Geometry: Cameras

Outline

• Setting up the camera
• Projections
  - Orthographic
  - Perspective
Controlling the camera

- Default OpenGL camera: At \((0, 0, 0)^T\) in world coordinates looking in \(Z\) direction with **up vector** \((0, 1, 0)^T\)
  - Up vector controls camera roll (rotation around z-axis)
- Changing position: `gluLookAt()`
  - \(\text{eye} = (\text{eye}_X, \text{eye}_Y, \text{eye}_Z)^T\): Desired camera position
  - \(\text{center} = (\text{center}_X, \text{center}_Y, \text{center}_Z)^T\): Where camera is looking
  - \(\text{up} = (\text{up}_X, \text{up}_Y, \text{up}_Z)^T\):
    Camera’s “up” vector

The Viewing Volume

- Definition: The region of 3-D space visible in the image
- Depends on:
  - Camera position, orientation
  - Field of view, image size
  - Projection type
    - Orthographic
    - Perspective
**gluLookAt()**: Details

- **To build the camera coordinate system, and find the rigid transformation \( \mathbf{CM} \) between world system and camera system.**

- **Steps**
  1. Compute vectors \( \mathbf{u}, \mathbf{v}, \mathbf{n} \) defining new camera axes in world coordinates
  2. Compute location \( \mathbf{c}_M \) of old camera position in terms of new location's coordinate system
  3. Fill in rigid transform matrix \( \mathbf{CM} \)

![Diagram](image1.png)

**gluLookAt()**: Axes

- **Form basis vectors**
  - New camera Z axis: \( \mathbf{n} = \mathbf{eye} - \mathbf{center} \)
  - New camera X axis: \( \mathbf{u} = \mathbf{up} \times \mathbf{n} \)
  - New camera Y axis: \( \mathbf{v} = \mathbf{n} \times \mathbf{u} \) (not necessarily same as \( \mathbf{up} \))

- **Normalize so that these are unit vectors**

![Diagram](image2.png)
**gluLookAt()**: Axes

- Form basis vectors
  - New camera Z axis: \( \mathbf{n} = \mathbf{eye} - \mathbf{center} \)
  - New camera X axis: \( \mathbf{u} = \mathbf{up} \times \mathbf{n} \)
  - New camera Y axis: \( \mathbf{v} = \mathbf{n} \times \mathbf{u} \) (not necessarily same as \( \mathbf{up} \))
- Normalize so that these are unit vectors
gluLookAt(): Axes

• Form basis vectors
  - New camera Z axis: \( n = \text{eye} - \text{center} \)
  - New camera X axis: \( u = \text{up} \times n \)
  - New camera Y axis: \( v = n \times u \) (not necessarily same as \( \text{up} \))
• Normalize so that these are unit vectors
**gluLookAt()**: Axes

- Form basis vectors
  - New camera Z axis: \( \mathbf{n} = \text{eye - center} \)
  - New camera X axis: \( \mathbf{u} = \mathbf{up} \times \mathbf{n} \)
  - New camera Y axis: \( \mathbf{v} = \mathbf{n} \times \mathbf{u} \) (not necessarily same as \( \mathbf{up} \))

- Normalize so that these are unit vectors

---

**gluLookAt()**: Axes

- Now make 3 x 3 rotation matrix from formula on rigid transform slide:

\[
\mathcal{C}_W \mathbf{R} = (\mathcal{C}_W \mathbf{i}, \mathcal{C}_W \mathbf{j}, \mathcal{C}_W \mathbf{k})
\]

- Since \( \mathcal{C}_W \mathbf{R} = \mathcal{C}_W \mathbf{R}^T \), this is the same as:

\[
\mathcal{C}_W \mathbf{R} = (\mathcal{W}_i \mathcal{C}, \mathcal{W}_j \mathcal{C}, \mathcal{W}_k \mathcal{C})^T = \begin{pmatrix} \mathcal{W}_i^T \\ \mathcal{W}_j^T \\ \mathcal{W}_k^T \end{pmatrix} = \begin{pmatrix} \mathbf{u}^T \\ \mathbf{v}^T \\ \mathbf{n}^T \end{pmatrix}
\]
**gluLookAt()**: Location

- $c_{OW}$: World origin in camera coordinates

- $c_{OW}$: World origin in camera coordinates
- **eye** is in world coordinates, so project on camera axes:

$$c_{OW} = (-\text{eye} \cdot u, -\text{eye} \cdot v, -\text{eye} \cdot n)^T$$
gluLookAt(): Matrix

- Letting $t = c_o w$ and writing the vector components as $u = (u_x, u_y, u_z)^T$, etc., the final transformation matrix is given by:

$$\begin{pmatrix}
 u_x & u_y & u_z & t_x \\
 v_x & v_y & v_z & t_y \\
 n_x & n_y & n_z & t_z \\
 0 & 0 & 0 & 1
\end{pmatrix}$$

Transformations vs. Projections

- **Transformation**: Mapping within $n$-D space
  - Moves points around, effectively warping space
- **Projection**: Mapping from $n$-D space down to lower-dimensional subspace
  - E.g., point in 3-D space to point on plane (a 2-D entity) in that space
  - We will be interested in such 3-D to 2-D projections where the plane is the image

[Diagram of parallel projection along direction $d$ onto a plane]
Parallel Projections

Oblique: \( \mathbf{d} \) in general position relative to plane normal \( \mathbf{n} \)

Orthographic: \( \mathbf{d} \) parallel to \( \mathbf{n} \)

Orthographic Projection

- Projection direction \( \mathbf{d} \) is aligned with \( Z \) axis
- Viewing volume is “brick”-shaped region in space
  - Not the same as image size
- No perspective effects—distant objects look same as near ones, so camera \((x, y, z)\) => image \((x, y)\)
Orthographic Projection in OpenGL

- Setting up the **viewing volume** (VV):
  - `glOrtho()`
    - `left, right, bottom, top`: Coordinates of sides of viewing volume
    - `znear, zfar`: Distance to arbitrarily designated front, back sides of VV
      - Negative = Behind camera
    - `gluOrtho2D(): glOrtho()` with `near = -1, far = 1`
- Modifies top 4 x 4 matrix of **GL_PERSPECTIVE** matrix stack
  - Applied after **GL_MODELVIEW** transformation has put things in camera coordinates
  - Actual matrix scales viewing volume (VV) to **canonical VV** (CVV): Cube extending from -1 to 1 along each dimension

---

Perspective with a Pinhole Camera

Instead of single direction $d$ characteristic of parallel projections, rays emanating from single point $C$ define perspective projection
Perspective Projection

- Characteristic shape is a frustum—a truncated pyramid
Perspective Projection: Properties

- Far objects appear smaller than near ones
- Lines are preserved
- Parallel lines in plane $\Pi$ converge at infinity

Pinhole Camera Terminology
Perspective Projection

- Letting the camera coordinates of the projected point be \( \mathbf{x}_{\text{cam}} = (x, y, z)^T \) leads by similar triangles to:

\[
\mathbf{x}_{\text{im}} = \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} fx/z \\ fy/z \end{pmatrix}
\]

Perspective Projection Matrix

- Using homogeneous coordinates, we can describe a perspective transformation with a 4 x 4 matrix multiplication:

\[
\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z/f \end{pmatrix} \rightarrow \begin{pmatrix} fx/z \\ fy/z \\ f \end{pmatrix}
\]

(by the rule for converting between homogeneous and regular coordinates—this is called the perspective division)

- Projection to \((u, v)^T\) is again accomplished by simply dropping the \(z\) coordinate
Perspective Projections in OpenGL

- **glFrustum()** sets transformation to CVV
  - Arguments like `glOrtho()`, but **znear**, **zfar** must be positive

- A final transform, **GL VIEWPORT** (see `glViewport()`) shifts NDC to image coordinate origin and scales to fit window

---

gluPerspective()

- Simplifies call to **glFrustum()**
- Arguments:
  - **fovy**: Field of view angle (degrees) in Y direction
  - **aspect**: Ratio of width to height of viewing frustum
  - **near**, **far**: Same as **glFrustum()**