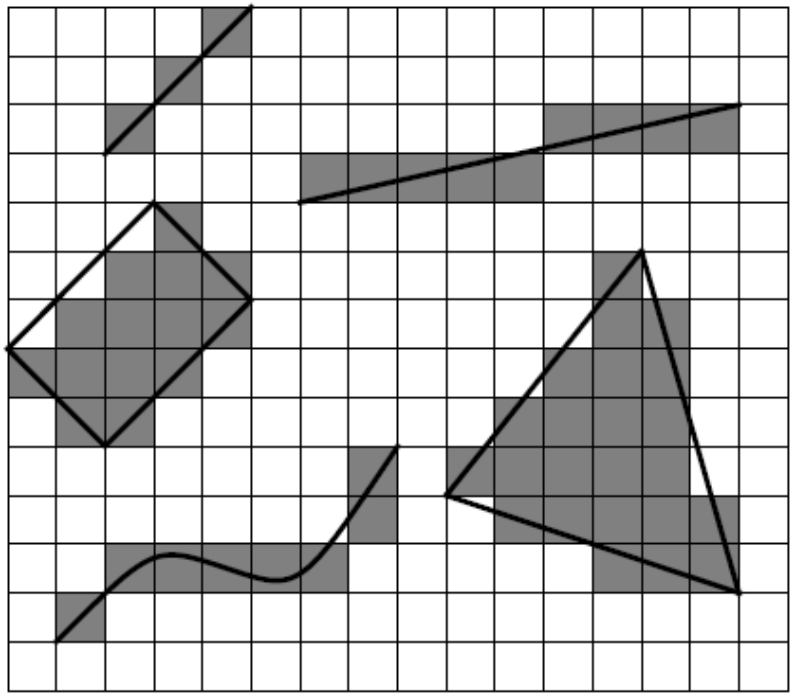
Rasterization, Depth Sorting and Culling

Rastrerzation



- How can we determine which pixels to fill?

Reading Material

- These slides
- OH 17-26, OH 65-79 and OH 281-282, by Magnus Bondesson
- You may also read chapter 7 of course book, Angel, ”Interactive Computer Graphics – A Top Down Approach”

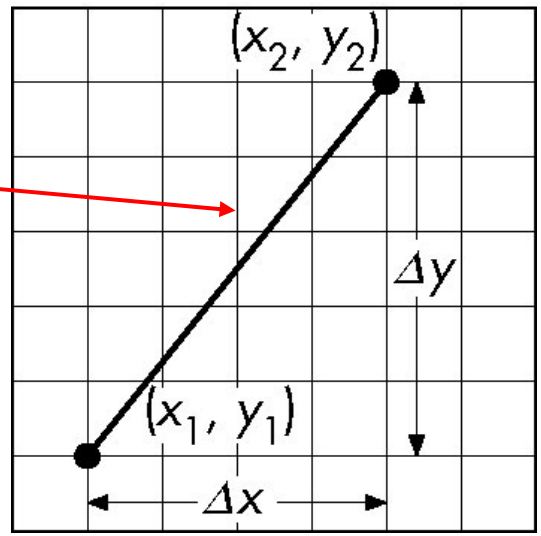
- Let’s start with Line Drawing Algorithms
  - DDA
  - Bresenham

Scan Conversion of Line Segments

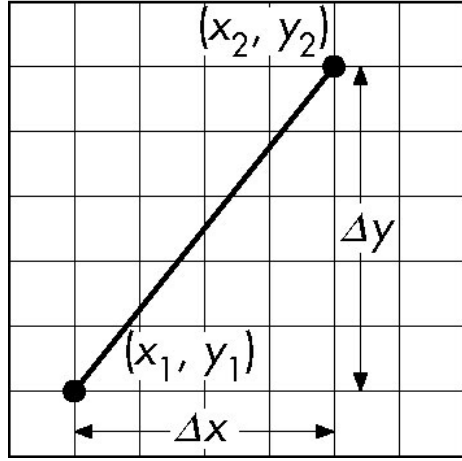
- Start with line segment in window coordinates with integer values for endpoints
- Assume implementation has a **`write_pixel`** function

y = kx + m

$$k = \frac{\Delta y}{\Delta x}$$



DDA Algorithm



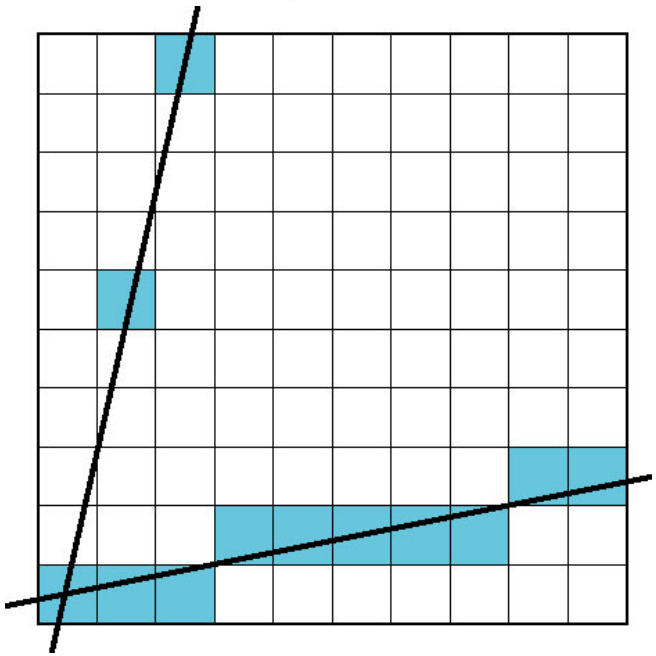
- Digital Differential Analyzer
  - DDA was a mechanical device for numerical solution of differential equations
  - Line $y=kx+m$ satisfies differential equation

    $dy/dx = k = \Delta y/\Delta x = y_2-y_1/x_2-x_1$
- Along scan line $\Delta x = 1$

```
y=y1;
For(x=x1; x<=x2,ix++) {
  write_pixel(x, round(y), line_color)
  y+=k;
}
```

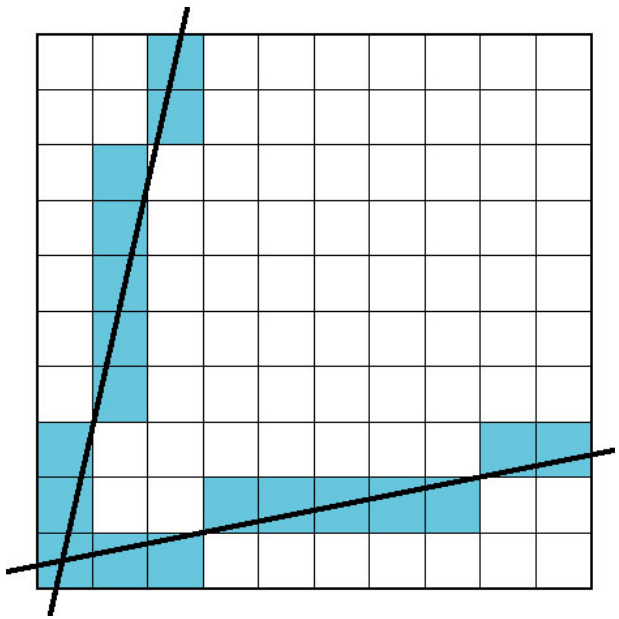
Problem

- DDA = for each x plot pixel at closest y
  - Problems for steep lines



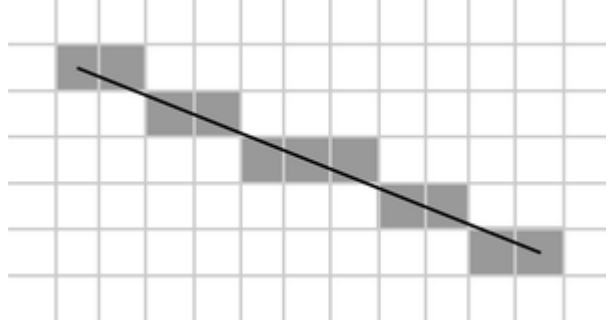
Using Symmetry

- Use for $1 \geq k \geq 0$
- For k > 1, swap role of x and y
  - For each y, plot closest x



- The problem with DDA is that it uses floats which was slow in the old days
- Bresenhams algorithm only uses integers

Bresenham's line drawing algorithm

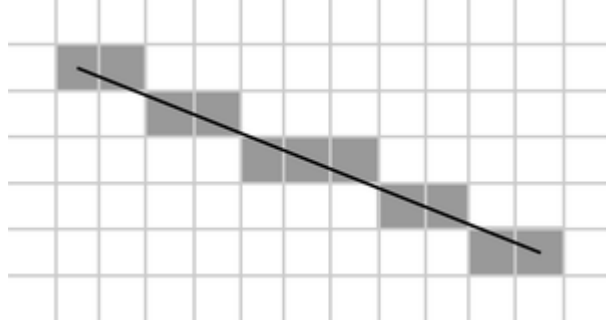


- The line is drawn between two points (x_0, y_0) and (x_1, y_1)
- Slope $k = \frac{(y_1 - y_0)}{(x_1 - x_0)}$ ($y = kx + m$)
- Each time we step 1 in x-direction, we should increment *y* with *k.* Otherwise the error in y increases with *k.*
- If the error surpasses 0.5, the line has become closer to the next *y*-value, so we add 1 to *y* simultaneously decreasing the error by 1

```
function line(x0, x1, y0, y1)
    int deltax := abs(x1 - x0)
    int deltay := abs(y1 - y0)
    real error := 0
    real deltaerr := deltay / deltax
    int y := y0
    for x from x0 to x1
        plot(x,y)
        error := error + deltaerr
        if error ≥ 0.5
            y := y + 1
            error := error - 1.0
```

See also
http://en.wikipedia.org/wiki/Bresenham's_line_algorithm

Bresenham's line drawing algorithm



- Now, convert algorithm to only using integer computations'
- The trick we use is to multiply all the fractional numbers above by (x_1, x_0), which enables us to express them as integers.
- The only problem remaining is the constant 0.5—to deal with this, we multiply both sides of the inequality by 2

Old float version:

```
function line(x0, x1, y0, y1)
    int deltax := abs(x1 - x0)
    int deltay := abs(y1 - y0)
    real error := 0
    real deltaerr := deltay / deltax
    int y := y0
    for x from x0 to x1
        plot(x,y)
        error := error + deltaerr
        if error ≥ 0.5
            y := y + 1
            error := error - 1.0
```

New integer version:

```
function line(x0, x1, y0, y1)
    int deltax := abs(x1 - x0)
    int deltay := abs(y1 - y0)
    real error := 0
    real deltaerr := deltay
    int y := y0
    for x from x0 to x1
        plot(x,y)
        error := error + deltaerr
        if 2*error ≥ deltax
            y := y + 1
            error := error - deltax
```

Complete Bresenham's line drawing algorithm

```
function line(x0, x1, y0, y1)
    boolean steep := abs(y1 - y0) > abs(x1 - x0)
    if steep then
        swap(x0, y0)
        swap(x1, y1)
    if x0 > x1 then
        swap(x0, x1)
        swap(y0, y1)
    int deltax := x1 - x0
    int deltay := abs(y1 - y0)
    int error := 0
    int ystep
    int y := y0
    if y0 < y1 then ystep := 1 else ystep := -1
    for x from x0 to x1
        if steep then plot(y,x) else plot(x,y)
        error := error + deltay
        if 2×error ≥ deltax
            y := y + ystep
            error := error - deltax
```

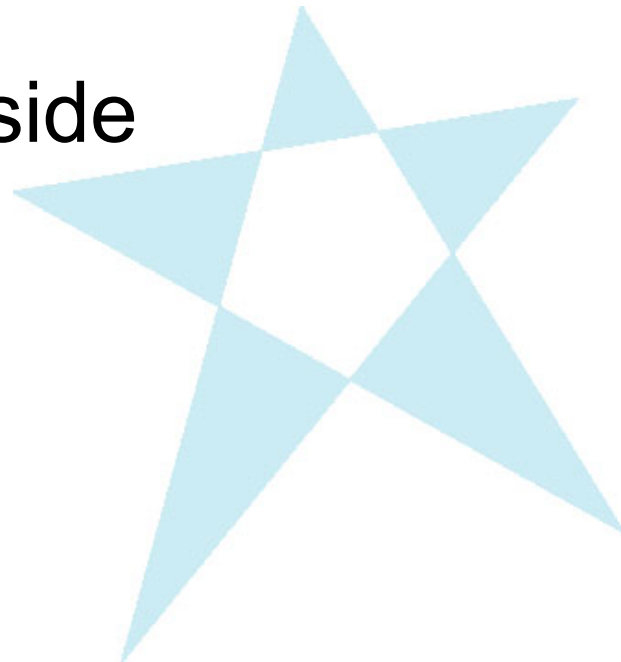
The first case is allowing us to draw lines that still slope downwards, but head in the opposite direction. I.e., swapping the initial points if x0 > x1.

To draw lines that go up, we check if y0 >= y1; if so, we step y by -1 instead of 1.

To be able to draw lines with a slope less than one, we take advantage of the fact that a steep line can be reflected across the line y=x to obtain a line with a small slope. The effect is to switch the x and y variables.

Polygon Scan Conversion

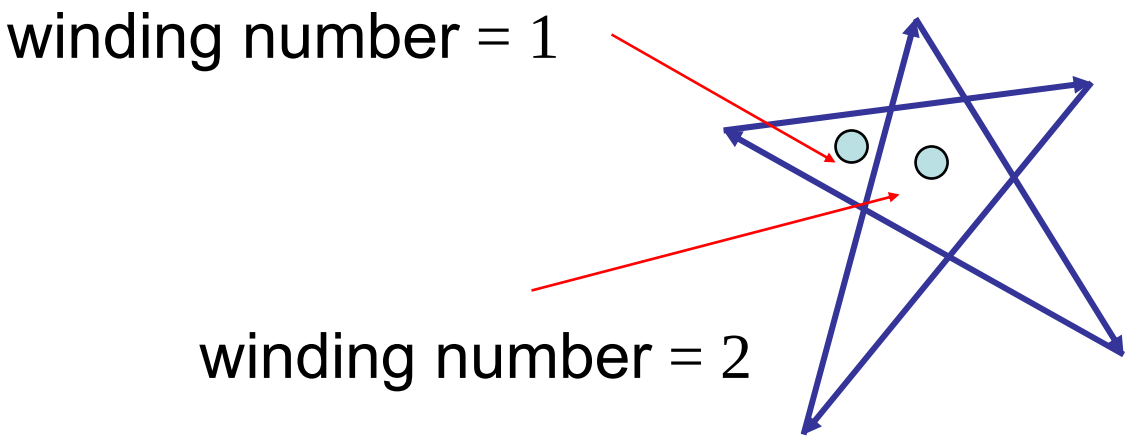
- Scan Conversion = Fill
- How to tell inside from outside
  - –Convex easy
  - –Nonsimple difficult
  - –Odd even test
    - Count edge crossings
  - –Winding number



odd-even fill

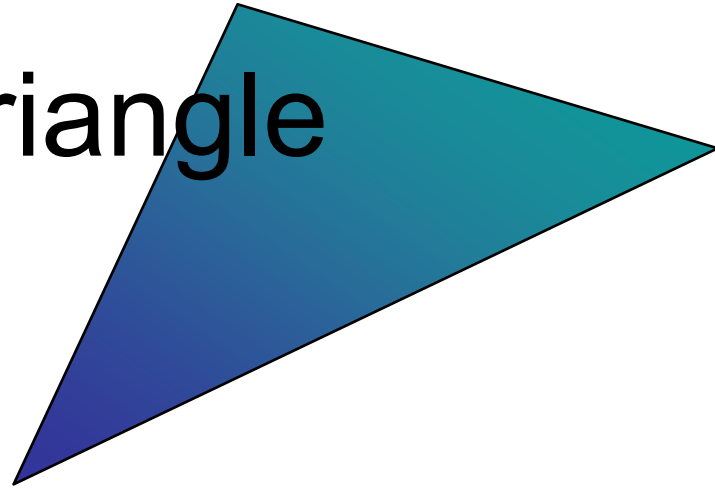
Winding Number

- Count clockwise encirclements of point



- Alternate definition of inside: inside if winding number $\neq 0$

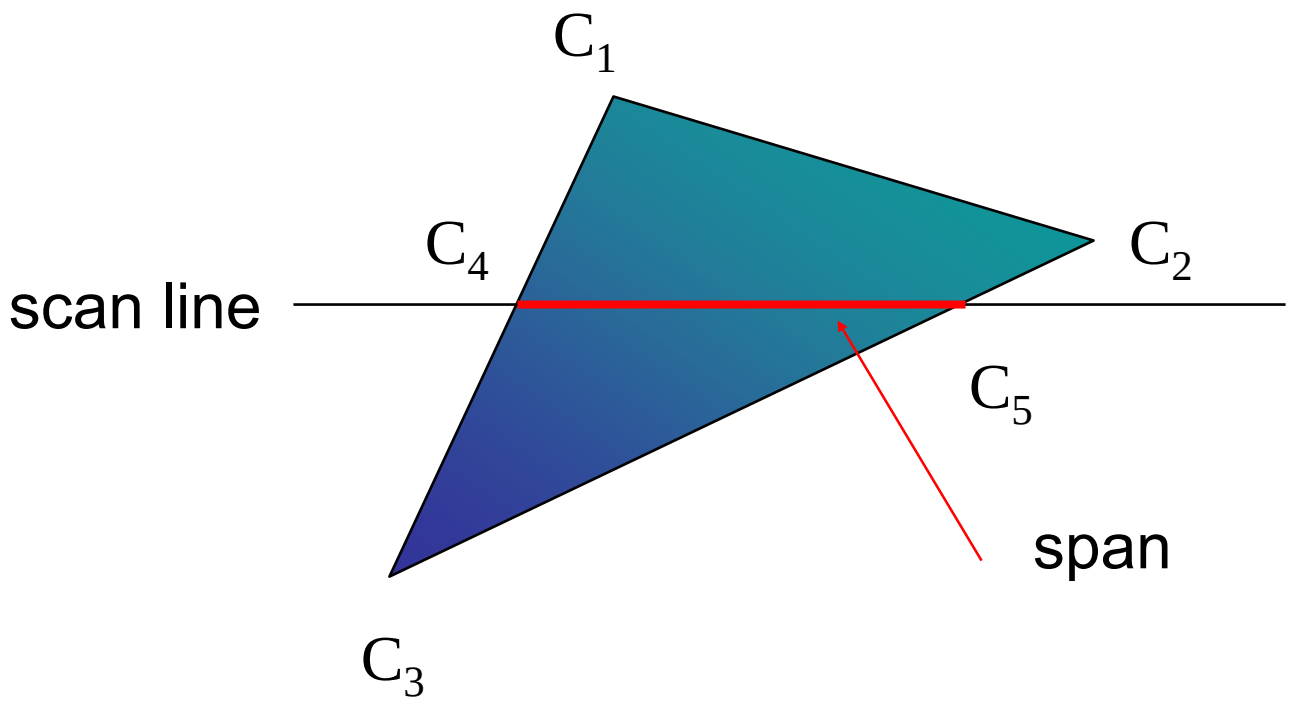
Rasterizing a Triangle



- Fill at end of pipeline
  - –Convex Polygons only
  - –Nonconvex polygons assumed to have been tessellated
  - –Shades (colors) have been computed for vertices (Gouraud shading)
  - –Combine with z-buffer algorithm
    - March across scan lines interpolating shades
    - Incremental work small

Using Interpolation

C_1 C_2 C_3 specified by **`glColor`** or by vertex shading
C_4 determined by interpolating between C_1 and C_2
C_5 determined by interpolating between C_2 and C_3
interpolate between C_4 and C_5 along span



Flood Fill

- Fill can be done recursively if we know a seed point located inside (WHITE)
- Scan convert edges into buffer in edge/inside color (BLACK)

```
flood_fill(int x, int y) {
    if(read_pixel(x,y)= = WHITE) {
       write_pixel(x,y,BLACK);
       flood_fill(x-1, y);
       flood_fill(x+1, y);
       flood_fill(x, y+1);
       flood_fill(x, y-1);
}   }
```

Scan Line Fill

Set active edges to AB and AC

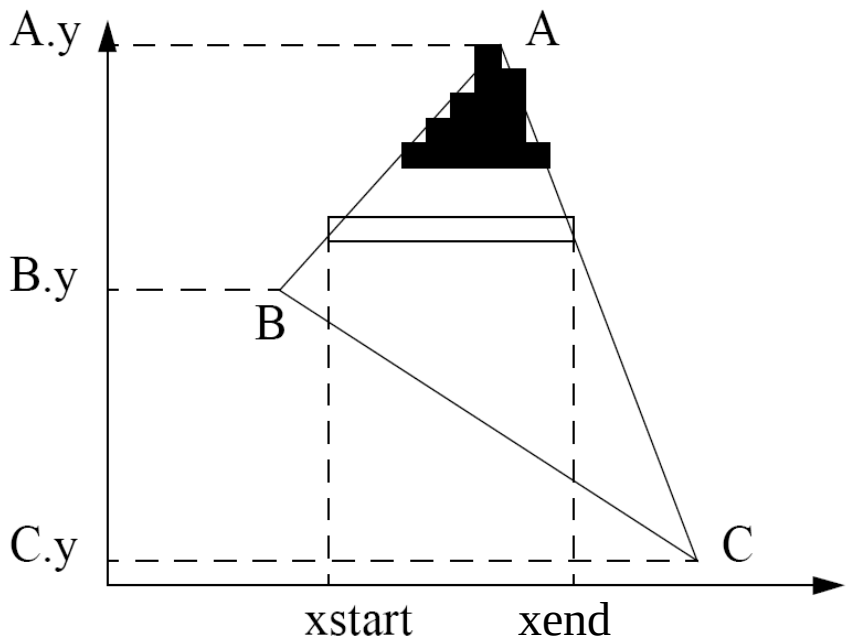
For y = A.y, A.y-1,...,C.y

If y=B.y $\rightarrow$ exchange AB with BC

Compute xstart and xend.
Interpolate color, depth, texcoords etc for points (xstart,y) and (xend,y)

For x = xstart, xstart+1, ...,xend

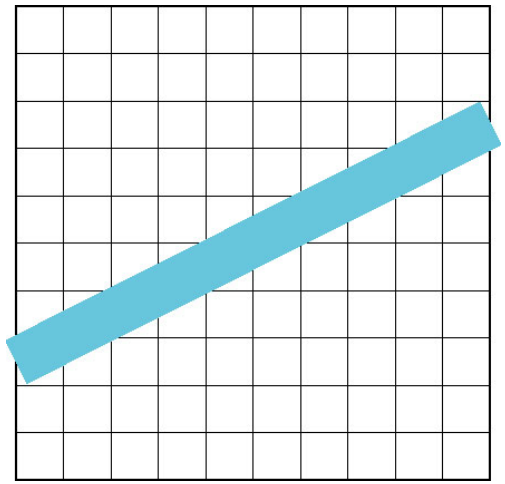
Compute color, depth etc for (x,y) using interpolation.



This is the modern way to rasterize a triangle

Aliasing

- Ideal rasterized line should be 1 pixel wide

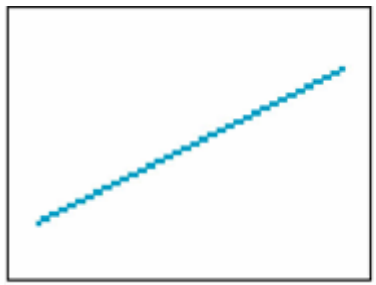


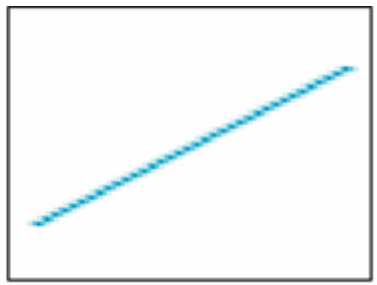
- Choosing best y for each x (or visa versa) produces aliased raster lines

Antialiasing by Area Averaging

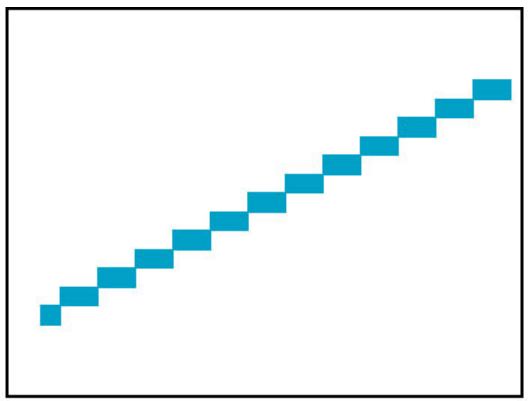
- Color multiple pixels for each x depending on coverage by ideal line

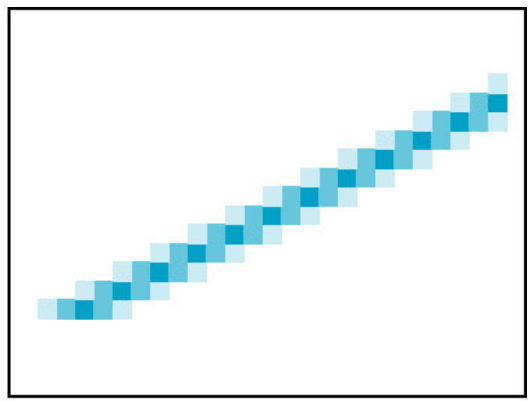
original





antialiased

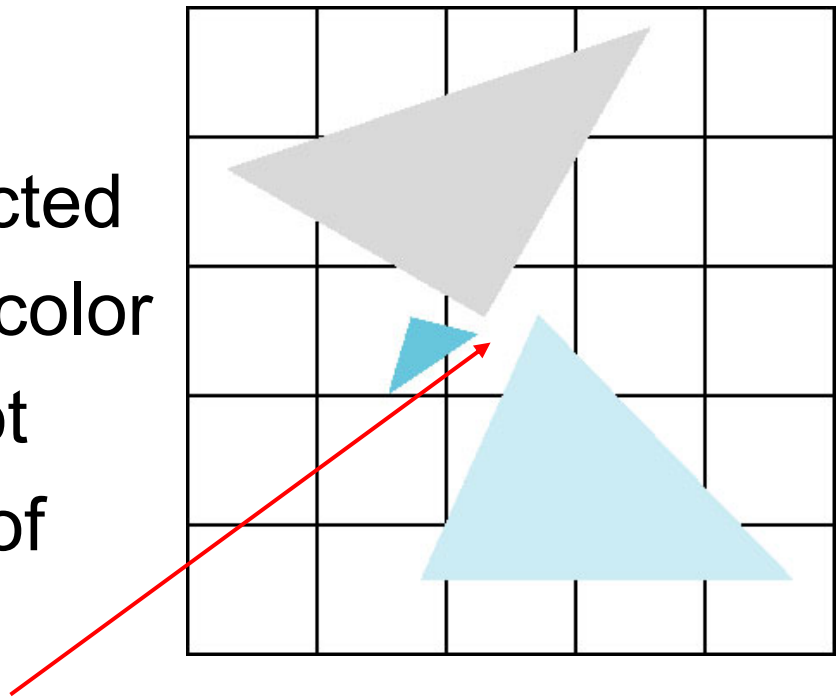




magnified

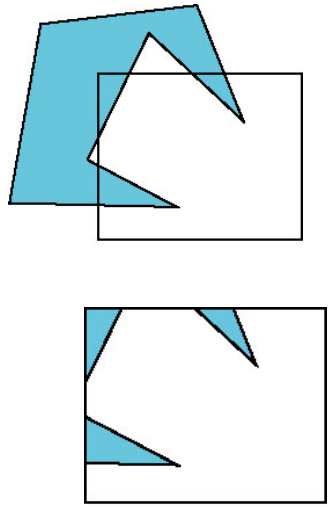
Polygon Aliasing

- Aliasing problems can be serious for polygons
  - –Jaggedness of edges
  - –Small polygons neglected
  - –Need compositing so color of one polygon does not totally determine color of pixel



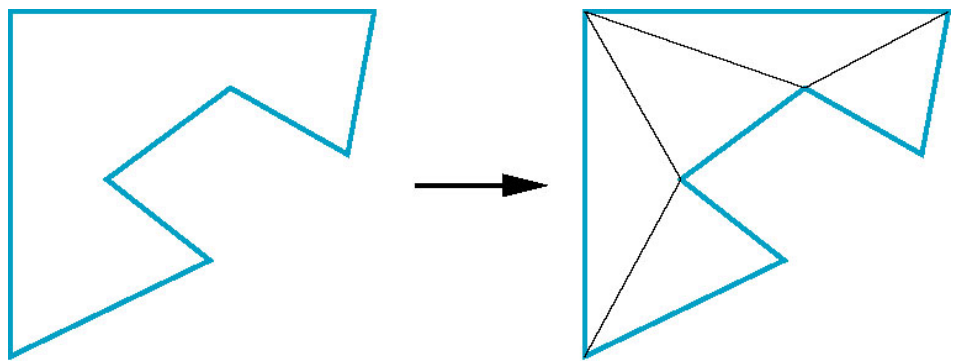
All three polygons should contribute to color

Polygon Clipping

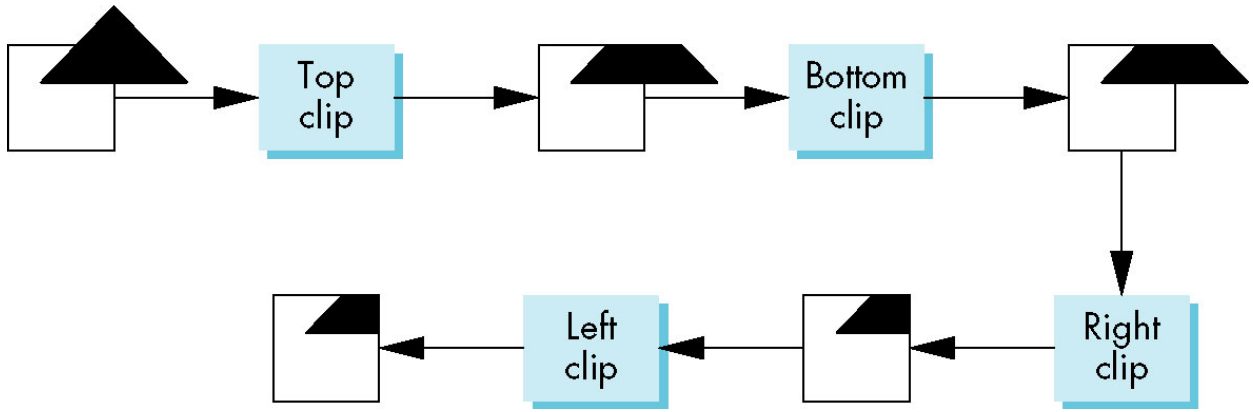


- Not as simple as line segment clipping
  - Clipping a line segment yields at most one line segment
  - Clipping a polygon can yield multiple polygons
  - However, clipping a convex polygon can yield at most one other polygon

- One strategy is to replace nonconvex (*concave*) polygons with a set of triangular polygons (a *tessellation*)
- Also makes fill easier
- Tessellation code in GLU library



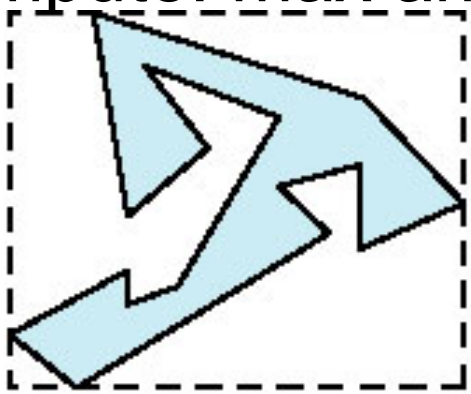
Pipeline Clipping of Polygons



- Three dimensions: add front and back clippers
- Strategy used in SGI Geometry Engine
- Small increase in latency

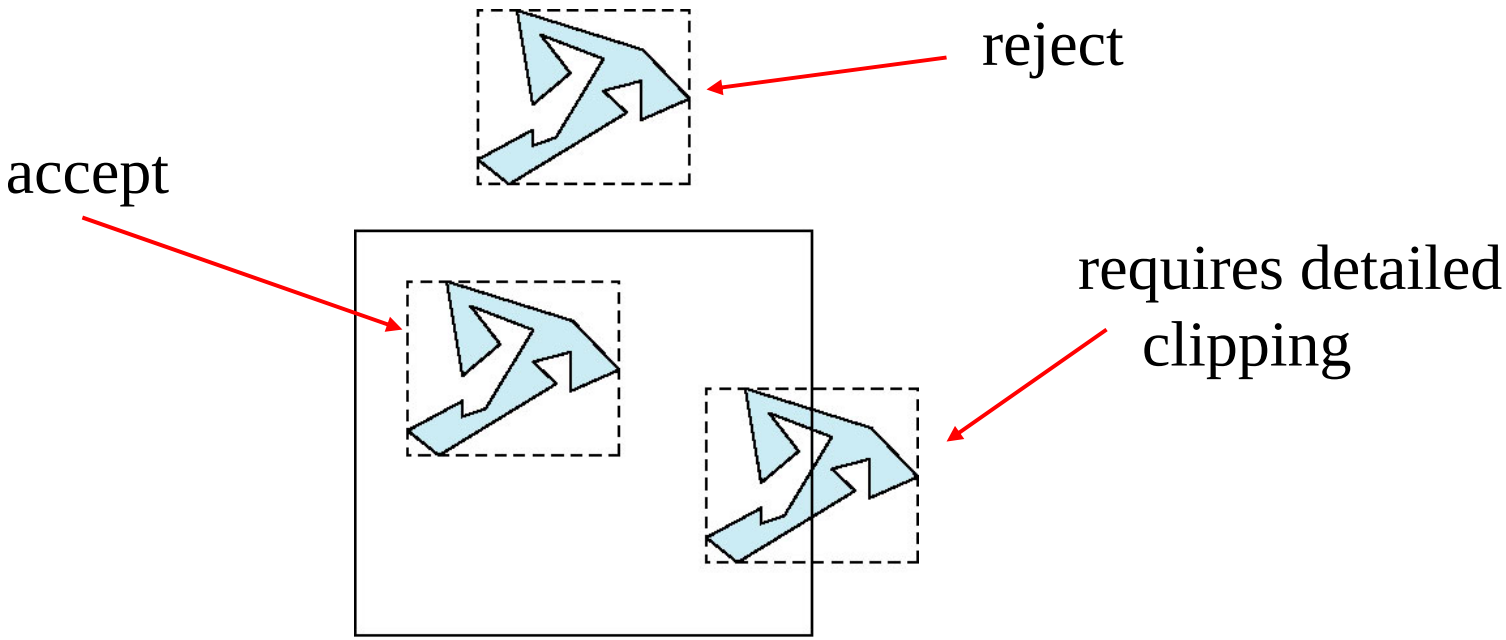
Bounding Boxes

- Rather than doing clipping on a complex polygon, we can use an *axis-aligned bounding box* or *extent*
  - –Smallest rectangle aligned with axes that encloses the polygon
  - –Simple to compute: max and min of x and y



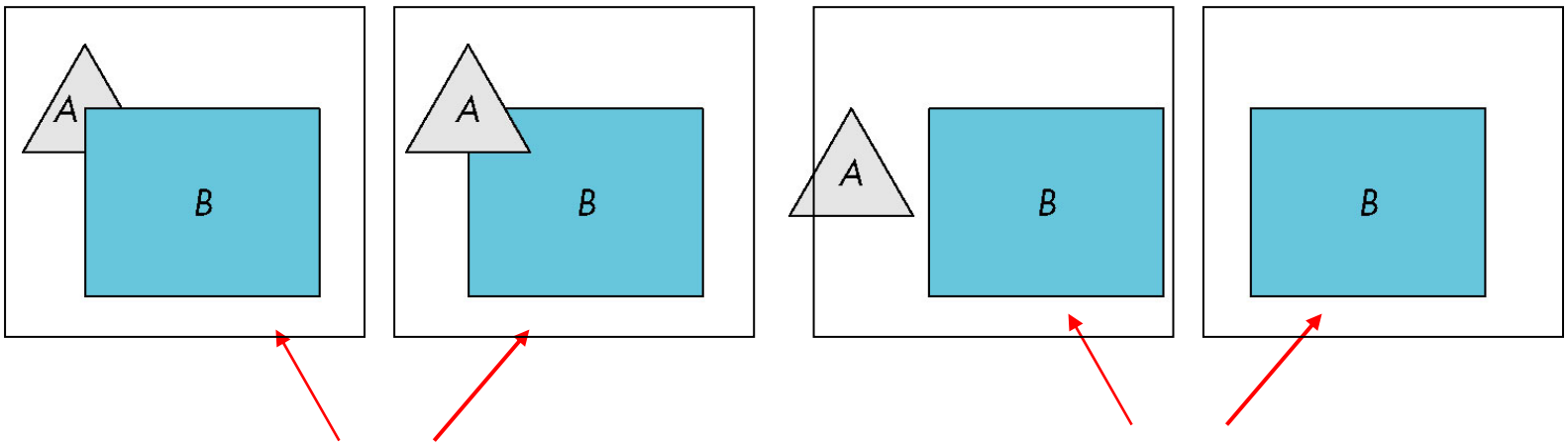
Bounding boxes

Can usually determine accept/reject based only on bounding box



Hidden Surface Removal

- Object-space approach: use pairwise testing between polygons (objects)



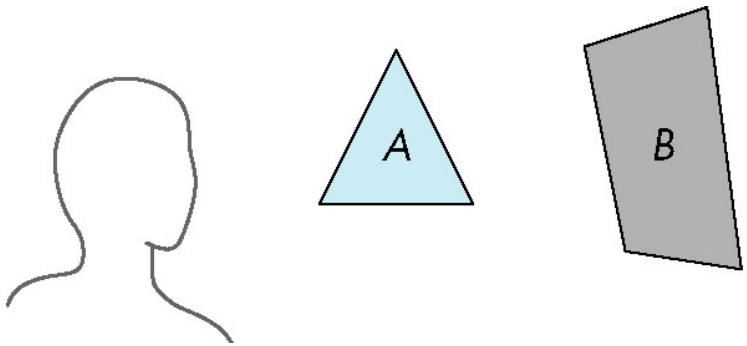
partially obscuring

can draw independently

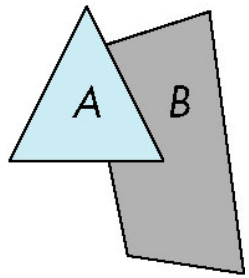
- Worst case complexity O(n log n) for n polygons

Painter's Algorithm

- Render polygons a back to front order so that polygons behind others are simply painted over



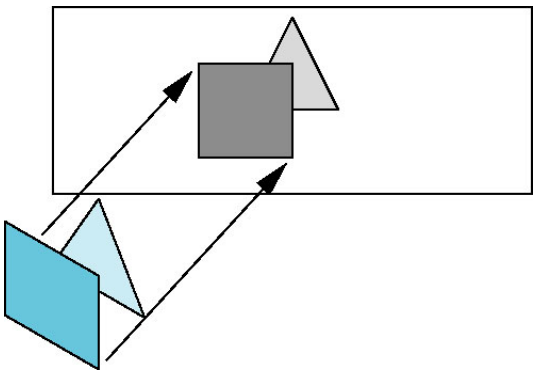
B behind A as seen by viewer



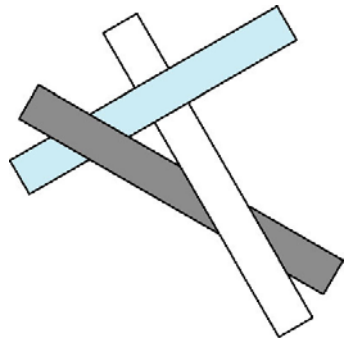
Fill B then A

- Requires ordering of polygons first
  - –O(n log n) calculation for ordering
  - –Not every polygon is either in front or behind all other polygons

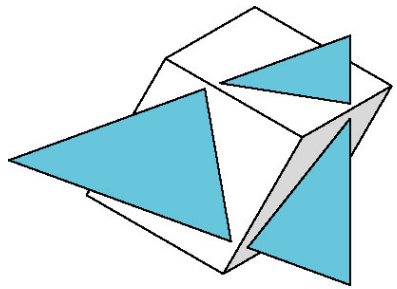
Hard Cases



Overlap in x, y, z,
but one object is fully on
one side of the other



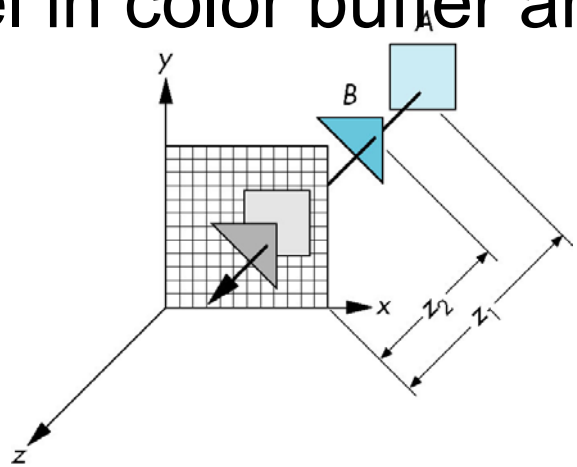
cyclic overlap



penetration

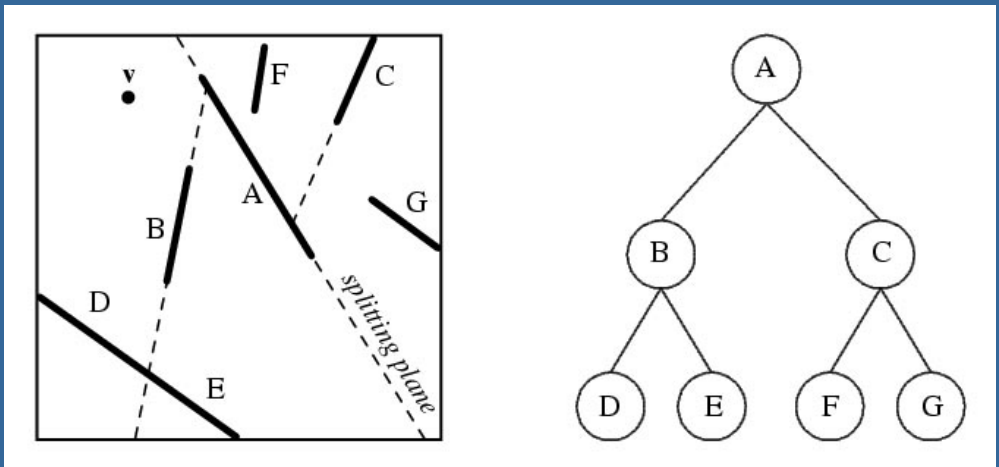
z-Buffer Algorithm

- Use a buffer called the z or depth buffer to store the depth of the closest object at each pixel found so far
- As we render each polygon, compare the depth of each pixel to depth in z buffer
- If less, place shade of pixel in color buffer and update z buffer



Polygon-aligned BSP tree

- Used for visibility and occlusion/depth testing (BSP=Binary Space Partitioning)
- Allows exact sorting
  - Each node stores:
    - Polygon (triangle)
    - The splitting plane (defined by the triangle)
    - Front and back subtree



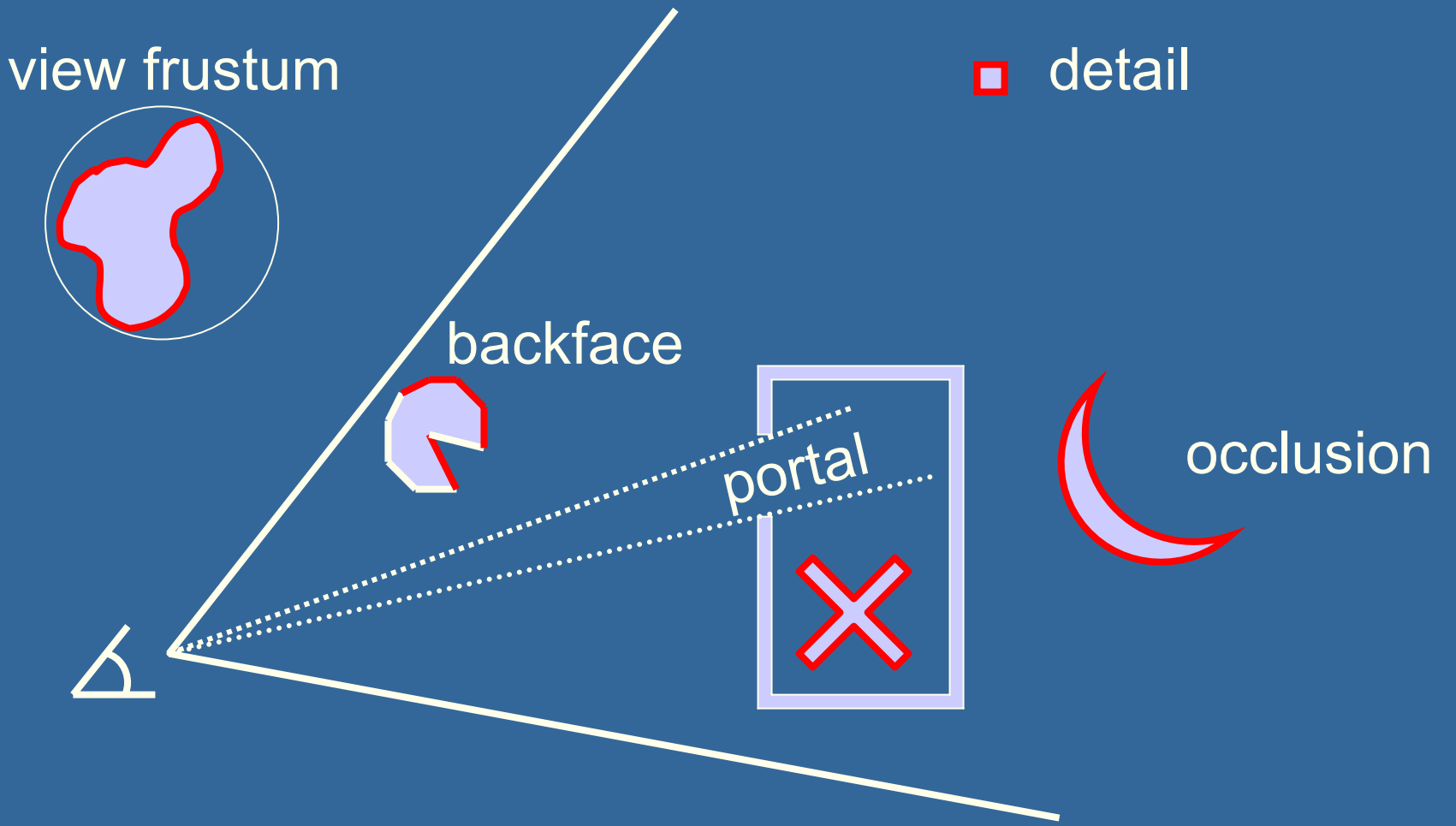
Algorithm for BSP trees

```
Tree SkapaBSP(PolygonLista L) {
    Om L tom returnera ett tomt träd;
    Annars: Välj en polygon P i listan.
    Bilda en lista B med de polygoner som ligger bakom
    P och en annan H med övriga. Returnera ett träd med
    P som rot och SkapaBSP(B) och SkapaBSP(H) som
    vänsterbarn respektive högerbarn.
}
```

Uppritningsteget (kolla även om trädet tomt! Fick ej plats i koden):

```
void RitaBSP(Tree t) {
    Om observatören hitom roten i t:
        RitaBSP(t:s vänsterbarn);
        Rita polygonen i t:s rot;
        RitaBSP(t:s högerbarn);
    Annars:
        RitaBSP(t:s högerbarn);
        Rita polygonen i t:s rot;
        RitaBSP(t:s vänsterbarn);
}
```

Different culling techniques
(red objects are skipped)

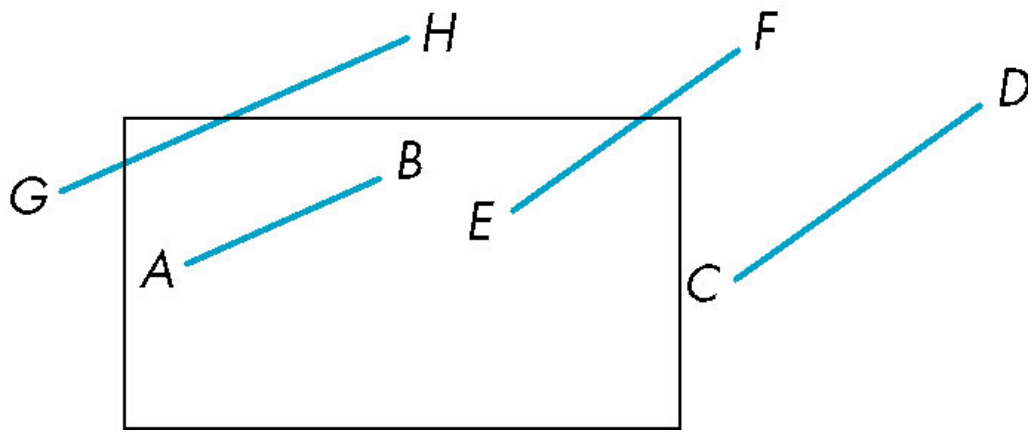


Bonus Material

Following slides are not part of the course but could be interesting for the curious students

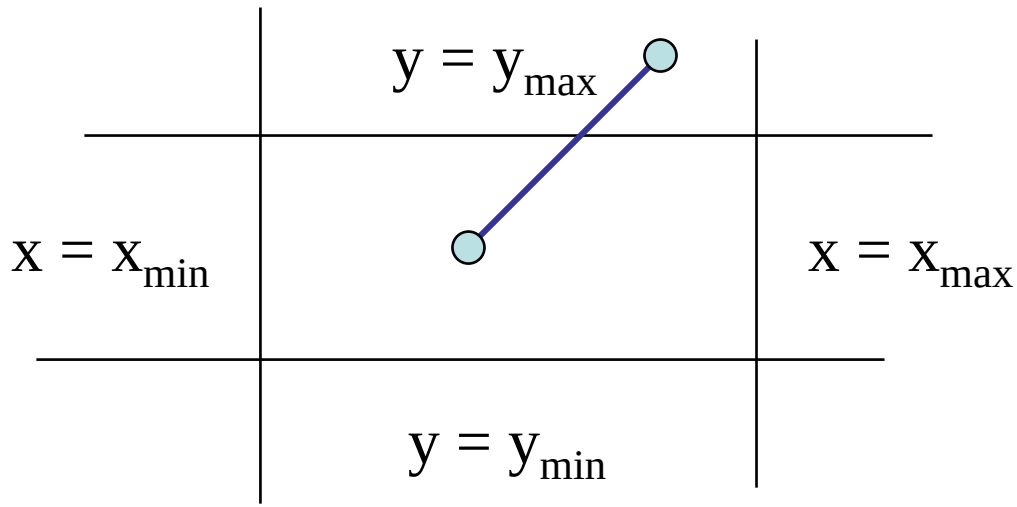
Clipping 2D Line Segments

- Brute force approach: compute intersections with all sides of clipping window
  - Inefficient: one division per intersection



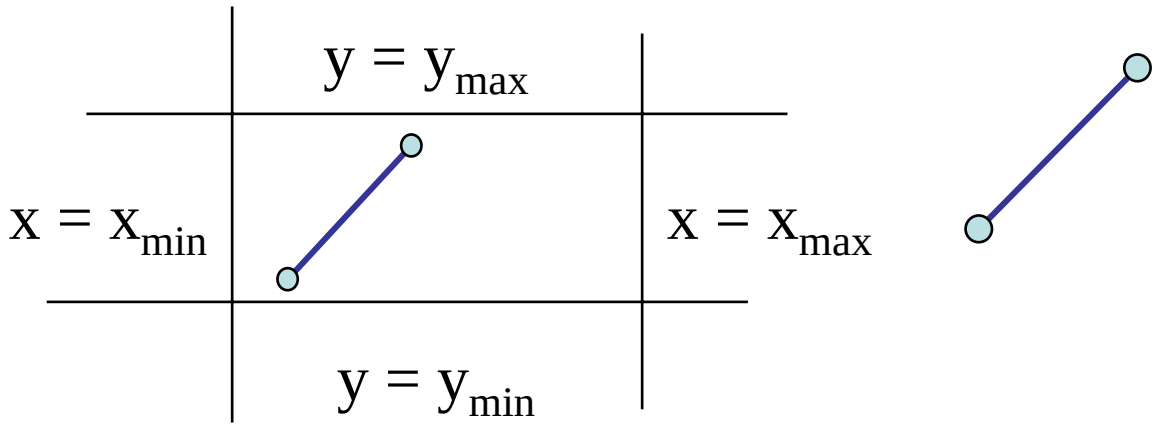
Cohen-Sutherland Algorithm

- Idea: eliminate as many cases as possible without computing intersections
- Start with four lines that determine the sides of the clipping window



The Cases

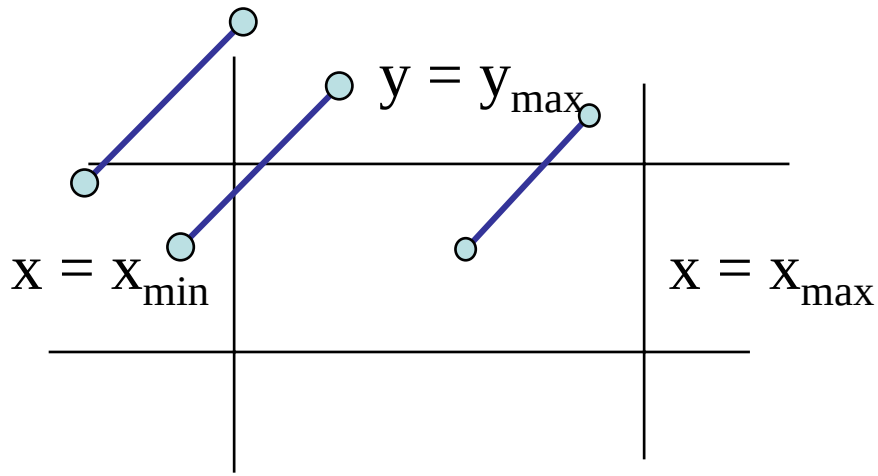
- Case 1: both endpoints of line segment inside all four lines
  - Draw (accept) line segment as is



- Case 2: both endpoints outside all lines and on same side of a line
  - Discard (reject) the line segment

The Cases

- Case 3: One endpoint inside, one outside
  - –Must do at least one intersection
- Case 4: Both outside
  - –May have part inside
  - –Must do at least one intersection

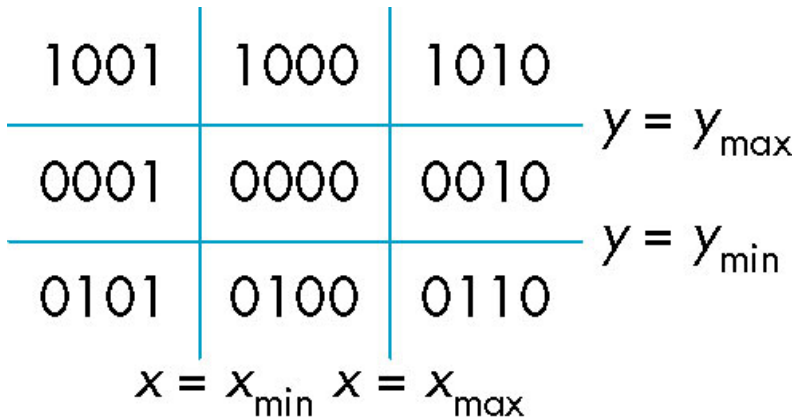


Defining Outcodes

- For each endpoint, define an outcode

$$b_0b_1b_2b_3$$

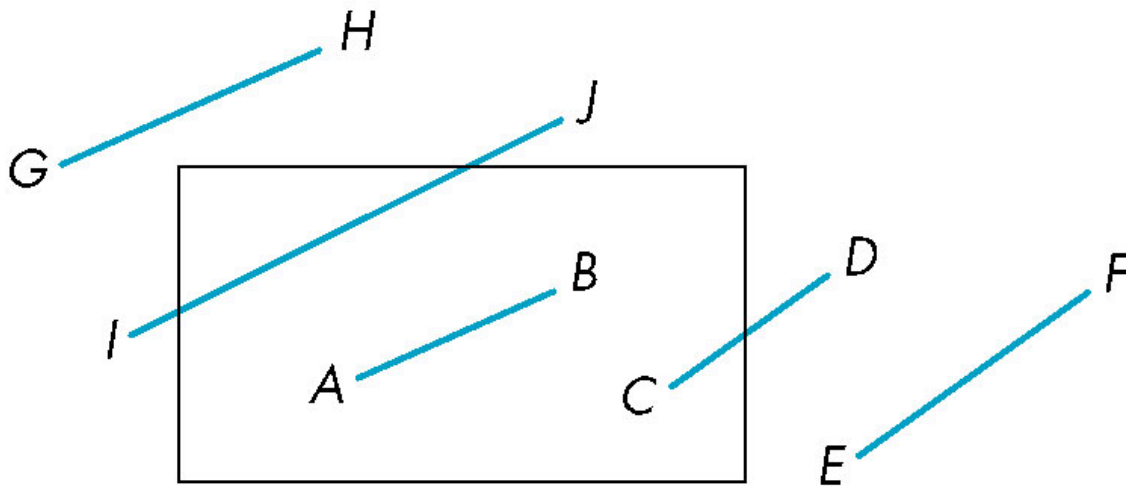
$b_0 = 1$ if $y > y_{max}$, 0 otherwise
$b_1 = 1$ if $y < y_{min}$, 0 otherwise
$b_2 = 1$ if $x > x_{max}$, 0 otherwise
$b_3 = 1$ if $x < x_{min}$, 0 otherwise



- Outcodes divide space into 9 regions
- Computation of outcode requires at most 4 subtractions

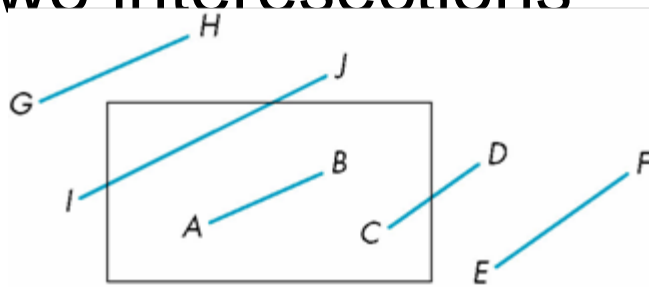
Using Outcodes

- Consider the 5 cases below
- AB: outcode(A) = outcode(B) = 0
  - –Accept line segment



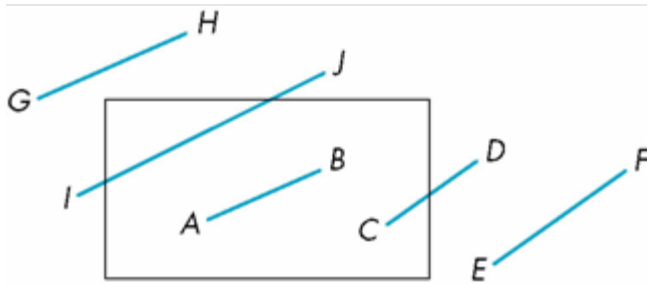
Using Outcodes

- CD: outcode (C) = 0, outcode(D) ≠ 0
  - –Compute intersection
  - –Location of 1 in outcode(D) determines which edge to intersect with
  - –Note if there were a segment from A to a point in a region with 2 ones in outcode, we might have to do two interesections



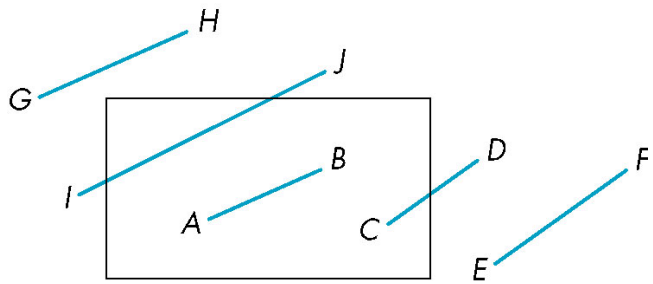
Using Outcodes

- EF: outcode(E) logically ANDed with outcode(F) (bitwise) ≠ 0
  - –Both outcodes have a 1 bit in the same place
  - –Line segment is outside of corresponding side of clipping window
  - –reject



Using Outcodes

- GH and IJ: same outcodes, neither zero but logical AND yields zero
- Shorten line segment by intersecting with one of sides of window
- Compute outcode of intersection (new endpoint of shortened line segment)
- Reexecute algorithm

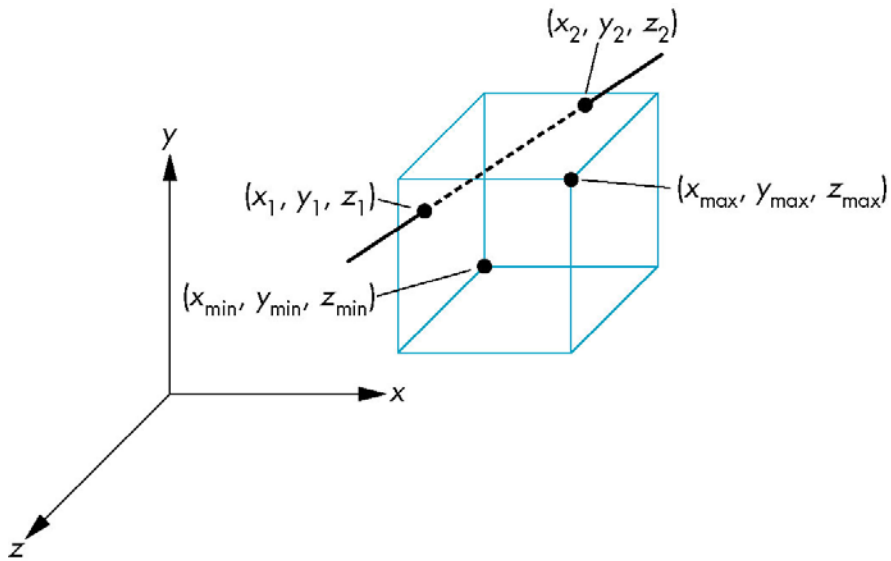


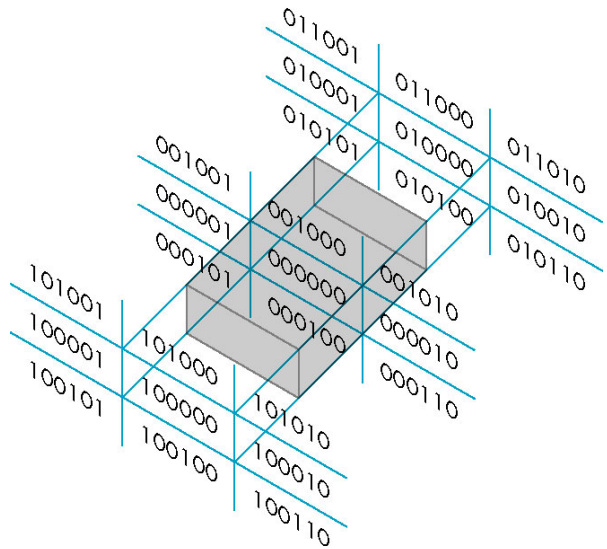
Efficiency

- In many applications, the clipping window is small relative to the size of the entire data base
  - –Most line segments are outside one or more side of the window and can be eliminated based on their outcodes
- Inefficiency when code has to be reexecuted for line segments that must be shortened in more than one step

Cohen Sutherland in 3D

- Use 6-bit outcodes
- When needed, clip line segment against planes

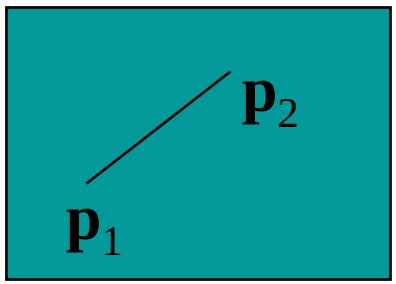




Liang-Barsky Clipping

- Consider the parametric form of a line segment

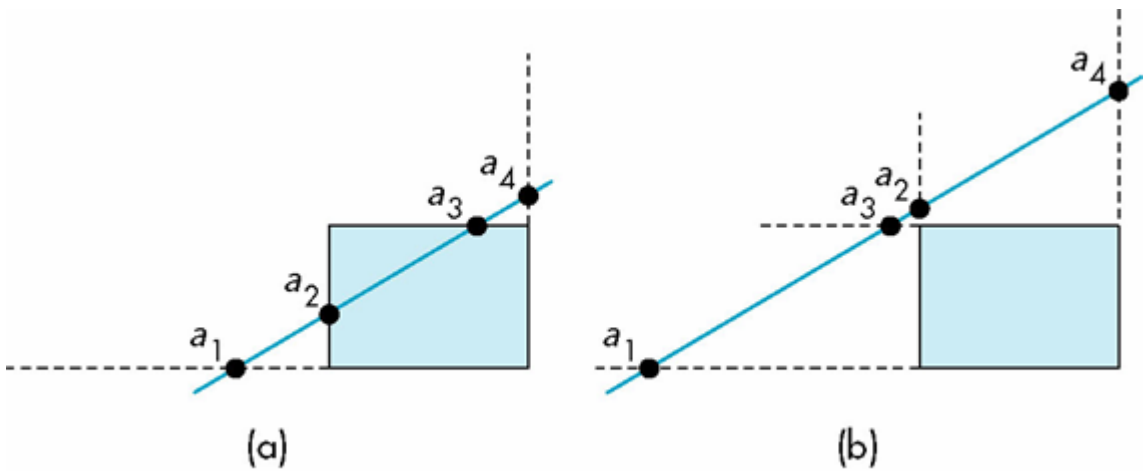
$$\mathbf{p}(\alpha) = (1-\alpha)\mathbf{p}_1 + \alpha\mathbf{p}_2 \quad 1 \geq \alpha \geq 0$$



- We can distinguish between the cases by looking at the ordering of the values of α where the line determined by the line segment crosses the lines that determine the window

Liang-Barsky Clipping

- In (a): $\alpha_4 > \alpha_3 > \alpha_2 > \alpha_1$
  - –Intersect right, top, left, bottom: shorten
- In (b): $\alpha_4 > \alpha_2 > \alpha_3 > \alpha_1$
  - –Intersect right, left, top, bottom: reject



Advantages

- Can accept/reject as easily as with Cohen-Sutherland
- Using values of α, we do not have to use algorithm recursively as with C-S
- Extends to 3D