Introduction to Computer Graphics

(Lecture No 08)

Filled-Area Primitives-I

So far we have covered some output primitives that is drawing primitives like point, line, circle, ellipse and some other variations of curves. Also we can draw certain other shapes with the combinations of lines like triangle, rectangle, square and other polygons (we will have some discussion on polygons coming ahead). Also we can draw some shapes using mixture of lines and curves or circles etc. So we are able to draw outline/ sketch of certain models but need is there to make a solid model.

Therefore, in this section we will see what are filled area primitives and what are the different issues related to them. There are two basic approaches to area filling on raster systems. One way is to draw straight lines between the edges of polygon called **scan-line polygon filling**. As said earlier there are several issues related to scan line polygon, which we will discuss in detail. Second way is to start from an interior point and paint outward from this point till we reach the boundary called **boundary-fill**. A slight variation of this technique is used to fill an area specified by cluster (having no specific boundary). The technique is called **flood-fill** and having almost same strategy that is to start from an interior point and start painting outward from this point till the end of cluster.

Now having an idea we will try to see each of these one by one, starting from scan-line polygon filling.

8.1 Scan-line Polygon Fill

Before we actually start discussion on scan-line polygon filling technique, it is useful to discuss what is polygon? Besides polygon definition we will discuss the following topics one by one to have a good understanding of the concept and its implementation.

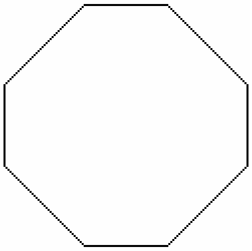
- Polygon Definition
- Filled vs. Unfilled Polygons
- Parity Definition
- Scan-Line Polygon Fill Algorithm
- Special Cases Handled By the Fill
- Polygon Fill Example

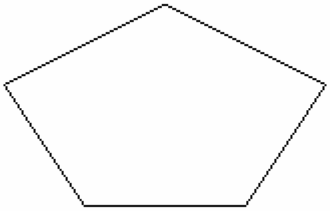
a) Polygon

A **polygon** can be defined as a shape that is formed by line segments that are placed end to end, creating a continuous closed path. Polygons can be divided into three basic types: **convex**, **concave**, and **complex**.

I. **Convex** polygons are the simplest type of polygon to fill. To determine whether or not a polygon is convex, ask the following question:

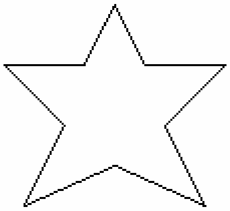
Does a straight line connecting ANY two points that are inside the polygon intersect any edges of the polygon?

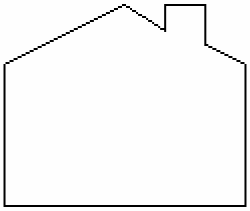




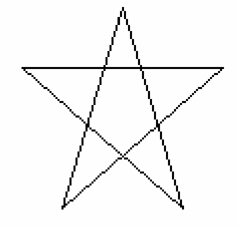
If the answer is no, the polygon is convex. This means that for any scan-line, the scan-line will cross at most two polygon edges (not counting any horizontal edges). Convex polygon edges also do not intersect each other.

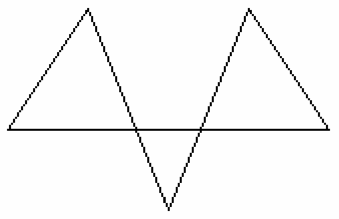
II. **Concave** polygons are a superset of convex polygons, having fewer restrictions than convex polygons. The line connecting any two points that lie inside the polygon may intersect more than two edges of the polygon. Thus, more than two edges may intersect any scan line that passes through the polygon. The polygon edges may also touch each other, but they may not cross one another.





Complex polygons are just what their name suggests: complex. Complex polygons are basically concave polygons that may have self-intersecting edges. The complexity arises from distinguishing which side is inside the polygon when filling it.

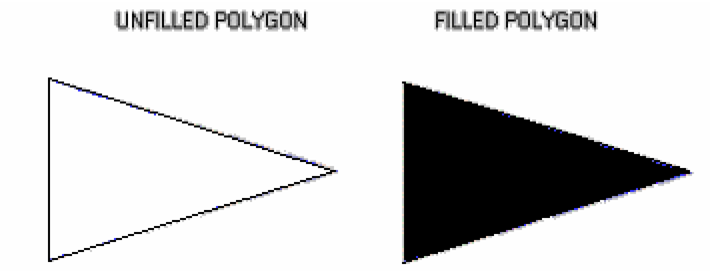




Difference between Filled and Unfilled Polygon

When an unfilled polygon is rendered, only the points on the perimeter of the polygon are drawn. Examples of unfilled polygons are shown in the next page.

However, when a polygon is filled, the interior of the polygon must be considered. All of the pixels within the boundaries of the polygon must be set to the specified color or pattern. Here, we deal only with solid colors. The following figure shows the difference between filled and unfilled polygons.

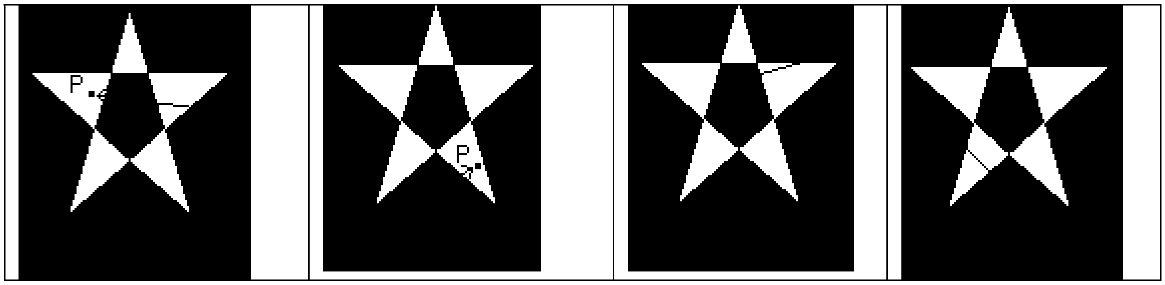


In order to determine which pixels are inside the polygon, the odd-parity rule is used within the scan-line polygon fill algorithm. This is discussed next.

b) Parity

What is parity? **Parity** is a concept used to determine which pixels lie within a polygon, i.e. which pixels should be filled for a given polygon.

The Underlying Principle: Conceptually, the odd parity test entails drawing a line segment from any point that lies outside the polygon to a point P, that we wish to determine whether it is inside or outside of the polygon. Count the number of edges that the line crosses. If the number of polygon edges crossed is **odd**, then P lies within the polygon. Similarly, if the number of edges is even, then P lies outside of the polygon. There are special ways of counting the edges when the line crosses a vertex. This will be discussed in the algorithm section. Examples of counting parity can be seen in the following demonstration.



Using the Odd Parity Test in the Polygon Fill Algorithm

The odd parity method creates a problem: How do we determine whether a pixel lies outside of the polygon to test for an inside one, if we cannot determine whether one lies within or outside of the polygon in the first place? If we assume our polygon lies entirely within our scene, then the edge of our drawing surface lies outside of the polygon.

Furthermore, it would not be very efficient to check each point on our drawing surface to see if it lies within the polygon and, therefore, needs to be colored.

So, we can take advantage of the fact that for each scan-line we begin with even parity; we have NOT crossed any polygon edges yet. Then as we go from left to right across our scan line, we will continue to have even parity (i.e., will not use the fill color) until we cross the first polygon edge. Now our parity has changed to odd and we will start using the fill color.

How long will we continue to use the fill color? Well, our parity won't change until we cross the next edge. Therefore, we want to color all of the pixels from when we crossed the first edge until we cross the next one. Then the parity will become even again.

So, you can see if we have a sorted list of x-intersections of all of the polygon edges with the scan line, we can simply draw from the first x to the second, the third to the forth and so on.

c) Polygon Filling

In order to fill a polygon, we do not want to have to determine the type of polygon that we are filling. The easiest way to avoid this situation is to use an algorithm that works for all three types of polygons. Since both convex and concave polygons are subsets of the complex type, using an algorithm that will work for complex polygon filling should be sufficient for all three types. The scan-line polygon fill algorithm, which employs the odd/even parity concept previously discussed, works for complex polygon filling.

Reminder: The basic concept of the scan-line algorithm is to draw points from edges of odd parity to even parity on each scan-line.

d) What is a scan-line?

A scan-line is a line of constant y value, i.e., y=c, where c lies within our drawing region, e.g., the window on our computer screen.

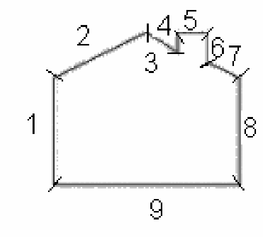
The **scan-line algorithm** is outlined next.

8.2 Algorithm

When filling a polygon, you will most likely just have a set of vertices, indicating the x and y Cartesian coordinates of each vertex of the polygon. The following steps should be taken to turn your set of vertices into a filled polygon.

1. **Initializing All of the Edges:**

The first thing that needs to be done is determine how the polygon's vertices are related. The all_edges table will hold this information.



Each adjacent set of vertices (the first and second, second and third, similarly last and first) defines an edge. In above figure vertices are shown by small lines and edges are numbered from 1 to 9 each between successive vertices.

For each edge, the following information needs to be kept in a table:

1. The minimum y value of the two vertices
2. The maximum y value of the two vertices
3. The x value associated with the minimum y value
4. The slope of the edge

The slope of the edge can be calculated from the formula for a line:

$y = mx + b;$

where m = slope, b = y-intercept,

y_0 = maximum y value,

y_1 = minimum y value,

x_0 = maximum x value,

x_1 = minimum x value The formula for the slope is as follows:

$m = (y_0 - y_1) / (x_0 - x_1).$

For example, the edge values may be kept as follows, where N is equal to the total number of edges - 1 (starting from 0) and each index into the all_edges array contains a pointer to the array of edge values.

Index		Y-min	Y-max	X-val	1/m
0	→	10	16	10	0
1	→	16	20	10	1.5
		-	-	-	-
		-	-	-	-
N	→	10	16	28	0

Table: All_edges

2. Initializing the Global Edge Table:

The global edge table will be used to keep track of the edges that are still needed to complete the polygon. Since we will fill the edges from bottom to top and left to right. To do this, the global edge table should be inserted with edges grouped by increasing minimum y values. Edges with the same minimum y values are sorted on minimum x values as follows:

1. Place the first edge with a slope that is not equal to zero in the global edge table.
2. If the slope of the edge is zero, do not add that edge to the global edge table.
3. For every other edge, start at index 0 and increase the index of the global edge table once each time the current edge's y value is greater than that of the edge at the current index in the global edge table.

Next, Increase the index to the global edge table once each time the current edge's x value is greater than and the y value is less than or equal to that of the edge at the current index in the global edge table.

If the index, at any time, is equal to the number of edges currently in the global edge table, do not increase the index.

Place the edge information for minimum y value, maximum y value, x value, and 1/m in the global edge table at the index.

The global edge table should now contain all of the edge information necessary to fill the polygon in order of increasing minimum y and x values.

3. Initializing Parity

The initial parity is even since no edges have been crossed yet.

4. Initializing the Scan-Line

The initial scan-line is equal to the lowest y value for all of the global edges. Since the global edge table is sorted, the scan-line is the minimum y value of the first entry in this table.

5. Initializing the Active Edge Table

The active edge table will be used to keep track of the edges that are intersected by the current scan-line. This should also contain ordered edges. This is initially set up as follows:

Since the global edge table is ordered on minimum y and x values, search, in order, through the global edge table and, for each edge found having a minimum y value equal to the current scan-line, append the edge information for the maximum y value, x value, and 1/m to the active edge table. Do this until an edge is found with a minimum y value greater than the scan line value. The active edge table will now contain ordered edges of those edges that are being filled as such:

Index	**Y-max**	**X-val**	**1/m**
0	16	10	0
1	20	10	1.5
	-	-	-
	-	-	-
N	16	28	0

Active

6. Filling the Polygon

Filling the polygon involves deciding whether or not to draw pixels, adding to and removing edges from the active edge table, and updating x values for the next scan-line.

Starting with the initial scan-line, until the active edge table is empty, do the following:

1. Draw all pixels from the x value of odd to the x value of even parity edge pairs.
2. Increase the scan-line by 1.
3. Remove any edges from the active edge table for which the maximum y value is equal to the scan line.
4. Update the x value for each edge in the active edge table using the formula $x_1 = x_0 + 1/m$. (This is based on the line formula and the fact that the next scan-line equals the old scan-line plus one.)
5. Remove any edges from the global edge table for which the minimum y value is equal to the scan-line and place them in the active edge table.

6. Reorder the edges in the active edge table according to increasing x value. This is done in case edges have crossed.

Special Cases

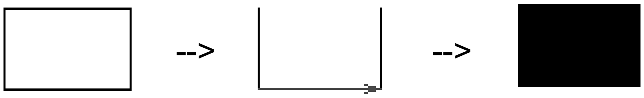
There are some special cases, the scan-line polygon fill algorithm covers these cases, but you may not understand how or why. The following will explain the handling of special cases to the algorithm.

1. **Horizontal Edges**:

Here we follow the minimum y value rule during scan-line polygon fill. If the edge is at the minimum y value for all edges, it is drawn. Otherwise, if the edge is at the maximum y value for any edge, we do not draw it. (See the next section containing information about top vs. bottom edges.)

This is easily done in the scan-line polygon fill implementation. Horizontal edges are removed from the edge table completely.

Question arises that how are horizontal lines are filled then? Since each horizontal line meets exactly two other edge end-points on the scan-line, the algorithm will allow a fill of the pixels between those two end-point vertices when filling on the scan-line which the horizontal line is on, if it meets the top vs. bottom edge criteria.

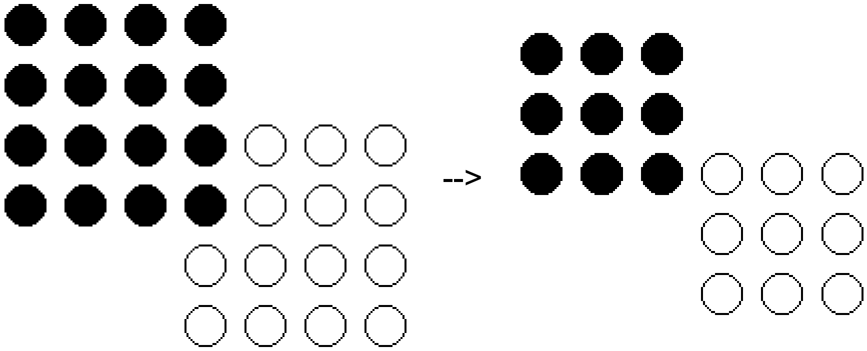


As can be seen above, if we start with a polygon with horizontal edges, we can remove the horizontal edges from the global edge table. The two endpoints of the edge will still exist and a line will be drawn between the lower edges following the scan-line polygon fill algorithm. (The blue arrowed line is indicating the scan-line for the bottom horizontal edge.)

2. **Bottom and Left Edges vs. Top and Right Edges:**

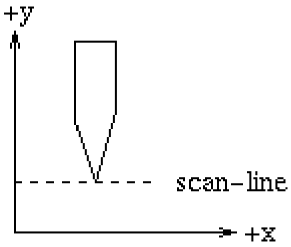
If polygons, having at least one overlapping edge the other, were filled completely from edge to edge, these polygons would appear to overlap and/or be distorted. This would be especially noticeable for polygons in which edges have limited space between them.

In order to correct for this phenomenon, our algorithm does not allow fills of the right or top edges of polygons. This distortion problem could also be corrected by not drawing either the left or right edges and not drawing either the top or bottom edges of the polygon. Either way, a consistent method should be used with all polygons. If some polygons are filled with the left and bottom edges and others with the bottom and right edges, this problem will still occur.



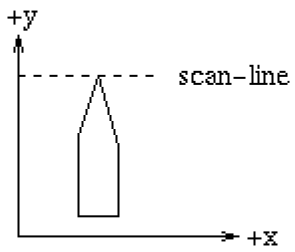
As can be seen above, if we remove the right and top edges from both polygons, the polygons no longer appear to be different shapes. For polygons with more overlap than just one edge, the polygons will still appear to overlap as was meant to happen.

3. How do we deal with two edges meeting at a vertex when counting parity? This is a scenario which needs to be accounted for in one of the following ways:

   1. 

When dealing with two edges; which meet at a vertex and for both edges the vertex is the minimum point, the pixel is **drawn** and is **counted twice for parity**.

Essentially, the following occurs. In the scan-line polygon fill algorithm, the vertex is drawn for the first edge, since it is a minimum value for that edge, but not for the second edge, since it is a right edge and right edges are not drawn in the scan-line fill algorithm. The parity is increased once for the first edge and again for the second edge.

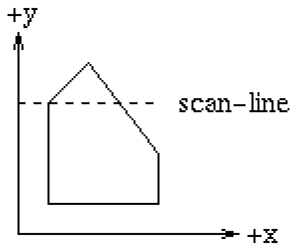


2. ____________________

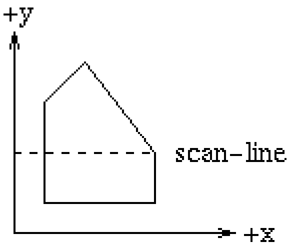
When dealing with two edges; which meet at a vertex and for both edges the vertex is the maximum point, the pixel is **not drawn** and is **counted twice for parity**.

Basically, this occurs because the vertex is not drawn for the first edge, since it is a maximum point for that edge, and parity is increased. The vertex is then not drawn for the second edge, since it is a right edge, and parity is The point should not be drawn since maximum y values for edges are not drawn in the scan-line polygon fill implementation.

3. When dealing with two edges; which meet at a vertex and for one edge the vertex is the maximum point and for the other edge the vertex is the minimum point, we must also consider whether the edges are left or right edges. Two edges meeting in such a way can be thought of as one edge; which is "bent".



If the edges are on the **left** side of the polygon, the pixel is **drawn** and is **counted once for parity** purposes. This is due to the fact that left edges are drawn in the scan-line polygon fill implementation. The vertex is drawn just once for the edge; which has this vertex as its minimum point. Parity is incremented just once for this "bent edge".

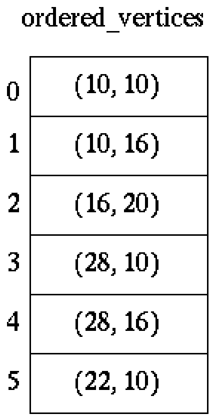


4.

If both edges are on the **right**, the pixel is **not drawn** and is **counted just once for parity** purposes. This is due to the fact that right edges are not drawn in the scan-line polygon fill implementation.

8.3 A Simple Example

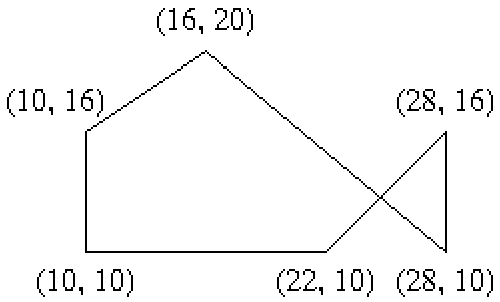
Just to reiterate the algorithm, the following simple example of scan-line polygon filling will be outlined. Initially, each vertices of the polygon is given in the form of (x,y) and is in an ordered array as such:



ordered_vertices

0	(10, 10)
1	(10, 16)
2	(16, 20)
3	(28, 10)
4	(28, 16)
5	(22, 10)

Unfilled, the polygon would look like this to the human eye:



We will now walk through the steps of the algorithm to fill in the polygon.

1. **Initializing All of the Edges:**

We want to determine the minimum y value, maximum y value, x value, and 1/m for each edge and keep them in the all_edges table. We determine these values for the first edge as follows:

Y-min:

Since the first edge consists of the first and second vertex in the array, we use the y values of those vertices to choose the lesser y value. In this case it is 10.

Y-max:

In the first edge, the greatest y value is 16.

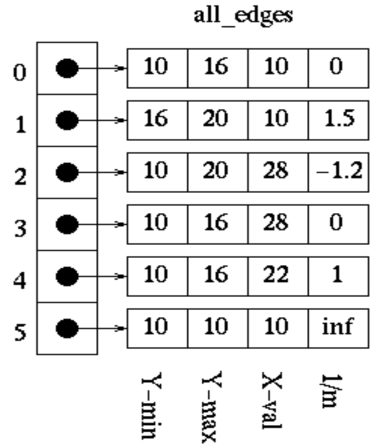
X-val:

Since the x value associated with the vertex with the highest y value is 10, 10 is the x value for this edge.

1/m:

Using the given formula, we get (10-10)/(16-10) for 1/m.

The edge value results are in the form of Y-min, Y-max, X-val, Slope for each edge array pointed to in the all_edges table. As a result of calculating all edge values, we get the following in the all_edges table.

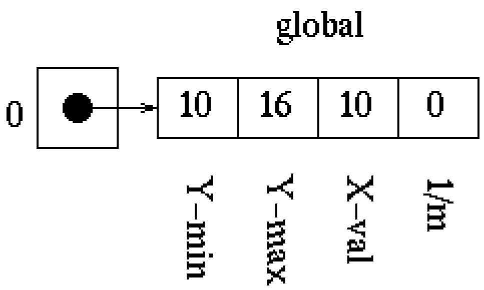
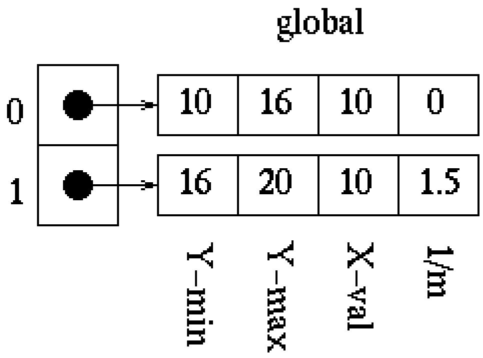
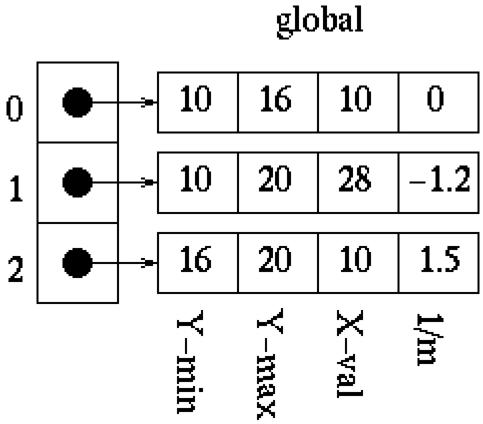


all_edges

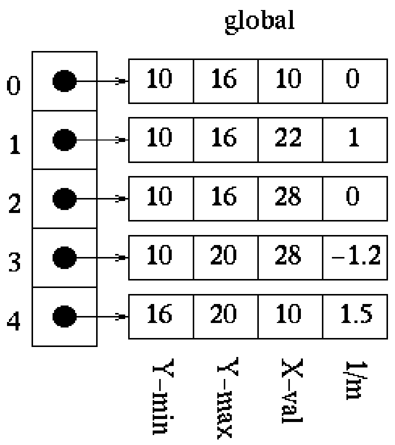
	Y-min	Y-max	X-val	1/m
0	10	16	10	0
1	16	20	10	1.5
2	10	20	28	−1.2
3	10	16	28	0
4	10	16	22	1
5	10	10	10	inf

2. **Initializing the Global Edge Table:**

We want to place all the edges in the global edge table in increasing y and x values, as long as slope is not equal to zero.

For the first edge, the slope is not zero so it is placed in the global edge table at index=0.	 global 0 10 16 10 0 Y–min Y–max X–val 1/m
For the second edge, the slope is not zero and the minimum y value is greater than that at zero, so it is placed in the global edge table at index=1.	 global 0 1 10 16 10 0 16 20 10 1.5 Y–min Y–max X–val 1/m
For the third edge, the slope is not zero and the minimum y value is equal the edge's at index zero and the x value is greater than that at index 0, so the index is increased to 1. Since the third edge has a lesser minimum y value than the edge at index 2 of the global edge table, the index for the third edge is not increased againg. The third edge is placed in the global edge table at index=1.	 global 0 1 2 10 16 10 0 10 20 28 –1.2 16 20 10 1.5 Y–min Y–max X–val 1/m

We continue this process until we have the following:



	Y-min	Y-max	X-val	1/m
0	10	16	10	0
1	10	16	22	1
2	10	16	28	0
3	10	20	28	-1.2
4	16	20	10	1.5

Notice that the global edge table has only five edges and the all_edges table has six. This is due to the fact that the last edge has a slope of zero and, therefore, is not placed in the global edge table.

3. **Initializing Parity**

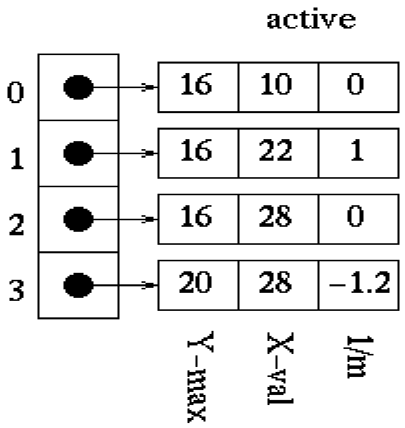
   Parity is initially set to even.

4. **Initializing the Scan-Line**

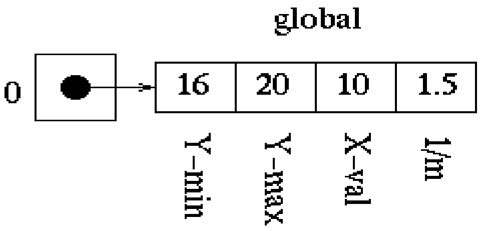
   Since the lowest y value in the global edge table is 10, we can safely choose 10 as our initial scan-line.

5. **Initializing the Active Edge Table**

   Since our scan-line value is 10, we choose all edges which have a minimum y value of 10 to move to our active edge table. This results in the following.



	Y-max	X-val	1/m
0	16	10	0
1	16	22	1
2	16	28	0
3	20	28	-1.2



	Y-min	Y-max	X-val	1/m
0	16	20	10	1.5

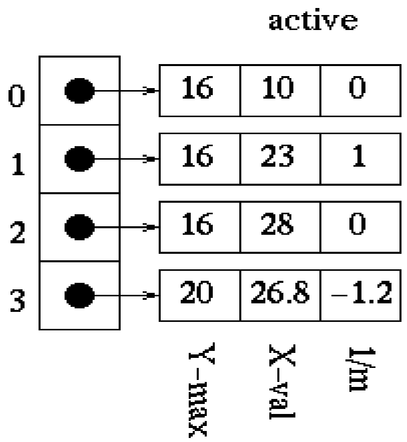
6. Filling the Polygon

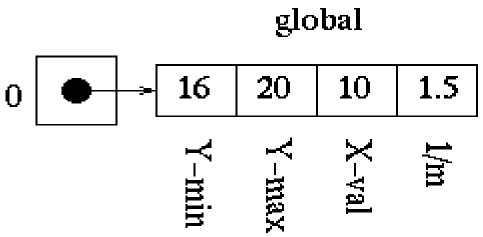
Starting at the point (0,10), which is on our scan-line and outside of the polygon, will want to decide which points to draw for each scan-line.

1. Scan-line = 10:

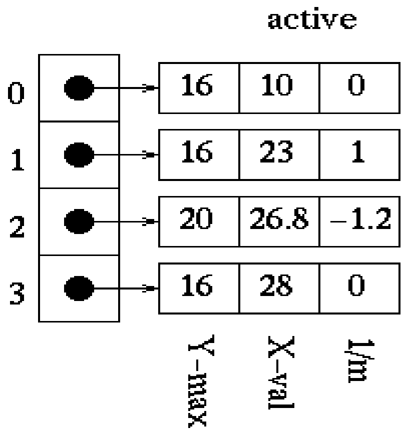
   Once the first edge is encountered at x=10, parity = odd. All points are drawn from this point until the next edge is encountered at x=22. Parity is then changed to even. The next edge is reached at x=28, and the point is drawn once on this scan-line due to the special parity case. We are now done with this scan-line.

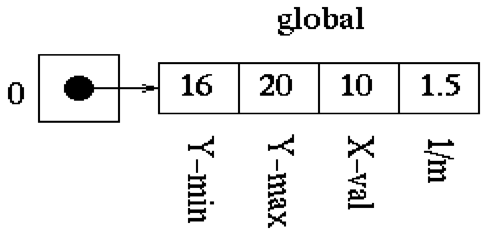
   First, we update the x values in the active edge table using the formula $x_1 = x_0 + 1/m$ to get the following:



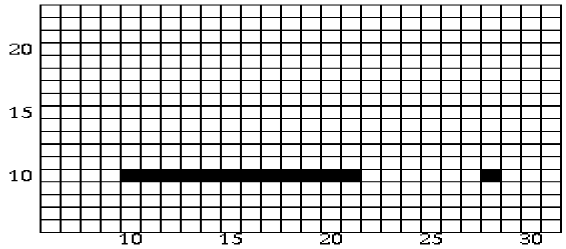


The edges then need to be reordered since the edge at index 3 of the active edge table has a lesser x value than that of the edge at index 2. Upon reordering, we get:





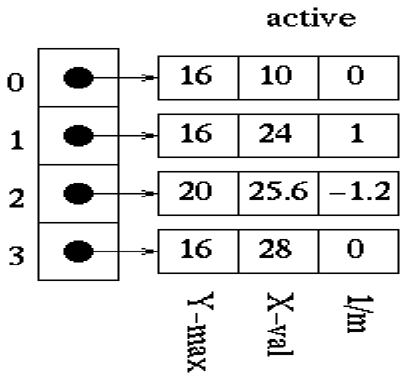
The polygon is now filled as follows:

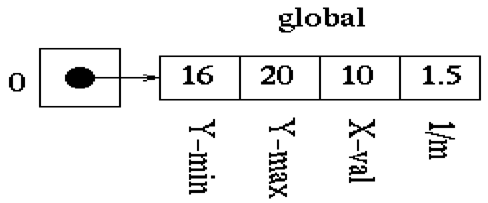


2. Scan-line = 11:

   Once the first edge is encountered at x=10, parity = odd. All points are drawn from this point until the next edge is encountered at x=23. Parity is then changed to even. The next edge is reached at x=27 and parity is changed to odd. The points are then drawn until the next edge is reached at x=28. We are now done with this scan-line.

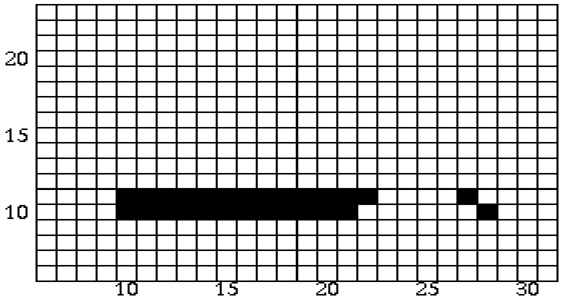
   Upon updating the x values, the edge tables are as follows:





It can be seen that no reordering of edges is needed at this time.

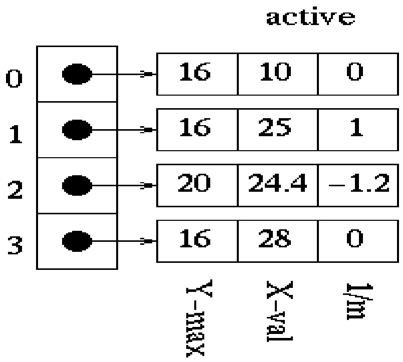
The polygon is now filled as follows:



3. Scan-line = 12:

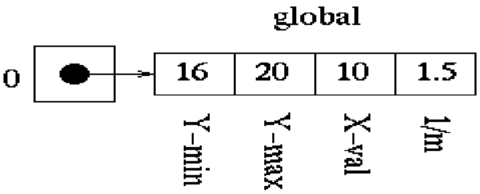
   Once the first edge is encountered at x=10, parity = odd. All points are drawn from this point until the next edge is encountered at x=24. Parity is then changed to even. The next edge is reached at x=26 and parity is changed to odd. The points are then drawn until the next edge is reached at x=28. We are now done with this scan-line.

   Updating the x values in the active edge table gives us:



active

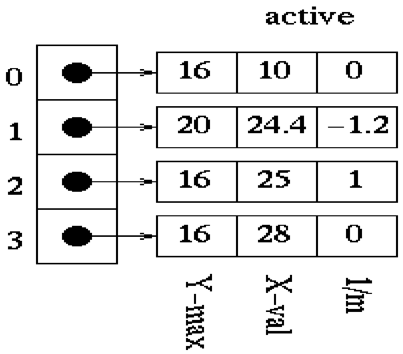
	Y-max	X-val	1/m
0	16	10	0
1	16	25	1
2	20	24.4	−1.2
3	16	28	0

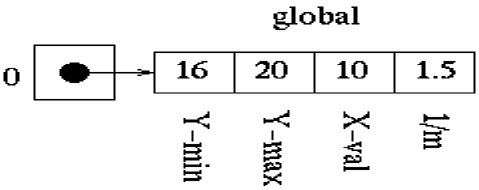


global

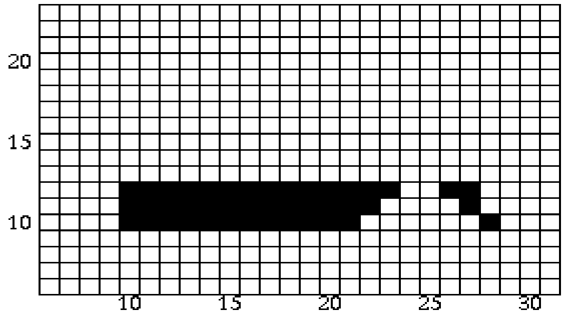
	Y-min	Y-max	X-val	1/m
0	16	20	10	1.5

We can see that the active edges need to be reordered since the x value of 24.4 at index 2 is less than the x value of 25 at index 1. Reording produces the following:





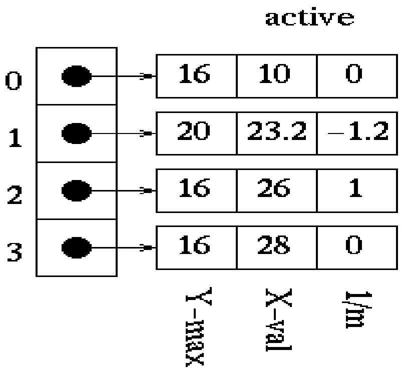
The polygon is now filled as follows:

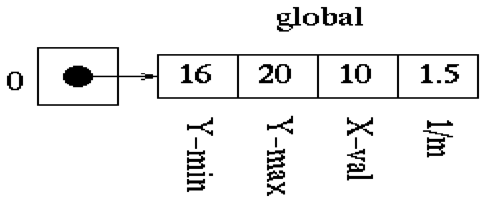


4. Scan-line = 13:

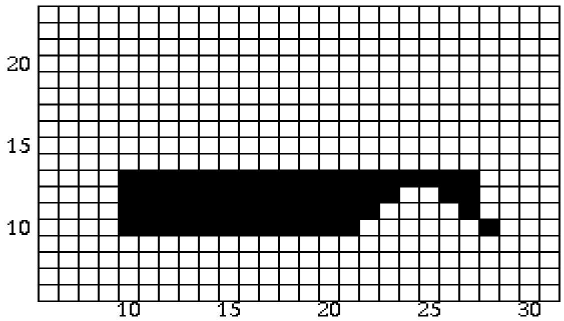
Once the first edge is encountered at x=10, parity = odd. All points are drawn from this point until the next edge is encountered at x=25 Parity is then changed to even. The next edge is reached at x=25 and parity is changed to odd. The points are then drawn until the next edge is reached at x=28. We are now done with this scan-line.

Upon updating the x values for the active edge table, we can see that the edges do not need to be reordered.





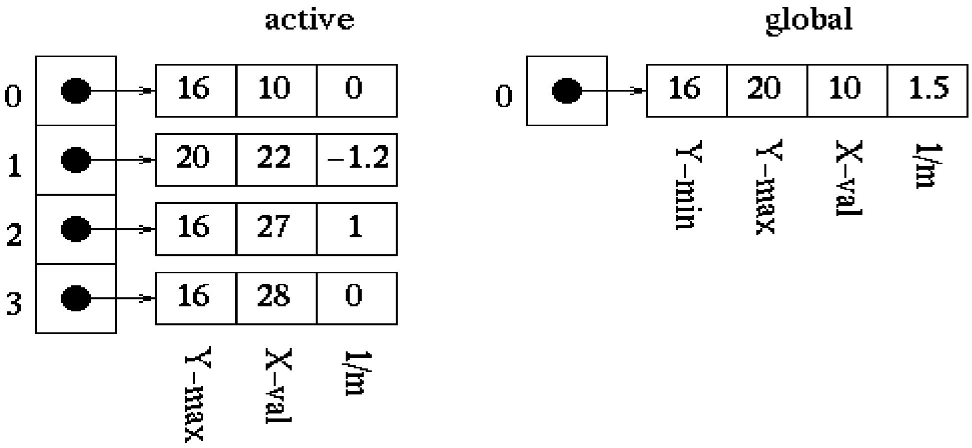
The polygon is now filled as follows:



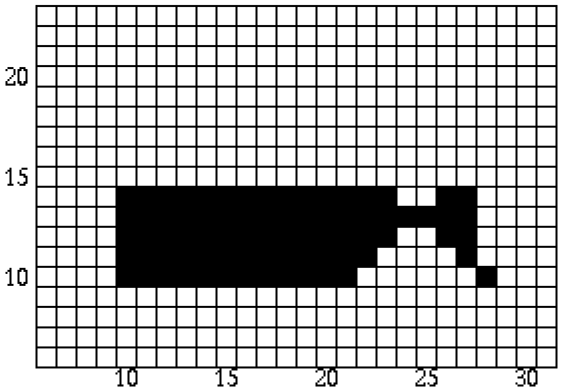
5. Scan-line = 14:

Once the first edge is encountered at x=10, 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 edge is reached at x=26 and parity is changed to odd. The points are then drawn until the next edge is reached at x=28. We are now done with this scan-line.

Upon updating the x values for the active edge table, we can see that the edges still do not need to be reordered.



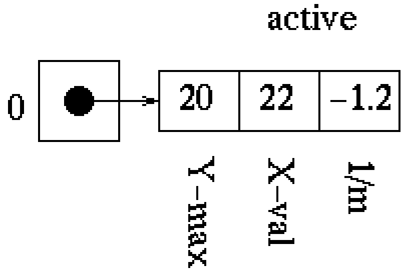
The polygon is now filled as follows:

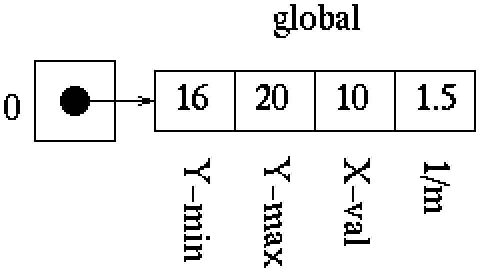


6. Scan-line = 15:

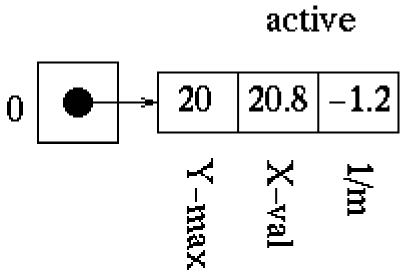
Once the first edge is encountered at x=10, parity = odd. All points are drawn from this point until the next edge is encountered at x=22. Parity is then changed to even. The next edge is reached at x=27 and parity is changed to odd. The points are then drawn until the next edge is reached at x=28. We are now done with this scan-line.

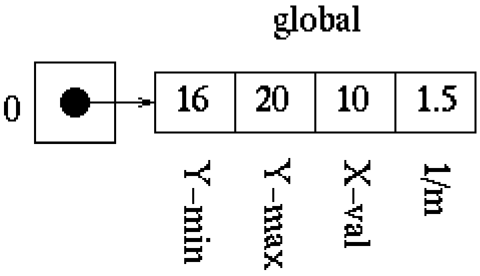
Since the maximum y value is equal to the next scan-line for the edges at indices 0, 2, and 3, we remove them from the active edge table. This leaves us with the following:



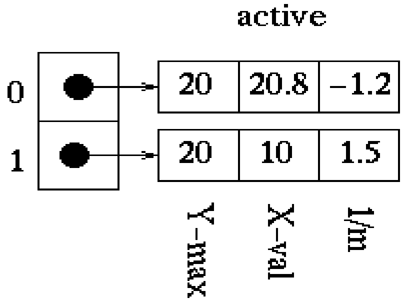


We then need to update the x values for all remaining edges.

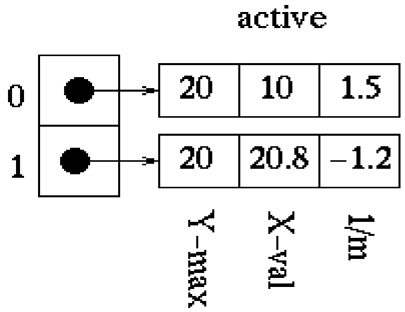




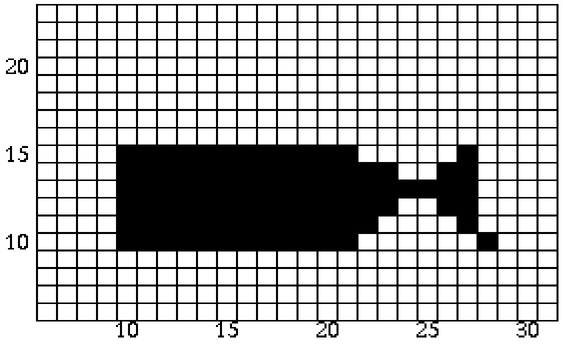
Now we can add the last edge from the global edge table to the active edge table since its minimum y value is equal to the next scan-line. The active edge table now look as follows (the global edge table is now empty):



These edges obviously need to be reordered. After reordering, the active edge table contains the following:

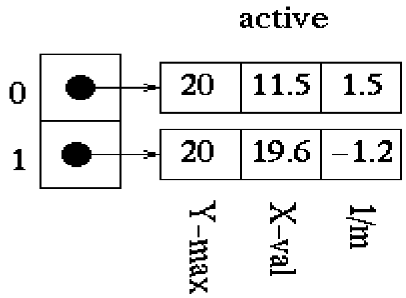


The polygon is now filled as follows:

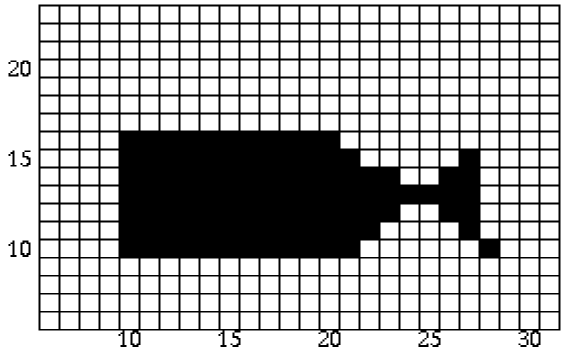


7. Scan-line = 16:

Once the first edge is encountered at x=10, parity = odd. All points are drawn from this point until the next edge is reached at x=21. We are now done with this scan-line. The x values are updated and the following is obtained:

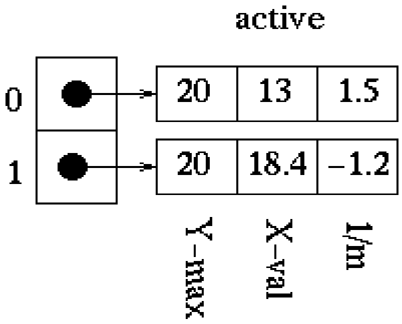


The polygon is now filled as follows:

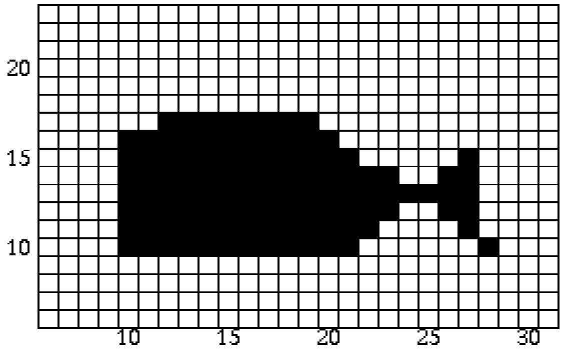


8. Scan-line = 17:

Once the first edge is encountered at x=12, parity = odd. All points are drawn from this point until the next edge is reached at x=20. We are now done with this scan-line. We update the x values and obtain:

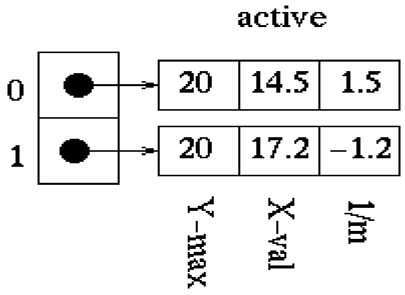


The polygon is now filled as follows:

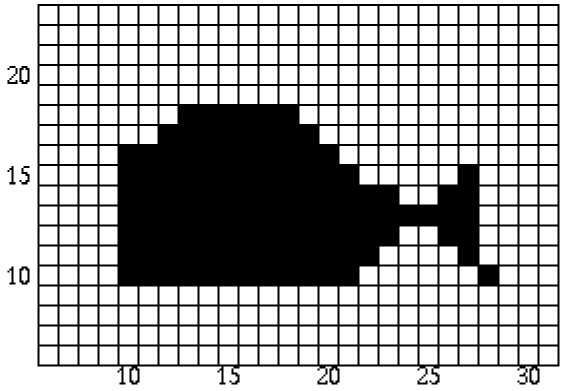


9. Scan-line = 18:

Once the first edge is encountered at x=13, parity = odd. All points are drawn from this point until the next edge is reached at x=19. We are now done with this scan-line. Upon updating the x values we get:



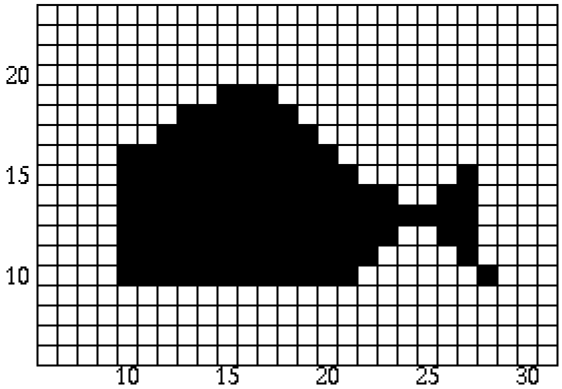
The polygon is now filled as follows:



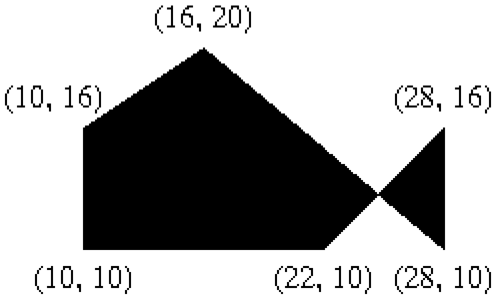
10. Scan-line = 19:

Once the first edge is encountered at x=15, parity = odd. All points are drawn from this point until the next edge is reached at x=18. We are now done with this scan-line. Since the maximum y value for both edges in the active edge table is equal to the next scan-line, we remove them. The active edge table is now empty and we are now done.

The polygon is now filled as follows:



Now that we have filled the polygon, let's see what it looks like to the naked eye:


(16, 20)
(10, 16)
(28, 16)
(10, 10)
(22, 10)
(28, 10)