3D Viewing Episode 2

1 Positioning and Orienting the Camera

- Recall that our projection calculations, whether orthographic or frustum/perspective, were made with the camera at $(0, 0, 0)$ looking down the $-z$ axis.
- The camera is also vertically oriented — i.e. "up" is the vector $\langle 0, 1, 0\rangle$.
- But we want the camera to be anywhere, looking anywhere; how is this done?
- Recall that transforming an object is equivalent to transforming its axes in the opposite manner: e.g. moving an object centered on the origin to the right by 10 units is visually identical to moving the origin (and its axes) to the *left* by 10 units.
- Suppose that we would like to position our camera at some point P — the *center of projection* (COP). The camera is looking toward Q — the *look-at* or *eye* point — and is oriented according to the *up* vector, which indicates what is vertical relative to the camera. Figure 1 illustrates this setup.

2 Deriving the Transformation

- The idea here is to derive the transformation such that $\langle 1, 0, 0\rangle \rightarrow x_e$, $\langle 0, 1, 0\rangle \rightarrow y_e$, and $\langle 0, 0, 1\rangle \rightarrow z_e$.
- x_e, y_e, and z_e are vectors, so they can be broken down in terms of their components as $\langle x_{e_x}, x_{e_y}, x_{e_z}\rangle$, $\langle y_{e_x}, y_{e_y}, y_{e_z}\rangle$, and $\langle z_{e_x}, z_{e_y}, z_{e_z}\rangle$.
- Thus, the matrix that transforms the standard cartesian axes into axes relative to the camera is:

$$\begin{bmatrix} x_{e_x} & y_{e_x} & z_{e_x} & 0 \\ x_{e_y} & y_{e_y} & z_{e_y} & 0 \\ x_{e_z} & y_{e_z} & z_{e_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

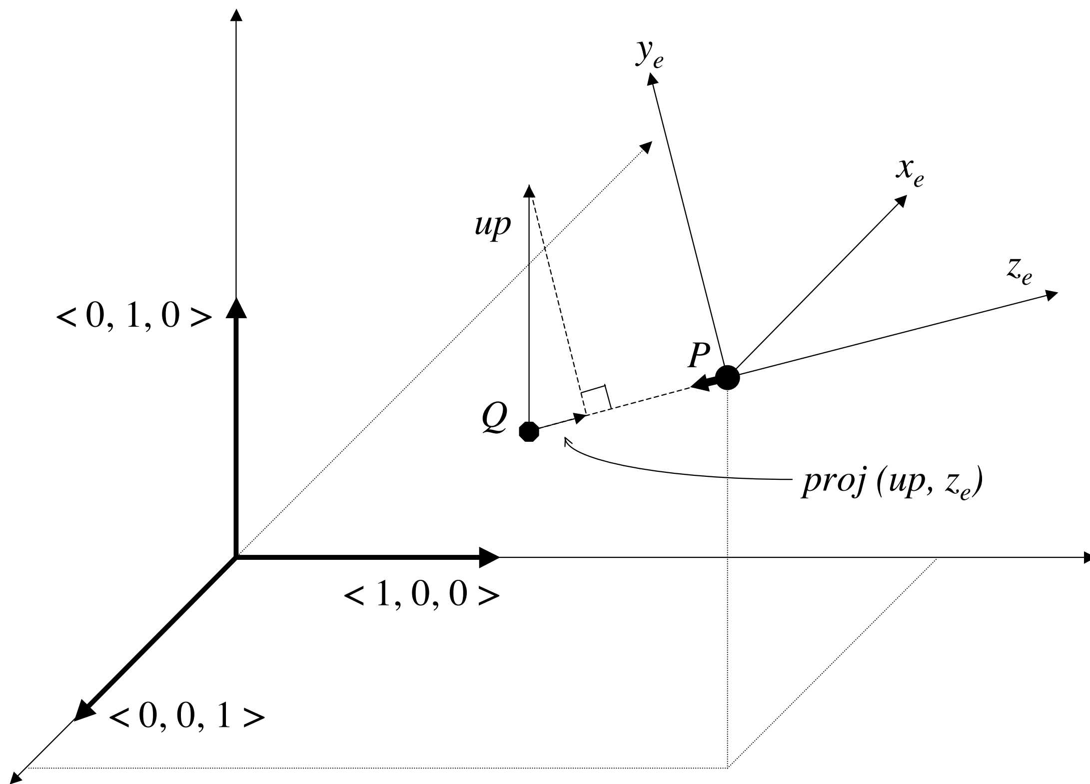

Figure 1: Setting up the camera in terms of a center of projection P looking toward the point Q with some *up* vector.

- So what are x_e, y_e, and z_e anyway? Easier than it seems — they can actually be derived from our look-at arguments:

$$z_e = U_{P-Q} \tag{2}$$

$$y_e = U_{up-proj(up,z_e)} \tag{3}$$

$$x_e = y_e \times z_e \tag{4}$$

  Note how, since z_e and y_e have been normalized into unit vectors, then x_e is automatically a unit vector.

- The matrix in (1) only rotates the coordinate system — it is still centered on the origin. Thus, we need to translate the axes to P (note how matrix composition reads from right to left):

$$\begin{bmatrix} 1 & 0 & 0 & P_x \\ 0 & 1 & 0 & P_y \\ 0 & 0 & 1 & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{e_x} & y_{e_x} & z_{e_x} & 0 \\ x_{e_y} & y_{e_y} & z_{e_y} & 0 \\ x_{e_z} & y_{e_z} & z_{e_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

- Remember that (5) is the transformation that converts the *axes* to match the desired axes for the camera. However, we want the transform that converts the *objects* in 3D space according to the camera. Thus, what we really want is the *inverse* of (5) — e.g. an object that is at the origin would have coordinates $(-P_x, -P_y, -P_z)$ from the perspective of a camera that is located at world coordinates (P_x, P_y, P_z).

- Instead of multiplying out the matrices and then inverting after, we use the matrix multiplication identity $(AB)^{-1} = B^{-1}A^{-1}$ to get:

$$\begin{bmatrix} x_{e_x} & x_{e_y} & x_{e_z} & 0 \\ y_{e_x} & y_{e_y} & y_{e_z} & 0 \\ z_{e_x} & z_{e_y} & z_{e_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

- All that's left is to multiply out the matrices — in addition, this matrix is multiplied to all other active transforms at that time:

$$\begin{bmatrix} x_{e_x} & x_{e_y} & x_{e_z} & -(P \cdot x_e) \\ y_{e_x} & y_{e_y} & y_{e_z} & -(P \cdot y_e) \\ z_{e_x} & z_{e_y} & z_{e_z} & -(P \cdot z_e) \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7}$$

  This is what `gluLookAt()` multiplies to the current matrix, of course substituting the x_e, y_e, and z_e with their derivations in (4), (3), and (2), respectively. And yes, it is affine!

3 Putting Them Together

- Did you notice that this matrix is applied to the world, and not during projection? This is why the OpenGL code sequence for setting up the camera is:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(px, py, pz, qx, qy, qz, 0.0, 1.0, 0.0);
/* Rest of scene goes here. */
```

- If you put everything together, every vertex in OpenGL thus goes through these matrices on its journey from the 3D world to the screen or window:

$$[\text{ viewport matrix }]\,[\text{ projection matrix }]\,[\text{ modelview matrix }] \tag{8}$$

- Note how we keep these three matrices distinct instead of multiplying them together. One reason is to maintain the ability to manipulate them separately — for example, if the window size changes, only the viewport and projection matrices will change, but not the modelview matrix (or at least it shouldn't).
- Another reason is further rendering algorithms such as hidden surface removal and clipping — these steps are best done at the gaps between these matrices.
- Up next — the details on these algorithms, and where they appear on the matrix train.