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

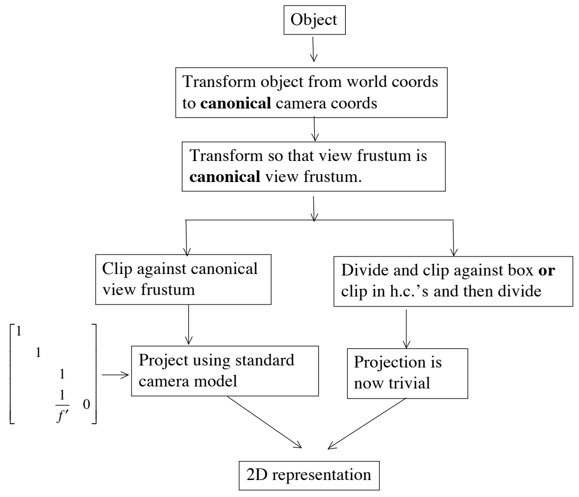
Recall that the matrix for the standard camera model using homogeneous coordinates is:

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

Reminder of the last steps

If we are working in homogenous coordinates, then we first 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
$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & \frac{1}{f'} & 0 \end{bmatrix}$
Project using standard camera model
Projection is now trivial
2D representation

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).

For the homeworks

For homework 3 you can use either the canonical frustum (A) directly or carry on, and map it to the canonical box (B).

If you take this approach, consider doing (A) as a first step.

For homework 4 you will need to use (B).

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.

Visibility

H&B chapter 9 (similar to notes)

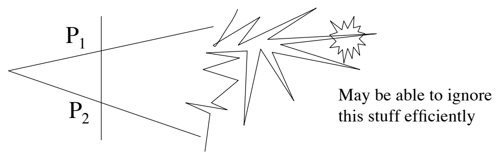
- Essential issues:
  - must be capable of handling complex rendering databases.
  - in many complex worlds, few things are visible
  - efficiency - don't render pixels many times.
  - accuracy - answer should be right, and behave well when the viewpoint moves
  - aliasing

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

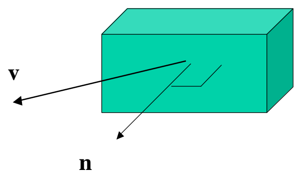
Object Precision

- 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.
- Why would the viewer see the back face?
  - Because they are on the back side of the plane of the polygon.

Visibility - Back Face Culling



v is direction from any 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
1) compute **n** early on, and transform it using the correct formula
2) 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.

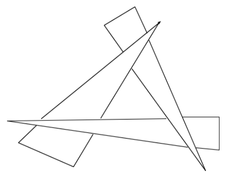
Here, **e**=(0,01) (why?), so the test **n.e**>0 is especially easy (n_z>0).

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
- Initialize this buffer to a value corresponding to the furthest point possible.
- 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 can be 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.