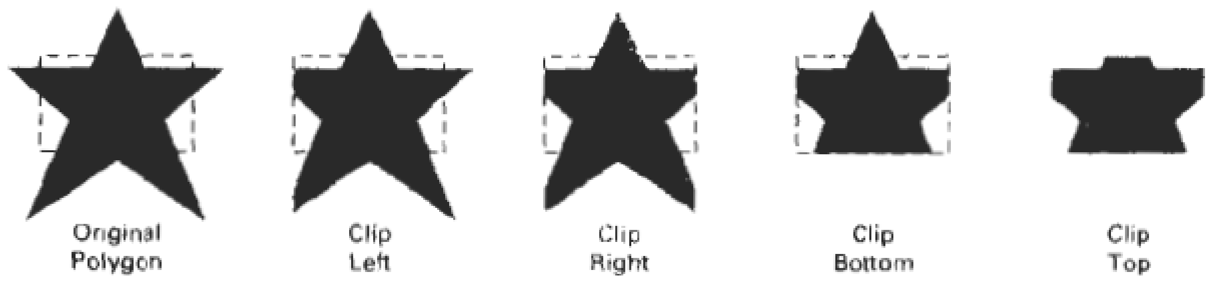
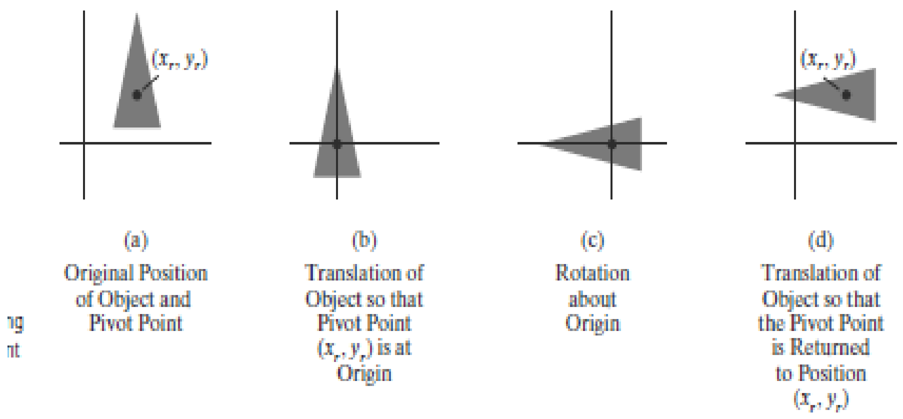
1. Explain Sutherland-Hodgeman polygon clipping algorithm with a neat diagram.

There are four possible cases when processing vertices in sequence around the perimeter of a polygon. As each pair of adjacent polygon vertices is passed to a window boundary clipper, we make the following tests: **(1)** If the first vertex is outside the window boundary and the second vertex is inside, both the intersection point of the polygon edge with the window boundary and the second vertex are added to the output vertex list. *(2)* If both input vertices are inside the window boundary, only the second vertex is added to the output vertex list. **(3)** li the first vertex is inside the ~ , i n J owbo undary and the second vertex is outside, only the edge intersection with the window boundary is added to the output vertex list. **(4)** If both input vertices are outside the window boundary, nothing is added to the output list. These four cases are illustrated in Fig for successive pairs of polygon vertices. Onct. all vertices have been processed for one clip window boundary, the output 11st of vertices is clipped against the next window boundary.



2. Explain how rotation is performed about an axis which is not parallel to any of the coordinate axes in 3D. [Arbitrary axis].

When a graphics package provides only a rotate function with respect to the coordinate origin, we can generate a two-dimensional rotation about any other pivot point (*xr* , *yr*) by performing the following sequence of translate-rotate-translate operations:


(x_r, y_r)
(a)
Original Position of Object and Pivot Point
(b)
Translation of Object so that Pivot Point (x_r, y_r) is at Origin
(c)
Rotation about Origin
(x_r, y_r)
(d)
Translation of Object so that the Pivot Point is Returned to Position (x_r, y_r)

- Translate the object so that the pivot-point position is moved to the coordinate origin.

- Rotate the object about the coordinate origin.
- Translate the object so that the pivot point is returned to its original position.

$$\begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r\sin\theta \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r\sin\theta \\ 0 & 0 & 1 \end{bmatrix}$$

This transformation sequence is illustrated in Figure 9. The composite transformation matrix for this sequence is obtained with the concatenation which can be expressed in the form

$$\mathbf{T}(x_r, y_r)\cdot\mathbf{R}(\theta)\cdot\mathbf{T}(-x_r, -y_r) = \mathbf{R}(x_r, y_r, \theta)$$

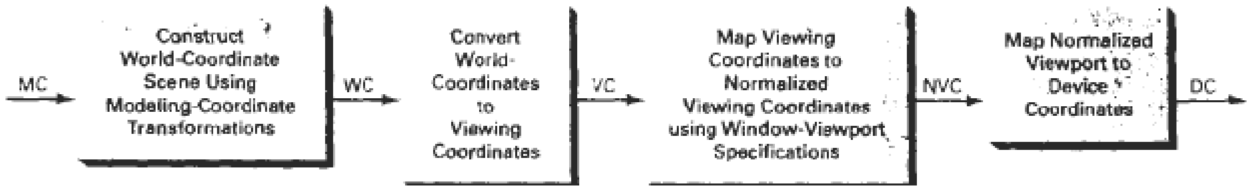
In general, a rotate function in a graphics library could be structured to accept parameters for pivot-point coordinates, as well as the rotation angle, and to generate automatically the rotation matrix of equation.

3. Explain 2D viewing pipeline with a neat block diagram.

A world-coordinate area selected for display is called a window. An area on a display device to which a window is mapped is called a viewport. The window defines ***whnt*** is to be viewed; the viewport defines ***where*** it is to be displayed. Often, windows and viewports are rectangles in standard position, with the rectangle edges parallel to the coordinate axes.

Next to obtain a particular orientation for the window, we can set up a two-dimensional viewing-coordinate system in the world-coordinate plane, and define a Window In the viewing-coordinate system. The viewing coordinate reference frame is used to provide a method for setting up arbitrary orientations for rectangular windows. Once the viewing reference frame is established, we can transform descriptions in world coordinates to viewing coordinates. We then define a \.viewport in normalized coordinates (in the range from **O** to I) and map the viewing-coordinate description of the scene to normalized coordinates. At the final step, .11I parts of the picture that he outside the viewport are clipped, and the contents of the viewport are transferred to device coordinates. Figure i1lustratt.s a rotated viewing-coordinate reference frame and the mapping to normalized coordinates.

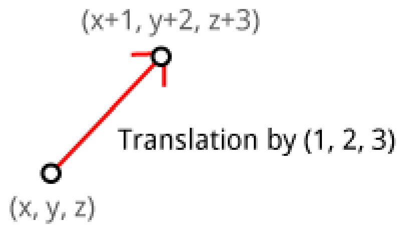
By changing the position of the viewport, we can view objects at different positions on the display area of an output device. Also, by varying the size of viewports, we can change the size and proportions of displayed objects.


MC
Construct World-Coordinate Scene Using Modeling-Coordinate Transformations
WC
Convert World-Coordinates to Viewing Coordinates
VC
Map Viewing Coordinates to Normalized Viewing Coordinates using Window-Viewport Specifications
NVC
Map Normalized Viewport to Device Coordinates
DC

4. Explain the OpenGL transformation functions.

Translation

To see why we're working with 4-by-1 vectors and subsequently 4-by-4 transformation matrices, let's see how a translation matrix is formed. A translation moves a vector a certain distance in a certain direction.

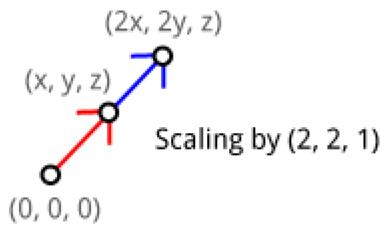


Can you guess from the multiplication overview what the matrix should look like to translate a vector by (X,Y,Z)?

Without the fourth column and the bottom 1 value a translation wouldn't have been possible.

Scaling

A scale transformation scales each of a vector's components by a (different) scalar. It is commonly used to shrink or stretch a vector as demonstrated below.

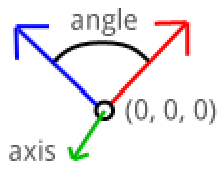


If you understand how the previous matrix was formed, it should not be difficult to come up with a matrix that scales a given vector by (SX,SY,SZ).

If you think about it for a moment, you can see that scaling would also be possible with a mere 3-by-3 matrix.

Rotation

A rotation transformation rotates a vector around the origin (0,0,0) using a given *axis* and *angle*. To understand how the axis and the angle control a rotation, let's do a small experiment.



Put your thumb up against your monitor and try rotating your hand around it. The object, your hand, is rotating around your thumb: the rotation axis. The further you rotate your hand away from its initial position, the higher the rotation angle.

In this way the rotation axis can be imagined as an arrow an object is rotating around. If you imagine your monitor to be a 2-dimensional XY surface, the rotation axis (your thumb) is pointing in the Z direction.

Objects can be rotated around any given axis, but for now only the X, Y and Z axis are important. You'll see later in this chapter that any rotation axis can be established by rotating around the X, Y and Z axis simultaneously.The matrices for rotating around the three axes are specified here. The rotation angle is indicated by the theta (θθ).

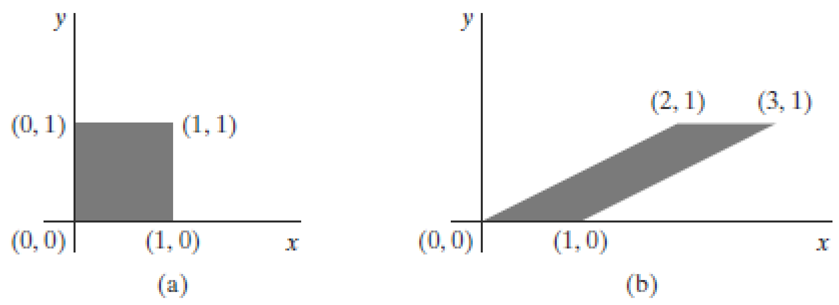
5. Explain 2D reflection and shear transformations.

Shear

A transformation that distorts the shape of an object such that the transformed shape appears as if the object were composed of internal layers that had been caused to slide over each other is called a **shear.** Two common shearing transformations are those that shift coordinate *x* values and those that shift *y* values.

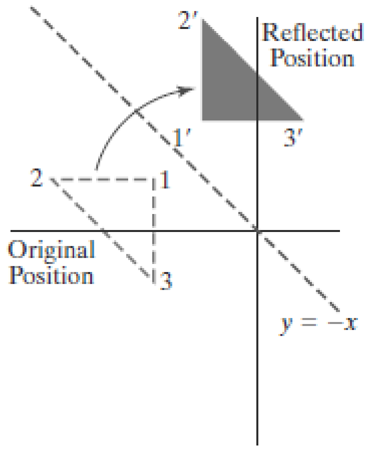
$$\begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$x' = x + sh_x \cdot y, \qquad y' = y$$

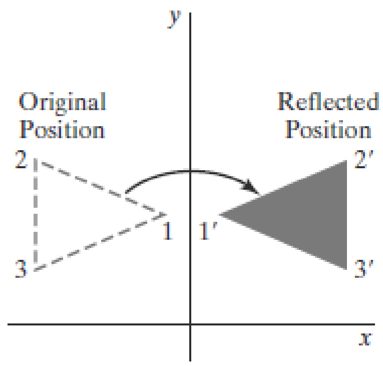


Reflection

A transformation that produces a mirror image of an object is called a **reflection.** For a two-dimensional reflection, this image is generated relative to an **axis of reflection** by rotating the object 180◦ about the reflection axis. We can choose an axis of reflection in the *xy* plane or perpendicular to the *xy* plane. When the reflection axis is a line in the *xy* plane, the rotation path about this axis is in a plane perpendicular to the *xy* plane. For reflection axes that are perpendicular to the *xy* plane, the rotation path is in the *xy* plane. Some examples of common reflections follow.

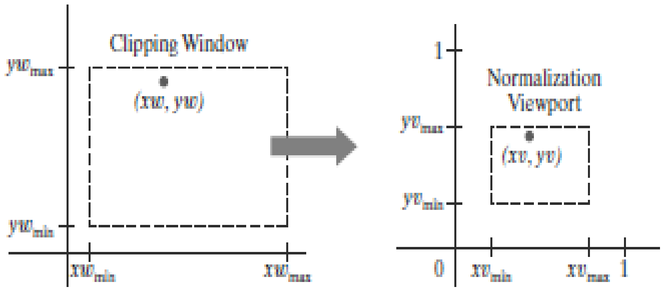


$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

6. Explain mapping of clipping window to a viewport.



$$\frac{xv - xv_{min}}{xv_{max} - xv_{min}} = \frac{xw - xw_{min}}{xw_{max} - xw_{min}}$$

$$\frac{yv - yv_{min}}{yv_{max} - yv_{min}} = \frac{yw - yw_{min}}{yw_{max} - yw_{min}}$$

Solving these expressions for the viewport position (xv, yv), we have

$$xv = s_x xw + t_x$$
$$yv = s_y yw + t_y$$

where the scaling factors are

$$s_x = \frac{xv_{max} - xv_{min}}{xw_{max} - xw_{min}}$$

$$s_y = \frac{yv_{max} - yv_{min}}{yw_{max} - yw_{min}}$$

and the translation factors are

$$t_x = \frac{xw_{max}xv_{min} - xw_{min}xv_{max}}{xw_{max} - xw_{min}}$$

$$t_y = \frac{yw_{max}yv_{min} - yw_{min}yv_{max}}{yw_{max} - yw_{min}}$$

Because we are simply mapping world-coordinate positions into a viewport that is positioned near the world origin, we can also derive Equations 3 using any transformation sequence that converts the rectangle for the clipping window into the viewport rectangle. For example, we could obtain the transformation from world coordinates to viewport coordinates with the following sequence:

- Scale the clipping window to the size of the viewport using a fixed-point position of (*xw*min, *yw*min).
- Translate (*xw*min, *yw*min) to (*xv*min, *yv*min). The scaling transformation in step (1) can be represented with the twodimensional matrix

7. Explain line clipping using Cohen-Sutherland algorithm

```
#include<stdio.h>
#include<GL/glut.h>
#define outcode int
double xmin=50,ymin=50,xmax=100,ymax=100;
double xvmin=200,yvmin=200,xvmax=300,yvmax=300;
const int RIGHT=8;
const int LEFT=2;
const int TOP=4;
const int BOTTOM=1;
outcode ComputeOutCode(double x,double y);
void CohenSutherLandLineClipAndDraw(double x0,double y0,double x1,double y1)
{
    outcode outcode0,outcode1,outcodeOut;
    bool accept=false,done=false;
    outcode0=ComputeOutCode(x0,y0);
    outcode1=ComputeOutCode(x1,y1);
    do
    {
            if(!(outcode0|outcode1))
            {
                    accept=true;
                    done=true;
            }
            else if(outcode0 & outcode1)
                    done=true;
            else
            {
                    double x,y;
                    outcodeOut=outcode0?outcode0:outcode1;
                    if(outcodeOut & TOP)
                    {
                            x=x0+(x1-x0)*(ymax-y0)/(y1-y0);
                            y=ymax;
                    }
                    else if(outcodeOut & BOTTOM)
                    {
                            x=x0+(x1-x0)*(ymin-y0)/(y1-y0);
                            y=ymin;
                    }
                    else if(outcodeOut & RIGHT)
                    {
                y=y0+(y1-y0)*(xmax-x0)/(x1-x0);
                      x=xmax;
                    }
                    else
                    {
                            y=y0+(y1-y0)*(xmin-x0)/(x1-x0);
                      x=xmin;
                    }
                    if(outcodeOut==outcode0)
                    {
                            x0=x;
                            y0=y;
                            outcode0=ComputeOutCode(x0,y0);
                    }
                            else
                            {
                                    x1=x;
                                    y1=y;
```

```
                                        outcode1=ComputeOutCode(x1,y1);
                              }
                    }
          }
          while(!done);
          if(accept)
          {
                    double sx=(xvmax-xvmin)/(xmax-xmin);
                    double sy=(yvmax-yvmin)/(ymax-ymin);
                    double vx0=xvmin+(x0-xmin)*sx;
                    double vy0=yvmin+(y0-ymin)*sy;
                    double vx1=xvmin+(x1-xmin)*sx;
                    double vy1=yvmin+(y1-ymin)*sy;

                    glColor3f(1.0,0.0,0.0);
                    glBegin(GL_LINE_LOOP);
                    glVertex2f(xvmin,yvmin);
                    glVertex2f(xvmax,yvmin);
                    glVertex2f(xvmax,yvmax);
                    glVertex2f(xvmin,yvmax);
                    glEnd();

                    glColor3f(0.0,0.0,1.0);
                    glBegin(GL_LINES);
                    glVertex2d(vx0,vy0);
                    glVertex2d(vx1,vy1);
                    glEnd();
          }
}
outcode ComputeOutCode(double x,double y)
{
          outcode code=0;
          if(y>ymax)
                    code|=TOP;
          else if(y<ymin)
                    code|=BOTTOM;
          if(x>xmax)
                    code|=RIGHT;
          else if(x<xmin)
                    code|=LEFT;
          return code;
}
void display()
{
          double x0=120,y0=10,x1=40,y1=130;
          glClear(GL_COLOR_BUFFER_BIT);

          glColor3f(1.0,0.0,0.0);
          glBegin(GL_LINES);
          glVertex2d(x0,y0);
          glVertex2d(x1,y1);
          glVertex2d(60,20);
```

```
            glVertex2d(80,120);
            glEnd();

            glColor3f(0.0,0.0,1.0);
            glBegin(GL_LINE_LOOP);
            glVertex2f(xmin,ymin);
            glVertex2f(xmax,ymin);
            glVertex2f(xmax,ymax);
            glVertex2f(xmin,ymax);
            glEnd();
            CohenSutherLandLineClipAndDraw(x0,y0,x1,y1);
            CohenSutherLandLineClipAndDraw(60,20,80,120);
  glFlush();
}
void myinit()
{
            glClearColor(1.0,1.0,1.0,1.0);
            glColor3f(1.0,0.0,0.0);
            glPointSize(1.0);
            glMatrixMode(GL_PROJECTION);
            glLoadIdentity();
            gluOrtho2D(0.0,499.0,0.0,499.0);
}
void main(int argc,char**argv)
{
            glutInit(&argc,argv);
            glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
            glutInitWindowSize(500,500);
            glutInitWindowPosition(0,0);
            glutCreateWindow("cohen suderland Line clipping Algorithm");
            glutDisplayFunc(display);
            myinit();
            glutMainLoop();
}
```

8. Write a program to model a color cube and spin it using OpenGL transformation functions.

```
#include<stdlib.h>
#include<GL/glut.h>

GLfloat vertices[][3]={{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
{1.0,1.0,-1.0},{-1.0,1.0,-1.0},{-1.0,-1.0,1.0},
{1.0,-1.0,1.0},{1.0,1.0,1.0},{-1.0,1.0,1.0}};

GLfloat colors[][3]={{1.0,0.0,0.0},{1.0,1.0,0.0},
{1.0,0.0001,0.0},{0.0,1.0001,0.0},{0.0,0.0,1.0},
{1.0,1.0,0.0},{1.0,0.0,0.0},{1.0,0.0,10.0}};

void polygon(int a,int b,int c,int d)
{
    glBegin(GL_POLYGON);
    //glColor3fv(colors[a]);
    glVertex3fv(vertices[a]);
```

```
     //glColor3fv(colors[b]);
     glVertex3fv(vertices[b]);
     //glColor3fv(colors[c]);
     glVertex3fv(vertices[c]);
     //glColor3fv(colors[d]);
     glVertex3fv(vertices[d]);
     glEnd();
}
 void colorcube(void)
 {
      polygon(0,3,2,1);
      glcolor3f(1.0,1.0,0.0);
      polygon(2,3,7,6);
      glcolor3f(1.0,0.0,0.0);
      polygon(0,4,7,3);
      glcolor3f(1.0,0.0,1.0);
      polygon(1,2,6,5);
      glcolor3f(1.0,0.0,0.0);
      polygon(4,5,6,7);
      glcolor3f(1.0,0.0,1.0);
      polygon(0,1,5,4);
      glcolor3f(1.0,0.0,0.0);
 }
 static GLfloat theta[]={0.0,0.0,0.0};
 static GLint axis=2;
void display(void)
{
     glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
     glLoadIdentity();
     glRotatef(theta[0],1.0,0.0,0.0);
     glRotatef(theta[1],0.0,1.0,0.0);
     glRotatef(theta[2],0.0,0.0,1.0);
     colorcube();
     glFlush();
     glutSwapBuffers();
}
void spinCube()
{
theta[axis]+=1.0;
if(theta[axis]>360.0)theta[axis]-=360.0;
glutPostRedisplay();
}
void mouse(int btn,int state,int x,int y)
{
     if(btn==GLUT_LEFT_BUTTON&&state==GLUT_DOWN)axis=0;
     if(btn==GLUT_MIDDLE_BUTTON&&state==GLUT_DOWN)axis=1;
     if(btn==GLUT_RIGHT_BUTTON&&state==GLUT_DOWN)axis=2;
}
void myReshape(int w,int h)
```

```
{
     glViewport(0,0,w,h);
     glMatrixMode(GL_PROJECTION);
     glLoadIdentity();
     glOrtho(-2.0,2.0,-2.0,2.0,-10.0,10.0);
     glMatrixMode(GL_MODELVIEW);
}

void main(int argc,char **argv)
{
          glutInit(&argc,argv);
          glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
          glutInitWindowSize(500,500);
          glutCreateWindow("roatating a color cube");
          glutReshapeFunc(myReshape);
          glutDisplayFunc(display);
          glutIdleFunc(spinCube);
          glutMouseFunc(mouse);
          glEnable(GL_DEPTH_TEST);
          glutMainLoop();
}
```