

The Rasterisation Pipeline



Ray-tracing: Very ‘realistic’ images but not usually real-time



Need photo-realism at interactive rates for games and VR

Summary

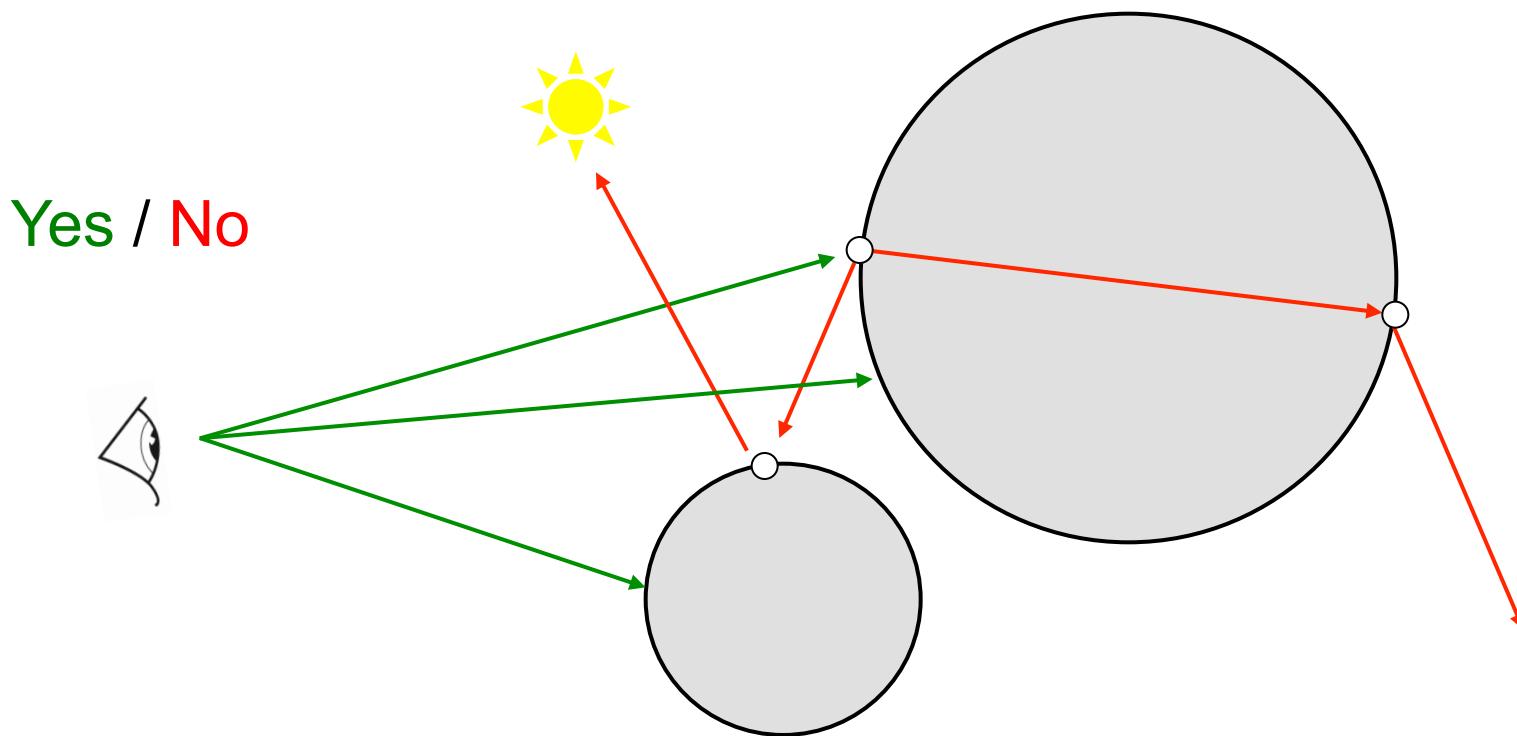
- Ray-tracing is elegant and general but slow
- Now we want to speed this up
- Main idea:
 - Don't **trace** rays
 - **Project** primitives
- Creates many new challenges

Ray-tracing cost

- The process of casting rays is very slow
- Example
 - 10,000 triangles,
 - 1000x1000 pixels
 - Result in 1,000,000 primary rays to cast
 - Each one testing for intersection with the 10,000 triangles (and then reflections, shadow rays, etc...)

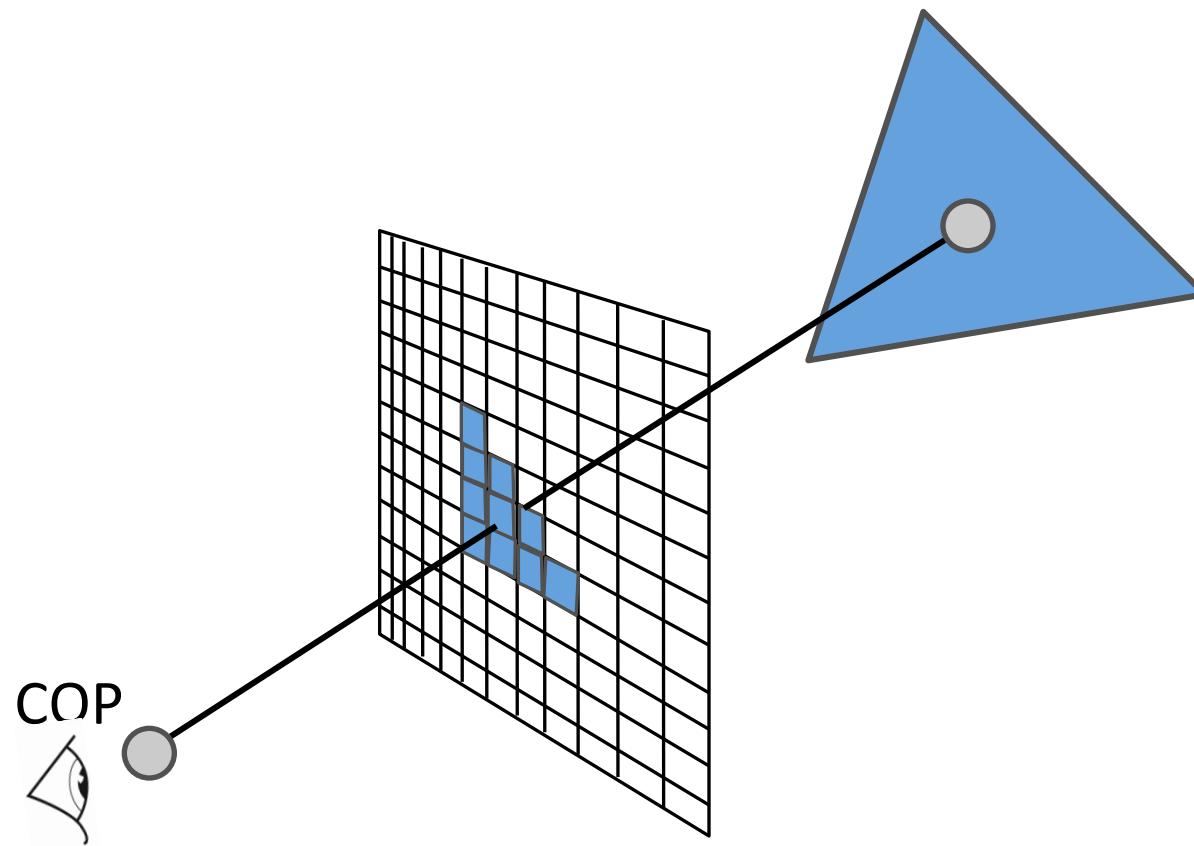
Simplifications from ray-tracing

- Rays only from one or a few specific points
- Only convex polygons
- No global illumination part (i.e. no recursion)



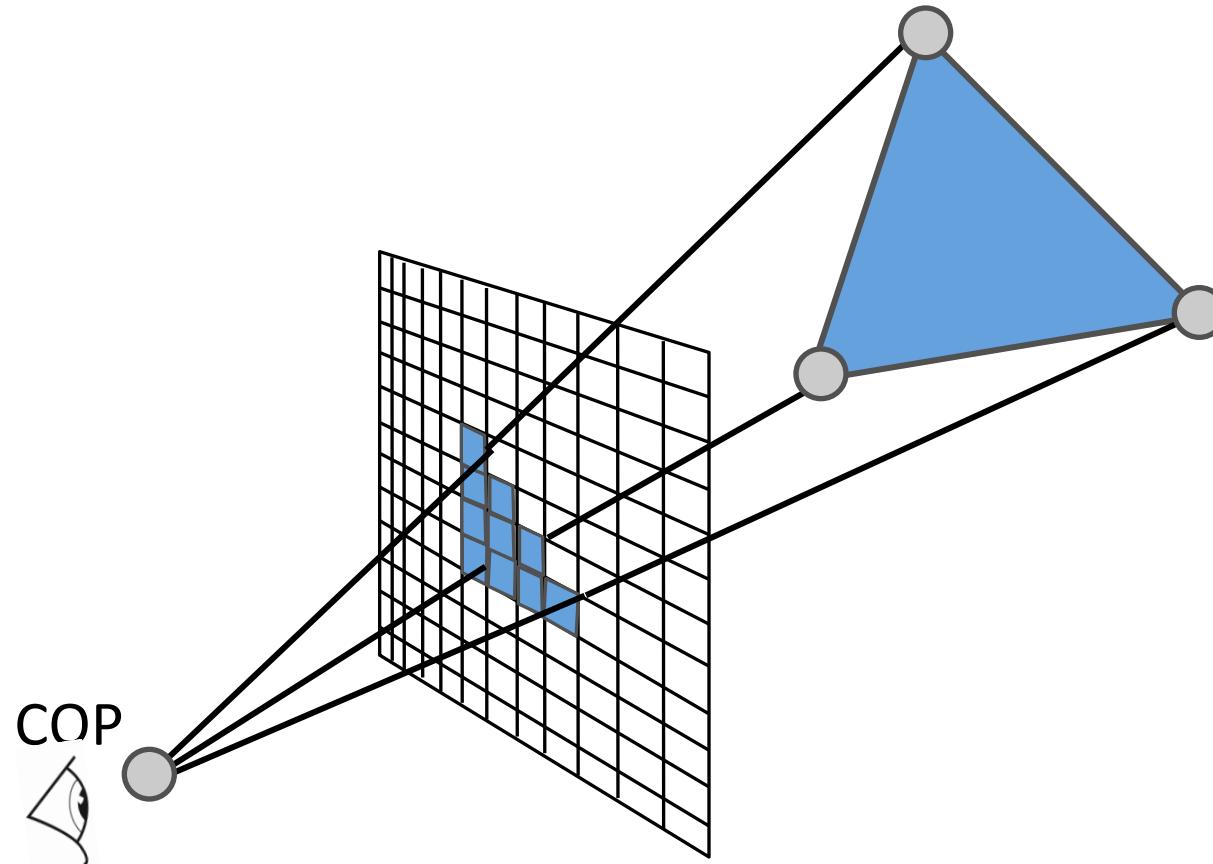
Ray-tracing

```
for all pixel rays i  
for all primitives j  
shade(intersect(pixel i, primitive j))
```

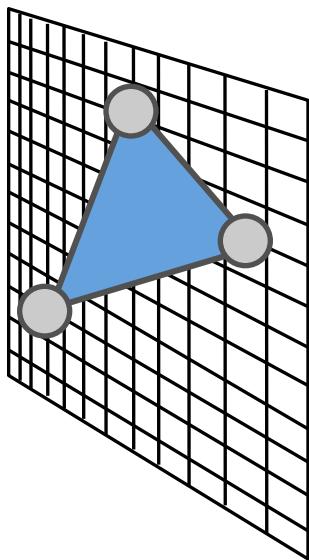


Rasterization

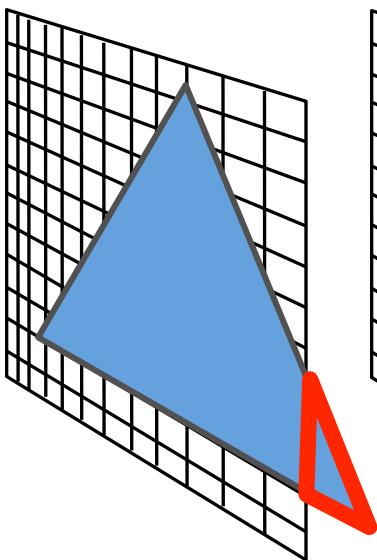
```
for all primitives i  
  for all pixels j in projection of primitive i  
    shade(pixel j, primitive i)
```



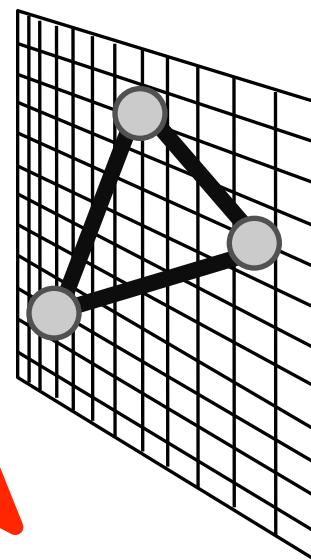
Challenges



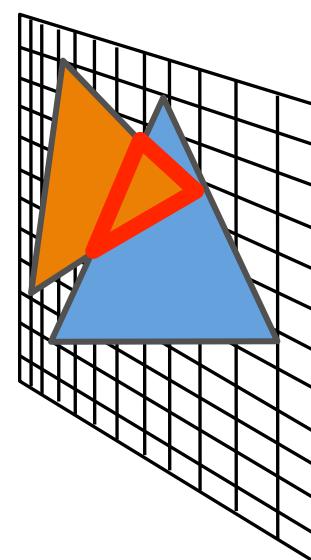
Projection



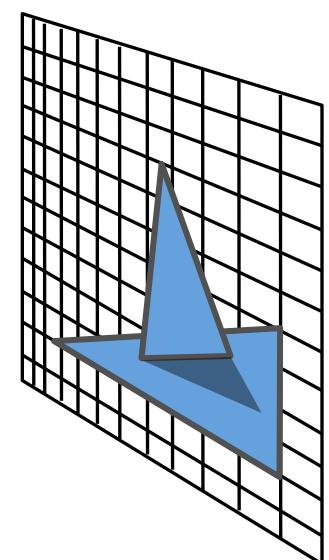
Clipping



Rasterization

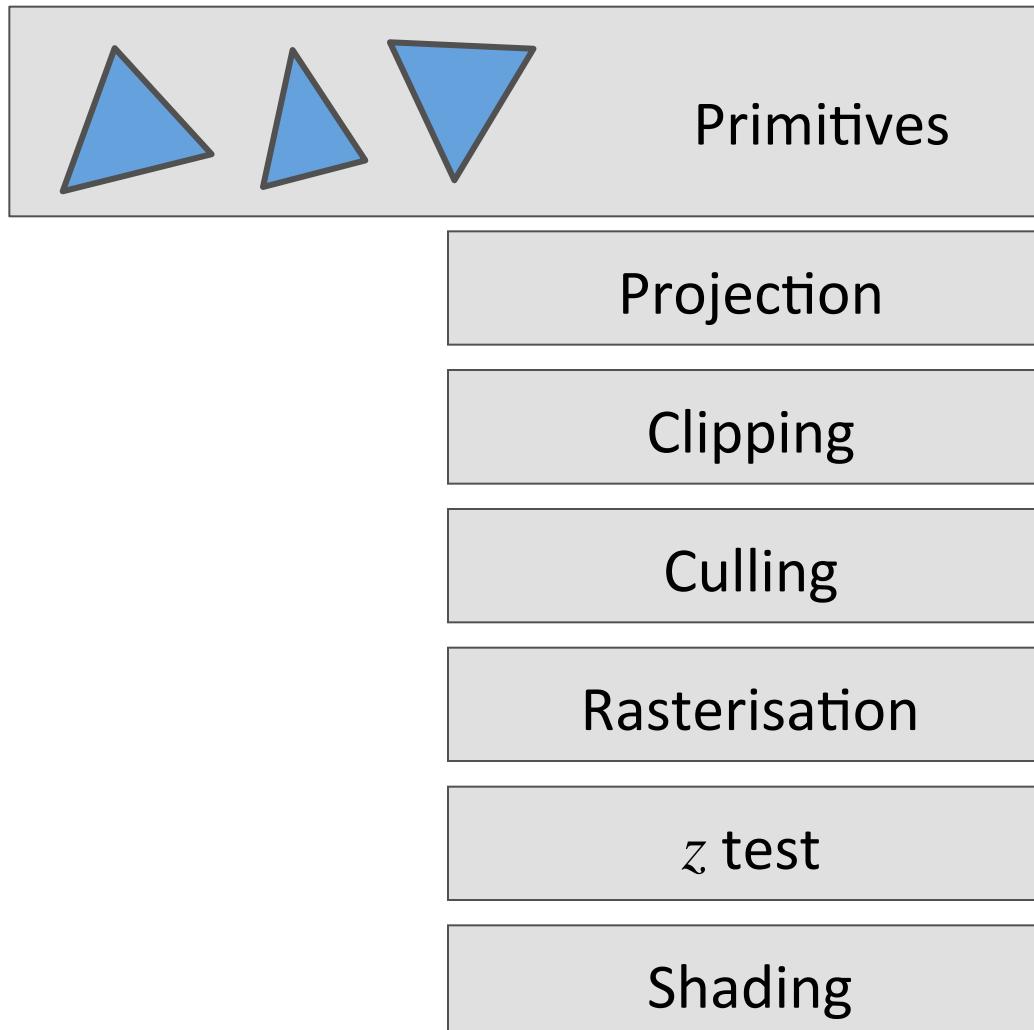


Visibility

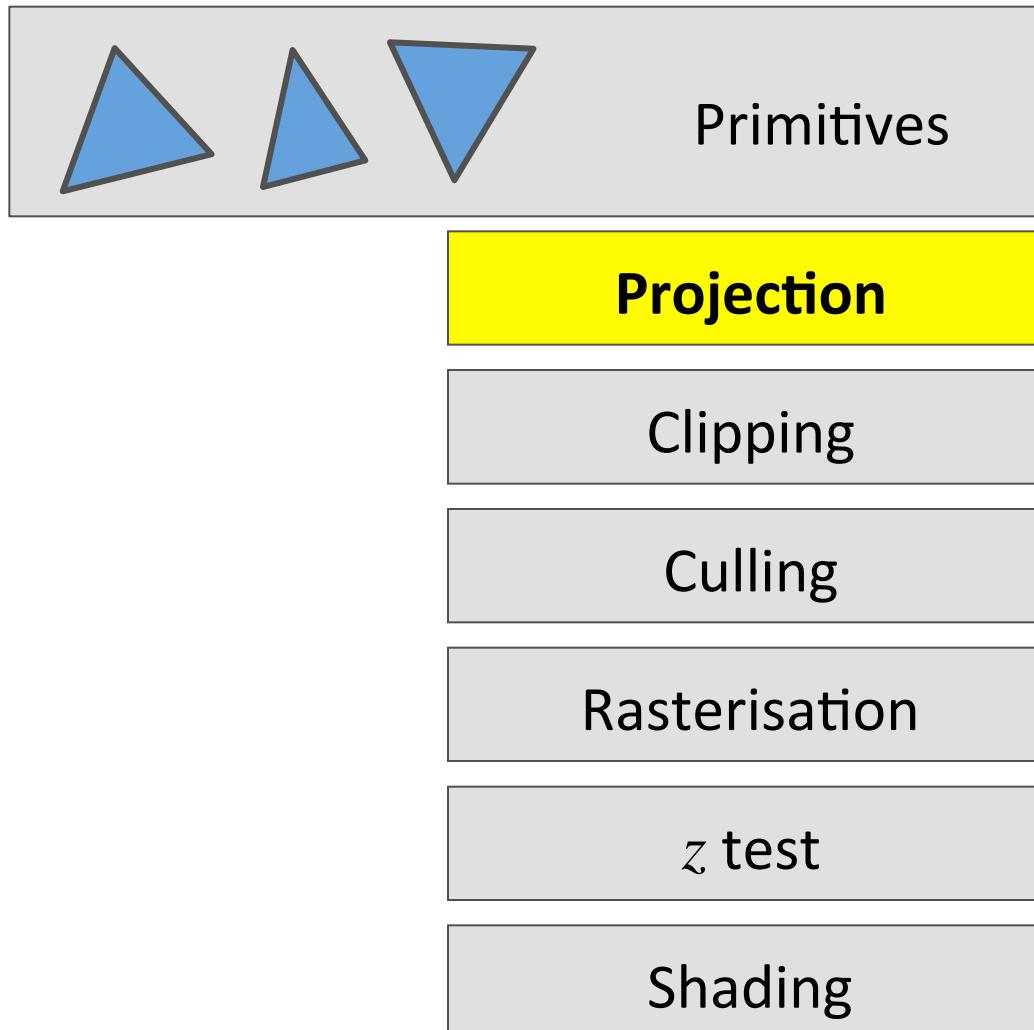


Shading

Pipeline

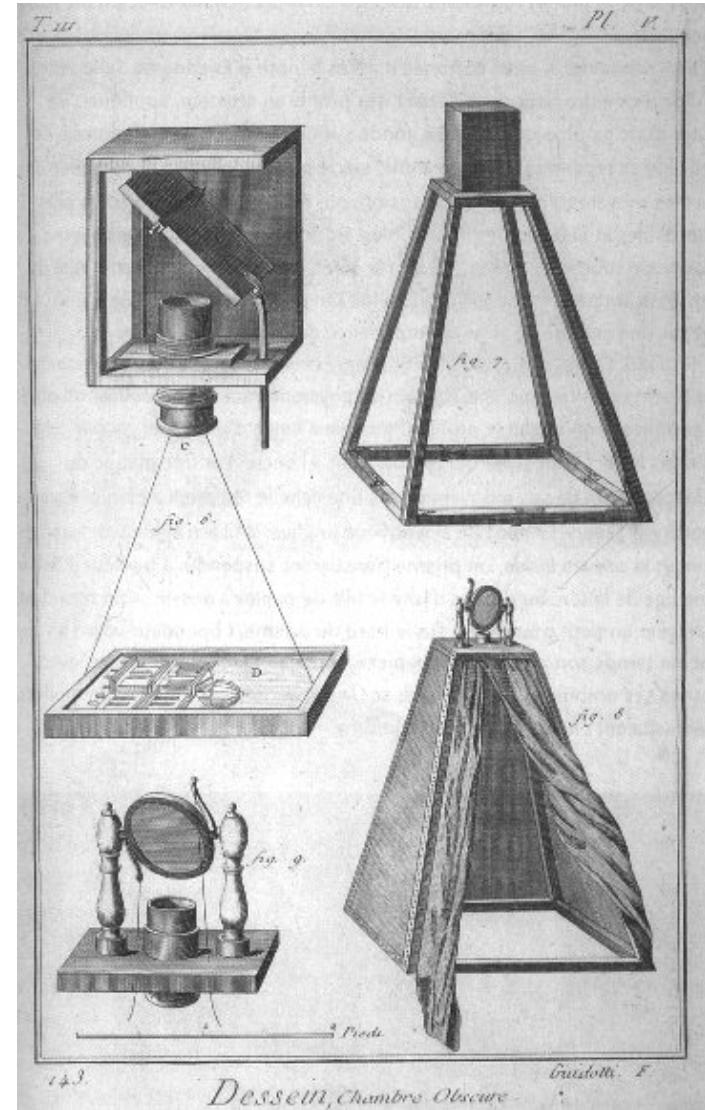
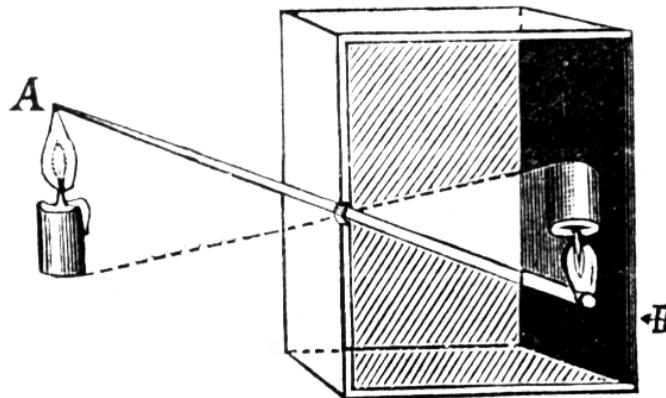


Pipeline



History of projection

- Camera Obscura
 - Earliest form of projection
 - Mo-Ti (470-390 BC)
 - Aristotle (384-322 BC)



History of Projection

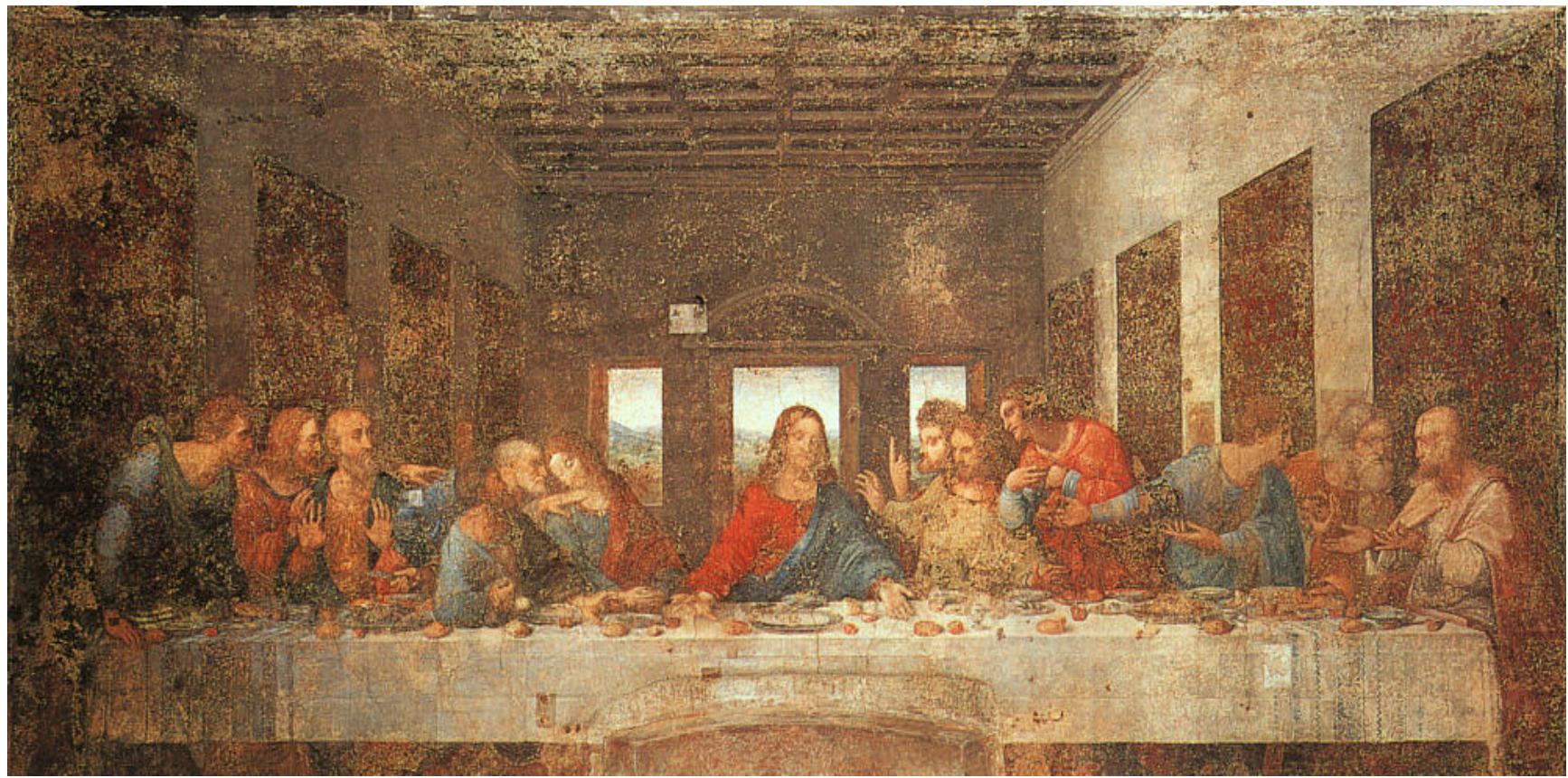
- Ancient Greeks knew the laws of perspective
- Renaissance: perspective is adopted by artists



Duccio c. 1308

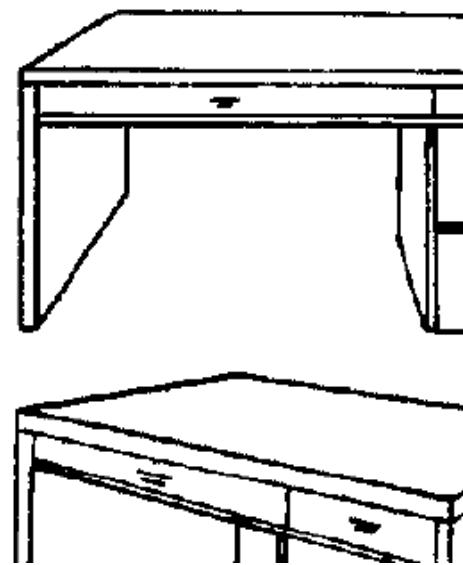
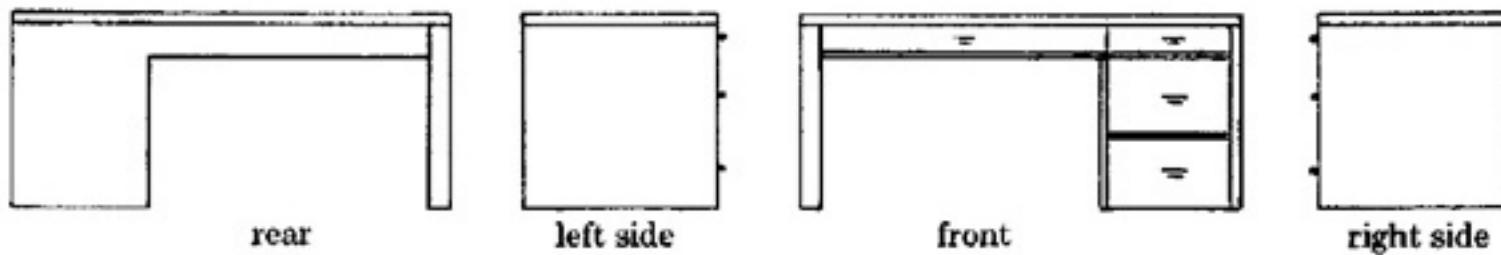
History of Projection

- High Renaissance: perspective formalized

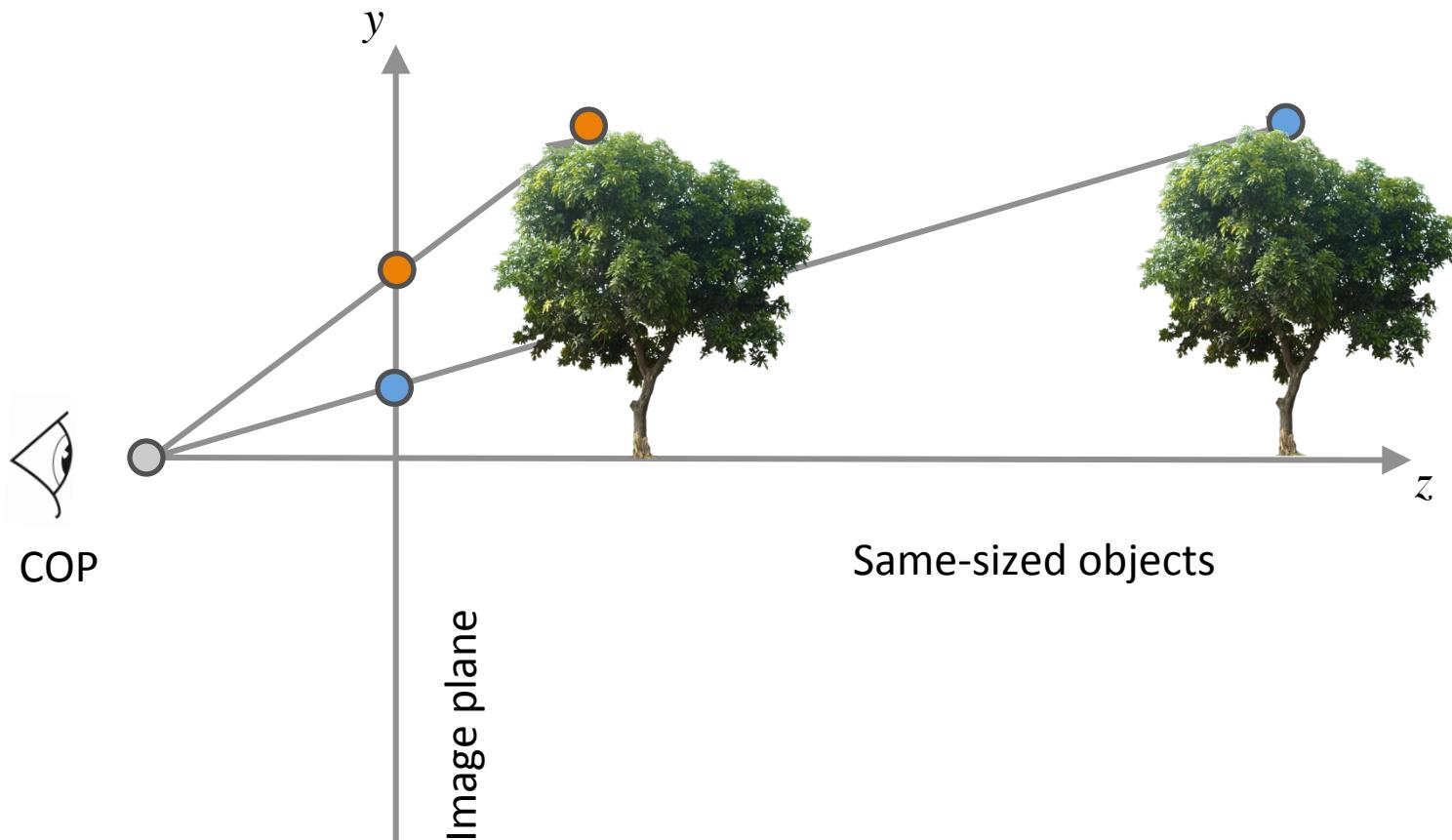


da Vinci c. 1498

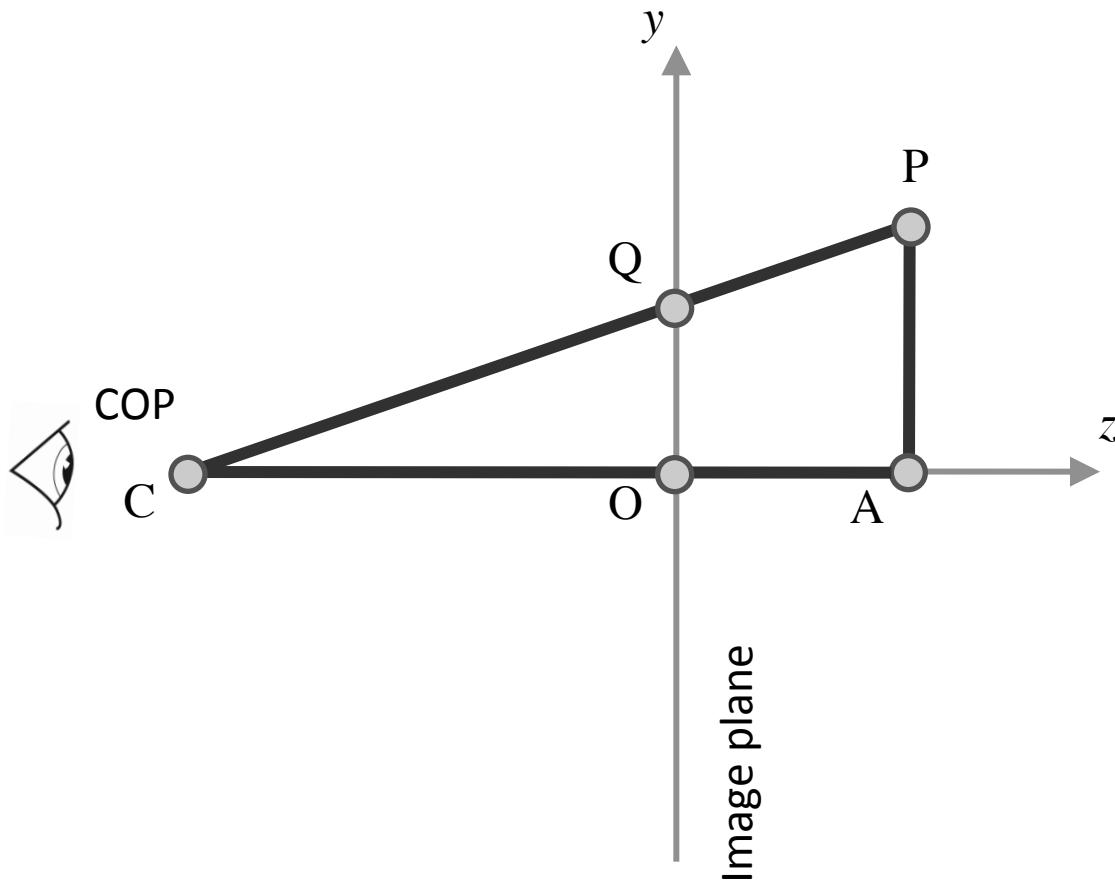
Orthographic vs. Perspective



Perspective projection



Perspective projection



We know:
 $QO/CO = PA/CA$

For x , we define

$$x' = QO$$

$$x = PA$$

$$d = CO$$

$$z = AO$$

$$x'/d = x / (d + z) \text{ so}$$
$$x' = x d / (d + z)$$

E.g., for $d = 1$:

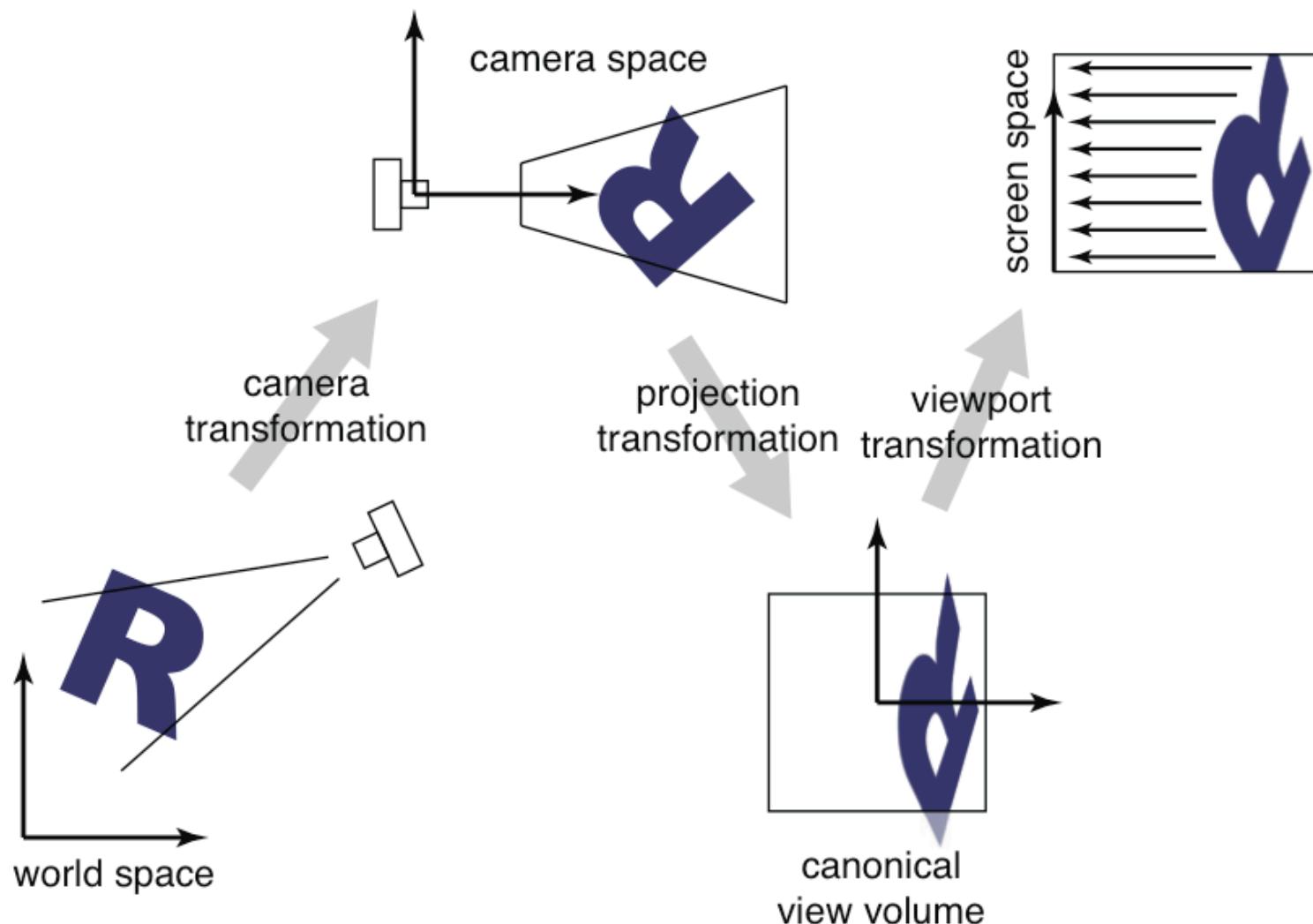
$$x' = x / (z + 1)$$

Perspective division

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

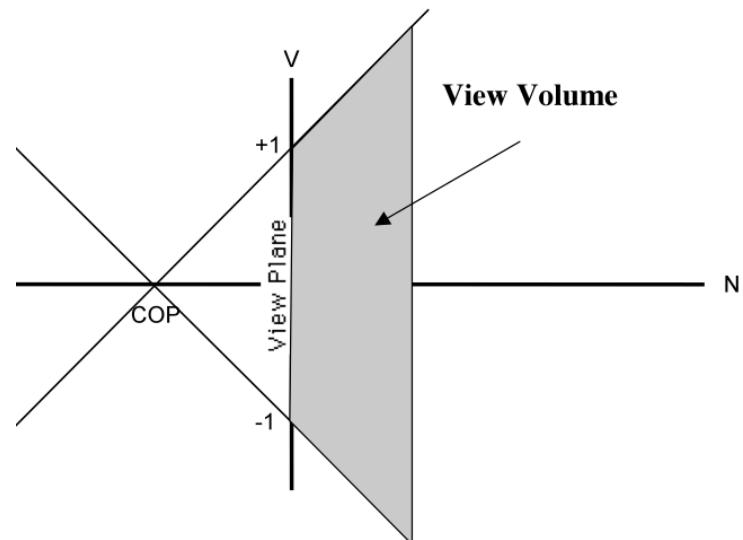
- Given a point $(x, y, z, 1)$
- Another good reason for 4x4 matrices
- Its transform is $(x, y, z, z+1) = \left(\frac{x}{z+1}, \frac{y}{z+1}, \frac{z}{z+1}, 1 \right)$

Projection Pipeline



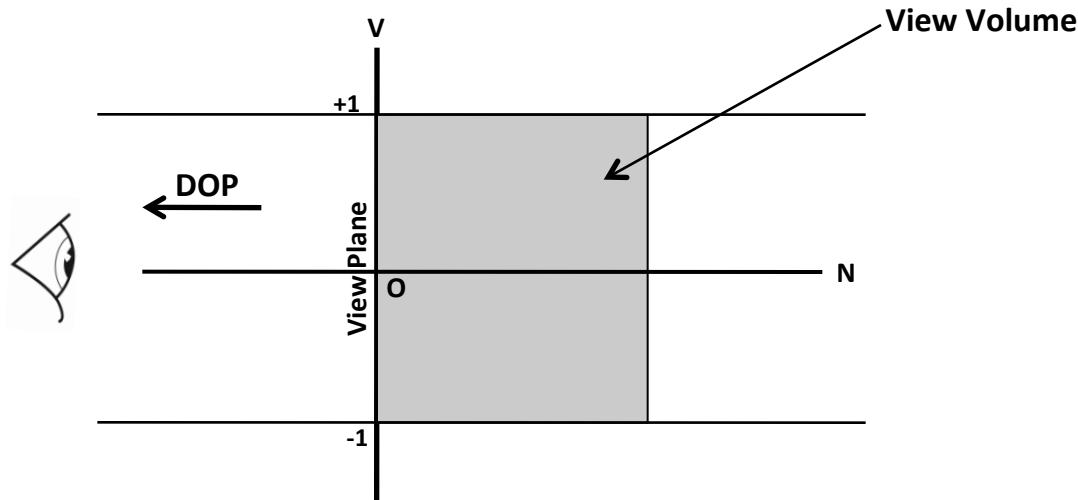
Canonical Frames

- We use *canonical frames* as intermediate stages from which we know how to proceed
- Canonical Frame for Perspective Projection:
 - COP at $(0, 0, -1)$
 - View plane coincident with UV plane
 - Viewplane window bounded by -1 to $+1$



Canonical Frame for Parallel Projection

- Orthographic parallel projection
- Direction of projection (DOP) is $(0,0,-1)$
- View volume bounded by -1 and $+1$ on U and V
- And by 0 and 1 on the N axis
- $p' = (x, y, 0)$

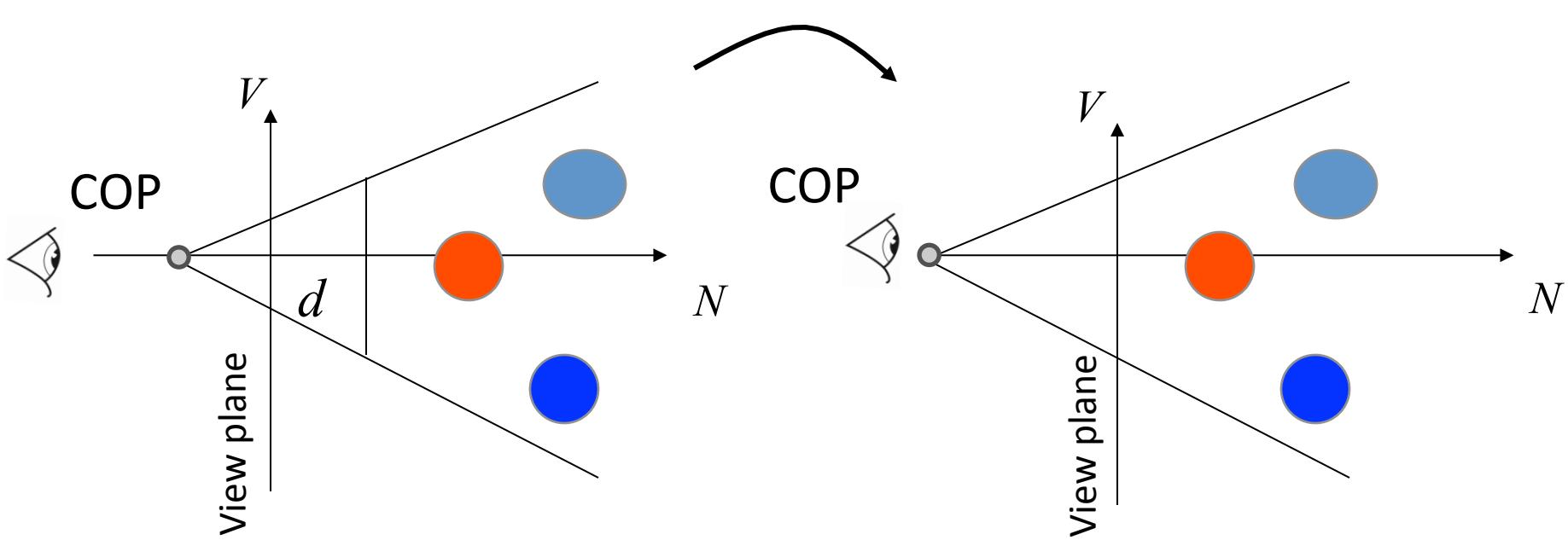


General Persp. to Canonical Persp.

- We will apply three transformation matrices
- Each ‘corrects’ one aspect of the projection
- Finally, we multiply them all together

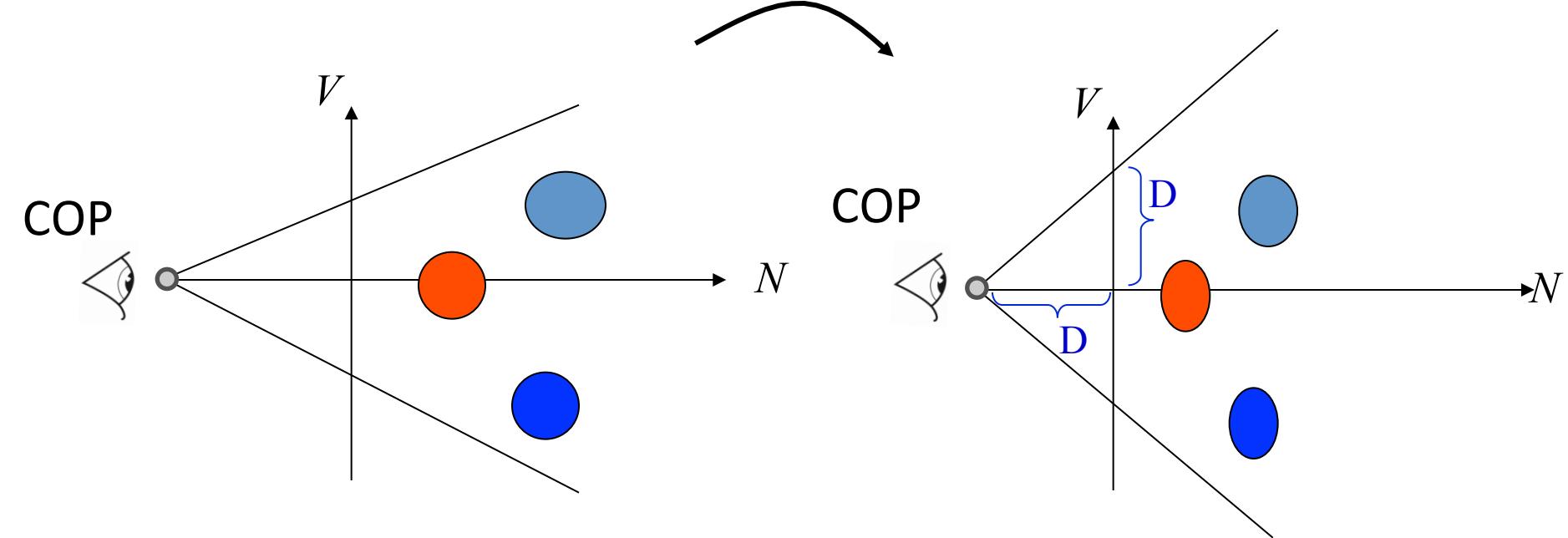
Step 1: Move to UV plane ($n = 0$)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -d & 1 \end{pmatrix}$$



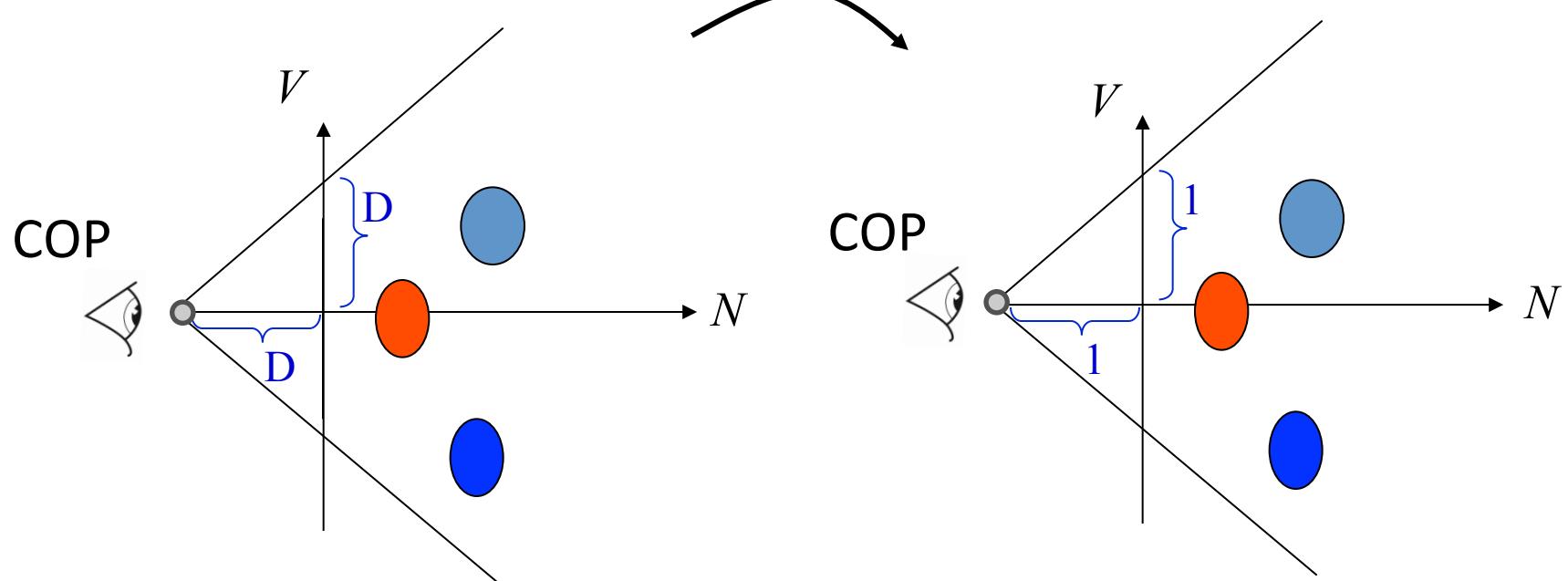
Step 2: Regular pyramid

$$\begin{pmatrix} 2D/dx & 0 & 0 & 0 \\ 0 & 2D/dy & 0 & 0 \\ -px/dx & -py/dy & 1 & 0 \\ -(px/dx)D & -(py/dy)D & 0 & 1 \end{pmatrix} \quad \begin{aligned} D &= d - c_z \\ dx &= x_2 - x_1 \\ dy &= y_2 - y_1 \\ px &= x_2 + x_1 \\ py &= y_2 + y_1 \end{aligned}$$



Step 3: Scale by $1/D$

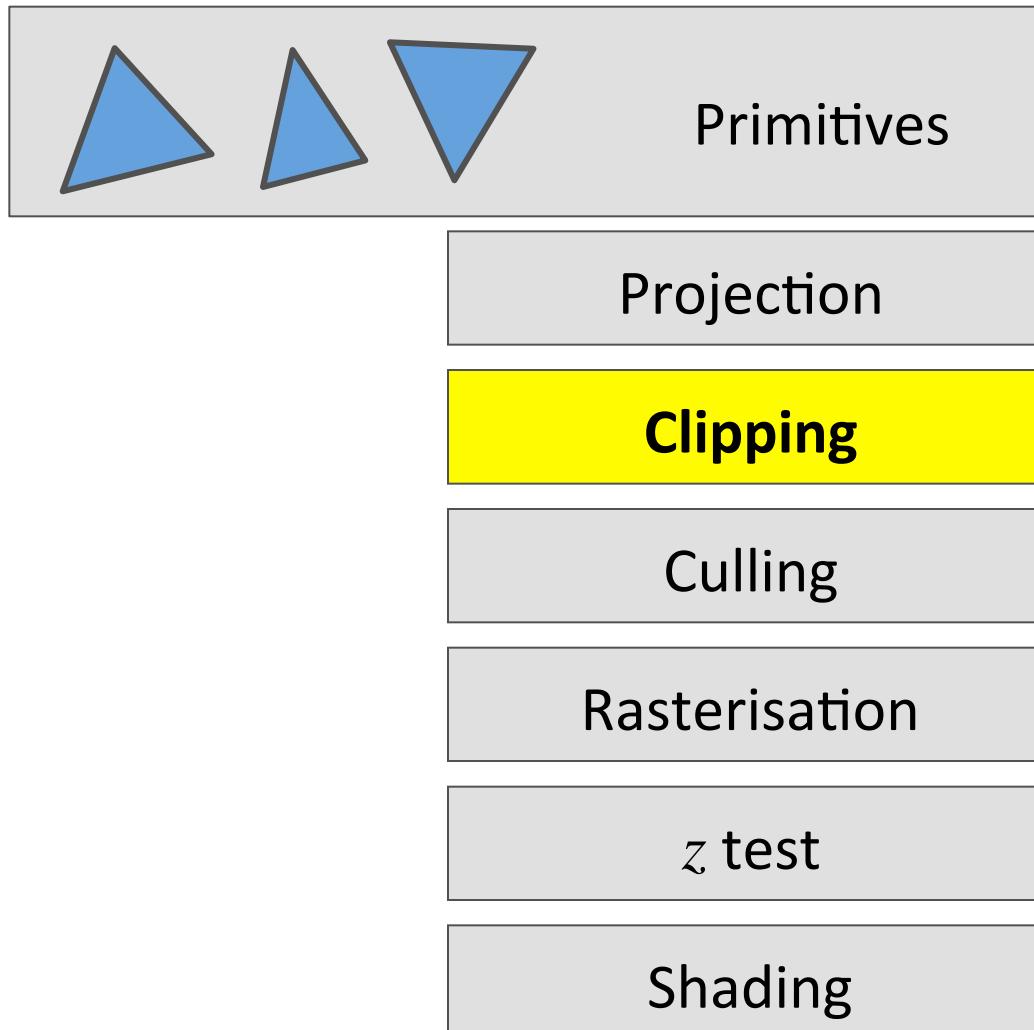
$$\begin{pmatrix} \frac{1}{D} & 0 & 0 & 0 \\ 0 & \frac{1}{D} & 0 & 0 \\ 0 & 0 & \frac{1}{D} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Two simplifications

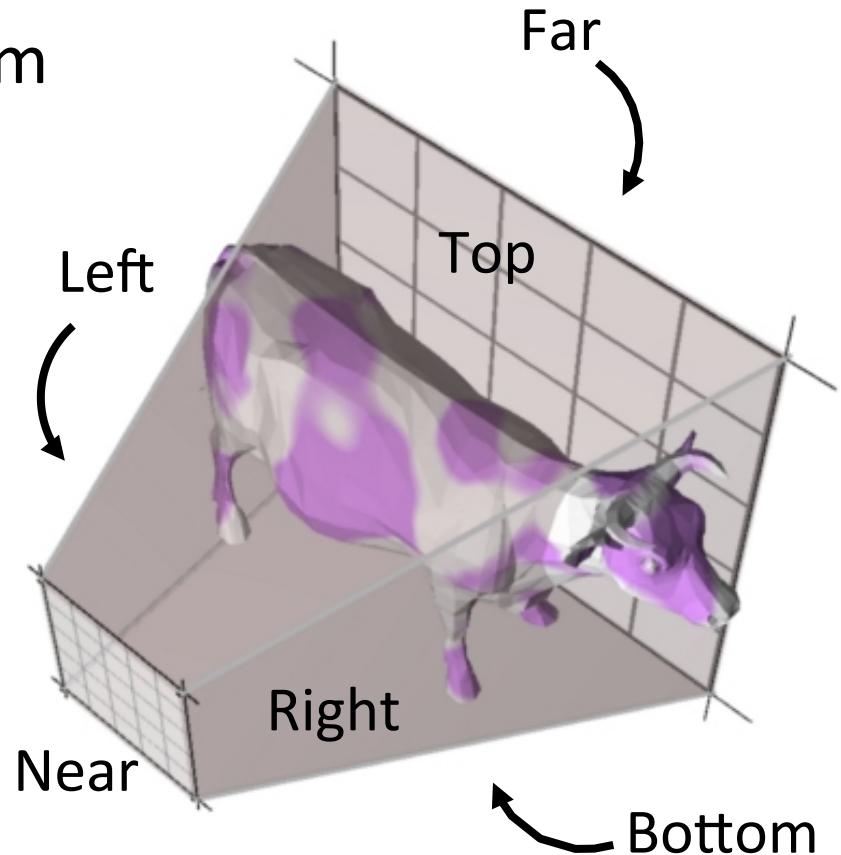
1. The COP might not fall on the z axis
 - Required in professional photography
 - Required in virtual reality (Stereo, HMDs)
2. Might want to re-scale z to fit a range
 - Will get back to this adjustment later

Pipeline



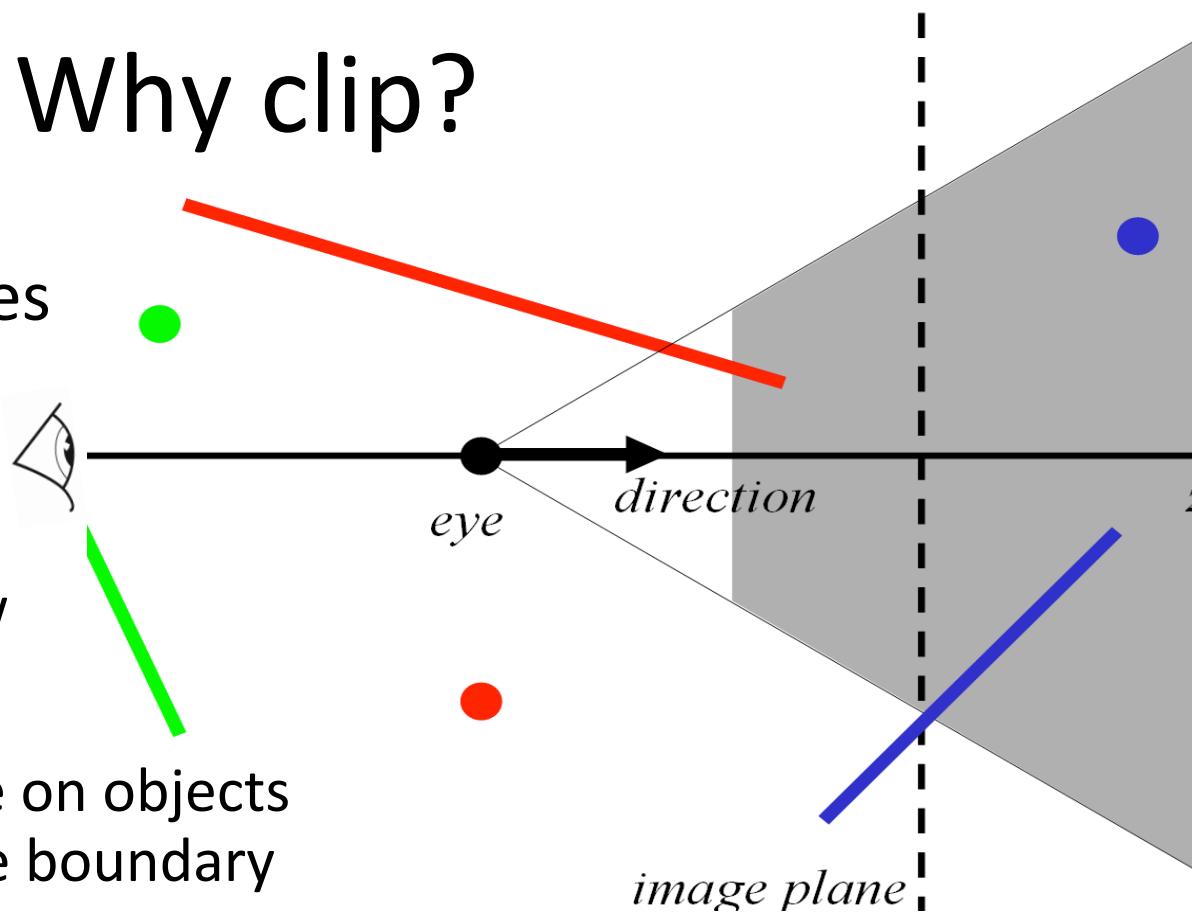
Clipping

- Eliminate portions of objects outside the viewing frustum
- View frustum
 - Boundaries of the image plane projected in 3D
 - A near & far clipping plane
- User may define additional clipping planes



Why clip?

- Avoid degeneracies
 - Don't draw stuff behind the eye
 - Avoid division by 0 and overflow
- Efficiency
 - Don't waste time on objects outside the image boundary
- Other graphics applications (often non-convex)
 - Hidden-surface removal, Shadows, Picking, Binning, CSG (Boolean) operations (2D & 3D)

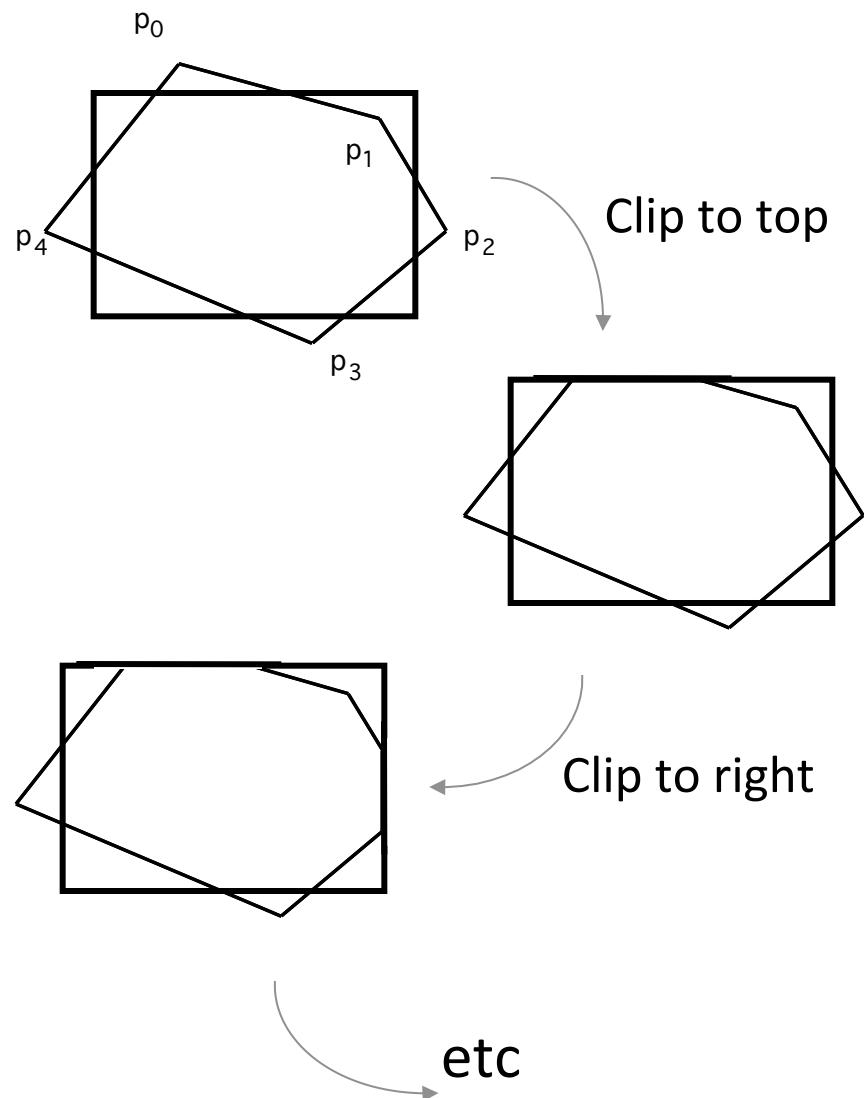


Clipping Summary

- It's the process of finding the exact part of a polygon lying inside the view volume
- To maintain consistency, clipping of a polygon should result in a polygon, not a sequence of partially unconnected lines
- We will first look at 2 different 2D solutions and then extend one to 3D

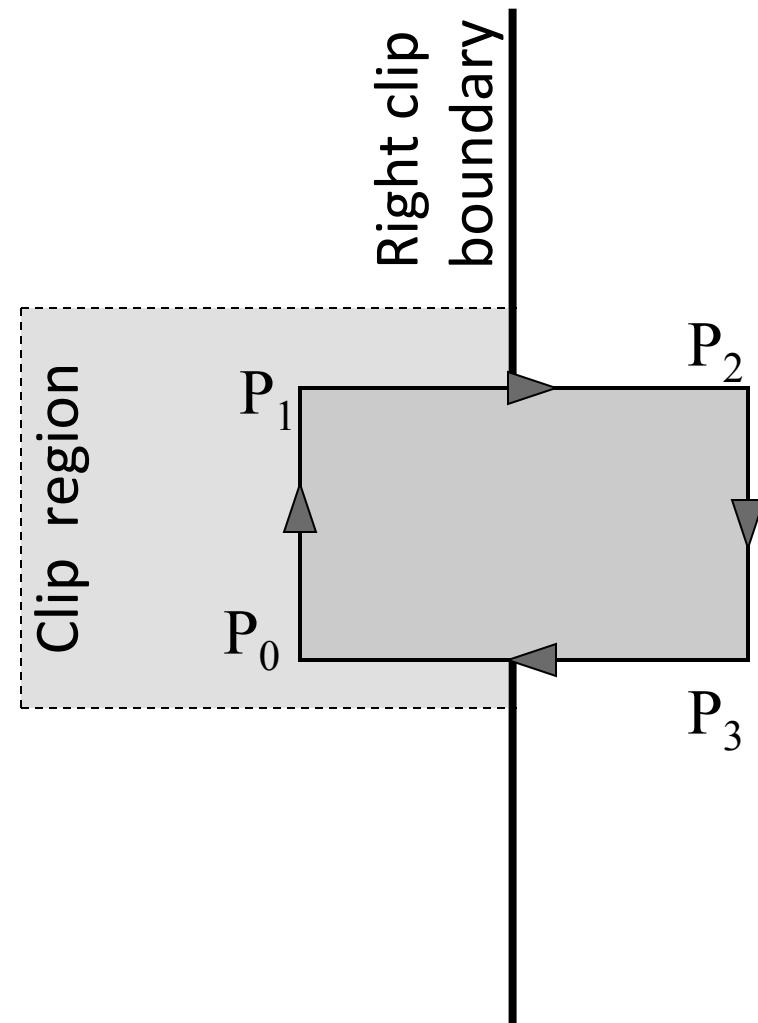
Sutherland-Hodgman Algorithm

- Clip the polygon against each boundary of the clip region successively
- Result is possibly NULL if polygon is outside
- Can be generalised to work for any polygonal clip region, not just rectangular



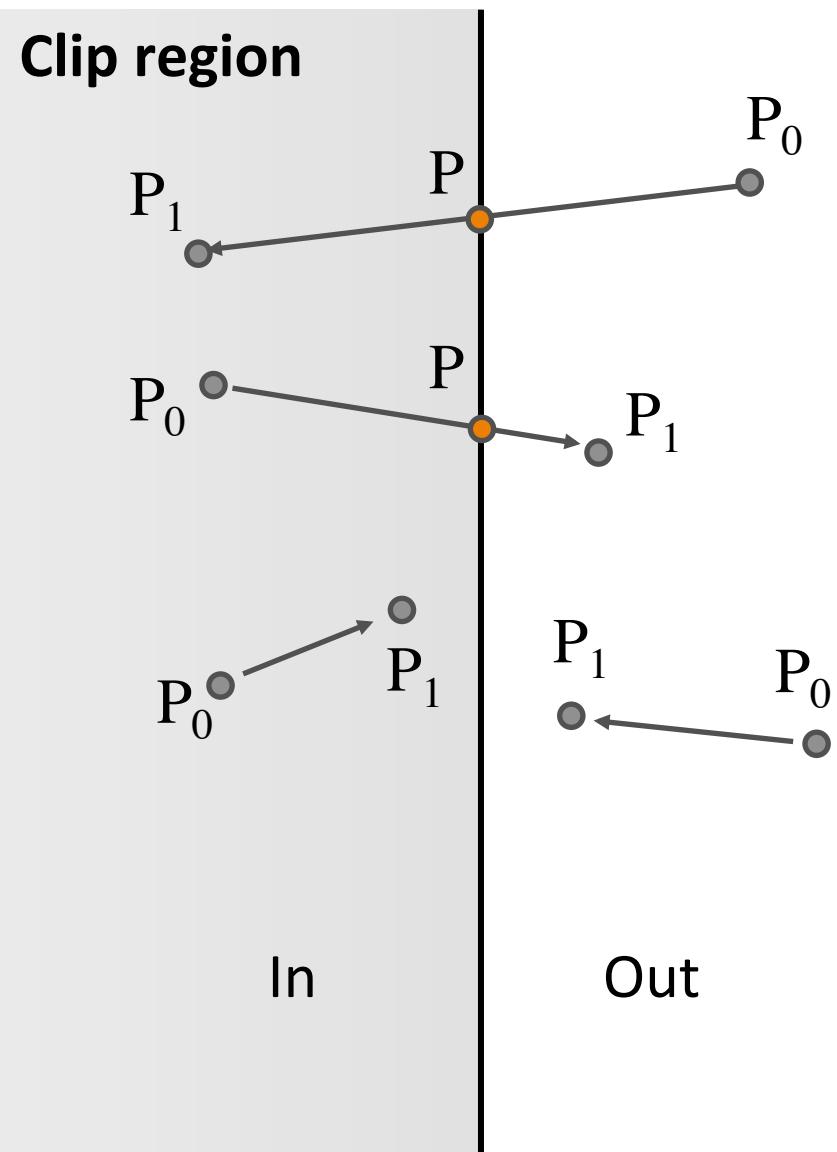
Clipping To A Region

- To find the new polygon
 - iterate through each of the polygon edges and construct a new sequence of points
 - starting with an empty sequence
 - for each edge there are 4 possible cases to consider



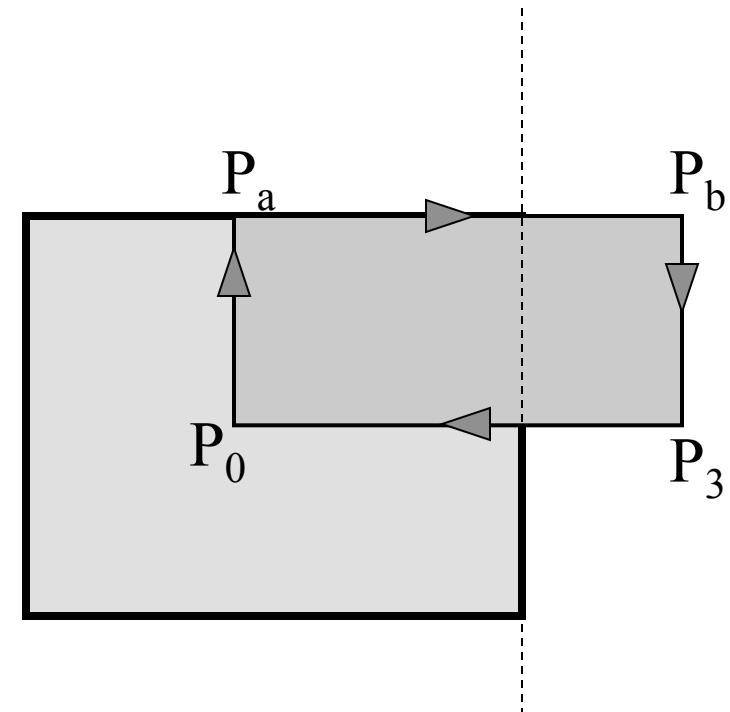
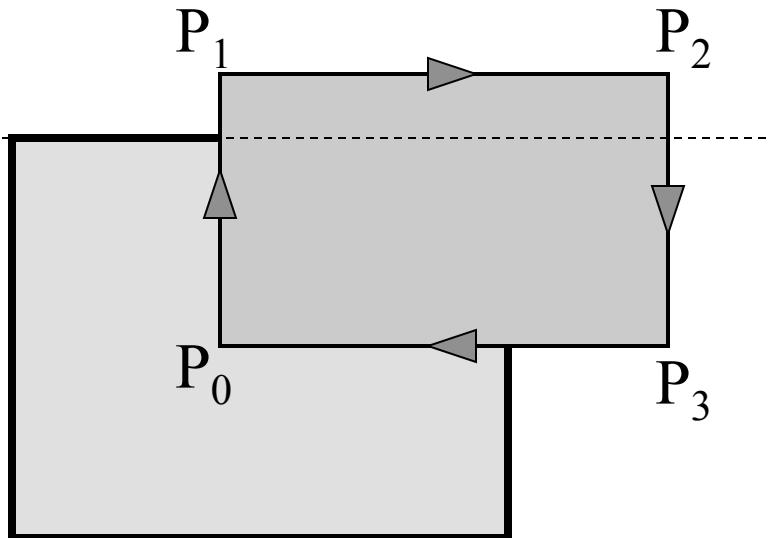
Clipping polygon edge against boundary

- Given an edge P_0, P_1 we have 4 cases. Can ...
 - enter the clip region, add P and P_1
 - leave the region, **add** only P
 - be entirely outside, do nothing
 - be entirely inside, **add** only P_1
- Where P is the point of intersection



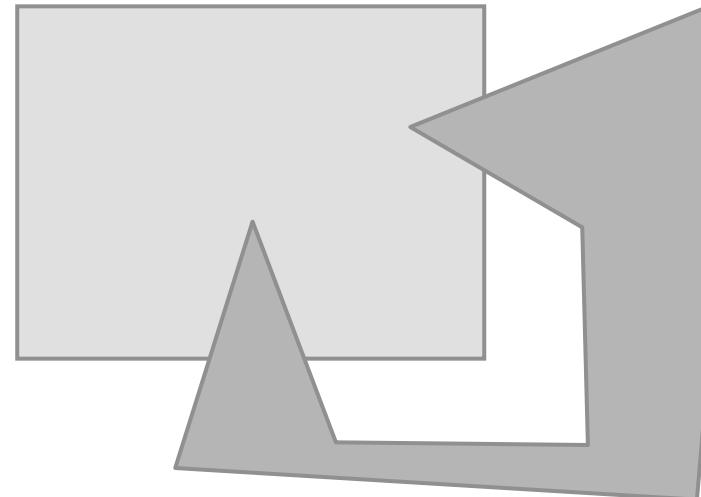
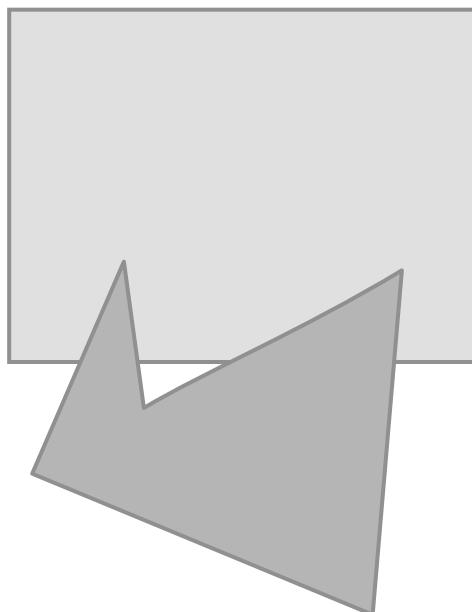
Still the Sutherland-Hodgman

- We can determine which of the 4 cases and also the point of intersection with just if statements
- Example:



Sutherland-Hodgman Problem

- Clipping is an example where concave polys make our lives easier
- Concave: Weiler-Atherton algorithm



Clipping Polygons in 3D

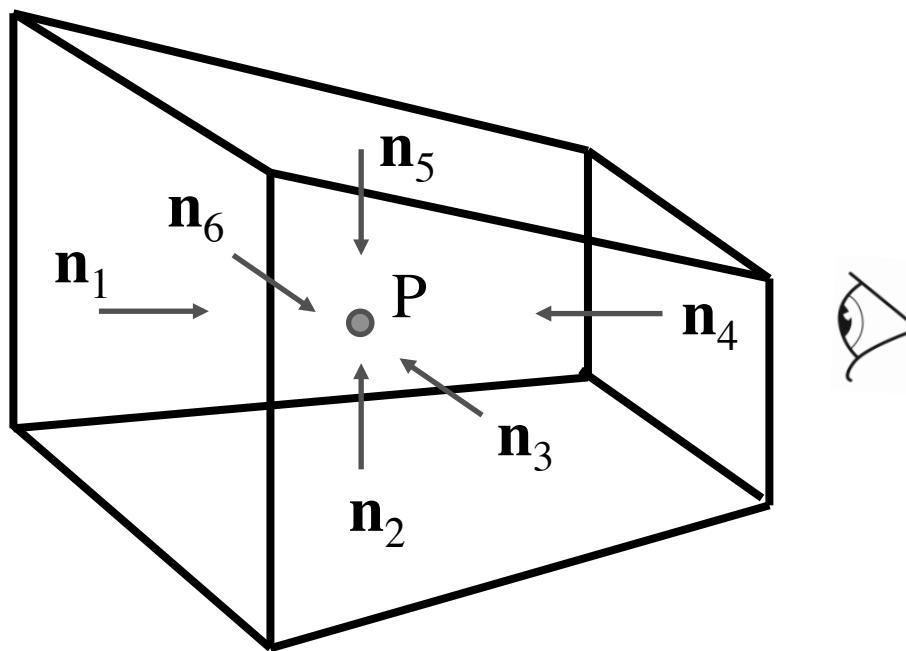
- The Sutherland-Hodgman can easily be extended to 3D
 - Clipping boundaries are 6 planes instead of 4 lines
 - Intersection calculation is done by comparing an edge to a plane instead of edge to edge

When to clip?

- Before perspective transform in 3D space
 - Use the equation of 6 planes
 - Natural, not too degenerate
- In homogeneous coordinates after perspective transform (Clip space)
 - **Before** perspective divide (4D space, weird w values)
 - Canonical, independent of camera
 - The simplest to implement in fact
- In the transformed 3D screen space after perspective division (canonical par.)
 - Problem: objects in the plane of the camera

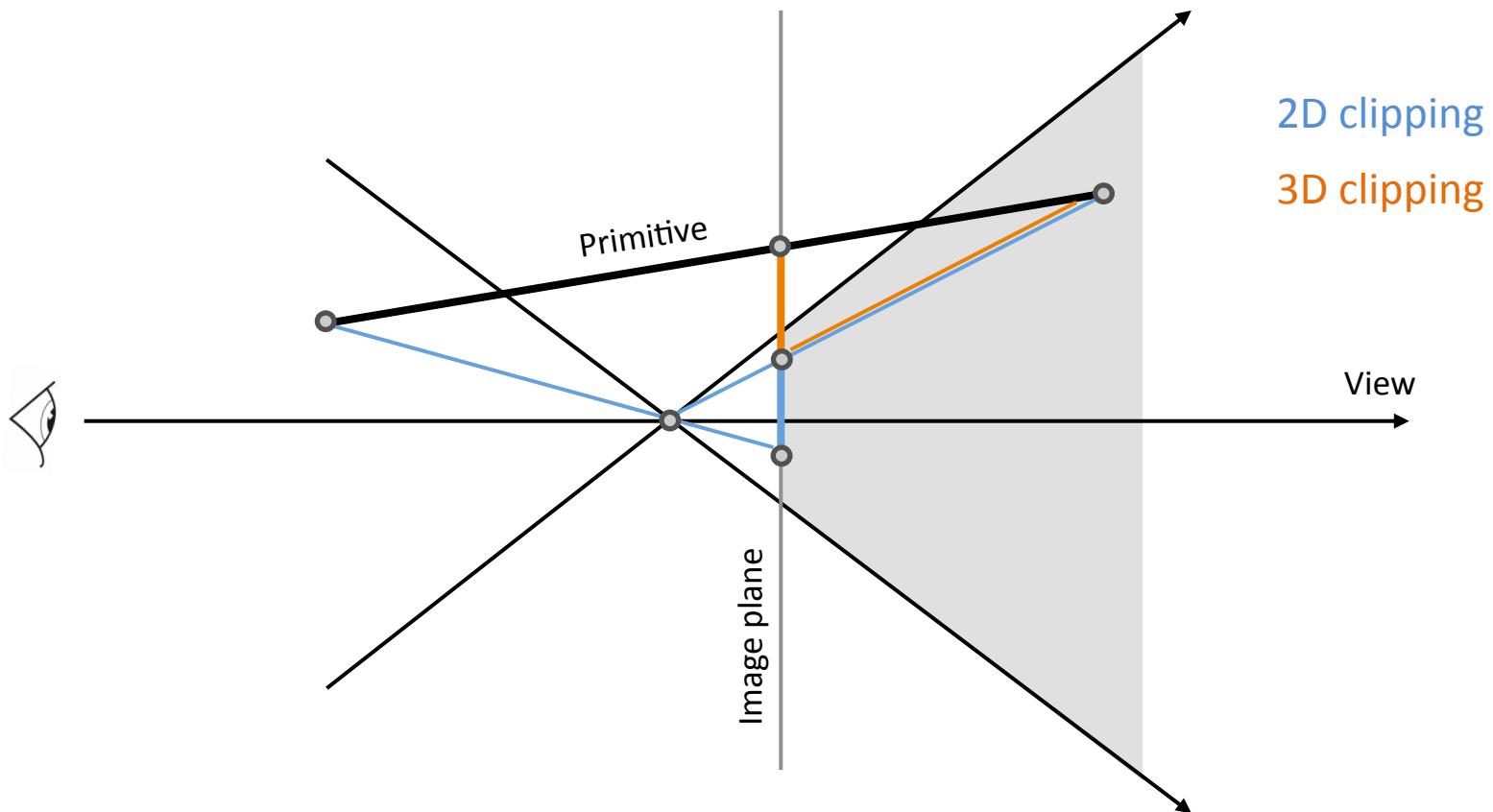
Clipping with respect to View Frustum

- Test against each of the 6 planes
 - Normals oriented towards the interior
- Clip / cull / reject point P if any $\langle \mathbf{n}_i, \mathbf{p} \rangle < d_i$

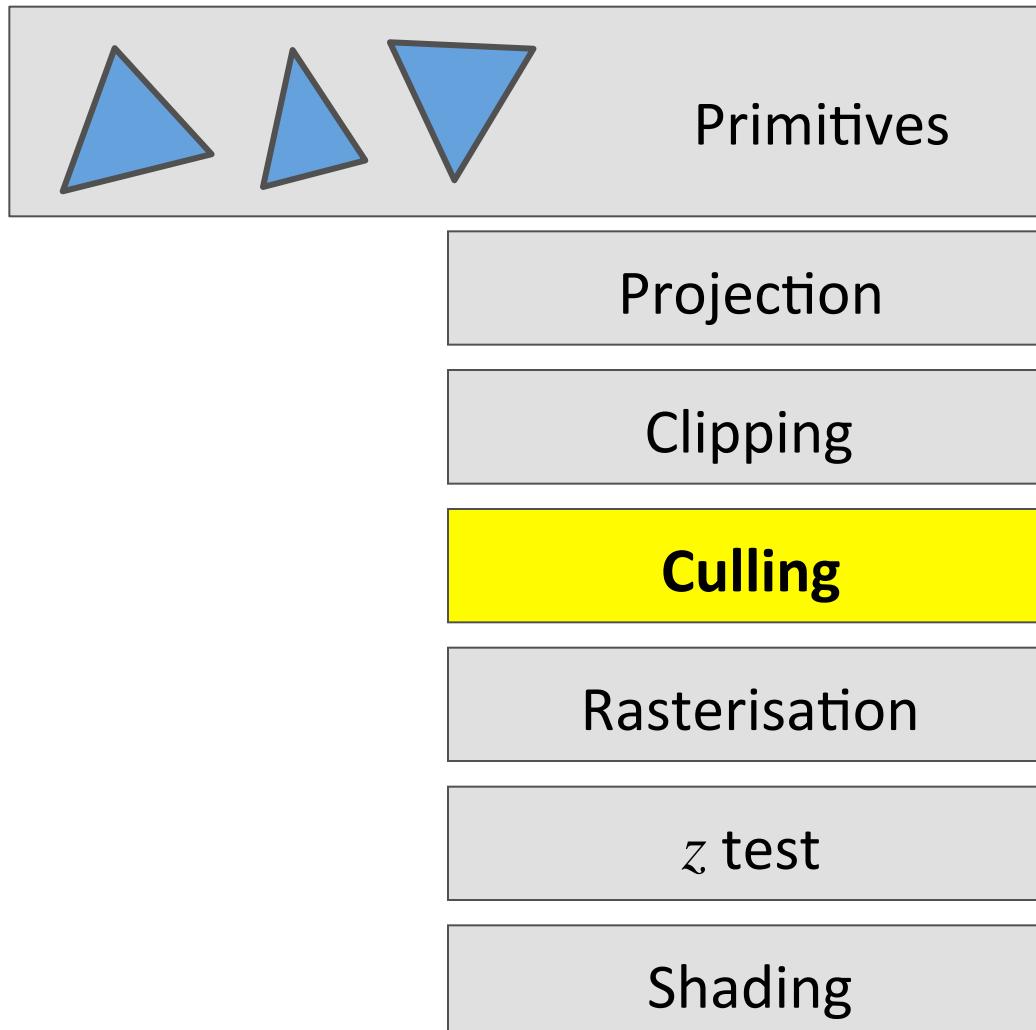


Clipping After Projecting

- When we have an edge that extends from the front to behind the COP, then if we clip after projection (which in effect is what e.g. the PS 1 did) we might get wrong results

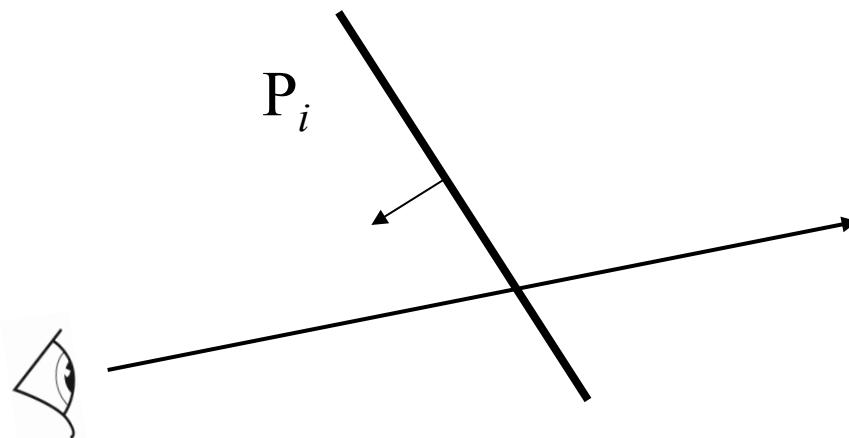


Pipeline



Idea

- Optional speed-up
- Remove primitives not facing the camera
- Polygons whose normal does not face the viewpoint, are not rendered



Back Face Culling

- Thus if we have the plane equation

$$l(x, y, z) = ax + by + cz - d = 0$$

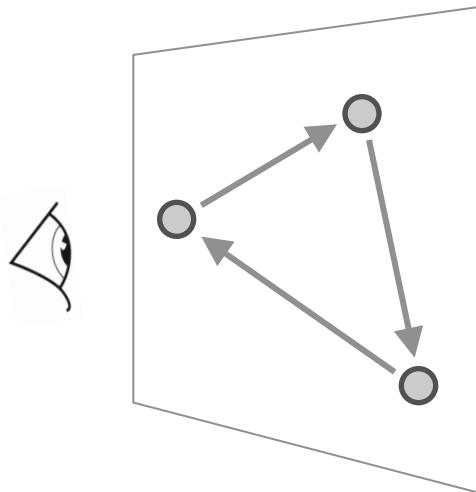
- and the COP (c_x, c_y, c_z), then

$$l(c_x, c_y, c_z) > 0$$

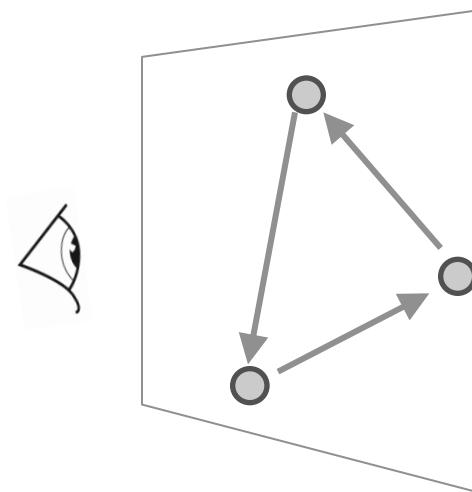
- if the COP is in front of the polygon
- Otherwise: cull

Clockwise/counter-clockwise

Typically done without normals: All polygons that are not counter-clockwise after projection are culled



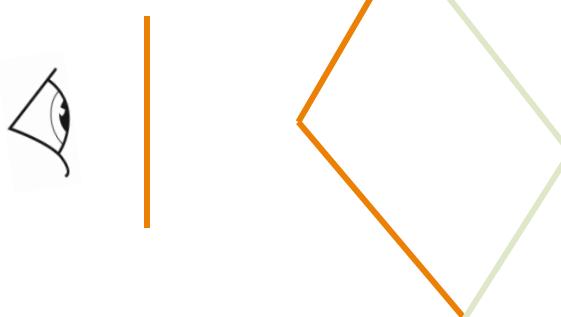
Clockwise: **Culled**



Counter clockwise: **Not culled**

Limitation

Does not ensure correct visible surfaces determination, unless there is only a single convex object



Convex: **Works**

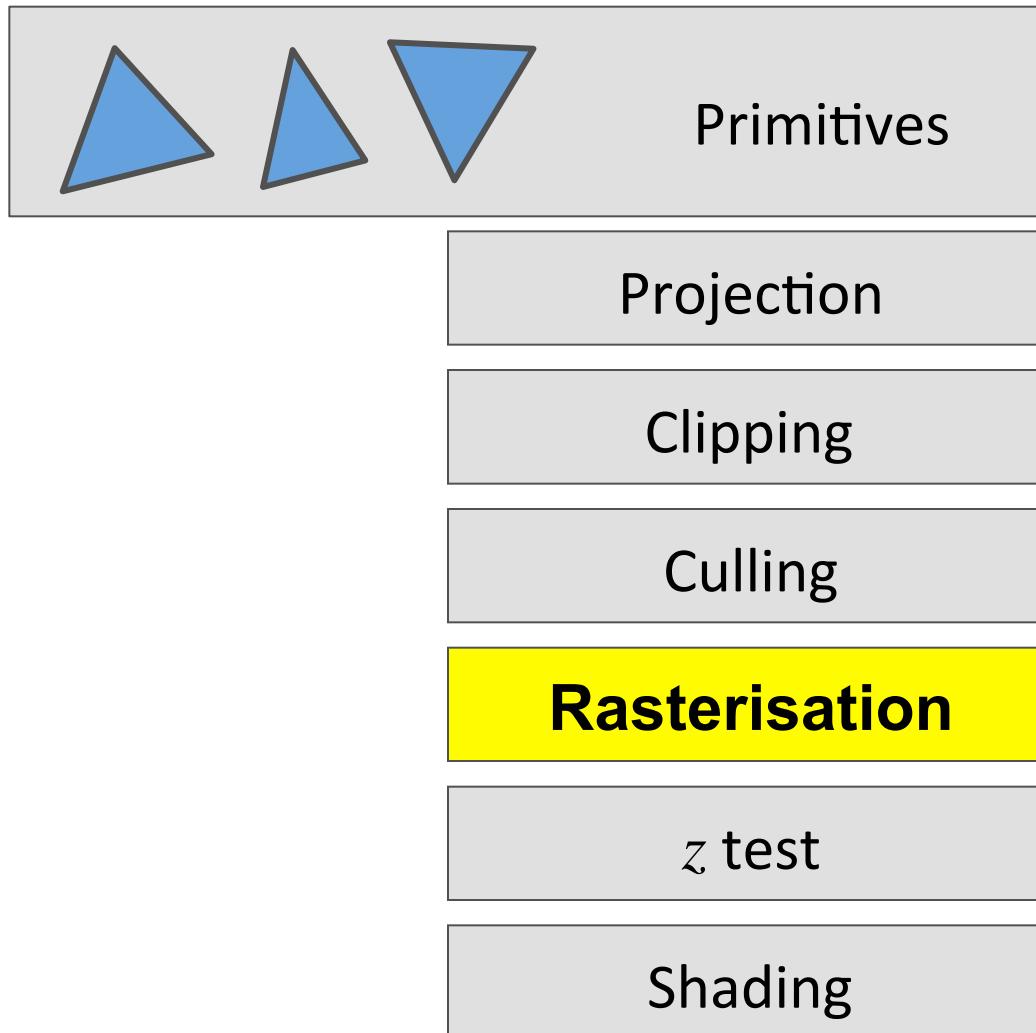


Concave: **Does not work**

Culling recap

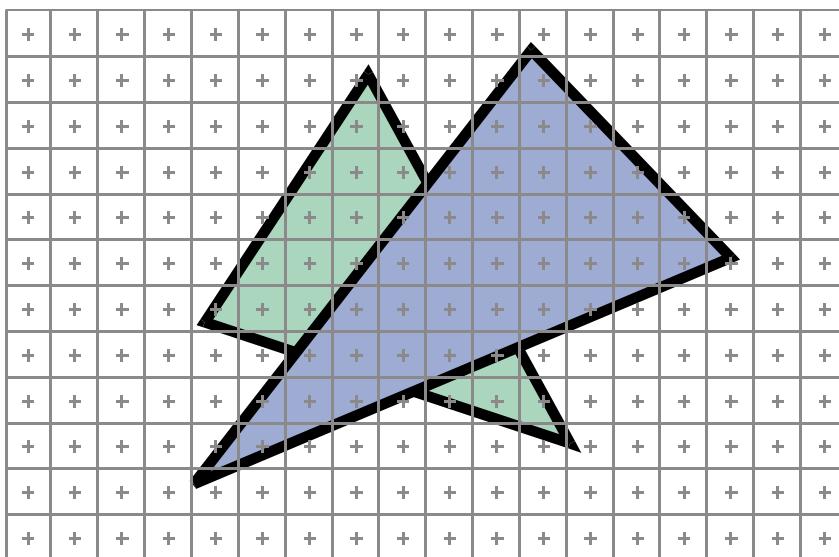
- Culling can easily & quickly remove (some!) invisible polygons from the pipeline
- Still some (partially) invisible ones remain

Pipeline



2D Scan Conversion

- Primitives are continuous; screen is discrete
 - Well, triangles are described by a discrete set of vertices
 - But they describe a continuous area on screen



2D Scan Conversion

- Solution: compute discrete approximation
 - Scan Conversion (Rasterization):
algorithms for efficient generation of the
samples comprising this approximation

Naïve Filling Algorithm

- Find a point inside the polygon
- Do a **flood fill**:
 - Keep a stack of points to be tested
 - When the stack is not empty
 - Pop the top point (Q)
 - Test if Q is inside or outside
 - If Inside, colour Q , push neighbours of Q if not already tested
 - If outside discard
 - Mark Q as tested

Critique

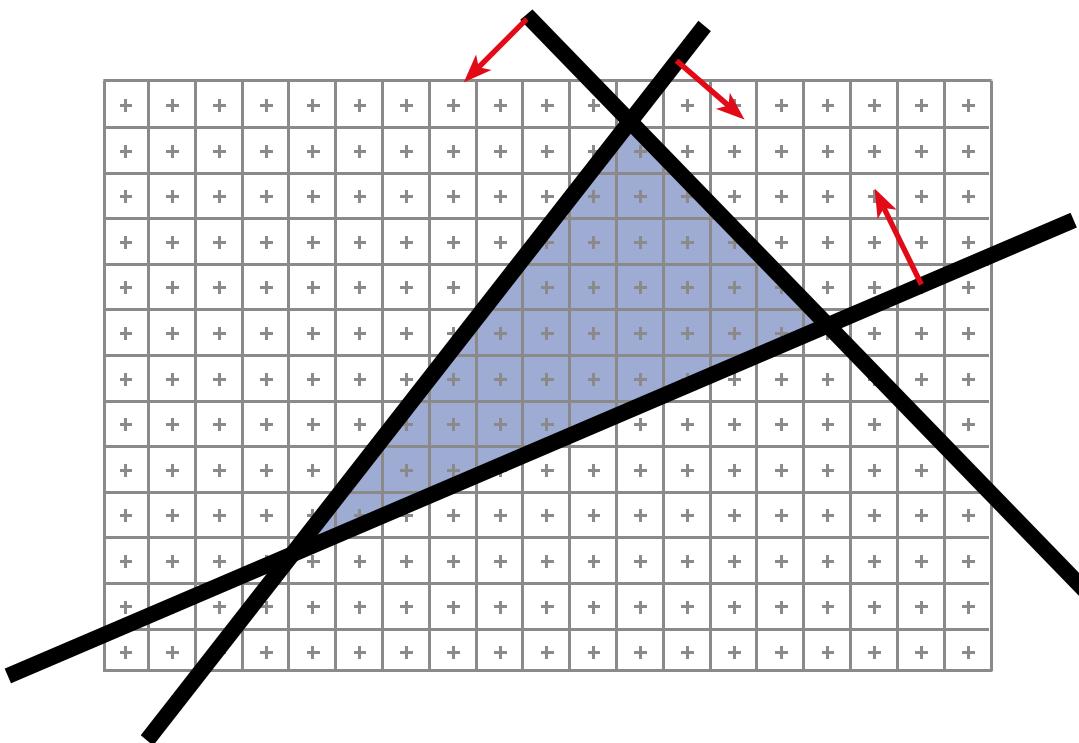
- Horribly slow
 - Explicit in/out test at every point
 - But still very common in paint packages!
 - Only solid color
- Stack might be very deep
- Want to do this
 - Without much memory
 - In parallel

Brute Force Solution for Triangles

For all pixels i

For all edges j

Test if i is inside j



Half-Space Test Reminder

- For each edge compute line equation (analogue to plane equation):

$$L_i(x, y) = a_i x + b_i y + c_i$$

- If $L_i(x, y) > 0$ point in **positive** half-space
- If $L_i(x, y) < 0$ point in **negative** half-space
- If all $L_{1,2,3}(x, y) \geq 0$ point inside triangle

Half-Space Test Reminder

- For each edge compute line equation (analogue to plane equation):

$$L_i(x, y) = a_i x + b_i y + c_i$$

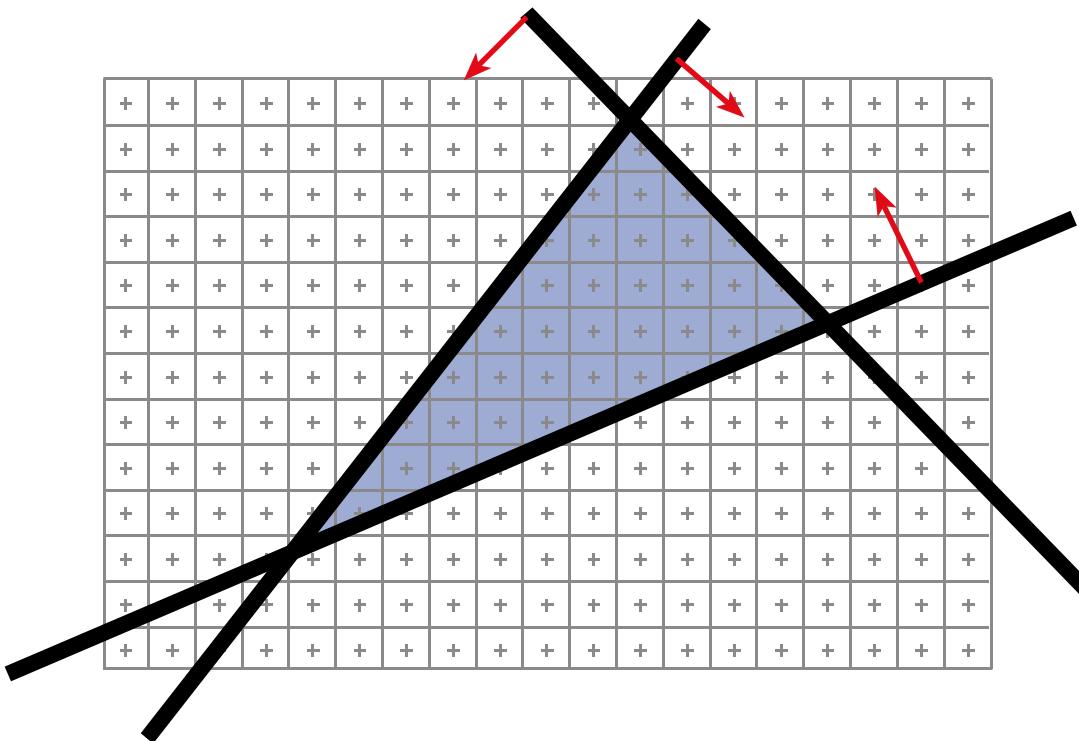
- Example:
 - Assuming $(2, 2)$ to $(4, 7)$,
 - The slope is $dy/dx = 5/2 = 2.5$
 - Hence $y - 7 = 2.5(x - 4)$,
 - So $L_1(x, y) = 2.5x - y - 3.$

Brute Force Solution for Triangles

For all pixels i

For all edges j

Test if i is inside j

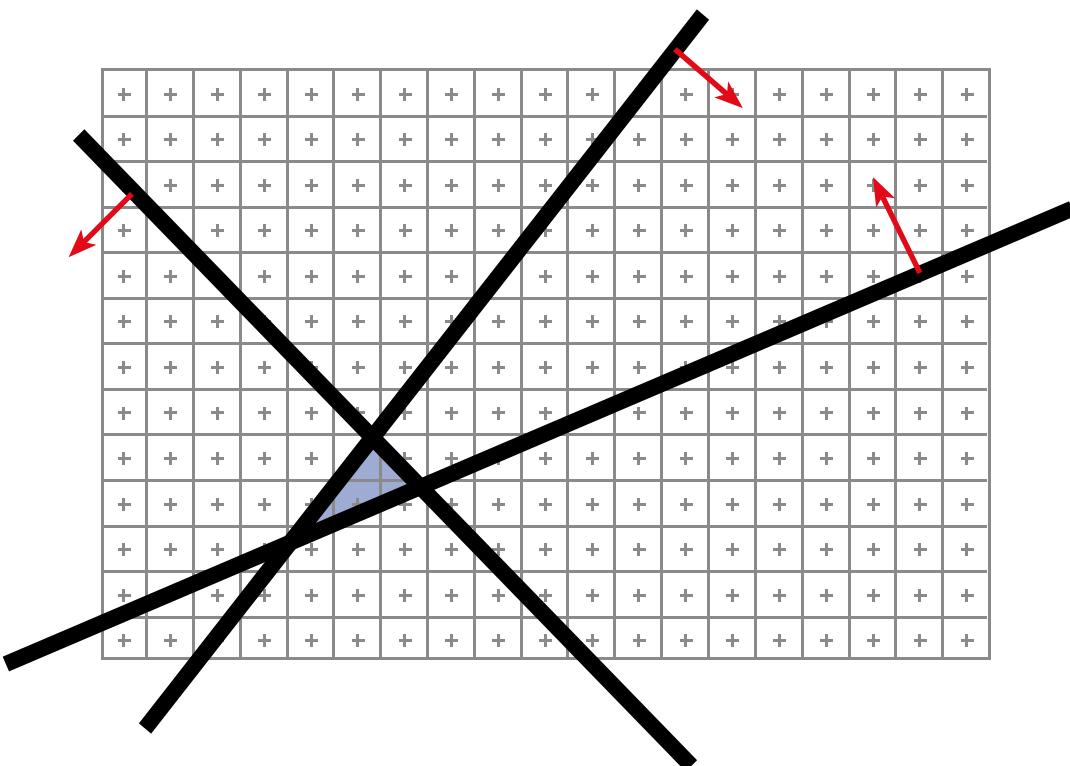


Brute Force Solution for Triangles

For all pixels i

For all edges j

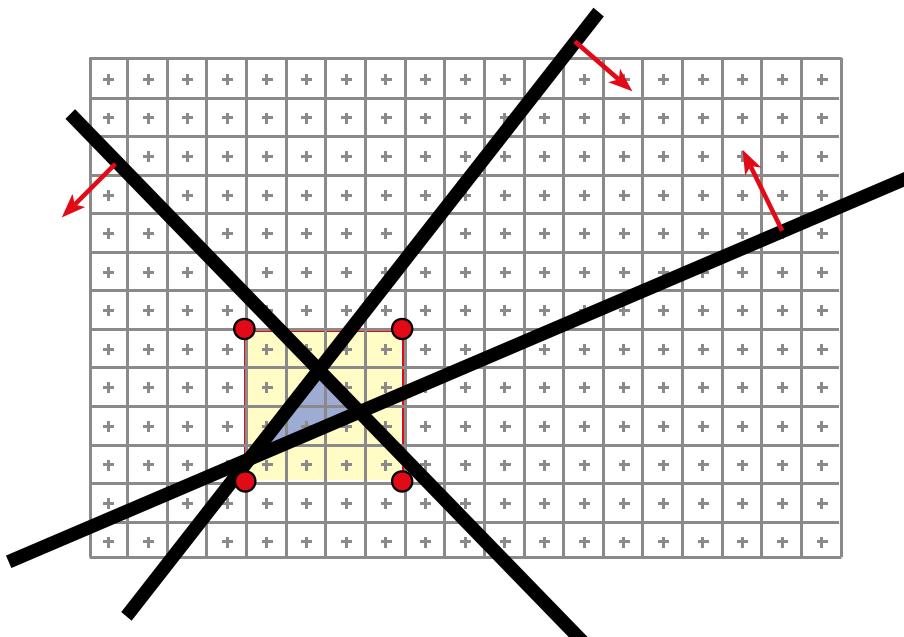
Test if i is inside j



Problem?
**If the triangle is small,
a lot of useless
computation!**

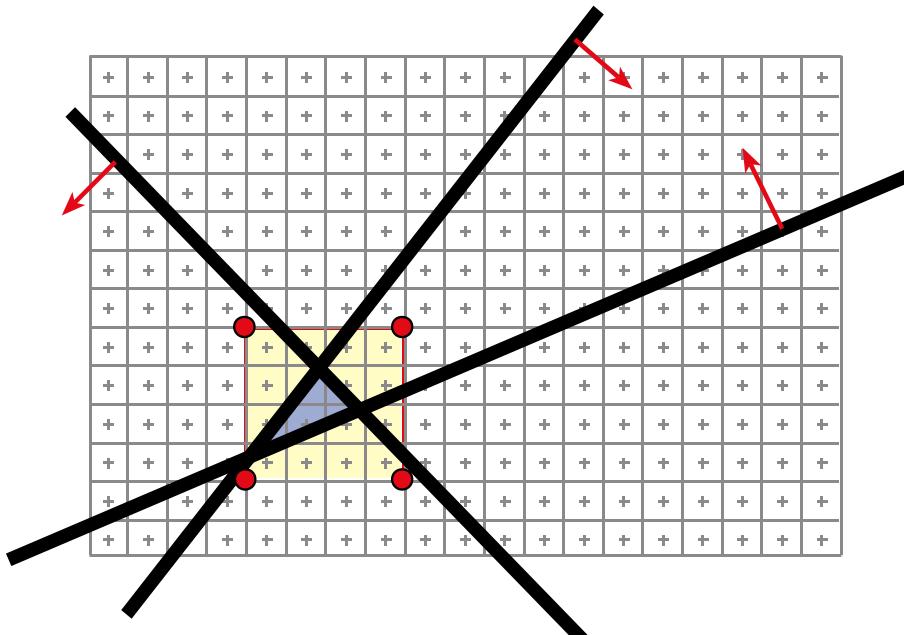
Brute Force Solution for Triangles

- Improvement: Compute only for the *screen bounding box* of the triangle
- How do we get such a bounding box?
 - $x_{\min}, x_{\max}, y_{\min}, y_{\max}$ of the triangle vertices



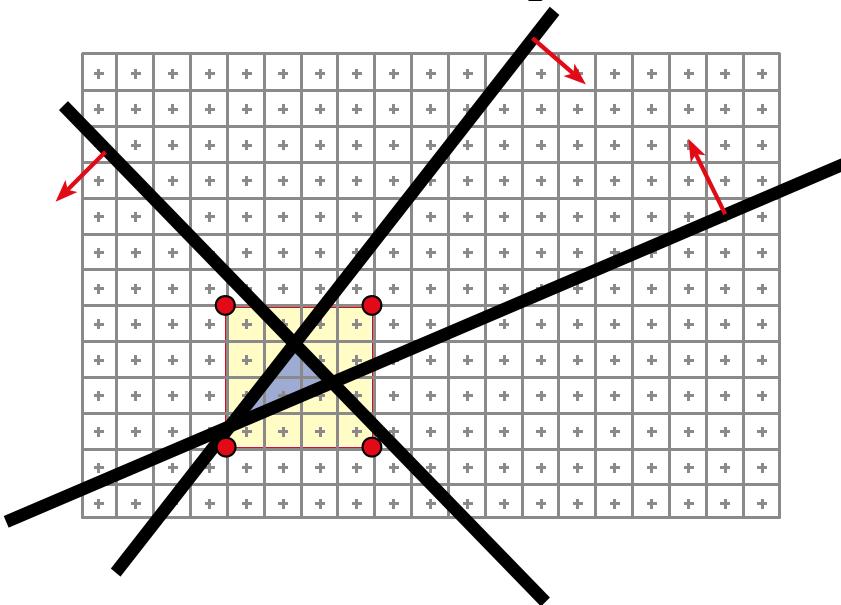
Rasterisation on Graphics Cards

- Triangles are usually very small
 - Setup cost are becoming more troublesome
- Brute force is tractable



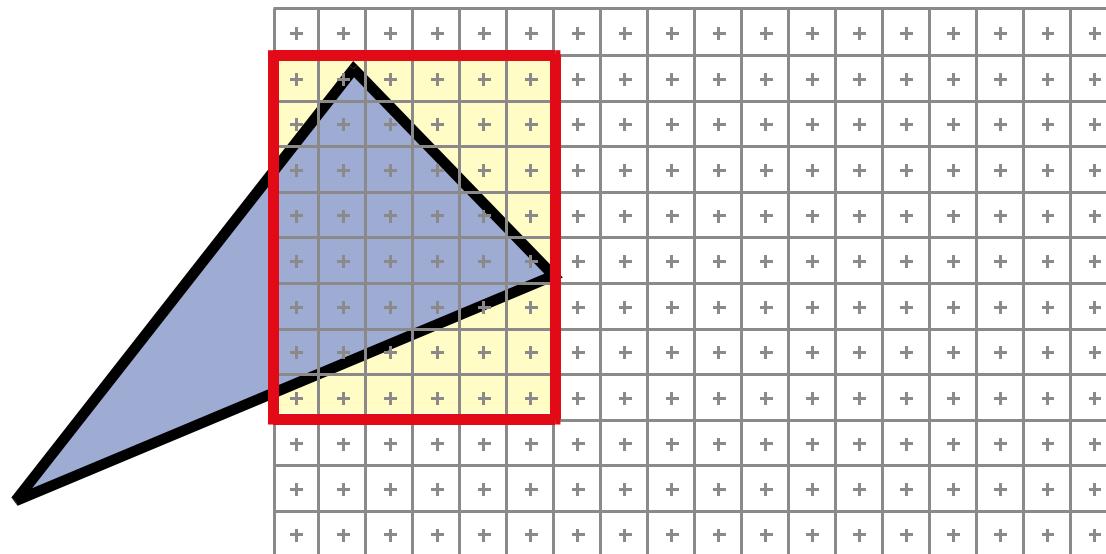
Rasterisation on Graphics Cards

```
For every triangle  
    ComputeProjection  
    Compute bbox, clip bbox to screen limits  
    Compute line equations  
    For all pixels in bbox  
        If all line equations > 0 // Pixel [x,y] in triangle  
            framebuffer[x, y] = color;
```



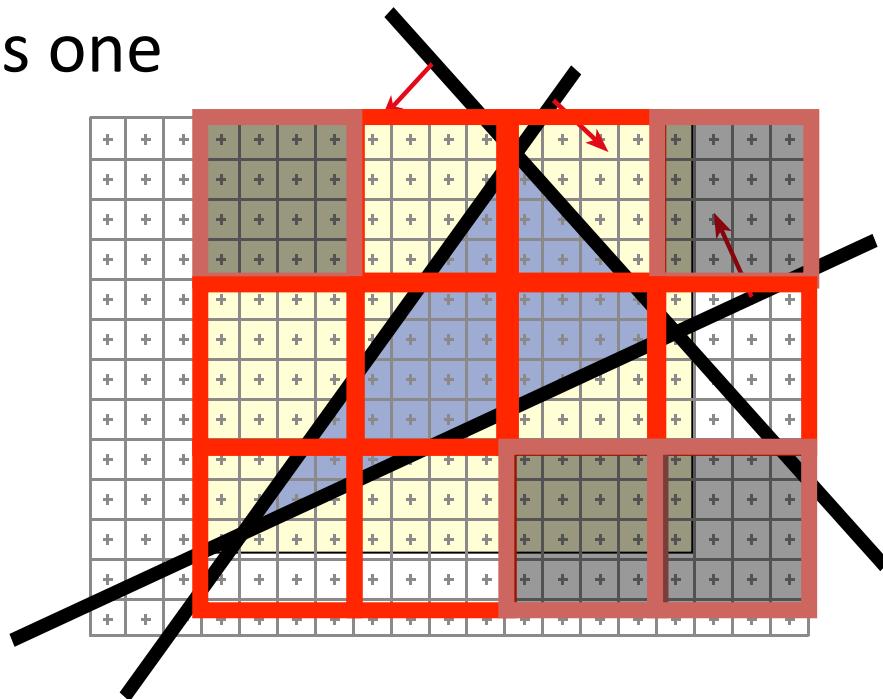
Rasterisation on Graphics Cards

Note that Bbox clipping is trivial, unlike triangle clipping



Tiling

- Subdivide BBox into smaller tiles
 - Early rejection of tiles
 - Memory access coherence
 - Each microprocessor does one



Recap

- Moving away from ray-tracing to projection
- How to
 - project a point
 - clip a polygon
 - cull a polygon
 - Fill the interior (rasterise)
- We'll spend next lectures tidying up the remaining problems!