Computer Graphics (CS 543)
Lecture 9: Clipping, Viewport Transformation & Hidden Surface Removal

Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)

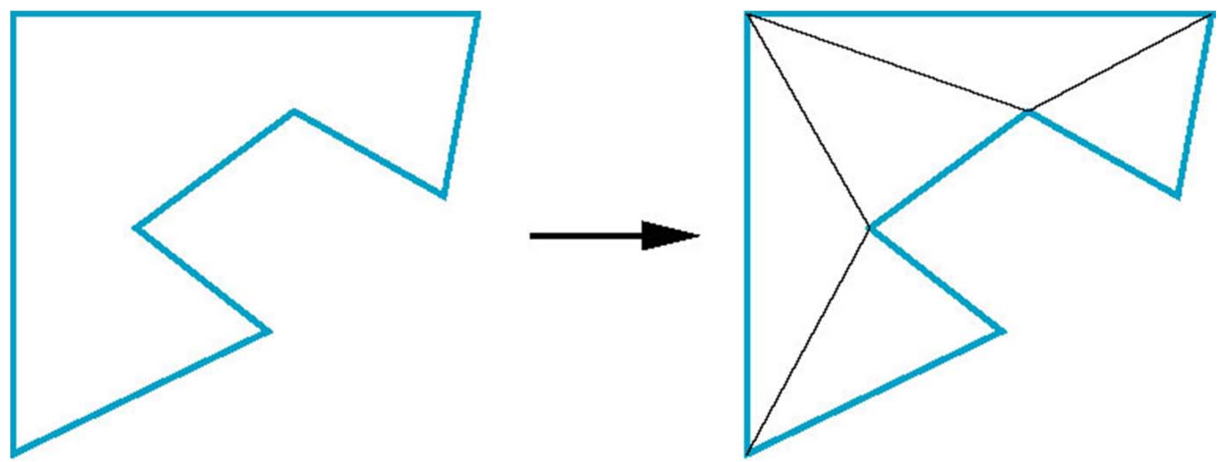
Polygon Clipping

- 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

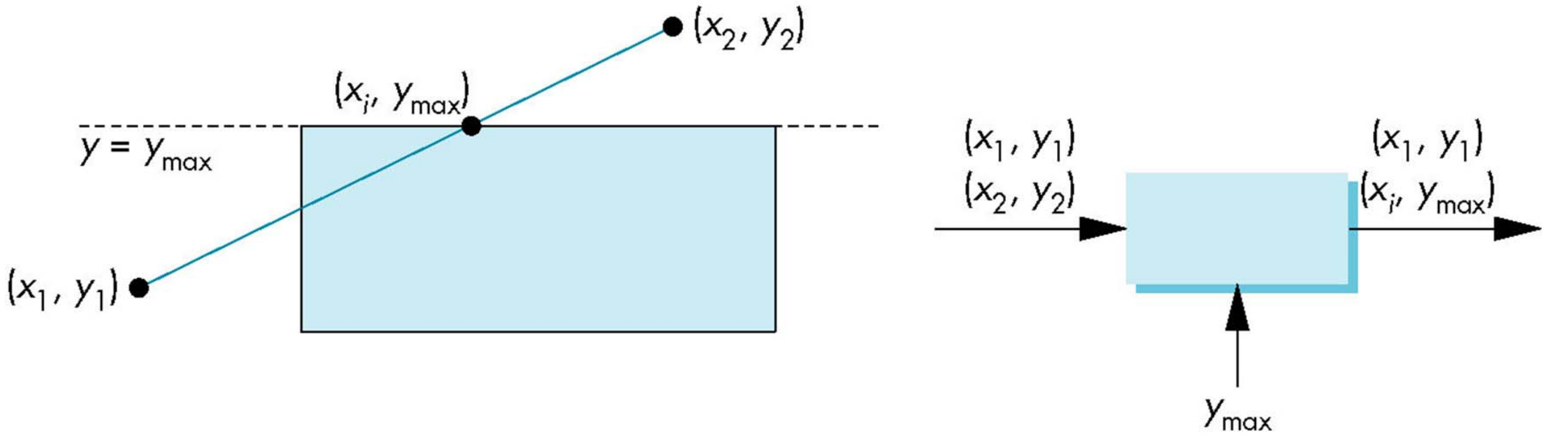
Tessellation and Convexity

- One strategy is to replace nonconvex (*concave*) polygons with a set of triangular polygons (a *tessellation*)
- Also makes fill easier
- Tessellation code in GLU library



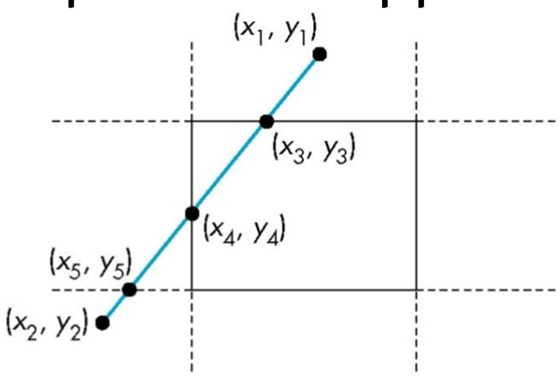
Clipping as a Black Box

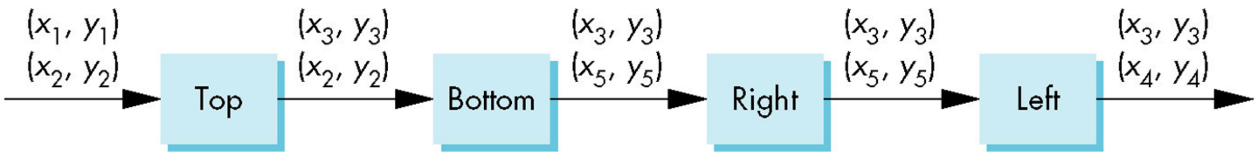
- Can consider line segment clipping as a process that takes in two vertices and produces either no vertices or the vertices of a clipped line segment



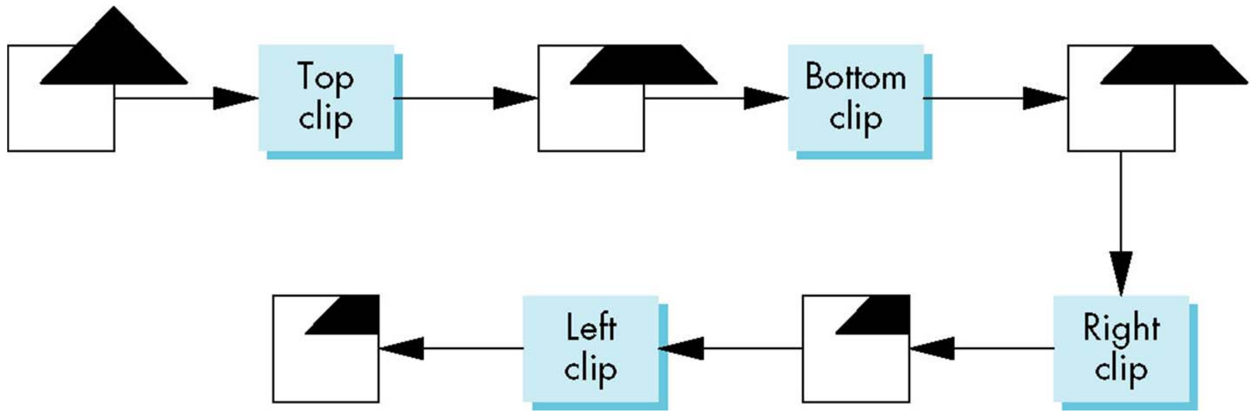
Pipeline Clipping of Line Segments

- Clipping against each side of window is independent of other sides
  - Can use four independent clippers in a pipeline





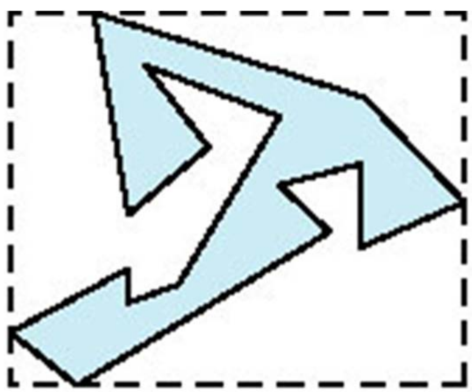
Pipeline Clipping of Polygons



- Three dimensions: add front and back clippers
- Strategy used in SGI Geometry Engine
- Small increase in latency

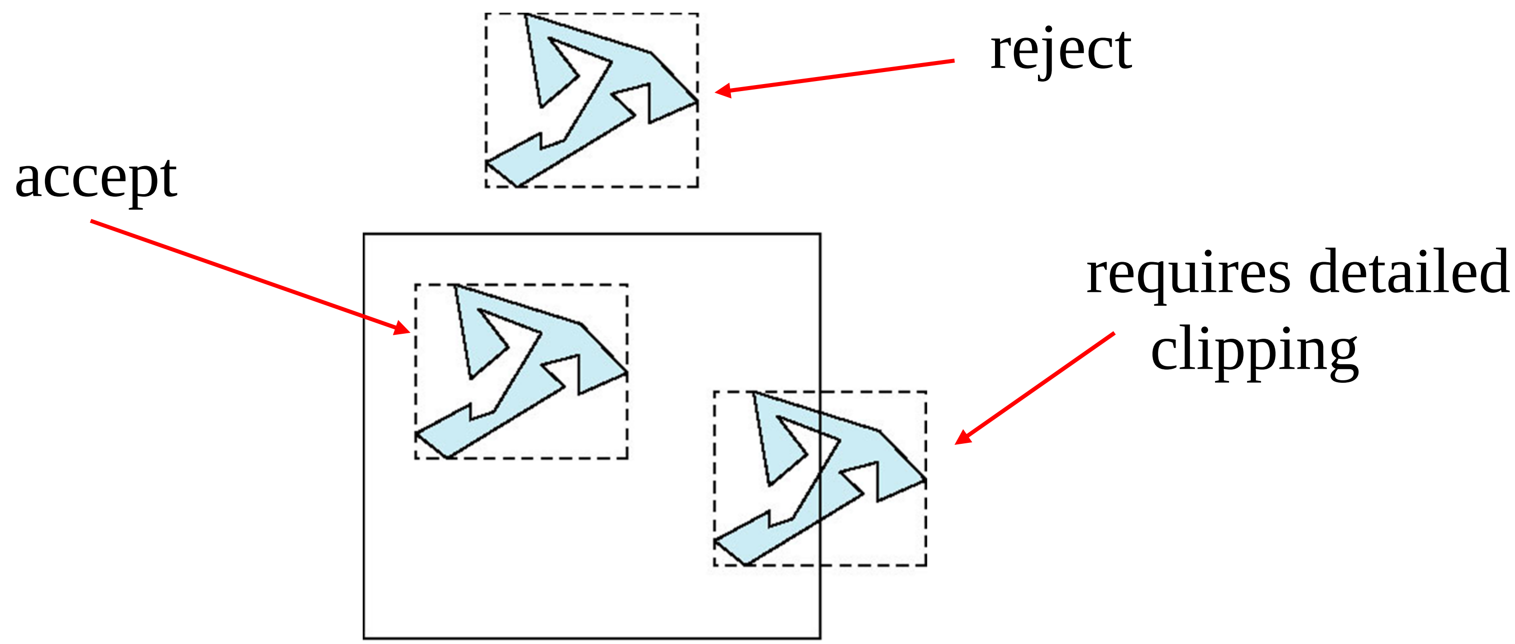
Bounding Boxes

- Rather than doing clipping on a complex polygon, we can use an *axis-aligned bounding box* or *extent*
  - Smallest rectangle aligned with axes that encloses the polygon
  - Simple to compute: max and min of x and y



Bounding boxes

Can usually determine accept/reject based only on bounding box



Clipping and Hidden Surface Removal

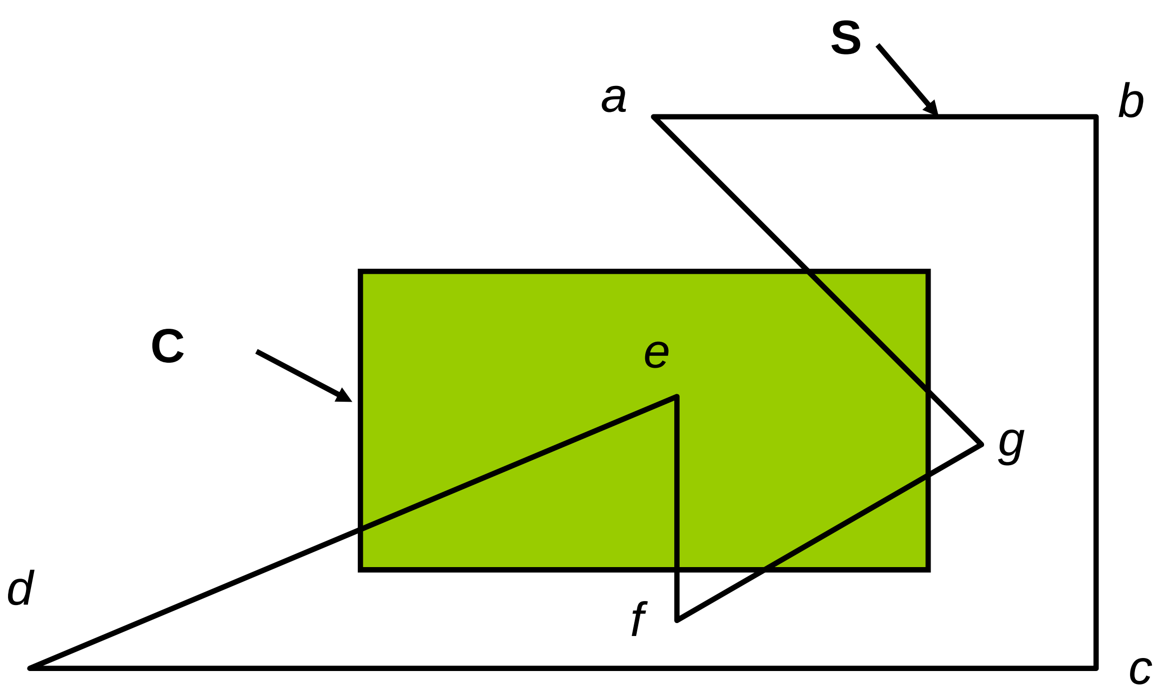
- Clipping has much in common with hidden-surface removal
- In both cases, we are trying to remove objects that are not visible to the camera
- Often we can use visibility or occlusion testing early in the process to eliminate as many polygons as possible before going through the entire pipeline

Clipping Polygons

- Cohen-Sutherland and Liang-Barsky clip line segments against each window in turn
- Polygons can be fragmented into several polygons during clipping
- May need to **add** edges
- Need more sophisticated algorithms to handle polygons:
  - *Sutherland-Hodgman:* any subject polygon against a convex clip polygon (or window)
  - *Weiler-Atherton:* Both subject polygon and clip polygon can be concave

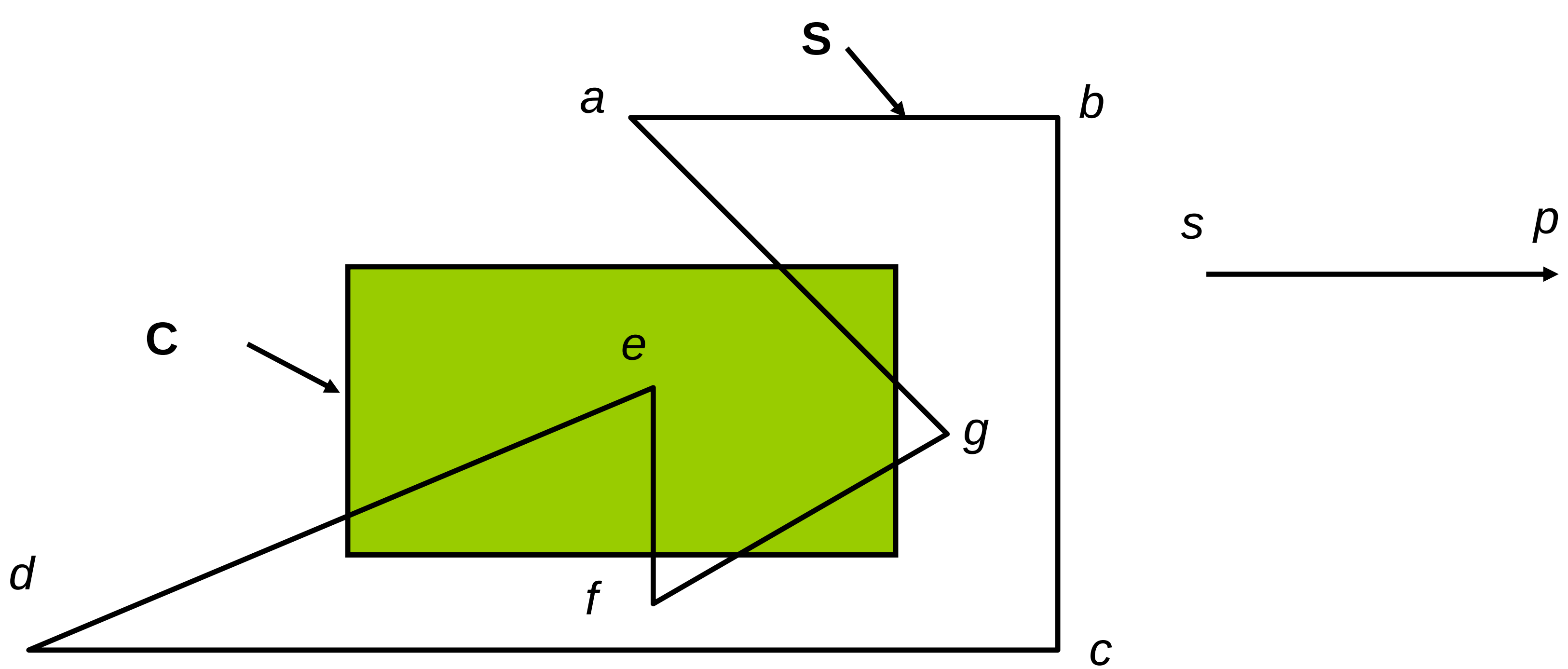
Sutherland-Hodgman Clipping

- Consider **Subject polygon, S** to be clipped against a **clip polygon, C**
- Clip each edge of S against C to get clipped polygon
- S is an ordered list of vertices *a b c d e f g*



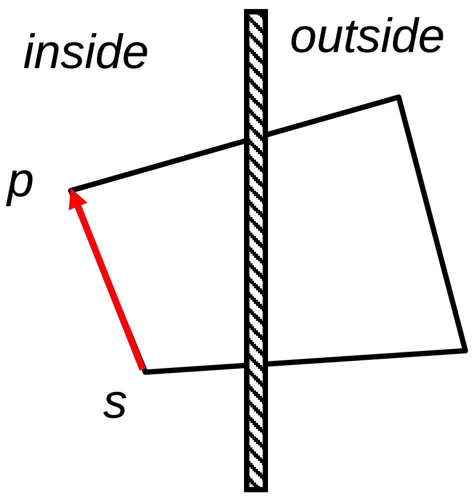
Sutherland-Hodgman Clipping

- Traverse S vertex list edge by edge
- i.e. successive vertex pairs make up edges
- E.g. *ab, bc, de*, … etc are edges
- Each edge has first point *s* and endpoint *p*



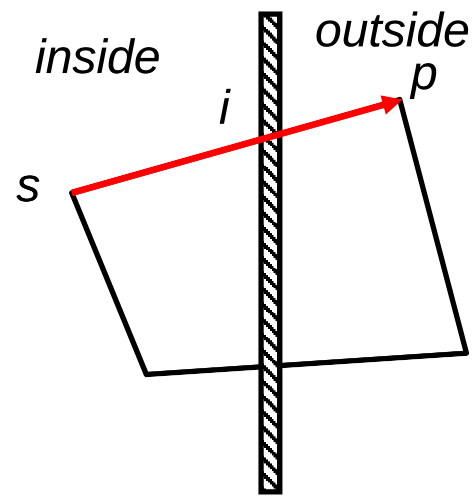
Sutherland-Hodgman Clipping

- For each edge of S, output to **new vertex** depends on whether *s* or/and *p* are inside or outside C
- 4 possible cases:



Case A: Both s and p are inside:

output p



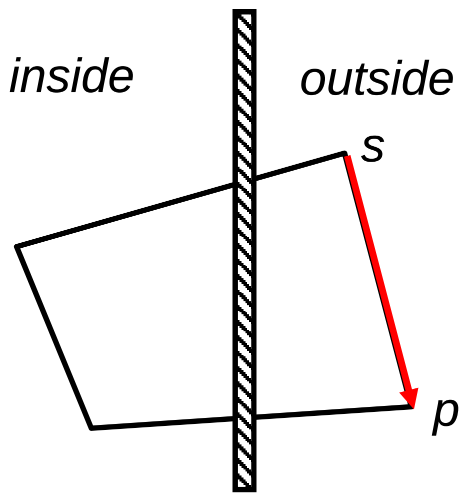
Case B: s inside, p outside:

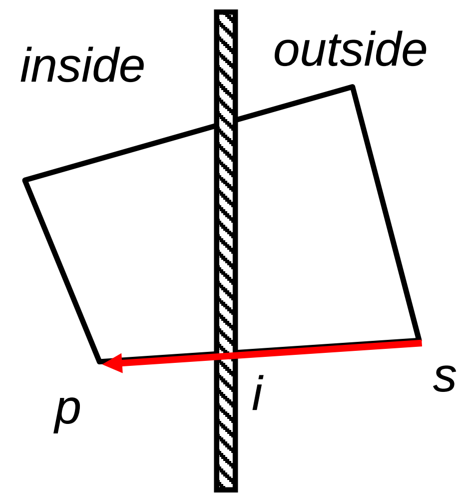
Find intersection i,

output i

Sutherland-Hodgman Clipping

- And....





Case C: Both s and p outside: ***output nothing***

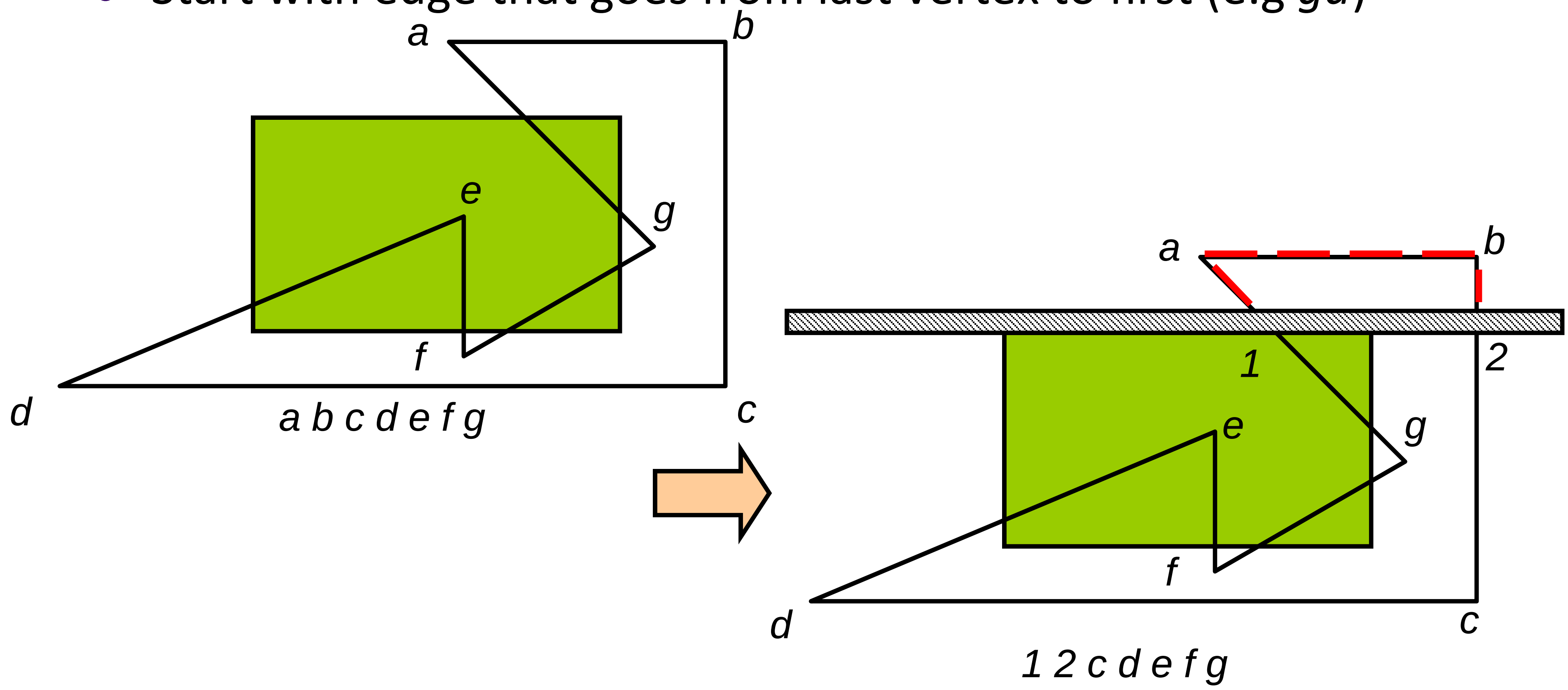
Case D: s outside, p inside:

Find intersection i,

output i and then p

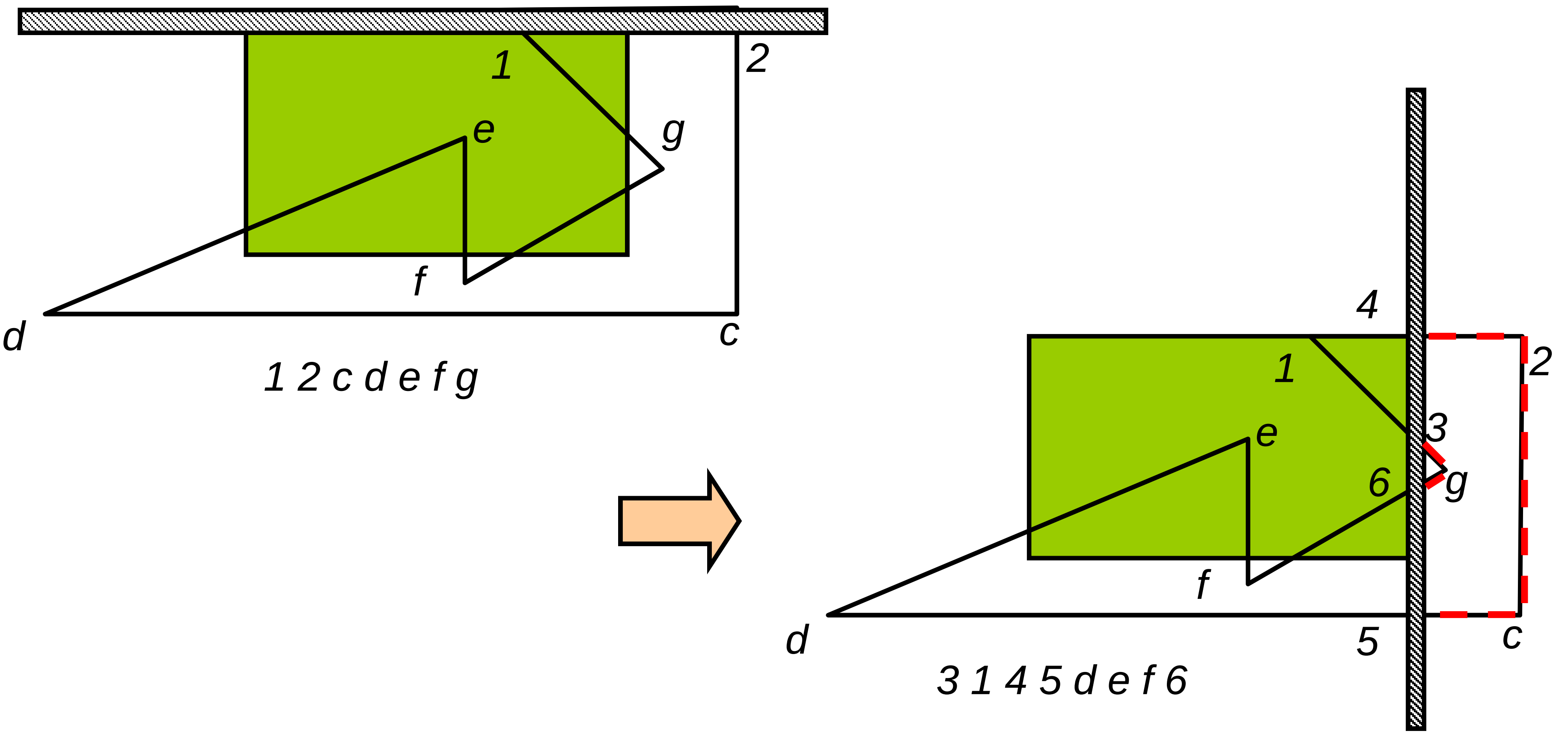
Sutherland-Hodgman Clipping

- Now, let's work through example
- Treat each edge of C as infinite plane to clip against
- Start with edge that goes from last vertex to first (e.g *ga*)



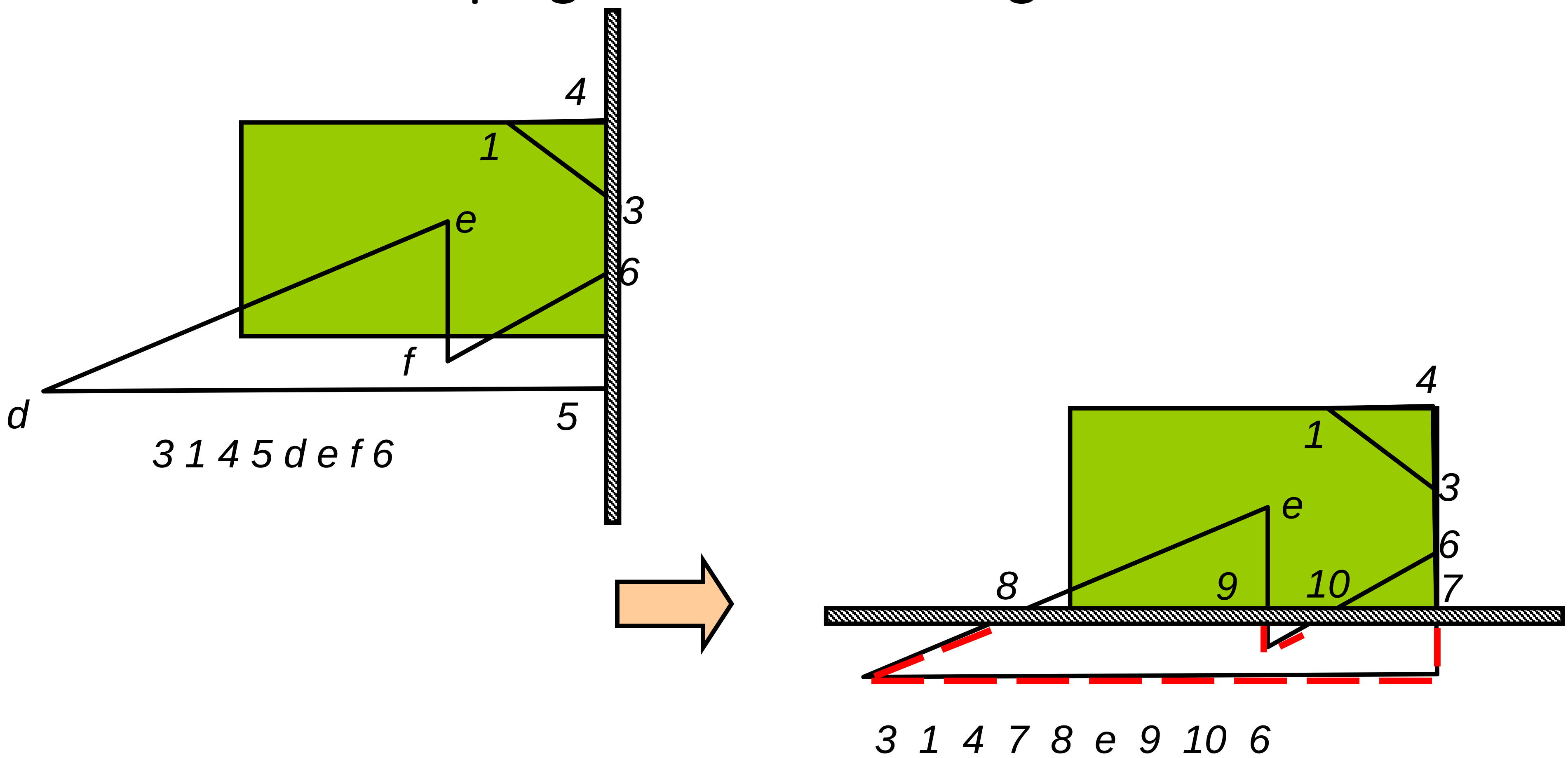
Sutherland-Hodgman Clipping

- Then chop against right edge



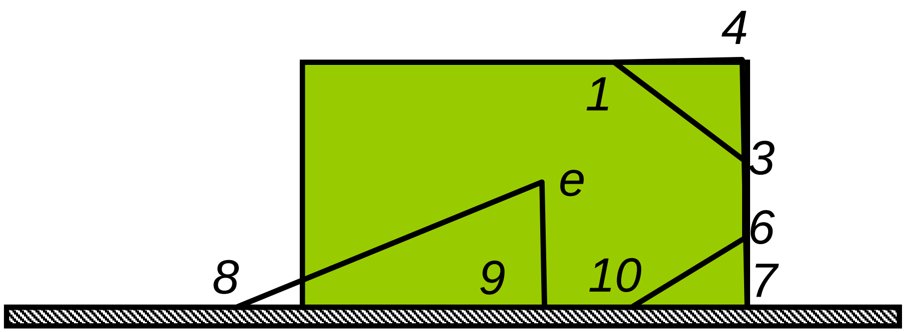
Sutherland-Hodgman Clipping

- Then chop against bottom edge

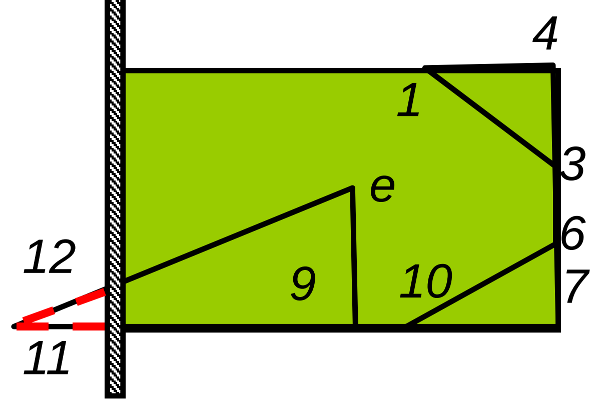


Sutherland-Hodgman Clipping

- Finally, clip against left edge



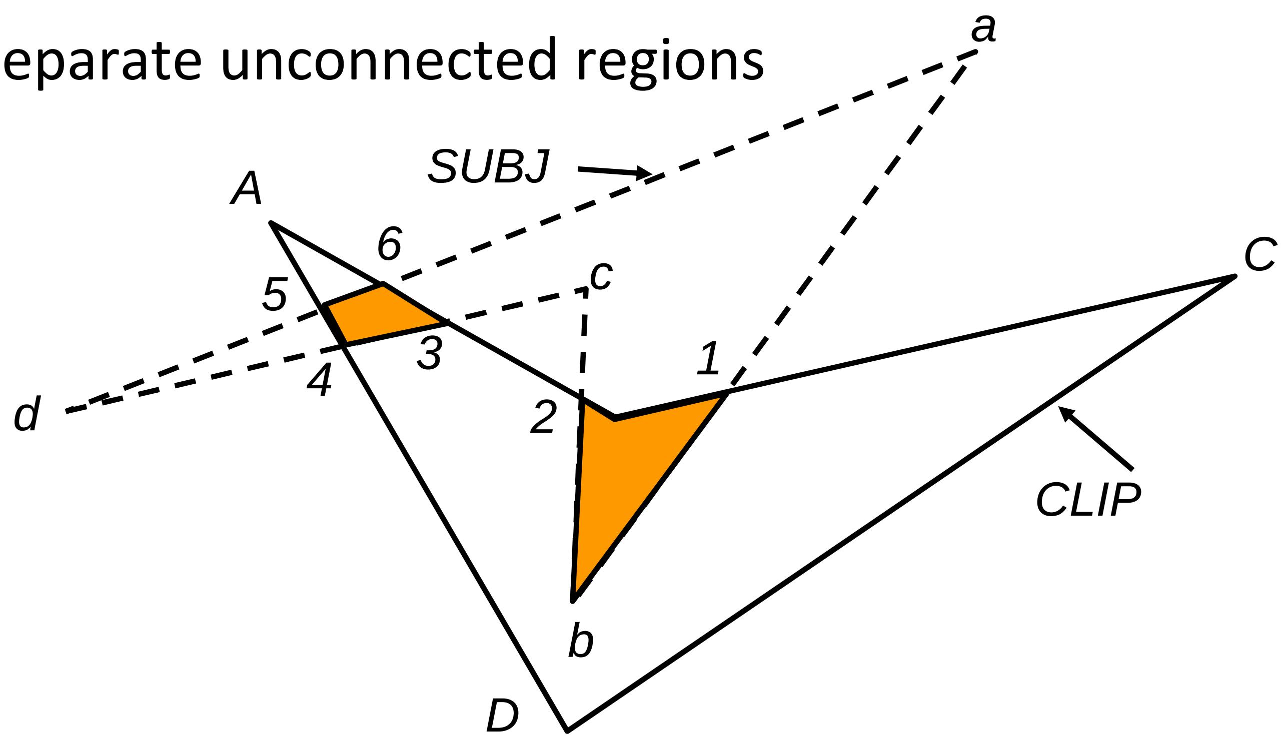
3 1 4 7 8 e 9 10 6



3 1 4 7 11 12 e 9 10 6

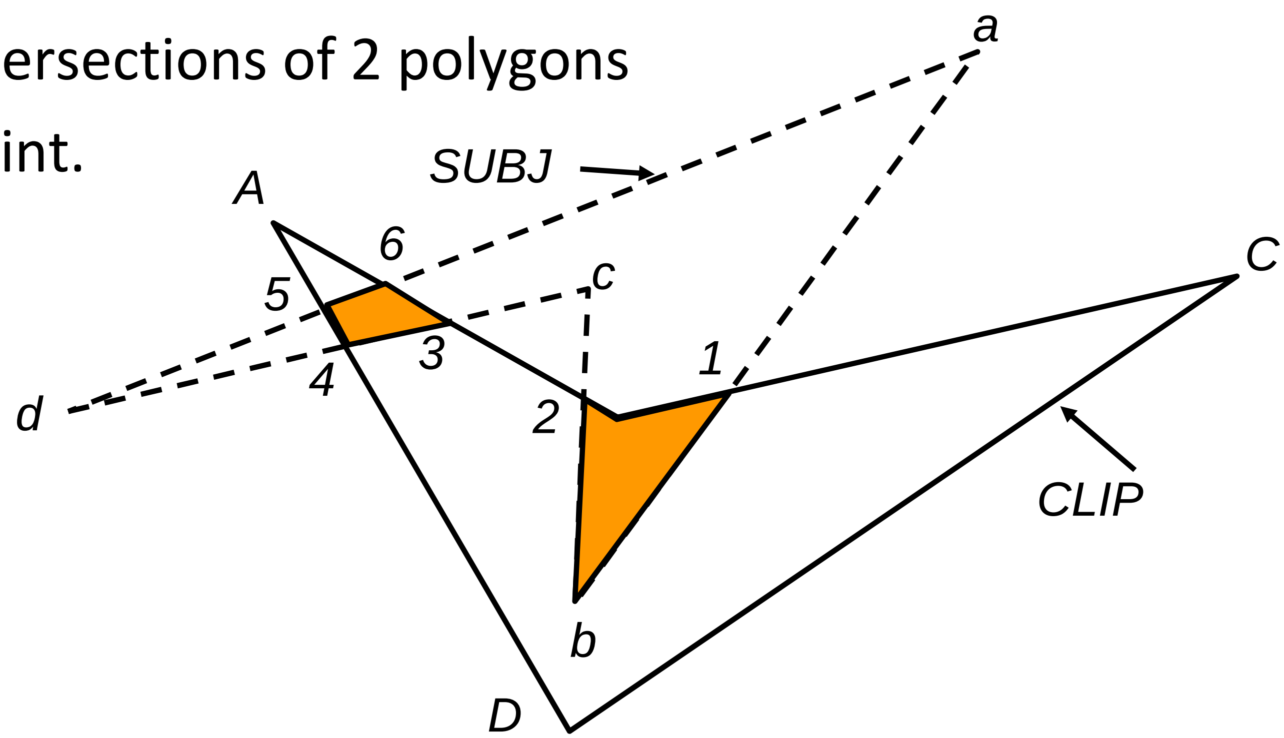
Weiler-Atherton Clipping Algorithm

- Sutherland-Hodgman required at least 1 convex polygon
- Weiler-Atherton can deal with 2 concave polygons
- Searches perimeter of SUBJ polygon searching for borders that enclose a clipped filled region
- Finds multiple separate unconnected regions



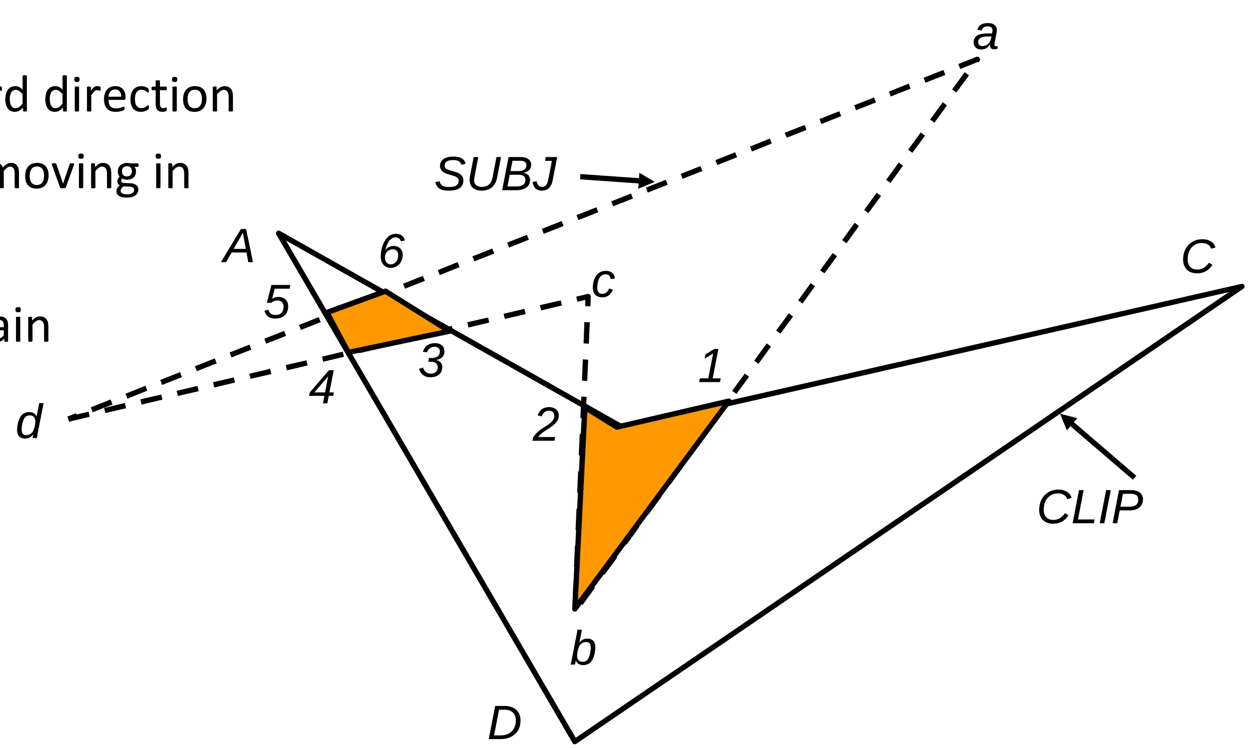
Weiler-Atherton Clipping Algorithm

- Follow detours along CLIP boundary whenever polygon edge crosses to outside of boundary
- Example: SUBJ = {a,b,c,d} CLIP = {A,B,C,D}
- Order: clockwise, interior to right
- First find all intersections of 2 polygons
- Example has 6 int.
- {1,2,3,4,5,6}



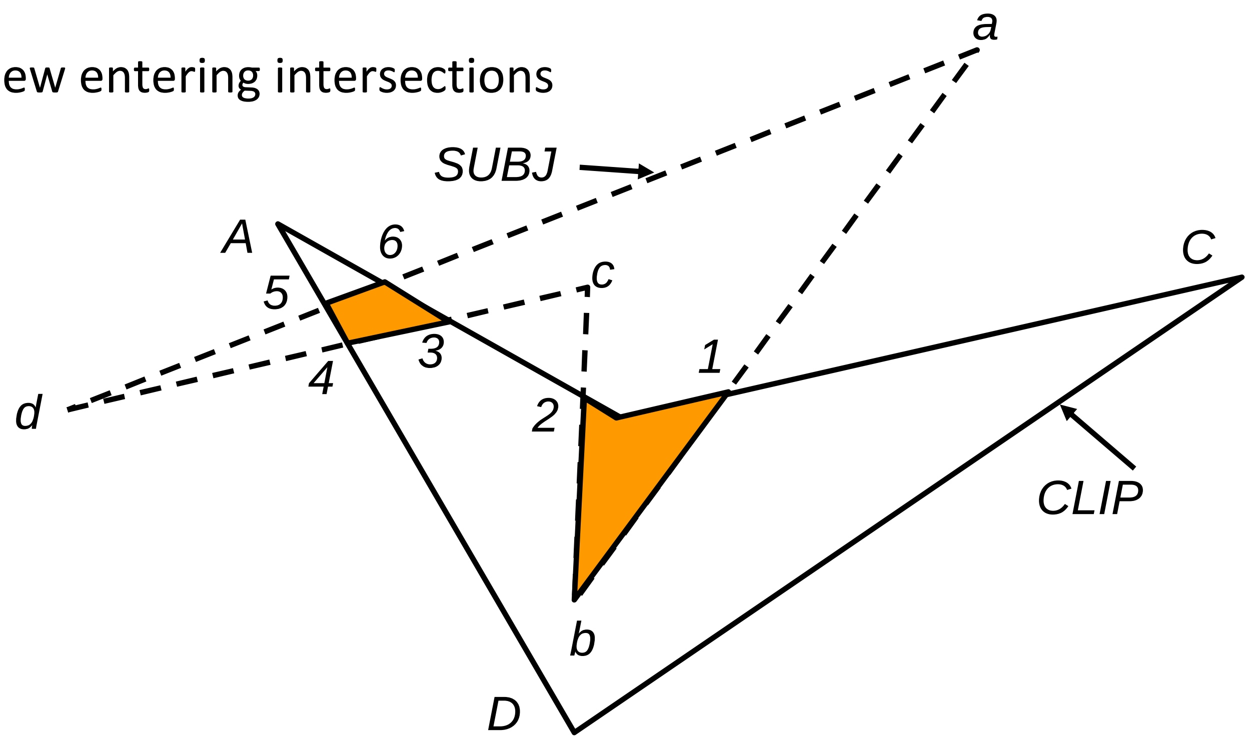
Weiler-Atherton Clipping Algorithm

- Start at *a*, traverse SUBJ in forward direction till first **entering intersection** (SUBJ moving outside-inside of CLIP) is found
- Record this intersection (1) to new vertex list
- Traverse along SUBJ till next intersection (2)
- Turn away from SUBJ at 2
- Now follow CLIP in forward direction
- Jump between polygons moving in forward direction till first intersection (1) is found again
- Yields: {1, b, 2}



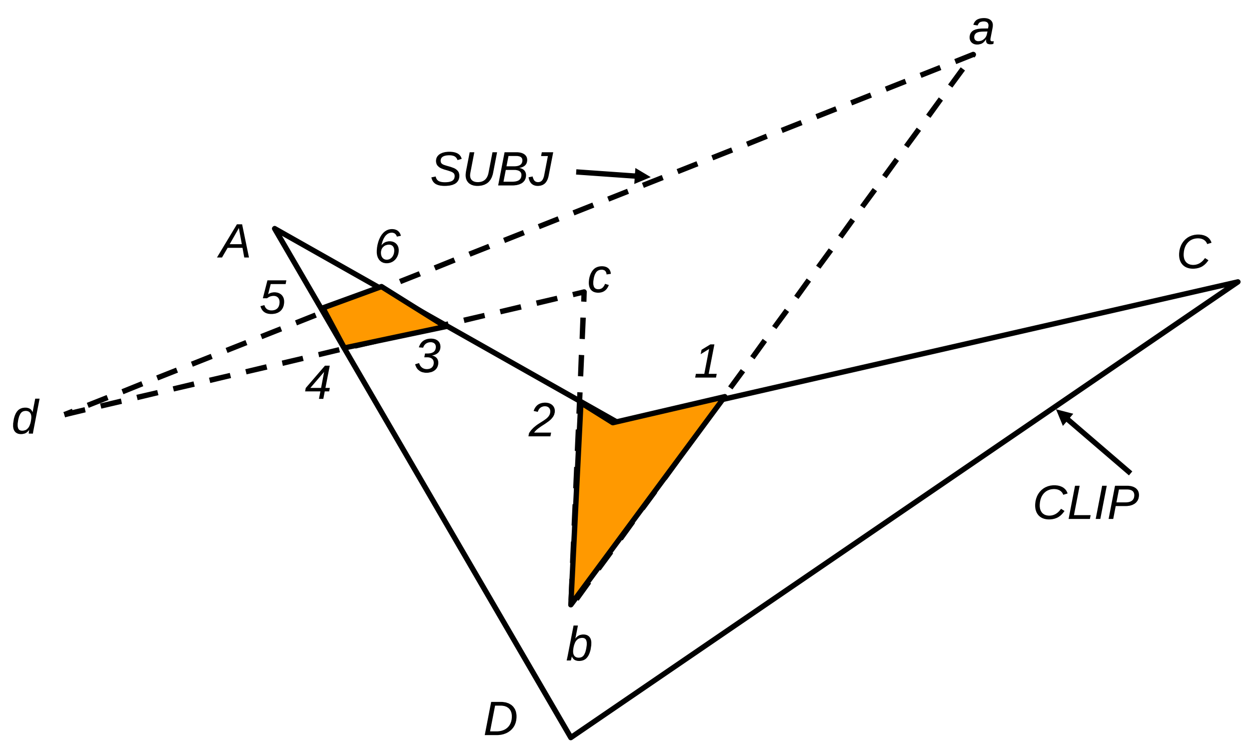
Weiler-Atherton Clipping Algorithm

- Start again, checking for next **entering intersection** of SUBJ
- Intersection (3) is found
- Repeat process
- Jump from SUBJ to CLIP at next intersection (4)
- Polygon {3,4,5,6} is found
- Further checks show no new entering intersections

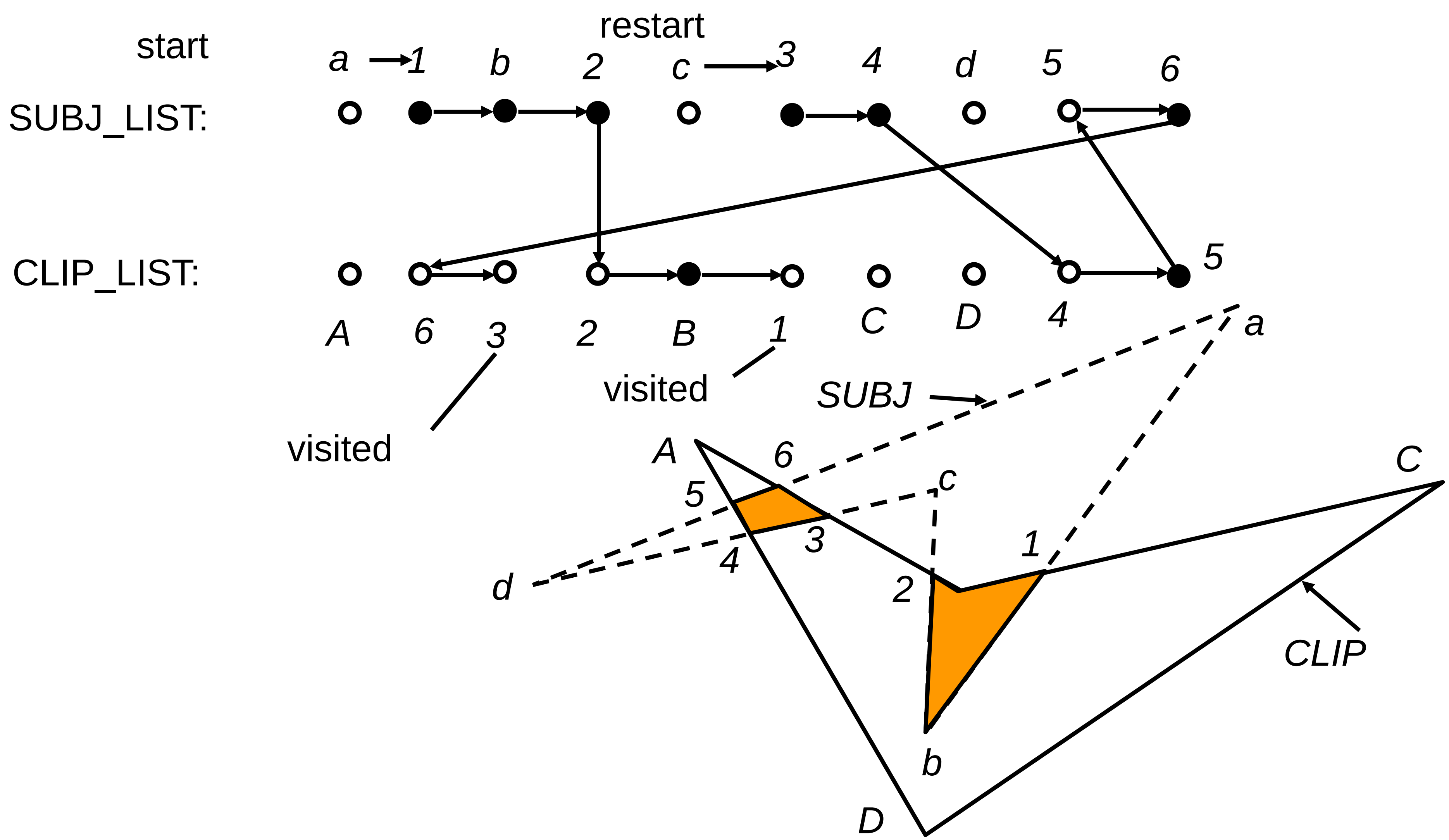


Weiler-Atherton Clipping Algorithm

- Can be implemented using 2 simple lists
- List all ordered vertices **and** intersections of SUBJ and CLIP
- SUBJ_LIST: a, 1, b, 2, c, 3, 4, d, 5, 6
- CLIP_LIST: A, 6, 3, 2, B, 1, C, D, 4, 5

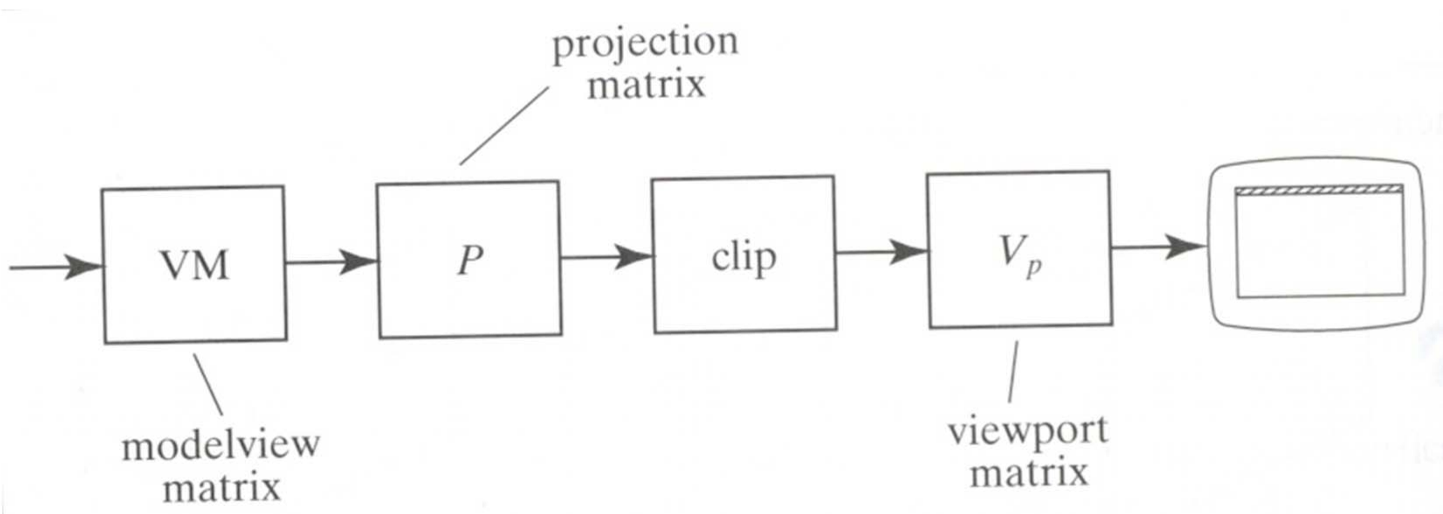


Weiler-Atherton Clipping Algorithm



Viewport Transformation

- After clipping, do viewport transformation
- We have used glViewport(x,y, wid, ht) before
- Use again here!!
- glViewport shifts x, y to screen coordinates
- Also maps pseudo-depth z from range [-1,1] to [0,1]
- Pseudo-depth stored in depth buffer, used for Depth testing (Will discuss later)

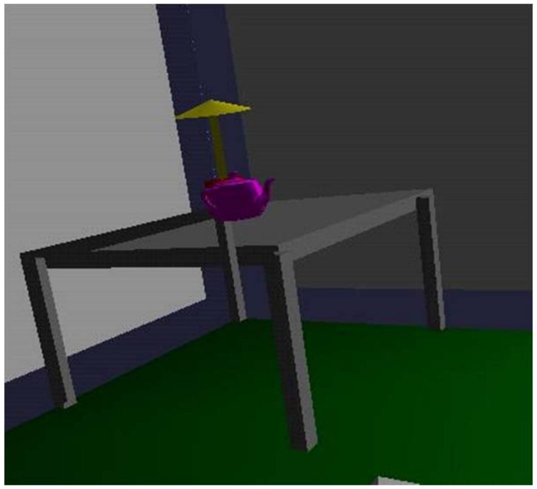


Hidden surface Removal

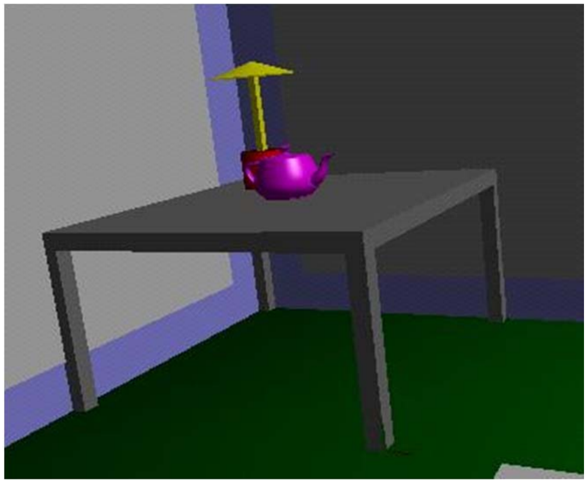
- Drawing polygonal faces on screen consumes CPU cycles
- We cannot see every surface in scene
- To save time, draw only surfaces we see
- Surfaces we cannot see and their elimination methods:
  - **Occluded surfaces:** hidden surface removal (visibility)
  - **Back faces:** back face culling
  - **Faces outside view volume:** viewing frustrum culling
- Definitions:
  - **Object space techniques:** applied before vertices are mapped to pixels
  - **Image space techniques:** applied after vertices have been rasterized

Visibility (hidden surface removal)

- A correct rendering requires correct visibility calculations
- Correct visibility – when multiple opaque polygons cover the same screen space, only the closest one is visible (remove the other hidden surfaces)



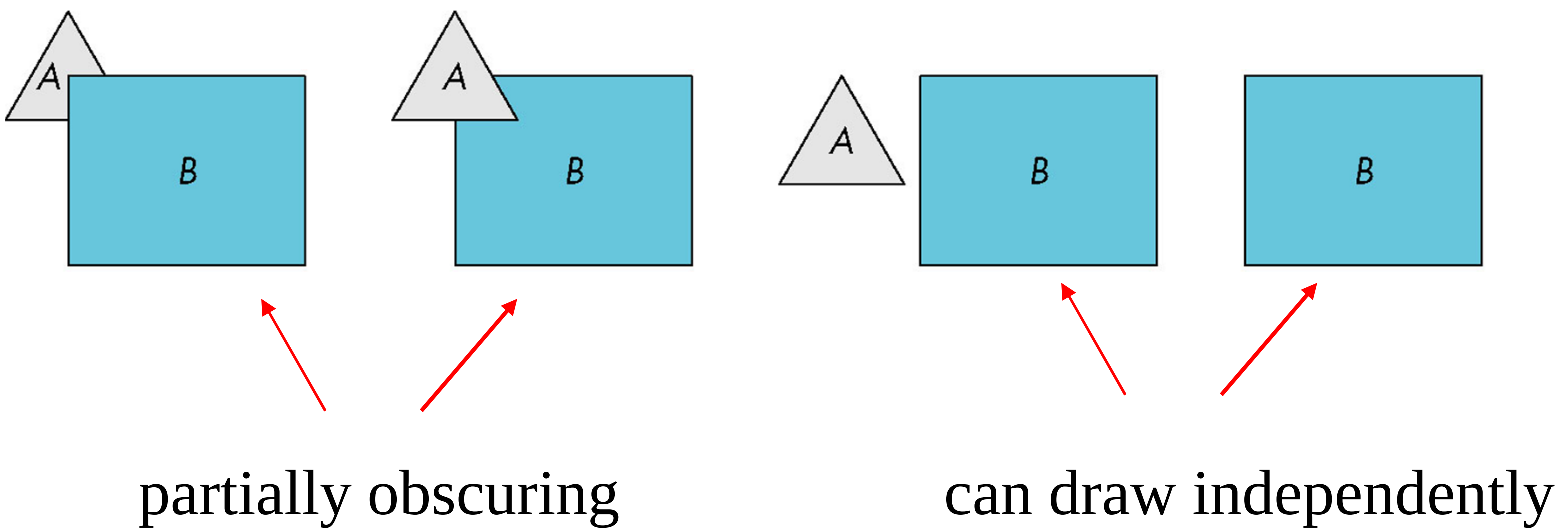
wrong visibility



Correct visibility

Hidden Surface Removal

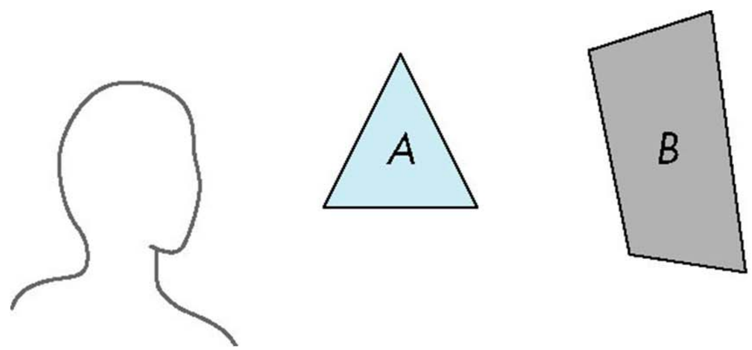
- Object-space approach: use pairwise testing between polygons (objects)



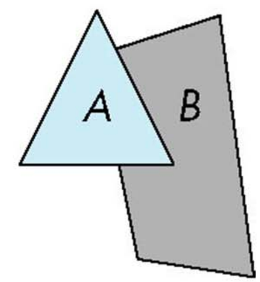
- Worst case complexity $O(n^2)$ for n polygons

Painter's Algorithm

- Render polygons a back to front order so that polygons behind others are simply painted over



B behind A as seen by viewer



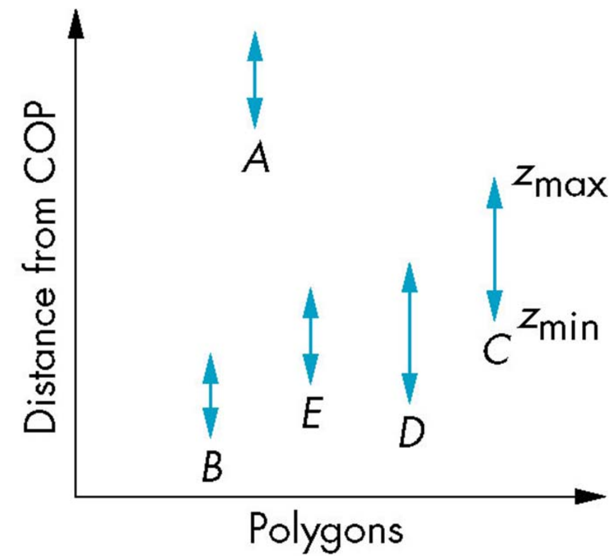
Fill B then A

Depth Sort

- Requires ordering of polygons first
  - O(n log n) calculation for ordering
  - Not every polygon is either in front or behind all other polygons

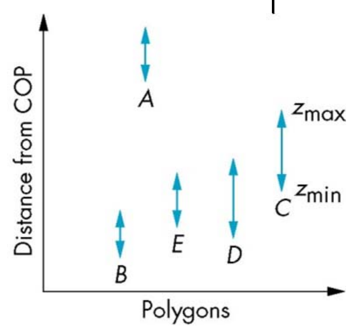
- Order polygons and deal with easy cases first, harder later

Polygons sorted by distance from COP

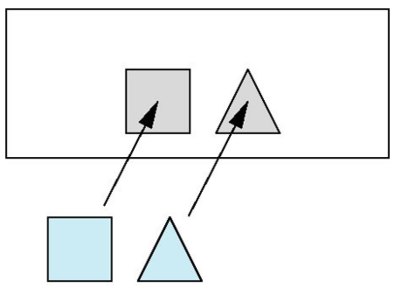


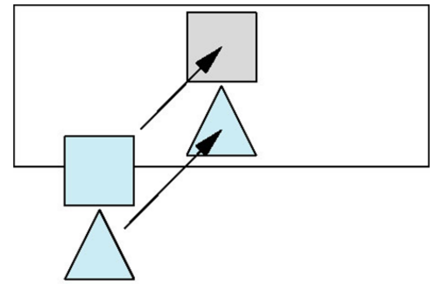
Easy Cases

- A lies behind all other polygons
  - Can render

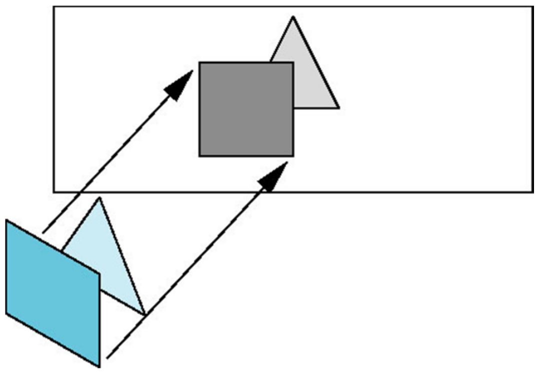


- Polygons overlap in z but not in either x or y
  - Can render independently

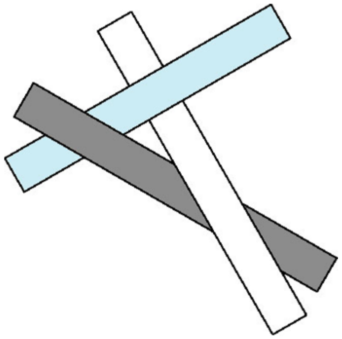




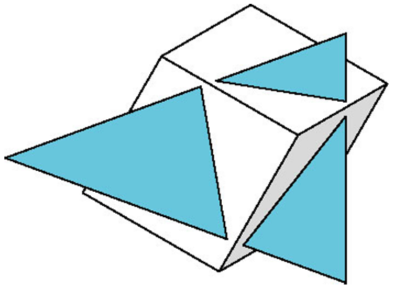
Hard Cases



Overlap in all directions but can one is fully on one side of the other



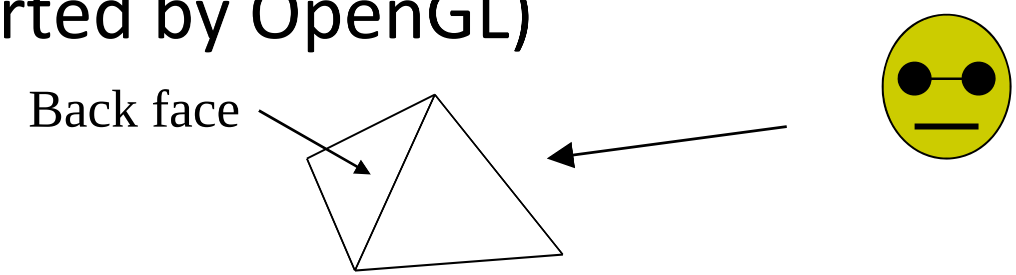
cyclic overlap



penetration

Back Face Culling

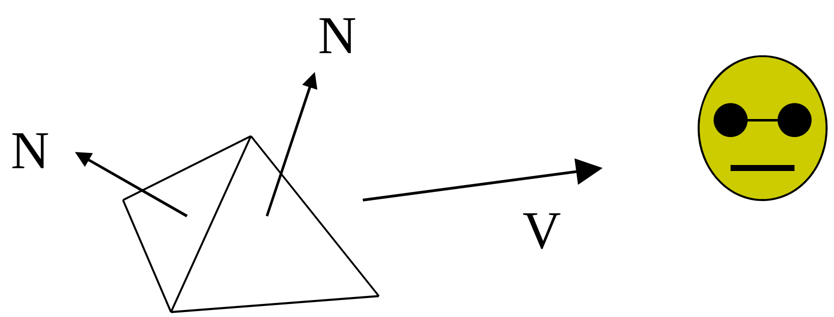
- Back faces: faces of opaque object which are "pointing away" from viewer
- Back face culling – remove back faces (supported by OpenGL)



- How to detect back faces?

Back Face Culling

- If we find backface, do not draw, save rendering resources
- There must be other forward face(s) closer to eye
- F is face of object we want to test if backface
- P is a point on F
- Form view vector, V as (eye – P)
- N is normal to face F



Backface test: F is backface if N.V < 0 why??

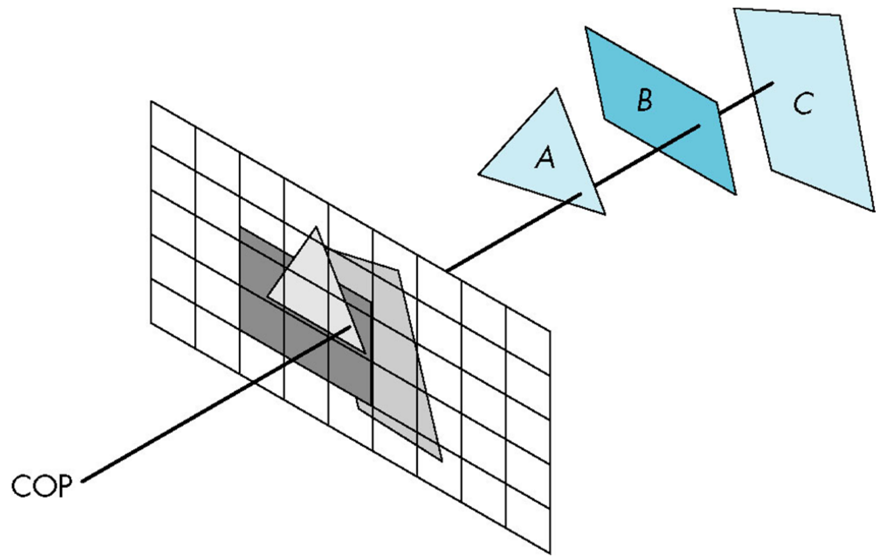
Back Face Culling: Draw mesh front faces

```
void drawFrontFaces( )
{
    for(int f = 0;f < numFaces; f++)
    {
        if(isBackFace(f, ….) continue;
        glDrawArrays(GL_POLYGON, 0, N);
    }
```

Note: In OpenGL we can simply enable culling
but may not work correctly if we have nonconvex objects

Image Space Approach

- Look at each projector (nm for an n x m frame buffer) and find closest of k polygons
- Complexity O(nmk)
- Ray tracing
- z-buffer



OpenGL HSR Commands

- Primarily three commands to do HSR
- **`glutInitDisplayMode(GLUT_DEPTH | GLUT_RGB)`** instructs openGL to create depth buffer
- **`glEnable(GL_DEPTH_TEST)`** enables depth testing
- **`glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`** initializes the depth buffer every time we draw a new picture

OpenGL - Image Space Approach

- Determine which of the n objects is visible to each pixel on the image plane
- Paint pixel with color of **closest** object

```
for (each pixel in the image) {
    determine the object closest to the pixel
    draw the pixel using the object's color
}
```

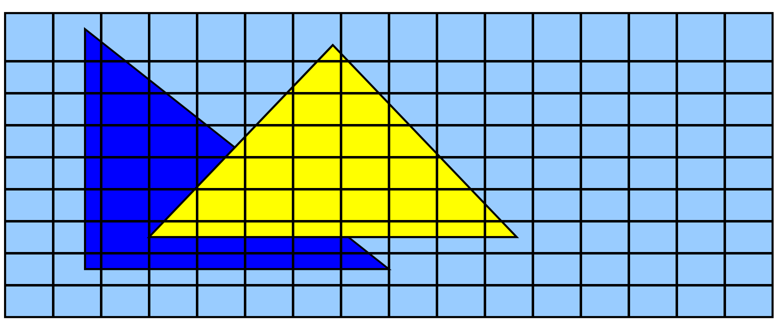


Image Space Approach – Z-buffer

- Method used in most of graphics hardware (and thus OpenGL): Z-buffer (or depth buffer) algorithm
- Requires lots of memory
- Recall: after projection transformation, in viewport transformation
  - x,y used to draw screen image, mapped to viewport
  - z component is mapped to pseudo-depth with range [0,1]
- Objects/polygons are made up of vertices
- Hence, we know depth z at polygon vertices

Image Space Approach – Z-buffer

- Basic Z-buffer idea:
  - rasterize every input polygon
  - For every pixel in the polygon interior, calculate its corresponding z value (by interpolation)
  - Track depth values of closest polygon (smallest z) so far
  - Paint the pixel with the color of the polygon whose z value is the closest to the eye.

Z (depth) buffer algorithm

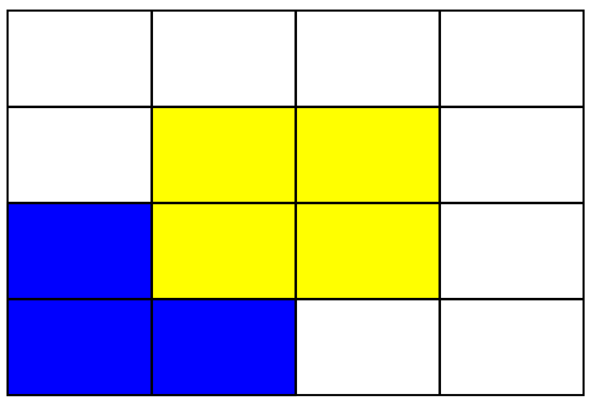
- How to choose the polygon that has the closet Z for a given pixel?
- Example: eye at z = 0, farther objects have increasingly positive values, between 0 and 1
  1. Initialize (clear) every pixel in the z buffer to 1.0
  2. Track polygon z's.
  3. As we rasterize polygons, check to see if polygon's z through this pixel is less than current minimum z through this pixel
  4. Run the following loop:

Z (depth) Buffer Algorithm

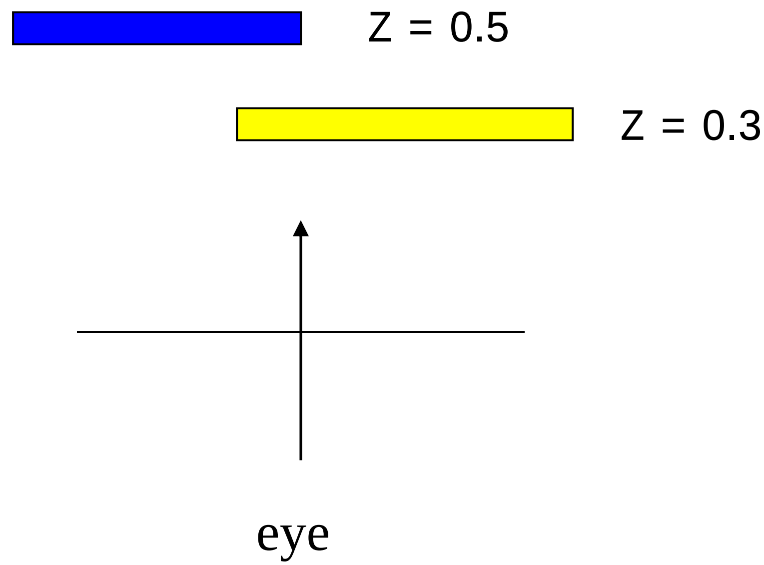
```
For  each polygon  {
   for each pixel (x,y) inside the polygon projection area  {
         if  (z_polygon_pixel(x,y) < depth_buffer(x,y) ) {
                depth_buffer(x,y) = z_polygon_pixel(x,y);
                color_buffer(x,y) = polygon color at (x,y)
          }
      }
  }
```

Note: know depths at vertices. Interpolate for interior z_polygon_pixel(x, y) depths

Z buffer example



Correct Final image



Top View

Z buffer example

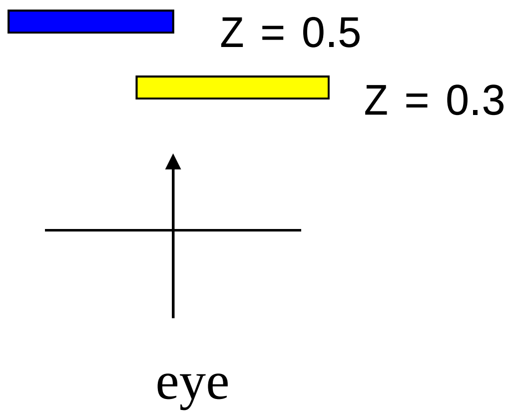
Step 1: Initialize the depth buffer

1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0

Z buffer example

Step 2: Draw the blue polygon (assuming the OpenGL program draws blue polyon first – the order does not affect the final result any way).

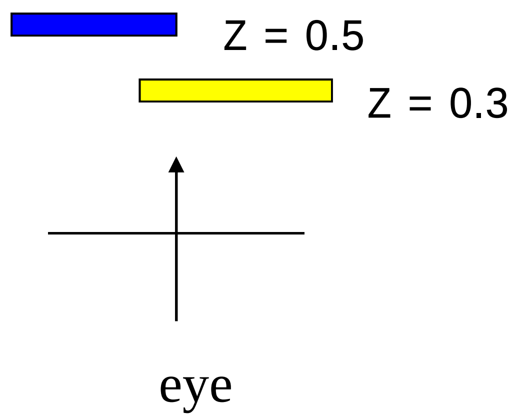
1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
0.5	0.5	1.0	1.0
0.5	0.5	1.0	1.0



Z buffer example

Step 3: Draw the yellow polygon

1.0	1.0	1.0	1.0
1.0	0.3	0.3	1.0
0.5	0.3	0.3	1.0
0.5	0.5	1.0	1.0



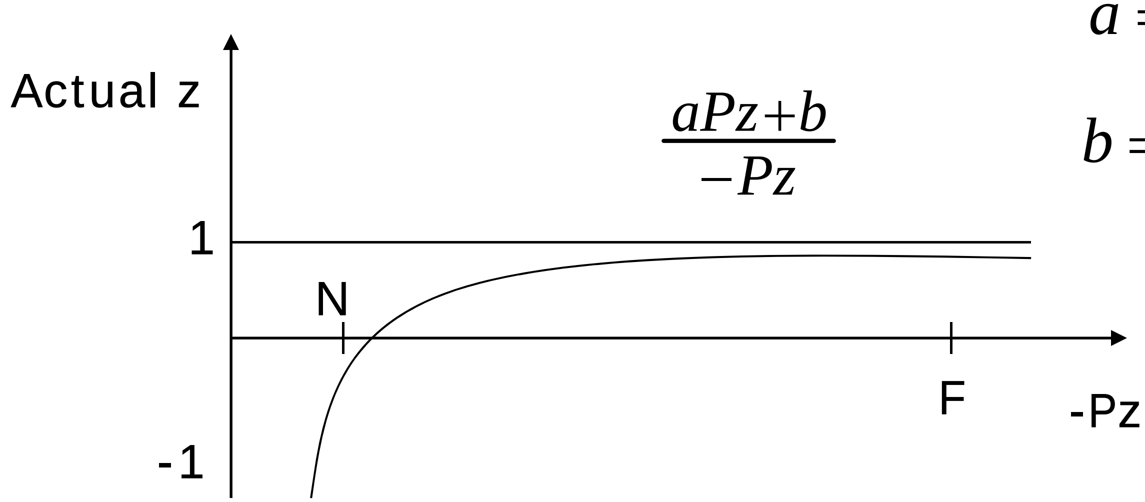
z-buffer drawback: wastes resources by rendering a face and then drawing over it

Z-Buffer Depth Compression

- Recall that we chose parameters a and b to map z from range [near, far] to **pseudodepth** range[0,1]
- This mapping is almost linear close to eye
- Non-linear further from eye, approaches asymptote
- Also limited number of bits
- Thus, two z values close to far plane may map to same pseudodepth: ***Errors!!***

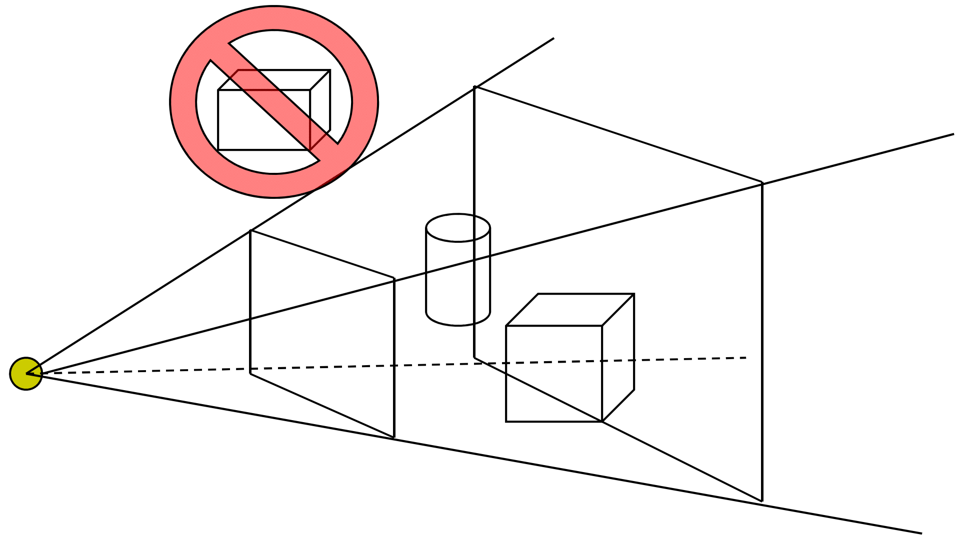
$$a = -\frac{F+N}{F-N}$$

$$b = -\frac{-2FN}{F-N}$$



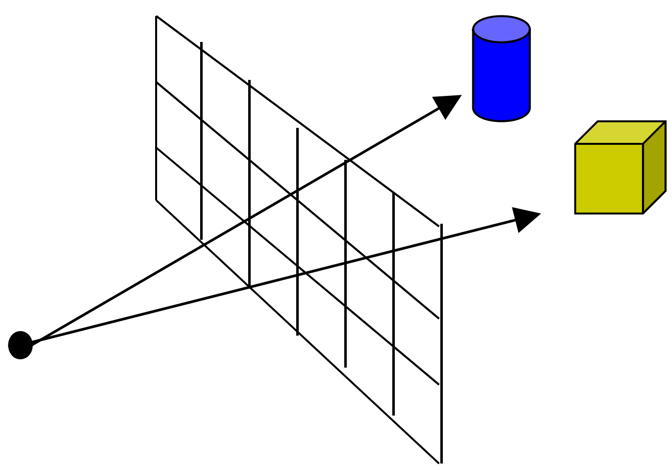
View-Frustum Culling

- Remove objects that are outside the viewing frustum
- Done by 3D clipping algorithm (e.g. Liang-Barsky)



Ray Tracing

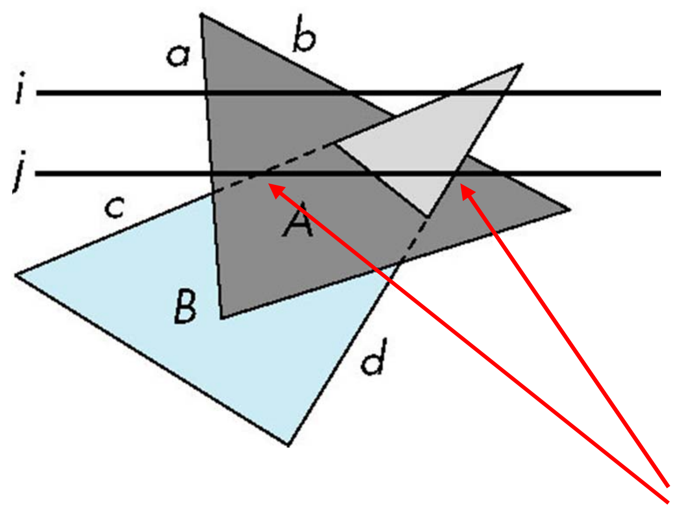
- Ray tracing is another example of image space method
- Ray tracing: Cast a ray from eye through each pixel to the world.
- Question: what does eye see in direction looking through a given pixel?



Will discuss more later

Scan-Line Algorithm

- Can combine shading and hsr through scan line algorithm

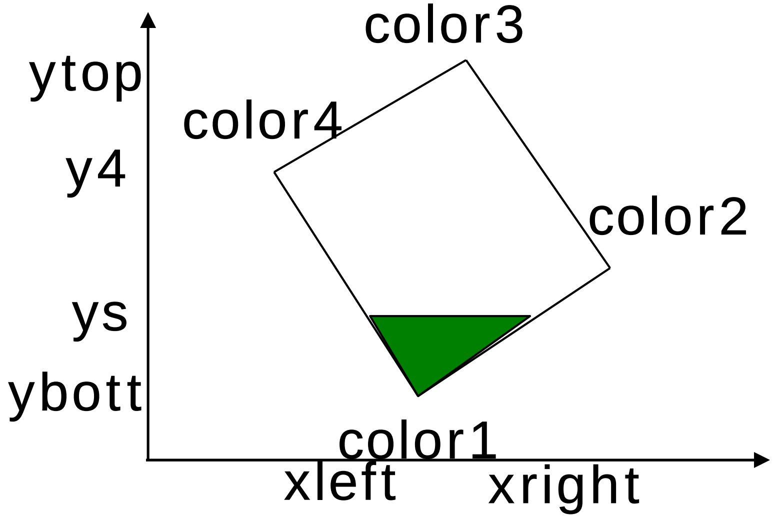


scan line i: no need for depth information, can only be in no or one polygon

scan line j: need depth information only when in more than one polygon

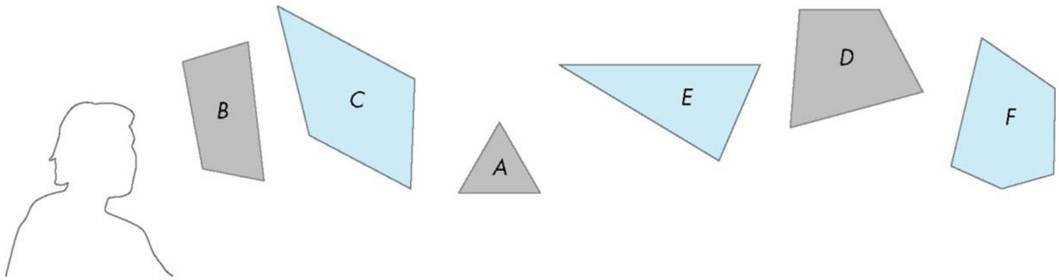
Combined z-buffer and Gouraud Shading (Hill)

```
for(int y = ybott; y <= ytop; y++)  // for each scan line
{
    for(each polygon){
    find xleft and xright
    find dleft and dright, and dinc
    find colorleft and colorright, and colorinc
    for(int x = xleft, c = colorleft, d = dleft; x <= xright;
                                          x++, c+= colorinc, d+= dinc)
    if(d < d[x][y])
    {
      put c into the pixel at (x, y)
      d[x][y] = d; // update closest depth
    }}
}
```

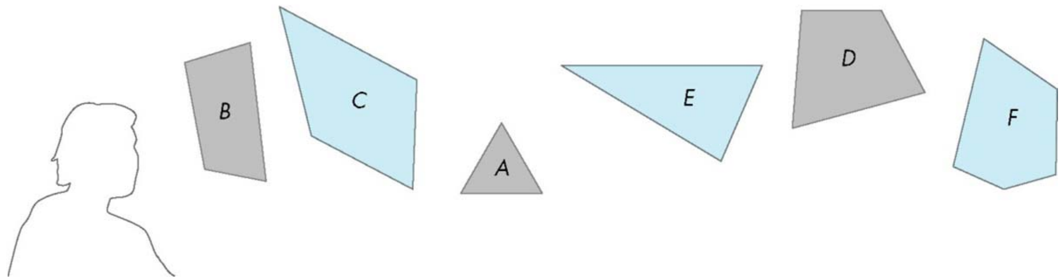


Visibility Testing

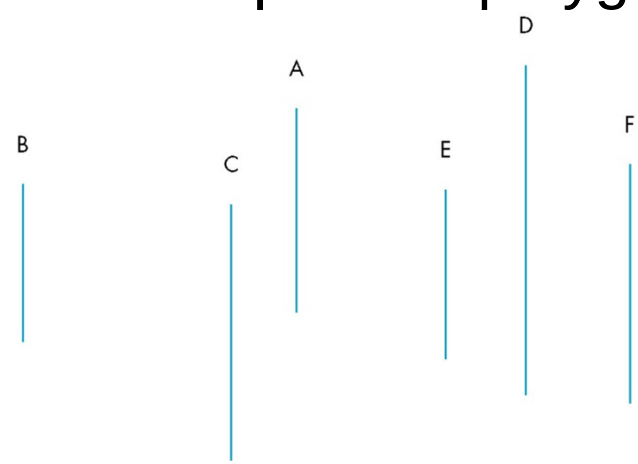
- In many realtime applications, such as games, we want to eliminate as many objects as possible within the application
  - Reduce burden on pipeline
  - Reduce traffic on bus
- Partition space with Binary Spatial Partition (BSP) Tree



Simple Example



consider 6 parallel polygons

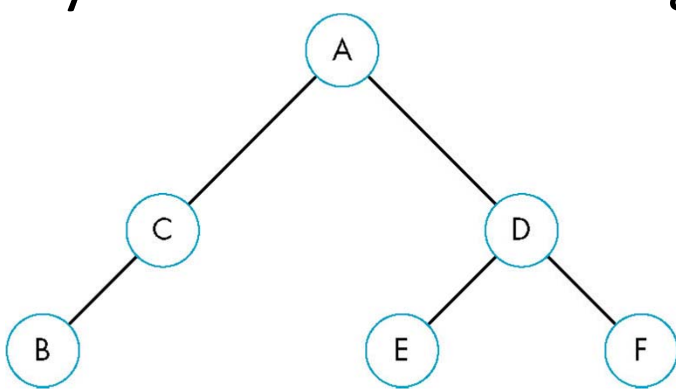


top view

The plane of A separates B and C from D, E and F

BSP Tree

- Can continue recursively
  - Plane of C separates B from A
  - Plane of D separates E and F
- Can put this information in a BSP tree
  - Use for visibility and occlusion testing



References

- Angel and Shreiner, Interactive Computer Graphics, 6th edition
- Hill and Kelley, Computer Graphics using OpenGL, 3rd edition