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., \(\vec{u}p\) 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 \(\vec{u}p\) 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 \(\langle 1, 0, 0 \rangle \rightarrow \vec{x}_c\), \(\langle 0, 1, 0 \rangle \rightarrow \vec{y}_c\), and \(\langle 0, 0, 1 \rangle \rightarrow \vec{z}_c\).

• \(\vec{x}_c\), \(\vec{y}_c\), and \(\vec{z}_c\) 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:

\[
\begin{bmatrix}
  x_{e_x} & y_{e_x} & z_{e_x} & 0 \\
  x_{e_y} & y_{e_y} & z_{e_y} & 0 \\
  x_{e_z} & y_{e_z} & z_{e_z} & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\] (1)
Figure 1: Setting up the camera in terms of a center of projection $P$ looking toward the point $Q$ with some $\mathbf{up}$ vector.
• So what are $\vec{x}_e$, $\vec{y}_e$, and $\vec{z}_e$ anyway? Easier than it seems — they can actually be derived from our look-at arguments:

\[
\vec{z}_e = U_{P - Q} \tag{2}
\]

\[
\vec{y}_e = U_{\vec{u} \cdot proj(\vec{u}, \vec{z}_e)} \tag{3}
\]

\[
\vec{x}_e = \vec{y}_e \times \vec{z}_e \tag{4}
\]

Note how, since $\vec{z}_e$ and $\vec{y}_e$ have been normalized into unit vectors, then $\vec{x}_e$ 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 $(AB)^{-1} = B^{-1}A^{-1}$. So what are the inverses of these matrices?

• The inverse of the translation matrix is straightforward: we translate by $(-P_x, -P_y, -P_z)$ instead of $(P_x, P_y, P_z)$:

\[
\begin{bmatrix}
1 & 0 & 0 & -P_x \\
0 & 1 & 0 & -P_y \\
0 & 0 & 1 & -P_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\tag{6}
\]

• The “axis orientation” matrix is also easy to state, but a little trickier to explain; it is the matrix’s transpose $(A^T)$:

\[
\begin{bmatrix}
x_{e_x} & x_{e_y} & x_{e_z} & 0 \\
y_{e_x} & y_{e_y} & y_{e_z} & 0 \\
z_{e_x} & z_{e_y} & z_{e_z} & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\tag{7}
\]

• Now, to see why this is the case, think about the characteristics of $\vec{x}_e$, $\vec{y}_e$, and $\vec{z}_e$. If you look at the product of the axis orientation matrix and its transpose, we get:
\[
\begin{bmatrix}
 x_{ex} & x_{ey} & x_{ez} & 0 \\
 y_{ex} & y_{ey} & y_{ez} & 0 \\
 z_{ex} & z_{ey} & z_{ez} & 0 \\
 0 & 0 & 0 & 1 
\end{bmatrix}
\begin{bmatrix}
 x_{ex} & y_{ex} & z_{ex} & 0 \\
 x_{ey} & y_{ey} & z_{ey} & 0 \\
 x_{ez} & y_{ez} & z_{ez} & 0 \\
 0 & 0 & 0 & 1 
\end{bmatrix}
= \begin{bmatrix}
 \vec{x}_e \cdot \vec{x}_e & \vec{x}_e \cdot \vec{y}_e & \vec{x}_e \cdot \vec{z}_e & 0 \\
 \vec{y}_e \cdot \vec{x}_e & \vec{y}_e \cdot \vec{y}_e & \vec{y}_e \cdot \vec{z}_e & 0 \\
 \vec{z}_e \cdot \vec{x}_e & \vec{z}_e \cdot \vec{y}_e & \vec{z}_e \cdot \vec{z}_e & 0 \\
 0 & 0 & 0 & 1 
\end{bmatrix}
\] (8)

\(\vec{x}_e, \vec{y}_e,\) and \(\vec{z}_e\) are all unit vectors — see (4), (3), and (2) — and so \(\vec{x}_e \cdot \vec{x}_e, \vec{y}_e \cdot \vec{y}_e,\) and \(\vec{z}_e \cdot \vec{z}_e\) are all equal to one. Further, \(\vec{x}_e, \vec{y}_e,\) and \(\vec{z}_e\) are mutually perpendicular to each other — and thus the dot product of any two of these vectors is equal to zero. If you substitute those values into the product, note that you get the identity matrix — and thus, \(A^{-1} = A^T\) for the “axis orientation” matrix!

- To wrap it all up, we now have our final lookAt matrix:

\[
\begin{bmatrix}
 x_{ex} & x_{ey} & x_{ez} & 0 \\
 y_{ex} & y_{ey} & y_{ez} & 0 \\
 z_{ex} & z_{ey} & z_{ez} & 0 \\
 0 & 0 & 0 & 1 
\end{bmatrix}
\begin{bmatrix}
 1 & 0 & 0 & -P_x \\
 0 & 1 & 0 & -P_y \\
 0 & 0 & 1 & -P_z \\
 0 & 0 & 0 & 1 
\end{bmatrix}
\] (9)

- All that’s left is to multiply the matrices, yielding:

\[
\begin{bmatrix}
 x_{ex} & x_{ey} & x_{ez} & -\langle P \cdot \vec{x}_e \rangle \\
 y_{ex} & y_{ey} & y_{ez} & -\langle P \cdot \vec{y}_e \rangle \\
 z_{ex} & z_{ey} & z_{ez} & -\langle P \cdot \vec{z}_e \rangle \\
 0 & 0 & 0 & 1 
\end{bmatrix}
\] (10)

This is what gluLookAt() multiplies to the current matrix, of course substituting the \(\vec{x}_e, \vec{y}_e,\) and \(\vec{z}_e\) with their derivations in (4), (3), and (2), respectively. And yes, this transform 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:

\[
\begin{bmatrix}
 \text{viewport matrix} \]
\begin{bmatrix}
 \text{projection matrix} \]
\begin{bmatrix}
 \text{modelview matrix} \]
\] (11)
• 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.