Chapter 8
Transformations and Clipping Area

After reading this chapter, you'll be able to do the following:

- Implementing 2D and 3D translation, scaling, and rotation.
- Implementing 3D deformation such as shearing, tapering, twisting, and bending
- Making codes for object transformation.

8.1 Introduction

Transformations are one of the primary vehicles used in computer graphics to manipulate objects in three-dimensional space. Their development is motivated by the process of converting coordinates between frames, which results in the generation of a 4x4 matrix. We can generalize this process and develop matrices that implement various transformations in space. In these notes, we discuss the basic transformations that are utilized in computer graphics - translation, rotation, scaling - along with several useful complex transformations, including those that work directly with the definition of a camera and the projections to image space. This chapter can be organized as follows: Section 2 introduces translation. In Section 3, we present scaling. Rotation transformation is described in Section 4. We present deformation in Section 5. Section 6 introduces program for transformation using OpenGL. Clipping area and viewpoint is described in Section 7.

8.2 Translation

Translation is one of the simplest transformations. A translation moves all points of an object a fixed distance in a specified direction. It can also be expressed in terms of two frames by expressing the coordinate system of object in terms of translated frames. Translation is a simple transformation that is calculated directly from the conversion matrix for two frames, one translate of the other. The translation matrix is most frequently applied to all points of an object in a local coordinate system resulting in an action that moves the object within this system.

Maps points (x, y) in one coordinate system to points (x', y') in another coordinate system:

$$x' = ax + by + c \tag{8.1}$$

$$y' = dx + ey + f \tag{8.2}$$

Why use transformations?

- Position objects in a scene (modelling)
- Change the shape of objects
- Create multiple copies of objects
- Projection for virtual cameras
- Animations

Rigid-Body / Euclidean Transforms

- Preserves distances
- Preserves angles

As shown in the following example:

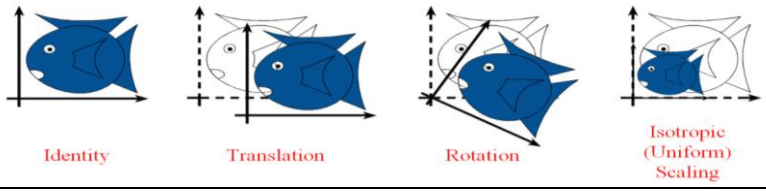


Fig.8-1: Preserve angle and shape.

8.2.1 Development of the transformation in terms of frames

Translation is a simple transformation. We can develop the matrix involved in a straightforward manner by considering the translation of a single frame.

If we are given a frame

$$F = (\vec{u}, \vec{v}, \vec{w}, O) \tag{8.3}$$

Then the translated frame would be one that is given by

$$F' = (\vec{u}, \vec{v}, \vec{w}, O') \tag{8.4}$$

that is, the origin is moved, the vectors stay the same.

If we write O' in terms of the previous frame by

$$O' = a\vec{u} + b\vec{v} + c\vec{w} + O \tag{8.5}$$

then we can write the frame F in terms of the frame F' by:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{bmatrix} \begin{bmatrix} \vec{u} \\ \vec{v} \\ \vec{w} \\ \boldsymbol{O} \end{bmatrix} = \begin{bmatrix} \vec{u} \\ \vec{v} \\ \vec{w} \\ \boldsymbol{O'} \end{bmatrix} \tag{8.6}$$

So a 4×4 matrix implements a frame-to-frame transformation for translated frames, and any matrix of this type (for arbitrary a, b, c) will translate the frame F. We call any matrix:

$$T_{a,b,c} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{bmatrix} \tag{8.7}$$

a translation matrix and utilize matrices of this type to implement translations.

8.2.2 Applying the transformation directly to the local coordinates of a point

Given a frame$F = (\vec{u}, \vec{v}, \vec{w}, O)$and a point P that has coordinates in$F$, if we apply the transformation to the coordinates of the point we obtain:

$$[u \quad v \quad w \quad 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{bmatrix} = [u + a \quad v + b \quad w + c \quad 1] \tag{8.8}$$

That is, we can translate the point *within the frame* F. An illustration of this is shown in the Fig.8-2.

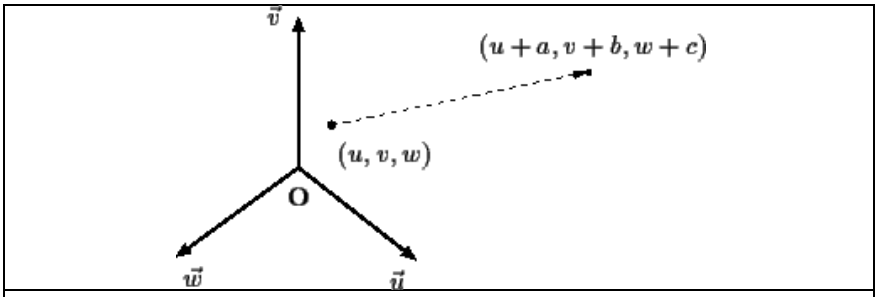


Fig.8-2: Translation(u, v, w) using (a,b,c) parameters.

8.3 Scaling

Scaling, like translation is a simple transformation which just scales the coordinates of an object. It is specified either by working directly with the local coordinates, or by expressing the coordinates in terms of Frames.

8.3.1 Development of the transformation via scaled frames

The scaling transformation can be represented by a simple 4×4 matrix whose only entries are on the diagonal. This transformation, when applied to an object multiplies each of the local coordinates of the object by a factor - effectively scaling the object about the origin. Scaling about other points can be done by combining the scaling transformation with two translation transformations.

Given a frame $F = (\vec{u}, \vec{v}, \vec{w}, O)$ a scaled frame would be one that is given by $F' = (\vec{u}, \vec{v}, \vec{w}, O')$ that is, we just expand (or contract) the lengths of the vectors defining the frame. It is fairly easy to see that we can write the frame F' in terms of the frame F by:

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \vec{u} \\ \vec{v} \\ \vec{w} \\ \boldsymbol{O} \end{bmatrix} = \begin{bmatrix} a\vec{u} \\ b\vec{v} \\ c\vec{w} \\ \boldsymbol{O} \end{bmatrix} \qquad (8.9)$$

So a 4×4 matrix implements a scaling transformation on frames, and any matrix of this type (for arbitrary *a*,*b*,*c*) will scale the frame F. We call the matrix:

$$S_{a,b,c} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.10}$$

a scaling matrix and utilize matrices of this type to implement our scaling operations.

8.3.2 Applying the transformation directly to the local coordinates

Given a frame $F = (\vec{u}, \vec{v}, \vec{w}, O)$ and a point P that has local coordinates $\vec{u}, \vec{v}, \vec{w}, O$ in F, if we apply the transformation $S_{a,b,c}$ to the local coordinates of the point, we obtain

$$[u \quad v \quad w \quad 1] \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [au \quad bv \quad cw \quad 1] \tag{8.11}$$

and we have scaled the point *within the frame F* . An illustration of this is shown in Fig.8-3.

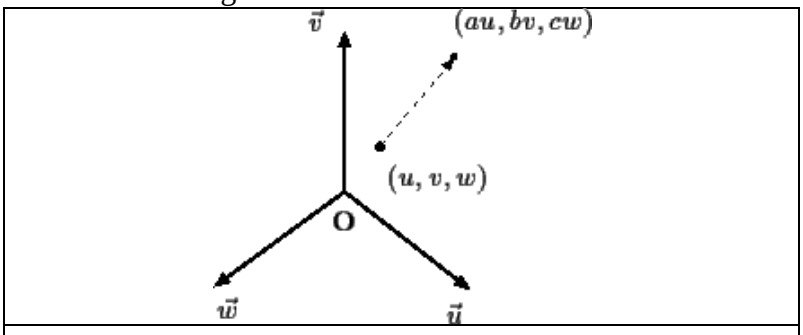


Fig.8-3: Scaling(u, v, w) using (*a*,*b*,*c*) parameters.

8.3.3 Scaling points cooresponding the origin

It is difficult to see the origin of the scaling operation when working only with coordinates - so for example, consider the eight vertices of a cube centered at the origin in the Cartesian frame.
(-1, -1, 1), (-1, 1, 1), (1, 1, 1), (1, -1, 1), (-1, -1, -1), (-1, 1, -1), (1, 1, -1), (1, -1, -1)
If we consider the ``scaling" transformation given by:

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (8.12)$$

and apply this matrix to each of the coordinates of the points relative to the standard frame, we obtain a new set of points (-2, -2, 2), (-2, 2, 2), (2, 2, 2), (2, -2, 2), (-2, -2, -2), (-2, 2, -2), (2, 2, -2), (2, -2, -2) which is an effective scaling of the cube (The resulting cube has volume 8 times the original as shown in Fig.8-4). This operation is illustrated in the following figure (Note that this figure is viewing the cube from along the z-axis).

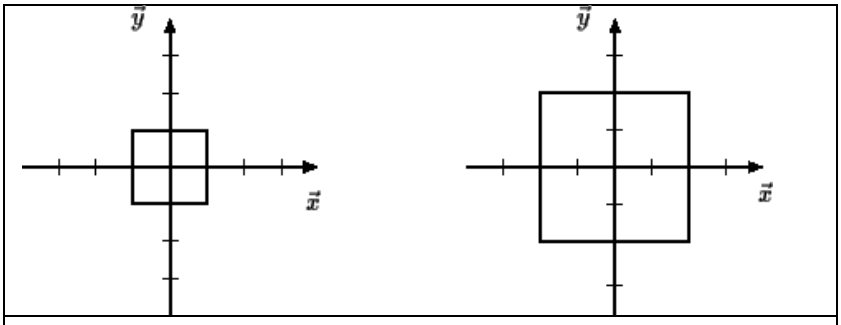


Fig.8-4: Cube around the original.

We note that this operation scales about the origin of the coordinate system. If the center of the object is not at the origin, this operation

will move the object away from the origin of the frame. If we consider a cube with the following coordinates at its corners (1, 1, 1), (1, 3, 1), (3, 3, 1), (3, 1, 1), (1, 1, 3), (1, 3, 3), (3, 3, 3), (3, 1, 3)
Then by applying the above transformation, this cube is transformed to a cube with the following coordinates (2, 2, 2), (2, 6, 2), (6, 6, 2), (6, 2, 2), (2, 2, 6), (2, 6, 6), (6, 6, 6), (6, 2, 6) which is a cube with volume 8 times the original, but centered at the point (4, 4, 4) as shown in Fig.8-5. This is illustrated in the following figure.

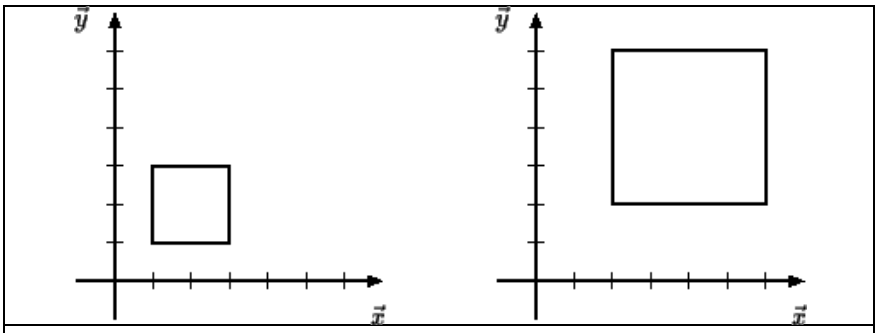


Fig.8-5: Cube around the point (4, 4, 4).

If the desired scaling point is not at the origin of the frame, we must utilize a combination of transformations to get an object to scale correctly. If the scaling point is at (u_s, v_s, w_s) in the frame, we can utilize the translation $T_{-u_s,-v_s,-w_s}$ to first move the point to the origin of the frame, then scale the object, and finally use the translation T_{u_s,v_s,w_s} to move the point back to the origin of the scaling. These transformations are all represented by 4×4 matrices:

$$T_{-u_s,-v_s,-w_s}\, S_{a,b,c}\, T_{u_s,v_s,w_s} \tag{8.13}$$

and we take their matrix product to create one 4×4 matrix that gives the transformation.

8.4 Rotation

Rotations are complex transformations. The primary complexity is that in three-dimensions, rotations are performed about an *axis* - usually specified by a point and a vector direction. The general idea is

to develop this transformation about the three coordinate axes of a frame, and then generalize this to be able to rotate about a general axis in the frame. The general idea for the matrix comes from rotation about a point in two-dimensions.

8.4.1 Development of the transformation via rotation matrix

The 3-dimensional rotation matrix, when rotating about one of the coordinate axes is quite similar to the 3×3 rotation matrix developed for rotation in two-dimensions. Here rotation is much simpler to describe, as rotation is about a point in two-dimensions. Here we develop the rotation matrix in two-dimensions that rotates a point about the origin in the Cartesian frame.

In Two Dimensions

Rotation about the origin in two-dimensions is given by a simple 3×3 matrix written in terms of the cosine and sine of the angle of rotation. This development can be applied directly to develop the rotation matrices for the three-dimensional rotations about the *x, y* and *z* axes.

In two dimensions, one rotates about a point. We will rotate about the origin, and will consider our 2-d frame to be the two-dimensional Cartesian frame. Consider the Fig. 8.6:

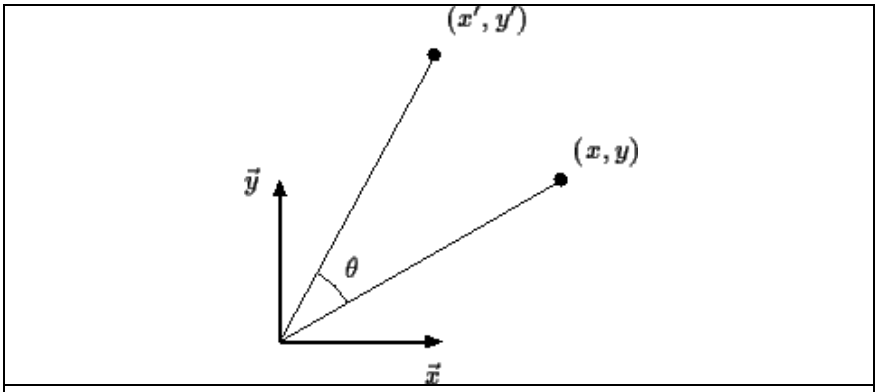


Fig.8-6: 2D point rotation.

where we depict a rotation of θ units about the origin and the point (x, y) is rotated into the point (x', y'). By considering the Fig. 8.7:

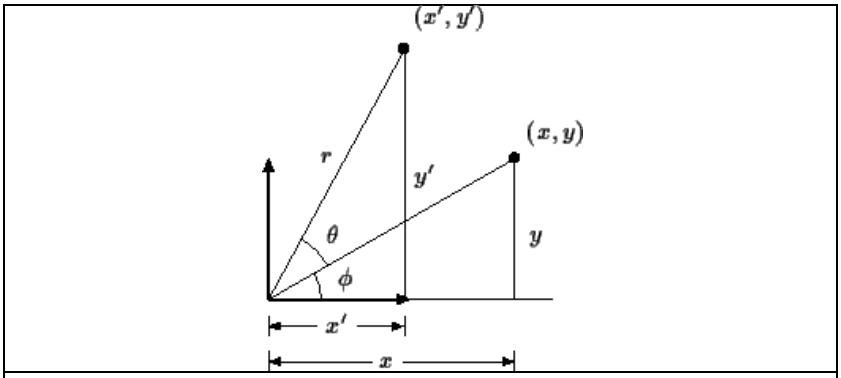


Fig.8-7: Rotation parameters.

we note that (x, y) can be written in polar coordinates as

$$(x, y) = (rcos\emptyset, rsin\emptyset) \tag{8.14}$$

and also that (x', y') can be written in polar coordinates as

$$(x', y') = \big(rcos(\emptyset + \theta), rsin(\emptyset + \theta)\big) \tag{8.15}$$

Expanding the description of (x', y') we obtain

$$\begin{aligned}(x', y') &= \big(rcos(\emptyset + \theta), rsin(\emptyset + \theta)\big) \\ &= (rcos\emptyset cos\theta - rsin\emptyset sin\theta, rsin\emptyset cos\theta + rcos\emptyset sin\theta) \\ &= (xcos\theta - ysin\theta, xsin\theta + ycos\theta)\end{aligned} \tag{8.16}$$

which can be written in matrix form as

$$[\,x \quad y \quad 1\,]\begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{8.17}$$

So in 2-dimensions, rotation is implemented as a 3×3 matrix given by

$$\begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (8.18)$$

Rotation about *x*-axis

We have developed a simple method using only basic transformations by which general rotation can be accomplished. It utilizes translation and the basic rotations about the x axis, the y axis, and the z axis to accomplish this task. This individual matrices specified may be multiplied together to give one 4×4 matrix that represents the general rotation.

Rotation about the x-axis is similar to rotation specified in two-dimensional space as in the three-dimensional rotation; the x-coordinate must remain constant.

Specification of the rotation matrix

The transformation for rotation of θ radians about the x-axis in the Cartesian frame is given by the matrix:

$$R_{x;\theta} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & sin\theta & 0 \\ 0 & -sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (8.19)$$

(Note that as a point is rotated about the x-axis, the x value of the point will not change.) If this transformation is applied to the coordinate (x, y, z) we obtain:

$$[\,x \;\; y \;\; z \;\; 1] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & sin\theta & 0 \\ 0 & -sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= [\,x \;\; ycos\theta - zsin\theta \;\; ysin\theta + zcos\theta \;\; 1] \quad (8.20)$$

The effect of this transformation is illustrated in Fig.8-8.

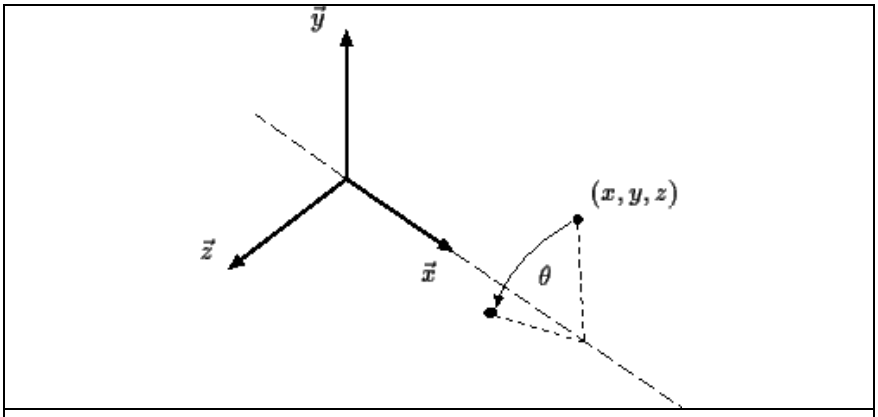


Fig.8-8: Transformation for rotation of θ radians about the x-axis.

Rotation about y-axis

Rotation about the y-axis is similar to rotation specified in two-dimensional space as in the three-dimensional rotation; the y coordinate must remain constant.

Specification of the rotation matrix

The transformation for rotation of θ radians about the y-axis is given by the 4×4 matrix:

$$R_{y;\theta} = \begin{bmatrix} cos\theta & 0 & -sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.21}$$

If this transformation is applied to the point (x, y, z), we obtain:

$$[x \quad y \quad z \quad 1]\begin{bmatrix} cos\theta & 0 & -sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
$$= [\, xcos\theta + zsin\theta \quad y \quad -xsin\theta + zcos\theta \quad 1] \tag{8.22}$$

This transformation is illustrated in Fig.8-9:

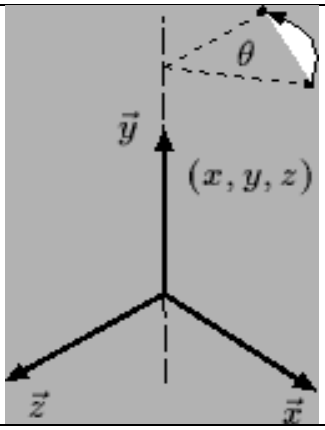


Fig.8-9: Transformation for rotation of θ radians about the y-axis.

Rotation about z-axis

Rotation about the z-axis is similar to rotation specified in two-dimensional space as in the three-dimensional rotation; the z-coordinate must remain constant.

Specification of the rotation matrix

The transformation for rotation of θ radians about the z-axis is given by the 4×4 matrix:

$$R_{z;\theta} = \begin{bmatrix} cos\theta & sin\theta & 0 & 0 \\ -sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.23}$$

If this transformation is applied to the point (x, y, z), we obtain

$$[\,x \;\; y \;\; z \;\; 1]\begin{bmatrix} cos\theta & sin\theta & 0 & 0 \\ -sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
$$= [\,xcos\theta - ysin\theta \quad xsin\theta + ycos\theta \quad z \quad 1] \qquad (8.24)$$

The effect of this transform is illustrated in Fig.8-10:

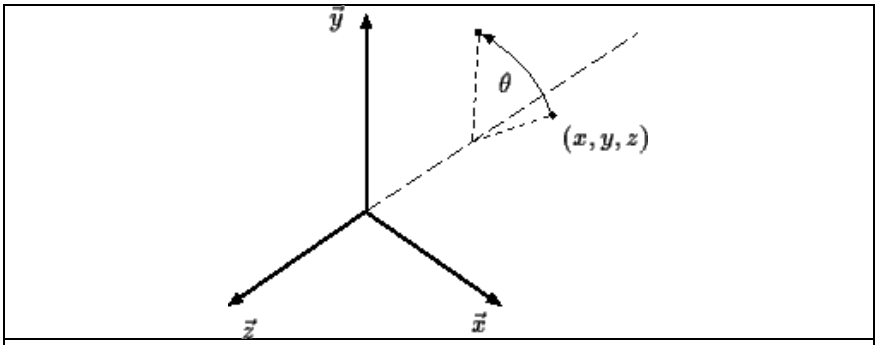


Fig.8-10: Transformation for rotation of θ radians about the z-axis.

General rotation about an axis

An axis in space is specified by a point P and a vector direction $\vec{t}$. Suppose that we wish to rotate an object about this arbitrary axis:

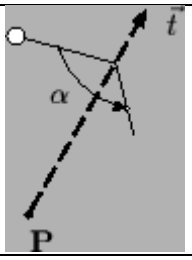


Fig.8-11: General rotation.

We know how to do this in the cases that the axis is the x-axis, the y-axis, or the z-axis in the Cartesian frame (these were just generalizations of the two-dimensional rotations), but the general case is more difficult. In these notes we present a solution to this problem that utilizes both translation and the above rotation matrices to accomplish this task. (One can also approach this problem through the use of frame-to-frame-conversion transformations.)

8.4.2 Developing the general rotation matrix

First assume that the axis of rotation can be specified in terms of Cartesian coordinates, i.e. can be represented by the point

$$P = (x_p, y_p, z_p) \tag{8.25}$$

and the vector

$$\vec{t} = (x_t, y_t, z_t) \tag{8.26}$$

Then a rotation of α degree about this axis can be defined by concatenating the following transformations

- Translate so that the point P moves to the origin $T(-x_p, -y_p, -z_p)$ (see Fig.8-12).

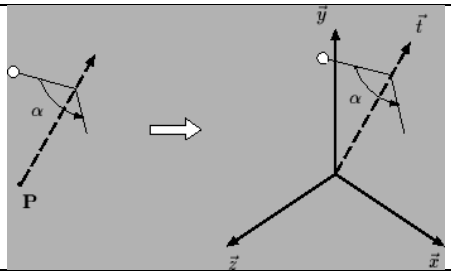


Fig.8-12: Translation P to the origin $T(-x_p, -y_p, -z_p)$.

Use the elementary rotation transformations to rotate the vector until it coincides with one of the coordinate axes. To do this, first rotate the vector until it is in the *yz* plane by using a rotation $R_{y,\theta}$ about the **y axis** (see Fig.8-13).

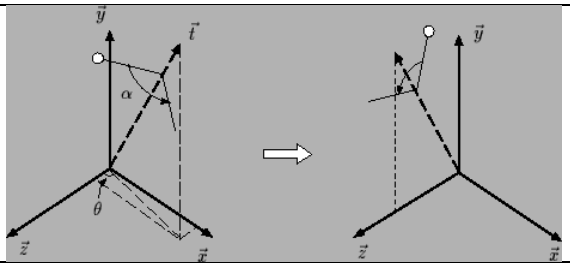


Fig.8-13: Rotation $R_{y,\theta}$ about the *y* axis.

where $\theta = \arctan\left(\frac{x_v}{z_v}\right)$, and then use an ***x*-axis rotation**, of $R_{x,\emptyset}$ to rotate the vector until it coincides with the ***z* axis** (see Fig.8-14):

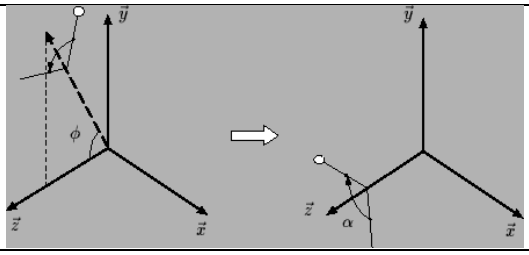


Fig.8-14: Rotation $R_{x,\emptyset}$about the *z* axis.

$$\text{where } \theta = \arctan\left(\frac{y_v}{\sqrt{z_v + x_v}}\right) \tag{8.27}$$

Then use an α rotation about **the *z* axis** $R_{x,\alpha}$(see Fig.8-15).

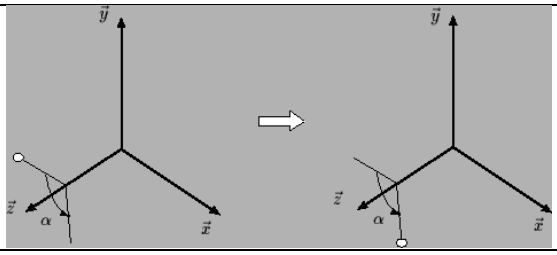


Fig.8-15: Rotation $R_{x,\alpha}$about the *z* axis.

- Use rotations and translations to reverse the first two processes: First by a rotation $R_{x,-\emptyset}$ about **the *x* axis**(see Fig.8-16)**:**

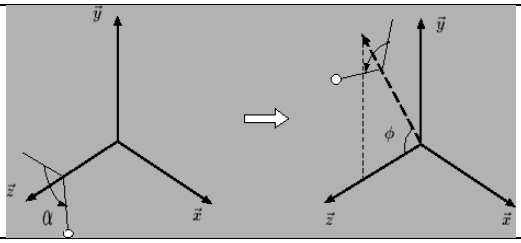


Fig.8-16: Rotation $R_{x,-\emptyset}$ about the *x* axis.

then by a rotation $R_{y,\theta}$ about **the *y* axis** (see Fig.8-17)**:**

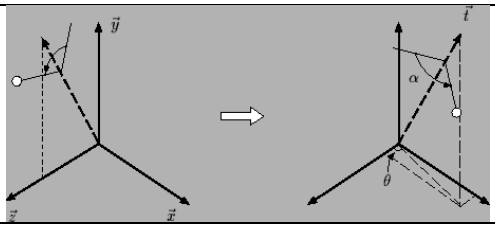


Fig.8-17: Rotation $R_{y,\theta}$ about the *y* axis.

and finally using **the translation**, $T(x_p, y_p, z_p)$ to translate back to the **original axis** (see Fig.8-18):

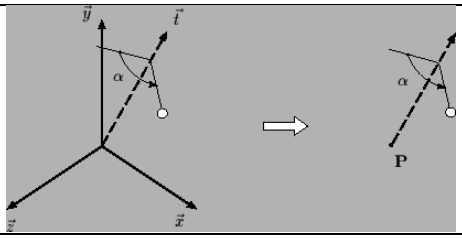


Fig.8-18: Translation, $T(x_p, y_p, z_p)$ to translate back to the original axis.

The matrix representation of the general rotation is given by the product of the above transformations.

$T(-x_p, -y_p, -z_p)\, R_{y,-\theta}\, R_{x,\emptyset}\, R_{z,\alpha}\, R_{x,-\emptyset}\, R_{y,\theta}\, T(x_p, y_p, z_p)$

These can be multiplied together (they are all 4×4 matrices) to give one 4×4 matrix which represents the general rotation.

What if the axis was specified in a local frame?

In this case, we just convert the coordinates of the point *P* and a vector $\vec{t}$ defining the axis to Cartesian coordinates using the frame-to-Cartesian-frame transformation, do the above operations, and then use the Cartesian-frame-to-frame to convert the resulting coordinates back to the local system. We can rotate the frame of coordinates as shown in Fig.8-19.

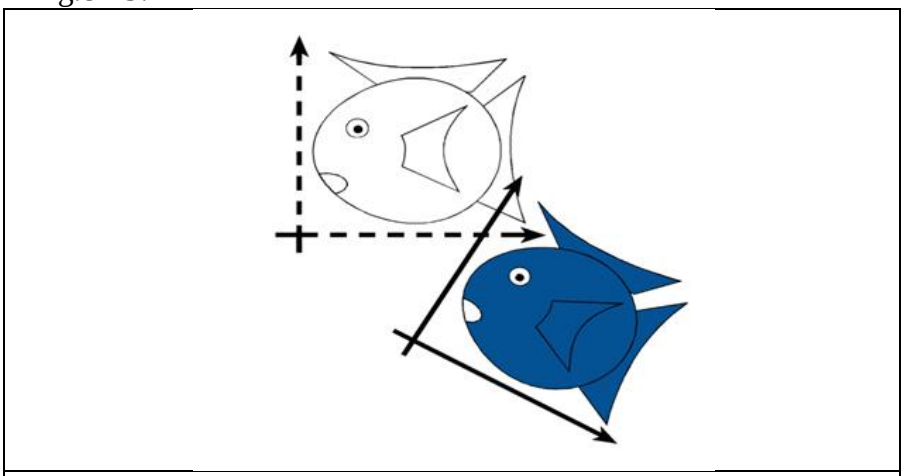


Fig.8-19: Transformation operators.

8.5 Deformations

It is the transformations that do not preserve shape and include:

- Shearing
- Tapering
- Twisting
- Bending

8.5.1 Shearing

A shear is something that pushes things sideways either in the *x* or *y* or *z* direction. A shear in the *x*,*y*,*z* direction would tilt the vertical axis as shown on Fig.8-20. The matrix of a shear transformation can be described as follows:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & s_{xy} & s_{xz} & 0 \\ s_{yx} & 1 & s_{yz} & 0 \\ s_{zx} & s_{zy} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{8.28}$$

$$s_{xy} = 1$$
$$s_{xz} = 0$$
$$s_{yx} = s_{yz} = 0$$
$$s_{zx} = s_{zy} = 0$$

8.5.2 Tapering

In mathematics, physics, and theoretical computer graphics, tapering is a kind of shape deformation. Just as an affine transformation, such as scaling or shearing, is a first-order model of shape deformation, there also exist higher-order deformations such as tapering, twisting, and bending. Tapering can be thought of as non-constant scaling by a given tapering function (see Fig.8-20). The resultant deformations can be linear or nonlinear. To create a taper, instead of scaling in *x* and *y* for all *z* with constants, we represent them as function as follows:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & f(x) & 0 & 0 \\ 0 & 0 & f(x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{8.29}$$

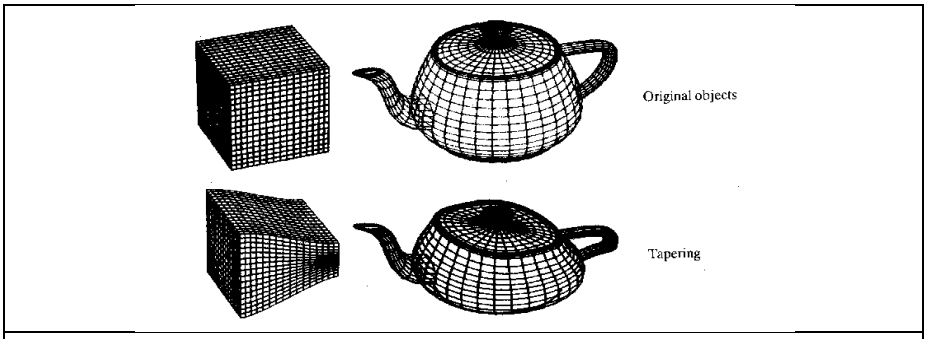


Fig.8-20: Tapering.

8.5.3 Twisting

Similar tapering, the twisting of the object can be shown in Fig.8-21, and the transformation is represented as:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta(y)) & 0 & -\sin(\theta(y)) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta(y)) & 0 & \cos(\theta(y)) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (30)$$

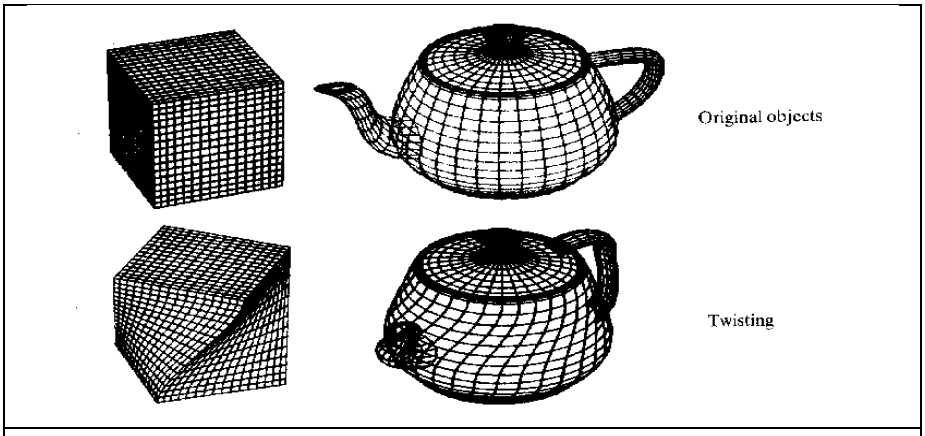


Fig.8-21: Twisting.

8.5.4 Bending

Similar tapering, the twisting of the object can be shown in Fig.8-22, and the transformation is represented as:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & f(y) & g(y) & 0 \\ 0 & h(y) & k(y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (8.31)$$

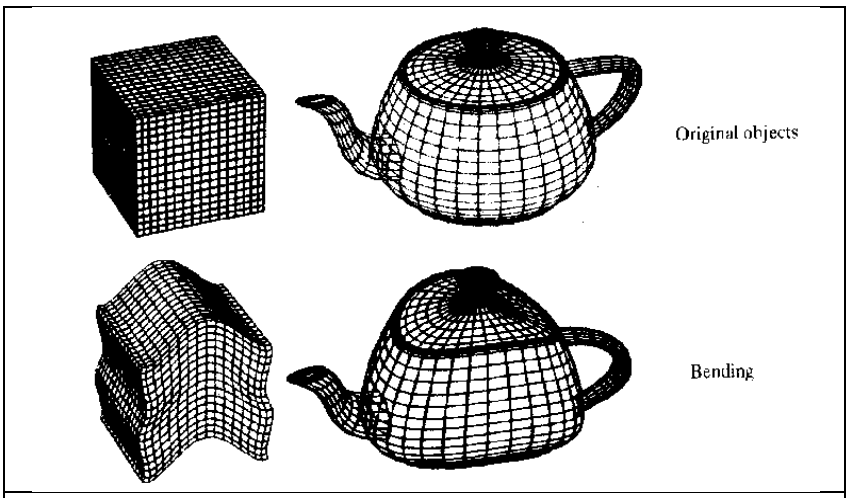


Fig.8-22: Bending.

8.6 Transformation in OpenGL

2D and 3D translation, scaling, and rotation OpenGL code are presented in this section.

Example 8-1: Two dimensional translations

glTranslatef command helps us to move objects from one location to another. This command has three parameters one for each axis. The following functions are used for drawing 3D object.

Name

glPushMatrix, glPopMatrix

Push and pop the current matrix stack

CSpecification

void**glPushMatrix**(void)

void**glPopMatrix**(void)

Description

There is a stack of matrices for eachof the matrix modes.

Pseudo code to draw one triangle and translate it.

```
voidPatternSegment(void)
{
      void Tri();
      glClear (GL_COLOR_BUFFER_BIT);
      glPushMatrix();
      glColor3f (1.0, 0.0, 0.0);
      Tri();
      glTranslatef(10.0,10.0,0.0);//Translation
      process
      glColor3f (0.0, 0.0, 1.0);
      Tri();
      glPopMatrix();
      glutSwapBuffers();
      glFlush ();
}
void Tri()
{
      glBegin(GL_TRIANGLES);
      glVertex2i(10, 10);
      glVertex2i(60, 10);
      glVertex2i(30, 60);
      glEnd();
}
```

Complete Program:

```
#include<windows.h>
#include<gl.h>
#include <glut.h>
voidPatternSegment(void);
voidinit (void);
voidPatternSegment(void)
{
void Tri();
glClear (GL_COLOR_BUFFER_BIT);
glPushMatrix();
glColor3f (1.0, 0.0, 0.0);
Tri();
glTranslatef(10.0,10.0,0.0);
glColor3f (0.0, 0.0, 1.0);
Tri();
glPopMatrix();
glutSwapBuffers();
glFlush ();
}
voidinit (void)
{
    glClearColor (1.0, 1.0, 1.0, 0.0);
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(0.0, 200.0, 0.0, 150.0);
}

void Tri()
{
    glBegin(GL_TRIANGLES);
    glVertex2i(10, 10);
    glVertex2i(60, 10);
    glVertex2i(30, 60);
    glEnd();
}
int main(intargc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize (400, 300);
glutInitWindowPosition (50, 100);
glutCreateWindow ("Muhammad");
init ();
glutDisplayFunc(PatternSegment);
glutMainLoop();
return 0;
}
```

Translation

Drawing

Fig.8-23: Complete program to draw one triangle and translate it.

Output:

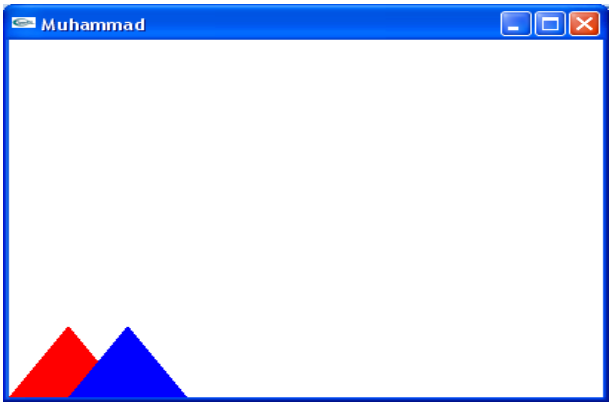


Fig.8-24: One triangle and translated triangle.

Example 8-2: Two dimensional scaling

To implement two dimensional scaling using glScalef command, we use the Example 8-1, and we just modified the void PatternSegment(void) function to be:

```
voidPatternSegment(void)
{
      glClear (GL_COLOR_BUFFER_BIT);
      glPushMatrix();
      glColor3f (1.0, 0.0, 0.0);
      Tri();
      glScalef(10.0,10.0,0.0);//scaling
      transformation
      glColor3f (0.0, 1.0, 0.0);
      Tri();
      glPopMatrix();
      glFlush ();
}
void Tri()
```

```
{
      glBegin(GL_TRIANGLES);
      glVertex2i(10, 10);
      glVertex2i(60, 10);
      glVertex2i(30, 60);
      glEnd();
}
```

Output:

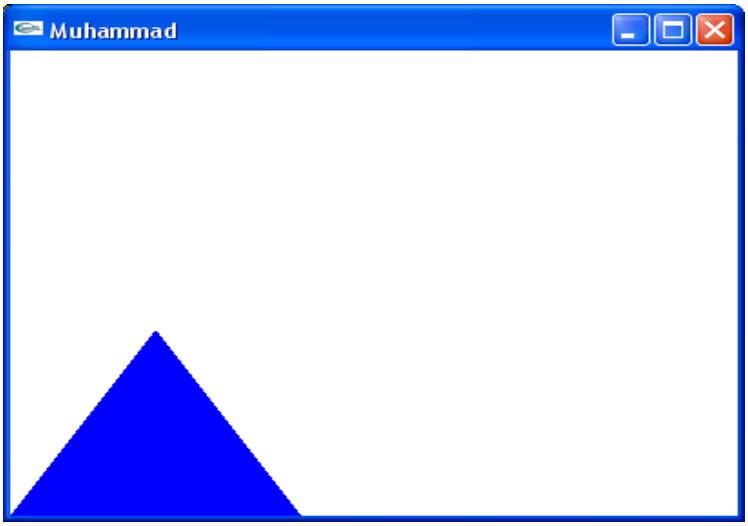


Fig.8-25: Scaling the triangle.

Example 8-3: two dimensional rotations

glRotatef command helps us to rotate objects from one angle to another. This command has four parameters one angle and rest for each axis. We use the Example 8-1, and we just modified the void PatternSegment (void) function to be:

```
voidPatternSegment(void)
{
      glClear (GL_COLOR_BUFFER_BIT);
      glPushMatrix();
      glColor3f (1.0, 0.0, 0.0);
```

```
        Tri();
        glRotatef(10.0,1.0,1.0,0.0);//Rotation process
        glColor3f (0.0, 0.0, 1.0);
        Tri();
        glPopMatrix();
        glFlush ();
}
void Tri()
{
        glBegin(GL_TRIANGLES);
        glVertex2i(10, 10);
        glVertex2i(60, 10);
        glVertex2i(30, 60);
        glEnd();
}
```

Output:

Fig.8-26: Rotate the triangle.

Example 8-4: Three Dimensional Drawing

gluPerspective - set up a perspective projection matrix

void**gluPerspective**(GLdouble *fovy*,
GLdouble *aspect*,
GLdouble *zNear*,
GLdouble *zFar*)

Parameters

*fovy*Specifies thefield of view angle, indegrees, in the *y*direction.

aspect Specifies theaspect ratio that determines the field of view in the *x* direction. The aspect ratio is the ratio of *x* (width) to *y* (height).

*zNear*Specifies the distance from the viewer to the near clipping plane (always positive).

*zFar*Specifies the distance from the viewer to the far clipping plane (always positive).

Description

gluPerspective **specifies** a viewing frustum into the world coordinate system. In general, the aspect ratio in gluPerspective should match the aspect ratio of the associated viewport. For example, aspect=2.0 means the viewer's angle of view is twice as wide in *x* as it is in *y*.

If the viewport is twice as wide as it is tall, it displays the image without distortion.

Name

glLoadIdentity - replace the current matrix with the identity matrix

CSpecification

void**glLoadIdentity**(void)

Description

glLoadIdentity replaces the current matrix with the identity matrix

Name

glEnable,glDisable - enable or disable server-side GL capabilities

CSpecification

void**glEnable**(GLenum*cap*)

Parameters

cap Specifies a symbolic constant indicating a GL capability.

CSpecification

void**glDisable**(GLenum*cap*)

Parameters

cap Specifies a symbolic constant indicating a GL capability.

Description

glEnable and **glDisable** enable and disable various capabilities.

Program:

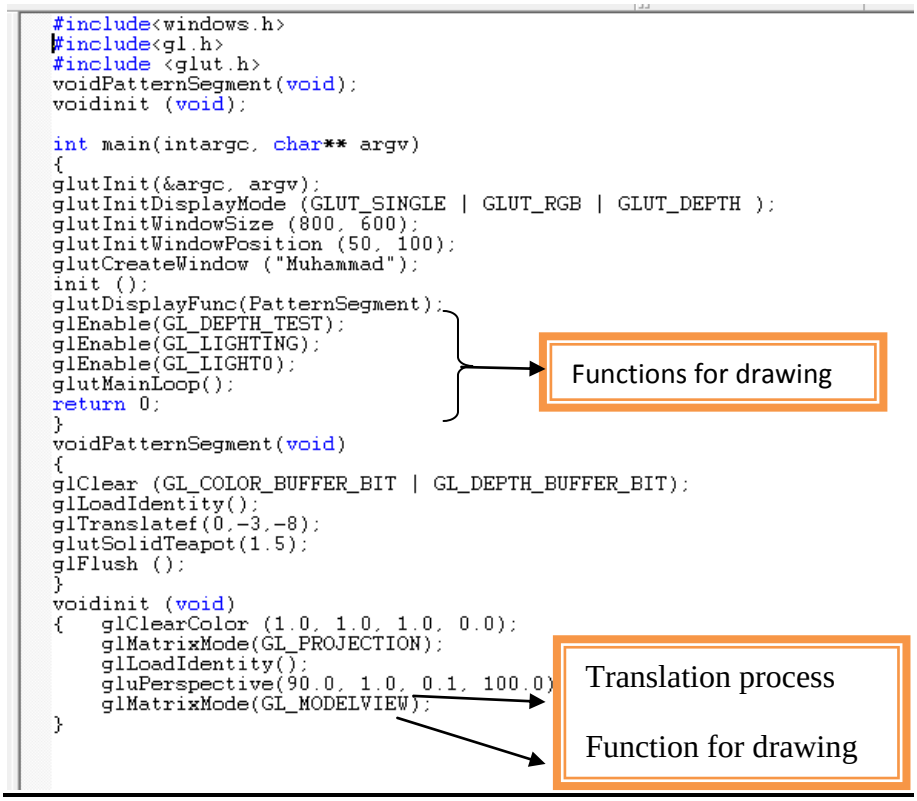


Fig.8-27: Complete program to draw three-dimensional object.

Output:

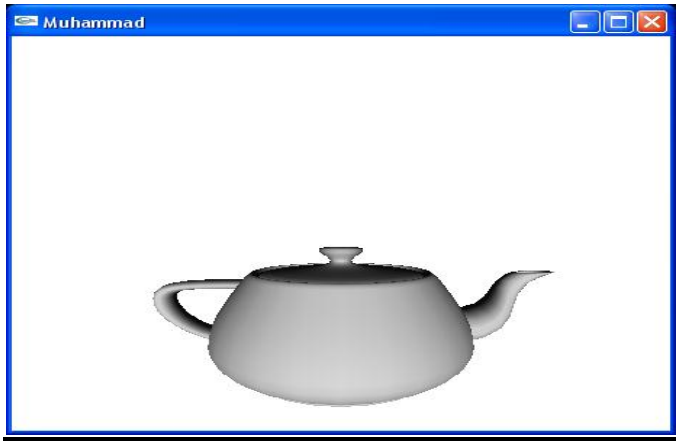


Fig.8-28: The output Teapot.

Example 5-6: Three dimensional transformations

```
voidPatternSegment(void)
{
      glClear (GL_COLOR_BUFFER_BIT |
      GL_DEPTH_BUFFER_BIT);
      glLoadIdentity();
      glTranslatef(0,-3,-8);
      glRotatef(60,0,1,0);
      glutSolidTeapot(3);
      glFlush ();
}
```

Output:

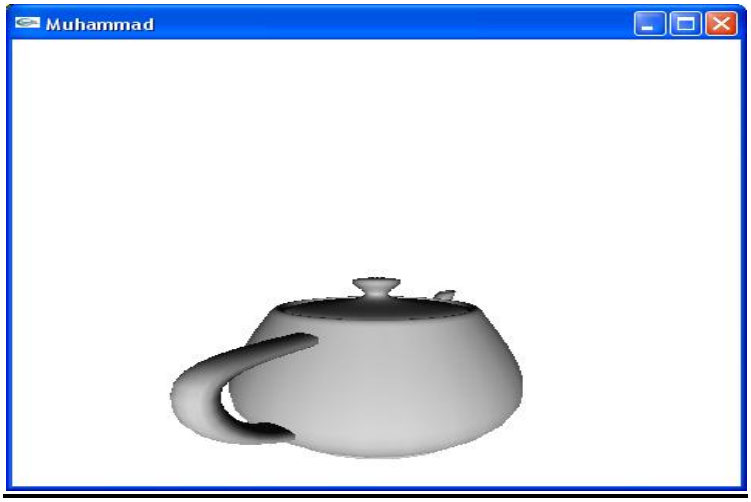


Fig.8-29: The output Teapot after transformation.

Example 5-6: Three dimensional transformations with keyboard

We can move the 3D objects to different locations using the keys of the keyboard.

Pseudo code:

```
voidPatternSegment(void)
{
      glClear (GL_COLOR_BUFFER_BIT |
      GL_DEPTH_BUFFER_BIT);
      glLoadIdentity();
      glRotatef(rot,0,1,0);
      glTranslatef(0,-3,-8);
      glutSolidTeapot(1);
      glFlush ();
}
voidinit (void)
{     glClearColor (1.0, 1.0, 1.0, 0.0);
```

```
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluPerspective(90.0, 1.0, 0.1, 100.0);
      glMatrixMode(GL_MODELVIEW);
}
void Keyboard(unsigned char Key, int x, int y)
{
      if (Key==27) exit(1);
      if(++rot>359) rot=0;
      glutPostRedisplay();  }
```

Output:

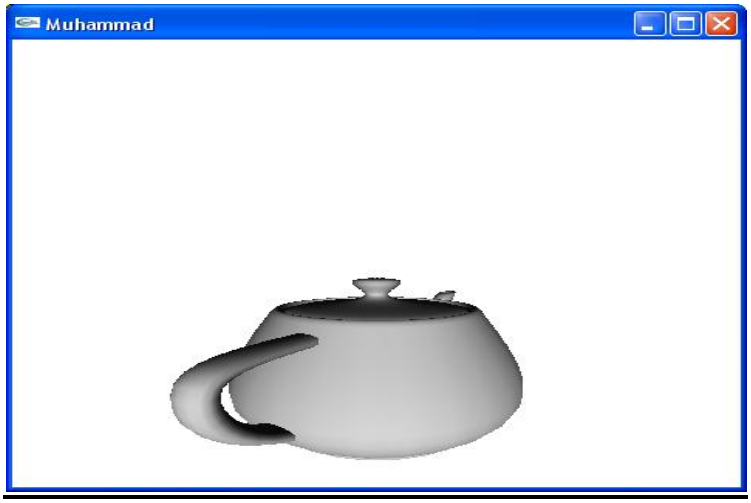


Fig.8-30: The output Teapot locations using the keys of the keyboard.

Example 5-7: Three dimensional transformations with mouse

We can move the 3D objects to different locations using the buttons of the mouse.

```
voidPatternSegment(void)
{
```

```
        glClear (GL_COLOR_BUFFER_BIT |
        GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        glRotatef(rot,0,1,0);
        glTranslatef(0,-3,-8);
        glutSolidTeapot(1);
        glFlush ();
}
void Mouse(int button, int state, int x, int y)
{     if(button == GLUT_RIGHT_BUTTON && state ==
GLUT_DOWN )
      {
            if (++rot>359) rot=0;
            glutPostRedisplay();}}
```

Output:

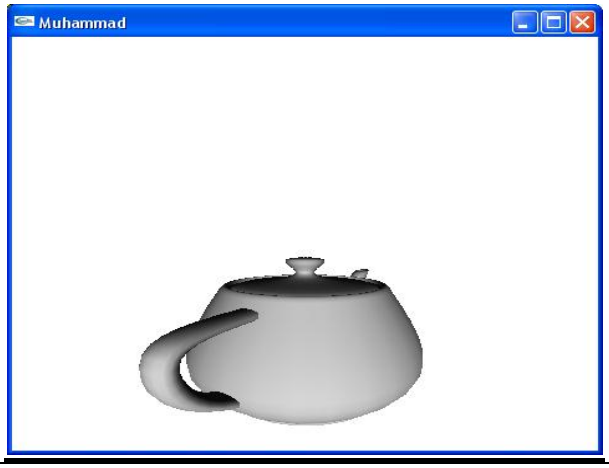


Fig.8-31: The output Teapot locations using the keys of the mouse.

8.7 Clipping-area and viewport

The following diagram shows the OpenGL 2D Coordinate System, which corresponds to the everyday 2D Cartesian coordinates with origin located at the bottom-left corner.

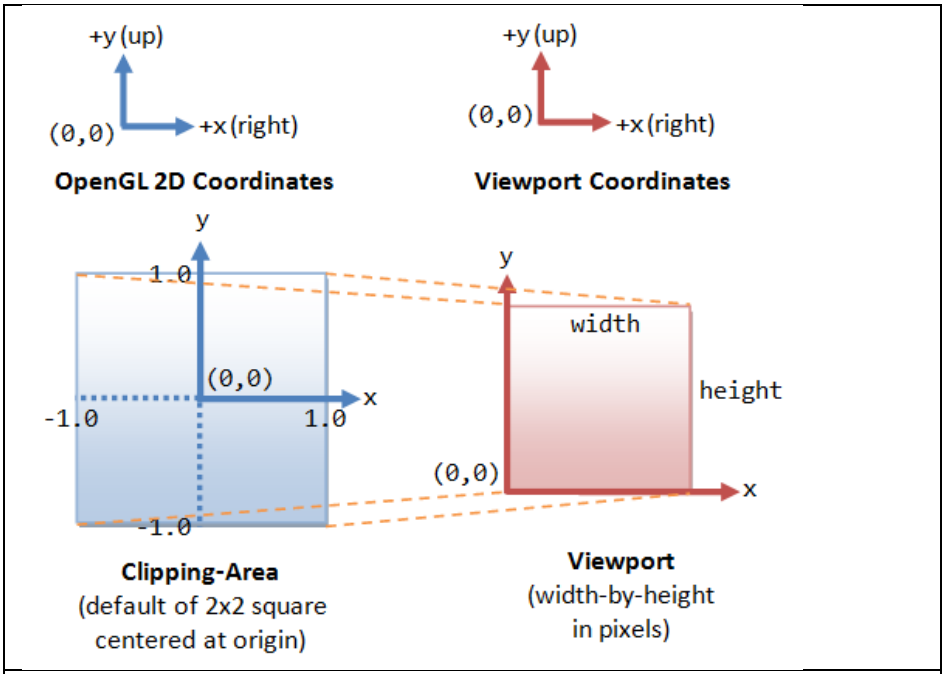


Fig.8-32: Clipping area and viewport.

The default OpenGL 2D clipping-area (i.e., what is captured by the camera) is an orthographic view with x and y in the range of -1.0 and 1.0, i.e., a 2×2 square with centered at the origin. This clipping-area is mapped to the viewport on the screen. Viewport is measured in pixels. Study the above example to convince yourself that the 2D shapes created are positioned correctly on the screen(see Fig.8-32). Fig.8-33 presents an illustrated example to show the clipping area and viewport.

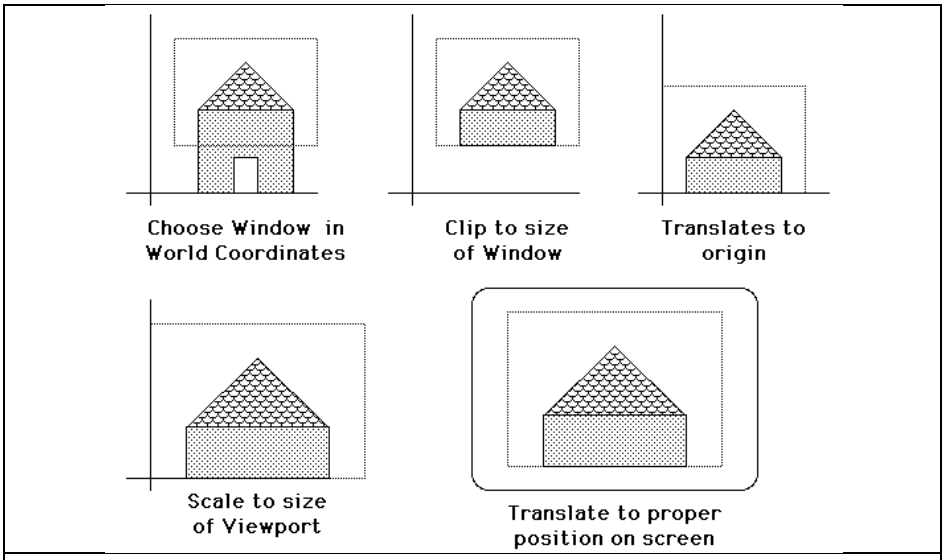


Fig.8-33: Example to show clipping area and viewport.

8.7.1 Clipping-area and viewport

Try dragging the corner of the window to make it bigger or smaller. Observe that all the shapes are distorted. We can handle the re-sizing of window via a callback handler reshape (), which can be programmed to adjust the OpenGL clipping-area according to the window's aspect ratio. The objects will be distorted if the aspect ratios of the clipping area and viewport are different as shown in Fig.8-34.

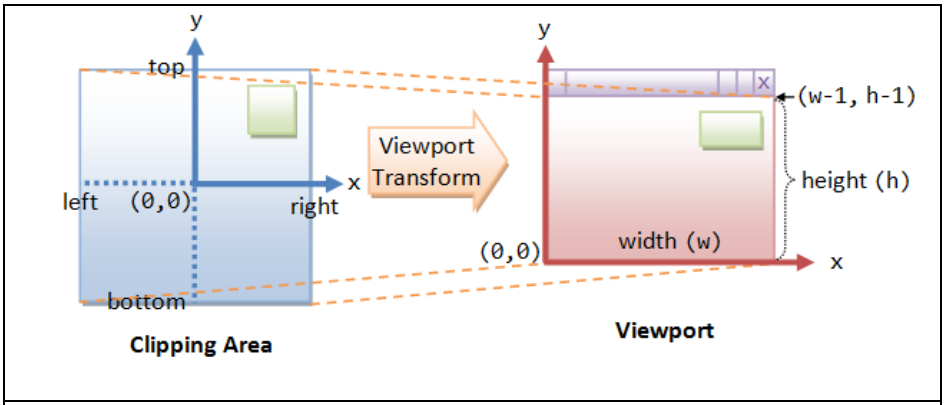


Fig.8-34: Relation between the clipping area and viewport.

Clipping Area: Clipping area refers to the area that can be seen (i.e., captured by the camera), measured in OpenGL coordinates. The function gluOrtho2D can be used to set the clipping area of 2D orthographic view. Objects outside the clipping area will be clipped away and cannot be seen:

void**gluOrtho2D**(GLdouble*left*, GLdouble*right*, GLdouble*bottom*, GLdouble*top*)

```
// The default clipping area is (-1.0, 1.0, -1.0, 1.0) in OpenGL coordinates,
// i.e., 2x2 square centered at the origin.
```

To set the clipping area, we need to issue a series of commands as follows: we first select the so-called projection matrix for operation, and reset the projection matrix to identity. We then choose the 2D orthographic view with the desired clipping area, via gluOrtho2D().

```
// Set to 2D orthographic projection with the specified clipping area
glMatrixMode(GL_PROJECTION);     // Select the Projection matrix for operation
glLoadIdentity();                // Reset Projection matrix
gluOrtho2D(-1.0, 1.0, -1.0, 1.0); // Set clipping area's left, right, bottom, top
```

Viewport: Viewport refers to the display area on the window (screen), which is measured in pixels in screen coordinates (excluding the title bar).The clipping area is mapped to the viewport. We can use glViewport function to configure the viewport.

void**glViewport**(GLint*xTopLeft*, GLint*yTopLeft*, GLsizei*width*, GLsizei*height*)

Suppose the clipping area's (left, right, bottom, top) is (-1.0, 1.0, -1.0, 1.0) (in OpenGL coordinates) and the viewport's (xTopLeft,

xTopRight, width, height) is (0, 0, 640, 480) (in screen coordinates in pixels), then the bottom-left corner (-1.0, -1.0) maps to (0, 0) in the viewport, the top-right corner (1.0, 1.0) maps to (639, 479). It is obvious that if the aspect ratios for the clipping area and the viewport are not the same, the shapes will be distorted.

8.7.2 Clipping-area and viewport without distorted object

A reshape() function, which is called back when the window first appears and whenever the window is re-sized, can be used to ensure consistent aspect ratio between clipping-area and viewport, as shown in Fig.8-35. The graphics sub-system passes the window's width and height, in pixels, into the reshape().

GLfloat aspect = (GLfloat) width / (GLfloat)height;

We compute the aspect ratio of the new re-sized window, given its new width and height provided by the graphics sub-system to the callback function reshape().

glViewport(0, 0, width, height);

We set the viewport to cover the entire new re-sized window, in pixels. Try setting the viewport to cover only a quarter (lower-right quadrant) of the window via glViewport(0, 0, width/2, height/2).

```
glMatrixMode(GL_PROJECTION);

glLoadIdentity();

if (width >= height) {

gluOrtho2D(-1.0 * aspect, 1.0 * aspect, -1.0,
1.0);

} else {

gluOrtho2D(-1.0, 1.0, -1.0 / aspect, 1.0 /
aspect);
```

```
}
```

We set the aspect ratio of the clipping area to match the viewport. To set the clapping area, we first choose the operate on the projection matrix via glMatrixMode(GL_PROJECTION). OpenGL has two matrices, a projection matrix (which deals with camera projection such as setting the clipping area) and a model-view matrix (for transforming the objects from their local spaces to the common world space). We reset the projection matrix via glLoadIdentity().

Finally, we invoke gluOrtho2D() to set the clipping area with an aspect ratio matching the viewport. The shorter side has the range from -1 to +1, as illustrated below:

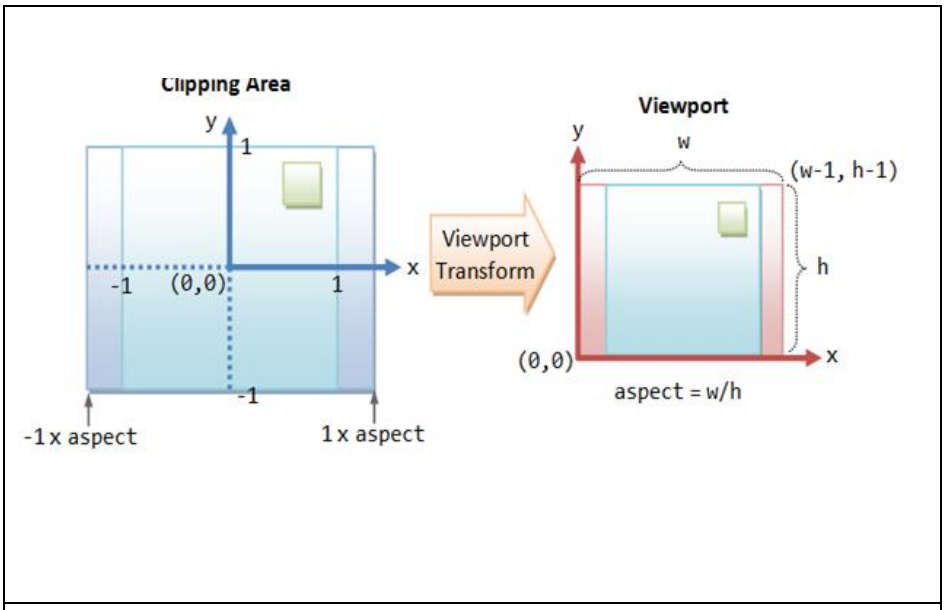


Fig.8-35: Clipping area.

Fig.8-35 shows clipping area and viewport to describe the same ratio for the clipping area and viewport to ensure the objects are not distorted. We need to register the reshape() callback handler with GLUT via glutReshapeFunc() in the main() as follows:

```
int main(intargc, char** argv) {

glutInitWindowSize(640, 480);

glutReshapeFunc(reshape);}
```

In the above main() function, we specify the initial window size to 640x480, which is non-squarest. Try re-sizing the window and observe the changes.

Note that the reshape() runs at least once when the window first appears. It is then called back whenever the window is re-shaped. On the other hand, the initGL() runs once (and only once); and the display() runs in response to window re-paint request (e.g., after the window is re-sized).

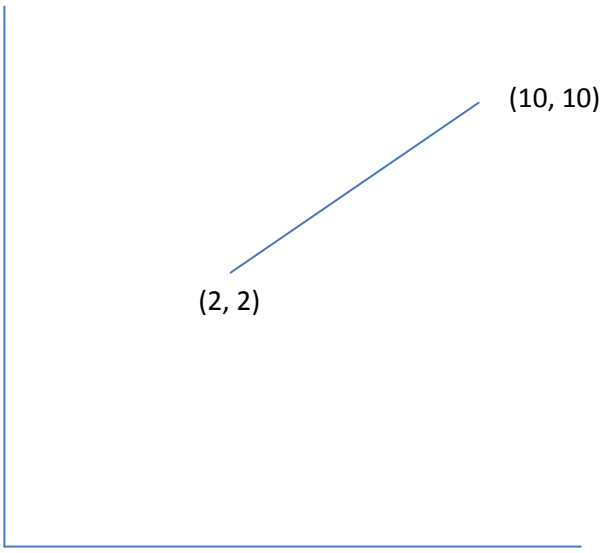
Bibliography

1- Foley, van Dam, et al., “Computer Graphics: Principles and Practice. Reading”, MA: Addison-Wesley Developers Press, 1990.
2- James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, “Computer Graphics: Principles & Practices”, Addison Wesley, 2nd edition in C, 1995
3- Edward Angel, “Interactive Computer Graphics: A Top-Down Approach with OpenGL”, 5th edition Addison-Wesley, 2008.

Exercises

1- Rotate the points (0,3), (1,1), and (4,5) under rotation angle 60 and then translate the resultant points using (1,5) transformation.

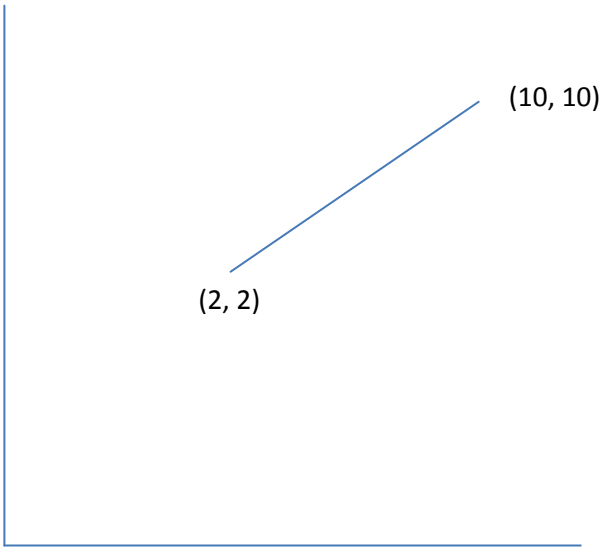
2- Let you have the following line:



3- Rotate the points (0, 3), (1, 1), and (4, 5) under rotation angle 60 and then translate them under (1, 5) transformation.

4- Let you have the following line:

   i- rotate the line according x-axis
   ii- rotate the line according y-axis
   iii- translate it using parameter (2,-2)
   iv- scaling it using parameter (5,10)



5. Suppose you have the point p' (3,-5, 4) after doing rotation using θ=50, find the original point before rotation.
6. Let you have the point p' (-4, 4, 1) after rotation and the point p (2,-1, 3) before rotation, find the angle that uses for this rotation.
7. Suppose you have the point p' (3,-5,4) after doing translation and point p(1,-1,-) before doing translation, find the translation parameter that uses in this example.
8. Let you have a point p (1, 1, and 3). Answer the following:
   - i- Rotate p around x axis.
   - ii- Translate the p using translation parameter (2,-1, 0).
   - iii- Tapering p using function f(x) $f(x) = x^2 + 1$
   - iv- Twisting p using function $\theta(y) = y^3$
   - v- Shearing using the parameters

   $s_{xy} = 3$
   $s_{xz} = 1$
   $s_{yx} = s_{yz} = -2$

$$s_{zx} = s_{zy} = 0$$

9. Let you have a point p (1, 3, -1). Answer the following:

   (i) Rotate *p* around x axis.

   (ii) Translate *p* using translation parameter (2,-1, 0).

   (iii) Let we have the point p' (-1, 3, 1) after twisting operation, find $\theta(y)$ that satisfy this deformation.

Solved Problems

1- A rectangle has coordinates (1, 1), (4, 1), (4, 3) and (1, 3). Find the coordinates of the image of the rectangle under the transformation represented by the matrix $\begin{pmatrix} 3 & -1 \\ -1 & 1 \end{pmatrix}$.

Solution

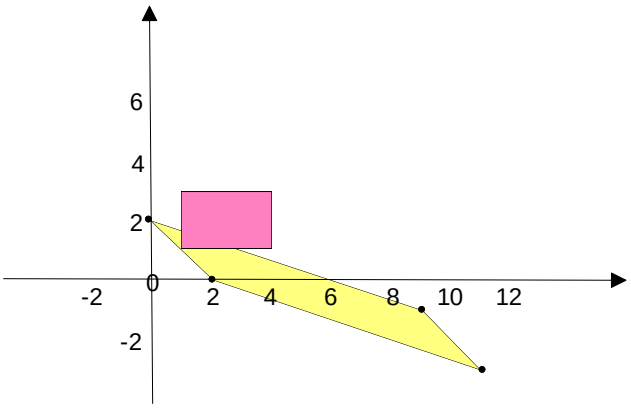
You could find the image of each vertex in turn by finding $\begin{pmatrix} 3 & -1 \\ -1 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \end{pmatrix}$, $\begin{pmatrix} 3 & -1 \\ -1 & 1 \end{pmatrix}\begin{pmatrix} 4 \\ 1 \end{pmatrix}$ etc.

However, it is more efficient to multiply the transformation matrix by a rectangular matrix containing the coordinates of each vertex:

$$\underbrace{\begin{pmatrix} 3 & -1 \\ -1 & 1 \end{pmatrix}}_{\text{transformation matrix}} \underbrace{\begin{pmatrix} 1 & 4 & 4 & 1 \\ 1 & 1 & 3 & 3 \end{pmatrix}}_{\text{matrix containing coordinates of each vertex}} = \begin{pmatrix} 2 & 11 & 9 & 0 \\ 0 & -3 & -1 & 2 \end{pmatrix}.$$

So the image has coordinates (2, 0), (11, -3), (9, -1) and (0, 2).

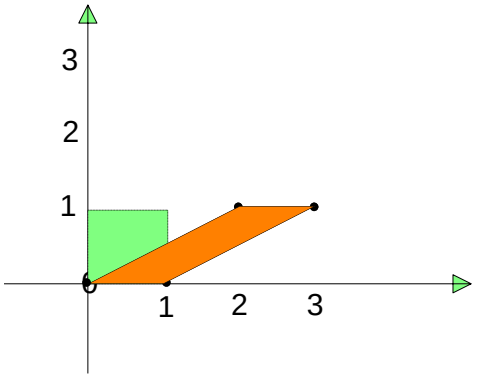
The diagram below shows the object and the image:



2- Find the image of the unit square under the transformation represented by the matrix $\begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$.

Solution

The image of (1, 0) is (1, 0) (i.e. the first column). The image of (0, 1) is (2, 1) (i.e. the second column). The image of (1, 1) is (3, 1) (i.e. add the entries in the top row and the bottom row together). We can show the unit square and its image in a diagram:



We notice that the points on the x-axis have not moved. This type of transformation is called a **shear**. Here the **invariant line** is the x-axis.

3- A transformation T is given by:

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

a) Find the image of the point A(3, 2).

b) Describe fully the transformation represented by T.

Solution

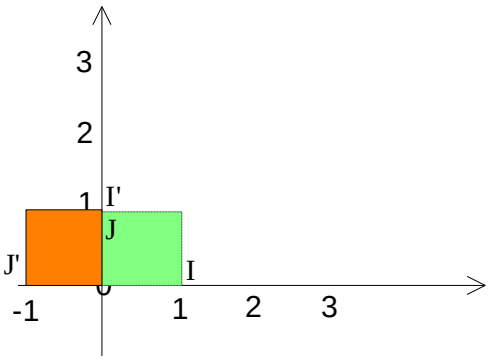
a) The image of A(3, 2) can be found by:

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

So the image of A is the point A'(-2, 3).

Note: The image of a point A is often denoted A'.

b) To describe the transformation we consider the image of the unit square:



The image of (1, 0) is (0, 1) (i.e. the first column)

The image of (0, 1) is (-1, 0) (i.e. the second column)

The image of (1, 1) is (-1, 1) (i.e. add the entries in the top row and the bottom row together).

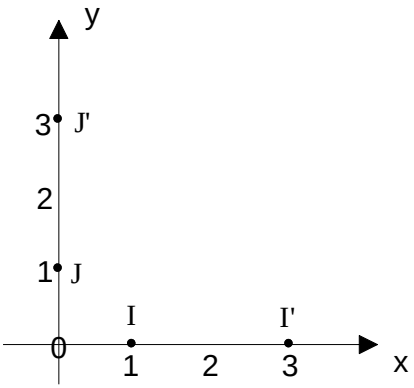
4- Find the matrix that represents an enlargement centre (0, 0), scale factor 3.

Solution

The image of the point I(1, 0) is (3, 0).

The image of the point J(0, 1) is (0, 3).

So the matrix is $\begin{pmatrix} 3 & 0 \\ 0 & 3 \end{pmatrix}$.



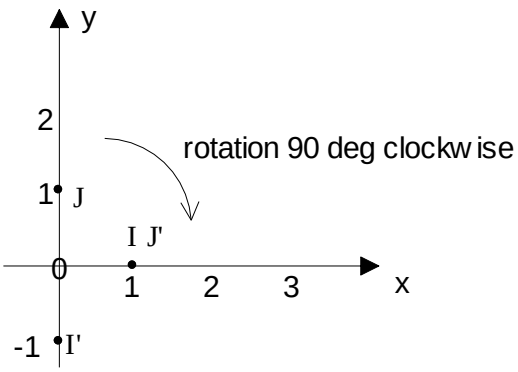
5- Find the matrix that represents a rotation centre (0, 0), 90 degrees clockwise.

Solution

The image of the point I(1, 0) is (0, -1).

The image of the point J(0, 1) is (1, 0).

So the matrix is $\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$.



6- Find the matrix of an anticlockwise rotation about the origin through 60°.

Solution

This matrix would be:

$$\begin{pmatrix} \cos 60 & -\sin 60 \\ \sin 60 & \cos 60 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & -\frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & \frac{1}{2} \end{pmatrix}.$$

7- Find the matrix that corresponds to a reflection in the line $y = 2x$.

Solution

Comparing the line $y = 2x$ with the form $y = (\tan\theta)x$, we see that $\tan\theta = 2$ so that $\theta = 63.43°$.

Therefore the matrix is:

$$\begin{pmatrix} \cos(2\times 63.43) & \sin(2\times 63.43) \\ \sin(2\times 63.43) & -\cos(2\times 63.43) \end{pmatrix} = \begin{pmatrix} -0.6 & 0.8 \\ 0.8 & 0.6 \end{pmatrix}.$$