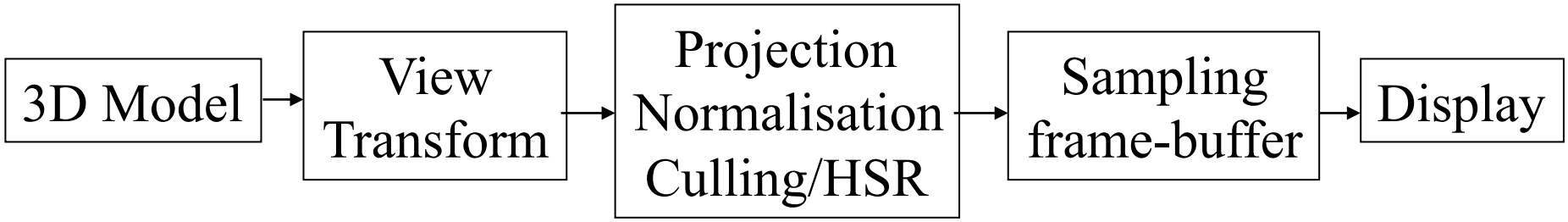
Rendering

Angel Ch. 7&9

What is Rendering?

Generation of discrete image ‘pixels’ from continuous lines and polygons

- sampling of lines
- filling polygons



How can we perform these operation efficiently?

- theoretical vs. practical performance
- hardware vs. software
- graphics pipeline architecture

Basic algorithms for implementing a graphics API

- OpenGL/PHIGS/Renderman

Graphics Pipeline

4 major tasks:

(1) **Modelling** - vertex based (points,lines,polygons)

(2) **Geometric Processing** - determine which objects are visible

(i) transform to camera coordinates

(ii) normalise the projection view volume

(iii) visibility culling/hidden-surface-removal/lighting calculation

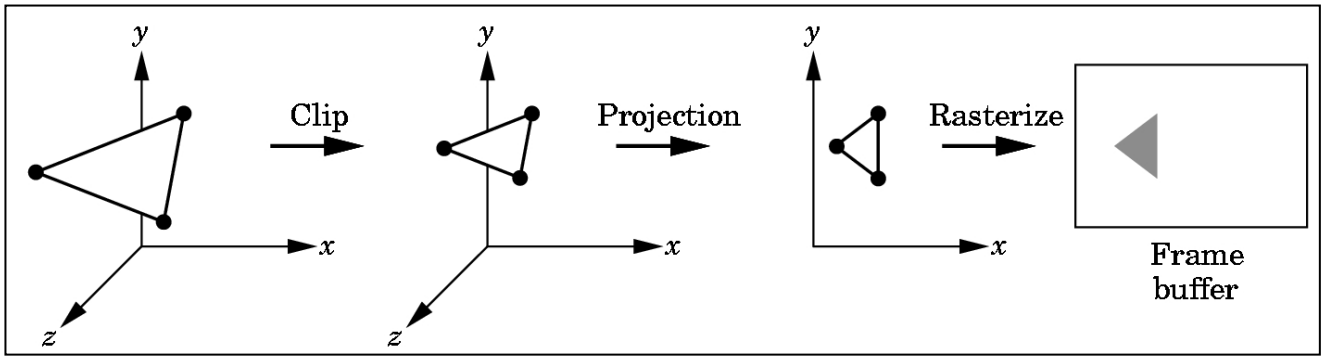
(iv) orthogonal projection (to 2D image plane)

(3) **Rasterization** - conversion of 2D geometric primites to pixel values

- discrete sampling: **rasterization or scan conversion**
- write pixel values to frame-buffer

(4) **Display** - take image from frame-buffer and draw on display

- map to quality of display (no. of colours/colour transform)
- **'anti-aliasing'** to avoid jagged edges
- read/write independent: dual ported frame-buffer memory



Implementation

2 Basic approaches

image-oriented (sort-first)

- loop over pixel rows or scan-lines
- for each pixel which scene geometry contribute
- eg. ray-tracing
- limitations: search through geometry primitives
  is slow/view dependent

object-oriented (sort-last)

- graphics pipeline of opengl
- project objects onto image plane and sort using visibility
  culling & hidden surface removal
- limitations: large memory requirement
  high cost of processing objects independently
- supported in hardware (>1million polygons/sec)
  graphics pipeline API's such as OpenGL

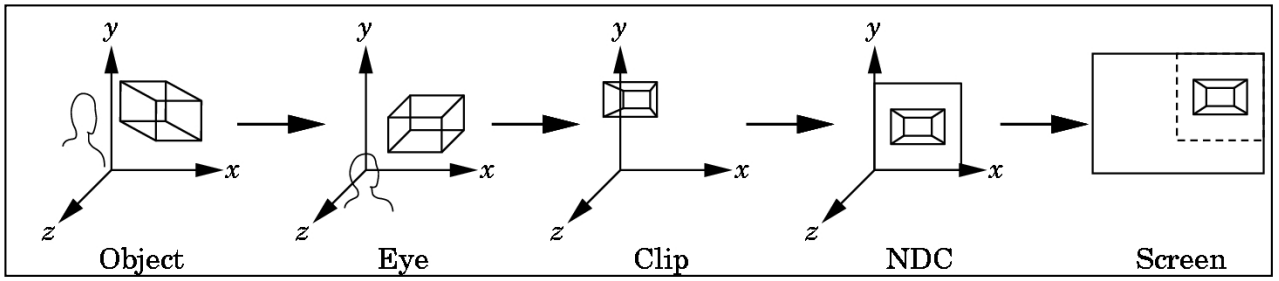
Both approaches require the 4 tasks

- (modelling/geometry processing/rendering/display)

Implementing Transforms

5 coordinate systems used in graphics pipeline:

1. World Co-ordinates
2. Camera or Eye Co-ordinates (ModelView)
3. Clip Co-ordinates: Normalised Projection View Frustum (Projection)
4. Normalised Device Co-ordinates
5. Screen Co-ordinates



Transformation from world to clip co-ordinates are 4x4 homogenous matrices

- 1 machine operation in hardware

Normalised devise co-ordinates are real (*xyz*) co-ordinates obtained from homogenous clip co-ordinates (*xyzw*) by dividing by *w*

- lie inside the cube centred on the origin: $-1<x,y,z<1$

Screen Co-ordinates perform orthographic projection and convert to units and dimensions of the display

Clipping

Clipping determines which primitives or parts of primitives appear on the display

- which part of primitive is inside the view volume frustum
- primitives outside the view frustum are **culled** or rejected
- primitives partly inside the view volume must be **clipped**

Clipping is performed with the normalised projection volume
in the clip or normalised device coordinates

Large number of algorithms proposed for clipping 2D & 3D
Consider algorithms which

- can be applied in 2D & 3D
- can be implemented in a graphics pipeline

Line Clipping: Cohen-Sutherland
Liang-Barsky

Line Clipping

2D line clipping algorithm

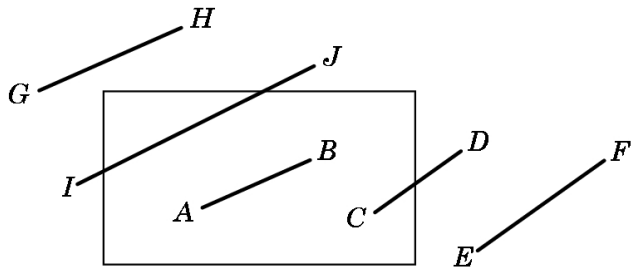
- clipping performed after lines projected to 2D

For line end-points we can test if line is

(i) inside - both ends inside AB

(ii) outside - both ends outside GH,EF

(iii) part-in - one or both ends outside CD,IJ



For case (iii) the line must be shortened before display to the part inside

Could compute intersection of lines with the view window

- requires expensive floating point division computation to find line-line intersections

Cohen-Sutherland 1963

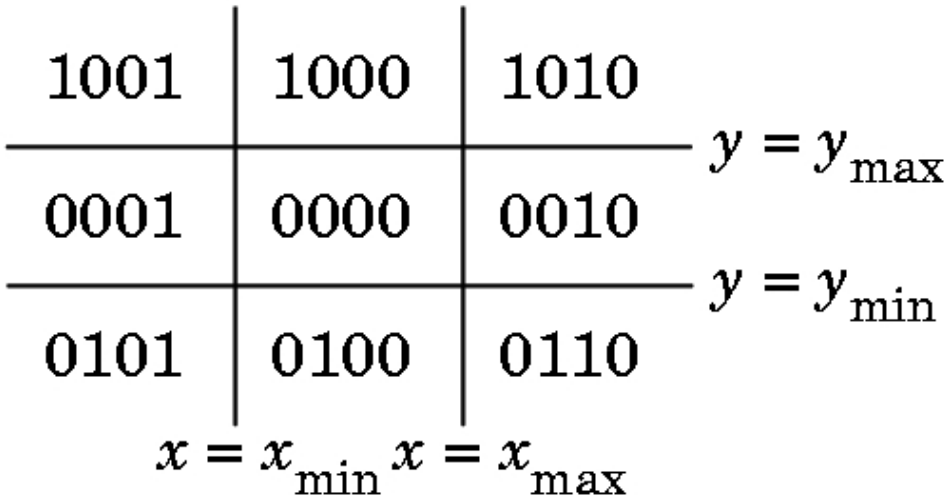
- replaced fp division with fp subtraction + logical bit operations

Cohen-Sutherland Clipping

Define a 4-bit **'outcode'** for the location of the line end-points wrt the sides of the view window

- extend the view window size to infinite lines
- split 2d projection plane into 9 regions
- each region has a unique bit code $b_0 b_1 b_2 b_3$

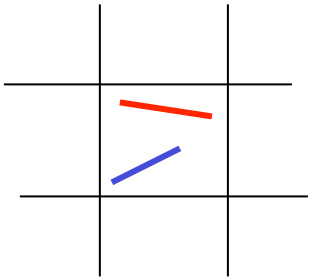
$$b_0 = \begin{cases} 1 & if\,(y > y_{\max}) \\ 0 & otherwise \end{cases} \quad b_1 = \begin{cases} 1 & if\,(y < y_{\min}) \\ 0 & otherwise \end{cases}$$



Cohen-Sutherland Clipping II

Line Clipping

Given a line $l(p_1,p_2)$ we have outcodes (o_1,o_2) for the endpoints this gives **3 cases where part of the line is inside:**

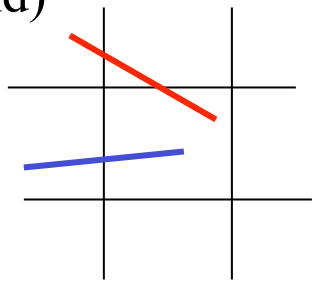


Case 1: $o_1=o_2=0$

- line inside (no clipping)

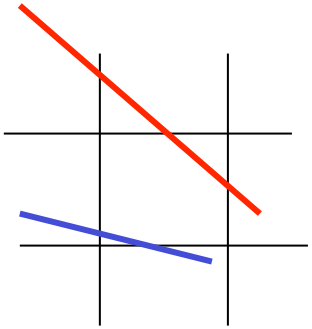
Case 2: $o_1 \neq o_2=0$ or $o_2 \neq o_1=0$ - one end-point inside (clip one end)

- non-zero outcode indicates which one or two sides of view window are intersected
- perform line-line intersection & test intersection point for outcode 0
- if 1st intersection point outcode not 0 test 2nd line intersection



Case 3: $o_1 \& o_2=0$ (logical AND) (clip both ends)

- end-points are both outside but on opposite faces
- may be an intersection
- computer intersection with one side and test if outcode is 0



Cohen-Sutherland Clipping III

All outcode checking is boolean

Floating point operations only required to compute line-line intersections
- only performed in cases 2 & 3

Algorithm performs best with many line segments few of which are displayed
- most lines lie outside and the endpoints are in the same region

$o_1 \& o_2 = 1$

Can be extended to 3D

Liang-Barsky Clipping

Represent lines in the parametric form

- efficient decisions about clipping without fp division
- more efficient solution
- Liang Barsky 1984

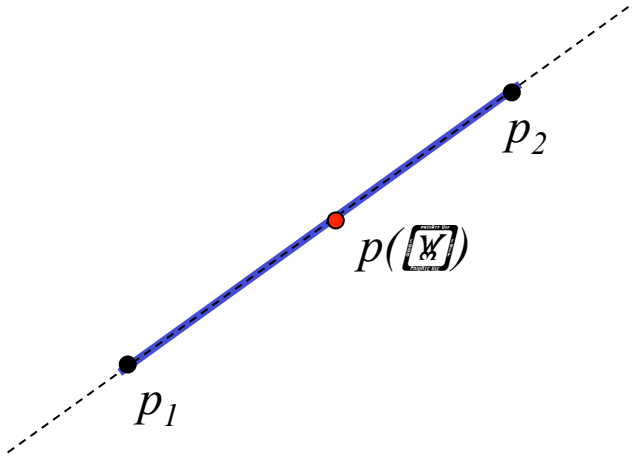
Parametric form for lines:

$$p(\alpha) = (x(\alpha), y(\alpha)) = (1-\alpha)\,p_1 + \alpha p_2$$

$$0 \le \alpha \le 1$$

$$x(\alpha) = (1-\alpha)x_1 + \alpha x_2$$

$$y(\alpha) = (1-\alpha)y_1 + \alpha y_2$$



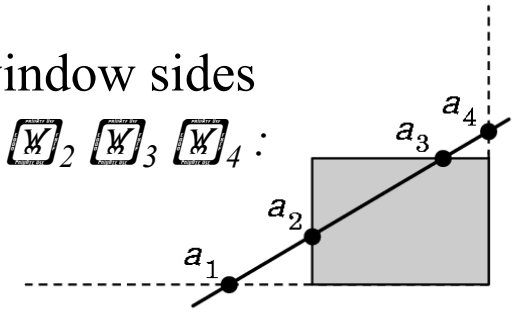
Note: parametric form is robust

- same representation works for horizontal & vertical lines ($y=ax+b$ is not)

Consider the intersection of the infinite line with the view window sides

There are 4 possible intersection with parameter values α_1 α_2 α_3 α_4 :

- unless line is horizontal or vertical
- only one point can correspond to line entering or leaving



Liang-Barsky Clipping II

Consider the order of w_1 w_2 w_3 w_4 along the line

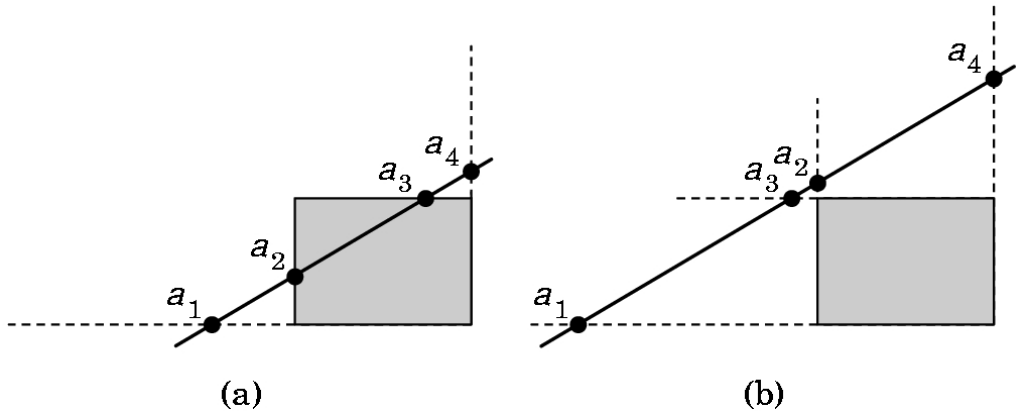
Case (a): $0 < w_1 < w_2 < w_3 < w_4 < 1$

- all 4 points are inside the line end-point
- **2 inner most** (w_2, w_3) **determine the clipped line segment**

Case (b): $0 < w_1 < w_3 < w_2 < w_4 < 1$

- all 4 points inside end-points
- order indicates that line does not intersect view window (line intersects the top&bottom before intersecting either side)

Similarly for other cases of ordering intersection



Liang-Barsky Clipping III

Efficient implementation requires that we only compute intersections that are required for clipped line segments

- computation of the intersection requires fp division
  for the side y=ymax:

$$\alpha = \frac{(y_{\max} - y_1)}{(y_2 - y_1)}$$

similarly for other sides

We can instead write: $\alpha(y_2 - y_1) = \alpha\Delta y = (y_{\max} - y_1) = \Delta y_{\max}$

All tests required for the order of intersection can be restated interms of $\Delta y, \Delta y_{\max}$ & similar terms for the other sides of the view window

Thus, all clipping decisions can be made without floating point division

- fp division only used when intersection point is required as a new end point of a shortened line segment inside the window

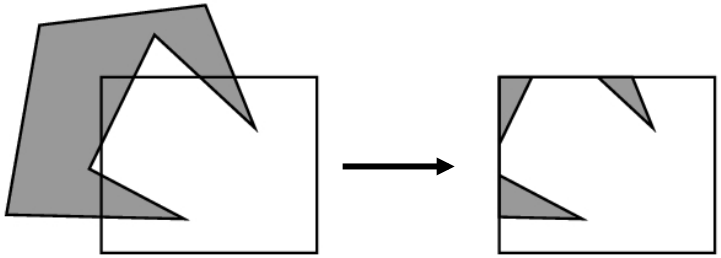
Avoids multiple intersection & shortening of line segments of Cohen-Sutherland

Other approaches to 2D line clipping do not extend readily to 3D

Polygon Clipping

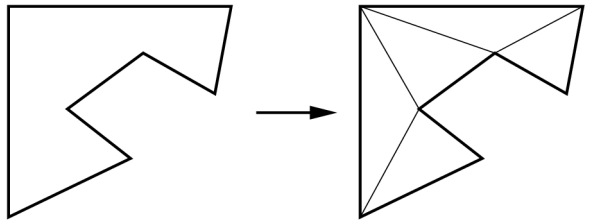
Required for: - view window clipping
- polygon-polygon clipping for shadows/hidden-surface/anti-aliasing

Clipping of concave polygons can generate multiple clipped polygons



Clipping convex polygons gives a single convex polygon

Therefore, **tesselate** concave polygon before clipping
- OpenGL GLU library includes tesselation



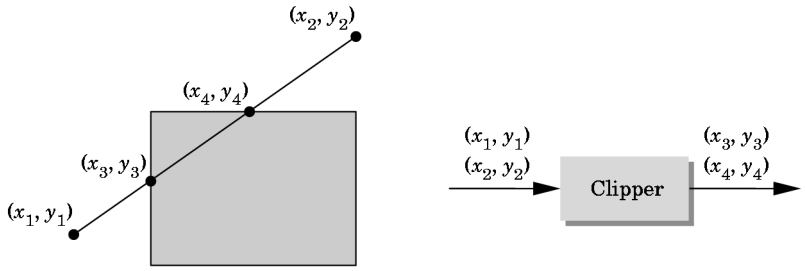
Assuming a concave polygon & a rectangular clipping region
- 2d line clipping algorithms (Cohen-Sutherland, Liang-Barsky)
  can be applied for each polygon edge to determine polygon clipping

Blackbox Clipping

A line-segment clipper can be treated as a blackbox:

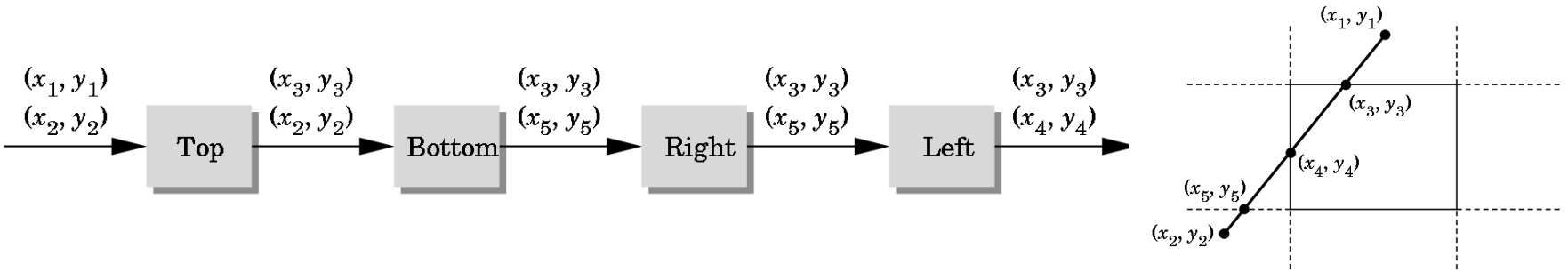
input: pair of verticies (line end-points)

output: pair of verticies for clipped line segment or nothing



Consider the 4 sides of the view-window as independent

- subdivide the clipping into 4 blackbox clippers each clipping against a single line (infinite line corresponding to view window side)

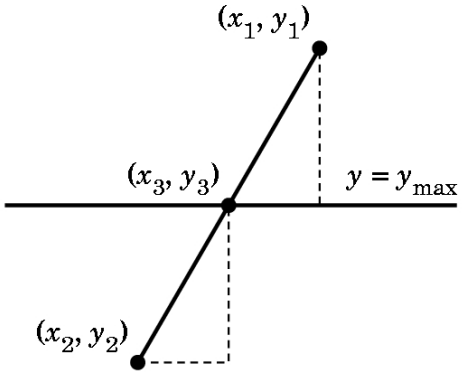


Blackbox Line Clipping

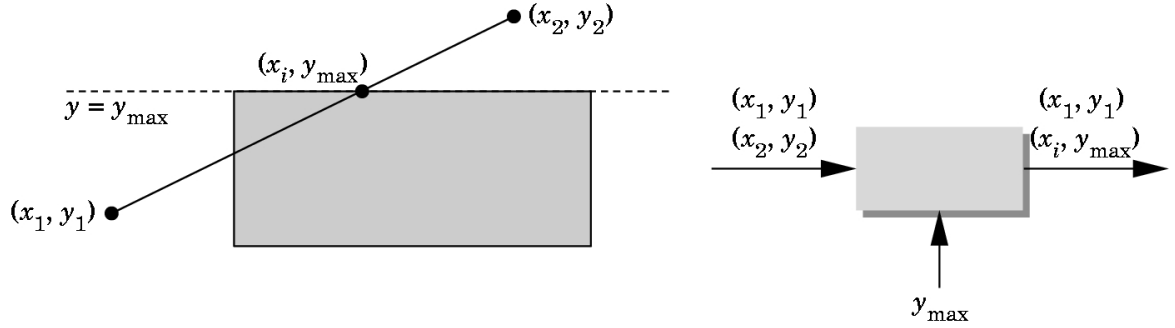
Clipping against a single line is achieved by computing the intersection for the side $y=y_{max}$

$$x_3 = x_1 + (x_2 - x_1)\frac{(y_{\max} - y_1)}{(y_2 - y_1)}$$

$$y_3 = y_{\max}$$



Can consider the clipper as a blackbox with y_{max} as an input parameter

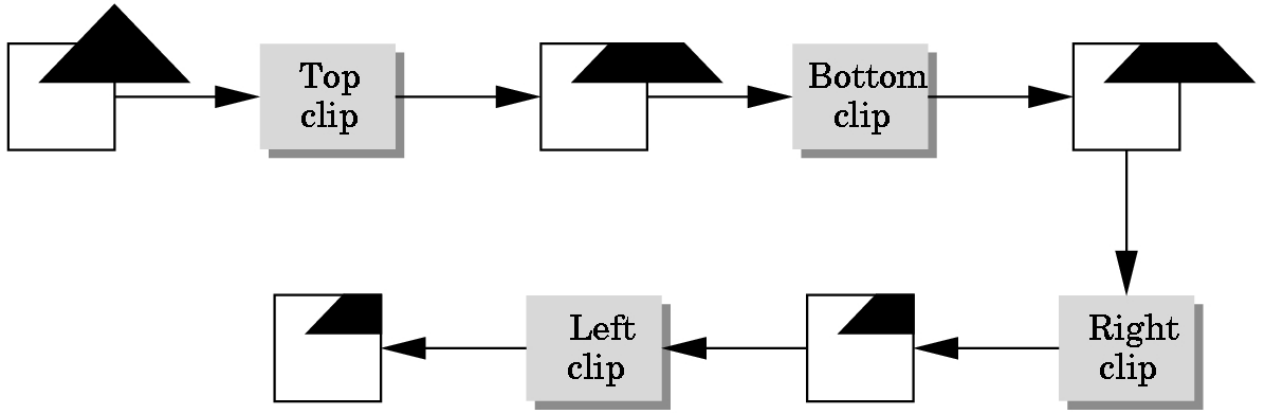


Combining multiple line clippers allows us to clip against each side of the view volume

- this is done at the expense of multiple floating point divisions

Example: Pipeline clipping of a polygon

The line corresponding to each side of a polygon are clipped successively to produce the final clipped polygon:

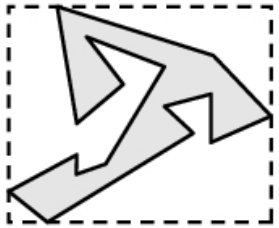


Bounding Box

For complex polygons/meshes and other curved surface primitives we often use the bounding box as an initial test for visibility culling

- often pre-compute & store the bounding box for complex primitives

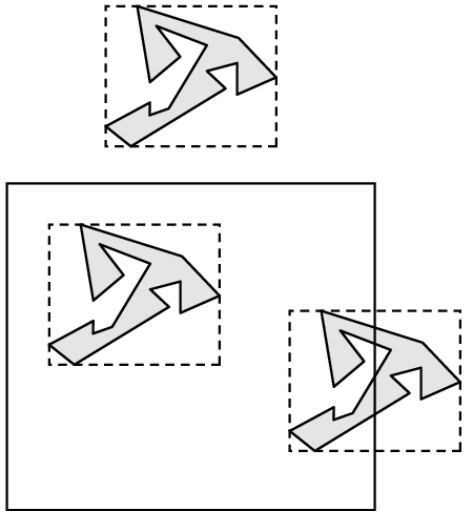
Bounding box is the limits of a primitive with respect to the axis of a co-ordinate system



- in 2D the smallest rectangle enclosing the primitive who's sides are aligned with the co-ordinate system
- in 3D the smallest parallelepiped enclosing the primitive

From the bounding box it is very simple to compute the possibility of intersection

- test if the bounding box is inside the view window

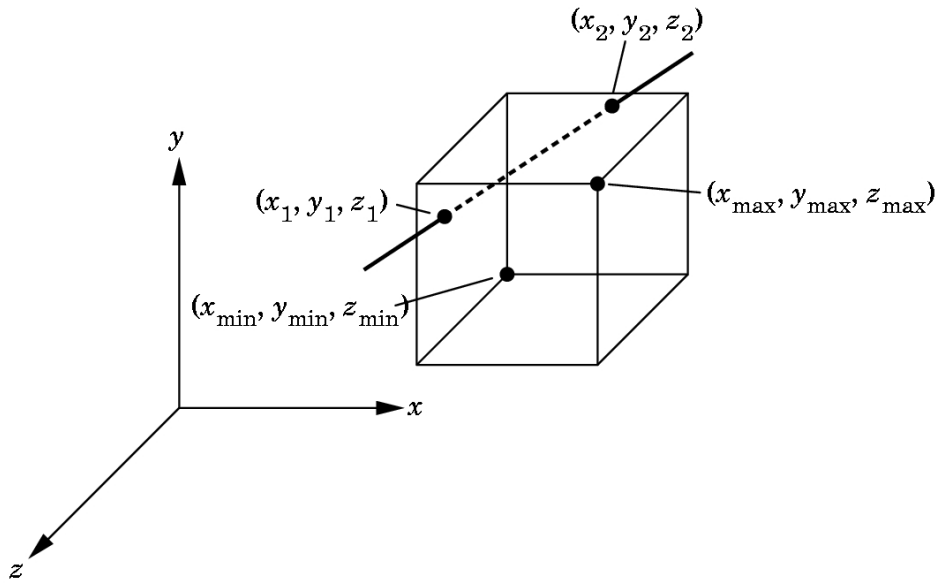


Clipping in 3D

Clip against a volume rather than 2D planar region

- clipping volume is a right parallelepiped

$$x_{\min} \le x \le x_{\max}$$
$$y_{\min} \le y \le y_{\max}$$
$$z_{\min} \le z \le z_{\max}$$



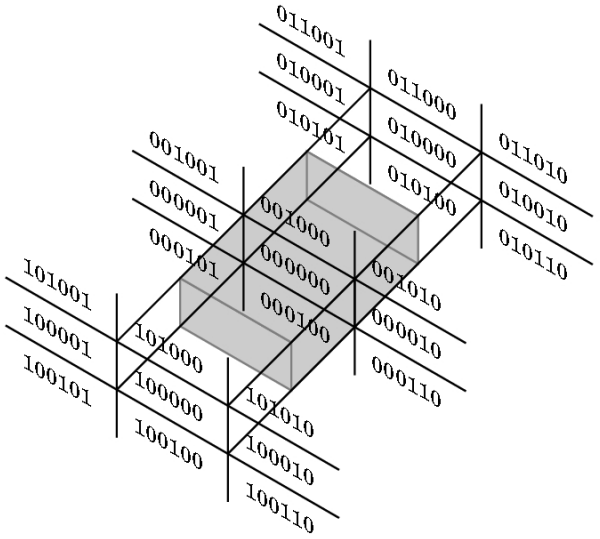
3D clipping performed by extending 2D algorithms to 3D

- Cohen-Sutherland or Liang-Barsky
- major difference in 3D is we clip lines against surfaces

Clipping in 3D II

Cohen-Sutherland in 3D

- 6 bit outcode for each line endpoint
- 27 possible end point locations
- include infront/behind clip volume
- evaluate line-plane intersections
- algorithm the same as 2D



Liang-Barsky in 3D

Add the parameteric expression for the z-component of the line:

$$z(\alpha) = (1-\alpha)z_1 + \alpha z_2$$

Consider the order of 6 intersection points with parameters $\alpha_1 ... \alpha_6$

- each line inside the volume has a maximum of 2 intersections
- use same logic as in 2D case base on intersection order

Blackbox clipping

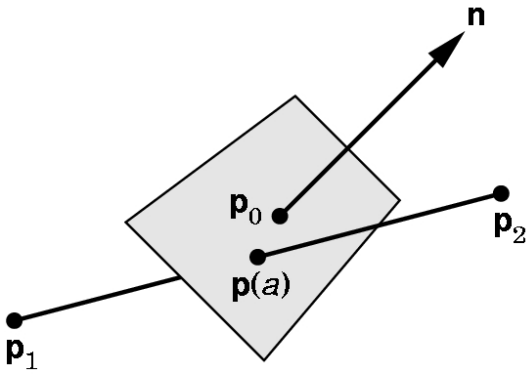
- Add additional clipping limits for z_{min}, z_{max}

Line-Surface Intersection

In 3 dimensions can write the equations for a line $l=(p_1,p_2)$ and plane $[n,p_0]$ as:

$$p(\alpha) = (1-\alpha)p_1 + \alpha p_2$$

$$n \cdot (p(\alpha) - p_0) = 0$$



where n is the plane normal p_0 is a point on the plane

The intersection of the line and the plane $p(\alpha)$ is given by:

$$\alpha = \frac{n \cdot (p_0 - p_1)}{n \cdot (p_2 - p_1)}$$

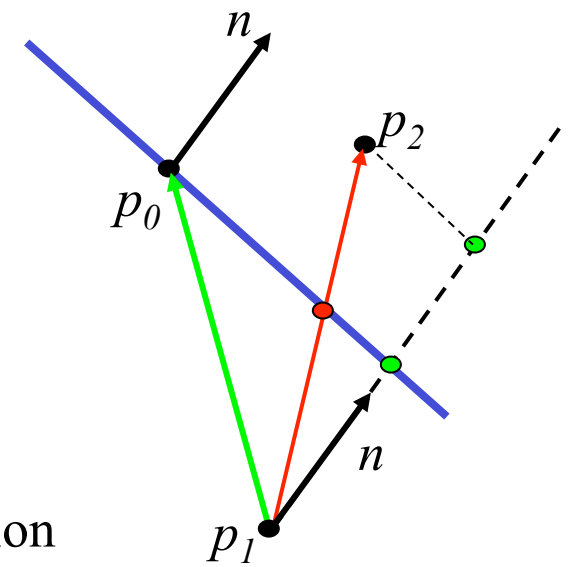
proof: by definition

$$\alpha = \frac{|\, p(\alpha) - p_1 \,|}{|\, p_2 - p_1 \,|} = \frac{n \cdot (p(\alpha) - p_1)}{n \cdot (p_2 - p_1)}$$

and from the normal constraint:

$$n \cdot (p(\alpha) - p_1) = n \cdot (p_0 - p_1)$$

QED.



α can be computed with 6 multiplications + a division
- or 1 division for standard view volume

Clipping in 3D III

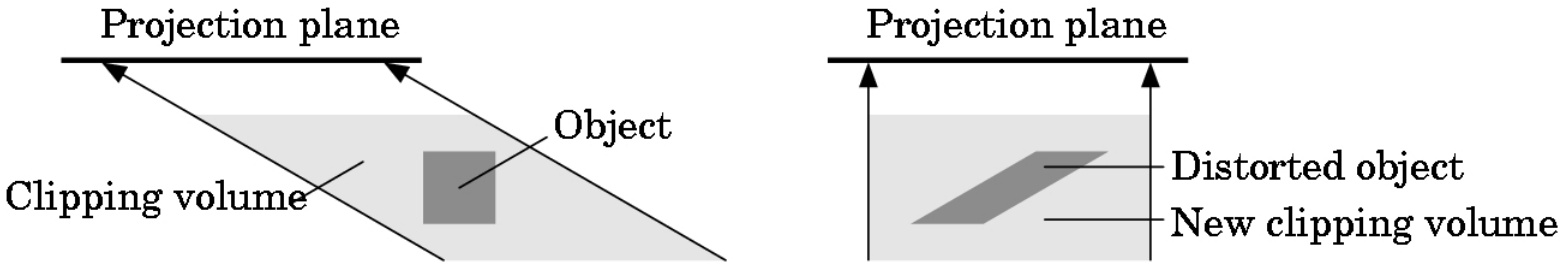
Standard view volume is a right parallelapiped

- each line-plane intersection can be computed with single division

Normalised view volume for perspective projection transforms the geometry to give a standard view volume

- simple computation of clipping
- followed by orthographic projection
- overall projection cost is the same

Demonstrates the importance of the normalisation process



Hidden-Surface Removal

Hidden-surface removal or 'visible-surface determination' determines the set of objects which are visible or obscured from a particular viewpoint

- after transformation to normalised view-volume & clipping to remove objects outside the view volume frustum

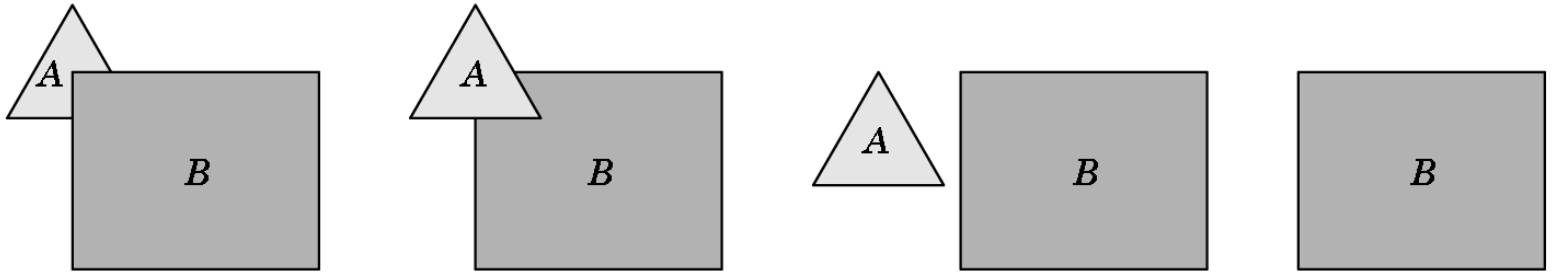
Consider only objects composed of planar polygons

Two approaches: **object-space or image-space**

Object-Space Hidden-Surface Removal

Generic Approach: consider objects pairwise gives 4 cases

4 cases: 1. A completely obscures B - display A only
2. B completely obscures A - diplay B only
3. A&B don't overlap - display A & B
4. A&B partially overlap - calculate visible parts



For a scene of *k* 3D opaque polygons each is treated as a separate object

1. Pick one of *k* polygons & compare to *k-1* other polygons
   (i) Test visibility
   (ii) Render visible region
2. Recursively repeat 1. with another polygon and compare k-i others

Complexity $O(k^2)$ - only possible with few polygons

Depth Sort and Painter's Algorithm

A common object-space algorithm for hidden-surface removal

Painter's Algorithm

If we have an ordered collection of polygons sorted by distance to viewer
Back-to-front rendering:

- paint farthest polygon completly
- recursively paint next farthest polygon until you reach front

2 questions:

(i) How do we sort polygons
(ii) What to do if polygons overlap

Depth Sort order polygons by how far from viewer their maximum z-value is

Problem when z-range overlap:
- can not render complete
  polygons in order

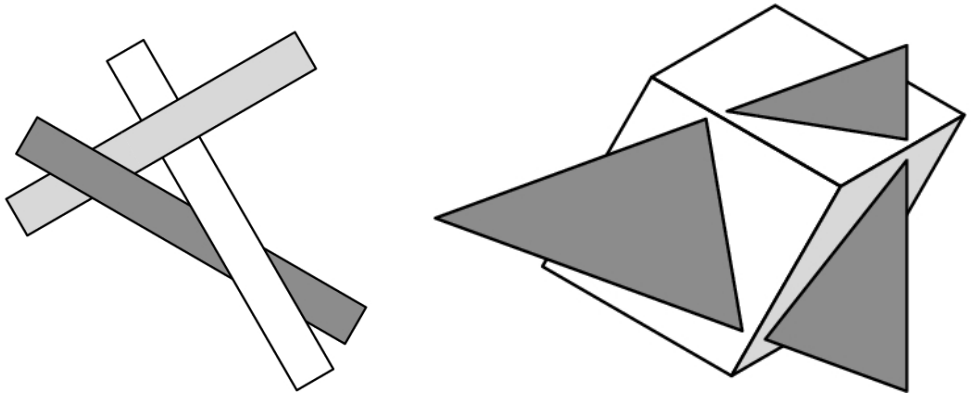


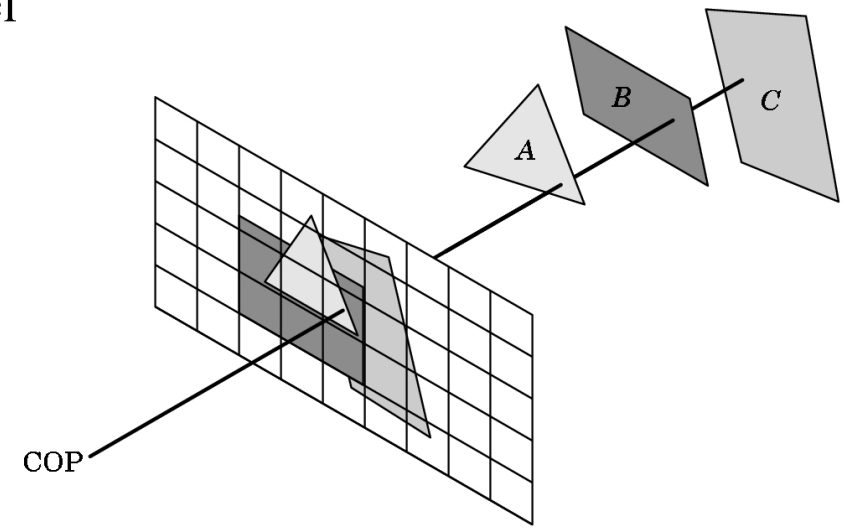
Image-Space Hidden Surface Removal

Generic Approach:

Ray leaves center-of-projection and passes through pixel

(i) intersect ray with planes of each of k polygons

(ii) determine intersection closest to centre of projection & colour pixel



Computation is order image size *(nxm) x k*

Giving *O(k)* complexity

- image-space approach is efficient vs. object-space
- but results in more jagged edges (due to pixel based sampling)

Highly efficient pipeline implementation using z-buffer

Z-buffer Algorithm

- most widely used Hidden-Surface removal algorithm
- image-based approach
- easy to implement in hardware as part of graphics pipeline
- small additional cost to standard rasterisation process
- Catmull 1975

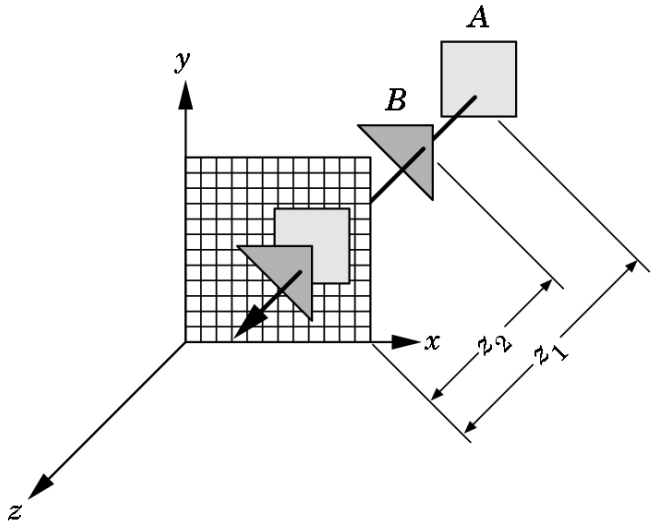
z-buffer is an array of pixels the same size as frame-buffer

- each pixel stores distance to the nearest polygon
- initialize z-buffer values to maximum depth

Rasterize polygon-by-polygon:

(i) for each pixel inside projected polygon store depth to nearest polygon

(ii) update depth and colour only if polygon intersection is closer $z_{poly} < z_{buffer}$



Z-buffer Algorithm II

OpenGL uses z-buffer for hidden-surface removal

- application must **explicitly** initialize the z-buffer for each new image generation

Polygon is part of a plane:

$$ax + by + cz + d = 0$$

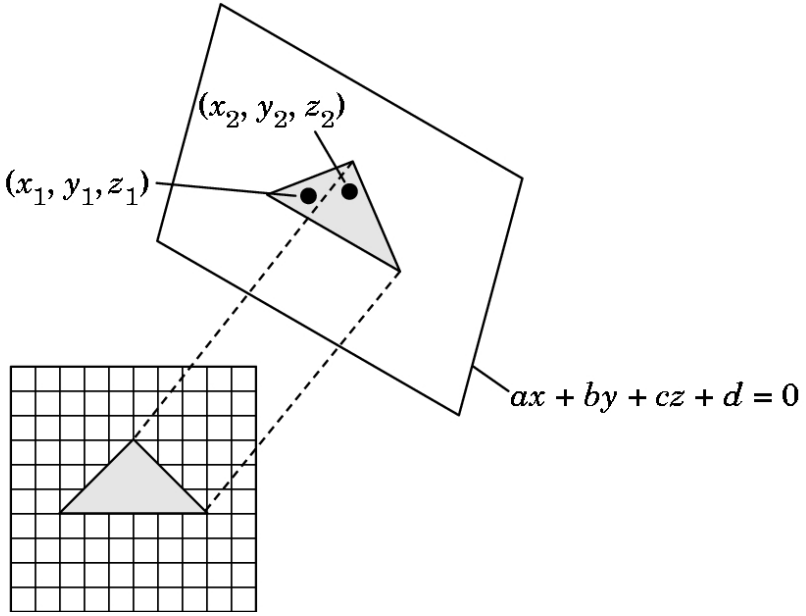
If (x_1,y_1,z_1) and (x_2,y_2,z_2) are two points on the polygon we can define the plane by the differential form:

$$a\Delta x + b\Delta y + c\Delta z = 0$$

$$\Delta x = (x_2 - x_1)$$

$$\Delta y = (y_2 - y_1)$$

$$\Delta z = (z_2 - z_1)$$



If we move along a **horizontal scan-line** in the image plane ***y=const*** then we have:

$$\Delta z = -\frac{c}{a}\Delta x$$

This is a constant which is only computed once per polygon

- therefore, **scan-line conversion using the z-buffer gives efficient HSR**

Back-face Removal

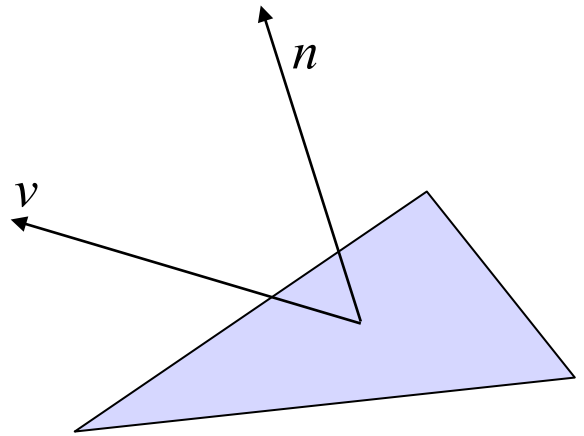
Prior to z-buffering we can apply to remove all back facing polygons

- reduces the no. of polygons to be rendered
- back facing polygons are generally not visible

Test for back facing polygons if angle between view direction v and normal n if:

$$n \cdot v \geq 0$$

If test is applied in normalised device coordinates need only check the sign of the z-component of the polygon normal



Scan-line algorithm

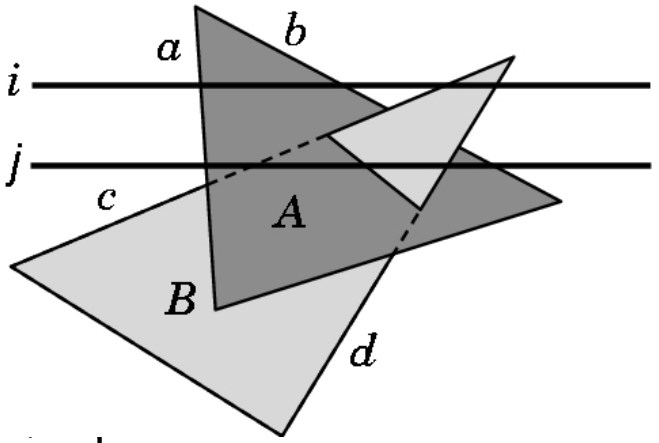
Rasterize polygons by scan-line

- dominant algorithm before z-buffer
- combines polygon conversion with hidden-surface removal
- fundamentally different to z-buffer
- requires a sophisticated data structure but lower memory cost than z-buf.

Consider 2 intersecting polygons

if we rasterize the polygon scanline by scanline we can incrementally compute the depth (as in z-buffer algorithm)

To achieve efficient evaluation we store an ordered list of edges for each scanline



Scan line i as we traverse left-to-right:

(i) cross edge a-b (only 1 poly. so no depth)
(ii) cross edges c-d (only 1poly. so no depth)

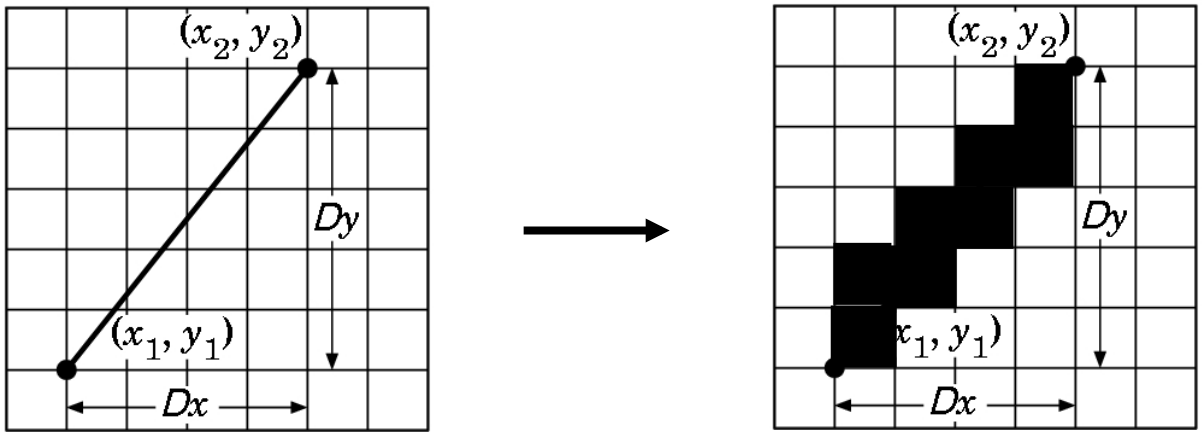
Scan line j:

(i) cross a-c (only 1 poly. no depth)
(ii) cross c-d 2 polys so depth computation required
(iii) cross d-b (only 1 poly.so no depth)

Scan Conversion

Scan-conversion converts continuous line or polygon representation to a discrete set of pixel samples.

- Conversion from geometric primitives to image pixels in the frame-buffer
- Starting point is a line or polygon specified by a set of vertices $v_i = (x_i, y_i)$ in screen co-ordinates
- Frame buffer is an *n*x*m* array of pixels



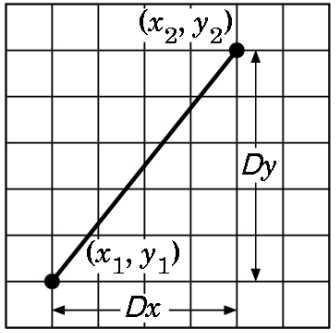
Rasterisation process is independent of display

- most frame-buffers have dual ported memory allowing simultaneous read/write allowing display to be written at required rate

Digital Differential Analyser (DDA) Algorithm

DDA is the simplest scan-coversion algorithm

- the name comes from an early electro-mechanical devices for digital simulation of differential equations
- because deriviative of a line = slope m ($y=mx+b$) generation of a line segment is equivalent to numerical solution of a simple differential equation



For a line segment defined by 2 end-points the slope m is:

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$$

Assuming $0<m<1$ (other values of m handled by symmetry)

We want to **determine the pixels intersected by the line segment**

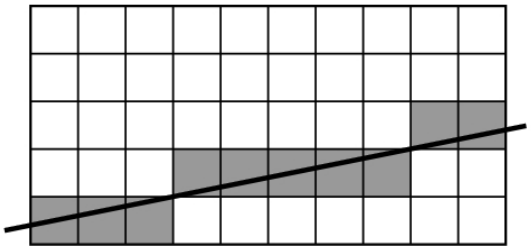
For any change in x the change in y is given by:

$$\Delta y = m\Delta x$$

As we move from x_1 to x_2 we increase x by increments of 1

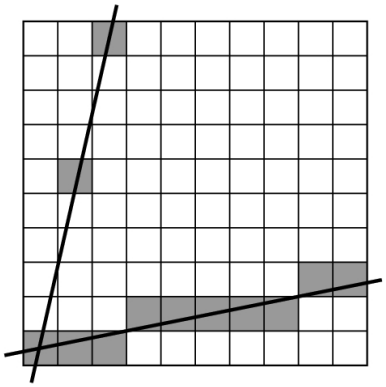
$$\Delta y = m$$

m is real so round $y+nm$ to nearest pixel & fill



Digital Differential Analyser (DDA) Algorithm II

Problem with $y+nm$ for $m>1$ separation between pixels in y direction is greater than 1 resulting in gaps (slope greater than 45 degrees)

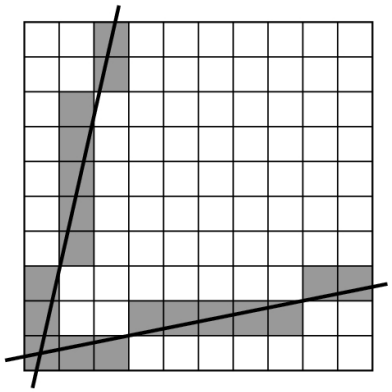


Solution is to reverse x and y and increment y by units of 1

$$\Delta x = \frac{1}{m}$$

$$\Delta y = 1$$

- resulting in all pixels being filled



Similar approach can be used for m<0,m<-1

Bresenham's Algorithm

DDA is simple and easily coded but requires floating-point addition for each pixel

Bresenham 1963 derived a line-rasterisation that uses no floating-point calculation
- standard algorithm used in hardware & software rasterisation

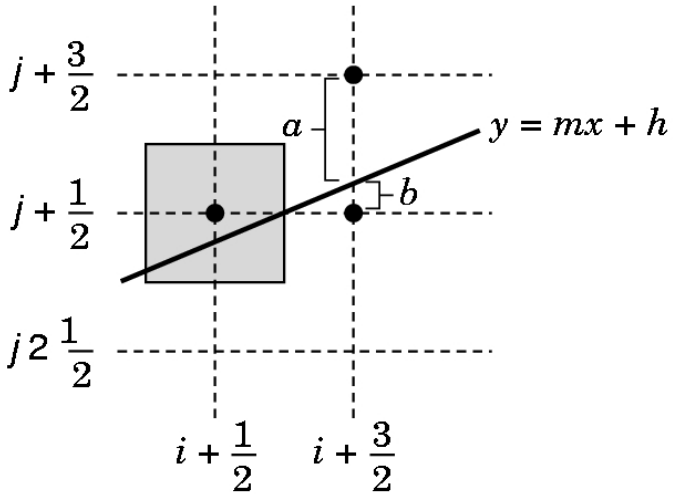
Assume line goes between integer endpoints (x_1y_1) (x_2y_2) and slope $0 \le m \le 1$
- slope condition is critical for algorithm

For a pixel along the line $(i+\frac{1}{2}, j+\frac{1}{2})$

Now consider the column $x = i+\frac{3}{2}$

the line must intersect

$$(i+\tfrac{3}{2}, j+\tfrac{1}{2}) or (i+\tfrac{3}{2}, j+\tfrac{3}{2})$$



We can reduce the decision to: $d=a-b$

$$y = \begin{cases} j+\frac{1}{2} & d > 0 \\ j+\frac{3}{2} & d < 0 \end{cases}$$

Therefore, we have reduced the decision to a single variable d

Bresenham's Algorithm II

Bresenham showed we can make the decision without floating point operations

(1) replace floating-point operations with fixed point

(2) apply algorithm incrementally

(1) Replace *d* with $d = (x_2 - x_1)(a - b) = \Delta x(a - b)$

All terms in this expression are integers

Proof:

substituting for *a* and *b* using $y = mx + h \quad m = \frac{\Delta y}{\Delta x} \quad h = y_2 - mx_2$

$$y_{\frac{3}{2}} = m(i + \tfrac{3}{2}) + h = \frac{\Delta y}{\Delta x}(i + \tfrac{3}{2}) + y_2 - \frac{\Delta y}{\Delta x} x_2$$

$$a = (j + \tfrac{3}{2}) - y_{\frac{3}{2}}$$

$$b = y_{\frac{3}{2}} - (j + \tfrac{1}{2})$$

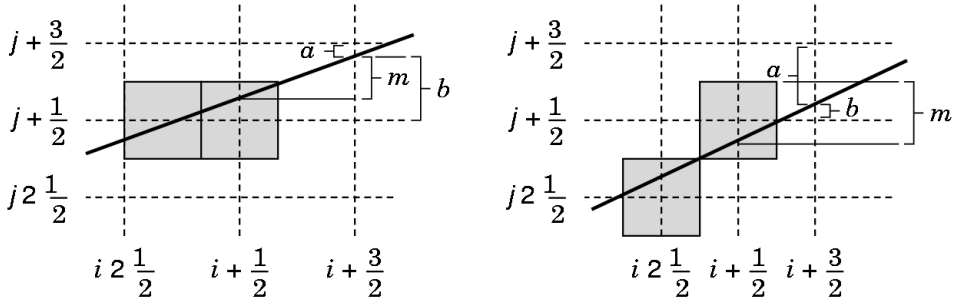
$$a - b = 2(j + 1 - y_{\frac{3}{2}})$$

$$\Delta x(a - b) = 2\Delta x(j + 1 - y_2) + \Delta y(2x_2 - 2i - 3)$$

all terms in this expression are integers QED.

Bresenham's Algorithm III

(2) Apply algorithm incrementally



Suppose d_k is the value at $x=k+1/2$
What is d_{k+1}

2 situations:

(i) y value was incremented at previous step
(ii) y value was same at previous step

- a increases by m only if y was increased at previous step
  otherwise a decreases by m-1
- likewise for b: decreases by -m if incremented otherwise increases 1-m

Multiplying by change in x gives possible increase:

$$d_{k+1} = d_k + \begin{cases} 2\Delta y & if\,(d_k > 0) \\ 2(\Delta y - \Delta x) & otherwise \end{cases}$$

Calculation for each pixel requires only 1 addition + sign test

Bresenham's algorithm gives very efficient scan conversion of a line to pixels

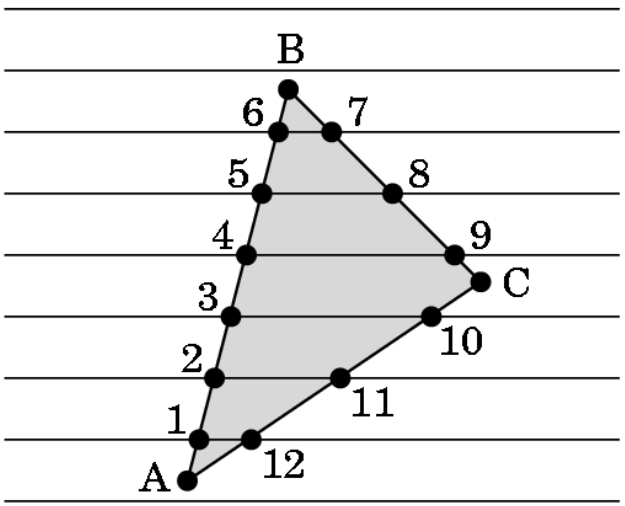
- standard algorithm for scan conversion of lines

Scan Conversion of Polygons

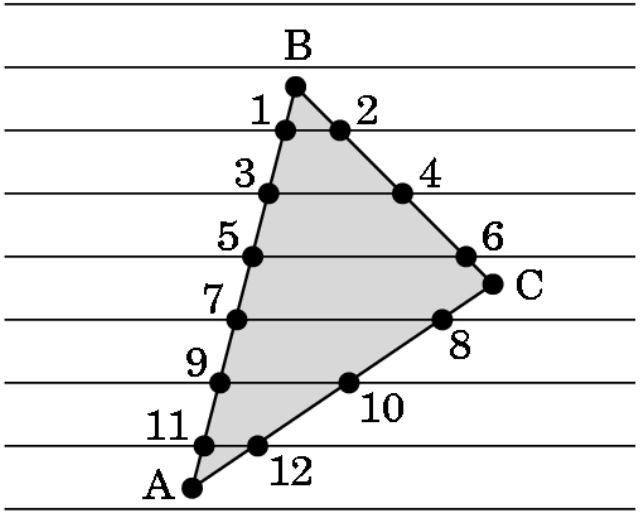
Scan-line algorithm

- Scan along horizontal lines in frame-buffer
- Rasterise polygon edges using Bresenham line scan algorithm
- convex polygons only

Edge crossing with scan lines



Scan-lines to be rasterised



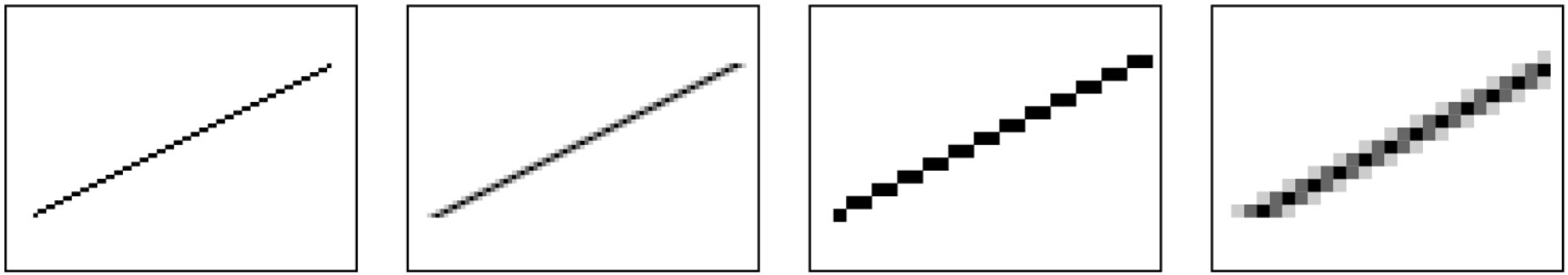
Fill sections along scan lines 1-2, 3-4 ,5-6.....

Pipeline process for filling polygons.

Aliasing

Rasterisation results in jagged lines/polgyons

- spatial aliasing (a,c)
- antialiasing: smooth edges of line to remove step change



(a) (b) (c) (d)

- approximating a continuous line by discrete approximation

Nyquist Sampling Theorem: to represent a continuous function must sample at twice the highest frequency

- For a grid **sampling frequency** is 1/(grid spacing)

Nyquist frequency is 1/(2 x grid spacing)

Summary

- Steps in implementing a renderer
  - conversion from continous coordinates to discrete raster

- Clipping
  - Cohen-Sutherland - binary outcodes
  - Liang-Barsky - line intersection order
  - Polygon Clipping - pipeline operations
  - Bounding boxes

- Scan Conversion
  - Bresenham's Algorithm - lines
  - Scan Line Algorithms
  - Aliasing