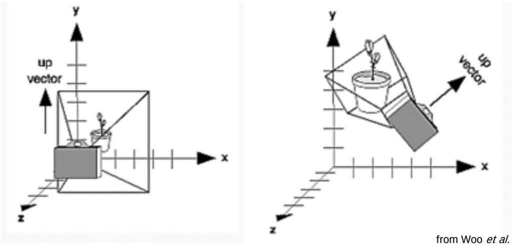
# Geometry: Cameras

## Outline

- Setting up the camera
- Projections
  - Orthographic
  - Perspective

## Controlling the camera

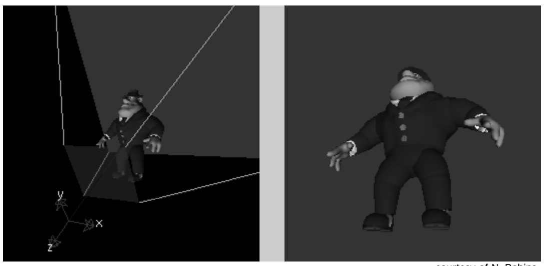
- Default OpenGL camera: At $(0, 0, 0)^T$ in world coordinates looking in  Z direction with **up vector** $(0, 1, 0)^T$
  - Up vector controls camera roll (rotation around z-axis)
- Changing position: **`gluLookAt()`**
  - **eye**  = (**`eyeX`**, **`eyeY`**, **`eyeZ`**)$^T$: Desired camera position
  - **center** = (**`centerX`**, **`centerY`**, **`centerZ`**)$^T$: Where camera is looking
  - **up** = (**upX**, **upY**, **upZ**)$^T$: Camera's "up" vector




from Woo *et al.*

## The Viewing Volume

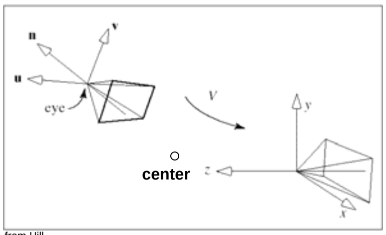
- Definition: The region of 3-D space visible in the image
- Depends on:
  - Camera position, orientation
  - Field of view, image size
  - Projection type
    - Orthographic
    - Perspective



courtesy of N. Robins

# `gluLookAt()`: Details

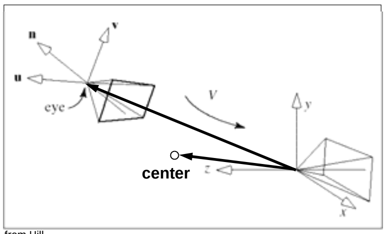
- To build the camera coordinate system, and find the rigid transformation ${}^{C}_{W}\mathbf{M}$ between world system and camera system.
- Steps
  1. Compute vectors **u**, **v**, **n** defining new camera axes in world coordinates
  2. Compute location ${}^{C}\mathbf{O}_{W}$ of old camera position in terms of new location's coordinate system
  3. Fill in rigid transform matrix ${}^{C}_{W}\mathbf{M}$




from Hill

# `gluLookAt()`: Axes

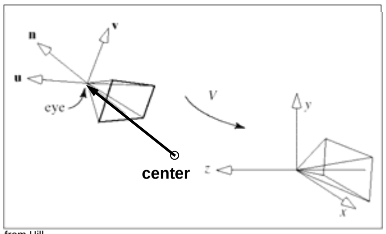
- Form basis vectors
  - New camera Z axis: **n** = **eye** - **center**
  - New camera X axis: **u** = **up** x **n**
  - New camera Y axis: **v** = **n x u** (not necessarily same as **up**)
- Normalize so that these are unit vectors




from Hill

## gluLookAt(): Axes

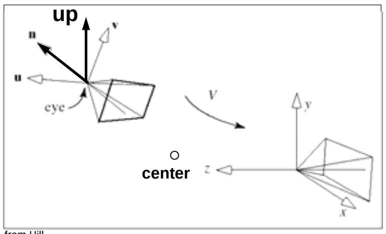
- Form basis vectors
  - New camera Z axis: **n** = **eye** - **center**
  - New camera X axis: **u** = **up** x **n**
  - New camera Y axis: **v** = **n x u** (not necessarily same as **up**)
- Normalize so that these are unit vectors




from Hill


## gluLookAt(): Axes

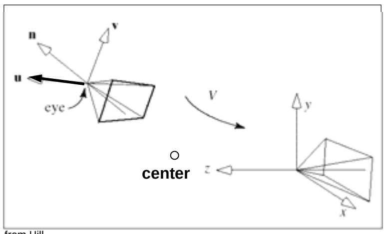
- Form basis vectors
  - New camera Z axis: **n** = **eye** - **center**
  - New camera X axis: **u** = **up** x **n**
  - New camera Y axis: **v** = **n x u** (not necessarily same as **up**)
- Normalize so that these are unit vectors


from Hill

## `gluLookAt()`: Axes

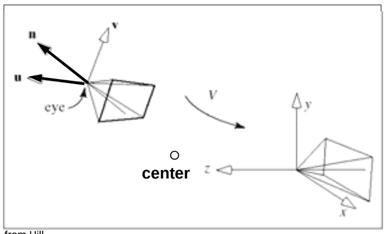
- Form basis vectors
  - New camera Z axis: **n** = **eye** - **center**
  - New camera X axis: **u** = **up** x **n**
  - New camera Y axis: **v** = **n x u** (not necessarily same as **up**)
- Normalize so that these are unit vectors




from Hill

## `gluLookAt()`: Axes

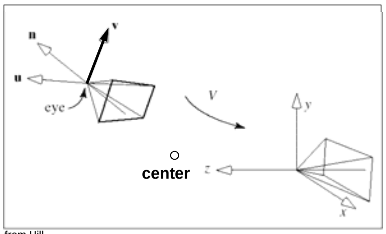
- Form basis vectors
  - New camera Z axis: **n** = **eye** - **center**
  - New camera X axis: **u** = **up** x **n**
  - New camera Y axis: **v** = **n x u** (not necessarily same as **up**)
- Normalize so that these are unit vectors




from Hill

## `gluLookAt()`: Axes

- Form basis vectors
  - New camera Z axis: **n** = **eye** - **center**
  - New camera X axis: **u** = **up** x **n**
  - New camera Y axis: **v** = **n x u** (not necessarily same as **up**)
- Normalize so that these are unit vectors




from Hill

## `gluLookAt()`: Axes

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

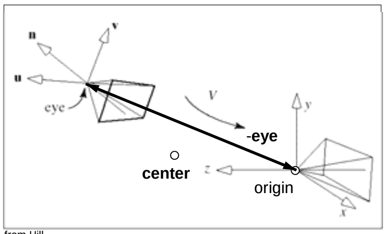
$$ {}^{\mathcal{C}}_{\mathcal{W}}\mathbf{R} = ({}^{\mathcal{C}}\mathbf{i}_{\mathcal{W}}, {}^{\mathcal{C}}\mathbf{j}_{\mathcal{W}}, {}^{\mathcal{C}}\mathbf{k}_{\mathcal{W}}) $$

- Since ${}^{\mathcal{C}}_{\mathcal{W}}\mathbf{R} = {}^{\mathcal{W}}_{\mathcal{C}}\mathbf{R}^T$, this is the same as:

$$ {}^{\mathcal{C}}_{\mathcal{W}}\mathbf{R} = ({}^{\mathcal{W}}\mathbf{i}_{\mathcal{C}}, {}^{\mathcal{W}}\mathbf{j}_{\mathcal{C}}, {}^{\mathcal{W}}\mathbf{k}_{\mathcal{C}})^T = \begin{pmatrix} {}^{\mathcal{W}}\mathbf{i}_{\mathcal{C}}^T \\ {}^{\mathcal{W}}\mathbf{j}_{\mathcal{C}}^T \\ {}^{\mathcal{W}}\mathbf{k}_{\mathcal{C}}^T \end{pmatrix} = \begin{pmatrix} \mathbf{u}^T \\ \mathbf{v}^T \\ \mathbf{n}^T \end{pmatrix} $$

## `gluLookAt()`: Location

- $c_{\mathbf{o}_\mathcal{W}}$ : World origin in camera coordinates

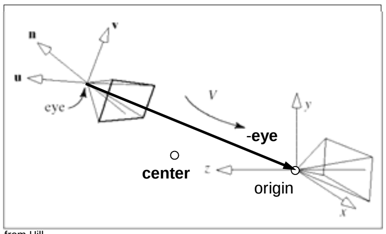



from Hill

## `gluLookAt()`: Location

- $c_{\mathbf{o}_\mathcal{W}}$ : World origin in camera coordinates
- **-eye** is in world coordinates, so project on camera axes:

$$c_{\mathbf{o}_\mathcal{W}} = (-\mathbf{eye} \cdot \mathbf{u}, -\mathbf{eye} \cdot \mathbf{v}, -\mathbf{eye} \cdot \mathbf{n})^T$$




from Hill

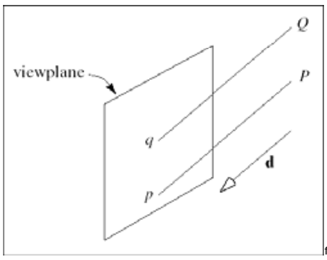
## `gluLookAt()`: Matrix

- Letting $\mathbf{t} = {}^{\mathcal{C}}\mathbf{o}_{\mathcal{W}}$ and writing the vector components as **u** = $(u_x, u_y, u_z)^T$, etc., the final transformation matrix is given by:

$$ {}^{\mathcal{C}}_{\mathcal{W}}\mathbf{M} = \begin{pmatrix} u_x & u_y & u_z & t_x \\ v_x & v_y & v_z & t_y \\ n_x & n_y & n_z & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

## Transformations vs. Projections

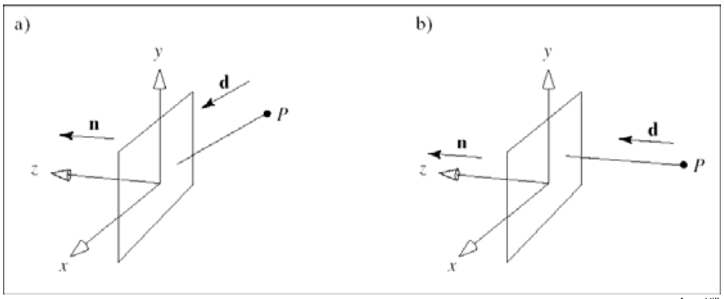
- **Transformation**: Mapping within n-D space
  - Moves points around, effectively warping space
- **Projection**: Mapping from n-D space down to lower-dimensional subspace
  - E.g., point in 3-D space to point on plane (a 2-D entity) in that space
  - We will be interested in such 3-D to 2-D projections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




from Hill

**Parallel projection** along direction **d** onto a plane

## Parallel Projections



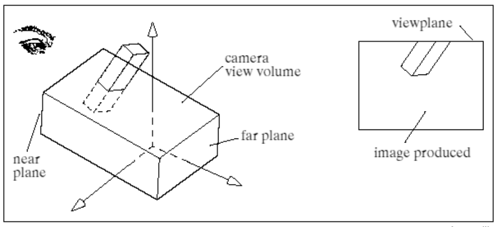

from Hill

**Oblique**: **d** in general position relative to plane normal **n**

**Orthographic**: **d** parallel to **n**

## Orthographic Projection

- Projection direction **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)

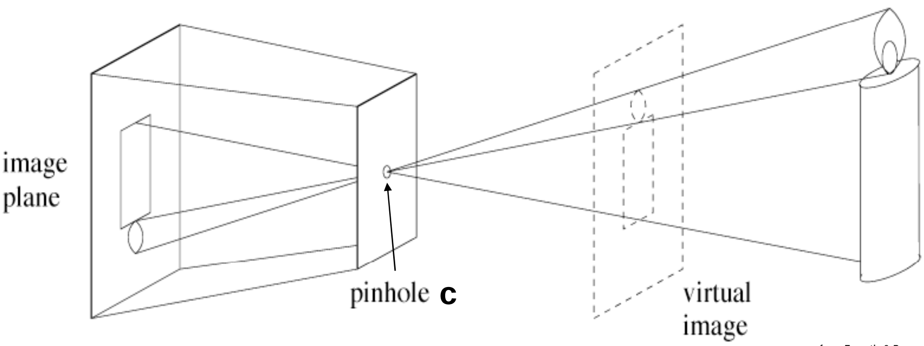



from Hill

## 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

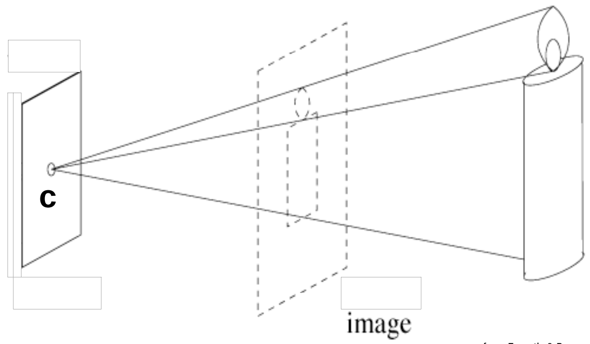
## Perspective with a Pinhole Camera




from Forsyth & Ponce

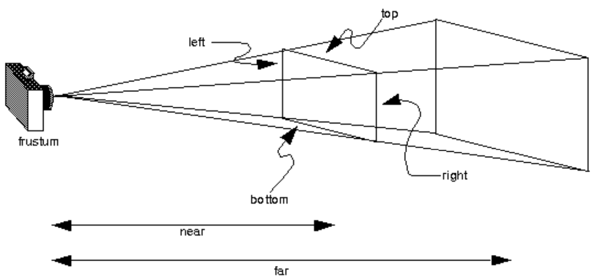
Instead of single direction **d** characteristic of parallel projections, rays emanating from single point **c** define perspective projection

## Perspective Projection



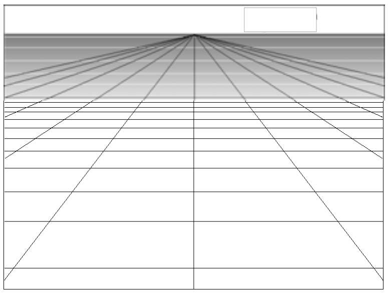

## Perspective Projection: Viewing Volume

- Characteristic shape is a **frustum**—a truncated pyramid



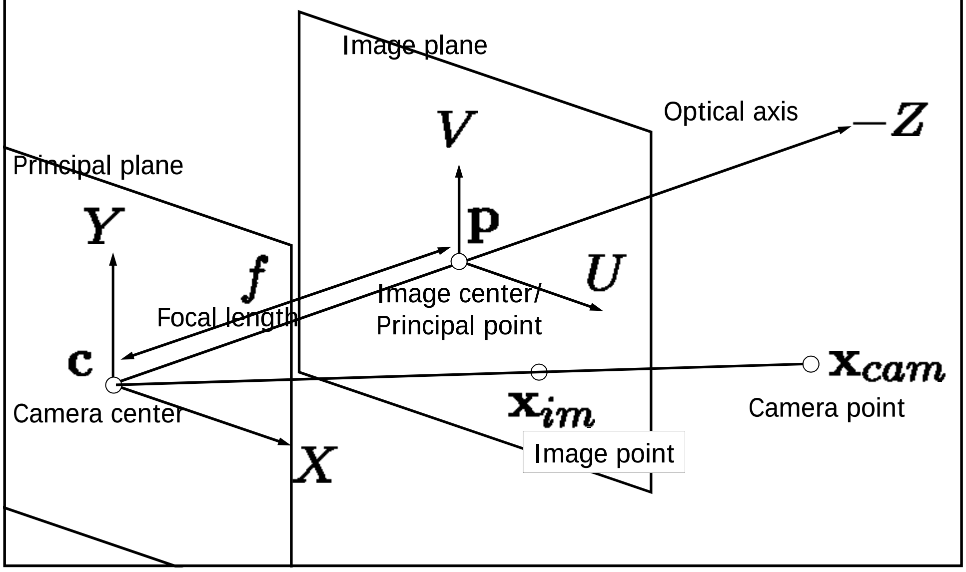
## Perspective Projection: Properties

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



from Hill

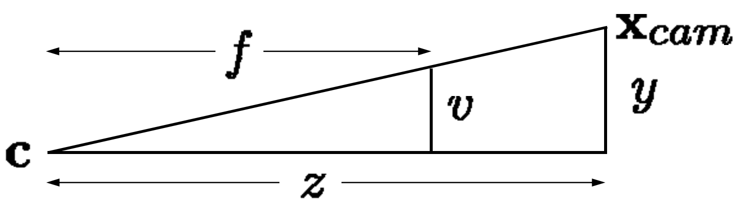
## Pinhole Camera Terminology



## Perspective Projection

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

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




## Perspective Projection Matrix

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

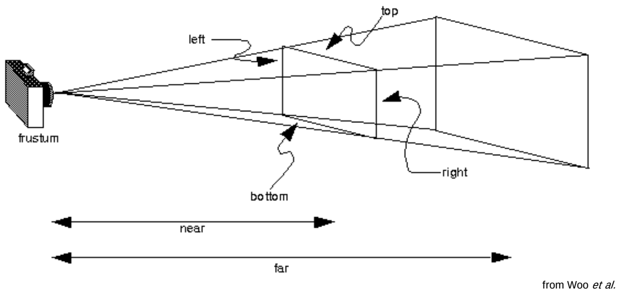
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z/f \end{pmatrix} \rightarrow \begin{pmatrix} fx/z \\ fy/z \\ f \end{pmatrix}$$

(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()`**