Computer Engineering and Intelligent Systems
ISSN 2222-1719 (Paper) ISSN 2222-2863 (Online)
Vol.5, No.4, 2014
www.iiste.org

# Implementation of an Efficient Scan-Line Polygon Fill Algorithm

Dr. Ismail Al-Rawi
Arab Open University (Kuwait Branch), P.O 830 Al Ardia, P.C 92400, Kuwait
*E-Mail: ism_49@hotmail.com

**Abstract**

Area filling or polygon filling is a fundamental operation in computer graphics. There are many methods used to fill a closed shape, but the most common one uses the property of raster scan display to perform the filling. These types of algorithms simply depend on finding the intersection points for each scan-line with polygon edges and fill between alternate pairs of intersection points. The implementation of these operations is a straight forward for simple polygons, like convex and concave, and can be more complicated for complex polygon where regions may overlap. This paper present implementation for an efficient algorithm that manages to fill all kinds of polygons depends on using very simple arrays data structure that can be easily programmed by C or Java language. The implementation in Java verified that the algorithm is theoretically and experimentally superior to many other conventional algorithms implementation.

**Keywords**: Polygon filling, Scan-line filling, Edge-Table, Active-Table, Odd–Even Parity.

## 1. Introduction

A polygon is a shape formed by connecting line segments end to end, creating closed path. Three types of polygon are recognized, figure (1): convex, concave, and complex. Convex polygon can be recognized by: If a straight line connecting any two points inside the polygon should always lies inside the polygon, in other word, the line should not intersect any edges of the polygon; any scan-line may cross at most two edges of the polygon (not counting the horizontal edges). Convex polygon edges do not intersect each other (Folly, 1994).

Concave polygon is the polygon which is not convex, that is mean any line connecting any two points that lie inside the polygon may intersect more than two edges. Thus, more than two edges may intersect any scan-line that pass through the polygon. Polygon edges may also touch each other.

Complex polygon in basic is a concave polygon that may have self-intersecting edges, and the complexity comes from the difficulty of recognizing the inside edges in the filling process (Zhang, 2007).

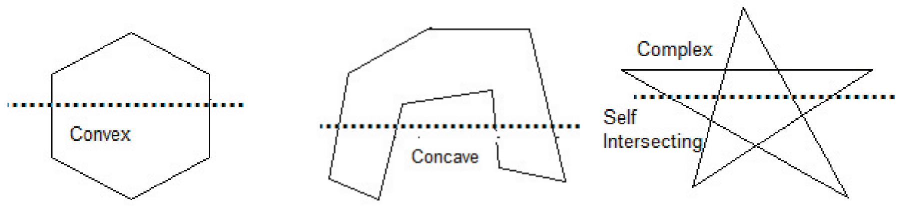


Figure (1)

## 2. Polygon Filling

Unfilled polygon different from filled ones by: only points on the perimeter of the polygon are drawn as shown in all of the above drawing. However when a polygon is filled all the points inside the polygon must be considered and set to the specified color or pattern (Cs, 2011). Therefore to determine which points are inside the polygon is the most important problem in any filling algorithm. In case of convex and concave polygon, the problem is clear, but in case of complex polygon where multiple regions with complicated relationships may be formed, the interior region problem is not always trivial (Sccs, 2002).

Two rules are identified to be used to determine whether a particular region is in the interior of a polygon (Zhang, 2007); the first is the Even-Odd parity rule; by drawing a line through the region and considering the number of crossing from left to right with the border of the region, region is interior if the number of crossing is odd, and exterior if the number is even, consequently the designation of interior and exterior alternate every time the line cross an edge figure (2). The second rule is called nonzero rule where the direction of the path crossing is taken into consideration, and the crossing number can be positive or negative. If the line cross the path from left to right as viewed in the direction of the path, the crossing number is increased by 1; otherwise it is decreased by 1. The region is interior if the signed crossing number is not zero. The nonzero rule defines the left side of the path as the interior and the right side as exterior.

Most algorithms use the parity concept to determine which points or pixels lie within a polygon including this algorithm where the alternate crossing is adopted (Lec, 2009), taking the advantage of the fact that each scan-line initially begin with even parity until it cross the first edge, then parity changed to odd and the filling process started until the next crossing where parity change to even and filling process stop. At the end, for each scan-line, there will be a sorted list of x-intersections, from left to right, with the polygon edges, then simply we can draw from the first x to second x, the third to the forth and so on, in other word, from odd parity to even parity, figure(2).

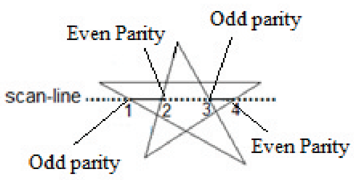


Figure (2)

## 3. The Algorithm

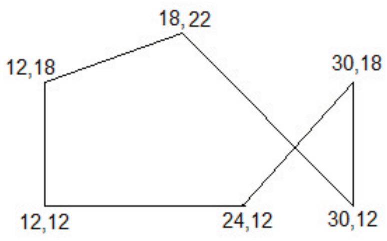
3.1 Data structure:

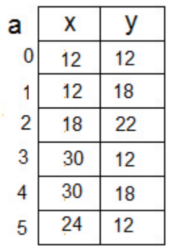
Int [][] a: two dimensional array to store the polygon vertices.
Int [][] b : two dimensional array to store the Edges-Table.
float [][] cc : two dimensional array to store the Active-Table.
float [][] ccc : two dimensional array to store the updated Active-Table.
float [] slop : one dimensional array to store the slop.
double [][] drawline : two dimensional array to store the drawing points.

3.2 Enter the polygon as a set of vertices, indicating the x and y Cartesian coordinate of each vertex in order. This polygon is the one which should be filled, figure (3). These vertices will be stored in two dimensional arrays to represent the Vertices-Table, figure (4).

```
// to get the polygon vertices interactively:
For (int row=0; row=a.length;col++)
      { for (int col=0; col< columns; col++)
        a[row][col]=input.nextint();}
```

Vertex-Table



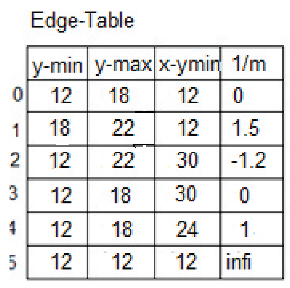


| a | x | y |
|---|---|---|
| 0 | 12 | 12 |
| 1 | 12 | 18 |
| 2 | 18 | 22 |
| 3 | 30 | 12 |
| 4 | 30 | 18 |
| 5 | 24 | 12 |

Figure (3) Figure (4)

3.3 Construct the Edge-Table in the form of the minimum y-value, maximum y-value and the inverse of the slop (1/m) between each two adjacent vertices in the Vertices-Table. At the end the last vertex is associated with the first vertex in the table, figure (5).

```
// maximum method
 public static int FinfMax (int [][]a,int i) {
    if (a[i][1] > a[i+1][1]){
        return a[i][1];}
    else {
        return a[i+1][1];}
                                        }
// minimum method
public static int FinfMin (int [][]a,int [][] b,int i){
    if (a[i][1] < a[i+1][1]){
        b[i][2]=a[i][0];
        return a[i][1];}
    else {
        b[i][2]=a[i+1][0];
        return a[i+1][1];}
                                }
```



Edge-Table

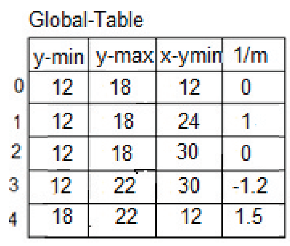
| | y-min | y-max | x-ymin | 1/m |
|---|---|---|---|---|
| 0 | 12 | 18 | 12 | 0 |
| 1 | 18 | 22 | 12 | 1.5 |
| 2 | 12 | 22 | 30 | -1.2 |
| 3 | 12 | 18 | 30 | 0 |
| 4 | 12 | 18 | 24 | 1 |
| 5 | 12 | 12 | 12 | infi |

Figure (5)

To find the inverse of the slop the slop function: 1/m=(x2-x1) / (y2-y1) is used.

```
// slop method
  public static double FinSlop  (int [][]a,float[]slope, int i) {
        int dy=a[i+1][1]-a[i][1];
        int dx=a[i+1][0]-a[i][0];
         if(dy==0) {slope[i]=1;}
         if(dx==0) {slope[i]=0;}
         if((dy!=0)&&(dx!=0)) /*- calculate inverse slope -*/
         slope [i]=  ((float)dx/(float)dy);
         return (float) slope[i];
```

3.4 Produce the Global-Table by sorting the Edge-Table in ascending order with respect to y and x values, as long as slop is not equal to zero, figure(6).



Global-Table

| | y-min | y-max | x-ymin | 1/m |
|---|---|---|---|---|
| 0 | 12 | 18 | 12 | 0 |
| 1 | 12 | 18 | 24 | 1 |
| 2 | 12 | 18 | 30 | 0 |
| 3 | 12 | 22 | 30 | -1.2 |
| 4 | 18 | 22 | 12 | 1.5 |

Figure (6)

Notice that the Global-Table has only five edges, since the last edge in the Edge-Table has a slop of zero.

```
// to generate the Global-Table
     for (int c = 0; c < ( n - 1 ); c++) {
     for (int d = 0; d < (n - c - 1); d++) {
       if (((b[d][0]> b[d+1][0])||(b[d][1]>b[d+1][1]))||
          ((b[d][0]==b[d+1][0])&&(b[d][2]> b[d+1][2])))
          {
        swap = b[d][0];
        b[d][0] = b[d+1][0];
        b[d+1][0] = swap;
        swap = b[d][1];
        b[d][1] = b[d+1][1];
        b[d+1][1] = swap;
        swap = b[d][2];
        b[d][2] = b[d+1][2];
        b[d+1][2] = swap;
        swapsl= slope[d];
        slope[d]=slope[d+1];
        slope[d+1]=swapsl;
      }
    }}
```

In this algorithm, all polygons assumed to be drawn inside the display screen; therefore all scan-lines started from a point outside the polygon and have even parity.

Since the Global-Table designate the lowest and the highest y-value of all vertices, there is no need to scan all the display screen for the points of intersection, only those scan-line between the lowest and the highest y-value is need to be considered. This will save considerable amount of processing time.

3.5 Initiate an Active-Table to keep track of the edges that are intersected by every scan-line. This table is set up by appending the edge information for the y-max, x-value and 1/m for all edges with y-min equal to the first scan-line in the Global-Table, and keeping the information of other edges in the Global-Table, figure (7).

Active-Table

| | Y-max | X-Ymin | 1/m |
|---|---|---|---|
| 0 | 18 | 12 | 0 |
| 1 | 18 | 24 | 1 |
| 2 | 18 | 30 | 0 |
| 3 | 22 | 30 | -1.2 |

Global-Table

| | | | | |
|---|---|---|---|---|
| 0 | 18 | 22 | 12 | 1.5 |

Figure (7)

```
// to generate the Active-Table
    scaline=b[0][0]; \* this represents the first scan-line (the minimum value of y).*\
    for (int ii =0; ii<n+1; ii++){
       if (b[ii][0] == scaline)
       {
         cc[ii][0]=b[ii][1];
         cc[ii][1]=b[ii][2];
         cc[ii][2]=slope[ii];
         nn=ii;
       }
       else
         {
        mascan[0][0]=b[ii][0];
        mascan[0][1]=b[ii][1];
        mascan[0][2]=b[ii][2];
        mascan[0][3]=slope[ii];
         } }
```

## 4. Polygon filling

Filling the polygon is done by drawing lines between intersection points that are indicated between odd and even parity of each scan-line, Figure(8).

The two dimensional array (drawlines [ ][ ]) will keeps in order, the integer values of the coordinates for all the intersection points in order to be drawn in later time.

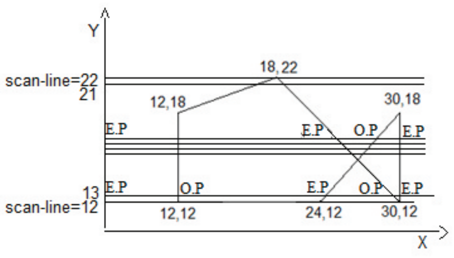
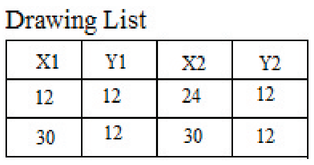


Figure (8)

4.1 For the first scan-line (12), the first edge encountered at x=12, parity=odd. All points are drawn from this point until the next edge is encountered at x=24, parity is then changed to even. The next is met at x=30 and the point is drawn only once. These intersection points will be stored in the drawing list as in figure (9).

```
 // to produce the drawing list
      int l=0;
// from first scan-line to the last scan-line
      for (int scan=min-y; scan<Max-y; scan++){
         if (cc[0][0]!=scan){
          for (int i=0;i<n-1;i+=2){
            drwline[l][0]= Math.ceil(cc[i][1]);
            drwline[l][1]=scan;
            drwline[l][2]=Math.ceil(cc[i+1][1])
            drwline[l][3]=scan;
            l=l+1; }}}
```



Drawing List

| X1 | Y1 | X2 | Y2 |
|---|---|---|---|
| 12 | 12 | 24 | 12 |
| 30 | 12 | 30 | 12 |

Figure(9)

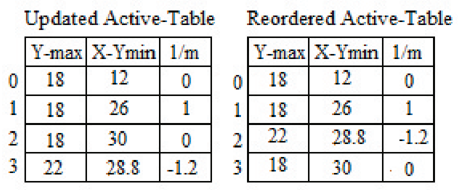
4.2 Before the next scan-line is considered, the active-Table must be:

a. Updated by using the formula $x_1 = x_0 + 1/m$.

b. Reordered if necessary, to keep the x value of y-min in ascending order.

After these operations the Active-Table becomes as in the figure (10).

```
// to update the Active-Table
   public static float[][] acupdate (float [][]cc, int n) {
            for (int i=0;i<n;i++) {
               cc[i][1]=cc[i][1]+cc[i][2];
                           }
               return cc; }
```



Updated Active-Table

| | Y-max | X-Ymin | 1/m |
|---|---|---|---|
| 0 | 18 | 12 | 0 |
| 1 | 18 | 26 | 1 |
| 2 | 18 | 30 | 0 |
| 3 | 22 | 28.8 | -1.2 |

Reordered Active-Table

| | Y-max | X-Ymin | 1/m |
|---|---|---|---|
| 0 | 18 | 12 | 0 |
| 1 | 18 | 26 | 1 |
| 2 | 22 | 28.8 | -1.2 |
| 3 | 18 | 30 | 0 |

Figure (10)

```
// to reorder the Active-Table
  public static float [][] acsort (float [][]cc, int n) {
  for (int i=0; i< n-1; i++){
     if (cc[i][1] > cc[i+1][1]){
        float swap=cc[i][0];
        cc[i][0]=cc[i+1][0];
        cc[i+1][0]=swap;
        swap=cc[i][1];
        cc[i][1]=cc[i+1][1];
        cc[i+1][1]=swap;
        swap=cc[i][2];
        cc[i][2]=cc[i+1][2];
        cc[i+1][2]=swap;
                       } }
        return cc;   }
```

4.3 process 1 and 2 is repeated until scan-line 17 is reached, where y-max is equal to the next scan-line (18) for the edges at indices 0,2 and 3, these edges is removed from the Active-Table leaving it as shown in the figure(11).

Active-Table

| | Y-max | X-Ymin | 1/m |
|---|---|---|---|
| 0 | 22 | 24 | -1.2 |

Global-Table

| | | | | |
|---|---|---|---|---|
| 0 | 18 | 22 | 12 | 1.5 |

Figure (11)

Now the Active-Table need to be updated, then last edge from the Global-Table can be add to the Active-Table to become as in figure (12).

Active-Table

| | Y-max | X-Ymin | 1/m |
|---|---|---|---|
| 0 | 22 | 22.8 | -1.2 |
| 1 | 22 | 12 | 1.5 |

Figure (12)

After reordering, the Active-Table it becomes as in the Figure (13) and the drawing list as in figure (14); the algorithm continue to find the drawing points for the rest of the scan lines until the Active-Table becomes as in the figure(15), where the y-max value for both edges is equal to the next scan line (22). Then these edges are removed leaving the Active-Table empty and the drawing list as in figure (16). It's noticed from the drawing list that the last scan-line (22) is not considered, because it touches two edges at a meeting vertex with maximum y-value.

By this the whole operations of the algorithm are done.

Active-Table

| | Y-max | X-Ymin | 1/m |
|---|---|---|---|
| 0 | 22 | 12 | 1.5 |
| 1 | 22 | 22.8 | -1.2 |

Figure (13)

Drawing List

| X1 | Y1 | X2 | Y2 |
|---|---|---|---|
| 12 | 12 | 24 | 12 |
| 30 | 12 | 30 | 12 |
| -- | -- | -- | -- |
| -- | -- | -- | -- |
| 12 | 17 | 24 | 17 |
| 29 | 17 | 30 | 17 |
| 12 | 18 | 23 | 18 |

Figure (14)

Active-Table

| Y-max | X-Ymin | 1/m |
|---|---|---|
| 22 | 16.5 | 1.5 |
| 22 | 19.2 | -1.2 |

Figure (15)

Drawing List

| X1 | Y1 | X2 | Y2 |
|---|---|---|---|
| 12 | 12 | 24 | 12 |
| 30 | 12 | 30 | 12 |
| -- | -- | -- | -- |
| -- | -- | -- | -- |
| 12 | 17 | 24 | 17 |
| 29 | 17 | 30 | 17 |
| 12 | 18 | 23 | 18 |
| -- | -- | -- | -- |
| 17 | 21 | 20 | 21 |

Figure (16)

## Conclusions

One of the main reasons that make the implementation of this algorithm is successful; is the way of treating the special cases: first, horizontal edges are removed from the Edge-Table completely; second the algorithm does not allow to fills the right or top edges of the polygon; third, when two edges meet at a vertex and for both edges the vertex is the minimum point, the pixel is drawn once and is counted twice for parity; forth, when two edges meet at a vertex and for one edge the value is the maximum point and for the other edge the value is the minimum point, these edges considered as one bent edge, therefore if the edges are on the left side of the polygon, the pixel is drawn and is counted once for parity, if both edges are on the right , the pixel is not drawn and is counted just once for parity.

This algorithm does not cover polygon with boundary defined by curves such as, circle, ellipses, parabolic or spline. These types of polygons will be the target of the next research by approximating these curves by a series of small straight lines (edges). Filling with different patterns can be also addressed in the next research paper.

## References

Foley, James D., et. al., 1994: Introduction to Computer Graphics. Reading: Addison-Wesley Publishing Company, 1994.

Hong Zhang, Y. Daniel Liang, 2007: Computer Graphics using Java 2D and 3D: Pearson Prentice Hall, Pearson Education, Inc.

CS, 2011: Polygon Fill Teaching Tool.
http://www.cs.rit.edu/~icss571/filling/

Sccs, 2002, Scanline Fill Algorithm,
http://www.sccs.swarthmore.edu/users/02/jill/graphics/hw3/hw3.html

Lec, 2009: Scanline Polygon Fill Algorithm.
http://ebookbrowsee.net/lect11-2009-areafill-transformations-pdf-d17283659.