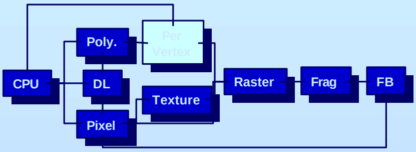
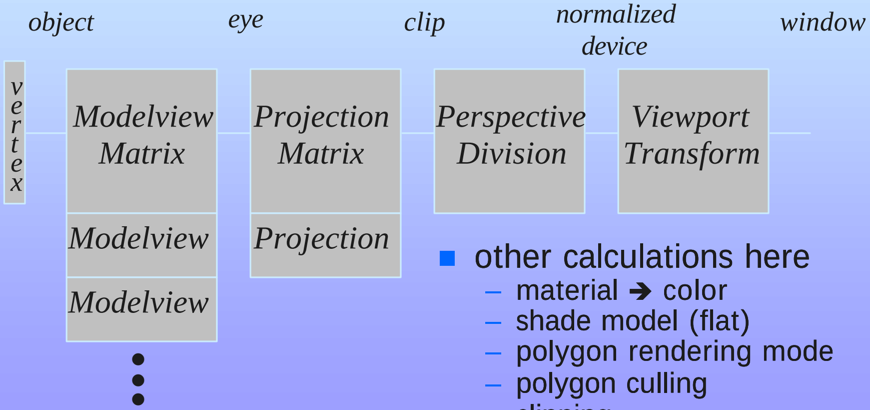
Realtime 3D Computer Graphics & Virtual Reality

Viewing

Transformation Pipeline





object

eye

clip

normalized device

window

vertex

Modelview Matrix

Projection Matrix

Perspective Division

Viewport Transform

Modelview

Modelview

Projection

- other calculations here
  - material ➔ color
  - shade model (flat)
  - polygon rendering mode
  - polygon culling
  - clipping

Example Frame Rending

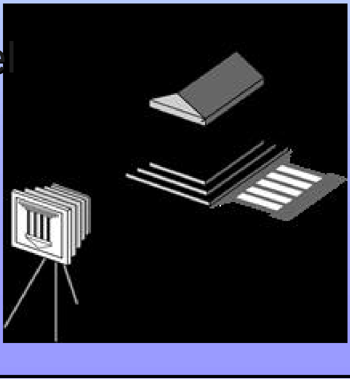
- Establish World
- **Establish Viewpoint & Display Plane**
- **Perform 3D Clipping**
- **Projection of World onto Display Plane**
- Perform Hidden Surface Removal
- Determine Display Colors
- Rasterization of 2D Display

*Rendering order may be changed based on algorithms involved!

Classical and General Viewing

Viewing

- Process for “Seeing” a world
- Projection of a 3D world onto a 2D plane
- Synthetic Camera Model



Viewing Issues

- Location of viewer
- Location of view plane
- What can be seen (Clipping)
- How relationships are maintained
  - Parallel Lines
  - Angles
  - Distances (Foreshortening)
  - Relation to Viewer

Objects vs. Scenes

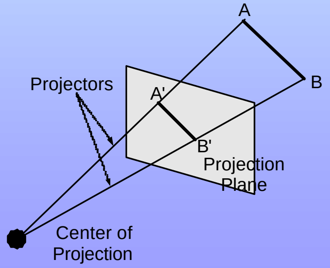
- Some viewing techniques better suited for viewing single objects rather than entire scenes
- Viewing an object from the outside (external viewing)
  - Engineering, External Buildings
- Viewing an object from within (internal viewing)
  - Internal Buildings, Games

Definitions

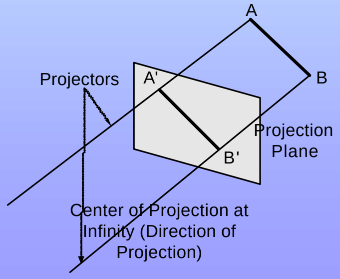
- *Projection*: a transformation that maps from a higher dimensional space to a lower dimensional space (e.g. 3D->2D)
- *Center of projection* (CoP): the position of the eye or camera with respect to which the projection is performed (also eye point, camera point, proj. reference point)
- *Projection plane*: in a 3D->2D projection, the plane to which the projection is performed (also view plane)

Projectors

- Projectors: lines from coordinate in Original Space to coordinate in Projected Space



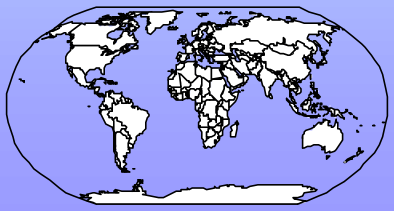
Perspective: Distance to CoP is finite



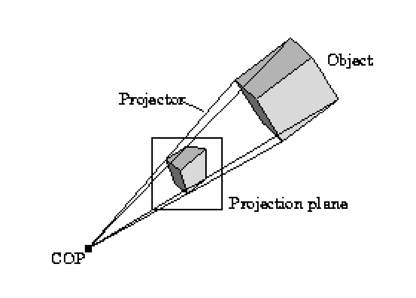
Parallel: Distance to CoP is infinite

Planar Geometric Projections

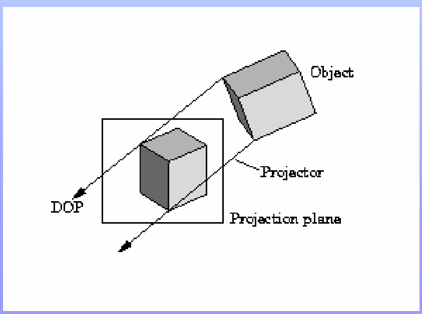
- Projection onto a plane
- Projectors are straight lines
- Alternatives:
  - Some Cartographic Projections
  - Omnimax



Center of Projection



Perspective View

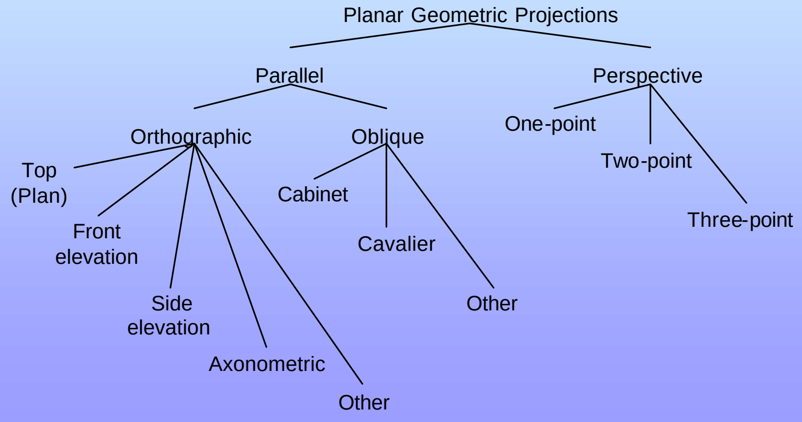


Parallel View

Viewing Classification – Planar Geometric Projections

- Parallel
  - Orthographic
    - Top (Plan)
    - Front
    - Side
    - Axonometric
      - Isometric
      - Dimetric
      - Trimetric
  - Oblique
    - Cabinet
    - Cavalier
    - Other
- Perspective
  - One-Point
  - Two-Point
  - Three-Point

Projections



- Planar Geometric Projections
  - Parallel
    - Orthographic
      - Top (Plan)
      - Front elevation
      - Side elevation
      - Axonometric
      - Other
    - Oblique
      - Cabinet
      - Cavalier
      - Other
  - Perspective
    - One-point
    - Two-point
    - Three-point

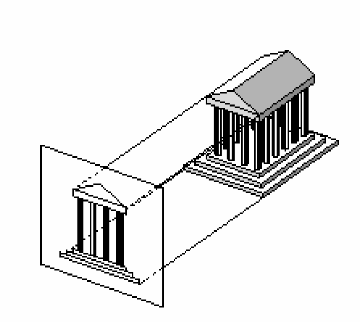
Parallel Projections

- Orthographic: Direction of projection is orthogonal to the projection plane
  - Elevations: Projection plane is perpendicular to a principal axis
    - Front
    - Top (Plan)
    - Side
  - Axonometric: Projection plane is not orthogonal to a principal axis
  - Isometric: Direction of projection makes equal angles with each principal axis.

- Oblique: Direction of projection is not orthogonal to the projection plane; projection plane is normal to a principal axis
  - Cavalier: Direction of projection makes a 45° angle with the projection plane
  - Cabinet: Direction of projection makes a 63.4° angle with the projection plane

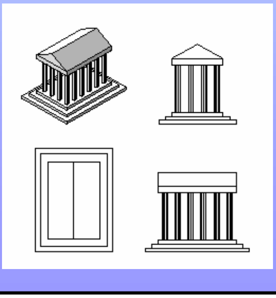
Parallel Projections: Orthographic Projections

- Parallel Projectors Perpendicular to Projection Plane
- Special Case of Perspective Projection



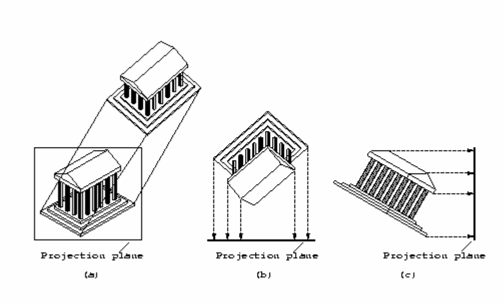
Orthographic Projections: Multiview

- Classical Drafting Views
- Preserves both distance and angles
- Suitable to Object Views, not scenes
- Front-Elevation
- Projection Plane Parallel to Principle Faces
- Top or Plan-Elevation
- Side-Elevation



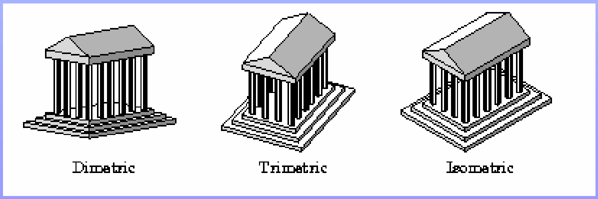
Orthographic Projections: Axonometric Projections

- Projection plane can have any orientation to object



Axonometric Projections

- Isometric
  - Symmetric to three faces
- Dimetric
  - Symmetric to two faces
- Trimetric
  - General Axonometric case

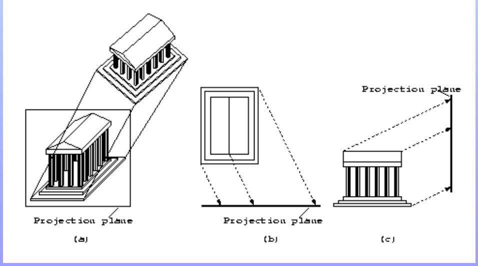


Axonometric Projections Cont.

- Foreshortening:
  - Length is shorter in image space than an object space
  - Uniform Foreshortening
  - (As opposed to perspective projections where foreshortening is dependent on distance from object to COP
- Parallel lines preserved
- Angles are not preserved

Oblique Projections

- **Parallel Projections not Perpendicular to Projection Plane**



Oblique Projection Types

- **Cavalier**
  - 45-degree Angles from Projection Plane
- **Cabinet**
  - Arctan(2) or 63.4-degree Angles from Projection Plane

Perspective Projections

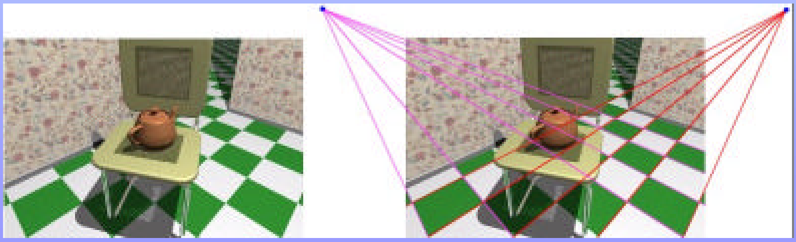
- One-point:
  - One principal axis cut by projection plane
  - One axis vanishing point

- Two-point:
  - Two principal axes cut by projection plane
  - Two axis vanishing points

- Three-point:
  - Three principal axes cut by projection plane
  - Three axis vanishing points

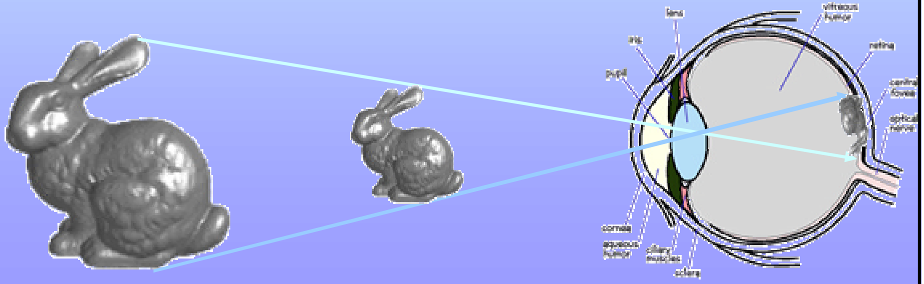
Perspective Projections

- First discovered by Donatello, Brunelleschi, and DaVinci during Renaissance
- Objects closer to viewer look larger



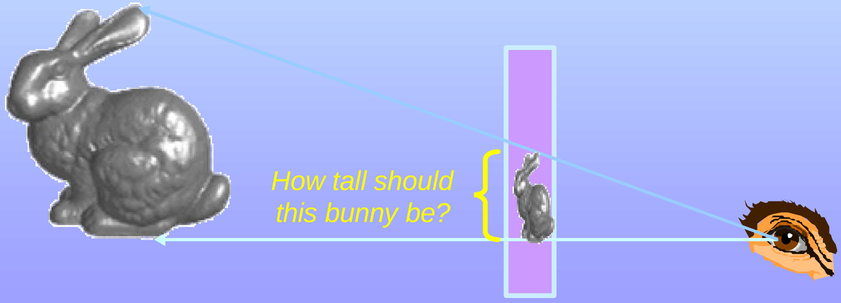
Perspective Projection

- In the real world, objects exhibit *perspective foreshortening*: distant objects appear smaller
- The basic situation:



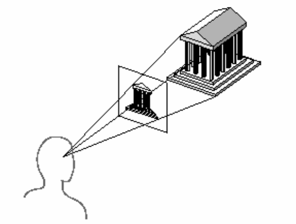
Perspective Projection

- When we do 3-D graphics, we think of the screen as a 2-D window onto the 3-D world:

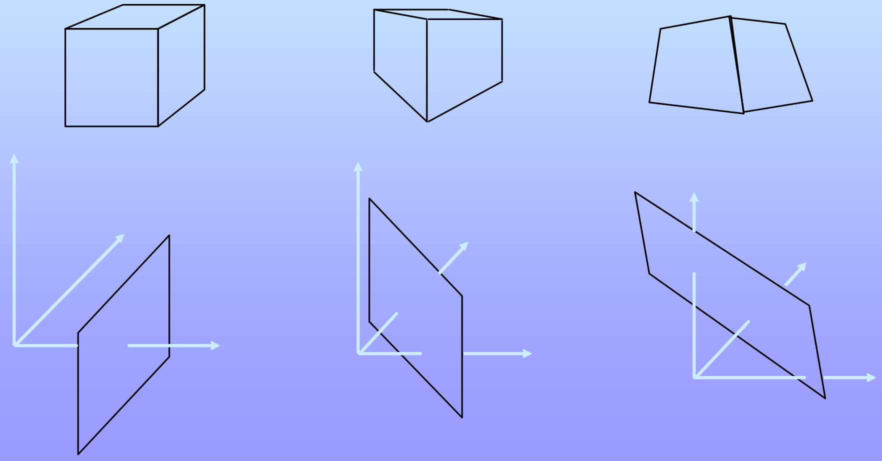


Perspective Views

- Objects further away look smaller
- Natural look
- Length is not preserved
  - Foreshortening of lines depends on distance from viewer



Perspective Projections



Vanishing Points

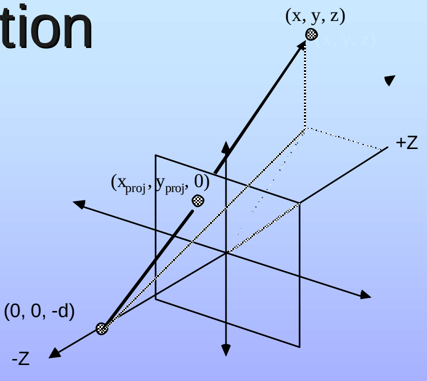
- Perspective Projection of any set of parallel line (not perpendicular to the projection plane) converge to a *vanishing point*
- Infinity of vanishing points
  - one of each set of parallel lines

Axis Vanishing Points

- Vanishing point of lines parallel to one of the three principal axes
- There is one axis vanishing point for each axis cut by the projection plane
- At most, 3 such points
- Perspective Projections are categorized by number of axis vanishing points

---

One-Point Projection



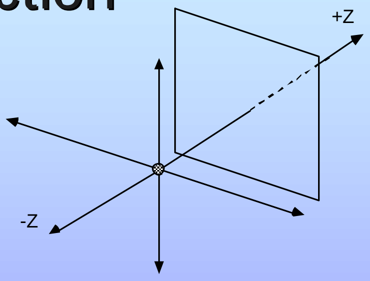
Center of Projection on the negative z-axis with viewplane in the x-y plane.

$$x_{projected} = xd/(d+z) = x/(1+(z/d))$$

$$y_{projected} = yd/(d+z) = y/(1+(z/d))$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \\ 1+(z/d) \end{bmatrix} = \begin{bmatrix} x/(1+(z/d)) \\ y/(1+(z/d)) \\ 0 \\ 1 \end{bmatrix}$$

Another One-Point Projection



Center of Projection at the origin with view plane parallel to the x-y plane a distance d from the origin.

$x_{projected}$ **= dx/z = x/(z/d)**

$y_{projected}$ **= dy/z = y/(z/d)**

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} x/(z/d) \\ y/(z/d) \\ d \\ 1 \end{bmatrix}$$

$\mathbf{M}_{\mathbf{per}}$

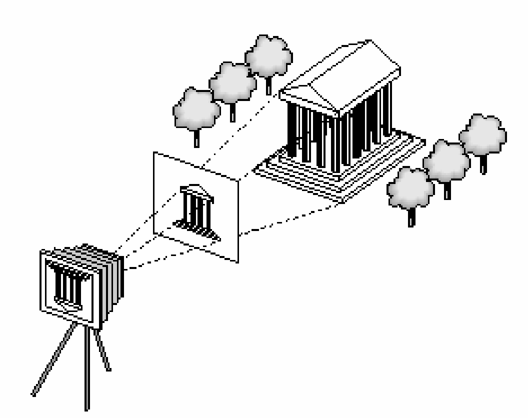
Points plotted are x/w, y/w where w = z/d

Specifying An Arbitrary 3-D View

- Two coordinate systems
  - World reference coordinate system *(WRC)*
  - Viewing reference coordinate system *(VRC)*
- First specify a view plane and coordinate system *(WRC)*
  - View Reference Point (VRP)
  - View Plane Normal (VPN)
  - View Up Vector (VUP)
- Specify a window on the view plane *(VRC)*
  - Max and min u,v values (Center of the window (CW))
  - Projection Reference Point (PRP)
  - Front (F) and back (B) clipping planes (hither and yon)

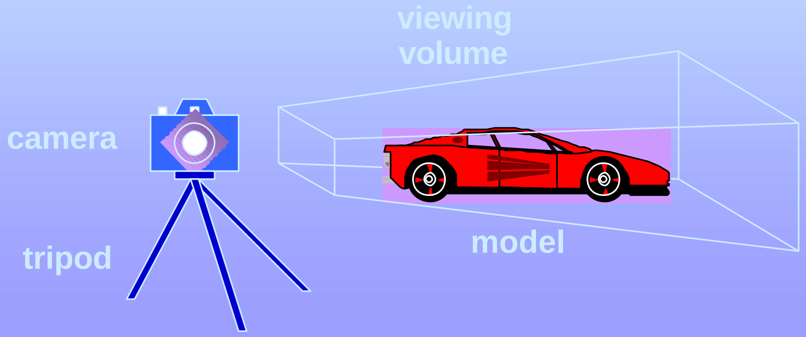
Synthetic Camera Model

Synthetic Camera Model

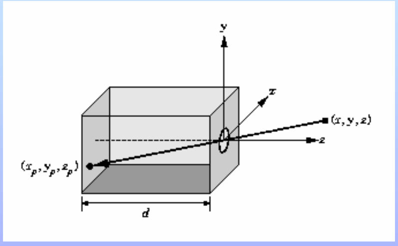


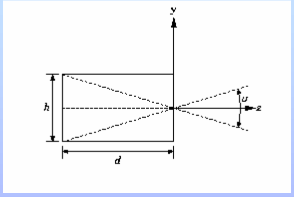
Camera Analogy

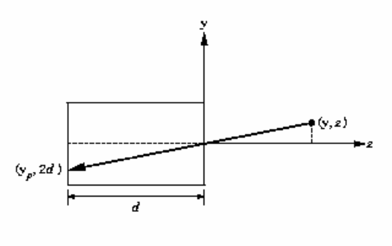
- 3D is just like taking a photograph (lots of photographs!)



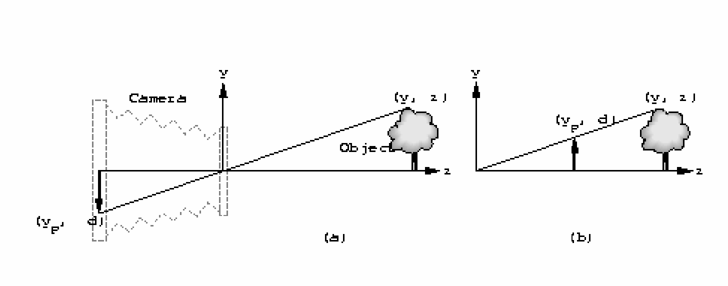
Camera







Camera Projection



Camera Analogy and Transformations

- Projection transformations
  - – adjust the lens of the camera
- Viewing transformations
  - – tripod–define position and orientation of the viewing volume in the world
- Modeling transformations
  - – moving the model
- Viewport transformations
  - – enlarge or reduce the physical photograph

Coordinate Systems and Transformations

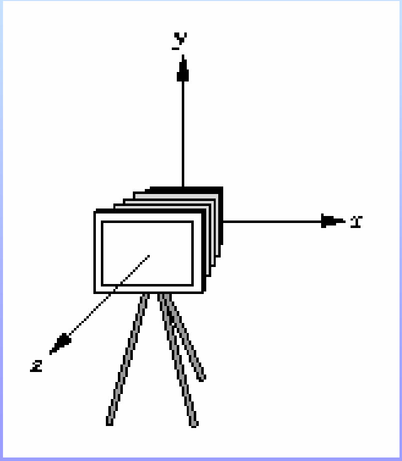
- Steps in Forming an Image
  - specify geometry (world coordinates)
  - specify camera (camera coordinates)
  - project (window coordinates)
  - map to viewport (screen coordinates)
- Each step uses transformations
- Every transformation is equivalent to a change in coordinate systems (frames)

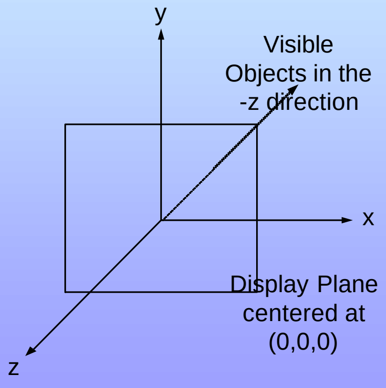
Projection Specification

Projection Characteristics

- Camera/Viewpoint Location
- Camera/Viewpoint Direction
- Camera/Viewpoint Orientation
- Camera/Viewpoint Lens (View Volume)
  - Width/Height of Lens
  - Front/Back Clipping Planes
- Parallel or Perspective Projections
  - Parallel is special case of Perspective

OpenGL Camera or Projection Coordinates





Visible Objects in the -z direction

Display Plane centered at (0,0,0)

Specifying Viewing Characteristics

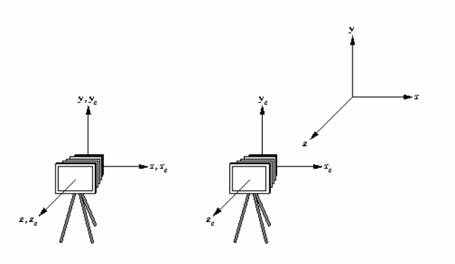
- Fixed Camera / Move World
- “Look At”
- View Volume / Display Plane Specification
- Vector Specification

Camera Location Relative to World Coordinates

- Can be thought of in two ways:
  - Camera location is specified in world coordinates
  - World coordinate frame is located in camera coordinates
- Camera transformations are reverse of World transformations

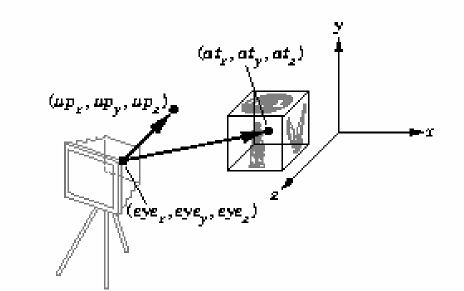
Fixed Camera / Move World

- Fix the camera at a specific location/orientation
- Transform the world such that the camera sees the world the “right” way – I.e., move the world, not the camera
- Typical approach for OpenGL



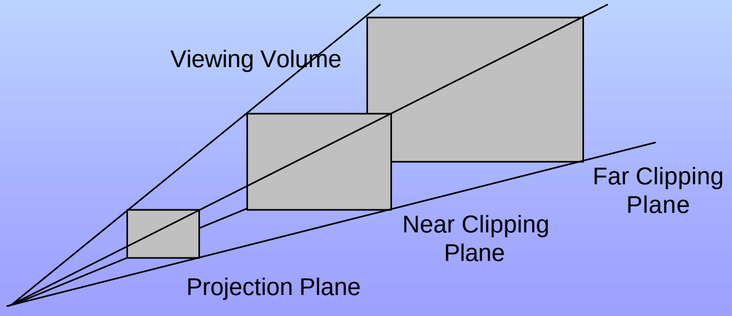
“Look At”

- Setup the camera to “Look At” the world
- Set the camera location
- Set the “look at” point
- Set the camera rotation angle
- Example: gluLookAt



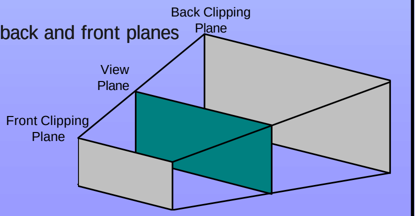
View Volume / Display Plane Specification

- Specify View Volume and Display Plane



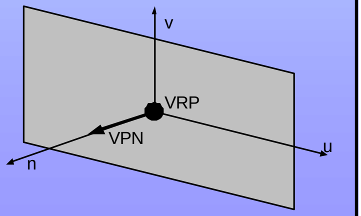
Fun with View Volumes

- All projection information is captured in view volume and projection plane.
- What happens if we play with these values?
  - Oblique Parallel Projections
  - Non-Right Frustum Perspective Projections
- View Volume Values
  - Box
    - Lower-Left Corner/Upper Right Corner of back and front planes
    - Back Plane Angles/Front Plane Angles
  - Projection Plane
    - Lower-Left Corner/Upper-Right Corner
    - Angle
    - Location Relative to View Volume



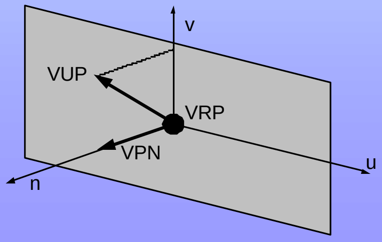
Vector Specification

- Most flexible method of viewing specification
- View plane may be anywhere with respect to the world
- Locate the View Plane (I.e. Projection Plane) by:
  - *View reference point* (VRP)
  - *View-plane normal* (VPN) (*n* axis)



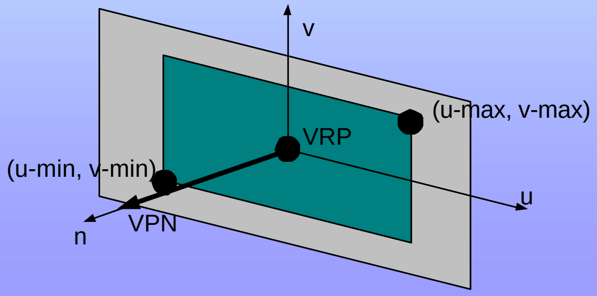
VRC Coordinate System

- Viewing-Reference Coordinate (VRC) System
  - *n*-axis - *along View-plane normal* (VPN)
  - *v*-axis – projection of *View-up Vector* (VUP)
  - *u*-axis – Right hand coordinate system



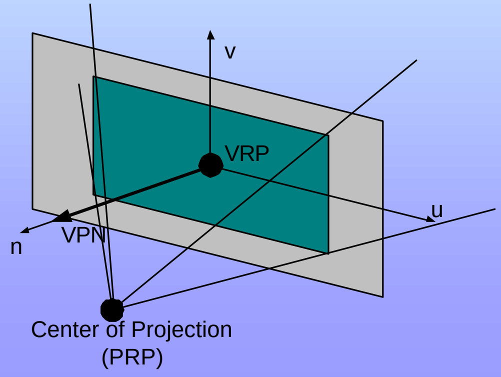
Viewing Window

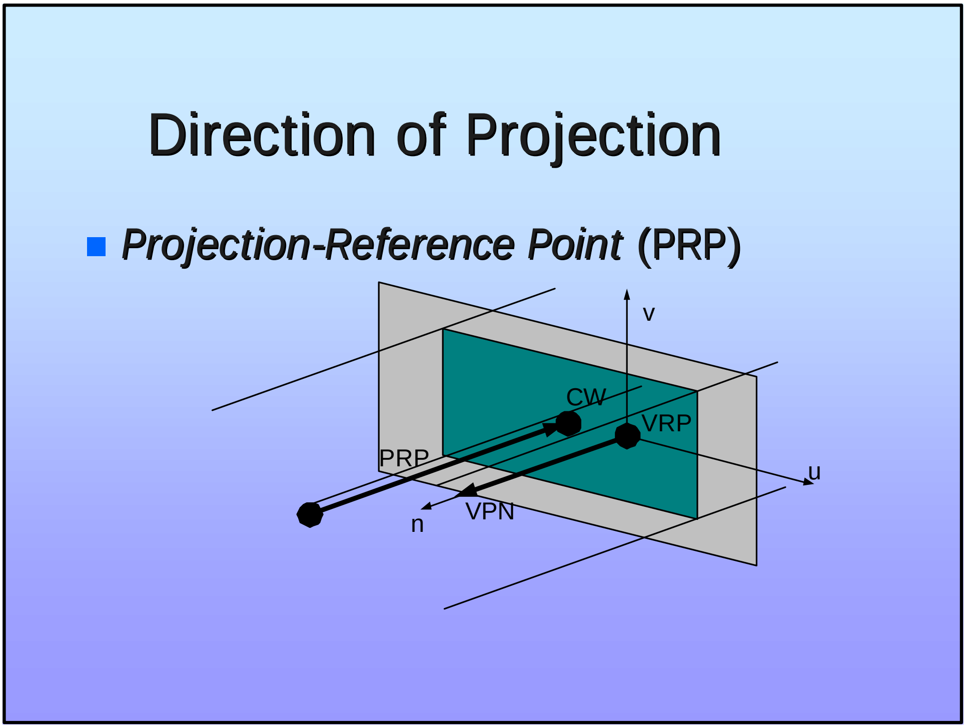
- Min/Max *u* and *v* ranges
- Need not be symmetrical around VRP



Center of Projection

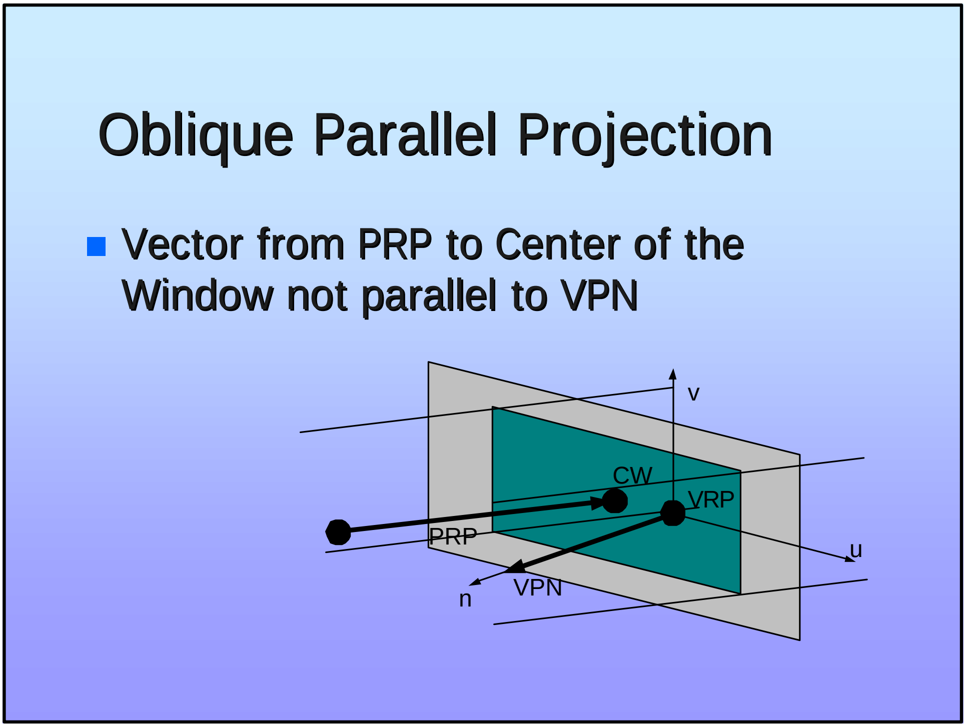
- *Projection-Reference Point* (PRP)





Direction of Projection

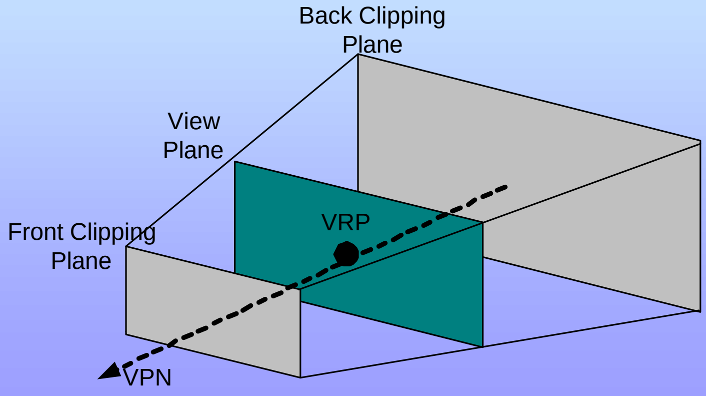
- *Projection-Reference Point* (PRP)



Oblique Parallel Projection

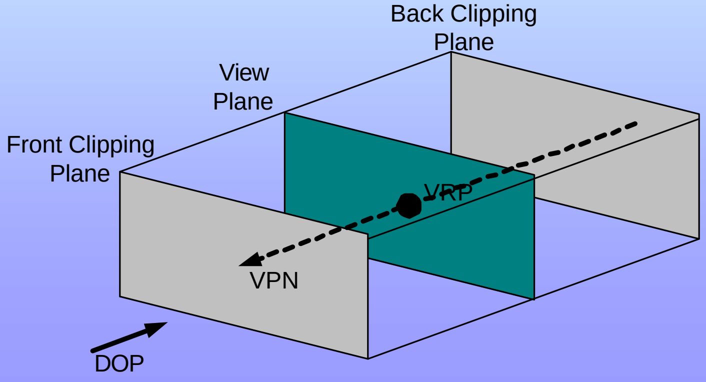
- Vector from PRP to Center of the Window not parallel to VPN

Perspective View Volumes

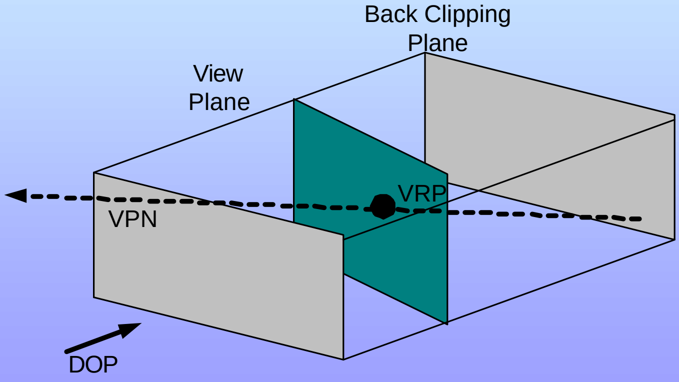


Orthogonal View Volumes

- Specification of Viewing Volume



Oblique Projections



Projection Specification in OpenGL

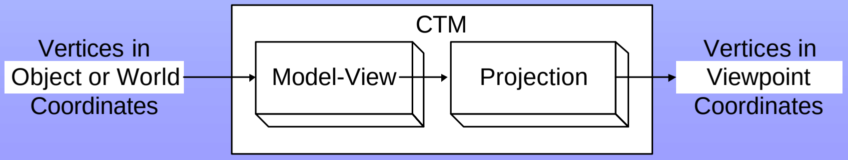
Projection Specification in OpenGL

- Locate World in Camera Coordinates
  - Alternative: gluLookAt encapsulates world to camera coordinate transformations
- Select View Type (Pespective or Orthogonal)
- Set View Volume (glFrustum, glOrtho, gluPerspective)
  - (near clipping plane is projection plane)
- Conceptually not as flexible as VRC system
  - Although with world coordinate transformation, you can obtain the same results

---

Projections in OpenGL

- Camera located at (0,0,0) pointing in the –*z* direction. Up is positive y direction.
- GL_Modelview Matrix controls the world
- GL_Projection Matrix controls the camera

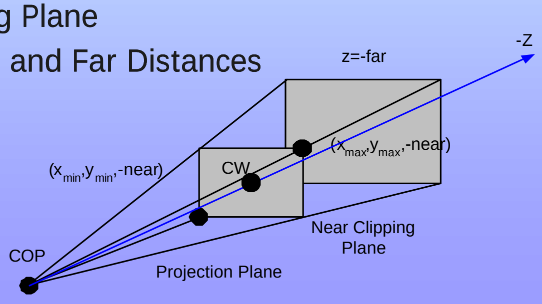


OpenGL

- **Move the world relative to the camera**
- **For example:**
  - Moving the camera +10 along the Z axis is equivalent to moving the world –10 along the Z axis
  - Is rotating the camera positively around the y axis the same as rotating the world negatively around the y axis?

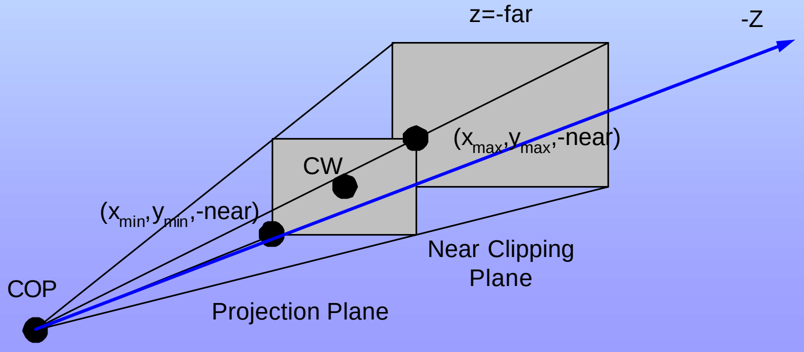
OpenGL Implementation of View Volume / Display Plane

- glFrustum
  - Display Plane is same as Near Clipping Plane
  - Specify Lower-Left and Upper Right Corners of Near Clipping Plane
  - Specify Near and Far Distances



glFrustum Variations

- Display Axis (-z) does not have to go through the center of the window



OpenGL Alternatives

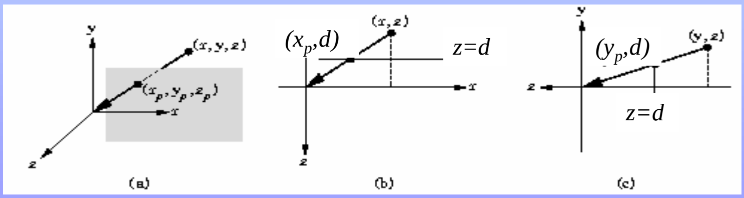
- **Alternative: gluPerspective**
  - Specifies Frustum via viewing angles
  - (ala Camera Lens)
- **glOrtho – Orthogonal Projection**

Projection Transformations

NOTE: Throughout the following discussions, we assume an OpenGL-like camera coordinate system (COP at the origin, DOP along the z axis, $z_{vp} < 0$). The concepts are the same for any arbitrary viewing configuration but the math is slightly more complicated.

Projecting Points onto a Display Plane*

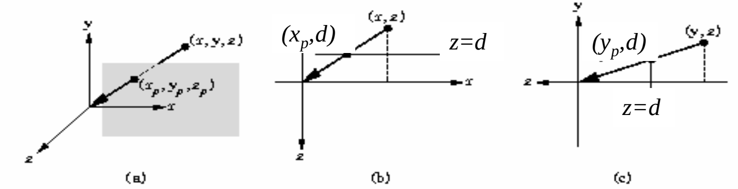
- Projection of world onto display plane involves a perspective transformation:
  - $p' = M_{per} p$
  - Not affine (parallel lines do not remain parallel)
  - Non-reversable



*Note: Angels book has several errors in the mathematics in this area

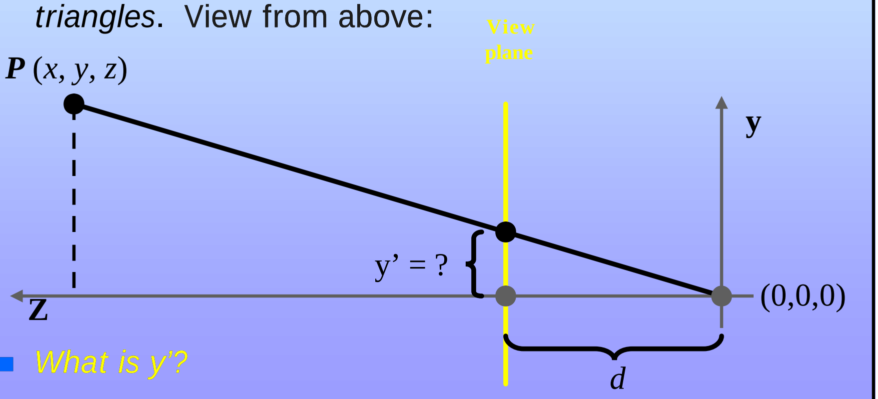
Projection of point onto display plane

- Observe: $\frac{x}{z} = \frac{x_p}{d}, \quad x_p = \frac{x}{z/d}$
- (Same for y)
- Results in non-uniform foreshortening



Perspective Projection

- The geometry of the situation is that of *similar triangles*. View from above:



- *What is y’?*

Perspective Projection

- Desired result for a point $[x, y, z, 1]^T$ projected onto the view plane:

$$\frac{x'}{d} = \frac{x}{z}, \quad \frac{y'}{d} = \frac{y}{z}$$

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

- *What could a matrix look like to do this?*

Use of Homogeneous Coordinates

- We can use homogeneous coordinates to make perspective transformation easier
- Homogeneous Representation of point: $p = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$
- Suppose, instead: $p = \begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix}$
- As long as w <> 0, we can recover original point

Use of w in homogeneous coordinates

- Let matrix *M* transforms point *p* into point *q*:

$$q = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = Mp = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

What is q?

- Then convert *q* back to a 3D point:

$$q' = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ \frac{z}{z/d} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}$$

- **Thus *q'* is our projection of *p* onto the display plane!**

A Perspective Projection Matrix

- Example:

$$\begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Or, in 3-D coordinates:

$$\left(\frac{x}{z/d}, \quad \frac{y}{z/d}, \quad d \right)$$

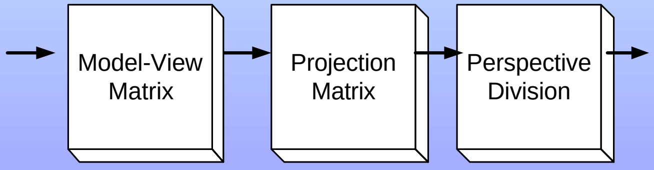
Thus, M is $\mathbf{M}_{\text{pers}}$

- **Thus, the matrix M is used to project points onto a perspective display plane**

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

Simple Projection Pipeline

- Simple OpenGL-like Projection Matrix:



Orthogonal Projections, M_{ortho}

- Special Case of Perspective Projection
- Display Plane at *z*=0

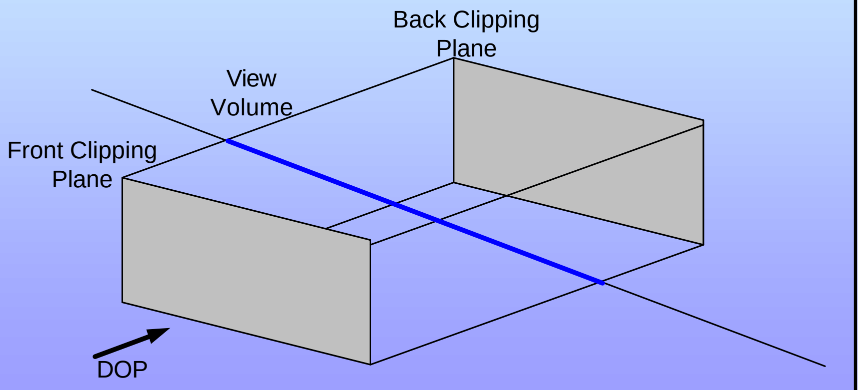
$$x_p = x,\quad y_p = y,\quad z_p = 0$$

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Now What?

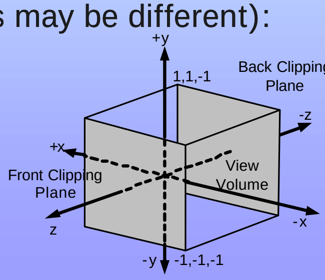
- Several issues are not address with the simple projection matrixes we have developed:
  - 3D Clipping Efficiency in a Frustum Viewing Volume
  - Hidden Surface Efficiency
- Solution: Use Projection Normalization
  - Get ride of perspective and other problem projections!
  - Everything is easier in an canonical, orthogonal world!

Clip Against View Volume



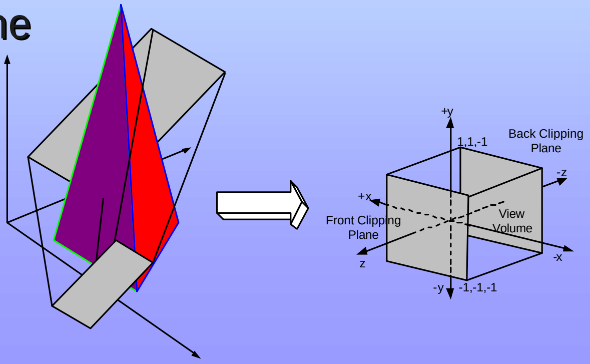
Canonical View Volume

- Define Viewing Volume via Canonical View Volumes
- Plus: Easier Clipping
- Minus: Another Transformation
- OpenGL volume (other APIs may be different):



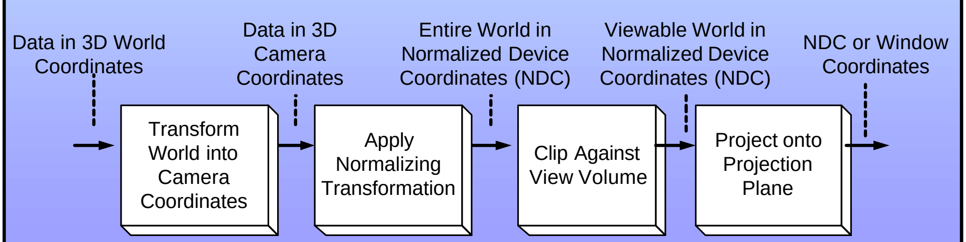
Projection Normalization

- Distort world until viewing volume in world fits into a parallel canonical view volume

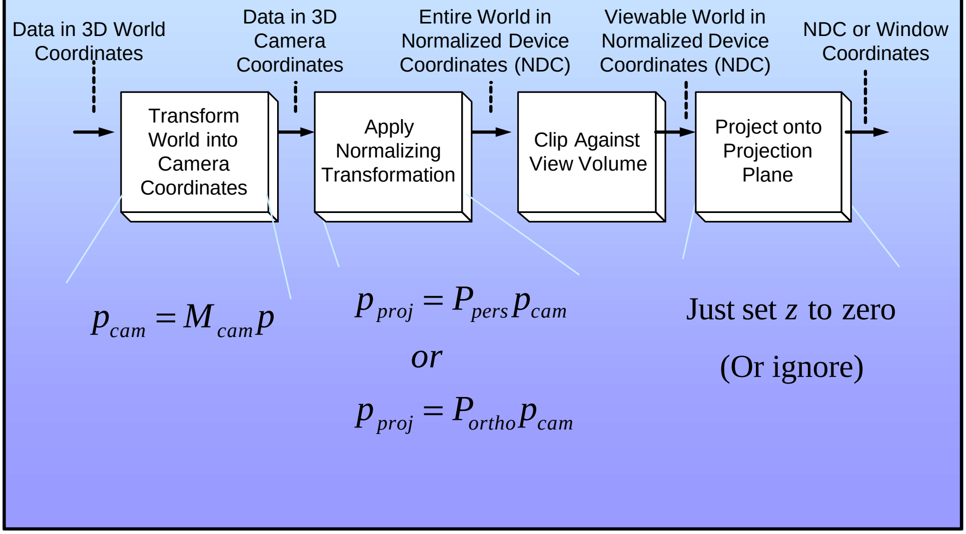


3D Viewing Transformation

- Input 3D World Coordinates
- Output 3D Normalized Device Coordinates
  - (a.k.a. Window Coordinates)



Projection Matrixes



$$p_{cam} = M_{cam} p$$

$$p_{proj} = P_{pers} p_{cam}$$

or

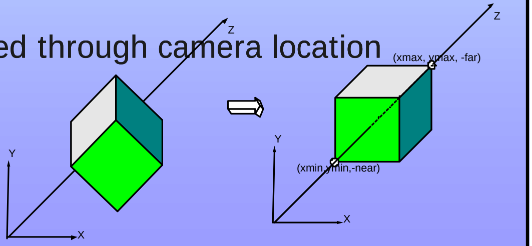
$$p_{proj} = P_{ortho} p_{cam}$$

Just set *z* to zero

(Or ignore)

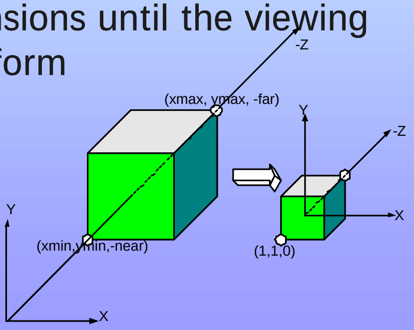
Camera Transformation in Orthogonal Views (M_{cam})

- Convert World to Camera Coordinates
  - Camera at origin, looking in the –z direction
    - Display plane center along the z axis
  - Combinations of translate, scale, and rotate transformations
  - Can be accomplished through camera location specification



Projection Normalization for Orthographic Views (P_{ortho})

- Translate along the z axis until the front clipping plane is at the origin
- Scale in all three dimensions until the viewing volume is in canonical form

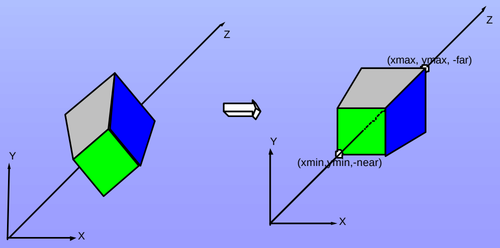


Projection Normalization for Orthographic Views

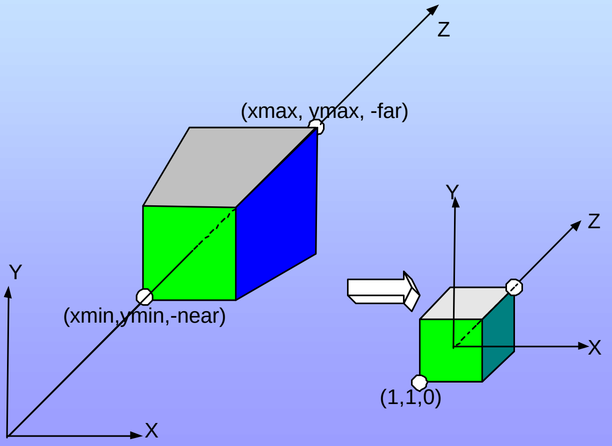
$$P_{ortho} = S_{ortho} T_{ortho} = \begin{bmatrix} \frac{2}{x_{\max} - x_{\min}} & 0 & 0 & -\frac{x_{\max} + x_{\min}}{x_{\max} - x_{\min}} \\ 0 & \frac{2}{y_{\max} - y_{\min}} & 0 & -\frac{y_{\max} + y_{\min}}{y_{\max} - y_{\min}} \\ 0 & 0 & \frac{-2}{far - near} & -\frac{far + near}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Camera Transformation for Perspective Views (M_{cam})

- Convert World to Camera Coordinates
  - Camera (COP) at origin, looking in the –z direction
    - Display plane center along the z axis
  - Combinations of translate, scale, and rotate transformations
  - Can be accomplished through camera location specification

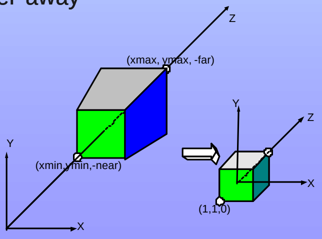


Projection Normalization for Perspective Views (P_{pers})



Projection Normalization for Perspective Views (P_{pers})

- 1) Convert viewing box to right frustum
  - This is because many APIs including OpenGL allow non-right viewing volumes
- 2) Scale the right frustum into canonical form
- 3) Convert viewing box (right frustum) to a right parallelpiped
  - “Shrinking” objects that are further away



Perspective-Normalization Matrix (N_{per})

- Converts Frustum View Volume into Canonical Orthogonal View Volume

$$N_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -\dfrac{far + near}{far - near} & -\dfrac{2\, far \cdot near}{far - near} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Projection Normalization for Perspective Views

$$P_{pers} = N_{per} SH = \begin{bmatrix} \dfrac{2(-near)}{x_{\max} - x_{\min}} & 0 & \dfrac{x_{\max} + x_{\min}}{x_{\max} - x_{\min}} & 0 \\ 0 & \dfrac{2(-near)}{y_{\max} - y_{\min}} & \dfrac{y_{\max} + y_{\min}}{y_{\max} - y_{\min}} & 0 \\ 0 & 0 & -\dfrac{far + near}{far - near} & -\dfrac{2\, far \cdot near}{far - near} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

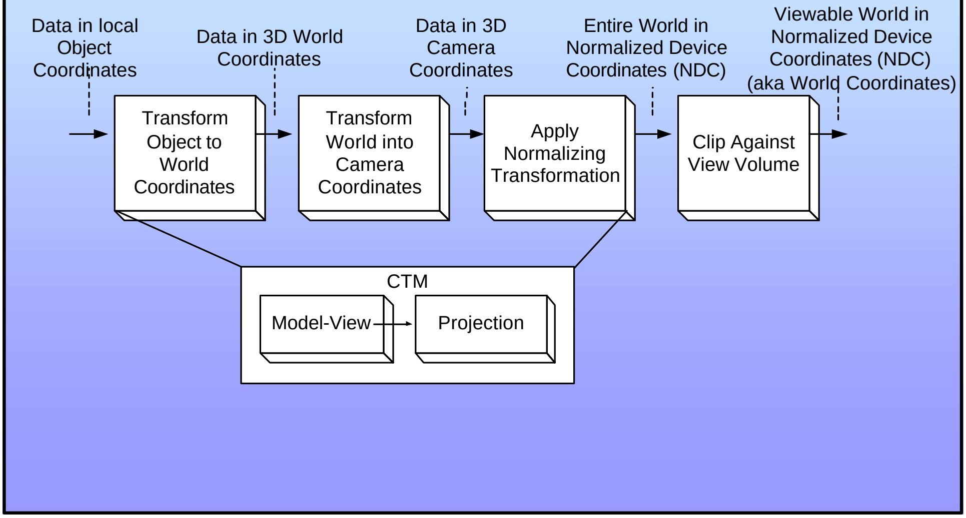
- Where H converts a non-right frustum to a right frustum
- Where S scales the frustum into a canonical perspective view volume
- Where N is the Perspective-Normalization Matrix

Project onto Projection Plane

- Since normalization changed all projections into an orthogonal projection:
  - Just ignore the z value!
  - In effect, a non-event!
- In reality, we retain the z-value for hidden-surface removal and shading effects.
- Viewable world now in Normalized Device Coordinates (NDC) or Window Coordinates

---

3D Viewing Summary



Data in local Object Coordinates

Transform Object to World Coordinates

Data in 3D World Coordinates

Transform World into Camera Coordinates

Data in 3D Camera Coordinates

Apply Normalizing Transformation

Entire World in Normalized Device Coordinates (NDC)

Clip Against View Volume

Viewable World in Normalized Device Coordinates (NDC) (aka World Coordinates)

CTM

Model-View

Projection

Matrix Operations

- Specify Current Matrix Stack

  `glMatrixMode( GL_MODELVIEW or GL_PROJECTION )`
- Other Matrix or Stack Operations

  `glLoadIdentity()` `glPushMatrix()`

  `glPopMatrix()`
- Viewport
  - usually same as window size
  - viewport aspect ratio should be same as projection transformation or resulting image may be distorted

  `glViewport( x, y, width, height )`

Projection Transformation

- Shape of viewing frustum
- Perspective projection

  `gluPerspective( fovy, aspect, zNear, zFar )`

  `glFrustum( left, right, bottom, top, zNear, zFar )`
- Orthographic parallel projection

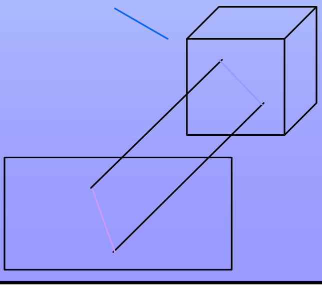
  `glOrtho( left, right, bottom, top, zNear, zFar )`

  `gluOrtho2D( left, right, bottom, top )`
  - calls **`glOrtho`** with z values near zero

Applying Projection Transformations

- Typical use (orthographic projection)

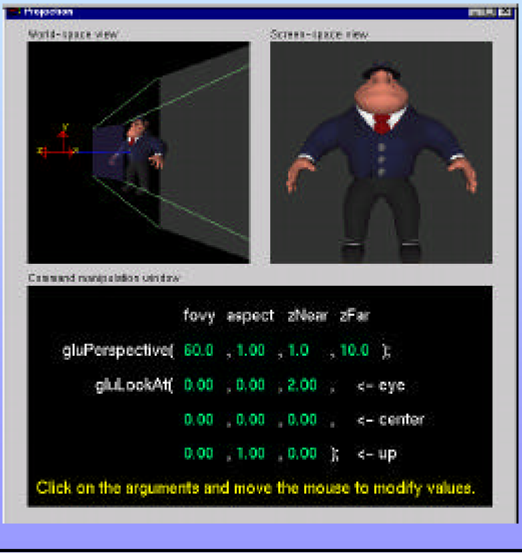
```
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
glOrtho( left, right, bottom, top, zNear, zFar );
```



Viewing Transformations

- **Position the camera/eye in the scene**
  - place the tripod down; aim camera
- **To “fly through” a scene**
  - change viewing transformation and redraw scene
- gluLookAt(eye$_x$, eye$_y$, eye$_z$,
  aim$_x$, aim$_y$, aim$_z$,
  up$_x$, up$_y$, up$_z$)
  - up vector determines unique orientation
  - careful of degenerate positions

Projection Tutorial



```
               fovy   aspect   zNear   zFar
gluPerspective(  60.0  , 1.00  , 1.0   , 10.0  );
    gluLookAt(   0.00  , 0.00  , 2.00  ,     <- eye
                 0.00  , 0.00  , 0.00  ,     <- center
                 0.00  , 1.00  , 0.00  );    <- up
Click on the arguments and move the mouse to modify values.
```

Modeling Transformations

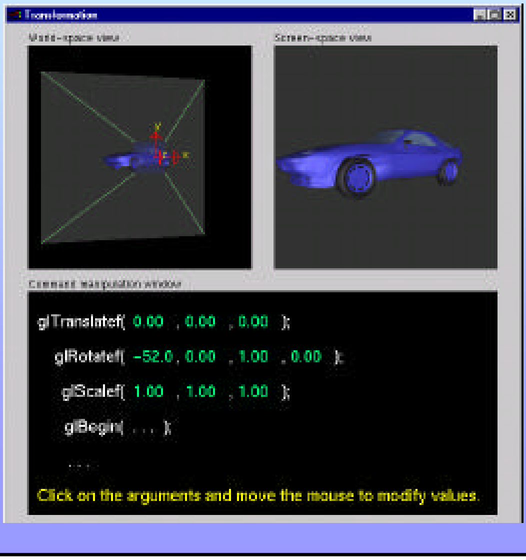
- Move object

  `glTranslate{fd}( x, y, z )`
- Rotate object around arbitrary axis $(x \quad y \quad z)$

  `glRotate{fd}( angle, x, y, z )`
  - angle is in degrees
- Dilate (stretch or shrink) or mirror object

  `glScale{fd}( x, y, z )`

Transformation Tutorial



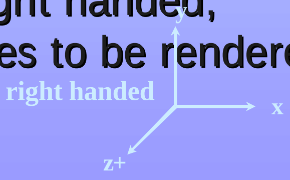
```
glTranslatef( 0.00  , 0.00  , 0.00  );
   glRotatef( -52.0 , 0.00  , 1.00  , 0.00  );
    glScalef( 1.00  , 1.00  , 1.00  );
     glBegin( . . . );
      . . .
Click on the arguments and move the mouse to modify values.
```

Connection: Viewing and Modeling

- Moving camera is equivalent to moving every object in the world towards a stationary camera
- Viewing transformations are equivalent to several modeling transformations
  - **`gluLookAt()`** has its own command
  - can make your own *polar view* or *pilot view*

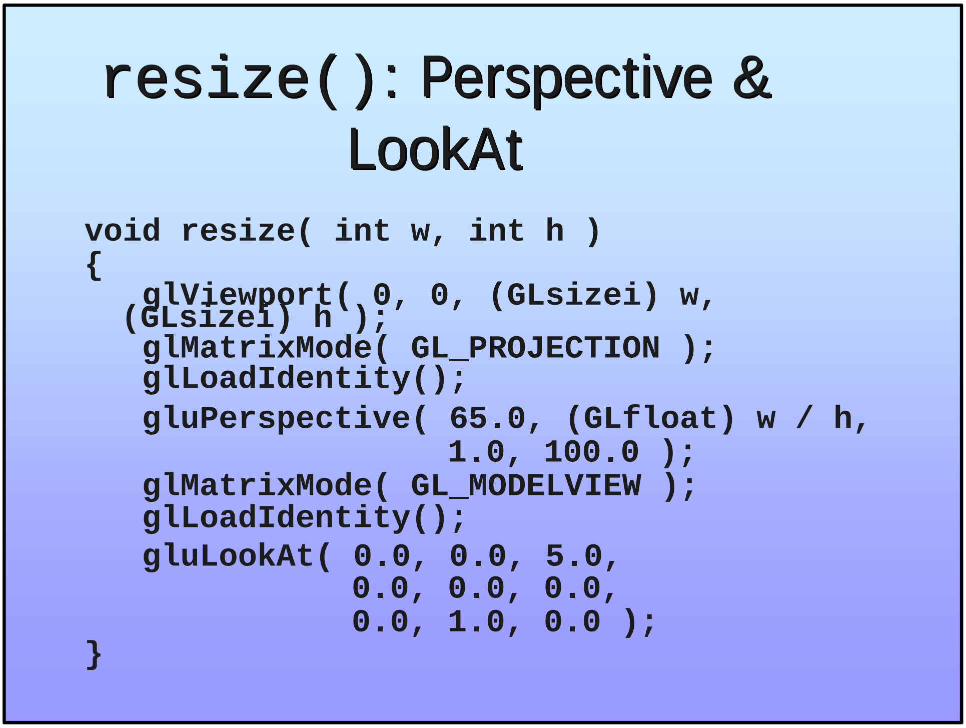
Projection is left handed

- Projection transformations (`gluPerspective, glOrtho`) are left handed
  - think of ***`zNear`*** and ***`zFar`*** as distance from view point
- Everything else is right handed, including the vertexes to be rendered



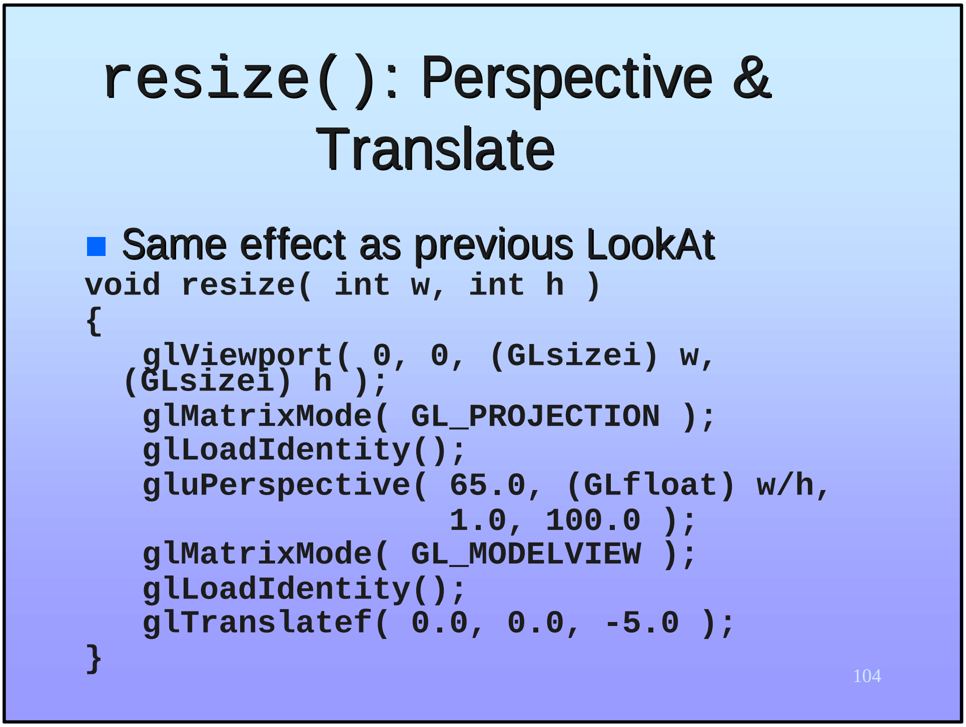
Common Transformation Usage

- 3 examples of **`resize()`** routine
  - restate projection & viewing transformations
- Usually called when window resized
- Registered as callback for **`glutReshapeFunc()`**



`resize()`: Perspective & LookAt

```
void resize( int w, int h )
{
   glViewport( 0, 0, (GLsizei) w,
  (GLsizei) h );
   glMatrixMode( GL_PROJECTION );
   glLoadIdentity();
   gluPerspective( 65.0, (GLfloat) w / h,
                   1.0, 100.0 );
   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity();
   gluLookAt( 0.0, 0.0, 5.0,
              0.0, 0.0, 0.0,
              0.0, 1.0, 0.0 );
}
```



`resize()`: Perspective & Translate

- Same effect as previous LookAt

```
void resize( int w, int h )
{
   glViewport( 0, 0, (GLsizei) w,
  (GLsizei) h );
   glMatrixMode( GL_PROJECTION );
   glLoadIdentity();
   gluPerspective( 65.0, (GLfloat) w/h,
                   1.0, 100.0 );
   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity();
   glTranslatef( 0.0, 0.0, -5.0 );
}
```

`resize()`: Ortho (part 1)

```
void resize( int width, int height )
{
   GLdouble aspect = (GLdouble) width /
  height;
   GLdouble left = -2.5, right = 2.5;
   GLdouble bottom = -2.5, top = 2.5;
   glViewport( 0, 0, (GLsizei) w, (GLsizei) h
  );
   glMatrixMode( GL_PROJECTION );
   glLoadIdentity();
```

… continued …

`resize()`: Ortho (part 2)

```
   if ( aspect < 1.0 ) {
      left /= aspect;
      right /= aspect;
   } else {
      bottom *= aspect;
      top *= aspect;
   }
   glOrtho( left, right, bottom, top, near,
  far );
   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity();
}
```

Compositing Modeling Transformations

- Problem 1: hierarchical objects
  - one position depends upon a previous position
  - robot arm or hand; sub-assemblies
- Solution 1: moving local coordinate system
  - modeling transformations move coordinate system
  - post-multiply column-major matrices
  - OpenGL post-multiplies matrices

Compositing Modeling Transformations

- **Problem 2: objects move relative to absolute world origin**
  - my object rotates around the wrong origin
    - make it spin around its center or something else
- **Solution 2: fixed coordinate system**
  - modeling transformations move objects around fixed coordinate system
  - pre-multiply column-major matrices
  - OpenGL post-multiplies matrices
  - must reverse order of operations to achieve desired effect

Additional Clipping Planes

- At least 6 more clipping planes available
- Good for cross-sections
- Modelview matrix moves clipping plane
- $Ax + By + Cz + D < 0$ clipped
- `glEnable( GL_CLIP_PLANEi )`
- `glClipPlane( GL_CLIP_PLANEi, GLdouble* coeff )`

Reversing Coordinate Projection

- Screen space back to world space

```
glGetIntegerv( GL_VIEWPORT, GLint viewport[4] )
glGetDoublev( GL_MODELVIEW_MATRIX, GLdouble mvmatrix[16] )
glGetDoublev( GL_PROJECTION_MATRIX,
                 GLdouble projmatrix[16] )
gluUnProject( GLdouble winx, winy, winz,
                 mvmatrix[16], projmatrix[16],
                 GLint viewport[4],
                 GLdouble *objx, *objy, *objz )
```

- `gluProject` goes from world to screen space