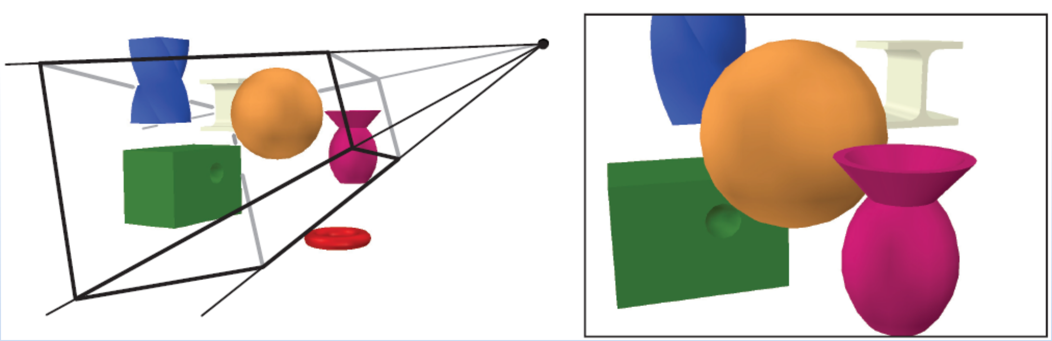
Clipping

The Computer Graphics Pipeline

- Clipping
  - Clip scene against view volume
    - Removes parts of scene not within view volume
  - **Clip space**
    - Coordinate system where clipping is performed



Vertex data

↓

Modelling Transformations

Viewing Transformation

Lighting

Projection

Clipping

Rasterization

Fragment Processing

↓

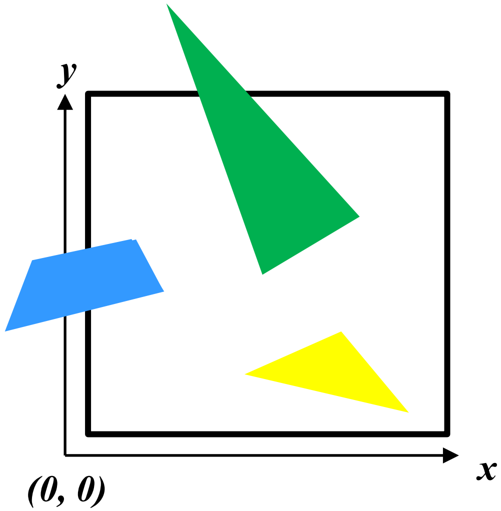
Pixels for display

Overview

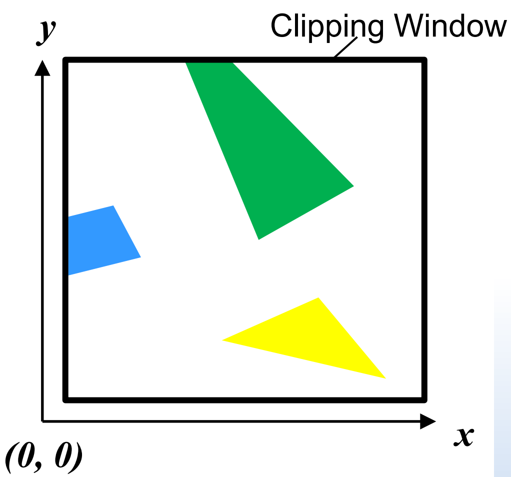
- Clipping
  - ➢ Point
  - ➢ Line clipping algorithm
    - Cohen-Sutherland
  - ➢ Polygon clipping algorithms
    - Hodgman-Sutherland
    - Weiler-Atherton

Clipping

- What is clipping?
  - ➢ The process of determining the portions of a primitive lying within a region called the ‘clipping region’ or ‘clipping window’



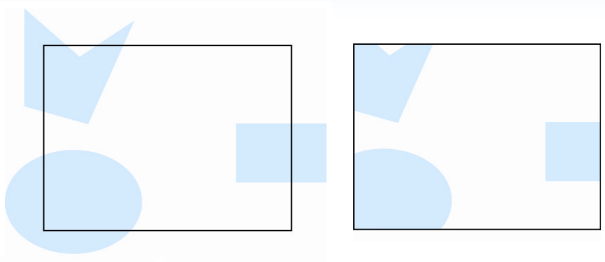
World Space
(Common to all objects, not just confined with the display area)

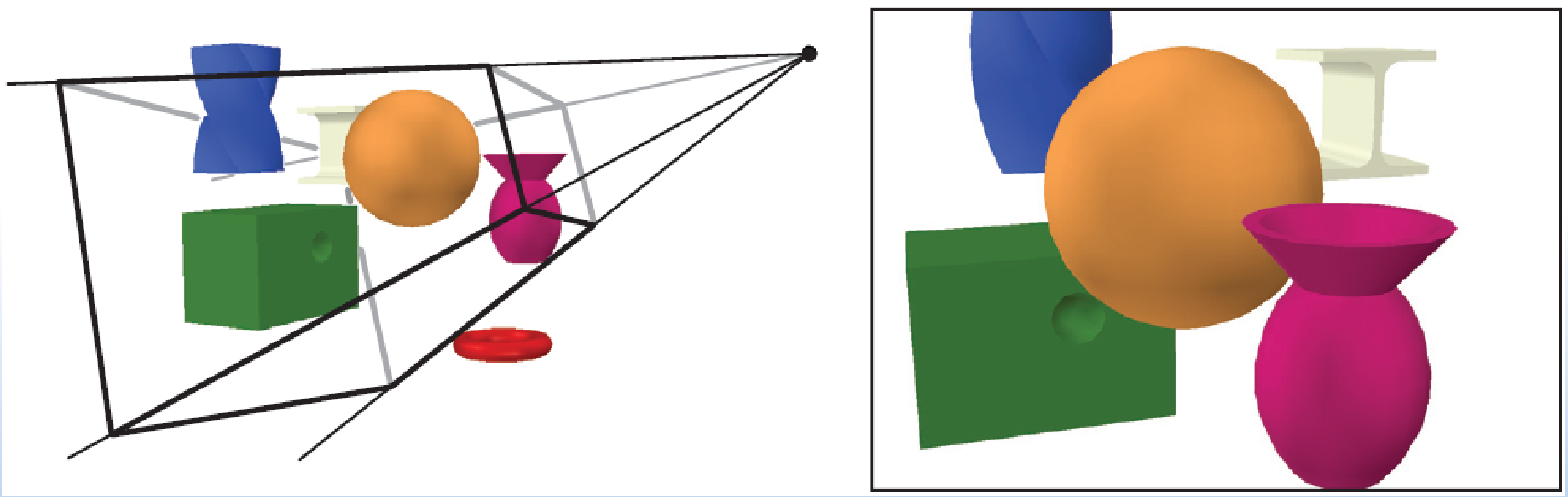


Clip Space

Clipping

- 2D versus 3D clipping
  - ➢ 2D clipping
    - Objects clipped to a clipping window
  - ➢ 3D clipping
    - Objects clipped against a 3D viewing volume



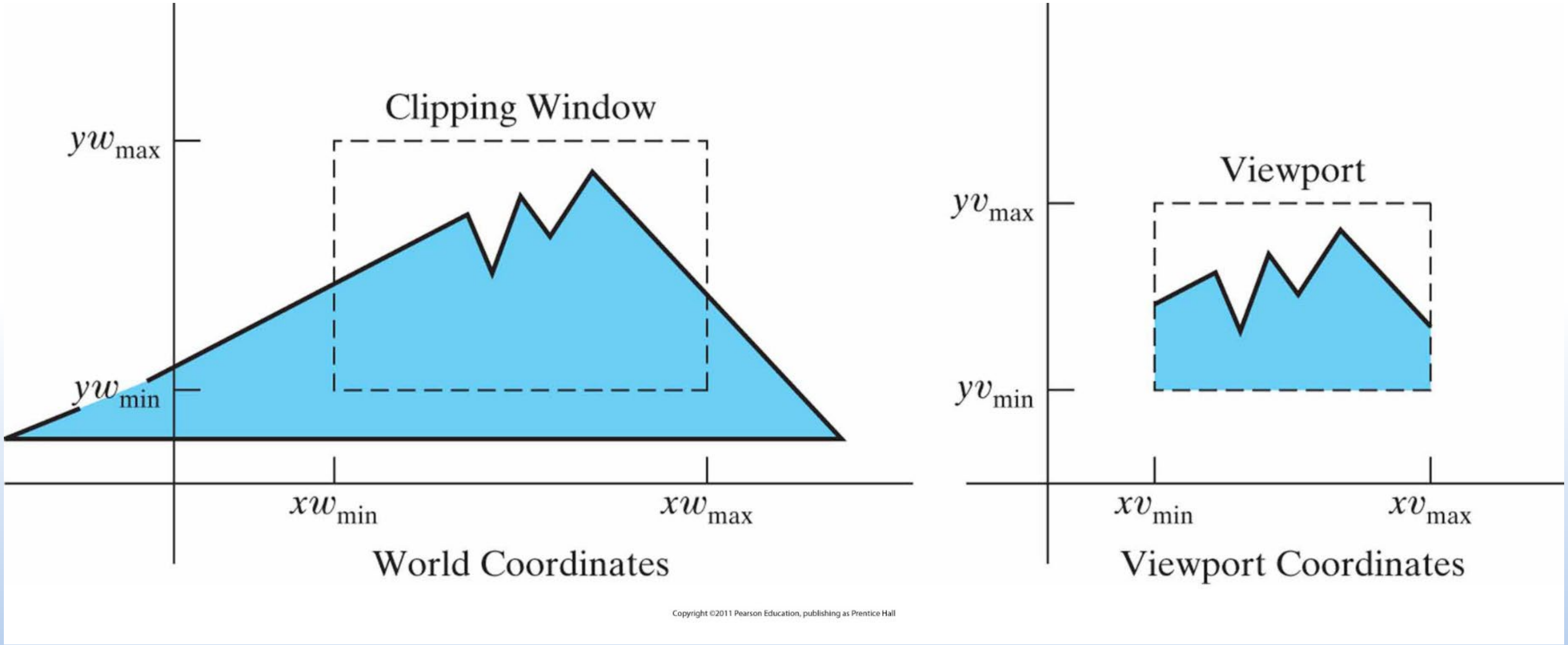


Clipping

- The purpose of clipping?
  - For preventing
    - Activity in one window from affecting pixels in other windows
    - Mathematical overflow and underflow from primitives passing behind the eye point or at great distances (in 3D)
  - Rasterization is computationally expensive
    - More or less linear with number of pixels created
    - After clipping – only rasterize that which is in the viewable region
    - A few clipping operations will save many other operations later on

Clipping

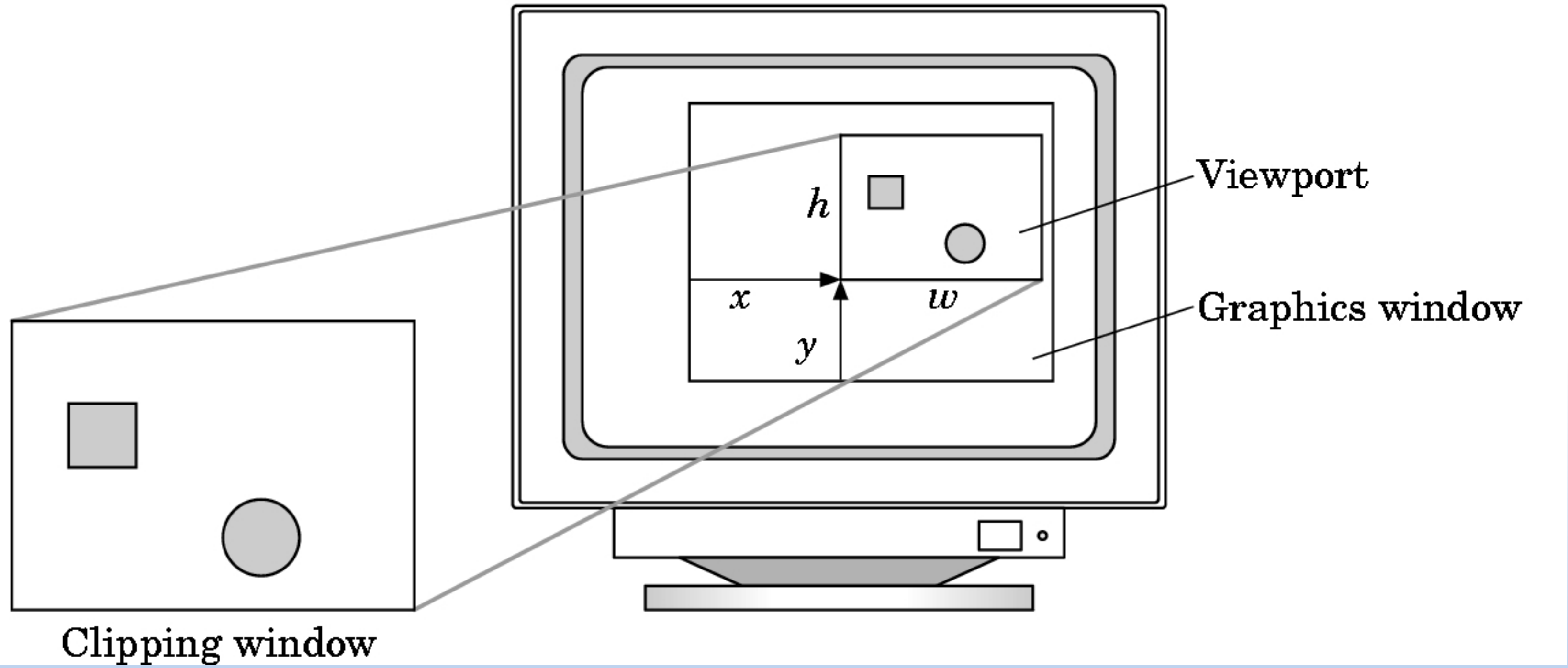
- Clipping
  - Objects inside the clipping window are mapped to the viewport
    - It is the viewport that is positioned within the display window



Clipping

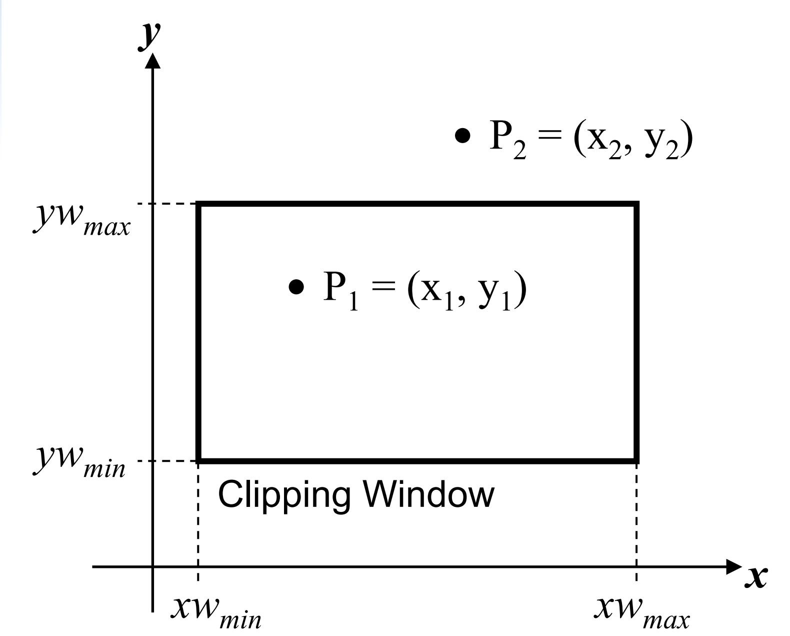
- Clipping
  - ➢ OpenGL viewport

```
glViewport( x, y, w, h );
```



Clipping

- Point clipping
  - Really easy!
  - For point P = (x, y) check

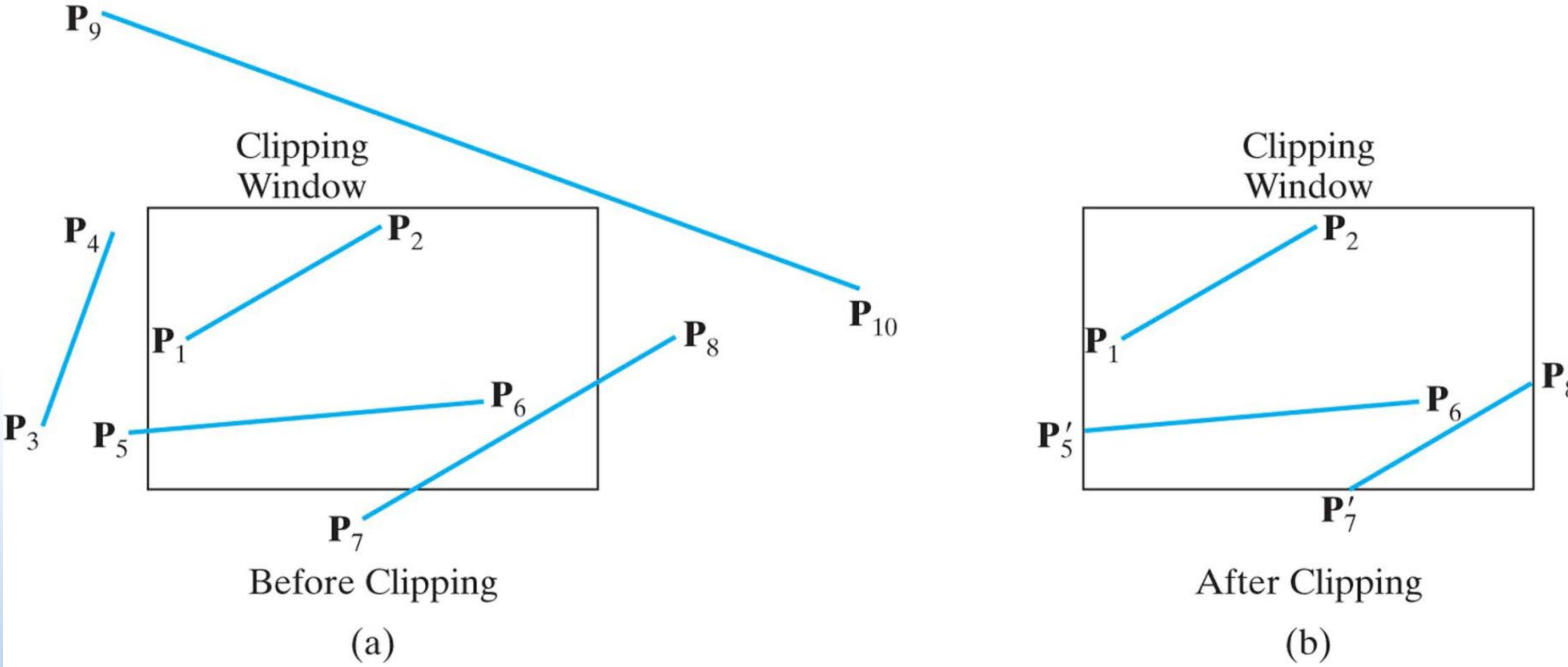


$$xw_{\min} \leq x \leq xw_{\max}$$
and
$$yw_{\min} \leq y \leq yw_{\max}$$

- If any of the 4 inequalities not satisfied, the point is clipped
  - i.e. not saved for display
- Note: The viewport does not necessarily have to be the same as the screen size or display area

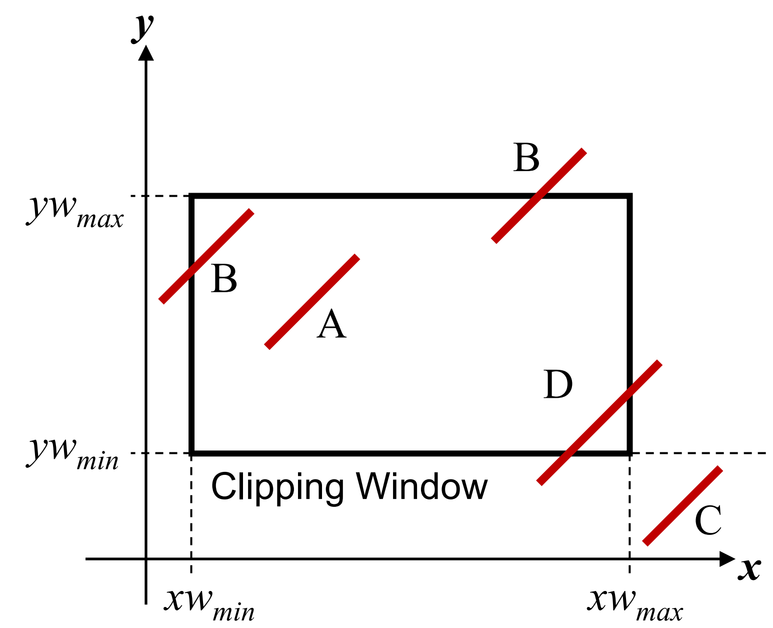
Clipping

- Line clipping
  - ➢ Shortened line passed to rasterization stage
  - ➢ Must be no difference to the lines in the final image



Clipping

- Line clipping (cont.)
  - ➢ Test endpoints of a line using point clipping test
    - Four possible cases
      A. Both points inside
      B. Only one point inside
      C. Both points outside
      D. Both points outside,
         but part of line inside



    - Case A is easy, but what about B, C and D?
    - Brute force approach
      - – Determine intersection of line segment with clipping window edge

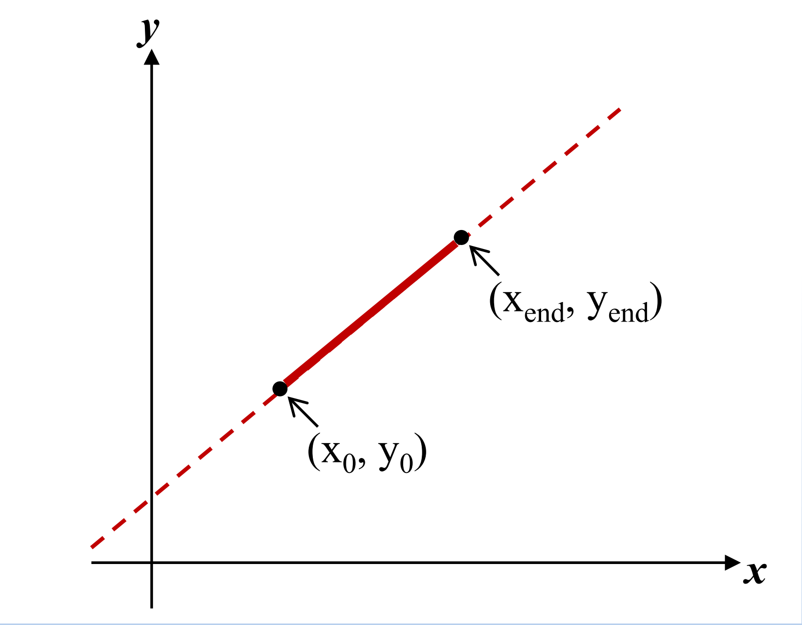
Clipping

- Line clipping (cont.)
  - First determine whether line segment completely inside or outside clipping window
    - Use point clipping test to test endpoints against all 4 clipping window boundaries
      - If endpoints are both inside, then keep
      - If both outside (the same clipping boundary), then remove
    - If test fails
      - Means that the line segment intersects at least one clipping boundary and may or may not cross into the interior of the clipping window

Clipping

- Line clipping (cont.)
  - A line segment can be represented as

$$x = x_0 + u\,(x_{end} - x_0)$$
$$y = y_0 + u\,(y_{end} - y_0) \qquad\qquad 0 \le u \le 1$$



Clipping

- Line clipping (cont.)

$$x = x_0 + u\,(x_{end} - x_0)$$
$$y = y_0 + u\,(y_{end} - y_0) \qquad\qquad 0 \le u \le 1$$

- ➢ For each clipping window edge
  - Test whether line crosses that edge, by assigning that edge's coordinate value to either *x* or *y* and solve for *u*
    - – If u outside range 0 to 1, line does not intersect window border
    - – If u within the range 0 to 1, then part of the line is inside that border
- ➢ Process this until entire line is clipped

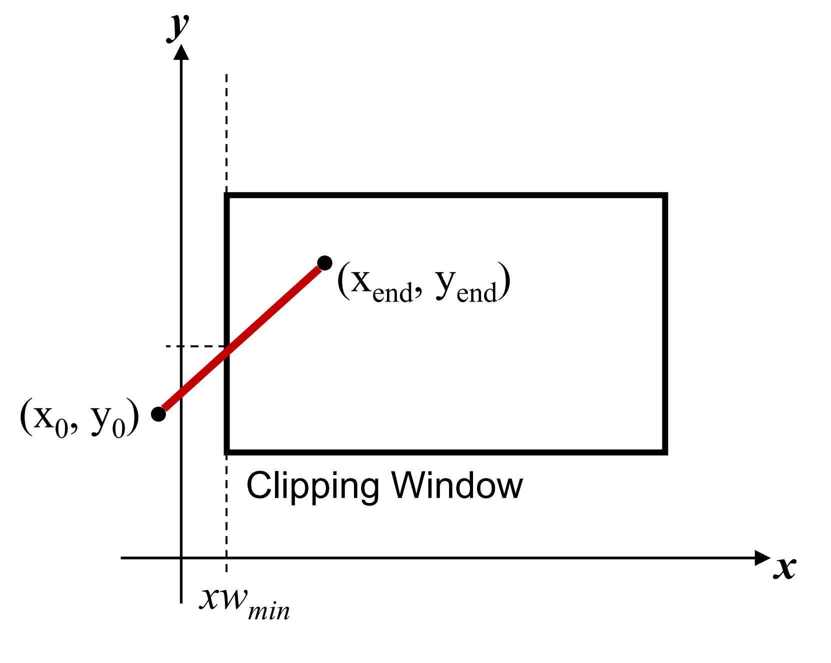
Clipping

- Line clipping (cont.)

$$x = x_0 + u\,(x_{end} - x_0)$$
$$y = y_0 + u\,(y_{end} - y_0) \qquad\qquad 0 \le u \le 1$$

➢ Example

- For left window boundary
  - Substitute xw_{min} into equation and solve for u
  - If u range within [0, 1] then line intersects border
    - » Use u to find y intersection value



Clipping

- Line clipping (cont.)

$$x = x_0 + u\,(x_{end} - x_0)$$
$$y = y_0 + u\,(y_{end} - y_0) \qquad\qquad 0 \le u \le 1$$

  - Example
    - For left window boundary
      - Substitute $xw_{\min}$ into equation and solve for u

$$xw_{\min} = x_0 + u\left(x_{end} - x_0\right)$$

$$u = \frac{xw_{\min} - x_0}{x_{end} - x_0}$$

      - If u range within [0, 1] then line intersects border, use u to find y intersection value

Clipping

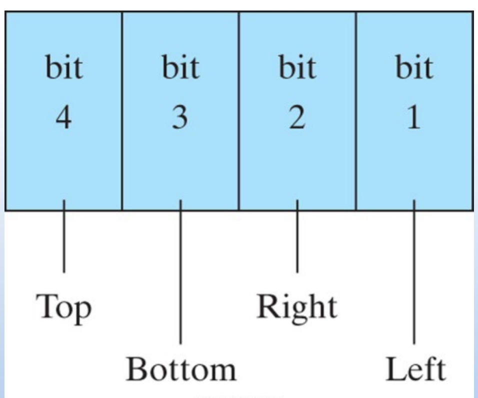
- Line clipping (cont.)
  - Cohen-Sutherland clipping algorithm (cont.)
    - For each endpoint, say (x, y), of a line segment assign a 4-bit 'region code'

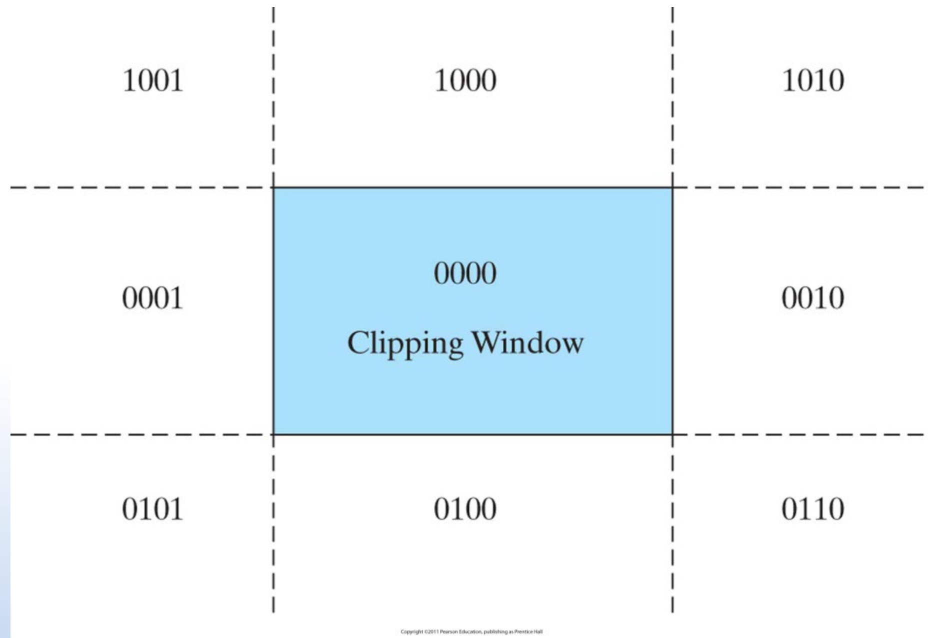
      If ($x < xw_{min}$), bit 1 = 1

      If ($x > xw_{max}$), bit 2 = 1

      If ($y < yw_{min}$), bit 3 = 1

      If ($y > yw_{max}$), bit 4 = 1





Clipping

- Line clipping (cont.)
  - Cohen-Sutherland clipping algorithm (cont.)
    - Then go through these steps
      1. If both endpoints 0000 (i.e. code1 | code2 = 0), ACCEPT
         (| is bitwise OR)
      2. Else If (code1 & code2) != 0, REJECT
         (& is bitwise AND)
      3. Else { Clip line against one viewport boundary
      4. Assign the new endpoint with a 4-bit region code
      5. Goto 1}

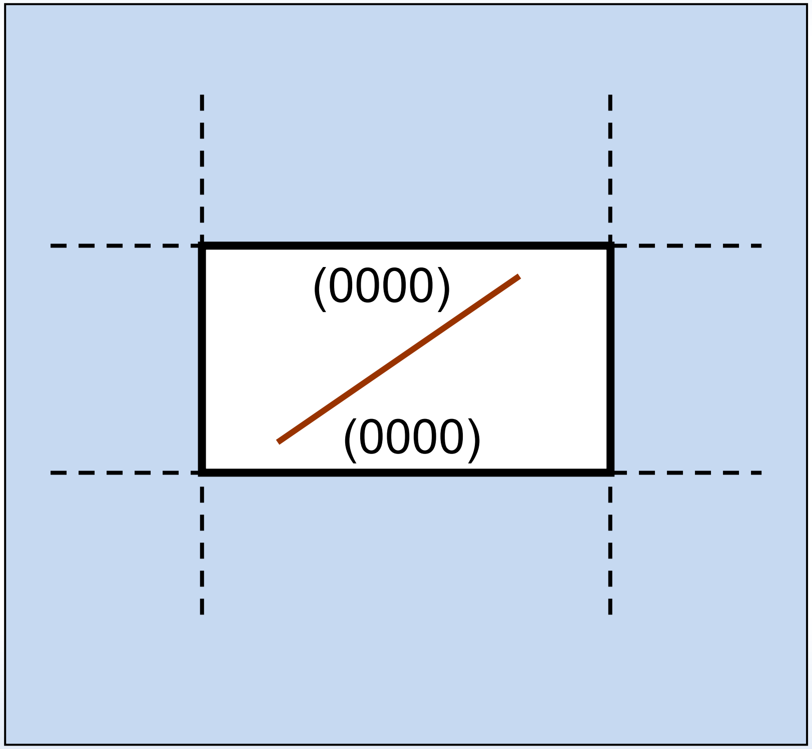
Cohen-Sutherland Clipping Algorithm

1. ***If both endpoints 0000, ACCEPT***
2. Else If (code1 & code2) != 0, REJECT
3. Else, {

   Clip line against one viewport boundary

4. Assign the new endpoint with a 4-bit region code
5. Goto 1

}



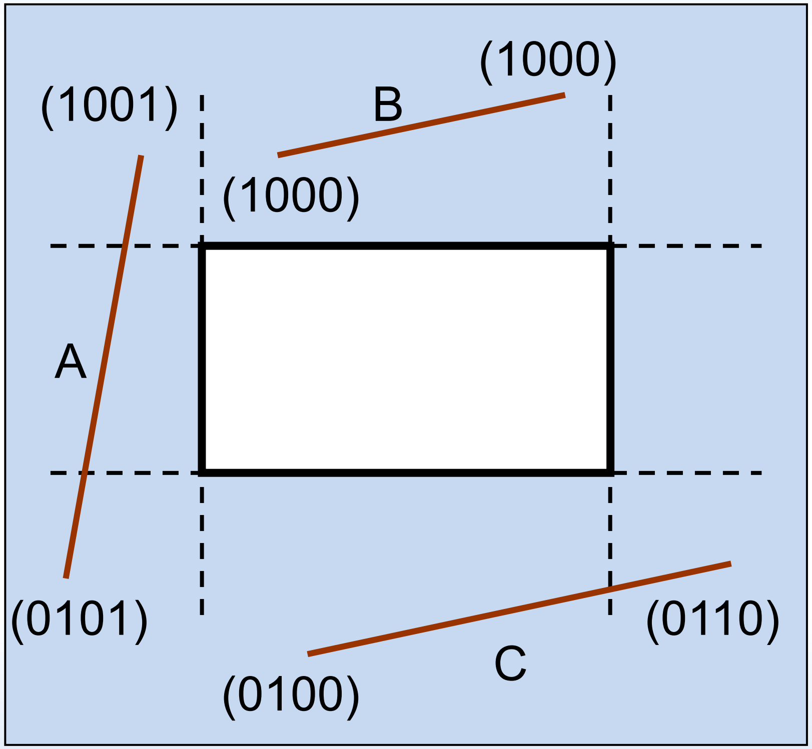
Cohen-Sutherland Clipping Algorithm

1. If both endpoints 0000, ACCEPT
2. ***Else If (code1 & code2) != 0, REJECT***
3. Else, {

    Clip line against one viewport boundary

4. Assign the new endpoint with a 4-bit region code
5. Goto 1

}



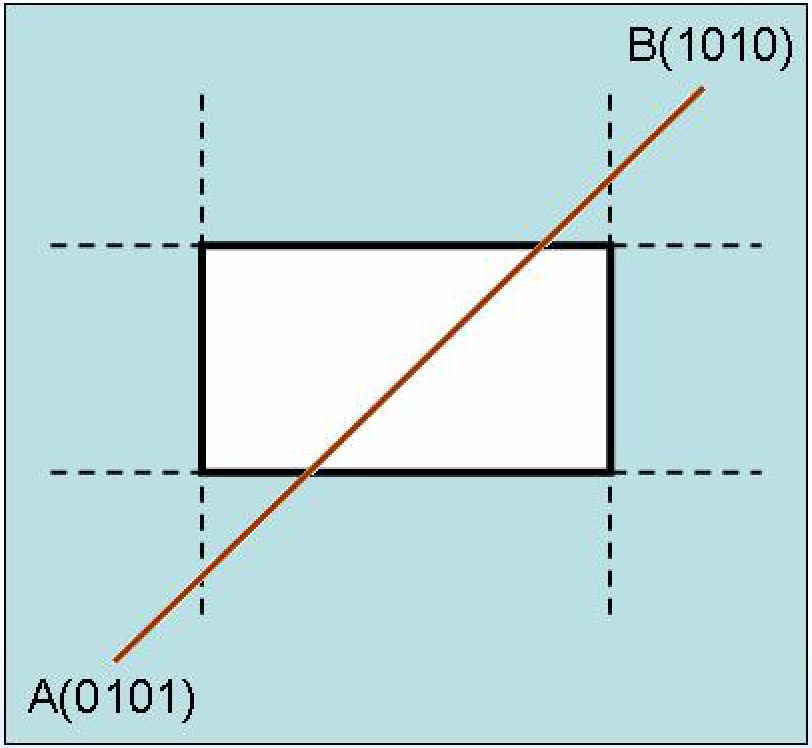
Cohen-Sutherland Clipping Algorithm

1. If both endpoints 0000, ACCEPT
2. Else If (code1 & code2) != 0, REJECT
3. ***Else, {***

    Clip line against one viewport boundary

4. ***Assign the new endpoint with a 4-bit region code***
5. ***Goto 1***

}



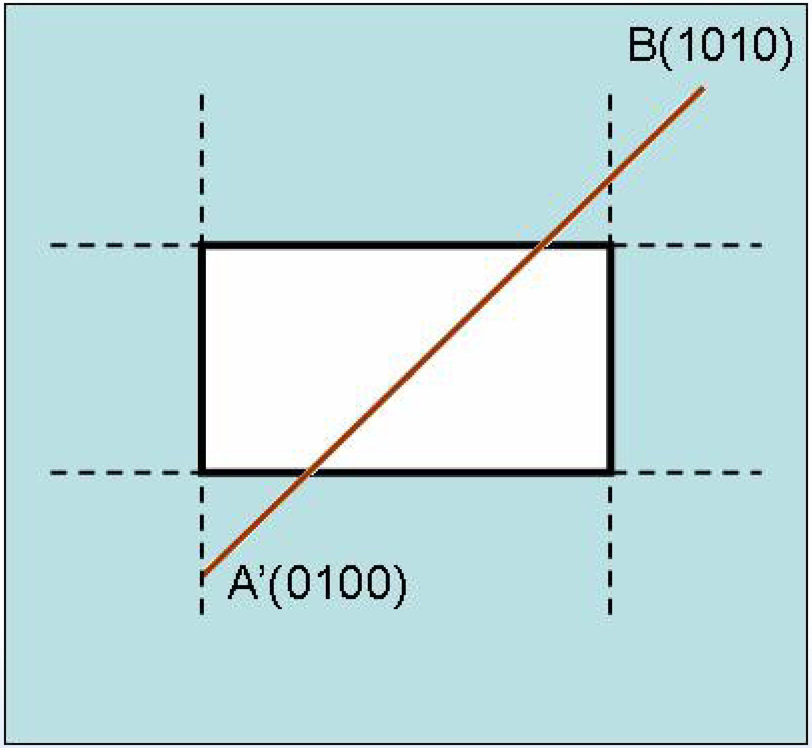
Cohen-Sutherland Clipping Algorithm

1. If both endpoints 0000, ACCEPT
2. Else If (code1 & code2) != 0, REJECT
3. ***Else, {***

   Clip line against one viewport boundary

4. ***Assign the new endpoint with a 4-bit region code***
5. ***Goto 1***

}

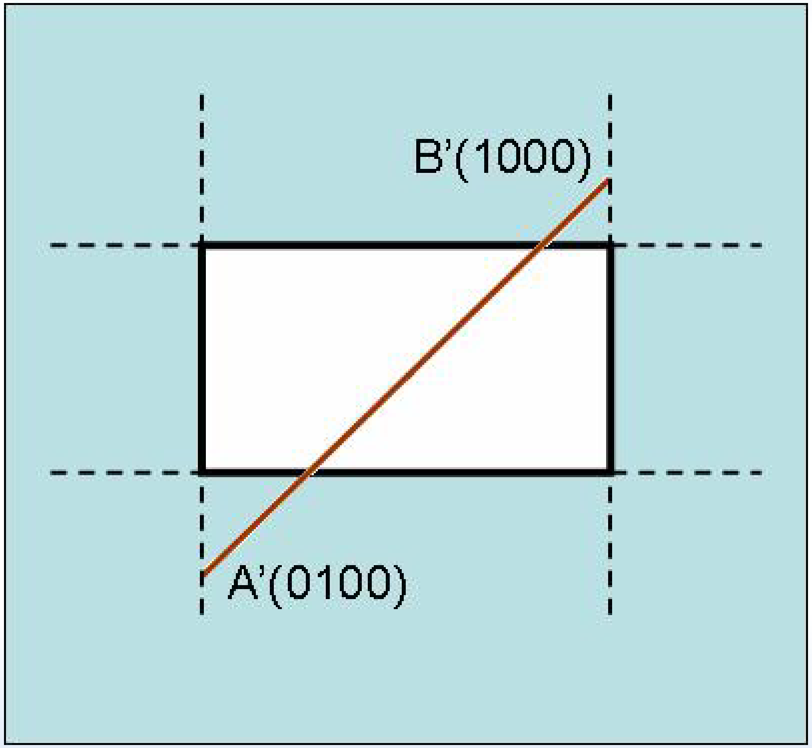


Cohen-Sutherland Clipping Algorithm

1. If both endpoints 0000, ACCEPT
2. Else If (code1 & code2) != 0, REJECT
3. ***Else, {***

   Clip line against one viewport boundary
4. ***Assign the new endpoint with a 4-bit region code***
5. ***Goto 1***

}



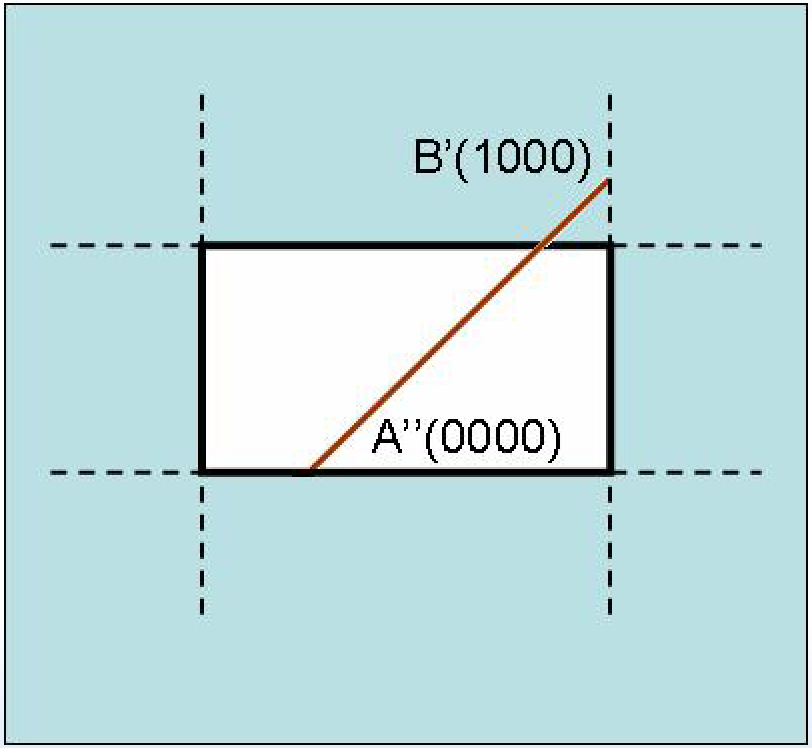
Cohen-Sutherland Clipping Algorithm

1. If both endpoints 0000, ACCEPT
2. Else If (code1 & code2) != 0, REJECT
3. ***Else, {***

   Clip line against one viewport boundary

4. ***Assign the new endpoint with a 4-bit region code***
5. ***Goto 1***

   }



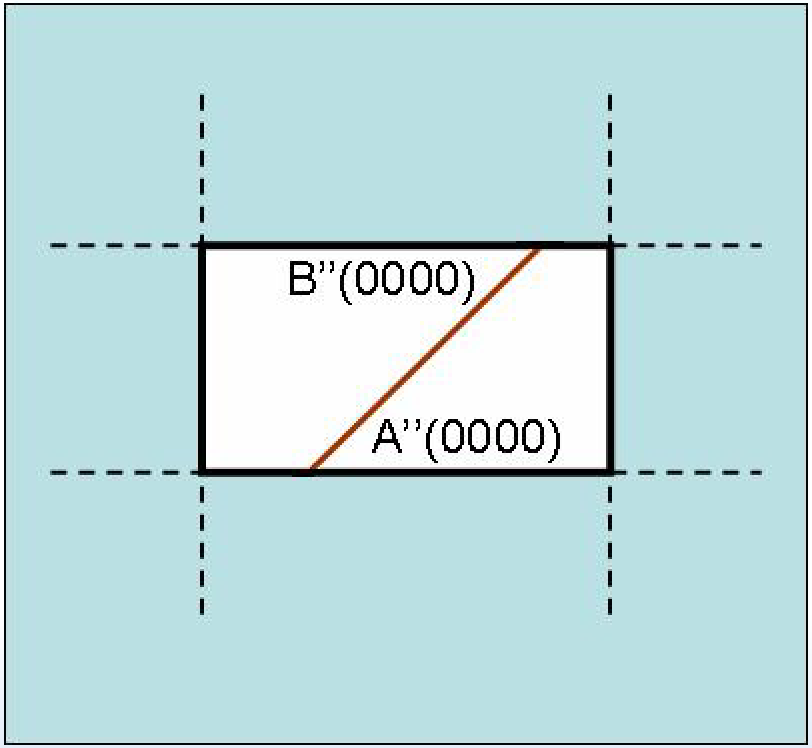
Cohen-Sutherland Clipping Algorithm

1. ***If both endpoints 0000, ACCEPT***
2. Else If (code1 & code2) != 0, REJECT
3. Else, {

   Clip line against one viewport boundary

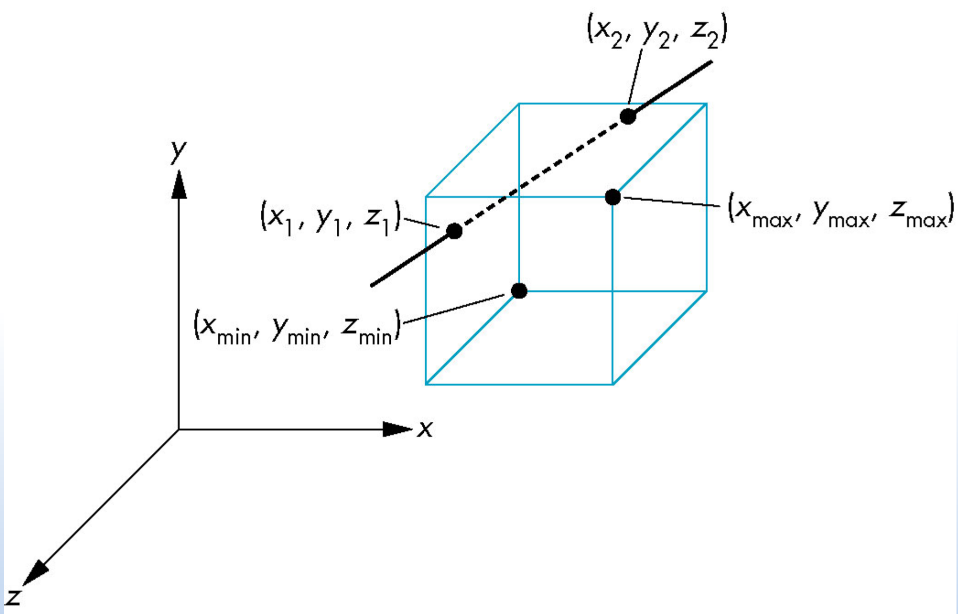
4. Assign the new endpoint with a 4-bit region code
5. Goto 1

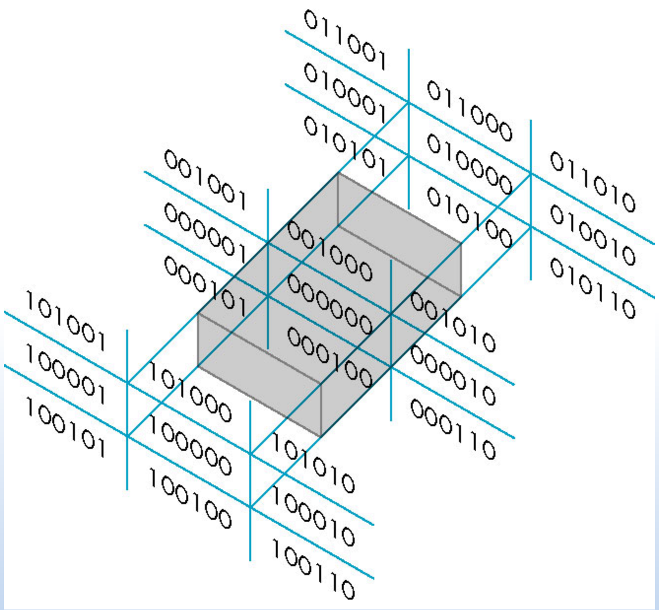
}



Clipping

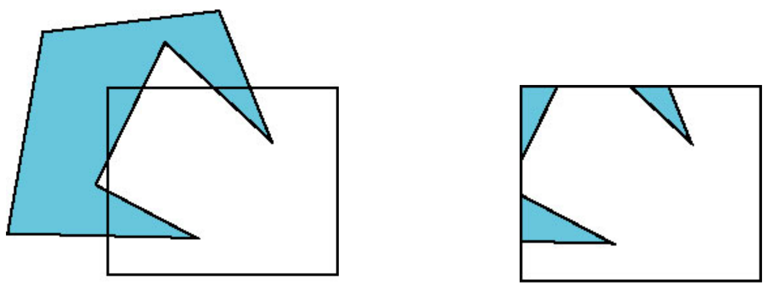
- Line clipping (cont.)
  - Cohen-Sutherland clipping algorithm in 3D
    - Need 6-bit region codes
    - Clip line segment against planes





Clipping

- Polygon fill-area clipping
  - Not as simple as line segment clipping
    - Clipping a line segment yields at most one line segment
    - Clipping a polygon can yield multiple polygons

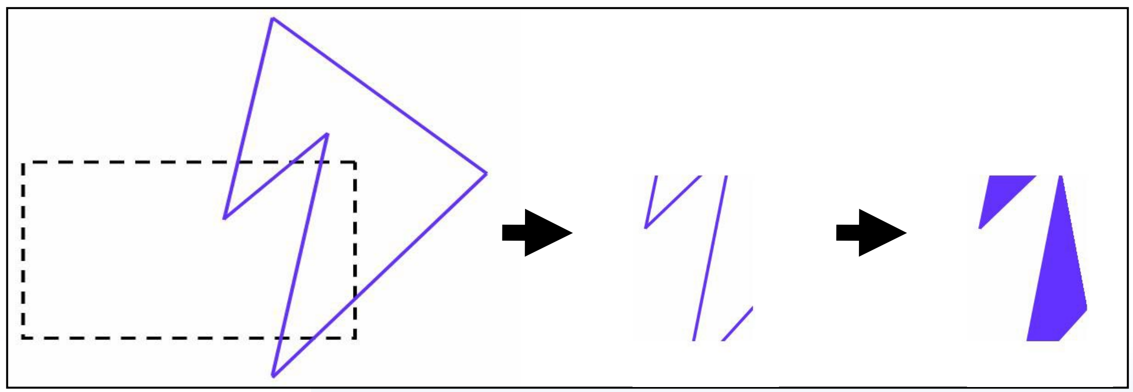


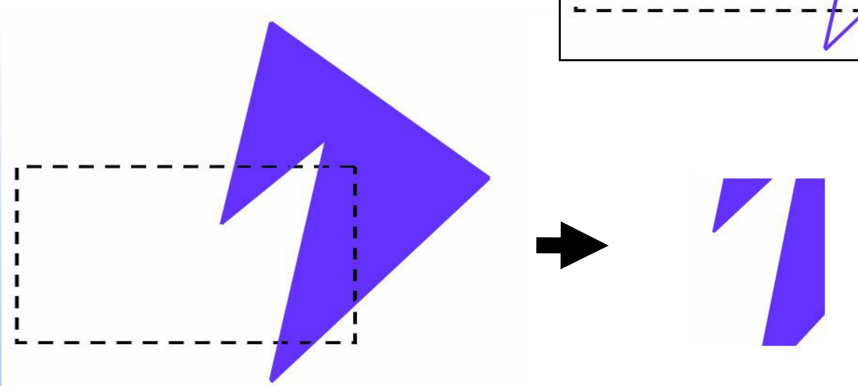
  - However, clipping a convex polygon can yield at most one other polygon

Clipping

- Polygon fill-area clipping (cont.)
  - Not as simple as line segment clipping (cont.)
    - Line clipper by itself often produces a disjoint set of lines with no complete information about how to form closed fill area

Line clipping





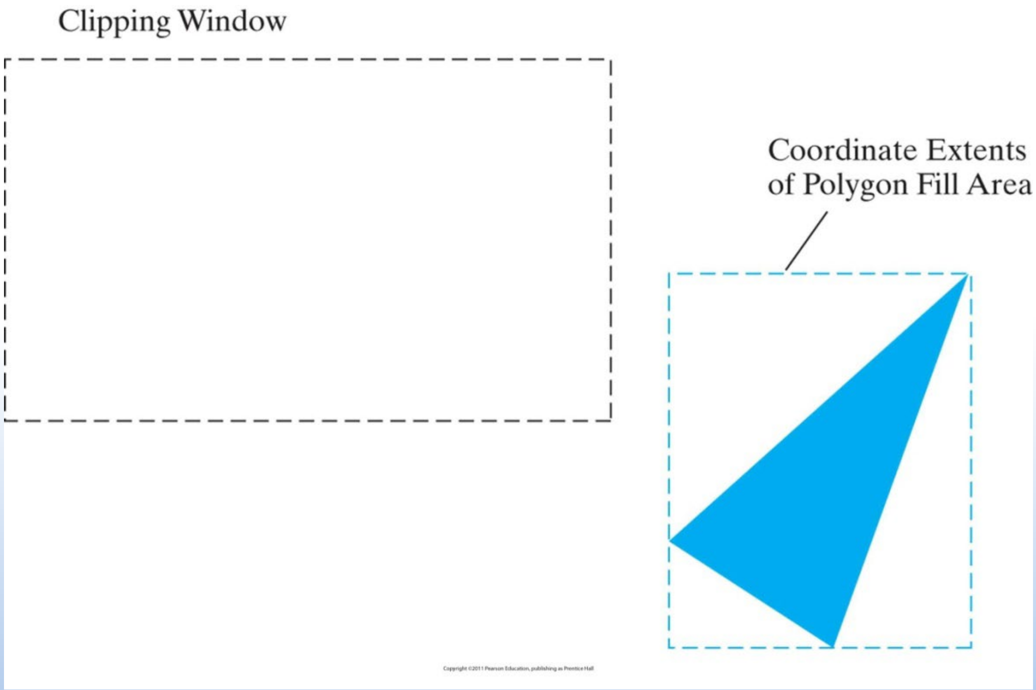
Polygon clipping

Clipping

- Polygon fill-area clipping (cont.)
  - Can process a polygon fill-area, against the borders of a clipping window, using same general approach as line clipping
    - Interior fill not applied yet
    - Line segment defined by two endpoints
    - Endpoints processed through a line clipping procedure to produce a new set of clipped endpoints at each clipping window boundary
    - Need to maintain a fill area as *an entity* (not just disjoint set of lines) as it is processed through the clipping stages

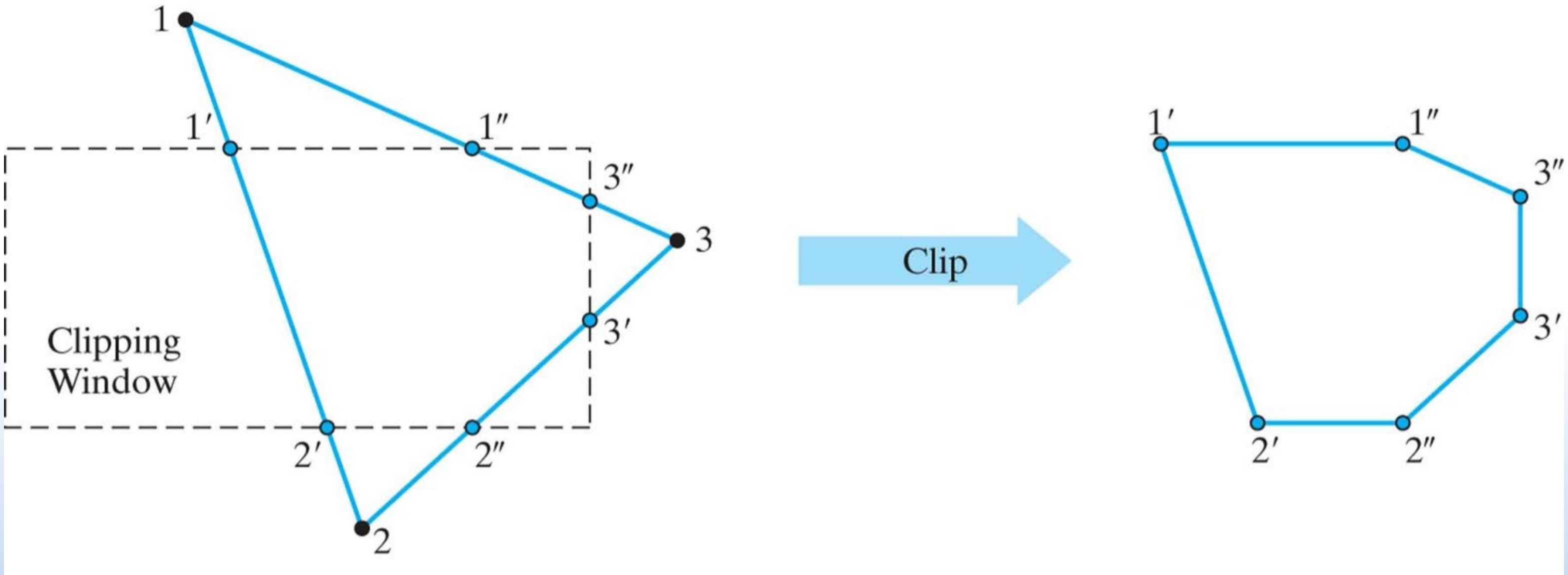
Clipping

- Polygon fill-area clipping (cont.)
  - If a polygon fill area with coordinate extents are all outside any of the clipping window boundaries
    - Eliminate polygon from further processing



Clipping

- Polygon fill-area clipping (cont.)
  - Create a new vertex list at each clipping boundary, then pass this new vertex list to the next boundary clipper

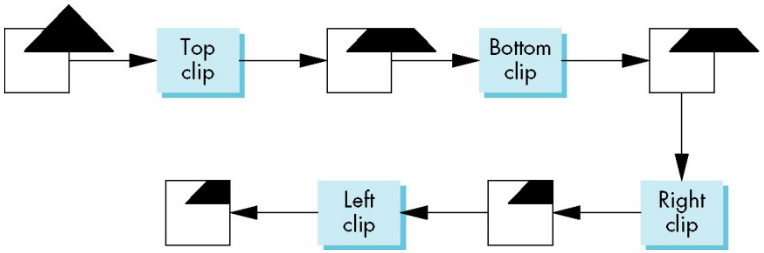


(a) (b)

Clipping

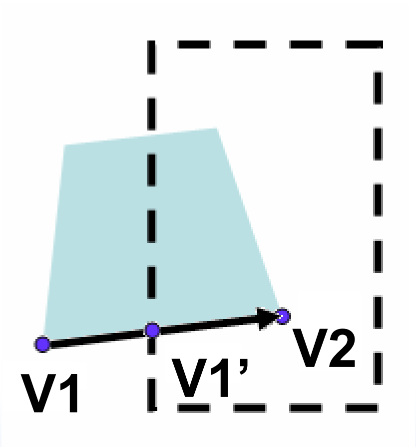
- Polygon fill-area clipping (cont.)
  - ➢ Hodgman-Sutherland polygon clipping
    - Parse all polygon edges (either in clockwise or counter-clockwise direction)
    - Clip edges against one boundary
    - Repeat for other 3 boundaries
    - 4 possible cases
      - Out → In : Save intersection point and endpoint
      - In → In : Save endpoint
      - In → Out : Save intersection point
      - Out → Out : Save nothing

    [Note: For intersection points, create intermediate vertices (these might be replaced later on)]



Clipping

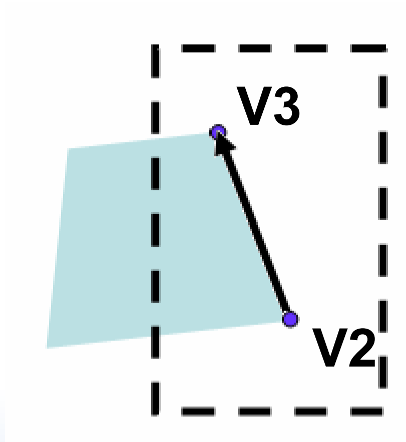
- Polygon fill-area clipping (cont.)
  - Hodgman-Sutherland polygon clipping (cont.)
    - Example for left clipping boundary only



(1)

Out → In

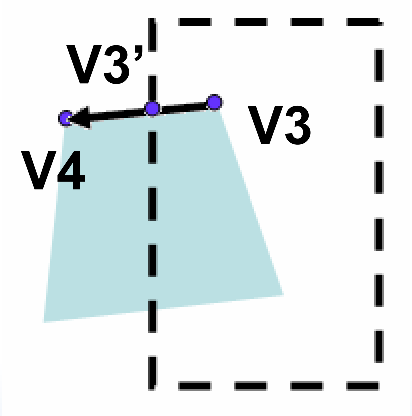
Save: V1’, V2



(2)

In → In

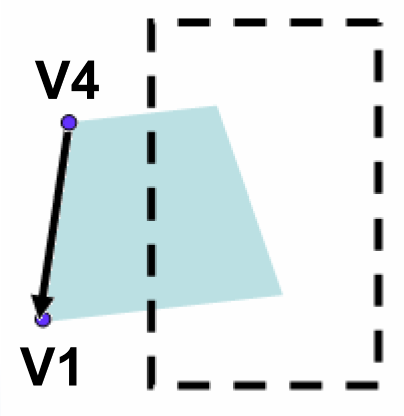
Save: V3



(3)

In → Out

Save: V3’



(4)

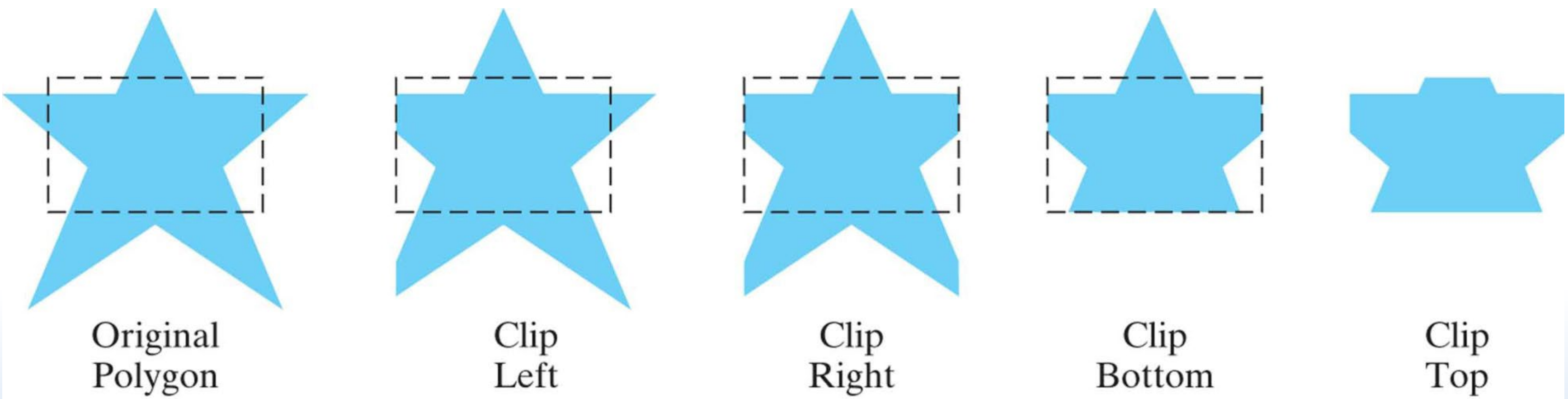
Out → Out

Save: nothing

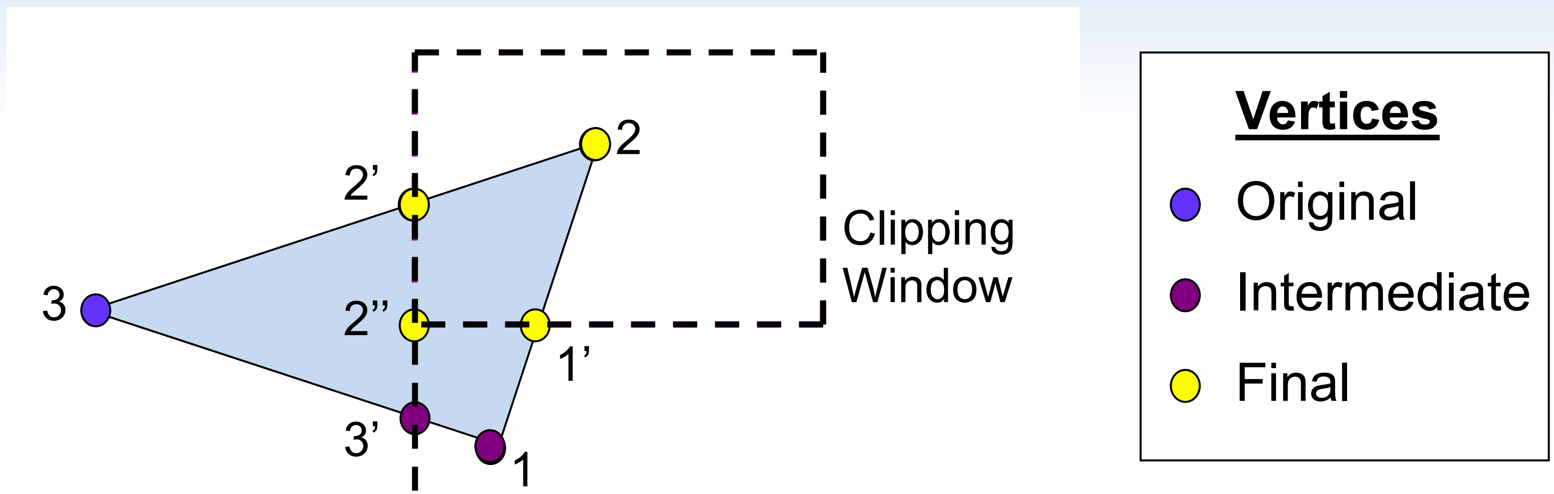
$\boxed{V1' \rightarrow V2 \rightarrow V3 \rightarrow V3'}$

Clipping

- Polygon fill-area clipping (cont.)
  - ➢ Hodgman-Sutherland polygon clipping (cont.)
    - Processing a polygon fill area against successive clipping-window boundaries



Hodgman-Sutherland Polygon Clipping

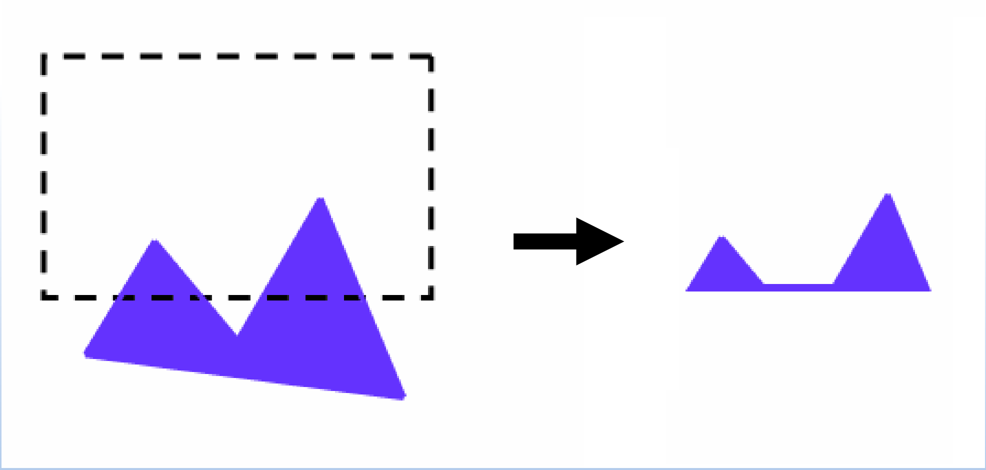


Left Clipper → Right Clipper → Bottom Clipper → Top Clipper

Left Clipper	Right Clipper	Bottom Clipper	Top Clipper
[1,2]:(in – in)→[2]	[2,2’]:(in–in)→[2’]	[2’,3’]:(in–out)→[2’’]	[2’’,1’]:(in–in)→[1’]
[2,3]:(in–out)→[2’]	[2’,3’]:(in–in)→[3’]	[3’,1]:(out–out)→[ ]	[1’,2]: (in – in) → [2]
[3,1]:(out–in)→[3’,1]	[3’,1]:(in–in)→[1]	[1,2]:(out–in)→[1’,2]	[2,2’]: (in – in) → [2’]
	[1,2]:(in–in)→ [2]	[2,2’]:(in–in)→[2’]	[2’,2’’]:(in – in)→[2’’]

Clipping

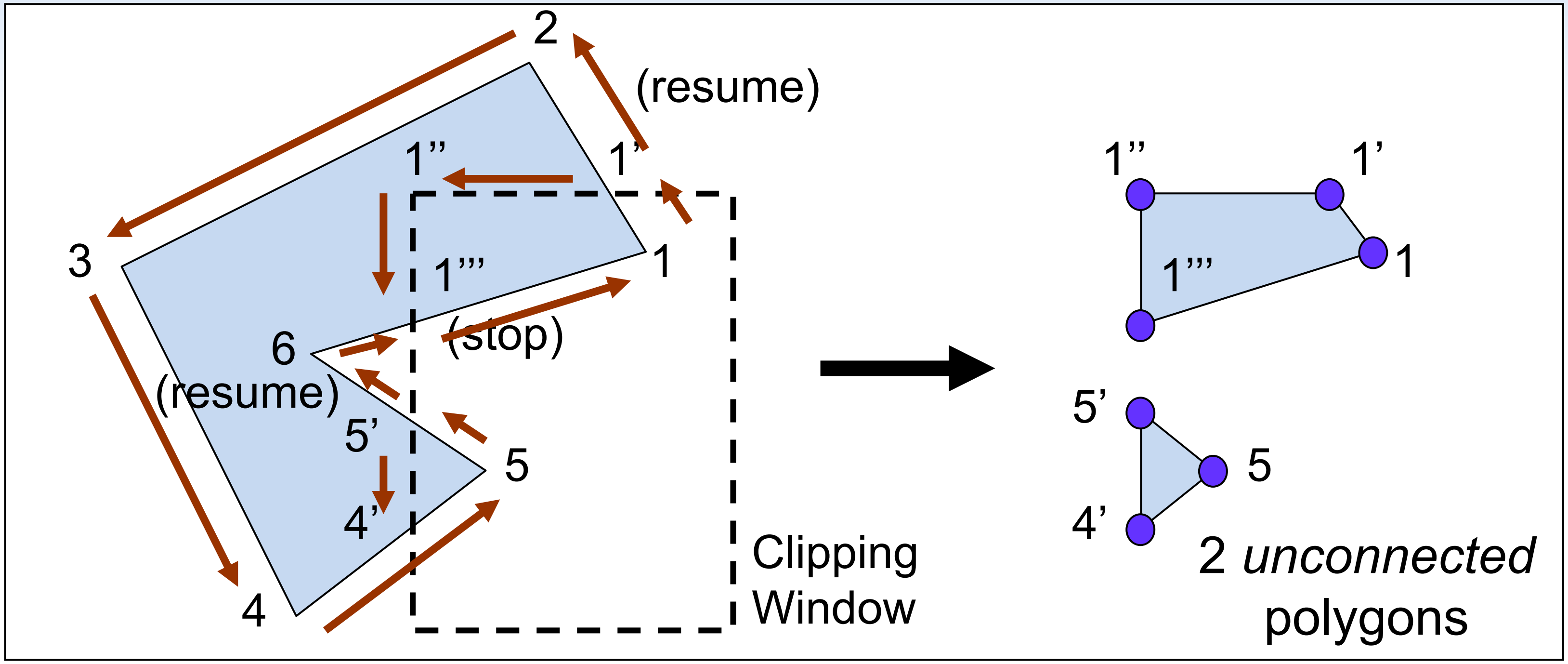
- Polygon fill-area clipping (cont.)
  - Hodgman-Sutherland polygon clipping (cont.)
    - Potential problem
      - Only produces one list of output vertices
        - » Fine for convex polygons
        - » Extraneous lines may be displayed when clipping concave polygons



Clipping

- Polygon fill-area clipping (cont.)
  - ➢ Weiler-Atherton polygon clipping
    - More general, can be used to clip either convex or concave area filled polygons
    - Instead of just tracing the polygon’s perimeter (i.e. the previous algorithm)
      - When exit-intersection point encountered take a detour and trace along the clipping boundary
      - Need to ‘remember’ exit-intersection point in order to continue tracing from here later
    - Edge traversal can be clockwise or counter-clockwise but must maintain same direction

Weiler-Atherton Polygon Clipping



<u>Vertex List</u>

[1, 2, 3, 4, 5, 6]

<u>New Vertex List</u>

[1, 1', 1'', 1''']

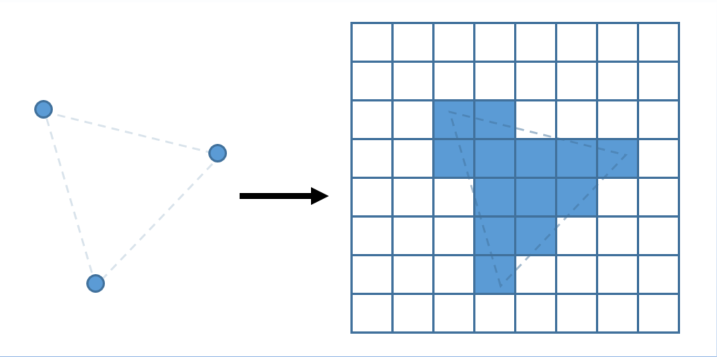
[4', 5, 5']

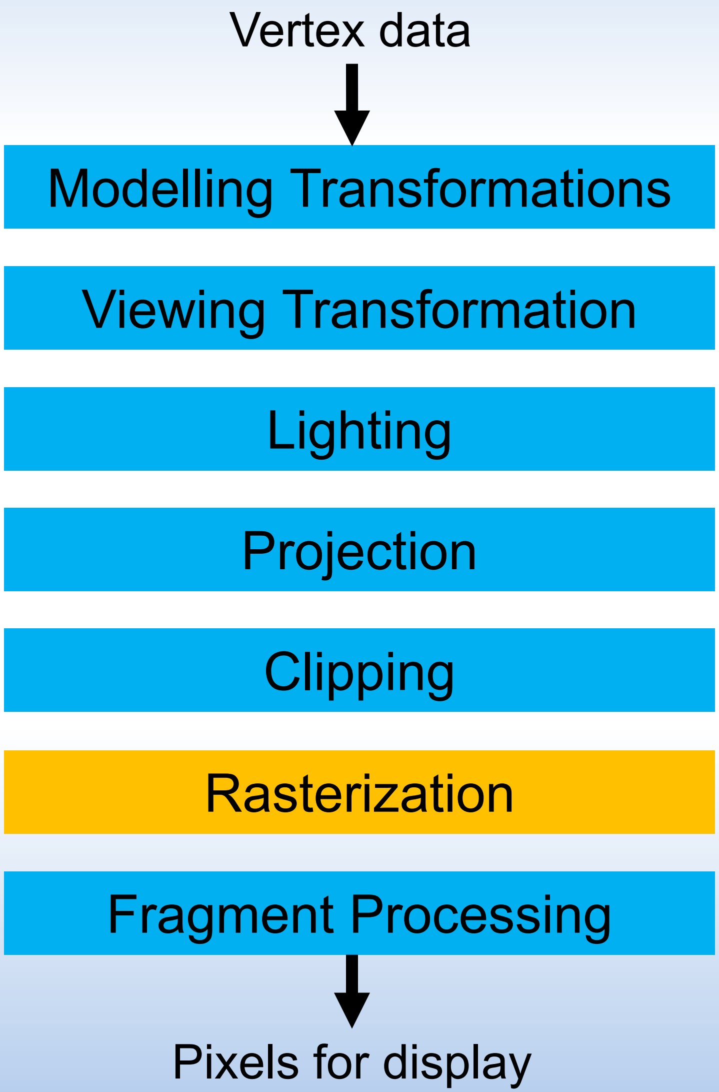
Result = Two *unconnected* polygons

Rasterization

The Computer Graphics Pipeline

- Rasterization
  - ➢ Scan converts vertices into **screen space**
    - Input: vertices (after clipping)
    - Output: fragments
  - ➢ After rasterization, everything in fragments
    - No longer polygons or vertices





Overview

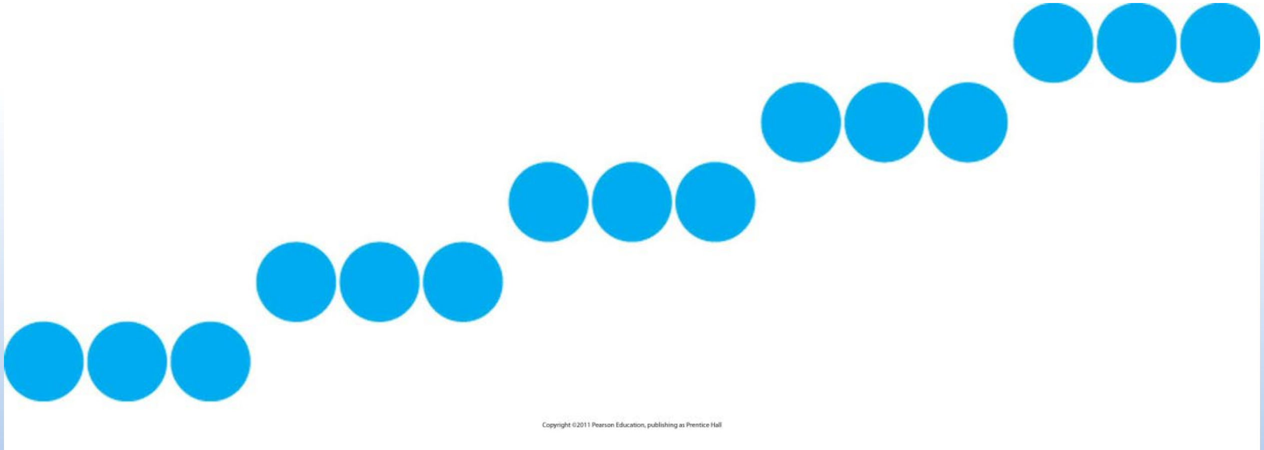
- Rasterization
  - Line drawing algorithms
    - Digital Differential Analyzer (DDA) algorithm
    - Bresenham's algorithm
  - Polygon fill algorithms
    - Scan-line polygon fill

Rasterization

- What is rasterization (scan conversion)?
  - Determine which pixels that are inside primitive specified by a set of vertices
    - Want to convert continuous geometry, inside viewing region, into discrete pixels
    - Fragments have a location (pixel location) and other attributes such colour and texture coordinates that are determined by interpolating values at vertices
      - Pixel colours determined later using colour, texture, and other vertex properties
    - Want to achieve fast yet accurate results

Line-Drawing Algorithms

- Line-drawing algorithms
  - ➢ Line segments defined by endpoints
  - ➢ Must project endpoints to integer screen coordinates
    - Then determine the nearest pixel positions along the line path between endpoints
    - This digitises the line into a set of discrete integer positions, in general, only approximates the actual line path

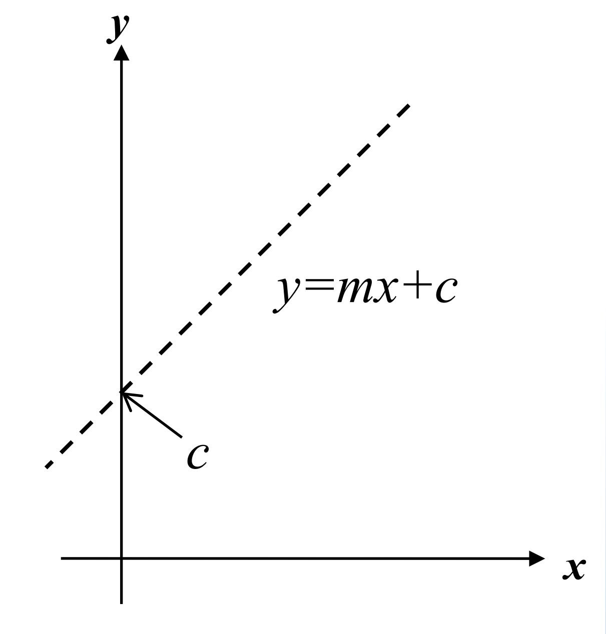


Line-Drawing Algorithms

- Line equation
  - Determine pixel positions from geometric properties of the line
    - Cartesian slope-intercept equation for a straight line

$$y = mx + c$$

    - *m* is the slope of the line
    - *c* is the y intercept



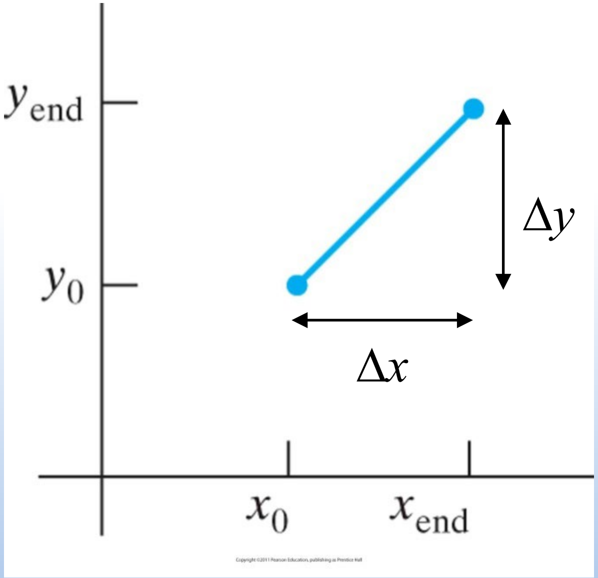
Line-Drawing Algorithms

- Line equation

$$y = mx + c$$

  - *m* is the slope of the line
  - *c* is the y intercept

➢ Given two endpoints (x_0, y_0) and (x_{end}, y_{end})

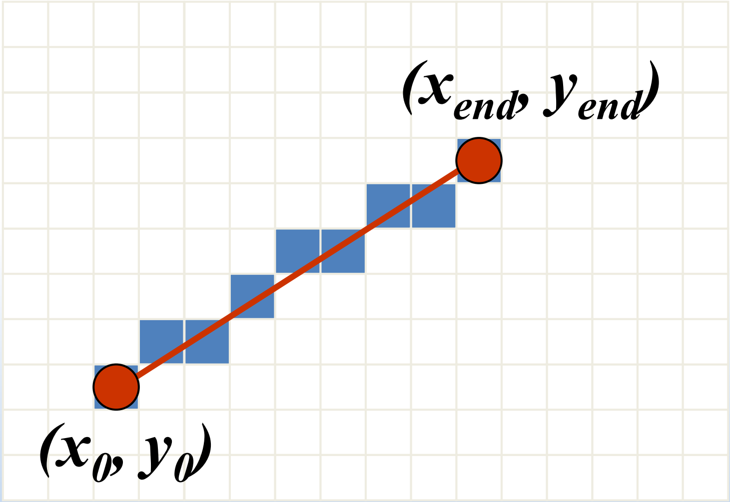


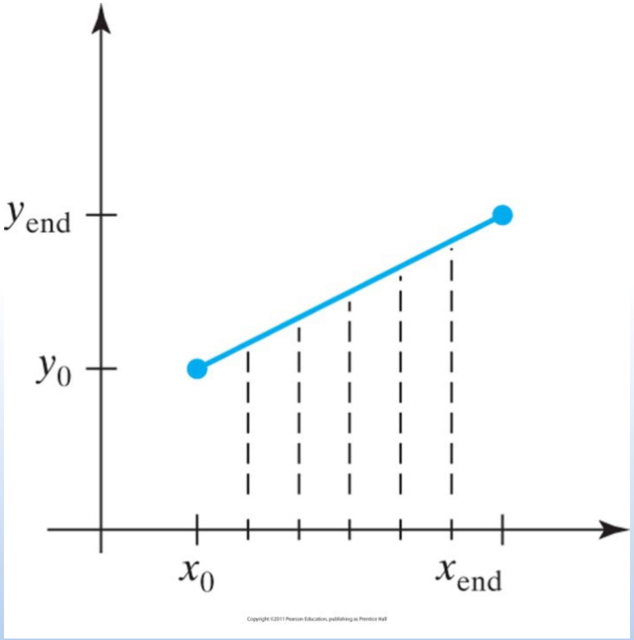
$$m = \frac{\Delta y}{\Delta x} = \frac{y_{end} - y_0}{x_{end} - x_0}$$

$$c = y_0 - mx_0$$

Line-Drawing Algorithms

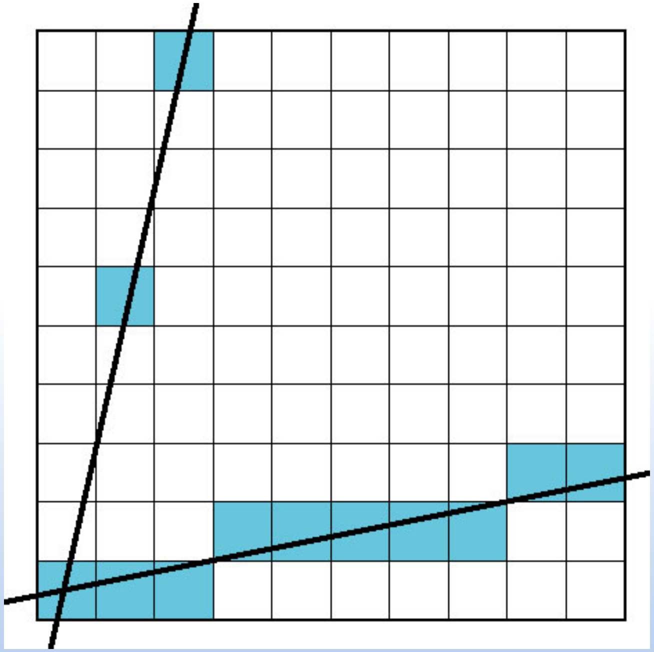
- Line-drawing algorithms
  - Given two endpoints (x_0, y_0) and (x_{end}, y_{end}) want to find set of pixels that will represent the line
  - Needs to be continuous (without gaps), of uniform thickness and brightness

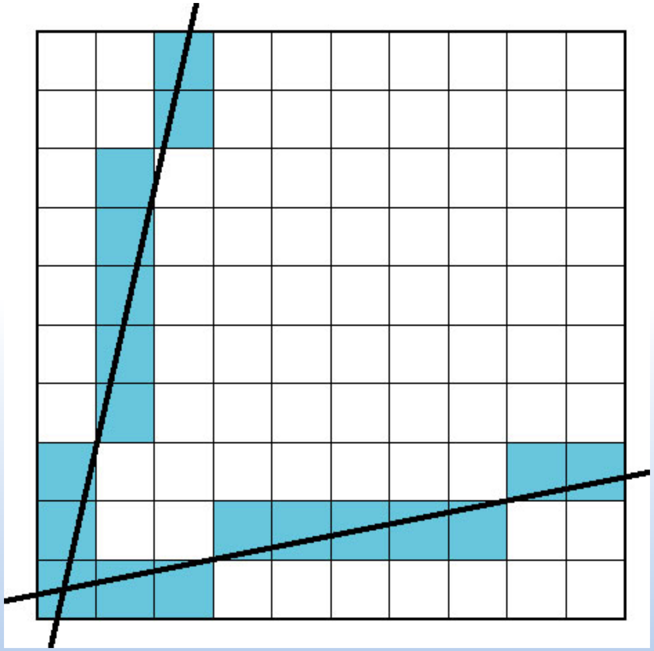




Line-Drawing Algorithms

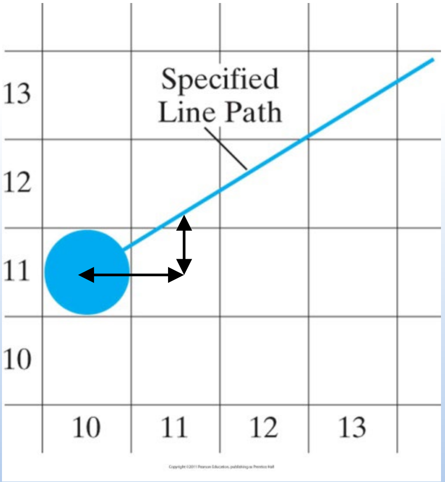
- Line-drawing algorithms
  - ➢ Needs to be continuous (without gaps)
    - For $|m| < 1$, sample unit positions along x axis
    - For $|m| > 1$, sample unit positions along y axis

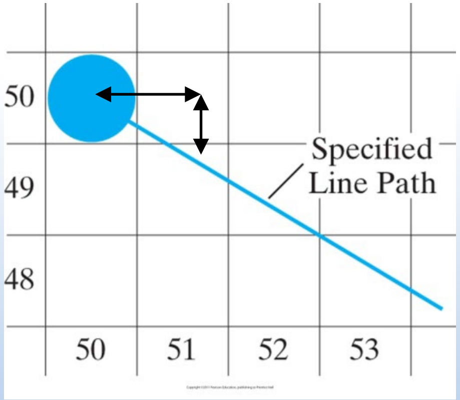




Line-Drawing Algorithms

- Digital Differential Analyzer (DDA) algorithm
  - ➢ Line sampled at *unit interval* steps along one axis
    - Which axis depends on the value of $|m|$
    - Corresponding integer value *nearest* to the line path determined as the other coordinate
      - – Every step increment by an amount less than a unit
      - – Round floating point value to an integer

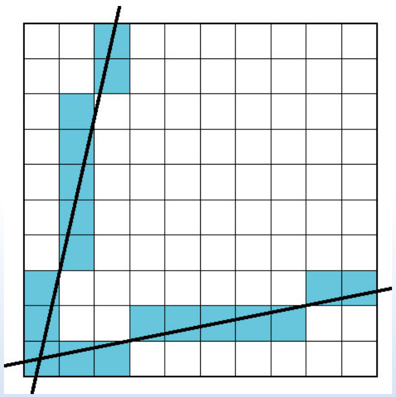




Line-Drawing Algorithms

- DDA algorithm

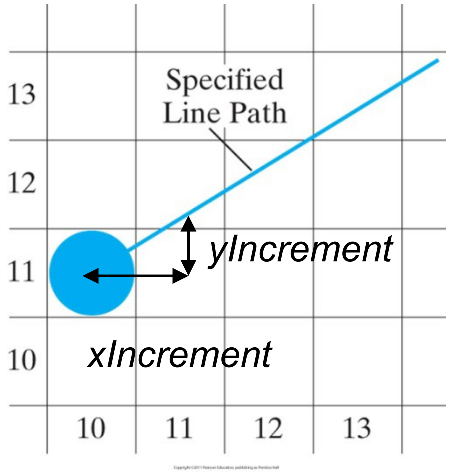
```
inline int round (const float a) {return int (a + 0.5);}

void lineDDA(int x0, int y0, int xEnd, int yEnd)
{
   int dx = xEnd - x0, dy = yEnd - y0, steps, k;
   float xIncrement, yIncrement, x = x0, y = y0;

   if (fabs(dx) > fabs(dy))
      steps = fabs(dx);
   else
      steps = fabs(dy);

   //continue...
```



Line-Drawing Algorithms

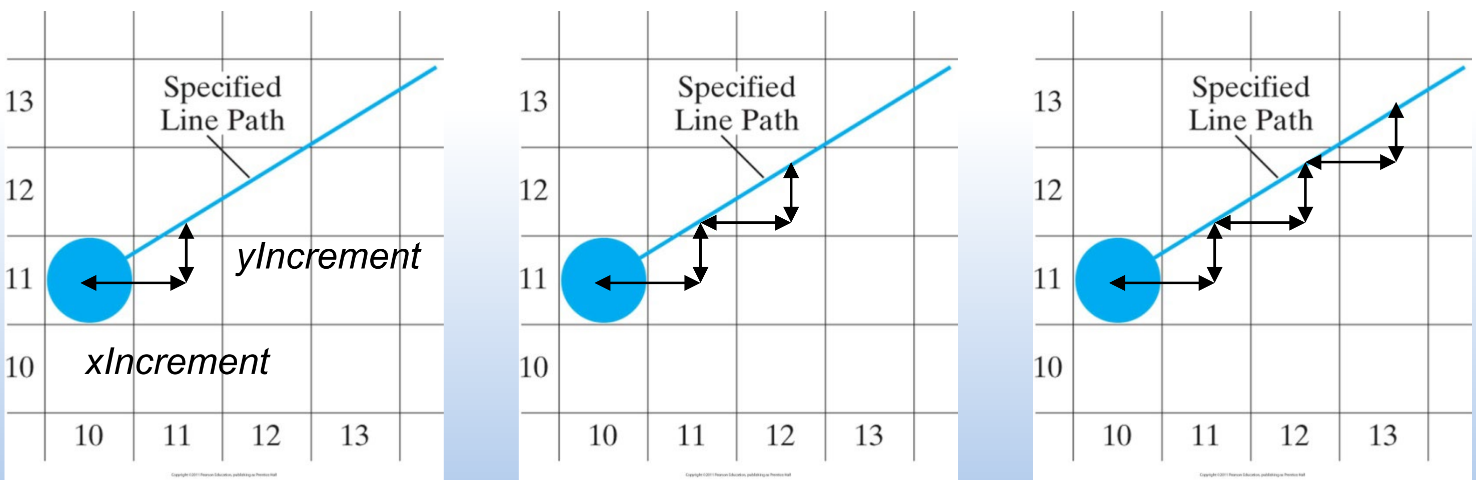
- DDA algorithm

```
//continue...

xIncrement = (float)dx / (float)steps;
yIncrement = (float)dy / (float)steps;

setPixel(round(x), round(y));

for(k = 0; k < steps; k++)
{
   x += xIncrement;
   y += yIncrement;
   setPixel(round(x), round(y));
}
}
```



Line-Drawing Algorithms

- DDA algorithm
  - ➢ Example
    - Determine number of steps, xIncrement and yIncrement
    - Set first endpoint (x_0, y_0)
    - Repeat for number of steps
      – Step by unit interval in one coordinate, add increment in other coordinate and round to nearest integer value

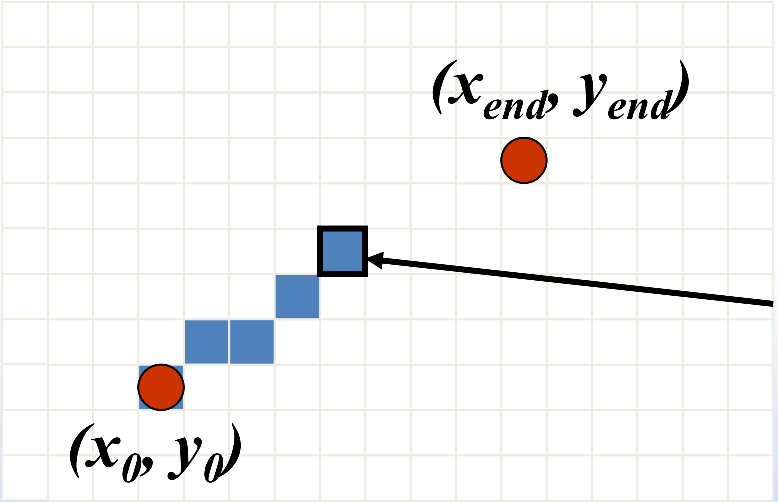


Line-Drawing Algorithms

- DDA algorithm
  - ➢ What's bad about this algorithm?
    - Has to deal with floating arithmetic and rounding operations which is time consuming
    - Accumulation of round off errors of floating-point increment can cause calculated pixel positions to drift away from the true line path
  - ➢ Not very efficient in terms of speed and accuracy

Line-Drawing Algorithms

- Bresenham's line algorithm
  - Only uses incremental integer calculations
  - Can be adapted to display circles and other curves
  - Determine next sample from current pixel position

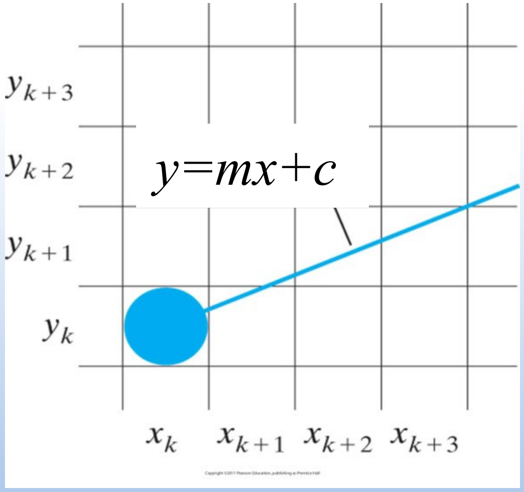


Current pixel, which pixel to draw next?

Line-Drawing Algorithms

- Bresenham's line algorithm
  - ➢ Consider the case of positive slope less than 1.0

    (i.e. $0 < m < 1$)
    - Pixel positions along line are determined by sampling at unit *x* intervals
      - – Progress using increments of unit x (i.e. x_k+1)
      - – At pixel (x_p, y_p), next pixel at will either be (x_p+1, y_p) or (x_p+1, y_p+1)
      - – Which pixel to choose?



Line-Drawing Algorithms

- Bresenham's line algorithm

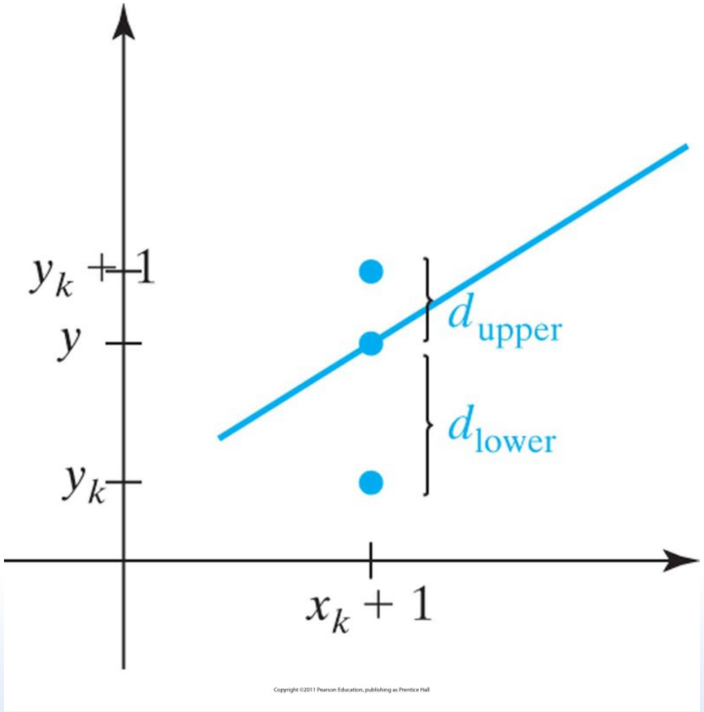
$$y = m(x_k + 1) + c$$

$$d_{upper} = (y_k + 1) - y$$

$$d_{upper} = y_k + 1 - m(x_k + 1) + c$$

$$d_{lower} = y - y_k$$

$$d_{lower} = m(x_k + 1) + c - y_k$$



$$d_{lower} - d_{upper} = 2m(x_k + 1) - 2y_k + 2c - 1$$

Line-Drawing Algorithms

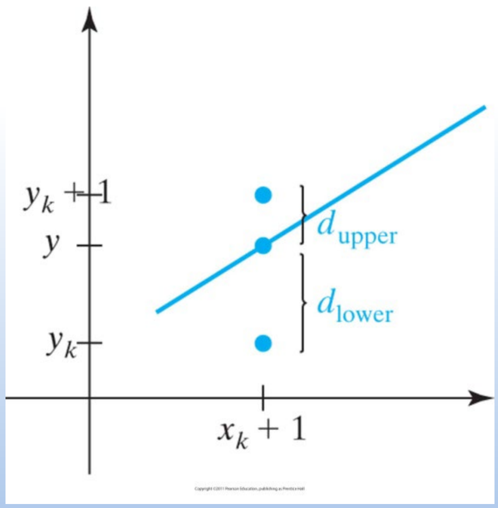
- Bresenham's line algorithm
  - Decision parameter p_k for the kth step
  - Want only integer calculations

$$d_{lower} - d_{upper} = 2m(x_k + 1) - 2y_k + 2c - 1 \qquad m = \frac{\Delta y}{\Delta x}$$

$$d_{lower} - d_{upper} = 2\frac{\Delta y}{\Delta x}(x_k + 1) - 2y_k + 2c - 1$$

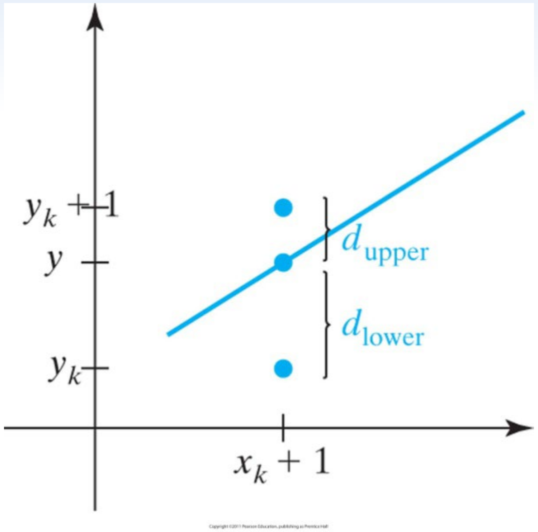
$$p_k = \Delta x(d_{lower} - d_{upper})$$

$$p_k = 2\Delta y x_k - 2\Delta x y_k + 2\Delta y + \Delta x(2c - 1)$$



Line-Drawing Algorithms

- Bresenham's line algorithm
  - Only the sign of p_k matters
    - As this is the same sign as $d_{lower} - d_{upper}$
    - If $p_k < 0$, use lower pixel
  - p_k can be calculated incrementally
    - Separate the pixel independent portion



Line-Drawing Algorithms

- Bresenham's line algorithm
  - For the case where $|m| < 1.0$
    1. Input two endpoints and store left endpoint in (x_0, y_0)
    2. Set colour for (x_0, y_0), i.e. plot first point
    3. Calculate constants Δx, Δy, $2\Delta y$ and $2\Delta y - 2\Delta x$, and obtain starting decision parameter $p_0 = 2\Delta y - \Delta x$
    4. At each x_k, starting at $k = 0$
       – If $p_k < 0$, use lower pixel, i.e. (x_k+1, y_k), and
         $p_{k+1} = p_k + 2\Delta y$
       – Otherwise, use upper pixel, i.e. (x_k+1, y_k+1), and
         $p_{k+1} = p_k + 2\Delta y - 2\Delta x$
    5. Repeat step 4, $\Delta x - 1$ more times

Line-Drawing Algorithms

- Bresenham's line algorithm
  - For the case where $|m| < 1.0$
    - Calculate constants Δx, Δy, $2\Delta y$ and $2\Delta y - 2\Delta x$, and obtain starting decision parameter $p_0 = 2\Delta y - \Delta x$

```
void lineBres(int x0, int y0, int xEnd, int yEnd)
{
   int dx = fabs(xEnd - x0), dy = fabs(yEnd - y0);
   int p = 2*dy - dx;
   int twoDy = 2*dy,  twoDyMinusDx = 2*(dy - dx);
   int x, y;

//continue...
```

Line-Drawing Algorithms

- Bresenham's line algorithm
  - For the case where $|m| < 1.0$

```
//continue...

/* Determine which endpoint to use as start position. */
   if (x0 > xEnd) {
      x = xEnd;
      y = yEnd;
      xEnd = x0;
   }
   else {
      x = x0;
      y = y0;
   }
   setPixel (x, y);

//continue...
```

- Input two endpoints and store left endpoint in (x_0, y_0)
- Set colour for (x_0, y_0), i.e. plot first point

Line-Drawing Algorithms

- Bresenham's line algorithm
  - For the case where $|m| < 1.0$

```
//continue...

   while (x < xEnd) {
      x++;
      if (p < 0)
         p += twoDy;
      else {
         y++;
         p += twoDyMinusDx;
      }
      setPixel (x, y);
   }
}
```

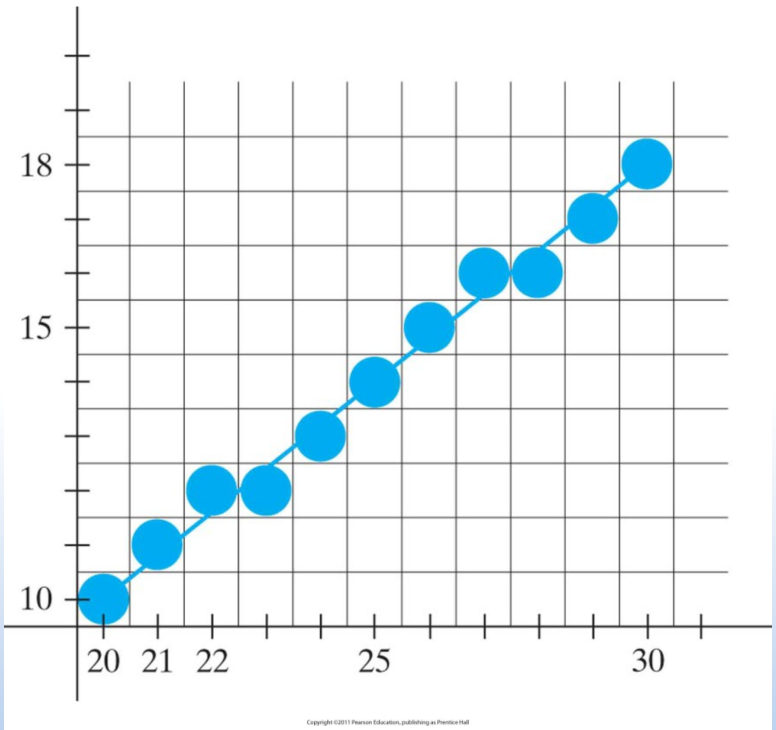
- At each x_k, starting at $k = 0$
  - If $p_k < 0$, use lower pixel and
    $p_{k+1} = p_k + 2\Delta y$
  - Otherwise, use upper pixel and
    $p_{k+1} = p_k + 2\Delta y - 2\Delta x$
- Repeat $\Delta x - 1$ more times

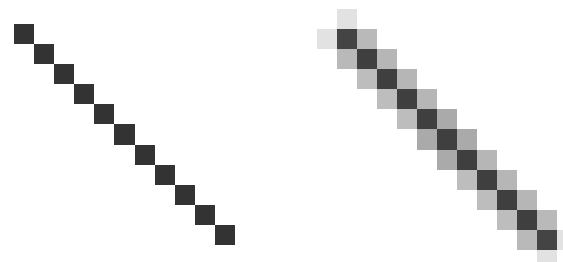
Line-Drawing Algorithms

- Bresenham's line algorithm
  - So far considered case where $|m| < 1.0$
  - Other cases
    - For –ve slope just do opposite (i.e. down instead of up)
    - Obviously if slope is 1 or -1, then just choose diagonal pixels
    - For $|m| > 1.0$ just interchange the roles of the x and y directions

Rasterization

- Aliasing
  - Discrete sampling of continuous geometry will produce aliasing artifacts

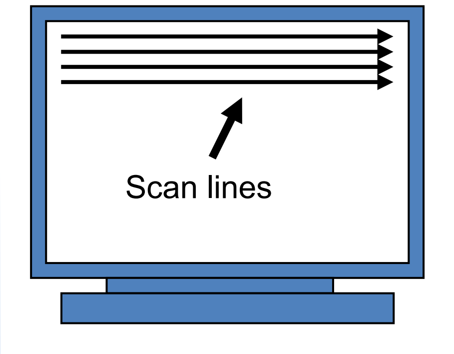


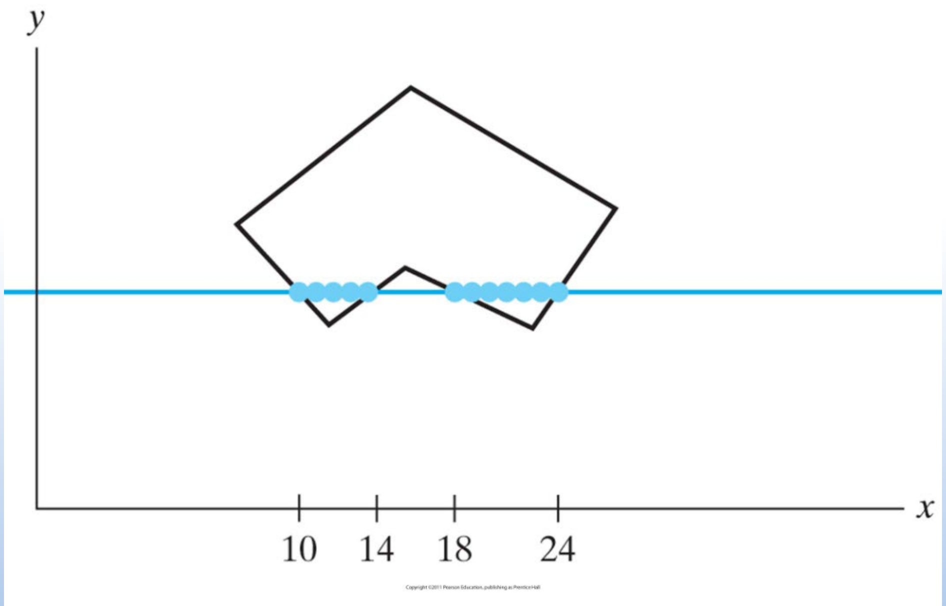


Anti-aliasing techniques can be used to smoothen aliasing artifacts

Polygon Fill Algorithms

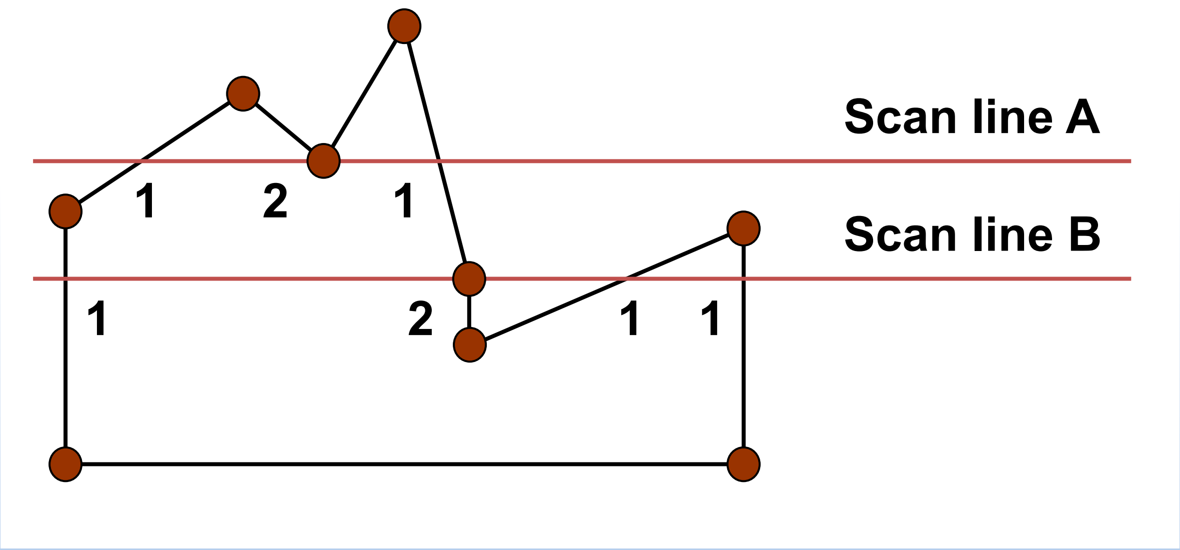
- Scan-line polygon fill
  - First determine intersection positions of polygon boundaries with the screen scan lines
  - Then apply fill colour to each section of the scan line that lies within the interior of the fill region





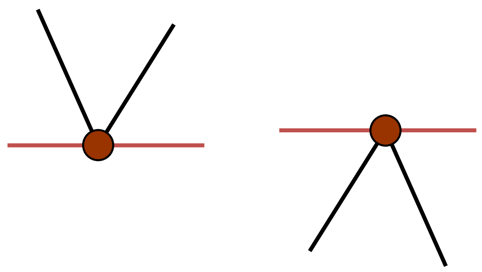
Polygon Fill Algorithms

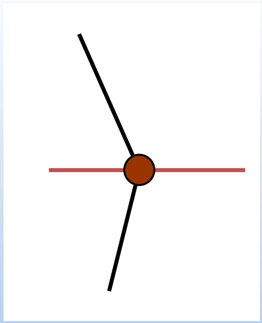
- Scan-line polygon fill
  - ➢ Whenever a scan-line passes through a vertex, it intersects two polygon edges at that point
    - Scan line A even number of intersections
    - Scan line B odd number of intersections



Polygon Fill Algorithms

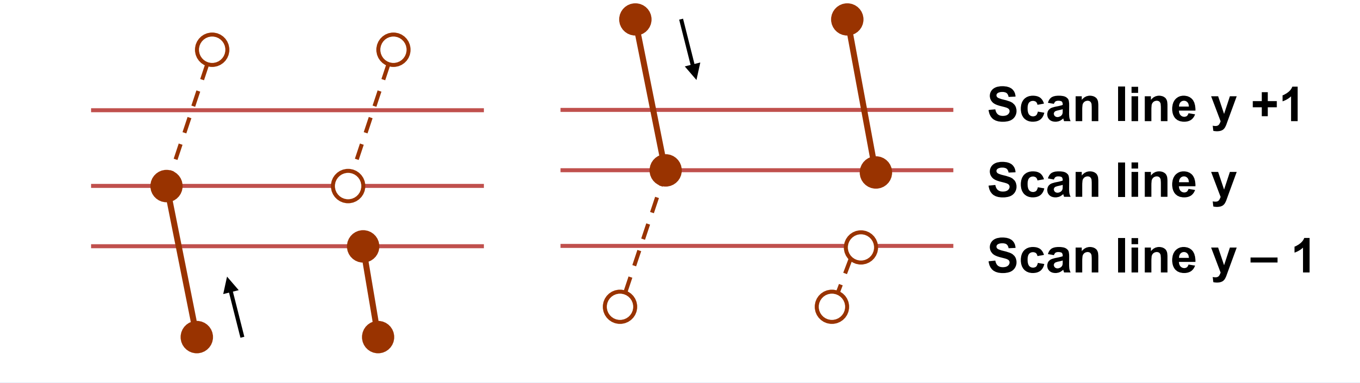
- Scan-line polygon fill
  - ➢ Can detect topological difference between scan-lines
  - ➢ Trace around polygon boundary, either clockwise or counterclockwise order, and observe relative changes in y
    - Local maximum or minimum, not a problem count twice
    - Monotonically increase or decrease, need to count only once
      - – One method to adjust this is to shorten polygon edges by splitting vertices





Polygon Fill Algorithms

- Scan-line polygon fill
  - Splitting vertices
    - Process edges in order around the polygon perimeter
    - If edge monotonically increasing or decreasing, shorten lower edge

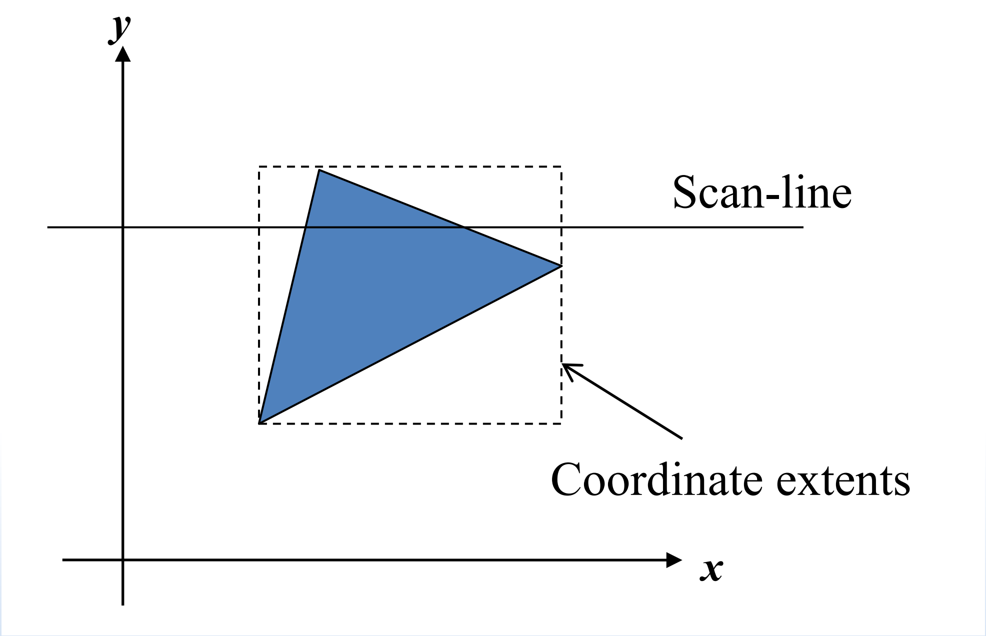


Polygon Fill Algorithms

- Scan-line polygon fill
  - Simpler for convex polygons
    - For each scan-line, process polygon edges only until two boundary intersection crossing polygon interior found
    - Use coordinate extents to determine which edges cross a scan-line
    - Intersections with edges determine interior pixel span for scan-line, any vertex crossing counted as single boundary intersection point
    - When scan-line hits a single vertex (e.g. at an apex), only plot that point
    - Triangles only 3 edges to process

Polygon Fill Algorithms

- Scan-line polygon fill
  - For convex polygons



References

- Among others, material sourced from
  - Hearn, Baker & Carithers, “Computer Graphics with OpenGL”, Pearson/Prentice-Hall
  - Angel & Shreiner, “Interactive Computer Graphics: A Top-Down Approach with OpenGL”, Pearson/Addison Wesley
  - Akenine-Moller, Haines & Hoffman, “Real Time Rendering”, A.K. Peters
  - http://en.wikipedia.org/wiki/