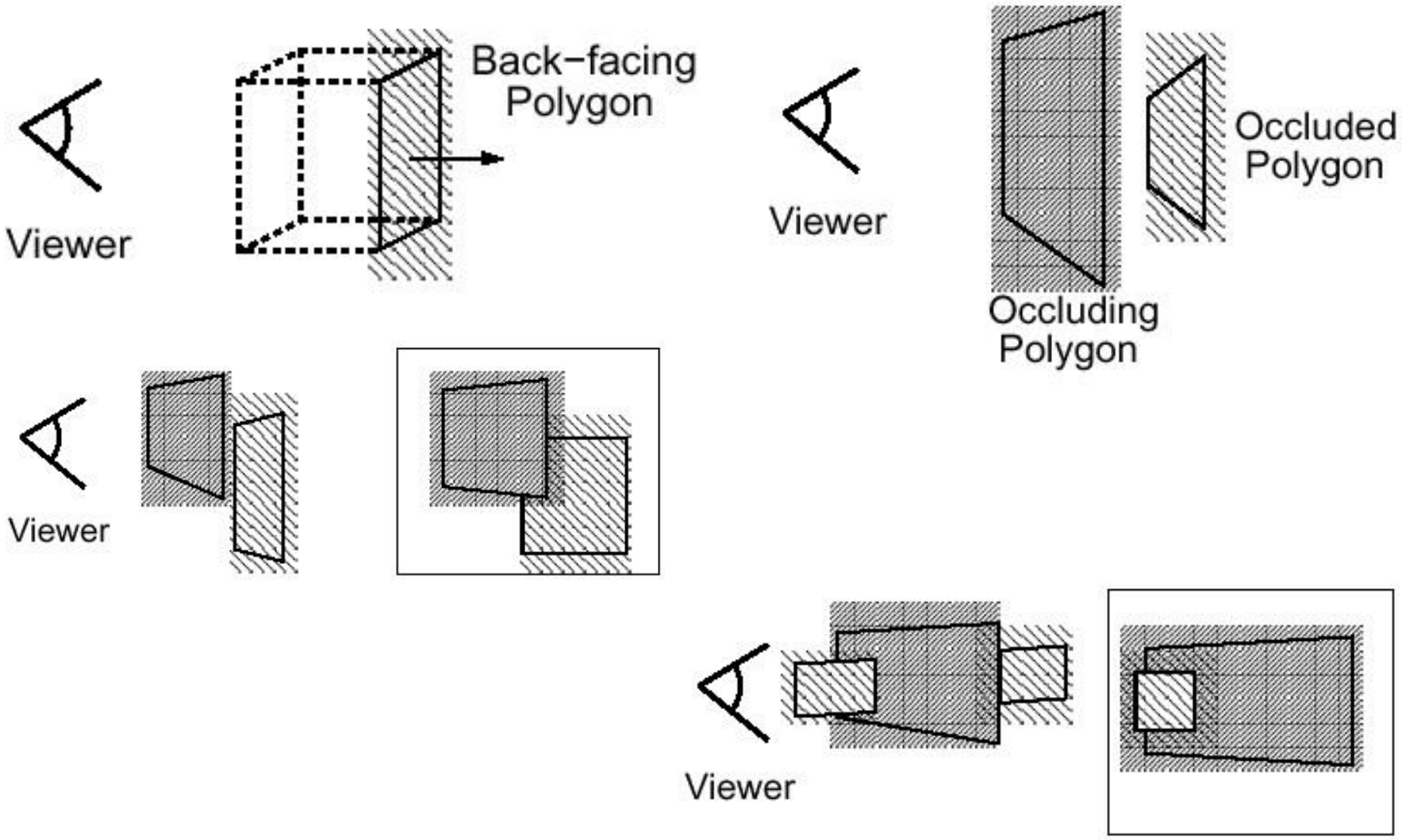
Hidden Line / Hidden Surface Removal

Concepts

- HL/HSR is the process of determining which lines or surfaces of objects are visible, either from the center of projection for perspective projections or along the direction of projection for parallel projections
- Allows only the visible lines or surfaces to be displayed

Difficulties

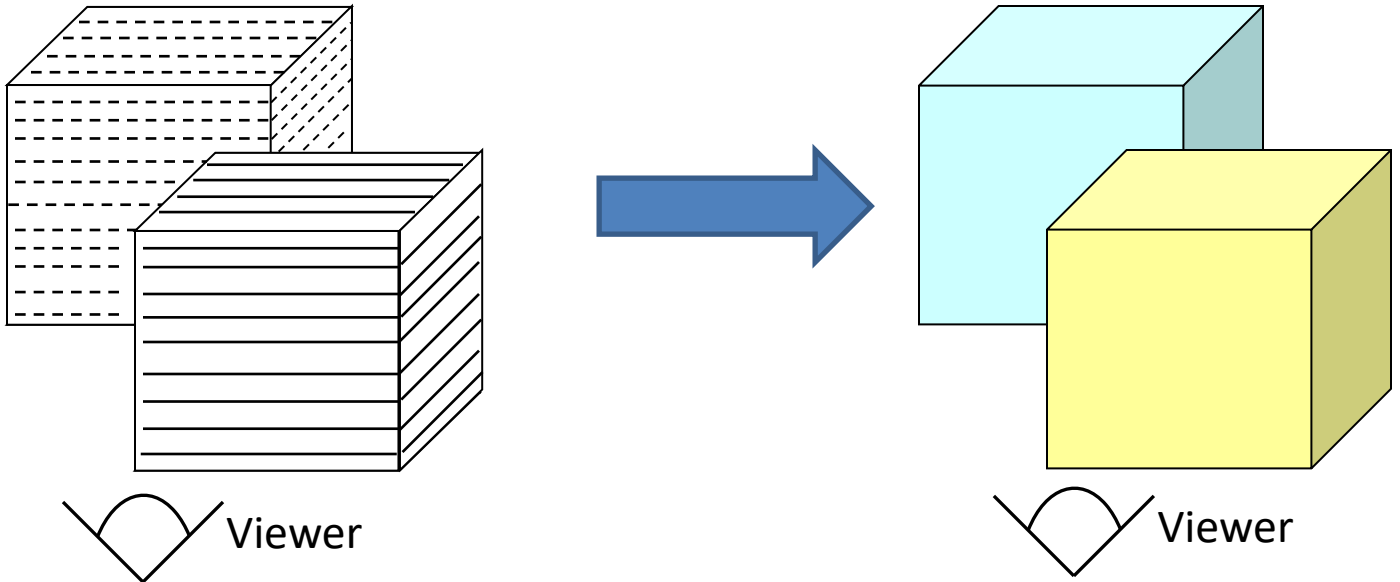


Approaches

- **Image-precision** is performed at the resolution of the device and determines the visibility of each pixel; it is performed AFTER rasterization.
- **Object-precision** is performed in NPC space (after viewing); it is followed by operations to map to device coordinates and physically render the picture.

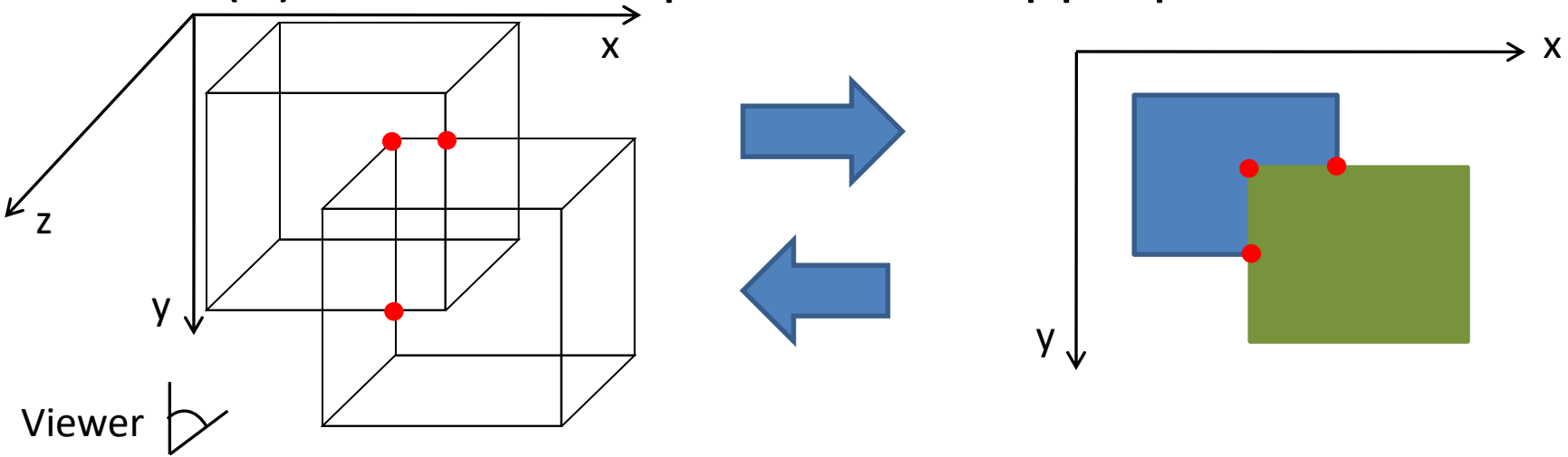
Image-precision methods

- These methods are straight forward, but brute force by examining every pixel in each object.
  - (a) determines the object closest to the viewer that is pierced by the projector through the pixel, and
  - (b) draws the pixel in the appropriate color



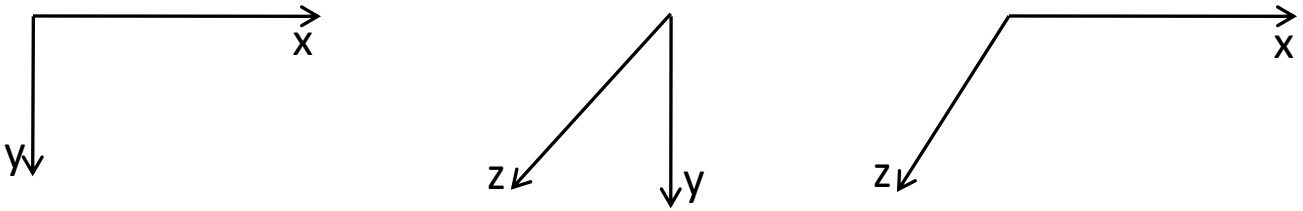
Object-precision methods

- Eliminating entire objects or portions of them that are not visible by
  - (a) determines those parts of the object whose view is unobstructed by other parts of it or any other object
  - (b) draws those parts in the appropriate color



Thinking about solutions

- ...determining whether or not a projector and an object intersect and where they intersect



- ...determining the intersection for multiple projections/objects
- ...for intersections, determining which object is closest to the viewer and therefore visible
- …determining whether or not one planar polygon is visible from the angle of viewer.

HL/HSR algorithms

- HL/HSR algorithms can be optimized using techniques of
  - …coherence
  - ...depth comparisons
  - ...trivial rejection: *pruning*
  - ...backface culling
- Common HL/HSR algorithms include:
  - Backface culling, depth-sorting, Back-face detection, Painter's algorithm, Ray casting, Z-buffering, Scan-line, area subdivision, etc.

Some Results

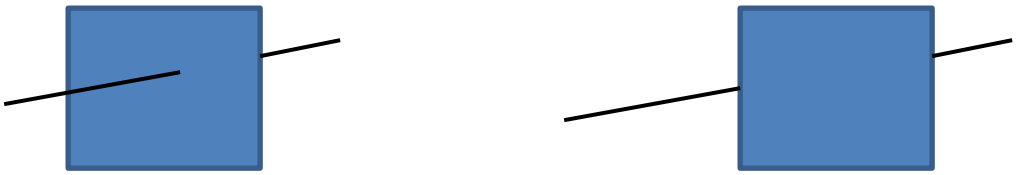
No hidden surface removal

Backface Culling only

Z-Buffering Only

Optimized techniques

- Coherence includes
  - Edge coherence: an edge may change visibility only where it crosses behind a visible edge or penetrates a face.



  - Depth coherence: adjacent parts of the same surface are typically close in depth; once the depth is calculated, the depth of the rest of the surface can be calculated using simple difference equations.

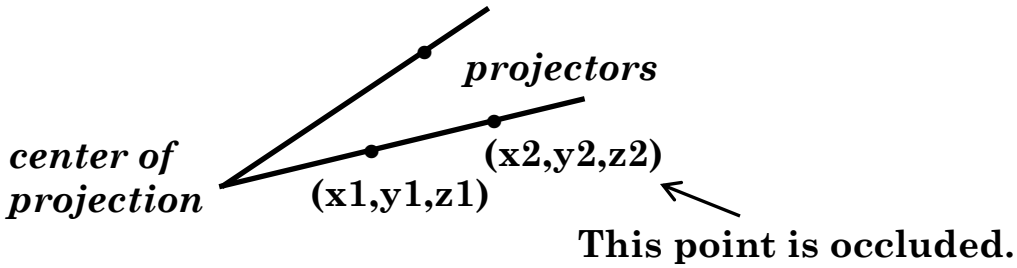
Optimized techniques

- Object coherence: If two objects are completely separate from one another, comparisons only need to be done between the faces of two objects.

- Face coherence: Existent intersection of two planar faces is a segment line and segment line is always visible if one of faces is visible.

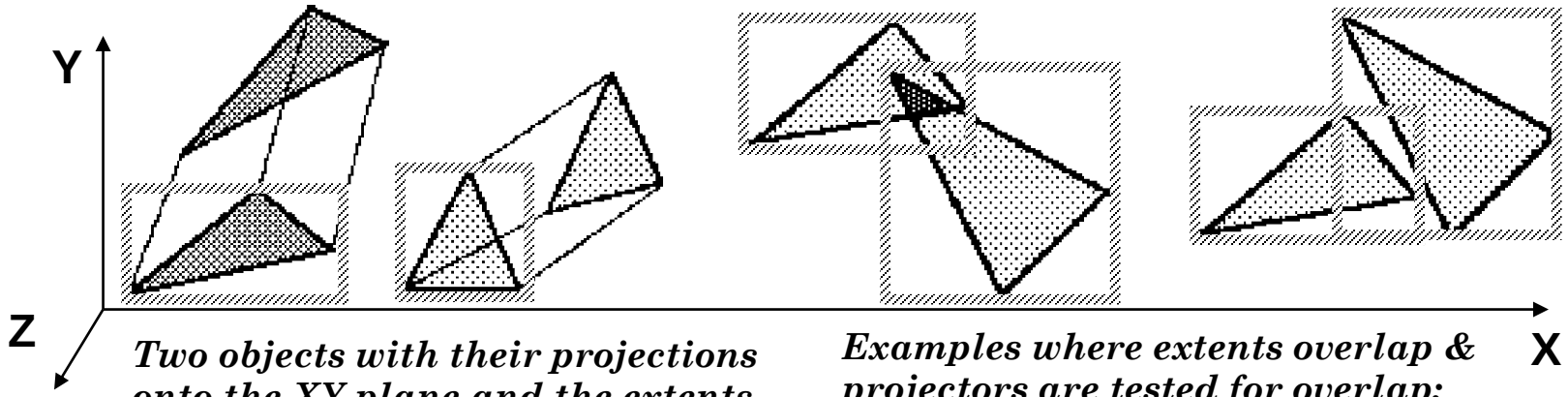
Optimized techniques

- Depth comparisons
  - determine if two or more points obscure one another *(i.e., if two or more points lie on the same projector)*
  - determine which point is closer when points have the same projector



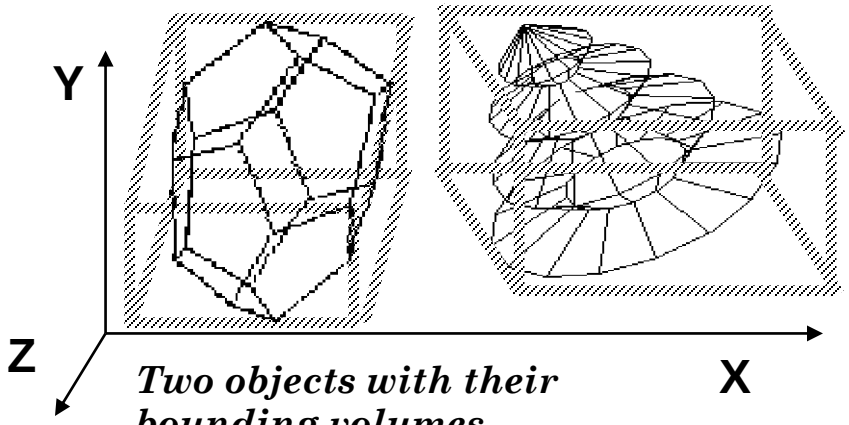
Optimized techniques

- Trivial rejection

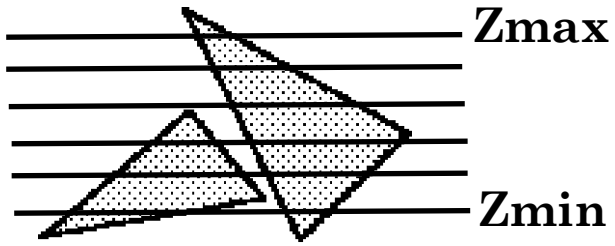


Two objects with their projections onto the XY plane and the extents surrounding the projections

Examples where extents overlap & projectors are tested for overlap; remember there is no guarantee!

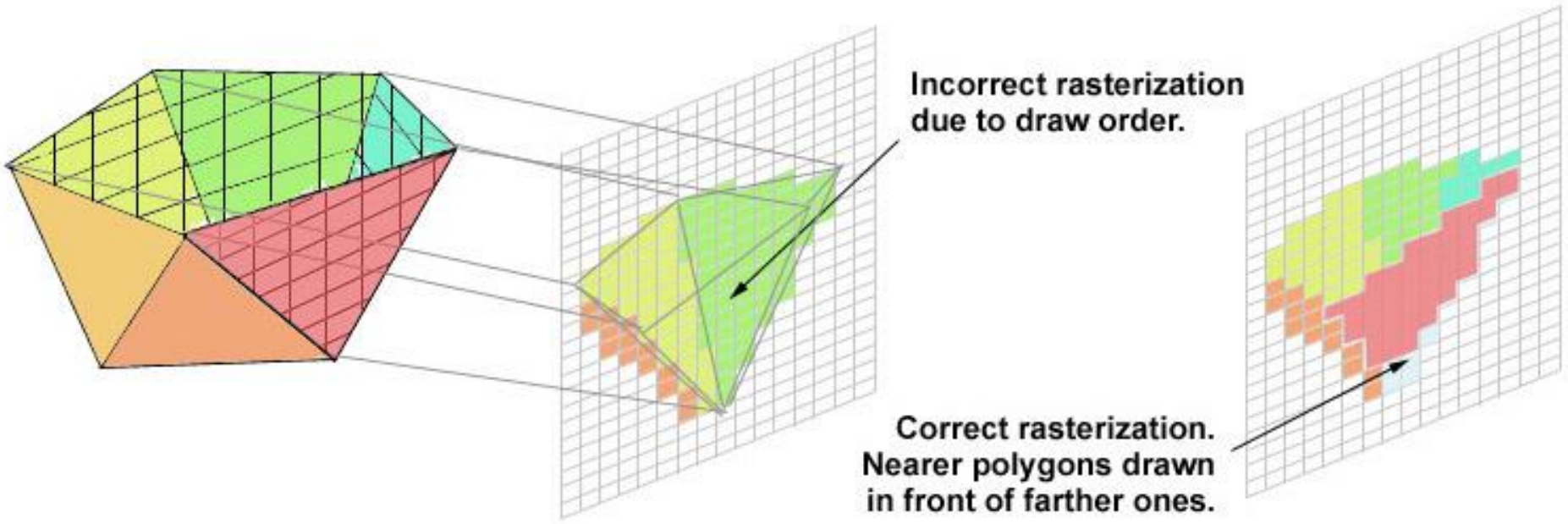


Two objects with their bounding volumes



Using one dimensional extends to determine if objects overlap

Counter example



Given a collection of polygons:
What order do we draw them in?

Simple Z-buffering

- Simple to include in scanline algorithm.
- Interpolate z during scan conversion.
- Maintain a depth (range) image in the frame buffer (16 or 24 bits common).
- When drawing, we can compare with the currently stored z value.
- Pixel given intensity of nearest polygon.

Implementation

- Initialise frame buffer to background colour.
- Initialise depth buffer to z = max.value for far clipping plane.
- Need to calculate value of z for each pixel
  - But only for polygons intersecting that pixel.
  - Could interpolate from values at vertices.
- Update both Output image and depth buffer.

Determining Depth

Use plane equation:

$$Ax + By + Cz + D = 0$$

$$z = \frac{-D - Ax - By}{C}$$

If at (x, y), z value evaluates to z_1,

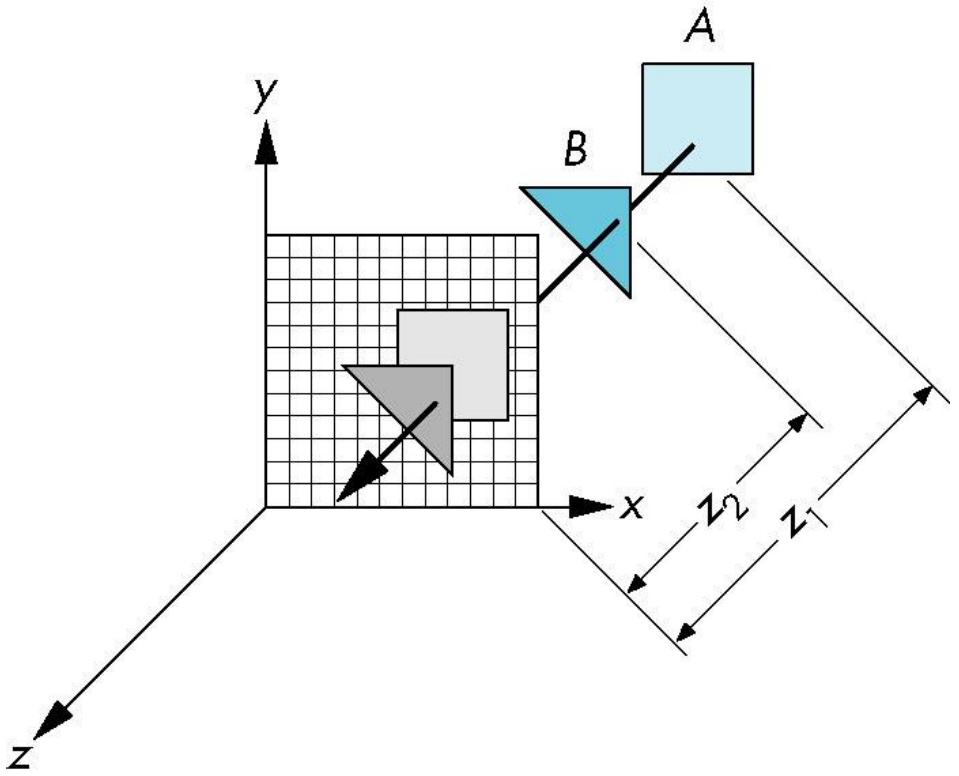
at $(x + \Delta x, y)$, now value of z is:

$$z_1 - \frac{A}{C}(\Delta x)$$

-Only one subtraction needed
-Depth coherence.

Z-Buffer Algorithm

- As we render each polygon, compare the depth of each pixel to depth in z buffer
- If less, place shade of pixel in color buffer and update z buffer



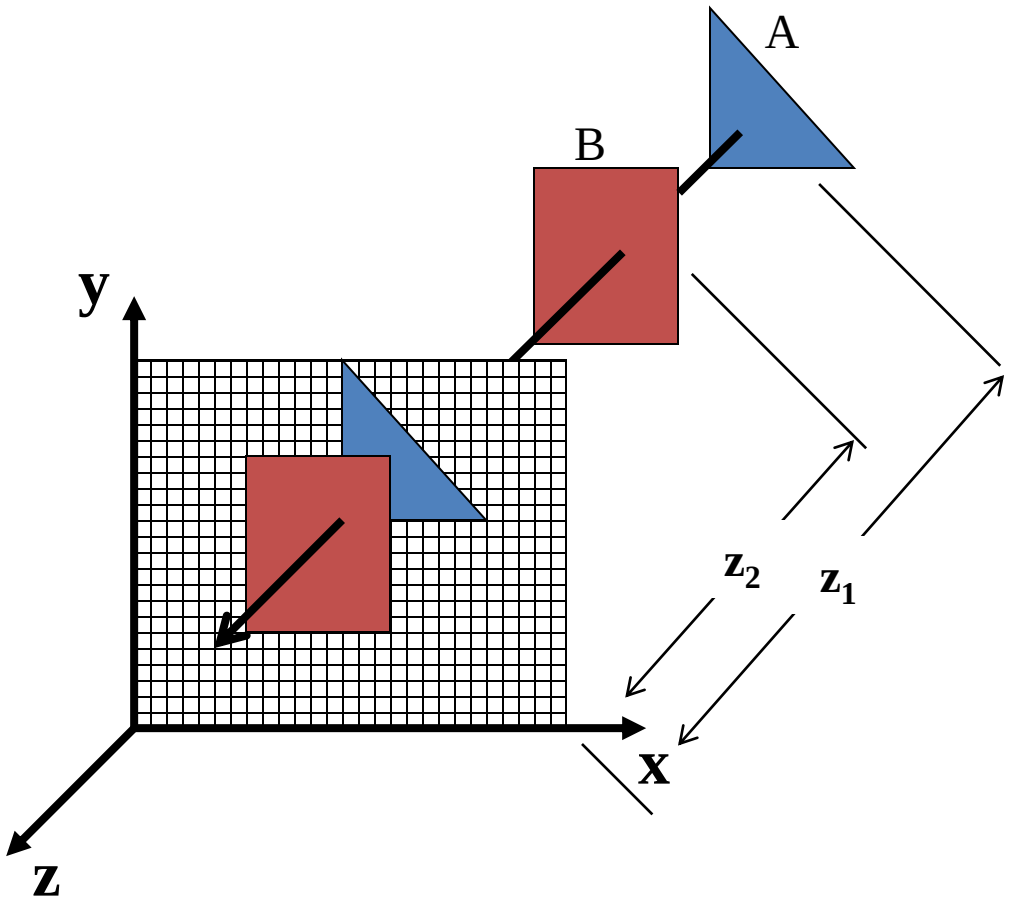
The Z-buffer Algorithm

1. initialize all *depth(x,y)* to 0 and *OutputImage(x,y)* to background colour

2. for each pixel
   1. Get current value depth(x,y)
   2. Evaluate depth value z

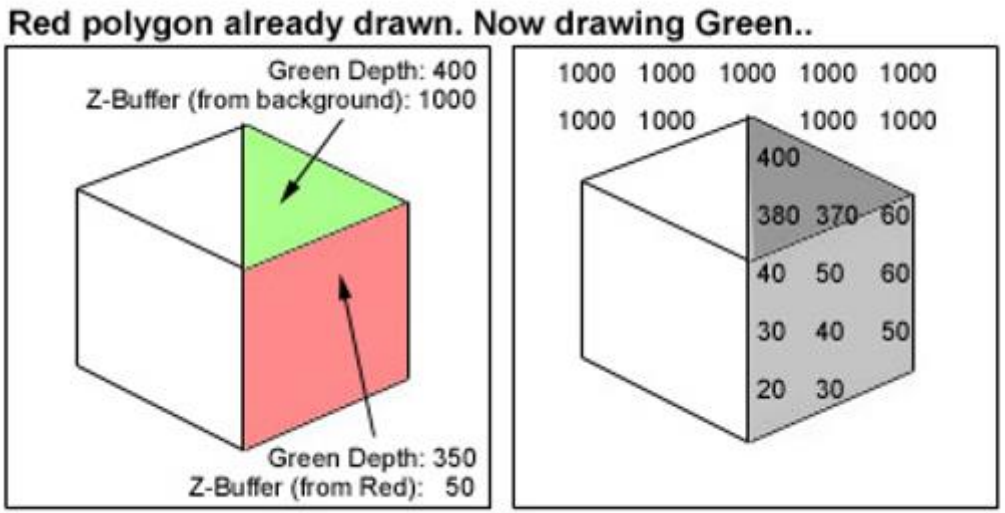
```
if z > depth(x,y)
then
{
    depth(x,y) = z
    OutputImage(x,y) = Is (x,y)
}
```

Calculate this using shading algorithm/illumination/fill color/texture



- After calculation, the depth buffer will contain depth value for the visible surface and the refresh buffer will contain the intensity values for those surfaces .

Z-Buffer Algorithm-Example



For *each pixel in polygon...*

```
If  polygon pixel depth < image pixel depth Then
    Polygon pixel is closer than any previous pixels.
    Draw the pixel
    Set image pixel depth = polygon depth
Else
    Polygon pixel is farther than another pixel.
    Don't draw the pixel.
    Do not modiy image pixel depth (Z-buffer)
End if
```

Why is z-buffering so popular ?

Advantage

- Simple to implement in hardware.
  - Add additional z interpolator for each primitive.
  - 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

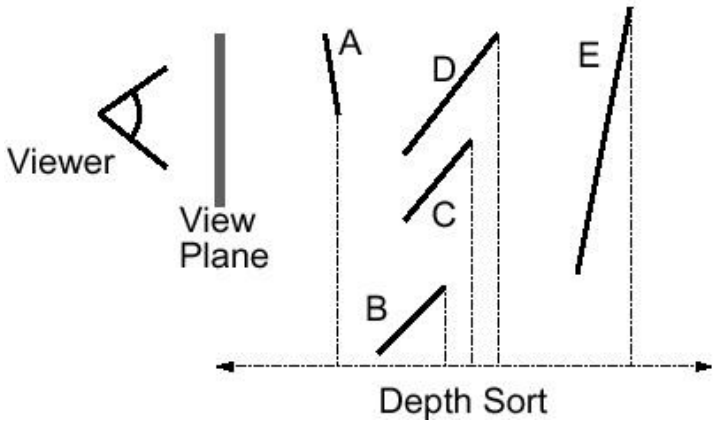
- Extra memory and bandwidth
- Waste time drawing hidden objects
- Z-precision errors
- May have to use point sampling

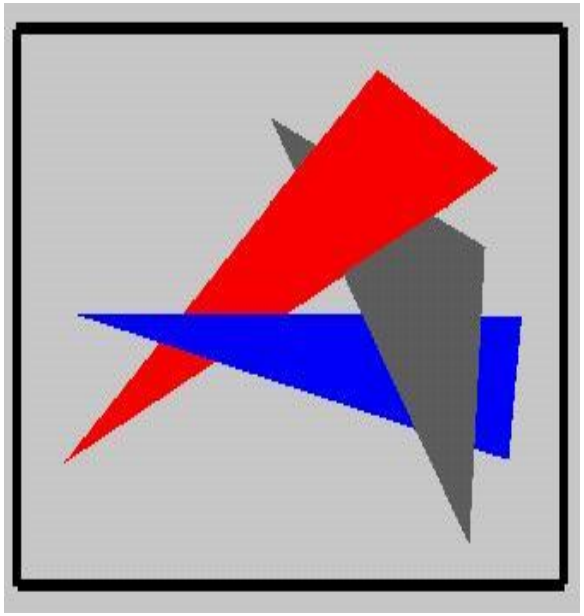
Z-buffer Performance

- Brute-force image-space algorithm scores best for complex scenes – not very accurate but is easy to implement and is very general.
- Working set size: O(1)
- Storage overhead: O(1)
- Time to resolve visibility to screen precision: O(n)
- Overdraw: maximum
- But even O(n) is now intolerable!

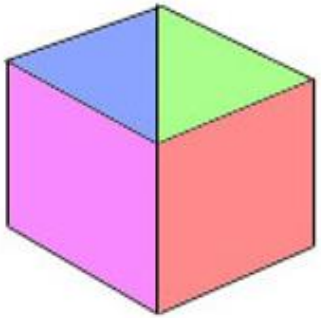
Painters Algorithm (object space)

- Draw surfaces in back to front order – nearer polygons “paint” over farther ones.
- Supports transparency.
- Key issue is order determination.
- Doesn’t always work – see image at right.

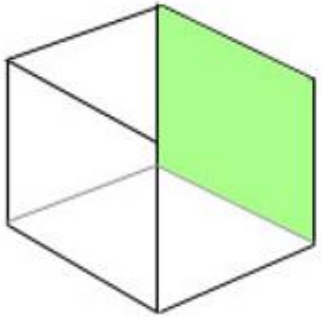




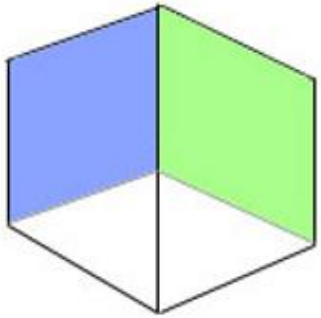
Painters Algorithm- Example



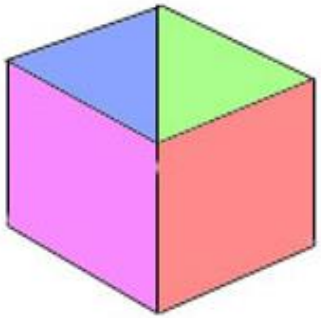
Example: Open Cube.



Paint farthest polygon first.



Next polygon in back-to-front order.

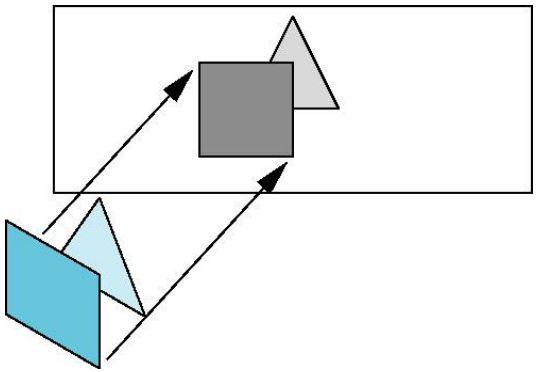


Closest polygons last, overwrite existing ones.

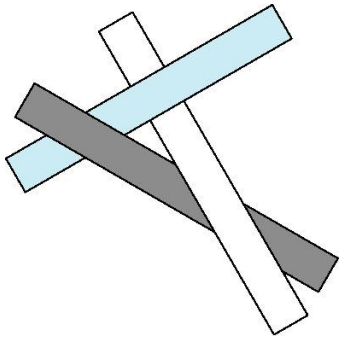
1. Sort polygons from back to front.
2. Draw each sorted polygon, painting over the pixels of existing ones.

- Requires sorting of polygons for each rendering.
- Which point is used for sorting? Center of polygon is typical.

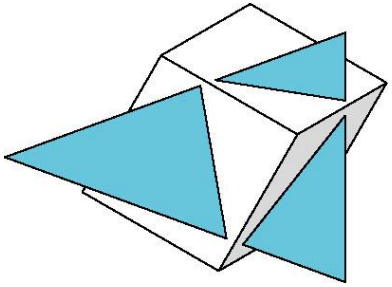
Painters Algorithm- Difficult Case



Easy cases – One polygon is completely behind the other.
Just “paint” over it.



cyclic overlap



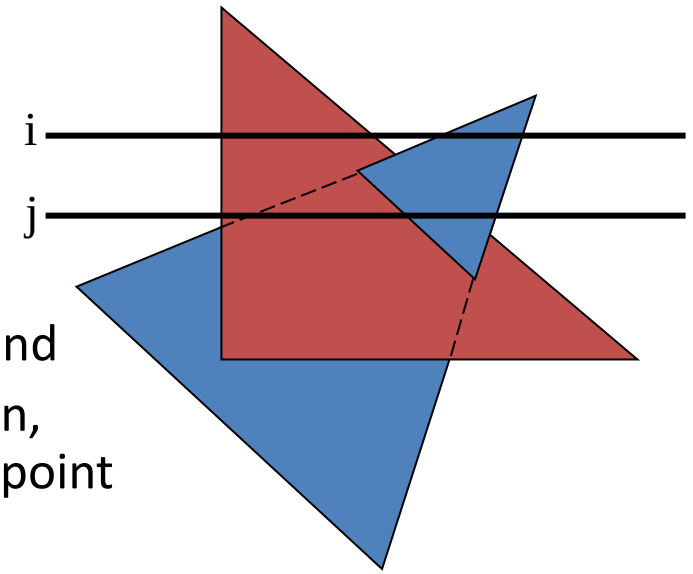
penetration

Scan Line Algorithm

- Similar in some respects to the z-buffer method but handles the image scan-line by scan-line

1. Rasterize all polygon boundaries (edges)

2. Scanning across each scan line we determine the color of each pixel

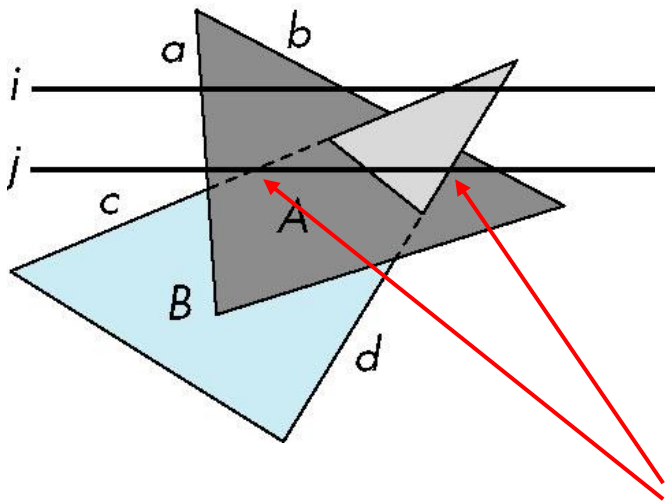


   a. By default color everything as background
   b. if we encounter the edge of one polygon, start evaluating polygon color* at each point and shade the scan line it accordingly
   c. For multiple edges do depth evaluation to see which polygon is the nearest viewer.

- Due to coherency in data, this can be relatively efficient.

Scan-Line Algorithm

- Can combine shading and hsr through scan line algorithm

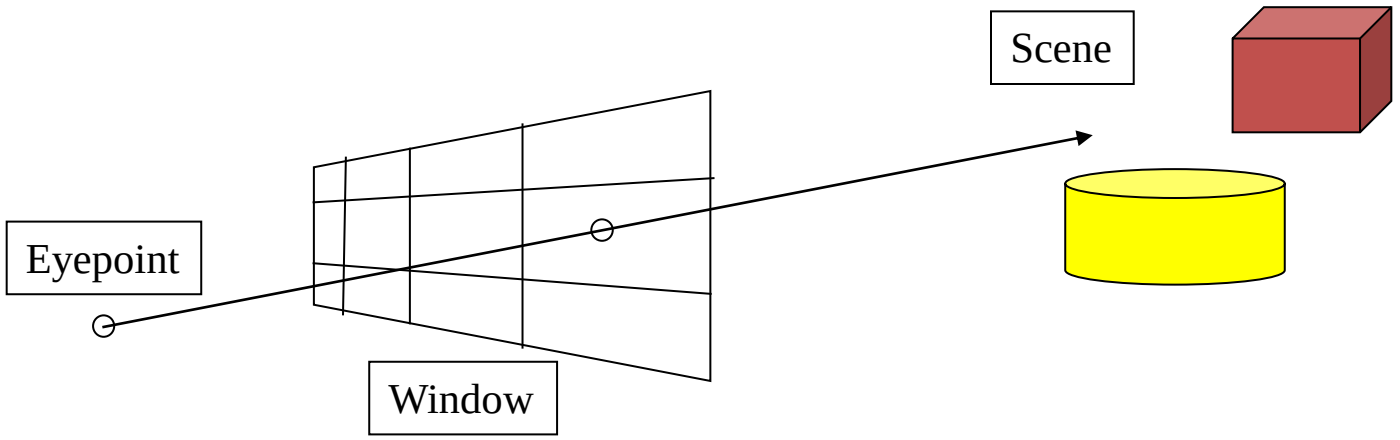


scan line i: no need for depth information, can only be in no or one polygon

scan line j: need depth information only when in more than one polygon

Ray Casting

- Sometimes referred to as *Ray-tracing*.
- Involves projecting an imaginary ray from the centre of projection (the viewers eye) through the centre of each pixel into the scene.



Computing Ray-Object Intersections

- Example: Ray tracing with a sphere.

Express line in parametric form.

$$x = x_0 + t\Delta x \;\; ; \;\; y = y + t\Delta y \;\; ; \;\; z = z + t\Delta z$$

Equation for a sphere:

$$(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2$$

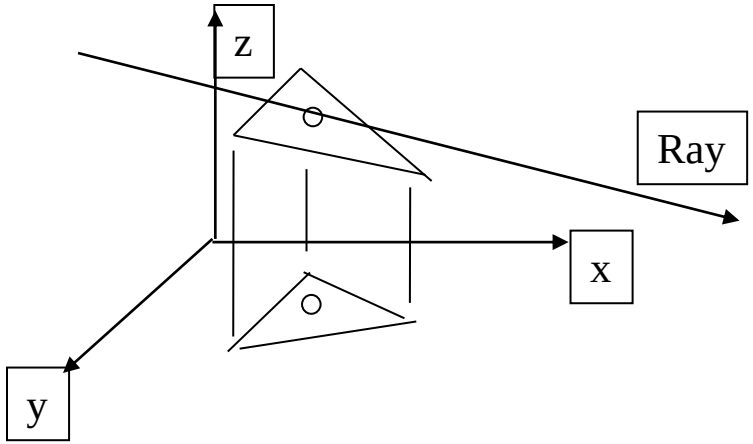
Expand, substitute for *x*, *y* & *z*.
Gather terms in *t*.
$\Rightarrow$ Quadratic equation in t.

Solve for t.

-No roots – ray doesn't intersect.
- 1 root – ray grazes surface.
- 2 roots – ray intersects sphere,
(entry and exit)

Ray-Polygon Intersection

- It needs two steps as following;
  1. Determine whether ray intersects polygon's plane.
  2. Determine whether intersection lies within polygon.
- Easiest to determine (2) with an orthographic projection onto the nearest axis and the 2D point-in-polygon test.

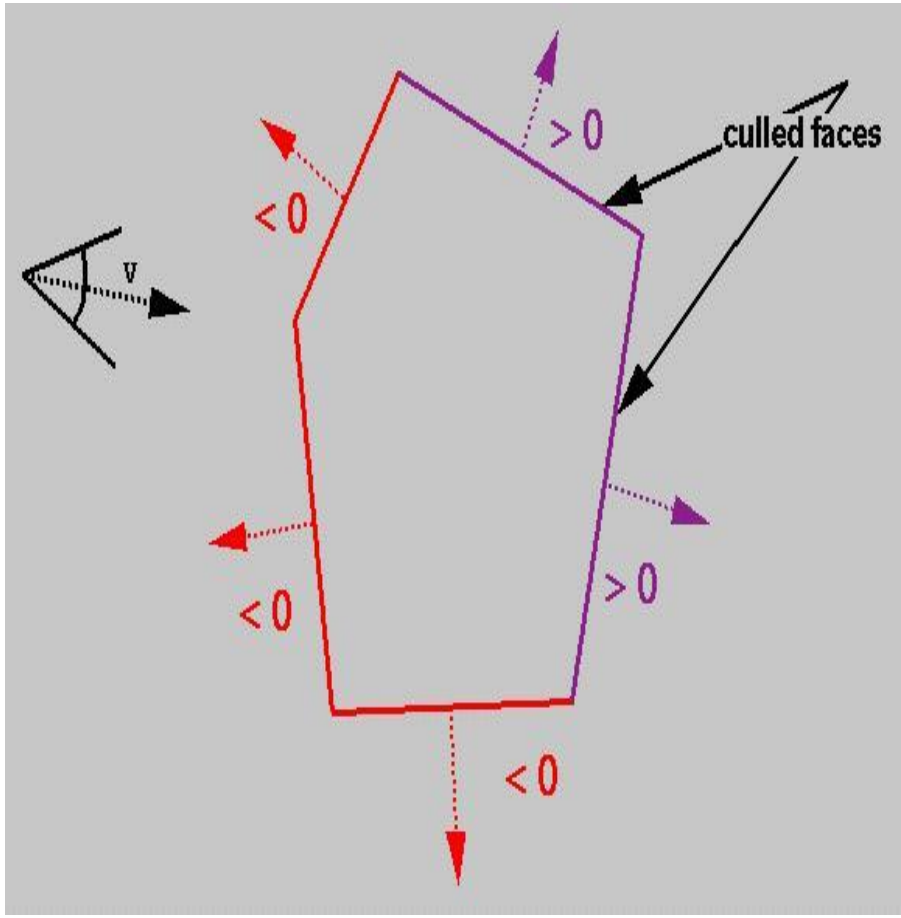


Ray Casting

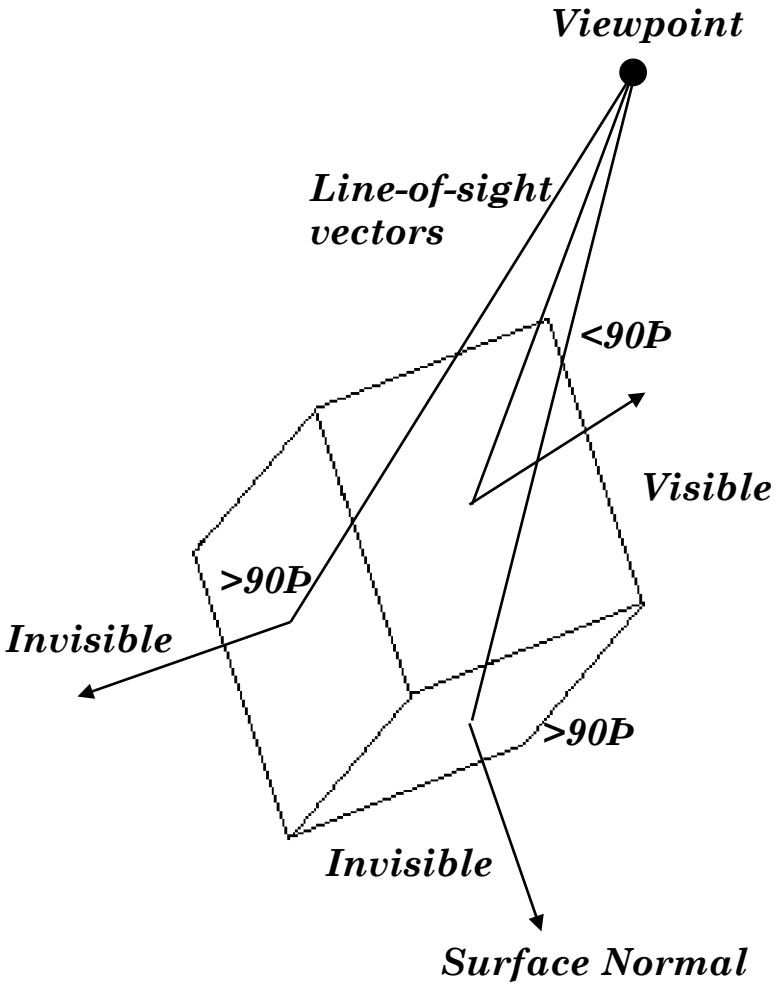
- Easy to implement for a variety of primitives – only need a ray-object intersection function.
- Pixel adopts colour of nearest intersection.
- Can draw curves and surfaces exactly – not just triangles !
- Can generate new rays inside the scene to correctly handle visibility with reflections, refraction etc – *recursive ray-tracing*.
- Can be extended to handle global illumination.
- Can perform area-sampling using ray super-sampling.
- But… expensive for real-time applications.

Back Face Culling

- We saw in modelling, that the vertices of polyhedra are oriented in an anticlockwise manner when viewed from outside – surface normal N points out.
- Project a polygon.
  - Test z component of surface normal. If negative – cull, since normal points away from viewer.
  - Or if N.V > 0 we are viewing the back face so polygon is obscured.
- Only works for convex objects without holes, ie. closed orientable manifolds.

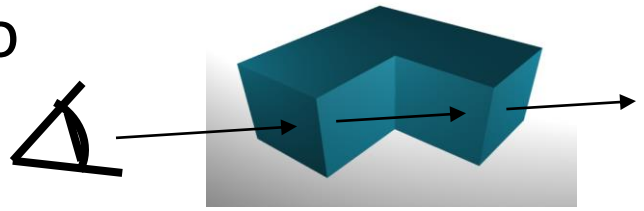


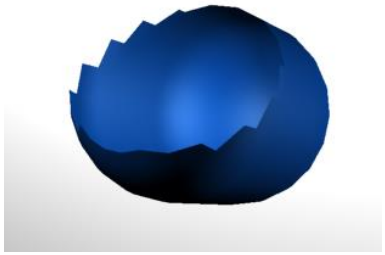
Back Face Culling



Back Face Culling

- Back face culling can be applied anywhere in the pipeline: world or eye coordinates, and image space.
- But it does not work well for:
  - Overlapping front faces due to
    - Multiple objects
    - Concave objects
  - Non-polygonal models
  - Non-closed Objects





Object Culling: 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).

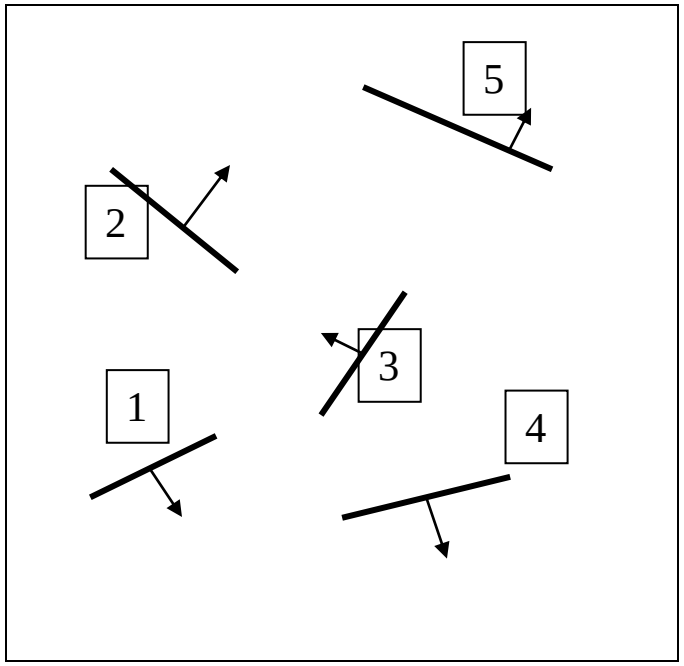
•Choose polygon arbitrarily

•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.



View of scene from above

BSP Tree

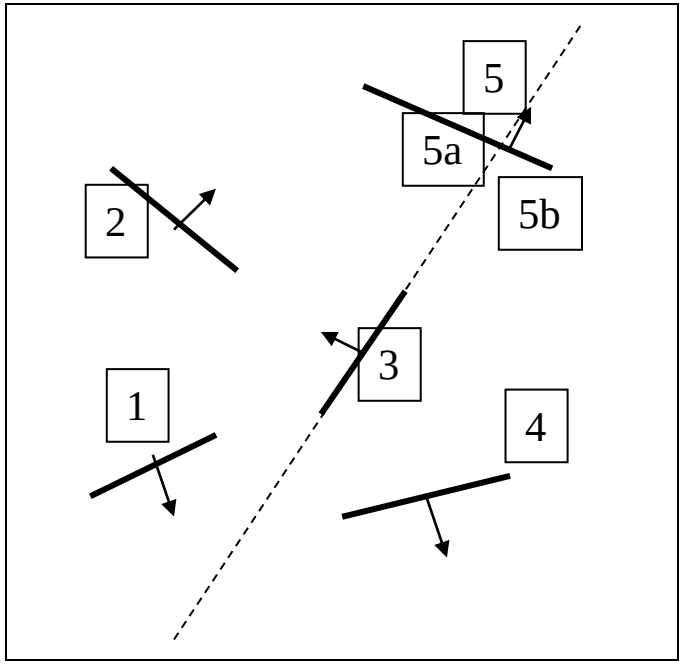
•**Choose polygon arbitrarily**

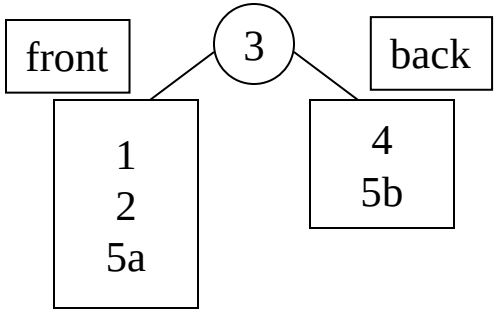
•**Divide scene into front (relative to normal) and back half-spaces.**

•**Split any polygon lying on both sides**.

•Choose a polygon from each side – split scene again.

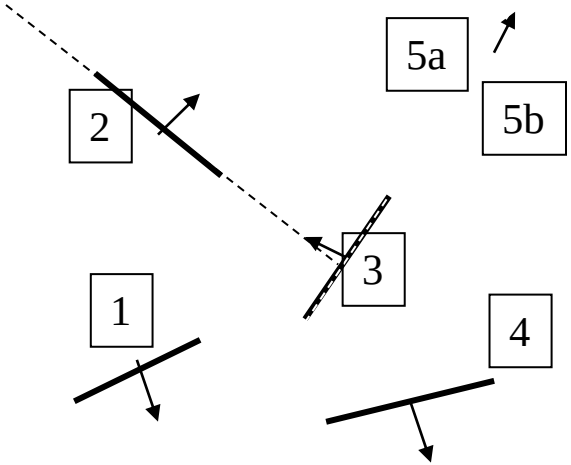
•Recursively divide each side until each node contains only 1 polygon.

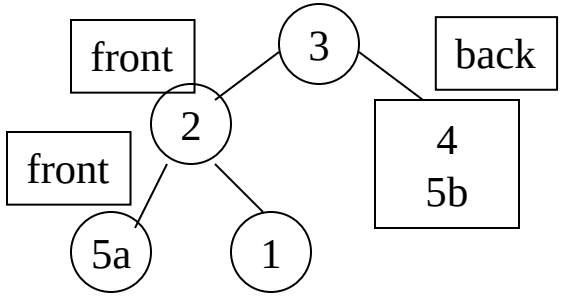




BSP Tree

- 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

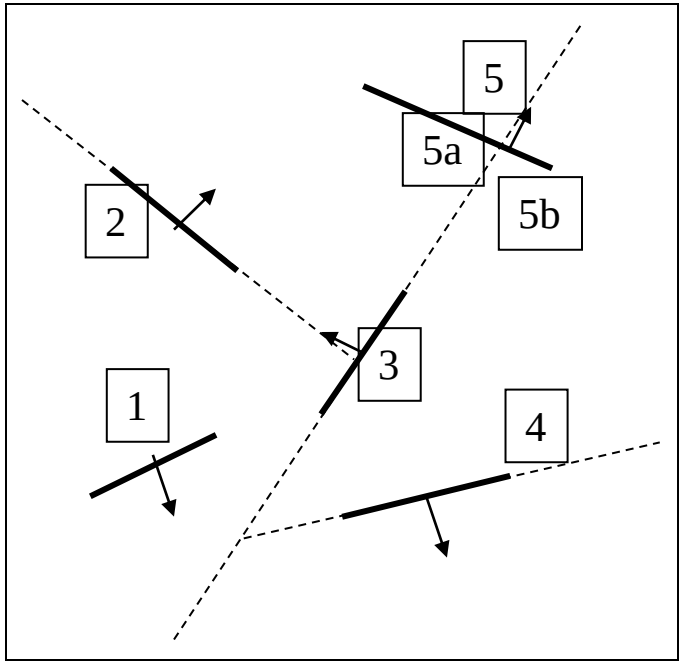
•Choose polygon arbitrarily

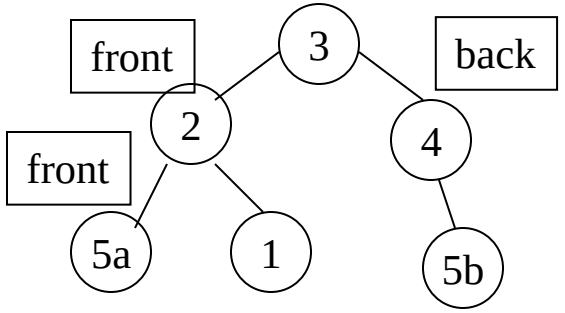
•Divide scene into front (relative to normal) and back half-spaces.

•Split any polygon lying on both sides.

•Choose a polygon from each side – split scene again.

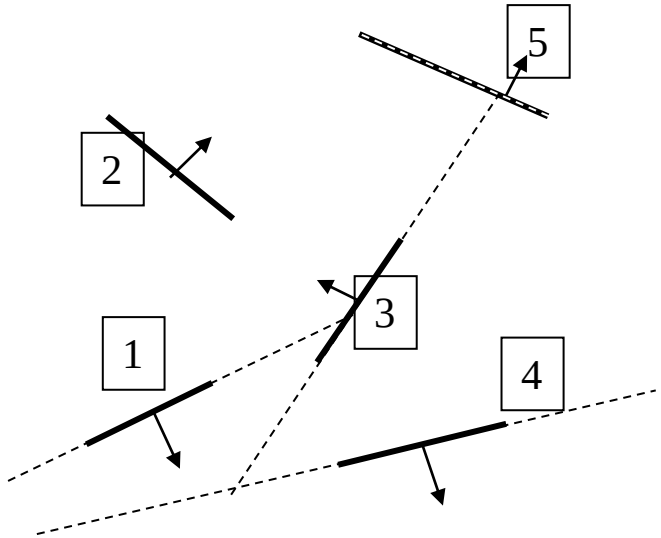
•**Recursively divide each side until each node contains only 1 polygon.**

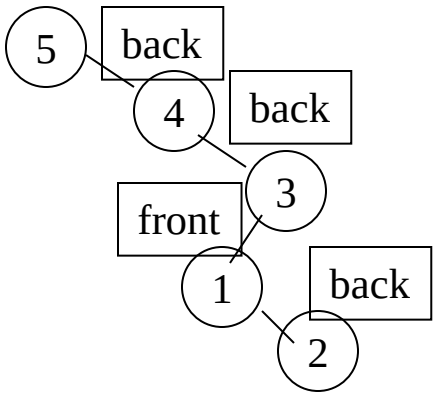




BSP Tree

- 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.





Alternate formulation starting at 5

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 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 Algorithm

```
void BSP_displayTree(BSP_tree* tree)
{
if ( tree is not empty )
if ( viewer is in front of root ) {
BSP_displayTree(tree->backChild);
displayPolygon(tree->root);
BSP_displayTree(tree->frontChild)
}
else {
BSP_displayTree(tree->frontChild);
/* ignore next line if back-face culling desired */
displayPolygon(tree->root);
BSP_displayTree(tree->backChild)
}
}
```

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.

References

[1] Foley, Van Dam, Feiner, Hughes, Computer Graphics - Principles and Practices 2nd Ed. In C, Addison Wesley, 1997.

[2] K. S. Fant, CG-Course Slide, Portland State University.

[3]J. Dingliana, CG-Course Slide, Trinity College Dublin, The University of Dublin.

[4] H. Kiem, D.A. Duc, L.D. Duy, V.H. Quan, Cơ Sở Đồ Họa Máy Tính, NXB. Giáo Dục, 2005.