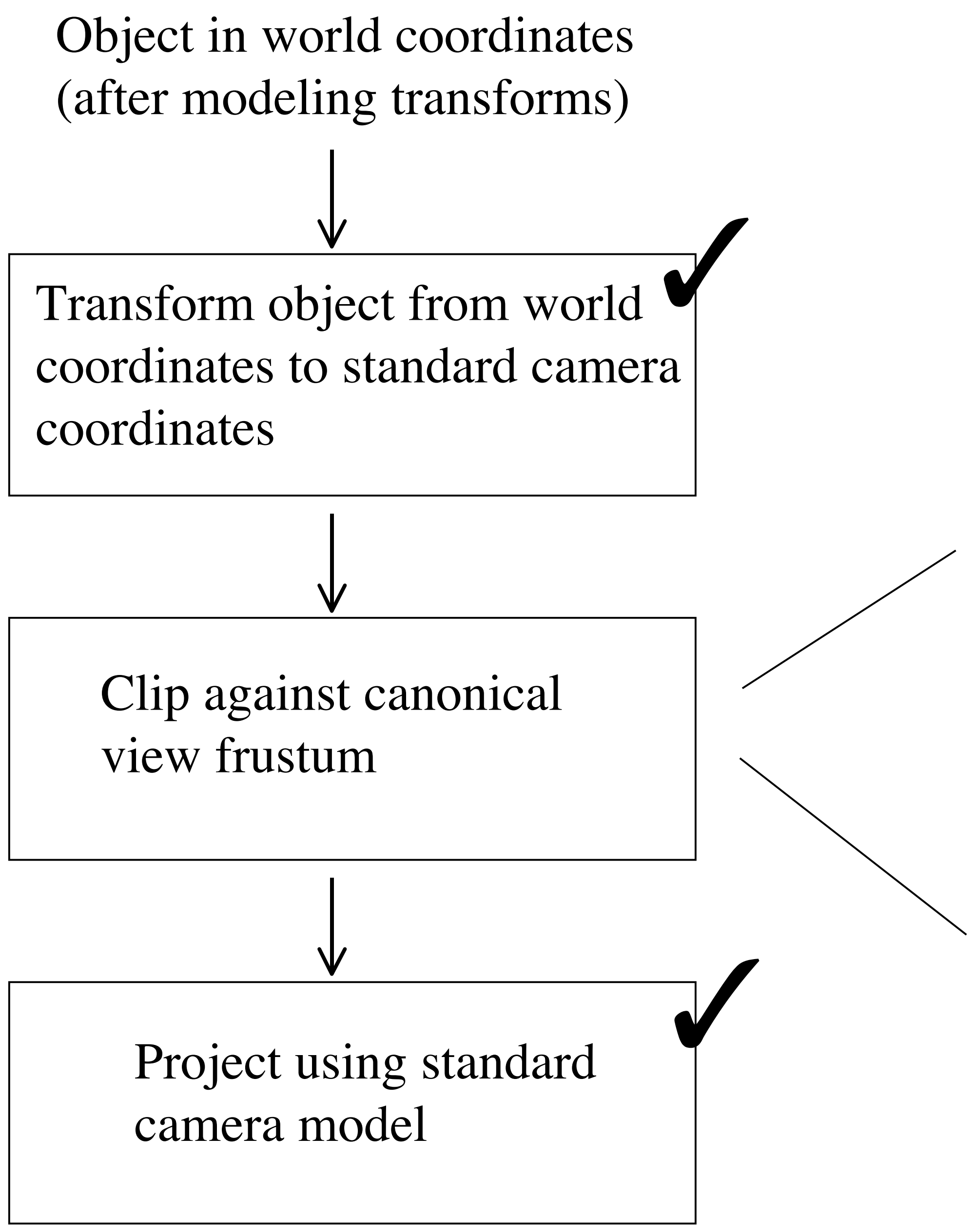

Object in world coordinates (after modeling transforms)
Transform object from world coordinates to standard camera coordinates ✓
Clip against canonical view frustum
Project using standard camera model ✓

Plan A: Clip against canonical frustum (relatively easy—we chose the canonical frustum so that it would be easy!)

Plan B: Be even more clever. Further transform to cube and clip in homogenous coordinates.

Plan A: Clipping against the canonical frustum

2D algorithms are easily extended (see slides from previous lectures). The clipping boundaries are now.

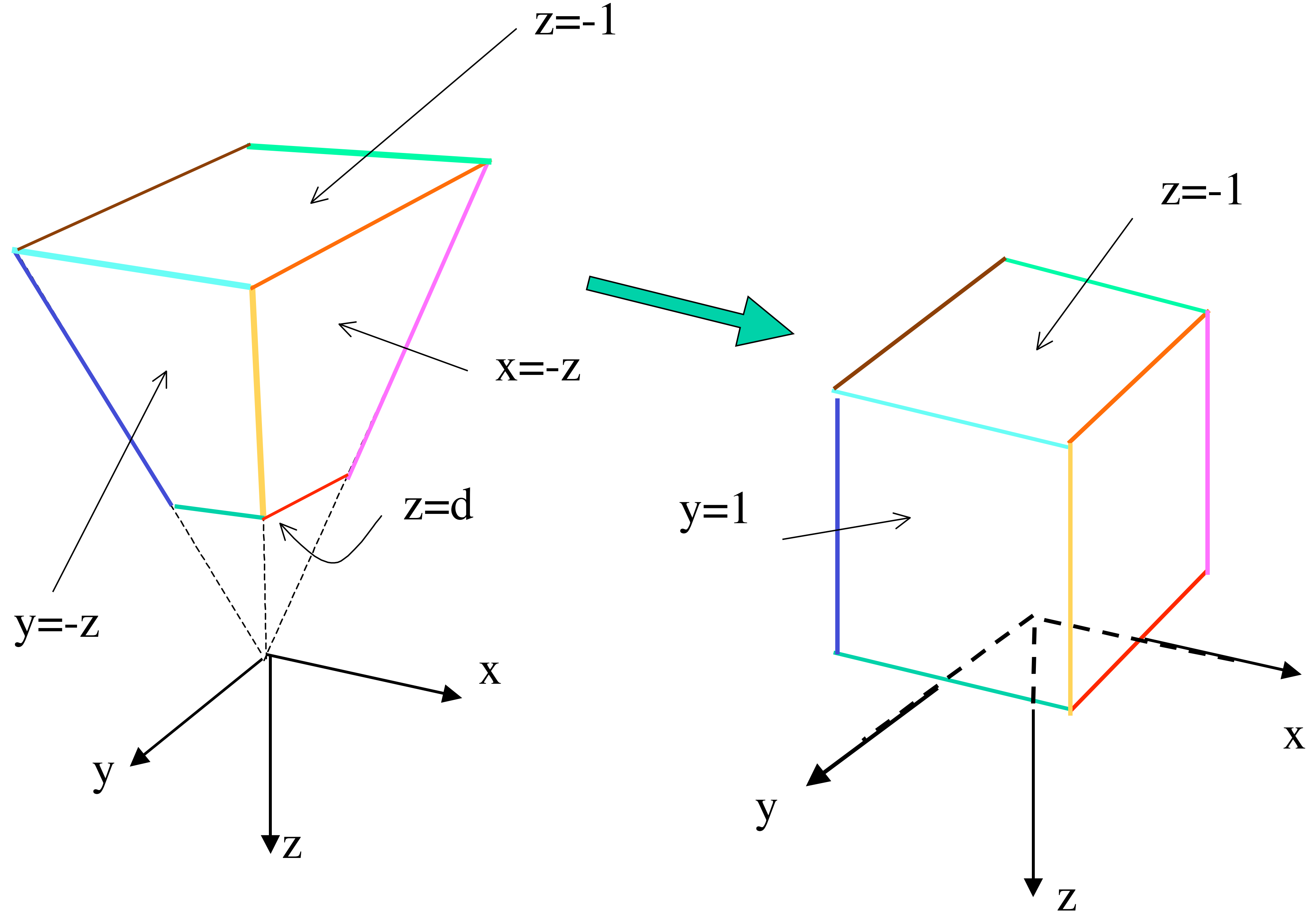
$y>-z \quad y<z \quad x>-z \quad x<z \quad z<-1 \quad z>z_{min}$

$(z_{min} = (f-F)/(B-f))$

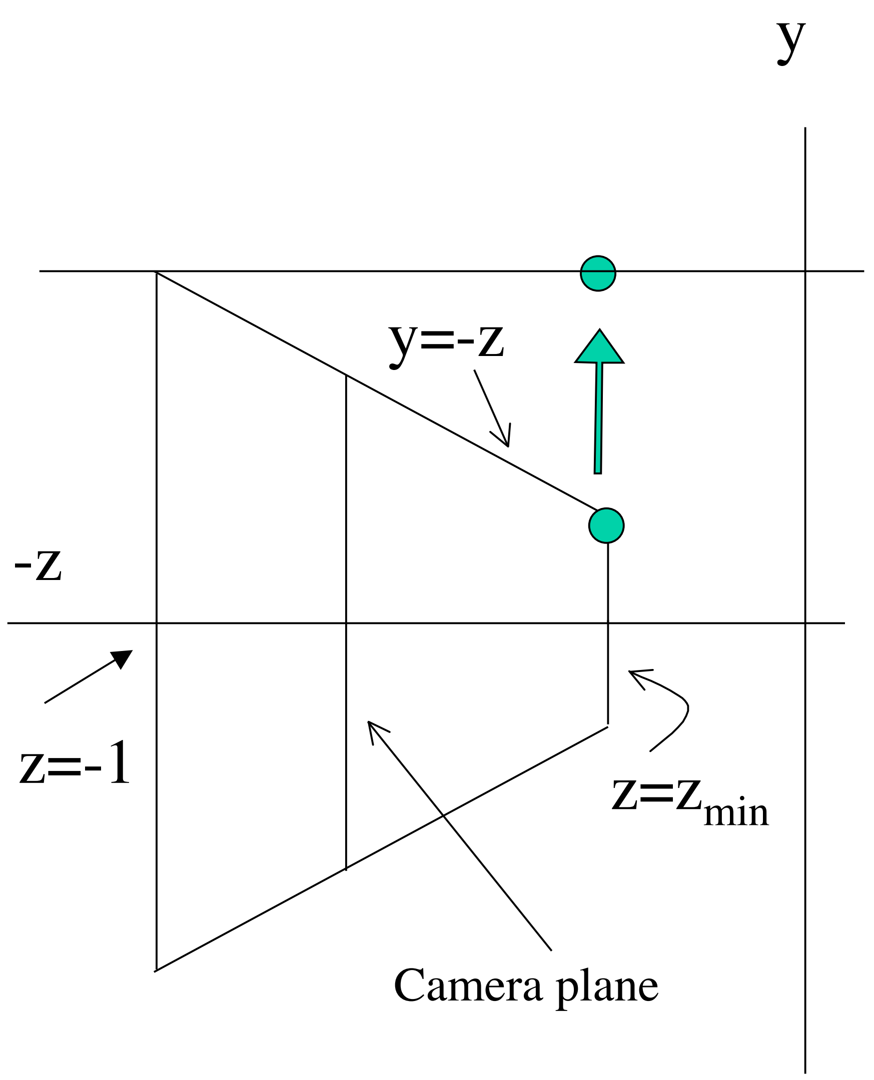
Plan B: Clipping in homogenous coords

- For any camera, can turn the view frustrum into a regular parallelepiped (box). We will use the box bounded by $x = \pm 1$, $y = \pm 1$, $z = -1$, and $z = 0$.
- Advantages
  - Simplified clipping in homogenous coordinates
  - Extends to cases where we use homogenous coordinates to represent additional information (and w could be negative).
  - Can simplify visibility algorithms.
- Approach: clever use of homogenous coordinates

Transforming canonical frustum to box



Transforming canonical frustum to box



On top, y—> 1, so scaling is (1/y) Recall that y=-z there.

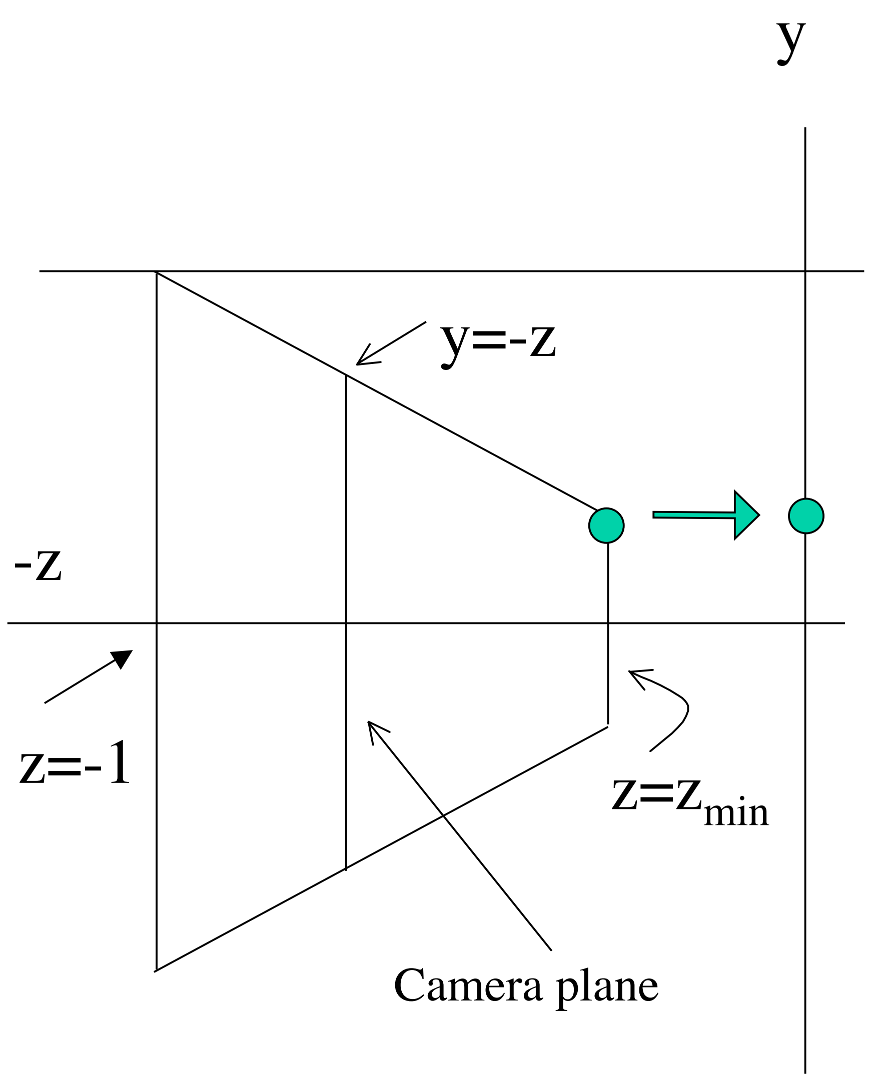
On bottom, y—> -1 so scaling is (-1/y). Recall that y=z there.

So scaling is y'= y/(-z)

Similarly, x' = x/(-z)

Transformation is **non-linear**, but in h.c., we can make w = (-z).

Transforming canonical frustum to box



For z, we translate near plane to origin. But now box is too small. Specifically it has z dimension ($1 + z_{min}$) (recall z_{min} is negative)

So we have an extra scale factor $1 / (1 + z_{min})$ and thus
$z' = (z - z_{min}) / (1 + z_{min})$

But we want x and y to work nicely in h.c., with w=-z, so we end up with

$z' = ((z - z_{min}) / (1 + z_{min}))/(-z)$

(Thus in our box, depth also transforms **non-linearly**)

In h.c.,

$x \Rightarrow x$

$y \Rightarrow y$

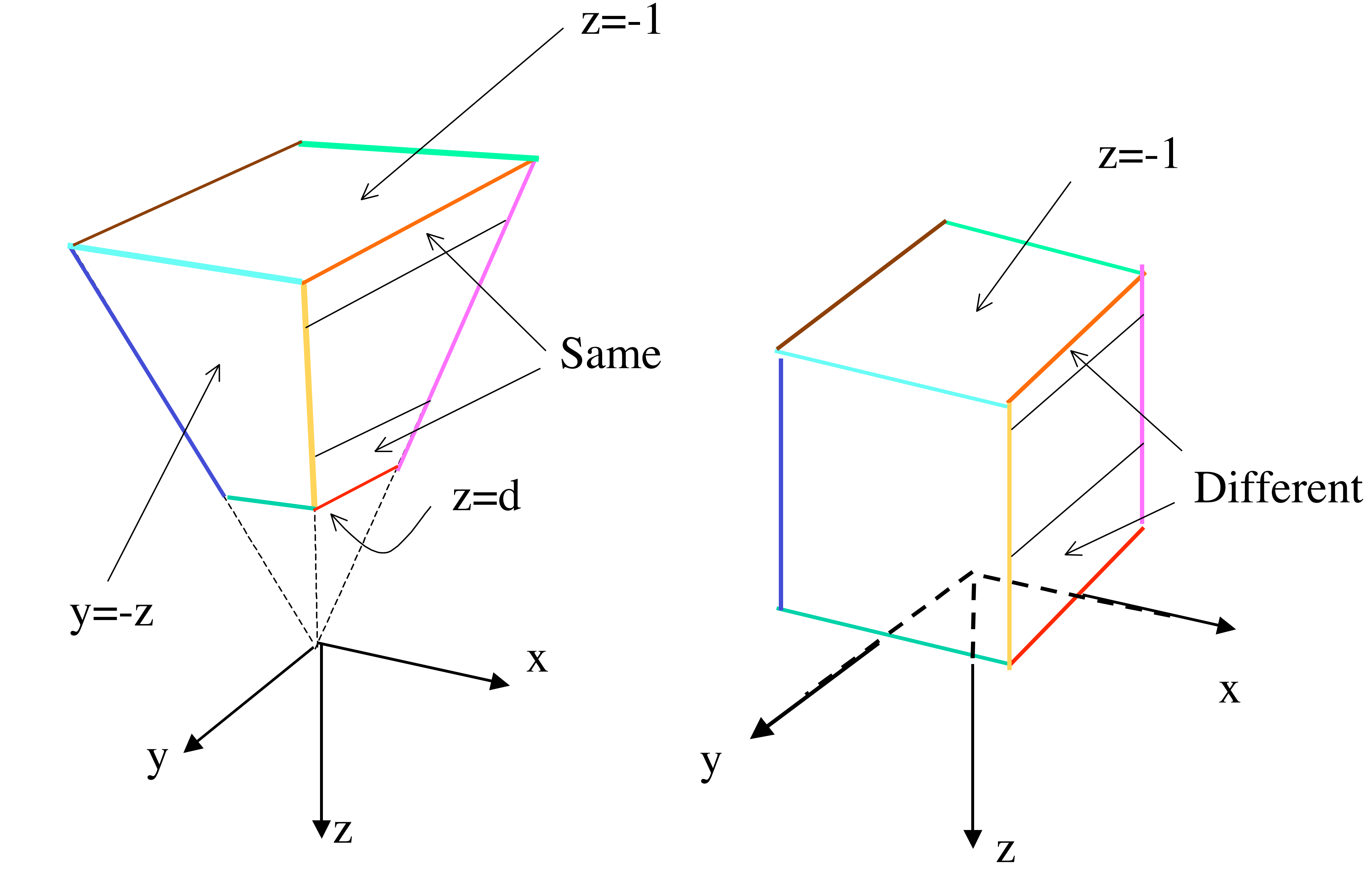
$z \Rightarrow (z - z_{min}) / (1 + z_{min})$

$1 \Rightarrow -z$

So, the matrix is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+z_{\min}} & \frac{-z_{\min}}{1+z_{\min}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Transforming canonical frustum to box



Mapping to standard view volume (additional comments)

- 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!).

Clipping in homogeneous coordinates

- We have a cube, but it is not in homogeneous coordinates, so we must divide if we want to take advantage of this particularly nice clipping situation.
- However, dividing before clipping is inefficient if many points are excluded, so we generally clip in homogeneous coordinates.

Clipping in homogeneous coord.'s

- Write h.c.'s in caps, ordinary coords in lowercase.
- Consider case of clipping stuff where x>1, x<-1
- Rearrange clipping inequalities:

$$\left(\frac{X}{W}\right) > 1 \qquad \left(\frac{X}{W}\right) < -1$$

becomes

$$X > W, \quad X < -W, \quad W > 0$$

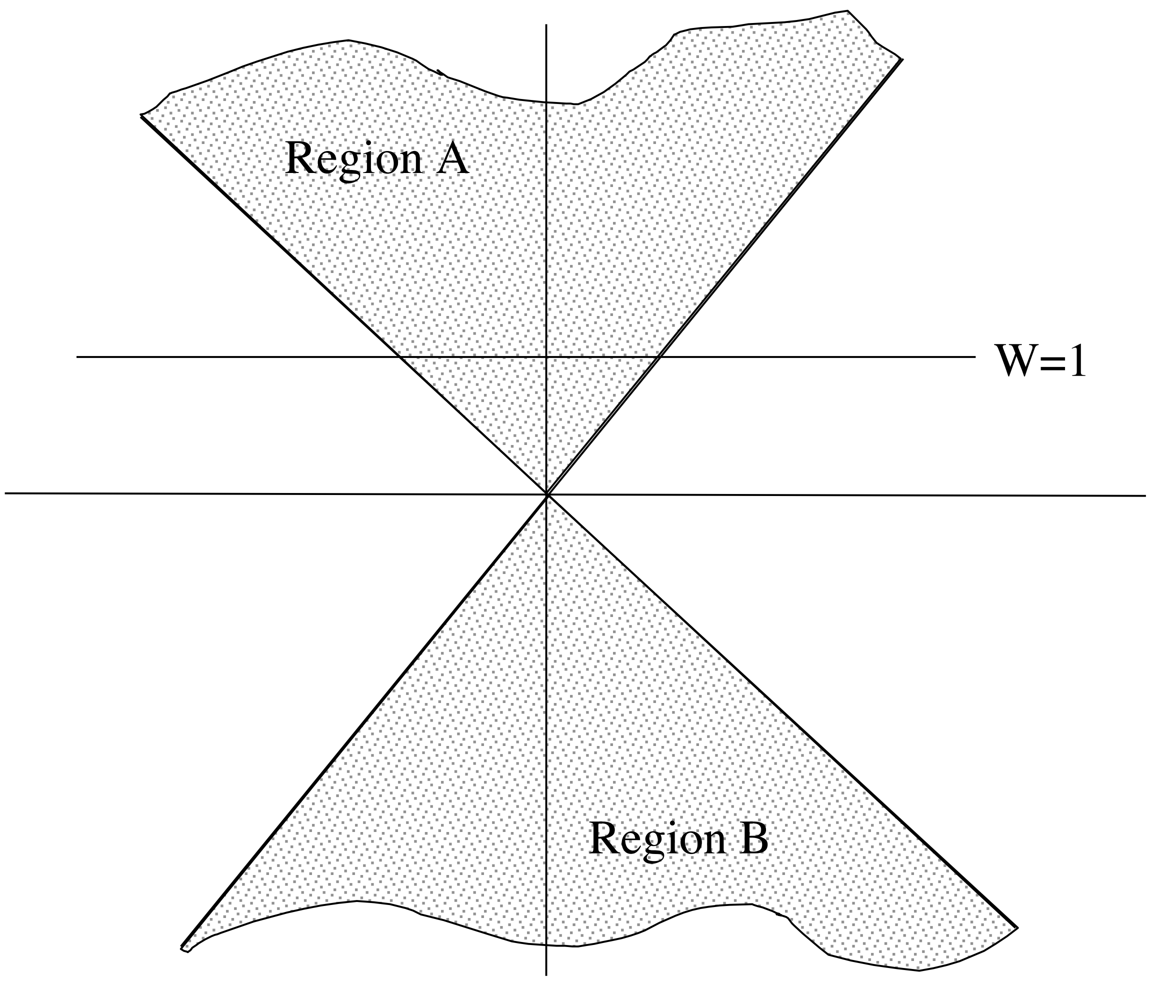
AND

$$X < W, \quad X > -W, \quad W < 0$$

(So far W is positive, but negatives occur if we further overload the use of h.c.'s)

Clipping in homogeneous coord.'s

The clipping volume in cross section



Clipping in homogeneous coord.’s

- 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.
- Case where object has both positive and negative W is a little more complex.
- Notice that the actual clipping computations are not that different from the case in Plan A---no free lunch!

Further comments on standard view box

It may appear that mapping from canonical view frustum to standard view box loses information that might be stored with W because it gets shifted to Z, which does not affect where things end up in 2D.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+z_{\min}} & \frac{-z_{\min}}{1+z_{\min}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

However, the whole trick of putting deferred division into W is to replace a point in h.c.'s with one which will project to the same place. It does not matter if you project (1,2,3) or (2,4,6) onto z=1 except when you worry about **visibility** (what is in front) and this is encoded in Z.

Reminder of the last step

In both plans we need to project into 2D.

If we are working in the canonical view space, then we project using the standard camera model (easy) and divide

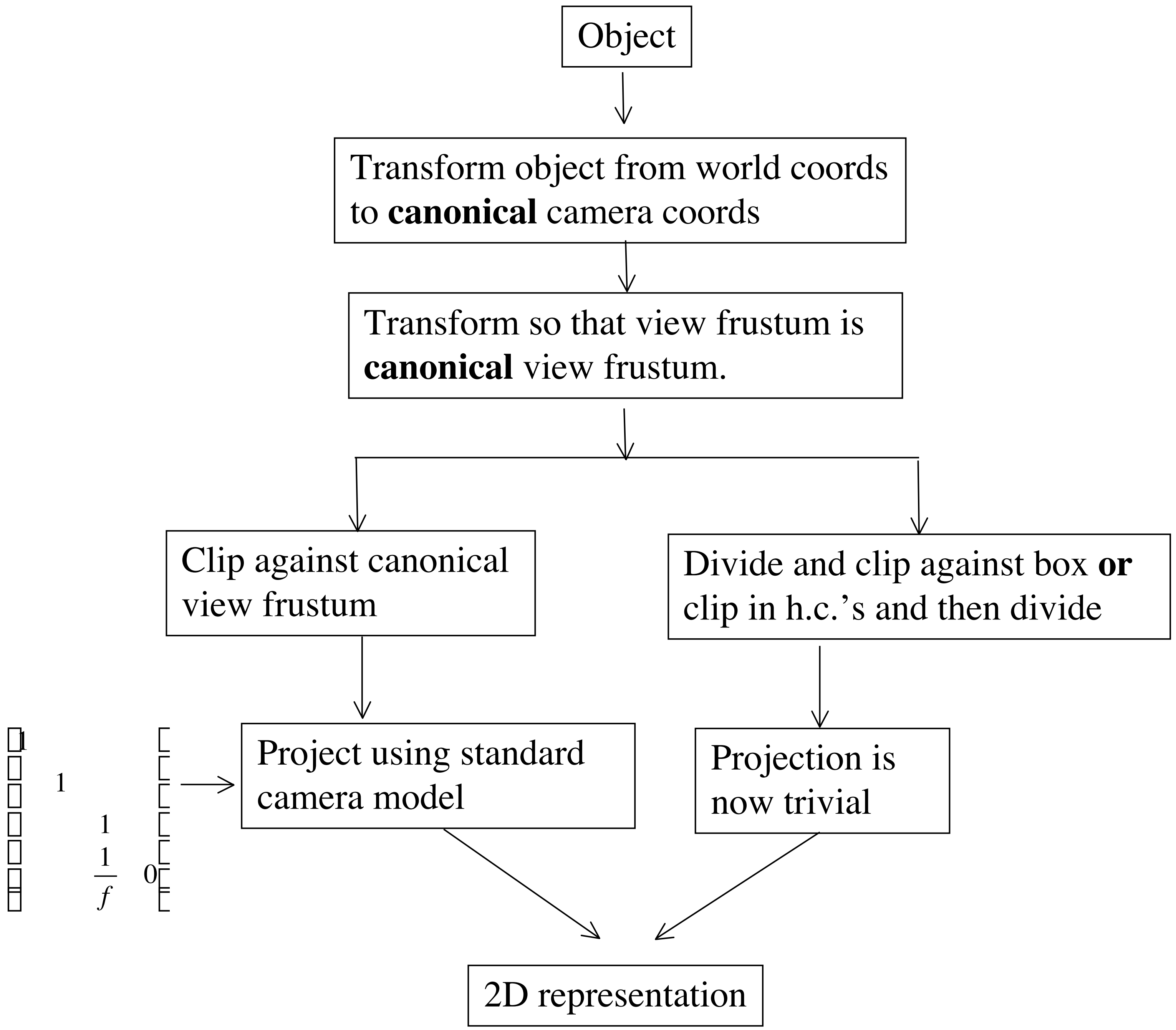
Recall that the matrix for the standard camera model using homogeneous coordinates is:

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & \frac{1}{f} & 0 \end{bmatrix}$$

Reminder of the last step

If we are working in homogenous coordinates, then we divide and then projection is even easier (ignore z coordinate).

The mapping to the box—which was complete once the division was done—implicitly did the perspective projection—essentially we transformed the world so that orthographic projections holds.


Object
Transform object from world coords to **canonical** camera coords
Transform so that view frustum is **canonical** view frustum.
Clip against canonical view frustum
Divide and clip against box **or** clip in h.c.'s and then divide
1
1
1
$\frac{1}{f}$
0
Project using standard camera model
Projection is now trivial
2D representation

Visibility

- 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.

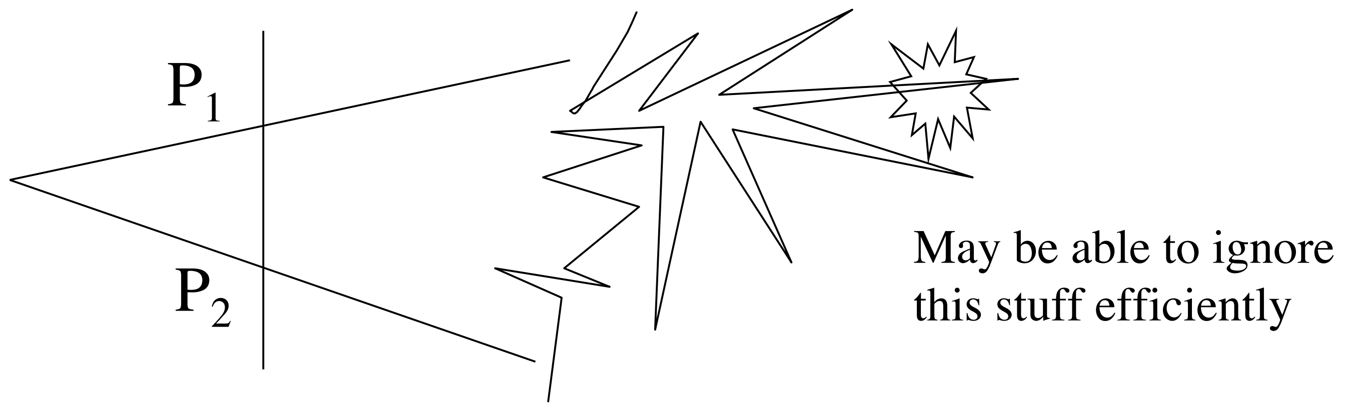
- Essential issues:
  - must be capable of handling complex rendering databases.
  - in many complex worlds, few things are visible
  - efficiency - don’t render pixels many times.
  - accuracy - answer should be right, and behave well when the viewpoint moves
  - aliasing

Image Precision

- Typically simpler algorithms (e.g., Z-buffer, ray cast)
- Pseudocode (conceptual!)
  - For each pixel
    - Determine the 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

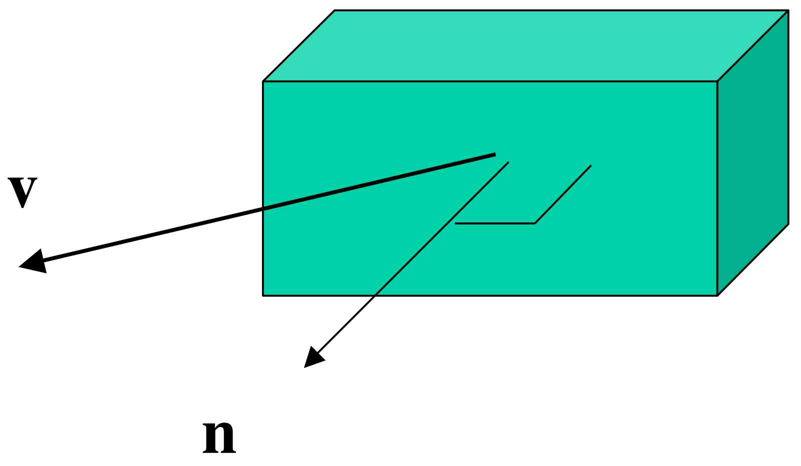
Object Precision

- 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



v is direction from a point on the plane to the center of projection (the eye).

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

Note that we are calculating which side of the plane the eye is on.

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 render the parallelepiped, you have to create the faces which are sequences of vertices.

To compute **n** from vertices, use cross product.

You need to store vertices consistently so that you can get the sign of n. Consider storing them so that you can get the sign of **n** by RHR.

Depending on the situation you may find it easier to compute **n** early on, and transform it, or recompute it from transformed vertices.

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. It is very natural to do this in the standardized view box where perspective projection has become parallel projection.

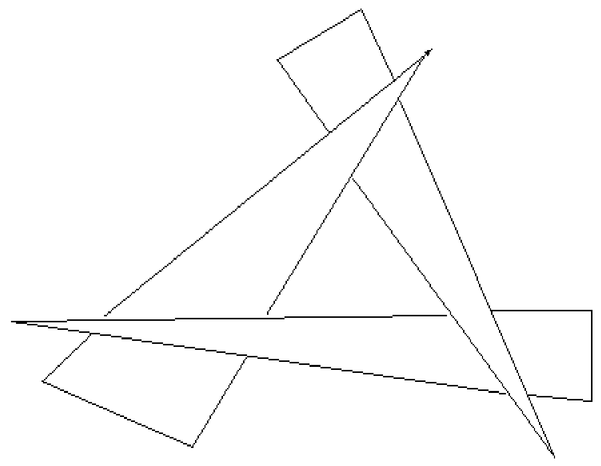
Here, **e**=(0,01) (why?), so the test **n.e**>0 is especially easy (n_z>0).

But be careful with the transformation which lead to **n**!

Also, an efficiency argument can be made for culling before division.

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.
  - ok with lots of surfaces (if there are lots, they tend to be small, and not much difference to this algorithm)
- 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, or filtering for anti-aliasing.