MMTE-004
COMPUTER GRAPHICS

Indira Gandhi Ntional Open University
School of Sciences

Block

3

3D TRANSFORMATIONS AND VIEWING

Curriculum Design Committee

Dr. B.D. Acharya
Dept. of Science & Technology, New Delhi

Prof. Adimurthi
School of Mathematics
TIFR, Bangalore

Prof. Archana Aggarwal
CESP, School of Social Sciences
JNU, New Delhi

Prof. R.B. Bapat
Indian Statistical Institute
New Delhi

Prof. M.C. Bhandari
Dept. of Mathematics
IIT, Kanpur

Prof R. Bhatia
Indian Statistical Institute, New Delhi

Prof. A.D. Dharmadhikari
Dept. of Statistics
University of Pune

Prof. O.P. Gupta
Dept. of Financial Studies
University of Delhi

Prof. S.D. Joshi
Dept. of Electrical Engineering
IIT Delhi

Dr. R.K. Khanna
Scientific Analysis Group, DRDO, Delhi

Prof. Susheel Kumar
Dept. of Management Studies
IIT, Delhi

Prof. Veni Madhavan
Scientific Analysis Group
DRDO, Delhi

Prof. J.C. Misra
Dept. of Mathematics
IIT, Kharagpur

Prof. C. Musili
Dept. of Mathematics and Statistics
University of Hyderabad

Prof. Sankar Pal
ISI, Kolkata

Prof. A.P. Singh
PG Dept. of Mathematics
University of Jammu

Faculty Members
School of Sciences, IGNOU
Prof. Poornima Mital
Dr. Atul Razdan
Prof. Parvin Sinclair
Prof. Sujatha Varma
Dr. S. Venkataraman

Course Design Committee

Prof. Sujatha Varma
Director (SOS), IGNOU

Prof. Chandan Singh(Retd.)
Department of Computer Science,
Punjabi University, Patiala

Prof. S. Balasundaram (Retd.)
School of Computer and Systems Sciences,
JNU, New Delhi

Faculty Members
School of Sciences, IGNOU

Prof. Parvin Sinclair
Dr. S. Venkataraman
Dr. Deepika
Dr. Pawan Kumar

*The syllabus was designed and unitised as per the decision of the Course Expert Committee which met on 17th March, 2021.

Block Preparation Team

Prof. B.S. Panda (Editor)
Dept. of Mathematics
IIT Delhi

Dr. Pawan Kumar
School of Sciences
IGNOU

Course Coordinator: Dr. Pawan Kumar

Acknowledgements: To Mr.Surender Singh Chauhan for typesetting the manuscript and preparing the CRC.

……., 2023

ISBN-8]-

BLOCK INTRODUCTION

This is the third and the last block of the course Computer Graphics, which also has 2 units. Here our focus is on transforming and viewing three-dimensional objects.

Unit 6 presents transformations in three dimensions, some of which are simple generalisations of their 2D counterparts, such as translations and scaling. However, as you will see, 3D rotations are more complex as they offer greater possibility for rotating an object. We start with 3D rotations about coordinate axes and use them along with necessary translations to obtain general three-dimensional rotations. We lastly, look at the three-dimensional viewing transformations.

Unit 7 mainly talks about the concept of projection. It is quite useful in viewing and modelling three-dimensional objects. We discuss two types of projections namely, parallel and perspective projections.

NOTATIONS AND SYMBOLS

$T(a, b)$	2D translation by (a, b)
$R(\theta)$	2D rotation about the origin by θ
$S(\alpha, \beta)$	2D scaling by α, β
$R(a, b; \theta)$	2D rotation by θ about pivot (a, b)
M_x	reflection about x-axis
M_y	reflection about y-axis
$M_{x=y}$	reflection about the line $y = x$
$M_{y=-x}$	reflection about the line $y = -x$
$S_x^{shear}(\alpha)$	x-direction shear by α
$S_y^{shear}(\beta)$	y-direction shear by β
$S^{shear}(\alpha, \beta)$	simultaneous shear by α, β
$T(a, b, c)$	3D translation by (a,b, c)
$R_x(\theta)$	3D rotation matrix about x-axis by θ
$R_y(\theta)$	3D rotation matrix about y-axis by θ
$R_z(\theta)$	3D rotation matrix about z-axis by θ
$S(\alpha, \beta, \gamma)$	3D scaling by α, β, γ

UNIT 6

3D TRANSFORMATIONS

Structure **Page No.**

6.1 INTRODUCTION

For displaying a 3D scene on a 2D display, one requires a sequence of transformations. First you need to fix a position from where you would like to view the scene. This is called camera or eye position. From that particular orientation, you can then plan what to display and where to display. To model the view angle and orientation, you need to apply a sequence of such 3D primitive geometric transformations.

In Section 6.2, we shall first see 3D primitive geometric transformations as natural extensions of their 2D counterparts. Then we shall look at various compositions of such transformations. Section 6.3 takes you to the general three-dimensional rotations, where we also discuss the Open GL routnines available for performing such transformations.

Objectives

After reading this unit you should be able to

- explain basic 3D transformations–translation, rotation, scaling, shear and reflections–applied to objects in space;
- explain how to transform between two coordinate systems;
- explain how to apply basic geometric transformations on objects;

- build model view matrix capturing the scene from camera position.

6.2 3D PRIMITIVE AND COMPOSITE TRANSFORMATIONS

We start with the elementary transformations namely translation, rotation, and scaling. Translation of a point $P(x, y, z)$ by a point (a, b, c) is obtained by the equations $x' = x + a,\ y' = y + b,\ z' = z + c.$ To present the transformation in matrix form, we shall use homogeneous coordinates. Any point P with coordinates (x, y, z) would be represented in homogeneous coordinates as

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

The translation of P to a point P' by the point $(\mathrm{a}, \mathrm{b}, \mathrm{c})$ is done as follows:

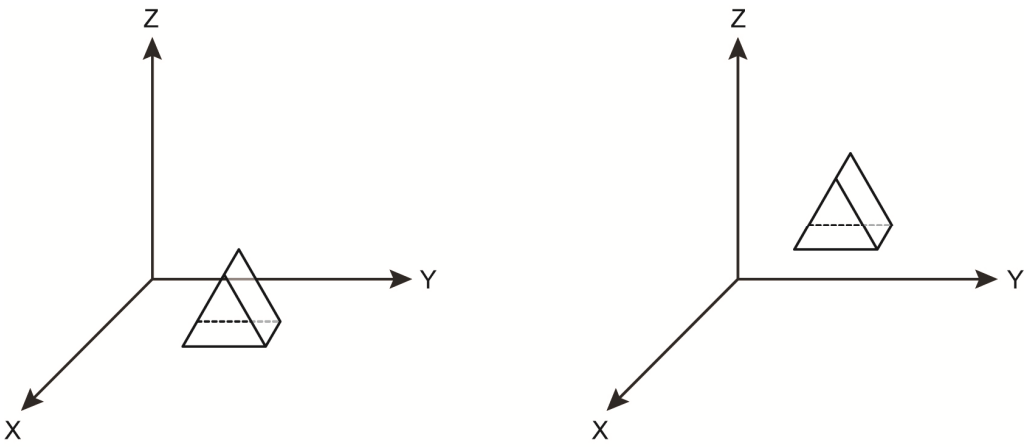
$P' = T(a, b, c) \cdot P,$

where $T(a, b, c) = \begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix}$ and $P' = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}.$

For instance, the point $P(3, 4, 2)$ when translated by $(1, 3, -5)$ becomes the point $P'(x', y', z')$, where

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 3 \\ 4 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 7 \\ -3 \\ 1 \end{pmatrix}.$$

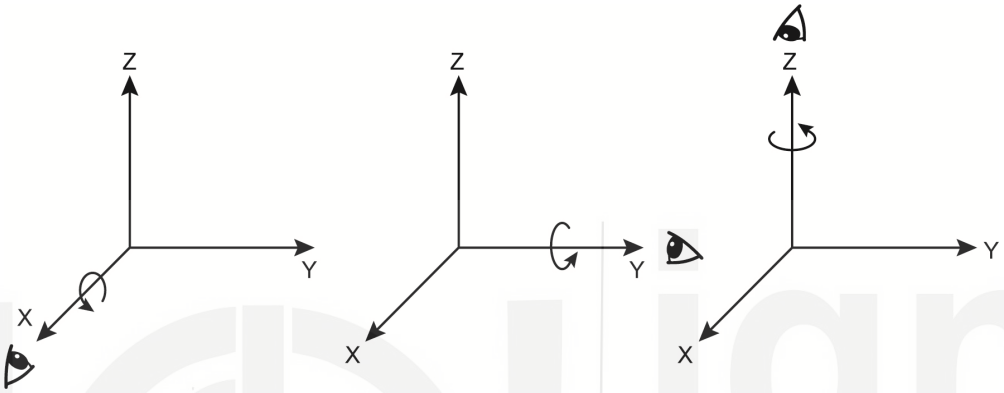
The following picture illustrates the translation of a prism.



You can easily verify that $T(a, b, c) \cdot T(-a, -b, -c)$ is a 4×4 identity matrix.

That is, $T(-a, -b, -c)$ is the inverse translation of $T(a, b, c)$.

To rotate an object in 3 dimensions, we need an "angle" through which the object is rotated, and an "axis" about which the object is rotated. Thus three-dimensional rotations offer much variety. The easiest rotations are about the coordinate axes. Suppose we wish to rotate an object about a coordinate axis. Then, by convention, the direction of rotation is taken as counter-clockwise when we view from the positive side of that axis towards the origin. This is illustrated in the following pictures.



Note that the rotation of a point $P(x, y, z)$ **about the** z **-axis** through an angle θ is the same as the two-dimensional rotation of the point (x, y) through θ in the counter clockwise direction, while keeping the z -coordinates fixed. Thus, we obtain

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta \qquad (1)$$
$$z' = z.$$

These can be written in matrix form as

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}.$$

or compactly as $P' = R_z(\theta)P,$ where

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

is called the **rotation matrix** about z -axis through θ.

Now if we replace x with y, y with z and z with x in Eqn. (1), we obtain **the rotation transformations about the** x **-axis** through θ as follows.

$$y' = y\cos\theta - z\sin\theta$$
$$z' = y\sin\theta + z\cos\theta \quad (2)$$
$$x' = x.$$

The corresponding rotation matrix is defined as

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

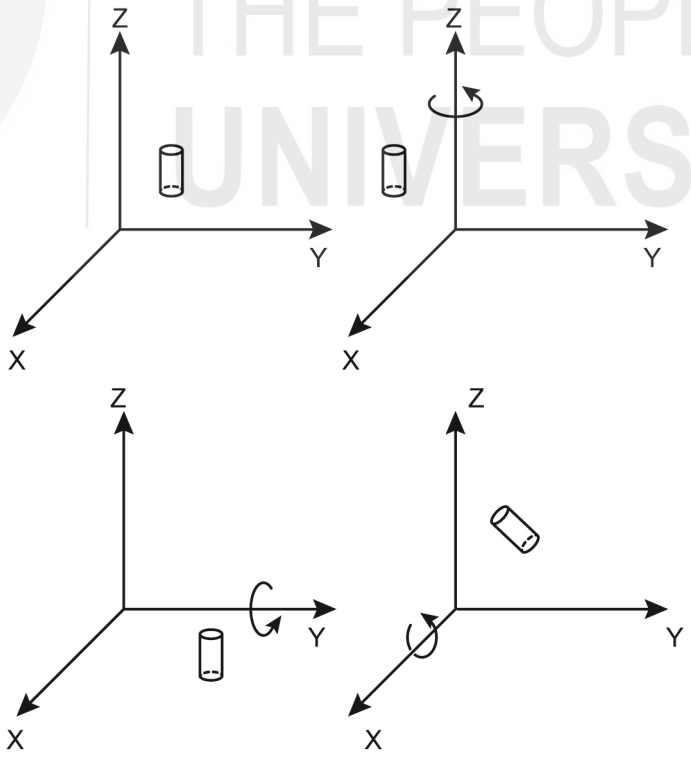
Similarly, if we replace x with y, y with z and z with x in Eqn. (2) we obtain **rotation transformations about the y-axis** through θ as follows:

$$z' = z\cos\theta - x\sin\theta$$
$$x' = z\sin\theta + x\cos\theta \quad (3)$$
$$y' = y.$$

The corresponding rotation matrix is given by

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The following picture illustrate the rotations about each of the three coordinate axes.



As in the case of 2 dimensions, it can be observed that 3D translations and rotations are rigid-body transformations. That is, do not change the object shape or size.

We consider, here, an example.

Example 1: Rotate the circle $x^2 + y^2 = 1,\ z = 2$ about the y-axis by 90°.

Solution: Take any point $P(x,\ y,\ z)$ on the circle. Using the matrix $R_y(\theta)$, with $\theta = 90^\circ$, we get

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} z \\ y \\ -x \\ 1 \end{pmatrix}$$

So, we put $z = x',\ y = y'$ and $x = -z'$ in circle equation to get

$$y'^2 + z'^2 = 1,\ x' = 2.$$

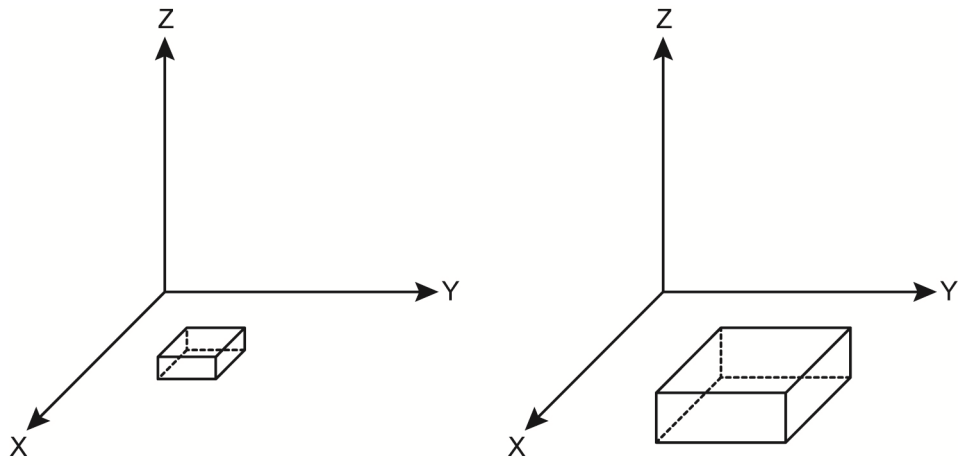
Since $(x',\ y',\ z')$ is arbitrary, we can simply write the new equation as This equation also represents a circle, with the same radius as of the original one.

***

Next, we discuss the scaling of an object, which is a non-rigid body transformation. Recall that in two dimensions a scaling can be in x or y direction. In three dimension, we can scale an object in each of the three dimensions. The three-dimensional scaling matrix is given below.

$$S(\alpha,\ \beta,\ \gamma) = \begin{pmatrix} \alpha & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \gamma & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Here α, β, γ can take any positive value. When $\alpha = \beta = \gamma$, the scaling is called **uniform**, otherwise **differential**. In uniform scaling *the object shape does not change after scaling* – an important thing to note. This is illustrated below.



If the value of any of the parameters α, β, γ is greater than 1, the object after scaling shifts away from the corresponding coordinate axis. On the other hand, if the value is less than 1, then the object comes closer to that coordinate axis. The following example illustrate this.

Example 2: Consider the pyramid $ABCDE$, where $\mathrm{A} = (2, 1, 0), \mathrm{B} = (2, 2, 0), \mathrm{C} = (1, 2, 0), \mathrm{D} = (1, 1, 0)$ and $E = \left(\frac{3}{2}, \frac{3}{2}, 1\right)$. Scale this pyramid with scaling factors $\alpha = \frac{1}{2}$, $\beta = \frac{1}{2}$ and $\gamma = 2$.

Solution: If P is the matrix of homogeneous coordinates of A, B, C, D, E, then we obtain the transformed matrix P' as follows:

$$P' = S\left(\tfrac{1}{2}, \tfrac{1}{2}, 2\right) P$$

$$= \begin{pmatrix} \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 2 & 1 & 1 & \frac{3}{2} \\ 1 & 2 & 2 & 1 & \frac{3}{2} \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 1 & \frac{1}{2} & \frac{1}{2} & \frac{3}{4} \\ \frac{1}{2} & 1 & 1 & \frac{1}{2} & \frac{3}{4} \\ 0 & 0 & 0 & 0 & 2 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Thus the pyramid after scaling has vertices $A'\left(1, \frac{1}{2}, 0\right)$, $B'(1, 1, 0)$, $C'\left(\frac{1}{2}, 1, 0\right)$, $D'\left(\frac{1}{2}, \frac{1}{2}, 0\right)$, and $E'\left(\frac{3}{4}, \frac{3}{4}, 2\right)$.

***

Now we shall look at the compositions of some elementary transformations. First observe that

$$T(a, b, c)T(a', b', c') = T(a', b', c')\ T(a, b, c) = T(a + a', b + b', c + c').$$

This means 3D translations are commutative as well as additive in nature, just like their 2D fellows. Likewise, you can also see that two 3D scaling are commutative and multiplicative in nature, i.e.,

$$S(\alpha, \beta, \gamma)S(\alpha', \beta'\ \gamma') = S(\alpha', \beta', \gamma')S(\alpha, \beta, \gamma) = S(\alpha\alpha', \beta\beta', \gamma\gamma').$$

However, this is not the case with elementary rotations. For instance, just verify that $R_x(\theta)R_y(\phi) \neq R_y(\phi)R_x(\theta)$. This implies that you have to be careful while choosing two successive rotations for an object. In fact, elementary rotations do not commute with translations or scaling as well. (See E2.) What do you think about the commutativity of a translation with a scaling? See the following example.

Example 3: Show that $T(a, b, c)\ S(\alpha, \beta, \gamma) \neq S(\alpha, \beta, \gamma)\ T(a, b, c)$.

Solution:

$$\text{LHS} = \begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \gamma & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \alpha & 0 & 0 & a \\ 0 & \beta & 0 & b \\ 0 & 0 & \gamma & c \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Likewise, compute that

$$\text{RHS} = \begin{pmatrix} \alpha & 0 & 0 & a\alpha \\ 0 & \beta & 0 & b\beta \\ 0 & 0 & \gamma & c\gamma \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Thus LHS $\neq$ RHS.

***

Now you may try the following exercise.

E1) Translate the object represented by the equation $x^2 + y^2 + z^2 - 2 - 4y + 8z + 20 = 0$ by the point $(-1, -2, 4)$.

E2) Show that

i) $R_x(\theta)\ T(a, b, c) \neq T(a, b, c)\ R_x(\theta)$.

ii) $R_x(\theta)\ S(\alpha, \beta, \gamma) \neq (\alpha, \beta, \gamma)\ R_x(\theta)$.

E3) Show that

i) $R_x^{-1}(\theta) = R_x(-\theta) = R_x(\theta)^T$.

ii) $R_y^{-1}(\theta) = R_y(-\theta) = R_y(\theta)^T$.

iii) $R_z^{-1}(\theta) = R_z(-\theta) = R_z(\theta)^T$.

E4) An object is rotated about the z-axis with an angle of 60° and then it is uniformly scaled up by a factor of 2. Find the resultant transformation matrix.

In the next section, we shall see general three dimensional rotations as a combination of elementary rotations and translations.

6.3 GENERAL THREE-DIMENSIONAL ROTATION

Obtaining the transformation matrices for general three-dimensional rotation about an arbitrary axis involves a combination of translations and rotations about the coordinate axes. Let us first deal with the special case, where the rotation axis is parallel to one of the coordinate axes. More specifically, let L

be the rotation axis that is parallel to the x-axis and which passes through the point (a, b, c). Then we need to first apply a translation so that L coincides with the x-axis. This can be achieved by multiplying every point of the object by $T(-a, -b, -c)$. Next, we rotate the resulting object about the x-axis through θ, using the matrix $R_x(\theta)$. Finally, we translate back the resulting object using the matrix $T(a, b, c)$. Thus the sequence of transformations is

$$T(a, b, c) \cdot R_x(\theta)\, T(-a, -b, -c).$$

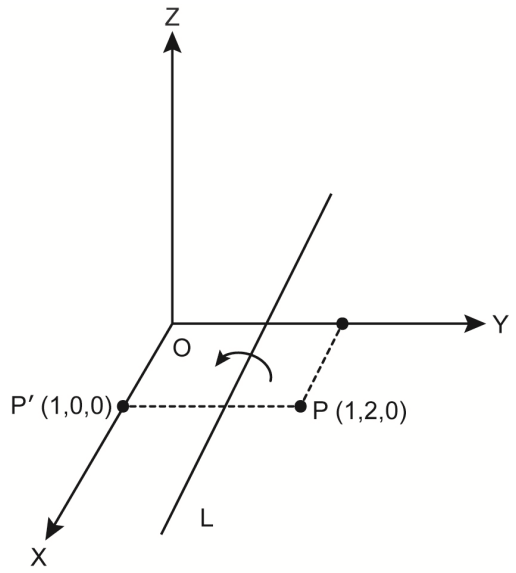
Suppose for instance, we wish to rotate the point $P(1, 2, 0)$ about the line parallel to the x-axis and passing through $(0, 1, 0)$ by the angle π. Then the resulting transformation matrix is

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 2 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Hence $P' = MP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 2 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$.

Thus P transforms into the point $P'(1, 0, 0)$ as shown below.



The other two transformation matrices corresponding to the rotation parallel to y and z-axes, respectively, are given below:

i) $T(a, b, c)R_y(\theta)T(-a, -b, -c)$,

ii) $T(a, b, c)R_z(\theta)T(-a, -b, -c)$.

Example 4: Rotate an object through an angle of 45 degrees with centre of the object on $(1, 2, 3)$. The axis of rotation is given by the direction vector $(1, 1, 1)$ passing through $(1, 0, 0)$.

Solution: Here

$$P = (1, 2, 3),\ V = (1, 1, 1),\ a = b = c = \frac{1}{\sqrt{3}},\ d = \sqrt{\frac{2}{3}}.$$

$$\cos\alpha = \sin\alpha = \frac{1}{\sqrt{2}},\ \cos\beta = \sqrt{\frac{2}{3}},\ \sin\beta = -\frac{1}{\sqrt{3}}.$$

Therefore

$$\mathbf{R}_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}..$$

Thus

$$\mathbf{R}_y(\beta) = \begin{pmatrix} \sqrt{\frac{2}{3}} & 0 & -\frac{1}{\sqrt{3}} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{1}{\sqrt{3}} & 0 & \sqrt{\frac{2}{3}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

So, the object will be rotated with its centre P now on the position P' with P' given by the following concatenation of the matrices.

$$P' = T(1, 0, 0)R_x(\alpha)^T R_y(\beta)^T R_z(45^\circ)R_y(\beta)R_x(\alpha)T(-1, 0, 0)P$$

Now you can calculate the coordinates of the changed position of P by multiplying by the matrices in the above order.

***

Now we shall write the sequence of transformations for **rotation about an arbitrary rotation axis**. Recall that a line in three dimensions can be specified by a point on it and its direction cosines. Let $V = (v_x, v_y, v_z)$ be the vector giving the direction cosines of the rotation axis that passes through the point (x_0, y_0, z_0). Now define

$$a = \frac{v_x}{|V|},\ b = \frac{v_y}{|V|},\ c = \frac{v_z}{(V)},\ d = \sqrt{b^2 + c^2}.$$

Set

$$\cos\alpha = \frac{c}{d},\ \sin\alpha = \frac{b}{d},\ \cos\beta = d,\ \sin\beta = -a.$$

Then the sequence of transformations that rotates an object by an angle of θ about the line passing through $(x_0,\ y_0,\ z_0)$ and having direction vector as V is

$$R(\theta) = T(x_0,\ y_0,\ z_0)R_x^{-1}(\alpha)R_y^{-1}(\beta)R_z(\theta)R_y(\beta)R_x(\alpha)T(-x_0,\ -y_0,\ -z_0)$$

$$= T(x_0,\ y_0,\ z_0)R_x(\alpha)^T R_y(\beta)^T R_z(\theta)R_y(\beta)R_x(\alpha)T(-x_0,\ -y_0,\ -z_0).$$

For further understanding of the general 3D transformation about an arbitrary axis, let us consider the following examples.

Example 5: An object has to be rotated by $\frac{\pi}{6}$ about an axis passing through the points $(1,\ 0,\ 1),\ (1,\ 3,\ 1)$. What will be the resulting rotation matrix?

Solution: The direction vector of the axis of rotation is $V = (1-1,\ 3-0,\ 1-1) = (0,\ 3,\ 0)$. So, the axis is parallel to y axis. Take $(a,\ b,\ c) = (1,\ 0,\ 1)$. Then the sequence of matrix transformations is given by

$$R = T(a,\ b,\ c)R_y(\theta)T(-a,\ -b,\ -c)$$

$$= \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\frac{\pi}{6} & 0 & \sin\frac{\pi}{6} & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\frac{\pi}{6} & 0 & \cos\frac{\pi}{6} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{\sqrt{3}}{2} & 0 & \frac{1}{2} & \frac{1}{2}(1-\sqrt{3}) \\ 0 & 1 & 0 & 0 \\ -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2} & \frac{1}{2}(3-\sqrt{3}) \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

***

Example 6: Scale a sphere centered on the point $(1,\ 2,\ 3)$ with radius 1, so that the new sphere has the same centre with radius 2.

Solution: Since the centre of the sphere remains unchanged, we choose the fixed point to be $(a,\ b,\ c) = (1,\ 2,\ 3)$. Also the scaling is uniform, with $\alpha = \beta = \gamma = 2$. The required transformation is

$$R = T(a,\ b,\ c)\ S(\alpha,\ \beta,\ \gamma)T(-a,\ -b,\ -c)$$

$$= \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 2 & 0 & 0 & -1 \\ 0 & 2 & 0 & -2 \\ 0 & 0 & 2 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

***

You may now try the following exercises.

E5) A scene is modelled in 3D. Write the transformation of matrix that will scale the scene by a factor of 2 in y direction while leaving the position of $(2, -1, 1)$ unchanged. Under this transformation, what are the new equations of the line $x = y = z$?

E6) Rotate a scene modelled in 3D by an angle of θ degrees such that the axis of rotation is along the vector $(1, 1, 0)$ and the origin remains unchanged. Write the transformation matrix.

You have studied and implemented OpenGL functions for 2D transformations. As you know in OpenGL every entity is treated as a 3D entity. Actually the transformation functions are also created with this philosophy only. Hence the OpenGL functions that you have used for 2D transformations are also used for 3D geometric transformations. The relevant OpenGL routines are given below:

- `glScaled` (α, β, γ) : Scale by α in x direction, by β in y-direction and by γ in z-coordinate direction.

- `glTranslated(a,b,c)` : Translate by a in x-direction, by b in y-direction and by c in z direction.

- `glRotated(angle,vx,vy,vz)` : Rotate by '`angle`' degrees about the vector (vx, vy, vz). If the vector is not unit vector, the routine automatically converts that to unit vector.

Note that, you have to be careful while applying OpenGL matrix transformation. The order in which they are applied is just opposite of the

order in which they are written. For example, if you call the following functions in the order given below -

```
glScaled(1.0, 2.0, 1.5); // S
glRotatef(45.0, 0.0, 1.0, 0.0); // R
glTranslatef(3.0, 4.0, 1.0); // T
```

the transformations occur in the reverse order

$$S * R * T * P = S * (R * (T * P)),$$

where P is the point on which you are applying the transformation.

Let us look at an example.

Example 7: Suppose you wish to apply the following transformations on a point $P(2, 0, 4)$ in the given order:

i) rotate by 30° about the x-axis.

ii) Scale by 2 along the x-axis, by 4 along the y-axis and by $\frac{1}{2}$ along the z-axis.

iii) Rotate by -60° about the line with direction ratios $3, 1, 2$.

Write OpenGL code for performing these transformations.

Solution: The required code is given below.

```
glrotated(-60, 3, 1, 2);
glscaled(2, 4, 0.5);
glrotated(30, 1, 0, 0);
glBegin(GL_POINTS);
glVertex3d(2, 0, 4);
glEnd();
```

***

But before using these transformations, you need to know about the concept of viewing and related operations. So, we move to the next section where you will learn about setting up a view direction and transforming your coordinate system accordingly.

6.4 THREE-DIMENSIONAL VIEWING

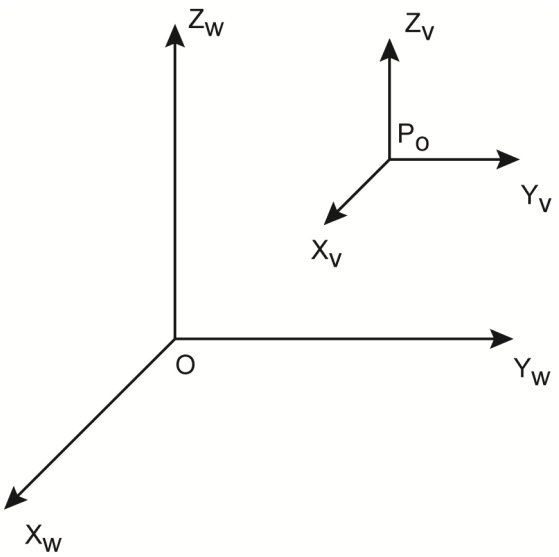
Three dimensional objects are created using modelling coordinate system. The modelled objects are then placed in locations specified in the scene with respect to the world coordinate system. You can view the scene from any orientation and this is what you call as the view of the scene. Viewing

coordinates are the coordinates generated with respect to the frame of reference of the viewer, viewing transformation is a composite transformation, which transforms the scene from world coordinate system to viewing coordinate system.

To understand what viewing coordinate system and the viewing transformation are, let us consider a simple example. Suppose you have a vacant room and you want to organize its interior to make it an office. Chairs, tables, shelves, etc. are created somewhere else (in some factory) using some coordinate system. Let that be treated as a modelling coordinate system. Then these items will be brought in the room (Transform to locations in the room). Finally every object will be placed in appropriate position. This means the modelled objects are now placed in the world coordinate system. Now you view it from various angles and then become satisfied with the arrangements. When you are watching it from various angles, each time you change your position, the relative positions of the objects in the room also change. Objects closer to you become farther when you go on the other side of the room. This is the viewing coordinate system and your eyes make an automatic transformation of objects from world coordinate system to viewing coordinate system.

Let us now discuss how to transform from world coordinate system to viewing coordinate system.

Given the world coordinate system $OX_wY_wZ_w$ we construct a **viewing coordinate system** $P_oX_vY_vZ_v$ where $P_o(x_o, y_o, z_o)$ is some arbitrary point in the world coordinate system. We call P_o the **view reference point**. (see the following picture.)



Next, we select the positive direction for viewing the Z_v-axis, through a vector called the **view-plane normal vector** N. The vector N is defined usually in world coordinates. Note that the selection of N automatically specifies the view-plane. Then we need a vector V called the **view-up vector** that specifies the up direction for the view, that is, for the Y_v-axis. Although, the vector V must be perpendicular to N, we can select V as any vector not parallel to N. The graphic package would adjust the direction of V with the help of another vector U that is perpendicular to both N and V. The vector U specifies the direction of the X_v-axis.

The first task in viewing a three-dimensional object is to translate the view reference point $P_o(x_o, y_o, z_o)$ to the origin of the world-coordinate system. This is achieved by the matrix $T(-x_o, -y_o, -z_o)$. Next, we rotate the viewing coordinate system so that it is aligned with the world coordinate system. This is done by computing the matrix R as follows.

Define vectors n, u and v as shown below.

$$n = \frac{N}{|N|} = (n_1, n_2, n_3)$$

$$u = \frac{V \times N}{|V \times N|} = (u_1, u_2, u_3)$$

$$v = n \times u = (v_1, v_2, v_3).$$

Then R is given by

$$R = \begin{pmatrix} u_1 & u_2 & u_3 & 0 \\ v_1 & v_2 & v_3 & 0 \\ n_1 & n_2 & n_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Now the **world-to-viewing coordinate transformation** matrix is defined as $M = R.T(-x_0, -y_0, -z_0).$

Note that some graphics packages use a vector L called the **look-at vector** to determine the direction of N. The vector L is a point in the world coordinate system where the camera or eye is looking at. In such a case, we set $N = P_o - L.$

Let us now consider an example.

Example 8: Transform the scene in the world coordinate system to the viewing coordinate system with view reference point at $(2, 2, 2)$. The view plane normal vector is $(-1, -1, -1)$ and the view up vector is $(0, 1, 0)$.

Solution: We have $P_o = (2, 2, 2)$, $\boldsymbol{N} = (-1, -1, -1)$ and $\boldsymbol{V} = (0, 1, 0)$.

Therefore $V \times N = (-1, 0, 1)$ and $|V \times N| = \frac{1}{\sqrt{2}}$.

$$\boldsymbol{n} = \frac{-1}{\sqrt{3}}(1, 1, 1), \boldsymbol{u} = \frac{1}{\sqrt{2}}(-1, 0, 1)$$

So, $v = n \times u = \begin{vmatrix} i & j & k \\ -\frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{3}} \\ -\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{vmatrix} = -\frac{1}{\sqrt{6}} i + \frac{2}{\sqrt{6}} j - \frac{1}{\sqrt{6}} k$

$$= \left(-\frac{1}{\sqrt{6}}, \frac{2}{\sqrt{6}}, -\frac{1}{\sqrt{6}} \right).$$

The resulting sequence of matrix operations is therefore given by

$$M = R.T(-2, -2, -2)$$

$$= \begin{pmatrix} \frac{-1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ \frac{-1}{\sqrt{6}} & \frac{2}{\sqrt{6}} & \frac{-1}{\sqrt{6}} & 0 \\ \frac{-1}{\sqrt{3}} & \frac{-1}{\sqrt{3}} & \frac{-1}{\sqrt{3}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -2 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} -\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{6}} & \frac{2}{\sqrt{6}} & -\frac{1}{\sqrt{6}} & 0 \\ -\frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{3}} & 2\sqrt{3} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

***

You may now try the following exercise.

E7) Your camera is located at $(2, 0, 0)$ in the world coordinate system. You are looking at $(0, 1, 0)$. If the viewup vector is $(2, 3, 4)$, how will you transform the scene from the world coordinate system to viewing coordinate system?

OpenGL Model View and related Matrix Operations

Since modelling is constructing the 3D model, modelling transformations are concerned with positioning (translation), orientation (rotation) and scaling of the objects in the virtual world and with their relative positions, orientations and sizes (scale). Following functions facilitate you to perform the operations related to modelling the scene.

- `glMatrixMode(GL_MODELVIEW );`
  - Specifies that we require space for a 4×4 matrix to be used for setting up the view of the model and in common with all the derived matrices. The function glMatrixMode( ) is used for setting up both viewing- modellling and projection. When you wish to set up the modelling and viewing transformation, use the symbolic constant `GL_MODELVIEW`.
  - GL_MODELVIEW matrix combines viewing matrix and modelling matrix into one matrix.
  - The matrix mode is one of OpenGL states it remembers. Basically it means that all subsequent operations will be performed on whatever matrix you specify in glMatrixMode( ) until you change it to a different mode. So if you specify the modelling and viewing matrix GL_MODELVIEW, it will perform all subsequent matrix operations treating this as the initial matrix.
- `glLoadIdentity( );`.
  - Initializes the current matrix as the identity matrix.
- `gluLookAt` $(e_Z, e_y, e_z, C_x, C_y, C_Z, up_X, up_Y, up_Z)$;
  - Used for positioning, pointing and orienting the virtual camera. Here
    - (e_Z, e_y, e_z) give the location of the camera / eye.
    - (C_x, C_y, C_Z) specify the pointing direction of the camera.
    - (up_X, up_Y, up_Z) give the orientation of the camera.

For instance, a call `gluLookAt` $(0, 0, 10, 1, 1, 2, 0, 1, 0)$ indicates that the camera is positioned at $(0, 0, 10)$ pointing at the point $(1, 1, 2)$ and having y-axis as the up vector.

We now end the unit by giving a summary of what we have covered in it.

6.5 SUMMARY

In this unit, we have covered the following points:

1) We have discussed three-dimensional translations, scaling and coordinate axes rotations.

2) We have seen that the rotation about an arbitrary axis is a composition of several rotations and translation operations.

3) The compositions of different 3D transformations are also discussed.

4) We have seen how to transform from the world coordinate system to viewing coordinate system.

5) Finally, we have seen the OpenGL functions for modelling and viewing transformations which are:

   a) `glMatrixMode(GL_MODELVIEW );`

   b) `gluLookAt (eyeX, eyeY, eyeZ, atX, atY, atZ, upX, upY, upZ);`

6.6 SOLUTIONS/ANSWERS

E1) The translation equations are $x' = x - 1,\ y' = y - 2,\ z' = z + 4.$ These equations give $x = x' + 1,\ y = y' + 2$ and $z = z' - 4.$ Now substituting these values in the given equation, we get

$$(x'+1)^2 + (y'+2)^2 + (z'-4)^2 - 2(x'+1) - 4(y'+2) + 8(z'-4) + 20 = 0.$$

E2) i)
$$R_x(\theta)T(a, b, c) = \begin{pmatrix} 1 & 0 & 0 & a \\ 0 & \cos\theta & -\sin\theta & b\cos\theta - c\sin\theta \\ 0 & \sin\theta & \cos\theta & b\sin\theta + c\cos\theta \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$T(a, b, c)R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & a \\ 0 & \cos\theta & -\sin\theta & b \\ 0 & \sin\theta & \cos\theta & c \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

$\therefore R_x(\theta)T(a, b, c) \neq T(a, b, c)R_x(\theta).$

ii) Do yourself.

E3) i) Observe that $R_x(\theta)R_x(-\theta) = I = R_x(-\theta)R_x(\theta).$ Therefore, $R_x^{-1}(\theta) = R_x(-\theta).$ Also, it can be easily seen that

$R_x(-\theta) = R_x(\theta)^T$.

ii) Try yourself.

iii) Try yourself.

E4) $\begin{pmatrix} 1 & -\sqrt{3} & 0 & 0 \\ \sqrt{3} & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$.

E5) Translate by $(-2, 1, -1)$, scale by a factor of 2 in y-direction and then translate by $(2, -1, 1)$. The required transformation matrix is:

$$M = T(2, -1, 1)\ S(1, 2, 1)\ T(-2, 1, -1)$$

$$= \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Any point (x, y, z) under this transformation changes to

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ 2y+1 \\ z \\ 1 \end{pmatrix}.$$

This gives, $x = x'$, $y = \frac{y'-1}{2}$, $z = z'$. Substituting these values in $x = y = z$, we get

$$x' = \frac{y'-1}{2} = z.$$

E6) Here $V = (1, 1, 0)$ and $(x_0, y_0, z_0) = (0, 0, 0)$. So, $a = \frac{1}{\sqrt{2}}$, $b = \frac{1}{\sqrt{2}}$,

$c = 0$, $d = \frac{1}{\sqrt{2}}$.

$\cos\alpha = 0$, $\sin\alpha = 1$, $\cos\beta = \frac{1}{\sqrt{2}}$, $\sin\beta = -\frac{1}{\sqrt{2}}$.

$$R_x(\alpha)=\begin{pmatrix}1&0&0&0\\0&0&-1&0\\0&1&0&0\\0&0&0&1\end{pmatrix},\ R_y(\beta)=\begin{pmatrix}\frac{1}{\sqrt{2}}&0&-\frac{1}{\sqrt{2}}&0\\0&1&0&0\\\frac{1}{\sqrt{2}}&0&\frac{1}{\sqrt{2}}&0\\0&0&0&1\end{pmatrix}.$$

$$\therefore R_y(\beta)R_x(\alpha)=\begin{pmatrix}\frac{1}{\sqrt{2}}&-\frac{1}{\sqrt{2}}&0&0\\0&0&-1&0\\\frac{1}{\sqrt{2}}&\frac{1}{\sqrt{2}}&0&0\\0&0&0&1\end{pmatrix}$$

$$R_x(\alpha)^T R_y(\beta)^T=\begin{pmatrix}\frac{1}{\sqrt{2}}&0&\frac{1}{\sqrt{2}}&0\\-\frac{1}{\sqrt{2}}&0&\frac{1}{\sqrt{2}}&0\\0&-1&0&0\\0&0&0&1\end{pmatrix}$$

Also $T(x_0,\ y_0,\ z_0)=T(-x_0,-y_0,-z_0)=I.$ Thus the required transformation matrix is

$$R(\theta)=\begin{pmatrix}\frac{1}{\sqrt{2}}&0&\frac{1}{\sqrt{2}}&0\\-\frac{1}{\sqrt{2}}&0&\frac{1}{\sqrt{2}}&0\\0&-1&0&0\\0&0&0&1\end{pmatrix}\begin{pmatrix}\cos\theta&-\sin\theta&0&0\\\sin\theta&\cos\theta&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}\begin{pmatrix}\frac{1}{\sqrt{2}}&-\frac{1}{\sqrt{2}}&0&0\\0&0&-1&0\\\frac{1}{\sqrt{2}}&\frac{1}{\sqrt{2}}&0&0\\0&0&0&1\end{pmatrix}$$

$$=\begin{pmatrix}\frac{1+\cos\theta}{2}&\frac{1-\cos\theta}{2}&\frac{1}{\sqrt{2}}\sin\theta&0\\\frac{1-\cos\theta}{2}&\frac{1+\cos\theta}{2}&-\frac{1}{\sqrt{2}}\sin\theta&0\\-\frac{1}{\sqrt{2}}\sin\theta&\frac{1}{\sqrt{2}}\sin\theta&\cos\theta&0\\0&0&0&1\end{pmatrix}$$

E7) Here $P_0=(2,\ 0,\ 0),\ L=(0,\ 1,\ 0).$ So $N=P_0-L=(2,-1,\ 0).$ Also, $V=(2,\ 3,4).$

Now $n=\frac{N}{|N|}=\left(\frac{2}{\sqrt{5}},-\frac{1}{\sqrt{5}},\ 0\right).$

$$u=\frac{V\times N}{|V\times N|}=\frac{1}{3}(1,\ 2,-2).$$

$$v=n\times u=\left(\frac{2}{3\sqrt{5}},\frac{4}{3\sqrt{5}},\frac{5}{3\sqrt{5}}\right)$$

$$\therefore R = \begin{pmatrix} \frac{1}{3} & \frac{2}{3} & -\frac{2}{3} & 0 \\ \frac{2}{3\sqrt{5}} & \frac{4}{3\sqrt{5}} & \frac{5}{3\sqrt{5}} & 0 \\ \frac{2}{\sqrt{5}} & -\frac{1}{\sqrt{5}} & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Therefore, the world-to-viewing transformation matrix is

$$M = R.T(-2, 0, 0) = \begin{pmatrix} \frac{1}{3} & \frac{2}{3} & -\frac{2}{3} & \frac{2}{3} \\ \frac{2}{3\sqrt{5}} & \frac{4}{3\sqrt{5}} & \frac{5}{3\sqrt{5}} & \frac{4}{3\sqrt{5}} \\ \frac{2}{\sqrt{5}} & -\frac{1}{\sqrt{5}} & 0 & \frac{4}{\sqrt{5}} \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

UNIT 7

DISPLAY METHODS

Structure **Page No.**

7.1 INTRODUCTION

In the last unit, you have been introduced to a number of geometric transformations for three-dimensional objects. You must have also learned how to transform an object from world coordinates to viewing coordinates.

In this unit beginning from Section 7.2, we shall learn a few more methods for displaying three-dimensional objects. Viewing a 3D scene on a 2D display device actually requires to project the image on the plane of the display device. There are two types of projections namely *parallel projection* and *perspective projection,* each focussing on a different aspect of picture viewing. We shall talk about these projections in Sections 7.3 and 7.4 in detail. Finally, in Section 7.5, we discuss view volumes and general projection transformations.

Objectives

After reading this unit you should be able to

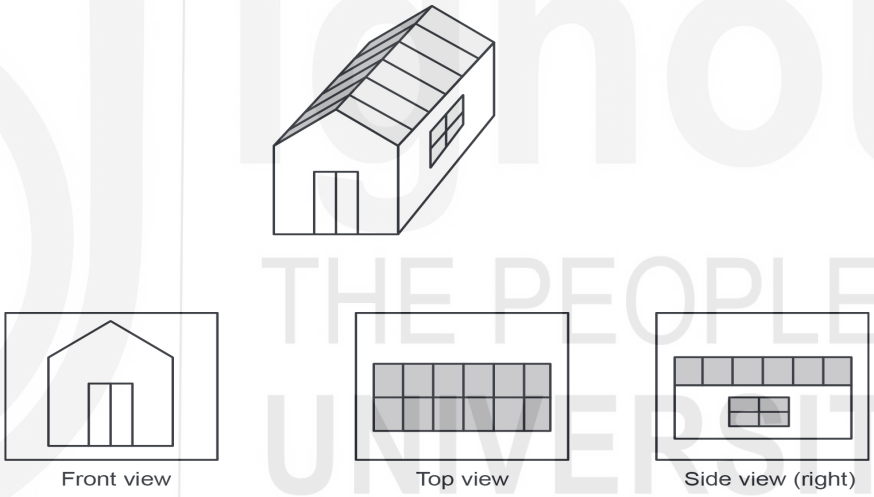
- explain the concepts used in rendering and projecting a 3D scene onto a 2D display;

- build model view matrix capturing the scene from camera position;
- implement projection transformations;
- distinguish between parallel and perspective projections.

7.2 THREE-DIMENSIONAL DISPLAY METHODS

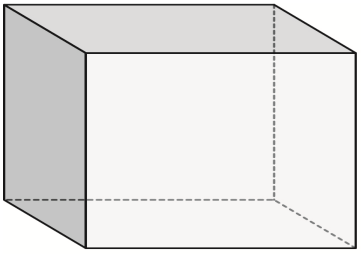
Three dimensional objects are created using modelling coordinate system, and then transformed to the specified locations in the scene with respect to the world coordinate system. You can view the scene from any orientation viewing coordinates. The coordinate reference frame for the "camera" defines the position and orientation for the plane of the camera film. The scene or objects are finally displayed on this plane after projecting them on the camera film.

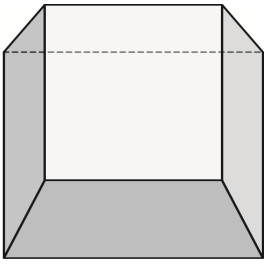
One method for projecting a 3D image is **parallel projection** under which parallel lines in the world coordinates project into parallel lines on the 2D plane. For instance, look at the following picture of a house.



Three different views – front view, side view and top view – give three distinct plane images of the same house. A distinctive feature of parallel projection is that the relative proportions of different parts of the object are maintained.

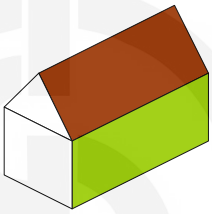
Another method for viewing a 3D image is **perspective projection**. Under this projection the object positions transform to the 2D view plane along lines that converge to a distant point called the **projection reference point**. The projected image is actually obtained by finding the intersection of the projection lines with the view plane. Under a perspective projection objects that are closer to viewing positing appear larger than the ones which are farther from the viewing position. This is what we see in realistic pictures. The camera system as well as our eye capture images using perspective projection. For instance, given below are two pictures of a cube where the left one is drawn using parallel projection and the right one using perspective projection.



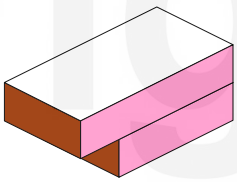


While projecting an image there are many other factors that need to be taken into account. **Depth information** is one of them which tells which part of the object lies to the front and which one at the back. Both the pictures of the cube given above lack depth information, due to which their front and back side keep on changing as one lingers on them. One way to incorporate the depth information is by using varied colour or light density or using different colour combinations. Depth information can also be presented through three dimensional display devices or by projecting stereoscopic views.

E1) Draw an approximate (i) front view (ii) top view and (iii) view from left, of the objects shown below.



(a)

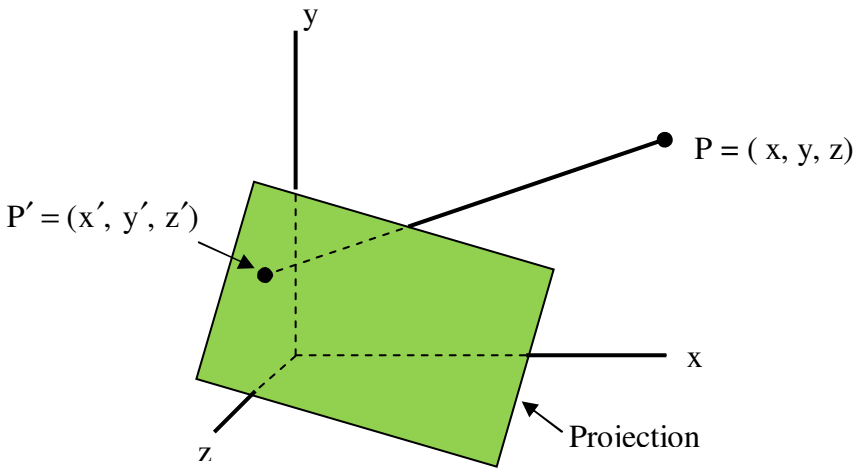


(b)

Our primary concern here is to learn about the two projection methods in detail.

7.3 PROJECTIONS

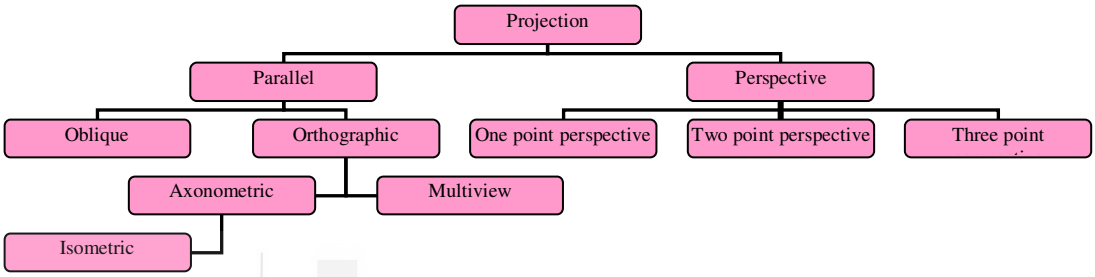
Mathematically, the projection of a point $P = (x, y, z)$ in the 3D world is a point $P' = (x', y', z')$ in a plane called the **projection plane** or the **view plane** as shown below.



The point P' is actually obtained as the intersection of the view plane with the line passing through P along a particular direction, specified by a direction vector called the **projection vector**.

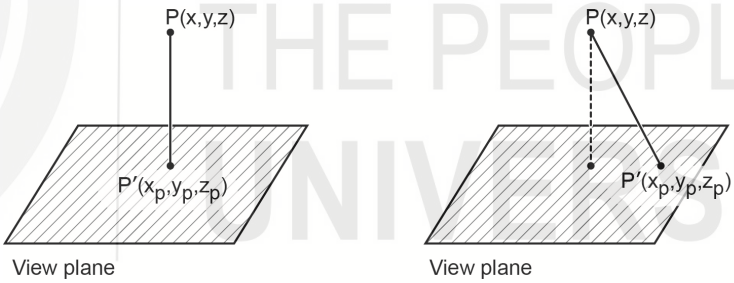
Projections can be classified into two types – parallel projections and perspective projections. Depending on the type of application, one of the projection types is used. While parallel projections are extensively used in architectural drawing, perspective projections are used to bring reality to the projected image, e.g., in games and virtual reality problems.

The taxonomy of projections is as shown below.

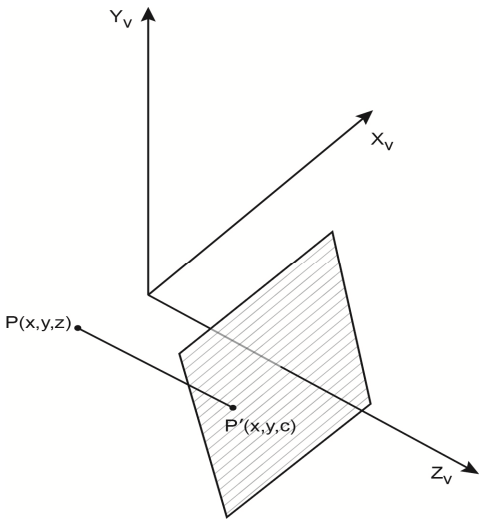


7.3.1 Parallel Projection

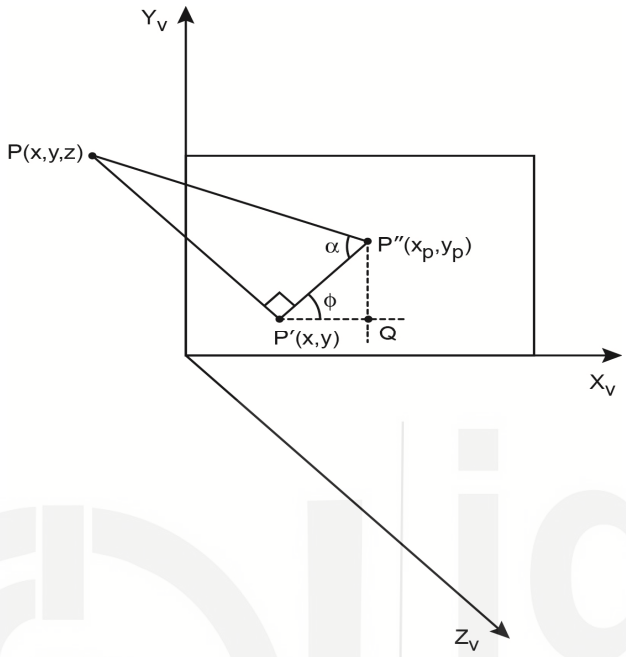
To specify a parallel projection we need a **projection vector** that defines the direction of the projection lines. This vector, of course, cannot be parallel to the view plane. If this vector is perpendicular to the view plane, then the projection is called **orthographic projection**, otherwise **oblique projection**. See the following pictures below for illustration.



For instance, if the view pane is parallel to the X_vY_v-plane and is situated at a distance c from the origin, then the projection of $P(x, y, z)$ is the point $P'(x, y, c)$ as shown below.



Now we shall derive the transformation matrix for parallel projection taking X_vY_v-plane as the view plane. Consider any point $P(x, y, z)$. Let $P'(x, y)$ be its orthographic projection and $P''(x_p, y_p)$ its oblique projection. See the picture below.



Further assume that the projection vector makes an angle α with the view plane. That is the line PP'' makes an angle α with the line $P'P''$. Also, let ϕ be the angle that the line $P'P''$ makes with the X_v-axis. Now we have

$$\cos\phi = \frac{|P'Q|}{|P'P''|} = \frac{x_p - x}{|P'P''|} \quad (1)$$

$$\sin\phi = \frac{|P''Q|}{|P'P''|} = \frac{y_p - y}{|P'P''|} \quad (2)$$

From the $\Delta PP'P''$, we find that $\tan\alpha = \frac{|PP'|}{|P'P''|} = \frac{z}{|P'P''|}$. So, $|P'P''| = \frac{z}{\tan\alpha}$.

Substituting this in Eqns. (1) and (2) and solving for x_p, y_p we get

$$x_p = x + z\frac{\cos\phi}{\tan\alpha}$$

$$y_p = y + z\frac{\sin\phi}{\tan\alpha}.$$

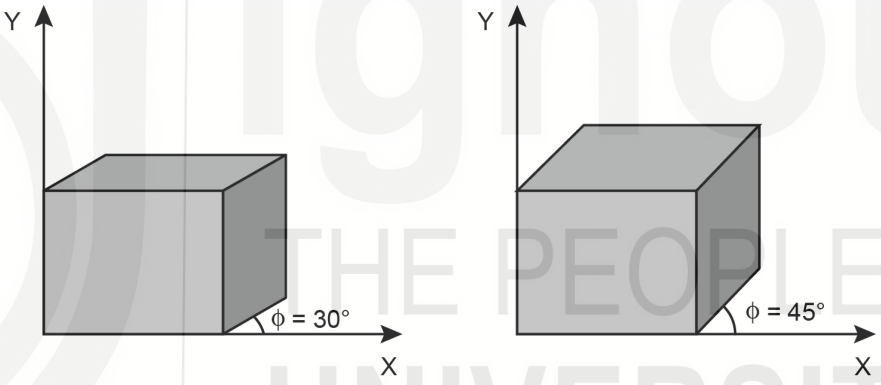
In homogeneous matrix form these equations can be written as

$$\begin{pmatrix} x_p \\ y_p \\ z_p \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \frac{\cos\phi}{\tan\alpha} & 0 \\ 0 & 1 & \frac{\sin\phi}{\tan\alpha} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}.$$

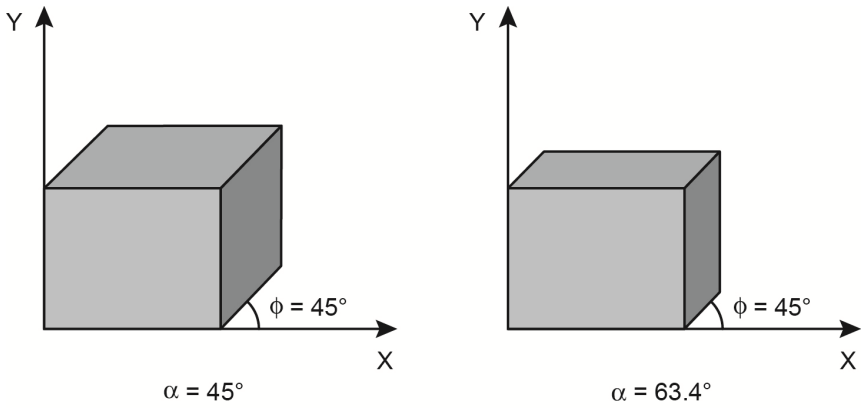
Thus the parallel projection matrix is

$$M_{parallel} = \begin{pmatrix} 1 & 0 & \frac{\cos\phi}{\tan\alpha} & 0 \\ 0 & 1 & \frac{\sin\phi}{\tan\alpha} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Thus the angles α and ϕ completely describe a parallel projection. When $\alpha = 90^\circ$, the projection is orthographic and in this case the third column of $M_{parallel}$ becomes zero. To project any 3D object on a 2D plane we need to specify the values of α and ϕ, which help us draw front, side and top (or bottom) views. Suppose you wish to draw a cube on a paper. The front face can be drawn as a square. To draw the side face we must know the angle its edges make with the horizontal axis. This angle is ϕ. Two commonly used values for ϕ are 30° and 45°. With these values and keeping α fixed, the cube can be drawn as follows.



How long the side edges should be? It depends on the value of α. The value of α is usually chosen in such a way that $\tan\alpha = 1$ or $\tan\alpha = 2$. For $\tan\alpha = 1$, we get $\alpha = 45^\circ$ and this case the parallel projection is called **cavalier projection**. On the other hand, for $\tan\alpha = 2$ we get $\alpha = 63.4^\circ$ and this case the parallel projection is called **cabinet projection**. Now keeping $\phi = 45^\circ$, we the cube with cavalier and cabinet projections as follows.



Note that the length of the lines perpendicular to the view plane remains unchanged in cavalier projection. However, in cabinet projection the depth of an object is half of its width or height, and hence the object appear more realistic than in cavalier projection.

For a better understanding of different types of projections we are giving here an example.

Example 1: Imagine a square with vertices $(0, 0, 1), (1, 0, 1), (1, 0, 2)$ and $(0, 0, 2)$. Find out its image on the view plane aligned with the xy-plane under following projections (i) Orthographic parallel projection (ii) Oblique parallel projection with $\alpha = \phi = 45^\circ$.

Solution: For an orthographic projection the view direction has to be perpendicular to the view plane and the square object will be projected as given below in (i). In case of an oblique parallel projection with projection angle parameters $\alpha = \phi = 45^\circ$, a point (x, y, z) will be projected to $\left(x + \frac{z}{\sqrt{2}}, y + \frac{z}{\sqrt{2}}, 0\right)$. You can now easily compute the projection of each vertex as per the rule. The transformed points given in the sequence are as follows.

(i) orthographic projection – $(0, 0), (1, 0), (1, 0), (0, 0)$.

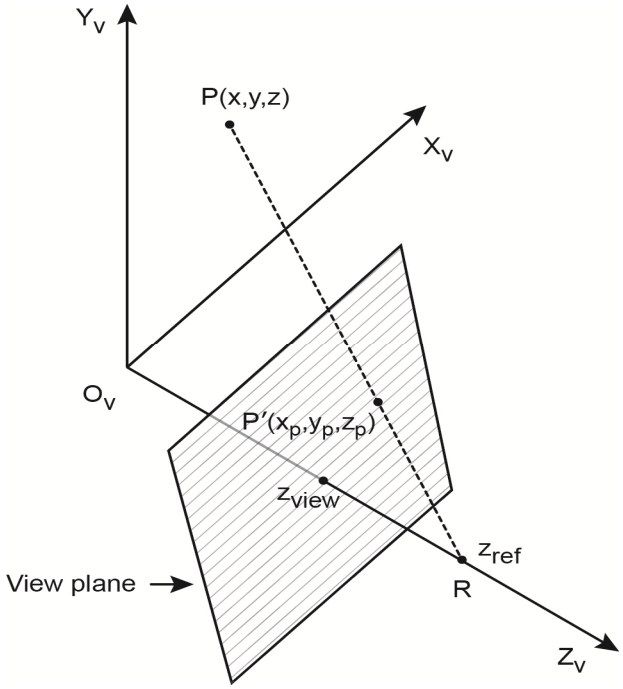
(ii) oblique parallel projection – $\left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right), \left(1+\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right), (1+\sqrt{2}, \sqrt{2}), (\sqrt{2}, \sqrt{2})$.

Try drawing the result on the paper. Notice that while the shape of square gets deformed with oblique parallel projection, in orthographic projection, it reduces to a straight line segment.

***

7.3.2 Perspective Projection

Consider a viewing coordinate system $X_vY_vZ_v$ with origin at O_v. Let the view plane be parallel to the X_vY_v-plane lying at a distance z_{view} from the origin. Also let the projection reference point be lying on the z_v-axis at $R(0, 0, z_{ref})$. Then the perspective projection of a point $P(x, y, z)$ in world coordinates is the point $P'(x_p, y_p, z_p)$ on the view plane along the projection line PR. (see the picture given below.



For convenience, we shall represent PR in parametric form as $P(t) = (x_t,\ y_t,\ z_t)$, where t varies from 0 to 1 and

$$P(0) = (x,\ y,\ z),\ P(1) = (0,\ 0,\ z_{ref}).$$

The equations that satisfy the above conditions are

$$x_t = x(1-t)$$

$$y_t = y(1-t)$$

$$z_t = z(1-t) + z_{ref}\ t.$$

Clearly $z_p = z_{view}$. Hence the value of t for which $z_t = z_p$ would also determine x_p and y_p. Therefore,

$$z_{view} = z(1-t) + z_{ref}t \Rightarrow t = \frac{z_{view} - z}{z_{ref} - z}.$$

Then $x_p = x\left(1 - \frac{z_{view} - z}{z_{ref} - z}\right) = x\left(\frac{z_{ref} - z_{view}}{z_{ref} - z}\right)$.

If we define $d = z_{ref} - z_{view}$, which represent the distance between the view plane and the projection reference point, then we have

$$x_p = \frac{xd}{z_{ref} - z},\ \text{and}\ y_p = \frac{yd}{z_{ref} - z}.$$

Further, if we define homogeneous factor h as $h = \frac{z_{ref} - z}{d}$, then the homogenous coordinate representation for the projection transformation can be written as

$$\begin{pmatrix} hx_p \\ hy_p \\ hz_p \\ h \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{z_{view}}{d} & \frac{z_{view}.z_{ref}}{d} \\ 0 & 0 & -\frac{1}{d} & \frac{z_{ref}}{d} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

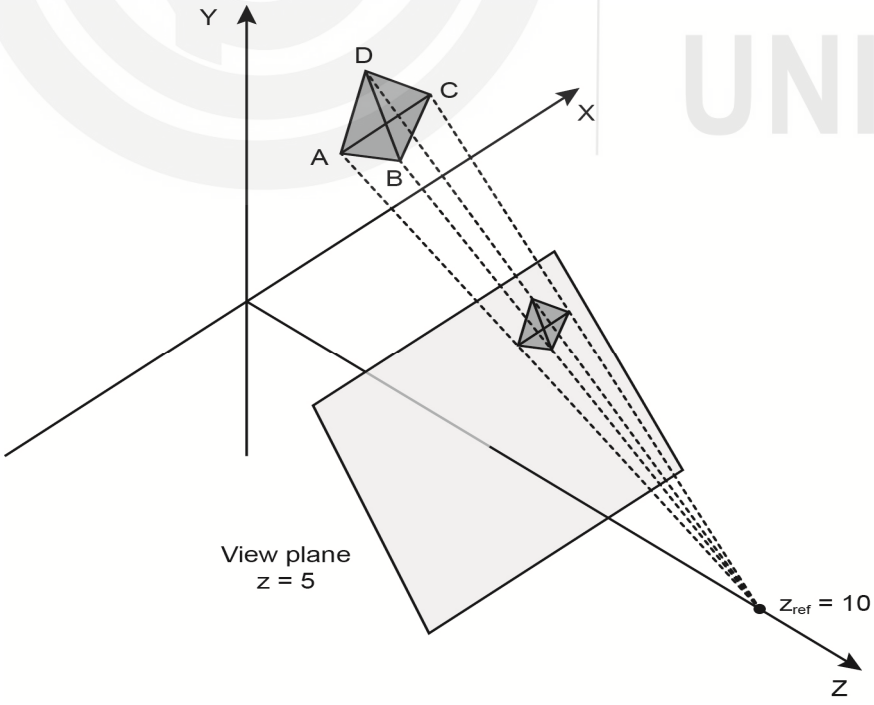
Then $x_p = \frac{x}{h},\ y_p = \frac{y}{h},\ z_p = z_{view}.$

There are two special cases that commonly arise. In the first case $z_{view} = 0,$ i.e., the view plane is the X_vY_v-plane. In this case, the projection coordinates simply reduce to $x_p = \frac{xz_{ref}}{z_{ref} - z}$ and $y_p = \frac{yz_{ref}}{z_{ref} - z}.$ The second case is $z_{ref} = 0,$ i.e., the projection reference point is at the origin. In this case, we get

$x_p = \frac{xz_{view}}{z},\ y_p = \frac{yz_{view}}{z}.$

Let us look at an example.

Example 2: Obtain the perspective projection of the pyramid $ABCD,$ where $A = (1, 1, 0),\ B = \left(\frac{3}{2}, 1, 1\right),\ C = (2, 1, 0)$ and $D = \left(\frac{3}{2}, \frac{3}{2}, 1\right),$ the view plane is parallel to the XY-plane lying 5 units apart the origin and the projection reference point is $(0, 0, 10).$ (See the picture below.)



Solution: Here we take $z_{view} = 5,\ z_{ref} = 10$ and so $d = 5.$ Then the projection coordinates are

$$x_p = \frac{5x}{10-z},\ y_p = \frac{5y}{10-z},\ z_p = 5.$$

Using these formulas, we find the projection coordinates for A, B, C, and D, respectively as follows.

$$A'(x_p, y_p, z_p) = \left(\frac{5}{10}, \frac{5}{10}, 5\right) = \left(\frac{1}{2}, \frac{1}{2}, 5\right)$$

$$B'(x_p, y_p, z_p) = \left(\frac{5\times\frac{3}{2}}{9}, \frac{5\times 1}{9}, 5\right) = \left(\frac{5}{6}, \frac{5}{9}, 5\right)$$

$$C'(x_p, y_p, z_p) = \left(\frac{10}{10}, \frac{5}{10}, 5\right) = \left(1, \frac{1}{2}, 5\right)$$

$$D'(x_p, y_p, z_p) = \left(\frac{5\times\frac{3}{2}}{9}, \frac{5\times\frac{3}{2}}{9}, 5\right) = \left(\frac{5}{6}, \frac{5}{6}, 5\right).$$

***

You may now try the following exercises.

E2) Distinguish between parallel and perspective projection.

E3) Find out the matrix of projection for cavalier projection.

E4) Find the perspective projection of the cube $ABCDEFGH$ on the XY-plane with projection reference point at $(0, 0, 4)$, where

$$A = \left(-\frac{1}{2}, -\frac{1}{2}, -1\right),\ B = \left(\frac{1}{2}, -\frac{1}{2}, -1\right),\ C = \left(\frac{1}{2}, \frac{1}{2}, -1\right),$$

$$D = \left(-\frac{1}{2}, \frac{1}{2}, -1\right),\ E = \left(-\frac{1}{2}, -\frac{1}{2}, -2\right),\ F = \left(\frac{1}{2}, -\frac{1}{2}, -2\right),$$

$$G = \left(\frac{1}{2}, \frac{1}{2}, -2\right),\ H = \left(-\frac{1}{2}, \frac{1}{2}, -2\right).$$

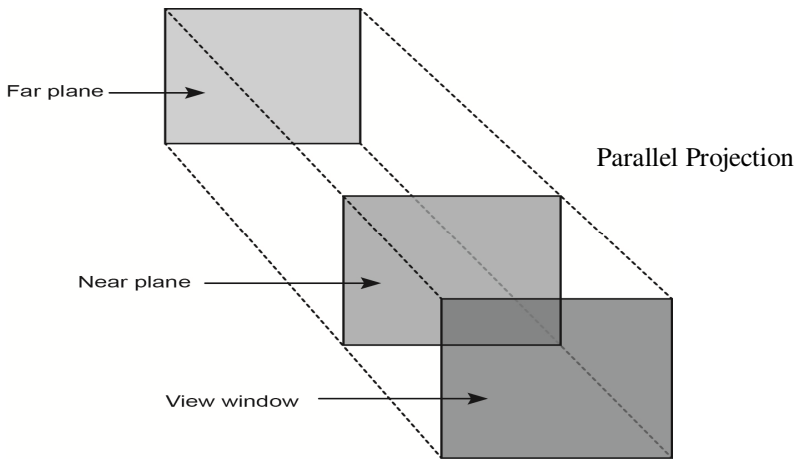
When you watch outside a window, you can view more if you are closer to the window (wide angle view) as compared to the visible volume when you are away from window. This is equivalent to saying that you are able to capture more volume of the scene when you have a wide angle lens in your camera. A view volume which we shall discuss in the next section is the spatial extent that is visible through a window in the view plane. Given the view window specification, you can set up a view volume using the window boundary. The

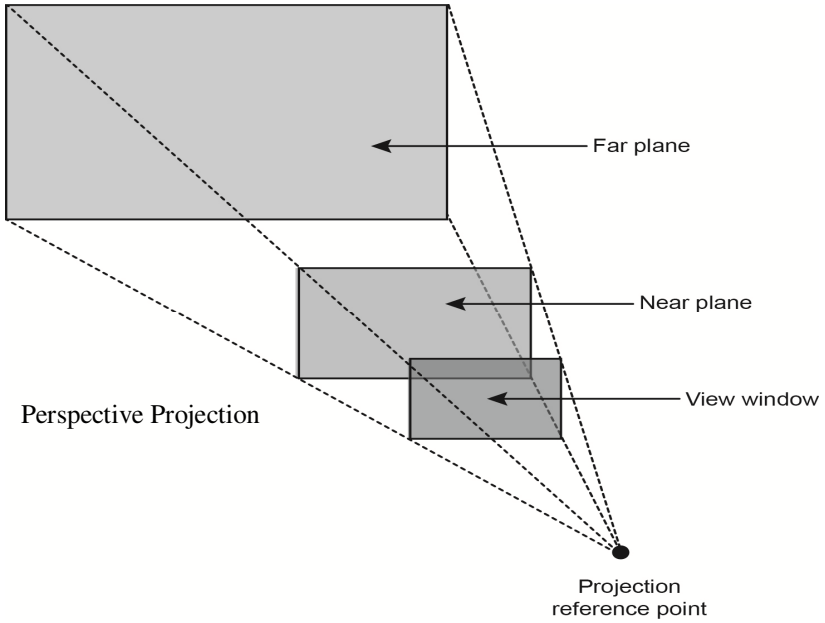
view volume is dependent on the window shape and size and the projection specifications.

7.4 VIEW VOLUMES AND GENERAL PROJECTION TRANSFORMATIONS

By now you must have understood what is a view plane and how its position or orientation affects the view of an object. View plane actually shows the entire view of an object visible from a given point or the projection reference point. To capature a particular portion of the object we select a rectangular window on the view plane, whose sides are parallel to the X_v and Y_v-axes. The view window clips the portions of the scene that lie entirely to the left, to the right, above the top or below the bottom edge. However, as you must know a 3D scene contains portions that may well be within the view window boundary yet far apart or too close to the eye, such portions must also be clipped. This can be done by specifying two planes parallel to the view plane, and we call them **far plane** and **near plane**. These planes are chosen in such a way that both of them lie to the same side of the projection reference point. Further, the near plane is nearer to the projection reference point than the far plane. Thus the four planes that pass through the view window edges along with the near and far planes comprise the **view volume** of the scene to be displayed.

In case of a parallel projection view volume is a parallelepiped (box) which can be rectangular or oblique depending upon the type of the projection. On the other hand, for a perspective projection the view volume is a frustum obtained by truncating the infinite pyramid passing through the projection reference point and the view window. (See the following pictures.)





View volume depends on the type of projection and the shape and size of the window. OpenGL provides following functions for performing parallel and perspective projections.

For orthographic parallel projection:

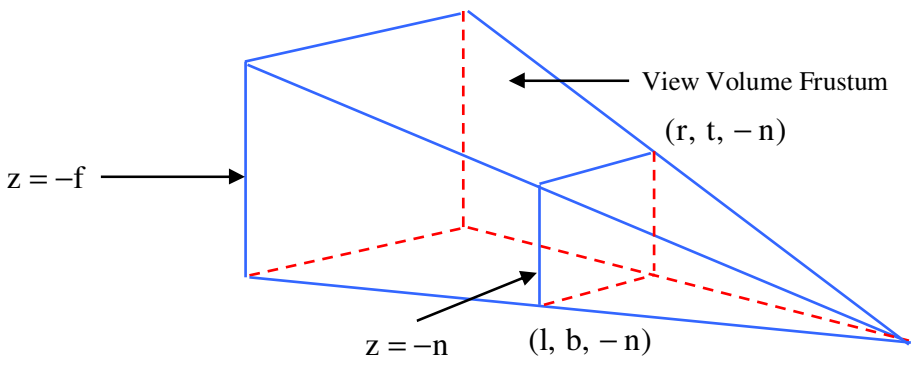
```
glOrtho(left, right, bottom, top, near, far);
glOrtho2D(left, right, bottom, top);
```

Here left, right define the x-direction extents, bottom, top define y-direction extents and near, far define the z-direction extents of the view volume. `gluOrtho2D` is exactly like `glOrtho`, but with near = –1 and far = 1. That is, gluOrtho2D views points that have *z*-value between –1 and 1. Usually, gluOrtho2D is used when drawing two-dimensional figures that lie in the xy-plane, with $z = 0$.

For perspective projection:

```
glFrustum(float l, float r, float b, float t, float n,
float f );
```

The quantities l, r specify the left and right planes; b, t specify the bottom and top planes, and n, f specify the near and far planes. (See the following picture.) The points $(l, b, -n)$ and $(r, t, -n)$ represent the bottom-left corner and the top right corner of the near plane, respectively.



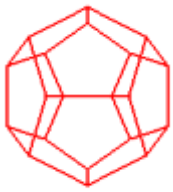
Another way to specify the view volume under the perspective projection is using the following command.

```
gluPerspective( float θ, float aspect, float n, float f );
```

Here θ is an angle (measured in degrees) specifying the vertical field of view (θ is the solid angle between the top bounding plane and the bottom bounding plane of the frustum). The parameter *aspect* specifies the ratio of the width of the frustum to the height of the frustum.

Following simple program will give you an idea about how to use the forgoing functions in a C program for displaying images of 3D objects. The program displays a dodecahedron using orthographic projection.

```
//program to draw a wireframe model of dodecahedron
#include <GL/glut.h>
GLint w=500,h=500;

void init(void)
{
    glClearColor(1, 1, 1, 1);
    glShadeModel(GL_FLAT);
}
void display(void)
{

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1, 0, 0);
    glLoadIdentity();
    gluLookAt(0, 0, 5, 0, 0, 0, 0, 1, 0);
    glScalef(0.8, 0.8, 0.8);
    glutWireDodecahedron();  //draws a wireframe dodecahedron
    glFlush();
}

void reshape(GLint w, GLint h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-1, 1, -1, 1, 1, 20);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowPosition(0, 0);
    glutInitWindowSize(w,h);
    glutCreateWindow("3D View");
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```



Listing 1: Drawing a wireframe model of Dodecahedron

The output will be as shown in the margin.

A 3D program requires much more than these settings. For more information on 3D graphics programming you may refer the coding resources and tutorials available on the offical website www.opengl.org of OpenGL.

You may now try the following exercises.

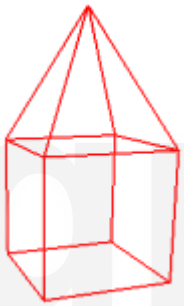
E5) State whether the following statements are true or false. Give reasons/examples to justify your answer.

a) Cavalier projection is a parallel projection.

b) Angles are not preserved in perspective projection in general.

c) Perspective projection is an affine transformation.

E6) Write a program to draw a model as shown below.



Modify your program to get different orthographic views of the model. For instance, show how it looks from the top or from the front.

We now end the unit by giving a summary of what we have covered in it.

7.5 SUMMARY

In this unit, we have covered the following:

1. We have seen how to project a 3D image on a 2D device, and we have discussed that there are two types of projections – parallel projection and perspective projection.

2. In parallel projection, objects in scene are projected onto the 2D view plane along rays parallel to a projection vector.

3. Parallel projection is orthographic if the projection vector is orthogonal to the view plane. Otherwise, it is called oblique parallel projection. Farther, we have discussed cavalier and cabinet projections.

4. Perspective projection gives more realistic appearance and uses the same principle as used in camera.

5. OpenGL functions for modelling and viewing transformations are:

```
1. glMatrixMode(GL_MODELVIEW );
2. gluLookAt (eyeX, eyeY, eyeZ, atX, atY, atZ, upX, upY,
   upZ);
```

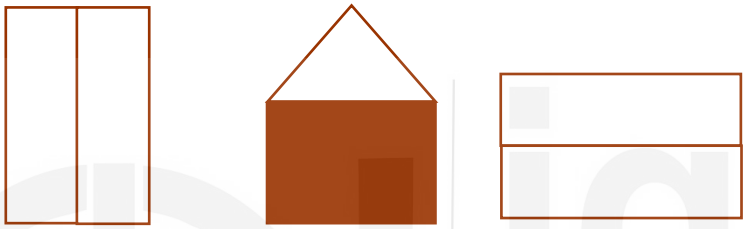
6. OpenGL functions for parallel and perspective projections.

- `glOrtho (left, right, bottom, top, near, far)`
- `glFrustum(float l, float r, float b, float t, float n, float f );`
- `gluPerspective( float θ, float aspect, float n, float f );`

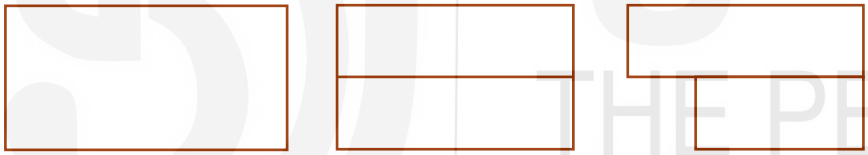
7.6 SOLUTIONS/ANSWERS

E1) The top view, front view and the side views of the objects in each case are as follows.

(a)



(b)



E2) The differences are as follows.

Parallel Projection	**Perspective projection**
Coordinate positions are transformed to view plane along parallel lines. The vector defining the direction of lines is called **projection vector**.	Coordinate positions are transformed to view plane along lines that converge to a point called **projection reference point** .
Object distance from the view plane is not significant.	Object's distance from the projection reference point and view plane is important. It plays a role in determining size of the image. Farther the object, smaller is the image.
Parallel lines remain parallel.	Parallel lines may not remain parallel.
Ratios are preserved in general.	Ratios are not preserved in general.
Useful in civil, architectural design, machine design, sectional view design, etc.	Useful in virtual reality, gaming and other such applications.
Less realistic.	More realistic.

E3) Cavalier projection corresponds to $\tan\alpha = 1$, leading to the following matrix.

$$M_{parallel} = \begin{bmatrix} 1 & 0 & cos\,\phi & 0 \\ 0 & 1 & sin\,\phi & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

E4) Here $Z_{ref} = 4$ and the view plane is the XY-plane. Therefore

$$x_p = \frac{x\ Z_{ref}}{Z_{ref} - Z} = \frac{4x}{4 - z},\ y_p = \frac{y\ Z_{ref}}{Z_{ref} - Z} = \frac{4y}{4 - Z}$$

The projected points are:

$$A'\left(-\frac{2}{3}, -\frac{2}{5}\right),\ B'\left(\frac{2}{5}, -\frac{2}{5}\right),\ C'\left(\frac{2}{5}, \frac{2}{5}\right),\ D'\left(-\frac{2}{5}, \frac{2}{5}\right),$$
$$E'\left(-\frac{1}{2}, -\frac{1}{3}\right),\ F'\left(\frac{1}{3}, -\frac{1}{3}\right),\ G'\left(\frac{1}{3}, \frac{1}{3}\right),\ H'\left(-\frac{1}{3}, \frac{1}{3}\right).$$

E5) a) True. It is a type of oblique parallel projection when $\tan\alpha = 1$.

b) True. Parallel lines appear converging to a point.

c) False. The perspective projection contains z coordinate in the denominator. Hence it cannot be an affine transformation.

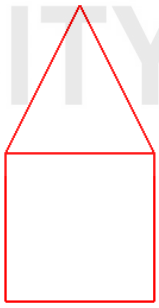
d) False. A vanishing point for any set of lines that are parallel to one of the principal coordinate axes of object is called a principal vanishing point. So there can be 1, 2 or 3 principal vanishing points.

E6) The required program is given below.

```
#include <GL/glut.h>
GLint w=500,h=500;

void draw_model(void)
{
  glBegin(GL_LINE_STRIP);
        glVertex3d(0,0,0);
        glVertex3d(0,0,1);
        glVertex3d(1,0,1);
        glVertex3d(1,0,0);
        glVertex3d(0,0,0);
        glVertex3d(0,1,0);
        glVertex3d(0,1,1);
        glVertex3d(1,1,1);
        glVertex3d(1,1,0);
        glVertex3d(0,1,0);
        glVertex3d(0.5,2,.5);
        glVertex3d(0,1,1);
        glVertex3d(0,0,1);
    glEnd();
    glBegin(GL_LINE_STRIP);
        glVertex3d(1,0,0);
        glVertex3d(1,1,0);
        glVertex3d(0.5, 2, 0.5);
        glVertex3d(1,1,1);
```

```
            glVertex3d(1,0,1);
        glEnd();
        glFlush();
    }
    void display(void)
    {
        glClearColor(1, 1, 1, 0);
        glClear(GL_COLOR_BUFFER_BIT);
        glColor3f(1, 0, 0);
        glLoadIdentity();
        gluLookAt(-2, 3, 8, 0.5, 1, 0.5, 0, 1, 0);
        glScalef(2, 2, 2);
        draw_model();
    }

    void reshape(GLint w, GLint h)
    {
        glViewport(0, 0, w, h);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(60, 1, 1, 20);
        glMatrixMode(GL_MODELVIEW);
    }

    int main(int argc, char** argv)
    {
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
        glutInitWindowPosition(0, 0);
        glutInitWindowSize(w,h);
        glutCreateWindow("3D Perspective View");
        glutDisplayFunc(display);
        glutReshapeFunc(reshape);
        glutMainLoop();
        return 0;
    }
```

To get the orthographic parallel projection of the model, we use glOrtho() routine instead of `gluPerspective`(). Also, in this case we do not need the `gluLookAt()` routine. For instance, to produce the front view as shown in the right margin, we modify the relevant code as follows.



Front view

```
void display(void)
{
    glClearColor(1, 1, 1, 0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1, 0, 0);
    glLineWidth(2);
    glLoadIdentity();
    glScalef(0.5, 0.5, 0.5);
    //glRotated(90, 1, 0, 0);
    draw_model();
}

void reshape(GLint w, GLint h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1, 2, -1, 2, 2, -1);
    glMatrixMode(GL_PROJECTION);
}
```

Top view

The top view of the model is also shown in the right margin. To get the top view just remove the comment before the `glRotate()` command given in the above code, which rotates the object about the x-axis by an angle of 90°. Likewise you can try rotating the object about other axes to get more views. Observe that if you rotate the object by -90° about the x-axis, you would get the same view as the top view.

APPENDIX A

OPENGL–A QUICK START

Structure **Page No.**

1.1 INTRODUCTION

OpenGL stands for **Open Graphics Library**. By using it, you can create interactive applications that render high-quality color images composed of 3D geometric objects and images. Here the term 'Open' indicates open standards, which means that many companies are able to contribute to the development of OpenGL.

OpenGL is a powerful low-level graphics rendering and imaging library available on all major platforms. It provides the programmers an interface to the systems graphics hardware. OpenGL is designed to be a device which is window and operating system independent and allows portability between various computer platforms (refer Sec. 1.2, about the portability of your program). It is referred to as an API (application programming interface), that insulates the programmer from device differences and how they vary from one system to another.

OpenGL was originally **developed by Silicon Graphics Inc. (SGI)** as a multipurpose, platform independent graphics API. Since 1992, OpenGL

Architecture Review Board (ARB) has been looking after the development of OpenGL. This board consists of some of the major graphics vendors and other industry leaders such as Hewlett-Packard, IBM, NVIDIA, Sun Microsystems and Silicon Graphics etc. In view of the fast development of graphics hardware, ARB is committed to annual updates and the present version of OpenGL is 3.0. OpenGL is now widely used in CAD, virtual reality, scientific visualization, flight simulation and video games.

Purpose of this Appendix is to give you a quick exposure to graphics programming in C using OpenGL and make you acquainted with basic structure of a graphics program using OpenGL. We do not intend to give here a complete introduction to OpenGL as we have already incorporated the required details in the units at various places.

Objectives

After reading this supplement, you should be able to

- outline the basics of OpenGL;
- draw 2D objects with specified fill colors;
- use OpenGL transformation function for 2D and 3D geometric transformations;
- use simple animation to objects to be displayed in the scene;
- write interactive programs using mouse and keyboard functions of OpenGL.

1.2 A Brief Overview

As you know OpenGL is a 3D graphics application programmers interface (API) to produce high-quality, color images of 3D objects (group of geometric primitives) and images (bitmaps and raster rectangles). Remember that it is not a programming language like C or Java. To give you an idea of what is primarily possible with OpenGL, we list here a few basic operations that you can perform with the help of OpenGL.

- Creating interactive applications which render high-quality color images composed of 3D geometric objects and images.
- Specification (modelling) of an arbitrarily complex set of objects in 3D space - creation of a 3D virtual world. This includes
  - Object specifications --- In terms of drawing primitives.
  - Relative positions of multiple objects defined by transformations.
  - Coloring objects, specifying light sources and light direction.
  - Specification of a virtual camera by which to view the 3D virtual world.

- Projection to 2D display.

You will explore many more features of OpenGL later in this appendix.

When a program is executed, OpenGL does the following:

- Assembles the virtual world.
- Points the virtual camera at the world. This means setting up a view angle or direction and the volume or a frustum which you would like to keep in your scene.
- Projects the scene onto a 2D projection plane
- Performs the equivalent of spatial sampling and rasterization.

To be more specific on the **portability of your program**, part of your application which does rendering is platform independent. However, in order for OpenGL to be able to render, it needs a window to draw into. OpenGL is not capable of opening windows or responding to interrupts from a mouse or keyboard. Generally, this is controlled by the windowing system on whatever platform you are working on. As OpenGL is platform independent, we need some way to integrate OpenGL into each windowing system. Every windowing system where OpenGL is supported has additional API calls for managing OpenGL windows, color maps, and other features. These additional APIs are platform dependent. For the sake of simplicity, we will use an additional freeware library for simplifying interaction with windowing systems. This is called **GLUT (GL Utility Toolkit).** GLUT is a library which makes writing of OpenGL programs, regardless of windowing systems, much easier. The GLUT libraries provide facilities to define and open windows and respond to mouse and keyboard functions.

Finally we conclude this section by telling you something about the **rendering pipeline** of OpenGL. OpenGL Pipeline has a series of processing stages in order (see Fig.1). Two graphical information, vertex-based data and pixel-based data, are processed through the pipeline, combined together and then written into the frame buffer. Notice that OpenGL can send the processed data back to your application.

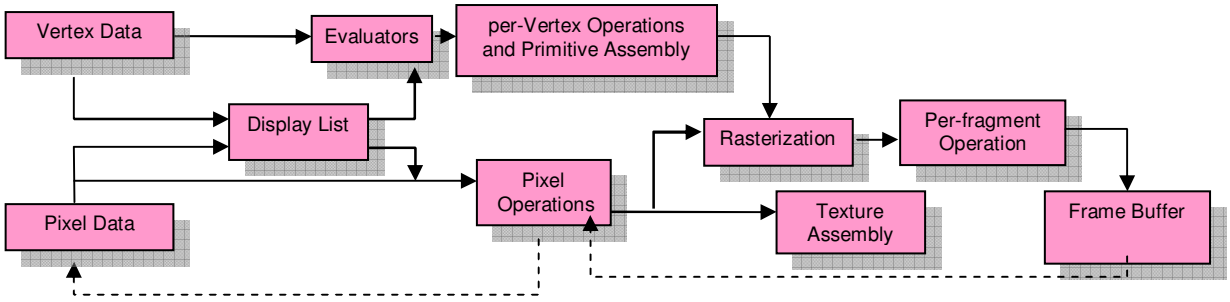


Fig. 1.

Display List: Because of its structure, OpenGL require lots of procedure calls to render a complex object. To improve efficiency, OpenGL allows you to generate an object, called a display list, into which OpenGL commands are stored in an efficient internal format, which can be called back again in the future.

Vertex Operations: Vertex and normal coordinates are transformed by GL_MODELVIEW matrix (from object coordinates to eye coordinates). Also, if lighting is enabled, the lighting calculation per vertex is performed using the transformed vertex and normal data. This lighting calculation updates new color of the vertex.

Pixel Transfer Operations**: After the pixels from client's memory are unpacked (read), scaling, bias, mapping and clamping are performed on the data. These operations are called Pixel Operations. The transferred data are either stored in texture memory or rasterized directly to fragments.**

Primitive Assembly**: After vertex operation, the primitives (point, line, and polygon) are transformed once again by projection matrix. These primitives are clipped against viewing volume clipping planes, changing from eye coordinates to clip coordinates. After that, perspective division by w occurs and viewport transform is applied in order to map 3D scene to window space coordinates. Last thing to do in Primitive Assembly is culling test if culling is enabled.**

Texture Memory: Texture images are loaded into texture memory to be applied onto geometric objects.

Rasterization: Rasterization is the conversion of both geometric and pixel data into fragment. Fragments are a rectangular array containing color, depth, line width, point size and antialiasing calculations.

Fragment Operation: It is the last process to convert fragments to pixels onto frame buffer. The first process in this stage is texture generation. A texture element is generated from texture memory and it is applied to the each fragment. Then fog calculations are applied. After this a variety of different fragment tests are applied. Finally, blending, dithering, logical operation and a few more operations are performed and actual pixel data are stored in frame buffer.

OpenGL as a State Machine

OpenGL is a state machine. You put it into various states (or modes) that remain in effect until you change them. For example, the current color is a state variable. Suppose you set the current color to white. Every object is then

drawn with white color until you set the current color to something else. There are many such state variables that OpenGL maintains. Another example that you might be using in your codes quite often would be point size. You can take it to be one pixel thick or it can be thicker with more pixels.

1.3 SETTING UP OPENGL IN CODE::BLOCKS

One can use any language and any environment for writing and executing computer graphics programs. We shall use C language, which you are already familiar with, from your first course MMT-001. Further, we advise you to use Code::Blocks for this purpose, which is an integrated development environment for C/C++. It is available on the following website:

https://www.codeblocks.org

Note that OpenGL comes with every modern computer by default. What you need to install are some graphics libraries for making your C program executable. We shall show here how to install these libraries in Code::Blocks for both Linux as well as Windows operating systems.

Install OpenGL on Ubuntu
For installing OpenGL on Ubuntu, just execute the following command (like installing any other thing) in terminal:

```
sudo apt-get install freeglut3-dev
```

For working on Ubuntu operating system:

```
gcc filename.c -lGL -lGLU -lglut
```

where filename.c is the name of the file with which this program is saved. To compile the program you need to give the following command.

```
gcc filename.c -lGL -lGLU -lglut -lm
```

Install OpenGL on windows in Code::Blocks

- Go to the website, transmissionzero.co.uk, and download the latest version of freeglut for MinGW. Extract the files in the folder FreeGLUT, say.
- Open notepad with **run as administrator**
  1. Open file from

     `C/Program Files/CodeBlocks/share/CodeBlocks/templates`,

     then click to show All Files.

  2. Next, open **glut.cbp** and search all **glut32** and replace with **freeglut**.

  3. Then, open

     `C/Program Files/CodeBlocks/share/CodeBlocks/templates/wizard/glut`

and, then click to show All Files.

4. Open wizard.script and here, also replace all **glut32** with **freeglut**

- Then go to the folder FreeGLUT, and do the following:
  1. Copy all files form `FreeGLUT/include/GL`, and patse them into `C/…/MinGW/include/GL`.
  2. Copy the two files from FreeGLUT/lib, and paste them into `C/ …/MinGW/lib`.
  3. Copy the file freeglut.dll from FreeGLUT/bin, and paste it into `C/Windows/SysWOW64`.
- Now open Code::Blocks, and go to
  1. File → New → Project → GLUT project → Next.
  2. Give project title anything, and then choose Next.
  3. For selecting GLUT's location : C/…/MinGW.
  4. Press OK → Next → Finish.

You may also refer the YouTube link as given below.

https://www.youtube.com/watch?v=NPcnymtP2SE

1.4 YOUR FIRST PROGRAM USING OPENGL

Let us begin by a simple C program which illustrates how you open a window and specify its position and size. You need to structure your program as follows:

- **Initializing the window:** Call the following function to initialize the GLUT library.

```
glutInit(&argc, argv);
```

Its arguments should be the same as that of your `main()` function.

Now choose the type of window that you need for your application and initialize it. This is done by the function

```
glutInitDisplayMode()
```

which takes an argument of the form $x \mid y$, where | is the logical OR operator. The arguments x and y could be the built-in constants GLUT_SINGLE and GLUT_RGB, respectively, which specify that a single display buffer is to be used along with the desired amount of red, green and blue colours. Other options are also available. For example, you may use GLUT_DOUBLE for using a double buffer. In animation programs, it is advisable to use double buffer.

The function

```
glutInitWindowPosition (x, y);
```

specifies the window position at (x,y) pixel location. This means, the window is x pixels to the right and y pixels down the top left corner of the display screen.

```
glutInitWindowSize (w, h);
```

specifies the window size to be h units high and w units wide. You may choose height and width as per your program's specifications and object size.

```
glutCreateWindow("A Rectangle");
```

creates an OpenGL enabled window.

- **State Initialization**: Initialize any OpenGL state that you don't want to change in every frame of your program. This might include many states such as background color, positions of light sources, texture maps etc. For example, you would need to specify the RGB component of background color to be used when clearing the color buffer using the following function.

```
glClearColor(R, G, B, alpha)
```

where `R, G, B` and alpha are floating point parameters ranging from 0 to 1. Here `R` stands for **red**, `G` for **green** and `B` for **blue**. The parameter `alpha` indicates transparency. The function

```
glClear(GL_COLOR_BUFFER_BIT);
```

clears the window with the current clearing colour.

- **Register Callback Functions**: A callback function is an executable code that is passed as an argument to other code. You need to register the callback functions that you will need in you main program. Then GLUT calls these functions when a certain sequence of events occur, like a user input through mouse or the window needing to be refreshed.

  The most important callback function is the following which renders your scene.

```
glutDisplayFunc(display);
```

  Here `display()` is the function you will write to create your objects. The program `Blank.c` displays nothing, except the blank window. This is

because your display function does not involve any display function. Use of

```
glFlush() ;
```

flushes every OpenGL command and buffers for display.

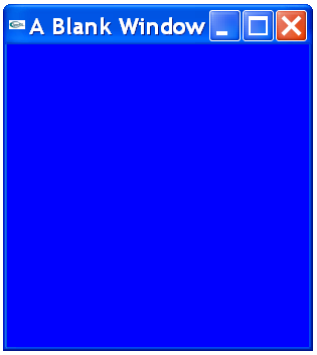
- **Enter the main event processing loop**: This is where your application receives events, and starts the GLUT main processing loop.

```
glutMainLoop();
```

Let us now have a look at the complete C-code.

```
//Blank.c

#include<stdio.h>
#include<GL/glut.h>
// functions to initialize
void myInit (void)
{
    gluOrtho2D(-500, 500, -500, 500);    // window dimension
    glClearColor(0, 0, 1, 1);      //window colour
}
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
}
int main (int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

    glutInitWindowSize(500, 500);          // window size
    glutInitWindowPosition(0, 0);          // window position
    glutCreateWindow("A Blank Window");  // window name

    myInit();
    glutDisplayFunc(display);

    glutMainLoop();
}
```

What you will get as an output will be the following.

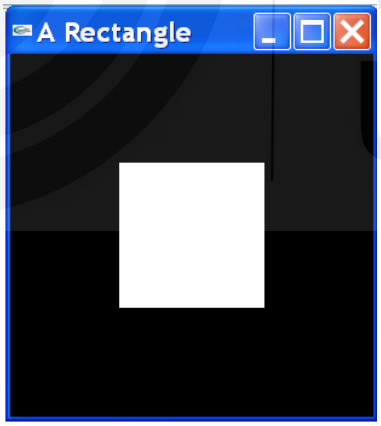


Let us now modify our program to draw a square in the window. This time set the background color to black. Color of the solid square by default will be

white. Note that there is a very little change in your display function and in initialization.

```
//Blank.c

#include<stdio.h>
#include<GL/glut.h>
// functions to initialize
void myInit (void)
{
    gluOrtho2D(-500, 500, -500, 500);    // window dimension
    glClearColor(0, 0, 1, 1);      //window colour
}
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glRectf(150, 150, 350, 350);
    glFlush();
}
int main (int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

    glutInitWindowSize(500, 500);          // window size
    glutInitWindowPosition(0, 0);          // window position
    glutCreateWindow("A Blank Window");   // window name

    myInit();
    glutDisplayFunc(display);

    glutMainLoop();
}
```

The output looks as follows.



For plotting points you can use the following code and add it to your display function.

```
glBegin(GL_POINTS);
glVertex3f(100, 150, 150);
glVertex3f(200, 150, 200);
glEnd() ;
```

There are a number of primitives for object modelling and rendering the scene. You are advised to visit the official website of OpenGL to see the reference manual for more information on drawing and other primitives.

Window Reshape Function: Every program displaying its output on window uses a reshape function that indicates what action is to be taken when the window is resized. This function has the following syntax.

```
glutReshapeFunc(void (* func)(int w, int h));
```

The function allows registration of a callback function that has to be called when the window is resized. As an example of a callback function, we give the following function `reshape( )`.

```
void reshape( GLsizei w, GLsizei h)
{
   if(h==0) h = 1;
   glViewport(0,0,w,h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if( w<=h)
      glOrtho( 0, 250, 0, 250*h/w, 1, -1);
   else
      glOrtho( 0, 250*w/h, 0, 250, 1, -1);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
```

Then register the callback function

```
glutReshapeFunc(reshape);
```

in your main function to get the desired effect.

1.5 OPENGL FUNCTIONS, CONSTANTS, AND DATA TYPES

Naming Conventions

Function names in the basic OpenGL library begin with the letters gl, and each component word within a name has its first letter capitalized. For example- glClear(), glBegin(). Fig. 2 for the glVertex*( ) function on the next page may help you understand better the general naming conventions in gl and glut libraries.

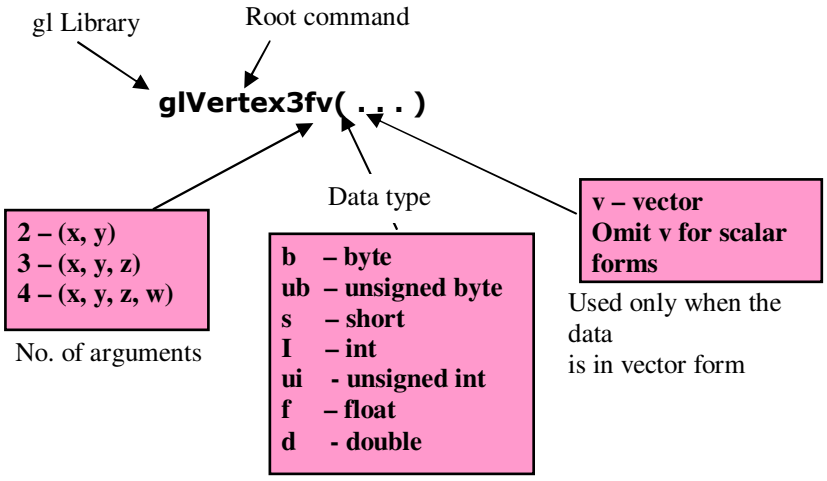


Fig. 2.

There are certain functions which require arguments that are constant specifying a parameter name, a value for a parameter, a particular mode, etc. All such constants begin with the letters GL, component words in the constant are written in capital letters, and the underscore " _ " is used as a separator between the words. For example, we have used GL_RGB and GL_COLOR_BUFFER_BIT in our earlier program.

For making the program platform independent and for fast processing of various routines, OpenGL has defined its own data types which are as given in Table-1 below.

Table-1

Suffix	Data Type	C/C++ type	OpenGL type name
b	8-bit int	signed char	GLbyte
s	16-bit int	short	GLshort
i	32-bit int	int or long	GLint, GLsizei
f	32-bit float	float	GLfloat, GLclampf
d	64-bit float	double	GLdouble, GLclampd
ub	8-bit unsigned no.	unsigned char	GLubyte, GLboolean
us	16-bit unsigned no.	unsigned short	GLushort
ui	32-bit unsigned number	unsigned int or unsigned short	GLuint, GLenum, GLbitfield

In addition to the basic, or core, library of functions, a set of "macro" routines that use core functions are available in the OpenGL Utility Library (GLU). These routines provide methods for setting up viewing projection matrices, describing complex objects with line and polygon approximations, surface rendering, and other complex tasks. In particular, GLU provides methods for displaying quadrics using linear-equation approximations.

OpenGL Colours and Display Modes

OpenGL colours are typically defined using RGB (red, green, blue) components and a special A (or alpha) component. There are various ways in which you could interpret A depending on the context (especially with reference to transparency and colour blending). Usually A = 1 when no special effects are desired. We have used the function `glutInitDisplayMode(unsigned int mode)` for setting up display

mode. The mode is the logical-OR denoted by "`x | y`" with various choices of `x` and `y` from the following:

GLUT_RGB for RGB colours, GLUT_RGBA for RGBA colours and GLUT_INDEX for using colour-mapped colours (not recommended) and GLUT_SINGLE for single-buffering, GLUT_DOUBLE to allow double-buffering (for smooth animation). Further you can use GLUT_DEPTH for depth buffering for hidden surface removal.

Finally a brief discussion about the projection matrices. Fig. 3 gives you an idea about the pipeline of transformations that you need to apply to render a 3D scene. Remember that OpenGL is a 3D graphics standard and treats every object as a 3D object. If the object is 2D, you can simply define that as a 3D object sitting in the plane z = 0.

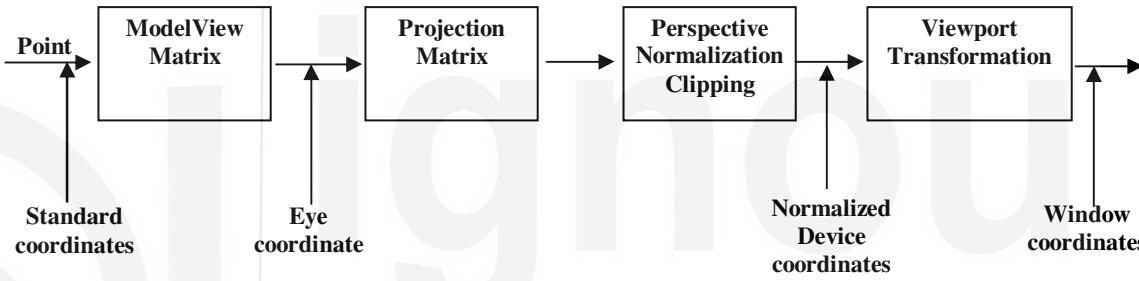


Fig. 3.

When you study 3D geometric and projection transformations in your Computer Graphics course, you will be able to appreciate better this pipeline. Because of this pipeline, you need to introduce the following callback functions in your program.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, 500, 0, 500, 1, -1);
```

For more details, you are advised to refer to Unit 7.

Some Output Primitives

You can use some of the following output primitives of OpenGL to practice more on OpenGL programming.

Assume there are n points inside `glBegin() ... glEnd()` given in a sequence as $P_1, P_2, ..., P_n$.

Argument	Purpose
GL_POINTS	Individual points drawn as $P_1, P_2, ..., P_n$
GL_LINES	Line segments with endpoints $(P_1, P_2), .(P_3, P_4), .., (P_{n-1}, P_n)$. The last point is ignored if n is odd.

GL_LINE_LOOP	Polygon with vertices $P_1, P_2, ..., P_n$
GL_TRIANGLES	Triangles with vertices as $(P_1, P_2, .P_3), (P_4, P_5, P_6)$, ... The leftover vertices are ignored if n is not a multiple of 3.
GL_TRIANGLE_STRIP	Triangles with vertices as $(P_1, P_2, P_3), (P_2, P_3, P_4), (P_3, P_4, P_5), ...$
GL_TRIANGLE_FAN	Triangles with vertices as $(P_1, P_2, P_3), (P_1, P_3, P_4), (P_1, P_4, P_5)$, ...
GL_QUADS	Quadrilaterals with vertices as $(P_1, P_2, P_3, P_4), (P_5, P_6, P_7, P_8), ...$ The leftover vertices are ignored if n is not a multiple of 4.
GL_QUAD_STRIP	Quadrilaterals with vertices as $(P_1, P_2, P_4, P_3), (P_3, P_4, P_6, P_5), (P_5, P_6, P_8, P_7), ...$
GL_POLYGON	

For instance, if you wish to draw a triangle with vertices (0,0), (4,5) and (8,1), then the required code will be :

```
glBegin(GL_TRIANGLES);
      glVertex3f(0, 0);
      glVertex3f(4, 5);
      glVertex3f(8, 1);
glEnd();
```

You can refer to the OpenGL manual for more information.

1.6 OPENGL TRANSFORMATION FUNCTIONS

As already discussed, OpenGL treats every thing as 3D. Hence all the transformations that are defined in OpenGL are 3D in nature. For using them in 2D applications, you simply need to ignore the third dimension parameter. For example, following function translates the given object by (1, 2) in 2D, since we have chosen the third parameter to be zero.

```
glTranslatef(1, 2, 0);
```

Similarly, rotation about the origin in 2D is essentially the same as 3D rotation about z-axis. Hence

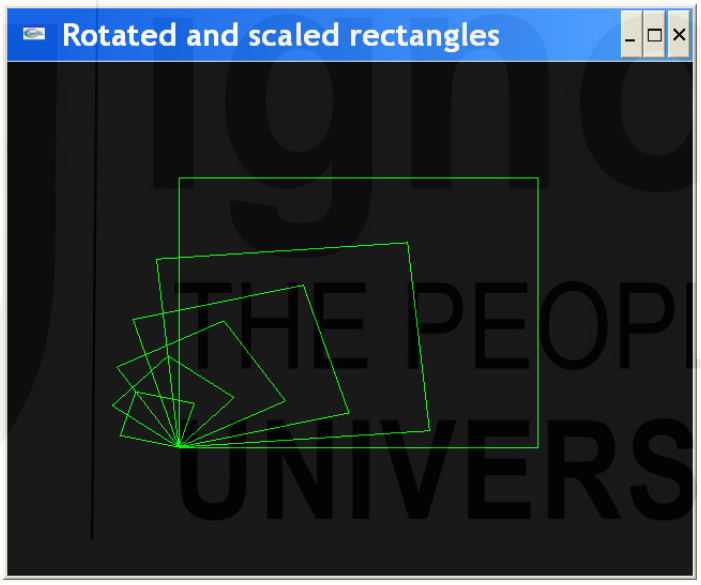
```
glRotatef(45, 0, 0, 1);
```

will rotate the object about z-axis by an angle of 45 degrees around the origin. Similarly, you can use OpenGL scaling function glScale*( ) for 2D transformations. Following example of display function will help you understand how rotation and scaling of a square have been performed with respect to a pivot point (0.5, 0.5)

```
void myDisplay ( )
```

```
{
      glClear ( GL_COLOR_BUFFER_BIT );
      glColor3f(0, 1, 0);
      for(float angle=0; angle <30; angle=angle + 5)
      {
            glTranslated(-0.5f, -0.5f, 0.0f);
            glScaled(0.7,0.7,10);
            glRotated( angle, 0.0, 0.0, 1.0);
            glTranslated(0.5f, 0.5f, 0.0f);
            glBegin(GL_LINE_LOOP);
                  glVertex3f(-0.5, -0.5, 0.0);
                  glVertex3f(1.0, -0.5, 0.0);
                  glVertex3f(1.0, 1.0, 0.0);
                  glVertex3f(-0.5, 1.0, 0.0);
            glEnd();
      }
glFlush();
}
```

To see the effect, replace the display function of one of your previous c-program code with the above and you will see the following output on the display window.



Unit 6 gives an introduction to 3D transformations with a brief description of OpenGL 3D transformations.

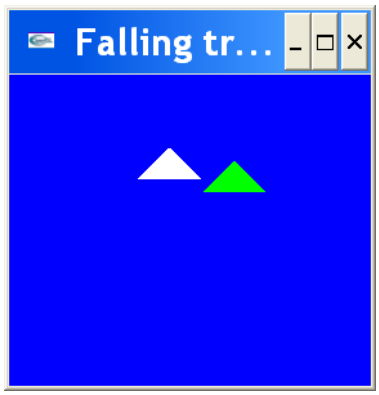
1.7 ANIMATION

OpenGL provides good support for animation design. Here we give a simple example code showing the animation for two triangles. The executed code displays falling triangles.

```
#include <gl/glut.h>
#include <math.h>

GLfloat x=100, y=100, w, h;

void myDisplay(void)
{
```

```
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1, 1, 1);
    glBegin(GL_TRIANGLES);
        glVertex2f(x, y);
        glVertex2f(x+50, y);
        glVertex2f(x+25, y+25);
    glEnd();
    glColor3f(0, 1, 0);
    glBegin(GL_TRIANGLES);
        glVertex2f(x+50, y-10);
        glVertex2f(x+100, y-10);
        glVertex2f(x+75, y+15);
    glEnd();
    glutSwapBuffers();
}

void TimerFunction(int value)
{
    //If triangles go down the window, lift them up
    if(y <  0)
        y = 250 + y;
    y=y - 5;
    //Redraw the scene with changed coordinates
    glutPostRedisplay();
    glutTimerFunc(100,TimerFunction,1);
}

void reshape(GLsizei w, GLsizei h)
{
    if(h==0) h=1;
    glViewport(0,0,w,h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    //Preserve the proportions in changing window
    if(w<=h){
        w=250*h/w;
        h=250.0f;
    }
    else
    {   w=250*w/h;
        h=250;
    }
    glOrtho(0, w, 0, h, 1, -1);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void myInit(void)
{
    glClearColor(0.0f,0.0f,1.0f,1.0f);
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
    glutCreateWindow("Falling Triangles");
    glutDisplayFunc(myDisplay);
    glutReshapeFunc(reshape);
    glutTimerFunc(100,TimerFunction,1);
    myInit();
    glutMainLoop();
    return 0;
}
```

A snapshot of the output is shown below.



A few words about the functions that we have used in making this animation.

```
void glutPostRedisplay(void) ;
```

This function tells GLUT that the window needs to be repainted with current changes. So if you need to re-draw the scene with changes, call this function.

```
void glutSwapBuffers(void) ;
```

This function flushes the OpenGL commands and swaps the buffer, in case you have given the option of double buffer (GLUT_DOUBLE) in your code. If double buffer is not used, then also OpenGL commands are flushed.

```
void glutTimerFunc(unsigned int m, (*func)(int value), int
value);
```

This function registers a callback function func that should be called after m milliseconds have elapsed. The integer value is the user specified value that is passed to the callback function. Most of the OpenGL implementations provide double-buffering. When one is being displayed, the other is getting prepared for display. When the drawing of a frame is complete, the two buffers are swapped, so that one that was being viewed now becomes available for drawing, and vice versa.

1.8 MOUSE AND KEYBOARD FUNCTIONS

In order to make your program interactive, you can effectively use the mouse and keyboard with the help of OpenGL mouse and keyboard functions. Some of the functions that are frequently used in interactive programs are as follows:

```
glutMouseFunc(mouseFunc);
```

This function allows you to link a mouse button with a function that is invoked when a mouse button is pressed or released. Here `mouseFunc` is the name of the function that is invoked on mouse event.

```
glutKeyboardFunc(keyFunc);
```

This function specifies a function that is to be executed when a particular key character is selected. This function can also return the current mouse position in window coordinates.

```
glutMotionFunc(motionFunc);
```

This function specifies a function that is to be executed when a mouse moves within the window while one or more buttons are pressed.

You can try the following piece of code to understand how these functions work.

```
//button = mouse button, action can be pressed or released
// x, y indicate mouse position

void mouseFunc( GLint button, GLint state, GLint x, GLint y)
{
if( button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
      glRectf(x, h- y, x+50, h -(y+50)); // h is window height
else
      if ( button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
            { glBegin(GL_TRIANGLES);
                  glVertex2i(x, 250 - y);
                  glVertex2i(x+50, 250 - y);
                  glVertex2i(x+25, 250 -(y+25));
              glEnd();
            }
glFlush( );
}
```

The above function is required to be registered in your main program using the following command.

```
glutMouseFunc(mouseFunc);
```

Similarly, a keyboard function example is given below.

```
 void keyFunc(GLubyte key, GLint x, GLint y)
{
GLint mouse_x = x;
GLint mouse_y = h - y; //h is the window height
switch(key)
      {
      //Plot rectangle
      case 'r':
      glRectf(mouse_x, mouse_y, mouse_x+50, mouse_y+50);
      break;
      // Plot triangle
      case 't':
      glBegin(GL_TRIANGLES);
            glVertex2i(mouse_x, mouse_y);
            glVertex2i(mouse_x+50, mouse_y);
            glVertex2i(mouse_x+25, mouse_y + 25);
```

```
        glEnd();
        break;
        default:
        break;
        }

glFlush( );
}
```

The above callback needs to be registered in your main function as follows.

```
glutKeyboardFunc(keyFunc);
```

This gives a quick introduction to OpenGL primitives for graphics support. OpenGL has a vast range of functions for modelling, rendering, animation and image processing. For detailed study on OpenGL, you are advised to refer the following:

- Dave Shreiner, Mason Woo, Jakie Neider and Tom Davis, OpenGL Programming Guide: The Official Guide to Learning OpenGL, Sixth Edition, Pearson Education, 2008.

- www.opengl.org/ (Official site of OpenGL).

APPENDIX B

List of Practical Sessions

Sessions 1 and 2 (Block 1)

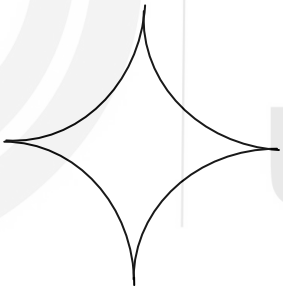
1. Writing a C program to create a window of specified size and position and draw the following objects with given dimensions, to fit within the window.

   (a) A point (b) A rectangle (c) A line segment

   (d) A triangle (e) A filled rectangle.

2. Writing a C program which takes points as input from mouse clicks (left button) and then performs an action. Applying the program to generate a closed polygon as follows: Every time a left button is pressed, a point on the screen is selected at mouse position and corresponding edge is drawn. When the right button is clicked, the polygon is created. (Use one of the line drawing algorithms to plot a line segment.)

3. Writing a simple C-code to generate a circular arc between two angle values. Use this to draw the following figure.



4. Writing a C-code which generates a font interactively. This means after every *n* mouse clicks, a Bezier curve is generated, and then the terminal point of the last drawn Bezier curve is taken as the initial point of the next Bezier curve and so on, until the user right clicks the mouse. In this case the curve of desired degree is drawn and the multi piece font boundary is generated. If the font generation required further drawing, the program allows starting again on the same window with previously drawn multi-piece curve. But this time the new curve will start with the fresh selected new point.

Session 3 (Block 1)

5. Implementing the Scan line polygon fill algorithm for any arbitrary polygon in C-language and then using the code to fill each of the following type of polygon.
   i) Convex polygon
   ii) Concave polygon
   iii) Self intersecting polygon.

Session 4 (Block 1)

6. Implementing the boundary fill algorithm and flood fill algorithm in C-language and using the code to fill two different types of closed areas such as
   i) A Circle
   ii) A self intersecting polygon
   Comparing the results of the two algorithms for self intersecting polygon.

7. Write a code using C-language to generate an arbitrary character by using
   (i) bitmap method
   (ii) outline method.
   Use this code to generate three different characters of your own mother tongue.

8. Writing a code in C-language for character generation using the standard OpenGL functions (both bitmap and outline style).

Session 5 (Block 1)

9. Writing a C-code for an interactive program which allows a user to draw a polygon object in a window and then gives various choices of geometric transformations on the polygon. Once the user selects a choice, the transformed polygon is also plotted on the window.

   OR

   Write a C-code that plots an object on the window, and on the user's click of mouse on the window the object starts rotating continuously until the user presses the mouse again.

10. Using the C-code of character generation to draw the same character with regular font type, and italic font type by making use of shear transformation.

Session 6 (Block 2)

11. Implementing the Cohen Sutherland and Liang Barsky line clipping algorithms in C-language. Test your code for line segments with end points falling in various regions.

Session 7 (Block 3)

12. Writing a code to generate a composite matrix for general 3D rotation matrix. Testing the code and rotating continuously a cube about an axis.

13. Writing a code for (i) general parallel projection where the view vector is at the choice of user and projection plane is x-y plane (ii) perspective projection where the projection reference point is at the user's choice and the projection plane is again xy-plane.