A Parallel Algorithm for Polygon Rasterization

Juan Pineda
Apollo Computer Inc.
Chelmsford, MA 01824
juan@apollo.uucp

Abstract

A parallel algorithm for the rasterization of polygons is presented that is particularly well suited for 3D Z-buffered graphics implementations. The algorithm represents each edge of a polygon by a linear **edge function** that has a value greater than zero on one side of the edge and less than zero on the opposite side. The value of the function can be interpolated with hardware similar to hardware required to interpolate color and Z pixel values. In addition, the edge function of adjacent pixels may be easily computed in parallel. The coefficients of the "Edge function" can be computed from floating point endpoints in such a way that sub-pixel precision of the endpoints can be retained in an elegant way.

CR catagories and subject descriptors: I.3.1 [Computer Graphics]: Hardware Architecture - *Raster display devices*; I.3.3 [Computer Graphics]: Picture/Image Generation - *Display algorithms*.

General terms: Algorithms.

Additional keywords and phrases: Polygon rasterization, sub-pixel vertices, linear edge function, parallel processing.

1. Introduction

The fast rendering of 3D Z-buffered linearly interpolated polygons is a problem that is fundamental to state of the art workstations. In general, the problem consists of two parts: 1) the 3D transformation, projection and light calculation of the vertices, and 2) the rasterization of the polygon into a frame buffer. This paper deals with one aspect of the latter problem: the computation of the boundaries of the polygon.

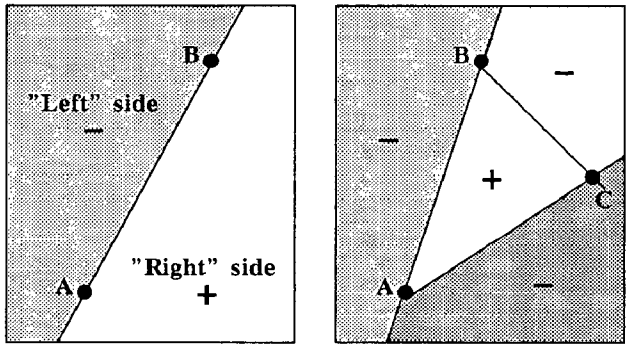
Traditionally, the edges of a polygon are computed by a line interpolation algorithm, and each scan line is filled with linearly interpolated color and Z values [2]. This method is generally scan line serial, and is consequently not so convenient for frame buffers with more desirable rectangular word organizations [5].

The "PIXEL-PLANES" [3] system uses a parallel multiplier tree to simultaenously compute, for all pixels in the frame buffer, a linear function which is used to define edges. This method has the nice property that it is highly parallel, but it has the disadvantage that it requires dedicated logic for each pixel, and consequently requires custom memory chips.

The algorithm presented in this paper also uses a linear function to define polygon edges, but it allows for painting algorithms that are better suited to frame buffers using conventional DRAM and VRAM technology. The algorithm is inherently parallel, so that the rendering performance is memory bandwidth limited, rather than computation limited.

 ACM-0-89791-275-6/88/008/0017 $00.75

The **edge function** is a linear function which can be used to classify points on a 2D plane that is subdivided by a line, into three regions: the points to the "left" of the line, the points to the "right" of the line, and the points on the line. The function has the property that points to the "left" of the line have a value greater than zero, points to the "right" have a value less than zero, and points exactly on the line have a value of zero. Since the function is linear, it can be computed incrementally in the same way as color and Z values.



Subdivision of plane by line through points A and B

Triangle formed by union of right sides of AB, BC and CA

Figure 1. A Triangle Can be Formed by Combination of Edges

Figure 1 shows how it is possible to define a triangle by the union of three edges which are specified by edge functions. It is possible to define more complex polygons by using boolean combinations of more than three edges. Note that a tie breaker rule must be applied to the points that lie exactly on any of the edges to determine whether the points are to be considered interior or exterior to the polygon.

With this formalism, it is possible to compute at each pixel center on the plane an n-tupple: (R, G, B, Z, E1...En), where R, G, B and Z components form the fill value, and E1...En are the values of the edge functions which are used to determine whether the pixel is interior or exterior to the polygon. Given the value of this n-tupple at a single pixel position, the n-tupple of adjacent pixels can be computed by simple linear interpolators that require one addition per component per iteration.

E1..En can then be used as a "stencil" that allows a pixel to be modified only if it is interior to the polygon. The process of painting the polygon can then be reduced to an algorithm that traverses an area that includes the interior of the triangle, but that does not have

to be particularly careful about the edges because the "stencil" forms the actual edge. The particular order of traversal is not important, only that each interior pixel is covered once and only once.

Any traversal algorithm that touches all interior points once and only once will produce the correct result, but some may be more efficent than others, depending on how many pixels are covered by the traversal that are not actually drawn.

The elegance of this approach is in the way that it orthogonalizes and unifies the traversal of a polygon and the filling of interior pixels. The orthogonality is convenient for the formulation of efficient strategies for painting a polygon with the least number of memory cycles and is especially useful with parallel rectangular memory organizations.

2. The Edge Function

Consider, as shown in figure 2, a vector defined by two points: (X,Y) and (X+dX,Y+dY), and the line that passes through both points. This vector and line can be used to divide the two dimensional space into three regions: all points to the "left" of, to the "right" of, and exactly on the line.

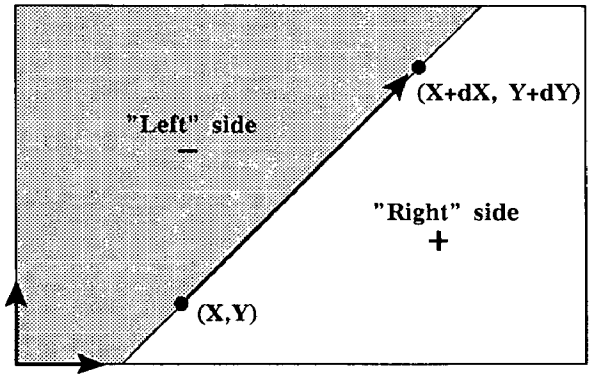


Figure 2. An Edge is Defined by a Vector

We define the **edge function** E(x,y) as:

$$E(x,y) = (x-X)\,dY - (y-Y)\,dX$$

This function has the useful property that its value is related to the position of the point (x,y) relative to the edge defined by the points (X,Y) and (X+dX, Y+dY):

E(x,y) > 0 if (x,y) is to the "right" side
E(x,y) = 0 if (x,y) is exactly on the line
E(x,y) < 0 if (x,y) is to the "left " side

To convince oneself that this is true, recognize that the formula given for E(x,y) is the same as the formula for the magnatude of the cross product between the vector from (X,Y) to (X+dX, Y+dy), and the vector from (X,Y) to (x,y). By the well know property of cross products, the magnatude is zero if the vectors are colinear, and changes sign as the vectors cross from one side to the other.

This function is convenient for rasterization algorithms, since it can be computed incrementally by simple addition:

$$E(x+1,y) = E(x,y) + dY$$
$$E(x,y+1) = E(x,y) - dX$$

The edge function is related to the error value or "draw control variable" (DCV) in Bresenham line drawing algorithms [1,2]. The difference is that Bresenham line drawing algorithms maintain the DCV value only for pixels within 1/2 pixel of the line, while E(x,y) is defined for all pixels on the plane. In addition, the value of the DCV at a given point differs from E(x,y) by a constant offset. In any case, the reason that both algorithms work is fundamentally the same.

As mentioned earlier, this same property of E(x,y) is used by the "PIXEL-PLANES" [3] graphics system, where this function is computed in parallel for all pixels in the frame buffer by a multiplier tree.

3. Incremental Classification of Points around a Convex Polygon

Consider a convex polygon defined by the vertices (Xi, Yi) 0< i <=N. For the convenience of notation, take (X0, Y0) = (XN, YN), and consider the i'th edge as the edge between the i'th and the [i–1] vertex. The initial values of the edge function interpolators at a starting point (Xs, Ys) would then be:

$$dXi = Xi - X[i-1]$$
$$dYi = Yi - Y[i-1]$$
$$Ei(Xs, Ys) = (Xs - Xi)\,dYi - (Ys - Yi)\,dXi$$

for 0< i <=N

The edge functions may then be computed incrementally for a unit step in the X or Y direction:

$$Ei(x+1, y) = Ei(x, y) + dYi,$$
$$Ei(x-1, y) = Ei(x, y) - dYi,$$
$$Ei(x, y+1) = Ei(x, y) - dXi,$$
$$Ei(x, y-1) = Ei(x, y) + dXi.$$

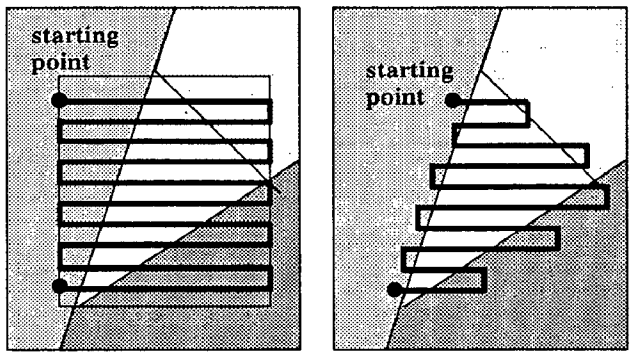
If we use a tie breaker rule that considers a point on an edge as interior to the edge, then a point is interior to the convex polygon if:

$$Ei >= 0 \text{ for all } i : 0<i<=N .$$

4. Traversing the Polygon

Given the initialized edge interpolators, the interpolation coefficients, the tie breaker rule, and the Boolean function for combining the edges, we still need to traverse the area of the triangle in order to paint it.

The polygon can be traversed by any algorithm that is guaranteed to cover all of the pixels. Figure 3 shows two simple algorithms. Simply traversing the bounding box is perhaps the simplest strategy, but generally not the most efficient. A smarter algorithm would advance to the next line when it walked off the edge of a triangle.



Traversing the Bounding Box *A More Efficient Traversal Algorithm*

Figure 3. Simple Traversal Algorithms

One complicaton of the smart algorithm is that when it advances to the next line, it may advance to a point inside the triangle. In that case, the algorithm must search for the outside of the edge before it begins the next scan line. An example of this problem is shown on the top right hand edge of the triangle in figure 4.

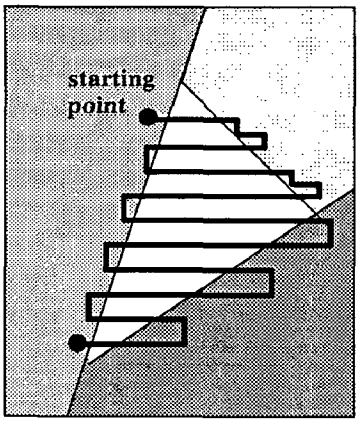


Figure 4. Traversal Algorithms May Have to Search for Edge

A smarter algorithm is shown in figure 5. It proceeds down from the starting point, working its way outward from a center line. The advantage of this algorithm over the simpler algorithm is that it never has to search for an edge, then double back. The tradeoff is that the interpolator state for the center line must be saved while traversing the outer points, since the interpolators must be restarted back at the center line. Notice that at the bottom, the "center" line shifts over if it ends up exterior to the triangle.

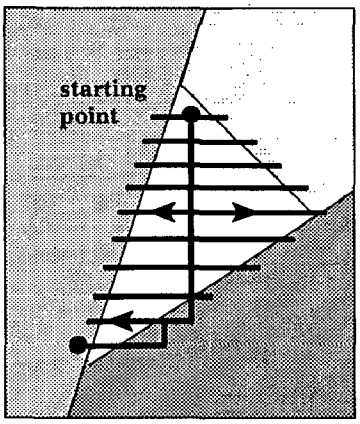


Figure 5. Smarter Algorithm Proceeds Outward From Center Line

There are many traversal algorithms possible. The best algorithm will depend on the cost/performance tradeoffs in the implementation.

5. Clipping

Left and right clipping can be viewed as additional polygon edges that are part of the pixel's value: (R, G, B, Z, E1..En, El, Er), where El and Er represent the left and right clip "edge functions". If the traversal algorithm views them as edges, then a smart traversal algorithm will turn back when it crosses a clip boundary and it will not spend time rendering clipped areas of a polygon.

The top clip boundary can be used to control the starting point, while the bottom clip boundary can be used to control the last scan line rendered.

Figure 6 shows clipping.

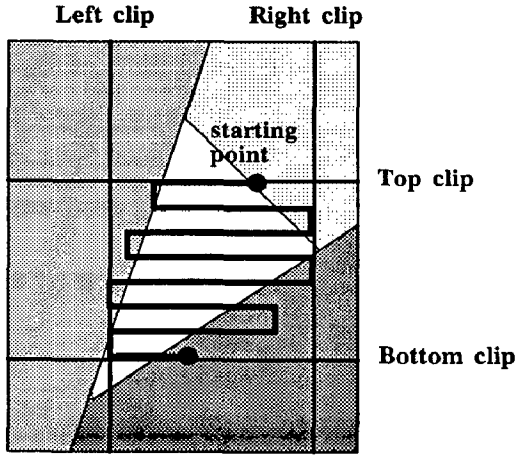


Figure 6. Clipping a Triangle

6. Sub-pixel Accuracy of Vertices

Typically in 3D graphics rendering, polygon vertices are in floating point format after 3D transformation and projection. Some implementations round the X and Y floating point ordinates to integer values, so that simple integer line algorithms can be used compute the triangle edges. This rounding can leave gaps on the order of 1/2 pixel wide between adjacent polygons that do not share common vertices. Gaps also occur as a result of the finite precision used in specifying the endpoints, but these gaps are much narrower. Some implementations attempt to eliminate these gaps by growing the edges of triangles to insure overlap, but these solutions cause other artifacts.

In order to minimize these artifacts, it is desirable to render triangle edges as close as possible to the real line between two vertices. This is conveniently done with this algorithm by performing the interpolator setup computations in floating point, and converting to fixed point at the end:

$$dXi = Xi - X[i-1]$$
$$dYi = Yi - Y[i-1]$$
$$Ei(Xs, Ys) = (Xs - Xi)\, dYi - (Ys - Yi)\, dXi$$

$$dXi' = FIX(dXi)$$
$$dYi' = FIX(dYi)$$
$$Ei' = FIX(Ei)$$

Note that as in any digital interpolator, the fractional precision used in the iteration must be chosen to give an acceptable error across the interpolation.

While this computation does require five floating point additions and two floating point multiplies per edge, the cost is small when compared with the other computations required to transform and set up a 3D triangle.

Notice that the computation only modifies the setup values of the Ei's, but does not require any special treatment of the endpoints, except to insure that the traversal algorithm covers the entire area including the endpoints.

7. Parallel Implementation

Since the edge function is linear, it is possible to compute the value of the edge function for a pixel an arbitrary distance L away from a given point (x,y):

$$E(x+L, y) = E(x) + L\, dy$$

This property allows a group of interpolators, each responsible for a pixel within a block of contiguous pixels, to simultaneously compute the edge function of an adjacent block in a single cycle. If the blocks were L pixels wide, then there would be L interpolators. In order to compute the edge function of the block L pixels away in the +x direction, each interpolator would increment by (L dx).

Since color and Z components are linear as well, they may also be computed in parallel.

Graphics frame buffers are usually organized to provide simultaneous access to a block of adjacent pixels [5]. The block is usually called a word, and the pixels within the block are called interleaves. If a group of interpolators are dedicated to each interleave, then the RGBZ value and whether the pixel should be drawn can be computed in parallel for an entire word. If the interpolator cycle time is at least as fast as the memory cycle time (which is the case with current gate array and DRAM technology), then shaded triangles can be rendered at the memory cycle time.

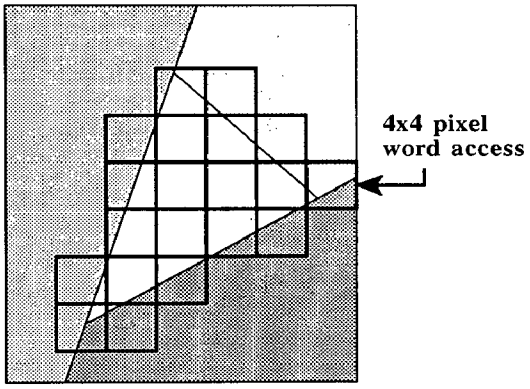


Figure 7. Covering a Triangle by Rectangular Accesses

8. Future Work and Extensions

The edge functions described in this paper have been linear. It is possible to compute higher order edge functions. A second order edge function would yield conic edges such as circles or elipses. Combined with more interesting Boolean functions, it would be possible to efficiently compute complex shapes, such as wide lines with rounded endpoints.

By proper normalization of the partial derivatives and intial value of E(x,y), it is possible to perform the interpolation of the edge functions in a floating point like manner, thus maximizing the precision obtained from a finite width interpolator. This method has the desirable property that it gradually looses precision as triangles get larger, rather than abruptly breaking.

Because triangle edges are specified by the coefficients in the edge function, rather than end points, it is actually possible to transform the coefficients directly, rather than forming a triangle from the transformed and projected vertices. While this does not directly save computation, it may reduce some of the computational difficulties encountered in perspective projection. Specifically, since the objects being transformed are edges rather than points, the precision and dynamic range problems encountered with perspective projection of points near the eye point or behind the hither plane can be eliminated. This means that it may be possible to transform polygons without the need for exception cases for vertices behind the hither plane. This property could be useful in pipeline implementations where the exception case would limit performance.

Since the value of the edge function is proportional to the distance of a point from the edge, it is possible to use this value to anti-alias edges. This could be performed using a method proposed by Fujimoto and Iwata [4]. In this method, the Bresenham DCV value and increments for a polygon edge are put through a lookup table that performs a divide and computes a low precision estimate of the distance of the pixel center from the edge center line. This distance is then used to adjust the contribution of the fill value to the background value.

Going one step further with anti-aliasing, it is possible to have the lookup table produce a crude sub-pixel resolution bitmap for each edge. The bitmaps of all edges would then be anded together, and the number of set sub-pixels would be counted to determine the contribution for that pixel. Since the method aproximates the actual sub-pixel bitmap for the triangle, it has excellent behavior at verticies, and at places where edges are less than 1 pixel apart.

9. Conclusion

An algorithm for rasterization of polygon edges has been presented. The algorithm has several useful properties: 1) it can be conveniently computed in parallel and used with common refresh buffer word organizations, 2) it can be computed with hardware similar to hardware required to interpolate color and Z values for 3D solids, and 3) it elegantly maintains the subpixel accuracy of vertices. These properties make the algorithm particularly attractive for use in 3D solid graphics implementations.

10. Acknowledgements

I would like to give special thanks to Bill Brandt for time spent on numerous discussions during the intial formulation of the ideas presented here. I would also like to thank Casey Dowdell, Bill Brandt, John Beck, Jane Critchlow and the conference reviewers for their time spent reviewing this text and for their helpful suggestions. Finally, I would like to thank Kathy Ford for help in the preparation of the final copy for publication.

References

1. Bresenham, J. Algorithm for Computer Control of Digital Plotter. *IBM Systems Journal 4,1* (1965), 25-30.
2. Foley, J. and A. Van Dam, "Fundamentals of Interactive Computer Graphics."
3. Fuchs, H. and Poulton J. PIXEL-PLANES: A VLSI-Oriented Design for a Raster Graphics Engine. *VLSI DESIGN (Third Quarter 1981)*, 20-28.
4. Fujimoto, A. and Iwata, K. Jag-Free Images on Raster Displays. *IEEE Computer Graphics and Applications 3,9* (December 1983), 26-34.
5. Whitton, M. Memory Design for Raster Graphics Displays. *IEEE Computer Graphics and Applications 4,3* (March 1984), 48-65.