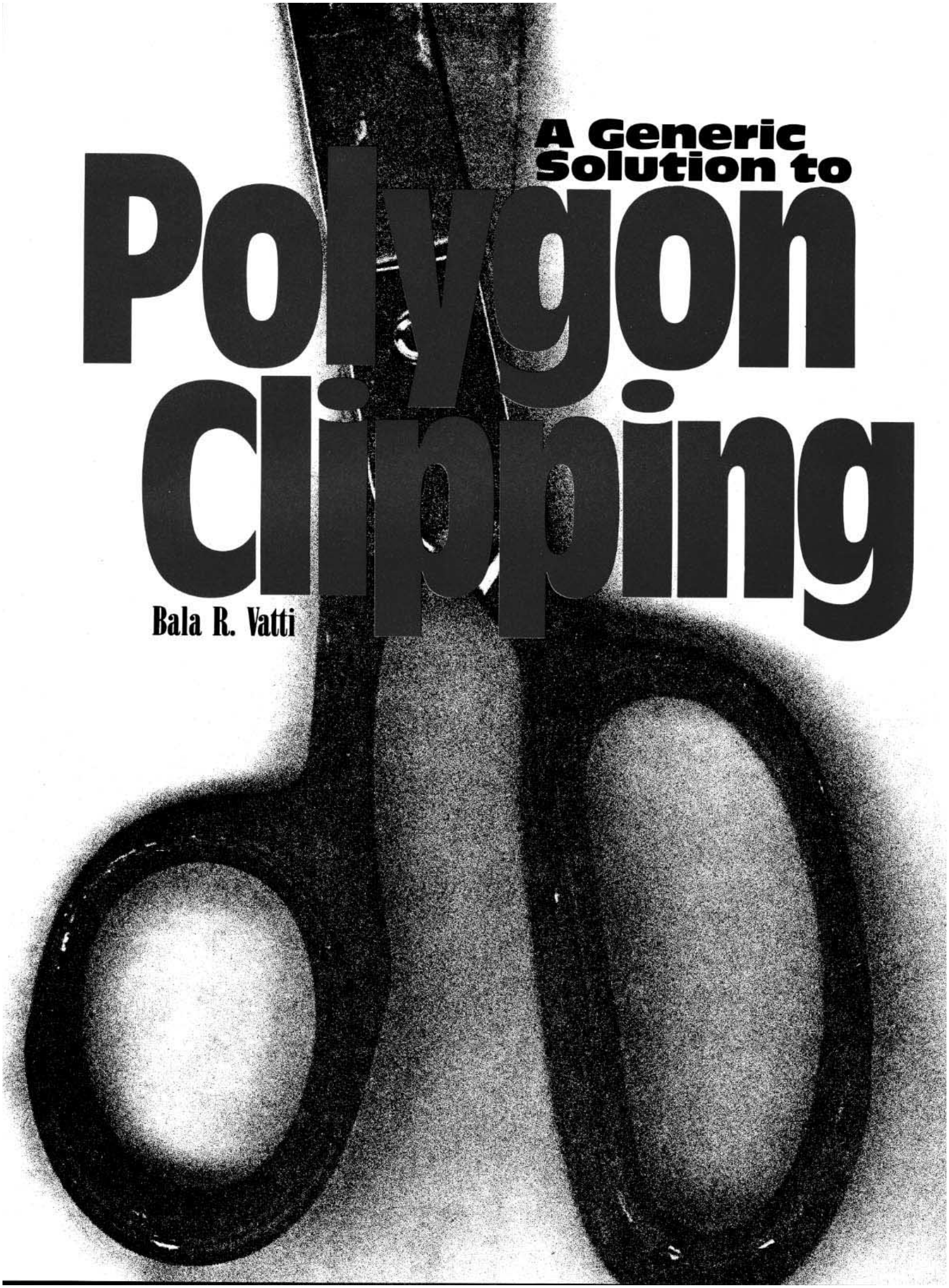
A Generic Solution to Polygon Clipping

Bala R. Vatti



Clipping is an essential part of image synthesis. Traditionally, polygon clipping has been used to clip out the portions of a polygon that lie outside the window of the output device to prevent undesirable effects. In the recent past polygon clipping is used to render 3D images through hidden surface removal [9, 11] and to produce high-quality surface details using techniques such as Beam tracing [3]. Polygon clipping is also used in distributing the objects of a scene to appropriate processors in multiprocessor raytracing systems to improve rendering speeds [1]. Clipping an arbitrary polygon against an arbitrary polygon has been a complex task. The existing solutions for polygon clipping are either limited to certain types of polygons or tend to be very complex and time consuming.

The reentrant polygon clipping by Sutherland and Hodgeman is limited to convex clip polygons [2, 9, 11]. This also produces degenerate edges in certain concave/self intersecting polygons that need to be removed as a special extension to the main algorithm. Another approach for polygon clipping developed by Liang and Barsky assumes that the clip polygon is always rectangular with sides parallel to the coordinate axes [6]. More general clipping algorithms, presented in [8, 10, 12], are capable of clipping a concave polygon with holes to the borders of a concave polygon with holes. These algorithms may not permit self-intersecting polygons. Montani and Re presented a solution that divides polygons into parallel-connected horizontal stripes of a constant height to remove the hidden portions of the polygons in which the selected height determines the output resolution [8]. A more efficient polygon division was made based on the complexity of the polygons [10].

One of the main reasons for clipping a polygon is to fill it correctly. Therefore, filling a polygon seems to be a natural step after clipping. Some output devices may be capable of rendering complete polygons or trapezoids while the others can render only a line segment at a time. Output in the form of trapezoids is particularly useful for scan-line-based rendering algorithms and it is quite suitable for hardware implementations [13]. A few algorithms are available to decompose a polygon into trapezoids [5, 4, 7]. The clipping algorithm presented in this article lends itself to producing the output in the form of trapezoids directly if required.

General Description

A polygon may be a single polygon or a polygon set. Each individual polygon in a polygon set may be convex, concave or self-intersecting. Figure 1 shows a polygon with its interior areas shaded. The algorithm described in this article clips a polygon (referred to as the subject polygon) against a polygon (referred to as the clip polygon). Clipping is defined as interaction of the subject and the clip polygons. The algorithm is also suitable for 2D Boolean operations union and difference.

A polygon is perceived as being formed by a set of left and a set of right bounds. Each bound starts at a local minimum and ends at a local maximum. All the edges on the left bound are called left edges and those on the right are called right edges. The left and right sides are defined with respect to the interior(s) of the polygon. The edges of the polygon may cross, in which case the algorithm converts the polygon to a nonintersecting polygon by inserting the points of intersections as the clipping is being performed.

The subject and the clip polygons are traversed once, and all the bounds are formed. Each edge of a bound is assigned a flag indicating the type (clip/subject) of the edge.

The polygons are scanned from bottom to top using *scanbeams*. Each scanbeam is the horizontal sweep area between two successive events as described in [10]. In other words, a scanbeam is defined as the area between two successive horizontal lines from a set of horizontal lines drawn through all the vertices. An Active Edge Table (AET) is maintained to indicate the list of all the edges that are intersected by the current scanbeam. The edges in the AET are sorted in the ascending order of the x coordinates at the bottom of the current scanbeam. The x coordinate values are updated as the scanning proceeds from one scanbeam to the next.

The first vertex of each bound

corresponds to a local minimum and the last vertex to a local maximum. The remainder of the vertices are called intermediate vertices. The intermediate vertices on the left bounds are referred to as the left intermediate vertices and those on the right bounds as the right intermediate vertices. Processing of horizontal edges is explained in the 'Extensions' section of this article.

The edge intersections are computed as the polygons are scanned. Each intersection is classified similar to the vertex classification. This classification is made based on some rules which are, in turn, based on the definition of the clipping operation. This classification is explained in detail in a later section.

When the first edges of a pair of bounds become active at a local minimum, one edge is assigned as the left edge and the other is assigned as the right edge. This assignment is based on the even/odd parity of the edge with respect to the rest of the edges of the same type in the AET. The local minimum is considered as a contributing or noncontributing vortex, based on the position with respect to the other types of edges. If it is contributing, then a polygon node is created and is assigned to both the edges, and these edges are considered as contributing edges. If not, a null pointer is assigned, indicating that the edges are noncontributing. A contributing edge means the edge is currently in the process of contributing to the output. When the scanning reaches the top of an edge, the edge is replaced by its successor edge and the polygon pointer and the left-right flag are passed to the successor edge. Thus the successor edge assumes the properties of the edge it replaces.

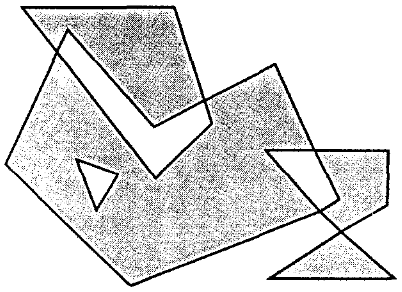


Figure 1. Example of a clip/subject polygon

The polygon pointer indicates whether or not there should be output at a given vertex of a given edge. The vertex may be an end point of the edge or a point of intersection. The class of a vertex indicates how the output should be generated, which results in closed contour(s) at the end of clipping.

Vertices have to be extracted as an ordered list as efficiently as possible. When we reach the upper vertex of a contributing edge we need to determine the class of the vertex in order to produce the output. Simple checks will determine the class of a given vertex. If the edge has no successor edge (null pointer), then it is considered a local maximum, otherwise it is an intermediate vertex. Processing at each vertex depends on the class of a given vertex as follows:

1. Local minimum: create a polygon node and assign the local minimum to the vertex list of the polygon. As mentioned earlier, additional local minima may be formed through unlike edge intersections. These local minima should be treated as contributing local minima. Edges connected to a contributing local minima become contributing edges through the assignment of output polygons to these edges.

2. Left and right intermediate: When an intermediate vertex is found, the vertex need be added only to the left end or to the right end of the vertex list of the polygon assigned to the edge, depending on whether the edge side is left or right.

3. Local maximum: At a local maximum a pair of bounds meet. Both may belong to the same polygon in which case we have a closed contour for the polygon. If they belong to different polygons, one is appended to the other polygon. For a given polygon node, there would be two edges contributing to it at any time: one edge contributes to the left end and the other to the right end of the vertex list of the polygon node. When polygon P is appended to polygon Q at a local maximum, there would be four contributing edges, two for each polygon, say Ep1, Ep2 and Eq1, Eq2. After we append P to Q, the middle two edges Ep2 and Eq1 become noncontributing and the edge Ep1 will be contributing to Q. Therefore the polygon pointer of Ep1 should be set to Q. Note that Ep1 and Ep2 are adjacent edges in the AET, thus there is no need to search for Ep1.

At each unlike edge (edges of different types) intersection of a contributing edge becomes a noncontributing edge and noncontributing edge becomes a contributing edge by exchanging output pointers. At each like edge intersection, a left edge becomes a right edge and a right edge becomes a left edge. Both the intersecting edges are swapped at the intersection to maintain x sort in the AET.

Thus the polygon contour can be formed very naturally without sorting, searching or complex data structures. At the end of the scanning we will have a set of closed contours defining the output polygons. Processing time varies linearly with the total number of edges in the subject and clip polygons. In a later section, on trapezoid generation, we will explain how trapezoids can be output instead of polygons.

Intersection Classification

The edge intersections can be classified into two types: intersections formed between like edges and those formed between unlike edges. A like edge intersection should be considered only if both the edges are contributing. Note that if one edge is contributing, so is the other edge of a pair of like intersecting edges. An intersection between like edges must be treated as both left and right intermediate vertices. In the case of unlike edge intersections, all the intersections must be considered. The resulting vertex classification depends on the types (clip/subject), on the sides (left/right) and on their relative position in AET.

We can now define a set of rules to classify intersections. The following convention is used to formalize the rules: An edge is identified by a two-letter word. The first letter indicates whether the edge is left (L) or right (R) edge, and the second letter indicates whether the edge belongs to subject (S) or clip (C) polygon. An edge intersection is indicated by *X*. The vertex formed at the intersection is assigned one of the four vertex classifications: Local minimum (MN), local maximum (MX), left intermediate (LI) and right intermediate (RI). The symbol '|' is used to mean 'or'.

Rules to classify intersections between unlike edges are:
Rule 1: LC × LS|LS × LC = LI
Rule 2: RC × RS|RS × RC = RI
Rule 3: LS × RC|LC × RS = MX
Rule 4: RS × LC|RC × LS = MN
Rules to classify intersections between like edges are:
Rule 5: LC × RC|RC × LC = LI and RI
Rule 6: LS × RS|RS × LS = LI and RI

Figure 2 shows the vertex classifications formed by these rules for intersection operation. Rule 1 can be stated as follows: Intersection of left clip edge followed by left subject edge OR intersection of left subject edge followed by left clip edge produces left intermediate vertex. We will define the rules to produce union and difference operations in later sections.

Implementation

The following data structure is defined to represent an edge.

Local Minima Table (LMT): This is a linked list of nodes sorted in ascending order of *y* coordinate. Each node points to a list of bounds that start at the *y* coordinate. Thus each node corresponds to the *y* coordinate of one or more local minima. The LMT is built at the time of forming the bounds, prior to clipping. Figure 3 shows a polygon with its edges assigned to the LMT.
Scanbeam Table (SBT): This is built as the polygons are scanned to keep the length of the list to a minimum. The upper end of the current scanbeam is determined by the smaller of the minimum 'ytop' of all the edges in the AET and the next LMT entry. This is the next entry in the SBT.
Let us define the following operations to simplify the algorithm presented later:

MarkSBT(y)
Insert a node representing 'y' in sorted order in SBT, if it does not exist.

AddLocalMin(edge,p)
Create a polygon node and set right(R) and left(L) pointers to point to 'p'.
Assign the polygon(P) to the edge and its next edge as the output polygon.

AddLeft(edge,p)
Add vertex(p) to the left end(L) of the vertex list of the polygon assigned to the edge.
Update L to point 'p'

AddRight(edge,p)
Add vertex(p) to the right end(R) of the vertex list of the polygon assigned to the edge.
Update R to point to 'p'.

AddEdges(LMTnode)
```
For each pair of edges (edge1,
  edge2) of LMTnode do;
    Add edge1, edge2 to the AET in
      sorted order.
    Assign the side (left/right) of
      edge1 and edge2.
      (note that edge1 and edge2
      start from a local minimum 'p')
    If it is a contributing local
      minimum then
      AddLocalMin(edge1,p);
    MarkSBT(edge1->ytop);
    MarkSBT(edge2->ytop);
    end;
```

AppendPolygon(edge1,edge2)
```
Set P = edge1->poly and
  Q = edge2->poly;
Assign vertex list of P to the left or
  right end of vertex list of Q
  depending on the side of edge1;
Set (edge1->prev)->poly = Q;
```

AddLocalMax(edge,p)
```
If the edge is left edge
  AddLeft(edge,p);
Else AddRight(edge,p);
If the edge and its next edge have
the same output polygon return;
AppendPolygon(edge1,edge2);
```

Edge Intersections

All the intersections in the current

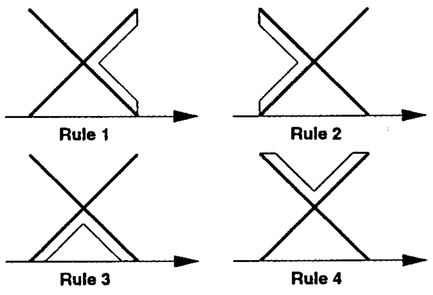


Figure 2. Intersection classification

Table 1.
Edge data structure

Field	Description
xbot	lower x coordinate
ytop	upper y coordinate
delx	change in x for a unit increase in y
type	clip/subject edge flag
side	left/right flag
poly	output polygon pointer
next	next edge in the AET
prev	previous edge in the AET
succ	successor edge (edge connected to the upper end)

scanbeam must be processed before we move on to the next scanbeam. We know that all the edges in the AET are already in sorted order. When the scanning reaches the top of the current scanbeam (bottom of the next scanbeam) we need to update 'xbot' values of all the edges. If edges intersect in the current scanbeam, the updated 'xbot' values (xtop values) will not be in sorted order. The number of jumps an edge must make to find its sorted location will give us the exact intersections of the edge with the remaining edges. We create temporary *Sorted Edge Table* (ST) and *Intersection Table* (IT) to identify and store all the intersections in the current scanbeam. Each node of the ST stores the pointer to an edge and its xtop value. The IT is a linked list of intersection nodes sorted in y coordinates of the intersections. Each node contains the pointers of both the intersecting edges and the intersection. The algorithm to find the intersecting edge pairs is as follows:

```
Set Dy to the height of the current
  scanbeam.
Initialize the first node (right end)
  of ST using the first edge
  of AET.
Set STedge to the first ST node.
For each AETedge of the
  remaining edges in AET do;
  xtop = AETedge->xbot +
    AETedge->delx*Dy;
  While (Xtop < STedge->xtop)do;
    Compute intersection between
      STedge and AETedge;
    Insert the intersection and
      both the edge pointers in IT.
    Set STedge to its left STedge;
    End;
  Insert AETedge to the right of
    STedge;
  End;
```

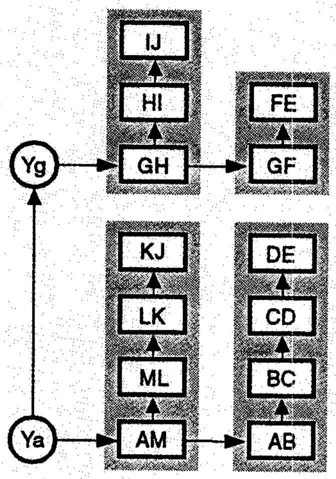
Clipping Algorithm

The clipping algorithm can be stated as follows:

```
Build LMT;
Set AET to null;
For each scanbeam do;
  Set yb and yt to bottom and top
    of the scanbeam.
  If LMT node corresponding to
    yb exists AddEdges(LMTnode);
  Build Intersection Table (IT) for
    the current scanbeam;
  For each node in IT;
    Set edge1 and edge2 from the
      IT node; /* edge1 precedes
        edge2 in AET */
    Classify the point of
      intersection 'p';
    Switch (class of p) do;
      Case(LIKE_EDGE_INT):
        If edge 1 is contributing
          then
          AddLeft(edge1,p);
          AddRight(edge2,p);
          Exchange edge1->side
            and edge2->side;
      Case(LOCAL_MAX):
        AddLocalMax(edge1,p);
      Case(LEFT_INT):
        AddLeft(edge2,p);
      Case(RIGHT_INT):
        AddRight(edge1,p);
      Case(LOCAL_MIN):
        AddLocalMin(edge1,p);
      End; /* switch */
    Swap edge1 and edge2
      positions in the AET;
    Exchange edge1->poly and
      edge2->poly;
    End; /* IT loop */
  For each AETedge do;
    If AETedge is terminating at yt
      do;
      Classify the upper end vertex
        'p' of AETedge;
      Switch (class of p)
        Case(LOCAL_MAX):
          AddLocalMax(AET
            edge,p);
          Delete AETedge and
            AETedge->next from
            the AET;
        Case(LEFT_INT):
          AddLeft(edge2,p);
          Replace AETedge by
            AETedge-> succ;
        Case(RIGHT_INT):
          AddRight(edge1,p);
          Replace AETedge by
            AETedge->succ;
      End; /* switch */
    End; /* if */
  End; /* AET loop */
End; /* SBT loop */
```

Trapezoid Generation

Trapezoids can be generated as output with a few modifications to the preceding algorithm. A local minimum starts a trapezoid strip or breaks an existing trapezoid strip into two, depending on whether the local minimum is formed by a left-right or a right-left edge pair. At each contributing local minimum we create a trapezoid node, similar to polygon node. Each node requires to store only bottom *x* coor-



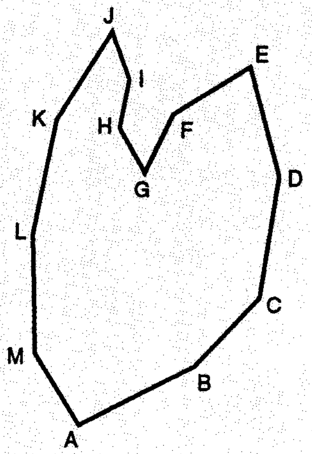


Figure 3. Local minima table

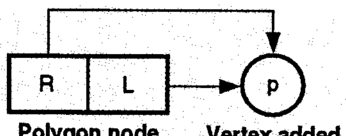


Figure 4. Processing local minimum

dinates and the corresponding *y* coordinate. Whenever a local maximum, left intermediate or right intermediate vertex is encountered, a trapezoid should be output. A local minimum formed by a right-left pair should also output the trapezoid it splits and the trapezoid pointers for the participating edges should be reassigned appropriately.

Trapezoids can be output in the form (*X*left,*X*right,*Y*bot,*DX*left, *DX*right,*Y*top) and these can be scan-converted very easily in a simple loop as follows:

```
For Y = Ybot to Ytop do;
  DrawLine(Xleft,Xright,Y);
  Xleft = Xleft + DXleft;
  Xright = Xright + DXright;
  End loop;
```

Example

Let us denote an output polygon node with P[R:L](*p*1,*p*2,...,*pn*), where P is the polygon pointer, R is right end vertex pointer, L is left end vertex pointer and the list *p*1,*p*2,...*pn* are the vertices of the polygon. Figure 8 shows subject polygon S(*s*1,*s*2,...,*s*8) and clip polygon C(*c*1,*c*2,...,*c*9). The edge intersections are denoted by *i*1,*i*2,...,*i*8. The following table describes only those events at which output is generated.

Results

The algorithm was implemented in C and executed on a MIPS processor R2000 under a Unix™ operating system. Trapezoids were generated as output of the clipping and these were filled as described earlier. Actual machine cycles were obtained for several cases and the average timings were computed for each case and listed under Method 1. Similar timings were obtained for the Reentrant Polygon Clipping [9] and scanline fill algorithms and listed under Method 2. Since the reentrant polygon clipping algorithm does not permit concave clip polygons, only convex clip polygons and concave subject polygons were used for the test cases. The sizes of clip and subject polygons are selected such that they fit in a 1,280 × 1,024 resolution.

Improvement factors were computed for each case by dividing performance values obtained for the Method 2 by those in Method 1. The averages of such improvement factors are listed under 'Improvement' columns. Table 3 shows the results of clipping subject polygons with a varying number of edges using the same clip polygon.

The total timings of clip and fill indicate that there is always considerable performance improvement over SH clip followed by scanline fill operations. This table also indicates that both clipping and filling can be done faster than the scanline fill algorithm alone. Further, we can see that the relative clipping performance improved as the number of edges increased.

Extensions

Union and difference operations: The discussions presented assumed the clipping is the intersection of two polygons. If we want to output polygon as the union of two polygons, the output rules should be modified as follows:

1. LC × LS|LS × LC = LI
2. RC × RS|RS × RC = RI
3. LS × RC|LC × RS = MN
4. RS × LC|RC × LS = MX

All local minima of subject polygon which lie outside clip polygon and all local minima of clip polygon which lie outside subject polygon should be treated as contributing local minima.

For difference operation (subject polygon minus clip polygon), the rules should be as follows:

1. RC × LS|LS × RC = LI
2. RS × LC|LC × RS = RI
3. RS × RC|LC × LS = MN
4. RC × RS|LS × LC = MX

All local minima of subject polygon which lie outside clip polygon should be treated as contributing local minima.

Horizontal edges: Processing of horizontal edges becomes a special case for this algorithm. The determination of vertex classifications should be made based on the assumption that the horizontal edge is absent. Since horizontal edge intersections are available at the top and bottom of scanbeams, horizontal edges can be processed efficiently as a special case. The algorithm can be optimized for rectangular clip bounds, which is a typical usage of clipping. The algorithm can also be optimized for clipping several subject polygons

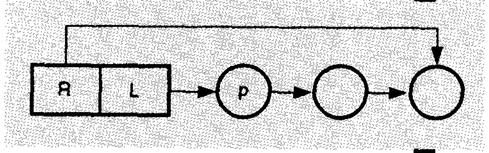


Figure 5. Processing left intermediate

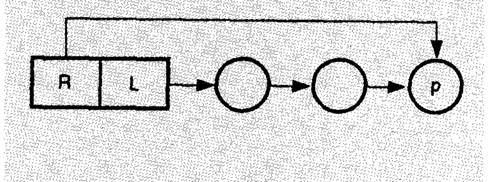


Figure 6. Processing right intermediate

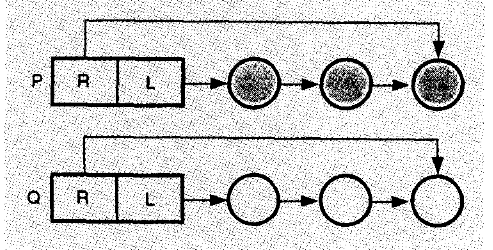


Figure 7a. Appending polygons (before)

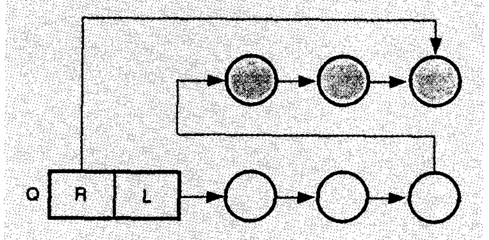


Figure 7b. Appending polygons (after)

Table 2.
Polygon clipping example

Event	Output Generated	Description
s1	P[s1:s1](s1)	Contributing local minimum
s7	P[s1:s7](s7,s1)	Left intermediate
i1	P[i1:s7](s7,s1,i1)	Intersection Rule 2
i2	P[i1:i2](i2,s7,s1,i1)	Intersection Rule 3. Polygon P closed.
i3	Q[i3:i3](i3)	Intersection Rule 4. Polygon Q created.
i4	Q[i3:i4](i4,i3)	Intersection Rule 1
c4	Q[c4:i4](i4,i3,c4)	Right intermediate
i5	Q[i5:i5](i5,i4,i3,c4,i5)	Like edge intersection
i6	Q[i5:i6](i6,i5,i4,i3,c4,i5)	Intersection Rule 1
i7	Q[i7:i6](i6,i5,i4,i3,c4,i5,i7)	Intersection Rule 2
s6	Q[i7:s6](s6,i6,i5,i4,i3,c4,i5,i7)	Left intermediate
i8	Q[i8:i8](i8,s6,i6,i5,i4,i3,c4,i5,i7,i8)	Like edge intersection
s3	Q[i8:s3](s3,i8,s6,i6,i5,i4,i3,c4,i5,i7,i8)	Left intermediate
s5	Q[s5:s3](s3,i8,s6,i6,i5,i4,i3,c4,i5,i7,i8,s5)	Right intermediate
s4	Q[s5:s4](s4,s3,i8,s6,i6,i5,i4,i3,c4,i5,i7,i8,s5)	Local maximum

Table 3.
Results of the clipping and filling operations

	Method1			Method2			Improvement		
Edges	Clip	Fill	Total	Clip	Fill	Total	Clip	Fill	Total
4	1.20	1.97	3.17	1.88	24.37	26.26	1.62	12.40	8.33
10	3.06	6.17	9.22	4.89	57.55	62.44	1.67	9.39	6.27
20	5.35	11.41	16.77	9.10	98.92	108.03	1.83	8.19	5.59
40	9.73	21.60	31.34	18.16	181.71	199.86	2.05	7.45	5.20
100	23.76	52.65	76.41	47.85	430.11	477.96	2.18	6.23	4.86
200	45.77	104.39	150.15	100.99	844.15	945.15	2.44	5.62	4.81

against a constant clip polygon by preassigning the edges of the clip polygon to the LMT.

Conclusion

A general and efficient polygon-clipping algorithm is presented. The term 'clipping' is also defined in a more general sense which may mean intersection, union or difference. The output of the algorithm can be polygons or trapezoids. The results indicated significant performance improvements over traditional polygon-clipping and filling operations.

Acknowledgments
The author would like to thank David Bailey for his support and Tyler Brown and George Schaeffer for their suggestions, all of which made this paper more readable.

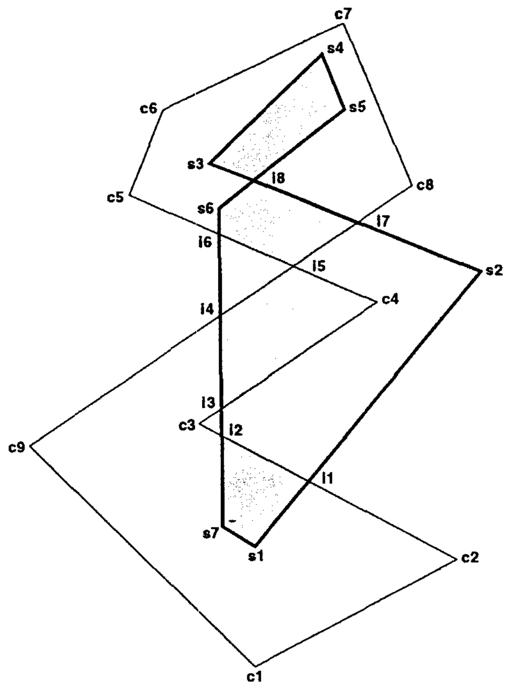


Figure 8. Polygon clipping example

References

1. Cleary G.J., Wyvill B., Birtwistle G.M. and Vatti R. Multiprocessor raytracing. Tech. Rep. 83/128/17, Department of Computer Science, The University of Calgary, Oct. 1983.
2. Foley J. and Vandam A. *Fundamentals of Computer Graphics.* Addison-Wesley, Reading, Mass., 1984.
3. Heckbert P.S., and Hanrahan P. Beam tracing polygonal objects. *Comput. Graph. 18,* 3 (July 1984), 119–127.
4. Jackson, J.H. Dynamic scan-converted images with a frame buffer display device, *Comput. Graph. 14,* 3 (July, 1980), 163–169.
5. Lee, D.T. Shading regions on vector display devises, *Comput. Graph. 15,* 3 (1981).
6. Liang Y. and Barsky B.A. An analysis and algorithm for polygon clipping. *Commun. ACM 11,* 26 (Nov. 1983), 868–877.
7. Little W.D. and Heuft R. An area shading graphics display system. *IEEE Trans. Comput. c-28,* 7 (July 1978), 528–530.
8. Montani C. and Re M. Vector and raster hidden surface removal using parallel connected stripes. *IEEE Comput. Graph. Appl. 7,* 7 (July 1987), 14–23.
9. Newman W.M. and Sproull R.F. *Principles of Interactive Computer Graphics.* Second ed., Mcgraw-Hill, N.Y.
10. Sechrest S. and Greenberg D. A visible polygon reconstruction algorithm. *Comput. Graph. 15,* 3 (1981), 17–26.
11. Sutherland E.E. and Hodgeman G.W. Reentrant polygon clipping. *Commun. ACM 17,* 1 (Jan. 1974), 32–42.
12. Weiler K. and Atherton P. Hidden surface removal using polygon area sorting. In *Proceedings of SIGGRAPH 11,* 2 (Summer, 1977), pp. 214–222.
13. Winberg R. Parallel processing image synthesis and anti-aliasing. *Comput. Graph. 15,* 3 (Aug. 1981), 55–61.

CR Categories and Subject Descriptors: E.2 [**Data**]: Data Storage Representations—*linked representations;* I.3.3 [**Computer Graphics**]: Picture/Image Generation—*display algorithms;* I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*geometric algorithms, languages, and systems*

General Terms: Algorithms, performance

Additional Keywords and phrases: connectivity coherence, contributing edge, contributing local minimum, difference, hidden surface, intersection, polygon clipping, scanbeam, spatial coherence, successor edge, trapezoids, union, vertex classification

About the Author:
BALA R. VATTI is a principal software engineer at Lockheed Commercial Electronics Company (LCEC) in Hudson, N.H. His research interests include high-performance graphics systems for CAD/CAM applications. **Author's Present Address:** LCEC, 65 River Road, Hudson, N.H. 03051. email: vatti@waynar.lcec.lockheed.

CARE plants the most wonderful seeds on earth.

Seeds of self-sufficiency that help starving people become healthy, productive people. And we do it village by village by village. Please help us turn cries for help into the laughter of hope.