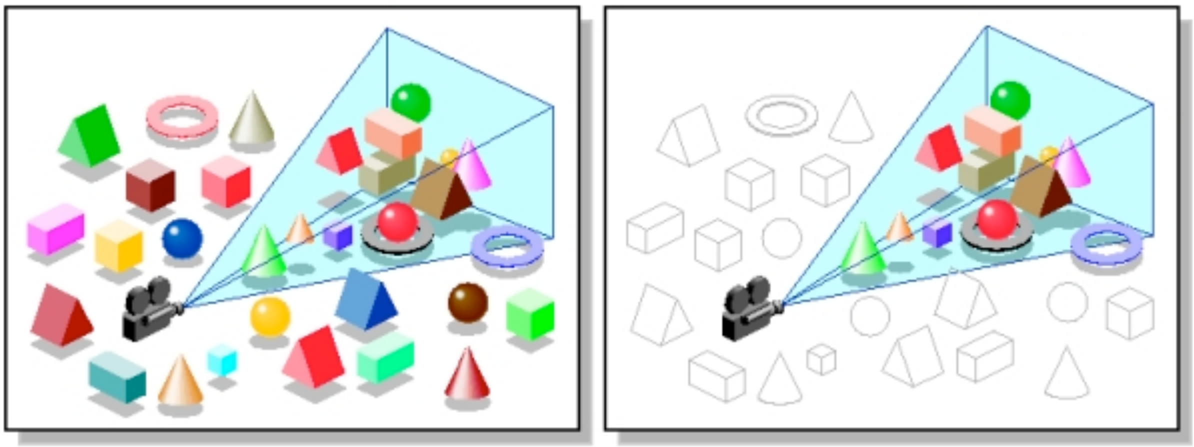
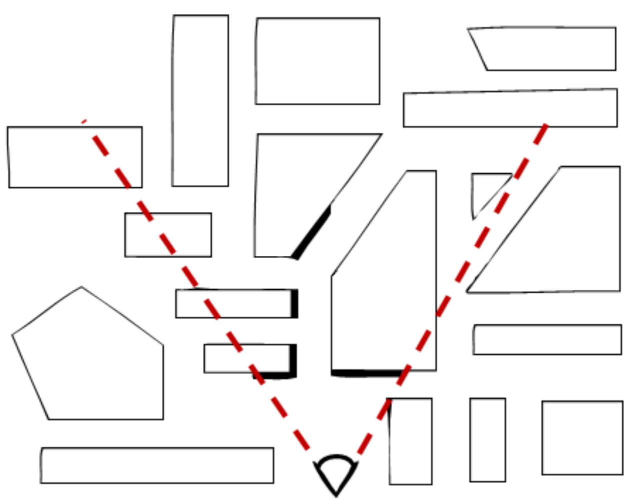
1. Clipping

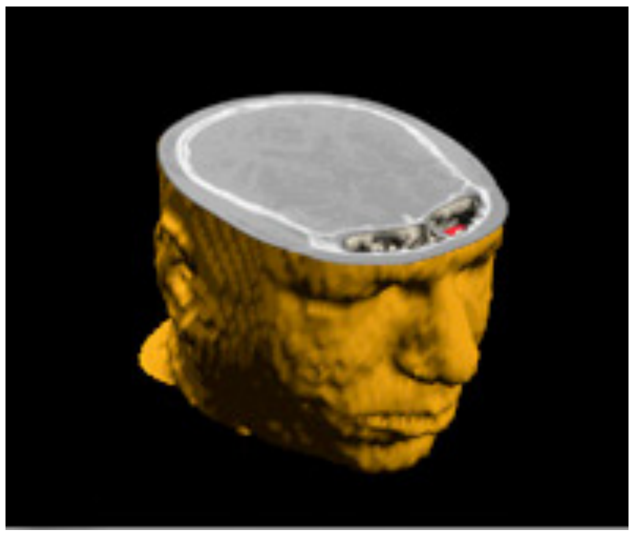


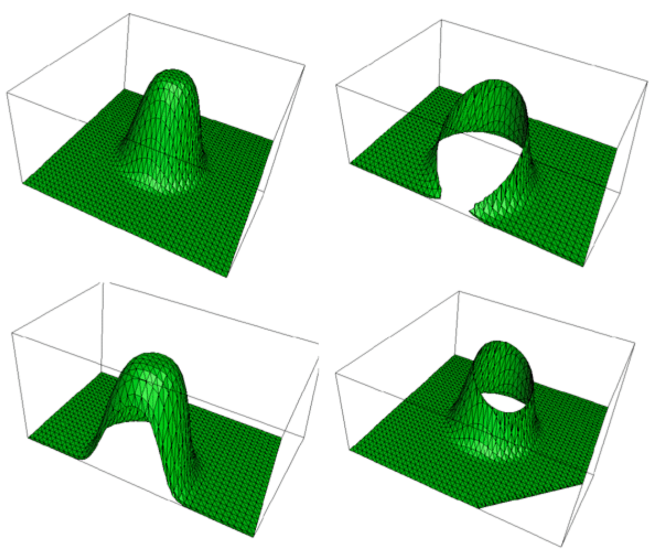
2. Visible Surface Determination



Clipping

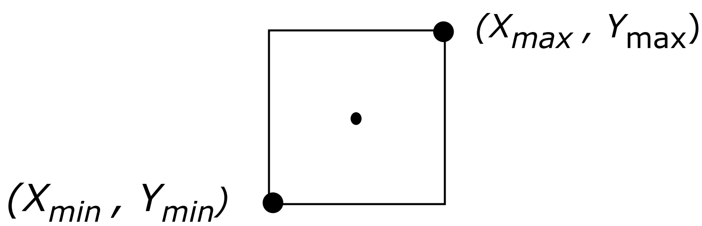
- part of the rendering pipeline, right before viewport mapping and scan conversion
- but also common in other apps





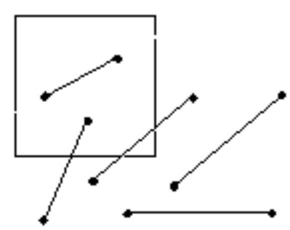
Line Clipping

- Clipping endpoints



$x_{min} \leq x \leq x_{max}$ **and** $y_{min} \leq y \leq y_{max}$ $\Longrightarrow$ point inside

- Endpoint analysis for lines:



  - if both endpoints in , do "trivial acceptance"
  - if one endpoint inside, one outside, must clip
  - if both endpoints out, don't know

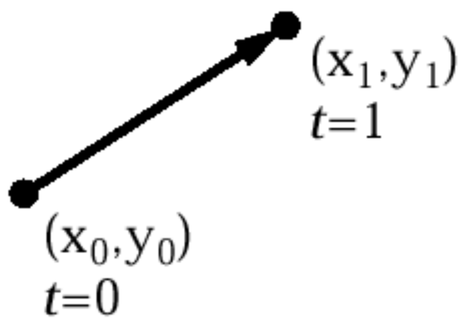
- Brute force clip: solve simultaneous equations using $y = mx + b$ for line and four clip edges
  - slope-intercept formula handles infinite lines only
  - doesn't handle vertical lines

Parametric Line Formulation For Clipping

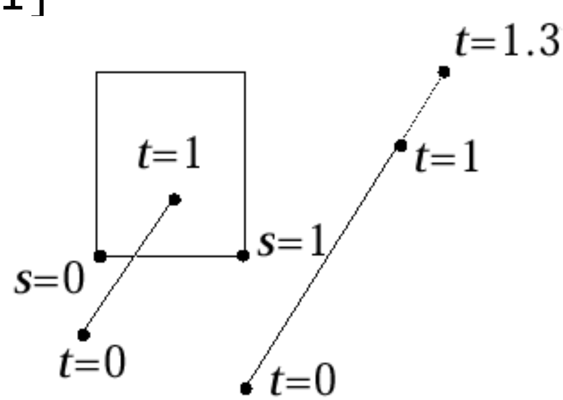
- Parametric form for line segment

$X = x_0 + t(x_1 - x_0) \qquad 0 \leq t \leq 1$
$Y = y_0 + t(y_1 - y_0)$

$P(t) = P_0 + t(P_1 - P_0)$

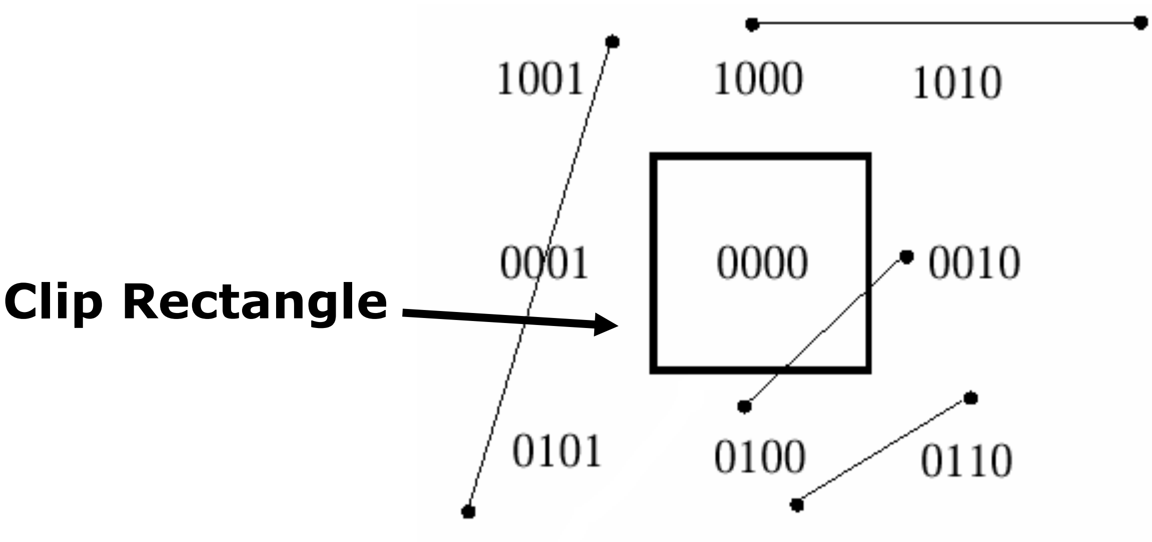


- "true," i.e., interior intersection, if s_{edge} and t_{line} in [0,1]



Outcodes for Cohen-Sutherland Line Clipping in 2D

- Divide plane into 9 regions
- Compute the sign bit of 4 comparisons between a vertex and an edge
  - $y_{max} - y$; $y - y_{min}$; $x_{max} - x$; $x - x_{min}$
  - point lies inside only if all four sign bits are 0, otherwise exceeds edge



- 4 bit outcode records results of four bounds tests:

First bit: outside halfplane of top edge, above top edge
Second bit: outside halfplane of bottom edge, below bottom edge
Third bit: outside halfplane of right edge, to right of right edge
Fourth bit: outside halfplane of left edge, to left of left edge

- Lines with $OC_0 = 0$ and $OC_1 = 0$ can be *trivially accepted*
- Lines lying entirely in a half plane outside an edge can be *trivially rejected*: OC_0 AND $OC_1 \neq 0$ (i.e., they share an "outside" bit)

Outcodes for Cohen-Sutherland Line Clipping in 3D

- Very similar to 2D
- Divide volume into 27 regions

Front plane			Center plane			Rear plane		
011001	011000	011010	001001	001000	001010	101001	101000	101010
010001	010000	010010	000001	000000	000010	100001	100000	100010
010101	010100	010110	000101	000100	000110	100101	100100	100110

- 6 bit outcode records results of four bounds tests:

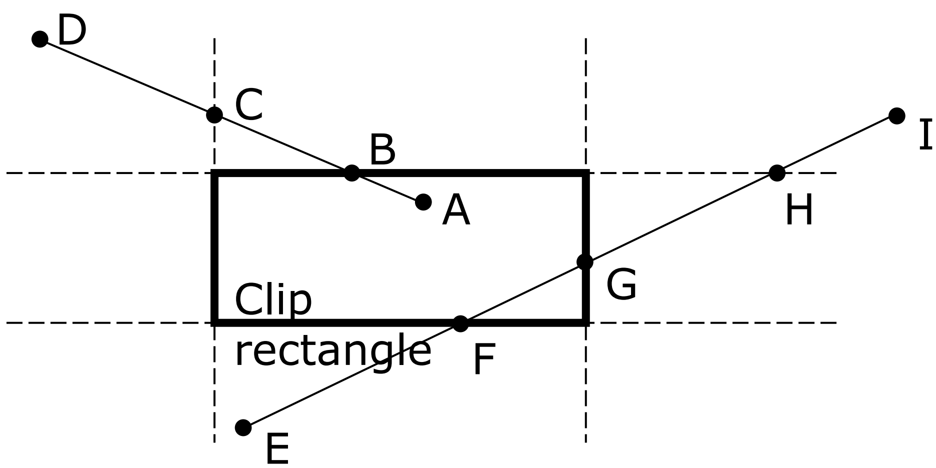
First bit: outside back plane, behind back plane
Second bit: outside front plane, in front of front plane
Third bit: outside top plane, above top plane
Fourth bit: outside bottom plane, below bottom plane
Fifth bit: outside right plane, to right of right plane
Sixth bit: outside left plane, to left of left plane

- Lines with $OC_0 = 0$ and $OC_1 = 0$ can be *trivially accepted*

- Lines lying entirely in a volume on outside of a plane can be *trivially rejected*: OC_0 AND $OC_1 \neq 0$ (i.e., they share an "outside" bit)

Cohen-Sutherland Algorithm

- If we can neither trivially reject/accept, divide and conquer
- subdivide line into two segments; then T/A or T/R one or both segments:



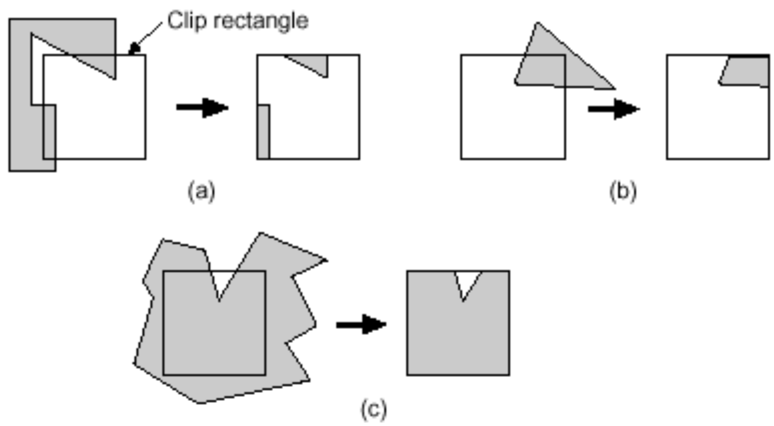
  - use a clip edge to cut line
  - use outcodes to choose edge that is crossed
    - if outcodes differ in the edge's bit, endpoints must straddle that edge
  - pick an order for checking edges
    - top – bottom – right – left
  - compute the intersection point
    - the clip edge fixes either *x* or *y*
    - can substitute into the line equation
  - iterate for the newly shortened line
  - "extra" clips may happen (e.g., E-I at H)

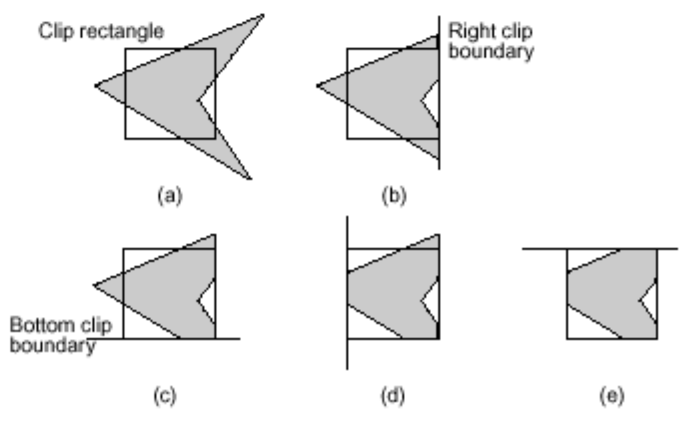
Pseudocode for the Cohen-Sutherland Algorithm

- y = y0 + slope*(x - x0) and x = x0 + (1/slope)*(y - y0)

```
ComputeOutCode(x0, y0, outcode0)
ComputeOutCode(x1, y1, outcode1)
repeat
    check for trivial reject or trivial accept
    pick the point that is outside the clip rectangle
    if TOP then
        x = x0 + (x1 – x0) * (ymax – y0)/(y1 – y0); y = ymax;
    else if BOTTOM then
        x = x0 + (x1 – x0) * (ymin – y0)/(y1 – y0); y = ymin;
    else if RIGHT then
        y = y0 + (y1 – y0) * (xmax – x0)/(x1 – x0); x = xmax;
    else if LEFT then
        y = y0 + (y1 – y0) * (xmin – x0)/(x1 – x0); x = xmin;
    end {calculate the line segment}
    if (outcode = outcode0) then
      x0 = x; y0 = y; ComputeOutCode(x0, y0, outcode0)
    else
      x1 = x; y1 = y; ComputeOutCode(x1, y1, outcode1)
    end {Subdivide}
until done
```

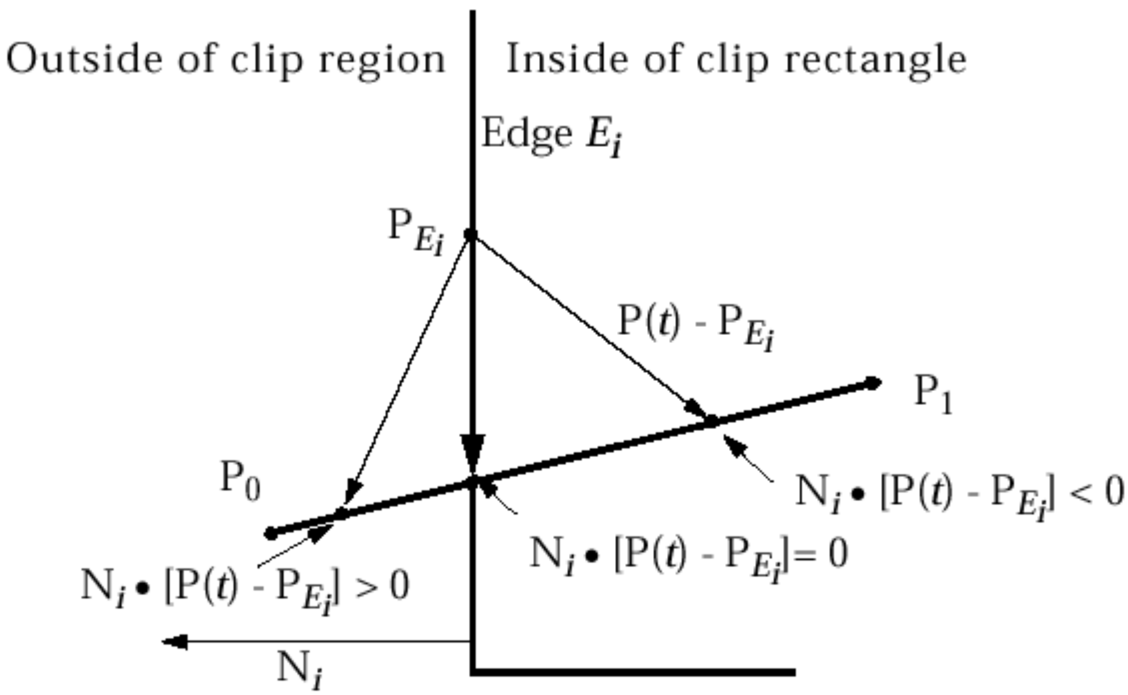
Sutherland-Hodgman Polygon Clipping





Cyrus-Beck/Liang-Barsky Parametric Line Clipping

- Uses parametric line formulation
  $P(t) = P_0 + (P_1 - P_0)t$
- Finds the four *t*'s for the four clip edges, then decides which form true intersections and calculate (*x*, *y*) for those only ($\leq$ 2)

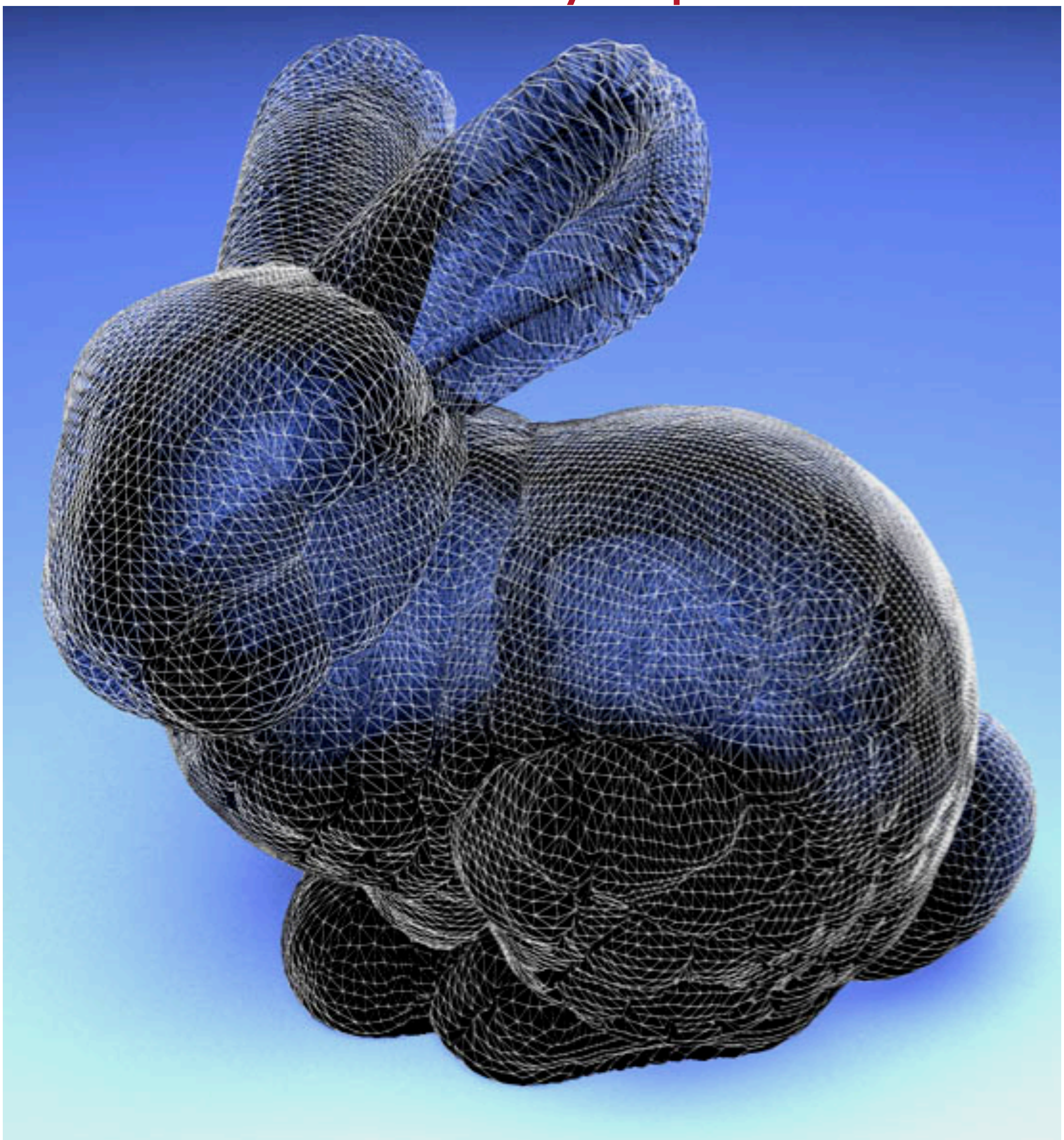


- Better than Cohen-Sutherland when many line segments in the scene intersect the clipping volume

Visible Surface Determination

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

How Many Ops?



Visible Surface Determination

Definition

- Given a set of 3-D objects and a view specification (camera), determine which lines or surfaces of the object are visible
  - you've already seen a VSD step…computing smallest non-negative t value along a ray
  - why might objects not be visible?
    - occlusion vs. clipping
  - clipping is one object at a time while occlusion is global
- Also called Hidden Surface Removal (HSR)

A computational trick: Culling

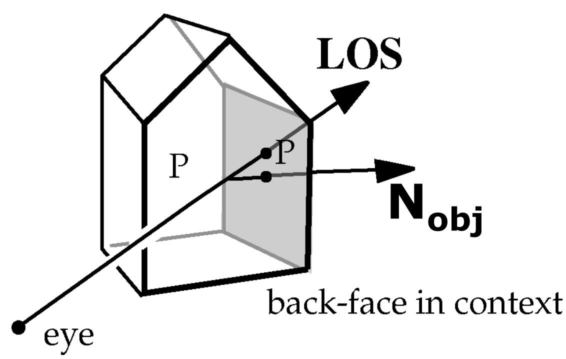
- Before performing the general VSD algorithm, apply heuristics to remove objects or object faces that are obviously not visible (*culling*)

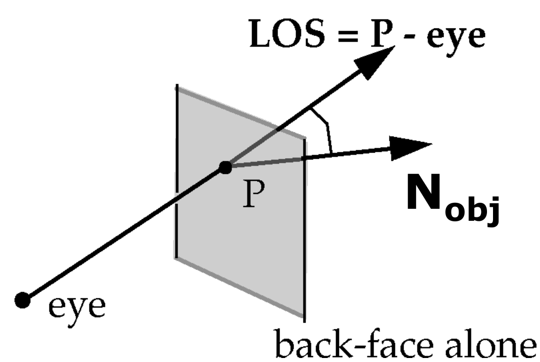
- Three common forms of culling:
  - View Frustum culling
    - if polygon does not lie within the view frustum (i.e., within the region that is visible to the user), then it does not need to be rendered
    - automatically eliminates polygons that lie behind the viewer
    - same as clipping – but the 3D version; Liang-Barsky can be generalized to do this
  - Back-face culling
  - Visibility culling (portals, occlusion culling)

Back-Face Culling

- Determines whether a polygon of a graphical object is a back face and thus invisible, depending on the position of the camera

- If normal is facing in same direction as LOS (line of sight), it's a back face:
  - if LOS • $N_{obj} \geq 0$, then polygon is invisible – discard
  - if LOS • $N_{obj} < 0$, then polygon may be visible





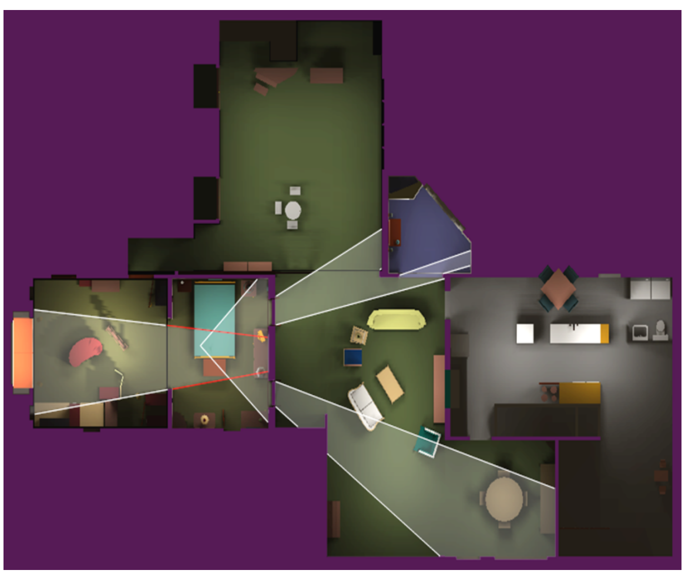
- Makes rendering objects quicker and more efficient by reducing the number of polygons for the program to draw.

- For example: in a city-street scene, there is no need to draw the polygons on the sides of the buildings facing away from the camera; they are completely occluded by the sides facing the camera.

Advanced Techniques (1/2)

Portals

- Indoor spaces are mostly rooms with doorways
- Why draw the geometry in the next room if the door is closed?
- If there is a portal (open door or hallway) only draw geometry visible through the portal
- Really useful as a pre-computation step – geometry visible through a portal remains constant - not too good for outdoor scenes

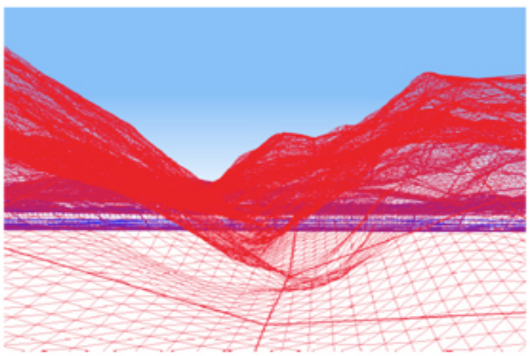


Picture Courtesy: David P. Luebke and Chris Georges, "Simple, Fast Evaluation of Potentially Visible Sets"
http://www.cs.virginia.edu/~luebke/publications/portals.html

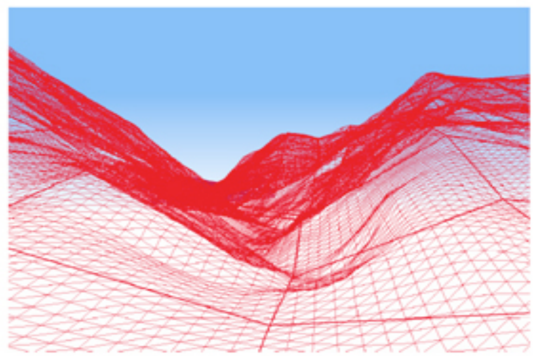
Advanced Techniques (2/2)

Occlusion Culling

- If a big object fills a good portion of the screen, don't draw geometry that it covers up
- Many new graphics cards have support for parts of the process



Without OC – lots of geometry drawn, most is not seen (drawn in blue)

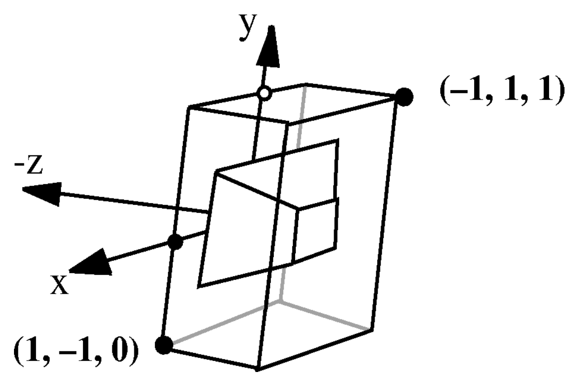


With OC – algorithm ignores blue geometry behind the hills

- Algorithm:
  - Create list of all objects potentially visible in frustum (per polygon or per shape)
  - For each pair of objects i and j, if i occludes j remove j
- $O(n^2)$! Lots of ways to make this faster:
  - Coorg, S., and S. Teller, "Real-Time Occlusion Culling for Models with Large Occluders", in *1997 Symposium on Interactive 3D Graphics*
  - Gamasutra overview of Occlusion Culling algorithms: http://www.gamasutra.com/features/19991109/moller_haines_01.htm
- Bad for indoor scenes with lots of small objects

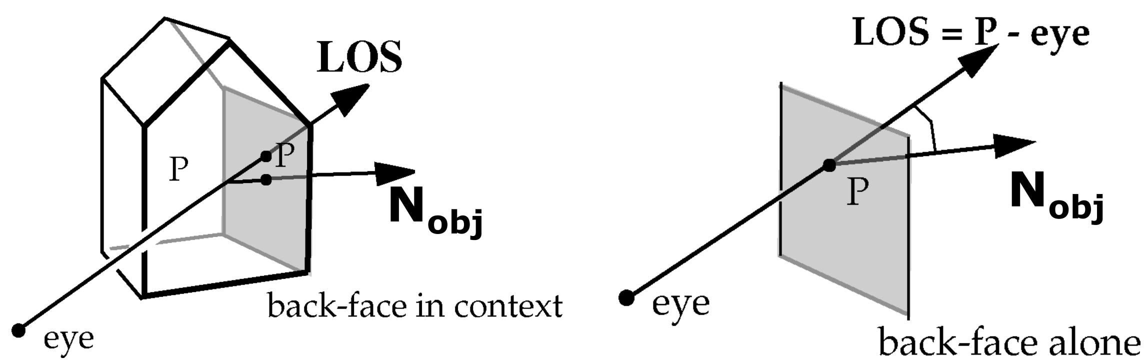
Visible Surface Determination (1/5)

- First apply perspective transformation on vertices; keep the z.



Canonical perspective-projection view volume with cube

- Perform backface culling; keep the z.

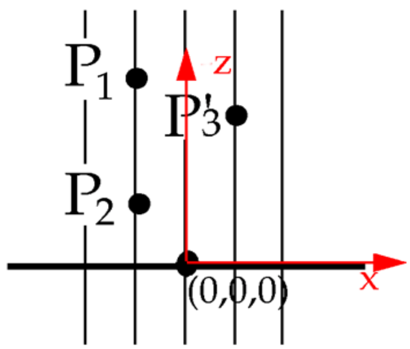


- Next clip against normalized view volume; keep the z
  ($-1 \leq x \leq 1$), ($-1 \leq y \leq 1$), ($0 \leq z \leq 1$)

- Last, VSD

Visible Surface Determination (2/5)

– VSD: need to determine object occlusion



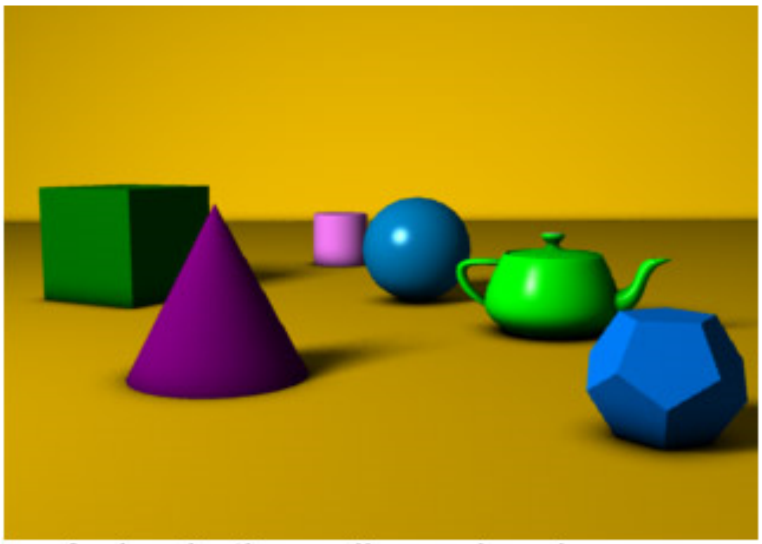
P2 should occlude P1

How do we determine which point is closer?

The *Z-buffer* algorithm

- Z-buffer is initialized to background value (furthest plane of view volume = 1.0)
- As each object is traversed, *z*-values of all its sample (pixel!) points are compared to *z*-value in same *(x, y)* location in Z-buffer
- If new point has *z* value less than previous one (i.e., closer to eye), its *z*-value is placed in z-buffer and its color placed in frame buffer at same *(x, y)*; otherwise previous z-value and frame buffer color are unchanged
- Can store depth as integers or floats or fixed points
  - i.e. for 8-bit (1 byte) integer z-buffer, set 0.0 -> 0 and 1.0 -> 255
  - far plane and precision of z-buffer can have dramatic effect on rendered image

Z-Buffer Algorithm (3/5)

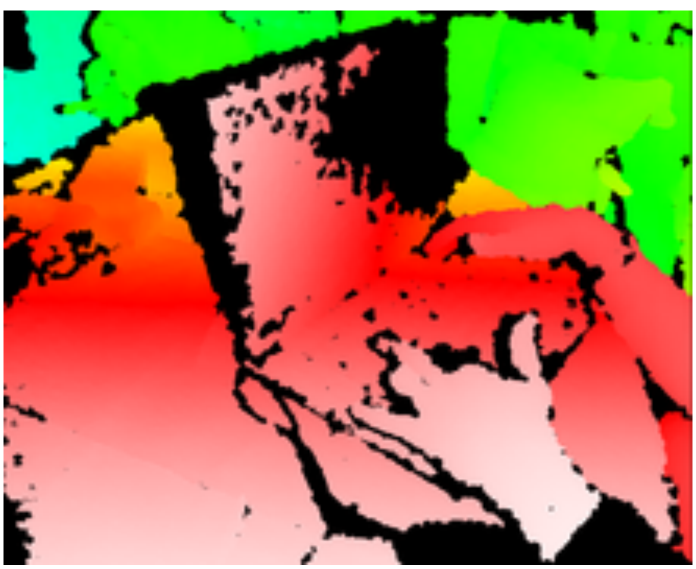


A simple three dimensional scene

Z-buffer representation

(Source: Wikipedia)

Aside: Kinect Depth Maps



Depth cameras go by many names: ranging camera, flash lidar, time-of-flight (ToF) camera, and RGB-D camera.

They all provide traditional (sometimes color) images and depth information for each pixel at framerate

The Kinect depth sensor uses an IR laser projector combined with a CMOS sensor (like the one in your smartphone camera)

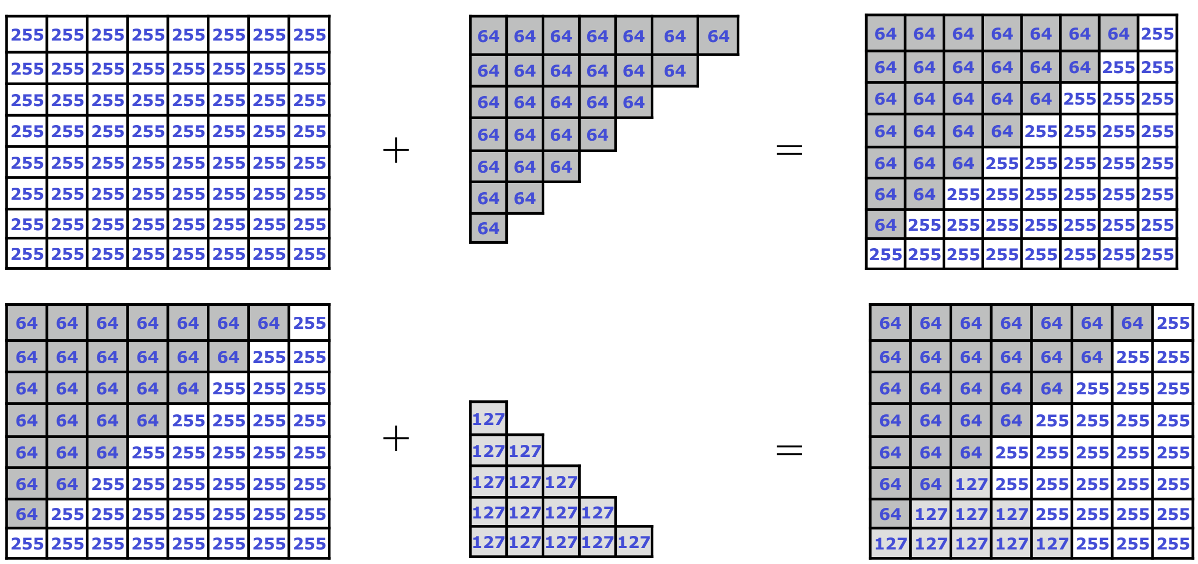
wikipedia.org

http://www.youtube.com/watch?feature=player_embedded&v=rm1JuukxhLQ#!

Z-Buffer Algorithm (4/5)

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 far depth = 255
- Polygons are scan-converted in arbitrary order. When pixels overlap, use Z-buffer to decide which polygon "gets" that pixel

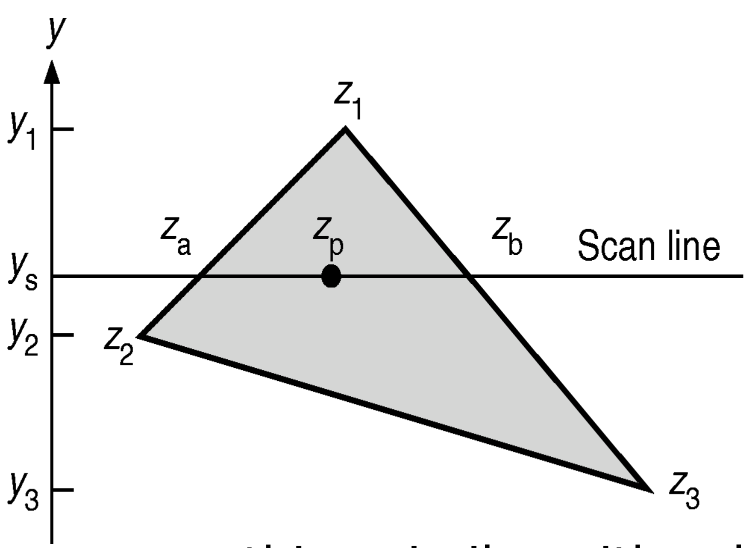


Above: example using integer Z-buffer with near = 0, far = 255; simplified example with all polygons perpendicular to Z

Z-Buffer Algorithm (5/5)

So how do we compute this efficiently?

- Do it incrementally!
- Remember scan conversion/polygon filling? As scan moves along Y-axis, track *x* position where each edge intersects scan-line
- Do same thing for *z* coordinate using calculations with *y-z* slope



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

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

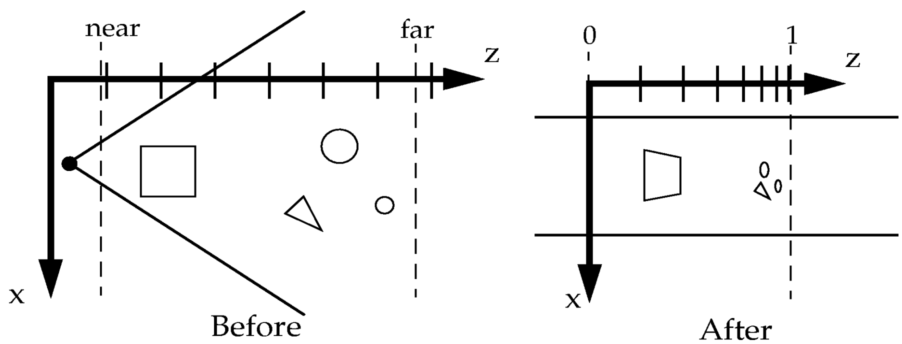
$$z_p = z_b - (z_b - z_a)\frac{x_b - x_p}{x_b - x_a}$$

something similar with calculating color per pixel... (Gouraud shading)

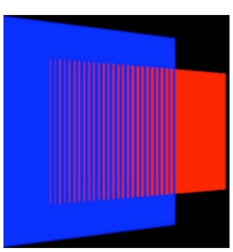
- brute force, but it is fast!
- no pre-sorting of polygons necessary

Z-Buffer Cons

- Precision problem: *perspective foreshortening*
  - this is a compression in the *z* axis caused by perspective foreshortening, which maps *z* to $\frac{z + k}{z - zk}$



  - objects that are far away from the camera end up having smaller Z-values that are very close to each other
  - depth information loses precision rapidly, which gives Z-ordering bugs (artifacts) for distant objects



  - co-planar polygons (e.g., shadows, reflections) exhibit "z-fighting" – you need to offset the further polygon slightly (can't have co-planar, yet intersecting polygons in nature)
  - floating-point values won't completely cure this problem

Z-Buffer Algorithm: Recap

- a.k.a. the Depth-Buffer Algorithm
- for each pixel we store in the frame buffer not only its color, but also its depth (a.k.a. its z-coordinate)

Initialize depth-buffer to the maximum depth possible value

For each point

- recall that after applying the perspective-with-depth transformation all depth values have been scaled to the range [-1;1]
- scale the depth value to the range of the depth-buffer and convert this to an integer
- if this depth <= crt depth at this point of the buffer, store the RGB value of point in the color buffer;
- otherwise do nothing

Summary

- Clipping
  - Cohen-Sutherland line clipping
    - pseudo bit codes to quickly T/A & T/R
    - fastest when most lines are either T/R or T/A
    - know how to mimic its behavior
  - Sutherland-Hodgman polygon clipping
    - a systematic approach to clipping polygons
    - know how to mimic its behavior
  - Cyrus-Beck/Liang-Barsky line clipping
    - alternative to C-S
    - faster when most lines need to be clipped

- Visible Surface Determination (VSD):
  - determine which lines or surfaces of an object are visible
  - culling
    - back-face, portals, occlusion culling
  - Z-buffer algorithm
    - know how it works
    - precision & z-fighting

- Both Clipping and VSD help reduce rendering load

The Graphics Card

- relieves the computer's main processor from much of the mundane repetitive effort involved in maintaining the frame buffer

- typically, it provides assistance for a number of operations including the following:

  - **Transformations:** Rotations and scalings used for moving objects and the viewer's location.

  - **Clipping:** Removing elements that lie outside the viewing window.

  - **Projection:** Applying the appropriate perspective transformations.

  - **Shading and Coloring:** The color of a pixel may be altered by increasing its brightness. Simple shading involves smooth blending between some given values. Modern graphics cards support more complex procedural shading.

  - **Texturing:** Coloring objects by "painting" textures onto their surface. Textures may be generated by images or by procedures.

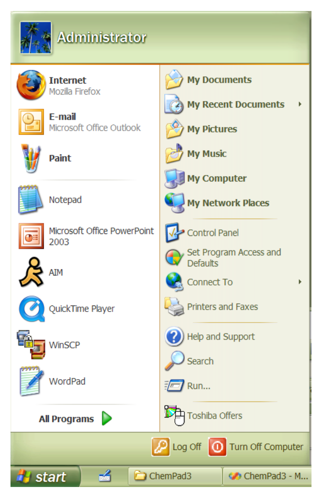
  - **Hidden-surface elimination:** Determines which of the various objects that project to the same pixel is closest to the viewer and hence is displayed.

Bird's Eye View of the Course

- Basic 3D scene management
  - tessellation of curved surfaces
  - transformation (translation, rotation, scale)
  - scenegraph traversal
  - virtual camera model; viewing
  - intersecting rays with simple solids

- 2D raster graphics
  - scan conversion
  - clipping/VSD
  - color

- Modeling and rendering
  - lighting and shading of polygonal models
  - texture mapping
  - raytracing



- Other Topics
  - animation
  - user interfaces
  - video games