# Computer Graphics

## - Rasterization -

**Philipp Slusallek**

# Overview

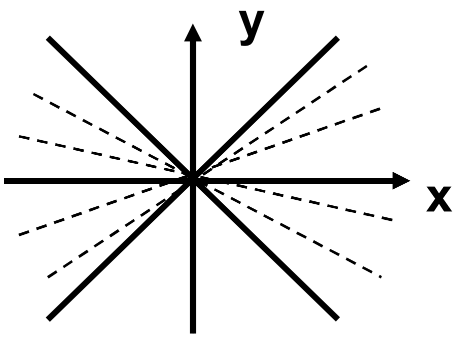
- **So far:**
  - OpenGL
  - Programmable Graphics Hardware
- **Today:**
  - Line rasterization
  - Surface rasterization (scan conversion)

# Rasterization

- **Definition**
  - Given a primitive (usually 2D lines, circles, polygons), specify which pixels on a raster display are covered by this primitive
  - Extension: specify what part of a pixel is covered
    → filtering & anti-aliasing
- **OpenGL lecture**
  - From an application programmer‘s point of view
- **This lecture**
  - From a graphics package implementer‘s point of view
- **Usages of rasterization in practice**
  - 2D-raster graphics
    - e.g. Postscript
  - 3D-raster graphics
  - 3D volume modeling and rendering
  - Volume operations (CSG operations, collision detection)
  - Space subdivision
    - Construction and traversing

# Rasterization

- **Assumption**
  - Pixels are sample ***points*** on a 2D-integer-grid
    - OpenGL: integer-coordinate bottom left; X11, Foley: in the center
  - Simple raster operations
    - Just setting pixel values
      - Antialiasing later
  - Endpoints at pixel coordinates
    - simple generalization with fixed point
  - Limiting to lines with gradient $|m| \leq 1$
    - Separate handling of horizontal and vertical lines
    - Otherwise exchange of x & y: $|1/m| \leq 1$
  - Line size is one pixel
    - $|m| \leq 1$: 1 pixel per column (X-driving axis)
    - $|m| > 1$: 1 pixel per row (Y-driving axis)



# Lines: As Functions

- **Specification**
  - Initial and end points: $(x_0, y_0), (x_e, y_e)$
  - Functional form: $y = mx + B$ with $m = dy/dx$
- **Goal**
  - Find pixels whose distance to the line is smallest
- **Brute-Force-Algorithm**
  - It is assumed that +X is the driving axis

```
for xi = x0 to xe
   yi = m * xi + B
   setpixel(xi, Round(yi))    // Round(yi)=Floor(yi+0.5)
```

- **Comments**
  - Variables m and $y_i$ must be calculated in floating-point
  - Expensive operations per pixel (e.g. in HW)

# Lines: DDA

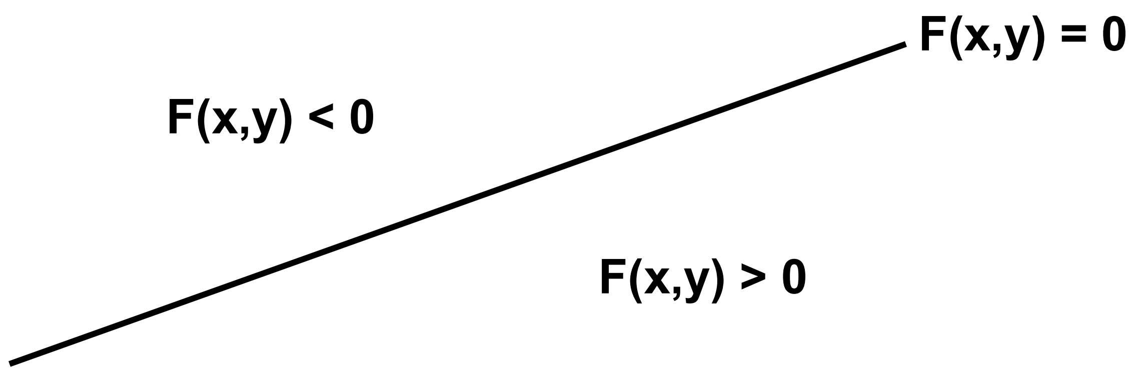
- **DDA: Digital Differential Analyzer**
  - Origin of solvers for simple incremental differential equations (the Euler method)
    - Per step in time: `x´ = x + dx/dt, y´ = y + dy/dt`
- **Incremental algorithm**
  - Per pixel
    - $\mathbf{x_{i+1} = x_i + 1}$
    - $\mathbf{y_{i+1} = m\ (x_i + 1) + B = y_i + m}$
    - $\mathbf{setpixel(x_{i+1},\ Round(y_{i+1}))}$
- **Remark**
  - Utilization of line coherence trough incremental calculation
    - Avoid the costly multiplication
  - Accumulates error over the length of the line
  - Floating point calculations may be moved to fixed point
    - Must control accuracy of fixed point representation

# Lines: Bresenham (´63)

- **DDA analysis**
  - Critical point: decision by rounding up or down
  - Integer-based decision through implicit functions
- **Implicit version**

$$F(x,y) = dy\,x - dx\,y + dx\,B = 0$$

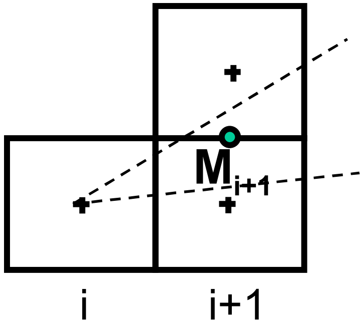
$$F(x,y) = ax + by + c = 0 \quad \text{where } a = dy, b = -dx, c = Bdx$$



# Lines: Bresenham

- **Decision variable (the midpoint formulation)**
  – Measures the vertical distance of midpoint from line:

    $d_{i+1} = F(M_{i+1}) = F(x_i+1, y_i+1/2) = a(x_i+1) + b(y_i+1/2) + c$



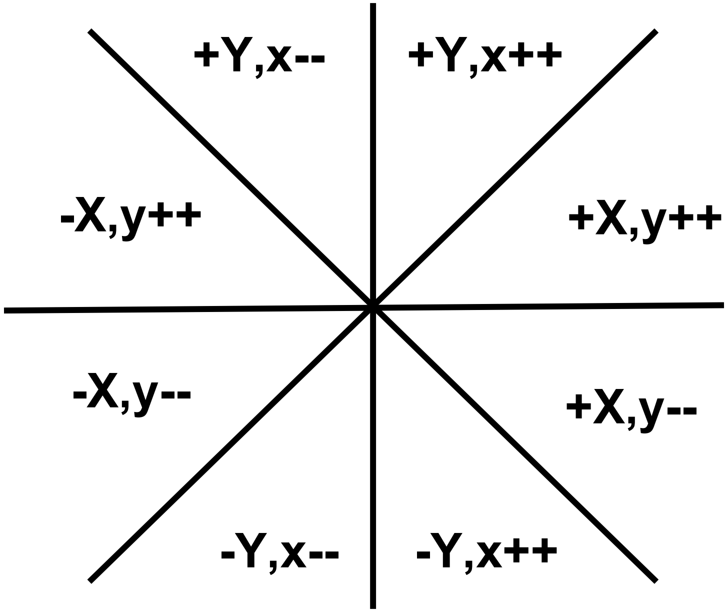

- **Preparations for the next pixel**
  – if ($d_i \leq 0$)
    - $d_{i+1} = d_i + a = d_i + dy$ // incremental calculation
  – else
    - $d_{i+1} = d_i + a + b = d_i + dy – dx$
    - y= y + 1
  – x = x +1

# Lines: Integer Bresenham

- **Initialization**
  - $d_{start} = F(x_0+1, y_0+1/2) = a(x_0+1) + b(y_0+1/2) + c =$
    $ax_0 + by_0 + c + a + b/2 = F(x_0, y_0) + a + b/2 = a + b/2$
  - Because $F(x_0, y_0)$ is zero by definition (line goes through end point)
    - Pixel is always set
- **Elimination of fractions**
  - Any positive scale factor maintains the sign of F(x,y)
  - $F(x_0, y_0) = 2(ax_0 + by_0 + c) \rightarrow d_{start} = 2a + b$
- **Observation:**
  - When the start and end points have integer coordinates then b= dx and a= -dy have also integer values
  - Floating point computation can be eliminated

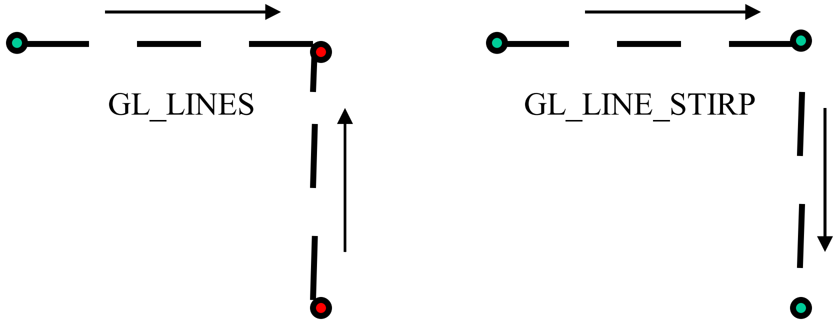
# Lines: Arbitrary Directions

- **8 different cases**
  - Driving (active) axis: ±X or ±Y
  - Increment/decrement of y or x, respectively

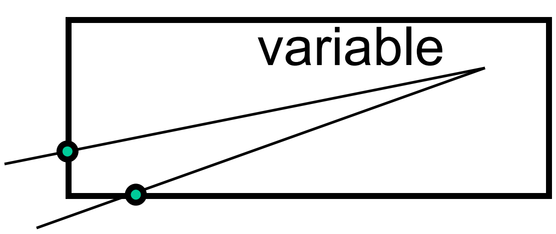


# Lines: Some Remarks

- **Reversed endpoints order – consistency of pixel choices**
  - m > 0: (d ≤ 0)?
  - m < 0: (d ≥ 0)?

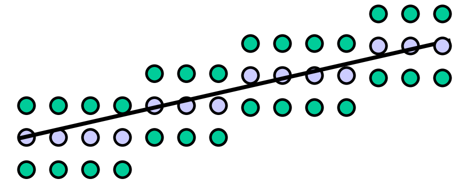



- **Dashed lines**
  - glLineStipple(Factor, 16-BitSample)
  - if (BitSample[(n++/Factor)%16]) then setpixel(...)
  - Consistent continuation of dashing for line strips and loops
- **Weaker intensity of diagonal lines**
  - Same number of pixel on a larger distance (up to 41%)
- **Sub-pixel precision**
  - Clipping requires sub-pixel coordinates
  - Correct initialization of the decision



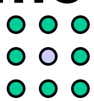
# Thick Lines

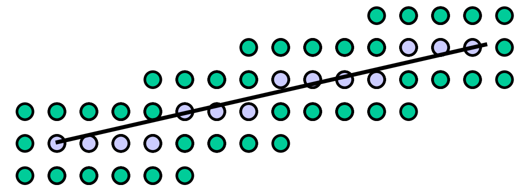
- **Pixel replication**



  - Problems with even-numbered widths,
  - Varying intensity of a line as a function of slope

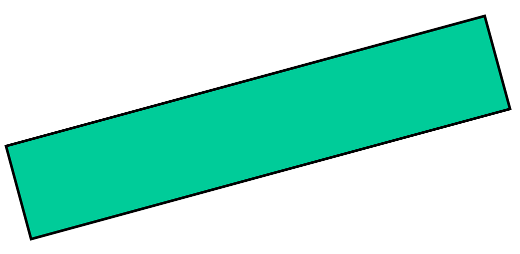
- **The moving pen**





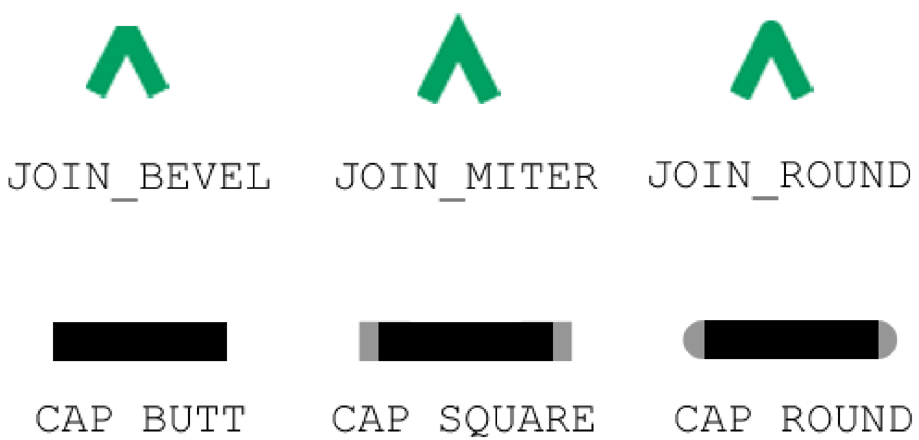
  - For some pen footprints the thickness of a line might change as a function of its slope
  - Should be as „round“ as possible

- **Filling areas between boundaries**



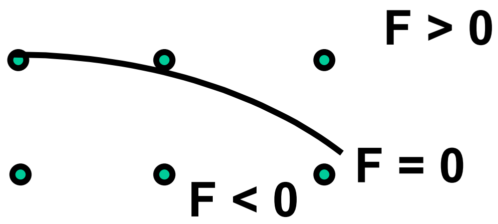
# Handling Start And End Points

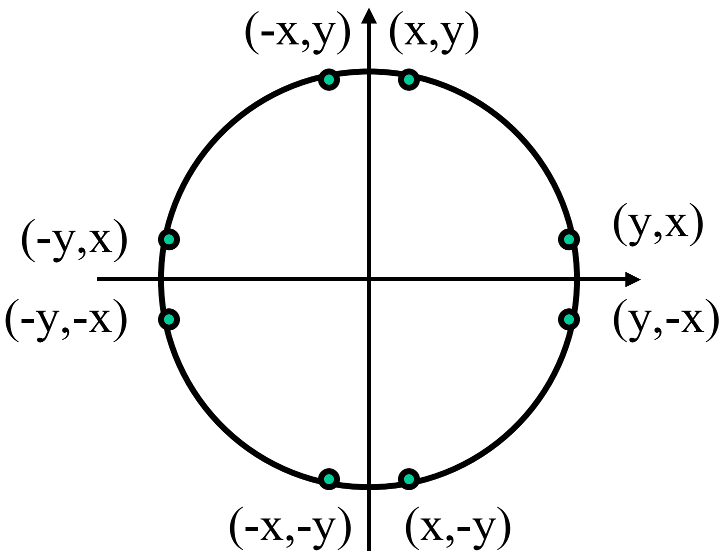
- **End points handling**
  - Capping: Handling of end point
    - Butt: End line orthogonally at end point
    - Round: End line with radius of half the line width
    - Square: End line with oriented square
  - Joining: Handling of joints between lines
    - Bevel: Connect outer edges by straight line
    - Round: Join with radius of half the line width
    - Miter: Join by extending outer edges to intersection



# Bresenham: Circle

- **Eight different cases, here +X, y--**
  - Initialization: x=0, y= R
  - $F(x,y)=x^2+y^2-R^2$
  - d=F(x+1, y-1/2)
  - d < 0:
    - d=F(x+2,y-1/2)
  - d > 0:
    - d=F(x+2,y-3/2)
    - y=y-1
  - x=x+1



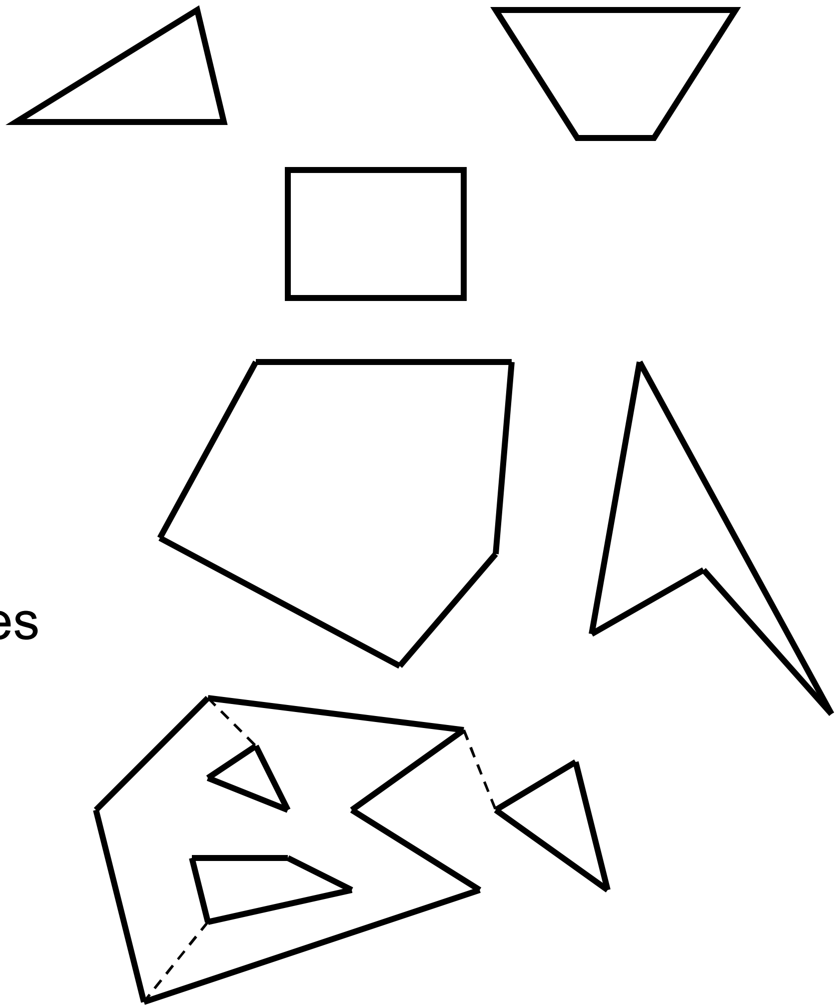

- **Eight-way symmetry: only one 45° segment is needed to determine all pixels in a full circle**

# Bresenham: More General

- **Midpoint method works well for ellipses and other implicitly defined objects**
  - Parabolas, hyperbolas, ...

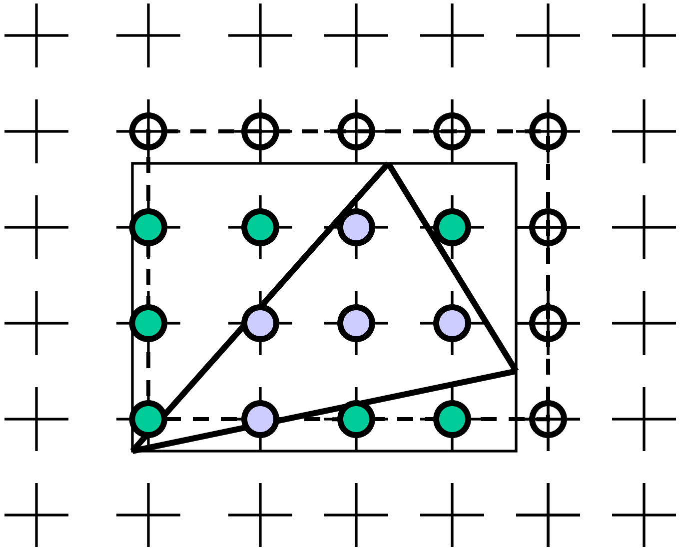
# Reminder: Polygons

- **Types**
  - Triangles
  - Trapezoids
  - Rectangles
  - Convex polygons
  - Concave polygons
  - Arbitrary polygons
    - Holes
    - Non-coherent
- **Two approaches**
  - Polygon tessellation into triangles
    - edge-flags for internal edges
  - Direct scan-conversion



# Triangle Rasterization

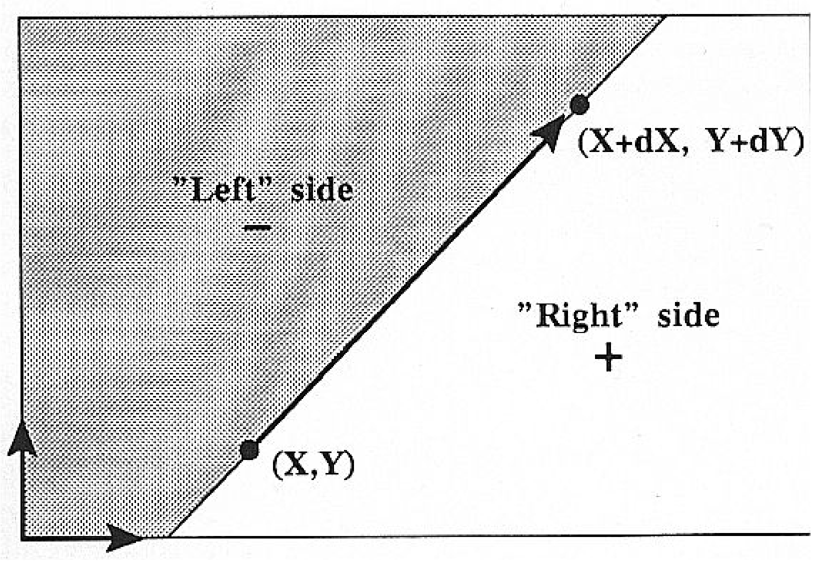
```
Raster3_box(vertex v[3])
{
  int x, y;
  bbox b;
  bound3(v, &b);
  for (y= b.ymin; y < b.ymax; y++)
    for (x= b.xmin; x < b.xmax; x++)
      if (inside(v, x, y))
        fragment(x,y);
}
```



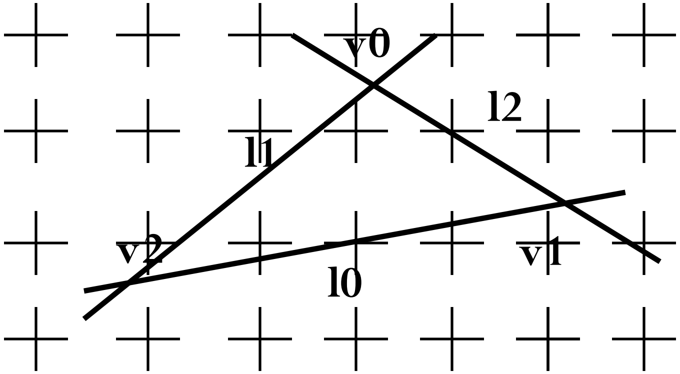
- **Brute-Force algorithm**
- **Possible approaches for dealing with scissoring**
  - Iterate over intersection of scissor box and bounding box, then test against triangle (as above)
  - Iterate over triangle, then test against scissor box

# Incremental Rasterization

- **Approach**
  - Implicit edge functions to describe the triangle $F_i(x,y)= ax+by+c$
  - Point inside triangle, if every $F_i(x,y) <= 0$
  - Incremental evaluation of the linear function F by adding a or b



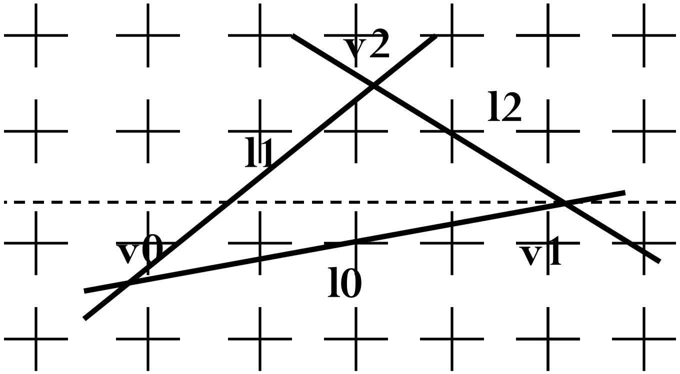
# Incremental Rasterization




```
Raster3_incr(vertex v[3])
{
  edge l0, l1, l2;
  value d0, d1, d2;
  bbox b;
  bound3(v, &b);
  mkedge(v[0],v[1],&l2);
  mkedge(v[1],v[2],&l0);
  mkedge(v[2],v[0],&l1);

  d0 = l0.a * b.xmin + l0.b * b.ymin + l0.c;
  d1 = l1.a * b.xmin + l1.b * b.ymin + l1.c;
  d2 = l2.a * b.xmin + l2.b * b.ymin + l2.c;

  for( y=b.ymin; y<b.ymax, y++ ) {
    for( x=b.xmin; x<b.xmax, x++ ) {
      if( d0<=0 && d1<=0 && d2<=0 ) fragment(x,y);
      d0 += l0.a; d1 += l1.a; d2 += l2.a;
    }
    d0 += l0.a * (b.xmin - b.xmax) + l0.b; . . . }
}
```

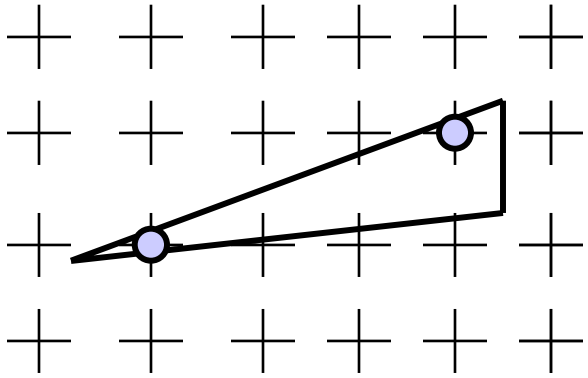
# Triangle Scan Conversion

```
Raster3_scan(vert v[3])
{
  int y;
  edge l, r;
  value ybot, ymid, ytop;

  ybot = ceil(v[0].y);
  ymid = ceil(v[1].y);
  ytop = ceil(v[2].y);

  differencey(v[0],v[2],&l,ybot);
  differencey(v[0],v[1],&r,ybot);

  for( y=ybot; y<ymid; y++ ) {
    scanx(l,r,y);
    l.x += l.dxdy; r.x += r.dxdy;
  }
  differencey(v[1],v[2],&r,ymid);
  for( y=ymid; y<ytop; y++ ) {
    scanx(l,r,y);
    l.x += l.dxdy; r.x += r.dxdy;
  }
}
```

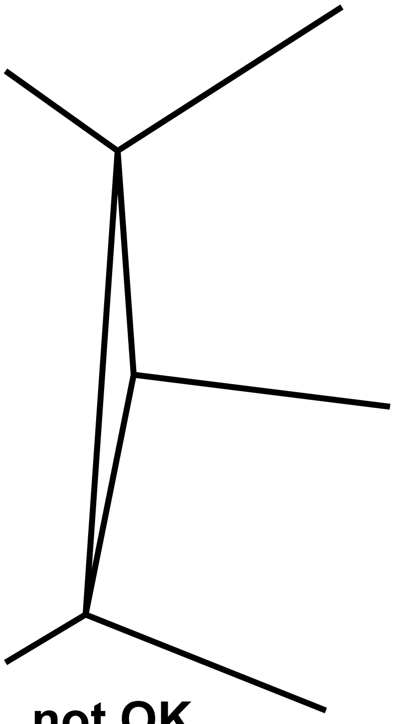



```
differencey(vert a, vert b,
                 edge* e, int y) {
   e->dxdy=(b.x-a.x)/(b.y-a.y);
   e->x=a.x+(y-a.y)*e->dxdy;
}

scanx(edge l, edge r, int y){
  lx= ceil(l.x);
  rx= ceil(r.x);
  for (x=lx; x < rx; x++)
    // ggf. Scissor-Test
    fragment(x,y);
}
```

# Gap and T-Vertices



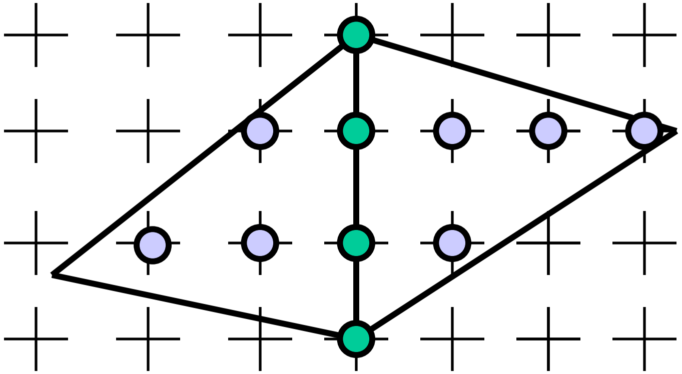
**OK**

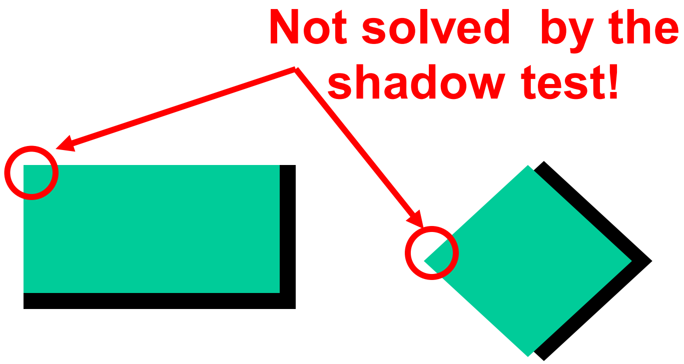


**not OK**
**Modeling problem**

# Problem on Edges

- **Singularity**
  - If term d= ax+by+c = 0
  - Multiple pixels for d <= 0:
    - Problem with some algorithms
      - transparency, XOR, CSG, ...
  - Missing pixels for d < 0:
- **Partial solution: shadow test**
  - Pixels are not drawn on the right and bottom edges
  - Pixels are drawn on the left and upper edges

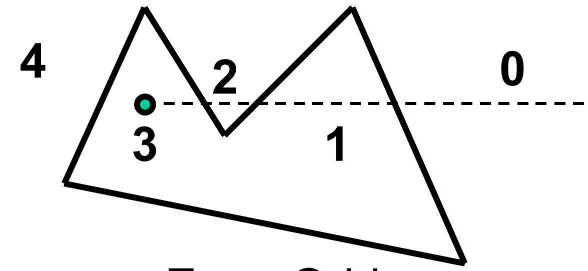




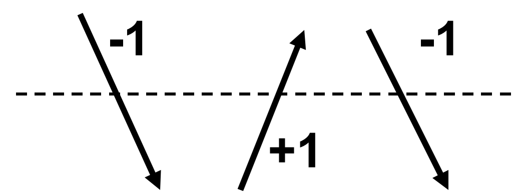

```
inside(value d, value a, value b) { // ax + by + c = 0
  return (d < 0) || (d == 0 && !shadow(a,b));
shadow(value a, value b) {
  return (a > 0) || (a == 0 && b > 0) }
```

# Inside-Outside Tests

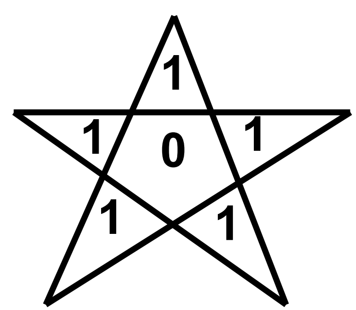
- **What is the interior of a polygon?**
  - Jordan Curve Theorem
    - Any continuous *simple* closed curve in the plane, separates the plane into two disjoint regions, the inside and the outside, one of which is bounded.
  - Even-odd rule (odd parity rule)
    - Counting the number of edge crossings with a ray starting at the queried point **P**
    - Inside, if the number of crossings is odd
  - Nonzero winding number rule
    - Signed intersections with a ray
    - Inside, if the number is not equal to zero
  - Differences only in the case of non-simple curves (self-intersection)



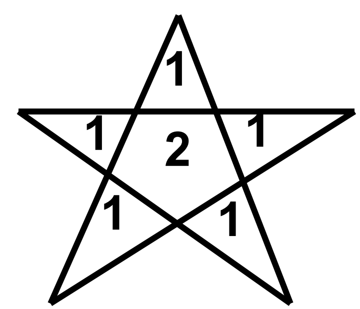

Even-Odd




Winding



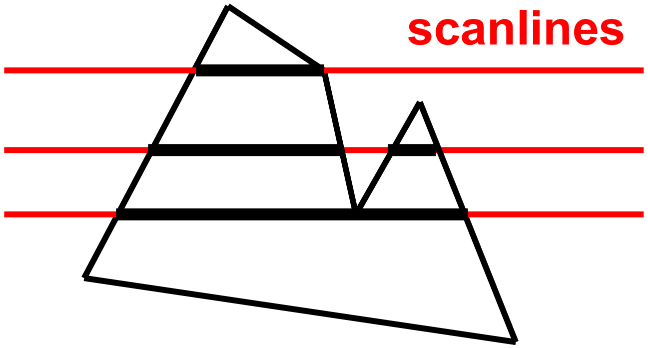

Even-Odd

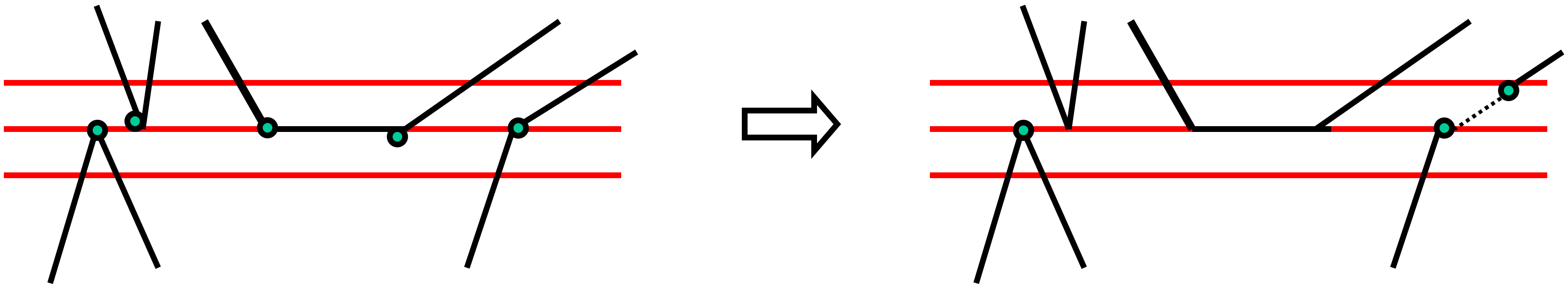



Winding

# Polygon Scan-Conversion

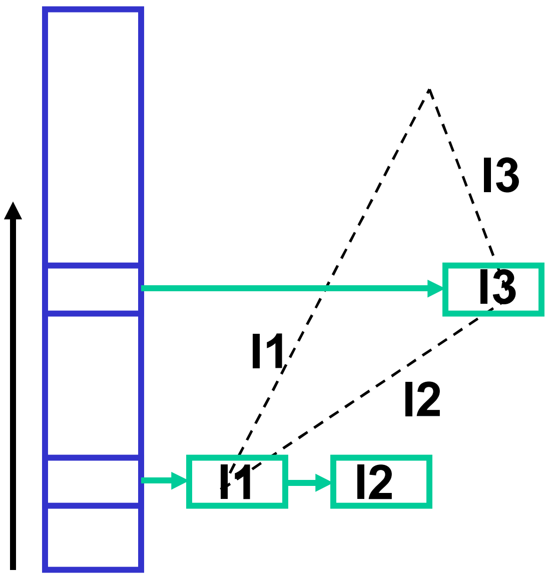
- **Special cases**
  - Edge along a scanline
    - shadow test:
      - draw the upper edge
      - skip the bottom edge
  - Vertex at a scanline
    - If edges sharing the vertex are located on the **same side** of the scanline – properly handled
    - If edges sharing the vertex are located on the **opposite sides** of the scanline – one edge (bottom) is shortened: the $y_{min}/y_{max}$ rule
    - Complex situations
      - In general use randomization: Offset point by ε



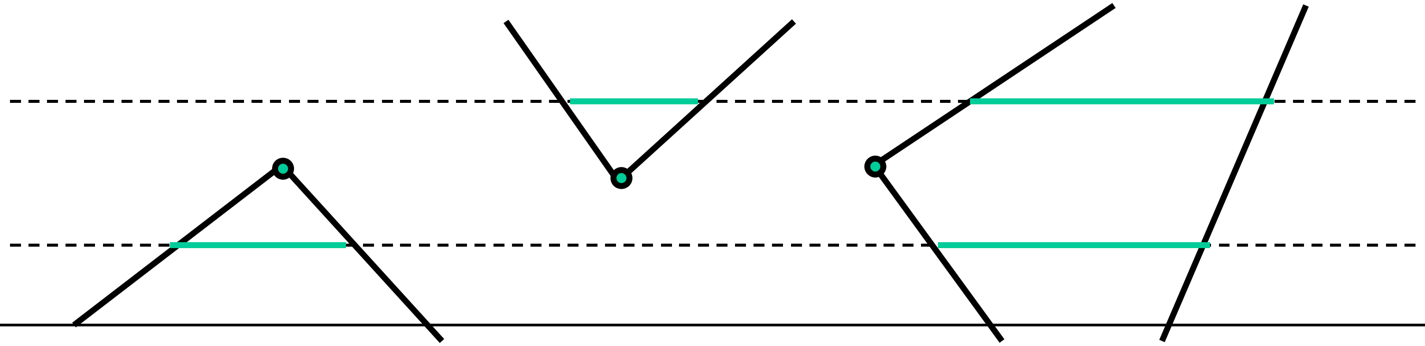
# Scanline Algorithm

- **Incremental algorithm**
  - Use the odd-even parity rule to detemine that a point is inside a polygon
  - Utilization of coherence
    - along the edges
    - on scanlines
    - „sweepline-algorithm“
  - Edge-Table initialization :
    - Bucket sort (one bucket for each scanline)
    - Edges ordered by xmin
    - Linked list of edge-entries
      - ymax
      - xmin
      - dx/dy
      - link to triangle data



# Scanline Algorithm

- **For each scan line**
  - Update the Active-Edge-Table
    - Linked-list of entries
      - Link to edge-entries,
      - x, horizontal increment of depth, color, etc
    - Remove edges if theirs ymax is reached
    - Insert new edges (from Edge-Table)
  - Sorting
    - Incremental update of x
    - Sorting by X-coordinate of the intersection point with scanline
  - Filling the gap between pairs of entries



# Scanline Algorithm

- **Remarks**
  - Used in software implementations
  - Convex and concave polygons, holes
  - Pixel supersampling
    - Multiple scanlines per pixel