EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
SCIENCE
Computer Graphics

Computer Graphics (Graphische Datenverarbeitung)

- Rasterization & Clipping -

Hendrik Lensch

WS 2021/2022

Corona

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Regular random lookup of the 3G certificates

- Contact tracing: We need to know who is in the class room
  - New ILIAS group for every lecture slot
  - Register via ILIAS or this QR code (only if you are present in this room)

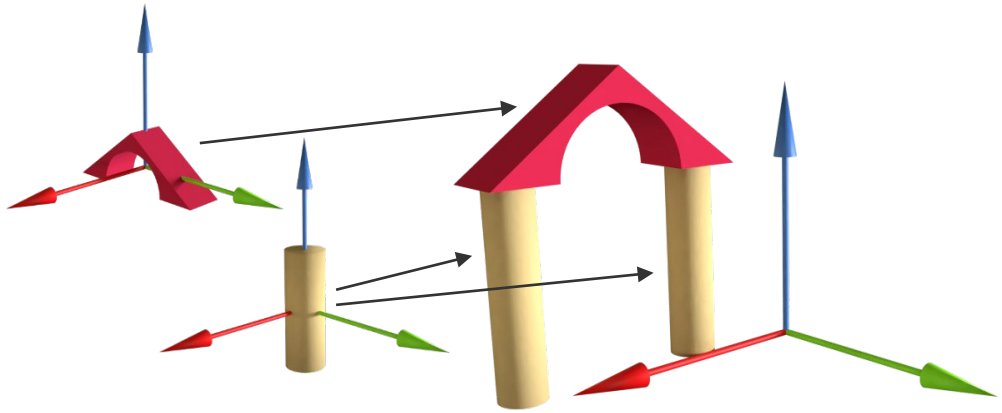
Overview

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Last lecture:
  - Camera Transformations
  - Projection
- Today:
  - Rasterization of Lines and Triangles
  - Clipping
- Next lecture:
  - OpenGL

Camera Transformations

- Model transformation
  - Object space to world space



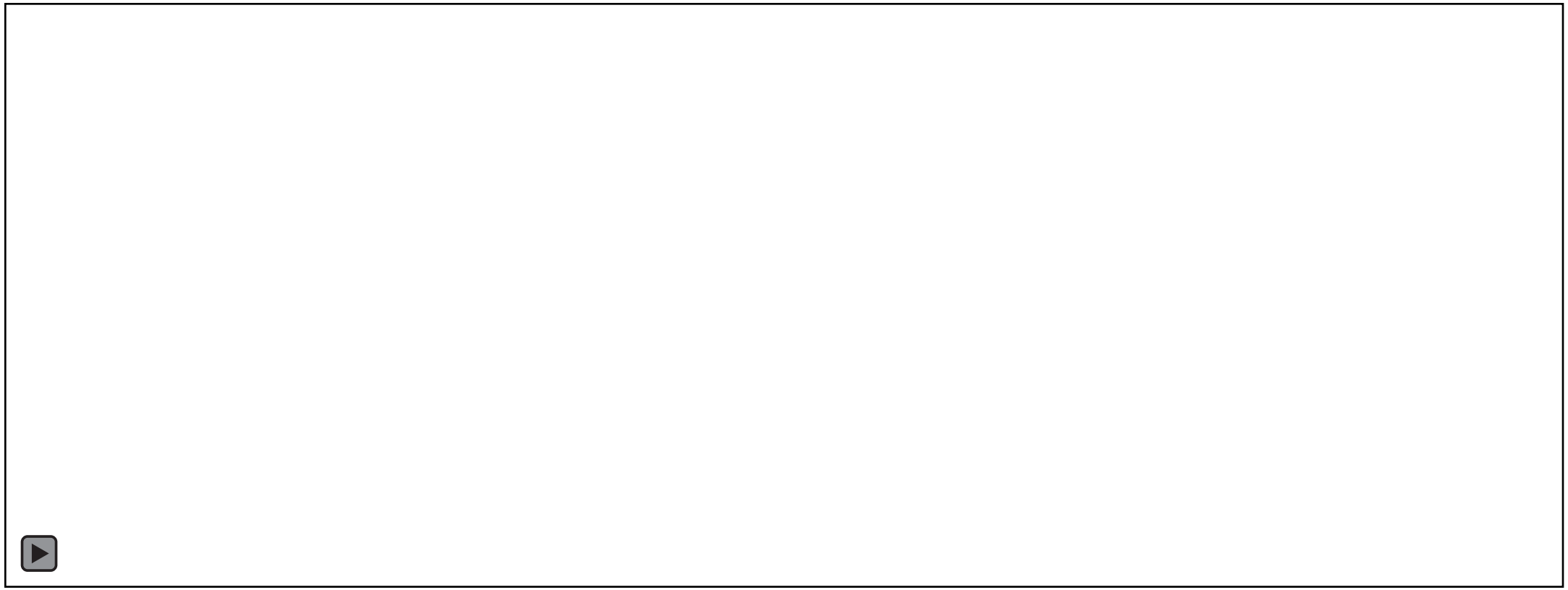
- View transformation
  - World space to eye space

- Combination:
  Modelview transformation
  - Used by OpenGL

Camera Transformation

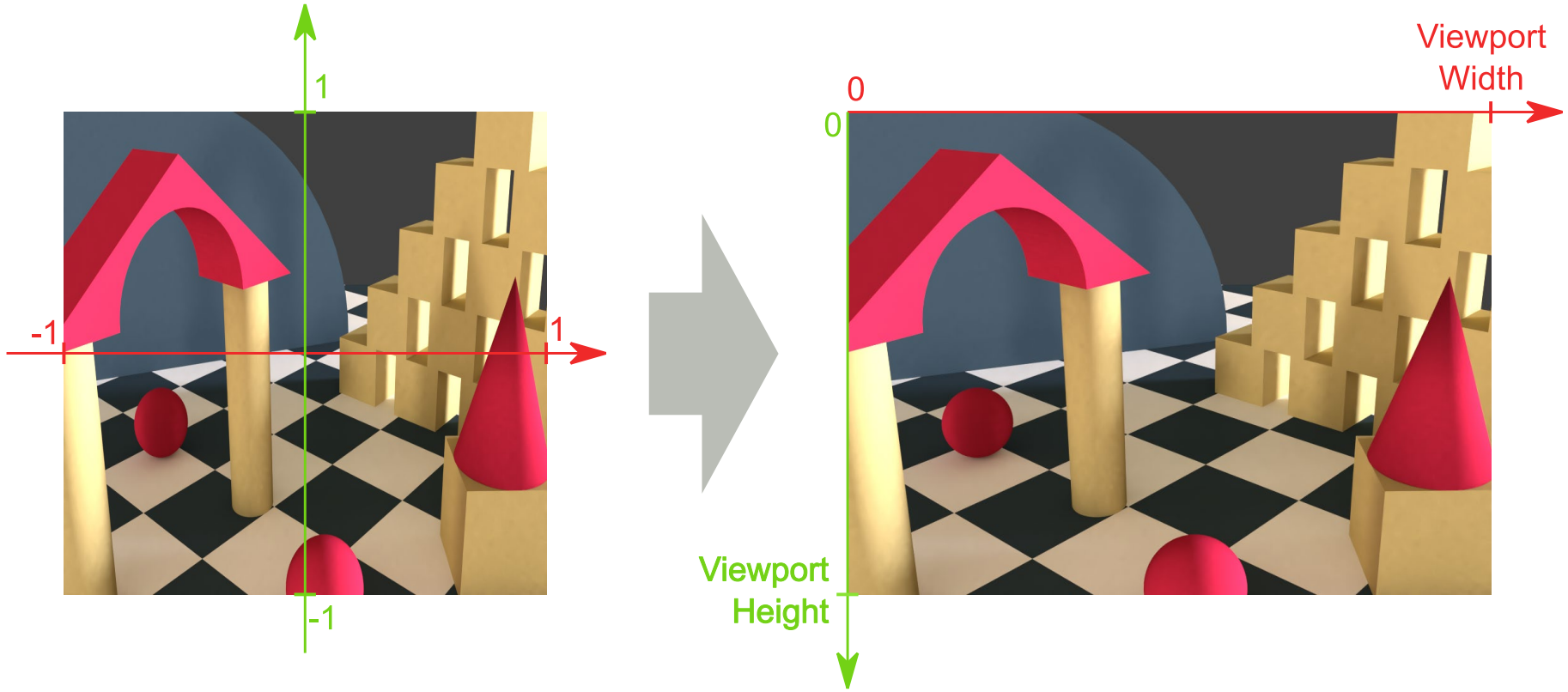
EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Projection transformation
  - Eye space to
    normalized device space
  - Parallel or perspective projection



Camera Transformation

- Viewport transformation
  - Normalized device space to window (raster) coordinates



Camera Transformation

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Complete Transformation
  - Perspective Projection

$$K = T_{raster} S_{raster} \quad P_{parallel} \quad P_{persp} S_{far} S_{xy} H \quad RT$$

  - Orthographic Projection

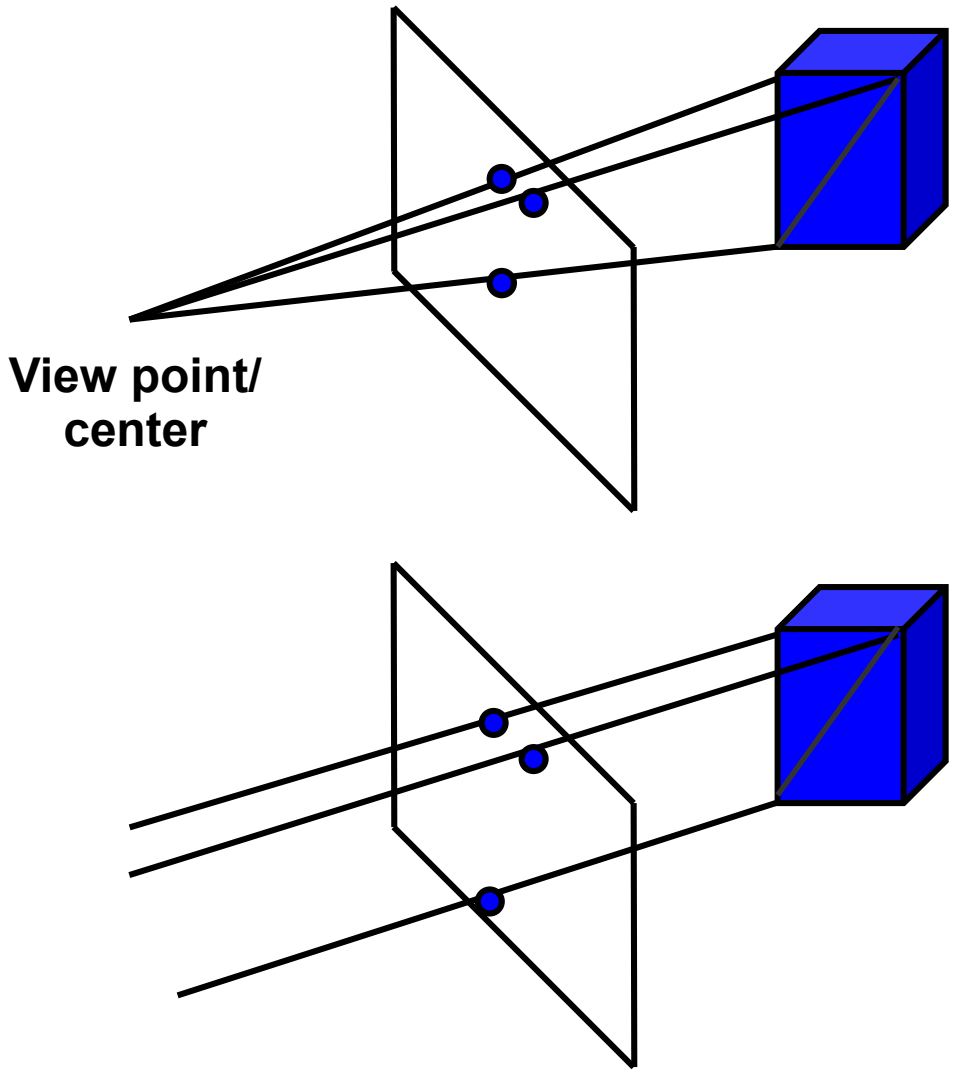
$$K = T_{raster} S_{raster} \quad P_{parallel} \quad S_{xyz} T_{near} H \quad RT$$

- Other representations
  - Different camera parameters as input
  - Different canonical viewing frustum
  - Different normalized coordinates
    - $[-1 .. 1]^3$ versus $[0 .. 1]^3$ versus ...
  - ...

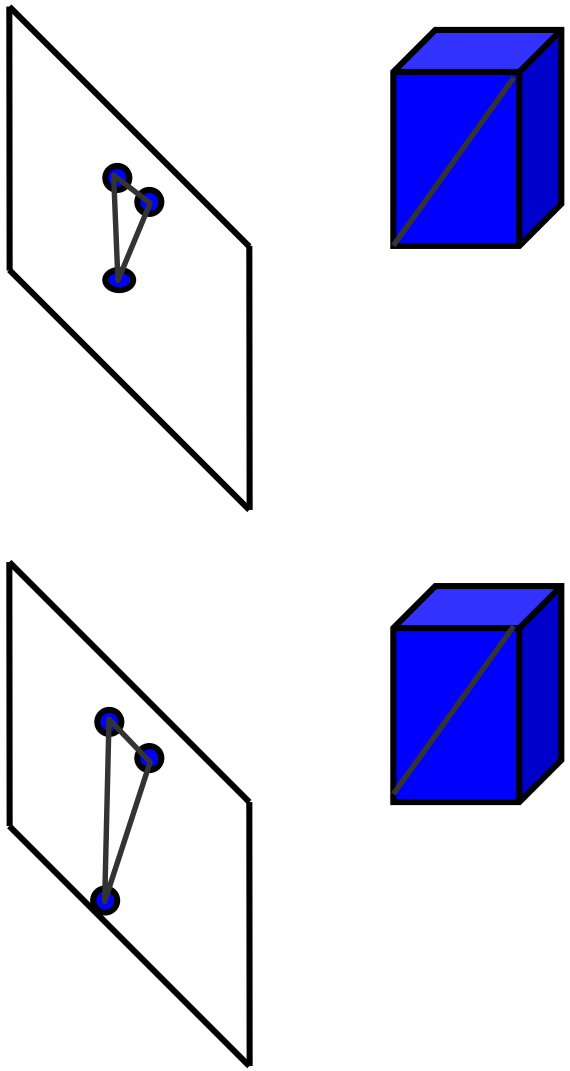
→ ***Different transformation matrices***

Projection of Vertices

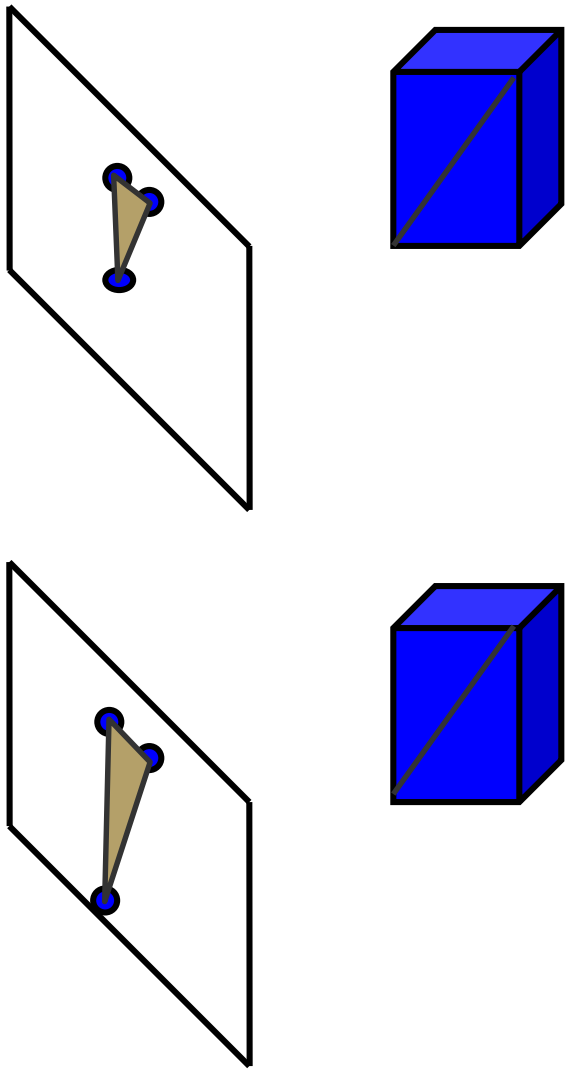
$$v_1' = Kv_1$$
$$v_2' = Kv_2$$
$$v_3' = Kv_3$$



Draw Outline



Fill Outline



Rasterization

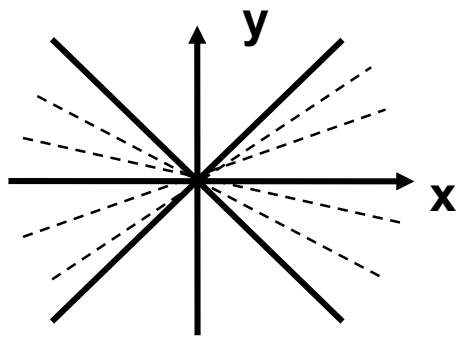
Rasterization

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- 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, PDF display
  - 3D-raster graphics
  - 3D volume modeling and rendering
  - Volume operations (CSG operations, collision detection)
  - Space subdivision
    - Construction and traversing

Line 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

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- 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 x_i = x_0 to x_e
  y_i = m * x_i + B
  setpixel(x_i, Round(y_i))     // Round(y_i)=Floor(y_i+0.5)
```

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

Lines: DDA

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

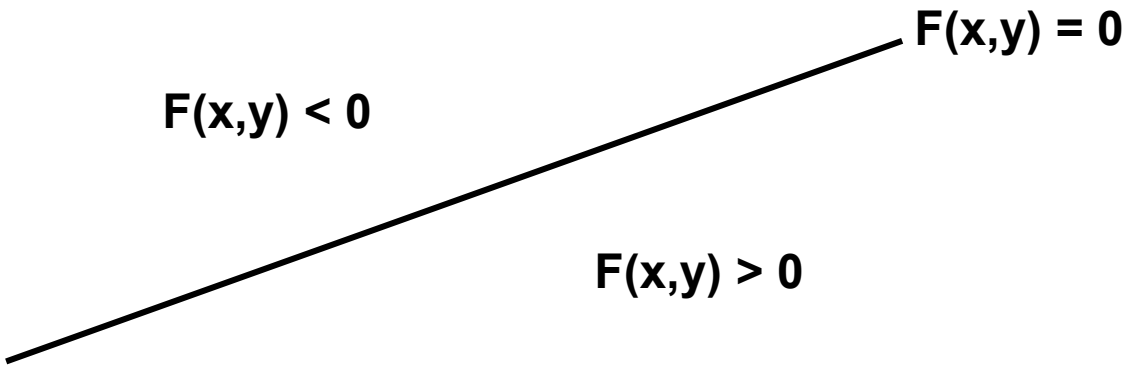
- DDA: Digital Differential Analyzer
  - Origin of solvers for simple incremental differential equations (the Euler method)
    - Per step in time: $x' = x + dx/dt,\ y' = y + dy/dt$
- Incremental algorithm
  - Per pixel
    - $x_{i+1} = x_i + 1$
    - $y_{i+1} = m\ (x_i + 1) + B = y_i + m$
    - $\mathtt{setpixel}(x_{i+1}, \mathtt{Round}(y_{i+1}))$
- Remark
  - Utilization of line coherence through 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

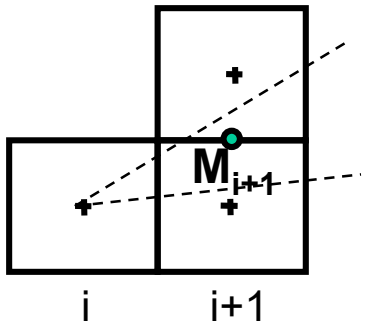
$$F(x,y) = dy\,x - dx\,y + dx\,B = 0$$

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



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
    - $d_{i+1}= d_i + a + b = d_i + dy – dx$
    - $y= y + 1$
  - $x = x +1$

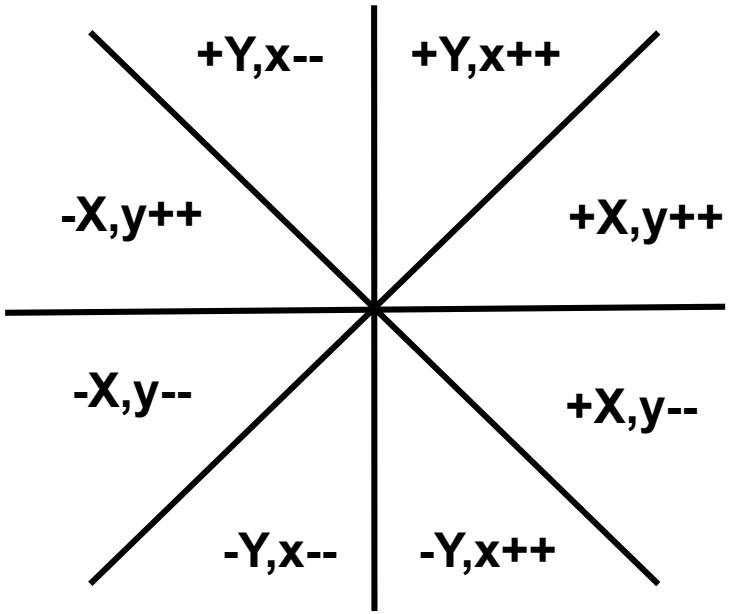
Lines: Integer Bresenham

- 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
  - No accumulated error

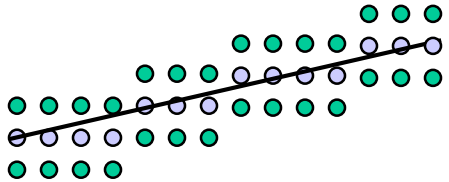
Lines: Arbitrary Directions

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



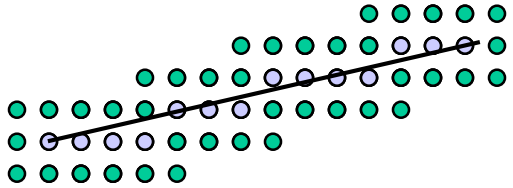
Thick Lines

- Pixel replication



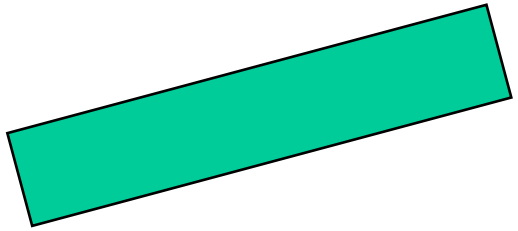
  - Problems with even-numbered widths,
  - Varying intensity of a line as a function of slope

- The moving pen



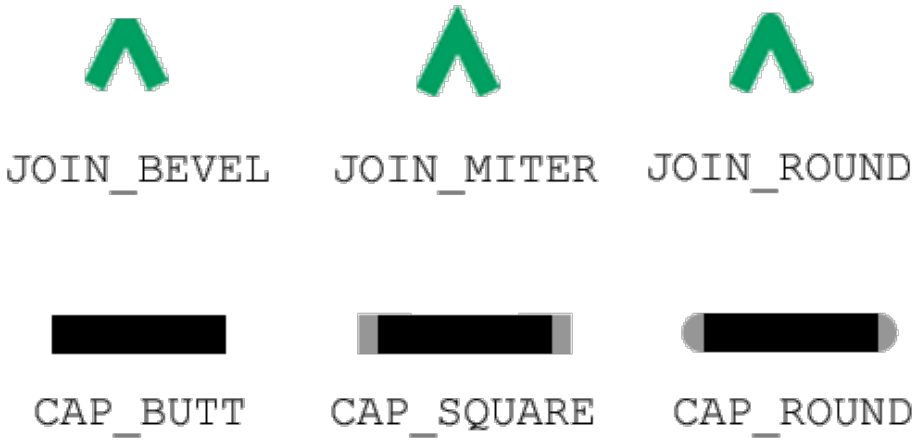
  - For some pen footprints the thickness of a line might change as a function of its slope
  - Should be as „round“ as possible

- Filling areas between boundaries



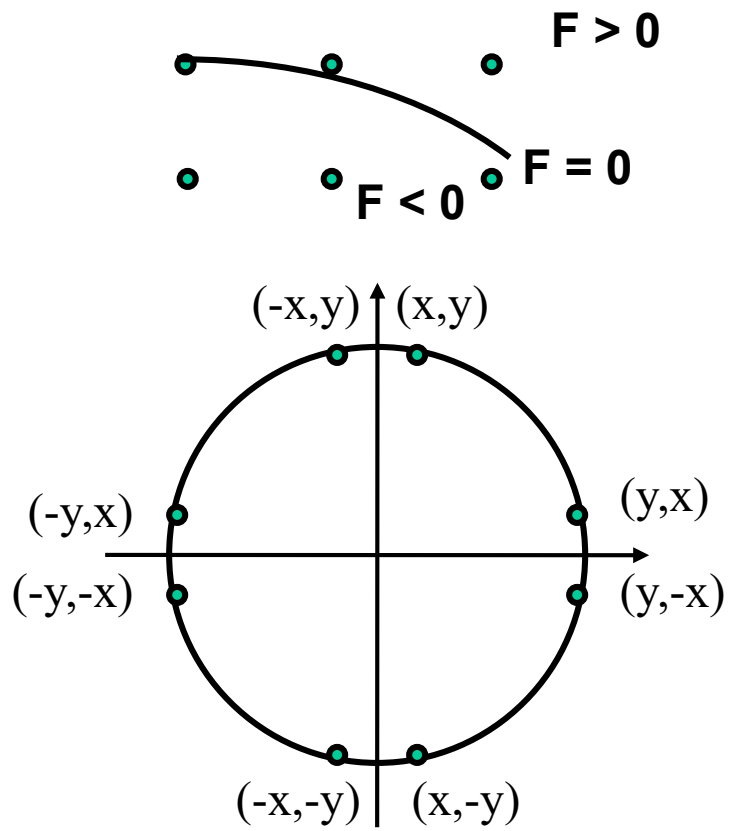
Handling Start And End Points

- 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

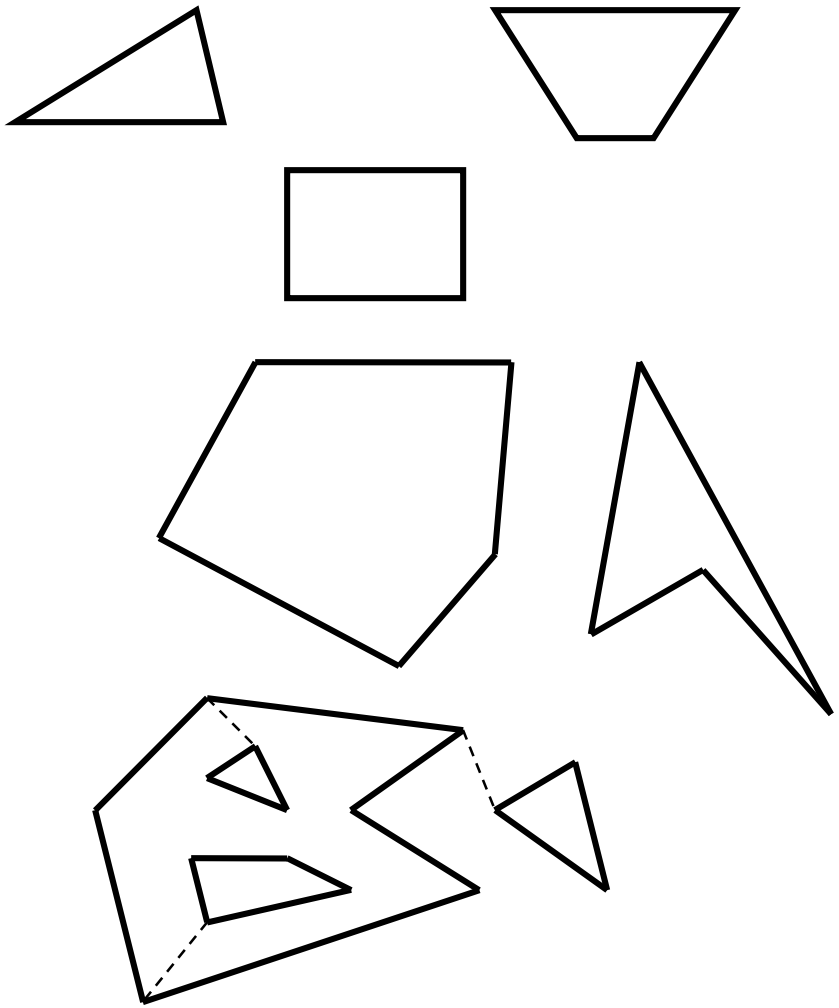


- Eight-way symmetry: only one 45^O segment is needed to determine all pixels in a full circle

Polygon Rasterization

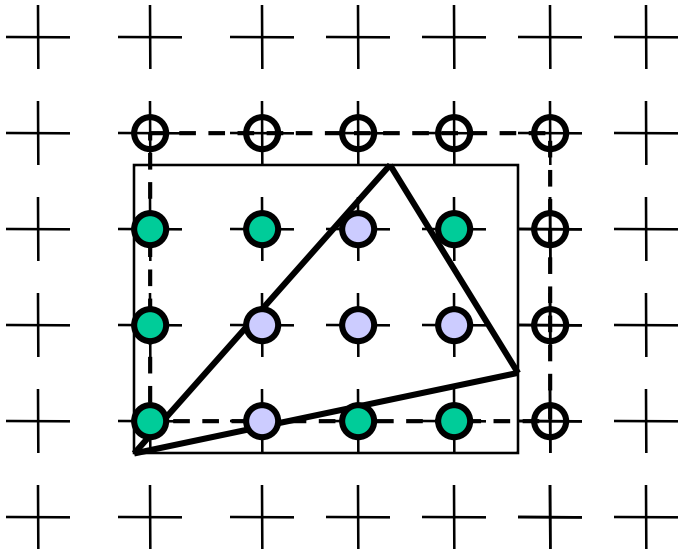
Reminder: Polygons

- 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

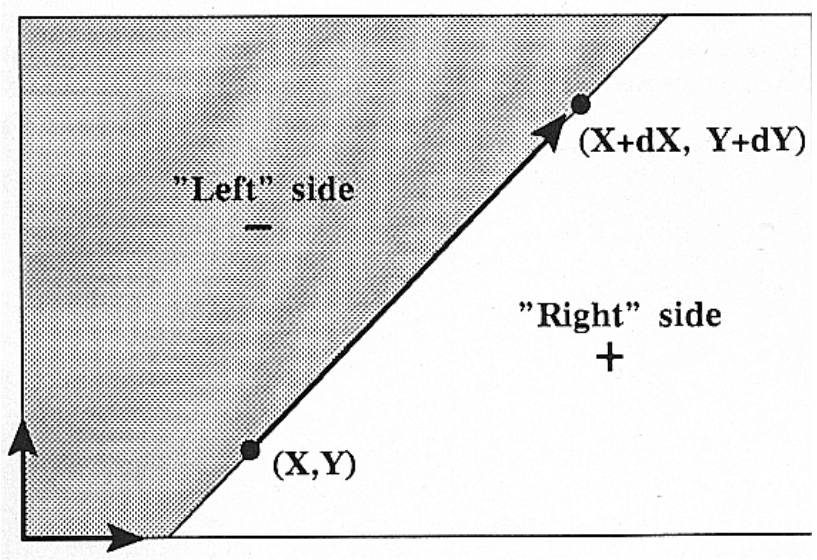
```
Raster3_box(vertex v[3])
{
  int x, y;
  bbox b;
  bound3(v, &b);
  for (y= b.ymin; y < b.ymax; y++)
    for (x= b.xmin; x < b.xmax; x++)
      if (inside(v, x, y))
       fragment(x,y);
}
```



- **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

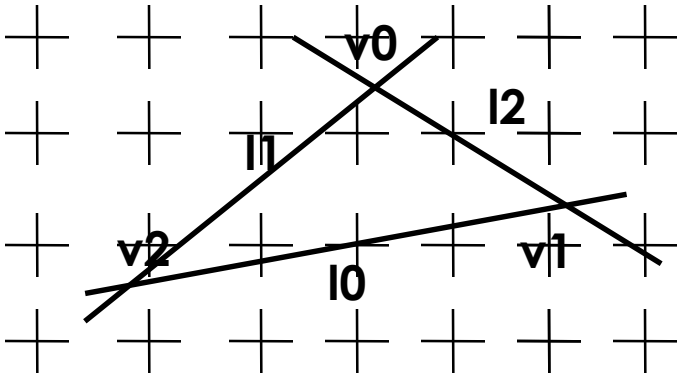
Incremental Rasterization

- 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



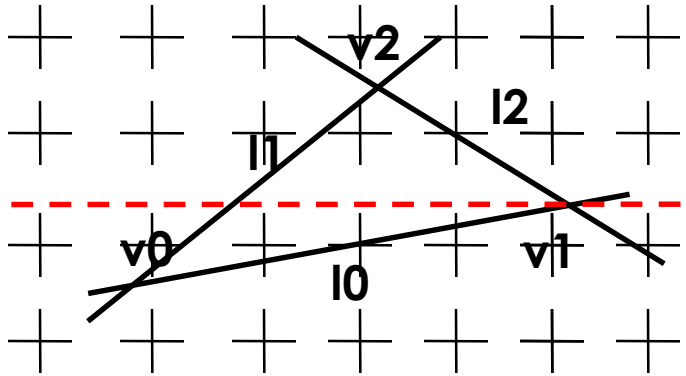
Incremental Rasterization

```
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;
       . . .
  }
}
```



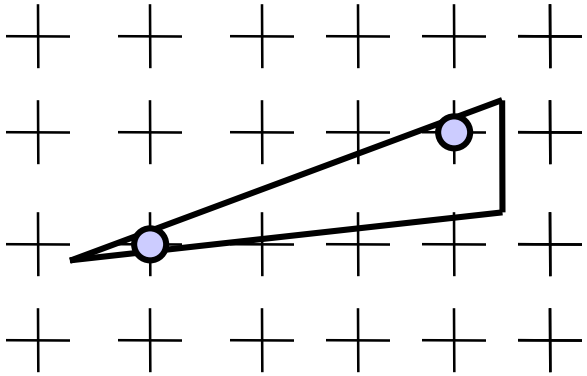
Triangle Scan Conversion

```
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;
  }
}
```

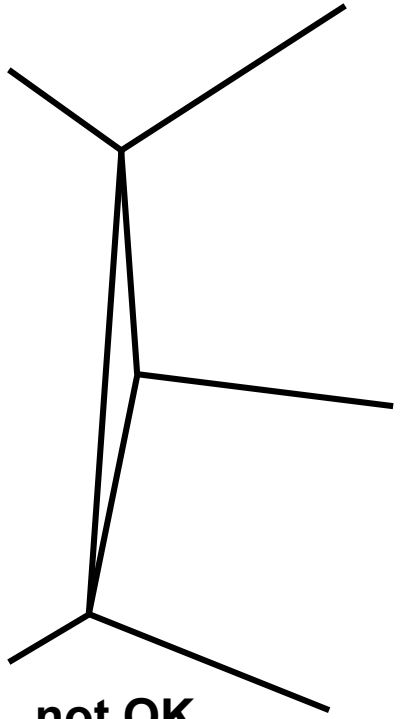



```
differencey(vert a, vert b,
            edge* e, int y) {
   e->dxdy=(b.x-a.x)/(b.y-a.y);
   e->x=a.x+(y-a.y)*e->dxdy;
}

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



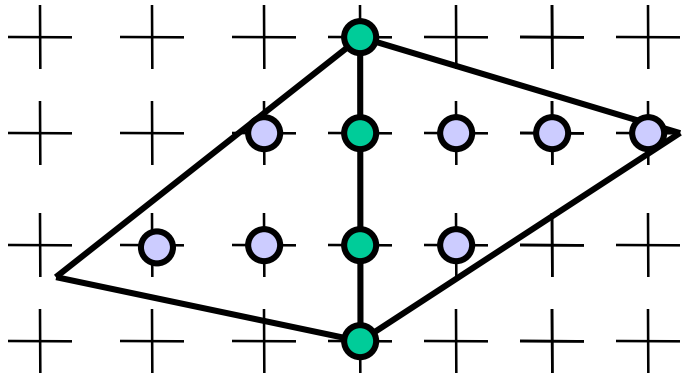
OK

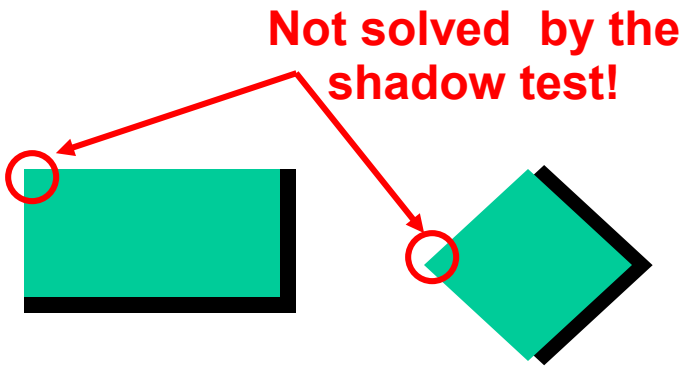


not OK
Modeling problem

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



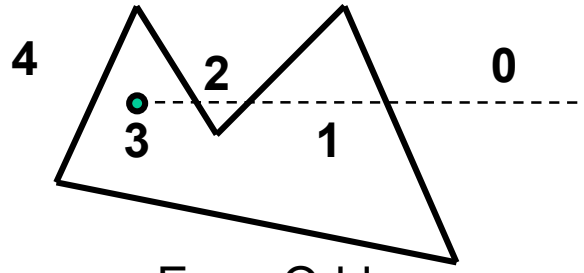



```
inside(value d, value a, value b) { // ax + by + c = 0
  return (d < 0) || (d == 0 && !shadow(a,b));
shadow(value a, value b) {
  return (a > 0) || (a == 0 && b > 0) }
```

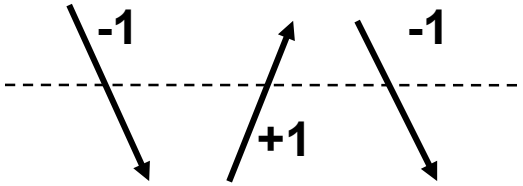
Inside-Outside Tests

Inside-Outside Tests

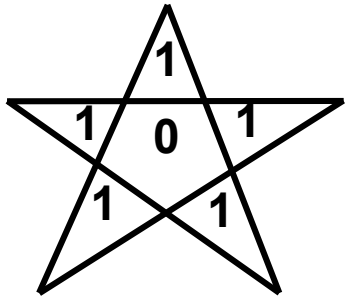
- What is the interior of a polygon?
  - Jordan Curve Theorem
    - Any continuous *simple* closed curve in the plane, separates the plane into two disjoint regions, the inside and the outside, one of which is bounded.
  - Even-odd rule (odd parity rule)
    - Counting the number of edge crossings with 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)



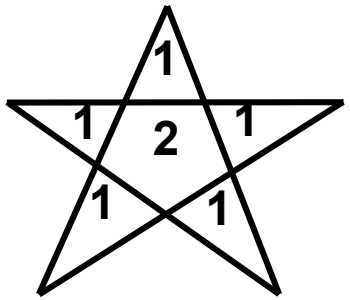
Even-Odd



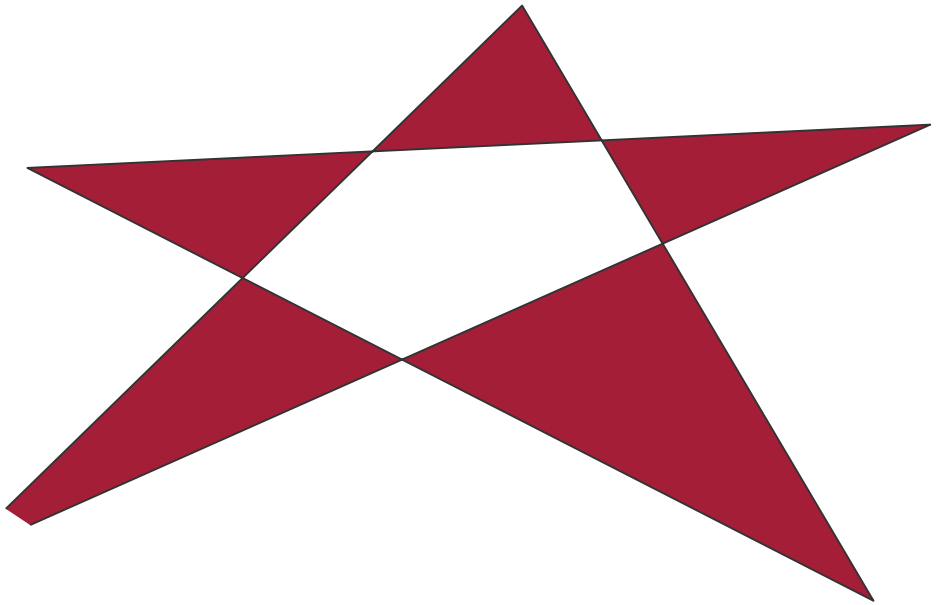
Winding



Even-Odd

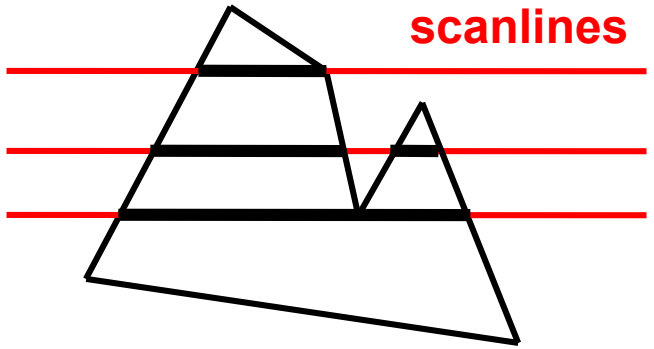


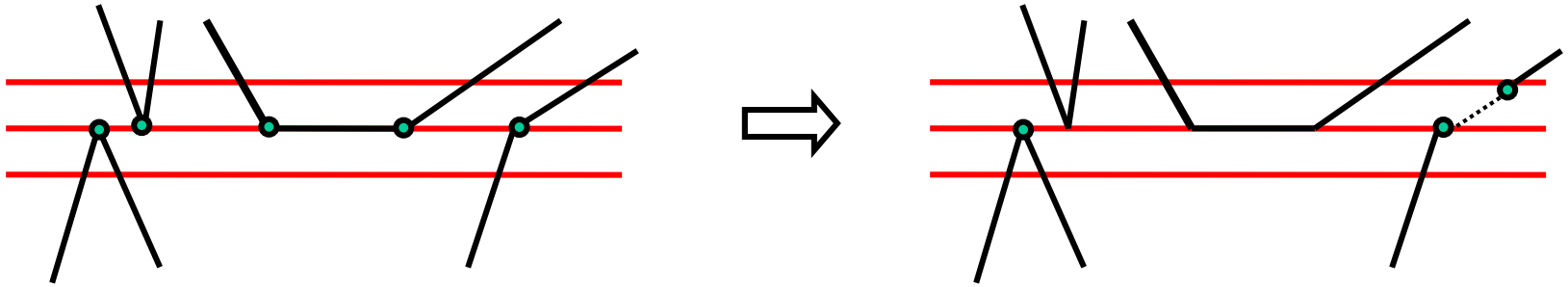
Winding



Polygon Scan-Conversion

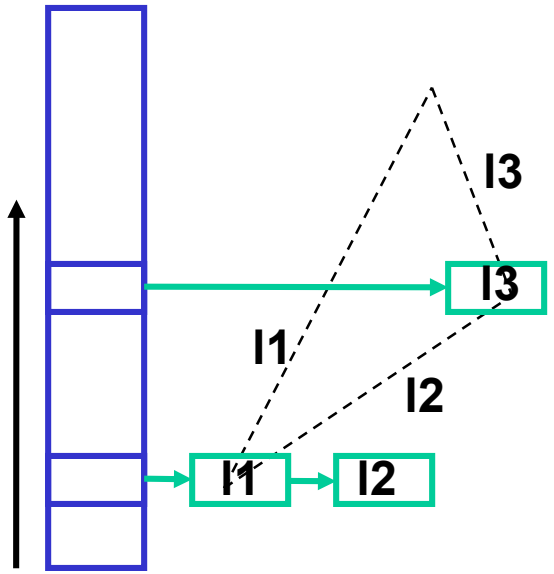
- Special cases
  - Edge along a scanline
    - shadow test:
      - draw the upper edge
      - skip the bottom edge
  - Vertex at a scanline
    - If edges sharing the vertex are located on the **same side** of the scanline – properly handled
    - If edges sharing the vertex are located on the **opposite 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





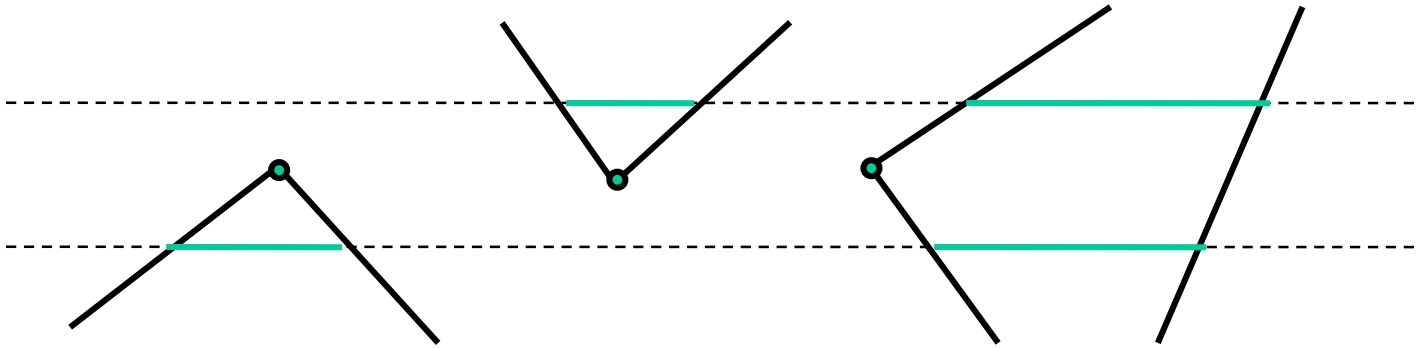
Scanline Algorithm

- 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



Clipping

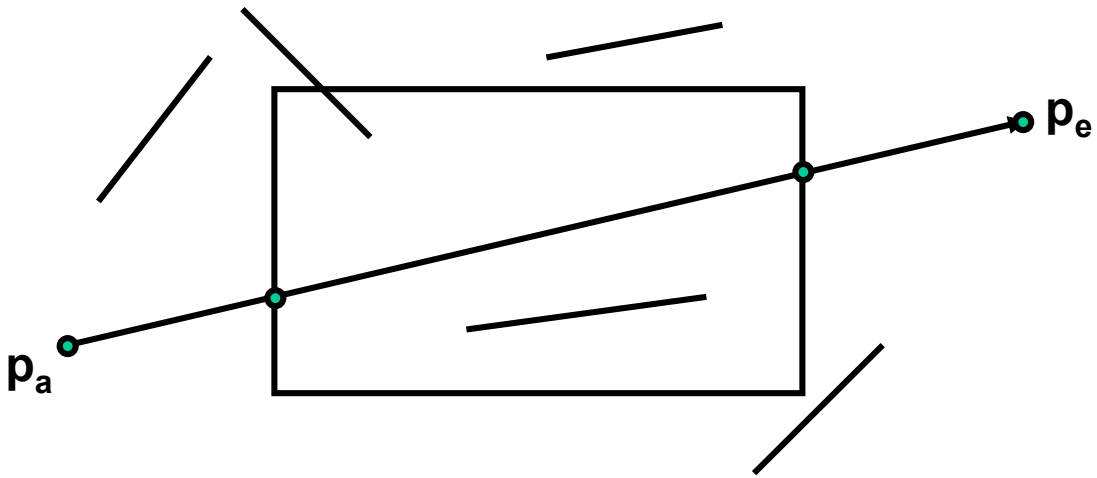
Clipping

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Motivation
  - Happens after transformation from 3D to 2D
  - Many primitives will fall (partially) outside of display window
    - E.g. if standing inside a building
  - Eliminates non-visible geometry early in the pipeline
  - Must cut off parts outside the window
    - Cannot draw outside of window (e.g. plotter)
    - Outside geometry might not be representable (e.g. in fixed point)
  - Must maintain information properly
    - Drawing the clipped geometry should give the correct results
    - Type of geometry might change
      - Cutting off a vertex of a triangle produces a quadrilateral
      - Might need to be split into triangle again
    - Polygons must remain closed after clipping

Line clipping

- Definition Clipping:
  - Cut off parts of objects, which lie outside/inside of a defined region.
  - Often: Clipping against a viewport (2D) or
    a canonical view-volume (3D)

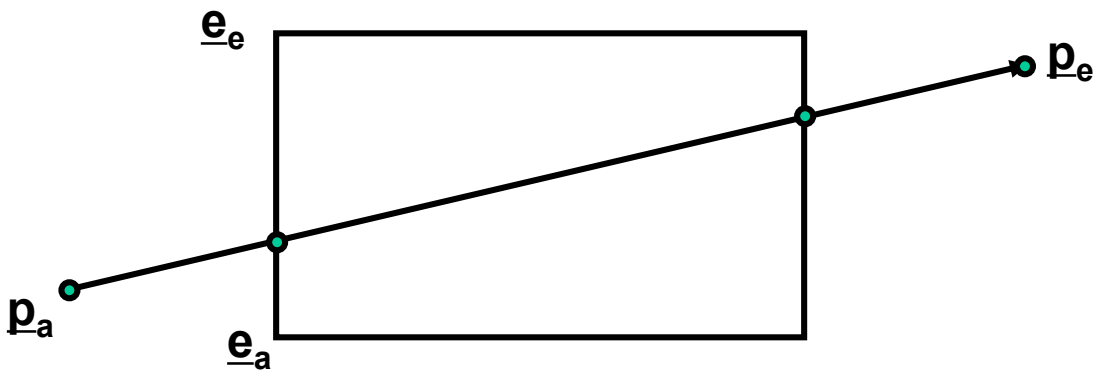


Brute-force method

- Brute-force line clipping at the viewport
  - If both points $\underline{p}_a$ and $\underline{p}_e$ are inside,
    - Accept the whole line
  - Otherwise, clip the line at each edge

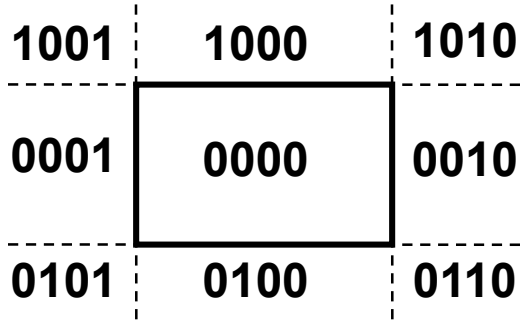
$$p = p_a + t_{line}(p_e - p_a) = e_a + t_{edge}(e_e - e_a)$$

  - Intersection point, if $0 \le t_{line}, t_{edge} \le 1$
  - Pick up suitable end points from the intersection points for the line



Cohen-Sutherland (´74)

- Advantage: divide and conquer
  - Efficient trivial accept and trivial reject
  - Non-trivial case: divide and test
- Outcodes of points:
  - Bit encoding (**o**ut**c**ode, OC)
    - Each edge defines a half space
    - Set bit, if point is outside
- Trivial cases
  - Trivial accept:
    - $(OC(p_a)$ **OR** $OC(p_e)) = 0$
  - Trivial reject:
    - $(OC(p_a)$ **AND** $OC(p_e)) \neq 0$
  - Edges has to be clipped to all edges where bits are set:
    - $OC(p_a)$ **XOR** $OC(p_e)$



Bit order: **Top, Bottom, Right, Left**

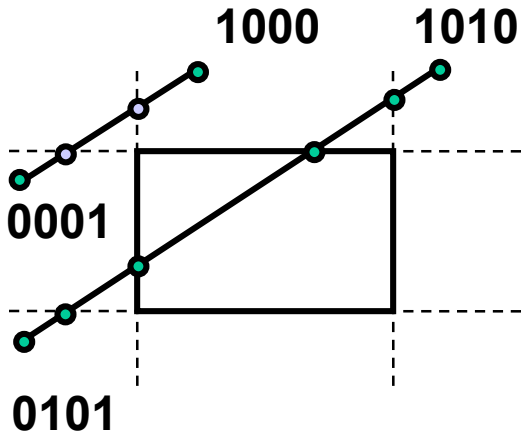
Viewport (x_{min}, y_{min}, x_{max}, y_{max})

Cohen-Sutherland

- Clipping

```
... // trivial cases
for each vertex p
  oc= OC(p)
  for each edge e
    if (oc[e]) {
      p= cut(p,e);
      oc= OC(p);
    }
  Reject, if point outside
```

- Intersection calculation for $x=x_{min}$

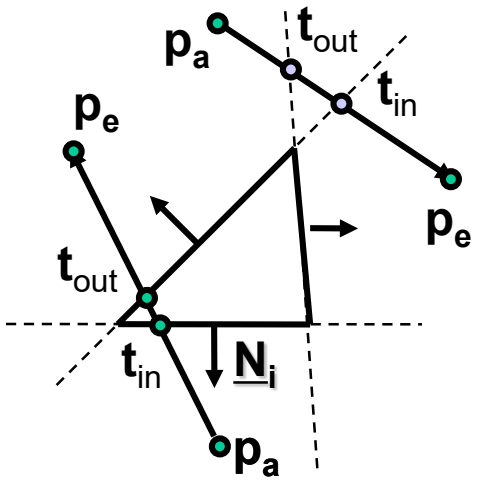


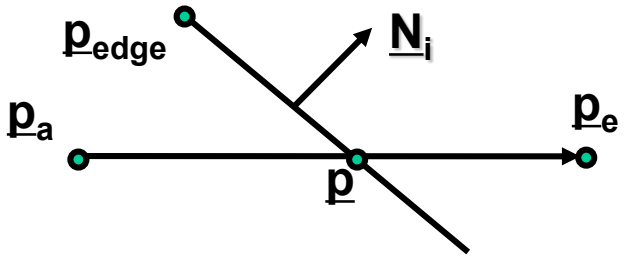
$$\frac{y - y_a}{y_e - y_a} = \frac{x - x_a}{x_e - x_a}$$

$$y = y_a + (x - x_a)\frac{y_e - y_a}{x_e - x_a}$$

Cyrus-Beck (´78) Clipping against Polygons

- Parametric line-clipping algorithm
  - Only convex polygons: max. 2 intersection points
  - Use edge orientation
- Idea:
  - Clipping line $\underline{p}_a + t_i(\underline{p}_e - \underline{p}_a)$ with each edge
  - Intersection points sorted by parameter t_i
  - Select
    - t_{in}: entry point $((\underline{p}_e - \underline{p}_a)\cdot\underline{N}_i < 0)$ with largest t_i and
    - t_{out}: exit point $((\underline{p}_e - \underline{p}_a)\cdot\underline{N}_i > 0)$ with smallest t_i
  - If $t_{out} < t_{in}$, line lies completely outside
- Intersection calculation:





$$\left(p - p_{edge}\right)\cdot N_i = 0$$

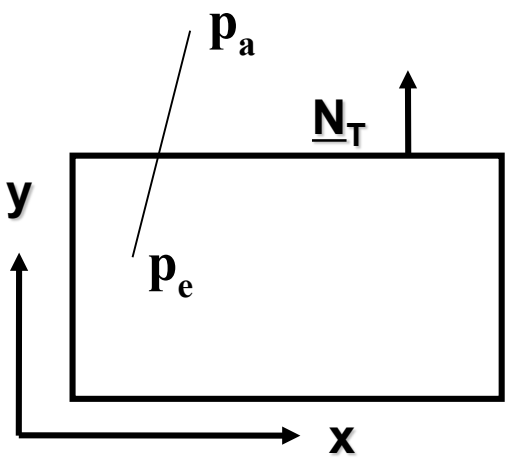
$$t_i\left(p_e - p_a\right)\cdot N_i + \left(p_a - p_{edge}\right)\cdot N_i = 0$$

$$t_i = \frac{\left(p_{edge} - p_a\right)\cdot N_i}{\left(p_e - p_a\right)\cdot N_i}$$

Liang-Barsky (´84)

- Cyrus-Beck for axis-parallel rectangles
  - Using Window-Edge-Coordinates (with respect to an edge T)

$$WEC_T(p) = (p - p_T) \cdot N_T$$



- Example: top (y= y_{max})

$$N_T = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, p_a - p_T = \begin{pmatrix} x_a - x_{max} \\ y_a - y_{max} \end{pmatrix}$$

$$t_T = \frac{(p_a - p_T) \cdot N_T}{(p_a - p_e) \cdot N_T} = \frac{WEC_T(p_a)}{WEC_T(p_a) - WEC_T(p_e)} = \frac{y_a - y_{max}}{y_a - y_e}$$

  - Window-Edge-Coordinate (WEC): Decision function for an edge
    - Directed distance to edge
      - Only sign matters, similar to Cohen-Sutherland opcode
    - Sign of the dot product determines whether the point is in or out
    - Normalization unimportant

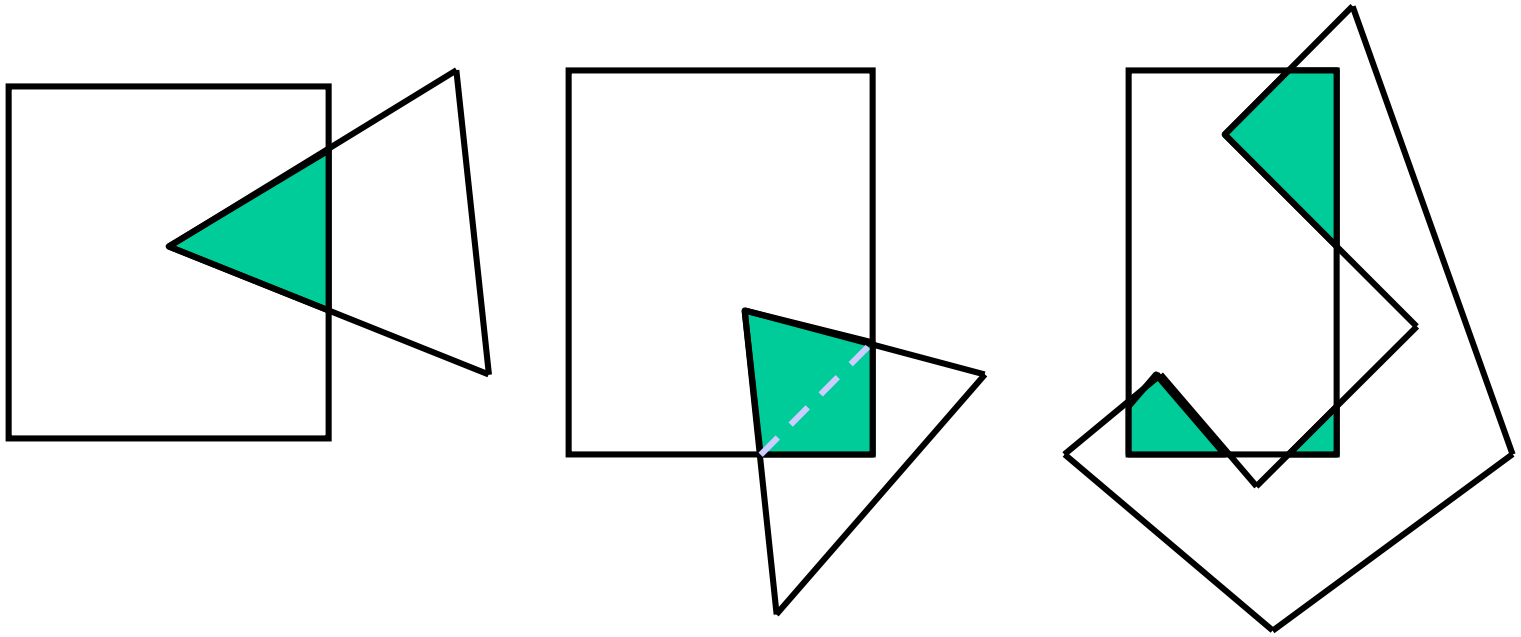
Line clipping - Summary

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

- Cohen-Sutherland, Cyrus-Beck, and Liang-Barsky algorithms readily extend to 3D
- Cohen-Sutherland algorithm
  - Efficient when a majority of lines can trivially accepted or rejected
    - Very large clip rectangles: almost all lines inside
    - Very small clip rectangles: almost all lines outside
  - Repeated clipping for remaining lines
  - Testing for 2D/3D point coordinates
- Cyrus-Beck (Liang-Barsky) algorithms
  - Efficient when many lines must be clipped
  - Testing for 1D parameter values
  - Testing intersections always for all clipping edges (in the Liang-Barsky trivial rejection testing possible)

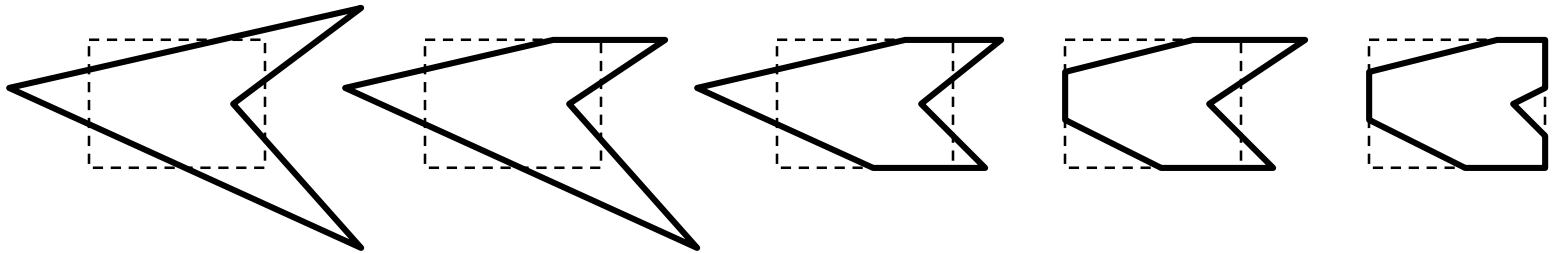
Polygon Clipping

- Extending line clipping
  - Polygons have to remain closed
    - Filling, hatching, shading, ...

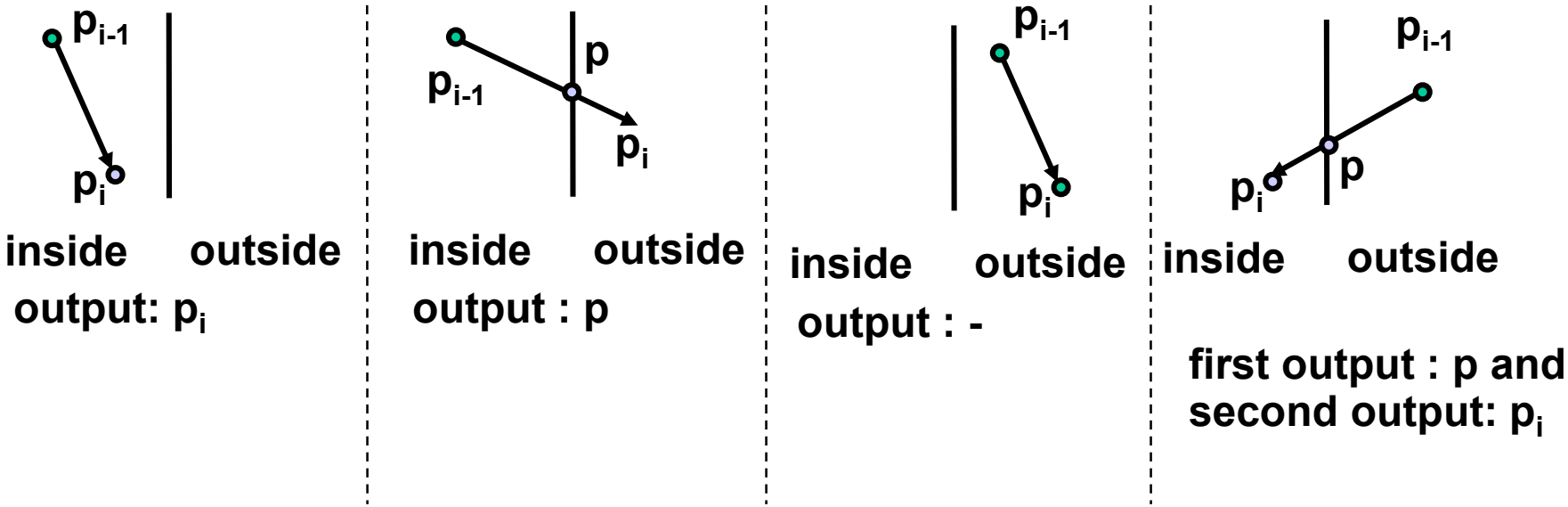


Sutherland-Hodgeman (´74)

- Idea:
  - Iterative clipping against each clipping line

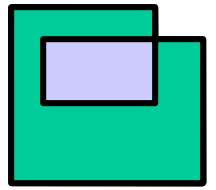


  - Local operations on pi-1 and pi



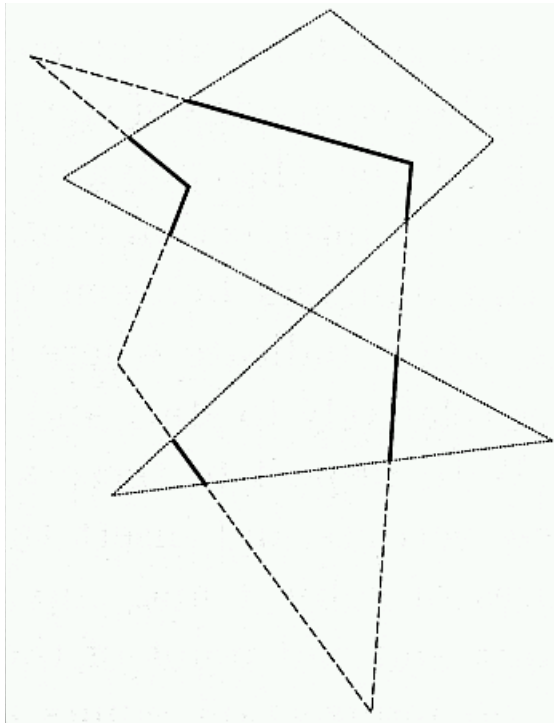
Other clipping algorithms

- Weiler & Atherton (´77)
  - Arbitrary concave polygons with holes against each other
- Vatti (´92)
  - Also with self-overlap
- Greiner & Hormann (TOG ´98)
  - Simpler and faster as Vatti
  - Also supports boolean operations
  - Idea:
    - Odd winding number rule
      - Intersection with the polygon leads to a winding number ±1
    - Walk along both polygons
    - Alternate winding number
    - Mark point of entry and point of exit
    - Combine results

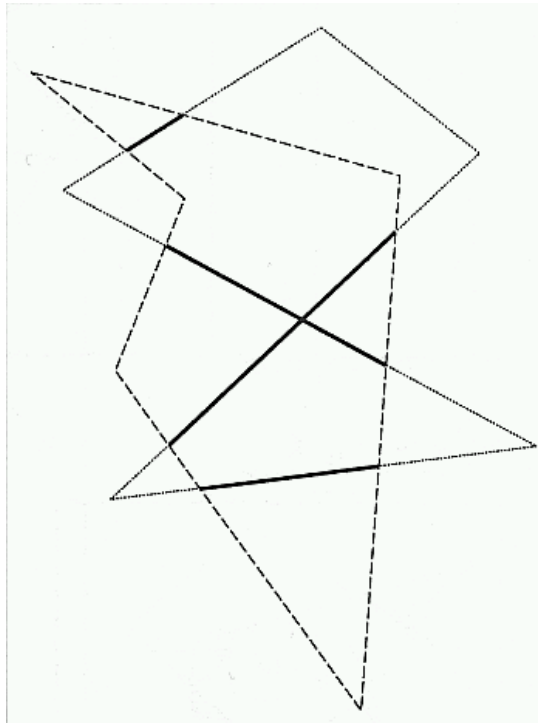


Non-zero WN: In
Even WN: Out

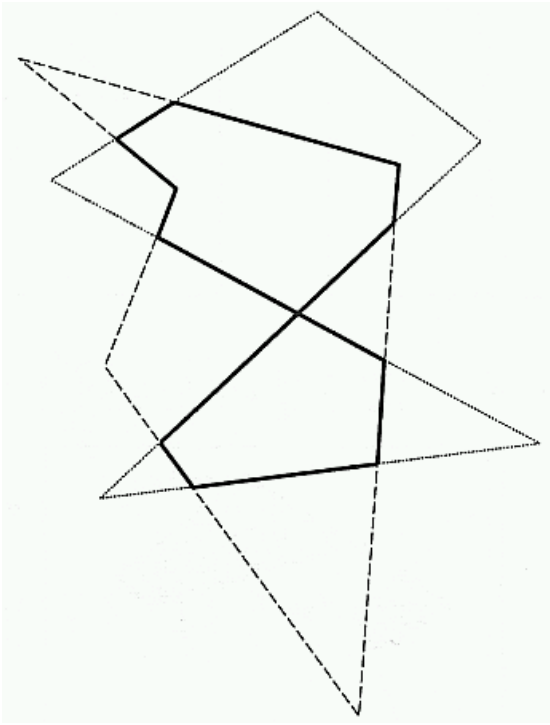
Greiner & Hormann



A in B



B in A



(A in B) ∪ (B in A)

3D Clipping against View Volume

- Requirements
  - Avoid unnecessary rasterization
  - Avoid overflow on transformation at fixed point !
- Clipping against viewing frustum
  - Enhanced Cohen-Sutherland with 6-bit outcode
  - After perspective division
    - $-1 < y < 1$
    - $-1 < x < 1$
    - $-1 < z < 0$
  - Clip against side planes of the viewing frustum
  - Works analogous with Liang-Barsky or Sutherland-Hodgeman

3D Clipping against View Volume

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Clipping in homogeneous coordinates
  - Avoid division by w
  - Inside test with a linear distance function (WEC)
    - Left: X/W > -1 ➔ W+X= $WEC_L(\underline{p}) > 0$
    - Top: Y/W < 1 ➔ W -Y= $WEC_T(\underline{p}) > 0$
    - Back: Z/W > -1 ➔ W+Z= $WEC_B(\underline{p}) > 0$
    - ...
  - Intersection point calculation (before homogenizing)
    - Test: $WEC_L(\underline{p}_a) > 0$ and $WEC_L(\underline{p}_e) < 0$
    - Calculation:

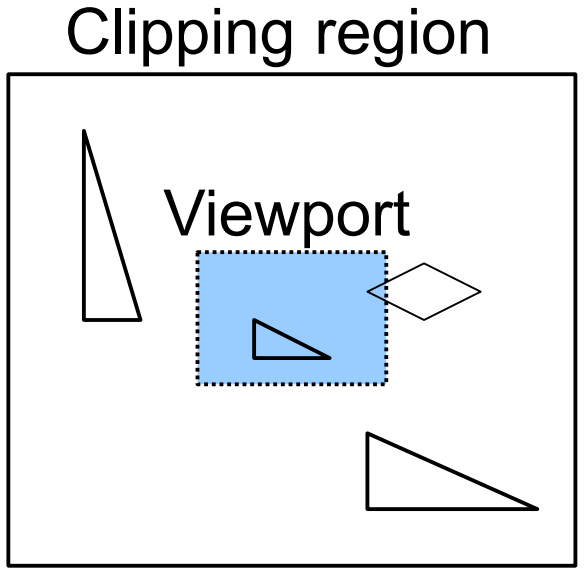
$$WEC(p_a + t(p_e - p_a)) = 0$$

$$W_a + t(W_e - W_a) + X_a + t(X_e - X_a) = 0$$

$$t = \frac{W_a + X_a}{(W_a + X_a) - (W_e + X_e)} = \frac{WEC_L(p_a)}{WEC_L(p_a) - WEC_L(p_e)}$$

Practical Implementations

- Combining clipping and scissoring
  - Clipping is expensive and should be avoided
    - Intersection calculation
    - Variable number of new points
  - Enlargement of clipping region
    - Larger than viewport, but
    - Still avoiding overflow due to fixed-point representation
  - Result
    - Less clipping
    - Applications should avoid drawing objects which are lying outside of the viewing frustum
    - Objects which are lying partially outside will be clipped implicitly during rasterization.



Wrap-Up

- Rasterization
  - Draw lines
  - Bresenham Algorithm
  - Draw polygons

- Clipping
  - Line clipping
  - Polygon clipping
  - Inside / Outside test
  - Window Edge coordinates