CS100433
Visible Surface Detection

Junqiao Zhao 赵君峤

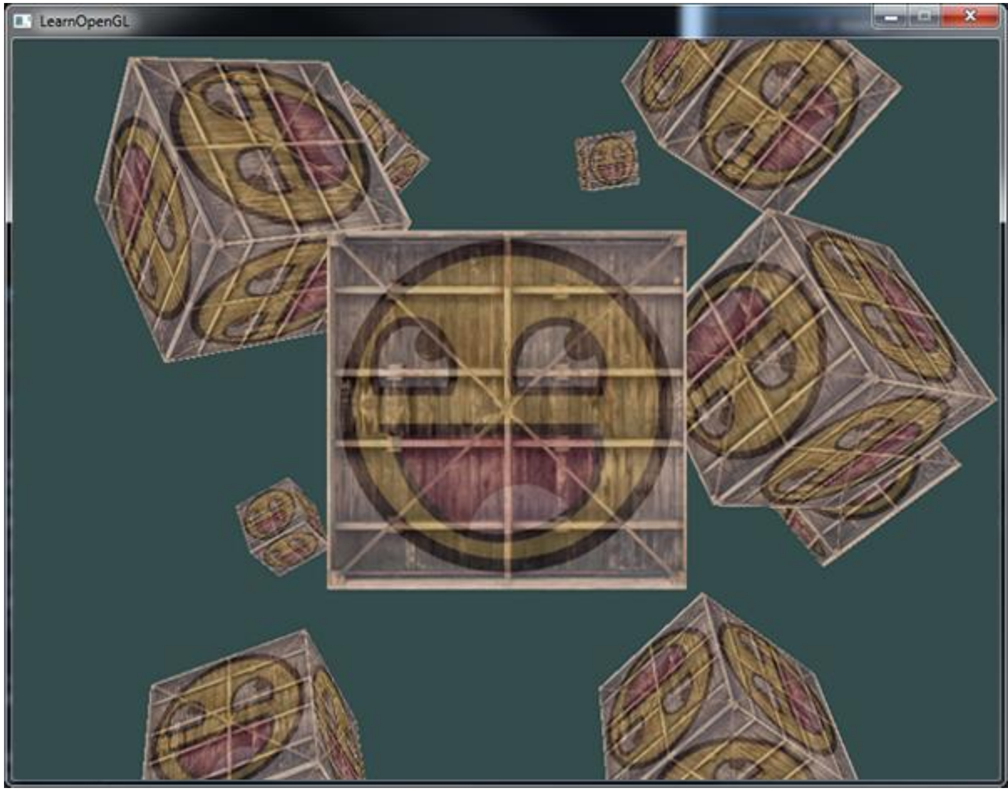
Department of Computer Science and Technology

College of Electronics and Information Engineering

Tongji University

Why?

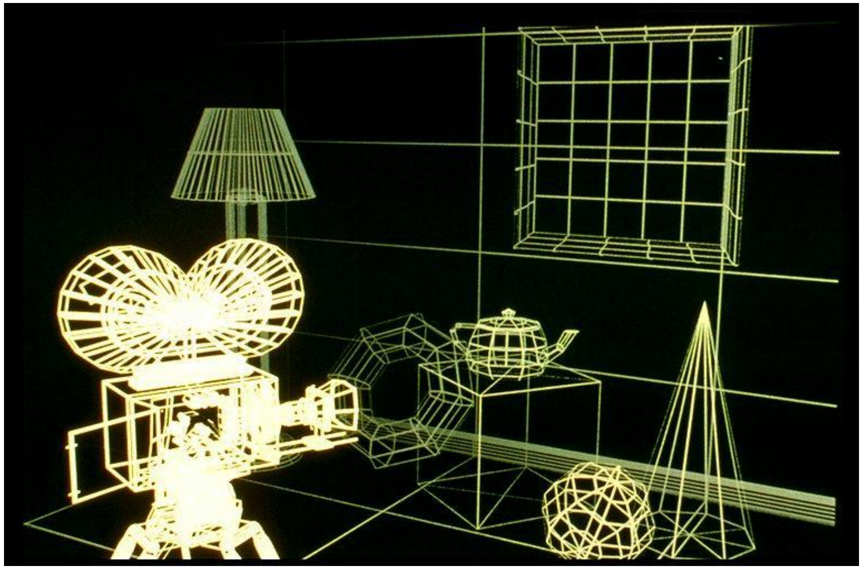
- glEnable(GL_DEPTH_TEST)
- glDepthFunc(GL_LESS)

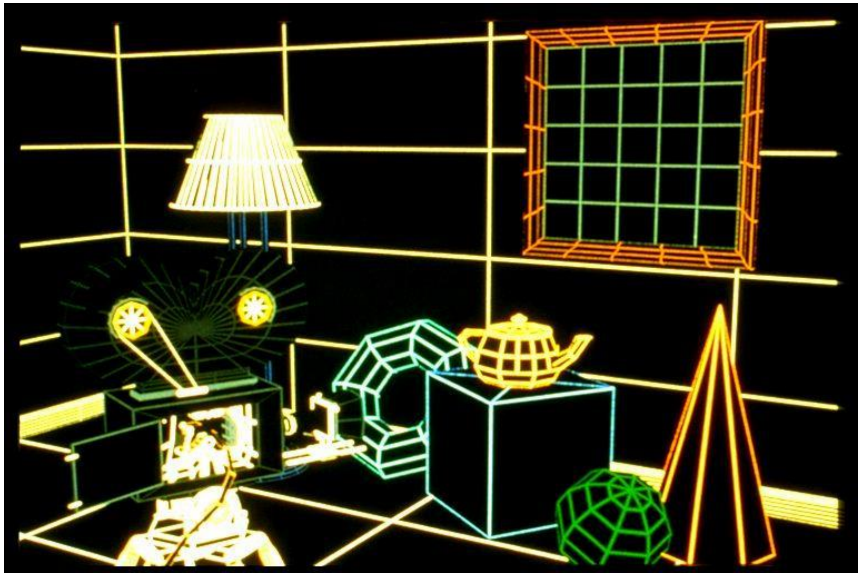


(http://learnopengl.com/)

Why?

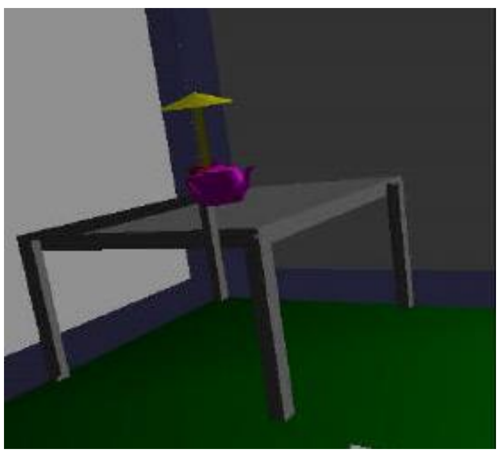
- Depth cueing
- Output sensitive

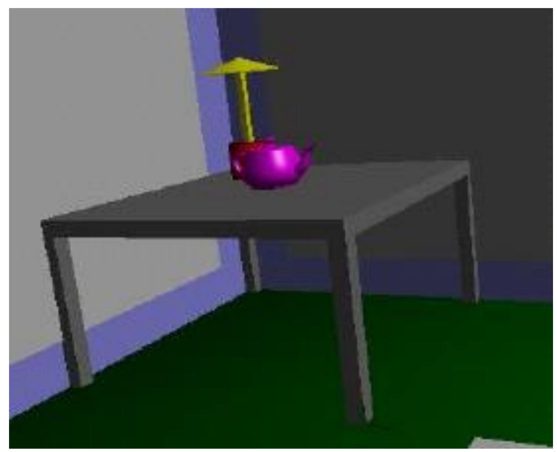




Visibility (hidden surface removal)

- A correct rendering requires correct visibility calculations
- Correct visibility
  - when multiple opaque polygons cover the same screen space, only the closest one is visible (remove the other hidden surfaces)


wrong visibility


correct visibility

Output Sensitive

- Drawing polygonal faces on screen consumes CPU cycles
  - Illumination
- We cannot see every surface in scene
  - We don’t want to waste time rendering primitives which don’t contribute to the final image.

Visibility of primitives

- A scene primitive can be invisible for 3 reasons:
  - Primitive lies outside field of view (Clipping)
  - Primitive is *back-facing*
  - Primitive is occluded by one or more objects nearer the viewer

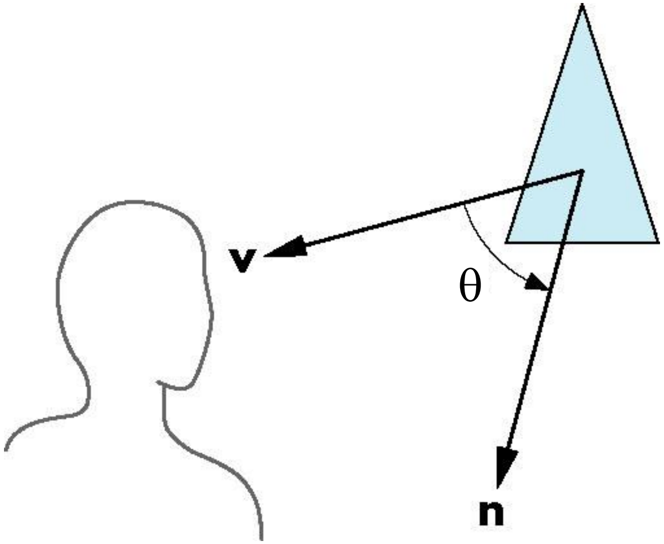
Visible surface algorithms.

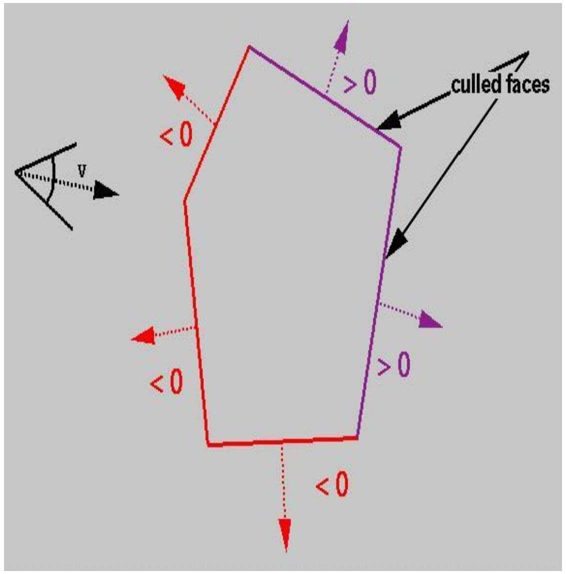
- Object space techniques: applied before vertices are mapped to pixels
  - Back face culling, Painter’s algorithm, BSP trees etc.

- Image space techniques: applied while the vertices are rasterized
  - Z-buffering

Back-Face Removal (Culling)

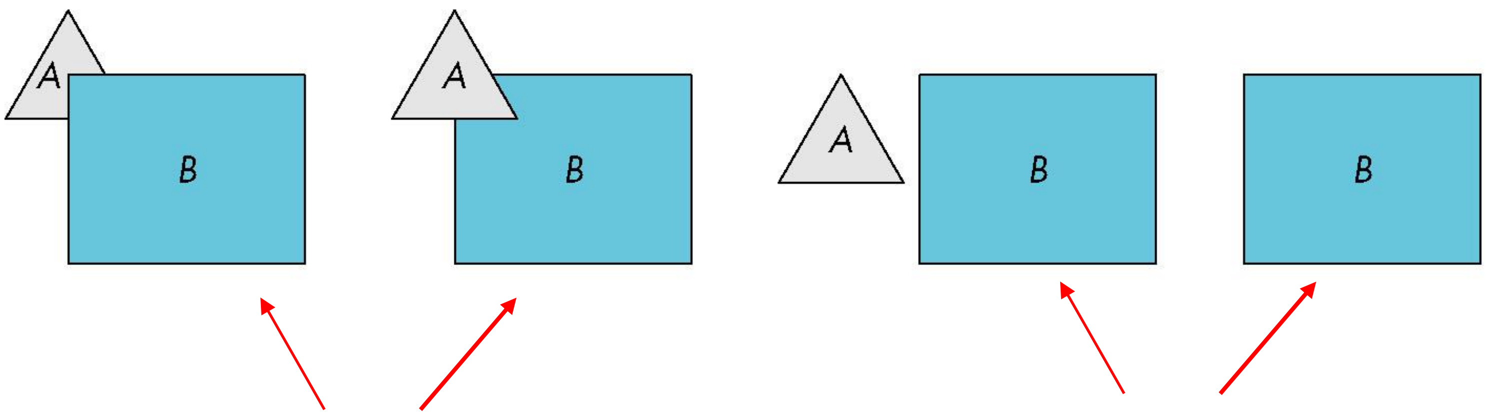
- Face is visible iff $90 \geq \theta \geq -90$
- Equivalently $\cos\theta \geq 0$ or $\mathbf{v} \bullet \mathbf{n} \geq 0$
- In eye coordinate frame need only test the sign of the Z component
- **GL_CULL_FACE**
- **In other cases back-face culling should be disabled**





Hidden Surface Removal

- Object-space approach: use pairwise testing between polygons (objects)



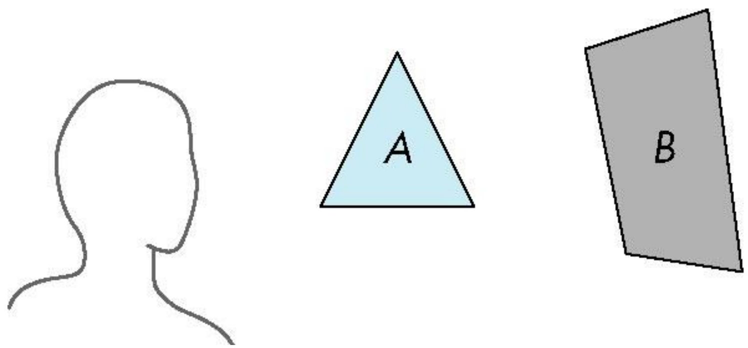
partially obscuring

can draw independently

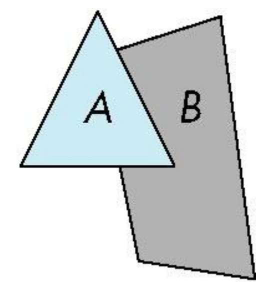
- Worst case complexity $O(n^2)$ for n polygons

Painter’s Algorithm

- Render polygons a back to front order so that polygons behind others are simply painted over



B behind A as seen by viewer



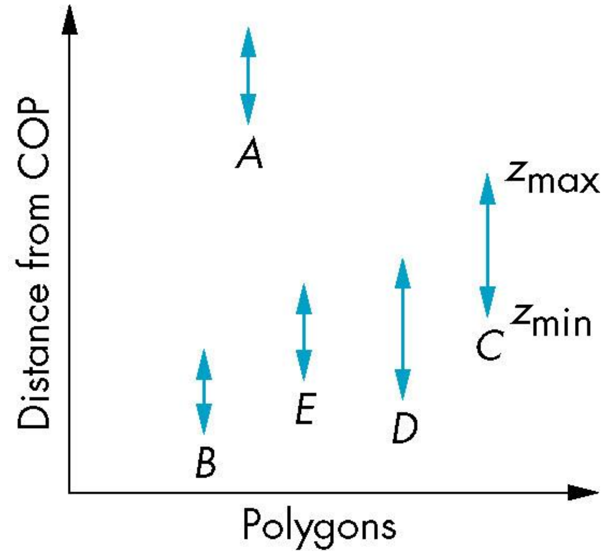
Fill B then A

Depth Sort

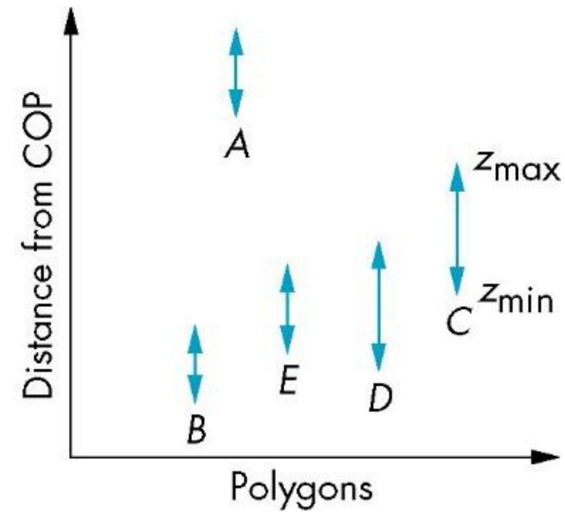
- Requires ordering of polygons first
  - O(n log n) calculation for ordering
  - Not every polygon is either in front or behind all other polygons

- Order polygons and deal with easy cases first, harder later

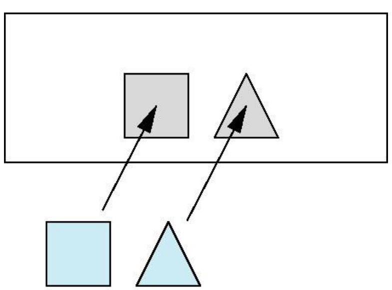
Polygons sorted by distance from COP

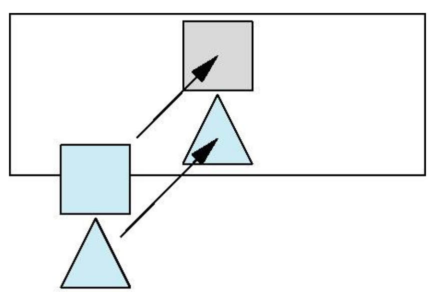


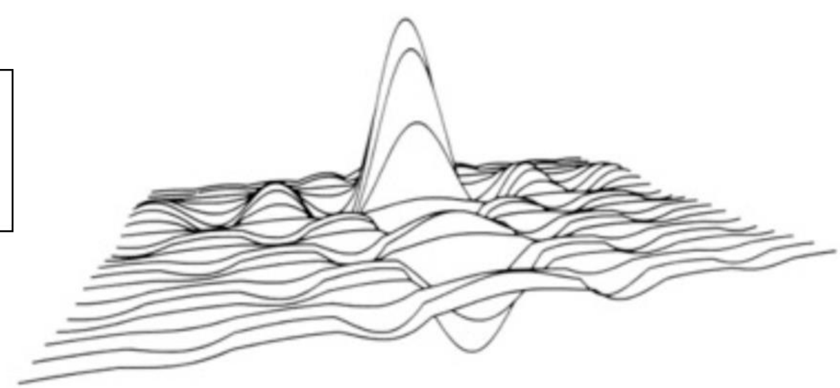
Easy Cases



- A lies behind all other polygons
  - Can render
- Polygons overlap in z but not in either x or y
  - Can render independently
- Useful when a valid order is easy to come by

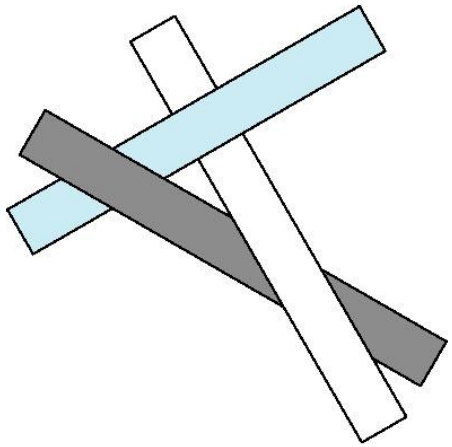




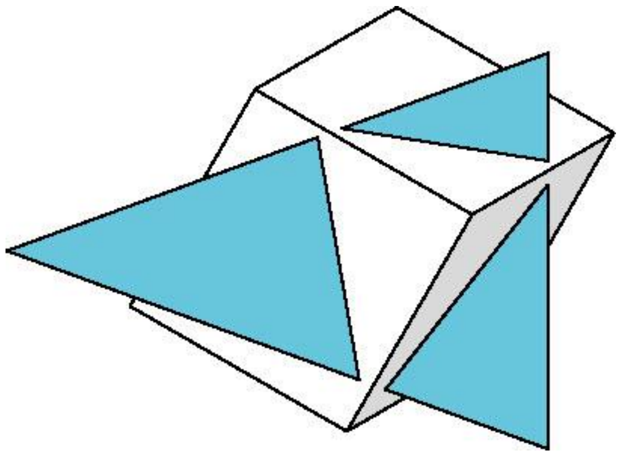


Hard Cases

- Difficult to sort without decomposition!



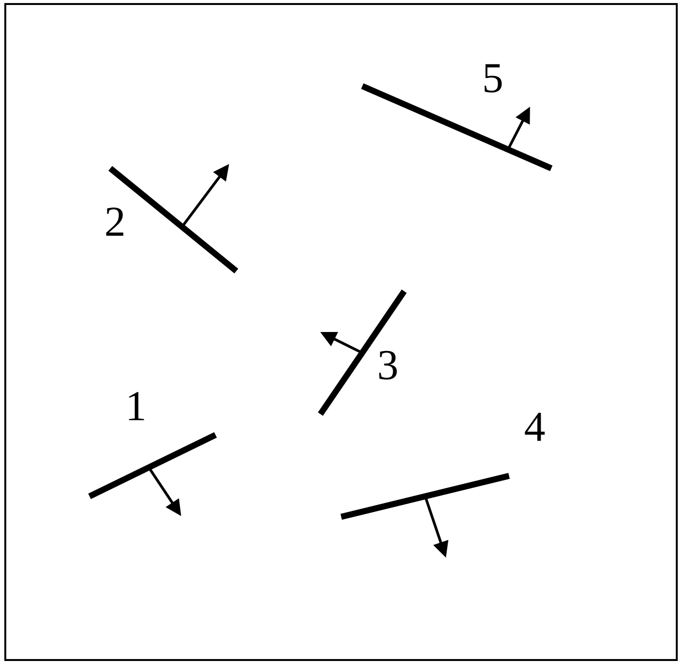
cyclic overlap



penetration

BSP (Binary Space Partitioning) Tree

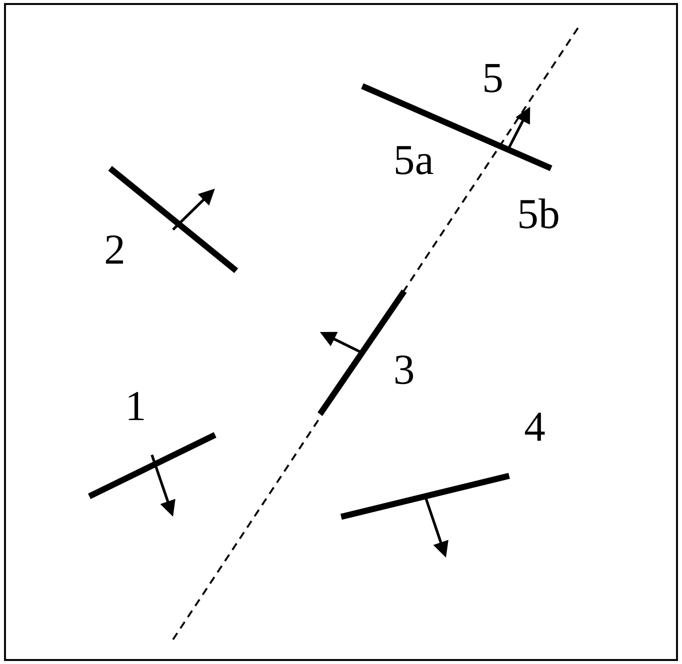
- One of class of “list-priority” algorithms – returns ordered list of polygon fragments for specified view point (static pre-processing stage)

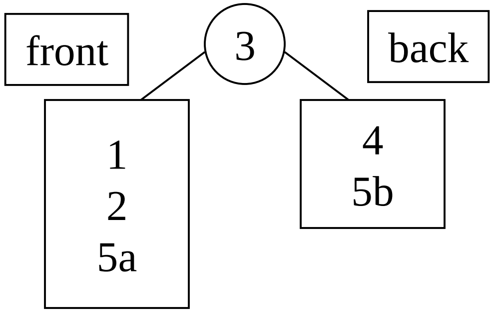


View of scene from above

BSP Tree

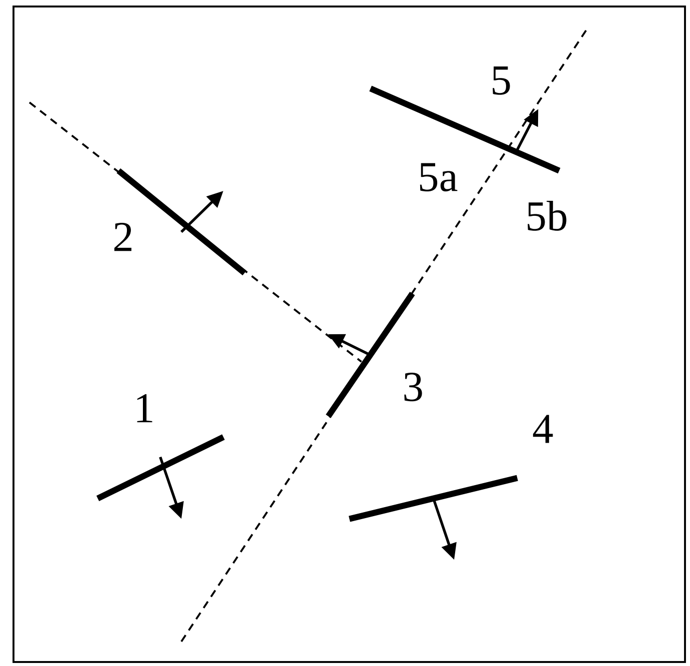
- **Choose polygon arbitrarily**
- **Divide scene into front (relative to normal) and back half-spaces.**
- **Split any polygon lying on both sides**.
- Choose a polygon from each side – split scene again.
- Recursively divide each side until each node contains only 1 polygon.

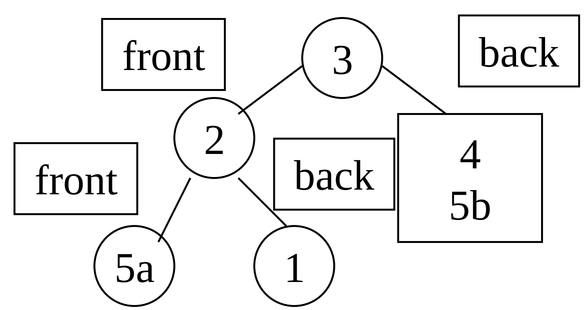




BSP Tree

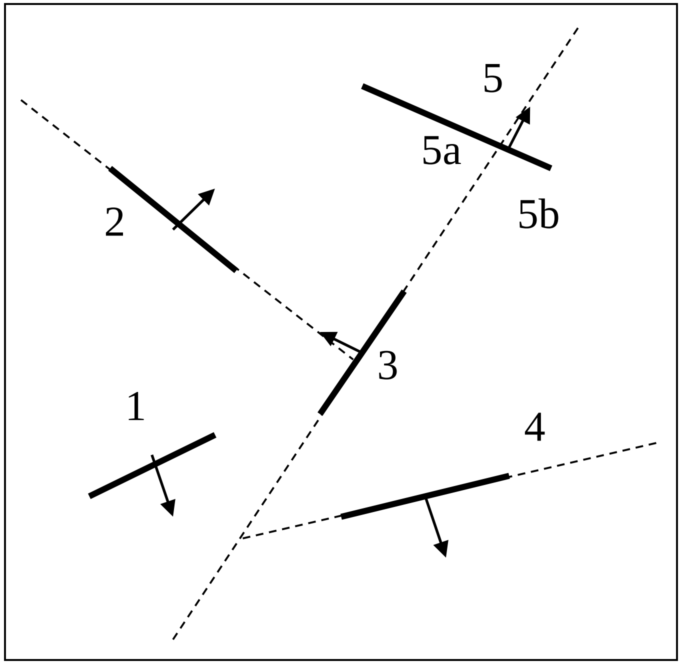
- Choose polygon arbitrarily
- Divide scene into front (relative to normal) and back half-spaces.
- Split any polygon lying on both sides.
- **Choose a polygon from each side – split scene again.**
- Recursively divide each side until each node contains only 1 polygon.

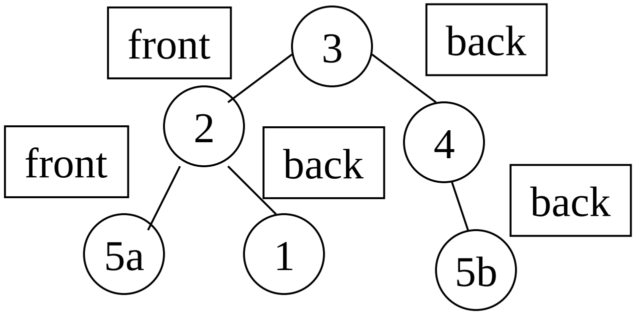




BSP Tree

- Choose polygon arbitrarily
- Divide scene into front (relative to normal) and back half-spaces.
- Split any polygon lying on both sides.
- Choose a polygon from each side – split scene again.
- **Recursively divide each side until each node contains only 1 polygon.**





Displaying a BSP tree.

- Once we have the regions – need priority list
- BSP tree can be traversed to yield a correct priority list for an arbitrary viewpoint.
- Start at root polygon.
  - If viewer is in front half-space, draw polygons behind root first, then the root polygon, then polygons in front.
  - If viewer is in back half-space, draw polygons in front of root first, then the root polygon, then polygons in the back.
  - If polygon is on edge – either can be used.
  - Recursively descend the tree.
- If eye is in rear half-space for a polygon – then can back face cull.

BSP Tree

- A lot of computation required at start.
  - Try to split polygons along good dividing plane
  - Intersecting polygon splitting may be costly
- Cheap to check visibility once tree is set up.
- Can be used to generate correct visibility for arbitrary views.

$\Rightarrow$Efficient when objects don't change very often in the scene.

Octree

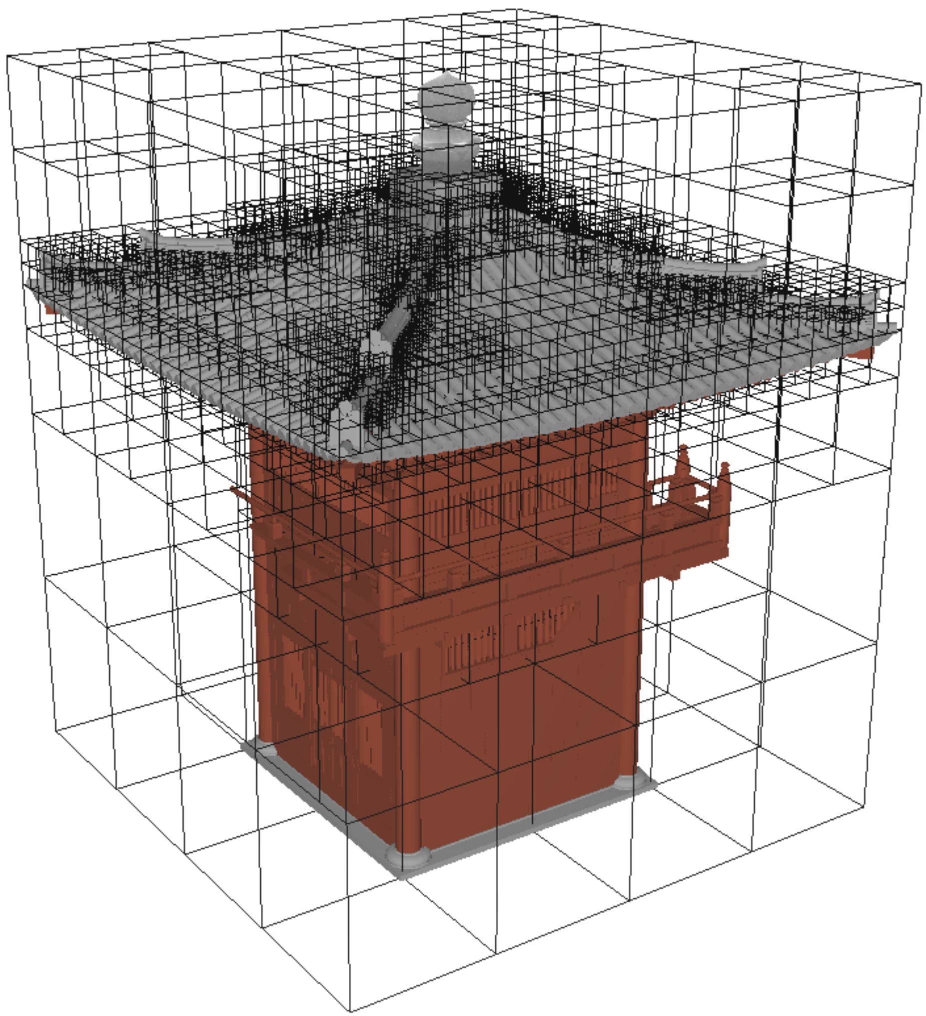
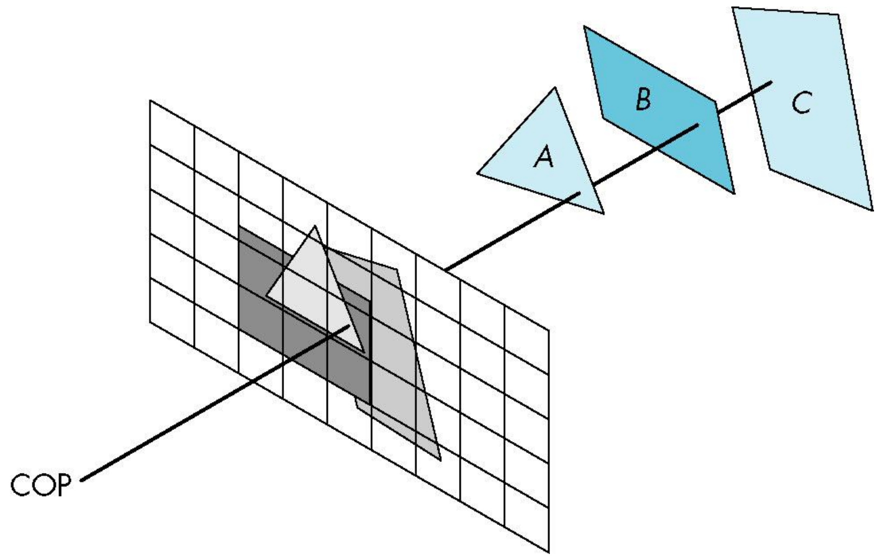


Image Space Approach

- Look at each projector (nm for an $n \times m$ frame buffer) and find closest of k polygons
- Complexity $O(n*m*k)$
- Ray tracing/casting
- z-buffer
- **Because maintaining**

a Z sort is expensive

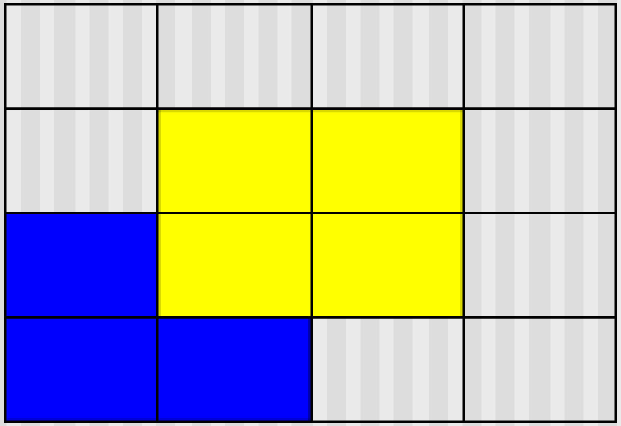


Z-buffering (depth buffer)

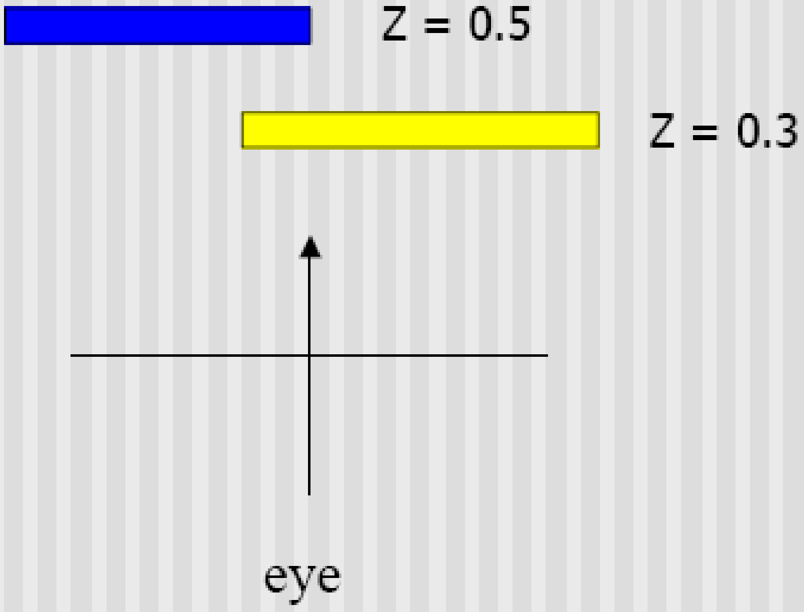
Basic Z-buffer idea:

- rasterize every input polygon
- For every pixel in the polygon interior, calculate its corresponding z value (by interpolation)
- Track depth values of closest polygon (smallest z) so far
- Paint the pixel with the color of the polygon whose z value is the closest to the eye.
- **Draw in any order, keep track of closest**

Z buffer example



Correct Final image



Top View

Z buffer example

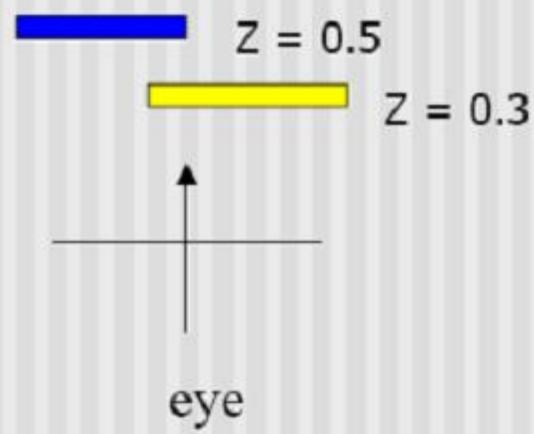
Step 1: Initialize the depth buffer

1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0

Z buffer example

Step 2: Draw the blue polygon (assuming the program draws blue polyon first – the order does not affect the final result any way).

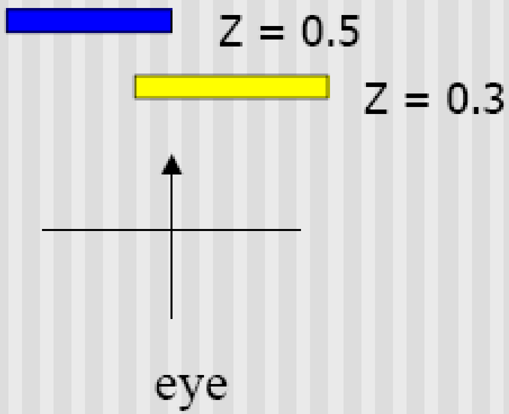
1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
0.5	0.5	1.0	1.0
0.5	0.5	1.0	1.0



Z buffer example

Step 3: Draw the yellow polygon

1.0	1.0	1.0	1.0
1.0	0.3	0.3	1.0
0.5	0.3	0.3	1.0
0.5	0.5	1.0	1.0



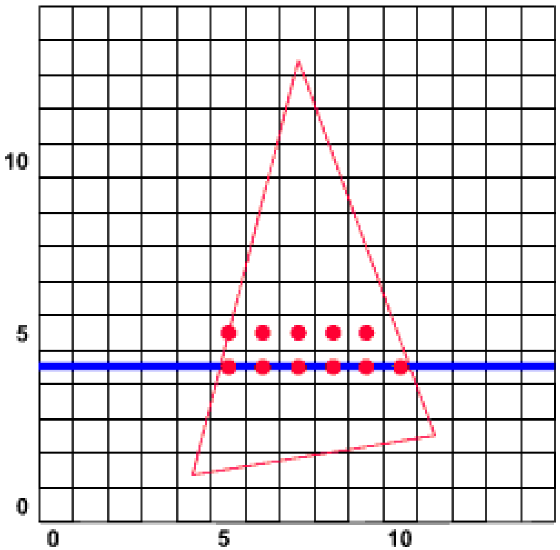
z-buffer drawback: wastes resources by rendering a face and then drawing over it

Implementation

- Initialise frame buffer to background colour.
- Initialise depth buffer to z = max. value for far clipping plane
- For each triangle
  - Calculate value for z for each pixel inside
  - Update both frame and depth buffer

Filling in Triangles

- Scan line algorithm
  - Filling in the triangle by drawing horizontal lines from top to bottom
- Barycentric coordinates
  - Checking whether a pixel is inside / outside the triangle



Triangle Rasterization

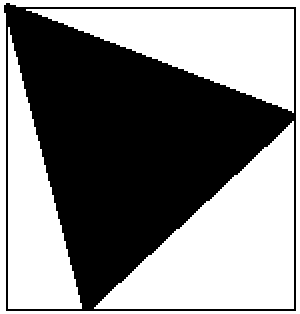
- Consider a 2D triangle with vertices *p0 , p1 , p2*.
- Let *P* be any point in the plane. We can always find a, b, c such that $a, b, c \in \mathbb{R}$

$$P = aP_0 + bP_1 + cP_2 \;\; where \; a + b + c = 1$$

- We will have $a, b, c \in [0,1]$ iff p is inside the triangle.
- We call (a, b, c) the **barycentric coordinates** of *P*.

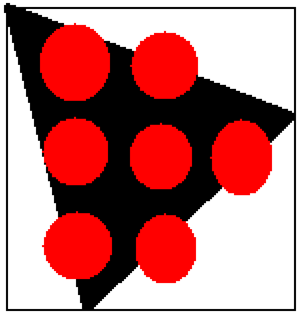
Bounding box of the triangle

- First, identify a rectangular region on the canvas that contains all of the pixels in the triangle (excluding those that lie outside the canvas).
- Calculate a tight bounding box for a triangle: simply calculate pixel coordinates for each vertex, and find the minimum/maximum for each axis



Scanning inside the triangle

- Once we've identified the bounding box, we loop over each pixel in the box.
- For each pixel, we first compute the corresponding (x, y) coordinates in the canonical view volume
- Next we convert these into barycentric coordinates for the triangle being drawn.
- Only if the barycentric coordinates are within the range of [0,1], we plot it (and compute the depth)



Why is z-buffering so popular ?

Advantage

- Simple to implement in hardware.
    - Memory for z-buffer is now not expensive
- Diversity of primitives – not just polygons.
- Unlimited scene complexity
- Don’t need to calculate object-object intersections.

Disadvantage

- Extra memory and bandwidth
- Waste time drawing hidden objects

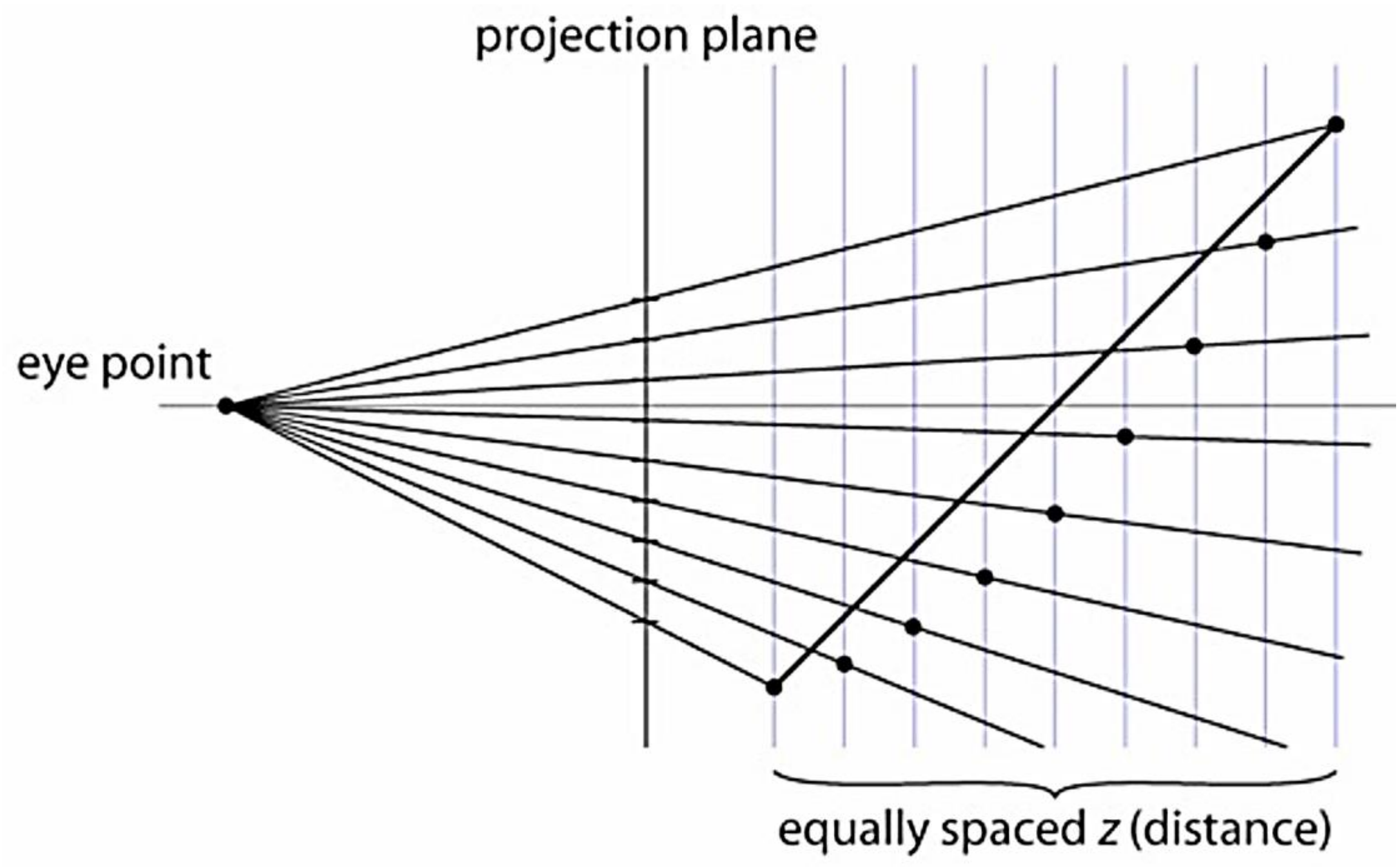
Z-precision errors

- May have to use point sampling

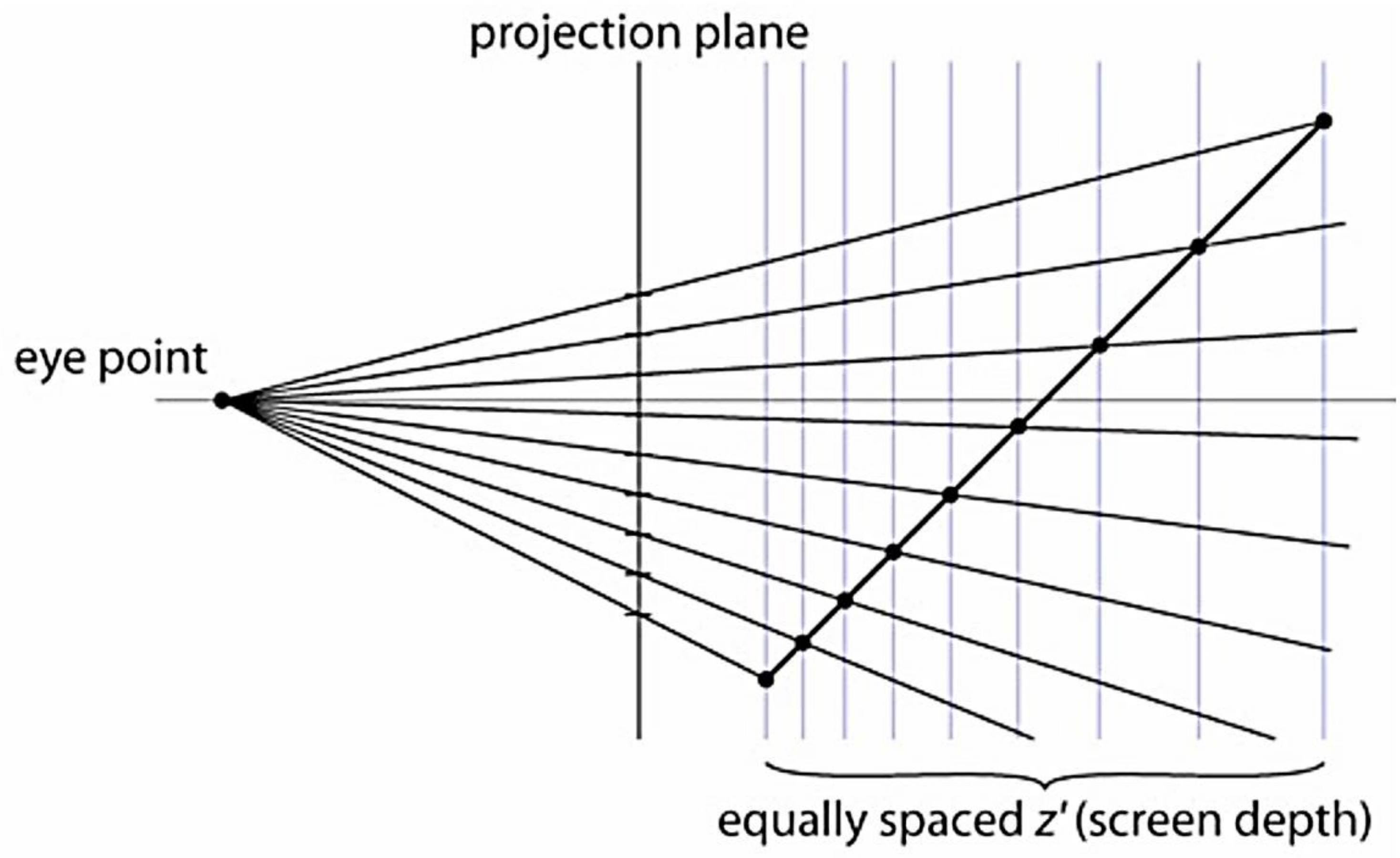
Z-buffer precision

- The precision is distributed between the near and far clipping planes
  - this is why these planes have to exist
  - also why you can't always just set them to very small and very large distances
- Generally use z' (not world z) in z buffer

Depth Interpolating



Depth Interpolating

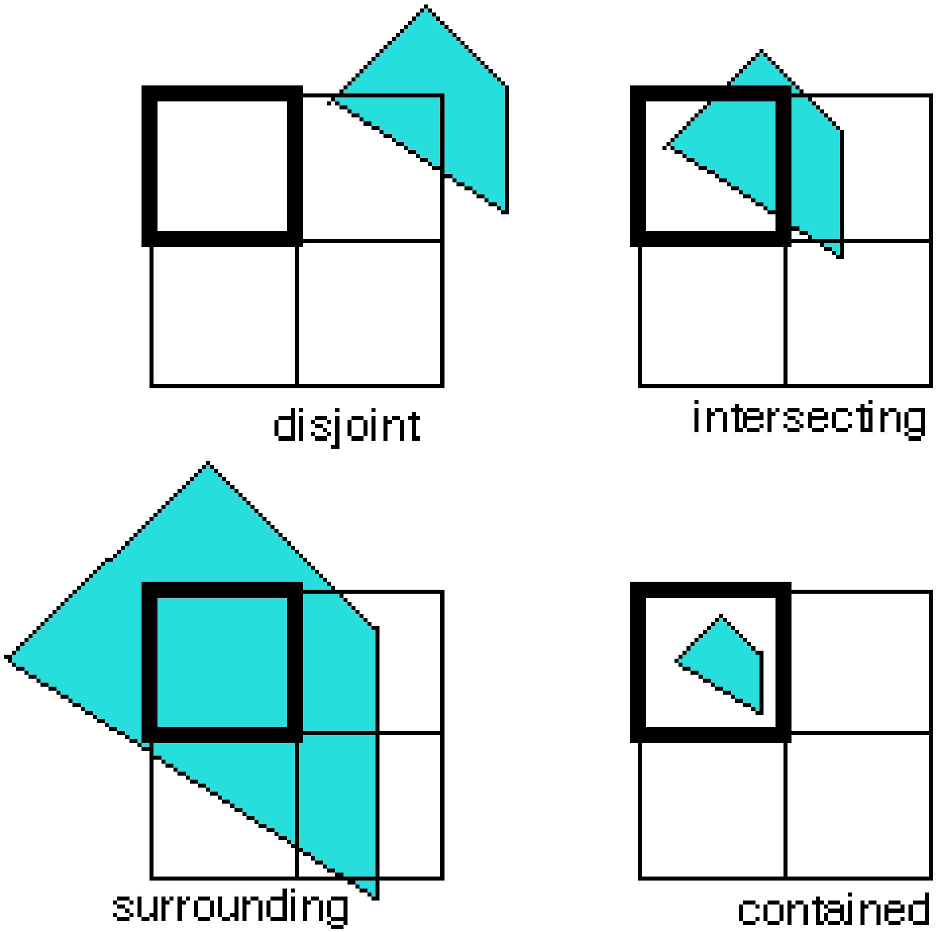


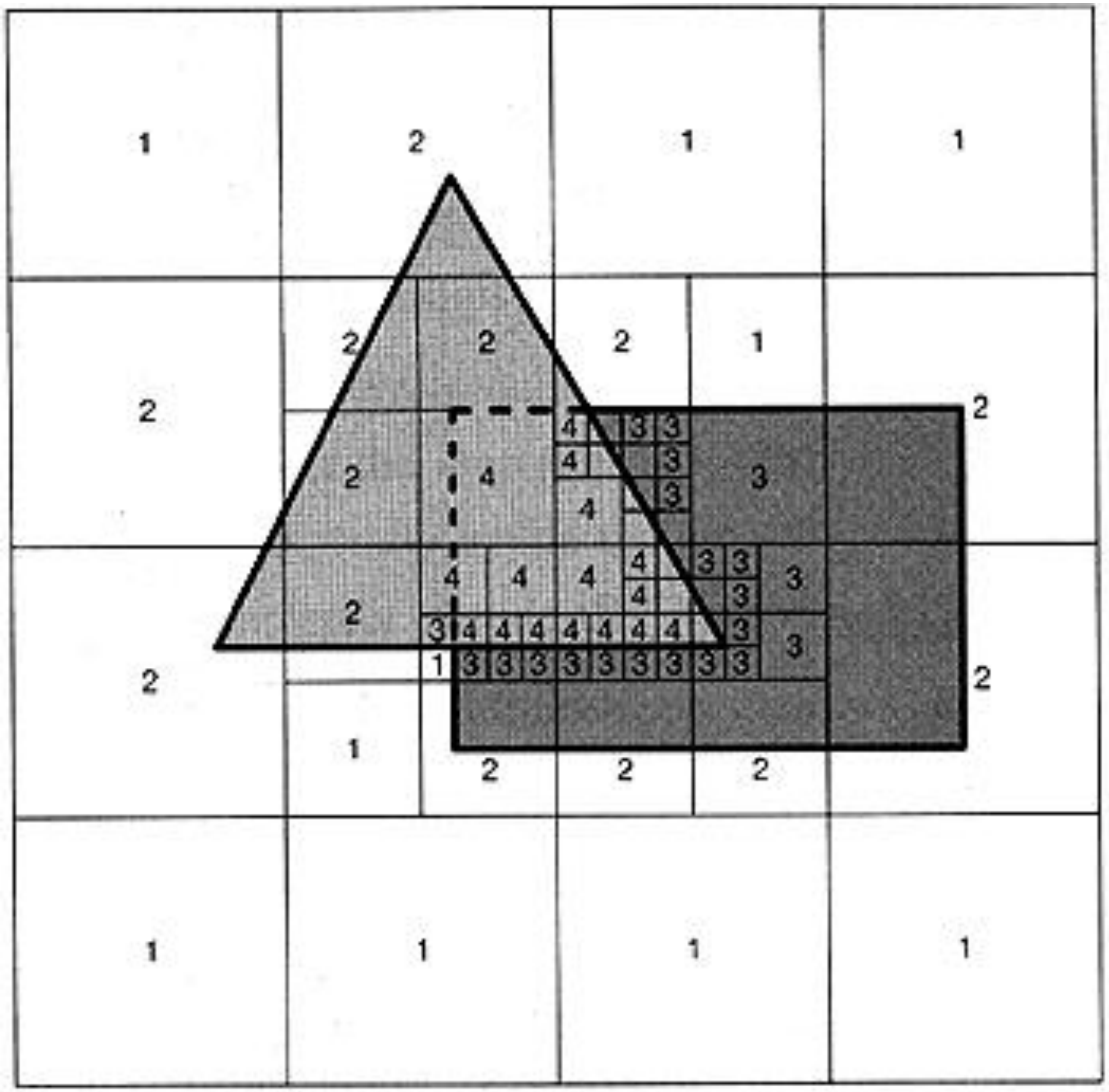
Z-buffer precision

- Caused by the perspective division z/w
- You need to do whatever you can to push the *zNear* clipping plane out and pull the *zFar* plane in as much as possible.
- https://www.opengl.org/archives/resources/faq/technical/depthbuffer.htm

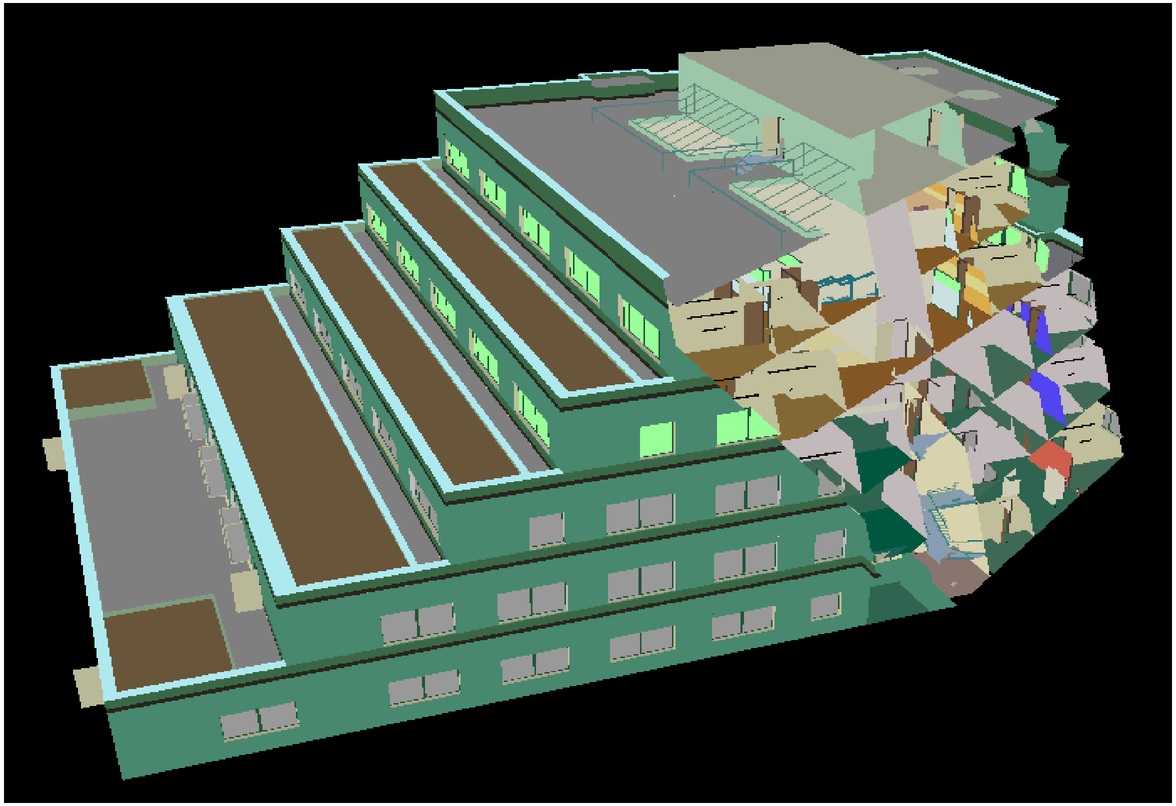
Area Subdivision (Warnock's Algorithm)

- The area is subdivided into smaller parts and the algorithm reoccurs.
- Eventually an area will be represented by a single non-intersecting polygon.





Ex. Architectural scenes



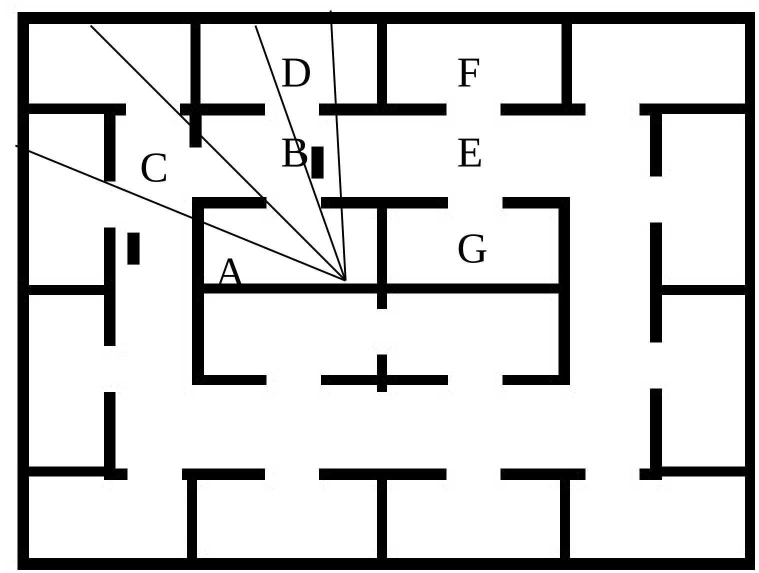
Occlusion at various levels

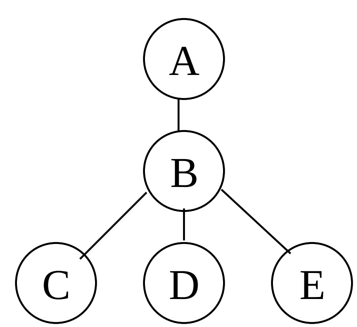
Portal Culling (object-space)

Model scene as a graph:
- Nodes: Cells (or rooms)
- Edges: Portals (or doors)

Graph gives us:
- Potentially visible set

1. Render the room
2. If portal to the next room is visible, render the connected room in the portal region
3. Repeat the process along the scene graph





Summary

- Z-buffer is easy to implement on hardware and is an important tool
- We need to combine it with an object-based method especially when there are too many polygons

References

- Ed Angel, CS/EECE 433 Computer Graphics, University of New Mexico
- Steve Marschner, CS4620/5620 Computer Graphics, Cornell
- Tom Thorne, COMPUTER GRAPHICS, The University of Edinburgh
- Elif Tosun, Computer Graphics, The University of New York
- Lin Zhang, Computer Graphics, Tongji Unviersity

- Questions?