CS770/870 Spring 2017
3D Viewing

Related material in Angel text
6e: Ch 4.1-8

Scene Description

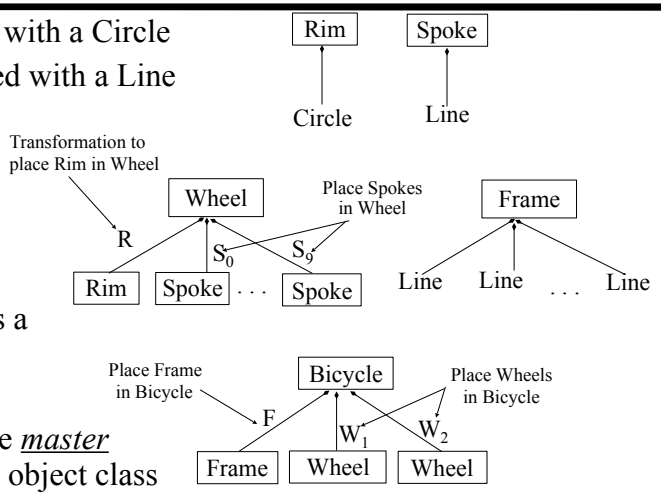
- A scene to be rendered is defined in a *world coordinate system* -- arbitrary floating point
- A scene is composed of multiple objects
  - Each object is defined (modeled) in its own *object coordinate system*
- Object coordinates must be mapped to world coordinates in order to be placed in the scene

Object Definition (Modeling)

- Objects are defined in *object coordinates*
  - usually the object is centered around the origin of the object coordinate system
- An object can be defined in terms of *other* objects (actually, *instances* of other objects)
  - the component object must be placed in the coordinate system of the composite object
  - requires mapping one *object coordinate system* to another

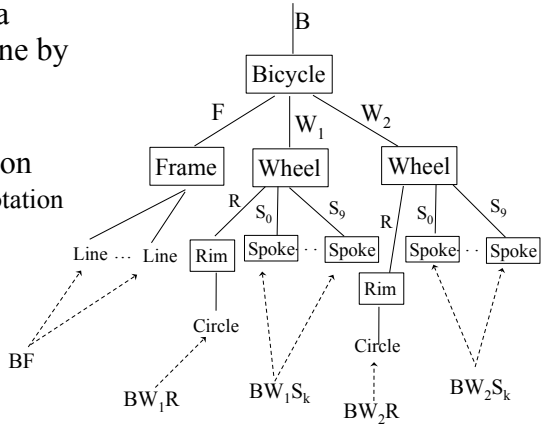
Composite Object Definition

- A Rim is modeled with a Circle
- A Spoke is modeled with a Line
- A Wheel contains
  - a Rim
  - 10 Spokes
- A Frame contains
  - a bunch of Lines
- A Bicycle contains a
  - Frame
  - 2 Wheels
- These represent the *master* definition for each object class

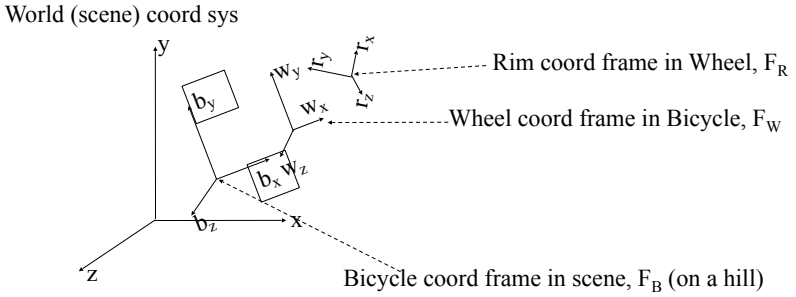


Composite Object Instance

- Represent an *instance* of a Bicycle, placed in the scene by matrix B
- Each component needs a transformation specification
  - We label the arcs with a notation for the matrix
- What is the composite matrix for each set of primitives?



Composition as Change of Coordinate Frame



To map the Rim's circle to the world, coordinates of the circle will be transformed by: $F_B F_W F_R = BWR$

So, the modeling transforms are the coordinate frame transforms

Rendering a Scene

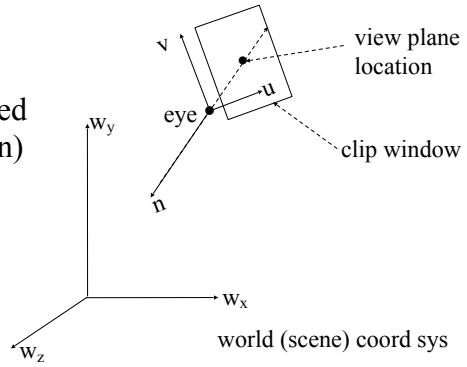
- Fixed view model
  - All rendering is based on a fixed view (one for parallel projection, one for perspective projection)
  - The user application transforms the objects in the scene so the desired portion of the scene is located in the view
- Variable view model
  - User defines a (more or less) arbitrary view
  - The graphics system transforms the scene to the viewing coordinate system

View/Camera Specification

- Transform a scene to an image needs:
  - a *view* specification
    - where is viewer
    - what is field of view
  - a *projection* specification
    - how is visible 3D scene projected onto 2D
- Old OpenGL viewing model still useful
  - it is straightforward and well-known
  - there is existing library support available

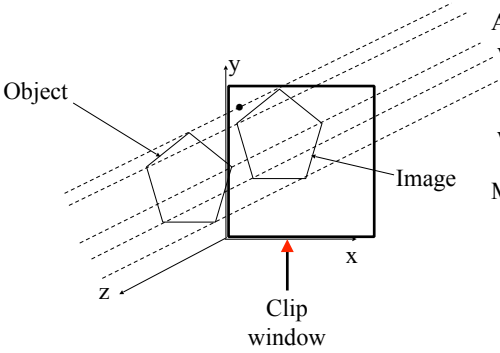
View Specification Model

- Allow user to define an eye coordinate system (a.k.a. view coordinate system) in world coordinates:
  - eye point
  - view plane normal (n)
  - view up direction (v)
  - u axis: make uvn right handed
  - view plane location, (along n)
  - clip window in view plane



Parallel Projection

- Object coordinates are "projected" using parallel projectors onto a 2D plane
- A rectangular portion of that plane (the clip window) is mapped to the display window

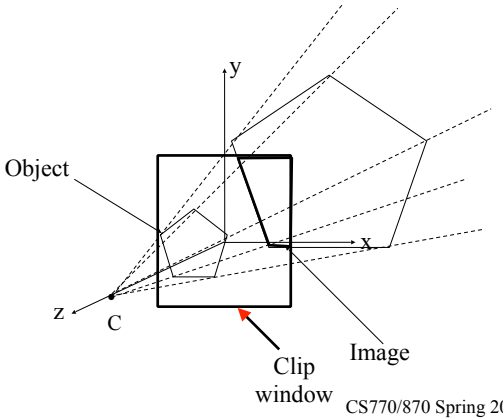


A common fixed parallel projection view:
view plane: xy plane
view plane normal = (0 0 1)
point on the plane = (0 0 0)
window: (0,0) to (1,1)

Makes projection and clipping very easy:
Ignore *z*
map clip window to display:
d_x = (int) w_x * windowWidth
d_y = (int) (1-w_y) * windowHeight

Perspective Projection

- Projectors originate at a center of projection; intersection with the projection plane yields 2D point
- Still need a clip window on the projection plane



One common fixed perspective view:
view plane: xy plane
view plane normal = (0 0 1)
window: (-1,-1) to (1,1)
Makes projection and clipping very easy:
See next slide
map clip window to display:
$d_x = (1 + w_x)* \text{displayW} / 2$
$d_y = (1 - w_y) * \text{displayH} / 2$
fill polygon

Perspective in Homogeneous Matrix

- Consider P_1:
  - *w* coordinate (4th) is not 1!
  - *normalizing* vector (divide by *w*) gets **perspective projection** of the point onto z = -1!

$$P_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ -z \end{bmatrix} \rightarrow \begin{bmatrix} -x/z \\ -y/z \\ -1 \\ 1 \end{bmatrix}$$

- Consider P_2:
  - x, y are same
  - z doesn't map to constant
  - z is called *pseudo depth*
  - it's called a **perspective transformation**

$$P_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z+1 \\ -z \end{bmatrix} \rightarrow \begin{bmatrix} -x/z \\ -y/z \\ -(z+1)/z \\ 1 \end{bmatrix}$$

This fails if z=0, but since the center of projection is on the z=0 plane, the ray from cop to a point on the same plane cannot intersect the projection plane.

Perspective Depth

- Is depth ordering preserved?

  Assume $z_1 < z_2$ and $z_1 < 0$ and $z_2 < 0$.

  Show that $z'_1 < z'_2$ where $z'_k = -(z_k+1)/z_k$

  Take inverse of both sides; inverse reverses inequality

  $1/z_1 > 1/z_2$ *Note: this would* ***not*** *be true if $z_1 < 0$ and $z_2 > 0$*

  Add 1 to both sides; inequality doesn't change

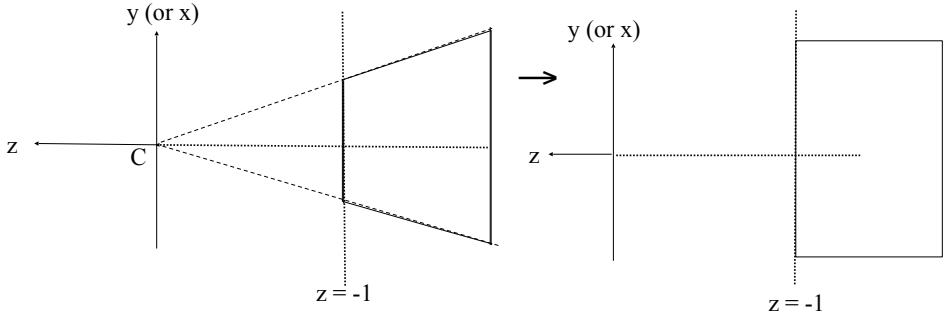
  $1+1/z_1 > 1+1/z_2 \equiv (z_1+1)/z_1 > (z_2+1)/z_2$

  Multiply both sides by -1, reverses inequality

  $-(z_1+1)/z_1 < -(z_2+1)/z_2 \equiv z'_1 < z'_2$ *QED*

Perspective Transformation

- If we define the *perspective projection* matrix as P_2 and we do the homogenous division (in this case, the perspective division), the frustum is mapped to a box:



Clipping Coordinates

- Perspective transformation maps 3D frustum to a 3D box in which z-order is preserved
- Orthographic view volume is already a 3D box
- Clipping is simpler if clipping volume is a box and all sides are defined along the major axes:
  - translate/scale box to cube centered at origin of size 2
- This defines the *clipping coordinate system*

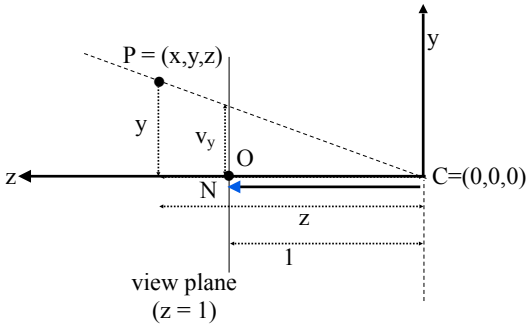
Note: this approach unifies orthographic and perspective viewing specifications!

Normalized Projection Coordinates

- 3D version of Normalized Device Coords
- Exactly same coordinates as Clipping Coordinates, but an object in NPC should have no vertices outside the clipping volume
- Objects in NPC are mapped to screen coords
  - projected orthographically and converted to screen coords inside 2D viewport (in the display's "window")

Perspective Projection Example

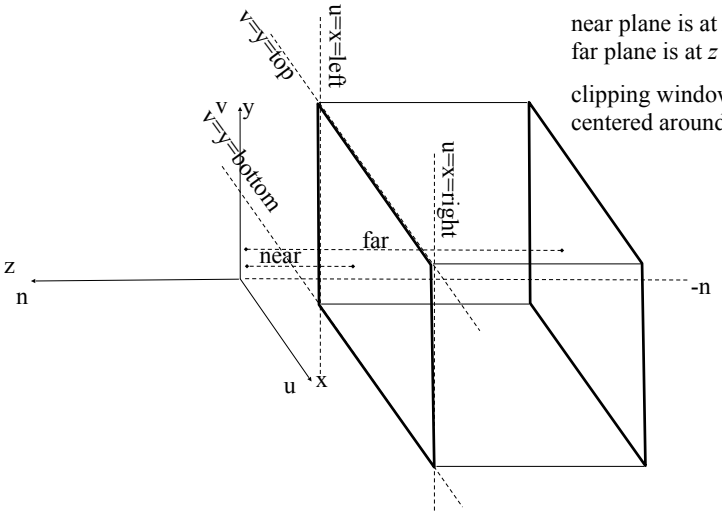
- Simple projection specification yields simple computations:
  - center of projection: C = (0 0 0)
  - view plane origin, O: (0 0 1) — a point on the view plane
  - view plane normal, N: (0 0 1)
  - window bounds: (-1,-1) to (1,1)



1. Project P=(x,y,z) onto view plane
   What is v_y ?
   by similar triangles:
   $v_y / y = 1 / z$
   i.e., $v_y = y / z$
   and $v_x = x / z$
   where v_x , v_y are *view plane* coordinates

2. Map clip window to display:
   $d_x = (1+v_x) * displayW / 2$
   $d_y = (1-v_y) * displayH / 2$

OpenGL Default Orthographic View

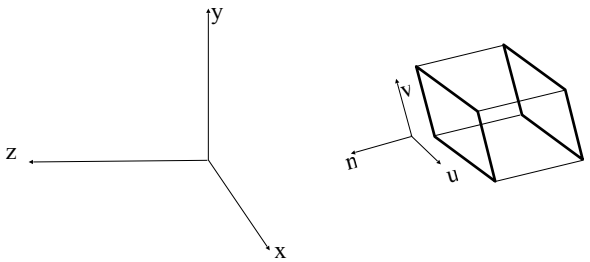


near plane is at $z = -near$
far plane is at $z = -far$

clipping window need not be centered around (0,0)

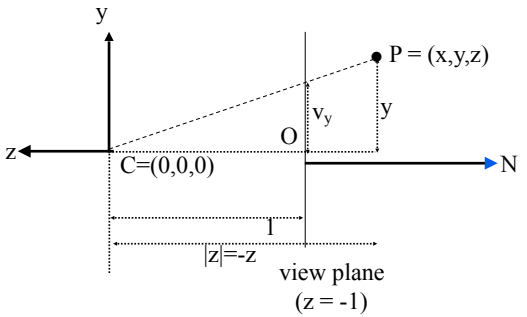
"Arbitrary" OpenGL View

- *gluLookAt* (and *glOrtho* can place eye coord system (frame) anywhere



OpenGL Perspective Projection

- OpenGL's default view is a little different: scene is in –z volume
  - center of projection: C = (0 0 0)
  - view plane origin, O: (0 0 -1)
  - view plane normal, N: (0 0 -1)
  - window bounds: (-1,-1) to (1,1)

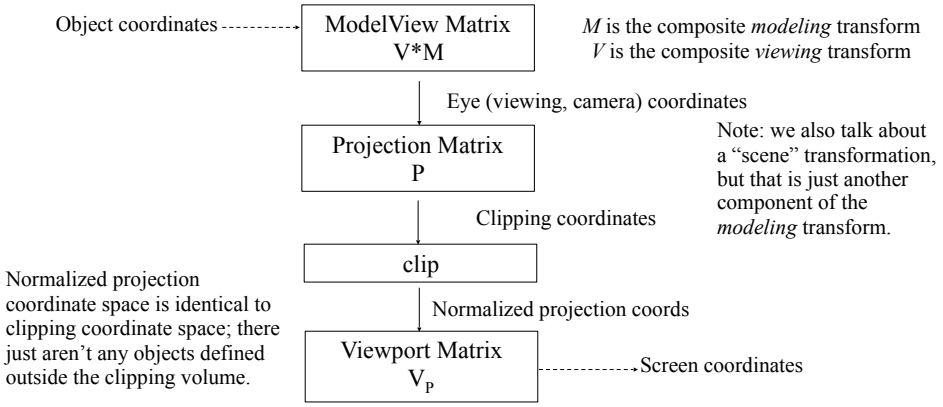


1. Project P=(x,y,z) onto view plane
   What is v_y ?
   by similar triangles:
   $v_y / y = 1 / |z| = 1/(-z) = -1/z$
   i.e., $v_y = -y / z$
   and $v_x = -x / z$
   where v_x , v_y are *view plane* coordinates

2. Map clip window to display:
   $d_x = (1+v_x) * displayW / 2$
   $d_y = (1-v_y) * displayH / 2$

"Old" OpenGL Viewing Pipeline

- Viewing pipeline transforms scene to screen



All these components are in the modern OpenGL/GLSL Pipeline

"Old" OpenGL Synthetic Camera

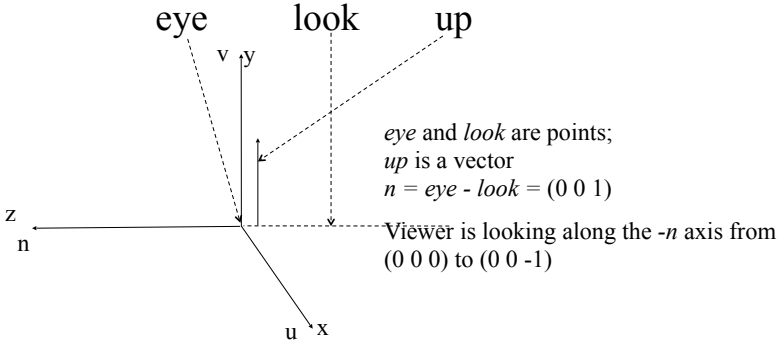
- OpenGL's deprecated version of the synthetic camera is represented by the *gluLookAt* utility function

  gluLookAt(eye.x, eye.y, eye.z,
  look.x, look.y, look.z,
  up.x, up.y, up.z)

  - eye: is the origin of the eye coordinate system in world coords
  - look: viewer is looking *at* the look point in world coords
    n-axis of eye coordinate system = look - eye
  - up: any line parallel to up (in world coords) is vertical in the image
    v-axis of eye coord system is computed from *n* and up
    u-axis of eye coord system: $u = v \times n$.
- Also need a **projection** specification (in view coordinates):
  glFrustum(left, right, bottom, top, near, far)

gluLookAt

- Default : (0, 0, 0, 0, 0, -1, 0, 1, 0)

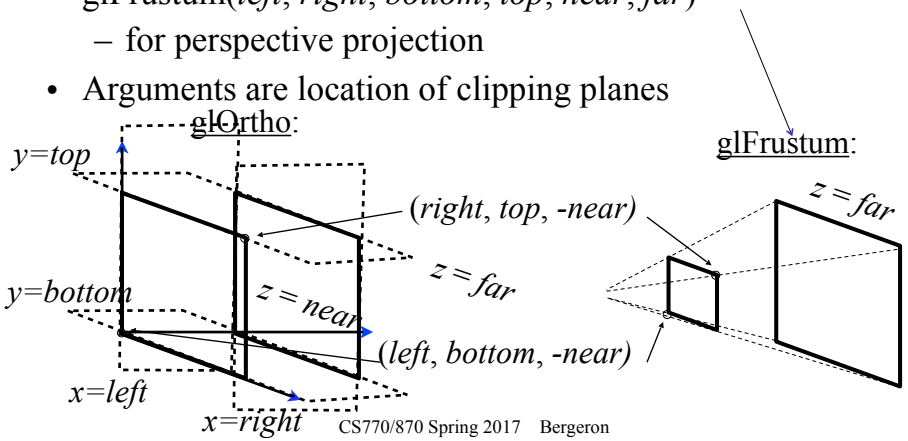


OpenGL Orthographic Projection

- Simplest deprecated OpenGL view is orthographic:
  glOrtho(left, right, bottom, top, near, far)
  - parameters are in eye coordinates
  - *left, right, bottom, top* define the bounds of the 2D window in the projection plane
    - *left* and *right* are measured on the *u*-axis
    - *bottom* and *top* are measured on the *y*-axis
  - *near, far* are the distances of the near and far clipping planes from (0,0,0) along the -*n* axis!

Specify Camera By Boundaries

- glOrtho(*left*, *right*, *bottom*, *top*, *near*, *far*)
  - for orthographic projection
- glFrustum(*left*, *right*, *bottom*, *top*, *near*, *far*)
  - for perspective projection
- Arguments are location of clipping planes



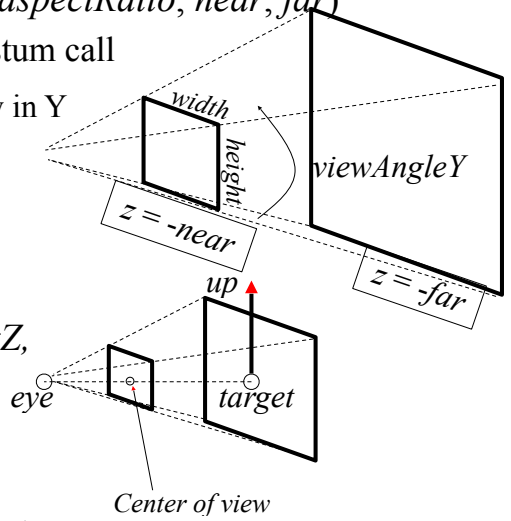
Specify Camera By Properties

- perspective(*viewAngleY*, *aspectRatio*, *near*, *far*)
  - Alternative to the glFrustum call

  viewAngleY is field of view in Y

  $aspectRatio = \frac{width}{height}$
- OR:

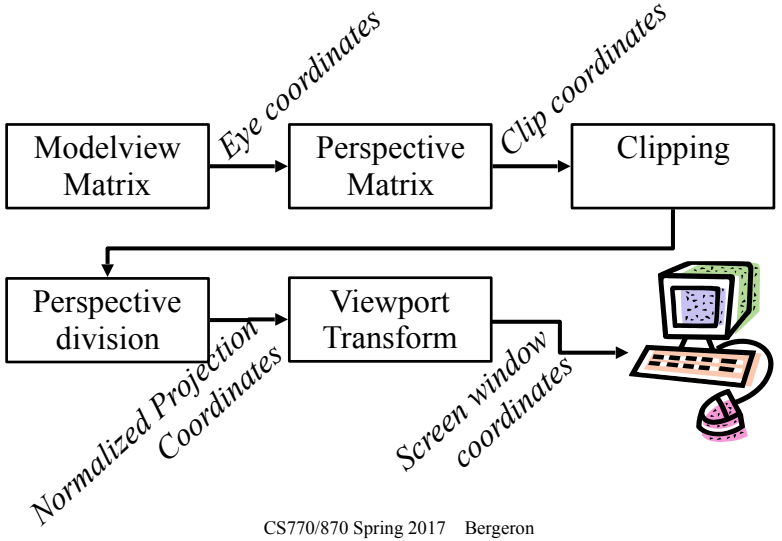
  lookAt(
  eyeX, eyeY, eyeZ,
  targetX, targetY, targetZ,
  upX, upY, upZ)



Best Way to Learn It?

- Use it, play with it
  - Parameterize view settings
  - Hook up view settings to interactive controls
  - Put 3D objects in a scene
  - Play with the controls & see what happens

Under the Hood: Graphics Pipeline



GLSL Matrix Warning

- GLSL chose to continue the C/OpenGL convention of storing 2d arrays in memory in *column-major* order. See demos

```
float[] model = { xSize,   0,    0, 0,
                    0,    ySize, 0, 0,
                    0,      0,   1, 0,
                  xLoc,   yLoc,  0, 1 };
```

- The "row" in the code is actually a column in the matrix!

Review

- Scene description
  - Hierarchical object modeling
- Scene rendering
- Projections
  - Parallel projection
  - Perspective projection
- Synthetic camera model
- OpenGL Viewing