Lecture 9

Scan Conversion: Polygons

1 Introduction

This chapter discusses scan conversion of polygons to pixels. This is commonly referred to as tiling the polygon. We discuss this is some detail because it is very tricky to both get right and make efficient. If it is not done right, gaps and other artifacts may occur, and if it is not efficient, the graphics system will be very slow.

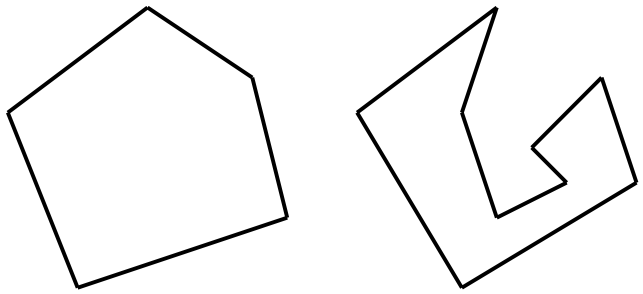
The approach we take will be based on point sampling or hit testing, i.e., finding the set of pixels that are inside the polygon. There exist other criteria for determining what pixels to output, but these criteria either have no precise geometric definition of the set of pixels comprising the rasterized polygon, or they can be transformed into point sampling after first distorting the polygon. For example, finding all the pixels within a unit distance of the polygon is equivalent to convolving the polygon with a circle of unit radius and point sampling the result.

We begin by considering two different but equivalent inside tests that are related to the two common representations of polygons. These inside tests are carefully formulated so that singularities are handled correctly. We then derive more efficient *scanline* or *sweepline* algorithms that take advantage of spatial coherence, but which are equivalent to those based on *ab initio* inside testing. Finally, we discuss methods for interpolating colors and other values across polygons and for antialiasing polygonal edges.

2 Polygons and Insidedness

Let's begin with a discussion of the basic types of polygons as shown in Figure 1. *Simple* polygons are those comprised of a single loop of vertices, but whose edges are do not intersect one another. Simple polygons can be represented by a single array of n vertices. Simple polygons include both *convex* and *concave* polygons. A convex polygon has the property that a line can only intersect its boundary in two places; equivalently, a line connecting two interior points can never pass outside the polygon. Convex polygons also have the property that the interior angle at all vertices is acute ($< 180°$). Concave polygons don't have these properties. Special purpose fast algorithms are often possible if the number of vertices are fixed. The most common examples are triangles and quadrilaterals. Fixing the number of vertices allows the program to preallocate storage or to unroll loops that depend on the number of vertices. Triangles are always convex, whereas quadrilaterals may be convex or concave. Thus, restricting the number of vertices may restrict the type of the polygon and lead to simpler, more efficient tilers. If the edges of a polygon intersect one another the polygon is said to be *self-intersecting*. To be robust, the system should produce predictable results when given a self-intersecting polygon. A *general* polygon is one consisting of multiple loops of vertices. This implies that the polygon may have several disconnected regions and each of these may contain holes.

Let's start off with a scan conversion algorithm for triangles. We wish to generate all the samples inside a triangle. The easiest and most inefficient way to do this is to test all the points in the image to see if they are inside the triangle. This is very inefficient if the area of the triangle is small compared to the size of the


Simple Polygons

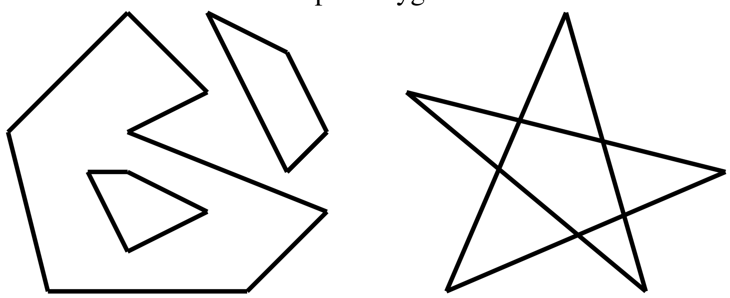

General Polygons

Figure 1: Basic Polygon Types

image. This simple algorithm can be improved on by considering only points inside the bounding box of the triangle.

```
void
tile3( vert v[3] )
{
    int x, y;
    int xmin, xmax, ymin, ymax;
    bbox b;

    bound3( v, &b );
    xmin = ceil(b.min.x);
    xmax = ceil(b.max.x);
    ymin = ceil(b.min.y);
    ymax = ceil(b.max.y);

    for( y=ymin; y<ymax; y++ )
        for( x=xmin; x<xmax; x++ )
            if( inside3(v,x,y) )
                plot( x, y );
}
```

The procedure `bound3` computes the minimum and maximum extents of the triangle in x and y. The

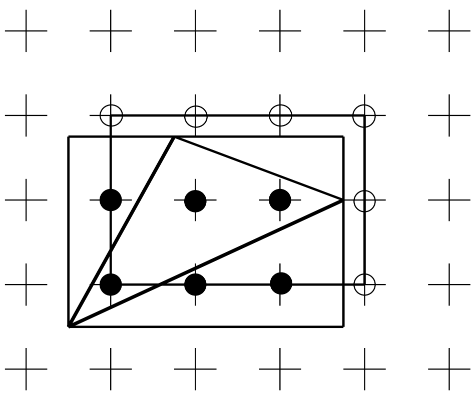


Figure 2: Potential Inside Pixels

coordinates of the vertices of the triangle are of type `value`, which may be a float, an int, or a fixed point number. If the type is not an integer, the polygon vertices are not required to fall on pixel centers and in general occupy subpixel position. To ensure that no inside pixels are missed, all pixels inside the bounding box must be tested (see Figure 2). One convenient way to do this is to find the *ceiling* of both the minimum and maximum values, and then consider all the integer values within the half open interval $[\lceil min \rceil, \lceil max \rceil)$. If the minimum and maximum lie in general positions, this interval contains all the pixels. However, if the maximum falls exactly at an integer. choosing the half-open interval causes the pixel with this value to be excluded. As will be seen this is a *feature*, not a bug.

How can we determine if a point is inside a triangle? There are two common ways of doing this which reflect the two common ways to represent polygons (and polyhedra in 3D).

2.1 Insidedness: Set Operations

The first method considers the triangle to be the intersection of three half-spaces. Recall that a line divides the plane into three regions: those points on the line, inside the line, and outside the line.

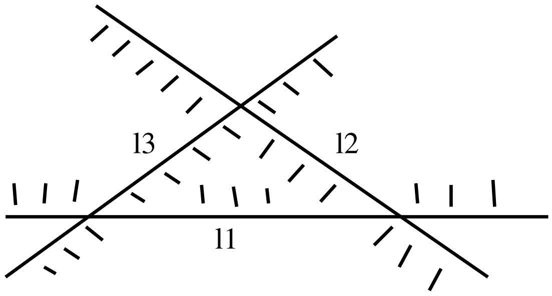
$$l(x,y) = ax + by + c \begin{cases} < 0 & \text{inside} \\ = 0 & \text{on} \\ > 0 & \text{outside} \end{cases}$$

Thus, substituting a point's coordinates into the line equation and then testing the resulting sign provides a simple and efficient inside test for the primitive half-space represented by the line. This process of determining where a point lies is called a *point membership classification*.

The three edges of a triangle are part of three lines.

$$\begin{aligned} l_1(x,y) &= a_1x + b_1y + c_1 \\ l_2(x,y) &= a_2x + b_2y + c_2 \\ l_3(x,y) &= a_3x + b_3y + c_3 \end{aligned}$$

The lines need to be oriented so that their inside contains the triangle.



Computing the line equation as

$$
\begin{aligned}
a_3 &= y_2 - y_1 \\
b_3 &= x_1 - x_2 \\
c_3 &= x_2 y_1 - x_1 y_2
\end{aligned}
$$

will orient the line so that its inside is on the left as we move from the first point to the second point. So if the triangle is described by three vertices, we can form its description in terms of three lines by considering consecutive pairs of vertices as we move counterclockwise around the triangle. A triangle inside test can be performed by testing whether the point is inside each of the three halfspaces using the appropriate point membership classification function and then forming the Boolean *and* of the three results.

This yields the following algorithm:

```
void
mkedge( vert *v1, vert *v2, value *a, value *b, value *c )
{
    *a = v2->y - v1->y;
    *b = v1->x - v2->x;
    *c = v2->x * v1->y - v1->x * v2->y;
}
```

```
#define inside(e) (e) <= 0

tile3( vert v[3] )
    value a1, b1, c1, e1;
    value a2, b2, c2, e2;
    value a3, b3, c3, e3;
    bbox b;
    int x, y;
    int xmin, xmax, ymin, ymax;

    /* bound and compute range */
    ...

    mkedge( &v[0], &v[1], &a1, &b1, &c1 );
    mkedge( &v[1], &v[2], &a2, &b2, &c2 );
    mkedge( &v[2], &v[0], &a3, &b3, &c3 );

    e1 = a1 * xmin + b1 * ymin + c1;
    e2 = a2 * xmin + b2 * ymin + c2;
    e3 = a3 * xmin + b3 * ymin + c3;
    for( y=ymin; y<ymax; y++ ) {
        for( x=xmin; x<xmax; x++ ) {
            if( inside(e1) && inside(e2) && inside(e3) )
                plot( x, y );
            e1 += a1;
            e2 += a2;
            e3 += a3;
        }
        e1 += (xmin-xmax)*a1 + b1;
        e2 += (xmin-xmax)*a2 + b2;
        e3 += (xmin-xmax)*a3 + b3;
    }
}
```

Notice how the value of the line equation can be computed incrementally by taking advantage of the step-wise nature of the scan.

Very often an object is drawn as a collection of polygons that are part of a topological mesh where polygons share common vertices and edges. When scan converting such a collection of polygons, it is very important that all the pixels along common edges be output. If some are missed, the gaps between adjacent polygons cause ugly drop outs in the resulting image. It is also important that no pixels be output twice. This is often referred to as a double hit, and not only results in excess work, but also must be avoided if the polygon is being drawn with transparency. A double hit causes what is underneath that point to be attenuated twice, whereas a single hit would attenuate it only once. (Double hits must also be avoided when performing CSG evaluation). One big advantage of using the inside criteria for scan conversion is that a pixel is only

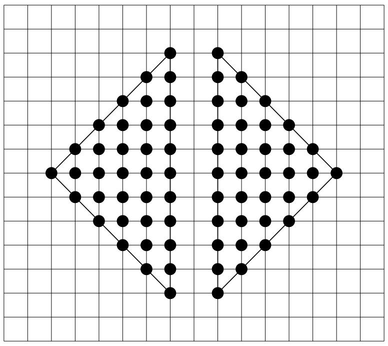


Figure 3: Common Edge Between Two Triangles

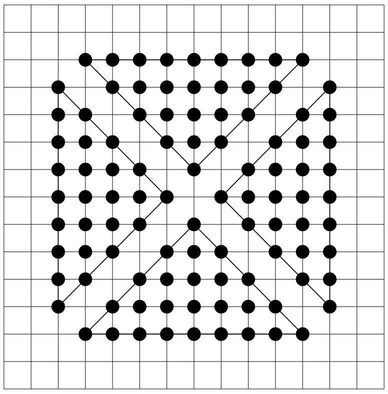
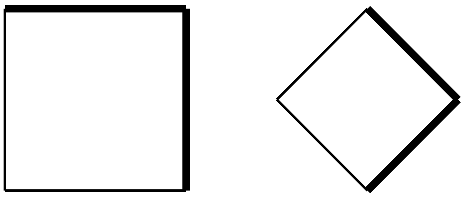


Figure 4: Common Vertex Shared by Four Triangles

output if it is inside a polygon. Assuming the polygons don't fold over or overlap, this implies that each pixel will be output only once.

This argument holds as long as there are no *singularities*. A singularity occurs if a pixel falls *exactly* on an edge or a vertex. When scan converting, we need to answer the question: Should a pixel be output or not? On the other hand, the point membership classification function returns three values: *inside*, *outside*, or *on*. The above algorithm treats the *on* case as being inside. But this means that pixels falling on edges between polygons will be output twice and pixels falling on vertices are output once per polygon incident to the vertex. Figure 3 shows two triangles; note that if the right triangle were to be moved left one pixel so that the two triangles abutted, then all the pixels along the common edge would be output twice. Figure 4 illustrates a similar situation involving four triangles sharing a vertex. If these four triangles were to be pushed together, then all the pixels along the edges would be output twice, and the pixel at the common vertex would be output four times.

One way around the multiple hit problem is to split polygon edges into two groups: shadowed and non-shadowed edges. Shadowed edges are considered to be outside of the polygon, whereas non-shadowed edges are considered to be inside. If we can guarantee that an edge common to two polygons will be shadowed once and illuminated the other time, then pixels on the edge will be output only once. The simplest method for separating the edges, is to consider those facing right to be in shadow and those facing left to be illuminated. If an edge is horizontal, it is considered in shadow if it facing upward and illuminated if it is facing downward. This is illustrated below for a square and a diamond:



Here the shadowed edges are shown as thick dark lines.

Recall that the first two coefficients of the line equation define a vector normal to the line facing outward. We consider an edge to be facing in the direction of its normal. Thus, the above shadow test can be implemented with the following code.

```
int shadow( value a, value b )
{
    return (a > 0) || (a == 0 && b > 0)
}
```

If two adjacent polygons are both oriented in the same way, either clockwise or counterclockwise, then the normals to the common edge will point in opposite directions. The above shadow test guarantees that normals in opposite directions are classified differently, and hence a common edge will be considered in shadow once and illuminated once.

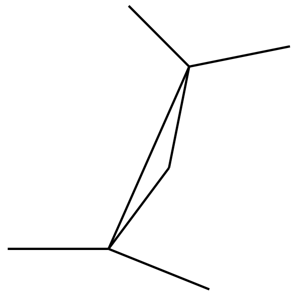
This refinement leads to the following inside test.

```
int inside( value e, value a, value b )
{
    return (e == 0) ? !shadow(a,b) : (e < 0);
}
```

The effect of shadowing is that pixels on the right edge of a polygon are considered outside and pixels on the left edge are considered inside. This inside criterion is very similar to the use of half-open intervals. When two polygons abut along an edge, that edge will be on the left side of one polygon and on the right side of the other polygon, so pixels on the edge can only be inside one of the polygons. When a pixel falls on a vertex, it will be considered inside a polygon only if both edges incident to the vertex are left edges. But this can only happen for one polygon, so again that pixel is output only once.

Correctly handling singularities is very important otherwise the resulting gaps and double hits cause unsightly dinks and drop outs in the scan converted imagery. How often do such singularities occur? This depends on the input data, in particular, the location and precision of vertices. Vertex coordinates can be represented in floating point, fixed point, or as integers. The lower the precision of the data, the more likely are singularities. Unfortunately, most implementations of scan conversion map vertices to integers, throwing away precision and hence increasing the likelihood of artifacts. Moreover, many people think increasing the precision completely prevents artifacts, but this is not the case.

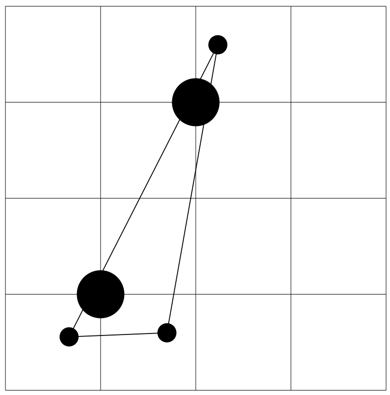
There is another situation that can cause gaps and drop outs. This is when t-vertices occur. A t-vertex occurs when the topology along an edge separating two polygons is not the same. One polygon has a single edge, whereas the other polygon's edge has been divided into two pieces by the t-vertex.



This situation is very common when approximating curved surfaces with polygons. Sometimes the curved surface is first divided into pieces separated by boundary curves and then each piece is converted into polygons. This means that the boundary curve is approximated by straight edges. If each piece of the surface uses a different number of edges to approximate the boundary curve, then the vertices on each curve do not line up and t-vertices may occur. The problem with t-vertices is that because of limited precision, the t-vertex may not lie exactly on the other polygon's edge. This causes a crack between the three edges.

Finally, let us conclude with a few other points.

1. Notice that the set of pixels inside the polygon is not necessarily connected. That is, some pixels may exist as isolated islands. This is a form of aliasing caused by sampling very small objects.

   If a small polygon is part of a larger mesh, then these isolated pixels will eventually be joined together. However, if the small polygon is not connected to other polygons, then these isolated pixels are not always desirable. For example, characters are normally drawn in isolation and also can contain very narrow stems and serifs. Other scan conversion algorithms guarantee that the pixels produced are connected. The most common way to do this is to always output at least one pixel per scan line.

2. The above algorithms generate sample locations that are always at integers. Assuming a constant step size, this allows the line equation to be computed incrementally. However, the inside test itself does not depend on this and can be used to test whether any point is inside the triangle. Thus, this algorithm is ideal if we wish to sample a polygon with points that aren't uniformly spaced or that come in random order.

3. Another advantage of these algorithms is that they are easy to parallelize. Consider assigning processors to individual pixels by interleaving 4 x 4 processors in a checkerboard-like pattern:

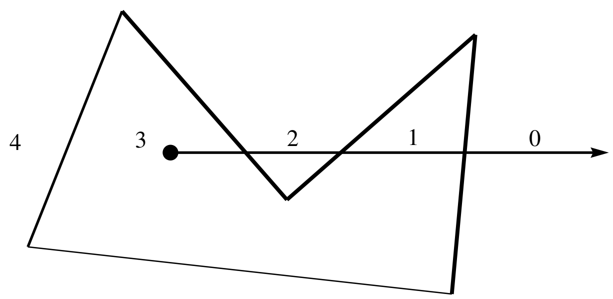
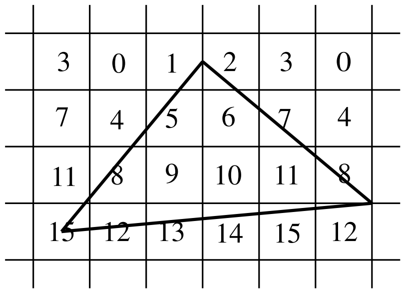


Figure 5: Jordan Curve Theorem



Each hit test may be done independently of the others and so the algorithm runs fully in parallel.

2.2 Insidedness: Jordan Curve Theorem

The second method for determining whether a point is inside a polygon is to count the number of times a ray emanating from the point intersects the polygon's boundary (see Figure 5). If the ray intersects the boundary an even number of times, the point is outside the polygon; on the other hand, if the ray intersects an odd number of times the point in inside the polygon. This test is based on the Jordan Curve Theorem. Note that this inside test works whether the polygon is convex, concave, or concave with holes.

The procedure `inside` implements this test for any simple polygon.

```
int inside ( vert v[], int n, value x, value y )
{
    int cross;
    value x0, y0;
    value x1, y1;

    cross = 0;
    x0 = v[n-1].x - x;
    y0 = v[n-1].y - y;
    while( n-- ) {
        x1 = v->x - x;
        y1 = v->y - y;
        if( y0 > 0 ) {
            if( y1 <= 0 )
                if( x1*y0 > y1*x0 )
                    cross++;
        }
        else {
            if( y1 >  0 )
                if( x0*y1 > y0*x1 )
                    cross++;
        }
        x0 = x1;
        y0 = y1;
        v++;
    }
    return cross & 1;
}
```

The procedure `inside` shoots a ray from the sample point to the right. To simplify the calculation the polygon is translated so that the sample point is at the origin. Each edge of the polygon is then considered in turn. If the edge lies entirely above or below the x axis, then the edge cannot cross the ray, so no further calculation needs to be done. If, on the other hand, the edge does span the x axis, then it may cross the ray. To determine whether it does, the x intercept is computed. If the intercept lies to the right of the origin, the ray has crossed the edge and the number of crossings is incremented, otherwise a crossing has not occurred.

As with the inside test based on set operations, we would like this inside test to handle singularities correctly so that no gaps or double hits are produced. Unfortunately, we not only must deal with sample points on the polygon, but also with the cases arising when vertices lie on the horizontal ray. Some of these are illustrated in Figure 6.

The easiest way to handle these cases is to consider them not special at all. This can be done by using half open intervals again. The procedure `inside` considers all points that lie on the line $y = 0$ to be in the lower half plane and not on the ray. An edge spans the ray only if one endpoint lies in the upper half plane and the other endpoint lies in the lower half plane. Notice that this spanning test causes horizontal edges to be ignored. Notice also that vertices that are relative minima and maxima either yield 0 or 2 crossings.

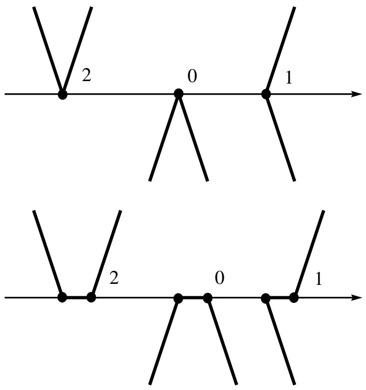
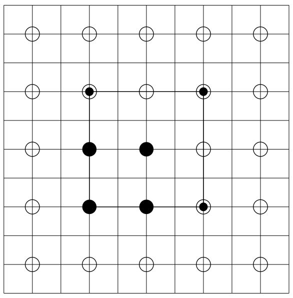


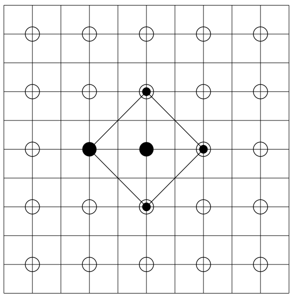
Figure 6: Singularities

This implies that passing through a relative minimum or maximum has no effect on the outcome of the test. Finally, the crossing number is computed correctly when a vertex is not a relative minimum or maximum. If it is not a minimum or maximum, one incident edge goes upward and another goes downward. The upper edge yields a crossing because the point on the x axis is in the lower half plane and the other point is in the upper half plane, whereas the lower edge does not generate a crossing because both endpoints are in the lower half plane. This yields a count of one crossing, signifying a transition from inside to outside or vice versa—the correct answer for this situation.

The other possible singularity occurs when the sample point is on an edge or a vertex. When this occurs the x intercept will be 0, since the edge must cross through the origin. Suppose these edges are simply ignored. This has the effect that points on shadow edges are considered outside because there no crossings. On the other hand, points on edges not in shadow are considered inside because ignoring the on case on the left side leaves an odd number of edge crossings to the right. Points on horizontal edges are also handled correctly for similar reasons. The fact that upper facing horizontal edges are considered in shadow (and thus not giving rise to crossings) arises from the fact that we choose to consider the x axis to be part of the lower half plane. In summary, the above inside test based on the Jordan Curve Theorem yields the same results as the shadow-based inside test based on set operations.

Consider the following two "special" cases:





The results of various inside tests are shown as circles. If the circle is filled, the point is inside the polygon, whereas if the circle is empty, the point is outside the polygon. Verify the classification produced by the above algorithm for each point for each shape. Do the classifications agree with what would be produced by the shadow-based inside test in the last section?

Finally, consider testing whether a point is inside a self-intersecting polygon. There are two ways of

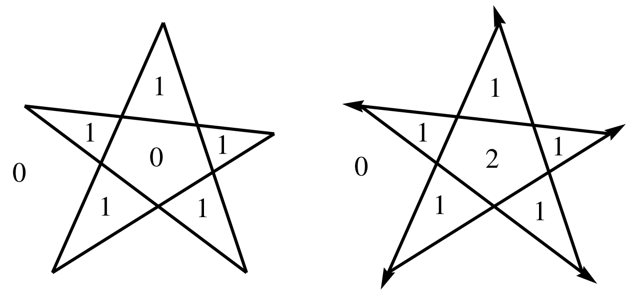


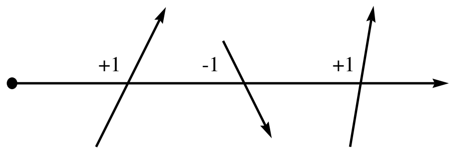
Figure 7: Star Shaped Polygon Broken into Regions

doing this: (1) the even-odd or parity rule and (2) the winding number rule. For non self-intersecting polygons these two criteria always yield the same answer.

- *Even-Odd Rule*: The point is outside if the ray intersects the polygon an even number of times; if it intersects an odd number of times, the point is outside.
- *Non-Zero Winding Number*: The point is inside if the winding number is non-zero; otherwise the point is outside.

Figure 7 illustrates the application of these rules to various regions inside a star-shaded polygon. On the left is the parity of each region and on the right is the crossing number within each region.

The procedure `inside` implements the even-odd rule. To apply the winding rule, we also take into consideration the direction in which a crossing intersects the x axis. If the edge crosses from below to above, we add one to the winding number. If the edge crosses from above to below, we substract one from the winding number.



The winding number corresponds to how many times the polygon encircles the sample point. We consider points inside if the winding number is non-zero although it is also possible to consider points with a winding number less than 0 to be inside.

3 Scanline or Sweepline Algorithms

The inside test based on the Jordan curve theorem can be significantly optimized if all the pixels inside a polygon are to be output. This is an example of the use of spatial coherence and relies on the fact that adjacent pixels have similar properties.

The basic idea is to sweep a horizontal scanline from the bottom of the polygon to the top. This takes advantage of spatial coherence in two ways:

1. During the vertical sweep we maintain a list of active edges spanning the current scanline. The active edge list makes the inside test more efficient by eliminating the need for redundant edge spanning tests in the inner loop. The active edge list can be maintained efficiently as the scan proceeds upward,

by noticing that its members change only when a vertex is passed. These updates can be managed efficiently by maintaining a priority queue of vertices sorted by their y value and inserting and deleting edges as appropriate when vertices are encountered in the sweep. The uniform nature of the scan also allows us to efficiently compute the x-intercepts and other properties associated with vertices that we wish to interpolate across the polygon. The new values are computed from the values on the previous scanline using finite differencing.

2. The pixels on a scanline can be classified very efficiently. Remember that the inside test is based on firing a horizontal ray to the right and counting the number of crossings. Adjacent pixels on a scanline will have the same crossing number unless an edge falls between them. If the active edge list is sorted based on the edges' x intercepts, the classification of inside vs. outside will flip every time an edge is enountered as we sweep from left to right. This corresponds to grouping all the edges into an ordered list of edge pairs. The inside pixels are those between the left and right edge of each pair. If values are to be interpolated along a scanline, this can be done efficiently using finite differences as we step from left to right.

3.1 Convex Polygons

In the case of convex polygons the active edge list will only contain a single left and right edge. This leads to a very efficient algorithm first developed by Frank Crow.

```
void
tile( vert v[], int n )
{
    int i, imin;

    imin = 0;
    for( i=1; i<n; i++ )
        if( v[i].y < v[imin].y )
            imin = i;
    scany( v, n, imin );
}
```

The first part of this procedure finds the vertex with the smallest y coordinate and then calls `scany` with the index of that vertex.

The procedure `scany` starts at the minimum y value and scans upward until reaching the maximum y value of the polygon. As the algorithm proceeds it replaces the left and/or right edge with a new edge whenever a vertex is passed. At each scanline, the set of pixels between the left and the right edge are output by the procedure `scanx`, and the positions of the left and right edges are updated.

```
void
scany( vert v[], int n, int i )
{
    int li, ri; /* l and r upper endpoint indices */
    int ly, ry; /* l and r upper endpoint y's */
    vert l, dl; /* l edge */
    vert r, dr; /* r edge */
    int rem;    /* number of vertices remaining */
    int y;      /* current scanline */

    li = ri = i;
    ly = ry = y = ceil(v[i].y);

    /* Assume counter-clockwise */
[1] for( rem = n; rem > 0; ) {
        /* find next l edge */
[2]     while( ly <= y && rem > 0 ) {
            rem--;
            i = li - 1; if( i < 0 ) i = n - 1; /* step cw */
            ly = ceil( v[i].y );
            if( ly > y ) /* replace l edge */
                differencey( &v[li], &v[i], &l, &dl, y );
            li = i;
        }
        /* find next r edge */
        while( ry <= y && rem > 0 )  {
            rem--;
            i = ri + 1; if( i >= n ) i = 0;    /* step ccw */
            ry = ceil( v[i].y );
            if( ry > y ) /* Replace r edge */
                differencey( &v[ri], &v[i], &r, &dr, y );
            ri = i;
        }
        /* while l & r span y */
[3]     for( ; y < ly && y < ry; y++ ) {
            /* scan and interpolate edges */
            scanx( &l, &r, y );
            increment( &l, &dl );
            increment( &r, &dr );
        }
    }
}
```

The outer loop (1) sweeps until there are no more edges remaining. Inside this loop is the main loop

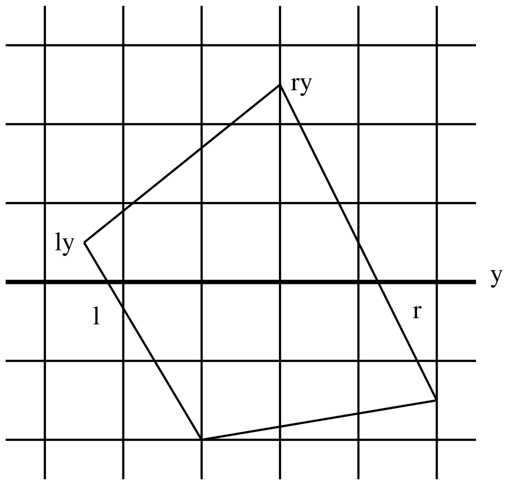
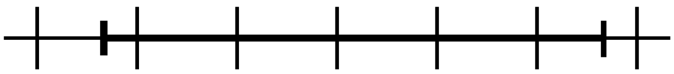


Figure 8: Crow's Algorithm for Convex Polygons

(3) that sweeps upward, outputting spans and incrementing edge properties, until we encounter the upper endpoint of the left or right edge. When this occurs we need to replace that edge (or both edges) with a new edge. We do this by looping until we find the next edge whose upper endpoint lies above the current scanline (2). Since, when this loop was entered, the previous endpoint was on or below the current scanline, this guarantees that the opposite endpoint lies in the lower halfplane, and thus the edge spans the current scanline. It is important to note that the tests for whether an edge spans a scanline are equivalent to those used in the procedure `inside` discussed previously. We may need to loop multiple times because horizontal edges, and edges whose vertical distance between endpoints is less than one pixel, may not span the current scanline. Notice that this replacement strategy will produce correct results as long as the chain of edges on the left and right sides of the polygon are monotonic— that is, the y coordinate of the vertices must never decrease as we advance to the next edge—and does not require the polygon to be convex. The counter-clockwise assumption means as we increase the vertex array index we trace out the polygon in counter-clockwise order. Starting at a relative minimum, we trace the left boundary by moving clockwise around the polygon decrementing the array index, and we trace the right boundary by moving counter-clockwise by incrementing the index. The edge replacement loop may also terminate if there are no more edges to add.

This code is very compact, but its boundary conditions are a bit tricky. First note how the main loop is entered. Initially there are no active edges so we set the y location of the endpoints of the left and right edges to force them to be added the first time through the loop. Second note how the loop may be exited. If the top edge of the polygon is horizontal, then we attempt to add it as a left edge, but fail because it does not span the scanline. In this case the while loop (3) is never entered because neight the left nor the right edge spans the current scanline and then the outer for loop also terminates.

Once the left and right edges are determined for the current scanline, the next step is to find the pixels inside or between these two edges. As before, we need to deal with singularities, but this is easily done using half-open intervals.



Note that this interval can be empty.

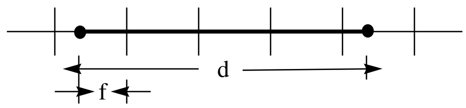
```
void
scanx( vert *l, vert *r, int y )
{
    int x;
    int lx, rx;
    vert s, ds;

    lx = ceil( l->x );
    rx = ceil( r->x );
    if( lx < rx ) {
        differencex( l, r, &s, &ds, lx );
        for( x = lx; x < rx; x++ ) {
            plot( x, y );
            increment(&s, &ds);
        }
    }
}
```

Within `scany`, whenever an edge is replaced we compute the x intercept at the current scanline and the change in x caused by stepping up to the next scanline. This process also can be used to interpolate other values such as color and alpha across the polygon. This will be discussed in the next section. The same process is performed within `scanx` to compute the new values at the left pixel center and the change in the values as we step right. The key to these functions is to carefully keep track of values at integer locations and compensate for fractional offsets.

```
void
difference( vert *v1, vert *v2, vert *e, vert *de,
    value d, value f )
{
    de->x = (v2->x - v1->x) / d;
     e->x = v1->x + f * de->x;
}
```

The procedure `difference` takes as input the two endpoints `v1` and `v2`, and then computes the current value and delta value and stores them in `e` and `de`. The procedure uses the total amount of change in the independent coordinate `d`, and the fractional offset `f` between the first real and integer value.

Here is the geometry for the horizontal case:

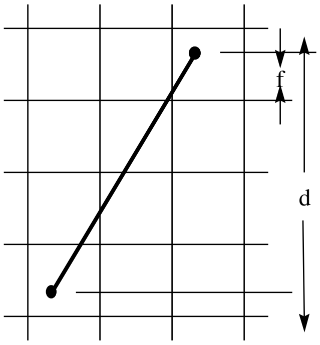


and the procedure that implements it.

```
void
differencex( vert *v1, vert *v2, vert *e, vert *de, int x )
{
    difference( v1, v2, e, de, v2->x - v1->x, x - v1->x );
}
```

In this case the independent coordinate is x.

Here is the geometry for the vertical case:



and the procedure that implements it.

```
void
differencey( vert *v1, vert *v2, vert *e, vert *de, int y )
{
    difference( v1, v2, e, de, v2->y - v1->y, y - v1->y );
}
```

In this case the independent coordinate is y.

Once the setup is done, the incremental computation is very simple.

```
void
increment( vert *e, vert *de )
{
    e->x += de->x;
}
```

4 Interpolation

The above scan conversion algorithms output all the pixels inside the polygon. These pixels are set to the color of the polygon, and then combined using the current imaging model with the pixels in the framebuffer. If the polygon has a constant color, then all pixels output have the same color. An alternative is to provide colors at vertices and to interpolate these colors across the polygon. A topological mesh of polygons can be smoothly shaded using this technique.

One method for achieving this effect is to linearly interpolate colors along an edge and then linearly interpolate the resulting edge colors across each scanline span. This is very easy to do by providing enhanced versions of `difference` and `increment`.

```
void
difference( vert *v1, vert *v2, vert *e, vert *de,
    value d, value f )
{
    d = 1. / d;

    de->x = (v2->x - v1->x) * d;
     e->x = v1->x + f * de->x;

    de->r = (v2->r - v1->r) * d;
     e->r = v1->r + f * de->r;
    de->g = (v2->g - v1->g) * d;
     e->g = v1->g + f * de->g;
    de->b = (v2->b - v1->b) * d;
     e->b = v1->b + f * de->b;
    de->a = (v2->a - v1->a) * d;
     e->a = v1->a + f * de->a;
}


void
increment( vert *v, vert *dv )
{
    v->x += dv->x;

    v->r += dv->r;
    v->g += dv->g;
    v->b += dv->b;
    v->a += dv->a;
}
```

Of course, there is no limit to what can be interpolated across the polygon using this technique. For example, in 3D graphics the depth or z coordinate, the surface normal, and the texture coordinates are often interpolated across a polygon.

This method of interpolation, although very common, has its problems. Since each value is interpolated separately, we can analyze the properties of the interpolation by examining the interpolation of a single value. Assume that the polygon lies in the $x - y$ plane and interpret the value to be interpolated as a height. Then each vertex maps to a 3D point above the polygon, and the interpolated values define a surface inside the polygon. The two procedures above generate a peculiar ruled surface. The heights of all points along the polygon's edges are formed by linearly interpolating between two vertices, and thus, along an edge the height of the surface lies on the line connecting the two 3D points. The interior surface is formed by interpolating between edges. Since we are doing linear interpolation across a scanline, the intersection of the surface with a vertical plane through the scanline is also a line connecting the left edge to the right edge. As we sweep the scanline up and down the positions of the left and right endpoints change and we create a ruled surface above the polygon.

There are two problems with this method of interpolation. First, the surface that is generated depends on the orientation of the scanlines with respect to the polygon. If the polygon is rotated, the directions of the scanlines change, and hence, the shape of the ruled surface also changes. This is undesirable because we expect the color texture to be fixed to the polygon. This is true along the edges, but not true on the inside, so the visual effect is that the polygon's texture is deformed like a rubber sheet. Second, the surface may have sharp discontinuities if the polygon is concave. Consider what happens when a vertex at a relative minimum is encountered. At the scanline below that vertex the surface lies on a line connecting two edges. If the vertex lies between the left and right edges of that span, then on the next scanline the surface will go from the original left edge to the vertex and then to the original right edge. Thus, the addition of the vertex causes a tear in the surface.

Fortunately, this method of interpolation does not have these undesirable properties for triangles. Since three points determine a unique plane, the ruled surface above a triangle is unique and independent of the orientation. Also, triangles are always convex, so there can be no discontinuities in the interpolated surface.