

Translation of objects in computer graphics

In computer graphics, we have seen how to draw some basic figures like line and circles. In this post we will discuss on basics of an important operation in computer graphics as well as 2-D geometry, which is **transformation**.

In computer graphics, transformation of the coordinates consists of three major processes:

- Translation
- Rotation
- Scaling

In this post we will discuss about **translation** only.

What is translation?

A translation process moves every point a constant distance in a specified direction. It can be described as a rigid motion. A translation can also be interpreted as the addition of a constant vector to every point, or as shifting the origin of the coordinate system.

Suppose, If point (X, Y) is to be translated by amount D_x and D_y to a new location (X', Y') then new coordinates can be obtained by adding D_x to X and D_y to Y as:

$$X' = Dx + X$$

$$Y' = Dy + Y \text{ or}$$

$$P' = T + P \text{ where}$$

$$P' = (X', Y'),$$

$$T = (Dx, Dy),$$

$$P = (X, Y)$$

Here, $P(X, Y)$ is the original point. $T(Dx, Dy)$ is the **translation factor**, i.e. the amount by which the point will be translated. $P'(X', Y')$ is the coordinates of point P after translation.

Examples:

Input : $P[] = \{5, 6\}$, $T = \{1, 1\}$

Output : $P'[] = \{6, 7\}$

Input : $P[] = \{8, 6\}$, $T = \{-1, -1\}$

Output : $P'[] = \{7, 5\}$

Whenever we perform translation of any object we simply translate its each and every point. Some of basic objects along with their translation can be drawn as:

1.Point Translation $P(X, Y)$: Here we only translate the x and y coordinates of given point as per given translation factor dx and dy respectively.

Below is the C++ program to translate a point:

```
// C++ program for translation  
// of a single coordinate
```

```

#include<bits/stdc++.h>
#include<graphics.h>

using namespace std;

// function to translate point
void translatePoint ( int P[], int T[])
{
    /* init graph and putpixel are used for
    representing coordinates through graphical
    functions
    */

    int gd = DETECT, gm, errorcode;
    initgraph (&gd, &gm, "c:\\tc\\bgi");

    cout<<"Original Coordinates :"<<P[0]<<","<<P[1];
    putpixel (P[0], P[1], 1);

    // calculating translated coordinates

    P[0] = P[0] + T[0];
    P[1] = P[1] + T[1];

    cout<<"\nTranslated Coordinates :"<< P[0]<<","<<
P[1];

    // Draw new coordinatses

    putpixel (P[0], P[1], 3);
    closegraph();
}

// driver program
int main()
{
    int P[2] = {5, 8}; // coordinates of point
    int T[] = {2, 1}; // translation factor
    translatePoint(P, T);
    return 0;
}

```

Output:

Original Coordinates : 5, 8 Translated Coordinates : 7,
9

2.Line Translation: The idea to translate a line is to translate both of the end points of the line by the given translation factor(dx, dy) and then draw a new line with inbuilt graphics function. Below is the C++ implementation of above idea:

```
// cpp program for translation
// of a single line

#include<bits/stdc++.h>
#include<graphics.h>

using namespace std;

// function to translate line

void translateLine ( int P[][2], int T[])
{
    /* init graph and line() are used for
    representing line through graphical
    functions
    */

    int gd = DETECT, gm, errorcode;
    initgraph (&gd, &gm, "c:\\tc\\bgi");

    // drawing original line using graphics functions

    setcolor (2);
    line(P[0][0], P[0][1], P[1][0], P[1][1]);

    // calculating translated coordinates

    P[0][0] = P[0][0] + T[0];
```

```
P[0][1] = P[0][1] + T[1];  
P[1][0] = P[1][0] + T[0];  
P[1][1] = P[1][1] + T[1];
```

```
// drawing translated line using graphics functions
```

```
setcolor(3);
```

```
line(P[0][0], P[0][1], P[1][0], P[1][1]);
```

```
closegraph();
```

```
}
```

```
// driver program
```

```
int main()
```

```
{
```

```
    int P[2][2] = {5, 8, 12, 18}; // coordinates of point  
    int T[] = {2, 1}; // translation factor
```

```
    translateLine (P, T);
```

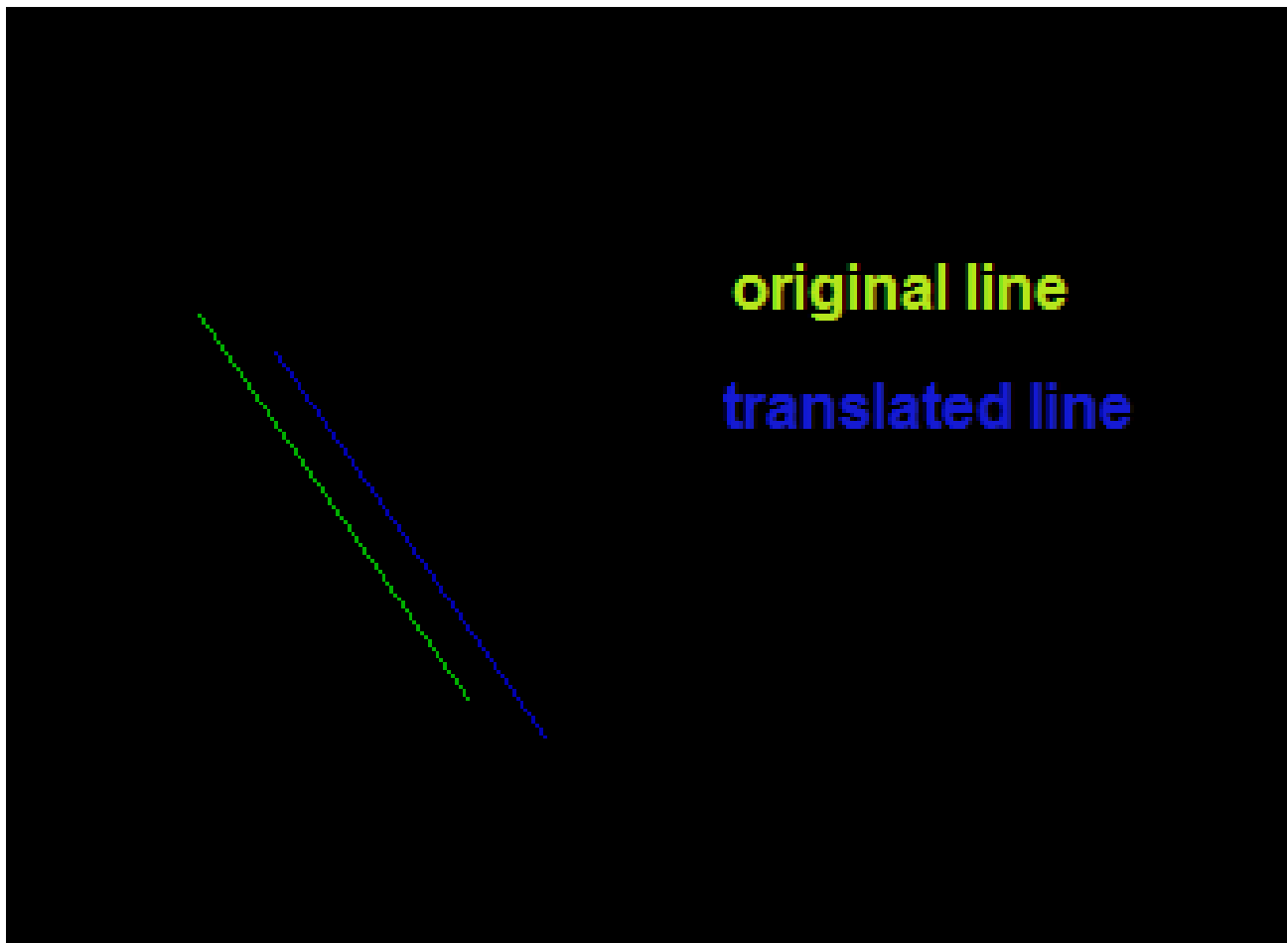
```
    return 0;
```

```
}
```

Output:



DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Pr



3.Rectangle Translation : Here we translate the x and y coordinates of both given points A(top left) and B(bottom right) as per given translation factor dx and dy respectively and then draw a rectangle with inbuilt graphics function

```

// C++ program for translation
// of a rectangle

#include<bits/stdc++.h>
#include<graphics.h>

using namespace std;

// function to translate rectangle
void translateRectangle ( int P[][2], int T[])
{
    /* init graph and rectangle() are used for
    representing rectangle through graphical functions */

    int gd = DETECT, gm, errorcode;
    initgraph (&gd, &gm, "c:\\tc\\bgi");
    setcolor (2);

    // rectangle (Xmin, Ymin, Xmax, Ymax)
    // original rectangle

    rectangle (P[0][0], P[0][1], P[1][0], P[1][1]);

    // calculating translated coordinates

    P[0][0] = P[0][0] + T[0];
    P[0][1] = P[0][1] + T[1];
    P[1][0] = P[1][0] + T[0];
    P[1][1] = P[1][1] + T[1];

    // translated rectangle (Xmin, Ymin, Xmax, Ymax)
    // setcolor(3);

    rectangle (P[0][0], P[0][1], P[1][0], P[1][1]);

    // closegraph();
}

// driver program

```

```

int main()
{
    // Xmin, Ymin, Xmax, Ymax as rectangle
    // coordinates of top left and bottom right points


    int P[2][2] = {5, 8, 12, 18};
    int T[] = {2, 1}; // translation factor

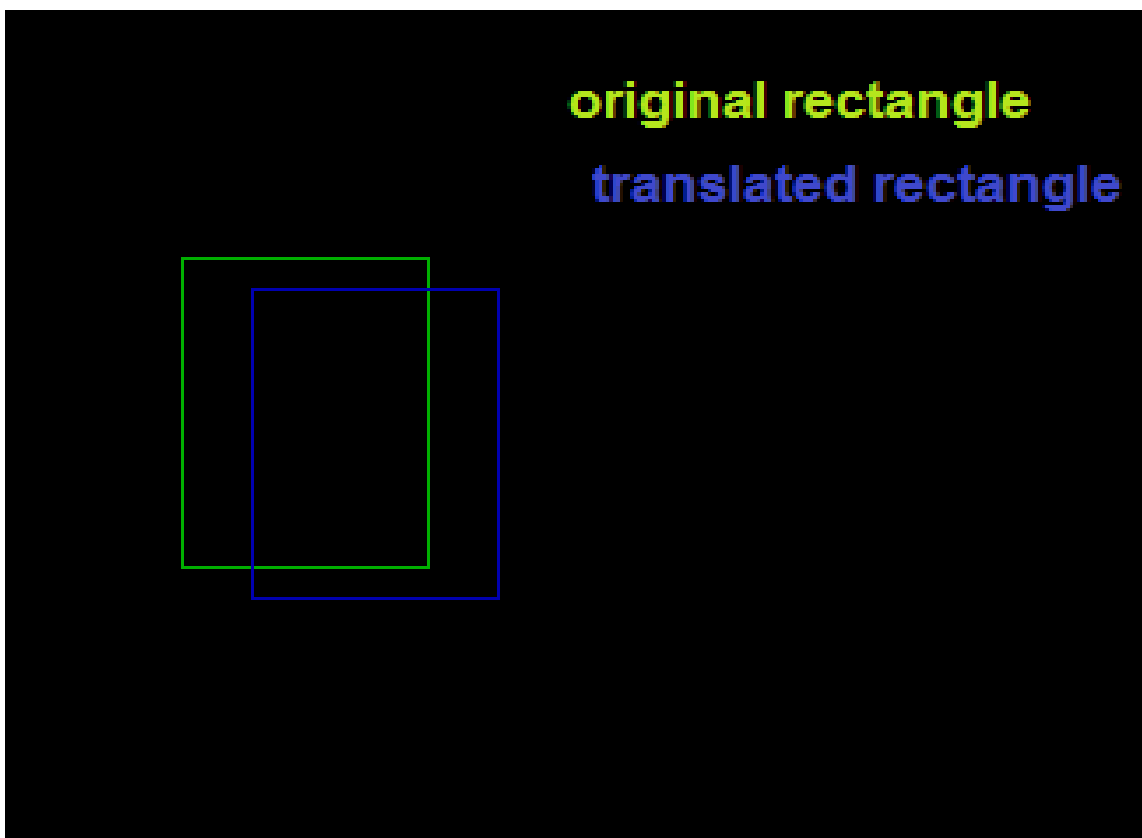
    translateRectangle (P, T);

    return 0;
}

```

Output:

 DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, F



Scaling

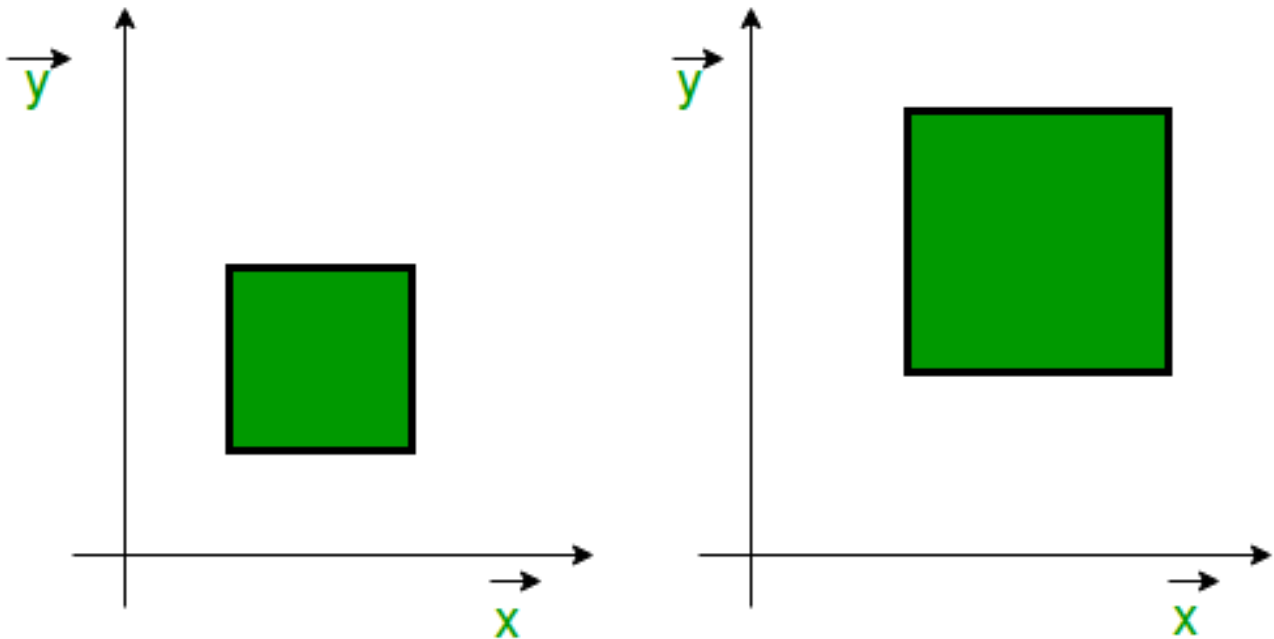
A scaling transformation alters size of an object. In the scaling process, we either compress or expand the dimension of the object. Scaling operation can be achieved by multiplying each vertex coordinate (x, y) of the polygon by scaling factor s_x and s_y to produce the transformed coordinates as (x', y').

So, $x' = x * s_x$ and $y' = y * s_y$. The scaling factor s_x, s_y scales the object in X and Y direction respectively. So, the above equation can be represented in matrix form:

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}$$

Or $P' = S . P$

Scaling Process:



Note: If the scaling factor S is less than 1, then we reduce the size of the object. If the scaling factor S is greater than 1, then we increase size of the object.

Algorithm:

1. Make a 2×2 scaling matrix S as: $\begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$. For each point of the polygon. (i) Make a 2×1 matrix P , where $P[0][0]$ equals to x coordinate of the point and $P[1][0]$ equals to y coordinate of the point. (ii) Multiply scaling matrix S with point matrix P to get

the new coordinate.3. Draw the polygon using new coordinates.

Below is C implementation:

```
// C program to demonstrate scaling of abjects
#include<stdio.h>
#include<graphics.h>

// Matrix Multiplication to find new Coordinates.
// s[][] is scaling matrix. p[][] is to store
// points that needs to be scaled.
// p[0][0] is x coordinate of point.
// p[1][0] is y coordinate of given point.

void findNewCoordinate(int s[][2], int p[][1])
{
    int temp[2][1] = { 0 };

    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 1; j++)
            for (int k = 0; k < 2; k++)
                temp[i][j] += (s[i][k] * p[k][j]);

    p[0][0] = temp[0][0];
    p[1][0] = temp[1][0];
}

// Scaling the Polygon

void scale(int x[], int y[], int sx, int sy)
{
    // Triangle before Scaling

    line(x[0], y[0], x[1], y[1]);
    line(x[1], y[1], x[2], y[2]);
    line(x[2], y[2], x[0], y[0]);

    // Initializing the Scaling Matrix.

    int s[2][2] = { sx, 0, 0, sy };
```

```

int p[2][1];
// Scaling the triangle
for (int i = 0; i < 3; i++)
{
    p[0][0] = x[i];
    p[1][0] = y[i];
    findNewCoordinate(s, p);
    x[i] = p[0][0];
    y[i] = p[1][0];
}
// Triangle after Scaling
line(x[0], y[0], x[1], y[1]);
line(x[1], y[1], x[2], y[2]);
line(x[2], y[2], x[0], y[0]);
}

```

// Driven Program

```

int main()
{
    int x[] = { 100, 200, 300 };
    int y[] = { 200, 100, 200 };
    int sx = 2, sy = 2;

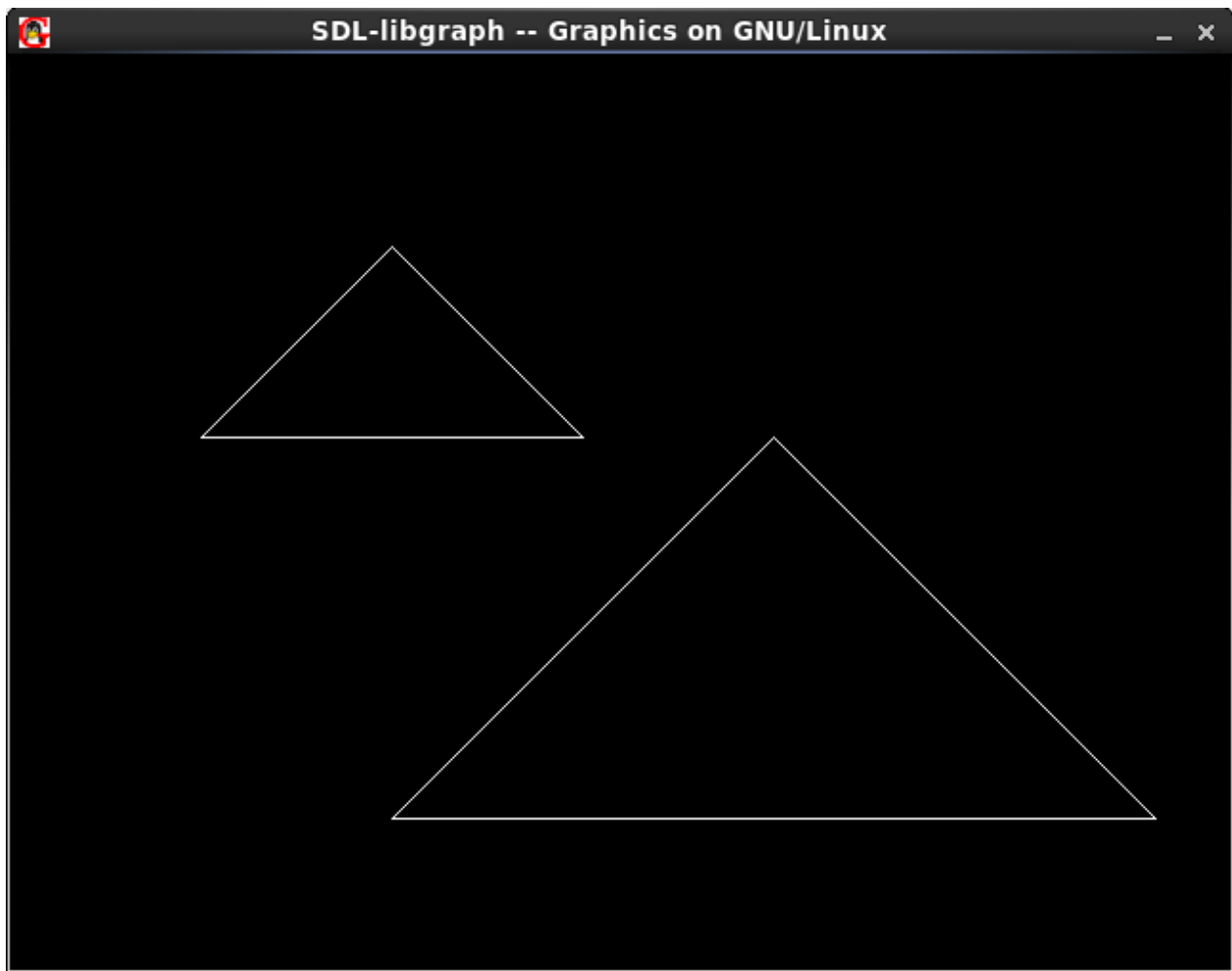
    int gd, gm;
    detectgraph(&gd, &gm);
    initgraph(&gd, &gm, " ");

    scale(x, y, sx, sy);
    getch();

    return 0;
}

```

Output:



Rotation

We have to rotate an object by a given angle about a given pivot point and print the new co-ordinates.

Examples:

Input : $\{(100, 100), (150, 200), (200, 200), (200, 150)\}$
is to be rotated about

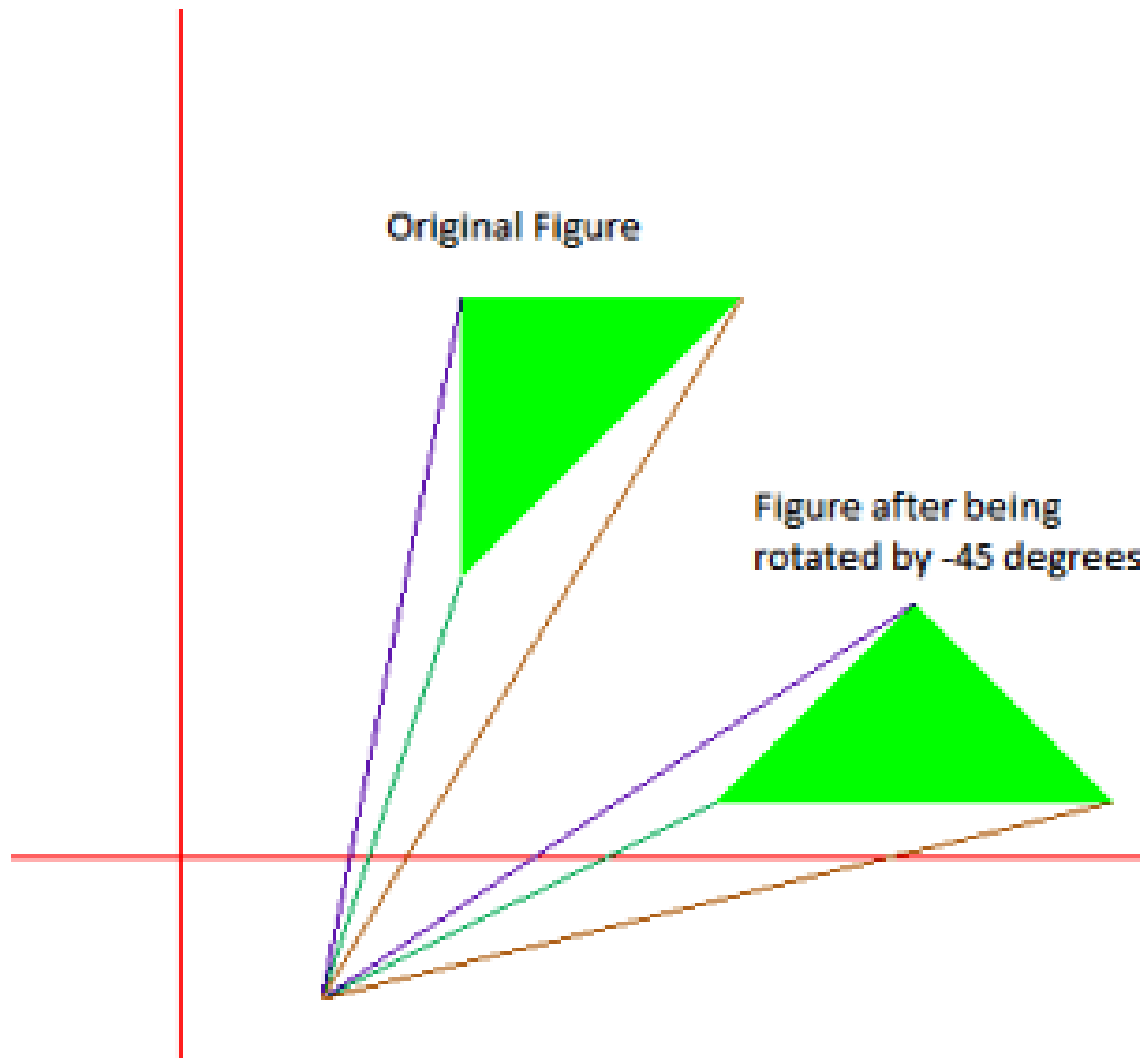
(0, 0) by 90 degrees

Output :
 $(-100, 100), (-200, 150), (-200, 200), (-150, 200)$

Input : $\{(100, 100), (100, 200), (200, 200)\}$
is to be rotated about $(50, -50)$ by -45 degrees

Output :

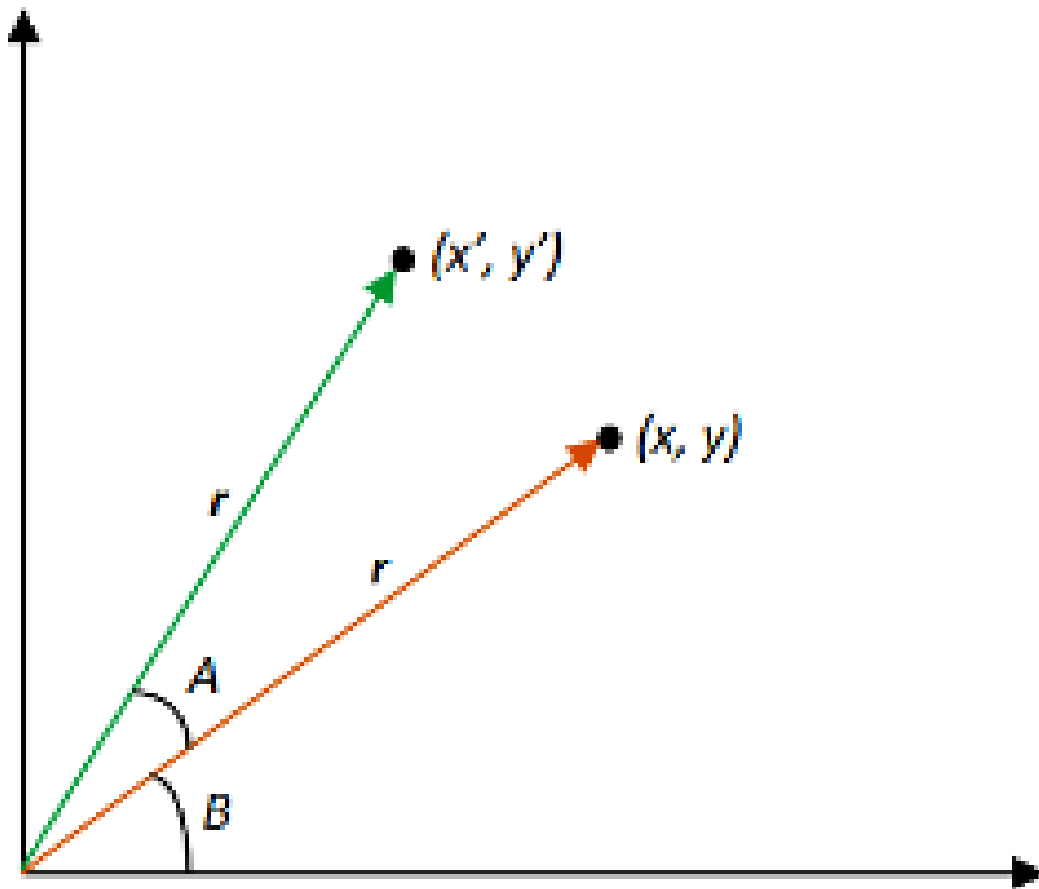
$(191.421, 20.7107), (262.132, 91.4214), (332.843, 20.7107)$



Recommended: Please try your approach on **{IDE}** first, before moving on to the solution.

In order to rotate an object we need to rotate each vertex of the figure individually.

On rotating a point $P(x, y)$ by an angle A about the origin we get a point $P'(x', y')$. The values of x' and y' can be calculated as follows:-



We know that,

$$x = r\cos B, y = r\sin B$$

$$x' = r\cos(A+B) = r(\cos A\cos B - \sin A\sin B)$$

$$= \underline{r\cos B}\cos A - \underline{r\sin B}\sin A$$

$$= \mathbf{x\cos A - y\sin A}$$

$$\mathbf{y' = r\sin(A+B) = r(\sin A\cos B + \cos A\sin B)}$$

$$= \underline{r\cos B}\sin A + \underline{r\sin B}\cos A = \mathbf{x\sin A + y\cos A}$$

Rotational Matrix Equation:-

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} \begin{bmatrix} \cos A & -\sin A \\ \sin A & \cos A \end{bmatrix}$$

```
// C program to rotate an object by
// a given angle about a given point

#include<stdio.h>
#include<math.h>

// Using macros to convert degree to radian
// and call sin() and cos() as these functions
// take input in radians

#define SIN(x) sin(x * 3.141592653589/180)
#define COS(x) cos(x * 3.141592653589/180)

// To rotate an object

void rotate(float a[][2], int n, int x_pivot,
            int y_pivot, int angle)
{
    int i = 0;
    while (i < n)
    {
        // Shifting the pivot point to the origin
```



```

    // and the given points accordingly

    int x_shifted = a[i][0] - x_pivot;
    int y_shifted = a[i][1] - y_pivot;

    // Calculating the rotated point co-ordinates
    // and shifting it back

    a[i][0] = x_pivot + (x_shifted * COS(angle)
                        - y_shifted * SIN(angle));
    a[i][1] = y_pivot + (x_shifted * SIN(angle)
                        + y_shifted * COS(angle));

    printf("(%f, %f) ", a[i][0], a[i][1]);

    i++;
}
}

// Driver Code

int main()
{
    // 1st Example
    // The following figure is to be
    // rotated about (0, 0) by 90 degrees

    int size1 = 4; // No. of vertices

    // Vertex co-ordinates must be in order

    float points_list1[][2] = {{100, 100}, {150, 200},
                                {200, 200}, {200, 150}};

    rotate(points_list1, size1, 0, 0, 90);

    // 2nd Example
    // The following figure is to be
    // rotated about (50, -50) by -45 degrees
    /*int size2 = 3; // No. of vertices
    float points_list2[][2] = {{100, 100}, {100, 200},
                                {200, 200}};
*/
}

```

```
rotate(points_list2, size2, 50, -50, -45);*/  
return 0;  
}
```

Output:(-100, 100), (-200, 150), (-200, 200), (-150, 200)

Homogeneous Coordinates

The rotation of a point, straight line or an entire image on the screen, about a point other than origin, is achieved by first moving the image until the point of rotation occupies the origin, then performing rotation, then finally moving the image to its original position.

The moving of an image from one place to another in a straight line is called a translation. A translation may be done by adding or subtracting to each point, the amount, by which picture is required to be shifted.

Translation of point by the change of coordinate cannot be combined with other transformation by using simple matrix application. Such a combination is essential if we wish to rotate an image about a point other than origin by translation, rotation again translation.

To combine these three transformations into a single transformation, homogeneous coordinates are used. In homogeneous coordinate system, two-dimensional coordinate positions (x, y) are represented by triple-coordinates.

Homogeneous coordinates are generally used in design and construction applications. Here we perform

translations, rotations, scaling to fit the picture into proper position.

Example of representing coordinates into a homogeneous coordinate system: For two-dimensional geometric transformation, we can choose homogeneous parameter h to any non-zero value. For our convenience take it as one. Each two-dimensional position is then represented with homogeneous coordinates $(x, y, 1)$.

Following are matrix for two-dimensional transformation in homogeneous coordinate:

1. Translation

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix} \text{ or } \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

2. Scaling

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3. Rotation (clockwise)

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

4. Rotation (anti-clock)

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

5. Reflection against X axis

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

6. Reflection against Y axis

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

7. Reflection against origin

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

8. Reflection against line Y=X

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

9. Reflection against Y= -X

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

10. Shearing in X direction

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

11. Shearing in Y direction

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

12. Shearing in both x and y direction

$$\begin{bmatrix} 1 & Sh_y & 0 \\ Sh_x & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Reflection:

It is a transformation which produces a mirror image of an object. The mirror image can be either about x-axis or y-axis. The object is rotated by 180° .

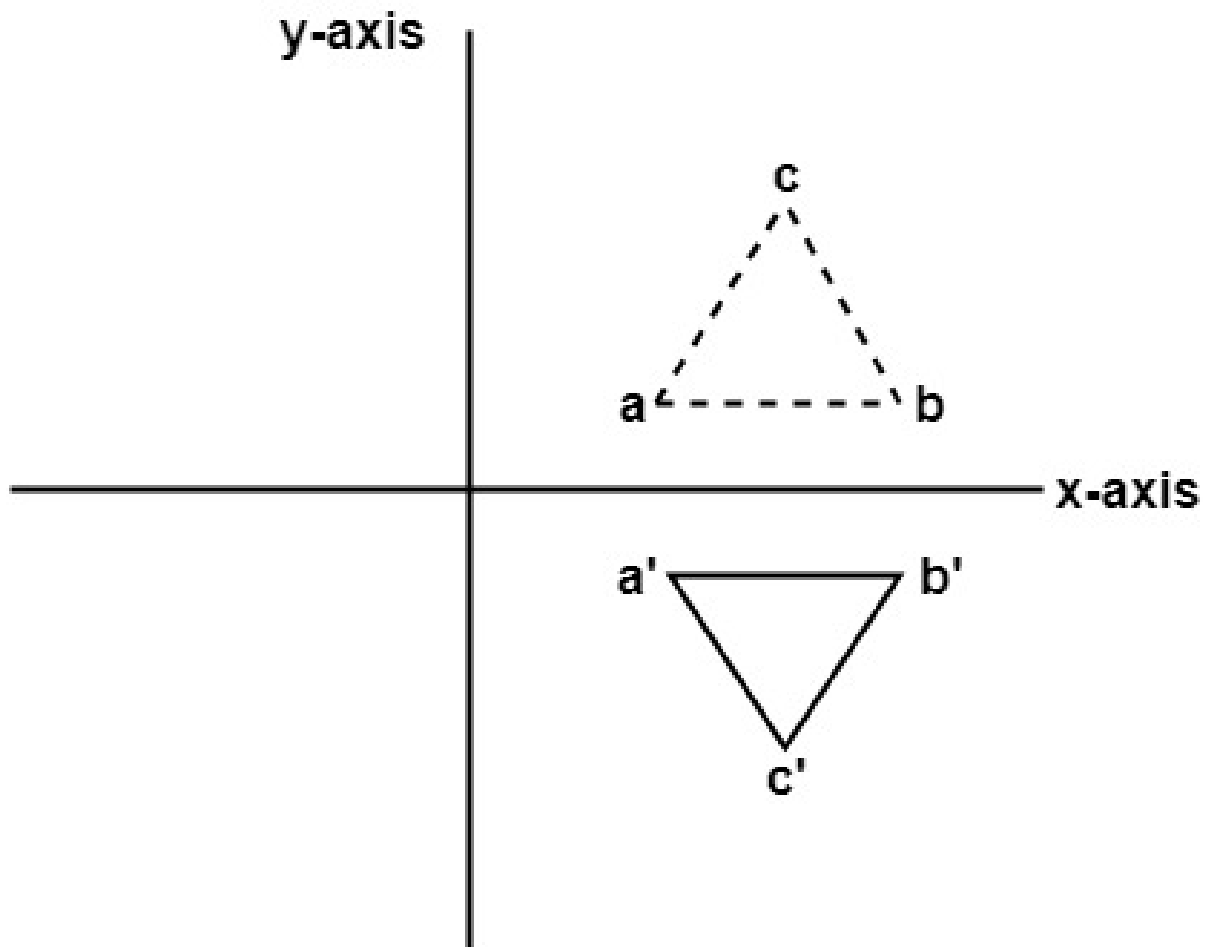
Types of Reflection:

4. Reflection about the x-axis
5. Reflection about the y-axis
6. Reflection about an axis perpendicular to xy plane and passing through the origin
7. Reflection about line $y=x$

1. Reflection about x-axis: The object can be reflected about x-axis with the help of the following matrix

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

In this transformation value of x will remain same whereas the value of y will become negative. Following figures shows the reflection of the object axis. The object will lie another side of the x-axis.

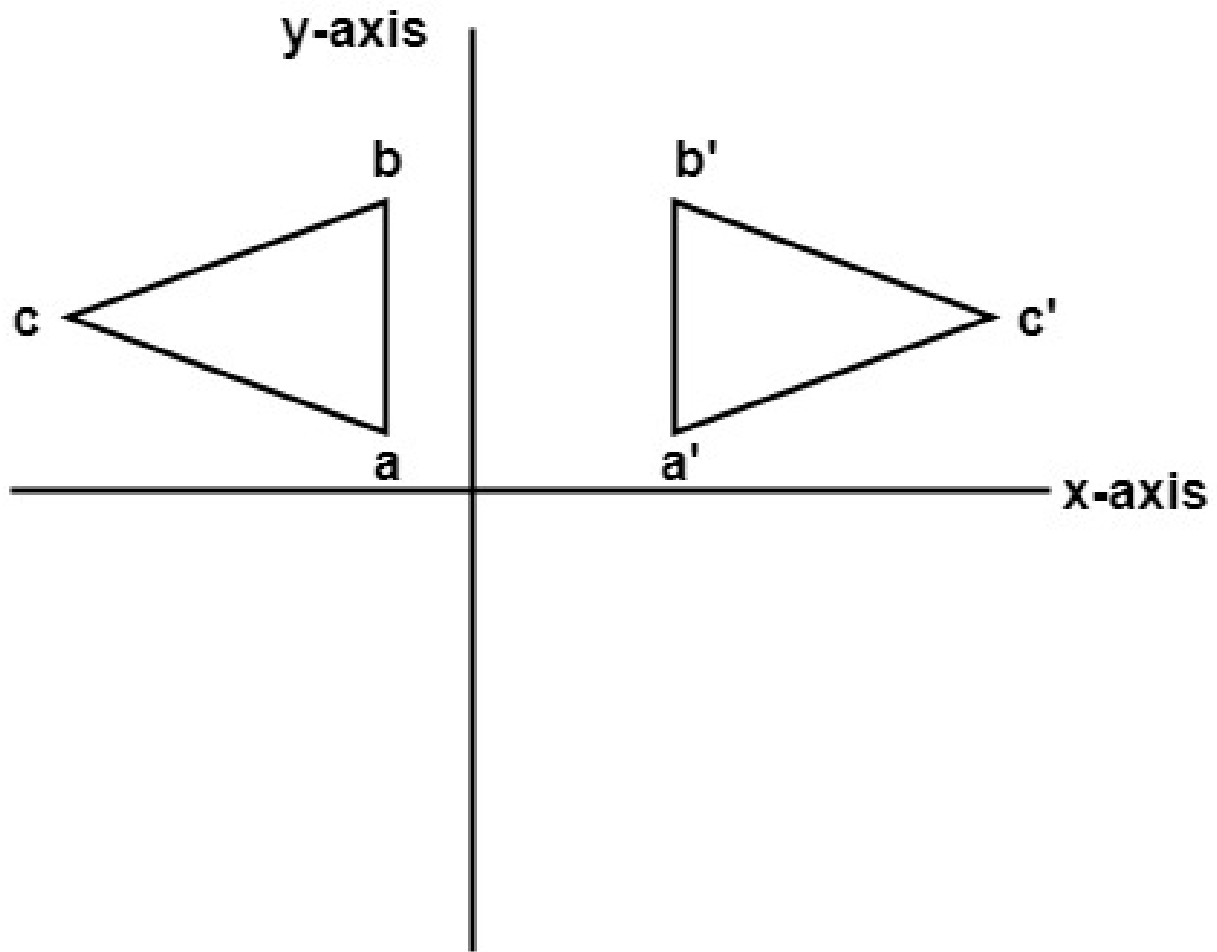


2. Reflection about y-axis: The object can be reflected about y-axis with the help of following transformation matrix

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

Here the values of x will be reversed, whereas the value of y will remain the same. The object will lie another side of the y-axis.

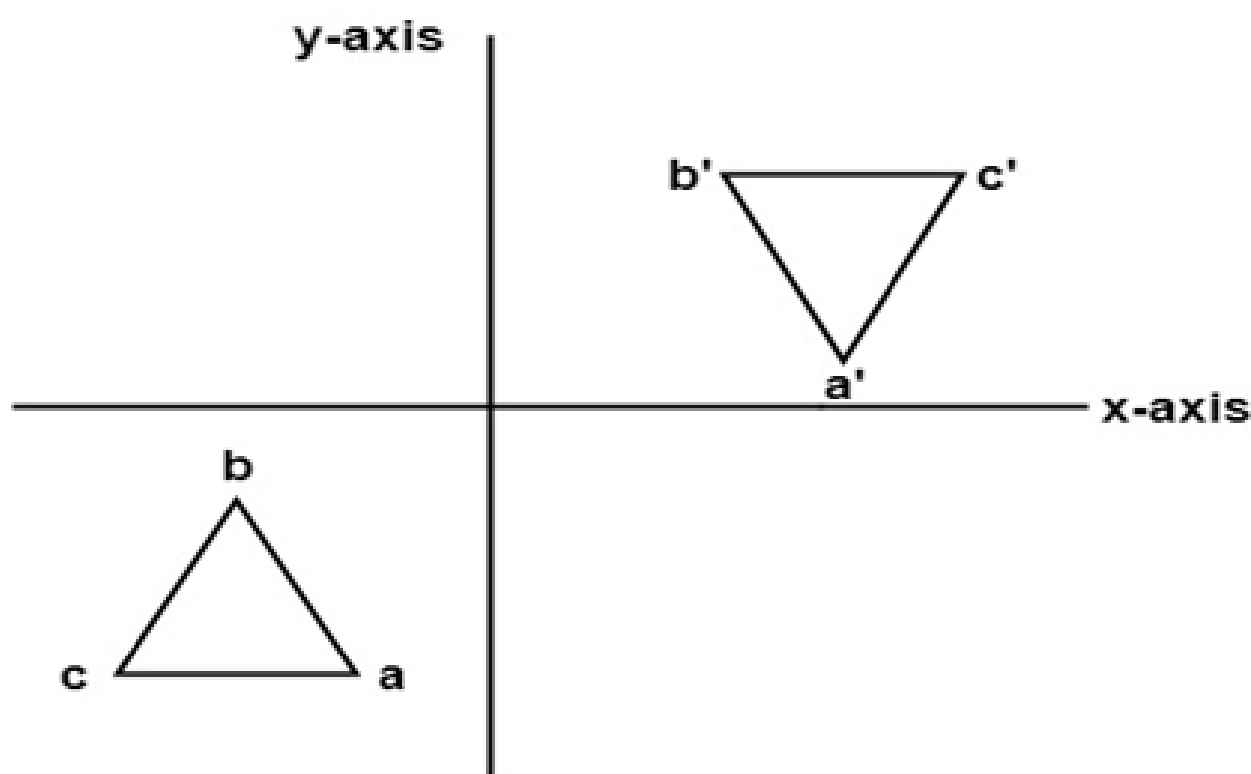
The following figure shows the reflection about the y-axis



**3. Reflection about an axis perpendicular to xy plane and passing through origin:
In the matrix of this transformation is given below**

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

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

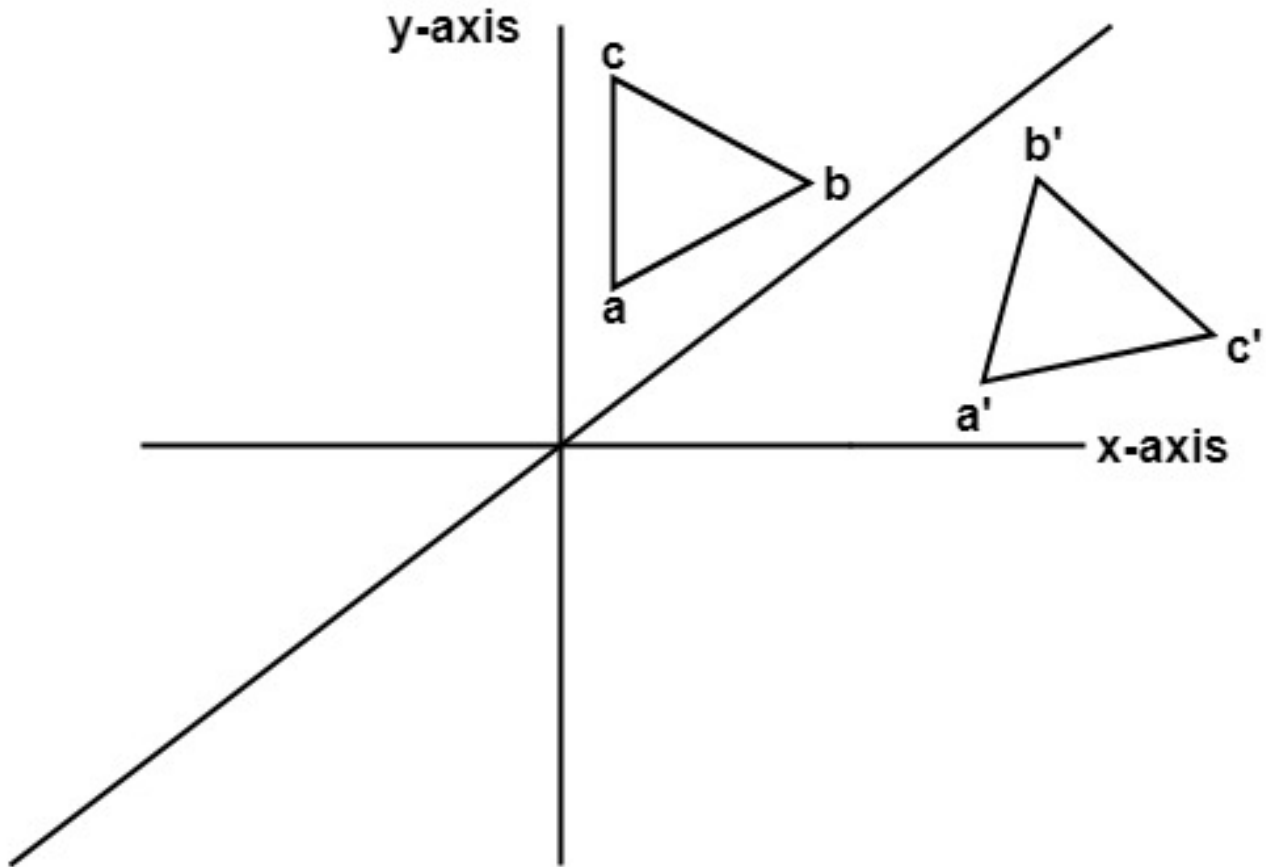


In this value of x and y both will be reversed. This is also called as half revolution about the origin.

4. Reflection about line $y=x$: The object may be reflected about line $y = x$ with the help of following transformation matrix

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

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



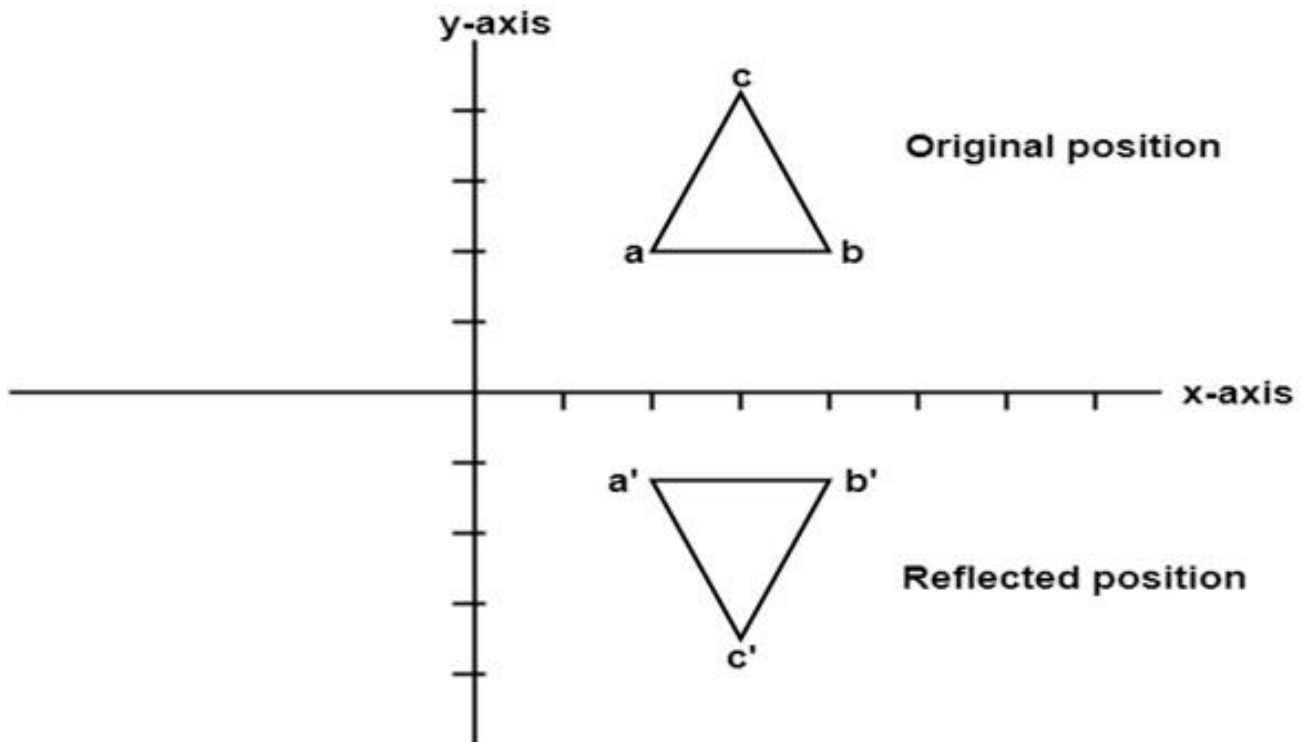
First of all, the object is rotated at 45° . The direction of rotation is clockwise. After it reflection is done concerning x-axis. The last step is the rotation of $y=x$ back to its original position that is counterclockwise at 45° .

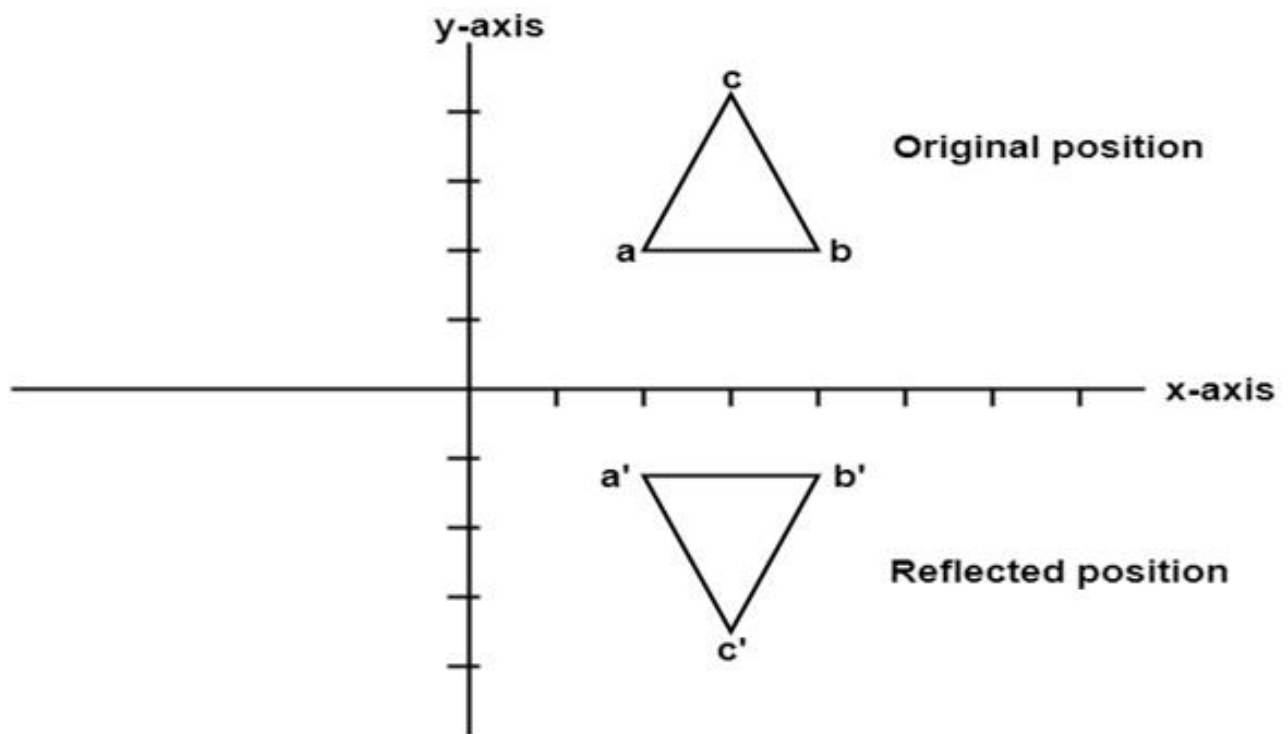
Example: A triangle ABC is given. The coordinates of A, B, C are given as

A	(3	4)
B	(6	4)
C	(4	8)

Find reflected position of triangle i.e., to the x-axis.

Solution:





The matrix for reflection about x axis $\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

The a point coordinates after reflection

$$(x, y) = (3, 4) \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$(x, y) = [3, -4]$$

The b point coordinates after reflection

$(x, y) = (6, 4) \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ The coordinate of point c after reflection
 $(x, y) = [6, -4]$

$(x, y) = (4, 8) \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ a (3, 4) becomes a^1 (3, -4)
 $(x, y) = [4, -8]$ b (6, 4) becomes b^1 (6, -4)
c (4, 8) becomes c^1 (4, -8)

Program to perform Mirror Reflection about a line:

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
#include <graphics.h>
```

```
#include <math.h>
```

```
#include <stdlib.h>
```

```
#define pi 3.14
```

```
class arc
```

```
{ float x[10],y[10],theta,ref[10][10],ang;
```

```
float p[10][10],p1[10][10],x1[10],y1[10],xm,ym;
```

```
int i,k,j,n;
```

```
public:
```

```
void get();
```

```
void cal ();
```

```
void map ();
```

```
void graph ();
```

```
void plot ();
```

```
void plot1();
```

```
};
```

```
void arc::get ()
```

```
{  
    cout<<"\n ENTER ANGLE OF LINE INCLINATION AND  
Y INTERCEPT";
```

```
    cin>> ang >> b;  
    cout <<"\n ENTER NO OF VERTICES";  
    cin >> n;  
    cout <<"\n ENTER";  
    for (i=0; i<n; i++)  
    {  
        cout<<"\n x["<<i<<"] and y["<<i<<"]";  
    }  
    theta =(ang * pi)/ 180;  
    ref [0] [0] = cos (2 * theta);  
    ref [0] [1] = sin (2 * theta);  
    ref [0] [2] = -b *sin (2 * theta);  
    ref [1] [0] = sin (2 * theta);  
    ref [1] [1] = -cos (2 * theta);  
    ref [1] [2] = b * (cos (2 * theta)+1);  
    ref [2] [0]=0;  
    ref [2] [1]=0;  
    ref [2] [2] = 1;
```

```
}
```

```
void arc :: cal ()
```

```
{  
    for (i=0; i < n; i++)
```

```

{
    p[0] [i] = x [i];
    p [1] [i] = y [i];
    p [2] [i] = 1;
}
for (i=0; i<3;i++)
{
    for (j=0; j<n; j++)
    {
        p1 [i] [j]=0;
        for (k=0;k<3; k++)
        }
        p1 [i] [j] + = ref [i] [k] * p [k] [j];
    }
for (i=0; i<n; i++)
{
    x1 [i]=p1[0] [i];
    y1 [i] = p1 [1] [i];
}
}
void arc :: map ()
{
    int gd = DETECT,gm;
    initgraph (&gd, &gm, " ");
    int errorcode = graphresult ();
    /* an error occurred */
    if (errorcode != grOK)

```



```

    {
        printf ("Graphics error: %s \n", grapherrormsg
(errorcode));
        printf ("Press any key to halt:");
        getch ();
        exit (1); /* terminate with an error code */
    }
}
void arc :: graph ()
{
    xm=getmaxx ()/2;
    ym=getmaxy ()/2;
    line (xm, 0, xmm 2*ym);
}
void arc :: plot 1 ()
{
    for (i=0; i <n-1; i++)
    {
        circle (x1[i]+xm, (-y1[i]+ym), 2);
        line (x1[i]+xm, (-y1[i]+ym), x1[i+1]+xm, (-
y1[i+1]+ym));
    }
    line (x1[n-1]+xm, (-y1[n-1]+ym), x1[0]+xm, (-
y1[0]+ym));
    getch();
}
void arc :: plot ()
{

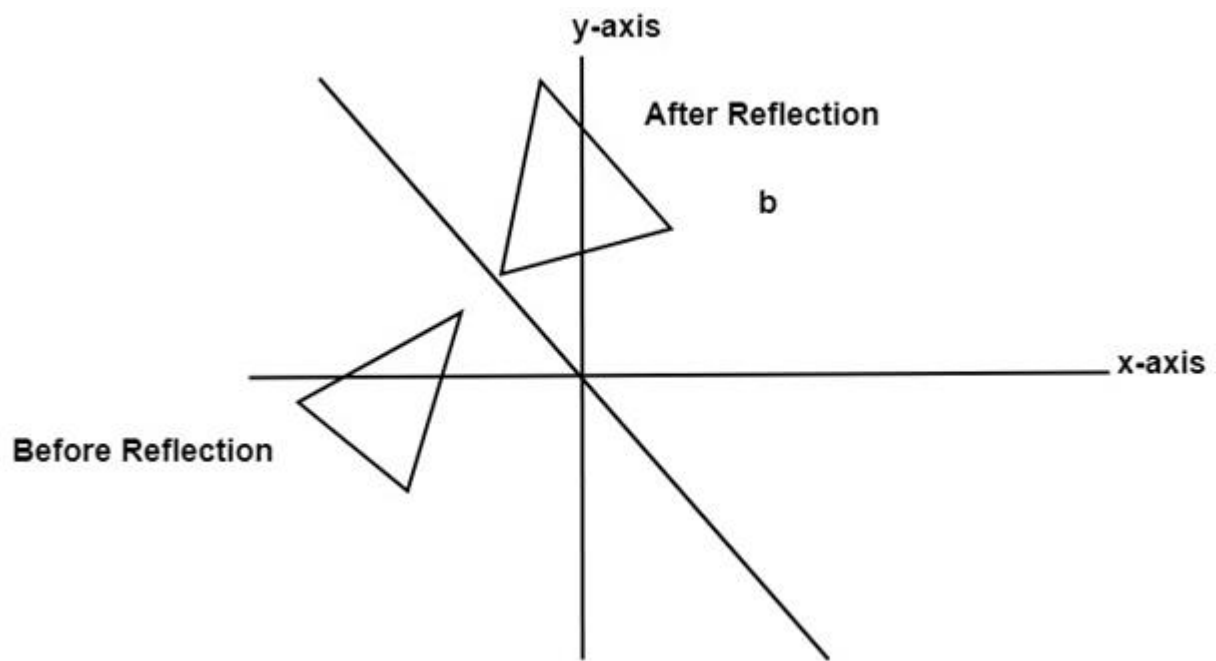
```

```

    for (i=0; i <n-1; i++)
    {
        circle (x1[i]+xm, (-y1[i]+ym, 2);
        line (x1[i]+xm, (-y1[i]+ym), x[i+1]+xm, (-
y1[i+1]+ym));
    }
    line (x[n-1]+xm, (-y1[n-1]+ym), x[0]+xm, (-
y[0]+ym));
    getch();
}
void main ()
{
    class arc a;
    clrscr();
    a.map();
    a.graph();
    a.get();
    a.cal();
    a.plot();
    a.plot1();
    getch();
}

```

Output:

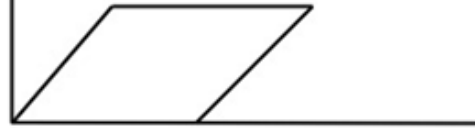


Shearing

$$\begin{bmatrix} 1 & 0 & 0 \\ Sh_x & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ Sh_x & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{matrix} \text{ject.} \\ \text{be in} \\ \text{w:} \end{matrix}$$



Original Object



Shear in X direction



Shear in Y direction



Shear in both directions

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

one by
slided
in
of the
th

Inverse Transformations

These are also called as opposite transformations. If T is a translation matrix than inverse translation is representing using T^{-1} . The inverse matrix is achieved using the opposite sign.

Example1: Translation and its inverse matrix

Translation matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{pmatrix}$$

Inverse translation matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -T_x & -T_y & -T_z & 1 \end{pmatrix}$$

Example2: Rotation and its inverse matrix

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Inverse Rotation Matrix

$$\begin{pmatrix} -\cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & -\cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$