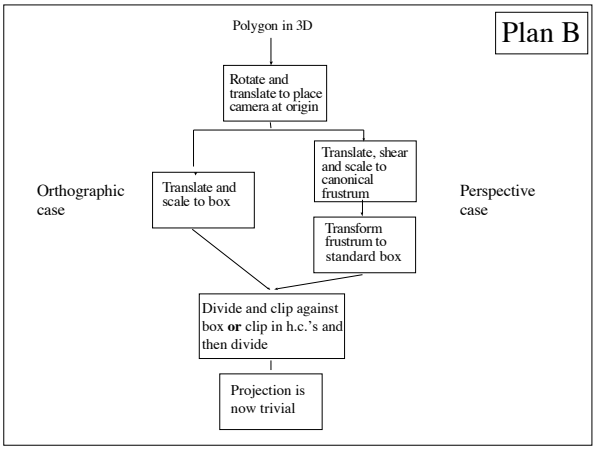
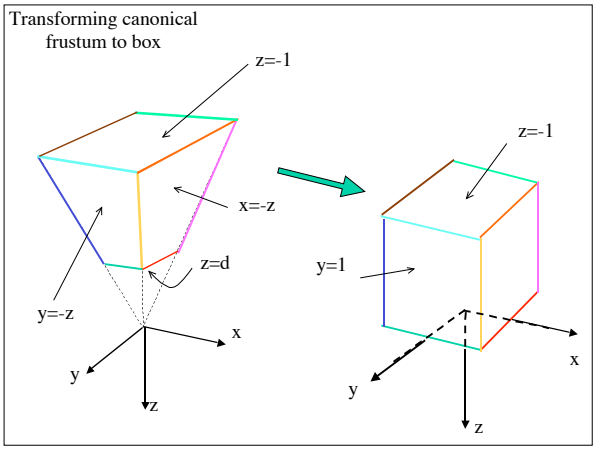




Clipping in homogeneous coordinates

- We have a cube in (x,y,z), but it is **not** a cube in homogeneous coordinates, so we must divide if we want to take advantage of this particularly nice clipping situation.
- However, dividing before clipping might be inefficient if many points are excluded, so we often clip in homogeneous coordinates.

Clipping in homogeneous coord.'s

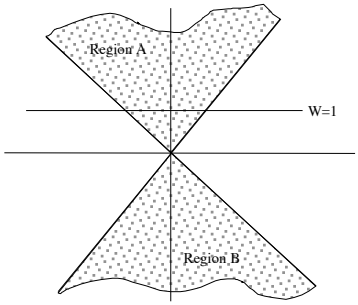
- Write h.c.'s in caps, ordinary coords in lowercase.
- Consider case of clipping stuff where x>1, x<-1
- Rearrange clipping inequalities:

$$\left|\frac{X}{W}\right| > 1 \quad \text{becomes} \quad \begin{array}{l} X > W, \\ X < -W, \\ W > 0 \end{array} \quad \text{AND} \quad \begin{array}{l} X < W, \\ X > -W, \\ W < 0 \end{array}$$

(So far W has been 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!

Reminder of the last steps

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

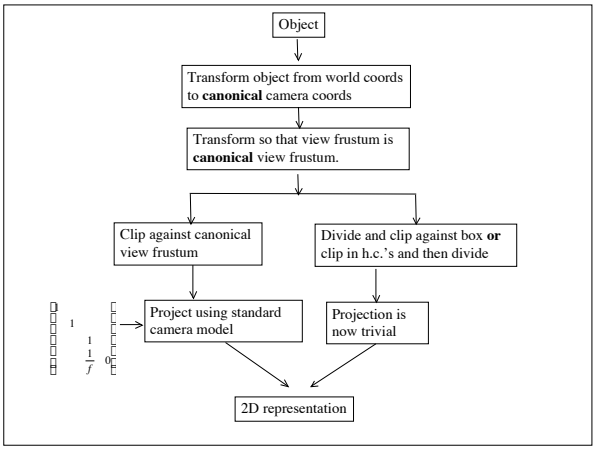
Recall that the matrix for the standard camera model using homogeneous coordinates is:

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

Reminder of the last steps

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.



Reminder of the last steps

Finally, we may need to do additional 2D transformations.

In the canonical frustum case, our (x,y) coordinates are relative to (-f',f'). They need to be mapped to the viewport (possibly implicitly by the graphics package).

In the canonical box case, our (x,y) coordinates are relative to (-1,1). They need to be mapped to the viewport (possibly implicitly by the graphics package).

Visibility

H&B chapter 9 (similar to notes)

- Of these polygons, which are visible? (in front, etc.)
- Very large number of different algorithms known. Two main (very rough) classes:
  - Object precision: computations that decompose polygons in world coordinates
  - 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

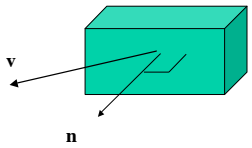
Object Precision

- 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 $\mathbf{n}.\mathbf{v} > 0$, 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 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 perhaps more natural to attempt do this in the standardized view box where perspective projection has become parallel projection.

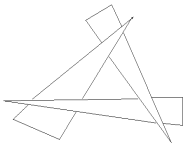
Here, **e**=(0,01) (why?), so the test **n.e**>0 is especially easy (n_z>0).

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 help with transparency, or filtering for anti-aliasing.

The A - buffer

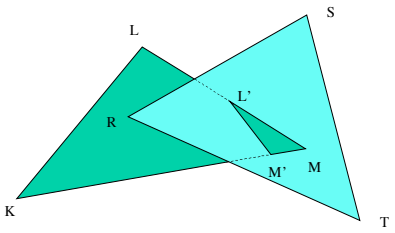
- For transparent surfaces and filter anti-aliasing:
- Algorithm: filling buffer
  - at each pixel, maintain a pointer to a list of polygons sorted by depth.
  - when filling a pixel:
    - if polygon is opaque and covers pixel, insert into list, removing all polygons farther away
    - if polygon is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

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).