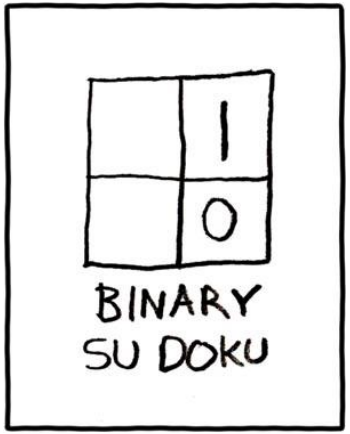


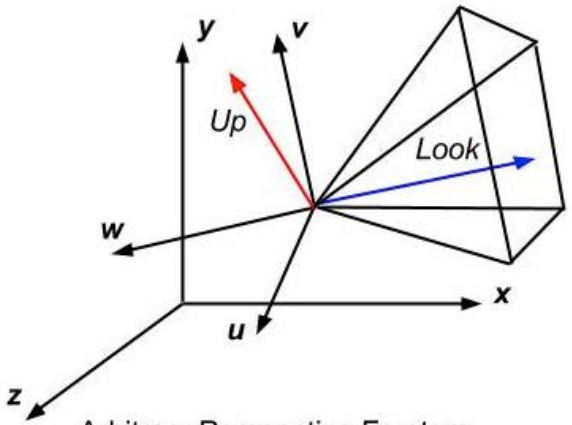
It looks like a matrix…
Sort of…

Viewing III

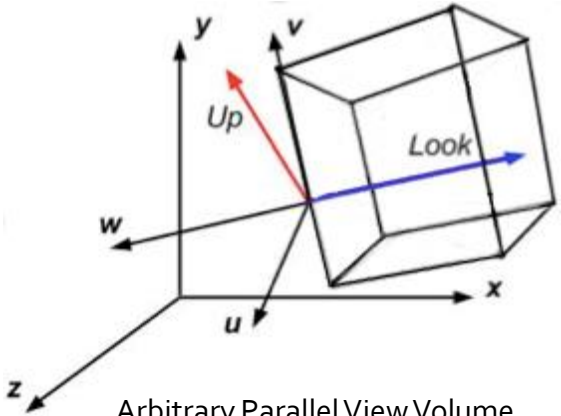
Projection in Practice

Arbitrary 3D views

- Now that we have familiarity with terms we can say that these view volumes/frusta 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 that camera coordinate system ***(u, v, w)*** is defined in the world ***(x, y, z)*** coordinate system



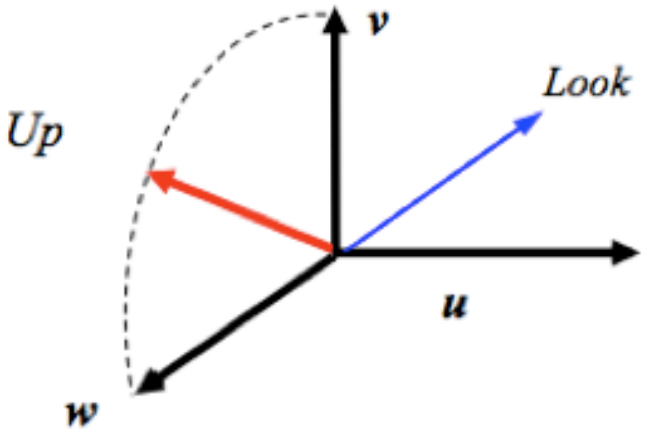
Arbitrary Perspective Frustum



Arbitrary Parallel View Volume

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

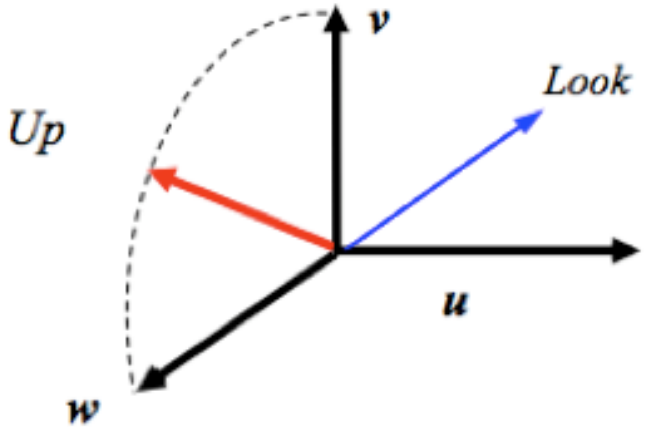
- Want the ***u***, ***v***, ***w*** camera coordinate system axes to have the following properties:
  - Our arbitrary ***look*** vector will lie along the negative ***w***-axis
  - The ***v***-axis will be defined by the vector normal to ***look*** in the plane defined by ***look*** and ***up***
  - 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 ***wv***-plane



Finding ***u***, ***v***, and ***w*** from *Position*, *Look*, and *Up* (2/5)

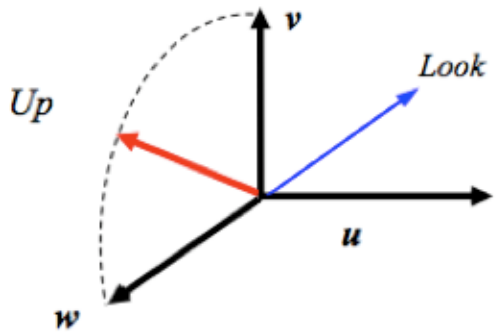
- Finding ***w*** is easy. ***look*** in the canonical volume lies on –***w***, so ***w*** is a normalized vector pointing in the direction opposite to ***look***

$$\mathbf{w} = \frac{-look}{\|look\|}$$



Finding ***u***, ***v***, and ***w*** from *Position*, *Look*, and *Up* (3/5)

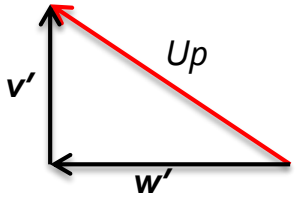
- Finding ***v***
- Problem: find a unit vector ***v*** perpendicular to the unit vector ***w*** in the ***look-up*** plane
- Solution: subtract the ***w*** component of the ***up*** vector to get ***v′*** and normalize. To get ***w*** component ***w′*** of ***up***, scale ***w*** by projection of ***up*** on ***w***:



$$up = \mathbf{w}' + \mathbf{v}'$$

$$\boldsymbol{v}' = up - (up \bullet \boldsymbol{w})\boldsymbol{w}$$

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

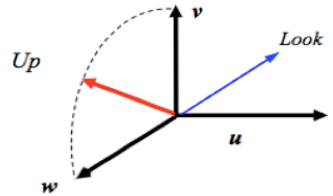


Looking directly at ***wv***-plane

Finding ***u***, ***v***, and ***w*** from *Position*, *Look*, and *Up* (4/5)

- Finding ***u***
- We can use the cross-product, but which? Both ***w*** × ***v*** and ***v*** × ***w*** are perpendicular to the plane, but they go in opposite directions.
- Answer: cross-products are "right-handed," so use ***v*** × ***w*** to create a right-handed coordinate frame

$$\boldsymbol{u} = \boldsymbol{v} \times \boldsymbol{w}$$



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

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

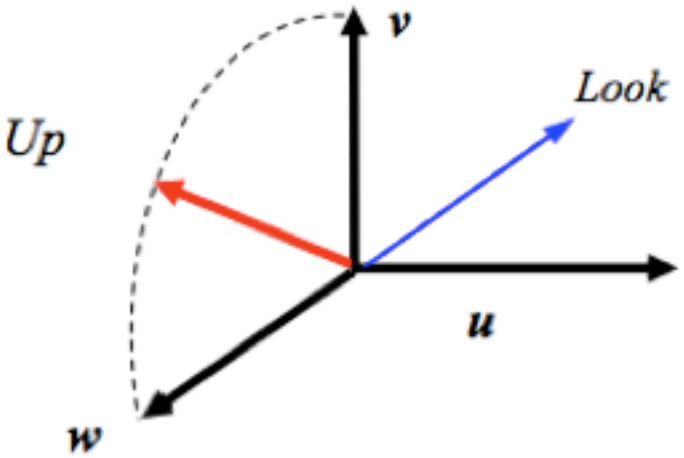
Finding ***u***, ***v***, and ***w*** from *Position*, *Look*, and *Up* (5/5)

- To Summarize:

$$\mathbf{w} = \frac{-look}{\|look\|}$$

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

$$\boldsymbol{u} = \boldsymbol{v}' \ \boldsymbol{w}$$



- Now that we have defined our camera coordinate system, how do we calculate projection?

The Canonical View Volume

- How to take contents of an arbitrary view volume and project them to a 2D surface?
  - arbitrary view volume is too complex…
- Reduce it to a simpler problem! The **canonical view volume!**
- Easiest case: parallel view volume (aka *standard* view volume)
  - Specific orientation, position, height and width that simplify clipping, VSD (visible surface determination) and projecting
  - Transform complex view volume and all objects in volume to the canonical volume **(normalizing transformation)** and then project contents onto normalized film plane
    - This maintains geometric relationships between camera and objects, but computationally, we only transform objects
    - Don't confuse with animation where camera may move relative to objects! Normalization applies to an arbitrary camera view at a given instant

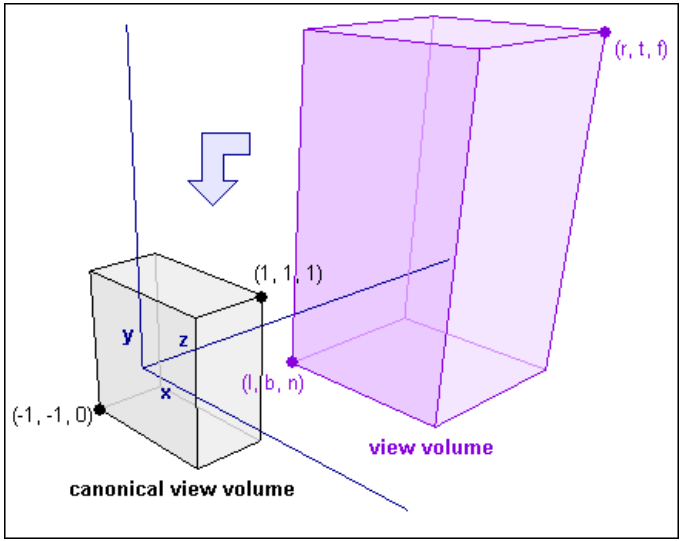
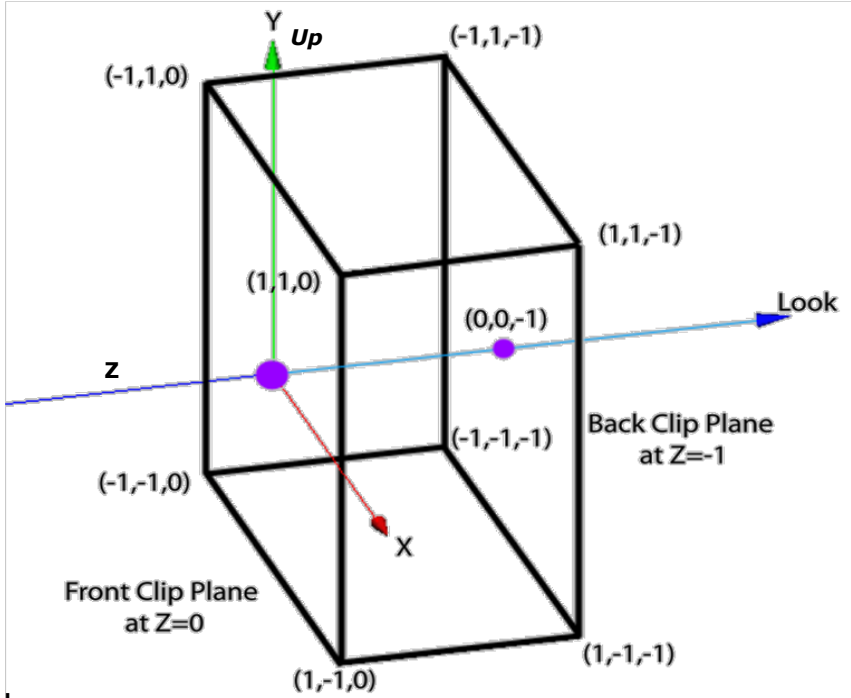


Image credit:
http://www.codeguru.com/cpp/misc/misc/math/article.php/c10123__2/

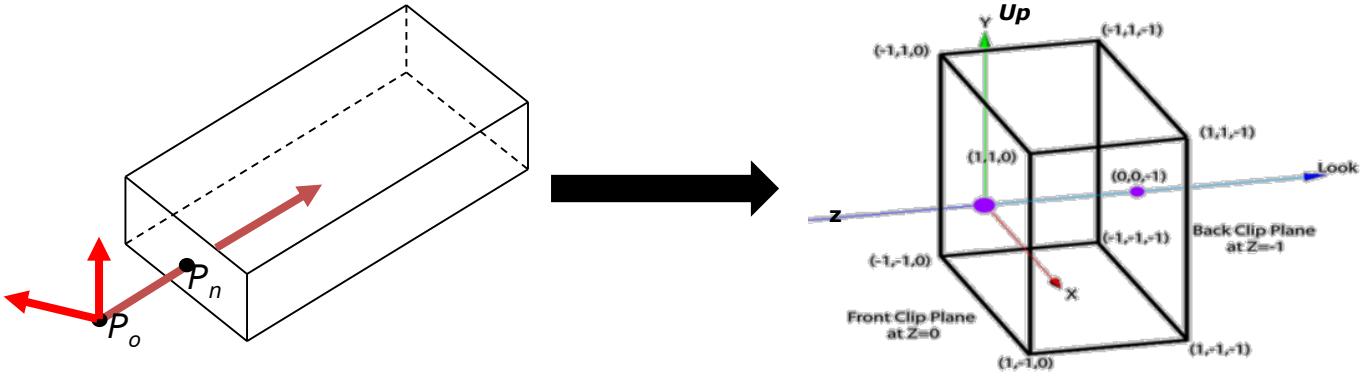
The Canonical Parallel View Volume

- Sits at the origin:
  - Center of near clipping plane = *(0,0,0)*
- Looks along negative ***z***-axis (corresponds to scene behind the "looking glass")
  - ***look*** vector = *(0,0,-1)*
- Oriented upright (along y-axis):
  - ***up*** vector = *(0,1,0)*
- Viewing window bounds normalized:
  - -1 to 1 in ***x*** and ***y*** directions
- Near and far clipping planes:
  - Near at *z = 0* plane ('front' in diagram)
  - Far at *z = -1* plane ('back' in diagram)
- Note: making our film plane bounds -1 to 1 makes the arithmetic easier



The Normalizing Transformation

- Goal: transform arbitrary view and scene to canonical view volume, maintaining relationship between view volume and scene, then render
- For a parallel view volume, need only translation to the origin, rotation to align ***u***, ***v***, ***w*** with ***x***, ***y***, ***z***, and scaling to size
- The composite transformation is a 4x4 homogeneous matrix called the normalizing transformation (the inverse is called the viewing transformation and turns a canonical view volume into an arbitrary one)



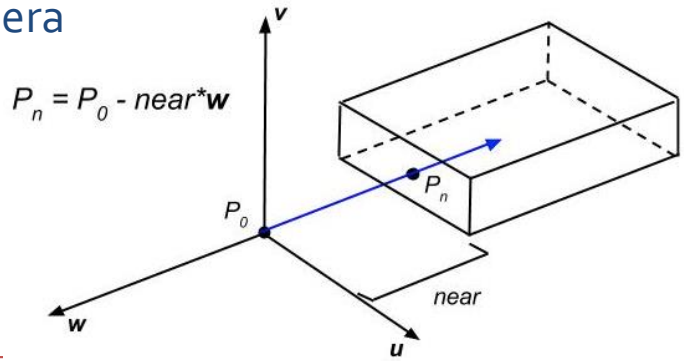
Remember our camera is just an abstract model; The normalizing matrix needs to be applied to every vertex in our scene to simulate this transformation.

- Note: the scene resulting from normalization will not appear any different from the original – every vertex is transformed in the same way. The goal is to simplify calculations on our view volume, not change what we see.
- Normalizing demo: http://cs.brown.edu/courses/cs123/demos/camera/

View Volume Translation

- Our goal is to send the ***u***, ***v***, ***w*** axes of camera's coordinate system to coincide with the ***x***, ***y***, ***z*** axes of the world coordinate system
- Start by moving camera so the center of the near clipping plane is at the origin
  - Given camera position P_o defining the origin of the ***uvw***-coordinate system, ***w*** axis, and the distances to the *near* and *far* clipping planes, the center of the near clipping plane is located at $P_n = P_o - near*\mathbf{w}$
  - This matrix will translate all world points and camera so that P_n is at the origin

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



View Volume Rotation (1/3)

- Rotating the camera/scene can't be done easily by inspection
- Our camera is now at the origin; we need to align the ***u***, ***v***, ***w*** axes with the ***x***, ***y***, ***z*** axes
- This can be done by separate rotations about principle axes (as discussed in the Transformations lecture), but we are going to use a more direct (and simpler) approach
- Let's leave out the homogeneous coordinate for now
- Consider the standard unit vectors for the ***XYZ*** world coordinate system:

$$\boldsymbol{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \boldsymbol{e}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \boldsymbol{e}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

- We need to rotate ***u*** into $\boldsymbol{e}_1$ and ***v*** into $\boldsymbol{e}_2$ and ***w*** into $\boldsymbol{e}_3$
- Need to find some composite matrix $\boldsymbol{R}_{rot}$ such that:
  - $\boldsymbol{R}_{rot}\boldsymbol{u} = \boldsymbol{e}_1$ $\boldsymbol{R}_{rot}\boldsymbol{v} = \boldsymbol{e}_2$ $\boldsymbol{R}_{rot}\boldsymbol{w} = \boldsymbol{e}_3$

View Volume Rotation (2/3)

- How do we find R_{rot}? Let's manipulate the equations to make the problem easier and first find R_{rot}^{-1}. After multiplying on both sides by R_{rot}^{-1}, we get:
  - $u = R_{rot}^{-1} e_1$
  - $v = R_{rot}^{-1} e_2$
  - $w = R_{rot}^{-1} e_3$
- Recall that this means exactly that u is the first column of R_{rot}^{-1}, v is the second column, and w is the third column
- Therefore, we have

$$R_{rot}^{-1} = \begin{bmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{bmatrix}$$

View Volume Rotation (3/3)

- Now we just need to invert $\boldsymbol{R}_{rot}^{-1}$
- We know that the axes ***u***, ***v***, and ***w*** are orthogonal, and since they are unit vectors, they are also orthonormal
- This makes $\boldsymbol{R}_{rot}^{-1}$ an orthonormal matrix (its columns are orthonormal vectors). This means that its inverse is just its transpose (we proved this in the Transformations lecture!)
- Therefore, in non-homogeneous coordinates:

$$\boldsymbol{R}_{rot} = \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix} \xrightarrow{\text{homogeneous}} \boldsymbol{R}_{rot} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling the View Volume

- Now we have a view volume sitting at the origin, oriented upright with ***look*** pointing down the ***–z*** axis
- But the size of our volume has not met our specifications yet
- We want the *(x, y)* bounds to be at -1 and 1 and we want the far clipping plane to be at *z = -1*
- Given *width*, *height*, and far clipping plane distance, *far*, of a parallel view volume, our scaling matrix $\boldsymbol{S}_{xyz}$ is:

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

- Now all vertices post-clipping are bounded in between planes *x = (-1, 1)*, *y = (-1, 1)*, *z = (0, -1)*

The Normalizing Transformation (parallel) and Re-Homogenization

- Now have a complete transformation from an arbitrary parallel view volume to canonical parallel view volume
  - First translate P_n (center of near plane) to origin using translation matrix $\boldsymbol{T_{trans}}$
  - Then align ***u, v, w*** axes with ***x, y, z*** axes using rotation matrix $\boldsymbol{R_{rot}}$
  - Finally, scale view volume using scaling matrix $\boldsymbol{S_{xyz}}$
- Composite normalizing transformation is simply $\boldsymbol{S_{xyz}R_{rot}T_{trans}}$

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

- Since each individual transformation results in $w = 1$, no division by w to re-homogenize is necessary!

Notation

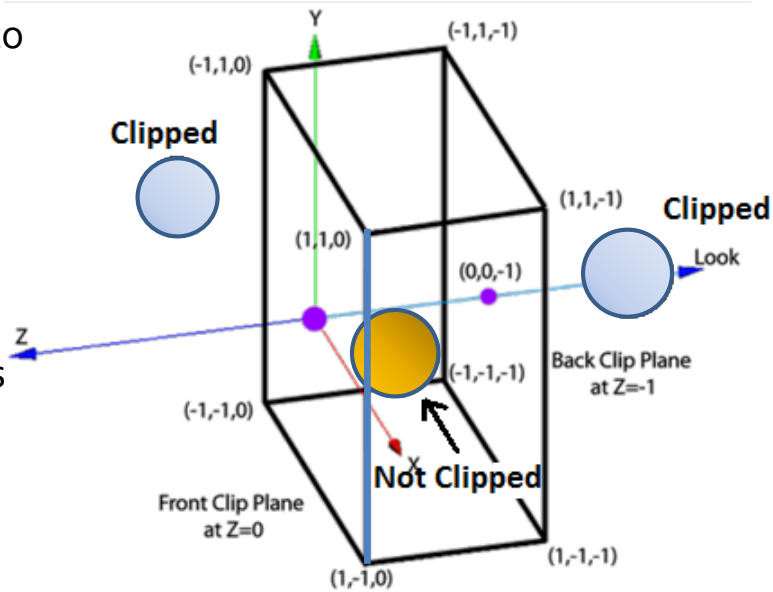
- The book groups all of these three transformations together into one transformation matrix
- For the parallel case, we will call it $\boldsymbol{M}_{orthogonal}$
- For the perspective case, which we will get to next, it is called $\boldsymbol{M}_{perspective}$
- For ease of understanding, we split all three up but they can be represented more compactly by the following, where $\boldsymbol{N}$ is the 3x3 matrix representing rotations and scaling:

$$\boldsymbol{N} = \begin{bmatrix} 2/width & 0 & 0 \\ 0 & 2/height & 0 \\ 0 & 0 & 1/far \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix}$$

$$\boldsymbol{M}_{orthogonal} = \begin{bmatrix} & & & -P_{n_x} \\ & \boldsymbol{N} & & -P_{n_y} \\ & & & -P_{n_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Clipping Against the Parallel View Volume

- Before returning to original goal of projecting scene onto film plane, how do we clip?
- With arbitrary view volume, the testing to decide whether a vertex is in or out is done by solving simultaneous equations
- With canonical view volume, clipping is much easier: after applying normalizing transformation to all vertices in scene, anything that falls outside the bounds of the planes $x = (-1, 1)$, $y = (-1, 1)$, and $z = (0, -1)$ is clipped
- Primitives that intersect the view volume must be partially clipped
- Most graphics packages, such as OpenGL, will do this step for you



Note: Clipping edges that intersect the boundaries of view volume is another step explored in clipping lecture

Projecting in the Normalized View Volume

- So how do we project the scene in this normalized view volume onto the *(x, y)* plane, where the film plane is now located? (film plane can be anywhere, having it at origin makes arithmetic easy
- To project a point *(x, y, z)* onto the *(x, y)* plane, just get rid of the *z* coordinate! We can use the following matrix:

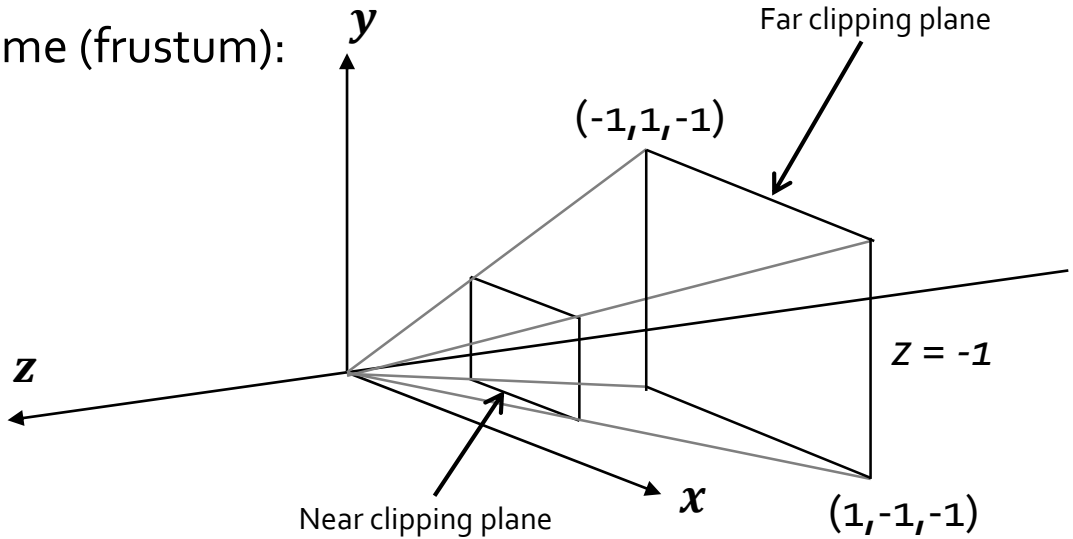
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix}$$

- Most graphics packages will also handle this step for you

Next: The Perspective View Volume

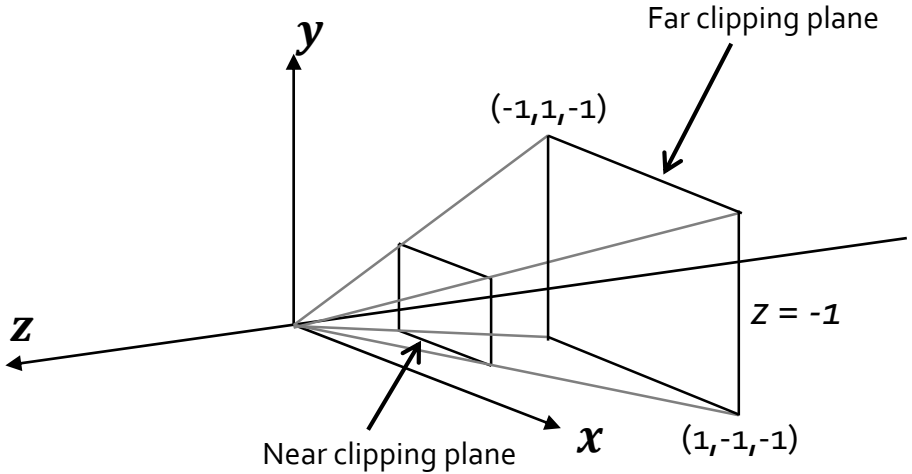
- Need to find a transformation to turn an arbitrary perspective view volume into a canonical (unit) perspective view volume

Canonical view volume (frustum):



Properties of the Canonical Perspective View Volume

- Sits at origin:
  - *Position = (0, 0, 0)*
- Looks along negative *z*-axis:
  - ***look*** vector = *(0, 0, -1)*
- Oriented upright
  - ***up*** vector = *(0, 1, 0)*
- Near and far clipping planes
  - Near plane at *z = c = -near/far* (we'll explain this)
  - Far plane at *z = -1*
- Far clipping plane bounds:
  - *(x, y)* from *-1* to *1*
- Note: the perspective canonical view volume is just like the parallel one except that the "film/projection" plane is more ambiguous here; we'll finesse the question by transforming the normalized frustum into the normalized parallel view volume before clipping and projection!

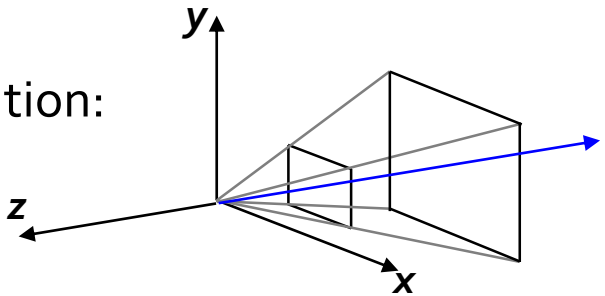


Translation and Rotation

- For our normalizing transformation, the first two steps are the same
  - The translation matrix T_{trans} is even easier to calculate this time, since we are given the point P_o to translate to origin. We use the same matrix R_{rot} to align camera axes:

$$T_{trans} = \begin{bmatrix} 1 & 0 & 0 & -P_{0_x} \\ 0 & 1 & 0 & -P_{0_y} \\ 0 & 0 & 1 & -P_{0_z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R_{rot} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Our current situation:

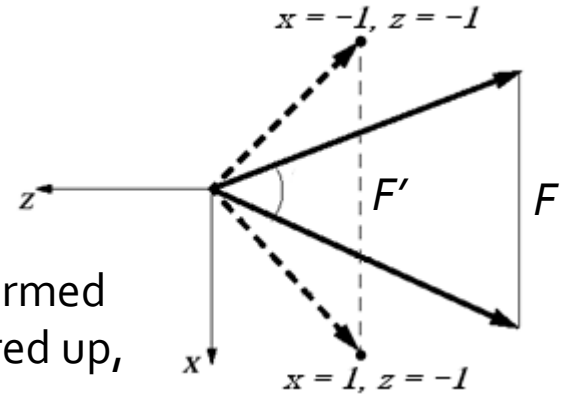


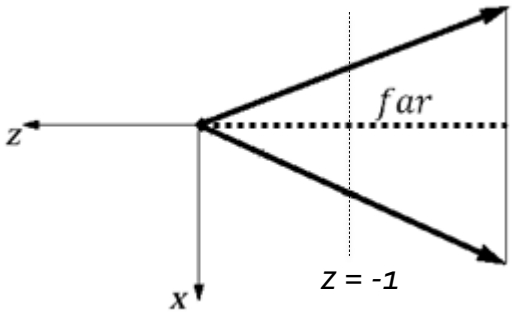
Scaling

- For perspective view volumes, scaling is more complicated and requires some trigonometry
- Easy to scale parallel view volume if we know width and height
- Our definition of frustum, however, doesn't give these two values, only θ_w and θ_h
- We need a scaling transformation $\boldsymbol{S_{xyz}}$ that:
  - Finds width and height of far clipping plane based on width angle θ_w, height angle θ_h, and distance *far* to the far clipping plane
  - Scales frustum based on these dimensions to move far clipping plane to $z = -1$ and to make corners of its cross section move to ±1 in both x and y
- Scaling position of far clipping plane to $z = -1$ remains same as parallel case, since we are still given *far*; however, unlike parallel case, near plane not mapped to $z = 0$

Scaling the Perspective View Volume (1/4)

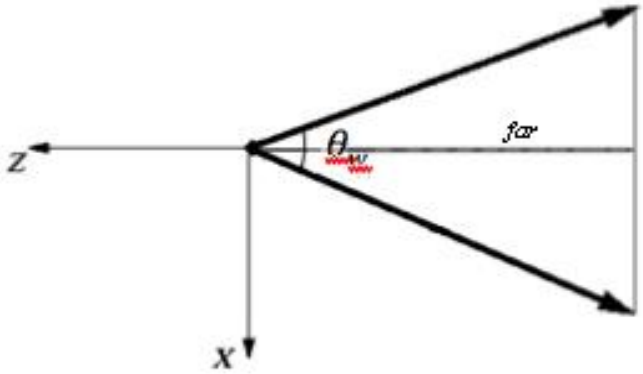
- Top-down view of the perspective view volume with arbitrary rectangular cross-section:
- Goal: scale the original volume so the solid arrows are transformed to the dotted arrows and the far plane's cross-section is squared up, with corner vertices at (±1, ±1, -1)
  - i.e., scale the original (solid) far plane cross-section *F* so it lines up with the canonical (dotted) far plane cross-section *F'* at *z* = -1
- First, scale along ***z*** direction
  - Want to scale so far plane lies at *z* = -1
  - Far plane originally lies at *z* = -*far*
  - Multiply by *1/far*, since –*far*/*far* = -1
  - So, $Scale_z = 1/far$

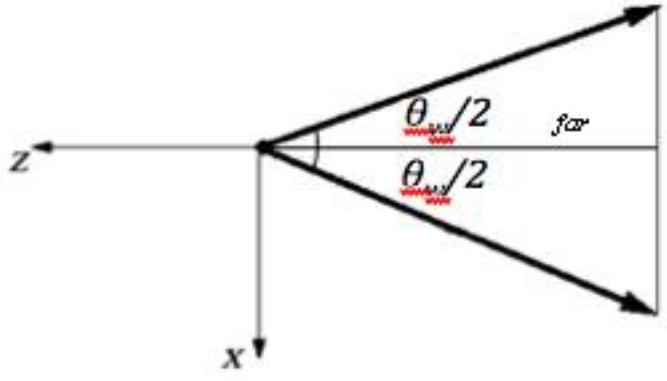




Scaling the Perspective View Volume (2/4)

- Next, scale along ***x*** direction
  - Use the same trick: divide by size of volume along the ***x***-axis
- How long is the (far) side of the volume along ***x***? Find out using trig…
  - Start with the original volume
  - Cut angle in half along the ***z***-axis

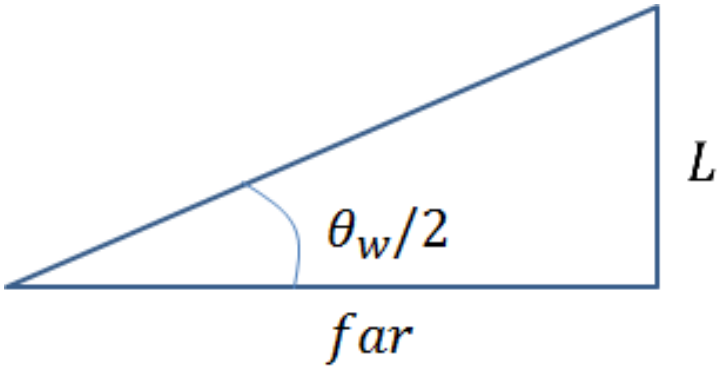




Scaling the Perspective View Volume (3/4)

- Consider just the top triangle
- Note that *L* equals the *x*-coordinate of a corner of the perspective view volume's cross-section at *far*. Ultimately want to scale by *1/L* to make *L* --> *1*

$$\frac{L}{far} = \tan\left(\frac{\theta_w}{2}\right) \rightarrow L = far \tan\left(\frac{\theta_w}{2}\right)$$



- Thus

$$Scale_x = \frac{1}{far \tan\left(\frac{\theta_w}{2}\right)}$$

Scaling the Perspective View Volume (4/4)

- Finally, scale along ***y*** direction
  - Use the same trig as in ***x*** direction, but use the height angle instead of the width angle:

$$Scale_y = \frac{1}{far \tan\left(\frac{\theta_h}{2}\right)}$$

- Together with the ***x***- and ***z***-scale factors, we have:

$$\boldsymbol{S}_{xyz} = \begin{bmatrix} 1/far\tan(\theta_w/2) & 0 & 0 & 0 \\ 0 & 1/far\tan(\theta_h/2) & 0 & 0 \\ 0 & 0 & 1/far & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The Normalizing Transformation (perspective)

- Our current perspective transformation takes on the same form as the parallel case:
  - Takes the camera's position and moves it to the world origin
  - Takes the ***look*** and ***up*** vectors and orients the camera to look down the –***z*** axis
  - Scales the view volume so that the far clipping plane lies on ***z***=-1 plane, with corners are at (±1, ±1, -1)

$$\boldsymbol{S}_{xyz}\boldsymbol{R}_{rot}\boldsymbol{T}_{trans} = \begin{bmatrix} 1/far\tan(q_w/2) & 0 & 0 & 0 \\ 0 & 1/far\tan(q_h/2) & 0 & 0 \\ 0 & 0 & 1/far & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_{0_x} \\ 0 & 1 & 0 & -P_{0_y} \\ 0 & 0 & 1 & -P_{0_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Multiplying any point *P* by this matrix, the resulting point *P'* will be the normalized version
- The projected scene will still look the same as if we had projected it using the arbitrary frustum, since same composite is applied to all objects in the scene, leaving the camera-scene relationship invariant.

Notation

- We can represent this composite matrix as $\boldsymbol{M}_{perspective}$ by the following:

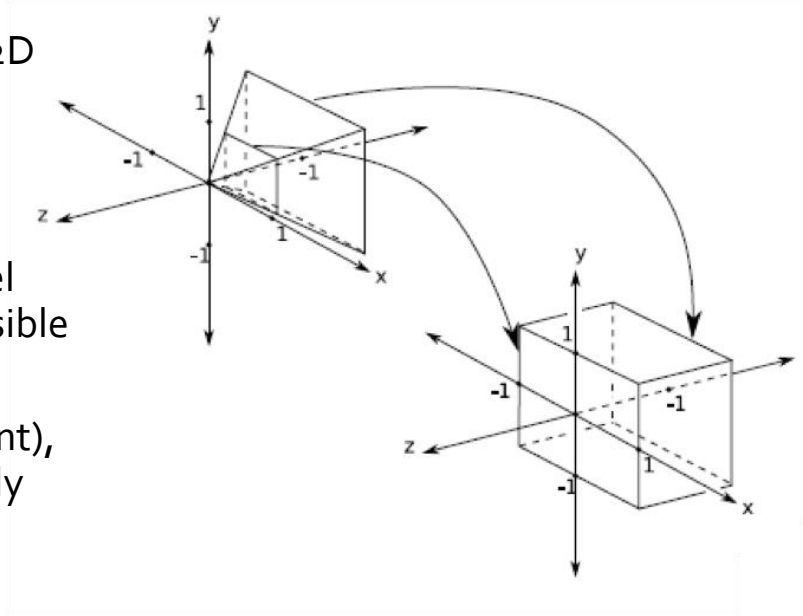
$$\boldsymbol{M}_{perspective} = \begin{bmatrix} & & & -P_{0_x} \\ & \boldsymbol{N} & & -P_{0_y} \\ & & & -P_{0_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Here, $\boldsymbol{N}$ is the 3x3 matrix representing rotations and scaling

$$\mathbf{N} = \begin{bmatrix} 1/far\tan(q_w/2) & 0 & 0 \\ 0 & 1/far\tan(q_h/2) & 0 \\ 0 & 0 & 1/far \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix}$$

Perspective and Projection

- Now we have our canonical perspective view volume
- However, projecting a perspective view volume onto a 2D plane is more difficult than it was in the parallel case
- Again: reduce it to a simpler problem!
- The final step of our normalizing transformation – transforming the perspective view volume into a parallel one – will preserve *relative* depth, which is crucial for Visible Surface Determination, i.e, the occlusion problem
- Simplifies not only projection (just leave off ***z*** component), but also clipping and visible surface determination – only have to compare z-values (z-buffer algorithm)
- Performs crucial perspective foreshortening step
- Think of this perspective-to-parallel transformation ***pp*** as the *unhinging transformation*, *represented by matrix* $\boldsymbol{M_{pp}}$



Effect of Perspective Transformation on Near Plane (1/2)

- Previously we transformed perspective view volume to canonical position, orientation, and size
- We'll see in a few slides that $\boldsymbol{M}_{pp}$ leaves the far clip plane at $z=-1$, and its cross-section undistorted, with corners at ± 1; *all other cross-sections will be "perspectivized"*
- Let's first look how $\boldsymbol{M}_{perspective}$ maps a particular point on the original near clipping plane lying on ***look*** (we denote the normalized ***look*** vector by ***look'***):

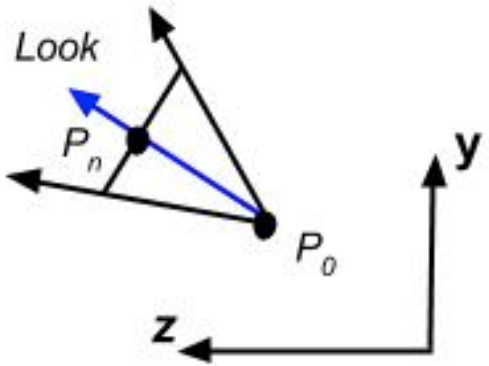
$$P_n = P_0 + near * look'$$

- It gets moved to a new location:

$$P_n' = \boldsymbol{S}_{xyz}\boldsymbol{R}_{rot}\boldsymbol{T}_{trans}P_n$$

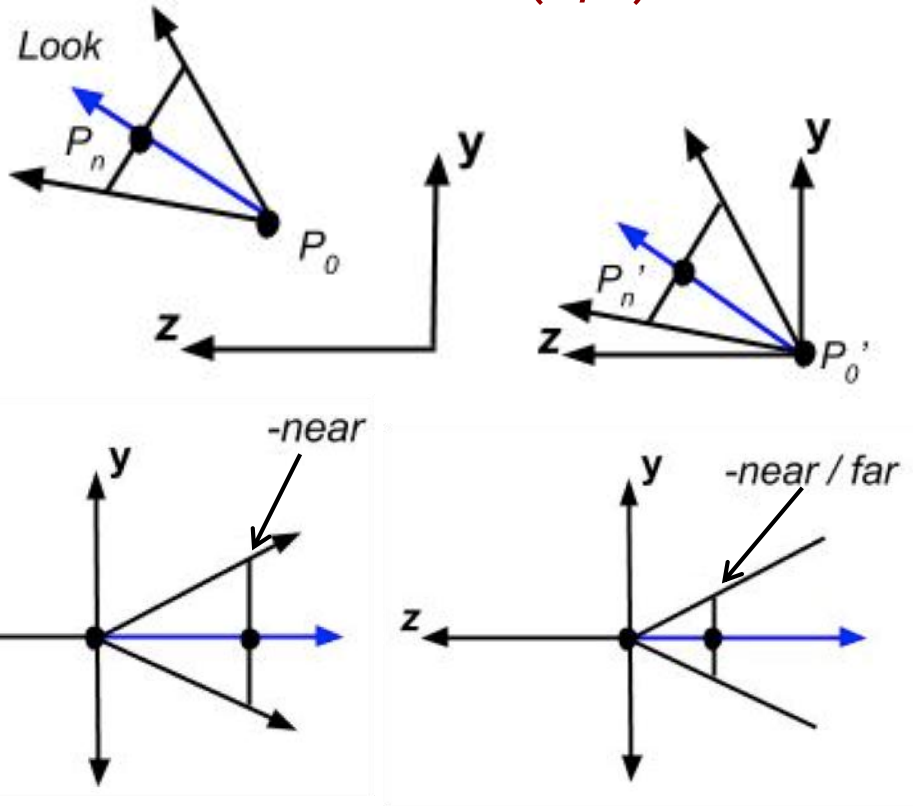
- On the negative z-axis, say:

$$P_n' = \begin{pmatrix} 0 & 0 & c \end{pmatrix}$$



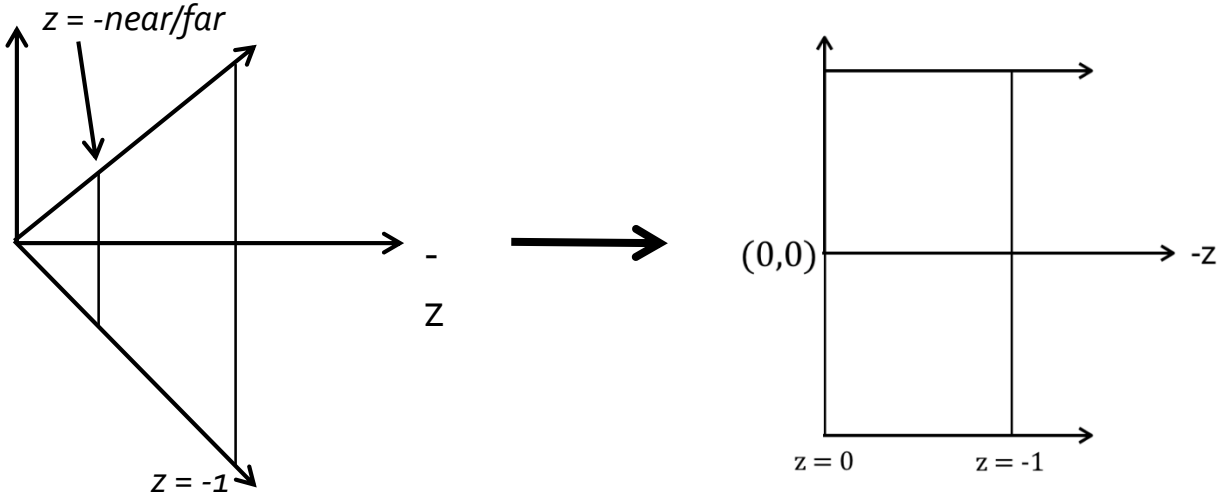
Effect of Perspective Transformation on Near Plane (2/2)

- What is the value of *c*? Let's trace through the steps.
- P_o first gets moved to the origin
- The point P_n is then distortion-free (rigid-body) rotated to *–near**z*
- The ***xy*** scaling has no effect, and the far scaling moves P_n to *(-near/far)**z*, so *c= -near/far*



Unhinging Canonical Frustum to Be a Parallel View Volume(1/4)

- Note from figure that far clipping plane cross-section is already in right position with right size
- Near clipping plane at *–near/far* should transform to the plane *z=o*



Unhinging Canonical Frustum to Be a Parallel View Volume (2/4)

- The derivation of our unhinging transformation is complex. Instead, we will give you the matrix and show that it works by example ("proof by vigorous assertion/demonstration")
- Our unhinging transformation perspective to parallel matrix, $\boldsymbol{M_{pp}}$

$$\boldsymbol{M_{pp}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+c} & \frac{-c}{1+c} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- Remember, c = *-near/far*

Unhinging View Volume to Become a Parallel View Volume(3/4)

- Our perspective transformation does the following:
  - Sends all points on the $z = -1$ far clipping plane to themselves
    - We'll check top-left (-1, 1, -1, 1) and bottom-right (1, -1, -1, 1) corners
  - Sends all points on the $z = c$ near clipping plane onto the $z = 0$ plane
    - Note that the corners of the square cross section of the near clipping plane in the frustum are $(\pm c, \pm c, c, 1)$ because it is a rectangular (square) pyramid w/ 45° planes
    - We'll check to see that top-left corner $(c, -c, c, 1)$ gets sent to (-1, 1, 0, 1) and that top-right corner $(-c, c, c, 1)$ gets sent to (1, -1, 0, 1)
  - Let's try $c = -1/2$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+c} & \frac{-c}{1+c} \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

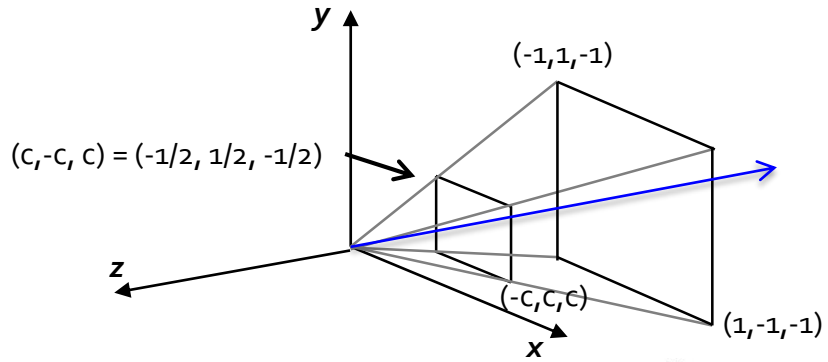
Unhinging View Volume to Become a Parallel View Volume(4/4)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \end{bmatrix}$$

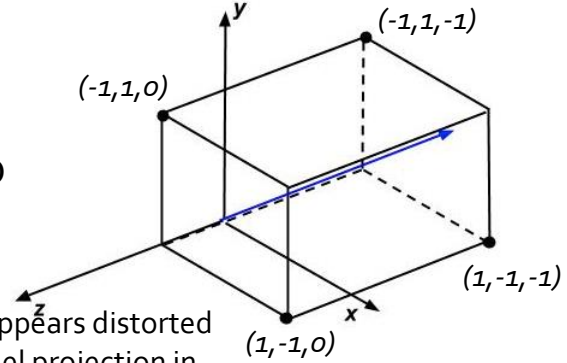
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} -1/2 \\ 1/2 \\ -1/2 \\ 1 \end{bmatrix} = \begin{bmatrix} -1/2 \\ 1/2 \\ 0 \\ 1/2 \end{bmatrix} \xrightarrow{\text{hom.}} \begin{bmatrix} -1 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 1/2 \\ -1/2 \\ -1/2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1/2 \\ -1/2 \\ 0 \\ 1/2 \end{bmatrix} \xrightarrow{\text{hom.}} \begin{bmatrix} 1 \\ -1 \\ 0 \\ 1 \end{bmatrix}$$



Don't forget to homogenize!



Note: Diagram appears distorted because of parallel projection in illustration, but center of front clipping plane is at origin

Perspective Foreshortening Affects *x*, *y*, and *z* Values

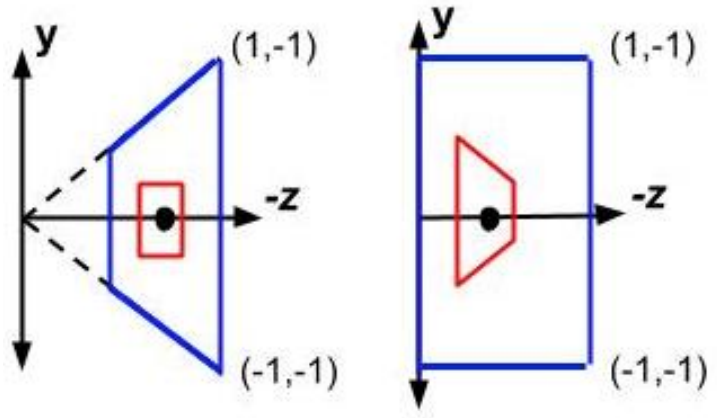
- The closer a *z* value is to 0 (the CoP/eye/camera), the larger the *x* and *y* values scale up
  - This is pre-clipping – if x and y exceed 1, they'll be clipped, hence the utility of the near plane which prevents such unnecessary clipping and obscuration of rest of scene

$$M_{pp}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+c} & \frac{-c}{1+c} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ \frac{z-c}{1+c} \\ -z \end{bmatrix} \xrightarrow{\text{hom.}} \begin{bmatrix} -x/z \\ -y/z \\ \frac{c-z}{z+zc} \\ 1 \end{bmatrix}$$

- We'll look at the effects on *z* (*z*-compression) on slide 44-46

Practical Considerations: *z*-buffer for Visible Surface Determination

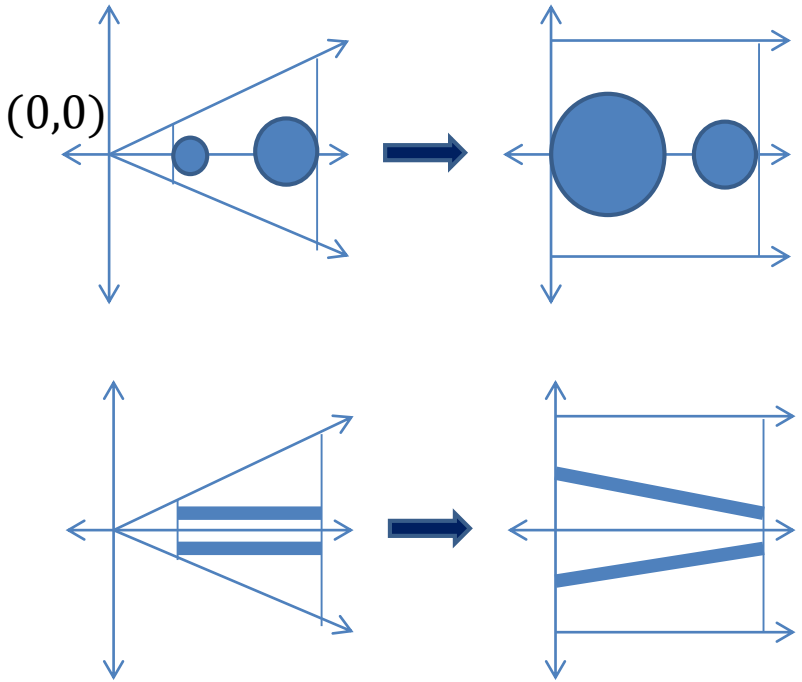
- Cross-sections inside view volume are scaled up the closer they are to near plane to produce perspective foreshortening, and *z* values decrease, but relative ***z***-distances and therefore order are preserved
- Depth testing using a ***z***-buffer (aka depth-buffer) that stores normalized ***z***-values of points compares a point on a polygon being rendered to one already stored at the corresponding pixel location – only relative distance matters
- ***z***-buffer uses alternate form of M_{pp} that does the same unhinging as the original but negates the ***z***-term to make the volume point down the positive ***z***-axis (use this one in camtrans lab, since we use a ***z***-buffer)
- The post-clipping range for these *z*-values is [0.0,1.0], where 0.0 corresponds to the *near* plane, and 1.0 to *far*



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+c} & \frac{-c}{1+c} \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{-1}{1+c} & \frac{c}{1+c} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

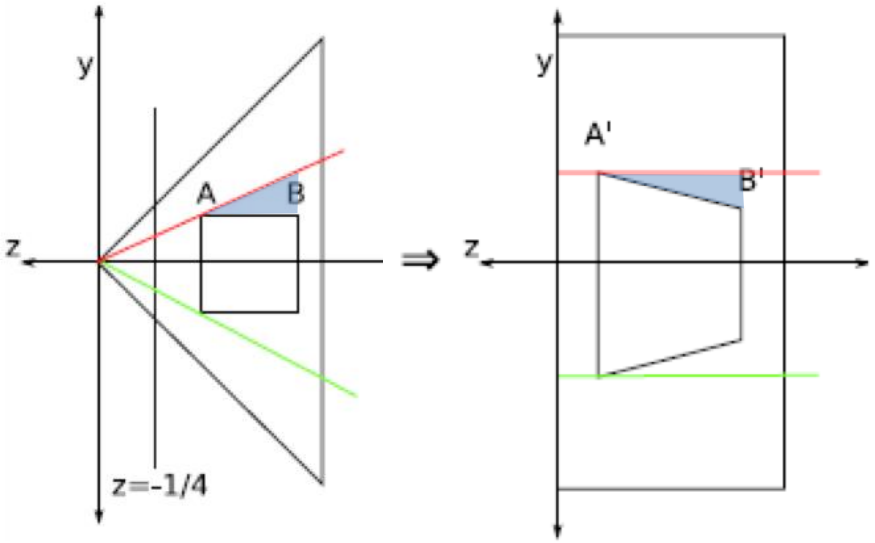
Why Perspective Transformation Works (1/2)

- Take an intuitive approach to see this
  - The closer the object is to the near clipping plane, the more it is enlarged during the unhinging step
  - Thus, closer objects are larger and farther away objects are smaller, as expected
- Another way to see it is to use parallel lines
  - Draw parallel lines in a perspective volume
  - When unhinge volume, lines fan out at *near* clipping plane
  - Result is converging lines, e.g., railroad track coinciding at vanishing point



Why Perspective Transformation Works(2/2)

- Yet another way to demonstrate how this works is to use occlusion (when elements in scene are blocked by other elements)
- Looking at top view of frustum, see a square
- Draw a line from eye point to left corner of square - points behind this corner are obscured by square
- Now unhinge perspective and draw a line again to left corner - all points obscured before are still obscured and all points that were visible before are still visible



The Normalizing Transformation (perspective)

- We now have our final normalizing transformation; call it to convert an arbitrary perspective view volume into a canonical parallel view volume

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+c} & \frac{-c}{1+c} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 1/far\tan(q_w/2) & 0 & 0 & 0 \\ 0 & 1/far\tan(q_h/2) & 0 & 0 \\ 0 & 0 & 1/far & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_{0_x} \\ 0 & 1 & 0 & -P_{0_y} \\ 0 & 0 & 1 & -P_{0_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Remember to homogenize points after you apply this transformation – does the perspective!
- Clipping and depth testing have both been simplified by transformation (use simpler bounds checking and trivial z-value comparisons resp.)
- Additionally, can now project points to viewing window easily since we're using a parallel view volume: just omit ***z***-coordinate! Avoids having to pick a film/projection plane
- Then map contents to viewport using window-to-viewport mapping (windowing transformation)

The Windowing Transformation (1/2)

- The last step in our rendering process after projecting is to resize our clip rectangle on the view plane, i.e., the cross-section of the view volume, to match the dimensions of the viewport
- To do this, we want to have a viewing/clipping window with its lower left corner at *(o,o)* and with width and height equal to those of the viewport
- This can be done using the windowing transformation – derived on next slide

$$\boldsymbol{M}_{wind} = \begin{bmatrix} width & 0 & 0 & 0 \\ 0 & height & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1/2 & 0 & 0 & 1/2 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

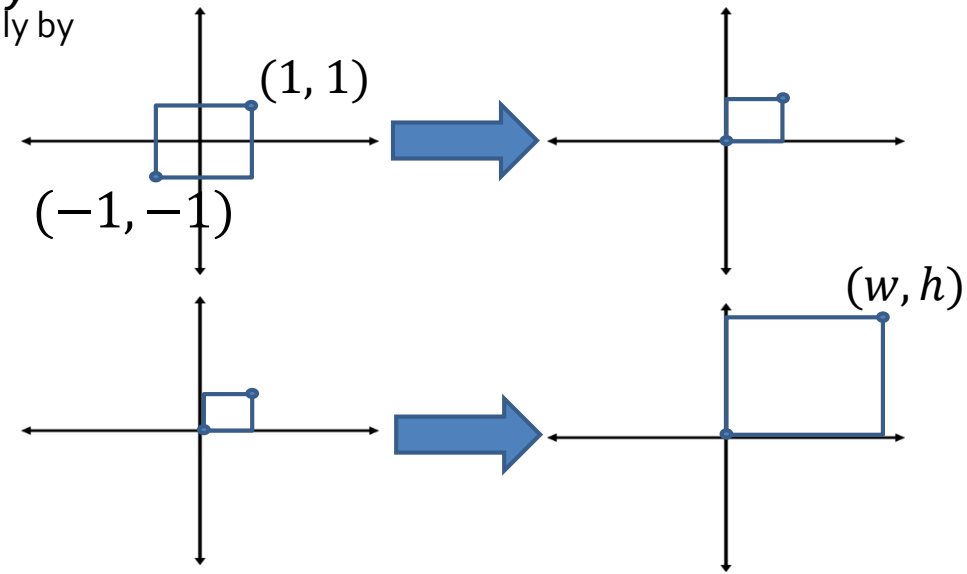
The Windowing Transformation (2/2)

- First translate viewing window by 1 in both the ***x*** and ***y*** directions to align with origin and then scale uniformly by ½ to get proper unit size:

$$\begin{bmatrix} 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1/2 & 0 & 0 & 1/2 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Then scale viewing window by width and height of viewport to get our desired result

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



- Finally, translate the viewing window to be located at the origin of the viewport (any part of the screen)
- Can confirm this matches more general windowing transformation in Transformations lecture -- handled by most graphics packages

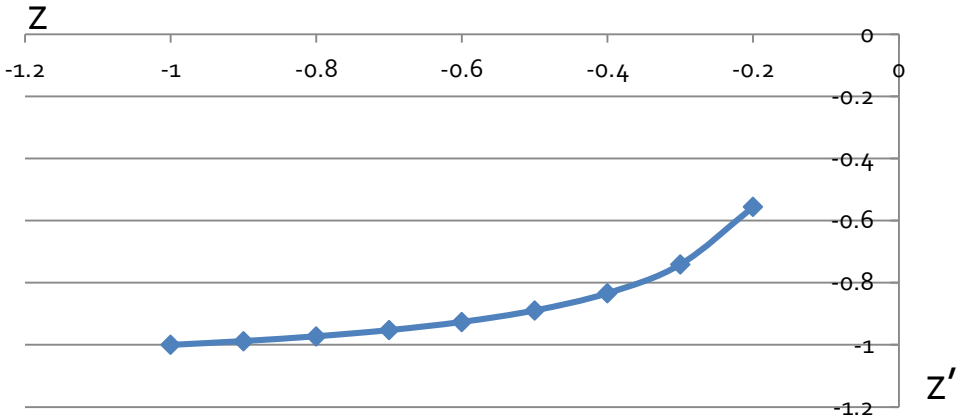
Perspective Transform Causes **z**-Compression (1/3)

- Points are compressed towards the far clipping plane
- Let's look at the general case of transforming a point by the unhinging transformation:

$$M_{pp}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+c} & \frac{-c}{1+c} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ \frac{z-c}{1+c} \\ -z \end{bmatrix} \xrightarrow{\text{hom.}} \begin{bmatrix} -x/z \\ -y/z \\ \frac{c-z}{z+zc} \\ 1 \end{bmatrix}$$

- First, note that *x* and *y* are both shrunk for $z > 1$ (perspective foreshortening increasing with increasing *z*) – corresponds to what we first saw with similar triangles explanation of perspective
- Now focus on new *z*-term, called *z'*. This represents new depth of point along ***z***-axis after normalization and homogenization
- $z' = (c-z)/(z+zc)$, now let's hold *c* constant and plug in some values for *z*
- Let's have *near* = 0.1, *far* = 1, so $c = -0.1$
- The following slide shows a graph of *z'* dependent on *z*

Perspective Transform Causes *z*-Compression (2/3)



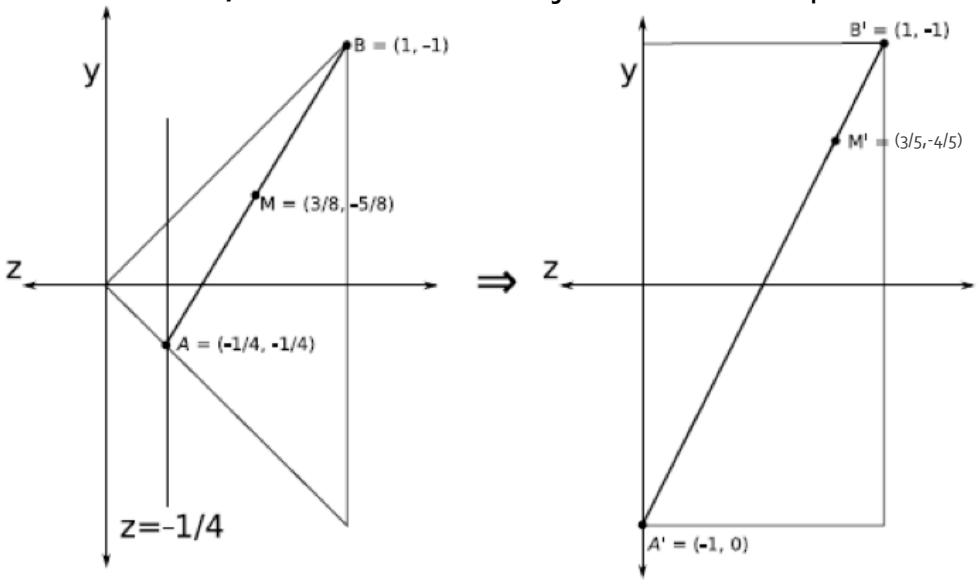
- We can see that the *z*-values of points are being compressed towards *z = -1* in our canonical view volume -- this compression is more noticeable for points originally closer to the far clipping plane
- Play around with near and far clipping planes: observe that as you bring the near clipping plane closer to *z=o*, or extend the far clipping plane out more, the *z*-compression becomes more severe
- Caution: if *z*-compression is too severe, *z*-buffer depth testing becomes inaccurate near the back of the view volume and rounding errors can cause objects to be rendered out of order, i.e., "bleeding through" of pixels occurs (upcoming VSD lecture)

Perspective Transform Causes *Z*-Compression (3/3)

- It might seem tempting to place the near clipping plane at $z = o$ or place the far clipping plane very far away (maybe at $z = \infty$)
- First note that the value of $c = -near/far$ approaches o as either *near* approaches o or as *far* approaches ∞
- Applying this to our value of $z' = (c-z)/(z+zc)$, we substitute o for c to get $z' = -z/z = -1$
- From this, we can see that points will cluster at $z = -1$ (the far clipping plane of our canonical view volume) and depth-buffering essentially fails

Aside: Projection and Interpolation(1/3)

- This converging of points at the far clipping plane also poses problems when trying to interpolate values, such as the color between points
- Say for example we color the midpoint *M* between two vertices, *A* and *B*, in a scene as the average of the two colors of *A* and *B* (assume we're looking at a polygon edge-on or just an edge of a tilted polygon)
- Note that if we were just using a parallel view volume, it would be safe to just set the midpoint to the average and be done
- Unfortunately, we can't do this for perspective transformations since the point that was originally the midpoint gets compressed towards the far clipping plane. It isn't the actual midpoint anymore.
- Another way to say this is that the color *G* does not interpolate between the points linearly anymore. We can't just assign the new midpoint the average color.



Aside: Projection and Interpolation(2/3)

- However, while *G* does not interpolate linearly, *G/w* does, where *w* is the homogenous coordinate after being multiplied by our normalizing transformation, but before being homogenized
  - In our case *w* will always be *-z*
- Knowing this, how can we find the color at this new midpoint?
- When we transform *A* and *B*, we get two *w* values, w_A and w_B
- We also know the values of G_A and G_B
- If we interpolate linearly between G_A / w_A and G_B / w_B (which in this case is just taking the average), we will know the value for the new midpoint G_M / w_M
- We can also find the average of $1 / w_A$ and $1 / w_B$ and to get $1 / w_M$ by itself
- Dividing by G_M / w_M by $1 / w_M$, we can get our new value of G_M

Aside: Projection and Interpolation(3/3)

- Let's make this slightly more general
- Say we have a function *f* that represents a property of a point (we used color in our example)
- The point *P* between points *A* and *B* to which we want to apply the function is *(1-t)A + tB* for some $o \leq t \leq 1$. The scalar *t* represents the fraction of the way from point *A* to point *B* your point of interest is (in our example, $t = 0.5$)
- Goal: Compute *f(P)*. We know
  - $1 / w_t = (1-t) / w_a + t / w_b$
  - $f(P) / w_t = (1-t)f(A) / w_a + tf(B) / w_b$
- So to find the value of our function at the point specified by *t* we compute:
  - $f(P) = (f(P) / w_t) / (1 / w_t)$

Proof by Example

- Let's revisit the setup from this image:
- Say we want $f(A) = 0$, $f(B) = 1$, and thus $f(M) = 0.5$
- After unhinging transformation, the new midpoint, is 4/5 of the way from A' to B', which can be found with some algebra:
  - $M'_y = (1-t)A'_y + tB'_y$
  - $t = (M'_y - A'_y) / (B'_y - A'_y) = 4/5$
- Like $f(M)$, $f(M')$ should be 0.5. We check this below:
  - $w_a = 0.25$ and $w_b = 1$
  - $1/w_t = (1-0.8)*(1/0.25) + 0.8*(1/1) = 1.6$
  - $f(P)/w_t = (1-0.8)*(0/0.25) + 0.8*1/1 = 0.8$
  - $f(M') = (f(P)/w_t) / (1/w_t) = 0.5$