UNIT I - 2D PRIMITIVES

Output primitives – Line, Circle and Ellipse drawing algorithms - Attributes of output primitives – Two dimensional Geometric transformation - Two dimensional viewing – Line, Polygon, Curve and Text clipping algorithms

Introduction

A picture is completely specified by the set of intensities for the pixel positions in the display. Shapes and colors of the objects can be described internally with pixel arrays into the frame buffer or with the set of the basic geometric – structure such as straight line segments and polygon color areas. To describe structure of basic object is referred to as output primitives.

Each output primitive is specified with input co-ordinate data and other information about the way that objects is to be displayed. Additional output primitives that can be used to constant a picture include circles and other conic sections, quadric surfaces, Spline curves and surfaces, polygon floor areas and character string.

Points and Lines

Point plotting is accomplished by converting a single coordinate position furnished by an application program into appropriate operations for the output device. With a CRT monitor, for example, the electron beam is turned on to illuminate the screen phosphor at the selected location

Line drawing is accomplished by calculating intermediate positions along the line path between two specified end points positions. An output device is then directed to fill in these positions between the end points

Digital devices display a straight line segment by plotting discrete points between the two end points. Discrete coordinate positions along the line path are calculated from the equation of the line. For a raster video display, the line color (intensity) is then loaded into the frame buffer at the corresponding pixel coordinates. Reading from the frame buffer, the video controller then plots “the screen pixels”.

Pixel positions are referenced according to scan-line number and column number (pixel position across a scan line). Scan lines are numbered consecutively from 0, starting at the bottom of the screen; and pixel columns are numbered from **0,** left to right across each scan line

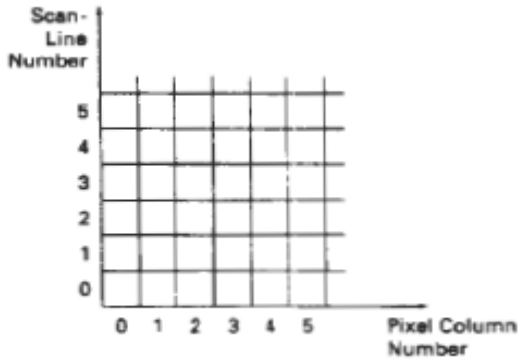


Figure : Pixel Postions reference by scan line number and column number

To load an intensity value into the frame buffer at a position corresponding to column x along scan line y,

setpixel (x, y)

To retrieve the current frame buffer intensity setting for a **specified** location we use a low level function

getpixel (x, y)

Line Drawing Algorithms

- Digital Differential Analyzer (DDA) Algorithm
- Bresenham's Line Algorithm
- Parallel Line Algorithm

The Cartesian ***slope-intercept equation*** for a straight line is

$$y = m \cdot x + b \qquad (1)$$

Where m as slope of the line and b as the y intercept

Given that the two endpoints of a line segment are specified at positions (x_1,y_1) and (x_2,y_2) as in figure we can determine the values for the slope m and y intercept b with the following calculations

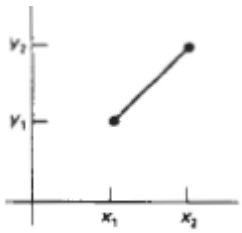


Figure : Line Path between endpoint positions (x_1,y_1) and (x_2,y_2)

$$m = \Delta y / \Delta x = y_2 - y1 / x_2 - x1 \quad (2)$$

$$b = y1 - m \,.\, x1 \quad (3)$$

For any given x interval Δx along a line, we can compute the corresponding y interval Δy

$$\Delta y = m \, \Delta x \quad (4)$$

We can obtain the x interval Δx corresponding to a specified Δy as

$$\Delta x = \Delta y / m \quad (5)$$

For lines with slope magnitudes $|m| < 1$, Δx can be set proportional to a small horizontal deflection voltage and the corresponding vertical deflection is then set proportional to Δy as calculated from Eq (4).

For lines whose slopes have magnitudes $|m| > 1$, Δy can be set proportional to a small vertical deflection voltage with the corresponding horizontal deflection voltage set proportional to Δx, calculated from Eq (5)

For lines with $m = 1$, $\Delta x = \Delta y$ and the horizontal and vertical deflections voltage are equal.

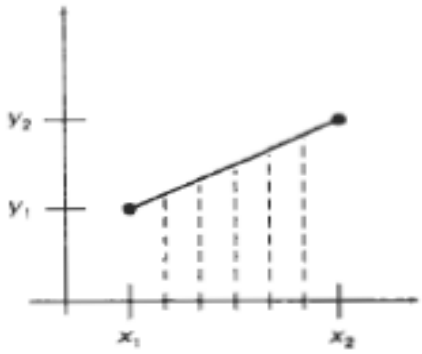


Figure : Straight line Segment with five sampling positions along the x axis between x1 and x2

Digital Differential Analyzer (DDA) Algortihm

The digital differential analyzer (DDA) is a scan-conversion line algorithm based on calculation either Δy or Δx

The line at unit intervals in one coordinate and determine corresponding integer values nearest the line path for the other coordinate.

A line with positive slop, if the slope is less than or equal to 1, at unit x intervals ($\Delta x=1$) and compute each successive y values as

$$y_{k+1} = y_k + m \qquad (6)$$

Subscript k takes integer values starting from 1 for the first point and increases by 1 until the final endpoint is reached. ***m*** can **be** any real number between 0 and 1 and, the calculated y values must be rounded to the nearest integer

For lines with a positive slope greater than 1 we reverse the roles of x and y, ($\Delta y=1$) and calculate each succeeding x value as

$$x_{k+1} = x_k + (1/m) \qquad (7)$$

Equation (6) and (7) are based on the assumption that lines are to be processed from the left endpoint to the right endpoint.

If this processing is reversed, $\Delta x=-1$ that the starting endpoint is at the right

$$y_{k+1} = y_k - m \qquad (8)$$

When the slope is greater than 1 and $\Delta y = -1$ with

$$x_{k+1} = x_k - (1/m) \qquad (9)$$

If the absolute value of the slope is less than 1 and the start endpoint is at the left, we set $\Delta x = 1$ and calculate y values with Eq. **(**6)

When the start endpoint is at the right (for the same slope), we set $\Delta x = -1$ and obtain y positions from Eq. (8). Similarly, when the absolute value of a negative slope is greater than 1, we use $\Delta y = -1$ and Eq. (9) or we use $\Delta y = 1$ and Eq. (7).

Algorithm

```
#define ROUND(a) ((int)(a+0.5))
void lineDDA (int xa, int ya, int xb, int yb)
{
int dx = xb - xa, dy = yb - ya, steps, k;
float xIncrement, yIncrement, x = xa, y = ya;
if (abs (dx) > abs (dy) steps = abs (dx) ;
else steps = abs dy);
xIncrement = dx / (float) steps;
yIncrement = dy / (float) steps
setpixel (ROUND(x), ROUND(y) ) :
for (k=0; k<steps; k++)
{
x += xIncrement;
y += yIncrement;
setpixel (ROUND(x), ROUND(y));
}
}
```

Algorithm Description:

Step 1 : Accept Input as two endpoint pixel positions
Step 2: Horizontal and vertical differences between the endpoint positions *are* assigned to parameters dx and dy (Calculate dx=xb-xa and dy=yb-ya).
Step 3: The difference with the greater magnitude determines the value of parameter steps.
Step 4 : Starting with pixel position (xa, ya), determine the offset needed at each step to generate the next pixel position along the line path.
Step 5: loop the following process for steps number of times

a. Use a unit of increment or decrement in the x and y direction
b. if xa is less than xb the values of increment in the x and y directions are 1 and m
c. if xa is greater than xb then the decrements -1 and – m are used.

Example : Consider the line from (0,0) to (4,6)

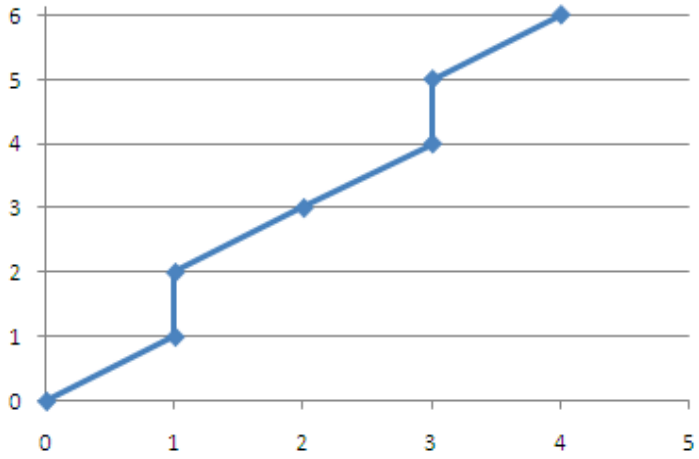
1. xa=0, ya =0 and xb=4 yb=6
2. dx=xb-xa = 4-0 = 4 and dy=yb-ya=6-0= 6
3. x=0 and y=0
4. 4 > 6 (false) so, steps=6
5. Calculate xIncrement = dx/steps = 4 / 6 = 0.66 and yIncrement = dy/steps =6/6=1

6. Setpixel(x,y) = Setpixel(0,0) (Starting Pixel Position)
7. Iterate the calculation for xIncrement and yIncrement for steps(6) number of times

k	x	Y	Plotting points (Rounded to Integer)
0	0+0.66=0.66	0+1=1	(1,1)
1	0.66+0.66=1.32	1+1=2	(1,2)
2	1.32+0.66=1.98	2+1=3	(2,3)
3	1.98+0.66=2.64	3+1=4	(3,4)
4	2.64+0.66=3.3	4+1=5	(3,5)
5	3.3+0.66=3.96	5+1=6	(4,6)

8. Tabulation of the each iteration

Result :



Advantages of DDA Algorithm

1. It is the simplest algorithm
2. It is a is a **faster method** for calculating pixel positions

Disadvantages of DDA Algorithm

1. Floating point arithmetic in DDA algorithm is still time-consuming
2. End point accuracy is poor

Bresenham's Line Algorithm

An accurate and efficient raster line generating algorithm developed by Bresenham, that uses only incremental integer calculations.

In addition, Bresenham's line algorithm can be adapted to display circles and other curves.

To illustrate Bresenham's approach, we- first consider the scan-conversion process for lines with positive slope less than 1.

Pixel positions along a line path are then determined by sampling at unit x intervals. Starting from the left endpoint (x_0,y_0) of a given line, we step to each successive column (x position) and plot the pixel whose scan-line y value is closest to the line path.

To determine the pixel (x_k,y_k) is to be displayed, next to decide which pixel to plot the column $x_{k+1}=x_{k+1}.(x_{k+1},y_k)$ and $.(x_{k+1},y_{k+1})$. At sampling position x_{k+1}, we label vertical pixel separations from the mathematical line path as d_1 and d_2. The y coordinate on the mathematical line at pixel column position x_{k+1} is calculated as

$$y = m(x_{k+1})+b \qquad (1)$$

Then

$$d_1 = y-y_k$$
$$= m(x_{k+1})+b-y_k$$
$$d_2 = (y_{k+1})-y$$
$$= y_{k+1}-m(x_{k+1})-b$$

To determine which of the two pixel is closest to the line path, efficient test that is based on the difference between the two pixel separations

$$d_1- d_2 = 2m(x_{k+1})-2y_k+2b-1 \qquad (2)$$

A decision parameter P_k for the k^{th} step in the line algorithm can be obtained by rearranging equation (2). By substituting $m=\Delta y/\Delta x$ where Δx and Δy are the vertical and horizontal separations of the endpoint positions and defining the decision parameter as

$$p_k = \Delta x\,(d_1- d_2)$$
$$= 2\Delta y\, x_k.-2\Delta x.\, y_k + c \qquad (3)$$

The sign of p_k is the same as the sign of $d_1- d_2$,since $\Delta x>0$

Parameter C is constant and has the value $2\Delta y + \Delta x(2b-1)$ which is independent of the pixel position and will be eliminated in the recursive calculations for P_k.

If the pixel at y_k is "closer" to the line path than the pixel at y_{k+1} ($d_1 < d_2$) than decision parameter P_k is negative. In this case, plot the lower pixel, otherwise plot the upper pixel. Coordinate changes along the line occur in unit steps in either the x or y directions.
To obtain the values of successive decision parameters using incremental integer calculations. At steps k+1, the decision parameter is evaluated from equation (3) as

$$P_{k+1} = 2\Delta y\, x_{k+1} - 2\Delta x.\, y_{k+1} + c$$

Subtracting the equation (3) from the preceding equation

$$P_{k+1} - P_k = 2\Delta y\,(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k)$$

But $x_{k+1} = x_k + 1$ so that

$$P_{k+1} = P_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k) \qquad (4)$$

Where the term $y_{k+1} - y_k$ is either 0 or 1 depending on the sign of parameter P_k

This recursive calculation of decision parameter is performed at each integer x position, starting at the left coordinate endpoint of the line.

The first parameter P_0 is evaluated from equation at the starting pixel position (x_0, y_0) and with m evaluated as $\Delta y/\Delta x$

$$P_0 = 2\Delta y - \Delta x \qquad (5)$$

Bresenham's line drawing for a line with a positive slope less than 1 in the following outline of the algorithm.

The constants $2\Delta y$ and $2\Delta y - 2\Delta x$ are calculated once for each line to be scan converted.

Bresenham's line Drawing Algorithm for |m| < 1

1. Input the two line endpoints and store the left end point in (x_0, y_0)
2. load (x_0, y_0) into frame buffer, ie. Plot the first point.
3. Calculate the constants Δx, Δy, $2\Delta y$ and obtain the starting value for the decision parameter as $P_0 = 2\Delta y - \Delta x$
4. At each x_k along the line, starting at k=0 perform the following test

If $P_k < 0$, the next point to plot is(x_{k+1},y_k) and

$P_{k+1} = P_k + 2\Delta y$

otherwise, the next point to plot is (x_{k+1},y_{k+1}) and

$P_{k+1} = P_k + 2\Delta y - 2\Delta x$

5. Perform step4 Δx times.

Implementation of Bresenham Line drawing Algorithm

```
void lineBres (int xa,int ya,int xb, int yb)
{
int dx = abs( xa – xb) , dy = abs (ya - yb);
int p = 2 * dy – dx;
int twoDy = 2 * dy, twoDyDx = 2 *(dy - dx);
int  x , y, xEnd;

/*  Determine which point to use as start, which as end * /

if (xa > x b )
{
x = xb;
y = yb;
xEnd = xa;
}
 else
{
x = xa;
y = ya;
xEnd = xb;
}
setPixel(x,y);
while(x<xEnd)
{
x++;
if (p<0)
p+=twoDy;
else
{
y++;
p+=twoDyDx;
}
setPixel(x,y);
```

}
}

Example : Consider the line with endpoints (20,10) to (30,18)

The line has the slope m= (18-10)/(30-20)=8/10=0.8

$\Delta x = 10$ $\Delta y=8$

The initial decision parameter has the value

$p_0 = 2\Delta y- \Delta x = 6$

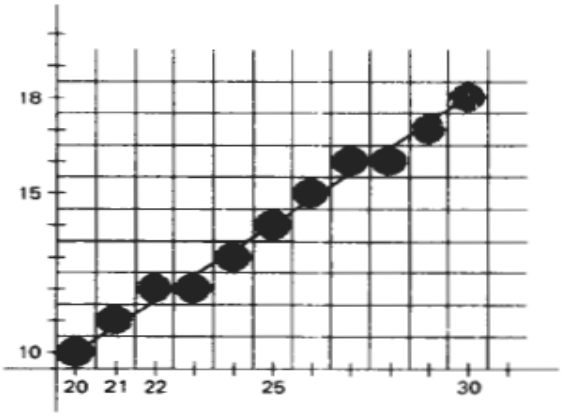
and the increments for calculating successive decision parameters are

$2\Delta y=16$ $2\Delta y-2 \Delta x= -4$

We plot the initial point $(x_0,y_0) = (20,10)$ and determine successive pixel positions along the line path from the decision parameter as

Tabulation

k	p_k	(x_k+1, y_K+1)
0	6	(21,11)
1	2	(22,12)
2	-2	(23,12)
3	14	(24,13)
4	10	(25,14)
5	6	(26,15)
6	2	(27,16)
7	-2	(28,16)
8	14	(29,17)
9	10	(30,18)



Advantages

- Algorithm is Fast
- Uses only integer calculations

Disadvantages

It is meant only for basic line drawing.

Line Function

The two dimension line function is **Polyline(n,wcPoints)** where **n** is assigned an integer value equal to the number of coordinate positions to be input and **wcPoints** is the array of input world-coordinate values for line segment endpoints.

polyline function is used to define a set of n – 1 connected straight line segments

To display a single straight-line segment we have to set n=2 and list the x and y values of the two endpoint coordinates in wcPoints.

Example : following statements generate 2 connected line segments with endpoints at (50, 100), (150, 250), and (250, 100)

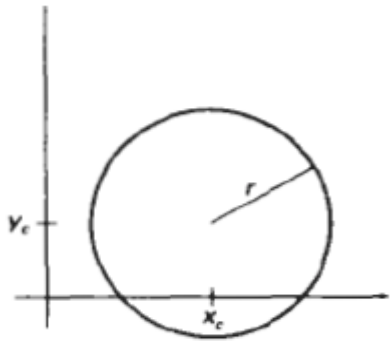
```
typedef struct myPt{int x, y;};
myPt wcPoints[3];
wcPoints[0] .x = 50; wcPoints[0] .y = 100;
wcPoints[1] .x = 150; wcPoints[1].y = 50;
wcPoints[2].x = 250; wcPoints[2] .y = 100;
polyline ( 3 , wcpoints);
```

Circle-Generating Algorithms

General function is available in a graphics library for displaying various kinds of curves, including circles and ellipses.

Properties of a circle

A circle is defined as a set of points that are all the given distance (x_c,y_c).



This distance relationship is expressed by the pythagorean theorem in Cartesian coordinates as

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \qquad (1)$$

Use above equation to calculate the position of points on a circle circumference by stepping along the x axis in unit steps from x_c-r to x_c+r and calculating the corresponding y values at each position as

$$y = y_c + (-) (r^2 - (xc - x)^2)^{1/2} \qquad (2)$$

This is not the best method for generating a circle for the following reason

Considerable amount of computation
Spacing between plotted pixels is not uniform

To eliminate the unequal spacing is to calculate points along the circle boundary using polar coordinates r and θ. Expressing the circle equation in parametric polar from yields the pair of equations

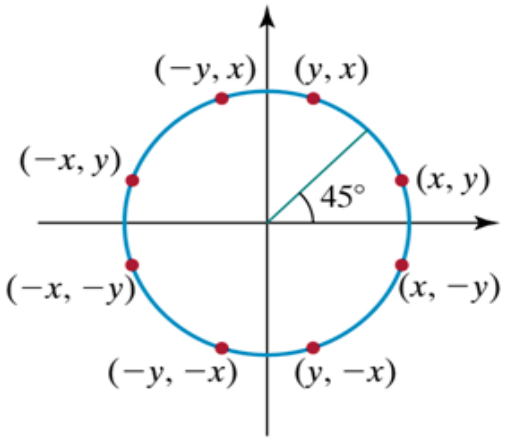
$$x = x_c + r\cos\theta \qquad y = y_c + r\sin\theta$$

When a display is generated with these equations using a fixed angular step size, a circle is plotted with equally spaced points along the circumference. To reduce calculations use a large angular separation between points along the circumference and connect the points with straight line segments to approximate the circular path.

Set the angular step size at 1/r. This plots pixel positions that are approximately one unit apart. The shape of the circle is similar in each quadrant. To determine the curve positions in the first quadrant, to generate he circle section in the second quadrant of the xy plane by nothing that the two circle sections are symmetric with respect to the y axis and circle section in the third and fourth quadrants can be obtained from sections in the first and second quadrants by considering symmetry between octants.

Circle sections in adjacent octants within one quadrant are symmetric with respect to the 45^0 line dividing the two octants. Where a point at position (x, y) on a one-eight circle sector is mapped into the seven circle points in the other octants of the xy plane.

To generate all pixel positions around a circle by calculating only the points within the sector from x=0 to y=0. the slope of the curve in this octant has an magnitude less than of equal to 1.0. at x=0, the circle slope is 0 and at x=y, the slope is -1.0.



Bresenham's line algorithm for raster displays is adapted to circle generation by setting up decision parameters for finding the closest pixel to the circumference at each sampling step. Square root evaluations would be required to computer pixel siatances from a circular path.

Bresenham's circle algorithm avoids these square root calculations by comparing the squares of the pixel separation distances. It is possible to perform a direct distance comparison without a squaring operation.

In this approach is to test the halfway position between two pixels to determine if this midpoint is inside or outside the circle boundary. This method is more easily applied to other conics and for an integer circle radius the midpoint approach generates the same pixel positions as the Bresenham circle algorithm.

For a straight line segment the midpoint method is equivalent to the bresenham line algorithm. The error involved in locating pixel positions along any conic section using the midpoint test is limited to one half the pixel separations.

Midpoint circle Algorithm:

In the raster line algorithm at unit intervals and determine the closest pixel position to the specified circle path at each step for a given radius r and screen center position (x_c,y_c) set up our algorithm to calculate pixel positions around a circle path centered at the coordinate position by adding x_c to x and y_c to y.

To apply the midpoint method we define a circle function as

$$f_{circle}(x,y) = x^2+y^2-r^2$$

Any point (x,y) on the boundary of the circle with radius r satisfies the equation $f_{circle}(x,y)=0$. If the point is in the interior of the circle, the circle function is negative. And if the point is outside the circle the, circle function is positive

$f_{circle}(x,y) <0$, if (x,y) is inside the circle boundary
$=0$, if (x,y) is on the circle boundary
>0, if (x,y) is outside the circle boundary

The tests in the above eqn are performed for the midposition sbteween pixels near the circle path at each sampling step. The circle function is the decision parameter in the midpoint algorithm.

Midpoint between candidate pixels at sampling position x_{k+1} along a circular path. Fig -1 shows the midpoint between the two candidate pixels at sampling position x_{k+1}. To plot the pixel at (x_k,y_k) next need to determine whether the pixel at position (x_{k+1},y_k) or the one at position (x_{k+1},y_{k-1}) is circular to the circle.

Our decision parameter is the circle function evaluated at the midpoint between these two pixels

$$P_k= f_{circle}(x_{k+1},y_k-1/2)$$
$$=(x_{k+1})^2+(y_k-1/2)^2-r^2$$

If $P_k <0$, this midpoint is inside the circle and the pixel on scan line y_k is closer to the circle boundary. Otherwise the mid position is outside or on the circle boundary and select the pixel on scan line y_k -1.

Successive decision parameters are obtained using incremental calculations. To obtain a recursive expression for the next decision parameter by evaluating the circle function at sampling position $x_{k+1+1}= x_{k+2}$

$$P_k= f_{circle}(x_{k+1}+1,y_{k+1}-1/2)$$
$$=[(x_{k+1})+1]^2+(y_{k+1}-1/2)^2-r^2$$
or
$$P_{k+1}=P_k+2(x_{k+1})+(y^2_{k+1}-y^2_k)-(y_{k+1}-y_k)+1$$

Where y_{k+1} is either y_k or y_{k-1} depending on the sign of P_k .
Increments for obtaining P_{k+1} are either $2x_{k+1}+1$ (if P_k is negative) or $2x_{k+1}+1-2\ y_{k+1}$.

Evaluation of the terms $2x_{k+1}$ and $2\ y_{k+1}$ can also be done incrementally as

$$2x_{k+1}=2x_{k+2}$$
$$2\ y_{k+1}=2\ y_{k-2}$$

At the Start position (0,r) these two terms have the values 0 and 2r respectively. Each successive value for the $2x_{k+1}$ term is obtained by adding 2 to the previous value and each successive value for the $2y_{k+1}$ term is obtained by subtracting 2 from the previous value.

The initial decision parameter is obtained by evaluating the circle function at the start position $(x_0,y_0)=(0,r)$

$$P_0= f_{circle}(1,r-1/2)$$
$$=1+(r-1/2)^2-r^2$$

or

$$P_0=(5/4)-r$$

If the radius r is specified as an integer

$$P_0=1-r(\text{for r an integer})$$

Midpoint circle Algorithm

1. Input radius r and circle center (x_c,y_c) and obtain the first point on the circumference of the circle centered on the origin as

$(x_0,y_0) = (0,r)$

2. Calculate the initial value of the decision parameter as $P_0=(5/4)-r$
3. At each x_k position, starting at k=0, perform the following test. If $P_k <0$ the next point along the circle centered on (0,0) is (x_{k+1},y_k) and $P_{k+1}=P_k+2x_{k+1}+1$

Otherwise the next point along the circle is $(x_{k+1,}y_{k-1})$ and $P_{k+1}=P_k+2x_{k+1}+1-2\ y_{k+1}$

Where $2x_{k+1}=2x_{k+2}$ and $2y_{k+1}=2y_{k-2}$

4. Determine symmetry points in the other seven octants.

5. Move each calculated pixel position (x,y) onto the circular path centered at (x_c,y_c) and plot the coordinate values.

$x=x+x_c \quad y=y+y_c$

6. Repeat step 3 through 5 until x>=y.

Example : Midpoint Circle Drawing

Given a circle radius r=10

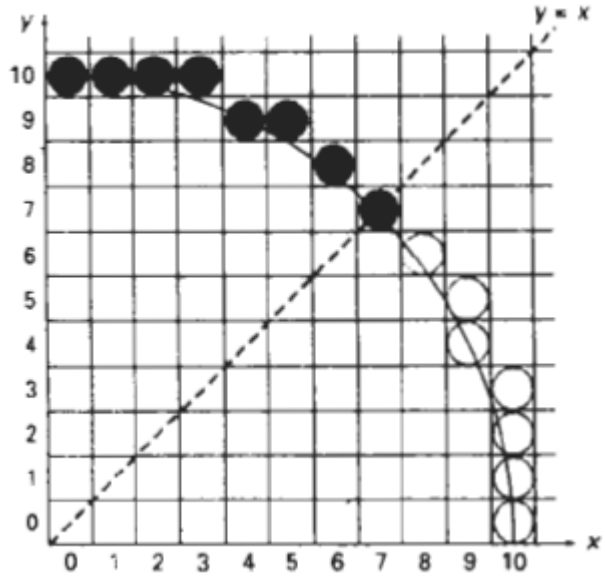
The circle octant in the first quadrant from x=0 to x=y. The initial value of the decision parameter is $P_0=1-r = -9$

For the circle centered on the coordinate origin, the initial point is $(x_0,y_0)=(0,10)$ and initial increment terms for calculating the decision parameters are

$2x_0=0 \quad , \quad 2y_0=20$

Successive midpoint decision parameter values and the corresponding coordinate positions along the circle path are listed in the following table.

k	p_k	(x_{k+1}, y_{k-1})	$2x_{k+1}$	$2y_{k+1}$
0	-9	(1,10)	2	20
1	-6	(2,10)	4	20
2	-1	(3,10)	6	20
3	6	(4,9)	8	18
4	-3	(5,9)	10	18
5	8	(6,8)	12	16
6	5	(7,7)	14	14



Implementation of Midpoint Circle Algorithm

```
void circleMidpoint (int xCenter, int yCenter, int radius)
{
int x = 0;
int y = radius;
int p = 1 - radius;
void circlePlotPoints (int, int, int, int);
/*  Plot first set of points */
circlePlotPoints (xCenter, yCenter, x, y);
while (x < y)
{
x++ ;
if (p < 0)
p +=2*x +1;
else
{
y--;
p +=2* (x - Y) + 1;
}
circlePlotPoints(xCenter, yCenter,  x,  y)
}
}
void circlePlotPolnts (int xCenter, int yCenter, int x, int y)
{
setpixel (xCenter + x, yCenter + y ) ;
```

```
setpixel (xCenter - x. yCenter + y);
setpixel (xCenter + x, yCenter - y);
setpixel (xCenter - x, yCenter - y ) ;
setpixel (xCenter + y, yCenter + x);
setpixel (xCenter - y , yCenter + x);
setpixel (xCenter t y , yCenter - x);
setpixel (xCenter - y , yCenter - x);
}
```

Ellipse-Generating Algorithms

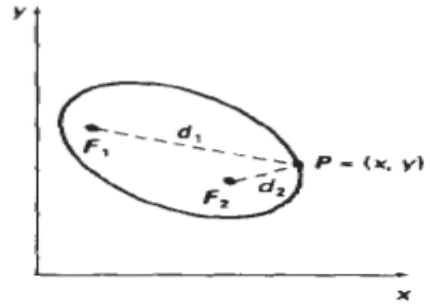
An ellipse is an elongated circle. Therefore, elliptical curves can be generated by modifying circle-drawing procedures to take into account the different dimensions of an ellipse along the major and minor axes.

Properties of ellipses

An ellipse can be given in terms of the distances from any point on the ellipse to two fixed positions called the **foci** of the ellipse. The sum of these two distances is the same values for all points on the ellipse.

If the distances to the two focus positions from any point p=(x,y) on the ellipse are labeled d1 and d2, then the general equation of an ellipse can be stated as

d1+d2=constant



Expressing distances d1 and d2 in terms of the focal coordinates F1=(x_1,y_2) and F2=(x_2,y_2)

$$\mathbf{sqrt((x-x_1)^2+(y-y_1)^2)+sqrt((x-x_2)^2+(y-y_2)^2)=constant}$$

By squaring this equation isolating the remaining radical and squaring again. The general ellipse equation in the form

$$Ax^2+By^2+Cxy+Dx+Ey+F=0$$

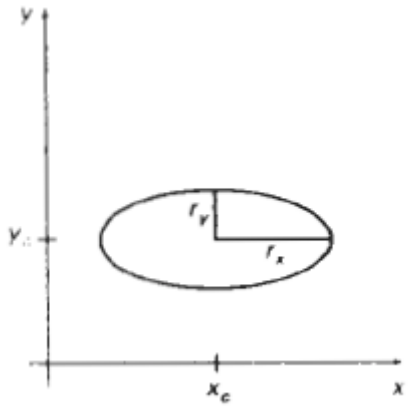
The coefficients A,B,C,D,E, and F are evaluated in terms of the focal coordinates and the dimensions of the major and minor axes of the ellipse.

The major axis is the straight line segment extending from one side of the ellipse to the other through the foci. The minor axis spans the shorter dimension of the ellipse, perpendicularly bisecting the major axis at the halfway position (ellipse center) between the two foci.

An interactive method for specifying an ellipse in an arbitrary orientation is to input the two foci and a point on the ellipse boundary.

Ellipse equations are simplified if the major and minor axes are oriented to align with the coordinate axes. The major and minor axes oriented parallel to the x and y axes parameter r_x for this example labels the semi major axis and parameter r_y labels the semi minor axis

$$((x-x_c)/r_x)^2+((y-y_c)/r_y)^2=1$$

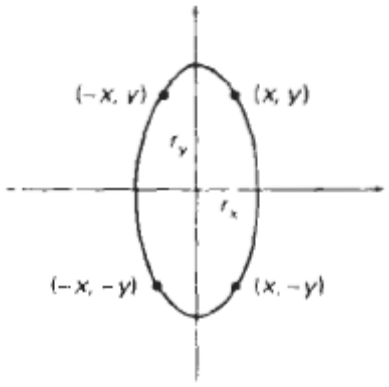


Using polar coordinates r and θ, to describe the ellipse in Standard position with the parametric equations

$$x=x_c+r_x\cos\theta$$
$$y=y_c+r_x\sin\theta$$

Angle θ called the eccentric angle of the ellipse is measured around the perimeter of a bounding circle.

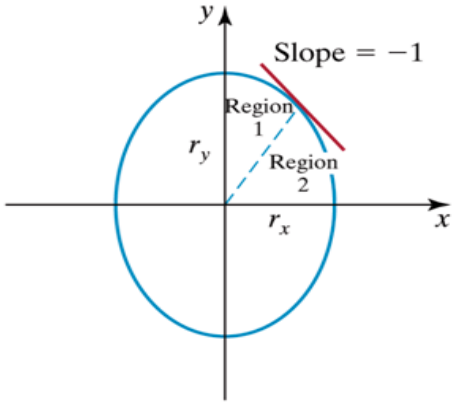
We must calculate pixel positions along the elliptical arc throughout one quadrant, and then we obtain positions in the remaining three quadrants by symmetry



Midpoint ellipse Algorithm

The midpoint ellipse method is applied throughout the first quadrant in two parts.

The below figure show the division of the first quadrant according to the slope of an ellipse with rx<ry.



In the x direction where the slope of the curve has a magnitude less than 1 and unit steps in the y direction where the slope has a magnitude greater than 1.

Region 1 and 2 can be processed in various ways

1. Start at position $(0,r_y)$ and step clockwise along the elliptical path in the first quadrant shifting from unit steps in x to unit steps in y when the slope becomes less than -1
2. Start at $(r_x,0)$ and select points in a counter clockwise order.
   2.1 Shifting from unit steps in y to unit steps in x when the slope becomes greater than -1.0

2.2 Using parallel processors calculate pixel positions in the two regions simultaneously

3. Start at (0,ry)

step along the ellipse path in clockwise order throughout the first quadrant ellipse function $(x_c,y_c)=(0,0)$

$f_{ellipse}(x,y)=ry^2x^2+rx^2y^2-rx^2\,ry^2$

which has the following properties:

$f_{ellipse}(x,y)$ <0, if (x,y) is inside the ellipse boundary

=0, if(x,y) is on ellipse boundary

>0, if(x,y) is outside the ellipse boundary

Thus, the ellipse function $f_{ellipse}(x,y)$ serves as the decision parameter in the midpoint algorithm.

Starting at (0,ry):

Unit steps in the x direction until to reach the boundary between region 1 and region 2. Then switch to unit steps in the y direction over the remainder of the curve in the first quadrant.

At each step to test the value of the slope of the curve. The ellipse slope is calculated

$$dy/dx= -(2ry^2x/2rx^2y)$$

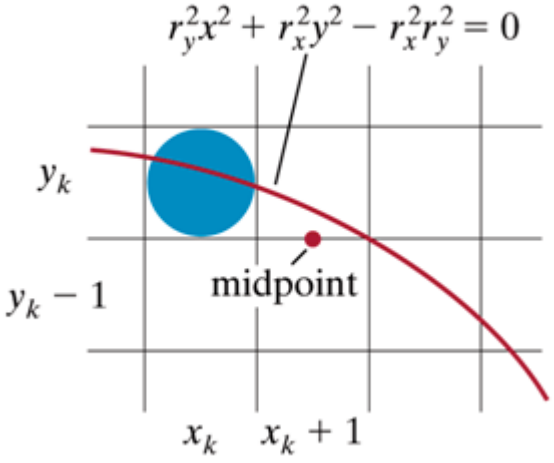
At the boundary between region 1 and region 2

$$dy/dx = -1.0 \text{ and } 2ry^2x=2rx^2y$$

to more out of region 1 whenever

$$2ry^2x>=2rx^2y$$

The following figure shows the midpoint between two candidate pixels at sampling position x_{k+1} in the first region.



To determine the next position along the ellipse path by evaluating the decision parameter at this mid point

$$P1_k = f_{ellipse}(x_{k+1}, y_k - 1/2)$$
$$= ry^2 (x_{k+1})^2 + rx^2 (y_k - 1/2)^2 - rx^2 ry^2$$

if $P1_k < 0$, the midpoint is inside the ellipse and the pixel on scan line y_k is closer to the ellipse boundary. Otherwise the midpoint is outside or on the ellipse boundary and select the pixel on scan line y_{k-1}

At the next sampling position ($x_{k+1}+1=x_k+2$) the decision parameter for region 1 is calculated as

$$p1_{k+1} = f_{ellipse}(x_{k+1} + 1, y_{k+1} - ½)$$

$$= r_y^2[(x_k + 1) + 1]^2 + r_x^2 (y_{k+1} - ½)^2 - r_x^2 r_y^2$$

Or

$$p1_{k+1} = p1_k + 2 r_y^2(x_k + 1) + r_y^2 + r_x^2 [(y_{k+1} - ½)^2 - (y_k - ½)^2]$$

Where y_{k+1} is y_k or y_{k-1} depending on the sign of $P1_k$.

Decision parameters are incremented by the following amounts

$$\text{increment} = \{ 2 r_y^2(x_k + 1) + r_y^2 \quad \text{if } p1_k < 0 \}$$
$$\{ 2 r_y^2(x_k + 1) + r_y^2 - 2r_x^2 y_{k+1} \quad \text{if } p1_k \geq 0 \}$$

Increments for the decision parameters can be calculated using only addition and subtraction as in the circle algorithm.

The terms $2ry^2$ x and $2rx^2$ y can be obtained incrementally. At the initial position $(0,r_y)$ these two terms evaluate to

$$2\, r_y^2 x = 0$$

$$2r_x^2\, y = 2r_x^2\, r_y$$

x and y are incremented updated values are obtained by adding $2ry^2$ to the current value of the increment term and subtracting $2rx^2$ from the current value of the increment term. The updated increment values are compared at each step and more from region 1 to region 2. when the condition 4 is satisfied.

In region 1 the initial value of the decision parameter is obtained by evaluating the ellipse function at the start position

$$(x_0,y_0) = (0,r_y)$$

region 2 at unit intervals in the negative y direction and the midpoint is now taken between horizontal pixels at each step for this region the decision parameter is evaluated as

$p1_0 = f_{ellipse}(1,r_y - ½)$

$= r_y^2 + r_x^2 (r_y - ½)^2 - r_x^2\, r_y^2$

Or

$p1_0 = r_y^2 - r_x^2\, r_y + ¼\, r_x^2$

over region 2, we sample at unit steps in the negative y direction and the midpoint is now taken between horizontal pixels at each step. For this region, the decision parameter is evaluated as

$p2_k = f_{ellipse}(x_k + ½ , y_k - 1)$

$= r_y^2 (x_k + ½)^2 + r_x^2 (y_k - 1)^2 - r_x^2\, r_y^2$

1. If $P2_k > 0$, the mid point position is outside the ellipse boundary, and select the pixel at x_k.

2. If $P2_k <= 0$, the mid point is inside the ellipse boundary and select pixel position x_{k+1}.

To determine the relationship between successive decision parameters in region 2 evaluate the ellipse function at the sampling step : $y_{k+1} -1= y_{k-2}$.

$$P2_{k+1} = f_{ellipse}(x_{k+1} +½, y_{k+1} -1)$$

$$=r_y^2(x_k +½)^2 + r_x^2 [(y_{k+1} -1) -1]^2 - r_x^2 r_y^2$$

or

$$p2_{k+1} = p2_k -2 r_x^2(y_k -1) + r_x^2 + r_y^2 [(x_{k+1} +½)^2 - (x_k +½)^2]$$

With x_{k+1}set either to $x_k$or x_{k+1}, depending on the sign of P2k. when we enter region 2, the initial position (x_0,y_0) is taken as the last position. Selected in region 1 and the initial decision parameter in region 2 is then

$$p2_0 = f_{ellipse}(x_0 +½ , y_0 - 1)$$

$$= r_y^2 (x_0 +½)^2 + r_x^2 (y_0 - 1)^2 - r_x^2 r_y^2$$

To simplify the calculation of P20, select pixel positions in counter clock wise order starting at $(r_x,0)$. Unit steps would then be taken in the positive y direction up to the last position selected in region 1.

Mid point Ellipse Algorithm

1. Input r_x,r_y and ellipse center (x_c,y_c) and obtain the first point on an ellipse centered on the origin as
   $(x_0,y_0) = (0,r_y)$

2. Calculate the initial value of the decision parameter in region 1 as
   $P1_0=r_y^2-r_x^2r_y +(1/4)r_x^2$

3. At each x_k position in region1 starting at k=0 perform the following test. If $P1_k<0$, the next point along the ellipse centered on (0,0) is (x_{k+1}, y_k) and

   $$p1_{k+1} = p1_k +2 r_y^2 x_{k+1} + r_y^2$$

Otherwise the next point along the ellipse is (x_k+1, y_k-1) and

$$p1_{k+1} = p1_k +2 r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_y^2$$

with

$$2r_y^2x_k + 1 = 2r_y^2x_k + 2r_y^2$$
$$2r_x^2y_k + 1 = 2r_x^2y_k + 2r_x^2$$

And continue until $2ry^2 x >= 2rx^2 y$

4. Calculate the initial value of the decision parameter in region 2 using the last point (x_0,y_0) is the last position calculated in region 1.

$$p2_0 = r_y^2(x_0+1/2)^2 + r_x^2(y_o-1)^2 - r_x^2r_y^2$$

5. At each position yk in region 2, starting at k=0 perform the following test, If $p2_k>0$ the next point along the ellipse centered on (0,0) is (x_k,y_{k-1}) and

$$p2_{k+1} = p2_k - 2r_x^2y_{k+1} + r_x^2$$

Otherwise the next point along the ellipse is (x_k+1,y_k-1) and

$$p2_{k+1} = p2_k + 2r_y^2x_{k+1} - 2r_x^2y_{k+1} + r_x^2$$

Using the same incremental calculations for x any y as in region 1.

6. Determine symmetry points in the other three quadrants.
7. Move each calculate pixel position (x,y) onto the elliptical path centered on (x_c,y_c) and plot the coordinate values
$$x=x+x_c, \quad y=y+y_c$$
8. Repeat the steps for region1 unit $2r_y^2x >= 2r_x^2y$

Example : Mid point ellipse drawing

Input ellipse parameters $r_x=8$ and $r_y=6$ the mid point ellipse algorithm by determining raster position along the ellipse path is the first quadrant. Initial values and increments for the decision parameter calculations are

$2r_y^2 x=0$ (with increment $2r_y^2=72$)
$2r_x^2 y=2r_x^2 r_y$ (with increment $-2r_x^2= -128$)

For region 1 the initial point for the ellipse centered on the origin is (x_0,y_0) = (0,6) and the initial decision parameter value is

$p1_0=r_y^2-r_x^2r_y^2+1/4r_x^2=-332$

Successive midpoint decision parameter values and the pixel positions along the ellipse are listed in the following table.

k	$p1_k$	x_{k+1},y_{k+1}	$2r_y^2x_{k+1}$	$2r_x^2y_{k+1}$
0	-332	(1,6)	72	768
1	-224	(2,6)	144	768
2	-44	(3,6)	216	768
3	208	(4,5)	288	640
4	-108	(5,5)	360	640
5	288	(6,4)	432	512
6	244	(7,3)	504	384

Move out of region 1, $2ry2x > 2rx^2y$.

For a region 2 the initial point is $(x_0,y_0)=(7,3)$ and the initial decision parameter is

$$p2_0 = f_{ellipse}(7+1/2,2) = -151$$

The remaining positions along the ellipse path in the first quadrant are then calculated as

k	$P2_k$	x_{k+1},y_{k+1}	$2r_y^2x_{k+1}$	$2r_x^2y_{k+1}$
0	-151	(8,2)	576	256
1	233	(8,1)	576	128
2	745	(8,0)	-	-

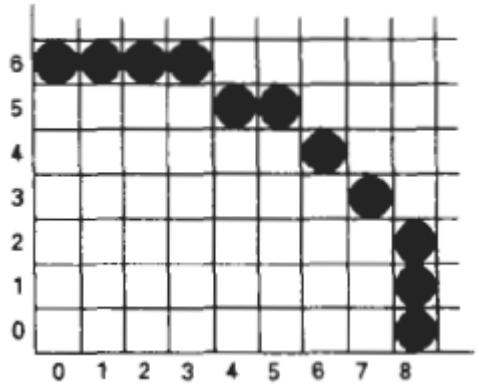
Implementation of Midpoint Ellipse drawing

```
#define Round(a) ((int)(a+0.5))
void ellipseMidpoint (int xCenter, int yCenter, int Rx, int Ry)
{
int  Rx2=Rx*Rx;
int  Ry2=Ry*Ry;
int twoRx2 = 2*Rx2;
int twoRy2 = 2*Ry2;
int p;
int x = 0;
```

```
int y = Ry;
int px = 0;
int py = twoRx2* y;
void ellipsePlotPoints ( int , int , int , int ) ;
/*  Plot the first set of points */
ellipsePlotPoints (xcenter, yCenter, x,y ) ;

/ * Region 1 */
p = ROUND(Ry2 - (Rx2* Ry) + (0.25*Rx2));
while (px < py)
{
x++;
px += twoRy2;
i f (p < 0)
p += Ry2 + px;
else
{
y - - ;
py -= twoRx2;
p += Ry2 + px - py;
}
ellipsePlotPoints(xCenter, yCenter,x,y);
}
/* Region 2 */
p = ROUND (Ry2*(x+0.5)*' (x+0.5)+ Rx2*(y- l )* (y- l ) - Rx2*Ry2);
while (y > 0 )
{
y--;
py -= twoRx2;
i f (p > 0)
p += Rx2 - py;
else
{
x++;
px+=twoRy2;
p+=Rx2-py+px;
}
ellipsePlotPoints(xCenter, yCenter,x,y);
}
}
void ellipsePlotPoints(int xCenter, int yCenter,int x,int y);
{
```

```
setpixel (xCenter + x, yCenter + y);
setpixel (xCenter - x, yCenter + y);
setpixel (xCenter + x, yCenter - y);
setpixel (xCenter- x, yCenter - y);
}
```



Attributes of output primitives

Any parameter that affects the way a primitive is to be displayed is referred to as an attribute parameter. Example attribute parameters are color, size etc. A line drawing function for example could contain parameter to set color, width and other properties.

1. Line Attributes
2. Curve Attributes
3. Color and Grayscale Levels
4. Area Fill Attributes
5. Character Attributes
6. Bundled Attributes

Line Attributes

Basic attributes of a straight line segment are its type, its width, and its color. In some graphics packages, lines can also be displayed using selected pen or brush options

- Line Type
- Line Width
- Pen and Brush Options
- Line Color

Line type

Possible selection of line type attribute includes solid lines, dashed lines and dotted lines. To set line type attributes in a **PHIGS** application program, a user invokes the function

setLinetype (lt)

Where parameter lt is assigned a positive integer value of 1, 2, 3 or 4 to generate lines that are solid, dashed, dash dotted respectively. Other values for line type parameter it could be used to display variations in dot-dash patterns.

Line width

Implementation of line width option depends on the capabilities of the output device to set the line width attributes.

setLinewidthScaleFactor(lw)

Line width parameter lw is assigned a positive number to indicate the relative width of line to be displayed. A value of 1 specifies a standard width line. A user could set lw to a value of 0.5 to plot a line whose width is half that of the standard line. Values greater than 1 produce lines thicker than the standard.

Line Cap

We can adjust the shape of the **line** ends to give them a better appearance by adding line caps.

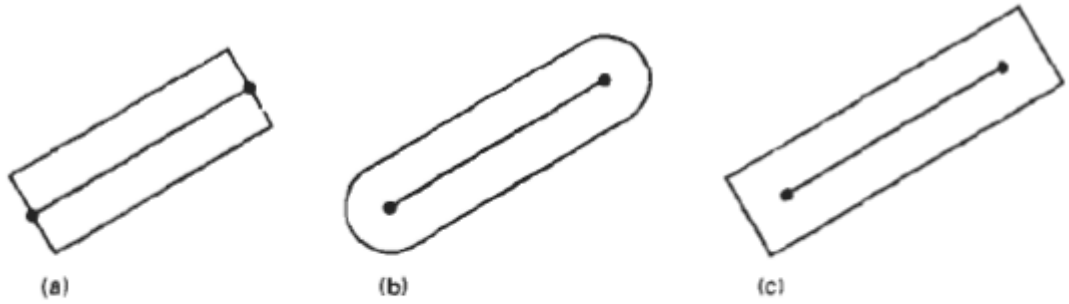
There are three types of line cap. They are

- Butt cap
- Round cap
- Projecting square cap

Butt cap obtained by adjusting the end positions of the component parallel **lines** so that the thick line is displayed with square ends that are perpendicular to the line path.

Round cap obtained by adding a filled semicircle to each butt cap. The circular arcs are centered on the line endpoints and have a diameter equal to the line thickness

Projecting square cap extend the line and add butt caps that are positioned one-half of the line width beyond the specified endpoints.

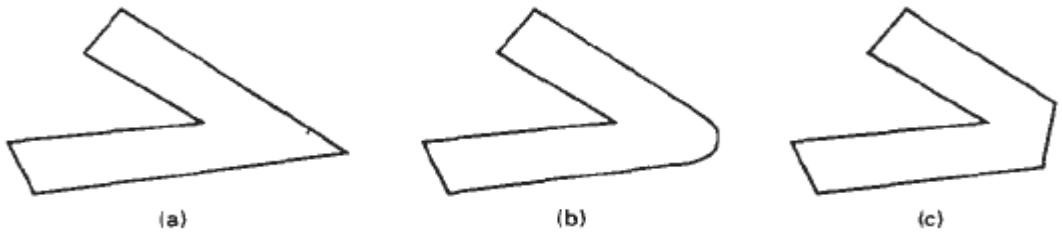


Thick lines drawn with (a) butt caps, (b) round caps, and (c) projecting square caps.

Three possible methods for smoothly joining two line segments

- Mitter Join
- Round Join
- Bevel Join

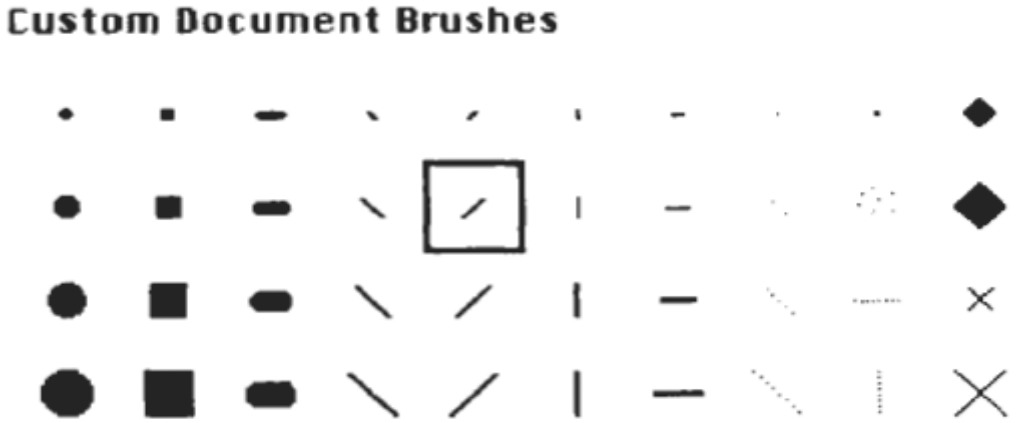
1. A **miter join** accomplished by extending the outer boundaries of each of the two lines until they meet.
2. A **round join** is produced by capping the connection between the two segments with a circular boundary whose diameter is equal to the width.
3. A **bevel join** is generated by displaying the line segment with but caps and filling in tri angular gap where the segments meet



Thick line segments connected with (a) miter join, (b) round join, and (c) bevel join.

Pen and Brush Options

With some packages, lines can be displayed with pen or brush selections. Options in this category include shape, size, and pattern. Some possible pen or brush shapes are given in Figure



Line color

A poly line routine displays a line in the current color by setting this color value in the frame buffer at pixel locations along the line path using the set pixel procedure.
We set the line color value in **PHIGS** with the function

setPolylineColourIndex (lc)

Nonnegative integer values, corresponding to allowed color choices, are assigned to the line color parameter lc

Example : Various line attribute commands in an applications program is given by the following sequence of statements

```
setLinetype(2);
setLinewidthScaleFactor(2);
setPolylineColourIndex (5);
polyline(n1,wc  points1);
setPolylineColorIindex(6);
poly line (n2, wc points2);
```

This program segment would display two figures, drawn with double-wide dashed lines. The first is displayed in a color corresponding to code 5, and the second in color 6.

Curve attributes

Parameters for curve attribute are same as those for line segments. Curves displayed with varying colors, widths, dot –dash patterns and available pen or brush options

Color and Grayscale Levels

Various color and intensity-level options can be made available to a user, depending on the capabilities and design objectives of a particular system

In a color raster system, the number of color choices available depends on the amount of storage provided per pixel in the frame buffer

Color-information can be stored in the frame buffer in two ways:

- We can store color codes directly in the frame buffer
- We can put the color codes in a separate table and use pixel values as an index into this table

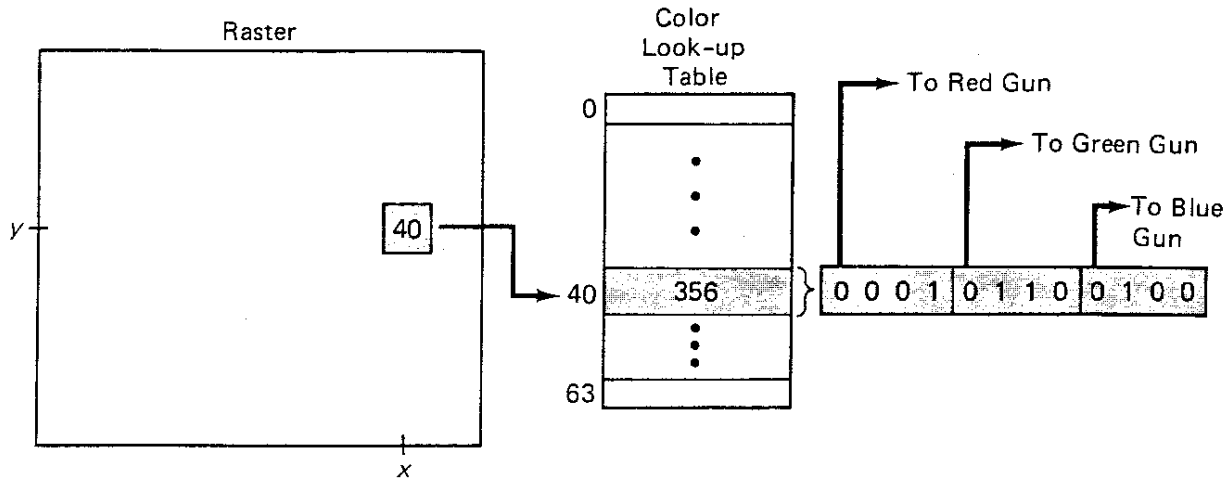
With the direct storage scheme, whenever a particular color code is specified in an application program, the corresponding binary value is placed in the frame buffer for each-component pixel in the output primitives to be displayed in that color.

A minimum number of colors can be provided in this scheme with 3 bits of storage per pixel, as shown in Table

THE EIGHT COLOR CODES FOR A THREE-BIT PER PIXEL FRAME BUFFER

Color	*Stored Color Values in Frame Buffer*			*Displayed Color*
Code	*RED*	*GREEN*	*BLUE*	
0	0	0	0	Black
1	0	0	1	Blue
2	0	1	0	Green
3	0	1	1	Cyan
4	1	0	0	Red
5	1	0	1	Magenta
6	1	1	0	Yellow
7	1	1	1	White

Color tables(Color Lookup Tables) are an alternate means for providing extended color capabilities to a user without requiring large frame buffers



3 bits - 8 choice of color
6 bits – 64 choice of color
8 bits – 256 choice of color

A user can set color-table entries in a PHIGS applications program with the function

setColourRepresentation (ws, ci, colorptr)

Parameter **ws** identifies the workstation output device; parameter ci specifies the color index, which is the color-table position number **(0** to **255**) and parameter colorptr points to a trio of RGB color values **(r, g, b)** each specified in the range from **0** to 1

Grayscale

With monitors that have no color capability, color functions can be used in an application program to set the shades of gray, or grayscale, for displayed primitives. Numeric values over the range from 0 to 1 can be used to specify grayscale levels, which are then converted to appropriate binary codes for storage in the raster.

INTENSITY CODES FOR A FOUR-LEVEL GRAYSCALE SYSTEM

Intensity Codes	*Stored Intensity Values In The Frame Buffer (Binary Code)*		*Displayed Grayscale*
0.0	0	(00)	Black
0.33	1	(01)	Dark gray
0.67	2	(10)	Light gray
1.0	3	(11)	White

Intensity = 0.5[min(r,g,b)+max(r,g,b)]

Area fill Attributes

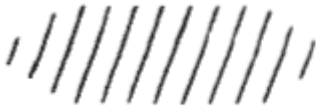
Options for filling a defined region include a choice between a solid color or a pattern fill and choices for particular colors and patterns

Fill Styles

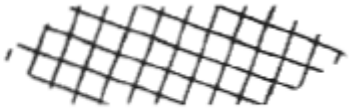
Areas are displayed with three basic fill styles: hollow with a color border, filled with a solid color, or filled with a specified pattern or design. A basic fill style is selected in a **PHIGS** program with the function

setInteriorStyle(fs)

Values for the fill-style parameter f**s** include hollow, solid**,** and pattern. Another value for fill style is hatch, which is used to fill an area with selected hatching patterns-parallel lines or crossed lines



Diagonal
Hatch Fill



Diagonal
Cross-Hatch Fill

The color for a solid interior or for a hollow area outline is chosen with where fill color parameter fc is set to the desired color code

setInteriorColourIndex(fc)

Pattern Fill

We select fill patterns with setInteriorStyleIndex (pi) where pattern index parameter pi specifies a table position

For example, the following set of statements would fill the area defined in the fillArea command with the second pattern type stored in the pattern table:

SetInteriorStyle(pattern)
SetInteriorStyleIndex(2);
Fill area (n, points)

Index (pi)	*Pattern (cp)*
1	$\begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix}$
2	$\begin{bmatrix} 2 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 1 & 2 \end{bmatrix}$

Character Attributes

The appearance of displayed character is controlled by attributes such as font, size, color and orientation. Attributes can be set both for entire character strings (text) and for individual characters defined as marker symbols

Text Attributes

The choice of font or type face is set of characters with a particular design style as courier, Helvetica, times roman, and various symbol groups.

The characters in a selected font also be displayed with styles. (solid, dotted, double) in **bold face** in *italics*, and in outline or shadow styles.

A particular font and associated stvle is selected in a PHIGS program by setting an integer code for the text font parameter tf in the function

setTextFont(tf)

Control of text color (or intensity) is managed from an application program with

setTextColourIndex(tc)

where text color parameter tc specifies an allowable color code.

Text size can be adjusted without changing the width to height ratio of characters with

SetCharacterHeight (ch)

Parameter ch is assigned a real value greater than 0 to set the coordinate height of capital letters

The width only of text can be set with function.

SetCharacterExpansionFactor(cw)

Where the character width parameter cw is set to a positive real value that scales the body width of character

width 0.5

width 1.0

width 2.0

Spacing between characters is controlled separately with

setCharacterSpacing(cs)

where the character-spacing parameter **cs** can he assigned any real value

Spacing 0.0

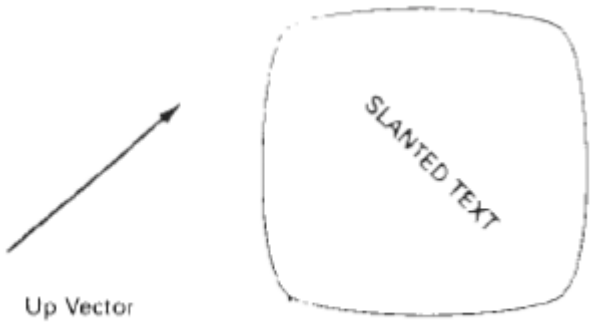
Spacing 0.5

Spacing 1.0

The orientation for a displayed character string is set according to the direction of the character up vector

setCharacterUpVector(upvect)

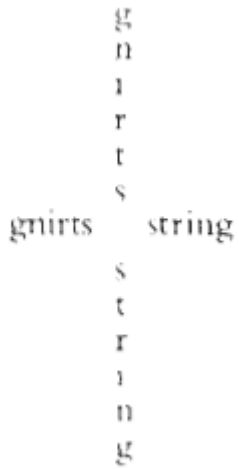
Parameter **upvect** in this function is assigned two values that specify the ***x*** and y vector components. For example, with **upvect** = (1, 1), the direction of the up vector is **45^o** and text would be displayed as shown in Figure.



To arrange character strings vertically or horizontally

setTextPath (tp)

Where the text path parameter tp can be assigned the value: right, left, up, or down



Another handy attribute for character strings is alignment. This attribute specifies how text is to be positioned with respect to the $start coordinates. Alignment attributes are set with

setTextAlignment (h,v)

where parameters h and v control horizontal and vertical alignment. Horizontal alignment is set by assigning h a value of left, center, or right. Vertical alignment is set by assigning v a value of top, cap, half, base or bottom.

A precision specification for text display is given with

setTextPrecision (tpr)

tpr is assigned one of values string, char or stroke.

Marker Attributes

A marker symbol is a single character that can he displayed in different colors and in different sizes. Marker attributes are implemented by procedures that load the chosen character into the raster at the defined positions with the specified color and size. We select a particular character to be the marker symbol with

setMarkerType(mt)

where marker type parameter mt is set to an integer code. Typical codes for marker type are the integers 1 through 5, specifying, respectively, a dot (.) a vertical cross (+), an asterisk (*), a circle (o), and a diagonal cross (X).

We set the marker size with

setMarkerSizeScaleFactor(ms)

with parameter marker size ms assigned a positive number. This scaling parameter is applied to the nominal size for the particular marker symbol chosen. Values greater than 1 produce character enlargement; values less than 1 reduce the marker size.

Marker color is specified with

setPolymarkerColourIndex(mc)

A selected color code parameter mc is stored in the current attribute list and used to display subsequently specified marker primitives

Bundled Attributes

The procedures considered so far each function reference a single attribute that specifies exactly how a primitive is to be displayed these specifications are called individual attributes.

A particular set of attributes values for a primitive on each output device is chosen by specifying appropriate table index. Attributes specified in this manner are called bundled attributes. The choice between a bundled or an unbundled specification is made by setting a switch called the aspect source flag for each of these attributes

setIndividualASF(attributeptr, flagptr)

where parameter attributer ptr points to a list of attributes and parameter flagptr points to the corresponding list of aspect source flags. Each aspect source flag can be assigned a value of individual or bundled.

Bundled line attributes

Entries in the bundle table for line attributes on a specified workstation are set with the function

setPolylineRepresentation (ws, li, lt, lw, lc)

Parameter ws is the workstation identifier and line index parameter li defines the bundle table position. Parameter lt, lw, tc are then bundled and assigned values to set the line type, line width, and line color specifications for designated table index.

Example

setPolylineRepresentation(1,3,2,0.5,1)
setPolylineRepresentation (4,3,1,1,7)

A poly line that is assigned a table index value of 3 would be displayed using dashed lines at half thickness in a blue color on work station 1; while on workstation 4, this same index generates solid, standard-sized white lines

Bundle area fill Attributes

Table entries for bundled area-fill attributes are set with

setInteriorRepresentation (ws, fi, fs, pi, fc)

Which defines the attributes list corresponding to fill index fi on workstation ws. Parameter fs, pi and fc are assigned values for the fill style pattern index and fill color.

Bundled Text Attributes

setTextRepresentation (ws, ti, tf, tp, te, ts, tc)

bundles values for text font, precision expansion factor size an color in a table position for work station ws that is specified by value assigned to text index parameter ti.

Bundled marker Attributes

setPolymarkerRepresentation (ws, mi, mt, ms, mc)

That defines marker type marker scale factor marker color for index mi on workstation ws.

Inquiry functions

Current settings for attributes and other parameters as workstations types and status in the system lists can be retrieved with inquiry functions.

inquirePolylineIndex (lastli) and

inquireInteriorcColourIndex (lastfc)

Copy the current values for line index and fill color into parameter lastli and lastfc.

SUMMARY OF ATTRIBUTES

Output Primitive Type	*Associated Attributes*	*Attribute-Setting Functions*	*Bundled-Attribute Functions*
Line	Type	`setLinetype`	`setPolylineIndex`
	Width	`setLineWidthScaleFactor`	`setPolylineRepresentation`
	Color	`setPolylineColourIndex`	
Fill Area	Fill Style	`setInteriorStyle`	`setInteriorIndex`
	Fill Color	`setInteriorColorIndex`	`setInteriorRepresentation`
	Pattern	`setInteriorStyleIndex`	
		`setPatternRepresentation`	
		`setPatternSize`	
		`setPatternReferencePoint`	
Text	Font	`setTextFont`	`setTextIndex`
	Color	`setTextColourIndex`	`setTextRepresentation`
	Size	`setCharacterHeight`	
		`setCharacterExpansionFactor`	
	Orientation	`setCharacterUpVector`	
		`setTextPath`	
		`setTextAlignment`	
Marker	Type	`setMarkerType`	`setPolymarkerIndex`
	Size	`setMarkerSizeScaleFactor`	`setPolymarkerRepresentation`
	Color	`setPolymarkerColourIndex`	

Two Dimensional Geometric Transformations

Changes in orientations, size and shape are accomplished with geometric transformations that alter the coordinate description of objects.

Basic transformation

- Translation
  - T(t_x, t_y)
  - Translation distances
- Scale
  - S(s_x,s_y)
  - Scale factors
- Rotation
  - R(θ)
  - Rotation angle

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, tx, ty to original coordinate position (x,y) to move the point to a new position (x',y') to

x' = x + tx, y' = y + ty

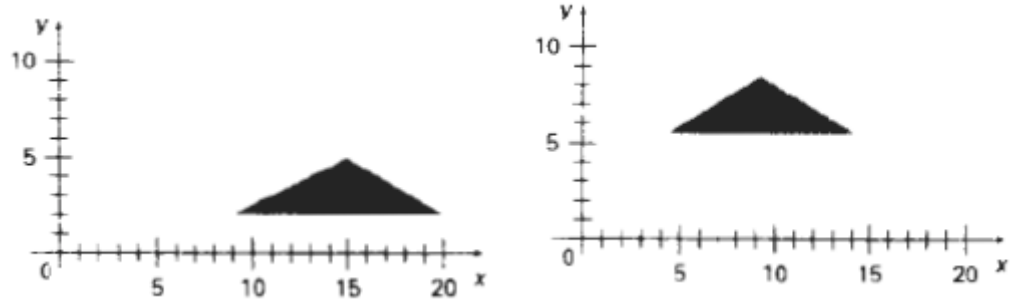
The translation distance point (tx,ty) is called translation vector or shift vector.

Translation equation can be expressed as single matrix equation by using column vectors to represent the coordinate position and the translation vector as

$$P = (x, y)$$
$$T = (t_x, t_y)$$

$$x' = x + t_x$$
$$y' = y + t_y$$

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

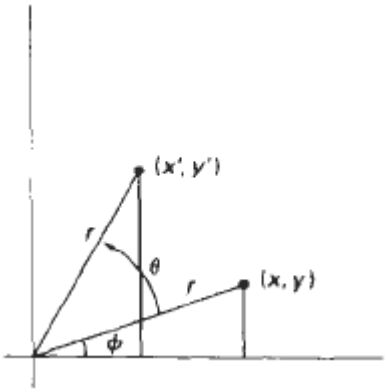


Moving a polygon from one position to another position with the translation vector (-5.5, 3.75)

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 θ 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 θ 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 Φ is the original angular of the point from horizontal and θ is the rotation angle.

The transformed coordinates in terms of angle θ and Φ

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

$$y' = r\sin(\theta+\Phi) = r\sin\theta\cos\Phi + r\cos\theta\sin\Phi$$

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 θ 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 counterclockwise rotations about the rotation point and negative values rotate objects in the clockwise.

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 Sx & Sy to produce the transformed coordinates (x',y')

$$\mathbf{x' = x.Sx} \qquad \mathbf{y' = y.Sy}$$

scaling factor Sx scales object in x direction while Sy scales in y direction.

The transformation equation in matrix form

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

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

Uniform scaling
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.

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 X 3 instead of 2 X 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

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

$$P' = T(t_x, t_y) \bullet P$$

For Scaling

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

$$P' = S(s_x, s_y) \bullet P$$

For rotation

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

$$P' = R(\theta) \bullet P$$

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 (tx1,ty1) and (tx2,ty2) are applied to a coordinate position P, the final transformed location P' is calculated as

P'=T(tx2,ty2).{T(tx1,ty1).P}

={T(tx2,ty2).T(tx1,ty1)}.P

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

$$\begin{bmatrix} 1 & 0 & tx2 \\ 0 & 1 & ty2 \\ 0 & 0 & 1 \end{bmatrix}.\begin{bmatrix} 1 & 0 & tx1 \\ 0 & 1 & ty1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & tx1 + tx2 \\ 0 & 1 & ty1 + ty2 \\ 0 & 0 & 1 \end{bmatrix}$$

Or

T(tx2,ty2).T(tx1,ty1) = T(tx1+tx2,ty1+ty2)

Which demonstrated the two successive translations are additive.

Rotations

Two successive rotations applied to point P produce the transformed position

P'=R(θ2).{R(θ1).P}={R(θ2).R(θ1)}.P

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

R(θ2).R(θ1) = R(θ1+ θ2)

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

P' = R(θ1+ θ2).P

$$\begin{bmatrix} \cos\theta 2 & -\sin\theta 2 & 0 \\ \sin\theta 2 & \cos\theta 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}.\begin{bmatrix} \cos\theta 1 & -\sin\theta 1 & 0 \\ \sin\theta 1 & \cos\theta 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta 2 + \theta 1) & -\sin(\theta 2 + \theta 1) & 0 \\ \sin(\theta 2 + \theta 1) & \cos(\theta 2 + \theta 1) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Scaling

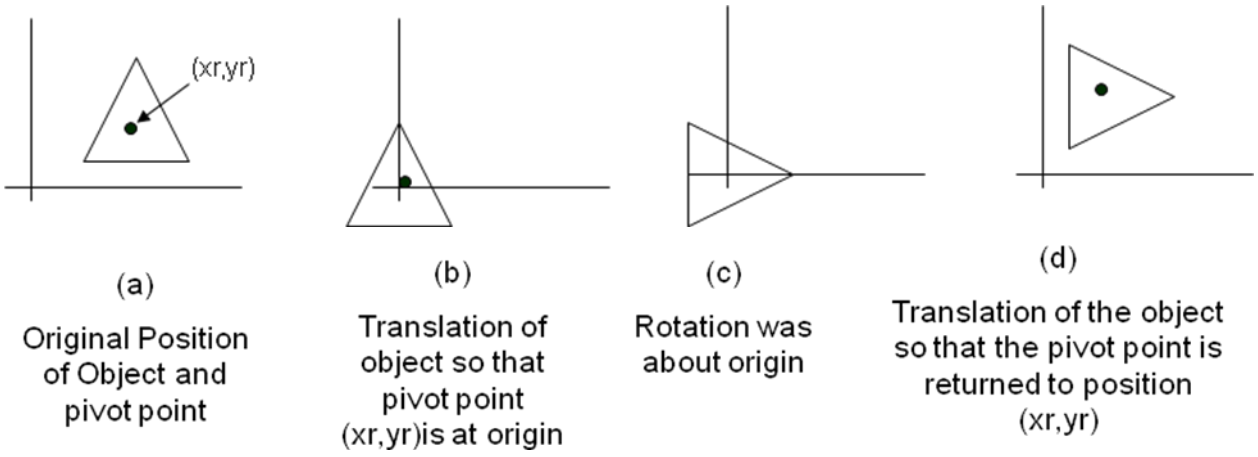
Concatenating transformation matrices for two successive scaling operations produces the following composite scaling matrix

$$\begin{bmatrix} sx2 & 0 & 0 \\ 0 & sy2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} sx1 & 0 & 0 \\ 0 & sy1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} sx2.sx1 & 0 & 0 \\ 0 & sy2.sy1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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 obtain with the concatenation

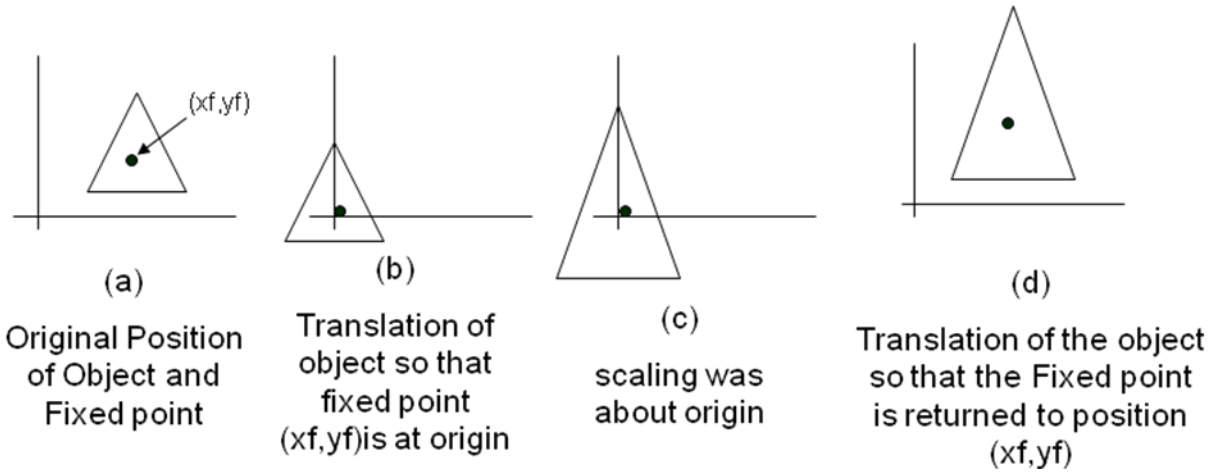
$$\begin{bmatrix} \cos\theta & -\sin\theta & xr(1-\cos\theta)+ yr\sin\theta \\ \sin\theta & \cos\theta & yr(1-\cos\theta) - xr\sin\theta \\ 0 & 0 & 1 \end{bmatrix}$$

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

Which can also be expressed as $T(xr,yr).R(\theta).T(-xr,-yr) = R(xr,yr,\theta)$

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 matix

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

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

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

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \left(\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\Theta & -\sin\Theta & 0 \\ \sin\Theta & \cos\Theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \right) \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

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[il[j] = (i == j);
}

/ * Multiplies  matrix a times b, putting result in b */
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[l][c] + a[r][2]*b[2][c]:
for (r = 0: r < 3: r++)
for Ic = 0; c < 3: c++)
b[r][c]=- tmp[r][c]:
}

void translate2 (int tx, int ty)
{
Matrix3x3 m:
rnatrix3x3SetIdentity (m) :
m[0][2] = tx;
m[1][2] = ty:
matrix3x3PreMultiply (m, theMatrix);
}
vold scale2 (float sx. float sy, wcPt2 refpt)
(
Matrix3x3 m.
matrix3x3SetIdentity (m);
m[0] [0] = sx;
m[0][2] = (1 - sx)* refpt.x;
```

```
m[l][l] = sy;
m[10][2] = (1 - sy)* refpt.y;
matrix3x3PreMultiply (m, theMatrix);
}

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

void transformPoints2 (int npts, wcPt2 *pts)
{
int k:
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);

}
```

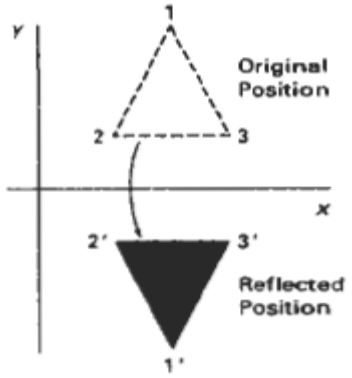
Other Transformations

1. Reflection
2. Shear

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^o about the reflection axis. We can choose an axis of reflection in the xy plane or perpendicular to the xy plane or coordinate origin

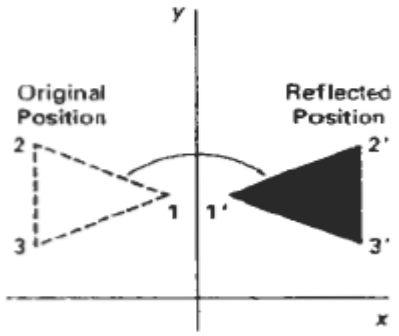
Reflection of an object about the x axis



Reflection the x axis is accomplished with the transformation matrix

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

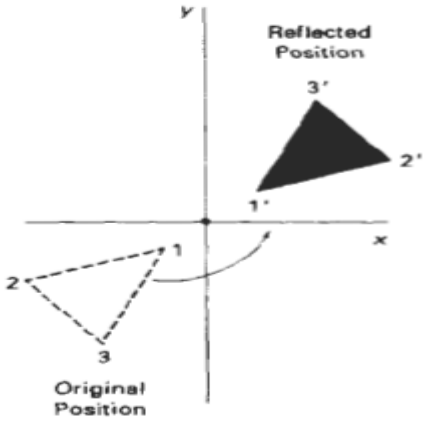
Reflection of an object about the y axis



Reflection the y axis is accomplished with the transformation matrix

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

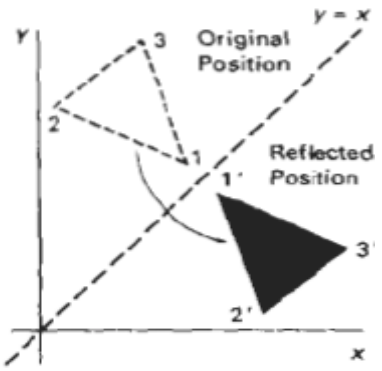
Reflection of an object about the coordinate origin



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



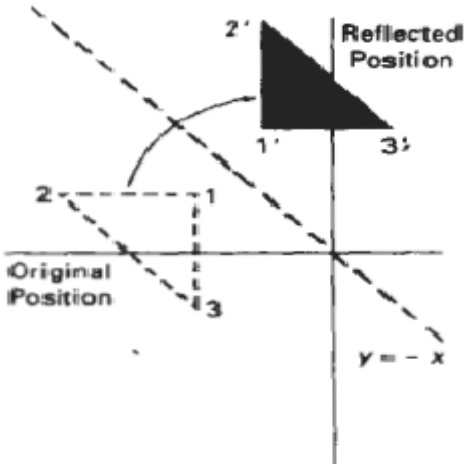
To obtain transformation matrix for reflection about diagonal y=x the transformation sequence is

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=x 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

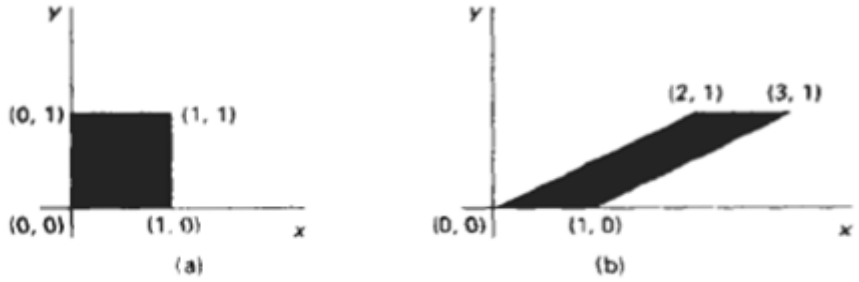
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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



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

$x' = x + sh_x . y$

$y' = y$

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

$x' = x$

$y' = y + sh_y . x$

XY-Shear

The transformation matrix for xy-shear

$$w \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$ rdinates as

$x' = x + sh_x . y$

$y' = y + sh_y . x$

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

$$\begin{bmatrix} 1 & sh_x & -sh_x.y_{ref} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

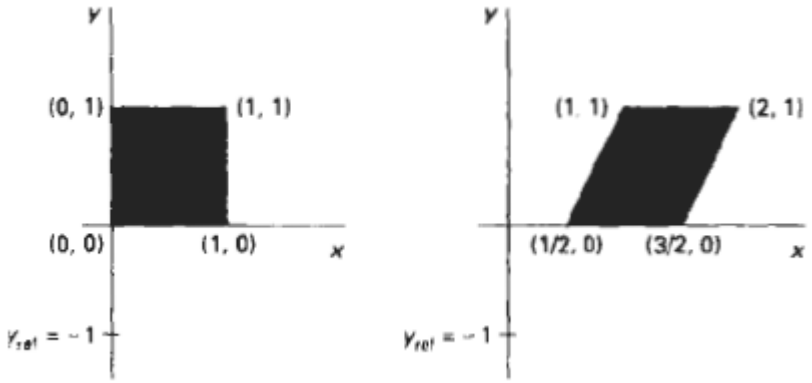
which transforms the coordinates as

$x' = x + sh_x(y - y_{ref})$

$y' = y$

Example

$Sh_x = ½$ and $y_{ref} = -1$



Y shear with x reference line

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

$$\begin{bmatrix} 1 & sh_x & -sh_x.y_{ref} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

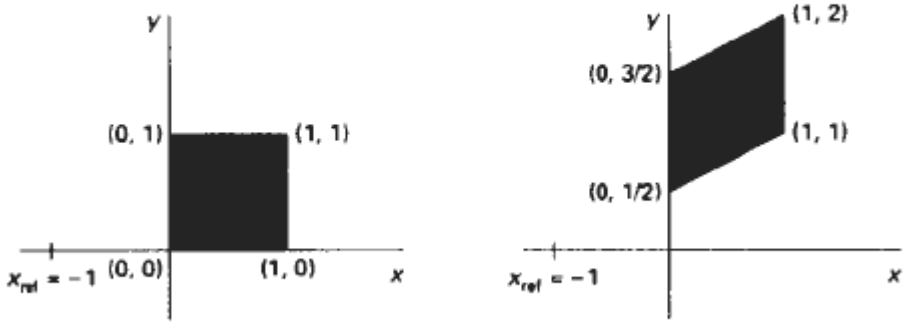
which transforms the coordinates as

$x' = x$

$y' = sh_y (x - x_{ref}) + y$

Example

$Sh_y = ½$ and $x_{ref}=-1$



Two dimensional viewing

A world coordinate area selected for display is called a **window**

An area on a display device to which a window a mapped is called a **view port**

The **window** defines **what** is to be viewed

The **viewport** defines **where** it is to be displayed

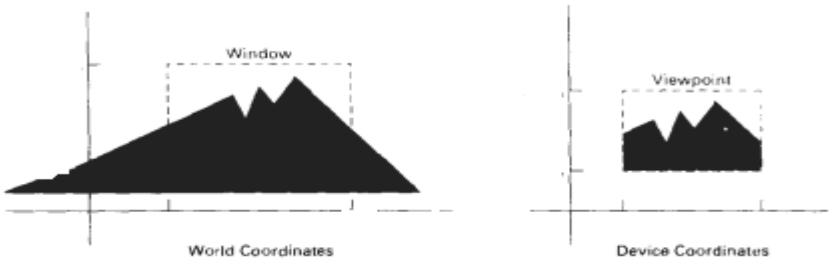
The mapping of a part of a world coordinate scene to device coordinates is referred to as a **viewing transformation**

The viewing pipeline

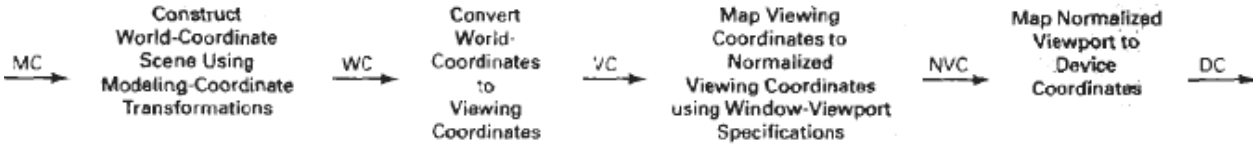
A world coordinate area selected for display is called a window. An area on a display device to which a window is mapped is called a view port. The window defines what is to be viewed the view port defines where it is to be displayed.

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

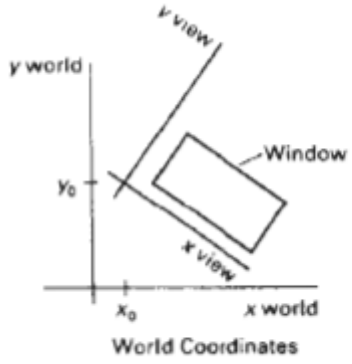


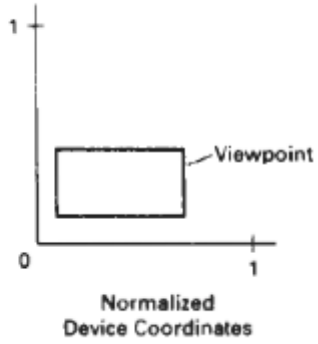
The viewing transformation in several steps, as indicated in Fig. First, we construct the scene in world coordinates using the output primitives. Next to obtain a particular orientation for the window, we can set up a two-dimensional viewing-coordinate system in the world coordinate plane, and define 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.

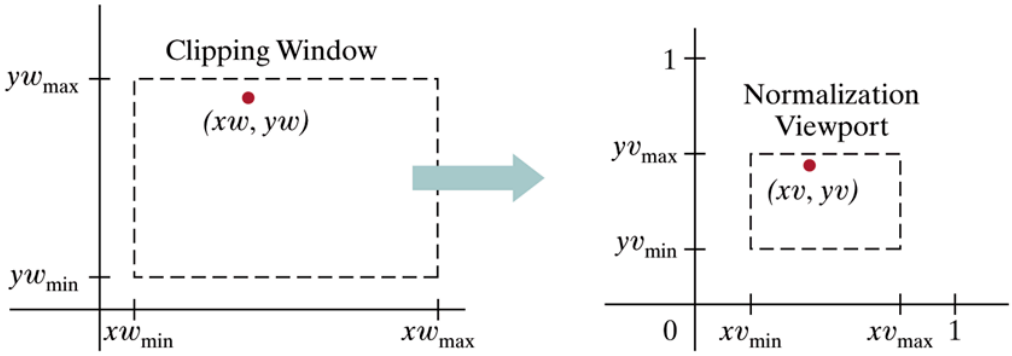
We then define a viewport in normalized coordinates (in the range from 0 to 1) and map the viewing-coordinate description of the scene to normalized coordinates.

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 (xw, yw) in a world-coordinate clipping window is mapped to viewport coordinates (xv, yv), 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

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

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

solving these expressions for view port position (x_v, y_v)

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

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

where scaling factors are

$$sx = \frac{xv_{max} - xv_{min}}{xw_{max} - xw_{min}} \qquad sy = \frac{yv_{max} - yv_{min}}{yw_{max} - yw_{min}}$$

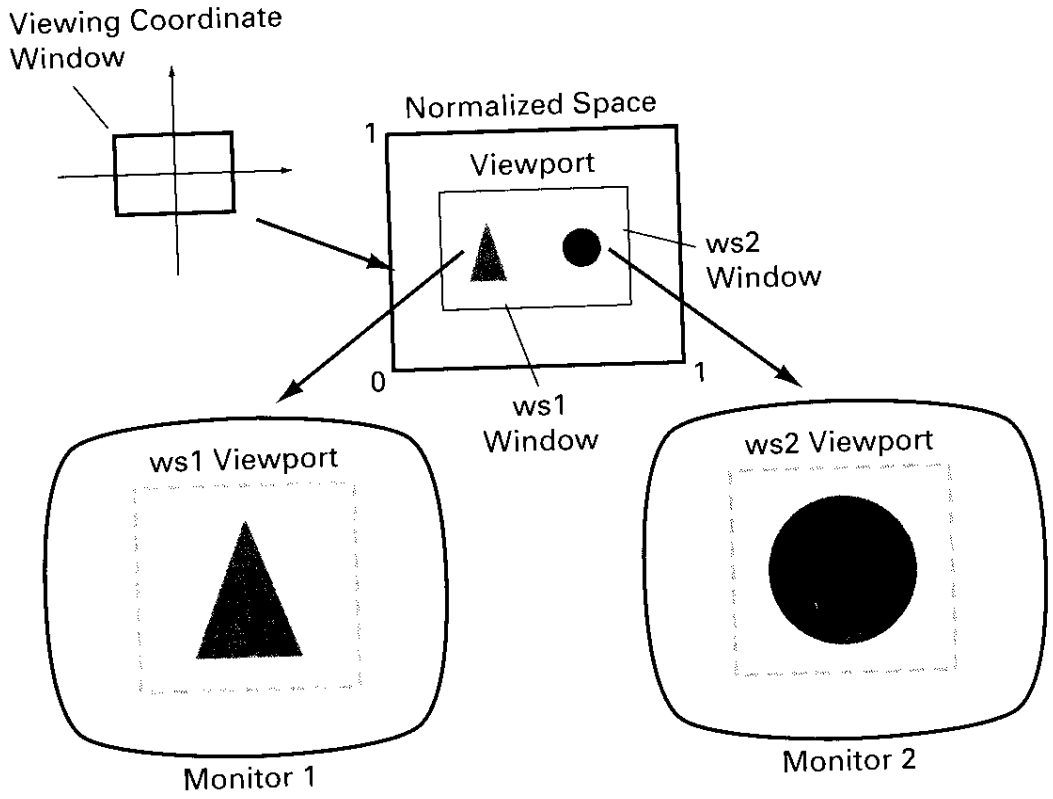
The conversion is performed with the following sequence of transformations.

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.

evaluateViewOrientationMatrix(x_0,y_0,x_v,y_v,error, viewMatrix)

where x_0,y_0 are coordinate of viewing origin and parameter x_v, y_v are the world coordinate positions for view up vector.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

evaluateViewMappingMatrix ($x_w$min, $x_w$max, $y_w$min, $y_w$max, $x_v$min, $x_v$max, $y_v$min, $y_v$max, error, viewMappingMatrix)

Here window limits in viewing coordinates are chosen with parameters $x_w$min, $x_w$max, $y_w$min, $y_w$max and the viewport limits are set with normalized coordinate positions $x_v$min, $x_v$max, $y_v$min, $y_v$max.
The combinations of viewing and window view port mapping for various workstations in a viewing table with

setViewRepresentation(ws,viewIndex,viewMatrix,viewMappingMatrix,
xclipmin, xclipmax, yclipmin, yclipmax, clipxy)

Where parameter ws designates the output device and parameter view index sets an integer identifier for this window-view port point. The matrices viewMatrix and viewMappingMatrix can be concatenated and referenced by 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 work station 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.

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:

$$xw_{min} <= x <= xw_{max}$$

$$yw_{min} <= y <= yw_{max}$$

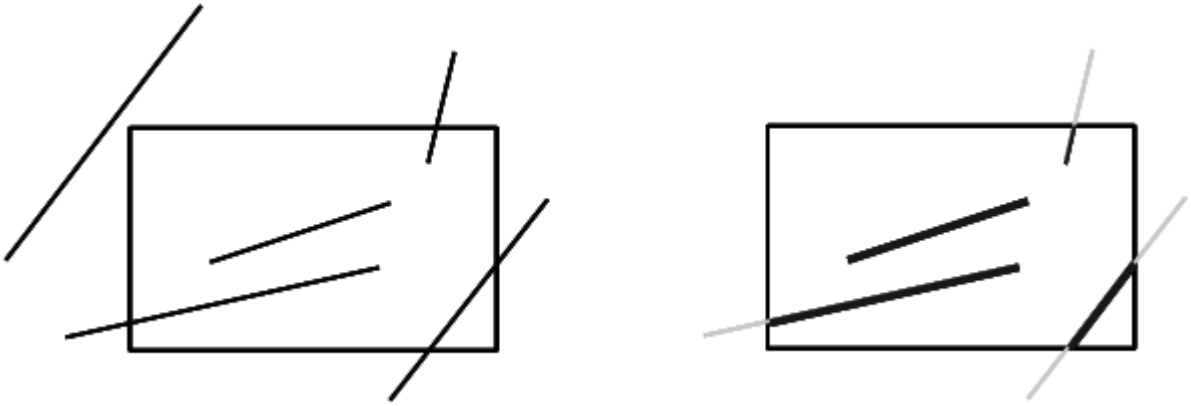
where the edges of the clip window $(xw_{min}, xw_{max}, yw_{min}, yw_{max})$ can be either the world-coordinate window boundaries or viewport boundaries. If any one of these four inequalities is not satisfied, the point is clipped (not saved for display).

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 can not 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 P3P4 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

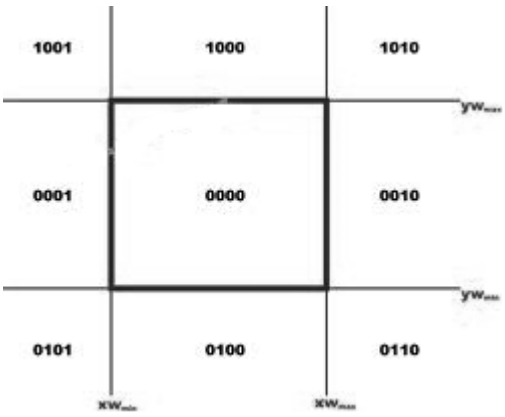
$$x = x_1 + u(x_2 - x_1),$$
$$y = y_1 + u(y_2 - y_1), \quad 0 \le u \le 1$$

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.

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.

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

bit4: 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 $x < xw_{min}$.

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 $x - xw_{min}$

bit 2 is the sign bit of $xw_{max} - x$

bit 3 is the sign bit of $y - yw_{min}$

bit 4 is the sign bit of $yw_{max} - y$.

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.

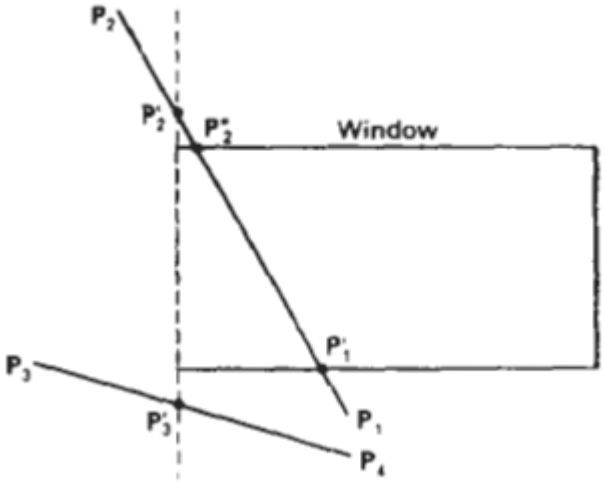
Any lines that are completely contained within the window boundaries have a region code of 0000 for both endpoints, and we accept

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.



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 $\mathbf{P_1}$ to $\mathbf{P_2}$, 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 xw_{min} or to xw_{max} and slope of line is calculated as

$$m = (y_2 - y_1) / (x_2 - x_1)$$

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

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

with y set to either to yw_{min} or to yw_{max}.

Implementation of Cohen-sutherland Line Clipping

```
#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));
}
```

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 :

$$x = x_1 + u\,\Delta x$$
$$y = y_1 + u\,\Delta y \qquad 0<=u<=1$$

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 :

$$xw_{min} <= x_1 + u\,\Delta x <=. xw_{max}$$

$$yw_{min} <= y_1 + u\,\Delta y <= yw_{max}$$

Each of these four inequalities can be expressed as

$$\mu p_k <= q_k. \qquad k=1,2,3,4$$

the parameters p & q are defined as

$p_1 = -\Delta x$ $q_1 = x_1 - xw_{min}$
$p_2 = \Delta x$ $q_2 = xw_{max} - x_1$
$P_3 = -\Delta y$ $q_3 = y_1 - yw_{min}$
$P_4 = \Delta y$ $q_4 = yw_{max} - y_1$

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 out side 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

$$u = qk / pk$$

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 u1 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

$$r_k = qk / pk$$

The value of u1 is taken as largest of set consisting of 0 and various values of r. The value of u2 is determined by examining the boundaries for which lines proceeds from inside to outside (P>0).

A value of $r_k$is calculated for each of these boundaries and value of u2 is the minimum of the set consisting of 1 and the calculated r values.

If u1>u2, the line is completely outside the clip window and it can be rejected.

Line intersection parameters are initialized to values u1=0 and u2=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 u1.

When p>0, the parameter r is used to update u2.

If updating u1 or u2 results in u1>u2 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 u1 and u2.

The Liang-Barsky algorithm is more efficient than the Cohen-Sutherland algorithm since intersections calculations are reduced. Each update of parameters u1 and u2 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

```
#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));
}
}
}
```

Nicholl-Lee-Nicholl Line clipping

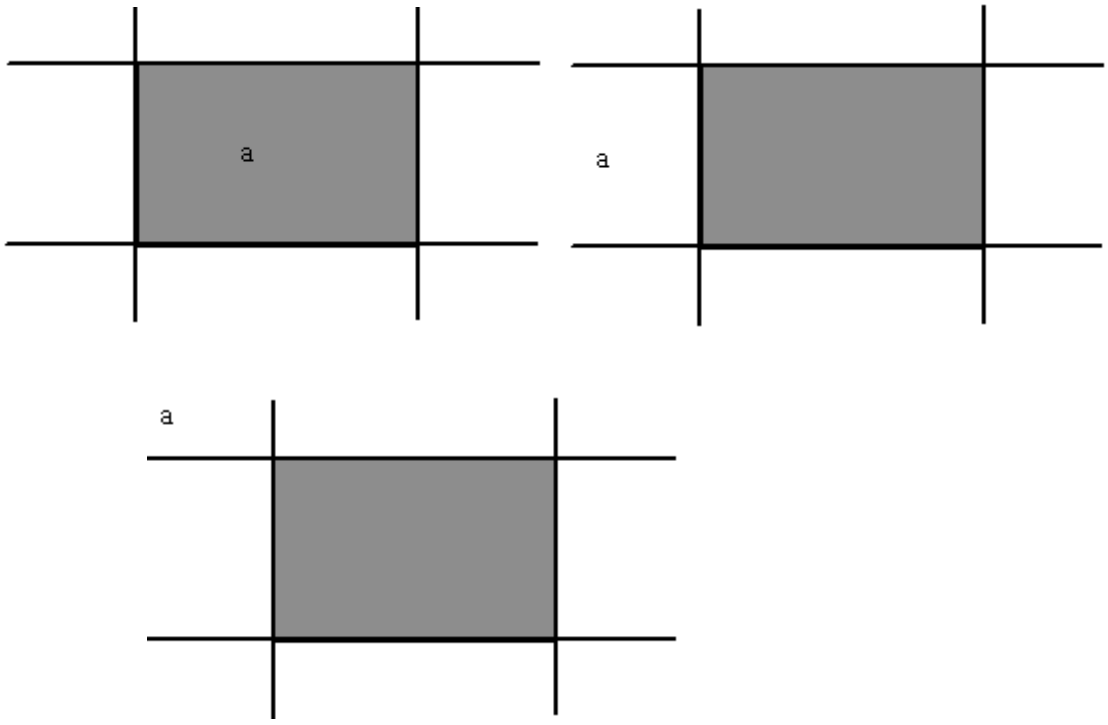
By creating more regions around the clip window, the Nicholl-Lee-Nicholl (or NLN) algorithm avoids multiple clipping of an individual line segment. In the Cohen-Sutherland method, multiple intersections may be calculated.These extra intersection

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 dipping, 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



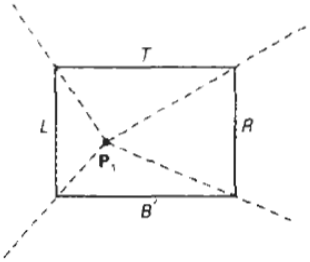
Case 1: p1 inside region

Case 2: p1 across edge

Case 3: p1 across corner

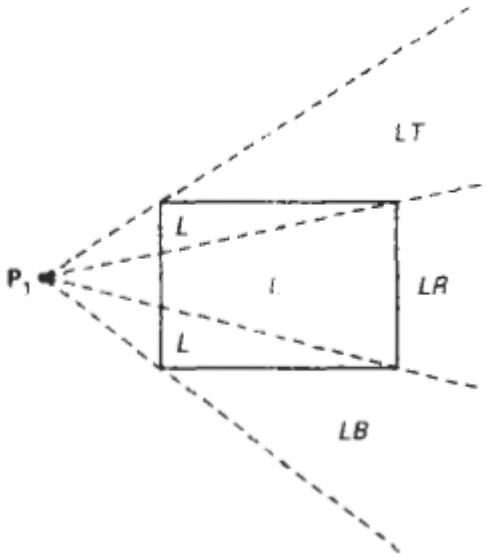
Next, we determine the position of P2 relative to P1**.** To do this, we create some new regions in the plane, depending on the location of P1. Boundaries of the new regions are half-infinite line segments that start at the position of P1 and pass through the window corners. If P1 is inside the clip window and P2 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 P2. If both P1 and P2 are inside the clipping rectangle, we simply save the entire line.

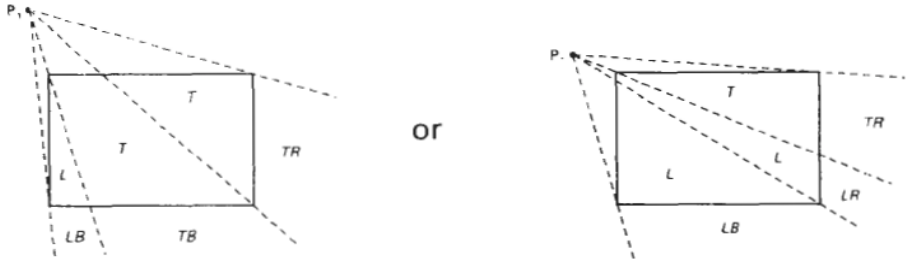
If P1 is in the region to the left of the window, we set up the four regions, ***L, LT, LR,*** and ***LB,*** shown in Fig.



These four regions determine a unique boundary for the line segment. For instance, if P2 is in region L, we clip the line at the left boundary and save the line segment from this intersection point to P2**.** But if P2 is in region **LT,** we save the line segment from the left window boundary to the top boundary. If P2 is not in any of the four regions, L, LT**,** LR, or LB**,** the entire line is clipped.

For the third case, when P1 is to the left and above the clip window, we usethe clipping regions in Fig.

Fig : The two possible sets of clipping regions used in NLN algorithm when P1 is above and to the left of the clip window



In this case, we have the two possibilities shown, depending on the position of P1, relative to the top left corner of the window. If P2**,** 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 P2 is located, we compare the slope of the

line to the slopes of the boundaries of the clip regions. For example, if P1 is left of

the clipping rectangle (Fig. a), then P2**,** is in region LT if

$\text{slope}\overline{P_1P_{TR}} < \text{slope}\overline{P_1P_2} < \text{slope}P_1P_{TL}$

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 $\mathbf{x = x_{L,,}}$ with

$u= (x_L - x_1)/ (x_2 - x_1)$ so that the y-intersection position is

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1} (x_L - x_1)$$

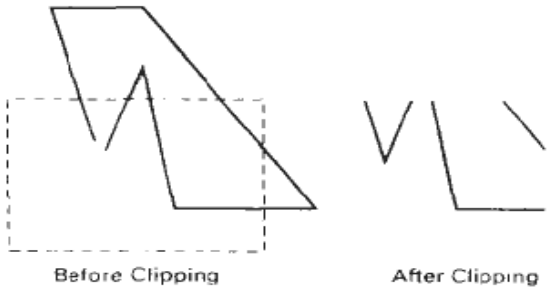
And an intersection position on the top boundary has $y = \mathbf{y_T}$ and $u = (y_T - y_1)/ (y_2 - y_1)$ with

$$x = x_1 + \frac{x_2 - x_1}{y_2 - y_1} (y_T - y_1)$$

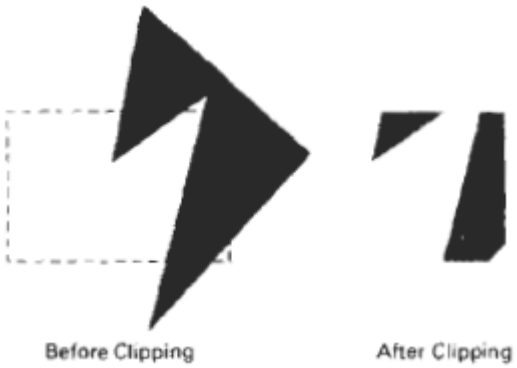
POLYGON CLIPPING

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 (Fig.), 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.



Sutherland – Hodgeman polygon clipping:

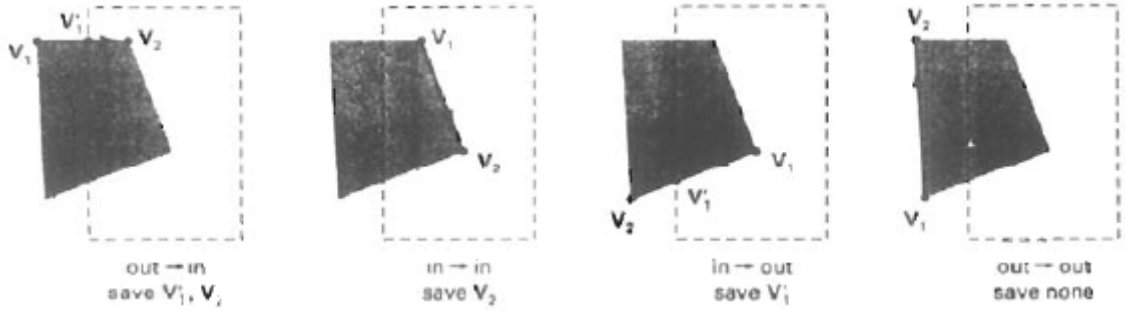
A polygon can be clipped by processing the polygon boundary as a whole against each window edge. This could be accomplished by processing all polygon vertices against each clip rectangle boundary.

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:

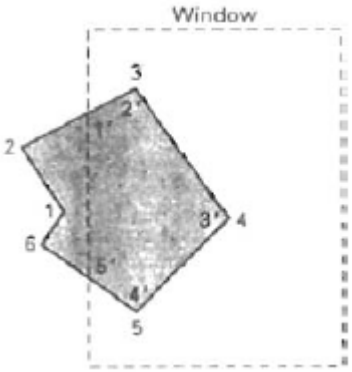
1. If the first vertex is outside the window boundary and second vertex is inside, both the intersection point of the polygon edge with window boundary and second vertex are added to output vertex list.
2. If both input vertices are inside the window boundary, only the second vertex is added to the output 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.

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



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.

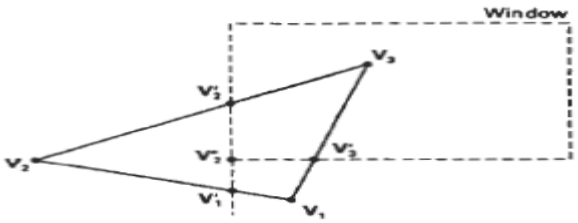


vertices 1 and 2 are found to be on outside of boundary. Moving along vertex 3 which is inside, calculate the intersection and save both the intersection point and vertex 3. Vertex 4 and 5 are determined to be inside and are saved. Vertex 6 is outside so we find and save the intersection point. Using the five saved points we repeat the process for next window boundary.

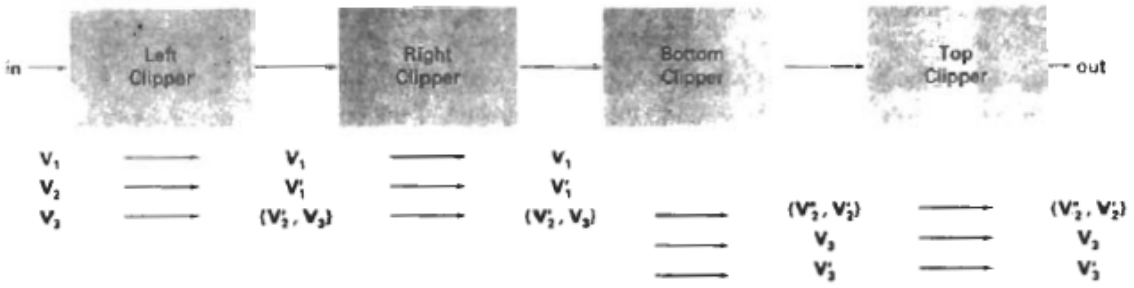
Implementing the algorithm as described requires setting up storage for an output list of vertices as a polygon clipped against each window boundary. We eliminate the intermediate output vertex lists by simply by clipping individual vertices at each step and passing the clipped vertices on to the next 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



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)
{
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
{
pout[*cnt]=ipt;
(*cnt)++;
}
}
s[b]=p;
if(inside(p,b,wmin,wmax))
if(b<top)
clippoint(p,b+1,wmin,wmax,pout,cnt,first,s);
else
{
pout[*cnt]=p;
(*cnt)++;
}
}
void closeclip(dcPt wmin,dcPt wmax, wcPt2  *pout,int *cnt,wcPt2 *first[], wcPt2 *s)
{
wcPt2 iPt;
Edge b;
for(b=left;b<=top;b++)
{
if(cross(s[b],*first[b],b,wmin,wmax))
```

```
{
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);
}
```

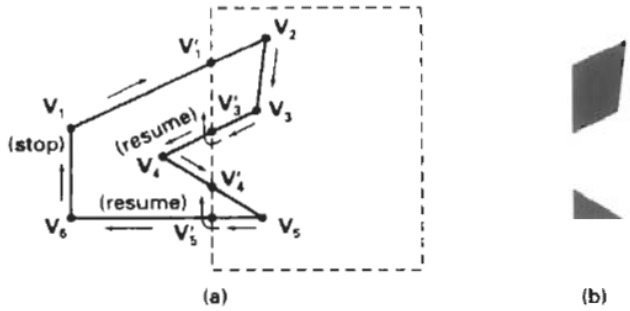
Weiler- Atherton Polygon Clipping

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 (clockwise or counterclockwise) 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.



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.

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.

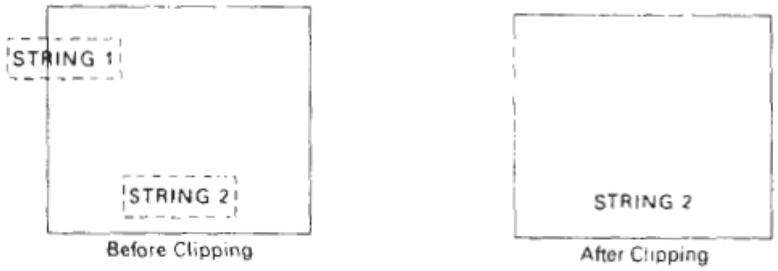
Clipping a filled circle

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.

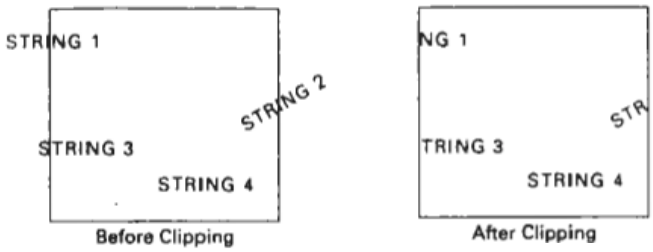
The simplest method for processing character strings relative to a window boundary is to use the **all-or-none string-clipping** strategy shown in Fig. . If all of the string is inside a clip window, we keep it. Otherwise, the string is discarded. This procedure is implemented by considering a bounding 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



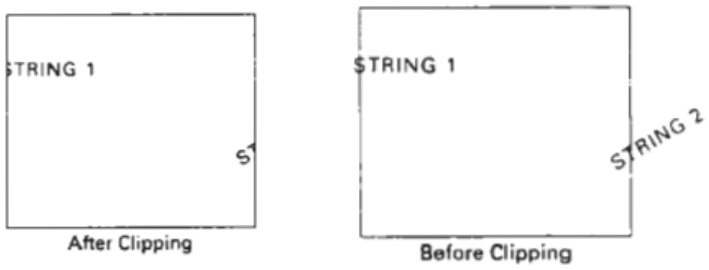
An alternative to rejecting an entire character string that overlaps a window boundary is to use the **all-or-none character-clipping** strategy. Here we discard only those characters that are not completely inside the window .In this case, the boundary limits of individual characters are compared to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



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.