MMTE-004
COMPUTER GRAPHICS

Indira Gandhi Ntional Open University
School of Sciences

Block

2

2D TRANSFORMATIONS AND CLIPPING

UNIT 4

2D Transformations **87**

UNIT 5

Clipping Algorithms **109**

Curriculum Design Committee

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

Prof. Adimurthi
School of Mathematics
TIFR, Bangalore

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

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

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

Prof R. Bhatia
Indian Statistical Institute, New Delhi

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

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

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

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

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

Prof. Veni Madhavan
Scientific Analysis Group
DRDO, Delhi

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

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

Prof. Sankar Pal
ISI, Kolkata

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

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

Course Design Committee

Prof. Sujatha Varma
Director (SOS), IGNOU

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

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

Faculty Members
School of Sciences, IGNOU

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

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

Block Preparation Team

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

Dr. Pawan Kumar
School of Sciences
IGNOU

Course Coordinator: Dr. Pawan Kumar

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

……., 2023

ISBN-8]-

BLOCK INTRODUCTION

This is the second block of the course Computer Graphics, which is divided into 2 units we discuss the two-dimensional transformations and some clipping methods.

In Unit 4 we introduce the primitive transformations for two-dimensional objects. These include translation, rotation and scaling, which can also be composed to obtain more complex transformations such as reflections and shear. We shall also see how to represent geometric transformations through homogeneous coordinates. Finally we talk about the transformations involved in viewing two-dimensional objects.

Unit 5 discusses the algorithms involving in clipping two-dimensional objects. We have described how to clip points, lines or curved regions. Two specific algorithms we have discussed in detail are Cohen-Sutherland Algorithm and Liang-Barsky Algorithms. The unit also includes a topic on Bezier and B-spline curves.

NOTATIONS AND SYMBOLS

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

UNIT 4

2D TRANSFORMATIONS

Structure **Page No.**

4.1	Introduction	87
	Objectives	
4.2	Two-dimensional Geometric Transformations	88
4.3	Homogeneous Coordinate Systems	91
	Transformations in Homogeneous Coordinates	
	Composite Transformations	
4.4	Reflection and Shear Transformations	96
4.5	Two-dimensional Viewing	100
4.6	Summary	105
4.7	Solutions/Answers	105

4.1 INTRODUCTION

In the last block you learned methods for drawing and filling geometric shapes of different kinds. Complex scenes often have multiple copies of a shape in different sizes and orientations. In this unit, our aim is to acquaint you with the basic concepts involved in transforming and viewing geometric objects.

Section 4.2 introduces you the concepts of two-dimensional transformations. The basic transformations you will study here are translation, rotation and scaling. Section 4.3 tells you how to represent the coordinates of a point in homogeneous coordinates. This is essential for applying a sequence of transformations on an object. Next, in Section 4.4 we talk about the reflection and shear transformations. Finally, in Section 4.5, we discuss the transformations involved in two-dimensional viewing.

Objectives

After reading this unit you should be able to:

- describe how basic 2D geometric transformations such as translation, rotation, scaling, reflection, etc. are applied on geometric objects;

- combine basic 2D transformations to obtain more useful composite transformations for applications;
- obtain window-to-viewport normalisation transformation matrix;
- implement these transformations using a C code with OpenGL.

4.2 TWO-DIMENSIONAL GEOMETRIC TRANSFORMATIONS

When a real-life object is modelled using shape primitives, there are several possible applications. You may be required to do further processing with the objects. For example, you may like to view it from different angles, or you may like to create another model with a slight variation in its shape or size. Similarly, you may want to show an object moving along a path, or rotate it about a given pivot point. All this can be achieved by using simple mathematical transformations called affine transformations. Since these transformations help change the geometry of the object in terms of shape, size or position, we call them geometric transformations. Present section deals with two-dimensional (2D) geometric transformations, which can be broadly classified as – (i) Rigid body transformations (ii) Non-rigid body transformations. **Rigid body transformations** do not change the object dimensions, while **non-rigid body transformations** do. For example, when you resize a rectangle, the transformation is non-rigid body, but when you rotate an object its shape or size does not change, hence it is rigid body transformation. We begin with the rigid body transformations.

Translation

Translation is a rigid-body transformation that moves an object without deformation. That is, each point of the object is displaced by the same amount in a direction. Displacement can take place in both the directions, say by a units along the x-axis and by b units along the y-axis. Then a point $P(x, y)$ moves to a new position $P'(x', y')$, where $x' = x + a,\ \ y' = y + b$

The vector $T = (a, b)$ is called the **translation** or **shift** vector. If we write $P = (x, y)$ and $P' = (x', y')$, then the matrix form of the translation becomes $P' = P + T$. Fig. 1 illustrates how an object translates from one position to another.

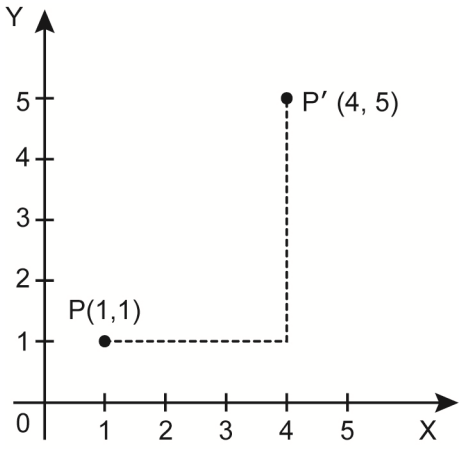


Fig. 1: A translation with $T = (3,4)$.

Rotation

Rotation is another rigid-body transformation. Under rotation an object rotates about a point called the **pivot point** with a specified angle, called the **rotation angle**. Consider a point $P(x, y)$ in the plane as shown below.

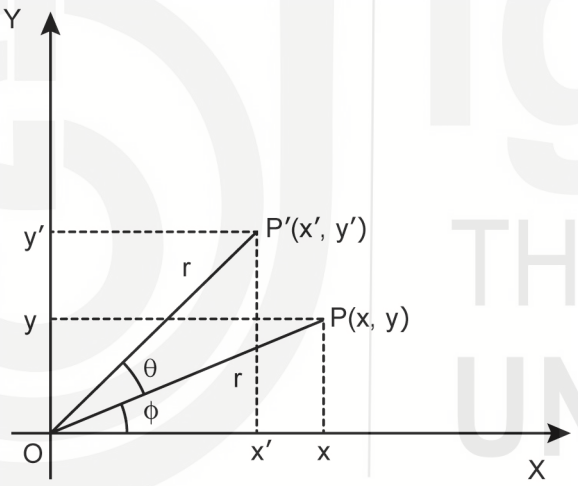


Fig. 2: A rotation of $P(x, y)$ by θ.

If r is the distance of P from the origin, and ϕ is the angle the line OP makes with the x-axis, then we can write $x = r\cos\phi$ and $y = r\sin\phi$. Now rotating P about the origin (as the pivot point) with an angle θ, we obtain a point $P'(x', y')$, where $x' = r\cos(\phi + \theta)$ and $y' = r\sin(\phi + \theta)$.

Simplifying these equations, we get

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta.$$

In the matrix form these equations can be written as $P' = RP$,

where $R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$ is the **rotation matrix**.

For example, if we apply the rotation by an angle of $\pi/4$ on the point $P(3, 5)$, then the point transforms to $P'(x', y')$ where

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\frac{\pi}{4} & -\sin\frac{\pi}{4} \\ \sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{pmatrix} \begin{pmatrix} 3 \\ 5 \end{pmatrix} = \begin{pmatrix} -\sqrt{2} \\ 4\sqrt{2} \end{pmatrix}.$$

Scaling

Scaling an object, intuitively, means stretching it or contracting it in a direction. Just like translation, scaling can take place in both directions, say by a factor α along the x-axis and by a factor β along the y-axis. Scaling a point $P(x, y)$ to a new point $P'(x', y')$ by the **scaling factors** α and β is done as follows:

$$x' = \alpha . x,\ y' = \beta . y$$

Note that α and β are positive real numbers. When $\alpha = \beta = 1,$ no change takes place in the position of an object. When $\alpha = 2$ and $\beta = 1,$ the scaling doubles the size of the object along the x-axis without any change along the y-axis. The above transformations can be represented in the matrix form as follows:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \alpha & 0 \\ 0 & \beta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

or $P' = S.P,$ where $S = \begin{pmatrix} \alpha & 0 \\ 0 & \beta \end{pmatrix}$ is called the **scaling matrix**. Fig. 3 shows how an object looks like after scaling.

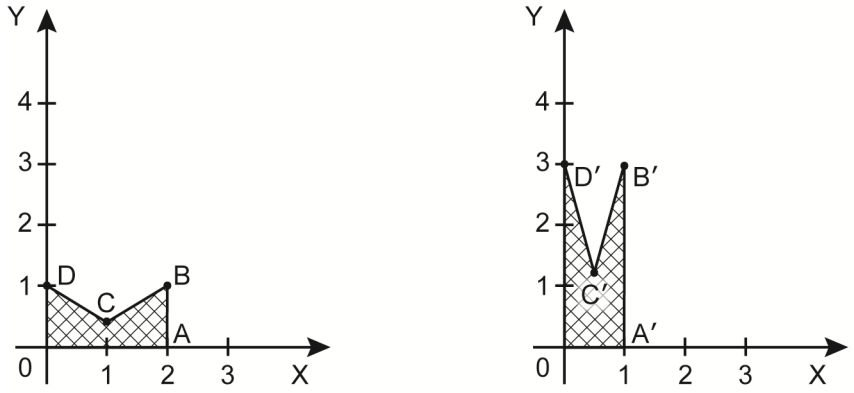


Fig. 3: Effect of scaling

When $\alpha = \beta,$ the scaling is called **uniform**, and **differential** otherwise. For instance, when an object is doubled or reduced to one-third of its size, we

mean uniform scaling. But for more realistic pictures differential scaling is applied.

Now, try the following exercise.

E1) Apply the following transformations on the rectangle with coordinates of its vertices as $(0, 2), (4, 2), (4, -1)$ and $(0, -1)$.

a) translation by $(1, 1)$

b) rotation by 60°

c) scaling by $\alpha = \frac{3}{4}, \beta = \frac{3}{2}$.

In the next section we shall revisit all these transformations, *albeit in a new form*!

4.3 HOMOGENEOUS COORDINATE SYSTEMS

In real life applications one often need to apply a sequence of transformations on an object. For instance, one may like to rotate an object after translation. To carry out such a sequence of transformations easily, we shall represent the transformations through 3×3 matrices, and for this we need the concept of homogenous coordinates.

Transformations in Homogenous Coordinates

Given a two-dimensional coordinate position (x, y), its **homogenous** counterpart is the triple (xh, yh, h), where h is a nonzero real number. For convenience, we take $h = 1$. Then the homogenous coordinates corresponding to (x, y) are $(x, y, 1)$. Using this notation, the translation $T(a, b)$, rotation $R(\theta)$ and the scaling $S(\alpha, \beta)$ matrices are represented as follows:

$$T(a,b) = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix}, R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}, S(\alpha, \beta) = \begin{pmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

So, let $P(x, y, 1)$ and $P'(x', y', 1)$ be the original and the transformed points in homogeneous coordinates, respectively. Then the translation, rotation and scaling transformations are given by $P' = T(a, b)P$, $P' = R(\theta)P$ and $P' = S(\alpha, \beta)P$, respectively. For instance, the point $P(2, 3, 1)$ in homogeneous coordinates, when rotated by 45°, becomes

$$P' = \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} -\frac{1}{\sqrt{2}} \\ \frac{5}{\sqrt{2}} \\ 1 \end{pmatrix}.$$

Composite Transformations

A **composite transformation** is just a product of two or more transformations. In fact, as we shall see, every composite transformation can be obtained through a sequence of elementary transformations, i.e., translation, rotation and scaling. With the help of the homogenous coordinates, the multiplication of any two transformation matrices gives us the desired result.

For instance, two successive translations by (a, b) and (a', b') on a point P are normally obtained by first computing $P' = T(a, b)P$ and then $P'' = T(a', b')P'$. Or equivalently, $P'' = T(a', b')T(a, b)P$. Now observe that

$$T(a', b') \cdot T(a, b) = \begin{pmatrix} 1 & 0 & a' \\ 0 & 1 & b' \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a+a' \\ 0 & 1 & b+b' \\ 0 & 0 & 1 \end{pmatrix} = T(a+a', b+b')$$

The matrix $T(a+a', b+b')$ is the composite transformation matrix for two successive translations through the points (a, b) and (a', b'). Similar computations can be done to show that $R(\theta)R(\phi) = R(\theta+\phi)$. (See E2.) Thus the successive rotations are **additive**. This means that if you have to rotate an object by an angle θ followed by an angle ϕ, then this can be done by just rotating the object by $\theta+\phi$ in one go.

Likewise, you can prove that $S(\alpha, \beta)S(\alpha', \beta') = S(\alpha\alpha', \beta\beta')$. (See E2.) So, the successive scalings are **multiplicative**. In other words, applying two successive scalings to an object by (α, β) and (α', β') is equivalent to applying the scaling $(\alpha\alpha', \beta\beta')$ to the object. For instance, in a particular direction first you double the size of an object, and then you triple the size of the resulting object, then the final size of the object is 6 times of the original size in that direction.

Thus far we have seen rotations about the origin only. Composite transformations can be used to rotate an object about any **pivot point** (a,b) through an angle θ. The sequence of transformations you have to use is:

i) translate the object at $(-a,-b)$

ii) rotate the resulting object by θ about the origin.

iii) translate the resulting object at (a,b).

The corresponding composite transformation matrix is

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

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

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

$$= \begin{pmatrix} \cos\theta & -\sin\theta & a' \\ \sin\theta & \cos\theta & b' \\ 0 & 0 & 1 \end{pmatrix},$$

where $a' = a(1-\cos\theta) + b\sin\theta$ and $b' = b(1-\cos\theta) - a\sin\theta$. Observe that the rotation about an arbitrary pivot point (a, b) through an angle θ is equivalent to the rotation about the origin through θ followed by a translation through (a', b'). That is, $R(a, b; \theta) = T(a', b')R(\theta)$.

You have seen that a 2D geometric transformation can be represented by a 3×3 matrix. Since matrix multiplication is not commutative in general, it follows that the composite of two primitive transformations has to be carefully constructed. For example, an object shifted to a position after scaling or an object scaled after shifting, may give different results.

To illustrate this, consider a triangle with vertices in 2D plane given by $(0, 0), (1, 0)$ and $(0, 1)$ (called unit triangle). Translate it by $(1, 1)$ and then scale in each coordinate direction by $1/2$. You get a triangle with vertices $(1/2, 1/2), (1, 1/2)$ and $(1/2, 1)$. Now if you first scale by $1/2$ and then translate it by $(1, 1)$, you will get a different triangle with vertices $(1, 1), (3/2, 1), (1, 3/2)$.

The following example explains how to compute composite transformations in matrix form.

Example 1: Consider a triangle with vertices $(0, 0), (2, 0)$ and $(1,1)$. Rotate it about the point $(4, 3)$ by an angle of $\pi/4$.

Solution: The triangle vertices in homogenous coordinates, $P = \begin{pmatrix} 0 & 2 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$.

We have to actually compute $R_{a',b'}\left(\frac{\pi}{4}\right) \cdot P$, where

$$a' = 4\left(1-\cos\frac{\pi}{4}\right)+3\sin\frac{\pi}{4} = 4\left(1-\frac{1}{\sqrt{2}}\right)+\frac{3}{\sqrt{2}}$$

$$b' = 3\left(1-\cos\frac{\pi}{4}\right)-4\sin\frac{\pi}{4} = 3\left(1-\frac{1}{\sqrt{2}}\right)-\frac{4}{\sqrt{2}}.$$

Thus, we get

$$R\left(4, 3; \frac{\pi}{4}\right) \cdot P = \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 4(1-\frac{1}{\sqrt{2}})+\frac{3}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 3(1-\frac{1}{\sqrt{2}})-\frac{4}{\sqrt{2}} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 2 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 4-\frac{1}{\sqrt{2}} & 4+\frac{1}{\sqrt{2}} & 4-\frac{1}{\sqrt{2}} \\ 3-\frac{7}{\sqrt{2}} & 3-\frac{5}{\sqrt{2}} & 3-\frac{5}{\sqrt{2}} \\ 1 & 1 & 1 \end{pmatrix}$$

Thus, the new coordinates of the vertices of the triangle are

$$\left(4-\frac{1}{\sqrt{2}}, 3-\frac{7}{\sqrt{2}}\right), \left(4+\frac{1}{\sqrt{2}}, 3-\frac{5}{\sqrt{2}}\right), \left(4-\frac{1}{\sqrt{2}}, 3-\frac{5}{\sqrt{2}}\right).$$

***

The OpenGL routines available for applying 2D primitive transformations are as follows.

- `glScale*(`α, β`,1)`: Scale by α in x direction, by β in y direction and no scaling in z direction.
- `glTranslate*(a,b,0)`: Translate by a in x direction, by b in y-direction and no translation in z direction.
- `glRotate*(angle,0,0,1)`: Rotate by 'angle' degrees in anticlockwise direction about the origin. The vector (0, 0, 1) specifies rotation about z-axis.

Here * can be replaced by any one of suffixes f or d, depending on whether the arguments are *floating point* or *double*. These routines are actually in generalised form, that is, they perform 3D transformations. We shall revisit them in Unit 6.

Following simple code demonstrates how these primitives can be applied in a C-program.

```
//Translation, rotation, scaling demo

void display ( void )
{
      glClear (GL_COLOUR_BUFFER_BIT );
      glColour3f(0, 1, 0);
      for(float angle=0; angle <40; angle=angle + 5)
      {
            glScaled(1.5,1.5,1);
            glRotated( angle, 0, 0, 1);
            glTranslated(5,5,0);
            glRectd(-3,-3,3,3);
      }
      glFlush();
}
```

Note the order of the transformations in above code. For each value of angle, first a rectangle is created with $(-3, -3)$ and $(3, 3)$ as its opposite corners. Then it is translated by $(5, 5)$. Next the translated object is rotated by angle. Finally, the rotated object is scaled by 1.5 in both x and y directions.

You may now try the following exercises.

E2) Show that

i) $R(\theta)R(\phi) = R(\theta + \phi)$

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

E3) Show that two successive rotations commute, i.e., $R(\theta)R(\phi) = R(\phi)R(\theta)$. What about two successive translations or scalings?

E4) Does a rotation about the origin through an angle θ commute with a translation through the point (a, b)? Justify.

E5) Magnify a triangle with vertices $A = (1, 1)$, $B = (3, 1)$ and $C = (2, 2)$ to twice its size in such a way that A remains in its original position.

E6) Write the homogeneous transformation matrix for the transformation that rotates an object counterclockwise about the origin by $\pi/2$ and then scales down by one-half in x-direction. Under this transformation, what does the square with vertices $A(1, 0)$, $B(2, 1)$, $C(3, 0)$ and $D(2, -1)$ transform into?

Apart from the basic ones there are some more transformations, which we shall study in the next section.

4.4 REFLECTION AND SHEAR TRANSFORMATIONS

There are a few important transformations which are not primitive in the sense that they can be expressed as compositions of translation, rotation and scaling. These transformations are used extensively in many application softwares using graphics. Two such transformations are *reflection* and *shear*. When you watch a cartoon animation movie for example, many a times you will find that shapes are deformed and a cartoon character is shaking like a pendulum standing at one place. This is basically an application of a continuous shear transformation to give you the appearance of a shaking body. Similarly you will find plenty of applications of reflection in many ad films, cartoon films and also in engineering design softwares.

Let us look at how to obtain reflection matrices, under special cases.

Reflection

Reflecting, in general, a point $P(x, y)$ about a line L means to obtain a point $P'(x', y')$ in such a way that L is a perpendicular bisector of PP'.

Suppose for instance, you wish to reflect a point $P(x, y)$ about the x-axis. We know that the equation of the x-axis is $y = 0$. So, after reflection the x-coordinate remains the same whereas the y-coordinate becomes opposite in sign. That is the reflected point P' has coordinates $(x, -y)$. This can be achieved by the matrix

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

It can be verified that $P' = M_x P$, of course, when P and P' are in homogeneous coordinates.

On the other hand, if you reflect P about the y-axis $(x = 0)$, then the corresponding matrix is

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

In this case, the point after reflection is $P'(-x, y)$.

Likewise, when you reflect a point $P(x, y)$ about the line $y = x$ the

coordinates of the reflected point are $P'(y, x)$. The corresponding reflection matrix is

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

Finally, the reflection matrix for the reflection of a point about the line $y = -x$ is

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

Thus under reflection about $y = -x$ a point $P(x, y)$ becomes $P'(-y, -x)$.

Let us consider an example.

Example 2: Obtain the new coordinates of the vertices of the triangle joining the points $(1, 1), (2, 0)$ and $(3, 1)$ when it is reflected about the lines

i) $y = 0$, ii) $x = 0$, iii) $y = x$, iv) $y = -x$.

Solution:

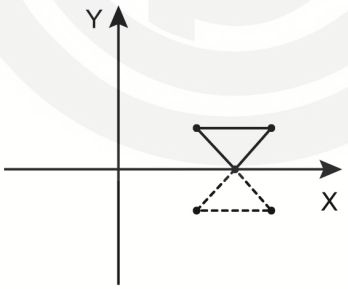
i) About $y = 0$ the new vertices are $(1, -1), (2, 0)$ and $(3, -1)$.

ii) About $x = 0$ the new vertices are $(-1, 1), (-2, 0)$ and $(-3, 1)$.

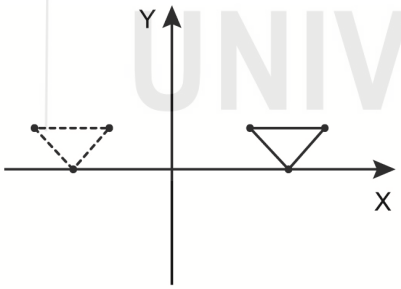
iii) About $y = x$ the new vertices are $(1, 1), (0, 2)$ and $(1, 3)$.

iv) About $y = -x$ the new vertices are $(-1, -1), (0, -2)$ and $(-1, -3)$.

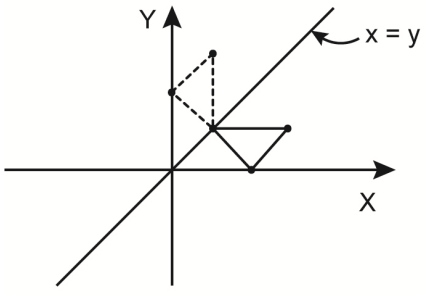
These reflections are geometrically shown below.



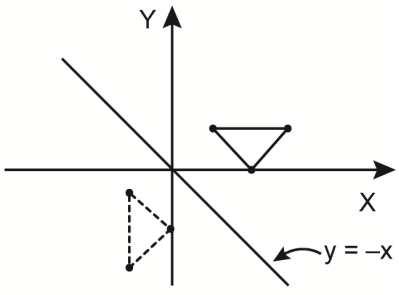
About y = 0



About x = 0



About y = x



About y = –x

***

Shear

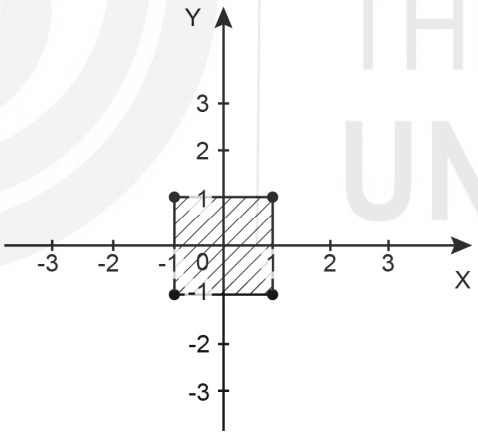
A **shear transformation** is a non-rigid body transformation that causes an object to tilt towards a given direction. The transformation matrix for shearing in the x-direction by a factor α is given below.

$$S_x^{\text{shear}}(\alpha) = \begin{pmatrix} 1 & \alpha & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

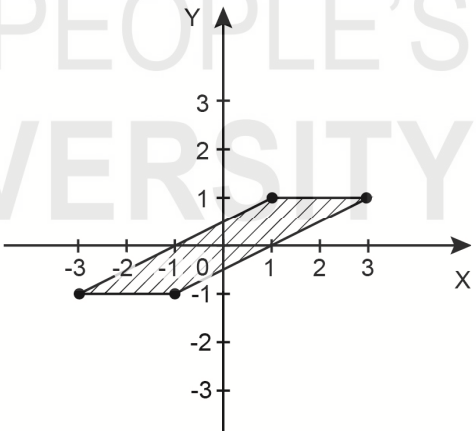
Here α can be any real number. Under this transformation a point $P(x, y)$ changes to a point $P'(x', y')$ as follows:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & \alpha & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + \alpha y \\ y \\ 1 \end{pmatrix}$$

Equivalently, $x' = x + \alpha y,\ y' = y$. Thus, shearing in the x-direction causes no change in the y-coordinate. On the other hand, the shifting in the x-coordinates can be in positive or negative x-direction depending upon the values of α and y. For instance, if both α and y have the same sign then the shifting takes in the positive x-direction, otherwise in the negative x-direction. Obviously, $\alpha = 0$ means no change in the object position. This is illustrated in the following picture.



Original Object



After x direction shear with $\alpha = 2$

The transformation matrix for the shear transformation in the y-direction by a factor β is given below.

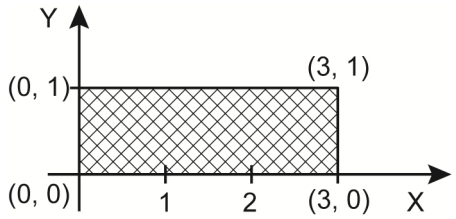
$$S_y^{\text{shear}}(\beta) = \begin{pmatrix} 1 & 0 & 0 \\ \beta & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Under this transformation a point $P(x, y)$ changes to the point $P'(x', y')$ with $x' = x$ and $y' = \beta x + y$. If both β and x have the same sign, then the

shearing takes place in positive y-direction, otherwise in negative y-direction.

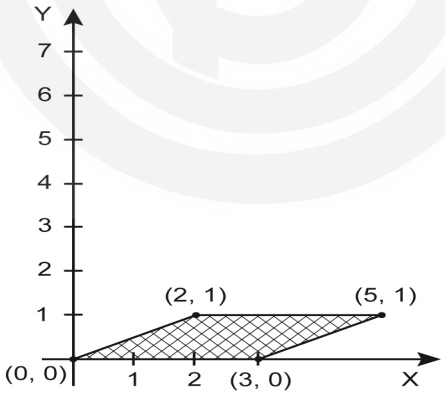
Let us consider an example.

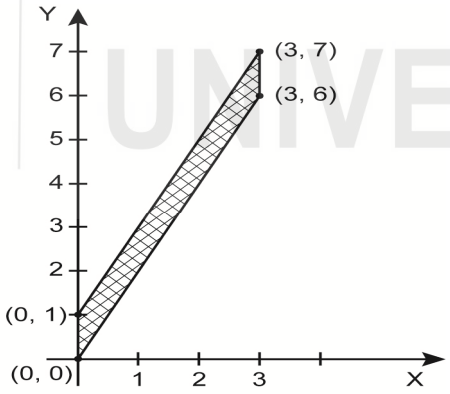
Example 3: Shear the following object in x and y-directions with shearing factors 2 and 1.5, respectively.



Solution: First we shear the object in the x-direction with shearing factor 2.
The coordinates of the points change according to the rule $x' = x + 2y, \quad y' = y.$
Thus the new coordinates of the corners of the object are
$(0, 0), (3, 0), (2, 1), (5, 1).$

Now the shearing in y-direction takes place according to the rule $x' = x$ and $y' = 1.5x + y.$ Therefore, the coordinates of the corners of the object are $(0, 0), (0, 1), (3, 4.5), (3, 5.5).$ The objects after shearing in each direction are drawn as below.





***

Shearing of an object can also be performed in both the directions simultaneously using the following matrix

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

You may now check your understanding of 2D geometric transformations by trying out the following exercises.

E7) A 2D geometric shape is rotated about the point $(1, 2)$ by 90° in a clockwise direction. Then, the shape is scaled in the x-coordinate by 2 times and in the y-coordinate by 3 times. Finally, the shape is reflected about the x-axis. Derive the single combined transformation matrix for these operations.

E8) If you perform an x-direction shear transformation and then a y-direction shear transformation, will the result be the same as the one which is obtained when it is simultaneous shear in both the directions? Justify your answer by appropriate reasoning.

E9) Show that the reflection about one coordinate axis followed by another coordinate axis is equivalent to the rotation about the origin through 180°. That is, show that $M_x M_y = M_y M_x = R(\pi)$.

E10) What is the effect of two successive shearings in the x-direction by a factor of 3 on the point $(1, 5)$?

E11) Show that $S_x^{\text{shear}}(\alpha) S_x^{\text{shear}}(\beta) = S_x^{\text{shear}}(\alpha + \beta)$, for all $\alpha, \beta \in \mathbb{R}$.

After understanding the basic 2D transformations, we shall next discuss the viewing aspects of a 2D image.

4.5 TWO-DIMENSIONAL VIEWING

There are two aspects of viewing a two-dimensional picture on a computer screen:

i) what portion of the picture is to be displayed
ii) where that portion is to be displayed

First of all, for displaying a picture, a Cartesian coordinate system is needed, which we shall call the **world-coordinate system**. The area of the world coordinates selected for display is called a **window** and the area of a display device to which a window is mapped is called a **viewport**. Thus a window defines what is to be displayed, and a viewport defines where it is to be displayed. Note that the window coordinates need not be aligned with the viewport coordinates, see, for instance, Fig. 4.

Viewport coordinates are typically given in normalized coordinates, i.e., in unit square. So, one has to transform the window coordinates into viewing reference frame by appropriate translation and rotation. The translation takes

place at some world position $P_0 = (x_0, y_0)$. Then the rotation takes place according to a **view up vector.**

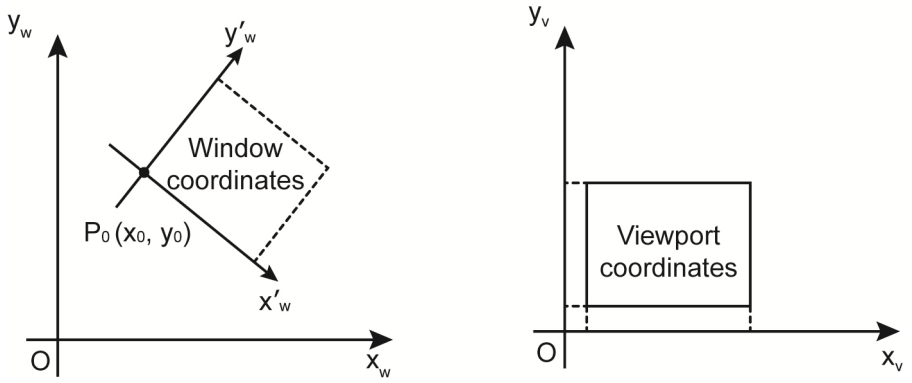


Fig. 4: Window and viewport coordinates

Now having the viewing reference frame aligned with the viewport coordinate axes, the next task is to transform the coordinates from window to viewport. So, let (x_w, y_w) be a point in window coordinates that is mapped to a point (x_v, y_v) in viewport coordinates. (See Fig. 5 below)

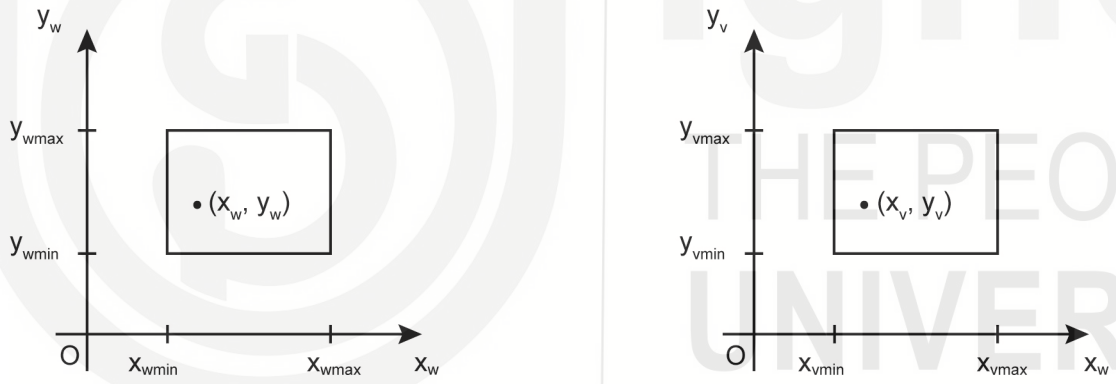


Fig. 5: Window to viewport mapping

Let $\Delta x_w = x_{w\max} - x_{w\min}, \Delta y_w = y_{w\max} - y_{w\min}$,

$\Delta x_v = x_{v\max} - x_{v\min}, \Delta y_v = y_{v\max} - y_{v\min}$.

The mapping of (x_w, y_w) to (x_v, y_v) has to be done in such a way that

$$\frac{x_w - x_{w\min}}{\Delta x_w} = \frac{x_v - x_{v\min}}{\Delta x_v} \text{ and } \frac{y_w - y_{w\min}}{\Delta y_w} = \frac{y_v - y_{v\min}}{\Delta y_v}$$

Solving these equations for x_v and y_v, we get

$$x_v = x_{v\min} + \frac{\Delta x_v}{\Delta x_w}(x_w - x_{w\min})$$

$$y_v = y_{v\min} + \frac{\Delta y_v}{\Delta y_w}(y_w - y_{w\min}).$$

If we write $\alpha = \frac{\Delta x_v}{\Delta x_w}$ and $\beta = \frac{\Delta y_v}{\Delta y_w}$, then the above equations reduce to

$$x_v = x_{v\min} + \alpha(x_w - x_{w\min})$$
$$y_v = y_{v\min} + \beta(y_w - y_{w\min}).$$

In the homogeneous matrix notation, we can write them as

$$\begin{pmatrix} x_v \\ y_v \\ 1 \end{pmatrix} = N \begin{pmatrix} x_w \\ y_w \\ 1 \end{pmatrix},$$

where $N = T(x_{v\min}, y_{v\min}) \cdot S(\alpha, \beta) \cdot T(-x_{w\min}, -y_{w\min})$ is called the **window-to-viewport normalisation transformation matrix**.

Let us look at some examples.

Example 4: Find the normalisation transformation that maps a window with opposite corners at $(1, 1)$ and $(3, 5)$ to a viewport that is the entire normalised device screen.

Solution: We have

$$x_{w\min} = 1,\ y_{w\min} = 1,\ x_{w\max} = 3,\ y_{w\max} = 5$$
$$x_{v\min} = 0,\ y_{v\min} = 0,\ x_{v\max} = 1,\ y_{v\max} = 1$$
$$\therefore\ \alpha = \frac{\Delta x_v}{\Delta x_w} = \frac{x_{v\max} - x_{v\min}}{x_{w\max} - x_{w\min}} = \frac{1-0}{3-1} = \frac{1}{2}$$
$$\beta = \frac{\Delta y_v}{\Delta y_w} = \frac{y_{v\max} - y_{v\min}}{y_{w\max} - y_{w\min}} = \frac{1-0}{5-1} = \frac{1}{4}.$$

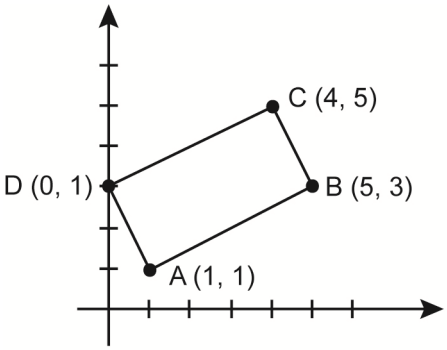
The required transformation is

$$x_v = \frac{1}{2}(x_w - 1),\ y_v = \frac{1}{4}(y_w - 1).$$

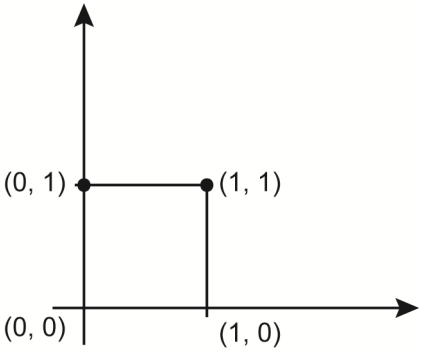
***

Example 5: Find the normalisation transformation matrix N that maps the window with corners at $A(1, 1), B(5, 3), C(4, 5)$ and $D(0, 3)$ to the entire viewport with corners at $(0, 0), (0, 1), (1, 0)$ and $(1, 1)$.

Solution: The window and the viewport are shown below:



Window



Viewport

First, we need to apply translation so that the corner A of the window coincides with the origin. The required translation matrix for this is

$$T(-1,-1)=\begin{pmatrix}1 & 0 & -1\\ 0 & 1 & -1\\ 0 & 0 & 0\end{pmatrix}.$$

Next, we apply a rotation on the window so that its side AB lies on the x-axis. The slope of AB is $\tan\theta=\frac{3-1}{5-1}=\frac{1}{2}$. Therefore, $\cos\theta=\frac{2}{\sqrt{5}}$, $\sin\theta=\frac{1}{\sqrt{5}}$.

The required rotation matrix is

$$R(-\theta)=\begin{pmatrix}\cos(-\theta) & -\sin(-\theta) & 0\\ \sin(-\theta) & \cos(-\theta) & 0\\ 0 & 0 & 1\end{pmatrix}=\begin{pmatrix}\frac{2}{\sqrt{5}} & \frac{1}{\sqrt{5}} & 0\\ -\frac{1}{\sqrt{5}} & \frac{2}{\sqrt{5}} & 0\\ 0 & 0 & 1\end{pmatrix}.$$

Finally, we scale the sides of the window so that it superimposes the viewport. We have $AB=2\sqrt{5}$ and $BC=\sqrt{5}$. Thus to make the length of both the sides equal to 1, we apply the scaling with the factors $\alpha=\frac{1}{2\sqrt{5}}$ and $\beta=\frac{1}{\sqrt{5}}$. The required scaling matrix is

$$S(\alpha,\beta)=\begin{pmatrix}\frac{1}{2\sqrt{5}} & 0 & 0\\ 0 & \frac{1}{\sqrt{5}} & 0\\ 0 & 0 & 1\end{pmatrix}.$$

Thus, the sequence of transformations is

$$N=S(\alpha,\beta)R(-\theta)T(-1,-1)=\begin{pmatrix}\frac{1}{5} & \frac{1}{10} & -\frac{3}{10}\\ -\frac{1}{5} & \frac{2}{5} & -\frac{1}{5}\\ 0 & 0 & 1\end{pmatrix}.$$

***

The following program draws a square with (0,0) and (100, 100) as its opposite corners, and then rotates it continuously by 1 degree about the origin.

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

GLint theta = 0;

void MyInit(void)
{
```

```
    glClearColor(0, 0, 0, 0);
    gluOrtho2D(-500, 500, -500, 500);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
}
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glPushMatrix(); //push transformation matrix to stack
    glRotatef(theta, 0, 0, 1);
    glColor3f(0, 0, 1);
    glRectf(0, 0, 100, 100);
    glPopMatrix();      // pop the matrix from stack
    glutSwapBuffers();  //swap buffers
}


void rotate(void)
{
    theta += 1;     //increase the angle by 1 degree
    theta = theta%360;
    glutPostRedisplay(); //redisplay current window
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutCreateWindow("Moving square ");
    MyInit();
    glutDisplayFunc(display);
    glutIdleFunc(rotate);
    glutMainLoop();
    return 0;
}
```

Listing 1: A moving square program using transformations

To illustrate this program, let us first see what is inside the `display()` function. Note the function call `glRotatef(theta,0,0,1)` before `glRectf(0, 0, 100, 100)`. This means the square is drawn after rotating it by `theta`. To make the rotation effective we have used the function `glPushMatrix()` which pushes the rotation matrix to the stack for multiplication of the corners of the square. Once the square is drawn the rotation matrix must be popped out of the stack, which is done by the function `glPopMatrix()`. Lastly the function `glutSwapBuffers()` just swaps the buffers fast enough to match the animation speed. This is useful when an object is drawn multiple times possibly after applying different transformations on it.

Now look at the function `rotate()`. It just increments `theta` by 1, and if `theta` becomes 360 it resets `theta` to 0. Finally, it calls the function `glutPostRedisplay()` which in turn calls the `display()` function to redraw the current window. The task of the `glutIdleFunc()` is to execute its

argument function, which is rotate ( ) in this case, when no other events are pending.

Now, try the following exercises.

E12) Obtain the viewing transformation matrix, when the window coordinates are $(2, 2), (5, 2), (5, 5), (2, 5)$ and the viewport location is $\left(\frac{1}{2}, 0\right), (1, 0), \left(1, \frac{1}{2}\right), \left(\frac{1}{2}, \frac{1}{2}\right)$.

E13) Modify the program given in Listing 1 so that it rotates a triangle about one of its corners.

We end this unit here.

4.6 SUMMARY

In this unit we have covered the following points.

1. We have discussed the elementary 2D transformations namely, translation, rotation and scaling.
2. We have seen how to use homogeneous coordinates to represent these transformations and also have represented the composition of elementary transformation.
3. We have also discussed reflection and shearing.

4.7 SOLUTIONS / ANSWERS

E1) a) $(1, 3), (5, 3), (5, 0), (1, 0)$

b) $(-\sqrt{3}, 1), (2-\sqrt{3}, 2\sqrt{3}+1), \left(2+\frac{\sqrt{3}}{2}, 2\sqrt{3}-\frac{1}{2}\right), \left(\frac{\sqrt{3}}{2}, -\frac{1}{2}\right)$

c) $(0, 3), (3, 3), \left(3, -\frac{3}{2}\right), \left(0, -\frac{3}{2}\right)$

E2) i) $$R(\theta)\ R(\phi) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

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

ii) Try yourself.

E3) It is evident from E2(i) that $R(\theta)R(\phi) = R(\theta+\phi) = R(\phi+\theta) = R(\phi)R(\theta)$. Similarly, show that $T(a, b)\ T(a', b') = T(a', b')T(a\ b)$ and $S(\alpha, \beta)\ S(\alpha', \beta') = S(\alpha', \beta')\ S(\alpha, \beta)$.

E4) No. We have $R(\theta)T(a, b) \neq T(a, b)\ R(\theta)$ as

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

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

E5) Since A must be kept fixed, A is the pivot point. So, we translate the object at $(-1, -1)$ first. Then a uniform scaling with factor $\alpha = 2$ is applied. Finally, the object is translated back to the point $(1, 1)$. So, the sequence of transformations is

$$M = T(1, 1)S(2, 2)T(-1, -1) = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix}$$

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

Now the transformed points in matrix form are

$$P' = MP = \begin{pmatrix} 2 & 0 & -1 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 3 & 2 \\ 1 & 1 & 2 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 5 & 3 \\ 1 & 1 & 3 \\ 1 & 1 & 1 \end{pmatrix}$$

Thus, the required vertices of the triangle after transformation are $(1, 1)$, $(5, 1)$ and $(3, 3)$.

E6) The required transformation matrix is $M = S\left(\frac{1}{2}, 1\right)R\left(\frac{\pi}{2}\right) = \begin{pmatrix} 0 & -\frac{1}{2} & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

Let P be the matrix whose columns represent the homogenous coordinates of the vertices A, B, C, D of the given square. Then

$$P' = MP = \begin{pmatrix} 0 & -\frac{1}{2} & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 2 & 3 & 2 \\ 0 & 1 & 0 & -1 \\ 1 & 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & -\frac{1}{2} & 0 & \frac{1}{2} \\ 1 & 2 & 3 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix}.$$

So, the rectangle $ABCD$ transforms into the quadrilateral $A'B'C'D'$ where $A' = (0, 1)$, $B' = \left(-\frac{1}{2}, 2\right)$, $C' = (0, 3)$ and $D' = \left(\frac{1}{2}, 2\right)$. Observe that $A'B' = B'C' = C'D' = A'D' = \frac{\sqrt{5}}{2}$. Also, $A'C'$ and $B'D'$ are perpendicular. Therefore, $A'B'C'D'$ is a rhombus.

E7) The required transformation is

$$M = M_x S(2, 3)\ R(90^\circ) T(1, 2)$$

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

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

E8) No. For instance,

$$S_y^{shear}(3) S_x^{shear}(2) = \begin{pmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 2 & 0 \\ 3 & 7 & 0 \\ 0 & 0 & 1 \end{pmatrix} \neq S^{shear}(2, 3).$$

E9) $$M_x M_y = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = R(\pi).$$

Similarly, $M_y M_x = R(\pi)$.

E10) The required transformation matrix is

$$M = S_x^{\text{shear}}(3) S_x^{\text{shear}}(3) = \begin{pmatrix} 1 & 3 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 3 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 6 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Hence, $MP = \begin{pmatrix} 1 & 6 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 5 \\ 1 \end{pmatrix} = \begin{pmatrix} 31 \\ 5 \\ 1 \end{pmatrix}$.

$\therefore$ The point $(1, 5)$ transforms to the point $(31, 5)$.

E11) For all $\alpha, \beta \in \mathbb{R}$,

$$S_x^{shear}(\alpha)S_x^{shear}(\beta)=\begin{pmatrix}1&\alpha&0\\0&1&0\\0&0&1\end{pmatrix}\begin{pmatrix}1&\beta&0\\0&1&0\\0&0&1\end{pmatrix}=\begin{pmatrix}1&\alpha+\beta&0\\0&1&0\\0&0&1\end{pmatrix}=S_x^{shear}(\alpha+\beta)$$

E12) Here $x_{v\min}=\frac{1}{2},\ y_{v\min}=0,\ x_{v\max}=1,\ y_{v\max}=\frac{1}{2}.$ So $\Delta x_v=\frac{1}{2},\ \Delta y_v=\frac{1}{2}.$

Also, $x_{w\min}=2,\ y_{w\min}=2,\ x_{w\max}=5,\ y_{w\max}=5.$ Thus $\Delta x_w=\Delta y_w=3.$

$$\therefore \alpha=\frac{\Delta x_v}{\Delta x_w}=\frac{1}{6},\ \beta=\frac{\Delta y_v}{\Delta y_w}=\frac{1}{6}.$$

The required transformation matrix is

$$N=T\left(\frac{1}{2},0\right)S\left(\frac{1}{6},\frac{1}{6}\right)T(-2,-2)$$

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

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

E13) You need to only replace the display function of Listing 1 by the following.

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1, 0, 0);
    glPushMatrix();
    glRotatef(theta, 0, 0, 1);
    glColor3f(0, 0, 1);
    glBegin(GL_LINE_LOOP);
        glVertex2d(0, 0);
        glVertex2d(100, 0);
        glVertex2d(50, 50);
    glEnd();
    glPopMatrix();
    glutSwapBuffers();
}
```

UNIT 5

CLIPPING ALGORITHMS

Structure **Page No.**

5.1	Introduction Objectives	109
5.2	Clipping Operations in 2D	110
5.3	Cohen-Sutherland Line Clipping	111
5.4	Liang Barsky Line Clipping	117
5.5	Curve and Text Clipping	121
5.6	Bézier and B-Spline Curves	122
5.7	Summary	125
5.8	Solutions/Answers	125

5.1 INTRODUCTION

This unit introduces you to three important concepts of Computer Graphics, line clipping, curve and text clipping, Bézier and B-spline curves. Section 5.2 discusses the concept of **clipping,** which is used to identify and discard the part of an object that lies outside a specified region. This simple concept has far reaching consequences in computer animation and especially in video games which is a fast growing industry. Good clipping strategy is critically important for speeding up the rendering of the current scene and hence plays a crucial role in maximizing game's performance and visual quality. Here we shall discuss two important line clipping algorithms one by Cohen and Sutherland, and another by Liang and Barsky. These algorithms are given in Sections 5.3 and 5.4, respectively. Section 5.5, briefly talks about curve and text clipping.

For surface rendering most suited and common methods are based on **Bézier and B-spline representations** for simple reasons–ease of use and efficiency. Section. 5.6 discusses the Bézier and B-spline curves.

Objectives

After reading this unit you should be able to:

- clip 2D line segments against a rectangular clipping boundary using one of the two line clipping algorithms–Cohen Sutherland line clipping algorithm and Liang Barsky line clipping algorithm;
- explain how a curve and a text string is clipped against a window boundary;
- find equations of cubic Bézier and B-spline curves.

5.2 CLIPPING OPERATIONS IN 2D

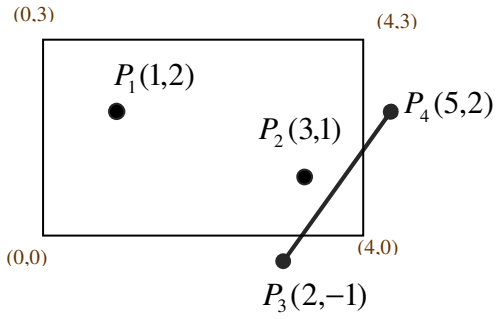
Clipping is an operation that eliminates invisible objects from the view window, often called as the **clip window**. To understand clipping, recall that when we take a snapshot of a scene, we adjust our aperture and focus to bring only desirable objects within the window and the final shot is taken once the window contains only those objects within the frame. From the point of view of mathematics, the problem of clipping looks simple. All that we have to do is to find out the extremities of the object and check their intersection with the clip window. This, however, is a critical problem from the point of view of processing in computer. In gaming industry especially, when animation at a high speed has to be shown, time complexity of clipping methods is an important parameter. You will observe that due to this reason in most of the clipping algorithms, intersection calculations are avoided as far as possible. The basic idea behind these algorithms is to maximize the performance and efficiency by deciding the visibility/partial visibility/invisibility of an object using the minimum number of intersection calculations.

We begin with point clipping which is very simple. All you need to check is whether a point is inside the window extremes in x- and y-directions. Although clipping can be done with respect to any region, in this unit, we shall consider rectangular clip windows only. Moreover, the window is aligned with the coordinate axes.

So, let W be the clip window with boundaries at the lines

$$x = x_{\text{wmin}},\ x = x_{\text{wmax}},\ y = y_{\text{wmin}},\ y = y_{\text{wmax}}.$$

Let $P(x, y)$ be any point. If the coordinates (x, y) of P satisfy the inequalities $x_{\text{wmin}} \le x \le x_{\text{wmax}}$ and $y_{\text{wmin}} \le y \le y_{\text{wmax}}$, then P lies inside W, and hence it is saved for display. Otherwise, P is clipped, i.e., discarded. For instance, look at the following picture



The points P_1 and P_2 lie inside the clip window, and hence they are saved for display. On the other hand, P_3 and P_4 both lie outside the clip window, and hence these points are clipped. Now look at the line segment P_3P_4. Note that despite P_3 and P_4 lying outside clip window, some portion of P_3P_4 lies inside the window. Hence, the line segment cannot be simply clipped.

5.3 COHEN-SUTHERLAND LINE CLIPPING

For line clipping several algorithms have been introduced over the years and many refinements have also been suggested by people in order to improve the performance of the algorithms. One of the first algorithms that is still being used in applications is the **Cohen-Sutherland Line Clipping Algorithm**. It divides line segments into three categories (i) trivially rejected (ii) trivially accepted (iii) partially visible or totally invisible. For category (i) and (ii) no intersection calculations are required whereas category (iii) contains all those line segments that need further processing for determining if they are partially visible or totally invisible.

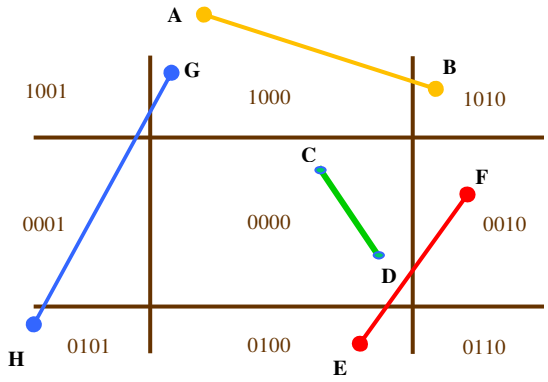
Consider a rectangular clip window W aligned with the coordinate axes of the viewport. Let its boundaries be determined by the lines

$$x = x_{wmin},\ x = x_{wmax},\ y = y_{wmin},\ y = y_{wmax}.$$

Every point of the plane is assigned a **region code**, which is a 4-bit binary code

T	B	R	L

as shown below.

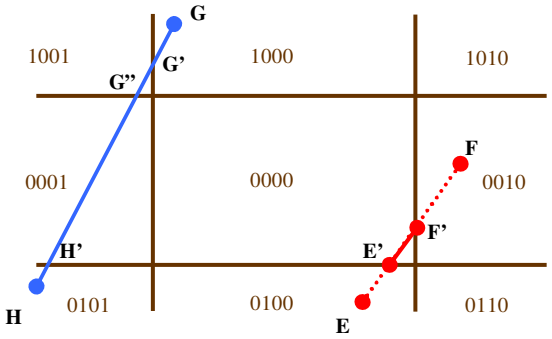


Here, **T** stands for top, **B** for bottom**, R** for right, and **L** for left. The region code of a point tells the position of the point with respect to the clip window. For instance, a point with region code 1010 lies to the top right of the window. A point with region code 0000 lies inside the window.

So, clipping a line requires checking the region codes of each of its end points. Trivially, if both the endpoints of the line have the region code 0000, then the line is saved for display. The line CD shown above is such a line. If the region codes of both the endpoints of a line have a 1 at the same bit position, then the line lies entirely to the left, right, top or bottom of the window. In such a case, the line is **trivially rejected**. So, trivial rejection, technically requires **nonzero bitwise intersection** of the region codes of the endpoints of the line. For instance, look at the line AB. We have $\text{code}(A)=1000$ and $\text{code}(B)=1010$. So, $\text{code}(A)\ \&\ \text{code}(B)=1000$, which is nonzero. Therefore, AB is rejected. Here & denotes the "**bitwise and**" operator.

When a line is not trivially accepted or trivially rejected, it requires a further processing. The lines EF and GH are such lines. Note that such a line has at least one endpoint lying outside the window. Starting from this point, we find the next point where the line intersects with a window edge. In order to do this, we compute the **bitwise intersection** of the code of the point with each of the four window regions LEFT(0001), RIGHT(0010), BOTTOM(0100) and TOP(1000). Note that exactly one of these four regions would have a nonzero intersection with the point.

For instance, consider the line EF in the picture given below. Both of its endpoints E and F have a nonzero region code. Take the endpoint E. We have $\text{code}(E)\ \&\ \text{BOTTOM}=0100$, which is nonzero. So, we replace E with E', where E' is the point of intersection of the line and the window bottom edge.



Discarding the portion EE', we are left with the line segment $E'F$ for further processing. Here, $\text{code}(F) \neq 0$. So, we find $\text{code}(F)\ \&\ \text{RIGHT} = 0010$, which is also nonzero. In this case, we replace F by F', where F' is the point where the line intersects the right window edge. So, we discard the portion FF'. The remaining line segment is $E'F'$ which is saved for display.

Now consider the line GH. If we start with G, then it can be processed in two steps. First discard the portion GG', and then the entire segment HG' as it lies to the left of the left edge. If we start with H, then we need three steps. First discard the portion HH', then GG' and finally $G''H'$.

The detailed procedure of the algorithm is given below.

Procedure (Cohen-Sutherland Line Clipping Algorithm):

Consider a line PQ and the clip window W determined by the lines $x = x_{\text{wmin}}$, $x = x_{\text{wmax}}$, $y = y_{\text{wmin}}$, $y = y_{\text{wmax}}$.

1. Define $\text{LEFT} = 0001,\ \text{RIGHT} = 0010,\ \text{BOTTOM} = 0100,\ \text{TOP} = 1000.$
2. For each point $A(x, y)$ in the plane, compute $\text{code}(A)$ as follows.

   Begin with $\text{code}(A) = 0$. If $x < x_{\text{wmin}}$, set $\text{code}(A) = \text{code}(A) \mid \text{LEFT}$ else if $x > x_{\text{wmax}}$, set $\text{code}(A) = \text{code}(A) \mid \text{RIGHT}$. Further, if $y < y_{\text{wmin}}$, set $\text{code}(A) = \text{code}(A) \mid \text{BOTTOM}$ else if $y > y_{\text{wmax}}$ set $\text{code}(A) = \text{code}(A) \mid \text{TOP}.$

   (Here, | represents the "**bitwise or**" operator.)
3. Compute the slope m of the line PQ.
4. Compute $\text{code}(P)$ and $\text{code}(Q)$ using Step 2. If $\text{code}(P) = \text{code}(Q) = 0$, then trivially accept PQ and store it in the frame buffer for display, and stop.
5. If $\text{code}(P)$ & $\text{code}(Q)$ ≠ 0, then trivially reject PQ and stop. Otherwise, at least one of $\text{code}(P)$ and $\text{code}(Q)$ is nonzero, say $\text{code}(P) \neq 0$.
6. Replace the coordinates (x, y) of P with one of the following and go to Step 4.

   i) $(x_{\text{wmin}}, y + m(x_{\text{wmin}} - x))$ if $\text{code}(P)\ \&\ \text{LEFT} \neq 0$

   ii) $(x_{\text{wmax}}, y + m(x_{\text{wmax}} - x))$ if $\text{code}(P)\ \&\ \text{RIGHT} \neq 0$

   iii) $\left(x + \frac{y_{\text{wmin}} - y}{m}, y_{\text{wmin}}\right)$ if $\text{code}(P)\ \&\ \text{BOTTOM} \neq 0$

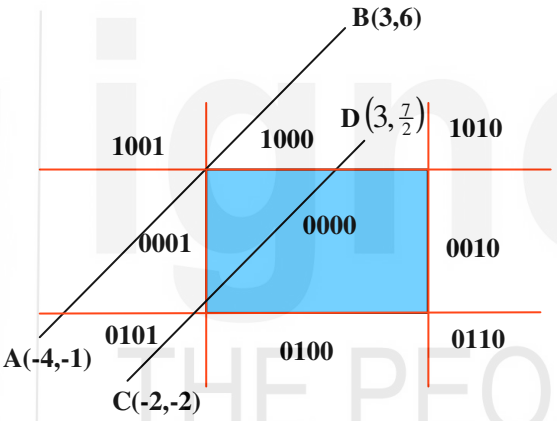
iv) $\left(x + \frac{y_{wmax} - y}{m}, y_{wmax}\right)$ if $\text{code}(P) \,\&\, \text{TOP} \neq 0$

Note that in case of vertical lines $m = \infty$. So, in Step 6(iii) or 6(iv), x-coordinate would not change. These things are handled during the implementation of the algorithm.

Let us consider an example, now.

Example 1: Use Cohen-Sutherland Line Clipping Algorithm to clip the lines AB and CD against the clip window W with corners at (0, 0), (4, 0), (4, 3) and (0, 3), given that $A = (-4,-1), B = (3,6), C = (-2,-2)$, and $D = \left(3, \frac{7}{2}\right)$.

Solution: The lines AB and CD along with the clip window W are drawn as below.



We have $x_{wmin} = 0,\ x_{wmax} = 4,\ y_{wmin} = 0,\ y_{wmax} = 3.$
Let us first process AB for clipping. The slope of AB is $m = 1$. Note that $\text{code}(A) = 0101 \neq 0$ and $\text{code}(B) = 1000 \neq 0$. So, the line is not trivially accepted. Also, $\text{code}(A) \,\&\, \text{code}(B) = 0000 = 0$. So, AB cannot be trivially rejected as well. We need to process it further. Pick the point A as $\text{code}(A) \neq 0$. We have $\text{code}(A) \,\&\, \text{BOTTOM} = 0101 \,\&\, 0100 = 0100 \neq 0$. So, the new coordinates of A are $\left(x + \frac{y_{wmin} - y}{m}, y_{wmin}\right) = (-4 + 1, 0) = (-3, 0).$

Again, we have $\text{code}(A) = 0001 \neq 0$. Also, $\text{code}(A) \,\&\, \text{LEFT} = 0001 \,\&\, 0001 = 0001 \neq 0$. Therefore, the new coordinates of A are $(x_{wmin}, y + m(x_{wmin} - x)) = (0, 3)$. This time we have $\text{code}(A) = 0$. So, we pick $B(3,6)$ as $\text{code}(B) \neq 0$. We find that $\text{code}(B) \,\&\, \text{TOP} = 1000 \neq 0$. Hence, the new coordinates of B are $\left(x + \frac{y_{wmax} - y}{m}, y_{wmax}\right) = \left(3 + \frac{3-6}{1}, 3\right) = (0, 3).$

Now both A and B have code zero. Therefore, we save AB for display, which is actually a point.

Now we clip the line CD which has slope $m = \frac{11}{10}$. Note that $\text{code}(C) = 0101 \neq 0$ and $\text{code}(D) = 1000 \neq 0$. Hence, CD is not trivially accepted. Also, $\text{code}(C) \,\&\, \text{code}(D) = 0$, so CD is not trivially rejected as well. So, we process it further, starting from the point C. Since, $\text{code}(C) \,\&\, \text{BOTTOM} = 0100 \neq 0$, the new coordinates of C are $\left(-2 + \frac{0+2}{11/10}, 0\right) = \left(-\frac{2}{11}, 0\right)$. Again, $\text{code}(C) = 0001 \neq 0$. So, we find that $\text{code}(C) \,\&\, \text{LEFT} = 0001 \neq 0$. Therefore, the coordinates of C now become $\left(0, \frac{1}{5}\right)$. This time $\text{code}(C) = 0$. So, we pick $D\left(3, \frac{7}{2}\right)$ as $\text{code}(D) \neq 0$. We have $\text{code}(D) \,\&\, \text{TOP} = 1000 \neq 0$. Hence, the new coordinates of D are $\left(3 + \frac{3 - \frac{7}{2}}{\frac{11}{10}}, 3\right) = \left(\frac{28}{11}, 3\right)$.

Now we have $\text{code}(D) = 0$. Therefore, the line CD after clipping has endpoints $\left(0, \frac{1}{5}\right)$ and $\left(\frac{28}{11}, 3\right)$.

***

The C program that implements the Cohen Sutherland Algorithm is given in Listing 1 below.

```
// Cohen-Sutherland Line Clipping Algorithm
#include<stdio.h>
#include<GL/glut.h>

#define LEFT 0x1
#define RIGHT 0x2
#define BOTTOM 0x4
#define TOP 0x8

#define FALSE 0
#define TRUE 1

#define inside(a) (!a)
#define reject(a, b) (a & b)
#define accept(a, b) (!(a|b))

typedef struct point{ float x, y; }Point;


void clip_line_cohen_sutherland(Point Wmin, Point Wmax, Point
P, Point Q)
{
    unsigned char code1, code2;
    int done = FALSE, draw = FALSE;
    float m;
    while(!done)
    {
        code1 = encode(P, Wmin, Wmax);
        code2 = encode(Q, Wmin, Wmax);
        if(accept(code1, code2)){
            done = TRUE;
```

```
                draw = TRUE;
            }
            else if(reject(code1, code2))
                    done = TRUE;
                else
                {       //make sure that P is outside the window
                    if(inside(code1)){
                       swap_points(&P, &Q);
                       swap_codes(&code1, &code2);
                    }
                    if(Q.x != P.x)
                        m = (Q.y-P.y)/(Q.x-P.x);
                    if(code1 & LEFT){
                        P.y += m*(Wmin.x - P.x);
                        P.x = Wmin.x;
                    }
                    else if(code1 & RIGHT){
                            P.y += m*(Wmax.x - P.x);
                            P.x = Wmax.x;
                        }
                        else if(code1 & BOTTOM){
                                if(Q.x != P.x)
                                    P.x += (Wmin.y-P.y)/m;
                                P.y = Wmin.y;
                            }
                            else if(code1 & TOP){
                                    if(Q.x != P.x)
                                        P.x += (Wmax.y - P.y)/m;
                                    P.y = Wmax.y;
                                }
                }
        }
        if(draw)
            draw_line(P, Q);
}

void myInit (void)
{
    gluOrtho2D(0, 1000, 0, 1000);
    glClearColor(0, 0, 0, 0.0);
    glColor3f(1, 1, 1);
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    Point Wmax, Wmin, P, Q;

    printf("Enter clip window dimensions:\n ");
    printf("bottom left corner : ");
    scanf("%f%f", &Wmin.x, &Wmin.y);
    printf("top right corner :");
    scanf("%f%f", &Wmax.x, &Wmax.y);
    printf("Enter the endpoints of the line to be clipped.\n");
    printf("First endpoint  :");
    scanf("%f%f", &P.x, &P.y);
    printf("Second endpoint :");
    scanf("%f%f", &Q.x, &Q.y);


    glRectf(Wmin.x, Wmin.y, Wmax.x, Wmax.y);
    glColor3f(0, 0, 1);
    draw_line(P, Q);

    glColor3f(0, 1, 0);
    clip_line_cohen_sutherland(Wmin, Wmax, P, Q);
```

```
}


int main (int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

    glutInitWindowSize(1000, 1000);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Cohen-Sutherland Line Clipping");

    myInit();
    glutDisplayFunc(display);
    glutMainLoop();
}
```

Listing 1: Cohen-Sutherland Line Clipping Algorithm

Some of the functions we have not defined in the above program. You are asked to write their definitions. (See E3.)

You may now test your understanding by trying the following exercises.

E1) Let W be a window having two diagonally opposite corners at $(1, 1)$ and $(5, 4)$. Trace Cohen-Sutherland Line Clipping Algorithm for the line segment having two endpoints $(0, 0)$ and $(4, 5)$.

E2) Let W be the window as given in E1). Clip a triangle with vertices $(0, 0)$, $(4, 5)$ and $(6, 1)$ against the window by tracing Cohen Sutherland line clipping algorithm for each of the edges.

E3) Write the definitions of the functions `encode()`, `swap_points()`, `swap_codes()` and `draw_line()` used in the Cohen Sutherland Line Clipping Algorithm.

In the next section, we shall discuss one more clipping algorithm.

5.4 LIANG BARSKY LINE CLIPPING

Laing Barsky Line Clipping Algorithm, or simply, **Liang Barsky algorithm** uses the parametric form of a line segment to clip it. This method is faster than the Cohen Sutherland method in general. Note that the line segment joining the points $P(x_1, y_1)$ and $Q(x_2, y_2)$ can be written in parametric form as

$$x = x_1 + t\Delta x, \quad y = y_1 + t\Delta y,$$

where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$. The parameter t can vary from 0 to 1, where $t = 0$ refers to the point P, and $t = 1$ to the point Q. To clip any point on PQ we must test the following point clipping conditions:

$$x_{wmin} \le x_1 + t\Delta x \le x_{wmax}$$
$$y_{wmin} \le y_1 + t\Delta y \le y_{wmax}$$

These can be rewritten as

$$x_{wmin} - x_1 \le t\Delta x \le x_{wmax} - x_1$$
$$y_{wmin} - y_1 \le t\Delta y \le y_{wmax} - y_1.$$

Let us now define the parameters p_k and q_k as follows:

$$p_1 = -\Delta x, \qquad q_1 = x_1 - x_{wmin}$$
$$p_2 = \Delta x, \qquad q_2 = x_{wmax} - x_1$$
$$p_3 = -\Delta y, \qquad q_3 = y_1 - y_{wmin}$$
$$p_4 = \Delta y, \qquad q_4 = y_{wmax} - y_1$$

Using these parameters, we can rewrite the above inequalities as

$$-q_1 \le -tp_1 \le q_2, \quad -q_1 \le tp_2 \le q_2$$
$$-q_3 \le -tp_3 \le q_4, \quad -q_3 \le tp_4 \le q_4$$

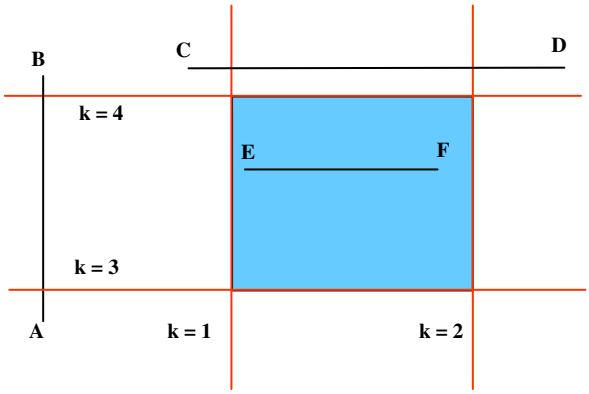
From these inequalities we find that if a point (x, y) lies inside W then the following must hold:

i) $q_k \ge 0$ whenever $p_k = 0$

ii) $t \ge \frac{q_k}{p_k}$ whenever $p_k < 0$

iii) $t \le \frac{q_k}{p_k}$ whenever $p_k > 0$.

In other words, $p_k = 0$ and $q_k < 0$ means that the point (x, y) lies outside W. Since $p_k = 0$ also means that the line is parallel to one of the window edges, it follows the whole line must lie outside W. For instance, look at the lines AB and CD in the picture given below.



For AB we have $p_1 = 0$ and $q_1 < 0$, and hence it is rejected. In case of CD we have $p_4 = 0$, $q_4 < 0$, so it is also rejected. On the other hand, the line EF is accepted because for it $p_3 = 0$, $q_3 \ge 0$ and $p_4 = 0$, $q_4 \ge 0$.

The cases (i) and (ii) can be combinedly written as

$$t_1 = \max_{\{k|p_k<0\}}\left\{0, \frac{q_k}{p_k}\right\} \le t \le \min_{\{k|p_k>0\}}\left\{\frac{q_k}{p_k}, 1\right\} = t_2,$$

where, of course, we have taken into account the fact that $0 \le t \le 1$. Thus if $t_1 \le t_2$, then the line can be clipped at the points corresponding to $t = t_1$ and $t = t_2$. Otherwise, the line is rejected.

Let us recapitulate the main steps involved in Liang-Barsky algorithm.

Procedure (Liang Barsky Line Clipping Algorithm):
Consider a line PQ with endpoints $P(x_1, y_1)$ and $Q(x_2, y_2)$. Let the clip window W be determined by the lines $x = x_{\text{wmin}}$, $x = x_{\text{wmax}}$, $y = y_{\text{wmin}}$, $y = y_{\text{wmax}}$. Assuming $\Delta x = x_2 - x_1$, $\Delta y = y_2 - y_1$, define the parametric equation of PQ as $x = x_1 + t\Delta x$; $y = y_1 + t\,\Delta y$, where $0 \le t \le 1$. The algorithm proceeds as follows:

1. Define the constants p_k, q_k $(k = 1, 2, 3, 4)$ as follows:

   i) $p_1 = -\Delta x, \quad p_2 = \Delta x, \quad p_3 = -\Delta y, \quad p_4 = \Delta y$

   ii) $q_1 = x_1 - x_{\text{wmin}}$, $q_2 = x_{\text{wmax}} - x_1$, $q_3 = y_1 - y_{\text{wmin}}$, $q_4 = y_{\text{wmax}} - y_1$.

2. Set $t_1 = 0$ and $t_2 = 1$. For $k = 1, \ldots, 4$

   i) if $p_k = 0$ and $q_k < 0$ then reject PQ, and stop.

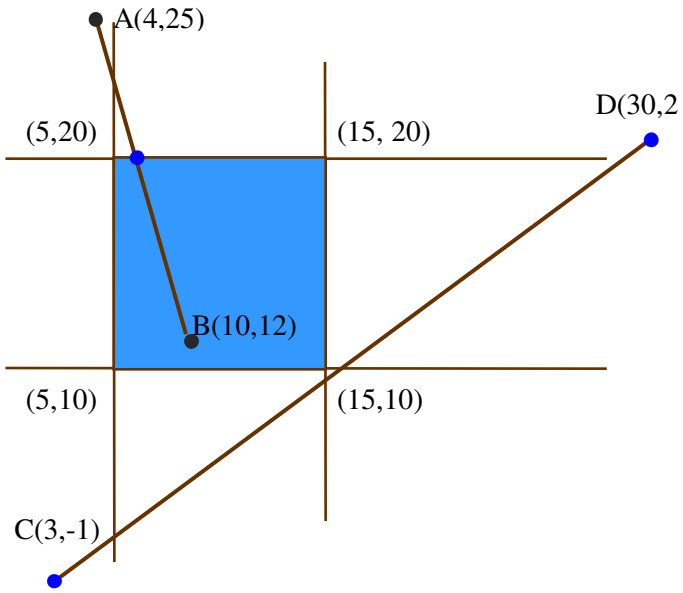
   ii) if $p_k < 0$, set $t_1 = \max\left\{t_1, \frac{q_k}{p_k}\right\}$.

   iii) if $p_k > 0$, set $t_2 = \min\left\{t_2, \frac{q_k}{p_k}\right\}$.

3. If $t_1 > t_2$ then reject PQ, else store it in the frame buffer for plotting for parameter values t such that $t_1 \le t \le t_2$.

Let us consider an example.

Example 2: Let W be the clip window with bottom left corner at $(5,10)$ and top right corner at $(15,20)$ as shown below. Use the Liang Barsky Line Clipping Algorithm to clip the lines AB and CD, where $A = (4,25), B = (10,12), C = (3,-1)$ and $D = (30, 22)$.



Solution: Here $x_{wmin} = 5, y_{wmin} = 10, x_{wmax} = 15, y_{wmax} = 20.$ Let us first consider the line AB for clipping. Take A as the starting point and B as the ending point, we have $x_1 = 4, y_1 = 25, x_2 = 10, y_2 = 12.$ So, $\Delta x = 10 - 4 = 6$ and $\Delta y = 12 - 25 = -13.$ The parametric equations are

$$x(t) = x_1 + t\Delta x = 4 + 6t\ ,\ \ y(t) = y_1 + t\Delta y = 25 - 13t.$$

So, the constants p_k and q_k are:

$$p_1 = -6, p_2 = 6, p_3 = 13, p_4 = -13,$$
$$q_1 = -1, q_2 = 11, q_3 = 15, q_4 = -5.$$

Since, p_1 and p_4 are negative, we compute

$$t_1 = \max\left\{0, \frac{q_1}{p_1}, \frac{q_4}{p_4}\right\} = \max\left\{0, \frac{1}{6}, \frac{5}{13}\right\} = \frac{5}{13}.$$

On the other hand, p_2 and p_3 are positive. So,

$$t_2 = \min\left\{1, \frac{q_2}{p_2}, \frac{q_3}{p_3}\right\} = \min\left\{1, \frac{11}{6}, \frac{15}{13}\right\} = 1.$$

Thus, $t_1 \le t_2$. Hence, the line after clipping has endpoints $(x(t_1), y(t_1))$ and $(x(t_2), y(t_2))$, where

$$x(t_1) = 4 + 6 \times \frac{5}{13} \approx 6.3,\ y(t_1) = 25 - 13 \times \frac{5}{13} = 20,$$
$$x(t_2) = 4 + 6 \times 1 = 10, y(t_2) = 25 - 13 \times 1 = 12.$$

Now let us process the line CD for clipping, taking C as the starting point and D as the ending point. We have $x_1 = 3, y_1 = -1, x_2 = 30, y_2 = 22.$ So, $\Delta x = 27$ and $\Delta y = 23.$ The parametric equations are $x(t) = 3 + 27t,\ y(t) = -1 + 23t$ and the constants p_k and q_k are:

$$p_1 = -27, p_2 = 27, p_3 = -23, p_4 = 23,$$
$$q_1 = 2, q_2 = 12, q_3 = -11, q_4 = 21.$$

Then $t_1 = \max\left\{0, \frac{q_1}{p_1}, \frac{q_3}{p_3}\right\} = \frac{11}{23} \approx 0.48$ and $t_2 = \min\left\{1, \frac{q_2}{p_2}, \frac{q_4}{p_4}\right\} = \frac{4}{9} \approx 0.44.$

Since $t_1 > t_2$, the line is rejected.

***

You may now test your understanding of the algorithm by trying the following exercise.

E4) Use Liang Barsky line clipping algorithm to clip a rectangle $PQRS$ with vertices $P(-1, 2), Q(3, -1), R(7, 2), S(3, 5)$ against the clip window with vertices $(0, 0), (6, 0), (6, 4)$ and $(0, 4)$.

As you proceed further, you would see that the methods for curve clipping and character clipping have also been developed based on bounding rectangles. This means for a given 2D object, you can find out the bounding rectangle with edges parallel to the coordinate axes (parallel to the sides of clipping window). In case the bounding rectangle does not intersect the clipping window, reject the object. Otherwise, find out partial/ total visible objects. Intersection calculations with the window boundary are performed for partial visible objects, whenever essential.

5.5 CURVE AND TEXT CLIPPING

Curve clipping methods use similar approaches as in line clipping. Since the equations of curves are nonlinear, curve clipping require more computations then line clipping. First a box enclosing the curve is identified, then the intersection of this box are computed with the clipping window. If the box completely lies inside the clip window, the curve is saved for display. If the box lies completely outside the clip window, the curve is clipped. If the box overlaps with the clip window, we need other ways to clip the portion of the curve falling outside the clip window. For instance, in circle clipping we can use the coordinate extents for preliminary checking. Similarly, for clipping ellipses, we use coordinate extents of individual quadrants for initial checking, and compute intersection of the curve with the clip window.

Text clipping can be done in several ways. Two simplest methods are ***all-or-none-string-clipping*** and ***all-or-none-character-clipping***. In **all-or-none-**

string-clipping method it is checked whether the bounding rectangle lies completely inside the clip window or not. If yes, the string is saved for display, otherwise it is discarded. In certain cases however, we might need to retain the portion of the string falling inside the clip window. For such cases, we use **all-or-none-character-clipping**. In this method each character box is checked for intersection with the clip window, and the portion that falls outside is clipped.

You can try the following exercise.

E5) Write an algorithm to clip an ellipse against a rectangular window.

E6) Design a text clipping algorithm using the "all-or-none character clipping" strategy. You may assume that all the characters have the same width.

We next turn to study some of the important curve and surface generation techniques. More precisely, we shall study Bezier and B-spline curves with emphasis on their properties suitable for geometric modelling.

5.6 BEZIER AND B-SPLINE CURVES

You have already seen the definition of Bézier curve in Unit 2. Given $n+1$ control points $P_0, P_1, \ldots, P_n$ a Bézier curve P of degree n is defined as

$$P(t) = \sum_{i=0}^{n} P_i B_i^n(t)$$
$$= P_0 B_o^n(t) + P_1 B_1^n(t) + \cdots + P_n B_n^n(t)$$

where B_i^n is the i^{th} Bernstein function defined by

$$B_i^n(t) = \binom{n}{i} (1-t)^{n-i} t^i .$$

The curve P is defined on the interval $[0, 1]$, i.e., t varies from 0 to 1. Note that $B_0^n(t) = (1-t)^n$ and $B_n^n(t) = t^n$. Thus

$$B_0^n(0) = 1, \; B_i^n(0) = 0 \text{ for } i = 1, 2, \ldots, n$$
$$B_n^n(1) = 1, \; B_i^n(1) = 0 \text{ for } i = 0, 1, 2, \ldots, n-1.$$

This means $P(0) = P_0$ and $P(1) = P_n$. That is, P passes through the first and the last control points. Indeed, P does not pass through any other control point as $B_i^n(t) \neq 0$ for $1 \leq i \leq n-1$.

Let us look at a particular case of Bézier curves, namely the **cubic Bézier curves**, due to their immense applications in curve design. They do not

require much computation also. A cubic Bézier curve is generated by four control points P_0, P_1, P_2 and P_3. It can be expressed in matrix form as

$$P(t) = P_0\ B_0^3(t) + P_1\ B_1^3(t) + P_2\ B_2^3(t) + P_3\ B_3^3(t)$$

$$= (B_0^3(t)\ \ B_1^3(t)\ \ B_2^3(t)\ \ B_3^3(t)) \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix}$$

$$= (-t^3 + 3t^2 - 3t + 1,\ 3t^3 - 6t^2 + 3t,\ -3t^3 + 3t^2,\ t^3) \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix}$$

$$= (t^3\ \ t^2\ \ t\ \ 1) \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix}.$$

Thus a cubic Bézier curve can easily be constructed with the help of the matrix

$$M_{BEZ} = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix},$$

which is called the **Bézier matrix**.

Let us look at an example.

Example 3: Find the equation of the cubic Bézier curve with control points $(0, 0), (14, 10), (4, 0)$ and $(–4, 2)$.

Solution: Here, $P_0 = (0, 0),\ P_1 = (14, 10),\ P_2 = (4, 0)$ and $P_3 = (–4, 2)$. The required equation is

$$P(t) = (t^3\ \ t^2\ \ t\ \ 1)\ M_{BEZ} \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix}$$

$$= (t^3\ \ t^2\ \ t\ \ 1) \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 14 & 10 \\ 4 & 0 \\ -4 & 2 \end{pmatrix}$$

$$= (t^3\ \ t^2\ \ t\ \ 1) \begin{pmatrix} 26 & 32 \\ -72 & -60 \\ 42 & 30 \\ 0 & 0 \end{pmatrix}$$

$$= (26t^3 - 72t^2 + 42t,\ 32t^3 - 60t^2 + 30t).$$

***

B-spline curves are piecewise polynomial curves with one or more polynomial pieces with a minimum smoothness requirement. For example, a C^2 cubic B-spline curve has one or more cubic pieces which are joined in such a way that the resulting composite curve is twice continuously differentiable.

B-spline curves have a very elegant expression similar to the Bezier representation with basis function as B-spline functions in place of Bernstein basis polynomials. B-spline basis functions can be computed iteratively using de Boor algorithm.

We shall present here a formula for **uniform cubic B-spline curve** generated by four control points say P_0, P_1, P_2 and P_3. The Formula is

$$P(t) = (t^3 \ t^2 \ t \ 1)\ M_B \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix},$$

where $M_B = \frac{1}{6}\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix}$ is called the B-spline matrix.

Let us take an example.

Example 4: Find uniform cubic B-spline curve generated by the control points $(1, 2), (2, -2), (4, 6), (8, 0)$.

Solution: Here $P_0 = (1, 2),\ P_1 = (2, -2),\ P_2 = (4, 6)$ and $P_3 = (8, 0)$. the required curve is

$$P(t) = \frac{1}{6}(t^3 \ t^2 \ t \ 1)\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix}\begin{pmatrix} 1 & 2 \\ 2 & -2 \\ 4 & 6 \\ 8 & 0 \end{pmatrix}$$

$$= \frac{1}{6}(t^3 \ t^2 \ t \ 1)\begin{pmatrix} 1 & -26 \\ 3 & 12 \\ 9 & 12 \\ 13 & 0 \end{pmatrix}$$

$$= \frac{1}{6}(t^3 + 3t^2 + 9t + 13,\ -26t^3 + 12t^2 + 12t)$$

The following exercises will further help you to test your understanding of the concepts learnt above.

E7) Let $P(t)$ be the cubic Bezier curve defined on the interval $[0, 1]$ with control points $P_0(0, 0)$, $P_1(6, 10)$, $P_2(10, 10)$ and $P_3(20, 0)$. Find $P(t)$.

E8) Find the uniform cubic B-spline curve generated by the control points $(1, 4)$, $(2, 6)$, $(4, 0)$ and $(5, 3)$.

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

5.7 SUMMARY

In this unit, we have covered the following:

1. The concept of clipping 2D objects is introduced.
2. Cohen Sutherland line clipping algorithm is discussed with C implementation.
3. Liang-Barsky Line Clipping algorithm is described.
4. Brief introduction to text and curve clipping is given.
5. Bezier and B-spline curves are discussed, in brief.

5.8 SOLUTIONS/ANSWERS

E1) Proceed step by step through the algorithm. Let $P = (0, 0)$ and $Q = (4, 5)$. We find that code $(P) = 0101$, code $(Q) = 1000$.

Clearly code $(P) \neq 0$ and code $(Q) \neq 0$. Also code (P) & code $(Q) = 0$. Therefore, the line segment requires further processing. For the point P, we see that code (P) & LEFT $\neq 0$. Therefore, the new coordinates of P are $\left(1, \frac{5}{4}\right)$. Now since code (Q) & TOP $\neq 0$, we find the new coordinates of Q as $\left(\frac{16}{5}, 4\right)$. Now code $(P) =$ code $(Q) = 0$. Therefore, the line segment PQ is saved for display.

E2) Let the vertices of the triangle be $P(0, 0)$, $Q(4, 5)$ and $R(6, 1)$. From E1 way have seen that the line segment PQ after clipping has endpoints

$P'\left(1, \frac{5}{4}\right)$ and $Q'\left(\frac{16}{5}, 4\right)$.

Next, we process QR. Note that code $(Q) = 1000$ and code $(R) = 0010$, so QR needs to be processed further. We start with Q. Since code (Q) & $TOP \neq 0$. Since the slope of QR is -2, the new coordinates of Q become $Q'\left(\frac{9}{2}, 4\right)$. Now code $(Q') = 0000$. But code $(R) \neq 0$. So, we take R for further processing. We see that code (R) & RIGHT $\neq 0$. Therefore, the new coordinates of R are $R'(5, 3)$. Now code $(R') = 0000$. Thus $Q'R'$ is saved for display

Finally we consider the line segment PR. Since code $(P) \neq 0$ and code $(R) \neq 0$, the line cannot be trivially accepted. Since code (P) & code $(R) = 0$, the line cannot be trivially rejected as well. Now consider P. Since code (P) & BOTTOM $\neq 0$, the new coordinates of P are $P' = (6, 1) = R$. So PR is rejected.

E3) The required codes are given below.

```
unsigned char encode(Point P, Point Wmin, Point Wmax)
{
    unsigned char code = 0x00;
    if(P.x < Wmin.x)
        code = code | LEFT;
    else if(P.x > Wmax.x)
            code = code | RIGHT;
    if(P.y < Wmin.y)
        code = code | BOTTOM;
    else if(P.y > Wmax.y)
        code = code | TOP;
    return (code);
}

void swap_points(Point *P, Point *Q)
{
    Point temp = *P;
    *P = *Q;
    *Q = temp;
}

void swap_codes(unsigned char *c1, unsigned char *c2)
{
    unsigned char temp = *c1;
    *c1 = *c2;
    *c2 = temp;
}

void draw_line(Point P, Point Q)
{
    glBegin(GL_LINES);
        glVertex2f(P.x, P.y);
        glVertex2f(Q.x, Q.y);
```

```
        glEnd();
        glFlush();
    }
```

E4) Note that $x_{w\min} = 0,\ x_{w\max} = 6,\ y_{w\min} = 0,\ y_{w\max=4}$.
First let us process the line segment PQ. Take $x_1 = -1,\ y_1 = 2,\ x_2 = 3,$ $y_2 = -1$. Then $\Delta x = 4, \Delta y = -3$. Hence, the parametric equations of PQ are

$$x(t) = -1 + 4t,\ y(t) = 2 - 3t$$

Now, we find

$$p_1 = -4,\ p_2 = 4,\ p_3 = 3,\ p_4 = -3$$
$$q_1 = -1,\ q_2 = 7,\ q_3 = 2,\ q_4 = 2$$

Since p_1 and p_4 are negative, we get

$$t_1 = \max\left\{0, \frac{q_1}{p_1}, \frac{q_4}{p_4}\right\} = \left\{0, \frac{1}{4}, -\frac{2}{3}\right\} = \frac{1}{4}.$$

Also p_2 and p_3 are positive, so

$$t_2 = \min\left\{1, \frac{q_2}{p_2}, \frac{q_3}{p_3}\right\} = \min\left\{1, \frac{7}{4}, \frac{2}{3}\right\} = \frac{2}{3}.$$

Thus, $t_1 \le t_2$. Therefore, the line segment PQ is clipped to $P'Q'$, where

$$P' = (x(t_1),\ y(t_1)) = \left(0, \frac{5}{4}\right),\ Q' = (x(t_2),\ y(t_2)) = \left(\frac{5}{3}, 0\right).$$

Similarly, you can clip other line segments. You must find that

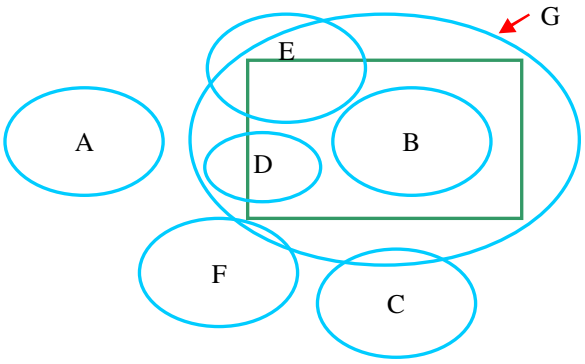
QR is clipped to $Q''R'$, where $Q'' = \left(\frac{13}{3}, 0\right),\ R' = \left(6, \frac{5}{4}\right)$

RS is clipped to $R''S'$, where $R'' = \left(6, \frac{11}{4}\right),\ S' = \left(\frac{13}{3}, 4\right)$

PS is clipped to $P''S''$, where $P'' = \left(0, \frac{11}{4}\right),\ S'' = \left(\frac{5}{3}, 4\right)$.

E5) For the sake of simplicity, assume that both the ellipse and the window have axes/edges parallel to the coordinate axes. Suppose the ellipse E has centre $(x_c,\ y_c)$ with semi-major and semi-minor axes of lengths a and b respectively. Further, let the window W have two extreme corners $(x_{w\min},\ y_{w\min})$ and $(x_{w\max},\ y_{w\max})$. Step by step algorithm, based on which you may write a routine for clipping an ellipse is as follows:

Define the bounding box R of E as the smallest rectangle that contains E. It is the rectangle with two extreme corners as $(x_c - a,\ y_c - b)$ and $(x_c + a,\ y_c + b)$. Some possible situations are shown below with ellipses from A to G.



Step 1: Trivial Reject Case: (Ellipse A)
Check if R is disjoint from W. This amounts to performing the following test– if $x_c + a < x_{w\min}$ or $x_c - a > x_{w\max}$ or $y_c + b < y_{w\min}$ or $y_c - b > y_{w\max}$, then E does not overlap with W. Reject E. Stop. Else goto next step.

Step 2: Trivial Select Case: (Ellipse B) This is when R is inside W. To check this, you need to perform the following test.

if $x_c - a \geq x_{w\min}$ and $x_c + a \leq x_{w\max}$ and $y_c - b \geq y_{w\min}$ and $y_c + b \leq y_{w\max}$, E is completely inside W. Send E to the frame buffer for plotting. Else, goto next step.

Step 3: E may be partially visible (All ellipses except ellipses A, B and C). You need to calculate intersections with boundaries. Process for each window boundary.

Intersection with left boundary: if $x_c - a \leq x_{w\min}$, E may cross the left boundary. In this case, find out the intersection with the left boundary by solving the following pair of equations for y:

$$\frac{(x - x_c)^2}{a^2} + \frac{(y - y_c)^2}{b^2} = 1,\ x = x_{w\min}.$$

Obtain two solutions of this pair of equations.

i) If both the points are inside (Ellipse D), send the portion of the ellipse to the frame buffer which has these two points on the boundary. Remember that here you need to apply a test to find out which portion of the ellipse is inside and which is outside. There are several ways of finding out the selected arc of the ellipse. One could be to find out whether the intersection point (x, y) lies in Region 1 or Region 2. We know that (x, y) is in Region 1 if

$x < \frac{a^2}{b^2} y$.. This will enable you to decide points on the ellipse to be sent to the frame buffer.

ii) If one point is in W (Ellipse E), select this point on the ellipse as first point of intersection. Else the ellipse may be outside the window boundary (Ellipse F, G). Keep the ellipse for the next boundary.

Process in a similar way for other window boundaries. If you get two intersection points of the ellipse with window boundaries, plot your ellipse accordingly. Else, reject it.

Important Note: Make sure before sending the selected segment of the ellipse to the frame buffer, that you are sending the right portion and not the clipped away portion. Using the above steps, write your routine.

E6) All or none character clipping

i) Define the rectangular window boundaries.

ii) Define N. (No. of characters in the given string).

iii) Define width w of each character in the string.

iv) Define the bounding range for each character on the string using its lower left corner. For example, a character 'G' in the string OPENGL is at raster position $(100, 50)$, then its bounding range would be $(100, 50)$ and $(100+\mathrm{w}, 50)$ (see the picture given below).

v) Generate the 4-bit code(above, below, right, left) for the lower left corner $\mathrm{LL(i)}$ of the i^{th} character, $\mathrm{i} = 1, 2, \ldots, \mathrm{N}$.

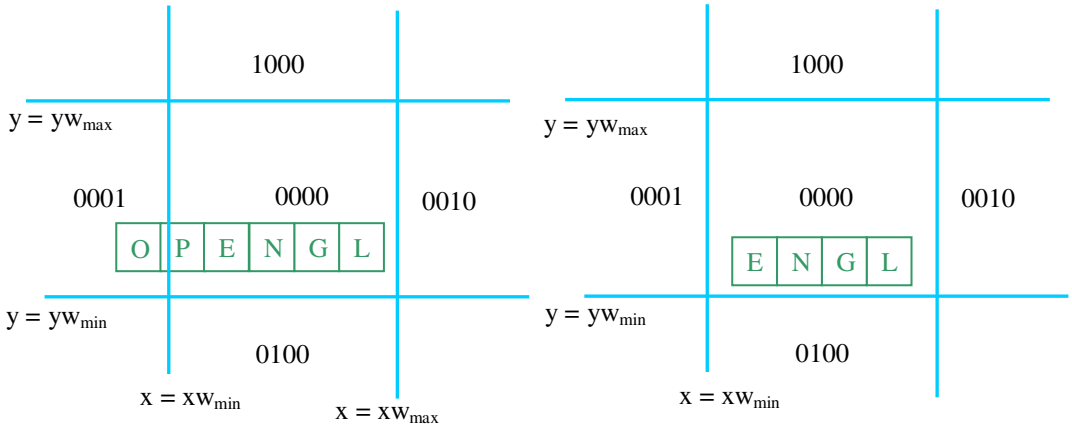
vi) Initialize $\mathrm{i} = 1$;

vii) while(i < = N)

if LL(i) $\neq 0$,

Continue;

else Display Character;



Before Clipping **After Clipping**

E7) $P(t) = (8t^3 - 6t^2 + 18t, -30t^2 + 30t).$

E8) $P(t) = \frac{1}{6}(-2t^3 + 3t^2 + 9t + 13, 17t^3 - 24t^2 - 12t + 28).$