Q. Describe the 3D viewing pipeline.

→ A computer generated 3D objects is passed through a series of transformations to produce that object on the output screen. These transformations are divided into different stages and together creates an input-output pipeline.

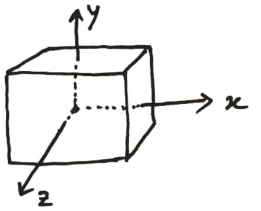
In each of these stages an object is described w.r.t. an stage specific coordinate system.

Local Coordinate system or modelling coordinate system (MCS):

An modelling software usually maintains a dedicated coordinate system for each object w.r.t. some reference point of that object. This helps to control local transformations on the object (independent of global positioning)

(i) The reference point can be the center of gravity or some corner etc.

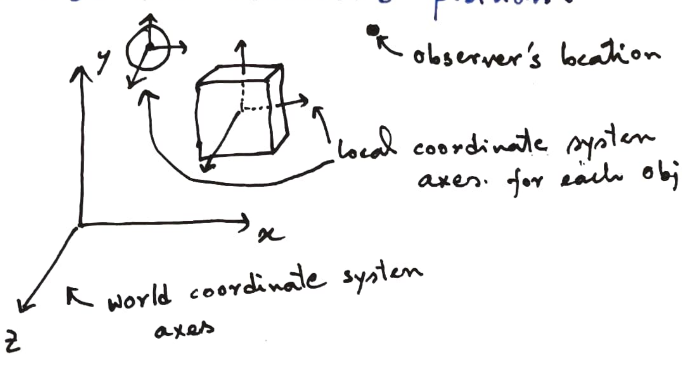
example: a cube having its center as the reference point.



Global coordinate system or world coordinate system (WCS):

As the name suggest it is the main coordinate system that is applied to the entire canvas/scenary. All objects (with their local transformations) are positioned according to some global ~~refer~~ point of reference. Here we also describe the observer's position.

(i) Example:

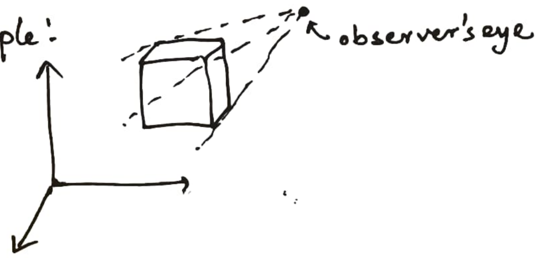


Viewing coordinate system (VCS) :

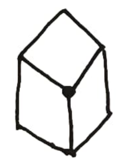
It describes the objects position and transformation w.r.t. to the observer's location.

This is analogus to moving the camera to the observer's position & present the scene from the point-of-view (POV) of the observer.

(i) Example:



⇒ (one possible overserver's view of the cuble)



Device coordinate (DC) system :

This describes the 2D (usually) coordinate system of the display screen or any other output devices (e.g. printer, plotter etc.). Here the points have integral coordinates and referts to a pixel on the screen.

Normalized device coordinate (NDC) system :

This device independent cartesian coordinate system (usually 2D) maps every point ~~the~~ in the normalized range i.e. between 0 and 1 $\begin{bmatrix} 0 \le x \le 1 \\ 0 \le y \le 1 \end{bmatrix}$

This plots/renders an object on a virtual display device.

NDC is used to ~~genrate~~ generate device independant points which then can be mapped to an actual device, viewport on demand. [Useful when rendering same objects on multiple screens]

modelling coordinates $\xrightarrow{\text{Modelling Transformation}}$ World coordinates $\xrightarrow{\text{Viewing Transformation}}$ viewing coordinates $\xrightarrow[\text{(Projection \& clipping)}]{\text{Normalization}}$ NDC $\xrightarrow{\text{Viewport Transformation}}$ DC

Viewing pipeline

Q. Define window and viewport.

→ A bounded region in the world coordinate system which defines the visible region of the scene is called <u>clipping window</u> or window.

The scenary is clipped against the window and then mapped to the display device. The area on the display device where an window is mapped to is called a <u>viewport</u>.

Usually both window and viewports are rectangular regions.

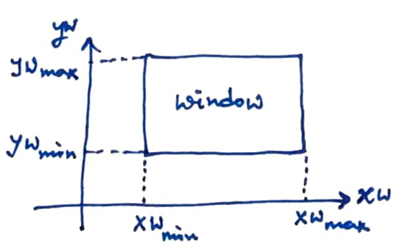
(i) The window & view ports maynot be of same size, thus a transformation is required.

(i) NDC is often used here as intermediary.

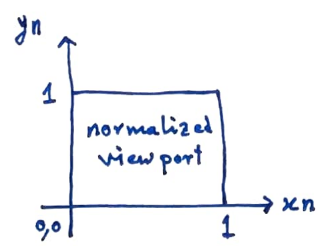
(i) Often <u>2D viewing pipeline</u> is called window-to-viewport transformation.

Q. Describe the window-to-viewport transformation.

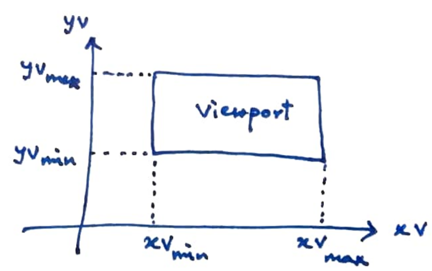
→ Suppose a ~~reetargular~~ window is located on a world coordinate space as shown below. (left fig). We wish to map a point (x_w, y_w) inside our window to a rectangular viewport on a display system. The viewport is located on the device coordinate space as shown below (right fig).



world coordinate system



Normalized device coordinate system



Device coordinate system

We also include normalised device coordinate (NDC) space as an intermediate stage of our ~~cu~~ transformation pipeline.

Suppose we have a point (x_w, y_w) inside our window. If this point gets mapped to (x_n, y_n) in NDC then we have.

$$x_n = (x_w - xw_{min}) / (xw_{max} - xw_{min})$$

$$y_n = \underbrace{(y_w - yw_{min})}_{\text{translation}} / \underbrace{(yw_{max} - yw_{min})}_{\text{scaling to normalize}}$$

Now if the corresponding point ~~of~~ on the device viewport is (xv, yv) then we have

$$xv = xn \cdot (xv_{max} - xv_{min}) + \overbrace{xv_{min}}^{\text{translation}}$$

$$yv = \underbrace{yn \cdot (yv_{max} - yv_{min})}_{\text{scaling to device resolution}} + yv_{min}$$

Thus,

$$xv = xv_{min} + (xw - xw_{min}) \cdot S_x \qquad \text{where,} \quad S_x = \frac{xv_{max} - xv_{min}}{xw_{max} - xw_{min}}$$

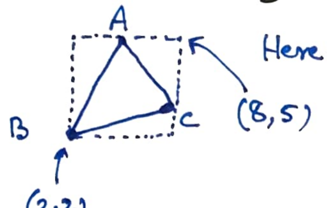
$$yv = yv_{min} + (yw - yw_{min}) \cdot S_y \qquad S_y = \frac{yv_{max} - yv_{min}}{yw_{max} - yw_{min}}$$

(i) Here we have assumed window & viewports are to be rectangle and their sides are parallel to coordinate axes. If the window is rotated or the window is of some other shape then the transformation gets complicated

Q. Consider a triangle ABC whose vertices are given as $A(5,5)$, $B(2,2)$, $C(8,3)$ on the world coordinate system. We define our window as the minimum enclosing axis parallel rectangle containing this triangle.

a) Map this triangle in the NDC system

→



Here for our window we have

$xw_{min} = \min(5,2,8) = 2$

$xw_{max} = \max(5,2,8) = 8$

$yw_{min} = \min(5,2,3) = 2$

$yw_{max} = \max(5,2,3) = 5$

for window to NDC transformation we have,

$$xn = (xw - xw_{min}) / (xw_{max} - xw_{min}) = (xw - 2)/6$$

and

$$yn = (yw - yw_{min}) / (yw_{max} - yw_{min}) = (yw - 2)/3$$

Thus point A is transformed to $((5-2)/6, (5-2)/3)$ or $(0.5, 1)$

point B - - - - - - - - $((2-2)/6, (2-2)/3)$ or $(0, 0)$

point C - - - - - - - - $((8-2)/6, (3-2)/3)$ or $(1, 1/3)$

b) Map the triangle to a screen having resolution 640×480.

→ Here the viewport is entire screen. Thus.

$xv_{min} = 0$ and $xv_{max} = 640$

$yv_{min} = 0$ $\quad$ $yv_{max} = 480$

for transforming NDC to viewport we have

$$xv = xv_{min} + xn(xv_{max} - xv_{min}) = xn \cdot 640$$

and $$yv = yv_{min} + yn(yv_{max} - yv_{min}) = yn \cdot 480$$

Thus point A is mapped to $(0.5 \times 640,\ 1 \times 480)$ or $(320, 480)$

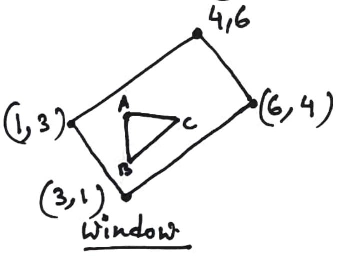
- - - B - - - - - - - - $(0 \times 640,\ 0 \times 480)$ or $(0, 0)$

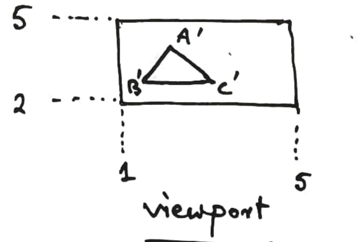
- - - C - - - - - - - - $(1 \times 640,\ \frac{1}{3} \times 480)$ or $(640, 120)$

(i) device points must be integers, so if fraction is obtained after transformation then round-off is applied.

---

*Q. Consider the following window containing a triangle. Transform this triangle into the following viewport.





A(3,4), B(3,2), C(5,4)

→ Hint: 1. translate the window to origin by (−3,−1)

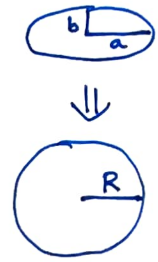
2. rotate by −θ, where tanθ is slope of the line joining (3,1) & (6,4) [or (1,3) & (4,6)]

3. apply scaling to match the rectangle sizes.

4. tanslate to (1,2) to get the final viewport.

*Q. How can we transform an elliptical window to a circular viewport?

→ Suppose elliptical window has semi-major axis = a
& semi-minor axis = b



and the viewport has a radius of R.

Here we need to apply scaling where $S_x = R/a$

$S_y = R/b$

If window or viewport has some translation, that can be addjusted similar to the rectangular case.

Suppose (xw_c, yw_c) and (xv_c, yv_c) denotes centers of the window & the viewport respectively.

So if a point (xw, yw) on the window is mapped to (xv, yv) on the viewport we have the following relation.

$$xv = xv_c + (xw - xw_c) \cdot S_x$$

and $$yv = yv_c + (yw - yw_c) \cdot S_y.$$

Q. What is clipping?

→ When the viewing window is small with respect to the entire scene/picture, then it is desirable that the output device only renders the visible portion of the scene and do not waste its resource and time on the scene outside of the window.

The <u>clipping</u> process removes the object and object segments that are outside the viewing window/clipping window and draws only the objects and object segments that are inside the window.

(solid lines are drawn
dotted lines are not drawn).

(i) clipping is also used for selecting a portion of an scene/image for copying, moving, erasing, transforming or appling other image processing techniques.

Q. How clipping procedure works for a point?

→ If a point belongs to the visible region i.e. falls inside the clipping window then the point is kept and discarded otherwise.

So when the clipping window is an axis parallel rectangle with its bottom left corner at (xw_{min}, yw_{min}) and top right corner at (xw_{max}, yw_{max}) the clipping decision is as follows for a point $P(x,y)$.

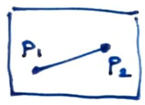
if $xw_{min} \leq x \leq xw_{max}$ and $yw_{min} \leq y \leq yw_{max}$ then keep P

otherwise discard P.

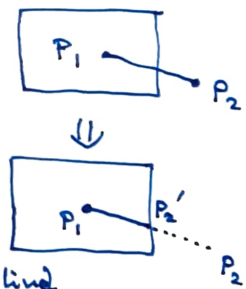
Q. Describe the idea of line clipping (against an axis parallel rectangular window)

→ A line segment is defined by its two end points. Now one of the ~~three~~ four cases might occur:

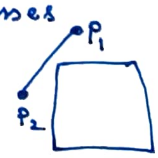
case 1 : if both the end point are inside the visible region (window) then entire line segment is visible, thus no clipping needed.



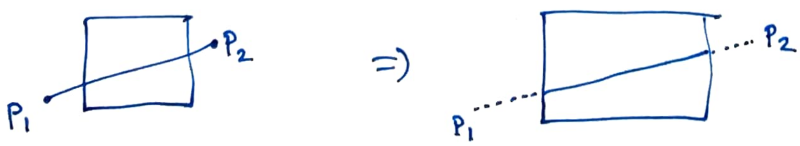
case 2 : if one of the end points is inside the rectangle and the other one is outside, then a clipping is needed. In this example we have P_1 inside & P_2 outside. We move the point P_2 on the boundary where the line intersects. lets call this point P_2' and we draw $P_1 - P_2'$ line segment and discard the $P_2' - P_2$ portion of the original line segment (shown in dotted line)



case 3 : if both endpoints are outside and the line segment does not passes through the visible region, we discard the line entirely

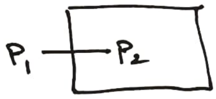


case 4 : if both end points are outside the window but some portion of the line passes through the visible region, we need to clip the line on both ends. This is usually done one by one just like case 2.

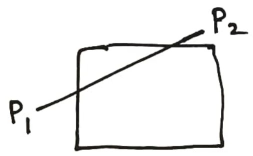


(i) Before we present a formal algorithm, let's look at the line clipping more closely.

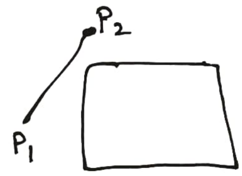
Suppose, a line segment $P_1 - P_2$ has its one end point (say) P_1 outside the window. Now depending on the position of P_2, we have three possible cases.



P_2 inside, clip one side



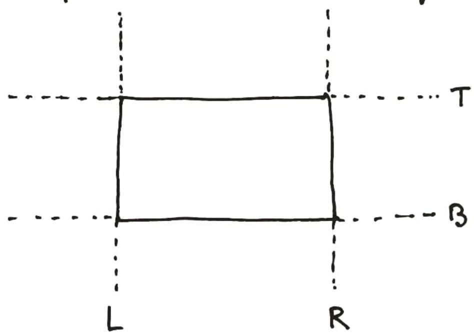
P_2 outside, clip both side
but line is partly visible



P_2 outside, discard the entire line segment.
line is completely invisible

Now how to detect whether a point is outside or inside? This can be done by the visibility test for a point.

Here we are usually also interested to know a point is outside against which boundary line. Here we have 4 axis parallel boundary lines obtained by extending the boundaries of the rectangular window, we call them Top (T), Bottom (B), Right (R) and Left (L). The line equations are given below.



T : $y = yw_{max}$

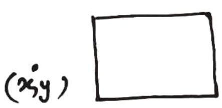
B : $y = yw_{min}$

R : $x = xw_{max}$

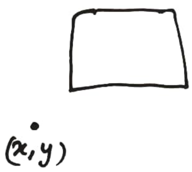
L : $x = xw_{min}$

Now a point might be outside the window against one or two boundary line.

example



$x < L$



$x < L$ and $y < B$

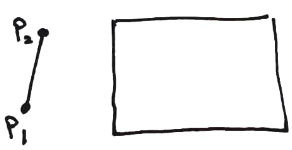
. And if a point is inside it must satisfy all four of these inequalities

$L \le x$ $\quad$ $B \le y$

$x \le R$ $\quad$ $y \le T$

So, if two end points are inside (w.r.t. all four boundary lines) then we have a trivial accept case, where the entire line is visible.

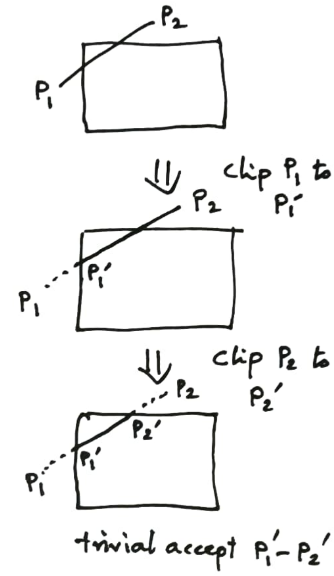
And if both end points are outside w.r.t same boundary line then ~~as~~ we have a trivial reject case, where the entire line is invisible



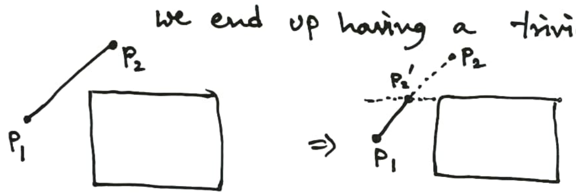
both P_1 & P_2 are left of line L

A <u>non trivial case</u> occurs when, both of the end points are outside but not against some same boundary line.

<u>non trivial case a</u>: Suppose both of the end points are outside and the line is partially visible. If we clip one side of the line segment and then the other end we end up having a trivial accept case.

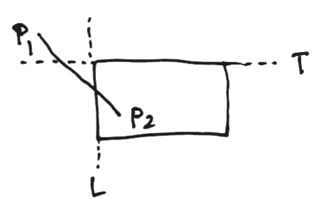


<u>non trivial case b</u>: Suppose both of the end points are outside and the line is completely invisible. Here again if we try to clip one ~~side~~ end of the line segment by moving the end point onto a boundary line and then apply the same process on the other end we end up having a trivial reject case.

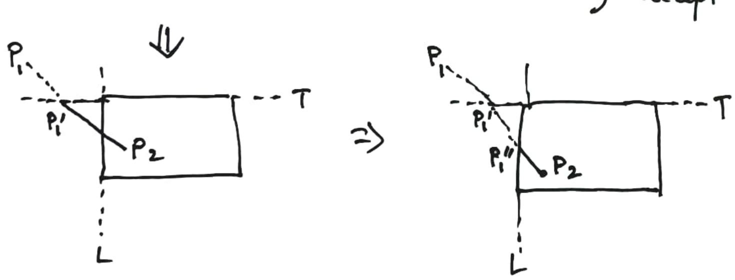


clip P_2 to P_2' then trivial reject P_1-P_2' as they are both left of L

Lastly, for systematic clipping operation we test each of two endpoints against the four boundary lines one by one. In doing so, we may need to clip twice on one end of the line segment.



Here P_1 is above of T thus we clip P_1 to P_1' first. Then we observe that P_1' is left of L ~~thus~~ so we clip it to P_1'' and trivially accept P_1''-P_2

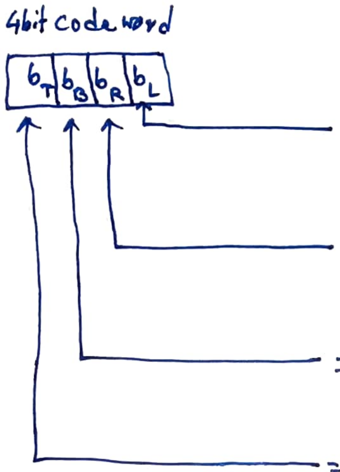


Finally we are ready to present a formal algorithm, namely Cohen-Sutherland line clipping algorithm which does this operation efficiently.

Q. Describe the Cohen-Sutherland line clipping algorithm.

→ Here the clipping window is of rectangular shape whose top, bottom, right and left boundaries are denoted by T, B, R & L respectively and they parallel to coordinate axes.

We assign a 4 bit code word to a point depending on its position w.r.t. the clipping window.



$b_L = \begin{cases} 0 & \text{when point is inside w.r.t line L (i.e. right of line L)} \\ 1 & \text{otherwise (i.e. left of line L)} \end{cases}$

$b_R = \begin{cases} 0 & \text{when point is inside w.r.t line R (i.e. left of line R)} \\ 1 & \text{otherwise (i.e. right of line R)} \end{cases}$

$b_B = \begin{cases} 0 & \text{when point is inside w.r.t line B (i.e. above of line B)} \\ 1 & \text{otherwise (i.e. below of line B)} \end{cases}$

$b_T = \begin{cases} 0 & \text{when point is inside w.r.t line T (i.e. below of line T)} \\ 1 & \text{otherwise (i.e. above of line T)} \end{cases}$

formally this can be written as :

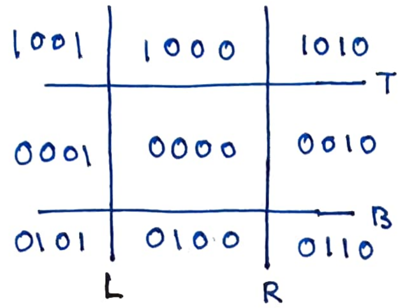
```
get_code( x, y ) {
    code = 0 // 4bit value
    if ( x < L ) {
        code = code OR 1
                       ↑ LEFT
    }
    if ( x > R ) {
        code = code OR 2
                       ↑ RIGHT
    }
    if ( y < B ) {
        code = code OR 4
                       ↑ BOTTOM
    }
    if ( y > T ) {
        code = code OR 8
                       ↑ TOP
    }
    return code
}
```

(i) OR will set a bit to 1

(i) Notice here we have not written else-if since a point might be outside w.r.t ~~to~~ more than one boundary lines.

(i) One can use enum construct to define constants LEFT = 1 (0001)
RIGHT = 2 (0010)
BOTTOM = 4 (0100)
TOP = 8 (1000)
for better readability.

This coding scheme divides the coordinate system into 9 regions as shown below



Let $code_1$ and $code_2$ denotes codewords generated for the two end points $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ of a line segment.

Now we make a few observations :

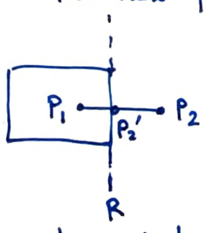
Observation 1 : the visible region points gets code 0000 (or simply 0) therefore if both $code_1$ and $code_2$ are 0 then we have a trivial accept case

Observation 2 : corner region has a code equivalent to the ORed value of its two adjacent region's codes. e.g. 1001 same as (1000 OR 0001). This is natural since a point in a corner region is outside w.r.t two boundary lines. Therefore, if a point belongs to a corner region then it ~~has~~ may need to be processed twice for the two boundary lines.

Observation 3 : if two points are outside w.r.t same boundary line then both of their codewords will have the same bit set. Therefore, ($code_1$ AND $code_2$) is a nonzero quantity if and only if P_1 & P_2 are outside w.r.t same boundary line, thus we have a trivial reject case.

Observation 4 : if neither trivial accept or trivial reject happens, then at least one of P_1 & P_2 is outside the visible region and thus needs clipping. We need to repeat this process until a trivial accept or trivial reject occurs

Observation 5 : When we clip a line, one of its end point moves to a boundary. The new point is a point on the same line but on that boundary.



P_2' is the intersection point of the line segment P_1-P_2 and line R

So coordinates of new point can easily be obtained from the line equation and position of the boundary line.

e.g. here P_2' has same x-coordinate as line R and y-coordinate of P_2' can be obtained from the line eqn.

Finally we present the Cohen-Sutherland algorithm. We assume that positions of the 4 boundary lines is globally available.

```
Cohen_Sutherland_line_clipping_algorithm (x1, y1, x2, y2) {

    code1 = get_code(x1, y1)

    code2 = get_code(x2, y2)

    while (true) {

        if ( code1 OR code2 == 0 ) { // trivial accept
            keep/draw the line from (x1,y1) to (x2,y2) and break
        } else if ( code1 AND code2 ≠ 0 ) { // trivial reject
            discard the line segment and break.
        } else { // non trivial case, at least one is outside and needs clipping

            code = max (code1, code2) // get code of the outside point
                                      if both are outside any one would do the job
                         LEFT
            if ( code AND 1 ≠ 0) { // region is left of L

                x = xw_min
                y = (x - x1) . (y2 - y1) / (x2 - x1) + y1     } intersection point of the line segment and L
                          RIGHT
            } else if ( code AND 2 ≠ 0) { // region is right of R

                x = xw_max
                y = (x - x1) . (y2 - y1) / (x2 - x1) + y1     } intersection point of the line segment and R
                         BOTTOM
            } else if ( code AND 4 ≠ 0) { // region is below B

                y = yw_min
                x = (y - y1) . (x2 - x1) / (y2 - y1) + x1     } intersection point of the line segment & B

            } else { // region is above T

                y = yw_max
                x = (y - y1) . (x2 - x1) / (y2 - y1) + x1     } intersection point of the line segment & T
            }
            if (code = code1) { // need to update P1
                x1 = x, y1 = y , code1 = get_code (x,y)       } updation for next iteration
            } else { // need to update P2
                x2 = x, y2 = y , code2 = get_code (x,y)
}       }       }
```

ⓘ this handles multiple clippings if needed.

ⓘ $\frac{x-x_1}{x_2-x_1} = \frac{y-y_1}{y_2-y_1}$ gives these eqn.

ⓘ here we have used else-if deliberately

loop until a trivial case occurs

Example 7.3 A Clipping window ABCD is located as follows:

$A(100, 10)$, $B(160, 10)$, $C(160, 40)$, $D(100, 40)$. Using Sutherland-Cohen clipping algorithm find the visible portion of the line segments EF, GH and P_1P_2. $E(50,0)$, $F(70, 80)$, $G(120, 20)$, $H(140, 80)$, $P_1(120, 5)$, $P_2(180, 30)$

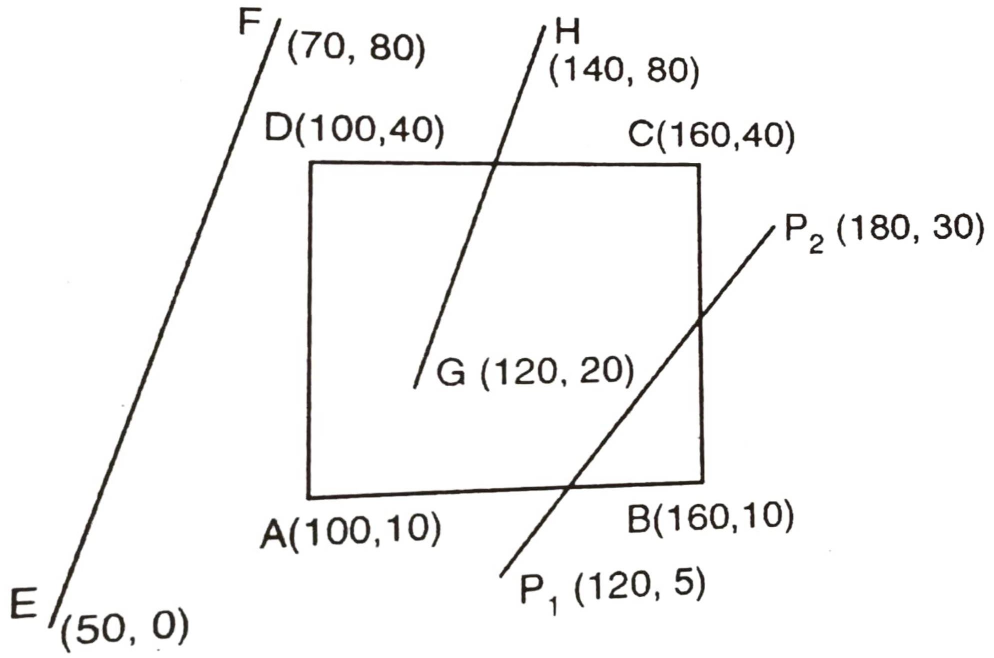


Fig. 7.13