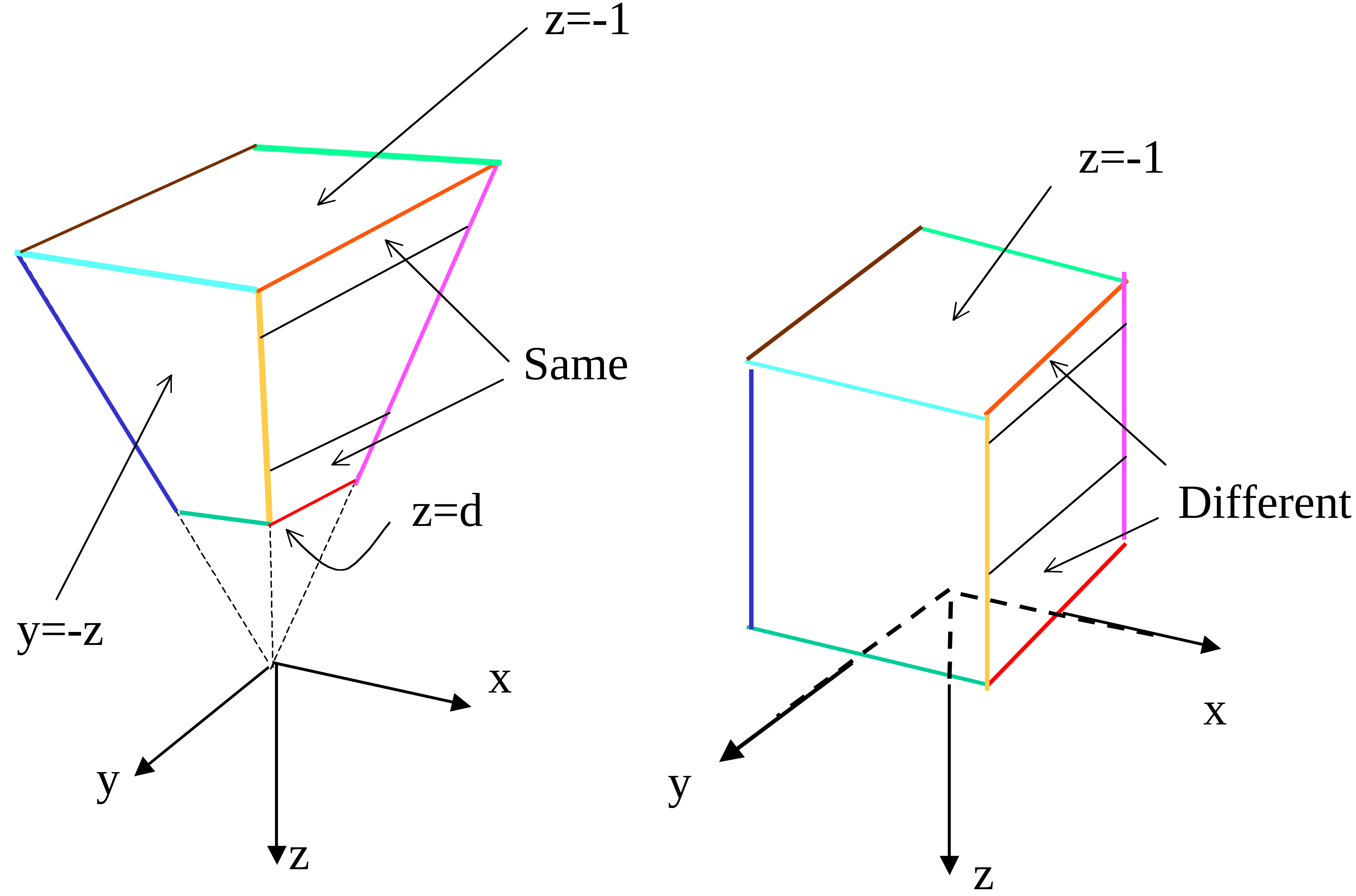
Mapping to standard parallel projection view volume (additional comments)

- Glossed over in the previous lecture--the mapping from [z_{min}, -1] to [0,-1] is non-linear. (Of course, there exists a linear mapping, but not if we want everything else to work out nicely in h.c.).
- So a change in depth of $\triangle$ D at the near plane maps to a larger depth difference in screen coordinates than the same $\triangle$ D at the far plane.
- But order is preserved (important!); the function is monotonic (proof?).
- And lines are still lines (proof?) and planes are still planes (important!).

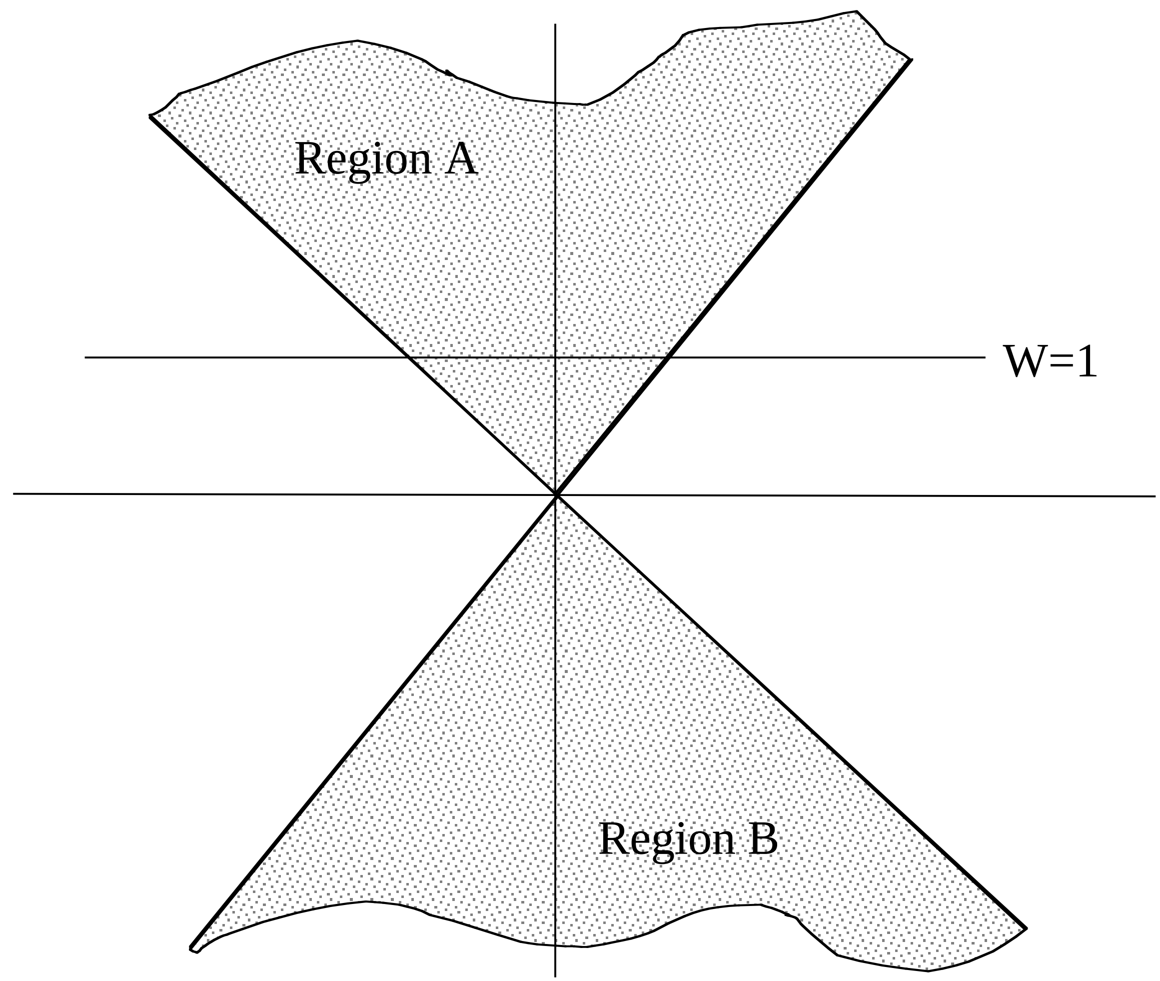
Transforming canonical frustum to box



Clipping in homogeneous coord.'s (extra comments)

The clipping volume in cross section

(Along the lines of Figure 6.44 in text–see §6.6.4 for further detail).



Clipping in homogeneous coord.'s (extra comments)

- If we know that W is positive (the case so far!), simply clip against region A
- If we are using the h.c. for additional deferred division, then W can be negative.
- If W is negative, then we use region B. The clipping can be done by negating the point, and clipping against A, due to the nature of A and B.
- Object has both positive and negative W?--see book for the case of a line.

Visibility (§13) - Intro

- Of these polygons, which are visible? (in front, etc.)
- Very large number of different algorithms known. Two main (rough) classes:
  - Object precision: computations that decompose polygons in world to solve
  - Image precision: computations at the pixel level
- Depth order in standard view box is same as depth order in 3D, so can work with the box.

- Essential issues:
  - must be capable of handling complex rendering databases.
  - in many complex worlds, few things are visible
  - efficiency - why render pixels many times?
  - accuracy - answer should be right, and behave well when the viewpoint moves
  - complexity - object precision visibility may generate many small pieces of polygon

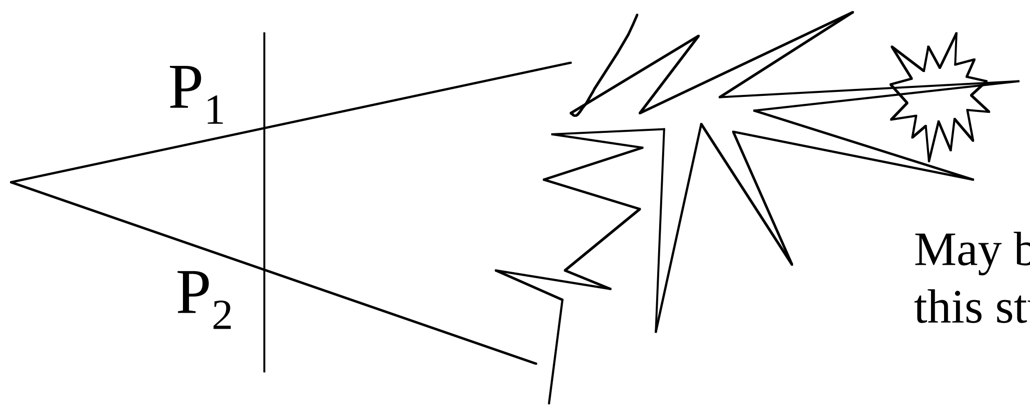
Image Precision

- Typically simpler algorithms (e.g., Z-buffer, ray cast)

- Pseudocode (conceptual!)
  - For each pixel
    - Determine he closest surface which intersects the projector
    - Draw the pixel the appropriate color

Image Precision

- "Image precision" means that we can save time not computing precise intersections of complicated objects



- But the algorithms are subject to aliasing problems, and the sampling needs to be redone when the view changes, even if only a simple window resize

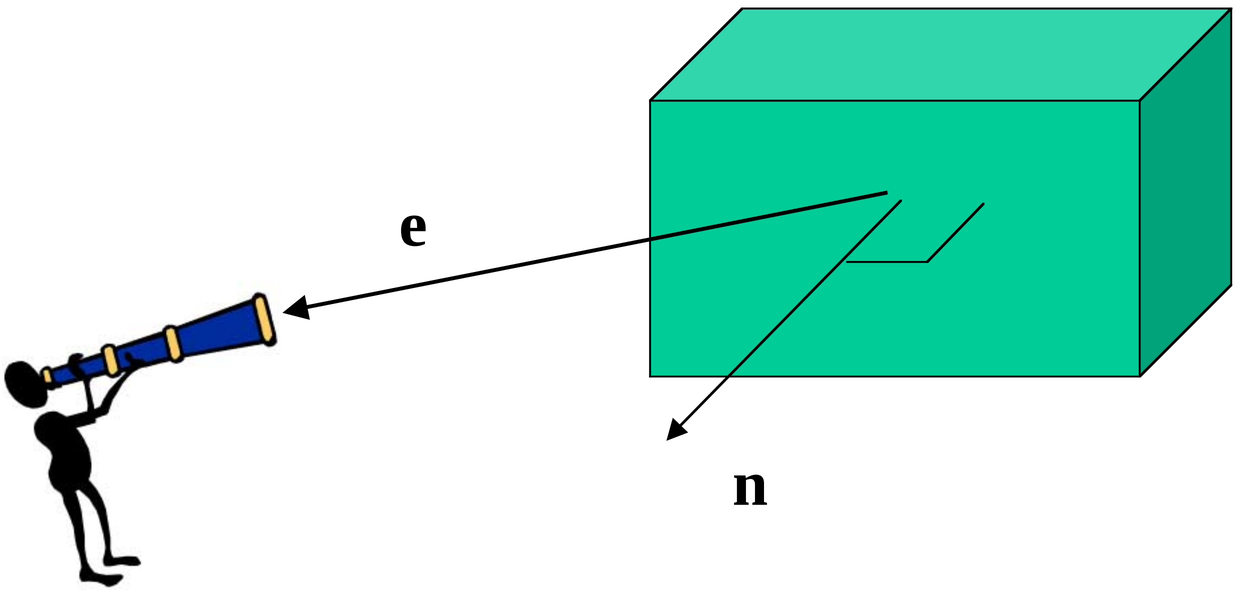
Object Precision

- The algorithms are typically more complex
- Pseudocode (conceptual!)
  - For each object
    - Determine which parts are viewed without obstruction by other parts of itself of other objects
    - Draw those parts the appropriate color

Visibility - Back Face Culling

- Simple, preliminary step, to reduce the amount of work.
- Polygons from solid objects have a front face and back face
- If the viewer sees the back face, then the plane can be culled.

Visibility - Back Face Culling



e is viewing direction (not to be confused with projector direction)

If **n**.**e** > 0, then display the plane

Question: How do we get **n**? (e.g., for the assignment)

Visibility - Back Face Culling

Question: How do we get **n**? (e.g., for the assignment)

Answer

When you read in the cube, you have to create the faces. Consider storing **n** here along with the face, and applying the required mappings to it.

Alternatively, store polygons so that the vertex order gives the sign of **n** by RHR.

To compute **n** from vertices, use cross product (but you don't necessarily need to do this for the faces of an axis aligned block).

Visibility - Back Face Culling

Question: In which coordinate frames can/should we do this?

Visibility - Back Face Culling

Question: In which coordinate frames can/should we do this?

Answer

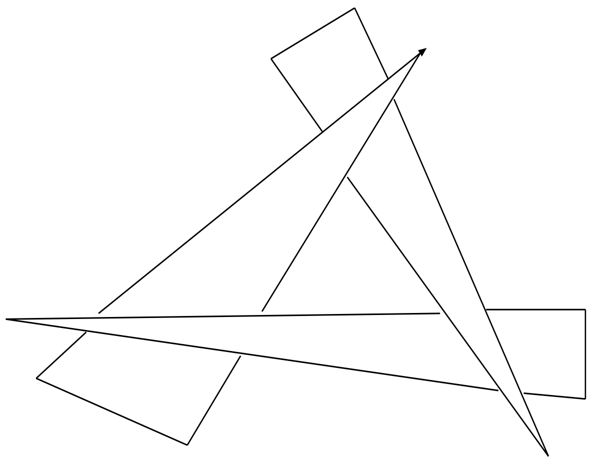
All of them. Most natural to do this in the standardized view box where perspective projection has become parallel projection.

Here, e=(0,01), so the test **n.e**>0 is especially easy (n_z>0).

Intuition: Does the plane normal have a component pointing in the view direction?

Visibility - painters algorithm

- Algorithm
  - Choose an order for the polygons based on some choice (e.g. depth to a point on the polygon)
  - Render the polygons in that order, deepest one first
- This renders nearer polygons over further.
- Works for some important geometries (2.5D - e.g. VLSI, mazes--but more efficient algorithms exist)
- Doesn’t work in this form for most geometries (see figure)



The Z - buffer

- For each pixel on screen, have a second memory location - called the z-buffer
- Set this buffer to a value corresponding to the furthest point
- As a polygon is filled in, compute the depth value of each pixel
  – if depth < z buffer depth, fill in pixel and new depth
  – else disregard
- Typical implementation: Compute Z while scan-converting. A ∂Z for every ∂X is easy to work out.

The Z - buffer

- Advantages:
  - simple; hardware implementation common
  - efficient z computations are easy.
- Disadvantages:
  - over renders - can be slow for very large collections of polygons - may end up scan converting many hidden objects
  - quantization errors can be annoying (not enough bits in the buffer)
  - doesn’t do transparency, filtering for anti-aliasing.

The A - buffer

- For transparent surfaces and filter anti-aliasing:
- Algorithm: filling buffer
  - at each pixel, maintain a pointer to a list of polygons sorted by depth.
  - When filling a pixel:
    - if polygon is opaque and covers pixel, insert into list, removing all polygons farther away
    - if polygon is opaque and only partially covers pixel, insert into list, but don't remove farther polygons
- Algorithm: rendering pixels
  - at each pixel, traverse buffer using brightness values in polygons to fill.
  - values are used either in transparency or for filtering
- Adv:
  - can do more than z-buffer
- Disadv:
  - over renders
  - quantization errors can be annoying