for representing real world objects using polygons. However, polygons can be in complicated shapes. Therefore, it is better if there is a way to partition a polygon into smaller pieces. Triangulation is a particular way of doing this from which polygons are partitioned into triangles. The basic triangulation algorithm is widely used in applications where 100% accuracy is necessary. Algorithms with better asymptotic order than the basic triangulation algorithm exist. However they are not 100% accurate and use advanced data structures causing higher memory consumption. This paper proposes a simple, efficient and 100% accurate algorithm which uses lowest amount of memory. The proposed algorithm is more suitable for embedded systems which do not possess large amount of memory. The proposed algorithm was experimentally compared with the basic triangulation algorithm. The experimental results prove that the proposed algorithm is faster than the basic triangulation algorithm.

Keywords: Computational Geometry, Computer Graphics Programming, Triangulation, Computational Statistics, Linear Programming

INTRODUCTION

Polygons can be used to represent the real world objects conveniently. Once the real world objects were represented using polygons, the resultant polygons are too complicated most of the time. In order to manipulate the polygons faster and easier in applications, the polygons need to be partitioned into simpler components such as triangles (Chazelle, 1982; Chazelle, 1991), trapezoids (Seidel, 1991) or even sub-polygons (Feng *et al.*, 1975).

Polygon triangulation can be defined as the decomposition of a polygonal area into a set of triangles (Berg *et al.*, 2008; Garey *et al.*, 1975). In other words, polygon triangulation can be defined as the creation of a set of triangles without non-intersecting interiors whose union is the original polygon.

According to the Meister's theorem (Meister, 1975), any simple polygon with at least four vertices without holes has at least two 'ears', which are triangles with two sides being the edges of the polygon and the third one completely inside it. Meister proposed a recursive algorithm which search for an ear and cut it off from current polygon. A new polygon is formed after the removal of an ear. That polygon also still meets the 'two ears' condition and repetitions can be done until there is only one triangle left.

The direct implementation of ear clipping method runs in O (n^3), with O (n) time taken on testing whether a triangle newly created is valid. But later an efficient method named 'prune-and-search' brought the time complexity from O (n^3) down to O (n^2) (Elgindy *et al.*, 1993). Further a simple reorganization of Meister's algorithm leads to run ear clipping algorithm in time complexity of O (n^2) (O'Rourke, 1998).

Kong, Everett, and Toussaint algorithm (Kong *et al.*, 1990) also utilizes ear clipping adopting the Graham scan method to select ears. This algorithm is sensitive to the shape of the

**Corresponding Author's Email*: krw19870829@gmail.com

polygon and runs in O (n(r+1)) time, where r is the number of reflex vertices of the polygon.

A triangle with a sharp angle is called a silver triangle. The silver triangles are not suitable for applications due to their poor shape. Sloan (cis.uab.edu, 2016) developed an ear clipping algorithm with O (n^2) time complexity based on O'Rourke's algorithm. An optimization process by swapping diagonals is done to remove silver triangles after performing the triangulation. Held (Held, 2001) developed a package for called FIST for polygon triangulation. It was based on ear clipping and can be applied to handle complex polygons.

Gang *et al* (Mei *et al.*, 2012) proposed a new algorithm based on ear clipping to generate high quality triangulations with fewer silver triangles. When locating an ear tip, the one with smallest interior angle is always selected and removed. If the ear tip with the smallest angle is not chosen at first, it will be divided into at least two smaller ones. Further silver triangles are reduced by edge swapping. But edge swapping is adopted during cutting ears rather than after performing the entire triangulation as in Sloan's algorithm (cis.uab.edu, 2016).

David (Eberly, 2016) proposed an algorithm of O (n^2) time complexity. It uses tree data structure for representation. According to David, even though the algorithms exist with better asymptotic order than his algorithm, they are more difficult to be implemented. An algorithm with O (n log n) time complexity uses horizontal decomposition into trapezoids followed by identification of monotone polygons that are themselves triangulated (Chazelle *et al.*, 1984; Fournier *et al.*, 1984). By improving further an incremental randomized algorithm produces an O (n log* n) where log* n is the iterated logarithm function (Seidel, 1991). This function is effectively a constant for very large values of n that can be observed in practice. Therefore the randomized method can be considered as having O (n) time complexity for practical purposes. An algorithm with O (n) time complexity exists (Chazelle, 1991). However it is extremely complex and no implementation has been proposed yet.

It has been mathematically proved that every polygon can be triangulated (O'Rourke, 1998). Basically triangulation algorithms can be divided into two categories: diagonal insertion and ear clipping. The output of diagonal insertion methods is a list of diagonals which separate the polygon into a set of triangles. The output of the ear clipping methods is a list of triangles in which the union forms the original polygon. Therefore ear subtraction methods are more useful when the application needs to find the set of triangles along with the corresponding coordinates of the vertices of those triangles. The proposed method is inspired by ear subtraction method.

A polygon can be defined as the region of a plane bounded by a finite collection of line segments forming 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 is mod n, conveying a cyclic ordering of the points, with v_0 following v_{n-1}, since $(n - 1) + 1 \equiv n \equiv 0$ (mod n). Let $e_0 = v_0v_1$, $e_1 = v_1v_2\ldots$, $e_i = v_iv_{i+1}\ldots$, $e_{n-1} = v_{n-1}v_0$ be n segments connecting the points. Then a polygon is bounded by these segments 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…, 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 vertices are collinear.

These line segments define a curve due to the fact that they are connected end to end. The curve is said to be closed since they form a cycle. Also this closed curve is simple since non adjacent segments do not intersect. The points v_i are called vertices of the polygon while the segments e_i are called its edges. According to the definition a polygon is a closed region of a plane. If P denotes a polygon then ∂P is used to denote the boundary of the polygon. A polygon divides the plane into two mutually exclusive regions as interior and exterior. The interior region is bounded while the exterior region is unbounded.

A line segment along with its end points is called a closed line segment. Any two vertices of a polygon P can be joined using a line segment. Let A and B are the end points of such kind of a line segment. If the intersection of the closed segment AB with ∂P is exactly the set {A, B} and the closed segment AB is nowhere exterior to the polygon then the line segment is called a diagonal of the polygon. Suppose P, Q and R be the three consecutive vertices of a polygon. If PR is a diagonal then PQR is called an ear of the polygon (O’Rourke, 1998).

The basic triangulation algorithm (O’Rourke, 1998) is a 100% accurate algorithm for polygon triangulation available in literature. It is a diagonal insertion method and based on the following theorem called triangulation theorem.

Triangulation Theorem: Every polygon P of n vertices may be partitioned into triangles by the addition of (zero or more) diagonals.

The method suggested by this theorem is an O (n^4) algorithm. There are ${}^nC_2$ = O (n^2) diagonal candidates. The cost of testing each for diagonal hood costs O (n). Since this O (n^3) should be repeated for each of the (n – 3) diagonals yields O (n^4).

The proposed algorithm stores the coordinates of the vertices of the polygon in two arrays. It detects the ears of the polygon and prunes them one by one. Thus the polygon is triangulated. The output of the proposed algorithm is a list of disjoint triangles in which the union forms the original polygon. This algorithm is 100% accurate algorithm and it avoids precision error. It utilizes static memory only. The proposed algorithm is faster than basic triangulation algorithm.

METHODOLOGY

In this section the proposed approach is discussed in detail. The implementation of the algorithm is given in the appendix in order to make it easy to understand the methodology.

Representation of the polygon

The polygon is stored in arrays. The *points* variable stores the number of vertices in the polygon. To store vertices of the polygon, two arrays are used. Then (x[i], y[i]) denotes the ith vertex of the polygon where i = 0, 1…, (*points* - 1).

Some issues in implementation

The coordinates of the vertices may be floating point numbers. Since the *line* function takes integers as its inputs, those coordinates should be approximated into nearest integer. That is done by *dpx* and *dpy* functions. Adding 0.5 to a floating point number and truncating the decimal part will approximate it to the nearest integer. The origin of the coordinate system in the used software is situated in the top left corner of the screen. But in conventional coordinate systems origin should be in bottom left corner of the screen. Therefore, inversion of the y-coordinate has been done inside *dpy* function where the inbuilt *getmaxy* function returns the maximum y-coordinate of the display screen. The *drawPolygon* function has been used to draw the polygon on the screen. Modulo operator (%) has been used to draw the polygon handling the cyclic nature of the polygon.

Mathematical properties

The proposed algorithm uses some of the known theorems in the literature. They have been already proved (O’Rourke, 1998) and here only the results are shown.

Theorem 1: Every polygon of $n \geq 4$ vertices has at least two non-overlapping ears.
Theorem 2: Every triangulation of a polygon P of n vertices uses n – 3 diagonals and consists of n – 2 triangles.

Consider a triangle with vertices $P_1(x_1, y_1)$, $P_2(x_2, y_2)$ and $P_3(x_3, y_3)$. And an expression can be written as $t = x_1 * (y_2 – y_3) + x_2 * (y_3 – y_1) + x_3 * (y_1 – y_2)$. If P_1, P_2 and P_3 are situated in clockwise order then $t < 0$ holds. Otherwise $t > 0$ holds (Green).

Identifying the convex vertices of the polygon

The first phase of the proposed algorithm is to identify the convex vertices of the polygon. None of the three consecutive vertices of the input polygon should be collinear by the definition of a polygon. Therefore, all the convex vertices of the input polygon are strictly convex. That means the interior angle formed by each vertex is less than π. It can be easily deduced that the lowest vertex of such a polygon is always strictly convex as shown in Figure 1.

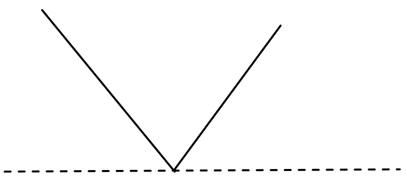

Figure 1: Lowest vertex of a polygon

A horizontal line has been drawn in Figure 1 through the lowest vertex of the polygon. Then the entire polygon is upward with respect to that horizontal line. Since the angle formed by a straight line is π, it is obvious that the lowest vertex should always be strictly convex. When there are multiple lowest vertices, each of them is also strictly convex applying the same reasoning. First the algorithm should find one of those lowest vertices. This task is done by the *lv* function. And that function returns the index of one of those lowest vertices.

Beginning from 0th vertex, the vertices of the polygon can be traversed. The traversal will be clockwise or anticlockwise depending on how the vertices are located in two dimensions. Suppose vertices are in anticlockwise order as shown in Figure 2. Suppose this is the original polygon provided to the algorithm as the input. The *lv* function will return the index of the lowest vertex as 0.

If the index of a vertex is v then three consecutive vertices (v – 1, v, v + 1) can be considered. When v = 0, v – 1 = -1, it should be corrected as v – 1 = (*points* - 1). When v = (*points* - 1), v + 1 = *points*, it should be corrected as v + 1 = 0. The *ts* function decides the sign of t value of three consecutive vertices when middle vertex is provided. The *isConvex* function has been designed to return 1, if the passed index to this function corresponds to a convex vertex. If the multiplication of *ts(v)* and *ts(lv())* is positive, then it can be decided that the vertex v and lowest vertex has same sign for their t expression. It has been mentioned earlier that lowest vertex should always be convex. Therefore, the vertex v should also be convex since they have the same sign. In this way, convex vertices of the polygon can be identified.

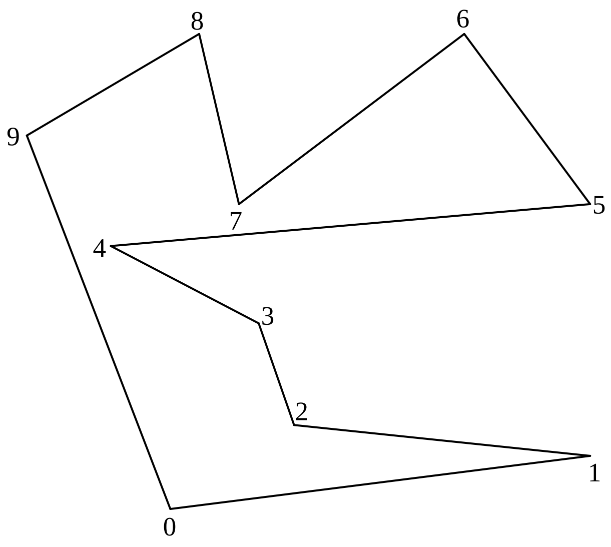

Figure 2: The original polygon

Identifying an ear of the polygon

The convex vertices of the polygon shown in Figure 2 are 0, 1, 3, 5, 6, 8, and 9. An ear should always be with a convex vertex. Then the triangles formed by those convex vertices with their two neighboring vertices are considered. If such a triangle does not contain any of the vertices of the polygons inside it, then that triangle should be an ear. In this way, ears can be found. To check whether a triangle is empty, *isEmpty* function is used.

To solve the problem inside *isEmpty* function, it is necessary to decide whether a given point is inside a triangle or not. Consider the triangle shown in Figure 3 where vertices are v, a, and b. This corresponds to the triangle inside *isEmpty* function.

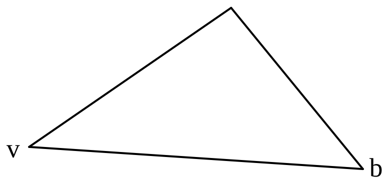

Figure 3: A triangle and a point

The purpose of *tv* function is to derive only the sign of the expression t. The value of the expression t is a large floating point number. Since it is necessary to calculate multiplications of several t values, to reduce the computational cost their signs are derived. Here *tsv* = *tv(v,a,b)*. According to the triangle in Figure 3, *tsv* = -1 since the vertices are in clockwise order. Suppose there is a point called i. If *tv(v, a, b)* and *tv(v, a, i)* have the same sign then the point i is in the same side of the triangle with respect to the edge av. In this way, it can find the position of the point i with respect to other two edges of the triangle. Also if the point i is in the same side of the triangle with respect to each three edges then the point is inside the triangle. If the point i is on and edges of the triangle then it is considered that the point is inside the triangle. Therefore, when checking each vertex of the polygon whether they are inside the triangle or not, own three vertices of the triangle should be excluded. This has been successfully done inside *isEmpty* function using *continue* keyword.

Now the algorithm seeks for ears in the available convex vertices (Theorem 1). In Figure 2, 0th vertex is not an ear. Then algorithm moves to the next convex vertex. 1st vertex is an ear.

Pruning an ear from the polygon

Once an ear was found then the algorithm prunes that ear from the original polygon. This is done by *prune* function. Figure 4 shows the polygon after pruning that ear from the original polygon.

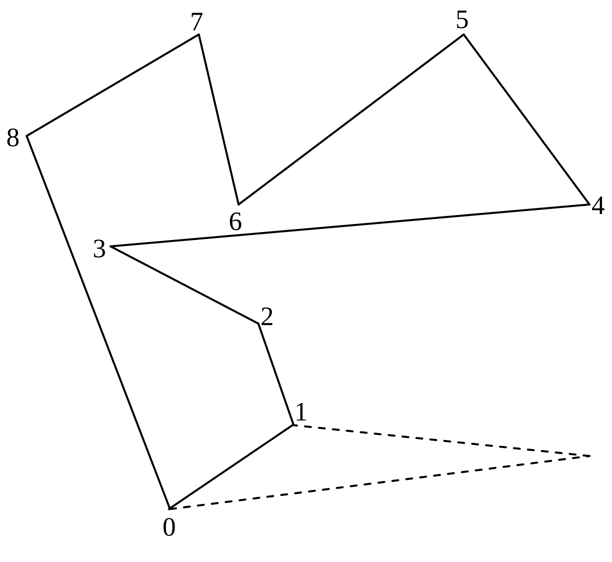

Figure 4: After pruning one ear

Iterative process

The modified polygon shown in Figure 4 can again be considered as the original polygon for the algorithm. The previous process is applied iteratively until (n – 3) diagonals can be found (Theorem 2). Figure 3 to Figure 10 shows the evolution of the polygon step by step.

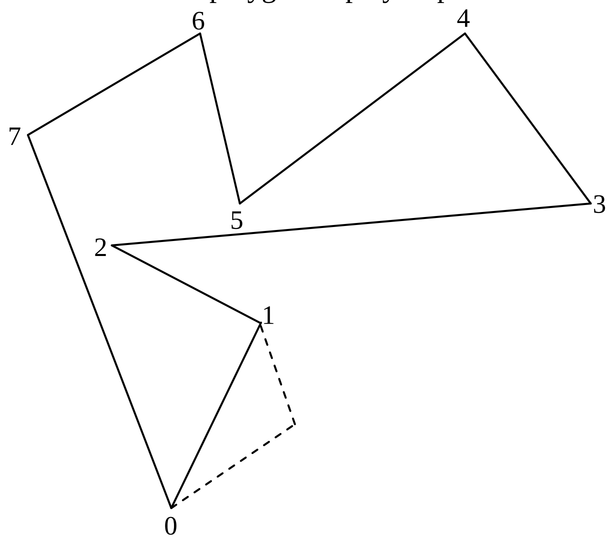

Figure 5: After pruning two ears

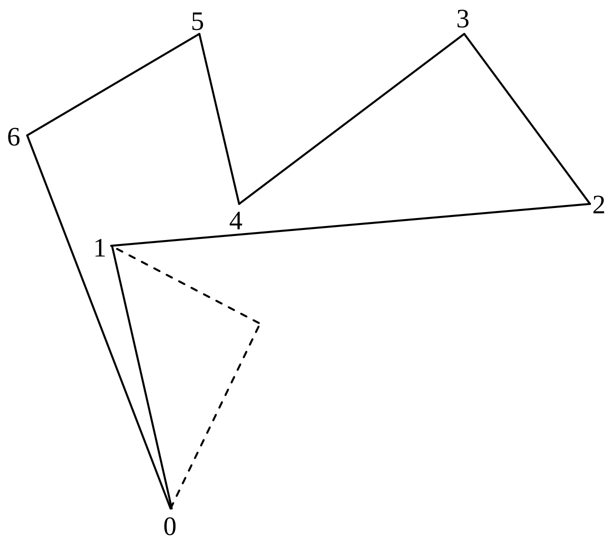

Figure 6: After pruning three ears

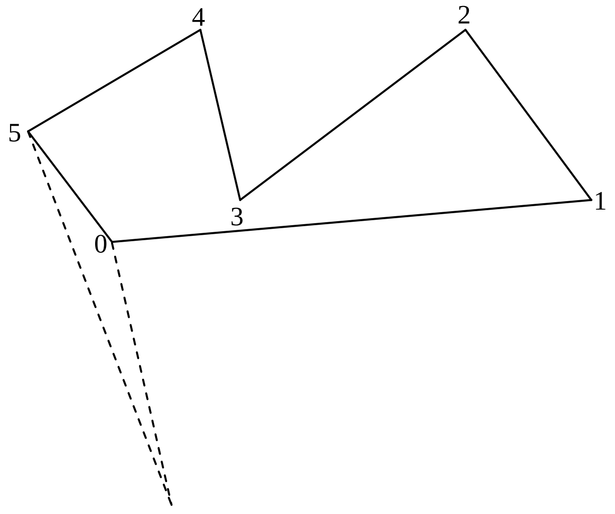

Figure 7: After pruning four ears

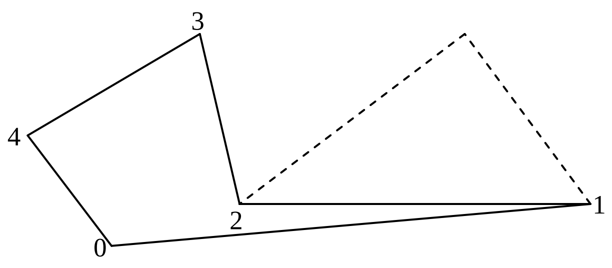

Figure 8: After pruning five ears

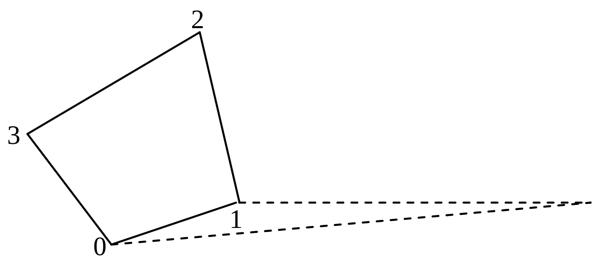

Figure 9: After pruning six ears

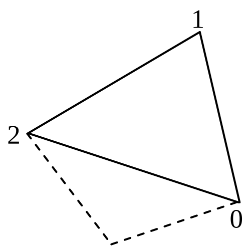

Figure 10: After pruning seven ears

In this way given polygon can be triangulated. The triangulated polygon is shown in Figure 11. Also in this algorithm has been designed without division operator in order to avoid precision error. Therefore, the algorithm can accurately perform even if the coordinates are floating point numbers.

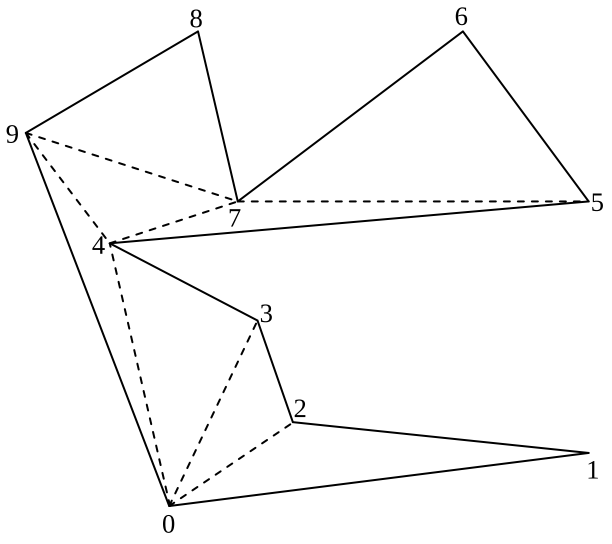

Figure 11: The triangulated polygon

RESULS AND DISCUSSION

The proposed algorithm is a 100% accurate algorithm. That means it outputs the 100% accurate coordinate values of the vertices of the triangles. The operations like division, square root calculation can lead to precision error. The proposed algorithm does not involve such operations. Therefore accuracy of the computations is 100%. The basic triangulation algorithm is also a 100% accurate triangulation algorithm (O'Rourke, 1998). Later more efficient algorithms were proposed. However they fail to maintain 100% accuracy and only provide approximate values of the coordinates of the triangles. The algorithm proposed by Gang *et al* (Mei *et al.*, 2012) computes the interior angles of each vertex of the original polygon. The computation of an angle involves sine and cosine values which leads to the precision error. The algorithm proposed by David (Eberly, 2016) also has the same drawback since it involves angle computation.

The proposed algorithm is the simplest algorithm available and it does not use any advanced mathematical concepts or advanced data structures. It provides a successful way to manipulate array data structure to achieve the goal. Even though many algorithms were proposed after the basic triangulation algorithm, the authors fail to provide successful implementations for them (Eberly, 2016). For an example, Chazelle *et al* (Chazelle *et al.*, 1982) proposed a very complicated algorithm with O (n) time complexity. That algorithm is just a theoretical methodology for triangulation and a suitable data structure to handle the algorithm has not yet been invented (Eberly *et al.*, 2016). David (Eberly *et al.*, 2016) claims his algorithm is simple, but it uses advanced data structures like trees.

Another advantage of the proposed algorithm is the minimum usage of memory. The coordinates of the vertices of the original polygon are stores using two arrays. Throughout the algorithm no additional memory is required and it is the minimum usage of memory. Further the arrays are created in static memory. Therefore the proposed algorithm is suitable for embedded systems where the memory is limited. The algorithm proposed by Meister (Meister, 1975) uses recursion which requires larger stack sizes. Therefore such algorithms are not suitable for embedded systems. The algorithm proposed by Gang *et al* (Mei *et al.*, 2012) also uses recursion. David's algorithm (Eberly, 2016) uses dynamic memory to implement tree data structure and needs lots of memory. Providing an implementation for the David's algorithm is impossible due to the hierarchical representation.

The basic triangulation algorithm is still used when 100% accuracy is necessary. The proposed algorithm is a more efficient algorithm to replace the basic triangulation algorithm. The proposed algorithm is simple, accurate, and consumes lowest memory. There are no algorithms available in literature possessing all these properties together. To validate the proposed algorithm its running time is compared with that of the basic triangulation algorithm.

Both the proposed algorithm and the basic triangulation algorithm were implemented using C programming language. Following software and hardware resources were used.

Computer: Intel(R) Pentium(R) Dual CPU; E2180 @ 2.00 GHz; 2.00 GHz, 0.98 GB of RAM;

IDE: Turbo C++; Version 3.0; Copyright(c) 1990, 1992 by Borland International, Inc;

The *clock()* function is used to measure the time consumed by algorithms in terms of clock cycles. However the time consumed by a single execution of an algorithm is very small and cannot be measured using clock cycles. Therefore an algorithm is iteratively executed 10^6 times and number of clock cycles was counted (Wijeweera *et al.*, 2013). Several polygons were used with different number of vertices as given in Table 1.

Table 1: The list of 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)

Using each polygon in Table 1, the number of clock cycles taken by 10^6 numbers of executions of each algorithm was counted and results are shown in Table 2. The problem in experimental comparison is that CPU is interrupted by the other processes running in the computer. Therefore, to compensate that average execution time is calculated after measuring the same execution five times. The results are shown in Table 2.

Table 2: Number of clock cycles comparison

Polygon	Number of Vertices	Proposed Algorithm	Basic Triangulation Algorithm	Average Ratio
1	4	21	24	1.14
2	5	19	53	2.79
3	6	42	107	2.55
4	7	68	216	3.18
5	8	101	313	3.1
6	9	82	509	6.21
7	10	181	779	4.3
8	11	244	1114	4.57
9	12	272	1588	5.84
10	13	409	2103	5.14

The average ratio is calculated by using the following formula,

Average ratio = (clock cycles for basic triangulation algorithm)/(clock cycles for proposed algorithm);

The average ratios in Table 2 prove that the proposed algorithm is faster than the basic triangulation algorithm. The running times of the basic triangulation algorithm increases with the number of vertices as seen in Table 2. This is compatible with its theoretical O (n^4) time complexity. However representing time complexity of the proposed algorithm using a function of n is impossible since its unexpected behavior. The running time of the proposed algorithm depends on two factors: the number of vertices of the polygon, the order in which the ears of the polygon are arranged. The second factor is unpredictable and that depends on the shape of the polygon and the orientation of the polygon. For an example, sudden drop of average ratio can be observed in 7th polygon in Table 2. Each ear in each iteration has been detected near at the end of the polygon vertex chain. Therefore the running time of the proposed algorithm has become longer. Similarly other sudden drops of average ratio can be explained.

To further investigate the behavior of the proposed algorithm relative to the basic triangulation algorithm, the set of polygons available from Kasun (academia.edu, 2016) is used. Since this set of polygons has large number of vertices, the execution time for 10^4 iterations is measured. The experimental results are shown in Table 3.

Table 3: Number of clock cycles comparison

Polygon	Number of Vertices	Proposed Algorithm	Basic Triangulation Algorithm	Average Ratio
1	28	39	458	11.74
2	18	6	77	12.83
3	25	22	295	13.41
4	25	9	295	32.78
5	20	11	123	11.18
6	19	7	100	14.29
7	15	13	39	3
8	19	6	100	16.67
9	19	4	100	25
10	18	5	77	15.4
11	18	11	77	7
12	18	4	77	19.25
13	17	7	62	8.86
14	22	52	177	3.4
15	24	25	246	9.84
16	24	15	246	16.4
17	23	28	206	7.36
18	20	6	122	20.33
19	14	5	28	5.6
20	15	5	39	7.8

The results in Table 3 also prove that the proposed algorithm is faster than the basic triangulation algorithm. The polygons with same number of vertices but with different shapes were also tested in Table 3. For an example, the polygons 2, 10, 11, and 12 all have 18 vertices. However the basic triangulation algorithm consumes same number of clock cycles for each different polygon. The proposed algorithm consumes different amount of time for each different polygon with same number of vertices. The proposed algorithm depends on the shape of the polygon. If the ear can be detected at the beginning of polygon chain, it takes less time. If not it takes more time.

CONCLUSION

A simple, efficient, accurate, and lowest memory consuming algorithm for triangulating a polygon was proposed. The output of the proposed algorithm is a list of triangles which is more useful in applications. It uses only the static memory and do not use mechanisms like recursion which needs larger stack memory sizes. Therefore it is more suitable for embedded systems than existing algorithms. The proposed algorithm works with 100% accuracy and suitable for applications where the exact accuracy is necessary. The proposed algorithm uses array data structure only and it does not use advanced data structures. Furthermore it uses only simple mathematical concepts. Therefore it can be used as a teaching tool for the students with little knowledge in computer programming. The experimental results show that the proposed algorithm is more efficient than the basic triangulation algorithm.

The proposed algorithm can be used for line clipping against any polygon. Since a polygon can be represented as a union of triangles, this algorithm can be used to clip a given line segment bylining against each of those triangles. Then merging those clipped line segments will give the final result of the line segment clipped against the polygon. The idea behind this algorithm could be extended to clip lines against polyhedrons.

REFERENCES

Chazelle, B. (1982). A Theorem on Polygon Cutting with Applications. *23rd Annual Symposium on Foundations of Computer Science*, pp. 339-349.

Chazelle, B.(1991). Triangulating a Simple Polygon in Linear Time. *Discrete Computational Geometry* **6** (5): 485-525.

Seidel, R. (1991). A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons. *Computational Geometry Theory and Applications.* **1**: 51-64.

Feng, H.Y.F. and Pavlidis, T. (1975), Decomposition of Polygons into Simpler Components: Feature Generation for Syntactic Pattern Recognition. *IEE Transactions on Computers.* **24**(6):636-650.

Berg, M., Cheong, O., Krefeld, M. and Overmars, M. (2008). Computational Geometry Algorithms and Applications. *Berlin Heidelberg:* Springer-Verlag.

Garey, M.R.D., Johnson, S., Preparata, F.P. and Tarjan, R. E. (1978). *Triangulating a Simple Polygon.* Information Processing Letters **7**:175-179.

Meisters, G.H. (1975). Polygons have Ears. *American Journal of Mathematics.* **82** (6): 648-651.

Elgindy, H., Everett, H. and Toussaint G. (1993). Slicing an Ear using Prune-and-Search. *Pattern Recognition Letters*. **14** (9): 719-722.

Rourke, J.O. (1998). Computational Geometry in C, *Cambridge University Press*.

Kong, X., Everett, H. and Toussaint, G. (1990). The Graham Scan Triangulates Simple Polygons. *Pattern Recognition Letters.* **11** (11): 713-716.

ftp://ftp.cis.uab.edu/pub/sloan/Software/triangulation/src/; accessed on 2016 November 11.

Held, M. (2001). FIST: Fast Industrial Strength Triangulation of Polygons. *Algorithmica*. **30**: 563-596.

Mei, G., Tipper, J.C. and Xu, N. Ear Clipping Based Algorithms of Generating High Quality Polygon Triangulation. *Proceedings of the 2012 International Conference on Information Technology and Software Engineering,* Volume 212 of the series Lecture Notes in Electrical Engineering, pp. 979-988.

Eberly, D. Triangulation by Ear Clipping, Geometric Tools. LLC, http://www.geometrictools.com; accessed on 2016 November 5.

Chazelle, B. and Incerpi, J. (1984). Triangulation and Shape Complexity. *ACM Transactions on Graphics.* **3**: 135-152.

Fournier, A. and Montuno, D.Y.(1984). Triangulating Simple Polygons and Equivalent Problems. *ACM Transactions on Graphics*. **3**: 153-174.

Green, S.L. *Advanced Level Pure Mathematics*. University Tutorial Press Ltd, Clifton House, Euston Road, London. N. W. I.

Wijeweera, K.R. and Pinidiyaarachchi, U.A.J. (2013), An Efficient Convex Hull Algorithm for a Planer Set of Points, *Ceylon Journal of Science (Physical Sciences)*. **17**: 21-30.

http://www.academia.edu/17161660/A_Collection_of_Polygons; accessed on 2016 November 09.

APPENDIX

This section includes the implementation of the proposed algorithm in C programming language.

```
#include<stdio.h>
#include<conio.h>
#include"D:/header/grap.h"

double x[]={140,220,140,250,180,100,10,120,150,150,110};
double y[]={110,110,70,70,210,160,10,30,20,60,40};
int points=11;

int dpx(double x)
{
int p;
p=(int)(x+0.5);
return p;
}

int dpy(double y)
{
int p;
p=(int)(y+0.5);
p=getmaxy()-p;
return p;
}

void drawPolygon()
{
int i,j;
        for(i=0;i<points;i++)
        {
                j=(i+1)%points;
                line(dpx(x[i]),dpy(y[i]),dpx(x[j]),dpy(y[j]));
        }
}

double t(int i,int j,int k)
{
return x[i]*(y[j]-y[k])+x[j]*(y[k]-y[i])+x[k]*(y[i]-y[j]);
}

int lv()
{
int i,index;
double miny;
miny=y[0];

        for(i=0;i<points;i++)
        {
                if(miny>y[i])
                {
                        miny=y[i];
                        index=i;
                }
        }

return index;
}

int ts(int v)
{
int a,b;
a=(v-1);
b=(v+1);
```

```
        if(a==-1)
        {
                a=points-1;
        }

        if(b==points)
        {
                b=0;
        }

        if(t(a,v,b)>0)
        {
                return 1;
        }

return -1;
}

int tv(int i,int j,int k)
{
double x=t(i,j,k);

        if(x>0)
        {
                return 1;
        }
        else if(x<0)
        {
                return -1;
        }
        else
        {
                return 0;
        }
}

int isConvex(int v)
{
        if(ts(v)*ts(lv())>0)
        {
                return 1;
        }

return 0;
}

int isEmpty(int v)
{
int a,b,i,tsv;
a=(v-1);
b=(v+1);

        if(a==-1)
        {
                a=points-1;
        }

        if(b==points)
        {
                b=0;
        }

        tsv=tv(v,a,b);

        for(i=0;i<points;i++)
        {
                if((i==v) || (i==a) || (i==b))
                {
                        continue;
                }

                if((tsv*tv(v,a,i)>=0) && (tsv*tv(a,b,i)>=0) && (tsv*tv(b,v,i)>=0))
                {
                        return 0;
                }
        }

return 1;
}

void prune(int v)
{
int i;
        points--;

        for(i=v;i<points;i++)
        {
                x[i]=x[i+1];
                y[i]=y[i+1];
        }
}

void main()
{
int i,n;
int diagonals;
ginit();

diagonals=points-3;

setcolor(15);
drawPolygon();
delay(2000);
setcolor(0);
drawPolygon();

for(n=0;n<diagonals;n++)
{
        for(i=0;i<points;i++)
        {
                if(isConvex(i) && isEmpty(i))
                {
                        prune(i);
                        break;
                }
        }

setcolor(15);
drawPolygon();
delay(1000);
setcolor(0);
drawPolygon();
}

getch();
gexit();
}
```