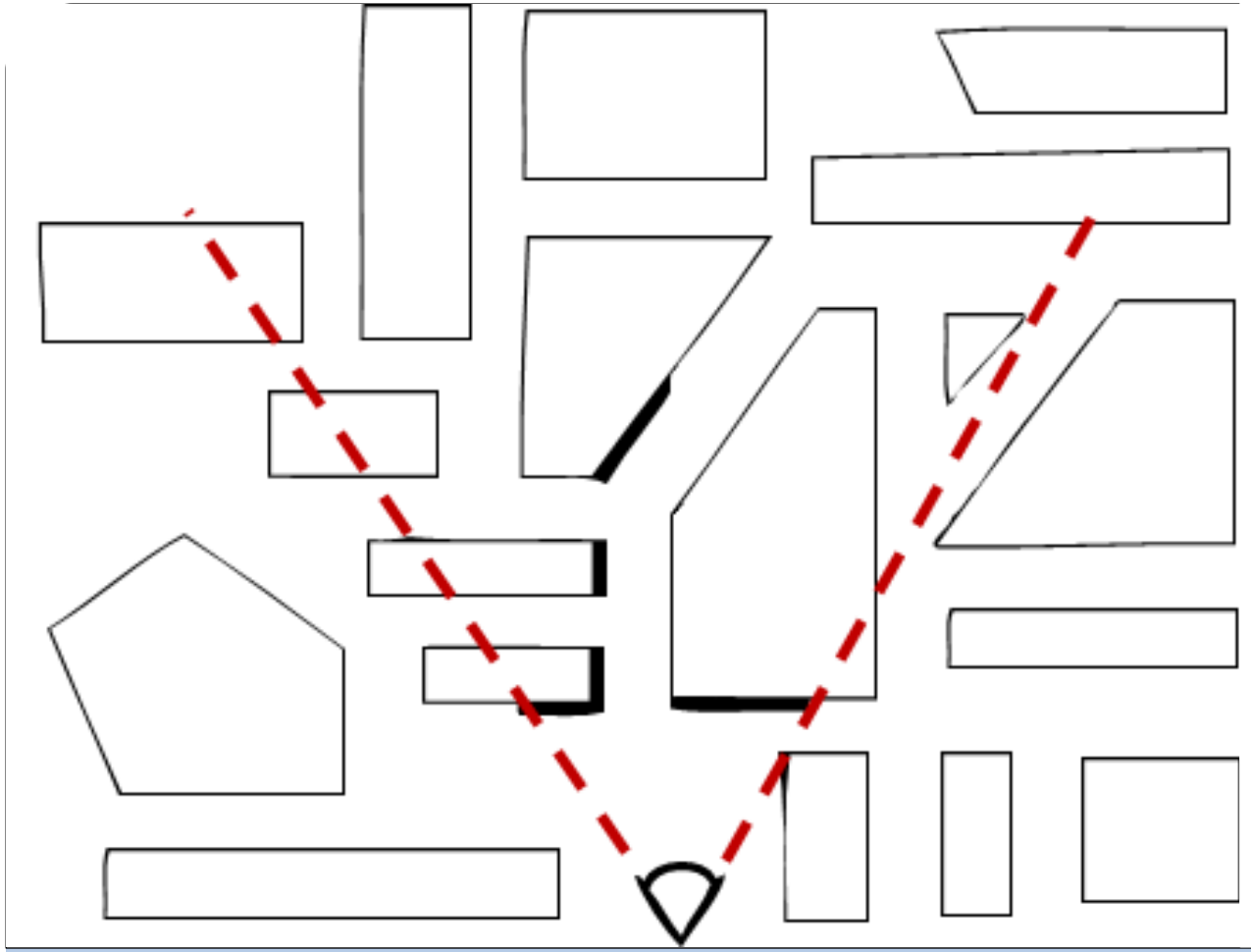


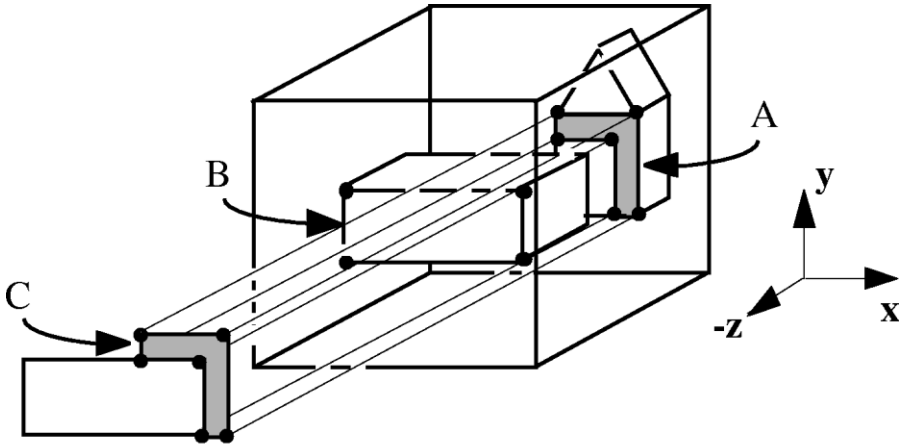
Computer Graphics

Διαγραφή Πίσω Επιφανειών και Απόκρυψη

Andreas Aristidou
andarist@ucy.ac.cy
http://www.andreasaristidou.com

Review: Rendering Pipeline

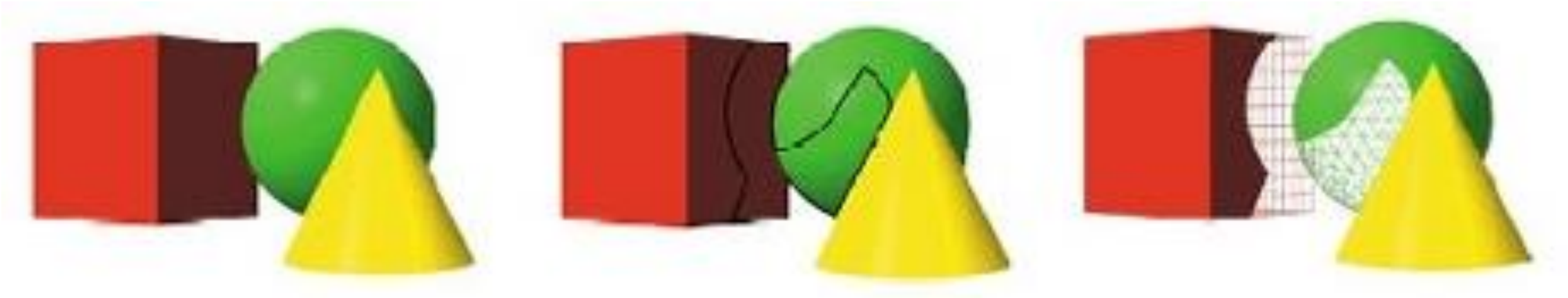
- We almost completed the rendering pipeline:
  - Modeling transformations
  - Viewing transformations
  - Projection transformations
  - Clipping
  - Scan conversion
- Now we know everything about how to draw a polygon on the screen, except for determining the non-visible subjects, or else ***visible surface detection***.



Polygon *A* is clipped by *B* which is in front of it. A new sub-polygon, *C*, is created.

Visible Surface Determination and Back Face Culling

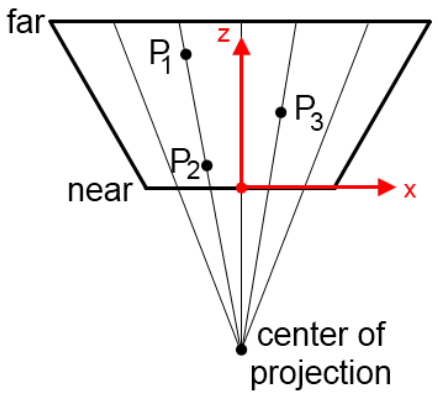
- Specifies the part of each object that is visible in the final image
- Why are **back face culling** algorithms needed?
  - Avoid creating incorrect images
  - Speed up the creation of images
- We must determine what is visible within a scene from a chosen viewing position
- For 3D worlds this is known as **visible surface detection** or **hidden surface elimination**

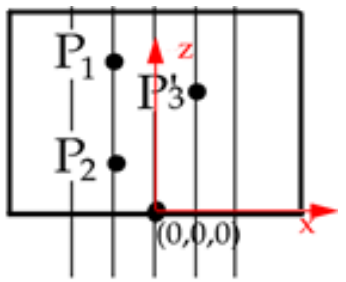


To render or not to render, that is the question…

Contents

- Today we will deal with visible surface detection techniques:
  - Why Surface Detection;
  - Detect the back of the object
  - Depth-buffer Method
  - A-buffer Method
  - Scan-line Method





Invisible entities

- *When would a polygon be invisible?*
  - Polygons that are outside the field of view
  - If from the polygon we see its back (backfacing)
  - If the polygon is hidden from another that is closer to the viewpoint
- As you can imagine, for performance reasons we do not want to deal with polygons that are invisible (outside field of view or backfacing)
- For rendering and proper visualization, we need to know if a polygon is hidden from another object.

Visible Surface Detection

There are many algorithms and techniques that have been developed from time to time to solve visible surface detection

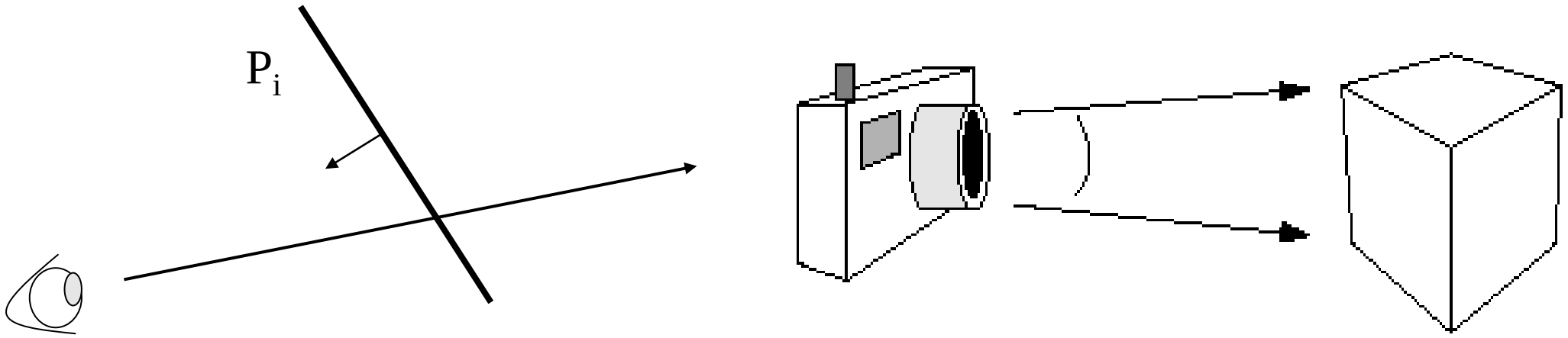
- Some methods require more processing time.
- Some methods require more memory.
- Others can only be applied to specific data types

Two Main Approaches

- Visible surface detection algorithms are broadly classified as:
  - **Object Space Methods:** Compares objects and parts of objects to each other within the scene definition to determine which surfaces are visible
  - **Image Space Methods:** Visibility is decided point-by-point at each pixel position on the projection plane
- Image space methods are by far the more common

Back-Face Detection

- We assume that our objects are solid
- Each polygon has "back and forth", depending on the order of its edges
- The simplest thing we can do is to find the faces on the back of the polyhedron and discard them
- Polygons whose slash does not look towards the camera are not rendered



Back-Face Detection

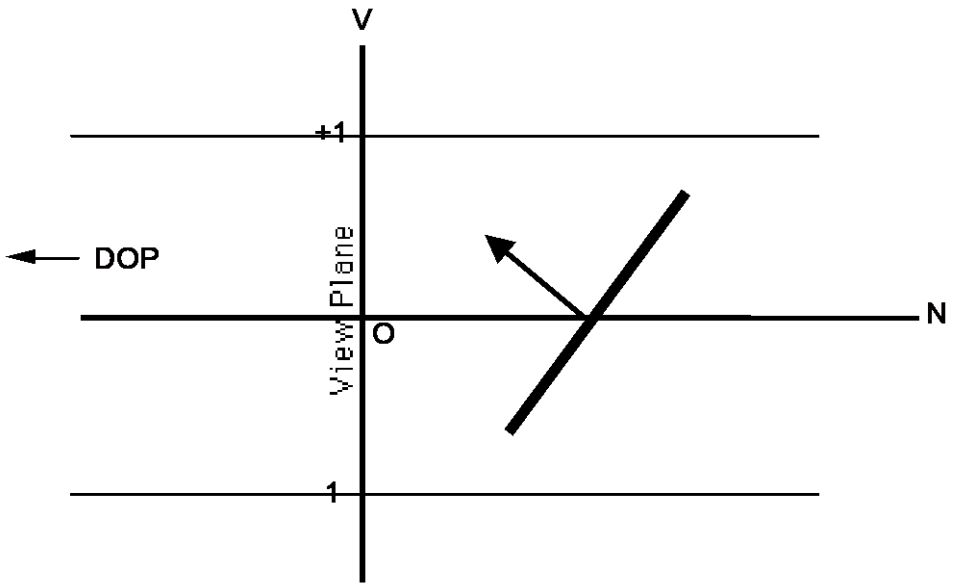
- We know from before that a point (x, y, z) is behind a polygon surface if:

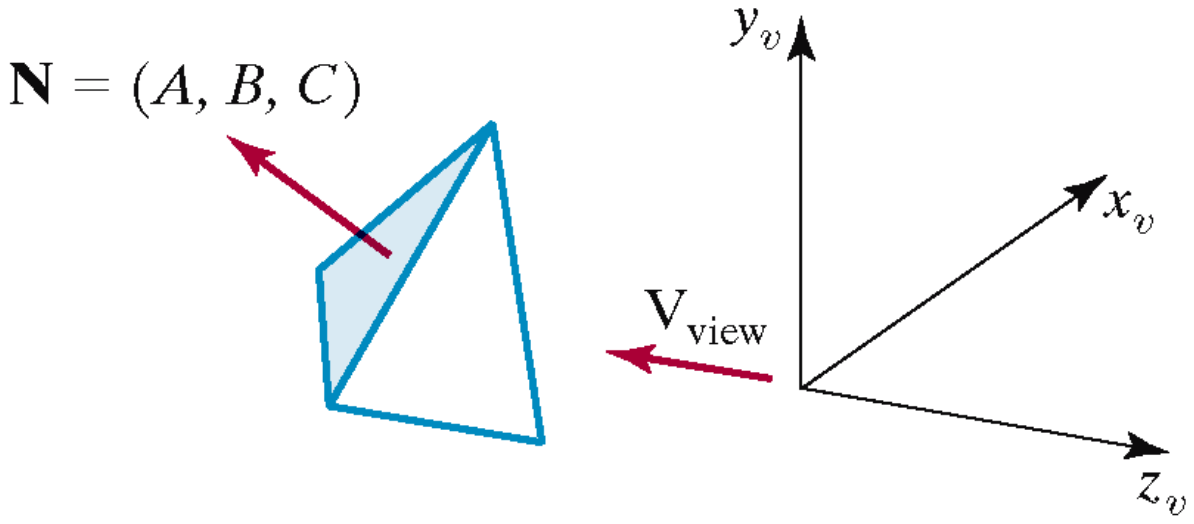
$$Ax + By + Cz + D < 0$$

- where A, B, C & D are the plane parameters for the surface
- This can actually be made even easier if we organise things to suit ourselves

Back-Face Detection

- Ensure we have a right handed system with the viewing direction along the negative Z-axis
- Now we can simply say that if the Z component of the polygon's normal is less than zero the surface cannot be seen





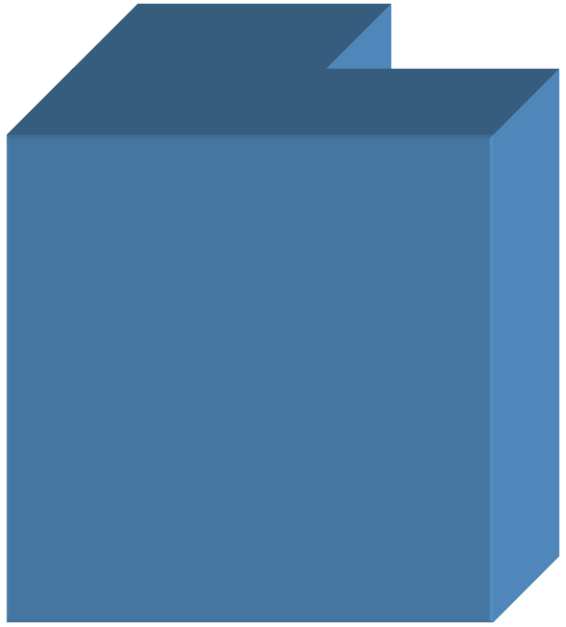
Back-Face Detection

- We can simplify this method if we assume that the vertical vector N on the surface of the polygon has the Cartesian values (A,B,C).

- If V is a vector in the direction of projection, then the polygon is a back surface and it is not visible if **V●N > 0.**

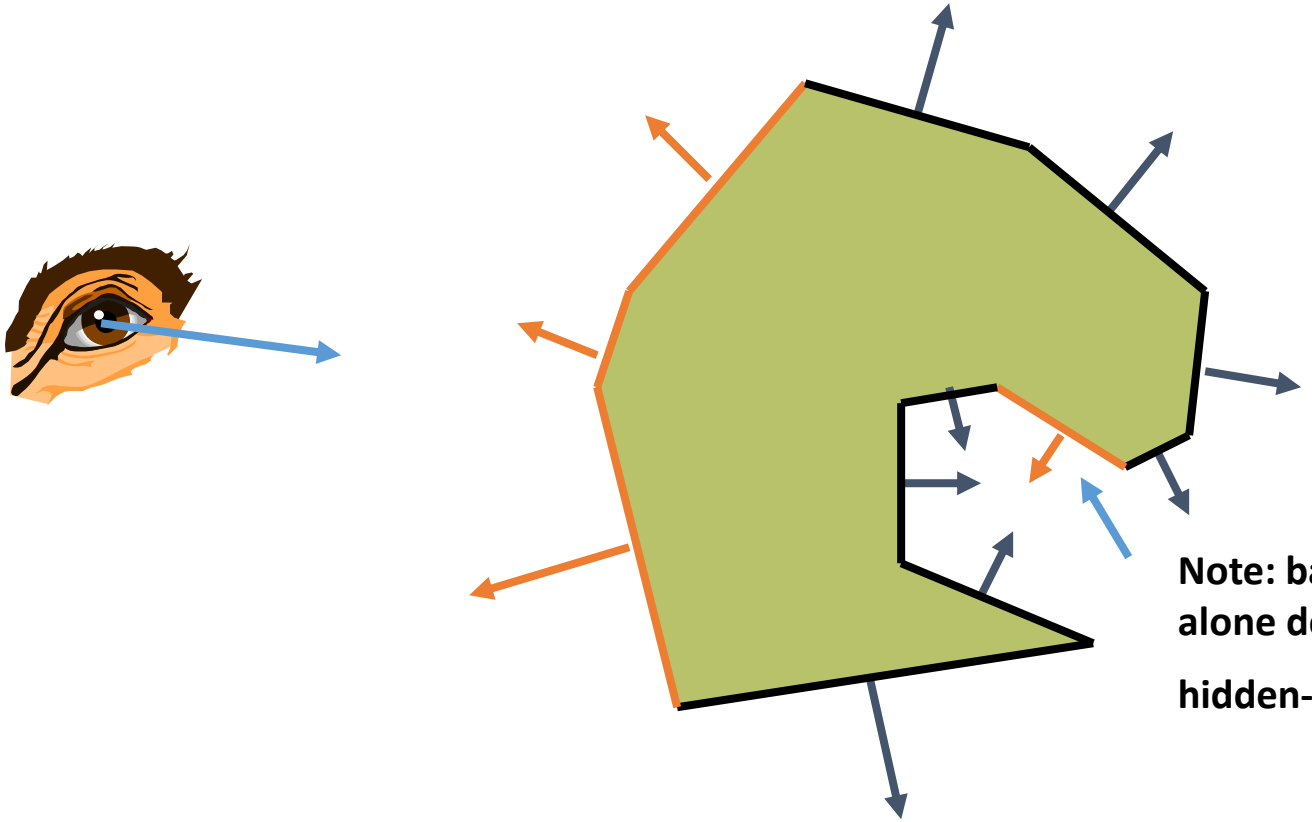
Back-Face Detection

- In general back-face detection can be expected to eliminate about half of the polygon surfaces in a scene from further visibility tests
- More complicated surfaces though scupper us!
- We need better techniques to handle these kind of situations



Back-Face Detection

- On the surface of polygons whose vertical vectors point away from the camera are always invisible:

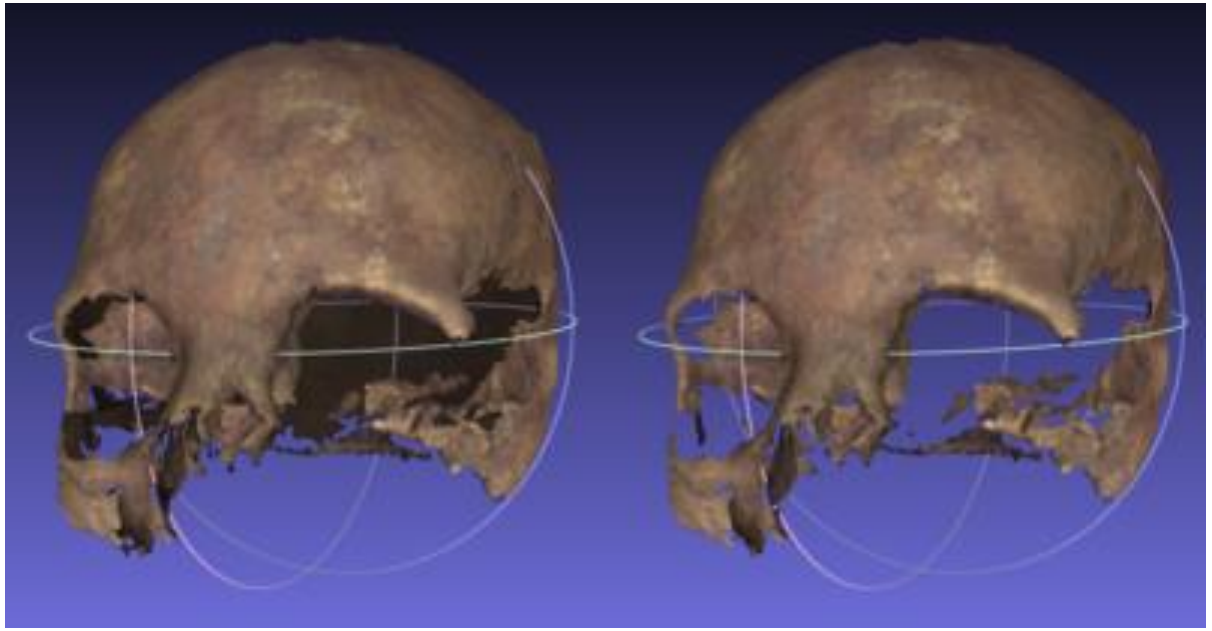


Note: backface detection alone doesn't solve the hidden-surface problem!

Backface culling assumes objects are closed

No Backface culling

With Backface culling

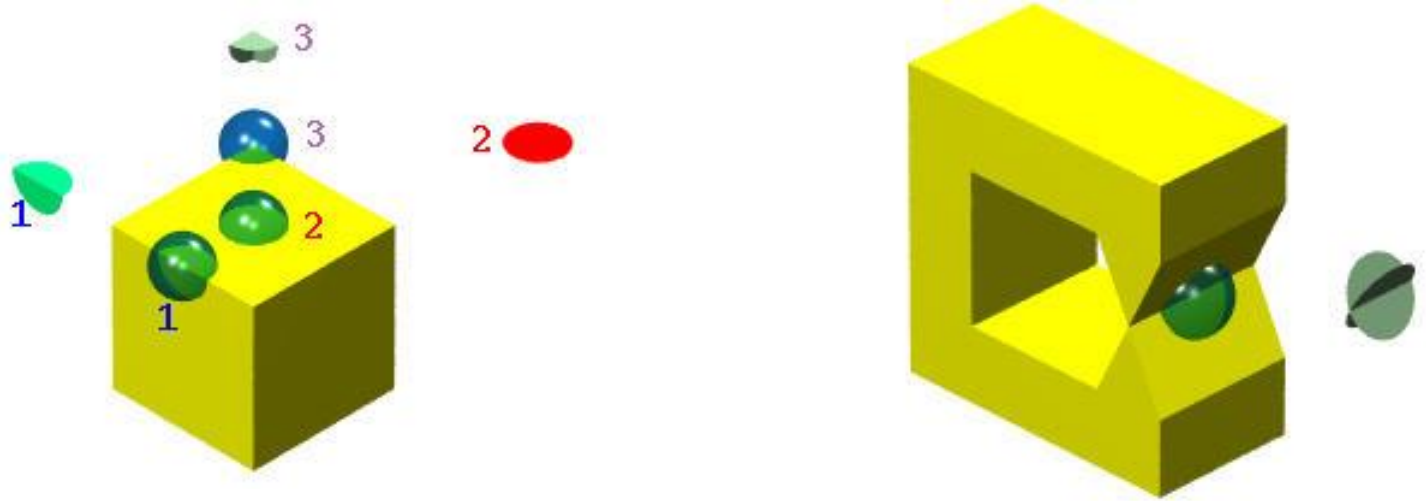


https://en.wikipedia.org/wiki/Back-face_culling

A triangle facing away from you is only guaranteed to be invisible if there's something else in front of.
This is only guaranteed for closed, solid objects (e.g. a sphere)
Objects with 'holes' may expose back-facing triangles to the viewer; backface culling results in errors (see skulls on right)

Back-Face Detection

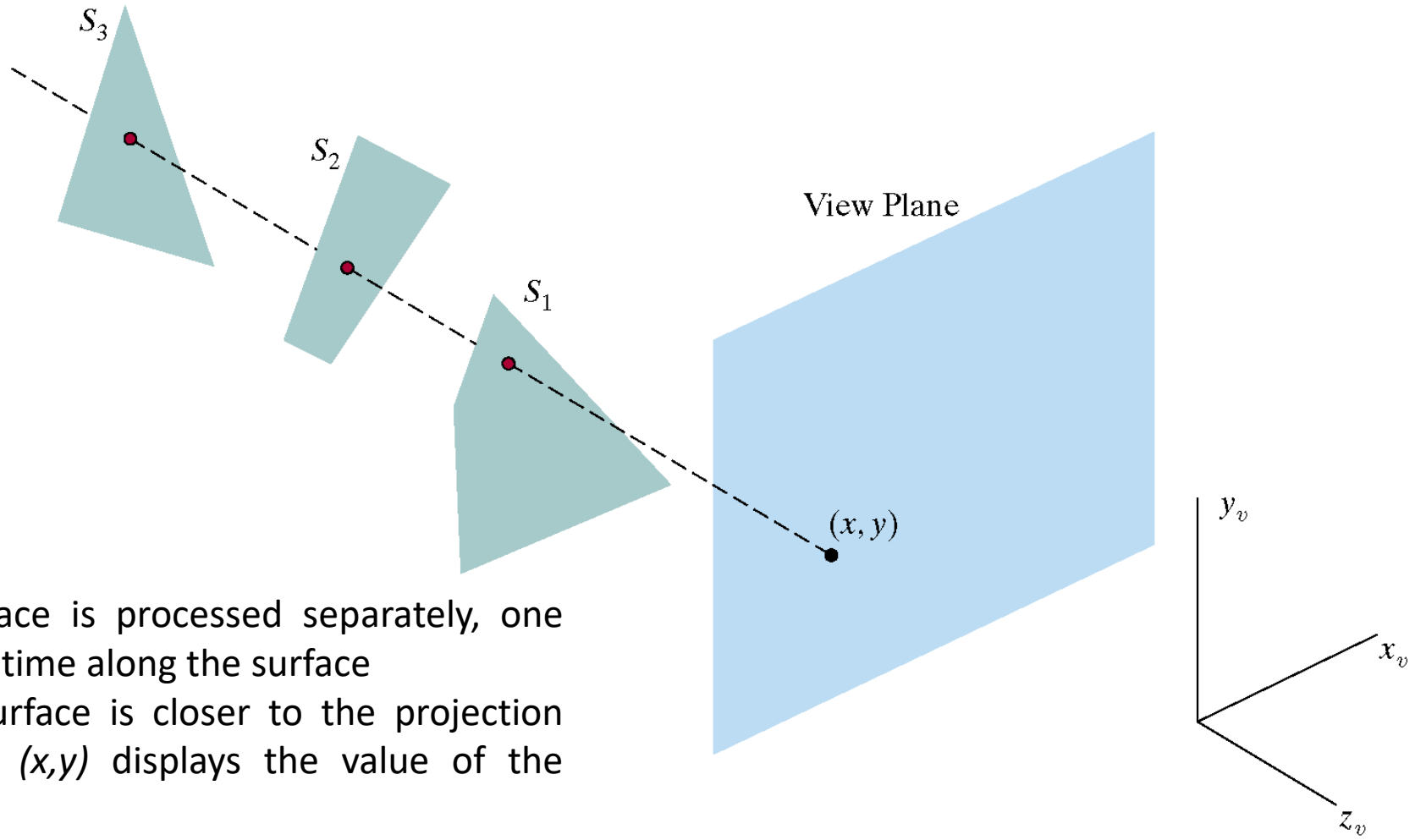
- Most objects on stage are usually "solid" and non-transparent!
  - What happens if it's not?



Depth-Buffer Methods

- Compares surface depth values throughout a scene for each pixel position on the projection plane
- Usually applied to scenes only containing polygons
- As depth values can be computed easily, this tends to be very fast
- Also often called the z-buffer method

Depth-Buffer Methods



- Each surface is processed separately, one point at a time along the surface
- The **S1** surface is closer to the projection plane, so *(x,y)* displays the value of the object **S1**.

Depth-Buffer Algorithm

1. Initialise the depth buffer and frame buffer so that for all buffer positions (*x, y*)

   depthBuff(x, y) = 1.0

   frameBuff(x, y) = bgColour

Depth-Buffer Algorithm

2. Process each polygon in a scene, one at a time
   - For each projected (x, y) pixel position of a polygon, calculate the depth z (if not already known)
   - If z < depthBuff(x, y), compute the surface colour at that position and set

     depthBuff(x, y) = z

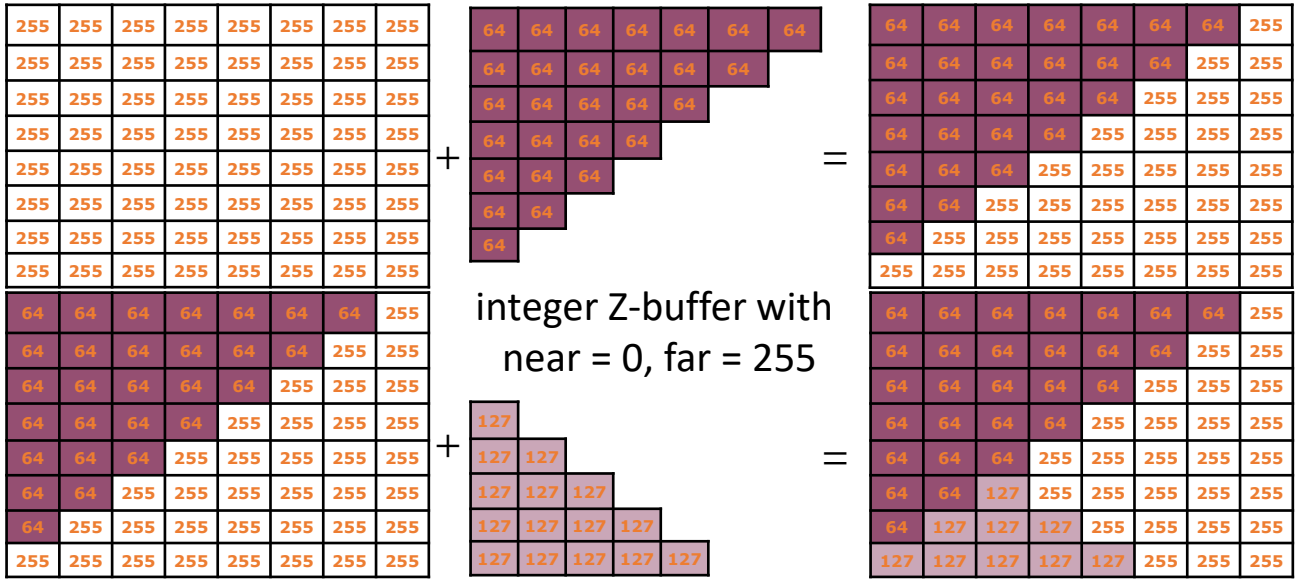
     frameBuff(x, y) = surfColour(x, y)

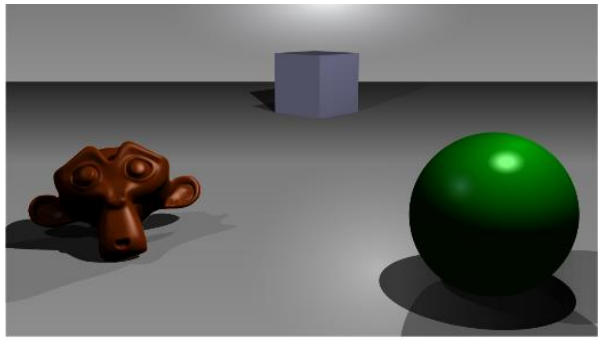
After all surfaces are processed depthBuff and frameBuff will store correct values

Depth-Buffer Algorithm

```
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);
      }
    }
  }
}
```

Depth-Buffer Algorithm





A simple three-dimensional scene

Z-buffer representation

Iterative Calculations

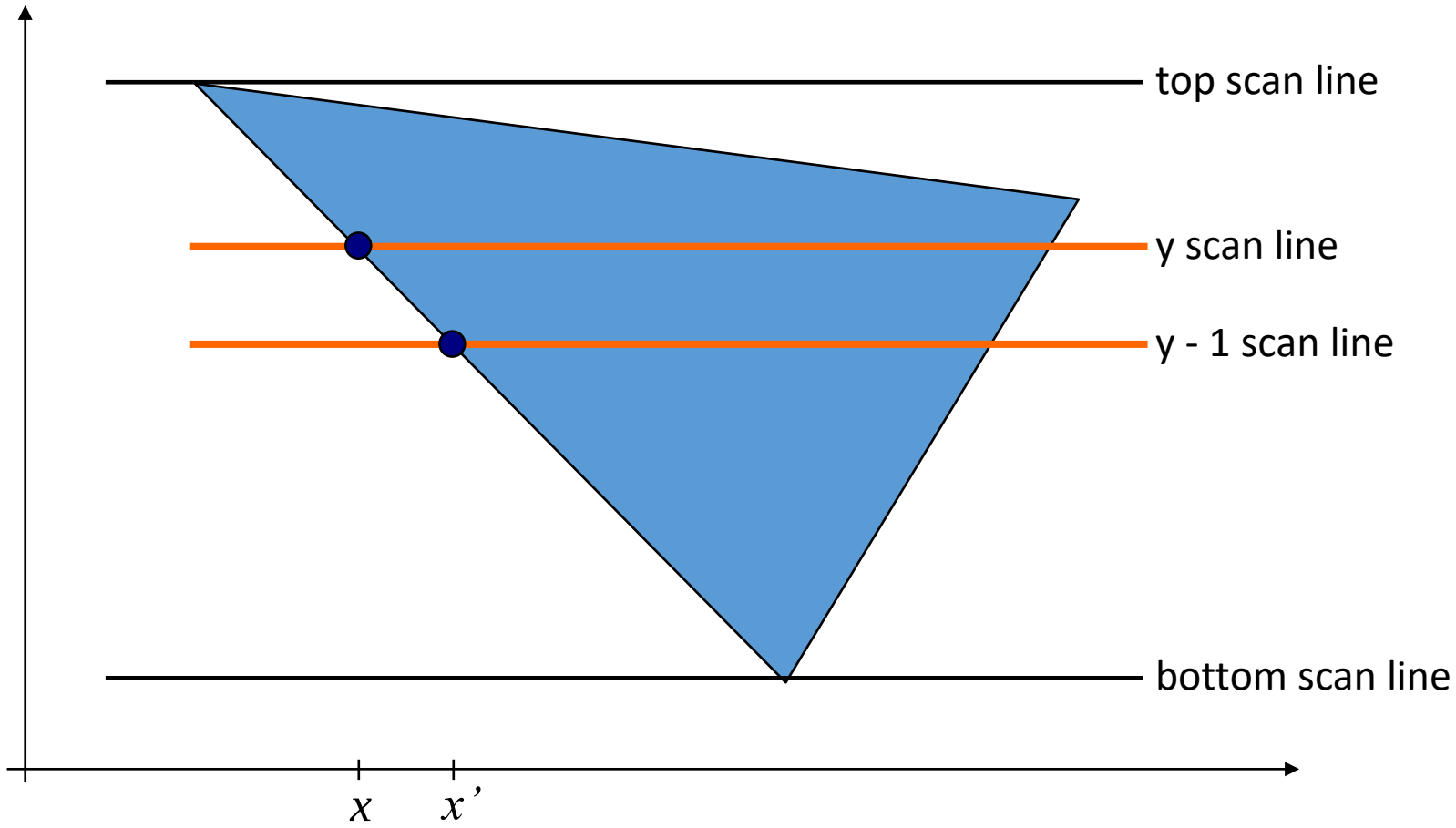
- The depth-buffer algorithm proceeds by starting at the top vertex of the polygon
- Then we recursively calculate the x-coordinate values down a left edge of the polygon
- The x value for the beginning position on each scan line can be calculated from the previous one

$$x' = x - \frac{1}{m}$$ where m is the slope

- Depth values along the edge being considered are calculated using

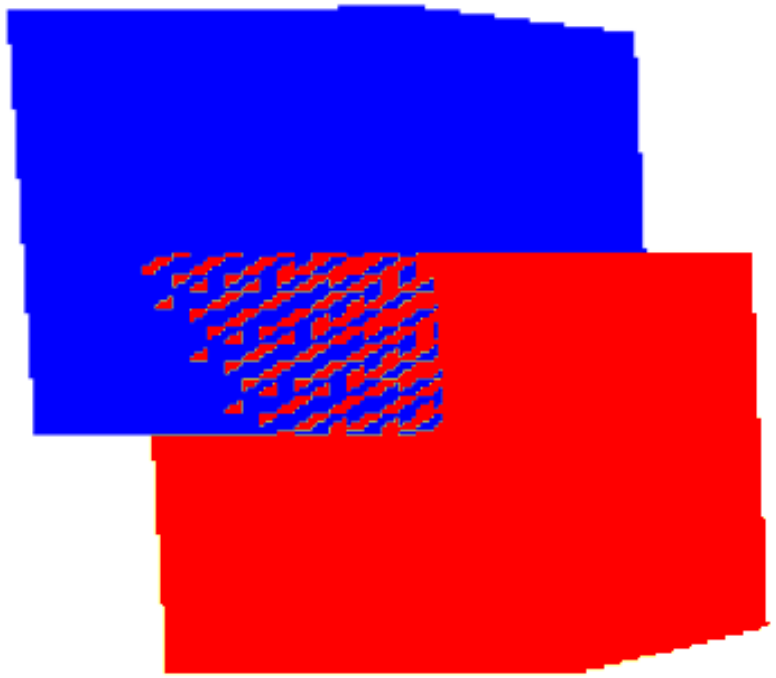
$$z' = z - \frac{A/m + B}{C}$$

Iterative Calculations



Z-Fighting

- Z-fighting occurs when two primitives have similar values in the z-buffer



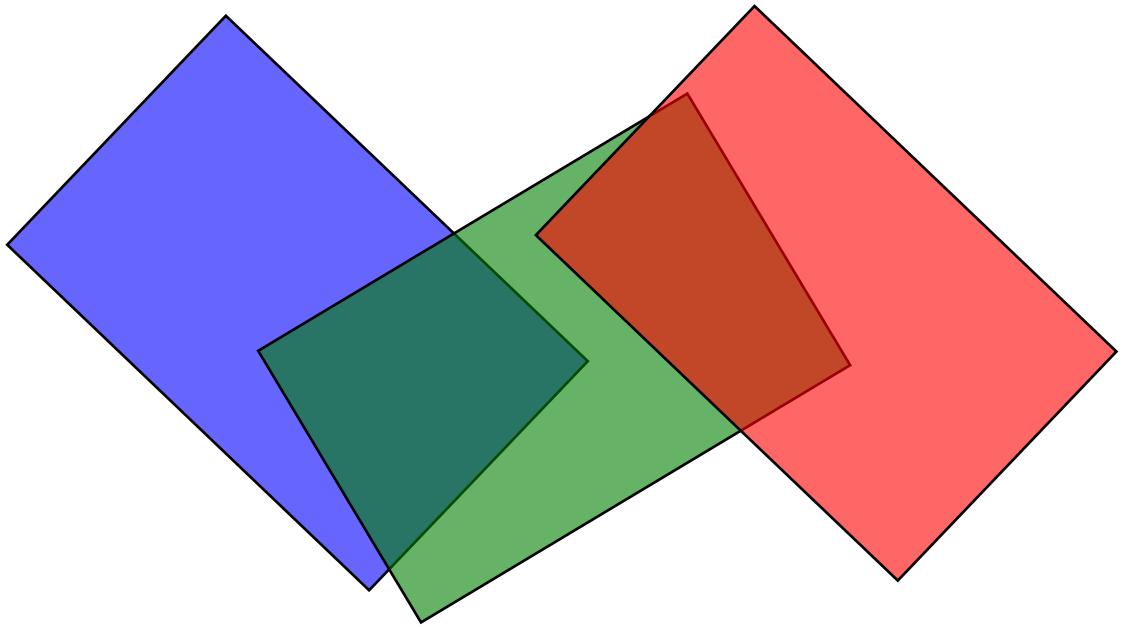
Two intersecting cubes

A-Buffer Method

- The A-buffer method is an extension of the depth-buffer method
- The A-buffer method is visibility detection method developed at Lucasfilm Studios for the rendering system REYES (**R**enders **E**verything **Y**ou **E**ver **S**aw)

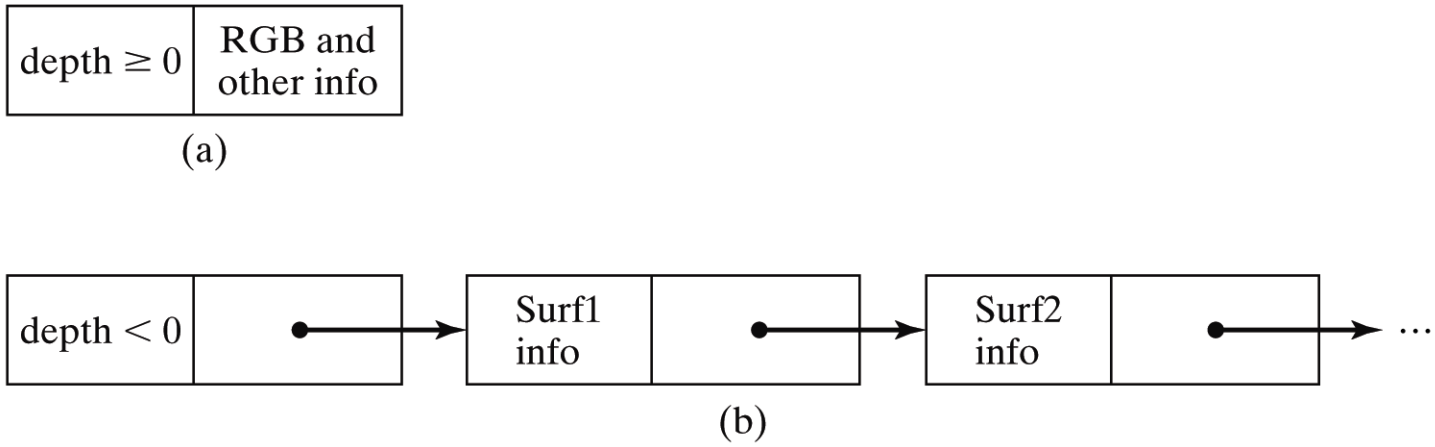
A-Buffer Method

- The A-buffer expands on the depth buffer method to allow transparencies
- The key data structure in the A-buffer is the *accumulation buffer*



A-Buffer Method

- If depth is >= 0, then the surface data field stores the depth of that pixel position as before
- If depth < 0 then the data filed stores a pointer to a linked list of surface data



A-Buffer Method

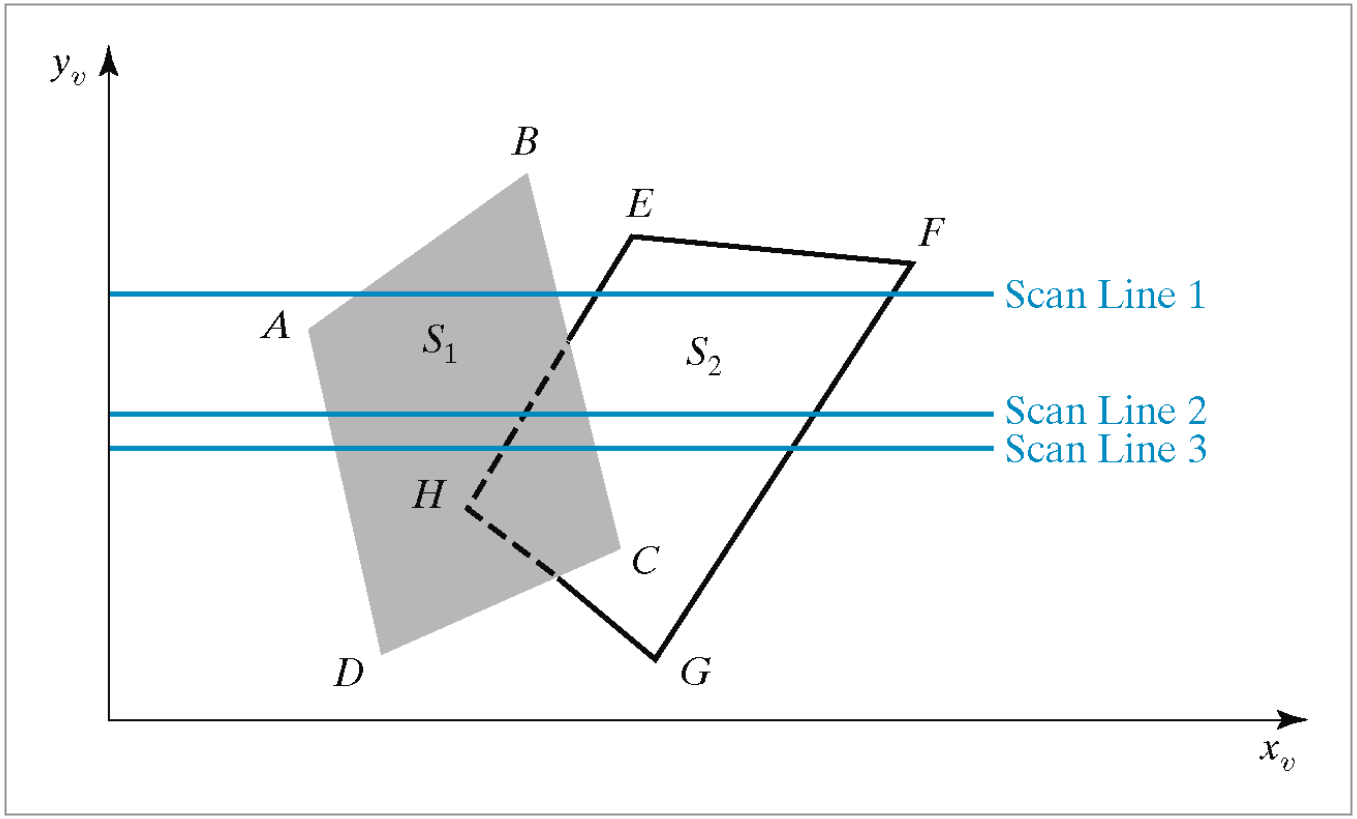
- Surface information in the A-buffer includes:
  - RGB intensity components
  - Opacity parameter
  - Depth
  - Percent of area coverage
  - Surface identifier
  - Other surface rendering parameters
- The algorithm proceeds just like the depth buffer algorithm
- The depth and opacity values are used to determine the final colour of a pixel

Painter's Algorithm

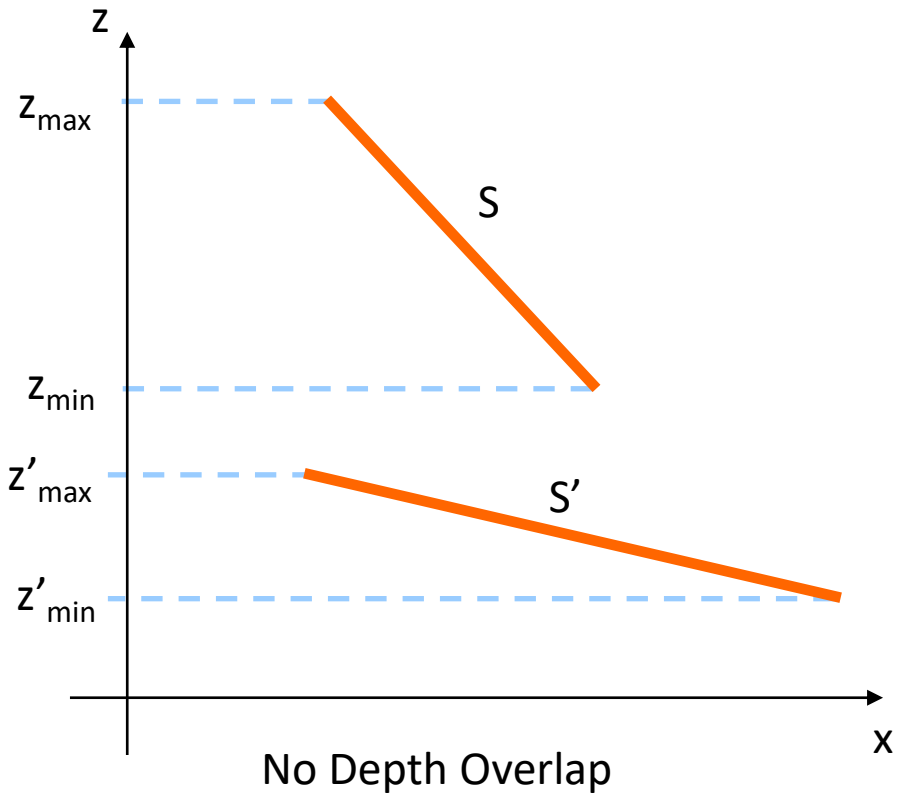
Deadpool © Twentieth Century Fox, Marvel Entertainment

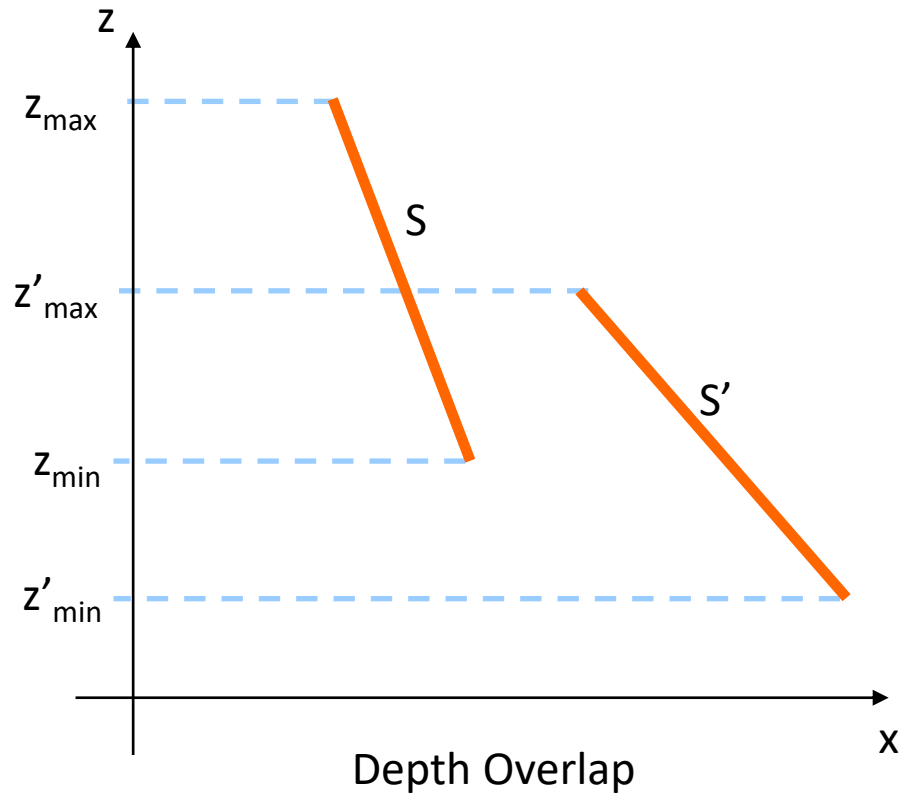
Scan-Line Method

- An image space method for identifying visible surfaces
- Computes and compares depth values along the various scan-lines for a scene



Depth-Sorting Method (Painter's Algorithm)

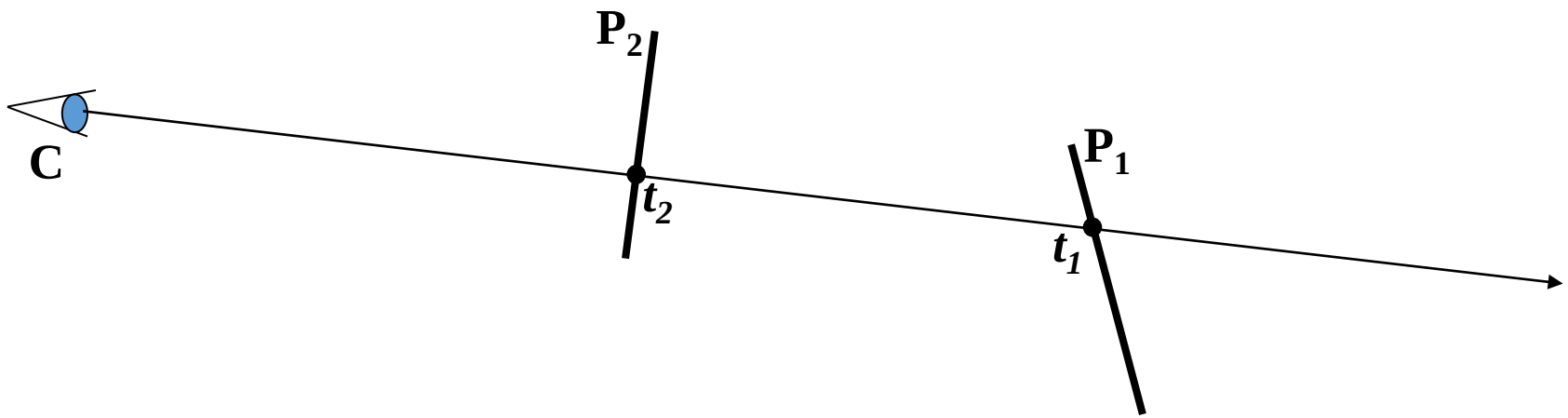




Visibility (Priority) Ordering

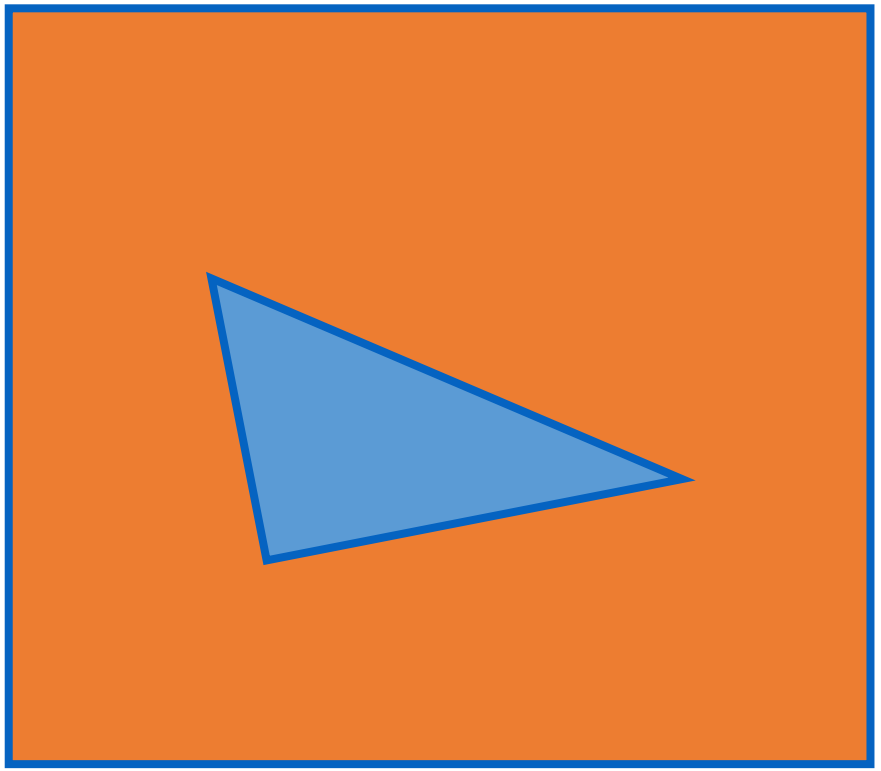
- ***You would draw the P1 after P2 to see the correct result (Painters Algorithm)***

- Given a set of S polygons and a viewpoint C, find a series of $\{P_1 \ldots P_n\}$ at *S* such that any P_i polygon does not hide any of the polygons $\{P_{i+1} \ldots P_n\}$.
- Another way to think about it: for every 2 polygons intersected by a ray through the C, $P_{i\,;}$ has a higher priority than $P_{j,}$ (with i < j)

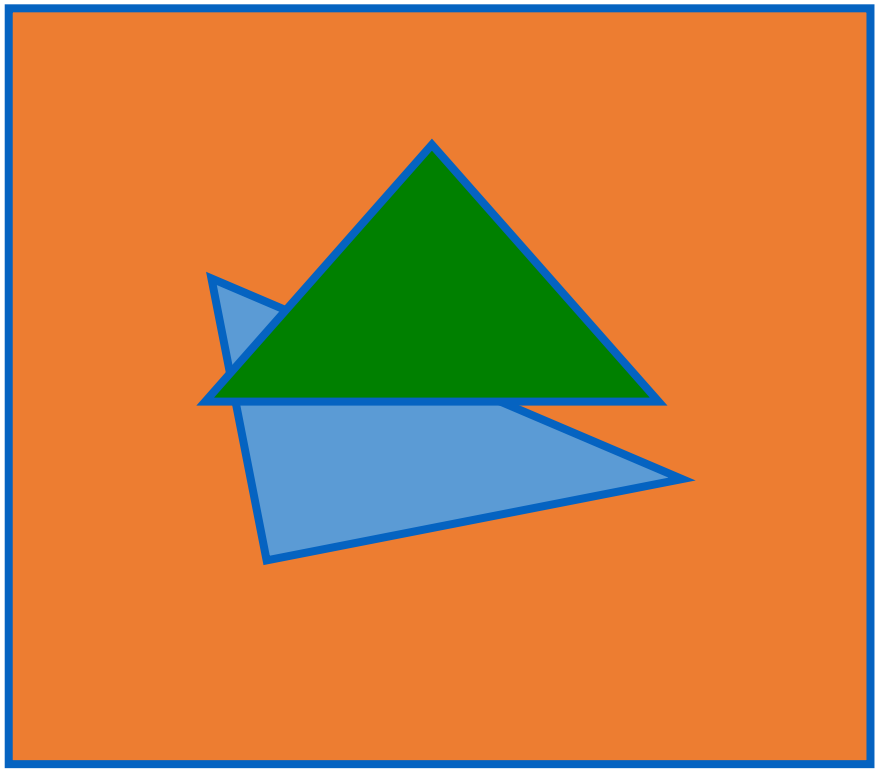


Depth-Sorting Algorithm (Painter's Algorithm)

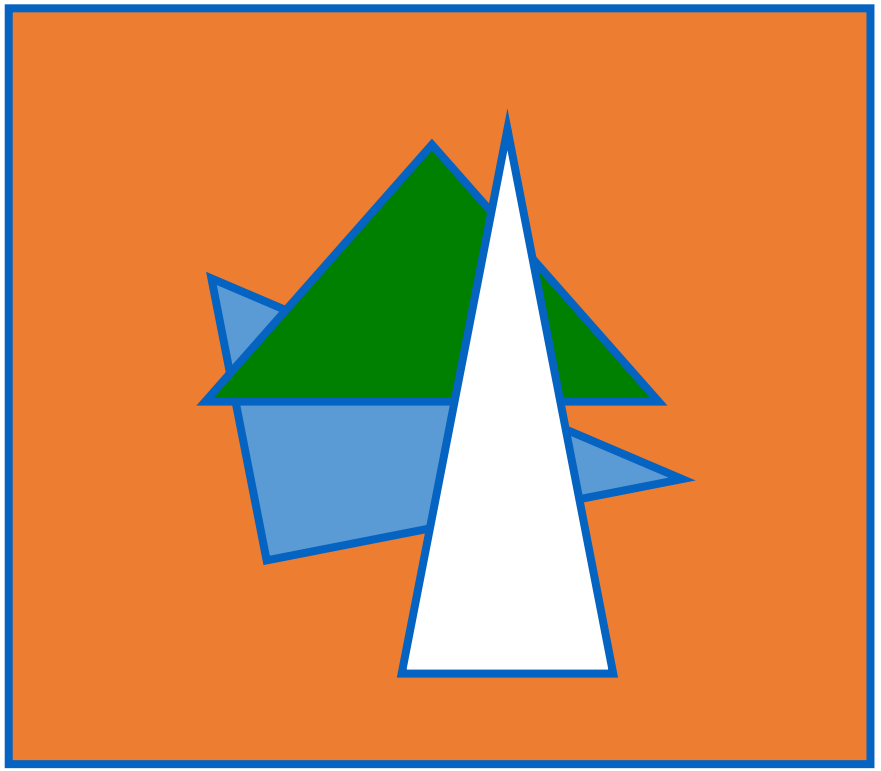
- Simple approach: Deliver the polygons from back to front, "painting above" from the previous polygons:
-



Depth-Sorting Algorithm (Painter's Algorithm)

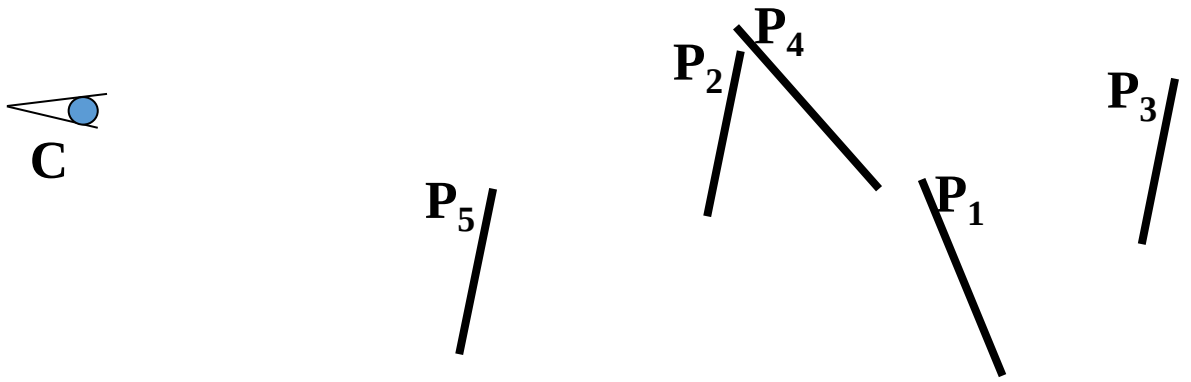


Depth-Sorting Algorithm (Painter's Algorithm)



Principle of the painter's algorithm

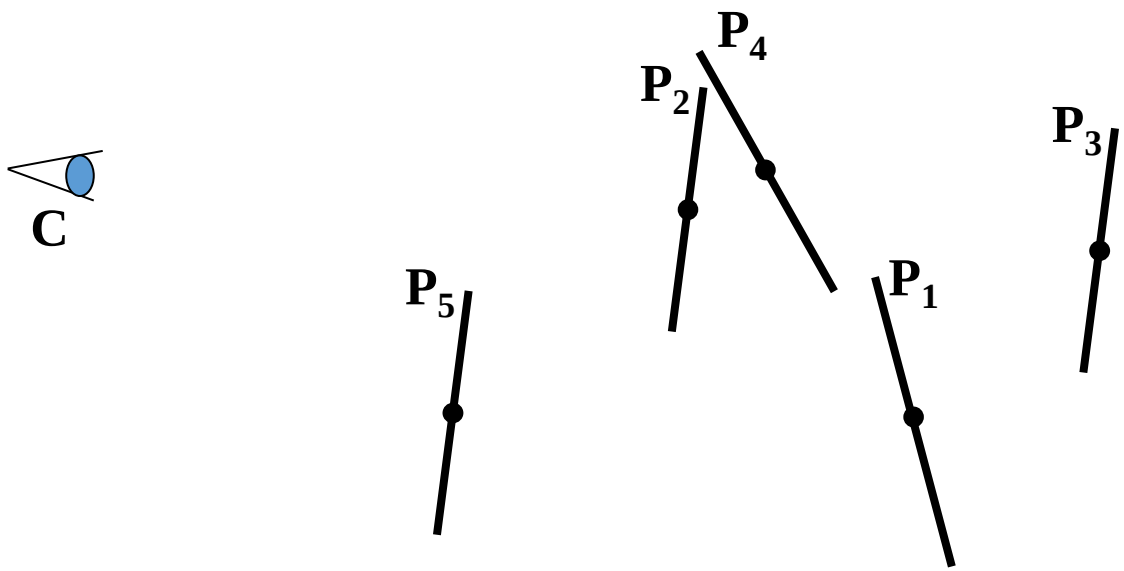
- Sort objects (or polygons) from the back to the front
- To get the final picture: we draw them in this order which allows the last (nearest) to be painted above (overwrite) the previous ones (further away)
-



- It is a hybrid approach as the first step, the sorting, is in the object space and the second in the image space
- **The hardest part is sorting**

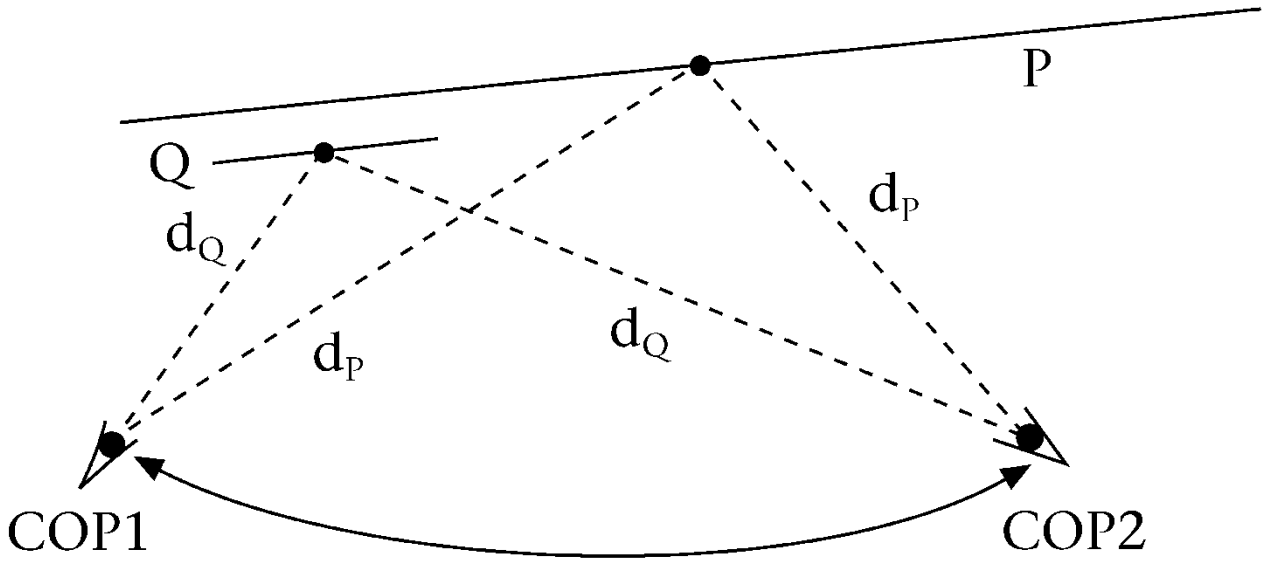
Z-sort at Projection Space

- Simple back-to-front sorting of all polygons based on their center



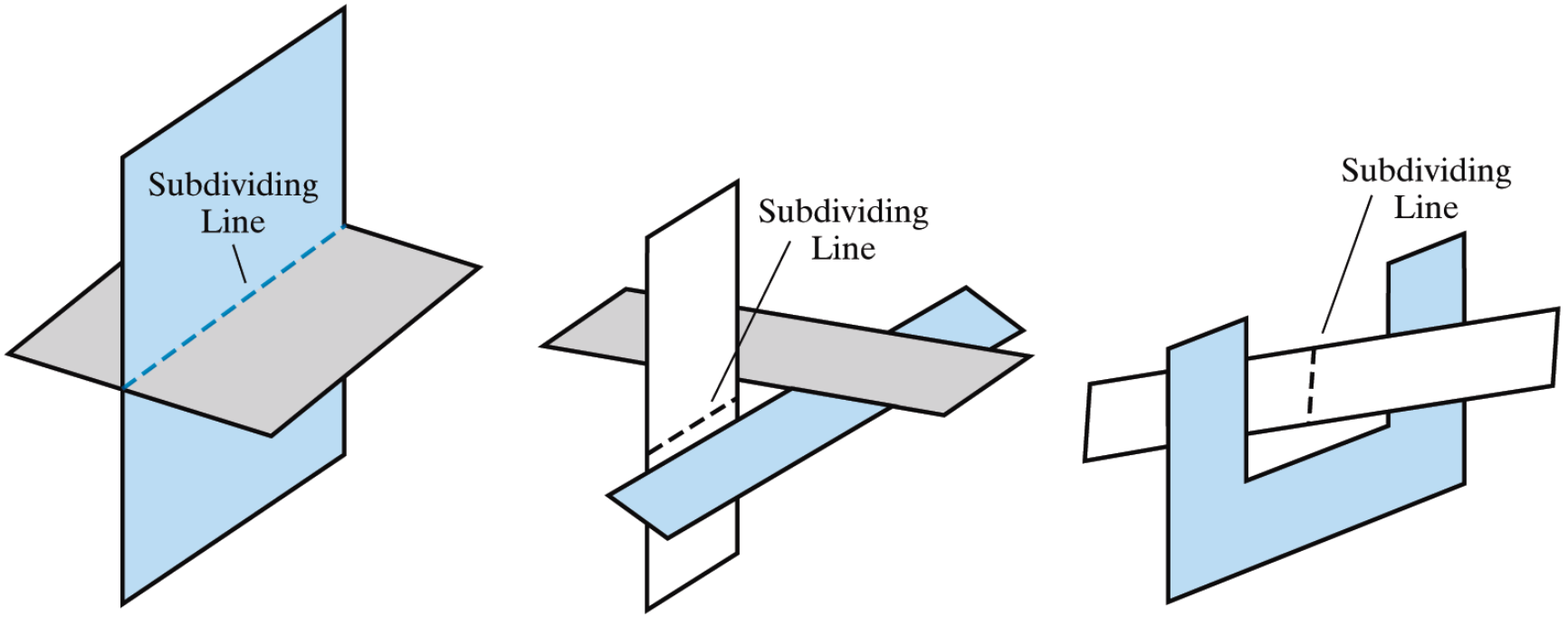
Z-sort at Projection Space

- It doesn't always work, e.g.
-



Scan-Line Method Limitations

- The scan-line method runs into trouble when surfaces cut through each other or otherwise cyclically overlap
- Such surfaces need to be divided



Scan-Line Method Limitations

- The ***Intersecting polygons*** also present a problem
- Even non-intersecting polygons can form a circle without a valid order of visibility:
-

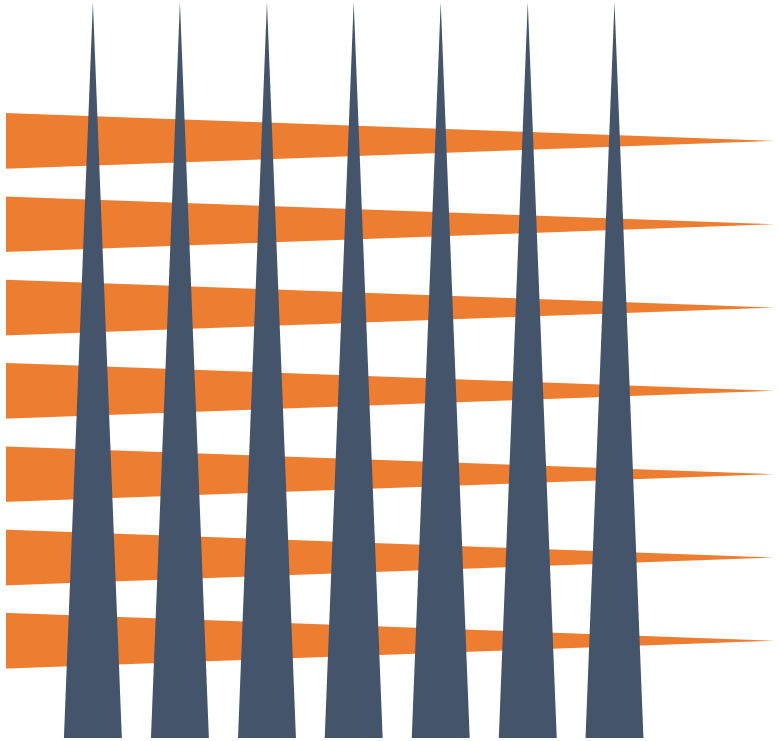
Analytic Visibility Algorithms

- Early visibility algorithms calculate the set of visible parts of the polygon directly, and then present the individual segments on a monitor:

Analytic Visibility Algorithms

- ***What is the minimum worst-case cost of computing the fragments for a scene composed of* n *polygons?***

- Answer:
  O(n^2)



Summary

- We need to make sure that we only draw visible surfaces when rendering scenes
- There are a number of techniques for doing this such as
  - Back face detection
  - Depth-buffer method
  - A-buffer method
  - Scan-line method
- Next time we will look at some more techniques and think about which techniques are suitable for which situations