*Ceylon Journal of Science 46(1) 2017: 55-66*
*DOI: http://doi.org/10.4038/cjs.v46i1.7418*

**RESEARCH ARTICLE**

# Convex partitioning of a polygon into smaller number of pieces 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: 20/02/2017*

**Abstract:** Designing an algorithm to deal with a convex shape is easier than that for a concave shape. Efficient algorithms are required to process concave shapes, because every application does not deal with convex shapes. An alternative approach is to first transform a concave shape into a set of convex shapes so that efficient algorithms available for convex shapes can be utilized. This paper proposes an algorithm for partitioning a concave polygon into smaller number of convex pieces. Each resultant convex piece then can be processed using a simple algorithm applicable to convex shapes. In this way, any arbitrary shape can be processed using such simple algorithms with the aid of the proposed algorithm. There are plenty of algorithms available in literature to solve convex partitioning problem. Hertel Mehlhorn algorithm is the most efficient algorithm, but it does not minimize the number of convex pieces satisfactorily. Further the Hertel Mehlhorn algorithm is the minimum memory consuming algorithm available in literature. Later Geene proposed an algorithm which gives the minimum number of convex pieces. It uses dynamic programming technique and needs high amount of memory. Therefore Geene's algorithm is not suitable for systems where the memory is limited. Chazelle proposed an efficient algorithm which gives minimum number of convex pieces. However a data structure to implement Chazelle's algorithm is not available in literature. The experimental results showed that the proposed algorithm gives smaller number of convex pieces than the Hertel Mehlhorn algorithm. The memory consumption of the proposed algorithm is also lower than the Hertel Mehlhorn algorithm.

***Keywords:*** Computational Geometry, Computer Graphics Programming, Coordinate Geometry, Euclidian Geometry, Computer Programming.

## INTRODUCTION

In computer graphics, images can be represented using polygons. Clipping those polygons against a given window and filling them using a particular color are two major tasks. And these two processes play as bottlenecks for any graphics application. Designing an algorithm for a convex polygon is easier than that for a concave polygon (O'Rourke, 1998). The reason is that then most of the indeterminate cases and special cases can be omitted. Therefore, clipping and filling algorithms for convex polygons performs faster than that of concave polygons. Once a concave polygon is partitioned into a set of convex polygons, each convex polygon can be clipped or filled. Therefore, partitioning a concave polygon into convex polygons is useful.

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 $v_0$, $v_1$, $v_2 \ldots$, $v_{n-1}$ 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 $e_0 = v_0v_1$, $e_1 = v_1v_2 \ldots$, $e_i = v_iv_{i+1} \ldots$, $e_{n-1} = v_{n-1}v_0$ 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 $i = 0..., n - 1$.
2) Nonadjacent 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 also the closed curve is simple since non adjacent segments do

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

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).

A polygon P is called convex if x in P and y in P implies that the segment xy is a subset of P. A vertex is reflex if its internal angle is greater than $\pi$; otherwise a vertex is called convex. Therefore, any polygon with a reflex vertex is not convex (Green). Partitioning into triangles is a special case of partitioning into convex pieces (Wijeweera *et al*., 2016). But triangulation is by no means optimal in the number of convex pieces.

There are two major goals of convex partitioning: (1) partition a polygon into as few pieces as possible and (2) perform the task as fast as possible. The two goals conflict of course. There are two main approaches. First, compromise on the number of pieces: Find a fast algorithm whose inefficiency in terms of the number of pieces is bounded with respect to the optimum. Second, compromise on the time complexity: Find an algorithm that gives an optimal partition, as fast as possible (O'Rourke, 1998). The proposed work focuses on the first approach.

Two types of partitioning of a polygon P can be identified: a partition by diagonals or a partition by segments. The difference is that diagonal endpoints must be vertices of the polygon. Partitioning by segments is normally more complex in that their endpoints must be computed somehow; however the freedom to look beyond the set of vertices often results in smaller number of partitions (O'Rourke, 1998).

To evaluate the number of convex pieces, it is necessary to have bounds on the best possible partition. Chazelle (O'Rourke, 1998) proposed a theorem to find the smallest number of convex pieces. **Theorem:** Let *k* be the fewest number of convex pieces into which a polygon may be partitioned. For a polygon of *r* reflex vertices, CEIL $(r/2) + 1 \leq k \leq r + 1$.

Hertel and Mehlhorn found a very clear algorithm that partitions with diagonals quickly and has bounded "badness" in terms of the number of convex pieces (O'Rourke, 1998). In some convex partitioning of a polygon by diagonals, call a diagonal *d* essential for vertex *v* if removal of *d* creates a piece that is non-convex at *v*. Clearly if *d* is essential it must be incident to *v*, and *v* must be reflex. A diagonal that is not essential for either of its endpoints is called inessential.

Hertel Mehlhorn's algorithm is simply this: Start with a triangulation of polygon P; remove an inessential diagonal; repeat. Obviously this algorithm gives a partitioning of P by diagonals into convex pieces. The algorithm can be accomplished in linear time with the use of appropriate data structures. Therefore the only problem is how far from the optimum number of pieces might it be. The Hertel Mehlhorn algorithm is the most efficient algorithm available in literature. Further, it can be implemented using only static memory which is useful in embedded systems programming.

Finding a convex partitioning optimal in the number of pieces is much more time consuming than finding a suboptimal one. Greene (O'Rourke, 1998) proposed the first algorithm to find an optimal convex partitioning of a polygon with diagonals. The Greene's algorithm runs in $O(r^2n^2)$ time. It uses dynamic programming approach to achieve the goal. This algorithm is very inefficient and in the worst case it has $O(n^4)$ time complexity. Its implementation uses recursion for dynamic programming. Therefore memory consumption is high and it is not suitable for embedded systems.

If the partitioning may be created with arbitrary line segments, then the problem is even more complicated, as it might be required to utilize line segments that do not touch the polygon boundary. Chazelle (O'Rourke, 1998) solved this problem with a very complicated $O(n + r^3)$ algorithm. However in the worst case, it needs $O(n^3)$ time complexity. Developing a data structure to handle this algorithm is extremely difficult. Therefore corresponding implementation of this algorithm is not available in literature.

Recent advances have also occurred in convex partitioning algorithms. Lien and Amato proposed an algorithm to decompose a polygon into approximate convex pieces. This algorithm runs in O (nr) time complexity where *r* is the number of reflex vertices of the polygon (Lien *et al*., 2006). Achieving the same time complexity, Hongguang and Guozhao also proposed another

algorithm for convex partitioning with approximate convex pieces (Hongguang *et al*., 2015). However these algorithms fail to produce exact convex pieces.

An algorithm that produces smaller number of convex pieces than the Hertel Mehlhorn algorithm is proposed in this paper. The memory usage of the proposed algorithm is lesser than the Hertel Mehlhorn algorithm. The proposed algorithm consists of three major steps.

1) Identifying the reflex vertices of the polygon.
2) Considering reflex vertices pair wise and trying for convex partitioning through the segments joining each pair if possible.
3) Partitioning the polygon by sectoring remaining reflex vertices.

## METHODOLOGY

This section describes the proposed algorithm. The methodology has been divided in to subsections in order to increase the understandability. The reader is supposed to use the program appear in appendix for easy understanding.

### Representation of the Polygon

The vertices of the polygon is stored using two arrays named *x* and *y*. And the *points* variable stores the number of vertices of the polygon. The $i^{th}$ vertex of the polygon is denoted by (*x*[*i*], *y*[*i*]) where *i* = 0…, (*points* - 1). Before displaying any point on the graphics screen, coordinates are modified using the *dpx* and *dpy* functions. The inbuilt *line* function which is used to draw a line segment on the screen takes coordinates of two end points in integer form. If the decimal values were sent as arguments they would be truncated. This can lead incorrect approximations. For an example, 1.9 will be approximated to 1 instead of 2. Therefore, better approximation should be done. Adding 0.5 to the coordinate values before truncation can solve this problem (Kodituwakku *et al*., 2013). This is done inside the *dpx* and *dpy* functions.

Furthermore, the integrated development environment used here has the origin in top left corner. To get the origin to the bottom left corner {*getmaxy*() – *y*} is used instead of *y* in *dpy* function. Here *getmaxy*() is an inbuilt function which returns the height of the graphics screen in pixels. The *drawPolygon* function is used to draw the polygon. There modulo (%) operator is used to handle the cyclic order of vertices.

### Finding the Lowest Vertex of the Polygon

The *lv* function returns the index of the lowest vertex of the polygon. There may be more than one lowest vertex for a given polygon. In such cases, the index of one particular lowest vertex should be returned. Inside the *lv* function, *miny* variable is used to store the minimum value of y-coordinates of the vertices of the polygon. Initially, *miny* is equal to *y*[0]. Using a *for* loop, each y-coordinate of the vertices are accessed. If a smaller value than current *miny* is found then *miny* is updated by that value.

**Lemma:** Lowest vertex is always convex.

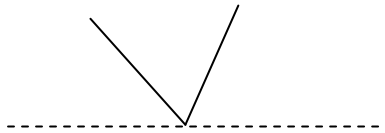

**Figure 1:** Lowest Vertex of the Polygon.

**Proof:** Suppose the Figure 1 shows the lowest vertex of the polygon. Since it is the lowest there cannot exist vertices below the horizontal line drawn through it. Therefore, the edges connecting to that vertex are above that horizontal line implying that the vertex should be convex. Thus the lemma is proved.

### An Important Mathematical Result

Suppose $P_1(x_1, y_1)$, $P_2(x_2, y_2)$ and $P_3(x_3, y_3)$ are three points situated on a plane. Consider the following equation,

$$t = x_1 * (y_2 - y_3) + x_2 * (y_3 - y_1) + x_3 * (y_1 - y_2);$$

If $P_1$, $P_2$ and $P_3$ are situated in clockwise order then *t* is less than zero. If $P_1$, $P_2$ and $P_3$ are situated in anticlockwise order then *t* is greater than zero. And if the three points are collinear then *t* is equal to zero. The value of *t* is calculated using the *t* function. The value of *t* is usually a large value and it would consume a higher computational cost to manipulate. To overcome this issue the *tv* function is used which returns the sign of *t* value using {-1, 0, +1} values.

### Calculating the Sign of a Vertex

Suppose *v* is the index of a given vertex of the polygon. The two neighboring vertices *a* and *b*

can be calculated by $a = v - 1$; $b = v + 1$. But there are two exceptions. First, when $v = 0$, $v - 1 = -1$; but actually it should be (*points* - 1). Second, when $v = points - 1$, $v + 1 = points$; but actually it should be 0. Now considering {*a*, *v*, *b*} points as {$P_1$, $P_2$, $P_3$} respectively, the sign of the vertex can be calculated. This is done by using the *getWise* function.

**Finding the Reflex Vertices of the Polygon**

As the first step of the algorithm, all the reflex vertices of the polygon should be found. The *reflex* array is used to store the state of a vertex. The $i^{th}$ element of the *reflex* array should be set to 1 if the $i^{th}$ vertex of the polygon is reflex or else it is 0. Initially all the elements of the *reflex* array are zero. The corresponding elements of the reflex array to the reflex vertices are set to 1 using *setReflex* function.

Inside the *setReflex* function, *min* variable stores the index of the lowest vertex derived from executing *lv* function. A *for* loop is used to access each vertex of the polygon. Using the *getWise* function the sign of the vertex is obtained. The lowest vertex is always convex as proved earlier. And it is obvious that if two vertices of the polygon have different signs then one should be convex and the other one should be reflex. Depending on these facts, it can be deduced that if the lowest vertex and a given vertex have different signs then that the given vertex is reflex. That means *getWise*(*i*) * *getWise*(*min*) < 0 implies $i^{th}$ vertex is reflex. In such case *reflex*[*i*] should be set to 1.

**Two Points with respect to a Line Segment**

Suppose L is a line segment with $P_1$ ($x_1$, $y_1$) and $P_2$ ($x_2$, $y_2$) as end points. Then $P_3$ ($x_3$, $y_3$) and $P_4$ ($x_4$, $y_4$) are the given two points. The *isTwoSides* function returns 1 if $P_3$ and $P_4$ are situated in two sides of the extended line segment L. Otherwise it returns 0. The *tv* function is used for this purpose. It is obvious that $P_3$ and $P_4$ are in two sides of L if the values of *tv*{$P_1$, $P_2$, $P_3$} and *tv*{$P_1$, $P_2$, $P_4$} have different signs. That means if the multiplication of these two values is negative then the two points are in two sides.

**Test for X-intersection of Two Line Segments**

Suppose $L_1$ and $L_2$ are two line segments. End points of $L_1$ are $P_1$ ($x_1$, $y_1$) and $P_2$ ($x_2$, $y_2$)} while end points of $L_2$ are $P_3$ ($x_3$, $y_3$) and $P_4$ ($x_4$, $y_4$). The *isIntersect* function returns 1 if these line segments intersect or else returns 0. If following two conditions are satisfied then $L_1$ and $L_2$ intersect.

1) $P_3$ and $P_4$ should be in two sides of $L_1$.
2) $P_1$ and $P_2$ should be in two sides of $L_2$.

Therefore, if the multiplication of *isTwoSides*{$P_1$, $P_2$, $P_3$, $P_4$} and *isTwoSides*{$P_3$, $P_4$, $P_1$, $P_2$} is equal to one then the two line segments should intersect. This kind of intersection is called exclusive intersection (x-intersection).

**Test whether a Given Point is Inside the Polygon or Not**

This is done by the *isInside* function. It returns 1 if the point (*xc*, *yc*) is inside the polygon. Otherwise it returns 0. In Figure 2, point P is outside the polygon and point Q is inside the polygon. A horizontal infinite line is drawn beginning from the given point. That lines corresponding to each of those two points are shown in dotted arrows. It can be observed that the arrow thorough the point P cuts polygon in two points. And the arrow through the point Q cuts the polygon in one point.

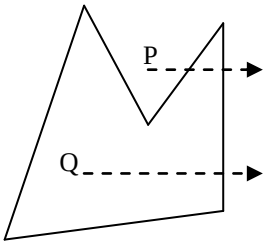


**Figure 2:** Number of Intersections.

Having a cut with the polygon means the arrow goes either inside or outside the polygon. Therefore, if the number of cuts is even then the given point should be outside the polygon. And if the number of cuts is odd then the given point should be inside the polygon (Wijeweera *et al.*, 2014).

The *pos* function is used to find the position of a vertex with index *k* with respect to the line y = *yc*. It returns 1, 0 or -1 if the point is above the line, on the line or below the line respectively. This idea can be used to check whether an edge of the polygon intersects the horizontal line or not. Suppose *i* and *j* are the indices of the vertices of an edge of the polygon. If *pos* (*i*) and *pos* (*j*) have opposite signs that

mean the vertices of the edge are in two sides of the horizontal line implying that the edge intersects the horizontal line.

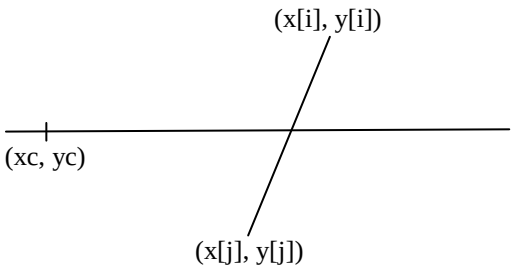


**Figure 3:** Intersection with the Line y = *yc*.

As shown in Figure 3, if an edge intersects with the y = *yc* line then the intersection point should be calculated as follows.

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[i] + (y – y[i]) * (x[j] – x[i]) / (y[j] – y[i]);

Let (*xp*, *yp*) be the intersection point. Then by substituting y = *yc* in above equation,

xp = x[i] + (yc – y[i]) * (x[j] – x[i]) / (y[j] – y[i]);

The variable *p* is maintained to store the number of intersections. For each new point, *p* is set to zero at the beginning. For each edge which satisfies the condition *xp* > *xc*, the value of *p* should be incremented by one. At this moment *p* stores the number of intersections of the horizontal line from the right side of the new point. The value of *p* is expected to decide whether the new point is inside or outside the polygon. There are four special cases which should be handled carefully when incrementing the value of *p* in addition to intersections. These four cases are not detected under intersections.

**Case 1:** This situation can be described as a vertex touching the horizontal line as shown in Figure 4.

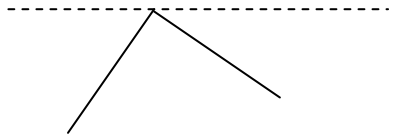

**Figure 4:** Vertex is Touching the Line y = *yc*.

Even if the touching point is on the right side of the new point, this situation is ignored.

**Case 2:** This situation can be described as the horizontal line goes through a vertex as shown in Figure 5.

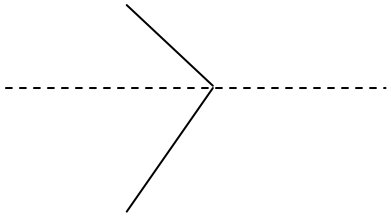

**Figure 5:** Line y = *yc* Goes through the Vertex.

In this kind of a situation, if the vertex is on the right side of the new point then the value of *p* should be incremented by one.

**Case 3:** This situation can be described as an edge touching the horizontal line as shown in Figure 6.

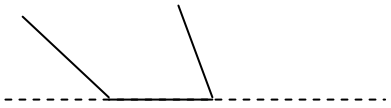

**Figure 6:** Edge Touching Line y = *yc*.

In this kind of a situation even if the end points of the touching edge is on the right side of the new point, this situation is ignored.

**Case 4:** This situation can be described as the horizontal line goes through an edge as shown in Figure 7.

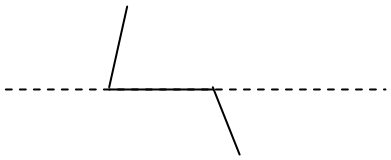

**Figure 7:** Line y = *yc* goes through the Edge

In this kind of a situation, if at least one of the end points of the touching edge is on the right side of the new point, then value of *p* should be incremented by one.

In the above four cases, case 1 and case 3 are not considered. Because those situations do not cause the horizontal line get inside or outside the polygon. But case 2 and case 4 should be considered as intersections since those two situations cause horizontal line get inside or outside the polygon.

The first two cases deal with a vertex. Touching vertices which are on the right side of

the new point can be identified by checking the condition (y = *yc* AND x > *xc*) where (*x*, *y*) is the coordinates of a vertex. Distinguishing case 1 and case 2 should be done as follows. Suppose *i* is the index of the vertex that is touching the horizontal line. Then indices of its two neighboring vertices *a* and *b* can be computed as $a = i - 1$, $b = i + 1$. But there are two exceptional cases. If $i = 0$ then $i - 1 = -1$. Therefore it should be set to (*points* - 1) since it is the actual index of the neighboring vertex. If $i = points - 1$ then $i + 1 = points$. Therefore, it should also be set to 0 since it is the actual index of the neighboring vertex. This is due to the cyclic ordering of the vertices. If *pos (a)* * *pos (b)* < 0 then it means the vertices *a* and *b* are in two sides of the horizontal line. In other words, it is the case 2. If this condition is satisfied only the value of *p* is incremented by one.

The second two cases deal with an edge. Let *i* and *j* be the indices of the end points of the touching edge where $j = (i + 1) \% points$. The edges which satisfies the condition (y = yc AND x> xc) where (*x*, *y*) is the coordinates of the $j^{th}$ point are considered. Then the indices of neighboring vertices *a* and *b* of the edge can be computed $a = i - 1$, $b = j + 1$. In this situation, two exceptional cases arise due to the cyclic ordering of the vertices. That should also be resolved as in first two cases. By considering *pos (a)* * *pos (b)* value as above two cases, case 3 and case 4 can be distinguished. Here only in case 4, value of *p* is incremented by one.

Finally, by checking the value of *p*, it can be decided that whether the new point is inside or outside the polygon. If *p* is even that means the new point is outside the polygon. And if *p* is odd that means the new point is inside the polygon. This fact is obvious and does not need a proof. Also this is done using modulo (%) operator.

**Finding the Primary Sectors**

The convex partitioning process is done by drawing line segments inside the polygon. Those line segments are called sectors. These sectors are found in two stages namely primary sectors and secondary sectors. This section describes the process of finding primary sectors.

The number of sectors is stored in the *secs* variable which is initially equal to zero. The end points of the sectors are stored in arrays. Four arrays *sx*, *sy*, *ex* and *ey* are used such that (*sx*[*i*], *sy*[*i*]) and (*ex*[*i*], *ey*[*i*]) denote the end points of the $i^{th}$ sector. Finding primary sectors is done using the *setSectors1* function.

Here all the line segments joining pairs of reflex vertices are tested for being sectors. Two nested *for* loops are used to access all possible pairs of vertices. From those pairs in which both the vertices are reflex should be selected. Therefore (*reflex*[*i*] * *reflex*[*j*] = 0) condition denies the situations where at least one of the vertices are not reflex. Here the *continue* keyword has been used to skip those situations. A line segment in which both vertices are reflex may coincide with an edge of the polygon if both of the vertices of the edge are reflex. Such cases are excluded by testing for the condition ($j - i > 0$). The *found* variable is initialized to zero. Then another *for* loop is used. Using this loop, each of the edges of the polygon is accessed and with the *isIntersect* function they are tested for x-intersection with the line segment. If at least one of the edges x-intersect with the line segment then *found* variable would be set into one.

If still the *found* variable is equal to zero then using a *for* loop, each vertex is tested whether the vertex is inside the line segment. If the *tv* value is equal to zero then it can be decided that the three points provided as arguments to the *tv* function are collinear. The projections on principle axes are considered to decide whether the given vertex is interior the line segment or not. The problem is divided into two categories as the line segment is parallel to y-axis and not parallel to y-axis. In first category, the projection on x-axis is just a point and therefore the projection on y-axis is considered. In second category, there is a possibility that the line segment may be parallel to x-axis causing the projection of y-axis is just a point. Therefore, projection on x-axis is considered in second category. If the vertex is inside the line segment then the *found* variable should be set into one.

If still the found variable is zero then that means the line segment does not have x-intersections with any of the edges of the polygon. And also none of the vertices of the polygon are inside the line segment. Now there are two possible situations. The line segment may lie inside the polygon or outside the polygon. First the midpoint (*midx*, *midy*) of the line segment is computed. If the midpoint is inside the polygon then the line segment also lies inside the polygon. If that line segment lies

inside the polygon and if the polygon was divided into two parts through the segment gives two convex polygons then the line segment is added as a segment into the list of segments. To check whether that line segment separates the polygon into two convex polygons the *isUseful* function is used.

**The *isUseful* function**

The line segment is shown in Figure 8 by dotted line. The angles *a*, *b*, *c*, *d* formed by the line segment should be less than or equal to π radians for the line segment to be eligible as a sector.

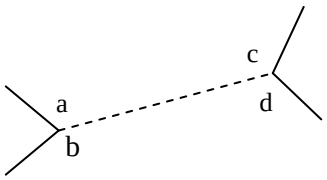


**Figure 8:** Angles in Two Sides.

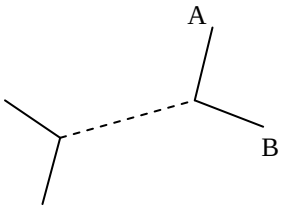


**Figure 9:** In Two Sides.

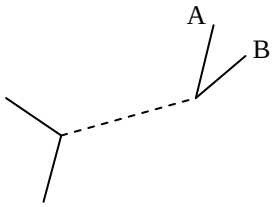


**Figure 10:** In Same Side.

Figure 9 and Figure 10 illustrate an important result. If points A and B are situated in the same side of the extended line segment then the line segment can't be a sector. In Figure 10, it shows that the *d* angle is greater than π. This is found by the *isUseful* function which returns one if the line segment is sector or else returns zero. There *i* and *j* represents the indices of the line segment. Initially the value of *t* is zero. First the vertex *i* is tested. Multiplication of *tv* values being positive implies that an angle greater than π is created. Therefore, in such cases *t* is set to one. The same test is done for the vertex *j* as well. If the value of *t* remains as zero after those two tests then the line segment is qualified as a sector.

**Finding the Intersection Point of Two Straight Lines**

Any two straight lines intersect unless they are parallel. The *findIntersection* function has been designed excluding the parallel case. A test should be performed to check for parallelism before invoking the function. The function is invoked only if the two lines are not parallel. Let's consider two lines $L_1$ and $L_2$ formed by end points $\{P_1 (x_1, y_1), P_2 (x_2, y_2)\}$ and $\{P_3 (x_3, y_3), P_4 (x_4, y_4)\}$ respectively as shown in Figure 11. These eight coordinate values are sent as arguments to the *findIntersection* function and the intersection point $(x_n, y_n)$ is calculated using the function. And let's take $(m_1, c_1)$ and $(m_2, c_2)$ as the (gradient, y-intercept) of each line respectively if they exist.

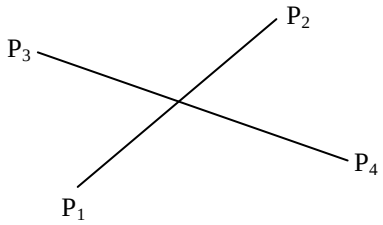


**Figure 11:** Intersection of Two Lines.

The diagram shown in Figure 12 illustrates the possibilities of those two lines.

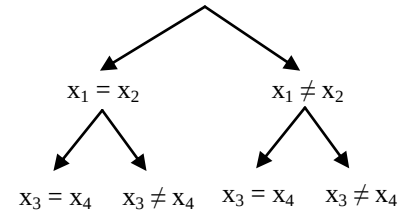


**Figure 12:** Possibilities of Two Lines.

**Case 1; ($x_1 = x_2$; $x_3 = x_4$):**

This case is ignored and not handled using the function. Only coordinates of the non-parallel lines are sent to the function.

**Case 2; ($x_1 = x_2$; $x_3 \neq x_4$):**

In this case, $L_1$ is parallel to the y-axis and $L_2$ is not parallel to the y-axis. Therefore, the gradient of $L_2$ can be calculated as follows.

$m_2 = (y_3 - y_4) / (x_3 - x_4)$;

The equation of a straight line is y = m * x + c where m and c are the gradient and y-intercept. By substituting ($x_3$, $y_3$) in this equation $c_2$ can be computed as follows.

$c_2 = y_3 - m_2 * x_3$;

Since $L_1$ is parallel to y-axis, $x_n = x_1$. By substituting this value in the equation of $L_2$, $y_n$ value can be computed as follows.

$y_n = m_2 * x_n + c_2$;

**Case 3; ($x_1 \neq x_2$; $x_3 = x_4$):**

In this case, $L_1$ is not parallel to y-axis and $L_2$ is parallel to y-axis. The computations can be done similarly as in case 2.

**Case 4; ($x_1 \neq x_2$; $x_3 \neq x_4$):**

Here both $L_1$ and $L_2$ are not parallel to y-axis. Therefore the gradients of both lines are defined. Then the equations of those lines are $y = m_1 * x + c_1$ and $y = m_2 * x + c_2$. Therefore,

$m_1 * x_n + c_1 = m_2 * x_n + c_2$;
$(m_1 - m_2) * x_n = c_2 – c_1$;
$x_n = (c_2 - c_1) / (m_1 - m_2)$;

By substituting $x_n$ value in the equation of $L_1$,

$y_n = m_1 * x_n + c_1$;

**The *setClosest* Function**

The *setClosest* function takes two parameters *i* and *j* which denote the indices of two end points of a given edge.

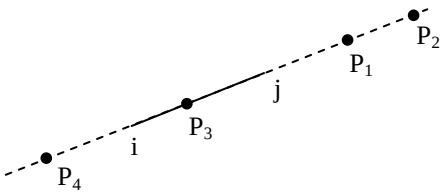


**Figure 13:** Points on the Edge.

Figure 13 shows an example edge with *i* and *j* indices for end vertices and four points on the extended edge. The purpose of the *setClosest* function is to find the closest point to *j* vertex which is situated in the opposite side of *i* vertex with respect to *j* vertex. According to the example, the goal point should be $P_1$.

For each point (*xn*, *yn*) the *setClosest* function is invoked. The current closest point is stored in (*xg*, *yg*) coordinates. The *found* variable is used to detect a particular invocation is the first invocation or not. Initially the *found* variable is equal to zero and after first invocation it is set to one. First *dx* and *dy* variables are calculated. The projections on x-axis and y-axis are considered to find the positions of points. If the absolute value of *dx* is greater than absolute value of *dy* then the projection on x-axis is considered, otherwise projection on y-axis is considered. This concept helps to avoid two situations which can cause erroneous results. First situation is when the edge is parallel to a particular axis. Then projection on the other axis is just a point and relative positions of the points cannot be detected. Therefore in this situation, projection on the parallel axis is considered. Second situation is when the projection on a given axis is approximately a point. In this situation, considering the projection on this axis can cause erroneous results due to precision error. Therefore, the projection on the other axis in considered.

Let's consider the situation where the projection on x-axis is considered. The expression {$dx * (xn – x[j]) < 0$} is true only for the points which are on the opposite side of vertex *i* with respect to vertex *j*. If the *found* variable is zero then it means this is the first invocation of the *setClosest* function. Then the (*xg*, *yg*) should be assigned (*xn*, *yn*). If not two possible situations should be considered as $dx > 0$ and $dx \leq 0$. If $dx > 0$ then $xn > xg$ means (*xn*, *yn*) is closer to vertex *j* than (*xg*, *yg*). Therefore (*xg*, *yg*) is updated by (*xn*, *yn*). Similarly, $dx \leq 0$ situation can be understood.

When the projection on y-axis is considered the similar mechanism can be used.

**An Illustrative Example**

In this section, the algorithm described so far is illustrated using the example polygon shown in Figure 14. The indices of first two vertices are shown. The ordering of vertices is anticlockwise. However the proposed algorithm works for both clockwise and anticlockwise polygons.

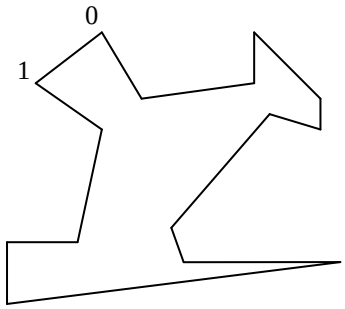


**Figure 14:** The Original Polygon.

First step is to find all the reflex vertices of the polygon. The reflex vertices have been marked using small circles in Figure 15.

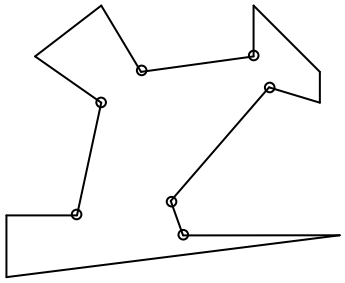

**Figure 15:** Reflex Vertices.

The primary sectors are shown in Figure 16 using dotted lines. The end vertices of a primary sector become non-reflex.

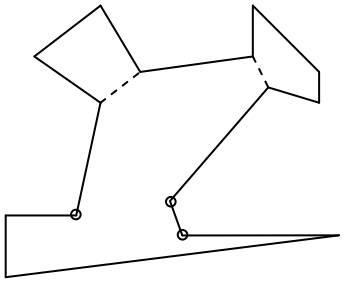

**Figure 16:** Primary Sectors.

## Finding the Secondary Sectors

Finding the secondary sectors is done using the *setSectors2* function. Each edge is accessed using a *for* loop. The *i* and *j* denotes the indices of end points of an edge where $j = (i + 1) \% points$. The edges where $reflex[j] = 1$ are selected using the *if* condition. The *found* variable is set to zero initially which is accessed inside the *setClosest* function.

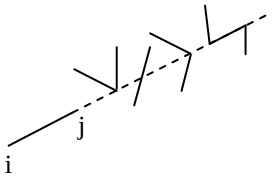


**Figure 17:** Extension of an Edge.

If the edge is extended in *ij* direction to infinity as shown in Figure 17. The extended part of the edge may x-intersect with edges of the polygon and sectors which are already available in sector list. In such cases, intersection points should be calculated. Also it may go through the vertices of the polygon and end points of the sectors. Out of those intersection points and go through points, the closest point to *j* vertex is selected. A new sector is added using each edge extensions by taking *j* vertex and the closest point to *j* vertex as its end points.

Five *for* loops are used to implement above task in the algorithm. Following sections describe the functionality of each *for* loop.

First *for* Loop: Each edge of the polygon is tested using the *isTwoSides* function whether their end points x-intersect the extended edge. The intersection point is calculated using the *findIntersection* function. Then the *setClosest* function is invoked.

Second *for* Loop: The collinear points with *i* and *j* vertices are found using the value of *t*. If the value of *t* is zero then it implies that the corresponding three points are collinear. Then the *setClosest* function is invoked.

Third *for* Loop: The sectors are tested. This is similar to first *for* loop.

Fourth *for* Loop: The starting points of the sectors are tested. This is similar to second *for* loop.

Fifth *for* Loop: The ending points of the sectors are tested. This is also similar to second *for* loop.

## Back to the Example

The Figure 16 shows the primary sectors of the polygon. This section is used to illustrate secondary sectors. According to the example (2, 3), (6, 7) and (7, 8) edges create secondary sectors.

Ceylon Journal of Science 46(1) 2017: 55-66

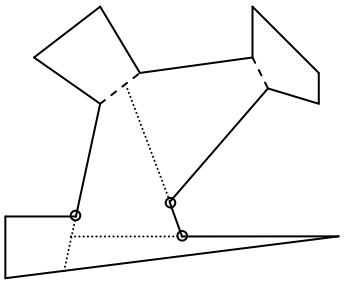

**Figure 18:** Secondary Sectors.

Figure 18 shows the polygon after finding the secondary sectors which are shown in different type of dotted lines. When an edge is extended to create a secondary sector then the relevant reflex angle is divided into two. On one side it creates angle π which is not considered as a vertex of a convex piece. A reflex angle is always less than 2π. Therefore on the other side the angle is always less than π. It can be observed that now the polygon has been divided into convex pieces.

## Comparison with Hertel Mehlhorn Algorithm

Following Figure 19 and Figure 20 illustrate the proposed algorithm and Hertel Mehlhorn algorithm respectively applied to the same polygon.

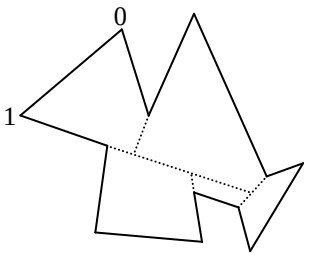


**Figure 19:** Proposed Partitions.

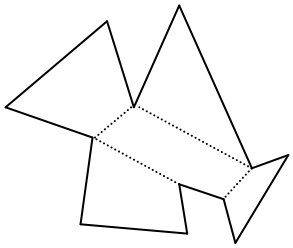

**Figure 20:** H-M Partitions.

According to above illustration, both algorithms give same number of convex partitions. However the set of convex pieces given by each algorithm is different in general. The Hertel Mehlhorn algorithm partitions the polygon using diagonals. The proposed algorithm uses both diagonals and line segments. The proposed algorithm gives equal or lesser number of convex pieces than the Hertel and Mehlhorn algorithm as proved in the results and discussion section through experimental results.

## Why is It Not the Minimum Convex Partitioning?

The proposed algorithm may fail to give the minimum convex partitioning for some polygons. To explain this, the same polygon shown in Figure 19 is used. The example can be partitioned using only four convex pieces as shown in Figure 21.

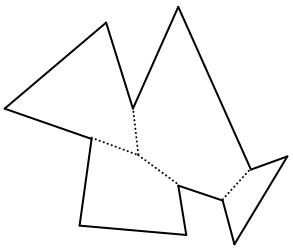

**Figure 21:** Minimum Partitions.

## RESULTS AND DISCUSSION

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

Several polygons were used with different number of vertices as given in Appendix.

The algorithm was compared with the implementation of the Hertel Mehlhorn for number of pieces. The results are shown in Table 1. The polygons were generated randomly and see appendix for the coordinates of the vertices of the polygons.

**Table 1:** The Number of Pieces Comparison.

| Polygon | Proposed algorithm | Hertel Mehlhorn algorithm |
|---|---|---|
| 1 | 14 | 19 |
| 2 | 7 | 10 |
| 3 | 11 | 14 |
| 4 | 10 | 13 |
| 5 | 8 | 9 |
| 6 | 8 | 12 |
| 7 | 7 | 10 |
| 8 | 9 | 11 |
| 9 | 8 | 11 |
| 10 | 8 | 11 |
| 11 | 7 | 10 |
| 12 | 7 | 9 |
| 13 | 7 | 9 |
| 14 | 9 | 14 |
| 15 | 11 | 13 |
| 16 | 10 | 15 |
| 17 | 12 | 12 |
| 18 | 8 | 11 |
| 19 | 6 | 7 |
| 20 | 6 | 7 |

The average ratio (Pieces, HM: Proposed) is computed using following equation.

Average Ratio (Pieces, HM: Proposed) = (∑ Number of pieces from HM algorithm)/ (∑ Number of Pieces from proposed algorithm)

Average ratio (Pieces, HM: Proposed) = 227/173 =1.3

The experimental results show that the proposed algorithm produces equal or lesser number of convex pieces. In general, the proposed algorithm is 1.3 times better than the Hertel Mehlhorn algorithm in number of pieces.

The efficiency of the proposed algorithm was also compared with that of the Hertel Mehlhorn algorithm. The time consumed for a single execution of the algorithms is not measurable since the duration is too small (Execution time is less than a clock cycle). Therefore the number of clock cycles consumed to execute each algorithm 10000 times was measured using following code segment.

```
clock_t start,end;
start=clock();
for(i=0;i<10000;i++)
{
        // Code of the algorithm
}
end=clock();
printf("%d",end-start);
```

The number of clock cycles consumed by each algorithm for the same set of polygons in Table 1 is shown in Table 2.

**Table 2:** The Number of Clock Cycles Comparison.

| Polygon | Proposed algorithm | Hertel Mehlhorn algorithm |
|---|---|---|
| 1 | 242 | 27 |
| 2 | 37 | 10 |
| 3 | 141 | 20 |
| 4 | 90 | 19 |
| 5 | 42 | 12 |
| 6 | 54 | 11 |
| 7 | 30 | 8 |
| 8 | 69 | 11 |
| 9 | 48 | 10 |
| 10 | 54 | 10 |
| 11 | 30 | 8 |
| 12 | 19 | 11 |
| 13 | 44 | 9 |
| 14 | 68 | 14 |
| 15 | 127 | 19 |
| 16 | 86 | 19 |
| 17 | 140 | 20 |
| 18 | 50 | 13 |
| 19 | 23 | 6 |
| 20 | 21 | 7 |

The average ratio (Time, Proposed: HM) is computed using following equation.

Average Ratio (Time, Proposed: HM) = (∑ Clock cycles for proposed algorithm)/ (∑ Clock cycles for HM algorithm)

Average ratio (Time, Proposed: HM) = 1415/264 = 5.4

The efficiency usually decreases in the attempt of improving the accuracy of an algorithm. The average ratio conveys that the proposed algorithm takes approximately 5.4 times of the execution time consumed by the Hertel Mehlhorn algorithm.

The proposed algorithm can be implemented using only static memory. The coordinates of the vertices of the polygon are stored using array data structure. The line segments which partition the polygon are also saved using array data structure. Therefore the proposed algorithm uses minimum amount of memory. The Hertel Melhorn algorithm was the algorithm which used minimum amount of memory in literature. However that algorithm needs the polygon triangulated first in order to process. Therefore triangulation result should be stored in memory until partitioning process is over. Therefore the proposed algorithm is better than Hertel and Mehlhorn algorithm in terms of memory consumption as well.

The Greene's algorithm (O'Rourke, 1998) gives the minimum number of convex pieces to a given polygon. However it is implemented using dynamic programming approach which consumes lots of memory due to recursion. There are programming languages which do not support recursion (Wijeweera *et al*. 2016). Further the time complexity of the Greene's algorithm is O ($n^2r^2$) and in the worst case it is O ($n^4$). Therefore Greene's algorithm is not applicable in such situations where memory and efficiency is critical.

The algorithm proposed by Chazelle (O'Rourke, 1998) is capable of producing minimum number of convex pieces with O (n + $r^3$) time complexity. In the worst case, it becomes O ($n^3$) time complexity. However this algorithm is extremely complicated and corresponding implementation is not available in literature.

There are three main functions in the proposed algorithm: *setReflex*, *setSectors1*, and *setSectors2*. Each function has O (n), O ($n^3$), and O ($n^2$) time complexities respectively. Therefore the time complexity of the proposed algorithm is O (n) + O ($n^3$) + O ($n^2$) = O ($n^3$). The Hertel Mehlhorn algorithm has O (n) time complexity if a triangulated polygon is provided as input. However triangulation can be done in O ($n^2$) of minimum time complexity (Wijeweera *et al*., 2016). Therefore Hertel Mehlhorn algorithm needs O ($n^2$) + O (n) = O ($n^2$) time complexity for convex partitioning a polygon.

## CONCLUSION

An algorithm to partition a polygon into smaller number of convex pieces was proposed. The proposed algorithm is able to partition a polygon into lesser number of convex pieces than the Hertel Mehlhorn algorithm. The proposed algorithm is 1.3 times better than the Hertel Mehlhorn algorithm in number of convex pieces. Naturally, the execution time increases with the attempt of trying to minimize the number of convex pieces. Therefore the proposed algorithm is 5.4 times slower than the Hertel Mehlhorn algorithm. The proposed algorithm can be implemented using only static memory. Further it has minimum memory consumption. The time complexity of the proposed algorithm is O ($n^3$). Therefore proposed algorithm is suitable for applications where memory is critical and smaller number of convex pieces is required. That means the proposed algorithm is suitable for embedded systems programming.

## REFERENCES

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

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

Wijeweera, K. R., Kodituwakku S. R. (2016). Accurate, Simple and Efficient Triangulation of a Polygon by Ear Removal, *Ceylon Journal of Science*. **45** (3): 65-76.

Lien, J. M., Amato N. M. (2006). Approximate Convex Decomposition of Polygons, *Parasol Laboratory*. Department of Computer Science, Texas A&M University.

Hongguang, Z., Guozhao, W. (2015). New Framework for Decomposing a Polygon with Zero or More Holes. *The Open Cybernetics and Systemics Journal*. **9**: 390-405.

Kodituwakku, S. R., Wijeweera, K. R. (2014). An Algorithm to Find the Largest Circle inside a Polygon. *Ceylon Journal of Science (Physical Sciences)*. **19**: 45-54.

## APPENDIX

The implementation of the proposed algorithm in C programming language is available from the following link:

https://www.academia.edu/31878764/Convex_Partitioning_of_a_Polygon_into_Smaller_Number_of_Pieces_with_Lowest_Memory_Consumption