Capturing the topology of the cube:

Does cube.c do this?
Vertex arrays:

- types: vertex, color, color index, normal, texture coordinate, and edge flag

- first enable:
  ```
  glEnableClientState
  ```

- then tell OpenGL about them:
  ```
  glVertexPointer
  ```

- and then render:
  ```
  glDrawElements.
  ```

See Section 4.4.6 for syntax and `cubev.c` for an example.
Capturing the topology of the cube:

How does `cubev.c` store this information?
Viewing and projection:

- our eyes collapse 3D world to 2D retinal image; brain then reconstructs 3D

- in computer graphics, this process occurs by projection
  
  - viewing transformations: camera position and direction
  
  - perspective/orthographic transformations: reduce 3D to 2D

- use homogeneous transformations!
The gory details:

Given: geometry in the world coordinate system

Want: scene rendered to viewport

Steps:

- transform to camera coordinate system
- transform (warp) into view volume
- clip
- project to display coordinates
- rasterize
Cyrus-Beck clipping:

Algorithm:

- for each boundary line (plane), in point-normal form:
  - find $t_{hit}$ of ray
  - determine direction of ray w.r.t. $\vec{n}$
  - keep track of latest $t_{enter}$ and earliest $t_{exit}$
  - stop if that interval vanishes or flips
What if the polygon is concave?

Algorithm:

- for each boundary line (plane), in point-normal form:
  - find $t_{hit}$ of ray
  - true intersection if $t_{hit} \in [0, 1]$
  - build a “hitlist” of these times, sorted by time
  - very earliest $t_{hit}$ is the first entry
  - successive pairs are endpoints of segments to clip
Back to the gory details:

Given: geometry in the world coordinate system

Want: scene rendered to viewport

Steps:

- transform to camera coordinate system
- transform (warp) into view volume
- *clip*
- project to display coordinates
- *rasterize*
Orthographic projection:

- focal point (or center of projection) is at infinity, so rays are parallel and orthogonal to the image plane:

- good model for telephoto lens; no perspective effects

- if \( xy \) plane is the image plane, simply discard \( z \) coordinate to obtain orthographic view
Perspective projection:

The canonical case:

- camera looks along the $z$ axis
- focal point is the origin
- image plane is parallel to the $xy$ plane at distance $d$ (aka focal length)
Different kinds of perspective projection:

(a) three point

(b) two point

(c) one point
Moving the camera: transformations

\[
\begin{bmatrix}
    1 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 \\
    0 & 0 & 1 & 0 \\
\end{bmatrix}
\]

\texttt{glTranslate} + \texttt{glRotate} applied to the modelview matrix:

\[
\text{glMatrixMode} (\text{GL}_\text{MODELVIEW});
\text{glLoadIdentity}();
\text{glTranslatef}(0.0, 0.0, -d);
\text{glRotatef}(-90.0, 0.0, 1.0, 0.0);
Recall:

- **OpenGL**: “current transformation matrix” is the product of the **modelview** matrix and the **projection** matrix:

![Diagram showing the process of transformation with vertices, modelview, projection, and camera matrices](image)

- **modelview**: positions world relative to camera

- **projection**: projects onto viewport
Moving the camera: `gluLookAt`

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

...which is the topic of the in-class problem.
FIGURE 6.70 The six quadric surfaces: (a) Ellipsoid. (b) Hyperboloid of one sheet. (c) Hyperboloid of two sheets. (d) Elliptic cone. (e) Elliptic paraboloid. (f) Hyperbolic paraboloid.
**FIGURE 6.71** Characterization of the six “generic” quadric surfaces.

<table>
<thead>
<tr>
<th>Name of quadric</th>
<th>Implicit form</th>
<th>Parametric form</th>
<th>v-range, u-range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ellipsoid</td>
<td>(x^2 + y^2 + z^2 - 1)</td>
<td>((\cos(v) \cos(u), \cos(v) \sin(u), \sin(v)))</td>
<td>((-\pi/2, \pi/2), (-\pi, \pi))</td>
</tr>
<tr>
<td>Hyperboloid of one sheet</td>
<td>(x^2 + y^2 - z^2 - 1)</td>
<td>((\sec(v) \cos(u), \sec(v) \sin(u), \tan(v)))</td>
<td>((-\pi/2, \pi/2), (-\pi, \pi))</td>
</tr>
<tr>
<td>Hyperboloid of two sheets</td>
<td>(x^2 - y^2 - z^2 - 1)</td>
<td>((\sec(v) \cos(u), \sec(v) \tan(u), \tan(v)))</td>
<td>((-\pi/2, \pi/2)^a)</td>
</tr>
<tr>
<td>Elliptic cone</td>
<td>(x^2 + y^2 - z^2)</td>
<td>((v \cos(u), v \sin(u), v))</td>
<td>any real numbers, ((-\pi, \pi))</td>
</tr>
<tr>
<td>Elliptic paraboloid</td>
<td>(x^2 + y^2 - z)</td>
<td>((v \cos(u), v \sin(u), v^2))</td>
<td>(v \geq 0, (-\pi, \pi))</td>
</tr>
<tr>
<td>Hyperbolic paraboloid</td>
<td>(-x^2 + y^2 - z)</td>
<td>((v \tan(u), v \sec(u), v^2))</td>
<td>(v \geq 0, (-\pi, \pi))</td>
</tr>
</tbody>
</table>

^aThe \(v\)-range for sheet \# 1 is \((-\pi/2, \pi/2)\) and for sheet \# 2 is \((\pi/2, 3\pi/2)\)
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: $\vec{v}'$, $\vec{n}'$, and $\vec{u}'$.
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.
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:

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

See section 5.3.2 for examples.
More on $w$:

Last week, 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?
Why allow \( w \) to be non-zero:

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:

\[
Q = \begin{bmatrix}
x \\
y \\
z \\
z/d
\end{bmatrix}
\]
Why allow $w$ to be non-zero, cont:

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

$$x' = \frac{x}{z/d}$$
$$y' = \frac{y}{z/d}$$
$$z' = \frac{z}{z/d} = d$$

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:
• specify near and far clipping planes
  – instead of mapping $z$ to $d$, transform $z$ between $z_{near}$ and $z_{far}$, onto a fixed range
  – used for $z$-buffer HSR

• specify field-of-view (fov) angle
The view volume in a perpective projection:

- truncated pyramid—a.k.a. frustum—in space, defined by focal point and window in the image plane (assuming window mapped to viewport)

- defines visible region of space

- pyramid edges are clipping planes

*why near plane?* prevent points behind camera from being seen

*why far plane?* allows $z$ to be scaled to a limited fixed-point value ($z$ buffering)