UNIT 6

VIEWING

Syllabus.

* classical and computer viewing
* viewing with a computer.
* position of the camera
* simple projections.
* projections in openGL
* Hidden surface Removal
* interactive mesh displays
* parallel projection matrices.
* perspective projection matrices.
* projections and shadows.

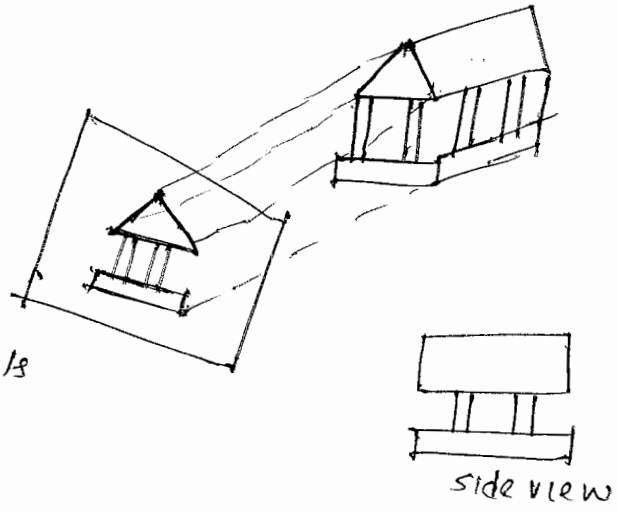
- 7 Hours.

CLASSICAL AND COMPUTER VIEWING

* There are four types of classical views (projections)
  1. orthographic projections
  2. Axonometric projections.
  3. oblique projections
  4. perspective projections.

orthographic projection.

* In all orthographic views, the projectors are perpendicular to the projection plane.
* Fig shows orthographic projection.
* Usually three views are used in an orthographic projection to display the objects
  - front view
  - Top view
  - side view



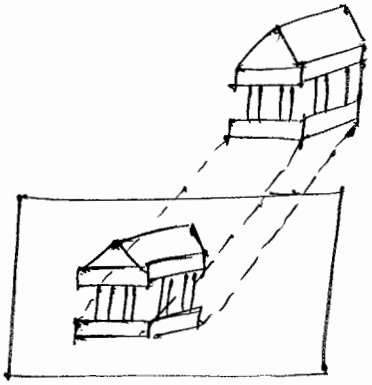
side view

* Adv: It preserves both distances and angles. Since there is no distortion, it is well suited for working drawings.

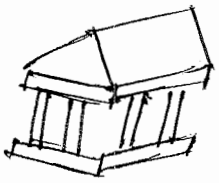
Axonometric projections.

* It is such a projection in which the projector is still orthogonal to the projection plane, but the projection plane can have any orientation wrt the object. using this view, more than one principle faces of the object would be visible.

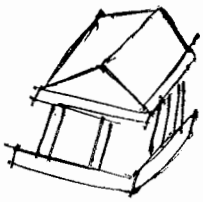
* If the projection plane is placed symmetrically w.r.t the three principal faces that meet at a corner of our rectangular object, then we have an isometric view
* If the projection plane is placed symmetrically wrt two of the principal faces, then the view is dimetric.
* The general case is a trimetric view.



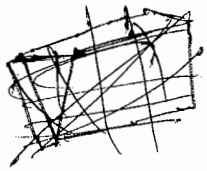
Axonometric projection.

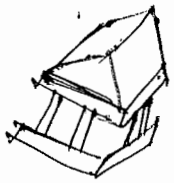


Dimetric



Trimetric.

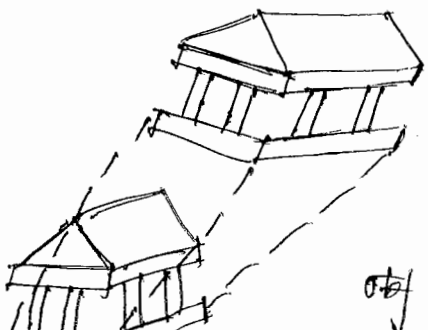




Isometric.

Oblique projections

* It is one of the most general parallel projections.
* This projection is obtained by allowing the projectors to make an arbitrary angle with the projection plane.
* Angles of the objects face that are parallel to the projection plane are preserved.
* oblique views (projection) are most difficult to construct by hand. They are somewhat unnatural.

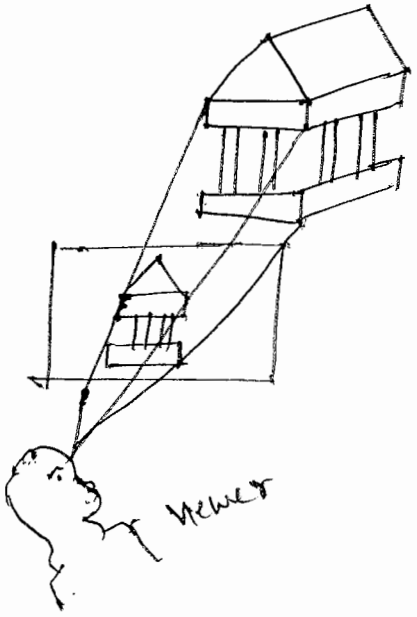


Oblique view.

perspective viewing.

* It is such a projection in which the viewer is located symmetrically wrt the projection plane.
All perspective projectors are characterized by diminution of size.
* when objects are moved farther from the viewer, their images become smaller.
This size change gives perspective projection its natural appearance. Hence it is widely used in architecture and animation

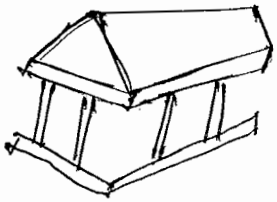
* figure shows perspective projections.



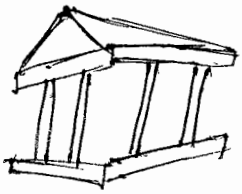
* there are three types of perspective views
- one point perspective
- two point perspective
- three point perspective.

based on how many no. of the three principal directions in the object are parallel to the projection plane.

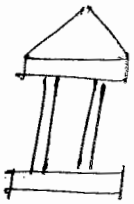
* In most general case, 3-point perspective, the parallel lines in each of the three principal directions converge at one point called - vanishing points. < fig (a) >
* in 2-point perspective, lines in only two of the principal directions converge at vanishing points. < fig(b) >
* In 1-point perspective, two of the principal directions converge at a single vanishing point.



3 point
fig (a)

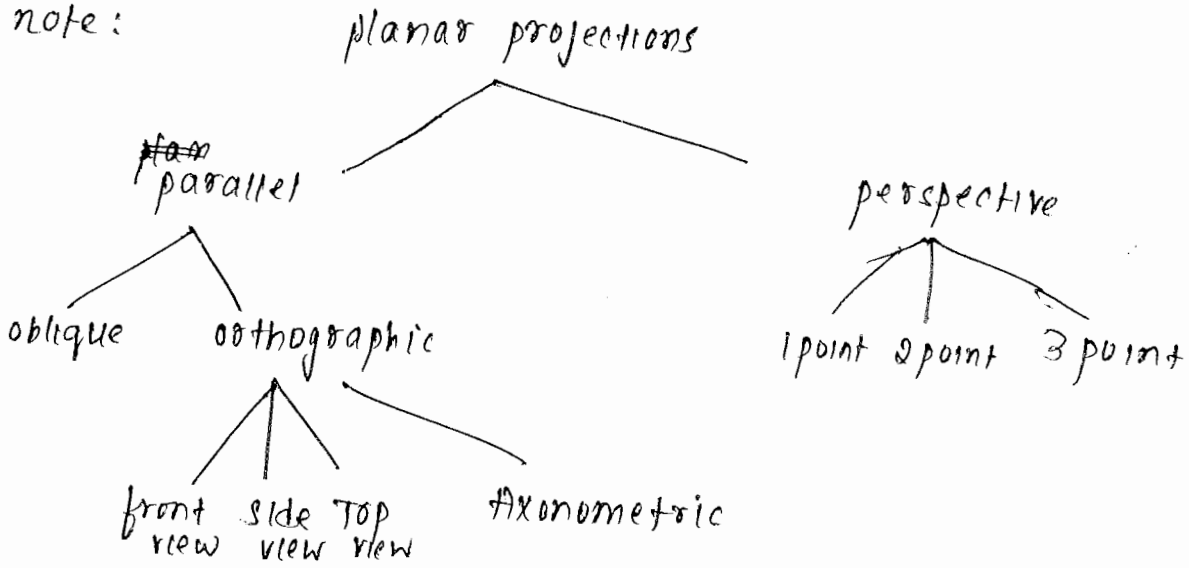


2 point
fig (b)



1 point
fig (c).

note:



VIEWING WITH A COMPUTER.

* Viewing using computers involve two fundamental operations

1. positioning and orienting the camera which can be achieved by using the model view matrix.
2. Application of the projection transformation. ie parallel projection or perspective projection which can be achieved using projection matrix.

unit 4

units

POSITIONING OF THE CAMERA

* camera can be positioned using openGL by modifying the <u>model-view matrix</u>.
* initially the model view matrix is an identity matrix and hence the camera frame and the object frame would be identical as shown below. fig(a).

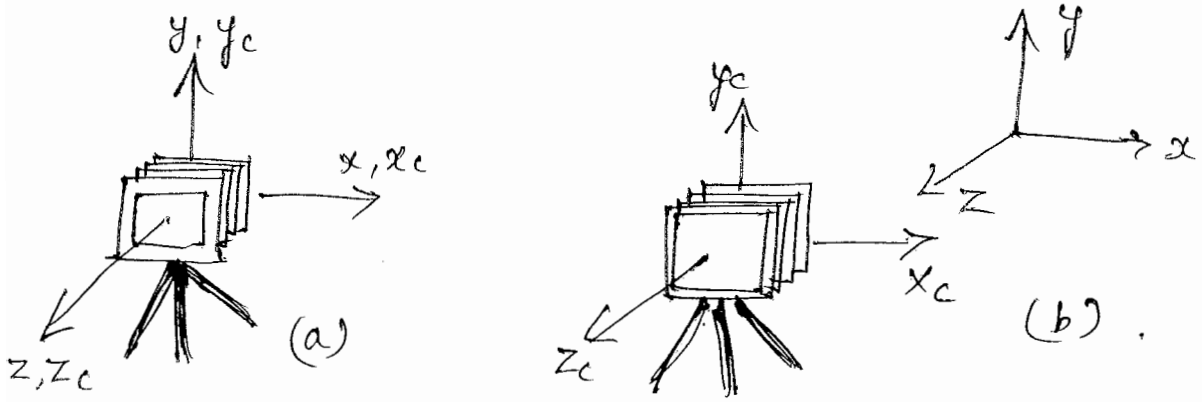


fig: movement of camera and object frames.
(a) initial configuration.
(b) configuration after change in model-view. matrix

* with fig (a) arrangement, all the objects present in the scene may not be visible and hence we will have to change the model-view matrix suitably to position the camera at the desired location. (refer fig b)

eg1: Suppose that the user is interested in viewing at the object from the -ve z axis. Then the camera can be positioned using,

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslate (0.0, 0.0, -d);
```

eg2: Suppose that user is interested in looking at the same object from the x axis. Then the camera can be positioned using,

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslatef (0.0, 0.0, -d);
glRotate (-90.0, 0.0, 1.0, 0.0);
```

eg3: To obtain an isometric view of the cube which is centered at the origin and aligned with the axes, we must place the camera anywhere along the line from the origin through the point (1,1,1). This can be achieved using,

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslatef (0.0, 0.0, -d);
glRotate (45.0, 0.0, 1.0, 0.0);
glRotate (35.26, 1.0, 0.0, 0.0);
```

* There is an altogether different approach that can be used to position the camera.
This approach is used in PHIGS and GKS-3D (which were one of the earliest graphics packages).
The steps followed are:

1. The camera is assumed to be initially positioned at origin, pointing in the -ve Z direction. Its desired location is referred to as the view reference point (VRP) which can be specified as follows

```
set_view_reference_point (x, y, z);
```

2. orientation of the camera can be specified using the view-plane-normal (VPN) using,

set_view_plane_normal (nx, ny, nz);

3. The up direction from the perspective of the camera can be specified using the view-up (VUP) as shown below,

set_view_up (VUP_x, VUP_y, VUP_z);

* Another approach is to use the LookAt function as shown below,

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
gluLookAt (eyex, eyey, eyez, atx, aty, atz,
                              upx, upy, upz);
/* define objects here */
```

* Another approach is to specify the azimuth and the twist angle to position the camera.

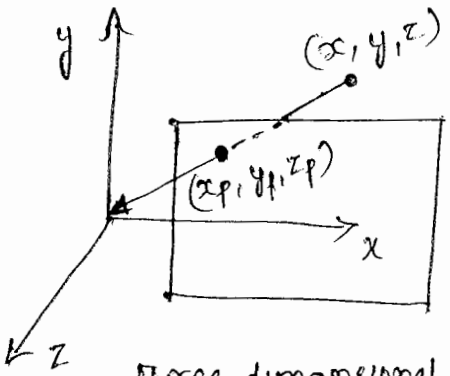
SIMPLE PROJECTIONS.

* There are two types of simple projections

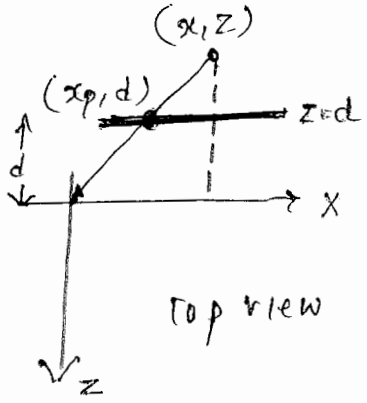
- perspective projections
- parallel (orthographic) projections.

perspective projection.

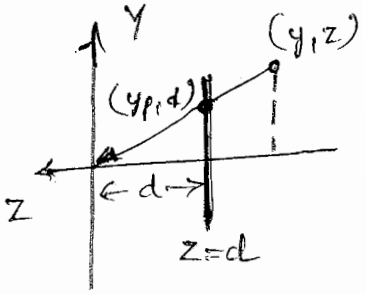
* Suppose that the camera is located at the origin pointing in the negative z-direction.
Assume that the projection plane is in front of the camera.
with the above arrangement, a point in space at the point (x, y, z) is projected along a projector into the point (x_p, y_p, z_p). All projectors pass through the origin as shown.



Three-dimensional view.



top view



side view.

* From the above, it can be noticed that

$$z_p = d$$

from top view, we see that two similar triangles whose tangents must be same

$$\frac{x}{z} = \frac{x_p}{d} \Rightarrow \boxed{x_p = \frac{x}{z/d}}$$

from side view,

$$\frac{y}{z} = \frac{y_p}{d} \Rightarrow \boxed{y_p = \frac{y}{z/d}}$$

these eqns are non linear. The division by z describes non uniform foreshortening.

* The above equations can be obtained in the matrix form as shown below.

Consider the point in space as

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Consider the matrix M as

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

* The matrix M transforms the point P to the point.

$$q = M * P.$$

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

By dividing first three terms with the fourth term, we get.

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

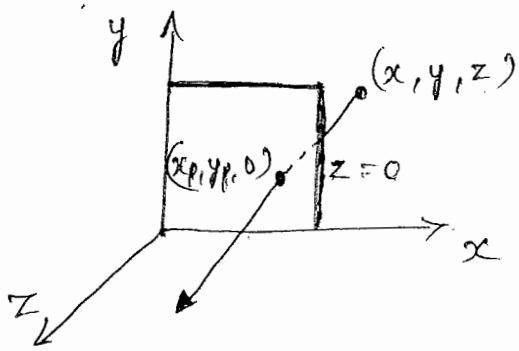
Therefore, $\boxed{x_p = \frac{x}{z/d}}$

$\boxed{y_p = \frac{y}{z/d}}$

and, $\boxed{z_p = \frac{z}{z/d} = d}$.

Orthogonal projections (parallel projections)

* Orthogonal projections are such projections in which the cameras have infinite focal length.
  It is as shown below.



* projectors are perpendicular to the view plane
* Above diagram shows a projection plane with $z=0$. As points are projected on to this plane, it can be noticed that they retain their x and y values ie diminution does not takes place.
  Therefore, in orthogonal projection,

$$\boxed{x_p = x} \qquad \boxed{y_p = y} \qquad \boxed{z_p = 0}$$

* Above equations can be obtained in matrix mode as shown below.
  Consider the point in space as

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

consider the matrix M such that

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

Then we can obtain orthogonal projection of the point p by multiplying M and p as shown below.

$$q = M * p$$

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

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

Therefore,

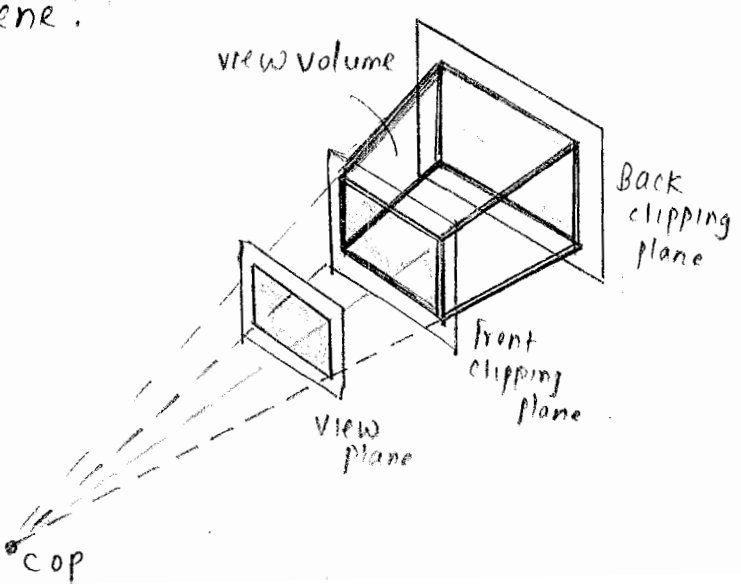
$x_p = x$ $\quad$ $y_p = y$ $\quad$ $z_p = 0$

PROJECTIONS IN OPENGL

Let's see how view volume are specified in opengl.

* view volume is also referred to as ~~frustrum~~. frustum. objects falling within the view volume are displayed where as the objects falling outside the view volume are clipped out of the scene.

* The view volume is a truncated pyramid with its apex at center of project (COP) as shown in the figure.



* openGL provides two functions for specifying perspective view volume and one function for specifying parallel view volume.

Perspective viewing in openGL

* perspective view volume can be specified using -

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
glFrustum (left, right, bottom, top, near, far);
```

* This specification creates the following view volume.

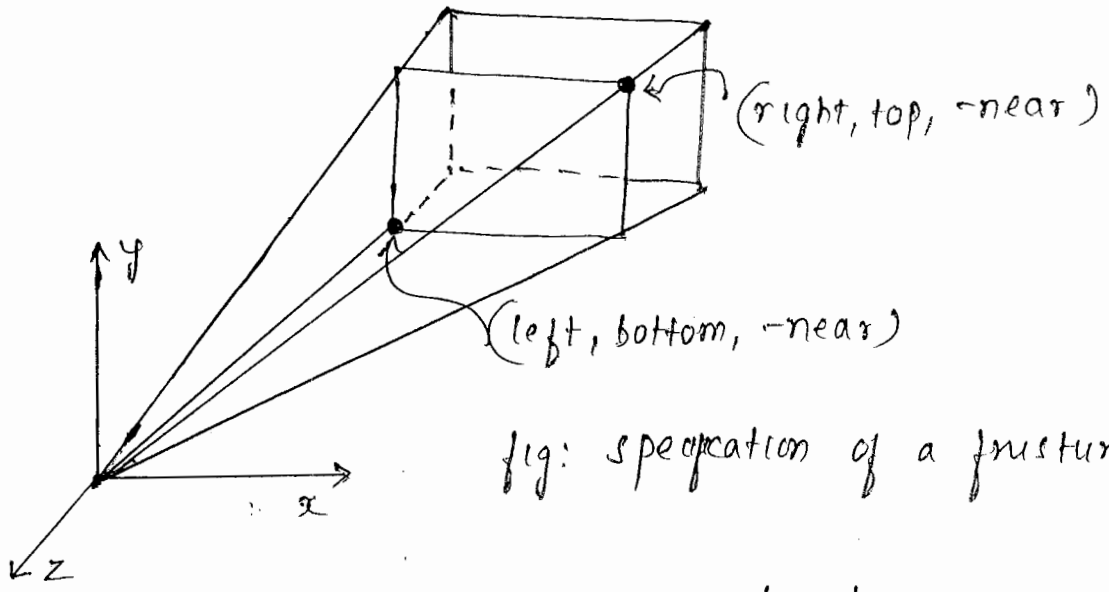


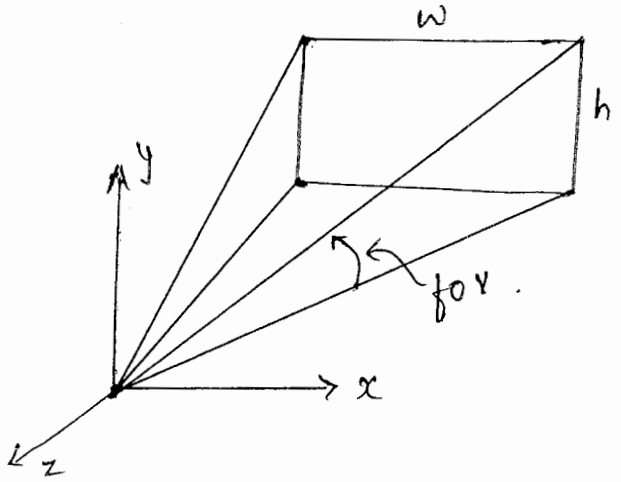
fig: specification of a frustum.

* The perspective view volume can also be specified by prescribing the angle fov, aspect ratio (ratio of width & height near & far parameters as shown below -

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluPerspective (fovy, aspect, near, far);
```

fov - field of view.

* This specification creates the following view volume.

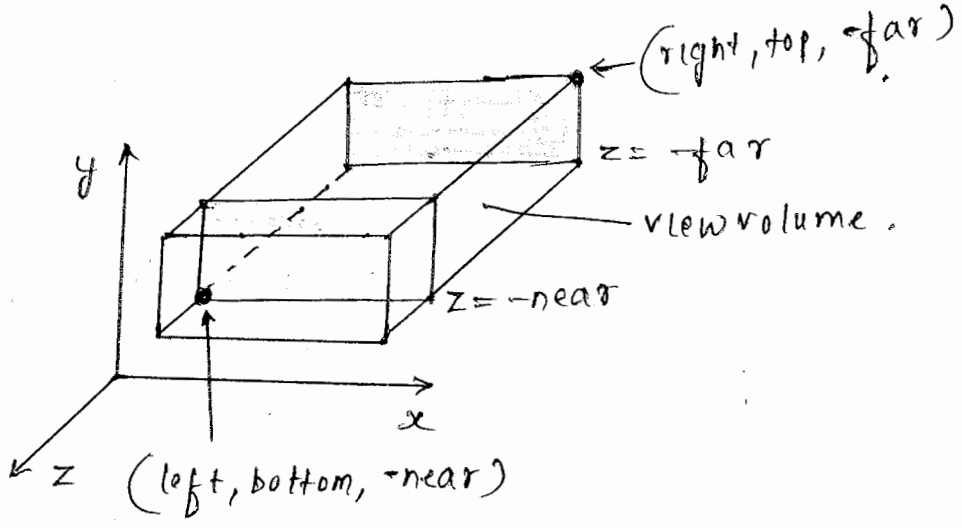


parallel (orthographic) viewing in openGL.

* orthographic viewing volume can be specified in openGL using,

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
glOrtho (left, right, Bottom, top, near, far);
```

* This specification creates the following view volume.



PARALLEL PROJECTION MATRICES

lets Derive the matrix for orthogonal and oblique projections in OpenGL.

* Basically there are two types of parallel projections namely -
  - orthogonal projection
  - oblique projection.

Orthogonal projection matrices

* orthogonal projection matrix can be obtained by performing the following steps -

step 1: creating a view volume equal to the canonical view volume which is a cube defined by the sides $x = \pm 1$, $y = \pm 1$, and $z = \pm 1$

This step can be performed as shown below -

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
glOrtho (-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

step 2:

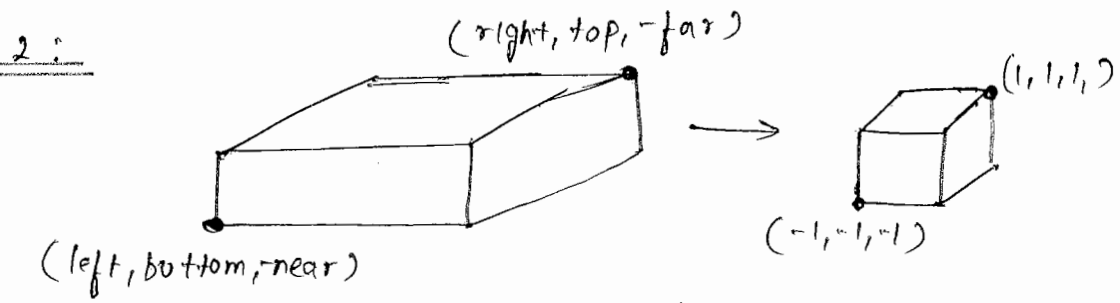


fig: mapping of view volume to the canonical view volume.

Mapping of the original view volume to the canonical view volume

This step can be performed by first translating the center of the original viewvolume to the center of the canonical view volume and then scaling the original view volume to the canonical view volume.

Hence the two transformations to be performed are:

Translation $\left(-(right + left)/2,\ -(top + bottom)/2,\ +(far + near)/2\right)$

scaling $\left(2/(right - left),\ 2/(top - bottom),\ 2/(far - near)\right)$

They are concatenated together to form the projection matrix as shown below.

$$P = ST = \begin{bmatrix} \frac{2}{right - left} & 0 & 0 & -\frac{left + right}{right - left} \\ 0 & \frac{2}{bottom - top} & 0 & -\frac{Top + Bottom}{Top - Bottom} \\ 0 & 0 & -\frac{2}{far - near} & -\frac{far + near}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

fig: Affine Transformations for normalization.

Oblique Projection Matrix

* Oblique projection Matrix can be obtained by performing the following steps.

step 1: shear of objects by $H(\theta, \phi)$

step 2: create a view volume equal to canonical view volume

step 3: Mapping the original view volume to canonical view volume by performing Translation and scaling

[step2 and step3 are same as in previous case]

* consider the following oblique clipping volume

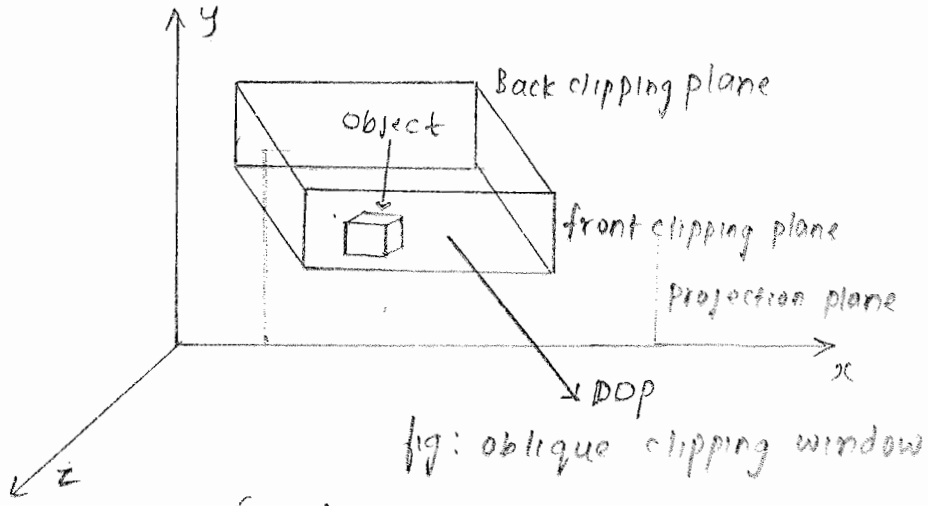


fig: oblique clipping window

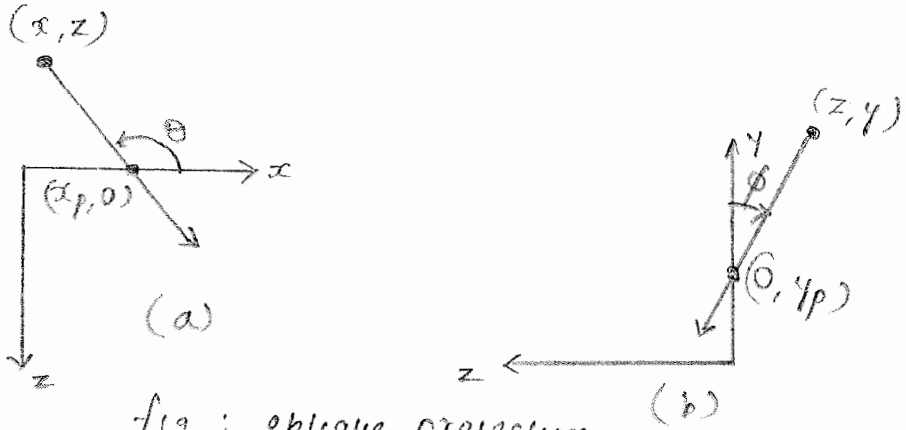


fig: oblique projection

(a) Top view

(b) side view

* from the above fig, it can be noticed that -

$$x_p = x + z\cot\theta \longrightarrow (\because \text{from top view, } \tan\theta = \frac{z}{x_p - x})$$

$$y_p = y + z\cot\phi$$

$$z_p = 0$$

* we can write these in terms of a homogeneous coordinate matrix

$$P = \begin{bmatrix} 1 & 0 & \cot\theta & 0 \\ 0 & 1 & \cot\phi & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

* This matrix can be expressed as the concatenation (product) of orthographic projection matrix (M_{ortho}) and shear matrix $H(\theta, \phi)$ as shown

$$P = M_{ortho} * H(\theta, \phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & \cot\theta & 0 \\ 0 & 1 & \cot\phi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

* However, since scaling and Translation also has to be performed to make the original view volume equal to canonical view volume, the projection matrix for oblique projection would be -

$$\boxed{P = M_{ortho} * S * T * H}$$

where $ST = \begin{bmatrix} & \\ & \end{bmatrix}$ ← refer prev. erase.

PERSPECTIVE PROJECTION MATRICES

lets Derive the matrix for perspective projection in openGL.

* perspective projection matrix can be obtained by performing following steps -

step1: Distortion (Normalization) of the object

step2: perform orthographic projection.

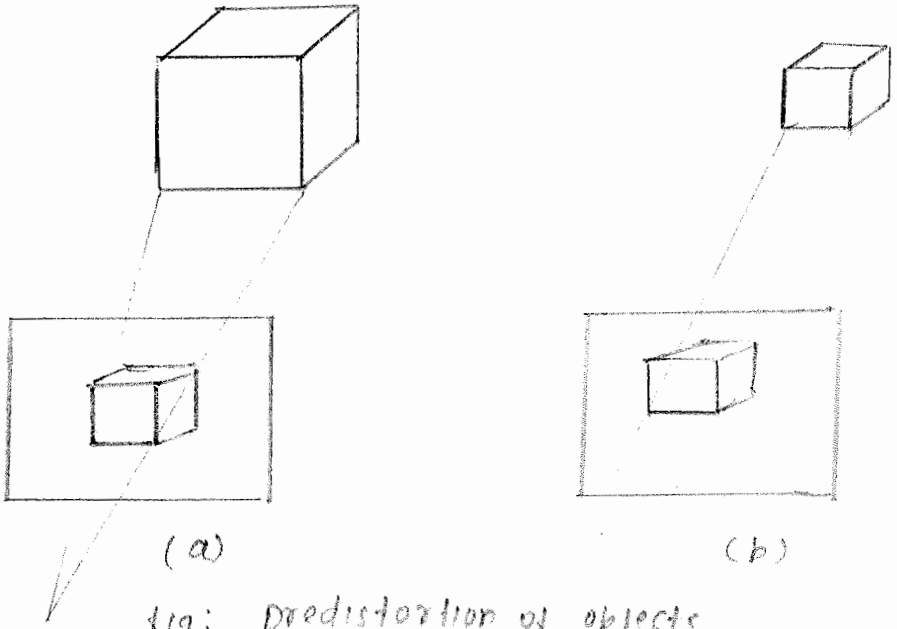


fig: predistortion of objects
(a) perspective view
(b) Orthographic projection of distorted object

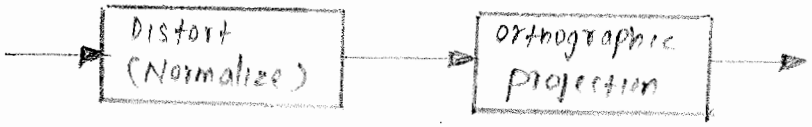


fig: Normalization transformation

* A simple projection matrix for the projection plane at $z = -1$ and the COP at the origin is -

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \rightarrow \text{simple perspective projection matrix.}$$

* consider the matrix -

$$N = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$ which is similar to M but is nonsingular.

* consider the point -

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

* By applying N on P we get

$$q = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}$$

where, $x' = x$
$y' = y$
$z' = \alpha z + \beta$
$w' = -z$

* After dividing by w', we have the 3-D point -

$$\begin{bmatrix} x'' \\ y'' \\ z'' \\ 1 \end{bmatrix} = \begin{bmatrix} -x/z \\ -y/z \\ -(\alpha + \beta/z) \\ 1 \end{bmatrix}$$

Therefore, we get

$$x_p = \frac{-x}{z} \qquad y_p = -\frac{y}{z}$$

* The same result can be obtained by applying an orthographic projection along the z-axis to N. we get

$$M_{ortho} * N = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

when this orthographic projection is applied on the point $p[x, y, z, 1]$. we get

$$p' = M_{ortho} * N * p = \begin{bmatrix} x \\ y \\ 0 \\ -z \end{bmatrix}$$

Therefore, we get $x_p = \frac{-x}{z}$ $y_p = \frac{-y}{z}$

Therefore, by applying N directly on the point yields the same result as applying the orthographic projection along z axis on N and then projecting the point

* original view volume can be normalized to perspective canonical view volume by choosing

$$x = \pm \frac{right - left}{-2 * near}$$

$$y = \pm \frac{top - bottom}{-2 * near}$$

$$z = -near$$

$$z = far.$$

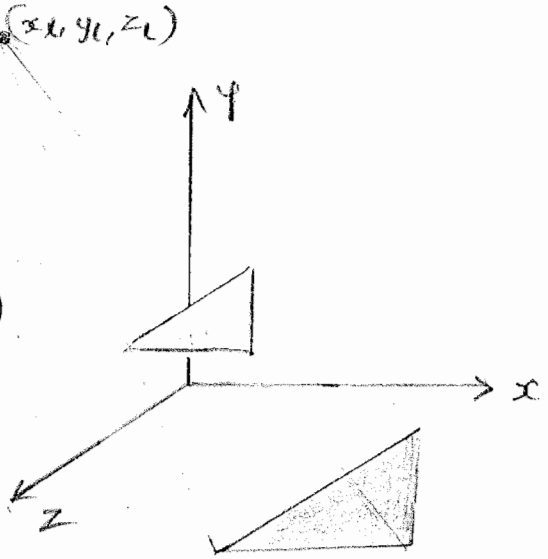
* Therefore, the resulting perspective projection matrix is $\Rightarrow P = N * S * H =$

$$\begin{bmatrix} \frac{-2*near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{-2*near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & \frac{2far*near}{far-near} \end{bmatrix}$$

PROJETION AND SHADOWS

lets see how shadows can be created and projected using openGL.

* simple shadows can be created in openGL. Although shadows are not geometric objects, yet they are important components of realistic images and give many visual clues.
* shadows require a light source to be present for simplicity, we assume that the shadow falls on ground ie $y=0$.
* A <u>shadow</u> is a flat polygon and is the projection of the original polygon onto the surface (ground). It is as shown below -



* The light source present at (x_L, y_L, z_L) must be bought the origin by performing a Translation $T(-x_L, -y_L, -z_L)$

then we have to perform a simple perspective projection through the origin.
The projection matrix is

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

Then we must translate the light source back to (x_L, y_e, z_e) by $T(x_L, y_L, z_L)$

~~Therefore we have~~ the concatenation of this matrix and the two translation matrices projects the vertex (x, y, z) to

$$x_p = x_e - \frac{x - x_e}{(y - y_e)/y_e} \qquad y_p = 0 \qquad z_p = z_e - \frac{z - z_e}{(y - y_e)/y_e}$$

note: how we got this?

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -x_L \\ 0 & 1 & 0 & -y_L \\ 0 & 0 & 1 & -z_L \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1/y_L & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & x_L \\ 0 & 1 & 0 & y_L \\ 0 & 0 & 1 & z_L \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_L - \dfrac{(x - x_L)}{(y - y_L)/y_L} \\ 0 \\ z_L - \dfrac{z - z_L}{(y - y_L)/y_L} \\ 1 \end{bmatrix}$$

* With openGL program, we can alter model-view matrix to form the desired polygon as follows.

```
GLfloat m[16];                          /* shadow projection */
for(i=0; i<m; i++)
    m[i] = 0.0;
m[0] = m[5] = m[10] = 1.0;
m[7] = -1.0/yl

glColor3fv (polygon_color);
glBegin (GL_POLYGON);
    .
    .                                   /* Draw the polygon
    .                                      normally */.
glEnd();
```

```
glMatrixMode (GL_MODELVIEW);
glPushMatrix ();                    /* save state */
glTranslatef (xl, yl, zl);          /* Translate back */
glMultMatrixf (m);                  /* project */
glTranslate (-xl, -yl, -zl);        /* move light to origin */
glColor3fv (shadow_color);
glBegin (GL_POLYGON);
        ≡                           /* Draw the polygon
        ≡                                  again */
        ≡
glEnd ();
glPopMatrix ();                     /* restore state */
```