Unit 2
Geometrical Transformations

Two Dimensional Geometric Transformations

In computer graphics, transformations of 2D objects are essential to many graphics applications. The transformations are used directly by application programs and within many graphics subroutines in application programs. Many applications use the geometric transformations to change the position, orientation, and size or shape of the objects in drawing. Rotation, Translation and scaling are three major transformations that are extensively used by all most all graphical packages or graphical subroutines in applications. Other than these, reflection and shearing transformations are also used by some graphical packages.

2D Translation

A translation is applied to an object by re-positioning it along a straight line path from one co-ordinate location to another. We translate a two-dimensional point by adding translation distances, t_x, t_y to the respective co-ordinate values of original co-ordinate position (x, y) to move the point to a new position (x', y') as:

$$x' = x + t_x$$

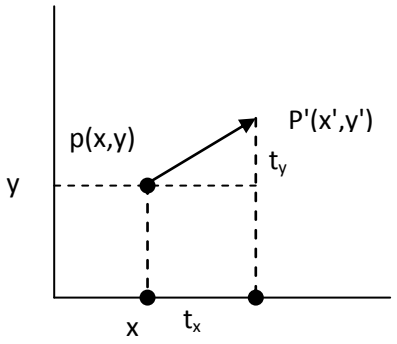
$$y' = x + t_y$$

The translation distance pair (t_x, t_y) is known as translation vector or shift vector. We can express translation equations as matrix representations as

$$P = \begin{bmatrix} x \\ y \end{bmatrix} \qquad P' = \begin{bmatrix} x' \\ y' \end{bmatrix} \qquad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$\therefore P' = P + T$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

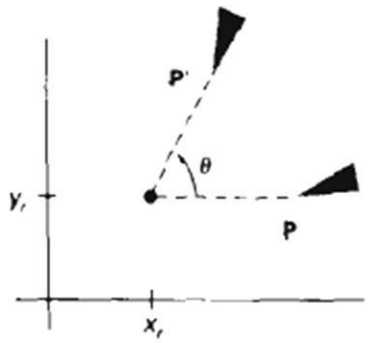


Sometimes matrix transformations are represented by co-ordinate rows vector instead of column vectors as,

$$P = (x, y) \qquad T = (t_x, t_y) \qquad P' = P + T.$$

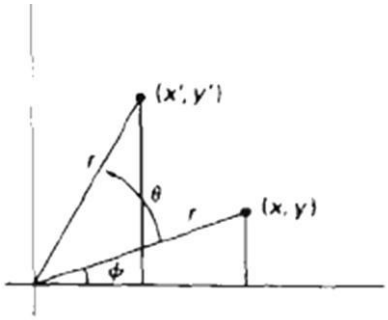
For translation of any object in Cartesian plane, we transform the distinct co-ordinates by the translation vector and re-draw image at the new transformed location.

2D Rotation



The 2D rotation is applied to re-position the object along a circular path in XY-plane. To generate rotation, we specify a **rotation angle** θ and a pivot point (rotation point) about which the object is to be rotated. Rotation can be made by angle θ either clockwise or anticlockwise direction. The positive θ rotates object in anti-clockwise direction and the negative value of θ rotates the object in clock-wise direction. A line perpendicular to rotating plane and passing through pivot point is called **axis of rotation**.

Let P(x, y) is a point in XY-plane which is to be rotated with angle θ. Also let OP = r (As in figure below) is constant distance from origin. Let r makes angle ϕ with positive X-direction as shown in figure.



When OP is rotated through angle θ and taking origin as pivot point for rotation, OP' makes angle $\theta+\phi$ with X-axis.

Now,

$$x' = r\cos(\phi+\theta) = r\cos\phi\cos\theta - r\sin\phi\sin\theta$$
$$y' = r\sin(\phi+\theta) = r\sin\phi\cos\theta + r\cos\phi\sin\theta$$

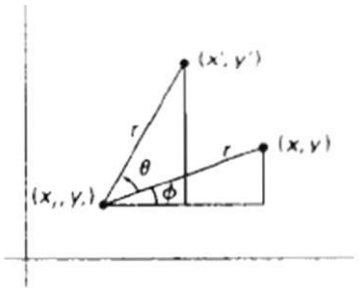
But, $r\cos\phi = x, r\sin\phi = y$ therefore we get

$x' = x\cos\theta - y\sin\theta$ ----------1

$y' = x\sin\theta + y\cos\theta$ ----------2 which are equation for rotation of (x, y) with angle θ and taking pivot as origin.

<u>Rotation from any arbitrary pivot point (x_r, y_r)</u>

- Let Q(x_r, y_r) is pivot point for rotation.
- P(x, y) is co-ordinate of point to be rotated by angle θ.
- Let ϕ is the angle made by QP with X-direction. θ.



Then angle made by QP' with X-direction is $\theta + \phi$

Hence,

$\cos(\phi + \theta) = (x' - x_r) / r$

or $r\cos(\phi + \theta) = (x' - x_r)$

or $x' - x_r = r\cos\phi\cos\theta - r\sin\phi\sin\theta$

since $r\cos\phi = x - x_r, rsiin\phi = y - y_r$

$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta \quad \text{..... (1)}$$

Similarly,

$\sin(\phi + \theta) = (y' - y_r) / r$

or $r\sin(\phi + \theta) = (y' - y_r)$

or $y' - y_r = r\sin\phi\cos\theta + r\cos\phi\sin\theta$

since $r\cos\phi = x - x_r, rsiin\phi = y - y_r$

$$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta \quad \text{..... (2)}$$

These equations (1) and (2) are the equations for rotation of a point (x, y) with angle θ taking pivot point (x_r, y_r).

The rotation about pivot point (x_r, y_r) can also be achieved by sequence of translation, rotation about origin and reverse translation.

- Translate the point (x_r, y_r) and P(x, y) by translation vector ($-x_r$, $-y_r$) which translates the pivot to origin and P(x, y) to ($x-x_r$, $y-y_r$).
- Now apply the rotation equations when pivot is at origin to rotate the translated point ($x-x_r$, $y-y_r$) as:

$$x_1 = (x - x_r)\cos\theta - (y - y_r)\sin\theta$$

$$y_1 = (x - x_r)\sin\theta + (y - y_r)\cos\theta$$

- Re-translate the rotated point (x_1, y_1) with translation vector (x_r, y_r) which is reverse translation to original translation. Finally we get the equation after successive transformation as

$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta \quad \text{..... (1)}$$

$$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta \quad \text{..... (2)}$$ which are the equations for rotation of (x, y) from the pivot point (x_r, y_r).

2D Scaling

A scaling transformation alters the size of the object. This operation can be carried out for polygon by multiplying the co-ordinate values (x, y) of each vertex by scaling factor s_x and s_y to produce transformed co-ordinates (x', y').

i.e. $x' = x.s_x$ and $y' = y.s_y$

Scaling factor s_x scales object in x- direction and s_y scales in y- direction. If the scaling factor is less then 1, the size of object is decreased and if it is greater than 1 the size of object is increased. The scaling factor = 1 for both direction does not change the size of the object.

- If both scaling factors have same value then the scaling is known as **uniform scaling**.
- If the value of s_x and s_y are different, then the scaling is known as **differential scaling**. The differential scaling is mostly used in the graphical package to change the shape of the object.

The matrix equation for scaling is:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} . \begin{bmatrix} x \\ y \end{bmatrix}$$ i.e. P' = S.P

Matrix Representations and Homogeneous co-ordinates

Many graphics applications involve sequences of geometric transformations. An animation, for example, might require an objet to be translated and rotated at each increment of the motion. In design and picture construction applications, we perform translations, rotations, and scalings to fit the picture components into their proper positions. Here we discuss how such transformation sequences can be efficiently processed using matrix notation.

- The homogeneous co-ordinate system provides a uniform framework for handling different geometric transformations, simply as multiplication of matrices.
- To express any two-dimensional transformation as a matrix multiplication, we represent each Cartesian coordinate position *(x, y)* with the homogeneous coordinate triple *(x_h, y_h, h),* where x = x_h/h , y = y_h/h .(h is 1 usually for 2D case).
- By using this homogeneous co-ordinate system a 2D point would be (x, y, 1).
    - The matrix formulation for **2D translation** for T(t_x, t_y) is :

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$ By which we can get $x' = x + t_x$ and $y' = x + t_y$.

    - For **2D Rotation**: R(θ)
        - **About origin** the homogeneous matrix equation will be

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

Which gives the

$x' = x\cos\theta - y\sin\theta$ ---------- 1

$y' = x\sin\theta + y\cos\theta$ ---------- 2 which are equation for rotation of (x, y) with angle θ and taking pivot as origin.

- Rotation **about an arbitrary fixed pivot point (x_r,y_r)**
  For a fixed pivot point rotation, we can apply composite transformation as Translate fixed point (x_r, y_r) to the co-ordinate origin by T(-x_r,-y_r). Rotate with angle θ → R(θ) .Translate back to original position by T(x_r,y_r).This composite transformation is represented as:
  P' = T(x_r, y_r).R(θ).T(-x_r, -y_r). This can be represented in matrix equation using homogeneous co-ordinate system as,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Expanding this equation we get

$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta$ ..... (1)

$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta$ ..... (2)

These are actually the equations for rotation of (x, y) from the pivot point (x_r, y_r).

- **Scaling with scaling factors** (s_x, s_y)

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This exactly gives the equations

x' = x.s_x and y' = y.s_y

Composite transformations

Translations

If two successive translation vectors (t_{x1}, t_{y1}) and (t_{x2}, t_{y2}) are applied to a coordinate position P, the final transformed location P' is calculated as

P' = T(t_{x2}, t_{y2}).{T(t_{x1}, t_{y1}).P }
= {T(t_{x2}, t_{y2}).T(t_{x1}, t_{y1})}.P

Where P and P are represented as homogeneous-coordinate column vectors.

Verification:

$$\begin{bmatrix} 1 & 0 & t_{x2} \\ 0 & 1 & t_{y2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{x1} \\ 0 & 1 & t_{y1} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{x1} + t_{x2} \\ 0 & 1 & t_{y1} + t_{y2} \\ 0 & 0 & 1 \end{bmatrix}$$

T(t_{x2}, t_{y2}).T(t_{x1}, t_{y1}) = T(t_{x1} + t_{x2}, t_{y1} + t_{y2})

which demonstrates that two successive translations are additive.

Rotations

Two successive rotations applied to point **p** produce the transformed position

$$P' = R(\theta_2).\{R(\theta_1).P\}$$
$$= \{R(\theta_2).R(\theta_1)\}.P$$

By multiplying the two rotation matrices, we can verify that two successive rotations are additive:

$$R(\theta_2).R(\theta_1) = R(\theta_1 + \theta_2)$$

so that the final rotated coordinates can be calculated with the composite rotation matrix as

$$P' = R(\theta_1 + \theta_2).P$$

Scalings

Concatenating transformation matrices for two successive scaling operations produces the following composite scaling matrix:

$$\begin{bmatrix} s_{x2} & 0 & 0 \\ 0 & s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{x1} & 0 & 0 \\ 0 & s_{y1} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{x1} \cdot s_{x2} & 0 & 0 \\ 0 & s_{y1} \cdot s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$S(s_{x2}, s_{y2}) \cdot S(s_{x1}, s_{y1}) = S(s_{x1} \cdot s_{x2}, s_{y1} \cdot s_{y2})$$

The resulting matrix in this case indicates that successive scaling operations are multiplicative.

General Fixed point scaling

Objects transformed with standard scaling equation are scaled as well as re-positioned. Scaling factor less then 1 moves object closer to the origin whereas value greater than 1 moves object away from origin.

- To control the location of scaled object we can choose the position called fixed point. Let co-ordinates of fixed point = (x_f, y_f). It can be any point on the object.
- A polygon is then scaled relative to (x_f, y_f) by scaling the distance from each vertex to the fixed point.
- For a vertex with co-ordinate (x, y), the scaled co-ordinates (x', y') are calculated as:

$$x' = x_f + (x-x_f)s_x$$
$$y' = y_f + (y-y_f)s_y$$

or equivalently,

$$x' = x.s_x + (1-s_x)x_f$$
$$y' = y.s_y + (1-s_y)y_f$$

Where $(1-s_x)x_f$ *and* $(1-s_y)y_f$ are constant for all points in object.

To represent fixed point scaling using matrix equations in homogeneous co-ordinate system, we can use composite transformation as in fixed point rotation.

1. Translate object to the origin so that (x_f, y_f) lies at origin by $T(-x_f, -y_f)$.
2. Scale the object with (s_x, s_y)
3. Re-translate the object back to its original position so that fixed point (x_f, y_f) moves to its original position. In this case translation vector is $T(x_f, y_f)$.

$$\therefore P' = T(x_f, y_f).S(s_x, s_y).T(-x_f, -y_f).P$$

The homogeneous matrix equation for fixed point scaling is

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_f(1-s_x) \\ 0 & s_y & y_f(1-s_y) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Directive scaling

Standard and fixed point scaling scales object along x and y axis only. Directive scaling scales the object in any direction.

Let S_1 and S_2 are given directions for scaling at angle Θ from co-ordinate axes as in figure below

1. First perform the rotation so that directions S_1 and S_2 coincide with x and y-axes.
2. Then the scaling transformation is applied to scale the object by given scaling factors (s_1, s_2).
3. Re-apply the rotation in opposite direction to return to their original orientation.

For any point P in object, the directive scaling position P' is given by following composite transformation.

$$P' = R^{-1}(\theta).S(s_1, s_2).R(\theta).P$$

For which the homogeneous matrix equation is

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Other transformations

Reflection

A reflection is a transformation that produces a mirror image of an object. In 2D-transformation, reflection is generated relative to an axis of reflection. The reflection of an object to an relative axis of reflection, is same as 180° rotation about the reflection axis.

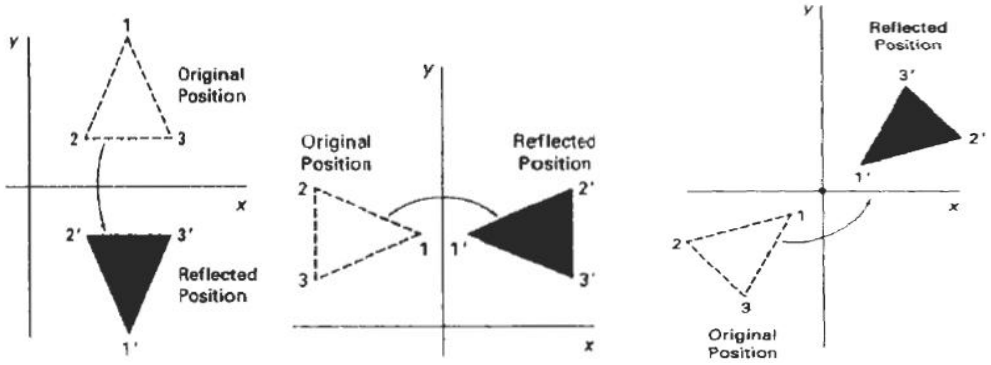


Fig: reflection of an object about an x-axis, y-axis and axis perpendicular to xy-plane (passing through origin) respectively.

a) Reflection about X-axis: The line representing x-axis is y = 0. The reflection of a point P(x, y) on x-axis, changes the y-coordinate sign i.e. Reflection about x-axis, the reflected position of P(x, y) will be P'(x, -y). Hence, reflection in x-axis is accomplished with transformation equation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$ gives the reflection of a point.

To reflect a 2D object, reflect each distinct points of object by above equation, then joining the points with straight line, redraws the image for reflected image.

b) Reflection about Y-axis: The line representing y-axis is x = 0. The reflection of a point P(x, y) on y-axis changes the sign of x-coordinate i.e. P(x, y) changes to P'(-x, y). Hence reflection of a point on y-axis is obtained by following matrix equation

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

For any 2D object, reflect each point and re-draw image joining the reflected images of all distinct points.

c) Reflection on an arbitrary axis: The reflection on any arbitrary axis of reflection can be achieved by sequence of rotation and co-ordinate axes reflection matrices.

- First, rotate the arbitrary axis of reflection so that the axis of reflection and one of the co-ordinate axes coincide.
- Reflect the image on the co-ordinate axis to which the axis of reflection coincides.
- Rotate the axis of reflection back to its original position.

For example consider a line y = x for axis of reflection, the possible sequence of transformation for reflection of 2D object is

d) Reflection about origin: The reflection on the line perpendicular to xy-plane and passing through flips x and y co-ordinates both. So sign of x and y co-ordinate value changes. The equivalent matrix equation for the point is:

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

Shearing

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 shear.

X-direction Shear: An X-direction shear relative to x-axis is produced with transformation matrix equation.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$ which transforms $x' = x + sh_x$ and y' = y

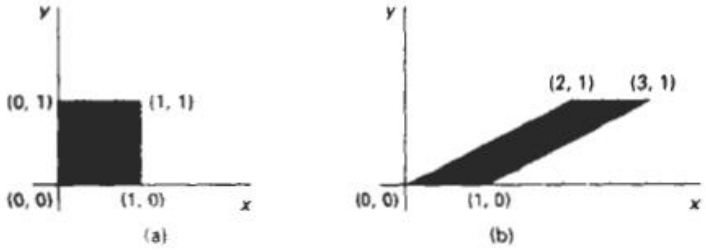


Fig: A unit square (a) is converted to a parallelogram (b) using the *x*-direction shear matrix with $sh_x = 2$.

Y-direction shear: Any-direction shear relative to y-axis is produced by following transformation equations.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$ which transforms $x' = x$ and $y' = x.sh_y + y$

```
/*
Example: Program for translation of 2D objects using homogeneous co-ordinates representation of
object for its vertices which translates the rectangle by given translation vector.
*/
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<graphics.h>
void matrixmultiply(int T[3][3],int V[3][5],int r[3][5]);
void main()
{

  int gdriver, gmode, errorcode;
  int i,j,k;
  int vertex[3][5]={{100,100,200,200,100},
                    {100,200,200,100,100},
                    {1,1,1,1,1},
                    };
  int translate[3][3] ={{1,0,100},{0,1,200},{0,0,1}};
  int result[3][5];
  gdriver=DETECT;/* request auto detection */
        /* initialize graphics and local variables */
  initgraph(&gdriver, &gmode, "\\tc\\bgi");
        /* read result of initialization */
  errorcode = graphresult();
        /* an error occurred */
```

```
  if (errorcode != grOk)
  {
    printf("Graphics error: %s\n", grapherrormsg(errorcode));
    printf("Press any key ....");
    getch();
                /* terminate with an error code */
  }

        setbkcolor(2);
        for(i=0;i<4;i++)
        {
                setcolor(BLUE);
                line(vertex[0][i],vertex[1][i],vertex[0][i+1],vertex[1][i+1]);

        }

        printf("Press any key for the translated line.....\n");
        getch();
        matrixmultiply(translate,vertex,result);
        for(i=0;i<4;i++)
        {
                setcolor(YELLOW);
                line(result[0][i],result[1][i],result[0][i+1],result[1][i+1]);

        }
        getch();
        closegraph();
}
void matrixmultiply(int translate[3][3],int vertex[3][5],int result[3][5])
{
        for(int i=0;i<=3;i++)
        {
                for(int j=0;j<=5;j++)
                {
                        result[i][j]=0;
                        for(int k=0;k<=3;k++)
                                result[i][j]+=translate[i][k]*vertex[k][j];
                }
        }
}
```

2D viewing

Two Dimensional viewing is the formal mechanism for displaying views of a picture on an output device. Typically, a graphics package allows a user to specify which part of a defined picture is to be displayed and where that part is to be placed on the display device. Any convenient Cartesian coordinate system, referred to as the world-coordinate reference frame, can be used to define the picture. For a two-dimensional picture, a view is selected by specifying a subarea of the total picture area. The picture parts within the selected areas are then mapped onto specified areas of the device coordinates. Transformations from world to device co ordinates involve translation, rotation, and scaling operations, as well as procedures for deleting those parts of the picture that are outside the limits of a selected display area.

Window: A world-coordinate area selected for display

Viewport: An area on a display device to which a window is mapped

"The window defines ***what*** is to be viewed; the viewport defines ***where*** it is to be displayed"

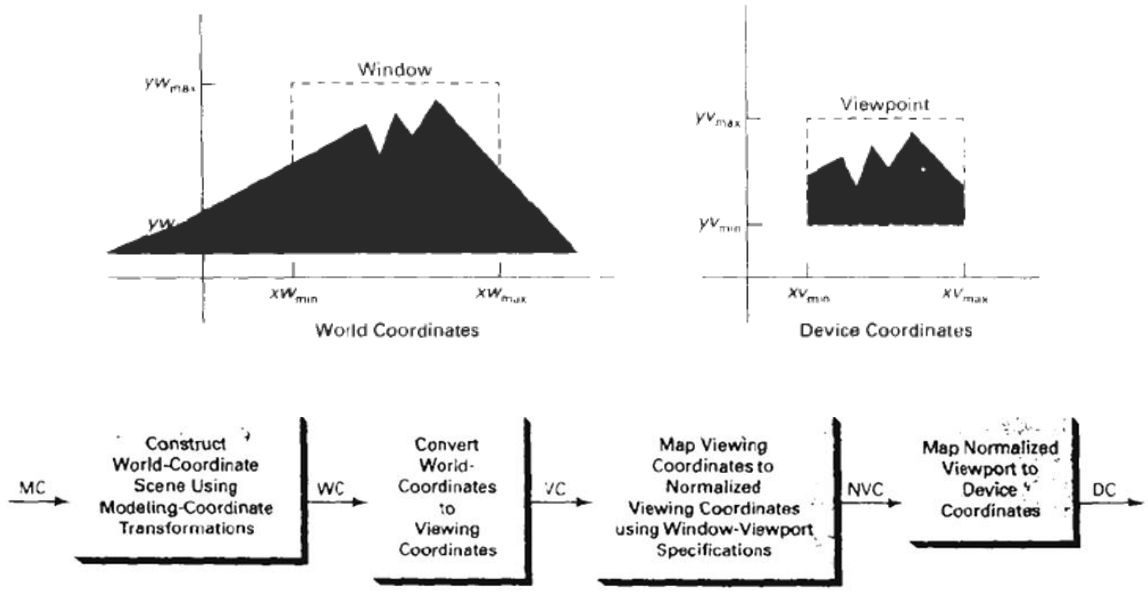


Fig: 2D viewing pipeline

Window-to-Viewport Coordinate Transformation

Once object descriptions have been transferred to the viewing reference frame, we choose the window extents in viewing coordinates and select the viewport limits in normalized coordinates. Object descriptions are then transferred to normalized device coordinates. We do this using a transformation that maintains the same relative placement of objects in normalized space as they had in viewing coordinates.

A point at position (xw, yw) in the window is mapped into position *(xv,* yv) in the associated viewport.

To maintain the same relative placement in the viewport as in the window, we require that:

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

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

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

$$xv = xv_{min} + (xw - xw_{min})sx$$

$$yv = yv_{min} + (yw - yw_{min})sy$$

Where the scaling factors are

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

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

Equations above can also be derived with a set of transformations that converts the window area into the viewport area. This conversion is performed with the following sequence of transformations:

1. Perform a scaling transformation using a fixed-point position of (xw_{min}, yw_{min}) that scales the window area to the size of the viewport.
2. Translate the scaled window area to the position of the viewport.

Relative proportions of objects are maintained if the scaling factors are the same (sx = sy). Otherwise, world objects will be stretched or contracted in either the *x* or *y* direction when displayed on the output device.

Clipping

Any procedure that identifies those portions of a picture that are either inside or outside of a specified region of space is referred to as a **clipping algorithm**, or simply **clipping**. The region against which an object is clipped is called a **clip window.**

Applications of clipping:

- Extracting part of a defined scene for viewing
- Identifying visible surfaces in three-dimensional views
- Antialiasing line segments or object boundaries
- Creating objects using solid-modeling procedures
- Displaying a multi-window environment
- Drawing and painting operations that allow parts of a picture to be selected for copying, moving, erasing, or duplicating.

Depending on the application, the clip window can be a general polygon or it can even have curved boundaries.

Point Clipping

Assuming that the clip window is a rectangle in standard position, we save a point **P = *(x,* y)** for display if the following inequalities are satisfied:

$$xw_{min} \leq x \leq xw_{max}$$
$$yw_{min} \leq y \leq yw_{max}$$

where the edges of the clip window **(xw_{min}, xw_{max}, yw_{min}, yw_{max})** can be either the world-coordinate window boundaries or viewport boundaries. If any one of these four inequalities is not satisfied, the **point is clipped** (not saved for display).

Although point clipping is applied less often than line or polygon clipping, it can be applied to scenes involving explosions or sea foam that are modeled with particles (points) distributed in some region of the scene.

Line Clipping

A line clipping procedure involves several parts:

- First, we can test a given line segment to determine whether it lies completely inside the clipping window.
- If it does not, we try to determine whether it lies completely outside the window.
- Finally, if we cannot identify a line as completely inside or completely outside, we must perform intersection calculations with one or more clipping boundaries. We process lines through the "inside-outside'' tests by checking the line endpoints.

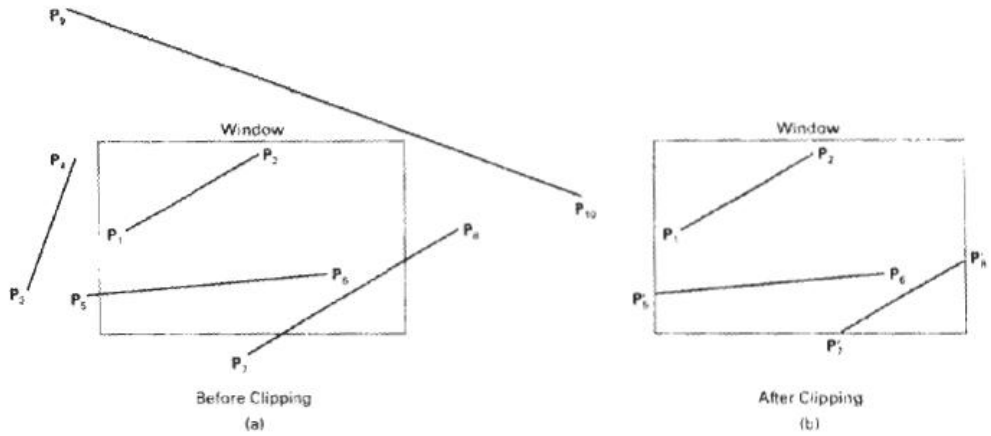


Fig: Line-clipping against a rectangular window

For a line segment with endpoints (x_1, y_1) and (x_2, y_2) and one or both endpoints outside the clipping rectangle, the parametric representation

$$x = x_1 + u(x_2 - x_1)$$
$$y = y_1 + u(y_2 - y_1), \quad 0 \leq u \leq 1$$

can be used to determine values of parameter **u** for intersections with the clipping boundary coordinates. If the value of **u** for an intersection with a rectangle boundary edge is outside the range **0** to **1,** the line does not enter the interior of the window at the boundary. If the value of ***u*** is within the range from **0** to **1**, the line segment does indeed cross into the clipping area. This method can be applied to each clipping boundary edge in turn to determine whether any part of the line segment is to be displayed. Line segments that are parallel to window edges can be handled as special cases.

Polygon Clipping

To clip polygons, we need to modify the line-clipping procedures discussed in the previous section. A polygon boundary processed with a line clipper may be displayed as a series of unconnected line segments depending on the orientation of the polygon to the clipping window.

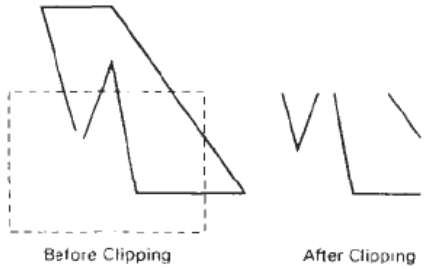


Fig: Display of a polygon processed by a line-dipping algorithm

What we really want to display is a bounded area after clipping, as in Fig. below.

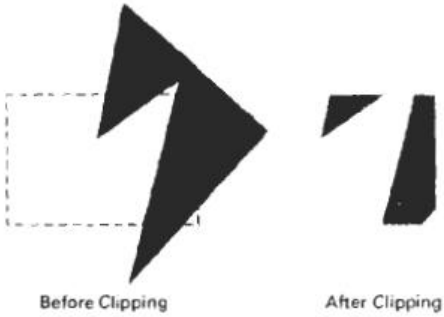


Fig: Display of a correctly clipped polygon

For polygon clipping, we require an algorithm that will generate one or more closed areas that are then scan-converted for the appropriate area fill. The output of a polygon clipper should be a sequence of vertices that defines the clipped polygon boundaries.

Curve Clipping

Curve-clipping procedures will involve nonlinear equations and this requires more processing than for objects with linear boundaries. The bounding rectangle for a circle or other curved object can be used first to test for overlap with a rectangular clip window.

- If the bounding rectangle for the object is completely inside the window, we save the object.
- If the rectangle is determined to be completely outside window, we discard the object.

In either case, there is no further computation necessary. But if the bounding rectangle test fails, we can look for other computation-saving approaches.

- For a circle, we can use the coordinate extents of individual quadrants and then octants for preliminary testing before calculating curve-window intersections.
- For an ellipse, we can test the coordinate extents of individual quadrants

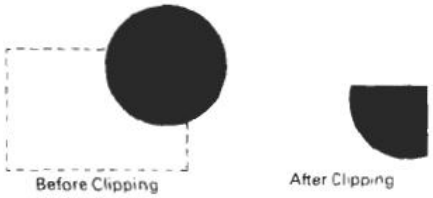


Fig: Clipping a filled Circle