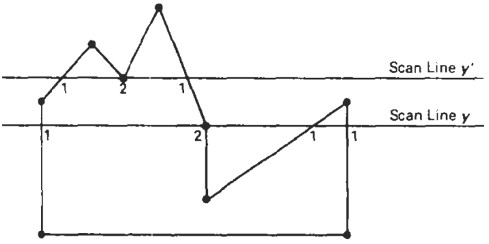
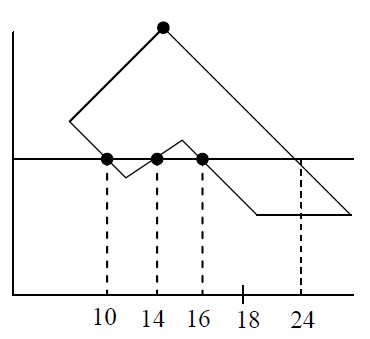
Filled Area primitives

A standard output primitive in general graphics package is solid color or patterned polygon area. Other kinds of area primitives are sometimes available, but polygons are easier to process since they have linear boundaries.

There are two basic approaches to area filling in raster systems. One way to fill an area is to   
determine the overlap intervals for scan lines that cresses the area. Another method for area   
filling is to start from a given interior position and point outward from this until a specified   
boundary is met.



SCAN-LINE Polygon Fill Algorithm:

In scan-line polygon fill algorithm, for each scan-line crossing a polygon, it locates the

intersection points of the scan line with the polygon edges. These intersection points are then

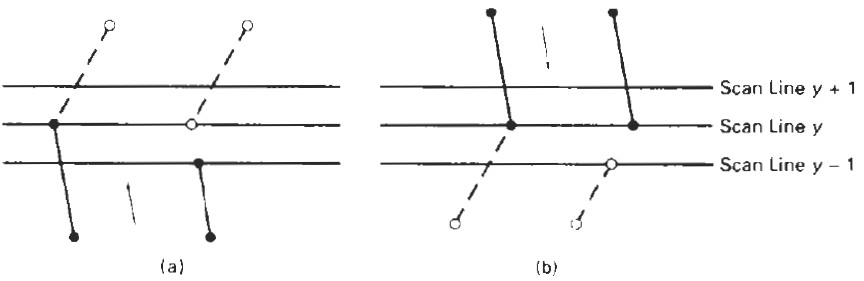
sorted from left to right, and the corresponding frame-buffer positions between each intersection pair are set to the specified color. In the figure i) below, the four pixel intersection positions with the polygon boundaries defined two stretches of interior pixel from x=10 to x=14 and from x=16 to x=24. Some scan-line intersections at polygon vertices require extra special handling. A scanline passing through a vertex intersect two polygon edges at that position, adding two points to the list of intersection for the scan-line.

Fig i) Interior pixels along a scan line Fig ii) Intersection points along scan lines that intersect

passing through a polygon area polygon vertices

Figure ii) shows two scan lines at position y and y' that intersect the edge points. Scan line at y intersects five polygon edges. Scan line at y' intersects 4 (even numbers) of edges though it passes through vertex.

Intersection points along scan line y' correctly identify the interior pixel spans. But with scan line   
y, we need to do some additional processing to determine the correct interior points. For scan   
line y, the two edges sharing the intersecting vertex are on opposite side of the scan-line. But for   
scan-line y' the two edges sharing intersecting vertex are on the same side (above) the scan line   
position. So the vertices those are on opposite side of scan line require extra processing.   
We can identify these vertices by tracing around the polygon boundary either in clockwise or   
counter clockwise order and observing the relative changes in vertex y coordinates as we move   
from one edge to next. If the endpoint y values of two consecutive edges monotonically increases   
or decrease, we need to count the middle vertex as a single intersection point for any scan line   
passing through that vertex. Otherwise the shared vertex represents a local extremum (minimum   
or maximum) on the polygon boundary, and the two edge intersections with the scan-line passing   
through that vertex.



One way to resolve the question as to whether we should count a vertex as one intersection or   
two is to shorten some polygon edges to split those vertices Filled-Area Primitives that should be   
counted as one intersection. We can process non horizontal edges around the polygon boundary   
in the order specified, either clockwise or counter clockwise. As we process each edge, we can   
check to determine whether that edge and the next non horizontal edge have either monotonically   
increasing or decreasing endpoint y values. If so, the lower edge can be shortened to ensure that   
only one intersection point is generated for the scan line going through the common vertex   
joining the two edges. Figure above illustrates shortening of an edge. When the endpoint y   
coordinates of the two edges are increasing, the y value of the upper endpoint for the current   
edge is decreased by 1, as in Fig.(a). When the endpoint y values are monotonically decreasing,   
as in Fig.(b), we decrease the y coordinate of the upper endpoint of the edge following the   
current edge.

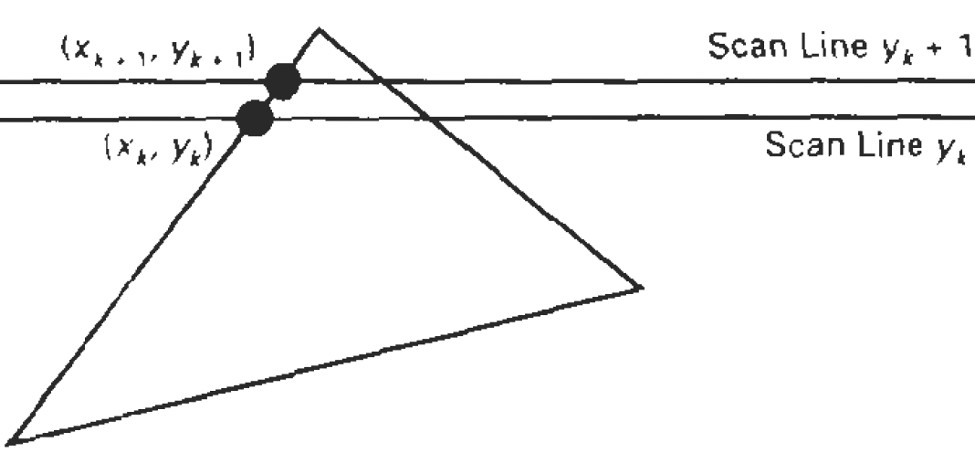
We can also use coherence which is simply that the properties of one part of a scene are related   
in some way to other parts of the scene so that the relationship can be used to reduce processing.   
Coherence methods often involve incremental calculations applied along a single scan line or   
between successive scan lines. In determining edge intersections, we can set up incremental   
coordinate calculations along any edge by exploiting the fact that the slope of the edge is   
constant from one scan line to the next. Figure below shows two successive scan lines crossing a   
left edge of a polygon. The slope of this polygon boundary line can be expressed in terms of the   
scan-line intersection coordinates:

m = ( yk+1 - yk) / (xk+1 - xk)

Since the change in y coordinates between the two scan lines is simply yk+1 - yk =1

the x-intersection value xk+1 on the upper scan line can be determined from the x-intersection value xk on the preceding scan line as

xk+1 = xk + 1/m



Each successive x intercept can thus be calculated by adding the inverse of the slope and rounding to the nearest integer. An obvious parallel implementation of the fill algorithm is to assign each scan line crossing the polygon area to a separate processor. Edge-intersection calculations are then performed independently. Along an edge with slope m, the intersection xk value for scan line k above the initial scan line can be calculated as

xk= x0+ k/m

slope m is the ratio of two integers: m = del y/ del x

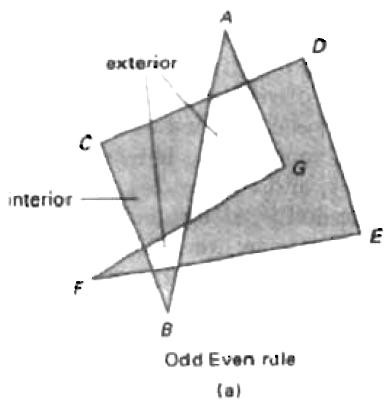
where del x and del y are the differences between the edge endpoint x and y coordinate values.   
Thus, incremental calculations of x intercepts along an edge for successive scan lines can be   
expressed as :

xk+1 = xk + del x / del y

Using this equation, we can perform integer evaluation of the x intercepts by initializing a counter to 0, then incrementing the counter by the value of del x each time we move up to a new scan line. Whenever the counter value becomes equal to or greater than del y, we increment the current x intersection value by 1 and decrease the counter by the value del y. This procedure is equivalent to maintaining integer and fractional parts for x intercepts and incrementing the fractional part until we reach the next integer value.

Inside-Outside Test:

Area filling algorithms and other graphics package often need to identify interior and exterior   
region for a complex polygon in a plane. For ex. in figure below, it needs to identify interior and   
exterior region.



We apply odd-even rule, also called odd-parity rule. To identify the interior or exterior point, we can draw a line from a point p to a distant point outside the coordinate extents of the object and count the number of intersecting edge crossed by this line. If the intersecting edge crossed by this line is odd, P is interior otherwise P is exterior.

Scan-Line Fill of Curved Boundary area

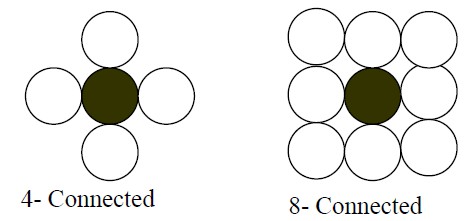
It requires more work then polygon filling, since intersection calculation involves nonlinear boundary for simple curves as circle, eclipses, performing a scan line fill is straight forward process. We only need to calculate the two scan-line intersection on opposite sides of the curve. Then simply fill the horizontal spans of pixel between the boundary points on opposite side of curve. Symmetries between quadrants are used to reduce the boundary calculation we can fill generating pixel position along curve boundary using midpoint method.

Boundary-fill Algorithm:

In Boundary filling algorithm starts at a point inside a region and paint the interior outward the boundary. If the boundary is specified in a single color, the fill algorithm proceeds outward pixel by until the boundary color is reached.

A boundary-fill procedure accepts as input the co-ordinates of an interior point (x,y), a fill color, and a boundary color. Starting from (x,y), the procedure tests neighbouring positions to determine whether they are of boundary color. If not, they are painted with the fill color, and their neighbours are tested. This process continues until all pixel up to the boundary color area have tested. The neighbouring pixels from current pixel are proceeded by two method:   
4- connected if they are adjacent horizontally and vertically.

8- connected if they adjacent horizontally, vertically and diagonally.



* Fill method that applies and tests its 4 neighbouring pixel is called 4- connected.
* Fill method that applies and tests its 8 neighbouring pixel is called 8- connected. The outline of this algorithm is:

void Boundary\_fill4(int x,int y,int b\_color, int fill\_color)   
{

int value=get pixel (x,y);

if (value! =b\_color&&value!=fill\_color)   
{

putpixel (x,y,fill\_color);

Boundary\_fill 4 (x-1,y, b\_color, fill\_color);   
Boundary\_fill 4 (x+1,y, b\_color, fill\_color);   
Boundary\_fill 4 (x,y-1, b\_color, fill\_color);   
Boundary\_fill 4 (x,y+1, b\_color, fill\_color);   
}

}

Boundary fill 8- connected:

void Boundary-fill8(int x,int y,int b\_color, int fill\_color)   
{

int current;

current=getpixel (x,y);

if (current !=b\_color&&current!=fill\_color) ( putpixel (x,y,fill\_color);

Boundary\_fill8(x-1,y,b\_color,fill\_color);   
Boundary\_fill8(x+1,y,b\_color,fill\_color);   
Boundary\_fill8(x,y-1,b\_color,fill\_color);   
Boundary\_fill8(x,y+1,b\_color,fill\_color);   
Boundary\_fill8(x-1,y-1,b\_color,fill\_color);   
Boundary\_fill8(x-1,y+1,b\_color,fill\_color);   
Boundary\_fill8(x+1,y-1,b\_color,fill\_color);   
Boundary\_fill8(x+1,y+1,b\_color,fill\_color);   
}

}

Recursive boundary-fill algorithm not fills regions correctly if some interior pixels are already   
displayed in the fill color. Encountering a pixel with the fill color can cause a recursive branch to   
terminate, leaving other interior pixel unfilled. To avoid this we can first change the color of any   
interior pixels that are initially set to the fill color before applying the boundary fill procedure.

Flood-fill Algorithm:

Flood\_fill Algorithm is applicable when we want to fill an area that is not defined within a single color boundary. If fill area is bounded with different color, we can paint that area by replacing a specified interior color instead of searching of boundary color value. This approach is called flood fill algorithm. We start from a specified interior pixel (x,y) and reassign all pixel values that are currently set to a given interior color with desired fill color.

Using either 4-connected or 8-connected region recursively starting from input position, The algorithm fills the area by desired color.

Algorithm:

void flood\_fill4(int x,int y,int fill\_color,int old\_color)   
{

int current;

current=getpixel (x,y);   
if (current==old\_color\_   
{

putpixel (x,y,fill\_color);

flood\_fill4(x-1,y, fill\_color, old\_color);   
flood\_fill4(x,y-1, fill\_color, old\_color);   
flood\_fill4(x,y+1, fill\_color, old\_color);   
flood\_fill4(x+1,y, fill\_color, old\_color);   
}

}

Similarly flood fill for 8 connected can be also defined.

We can modify procedure flood\_fill4 to reduce the storage requirements of the stack by filling horizontal pixel spans.

Filling Rectangle

Two things to consider

i. which pixels to fill

ii. with what value to fill

Move along scan line (from left to right) that intersect the primitive and fill in pixels that lay   
inside

To fill rectangle with solid color

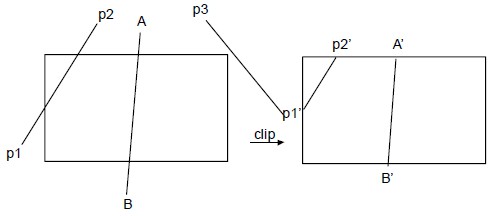
Set each pixel lying on scan line running from left edge to right with same pixel value, each span from xmax to xmin

for( y from ymin to ymax of rectangle) /\*scan line\*/

for( x from xmin to xmax of rectangle) /\*by pixel\*/

writePixel(x, y, value);

CLIPPING



Clipping may be described as the procedure that identifies the portions of a picture lie inside the region, and therefore, should be drawn or, outside the specified region, and hence, not to be drawn. The algorithms that perform the job of clipping are called clipping algorithms there are various types, such as:

• Line Clipping

• Polygon Clipping

• Curve Clipping

Further, there are a wide variety of algorithms that are designed to perform certain types of clipping operations, some of them which will be discussed in unit.

Line Clipping Algorithms:

• Cohen Sutherland Line Clippings

Polygon or Area Clipping Algorithm

• Sutherland-Hodgman Algorithm

LINE CLIPPING

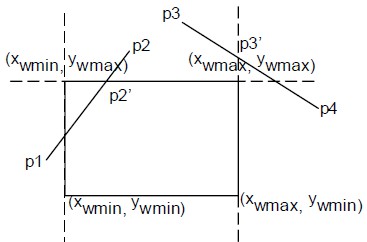
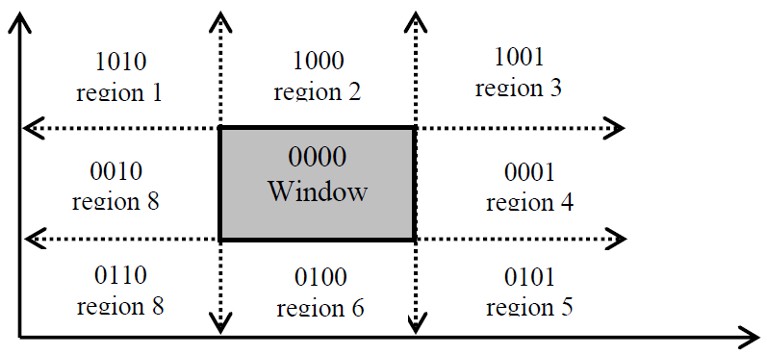
Line is a series of infinite number of points, where no two points have space in between them. So, the above said inequality also holds for every point on the line to be clipped. A variety of line clipping algorithms is available in the world of computer graphics, but we restrict our discussion to the following Line clipping algorithms, name after their respective developers:

Cohen Sutherland algorithm

In this method, every line endpoint is assigned a four digit binary code(region code) that identifies the location of the point relative to the boundary.

For,

b1 : left   
b2 : right   
b3 : below   
b4 : above



The value 1 indicates its relative position. If a point is within clipping rectangle then region code is 0000. So ,

If x - xwmin < 0 , b1 = 1   
If xwmax - x < 0 , b2 = 1   
If y - ywmin < 0 , b3 = 1   
If ywmin - y < 0 , b4 = 1

If the region codes of both end points are 0000 then we accept the line.

Now, to perform Line clipping for various line segment which may reside inside the window region fully or partially, or may not even lie in the widow region; we use the tool of logical ANDing between the b1b2b3b4 codes of the points lying on the line.

Logical ANDing (^) operation => 1 ^ 1 = 1; 1 ^ 0 = 0; between respective bits implies 0 ^ 1 = 0;

0 ^ 0 = 0

Any line that have one in the same bit position is rejected i.e if A AND B ≠ 0 Line is completely outside

The lines which con not be identified as completely inside or outside a window by these tests are checked for intersection with the window boundary. Such lines may or may not cross into the window interior.

In the fig. aside , region code of   
P1 = 0001

P2 = 1000

P1 AND P2 = 0

So we need further calculation. Starting form P1, Intersection of P1 with left boundary is calculated.

Region code of P1’ = 0000 P1’ AND P2 = 0 .

Intersecting of P2 with above boundary is calculated region code of P2’ = 0000

Since both end points have region codes (0000) .So P1’, P2’ portion of the line is saved.   
Similarly,

For P3, P4.   
P3 = 1000   
P4 = 0010

P3 AND P4 = 0

So we need further calculations; starting from P3 region code of P3 is 1000, i.e b4 is high, so intersection of P3 with upper boundary which yields P3’ having region code 1010.   
Again P3' AND P4 ≠ 0

So P3 P4 is totally clipped.

The intersection point with vertical boundary can be obtained by y = y1 + m(x-x1)

Where (x1,y1) and (x2,y2) are end points of line and y is the coordinate value of intersection point where x value is either xwmin or xwmax and

m = y2 -y1 / x2 - x1 .

Similarly , intersection point with horizontal boundary x = x1 + (y-y1)/m

Where , y = ywmin or ywmax

CLIPPING CIRCLES AND ELLIPSES

To clip a circle against a rectangle, we can first do a trivial accept/reject test by intersecting the   
circle’s extent (a square of the size of the circle‘s diameter) with the clip rectangle, If the circle   
intersects the rectangle, we divide it into quadrants and do the trivial accept reject test for each.   
These tests may lead in turn to tests for octants. We can then compute the intersection of the   
circle and the edge analytically by solving their equations simultaneously, and then scan convert   
the resulting arcs using the appropriately initialized algorithm with the calculated (and suitably

rounded) starting and ending points. If scan conversion is fast, or if the circle is not too large, it is probably more efficient to scissor on a pixel-by-pixel basis, testing each boundary pixel against the rectangle bounds before it is written. An extent check would certainly be useful in any ease. If the circle is filled, spans of adjacent interior pixels on each scan line can be filled without bounds checking by clipping each span and then filling its interior pixels .

To clip ellipses, we use extent testing at least down to the quadrant level as with circles. We can   
then either compute the intersections of ellipse and rectangle analytically and use those (suitably   
rounded) endpoints in the appropriately initialized scan—conversion algorithm, or clip as we   
scan convert.

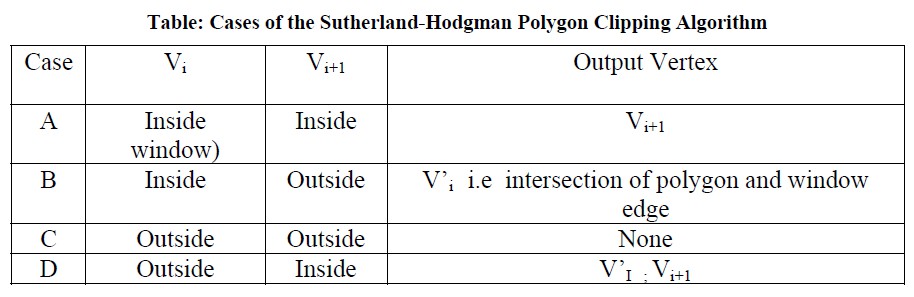
Clipping Polygons

Polygon is a surface enclosed by several lines. Thus, by considering the polygon as a set of line we can divide the problem to line clipping and hence, the problem of polygon clipping is simplified.Sutherland-Hodgman algorithm is one of the standard methods used for clipping arbitrary shaped polygons with a rectangular clipping window. It uses divide and conquer technique for clipping the polygon.

Sutherland-Hodgman Algorithm

Any polygon of any arbitrary shape can be described with the help of some set of vertices   
associated with it. When we try to clip the polygon under consideration with any rectangular   
window, then, we observe that the coordinates of the polygon vertices satisfies one of the four   
cases listed in the table shown below, and further it is to be noted that this procedure of clipping   
can be simplified by clipping the polygon edgewise and not the polygon as a whole. This   
decomposes the bigger problem into a set of sub problems, which can be handled separately as   
per the cases listed in the table below. Actually this table describes the cases of the Sutherland-  
Hodgman Polygon Clipping algorithm.

Thus, in order to clip polygon edges against a window edge we move from vertex Vi to the next   
vertexVi+1 and decide the output vertex according to four simple tests or rules or cases listed   
below:



In words, the 4 possible Tests listed above to clip any polygon states are as mentioned below:

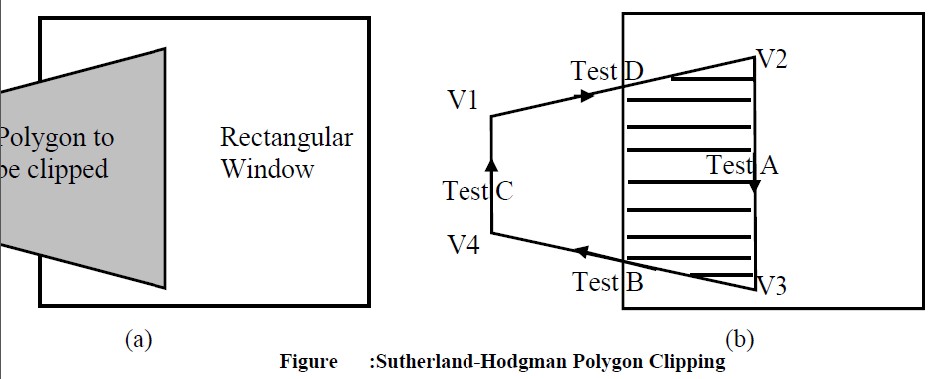
1) If both Input vertices are inside the window boundary then only 2nd vertex is added to output vertex list.

2) If 1st vertex is inside the window boundary and the 2nd vertex is outside then, only the intersection edge with boundary is added to output vertex.

3) If both Input vertices are outside the window boundary then nothing is added to the output list.

4) If the 1st vertex is outside the window and the 2nd vertex is inside window, then both the intersection points of the polygon edge with window boundary and 2nd vertex are added to output vertex list.

So, we can use the rules cited above to clip a polygon correctly. The polygon must be tested   
against each edge of the clip rectangle; new edges must be added and existing edges must be   
discarded, retained or divided. Actually this algorithm decomposes the problem of polygon   
clipping against a clip window into identical sub problems, where a sub problem is to clip all   
polygon edges (pair of vertices) in succession against a single infinite clip edge. The output is a   
set of clipped edges or pair of vertices that fall in the visible side with respect to clip edge. These   
set of clipped edges or output vertices are considered as input for the next sub problem of   
clipping against the second window edge. Thus, considering the output of the previous sub   
problem as the input, each of the sub problems are solved sequentially, finally yielding the   
vertices that fall on or within the window boundary. These vertices connected in order forms, the   
shape of the clipped polygon.



Pseudocode for Sutherland - Hodgman Algorithm

Define variables

inVertexArray is the array of input polygon vertices

outVertexArray is the array of output polygon vertices Nin is the number of entries in inVertexArray   
Nout is the number of entries in outVertexArray   
n is the number of edges of the clip polygon

ClipEdge[x] is the xth edge of clip polygon defined by a pair of vertices s, p are the start and end point respectively of current polygon edge i is the intersection point with a clip boundary

j is the vertex loop counter

Define Functions

AddNewVertex(newVertex, Nout, outVertexArray)

: Adds newVertex to outVertexArray and then updates Nout InsideTest(testVertex, clipEdge[x])

: Checks whether the vertex lies inside the clip edge or not; retures TRUE is inside else returns   
FALSE

Intersect(first, second, clipEdge[x])

: Clip polygon edge(first, second) against clipEdge[x], outputs the intersection point

{ : begin main

x = 1

while (x ≤ n) : Loop through all the n clip edges   
{

Nout = 0 : Flush the outVertexArray

s = inVertexArray[Nin] : Start with the last vertex in inVertexArray

for j = 1 to Nin do : Loop through Nin number of polygon vertices (edges)   
{

p = inVertexArrray[j]

if InsideTest(p, clipEdge[x] = = TRUE then : Case A and D if InsideTest(s, clipEdge[x] = = TRUE then

AddNewVertex(p, Nout, outVertexArray) : Case A   
else

i = Intersect(s, p, clipEdge[x]) : Case D   
AddNewVertex(i, Nout, outVertexArray)   
AddNewVertex(p, Nout, outVertexArray)   
end if

else : i.e. if InsideTest(p, clipEdge[x] = = FALSE (Cases 2 and 3)

if InsideTest(s, clipEdge[x]) = =TRUE then : Case B   
{

Intersect(s, p, clipEdge[x])

AddNewVertex(i, Nout, outVertexArray) end if : No action for case C

s = p : Advance to next pair of vertices   
j = j + 1

end if : end {for}   
}

x = x + 1 : Proceed to the next ClipEdge[x +1] Nin = Nout

inVertexArray = outVertexArray : The ouput vertex array for the current clip edge becomes the input vertex array for the next clip edge

} : end while

} : end main