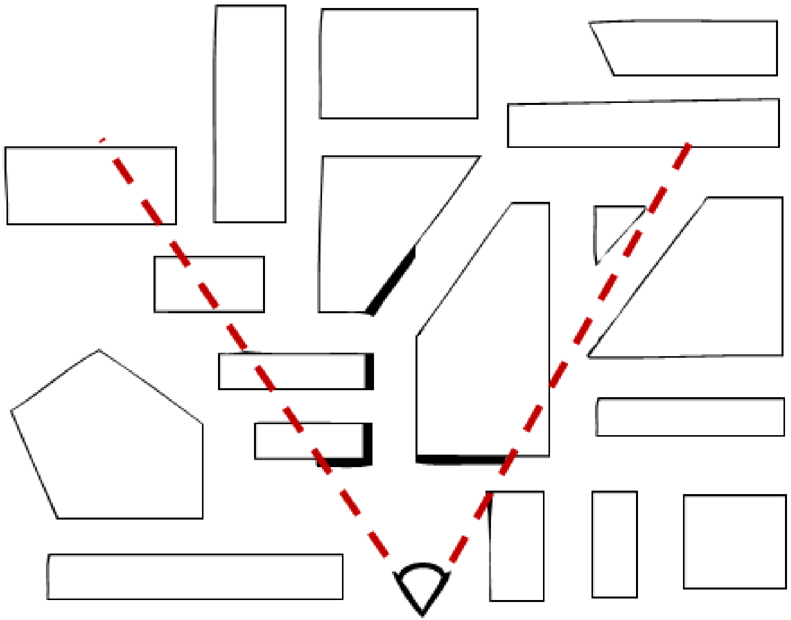


Visible Surface Determination (VSD)

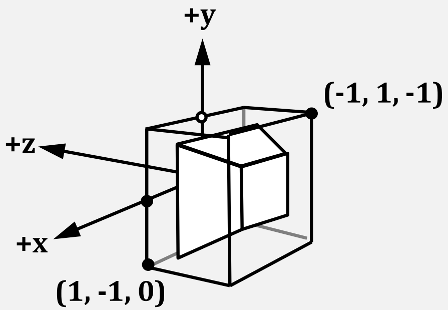
To render or not to render, that is the question…

What is it?

- Given a set of 3-D objects and a view specification (camera), determine which edges or surfaces of the object are visible
  - why might objects not be visible?
    occlusion vs. clipping
  - clipping works on the object level (clip against view volume)
  - occlusion works on the scene level (compare depth of object/edges/pixels against other objects/edges/pixels)
- Also called Hidden Surface Removal (HSR)
- We begin with some history of previously used VSD algorithms

Hardware Polygon Scan Conversion: Clipping

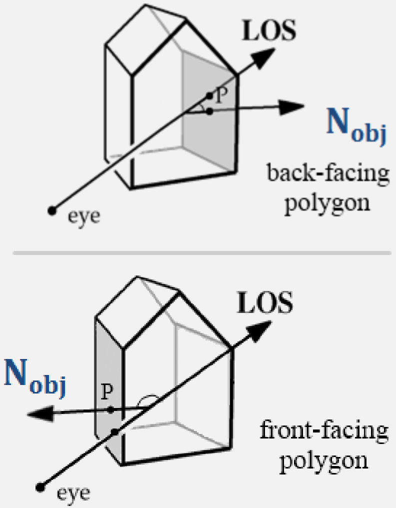
1



Canonical perspective-transformed view volume with cube

Perform backface culling 2

- If normal is facing in same direction as LOS (line of sight), it's a back face:
  - if $LOS \cdot N_{obj} > 0$, then polygon is invisible – discard
  - if $LOS \cdot N_{obj} < 0$, then polygon may be visible (if not, occluded)



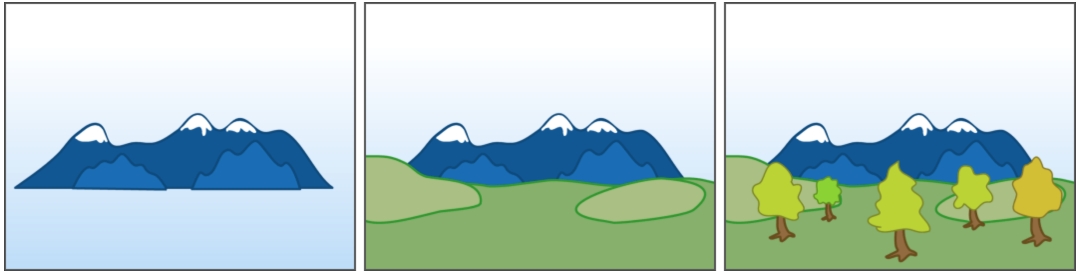
3

Finally, clip against normalized view volume

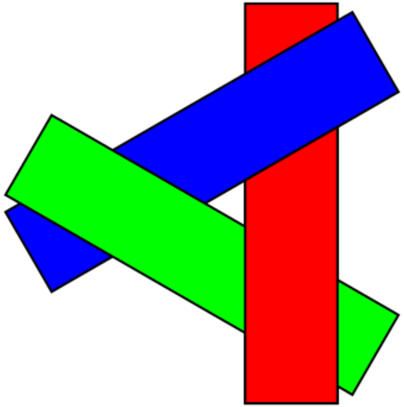
(-1 < x < 1), (-1 < y < 1), (-1 < z < 0)

Painter's Algorithm: occlusion

- Create drawing order so each polygon overwrites the previous one. This guarantees correct visibility at any pixel resolution

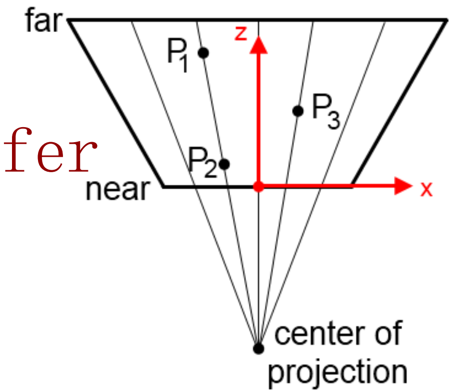


- Work back to front; find a way to sort polygons by depth (z), then draw them in that order
  - do a rough sort of polygons by smallest (farthest) z-coordinate in each polygon
  - scan-convert most distant polygon first, then work forward towards viewpoint ("painters' algorithm")
- Can this back-to-front strategy always be done?
  - problem: two polygons partially occluding each other – need to split polygons, very messy

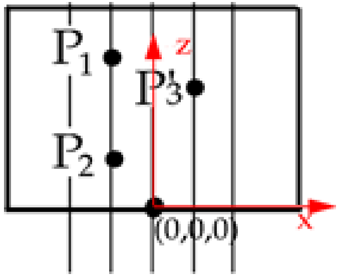


Interlocking polygons can cause the Painter's Algorithm to fail

Hardware Polygon Scan Conversion:Z-Buffer



- Determine object occlusion (point-by-point)
  - How to determine which point is closest?
    - i.e. P_2 is closer than P_1
  - In perspective view volume, have to compute projector and which point is closest along that projector – no projectors are parallel
  - Perspective transformation causes all projectors to become parallel
    - Simplifies depth comparison to z-comparison



- The ***Z-Buffer Algorithm***:
  - Z-buffer has scalar value for each screen pixel, initialized to far plane's z (maximum)
  - As each object is rendered, z value of each of its sample points is compared to z value in the same (x, y) location in z-buffer
  - If new point's z value less than or equal to previous one (i.e., closer to eye), its z-value is placed in the z-buffer and its color is placed in the frame buffer at the same (x, y); otherwise previous z value and frame buffer color are unchanged
  - Can store depth as integers or floats – z-compression a problem either way (see Viewing III - 38)
    - Integer still used in OGL

Z-Buffer Algorithm

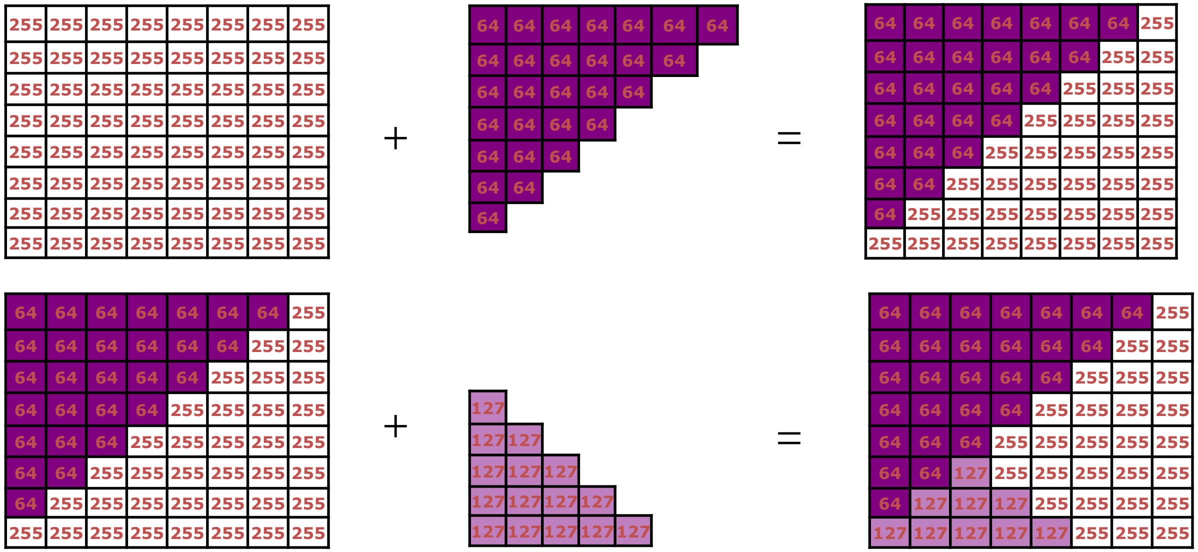
- Draw every polygon that we can't reject trivially (totally outside view volume)
- If we find a piece (one or more pixels) of a polygon that is closer to the front, we paint over whatever was behind it
- Use plane equation for polygon, z = f(x, y)
- Note: use positive z here [0, 1]
- Applet: http://debeissat.nicolas.free.fr/zbuffer.php

```
void zBuffer() {
   int x, y;
   for (y = 0; y < YMAX; y++)
   for (x = 0; x < XMAX; x++) {
       WritePixel (x, y, BACKGROUND_VALUE);
       WriteZ (x, y, 1);
   }
   for each polygon {
      for each pixel in polygon's projection {
         //plane equation
         double pz = Z-value at pixel (x, y);
         if (pz <= ReadZ (x, y)) {
            // New point is closer to front of view
            WritePixel (x, y, color at pixel (x, y))
            WriteZ (x, y, pz);
         }
      }
   }
}
```

Hardware Scan Conversion: VSD (3/4)

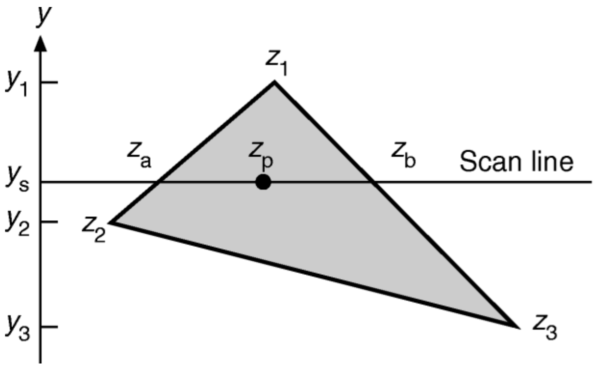
- Requires two "buffers"
  - Intensity Buffer: our familiar RGB pixel buffer, initialized to background color
  - Depth ("Z") Buffer: depth of scene at each pixel, initialized to 255
- Polygons are scan-converted in arbitrary order. When pixels overlap, use Z-buffer to decide which polygon "gets" that pixel

integer Z-buffer with near = 0, far = 255



Hardware Scan Conversion: VSD (4/4)

- After scene gets projected onto film plane we know depths only at locations in our depth buffer that our vertices got mapped to
- So how do we efficiently fill in all the "in between" z-buffer information?
- Simple answer: **incrementally**!
- Remember scan conversion/polygon filling? As we move along Y-axis, track x position where each edge intersects scan line
- Do the same for z coordinate with y-z slope instead of y-x slope



$$z_a = z_1 - (z_1 - z_2)\frac{y_1 - y_s}{y_1 - y_2}$$

$$z_b = z_1 - (z_1 - z_3)\frac{y_1 - y_s}{y_1 - y_3}$$

$$z_p = z_b - (z_b - z_a)\frac{x_b - x_p}{x_b - x_a}$$

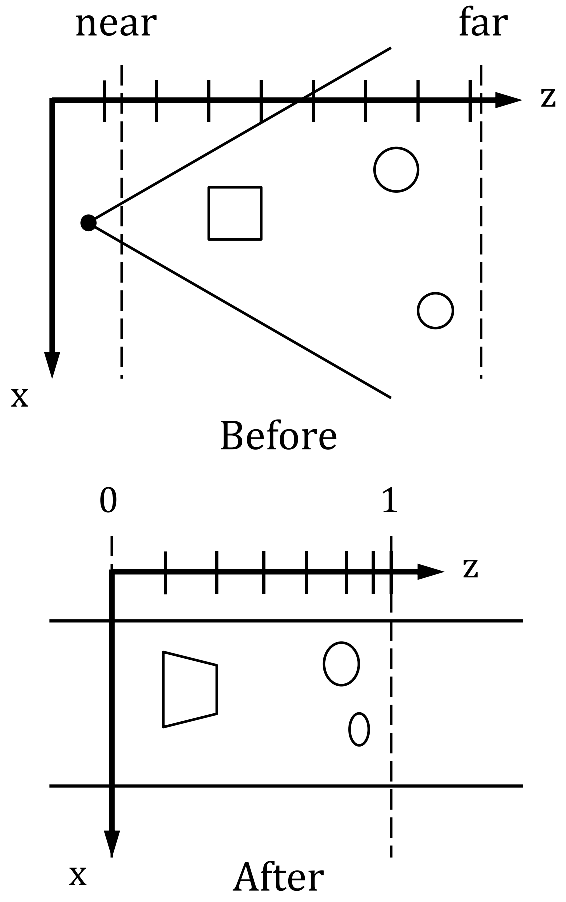
- Knowing z_1, z_2, and z_3 we can calculate z_a and z_b for each edge, and then incrementally calculate z_p as we scan.
- Similar to interpolation to calculate color per pixel (Gouraud shading)

Advantages of Z-buffer

- Dirt-cheap and fast to implement in hardware, despite brute force nature and potentially many passes over each pixel
- Requires no pre-processing, polygons can be treated in any order!
- Allows incremental additions to image – store both frame buffer and z-buffer and scan-convert the new polygons
  - Lost coherence/polygon id's for each pixel, so can't do incremental deletes of obsolete information.
- Technique extends to other surface descriptions that have (relatively) cheap z= f(x, y) computations (preferably incremental)

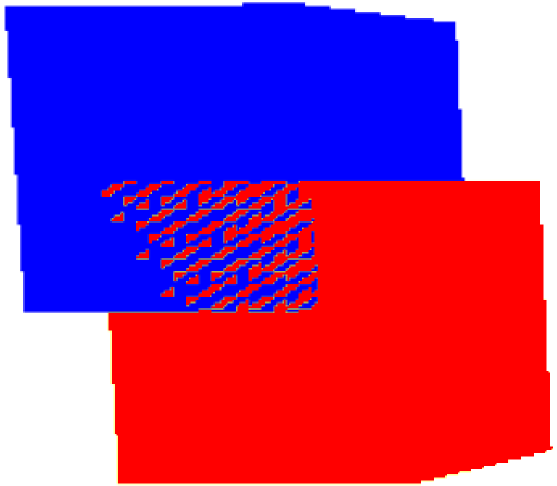
Disadvantages of Z-Buffer

- Perspective foreshortening
  - Compression in z-axis in post-perspective space
  - Objects far away from camera have z-values very close to each other
- Depth information loses precision rapidly
  - Leads to z-ordering bugs called z-fighting



Z-Fighting (1/4)

- Z-fighting occurs when two primitives have similar values in the z-buffer
  - Coplanar polygons (two polygons that occupy the same space)
  - One is arbitrarily chosen over the other, but z varies across the polygons and binning will cause artifacts, as shown on next slide
  - Behavior is deterministic: the same camera position gives the same z-fighting pattern



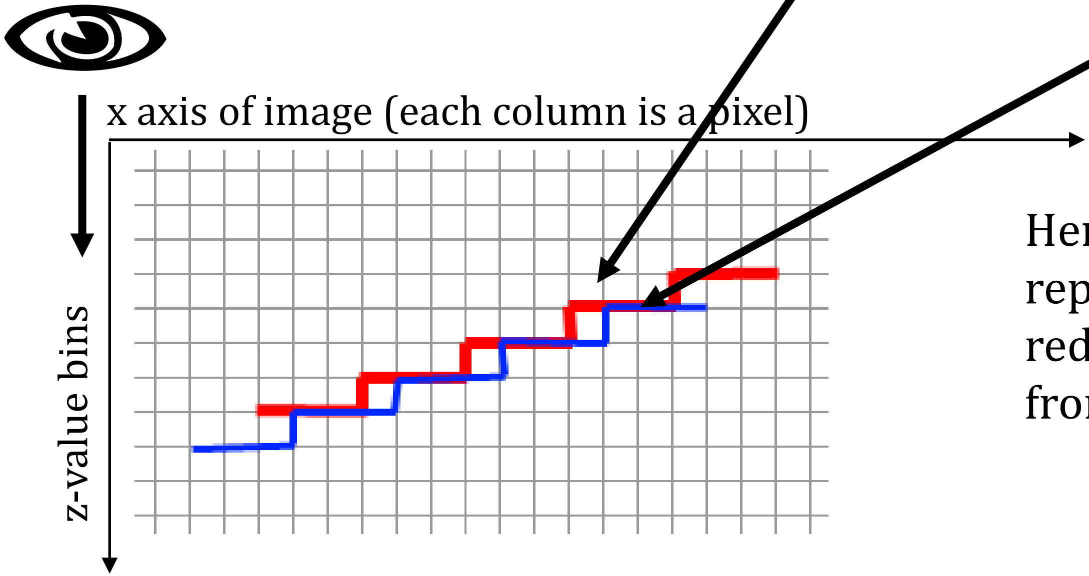
Two intersecting cubes

Z-Fighting (2/4)

Eye at origin,
Looking down Z axis

Red in front of blue

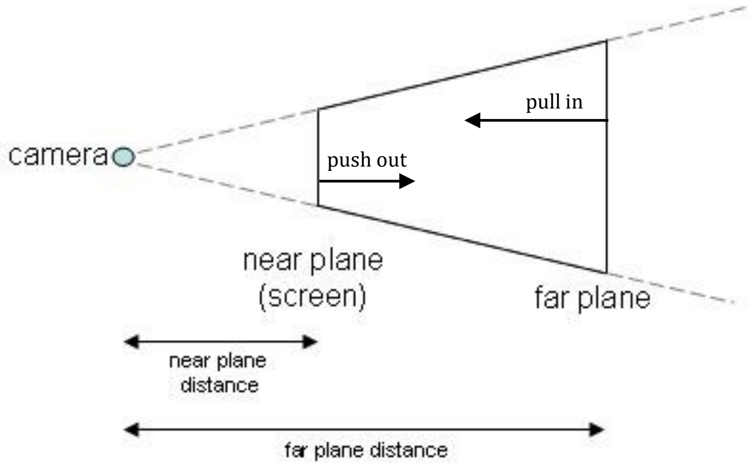
Blue, which is drawn after red, ends up in front of red [1]



[1] Overwrite if value in current z-value ≤ value in z-buffer

Here the red and blue lines represent cross-sections of the red and blue coplanar polygons from the previous slide

Z-Fighting (3/4)



- What if overwrite only if z-value is < current value in buffer?
  - The same problem will occur if the red polygon is drawn after the blue.
- What to do…
- To mitigate z-fighting, we can increase the precision of the depth buffer, and decrease the ratio $\frac{far}{near}$
  - Pull the far plane in, and the push near plane out
  - Bound the relevant part of the scene as tightly as possible
  - Don't want near plane too close to the eye
- If the ratio is too large, then unhinging transformation more likely to map large z-values to the same bin
  - Huge range has to be mapped to $[0,-1]$, further z-values in camera-space given very little of this range, squashed severely
  - Objects will small z-values are blown up, given a huge amount of this range (think of how distorted objects get when placed next to your eye)
  - Affects the homogenized z $= \frac{c-z}{z+zc}$ after projection (c = -near/far), very close to -1 for large z