# Further comments on standard view box

Mapping from canonical view frustum to standard view box appears to lose any information that might be stored with W originally (e.g. as in (x/w,y/w,z/w)).

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

However, if we have done any translations (as in the mapping to the standard view box), then w lives on in the first three components.

The fact that w does not necessarily appear in the final w makes sense. If we were already in the canonical view frustum, then w only affects depth (i.e., the value of the z' which is $(z - z_{min}) / (1 + z_{min})$

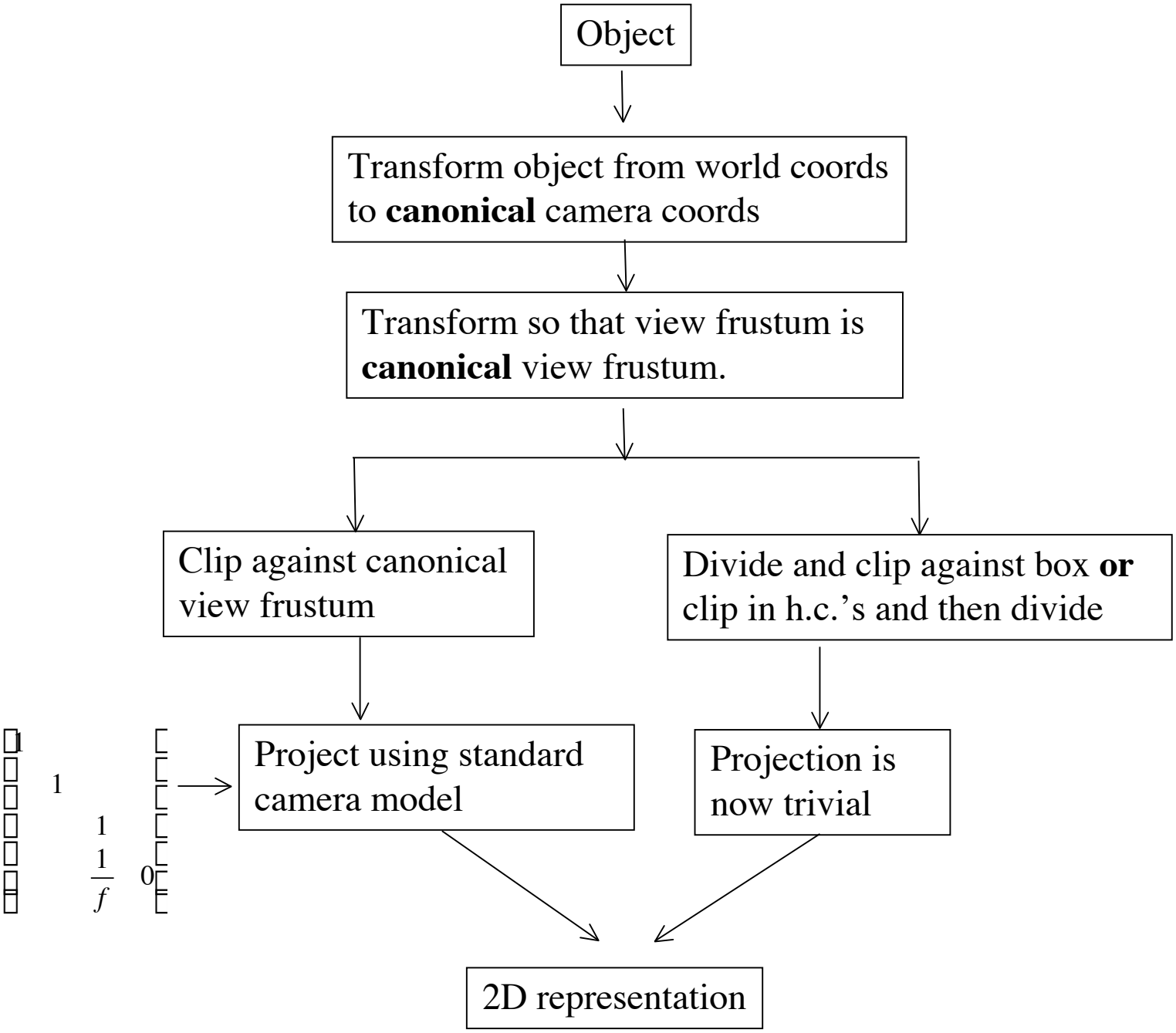
# Reminder of the last step

In both plan approaches 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

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

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


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
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.

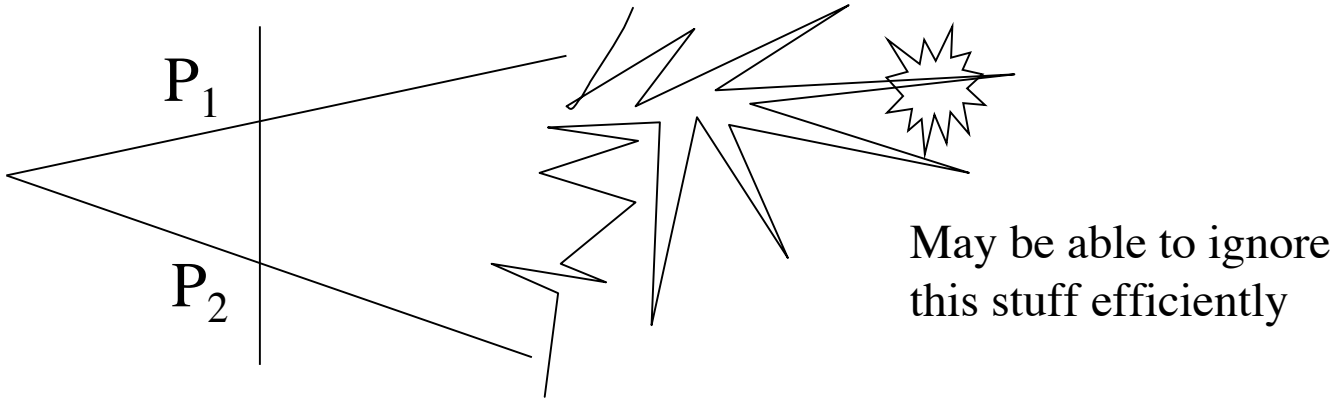
- Essential issues:
  - must be capable of handling complex rendering databases.
  - in many complex worlds, few things are visible
  - efficiency - don't render pixels many times.
  - accuracy - answer should be right, and behave well when the viewpoint moves
  - complexity - object precision visibility may generate many small pieces of polygon

# Image Precision

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

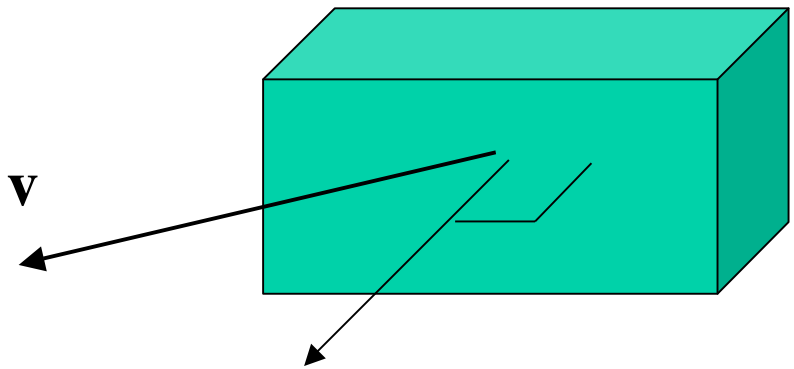
# Object Precision

- 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 $\mathbf{n}.\mathbf{v} > 0$, 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 read in the parallelepiped, 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?

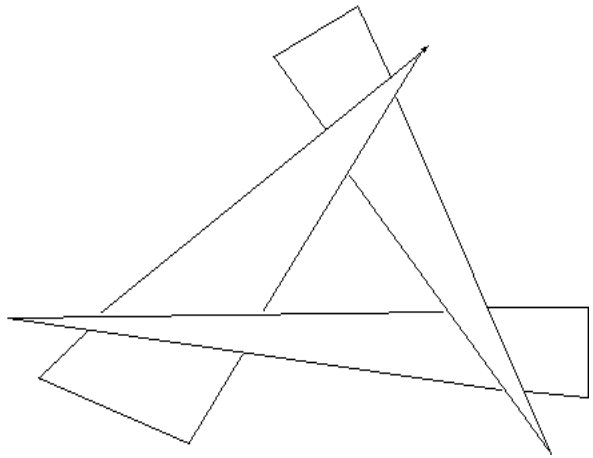
Answer

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

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

# 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 $\partial Z$ for every $\partial X$ 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.

# The A - buffer

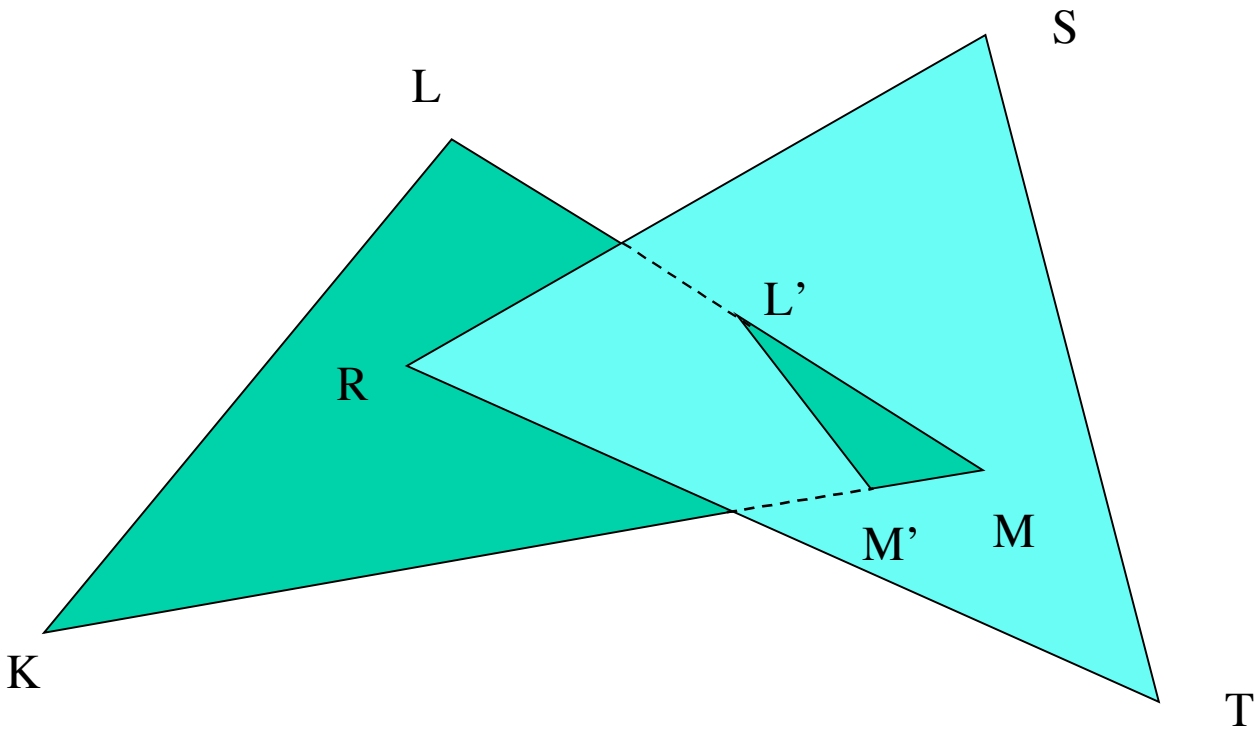
- For transparent surfaces and filter anti-aliasing:
- Algorithm: filling buffer
  - at each pixel, maintain a pointer to a list of polygons sorted by depth.
  - When filling a pixel:
    - if polygon is opaque and covers pixel, insert into list, removing all polygons farther away
    - if polygon is 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 for aliasing

# Scan line algorithm

- Assume polygons do not intersect one another.
- Observation: on any given scan line, the visible polygon can change only at an edge.
- Algorithm:
  - fill all polygons simultaneously at each scan line, have all edges that cross scan line in AEL
  - keep record of current depth at current pixel - use to decide which is in front in filling span

# Scan line algorithm

- To deal with penetrating polygons, split them up



# Scan line algorithm

- Advantages:
  - potentially fewer quantization errors (typically more bits available for depth, but this depends)
  - filter anti-aliasing can be made to work.
- Disadvantages:
  - invisible polygons clog AEL, ET (can get expensive for complex scenes).

# Depth sorting

- Sort in order of decreasing depth
- Render in sorted order
- Rendering:
  - for surface S with greatest depth
    - if no depth overlaps, render (like painter's algorithm)
    - if depth overlaps, test for problem overlap in image plane
    - if S, S' overlap in depth and in image plane, swap and try again
    - if S, S' have been swapped already, split and reinsert

- Testing image plane problem overlaps (test get increasingly expensive):
  - xy bounding boxes do not intersect
  - *or* S is behind the plane of S'
  - *or* S' is in front of the plane of S
  - *or* S and S' do not intersect
- Advantages:
  - filter anti-aliasing works fine
  - no depth quantization error
  - works well if not too much depth overlap (rarely get to expensive cases)
- Disadvantages:
  - gets expensive with lots of depth overlap (over-renders)