CSc 133 Lecture Notes

15 - Viewing Transformations

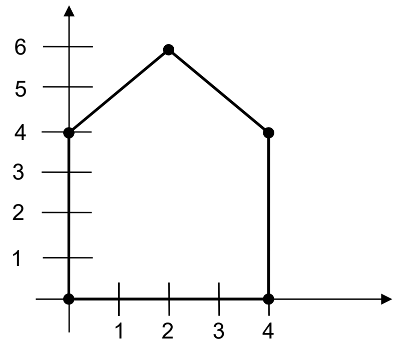
Computer Science Department
California State University, Sacramento

Overview

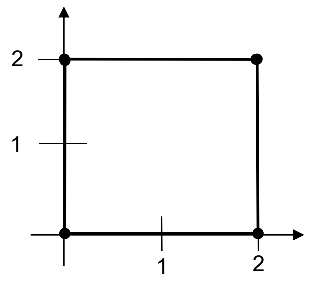
- The World Coordinate System
- Mapping From World to Display Coordinates
  - World Window, Normalized Device (ND), World-to-ND Transform, ND-to-Display Transform, the Viewing Transformation Matrix (VTM)
- 2D Viewing Operations (Zoom and Pan)
- Mapping User Input to World Coordinates
- Clipping and the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

Local Coordinate Systems

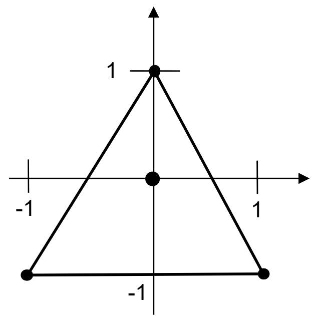
- Each object is defined in its "own space"



Pentagon

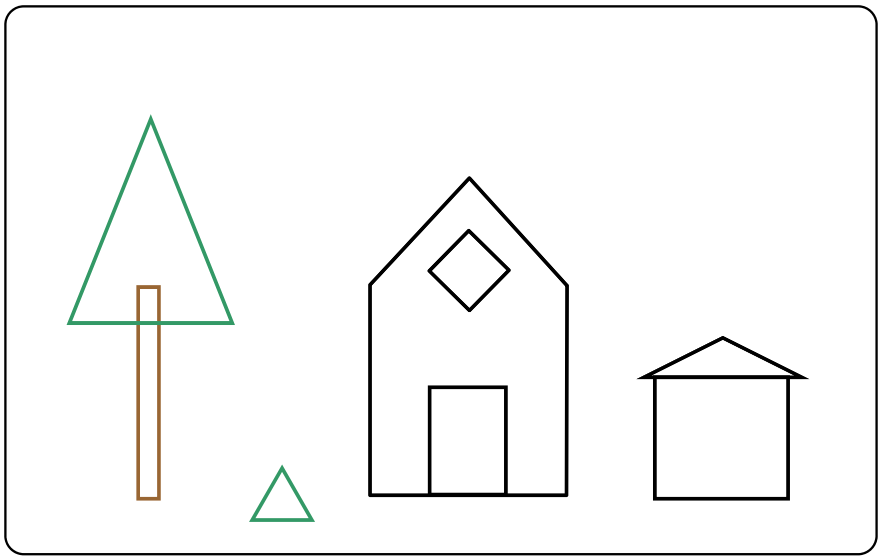


Box



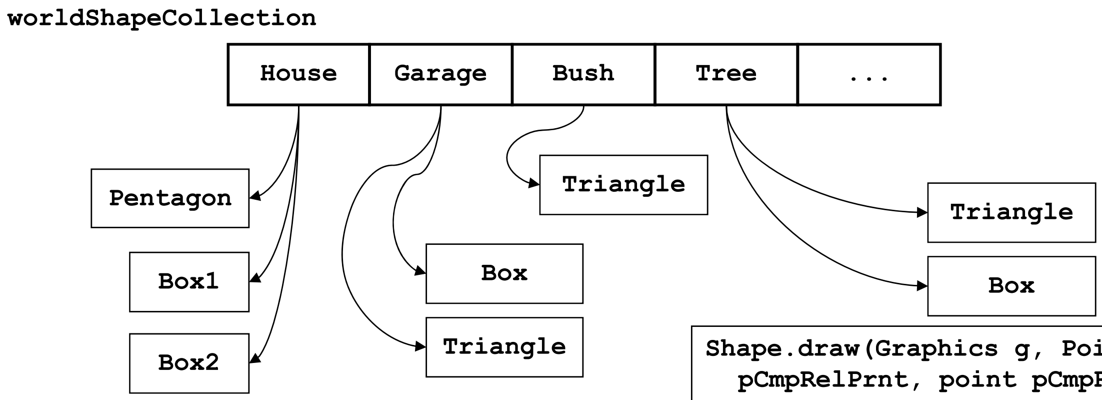
Triangle

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();
}
```

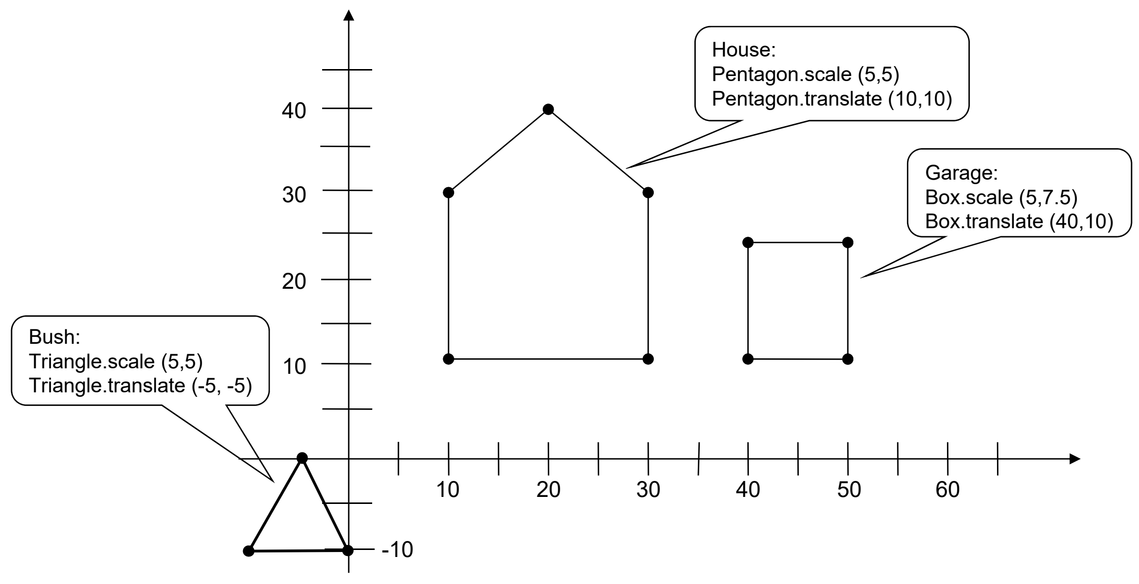
```
Shape.draw(Graphics g, Point
  pCmpRelPrnt, point pCmpRelScrn)
{ save xform in g as gOrigXform;
  apply LTs and "local origin"
                transforms to g;
  for (each sub shape) {
   sub.draw(g, pCmpRelPrnt,
                pCmpRelScrn);

  }
 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:



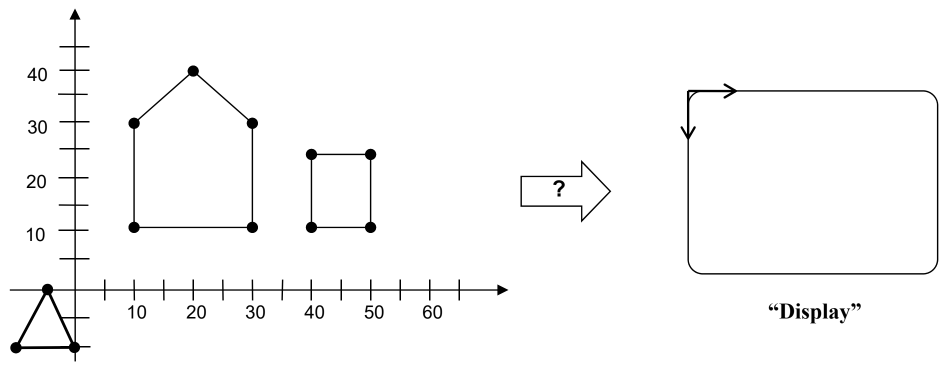
Local Transformations

- 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.)

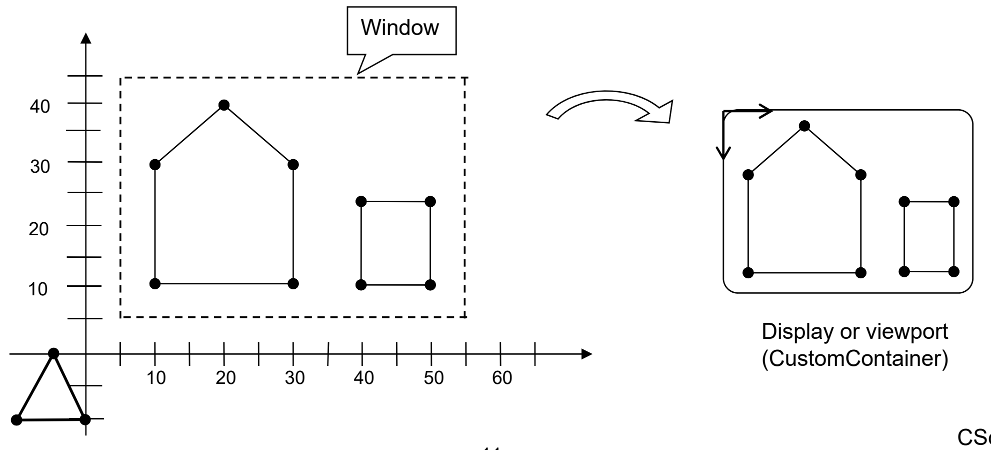
Solution:

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

The World "Window"

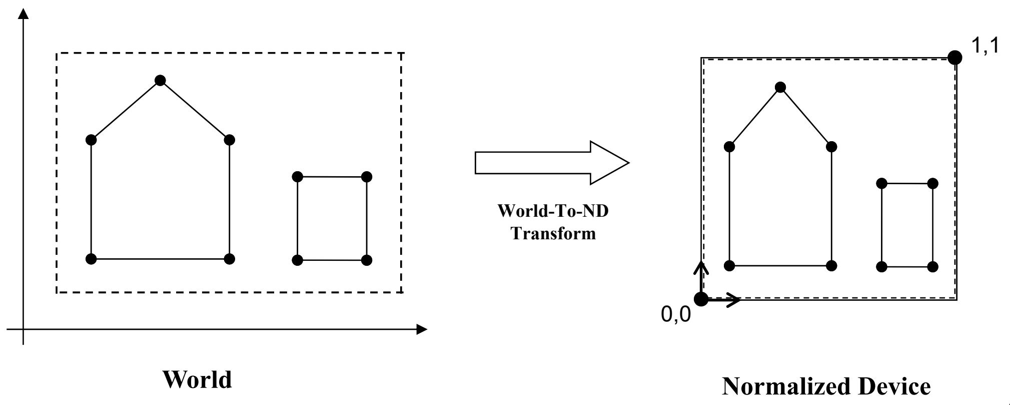
Defines the part of the world that appears on display

- 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"

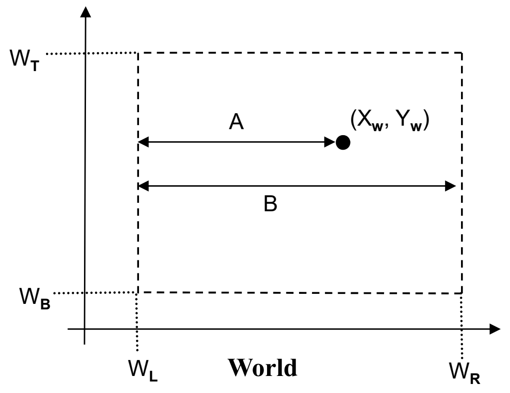
- Properties of the Normalized Device (ND):
  - Square
  - Fractional Coordinates (0.0 .. 1.0)
  - Origin at Lower Left
  - Corners correspond to world window



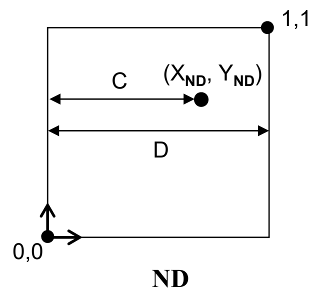
World-To-ND Transform

Consider a single point's X coordinate

- Need to achieve proportional positioning on the ND



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



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

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

World-To-ND Transform (cont.)

Consider the form of X_{ND} :

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

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

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

World-To-ND Transform (cont.)

X_{ND} = (X_W • Translate ($-W_L$)) • Scale (1 / WindowWidth)
Y_{ND} = (Y_W • Translate ($-W_B$)) • Scale (1 / WindowHeight)
or
P_{ND} = (P_W • Translate ($-W_L$, $-W_B$)) • Scale (1/WindowWidth, 1/WindowHeight)

- In Matrix Form:

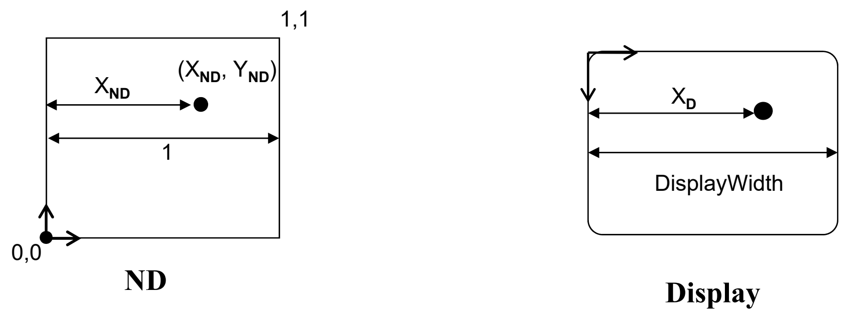
$$\begin{pmatrix} X_{ND} \\ Y_{ND} \\ 1 \end{pmatrix} = \left(\begin{pmatrix} 1/W_w & 0 & 0 \\ 0 & 1/W_h & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -W_L \\ 0 & 1 & -W_B \\ 0 & 0 & 1 \end{pmatrix} \right) \begin{pmatrix} X_W \\ Y_W \\ 1 \end{pmatrix}$$

X

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

ND-To-Display Transform

- A similar approach can be applied

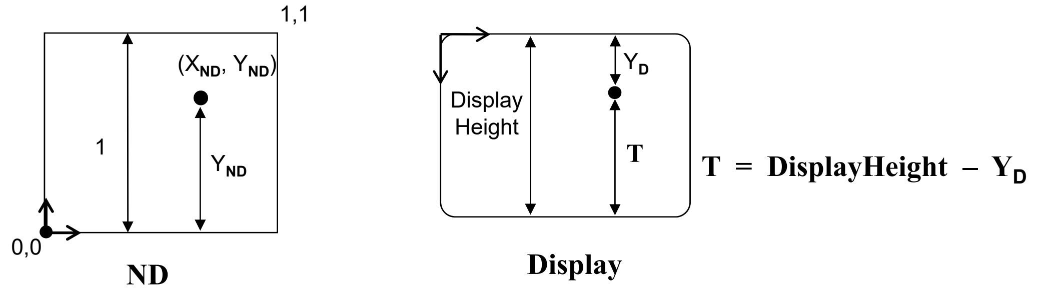


$$\frac{X_{ND}}{1} = \frac{X_D}{DisplayWidth} \; ; \quad \therefore \quad X_D = X_{ND} \times DisplayWidth$$

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

ND-To-Display Transform (cont.)

- Similarly for height:



$$\mathbf{T} = \mathbf{DisplayHeight} - \mathbf{Y_D}$$

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

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

ND-To-Display Transform (cont.)

X_D = (X_{ND} • Scale (DisplayWidth)) • Translate (0)
Y_D = (Y_{ND} • Scale (-DisplayHeight)) • Translate (DisplayHeight)
or
P_D = (P_{ND} • Scale (DisplayWidth, -DisplayHeight)) • Translate (0, DisplayHeight)

- In Matrix Form:

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

Combining Transforms

$$\begin{bmatrix} x_D \\ y_D \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} \text{ND} \\ \text{to} \\ \text{Display} \end{bmatrix} \times \Bigg(\begin{bmatrix} \text{World} \\ \text{to} \\ \text{ND} \end{bmatrix}} \times \begin{bmatrix} x_W \\ y_W \\ 1 \end{bmatrix} \Bigg)$$

$$\begin{bmatrix} x_D \\ y_D \\ 1 \end{bmatrix} = \begin{bmatrix} \text{“VTM”} \end{bmatrix} \begin{bmatrix} x_W \\ y_W \\ 1 \end{bmatrix}$$

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.)

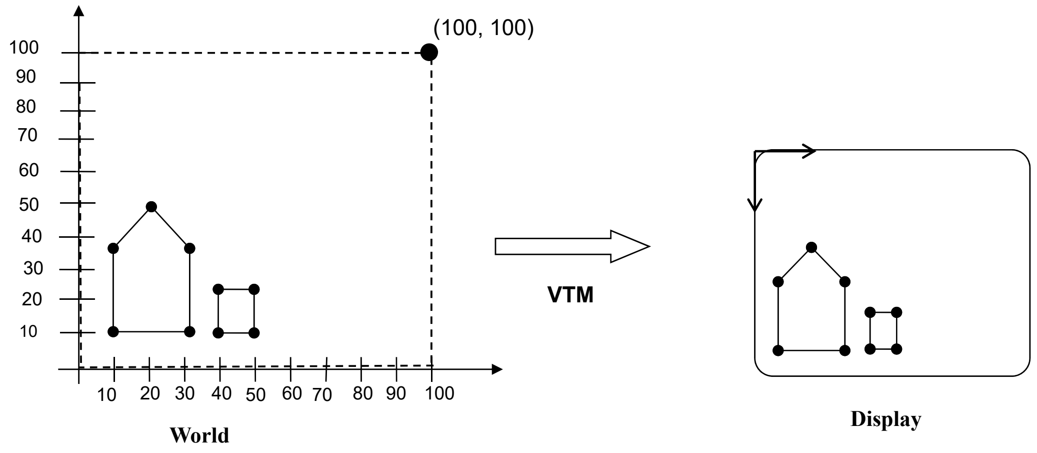
```
public class CustomContainer extends Container {
  Transform worldToND, ndToDisplay, theVTM ;
  private float winLeft, winBottom, winRight, winTop;
  public CustomContainer(){
    //initialize world window
    winLeft = 0;
    winBottom = 0;
    winRight = 931/2; //hardcoded value = this.getWidth()/2 (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
```

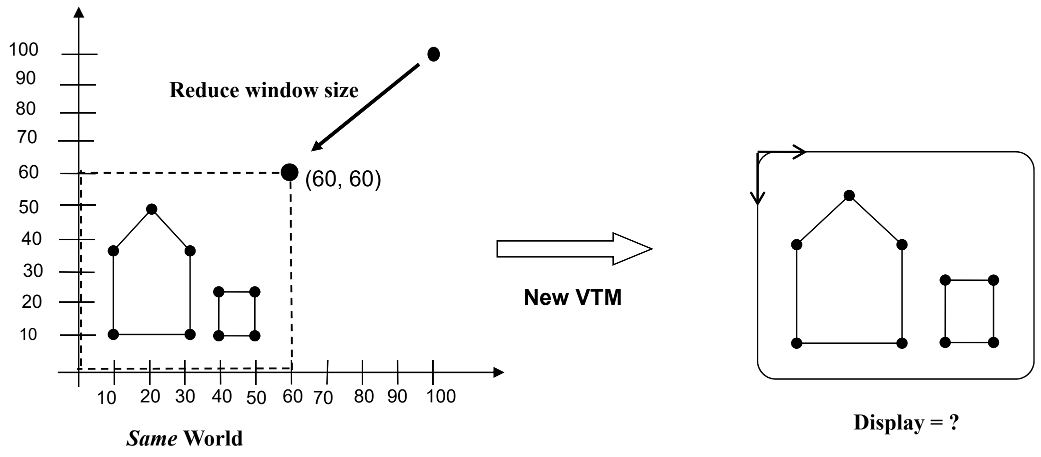
Changing the Window Size

Suppose we start with this:



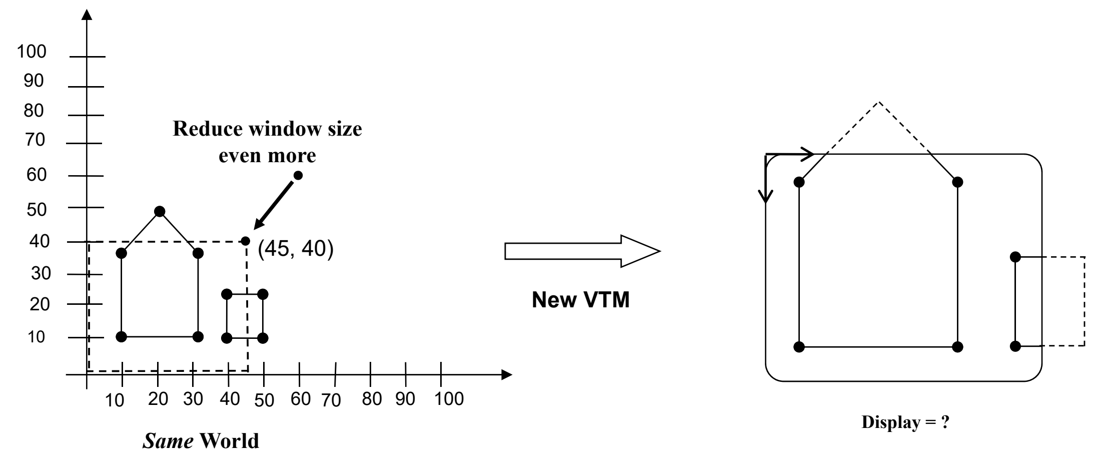
Changing the Window Size (cont.)

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



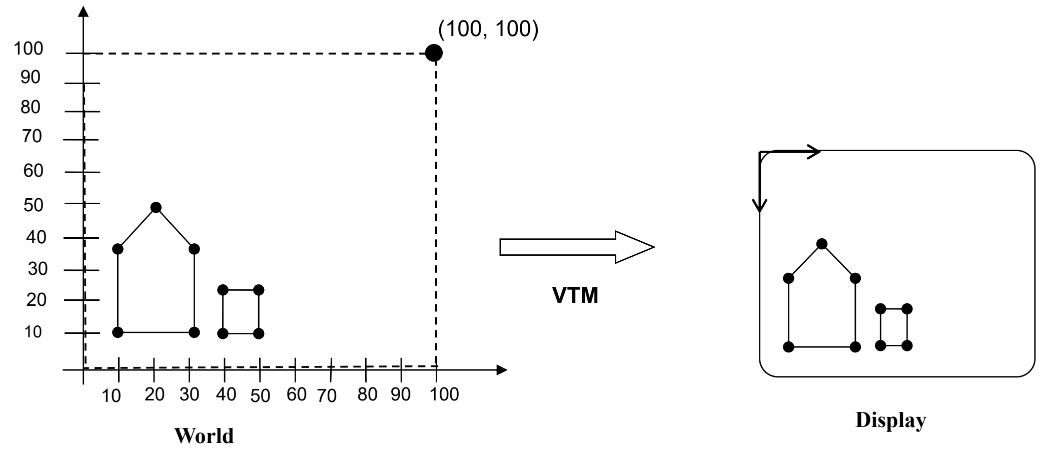
Changing the Window Size (cont.)

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



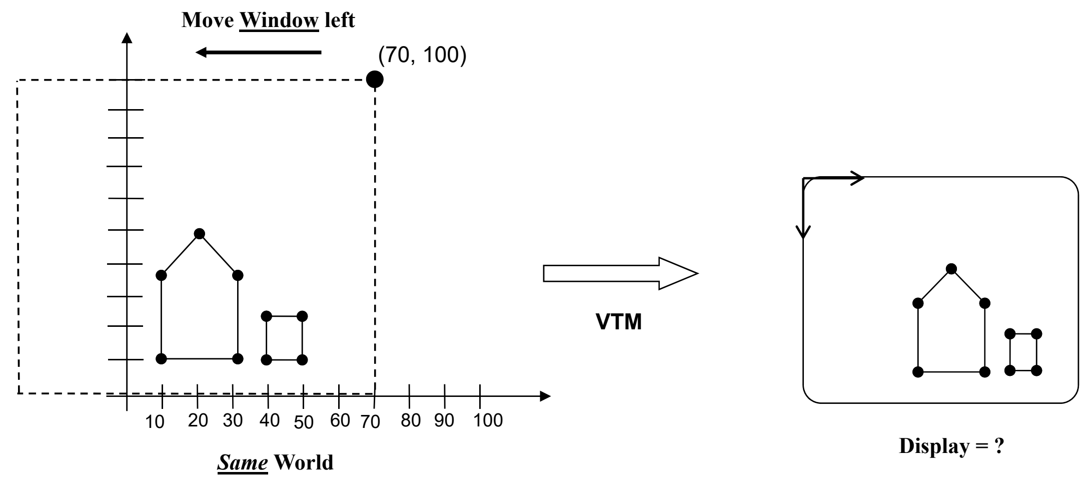
Changing Window *Location*

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 than 1000!!!
    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` built-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.)

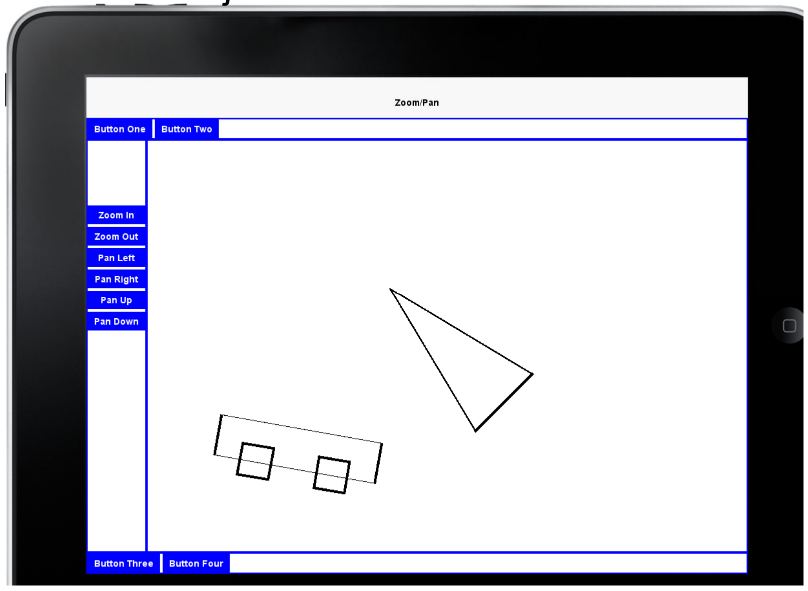
```
/* Override pinch() in CustomContainer to allow zooming with pinching*/
@Override
public boolean pinch(float scale){
    if(scale < 1.0){
        //Zooming Out: two fingers come closer together (on actual device), right mouse
        //click + drag towards the top left corner of screen (on simulator)
        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 bottom 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);
   }
```

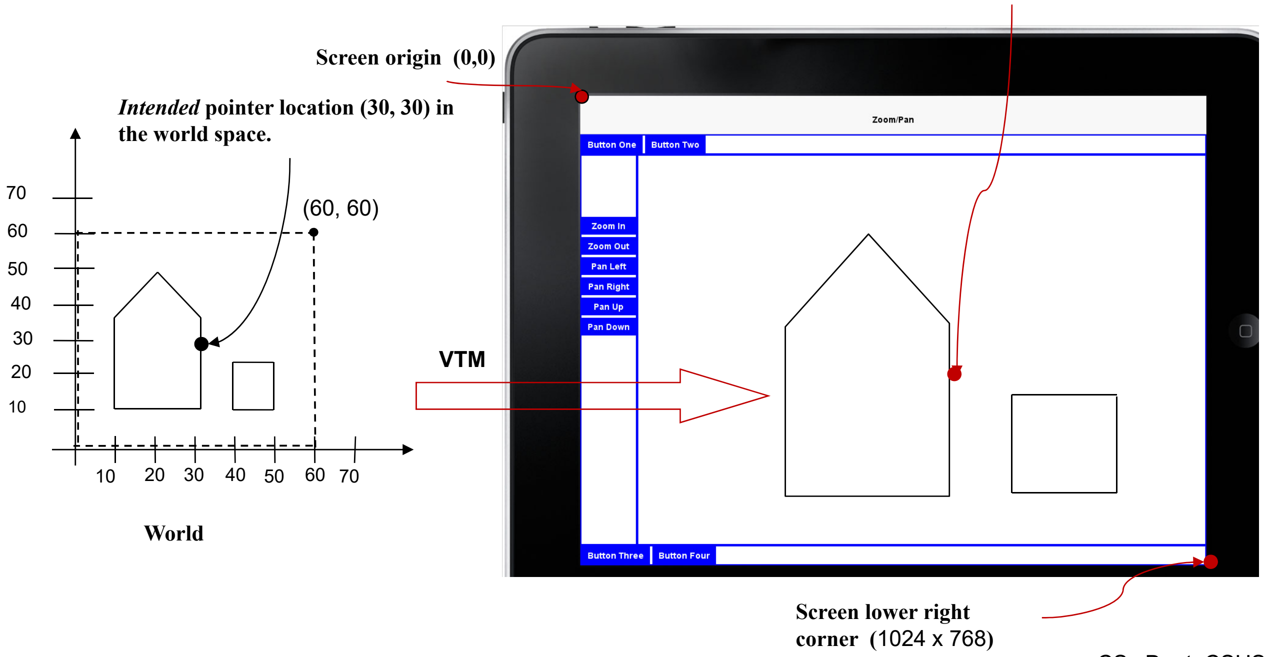
Zoom/Pan App ScreenShot

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`**.



Mapping Pointer To World Coords

Pointer click location: assume it is at exactly the middle of the display (CustomContainer) Since myCustomContainer.getAbsoluteX() = 93, myCustomContainer.getAbsoluteY() = 96 myCustomContainer.getWidth() = 931, myCustomContainer.getHeight() =639 then the pointer location in the screen space would be (93+931/2, 96 +639/2) = (559, 416)



Mapping Pointer To World Coords

When *drawing* (outputting) points, we apply the following transform:

$$\begin{pmatrix} X_D \\ Y_D \\ 1 \end{pmatrix} = \begin{pmatrix} \quad \mathrm{VTM} \quad \end{pmatrix} \times \begin{pmatrix} X_w \\ Y_w \\ 1 \end{pmatrix}$$

Display Point **World Point**

(Note that, in CN1, as usual, we also apply “local origin” transformations before and after applying VTM)

Mapping Pointer To World Coords (cont.)

- In CN1, from **`pointerPressed()`**, we get the pointer location relative to the *Screen* orgin. We make this point relative to the *Display* origin by deducting **`getAbsoluteX()/Y()`** value of the display.
- Then we can calculate the corresponding point in the world space. We need to go "backwards":

$$\begin{pmatrix} X_w \\ Y_w \\ 1 \end{pmatrix} = \begin{pmatrix} & & \\ & ?? & \\ & & \end{pmatrix} \times \begin{pmatrix} X_D \\ Y_D \\ 1 \end{pmatrix}$$

World Point **Display Point (given)**

Mapping Pointer To World Coords (cont.)

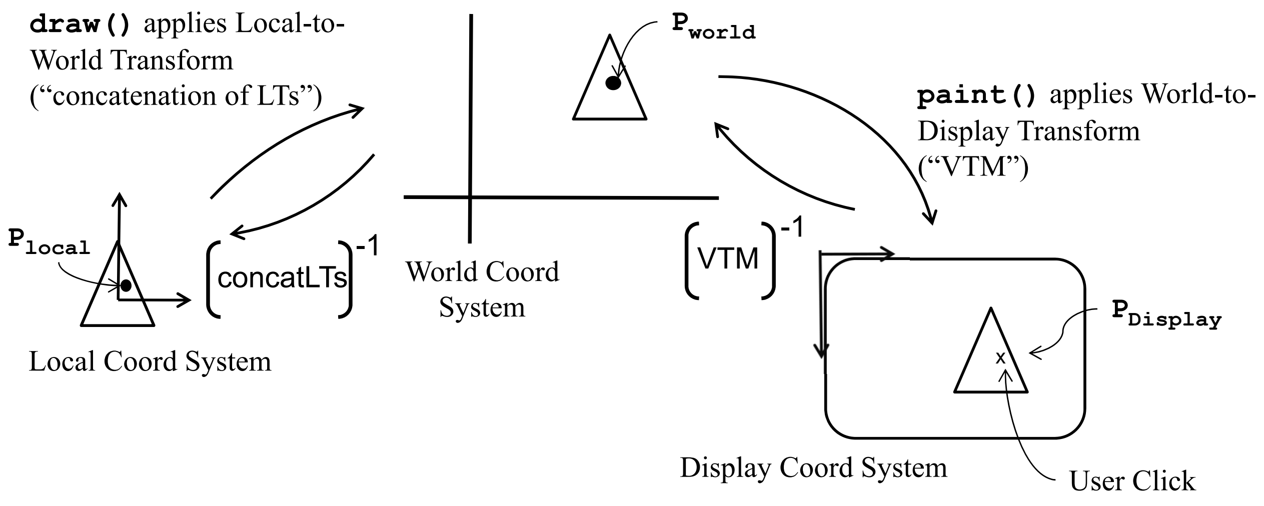
- **We need the *inverse* of the VTM**

$$\begin{pmatrix} X_w \\ Y_w \\ 1 \end{pmatrix} = \begin{pmatrix} & & \\ & \mathrm{VTM} & \\ & & \end{pmatrix}^{-1} \times \begin{pmatrix} X_D \\ Y_D \\ 1 \end{pmatrix}$$

World Point **Display Point**

```
Transform theVTM, inverseVTM ;
// ...code here to define the contents of theVTM.
inverseVTM = Transform.makeIdentity();
try {
   theVTM.getInverse(inverseVTM);
} catch (NotInvertibleException e) {
   System.out.println("Non invertible xform!");}
inverseVTM.transformPoint(fPtr, fPtr); //fPtr is a float array that first holds the
   //point in the display space, then it is transformed into the world space.
```

Selection / Containment


draw() applies Local-to-World Transform ("concatenation of LTs")
P_world
paint() applies World-to-Display Transform ("VTM")
P_local
concatLTs -1
World Coord System
VTM -1
P_Display
x
Local Coord System
Display Coord System
User Click

To generate the concatLTs, same order of transformations used in draw() must be used.

If the object is a hierarchical object the sub-shapes should also apply inverse of their concatLTs to see if they contain the point.

Selection of Hierarchical Objects

```
/* Create a hierarchical object in the constructor of CustomContainer */
public class CustomContainer extends Container {
 //...[constructor code starts]
 myHierObj = new HierObj((int)(winHeight/5));
 myHierObj.translate(winWidth/4, winHeight/4);
 myHierObj.rotate(-10);
 myHierObj.scale(2, 1);
 //...[rest of the constructor code]
}
/* Also, override the pointerPressed() in CustomContainer to get the pointer location in screen
space */
@Override
public void pointerPressed(int x, int y){
 //(x, y) is the pointer location relative to screen origin
 //make it relative to display origin
 float [] fPtr = new float [] {x - getAbsoluteX(), y - getAbsoluteY()};
 Transform inverseVTM = Transform.makeIdentity();
 try {
  theVTM.getInverse(inverseVTM);
 } catch (NotInvertibleException e) {
  System.out.println("Non invertible xform!");}
 //calculate the location of fPtr the in world space
 inverseVTM.transformPoint(fPtr, fPtr);
 if (myHierObj.contains(fPtr))
  myHierObj.setSelected(true);
 else
  myHierObj.setSelected(false);
 repaint();
}
```

Selection of Hierarchical Objects (cont.)

```
/* The constructor of the hierarchical object build the object from three sub-objects (all of which are
based on a "square" primitive). */
public HierObj(int size) {
 // create an array to hold the sub-objects
 sobjs = new Square[3];
 Square body = new Square(size); body.scale(1.0f, 0.5f); sobjs[0] = body;
 Square leg0 = new Square(size/5); leg0.scale(1.0f, 2f);
 leg0.translate(-size/4, -size/4); sobjs[1] = leg0;
 Square leg1 = new Square(size/5); leg1.scale(1.0f, 2f);
 leg1.translate(size/4, -size/4); sobjs[2] = leg1;
 //...[rest of the constructor code]}
/* contains() of HierObj apply inverse concatLTs to fPtr and call sub-objects's contains() methods*/
public boolean contains(float[] fPtr) {//signature of contains() in ISelectable has to be updated
 //concatenate all LTs (make sure to follow the same transformation order used in draw())
 Transform concatLTs = Transform.makeIdentity();
 concatLTs.translate(myTranslation.getTranslateX(), myTranslation.getTranslateY());
 concatLTs.concatenate(myRotation);
 concatLTs.scale(myScale.getScaleX(), myScale.getScaleY());
 //calculate inverse of concatLTs
 Transform inverseConcatLTs = Transform.makeIdentity();
 try {
  concatLTs.getInverse(inverseConcatLTs);
 } catch (NotInvertibleException e) {
  System.out.println("Non invertible xform!");}
 //fPtr is in the world space, calculate the corresponding point in the local space of HierObj
 inverseConcatLTs.transformPoint(fPtr, fPtr);
 for (Square sobj : sobjs) {
  //make sure that the point is not already transformed with previous sub-object's inverse concatLTs
  float[] fPtrCopy =  new float[] {fPtr[0], fPtr[1]};
  if ( sobj.contains(fPtrCopy))
    return true;
  }
 return false;}
```

Selection of Hierarchical Objects (cont.)

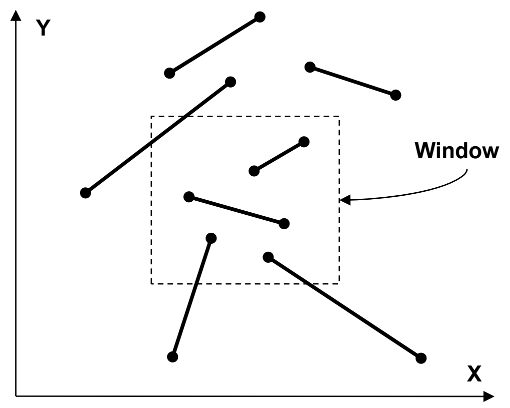
```
/* setSelected() of HierObj call sub-object's setSelected() methods, so that if a sub-object of HierObj
gets selected/unselected, all sub-objects would get selected/unselected */
public void setSelected(boolean b) {
 for (Square sobj : sobjs)
  sobj.setSelected(b);
}
/* The constructor of the square define the object points in local space */
public Square(int givenSize) {
 size = givenSize;
 lowerLeftInLocalSpace = new Point(-size/2, -size/2); //corresponds to upper left corner on screen
 //...[rest of the constructor code]
}
/* contains() of Square apply inverse concatLTs to fPtr and checks if the point is inside*/
public boolean contains(float[] fPtr) {
 //concatenate all LTs (make sure to follow the same order used in draw())
 Transform concatLTs = Transform.makeIdentity();
 concatLTs.translate(myTranslation.getTranslateX(), myTranslation.getTranslateY());
 concatLTs.concatenate(myRotation);
 concatLTs.scale(myScale.getScaleX(), myScale.getScaleY());
 Transform inverseConcatLTs = Transform.makeIdentity();
 try {  concatLTs.getInverse(inverseConcatLTs);
 } catch (NotInvertibleException e) {
  System.out.println("Non invertible xform!");}
 //fPtr is in the local space of HierObj, calculate the corresponding point in the local space of Square
 inverseConcatLTs.transformPoint(fPtr, fPtr);
 int px = (int)fPtr[0]; //pointer location relative to
 int py = (int)fPtr[1]; //local origin
 int xLoc = lowerLeftInLocalSpace.getX(); //square lower left corner
 int yLoc = lowerLeftInLocalSpace.getY(); //location relative to local origin
 if ( (px >= xLoc) && (px <= xLoc+size)  && (py >= yLoc) && (py <= yLoc+size) )
  return true;
 else
  return false;}
```

Selection of Hierarchical Objects (cont.)

```
/* setSelected() of Square would set the bSelected class field as before...*/
public void setSelected(boolean b) {
 bSelected = b;
}
/* also, draw() of Square would draw the shape by checking bSelected as before...*/
public void draw(Graphics g,Point pCmpRelPrnt, Point pCmpRelScrn) {
 g.setColor(ColorUtil.BLACK);
 Transform gXform = Transform.makeIdenity();
 g.getTransform(gXform);
 Transform gOrigXform = gXform.copy();
 //apply LTs (also "local origin" transformations)
 gXform.translate(pCmpRelScrn.getX(),pCmpRelScrn.getY());
 gXform.translate(myTranslation.getTranslateX(), myTranslation.getTranslateY());
 gXform.concatenate(myRotation);
 gXform.scale(myScale.getScaleX(), myScale.getScaleY());
 gXform.translate(-pCmpRelScrn.getX(),-pCmpRelScrn.getY());
 g.setTransform(gXform);
 //draw the shape
 if (bSelected)
  g.fillRect(pCmpRelPrnt.getX()+lowerLeftInLocalSpace.getX(),
             pCmpRelPrnt.getY()+lowerLeftInLocalSpace.getY(), size, size);
 else
  g.drawRect(pCmpRelPrnt.getX()+lowerLeftInLocalSpace.getX(),
             pCmpRelPrnt.getY()+lowerLeftInLocalSpace.getY(), size, size);
 g.setTransform(gOrigXform); //restore the original xform
}
```

Clipping

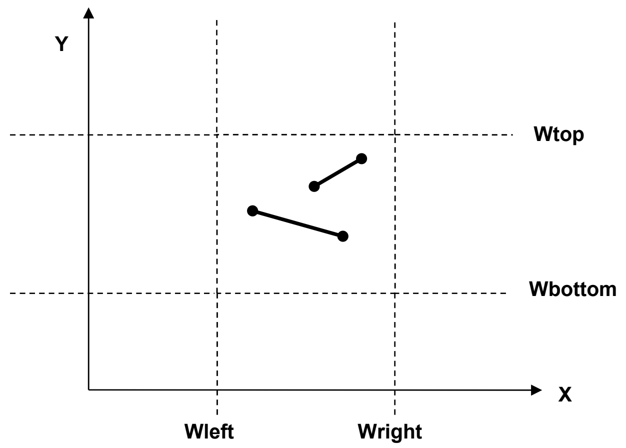
- **Need to suppress output that lies outside the window**
- **For <u>lines</u>, various possibilities:**
  - Both endpoints inside (totally visible)
  - One point inside, the other outside (partially visible)
  - Both endpoints outside (totally invisible ?)



Visibility Tests

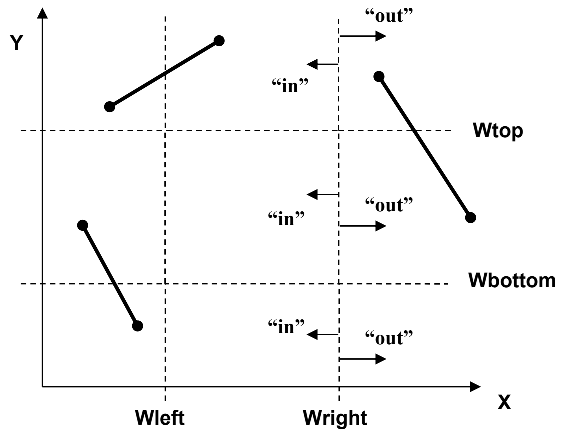
- **"Trivial Acceptance"**
    - Line is completely visible if both endpoints are:

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



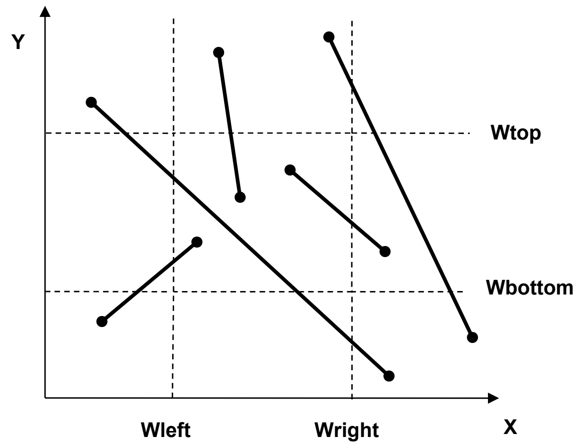
Visibility Tests (cont.)

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



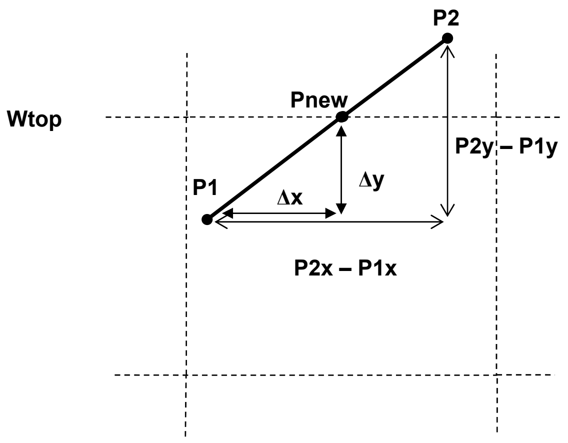
Visibility Tests (cont.)

- Some cases cannot be trivially accepted or rejected :



Clipping Non-Trivial Lines

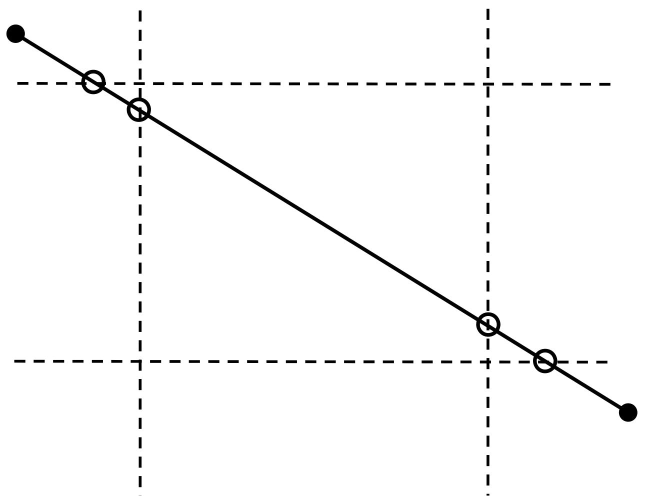
- At least ONE endpoint will be OUTSIDE
  - Compute intersection with (some) boundary
  - Replace “outside” point with Intersection point
  - Repeat as necessary (i.e. until acceptance or empty)



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

PnewY = Wtop

Δy / Δx = Slope

Δy = PnewY – P1y

Δx = Δy / Slope

PnewX = P1x + Δx
```

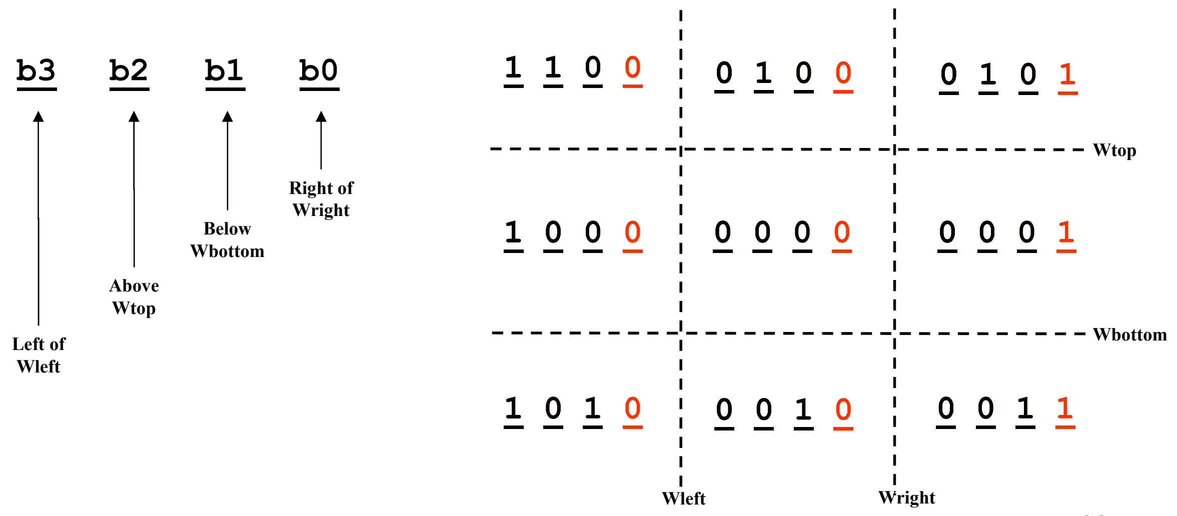
Clipping Non-Trivial Lines (cont.)

- 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) );
}
```