Clipping,
Hidden Surface Elimination

Course web page:

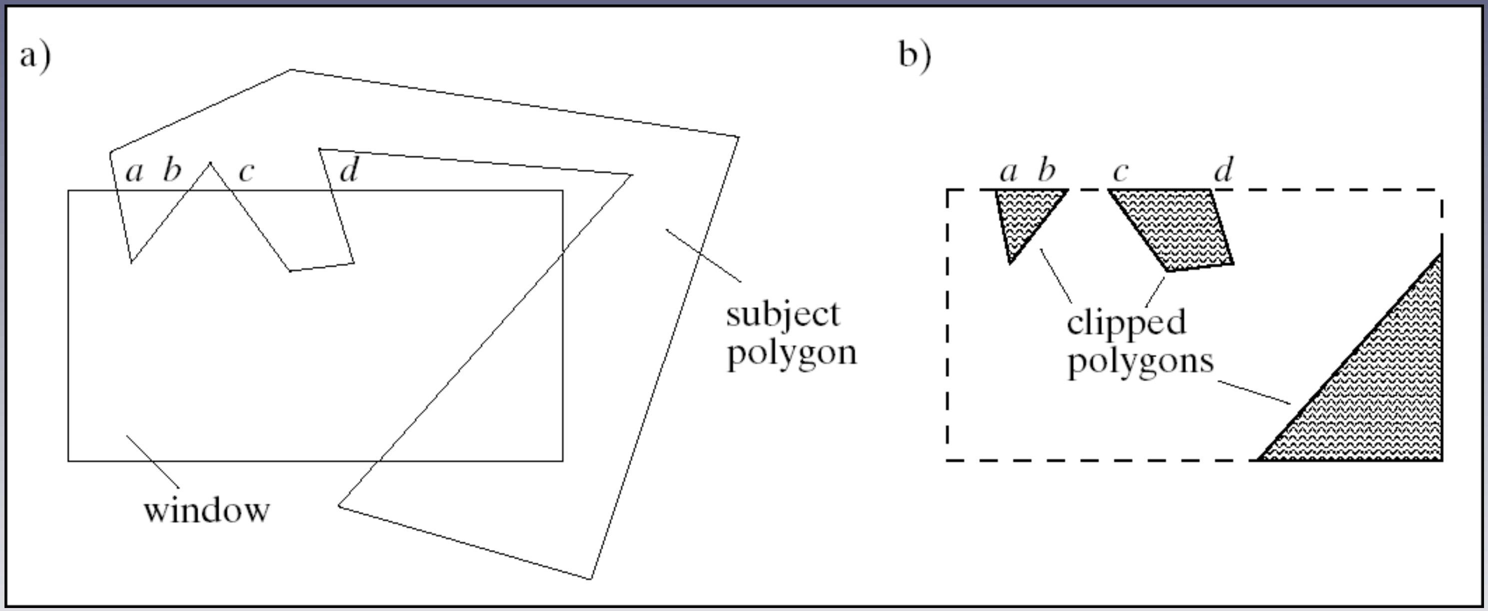
http://goo.gl/EB3aA

University of Delaware

Outline

- Clipping
  - Line clipping (from last time)
  - Polygon clipping
- Hidden surface elimination
  - Backface culling
  - BSP trees
  - Z-buffering

Polygon clipping

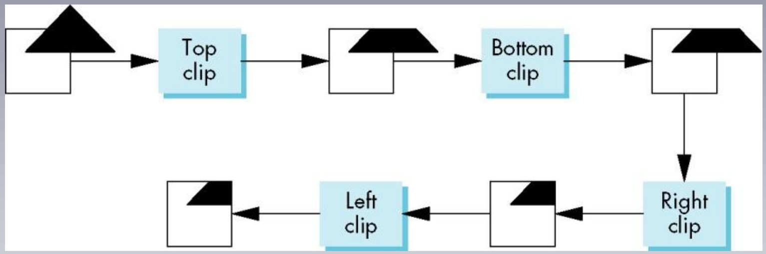


from Hill

- Simply clipping lines independently loses connectivity information
- Must clip in order and interpolate missing edges

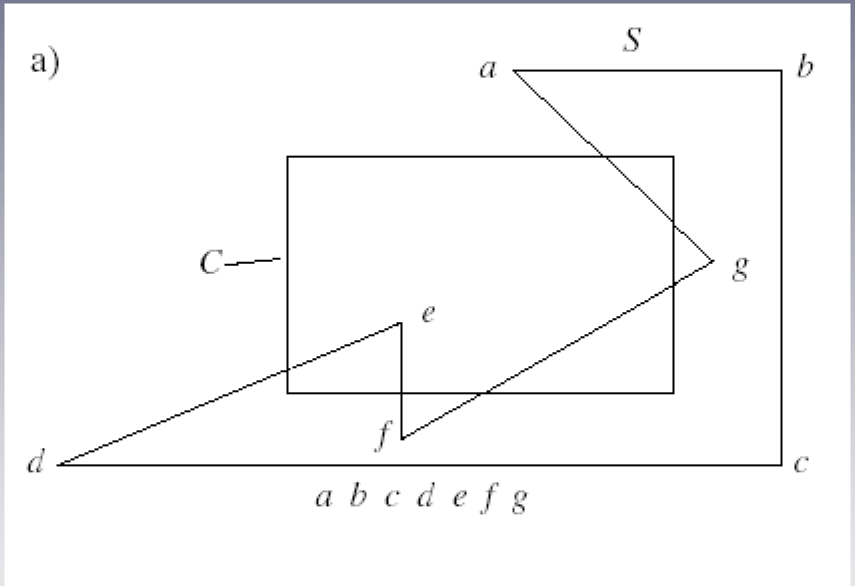
Sutherland-Hodgman 2-D polygon clipping

- Algorithm for clipping polygon *S* (convex or not) against a convex clip polygon *C*
- Basic approach: Clip *S* against each side ($\mathbf{c}_i$, $\mathbf{c}_j$) of *C* in sequence
  - Traverse vertex list of *S* & build new clipped vertex list
  - For each old edge ($\mathbf{s}_i$, $\mathbf{s}_j$) of *S*:
    - If both inside ($\mathbf{c}_i$, $\mathbf{c}_j$): Output new $\mathbf{s}_j$
    - If both outside: Output nothing
    - If $\mathbf{s}_i$ inside, $\mathbf{s}_j$ outside: Output intersection of ($\mathbf{s}_i$, $\mathbf{s}_j$) with ($\mathbf{c}_i$, $\mathbf{c}_j$)
    - If $\mathbf{s}_i$ outside, $\mathbf{s}_j$ inside: Output intersection of ($\mathbf{s}_i$, $\mathbf{s}_j$) with ($\mathbf{c}_i$, $\mathbf{c}_j$), then $\mathbf{s}_j$



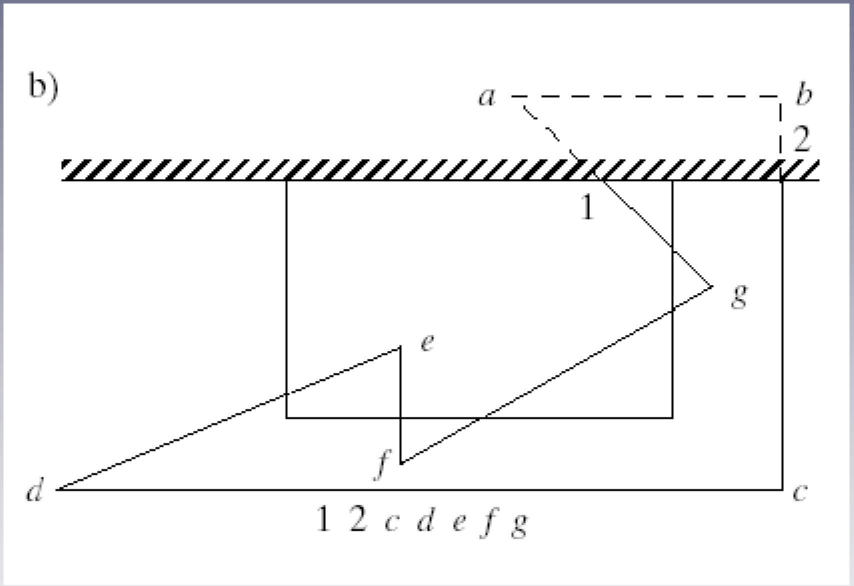
from E. Angel

Example: Sutherland-Hodgman



from Hill

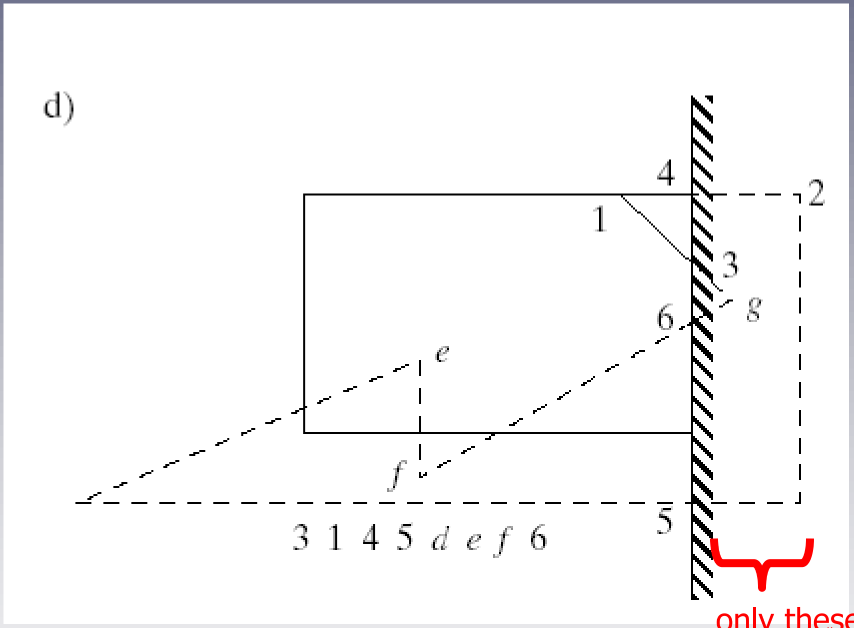
Example: Sutherland-Hodgman



from Hill

1. If both inside ($\mathbf{c}_i$, $\mathbf{c}_j$): Output new $\mathbf{s}_j$
2. If both outside: Output nothing
3. If $\mathbf{s}_i$ inside, $\mathbf{s}_j$ outside: Output intersection of ($\mathbf{s}_i$, $\mathbf{s}_j$) with ($\mathbf{c}_i$, $\mathbf{c}_j$)
4. If $\mathbf{s}_i$ outside, $\mathbf{s}_j$ inside: Output intersection of ($\mathbf{s}_i$, $\mathbf{s}_j$) with ($\mathbf{c}_i$, $\mathbf{c}_j$), then $\mathbf{s}_j$

Example: Sutherland-Hodgman

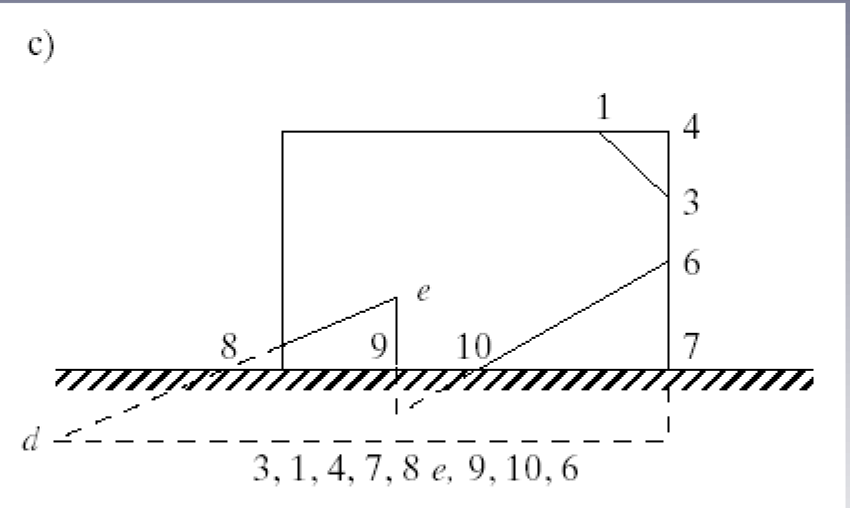


only these lines should be dashed

from Hill

1. If both inside ($\mathbf{c}_i$, $\mathbf{c}_j$): Output new $\mathbf{s}_j$
2. If both outside: Output nothing
3. If $\mathbf{s}_i$ inside, $\mathbf{s}_j$ outside: Output intersection of ($\mathbf{s}_i$, $\mathbf{s}_j$) with ($\mathbf{c}_i$, $\mathbf{c}_j$)
4. If $\mathbf{s}_i$ outside, $\mathbf{s}_j$ inside: Output intersection of ($\mathbf{s}_i$, $\mathbf{s}_j$) with ($\mathbf{c}_i$, $\mathbf{c}_j$), then $\mathbf{s}_j$

Example: Sutherland-Hodgman

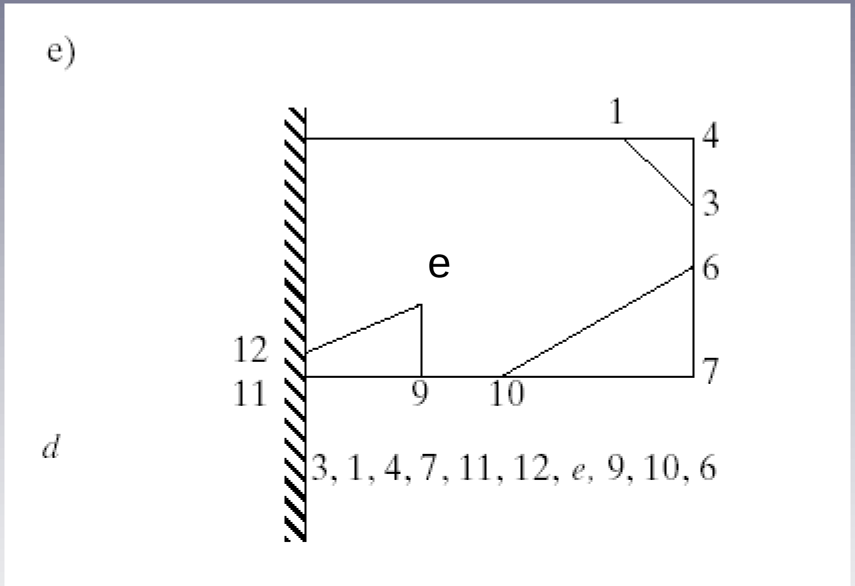


from Hill

1. If both inside $(\mathbf{c}_i, \mathbf{c}_j)$: Output new $\mathbf{s}_j$
2. If both outside: Output nothing
3. If $\mathbf{s}_i$ inside, $\mathbf{s}_j$ outside: Output intersection of $(\mathbf{s}_i, \mathbf{s}_j)$ with $(\mathbf{c}_i, \mathbf{c}_j)$
4. If $\mathbf{s}_i$ outside, $\mathbf{s}_j$ inside: Output intersection of $(\mathbf{s}_i, \mathbf{s}_j)$ with $(\mathbf{c}_i, \mathbf{c}_j)$, then $\mathbf{s}_j$

Example: Sutherland-Hodgman

- Note extraneous line segments introduced at (3, 6) and (9, 10)—these can be removed with post-processing

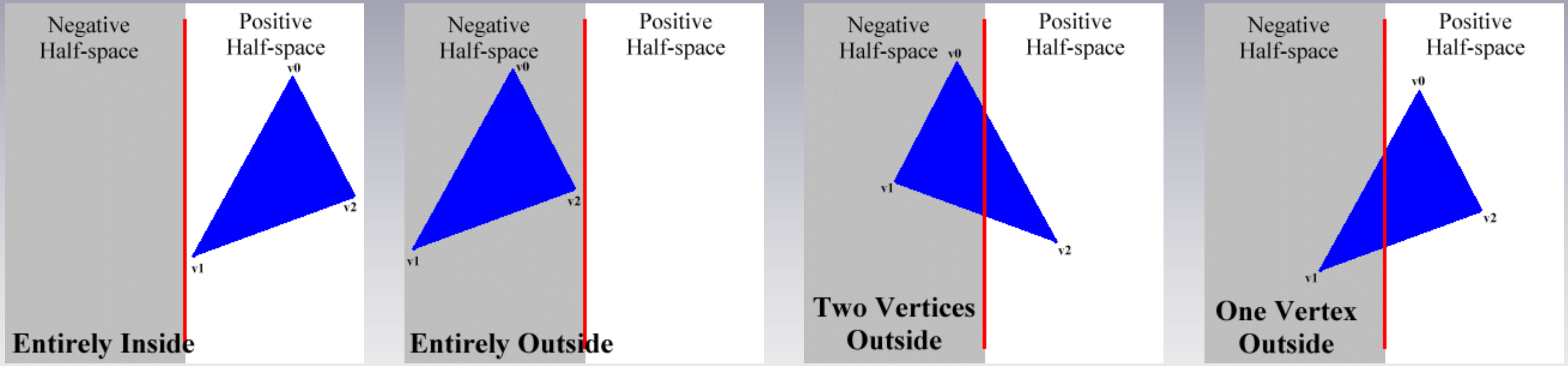


from Hill

1. If both inside ($\mathbf{c}_i$, $\mathbf{c}_j$): Output new $\mathbf{s}_j$
2. If both outside: Output nothing
3. If $\mathbf{s}_i$ inside, $\mathbf{s}_j$ outside: Output intersection of ($\mathbf{s}_i$, $\mathbf{s}_j$) with ($\mathbf{c}_i$, $\mathbf{c}_j$)
4. If $\mathbf{s}_i$ outside, $\mathbf{s}_j$ inside: Output intersection of ($\mathbf{s}_i$, $\mathbf{s}_j$) with ($\mathbf{c}_i$, $\mathbf{c}_j$), then $\mathbf{s}_j$

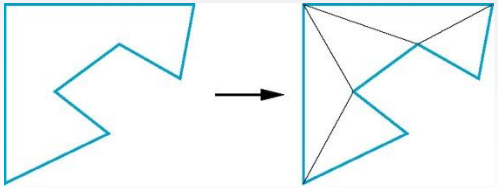
Sutherland-Hodgman: *S* is Triangle

- Simple case: *S* is a triangle, *C* a rectangle: S-H's general procedure for stepping through polygon vertices reduces to four cases for the whole triangle:



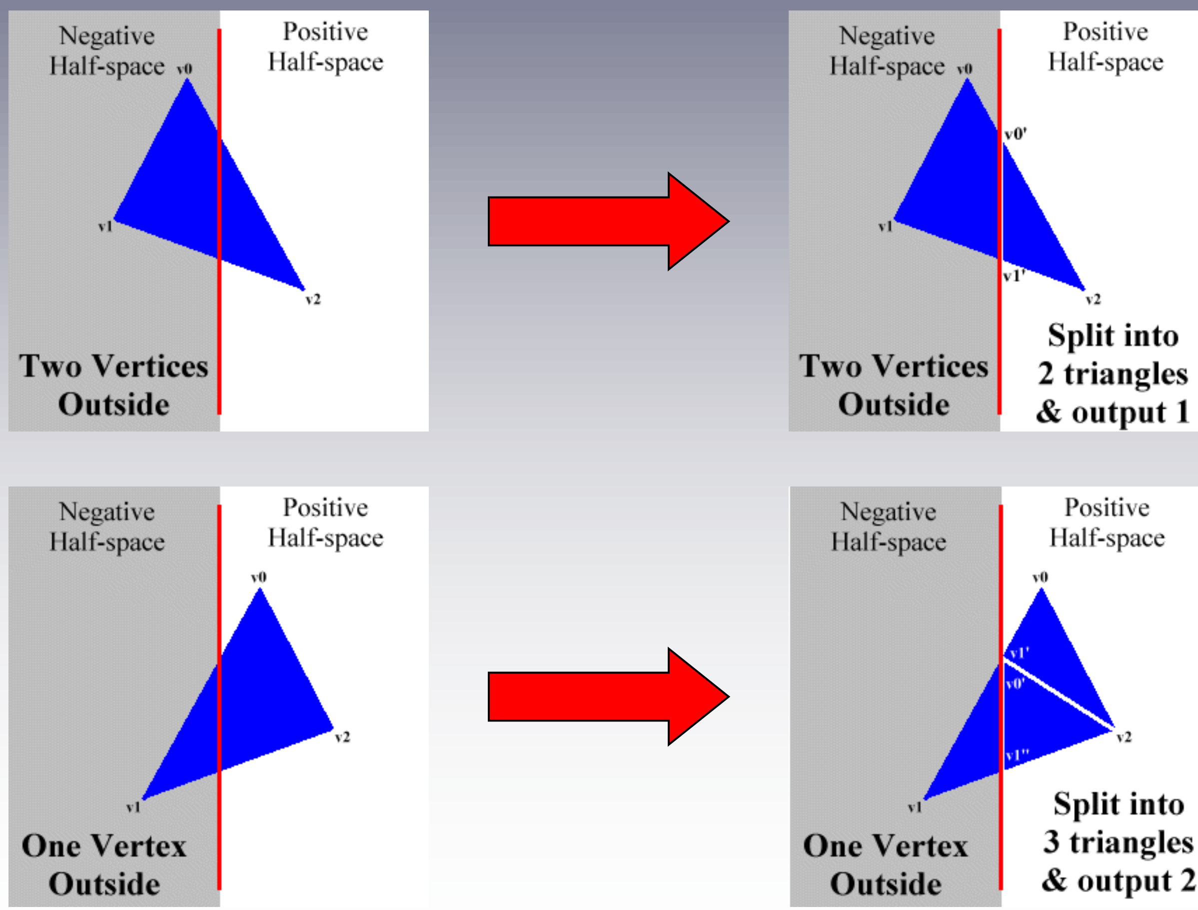
courtesy of L. McMillan

- Can convert arbitrary polygon to set of triangles via **tesselation** (e.g., **`gluTess*()`** functions)



Sutherland-Hodgman: *S* is Triangle

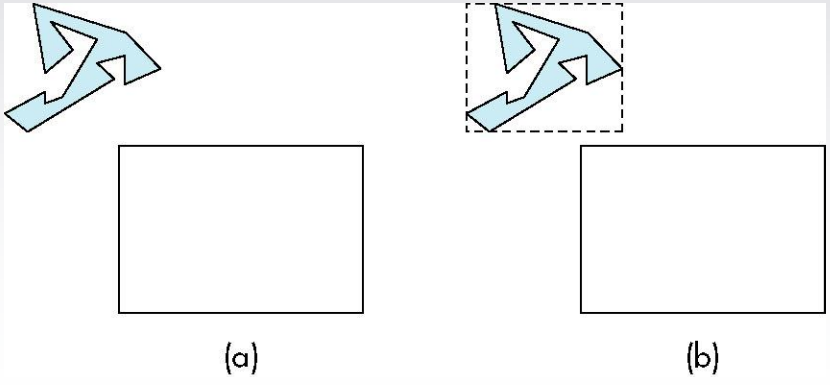
- Dealing with non-trivial cases:



courtesy of L. McMillan

Polygon clipping: Notes

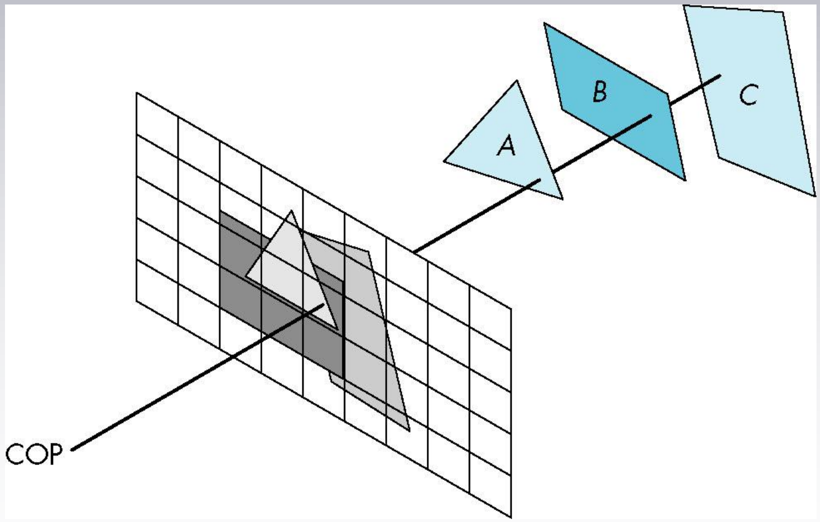
- Sutherland-Hodgman also extends to 3-D straightforwardly
- Often more efficient to fit **bounding areas** (e.g., boxes, spheres) to complex polygons and test those before clipping



from E. Angel

Hidden Surfaces: Why care?

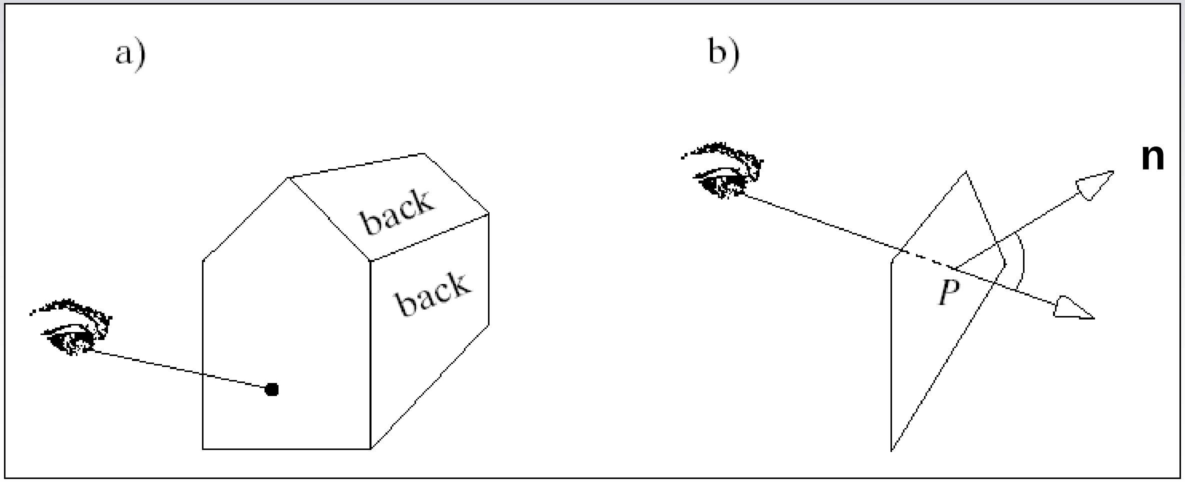
- Hidden surfaces are objects **inside** the viewing volume that should not be seen
- **Occlusion**: Closer (opaque) objects along same viewing ray obscure more distant ones
- Reasons to remove
  - Efficiency: As with clipping, avoid wasting work on invisible objects
  - Correctness: The image will look wrong if we don't model occlusion properly
  - Clarity: useful for wireframes



from Angel

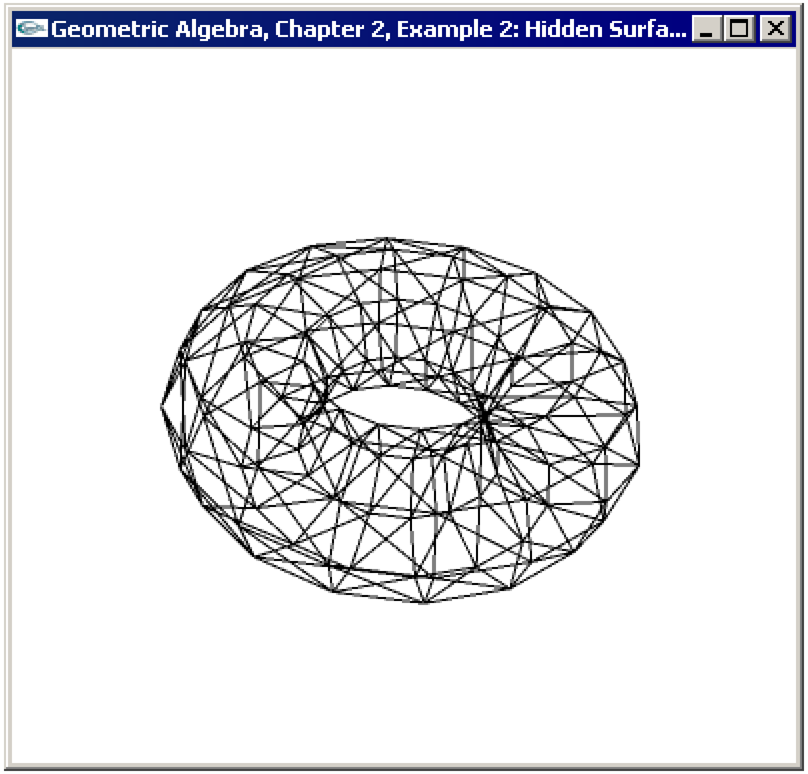
Backface Culling

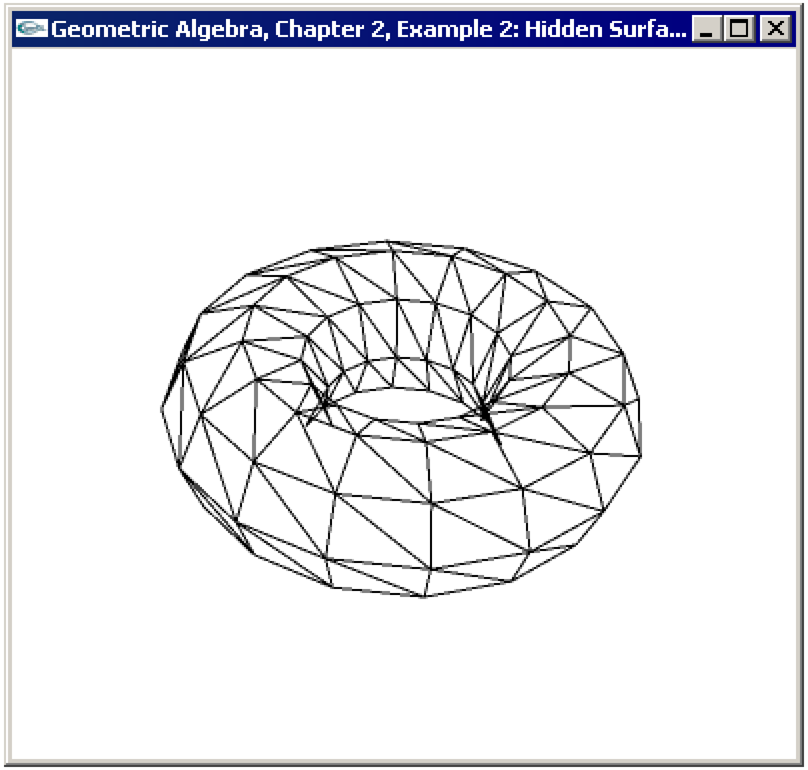
- Basic idea: We don't have to draw polygons that face away from the viewer, since front-facing polygons will occlude them
- A back-facing polygon's **normal** forms an acute angle with the view vector



from Hill

Backface Culling on wireframe





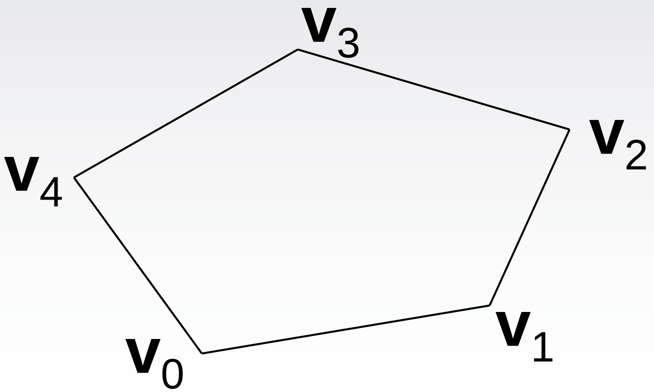
from geometricalgebra.org

Polygon normals

- Let polygon vertices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{n-1}$ be in counterclockwise order and co-planar
- Calculate normal with cross product:

$$\mathbf{n} = (\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_{n-1} - \mathbf{v}_0)$$

- Normalize to unit vector with $\mathbf{n}/|\mathbf{n}|$

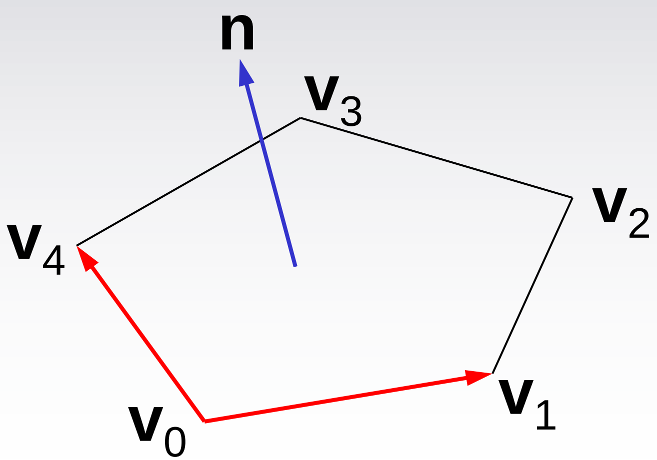


Polygon normals

- Let polygon vertices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{n-1}$ be in counterclockwise order and co-planar
- Calculate normal with cross product:

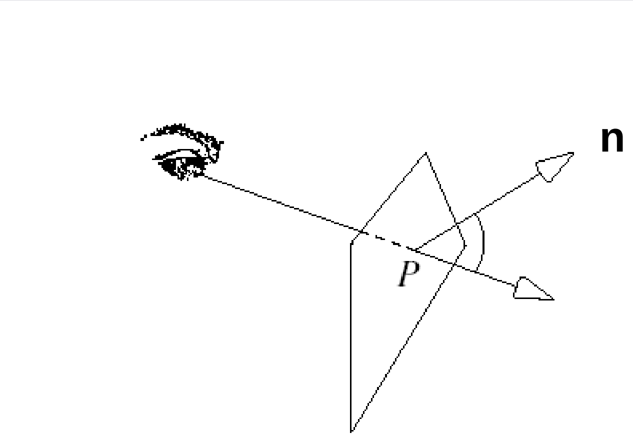
$$\mathbf{n} = (\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_{n-1} - \mathbf{v}_0)$$

- Normalize to unit vector with $\mathbf{n}/|\mathbf{n}|$



Backface culling: Test

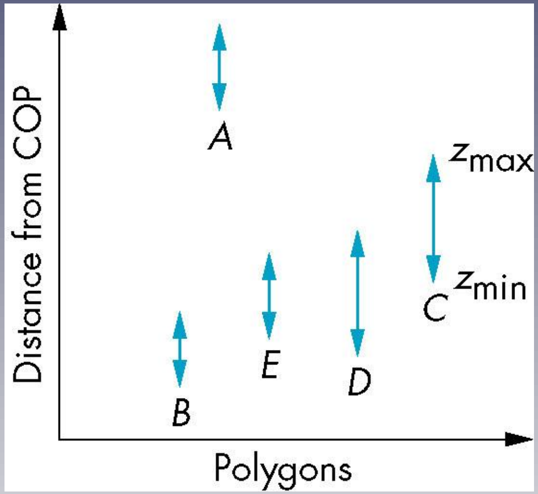
- Polygon with normal **n** is back-facing relative to view direction vector **v** (the Z-axis) if $\mathbf{n} \cdot \mathbf{v} > 0$ (because that means angle is less than 90 degrees)
  - Can also just use point on polygon **p** to define $\mathbf{v} = \mathbf{p} - \mathbf{eye}$ in world coordinates
- Available in OpenGL with **glCullFace()**



from Hill

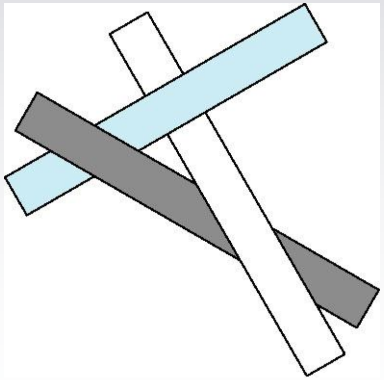
Painter's algorithm

- Idea: Sort primitives by minimum depth, then rasterize from furthest to nearest
- When there are depth overlaps, do more tests of bounding areas, etc. to see one actually occludes the other



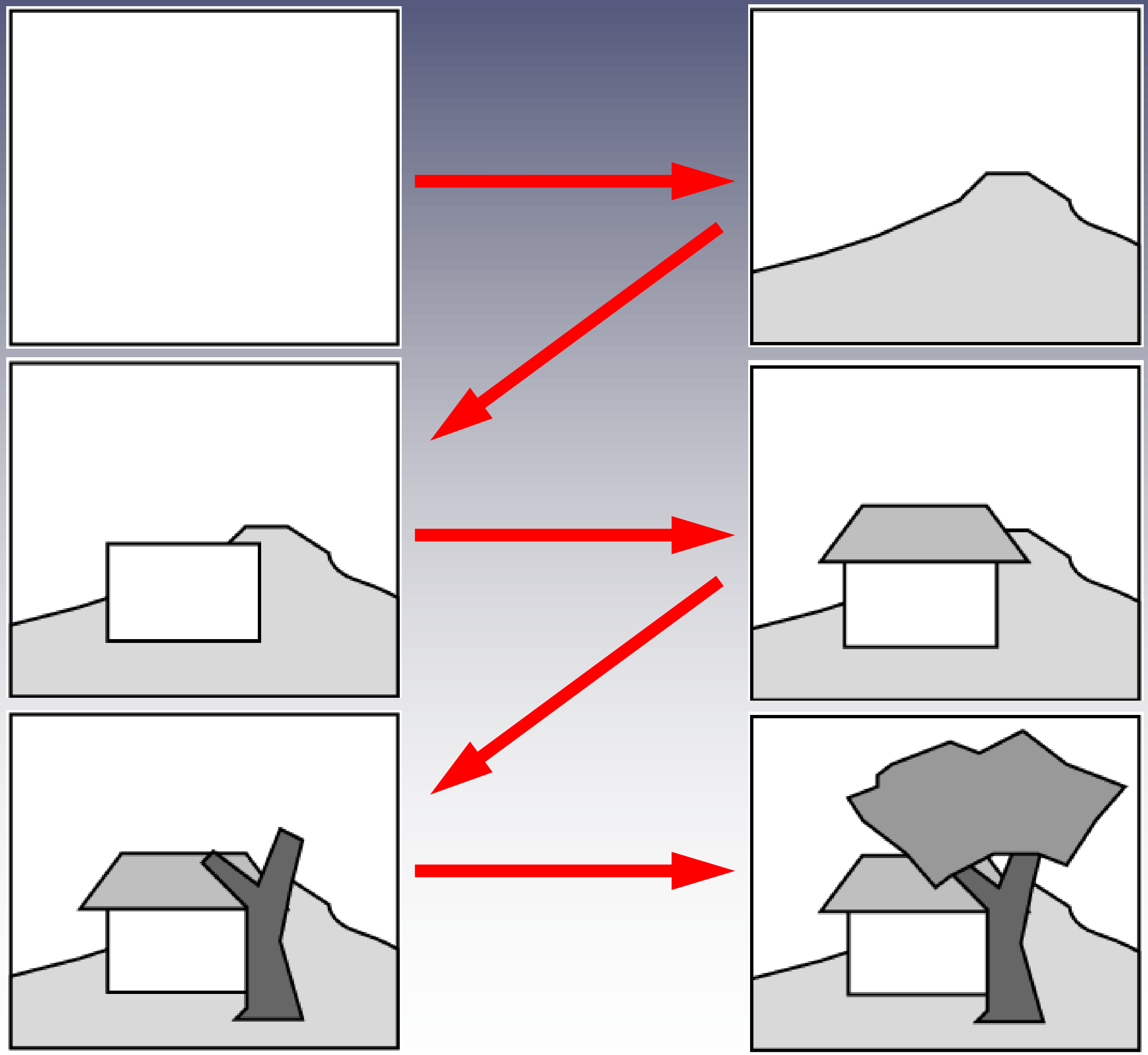
from Angel

- Cyclical overlaps are a problem



Painter's algorithm

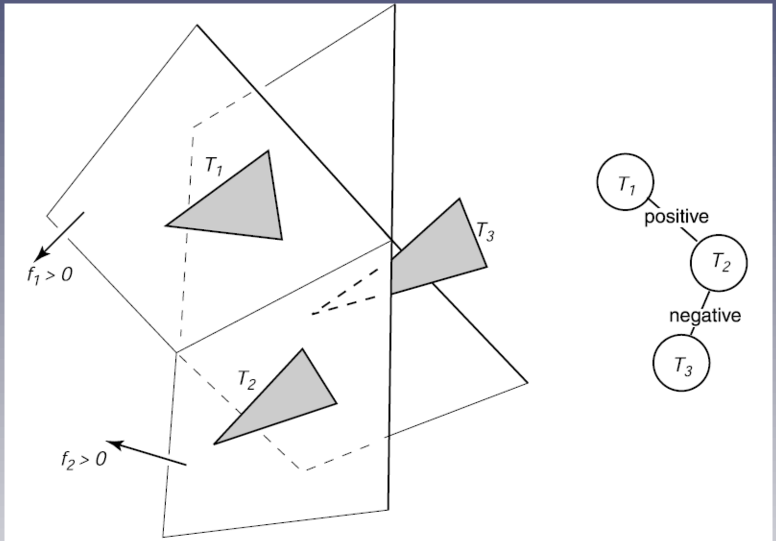
Draw primitives from back to front to avoid need for depth comparisons



from Shirley

BSP trees (a painter's-like approach)

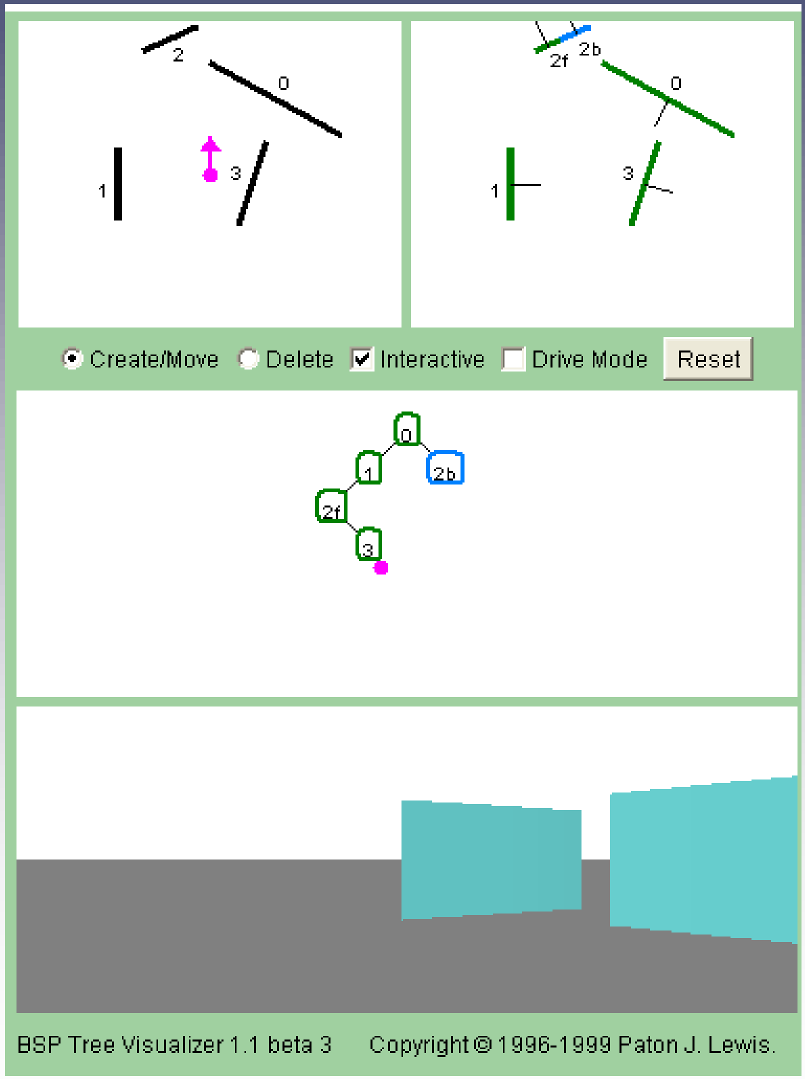
- **B**inary **S**pace **P**artitioning trees: Divide space into visibility regions
  - In 2-D, split with lines
  - In 3-D, split with planes



- Each divider is node of binary tree; left subtree has all objects on one side, right subtree contains other side
- Creating BSP tree offline can make rendering much faster

BSP trees: Creation

- We are just building a binary search tree, but with "which side of divider" test taking place of normal < test
  - Objects that span divider are split
- Use applet at http://pauillac.inria.fr/~levy/bsp
- Balance does not affect efficiency—minimizing splitting is key



BSP trees: Rendering

- Follow painter's algorithm: draw objects from farthest to nearest
- Note that every object is visited

```
void draw_tree(Point eye, bspTree *tree)
{
   if (!tree)
        return;
   if (in_front(eye, tree)) {           // eye is on "front" side of divider
        draw_tree(eye, tree->back);
        draw_object(tree);
        draw_tree(eye, tree->front);
   }
   else if (in_back(eye, tree)) {       // eye is on "back" side of divider
        draw_tree(eye, tree->front);
        draw_object(tree);
        draw_tree(eye, tree->back);
   }
   else {                               // eye is aligned with divider
        draw_tree(eye, tree->front);
        draw_tree(eye, tree->back);
   }
}
```

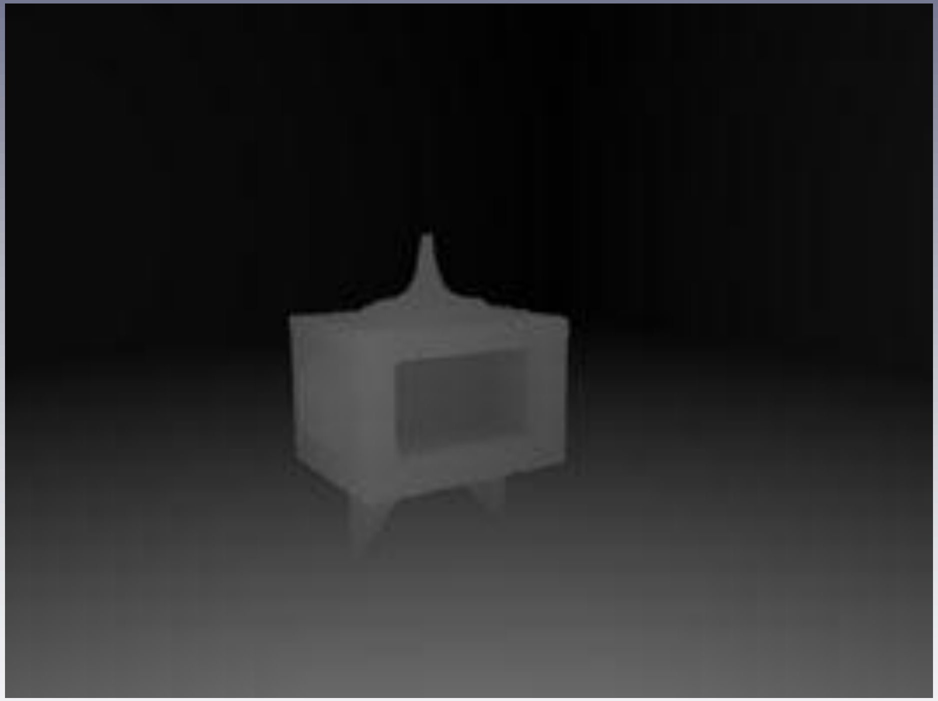
Z-Buffering

- Another hidden surface elimination technique
- Maintain an image-sized buffer **d** of the nearest depth drawn at each pixel so far
- Only draw a pixel if it's closer than what's been rendered already

```
for (each face F)
      for (each pixel (x,y) covering the face)
      {
          depth = depth of F at (x,y);
          if(depth < d[x][y])      //F is closest so far
           {
               c = color of F at (x, y);
               set the pixel color at (x, y) to c
               d[x][y] = depth; // update the depth buffer
           }
      }
```

from Hill

Z-buffer: Example

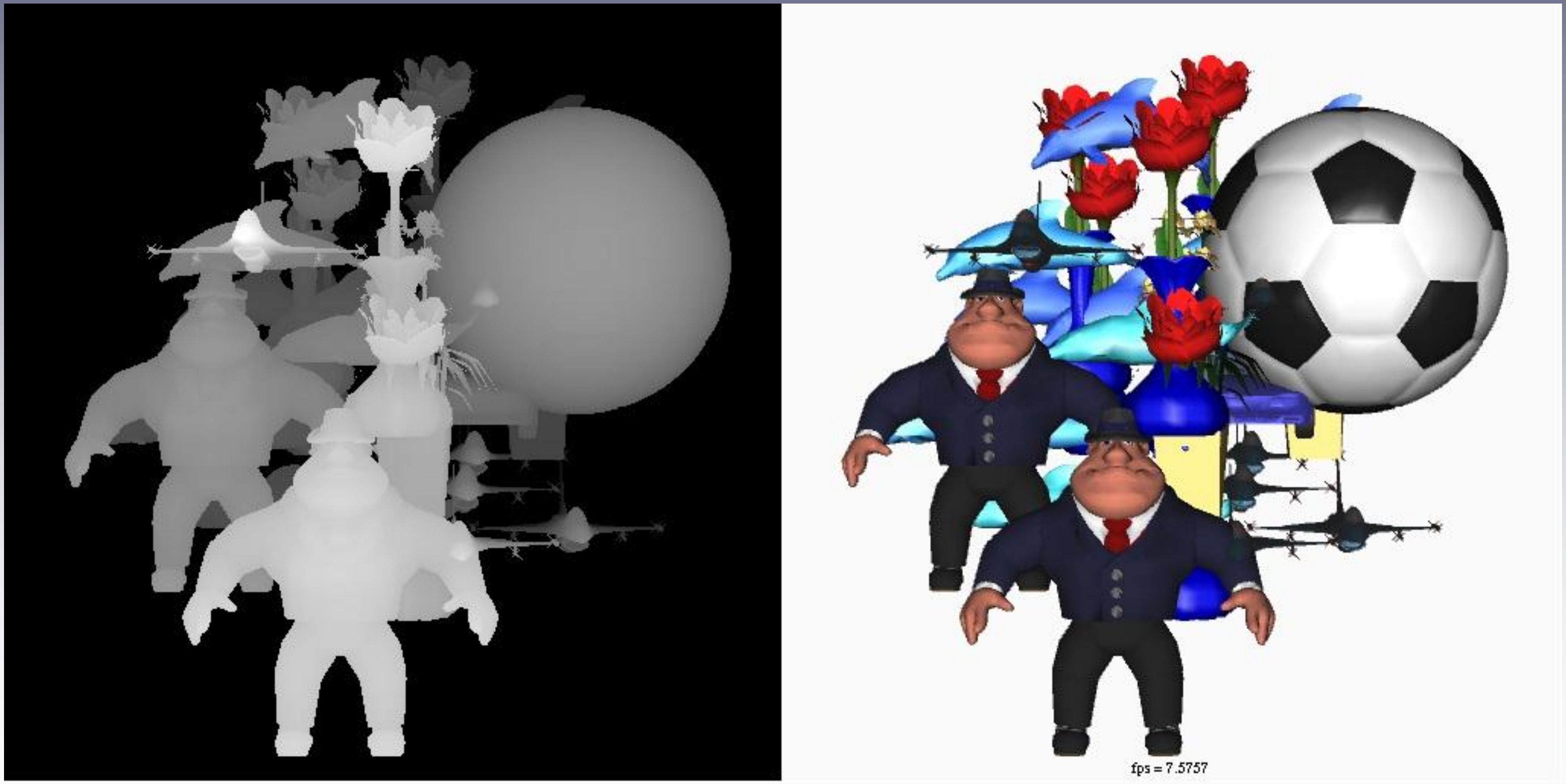


courtesy of DAM Entertainment

Color buffer

Depth buffer

Z-buffer: another example



courtesy of Brent Haley

Z-buffer: Implementation

- Key implementation detail: It is generally unnecessary to independently calculate the depth of each pixel
- Instead, calculate depths of polygon vertices and linearly **interpolate** depth across pixels in between
- E.g., for triangles:
  - Interpolate vertex depths along edges to get z_{left}, z_{right} for a scanline
  - Initialize $z = z_{left}$, increment along scanline by $\Delta z = (z_{right} - z_{left}) / (x_{right} - x_{left})$
- Two stages --> **bilinear interpolation**

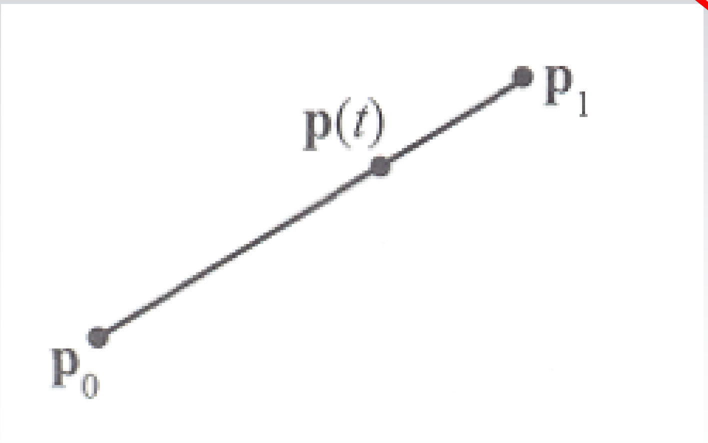
Linear Interpolation (aka `lerp`)

- Parametric definition of a line segment:

$$\mathbf{p}(t) = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0), \text{ where } t \text{ in } [0, 1]$$
$$= \mathbf{p}_0 - t\mathbf{p}_0 + t\mathbf{p}_1$$
$$= (1 - t)\mathbf{p}_0 + t\mathbf{p}_1$$

`= lerp(p0, p1, t)`

like a "blend" of the two endpoints

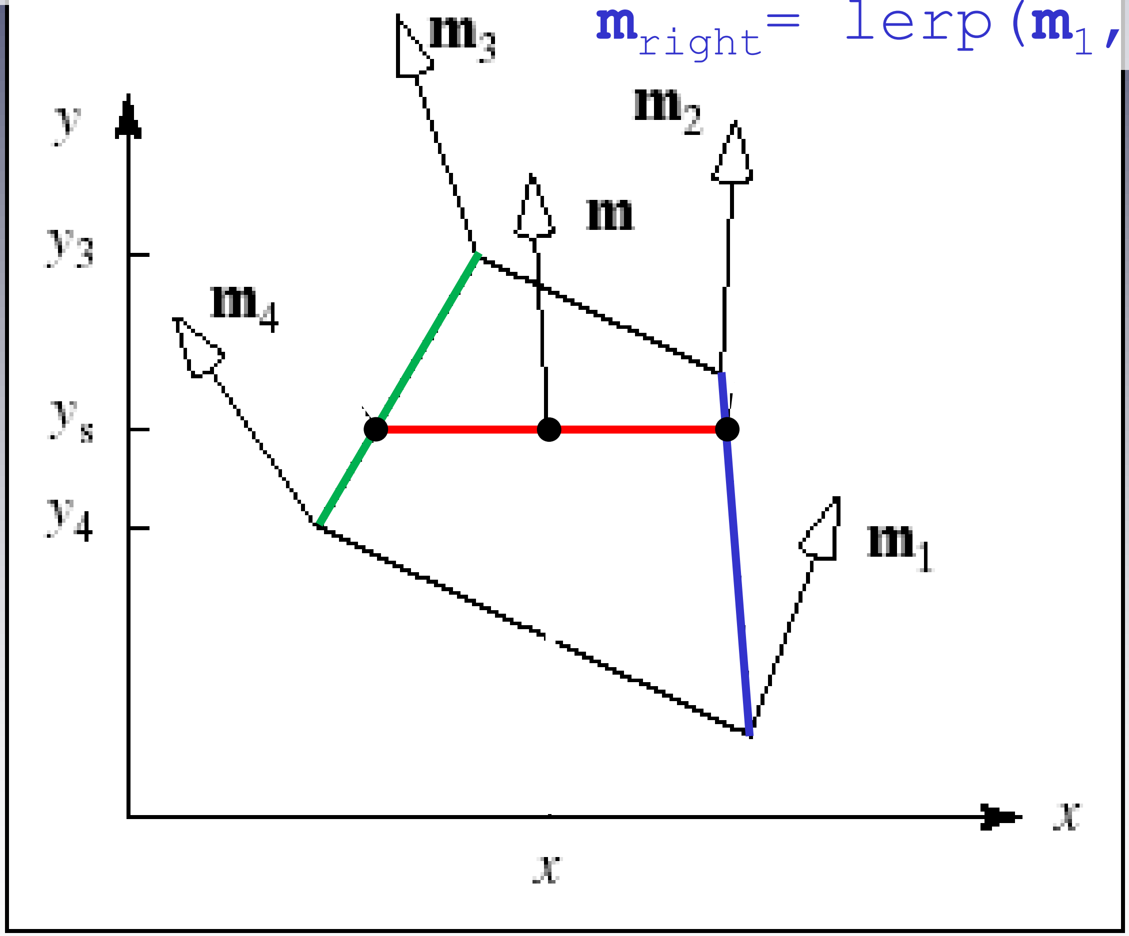


from Akenine-Möller & Haines

Bilinear interpolation for depths

$\mathbf{m}_{left} = \texttt{lerp}(\mathbf{m}_3,\ \mathbf{m}_4,\ t_{left})$

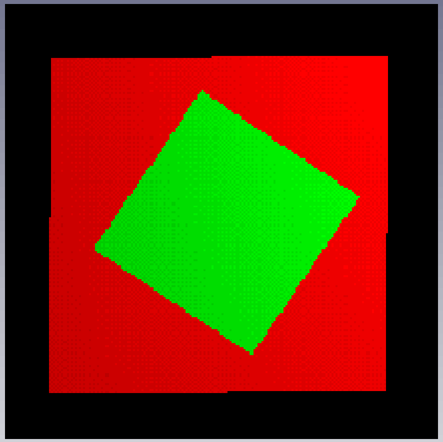
$\mathbf{m}_{right} = \texttt{lerp}(\mathbf{m}_1,\ \mathbf{m}_2,\ t_{right})$

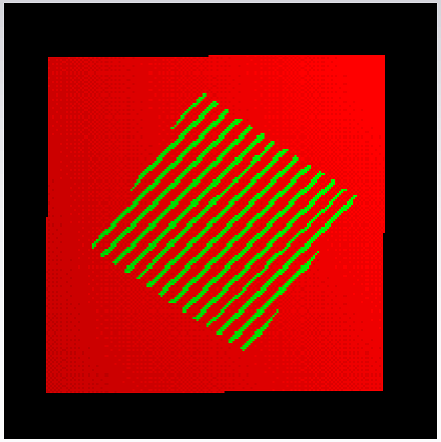


$\mathbf{m} = \texttt{lerp}(\mathbf{m}_{left},\ \mathbf{m}_{right},\ t)$

Z-buffer precision

- Store depth before perspective division or after?
  - If before, constant precision over range of depths
    - E.g., 8 bits with near/far clip plane difference of 10 meters = ~4 cm depth resolution
  - If after (most common), nonlinear function of depth providing more precision closer to viewer
- **Z fighting:** Objects closer to each other than minimum z discrimination mean interpenetration/improper display is possible
  - Example: piece of paper on a desk top
  - Minimize with high-precision Z buffer, pushing near clip plane out as far as possible, and/or polygon offset (depth biasing)





courtesy of SGI

Z fighting example

Z-buffering in OpenGL

- Create depth buffer by setting **`GLUT_DEPTH`** flag in **`glutInitDisplayMode()`**
- Enable per-pixel depth testing with **`glEnable(GL_DEPTH_TEST)`**
- Clear depth buffer by setting **`GL_DEPTH_BUFFER_BIT`** in **`glClear()`**

Z-buffering: Notes

- Pros
  - Interpolation of pixel values from vertex values is easy to do and a key idea in graphics
  - Nearly constant overhead
    - Expensive for simple scenes but good for complex ones
- Cons
  - Relatively late in pipeline
  - Extra storage
  - Precision of depth buffer limits accuracy of object depth ordering for large scale scenes (i.e., nearest to farthest objects)
  - No perfect scheme for handling translucent objects