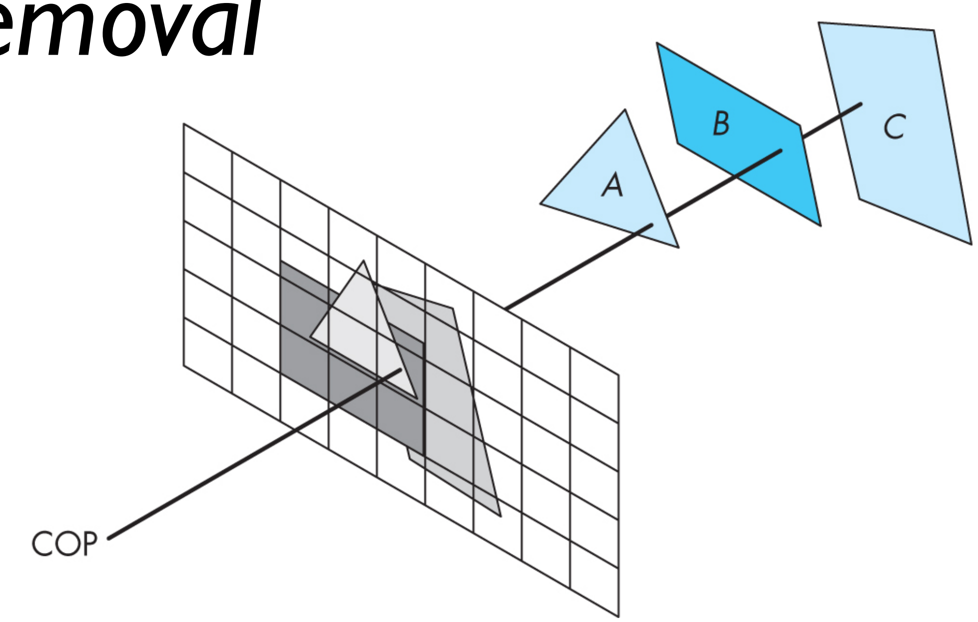
Visibility Determination

Introduction to Computer Graphics
CSE 555 / 333

Visibility Determination

- Determine surface patches that will be visible from a given viewpoint
- Also called *hidden surface removal*
- Three types of algorithms:
  1. Object precision (space)
  2. Image precision (space)
  3. List priority (hybrid object/image precision)



Object Precision Algorithms

foreach *object O* **do**

find the part *A* of *O* that is visible

display *A*

Image Precision Algorithms

foreach *pixel P on the screen* **do**
    Let *R* be the ray from viewer through pixel *P*
    determine the visible object *O* pierced by ray *R*
    if *there is such O* **then**
        display the pixel in the colour of *O*
    else
        display the pixel in the background colour

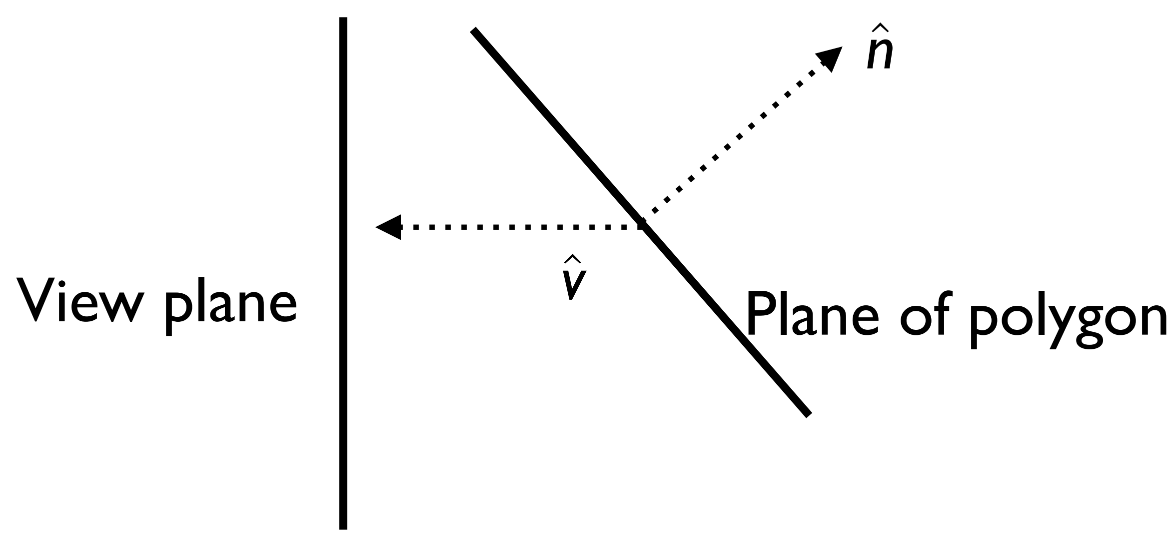
Comparison

- Object precision
  - Computes all visible parts
  - problems with aliasing
  - Complexity is based on the number of objects
- Image precision
  - Determines visibility in samples number of directions
  - Complexity is based on the resolution
- List priority
  - Between image space and object space
  - Most of the algorithms

Back Face Culling

Object Precision

- Discarding back facing polygons from rendering
- Consider counter-clockwise orientation outward



- Discard face if $\hat{v} \cdot \hat{n} < 0$

Ray Casting

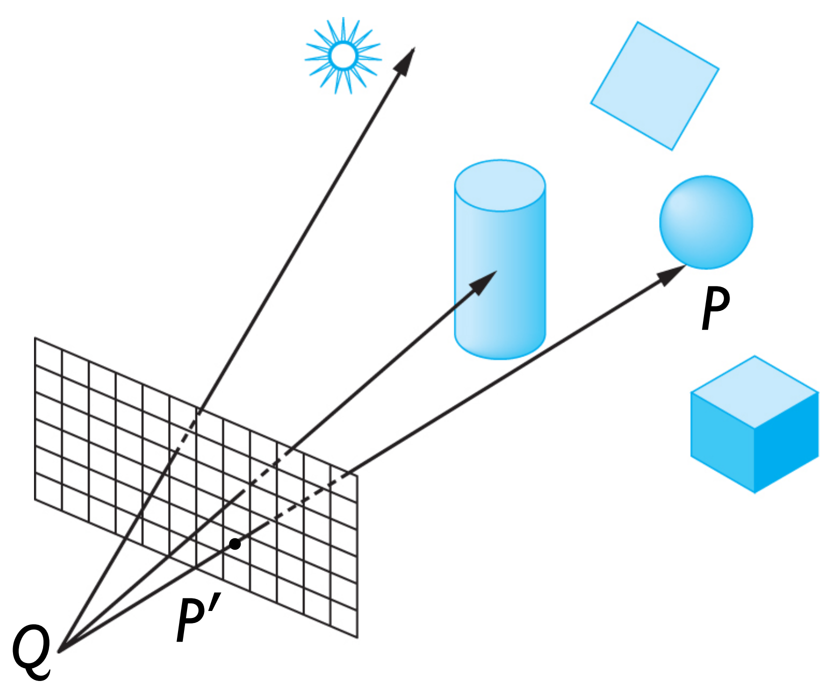
Image Precision

- Computes visibility function:

$$V(P,Q) = \begin{cases} 1 & : \text{if } P \text{ is result of intersection query on ray } R \\ 0 & : \text{otherwise} \end{cases}$$

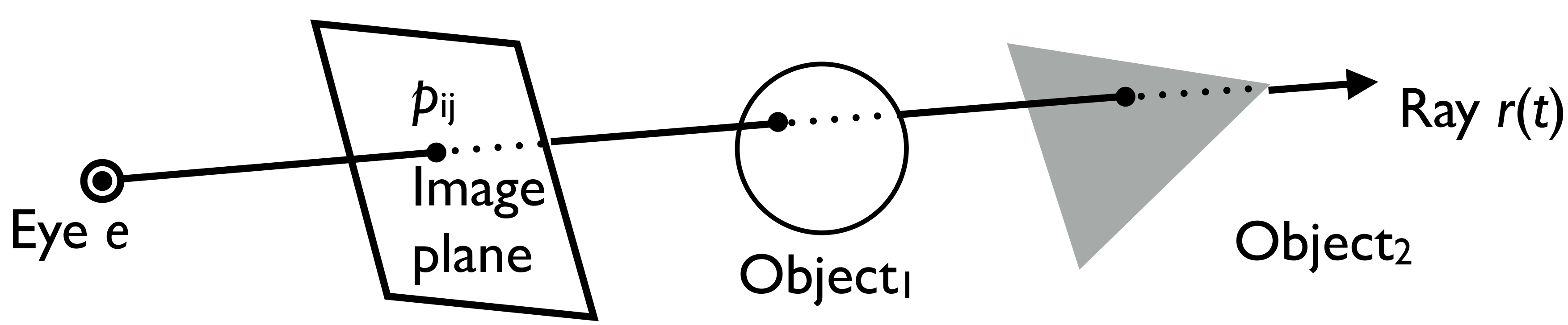
R is a ray with origin Q and direction $(P' - Q)/\|P' - Q\|$.

where p' is pixel center



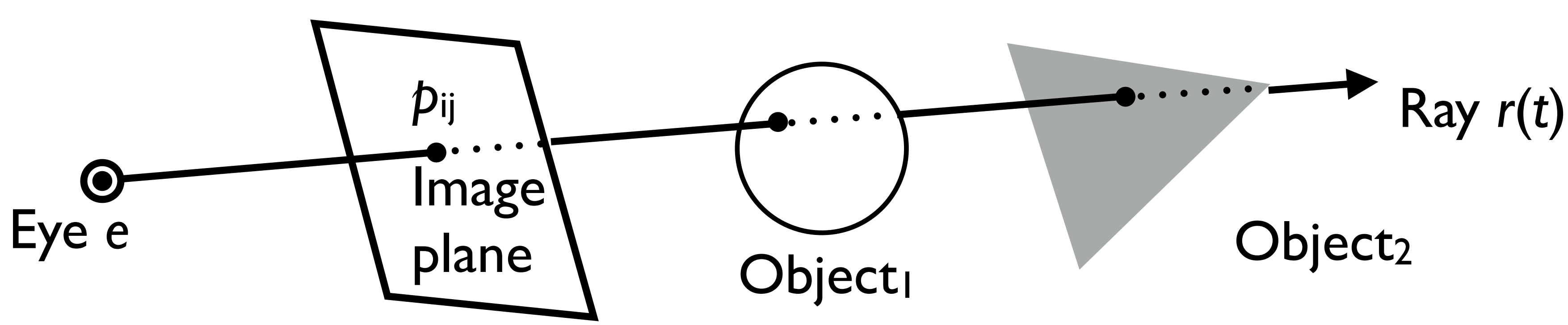
Ray Casting

- Partition the projection plane into pixels
- For each pixel, construct a ray emanating from the eye/camera passing through the center of the pixel and into the scene
- Intersect ray with every object in the scene
- Store the first hit object and color the pixel



Ray Casting

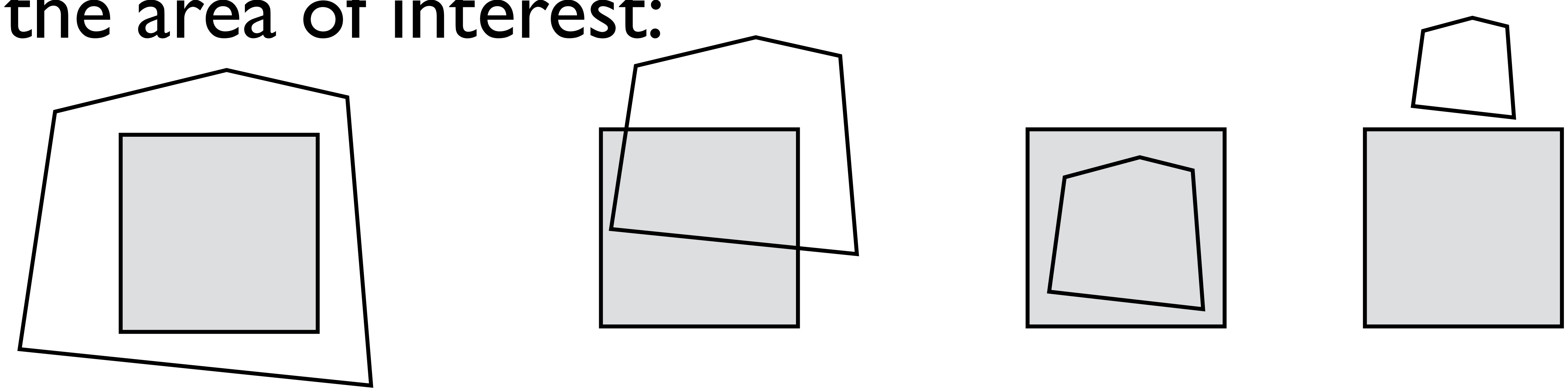
- Store the first hit object and color the pixel
  - Parameterise each ray: $r(t) = e + t(p_{ij} - e)$
  - Each object O_i returns $t_i > 1$ such that first intersection with O_i occurs at $r(t_i)$
  - Choose minimum of all such hit points to shade pixel p_{ij}



Warnock's Algorithm

Image Precision

- Elegant divide-and-conquer hidden surface algorithm
- Relies on area coherence of polygons to resolve visibility of many polygons in image space
- Each polygon has one of the four relationships to the area of interest:



Surrounding Intersecting Contained Disjoint

Warnock’s Algorithm

- Four cases:

1. If all polygons are disjoint from the area, fill area with background colour

2. Only one intersecting or contained polygon
   - First fill with background colour,
   - Then scan convert polygon

3. Only one surrounding polygon, fill area with polygon’s colour

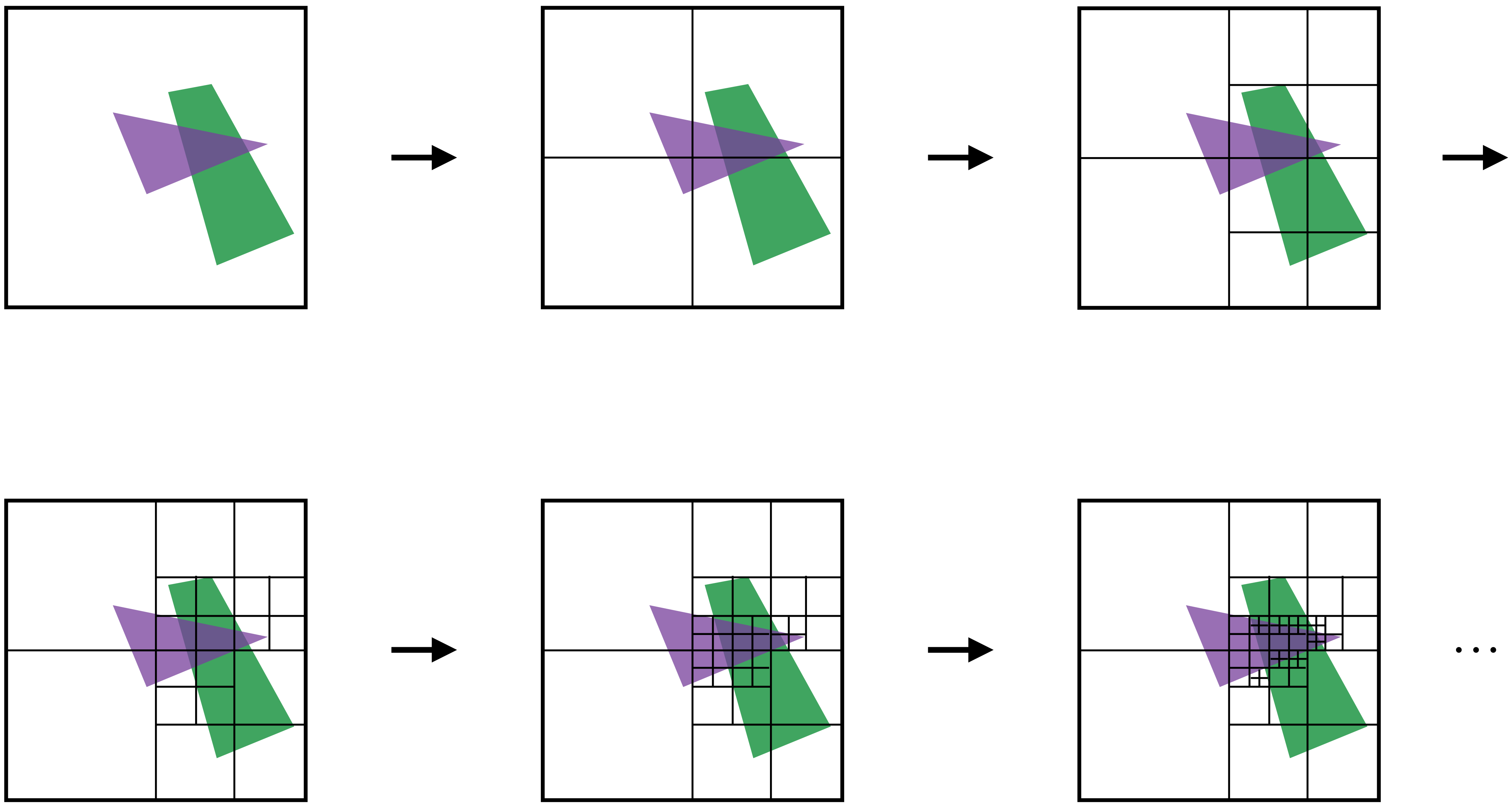
4. More than one polygon are surrounding, intersecting or contained, but one polygon is in front of rest, fill area with polygon’s colour

Warnock's Algorithm

- If none of the above cases occur, subdivide area into four parts, and recurse
- When the resolution of the image is reached, polygons are sorted by their z-values at the centre of the pixel and the color of closest polygon is used

Warnock's Algorithm

Quad-tree decomposition



Z-buffer Algorithm

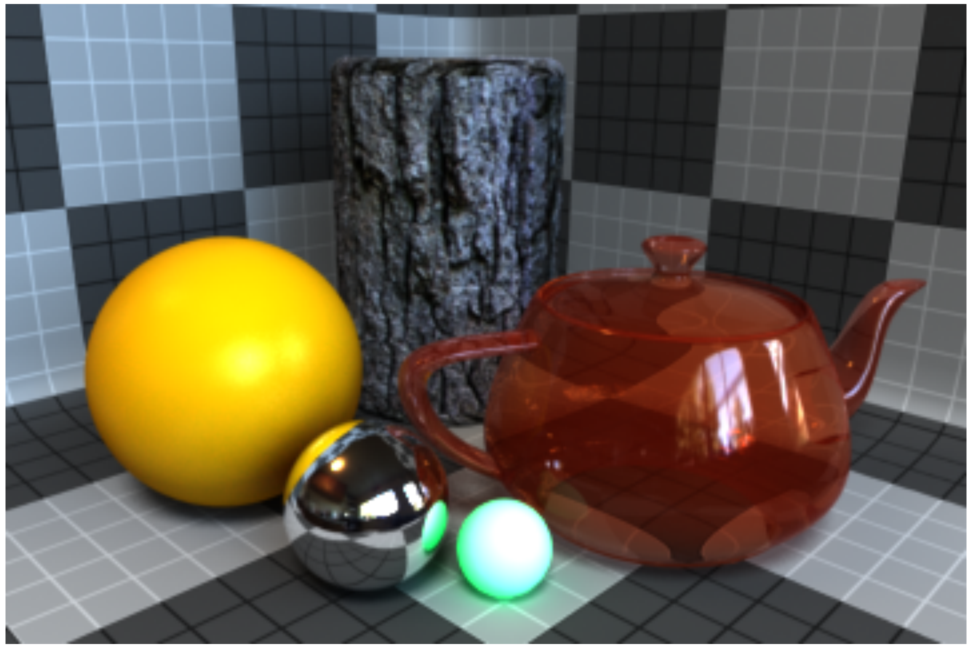
Image Precision

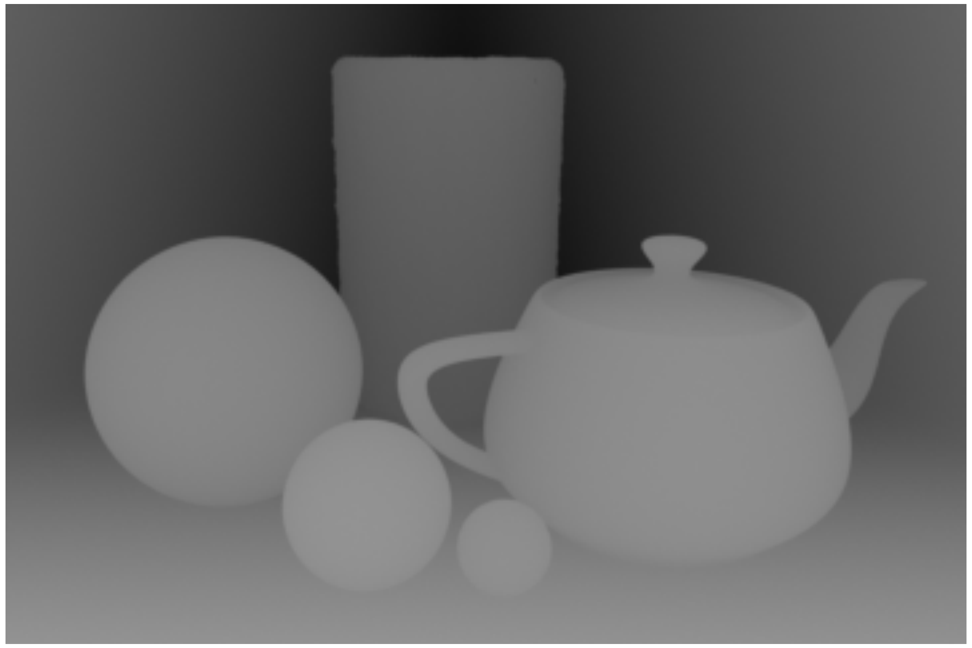
- Record depth information for each pixel
- Z-buffer
  - A 2D array of same size as the frame-buffer
  - Stores depth as real values
- Scan convert primitives in framebuffer and Z-buffer

Z-buffer Algorithm

Initialize *FRAMEBUFFER* to background colour
Initialize *DEPTH* to ∞
foreach *face F* **do**
    foreach *point p of F* **do**
        if *p projects to FRAMEBUFFER[i, j]* **then**
            if *Depth(p) < DEPTH[i, j]* **then**
                $FRAMEBUFFER[i, j]$ = colour of F at p
                $DEPTH[i, j] = Depth(p)$

Z-buffer Algorithm





Source: http://docs.luxology.com/modo/701/help/pages/shaderendering/RenderOutputs.html

Z-buffer - Precision

- In practice, the z-values in the buffer are non-negative integers
  - preferable over true floats
- Using an integer range of B values {0, 1, …, B -1}
  - Map 0 to the near clipping plane z = n
  - Map B -1 to the far clipping plane z = f;
  - $z, n, f > 0$ w.r.t the camera space
- Each z-value is sent to a bucket with depth:

$$\Delta z = (f - n)/B$$

Z-buffer - Precision

- If b bits are used to store the z-value, then $B = 2^b$
- We need enough bits to make sure any triangle in front of another triangle will have its depth mapped to distinct depth bins
- E.g.: In a scene where triangles have a separation of at least one meter, $\Delta z < 1$ will be fine
- Two ways to make Δz smaller:
  - Move n and f closer together, or
  - Increase b (may not be possible with many APIs)

Z-buffer - Precision

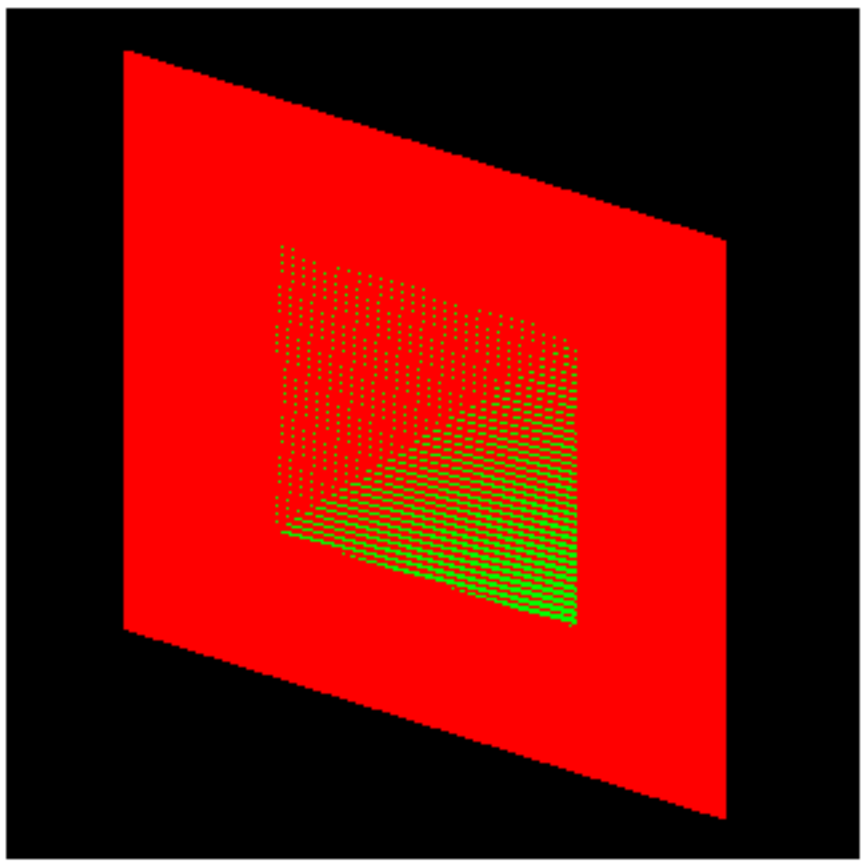
- In case of perspective transformation:

$$z = n + f - \frac{fn}{z_w}$$

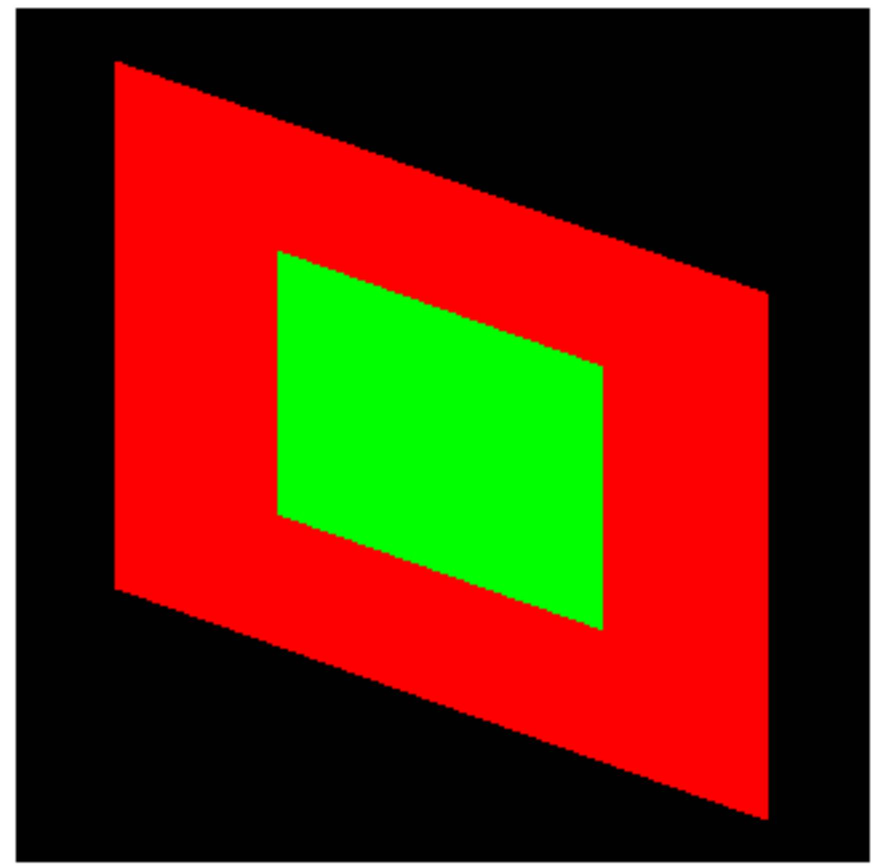
z_w: world depth
z : post-perspective divide depth

- Bin sizes vary with depth: $\Delta z_w \approx \dfrac{z_w^2 \Delta z}{fn}$
- Largest bin size for $z_w = f$: $\Delta z_w^{max} \approx \dfrac{f \Delta z}{n}$
- Cannot choose n= 0 now
- To make Δz_w^{max} as small as possible, we need to minimise f and maximise n

Z-fighting



With z-buffer



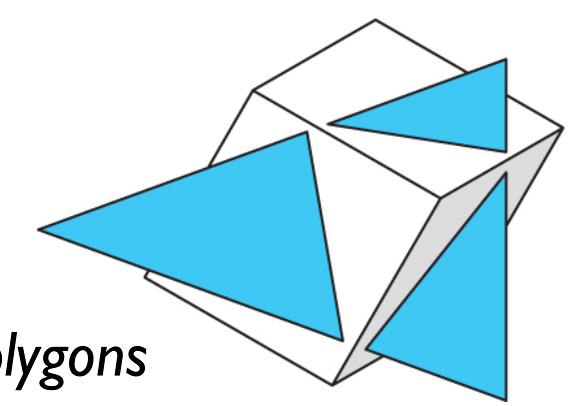
Without z-buffer

Source: http://commons.wikimedia.org/wiki/File:Z_buffer-Z_fighting.png

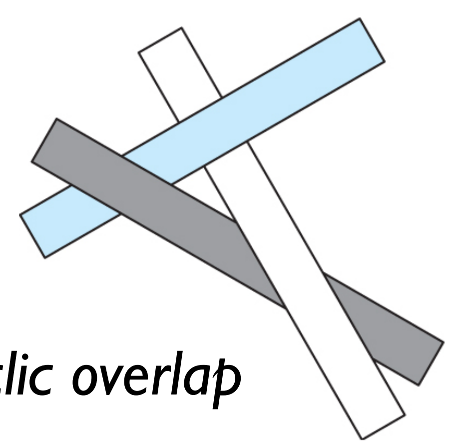
Painter's Algorithm

List Priority

- Faces in a scene are sorted back to front
- A face in front of a set of other faces will not be obscured by them
- Draw the faces from back to front
- Cannot be always done:

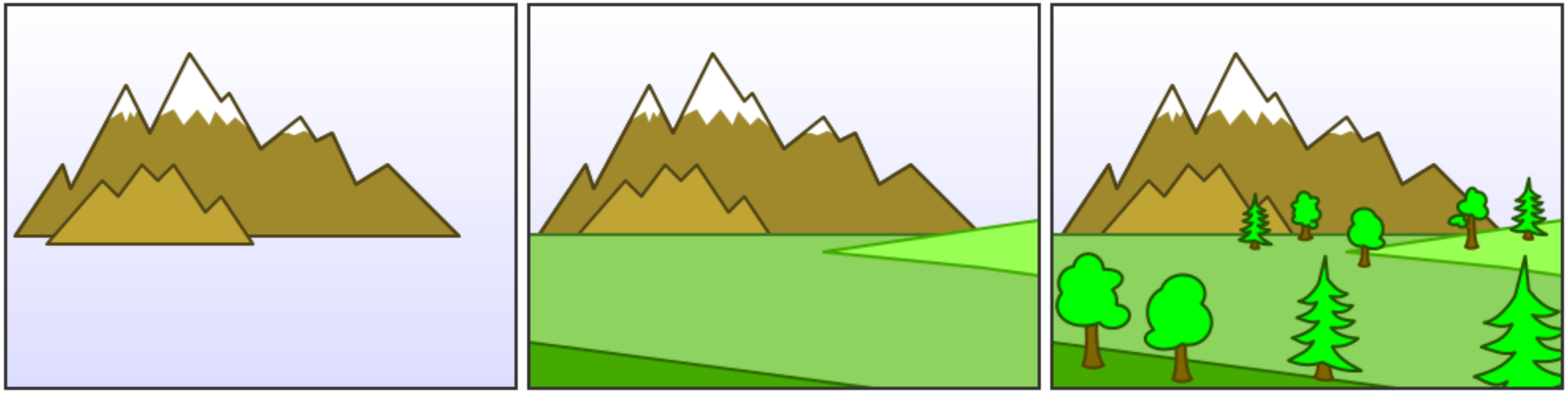


Piercing polygons



Cyclic overlap

Painter's Algorithm



Source: http://en.wikipedia.org/wiki/Painter's_algorithm

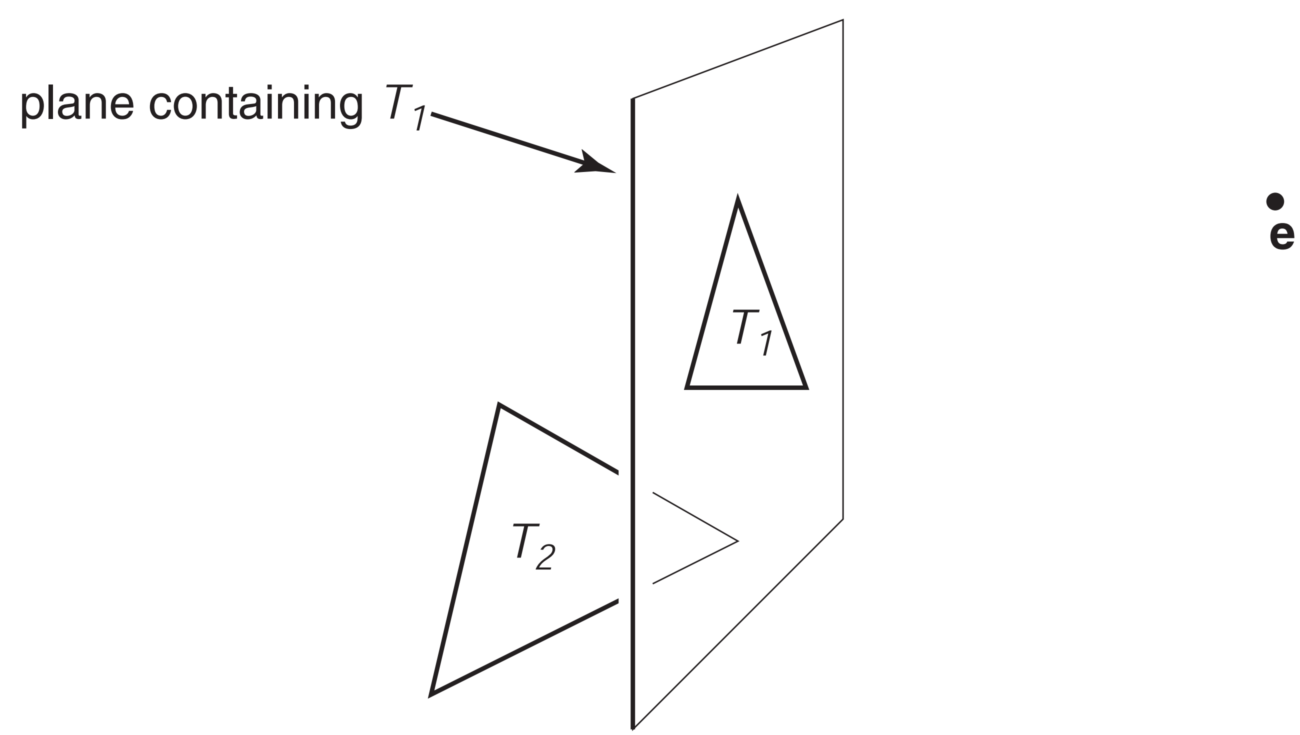
BSP Tree Algorithm

- An example of Painter's algorithm
- Is a list priority algorithm
- Works on any scene composed of polygons where no polygon crosses the plane defined by any other polygon
- This restriction is relaxed by a pre-processing step

BSP Tree: Basic Idea

- Consider two triangles T_1 and T_2
- T_1: $f_1(p) = 0$ for $p \in T_1$, and let $f_1(p) < 0$ for all points p in T_2
- Then for any viewpoint e, correct rendering is:

if $f_1(\mathrm{e}) < 0$ **then**
    draw T_1
    draw T_2
else
    draw T_2
    draw T_1



BSP Tree: Basic Idea

- This observation can be generalised to many objects provided none of them span the plane defined by T_1
- Construct a binary tree data structure with:
  - root: T_1,
  - negative branch: objects with vertices satisfying $f_1(p) < 0$,
  - positive branch: objects with vertices satisfying $f_1(p) > 0$

BSP Tree Algorithm

Procedure `draw`(*bsptree tree*, *point* e)

    if *tree.empty* **then**

        return

    if $f_{tree.root}(\mathrm{e}) < 0$ **then**

        `draw`(*tree.plus*, e)

        rasterise tree.triangle

        `draw`(*tree.minus*, e)

    else

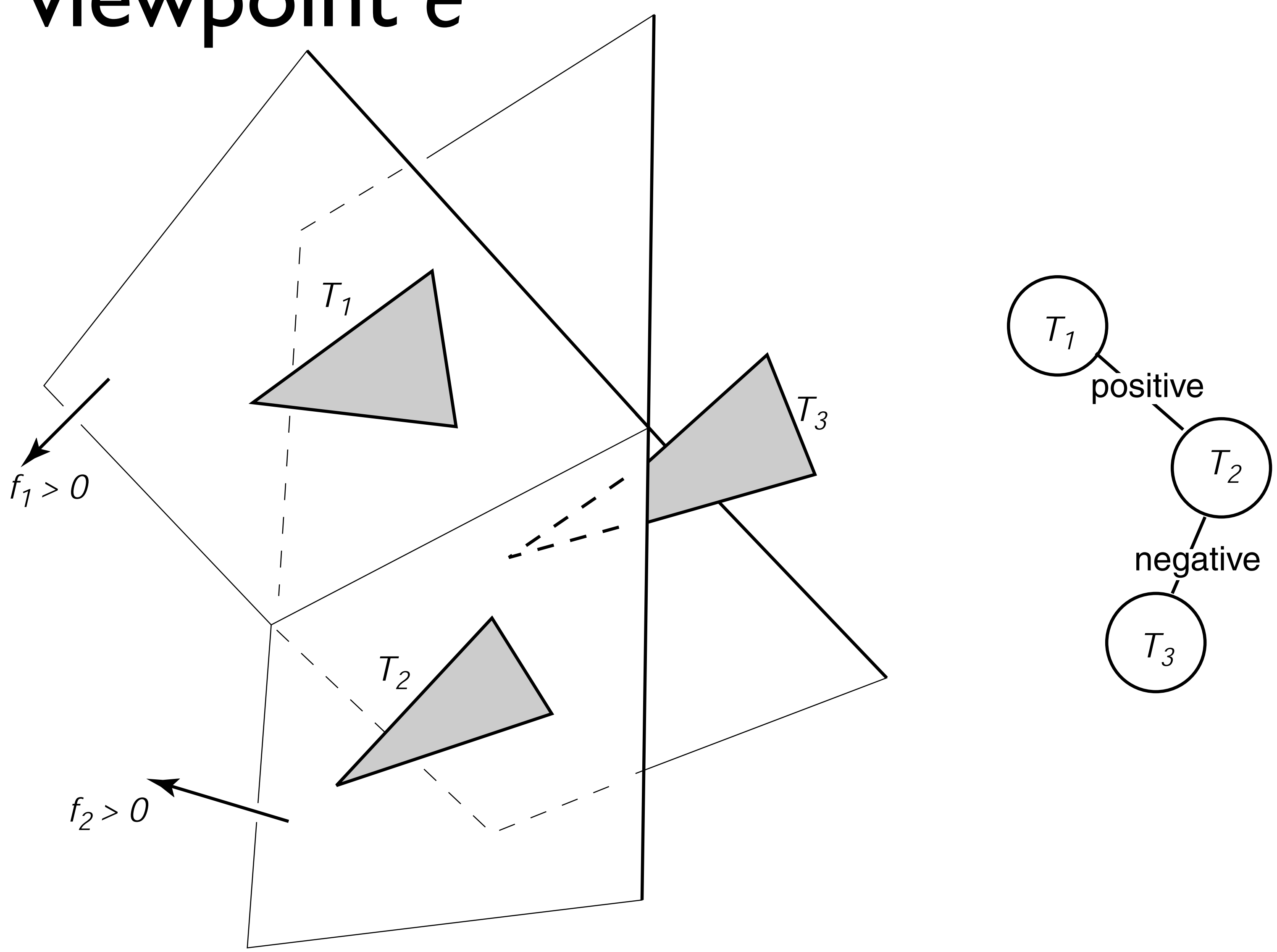
        `draw`(*tree.minus*, e)

        rasterise tree.triangle

        `draw`(*tree.plus*, e)

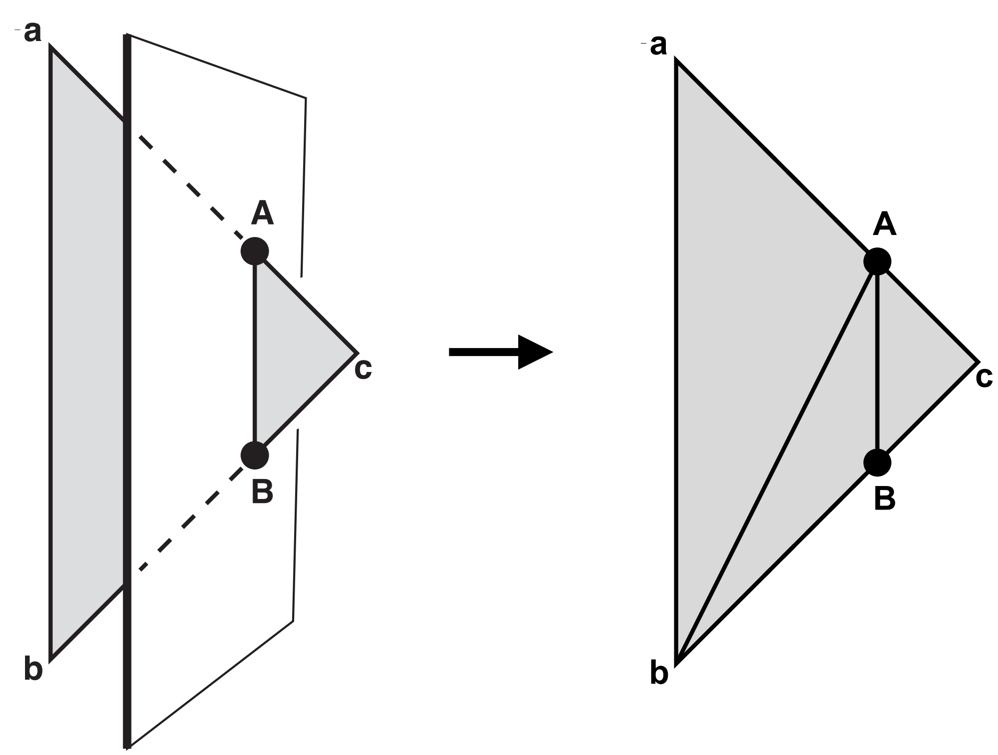
BSP Tree Algorithm

- Once the tree is *pre-computed*, rendering will work for any viewpoint e



BSP Tree Algorithm

- When a triangle spans a plane, there will be one vertex on one side and two on the other
- Splitting is performed in that case



Reading

- ICG: 6.11
- FCG: 8.2.3
- CG: 36 (Notes on visibility)

ICG: Interactive Computer Graphics, E. Angel, and D. Shreiner, 6th ed.
FCG: Fundamentals of Computer Graphics, P. Shirley, M. Ashikhmin, and S. Marschner, 3rd ed.
CG: Computer Graphics, principles and Practice, J. F. Hughes, et al.