Rasterization cont'd

15.4 Depth sorting by Z-buffering, clipping

Jaakko Lehtinen
Lots of slides from Fredo Durand

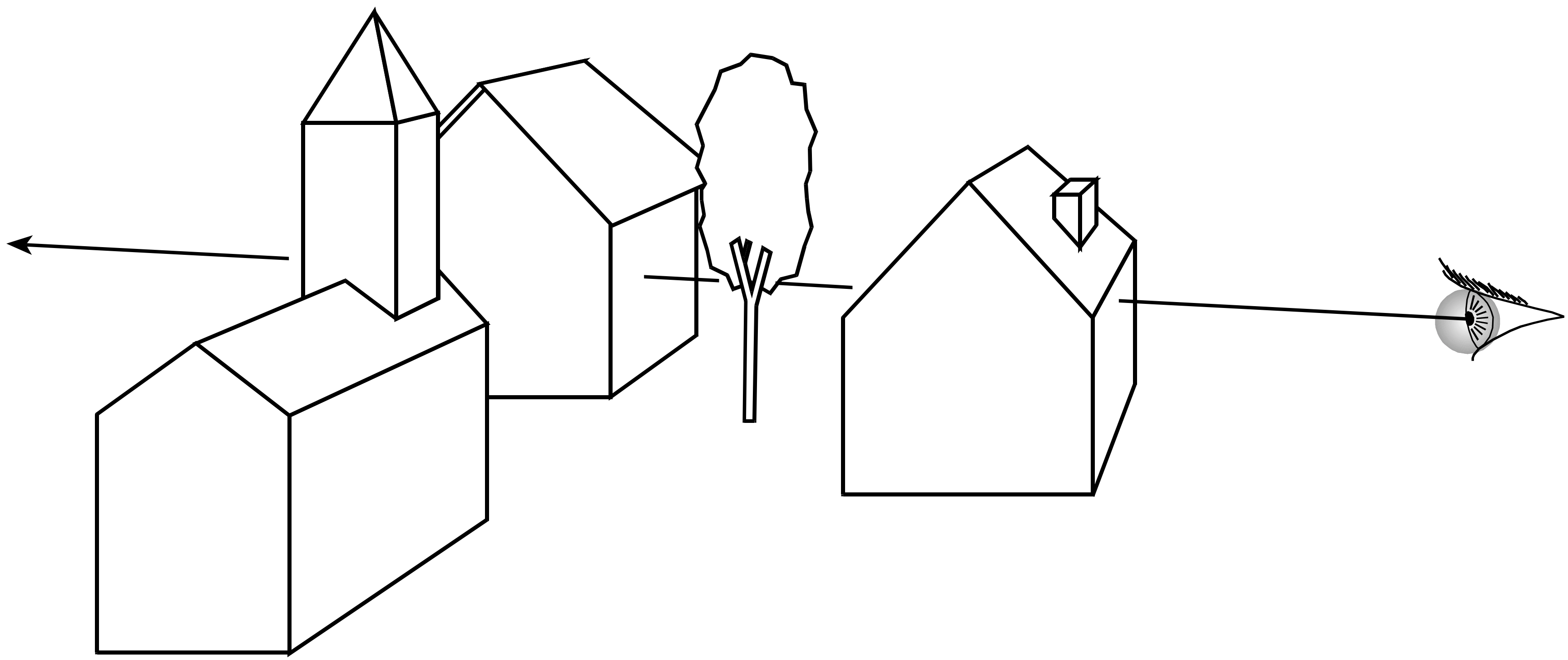
Call of Duty: Ghost / Activision

In These Slides

- To read on your own
  - Z-buffer: How to make sure we get the closest surface in each pixel when rasterizing
    - Hierarchical z-buffering
  - Interpolating attributes (like z) from vertices to pixels
  - Avoiding projection problems by clipping

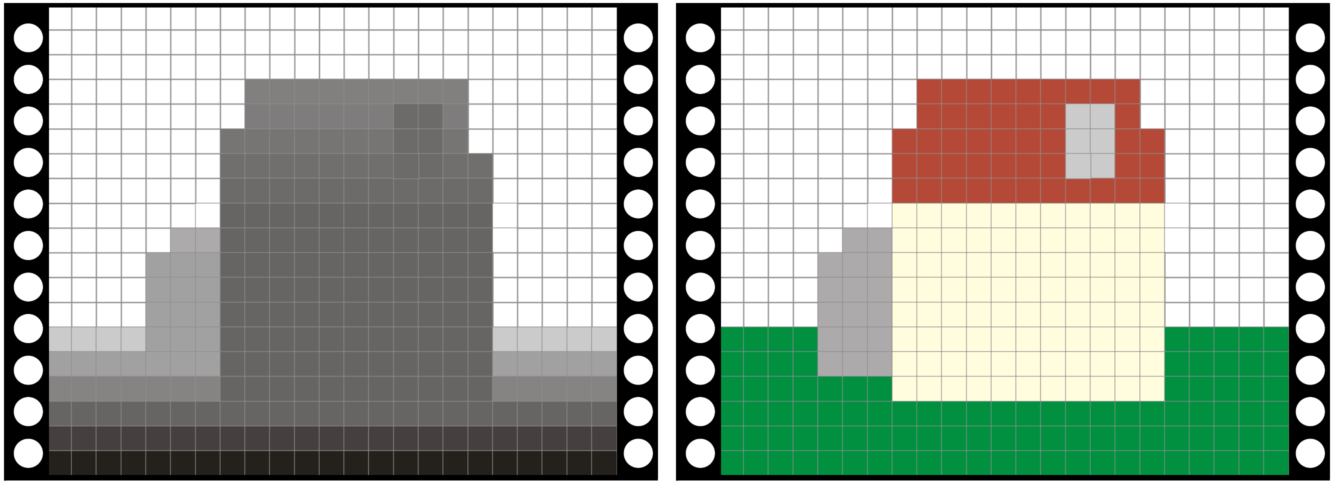
Figuring out what's visible: Z-Buffer

- In ray casting, use intersection with closest t
- Now we have swapped the loops (pixel, object)
- How do we do?



Z-buffer

- In addition to frame buffer (R, G, B), store z coordinate of rasterized points
- Pixel is updated only if new z is closer than z-buffer value

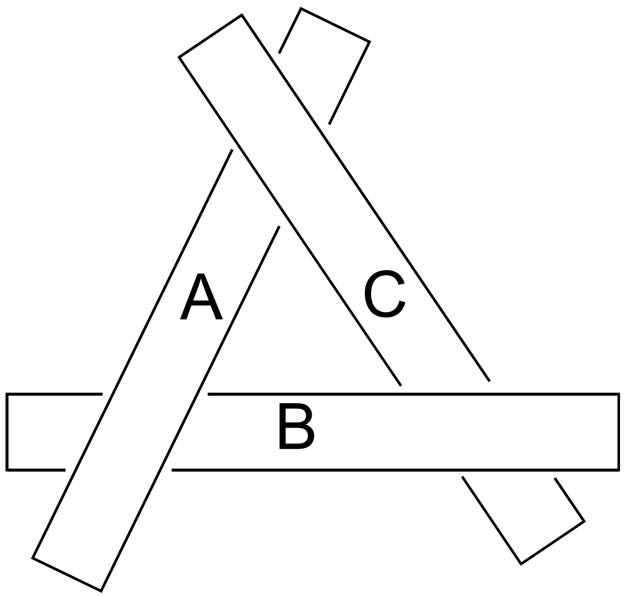


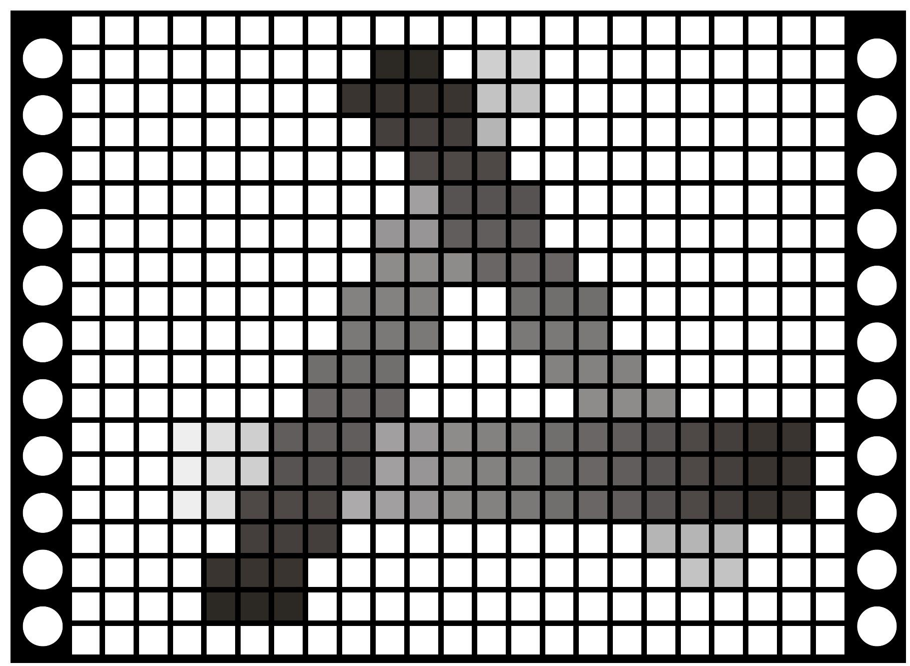
Z-buffer pseudo code

```
For every triangle
   Compute Projection, color at vertices
   Setup line equations
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Increment line equations
      Compute curentZ
      Increment currentColor
      If all line equations>0  //pixel [x,y] in triangle
         If currentZ<zBuffer[x,y]  //pixel is visible
            Framebuffer[x,y]=currentColor
            zBuffer[x,y]=currentZ
```

Z-buffer Main Benefit

- Works for hard cases!





Z-buffer Main Problem

- Works really only for opaque geometry, no general transparency
  - Why? The ordering of the surfaces is important to get transparency right
  - The Z-buffer just keeps the closest intersection
    - The ray tracer finds the closest one first, *then fires another ray*

- Funny enough, this is still an ~unsolved problem in real-time graphics even today!
  - "Order independent transparency"
  - However, great progress has been made in the last few years (Links 1 2)

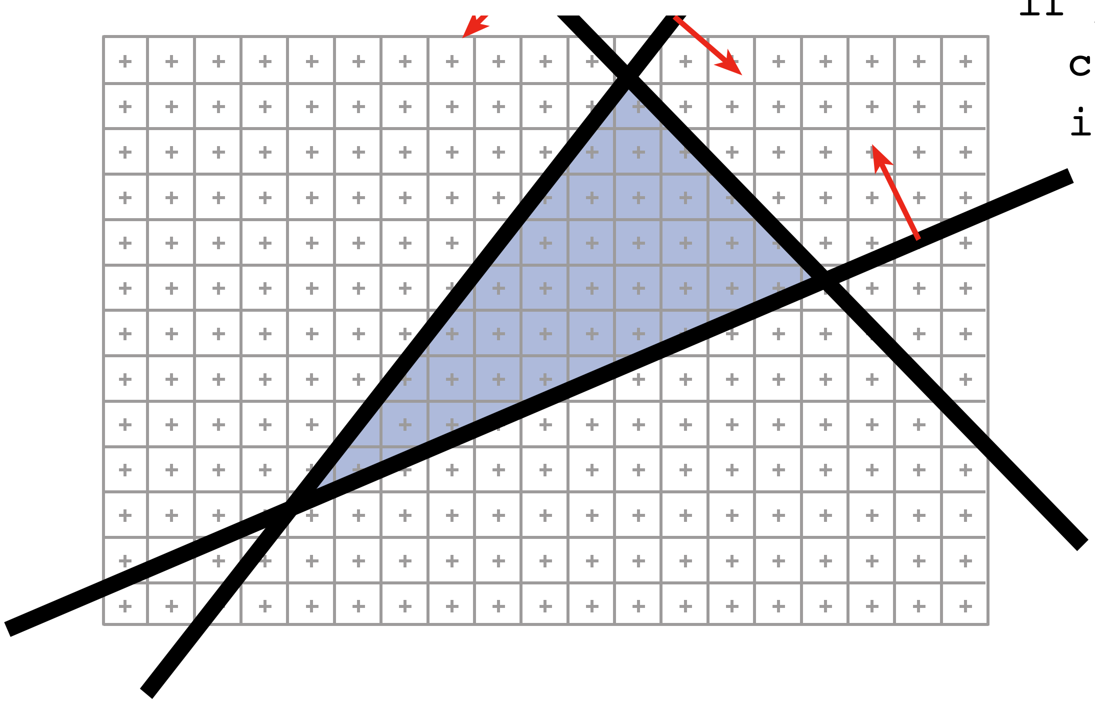
Z-buffer efficiency

- Looping over all triangles is not smart if most of them are occluded.

- What can we do?

Is That the Best We Can Do?

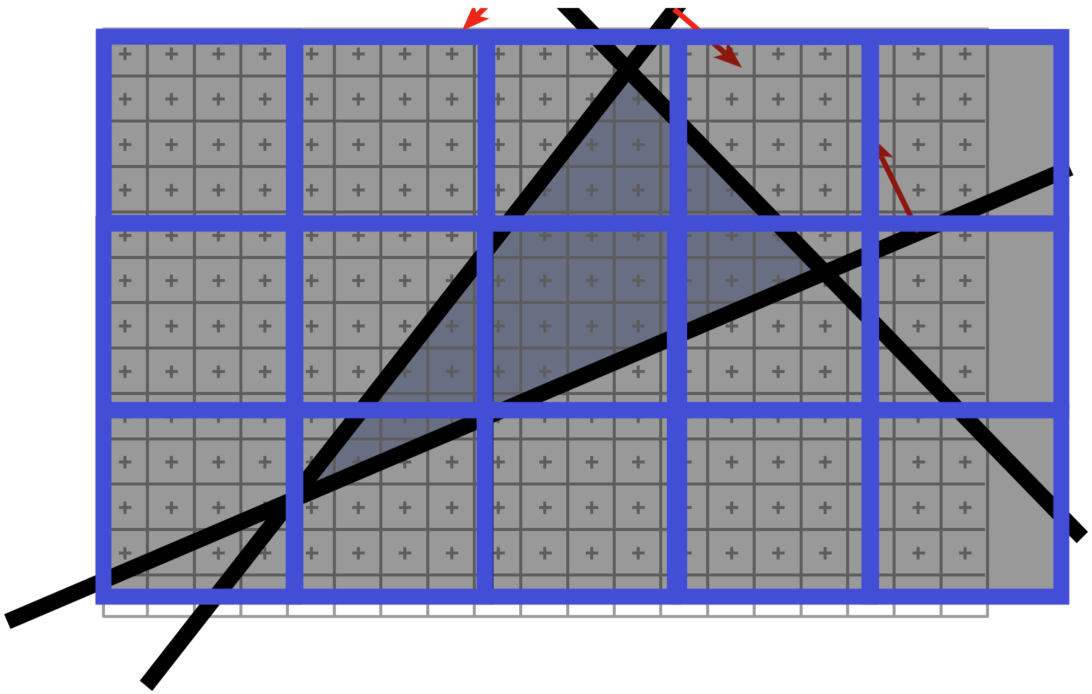
- Can we do better than just test each pixel individually after rasterization?



```
For each triangle
   for each pixel (x,y)
     if passes all edge equations
       compute z
       if z<zbuffer[x,y]
          zbuffer[x,y]=z
          framebuffer[x,y]=shade()
```

Hierarchical Z-Buffer

- Keep a Z_{min} and Z_{max} value for each tile
- Check all tiles within triangle bounding box: If all tiles' Z_{max} is closer than the minimum Z of the vertices, the triangle cannot be visible!



Otherwise, check the pixels like before. When updating the individual z values in the Z-buffer, also update the Z_{min} and Z_{max} values of the corresponding tile.

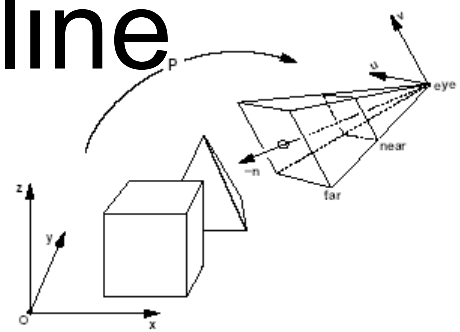
Occlusion Culling

- We can do even better!
- We can test an object's conservative (3D) bounding volume (usually box) against the hierarchical Z-buffer before drawing any of the triangles
  - If bounding box is not visible, don't submit the triangles
- Sorting objects front-to-back makes this efficient

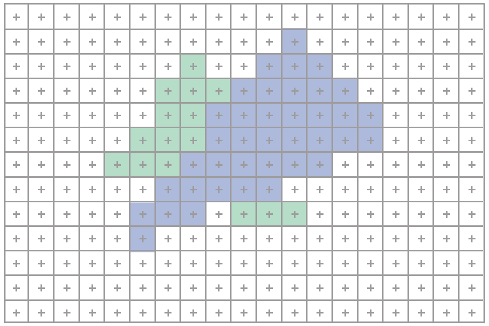
Occlusion Culling

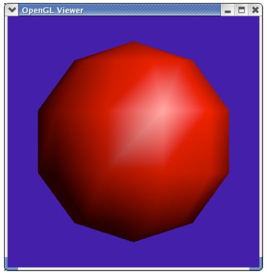
- We can do even better!
- We can test an object's conservative (3D) bounding volume (usually box) against the hierarchical Z-buffer before drawing any of the triangles
  - If bounding box is not visible, don't submit the triangles
- Sorting objects front-to-back makes this efficient
- OpenGL/DirectX "occlusion queries" and "predicated rendering" can be used to do this.
- There are neat algorithms that allow *output-sensitive* rendering of really large scenes
  - Cf. Umbra Software's engine middleware

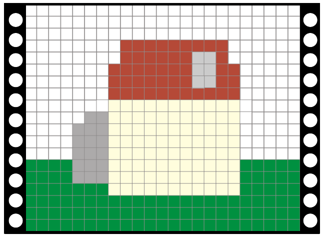
Recap: The Graphics Pipeline

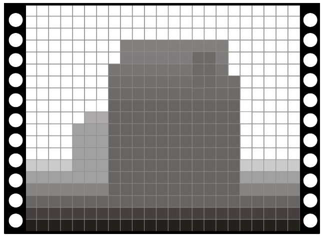



```
For each triangle
  transform into eye space
  project from 3D to 2D
  set up 3 edge equations
  for each pixel x,y
    if passes all edge equations
      compute z
      if z<zbuffer[x,y]
        zbuffer[x,y]=z
        framebuffer[x,y]=shade()
```







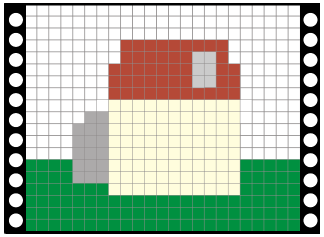


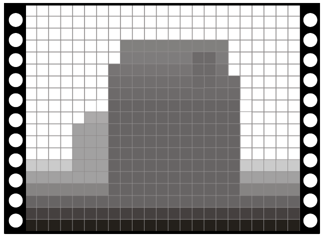
Why quotes? We are leaving out programmable stages and parallelism

Interpolation in Screen Space

- How do we get that Z value for each pixel?
  - We only know z at the vertices...
  - Must interpolate from vertices into triangle interior

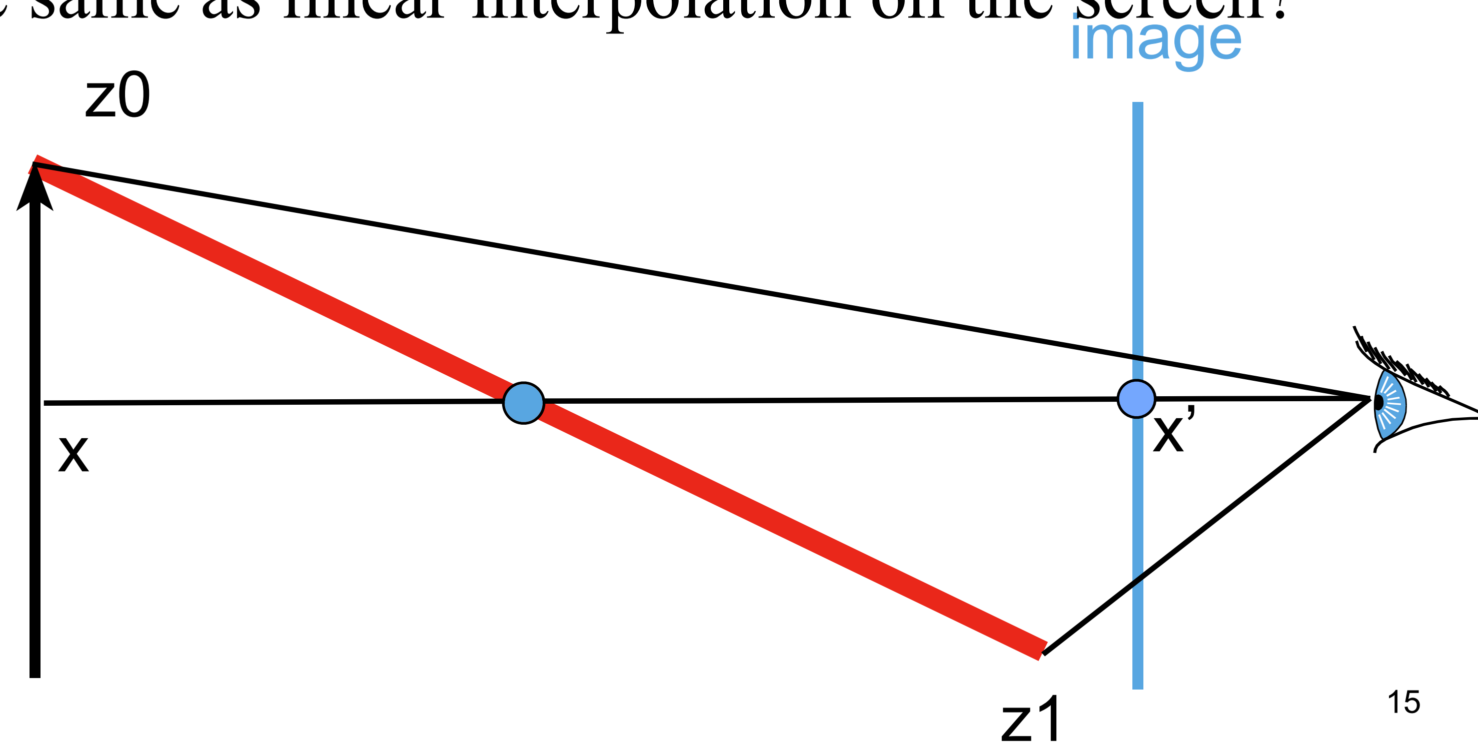
```
For each triangle
  for each pixel (x,y)
    if passes all edge equations
      compute z
      if z<zbuffer[x,y]
        zbuffer[x,y]=z
        framebuffer[x,y]=shade()
```



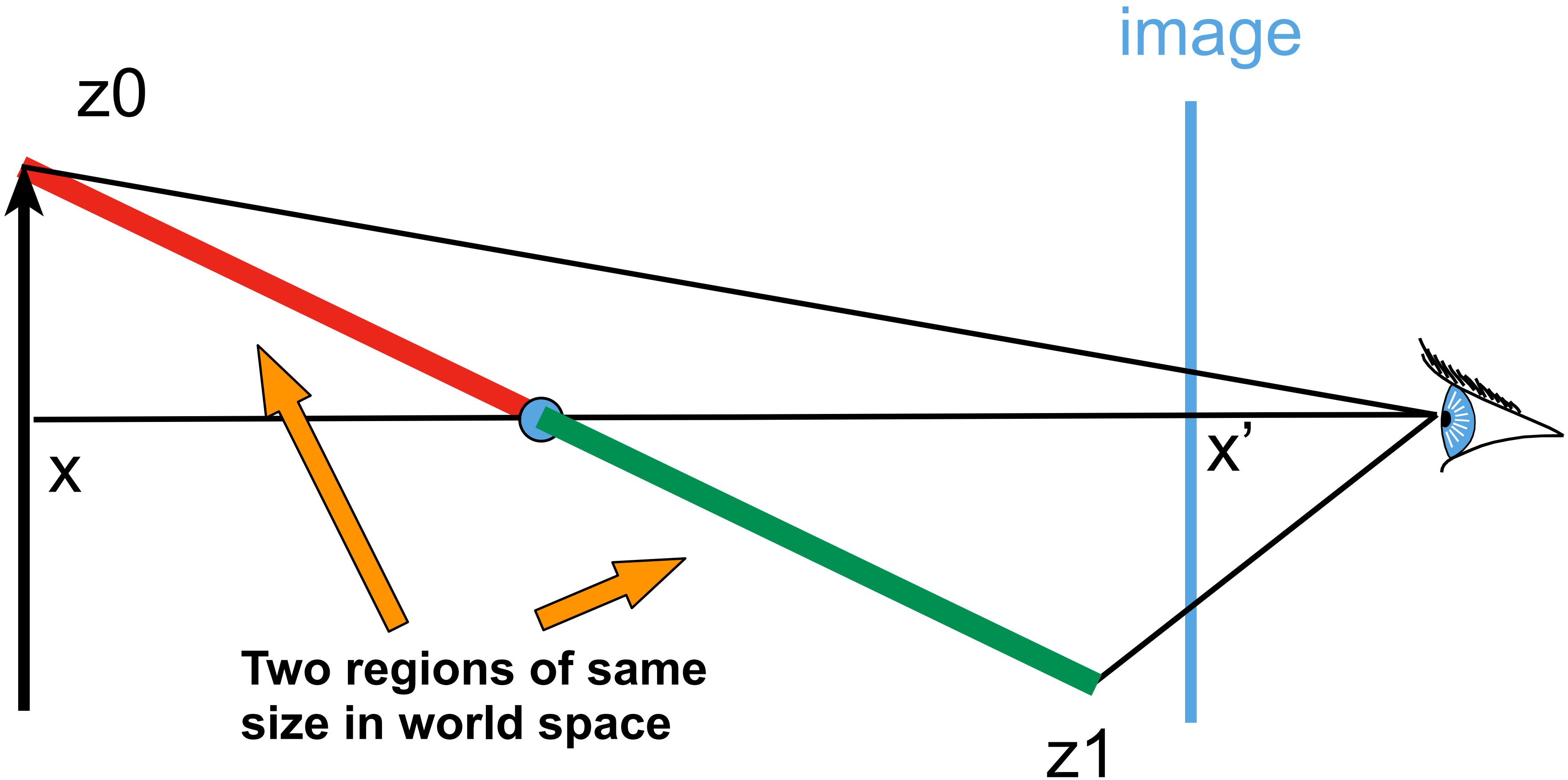


Interpolation in Screen Space

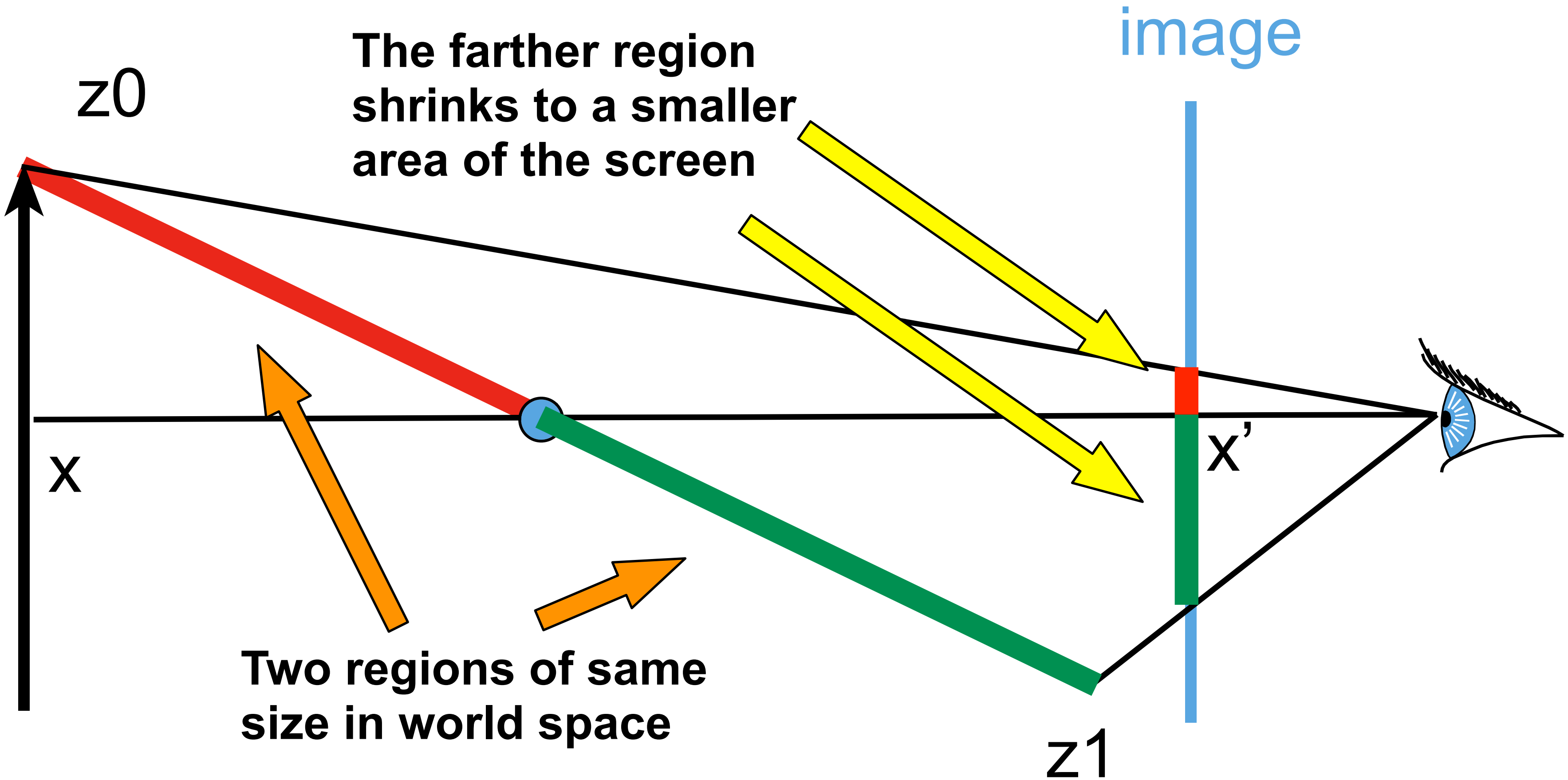
- Also need interpolate color, normals, texture coordinates, etc. between vertices
  - We did this with barycentrics in ray casting
    - Linear interpolation in object space
  - Is it the same as linear interpolation on the screen?



Interpolation in Screen Space

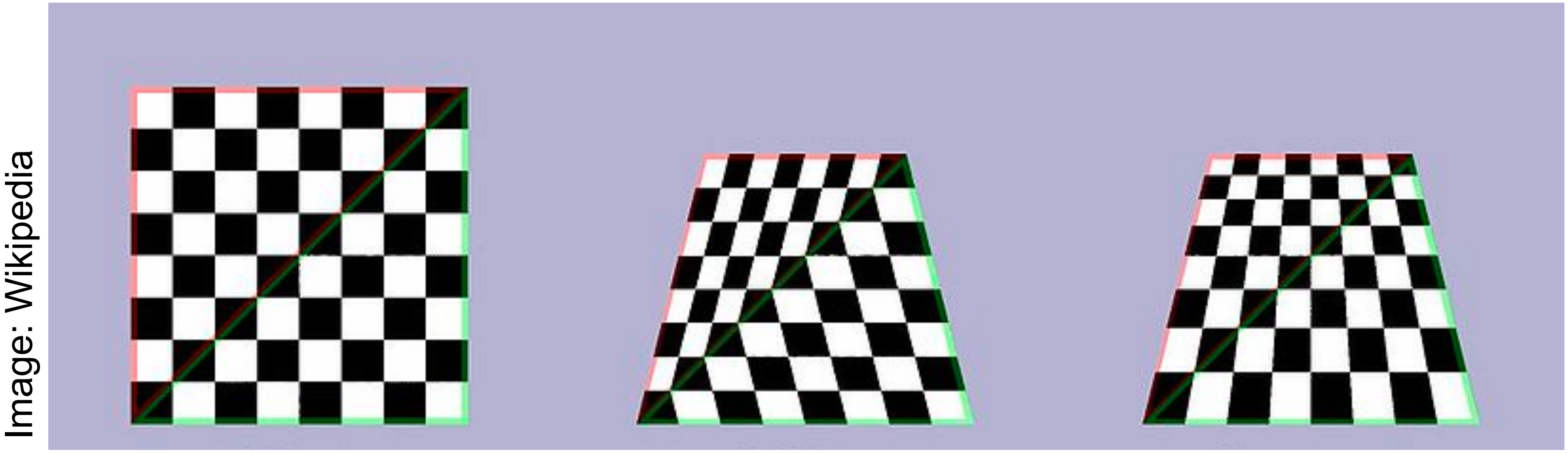


Interpolation in Screen Space



Nope, Not the Same

- Linear variation in world space does not yield linear variation in screen space due to projection
  - Think of looking at a checkerboard at a steep angle; all squares are the same size on the plane, but not on screen



Head-on view | “Gouraud interpolation” | Perspective-correct Interpolation

Solution: Barycentrics, Again

- Barycentric coordinates for a triangle (**a**, **b**, **c**)

$$P(\alpha, \beta, \gamma) = \alpha \boldsymbol{a} + \beta \boldsymbol{b} + \gamma \boldsymbol{c}$$

  – Remember, $\alpha + \beta + \gamma = 1, \quad \alpha, \beta, \gamma \geq 0$

Solution: Barycentrics, Again

- Barycentric coordinates for a triangle (**a**, **b**, **c**)

$$P(\alpha, \beta, \gamma) = \alpha \boldsymbol{a} + \beta \boldsymbol{b} + \gamma \boldsymbol{c}$$

  – Remember, $\alpha + \beta + \gamma = 1, \quad \alpha, \beta, \gamma \geq 0$

- Let's project point P by projection matrix **C**

$$\begin{aligned} \boldsymbol{C}P &= \boldsymbol{C}(\alpha \boldsymbol{a} + \beta \boldsymbol{b} + \gamma \boldsymbol{c}) \\ &= \alpha \boldsymbol{C} \boldsymbol{a} + \beta \boldsymbol{C} \boldsymbol{b} + \gamma \boldsymbol{C} \boldsymbol{c} \\ &= \alpha \boldsymbol{a}' + \beta \boldsymbol{b}' + \gamma \boldsymbol{c}' \end{aligned}$$

a', **b**', **c**' are the projected homogeneous vertices **before** division by w

Solution: Barycentrics, Again

$$CP = \alpha \boldsymbol{a}' + \beta \boldsymbol{b}' + \gamma \boldsymbol{c}'$$

a', **b**', **c**' are the projected homogeneous vertices

- The (x, y) screen coordinates of P are

$$(P_x/P_w,\ P_y/P_w) =$$

$$\left(\frac{\alpha a'_x + \beta b'_x + \gamma c'_x}{\alpha a'_w + \beta b'_w + \gamma c'_w}, \frac{\alpha a'_y + \beta b'_y + \gamma c'_y}{\alpha a'_w + \beta b'_w + \gamma c'_w} \right)$$

Solution: Barycentrics, Again

$$CP = \alpha \boldsymbol{a}' + \beta \boldsymbol{b}' + \gamma \boldsymbol{c}'$$

a', **b**', **c**' are the projected homogeneous vertices

- The (x, y) screen coordinates of P are

$$(P_x/P_w,\ P_y/P_w) =$$

$$\left(\frac{\alpha a'_x + \beta b'_x + \gamma c'_x}{\alpha a'_w + \beta b'_w + \gamma c'_w},\ \frac{\alpha a'_y + \beta b'_y + \gamma c'_y}{\alpha a'_w + \beta b'_w + \gamma c'_w} \right)$$

This looks familiar...

Solution: Barycentrics, Again

projective equivalence

$$\begin{pmatrix} P_x/P_w \\ P_y/P_w \\ 1 \end{pmatrix} \sim \begin{pmatrix} P_x \\ P_y \\ P_w \end{pmatrix} = \begin{pmatrix} a'_x & b'_x & c'_x \\ a'_y & b'_y & c'_y \\ a'_w & b'_w & c'_w \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix}$$

- It's a projective mapping from the barycentrics onto screen coordinates!
  - Represented by a 3x3 matrix
- The inverse of a projection is a projection...
  - We'll just take the inverse mapping to get from (x, y, 1) to the barycentrics!

From Screen to Barycentrics

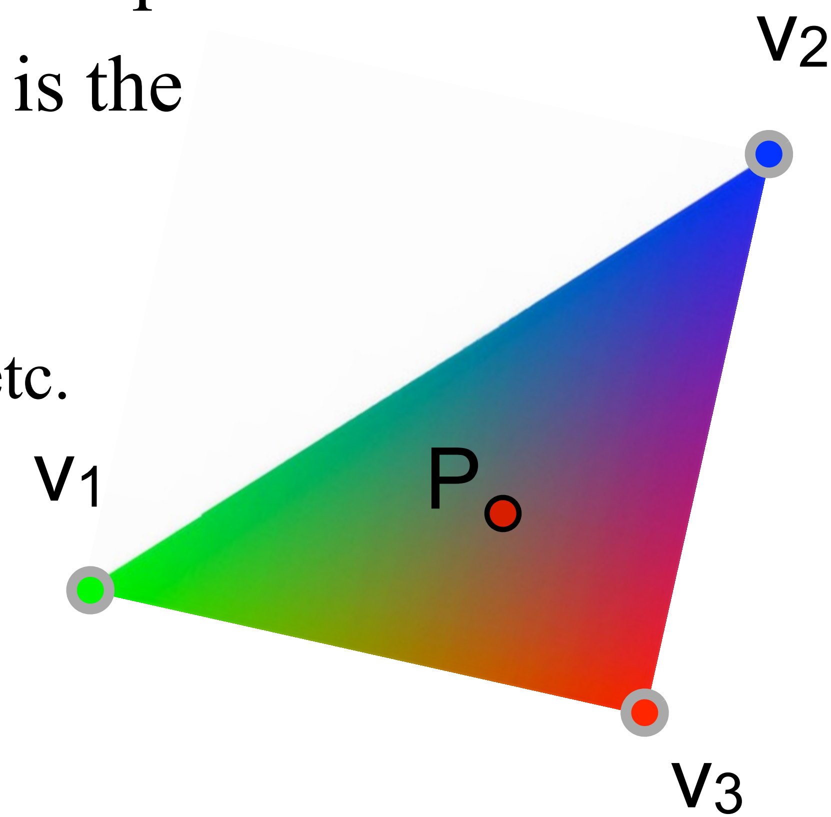
projective equivalence

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} \sim \begin{pmatrix} a'_x & b'_x & c'_x \\ a'_y & b'_y & c'_y \\ a'_w & b'_w & c'_w \end{pmatrix}^{-1} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

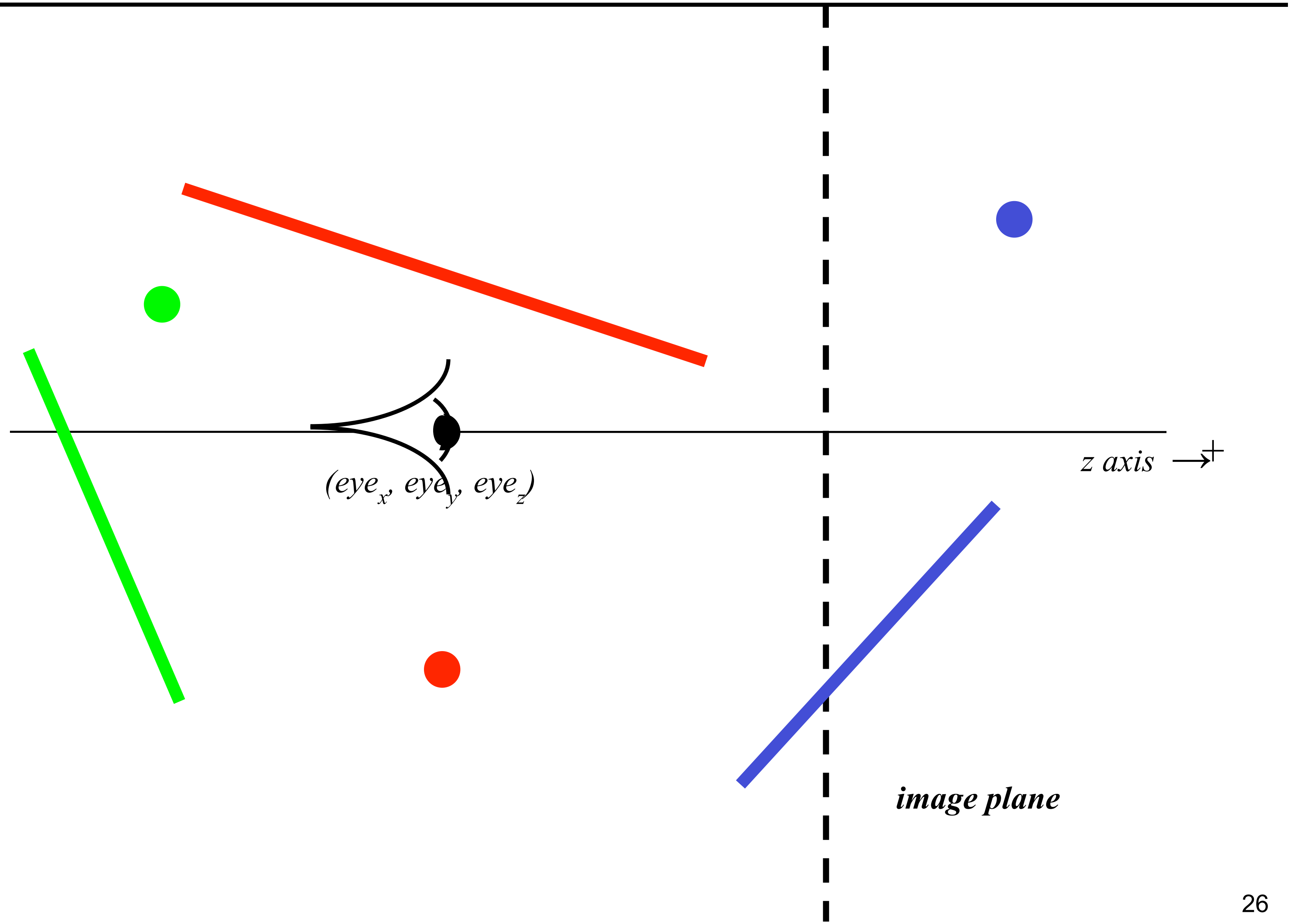
- Recipe
  - Compute projected homogeneous coordinates **a**', **b**', **c**'
  - Put them in the columns of a matrix, invert it
  - Multiply screen coordinates (x, y, 1) by inverse matrix
  - **Then divide by the sum of the resulting coordinates**
    - This ensures the result is sums to one like barycentrics should
  - Then interpolate value (e.g. Z) from vertices using them

Barycentric Interpolation Recap

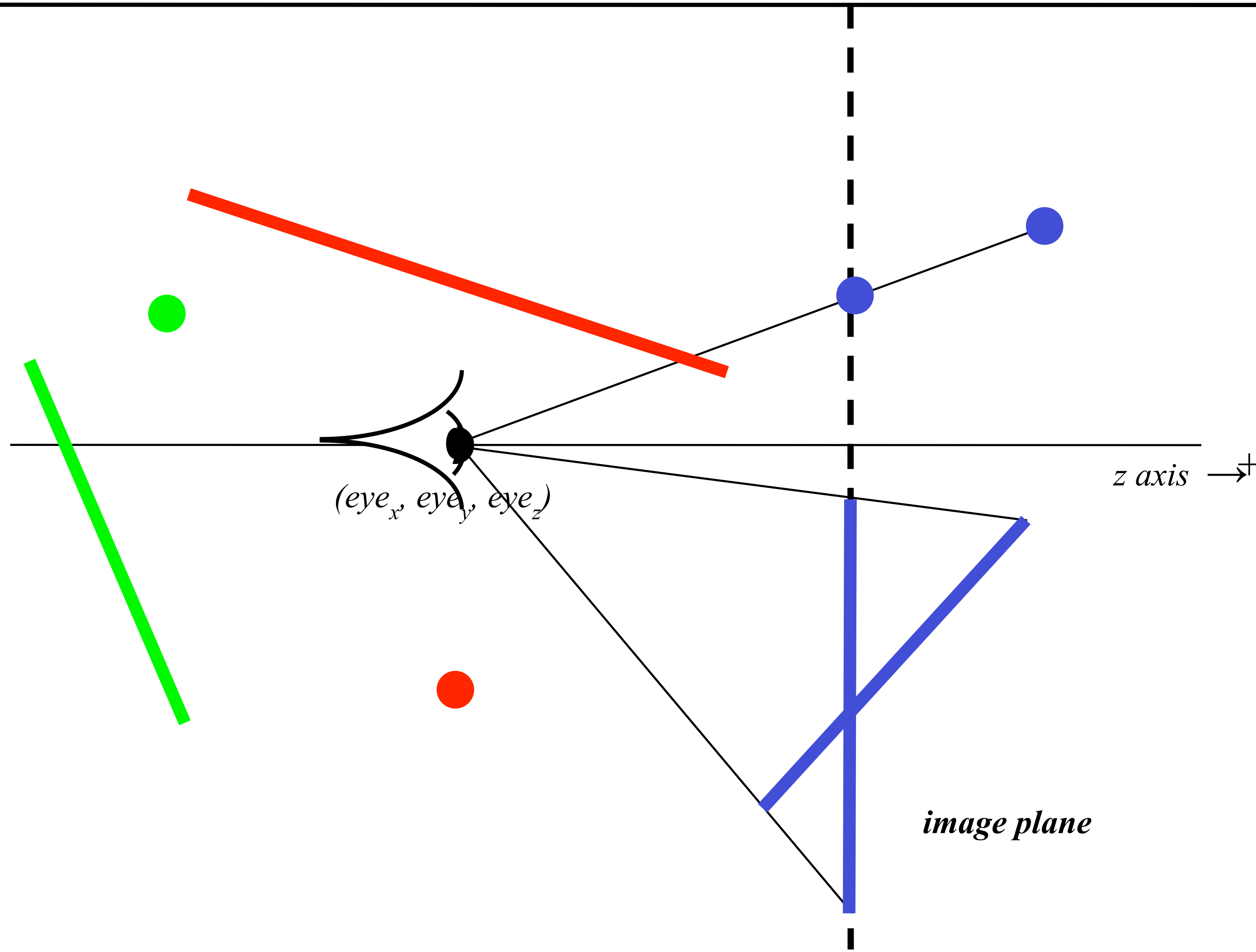
- Values v_1, v_2, v_3 defined at **a**, **b**, **c**
  - Colors, normal, texture coordinates, etc.
- $\mathbf{P}(\alpha, \beta, \gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$ is the point...
- $v(\alpha, \beta, \gamma) = \alpha v_1 + \beta v_2 + \gamma v_3$ is the barycentric interpolation of v_1-v_3 at point **P**
  - Sanity check: $v(1,0,0) = v_1$, etc.
- I.e, once you know α, β, γ, you can interpolate values using the same weights.
  - Convenient!



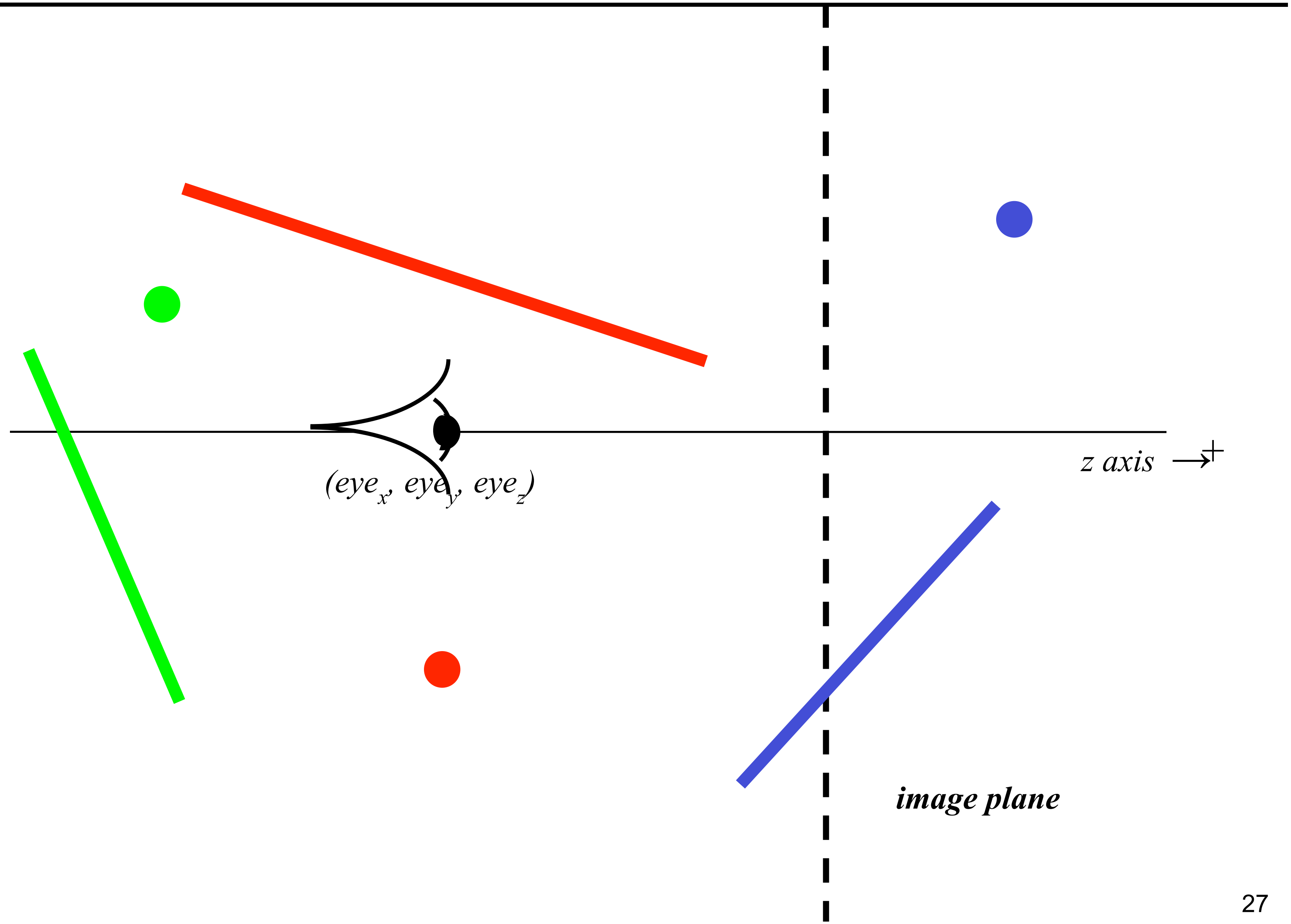
Clipping: What if the p_z is > eye_z?



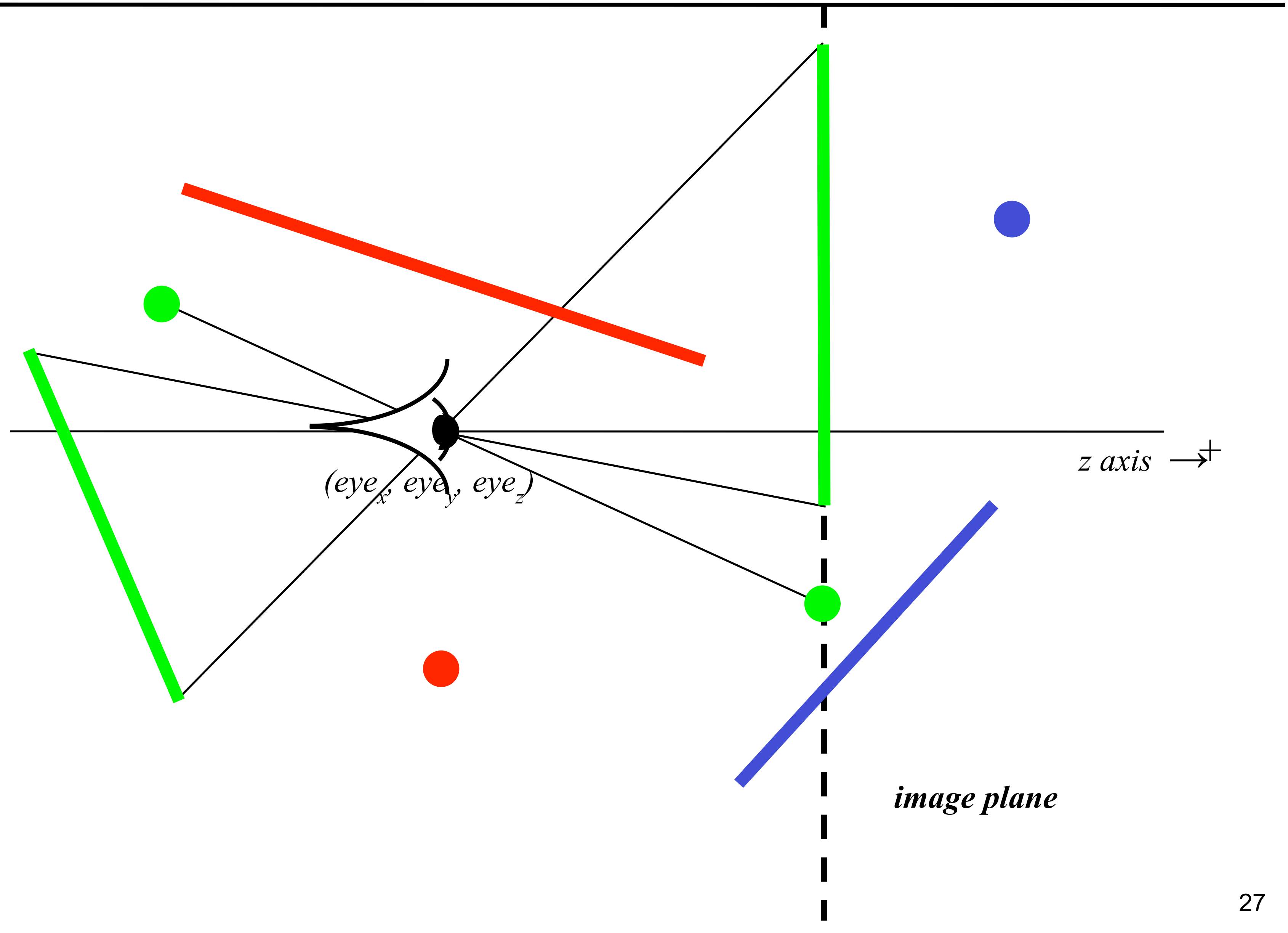
Clipping: What if the p_z is > eye_z?



What if the p_z is < eye_z?

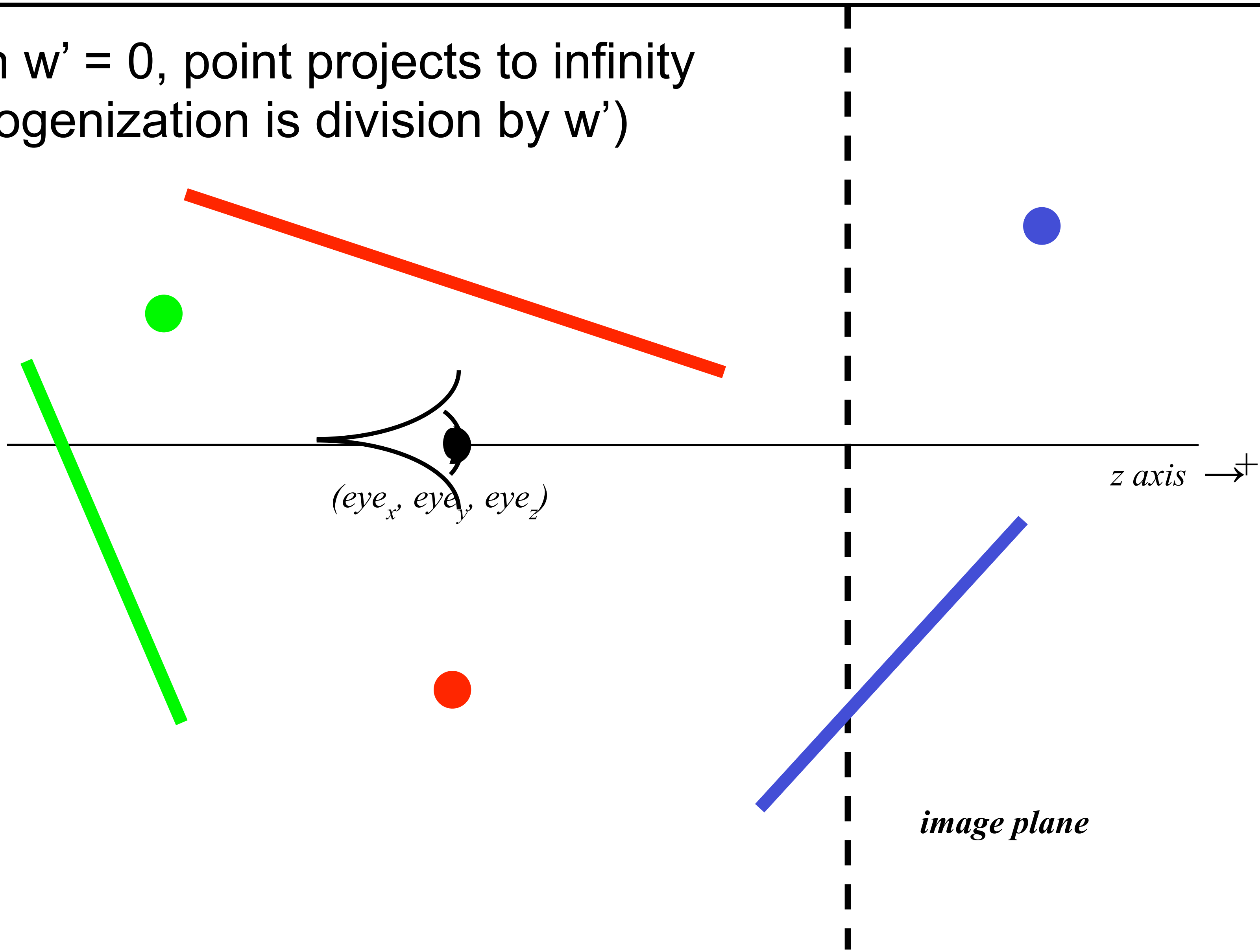


What if the p_z is < eye_z?



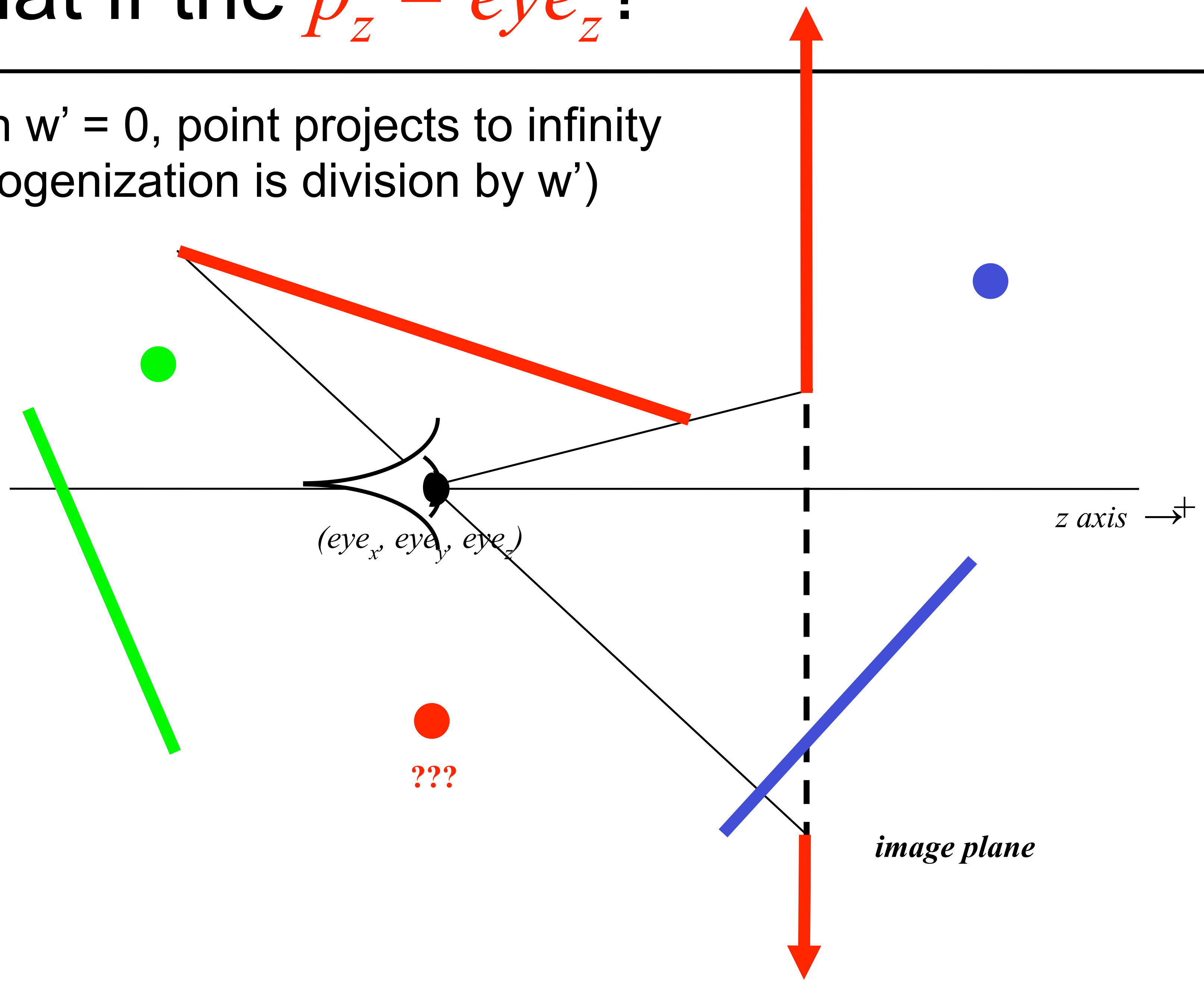
What if the $p_z = eye_z$?

When w' = 0, point projects to infinity
(homogenization is division by w')

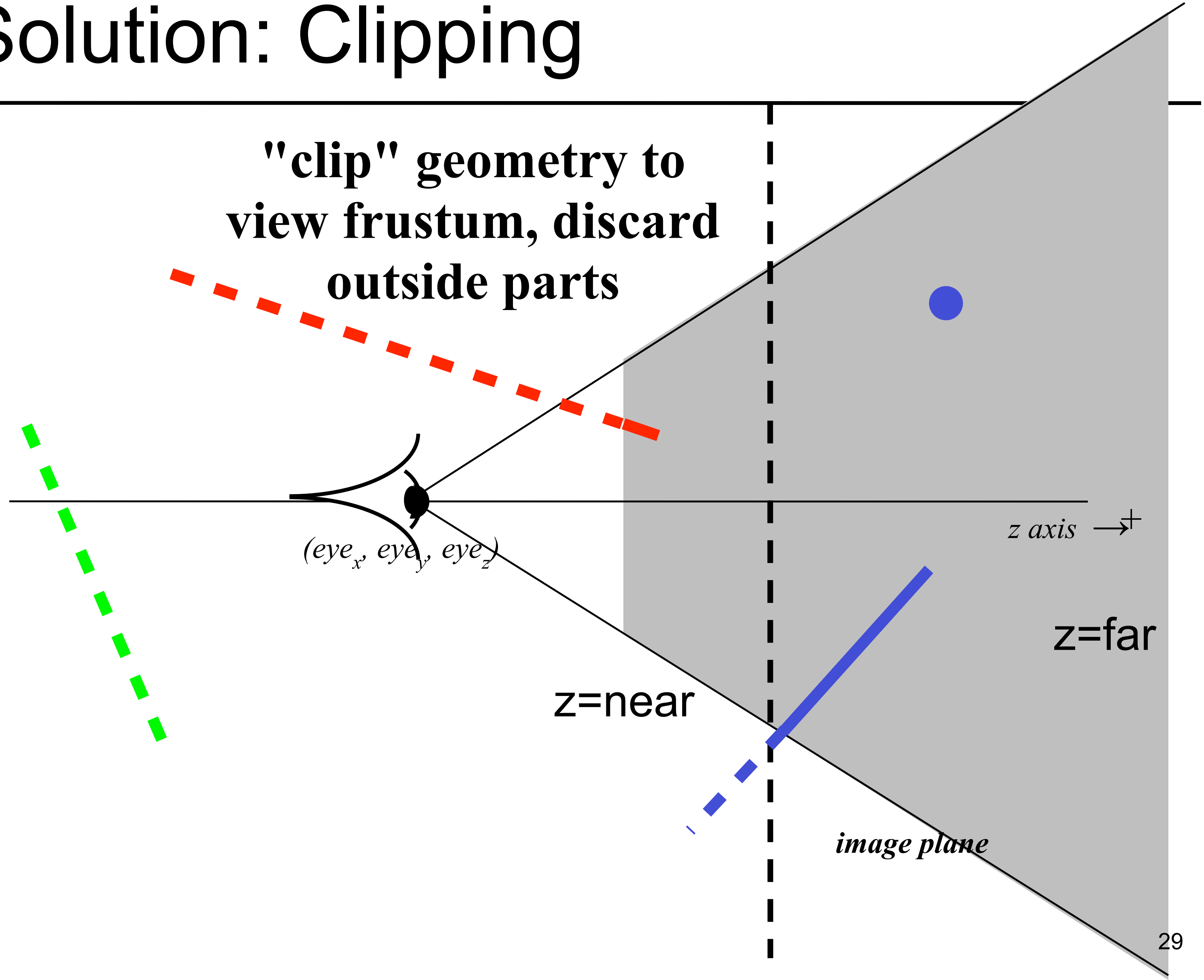


What if the $p_z = eye_z$?

When w' = 0, point projects to infinity
(homogenization is division by w')

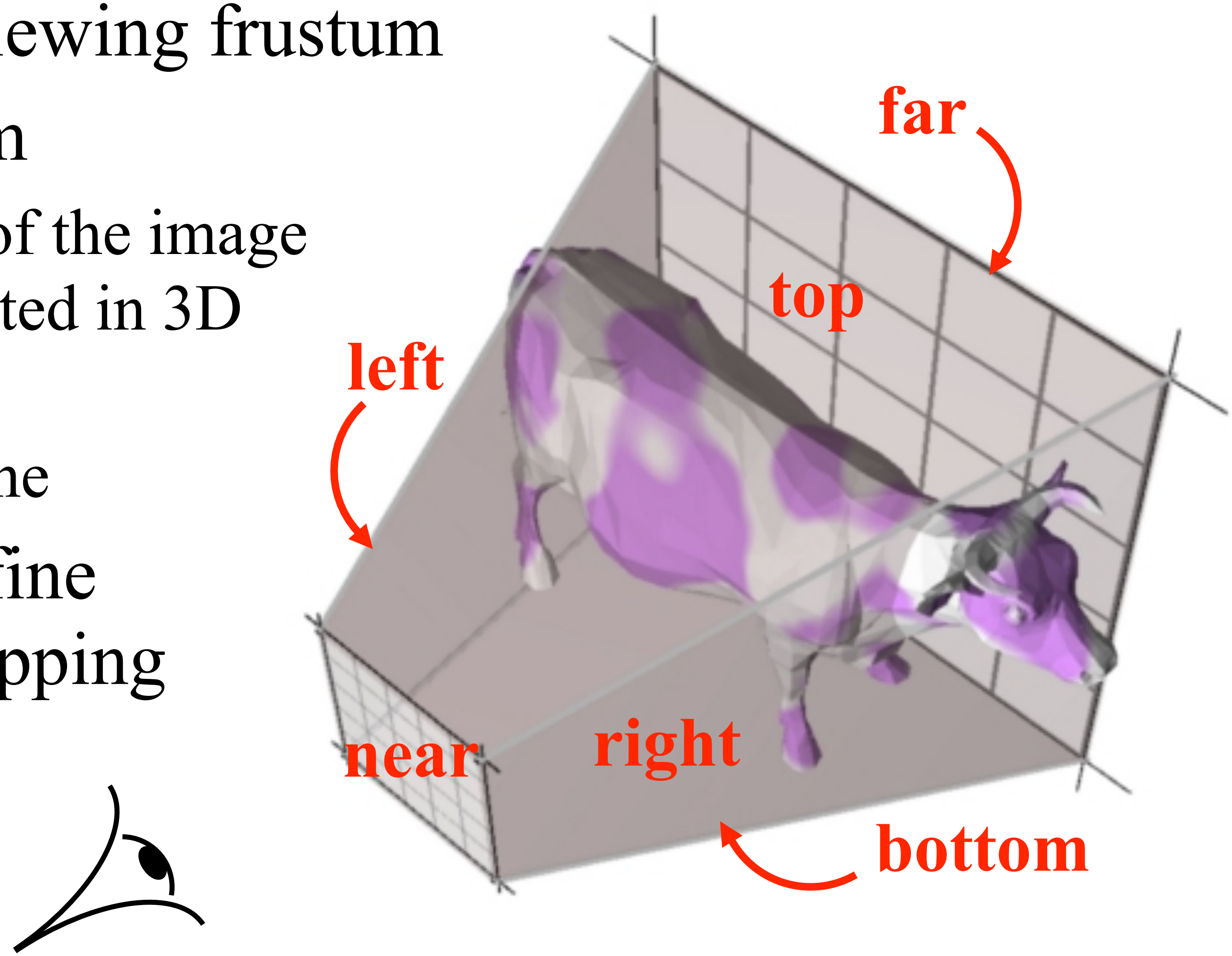


A Solution: Clipping



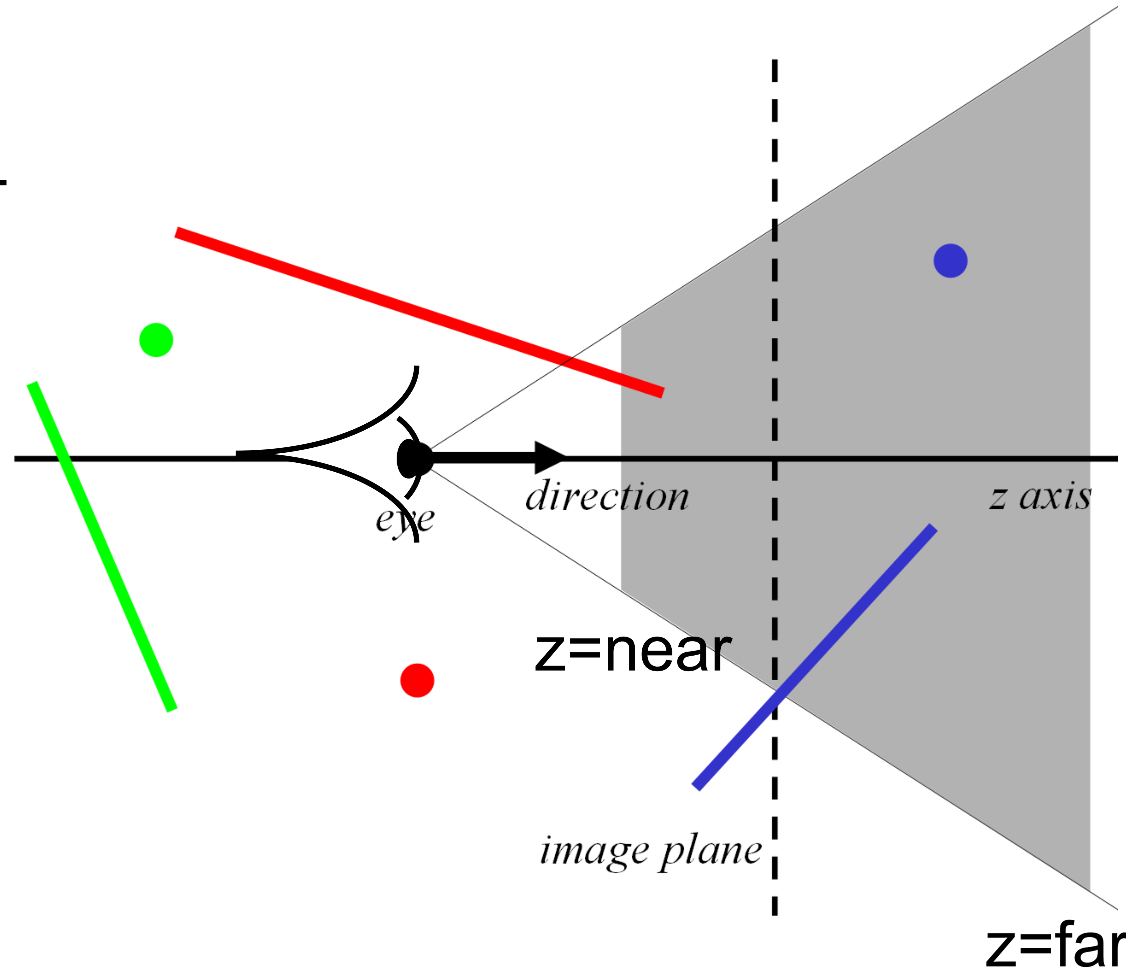
Clipping

- 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?

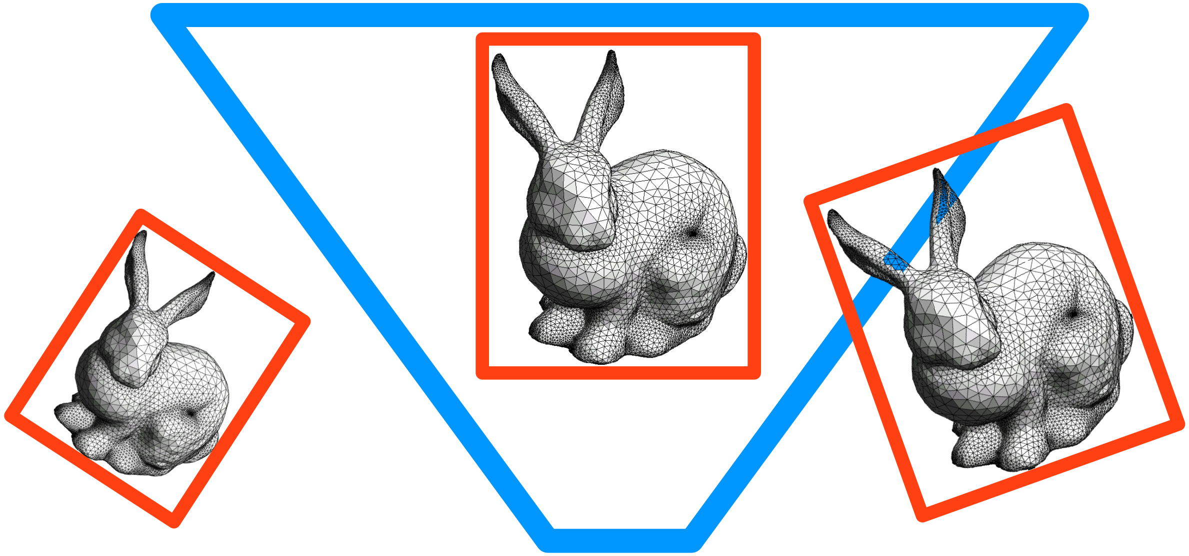
- Avoid degeneracies
  - Don't draw stuff behind the eye
  - Avoid division by 0 and overflow



Related Idea

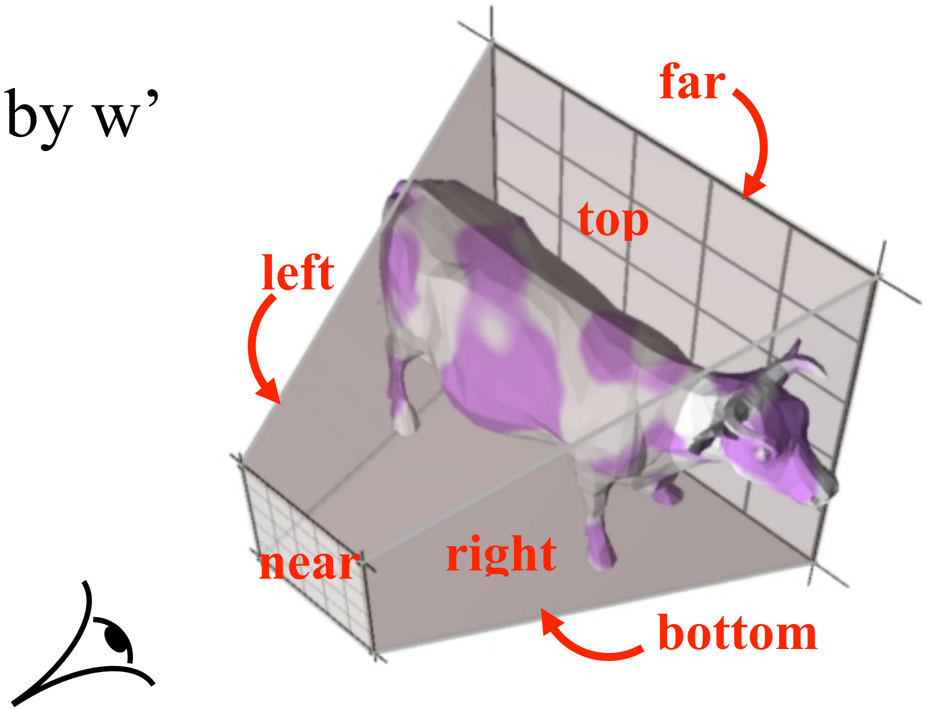
- "View Frustum Culling"
  - Use bounding volumes/hierarchies to test whether any part of an object is within the view frustum
    - Need "frustum vs. bounding volume" intersection test
    - Crucial to do hierarchically when scene has *lots* of objects!
    - Early rejection (different from clipping)

See e.g. Optimized view frustum culling algorithms for bounding boxes, Ulf Assarsson and Tomas Möller, journal of graphics tools, 2000.



Clipping

- Each side of the viewing frustum is a plane
- We'll clip the input triangles with these planes
- How do we get the plane equations?
  - We'll clip in homogeneous coordinates before division by w'



Clipping Planes

- In normalized screen coordinates, the left boundary of the screen is defined by the line x >= -1
  - Screen x = x'/w', so this can be written in homogeneous coordinates as

    x'/w' >= -1 <=> x' >= -w' <=> x' + w' >= 0

  - Using plane equation notation:

$$(1\ 0\ 0\ 1)\ (x'\ y'\ z'\ w')^T \geq 0$$

  - (x'y'z'w' are homogeneous coordinates after projection)

Clipping Planes

- In normalized screen coordinates, the left boundary of the screen is defined by the line x >= -1
  - Screen x = x'/w', so this can be written in homogeneous coordinates as

    x'/w' >= -1 <=> x' >= -w' <=> x' + w' >= 0

  - Using plane equation notation:

$$\boxed{(1\ 0\ 0\ 1)}\,(x'\ y'\ z'\ w')^T \geq 0$$

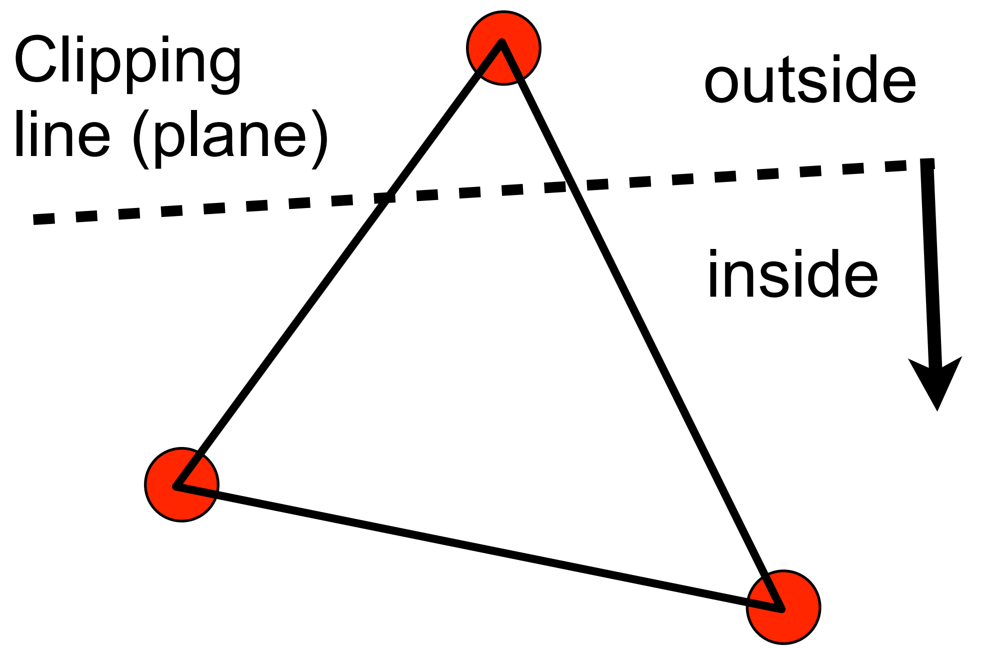
Similarly for all 5 other planes!

Homogeneous clipping plane
for left screen boundary

  - (x'y'z'w' are homogeneous coordinates after projection)

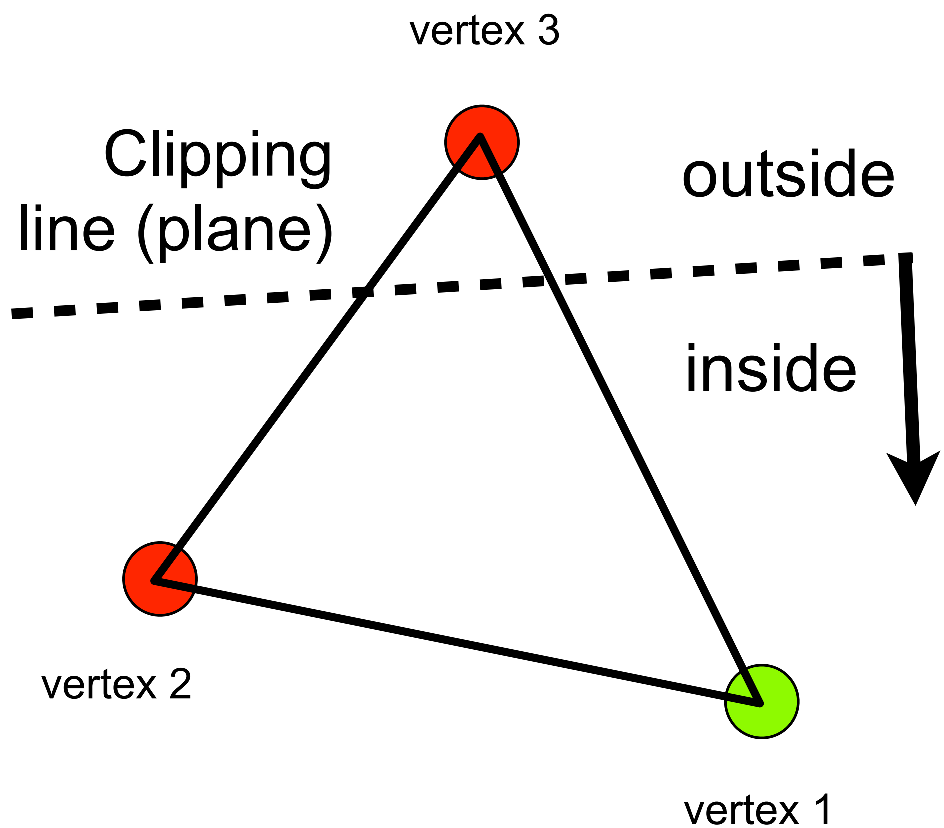
Sutherland-Hodgman Clipping

- 2D version; 3D is pretty much the same



Sutherland-Hodgman Clipping

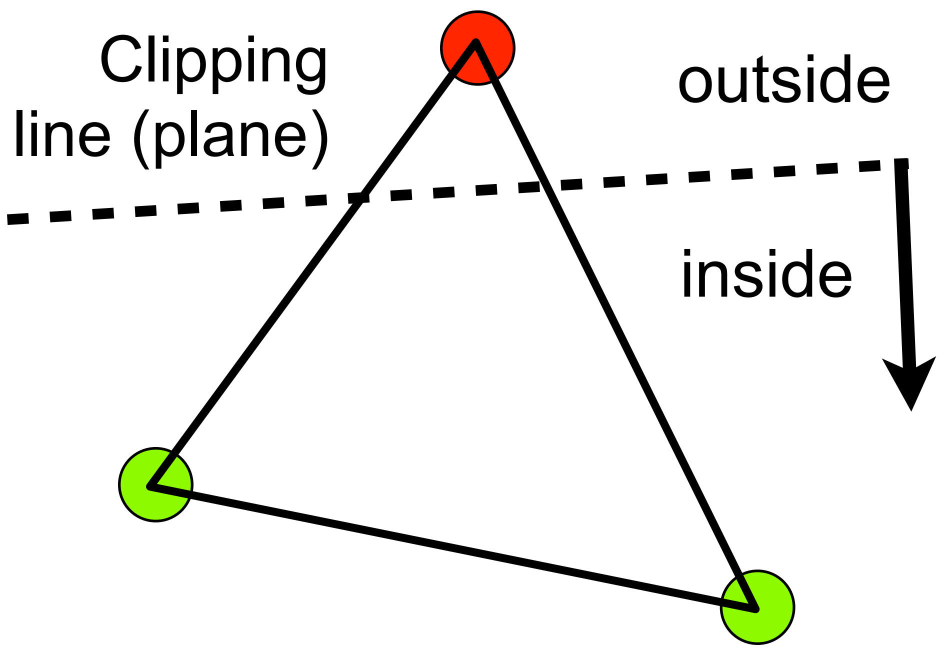
- Init: Output = empty
- Test vertex 1
  - It’s inside, add to output



Sutherland-Hodgman Clipping

- Init: Output = empty
- Test vertex 1
  - It's inside, add to output
- Test vertex 2
  - Inside, add to output

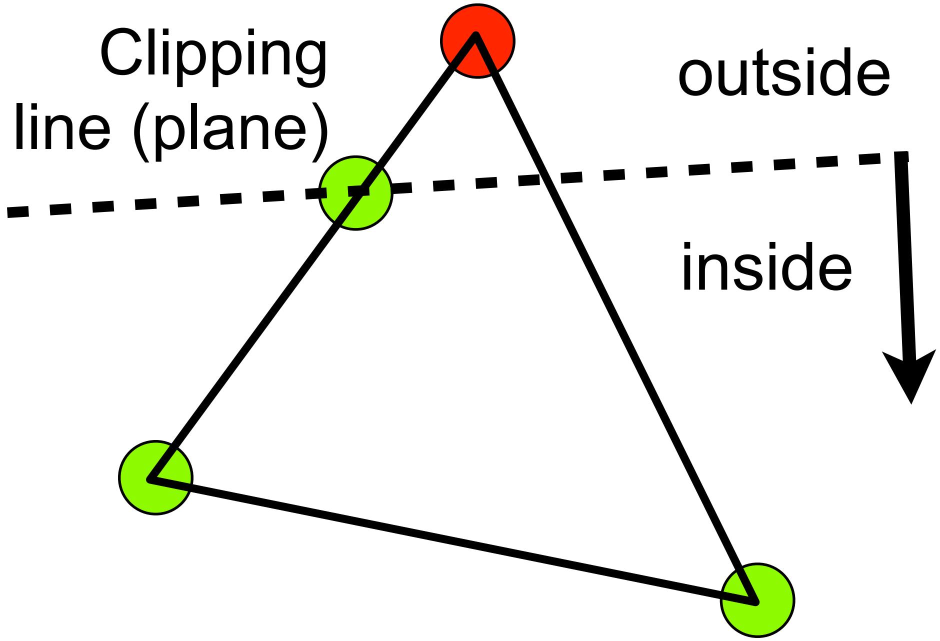
Current: In



Sutherland-Hodgman Clipping

- Init: Output = empty
- Test vertex 1
  - It's inside, add to output
- Test vertex 2
  - Inside, add to output
- Test vertex 3 (outside)
  - Compute intersection, add it to output

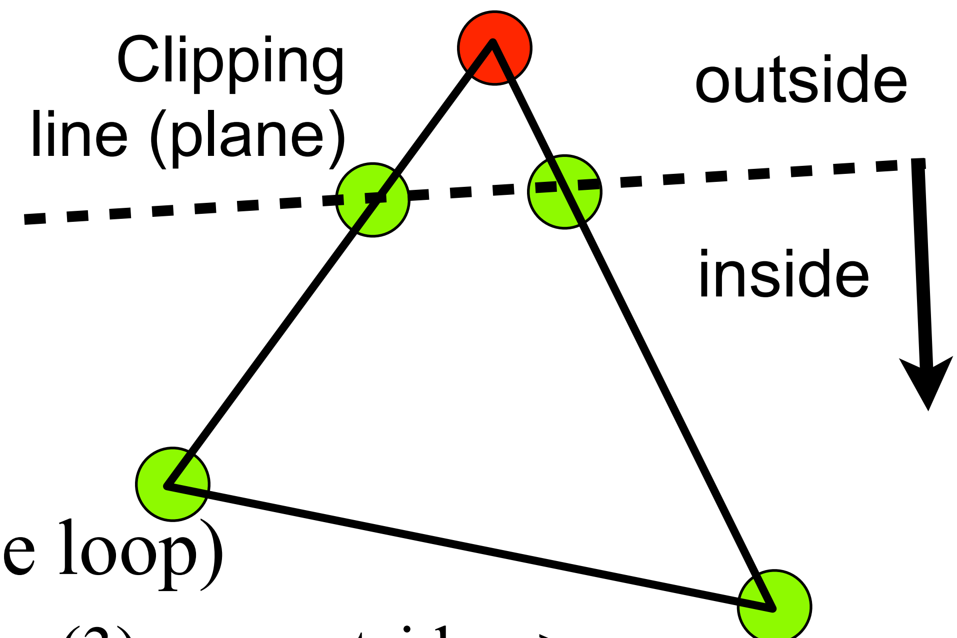
Current: Out



Sutherland-Hodgman Clipping

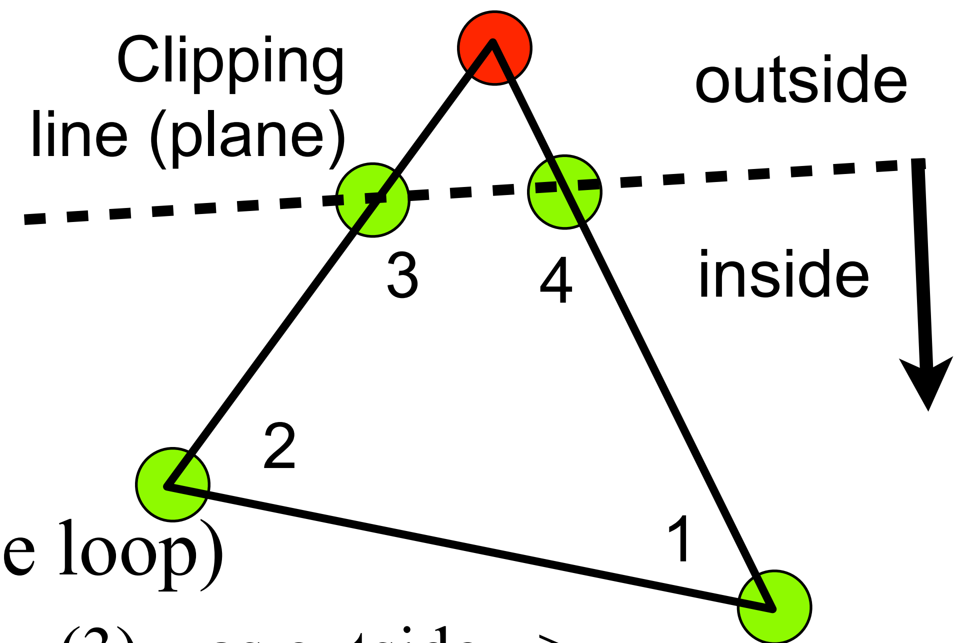
- Init: Output = empty
- Test vertex 1
  - It's inside, add to output
- Test vertex 2
  - Inside, add to output
- Test vertex 3 (outside)
  - Compute intersection, add it to output
- Test vertex 4=1 (to close loop)
  - It's inside, and last vertex (3) was outside => compute intersection, add to output **DONE**

Current: In



Sutherland-Hodgman Clipping

- Init: Output = empty
- Test vertex 1
  - It’s inside, add to output
- Test vertex 2
  - Inside, add to output
- Test vertex 3 (outside)
  - Compute intersection, add it to output
- Test vertex 4=1 (to close loop)
  - It’s inside, and last vertex (3) was outside => compute intersection, add to output **DONE**



More Information

- These links treat the “full” Sutherland-Hodgman algorithm that can also clip concave polygons
  – http://www.sunshine2k.de/stuff/Java/SutherlandHodgman/SutherlandHodgman.html
  – http://en.wikipedia.org/wiki/Sutherland-Hodgman_clipping_algorithm

  – Clipping triangles is an easier special case, don’t need to worry about concave inputs