3D Viewing

1

Introduction

- We saw how to use matrix transformations as a tool for arranging geometric objects in 2D or 3D space.
- A second important use of geometric transformations is in moving objects between their 3D locations and their positions in a 2D view of the 3D world. This 3D to 2D mapping is called a *viewing transformation*, and it plays an important role in object-order rendering
- We need to rapidly find the image-space location of each object in the scene.

2

2

Projection

- The 3D viewing process is inherently more complex than the 2 D viewing process
- The complexity added in the three dimensional viewing is because of the added dimension and the fact that even though objects are three dimensional the display devices are only 2D.
- The mismatch between 3D objects and 2D displays is compensated by introducing projections.
- The projections transform 3D objects into a 2D projection plane.

3

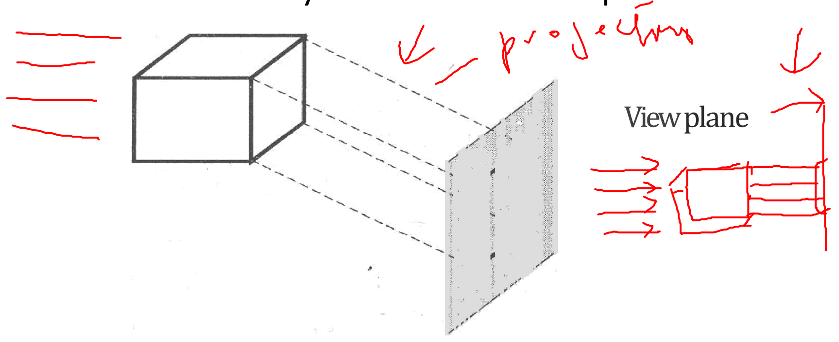
Projection

- There are two basic ways of projecting objects onto the view plane :
  - Parallel projection
  - Perspective projection

4

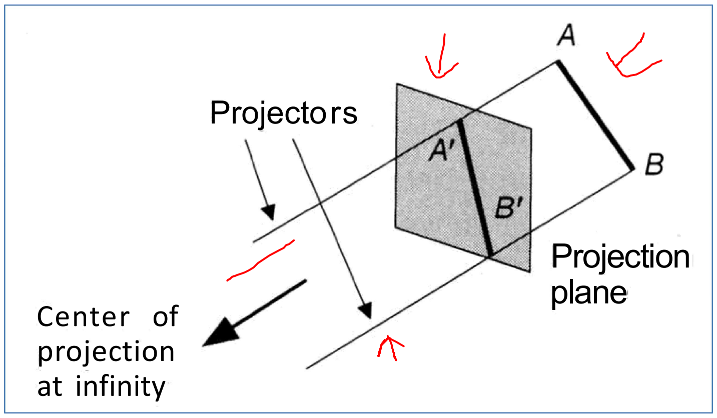
Parallel Projection

- In **parallel projection**, coordinate positions are transformed to the view plane along parallel lines.
- In parallel projection, z coordinate is discarded and parallel lined from each vertex on the object are extended until they intersect the view plane.



5

Parallel Projection



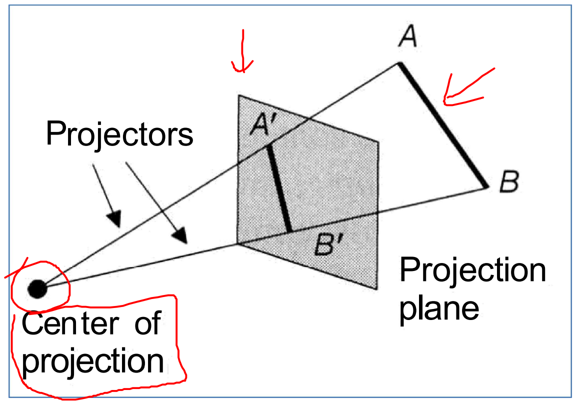
6

Perspective Projection

- In **perspective projection**, the lines of projection are not parallel. Instead, they all coverage at a single point called the center of projection or projection reference point.
- The perspective projection produces realistic views but does not preserve relative proportions.
- The object positions are transformed to the view plane along these converged projection lines and the projected view of an object is determines by calculating the intersection of the converged projection lines with the view plane

7

Perspective Projection

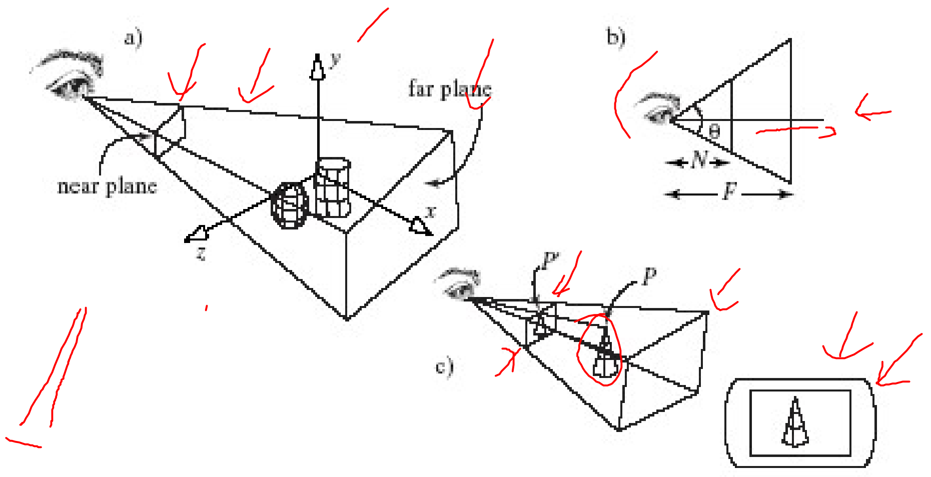


Camera
~~[illegible]~~
Eye

3

8

The Camera and Perspective Projection



9

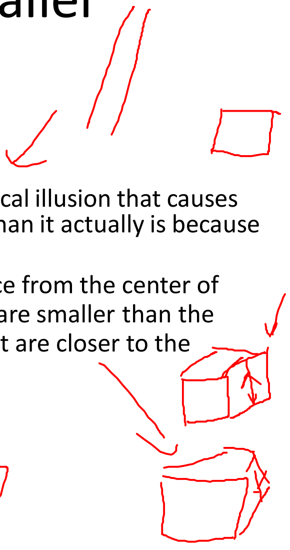
9

PROJECTIONS

- PARALLEL (parallel projectors)
  - Orthographic (projectors perpendicular to view plane)
    - Multiview (view plane parallel to principal planes)
    - Axonometric (view plane not parallel to principal planes)
      - Isometric
      - Dimetric
      - Trimetric
  - Oblique (projectors not perpendicular to view plane)
    - General
      - Cavalier
      - Cabinet
- PERSPECTIVE (converging projectors)
  - One point (one principal vanishing point)
  - Two point (Two principal vanishing point)
  - Three point (Three principal vanishing point)

10

Perspective v Parallel

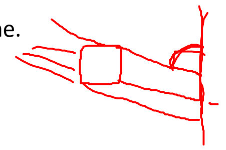


- **Perspective:**
  - visual effect is similar to human visual system...
  - has 'perspective foreshortening'
    - Foreshortening is the visual effect or optical illusion that causes an object or distance to appear shorter than it actually is because it is angled toward the viewer.
    - size of object varies inversely with distance from the center of projection. Projection of a distant object are smaller than the projection of objects of the same size that are closer to the projection plane.
- **Parallel:**
  - It preserves relative proportion of object.
  - less realistic view because of no foreshortening
  - however, parallel lines remain parallel.

11

Parallel Projection

- We can define a parallel projection with a projection vector that defines the direction for the projection lines.
- 2 types:
- Orthographic : when the projection is perpendicular to the view plane. In short,
  - direction of projection = normal to the projection plane.
  - the projection is perpendicular to the view plane.
- Oblique : when the projection is not perpendicular to the view plane. In short,
  - direction of projections ≠ normal to the projection plane.
  - Not perpendicular.



12

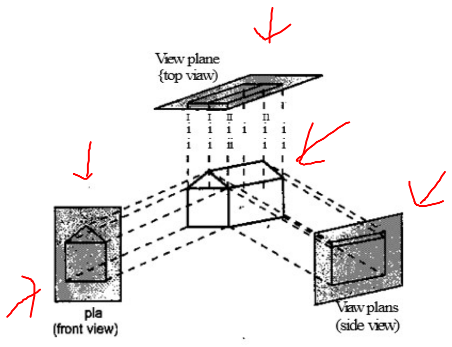
12

- Orthographic projection

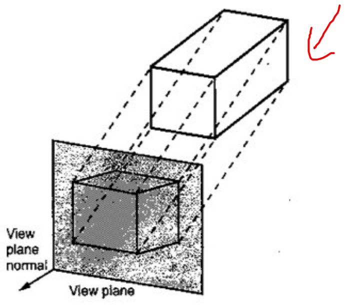
when the projection is perpendicular to the view plane

Oblique projection

when the projection is not perpendicular to the view plane



(a) Orthographic paallel pojection

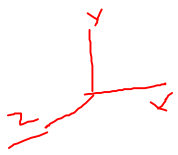


(b) Oblique parallal projection

13

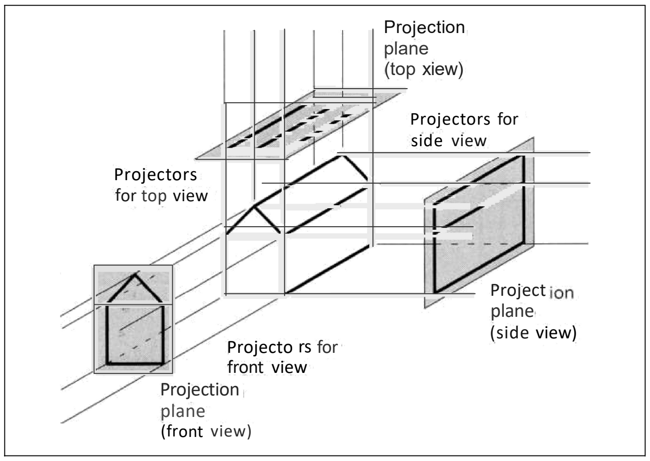
Orthographic projections

- Front, side and rear orthographic projection of an object are called elevations and the top orthographic projection is called plan view.
- all have projection plane perpendicular to a principle axes.
- Here length and angles are accurately depicted and measured from the drawing, so engineering and architectural drawings commonly employee this.



14

Orthographic projections (Multi view

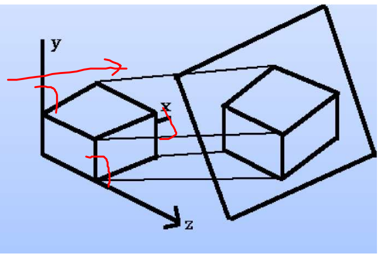

Projection plane (top xiew)
Projectors for side view
Projectors for top view
Projection plane (side view)
Projecto rs for front view
Projection plane (front view)

15

Axonometric orthographic projections

- Orthographic projections that *show more than one face of an object* are called axonometric orthographic projections.

- The most common axonometric projection is an **isometric** projection where the projection plane intersects each coordinate axis in the model coordinate system at an equal distance.



16

Axonometric orthographic projections

2

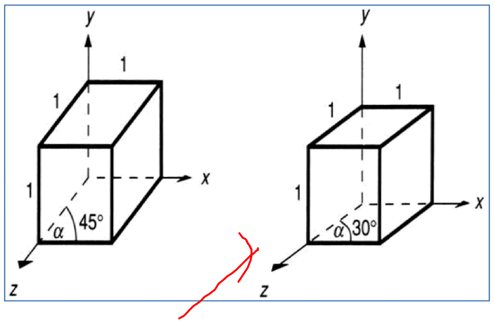
- **Dimetric projection** is defined as a way of drawing an object so that one axis has a different scale than the other two axis in the drawing.
- In **trimetric projection** the **projection** of the three angles between the axes are unequal.

$\alpha, \beta, \cancel{\gamma}\alpha = \beta = \gamma$

17

Types of Oblique Parallel Projections

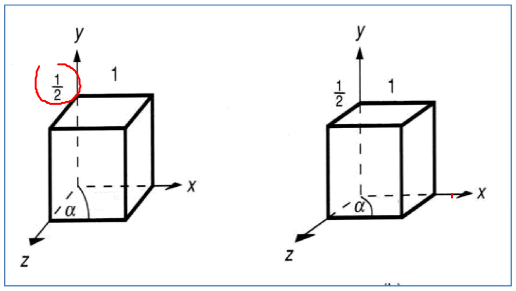
- Cavalier projection
  - All lines perpendicular to the projection plane are projected with no change in length.



18

Types of Oblique Parallel Projections

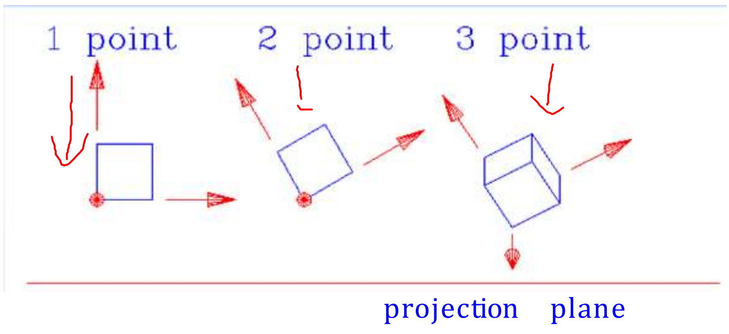
- Cabinet projection:
  - Lines which are perpendicular to the projection plane (viewing surface) are projected at 1 / 2 the length .
  - This results in foreshortening of the z axis, and provides a more “realistic” view.



19

Types of Perspective Projection

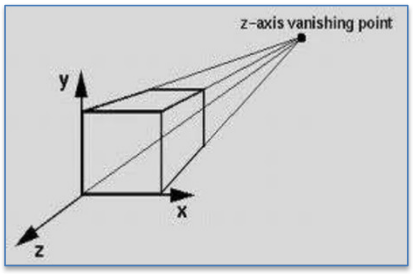
- One-Point Perspective
- Two-Point Perspective
- Three-Point Perspective



20

20

One-Point Perspective

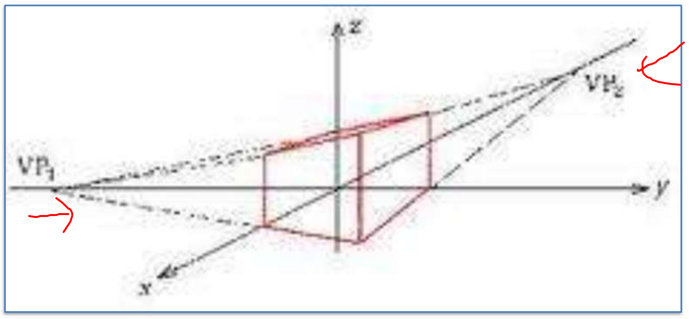


21

21

Two-Point perspective projection

- This is often used in architectural, engineering and industrial design drawings.

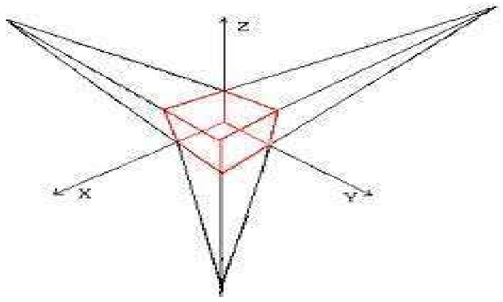


22

22

Three-Point perspective projection

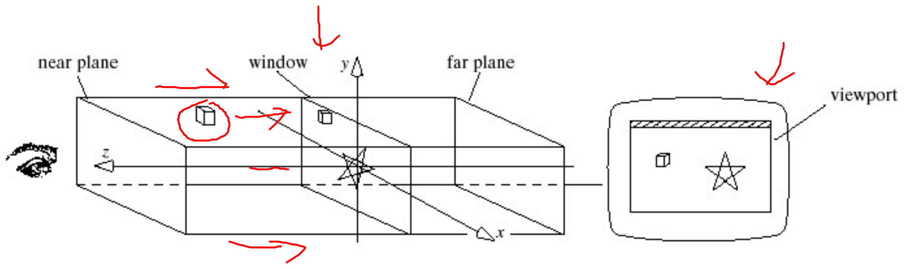
- Three-point perspective projection is used less frequently as it adds little extra realism to that offered by two-point perspective projection



23

The Viewing Process and the Graphics Pipeline

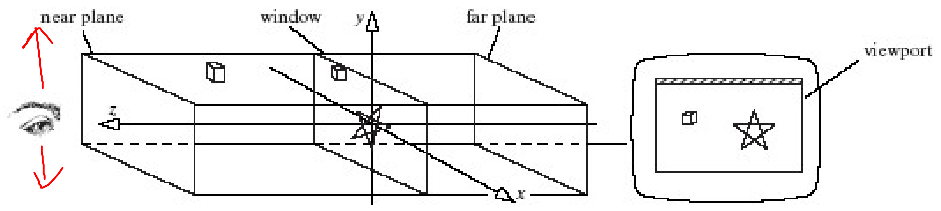
- The 2D drawing so far is a special case of 3D viewing, based on a simple parallel projection.
- The eye is looking along the z-axis at the world window, a rectangle in the *xy*-plane.



24

The Viewing Process and the Graphics Pipeline (2)

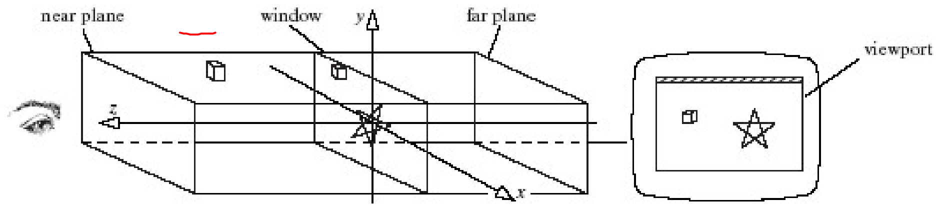
- *Eye* is simply a point in 3D space.
- The "orientation" of the eye ensures that the view volume is in front of the eye.
- Objects closer than *near* or farther than *far* are too blurred to see.



25

The Viewing Process and the Graphics Pipeline (3)

- The **view volume** of the camera is a rectangular parallelepiped.
- Its side walls are fixed by the window edges; its other two walls are fixed by a **near plane** and a **far plane**.



26

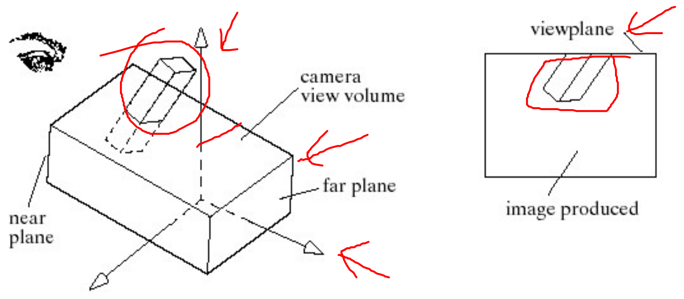
The Viewing Process and the Graphics Pipeline (4)

- Points inside the view volume are projected onto the window along lines parallel to the z-axis.
- We ignore their z-component, so that the 3D point (x_1 y_1, z_1) projects to (x_1, y_1, 0).
- Points lying outside the view volume are clipped off.
- A separate **viewport transformation** maps the projected points from the window to the viewport on the display device.

27

The Viewing Process and the Graphics Pipeline (5)

- In 3D, the only change we make is to allow the camera (eye) to have a more general position and orientation in the scene in order to produce better views of the scene.



28

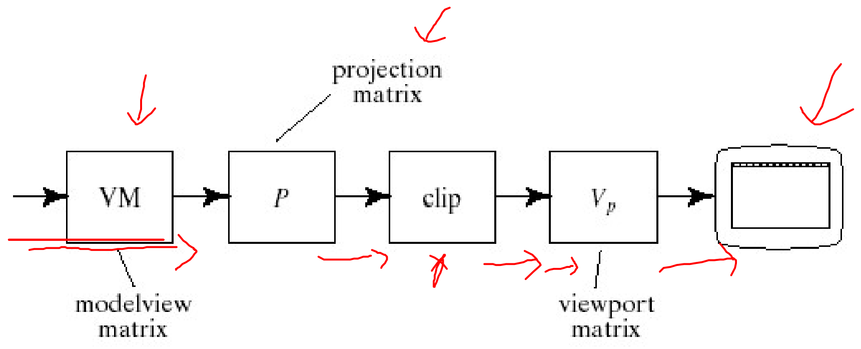
The Viewing Process and the Graphics Pipeline (6)

- The z axis points *toward* the eye. X and y point to the viewer's right and up, respectively.
- Everything outside the view volume is clipped.
- Everything inside it is projected along lines parallel to the axes onto the window plane (parallel projection).

29

The Viewing Process and the Graphics Pipeline (7)

- OpenGL provides functions for defining the view volume and its position in the scene, using matrices in the graphics pipeline.



30

The Viewing Process and the Graphics Pipeline (8)

- Each vertex of an object is passed through this pipeline using glVertex3d(x, y, z).
- The vertex is multiplied by the various matrices, clipped if necessary, and if it survives, it is mapped onto the viewport.
- Each vertex encounters three matrices:
  - The **modelview matrix;** ←
  - The **projection matrix;** ←
  - The **viewport matrix;** ←

31

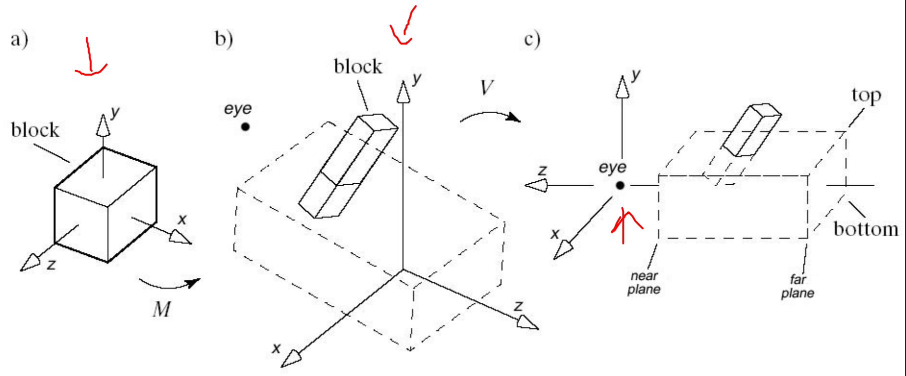
The Modelview Matrix

- The **modelview matrix** is the *CT* (current transformation).
- It combines modeling transformations on objects and the transformation that orients and positions the camera in space (hence *modelview*).
- It is a single matrix in the actual pipeline.
  - For ease of use, we will think of it as the product of two matrices: a modeling matrix *M*, and a viewing matrix *V*. The modeling matrix is applied first, and then the viewing matrix, so the modelview matrix is in fact the product *VM.*

32

The Modelview Matrix (M)

- A modeling transformation *M* scales, rotates, and translates the cube into the block.



33

The Modelview Matrix (V)

- The *V* matrix rotates and translates the block into a new position.
- The camera moves from its position in the scene to its generic position (eye at the origin and the view volume aligned with the *z*-axis).
- The coordinates of the block's vertices are changed so that projecting them onto a plane (e.g., the near plane) displays the projected image properly.

34

The Modelview Matrix (V)

- The matrix *V* changes the coordinates of the scene vertices into the **camera's coordinate system,** or into **eye coordinates**.
- To inform OpenGL that we wish it to operate on the modelview matrix we call glMatrixMode(GL_MODELVIEW);

35

The Projection Matrix

- The **projection matrix** scales and translates each vertex so that those inside the view volume will be inside a *standard cube* that extends from -1 to 1 in each dimension (Normalized Device Coordinates). -0.5, 0.5
- This cube is a particularly efficient boundary against which to clip objects.
- The image is distorted, but the viewport transformation will remove the distortion.
- The projection matrix also reverses the sense of the z-axis; increasing values of *z* now represent increasing values of depth from the eye.

36

The Projection Matrix (2)

- Setting the Projection Matrix:
  - glMatrixMode(GL_PROJECTION);
  - glLoadIdentity (); // initialize projection matrix
  - glOrtho (left, right, bottom, top, near, far); // sets the view volume parellelpiped. (All arguments are glDouble ≥ 0.0.)
- left ≤ vv.x ≤ right, bottom ≤ vv.y ≤ top, and -near ≤ vv.z ≤ -far (camera at the origin looking along -z).

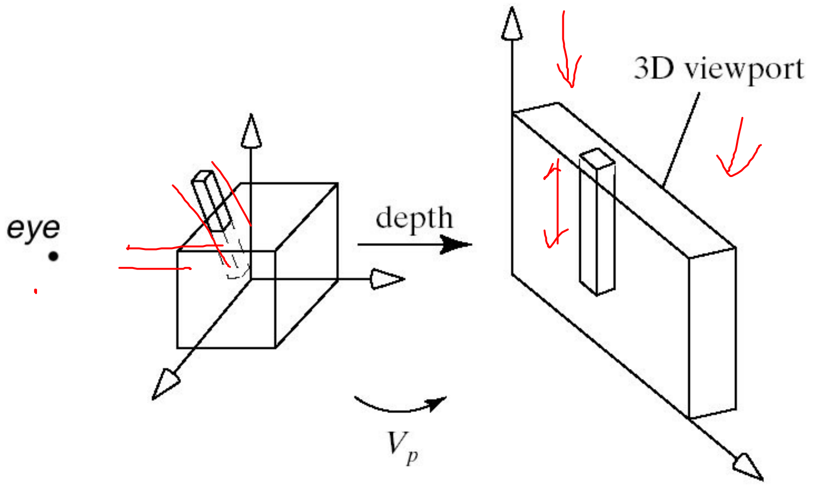
37

The Viewport Matrix

- The **viewport matrix** maps the standard cube into a 3D viewport whose *x* and *y* values extend across the viewport (in screen coordinates), and whose *z*-component extends from 0 to 1 (a measure of the depth of each point).
- This measure of depth makes hidden surface removal (do not draw surfaces hidden by objects closer to the eye) particularly efficient.

38

The Viewport Matrix (2)



39

Setting Up the Camera

- We shall use a **jib camera**.
- The photographer rides at the top of the tripod.
- The camera moves through the scene bobbing up and down to get the desired shots.

40

Setting Up the Scene (2)

```
glMatrixMode (GL_MODELVIEW);
   // set up the modelview matrix
glLoadIdentity ();
   // initialize modelview matrix
   // set up the view part of the matrix
   // do any modeling transformations on the
   scene
```

41

Setting Up the Projection

```
glMatrixMode(GL_PROJECTION);
   // make the projection matrix current
glLoadIdentity();
   // set it to the identity matrix
glOrtho(left, right, bottom, top, near, far);
   // multiply it by the new matrix
```

- Using 2 for *near* places the near plane at $z = -2$, that is, 2 units in front of the eye.
- Using 20 for *far* places the far plane at -20, 20 units in front of the eye.

42

Setting Up the Camera (View Matrix)

```
glMatrixMode (GL_MODELVIEW);
   // make the modelview matrix current
glLoadIdentity();
   // start with identity matrix
   // position and aim the camera
gluLookAt (eye.x, eye.y, eye.z,    // eye position
   look.x, look.y, look.z,       // the "look at" point
    0, 1, 0)   // approximation to true up direction
   // Now do the modeling transformations
```

43

View Matrix

- gluLookAt creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an UP vector.
- The matrix maps the reference point to the negative z axis and the eye point to the origin.
- The direction described by the UP vector projected onto the viewing plane.
- The UP vector must not be parallel to the line of sight from the eye point to the reference point.

44

Setting Up the Camera (2)

- What gluLookAt does is create a camera coordinate system of three mutually orthogonal unit vectors: **u**, **v**, and **n.**
- **n** = eye - look; **u** = **up** x **n**; **v** = **n** x **u**
- Normalize **n**, **u**, **v** (in the camera system) and let **e** = eye - $\mathcal{O}$ in the camera system, where $\mathcal{O}$ is the origin.

45

Setting Up the Camera (3)

- Then gluLookAt () sets up the view matrix

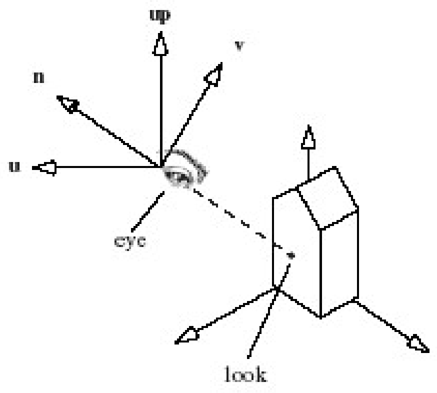
$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where **d** = (-**e**·**u**, -**e**·**v**, -**e**·**n**)
- **up** is usually (0, 1, 0) (along the y-axis), *look* is frequently the middle of the window, and *eye* frequently looks down on the scene.

46

The gluLookAt Coordinate System

- Camera in world coordinates:



47

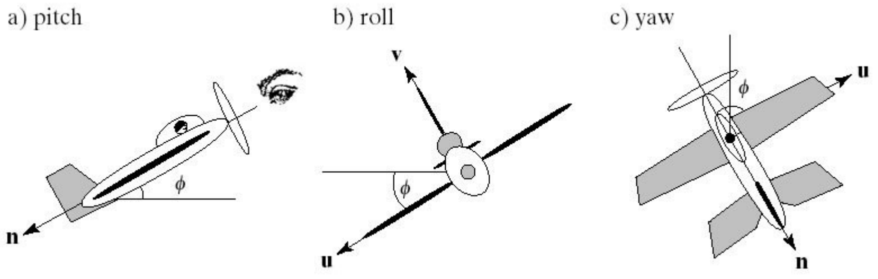
Example

```
glMatrixMode (GL_PROJECTION);
   // set the view volume (world coordinates)
glLoadIdentity();
glOrtho (-3.2, 3.2, -2.4, 2.4, 1, 50);
glMatrixMode (GL_MODELVIEW);
   // place and aim the camera
glLoadIdentity ();
gluLookAt (4, 4, 4, 0, 1, 0, 0, 1, 0);
   // modeling transformations go here
```

48

Changing Camera Orientation

- We can think of the jib camera as behaving like an airplane.
  - It can pitch, roll, or yaw from its position.



49

Changing Camera Orientation (2)

- **Pitch** – the angle between the longitudinal axis and world horizontal.
- **Roll –** the angle between the transverse axis and the world.
- **Yaw –** motion of the longitudinal axis causing a change in the direction of the plane’s flight.

50

Drawing 3D Shapes in OpenGL

- GLUT provides several 3D objects: a sphere, a cone, a torus, the five Platonic solids, and the teapot.
- Each is available as a **wireframe** model (one appearing as a collection of wires connected end to end) and as a solid model with faces that can be shaded.
- All are drawn by default centered at the origin.
- To use the solid version, replace Wire by Solid in the functions.

51

Drawing 3D Shapes in OpenGL (2)

- **cube:** glutWireCube (GLdouble size);
  - Each side is of length size.
- **sphere:** glutWireSphere (GLdouble radius, GLint nSlices, GLint nStacks);
  - nSlices is the number of “orange sections” and nStacks is the number of disks.
  - Alternately, nSlices boundaries are longitude lines and nStacks boundaries are latitude lines.

52

Drawing 3D Shapes in OpenGL (3)

- **torus:** glutWireTorus (GLdouble inRad, GLdouble outRad, GLint nSlices, GLint nStacks);
- **teapot:** glutWireTeapot (GLdouble size);
  - Why teapots? A standard graphics challenge for a long time was both making a teapot look realistic and drawing it quickly.

53

Drawing 3D Shapes in OpenGL (4)

- **tetrahedron:** glutWireTetrahedron ();
- **octahedron:** glutWireOctahedron ();
- **dodecahedron:** glutWireDodecahedron ();
- **icosahedron:** glutWireIcosahedron ();
- **cone:** glutWireCone (GLdouble baseRad, GLdouble height, GLint nSlices, GLint nStacks);

54

Drawing 3D Shapes in OpenGL (5)

- **tapered cylinder:** gluCylinder (GLUquadricObj * qobj, GLdouble baseRad, GLdouble topRad, GLdouble height, GLint nSlices, GLint nStacks);
- The **tapered cylinder** is actually a *family* of shapes, distinguished by the value of topRad.
  - When topRad is 1, there is no taper; this is the classic **cylinder**.
  - When topRad is 0, the tapered cylinder is identical to the **cone**.

55

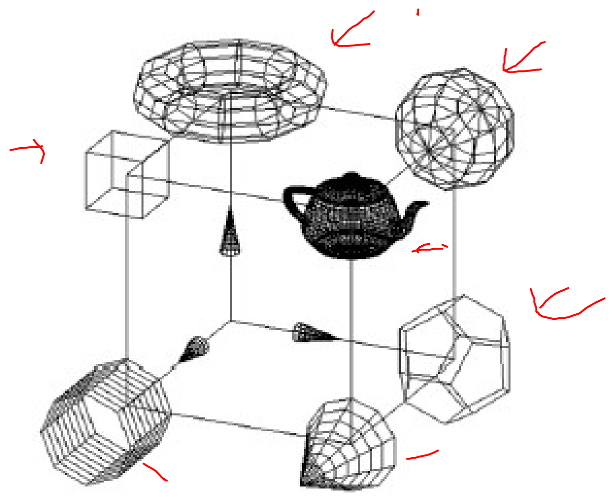
Drawing 3D Shapes in OpenGL (6)

- To draw the tapered cylinder in OpenGL, you must 1) define a new quadric object, 2) set the drawing style (GLU_LINE: wireframe, GLU_FILL: solid), and 3) draw the object:

```
GLUquadricObj * qobj = gluNewQuadric ();
   // make a quadric object
gluQuadricDrawStyle (qobj,GLU_LINE);
   // set style to wireframe
gluCylinder (qobj, baseRad, topRad, nSlices, nStacks);
   // draw the cylinder
```

56

Example



57

Code for Example

- The main() routine initializes a 640 by 480 pixel screen window, sets the viewport and background color, and specifies the drawing function as displayWire().
- In displayWire() the camera shape and position are established and each object is drawn using its own modeling matrix.
- Before each modeling transformation, a glPushMatrix() is used to remember the current transformation, and after the object has been drawn, this prior current transformation is restored with a glPopMatrix().

58

Code for Example (2)

- Thus the code to draw each object is imbedded in a glPushMatrix(), glPopMatrix() pair.
- To draw the *x*-axis, the *z*-axis is rotated 90^o about the *y*-axis to form a rotated system, and the axis is redrawn in its new orientation.
- This axis is drawn without immersing it in a glPushMatrix(), glPopMatrix() pair, so the rotation to produce the *y*-axis takes place in the already rotated coordinate system.

59