CSC 4356
Interactive Computer Graphics
Lecture 12: Hidden Surface Removal

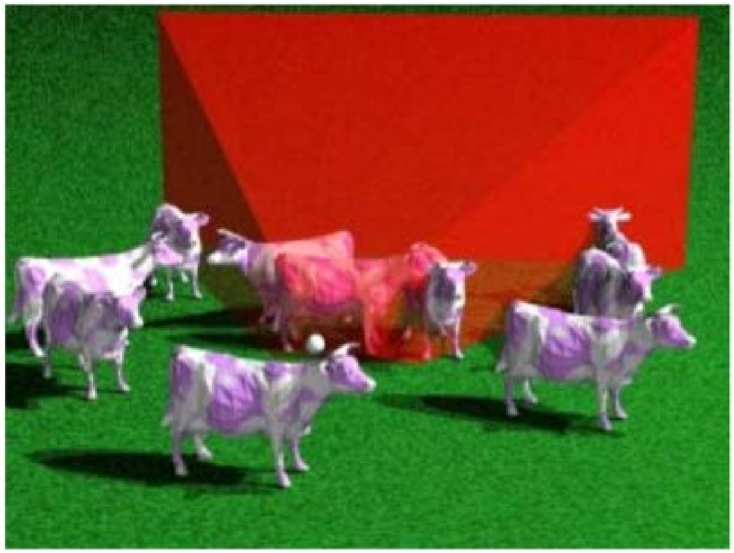
Jinwei Ye

http://www.csc.lsu.edu/~jye/CSC4356/

Tue & Thu: 10:30 - 11:50am
218 Tureaud Hall

Hidden Surfaces

- Back face (self-occlusion)
- Occlusion
- Behind the camera
- Out of the viewing frustum
- Out of the viewport

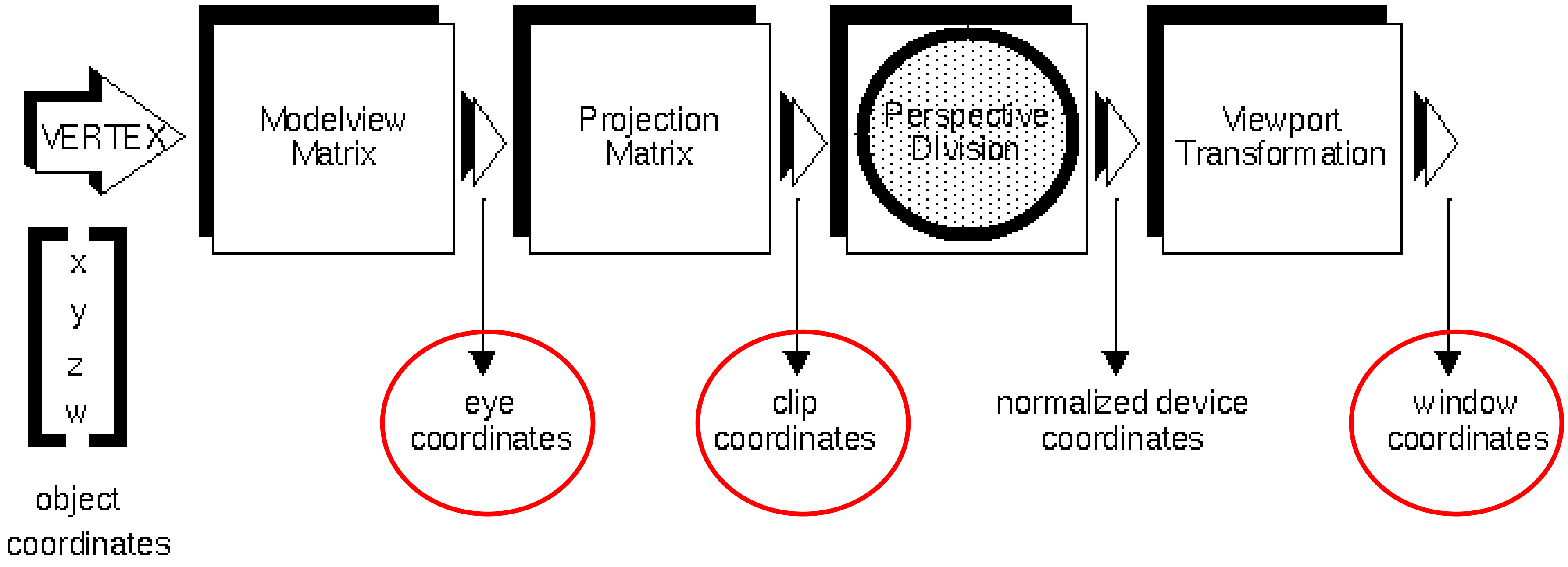


Clipping & Culling

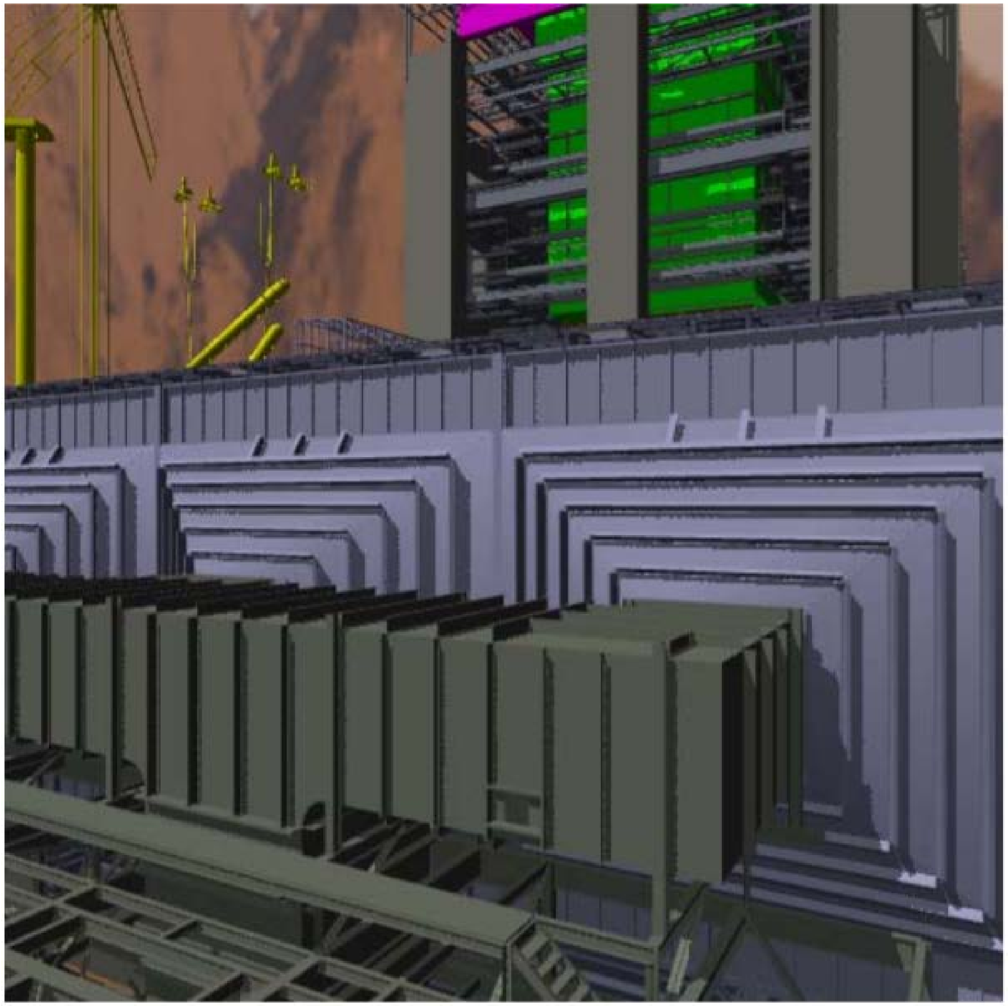
- Culling
  - Throws away entire objects or primitives that cannot possibly be visible
- Clipping
  - “Clips off” the invisible portion of a object or primitive
  - Used to create “cut-away” views of a model
- Optimize and speed up the rendering pipeline

When to clip & cull?

- The earlier in the pipeline invisible primitives are removed, the less computation (such as lighting, texturing etc.) is wasted on them



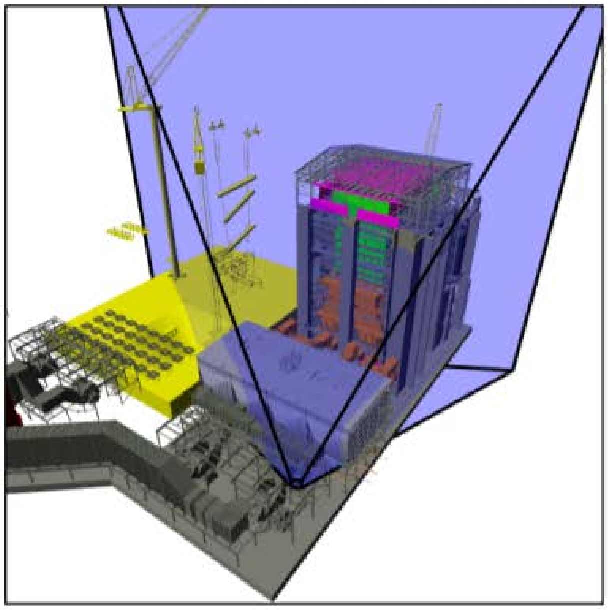
Example



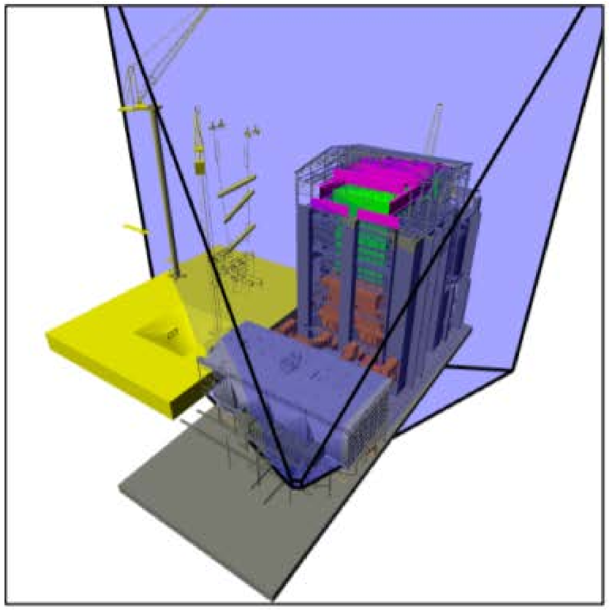
- Power plant model
  - 1.7 M triangles

By Brandon L.. Loyd, UNC

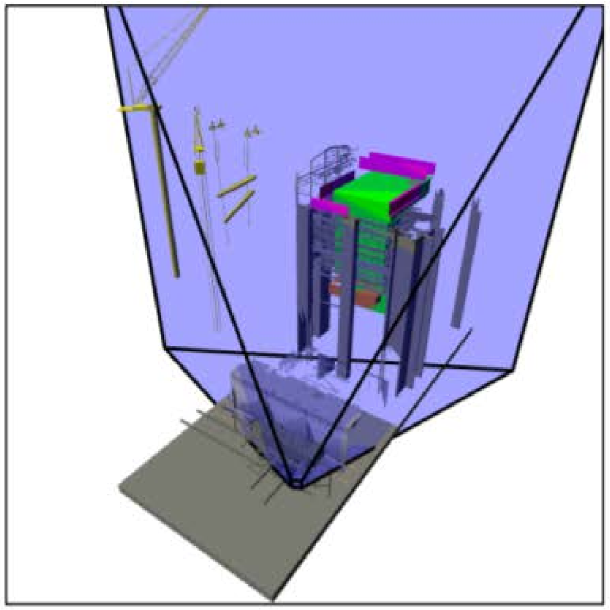
Example



Full model
1.7 Mtris



View frustum culling
1.4 Mtris



Occlusion culling
89 Ktris

5% of the original model!

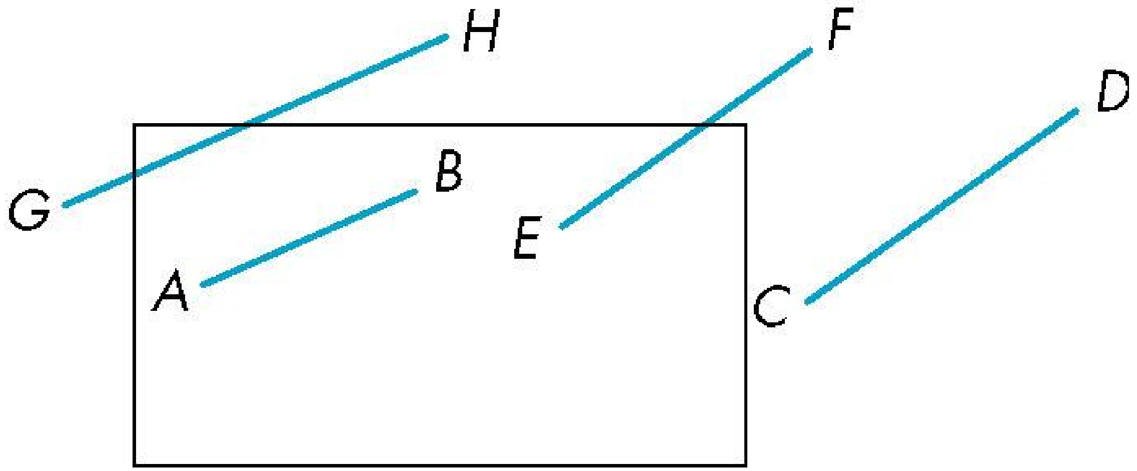
By Brandon L.. Loyd, UNC

Clipping

- Removal of portions of geometric primitives outside viewing volume
- Given a viewing volume (VV):
  - Trivial acceptance: Complete inside VV
  - Trivial rejection: Completely outside VV
  - Crossing **clip plane(s)**: Partially outside, so must trim to fit
- Different primitives require different methods
  - Points: Only trivial accept/reject
  - Lines: Chop at intersection with clip plane
  - Triangles/Polygons: Must trim so as to maintain connectivity

Line Clipping

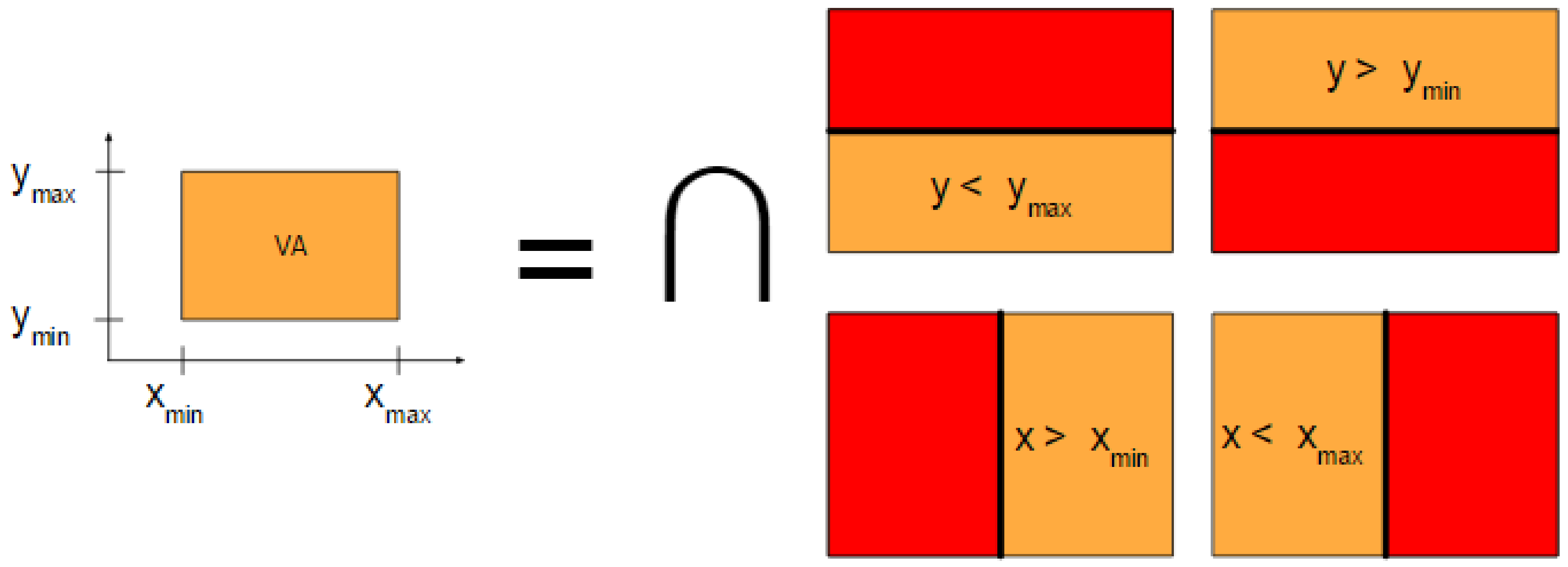
- Testing which side of a clipping plane
  - Which side of a line in 2D case
- How to trim primitives that span border
  - Find the intersection of a line segment and a clipping plane



Outcode Clipping

(a.k.a Cohen-Sutherland Clipping)

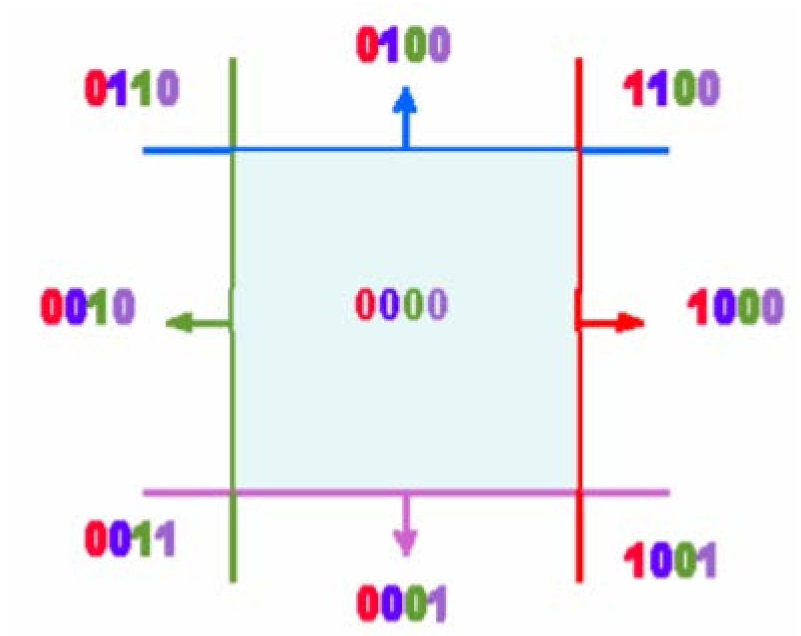
- Idea: Consider rectangular viewing area as intersection of half spaces defined by the four edges of the rectangle



The Outcode

- An outcode identifies the appropriate half plane location of a point relative to all of the clipping edges/planes
- Test half plane by edge equation

  $Ax + By + C = 0$
- Outcodes are usually stored as bit vectors.

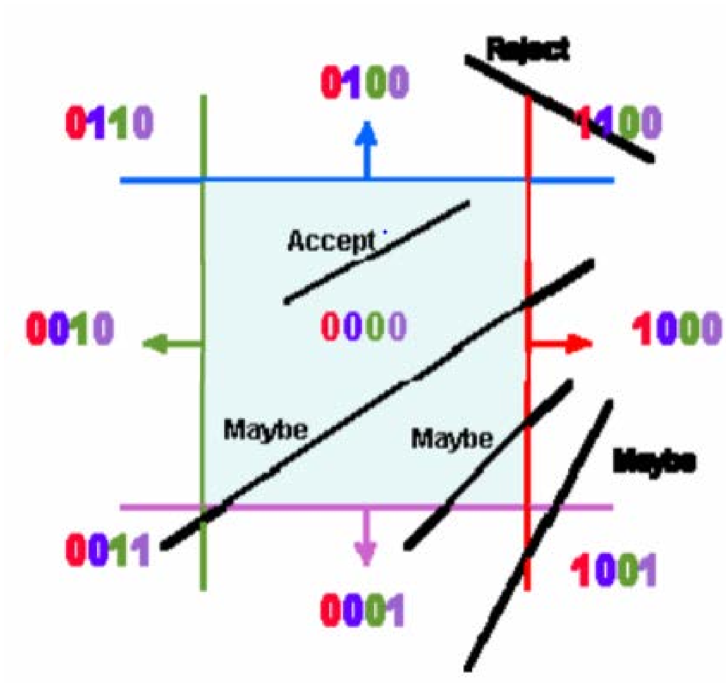


How to Clip using Outcode?

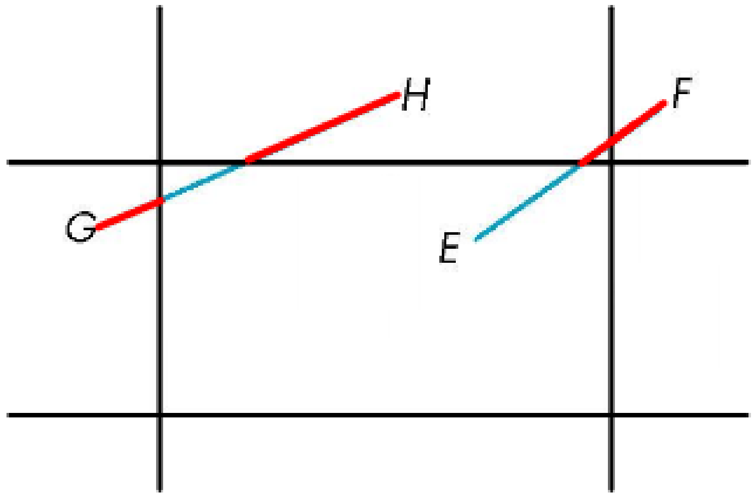
- An outcode identifies the location of a point relative to the viewing area
- Trivial line clipping cases
  - **Accept** line (**p**$_1$, **p**$_2$):

    Both **p**$_1$ and **p**$_2$ are inside
    - o(**p**$_1$) **= 0000** and o(**p**$_2$) **= 0000**
  - **Reject** line (**p**$_1$, **p**$_2$):

    Both **p**$_1$ and **p**$_2$ are outside and on the same side of a edge
    - Both outcodes have a **1** at the same bit position
    - o(**p**$_1$) **= 1100** and o(**p**$_2$) **= 1000**



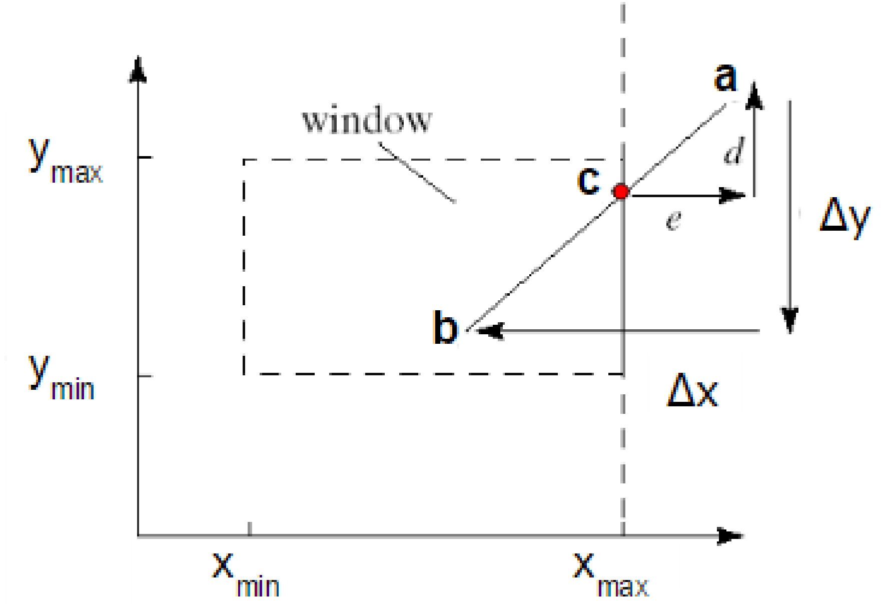
Non-Trivial Cases



- Basic idea: Subdivide non-trivial lines by sequentially clipping portions outside rectangle edge lines until what's left is trivial
  - Arbitrary order: Left, right, bottom, top

Computing Intersection

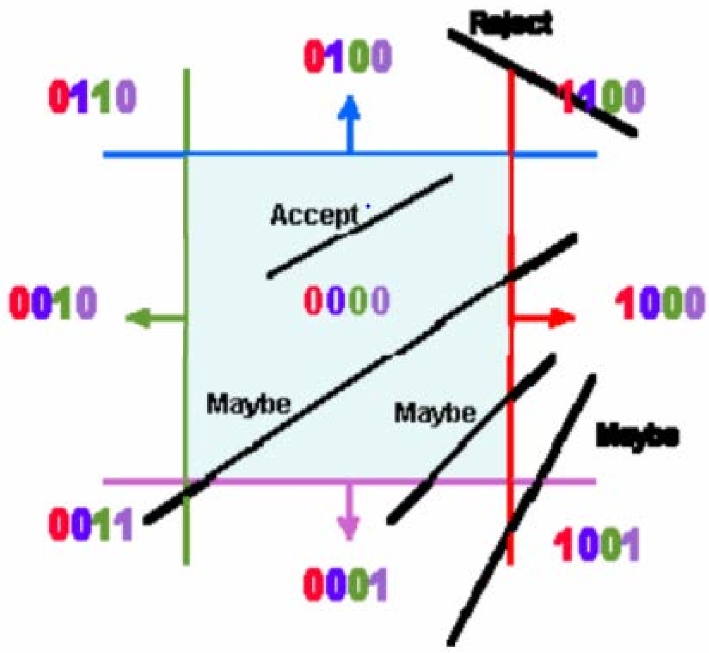
- What is the intersection point **c** = (c_x, c_y)?
  - Obviously, in this case $c_x = x_{max}$ and $c_y = a_y - d$
  - Noting that $e = a_x - x_{max}$ and $e/\Delta x = d/\Delta y$, we can compute $d = e\,\Delta y/\Delta x$ and obtain c_y
- A similar approach works for the other clip line



Psuedo-Code

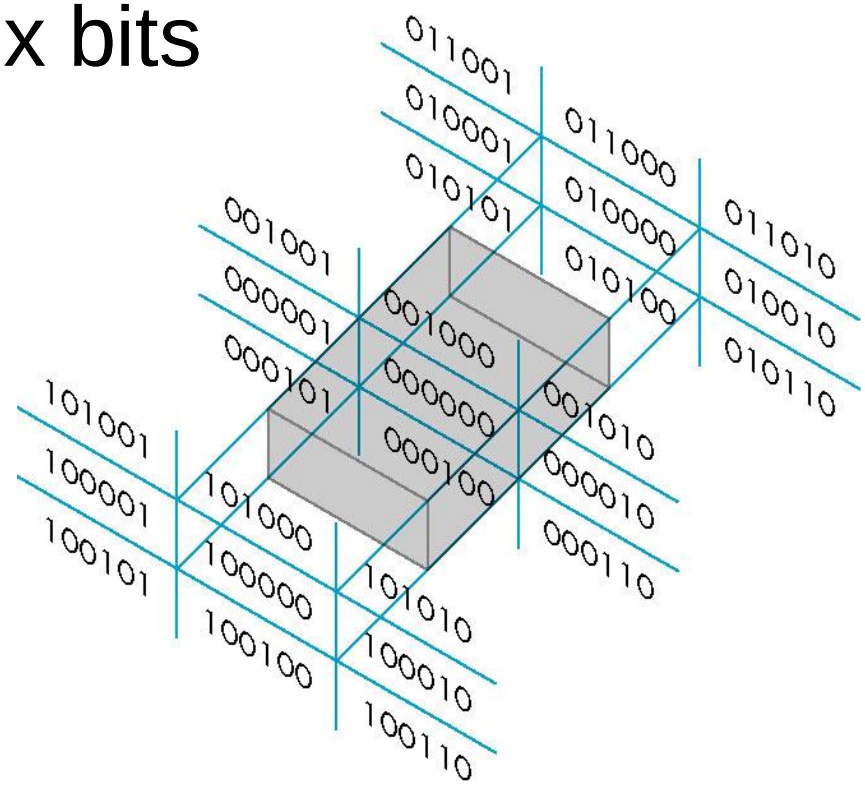
```
int clipSegment(Point2& p1, Point2& p2, RealRect W)
{
      do{
           if(trivial accept) return 1; // some portion survives
           if(trivial reject) return 0; // no portion survives

           if(p1 is outside )
           {
               if(p1 is to the left )  chop against the left edge
               else if(p1 is to the right ) chop against the right edge
               else if(p1 is below)  chop against the bottom edge
               else if(p1 is above)  chop against the top edge
           }
           else      // p2 is outside
           {
              if(p2 is to the left ) chop against the left edge
                   else if(p2 is to the right ) chop against the right edge
              else if(p2 is below)  chop against the bottom edge
              else if(p2 is above) chop against the top edge
           }
      }while(1);
}
```



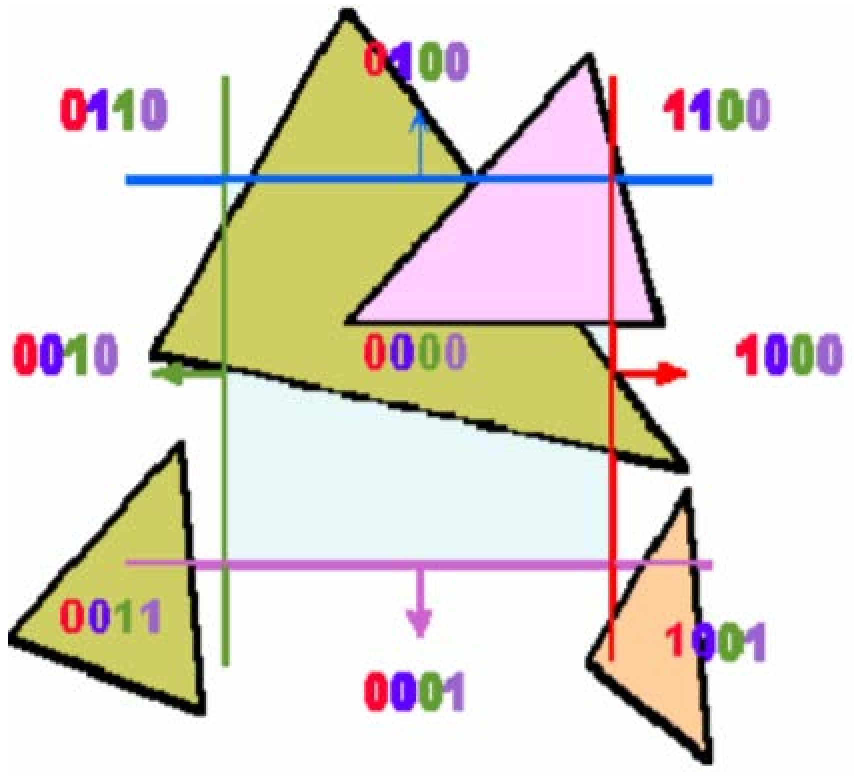
3D Generalization

- Instead of having four half-planes, there are six half-planes
- Outcodes now have six bits
  - Two more bits for the
  - near and far planes



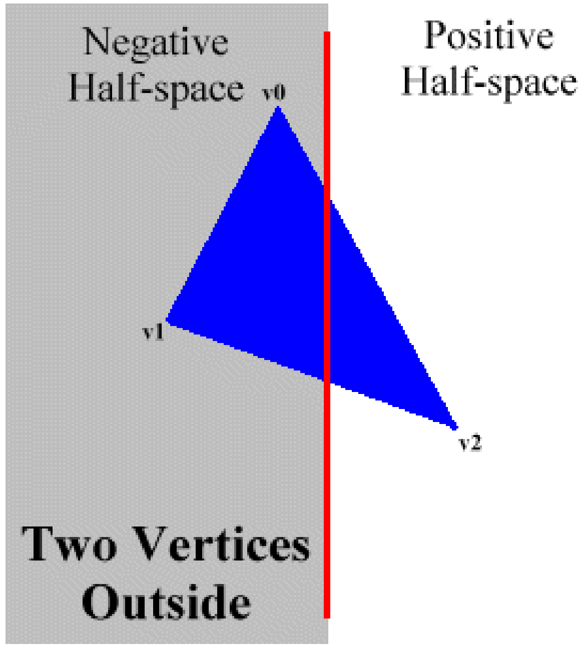
Triangle Clipping by Outcode

- For triangles, we need to perform inside/outside test for three vertices

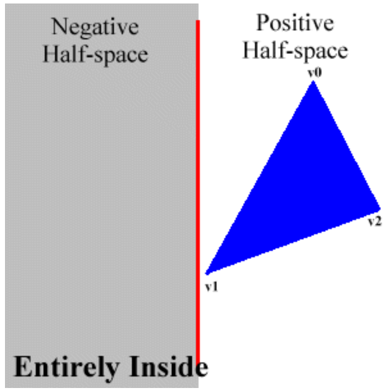


One-Plane-at-a-Time Clipping
(a.k.a. Sutherland-Hodgeman Algorithm)

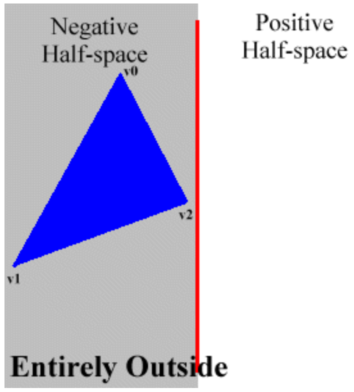
- The Sutherland-Hodgeman triangle clipping algorithm uses a divide-and-conquer strategy
- It first solves the simple problem of clipping a triangle against a single plane
  - Four cases
- Each of the clipping planes are applied in succession to every triangle.



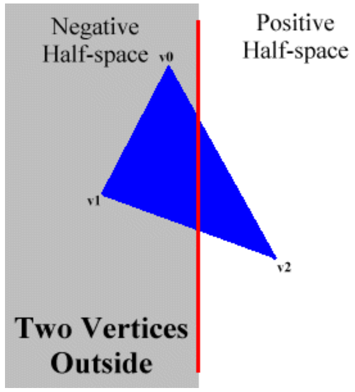
Triangle Cases



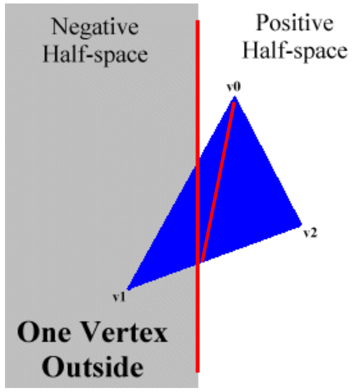
Keep Complete Triangle



Reject Complete Triangle



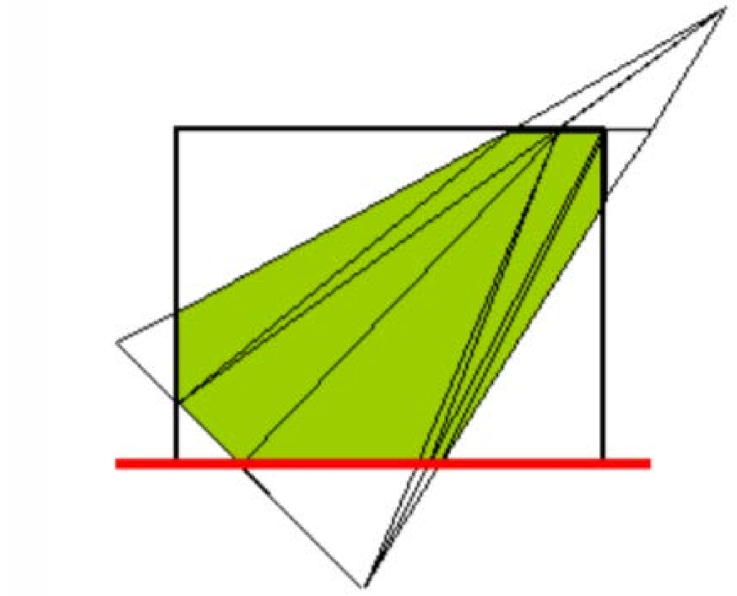
Keep Triangle Portion Inside



Keep Quadrilateral Portion Inside and Split into Two Triangles

One-Plane-at-a-Time Clipping

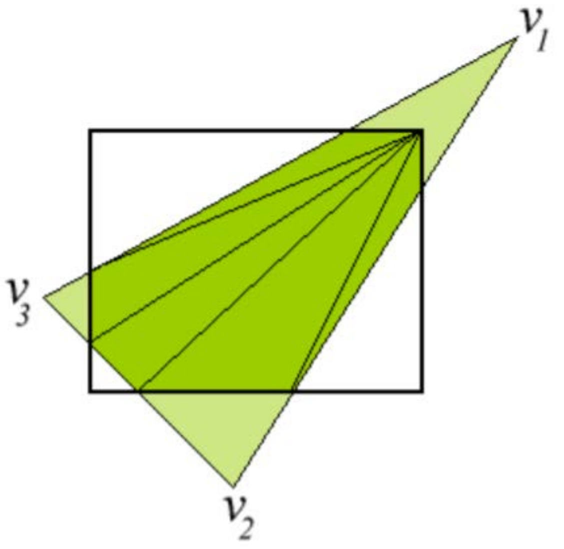
- The results of Sutherland-Hodgeman clipping can get complicated very quickly once multiple clipping planes are considered.
- However, the algorithm is still very simple. Each clipping plane is treated independently, and each triangle is treated by one of the four cases mentioned previously.



Alternatives to Plane-at-a-time

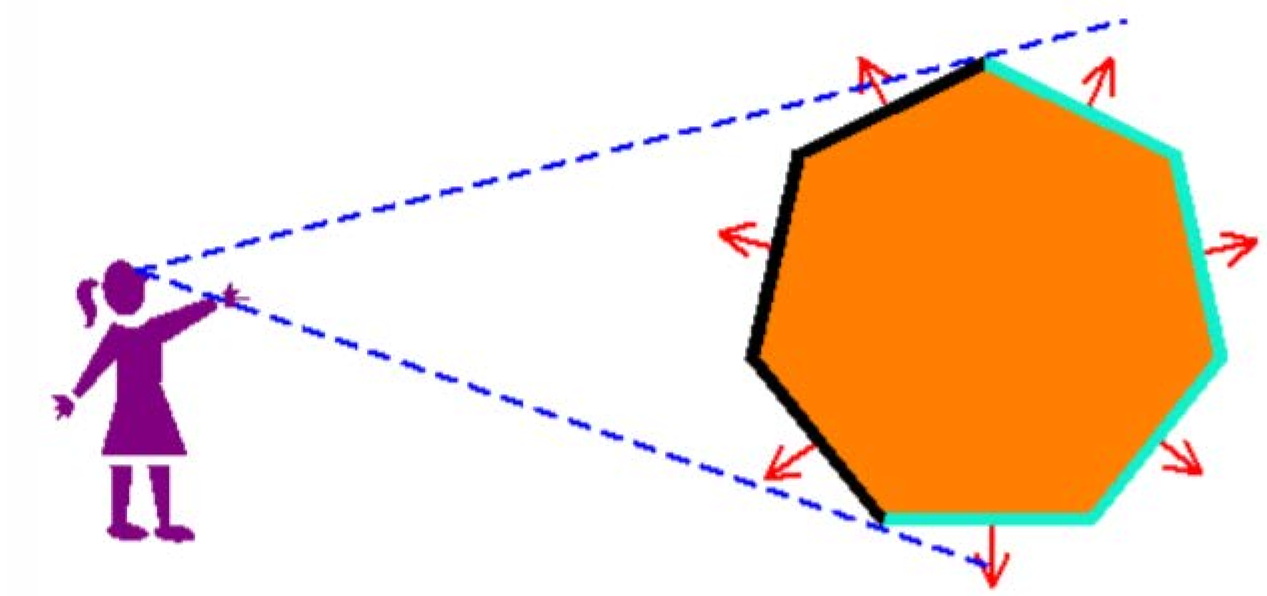
Over the years there have been improvements to one-plane-at-a-time clipping.

- **Wieler-Atherton clipping**: clipping against concave volumes
  - Can clip arbitrary polygons against arbitrary polygons
  - Maintains more state than plane-at-a-time clipping
- **Nicholle-Lee-Nicholle clipping**: handle all planes at once
  - It waits before generating triangles to reduce the number of clip sections generated



Back Face Culling

- Back-face culling addresses a special case of occlusion called convex self-occlusion
- If an object is closed (having a well defined inside and outside) then some parts of the outer surface must be blocked by other parts of the same surface.
- On such surfaces we need only consider the normals of surface elements to determine if they are visible.

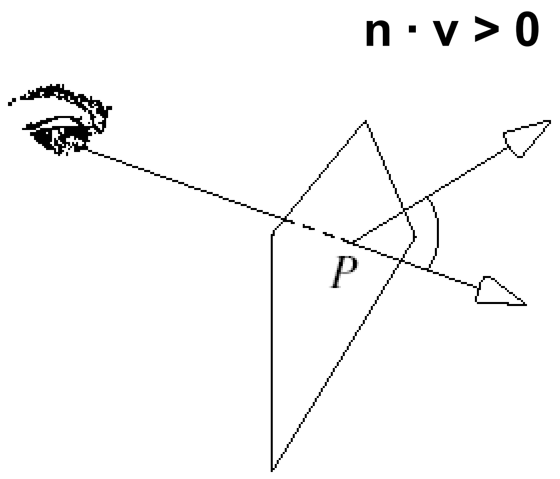


Removing Back-Faces

- Basic Idea: Compare the normal of each face with the viewing direction

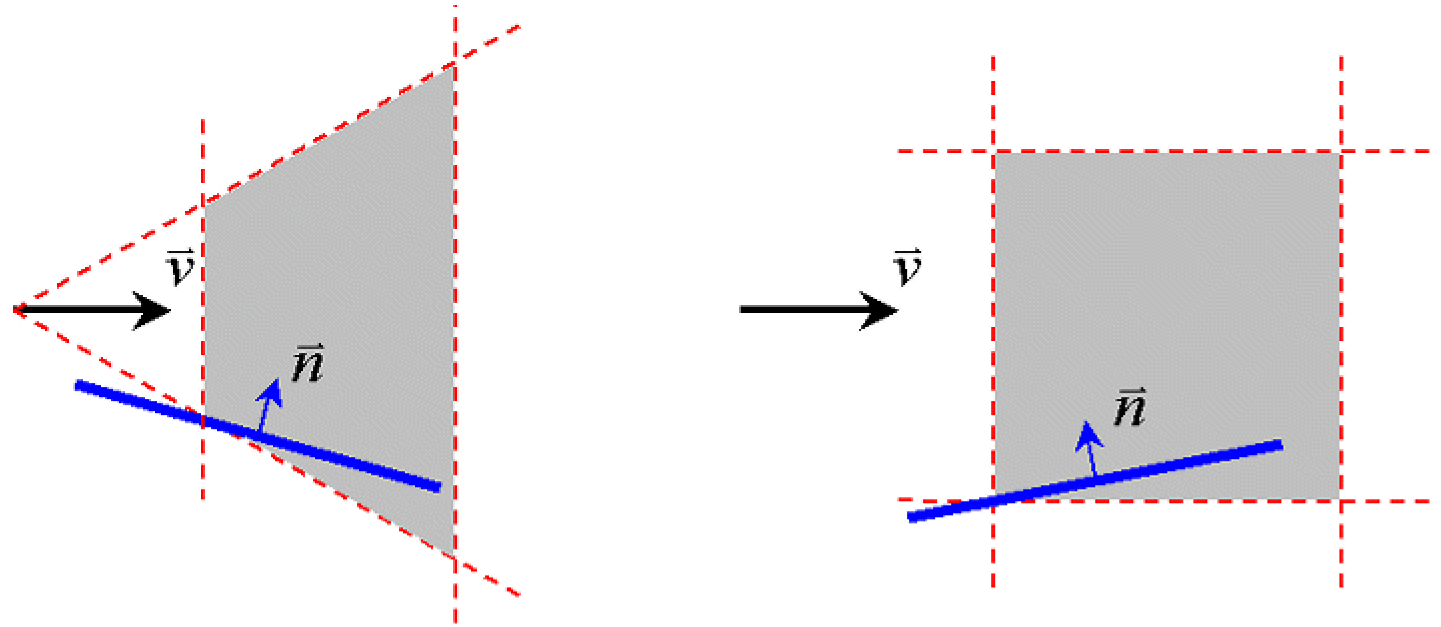
- Given n, the *outward*-pointing normal of F
  - for each face F of object
  - if (n · v > 0)
    throw away the face

- Does it always work?



Fixing the Problem

We can’t do view direction clipping just anywhere!



Downside: Projection comes fairly late in the pipeline. It would be nice to cull objects sooner.

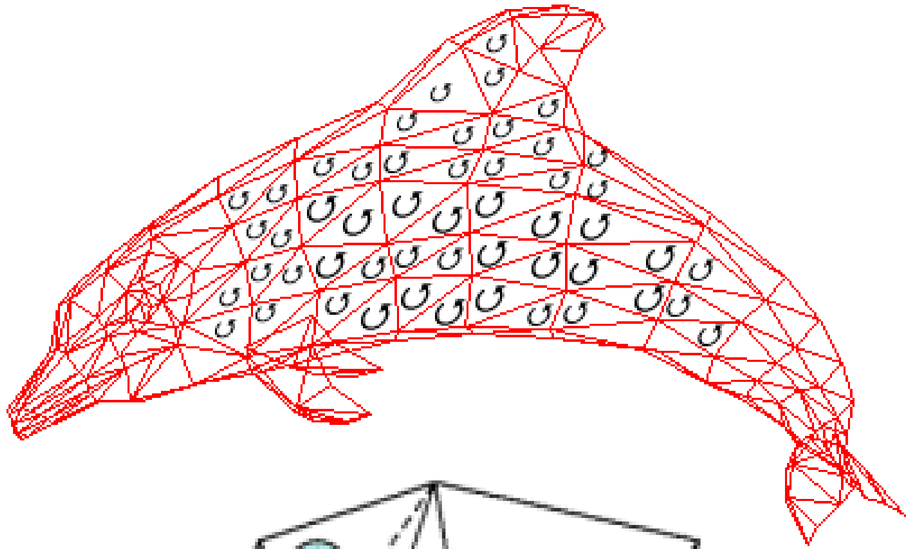
Upside: Computing the dot product is simpler.

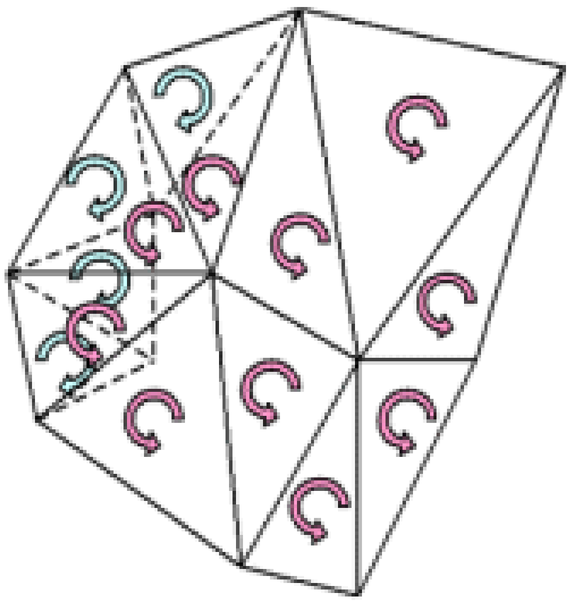
Culling Technique #2

Detect a change in screen-space orientation.

- If all face vertices are ordered in a consistent way, back-facing primitives can be found by detecting a reversal in this order. One choice is a counterclockwise ordering when viewed from outside of the manifold. This is consistent with computing face normals (Why?). If, after projection, we ever see a clockwise face, it must be back facing.

- This approach will work for all cases, but it comes even later in the pipe, at triangle setup. We already do this calculation in our triangle rasterizer. It is equivalent to determining a triangle with negative area.





Culling Plane Test

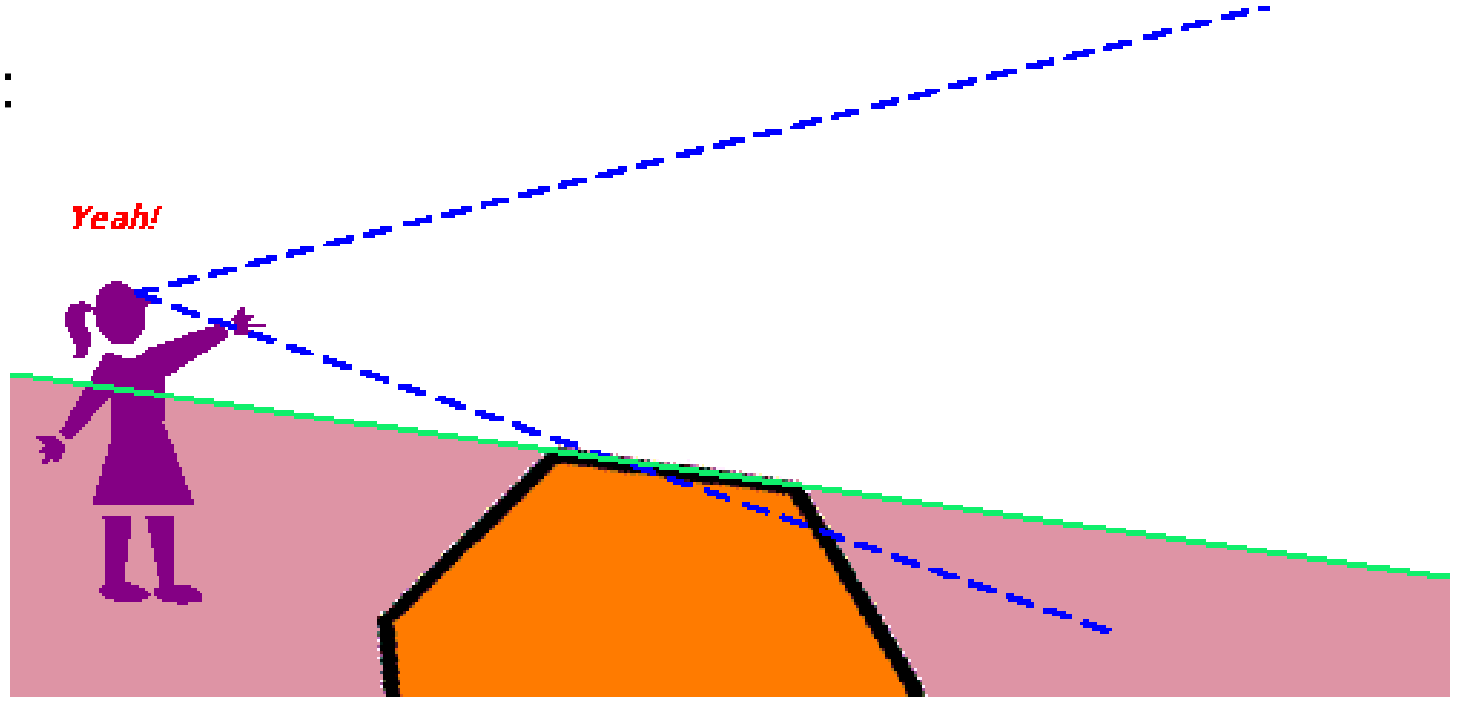
- Here is a culling test that will work anywhere in the pipeline.
- Remove faces that have the eye in their negative half-space. This requires computing a plane equation for each face considered.

$$\begin{bmatrix} n_x & n_y & n_z & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} - d = 0$$

Culling Plane Test

- Once we have the plane equation, we substitute the coordinate of the viewing point (the eye coordinate in our viewing matrix). If it is negative, then the surface is backfacing.

- Example:



Back Face Culling in OpenGL

- Back Face Culling is available as a mode setting in OpenGL
  - Very flexible (can cull fronts as well as backs)
  - It can double your performance!
- glEnable(GL_CULL_FACE)
  glCullFace(GL_BACK)