UCLA

CS174A : Introduction to Computer Graphics

Royce 190
TT 4-6pm

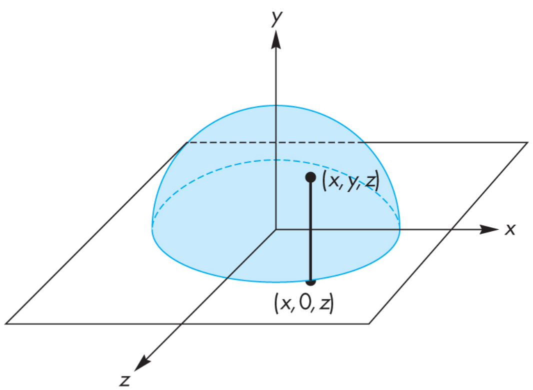
Scott Friedman, Ph.D
UCLA Institute for Digital Research and Education

Term Project

- Details will be introduced next Tuesday.
- Things to start considering now are:
  - Teams will be a minimum of *three* people.
    - Use the class forum to find partners if you need to.
  - Teams can have up to *five* people in your group.
  - Your team will submit a detailed project proposal.
    - "we are going to write a game", will not do for a proposal.
  - The proposals will be due the following week.
  - The TAs and I will review them and make comments.

Trackball

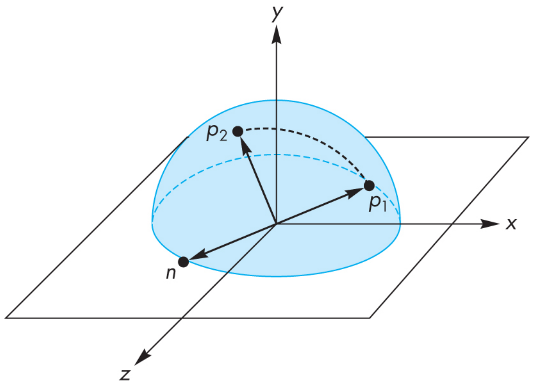
- Last time we talked about rotation…
  - A useful/intuitive interaction technique.
  - Plant a unit hemisphere into the x–z plane.
  - Using mouse positions we can reconstruct y
    - Because, $x^2 + y^2 + z^2 = 1$, $y = \sqrt{1 - x^2 - z^2}$



Trackball

- That's nice, we know *y*…
  - Now we can track, as the mouse moves, a position p_1 to p_2 on the surface of the hemisphere.
  - What we really want to do is rotate in the direction of the arc swept out from p_1 to p_2.
  - That axis of rotation can be found via the cross product of p_1 to p_2, the *normal*.

$$n = p_1 \times p_2$$



Trackball

- That's nice, we know n…
  - Conveniently, n can tell us the angle between p_1 to p_2 as well because we are using a *unit* hemisphere.

$$|\sin\theta| = |n|$$

  - Now we know an angle and a vector around which the rotation is supposed to occur.
  - The book mentions a nice trick when animating the rotation in small increments – and that is to recognize that for small
    - and you can avoid the inverse sine operation.

$$\theta \approx \sin\theta$$

Trackball

- A side note…
  - When animating a rotation in small increments
    - A lot of trigonometry is involved = slow
  - Helpful to recognize that for small $\theta \approx \sin\theta$
    - and then you can avoid the inverse sine operation.
  - This implies that, *for small angles* we can use
    - Another graphics "trick"

$$R = R_z(\psi)R_y(\phi)R_x(\theta) \approx \begin{bmatrix} 1 & -\psi & \phi & 0 \\ \psi & 1 & -\theta & 0 \\ -\phi & \theta & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Trackball

- That's nice, we know the *angle*…
  - But, we just know how to rotate around *x*, *y* and *z*!
  - Yes, but if we align the rotation vector with, say, the *z*-axis we perform our rotation. *Simple!* ☺

$$R = R_x(-\theta_x)R_y(-\theta_y)R_z(\theta_z)R_y(\theta_y)R_x(\theta_x)$$

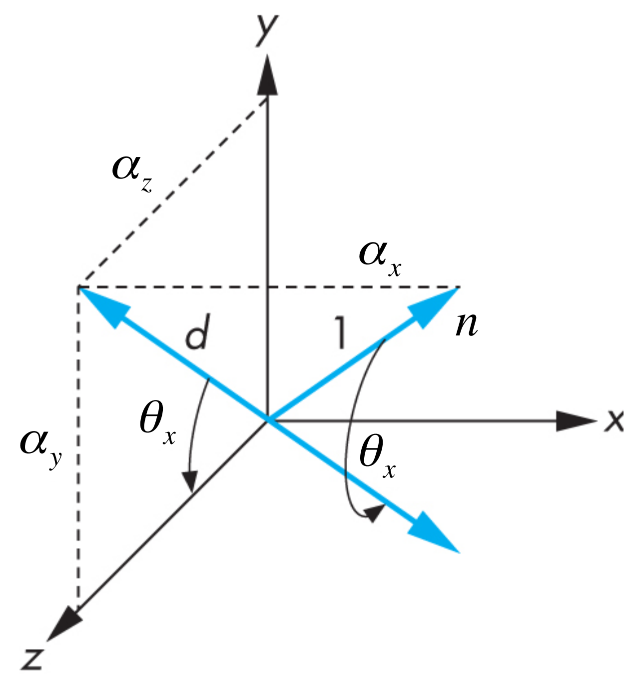
  - Ugh, we don't know $R_x(\theta_x)$ or $R_y(\theta_y)$ even if we decided that $R_z(\theta_z)$ was the rotation we wanted.
  - Yes, but let's understand what is going on first.

Trackball

- How can we find the *x* and *y* rotations?
  - First we need to rotate around the *x*-axis onto the *x*-*z* plane.
  - Recall that $\cos\theta_x = \alpha_x$
  - Then,

$$d = \sqrt{\alpha_y^2 + \alpha_z^2}$$

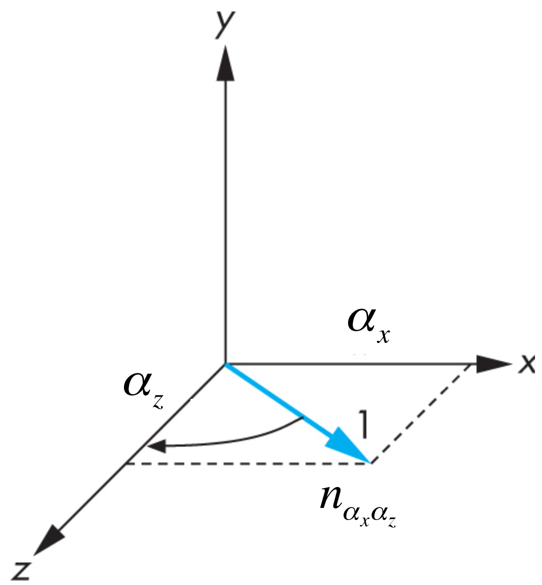
$$R_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \alpha_z / d & -\alpha_y / d & 0 \\ 0 & \alpha_y / d & \alpha_z / d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Trackball

- Now we need the *y* rotation?
  - We can follow the same process
  - Again, recall that $\cos\theta_y = \alpha_y$
  - Then,

$$R_y(\theta_y) = \begin{bmatrix} \alpha_z & 0 & -\alpha_x & 0 \\ 0 & 1 & 0 & 0 \\ \alpha_x & 0 & \alpha_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Trackball

- Actually, we can now rotate about *any* vector v fixed at a point p.
  - Concatenating into M

$$M = T(p)R_x(-\theta_x)R_y(-\theta_y)R_z(\theta_z)R_y(\theta_y)R_x(\theta_x)T(-p)$$

  - Now we can rotate our trackball vector!
  - Our point p is the **origin** and our vector is n.
  - This is a lot of work – is there a better way?

Quaternions

- Same result with less computation.
    - A quaternion has the form $q = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$.
    - The terms $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ are imaginary.
        - Fortunately, we can ignore this fact in this class.
        - But, they are what ultimately make quaternions work.
    - Lets consider them this way $q(w,x,y,z)$
        - Lets make w the rotation
        - Lets make x, y and z be the rotation vector.

Quaternions

- Same result with less computation.
  - It is ***critical*** that q be ***normalized***, *i.e.* $q=|q^2|$.
    - Or this does not work.
  - The resulting rotation matrix is

$$Q = \begin{bmatrix} 1-2(y^2+z^2) & 2(xy-wz) & 2(xz+wy) & 0 \\ 2(xy+wz) & 1-2(x^2+z^2) & 2(yz-wx) & 0 \\ 2(xz-wy) & 2(yz+wx) & 1-2(x^2+y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Quaternions

- Same result with less computation.
  - A very nice feature of quaternions is that they allow for straightforward smooth interpolation.
    - You do this with a current rotation R and an increment R_I
    - R starts with some initial/or no rotation and rotation vector
    - R_I has a small rotation and the same rotation vector.
    - Need to *renormalize* R occasionally to keep computation stable.

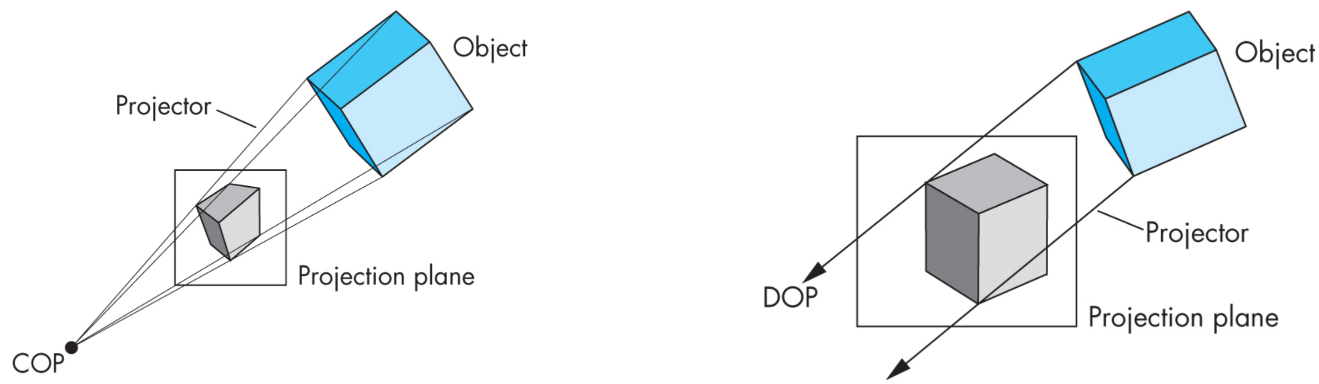
$$R = RR_I$$

Viewing

- We are going to concern ourselves with two types of *viewing*.
    - Perspective viewing
    - Parallel viewing

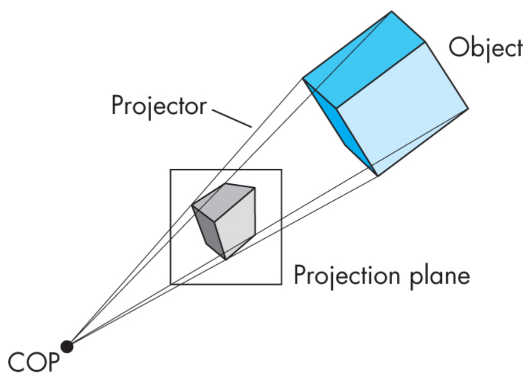
Viewing

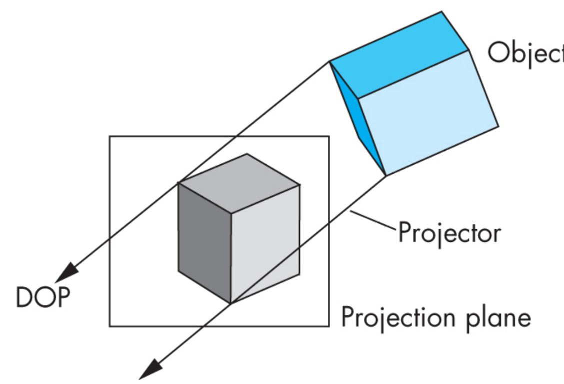
- In both cases we have
  - Objects,
  - Projection lines
  - Projection plane
  - Eye (center of projection)



Viewing

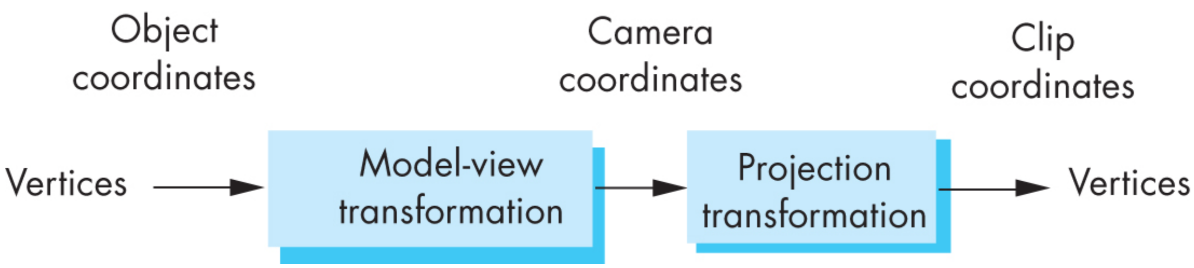
- Our goal is to
  - Use transformations to project the vertices of objects onto the projection plane.
  - Specifically we will create transformations to go from object to camera to clip coordinates.





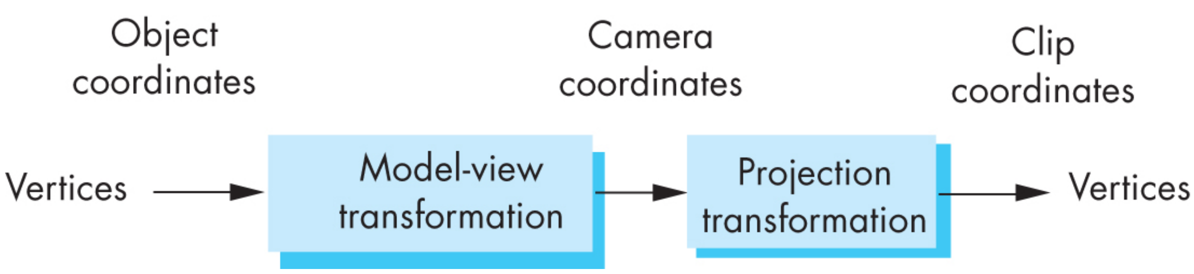
Viewing

- Previously
  - We used the default canonical view volume.
    - Which exists in clip coordinates, i.e. (-1,1),(-1,1),(-1,1).
  - Last time we saw how transformations can be combined to bring objects into camera (world) coordinates
    - Collectively, the ***model-view transformation***.



Viewing

- Model-view transformation
  - Does not take us all the way to clip coordinates.
  - we need a ***projection transformation*** for that.
  - Model-view gets objects in front of the camera, potentially.
  - A Projection defines which and how those objects will appear on the screen.



Instancing

- Useful for Assignment #2:
  - Take a single geometric object.
    - Clone it with only transformations (and possibly state).
  - Define the geometry of a cube (once).
    - Instance the cube by setting a transformation and setting some state (e.g. color) and drawing it.
    - Instance another cube by setting another transformation and setting state(e.g. color and scale) and drawing the *same geometry*.

UCLA

Viewing

- Positioning the (getting things in from of the) "camera"

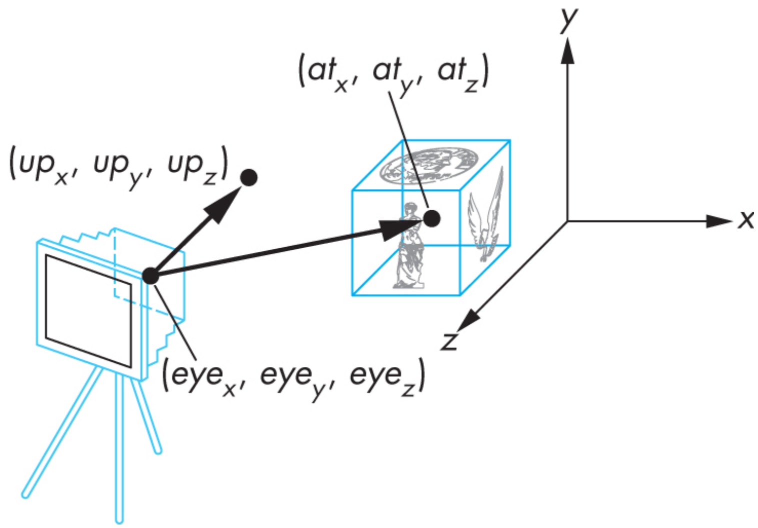
  – Recall that the default is "looking" down the $-z$ axis at the origin (0,0,0).
    - This is equivalent to model-view set to the identity matrix.

  – Remember, transformations are specified in *reverse*.
    - That means we specify the position of the camera first.

  – We are going to look at two methods
    – Look-at
    – Yaw, pitch and roll (euler angles)

Viewing

- Look-at
  - We define three terms
    - A point describing the location of the ***eye***.
    - A point the eye is looking ***at***.
    - An ***up*** direction for the camera.



Viewing

- Look-at
  - The ***at*** and ***eye*** points give us
    - the *view-plane-normal* or ***vpn***
  - the ***up*** vector is usually (0, 1, 0)
    - Or, (0, 1, 0, 0) in homogeneous coordinates!
  - We then calculate the following

$$vpn = at - eye \qquad u = \frac{up \times n}{|up \times n|}$$

$$n = \frac{vpn}{|vpn|} \qquad v = \frac{n \times u}{|n \times u|}$$

Viewing

- Look-at
  - Once ***u***, ***v*** and ***n*** are *normalized*
  - The following will position our camera

$$V = RT = \begin{bmatrix} u_x & u_y & u_z & -eye_x u_x - eye_y u_y - eye_z u_z \\ v_x & v_y & v_z & -eye_x v_x - eye_y v_y - eye_z v_z \\ n_x & n_y & n_z & -eye_x n_x - eye_y n_y - eye_z n_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

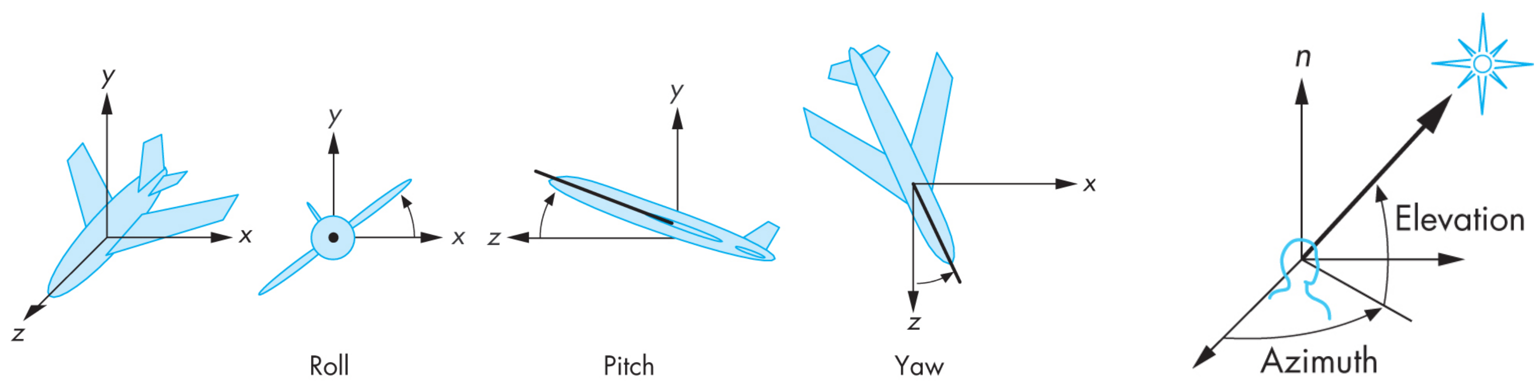
Viewing

- Look-at
  - Works reasonably well for positioning.
  - But not so well for moving the camera smoothly.

$$V = RT = \begin{bmatrix} u_x & u_y & u_z & -eye_x u_x - eye_y u_y - eye_z u_z \\ v_x & v_y & v_z & -eye_x v_x - eye_y v_y - eye_z v_z \\ n_x & n_y & n_z & -eye_x n_x - eye_y n_y - eye_z n_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Viewing

- Yaw, pitch and roll (like an airplane)
  - Here we, essentially, use polar coordinates.
  - A simplified version uses just *azimuth* and *elevation*.
    - Rotate camera in the direction we desire.
    - Translate camera to *eye* point.



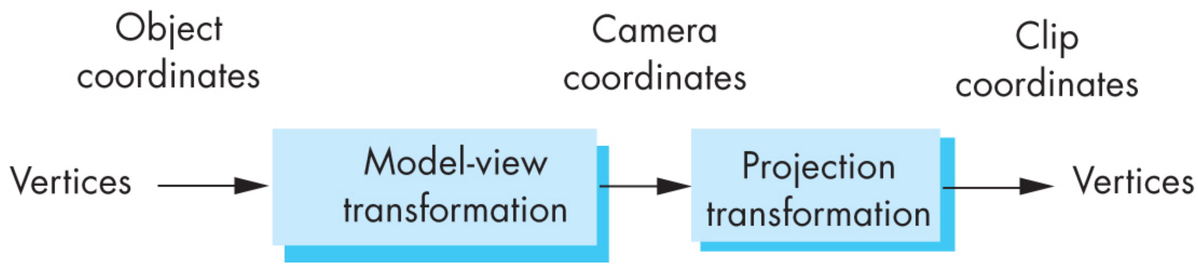
Viewing

- Yaw, pitch and roll
    - In reality we perform the *inverse* of what we want.
    - We are transforming world coordinates (all objects) into camera coordinates.
        - Move the world to the camera.
            - » That is, if I want to appear to rotate left 20 degrees.
            - » The transformation about the *y*-axis would be -20.
            - » Similarly, if I want to appear to move forward 5 units.
                - » I would transform everything by -5.
- So far we have only gotten things in front of the camera!

Viewing

- Projections – Parallel (orthographic)
  - Once in camera coordinates we need a projection transformation to get us to clip coordinates.
  - The transformation matrix that gives us an orthographic projection is:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

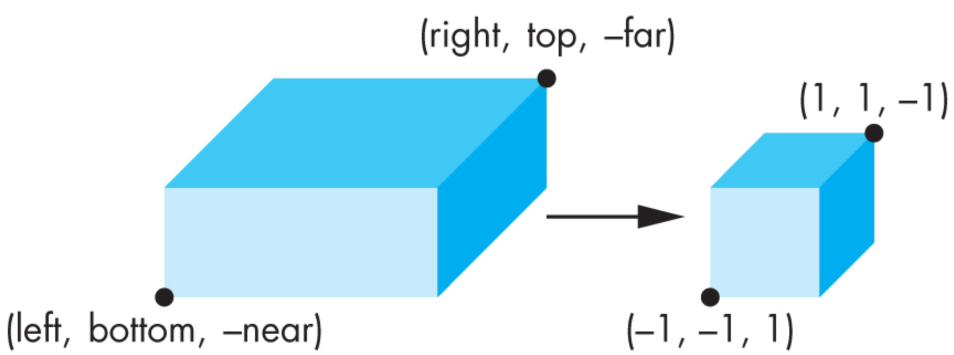


Viewing

- Projections – Parallel (orthographic)
  - However, M is applied in the hardware *after* the vertex shader.
    - Which is in clip coordinates
  - How do we “include” or “see” more of our scene?

Viewing

- Projections – Parallel (orthographic)
  - We *scale* what we want to "include" to fit within the canonical view volume. i.e. (-1,1),(-1,1),(-1,1)
  - OpenGL provides a function for this called
    - `glOrtho(`*left, right, bottom, top, near, far*`)`



Viewing

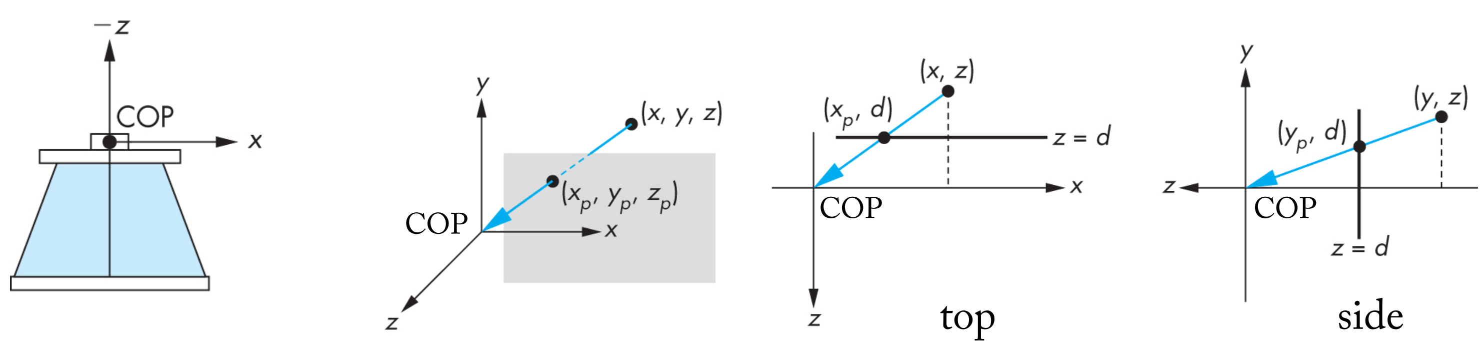
- Projections – Parallel (orthographic)
  - If you think about it `Ortho` contains a scale and translation.
  - Here is what the transformation matrix looks like.

$$N = ST = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{left+right}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Viewing

- Projections – Perspective
  - Basic symmetrical perspective projection
  - The point (*x, y, z)* is projected through the projection plane to the eye point (or center of projection COP)
  - We can compute the point of intersection with

$$x_p = \frac{x}{z/d}, \quad y_p = \frac{y}{z/d}$$



Viewing

- Projections – Perspective
  - The simple perspective projection matrix is

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

  - The important thing to notice here is the position of the *1/d* term.
  - This means our homogeneous coordinate, w, can be modified (will no longer be 1) when a vertex is multiplied by *M*.

Viewing

- Projections – Perspective
  - Uh oh, the homogeneous coordinate is no longer 1?

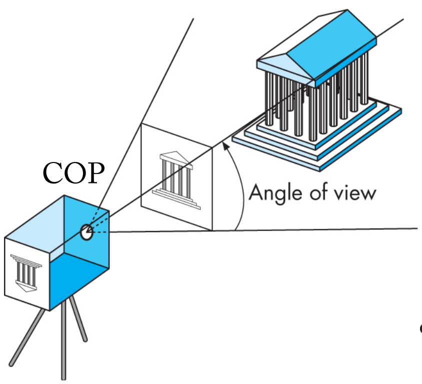
$$q = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

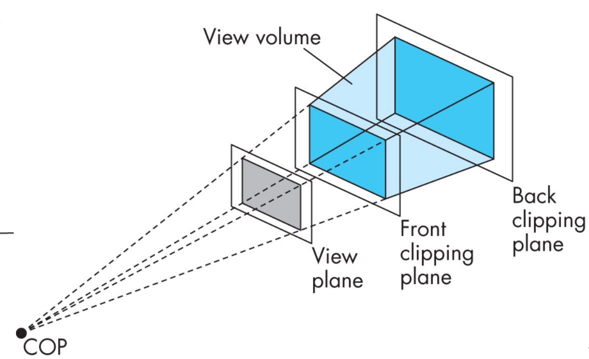
  - Not the end of the world, remember
    - We have to divide by the homogeneous coordinate to get back to 3D space.

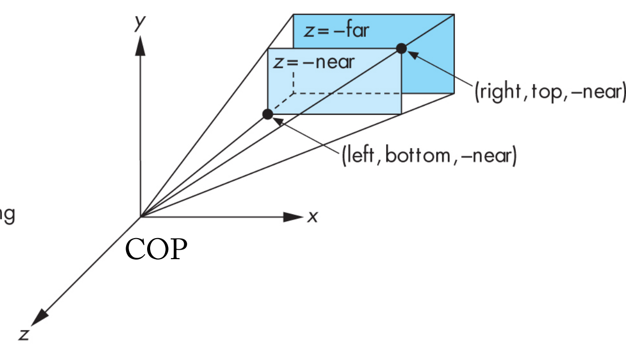
$$q' = \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix}$$

Viewing

- Projections – Perspective
  - That's nice, but only gets our points onto the projection plane.
  - We want a transformation into clip coordinates!
  - That requires not only the specification of a perspective projection, but of a view volume as well.

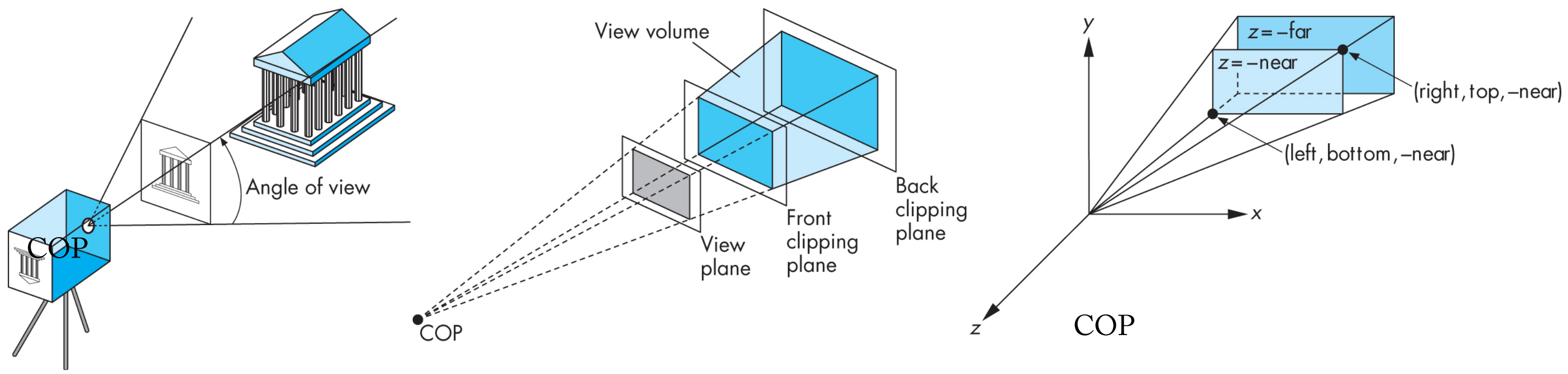






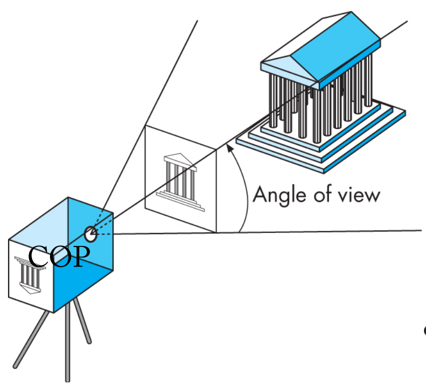
Viewing

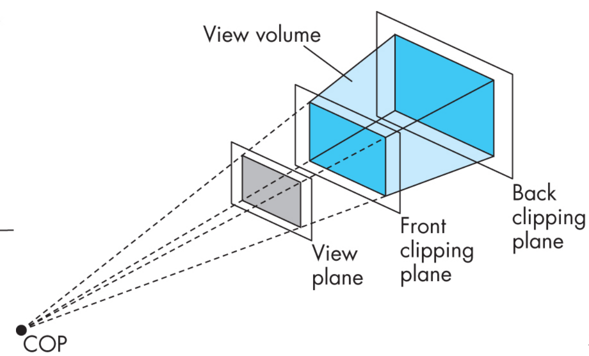
- Projections – Perspective
  - View volume is a pyramid with apex its at the COP.
  - Top and bottom are the near and far clip planes, respectively.
  - Notice that the view and clip planes do not necessarily need to be the same/coincident.

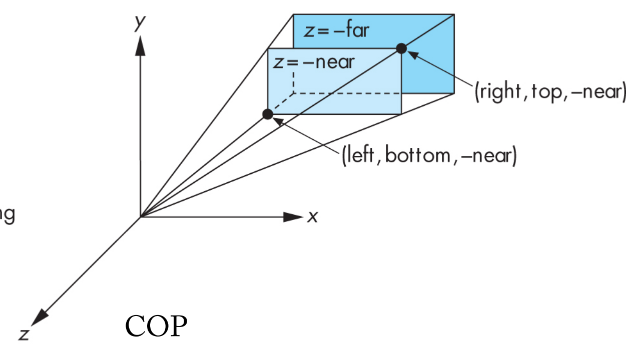


Viewing

- Projections – Perspective
  - OpenGL provides a function, similar to `Ortho`, called `Frustum` with the same parameters.
  - The edges specify the near clip plane.
  - The edges implicitly define the *angle of view* of the projection.

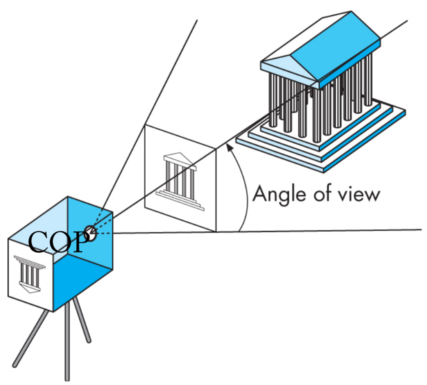


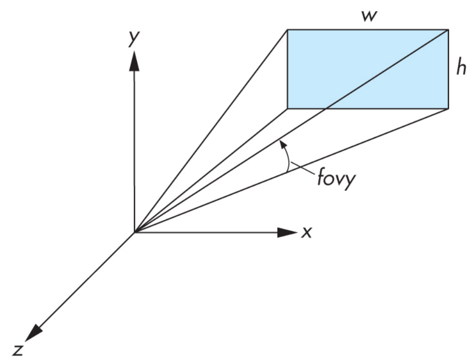




Viewing

- Projections – perspective
  - OpenGL provides a utility function
    - `Perspective(`*fovy, aspect, near, far*`)`
  - This form is sometimes more convenient to specify.
  - *Aspect* is the aspect ratio of the view volume.
    - *i.e. width / height*





Viewing

- Projections – perspective
  - Once again, what we had is a projection not the full transformation we need into clip coordinates.
  - The full matrix we do need is defined by:

$$P = NSH = \begin{bmatrix} \frac{right}{near} & 0 & 0 & 0 \\ 0 & \frac{near}{top} & 0 & 0 \\ 0 & 0 & \frac{-far+near}{far-near} & \frac{-2far \cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

UCLA

Viewing

- Projections – perspective
  - Matrices are passed to the vertex shader just like we have seen earlier.
  - Matrices are **uniform** variables where all the data parallel processors on the GPU will see the same value.

```
in vec4 vPosition;
uniform mat4 modelView;
uniform mat4 projection;

void main( )
{
    //
    // The perspective division actually happens to gl_Position immediately
    // after the vertex shader completes. i.e. divided by gl_Position.w
    //
    gl_Position = projection * modelView * vPosition;
}
```