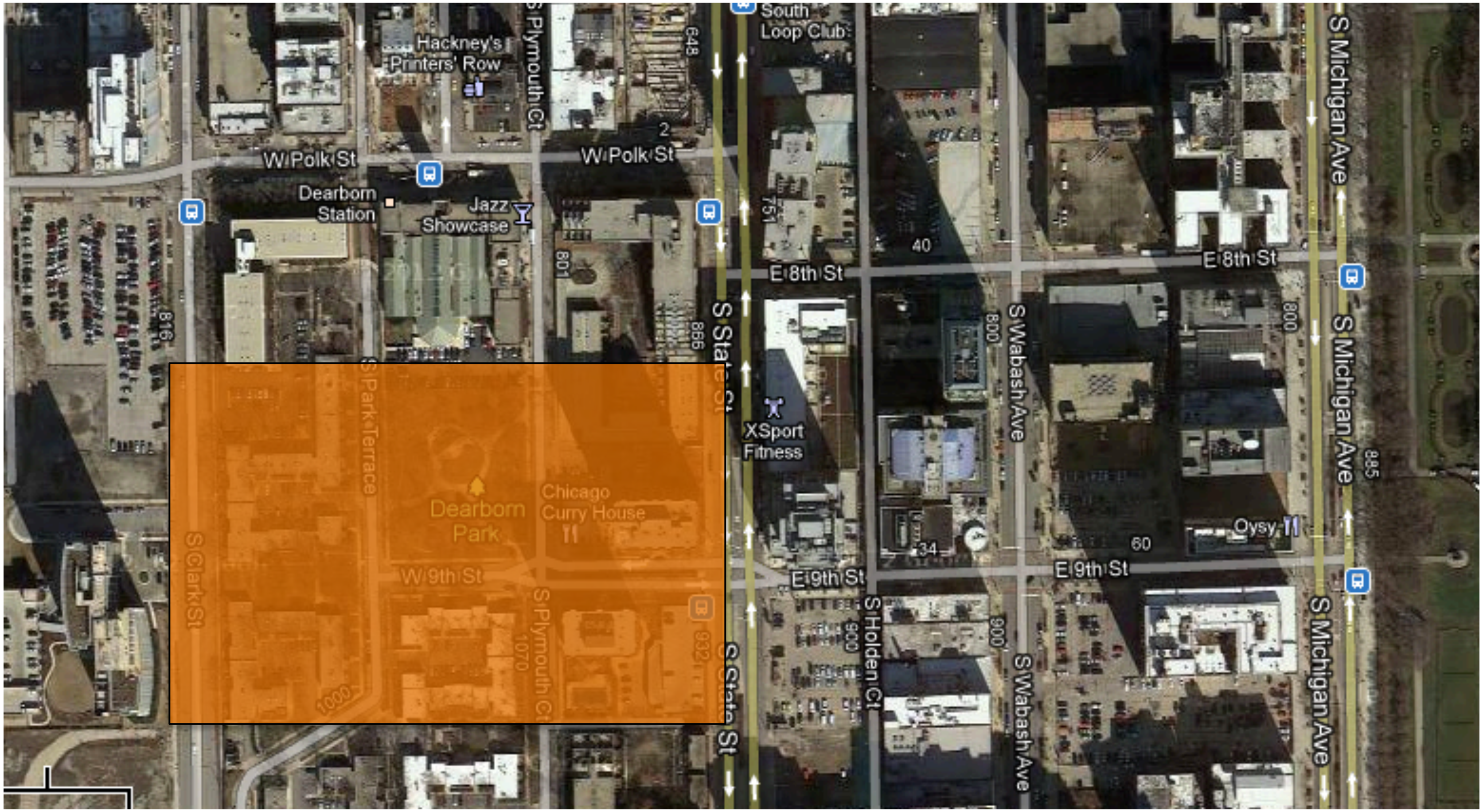
4003-570/4005-761
Computer Graphics 1

Clipping

Clip Window

- Defines part of the world (infinite canvas) that will appear in the image



Clipping

- Clipping is removing non-visible portions of the world

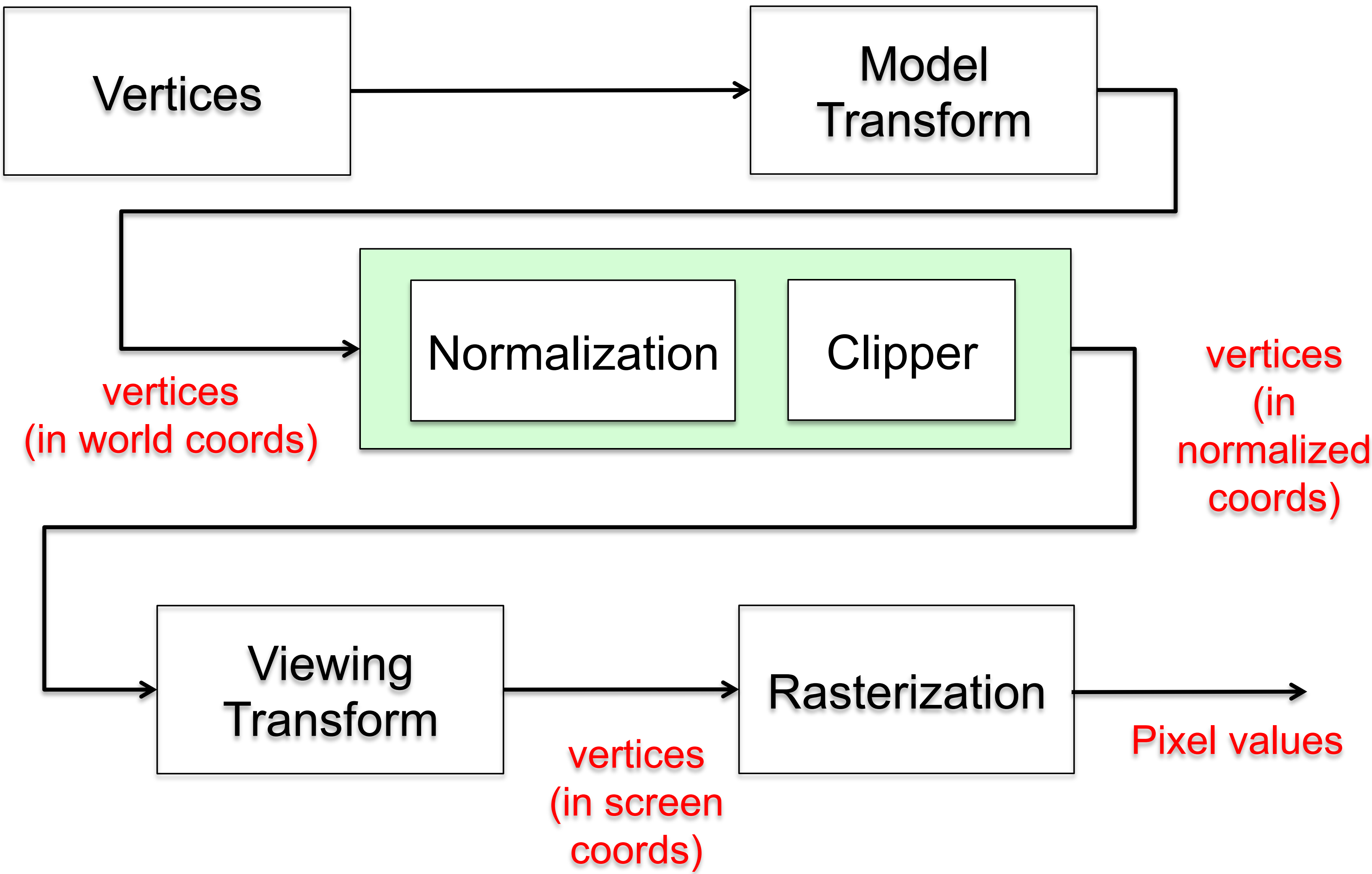
- Many different clipping algorithms
  - Point, line, polygon, curve, text, etc.

- Will study clipping against a general window
  - Then clipping about our normalized window.

- We'll study two in some detail:
  - Line clipping: Cohen-Sutherland Clipping Algorithm
  - Polygon clipping: Sutherland-Hodgman Polygon Clipper

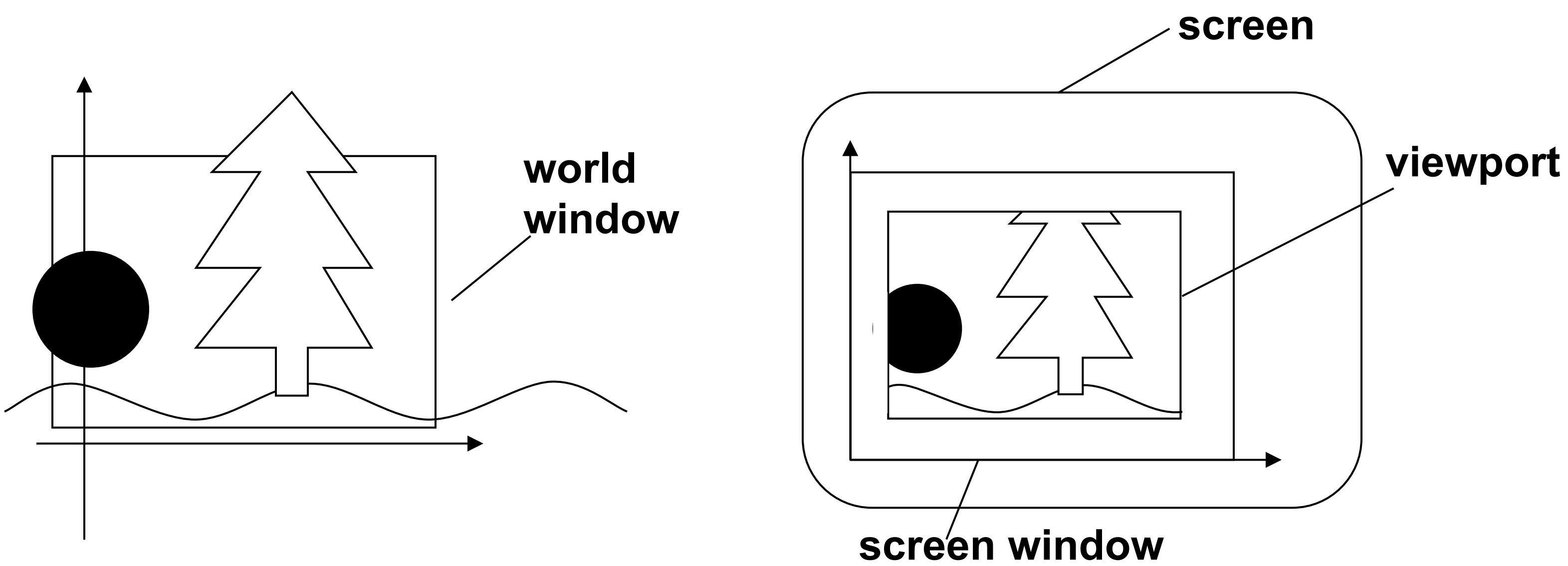
Note: clipping is done in continuous floating point space

2D Pipeline - Geometry



World vs. View

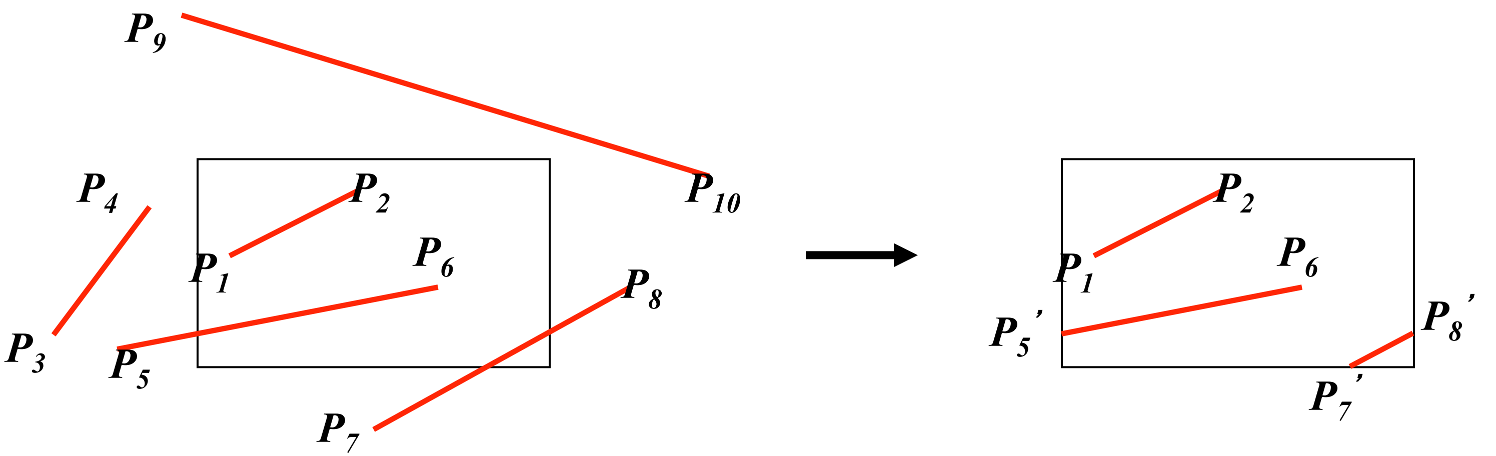
- Here is the relationship:



- Objects (and portions of objects) outside the world window are *clipped*
    - Only the portion of the world which is inside the window is visible

Line Clipping

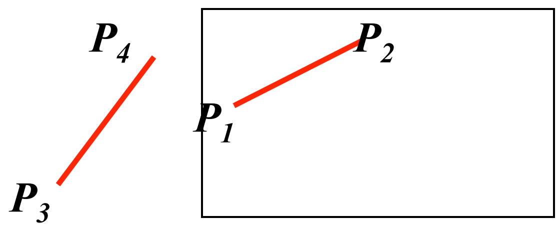
- Basic concept: determine which parts of line segments are inside the clipping region
  - Generally a rectangle, but can be any shape



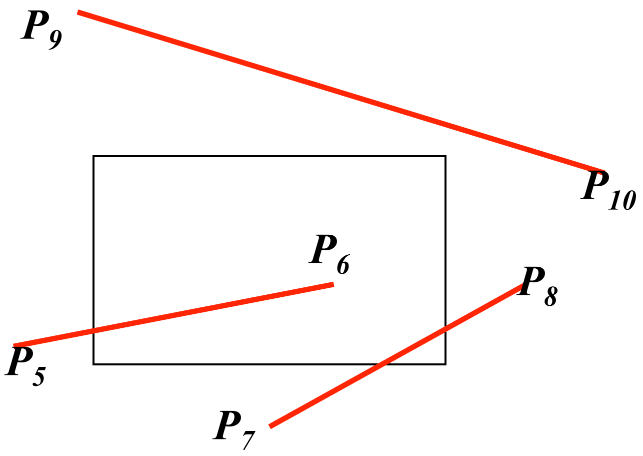
- Cohen-Sutherland Line Clipper is one of the earliest "fast" line clippers
  - Attempts to reduce computation time by doing some work before computing edge intersections

Trivial Cases

- Some combinations of endpoints are easy to handle:

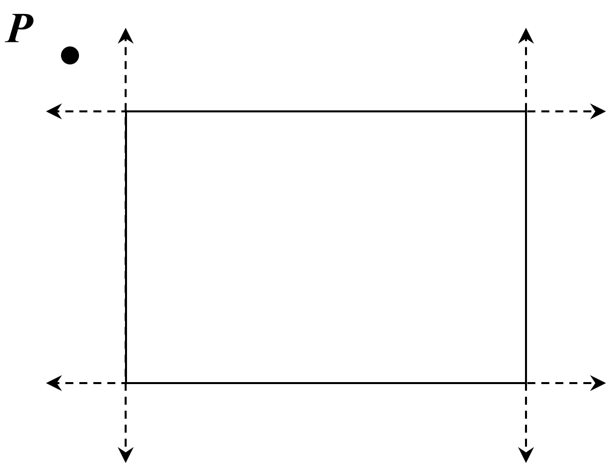


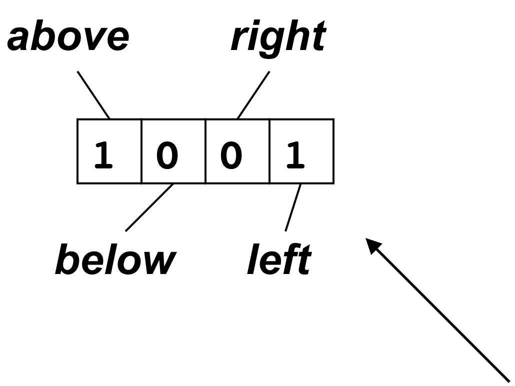
- Others, less so:



Determining Position

- Need a simple way to test for position of line segment relative to the clipping region
- Idea: compute *outcodes*
  - Called *region codes* in our text
  - Every line endpoint is assigned a region code
- Basic concept:
  - Associate one bit with each edge of the clipping region
  - Bit value computed as 1 if "outside" relative to that edge, 0 if "inside"

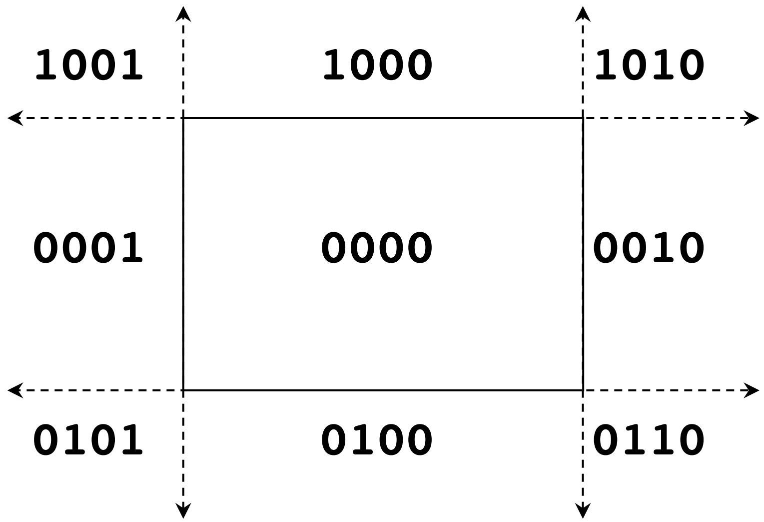




Can be specified in any order

Outcodes

- Each point will have one of nine code combinations:



- Trivial accept: OR of outcodes = 0000
  - Both vertices inside clipping region
- Trivial reject: AND of outcodes != 0000
  - Both vertices outside in same direction
  - Thus, entire segment is outside

Pseudocode

```
int clipSegment( Point2 &p1, Point2 &p2, RealRect w ) {
    do {
        recompute_outcodes
        if( trivial_accept ) return 1;
        if( trivial_reject ) return 0;
        if( p1_is_outside ) {
            if( p1_is_left ) clip_against_left_edge
            else if( p1_is_right ) clip_against_right_edge
            else if( p1_is_below ) clip_against_bottom_edge
            else if( p1_is_above ) clip_against_top_edge
        } else {  // p2 must be outside
            if( p2_is_left ) clip_against_left_edge
            else if( p2_is_right ) clip_against_right_edge
            else if( p2_is_below ) clip_against_bottom_edge
            else if( p2_is_above ) clip_against_top_edge
        }
    } while( 1 );
}
```

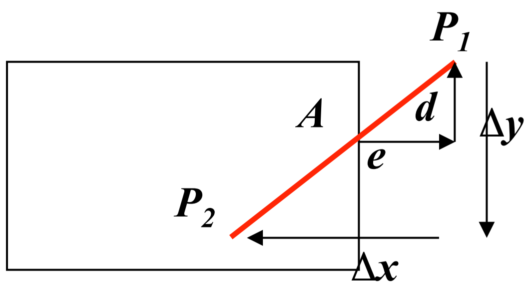
- Algorithm guaranteed to terminate
- Each iteration clips against one boundary
- After at most four iterations, remaining segment is either entirely inside or entirely outside

Cohen Sutherland Line Clipping Applet

http://www.cs.princeton.edu/~min/cs426/jar/clip.html

Clipping Method 1: Similar Triangles

- We clip by replacing point P_1 with point A



- We know: $x_A = xw_{max}, \qquad y_A = y_1 - d$

- Also, this ratio must hold: $d / \Delta y = e / \Delta x$

$$d = e \times \Delta y / \Delta x$$

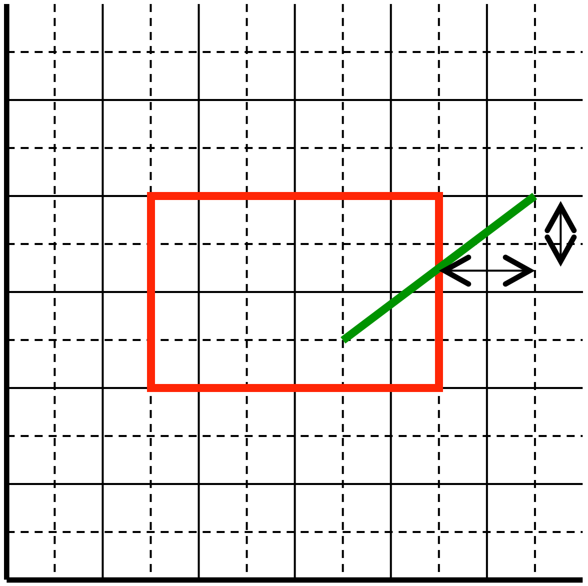
- Clearly, $e = x_1 - xw_{max}$

- So, $d = (x_2 - xw_{max}) \times \Delta y / \Delta x$

Note that for normalized window x value for right side = 1

Example – Similar Triangles

- Clipping rectangle: *(3,4)* to *(9,8)*
- Line: *(7,5)* to *(11,8)*



$$\Delta x = 11 - 7 = 4$$

$$\Delta y = 8 - 5 = 3$$

$$e = 11 - 9 = 2$$

$$d = e \times \Delta y / \Delta x = 2 \times 3 / 4 = 6 / 4 = 1.5$$

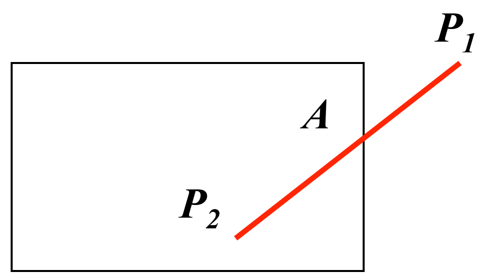
$$x_A = 9$$

$$y_A = y_1 - d = 8 - 1.5 = 6.5$$

$$\therefore A = (9, 6.5)$$

Clipping Method 2: Slope/Intercept Form

- Can calculate intercept from $y = mx + B$ form:



$$m = \Delta y / \Delta x = (y_1 - y_2)/(x_1 - x_2)$$

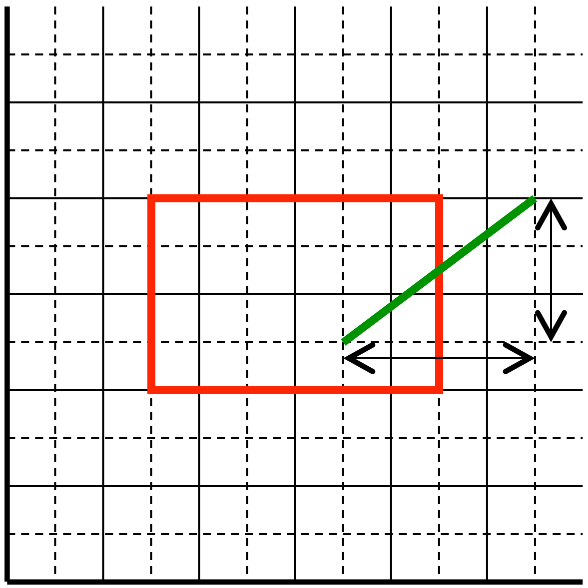
$$B = y_1 - m \times x_1$$

$$x_A = xw_{\max}$$

$$y_A = m \times xw_{\max} + B$$

Example – Slope/Intercept Form

- Clipping rectangle: *(3,4)* to *(9,8)*
- Line: *(7,5)* to *(11,8)*

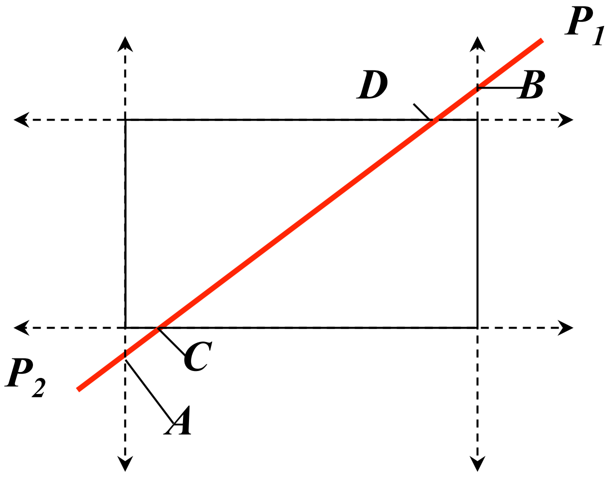


$$m = (8-5)/(11-7) = 3/4$$
$$= .75$$
$$B = y_1 - m \times x1 = 5 - .75 \times 7 = 5 - .25$$
$$= -.25$$
$$x_A = 9$$
$$y_A = .75 \times 9 - .25 = 6.75 - .25$$
$$= 6.5$$

$$\therefore A = (9,6.5)$$

Notes on Cohen-Sutherland Algorithm

- Order of clipping is (generally) irrelevant
  - We clipped against left, right, bottom, and top, in that order
- Can optimize if we know something about vertex positions
  - E.g., if we know that both are above yw_{min}
- Worst case: segment requires four clips

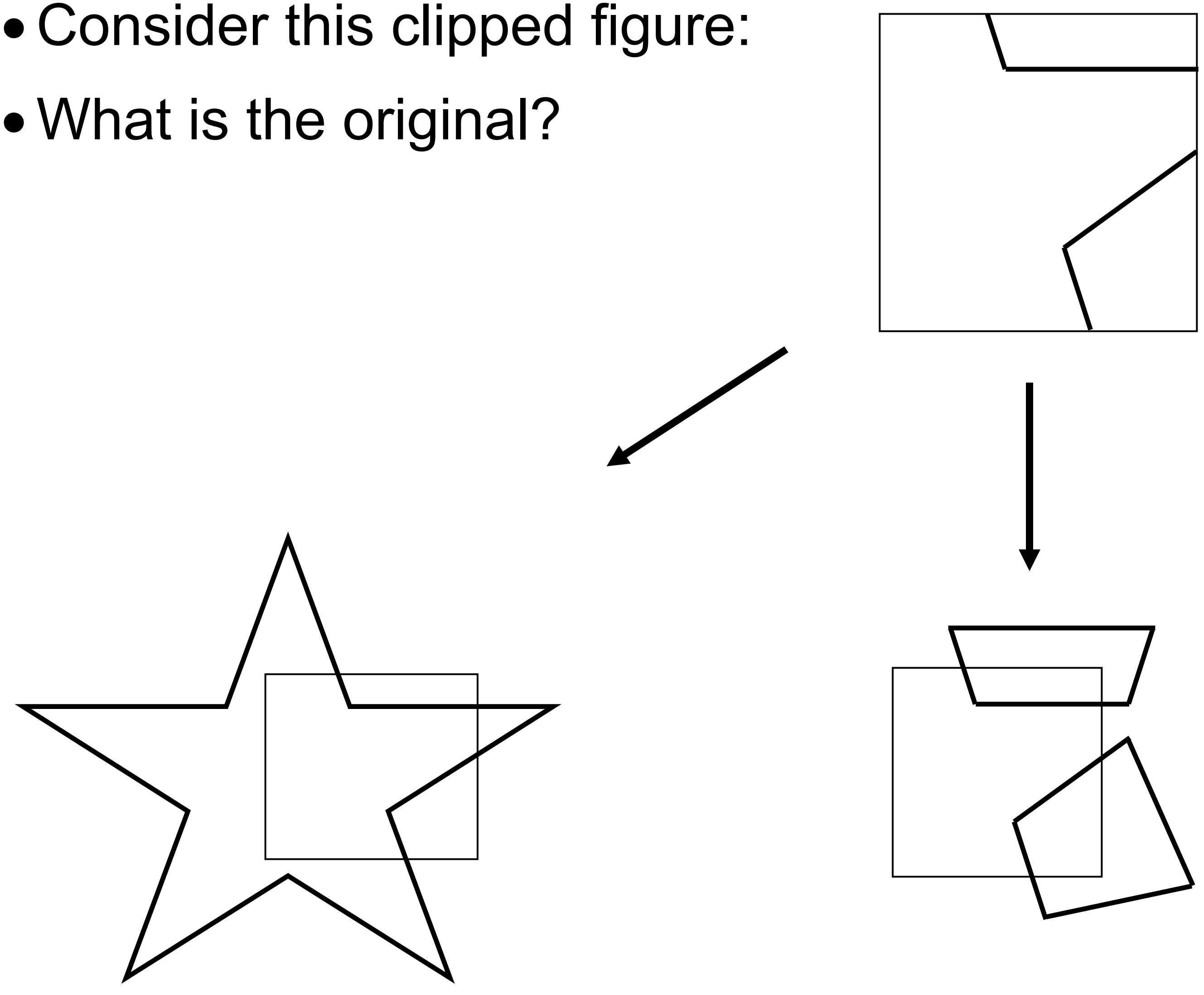


Other Line Clippers

- Even faster clippers exist
    - Cyrus-Beck
    - Liang-Barsky
    - Nicholl-Lee-Nicholl
- Textbook covers Liang-Barsky and NLN
    - Includes sample code for Liang-Barsky

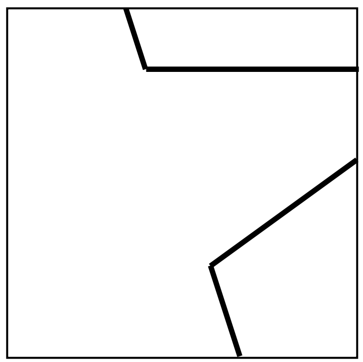
Polygon Clipping

- Consider this clipped figure:
- What is the original?

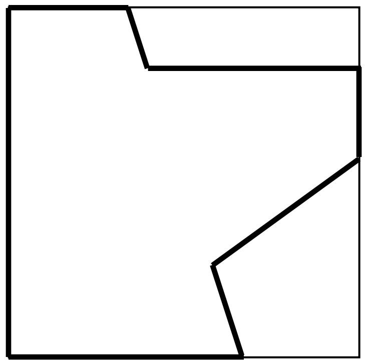


Polygon Clipping

- If we use a line clipper, we don’t know
  - The figure could have come from either source
- Also, it is no longer a polygon
  - It’s now simply four line segments

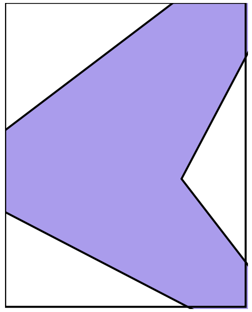


- We want clipped polygons to remain polygons
- This requires a different approach to clipping



Polygon Clipping

- We need to test each edge of the polygon against the clip region, and clip against each edge of the region
- Example: clip against right, bottom, left, and top edges:

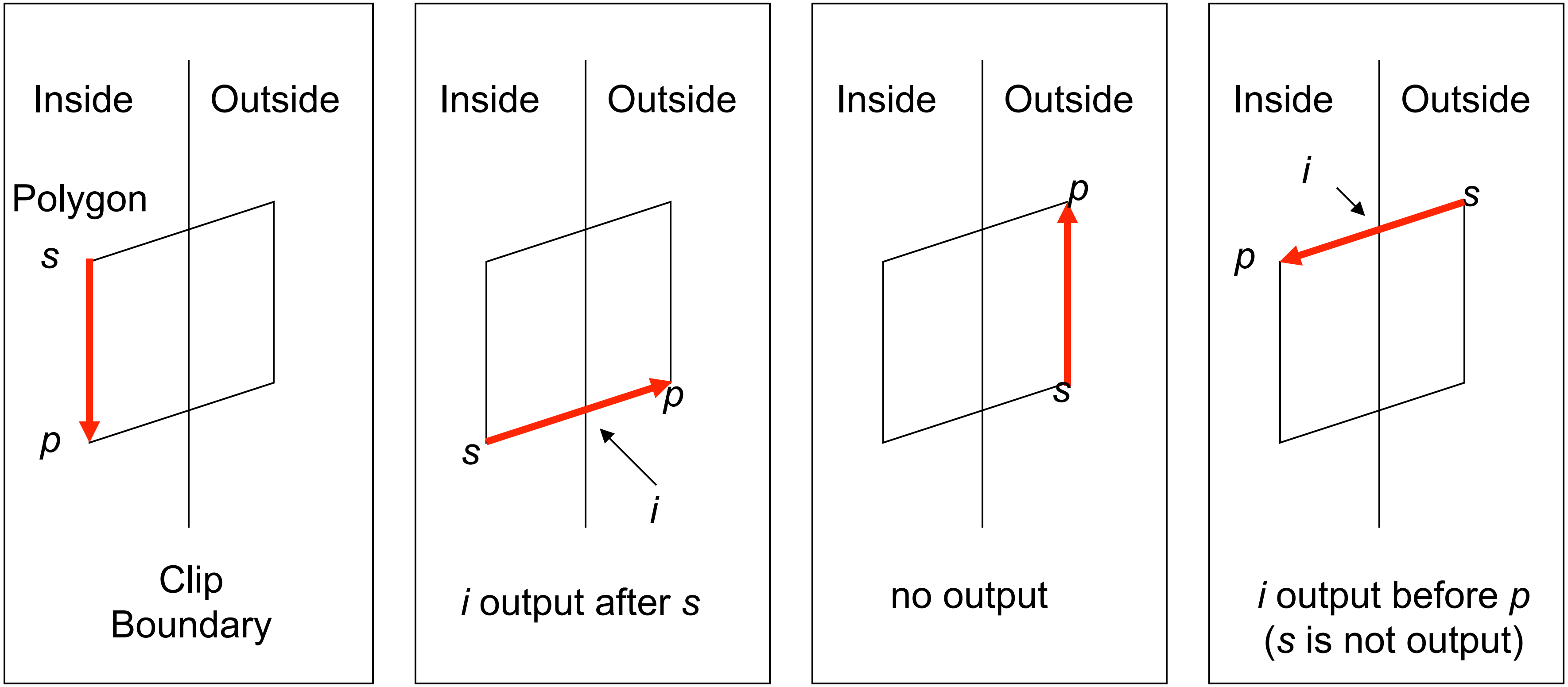


Sutherland-Hodgman Polygon Clipper

- This is a general-purpose polygon clipper
- Can clip against any convex clipping polygon
  - We'll look at a rectangle, but it could be a pentagon, etc.
- Works against convex and concave polygons
- Input: a series of vertices $v1, v2, ..., vn$
- Output: a series of vertices defining the clipped polygon
  - Note: produces a single list of result vertices, so clipped result is a single polygon
- Method: apply a general "clip against edge" routine to each side of the clip region in sequence

Cases

- Four cases to deal with:



Implementation

- Makes use of support routines:
  - inside(*point*,*boundary*) – is *point* inside *boundary*?
  - output(*point*,*length*,*vector*) – put *point* into *vector*, update *length*
  - intersect(*spoint*,*epoint*,*boundary*,*newpoint*) – compute intersection point, put point into *newpoint* parameter
- If we do edges CCW (right, top, left, bottom), "outside" is always to the "right"
- Start with last vertex as the initial "predecessor"
- Apply two rules to each vertex:
  - Does the line from this vertex to its predecessor intersect this edge?
    - Yes → add intersection point to output list
  - Is the vertex itself inside the edge?
    - Yes → add vertex to output list

Implementation

```
SHPC( inVertices, outVertices, inLength, outLength, clipboundary ) {

  outLength = 0;   p = inVertices[ inLength - 1 ];

  for ( j=0; j<inLength; j++) {
    v=inVertices[j];

    if( inside( v, clipboundary ) ) {          // Cases 1 & 4
      if ( inside( p, clipboundary )) {        // Case 1
        output( v );
      } else {                                              // Case 4
        intersect( p, v, clipboundary, i);
        output(i);
        output(v);
      }
    } else {                                                         // Cases 2 & 3
      if( inside ( p, clipboundary ) ) {       // Case 2
        intersect( p, v, clipboundary, i);
        output(i);
      }                                                     // Case 3 has no output
    }

    p = v;

  } // for
}
```

Implementation

- This algorithm does *one side* – to clip entire figure:

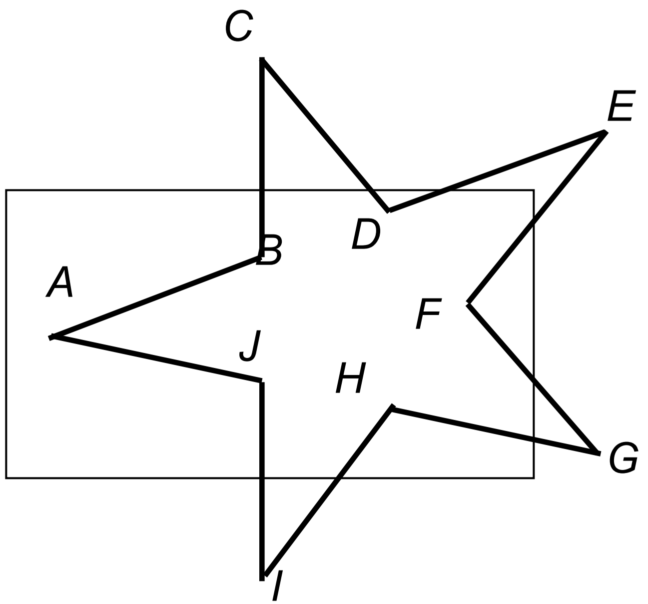
```
SHPC( in, out1, inlen, outlen1, edge1 );
SHPC( out1, out2, outlen1, outlen2, edge2 );
SHPC( out2, out3, outlen2, outlen3, edge3 );
SHPC( out3, out4, outlen3, outlen4, edge4 );
```

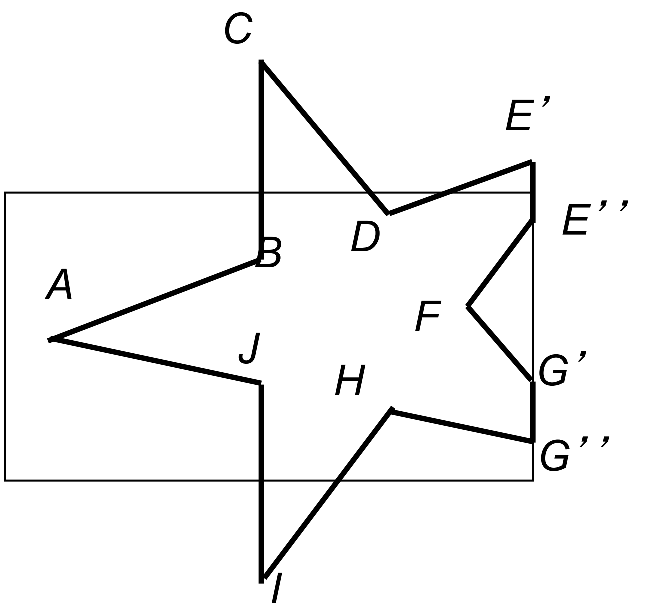
- Note that output from one pass is input to the next pass

http://www.cl.cam.ac.uk/teaching/0607/CompGraph/tutor.html

Method

- Start with a vertex list:
  - *A, B, C, D, E, F, G, H, I, J*
- Clip against the rectangle in this order:
  - Right, top, left, bottom
  - "Outside" is always to the right of the clipping edge
- At each step, replace "outside" vertices with one or more "edge" vertices



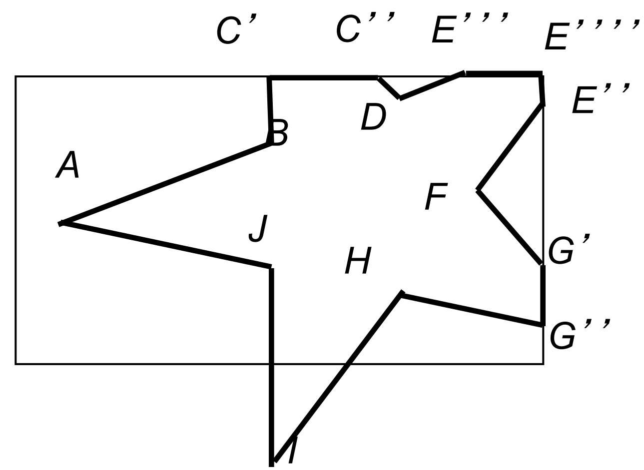


- Example: clip against right edge
- Replace *E* with *E'*, *E''*
- Replace *G* with *G'*, *G''*
- Output vertex list:

A, B, C, D, E', E'', F, G', G'', H, I, J

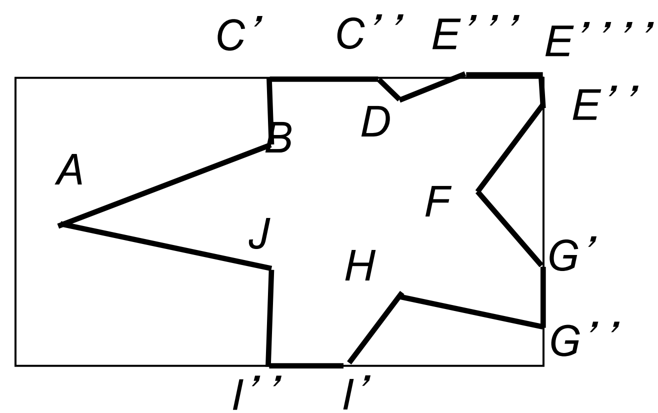
Method

- Next, clip against top
  - Replace C with C', C''
  - Replace E' with E''', E''''
  - Output list:
    - $A, B, C', C'', D, E''', E'''', E'', F, G', G'', H, I, J$



- Clip against left – no change
- Clip against bottom
- Replace I with I', I''
- Final output list:

$A, B, C', C'', D, E''', E'''', E'', F, G', G'', H, I', I'', J$

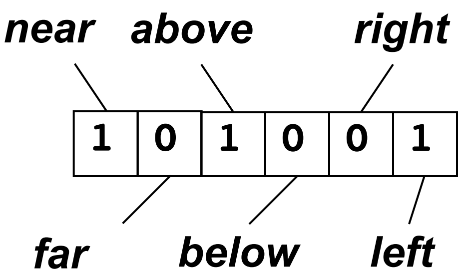


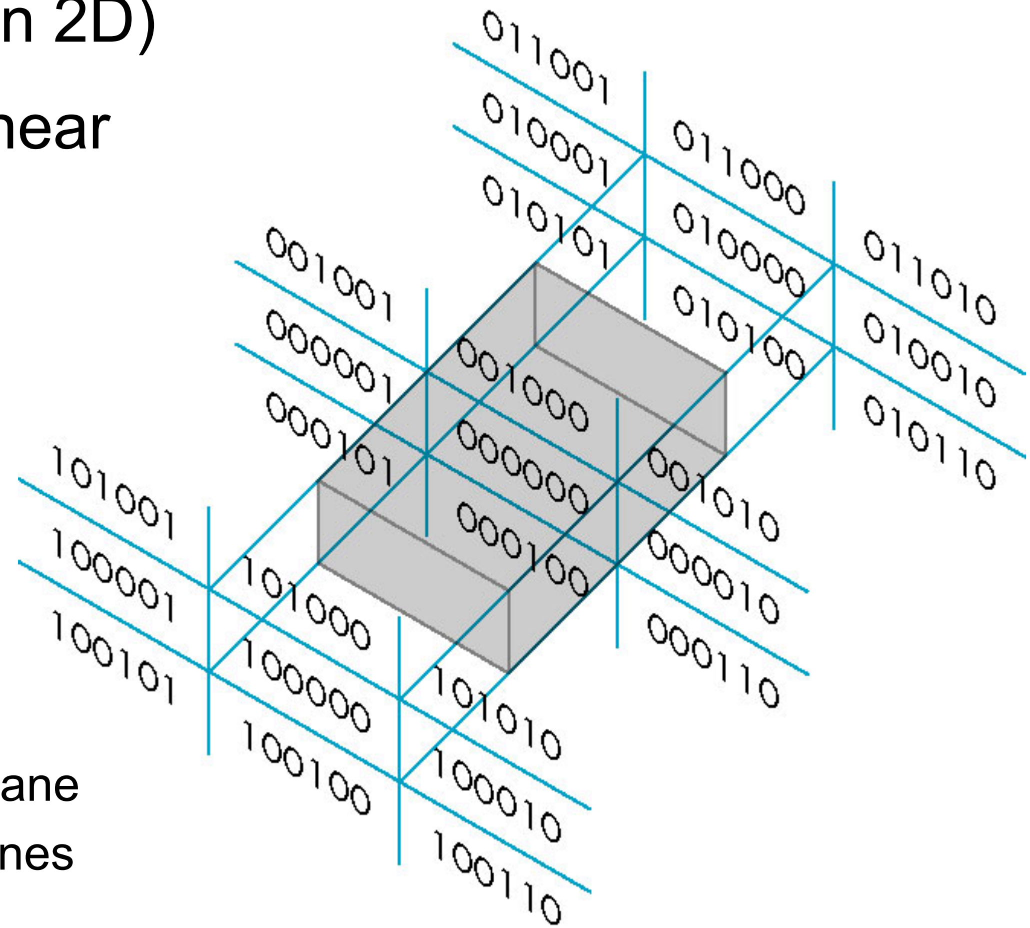
Clipping in 3D

- Similar to clipping in 2D, with the addition of depth
- Clipping in x and y is as before
- Clipping in z done with near and far clipping planes

Point and Line Clipping

- For point and line clipping, can use outcodes (as in 2D)
- Add two more bits for near and far clipping planes





- Three sets of codes:
  - In front of near clipping plane
  - Between near and far planes
  - Behind far plane

Polygon Clipping

- 3D polygons are often called *polyhedrons*
  - Polygon with depth
- Clipping is done to the six faces of the view volume
- May require construction of new surface facets