UNIT II  Two Dimensional Graphics

Two dimensional geometric transformations – Matrix representations and homogeneous coordinates, composite transformations; Two dimensional viewing – viewing pipeline, viewing coordinate reference frame; widow-to-viewport coordinate transformation, Two dimensional viewing functions; clipping operations – point, line, and polygon clipping algorithms.

2.1 Two Dimensional Geometric Transformations

- Changes in orientations, size and shape are accomplished with geometric transformations that alter the coordinate description of objects.
- Basic transformation
  1. Translation
  2. Scale
  3. Rotation

2.1.1 Translation

- A translation is applied to an object by representing it along a straight line path from one coordinate location to another adding translation distances, $t_x$, $t_y$ to original coordinate position $(x, y)$ to move the point to a new position $(x', y')$ to
  \[ x' = x + t_x, \quad y' = y + t_y \]
- The translation distance point $(t_x, t_y)$ is called translation vector or shift vector. Translation equation can be expressed as single matrix equation by using column vectors to represent the coordinate position and the translation vector as
  \[
  P = (x, y) \\
  T = (t_x, t_y)
  \]
  \[
  x' = x + t_x \\
  y' = y + t_y
  \]
  \[
  \begin{bmatrix}
  x' \\
  y'
  \end{bmatrix} = \begin{bmatrix}
  x \\
  y
  \end{bmatrix} + \begin{bmatrix}
  t_x \\
  t_y
  \end{bmatrix}
  \]
  \[
  P' = P + T
  \]
Moving a polygon from one position to another position with the translation

2.1.2 Rotations:

- A two-dimensional rotation is applied to an object by repositioning it along a circular path on xy plane.
- To generate a rotation, specify a rotation angle $\theta$ and the position $(x_r, y_r)$ of the rotation point (pivot point) about which the object is to be rotated.
- Positive values for the rotation angle define counter clock wise rotation about pivot point. Negative value of angle rotate objects in clock wise direction.
- The transformation can also be described as a rotation about a rotation axis perpendicular to xy plane and passes through pivot point.

Rotation of a point from position $(x, y)$ to position $(x', y')$ through angle $\theta$ relative to coordinate origin

- The transformation equations for rotation of a point position $P$ when the pivot point is at coordinate origin. In figure $r$ is constant distance of the point positions $\Phi$ is the original angular of the point from horizontal and $\theta$ is the rotation angle.
- The transformed coordinates in terms of angle $\theta$ and $\Phi$:

  \[ x' = r\cos(\theta + \Phi) = r\cos\theta \cos\Phi - r\sin\theta\sin\Phi \]
  \[ y' = r\sin(\theta + \Phi) = r\sin\theta \cos\Phi + r\cos\theta\sin\Phi \]
- The original coordinates of the point in polar coordinates
  \[ x = r \cos \Phi, \quad y = r \sin \Phi \]

- The transformation equation for rotating a point at position \((x,y)\) through an angle \(\theta\) about origin
  \[
  x' = x \cos \theta - y \sin \theta \\
  y' = x \sin \theta + y \cos \theta
  \]

**Rotation equation**

\[ P' = R \cdot P \]

**Rotation Matrix**

\[
R = \begin{bmatrix}
\cos(\theta) & -\sin(\theta) \\
\sin(\theta) & \cos(\theta)
\end{bmatrix}
\]

\[
\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix}
\cos(\theta) & -\sin(\theta) \\
\sin(\theta) & \cos(\theta)
\end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}
\]

**Note:** Positive values for the rotation angle define counter-clockwise rotations about the rotation point and negative values rotate objects in the clockwise.

### 2.1.3 Scaling

- A scaling transformation alters the size of an object. This operation can be carried out for polygons by multiplying the coordinate values \((x,y)\) to each vertex by scaling factor \(S_x\) & \(S_y\) to produce the transformed coordinates \((x',y')\)
  \[
  x' = x \cdot S_x \\
  y' = y \cdot S_y
  \]

- Scaling factor \(S_x\) scales object in \(x\) direction while \(S_y\) scales in \(y\) direction. The transformation equation in matrix form
  \[
  \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}
  \]

  \[
  \text{or} \\
  P' = S \cdot P
  \]

  Where \(S\) is 2 by 2 scaling matrix
Turning a square (a) Into a rectangle (b) with scaling factors \( sx = 2 \) and \( sy = 1 \).

- Any positive numeric values are valid for scaling factors \( sx \) and \( sy \). Values less than 1 reduce the size of the objects and values greater than 1 produce an enlarged object.
- There are two types of Scaling. They are
  1. Uniform scaling
  2. Non Uniform Scaling
- To get uniform scaling it is necessary to assign same value for \( sx \) and \( sy \). Unequal values for \( sx \) and \( sy \) result in a non-uniform scaling.

2.2 Matrix Representation and homogeneous Coordinates

- Many graphics applications involve sequences of geometric transformations. An animation, for example, might require an object to be translated and rotated at each increment of the motion.
- In order to combine sequence of transformations we have to eliminate the matrix addition. To achieve this we have represent matrix as \( 3 \times 3 \) instead of \( 2 \times 2 \) introducing an additional dummy coordinate \( h \). Here points are specified by three numbers instead of two.
- This coordinate system is called as **Homogeneous coordinate system** and it allows to express transformation equation as matrix multiplication.
- Cartesian coordinate position \((x,y)\) is represented as homogeneous coordinate triple \((x,y,h)\)
  - Represent coordinates as \((x,y,h)\)
  - Actual coordinates drawn will be \((x/h,y/h)\)

**For Translation**

\[
\begin{bmatrix}
  x' \\
  y'
\end{bmatrix} = \begin{bmatrix}
  1 & 0 & t_x \\
  0 & 1 & t_y \\
  0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
  x \\
  y
\end{bmatrix}
\]

\[P' = T(t_x,t_y) \cdot P\]
For Scaling

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

\[
P' = S(s_x, s_y) \cdot P
\]

For rotation

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

\[
P' = R(\theta) \cdot P
\]

2.2.1 Composite Transformations

- A composite transformation is a sequence of transformations; one followed by the other. We can set up a matrix for any sequence of transformations as a composite transformation matrix by calculating the matrix product of the individual transformations.

Translation

- If two successive translation vectors \((tx_1, ty_1)\) and \((tx_2, ty_2)\) are applied to a coordinate position \(P\), the final transformed location \(P'\) is calculated as:

\[
P' = T(tx_2, ty_2).\{T(tx_1, ty_1).P\}
\]

\[
= \{T(tx_2, ty_2).T(tx_1, ty_1).P\}
\]

Where \(P\) and \(P'\) are represented as homogeneous-coordinate column vectors.

\[
\begin{bmatrix}
1 & 0 & tx_2 \\
0 & 1 & ty_2 \\
0 & 0 & 1
\end{bmatrix} \cdot \begin{bmatrix}
1 & 0 & tx_1 \\
0 & 1 & ty_1 \\
0 & 0 & 1
\end{bmatrix} = \begin{bmatrix}
1 & 0 & tx_1 + tx_2 \\
0 & 1 & ty_1 + ty_2 \\
0 & 0 & 1
\end{bmatrix}
\]

Or

\[
T(tx_2, ty_2).T(tx_1, ty_1) = T(tx_1 + tx_2, ty_1 + ty_2)
\]

Which demonstrated the two successive translations are additive.
Rotations

- Two successive rotations applied to point P produce the transformed position

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

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

\[ R(\theta_2).R(\theta_1) = R(\theta_1 + \theta_2) \]

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

\[
\begin{bmatrix}
\cos \theta_2 & -\sin \theta_2 & 0 \\
\sin \theta_2 & \cos \theta_2 & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\cos \theta_1 & -\sin \theta_1 & 0 \\
\sin \theta_1 & \cos \theta_1 & 0 \\
0 & 0 & 1
\end{bmatrix}
= 
\begin{bmatrix}
\cos(\theta_2 + \theta_1) & -\sin(\theta_2 + \theta_1) & 0 \\
\sin(\theta_2 + \theta_1) & \cos(\theta_2 + \theta_1) & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

Scaling

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

\[
\begin{bmatrix}
sx_2 & 0 & 0 \\
0 & sy_2 & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
sx_1 & 0 & 0 \\
0 & sy_1 & 0 \\
0 & 0 & 1
\end{bmatrix}
= 
\begin{bmatrix}
sx_2.sx_1 & 0 & 0 \\
0 & sy_2.sy_1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]
General Pivot-point Rotation

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

The composite transformation matrix for this sequence is obtained with the concatenation

\[
\begin{bmatrix}
1 & 0 & x_r \\
0 & 1 & y_r \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
cos \theta & -sin \theta & 0 \\
sin \theta & cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & -x_r \\
0 & 1 & -y_r \\
0 & 0 & 1
\end{bmatrix}
\]

Which can also be expressed as:

\[
T(x_r, y_r). R(\theta). T(-x_r, -y_r) = R(x_r, y_r, \theta)
\]

\[
\begin{bmatrix}
cos \theta & -sin \theta & x_r(1- \cos \theta) + y_r \sin \theta \\
sin \theta & cos \theta & y_r(1- \cos \theta) - x_r \sin \theta \\
0 & 0 & 1
\end{bmatrix}
\]
General fixed point scaling

Translate object so that the fixed point coincides with the coordinate origin

Scale the object with respect to the coordinate origin

Use the inverse translation of step 1 to return the object to its original position

Concatenating the matrices for these three operations produces the required scaling matrix

\[
\begin{bmatrix}
1 & 0 & xf \\
0 & 1 & yf \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
sx & 0 & 0 \\
0 & sy & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & -xf \\
0 & 1 & -yf \\
0 & 0 & 1
\end{bmatrix}
\]

\[
= \begin{bmatrix}
sx & 0 & xf(1- sx) \\
0 & sy & yf(1- sy) \\
0 & 0 & 1
\end{bmatrix}
\]

Can also be expressed as \( T(xf, yf).S(sx, sy).T(-xf, -yf) = S(xf, yf, sx, sy) \)

Note: Transformations can be combined by matrix multiplication
Implementation of composite transformations

#include <math.h>
#include <graphics.h>
typedef float Matrix3x3 [3][3];
Matrix3x3 thematrix;

void matrix3x3SetIdentity (Matrix3x3 m)
{
    int i, j;
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            m[i][j] = (i == j);
}

void matrix3x3PreMultiply (Matrix3x3 a, Matrix3x3 b)
{
    int r, c;
    Matrix3x3 tmp;
    for (r = 0; r < 3; r++)
        for (c = 0; c < 3; c++)
            tmp[r][c] = a[r][0]*b[0][c] + a[r][1]*b[1][c] + a[r][2]*b[2][c];
    for (r = 0; r < 3; r++)
        for (c = 0; c < 3; c++)
            b[r][c] = tmp[r][c];
}

void translate2 (int tx, int ty)
{
    Matrix3x3 m;
    matrix3x3SetIdentity (m);

    \[
    \begin{bmatrix}
    x' \\
    y' \\
    w'
    \end{bmatrix} =
    \begin{bmatrix}
    1 & 0 & tx \\
    0 & 1 & ty \\
    0 & 0 & 1
    \end{bmatrix}
    \begin{bmatrix}
    \cos \Theta & -\sin \Theta & 0 \\
    \sin \Theta & \cos \Theta & 0 \\
    0 & 0 & 1
    \end{bmatrix}
    \begin{bmatrix}
    x \\
    y \\
    w
    \end{bmatrix}
    \]
\[ m[0][2] = tx; \]
\[ m[1][2] = ty; \]
\[ \text{matrix3x3PreMultiply}(m, \text{theMatrix}); \]

\[
\text{void scale2 (float sx, float sy, wcPt2 refPt)} \]
\[
\{ \]
\[
\text{Matrix3x3 m; } \]
\[
\text{matrix3x3SetIdentity}(m); \]
\[
m[0][0] = sx; \]
\[
m[0][2] = (1 - sx) \times \text{refPt.x}; \]
\[
m[1][1] = sy; \]
\[
m[1][2] = (1 - sy) \times \text{refPt.y}; \]
\[
\text{matrix3x3PreMultiply}(m, \text{theMatrix}); \]
\[
\}\]

\[
\text{void rotate2 (float a, wcPt2 refPt)} \]
\[
\{ \]
\[
\text{Matrix3x3 m;} \]
\[
\text{matrix3x3SetIdentity}(m); \]
\[
a = \text{pToRadians}(a); \]
\[
m[0][0] = \text{cosf}(a); \]
\[
m[0][1] = -\text{sinf}(a); \]
\[
m[0][2] = \text{refPt.x} \times (1 - \text{cosf}(a)) + \text{refPt.y} \times \text{sinf}(a); \]
\[
m[1][0] = \text{sinf}(a); \]
\[
m[1][1] = \text{cosf}(a); \]
\[
m[1][2] = \text{refPt.y} \times (1 - \text{cosf}(a)) - \text{refPt.x} \times \text{sinf}(a); \]
\[
\text{matrix3x3PreMultiply}(m, \text{theMatrix}); \]
\[
\}\]

\[
\text{void transformPoints2 (int npts, wcPt2 *pts)} \]
\[
\{ \]
\[
\text{int k; } \]
\[
\text{float tmp; } \]
for (k = 0; k< npts: k++)
{
    tmp = theMatrix[0][0]* pts[k].x * theMatrix[0][1] * pts[k].y+ theMatrix[0][2];
    pts[k].y = theMatrix[1][0]* pts[k] .x * theMatrix[1][1] * pts[k].y+ theMatrix[1][2];
    pts[k].x = tmp;
}

void main (int argc, char **argv)
{
    wcPt2 pts[3] = { 50.0, 50.0, 150.0, 50.0, 100.0, 150.0};
    wcPt2 refPt = {100.0, 100.0};
    long windowID = openGraphics (*argv, 200, 350);
    setbackground (WHITE) ;
    setcolor (BLUE);
    pFillArea(3, pts);
    matrix3x3SetIdentity (theMatrix);
    scale2 (0.5, 0.5, refPt);
    rotate2 (90.0, refPt);
    translate2 (0, 150);
    transformpoints2 ( 3 , pts)
    pFillArea(3, pts);
    sleep (10);
    closeGraphics (windowID);
}

Other Transformation: Reflection

- A reflection is a transformation that produces a mirror image of an object. The mirror image for a two-dimensional reflection is generated relative to an axis of reflection by rotating the object 180° about the reflection axis. We can choose an axis of reflection in the xy plane or perpendicular to the xy plane or coordinate origin
Reflection of an object about the x axis

Reflection of an object about the x axis is accomplished with the transformation matrix

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]

Reflection of an object about the y axis

Reflection of an object about the y axis is accomplished with the transformation matrix

\[
\begin{bmatrix}
-1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]

Reflection of an object about the coordinate origin
Reflection about origin is accomplished with the transformation matrix

\[
\begin{bmatrix}
-1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

Reflection axis as the diagonal line \( y = x \)

- To obtain transformation matrix for reflection about diagonal \( y=x \) the transformation sequence is
  1. Clock wise rotation by \( 45^0 \)
  2. Reflection about x axis
  3. counter clock wise by \( 45^0 \)

Reflection about the diagonal line \( y=\) is accomplished with the transformation matrix

\[
\begin{bmatrix}
0 & 1 & 0 \\
1 & 0 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

Reflection axis as the diagonal line \( y = -x \)
To obtain transformation matrix for reflection about diagonal $y=-x$ the transformation sequence is

1. Clock wise rotation by $45^0$
2. Reflection about y axis
3. counter clock wise by $45^0$

Reflection about the diagonal line $y=-x$ is accomplished with the transformation matrix

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

Other Transformation: Shear

- A Transformation that slants the shape of an object is called the shear transformation. Two common shearing transformations are used.
- One shifts x coordinate values and other shift y coordinate values. However in both the cases only one coordinate (x or y) changes its coordinates and other preserves its values.

X- Shear

- The x shear preserves the y coordinates, but changes the x values which cause vertical lines to tilt right or left as shown in figure

![Diagram of X-Shear](image)

- The Transformations matrix for x-shear is

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

which transforms the coordinates as

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

$$
y' = y$$
Y Shear

- The y shear preserves the x coordinates, but changes the y values which cause horizontal lines which slope up or down. The Transformations matrix for y-shear is

\[
\begin{bmatrix}
1 & 0 & 0 \\
sh_y & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

which transforms the coordinates as

\[
x' = x \\
y' = y + sh_y \cdot x
\]

XY-Shear

- The transformation matrix for xy-shear

\[
\begin{bmatrix}
x' \\
y' \\
1
\end{bmatrix} =
\begin{bmatrix}
1 & sh_x & 0 \\
sh_y & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
1
\end{bmatrix}
\]

which transforms the coordinates as

\[
x' = x + sh_x \cdot y \\
y' = y + sh_y \cdot x
\]

Shearing Relative to other reference line

- We can apply x shear and y shear transformations relative to other reference lines. In x shear transformations we can use y reference line and in y shear we can use x reference line.

X shear with y reference line

- We can generate x-direction shears relative to other reference lines with the transformation matrix

\[
\begin{bmatrix}
1 & sh_x & -sh_y \cdot y_{ref} \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

which transforms the coordinates as

\[
x' = x + sh_x \cdot (y - y_{ref}) \\
y' = y
\]
Example

\[ \text{Sh}_x = \frac{1}{2} \quad \text{and} \quad \text{y}_{\text{ref}} = -1 \]

\[ \text{Y shear with x reference line} \]

We can generate y-direction shears relative to other reference lines with the transformation matrix

\[
\begin{bmatrix}
1 & sh_x & -sh_x \cdot y_{\text{ref}} \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

which transforms the coordinates as

\[ x' = x \]

\[ y' = \text{sh}_y (x - x_{\text{ref}}) + y \]

Example

\[ \text{Sh}_y = \frac{1}{2} \quad \text{and} \quad x_{\text{ref}} = -1 \]
2.3 Two dimensional viewing

The viewing pipeline

- A world coordinate area selected for display is called a window. An area on a display device to which a window is mapped is called a view port. The window defines what is to be viewed, the view port defines where it is to be displayed.
- The mapping of a part of a world coordinate scene to device coordinate is referred to as viewing transformation.
- The two dimensional viewing transformation is referred to as window to view port transformation of windowing transformation.

A viewing transformation using standard rectangles for the window and viewport

The two dimensional viewing transformation pipeline

- The viewing transformation in several steps, as indicated in Fig. First, we construct the scene in world coordinates using the output primitives. Next to obtain a particular orientation for the window, we can set up a two-dimensional viewing-coordinate system in the world coordinate plane, and define a window in the viewing-coordinate system.
- The viewing-coordinate reference frame is used to provide a method for setting up arbitrary orientations for rectangular windows. Once the viewing reference frame is established, we can transform descriptions in world coordinates to viewing coordinates.
- We then define a viewport in normalized coordinates (in the range from 0 to 1) and map the viewing-coordinate description of the scene to normalized coordinates.
- At the final step all parts of the picture that lie outside the viewport are clipped, and the contents of the viewport are transferred to device coordinates. By
changing the position of the viewport, we can view objects at different positions on the display area of an output device.

**Window to view port coordinate transformation:**

A point \((x_w, y_w)\) in a world-coordinate clipping window is mapped to viewport coordinates \((x_v, y_v)\), within a unit square, so that the relative positions of the two points in their respective rectangles are the same.

- A point at position \((x_w, y_w)\) in a designated window is mapped to viewport coordinates \((x_v, y_v)\) so that relative positions in the two areas are the same. The figure illustrates the window to view port mapping.
- A point at position \((x_w, y_w)\) in the window is mapped into position \((x_v, y_v)\) in the associated view port. To maintain the same relative placement in view port as in window

\[
\frac{x_v - x_v_{min}}{x_v_{max} - x_v_{min}} = \frac{x_w - x_w_{min}}{x_w_{max} - x_w_{min}}
\]
\[
\frac{y_v - y_v_{min}}{y_v_{max} - y_v_{min}} = \frac{y_w - y_w_{min}}{y_w_{max} - y_w_{min}}
\]

solving these expressions for view port position \((x_v, y_v)\)
The conversion is performed with the following sequence of transformations.

1. Perform a scaling transformation using point position of \((x_w \text{ min, } y_w \text{ min})\) that scales the window area to the size of view port.

2. Translate the scaled window area to the position of view port. Relative proportions of objects are maintained if scaling factor are the same (Sx=Sy).

- Otherwise world objects will be stretched or contracted in either the x or y direction when displayed on output device. For normalized coordinates, object descriptions are mapped to various display devices.

- Any number of output devices can be open in particular application and another window view port transformation can be performed for each open output device.

- This mapping called the work station transformation is accomplished by selecting a window area in normalized apace and a view port are in coordinates of display device.

Mapping selected parts of a scene in normalized coordinate to different video monitors with work station transformation.

Two Dimensional viewing functions

- Viewing reference system in a PHIGS application program has following function.

\[
\begin{align*}
    x_v &= x_v \text{ min} + (x_w - x_w \text{ min}) \frac{x_v \text{ max} - x_v \text{ min}}{x_w \text{ max} - x_w \text{ min}} \\
    y_v &= y_v \text{ min} + (y_w - y_w \text{ min}) \frac{y_v \text{ max} - y_v \text{ min}}{y_w \text{ max} - y_w \text{ min}}
\end{align*}
\]

where scaling factors are

\[
\begin{align*}
    S_x &= x_v \text{ max} - x_v \text{ min} \\
    S_y &= y_v \text{ max} - y_v \text{ min} \\
    X_w &= x_w \text{ max} - x_w \text{ min} \\
    Y_w &= y_w \text{ max} - y_w \text{ min}
\end{align*}
\]

- An integer error code is generated if the input parameters are in error otherwise the view matrix for world-to-viewing transformation is calculated. Any number of viewing transformation matrices can be defined in an application.
- To set up elements of window to view port mapping

  \textbf{evaluateViewMappingMatrix} \ \( (x_{w\text{min}}, x_{w\text{max}}, y_{w\text{min}}, y_{w\text{max}}, x_{v\text{min}}, x_{v\text{max}}, y_{v\text{min}}, y_{v\text{max}}, \text{error}, \text{viewMappingMatrix}) \)

- Here window limits in viewing coordinates are chosen with parameters \( x_{w\text{min}}, x_{w\text{max}}, y_{w\text{min}}, y_{w\text{max}} \) and the viewport limits are set with normalized coordinate positions \( x_{v\text{min}}, x_{v\text{max}}, y_{v\text{min}}, y_{v\text{max}} \).

- The combinations of viewing and window view port mapping for various workstations in a viewing table with

  \textbf{setViewRepresentation} \ \( (ws, \text{viewIndex}, \text{viewMatrix}, \text{viewMappingMatrix}, x_{\text{clipmin}}, x_{\text{clipmax}}, y_{\text{clipmin}}, y_{\text{clipmax}}, \text{clipxy}) \)

Where

- \( ws \) designates the output device

- \( \text{viewIndex} \) sets an integer identifier for this window-view port point.

- \( \text{viewMatrix} \) and \( \text{viewMappingMatrix} \) can be referenced by \( \text{viewIndex} \).
setViewIndex(viewIndex)

selects a particular set of options from the viewing table.

At the final stage we apply a workstation transformation by selecting a workstation window viewport pair.

setWorkstationWindow (ws, xwsWindmin, xwsWindmax,
                         ywsWindmin, ywsWindmax)

setWorkstationViewport (ws, xwsVPortmin, xwsVPortmax,
                          ywsVPortmin, ywsVPortmax)

where was gives the workstation number. Window-coordinate extents are specified in the range from 0 to 1 and viewport limits are in integer device coordinates.

2.4 Clipping operation

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

- Algorithm for clipping primitive types:
  - Point clipping
  - Line clipping (Straight-line segment)
  - Area clipping
  - Curve clipping
  - Text clipping

- Line and polygon clipping routines are standard components of graphics packages.

Point Clipping

- Clip window is a rectangle in standard position. A point P=\( (x,y) \) for display, if following inequalities are satisfied:

  \[
  x_{W_{\text{min}}} <= x <= x_{W_{\text{max}}}
  \]

  \[
  y_{W_{\text{min}}} <= y <= y_{W_{\text{max}}}
  \]

  where the edges of the clip window \( (x_{W_{\text{min}}},x_{W_{\text{max}}},y_{W_{\text{min}}},y_{W_{\text{max}}}) \) can be either the world-coordinate window boundaries or viewport boundaries.

- If any one of these four inequalities is not satisfied, the point is clipped.
Line Clipping

- A line clipping procedure involves several parts. First we test a given line segment whether it lies completely inside the clipping window. If it does not we try to determine whether it lies completely outside the window.

- Finally if we cannot identify a line as completely inside or completely outside, we perform intersection calculations with one or more clipping boundaries.

- Process lines through “inside-outside” tests by checking the line endpoints. A line with both endpoints inside all clipping boundaries such as line from P1 to P2 is saved.

- A line with both end point outside any one of the clip boundaries line P3 P4 is outside the window.

Line clipping against a rectangular clip window

- All other lines cross one or more clipping boundaries. For a line segment with end points \((x_1,y_1)\) and \((x_2,y_2)\) one or both end points outside clipping rectangle, the parametric representation

\[
\begin{align*}
x &= x_1 + u(x_2 - x_1), \\
y &= y_1 + u(y_2 - y_1),
\end{align*}
\]

\[0 \leq u \leq 1\]

could be used to determine values of \(u\) for an intersection with the clipping boundary coordinates. If the value of \(u\) for an intersection with a rectangle boundary edge is outside the range of 0 to 1, the line does not enter the interior of the window at that boundary.

- If the value of \(u\) is within the range from 0 to 1, the line segment does indeed cross into the clipping area. This method can be applied to each clipping boundary edge in to determined whether any part of line segment is to displayed.

2.4.1 Cohen-Sutherland Line Clipping

- This is one of the oldest and most popular line-clipping procedures. The method speeds up the processing of line segments by performing initial tests that reduce the number of intersections that must be calculated.
• Every line endpoint in a picture is assigned a four digit binary code called a region code that identifies the location of the point relative to the boundaries of the clipping rectangle.

![Binary region codes assigned to line end points according to relative position with respect to the clipping rectangle.](image)

• Regions are set up in reference to the boundaries. Each bit position in region code is used to indicate one of four relative coordinate positions of points with respect to clip window: to the left, right, top or bottom.

• By numbering the bit positions in the region code as 1 through 4 from right to left, the coordinate regions are corrected with bit positions as

  bit 1: left
  bit 2: right
  bit 3: below
  bit 4: above

• A value of 1 in any bit position indicates that the point is in that relative position. Otherwise the bit position is set to 0. If a point is within the clipping rectangle the region code is 0000. A point that is below and to the left of the rectangle has a region code of 0101.

• Bit values in the region code are determined by comparing endpoint coordinate values (x,y) to clip boundaries. Bit1 is set to 1 if x < x_{min}.

• For programming language in which bit manipulation is possible region-code bit values can be determined with following two steps.

  (1) Calculate differences between endpoint coordinates and clipping boundaries.
  (2) Use the resultant sign bit of each difference calculation to set the corresponding value in the region code.

     bit 1 is the sign bit of x – x_{min}
     bit 2 is the sign bit of x_{max} - x
bit 3 is the sign bit of \( y - y_{\text{min}} \)

bit 4 is the sign bit of \( y_{\text{max}} - y \).

- Once we have established region codes for all line endpoints, we can quickly determine which lines are completely inside the clip window and which are clearly outside.

- Any lines that are completely contained within the window boundaries have a region code of 0000 for both endpoints, and we accept these lines. Any lines that have a 1 in the same bit position in the region codes for each endpoint are completely outside the clipping rectangle, and we reject these lines.

- We would discard the line that has a region code of 1001 for one endpoint and a code of 0101 for the other endpoint. Both endpoints of this line are left of the clipping rectangle, as indicated by the 1 in the first bit position of each region code.

- A method that can be used to test lines for total clipping is to perform the logical and operation with both region codes. If the result is not 0000, the line is completely outside the clipping region.

- Lines that cannot be identified as completely inside or completely outside a clip window by these tests are checked for intersection with window boundaries.

- \textbf{Line extending from one coordinates region to another may pass through the clip window, or they may intersect clipping boundaries without entering window.}

  - Cohen-Sutherland line clipping starting with bottom endpoint left, right, bottom and top boundaries in turn and find that this point is below the clipping rectangle.

  - Starting with the bottom endpoint of the line from \( P_1 \) to \( P_2 \), we check \( P_1 \) against the left, right, and bottom boundaries in turn and find that this point is below the clipping rectangle. We then find the intersection point \( P_1' \) with the bottom boundary and discard the line section from \( P_1 \) to \( P_1' \).

  - The line now has been reduced to the section from \( P_1' \) to \( P_2 \). Since \( P_2 \) is outside the clip window, we check this endpoint against the boundaries and find that it is to the left of the window. Intersection point \( P_2' \) is calculated, but this point is above
the window. So the final intersection calculation yields $P_2''$, and the line from $P_1'$ to $P_2''$ is saved. This completes processing for this line, so we save this part and go on to the next line.

- Point $P_3$ in the next line is to the left of the clipping rectangle, so we determine the intersection $P_3'$, and eliminate the line section from $P_3$ to $P_3'$. By checking region codes for the line section from $P_3$ to $P_4$ we find that the remainder of the line is below the clip window and can be discarded also.

- Intersection points with a clipping boundary can be calculated using the slope-intercept form of the line equation. For a line with endpoint coordinates $(x_1, y_1)$ and $(x_2, y_2)$ and the $y$ coordinate of the intersection point with a vertical boundary can be obtained with the calculation.

$$y = y_1 + m(x - x_1)$$

where $x$ value is set either to $x_{w_{\text{min}}}$ or to $x_{w_{\text{max}}}$ and

slope of line is calculated as

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

the intersection with a horizontal boundary the $x$ coordinate can be calculated as

$$x = x_1 + \left(\frac{y - y_1}{m}\right)$$

with $y$ set to either to $y_{w_{\text{min}}}$ or to $y_{w_{\text{max}}}$.

**Implementation of Cohen-sutherland Line Clipping**

```c
#define Round(a)      ((int)(a+0.5))
#define LEFT_EDGE    0x1
#define RIGHT_EDGE   0x2
#define BOTTOM_EDGE  0x4
#define TOP_EDGE     0x8
#define TRUE         1
#define FALSE        0
#define INSIDE(a)    (!a)
#define REJECT(a,b)  (a&b)
#define ACCEPT(a,b)   (!(a|b))
unsigned char encode(wcPt2 pt, dcPt winmin, dcPt winmax)
{
```
unsigned char code=0x00;
if(pt.x<winmin.x)
    code=code|LEFT_EDGE;
if(pt.x>winmax.x)
    code=code|RIGHT_EDGE;
if(pt.y<winmin.y)
    code=code|BOTTOM_EDGE;
if(pt.y>winmax.y)
    code=code|TOP_EDGE;
return(code);
}
void swappts(wcPt2 *p1,wcPt2 *p2)
{
    wcPt2 temp;
    tmp=*p1;
    *p1=*p2;
    *p2=tmp;
}
void swapcodes(unsigned char *c1,unsigned char *c2)
{
    unsigned char tmp;
    tmp=*c1;
    *c1=*c2;
    *c2=tmp;
}
void clipline(dcPt winmin, dcPt winmax, wcPt2 p1,ecPt2 point p2)
{
    unsigned char code1,code2;
    int done=FALSE, draw=FALSE;
    float m;
    while(!done)
{ 
    code1=encode(p1,winmin,winmax);
    code2=encode(p2,winmin,winmax);
    if(ACCEPT(code1,code2))
    {
        done=TRUE;
        draw=TRUE;
    } 
    else if(REJECT(code1,code2))
        done=TRUE;
    else
    {
        if(INSIDE(code1))
        {
            swappts(&p1,&p2);
            swapcodes(&code1,&code2);
        }
        if(p2.x!=p1.x)
            m=(p2.y-p1.y)/(p2.x-p1.x);
        if(code1 &LEFT_EDGE)
        {
            p1.y+=(winmin.x-p1.x)*m;
            p1.x=winmin.x;
        }
        else if(code1 &RIGHT_EDGE)
        {
            p1.y+=(winmax.x-p1.x)*m;
            p1.x=winmax.x;
        }
        else if(code1 &BOTTOM_EDGE)
        {
            
        }
if(p2.x!=p1.x)
p1.x+=(winmin.y-p1.y)/m;
p1.y=winmin.y;
}
else if(code1 &TOP_EDGE)
{
if(p2.x!=p1.x)
p1.x+=(winmax.y-p1.y)/m;
p1.y=winmax.y;
}
}
}
if(draw)
lineDDA(ROUND(p1.x),ROUND(p1.y),ROUND(p2.x),ROUND(p2.y));
}

2.4.2 Liang – Barsky line Clipping:

- Based on analysis of parametric equation of a line segment, faster line clippers have been developed, which can be written in the form :

\[
\begin{align*}
x &= x_1 + u \Delta x \\
y &= y_1 + u \Delta y \\
0 &\leq u \leq 1
\end{align*}
\]

where \( \Delta x = (x_2 - x_1) \) and \( \Delta y = (y_2 - y_1) \)

- In the Liang-Barsky approach we first the point clipping condition in parametric form :

\[
\begin{align*}
xw_{min} &\leq x_1 + u \Delta x \leq xw_{max} \\
yw_{min} &\leq y_1 + u \Delta y \leq yw_{max}
\end{align*}
\]

Each of these four inequalities can be expressed as

\[ \mu p_k \leq q_k, \quad k=1,2,3,4 \]

the parameters \( p \) & \( q \) are defined as

\[
\begin{align*}
p_1 &= -\Delta x \\
q_1 &= x_1 - xw_{min} \\
p_2 &= \Delta x \\
q_2 &= xw_{max} - x_1
\end{align*}
\]
\[ P_3 = -\Delta y \quad \text{and} \quad q_3 = y_1 - y_{W_{\text{min}}} \]
\[ P_4 = \Delta y \quad \text{and} \quad q_4 = y_{W_{\text{max}}} - y_1 \]

- Any line that is parallel to one of the clipping boundaries have \( p_k = 0 \) for values of \( k \) corresponding to boundary \( k = 1, 2, 3, 4 \) correspond to left, right, bottom and top boundaries. For values of \( k \), find \( q_k < 0 \), the line is completely outside the boundary.

  If \( q_k = 0 \), the line is inside the parallel clipping boundary.

  When \( p_k < 0 \) the infinite extension of line proceeds from outside to inside of the infinite extension of this clipping boundary.

  If \( p_k > 0 \), the line proceeds from inside to outside, for non-zero value of \( p_k \) calculate the value of \( u \), that corresponds to the point where the infinitely extended line intersect the extension of boundary \( k \) as

\[ u = q_k / p_k \]

For each line, calculate values for parameters \( u_1 \) and \( u_2 \) that define the part of line that lies within the clip rectangle.

The value of \( u_1 \) is determined by looking at the rectangle edges for which the line proceeds from outside to the inside (\( p < 0 \)).

For these edges we calculate

\[ r_k = q_k / p_k \]

The value of \( u_1 \) is taken as largest of set consisting of 0 and various values of \( r \).

The value of \( u_2 \) is determined by examining the boundaries for which lines proceed from inside to outside (\( p > 0 \)).

A value of \( r_k \) is calculated for each of these boundaries and value of \( u_2 \) is the minimum of the set consisting of 1 and the calculated \( r \) values.

If \( u_1 > u_2 \), the line is completely outside the clip window and it can be rejected.

Line intersection parameters are initialized to values \( u_1 = 0 \) and \( u_2 = 1 \). For each clipping boundary, the appropriate values for \( P \) and \( q \) are calculated and used by function

- Cliptest to determine whether the line can be rejected or whether the intersection parameter can be adjusted.

  When \( p < 0 \), the parameter \( r \) is used to update \( u_1 \).

  When \( p > 0 \), the parameter \( r \) is used to update \( u_2 \).

  If updating \( u_1 \) or \( u_2 \) results in \( u_1 > u_2 \) reject the line, when \( p = 0 \) and \( q < 0 \), discard the line, it is parallel to and outside the boundary. If the line has not been rejected after all four value of \( p \) and \( q \) have been tested, the end points of clipped lines are determined from values of \( u_1 \) and \( u_2 \).
• The Liang-Barsky algorithm is more efficient than the Cohen-Sutherland algorithm since intersections calculations are reduced. Each update of parameters $u_1$ and $u_2$ require only one division and window intersections of these lines are computed only once.

• Cohen-Sutherland algorithm, can repeatedly calculate intersections along a line path, even through line may be completely outside the clip window. Each intersection calculations require both a division and a multiplication.

**Implementation of Liang-Barsky Line Clipping**

```c
#define Round(a)   ((int)(a+0.5))
int clipTest (float p, float q, gfloat *u1, float *u2)
{
    float r;
    int retval=TRUE;
    if (p<0.0)
    {
        r=q/p
        if (r>*u2)
            retval=FALSE;
        else
            if (r>*u1)
                *u1=r;
    }
    else
    if (p>0.0)
    {
        r=q/p
        if (r<*u1)
            retval=FALSE;
        else
            if (r<*u2)
                *u2=r;
```
else
if (q<0.0)
    retVal=FALSE
return(retVal);
}
void clipLine (dcPt winMin, dcPt winMax, wcPt2 p1, wcpt2 p2)
{
    float u1=0.0, u2=1.0, dx=p2.x-p1.x,dy;
    if (clipTest (-dx, p1.x-winMin.x, &u1, &u2))
if (clipTest (dx, winMax.x-p1.x, &u1, &u2))
    {
    dy=p2.y-p1.y;
    if (clipTest (-dy, p1.y-winMin.y, &u1, &u2))
if (clipTest (dy, winMax.y-p1.y, &u1, &u2))
    {
    if (u1<1.0)
    {
    p2.x=p1.x+u2*dx;
p2.y=p1.y+u2*dy;
    }
    if (u1>0.0)
    {
    p1.x=p1.x+u1*dx;
p1.y=p1.y+u1*dy;
    }
    lineDDA(ROUND(p1.x),ROUND(p1.y),ROUND(p2.x),ROUND(p2.y));
    }
}
2.4.3 Nicholl-Lee-Nicholl Line clipping

- By creating more regions around the clip window, the Nicholl-Lee-Nicholl (or NLN) algorithm avoids multiple clipping of an individual line segment.

- In the Cohen-Sutherland method, multiple intersections may be calculated. These extra intersection calculations are eliminated in the NLN algorithm by carrying out more region testing before intersection positions are calculated.

- Compared to both the Cohen-Sutherland and the Liang-Barsky algorithms, the Nicholl-Lee-Nicholl algorithm performs fewer comparisons and divisions.

- The trade-off is that the NLN algorithm can only be applied to two-dimensional clipping, whereas both the Liang-Barsky and the Cohen-Sutherland methods are easily extended to three-dimensional scenes.

- For a line with endpoints P1 and P2 we first determine the position of point P1, for the nine possible regions relative to the clipping rectangle. Only the three regions shown in Fig. need to be considered.

- If P1 lies in any one of the other six regions, we can move it to one of the three regions in Fig., using a symmetry transformation. For example, the region directly above the clip window can be transformed to the region left of the clip window using a reflection about the line \( y = -x \), or we could use a 90 degree counterclockwise rotation.

Three possible positions for a line endpoint p1(a) in the NLN algorithm

- Case 1: p1 inside region
- Case 2: p1 across edge
- Case 3: p1 across corner
Next, we determine the position of $P_2$ relative to $P_1$. To do this, we create some new regions in the plane, depending on the location of $P_1$. Boundaries of the new regions are half-infinite line segments that start at the position of $P_1$ and pass through the window corners. If $P_1$ is inside the clip window and $P_2$ is outside, we set up the four regions shown in Fig.

The four clipping regions used in NLN alg when $p1$ is inside and $p2$ outside the clip window

- The intersection with the appropriate window boundary is then carried out, depending on which one of the four regions ($L$, $T$, $R$, or $B$) contains $P_2$. If both $P_1$ and $P_2$ are inside the clipping rectangle, we simply save the entire line.
- If $P_1$ is in the region to the left of the window, we set up the four regions, $L$, $LT$, $LR$, and $LB$, shown in Fig.

- These four regions determine a unique boundary for the line segment. For instance, if $P_2$ is in region $L$, we clip the line at the left boundary and save the line segment from this intersection point to $P_2$. 


But if $P_2$ is in region $LT$, we save the line segment from the left window boundary to the top boundary. If $P_2$ is not in any of the four regions, $L$, $LT$, $LR$, or $LB$, the entire line is clipped.

For the third case, when $P_1$ is to the left and above the clip window, we use the clipping regions in Fig.

The two possible sets of clipping regions used in NLN algorithm when $P_1$ is above and to the left of the clip window

![Clipping Regions](image)

- In this case, we have the two possibilities shown, depending on the position of $P_1$, relative to the top left corner of the window. If $P_2$ is in one of the regions $T$, $L$, $TR$, $TB$, $LR$, or $LB$, this determines a unique clip window edge for the intersection calculations. Otherwise, the entire line is rejected.

- To determine the region in which $P_2$ is located, we compare the slope of the line to the slopes of the boundaries of the clip regions. For example, if $P_1$ is left of the clipping rectangle (Fig. a), then $P_2$, is in region $LT$ if

$$\text{slope}_{P_1P_{TR}} < \text{slope}_{P_1P_{2}} < \text{slope}_{P_1P_{TL}}$$

or

$$\frac{y_T - y_1}{x_R - x_1} < \frac{y_2 - y_1}{x_2 - x_1} < \frac{y_T - y_1}{x_L - x_1}$$

And we clip the entire line if

$$(y_T - y_1)(x_2 - x_1) < (x_L - x_1)(y_2 - y_1)$$

- The coordinate difference and product calculations used in the slope tests are saved and also used in the intersection calculations. From the parametric equations

$$x = x_1 + (x_2 - x_1)u$$

$$y = y_1 + (y_2 - y_1)u$$

an x-intersection position on the left window boundary is $x = x_L$, with

$$u = \frac{(x_L - x_1)}{(x_2 - x_1)}$$

so that the y-intersection position is

$$y = y_1 + y_2 - y_1 \left(\frac{x_L - x_1}{x_2 - x_1}\right)$$
And an intersection position on the top boundary has \( y = y_T \) and \( u = (y_T - y_1) / (y_2 - y_1) \) with

\[
x = x_1 + x_2 - x_1 \left( \frac{y_T - y_1}{y_2 - y_1} \right)
\]

### 2.5 POLYGON CLIPPING

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

**Display of a polygon processed by a line clipping algorithm**

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

### 2.5.1 Sutherland–Hodgeman polygon clipping:

- A polygon can be clipped by processing the polygon boundary as a whole against each window edge. This could be accomplished by processing all polygon vertices against each clip rectangle boundary.
There are four possible cases when processing vertices in sequence around the perimeter of a polygon. As each point of adjacent polygon vertices is passed to a window boundary clipper, make the following tests:

1. If the first vertex is outside the window boundary and second vertex is inside, both the intersection point of the polygon edge with window boundary and second vertex are added to output vertex list.
2. If both input vertices are inside the window boundary, only the second vertex is added to the output vertex list.
3. If first vertex is inside the window boundary and second vertex is outside only the edge intersection with window boundary is added to output vertex list.
4. If both input vertices are outside the window boundary nothing is added to the output list.

**Clipping a polygon against successive window boundaries.**

![Clipping a polygon against successive window boundaries](image)

Successive processing of pairs of polygon vertices against the left window boundary

![Successive processing of pairs of polygon vertices against the left window boundary](image)

**Clipping a polygon against the left boundary of a window, starting with vertex 1.** Primed numbers are used to label the points in the output vertex list for this window boundary.

![Clipping a polygon against the left boundary of a window, starting with vertex 1.](image)
Where

Vertices 1 and 2 are found to be on outside of boundary.

Moving along vertex 3 which is inside, calculate the intersection and save both the intersection point and vertex 3. Vertex 4 and 5 are determined to be inside and are saved.

Vertex 6 is outside so we find and save the intersection point. Using the five saved points we repeat the process for next window boundary.

- Implementing the algorithm as described requires setting up storage for an output list of vertices as a polygon clipped against each window boundary.
- We eliminate the intermediate output vertex lists by simply by clipping individual vertices at each step and passing the clipped vertices on to the next boundary clipper.
- A point is added to the output vertex list only after it has been determined to be inside or on a window boundary by all boundary clippers. Otherwise the point does not continue in the pipeline.

**A polygon overlapping a rectangular clip window**

![Polygon Overlapping Window](image)

Processing the vertices of the polygon in the above fig. through a boundary clipping pipeline. After all vertices are processed through the pipeline, the vertex list is \{v2”, v2’, v3, v3’\}
Implementation of Sutherland-Hodgeman Polygon Clipping

typedef enum { Left, Right, Bottom, Top } Edge;
#define N_EDGE 4
#define TRUE 1
#define FALSE 0
int inside(wcPt2 p, Edge b, dcPt wmin, dcPt wmax)
{
    switch(b)
    {
    case Left: if(p.x<wmin.x) return (FALSE); break;
    case Right: if(p.x>wmax.x) return (FALSE); break;
    case bottom: if(p.y<wmin.y) return (FALSE); break;
    case top: if(p.y>wmax.y) return (FALSE); break;
    }
    return (TRUE);
}
int cross(wcPt2 p1, wcPt2 p2, Edge b, dcPt wmin, dcPt wmax)
{
    if(inside(p1, b, wmin, wmax)==inside(p2, b, wmin, wmax))
        return (FALSE);
    else
        return (TRUE);
}
wcPt2 (wcPt2 p1, wcPt2 p2, int b, dcPt wmin, dcPt wmax )
{
    wcPt2 iPt;
    float m;
    if(p1.x!=p2.x)
        m=(p1.y-p2.y)/(p1.x-p2.x);
    switch(b)
    {
```c
case Left:
    ipt.x=wmin.x;
    ipt.y=p2.y+(wmin.x-p2.x)*m;
    break;

case Right:
    ipt.x=wmax.x;
    ipt.y=p2.y+(wmax.x-p2.x)*m;
    break;

case Bottom:
    ipt.y=wmin.y;
    if(p1.x!=p2.x)
        ipt.x=p2.x+(wmin.y-p2.y)/m;
    else
        ipt.x=p2.x;
    break;

case Top:
    ipt.y=wmax.y;
    if(p1.x!=p2.x)
        ipt.x=p2.x+(wmax.y-p2.y)/m;
    else
        ipt.x=p2.x;
    break;
}
return(ipt);
}

void clippoint(wcPt2 p,Edge b,dcPt wmin,dcPt wmax, wcPt2 *pout,int *cnt, wcPt2 *first[],struct point *s)
{
    wcPt2 ipt;
    if(!first[b])
        first[b]=&p;
    ```
else
    if(cross(p,s[b],b,wmin,wmax))
      
      ipt=intersect(p,s[b],b,wmin,wmax);
      if(b<top)
        clippoint(ipt,b+1,wmin,wmax,pout,cnt,first,s);
      else
        
        
      
  }

s[b]=p;
if(inside(p,b,wmin,wmax))
  if(b<top)
    clippoint(p,b+1,wmin,wmax,pout,cnt,first,s);
  else
    
  
}

void closeclip(dcPt wmin,dcPt wmax, wcPt2  *pout,int *cnt,wcPt2 *first[], wcPt2 *s)
{
  wcPt2 iPt;
  Edge b;
  for(b=left;b<=top;b++)
    
    
  }

  i=intersect(s[b],*first[b],b,wmin,wmax);
if(b<top)
clippoint(i,b+1,wmin,wmax,pout,cnt,first,s);
else
{
    pout[*cnt]=i;
    (*cnt)++;
}
}

int clippolygon(dcPt point wmin,dcPt wmax,int n,wcPt2 *pin, wcPt2 *pout)
{
wcPt2 *first[N_EDGE]={0,0,0,0},s[N_EDGE];
int i,cnt=0;
for(i=0;i<n;i++)
    clippoint(pin[i],left,wmin,wmax,pout,&cnt,first,s);
closeclip(wmin,wmax,pout,&cnt,first,s);
return(cnt);
}

2.5.2 Weiler- Atherton Polygon Clipping

- This clipping procedure was developed as a method for identifying visible surfaces, and so it can be applied with arbitrary polygon-clipping regions.

- The basic idea in this algorithm is that instead of always proceeding around the polygon edges as vertices are processed, we sometimes want to follow the window boundaries.

- Which path we follow depends on the polygon-processing direction and whether the pair of polygon vertices currently being processed represents an outside-to-inside pair or an inside-to-outside pair.

- For clockwise processing of polygon vertices, we use the following rules:
    
    For an outside-to-inside pair of vertices, follow the polygon boundary.
    
    For an inside-to-outside pair of vertices, follow the window boundary in a clockwise direction.
• In the below Fig. the processing direction in the Weiler-Atherton algorithm and the resulting clipped polygon is shown for a rectangular clipping window.

![Weiler-Atherton Algorithm Diagram](image)

• An improvement on the Weiler-Atherton algorithm is the Weiler algorithm, which applies constructive solid geometry ideas to clip an arbitrary polygon against any polygon clipping region.

2.6 Curve Clipping

• Curve-clipping procedures will involve nonlinear equations, and this requires more processing than for objects with linear boundaries.

• The bounding rectangle for a circle or other curved object can be used first to test for overlap with a rectangular clip window.

• If the bounding rectangle for the object is completely inside the window, we save the object. If the rectangle is determined to be completely outside the window, we discard the object. In either case, there is no further computation necessary.

• But if the bounding rectangle test fails, we can look for other computation-saving approaches.

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

• The below figure illustrates circle clipping against a rectangular window. On the first pass, we can clip the bounding rectangle of the object against the bounding rectangle of the clip region.

• If the two regions overlap, we will need to solve the simultaneous line-curve equations to obtain the clipping intersection points.
2.7 Text clipping

- There are several techniques that can be used to provide text clipping in a graphics package. The clipping technique used will depend on the methods used to generate characters and the requirements of a particular application.

- The simplest method for processing character strings relative to a window boundary is to use the **all-or-none string-clipping** strategy.

- If all of the string is inside a clip window, we keep it. Otherwise, the string is discarded. This procedure is implemented by considering a bounding rectangle around the text pattern.

- The boundary positions of the rectangle are then compared to the window boundaries, and the string is rejected if there is any overlap. This method produces the fastest text clipping.

*Text clipping using a bounding rectangle about the entire string*

- An alternative to rejecting an entire character string that overlaps a window boundary is to use the **all-or-none character-clipping** strategy. Here we discard only those characters that are not completely inside the window.

- In this case, the boundary limits of individual characters are compared to the window. Any character that either overlaps or is outside a window boundary is clipped.
Text clipping using a bounding rectangle about individual characters.

- A final method for handling text clipping is to clip the components of individual characters. We now treat characters in much the same way that we treated lines.
- If an individual character overlaps a clip window boundary, we clip off the parts of the character that are outside the window.

Text Clipping performed on the components of individual characters

2.8 Exterior clipping

- Procedure for clipping a picture to the interior of a region by eliminating everything outside the clipping region.
- By these procedures the inside region of the picture is saved. To clip a picture to the exterior of a specified region. The picture parts to be saved are those that are outside the region. This is called as exterior clipping.
- Objects within a window are clipped to interior of window when other higher priority window overlap these objects. The objects are also clipped to the exterior of overlapping windows.
PART-A

1. What is Transformation?
2. Write short notes on active and passive transformations?
3. Define Translation.
4. Define Rotation.
5. Define Scaling and what the types of scaling are
6. Write the matrix representation and Homogeneous coordinates
7. What is Composite Transformation?
8. Define Reflection.
9. Define Shear.
10. Define Window.
11. Define view port.
12. What id Window to view port coordinate transformation
13. Define Clipping.
14. What are the types of Clipping?
15. What is Polygon Clipping?
16. What are the various types of Polygon Clipping?
17. What is the purpose of presentation graphics?
18. What is frame buffer?
19. Define affine transformation
20. What is covering (exterior clipping)

PART-B

1. Explain the following basic two dimensional geometric transformations
   (i)Translations (ii) Rotation
2. Explain the following composite transformations
   (i) Translations (ii) Rotation
3. Explain in detail the Sutherland-Hodgeman clipping algorithm with an example.
4. Write about Cohen-Sutherland line clipping algorithm with an example.
5. Write short notes on clipping operations..
6. Explain in detail about two dimensional viewing
7. Write about Liang-Barsky Line clipping algorithm with an example
8. Write about Nicholl-Lee – Nicholl Line clipping algorithm

9. Explain the following (i) Basic two dimensional scaling (ii) Composite transformation scalings

10. Explain (i) General Pivot point rotation (ii) General Fixed Point Scaling (iii) General Pivot Point Rotation