UNIT 8:

IMPLEMENTATION.

Syllabus

* Basic Implementation strategies.
* Major tasks
* clipping
* line-segment clipping
* polygon clipping
* clipping of other primitives
* clipping in three dimensions.
* Rasterization.
* Bresenham's algorithm.
* polygon rasterization.
* Hidden surface removal
* Antialiasing
* Display considerations

– 8 Hours.

BASIC IMPLEMENTATION STRATEGIES

* There are two strategies that are followed
  - object oriented approach
  - image oriented approach.

object oriented approach.

* Here the outer loop is over the objects:-

```
for (each-object)
{ render (object);
}
```

* vertices are defined by programs and flow through a sequence of modules that transforms them, colors them and determine whether they are visible or not.

* This approach uses a pipeline that contains hardware for each of the tasks. Data flows forward through the system as shown -

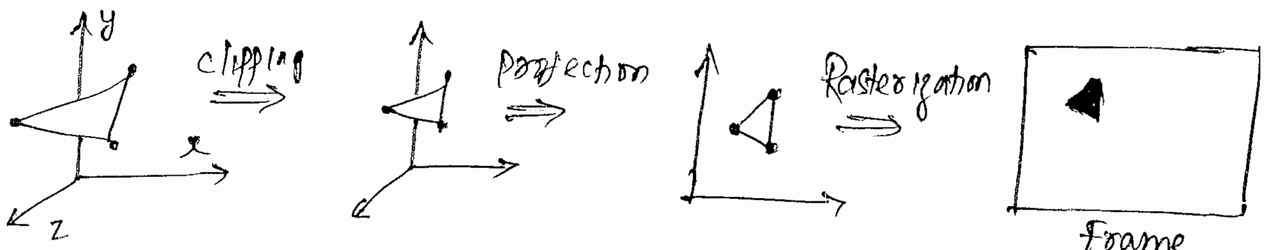


fig: object oriented approach.

Adv

1. Availability of geometric parallel pipelined processors makes processing simple since they can process millions of polygons per second.

Disadv

1. They cannot handle most global calculations. Because each geometric primitive is processed independently and in arbitrary order, complex shading effects that involve multiple objects cannot be handled efficiently.

Image oriented approach.

* Here, loop is over pixels. In pseudocode, the outer loop of such a program is of following form -

```
for (each-pixel)
  {   assign-a-color (pixel);
  }
```

Adv.

we need only limited display m/m at any time. Because results do not vary greatly from pixel to pixel. this coherenc can be used to develop incremental forms of algorithm.

Disadv

Complicated datastructures would be required.

FOUR MAJOR TASKS

* There are four major tasks that any graphics system must perform to render ~~to~~ a geometric entity -

1. Modeling
2. Geometry processing
3. Rasterization.
4. Fragment processing.

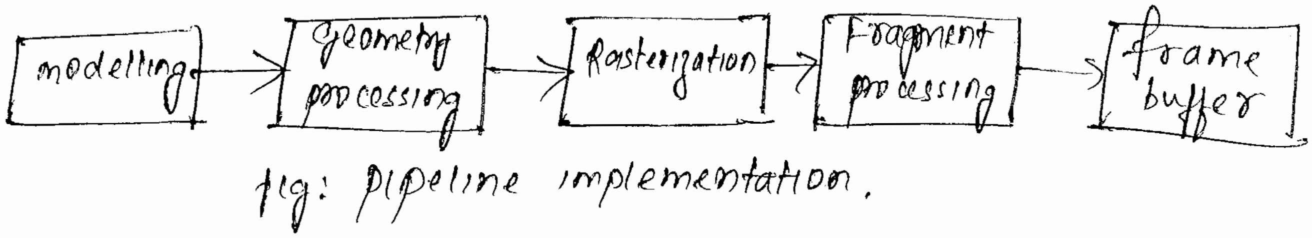


fig: pipeline implementation.

note: These tasks should be performed in both the approaches.

Modeling

* It refers to processing a set of vertices that specify a primitive (geometric object)
* modelers are black boxes that produce geometric objects and are usually interactive in nature.

Geometric processing

* modeling results in set of vertices that specify a group of geometric objects.
* geometric processing works with these vertices.
* Goal of geometry processor are to determine which geometric objects can appear on the display and to assign shades or colors to the vertices of these objects.
* four processes are required - projection, primitive assembly, clipping, and shading.
* First step in geometry processing is to change representation from ~~for~~ object coordinates to camera / eye coordinates.
* second step is to transform vertices using the projection transformation

Rasterization.

* For line segments, rasterization determines which fragments should be used to approximate a line seg.
* For polygons, rasterization determines which pixels lie inside the polygon.

Fragment processing.

* It refers to the process of hidden surface removal in case of opaque objects and blending of colors of pixels in case of translucent objects.

CLIPPING.

* Clipping is the process of determining which primitives, or parts of primitives fit within the clipping or view volume defined by the application program.
* In general, portions of all primitives that can be displayed lies within -

$$-w \leq x \leq w$$
$$-w \leq y \leq w$$
$$-w \leq z \leq w$$

LINE SEGMENT CLIPPING.

* A clipper decides which primitives or parts of primitives can possibly appear on the display and are passed onto the rasterizer.
* The two main line-segment clipping algorithm are -
    1. Cohen-Sutherland clipping algorithm
    2. Liang-Barsky clipping algorithm

note: primitives that fit within the specified view volume are "accepted" (ie pass through the clipper)
primitives that cannot appear on the display are eliminated, or "rejected", or culled.

Cohen-sutherland clipping

# The two dimensional clipping problem for line segments is shown in below fig.

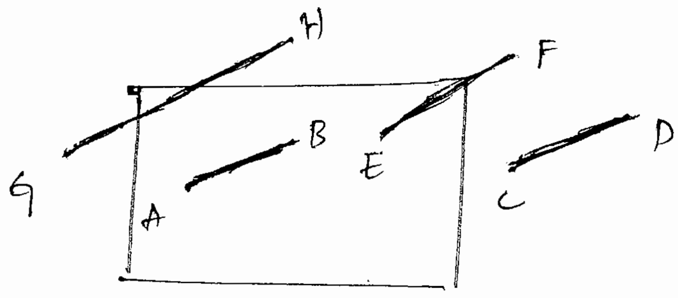
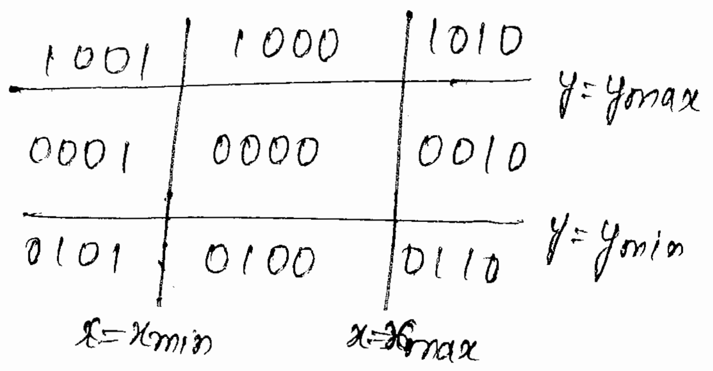


fig: Two-dimensional clipping.

# Algorithm starts by extending the sides of window to infinity, thus breaking up space into the nine regions as shown.



# Each region can be assigned a unique 4-bit binary no. or <u>outcode</u>, $b_0 b_1 b_2 b_3$ as follows -

suppose that (x, y) is a point in the region, then

$$b_0 = \begin{cases} 1 & \text{if } y > y_{max} \\ 0 & \text{otherwise} \end{cases} \qquad b_1 = \begin{cases} 1 & \text{if } y < y_{min} \\ 0 & \text{otherwise} \end{cases}$$

$$b_2 = \begin{cases} 1 & \text{if } x > x_{max} \\ 0 & \text{otherwise} \end{cases} \qquad b_3 = \begin{cases} 1 & \text{if } x < x_{min} \\ 0 & \text{otherwise} \end{cases}$$

# For each end point of a line segment, we first compute the end point's outcode.

(It may require 8 floating point subtractions per line segment)

* Consider a line segment whose outcodes are given by O_1 = Outcode (x_1, y_1) and O_2 = outcode (x_2, y_2)

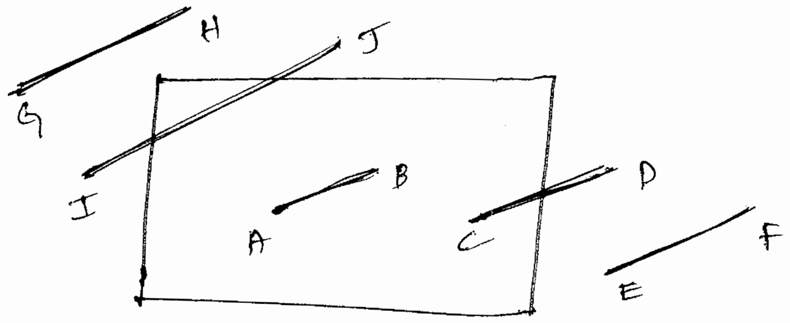


fig: cases of outcodes in cohen-sutherland algorithm.

There are 4 cases -

case 1:

If $(O_1 = O_2 = 0)$ then both end points are inside the clipping window (as in AB) and hence the segment can be sent on to be rasterized.

case 2:

If $(O_1 \neq 0, O_2 = 0$ or vice-versa) then one end point is inside the clipping window & the other is outside (as in CD) the line segment must be shortened.

case 3:

If $(O_1 \,\&\, O_2 \neq 0)$ then by taking bitwise AND of the outcodes, we determine whether or not the two end points lie on the same outside side of the window. If so, the line segment can be discarded (see EF in above fig)

case 4:

If $(O_1 \,\&\, O_2 = 0)$ then both end points are outside, but they are on the outside of different edges of the window. (see GH & IJ). We cannot tell from just outcodes whether the line segment can be discarded or ~~not~~ must be shortened. ∴ we intersect with one of the sides of the window & check the outcode of the resulting point.

Adv

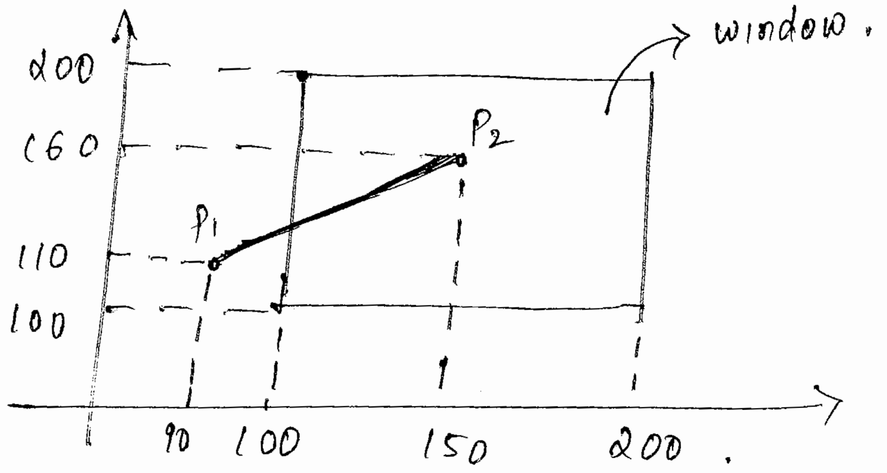
1. Avoids floating point division operations.
2. can be extended to three dimensions.

Disadv.

1. It must be used recursively.
2. It requires clipping window in a rectangular fashion.

Example:

consider below scenario:



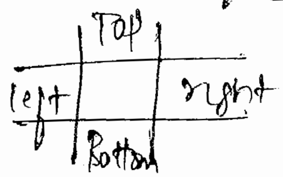
Steps to be followed

Step 1: Calc. outcodes for both end points.

step 2: if (P1 || P2 == 0) then the line is completely inside the window. ie line is accepted.

step 3: if (P1 && P2 > 0) reject the line (ie o/s)
else if (P1 && P2 == 0) then the line is partial.

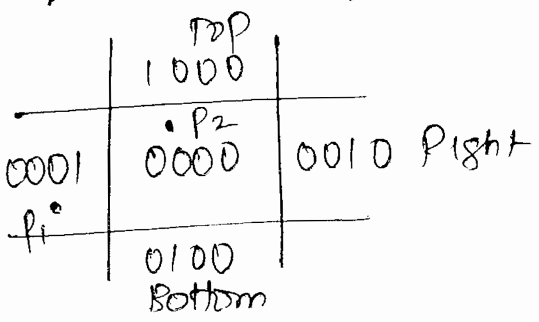
note: if (P1 && top > 0) The point is above the window.
if (P1 && bottom > 0) — " — below — " —
if (P1 && left > 0) point is less than xmin
if (P1 && right > 0) — " — more — " — xmax.



Step 1: Calculate the outcodes for both end points -

$P_1 = 0001$

$P_2 = 0000$



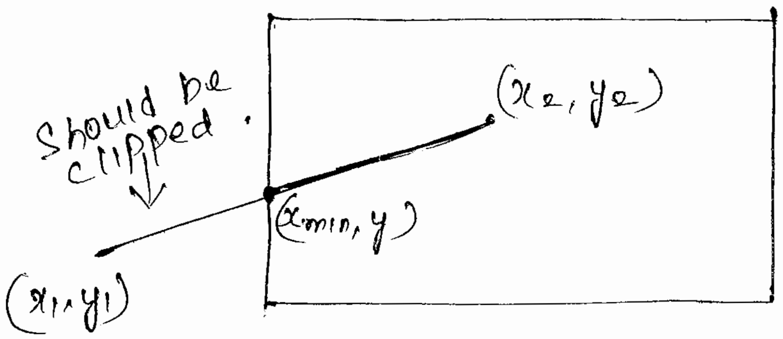
Step 2:

$P_1 \,||\, P_2 = 0001$, which is not equal to 0.

Step 3: $P_1 \,\&\&\, P_2 =$

$$\begin{array}{r} 0001 \\ 0000 \\ \hline 0000 \end{array}$$

∴ the line is partial.



Here, we need to find $y = ??$

~~formula~~:

$p1 \,\&\&\, top = 0000$

$p1 \,\&\&\, bottom = 0000$

$p1 \,\&\&\, left = 0001$ ∴ left portion should be clipped.

formula: $y = y_1 + \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_{min} - x_1)$, $x = x_{min}$

$= 110 + \frac{50}{60} \times 10$

$\Rightarrow \boxed{y = 118.3}$ $\boxed{x = 100}$

note:

* for clipping left portions:

~~y=~~ $x = x_{min}$

$$y = y_1 + \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_{min} - x)$$

* for clipping right portions:

$x = x_{max}$.

$$y = y_1 + \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_{max} - x_1).$$

* For clipping Top portions:

$y = y_{max}$

$$x = x_1 + \left(\frac{x_2 - x_1}{y_2 - y_1}\right)(y_1 - y_{max}).$$

* For clipping bottom portions:

$y = y_{min}$

$$x = x_1 + \left(\frac{x_2 - x_1}{y_2 - y_1}\right)(y_1 - y_{min})$$

Liang-Barsky clipping. (more efficient)

* makes use of the parametric form of lines.
* Suppose that we have a line segment defined by the two end points $p_1 = [x_1, y_1]^T$ and $p_2 = [x_2, y_2]^T$
* we can write the parametric form of this line segment as -

$$p(\alpha) = (1-\alpha) P_1 + \alpha P_2 \qquad 1 \geq \alpha \geq 0 .$$

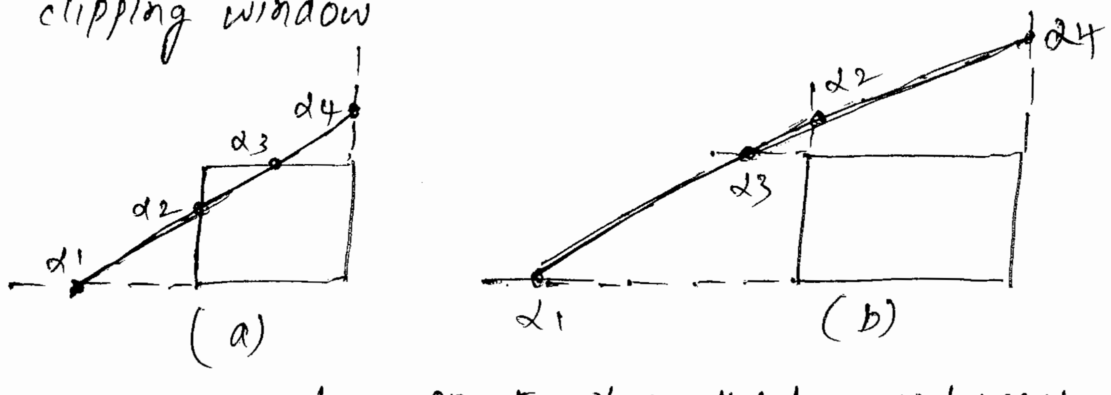
or as two scalar eqns -

$$x(\alpha) = (1-\alpha) x_1 + \alpha x_2$$
$$y(\alpha) = (1-\alpha) y_1 + \alpha y_2 .$$

As the parameter α varies from 0 to 1, we move along the segment from P_1 to P_2

$\alpha = -ve \Rightarrow$ yields points on the line on other side of P_1
$\alpha > 1 \Rightarrow$ yields points on the line past P_2, going off to ∞.

* Below fig shows two cases of parametric line & a clipping window



There are four points where the line intersects the extended sides of the window. These points correspond to the four values of the parameters : $\alpha_1, \alpha_2, \alpha_3$, & α_4. (order is very imp)

$\alpha_1 \rightarrow$ bottom $\qquad \alpha_3 \rightarrow$ top
$\alpha_2 \rightarrow$ left. $\qquad \alpha_4 \rightarrow$ right

* In fig (a)

$1 > \alpha_4 > \alpha_3 > \alpha_2 > \alpha_1$

ie line intersects right, top, left, bottom in order

$\Rightarrow$ ~~Accept~~, shorten.

* In fig (b)

$1 > \alpha_4 > \alpha_2 > \alpha_3 > \alpha_1$

ie line intersects right, left, ~~bo~~ top, bottom

$\Rightarrow$ Reject.

note:

To determine the intersection with top of the window, we find the intersection at the value -

$$\alpha = \frac{y_{max} - y_1}{y_2 - y_1} \quad -(1)$$

Similar eqns holds for other 3 sides of the window.

(1) can be written as -

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

$$\boxed{\alpha \Delta y = \Delta y_{max}}$$

POLYGON CLIPPING.

* It is not as simple as line segment clipping.
- clipping a line segment yields atmost one line segment.
- clipping a polygon can yield multiple polygons.
  However clipping a convex polygon can yield atmost one other polygon.

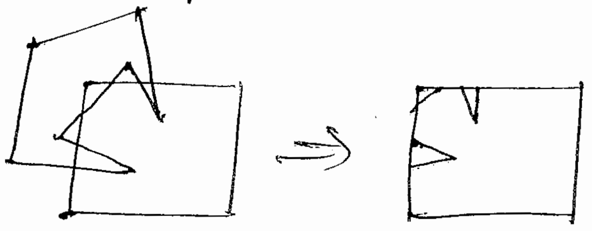
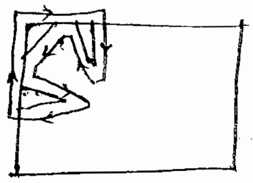


fig: clipping of a concave polygon.



It yields multiple polygons. Sol^n: assume them or create one single polygon as shown above (using traversing)
disadv: still complex.

* Therefore, we replace concave polygons with a set of triangular polygons. This process is called tessellation (ie divide a ~~con~~ given polygon into set of convex polygons)

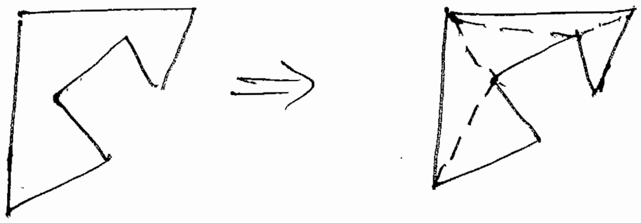
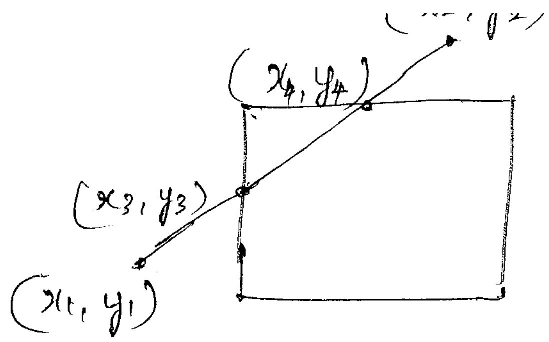


fig: Tessellation of a concave polygon.

* line segment clipping can be considered as (or envisioned as a black box whose i/p is the pair of vertices from the segment to be tested and clipped, and whose o/p either is a pair of vertices corresponding to the clipped line segment or is nothing if the input line segment lies ~~o/w~~ o/s the window.



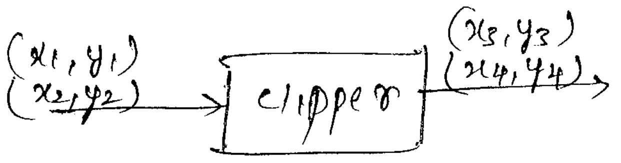
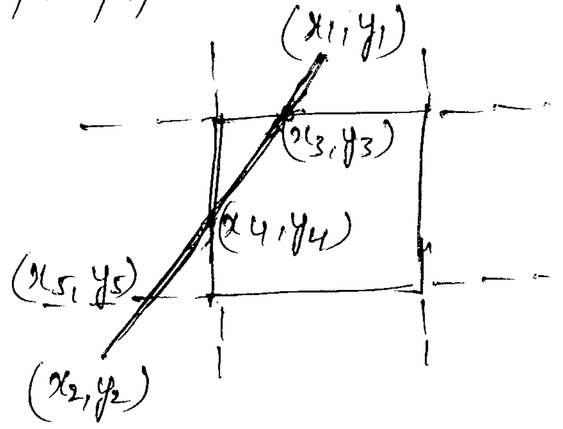
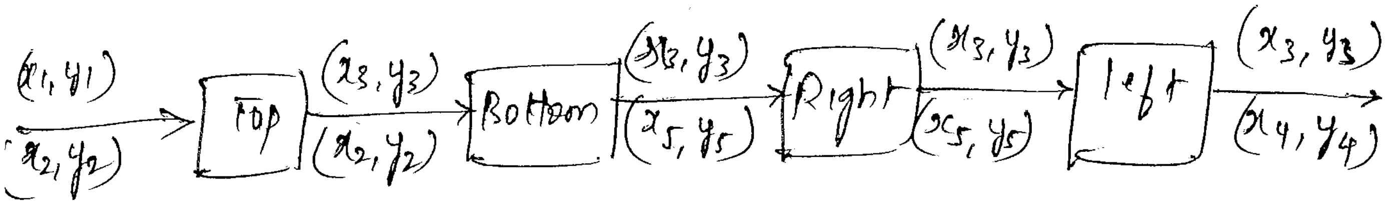


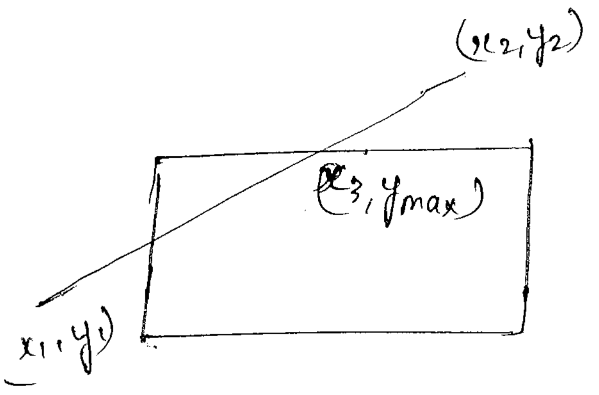
fig: clipper as a black box.

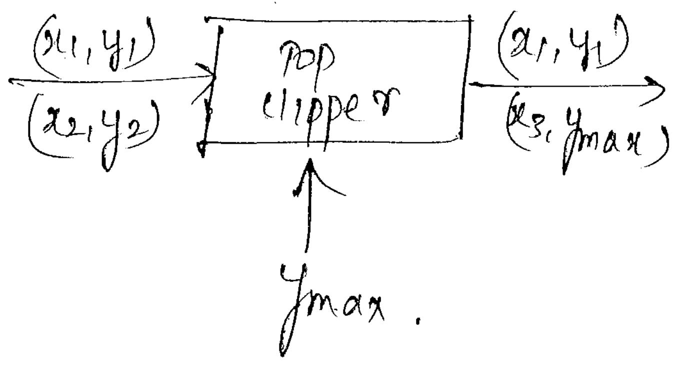
* clipping against each side of window is independent of other sides.
we can use four independent clippers arranged in a ~~pipepl~~ pipeline -





* for eg: top clipper would be like -





* Fig below shows a simple example of the effect of successiv clippers on a polygon.

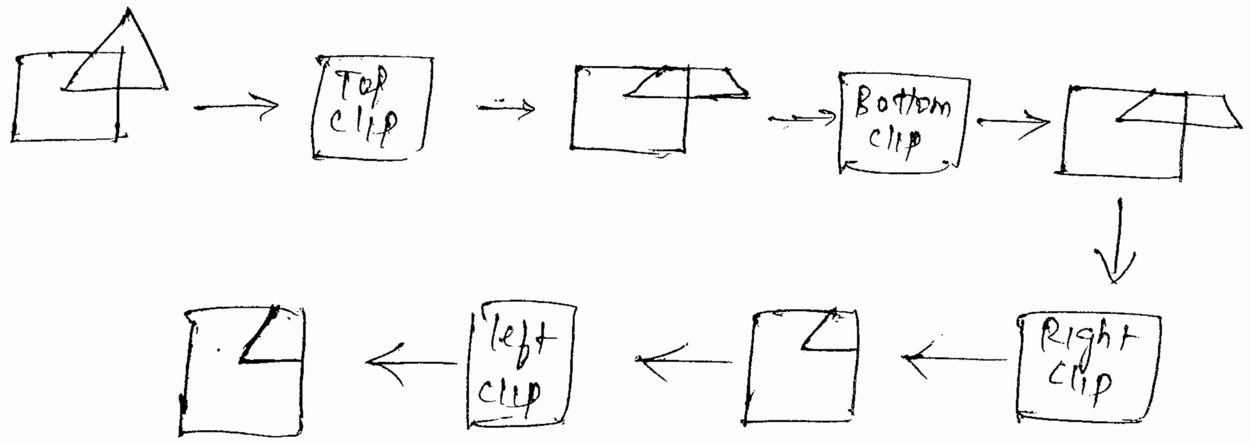
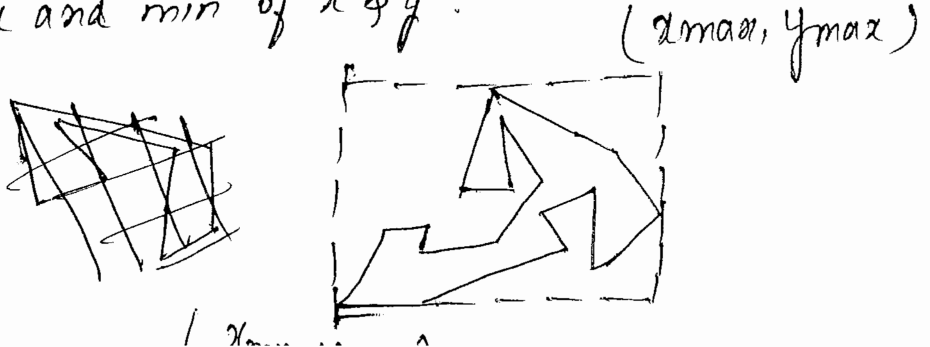


fig: pipe line clipping of polygons.

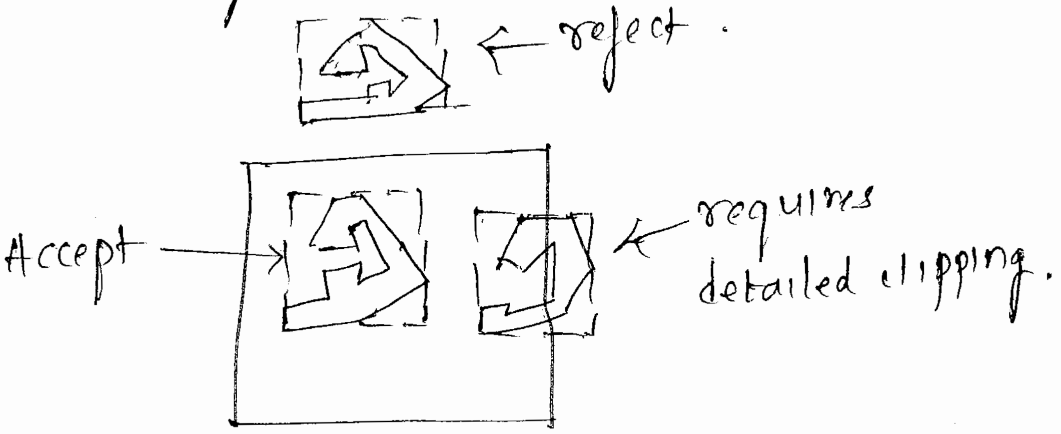
note: for three dimensional clipping - Add front and back clippers. It results in small increase in latency.

CLIPPING OF OTHER PRIMITIVES.

* Suppose we have an many sided (or complex) polygon, In such a case, Bounding boxes technique can be used for clipping purpose.
* Axis-aligned bounding box or extent of a polygon is the smallest ~~rectangular~~ rectangle (aligned with the window) that contains the polygon.
* Its very simple to compute the bounding box: just find max and min of x & y.



* we can usually determine accept/reject based only on bounding box.



RASTERIZATION.

* For line segments, rasterization determines which fragments should be used to approximate a line segment. for polygons, rasterization determines which pixels lie inside the polygon.
* ie Rasterization produces a set of fragments. Each fragment has a location in screen coordinates that corresponds to a pixel location in the color buffer.

DDA Algorithm

* simplest scan conversion algorithm for line segments is DDA (Digital Differential Analyzer) algorithm. DDA was an early electromechanical device for digital simulation of differential equations.
* Because a line satisfies the differential equation -

$\frac{dy}{dx} = m$ where $m \rightarrow$ slope,

generating a line segment is equivalent to solving a simple differential equation numerically.

* Suppose that we have a line segment defined by the end points (x_1, y_1) and (x_2, y_2). The slope is given by -

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

we assume that $0 \leq m \leq 1$. we can handle other values of m using symmetry.

note: color buffer is an $n \times m$ array of pixels. pixels can be set to a given color by a single function inside the graphics implementation of the following form -

```
write_pixel (int ix, int iy , int value);
```

value → either index in color-index mode or a pointer to an RGBA color.

* Our algo. is based on writing a pixel for each value of ix in write-pixel as x goes from x_1 to x_2.
If we are on the line seg as shown in below fig, for any change in x equal to Δx, the corresponding change in y must be -

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

As we move from x_1 to x_2, we increase x by 1 each iteration; thus we must increase y by -

$$\Delta y = m.$$

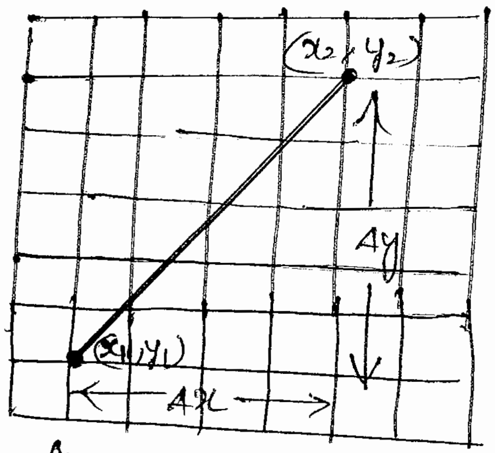


fig: line seg in window coordinates.

* Although x is an integer, y is not, Because 'm' is a floating point number. Therefore we must round it to find the appropriate pixel as shown -

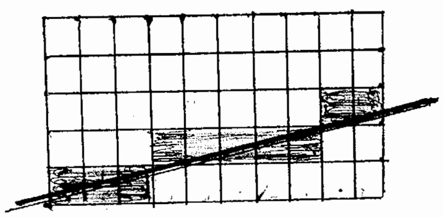


fig: pixels generated by DDA Algorithm.

Disadv: many floating point Calculations.

* our algorithm in pseudocode is -

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

problem with DDA algorithm

* Reason that we limited the max slope to 1 can be see from the fig shown -
ie for large slopes, the separation b/w pixels that are colored may be large, generating unacceptable approximation of line segment. ∵ our algo is of this form - for each x, find the best y.

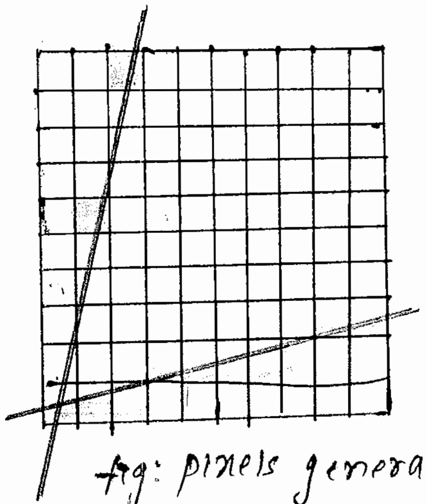
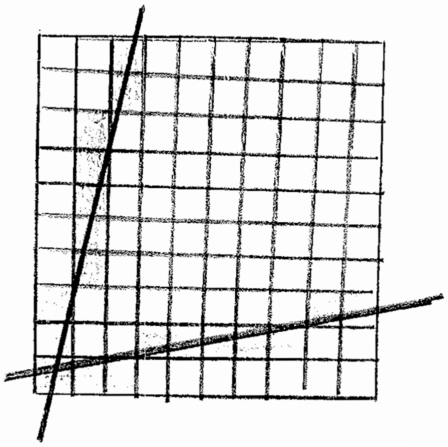


fig: pixels generated by high & low slope lines.

Soln :

For m > 1, we swap the roles of x and y. The algo becomes this - for each y, find the best x. ⟶

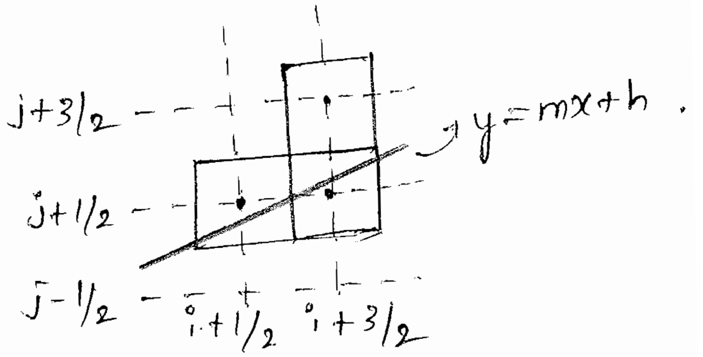


BRESSNHAM'S ALGORITHM.

* DDA Algorithm requires a floating point addition for each pixel generated.
* we can eliminate all floating point calculations through Bresenham's algorithm.
* consider end points of a line segment as (x_1, y_1) & (x_2, y_2) and the slope, $0 \leq m \leq 1$.
* Assume pixel centers are at half integers.

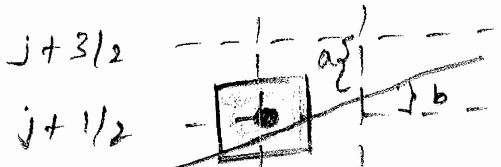
If we start at a pixel that has been written, there are only two candidates for the next pixel to be written into the frame buffer.

* Ref fig, suppose that we are somewhere in the middle of the scan conversion of our line segment & have just placed a pixel at, $(i+1/2, j+1/2)$.



we KT $y = mx + h$.

* The slope condition indicates that we must set the color of one of only two possible pixels - either the pixel at $(i+3/2, j+1/2)$ or the pixel at $(i+3/2, j+3/2)$. we use decision variable $d = b - a$ for this. where b and a are distances b/w the line and the upper & lower candidate pixels at $x = i + 3/2$ (refer below fig)



* If d is -ve, the line passes closer to the lower pixel, so we choose the pixel at $(i+3/2, j+1/2)$.
otherwise, we choose the pixel at $(i+3/2, j+3/2)$

Incremental form.

* more efficient if we look at d_K (value of decision variable at $x=K$)

$$d_{K+1} = d_K + \begin{cases} 2\Delta y & \text{if } d_K < 0 \\ 2(\Delta y - \Delta x) & \text{otherwise} \end{cases}$$

* The calculation of each ~~successiy~~ successive pixel in the color buffer requires only an addition an a sign test.
* this algo is so efficient: it has been incorporated as a single instruction on graphics chips.

CLIPPING IN THREE DIMENSION

* In 3D clipping, we clip against a bounded volume rather than against a bounded region in the plane.

The simplest extension of 2D clipping to 3D clipping is clipping for a right parallelopiped region.

The conditions are -

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

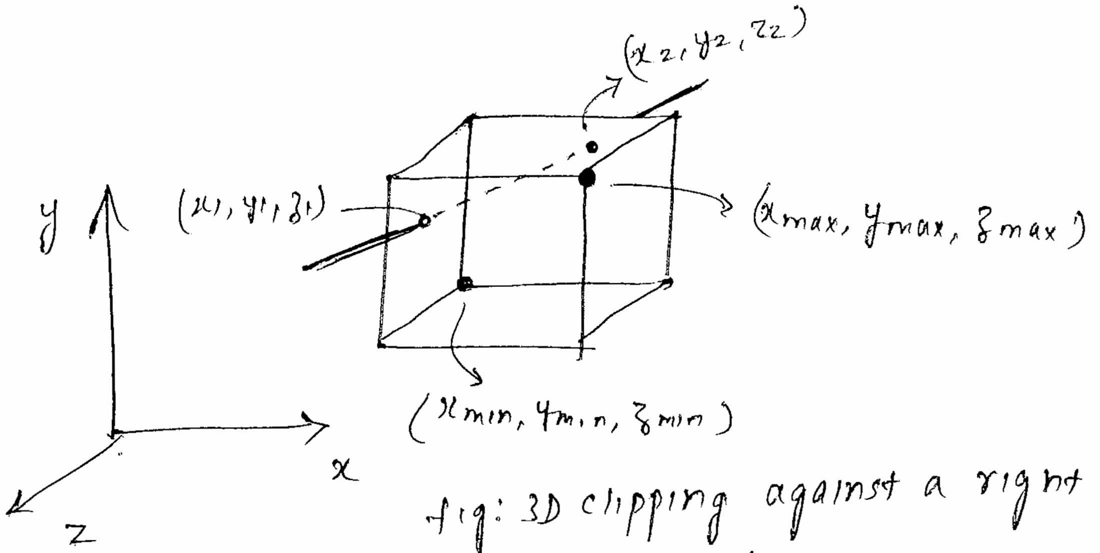
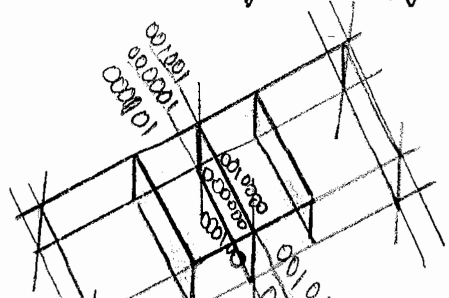


fig: 3D clipping against a right parallelopiped.

* Both cohen sutherland and Liang Barsky algorithms can be extended to clip in 3 Dimensions.

* For cohen sutherland algo, we replace the 4 bit outcodes with a 6 bit outcode. Additional two points are set if the points lie either in front of or behind the clipping volume. Testing strategy is identical ~~to 2D~~. for the 2D & 3D cases.



(refer fig in text

* for Liang Barsky algorithm, we add the equation
$z(\alpha) = (1-\alpha) z_1 + \alpha z_2$. to the existing eqns.-

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

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

we have to consider 6 intersections with the surfaces that forms the clipping volume.
pipeline clippers would add two more modules to clip against the front & back.

HIDDEN SURFACE REMOVAL.

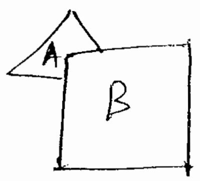
* Hidden surface removal refers to the process of removing the surface which would not be visible to the viewer from the display.

(or) Hidden surface removal (or visible surface determination) is done to discover what part (if any) of object in the view volume is visible to the viewer or is obscured from the viewer by other objects.

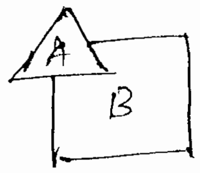
object space approach

* Here we consider the objects pair wise, as seen from the center of projection.

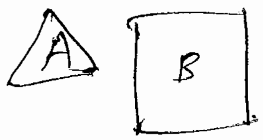
eg: consider two such polygons A & B. There are 4 possibilities (refer fig)



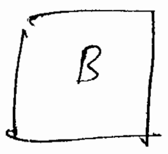
B partially obscures A



A partially obscures B



Both A & B are visible



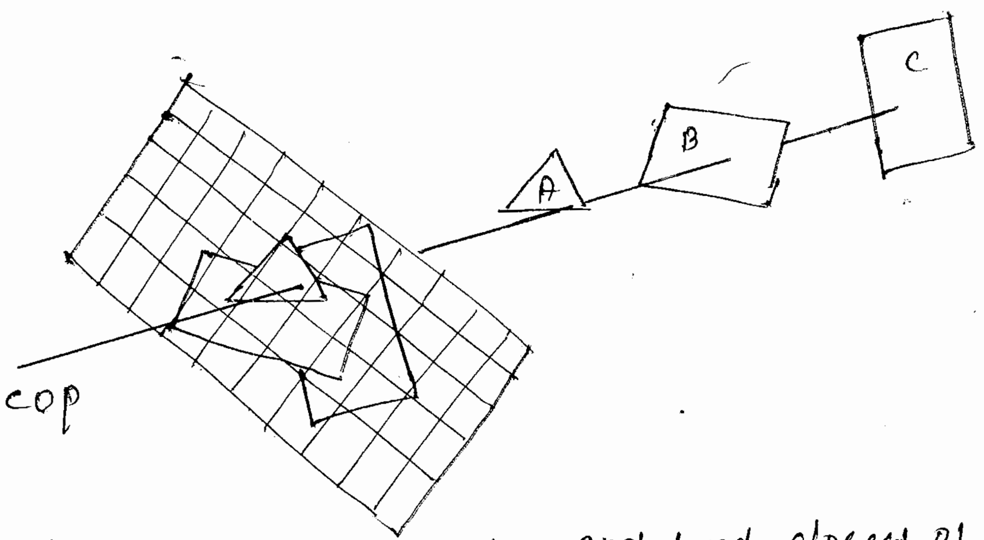
B totally obscures A.

* worst case complexity in this approach is $O(k^2)$
  ∵ given k polygons, we pick one of the k polygons and compare it pairwise with remaining k-1 polygons

* Ex for Hidden surface Algorithm techniques that uses object space approach are -
  1. painters Algorithm
  2. Depth sort
  3. Backface removal (culling)

image space approach.

* It follows our viewing and ray casting model (ref fig)

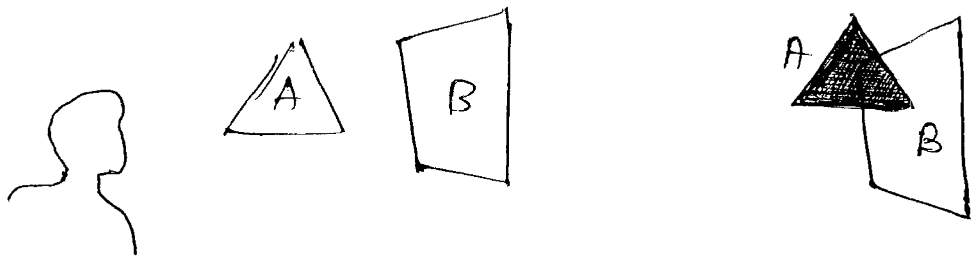


* idea: look at each projector and find closest of k polygons & color the corresponding pixel with appropriate shade.
* For n x m display (frame buffer), we have to carry out this operation ~~nm~~ nmk times, giving O(k) complexity.
* Ex for algorithms that uses this approach -

  1. Z buffer Algorithm
  2. scanline Algorithm.

painter's Algorithm

+ consider the polygons as shown in the fig -



+ We can see that the polygon in the front (ie A) partially obscures the other.
+ We can render this scene using painters algorithm as follows.

  Render the rear polygon first and then the front polygon, painting over the part of rear polygon not visible to the viewer in this process.
  Both polygons would be rendered completely, with the hidden surface removal being done as a consequence of the Back to front rendering of the polygons..

+ Two questions arises
  - How to do the sort ?? → ie which is far & which is nearer.
  - what ~~do s~~ to do if polygons overlap??

soln - Depth sort.

Depth sort.

+ First order all the polygons by how far away from the viewer their maximum z-value is.
+ suppose that the order is as shown in the fig.

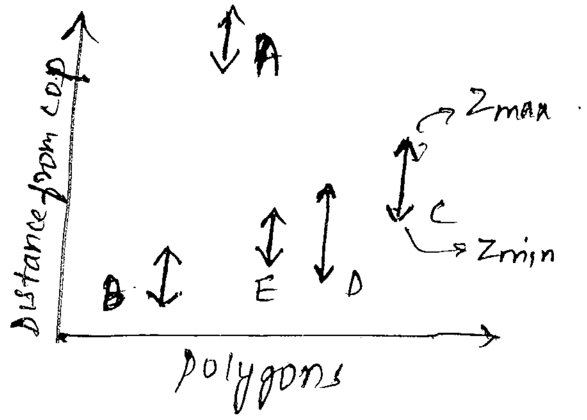
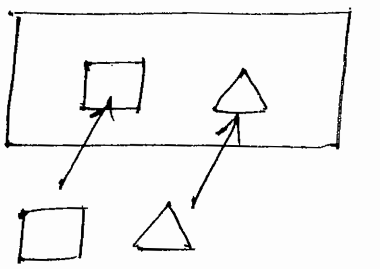
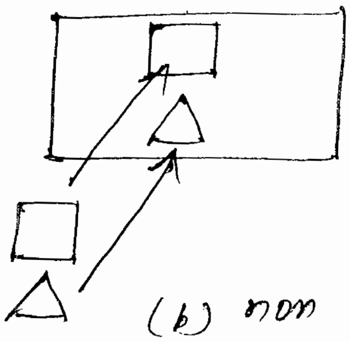


fig: z-extents of sorted polygons.

* We can see that polygon A is behind all the other polygons (ie not overlapping) and can be painted first. However, the others cannot be painted based solely on z-extent
* What to do if polygons overlap in z-extent? Consider a pair of polygons whose z-extents overlap and check their x and y-extents. (refer fig)


(a) non overlapping x-extents


(b) non overlapping y-extents.

* If either x or y ~~extends~~ extents do not overlap, neither polygon can obscure the other, & they can be painted in either order
* Even if these tests fail, it may be still possible to find an orde in which we can paint the polygons individually.

fig shows such a case. one is fully on one side of the other.

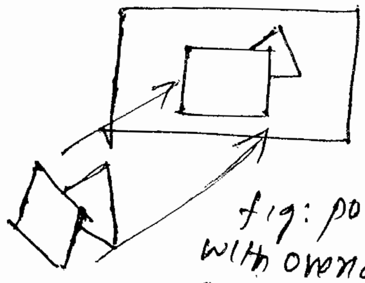
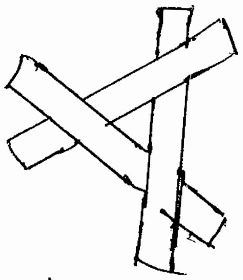
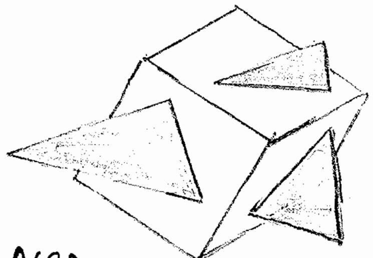


fig: polygon with overlapping extents.

* Two troublesome situation remain →


(a) cyclic overlap.


(b) piercing polygons

Back Face Removal (culling)

* The test for culling a back facing polygon can be derived from below fig

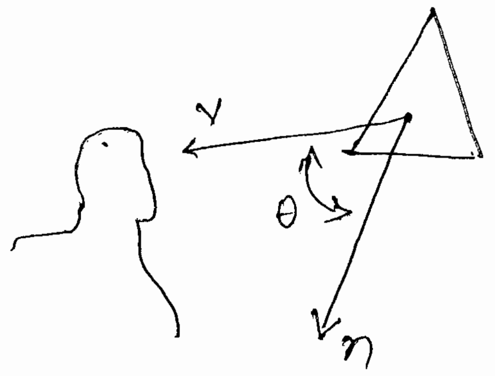


fig: Back face Test.

* If θ is the angle b/w the normal & the viewer, then the polygon is facing forword if and only if

$$-90 \le \theta \le 90 \quad \text{or}$$

equivalently, $\cos\theta \geq 0$.

or, $n \cdot v \geq 0$

↳ dot product.

* we can even simplify this test. in normalized device coordinates,

$$v = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Thus, if the polygon is on the surface,

$ax + by + cz + d = 0$ in normalized device coordinates,

we need ~~to~~ only to check the sign of c to determine whether we have a front or back facing polygon.

The Z-buffer Algorithm.

* most widely used.
* suppose that we are in the process of rasterizing one of the two polygons shown in the fig.

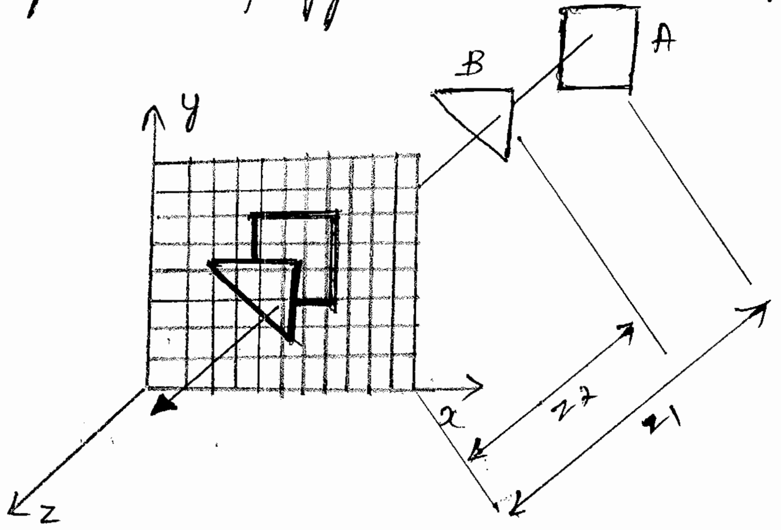


fig: The z-buffer algorithm

* use a buffer, the z-buffer with the same resolution as the frame buffer and with depth consistent with the resolution that we wish to use for ~~deep~~ distance.
  Initially each element in the depth buffer (z buffer) is initialized to a depth corresponding to the max. distance away from the COP.
  Color buffer is initialized to background color.

* Working: we rasterize polygon by polygon using any one of the methods.
  For each fragment on the polygon corresponding to the intersection of the polygon with a ray through a pixel, we compute the depth from COP.
  We compare this depth to the value in the z buffer corresponding to this fragment
  If this depth is greater than the depth in Z buffer, then we have already processed a polygon with a corresponding fragment

closer to the viewer, & this fragment is not visible.

If the depth is less than the depth in z-buffer, then we found a fragment closer to the viewer. we update the depth in the zbuffer & place the shade computed for this fragment at the corresponding location in the color buffer.

Efficiency.

+ suppose that we are rasterizing a polygon scanline by scanline the polygon is part of plane (ref fig) that can be represented as -

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

suppose that (x_1, y_1, z_1) and (x_2, y_2, z_2) are two points on the polygon.

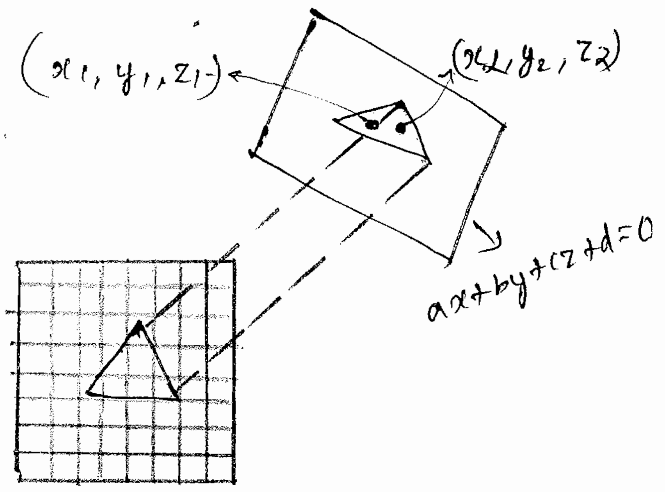
then eqn for plane can be written in differential form as -

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

where $\Delta x = x_2 - x_1$

$\Delta y = y_2 - y_1$

$\Delta z = z_2 - z_1$.



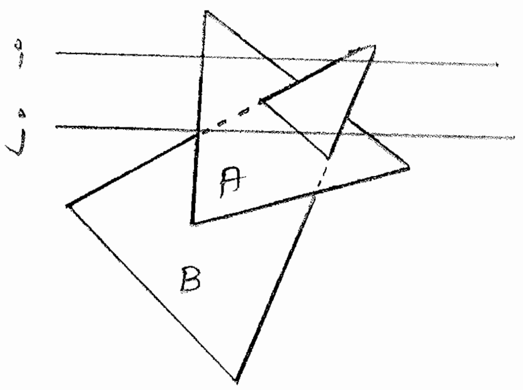
when we raster a polygon scanline by scanline, they $\Delta y = 0$ and ~~~~ we increase ~~~~ x in unit steps, so Δx is constant

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

This value is constant that needs to be computed only once for each polygon.

scan-line Algorithm

* can combine ~~sca~~ shading & hidden surface removal ~~ust~~ through scanline 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.