CHAPTER 1 – INTRODUCTION

1.1 POINTS, EDGES, AND SURFACES

The purpose of computer graphics is the manipulation and visual display of existing or imagined objects or data with physical dimensions. The display output of “traditional” computer graphics, ignoring virtual reality environments that provide the illusion of 3 dimensions, is the 2D representation on a display device. Many geometric and topological concepts are involved through algorithms and data structures that deal with points, lines, surfaces and volumes – 0-, 1-, 2-, and 3-dimensional entities.

Consequently, data structures and related algorithms in computer graphics are oriented around points (or vertices), lines (or edges), surfaces, and volumes. Lines, surfaces, and volumes must be represented eventually by points that are involved in forming the boundary of the higher dimensional entity. However, although points are viewed geometrically as having 0-dimension, they are represented with 3 coordinates since they must exist in 3-dimensional space in order to contribute to the concept of boundary of higher dimensional objects. In practice, 4 coordinates are used to represent a point with conventional homogeneous coordinates.

It is difficult to classify problems according to their dimension since the question must be asked, “Which part of the problem is being classified?” For example, the representation of solid objects, inherently 3 dimensional, may use a collection of volume elements or a collection of 2D surfaces which bound the solid. In turn, volume elements must be bound by surfaces, surfaces must be bound by edges, and edges must be bound by points. Eventually, points must be represented
somewhere in the relevant set of data structures. Typically, elements with dimension between points and that of the relevant object are represented implicitly if not explicitly.

It is true, however, that different approaches to the same problem may use points or edges or surfaces as the focus of an algorithm which deals with edges or surfaces or volumes. The “focus” of an algorithm can be considered as the controlling factor in loops or recursion and is often used in the complexity measure of the algorithm. For example, graph algorithms may measure complexity in terms of the number of vertices or the number of edges in the graph.

As represented in the title of this research, “Point Based Approaches in Graphics,” two areas are investigated with point-based algorithms and data structures that are traditionally viewed as edge oriented.

Solids, surfaces, and edges each consist of an infinite number of points. The difference between these items of different dimension is actually the dimensionality of the connectivity relationships between the infinite number of points composing each. It is sometimes not clear what type of connectivity to consider. For example, Arthur Loeb [1] approaches polyhedra with a type of duality that on the one hand stresses the concept of a polyhedron as a set of connected items of different dimensionalities but on the other hand as a set of rigorously defined points. These approaches give either a connectivity or a symmetry point of view.

In like manner, many problems of computer graphics and related areas can be viewed from multiple perspectives. The choice of viewpoint will effect the data structures and algorithms used in the problem solution.
The remainder of this section will consist of a brief survey of some classical problems of computer graphics and the dimensionality of typical solutions.

An example application that can be approached in a number of ways is that of solid modeling. One approach to organizing solid modeling techniques includes decomposition models, constructive models, and boundary models[2]. Decomposition and constructive models are described as viewing solids as point sets with decomposition approaches discretizing the set and constructive approaches constructing the set from simpler sets. The boundary model represents a solid indirectly by forming representations of the bounding surfaces.

These boundary models can be vertex-based, edge-based, or polygon-based. A vertex-based model defines bounding faces of the surface in terms of vertices with the actual coordinates stored in a separate vertex data structure. Edge-based models are useful if curved surfaces are involved and represents a face boundary as a closed sequence of edges (loop) with vertices represented only as end points of edges. A model that needs only to represent planar faces may use a polyhedral model where the faces are stored as polygons with each polygon represented as a sequence of coordinates. A solid is represented by a collection of faces.

The data structures of the edge-based models are perhaps the most interesting. Since an edge is the “middle” layer in the point to edge to surface hierarchy, data structures often represent each explicitly. The doubly-connected-edge-list is an example of such a data structure [3]. A slightly more elaborate structure that also includes loop information in edge nodes is the winged-edge data structure [4].

Perhaps the most obvious example of a class of problems that can be solved using multiple approaches based on varying dimensionality are graph algorithms. Graphs are a source of
problems to examine in discussing the usefulness of points and edges in algorithm development since a graph consists of only points and the edges between them and is not concerned with faces or any higher dimensional structure.

As expected there are multiple data structures that can be used to represent graphs and these data structures affect the algorithms used and their efficiency [5]. The standard adjacency matrix approach represents edges by the presence of a non-zero entry in a matrix indexed by points. This data structure represents edges but is indexed by points. The other common approach to graph representation, the adjacency-list uses a list of neighbor points to represent the edges incident on a given point. Here, edges are represented by the presence of a point in a list where the individual lists are indexed by the originating point. Both data structures index edges by points. Space considerations are also important. The adjacency matrix approach requires $O(V^2)$ space while the adjacency-list structure uses $O(V + E)$ where $V$ is the number of vertices and $E$ the number of edges in the graph.

Graph searching problems are important in a number of contexts including connectivity and checking for cycles. Since the vertices are the objects being searched it is natural for algorithms to be driven by vertices. However, there must be a path used to get to a vertex so that the edges must play a key role in any search algorithm. In some applications, such as maze traversal, the paths (edges) are the important part of the result.

Depth first search using the adjacency matrix approach moves from a given vertex by examining the matrix to determine if there is an edge to each of the remaining vertices. Thus, the algorithm has complexity proportional to $V^2$. If the adjacency-list structure is used, edges from the given vertex are considered and the complexity is proportional to $V + E$. The choice of data structure, and therefore algorithm, is based on the density of the graph.
An additional example of differing approaches driven by points or edges is the minimum spanning tree problem. The priority first search approach modifies a search algorithm to consider points in an order determined by the length of an edge from the currently constructed tree to vertices not yet included. The vertices of the graph are divided into three sets: vertices contained in the current tree, fringe vertices which are waiting to be examined, and unseen vertices whose existence has not been detected by the algorithm. The basic algorithm is to move one vertex from the fringe list to the tree and to move any newly discovered vertices to the fringe list. Note that the algorithm seems to be driven by the vertices of the graph since they are the objects of consideration at each stage and all vertices must be included in the final minimum spanning tree. However, the priority queue used to chose the next direction to “search”, while consisting of a collection of vertices, is ordered based on the length of edges. This algorithm has complexity for sparse graphs proportional to \((E + V) \log V\).

Kruskal’s algorithm is an alternative approach which seeks to add edges to an existing tree by finding the shortest edge that does not introduce a cycle. The algorithm starts with a forest of trees, each consisting of a single vertex, and combines trees based on edges between them until one tree remains. An important data structure is again the priority queue but the entries are edges rather than vertices. The complexity of this algorithm is \((E \log E)\).

Other similar examples, such as shortest path, could be discussed where either points or edges are used as the driving force behind the algorithm. In the graph algorithm examples the choice of algorithm is guided by the number of edges relative to the number of vertices. In the algorithms considered in this research, this consideration is not sufficient. Surface reconstruction problems begin with a set of points but have no knowledge of the edges or the number of edges needed to construct the surface. Polygon filling is, by nature, a much different type of problem where the
A polygon consists of an infinite number of points. The problem of polygon filling is one of mapping the points inside a polygon to pixels on a display. Each pixel represents a range of points. Perhaps the problem is more accurately stated as mapping a polygon to a display and then mapping each pixel inside the mapped polygon to a set of points so that the pixel best represents the points mapped to that portion of the display.

Each of the edges that bound the polygon consists of an infinite number of points that are also mapped to the display and to pixels. The edges are bound by a pair of points each of which is mapped to a pixel. The connectivity relationships are known and include the polygon edges, the points bounding the edges, pixels on scan lines, etc.

The problem, then, is one of mapping a polygon to a set of pixels with the needed connectivity relationships known. Traditional approaches use a mapping algorithm that is driven by polygon edges. This research presents two approaches that are driven by the points that are the vertices of the polygon.

1.3 SURFACE RECONSTRUCTION

The general problem of surface reconstruction begins with a finite number of points sampled from the infinite number of points on a surface. The initial steps in the problem solution are to
choose a subset of the sampled points and to establish relationships between these points to represent the surface. A subset of the points is chosen so that the resulting mesh or surface is not too dense to be useful.

The problem is, therefore, one of establishing connectivity among points to represent a surface. Edges are often used as an intermediate step or to represent the surface as a collection of polygons. Many algorithms for surface reconstruction begin with a large set of edges and work toward a simplified surface. The approach described in this research begins with points and uses edges only to represent the reconstructed surface.
SECTION I
POLYGON FILLING

CHAPTER 2 - BACKGROUND

2.1 PROBLEM DESCRIPTION

Fast display of shaded polygons on a raster device is important in computer graphics and is becoming more so as more polygons (typically triangles) are required for high resolution display of complex objects and scenes.

The problem addressed by this research is to investigate polygon filling algorithms that are edge based in the sense that the polygon is filled between edges but that the edges (and transitions between edges) are controlled by points - the vertices of the polygon.

Specifically, two algorithms are developed which fill arbitrary simple polygons with simple holes. The algorithms are based on the classification of the vertices of the polygons and are extensible to “polygons” which have curved sides.

2.2 PREVIOUS WORK

2. 2.1 Polygon Filling

By far the most common polygon filling algorithm is the scanline algorithm [6, 7, 8, 9]. In this technique the starting and stopping points (vertices) of each edge are bucket sorted according to
the scanline on which they begin. Starting at the bottom (or top) of the display, scanlines are
processed sequentially by filling the pixels between each pair of active edges. Active edges are
those that have begun but not ended and are stored in an active edge list. For each scanline, it is
necessary to check the bucket sorted array for new edges that begin and edges that end with the
active edge list adjusted accordingly.

Three dimensional polygons are commonly handled by a scanline approach which also utilizes a
Z-buffer to determine which polygon is closest to the viewer and should, therefore, determine the
color value for each pixel [10]. Properties such as depth coherence [11] and invisibility
coherence [12] can be used. More general surfaces can also be rendered by a scanline approach
[13]. Polygonal objects can be combined using the regularized set operations of constructive
solid geometry with a scanline based algorithm [14].

It is also common to break a polygon into trapezoids or triangles before it is filled.

2.2.2 Vertex Classification

Some of the decomposition algorithms that convert a polygon to trapezoids or triangles are based
on the classification of the vertices of the polygon. Fournier and Montuno [15] 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 (above and below) 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. 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 trapezoids. A Type 3 vertex marks the end of two trapezoids and the beginning of a new trapezoid or the end of a single trapezoid.

Vertex classification was used earlier by Lee [16] and by Brassel and Fegeas [17] for filling regions on vector display devices. Lee named the Type 1 vertices regular, Type 2 stalagmitic, and Type 3 stalactitic. In both algorithms, the region was broken into trapezoids. Thus, the techniques differed from the vertex classification approach in that the classified vertices were used as a pre-processing step to break a polygon into trapezoids that were then filled.

The classification of polygon vertices has also been used in a clipping algorithm [18] where one arbitrary polygon is clipped against another arbitrary polygon. The vertices are classified as local minimum, local maximum, left intermediate, and right intermediate. Edge intersections are also classified. The algorithms presented in the dissertation could be combined with these ideas to produce a clipped polygon with vertices properly classified for filling.

As referenced in the surface reconstruction section of this work, vertex classification [19] has also been used to reduce the number of triangles in a mesh representing a surface. The process, known as decimation, makes multiple passes over a triangular mesh using local geometry and topology to decide the vertices to be removed. The local characteristics are determined by classifying the vertices as simple, complex, boundary, interior edge, or corner based on the relationship of the vertex to the triangles incident to it.
2.3 FOCUS OF THIS RESEARCH

The purpose of this research is to investigate polygon filling algorithms based on the classification of data points - the vertices of the polygon. Polygons are filled by regions rather than on a scanline basis.

Two distinct algorithms have been developed. One, named Merge/Split, fills regions independently by shading pixels between two edges. The bounding edges of a region are changed when a vertex is encountered. Some vertices cause one of the bounding edges to be replaced by another. Vertices may also indicate that a region needs to be split into two regions or that two regions should be merged into one region. Obviously, vertices also signal the beginning and end of regions.

A modification of this algorithm stores the edges in a sorted active edge list, much like the scanline algorithm. However, the algorithm is driven by vertices and the resulting edge transitions rather than individual scan lines.

Either of these algorithms can be modified to fill polygons whose sides are curved rather than straight. The required modifications are simple and are based on classifying the endpoints of the curved segments in the same way that the vertices of a polygon with straight sides are classified.
CHAPTER 3 – EDGE BASED POLYGON FILLING

3.1 FOUNDATIONS

The polygon filling algorithms presented in this work are edge based rather than scanline based in the sense that polygons are filled by regions. Some earlier algorithms, as discussed in the previous chapter, segmented polygons into regions (trapezoids) defined by beginning and ending y coordinates (scanlines) and two edges. The area to be filled in a given step is, in general, a quadrilateral, but since edge transitions occur at scanlines, there is a break in the filling process at scanlines where such transitions occur. However, the unit of area defined for filling purposes is this trapezoid rather than a single scanline as in traditional approaches.

![Figure 3.1 Two edges with area between filled](image)

A more proper description of the algorithms includes the idea that the region being filled may actually begin with two edges incident at the same point, allow the bounding edges to change during the region fill process, and end with two edges incident at the same point.
It is also possible that a region may be split into two regions or two regions may be combined into one.
The algorithms are viewed as point-driven rather than edge-driven because the beginning, ending, and transitions of regions are determined by the vertices of the polygon. The vertices are classified based on the changes they cause in the bounding edges of the region. A second reason that the algorithms are viewed as point-based is that edges are not stored, but are created from the beginning and ending vertices as needed to bound a region. The edges even then do not exist except in the form of beginning point, ending point, and increment as needed to rasterize the line that is the edge.

A region can thus be defined as the area between two edges where the edges are understood to be dynamic and defined by the classified vertices that are the end points and transition points.

Since the region filling process is interrupted to change edges based on the vertices of the polygon, the vertices can be viewed as “event points” and the resulting algorithms can be classified as “plane-sweep” algorithms [20]. As is true for plane sweep algorithms in general, the dimension of the problem is reduced by one. The 2D filling algorithm fills a region continuously until an event point is encountered. Thus, the algorithms are 1D in the vertical direction.

3.2 OVERVIEW

If regions of a polygon are to be filled, the regions are defined by two edges, and the edges are defined and changed based on the vertices that define the edges, then the crucial part of the algorithm is the classification of the vertices so that the necessary action can be taken when a vertex is encountered as an event in the filling process.
The vertices are classified as a preprocessing step. As a polygon can be defined by its vertices arranged in order around the polygon, the polygon is defined for the filling process by its classified vertices arranged in the order in which they are encountered in the filling process.

The algorithms begin filling a region, or multiple regions, by finding vertices that indicate the beginning of a region. The algorithms continue with each region until a transition is necessary with the transition determined by the classification of the vertex encountered. The process continues until all regions are finished.
CHAPTER 4 – ROLE OF VERTEX CLASSIFICATION

4.1 USE IN POLYGON FILLING

The vertices of a polygon are classified based on the action taken by the filling algorithms in determining the transformations of the edges bounding the region being filled.

A START vertex is one where a region begins. A STOP vertex is one where a region ends. Vertices may also be labeled as CONTINUE, MERGE, or SPLIT. A CONTINUE vertex occurs where a bounding edge is replaced by another. A MERGE vertex occurs where two regions can be joined into one region. A SPLIT vertex occurs when a region must be split into two regions. Since there may be multiple regions being filled at a time even if only one polygon is present, the vertices in the polygon need to be ordered. The vertices are ordered from an arbitrary starting vertex by traversing the polygon counterclockwise so that the interior of the polygon is on the left as it is traversed. Therefore each vertex has a successor and a predecessor and links are created to the adjacent vertices. CONTINUE vertices, which cause an edge transition without changing the number of regions being filled, are classified as CONTINUE_PRED or CONTINUE_SUCC to indicate which vertex is to be used in defining the new edge. Edges are thus defined as being the line between a vertex and its successor or predecessor where the choice of successor or predecessor is determined by the “left-right” relationship of the edge to the region being filled.

The algorithm fills any simple polygon, including those with internal holes. The vertices that define the hole are classified by examining them in a clockwise manner. Therefore, the region being filled is on the same side of an edge that is defined by a vertex and its successor (or predecessor) as if the vertex were on the outside of the polygon. MERGE and SPLIT vertices
that are on the hole boundary indicate the same action as if they were on the outside of the polygon.

An additional complication arises if the polygon is being filled in a vertical direction and contains horizontal edges. For example, if a polygon has a horizontal “bottom” edge, two vertices determine the beginning of a region. Similar situations exist for STOP, CONTINUE, MERGE, and SPLIT vertices which bound a polygon or a hole.

Therefore, vertices are also classified as SUCC_HORIZON (indicating that the successor vertex is connected by a horizontal edge), PRED_HORIZON (indicating that the predecessor vertex is connected by a horizontal edge), or NO_HORIZON (indicating that there is no horizontal edge incident to the vertex). It is assumed that the polygon does not contain 3 adjacent collinear vertices.
For the algorithms to be general and robust they must handle any type of polygon. In addition to horizontal edges and holes, a difficult polygon might include one (or several) MERGE or SPLIT vertices with the same y value. Some examples of difficult regions are illustrated below.

![Figure 4.2 Polygon with hole and horizontal edges](image)

STT: START  
STP: STOP  
CP: CONTINUE PREDECESSOR  
CS: CONTINUE SUCCESSOR  
SPL: SPLIT  
MGR: MERGE

**Figure 4.2 Polygon with hole and horizontal edges**

It is also possible for multiple regions to begin, end, or have transitions at the same y value. This situation is trivially supported in the case of MERGE vertices where two regions end and are replaced by a new one that is bound by two of the edges present in the two regions. Also, in the case of SPLIT vertices one region ends and two begin which are bound by the two original edges and two new edges. Of course, it is also possible that an edge transition might occur at the same location.
Multiple START or STOP vertices with the same y value indicate independent regions. These regions could be filled in parallel.

4.2 MERGE/SPLIT ALGORITHM

In the Merge/Split algorithm regions are filled beginning with a START vertex until a STOP, MERGE, or SPLIT vertex is encountered.

A MERGE vertex indicates that the current region is to merged with another. This second region may already be filled, may be in the process of being filled by another process in a parallel implementation, or may be "waiting". In any case, one of the regions will be finished and need to wait on the other region to reach the same MERGE vertex. Thus, sufficient information must be stored and associated with the MERGE vertex. It is also possible that several MERGE vertices will occur at the same vertical location so that many regions will need to be combined.

A SPLIT vertex creates two regions to be filled. These regions can be viewed as independent. In practice it is more efficient to continue one vertex as if a CONTINUE vertex had been encountered and to store the second region for filling later or as another process. It is also possible that several SPLIT vertices occur at the same vertical location so that a region will need to be split into many regions. Such events can be handled easily if the stopping points (vertices) are sorted horizontally as well as vertically. A region is split into two regions and one of these regions is again split into two regions before any filling is actually done.
It is important to note that the terminology “stopping point” refers to any vertex when it is encountered and causes an action to be taken. STOP vertices are those that are determined to indicate the end of a region.

4.3 ACTIVE EDGE LIST ALGORITHM

A modification of the Merge/Split algorithm does not treat the regions being filled independently at the scan line level. In this algorithm, all regions active for a given scan line are filled at the same time by the same process.

The AEL algorithm fills a polygon in much the same manner as the standard scanline method. A list of bounding edges, named active edge list, is maintained. However, the bucket-sorted edge table of scanline is not used. While the scanline algorithm examines the data structures for each new scanline, the AEL algorithm modifies the active edge list for each vertex encountered as in a plane sweep algorithm.

The two algorithms might be filling regions of a polygon at a given time as illustrated below.
Figure 4.3 Regions being filled independently
CHAPTER 5 – CLASSIFICATION OF VERTICES

The vertices of the polygon are classified as a preprocessing step to the filling algorithm. In addition to the Edge Type and Vertex type classifications, the vertices must be sorted in the order they will be encountered as stopping points in the filling algorithms.

The vertices are sorted by x coordinate within y coordinate to serve as stopping points for the algorithm. A stopping point is only a true event point if it lies within the region being filled, but this is not determined until the point is encountered.

Assuming that the polygon is to be filled in a vertical direction on the display, the stopping points are sorted by increasing y value. In order to properly handle events that occur at the same y value, points with the same y value are also sorted by increasing x value.

A polygon is represented by a sequence of exterior vertices followed by a (possibly empty) set of disjoint sequences of interior vertices. Each sequence of interior vertices forms the boundary of a hole in the polygon. The exterior vertices are listed in counterclockwise order while the interior vertices are listed in clockwise order. Boundaries may intersect at their vertices, but interior boundaries may not intersect the exterior boundary.

The vertices are classified by considering them in order around the boundaries. The proper Edge Type and Vertex Type can both be determined by the spatial relationship of a vertex to its successor and predecessor vertices. In the case of horizontal edges, the successor of the successor or the predecessor of the predecessor must also be considered.

Each vertex is represented by a data structure which includes its coordinates, pointers to its predecessor and successor vertices on the polygon boundary, edge type classification, vertex type
classification, and an index to a sorted array of vertices to indicate the next vertex to be considered as a stopping point during filling. Two additional fields, LeftVert and MergeEdge, are necessary to handle the merging of multiple segments at the same y coordinate. These fields are needed only for the Merge/Split algorithm.

The vertex data structure is created as the vertices are input. All the fields except the classification and sorted fields are initialized during the input phase.

The data structure for an edge exists only when the edge is bounding a region that is currently being filled. This data structure contains pointers to the starting and ending points of the edge, current coordinates for the point on the edge at the scanline being filled, a flag to indicate if the edge has been initialized, and various fields needed to rasterize the edge. The AEL algorithm also needs a pointer to the next edge in the active edge list.

5.1 EDGE TYPE

Each vertex is classified as SUCC_HORIZON, PRED_HORIZON, or NO_HORIZON based on whether it is part of a horizontal edge or not and whether the horizontal edge extends from the first (leftmost) vertex to its successor or predecessor vertex.
5.2 VERTEX TYPE

A vertex v is either a START vertex or a SPLIT vertex if both the successor vertex and the predecessor vertex are above v. A vertex v is a START vertex if the predecessor vertex is to the left of the line that contains v and its successor vertex (Figure 5-2 (a)). A vertex v is classified as a SPLIT vertex if the predecessor vertex is to the right of the line that contains v and its successor vertex (Figure 5-2 (b)).

For the parallel implementation of the Merge/Split algorithm, the vertices that are classified as START are stored in a linked list so that independent regions may be begun simultaneously.

If both the predecessor and successor vertices are below v, the vertex is either a STOP vertex or a MERGE vertex. The vertex v is classified as a STOP vertex if the successor vertex is to the left of the line that contains v and its predecessor vertex (Figure 5-2 (c)) or as a MERGE vertex if the
successor vertex is to the right of the line that contains v and its predecessor vertex (Figure 5-2 (d)).

If either the successor vertex or the predecessor vertex is above v and the other is below v, the vertex is a CONTINUE vertex. The vertex is classified as CONTINUE_SUCCE or CONTINUE_PRED depending on whether the next edge created as the region is continued is formed by the vertex and its successor or predecessor. If the successor vertex is below v and the predecessor is above v, the vertex is classified as CONTINUE_PRED (Figure 5-2 (e)). The vertex is a CONTINUE_SUCCE if the opposite case is true (Figure 5-2 (f)). In the following figure, v represents the vertex being classified, P represents its predecessor and S represents its successor.

![Figure 5.2 Vertex type classification](image)

(a) START  (b) SPLIT  (c) STOP  
(d) MERGE  (e) CONTINUE_PRED  
(f) CONTINUE_SUCCE

The classification of vertices with an adjacent horizontal edge is performed in the same manner as when such an edge is not present except that the comparison is modified slightly. To determine if the successor and predecessor vertices are above or below v, the successor of the successor (SS)
must be used rather than S if v is SUCC_HORIZON or the predecessor of the predecessor (PP) must be used rather than P if v is PRED_HORIZON.

The comparison of a successor or predecessor to an edge must also be modified. In the case where no horizontal edge is present, a vertex is classified as a START vertex if S and P are above v and P is to the left of the line from v to S. If the vertex being classified is SUCC_HORIZON, P must be to the left of the line from S to SS. The case with no horizontal edge can also be stated as requiring S to be to the right of the line from v to P. The equivalent test for the START vertex that is also PRED_HORIZON is to require S to be to the right of the line from P to PP.

The above comparisons for horizontal edges can be simplified from a point and line comparison to a coordinate comparison. If both points (P and SS or PP and S) are above v, v is a START vertex if it is SUCC_HORIZON and v is to the left of S or if it is PRED_HORIZON and v is to the right of P.

The vertices of the following figure have the same classification as their counterparts in the previous figure. The successor of the successor of v is represented by SS and the predecessor of the predecessor is represented by PP.
When a horizontal edge is present, both endpoints of the horizontal edge have the same Vertex Type. One of the vertices is of Edge Type SUCC_HORIZON while the other is PRED_HORIZON.

If the vertex being classified (and, obviously, its successor and predecessor) are part of a hole boundary rather than part of the exterior boundary, the classifications are performed in an identical manner. Since the vertices are entered in clockwise rather than counterclockwise order, the region being filled is on the same side of an edge bounding a hole as an edge on the exterior boundary. Thus the region being filled lies to the left of an edge if the edge is traversed from a vertex to its successor. Figure 4.2 represents most of the cases for a polygon with an interior hole.
In the Vertex Classification Algorithm, pv is the vertex used as the predecessor for classification purposes while sv is the vertex used as the successor.

5.3 STOPPING POINTS

When a region is being filled, stopping points are encountered. These points may or may not cause the region being filled to be modified. Some stopping points are associated with other regions. In order to not interrupt the filling process for such points, each stopping point would have to be linked to a region. In order for this to be done, the regions of the polygon must be determined before the filling process is begun. Much of the same work would need to be repeated in order to handle CONTINUE vertices. Therefore, this possibility is not practical. The order of the vertices is maintained by constructing a sorted vertex list using pointers in the vertex data structure.
for each vertex v
    sv = SuccVert( v )
    pv = PredVert( v )
    if ( sv.y = v.y )
        v.EdgeType = SUCC_HORIZON
        ssv = SuccVert( sv )
    else
        ssv = sv
        if ( pv.y = v.y)
            v.EdgeType = PRED_HORIZON
            ppv = PredVert( pv )
        else
            ppv = pv
            v.EdgeType = NO_HORIZON
    if ( ppv.y > v.y )
        if ( ssv.y < v.y ) v.VertexType = CONTINUE_PRED
        else
            if ((( v.EdgeType = NO_HORIZON) and (pv is to right of v-sv))
                or
                (( v.EdgeType = PRED_HORIZON) and (v.x < pv.x))
                or
                (( v.EdgeType = SUCC_HORIZON) and ( v.x > sv.x)))
                v.VertexType = SPLIT
            else
                v.VertexType = START
    else
        if ( ssv.y > v.y ) v.VertexType = CONTINUE_SUCC
        else
            if ((( v.EdgeType = NO_HORIZON) and (pv is to left of v-sv))
                or
                (( v.EdgeType = PRED_HORIZON) and (v.x > pv.x))
                or
                (( v.EdgeType = SUCC_HORIZON) and ( v.x < sv.x)))
                v.VertexType = MERGE
            else
                v.VertexType = STOP

Vertex Classification Algorithm
CHAPTER 6 – MERGE/SPLIT ALGORITHM

6.1 OVERVIEW

The Merge/Split algorithm derives its name from the use of vertices classified as MERGE to combine regions being filled and vertices classified as SPLIT to create new regions.

The algorithm treats regions of the polygon, each defined by a pair of bounding edges, as independent areas to be filled. The classification of the vertices that are encountered as event points drives the algorithm and causes the region to be modified.

When a region bounded by two edges is being filled, the actual calculation of the proper values for the pixels takes place one scan line at a time. Therefore, the edges are scan converted one step in the y direction at a time.

The approach used to scan convert the edge is incremental so that linear interpolation of z-value, color, intensity, etc. can be done. Since it is possible that the next pixel generated does not have a different y value, the rasterization algorithm may need to be called more than once to assure that the next edge pixel is on the next display line.

The standard midpoint line algorithm as presented by Foley [21] and as originally presented by Pitteway [22] and modified by van Aken [23 ] is used in the integer implementation of Merge/Split. More recent algorithms were investigated [24, 25] but they present no advantage for this application since they typically take advantage of symmetry considerations and use double (or other multiple) step techniques. The symmetry considerations allow the second half of the line to be drawn from the first half with little additional effort. For the algorithms presented here,
the work might be wasted if there is an intervening vertex that causes a merge or split operation. Additionally, two edges are being processed at a time and the two edges will not typically have the same end coordinate so that converting edges from the opposite end would have the edges unsynchronized in y value.

In this application, since the edges are 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 being filled have additional attributes, such as z-value, surface normal, color, and intensity values, these values and the associated increments would be stored with the edge. The values are determined from the corresponding values stored as part of the vertex data structure for the vertices the define the edge.

When a region is being filled, the y value of the next vertex in the sorted vertex list is used as a stopping point for the filling process. When this y value is reached, the corresponding vertex is examined to see if it is the end point of one (or both) of the edges or if it is a SPLIT vertex which is located within the region being filled. The region inclusion check is simple since the current x value is stored for each edge and the x value of the vertex in question is known. Since the test is trivial at this point, it would be inefficient to check stopping point vertices for region inclusion for all possible regions as a pre-processing step.

If the stopping point is neither the end point of one of the edges nor a split vertex within the region, it must lie outside the region and the filling process is resumed until the next stopping point is reached.

The Merge/Split Algorithm and related procedures are presented below. Edges of a segment are e1 and e2, v is the vertex causing a transition, and vs is stopping point for the region. The
function PredEdge builds an edge from a vertex v to its predecessor and SuccEdge builds an edge from vertex v to its successor. NextVert returns the next vertex in sorted order to serve as the next stopping point. EndVert returns the ending vertex of the edge supplied as a parameter. The fill function fills a horizontal line between at value y between two supplied x values.

Note that when implemented, recursive calls would be replaced with continuation of a region as possible (e.g. CONTINUE_PRED).

for each vertex v in list of START vertices
  if ( v.EdgeType != PRED_HORIZON)
    e1 = PredEdge( v )
    if ( v.EdgeType = SUCC_HORIZON )
      e2 = SuccEdge( SuccVert( v ) )
      vs = NextVert( SuccVert( v ) )
    else
      e2 = SuccEdge( v )
      vs = NextVert( v )
  fill_segment( v, e1, e2, vs)

Merge/Split Algorithm
Procedure split( e1, e2, v )

    if ( e2.x < e1.x ) swap( e1, e2 )
    if ( v.EdgeType = NO_HORIZON)
        e3 = SuccEdge( v )
        e4 = PredEdge( v )
        vs = NextVert( v )
    else /* PRED_HORIZON */
    fill( v.x, PredVert( v ).x, v.y )
        e3 = SuccEdge( v )
        e4 = PredEdge( PredVert( v ) )
        vs = NextVert( PredVert( v ) )
    fill_segment( v, e1, e3, vs)
    fill_segment( v, e4, e2, vs )

Split Algorithm

Procedure merge( v, e )

    if ( v.MergeEdge != NULL )
        fill_segment( v, e, v.MergeEdge, NextVert( v ) )
    else if ( v.LeftVert != NULL )
        merge( v.LeftVert, e )
    else
        v.MergeEdge = e

Merge Algorithm
Procedure fill_segment( v, e1, e2, vs )

while ( e1.y < vs.y )
    rasterize e1 until e1.y changes, rasterize e2 until e2.y changes
    fill (e1.x, e2.x, e1.y)
if ( vs = EndVert( e1) or vs = PredVert( EndVert( e1)) )
    if ( vs.EdgeType != NO_HORIZON)  fill to predecessor or successor
    case ( vs.VertexType)
        STOP: return
        CONTINUE_PRED:
            if( vs.EdgeType = NO_HORIZON)
                e1p = PredEdge( vs )
                vsp = NextVert( vs )
            else if ( vs.EdgeType = PRED_HORIZON)
                e1p = PredEdge( PredVert( vs) )
                vsp = NextVert( PredVert( vs ) )
            else
                e1p = PredEdge( vs )
                vsp = NextVert( SuccVert( vs ) )
            fill_segment( vs, e1p, e2, vsp )
        MERGE:
            if ( vs.y = EndVert(e2).y )
                EndVert( e2 ).LeftVert = vs
                if ( EndVert( e2 ).MergeEdge != NULL )
                    merge( vs, EndVert( e2 ).MergeEdge )
            else  merge( vs, e2 )
            return
        else if ( vs = EndVert( e2) or vs = SuccVert( EndVert( e2)) )
            if ( vs.EdgeType != NO_HORIZON)
                fill to predecessor or successor as appropriate
            case ( vs.VertexType)
                CONTINUE_SUCC:
                    if ( vs.EdgeType = NO_HORIZON)
                        e2p = SuccEdge( vs )
                        vsp = NextVert( vs )
                    else if ( vs.EdgeType = SUCC_HORIZON)
                        e2p = SuccEdge( SuccVert( vs ) )
                        vsp = NextVert( SuccVert( vs ) )
                    else
                        e2p = SuccEdge( vs )
                        vsp = NextVert( PredVert( vs ) )
                    fill_segment( vs, e1, e2p, vsp )
                MERGE:
                    merge( vs, e1 )
                    return
        else if ( ( vs.VertexType = SPLIT) and ( vs between e1.x and e2.x ) )
            split( e1, e2, vs )
        else  fill_segment( vs, e1, e2, NextVert( vs ) )

    Fill Segment Algorithm
6.2 REGION FILLING

Filling a region is initiated from a vertex labeled START or from a pair of edges that form a new region after a SPLIT or MERGE vertex has been encountered. Before the region is begun, the edges are possibly not initialized.

If a region is begun from a START vertex, neither edge has been created and the region is defined only by the START vertex and its successor and predecessor vertices. The situation is slightly more complicated if a horizontal edge is present.

If a region is begun because of the action taken in response to a MERGE vertex, the two edges already exist. If a SPLIT vertex causes the two regions to be begun, each will have one edge (from the region being split) and will need to create a new edge from the SPLIT vertex and its successor or predecessor.

When a CONTINUE_PRED or CONTINUE_SUCC vertex is encountered, filling of the region continues with the terminated edge replaced by a new edge generated from the ending vertex and its predecessor or successor.

A region is filled until a STOP vertex is reached. It is possible that there will be two STOP vertices for a region since there may be a horizontal edge at this point. In this case both vertices will be labeled as STOP with one being labeled PRED_HORIZON and the other as SUCC_HORIZON.
6.2.1 Beginning a Region

If a new region is begun with a START vertex, V, one edge of the region, \( E_1 \), is created from V and its predecessor, P, while the other bounding edge, \( E_2 \), is created from V and its successor, S. If the START vertex is also part of a horizontal edge, a horizontal fill operation is done between the vertices. One of the bounding edges is created from V, which is labeled PRED_HORIZON, and its successor and the other from V, which is labeled SUCC_HORIZON, and its predecessor.

![Diagram of a region beginning](image)

**Figure 6.1 Beginning a region**

As discussed earlier a new region may begin as the result of a MERGE or SPLIT vertex. More detail of this process is presented in the sections describing the action taken when these vertices are encountered.

Once the new edge(s) are created, they are initialized and the filling process is begun.
6.2.2 Encountering a Vertex

When a vertex, $V_S$, which is also $S$ in Figure 6.1, is encountered as a stopping point and is part of the region, the region must be modified. If $V_S$ is labeled CONTINUE, one of the edges is replaced with a new edge. If $V_S$ is labeled MERGE two regions are combined. If $V_S$ is labeled SPLIT two regions are created.

As mentioned in the introduction to this chapter, encountering a STOP vertex indicates the end of the region.

6.2.2.1 CONTINUE Vertex

The case of a CONTINUE vertex is simple. Figure 6.2 illustrates the situation when vertex $V_S$ is encountered while filling the region between edge $E_1$ and edge $E_2$.

When $V_S$ is encountered edge $E_1$ is unmodified and edge $E_2$ is replaced by a new edge determined by $V_S$ and $S$. In this case $V_S$ is labeled CONTINUE_SUCC. Note that $V_S$ is the end point of edge $E_2$.

One approach to continuation of the filling process would be to treat the region as a new region and to execute a new call to the fill segment routine. However, to avoid the overhead of a function call, the region is treated as continuing with a modified edge. Since the initialization flag for the new edge has not been set, the filling routine must detect this situation and initialize the edge before continuing.
Figure 6.2 Encountering a continue vertex

If the vertex $V_S$ were associated with edge $E_1$, it would be labeled CONTINUE_PRED and similar processing would be necessary.

If $V_S$ is CONTINUE_SUCC and is also labeled PRED_HORIZON, the new edge is created from $V_S$ and $S$. The predecessor of $V_S$, vertex $P$, is then ignored as a stopping point.

Figure 6.3 Encountering a CONTINUE vertex that is also PRED_HORIZON

If $V_S$ is CONTINUE_SUCC and is also labeled SUCC_HORIZON, the new edge is formed from $S$ and $SS$ (the successor of the successor of $V$). Point $S$ is ignored as a stopping point.
6.2.2.2 MERGE Vertex

The most difficult case arises when the stopping point encountered is a MERGE vertex. Since the regions are being filled independently, possibly on different processors, and a MERGE vertex indicates two (or more) regions are to be combined, the region that is finished first must “wait” until the other region is finished. Here “finished” means that the same MERGE vertex is encountered in the filling of the second of the regions to be combined. If the processing is being done on a parallel system, the wait must be done in such a way so as not to prohibit the processor from working in another region.

The non-terminating edge of the first region to finish is associated with the MERGE vertex through the MergeEdge pointer field in the vertex data structure. When a region reaches a MERGE vertex, the MergeEdge pointer is checked for the presence of a waiting edge.
In the Figure 6.5, V is the MERGE vertex. In (a) the leftmost region is finished and the edge from A to B is associated with vertex V. Since the edge has been initialized and partially scan converted, the present position, \( P_1 \), is part of the edge data structure. In (b) the region bounded by the edges from C to D and from F to V reaches MERGE vertex V, the edge A to B is waiting and the new region is formed by two initialized edges, A to B with present position \( P_1 \) and C to D with present position \( P_2 \).

If a horizontal edge is present at the MERGE vertex, the continuing edge of the first region to finish is associated with the leftmost MERGE vertex regardless of which one is encountered first. The leftmost vertex can be easily determined by the PRED_HORIZON and SUCC_HORIZON labels.

![Figure 6.5 Encountering a MERGE vertex](image)

6.2.2.3 SPLIT Vertex

The result of encountering a SPLIT vertex within the region can be viewed in two ways. One view is that two new regions are created, each with one edge from the previous region. Therefore, two calls to the fill segment routine should be made. These calls may be recursive in a...
sequential implementation or may be handled by placing the regions in a queue of regions to be filled in a parallel implementation.

The other view is that the region is modified and one new region is created. This approach improves efficiency in that one of the regions does not need to be stored and the overhead of starting a new region is avoided. The continuing region is treated in the same manner as if the split vertex had been a CONTINUE vertex in that one of the two bounding edges is replaced while the other is not effected.

Figure 6.6 illustrates a SPLIT vertex, V, that is encountered while filling a region bounded by edges $E_1$ and $E_2$. Edge $E_1$ was initialized based on vertices A and B while edge $E_2$ was initialized based on vertices C and D. Edge $E_1$ has been scan converted up to the point $P_1$ and $E_2$ to the point $P_2$ when V is encountered. Vertex V has successor vertex S and predecessor vertex P.

The first of the new regions is bounded by the already initialized edge $E_1$ and a new edge, $E_3$, defined by vertices V and S. The second is bounded by $E_2$ and a new edge, $E_4$, defined by vertices V and P. Note that the SPLIT vertex causes two new edges to be created much like a START vertex, except that the edges are associated with different regions.

When a new edge is first used for filling it is initialized. The edge data structure contains an initialization flag so that the fact that edges $E_1$ and $E_2$ are already initialized can be determined.
If a horizontal edge is present at the SPLIT vertex, there are actually two vertices labeled SPLIT with one also labeled SUCC_HORIZON and the other PRED_HORIZON. In this case one new edges is formed from VP (vertex labeled PRED_HORIZON) and S and the other new edge is formed from VS (vertex labeled SUCC_HORIZON) and P.

6.2.2.4 Multiple Vertices

The presence of multiple vertices at the same y value complicates the algorithm somewhat. However, since the vertices are encountered in a left to right sorted order, many such cases result
in a region being started or continued only to stop again before any filling is done. The case
where both edges of a region have a CONTINUE vertex at the same y value is, therefore, trivial.

If more than one MERGE vertex occurs on the same scan line, the “middle” region will have no
continuing edges but may be finished first, last, or someone in between. In this case, the
continuing edge is associated with the leftmost MERGE vertex as in the case where the MERGE
vertex is part of a horizontal edge. However, it is not as simple to determine the leftmost
MERGE vertex as it is in the case where the only two such vertices are “connected” by the
PRED_HORIZON and SUCC_HORIZON labels.

When a “middle” region terminates, the continuing edge is associated with the leftmost MERGE
vertex. Note that this is the vertex that actually stops the filling of the region since it is
encountered first. The rightmost MERGE vertex is associated with the leftmost by placing a
pointer to the leftmost vertex in the LeftVert field of the vertex data structure.

Figure 6.8 illustrates a situation where there are 3 MERGE vertices with the same y value. When
region A terminates edge \( E_1 \) is associated with vertex \( V_1 \). If there is already an edge stored in the
vertex data structure, both edges are known and the new region is determined. When region B
finishes, vertex \( V_2 \) is checked for an associated edge. If one exists it is moved to vertex \( V_1 \) and
the LeftVert pointer of \( V_2 \) is set to point to \( V_1 \). If there is no edge associated with \( V_2 \) the pointer
is set and will be used later when an edge is associated with \( V_2 \).
Assume that the regions finish in the order C A D B. When C finishes, the LeftVert pointer of $V_3$ points to $V_2$. When A finishes edge $E_1$ is associated with $V_1$ through MergeEdge. When D finishes edge $E_2$ is associated with $V_3$. Since the LeftVert field of $V_3$ is points to $V_2$, edge $E_2$ is associated with $V_2$. When B finishes the edge associated with $V_2$ is passed to $V_1$. Since $V_1$ already has an edge associated with it, the new region is defined by $E_1$ and $E_2$.

It is also possible to have a SPLIT vertex combined with MERGE vertices as illustrated in Figure 6.9. Region B encounters $V_1$ before it encounters $V_2$. Therefore, region B is not split. However, the LeftVert field of $V_3$ will point to $V_1$ since $V_1$ and $V_3$ are end points of the edges defining region B and they both have the same y value. Region C will stop for $V_3$ and $E_2$ will be associated with $V_3$. Eventually, regardless of order, the region between $E_1$ and $E_2$ will be defined. However, since $V_1$ was the last stopping point for this region, $V_2$ will cause the region to be split into the regions D and E. When region E is being filled, $V_3$ will be encountered but will be ignored since it is a MERGE vertex that is not endpoint of either of the bounding edges $E_2$ or $E_4$. 

![Figure 6.8 Multiple MERGE vertices](image-url)
In a similar manner, arbitrary arrangements of vertices are successfully filled by this algorithm including cases where some of the MERGE and SPLIT vertices are part of a horizontal edge.
CHAPTER 7 – ACTIVE EDGE LIST ALGORITHM

7.1 OVERVIEW

When the Merge/Split algorithm is modified to fill regions on a given scanline at the same time, the regions are no longer independent and the algorithm becomes very similar to the standard scanline approach.

7.2 COMPARISON TO SCANLINE

In the standard scanline algorithm, edges are stored in a bucket sorted edge table. In this data structure there is, for each scanline, a list of edges that begin on the given scanline. The edge data structure contains information needed to rasterize the line and the maximum y value which serves as a stopping point. When a scanline is encountered, any edges in the list for that line are added to the active edge table (AET). In addition, edges in the AET must be checked to determine if they terminate at the given scanline. The AET is maintained in sorted order by x value and pixels are filled between pairs of edges in this table.

In the AEL algorithm presented here, the vertices are stored in a sorted vertex list as in the case of the Merge/Split algorithm. A list of edges much like the AET is maintained for filling purposes. This list is checked for modification only at event points (vertices.) An edge is added, removed, or replaced based on the classification of the vertex serving as the stopping point.
7.3 REGION FILLING

When a START vertex is encountered, the bounding edges are found as in Merge/Split and are added to the active edge list. The edges are the predecessor and successor edges of the START vertex. Horizontal edges are handled the same as in Merge/Split. A STOP vertex indicates the end of a region and the edges are removed.

A SPLIT vertex indicates the region is to split into 2 regions. Bounding edges of the new regions are the bounding edges of the original region and the predecessor and successor edges of the SPLIT vertex. Therefore, two edges are added to the list between the two existing edges for the region. If two SPLIT vertices have the same y value, the second SPLIT will cause the active edge list to again be modified before any of the region is filled.

A MERGE vertex causes the two edges incident to the MERGE vertex to be removed. They will be adjacent in the list. If there are multiple MERGE vertices at the same y value, they are acted upon independently as explained for SPLIT vertices.

When a CONTINUE vertex is encountered, an edge is replaced with the appropriate new edge.

If two vertices are endpoints of a horizontal edge, they will appear consecutively in the sorted vertex list and only the vertex with the smaller x value (the one occurring first) will modify the active edge list.

The Active Edge List Algorithm is presented below. Note that the routine add_edges must search the list for the proper location for the edges to be added. Similarly, remove_edges must find the first of the two edges and remove the adjacent pair. The routine replace_edge must also find the
edge to be replaced. Each of these routines will initialize the edge before it is inserted into the active edge list.

Procedure AEL:
(for n vertices stored in sorted vertex list v[ ])

set ael to empty
i=1
while (i < n)
    case (v[i].VertexType)
    START:
        e1 = PredEdge(v[i])
        x1 = v[i].x
        if (v[i].EdgeType = SUCC_HORIZON) i = i + 1
        e2 = SuccEdge(v[i])
        x2 = v[i].x
        if (x1 != x2) fill(x1,x2,v[i].y)
        add_edges(ael,e1,e2)
    MERGE:
        e1 = PredEdge(v[i])
        x1 = v[i].x
        if (v[i].EdgeType = SUCC_HORIZON) i = i + 1
        e2 = SuccEdge(v[i])
        x2 = v[i].x
        if (x1 != x2) fill(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) i = i + 1
        e2 = PredEdge(v[i])
        x2 = v[i].x
        if (x1 != x2) fill(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) i = i + 1
        e2 = PredEdge(v[i])
        x2 = v[i].x
        if (x1 != x2) fill(x1,x2,v[i].y)
        remove_edges(ael,e1,e2)

Active Edge List Algorithm
(continued on next page)
CONTINUE_SUCCEED:
    if (v[i].EdgeType = NO_HORIZON)
        e1 = PredEdge(v[i])
        e2 = SuccEdge(v[i])
    else if (v[i].EdgeType = PRED_HORIZON)
        e1 = PredEdge(v[i+1])
        e2 = SuccEdge(v[i])
        fill(v[i+1].x, v[i].x, v[i].y)
        i = i + 1
    else
        e1 = PredEdge(v[i])
        e2 = SuccEdge(v[i+1])
        fill(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)
        e1 = SuccEdge(v[i])
        e2 = PredEdge(v[i])
    else if (v[i].EdgeType = SUCC_HORIZON)
        e1 = SuccEdge(v[i+1])
        e2 = PredEdge(v[i])
        fill(v[i+1].x, v[i].x, v[i].y)
        i = i + 1
    else
        e1 = SuccEdge(v[i])
        e2 = PredEdge(v[i+1])
        fill(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

Active Edge List Algorithm

Procedure fill_segments(ael, v)

    while (current y < v.y – 1)
        for each pair of edges e1 and e2 in ael
            rasterize e1 until e1.y changes
            rasterize e2 until e2.y changes
            fill(e1.x, e2.x, e1.y)

Fill_segments Algorithm
8.1 GENERALIZED POLYGONS

The Merge/Split algorithm has been extended to allow rendering of 2D and 3D polygons that have curved sides. Normal definitions of polygons require that the area be bounded by 3 or more straight line segments. The term *generalized polygon* is used to refer to a flat plane region bounded by any number of edges where the edges may be either straight or curved.

As long as there is a technique to scan convert a curve in scanline increments, the curve may be an edge of a polygon filled by Merge/Split (or AEL).

Similar surfaces can be rendered more directly by several approaches. Such surfaces are often tensor product surfaces where the boundary curves are NURBS, Hermite polynomials, or Bezier curves. Such a surface is viewed as a rectangle in uv parameter space which is deformed into the 3D surface. Surfaces may also be “trimmed” by a set of curves [26]. Such curves are often converted to Bezier representation. Algorithms have also been presented to directly scan convert various types of surfaces [27, 28, 29, 30].

The surfaces rendered by the modified Merge/Split algorithm differ from tensor product surfaces in that they are “flat” surfaces bounded by curves of various types.
8.2 TYPES OF CURVED SIDES

A simple approach to scan converting a curved edge of a generalized polygon would be to convert the curve to a polyline with whatever resolution might be necessary. The obvious disadvantage of this technique is that a polygon with a very large number of sides would be created. Each of the edges would be bound by vertices and each of these vertices would need to be classified and used as a stopping point.

Various techniques have been developed to convert curved lines to polylines. Example curves are circles and ellipses [31] and generalized conic sections [32]. These algorithms are designed to generate as few line segments as possible given an error tolerance to the curve.

There has been much work done on the problem of directly rasterizing various types of curves. Adaptive forward differencing has been used to render parametric cubic spline curves [33, 34], cubic curves with integer arithmetic [27], NURBS using integer arithmetic [35, 36], and trimmed NURBS [37]. In the work of Klassen [34], Bezier curves are rasterized after they have been converted to monotone segments.

The implementation presented in this work is designed to handle curved edges that are expressed in Bezier form. Bezier curves are a natural form of representation of the sides of a polygon since two of the four control points (for cubic curves) are also vertices of the polygon. The edge data structure is modified to allow representation of the additional control points.
In addition, there are many techniques available to convert various types of curves to Bezier form [38]. The Bezier form is often used even if conversion from another form is required because [39]

1. Bezier curves have the convex hull property (the curve is contained within the convex hull of the control points)
2. subdividing the curve at a local minimum or maximum is easy
3. it is easy to raise the degree of a Bezier curve to match that of other segments

In addition, efficient methods have been developed to find local minima and maxima and the Berenstain basis is well suited for polynomial root finders used to find relative extrema [40].

Properties (2) and (3) above have also been demonstrated for other types of curves. For example, B-splines can be subdivided [41] and have their degree raised [42].

Other types of curves can be rasterized. A technique exists [23] to scan convert ellipses with a midpoint algorithm similar to that used for straight lines in this work. General types of curves which are not parametric can also be rasterized [43].

Another class of curves that can be rasterized directly includes nu splines [44] and tau splines [45]. Nu splines are piecewise approximations to polynomials under tension. Techniques have been developed to convert nu splines to Bezier representation [46]. Tau splines can also be converted to Bezier representation [47]. Thus, even curves under tension can be used as sides of polygons.
8.3 MONOTONE SEGMENTS

In order to rasterize a generalized polygon with curved edges, the curves must be broken into monotone segments. Then the vertices can be classified in the same manner as straight edges.

Klassen [34] lists several reasons the curve segments should be broken into monotone segments. One of the primary reasons for his work is that loops and cusps are properly handled. Simple polygons will not display such behavior, so this reason is not important for polygon edges.

In addition monotone edges are easier to clip. This consideration is not important in this work but might be so for a full practical implementation.

There are two reasons cited that are important in polygon filling. The first is that the intervals are better behaved numerically. The second is that splitting allows the curves, since they are monotone, to be rasterized in scanline increments.

The primary reason to break Bezier curves into monotone segments in a vertex classification scheme is that the vertices can be classified as if they were straight edges. The only points that need to be considered are the end points of the edge.

One possible complication is that curve segments could intersect. Since Merge/Split and AEL assume simple polygons, this complication is assumed to not be present.

Figures 8.1, 8.2, and 8.3 illustrate the classification of vertices that bound curved edges. Figure 8.1 is a polygon with each edge a curve segment. The end points of segments (polygon vertices) are represented by filled circles. In Figure 8.2, unfilled circles represent vertices added when
curved edges are broken into monotone segments. In Figure 8.3, the curved segments are replaced by straight lines and the vertices are classified accordingly.

Most of the vertices introduced to create monotone segments are classified as CONTINUE vertices. However, one introduced vertex is classified STOP and one is classified SPLIT. Note also that one of the original vertices is classified as a MERGE vertex.

The generalized polygon represented in these figures could also be a closed Bezier curve which could be filled as a polygon by using this approach.

![Figure 8.1 Generalized polygon](image)

Filled circles are vertices
Figure 8.2 Generalized polygon with monotone segments
Open circles are vertices added to create monotone segments.

Figure 8.3 Vertices are classified as if monotone edges were straight
There are, however, some situations where treating a curved edge as a straight edge connecting the two end points can result in incorrect vertex classification. Vertex v in Figure 8.4 illustrates the problem. Dotted lines indicate the relationships that would be used if the edges are assumed to be straight lines. Figure 8.5 illustrates the proper approach to vertex classification for a curved edge. The curved edge is treated as if it were the straight edge that is the tangent to the curve at the vertex being classified as indicated by the dotted line.

![Figure 8.4 Incorrect vertex classification](image1)

![Figure 8.5 Vertex classification using tangent lines](image2)
In order to break a Bezier curve (or other type) into monotone segments, it is first necessary to identify local minima and maxima in both x and y coordinates. Since the curves are parametric, the algorithm will be identical for either coordinate. Each will generate a sequence of 0, 1, or 2 relative extrema including the ends of the interval. Combining the results from each coordinate yields a sequence of 0, 1, 2, 3, or 4 points where the curve should be split so that all segments are monotone.

One simple approach would be to evaluate the Bezier curve in increments, but this method would be essentially the same as conversion to polylines that would result in adding many vertices to the polygon. Depending on curvature and desired resolution, more than one hundred vertices could be added for each curved edge. Another option is to use a general root finder on the derivatives. The derivative can be easily found by various methods [48, 49]. The root finder can be based on binary subdivision [50] or a more specialized subdivision approach [26].

The approach used in this work is based on a technique developed by Whitted [51,52] and used in [34]. In this approach the curve is converted into a power series in t for each of the x and y parametric representations. The derivative is taken, which for a cubic curve yields a quadratic equation. A point of inflection is also easily found if present. Relative extrema are detected by comparing the values of the derivative at the end points and at any points of inflection. At most one square root is necessary, and it is needed only if it is known that it will yield a real solution.

The values of the parameter for the extrema are ordered in a list of 0 to 4 points where the curve should be split, yielding 1 to 5 intervals which will be monotone edges of the polygon.
After the splitting points are determined, the Bezier curve needs to be split in such a way that the resulting curves are Bezier curves parametric in the interval 0 to 1 so that they will be acceptable polygon edges. New control points must be created so this condition is met.

The end points of the generated segments will be points on the curve which correspond to the t values at the splitting points. For example, for an original edge B(t) with splitting points $t_1$, $t_2$, and $t_3$, the new curves will be:

$$B_a (t_a : 0 \leq t_a \leq 1) = B (t: 0 \leq t \leq t_1)$$

$$B_b (t_b : 0 \leq t_b \leq 1) = B (t: t_1 \leq t \leq t_2)$$

$$B_c (t_c : 0 \leq t_c \leq 1) = B (t: t_2 \leq t \leq t_3)$$

$$B_d (t_d : 0 \leq t_d \leq 1) = B (t: t_3 \leq t \leq 1)$$

Figure 8.6 Curve with 3 extrema
For Bezier curves the subdivision process has been described by Lane [52] and Farin [49, 53]. It is also possible to subdivide the curves by representing them using a power series basis and limits on parameters to represent the different segments [51, 28]. Farin’s approach has advantages for adaptive forward difference methods, however, since it increases the smallest step size with each subdivision.

The method applied in the modified Merge/Split algorithm follows that of Farin as originally proposed by deCasteljau and is numerically stable. The approach is to take a Bezier curve defined over [0,1] and split it into two curves defined over [0,c] and [c,1] in such a way as to have two curves each defined over [0,1]. A parameter, s, is introduced over the interval [0,c] and the curve C is constructed from the original curve B such that C is part of B. By requiring identical derivatives at key points on C and B, the subdivision formula

\[ C_j = B'_s(c) \]

is derived and is easily calculated in the process of evaluating B at point c using the deCasteljau algorithm.

8.4 MODIFICATIONS TO MERGE/SPLIT

If the polyline approach is used to represent a curved edge, the data structures and algorithms are unchanged. The conversion to a polyline from some form of input curve can be viewed as a preprocessing step.
If the curve representation is to be used for rasterization, additional information must be stored as part of the edge data structure. The edge data structure should be created at input rather than waiting for the edge to be created during the filling process as is done for straight edges. For example, if the Bezier form is used, the input will consist of four control points that define the edge. The control points must be stored until they are needed. An obvious approach is to input the control points during vertex input and create the edge data structure during this phase of the algorithm.

Since the edges are associated with a beginning and ending vertex in the context of filling, it is reasonable to associate the edge data structure created at input with the vertex data structure. The vertex data structure will have two additional fields, PredEdge and SuccEdge, both of which are pointers to edge data structures. These pointers will be NULL for a new straight edge and can be checked during the filling process. Note that the field EdgeType is not used for curved edges since there will be no horizontal edges except in the case of degenerate curved edges.

The edge data structure will need to have a field to indicate the edge type so that the proper rasterization algorithm is used. This field could indicate a straight edge and differentiate between various types of curves that might be implemented. In addition, proper values such as control points and intermediate values used for rasterization are stored in the edge data structure.

For the Bezier representation, an edge structure contains the coordinates of the control points, current parameter value, current step size, values needed for forward difference calculations, and a flag to indicate that the edge has been completely rasterized.

The input format for the algorithm must also be modified. For Bezier segments, the middle control points must be input as well as the bounding vertices. If two adjacent edges are Bezier
segments, the common end point is a control point for both edges. It is, therefore, necessary to
code the input data to indicate if a value represents the end point of a straight edge or a control
point of a Bezier segment. The common end point of two adjacent segments should be so
indicated to avoid repetition.

One additional complication is that the vertex that is (arbitrarily) input first might also be a
control point for the last edge if the last edge is a curve.

If B-spline curves are used as polygon edges, the control points must be input. The actual
vertices could be input in addition or they could be calculated from the control points. The
control points overlap in the sense that one control point is involved in the specification of more
than one curve segment. If there are consecutive edges, control points could be linked or
repeated. Repetition results in an easier implementation but would be more difficult to manage if
interactive curve modification is allowed.

The first consecutive B-spline curve segment is represented by four control points, with
additional consecutive segments represented by fewer control points. The case where the first
edge input and the last edge input are consecutive B-splines is more problematic than with Bezier
curves since the first and second edges will need values not appearing in the input data stream
until the last segment unless control points are repeated.

After the data is input, the curves must be broken into monotone segments before edges are
created. This process creates new vertices of the polygon to bound the monotone edges. Once all
vertex end points bounding monotone segments are stored, the process of vertex classification is
done exactly as in the case of a polygon with all straight edges.
The only other modification to the algorithm is ensuring that the curve of a given type is properly rasterized.
CHAPTER 9 - RESULTS OF IMPLEMENTATION

For a new algorithm to be presented and accepted which performs a "standard" task such as filling a polygon, the algorithm must have some advantages in terms of flexibility, space overhead, or speed. The algorithm must also construct correct solutions for at least as large a class of inputs as standard algorithms. The correctness, flexibility, and performance of the algorithms are presented in the following sections.

New algorithms for standard tasks can also be interesting if they are based on a different approach. As documented previously, the Merge/Split and AEL algorithms are point based rather than edge based.

9.1 CORRECTNESS

Polygons with different numbers of sides and different arrangements of interior holes have been successfully filled using the vertex classification algorithms. Many of these polygons were constructed to verify that the algorithms correctly fill polygons with large numbers of events (merge, split, start, stop, etc. for either boundary or hole edges) occurring on the same scan line. Test polygons with curved edges were also filled correctly.

Several example filled polygons are exhibited in the following figures.
Figure 9.1 Polygon with 5 sides

Figure 9.2 Polygon with 7 sides
Figure 9.3 Polygon with 10 sides

Figure 9.4 Polygon with 7 sides and 7 sided hole
Figure 9.5 Polygon with 12 sides and 15 sided hole

Figure 9.6 Polygon with 74 sides
Figure 9.7 Polygon with 112 sides

Figure 9.8 Polygon with 25 sides and 13 sided hole - many vertices with same y value
The correctly filled polygons presented above demonstrate the robustness and flexibility of the algorithm in the presence of complicated combinations of vertices bounding exterior and hole edges. The algorithm also successfully fills polygons with multiple holes.
9.2 PERFORMANCE

The performance of the algorithms can be measured in terms of storage space and processing
time. Either algorithm, Merge/Split or AEL, require minimal storage for each vertex and only
enough storage for edges that are currently involved in the filling process.

The computational complexity of the algorithms for a polygon with N sides is \( O(N \lg N) \) since
the vertices must be sorted in a type of pre-processing step to serve as stopping points. After the
vertices are ordered, the algorithms are reduced to 1 dimensional by the plane sweep approach.
There is a constant amount of work performed to classify each vertex and each vertex is visited
only 1 time to create an edge during the filling process. Therefore, this portion of the algorithm is
linear.

However, when a segment is being filled, all the vertices of the polygon that have y values within
the range of the segment are check as possible stopping points resulting in a maximum
complexity for this portion of the algorithm of \( O(SN) \) where \( S \) is the number of segments.

Another approach to complexity for the algorithms is to consider the work done as a function of
the area filled. If there are \( P \) pixels contained inside a polygon, the complexity would be \( O( P) \)
since there is a constant amount of work for each pixel. The same argument would be valid for
any polygon filling algorithm.

The combined complexity could be stated, therefore, as \( O( ( N \lg N ) + N + SN + P ) \) where the \( N \)
lg N term is from preprocessing, the N term from the creation of new edges (and, perhaps,
regions) when a vertex is encountered, the SN from checking potential stopping points and the P
term from the area of the polygon being filled.
The most common algorithm for polygon filling, scanline, has a preprocessing step where edges are sorted. This sort, however, is a bucket sort where the number of possible values is known and is equal to the vertical resolution of the output. Therefore, this partial sort is $O(N)$. The remainder of the algorithm is driven by the number of scanlines and, is therefore, of complexity $O(V)$ where $V$ is the vertical resolution of the output. Thus the scanline algorithm could be viewed as $O(N + V + P)$.

In order to measure actual performance, the computation time required to fill randomly generated polygons was measured for Merge/Split, AEL, and scanline. Two versions of Merge/Split and AEL were tested: one that uses integer arithmetic and the Midpoint algorithm to rasterize the edges and one that uses floating point arithmetic and the inverse slope for rasterization. The implementation of scanline used floating point arithmetic. The tests were performed on a variety of hardware and software platforms. In each case, the polygons were constructed to provide a variety of sizes, shapes, and orientations. Previously, the AEL algorithm had been compared [54] with scanline for groups of triangles, quadrilaterals, and random polygons.

9.2.1 Performance for Single Polygons

For simple performance comparisons, the pixels of the polygons used to demonstrate correctness were calculated but not displayed. Using this approach, a variety of systems could be tested without display speed effecting the results. Integer and floating point versions of Merge/Split and AEL were compared against scanline. The hardware/software systems used were MSDOS and TurboC on a 386 processor, a 386-based Sequent system (single processor) under Dynix, an Ultrix VaxStation, a DEC MIPS Ultrix system, a Sun 3/60 with 68020 processor, and a Sun Sparcstation1+. Data is presented for the 7 polygons appearing in Figure 9.1 through Figure 9.7.
The data for various polygons on a given processor were normalized so that the time for scanline for the polygon on the given processor is 1. Although there was some variation between platforms for these few polygons, the overall trends are similar. Table 9.1 presents this data along with the mean of all the vertex classification algorithms.

<table>
<thead>
<tr>
<th></th>
<th>M/S (I)</th>
<th>M/S (F)</th>
<th>AEL (I)</th>
<th>AEL (F)</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poly1</td>
<td>0.392</td>
<td>0.324</td>
<td>0.426</td>
<td>0.380</td>
<td>0.381</td>
</tr>
<tr>
<td>Poly2</td>
<td>0.369</td>
<td>0.326</td>
<td>0.398</td>
<td>0.380</td>
<td>0.368</td>
</tr>
<tr>
<td>Poly3</td>
<td>0.322</td>
<td>0.334</td>
<td>0.373</td>
<td>0.380</td>
<td>0.352</td>
</tr>
<tr>
<td>Poly4</td>
<td>0.661</td>
<td>0.657</td>
<td>0.688</td>
<td>0.688</td>
<td>0.674</td>
</tr>
<tr>
<td>Poly5</td>
<td>0.680</td>
<td>0.720</td>
<td>0.717</td>
<td>0.745</td>
<td>0.716</td>
</tr>
<tr>
<td>Poly6</td>
<td>0.580</td>
<td>0.832</td>
<td>0.574</td>
<td>0.731</td>
<td>0.679</td>
</tr>
<tr>
<td>Poly7</td>
<td>0.790</td>
<td>0.961</td>
<td>0.901</td>
<td>1.18</td>
<td>0.958</td>
</tr>
</tbody>
</table>

Table 9.1 Time ratio to scanline mean of all platforms

![Figure 9.10 Time ratio to scanline All platforms (mean)]
9.2.2 Performance for random polygons of various sizes

Groups of 50 random convex polygons with various numbers of sides were generated and tested. These polygons, since they are convex, would have only one segment with most of the vertices being CONTINUE vertices. The time ratio to scanline was averaged for each group. Similar groups of polygons with a specified number of concave sections were also generated and tested. Depending on the random orientation, a concave section could contain multiple START vertices or a number of MERGE or SPLIT vertices as indicated in Figure 9.4.

![Figure 9.11 Effects of orientation on vertex classification](image-url)
The following tables and graphs present the time ratio to the scanline algorithm for the various vertex classification algorithms applied to convex polygons with varying numbers of sides. Each test involved 50 randomly generated convex polygons. The tests were performed on a Pentium processor running Linux.

<table>
<thead>
<tr>
<th>Number of sides</th>
<th>Minimum ratio</th>
<th>Maximum ratio</th>
<th>Mean ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.116</td>
<td>0.424</td>
<td>0.167</td>
</tr>
<tr>
<td>4</td>
<td>0.128</td>
<td>0.373</td>
<td>0.183</td>
</tr>
<tr>
<td>5</td>
<td>0.137</td>
<td>0.364</td>
<td>0.179</td>
</tr>
<tr>
<td>10</td>
<td>0.178</td>
<td>0.328</td>
<td>0.206</td>
</tr>
<tr>
<td>15</td>
<td>0.194</td>
<td>0.294</td>
<td>0.218</td>
</tr>
<tr>
<td>20</td>
<td>0.221</td>
<td>0.356</td>
<td>0.242</td>
</tr>
<tr>
<td>25</td>
<td>0.235</td>
<td>0.296</td>
<td>0.257</td>
</tr>
<tr>
<td>30</td>
<td>0.268</td>
<td>0.333</td>
<td>0.288</td>
</tr>
</tbody>
</table>

**Table 9.2 Time ratio integer Merge/Split to scanline – convex polygons**

<table>
<thead>
<tr>
<th>Number of sides</th>
<th>Minimum ratio</th>
<th>Maximum ratio</th>
<th>Mean ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.243</td>
<td>0.251</td>
<td>0.247</td>
</tr>
<tr>
<td>4</td>
<td>0.246</td>
<td>0.256</td>
<td>0.251</td>
</tr>
<tr>
<td>5</td>
<td>0.250</td>
<td>0.267</td>
<td>0.254</td>
</tr>
<tr>
<td>10</td>
<td>0.270</td>
<td>0.316</td>
<td>0.276</td>
</tr>
<tr>
<td>15</td>
<td>0.293</td>
<td>0.323</td>
<td>0.301</td>
</tr>
<tr>
<td>20</td>
<td>0.324</td>
<td>0.389</td>
<td>0.335</td>
</tr>
<tr>
<td>25</td>
<td>0.357</td>
<td>0.398</td>
<td>0.370</td>
</tr>
<tr>
<td>30</td>
<td>0.400</td>
<td>0.473</td>
<td>0.421</td>
</tr>
</tbody>
</table>

**Table 9.3 Time ratio floating point Merge/Split to scanline – convex polygons**
<table>
<thead>
<tr>
<th>Number of sides</th>
<th>Minimum ratio</th>
<th>Maximum ratio</th>
<th>Mean ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.139</td>
<td>0.480</td>
<td>0.224</td>
</tr>
<tr>
<td>4</td>
<td>0.155</td>
<td>0.410</td>
<td>0.211</td>
</tr>
<tr>
<td>5</td>
<td>0.163</td>
<td>0.415</td>
<td>0.208</td>
</tr>
<tr>
<td>10</td>
<td>0.203</td>
<td>0.368</td>
<td>0.239</td>
</tr>
<tr>
<td>15</td>
<td>0.223</td>
<td>0.344</td>
<td>0.257</td>
</tr>
<tr>
<td>20</td>
<td>0.255</td>
<td>0.426</td>
<td>0.281</td>
</tr>
<tr>
<td>25</td>
<td>0.274</td>
<td>0.329</td>
<td>0.294</td>
</tr>
<tr>
<td>30</td>
<td>0.293</td>
<td>0.379</td>
<td>0.321</td>
</tr>
</tbody>
</table>

Table 9.4 Time ratio integer AEL to scanline – convex polygons

<table>
<thead>
<tr>
<th>Number of sides</th>
<th>Minimum ratio</th>
<th>Maximum ratio</th>
<th>Mean ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.254</td>
<td>0.277</td>
<td>0.267</td>
</tr>
<tr>
<td>4</td>
<td>0.266</td>
<td>0.280</td>
<td>0.273</td>
</tr>
<tr>
<td>5</td>
<td>0.270</td>
<td>0.300</td>
<td>0.277</td>
</tr>
<tr>
<td>10</td>
<td>0.296</td>
<td>0.335</td>
<td>0.303</td>
</tr>
<tr>
<td>15</td>
<td>0.313</td>
<td>0.363</td>
<td>0.347</td>
</tr>
<tr>
<td>20</td>
<td>0.350</td>
<td>0.433</td>
<td>0.365</td>
</tr>
<tr>
<td>25</td>
<td>0.353</td>
<td>0.417</td>
<td>0.392</td>
</tr>
<tr>
<td>30</td>
<td>0.419</td>
<td>0.471</td>
<td>0.437</td>
</tr>
</tbody>
</table>

Table 9.5 Time ratio floating point AEL to scanline – convex polygons
Figure 9.12 Mean time ratio to scanline – convex polygons

The data seems to indicate increasing ratio for larger convex polygons. Assuming a linear relationship, the slope and intercept of the linear regression for each algorithm was calculated and used to extrapolate the size of a polygon for which the time ratio would be 1.0.

Correlation coefficients were calculated for the possible relationship between number of sides and ratio of time to scanline and tested for significance. The correlations were testing by using the null hypothesis that there is no correlation (\( \rho = 0 \)) for \( \alpha = 0.05 \) by calculating

\[
z = Z \cdot \sqrt{n - 3}
\]

where \( Z \) is defined as

\[
Z = \frac{1}{2} \cdot \ln \frac{1 + r}{1 - r}
\]

and requiring \( z < -1.96 \) or \( z > 1.96 \) to reject the null hypothesis. The table below presents these results.
The correlation coefficient used above was calculated for the mean ratio. The correlation coefficient also suggests a positive relationship for the minimum and maximum ratio for all algorithms except the integer versions for which there is a negative correlation calculated which is not statistically significant.

Similar tests were conducted for polygons that are not convex. A convex polygon will have one segment to be filled. For each concave area of a polygon, depending on the orientation, an additional segment to be filled may be created as illustrated in Figure 9.11. Therefore, the “complexity” of the polygon in terms of segments was measured by the number of concave sections on the polygon perimeter.

The following table presents the data for a polygon with 10 sides and a varying number of vertices creating a concave angle on the perimeter. The case of 1 vertex creating a concave area results in at most 2 segments to be filled. Two such vertices can create up to 3 segments, etc. Note that there is a maximum number of vertices that can create a concave area that is equal to (number of sides) div 3. Therefore, there are at most 3 such vertices for a 10 sided polygon.

<table>
<thead>
<tr>
<th></th>
<th>Correlation</th>
<th>$z$</th>
<th>Reject $\rho = 0$</th>
<th>Extrapolated number of sides for ratio = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>M/S – integer</td>
<td>0.9933</td>
<td>6.37</td>
<td>Yes</td>
<td>205</td>
</tr>
<tr>
<td>M/S – fl. Point</td>
<td>0.9888</td>
<td>5.79</td>
<td>Yes</td>
<td>127</td>
</tr>
<tr>
<td>AEL – integer</td>
<td>0.9857</td>
<td>5.52</td>
<td>Yes</td>
<td>202</td>
</tr>
<tr>
<td>AEL – fl. Point</td>
<td>0.9800</td>
<td>5.14</td>
<td>Yes</td>
<td>123</td>
</tr>
</tbody>
</table>

Table 9.6 Correlation between number of sides and filling time ratio for convex polygons
Table 9.7 Time ratio (mean) to scanline –
as function of number of segments, 10 sides

Since these data sets must be small because of the limit on the number of vertices creating a
concave area, the test was repeated for random polygons with 20 sides.

Table 9.8 Time ratio (mean) to scanline –
as function of number of segments, 20 sides

Figure 9.13 Time ratio (mean) to scanline –
as function of number of segments, 20 sides
As seen in the above graph and table, there appears to be a negative correlation indicating that the algorithms improve relative to scanline as segments are added to a 20-sided polygon. However, none of the calculated correlations are statistically significant.

An additional test performed compares the performance of the vertex classification algorithms to scanline for polygons of various sizes that have one vertex causing a concave region that may create a second segment.

<table>
<thead>
<tr>
<th>Number of sides</th>
<th>Minimum ratio</th>
<th>Maximum ratio</th>
<th>Mean ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.156</td>
<td>0.199</td>
<td>0.176</td>
</tr>
<tr>
<td>5</td>
<td>0.113</td>
<td>0.223</td>
<td>0.182</td>
</tr>
<tr>
<td>10</td>
<td>0.179</td>
<td>0.217</td>
<td>0.198</td>
</tr>
<tr>
<td>15</td>
<td>0.200</td>
<td>0.266</td>
<td>0.231</td>
</tr>
<tr>
<td>20</td>
<td>0.223</td>
<td>0.260</td>
<td>0.245</td>
</tr>
<tr>
<td>25</td>
<td>0.259</td>
<td>0.341</td>
<td>0.293</td>
</tr>
<tr>
<td>30</td>
<td>0.290</td>
<td>0.361</td>
<td>0.328</td>
</tr>
<tr>
<td>Number of sides</td>
<td>Minimum ratio</td>
<td>Maximum ratio</td>
<td>Mean ratio</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------</td>
<td>---------------</td>
<td>------------</td>
</tr>
<tr>
<td>4</td>
<td>0.208</td>
<td>0.266</td>
<td>0.244</td>
</tr>
<tr>
<td>5</td>
<td>0.191</td>
<td>0.336</td>
<td>0.255</td>
</tr>
<tr>
<td>10</td>
<td>0.272</td>
<td>0.282</td>
<td>0.277</td>
</tr>
<tr>
<td>15</td>
<td>0.274</td>
<td>0.400</td>
<td>0.331</td>
</tr>
<tr>
<td>20</td>
<td>0.330</td>
<td>0.340</td>
<td>0.335</td>
</tr>
<tr>
<td>25</td>
<td>0.345</td>
<td>0.464</td>
<td>0.411</td>
</tr>
<tr>
<td>30</td>
<td>0.409</td>
<td>0.510</td>
<td>0.461</td>
</tr>
</tbody>
</table>

Table 9.11 Time ratio floating point Merge/Split to scanline – polygons with one concave region

<table>
<thead>
<tr>
<th>Number of sides</th>
<th>Minimum ratio</th>
<th>Maximum ratio</th>
<th>Mean ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.175</td>
<td>0.223</td>
<td>0.201</td>
</tr>
<tr>
<td>5</td>
<td>0.137</td>
<td>0.255</td>
<td>0.209</td>
</tr>
<tr>
<td>10</td>
<td>0.213</td>
<td>0.250</td>
<td>0.232</td>
</tr>
<tr>
<td>15</td>
<td>0.194</td>
<td>0.292</td>
<td>0.236</td>
</tr>
<tr>
<td>20</td>
<td>0.265</td>
<td>0.307</td>
<td>0.286</td>
</tr>
<tr>
<td>25</td>
<td>0.232</td>
<td>0.343</td>
<td>0.294</td>
</tr>
<tr>
<td>30</td>
<td>0.292</td>
<td>0.384</td>
<td>0.337</td>
</tr>
</tbody>
</table>

Table 9.12 Time ratio integer AEL to scanline – polygons with one concave region

<table>
<thead>
<tr>
<th>Number of sides</th>
<th>Minimum ratio</th>
<th>Maximum ratio</th>
<th>Mean ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.226</td>
<td>0.294</td>
<td>0.266</td>
</tr>
<tr>
<td>5</td>
<td>0.217</td>
<td>0.367</td>
<td>0.275</td>
</tr>
<tr>
<td>10</td>
<td>0.294</td>
<td>0.306</td>
<td>0.300</td>
</tr>
<tr>
<td>15</td>
<td>0.251</td>
<td>0.409</td>
<td>0.318</td>
</tr>
<tr>
<td>20</td>
<td>0.350</td>
<td>0.373</td>
<td>0.361</td>
</tr>
<tr>
<td>25</td>
<td>0.304</td>
<td>0.455</td>
<td>0.384</td>
</tr>
<tr>
<td>30</td>
<td>0.378</td>
<td>0.486</td>
<td>0.434</td>
</tr>
</tbody>
</table>

Table 9.13 Time ratio floating point AEL to scanline – polygons with one concave region
For this data there also seems to be a relationship that the efficiency of each of the vertex classification algorithms decreases relative to scanline for increasing number of vertices in the polygon. The calculated correlation coefficients are 0.988, 0.982, 0.981, and 0.991 respectively which result in z values of 5.11, 4.70, 4.65, and 5.40 which indicates a statistically significant relationship.

The analysis was repeated for groups of random polygons with a varying number of sides each of which had the maximum number of concave regions. The correlation coefficients between the number of sides in the polygon and the time ratio for the columns of data are 0.857, 0.382, 0.994, and 0.995 which indicate a significant relationship between number of sides and the time ratio to scanline for all algorithms except floating point Merge/Split. As would be expected because of the relationship between number of sides and maximum number of segments, there is also a
correlation for the same algorithms between maximum number of sides and the time ratio. The correlation coefficients are 0.876, 0.365, 0.990, and 0.995.

<table>
<thead>
<tr>
<th>Number of sides</th>
<th>Maximum segments</th>
<th>Minimum ratio</th>
<th>Maximum ratio</th>
<th>Mean ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>0.159</td>
<td>0.198</td>
<td>0.182</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0.139</td>
<td>0.214</td>
<td>0.184</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>0.175</td>
<td>0.225</td>
<td>0.198</td>
</tr>
<tr>
<td>15</td>
<td>6</td>
<td>0.200</td>
<td>0.266</td>
<td>0.231</td>
</tr>
<tr>
<td>20</td>
<td>7</td>
<td>0.211</td>
<td>0.239</td>
<td>0.224</td>
</tr>
<tr>
<td>25</td>
<td>9</td>
<td>0.230</td>
<td>0.248</td>
<td>0.238</td>
</tr>
<tr>
<td>30</td>
<td>11</td>
<td>0.360</td>
<td>0.374</td>
<td>0.364</td>
</tr>
</tbody>
</table>

Table 9.14 Time ratio integer Merge/Split to scanline – polygons with maximum concave regions

<table>
<thead>
<tr>
<th>Number of sides</th>
<th>Maximum segments</th>
<th>Minimum ratio</th>
<th>Maximum ratio</th>
<th>Mean ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>0.248</td>
<td>0.252</td>
<td>0.250</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0.251</td>
<td>0.261</td>
<td>0.255</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>0.275</td>
<td>0.287</td>
<td>0.282</td>
</tr>
<tr>
<td>15</td>
<td>6</td>
<td>0.274</td>
<td>0.400</td>
<td>0.331</td>
</tr>
<tr>
<td>20</td>
<td>7</td>
<td>0.325</td>
<td>0.344</td>
<td>0.333</td>
</tr>
<tr>
<td>25</td>
<td>9</td>
<td>0.340</td>
<td>0.364</td>
<td>0.351</td>
</tr>
<tr>
<td>30</td>
<td>11</td>
<td>0.236</td>
<td>0.259</td>
<td>0.248</td>
</tr>
</tbody>
</table>

Table 9.15 Time ratio floating point Merge/Split to scanline – polygons with maximum concave regions

<table>
<thead>
<tr>
<th>Number of sides</th>
<th>Maximum segments</th>
<th>Minimum ratio</th>
<th>Maximum ratio</th>
<th>Mean ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>0.190</td>
<td>0.222</td>
<td>0.209</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0.160</td>
<td>0.246</td>
<td>0.211</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>0.207</td>
<td>0.265</td>
<td>0.232</td>
</tr>
<tr>
<td>15</td>
<td>6</td>
<td>0.194</td>
<td>0.292</td>
<td>0.236</td>
</tr>
<tr>
<td>20</td>
<td>7</td>
<td>0.244</td>
<td>0.266</td>
<td>0.257</td>
</tr>
<tr>
<td>25</td>
<td>9</td>
<td>0.245</td>
<td>0.292</td>
<td>0.272</td>
</tr>
<tr>
<td>30</td>
<td>11</td>
<td>0.275</td>
<td>0.302</td>
<td>0.290</td>
</tr>
</tbody>
</table>

Table 9.16 Time ratio integer AEL to scanline – polygons with maximum concave regions
### Table 9.17 Time ratio floating point AEL to scanline – polygons with maximum concave regions

<table>
<thead>
<tr>
<th>Number of sides</th>
<th>Maximum segments</th>
<th>Minimum ratio</th>
<th>Maximum ratio</th>
<th>Mean ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>0.267</td>
<td>0.277</td>
<td>0.273</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0.271</td>
<td>0.281</td>
<td>0.277</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>0.289</td>
<td>0.310</td>
<td>0.298</td>
</tr>
<tr>
<td>15</td>
<td>6</td>
<td>0.251</td>
<td>0.409</td>
<td>0.318</td>
</tr>
<tr>
<td>20</td>
<td>7</td>
<td>0.310</td>
<td>0.340</td>
<td>0.329</td>
</tr>
<tr>
<td>25</td>
<td>9</td>
<td>0.331</td>
<td>0.352</td>
<td>0.342</td>
</tr>
<tr>
<td>30</td>
<td>11</td>
<td>0.354</td>
<td>0.365</td>
<td>0.359</td>
</tr>
</tbody>
</table>

**Figure 9.15 Mean time ratio to scanline – polygons with maximum concave regions**
9.2.3 Summary of random polygon testing

For all random polygons tested the maximum ratio to scanline for any vertex classification algorithm is 0.510. The mean of all the ratios presented is 0.281. The set of polygons tested for correctness includes polygons with holes and polygons with many segments that were designed to verify correctness for “difficult” polygons. For these correctness polygons, only the floating point version of AEL was less efficient than scanline for one polygon. The average time ratio for the vertex classification algorithms for the same polygon if 0.958. The mean for the correctness polygons with holes or many transitions is 0.734.

Examination of the many tables and graphs seems to indicate somewhat constant relationships between integer and floating point versions of either algorithm and between the two algorithms for either integer or floating point. The following tables present these ratios for the case of convex polygons and polygons with a maximum number of concave regions.

<table>
<thead>
<tr>
<th>number of sides</th>
<th>M/S integer : float</th>
<th>AEL integer : float</th>
<th>Integer M/S : AEL</th>
<th>Float M/S : AEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.676</td>
<td>0.839</td>
<td>0.746</td>
<td>0.925</td>
</tr>
<tr>
<td>4</td>
<td>0.729</td>
<td>0.773</td>
<td>0.867</td>
<td>0.919</td>
</tr>
<tr>
<td>5</td>
<td>0.704</td>
<td>0.751</td>
<td>0.861</td>
<td>0.917</td>
</tr>
<tr>
<td>10</td>
<td>0.746</td>
<td>0.789</td>
<td>0.862</td>
<td>0.911</td>
</tr>
<tr>
<td>15</td>
<td>0.724</td>
<td>0.741</td>
<td>0.848</td>
<td>0.867</td>
</tr>
<tr>
<td>20</td>
<td>0.722</td>
<td>0.770</td>
<td>0.861</td>
<td>0.918</td>
</tr>
<tr>
<td>25</td>
<td>0.694</td>
<td>0.750</td>
<td>0.874</td>
<td>0.944</td>
</tr>
<tr>
<td>30</td>
<td>0.684</td>
<td>0.735</td>
<td>0.897</td>
<td>0.963</td>
</tr>
<tr>
<td>MEAN</td>
<td>0.710</td>
<td>0.768</td>
<td>0.852</td>
<td>0.921</td>
</tr>
</tbody>
</table>

Table 9.18 Time ratios for algorithms – convex polygons
The ratios indicate that the integer algorithms are almost always faster than the floating point versions and that Merge/Split is somewhat faster than AEL for most of the types of polygons tested.

### Table 9.19 Time ratios for algorithms – polygons with maximum concave regions

<table>
<thead>
<tr>
<th>number of sides</th>
<th>M/S integer : float</th>
<th>AEL integer : float</th>
<th>Integer M/S : AEL</th>
<th>Float M/S : AEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.728</td>
<td>0.766</td>
<td>0.871</td>
<td>0.916</td>
</tr>
<tr>
<td>5</td>
<td>0.722</td>
<td>0.762</td>
<td>0.872</td>
<td>0.921</td>
</tr>
<tr>
<td>10</td>
<td>0.702</td>
<td>0.779</td>
<td>0.853</td>
<td>0.946</td>
</tr>
<tr>
<td>15</td>
<td>0.698</td>
<td>0.742</td>
<td>0.979</td>
<td>1.041</td>
</tr>
<tr>
<td>20</td>
<td>0.673</td>
<td>0.781</td>
<td>0.872</td>
<td>1.012</td>
</tr>
<tr>
<td>25</td>
<td>0.678</td>
<td>0.795</td>
<td>0.875</td>
<td>1.026</td>
</tr>
<tr>
<td>30</td>
<td>1.468</td>
<td>0.808</td>
<td>1.255</td>
<td>0.691</td>
</tr>
<tr>
<td>MEAN</td>
<td>0.810</td>
<td>0.776</td>
<td>0.940</td>
<td>0.936</td>
</tr>
</tbody>
</table>

9.2.4 Performance of AEL for Z-Buffered Polygons

DeLetter [54] implemented a version of the AEL algorithm which maintained a separate active edge list for each polygon. The algorithm used a Z-buffer approach to scan convert segments of a polygon. The edges are classified in the same manner as vertices in this work based on the necessary action when the edge becomes active. The classifications used were BOTTOM, CONTINUE, MERGE, SPLIT, and STOP.

The results of DeLetter's work indicate that "merge and split", which is his name for the algorithm, is similar to Z-buffer in performance for small numbers of polygons but becomes slightly more efficient for large numbers of polygons. For yet larger numbers of polygons, merge and split becomes more efficient than scanline.
For drawing groups of random complex polygons, merge and split was comparable to Z-buffer through 450 polygons while merge and split became faster than scanline for more than 200 polygons.

For groups of rectangles, merge and split is more efficient than Z-buffer and becomes more efficient than scanline with approximately 1700 polygons.
CHAPTER 10 - PARALLEL MERGE/SPLIT

Since the Merge/Split algorithm fills individual segments of a polygon, multiple segments of a single polygon, or a group of polygons, could be filled simultaneously with appropriate parallel algorithms and hardware.

10.1 DIFFERENCES FROM SEQUENTIAL APPROACH

If the fill_segment procedure is modified to allow it to execute independently on several processors, the parallel implementation of Merge/Split only requires the main procedure to execute fill_segment on as many processors as are needed or available. As the algorithm begins, the number of processes would be equal to the number of START vertices in the polygon. A linked list (or queue depending on implementation) of START vertices is generated during the vertex classification phase.

As in the sequential version, if a CONTINUE vertex is encountered, the new edge is generated and the process continues on the same processor. When a STOP vertex is encountered, the processor is available to begin a new segment. If a SPLIT vertex is encountered, the processor may continue to fill one of the new segments while the other segment is begun on another processor.

A MERGE vertex presents the most difficult case as was true for the sequential version of Merge/Split. When a segment ends because a MERGE vertex is encountered, the continuing edge must be saved and communicated to another filling process, possibly on a different processor.
A practical system would allow many polygons to be filled at the same time. Assuming edges and vertices were properly maintained, the Merge/Split parallel algorithm could easily fill segments from different polygons simultaneously.

The parallel Merge/Split algorithm can be implemented with a variety of approaches. Obviously, the algorithm must be modified to provide for proper communication and/or synchronization among multiple processors. Both shared and distributed memory MIMD implementations are described below. A SIMD implementation would also be feasible.

10.2 IMPLEMENTATION

The parallel Merge/Split algorithm has been implemented on a transputer-based distributed memory system and on the Sequent shared memory system. The implementations are described below.

10.2.1 Transputer Implementation

Transputers are processors designed to be connected in various distributed memory parallel configurations. Communication between processors is handled by serial links with each processor having 4 data links which can be connected to links of other transputers in a reconfigurable manner. Transputers require a host link to a system that provides initialization, I/O, and program download support. Only the transputer connected directly to the host, known as the root transputer, can display graphics and perform I/O operations. Host systems may be PCs or Sun workstations.
The Merge/Split algorithm was implemented on a system hosted by a Microsoft Windows based 80486 which provides simple graphics display capabilities. A maximum of 6 T805 transputers are available on this system.

The T805 transputer is a 32 bit CPU with a 64 bit FPU and hardware process scheduling. The processor runs at 20 MHz and supports 4 bidirectional communication channels each of which can operate at up to 20 Mbits/second.

The sequential algorithm was modified so that 2 processes run on the root node. One process is a server routine that handles communication and queue management. The second process reads the vertices, classifies the vertices, and sends initial segments to the server routine to be placed on the queue. The sorted vertex list is passed to each node. The other nodes on the system execute a modified *fill_segment* process and begin by sending the server process a message indicating their availability to fill a segment.

When a filling process reaches a *STOP* vertex, a message is sent to the server process on the root node and a new segment to fill is received. New segments are sent to the queue only when a *SPLIT* vertex is encountered. The filling process continues with one of the segments and sends the other to the queue. When a *MERGE* vertex is encountered, the continuing edge is sent to the server process which associates it with the proper vertex and waits until a matching message is received from a filling process so that a new segment may be generated and placed on the queue.

The major communication problem is that all pixels to be displayed must be passed to the root processor.
Several communication configurations were tested. A sequential version of Merge/Split was also implemented on a single transputer. The sequential version executes faster than the parallel versions due to the large amount of message traffic generated. A typical situation involves a process sending a READY message to the scheduler process to request a segment to fill. When the "filling" involves only calculating pixels as was done for timing purposes, the process would fill the segment and generate a READY message before the server process could provide a segment to a different processor.

The table below presents the processing time for the sequential transputer version of Merge/Split. Time was measured both with the output displayed and not displayed. The time listed when the output was not displayed also does not include the time to transfer the pixel data to the host. In addition, time was measured including and not including vertex classification.

<table>
<thead>
<tr>
<th>Polygon</th>
<th>Segments</th>
<th>Output displayed</th>
<th></th>
<th></th>
<th>Output not displayed</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Poly5</td>
<td>4</td>
<td>6221</td>
<td>6331</td>
<td>288</td>
<td>398</td>
<td></td>
</tr>
<tr>
<td>Poly6</td>
<td>23</td>
<td>8252</td>
<td>8955</td>
<td>379</td>
<td>1069</td>
<td></td>
</tr>
<tr>
<td>Poly7</td>
<td>39</td>
<td>7696</td>
<td>9409</td>
<td>507</td>
<td>2196</td>
<td></td>
</tr>
</tbody>
</table>

Table 10.1 Time (microseconds)
Transputer sequential polygon filling
(VC indicates whether time for vertex classification is included in the time measurement.)

The initial transputer configuration consisted of a root node with the initialization and server process on the root node and a filling process on a second node. It is reasonable to expect this version to be slower than the sequential version even if communication speed were not a major factor since only one node is filling a segment at a time.
Analysis of the above data produces some interesting results. On either the sequential or the parallel implementation a large amount of the processing time is involved in the transfer and display of the pixels. Since the parallel version classifies vertices in a sequential manner the vertex classification time is essentially the same: 110 microseconds for Poly5, 697 microseconds for Poly6, and 1711 microseconds for Poly7.

The amount of time necessary to transfer and display the pixels is presented in Table 10.3 and is derived from Table 10.1 for sequential and Table 10.2 for parallel.
Table 10.3 Time in microseconds
vertex classification, sequential and parallel display

Even though there is additional pixel communication between the server and filling nodes in the 2
node version, the additional time needed to transfer and display the pixel data is less than for the
sequential version. Therefore, there must be some gain in having the filling process and the
server and communication process on separate processors.

The entries in the third column of Table 10.1 and Table 10.2 represent the segment filling time
since the measurement did not include either vertex classification or transfer and display of the
pixels. However, the parallel version also includes the time to download the vertices from the
root node to the filling node and the time to transfer the pixel data from the filling node to the root
node. Therefore, additional timing measurements were made as seen in Table 10.4. Note that
this data does not include vertex classification time.

<table>
<thead>
<tr>
<th></th>
<th>Not including pixel communication</th>
<th>Not including pixel communication or vertex download</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poly5</td>
<td>490</td>
<td>378</td>
</tr>
<tr>
<td>Poly6</td>
<td>616</td>
<td>495</td>
</tr>
<tr>
<td>Poly7</td>
<td>783</td>
<td>651</td>
</tr>
</tbody>
</table>

Table 10.4 Time in microseconds
Parallel not including vertex classification or display, and
only some communications
The difference between Table 10.2 column 3 and Table 10.4 column 3 is, therefore, the time required to download vertices and transfer pixel data between transputers. The difference between Table 10.4 column 2 and Table 10.4 column 3 is the vertex download time. The difference between these sets of numbers is the pixel communication time between transputers. It is reasonable that the pixel communication time is much larger than the vertex download time since more data is transferred and the data is transferred in smaller packets. There is additional overhead associated with messages regarding queue management. These times can be derived by subtracting the vertex download time, the pixel communication time, and the sequential segment filling time (Table 10.1 column 3) from the parallel filling time (Table 10.2 column 3). These overhead times are presented in Table 10.5.

<table>
<thead>
<tr>
<th></th>
<th>Vertex download</th>
<th>Pixel communication</th>
<th>Queue communication</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poly5</td>
<td>112</td>
<td>1010</td>
<td>90</td>
<td>1212</td>
</tr>
<tr>
<td>Poly6</td>
<td>121</td>
<td>1105</td>
<td>116</td>
<td>1342</td>
</tr>
<tr>
<td>Poly7</td>
<td>132</td>
<td>1235</td>
<td>144</td>
<td>1511</td>
</tr>
</tbody>
</table>

**Table 10.5 Parallel overhead time**

Table 10.6 summarizes these results by presenting the sequential filling time, the total parallel overhead and the ratio between the two values. Note that these ratios are essentially 3 or larger.

<table>
<thead>
<tr>
<th></th>
<th>Segment filling</th>
<th>Total parallel overhead</th>
<th>Ratio of overhead to processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poly5</td>
<td>288</td>
<td>1212</td>
<td>4.21</td>
</tr>
<tr>
<td>Poly6</td>
<td>379</td>
<td>1342</td>
<td>3.54</td>
</tr>
<tr>
<td>Poly7</td>
<td>507</td>
<td>1511</td>
<td>2.98</td>
</tr>
</tbody>
</table>

**Table 10.6 Comparison of processing and overhead time**
As noted in Table 10.3, the additional time to transfer pixel data to the host and display the pixel data is less for the parallel implementation. This effect is reasonable in that the root processor can be sending the data while the segment filling processor is preparing additional pixel data. The parallel version also must transfer the same data between the transputers. However, it is also reasonable that the communications between transputers is faster than the communication between the root node and the host.

A second version of parallel Merge/Split was implemented that has a communications process on each filling node so that a node may be filling a segment while pixels and other data are being communicated. This implementation was tested with 1 and 2 nodes dedicated to filling segments.

![Figure 10.2 Three node transputer configuration](image)

Filling time for the test polygons is presented in Table 10.7 for a 2 node configuration (1 filling node) and a 3 node configuration (2 filling nodes) along with the previous parallel implementation for comparison. The data in the table includes vertex classification but not pixel display. The time required with 2 nodes is nearly the same as the previous configuration with 2 nodes where there was not a separate communication process running on the filling node(s). When 3 nodes are used, however, the extra communications through the middle node significantly increases the filling time.
Table 10.7 Comparison of filling time
filling nodes with and without communication processes

<table>
<thead>
<tr>
<th></th>
<th>Original parallel (2 nodes)</th>
<th>Communications process (2 nodes)</th>
<th>Communications process (3 nodes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poly5</td>
<td>1607</td>
<td>1610</td>
<td>4738</td>
</tr>
<tr>
<td>Poly6</td>
<td>2420</td>
<td>2440</td>
<td>4569</td>
</tr>
<tr>
<td>Poly7</td>
<td>3714</td>
<td>3761</td>
<td>6133</td>
</tr>
</tbody>
</table>

A third transputer configuration was tested where each processor with a filling process also had a communication process but each processor was physically connected to the root processor as illustrated in Figure 10.3. Table 10.8 contains timing data for this configuration along with comparison data repeated from Table 10.7. The presence of direct communication pathways improved filling time but not enough to compete with a sequential implementation.

Various additional configurations were tested with nearly identical results. These configurations included links between processors dedicated to segment filling since the software and data loader could automatically take advantage of multiple communication links, but produced very minimal improvement. A typical configuration is shown in Figure 10.4.
Figure 10.3 Transputer configuration with direct links

<table>
<thead>
<tr>
<th></th>
<th>Original parallel (2 nodes)</th>
<th>3 nodes</th>
<th>3 nodes with direct links</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poly5</td>
<td>1607</td>
<td>4738</td>
<td>2525</td>
</tr>
<tr>
<td>Poly6</td>
<td>2420</td>
<td>4569</td>
<td>3446</td>
</tr>
<tr>
<td>Poly7</td>
<td>3714</td>
<td>6133</td>
<td>4720</td>
</tr>
</tbody>
</table>

Table 10.8 Comparison of filling time
nodes with and without direct links
10.2.2 Sequent Implementation

The data presented in Table 10.4 includes a column for filling time that does not include downloading vertex data to the individual processors nor communicating pixel data to a root processor for further communication to a host. This type of communication pattern is more consistent with a shared memory model where all processors have access to the same instances of data. A shared memory implementation appears reasonable for a problem such as polygon filling, at least when considering a single polygon at a time, since there would not be write contention for pixels or updating of edge data but several processors could share the sorted vertex list and other data structures.

The transputer parallel code was modified to run on a Sequent shared memory processor. In this model, a process must classify the vertices and create a queue of segments to be filled. Then, a
number of processes on different processors could retrieve segments to be filled from a global queue. The scheduling of processors is implemented by the execution of a loop in each segment filling process which checks the queue for a new segment when it has finished a segment or when the system is starting.

The interesting part of shared memory parallel programming is properly protecting shared data that can be accessed by multiple processors. In this case, the global queue must be protected for both insert and removal operations since any filling process can either insert a new segment or remove a segment to fill at any time.

When designing an algorithm for a shared memory system, it is important to have as little shared data as possible and to understand how and when the shared memory needs to be protected. In the Merge/Split algorithm, shared memory includes the queue of segments waiting to be filled, the vertex data structure, and the sorted vertex list. The vertices are classified and the sorted vertex list is constructed as a sequential process before multiple filling processes are created.

After the initial sequential work is finished, the process uses the m_fork system call to create multiple instances of the segment filling function. There are a user-determined number of such processes. A possible alternative implementation would create an appropriate number of filling processes as needed.

The insertion and removal of segments from the queue is protected by locks. Another access where data is modified, and, thus, must be protected by locks, occurs when a MERGE vertex requires the unfinished edge of a segment to be stored with the vertex data structure until the other edge for the new segment is available.
The major memory contention is for writes to pixel memory. If polygons on separate processors do not overlap, this issue is not important. If the polygons overlap, a Z-buffer maintenance process or hardware could be utilized.

The shared memory Sequent implementation demonstrates that such an approach is reasonable and gives good performance.

When comparing performance for a multiple processor system the choice of metric is important. For example, should the filling work be measured in terms of number of polygons, number of polygon segments, total area of polygons, or number of vertices in the polygons? Since the focus of this work is processing complex polygon segments in a manner that is driven by vertices, either the number of segments or the number of vertices is most appropriate. There is a relationship, though not a constant nor easily described one, between the number of vertices in a polygon and the number of segments in a polygon.

A group of polygons can be measured in terms of number of vertices easily while determining the number of segments would require at least that the vertex classification processing be performed. The initial vertex classification, which is done sequentially, is directly proportional to the number of vertices. Therefore, total number of vertices is used as the standard measure. It would be possible to perform vertex classification in parallel with each node responsible for a set of vertices. If this approach were used the time required would be proportional to the number of vertices divided by the number of processors.

In order to make comparisons based on this metric, additional analysis of some sequential data was performed. Figure 10.5 is a graph of actual filling time (rather than ratio to scanline) versus number of vertices for the sequential implementation. The data was gathered on a Pentium II
system running Linux and is the average of filling 50 random polygons of each class. The time is scaled so that filling the smallest convex polygon with 4 sides is assigned a value of 1. The data is for the integer version of Merge/Split which corresponds to the parallel implementations.

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Scaled time – convex</th>
<th>Scaled time – maximum segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>1.10</td>
</tr>
<tr>
<td>5</td>
<td>1.11</td>
<td>1.22</td>
</tr>
<tr>
<td>10</td>
<td>1.36</td>
<td>1.67</td>
</tr>
<tr>
<td>15</td>
<td>1.56</td>
<td>2.20</td>
</tr>
<tr>
<td>20</td>
<td>1.51</td>
<td>2.71</td>
</tr>
<tr>
<td>25</td>
<td>1.93</td>
<td>3.31</td>
</tr>
<tr>
<td>30</td>
<td>2.16</td>
<td>3.99</td>
</tr>
</tbody>
</table>

Table 10.9 Sequential fill time versus number of vertices

Figure 10.5 plots filling time per vertex as a function of number of vertices. The data is scaled as in Figure 10.5 so that a 4 vertex polygon with an assumed time of 1, has a value of 0.25.
Table 10.10 – Sequential fill time per vertex

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Scaled time – convex</th>
<th>Scaled time – maximum segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.25</td>
<td>0.28</td>
</tr>
<tr>
<td>5</td>
<td>0.22</td>
<td>0.24</td>
</tr>
<tr>
<td>10</td>
<td>0.14</td>
<td>0.17</td>
</tr>
<tr>
<td>15</td>
<td>0.10</td>
<td>0.15</td>
</tr>
<tr>
<td>20</td>
<td>0.076</td>
<td>0.14</td>
</tr>
<tr>
<td>25</td>
<td>0.077</td>
<td>0.13</td>
</tr>
<tr>
<td>30</td>
<td>0.072</td>
<td>0.13</td>
</tr>
</tbody>
</table>

Figure 10.6 Sequential fill time per vertex

Note that fill time per vertex becomes nearly constant after approximately 15 vertices are involved.

For the parallel implementation, a processor was added for each additional pair of vertices so that a 5 vertex polygon used 1 processor and a 7 vertex polygon used 2 processors. The fill time measured by execution time on the busiest processor remained essentially constant. Therefore,
for the situation where an adequate number of processors is available, the calculated speedup is very close to the theoretical value which is the actual number of processors.

Figure 10.7 plots speedup versus number of processors for the situation described above.

10.2.3 Potential SIMD Implementation

A SIMD implementation might be feasible and has been considered, but not implemented. In a SIMD system a number of processors are synchronized at the instruction level so that each is executing the same instruction at a given time. If a processor is not executing a part of the algorithm because of a conditional statement, it is idle until the instruction streams merge.
The vertex classification work could be done in a sequential manner before the filling process began and the filling algorithm could be executed for different segments on different processors. Since there are many paths through the code, many processors might be idle at some time during the filling process. In addition, if new polygons were to be introduced during execution, the vertex classification process would need to be done on a different processor.

In summary, a SIMD implementation would be feasible if the system were a hybrid system in the sense that vertex classification could be performed on a separate sequential processor with a queue of segments available for the SIMD array. A large number of polygon segments would be necessary to make the process efficient since SIMD machines typically have a large number of processors (perhaps 512 or more) and a certain fraction would be idle at a given time because of the many possible code paths.

10.3 CONCLUSIONS

The parallel polygon filling algorithms presented have been implemented and tested in a somewhat artificial manner in that it is not practical to use several general purpose cpus of a parallel processing system to fill polygons for immediate display. The results would be meaningful in one of two contexts: filling of large numbers of complicated polygons "off-line" or using hardware designed to implement the algorithms directly. The steps in the algorithm are straight-forward and rely on simple integer arithmetic so that a hardware implementation might be feasible.

A hardware implementation would not be based on a distributed memory message-passing paradigm, but on a shared memory (perhaps SMP) approach. In this context, the processors could write directly to display memory. Speed and proper data protection of the hardware
implementation would rely on hardware protection as is currently performed in any dedicated graphics system that uses multiple processing units to generate output pixels.

The consideration of such a system would be based on performance of the parallel polygon filling algorithm implemented in software. As documented in the sections above, the shared memory system would be a valid candidate for such consideration. The distributed memory (transputer) implementation needs to perform too much data communication relative to processing time to be reasonable.
CHAPTER 11 - POLYGON FILLING CONCLUSIONS

11.1 CONTRIBUTIONS

The following new approaches to polygon filling have been developed:

- A point based approach to polygon filling that is driven by vertices rather than edges has been presented. The concept of classifying vertices as an aid to processing has been extended. Two algorithms have resulted.

- One of the algorithms, Merge/Split, is conceptually different from other polygon filling algorithms. The other algorithm, AEL, is similar in many respects to the standard scanline approach.

- The algorithms, especially Merge/Split, have natural parallel implementations. Parallel Merge/Split, has been implemented in both a distributed memory and a shared memory model.

- The concept of polygon filling has been extended to generalized polygons that allow edges to be any type of curve that can be rasterized in single step increments.
The algorithms produce correct results for all polygons tested including polygons with many segments, polygons with many transitions at the same scan line, polygons with interior holes, polygons with curved edges, and polygons with various combinations of these features.

The algorithms are more efficient than scanline with an average time ratio of 0.281 for a wide variety of polygons. The Merge/Split algorithm is slightly more efficient than AEL for most polygons.

The shared memory parallel implementation of Merge/Split performs very well. The distributed memory implementation is less efficient than a sequential algorithm due to the amount of communication relative to arithmetic performed.

11.2 FUTURE WORK

Additional parallel implementations are possible. A SIMD implementation would be interesting as a group of processors could each be filling a polygon segment at the same time. Several issues would need to be considered including how to start new segments and deal with any change caused by a vertex at the end of an edge. To be practical such an implementation would need to spend the great majority of its time filling segments rather than processing vertices that alter segments.

A thread based implementation on a Symmetric Multiple Processor would be very similar to the shared memory implementation but would have the advantage of much lower context switch times.
Another area for exploration would be filling multiple polygons at the same time on a parallel system so that there would not be a fixed relationship between a polygon and a processor.

More careful analysis of the parallel algorithms could result in motivation to consider a hardware based implementation.