Polygon Filling Based on Vertex Classification

Technical Report No. 194-91

Fuhua Cheng  †

William E. Toll  ‡‡

† Department of Computer Science, University of Kentucky, Lexington, KY 40506

‡‡ Computing and System Sciences Department, Taylor University, Upland, IN 46989
Polygon Filling Based on Vertex Classification

Fuhua Cheng
Department of Computer Science
University of Kentucky
Lexington, KY 40506

William E. Toll
Computing and System Sciences Department
Taylor University
Upland, Indiana 46989

ABSTRACT

Fast display of shaded polygons on a raster device is important in many areas of computer graphics. Two algorithms are presented which fill an arbitrary simple polygon without prior decomposition. The polygon may contain holes and may have touching vertices or edges. The algorithms are based on the classification of vertices. The vertices are classified according to the action that needs to be taken during the filling process. The first algorithm fills sections of the polygon independently and, therefore, is easily adaptable to parallel processing. The second algorithm maintains a list of active edges in much the same way as the scanline algorithm, but updates the ordered list of edges based on classification of vertices encountered. Testing has been performed on a variety of sequential platforms. Testing results indicate that the algorithms are competitive on most of the machines, and they outperform the scanline algorithm for 2D polygons with a reasonable number of sides (≤50). The second algorithm has also been used for 3D Z-buffered implementation. It is competitive with the scanline algorithm for small number of polygons and becomes faster when the number of polygons is large (1200-1500 for triangles/quadrilaterals and 200 for random polygons).


General Term : Algorithms

Additional Key Words and Phrases : Scanline, z-buffer, plane sweep, merge/split, ael

* Supported in part by IBM and the University of Kentucky Center for Computational Sciences.
1. Introduction

The filling of a polygon for a raster display system is often accomplished by dividing the polygon into trapezoids or triangles. Some of the decomposition algorithms have been based on classification of the vertices of the polygon. Fournier and Montuno present an algorithm for triangulating an arbitrary simple polygon with holes by first decomposing the polygon into trapezoids and then breaking the trapezoids into triangles.

In their approach vertices of a simple polygon are classified into three types based on the relationship of their adjacent edges with respect to the horizontal (scan) lines going through the vertex. A Type 1 vertex has an adjacent edge on each side of the horizontal line through the vertex. Type 2 vertices have both edges below the horizontal line and Type 3 vertices have both edges above the horizontal line through the vertex. Type 1 vertices mark the beginning or end of a trapezoid. A Type 2 vertex marks the end of a trapezoid and the beginning of one or two new ones. A Type 3 vertex marks the end of two trapezoids and the beginning of a new one or the end of a single trapezoid. Although the concepts are similar to the approach taken in this paper, the vertices classified here as START and STOP refer to the vertical beginning or end of a region to be filled rather than the beginning or end of scan lines within a trapezoid. Thus, simple start and stop conditions in the two approaches are rotated 90 degrees from each other.

Vertex classification was used earlier by Lee and Brassei and Fegeas for filling regions on vector display devices. Lee named the Type 1 vertices regular, Type 2 stalagmitic and Type 3 stalactitic. The algorithms of both Lee and Brassei and Fegeas, in effect, break the region to be filled into trapezoids.

The approach taken here is to use the classification of a vertex to determine if the filling of a region of a polygon can be started or stopped, or the region needs to be split into two (or more) regions, or if two (or more) regions can be combined into one region to be filled.

Two algorithms are presented here. The first, called merge/split, begins filling a region at a START vertex and continues until the region is split, needs to be merged with another region, or the region ends. The filling process continues when it reaches a vertex that does not cause any of the above actions. Each region of the polygon is begun with a START vertex and can be started independently. One of the major complexities of the algorithm is in merging regions that have been started independently.

The second algorithm, named AEL for active edge list, was developed when a more sequential approach was investigated. In this algorithm, regions are filled by maintaining a list of active edges. The area between each pair of edges is filled in a manner much like the scanline algorithm as developed by Wylie, Romney, Evans, and Erdahl, Bouknight, and Watkins. The algorithm reported here differs from scanline in that the active edge list is examined only when a vertex is encountered. The classification of the vertex indicates the modifications that need to be made in the active edge list. Since filling continues until a vertex is encountered, multiple raster lines are filled in a single process rather than checking vertices at each raster line as is done in the scanline algorithm.
In the filling process of the *merge/split* algorithm, the main loop that fills a segment of the polygon is driven by the change in y coordinates as the region is filled. In the AEL algorithm, the main filling loop for the entire polygon is driven by vertices encountered rather than y-value. In the traditional scanline approach, the loop is driven by y value even more than in the merge/split algorithm, since each y-value results in checking for changes in the active edge list.

The AEL algorithm is also very similar in nature to typical *plane sweep* algorithms from computational geometry.\textsuperscript{13} Such algorithms are used for *point location problems* and *line segment intersection problems*.\textsuperscript{12} Such algorithms halt at "event points" where some action needs to be taken. Here the event points are the vertices and the action to be taken is determined by their classification. Sweep algorithms also must maintain a data structure which contains all the information necessary to continue the process. In this algorithm that data structure is the active edge list which is modified as a result of the classification of the vertex encountered.

Due to the similarity of the AEL approach to the scanline algorithm, generalization of the simple 2D scanline algorithm to handle 3D polygons with simple z-buffer methods\textsuperscript{10} and multiple polygons can be handled in much the same way as scanline approaches. Properties such as *depth coherence*\textsuperscript{19} and *invisibility coherence*\textsuperscript{1} may also be adapted. The technique should likewise be extensible to more general surfaces as done for scanline by Lane, Carpenter, Blinn and Whitted.\textsuperscript{11} The combination of polygonal objects using the *regularized set operations* of *constructive solid geometry* as done by Atherton\textsuperscript{2} with scanline might also be possible with AEL.

The remaining sections of this paper are organized as follows. In section 2, we present the classification process of the vertices of a polygon. The general concerns in filling a polygon are addressed in Section 3. The two filling algorithms are presented in Sections 4 and 5. Implementation and performance of the algorithms are discussed in Section 6. The possibility of parallelization is discussed in Section 7. Finally, Possible directions for future work are presented in Section 8.

2. Classification of Vertices

Vertices of the polygon are classified in two ways. First, each vertex is classified as either *SUCC_HORIZON*, *PRED_HORIZON*, or *NO_HORIZON*, based on whether it is part of a horizontal edge. This classification is called the edge type classification. Each vertex is then classified as either a *START*, *STOP*, *CONTINUE_PRED*, *CONTINUE_SUCCE*, *MERGE*, or *SPLIT* vertex, based on the action that needs to be taken when that vertex is reached as the polygon is filled. This classification is called the vertex type classification. Both classifications depend on the representation of the polygon.

A polygon is represented by a sequence of exterior vertices followed by a set of disjoint sequences of interior vertices. Each sequence of interior vertices bounds a hole of the polygon. The exterior vertices are listed in counterclockwise order while the interior vertices are listed in clockwise order. Boundaries can intersect at their vertices, but interior
boundaries are not allowed to intersect the exterior boundary as they would then be viewed as part of the exterior boundary.

Each vertex has a record structure which includes its $x$ and $y$ coordinates, pointers to its predecessor and successor vertices, edge type classification, vertex type classification, an index to a sorted array of vertices to indicate the next vertex in sorted order. For the merge/split algorithm two additional parameters are necessary to handle the merging of multiple segments.

typedef struct {
    int  $x$, $y$;
    vertex *PredVert, *SuccVert;
    vertex *LeftVert;
    int  VertexType, EdgeType;
    edge  *MergeEdge;
    int  NextVert;
} vertex;

Edges of the polygon are not explicitly listed, but are implicitly defined by consecutive vertices of the polygon. A record structure for an edge will be created only when the algorithm begins the filling process of the region bounded by the edge. The record structure contains starting and ending vertices of the edge, current $x$ and $y$ values for the current scanline and, for the merge/split version, a flag to indicate if the edge has been initialized. The AEL version requires a pointer to the next edge in the active edge list.

typedef struct {
    vertex *StartVert, *EndVert;
    int  $x$, $y$;
    int  Begen;
    edge *next;
    int  d, $d$, incr, incrp, octant;
} edge;

The following terminology and notation are needed for the rest of the presentation. Given a vertex $v$ of the polygon, the predecessor vertex and the successor vertex in the representation of the polygon are denoted by $PredVert(v)$ and $SuccVert(v)$, respectively. The edge between $v$ and its predecessor vertex is called the predecessor edge and is denoted by $PredEdge(v)$. The edge between $v$ and its successor vertex is called the successor edge and is denoted by $SuccEdge(v)$. It is assumed that adjacent edges of the polygon are not collinear.

2.1 Edge Type Classification

The edge type of a vertex is classified as $SUCC\_HORIZON$ if the successor edge of the vertex is horizontal. If the predecessor edge of the vertex is horizontal, the edge type of the vertex is $PRED\_HORIZON$. If neither holds, the edge type is classified as $NO\_HORIZON$. The purpose of the edge type classification is to make the vertex type classification and the filling process easier. The classification can be easily performed by comparing the $y$ coordinates of adjacent vertices.
2.2 Vertex Type Classification

The vertex classification is based on the action that should be taken when a vertex is reached in the filling process. The possible actions include: starting a new segment, ending a segment, splitting a segment into two (or, more) segments, merging two (or, more) segments, and continuation of a current segment with a different edge. We examine the simple case first, i.e., when the vertices are on the exterior boundary of the polygon and when there is no horizontal edge present (Figure 1). We assume the polygon is filled from the bottom of the screen to the top.

A vertex \( v \) is either a \textit{START} vertex or a \textit{SPLIT} vertex if both the successor vertex and predecessor vertex are above \( v \). \( v \) is classified as a \textit{START} vertex if the predecessor vertex is to the left of the line that contains \( v \) and its successor vertex (Figure 1(a)). A \textit{START} vertex represents the beginning of a segment to be filled. \( v \) is classified as a \textit{SPLIT} vertex if the predecessor is to the right of the line that connects \( v \) and its successor vertex (Figure 1(b)). A \textit{SPLIT} vertex will cause a split in the current segment. The algorithm will also handle the more general case of splitting a region into more than two regions.

![Figure 1. Vertex classification of simple cases.](image)

If both the successor and predecessor vertices are below \( v \), \( v \) is either classified as \textit{STOP} or \textit{MERGE}. \( v \) will be classified as a \textit{STOP} vertex if the successor vertex is to the left of the line that contains \( v \) and its predecessor vertex (Figure 1(c)). A \textit{STOP} vertex indicates that the filling process of the current segment should stop when the vertex is reached. If the successor is to the right of the line that contains \( v \) and its predecessor vertex, two segments need to be merged into a single segment beyond \( v \) (Figure 1(d)). Therefore, it is classified as a \textit{MERGE} vertex. The algorithm will handle the general case of more than two regions being merged.

If either the successor vertex or the predecessor vertex is above \( v \) and the other is below \( v \), \( v \) is classified as a \textit{CONTINUE} vertex. A \textit{CONTINUE} vertex indicates that the filling
process of the region is to be continued, but one of the edges bounding the region should be replaced with a new edge. \( v \) is classified as a \textit{CONTINUE\_PRED} vertex if the new edge connects \( v \) with its predecessor (Figure 1(e)). This configuration occurs when the successor vertex is below \( v \) and the predecessor is above. \( v \) is classified as \textit{CONTINUE\_SUCC} if the new edge connects \( v \) with its successor (Figure 1(f)). This classification is detected when the successor vertex is above \( v \) and the predecessor is below.

The classification of vertices with an adjacent horizontal edge is performed in the same manner as when one is not present except that either the predecessor vertex becomes the predecessor of the predecessor for comparison purposes or the successor vertex becomes the successor of the successor. If the edge type of a vertex \( v \) is \textit{PRED\_HORIZON} then the predecessor of the predecessor (labelled PP) is compared with \( v \) and with the successor vertex. If the edge type of a vertex \( v \) is \textit{SUCC\_HORIZON} then the successor of the successor (labelled SS) is compared with \( v \) and with the predecessor vertex. Thus, the vertex \( v \) in Figure 2(a) is classified as a \textit{START} vertex. The other cases are performed in a similar manner (Figure 2). The cases shown in Figure 2 parallel to that of Figure 1. Note that when a horizontal edge is present, both endpoints of the edge will have the same vertex type classification.

![Figure 2. Vertices with horizontal adjacent edges.](image)

The classification of interior vertices is also performed in a similar manner except that the vertices are input in clockwise order rather than counterclockwise. Consequently,

The classification of vertices of a polygon with an hole is shown Figure 3. The complete algorithm is shown in the figure \textit{Procedure classify\_vertices}.

\begin{procedure}
\textbf{Procedure: classify\_vertices}
\end{procedure}
for each vertex \( v \) do

if (SuccVert(\( v \), \( y = v.y \)) then

\( v.\text{EdgeType} = \text{SUCCE_HORIZON} \)
\( v' = \text{PredVert}(v) \)
\( v = \text{SuccVert}(\text{SuccVert}(v)) \)

else if (PredVert(\( v \), \( y = v.y \)) then

\( v.\text{EdgeType} = \text{PRED_HORIZON} \)
\( v' = \text{PredVert}(\text{PredVert}(v)) \)
\( v = \text{SuccVert}(v) \)

else

\( v.\text{EdgeType} = \text{NO_HORIZON} \)
\( v' = \text{PredVert}(v) \)
\( v'' = \text{SuccVert}(v) \)

if \( (v'.y > v.y) \) then

if \( (v''.y < v.y) \) then

\( v.\text{VertexType} = \text{CONTINUE_PRED} \)

else if (((\( v.\text{EdgeType} = \text{NO_HORIZON} \)) and (\( v' \) is to the right of line \( vv'' \))) or
((\( v.\text{EdgeType} = \text{PRED_HORIZON} \)) and (\( v. x < \text{PredVert}(v).x \))) or
((\( v.\text{EdgeType} = \text{SUCCE_HORIZON} \)) and (\( v. x > \text{SuccVert}(v).x \))) then

\( v.\text{VertexType} = \text{SPLIT} \)

else

\( v.\text{VertexType} = \text{START} \)

else

if \( (v''.y > v.y) \) then

\( v.\text{VertexType} = \text{CONTINUE_SUCCE} \)
else if (((v.EdgeType = NO_HORIZON) and (v' is to the left of line vv')) or
((v.EdgeType = PRED_HORIZON) and (v. x > PredVert(v). x)) or
((v.EdgeType = SUCC_HORIZON) and (v. x < SuccVert(v). x))) then
  v.VertexType = MERGE
else
  v.VertexType = STOP

Procedure input_and_classify_vertices

2.3 Additional Pre-Processing Steps

In addition to classifying the vertices as described above, the vertices are sorted by x and y coordinates to serve as (temporary) stopping points for the regions. When a region is begun, it is filled until a stopping point is reached. In the merge/split version, in order to not have to determine which points fall within a region the next point in the y direction is chosen as the stopping point for the region. When it is reached it is tested to see if filling the region should really be stopped. If not, the next stopping point is obtained from the sorted vertex list and the filling loop is continued. In the AEL version, the sorted list of vertices becomes the list of stopping points for the algorithm with adjustments made to the active edge list when each vertex in the sorted list is encountered.

In addition, for the merge/split version to allow for parallel processing, the START vertices are stored in a linked list in the vertex input phase of the program. In this way, independent regions can be started simultaneously without searching through all the vertices.

3. Filling a Region

Filling a region is initiated from a vertex labelled START or, in the case of the merge/split algorithm, recursively from the region filling routine in the case of a new region created by a MERGE or SPLIT vertex. CONTINUE_PRED and CONTINUE_SUCCE vertices merely modify one (or both) of the edges and continue.

3.1 Beginning a Region

If a new region is begun from a START vertex, the two edges created are the edge from the current (START) vertex to its successor and the edge from the current vertex to its predecessor. In case the vertex is either PRED_HORIZON or SUCC_HORIZON, a horizontal fill operation is done to the predecessor (or successor) vertex and the second edge for the region is created from the predecessor (or successor) vertex and its predecessor (or successor). Once the two edges are created, they need to be initialized.

Initializing a new edge consists of storing the starting coordinates of the edge as the current coordinates, storing the beginning and ending vertex numbers, and calculating the octant and increments for the edge for use in the rasterization algorithm which is a standard integer midpoint technique. For comparison purposes the rasterization was also done with a floating point inverse slope technique.
3.2 Scan Conversion of Edge Line Segments

When a region bounded by two edges is being filled, the actual filling takes place one scan line at a time. Therefore, the edges are scan converted one step in the y direction at a time. The algorithm used to do the line scan conversion needs to be incremental so that linear interpolation of z-value, color, intensity, etc. can be done one step at a time as well. Since it is possible that the next pixel generated does not change y value, the algorithm may need to be called more than once to assure that the next edge pixel to be used has a new y value.

A standard midpoint line algorithm from Foley, et. al. as originally presented by Pitteway and modified by Van Aken is used here. More recent algorithms were investigated, particularly that of Rokne, Wyvill and Wu but they present no advantage for this application. Rokne et al's algorithm, for example, takes advantage of symmetry considerations and uses a double step technique. The symmetry considerations allow the second half of the line to be drawn from the first half with very little additional effort. However, in our case, the algorithm may not generate the second half of the line during the filling of the current region due to intervening vertices that cause a merge or split operation. Also, since we are scan converting two edges at the same time, and the ending vertex of the two edges will not, in general, have the same y coordinate, the filling of the edges from the opposite end would not be synchronized in y value as required.

In this application, since the edges are being traversed and rasterized in the positive y direction, only octants 1, 2, 3 and 4 are possible. The initial value of "d", the decision variable, and the positive and negative increment values are calculated and stored with the edge. If the polygons to be filled had additional attributes such as z-value, surface normal direction, color and intensity values, etc., they would be stored with the edge and increments would be calculated so that linear interpolation could be used to obtain these values after each step in the rasterization of the edge.

3.3 Filling Individual Scan Lines Between Edges

Each scan line is filled by setting the pixels between the two edges to the appropriate values. Since the values of color, intensity, etc. are available for the given y value on the edges, these values can also be interpolated across the scan line. The two algorithms are described below.

4. Algorithm AEL

The AEL algorithm fills a polygon in much the same fashion as the standard scanline method. A list of bounding edges, called an active edge list (AEL), for the regions to be filled is maintained. However, the bucket-sorted edge table is not required. For each vertex of the sorted vertex list (SVL) the algorithm modifies the active edge list as in a plane sweep approach and fills the regions bounded by the edges contained in the AEL to the next point in the SVL. The AEL is modified by adding, removing or replacing edges based on the vertex type. The process is repeated until the last point in SVL is reached.

If a START vertex is reached, it represents the beginning of a new region and the bounding edges are added to the AEL. These edges are the predecessor and successor edges of the
START vertex.

A SPLIT vertex indicates the region contains a vertex which splits the region into two regions. Bounding edges of the new regions are bounding edges of the original region and predecessor edge and successor edge of the SPLIT vertex. Therefore, two edges are added to the AEL. If two SPLIT vertices have the same y coordinate, they will appear consecutively in the SVL. After the first pair of edges is added, the second SPLIT vertex will cause the AEL to again be modified before any of the region is filled. Similar considerations apply to vertices that cause a multiple merge.

A MERGE vertex indicates that two regions which share the MERGE vertex should be joined into a single region. Two of the bounding edges of these regions are no longer required and are removed from the AEL.

When a STOP vertex is reached, it represents the end of the filling process for a region and the bounding edges of this region are removed from the AEL.

A CONTINUE vertex means the filling process of the region containing the vertex continues but one of the bounding edge is replaced with a new edge. This edge is the predecessor edge of a CONTINUE_PRED vertex or the successor edge of a CONTINUE_SUCCE vertex. Both edges may terminate with a CONTINUE vertex at the same y value. If so, the situation is handled as outlined above for the SPLIT case.

The complete algorithm is presented in the figure Procedure AEL. Note that if two vertices are the endpoints of a horizontal edge, they will appear consecutively in the SVL and only the vertex with the smaller x coordinate will be processed. The fill_segments algorithm is presented in the figure Procedure fill_segments.

The AEL is maintained in sorted order. Therefore, the routines add_edges must find the proper location for the edges to be added. Since the added edges will be adjacent in the modified list, only one search is necessary. The routine remove_edges will delete an adjacent pair of edges from the AEL after finding the first of the pair. The routine replace_edge will remove one edge and add another at the same location in the linked list. Therefore, in either case, only one search on necessary. The routines add_edges and replace_edge will call the initialize_edge routine for each edge before it is inserted into the AEL.

---

Procedure: AEL

set AEL to empty
i = 1
while (i ≤ n - 1) do
   case (v[i].VertexType) of
      START:
         e1 = PredEdge(v[i])
         x1 = v[i].x
         if (v[i].EdgeType = SUCC_HORIZ) then i = i + 1
         e2 = SuccEdge(v[i])
         x2 = v[i].x
---
fill_pixels(x₁, x₂, v[i].y)
add_edges(AEL, e₁, e₂)

MERGE:
e₁ = PredEdge(v[i])
x₁ = v[i].x
if (v[i].EdgeType = SUCCE HORIZON) then i = i + 1
e₂ = SuccEdge(v[i])
x₂ = v[i].x
fill_pixels(x₁, x₂, v[i].y)
remove_edges(AEL, e₁, e₂)

SPLIT:
e₁ = SuccEdge(v[i])
x₁ = v[i].x
if (v[i].EdgeType = PRED_HORIZON) then i = i + 1
e₂ = PredEdge(v[i])
x₂ = v[i].x
fill_pixels(x₁, x₂, v[i].y)
add_edges(AEL, e₁, e₂)

STOP:
e₁ = SuccEdge(v[i])
x₁ = v[i].x
if (v[i].EdgeType = PRED_HORIZON) then i = i + 1
e₂ = PredEdge(v[i])
x₂ = v[i].x
fill_pixels(x₁, x₂, v[i].y)
remove_edges(AEL, e₁, e₂)

CONTINUE_SUC:
if (v[i].EdgeType = NO_HORIZON) then
e₁ = PredEdge(v[i])
e₂ = SuccEdge(v[i])
fill_pixels(v[i].x, v[i].x, v[i].y)
else if (v[i].EdgeType = PRED_HORIZON) then
e₁ = PredEdge(v[i + 1])
e₂ = SuccEdge(v[i])
fill_pixels(v[i].x, v[i + 1].x, v[i].y)
i = i + 1
else
e₁ = PredEdge(v[i])
e₂ = SuccEdge(v[i + 1])
fill_pixels(v[i].x, v[i + 1].x, v[i].y)
i = i + 1
replace_edge(AEL, e₁, e₂)

CONTINUE_PRED:
if (v[i].EdgeType = NO_HORIZON) then
e₁ = SuccEdge(v[i])
e₂ = PredEdge(v[i])
fill_pixels(v[i].x, v[i].x, v[i].y)
else if (v[i].EdgeType = SUCC_HORIZON) then
  e1 = SuccEdge(v[i+1])
  e2 = PredEdge(v[i])
  fill_pixels(v[i].x, v[i+1].x, v[i].y)
  i = i+1
else
  e1 = SuccEdge(v[i])
  e2 = PredEdge(v[i+1])
  fill_pixels(v[i].x, v[i+1].x, v[i].y)
  i = i+1
  replace_edge(AEL, e1, e2)
  fill_segments(AEL, v[i+1])
  i = i+1

Procedure AEL

Procedure: fill_segments(AEL, v)

initialize e.y and e.x for each edge in AEL if not already done
while (current raster line y < v.y - 1) do
  for each pair of edges e and e' in AEL do
    rasterize e until e.y changes
    rasterize e' until e'.y changes
    fill_pixels(e.x, e'.x, e.y)

Procedure fill_segments(AEL, v)

5. Algorithm merge/split

The merge/split algorithm fills regions individually rather than filling all regions for a given scanline at a time, as is done by scanline or AEL as described above. If a parallel processor is used, each region denoted by a START vertex can be started independently. When splits occur, two calls are made to the fill_segment routine (perhaps on different processors) to fill the two regions independently. For a sequential processor version, the separate regions are filled by calling the fill_segment routine for the second region after the first is complete in the case of multiple START vertices or by two consecutive recursive calls in the case of SPLIT vertices. One call to the fill_segment procedure fills the region between exactly two edges. Additional procedure calls are necessary to fill additional regions between other pairs of edge. This approach results in additional procedure call overhead, but also allows for the filling process to be carried out in parallel.

When a region is being filled, the y value of the next vertex in the SVL is used as an stopping point for the filling process. When this y value is reached, the next vertex in the SVL is examined to see if it lies on one of the edges of the current region or, in case it is a SPLIT vertex, if it lies inside the region bounded by the current pair of edges. If such a SPLIT
vertex does not lie in the region, the filling continues until the next vertex is reached. The inclusion of the \textit{SPLIT} vertex in the region being filled is not checked earlier because the \(x\) coordinates of the edges for the current \(y\) coordinate are available here, but would have to be generated specially if the \textit{SPLIT} vertex were to be labelled as inside or outside the region earlier. If the \textit{SPLIT} vertex lies in the region, two recursive calls to \textit{fill\_segment} are made, one for each of the new regions.

If the \textit{stopping point} is a \textit{STOP} vertex, the segment filling routine terminates. If the \textit{stopping point} is a \textit{CONTINUE} vertex, the edge terminated is replaced with the new edge and the filling loop continues.

The most difficult case arises when the \textit{stopping point} is a \textit{MERGE} vertex. Since the regions are being filled independently, possibly in parallel, and a \textit{MERGE} vertex indicates that two regions are to be combined, the region that is finished first must "wait" until the other region is filled before filling can continue. Thus, the non-terminating edge of the first region is associated with the \textit{MERGE} vertex (\textit{MergeEdge}) until the second edge is ready. The case can be more complicated if more than one \textit{MERGE} vertex occurs at the same \(y\) coordinate. Here the "middle" region will have no continuing edges, but could be the first region to be processed. To handle the case of multiple merge vertices at the same \(y\) coordinate, the continuing edge of a merged region is always associated with the leftmost \textit{MERGE} vertex. Thus, when the "middle" region terminates with a \textit{MERGE} vertex at the end of each edge, a pointer is placed at the rightmost vertex pointing to the leftmost vertex. When the filling of the rightmost region is finished, the algorithm either associates the continuing edge with the rightmost vertex (if the "middle" region has not been filled) or uses the pointer to the left vertex of the middle region to find the second edge of the merged region (if the "middle" region has been filled). Figure 4 presents an example of this situation.

The algorithm is presented in the figure \textit{Procedure mergesplit}. The procedures \textit{fill\_segment}, \textit{split}, and \textit{merge} required in the algorithm are shown in figures \textit{Procedure fill\_segment}, \textit{Procedure split}, and \textit{Procedure merge}, respectively.

Note that in the procedure \textit{mergesplit} the \textit{stopping vertex} for the \textit{START} vertex \(v\) is set to \textit{NextVert(SuccVert(v))} when its edge type is \textit{SUCC\_HORIZON}. This is due to the fact that interior vertices are not allowed to lie on exterior boundary. Otherwise, we might miss some of the \textit{stopping vertices} such as \(v''\) in Figure 5.

---

\textbf{Procedure: mergesplit}

\begin{verbatim}
for each vertex \(v\) in the sorted list of START vertices do
  if (\mbox{\textit{v.EdgeType} = \textit{PRED\_HORIZON}}) then
    \(e_1 = \textit{PredEdge(v)}\)
  else
    \(e_2 = \textit{SuccEdge(SuccVert(v))}\)
  \mbox{\textit{v} = NextVert(SuccVert(v))}
\end{verbatim}
Figure 4. Multiple MERGE vertices and SPLIT vertices.

\[ v' = \text{NextVert}(v) \]
\[ \text{fill_segment}(v, e_1, e_2, v') \]

**Procedure merge/split**

Figure 5. An interior vertex lying between two START vertices.

**Procedure: fill_segment(v, e_1, e_2, v')**

/* e_1 is to the left of e_2 */
initialize \( e_1.x, e_1.y, e_2.x, e_2.y \) and \( e_2.x \) if not already done

while \((e_1.y < v'.y)\) do
  rasterize \( e_1 \) until \( e_1.y \) changes
  rasterize \( e_2 \) until \( e_2.y \) changes
fill_pixels(e_1, x, e_2, x, e_1, y)
if (v = EndVert(e_1) or PredVert(EndVert(e_1))) then
  if (v.EdgeType <> NO_HORIZON) then
    fill to predecessor or successor as appropriate
    case (v.VertexType) of
      STOP:
        return
  CONTINUE_PRED:
    if (v.EdgeType = NO_HORIZON) then
      e\textsuperscript{'}_2 = PredEdge(v)
      v = NextVert(v)
    else if (v.EdgeType = PRED_HORIZON) then
      e\textsuperscript{'}_2 = PredEdge(PredVert(v))
      v = NextVert(PredVert(v))
    else
      e\textsuperscript{'}_2 = PredEdge(v)
      v = NextVert(SuccVert(v))
    fill_segment(v, e\textsuperscript{'}_1, e_2, v)
  MERGE:
    if (v.y = EndVert(e_2).y) then
      LeftVert(EndVert(e_2)) = v
    if (MergeEdge(EndVert(e_2)) = NULL) then
      merge(v, MergeVert(EndVert(e_2)))
    else
      merge(v, e_2)
    return
  else if (v = EndVert(e_2) or SuccVert(EndVert(e_2))) then
    if (v.EdgeType <> NO_HORIZON) then
      fill to predecessor or successor as appropriate
    case (v.VertexType) of
      CONTINUE_SUCCE:
        if (v.EdgeType = NO_HORIZON) then
          e\textsuperscript{'}_2 = SuccEdge(v)
          v = NextVert(v)
        else if (v.EdgeType = SUCC_HORIZON) then
          e\textsuperscript{'}_2 = SuccEdge(SuccVert(v))
          v = NextVert(SuccVert(v))
        else
          e\textsuperscript{'}_2 = SuccEdge(v)
          v = NextVert(PredVert(v))
        fill_segment(v, e\textsuperscript{'}_1, e_2, v)
      MERGE:
        merge(v, e_1)
        return
    else if ((v.VertexType = SPLIT) and (v is between e_1, x and e_2, x)) then
      split(e_1, e_2, v)
    else
      return
fill_segment(v', e_1, e_2, NextVert(v'))

Procedure fill_segment(v, e_1, e_2, v')

Procedure: split(e_1, e_2, v)

swap e_1 and e_2 if e_2.x < e_1.x
if (v.EdgeType = NO_HORIZON) then
  e_3 = SuccEdge(v)
  e_4 = PredEdge(v)
  v' = NextVert(v)
else /* v.EdgeType = PRED_HORIZON */
  fill_pixels(v.x, PredVert(v), x, v.y)
  e_3 = SuccEdge(v)
  e_4 = PredEdge(PredVert(v))
  v' = NextVert(PredVert(v))
fill_segment(v, e_1, e_3, v')
fill_segment(v, e_4, e_2, v')

Procedure split(e_1, e_2, v)

Procedure: merge(v, e)

if (MergeEdge(v) ≠ NULL) then
  fill_segment_2(v, e, MergeEdge(v), NextVert(v'))
else if (LeftVert(v) ≠ NULL) then
  merge(LeftVert(v), e)
else
  MergeEdge(v) = e

Procedure merge(v, e)

6. Performance Comparison

For a new algorithm to be presented and accepted which performs an old task such as filling a polygon, it must have some advantages in terms of flexibility, space overhead or speed. The algorithms presented here have been demonstrated to be very flexible since they will fill arbitrary simple polygons. The approaches require a minimal amount of storage for each vertex and only requires storage for edges that are currently being used in the filling process.

As far as speed is concerned, the algorithms have been tested on filling 2D polygons by repeated calculations for a set of polygons of varying complexity. The calculations necessary to fill the polygons were done on a variety of processors and repeated in a simple loop 1000 times to achieve timing accuracy. The polygons tested included: poly1 (5-sided polygon), poly2 (7-sided polygon with holes parallel to sides), poly3 (12-sided polygon with a 15-sided
hole), poly4 (74-sided polygon), poly5 (7-sided hourglass), poly6 (10-sided hourglass), and poly7 (112-sided polygon). The processors used were 80386 (MSDOS), 80386 (Sequent DYNIX), VaxStation 2000 Ultrix, DecStation 3100 Ultrix with MIPS R2000, SparcStation 1+ and Sun 3/60. The merge/split and AEL algorithms were tested both using the integer rasterization algorithm (I) described above and a simple floating point calculation (F) using inverse slope. In order to remove differences in actual display speed, the timing measurements were for code which did not actually generate a screen display, but calculated the coordinates of lines to be drawn on the screen.

The results of the above testing is presented in Tables 1-7. Table 2 presents the data for the same machine as in Table 1 when the polygons are not actually displayed. The comparisons shown in Tables 3-7 are all under UNIX. The data indicates that scanline is always slower than either merge/split or AEL for polygons with a small number of sides, but is faster on most machines for polygons with many sides (74 in our data). This is due to the classification (sorting) of vertices required in our algorithms. Merge/split is faster than AEL for polygons with a small number of sides on all machines. Integer versions of merge/split and AEL are faster than floating point versions for 80386 and 68020 machines even when the 80386 machines have floating point coprocessors. Floating point versions are faster than integer versions for polygons with a small number of sides on VAX and RISC (MIPS and SPARC) architectures.

The AEL algorithm has been used for 3D z-buffered implementation in a test system which generates large numbers of quadrilaterals, triangles and random polygons on a RISC System/6000 machine. Figures 6-8 summarize the test results. The algorithm is faster than the standard z-buffer routine and competitive with scanline technique for small numbers of polygons. When the number of polygons is large, the algorithm becomes faster than scanline (1200-1500 for triangles/quadrilaterals and 200 for random polygons). This is not surprising since, when a large number of polygons overlap, the maintenance needed to update the scanline’s active edge list and active polygon list degrades and results in worse performance. Note that our algorithm provides a smarter traversing of the polygons than the outward-from-center-line approach proposed by Pineda as it does not traverse any point outside the polygon.

<table>
<thead>
<tr>
<th>Poly</th>
<th>M/S(I)</th>
<th>M/S(F)</th>
<th>AEL(I)</th>
<th>AEL(F)</th>
<th>Scanline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.093</td>
<td>.098</td>
<td>.097</td>
<td>.099</td>
<td>.122</td>
</tr>
<tr>
<td>2</td>
<td>.143</td>
<td>.153</td>
<td>.143</td>
<td>.160</td>
<td>.275*</td>
</tr>
<tr>
<td>3</td>
<td>.209</td>
<td>.236</td>
<td>.214</td>
<td>.242</td>
<td>.379*</td>
</tr>
<tr>
<td>4</td>
<td>.506</td>
<td>.577</td>
<td>.499</td>
<td>.583</td>
<td>.681</td>
</tr>
<tr>
<td>5</td>
<td>.050</td>
<td>.055</td>
<td>.055</td>
<td>.060</td>
<td>.093</td>
</tr>
<tr>
<td>6</td>
<td>.209</td>
<td>.220</td>
<td>.214</td>
<td>.225</td>
<td>.280</td>
</tr>
<tr>
<td>7</td>
<td>.335</td>
<td>.430</td>
<td>.329</td>
<td>.443</td>
<td>.495</td>
</tr>
</tbody>
</table>

Table 2. Polygons calculated only on 386/25C (MSDOS and TurboC):

<table>
<thead>
<tr>
<th>Poly</th>
<th>M/S(I)</th>
<th>M/S(F)</th>
<th>AEL(I)</th>
<th>AEL(F)</th>
<th>Scanline</th>
</tr>
</thead>
</table>

* Poly2 and poly3 are drawn without hole by scanline.
<table>
<thead>
<tr>
<th>Poly</th>
<th>M/S(I)</th>
<th>M/S(F)</th>
<th>AEL(I)</th>
<th>AEL(F)</th>
<th>Scanline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.0104</td>
<td>.0143</td>
<td>.0121</td>
<td>.0159</td>
<td>.038</td>
</tr>
<tr>
<td>2</td>
<td>.0259</td>
<td>.0344</td>
<td>.0291</td>
<td>.0456</td>
<td>.069*</td>
</tr>
<tr>
<td>3</td>
<td>.0353</td>
<td>.0582</td>
<td>.0395</td>
<td>.0626</td>
<td>.082*</td>
</tr>
<tr>
<td>4</td>
<td>.0655</td>
<td>.140</td>
<td>.0637</td>
<td>.140</td>
<td>.203</td>
</tr>
<tr>
<td>5</td>
<td>.0105</td>
<td>.0149</td>
<td>.0121</td>
<td>.0171</td>
<td>.044</td>
</tr>
<tr>
<td>6</td>
<td>.0160</td>
<td>.0269</td>
<td>.0181</td>
<td>.0291</td>
<td>.071</td>
</tr>
<tr>
<td>7</td>
<td>.116</td>
<td>.2296</td>
<td>.0725</td>
<td>.2213</td>
<td>.231</td>
</tr>
</tbody>
</table>

Table 3. Sequent (DYNEX) with 80386 processor

<table>
<thead>
<tr>
<th>Poly</th>
<th>M/S(I)</th>
<th>M/S(F)</th>
<th>AEL(I)</th>
<th>AEL(F)</th>
<th>Scanline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.015</td>
<td>.016</td>
<td>.016</td>
<td>.017</td>
<td>.044</td>
</tr>
<tr>
<td>2</td>
<td>.038</td>
<td>.048</td>
<td>.039</td>
<td>.050</td>
<td>.068*</td>
</tr>
<tr>
<td>3</td>
<td>.052</td>
<td>.067</td>
<td>.054</td>
<td>.069</td>
<td>.091*</td>
</tr>
<tr>
<td>4</td>
<td>.100</td>
<td>.175</td>
<td>.086</td>
<td>.164</td>
<td>.214</td>
</tr>
<tr>
<td>5</td>
<td>.014</td>
<td>.017</td>
<td>.015</td>
<td>.018</td>
<td>.044</td>
</tr>
<tr>
<td>6</td>
<td>.022</td>
<td>.029</td>
<td>.025</td>
<td>.032</td>
<td>.078</td>
</tr>
<tr>
<td>7</td>
<td>.166</td>
<td>.304</td>
<td>.155</td>
<td>.276</td>
<td>.243</td>
</tr>
</tbody>
</table>

Table 4. VarStation 2000 (Ultrix)

<table>
<thead>
<tr>
<th>Poly</th>
<th>M/S(I)</th>
<th>M/S(F)</th>
<th>AEL(I)</th>
<th>AEL(F)</th>
<th>Scanline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.042</td>
<td>.028</td>
<td>.048</td>
<td>.035</td>
<td>.064</td>
</tr>
<tr>
<td>2</td>
<td>.114</td>
<td>.089</td>
<td>.118</td>
<td>.097</td>
<td>.099*</td>
</tr>
<tr>
<td>3</td>
<td>.166</td>
<td>.142</td>
<td>.161</td>
<td>.137</td>
<td>.128*</td>
</tr>
<tr>
<td>4</td>
<td>.332</td>
<td>.380</td>
<td>.295</td>
<td>.312</td>
<td>.294</td>
</tr>
<tr>
<td>5</td>
<td>.044</td>
<td>.030</td>
<td>.045</td>
<td>.034</td>
<td>.066</td>
</tr>
<tr>
<td>6</td>
<td>.066</td>
<td>.054</td>
<td>.076</td>
<td>.063</td>
<td>.117</td>
</tr>
<tr>
<td>7</td>
<td>.584</td>
<td>.601</td>
<td>.503</td>
<td>.543</td>
<td>.333</td>
</tr>
</tbody>
</table>

Table 5. DEC mips R2000 (Ultrix)

<table>
<thead>
<tr>
<th>Poly</th>
<th>M/S(I)</th>
<th>M/S(F)</th>
<th>AEL(I)</th>
<th>AEL(F)</th>
<th>Scanline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.00313</td>
<td>.00156</td>
<td>.00352</td>
<td>.00215</td>
<td>.00586</td>
</tr>
<tr>
<td>2</td>
<td>.00459</td>
<td>.00547</td>
<td>.00840</td>
<td>.00547</td>
<td>.00947*</td>
</tr>
<tr>
<td>3</td>
<td>.0109</td>
<td>.00859</td>
<td>.012</td>
<td>.00918</td>
<td>.0132*</td>
</tr>
<tr>
<td>4</td>
<td>crash</td>
<td>.0270</td>
<td>.0253</td>
<td>.0249</td>
<td>.0290</td>
</tr>
<tr>
<td>5</td>
<td>.00312</td>
<td>.00156</td>
<td>.00352</td>
<td>.00225</td>
<td>.00654</td>
</tr>
<tr>
<td>6</td>
<td>.00469</td>
<td>.00274</td>
<td>.00527</td>
<td>.00361</td>
<td>.0113</td>
</tr>
<tr>
<td>7</td>
<td>crash</td>
<td>.0539</td>
<td>.0465</td>
<td>.0466</td>
<td>.0314</td>
</tr>
</tbody>
</table>

Table 6. Sun Sparstation 1+

<table>
<thead>
<tr>
<th>Poly</th>
<th>M/S(I)</th>
<th>M/S(F)</th>
<th>AEL(I)</th>
<th>AEL(F)</th>
<th>Scanline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.0043</td>
<td>.0020</td>
<td>.0040</td>
<td>.0025</td>
<td>.0095</td>
</tr>
<tr>
<td>2</td>
<td>.0100</td>
<td>.0070</td>
<td>.0110</td>
<td>.0075</td>
<td>.0133*</td>
</tr>
<tr>
<td>3</td>
<td>.0145</td>
<td>.0110</td>
<td>.0165</td>
<td>.0125</td>
<td>.0183*</td>
</tr>
<tr>
<td>4</td>
<td>.0353</td>
<td>.0358</td>
<td>.0303</td>
<td>.0345</td>
<td>.0425</td>
</tr>
<tr>
<td>5</td>
<td>.0040</td>
<td>.0023</td>
<td>.0045</td>
<td>.0030</td>
<td>.0093</td>
</tr>
<tr>
<td>6</td>
<td>.0058</td>
<td>.0043</td>
<td>.0073</td>
<td>.0048</td>
<td>.0158</td>
</tr>
<tr>
<td>7</td>
<td>crash</td>
<td>.0728</td>
<td>.0605</td>
<td>.0655</td>
<td>.0483</td>
</tr>
</tbody>
</table>

Table 7. Sun 3/60 (68020 processor)

<table>
<thead>
<tr>
<th>Poly</th>
<th>M/S(I)</th>
<th>M/S(F)</th>
<th>AEL(I)</th>
<th>AEL(F)</th>
<th>Scanline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.0167</td>
<td>.0537</td>
<td>.0149</td>
<td>.0558</td>
<td>.1871</td>
</tr>
<tr>
<td>2</td>
<td>.0442</td>
<td>.1329</td>
<td>.0483</td>
<td>.1533</td>
<td>.2987*</td>
</tr>
<tr>
<td>3</td>
<td>.0621</td>
<td>.2037</td>
<td>.0650</td>
<td>.1992</td>
<td>.3942*</td>
</tr>
<tr>
<td>4</td>
<td>.1308</td>
<td>.3791</td>
<td>.1254</td>
<td>.3725</td>
<td>.9112</td>
</tr>
<tr>
<td>5</td>
<td>.0158</td>
<td>.0542</td>
<td>.0183</td>
<td>.0566</td>
<td>.1883</td>
</tr>
<tr>
<td>6</td>
<td>.0262</td>
<td>.0946</td>
<td>.0288</td>
<td>.1029</td>
<td>.3429</td>
</tr>
<tr>
<td>7</td>
<td>.222</td>
<td>.493</td>
<td>.219</td>
<td>.537</td>
<td>1.012</td>
</tr>
</tbody>
</table>
7. Possibilities for Parallelization

The *merge/split* algorithm presented in this paper has been implemented in a purely sequential fashion. There are, however, several approaches that could be taken to convert the algorithm to operate on parallel hardware. The approaches used would depend on the type of processing system and, especially, on the desired complexity of special-purpose processors that might be designed to implement the algorithm, or parts of it.

At a large-scale level, the algorithm could be adapted to parallel execution by starting all segments that begin with a START vertex for a given polygon at the same time. The processes would need to interact in order to form new regions when a *MERGE* vertex was found.

It would also be possible to implement in hardware a processor which would use linear interpolation to fill the pixels of a scanline. A processor to scan convert each edge would also be possible. Then, two processors could scan convert the edges and generate all the information necessary to define a region of a scan line to be filled which could then be sent to a third processor to generate the actual pixel values.

If a low-level hardware implementation were desired, processors could be designed so as to pass the vertices through a classification pipeline.

Some of the above parallel processing strategies could also be applied to the *AEL* version.

8. Possible Directions for Future Work

A possible direction in improving the performance of the algorithms is to use the *run length slice algorithm*\(^3\) in rasterizing the edges of the polygon. This technique is a structural method rather than incremental. All pixels for a line are calculated at the same time in the form of the run length slices. Some of the calculations for an edge would be for parts of the edge beyond an intervening vertex. A possible approach is to do this calculation in the edge initialization phase and store the results with the edge. The segments are then rasterized and filled between stopping vertices using this information by maintaining a pointer to current location in the run. However, calculations of color, intensity, z-value, etc. would still need to be linearly interpolated at each step along the edge.

Another possibility is to use our present technique to determine end points on a given scan line and then use Pineda’s approach\(^4\) to calculate groups of pixels on the scan line in parallel. Or, perhaps, use Pineda’s approach for regions of the plane determined by vertex classification, i.e., from present scan line until stopping vertex use *edge function* approach, perhaps in parallel while processor considered that region filled and continued to next region after stopping vertex.

It should also be feasible to include *clipping* as part of the algorithm so that the polygons would not need to undergo a preliminary clipping step. The clip boundaries parallel to the x axis could be incorporated into the edge structure easily as stopping points for filling with a flag set to indicate that filling should not begin until these vertices are reached. Clipping
parallel to the y axis would be more complicated and require intersection calculations of the nature needed by clipping. Pineda's clipping approach can also be employed if any of the above options were chosen.

One of the execution time comparisons that should be done is to compare the algorithm for filling general polygons with one where the polygons are first broken into triangles which are then filled. If this algorithm is efficient for filling triangles as is anticipated, it would be natural to break the polygon into triangles in a preprocessing step. The possibility of using the vertex classifications of this paper to triangulate, or decompose an arbitrary polygon into vertically-connected components will also be investigated.

References


Figure 6. Test results using various algorithms for quadrilaterals.
Figure 7. Test results using various algorithms for triangles.
AIX Risc 6000: Random complex polygons

Figure 8. Test results using various algorithms for random polygons.