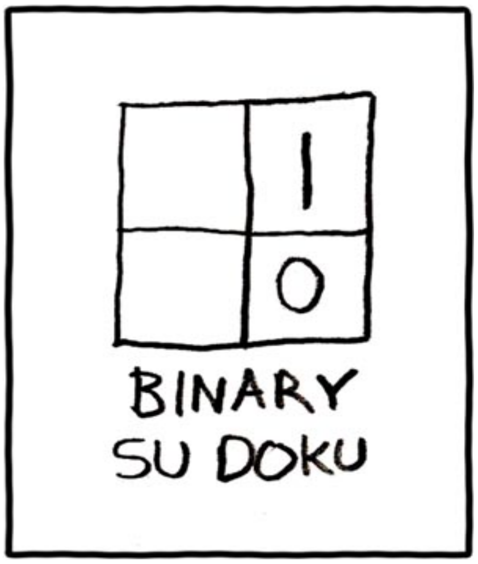
3D Camera Viewing

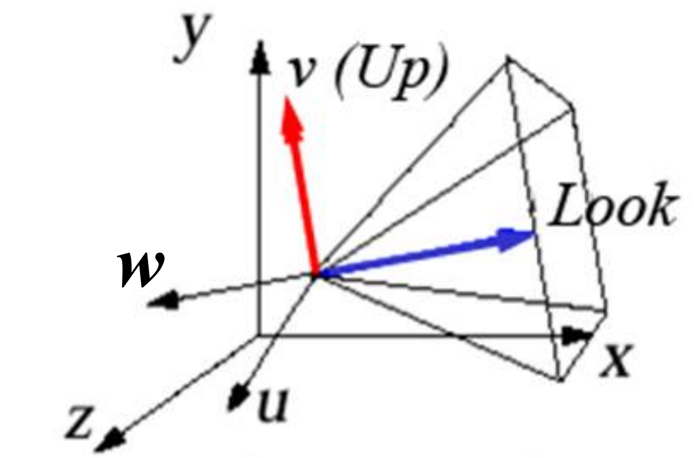
(or the great Local Coordinate System quest)



It looks like a matrix…
Sort of…

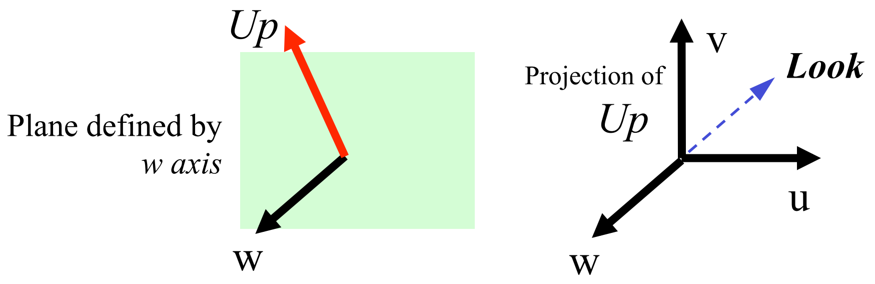
Arbitrary 3D views

- Now that we have familiarity with terms we can say that view volumes can be specified by **placement** and **shape**
- **Placement:**
  - ***Position*** (a point)
  - ***Look*** and ***Up*** vectors
- **Shape:**
  - horizontal and vertical ***view angles*** (for a perspective view volume)
  - front and back clipping planes
  - Note camera coordinate system ***(u, v, w)*** is defined in the world ***(x, y, z)*** coordinate system



Arbitrary Perspective Frustum

Finding u, v, and w from Position, Look, and Up (1/5)



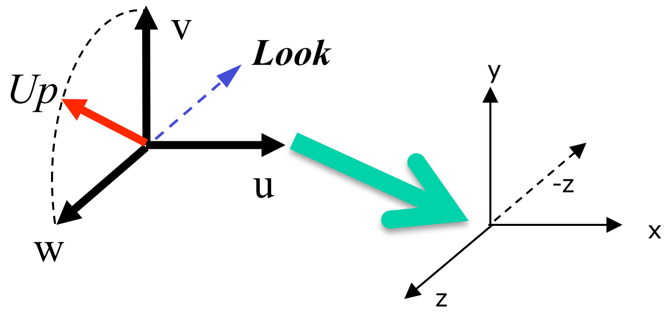
- We know that we want the (*u*, *v*, *w*) axes to have the following properties:
  - our arbitrary *Look Vector* will lie along the negative *w*-axis (roughly, w will correspond to Z, so our camera will look down –Z).
  - a projection of the *Up Vector* into the plane defined by the w-axis as its normal will lie along the v-axis
  - The *u*-axis will be mutually perpendicular to the *v* and *w*-axes, and will form a right-handed coordinate system
- Plan of attack: first find *w* from *Look*, then find *v* from *Up* and *w*, then find *u* as a normal to the plane defined by *w* and *v*

Finding u, v, and w (2/5)

Finding *w*

- Finding *w* is easy. *Look vector* lies on –*z*. Since *z* maps to *w*, *w* is a normalized vector pointing in the opposite direction from our arbitrary *Look vector*

$$w = \frac{-Look}{\|Look\|}$$



- Note that *Up* and *w* define a plane, and that *u* is a normal to that plane, and that *v* is a normal to the plane defined by *w* and *u*

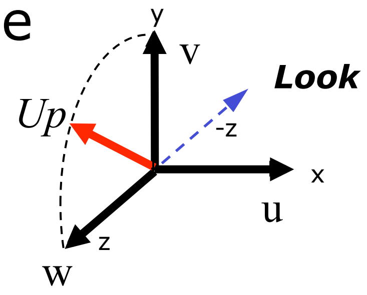
Finding u, v, and w (3/5)

*Finding **v***

- Problem: find a vector, ***v***, perpendicular to ***w***
- Solution: project out the ***w*** component of the *Up* vector and normalize

$$v = Up - (Up \bullet w)w$$

$$v = \frac{v}{\|v\|}$$

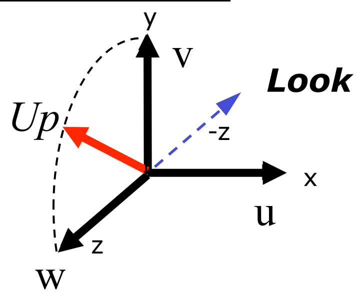


- ***w*** is unit length, but *Up* vector might not be unit length or perpendicular to ***w***, so we have to remove the ***w*** component and then normalize

- By removing the ***w*** component from the *Up* vector, the resulting vector is the component of *Up* in a direction perpendicular to ***w***

Finding u, v, and w (4/5)

Finding *u*



- We could use a cross-product
- But which one: *w* x *v* or *v* x *w* ?
  - *w* X *v* and *v* X *w* are both perpendicular to the plane, but in different directions . . .
- Answer: use *v* X *w* to create a right-handed coordinate system (similar to the World Coordinate System)

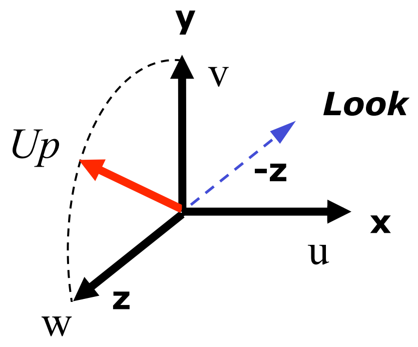
As a reminder, the cross product of two vectors *a* and *b* is:

$$a \times b = \begin{bmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{bmatrix}$$

Finding u, v, and w (5/5) (Remember THIS!!)

To summarize

$$w = \frac{-Look}{\|Look\|}$$



$$v = \frac{Up - (Up \bullet w)w}{\|Up - (Up \bullet w)w\|}$$

$$u = v \times w$$

Do **not** need to normalize *u* . . . why? Since *w* and *v* are unit length and perpendicular, their cross product is also unit length.

From (u,v,w) to 2D Film Plane

- Given (u,v,w), how do we calculate 2D projection on film plane ?
- Arbitrary view volume is too complex…
  - We have an arbitrary view with viewing parameters
  - Problem: map arbitrary view specification to 2D image of scene. This is hard.
- Reduce it to a simpler problem!
- The **canonical view volume!**

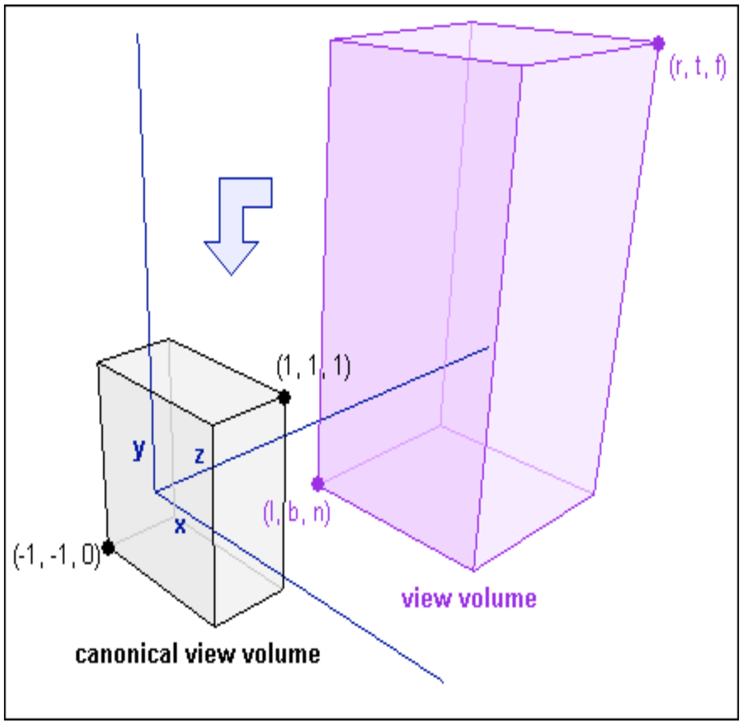
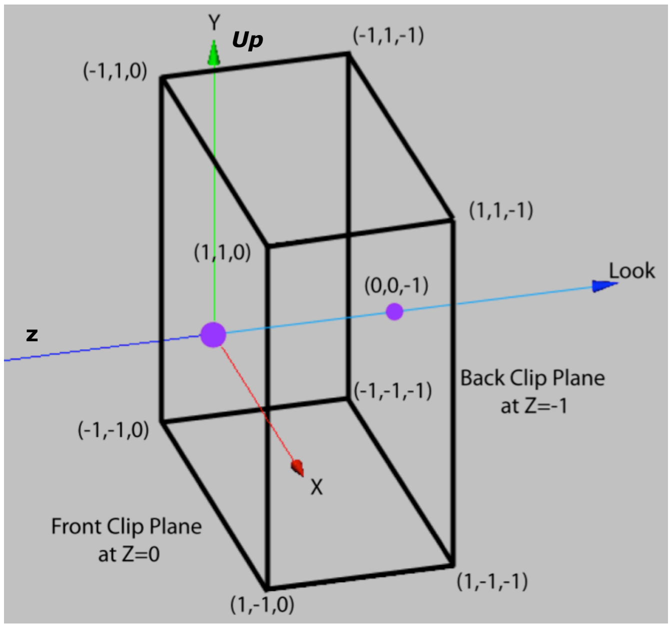


Image credit: http://www.codeguru.com/cpp/misc/misc/math/article.php/c10123__2/

Canonical View Volume

- Solution: reduce to a simpler problem and solve
  - – there is a view specification from which it is easy to take a picture. We'll call it the *canonical view*: from the origin, looking down the negative *z*-axis
  - – think of the scene as lying behind a window and we're looking through that window



– parallel projection

– sits at origin:

Position = (0, 0, 0)

– looks along negative *z*-axis:

Look vector = (0, 0, –1)

– oriented upright:

Up vector = (0, 1, 0)

– film plane extending from –1 to 1 in *x* and *y*

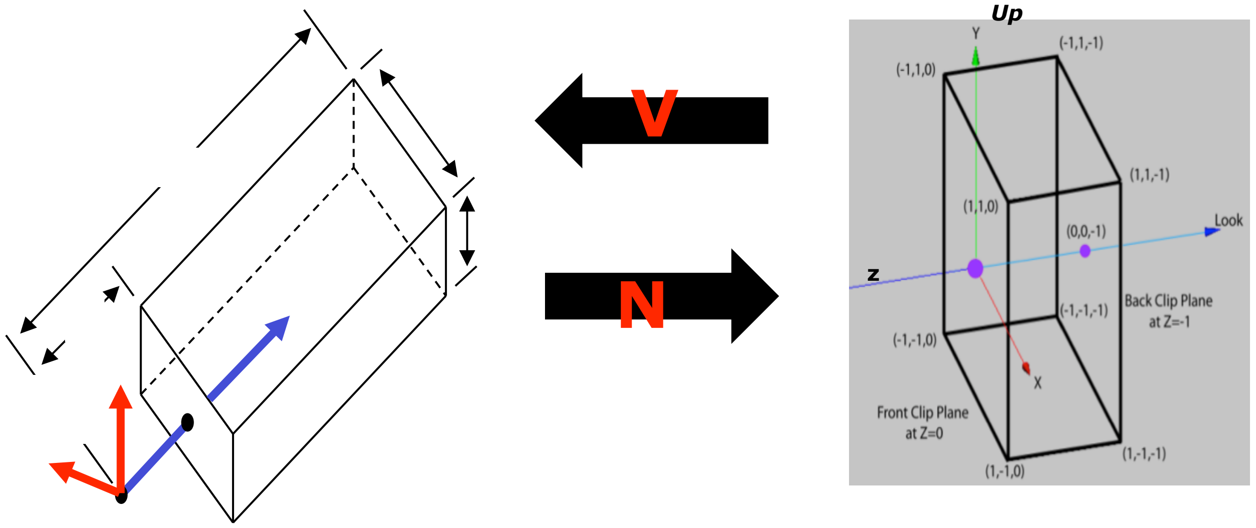
– *near* and *far* clipping planes at:

z = 0, respectively at z = 1

- Note: *Look vector* along negative, not positive, *z*-axis is odd choice but makes math easier; ditto choosing $-1 \leq x, y \leq 1$

The Normalizing Transformation

- Goal: transform arbitrary view and scene to canonical view volume, maintaining relationship between view volume and scene, then render
- For parallel view volumes need only rotations, scales, and translations
- The composite transformation composed of these scales, rotations and translations is a 4x4 homogenous matrix called the **normalizing transformation N** (the inverse is called the **viewing transformation V** and turns a canonical view volume into an arbitrary one)



Remember that our camera is just a model, there is no actual camera in our scene. The normalizing matrix N needs to be applied to every vertex in our scene to simulate this transformation.

View Volume Translation

First step in sending the (u,v,w) axes into (x,y,z)

- Start by moving the camera to the world origin
- Let P_n be the center of the camera's *near* plane:

  P_n = *Position* + *near* * ***w***

  - *near* is a scalar (distance from camera to its near plane); ***w*** is a vector; *Position* is a 3D point (the position of the camera); so P_n is a 3D point as well
- We want a matrix to translate

  (P_{nx}, P_{ny}, P_{nz}) to $(0, 0, 0)$

- The following matrix will do:

$$\begin{bmatrix} 1 & 0 & 0 & -P_{n_x} \\ 0 & 1 & 0 & -P_{n_y} \\ 0 & 0 & 1 & -P_{n_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- We will multiply by this matrix all the 3D vertices in our camera's view volume.

View Volume Rotation

Rotate the (u,v,n) axes so they line up with (x,y,z)

- Let's call the three *x*-axis, *y*-axis, and *z*-axis-aligned unit vectors e_1, e_2, e_3, and let's leave out the homogeneous coordinate, for now
- Writing out:

$$e_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad e_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad e_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

- Need to rotate *u* into e_1, *v* into e_2, and *w* into e_3
- Need to find some $\mathbf{R}_{\mathbf{rot}}$ rotation matrix such that

$$\mathbf{R}_{\mathbf{rot}}u = e_1$$
$$\mathbf{R}_{\mathbf{rot}}v = e_2$$
$$\mathbf{R}_{\mathbf{rot}}w = e_3$$

- Sudoku: think of each entry of the vector e_1 and compose a matrix, that when applied to *u*, turns it into e_1

Rotation Matrix (Remember THIS!!)

$$\begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Hint: ***u*, *v***, and ***w*** are unit length vectors and perpendicular to each other, hence

- dot product of each with itself is 1
- dot product of each with the other is 0

Scaling the View Volume

- We now have our camera view volume sitting at the world origin; it sits upright with its axes aligned with (x, y, z) and with its *Look* pointing towards –z
- But the size of the view volume is not the same as the canonical view volume size: we want (x,y) bounds to be at +1 and -1, and we want the far clipping plane to lie at z = -1
- Given the parallel view volume with dimensions *width* and *height* and its far clipping plane at distance *far,* our scaling matrix is as follows

$$\begin{bmatrix} \frac{2}{width} & 0 & 0 & 0 \\ 0 & \frac{2}{height} & 0 & 0 \\ 0 & 0 & \frac{1}{far} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The **N** Transform (parallel or orthogonal camera)

Now we have a complete transformation sequence:

- first translate camera to origin (or rather, its *near* plane)
- then rotate the (u,v,w) vectors into (x, y, z)
- then scale (squish) the view volume to fit between -1 and + 1 along x and y, and between 0 and -1 along z

$$\mathbf{N}_{\mathbf{ortho}} = S_{xyz}\, R_{rot}\, T_{trans}$$

$$\begin{bmatrix} \frac{2}{width} & 0 & 0 & 0 \\ 0 & \frac{2}{height} & 0 & 0 \\ 0 & 0 & \frac{1}{far} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_{n_x} \\ 0 & 1 & 0 & -P_{n_y} \\ 0 & 0 & 1 & -P_{n_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- ergo, our camera is not a real object, but a 4x4 matrix

The $\mathbf{N}_{\mathbf{perspective}}$ Transform

- Fairly similar construction to $\mathbf{N}_{\mathbf{ortho}}$, except slightly different scaling and an added distortion step at the end to stretch the perspective pyramid trunk into a cuboid

$$\mathbf{N}_{\mathbf{persp}} = M_{pp}\, S_{xyz}\, R_{rot}\, T_{trans}$$

with $M_{pp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{-1}{c+1} & \frac{c}{c+1} \\ 0 & 0 & -1 & 0 \end{bmatrix}$ where $c = -\,near/far$;

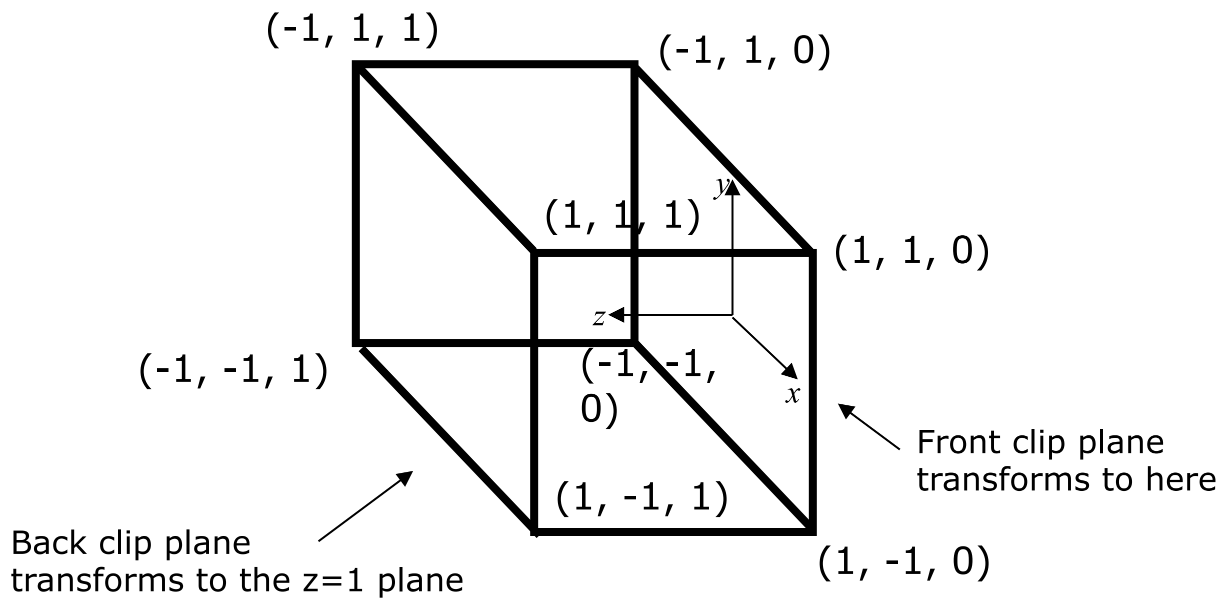
$$S_{xyz} = \begin{bmatrix} \frac{1}{\tan(\frac{\theta_w}{2})\,far} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{\theta_h}{2})\,far} & 0 & 0 \\ 0 & 0 & 1/far & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where θ_h and θ_w are the height and width angles.

$$R_{rot} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad T_{trans} = \begin{bmatrix} 1 & 0 & 0 & -Pos_x \\ 0 & 1 & 0 & -Pos_y \\ 0 & 0 & 1 & -Pos_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Clipping

- We said that taking the picture from the canonical view would be easy: final steps are clipping and projecting onto the film plane
- Need to clip scene against sides of view volume
- However, we've normalized our view volume into an axis-aligned cuboid that extends from –1 to 1 in *x* and *y* and from 0 to 1 in *z*



- Note: This is the flipped (in *z*) version of the canonical view volume
- Clipping is easy! Test *x* and *y* components of vertices against +/-1. Test *z* components against 0 and 1

Projection

- Can make an image by taking each point and "ignoring *z*" to project it onto the *xy*-plane
- A point (*x*,*y*,*z*) where

$$-1 \le x, y \le 1, 0 \le z \le 1$$

turns into the point (*x'*, *y'*) in screen space (assuming viewport is the entire screen) with

$$0 \le x', y' < 1024$$

by

- ignoring *z*

$$x' \rightarrow 512(x+1)^{1}$$

$$y' \rightarrow 512(y+1)$$

- If viewport is inside a Window Manager's window, then we need to scale down and translate to produce "window coordinates"
- Note: because it's a parallel projection we could have projected onto the front plane, the back plane, or any intermediate plane … the final pixmap would have been the same

1. Note that these functions are not exactly correct since if x or y is ever 1, then we will get x' or y' to be 1024 which is out of our range, so we need to make sure that we handle these cases gracefully. In most cases, making sure that we get the floor of 512(x+1) will address this problem since the desired x will be less than 1.

Summary

- The "camera" is a composite matrix multiplication of vertices, clipping, and a final matrix multiplication to produce screen coordinates.
- Final composite matrix (*CTM*) is composite of all modeling (instance) transformations (*CMTM*) accumulated during scene graph traversal from root to leaf, composited with the final composite normalizing transformation ***N*** applied to the root/ world coordinate system:

1) $N = \textit{either } N_{ortho} \textit{ or } N_{persp}$

2) $CTM = N \cdot CMTM$

3) $P' = CTM \cdot P$ for every vertex P defined in its own coordinate system

4) $P_{screen} = 512 \cdot P' + 1$ for all **clipped** P'

- Aren't homogeneous coordinates wonderfully powerful?
- the two **Remember THIS!!** slides: constructing a local coordinate system, *and* aligning a coordinate system with another