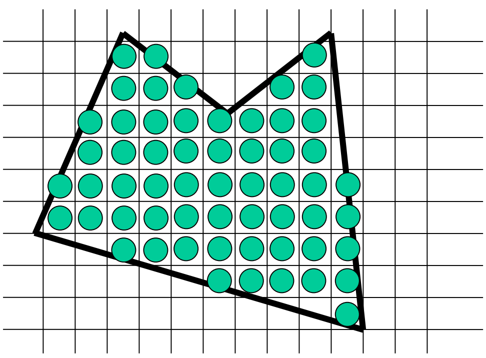
Lecture 6

Rasterizing polygons

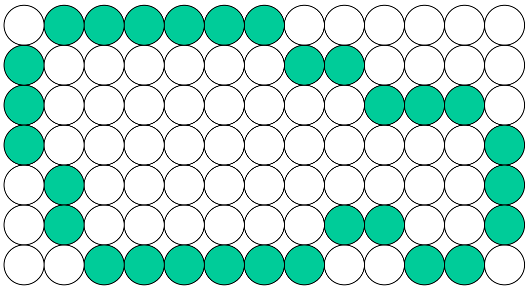
Rasterizing Polygons

Given a set of vertices and edges, find the pixels that fill the polygon.



Flood Fill

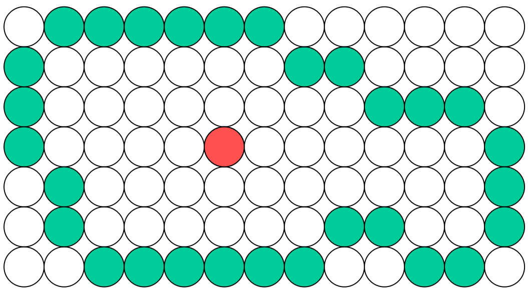
First, rasterizing its edges into the frame buffer using Bresenham’ s algorithm



How to fill polygons whose edges are already drawn?

Flood Fill

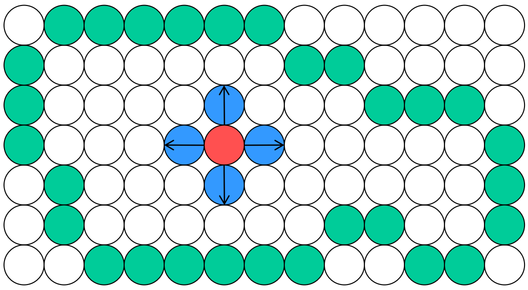
First, rasterizing its edges into the frame buffer using Bresenham’ s algorithm



Choose a point inside, and fill outwards

Flood Fill

First rasterizing its edges into the frame buffer using Bresenham’ s algorithm



Choose a point inside, and fill outwards

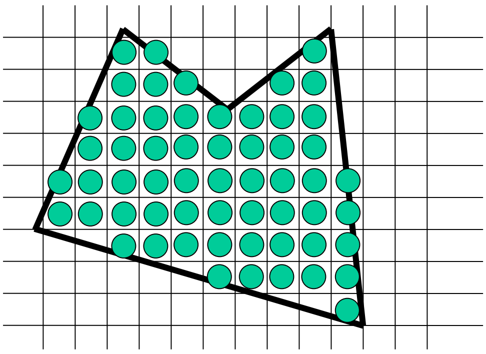
Flood Fill

Fill a point and recurse to all of its neighbors

```
floodFill(int x, int y, color c)
  if(stop(x,y,c))
    return;

  setPixel(x,y,c);
  floodFill(x-1,y,c);
  floodFill(x+1,y,c);
  floodFill(x,y-1,c);
  floodFill(x,y+1,c);

int stop(int x, int y, color c)
  return colorBuffer[x][y] == c;
```

Rasterizing Polygons

Given a set of vertices and edges,
find the pixels that fill the polygon.



Rasterizing Polygons

vList is an ordered list of the polygon's vertices

```
fillPoly(vertex vList[ ])
  boundingBox b = getBounds(vList);
  int xmin = b.minX;
  int xmax = b.maxX;
  int ymin = b.minY;
  int ymax = b.maxY;

  for(int y = ymin; y <= ymax; y++)
    for(int x = xmin; x <= xmax; x++)
      if(insidePoly(x,y,vList))
        setPixel(x,y);
```

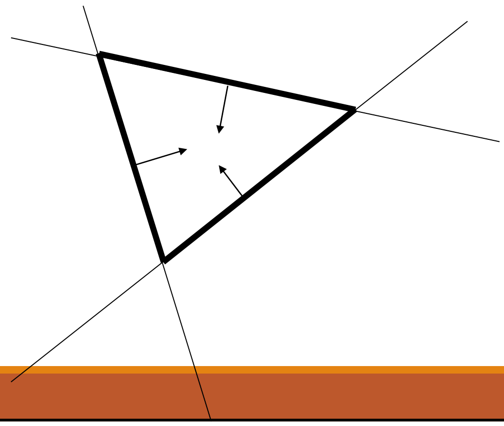
What does 'inside' mean?

How to test if a point is inside a polygon

1. Half-space tests
2. Jordan Curve Theorem (even/odd or +1/-1)
3. Winding number test

Half Space Tests

Given the edges of a triangle, the inside is
the intersection of half-spaces defined by the edges

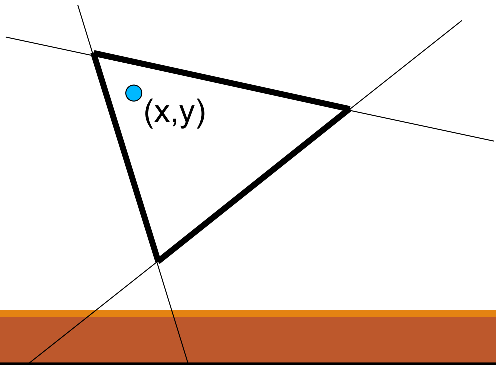


Half Space Tests

Easily computable:

$$l(x,y) = ax + by + c < 0$$

Iff (x,y) is inside



Half Space Tests

lineEq computes the implicit line value for 2 vertices & a point

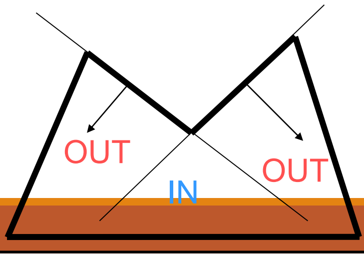
```
fillTriangle(vertex vList[3])
  //-- get the bounding box as before --//
  float e1 = lineEq(vList[0],vList[1],xmin,ymin);
  float e2 = lineEq(vList[1],vList[2],xmin,ymin);
  float e3 = lineEq(vList[2],vList[0],xmin,ymin);
  int xDim = xMax - xMin;

  for(int y = ymin; y <= ymax; y++)
    for(int x = xmin; x <= xmax; x++)
      if(e1<0 && e2<0 && e3<0)
        setPixel(x,y);
    e1 += a1; e2 += a2; e3+= a3;
  e1 += -xDim*a1+b1; e2 = -xDim*a2+b2; e3 = -xDim*a3+b3
```

Half Space Tests

Easily computable:

$$l(x,y) = ax + by + c < 0$$ Iff (x,y) is inside

Doesn't work on concave objects!!
→ triangulate



What does 'inside' mean?

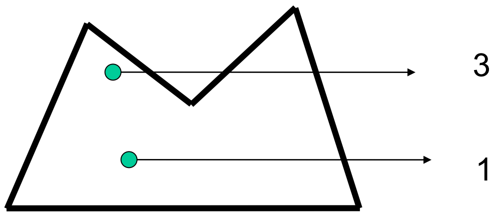
How to test if a point is inside a polygon

1. Half-space tests
2. Jordan Curve Theorem (even/odd or +1/-1)
   - Self-intersecting polygon OK
3. Winding number test

Jordan Curve Theorem

Even/odd approach

Hit test: inside or outside based on the number of intersected edges is even or odd

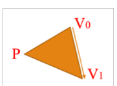


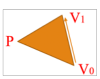
Any ray from a point inside a polygon will intersect the polygon's edges an odd number of times

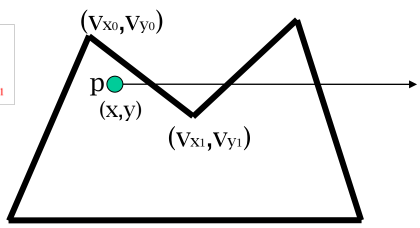
Jordan Curve Theorem

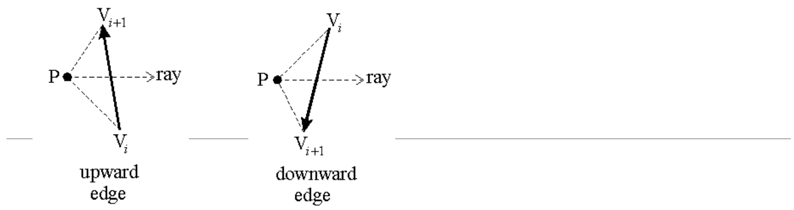
vList is an ordered list of the n polygon vertices

```
int jordanInside(vertex vList[ ], int n, float x, float y)
   int cross = 0;
   float x0, y0, x1, y1;

   x0 = vList[n-1].x – x;     y0 = vList[n-1].y – y;
   for(int i = 0; i < n; i++)
      x1 = vList[i].x – x;      y1 = vList[i].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;

   return cross & 1;
```





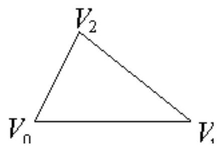




$$2A(\Delta) = \begin{vmatrix} (x_1 - x_0) & (x_2 - x_0) \\ (y_1 - y_0) & (y_2 - y_0) \end{vmatrix} = \begin{vmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix}$$

$$= (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

where $V_i = (x_i, y_i)$

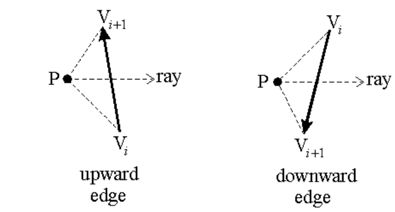


i.e., x0y1-x1y0

Vi=(xi,yi)

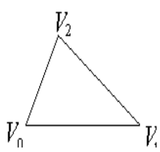
The signed area will be
-positive if the triangle *is* oriented counterclockwise
-negative if the triangle is oriented clockwise

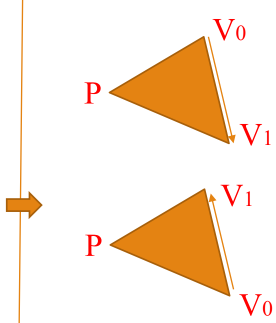
http://geomalgorithms.com/a03-_inclusion.html
http://geomalgorithms.com/a01-_area.html



$$2A(\Delta) = \begin{vmatrix} (x_1 - x_0) & (x_2 - x_0) \\ (y_1 - y_0) & (y_2 - y_0) \end{vmatrix} = \begin{vmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix}$$

$$= (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$





The signed area will be
-Positive, if the triangle *is* oriented counterclockwise
  - x0y1>x1y0
-negative, if the triangle is oriented clockwise
  - x1y0 >x0y1

2A = x0y1-x1y0

where
x0 = v0.x – p.x;
y0 = v0.y – p.y;

$2Area = x0y1 - x1y0$

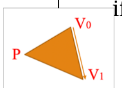
A> 0, if the triangle *is* oriented counterclockwise

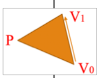
A< 0, if the triangle is oriented clockwise

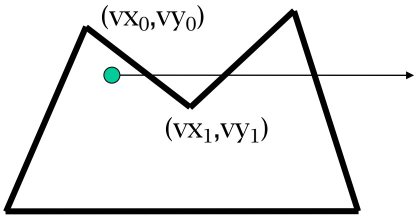
Jordan Curve Theorem

vList is an ordered list of the n polygon vertices

```
int jordanInside(vertex vList[ ], int n, float x, float y)
  int cross = 0;
  float x0, y0, x1, y1;

  x0 = vList[n-1].x - x;     y0 = vList[n-1].y - y;
  for(int i = 0; i < n; i++)
    x1 = vList[i].x - x;      y1 = vList[i].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;

  return cross & 1;
```





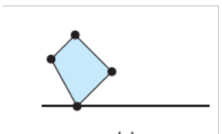


Jordan Curve Theorem

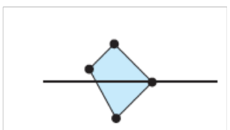
What if it goes through a vertex?

Treat these two cases differently:

- case (a): the vertex–scanline intersection is counted as either zero or two edge crossings
- case (b): the vertex–scanline intersection must be counted as one edge crossing.



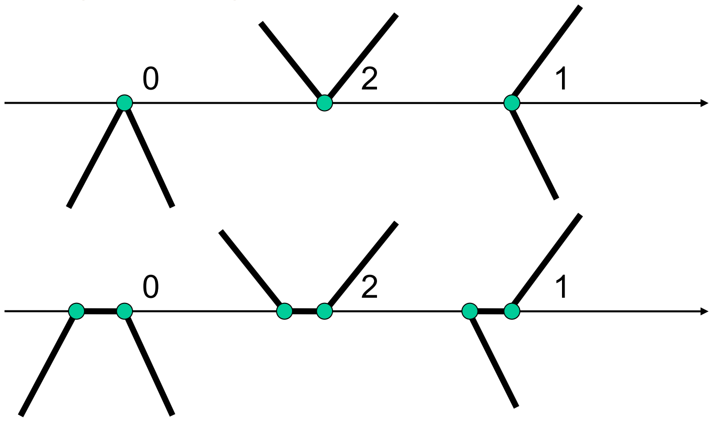
(a)



(b)

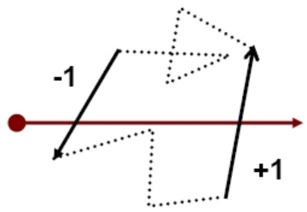
Jordan Curve Theorem

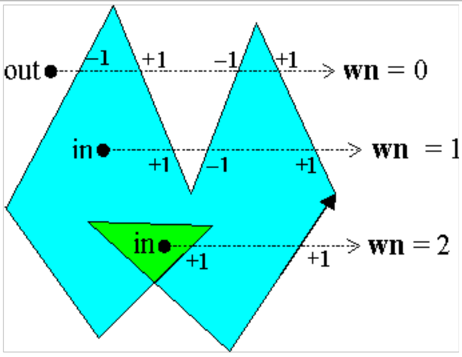
What if it goes through a vertex?

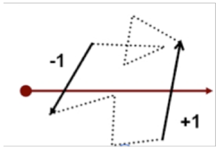


Jordan Curve Theorem

- Non-zero winding rule:
  - Draw a line from the test point to the outside
  - Count +1 if you cross an edge in an anti-clockwise sense
  - Count -1 if you cross an edge in a clockwise sense







count +1, if you cross an edge in an anti-clockwise sense
count -1, if you cross an edge in an clockwise sense
Inside point : non-zero

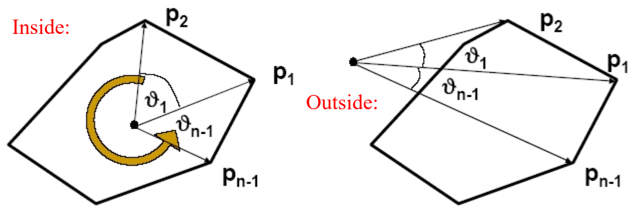
What does 'inside' mean?

How to test if a point is inside a polygon

1. Half-space tests
2. Jordan Curve Theorem (even/odd or +1/-1)
3. Winding number test

Winding Number Test-Method I

- Sum the angle subtended by the vertices

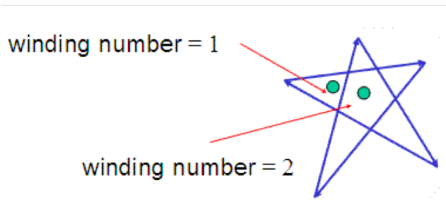


$$\sum_{i=1}^{n} \vartheta_i = 2\Pi \qquad \sum_{i=1}^{n} \vartheta_i \neq 2\Pi$$

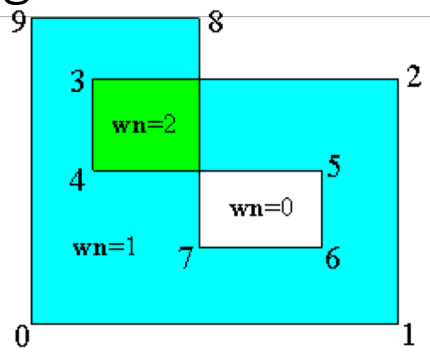
Winding Number Test – Method II

The number of times it is encircled by the edges of the polygon

- +1: if clockwise encirclements
- -1 : if counterclockwise encirclements
- a point is inside the polygon if its winding number is not zero

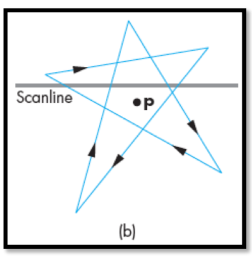


Winding Number Test II



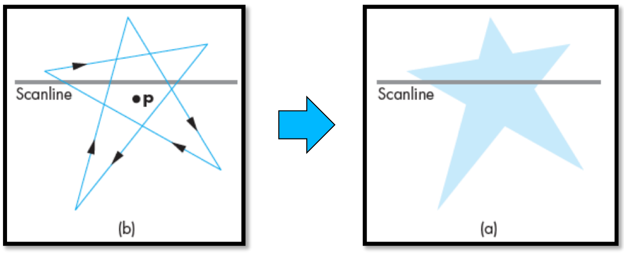
+1: if clockwise encirclements
-1 : if counterclockwise encirclements
a point is inside the polygon if its winding number is not zero

Winding Number Test – Method II



Filling with the Winding Number Test？

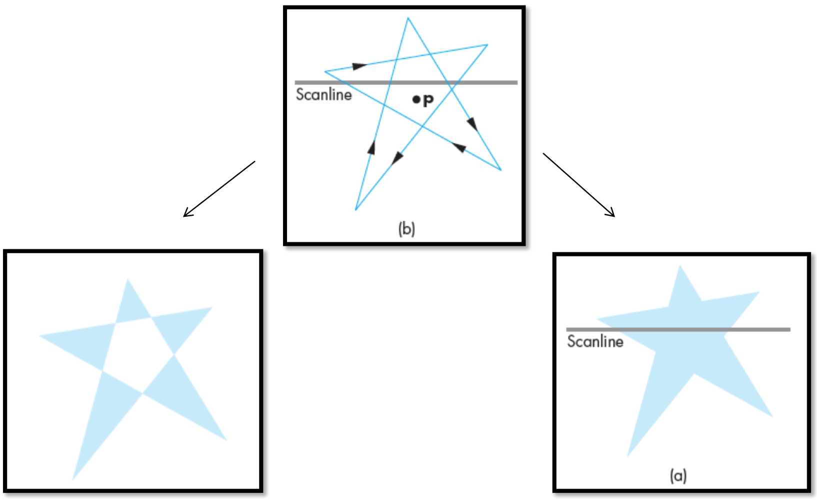
Winding Number Test



Filling with Winding Number Test II

Filling with the odd–even test？

Winding Number Test



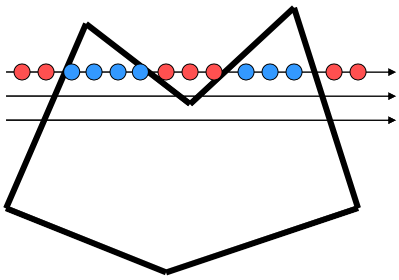
Filling with the odd–even test

Filling with Winding Number Test

rasterization algorithms

Scan Line Algorithms

Take advantage of coherence in "insided-ness"

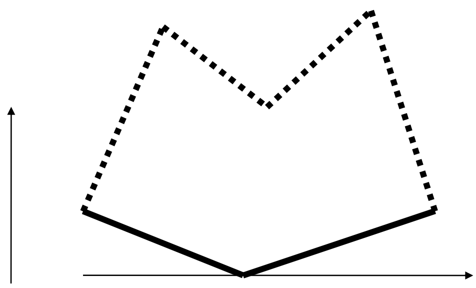


Inside/outside can only change at edge events

Current edges can only change at vertex events

Scan Line Algorithms

Create a list of the edges intersecting the first scanline



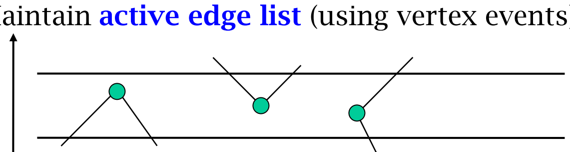
Sort this list by the edge's x value on the first scanline

Call this the active edge list

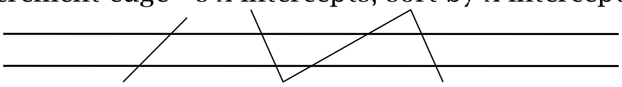
Scan Line Algorithms

For each scanline:

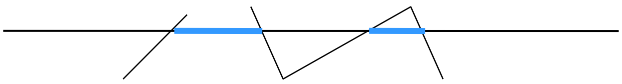
1. Maintain **active edge list** (using vertex events)



2. Increment edge's x-intercepts, sort by x-intercepts

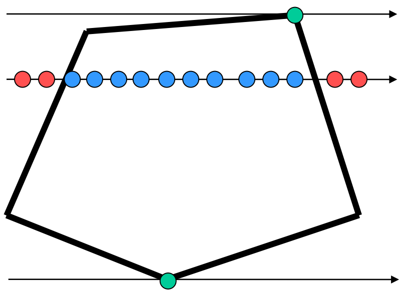


3. Output spans between left and right edges



Convex Polygons

Convex polygons only have 1 span

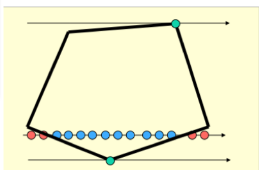


Insertion and deletion events happen only once

Crow's Algorithm

Step1: Find the vertex with the smallest y value to start

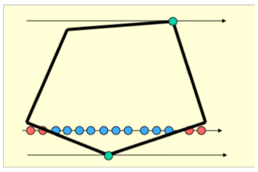
```
crow ( vertex vList[], int n)
        int iMin = 0;
        for(int i = 1; i < n; i++)
            if(vList[i].y < vList[iMin].y)
            iMin = i;

        scanY(vList,n,iMin);
```



Crow's Algorithm

Step2: Scan upward maintaining the **active edge list**

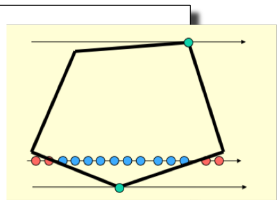
```
scanY(vertex vList[], int n, int i)
  int li, ri;    // left & right upper endpoint indices
  int ly, ry;    // left & right upper endpoint y values
  vertex l, dl;     // current left edge and delta
  vertex r, dr;     // current right edge and delta
  int rem;    // number of remaining vertices
  int y;        // current scanline

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

(1) for( rem = n; rem > 0)
 (2) // find appropriate left edge
     // find appropriate right edge
 (3) // while l & r span y (the current scanline)
       // draw the span
```



Crow's Algorithm

Find the appropriate next left edge

```
(2) while( ly < = y && rem > 0)
      rem--;
      i = li - 1;
      if(i < 0)
            i = n-1;  // go clockwise

      ly = ceil( v[i].y );
      if( ly > y ) // replace left edge
         differenceY( &vList[li], &vList[i], &l, &dl, y);
      li = i;  // index of the left endpoint
```



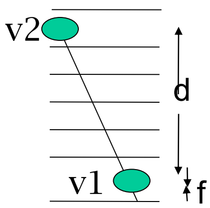
all vertices are in
counter-clockwise order
index: 0~(n-1)

Crow’s Algorithm

Calculate delta and starting values

```
differenceY(vertex *v1, vertex *v2, vertex *e, vertex *de, int y)
   difference(v1, v2, e, de, (v2.y – v1.y), y – v1.y);
```

```
difference(vertex *v1, vertex *v2, vertex *e, vertex *de, float d, float f)
   de.x = (v2.x – v1.x) / d;
   e.x   = v1.x + f * de.x;
```



Crow’s Algorithm

Draw the spans

```
(3)for( ; y < ly && y < ry;  y++)
      // scan and interpolate edges
      scanX(&l, &r, y);
      increment(&l,&dl);
      increment(&r,&dr);
```

Increment the x value

```
   increment(vertex *edge, vertex *delta)
      edge.x += delta.x;
```

Crow's Algorithm

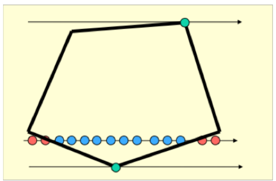
Draw the spans

```
scanX(vertex *l, vertex *r, int y)
  int x, lx, rx;
  vertex s, ds;

  lx = ceil(l.x);
  rx = ceil(r.x);
  if(lx < rx)
    for(x = lx, x < rx; x++)
      setPixel(x,y);
```

Crow's Algorithm

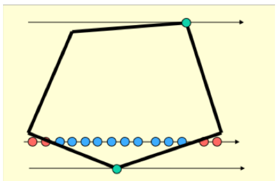
Step2: Scan upward maintaining the **active edge list**

```
scanY(vertex vList[], int n, int i)
  int li, ri;    // left & right upper endpoint indices
  int ly, ry;    // left & right upper endpoint y values
  vertex l, dl;     // current left edge and delta
  vertex r, dr;     // current right edge and delta
  int rem;    // number of remaining vertices
  int y;       // current scanline

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

(1) for( rem = n; rem > 0)
  (2) // find appropriate left edge
      // find appropriate right edge
  (3) // while l & r span y (the current scanline)
        // draw the span
```



Crow's Algorithm

Step2: Scan upward maintaining the **active edge list**



```
scanY(vertex vList[], int n, int i)
   //…
   li = ri = i;
   ly = ry = y = ceil(vList[i].y);

(1) for( rem = n; rem > 0)
      // find appropriate left edge
 (2)  while( ly < = y && rem > 0)
         //…
      // find appropriate right edge
      while( ry < = y && rem > 0)
         //…
 (3)
     for( ; y < ly && y < ry;  y++) // while l & r span y (the current scanline)
      scanX(&l, &r, y);      // draw the span
      increment(&l,&dl);
      increment(&r,&dr);
```

Crow's Algorithm

Draw the spans

```
scanX(vertex *l, vertex *r, int y)
   int x, lx, rx;
   vertex 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++)
         setPixel(x,y);
         increment(&s,&ds);
```

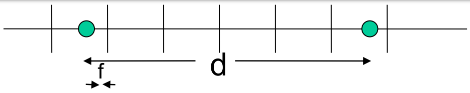
Interpolating other values
E.g, colors

Crow's Algorithm

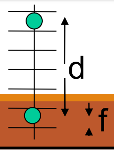
Calculate delta and starting values

```
differenceX(vertex *v1, vertex *v2, vertex *e, vertex *de, int x)
   difference(v1, v2, e, de, (v2.x - v1.x), x - v1.x);
```

```
difference(vertex *v1, vertex *v2, vertex *e, vertex *de, float d, float f)
   de.x = (v2.x - v1.x) / d;
   e.x   = v1.x + f * de.x;
```




```
differenceY(vertex *v1, vertex *v2, vertex *e, vertex *de, int y)
   difference(v1, v2, e, de, (v2.y - v1.y), y - v1.y);
```



Crow's Algorithm

Draw the spans

```
scanX(vertex *l, vertex *r, int y)
   int x, lx, rx;
   vertex 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++)
         setPixel(x,y);
         increment(&s,&ds);
```

```
increment(vertex *edge, vertex *delta)
   edge.x += delta.x;
```

```
differenceX(vertex *v1, vertex *v2, vertex *e, vertex *de, int x)
   difference(v1, v2, e, de, (v2.x - v1.x), x - v1.x);
```

```
difference(vertex *v1, vertex *v2, vertex *e, vertex *de, float d, float f)
   de->x = (v2.x - v1.x) / d;
   e->x   = v1.x + f * de.x;
```

Crow's Algorithm

Interpolating other values

```
difference(vertex *v1, vertex *v2, veretx *e, vertex *de, float d, float f)
   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;
```

```
increment( vertex *v, vertex *dv)
   v.x += dv.x;
   v.r += dv.r;
   v.g += dv.g;
   v.b += dv.b;
```