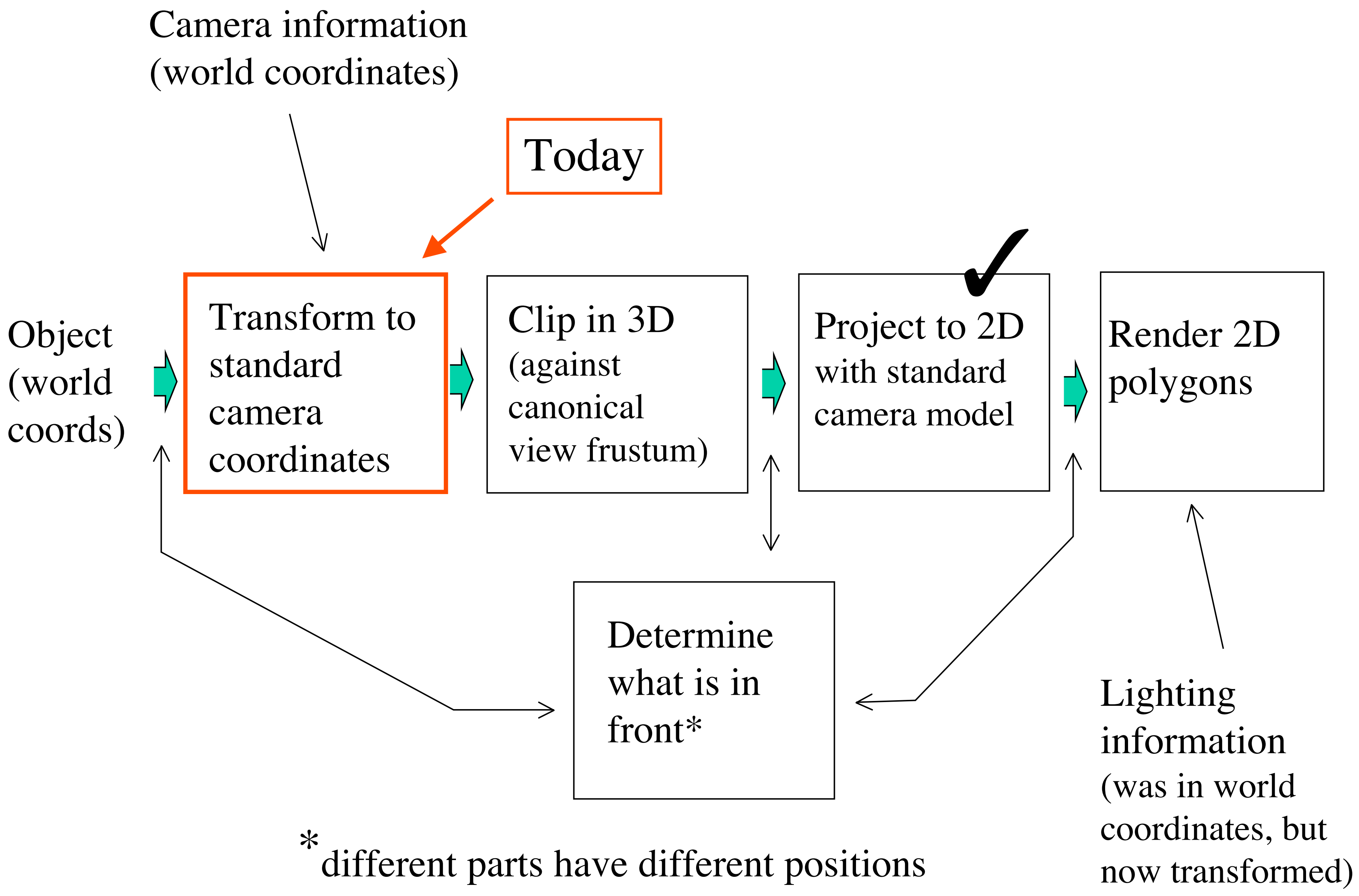
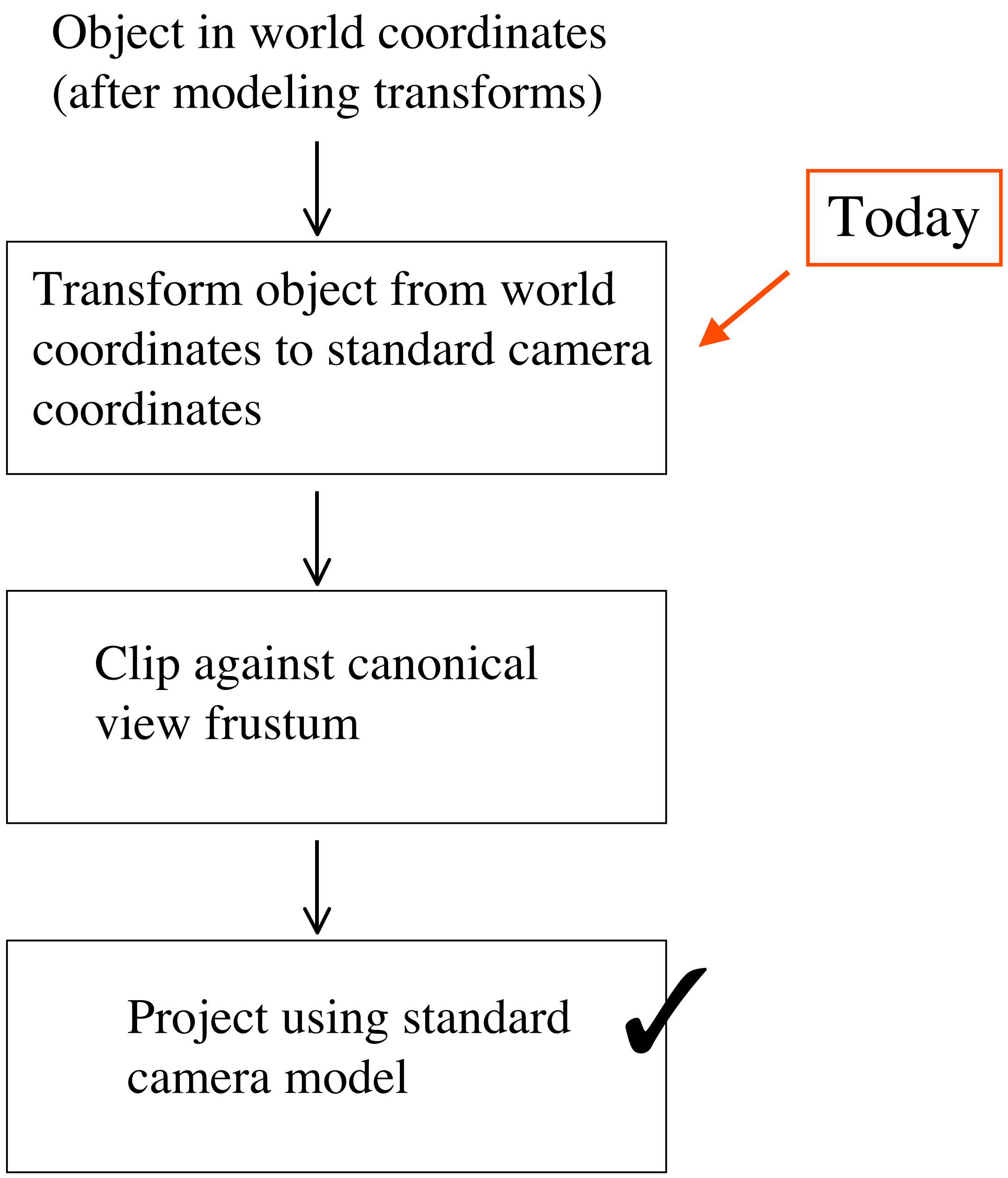
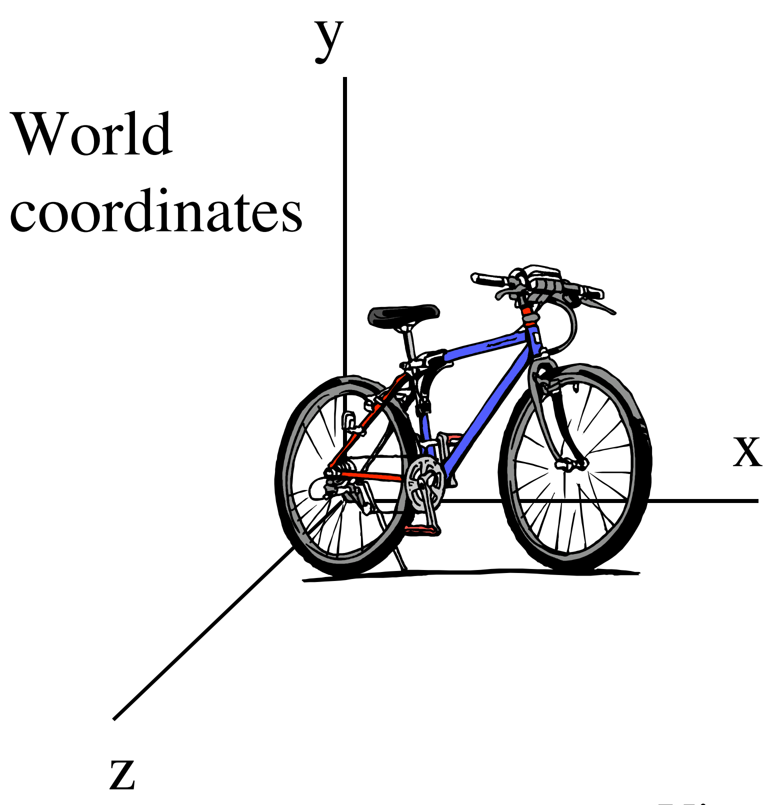
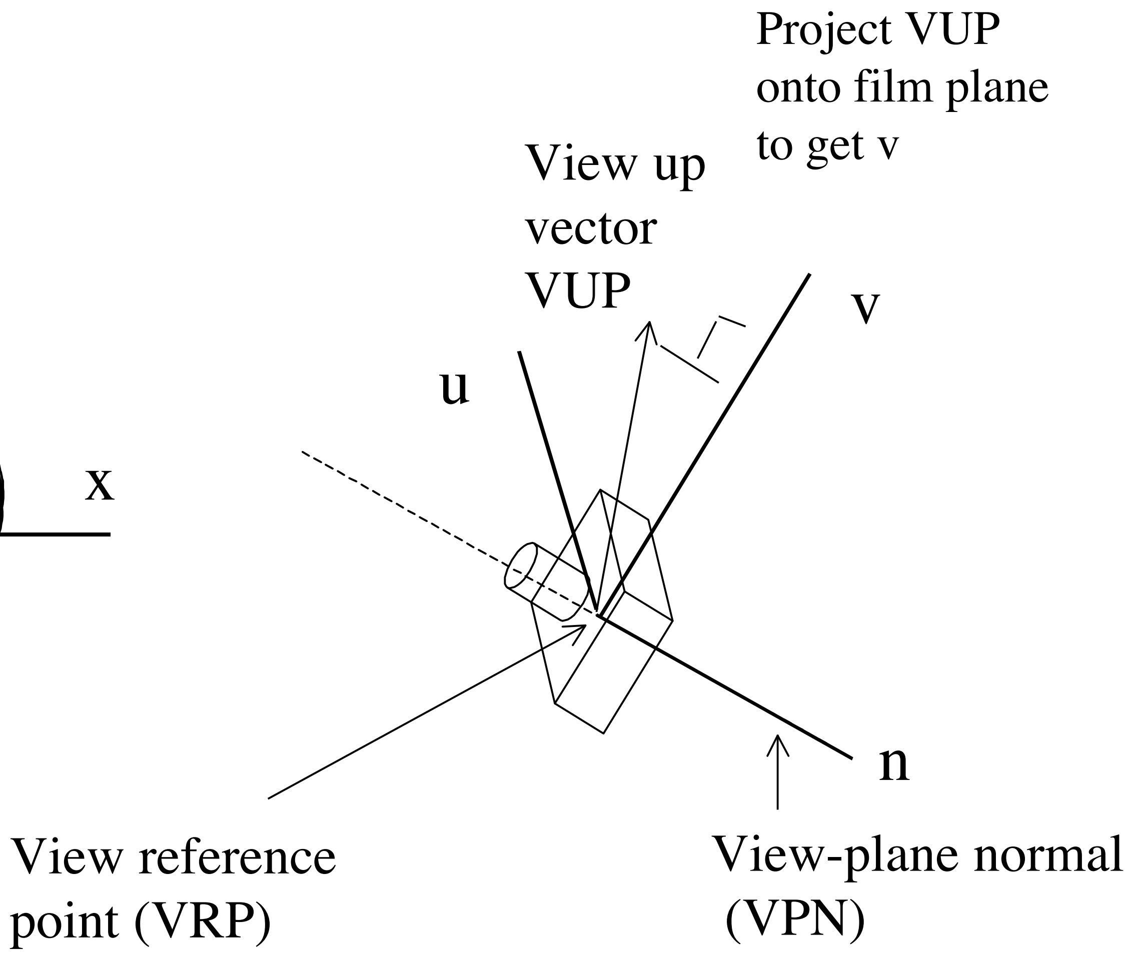

Camera information (world coordinates)
Today
Object (world coords)
Transform to standard camera coordinates
Clip in 3D (against canonical view frustum)
Project to 2D with standard camera model
✓
Render 2D polygons
Determine what is in front*
Lighting information (was in world coordinates, but now transformed)

*different parts have different positions in the pipeline, depending on strategy


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

Specifying a camera





Project VUP onto film plane to get v
View up vector VUP
v
u
n
View reference point (VRP)
View-plane normal (VPN)

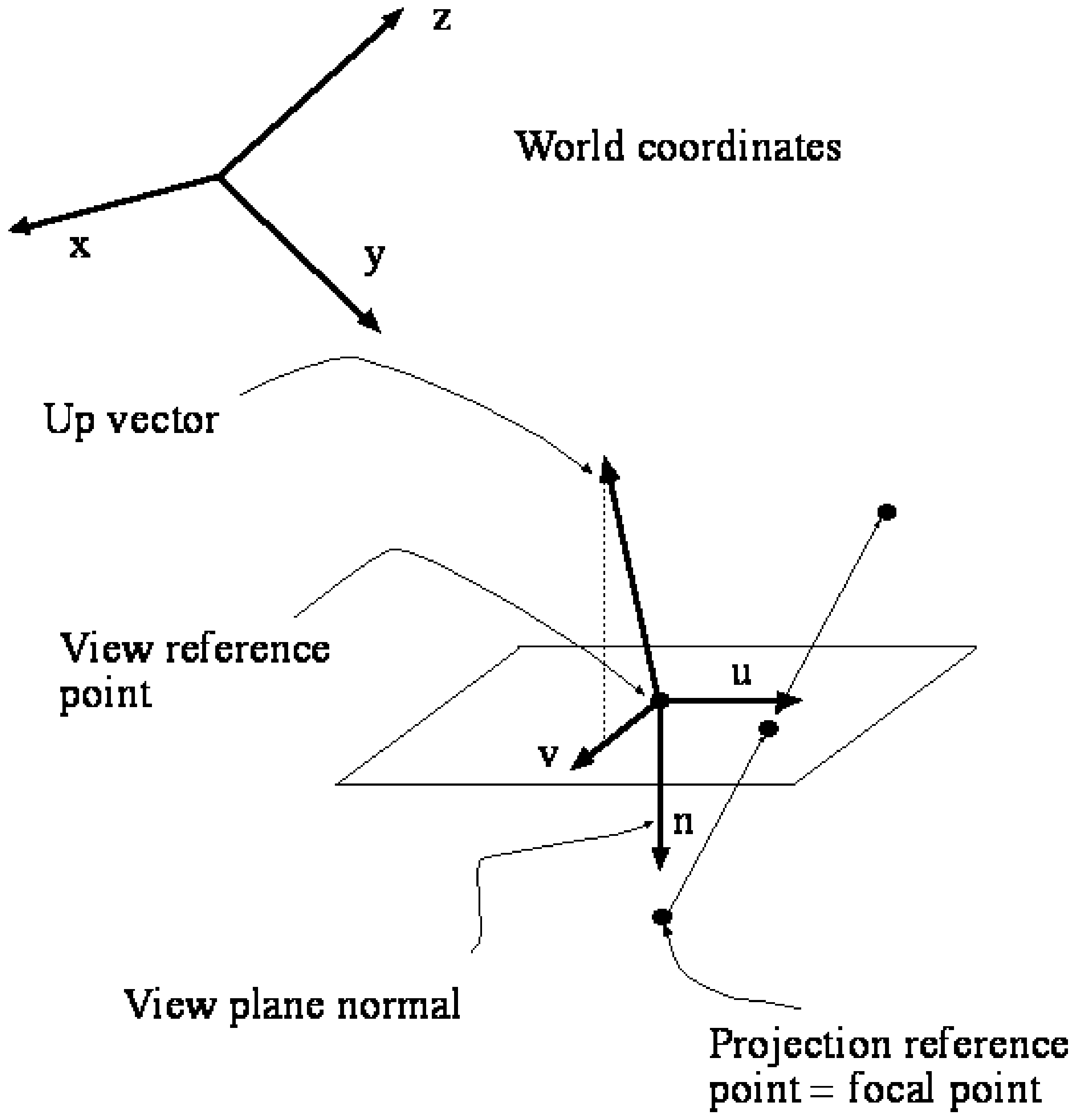
Specifying a camera

- Why use VUP?
  - Convenient for the user but there are other ways (OpenGL has several ways to negotiate camera parameters, including very much how we are doing it).
  - A world centric coordinate system is natural for the user. In particular, the user may think in terms of the camera rotation around the axis (**n**) relative to a natural horizon and/or "up" direction.
  - This will mean that VUP cannot be parallel to **n**. Often one will fix VUP (e.g. to the Y-axis) but this is too restrictive for some applications.
- Why use a "backwards" pointing **n**?
  - It is more natural to make the camera direction point the other way, but this makes the camera coordinates left handed. (You will see it done both ways).

View reference point and view plane normal specify image plane.

Up vector gives an "up" direction in the image plane, providing for user twist of camera about **n**. **v** is projection of up vector into image plane (formula for **v** to come soon).

u is chosen so that (**u**, **v**, **n**) is a right handed coordinate system; i.e. it is possible to translate/rotate so that (**x->u**, **y->v**, **z->n**) (and we'll do this shortly).

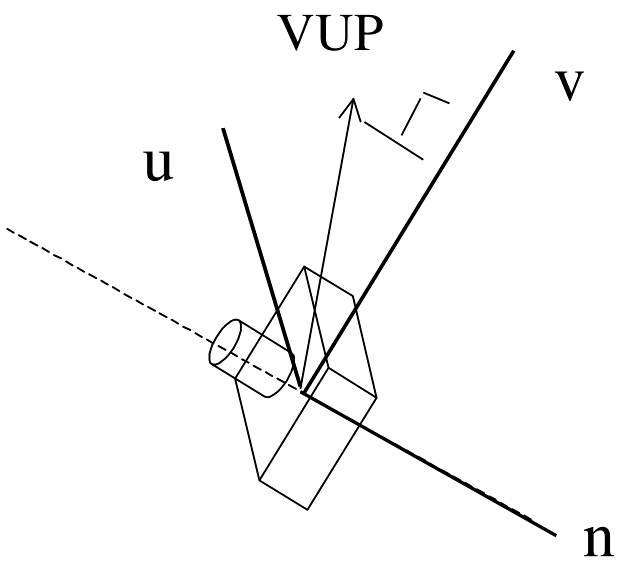


$$\mathbf{v} \parallel \mathbf{n} \prod \mathrm{VUP} \ \square \ \mathbf{n}$$

(We write || for parallel to---we still need to make the RHS into a unit vector to get **v**.)

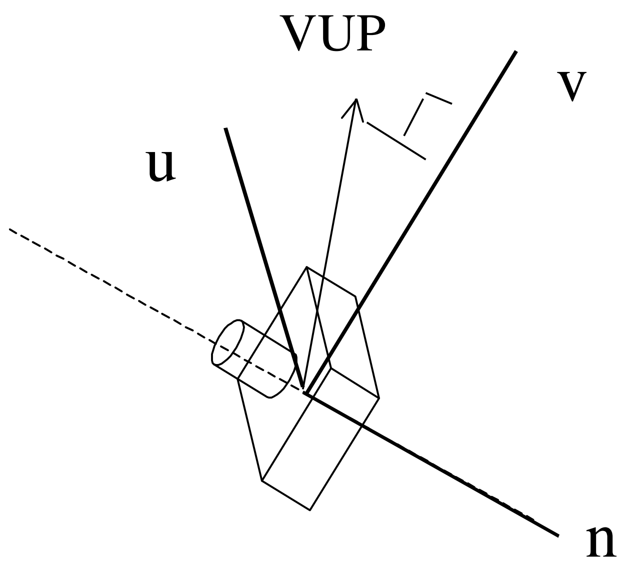
Why does this work?

Want **v** to be in plane of VUP and **n** and perpendicular to **n**

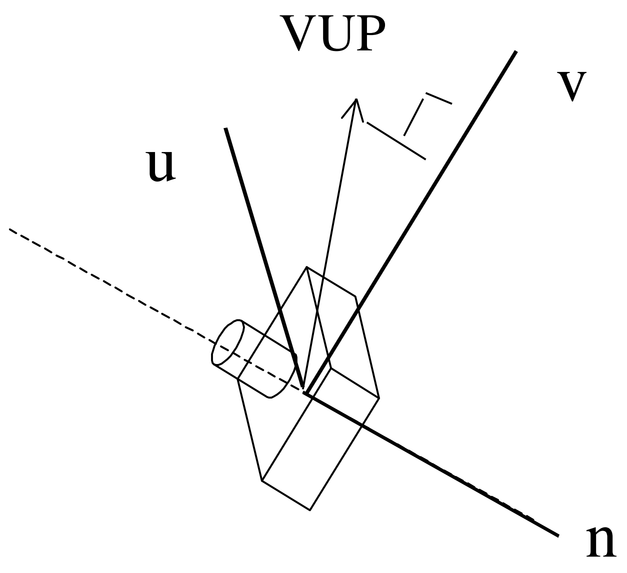


n | | VUP □ **n** is perpendicular to (VUP x **n)** and **n**

(VUP x **n**) gives a direction perpendicular to both VUP and **n**. So if you are perpendicular to that, then you must be back in the plane defined by VUP and **n** (there are only 3 dimensions!).



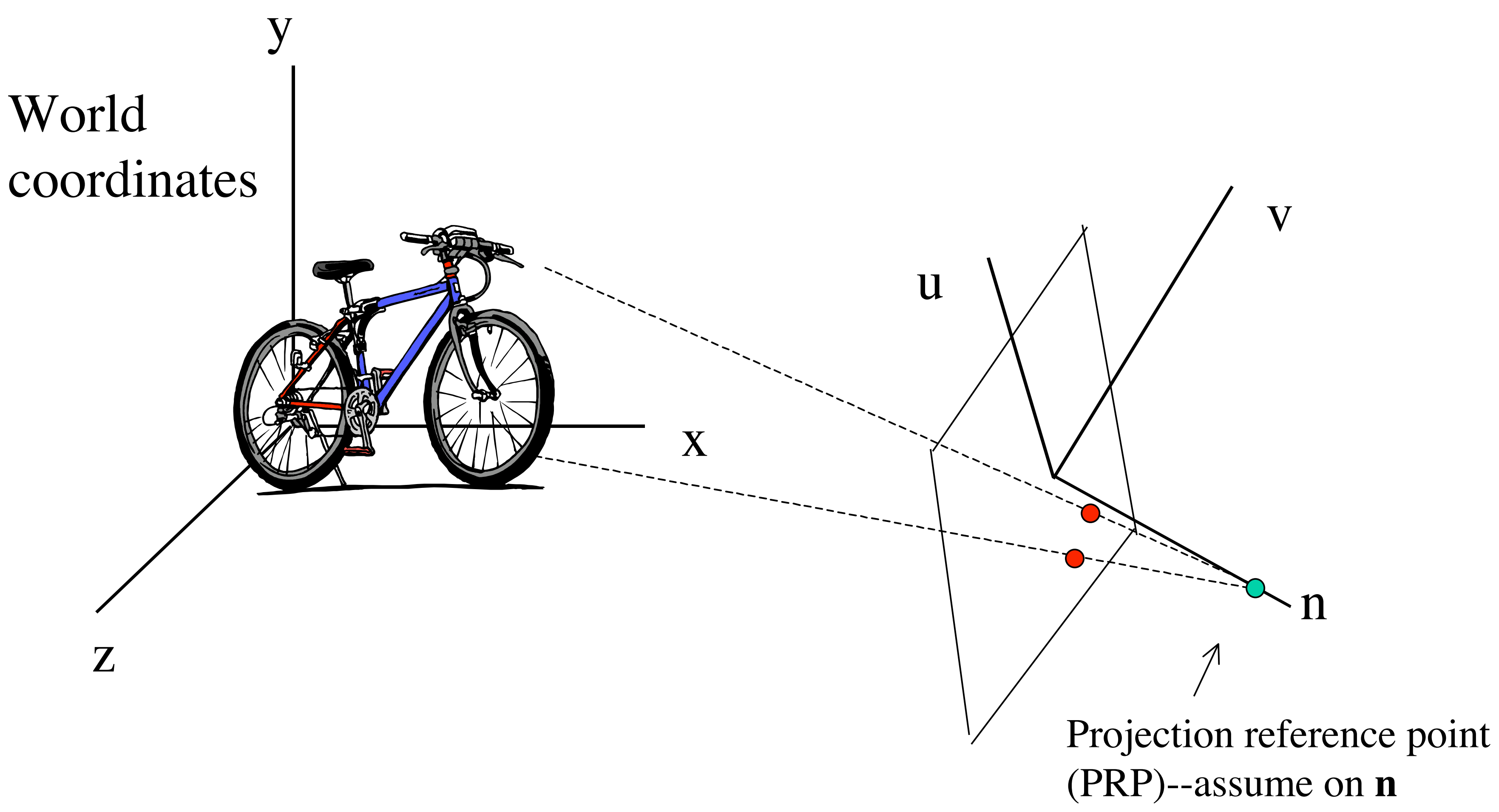
Now that we have **n** and **v**, we can compute **u**. How ?



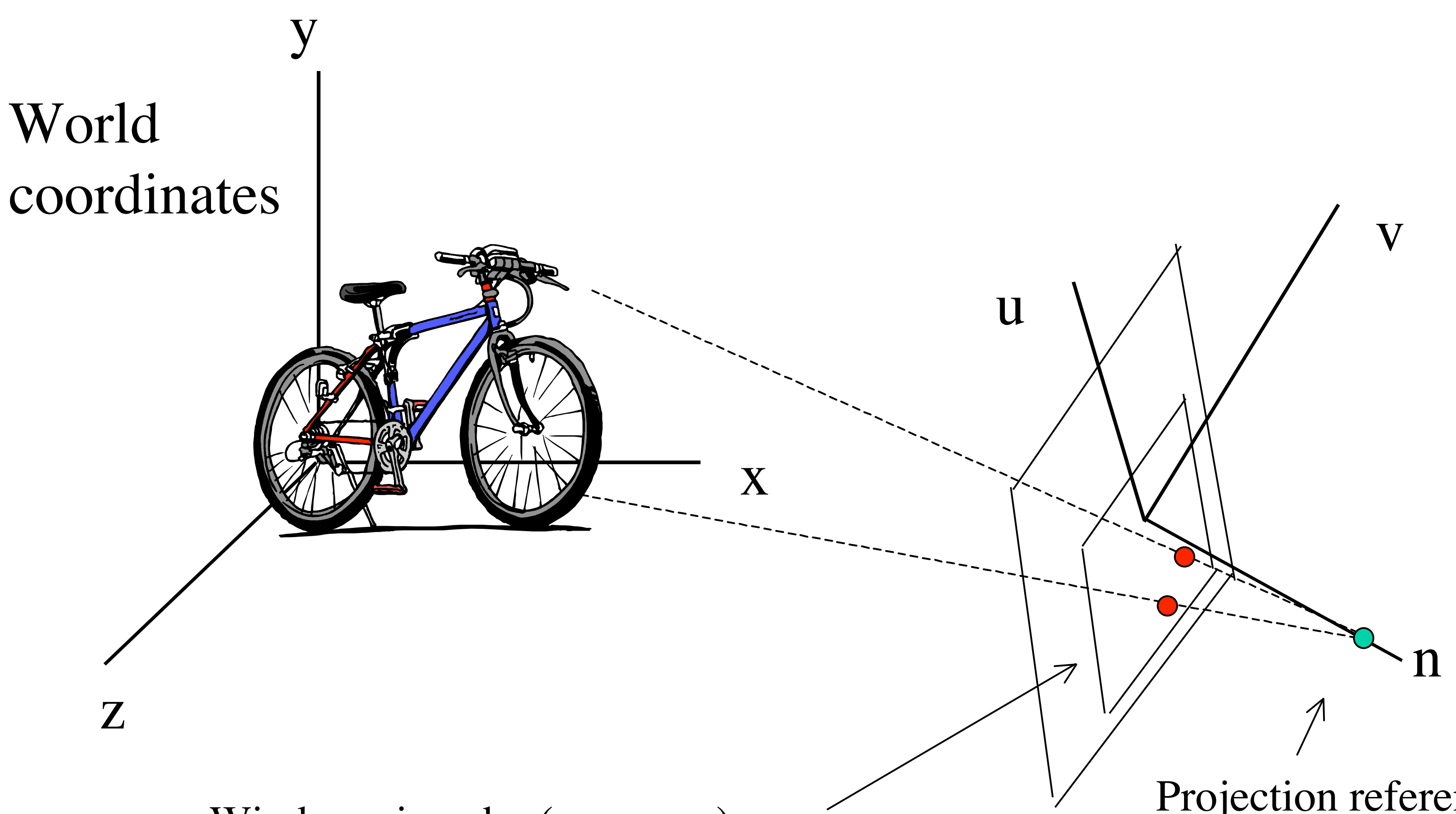
Now that we have **n** and **v**, we can compute **u** by:

u = **v** x **n**

Specifying a camera



Specifying a camera



Window given by (u_{min}, u_{max}), (v_{min}, v_{max}), denote center by CW

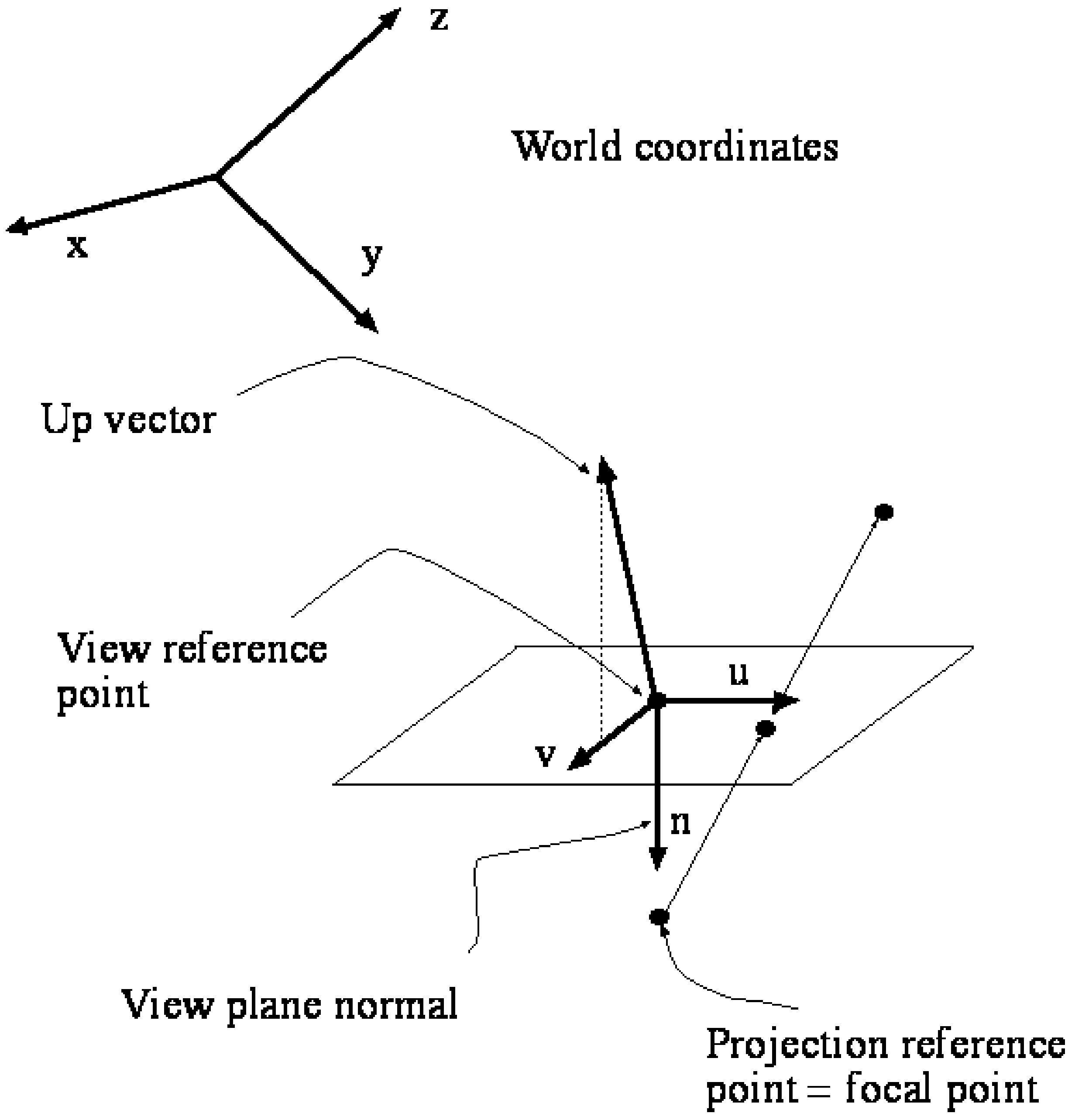
Projection reference point (PRP)--assume on **n**

VRP, VPN, VUP must be in world coords;

PRP (focal point) could be in world coords, but more commonly, camera coords

We will use camera coords, and further assume that it is simply (0,0,f).

(What follows will actually work fine for an off-axis PRP, but this is rarely needed).

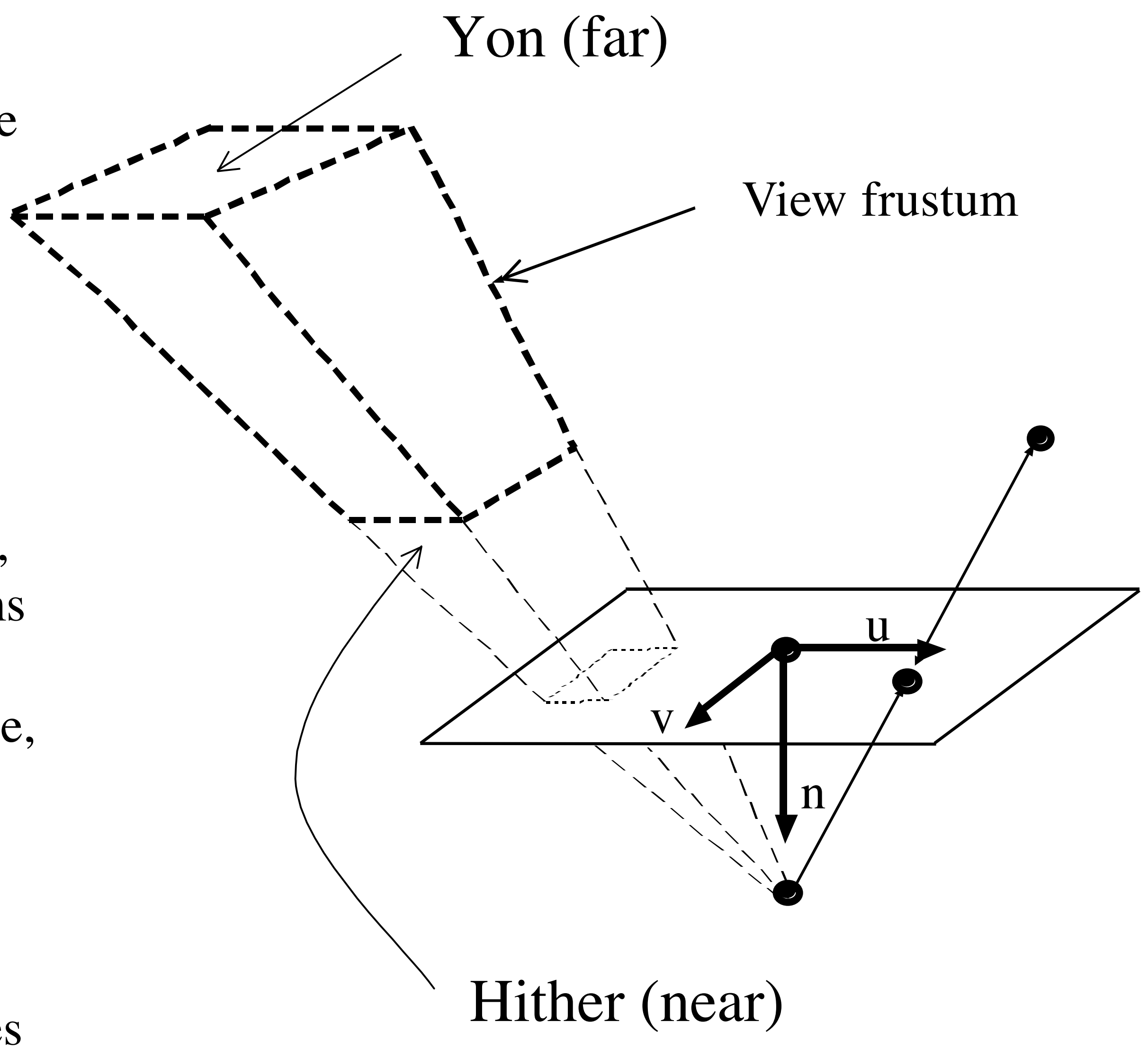


U, V can be used to specify a window in the image plane; only this section of image plane ends up on the screen.

This window defines four planes; points outside these planes are not rendered.

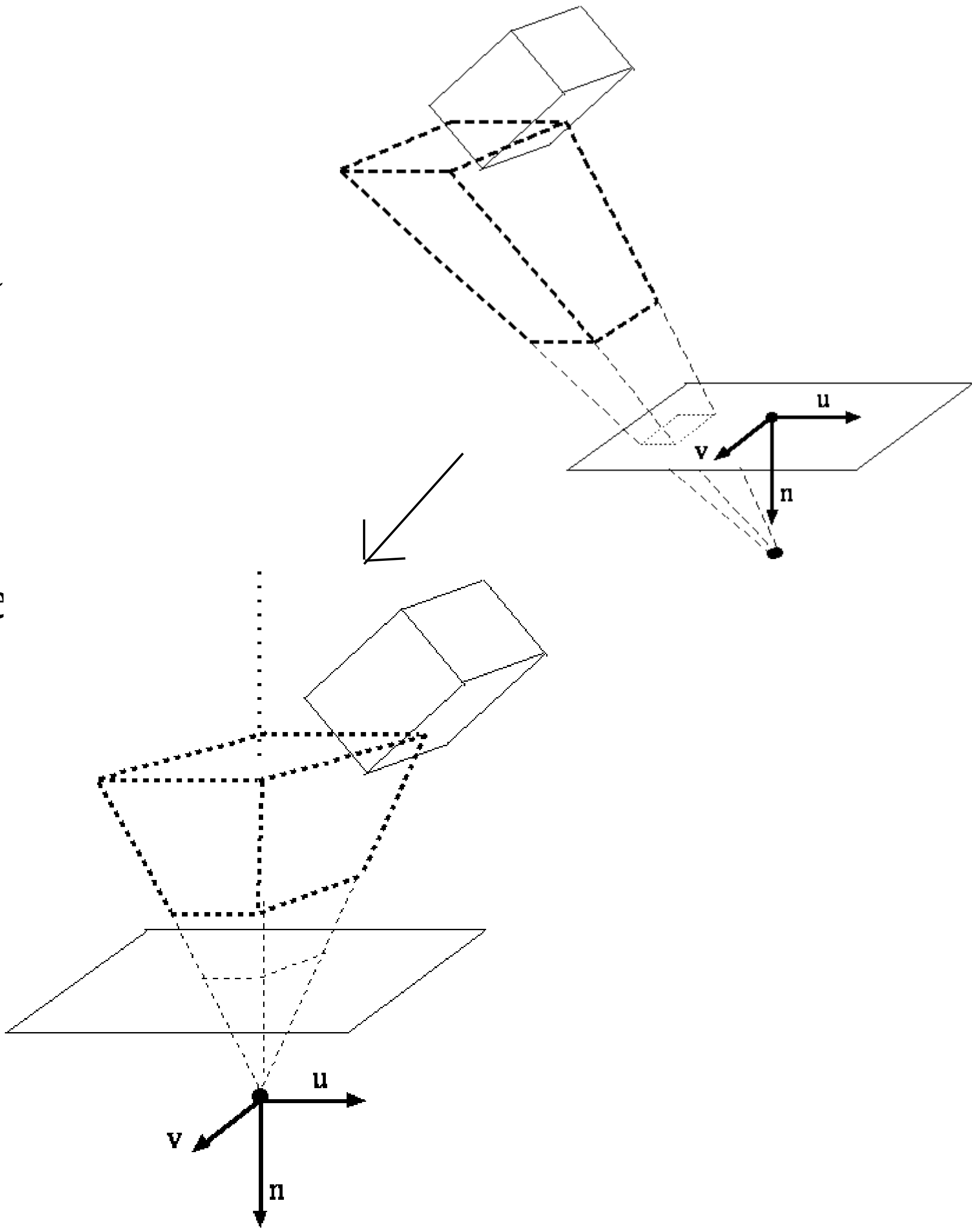
Hither and yon clipping planes, which are always given in terms of camera coordinates, and always parallel to the film plane, give a volume - known as the view frustum.

Orthographic case: - view frustum is cuboid (i.e. all angles right angles, but edges not necessarily of equal length).

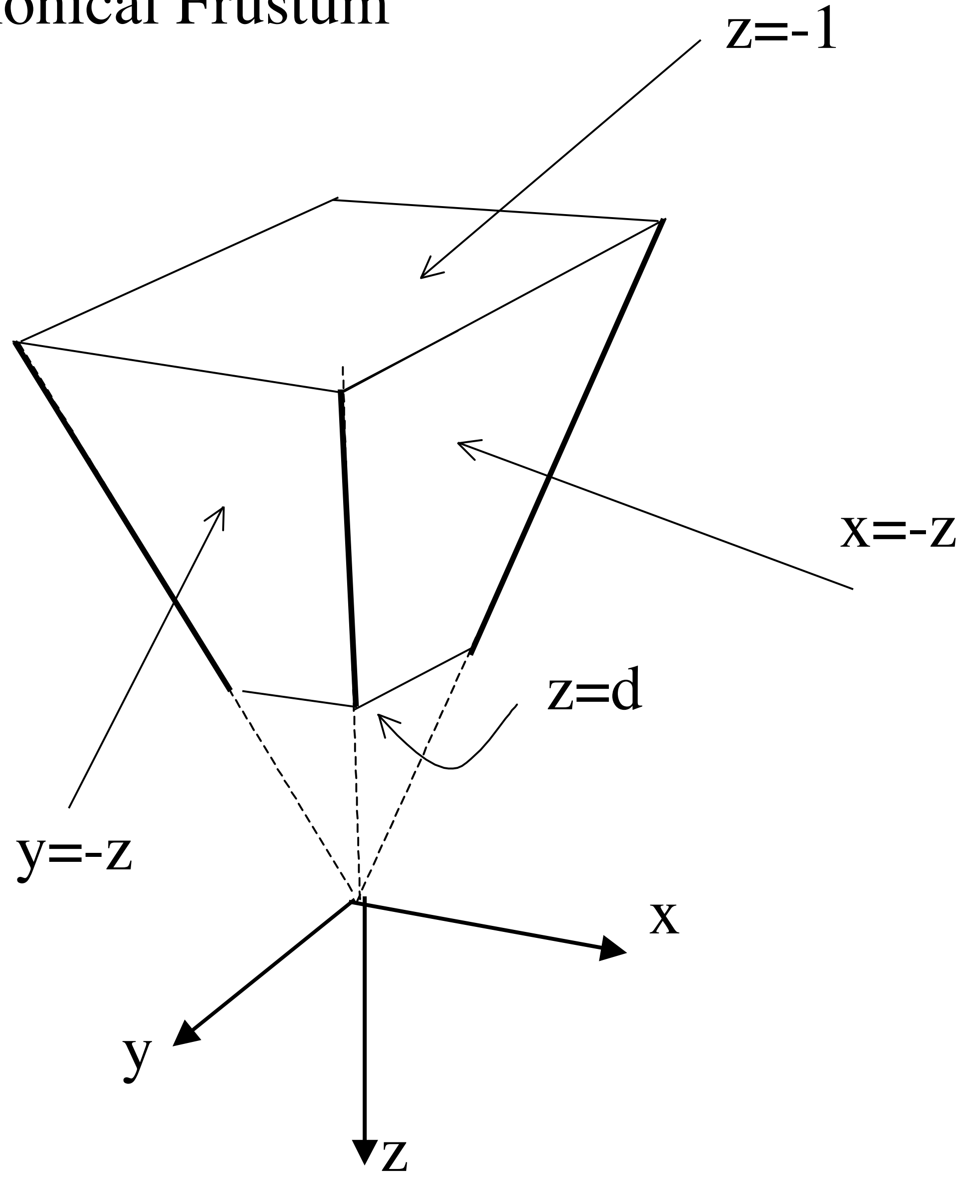


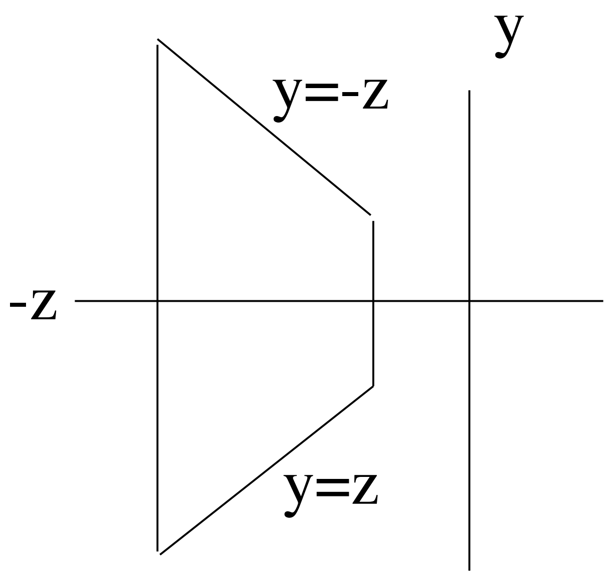
If we clip against the frustum blindly, clipping is hard - this is because planes bounding the frustum have a complex form

Solution: transform view frustrum into a canonical form, where clip planes have easy form—e.g. z=x, z=-x, z=y, z=-y, z=-1, z=d



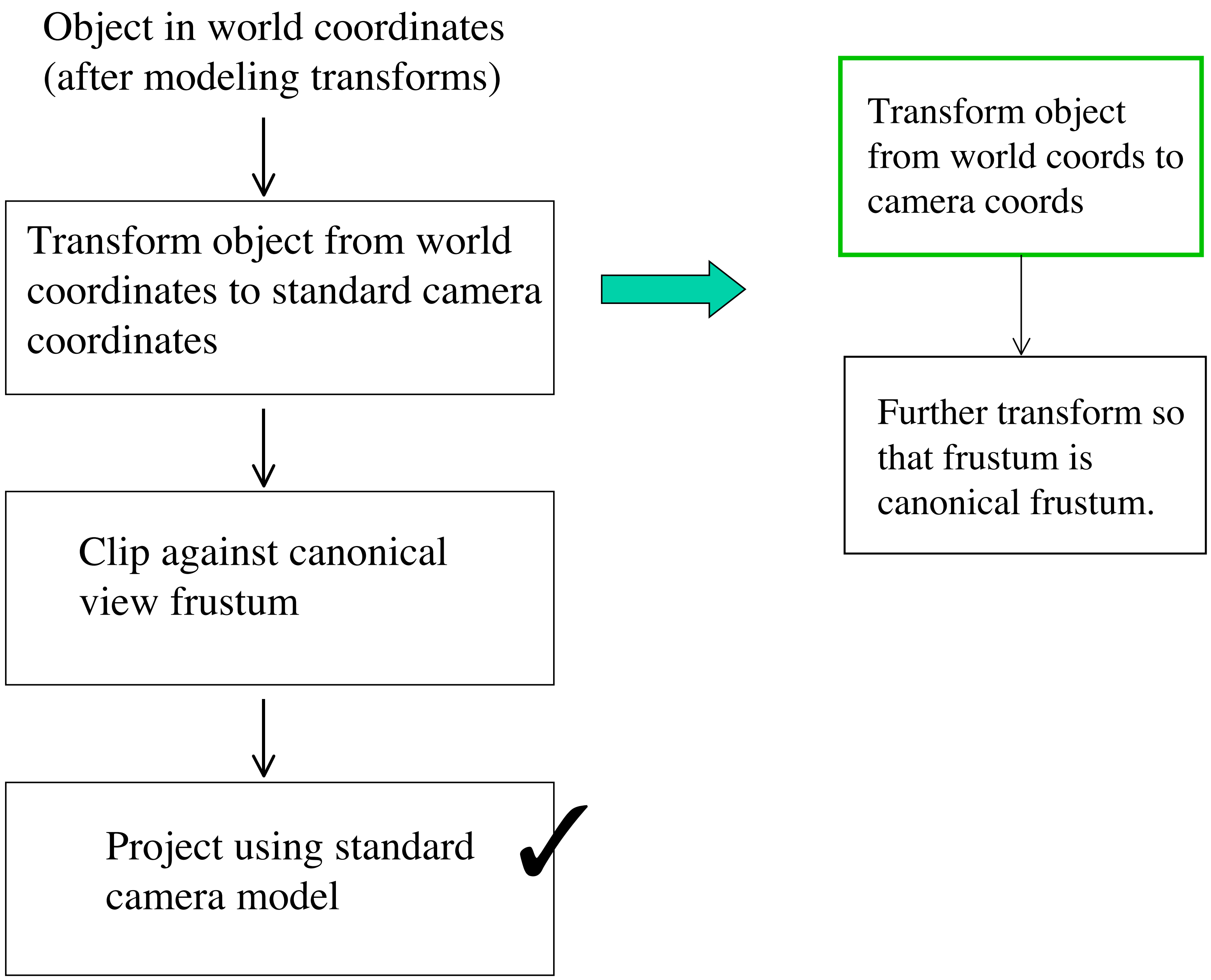
Canonical Frustum





If image plane transforms to z=m then in new frame, projection is easy:

(x, y, z) -> (m x /z, m y/z)


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 ✓
Transform object from world coords to camera coords
Further transform so that frustum is canonical frustum.

Transform object from world coords to camera coords

Step 1. Translate the camera at VRP to the world origin. Call this T_1.

Translation vector is simply negative VRP.

(We are changing the coordinate system of the world, which is the same thing mathematically as moving the camera. We want object world coordinates to **change** so that the camera location **becomes** the origin).

Transform object from world coords to camera coords

Step 2. Rotate camera coordinate frame (in w.c.) so that so that **u** is **x**, **v** is **y**, and **n** is **z**. The matrix is ?

(We are changing the coordinate system of the world, which is the same thing mathematically as moving the camera. We want object world coordinates to **change** so that the camera axis **becomes** the standard axis—e.g, **u** becomes (1,0,0), **v** becomes (0,1,0) and **n** becomes (0,0,1)).

Transform object
from world coords to
camera coords

Step 2. Rotate camera coordinate frame (in w.c.) so that so that **u** is **x**, **v** is **y**, and **n** is **z**. The matrix is:

$$\begin{vmatrix} \mathbf{u}^T & 0 \\ \mathbf{v}^T & 0 \\ \mathbf{n}^T & 0 \\ 0\ 0\ 0 & 1 \end{vmatrix}$$

(why?)

Transform object from world coords to camera coords

$$\begin{vmatrix} \mathbf{u}^T & 0 \\ \mathbf{v}^T & 0 \\ \mathbf{n}^T & 0 \\ 0\ 0\ 0 & 1 \end{vmatrix} \mathbf{u} = \begin{vmatrix} 1 \\ 0 \\ 0 \\ 0 \end{vmatrix}$$

In the current coords (world shifted so that VPR is at origin): **u** maps into the X-axis unit vector (1,0,0,0) which is what we want.

(Similarly, **v**-->Y-axis unit vector, **n**-->Z-axis unit vector)