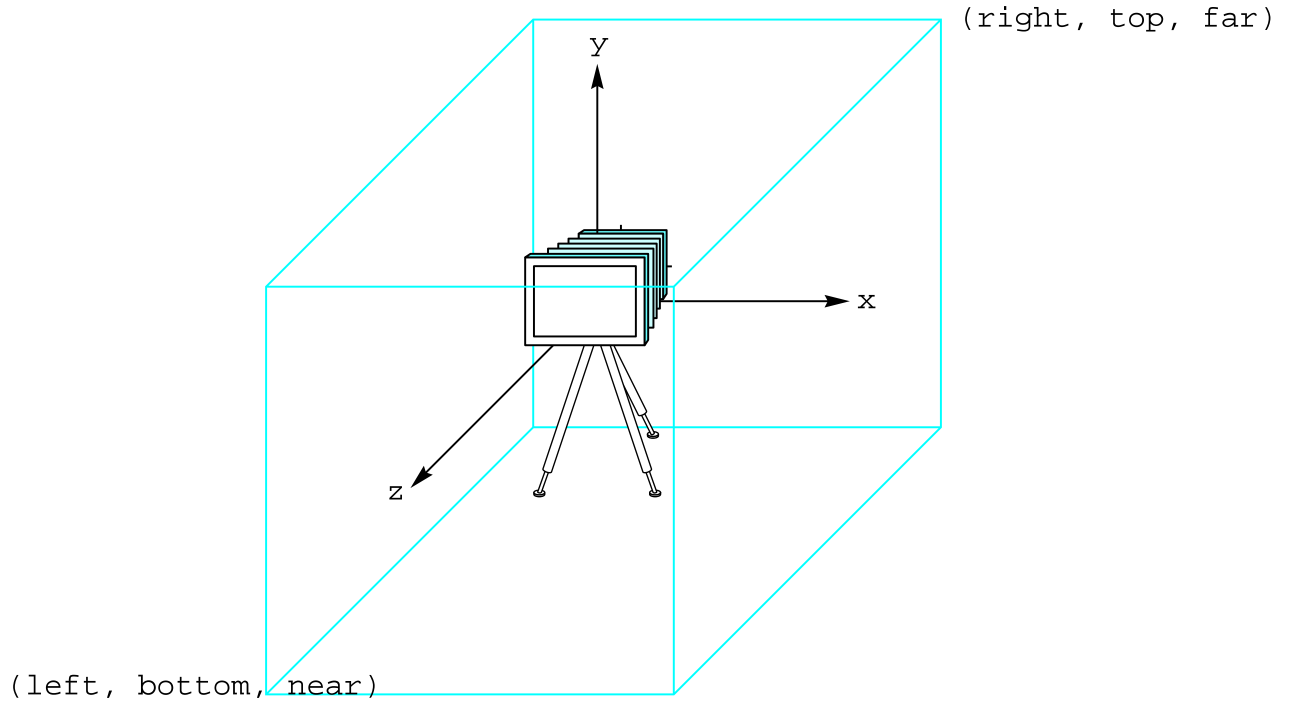
Viewing and projection:

- 3D world $\rightarrow$ 2D image
  - *viewing* transformations: camera position and direction
  - *perspective/orthographic* transformations: reduce 3D to 2D

Camera positioning: ideas

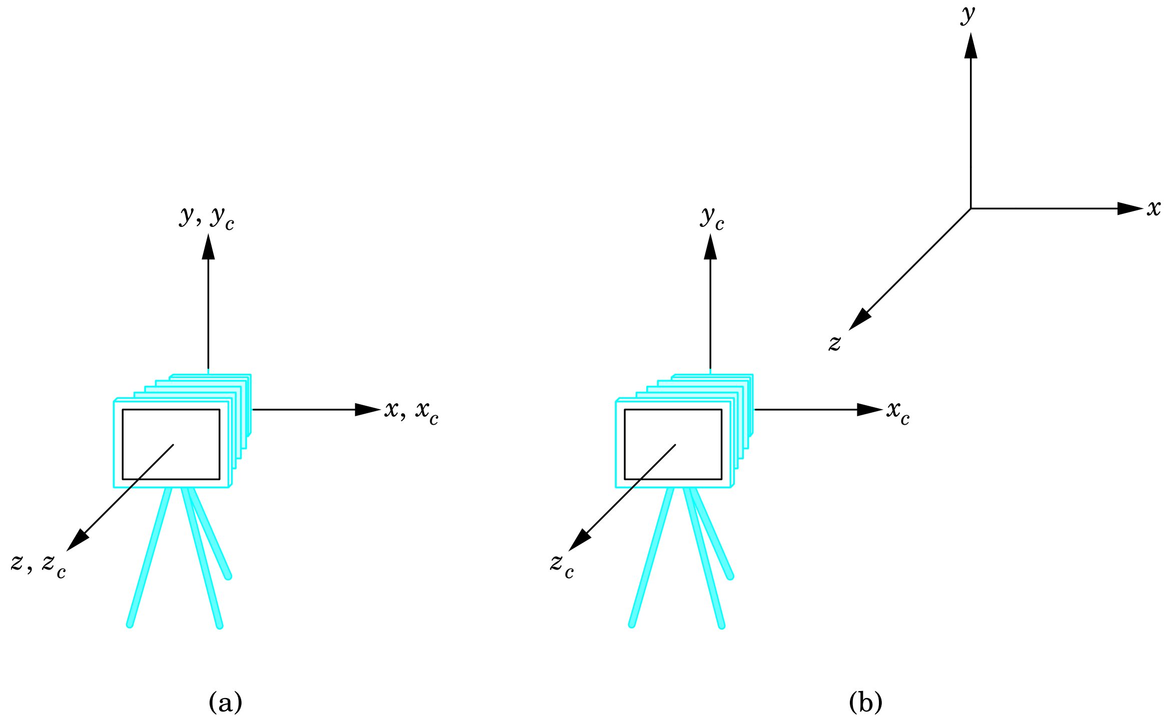
Camera default position is at the origin, facing in $-z$ direction:



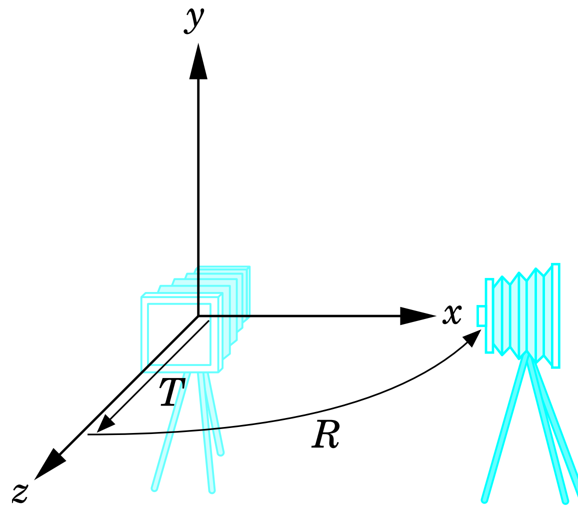
Viewing volume is a cube centered about the origin with sides of length 2.

Camera positioning: ideas, cont.

Use transformations on modelview matrix to move it around:



(a) (b)



Camera positioning: ideas, cont.

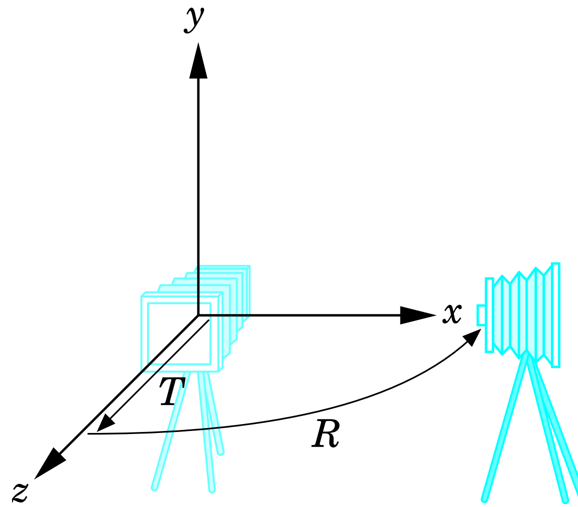
Remember that the modelview matrix is part of OpenGL's state, though!

If you specify a point p in the original frame and don't mess with the modelview matrix, that point has the same representation in both frames.

If you have the modelview matrix set to C and then specify a point q with `glVertex`, that point will be at q in the world frame and Cq in the camera frame.

(Recall that the modelview matrix encapsulates the relationship between the camera frame and the world frame, and contains all info needed to transform back and forth between the two coordinate systems.)

Camera positioning: implementation




```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0, 0.0, -d);
glRotatef(-90.0, 0.0, 1.0, 0.0);
```

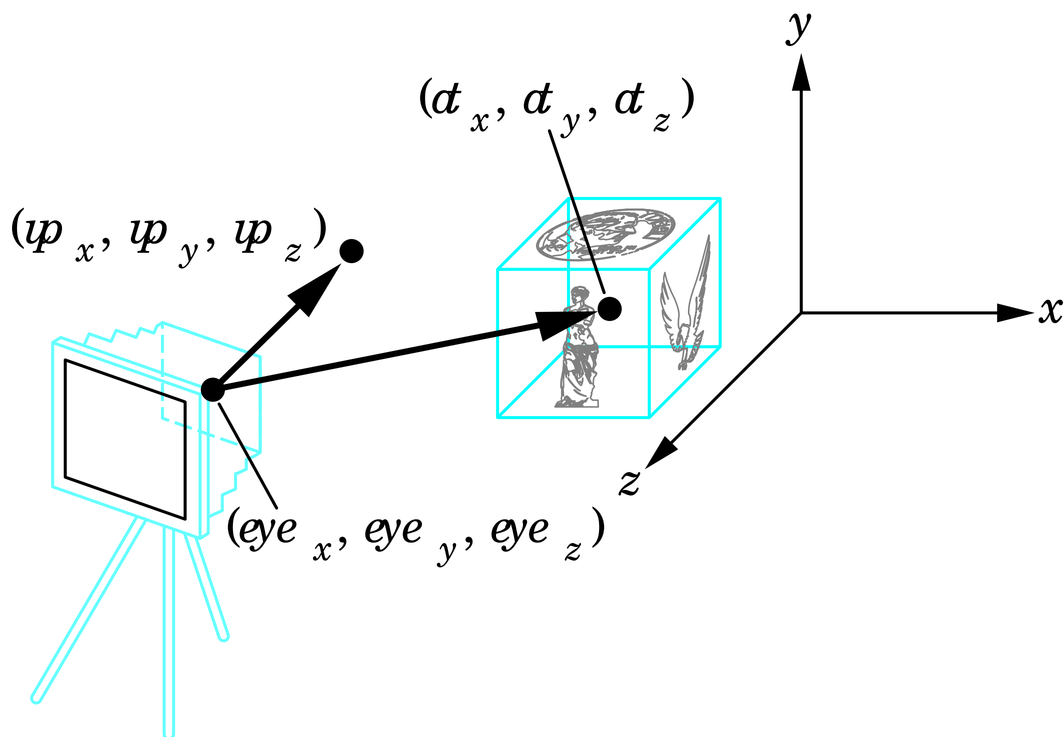
Things to note:

- order matters (as before)
- things may seem backwards (relativity)

Camera positioning: implementation, cont.

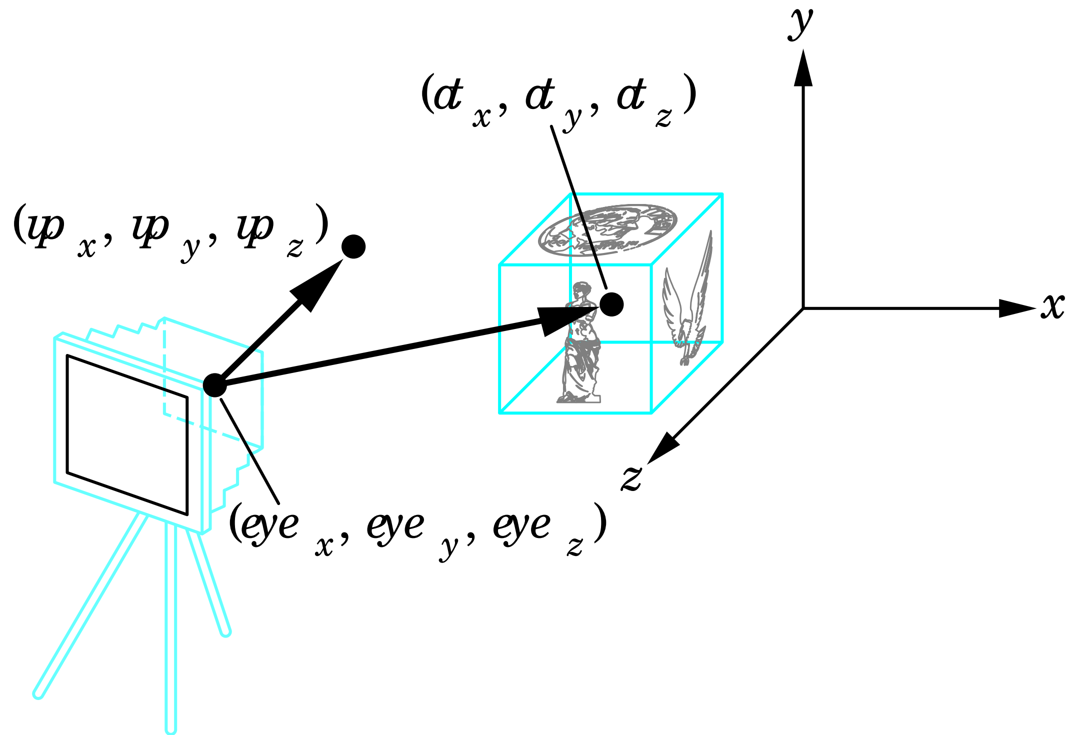
That's kind of a pain, so OpenGL encapsulates a useful set of stuff in the `gluLookAt` macro:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eyex, eyey, eyez,
          atx, aty, atz,
          upx, upy, upz);
```



Camera positioning: effects

The effects of changing **`eye`**, **`at`**, and **`up`** can be confusing, especially if you leave one fixed and have forgotten about it...



...and orthographic projection can exacerbate the confusion.

cubeview.c:

- look-at point fixed
- camera moves *in world frame*

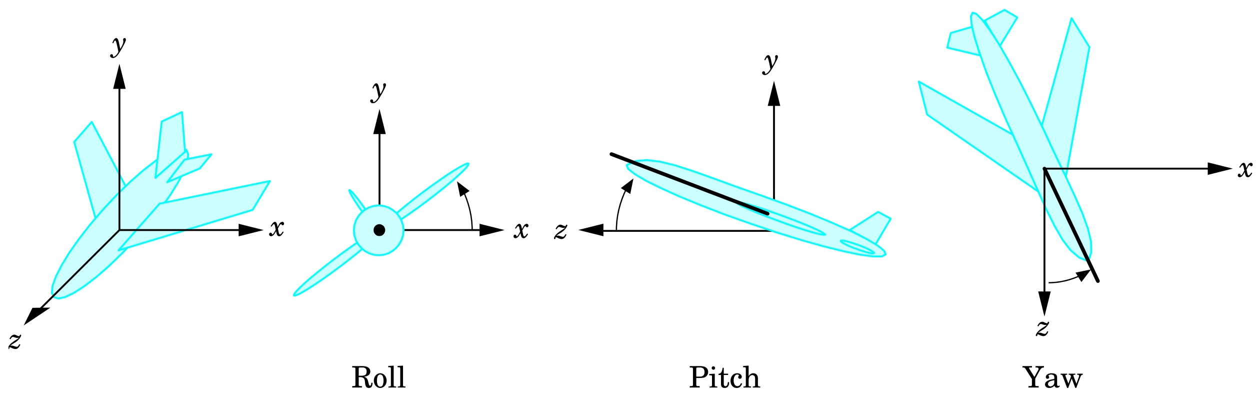
in display handler:

```
gluLookAt(viewer[0],viewer[1],viewer[2],
          0.0, 0.0, 0.0,
          0.0, 1.0, 0.0);
```

in keyboard handler:

```
   ...
   if(key == 'x') viewer[0]-= 1.0;
   if(key == 'X') viewer[0]+= 1.0;
   if(key == 'y') viewer[1]-= 1.0;
   ...
```

Roll, pitch, and yaw:



More complicated because they're *relative to the camera*

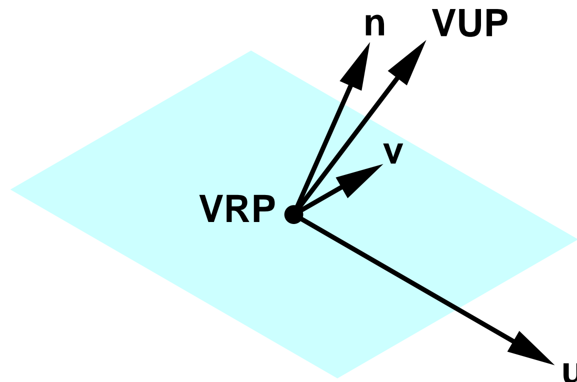
- What does roll change?
- What does pitch change?
- What does yaw change?

How PHIGS and GKS-3D do this:

A coordinate system that comprises:

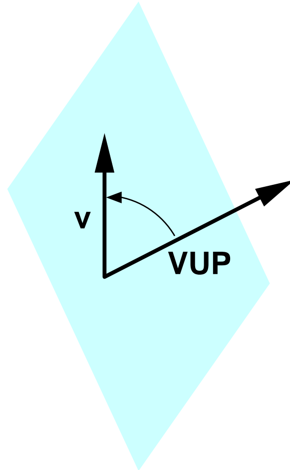
- view-reference point
- view-plane normal $\vec{n}$
- view-up vector

(all set explicitly by user)



Doing the math:

Note that the view-up vector is not constrained to lie in view plane, so have to project to set up the coordinate system:



Then take cross product of $\vec{v}$ and $\vec{n}$ to obtain a third orthogonal basis vector $\vec{u}$.

Normalize all three to make math easier: $\widehat{v}$, $\widehat{n}$, and $\widehat{u}$.

Doing the math, cont.:

Then do standard change-of-coordinates math to obtain a rotation matrix that moves points or vectors from the old frame into the camera frame:

$$M = \begin{bmatrix} u'_x & v'_x & n'_x & 0 \\ u'_y & v'_y & n'_y & 0 \\ u'_z & v'_z & n'_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

NB: use M^T, as in PS8.

Doing the math, cont.:

Also need to translate origin of camera frame to view-reference point (x, y, z): that is, a translation of $(-x, -y, -z)$.

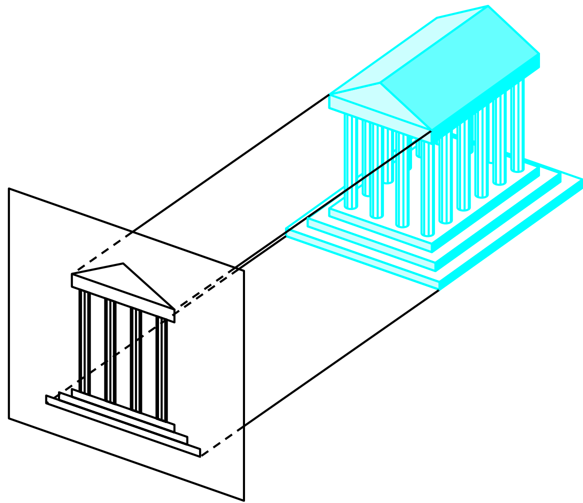
$$T = \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

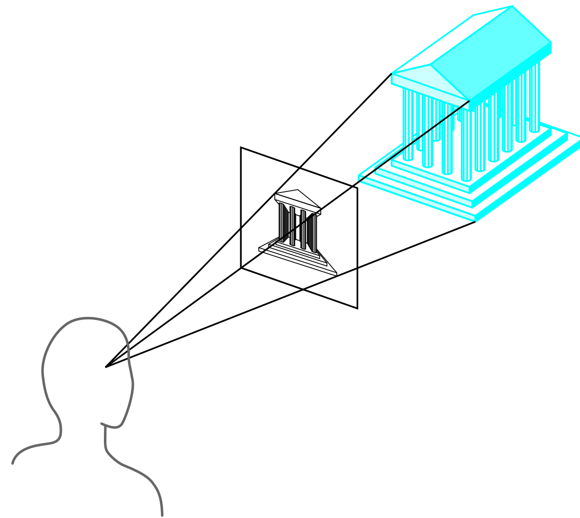
Do this by composing the rotation and translation matrices, producing a modelview matrix that looks like this:

$$M^T T = \begin{bmatrix} u'_x & u'_y & u'_z & -xu'_x - yu'_y - zu'_z \\ v'_x & v'_y & v'_z & -xv'_x - yv'_y - zv'_z \\ n'_x & n'_y & n'_z & -xn'_x - yn'_y - zn'_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

See section 5.3.2 for examples.

Projections – orthographic vs. perspective:





glOrtho **glFrustum**

Latter far more realistic; that's how our eyes work (finite focal length → nonuniform foreshortening).

Both non-invertible because they project multiple points in the world frame to the same screen pixel.

Perspective projection and w:

Before, said "for now, it's always 1.0"

Not always true. More generally, it's a *scaling factor* (as well as a 'placeholder' for the origin)

Now, represent the 3D point $(x, y, z)^T$ as:

$$P = \begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix}$$

As long as $w \neq 0$, we can recover the original point $(x, y, z)^T$ from this representation, right?

Perspective projection and w, cont.:

Can represent a broader class of transformations.

Consider this matrix:

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

...which transforms this point:

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

into this point:

$$Q = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

Perspective projection and w, cont.:

If you then re-normalize Q so that $w = 1$, you get:

$$
\begin{aligned}
x' &= \frac{x}{z/d} \\
y' &= \frac{y}{z/d} \\
z' &= \frac{z}{z/d} = d
\end{aligned}
$$

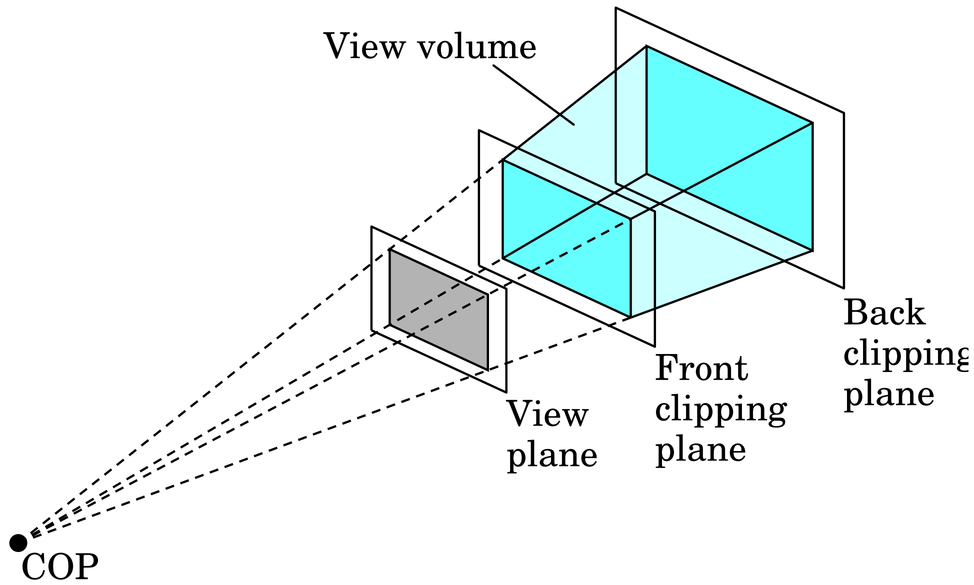
These are the equations for a standard perspective projection of Q!

(*viz.*, x and y get smaller as z gets larger, scaled by d)

Perspective division:

The "perspective division" inherent in the re-normalization is so common that it's part of the pipeline in many graphics APIs:

Specifying a perspective transformation:



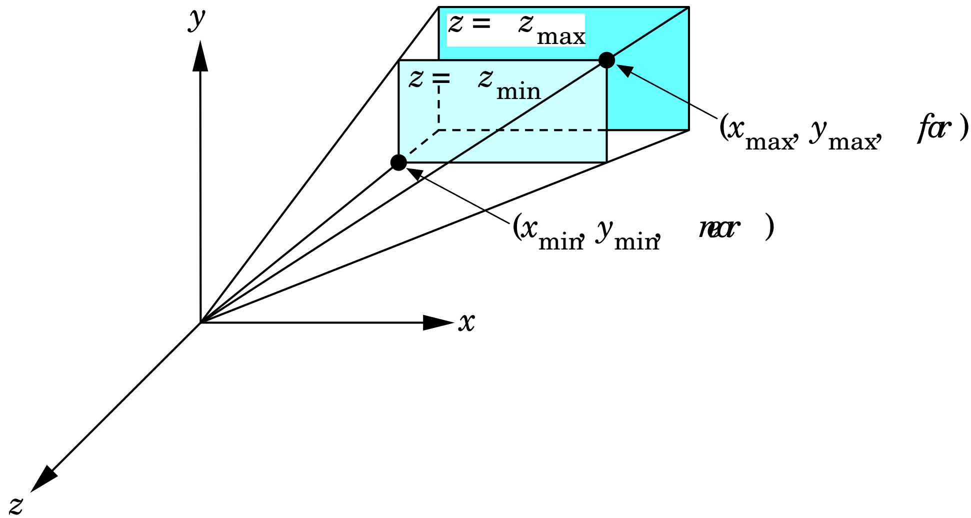
- truncated pyramid—a.k.a. *frustum*—in space
- specify near and far clipping planes
- specify field-of-view (fov) angle (or left and right edges)

How to clip in a frustum:

- transform the viewing volume to a cube
- clip (which is now easy)
- transform the viewing volume back to a frustum
- transformations are homogeneous

(OpenGL doesn't actually need to do this; rather, it clips first, while it's still working in a rectilinear world coordinate system, then projects to the frustum.)

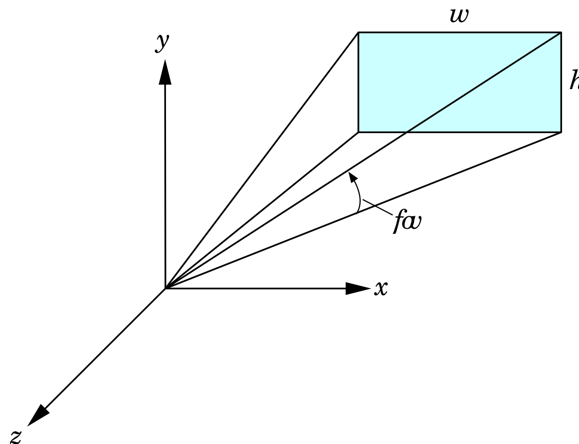
Specifying a perpective projection in OpenGL:



need not be symmetric!

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glFrustum (xmin, xmax, ymin, ymax, near, far);
```

can also use `gluPerspective(fovy, aspect, near, far)`



HSR review:

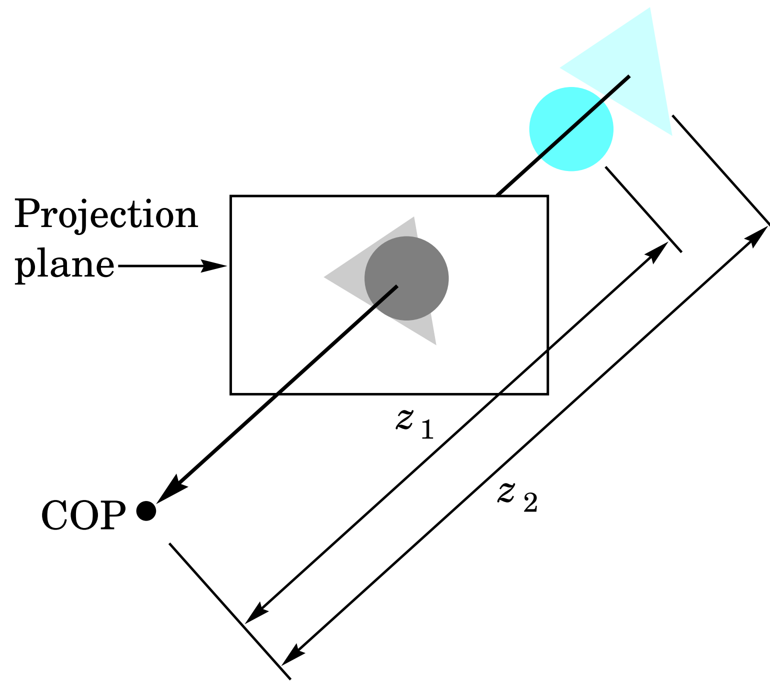
- work in the object space: e.g., render things back-to-front
- work in the image space: e.g., z-buffer

Former are lousy in pipelined architectures, which can mess with object order.

Latter much more common; explicitly determine who should go in front.

z buffering:

Want to pass a ray to each screen pixel and see which object was on top:



Implementation: during the rasterization step

- extra depth (or z) buffer, initialized to ∞ away

- as each pixel of an object is produced, check corresponding z buffer slot

- if new object pixel is behind existing top pixel, punt

- if new object pixel is on top, store it in the z buffer, along with its z (depth) value

OpenGL:

```
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB
| GLUT_DEPTH);
```

```
glEnable(GL_DEPTH_TEST);
```

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

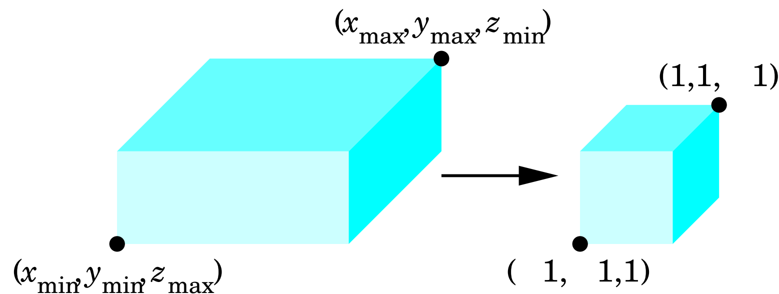
Culling: `glEnable(GL_CULL);`

Rolling your own projection:

Involves concocting and applying associated transformations and/or building the matrices (i.e., if you're going to be using it a lot and want it to be fast).

First, convert specified viewing volume to the default cube.

Simple example: `glOrtho(xmin, xmax, ymin, ymax);`



What are the transformations?

Rolling your own projection, cont.:

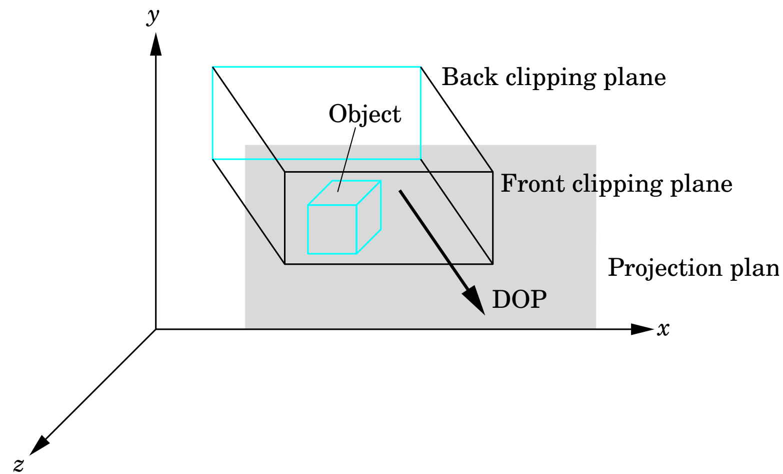
1. translate (xmin, ymin, zmin) to (-1, -1, -1)

2. scale all sides

Matrix:

$$P = \begin{bmatrix} \frac{2}{x_{max}-x_{min}} & 0 & 0 & -\frac{x_{max}+x_{min}}{x_{max}-x_{min}} \\ 0 & \frac{2}{y_{max}-y_{min}} & 0 & -\frac{y_{max}+y_{min}}{y_{max}-y_{min}} \\ 0 & 0 & \frac{2}{z_{max}-z_{min}} & -\frac{z_{max}+z_{min}}{z_{max}-z_{min}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

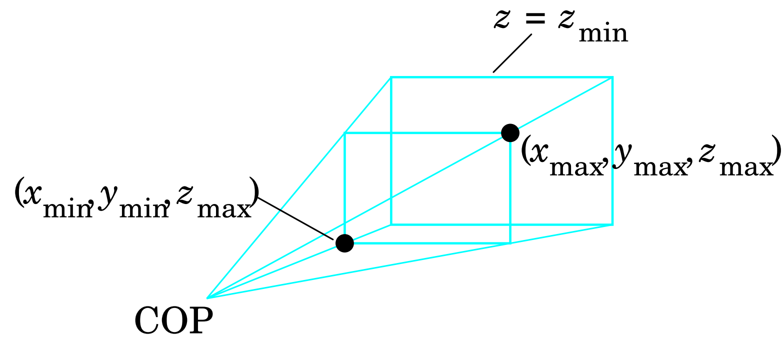
Oblique projections:



involve two shears, so matrix involves $\cot\theta$ and $\cot\phi$.

See pp254–255 for derivation.

OpenGL's frustum projection:



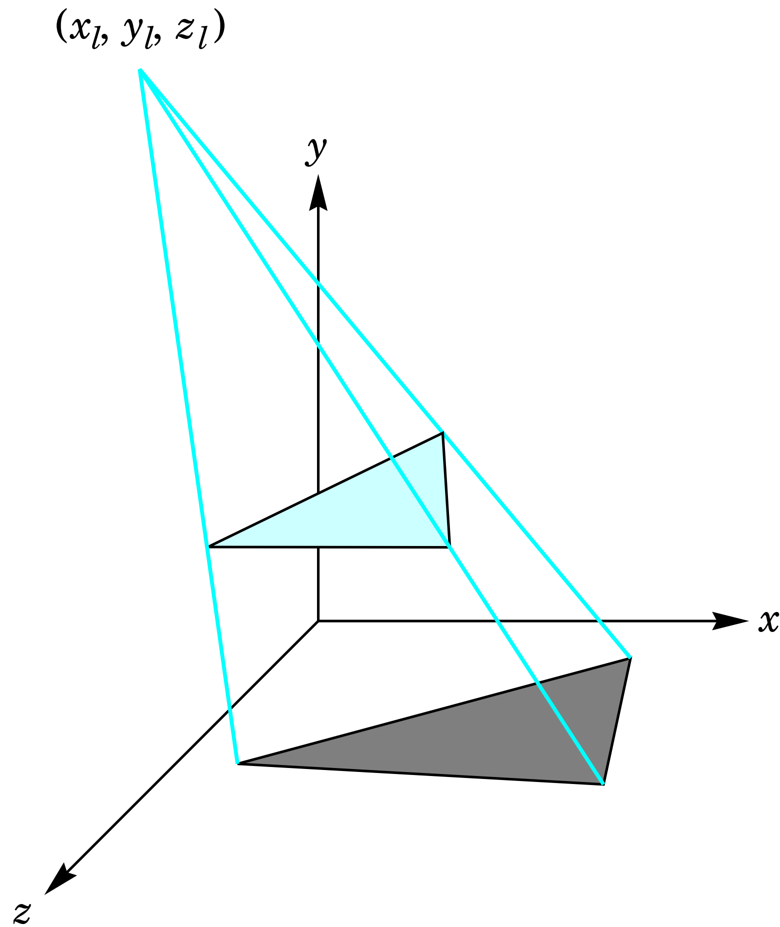
1. shear the frustum so it's symmetric (more cotangents)

2. scale its sides s.t. $x = \pm z$ and $y = \pm z$

3. move whole thing s.t. front is on $z = 1$ and back is on $z = -1$

See bottom of p260 for matrix.

An application: shadows



Homework problem: look at *shadow.c* and figure it out.