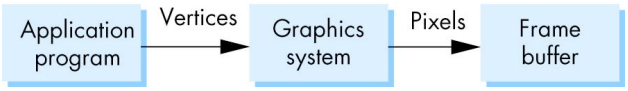
Universität Bielefeld

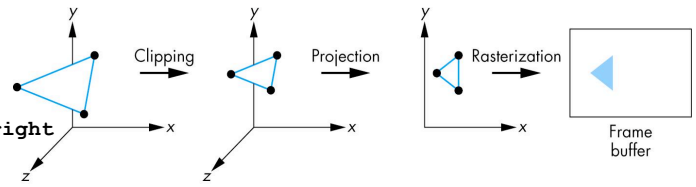
Realtime 3D Computer Graphics Virtual Reality

From Vertices to Fragments

Overview

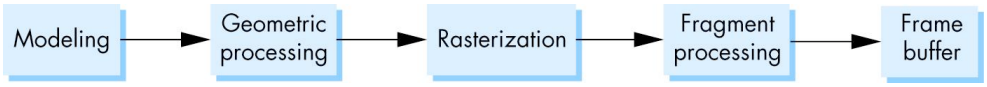


- Overall goal recapitulation:
  - Input: World description, e.g., set of vertices and states for objects, attributes, camera,…
  - Output: Array of colored pixels in framebuffer.
- After transformation and projection (with possible normalization)
  1. decide visibility of objects:
     - which objects will be seen due to frustum and output window:
       - **Culling:** in object or world space
       - **Clipping:** in canonical view or image space
     - which objects will be in front of others (depth-sorting):
       - **Occlusion culling**, **depth sort:** in object or world space
       - **z-buffer algorithm, w-buffer algorithm:** in canonical view or image space
  2. draw the remaining primitives; map from continuous space to discrete space
     - **Scan conversion, rasterization**
- Two different possibilities:

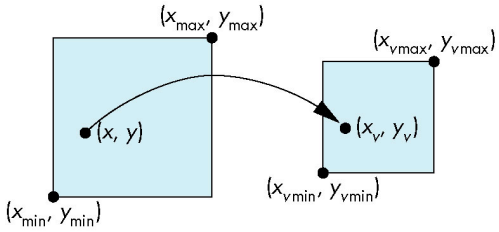



```
for(each_object) render(object); //see right
for(each_pixel) assign_a_color(pixel);
```

Tasks



- Modelling
- Geometry processing
  - Which objects appear on the screen
  - Assign color shades to vertices
  - Primitive assembly and clipping
  - Perspective division
- Rasterization
  - Maps from NDC* to window coords (see right):
  - **Lines**: Which fragments should be used to approximate
  - **Polygons**: Which pixels are inside
  - Assign color to fragments
- Fragment processing
  - Combine with image path (texturing)
  - Blending
  - Anti-aliasing



$$x_v = x_{v\min} + \frac{x+1.0}{2.0}\left(x_{v\max} - x_{v\min}\right)$$

$$y_v = y_{v\min} + \frac{y+1.0}{2.0}\left(y_{v\max} - y_{v\min}\right)$$

$$z_v = z_{v\min} + \frac{z+1.0}{2.0}\left(z_{v\max} - z_{v\min}\right)$$

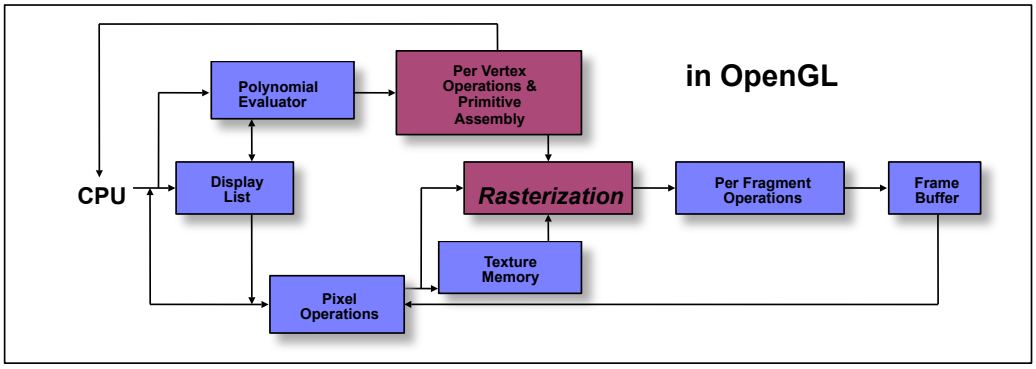
*NDC: All vertices lie in the cube with $-w \le a \le w, a \in \{x, y, z\}$

Realtime 3D Computer Graphics Virtual Reality

Clipping

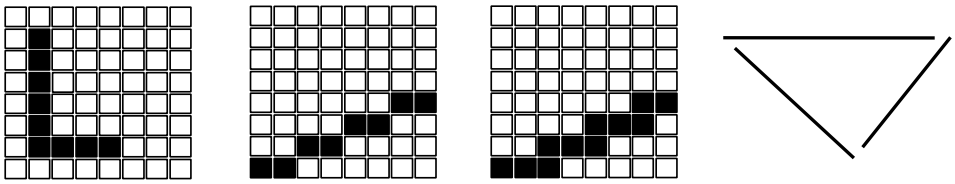
Line processing

- Lines:
  - Most common 2D primitive - done 100s or 1000s of times each frame, even 3D wireframes are eventually 2D lines!
  - Lines are *compatible* with vector displays but nowadays most displays are raster displays. Any render stage before viz might need discretization.
  - Optimized algorithms contain numerous tricks/techniques that help in designing more advanced algorithms for line processing.



Line Requirements

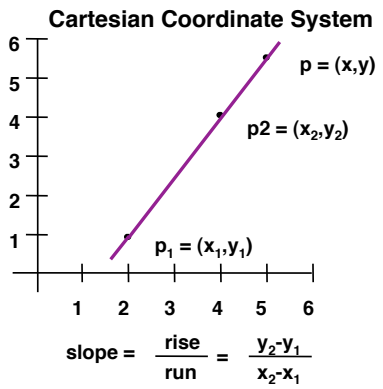
- Must compute integer coordinates of pixels which lie on or near a line or circle.
- Pixel level algorithms are invoked hundreds or thousands of times when an image is created or modified – must be fast!
- Lines must create visually satisfactory images.
  - Lines should appear straight
  - Lines should terminate accurately
  - Lines should have constant density
- Line algorithm should always be defined.



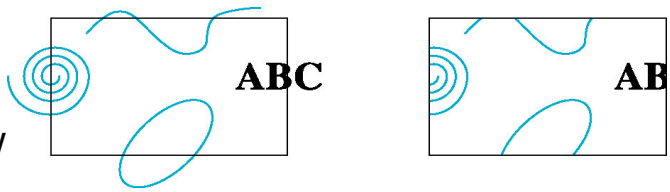
Basic Math Review

- **Point-slope formula for a Line:**
  - Given two points (x_1,y_1), (x_2, y_2)
  - Consider a third point on the line:
    - $p = (x,y)$
    - Slope $= (y_2 - y_1)/(x_2 - x_1)$
      $= (y - y_1)/(x - x_1)$
  - Solving For y
    - $y = [(y_2-y_1)/(x_2-x_1)]*(x-x_1)+ y_1$
  - or, plug in the point (0, b) to get the **Slope-intercept form:**
    - **$y = mx + b$**
- **Length of line segment between p_1 and p_2:**
  - $l = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
- **Midpoint of a line segment between p_1 and p_3:**
  - $p_2 = ((x_1+x_3)/2 , (y_1+y_3)/2)$
- Two lines are **perpendicular** iff
  - 1) $m_1 = -1/m_2$
  - 2) Cosine of the angle between them is 0.

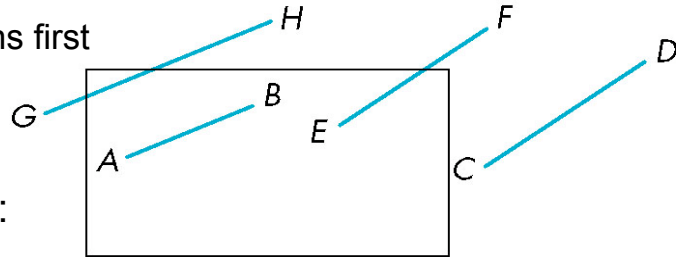
- **Parametric form**
  - Given points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$
    - $x = x_1 + \alpha(x_2-x_1)$, $y = y_1 + \alpha(y_2-y_1)$
    - ➢ $p(\alpha)=(1- \alpha)p_1+ \alpha p_2$
      α is called the parameter. When $\alpha = 0$ we get (x_1,y_1), $\alpha = 1$ we get (x_2,y_2)
  - As $0 < \alpha < 1$ we get all the other points on the line segment between (x_1,y_1) and (x_2,y_2).



Clipping



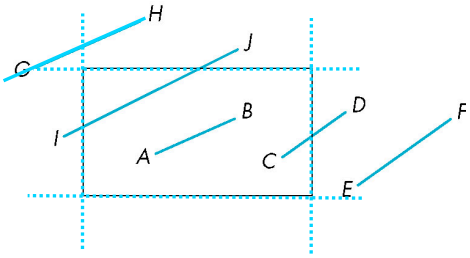
- 2D against clipping window
- 3D against clipping volume
- Easy for line segments/polygons
- Hard for curves and text
  - ➢ Convert to lines and polygons first



- 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
  - Assign binary opcodes for the 9 regions for fast processing: $b_0b_1b_2b_3$



1001	1000	1010	$y = y_{max}$
0001	0000	0010	$y = y_{min}$
0101	0100	0110	
$x = x_{min}$		$x = x_{max}$	

- Computation of outcode requires at most 4 subtractions:

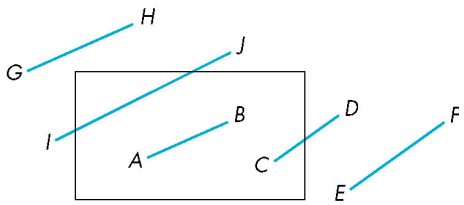
  $b_0 = 1$ if $y > y_{max}$, 0 otherwise
  $b_1 = 1$ if $y < y_{min}$, 0 otherwise
  $b_2 = 1$ if $x > x_{max}$, 0 otherwise
  $b_3 = 1$ if $x < x_{min}$, 0 otherwise

```
GH = [0001,1000]
IJ = [0001,1000]
AB = [0000,0000]
CD = [0000,0010]
EF = [0010,0010]
```


Cohen-Sutherland Algorithm

- Compute opcodes for each line segment s.
- ➢ Given: line segment by two opcodes $o_1(s)$, $o_2(s)$.

```
procedure cs-clip(segment s){
   if (o1(s) OR o2(s) = 0)
       raterize(s);
   else if (o1(s) AND o2(s) != 0)
       reject(s);
   else
       cs-clip( clipseg (s) );
}
function clipseg (s){
   // Line can be determined by opcode
   line l = find_line_by_opcode (s);
   return shortenseg (s,l);
}
```



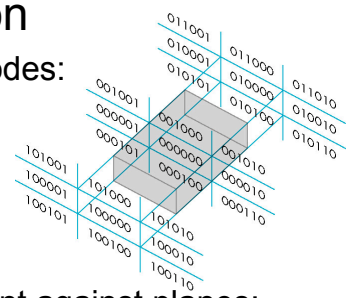

```
GH = [0001,1000]
IJ = [0001,1000]
AB = [0000,0000]
CD = [0000,0010]
EF = [0010,0010]
```

Cohen-Sutherland

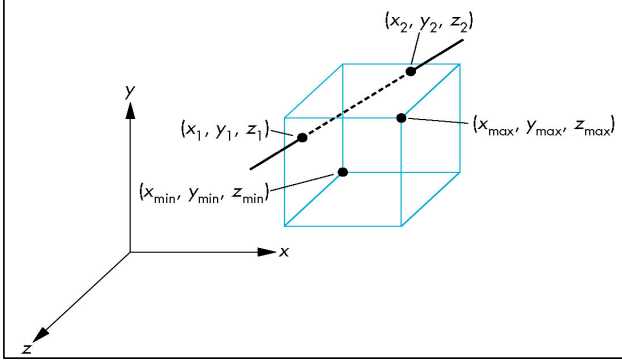
- Efficiency:
  - Often, clipping window is small relative to the size of the entire data base
  - Most segments are outside one or more window sides and can be eliminated based on their outcodes
  - Inefficiency when code has to be reexecuted for line segments that must be shortened in more than one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

3D-extension

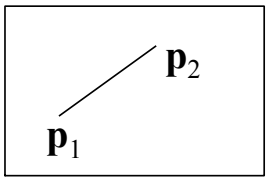
- Use 6-bit outcodes:



- When needed, clip line segment against planes:



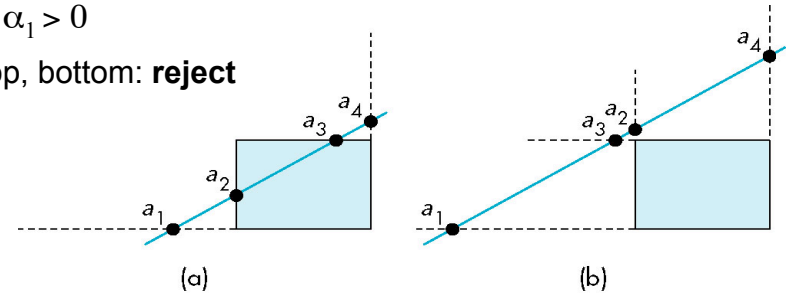
Liang-Barsky Clipping



- Consider the parametric form of a line segment

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

- Distinguish between cases
  - based on ordering of the values of α
  - where the line segment crosses the window
- In (a): $1 > \alpha_4 > \alpha_3 > \alpha_2 > \alpha_1 > 0$
  - Intersect right, top, left, bottom: **shorten**
- In (b): $1 > \alpha_4 > \alpha_2 > \alpha_3 > \alpha_1 > 0$
  - Intersect right, left, top, bottom: **reject**



Liang-Barsky Clipping

1. Given parametric form it follows:

$$\begin{aligned} \Delta x &= x_2 - x_1 \\ \Delta y &= y_2 - y_1 \\ x &= x_1 + t\Delta x \\ y &= y_1 + t\Delta y \quad 0 \le t \le 1 \end{aligned}$$

2. This is inside window; it follows:

$$xw_{min} \le x_1 + t\Delta x \le xw_{max}$$
$$yw_{min} \le y_1 + t\Delta y \le yw_{max}$$

3. These are 4 in-equations:

$$tp_k \le q_t \quad k = 1, 2, 3, 4$$

$$\begin{aligned} t(-\Delta x) &\le x_1 - xw_{min} \\ t\Delta x &\le xw_{max} - x_1 \\ t(-\Delta y) &\le y_1 - yw_{min} \\ t\Delta y &\le yw_{max} - y_1 \end{aligned}$$

Liang-Barsky Clipping

1. Test one by one: avoid comp. intersection as long as possible.
2. Rejection can be done on ordering often before all 4 are comp.
3. Using parametric form, it follows (similar for the other 3 sides):

$$\begin{aligned} p(\alpha) &= (1-\alpha)p_1 + \alpha p_2, 0 \le \alpha \le 1 \\ y_{max} &= (1-\alpha_3)y_1 + \alpha_3 y_2 \\ x_{min} &= (1-\alpha_2)x_1 + \alpha_2 x_2 \end{aligned}$$

$$\alpha_3 = \frac{y_{max} - y_1}{y_2 - y_1} \qquad \alpha_2 = \frac{x_{min} - x_1}{x_2 - x_1}$$

- avoid floating-point execution until intersection is actually required.

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

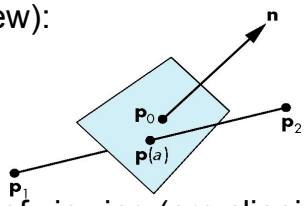
Liang-Barsky Clipping

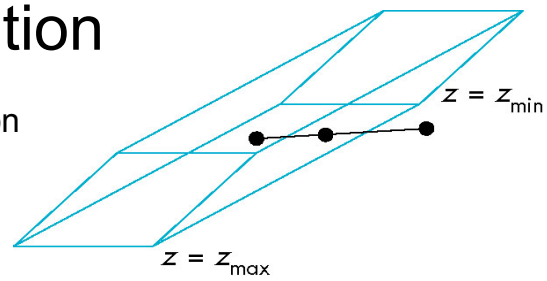
- Example: http://ls7-www.cs.uni-dortmund.de/students/projectgroups/acit/lineclip.shtml

Clipping and Normalization

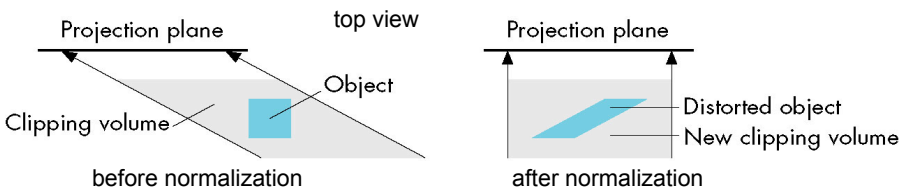
- General clipping in 3D requires intersection of line segments against arbitrary plane (Example: oblique view):

$$a = \frac{n \bullet (p_o - p_1)}{n \bullet (p_2 - p_1)}$$

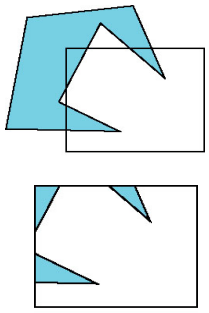




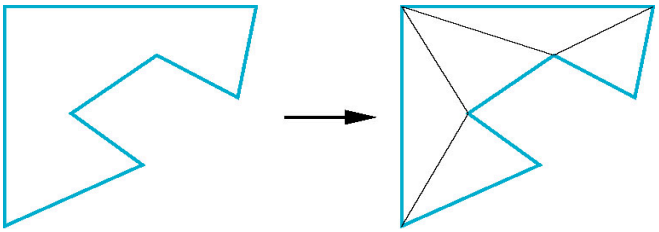
- Normalization is part of viewing (pre clipping)but after normalization, we clip against sides of right parallelepiped
  - Typical intersection calculation now requires only a floating point subtraction, e.g. is $x > x_{max}$?



Polygon Clipping

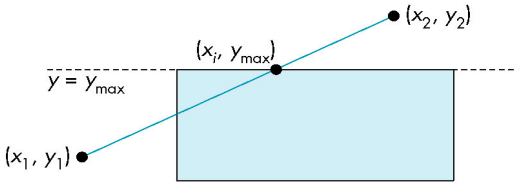


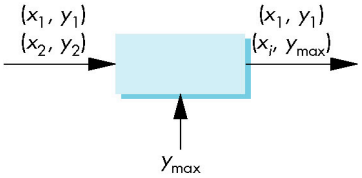
- Not as simple as line segment clipping
  - Clipping a line segment yields at most one line segment
  - Clipping a polygon can yield multiple polygons
  - However, clipping a convex polygon can yield at most one other polygon
- One strategy is to replace nonconvex (concave) polygons with a set of triangular polygons (a tessellation)
- Also makes fill easier (e.g., tessellation code available in GLU library)



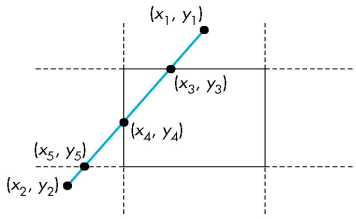
Clipping pipeline

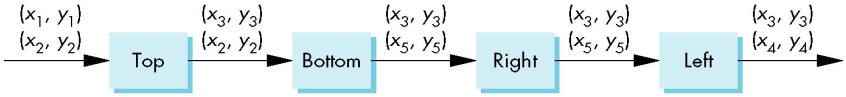
- Consider line segment clipping as a process that takes in two vertices and produces either no vertices or the vertices of a clipped line segment:



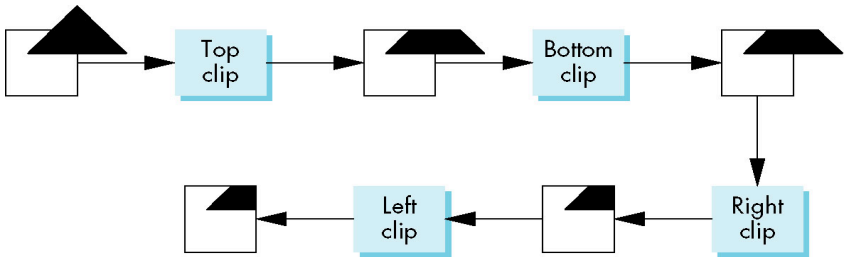


- Clipping against window sides is independent of other sides
- Place independent clippers in a pipeline:





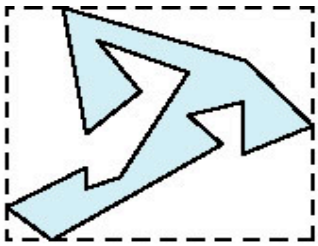
Pipeline Clipping of Polygons



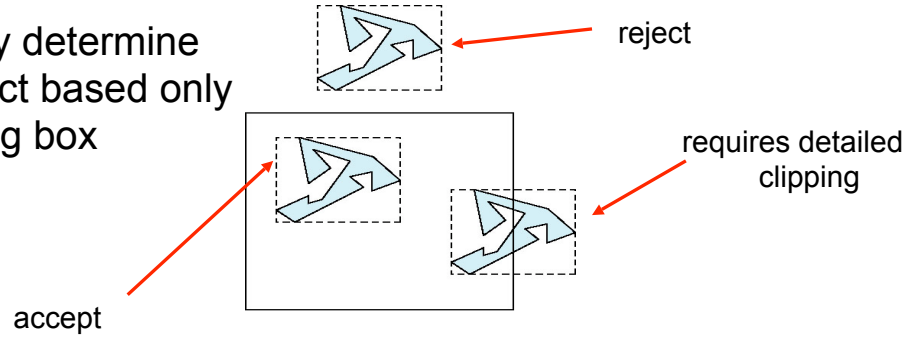
- Three dimensions: add front and back clippers
- Strategy used in SGI Geometry Engine
- Small increase in latency

Clipping 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



- Can usually determine accept/reject based only on bounding box



Realtime 3D Computer Graphics Virtual Reality

Rasterization – Scan conversion

Rasterization

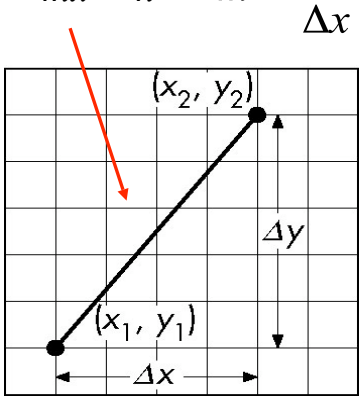
- Rasterization (scan conversion)
  - Determine which pixels that are inside primitive specified by a set of vertices
  - Produces a set of fragments
  - Fragments have a location (pixel location) and other attributes such color and texture coordinates that are determined by interpolating values at vertices
- Pixel colors determined later using color, texture, and other vertex properties

Scan Conversion of Line Segments

- Start with segment in window coordinates with integer values for endpoints
- Assume implementation has a **`write_pixel(...)`** function
- Digital Differential Analyzer
  - DDA was a mechanical device for numerical solution of differential equations
  - Line $y=mx+h$ satisfies differential equation

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

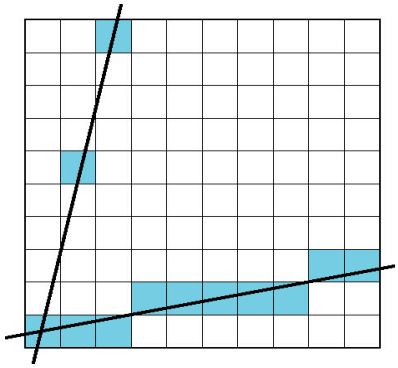
$$y = mx + h \qquad m = \frac{\Delta y}{\Delta x}$$



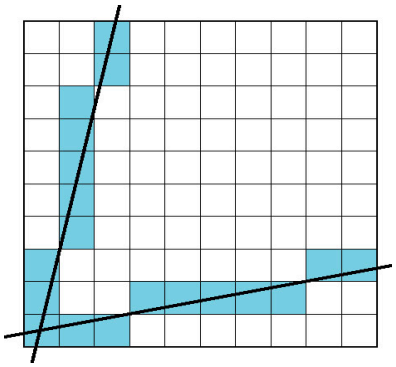

```
For(x=x1; x<=x2,x++) {
   y+=m;
   write_pixel(x, round(y), line_color)
}
```


Problem

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



- Solution: Using symmetry
  - Use for $1 \geq m \geq 0$
  - For $m > 1$, swap role of x and y
    - For each y, plot closest x



Simple DDA Line Algorithm

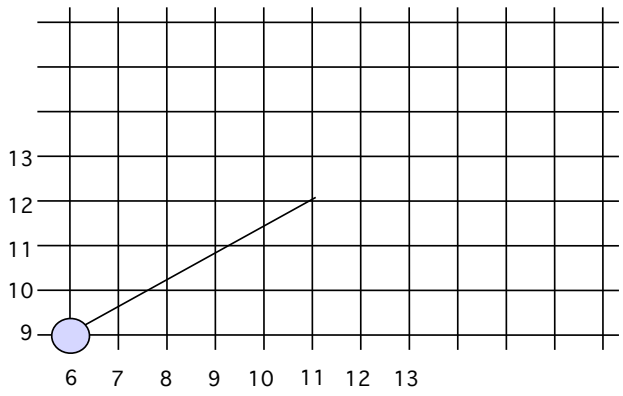
```
void DDA(int X1,Y1,X2,Y2)
{
   int    Length, I;
   float X,Y,Xinc,Yinc;

   Length = ABS(X2 - X1);
   if (ABS(Y2 - Y1)  > Length)
       Length = ABS(Y2-Y1);
   Xinc = (X2 - X1)/Length;
   Yinc = (Y2 - Y1)/Length;
   X = X1;
   Y = Y1;
   while(X<X2){
      write_pixel(Round(X),
                  Round(Y),
                  line_color);
      X = X + Xinc;
      Y = Y + Yinc;
   }
}
```

DDA creates good lines but it is too time consuming due to the round function and long operations on real values.

DDA Example

Compute which pixels should be turned on to represent the line from (6,9) to (11,12).

Length = ?
Xinc = ?
Yinc = ?

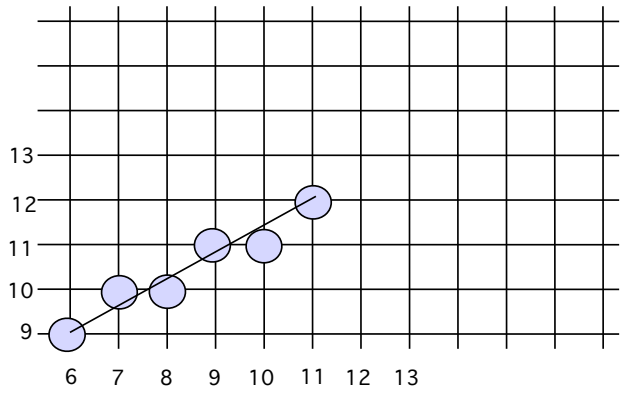


DDA Example

Line from (6,9) to (11,12).

Length := Max of (ABS(11-6), ABS(12-9)) = 5
Xinc := 1
Yinc := 0.6

Values computed are:
(6, 9)
(7, 9.6)
(8, 10.2)
(9, 10.8)
(10, 11.4)
(11, 12)

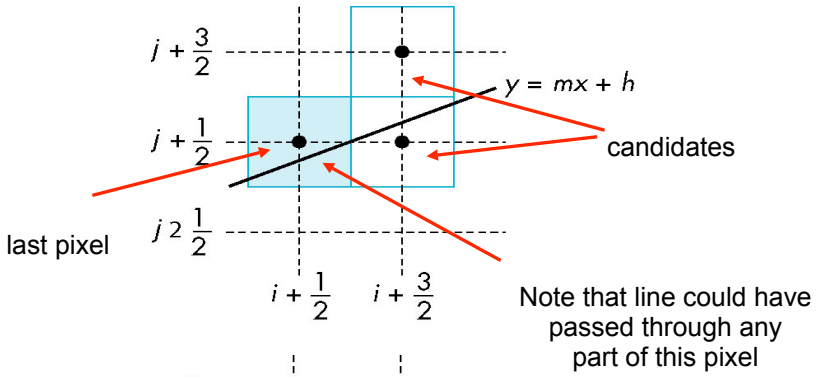


Bresenham's Algorithm

- DDA requires one floating point addition per step
- We can eliminate all fp through Bresenham's algorithm
- Consider only $1 \geq m \geq 0$
  - Other cases by symmetry
- Assume pixel centers are at half integers
- If we start at a pixel that has been written, there are only two candidates for the next pixel to be written into the frame buffer

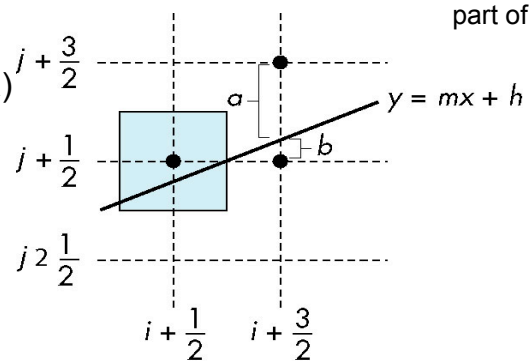
Idea

$1 \geq m \geq 0$



Decision variable: $d = \Delta x(b-a)$

d is an integer
d < 0 use lower pixel
d > 0 use upper pixel



Incremental Form

- More efficient if we look at d_k, the value of the decision variable at $x = k$

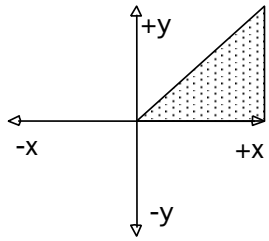
$$d_{k+1} = d_k - 2\Delta y, \quad \text{if } d_k > 0$$

$$d_{k+1} = d_k - 2(\Delta y - \Delta x), \quad \text{otherwise}$$

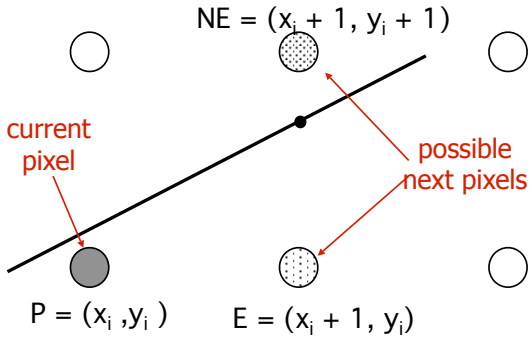
  - For each x, we need do only an integer addition and a test
  - Single instruction on graphics chips

Fast Lines – Derivation by Midpoint Method

- Simplifying assumptions: Draw line between points (0,0) and (a,b) with slope m between 0 and 1 (i.e. line lies in first quadrant).
  - General formula for line is **y = mx + B** where
    - **m** is the slope of the line and
    - **B** is the y-intercept.
    - From our assumptions **m** = b/a and **B** = 0.
    - ➢ y = (b/a)x + 0
      ⇒ <u>f(x,y) = bx - ay = 0</u> is an equation for the line.

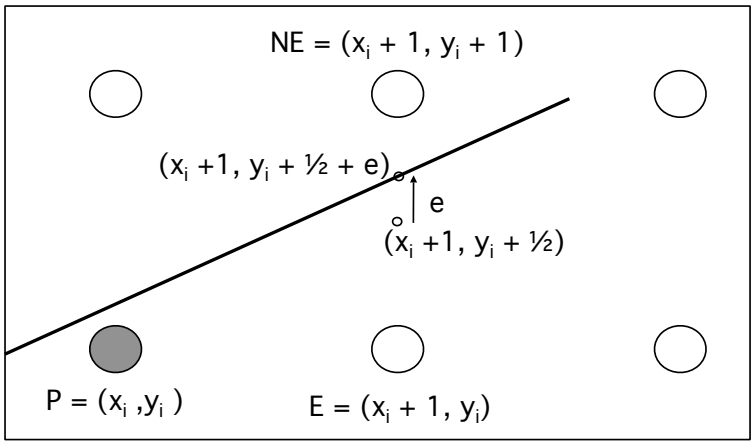


- For lines in the first quadrant, given one pixel on the line, the next pixel is to the right (E) or to the right and up (NE).
- Having turned on pixel P at (x_i, y_i), the next pixel is
  - NE at (x_i+1, y_i+1) or
  - E at (x_i+1, y_i).
- Choose pixel closer to the line $f(x, y) = bx - ay = 0$.



Fast Lines (cont.)

- The midpoint between pixels E and NE is $(x_i + 1, y_i + ½)$.
- Let e be the "upward" distance between the midpoint and where the line actually crosses between E and NE.
- If <u>e is positive</u> the line crosses above the midpoint and is closer to NE.
- If <u>e is negative</u>, the line crosses below the midpoint and is closer to E.
- ➢ To pick the correct point we only need to know the <u>sign</u> of e.



The Decision Variable

$$f(x_i+1, y_i+ ½ + e) = 0 \text{ (point on line)}$$
$$= b(x_i + 1) - a(y_i + ½ + e)$$
$$= b(x_i + 1) - a(y_i + ½) – ae$$
$$= f(x_i + 1, y_i + ½) - ae$$

$$\boxed{\rightarrow f(x_i + 1, y_i + ½) = ae}$$

Let $\mathbf{d_i} = f(x_i + 1, y_i + ½) = \mathbf{ae}$; $\mathbf{d_i}$ is known as the **decision variable**.
Since $a \geq 0$, d_i has the same sign as e.

Therefore, we only need to know the value of d_i to choose between pixels E and NE. If $d_i \geq 0$ choose NE, else choose E.

But, calculating d_i directly each time requires at least two adds, a subtract, and two multiplies -> too slow!

Decision Variable calculation

Algorithm:

Calculate d_0 directly, then for each i >= 0:

if $d_i \geq 0$ Then

Choose NE = $(x_i + 1, y_i + 1)$ as next point

$$d_{i+1} = f(x_{i+1} + 1, y_{i+1} + ½) = f(x_i + 1 + 1, y_i + 1 + ½)$$
$$= b(x_i + 1 + 1) - a(y_i + 1 + ½) = f(x_i + 1, y_i + ½) + b - a$$
$$= d_i + b - a$$

else

Choose E = $(x_i + 1, y_i)$ as next point

$$d_{i+1} = f(x_{i+1} + 1, y_{i+1} + ½) = f(x_i + 1 + 1, y_i + ½)$$
$$= b(x_i + 1 + 1) - a(y_i + ½) = f(x_i + 1, y_i + ½) + b$$
$$= d_i + b$$

→ Knowing d_i, we need only add a **constant** term to find d_{i+1} !

Fast Line Algorithm

The initial value for the decision variable, d_0, may be calculated directly from the formula at point (0,0).

$d_0 = f(0 + 1, 0 + 1/2) = b(1) - a(1/2) = b - a/2$

Therefore, the algorithm for a line from (0,0) to (a,b) in the first quadrant is:

```
x = 0;
y = 0;
d = b - a/2;
for(i = 0; i < a; i++) {
Plot(x,y);
if (d ≥ 0) {
        x = x + 1;
        y = y + 1;
        d = d + b - a;
}
else {
        x = x + 1;
        d = d + b
}
}
```

Note that the only non-integer value is a/2. If we then multiply by 2 to get d' = 2d, we can do all integer arithmetic. The algorithm still works since we only care about the sign, not the value of d.

Bresenham's Line Algorithm

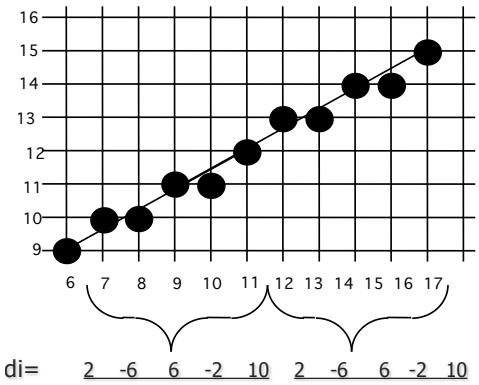
We can also generalize the algorithm to work for lines beginning at points other than (0,0) by giving x and y the proper initial values. This results in Bresenham's Line Algorithm.

```
{Bresenham for lines with slope between 0 and 1}
        a = ABS(xend - xstart);
        b = ABS(yend - ystart);
        d = 2*b - a;
        Incr1 = 2*(b-a);
        Incr2 = 2*b;
        if (xstart > xend) {
                x = xend;
                y = yend
        }
        else {
                x = xstart;
                y = ystart
        }
```

```
for (i = 0; i<a; i++){
Plot(x,y);
x = x + 1;
if (d ≥ 0) {
    y = y + 1;
    d = d + incr1;
}
else
    d = d + incr2;
}
}
```

Optimizations

- Speed can be increased even more by detecting cycles in the decision variable. These cycles correspond to a repeated pattern of pixel choices.
- The pattern is saved and if a cycle is detected it is repeated without recalculating.



Line Rendering References

Bresenham, J.E., "Ambiguities In Incremental Line Rastering," IEEE Computer Graphics And Applications, Vol. 7, No. 5, May 1987.

Eckland, Eric, "Improved Techniques For Optimising Iterative Decision-Variable Algorithms, Drawing Anti-Aliased Lines Quickly And Creating Easy To Use Color Charts," CSC 462 Project Report, Department of Computer Science, North Carolina State University (Spring 1987).

Foley, J.D. and A. Van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley 1982.

Newman, W.M and R.F. Sproull, Principles Of Interactive Computer Graphics, McGraw-Hill, 1979.

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

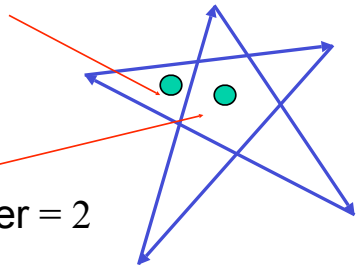
Filling in the Frame Buffer

- Fill at end of pipeline
- If a point is inside a polygon color it with the inside (polygon) color
- Three approaches:
  - Flood fill, Scan line fill, Odd-Even fill
- Polygon type matters:
  - Convex polygons preferred, non-convex polygons assumed to have been tessellated
- Shades (colors) have been computed for vertices (Gouraud shading)
- Combine with depth test: z-buffer algorithm
  - March across scan lines interpolating shades
  - Incremental work small

Winding Number

- Count clockwise encirclements of point

winding number = 1

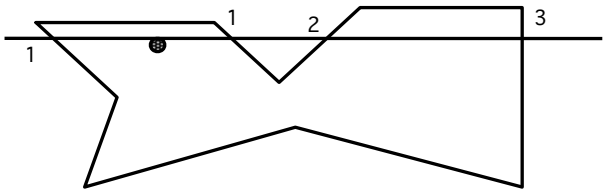


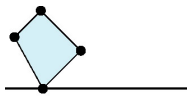
winding number = 2

- Alternate definition of inside: inside if winding number ≠ 0

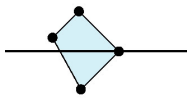
Inside Polygon Test

Inside test: A point P is inside a polygon if and only if a scanline intersects the polygon edges an odd number of times moving from P in either direction.





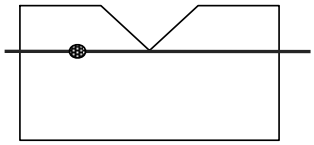
(a)



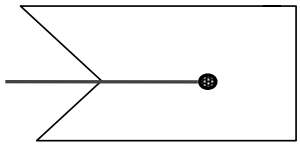
(b)

Problem when scan line crosses a vertex:

Does the vertex count as two points?

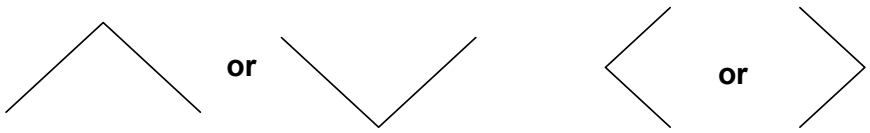


Or should it count as one point?

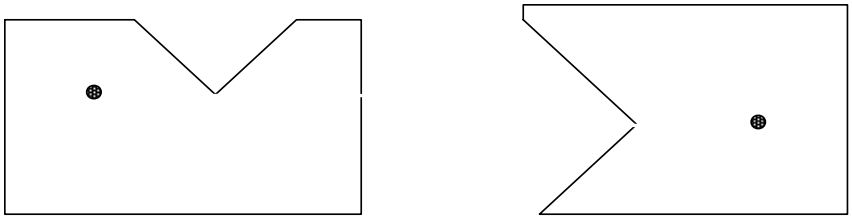


Max-Min Test

When crossing a vertex, if the vertex is a local maximum or minimum then count it twice, else count it once.

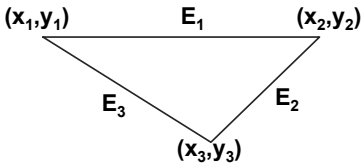


Count twice **Count once**



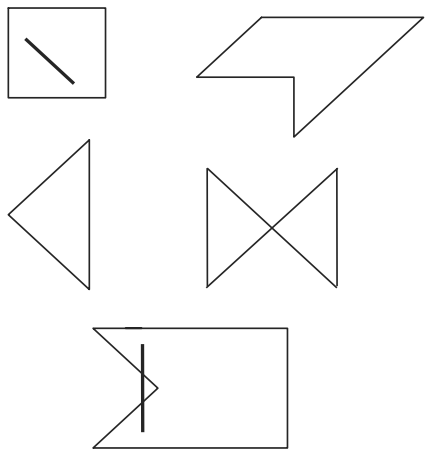
Polygons

$P = \{ (x_i , y_i) \}$ i=1,n



- A **polygon** is a many-sided **planar** figure composed of **vertices** and **edges**.
- **Vertices** are represented by points (x,y).
- **Edges** are represented as line segments which connect two points, (x_1,y_1) and (x_2,y_2).

- **Convex** Polygon:
  - For <u>any</u> two points P_1, P_2 inside the polygon, all points on the line segment which connects P_1 and P_2 are inside the polygon.
  - All points $P = uP_1 + (1-u)P_2$, u in [0,1] are inside the polygon provided that P_1 and P_2 are inside the polygon.
- **Concave** Polygon
  - A polygon which is not convex.
- **Simple** Polygons
  - Polygons whose edges do not cross.
- **Non simple** Polygons
  - Polygons whose edges cross.
  - E.g., two different OpenGL implementations may render non simple polygons differently. OpenGL does not check if polygons are simple.

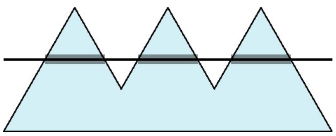


Flood Fill

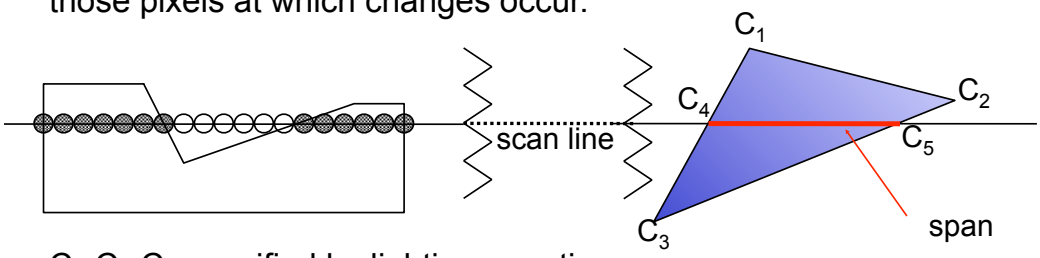
- Fill can be done recursively if we know a seed point located inside (WHITE)
- Scan convert edges into buffer in edge/inside color (BLACK)

```
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);
    }
}
```


Using Interpolation



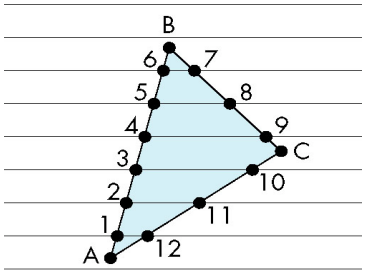
- Fill the polygon 1 scan line at a time
- Determine which pixels on each scan line are inside the polygon
  - set those pixels to the appropriate value.
- Key idea: Don't check each pixel for "inside-ness". Instead, look only for those pixels at which changes occur.



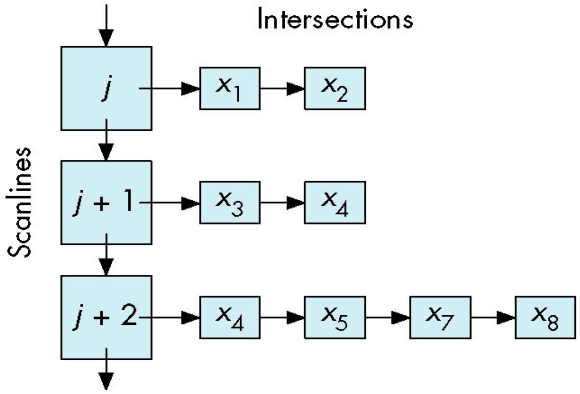
- C_1 C_2 C_3 specified by lighting equation
  - e.g., OpenGL: `glColor` or by **vertex shading**
- C_4 determined by interpolating between C_1 and C_2
- C_5 determined by interpolating between C_2 and C_3
- interpolate between C_4 and C_5 along span: **bilinear interpolation**

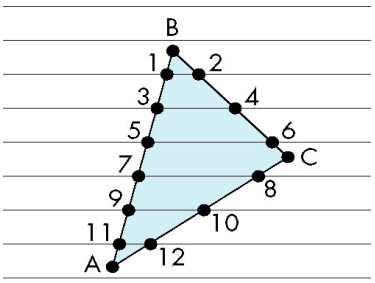
Scan-Line Fill

- Can also fill by maintaining a data structure of all intersections of polygons with scan lines
  - Sort by scan line
  - Fill each span



vertex order generated by vertex list





desired order

Scan-Line Algorithm

For each scan line:

1. Find the intersections of the scan line with all edges of the polygon.
2. Sort the intersections by increasing x-coordinate.
3. Fill in all pixels between pairs of intersections.

Problem:

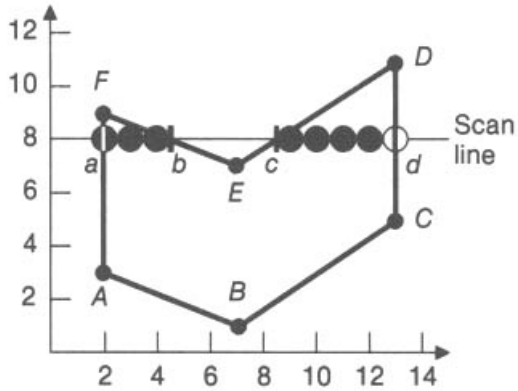
Calculating intersections is slow.

Solution:

Incremental computation / coherence

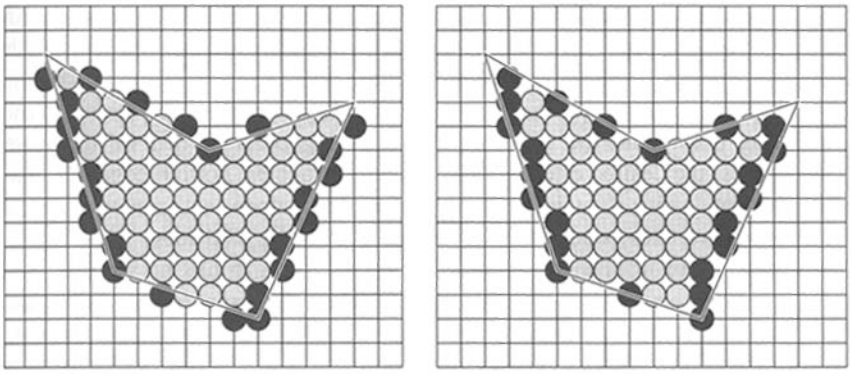
For scan line number 8 the sorted list of x-coordinates is (2,4,9,13) (b and c are initially no integers)

Therefore fill pixels with x-coordinates 2-4 and 9-13.



Filling using line drawings

- Information about “interior” is missing
- Pixels are choosen which are near the desired line
- Strategy adds some extra pixels which are not located inside the polygon



Edge Coherence

- Observation: Not all edges intersect each scanline.
- Many edges intersected by scanline i will also be intersected by scanline i+1

- Formula for scanline s is y = s, for an edge is y = mx + b
- Their intersection is

  $s = mx_s + b \dashrightarrow x_s = (s-b)/m$

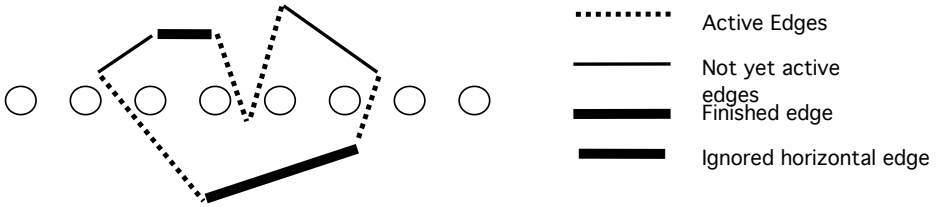
- For scanline s + 1,

  $x_{s+1} = (s+1 - b)/m = x_s + 1/m$

Incremental calculation: $x_{s+1} = x_s + 1/m$

Processing Polygons

- Polygon edges are sorted according to their minimum / maximum Y.
- Scan lines are processed in increasing (upward) / decreasing (downward) Y order.
- When the current scan line reaches the lower / upper endpoint of an edge it becomes active.
- When the current scan line moves above the upper / below the lower endpoint, the edge becomes inactive.



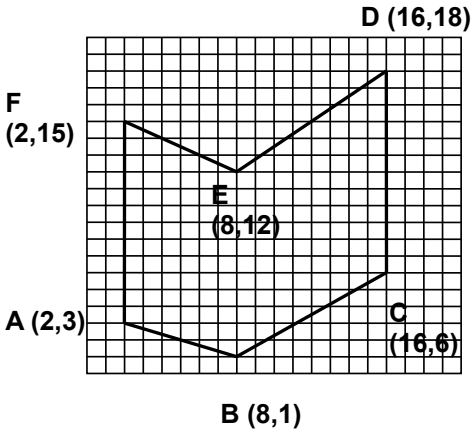
- Active edges are sorted according to increasing X. Filling the scan line starts at the leftmost edge intersection and stops at the second. It restarts at the third intersection and stops at the fourth. . . (spans)

Polygon fill rules (to ensure consistency)

1. Horizontal edges: Do not include in edge table
2. Horizontal edges: Drawn either on the bottom or on the top.
3. Vertices: If local max or min, then count twice, else count once.
4. Either vertices at local minima or at local maxima are drawn.
5. Only turn on pixels whose centers are *interior* to the polygon: round up values on the left edge of a span, round down on the right edge

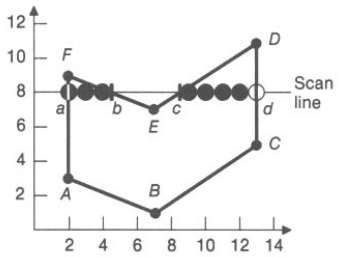
Polygon fill example

- The edge table (ET) with edges entries sorted in increasing y and x of the lower end.
  - y_{max}: max y-coordinate of edge
  - x_{min}: x-coordinate of lowest edge point
  - 1/m: x-increment used for stepping from one scan line to the next



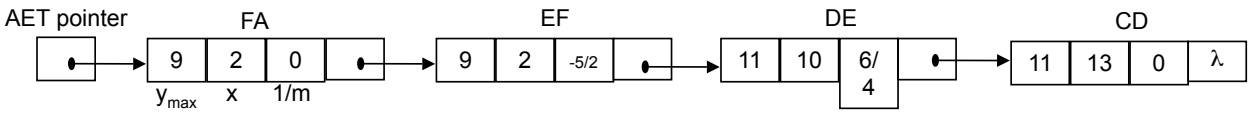
19	NULL								
18	NULL								
17	NULL								
16	NULL								
15	NULL								
14	NULL								
13	NULL								
12	->	15	8	-2	->	18	8	4/3	NULL
11	NULL								
10	NULL								
9	NULL								
8	NULL								
7	NULL								
6	->	18	16	0	NULL				
5	NULL								
4	NULL								
3	->	15	2	0	NULL				
2	NULL								
1	->	3	8	-3	->	6	8	8/5	NULL

Processing steps

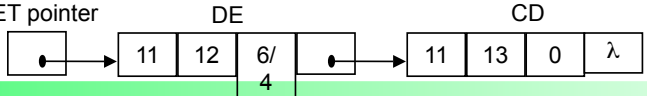


1. Set y to smallest y with entry in ET, i.e., y for the first non-empty bucket
2. Init Active Edge Table (AET) to be empty
3. Repeat until AET and ET are empty:
   1. Move form ET bucket y to the AET those edges whose y_{min}=y (entering edges)
   2. Remove from AET those edges for which y=y_{max} (not involved in next scan line), then sort AET (remember: ET is presorted)
   3. Fill desired pixel values on scan line y by using pairs of x-coords from AET
   4. Increment y by 1 (next scan line)
   5. For each nonvertical edge remaining in AET, update x for new y

scan line 9:

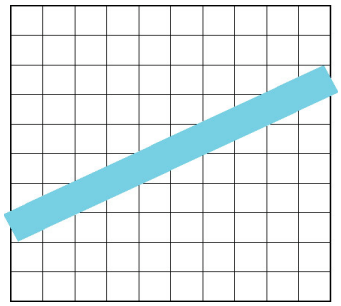


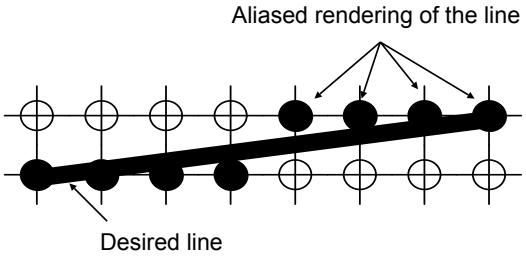
scan line 10:



Aliasing

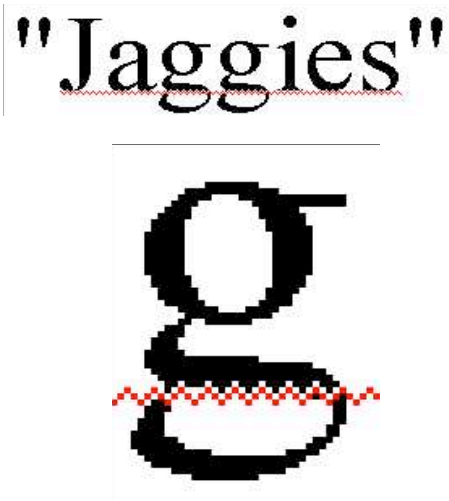
- Aliasing is caused by finite addressability of the display.
- Approximation of lines and circles with discrete points often gives a staircase appearance or "Jaggies".
- Ideal rasterized line should be 1 pixel wide

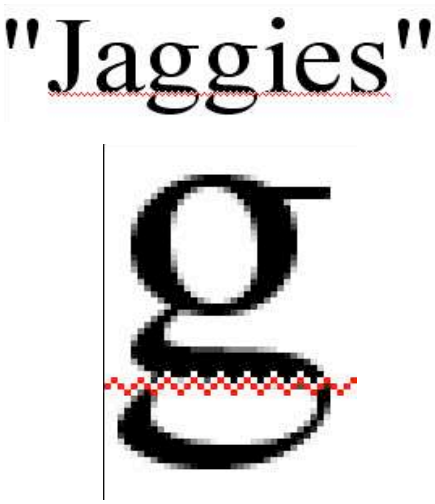




- Choosing best y for each x (or visa versa) produces aliased raster lines

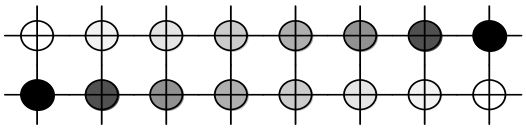
Aliasing / Antialiasing Examples





Antialiasing - solutions

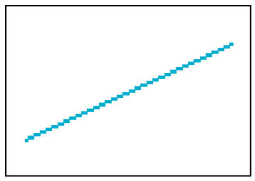
- Aliasing can be smoothed out by using higher addressability.
- If addressability is fixed but intensity is variable, use the intensity to control the address of a "virtual pixel".
  - Two adjacent pixels can be be used to give the impression of a point part way between them.
  - The perceived location of the point is dependent upon the ratio of the intensities used at each.
  - The impression of a pixel located halfway between two addressable points can be given by having two adjacent pixels at half intensity.
- An antialiased line has a series of virtual pixels each located at the proper address.

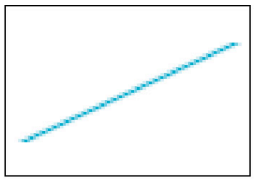


Antialiasing by Area Averaging

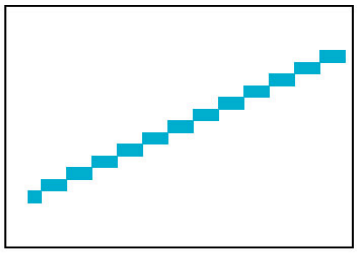
- Color multiple pixels for each x depending on coverage by ideal line

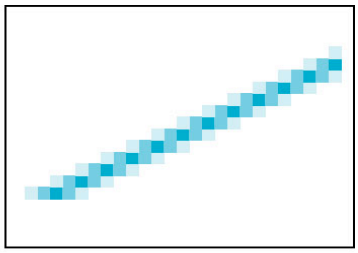
original





antialiased

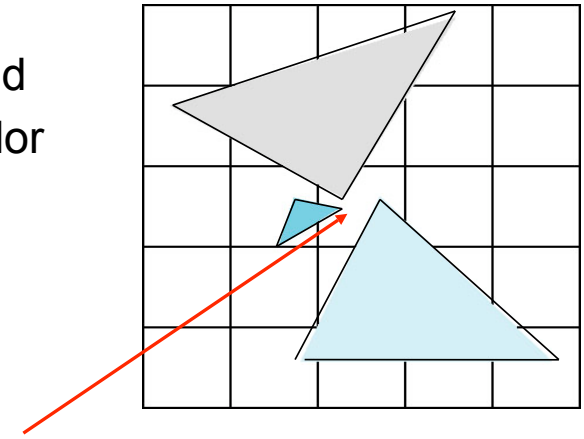




magnified

Polygon Aliasing

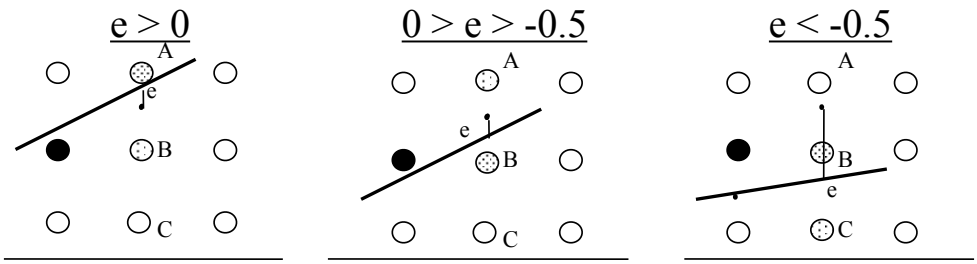
- Aliasing problems can be serious for polygons
  - Jaggedness of edges
  - Small polygons neglected
  - Need compositing so color of one polygon does not totally determine color of pixel



All three polygons should contribute to color

Antialiased Bresenham Lines

- Line drawing algorithms such as Bresenham's can easily be modified to implement virtual pixels. We use the distance ($e = d_i/a$) value to determine pixel intensities.
- Three possible cases which occur during the Bresenham algorithm:



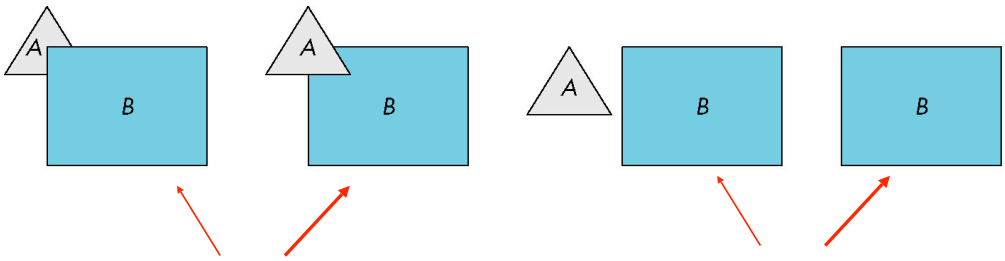
$e > 0$	$0 > e > -0.5$	$e < -0.5$
$A = 0.5 + e$	$A = 0.5 + e$	$A = 0$
$B = 1 - abs(e+0.5)$	$B = 1 - abs(e+0.5)$	$B = 1 - abs(e+0.5)$
$C = 0$	$C = 0$	$C = -0.5 - e$

Clipping and Visibility

- Clipping has much in common with hidden-surface removal
- In both cases, we are trying to remove objects that are not visible to the camera
- Often we can use visibility or occlusion testing early in the process to eliminate as many polygons as possible before going through the entire pipeline

Hidden Surface Removal

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

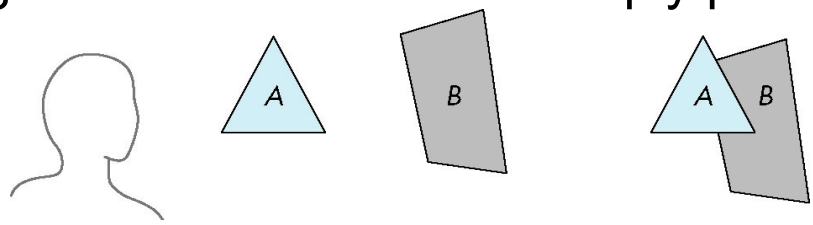


partially obscuring can draw independently

- Worst case complexity $O(n^2)$ for n polygons

Painter's Algorithm

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



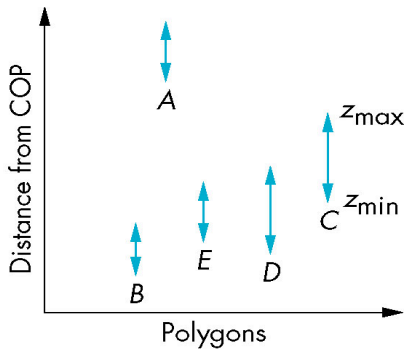
B behind A as seen by viewer

Fill B then A

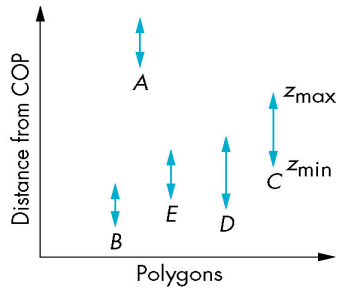
Depth Sort

- Requires ordering of polygons first
  - O(n log n) calculation for ordering
  - Not every polygon is either in front or behind all other polygons
- Order polygons and deal with easy cases first, harder later

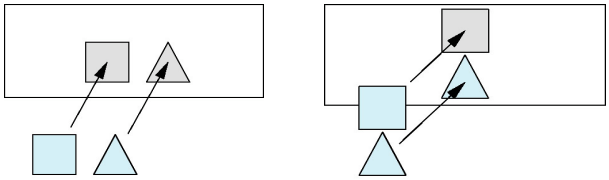
Polygons sorted by distance from COP



Depth sort cases

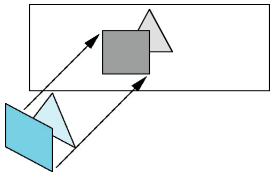


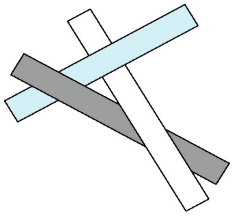
- Easy cases:
  - Lies behind all other polygons (can render):
  - Polygons overlap in z but not in either x or y (can render independently):



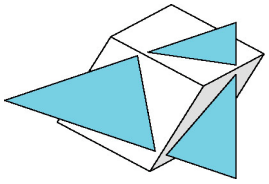
- Hard cases:

Overlap in all directions but can one is fully on one side of the other



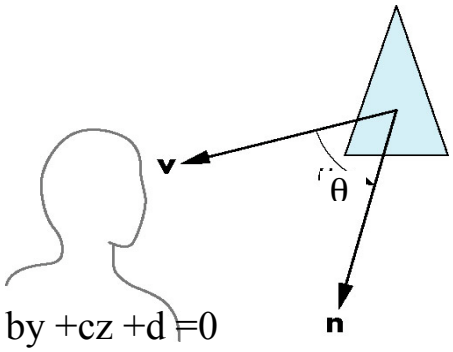


cyclic overlap



penetration

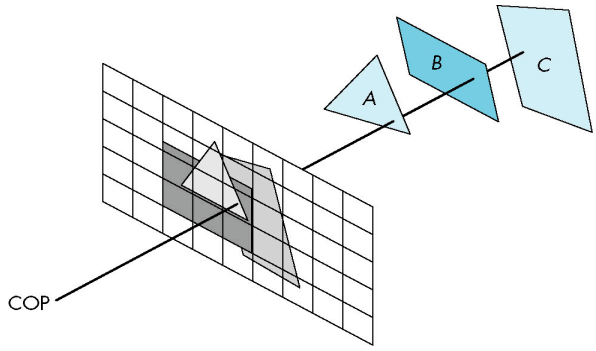
Back-Face Removal (Culling)



- face is visible iff $90 \geq \theta \geq -90$
  equivalently $\cos\theta \geq 0$
  or $\mathbf{v} \cdot \mathbf{n} \geq 0$
- plane of face has form $ax + by + cz + d = 0$
  but after normalization $\mathbf{n} = (0\ 0\ 1\ 0)^T$
- need only test the sign of c
- In OpenGL we can simply enable culling
  but may not work correctly if we have nonconvex objects

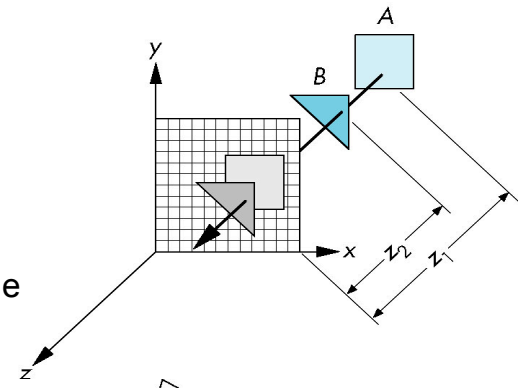
Image Space Approach

- Look at each projector (nm for an $n \times m$ frame buffer) and find closest of k polygons
- Complexity $O(nmk)$
- Ray tracing
- z-buffer

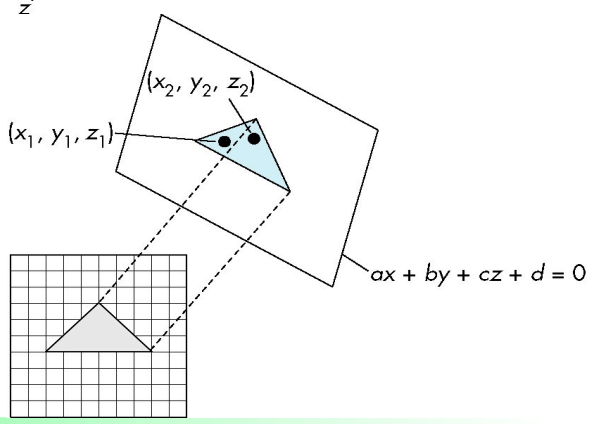


z-Buffer Algorithm

- Use a depth buffer called the z-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

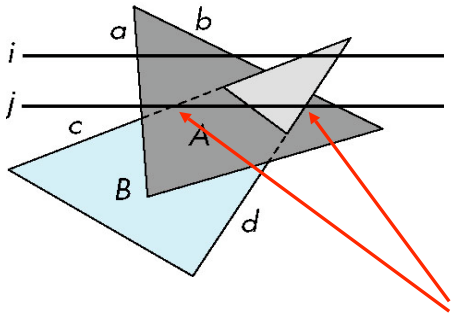


- <u>Efficency:</u>
  - If we work scan line by scan line as we move across a scan line, the depth changes satisfy $a\Delta x+b\Delta y+c\Delta z=0$
  - Along scan line $\Delta y = 0, \Delta z = -\frac{a}{c}\Delta x$
  - In screen space $\Delta x = 1$



Scan-Line Algorithm

- Can combine shading and hsr through scan line algorithm

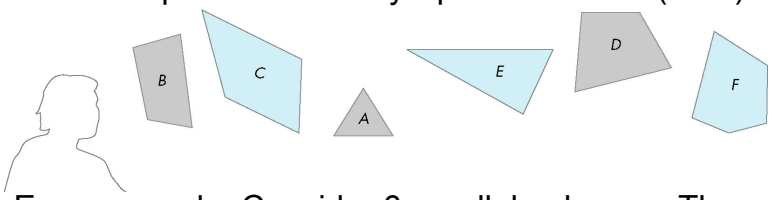


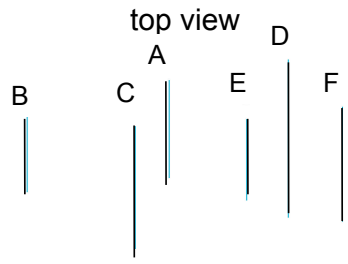
scan line i: no need for depth information, can only be in no or one polygon

scan line j: need depth information only when in more than one polygon

Visibility Testing

- In realtime applications, eliminate as many objects as possible within the application
  - Reduce burden on pipeline
  - Reduce traffic on bus
- Partition space with Binary Spatial Partition (BSP) Tree





- Easy example: Consider 6 parallel polygons. The plane of A separates B and C from D, E and F
  - Can continue recursively
    - Plane of C separates B from A
    - Plane of D separates E and F
  - Can put this information in a BSP tree
    - Use for visibility and occlusion testing