CSc 133 Lecture Notes

15 - Viewing Transformations

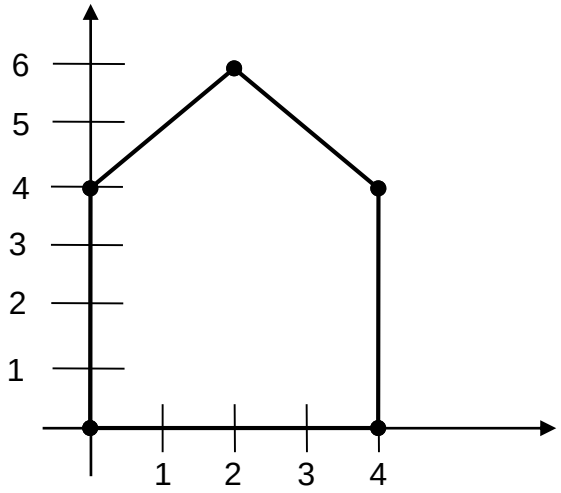
Computer Science Department
California State University, Sacramento

Overview

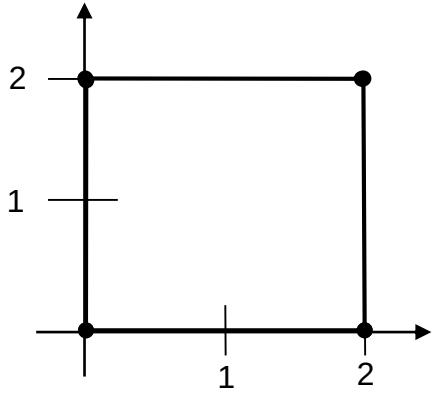
- **The World Coordinate System**
- **Mapping From World to Display Coordinates**
  - **World Window, Normalized Device (ND), World-to-ND Transform, ND-to-Display Transform, the Viewing Transformation Matrix (VTM)**
- **2D Viewing Operations (Zoom and Pan)**
- **Clipping and the Cohen-Sutherland Algorithm**

Local Coordinate Systems

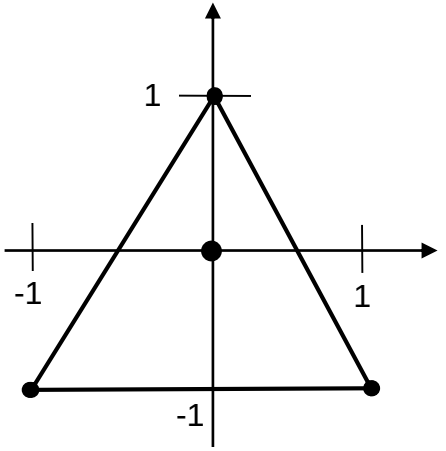
- Each object is defined in its “own space”



Pentagon

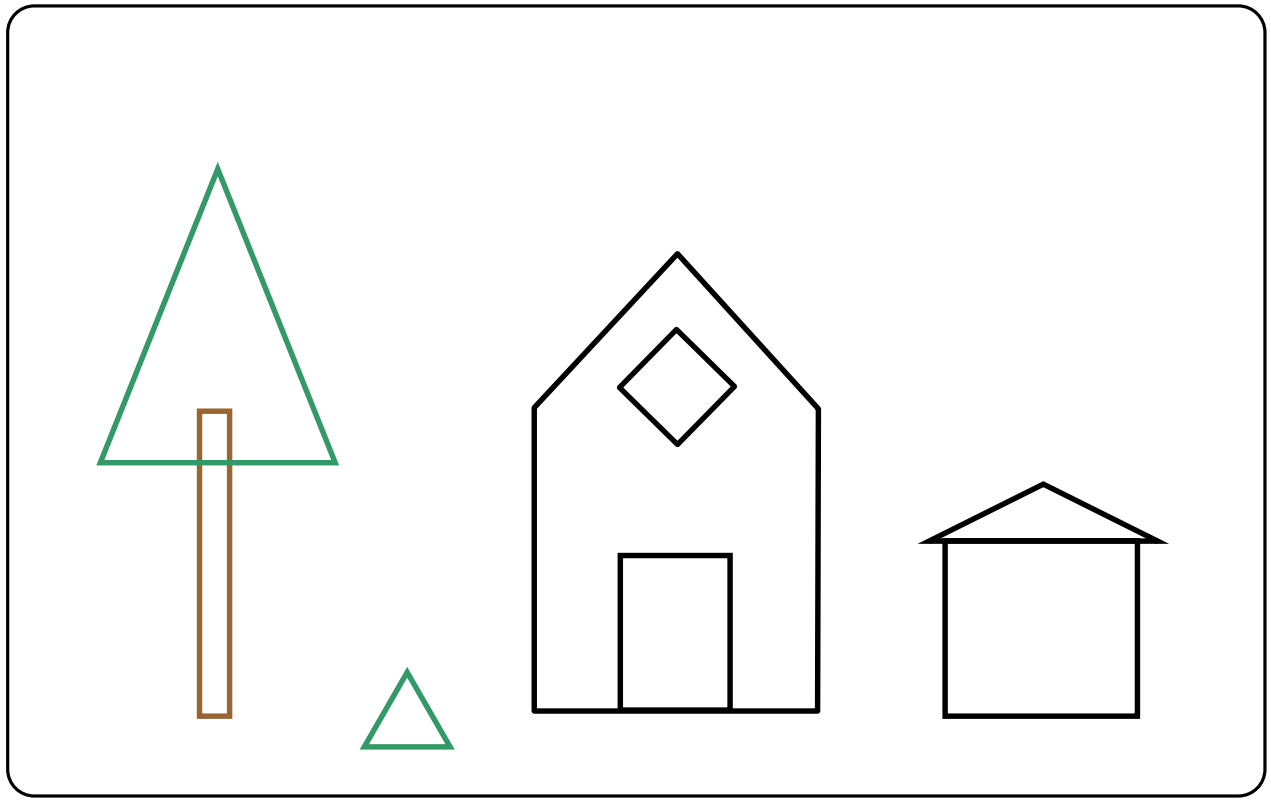


Box



Triangle

Creating A "World"



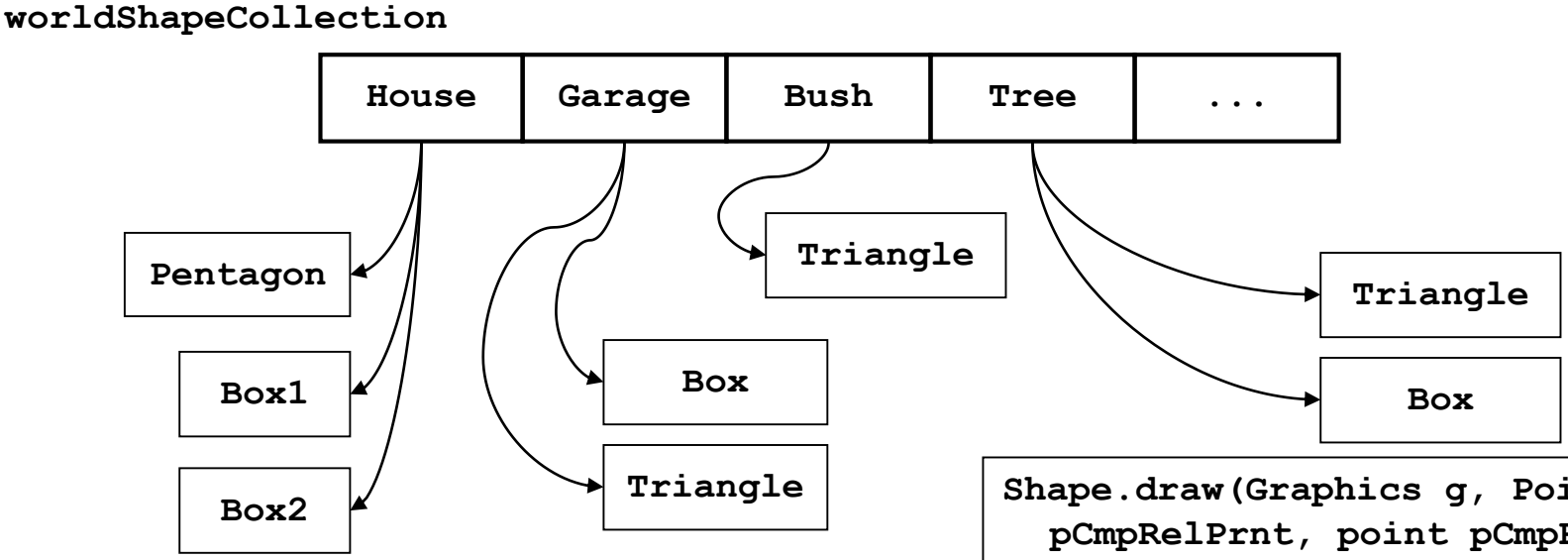
Tree
(Triangle + Box)

Bush
(Triangle)

House
(Pentagon + Two Boxes)

Garage
(Box + Triangle)

The World Object Collection




```
CustomContainer.paint (Graphics g){
  apply "display mapping" and "local
            origin" transforms to g;
  for (Shape s : worldShapeCollection){
    s.draw(g, pCmpRelPrnt, pCmpRelScrn);
  }
  restore xform in g with resetAffine();
}
```

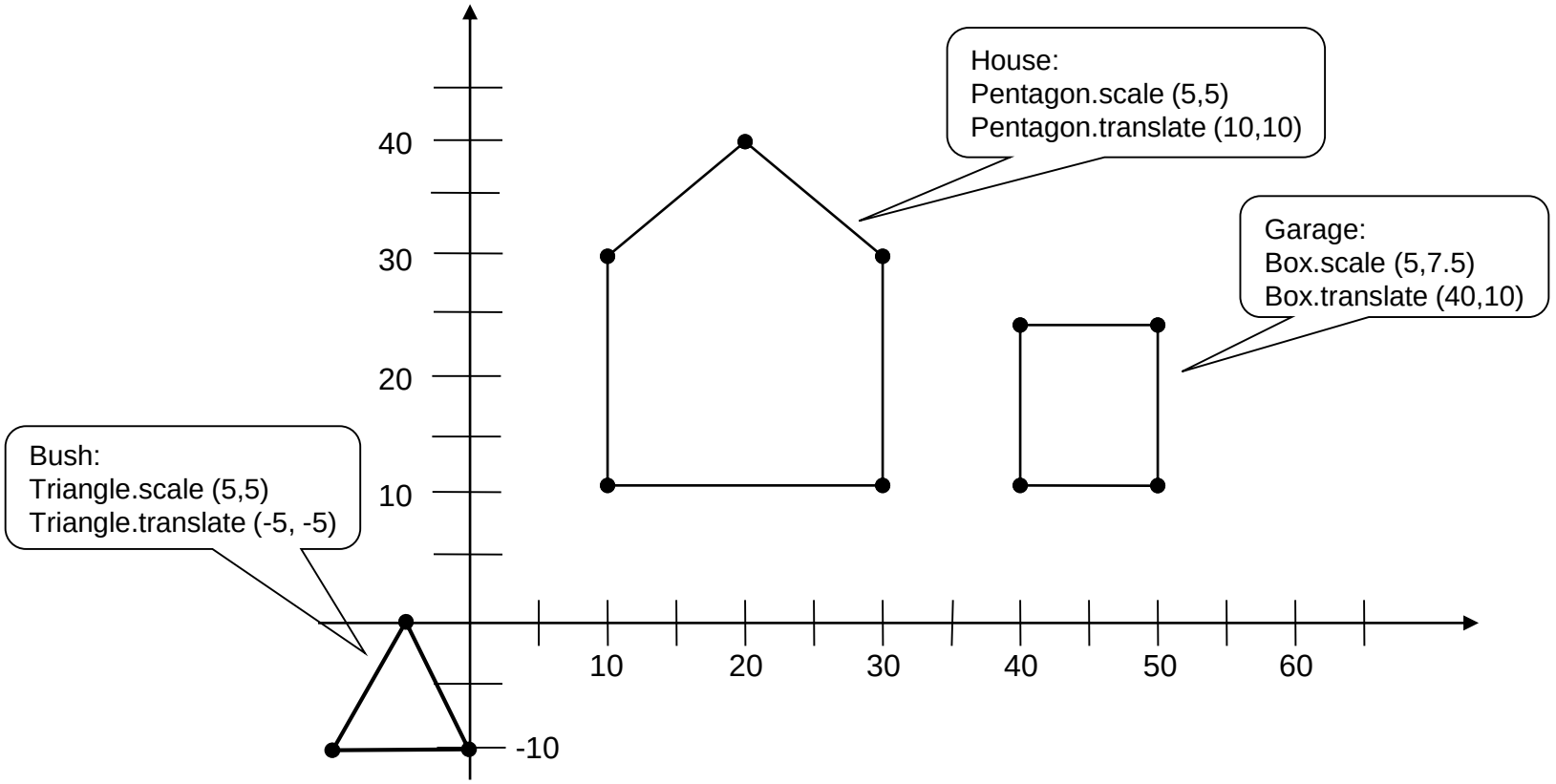
```
Shape.draw(Graphics g, Point
  pCmpRelPrnt, point pCmpRelScrn)
{ save xform in g as gOrigXform;
  apply LTs and "local origin"
                  transforms to g;
  for (each sub shape) {
   sub.draw(g, pCmpRelPrnt,
                  pCmpRelScrn);

  }
 restore xform in g with
       setTranform(gOrigXfrom);
}
```

The World Coordinate System

- "World" ("virtual" or "user") units
  - Independent of display
  - Can represent inches, feet, meters…
- Infinite in all directions
- Object instances are "placed" in the World via *local transformations*

World Coordinate System (cont.)

Example:



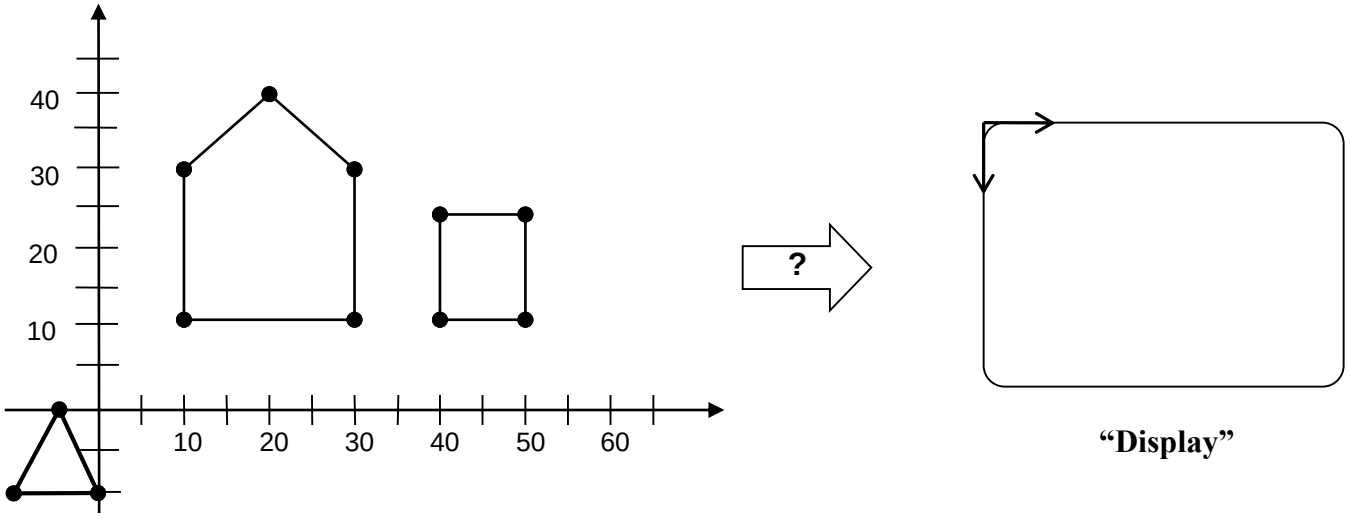
Local Transformations

- With the introduction of world coordinate system, Local Transformations (LTs) no longer place the objects on display, but instead place them in world.
- Hence, in the case of a simple object (e.g., an object which is drawn as a simple triangle) or a top-level object of an hierarchical object (e.g., FireOval object), LTs transform points from local space to world space. Remember that in the previous chapter, LTs were transforming points from local space to display space.
- In case of sub-objects of the hierarchical object (e.g, Flame sub-object of FireOval), just like in the previous chapter, LTs transform points from local space of sub-object to local space of the hierarchical object (apply local scale/rotate/translate to the sub-object to size, orient, and position it relative to the center of the hierarchical object).

Drawing The World On The Display

Needed:

- A way to determine what portion of the (infinite) World gets drawn on the (finite) display
- A "mapping" or *transformation* from *World* to *Display* coordinates



Drawing The World (cont.)

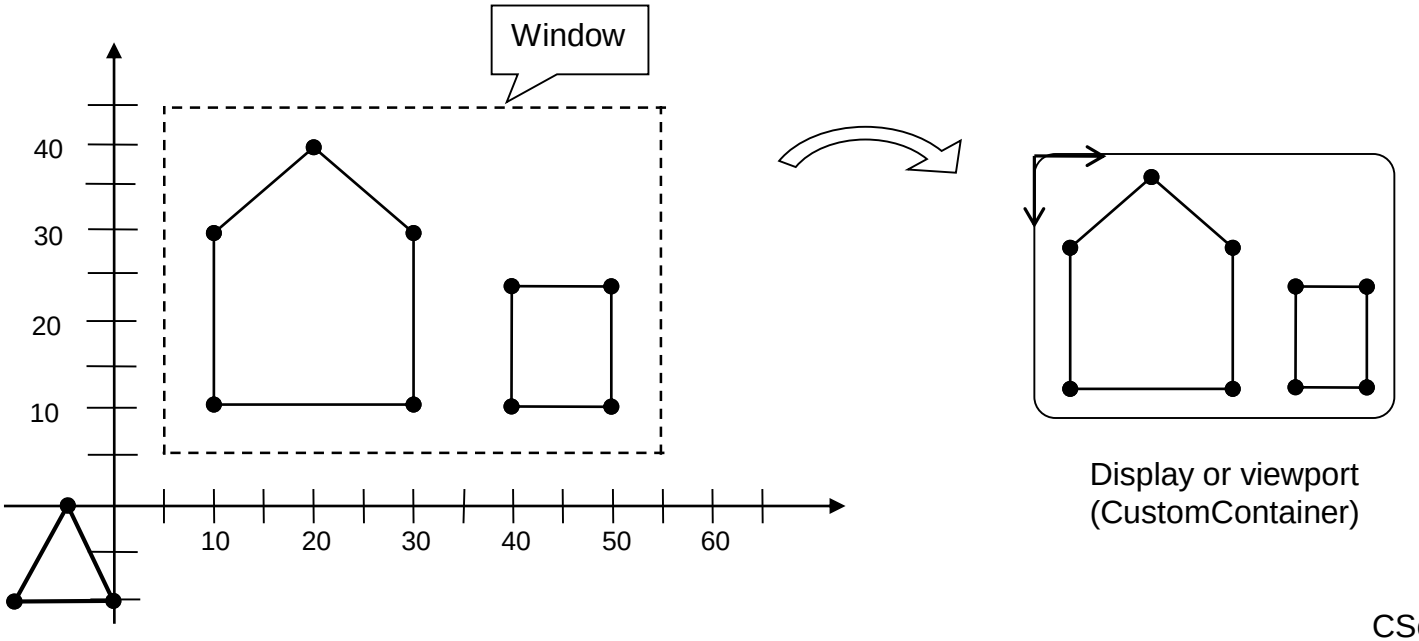
Solution:

- The “Virtual (World) Window”
- A *two-step* mapping through a “*Normalized Device*”

The World "Window"

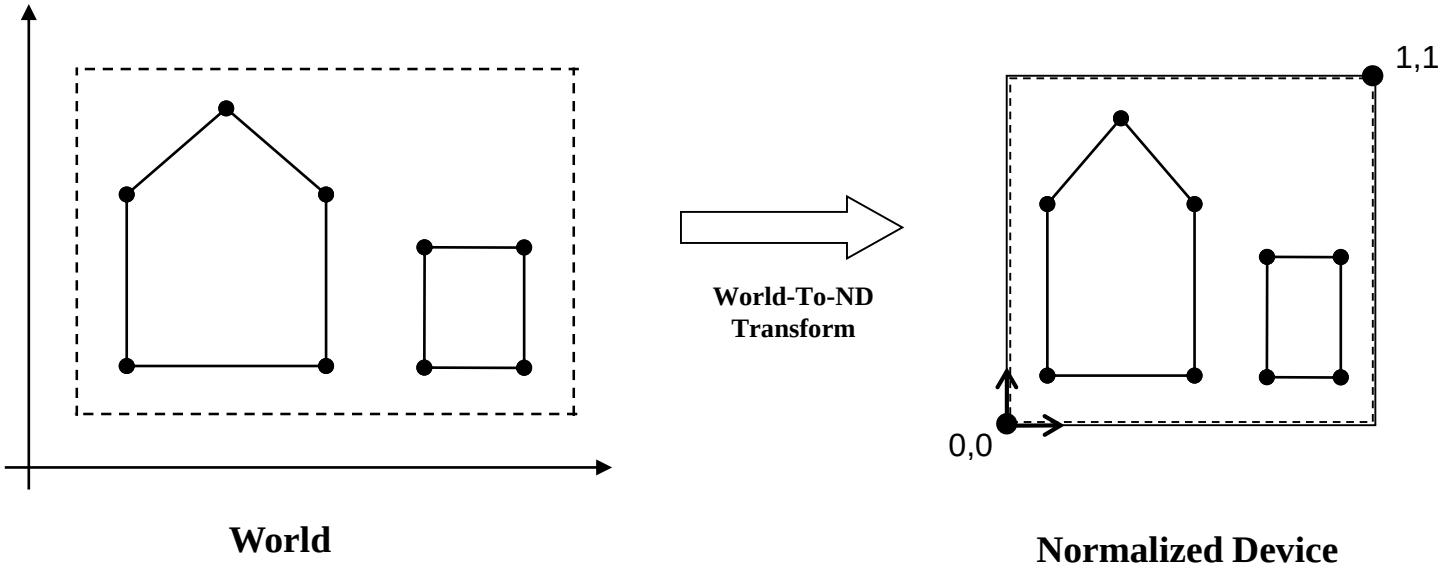
Defines the part of the world that appears on display

- Corners of the window match the corners of the display ("viewport")
- Objects inside window are positioned proportionally in the viewport
- Objects outside window are "clipped" (discarded)



The "Normalized Device"

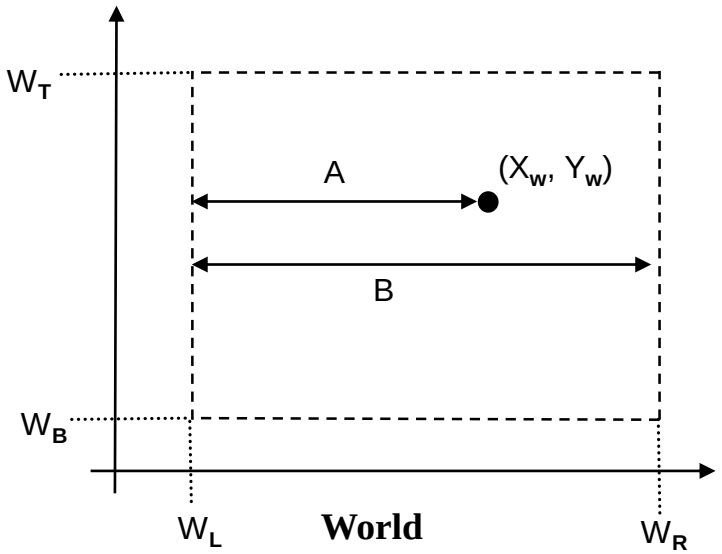
- Properties of the Normalized Device (ND):
  - Square
  - Fractional Coordinates (0.0 .. 1.0)
  - Origin at Lower Left
  - Corners correspond to world window



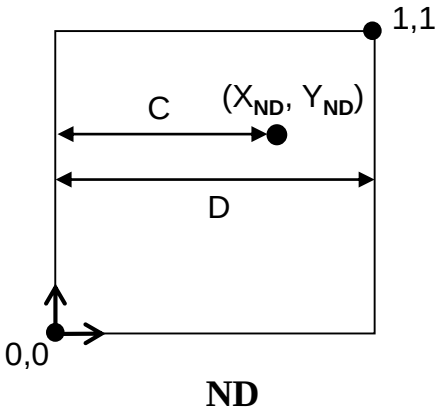
World-To-ND Transform

Consider a single point's X coordinate

- Need to achieve proportional positioning on the ND



$$\frac{A}{B} = \frac{C}{D}$$



$$A = X_w - W_L \; ; \qquad B = W_R - W_L \; ; \qquad D = 1 ;$$

$$\therefore \; C = X_{ND} = \frac{(X_w - W_L)}{(W_R - W_L)} = (X_w - W_L) * \frac{1}{(W_R - W_L)}$$

World-To-ND Transform (cont.)

Consider the form of X_{ND} :

$$X_{ND} = \underbrace{(X_w - W_L)}_{\text{A translation (by -WindowLeft)}} * \underbrace{\frac{1}{(W_R - W_L)}}_{\text{A scale (by 1/windowWidth)}}$$

Similar rules can be used to derive Y_{ND} :

$$Y_{ND} = \underbrace{(Y_w - W_B)}_{\text{A translation}} * \underbrace{\frac{1}{(W_T - W_B)}}_{\text{A scale}}$$

World-To-ND Transform (cont.)

$X_{ND} = (X_W \bullet \text{Translate}(-W_L)) \bullet \text{Scale}(1 / \text{WindowWidth})$

$Y_{ND} = (Y_W \bullet \text{Translate}(-W_B)) \bullet \text{Scale}(1 / \text{WindowHeight})$

or

$P_{ND} = (P_W \bullet \text{Translate}(-W_L, -W_B)) \bullet \text{Scale}(1/\text{WindowWidth}, 1/\text{WindowHeight})$

- In Matrix Form:

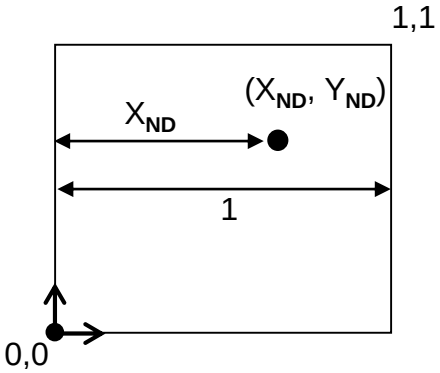
$$\begin{pmatrix} X_{ND} \\ Y_{ND} \\ 1 \end{pmatrix} = \left(\begin{pmatrix} 1/W_w & 0 & 0 \\ 0 & 1/W_h & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -W_L \\ 0 & 1 & -W_B \\ 0 & 0 & 1 \end{pmatrix} \right) \begin{pmatrix} X_W \\ Y_W \\ 1 \end{pmatrix}$$

X

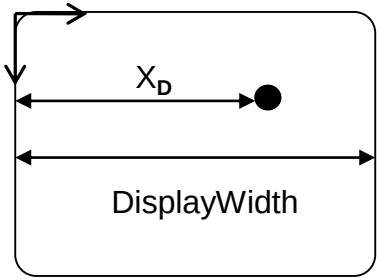
$$\begin{pmatrix} X_{ND} \\ Y_{ND} \\ 1 \end{pmatrix} = \begin{pmatrix} \text{World-to-Normalized-Device (W2ND) Transform} \end{pmatrix} \begin{pmatrix} X_W \\ Y_W \\ 1 \end{pmatrix}$$

ND-To-Display Transform

- A similar approach can be applied



ND



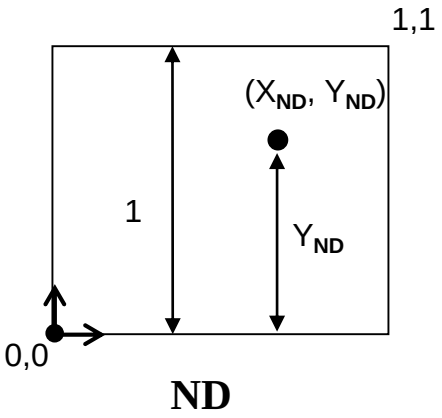
Display

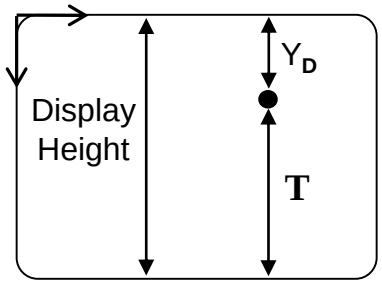
$$\frac{X_{ND}}{1} = \frac{X_D}{DisplayWidth} \; ; \quad \therefore \quad X_D = X_{ND} \times DisplayWidth$$

$$X_D = X_{ND} \bullet Scale(DisplayWidth)$$

ND-To-Display Transform (cont.)

- Similarly for height:





T = DisplayHeight – Y_D

$$\frac{Y_{ND}}{1} = \frac{T}{DisplayHeight} = \frac{(DisplayHeight - Y_D)}{DisplayHeight} \;;$$

$$Y_D = (Y_{ND} \times (-DisplayHeight)) + DisplayHeight$$

$$Y_D = (Y_{ND} \bullet Scale(-DisplayHeight)) \bullet Translate(DisplayHeight)$$

ND-To-Display Transform (cont.)

$X_D = (X_{ND} \bullet \text{Scale}(\text{DisplayWidth})) \bullet \text{Translate}(0)$

$Y_D = (Y_{ND} \bullet \text{Scale}(-\text{DisplayHeight})) \bullet \text{Translate}(\text{DisplayHeight})$

or

$P_D = (P_{ND} \bullet \text{Scale}(\text{DisplayWidth}, -\text{DisplayHeight})) \bullet \text{Translate}(0, \text{DisplayHeight})$

- In Matrix Form:

$$\begin{pmatrix} X_D \\ Y_D \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & D_{height} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} D_{width} & 0 & 0 \\ 0 & -D_{height} & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{\text{ND-to-Display Transform}} \begin{pmatrix} X_{ND} \\ Y_{ND} \\ 1 \end{pmatrix}$$

Combining Transforms

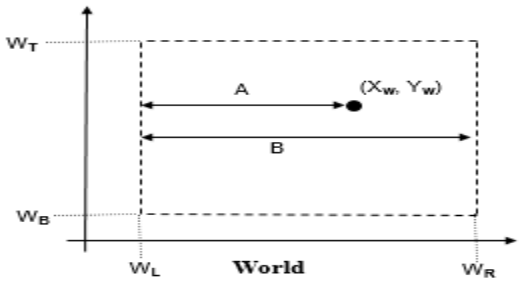
$$\begin{bmatrix} X_D \\ Y_D \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} \text{ND} \\ \text{to} \\ \text{Display} \end{bmatrix} \times \left(\begin{bmatrix} \text{World} \\ \text{to} \\ \text{ND} \end{bmatrix} \times \begin{bmatrix} X_W \\ Y_W \\ 1 \end{bmatrix} \right)}$$

$$\begin{bmatrix} X_D \\ Y_D \\ 1 \end{bmatrix} = \begin{bmatrix} \text{“VTM”} \end{bmatrix} \begin{bmatrix} X_W \\ Y_W \\ 1 \end{bmatrix}$$

Using The VTM

- Suppose we have
  - A container with access to a collection of *Shapes*
  - Each shape has a **`draw()`** method which:
    - applies the shape's *local transforms* to gXform
    - calls **`draw()`** on its *sub-shapes*, which applies the sub-shape's local transforms to gXform and draws the sub-shape in "local"coords
- Effect: all draws output *world coordinates*
- We need to *apply the VTM to all output coordinates*

Using The VTM (cont.)



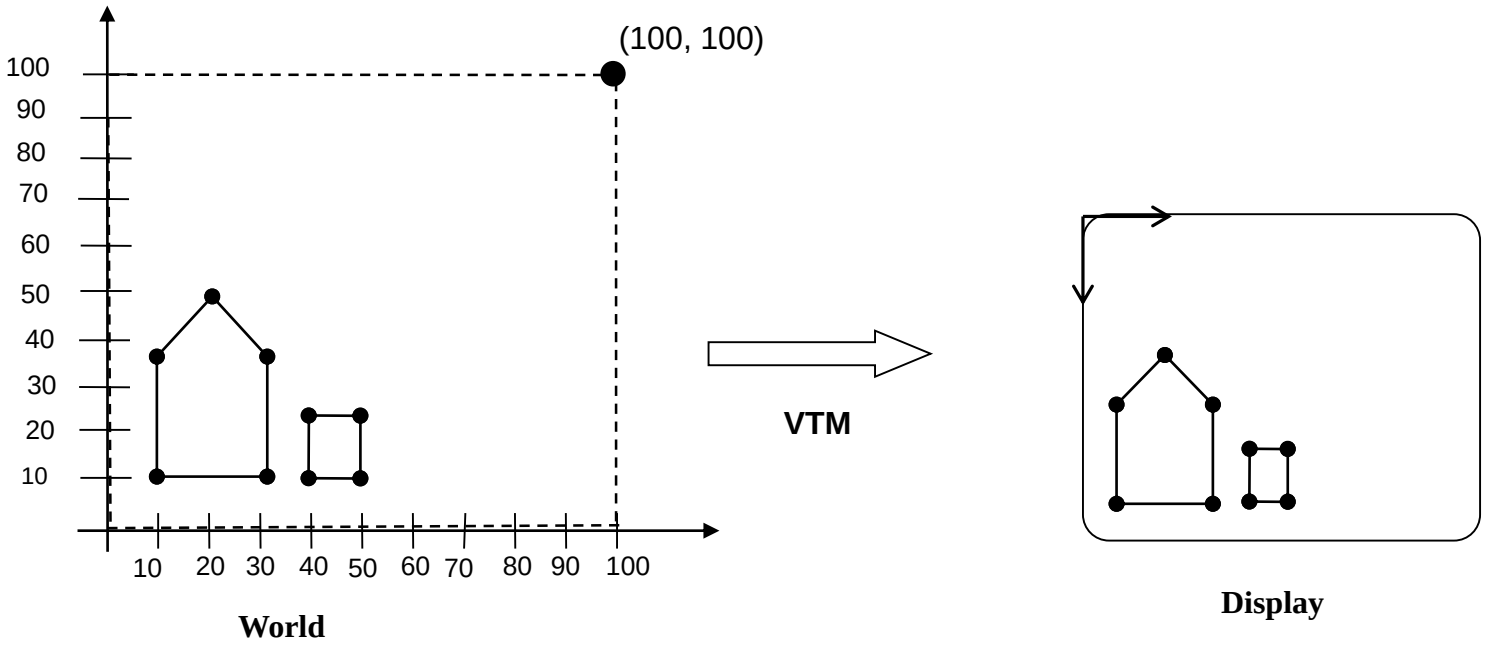

```
public class CustomContainer extends Container {
  Transform worldToND, ndToDisplay, theVTM ;
  private float winLeft, winBottom, winRight, winTop;
  public CustomContainer(){
    //initialize world window
    winLeft = 0;
    winBottom = 0;
    winRight = 931/2; //hardcoded value = this.getWidth()/2 (for the iPad skin)
    winTop = 639/2; //hardcoded value = this.getHeight()/2 (for the iPad skin)
    float winWidth = winRight - winLeft;
    float winHeight = winTop - winBottom;
    //create shapes
    myTriangle = new Triangle((int)(winHeight/5),(int)(winHeight/5));
    myTriangle.translate(winWidth/2, winHeight/2);
    myTriangle.rotate(45);
    myTriangle.scale(1, 2);
    //...[create other simple or hierarchical shapes and add them to collection]
  }
  public void paint (Graphics g) {
    super.paint(g);
    //...[calculate winWidth and winHeight]
    // construct the Viewing Transformation Matrix
    worldToND = buildWorldToNDXform(winWidth, winHeight, winLeft, winBottom);
    ndToDisplay = buildNDToDisplayXform(this.getWidth(), this.getHeight());
    theVTM =  ndToDisplay.copy();
    theVTM.concatenate(worldToND); // worldToND will be applied first to points!
    ... continued ...
```

Using The VTM (cont.)

```
   ... continued ...
   // concatenate the VTM onto the g's current transformation (do not forget to apply "local
   //origin" transformation)
   Transform gXform = Transform.makeIdentity();
   g.getTransform(gXfrom);
   gXform.translate(getAbsoluteX(),getAbsoluteY()); //local origin xform (part 2)
   gXform.concatenate(theVTM); //VTM xform
   gXform.translate(-getAbsoluteX(),-getAbsoluteY()); //local origin xform (part 1)
   g.setTransform(gXform);
   // tell each shape to draw itself using the g (which contains the VTM)
   Point pCmpRelPrnt = new Point(this.getX(), this.getY());
   Point pCmpRelScrn = new Point(getAbsoluteX(),getAbsoluteY());
   for (Shape s : shapeCollection)
     s.draw(g, pCmpRelPrnt, pCmpRelScrn);
   g.resetAffine() ;
 }
 private Transform buildWorldToNDXform(float winWidth, float winHeight, float
 winLeft, float winBottom){
   Transform  tmpXfrom = Transform.makeIdentity();
   tmpXfrom.scale( (1/winWidth) , (1/winHeight) );
   tmpXfrom.translate(-winLeft,-winBottom);
   return tmpXfrom;
 }
 private Transform buildNDToDisplayXform (float displayWidth, float displayHeight){
   Transform tmpXfrom = Transform.makeIdentity();
   tmpXfrom.translate(0, displayHeight);
   tmpXfrom.scale(displayWidth, -displayHeight);
   return tmpXfrom;
 }
//...[other methods of CustomContainer]
}//end of CustomContainer
```

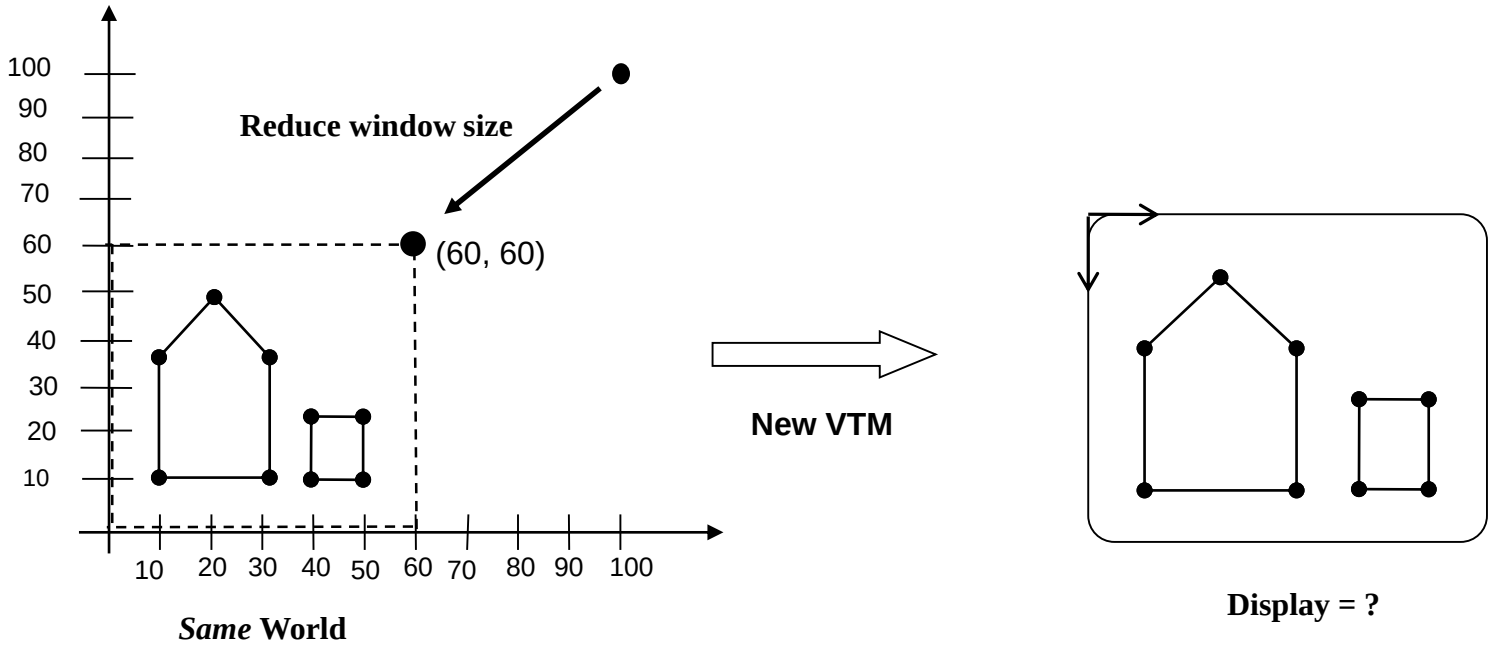
Changing the Window Size

Suppose we start with this:



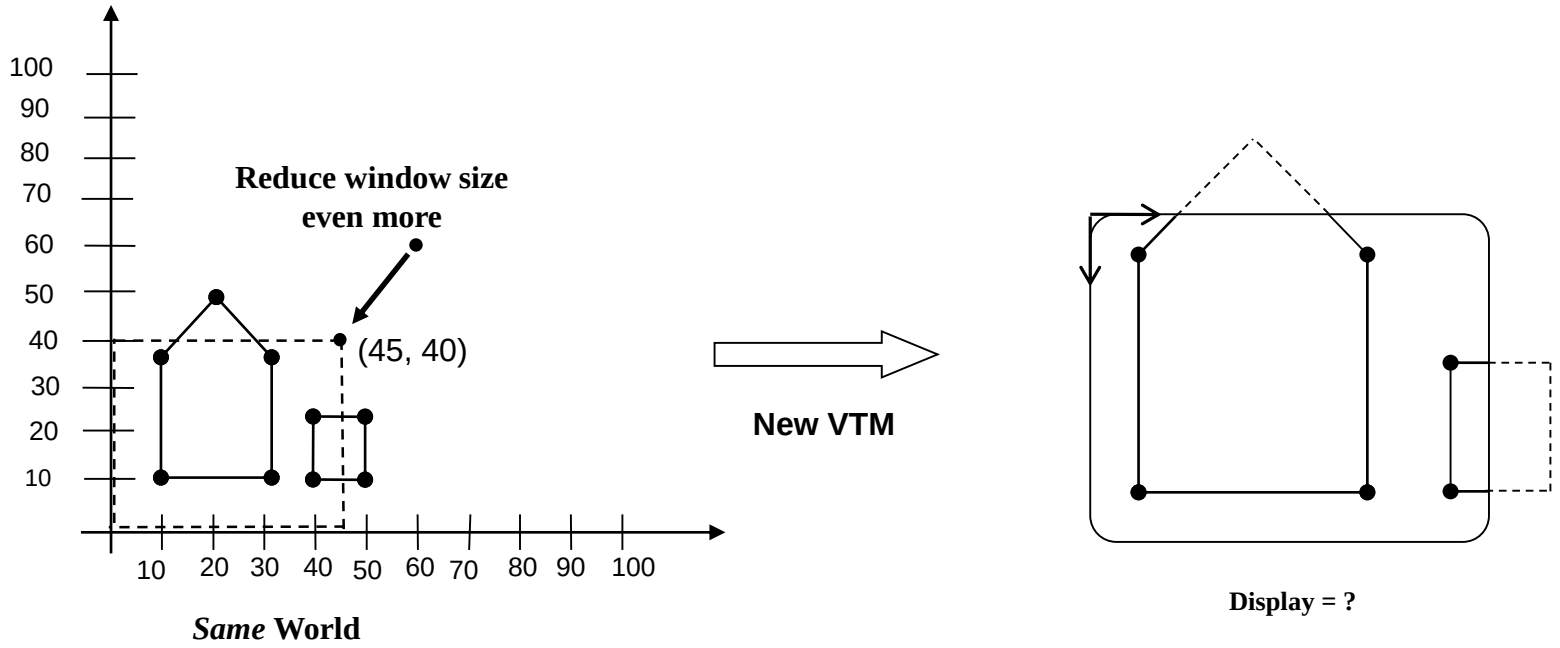
Changing the Window Size (cont.)

Now we change window size,
recompute the VTM, and repaint



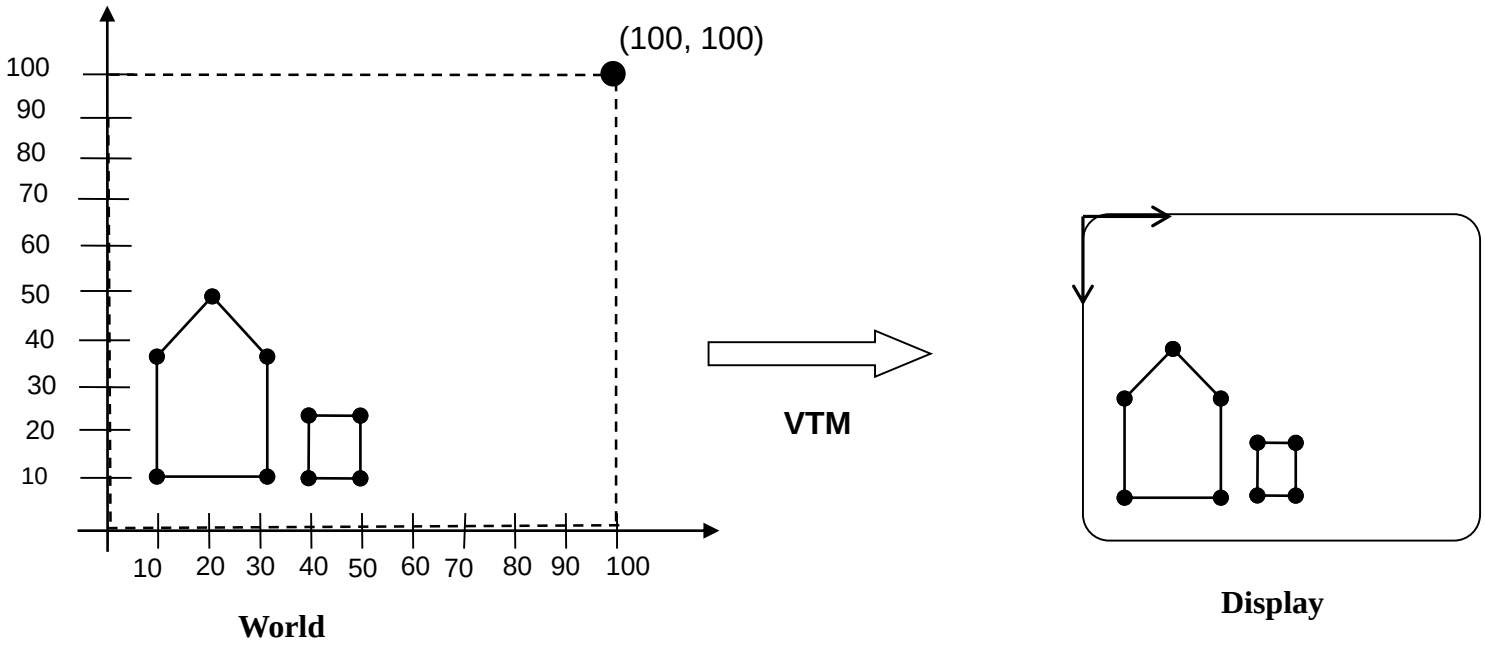
Changing the Window Size (cont.)

- Now we change window size *more*, recompute the VTM, and repaint again



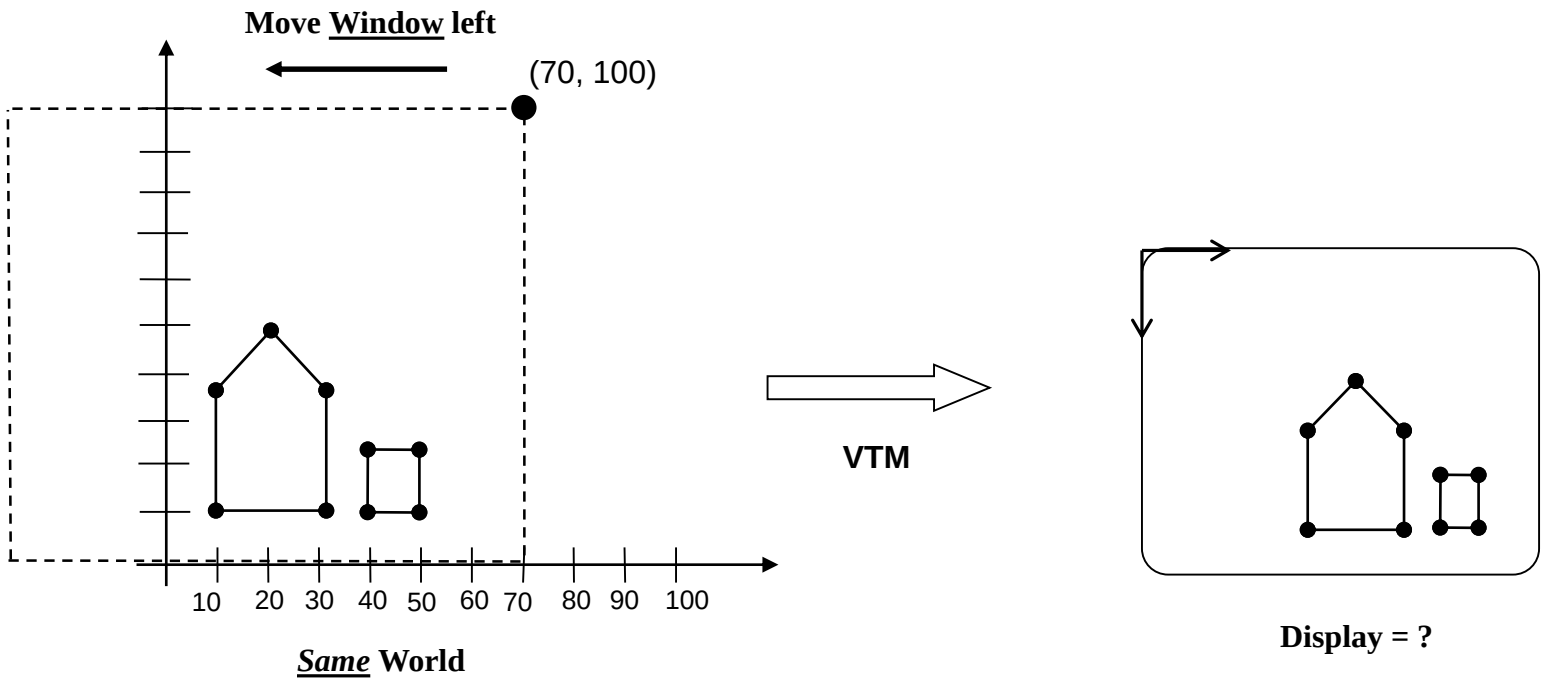
Changing Window *Location*

Suppose we start with this:



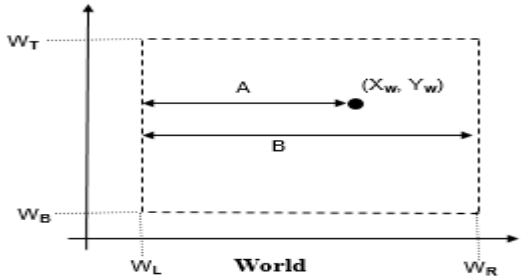
Changing the Window Location (cont.)

Now we change window *location*,
recompute the VTM, and repaint



Adding Zoom and Pan Functionality

```
/* Following methods should be added to CustomContainer to allow zooming and panning */
public void zoom(float factor) {
    //positive factor would zoom in (make the worldWin smaller), suggested value is 0.05f
    //negative factor would zoom out (make the worldWin larger), suggested value is -0.05f
    //...[calculate winWidth and winHeight]
    float newWinLeft = winLeft + winWidth*factor;
    float newWinRight = winRight - winWidth*factor;
    float newWinTop = winTop - winHeight*factor;
    float newWinBottom = winBottom + winHeight*factor;
    float newWinHeight = newWinTop - newWinBottom;
    float newWinWidth = newWinRight - newWinLeft;
    //in CN1 do not use world window dimensions greater
    if (newWinWidth <= 1000 && newWinHeight <= 1000 && newWinWidth > 0 && newWinHeight > 0 ){
        winLeft = newWinLeft;
        winRight = newWinRight;
        winTop = newWinTop;
        winBottom = newWinBottom;
    }
    else
        System.out.println("Cannot zoom further!");
    this.repaint();
}
public void panHorizontal(double delta) {
    //positive delta would pan right (image would shift left), suggested value is 5
    //negative delta would pan left (image would shift right), suggested value is -5
    winLeft += delta;
    winRight += delta;
    this.repaint();
}
public void panVertical(double delta) {
    //positive delta would pan up (image would shift down), suggested value is 5
    //negative delta would pan down (image would shift up), suggested value is -5
    winBottom += delta;
    winTop += delta;
    this.repaint();
}
```



Zoom with Pinching in CN1

`Component` build-in class has **`pinch()`** method. You can override it to call the zoom() method in the previous slide to zoom whenever the user pinches the display (the user's two fingers come "closer" together or go "away" from each other on display).

The simulator assumes one finger is always at the screen origin (the top left corner of the screen), hence:

"closer" pinching (zooming out) is simulated by simultaneous right mouse click and mouse movement towards the screen origin.

"away" pinching (zooming in) is simulated by simultaneous right mouse click and mouse movement going away from the screen origin.

Zoom with Pinching in CN1 (cont.)

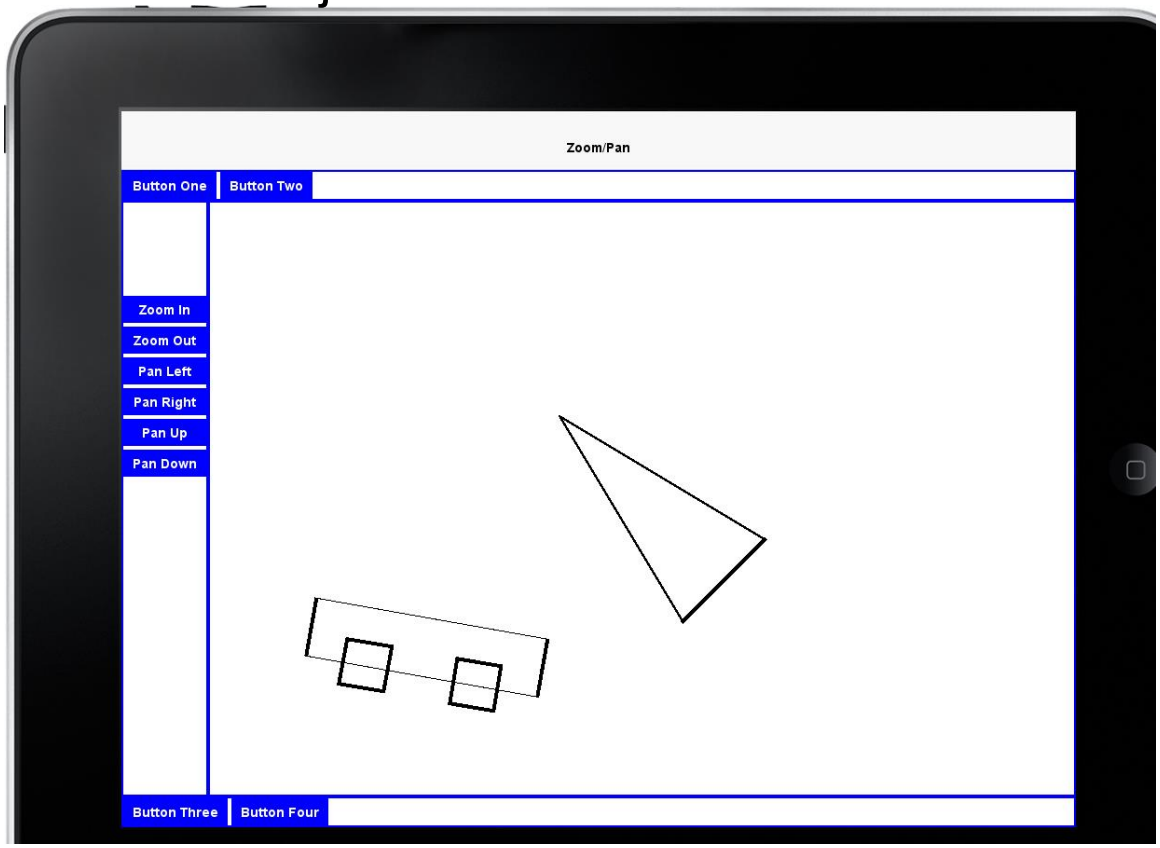
```
/* Override pinch() in CustomContainer to allow zooming with pinching*/
@Override
public boolean pinch(float scale){
    if(scale < 1.0){
        //Zooming Out: two fingers come closer together (on actual device), right mouse
        //click + drag towards the top left corner of screen (on simulator)
        zoom(-0.05f);
    }else if(scale>1.0){
        //Zooming In: two fingers go away from each other (on actual device), right mouse
        //click +  drag away from the top left corner of screen (on simulator)
        zoom(0.05f);
    }
    return true;
}
```

Pan with Pointer Dragging in CN1

```
    /* Override pointerDrag() in CustomContainer to allow panning with a pointer drag which is
simulated with a mouse drag (i.e., simultaneous mouse left click and mouse movement). Below
code moves the world window in the direction of dragging (e.g., dragging the pointer towards
left and top corner of the display would move the object towards the right and top corner of
the display) */
    private Point pPrevDragLoc = new Point(-1, -1);
    @Override
    public void pointerDragged(int x, int y)
    {
    if (pPrevDragLoc.getX() != -1)
    {
        if (pPrevDragLoc.getX() < x)
            panHorizontal(5);
        else if (pPrevDragLoc.getX() > x)
            panHorizontal(-5);
        if (pPrevDragLoc.getY() < y)
            panVertical(-5);
        else if (pPrevDragLoc.getY() > y)
            panVertical(5);
    }

    pPrevDragLoc.setX(x);
    pPrevDragLoc.setY(y);
    }
```

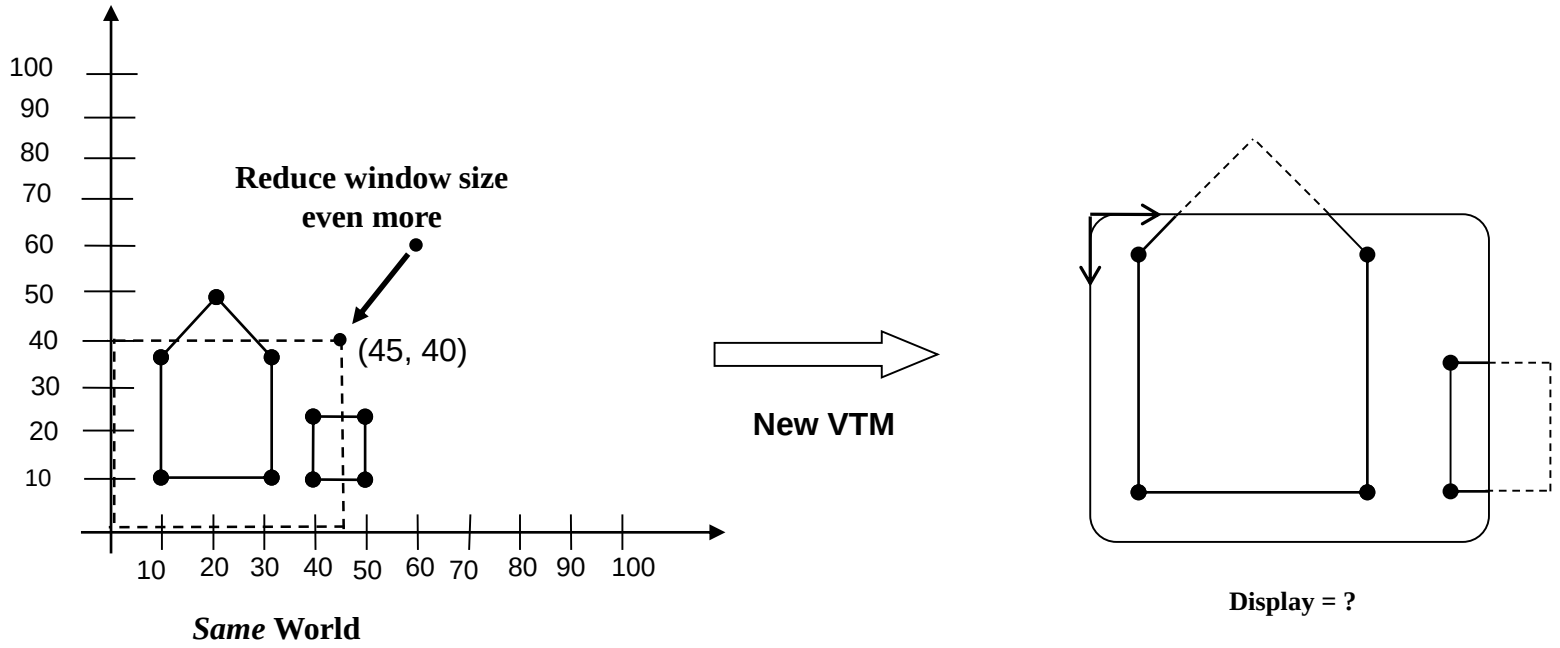
Zoom/Pan App ScreenShot

Create a form with a border layout and put the **`CustomContainer`** object to the center. Call zoom and pan methods of **`CustomContainer`** (with proper parameter values) when the buttons on the west container are clicked and when pinching and pointer dragging happen. In addition to a triangle, draw a hierarchical object on the **`CustomContainer`**.



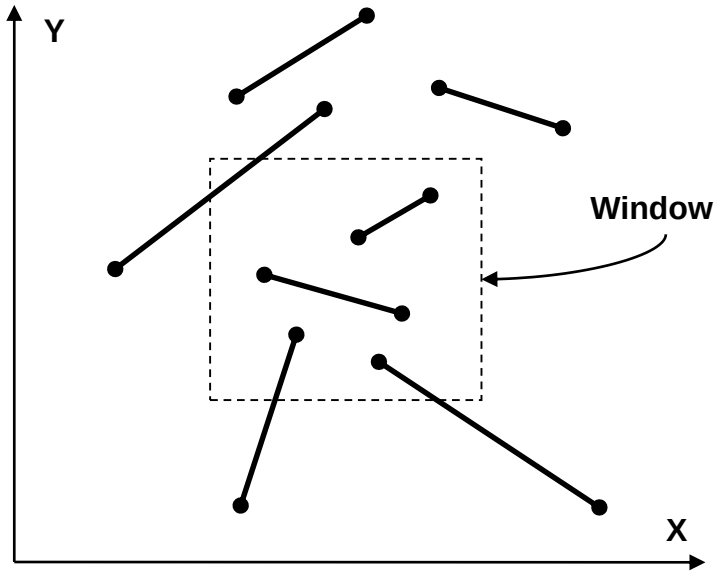
Recall: Changing the Window Size

- Now we change window size *more*, recompute the VTM, and repaint again



Clipping

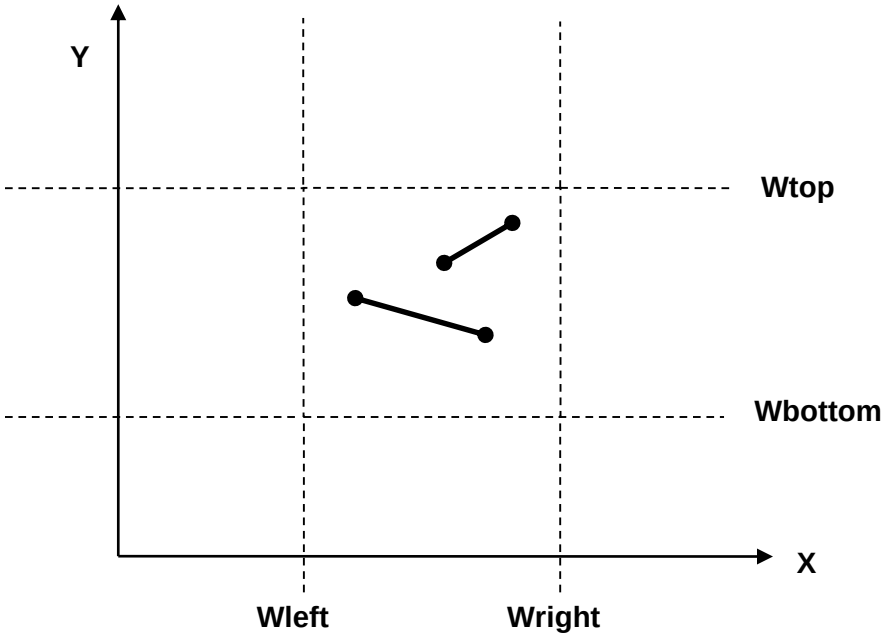
- **Need to suppress output that lies outside the window**
- **For lines, various possibilities:**
  - Both endpoints inside (totally visible)
  - One point inside, the other outside (partially visible)
  - Both endpoints outside (totally invisible ?)



Visibility Tests

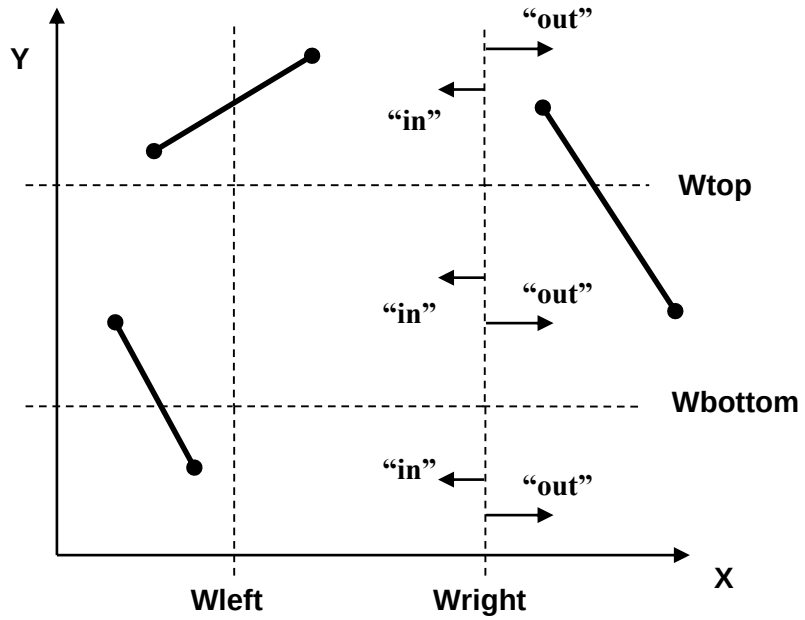
- **"Trivial Acceptance"**
  - Line is completely visible if both endpoints are:

`Below Wtop && Above Wbottom && rightOf Wleft && leftOf Wright`



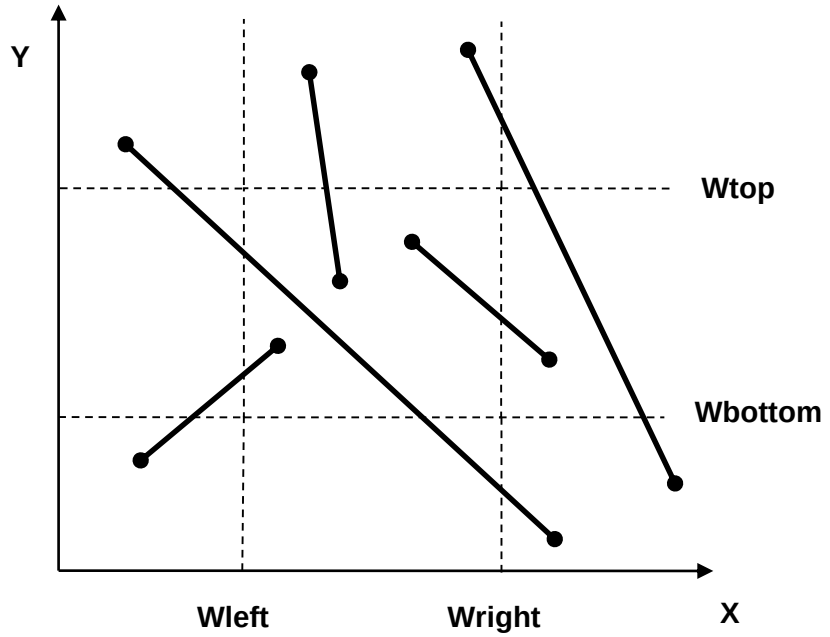
Visibility Tests (cont.)

- **"Trivial Rejection"**
  - Line is completely invisible if both endpoints are on the "out" side of any window boundary



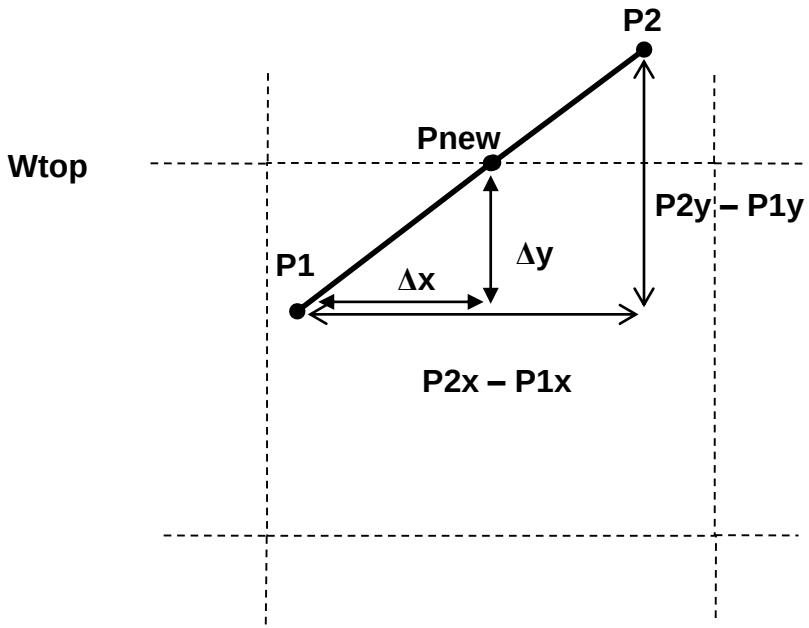
Visibility Tests (cont.)

- Some cases cannot be trivially accepted or rejected :



Clipping Non-Trivial Lines

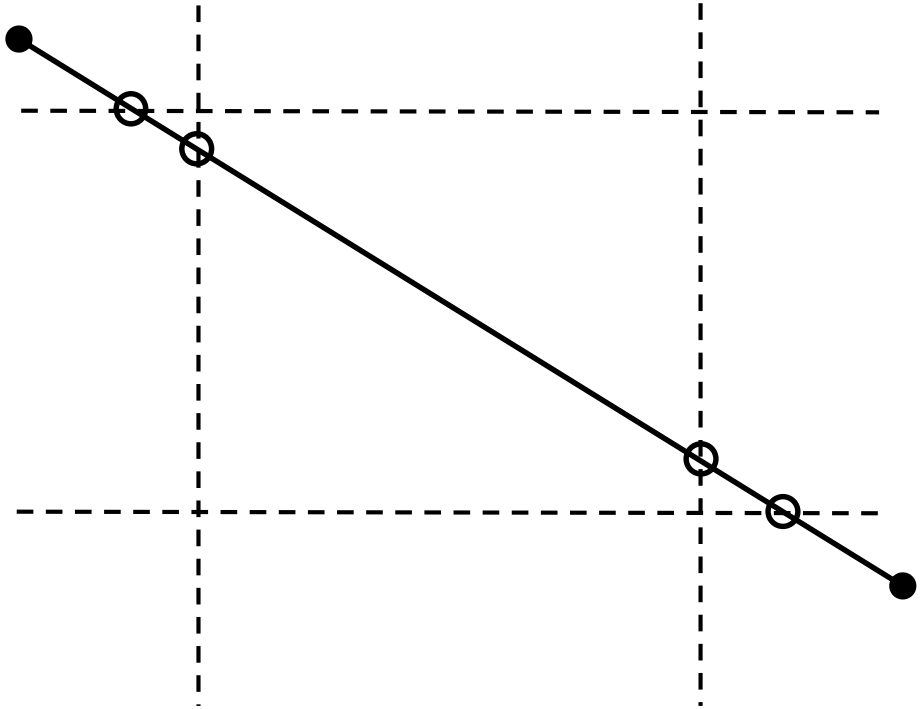
- At least ONE endpoint will be OUTSIDE
  - Compute intersection with (some) boundary
  - Replace “outside” point with Intersection point
  - Repeat as necessary (i.e. until acceptance or empty)




```
Slope = (P2y-P1y) / (P2x-P1x)

PnewY = Wtop

Δy / Δx = Slope

Δy = PnewY – P1y

Δx = Δy / Slope

PnewX = P1x + Δx
```

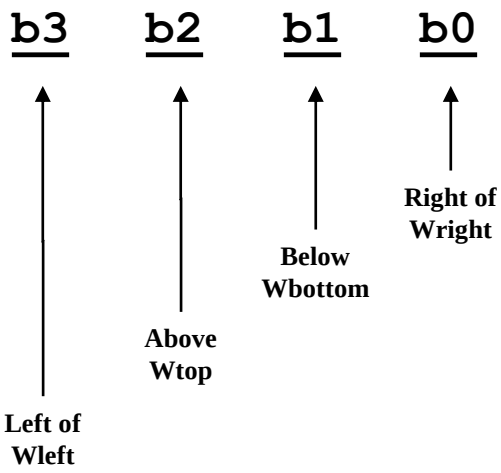
Clipping Non-Trivial Lines (cont.)

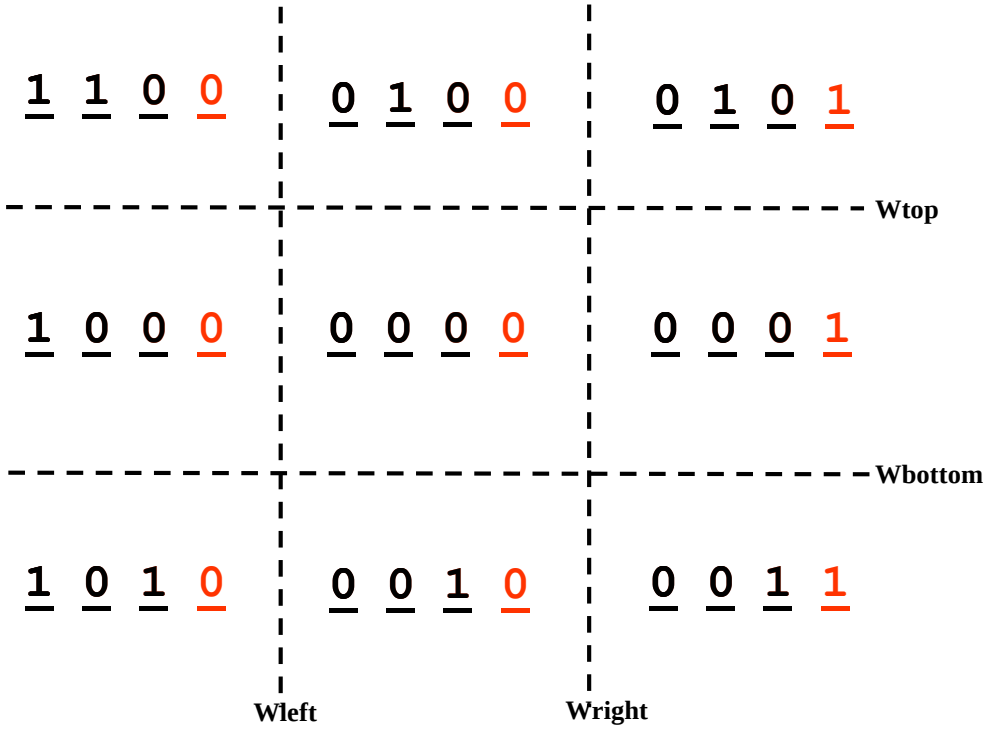
- Replacement may have to be done as many as FOUR times:



Cohen-Sutherland Clipping

- Assign ***4-bit codes*** to each Region
  - Each bit-position corresponds to IN/OUT for one boundary





Cohen-Sutherland Clipping (cont.)

- Compare the bit-codes for line end-points
  - Both codes = 0 → trivial acceptance!
    - Center (window) is the only region with code 0000
  - Logical AND of codes != 0 → trivial rejection!

code(P1):	b3	b2	b1	b0
code(P2):	b3	b2	b1	b0
code1 AND code2:	?	?	?	?

← What's required for this to be non-zero?

The Cohen-Sutherland Algorithm

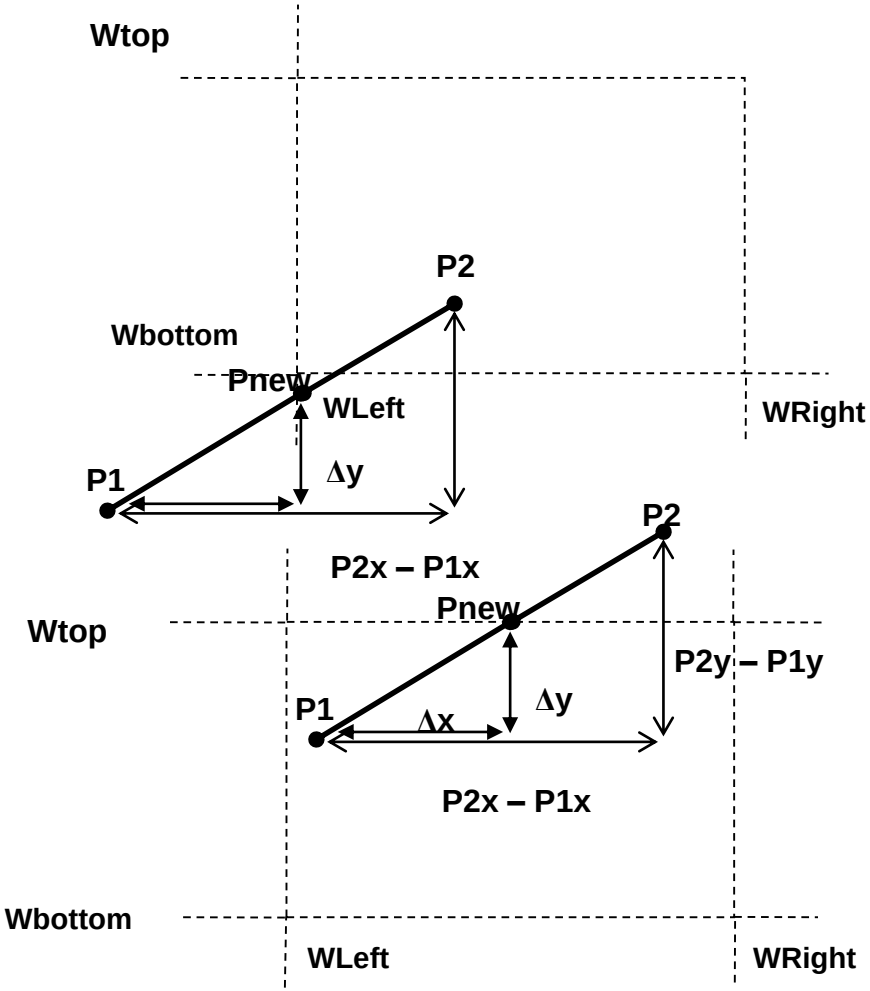
```
/** Clips the line from p1 to p2 against the current world window. Returns the visible
 *  portion of the input line, or returns null if the line is completely outside the window.
 */
Line CSClipper (Point p1,p2) {

  c1 = code(p1); //assign 4-bit CS codes for each input point
  c2 = code(p2);

  // loop until line can be "trivially accepted" as inside the window
  while not (c1==0 and c2==0) {

    // Bitwise-AND codes to check if the line is completely invisible
    if ((c1 & c2) != 0) {
      return null ;     // (logical-AND != 0) means we should reject entire line
    }

    // swap codes so P1 is outside the window if it isn't already
    // (the intersectWithWindow routine assumes p1 is outside)
    if (c1 == 0) {               // if P1 is inside the window
      swap (p1,c1, p2, c2);      // swap points and codes
    }

    // replace P1 (which is outside the window) with a point on the intersection
    // of the line with an (extended) window edge
    p1 = intersectWithWindow (p1, p2);
    c1 = code(p1) ; // assign a new code for the new p1
  }

  return ( new Line(p1,p2) ) ; // the line is now completely inside the window
}
```

The Cohen-Sutherland Algorithm

```
/** Returns a new Point which lies at the intersection of the line p1-p2 with an
 * (extended) window edge boundary line. Assumes p1 is outside the current window.
 */
Point intersectWithWindow (Point p1,p2) {
   if (p1 is above the Window) {
     // find the intersection of line p1-p2 with the window TOP
     x1 = intersectWithTop (p1,p2);          // get the X-intersect
     y1 = windowTop ;
   } else if (p1 is below the Window) {
     // find the intersection of p1-p2 with the window BOTTOM
     x1 = intersectWithBottom (p1,p2);       // get the X-intersect
     y1 = windowBottom ;
   } else if (p1 is left of the window) {
     // find intersect of p1-p2 with window LEFT side
     x1 = windowLeft  ;
     y1 = intersectWithLeftside (p1,p2)      // get the y-intersect
   } else if (p1 is right of the window) {
     // find intersection with RIGHT side
     x1 = windowRight ;
     y1 = intersectWithRightside (p1,p2);  // get the y-intersect
   } else {
     return null ; // error - p1 was not outside
   }
   // (x1,y1) is the improved replacement for p1
   return ( new Point(x1,y1) );
}
```

Rules for computing X/Y coordinates

- If line crosses Wleft/Wright then:
  - x = Wleft/Wright
  - y = y1 + m (x-x1)
  - m = slope
- If line crosses WBottom/WTop then:
  - y = Wbottom/Wtop
  - x = x1 + (y-y1) /m
  - m = slope



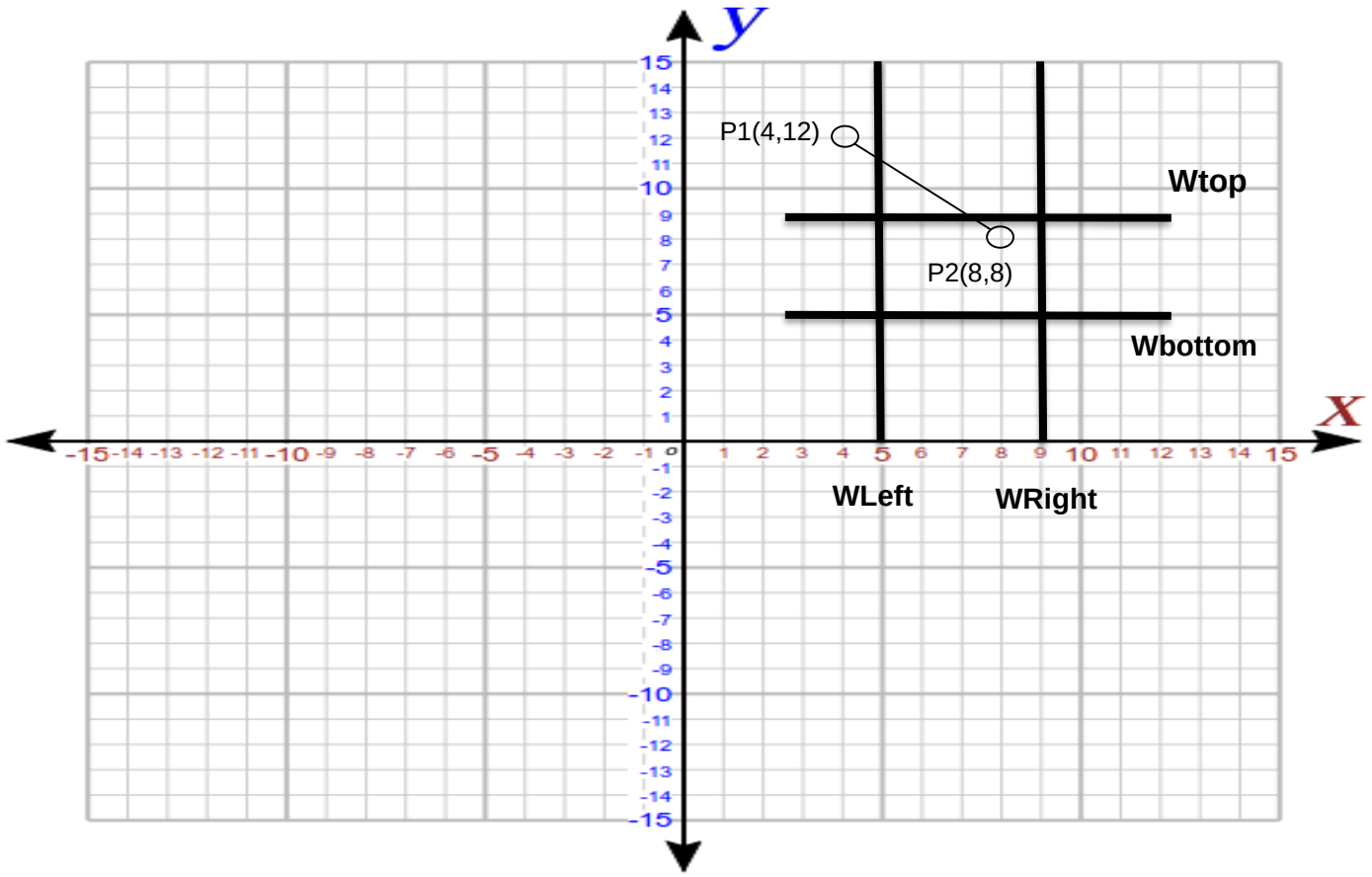
Practice Problem

- A clipping window has the following property:
  - Window(left,right,bottom,top) = (5,9,5,9)
  - A Line, L1, with the following end points is drawn in the word: P1 (4,12), P2 (8,8)
  - Show how the Cohen-SutherLand clipping algorithm will clip this line and what its final endpoints are?

Practice Problem (Cont)

- Draw window and points

  Window(left,right,bottom,top) = (5,9,5,9)
  A Line, L1, with the following end points is drawn in the word: P1 (4,12), P2 (8,8)



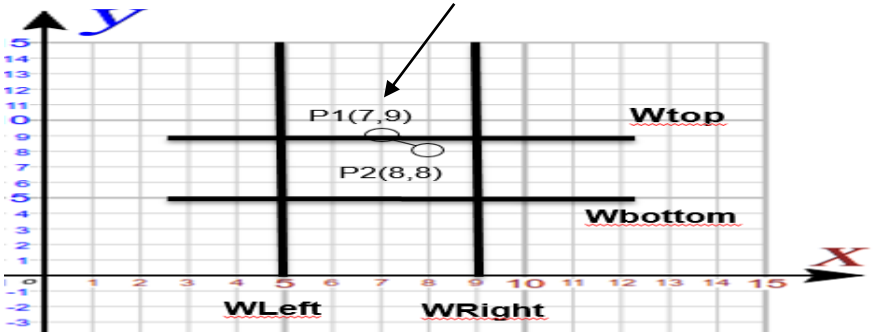
Practice Problem (Cont)

- Assign **4-bit CS codes for each input point: c1=code(p1)=1100, c2=code(p2)= 0000**
- **(c1 & c2) = 0000 – Not Trivial rejection** (Line is NOT completely invisible)
- P1 is not inside the window. Thus, no swapping.
- **Compute new p1: p1 = intersectWithWindow (p1, p2);**
  - line crosses Wleft/Wright then: x = Wleft/Wright , y = y1 + m (x-x1), m=slope
  - x=5, y=12+-1(5-4)=11, slope=(12-8)/(4-8)=-1
  - Line crosses WBottom/WTop then: y = Wbottom/Wtop, x = x2 + (y-y2) /m, m = slope
  - y=9, x=8+(9-8)/(-1)=7, slope=-1 (same as above step)
  - New p1 = (7,9) // ←---------- New P1

New p1 = (7,9)

- **c1=code(new p1)=0000**

(Before Clipping)



(After Clipping)

- **Now, both c1 and c2 are 0000, we return new line(p1,p2)**