# COMPUTER GRAPHICS
## UNIT –I

### OUTPUT PRIMITIVES: POINTS AND LINES

Graphics programming packages provide functions to describe a scene in terms of these basic geometric structures, referred to as ***Output Primitives***, and to group sets of output primitives into more complex structures.

Each output primitive is specified with input coordinate data and other information about the way that objects is to be displayed.

Points and Straight Line segments are the simplest geometric components of pictures. Additional output primitives that can be used to construct a picture include circles and other conic sections, quadric surfaces, spline curves and surfaces, polygon color areas, and character strings.

**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.

A Random-Scan (Vector) System stores point-plotting instructions in the display list, and coordinate values in these instructions are converted to deflection voltages that position the electron beam at the screen locations to be plotted during each refresh cycle.

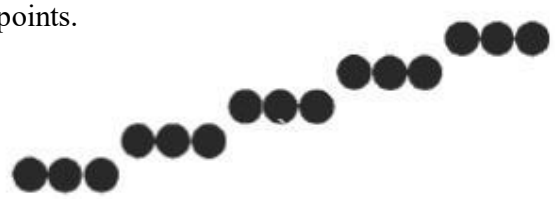
For a black-and-white raster system, a point is plotted by setting the bit value corresponding to a specified screen position within the frame buffer to 1. Then, as the electron beam sweeps across each horizontal scan line, it emits a burst of electrons (plots a point) whenever a value of 1 is encountered in the frame buffer.

With an RGB system, the frame buffer is loaded with the color codes for the intensities that are to be displayed at the screen pixel positions.

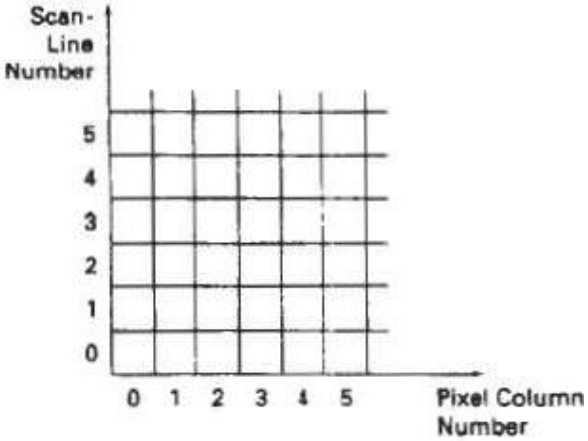
Line drawing is accomplished by calculating intermediate positions along the line path between two specified endpoint positions. An output device is then directed to fill in these positions between the endpoints.

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. Screen locations are referenced with integer values, so plotted positions may only approximate actual Line positions between two specified endpoints.

*For example,* a computed line position is (10.48, 20.51), it is rounded to (10, 21). This rounding of coordinate values to integers causes lines to be displayed with a stairstep appearance ("the jaggies"), as represented below. This stairstep shape is noticeable in low resolution systems.



For the raster-graphics device-level algorithms, object positions are specified directly in integer device coordinates.




To load a specified color into the frame buffer at a position corresponding to column ***x*** along scan line ***y***, we will assume we have available a low-level procedure of the form

```
setPixel (x, y)
```

Sometimes we want to retrieve the current frame-buffer intensity setting for a specified location. We accomplish this with the low-level function. We use,

```
getPixel (x, y)
```

## LINE-DRAWING ALGORITHMS

The Cartesian slope-intercept equation for a straight line is

$$y = mx + b \longrightarrow (1)$$

with **m** representing the 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 $(x_2, y_2)$.

We can determine the slope m and y intercept b with the following calculations:

$$m = \frac{(y_2 - y_1)}{(x_2 - x_1)} \longrightarrow (2)$$

$$b = y_1 - m \times x_1 \longrightarrow (3)$$

Algorithms for displaying straight lines are based on the line equations (1) and the calculations given in equations (2) and (3).

For any given x interval Δx (X2-X1) along a line, we can compute the corresponding y interval Δy (Y2-Y1)from equation (2) as,

$$y = m\Delta x \longrightarrow (4)$$

Similarly, we can obtain the x interval Δx corresponding to a specified Δy as

$$\Delta x = \frac{\Delta y}{m} \longrightarrow (5)$$

These equations form the basis for determining deflection voltages in analog devices.

For lines with slope magnitudes $|m| < 1$, $\Delta x$ can be set proportional to a small horizontal deflection voltage and the corresponding vertical deflection is then set proportional to $\Delta y$ as calculated from Equation 4.

For lines whose slopes have magnitudes $|m| > 1$, $\Delta y$ can be set proportional to a small vertical deflection voltage with the corresponding horizontal deflection voltage set proportional to $\Delta x$, calculated from Equation 5.

For lines with $m = 1$, $\Delta x = \Delta y$ and the horizontal and vertical deflections voltages are equal. In each case, a smooth line with slope m is generated between the specified endpoints.

### DDA algorithm:

The **Digital Differential Analyzer (DDA)** is a Scan-Conversion line algorithm based calculating either Δy or Δx using equations (4) and (5).

Consider first a line with **positive slope, less than or equal to 1**, we sample at unit intervals (Δx=1) and compute each successive y value as

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

subscript k takes integer values starting from 1, for the first point, and increases by 1 until the final endpoints is reached.

Since m can be any real number between 0 & 1, the calculated y values must be rounded to the nearest integer.

For lines with a **positive slope greater than 1**, we reserve the roles of x & y. That is, we sample at unit y intervals (Δy=1) and calculate each succeeding x value as

$$x_{k+1} = x_k + \frac{1}{m} \longrightarrow (7)$$

Equations (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, so that the starting endpoint is that, then either we have Δx= -1 and

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

or (when the slope is greater than 1) we have△y = -1 with

$$x_{k+1} = x_k - \frac{1}{m} \longrightarrow (9)$$

Equations (6), (7), (8) and (9) can also be used to calculate pixel positions along a line with negative slope. 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 equation (6).

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

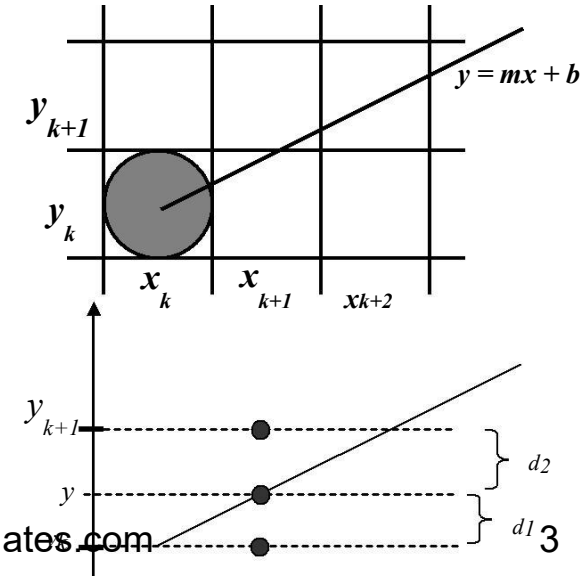
```
# define ROUND (a) ((int) (a+0.5))
void lineDDA (int xa, int ya, int xb, int yb)
{
     int dx = xb - xa, dy = yb - ya, steps, k;
     float xIncrement, yIncrement, x = xa, y =
     ya; if (abs (dx) > abs (dy))
          steps = abs (dx) ;
     else
          steps = abs (dy);
     xIncrement = dx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

The DDA algorithm is a faster method for calculating pixel position than the direct use of equation (1). We can improve the performance of the DDA algorithm by separating the increments m and 1/m into integer and fractional parts so that all calculations are reduced to integer operations.

**Bresenham's Line Drawing Algorithm.**

An accurate and efficient raster line-generating algorithm, developed by Bresenham, scans converts lines using only incremental integer calculations that can be adapted to display circles and other curves.

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 line, we step to each successive column (x position) and plot the pixel whose scan-line y value is closest to the line path.




The above figure demonstrates the $k^{th}$ step in this process. Assuming we have determined that the pixel at $(x_k, y_k)$ is to be displayed, we next need to decide which pixel to plot in column $x_{k+1}$. Our choices are the pixels at positions

$(x_k+1, y_k)$ 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 $d_2$ shown in the diagram. The y coordinate on the mathematical line at pixel column position $x_k+1$ is calculated as

$$y = m(x_k + 1) + b \longrightarrow (1)$$

Then

$$d_1 = y - y_k$$
$$= m(x_k + 1) + b - y_k$$

and

$$d_2 = (y_k+1) - y$$
$$= y_k + 1 - m(x_k + 1) - b$$

The difference between these two separations is

$$d_1 - d_2 = 2m(x_k + 1) - 2y_k + 2b - 1 \longrightarrow (2)$$

A decision parameter $p_k$ for the kth step in the line algorithm can be obtained by rearranging equation (2) so that it involves only integer calculations. We accomplish this by substituting $\mathbf{m = \Delta y / \Delta x}$, where $\Delta y$ and $\Delta x$ are the vertical and horizontal separations of the endpoint positions, and defining:

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

The sign of $p_k$ is the same as sign of $d_1$-$d_2$, since $\Delta x > 0$ for our examples. Parameter c is constant and has the value $2\Delta y + \Delta x$ (2b-1), which is independent of 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 $y_k+1$(i.e., $d_1<d_2$), then decision parameter $p_k$ is negative. In that case, we plot the lower pixel; otherwise, we plot the upper pixel.

Coordinate changes along the line occur in unit steps in either the x or y directions. Therefore, we can obtain the values of successive decision parameters using incremental integer calculations. At step k+1, the decision parameter is evaluated from equation (3) as

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

Substituting equation (3) from the preceding equation, we have

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

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

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1}-y_k) \longrightarrow (4)$$

where the term $y_{k+1} - y_k$ is either 0 or 1, depending on the sign 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 3 at the starting pixel position $(x_0,y_0)$ and with m evaluated as $\Delta y / \Delta x$:

$$p_0 = 2\Delta y - \Delta x \longrightarrow (5)$$

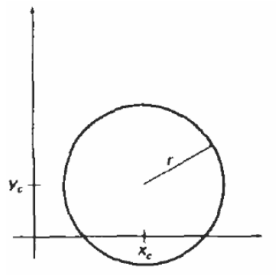
We can summarize Bresenham line drawing for a line with a positive slope less than 1 in the following listed steps:

1. *Input 2 endpoints, store left endpoint in* $(x_0, y_0)$.
2. *Load* $(x_0, y_0)$ *into frame buffer, i.e. plot the first point.*
3. *Calculate constants* $\Delta x, \Delta y, 2\Delta y, 2\Delta y - 2\,\Delta x$, *and initial value of decision parameter:* $p_0 = 2\Delta y - \Delta x$
4. *At each* $x_k$ *along the line, start at* $k=0$,
   *test: if* $p_k < 0$, *plot* $(x_{k+1}, y_k)$ *and*
   $$p_{k+1} = p_k + 2\Delta y$$
   *else plot* $(x_{k+1}, y_{k+1})$ *and*
   $$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$
5. *Repeat step (4)* $\Delta x$ *times.*

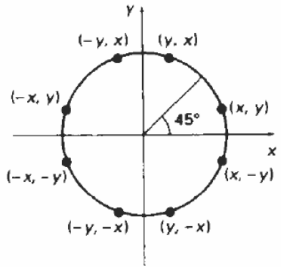
**CIRCLE –GENERATING ALGORITHMS:**
In general, a single procedure can be provided to display either ***Circular*** or ***Elliptical Curves*.**

**Properties of Circles:**
A Circle is defined as the set of points that are all at a given distance r from a center position ($x_c$, $y_c$). This distance relationship is expressed by the Pythagorean theorem in Cartesian coordinates as,

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

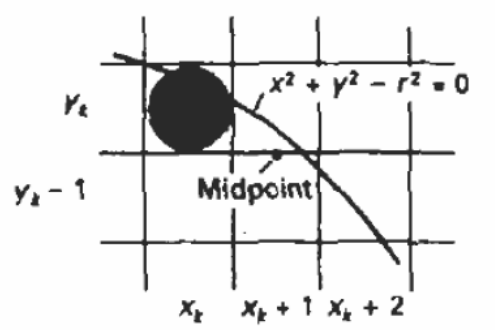



symmetry of circles. The shape of the circle is similar in each quadrant. These symmetry conditions are illustrated in the above diagram, where a point at position ***( x , y)*** on a one eighth circle sector is mapped into the seven circle points inthe other octants of the **xy** plane. Taking advantage of the circle symmetry in this way we can generate all pixel positions around a circle by calculating only the points within the sector from ***x=0 to x=y.***




**Midpoint Circle Algorithm:**
For a given radius ***r*** and screen center position ***($x_c$, $y_c$)*** we can first set up our algorithm to calculate pixel positions around a circle path centered at the coordinate origin ***(0, 0)***.
Then each calculated position ***(x, y)*** is moved to its proper screen position by adding $x_c$ to **x** and $y_c$ to **y**.
Along the circle section from ***x = 0*** to ***x = y*** in the first quadrant, the slope of the curve varies from ***0*** to ***-1***.
Positions in the other seven octants are then obtained by symmetry.
To apply the midpoint method, we define a circle function:
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.
To summarize, the relative position of any point ***(x, y)*** can be determined by checking the sign of the circle function:

$$f_{circle}(x, y) \begin{cases} < 0, & \text{if } (x, y) \text{ is inside the circle boundary} \\ = 0, & \text{if } (x, y) \text{ is on the circle boundary} \\ > 0, & \text{if } (x, y) \text{ is outside the circle boundary} \end{cases} \longrightarrow (5)$$




The diagram shows the midpoint between the two candidate pixels at Sampling position $x_k+1$.
Assuming we have just plotted the pixel at $(x_k, y_k)$, we 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 closer to the circle.
Decision Parameter is the circle function (4) evaluated at the midpoint between these two pixels:

$$p_k = f_{\text{circle}}\left(x_k + 1, y_k - \frac{1}{2}\right)$$

$$= (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2$$

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 midposition is outside or on the circle boundary, and we select the pixel on scanline $y_k - 1$.
Successive decision parameters are obtained using incremental calculations. We can summarize the steps in the Midpoint algorithm as follows:
1. Input radius **r** and circle center $(x_c, y_c)$, and obtain the first point on the circumference of a circle centered on the origin as

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

2. Calculate the initial value of the decision parameter as

$$p_0 = \frac{5}{4} - r$$

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

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1} \quad \text{where } 2x_{k+1} = 2x_k + 2 \text{ and } 2y_{k+1} = 2y_k - 2.$$

4. Determine symmetry points in the other seven octants.
5. Move each calculated pixel position **(x, y)** onto the circular path centered on **($x_c$, $y_c$)** and plot the coordinate values:

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

6. Repeat steps 3 through 5 until $x \geq y$.

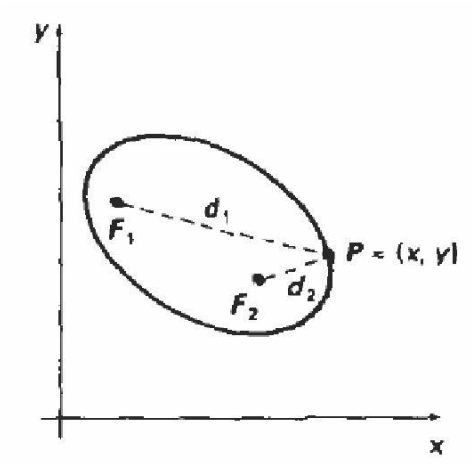
**ELLIPSE – GENERATING ALGORITHM:**

An Ellipse is an elongated circle. So, ***Elliptical Curves*** can be generated by modifying circledrawing procedures to take into account the different dimensions along the major and minor axes.

**Properties of Ellipses**

An ***Ellipse*** is defined as the set of points such that the sum of the distances from two fixed positions(foci) is the same for all points. f the distances to the two foci from any point ***P = (x, y)*** on the ellipse are labeled $d_1$ and $d_2$, then the general equation of an ellipse can be stated as,

$$d_1 + d_2 = constant \longrightarrow (1)$$




Expressing distances $d_1$ and $d_2$ in terms of the focal coordinates $F_1 = (x_1, y_1)$ and $F_2 = (x_2, y_2)$, we have

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

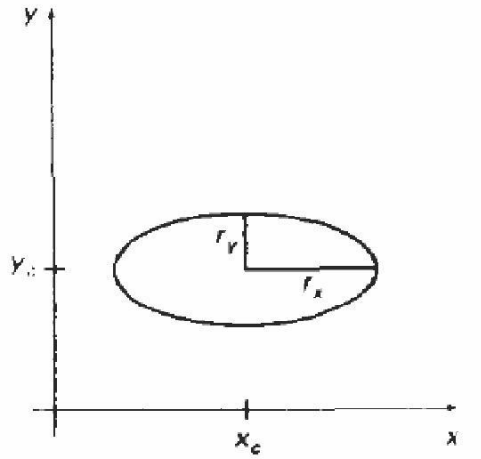
We can rewrite the general ellipse equation in the form,

$$Ax^2 + By^2 + Cxy + Dx + Ey + F = 0 \longrightarrow (3)$$

where 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.

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

In the diagram, we show an ellipse in "standard position" with major and minor axes oriented parallel to the ***x*** and y axes.



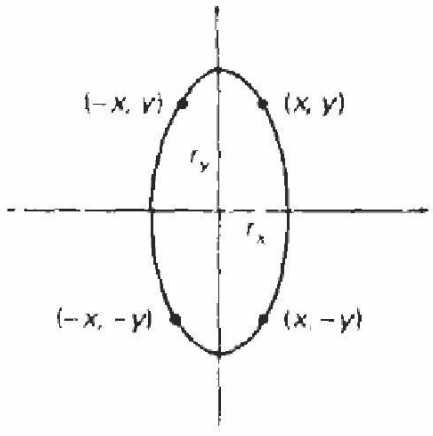

Parameter $r_x$ labels the semimajor axis, and parameter $r_y$ labels the semiminor axis. Using this the equation of the ellipse can be written as,

$$\frac{(x - x_c)^2}{r_x} + \frac{(y - y_c)^2}{r_y} = 1 \longrightarrow (4)$$

Using polar coordinates **r** and $\theta$, we can also describe the ellipse in standard position with the parametric equations:

$$x = x_c + r_x \cos\vartheta$$
$$y = y_c + r_y \sin\vartheta \longrightarrow (5)$$

Symmetry considerations can be used to further reduce computations. An ellipse in standard position is symmetric between quadrants, but unlike a circle, it is not symmetric between the two octants of a quadrant.




Thus, we must calculate pixel positions along the elliptical arc throughout one quadrant, and then we obtain positions in the remaining three quadrants by symmetry as shown below: I

**Midpoint Ellipse Algorithm:**

Our approach here is similar *to* that used in displaying a raster circle. Given parameters $\mathbf{r_x}$, $\mathbf{r_y}$, and $\mathbf{(x_c, y_c)}$, we determine points ***(x, y)*** for an ellipse in standard position centered on the origin, and then we shift the points so the ellipse is centered at $\mathbf{(x_c, y_c)}$.

The midpoint ellipse method is applied throughout the first quadrant in two parts. The following diagram shows the division of the first quadrant according to the slope of an ellipse with $r_x < r_y$.

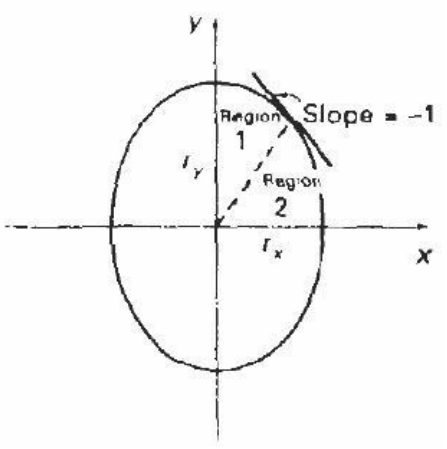
Regions 1 and 2 can be processed in various ways. We can start at position $\mathbf{(0, r_y)}$ and step clockwise along the elliptical path in the first quadrant, shifting from unit steps in ***x*** to unit steps in **y** when the slope becomes less than **-1.**



www.Jntufastupdates.com 

Alternatively, we could start at $(r_x, 0)$ and select points in a counterclockwise order, shifting from unit steps in **y** to unit steps in **x** when the slope becomes greater than **-1.**

With parallel processors, we could calculate pixel positions in the two regions simultaneously.




We define an Ellipse Function from equation (4) with $(x_c, y_c) = (0, 0)$ as

$$f_{ellipse}(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2$$

which has the following properties:

$$f_{ellipse}(x, y) \begin{cases} < 0, & \text{if (x, y) is inside the ellipse boundary} \\ = 0, & \text{if (x, y) is on the ellipse boundary} \\ > 0, & \text{if (x, y) is outside the ellipse boundary} \end{cases}$$

Thus the ellipse function serves as the decision parameter in the Midpoint 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,
$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2$$
3. At each **x,** position in region 1, starting at $k = 3$, perform the following test:
If $p1 < 0$, the next point along the ellipse centered on (0, 0) is $(x_{k+1}, y_k)$
$$\text{and } p1_{k+1} = p1_k + 2r_y^2 x_{k+1} + r_y^2$$
Otherwise, the next point along the circle is $(x_{k+1}, y_{k-1})$ and
$$p1_{k+1} = p1_k + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_y^2$$
with
$$2r_y^2 x_{k+1} = 2r_y^2 x_k + 2r_y^2,$$
$$2r_x^2 y_{k+1} = 2r_x^2 y_k - 2r_x^2$$
and continue until $2r_y^2 x \geq 2r_x^2 y$
4. Calculate the initial value of the decision parameter in region **2** using the last point $(x_0, y_0)$ calculated in region 1 as
$$p2_0 = r_y^2 x_0 + \frac{1}{2}^2 + r_x^2 (y_0 - 1)^2 - r_x^2 r_y^2$$
5. At each $y_k$ position in region **2,** starting at k = 0, perform the following test:
If $p_k > 0$, the next point along the ellipse centered on **(0, 0)** is $(x_k, y_k - 1)$
$$\text{and } p2_{k+1} = p2_k - 2r_x^2 y_{k+1} + r_x^2$$
Otherwise, the next point along the circle is $(x_k + 1, y_k - 1)$ and $p2_k$
$$_{+1} = p2_k + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_x^2$$
using the same incremental calculations for x and y as in region 1.
6. Determine symmetry points in the other three quadrants.
7. Move each calculated pixel position (x, y) onto the elliptical path centered on $(x_c, y_c)$ and plot the coordinate values:

$$x = x + x_c$$

$$, y = y + y_c$$

8. Repeat the steps for region 1 until $2r_y{}^2 x \geq 2r_x{}^2 y$ .



www.Jntufastupdates.com 

## ATTRIBUTES OF OUTPUT PRIMITIVES: LINE ATTRIBUTES

In general, any parameter that affects the way a primitive is to be displayed is referred to as an ***attribute parameter.*** Some attribute parameters, such as color and size, determine the fundamental characteristics of a primitive.

## 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:**

Possible selections for the line-type attribute include *solid lines, dashed lines,* and *dotted lines*. We modify a line drawing algorithm to generate such lines by setting the length and spacing of displayed solid sections along the line path. A dashed line could be displayed by generating an interdash spacing that is equal to the length of the solid sections. Similar methods are used to produce other line-type variations.

To set line type attributes in a PHICS application program, a user invokes the function

```
setLinetype (lt)
```

where parameter 1 is assigned a positive integer value of 1, 2, 3, or 4 to generate lines that are, respectively, solid, dashed, dotted, or dash-dotted.

Raster line algorithms display line-type attributes by plotting pixel spans. For the various dashed, dotted, and dot-dashed pattern, the line-drawing procedure outputs sections of contiguous pixels along the line path, skipping over a number of intervening pixels between the solid spans.

**Line Width:**

Implementation of line-width options depends on the capabilities of the output device. A heavy line on the video monitor could be displayed as adjacent parallel lines, while a pen plotter might require pen changes.

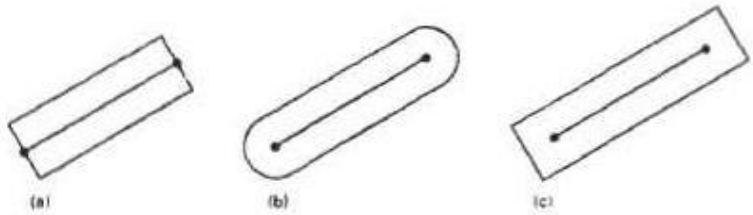
As with other PHIGS attributes, a line-width command is used to set the current line-width value in the attribute list.

We set the line-width attribute with the command:

```
setLinesidthScaleFactor (lw)
```

Line-width parameter **lw** is assigned a positive number to indicate the relative width of the line to be displayed. A value of 1 specifies a standard-width line.

For instance, on a pen plotter, 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.




*(a) butt caps, (b) round caps, and (c) projecting square caps*

Another problem with implementing width options using horizontal or vertical pixel spans is that the method produces lines whose ends are horizontal or vertical regardless of the slope of the line. This effect is more noticeable with very thick lines. We can adjust the shape of the line ends to give them a better appearance by adding ***line caps***.

One kind of line cap is the ***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. If the specified line has slope m, the square end of the thick line has slope - 1 / m .



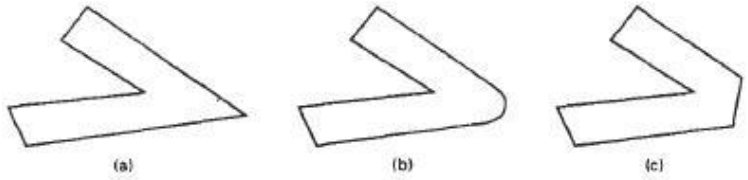
www.Jntufastupdates.com 

Another line cap is the ***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.

A third type of line cap is the ***projecting square cap***. Here, we simply extend the line and add butt caps that are positioned one-half of the line width beyond the specified endpoints.

Displaying thick lines using horizontal and vertical pixel spans, for example, leaves pixel gaps at the boundaries between lines of different slopes where there is a shift from horizontal spans to vertical spans. We can generate thick polylines that are smoothly joined at the cost of additional processing at the segment endpoints.

The following figure shows three possible methods for smoothly joining two line segments. A ***miter join*** is accomplished by extending the outer boundaries of each of the two lines until they meet. A ***round join*** is produced by capping the connection between the two segments with a circular boundary whose diameter is equal to the line width. And a ***bevel join*** is generated by displaying the line segments with butt caps and filling in the triangular gap where the segments meet.



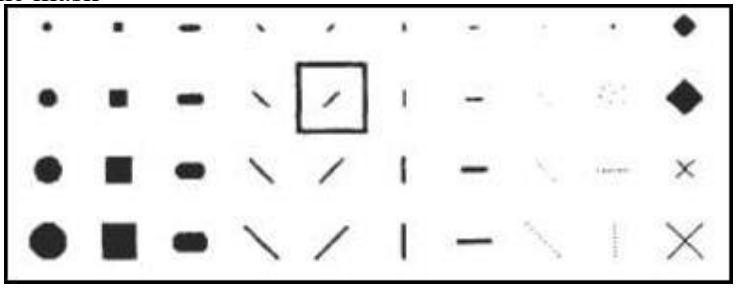

*(a) miter join, (b) round join, and (c) bevel join*

**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 below.

These shapes can be stored in a pixel mask that identifies the array of pixel positions that are to be set along the line path.

To avoid setting pixels more than once in the frame buffer, we can simply accumulate the horizontal spans generated at each position of the mask and keep track of the beginning and ending *x* positions for the spans across each scan line.



Lines generated with pen (or brush) shapes can be displayed in various widths by changing the size of the mask. Also, lines can be displayed with selected patterns by superimposing the pattern values onto the pen or brush mask. An additional pattern option that can be provided in a paint package is the display of simulated brush strokes.

**Line Color:**

When a system provides color (or intensity) options, a parameter giving the current color index is included in the list of system-attribute values. A polyline 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 ***setpixel*** procedure. The number of color choices depends on the number of bits available per pixel in the frame buffer.

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**. A line drawn in the background color is invisible, and a user can erase a previously displayed line by respecifying it in the background color

**CURVE ATTRIBUTES:**

Parameters for curve attributes are the same as those for line segments. We can display curves with varying colors, widths, dotdash patterns, and available pen or brush options.

Raster curves of various widths can be displayed using the method of horizontal or vertical pixel spans. Where the magnitude of the curve slope is less than 1, we plot vertical spans; where the slope magnitude is greater than 1, we plot horizontal spans.

Another method for displaying thick curves is to fill in the area between two parallel curve paths, whose separation distance is equal to the desired width. We could do this using the specified curve path as one boundary and setting up the second boundary either inside or outside the original curve path. This approach shifts the original curve path either inward or outward, depending on which direction we choose for the second boundary. We can maintain the original curve position by setting the two boundary curves at a distance of one-half the width on either side of the specified curve path.

Although this method is accurate for generating thick circles, it provides only an approximation to the true area of other thick curves.

Curves drawn with pen and brush shapes can be displayed in different sizes and with superimposed patterns or simulated brush strokes.

## 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. General purpose raster-scan systems usually provide a wide range of colors, while random-scan monitors typically offer only a few color choices.

Color options are numerically coded with values ranging from 0 through the positive integers. For CRT monitors, these color codes are then converted to intensity level settings for the electron beams.

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, or

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 the Table.

Each of the three bit positions is used to control the intensity level (either on or off) of the corresponding electron gun in an RGB monitor. The leftmost bit controls the red gun, the middle bit controls the green gun, and the rightmost bit controls the blue gun. Adding more bits per pixel to the frame buffer increases the number of color choices.

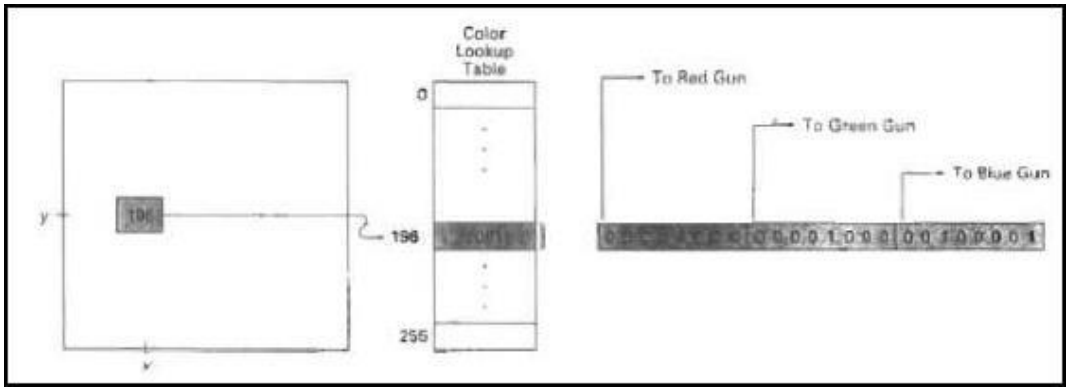
RGB system needs 3 megabytes of storage for the frame buffer. Color tables are an alternate means for providing extended color capabilities to a user without requiring large frame buffers. Lower-cost personal computer systems often use color tables to reduce frame-buffer storage requirements.

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:

The following figure illustrates a possible scheme for storing color values in a ***color lookup table*** (or ***video lookup table***), where frame-buffer values art- now used as indices into the color table.



Systems employing this particular lookup table would allow a user to select any 256 colors for simultaneous display 17 million colors. Compared to a full-color system, this scheme reduces the number of simultaneous colors that can be displayed, but it also reduces the frame buffer storage requirements to 1 megabyte.

Advantages in storing color codes in a lookup table are,

Use of a color table can provide a "reasonable" number of simultaneous colors without requiring large frame buffers. For most applications, 256 or 512 different colors are sufficient for a single picture.

Table entries can be changed at any time, allowing a user to be able to experiment easily with different color combinations.

Visualization applications can store values for some physical quantity, such as energy, in the frame buffer and use a lookup table to try out various color encodings without changing the pixel values.

In visualization and image-processing applications, color tables are a convenient means for setting color thresholds so that all pixel values above or below a specified threshold can be set to the same color.

For these reasons, some systems provide both capabilities for color-code storage, so that a user can elect either to use color tables or to store color codes directly in the frame buffer.

**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. This allows the intensity settings to be easily adapted to systems with differing grayscale capabilities.

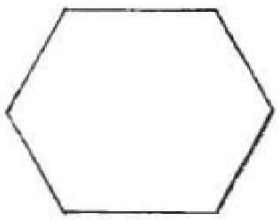
If additional bits per pixel are available in the frame buffer, the value of 0.33 would be mapped to the nearest level. With 3 bits per pixel, we can accommodate 8 gray levels; while 8 bits per pixel would give us 256 shades of gray.

An alternative scheme for storing the intensity information is to convert each intensity code directly to the voltage value that produces this grayscale level on the output device in use.

When multiple output devices are available at an installation, the same color-table interface may be used for all monitors. In this case, a color table for a monochrome monitor can be set up using a range of RGB values.

**TABLE 4-2**
INTENSITY CODES FOR A FOUR-LEVEL GRAYSCALE SYSTEM

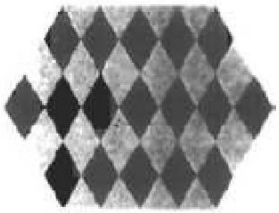
| Intensity Codes | Stored Intensity Values In The Frame Buffer (Binary Code) | Displayed Grayscale |
|---|---|---|
| 0.0 | 0 | |
| 0.33 | 1 | |
| 0.67 | 2 | |
| 1.0 | 3 | |



Hollow
(a)



Solid
(b)



Patterned
(c)

**AREA-FILL ATTRIBUTES:**

Options for filling a defined region include a choice between a solid color or a patterned fill and choices for the particular colors and patterns. These fill options can be applied to polygon regions or to areas defined with curved boundaries, depending on the capabilities of the available package. In addition, areas can be painted using various brush styles, colors, and transparency parameters.

**Fill Styles**

Areas are displayed with three basic fill styles: ***hollow with a color border, filled with a solid color, or Wed 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 fs include *hollow, solid,* and

Another value for fill style is hatch, which is used to fill an area with selected hatching patterns- *parallel lines or crossed lines.*

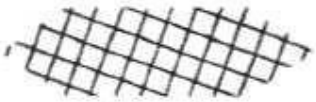
Fill selections for parameter fs are normally applied to polygon areas, but they can also be implemented to fill regions with curved boundaries.

***Hollow areas*** are displayed using only the boundary outline, with the interior color the same as the background color. A ***solid fill*** is displayed in a single color up to and including the borders of the region. The color for a solid interior or for a hollow area outline is chosen with where fillcolor parameter fc is set to the desired color code.

We can display area edges dotted or dashed, fat or thin, and in any available color regardless of how we have filled the interior.



Diagonal
Hatch Fill



Diagonal
Cross-Hatch Fill

**Pattern Fill**

We select fill patterns with

```
setInteriorStyleIndex (pi)
```

where pattern index parameter pi specifies a table position.

Separate tables are set up for hatch patterns. If we had selected hatch fill for the interior style in this program segment, then the value assigned to parameter pi is an index to the stored patterns in the hatch table.

For fill style *pattern,* table entries can be created on individual output devices with

```
SetPatternRepresentation( ws, pi, nx, ny, cp)
```

Parameter pi sets the pattern index number for workstation code ws, and cp is a two-dimensional array of color codes with nx columns and ny rows.

When a color array cp is to be applied to fill a region, we need to specify the size of the area that is to be covered by each element of the array. We do this by setting the rectangular coordinate extents of the pattern:

```
setPatternSize (dx, dy)
```

where parameters dx and dy give the coordinate width and height of the array mapping.

A reference position for starting a *pattern* fill is assigned with the statement

```
setPatternReferencePoint (positicn)
```

Parameter *position* is a pointer to coordinates *(xp, yp)* that fix the lower left comer of the rectangular pattern.

A WORKSTATION PATTERN TABLE WITH TWO ENTRIES, USING THE COLOR CODES OF TABLE 4-1

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

**Soft Fill:**

Modified boundary-fill and flood-fill procedures that are applied to repaint areas so that the fill color is combined with the background colors are referred to as ***soft-fill*** or ***tint-fill*** algorithms.

One use for these fill methods is to soften the fill colors at object borders that have been blurred to antialias the edges.

Another is to allow repainting of a color area that was originally filled with a semitransparent brush, where the current color is then a mixture of the brush color and the background colors "behind" the area.

In either case, we want the new fill color to have the same variations over the area as the current fill color.

As an example of this type of fill, the linear soft-fill algorithm repaints an area that was originally painted by merging a foreground color F with a single background color B, where **F ≠ B**.

Assuming we know the values for F and B, we can determine how these colors were originally combined by checking the current color contents of the frame buffer. The current RGB color P of each pixel within the area to be refilled is some linear combination of F and B:

$$\mathbf{P = tF + (1 - t)B}$$

where the "transparency" factor t has a value betwe en 0 nad 1 for each pixel.

The above vector equation holds for each RGB component of the colors, with

$$P = (P_R, P_C, P_R),\ F = (F_R, Fc, F_R),\ B = (B_R, Bc, B_B)$$

We can thus calculate the value of parameter $t$ using one of the RGB color components as

$$t = \frac{P - B}{kk}$$

where k = R, G, or B and $F_k \neq B_k$.

Similar soft-fill procedures can be applied to an area whose foreground color is to be merged with multiple background color areas, such as a checkerboard pattern.

## CHARACTER ATTRIBUTES

The appearance of displayed characters 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:

There are a great many text options that can be made available to graphics programmers. First of all, there is the choice of font (or typeface), which is a set of characters with a particular design style such as New York, Courier, Helvetica, London, Times Roman, and various special symbol groups.

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

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

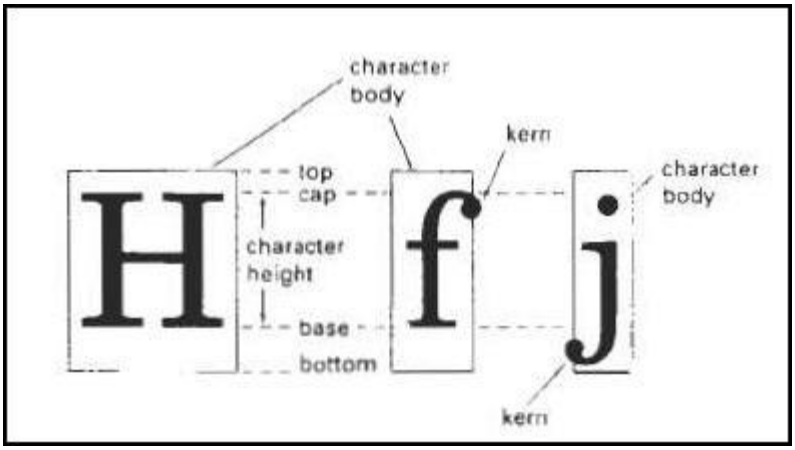
```
setTextFont (tf)
```

Color settings for displayed text are stored m the system attribute list and used by the procedures that load character definitions into the frame buffer. When a character string is to be displayed, the current color is used to set pixel values in the frame buffer corresponding to the character shapes and positions. 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.

We can adjust text size by scaling the overall dimensions (height and width) of characters or by scaling only the character width. Character size is specified by printers and compositors in *points,* where 1 point is 0.013837 inch (or approximately 1/72 inch).




The distance between the *bottomline* and the *topline* of the character body is the same for all characters in a particular size and typeface, but the body width may vary. *Proportionally spaced fonts* assign a smaller body width to narrow characters such as i, j, 1, and f compared to broad characters such as W or M. *Character height* is defined as the distance between the *baseline* and the *capline* of characters.

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

```
setCharacterHeight (ch)
```

The width only of text can be set with the function

```
setCharacterExpansionFactor (cw)
```

**Marker Attribute:**

A marker symbol is a single character that can he displayed in different colors and in different sizes. 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). Displayed marker types are centered on the marker coordinates. We set the marker size with

```
setMarkerSizeScaleFactor (ms)
```

with parameter marker size ms assigned a positive number.

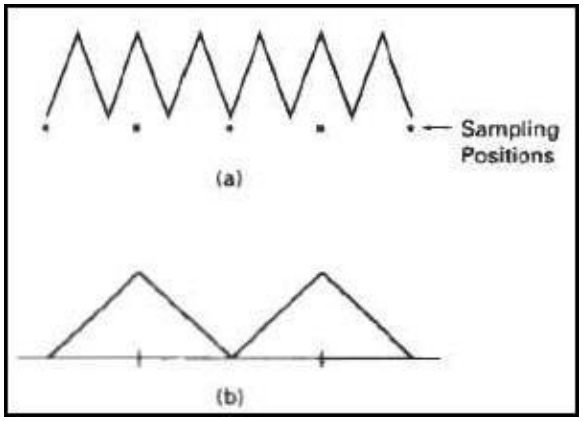
**ANTIALIASING**:

Displayed primitives generated by the raster algorithms have a jagged, or stairstep, appearance because the sampling process digitizes coordinate pints on an object to discrete integer pixel positions. This distortion of information due to low-frequency sampling (undersampling) is called aliasing.

We can improve the appearance of displayed raster lines by applying antialiasing methods that compensate for the undersampling process.

To avoid losing information from such periodic objects, we need to set the sampling frequency to at least twice that of the highest frequency occurring in the object, referred to as the ***Nyquist sampling frequency*** (or Nyquist sampling rate) $f_s$:

$$f_s = 2 f_{max}$$




Another way to state this is that the sampling interval should be no larger than one-half the cycle interval (called the Nyquist sampling interval). For x-interval sampling, the Nyquist sampling interval Ax, is

$$\Delta x_s = \frac{\Delta x_{cycle}}{2}$$

where $\Delta x_{cycle} = 1 / f_{max}$.

One way to increase sampling rate with raster systems is simply to display objects at higher resolution.

A straightforward antialiasing method is to increase sampling rate by treating the screen as if it were covered with a finer grid than is actually available. We can then use multiple sample points across this finer grid to determine an appropriate intensity level for each screen pixel. This technique of sampling object characteristics at a high resolution and displaying the results at a lower resolution is called ***supersampling*** (or ***postfiltering***, since the general method involves computing intensities, it subpixel grid positions, then combining the results to obtain the pixel intensities).

An alternative to supersampling is to determine pixel intensity by calculating the areas of overlap of each pixel with the objects to be displayed. Antialiasing by computing overlap areas is referred to as ***area sampling*** (or ***prefiltering***, since the intensity of the pixel as a whole is determined without calculating subpixel intensities). Pixel overlap areas are obtained by determining where object boundaries intersect individual pixel boundaries.

Raster objects can also be antialiased by shifting the display location of pixel areas. This technique, called ***pixel phasing***, is applied by "*micropositioning*" the electron beam in relation to object geometry.

Another advantage of supersampling with a finite-width line is that the total line intensity is distributed over more pixels.

## 2D TRANSFORMATIONS

### BASIC TRANSFORMATIONS

Changes in orientation, size, and shape are accomplished with geometric transformations that alter the coordinate descriptions of objects. The basic geometric transformations are translation, rotation, and scaling. Other transformations that are often applied to objects include reflection and shear.

### TRANSLATION

A ***translation*** is applied to an object by repositioning it along a straight-line path from one coordinate location to another. We translate a two-dimensional point by adding ***translation distances***, $t_x$ and $t_y$, to the original coordinate position ***(x, y)*** to move the point to a new position ***(x′, y′)***.

$$x' = x + t_x \quad , \quad y' = y + t_y$$

The translation distance pair $(t_x, t_y)$ is called a ***translation vector*** or ***shift vector***.

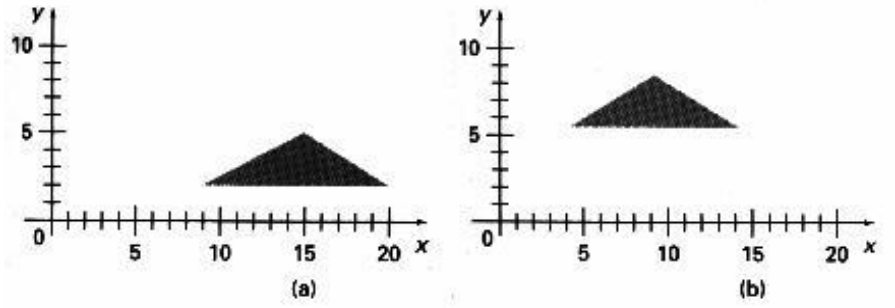
We can express the translation equations as a single matrix equation by using column vectors to represent coordinate positions and the translation vector:

$$P = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \qquad P' = \begin{bmatrix} x_1' \\ x_2' \end{bmatrix}, \qquad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

This allows us to write the two-dimensional translation equations in the matrix form:

$$P' = P + T$$

Sometimes matrix-transformation equations are expressed in terms of coordinate row vectors instead of column vectors. In this case, we would write the matrix representations as **P = [x y]** and **T = [$t_x$ $t_y$].**




Translation is a ***rigid-body transformation*** that moves objects without deformation, i.e., every point on the object is translated by the same amount.

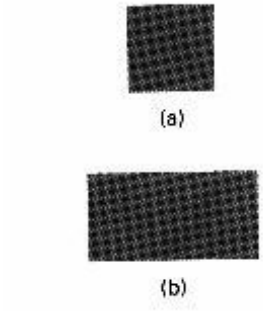
Polygons are translated by adding the translation vector to the coordinate position of each vertex and regenerating the polygon using the new set of vertex coordinates and the current attribute settings.

Similar methods are used to translate ***curved*** objects. To change the position of a circle or ellipse, we translate the center coordinates and redraw the figure in the new location. We translate other curves (splines) by displacing the coordinate positions defining the objects, and then we reconstruct the curve paths using the translated coordinate points.

**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)*** of each vertex by ***scaling factors*** $s_x$ and $s_y$ to produce the transformed coordinates ***(x', y'):***

$$x' = x \cdot s_x \quad , \quad y' = y \cdot s_y$$




Scaling factor $s_x$, scales objects in the ***x*** direction, while $s_y$ scales in the ***y*** direction. The transformation equations can be written in the matrix form as,

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

$$P' = S \cdot P$$

where S is the 2 by 2 scaling matrix.

Specifying a value of 1 for both $s_x$ and $s_y$ leaves the size of objects unchanged.

When $s_x$ and $s_y$ are assigned the same value, a ***uniform scaling*** is produced that maintains relative object proportions.

Unequal values for $s_x$ and $s_y$ result in a ***differential scaling*** that are often used in design applications, when pictures are constructed from a few basic shapes that can be adjusted by scaling and positioning transformations.

We can control the location of a scaled object by choosing a position, called the ***fixed point*** that is to remain unchanged after the scaling transformation.

Coordinates for the fixed point ***(xf , yf)*** can be chosen as one of the vertices, the object centroid, or any other position. A polygon is then scaled relative to the fixed point by scaling the distance from each vertex to the fixed point.

For a vertex with coordinates **(x, y)**, the scaled coordinates ***(x', y')*** are calculated as,

$$x' = x_f + (x - x_f)\, s_x \quad , \qquad y' = y_f + (y - y_f)\, s_y$$

We can rewrite these scaling transformations to separate the multiplicative and additive terms:

$$x' = x \cdot s_x + x_f(1 - s_x)$$

$$y' = y \cdot s_y + y_f(1 - s_y)$$

where the additive terms $x_f(1 - s_x)$ and $y_f(1 - s_y)$ are constant for all points in the object.

**ROTATION**

A two-dimensional rotation is applied to an object by repositioning it along a circular path in the **xy** plane. To generate a rotation, we specify a rotation angle 0 and the position ***(x, y)*** of the rotation point (or pivot point) about which the object is to be rotated.

Positive values for the rotation angle define counterclockwise rotations. Negative values rotate objects in the clockwise direction.

This transformation can also be described as a rotation about a rotation axis that is perpendicular to the **xy** plane and passes through the pivot point.

We first determine the transformation equations for rotation of a point position **P** when the pivot point is at the coordinate origin. The angular and coordinate relationships of the original and transformed point positions are shown in the diagram.




In this figure, r is the constant distance of the point from the origin, angle $\phi$ is the original angular position of the point from the horizontal, and $\theta$ is the rotation angle.

Using standard trigonometric identities, we can express the transformed coordinates in terms of angles $\theta$ and $\phi$ as

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

The original coordinates of the point in polar coordinates are,

$$x = r\cos\phi$$
$$y = r\sin\phi$$

Substituting expressions 2nd into 1st, we obtain the transformation equations for rotating a point at position ***(x, y)*** through an angle θ about the origin:

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

We can write the rotation equations in the matrix form:

$$P' = R \cdot P$$

where the rotation matrix is

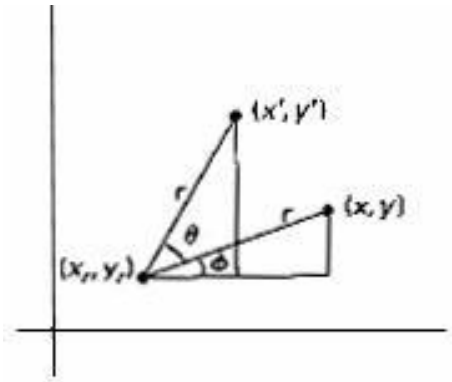
$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

When coordinate positions are represented as row vectors instead of column vectors, the matrix product in rotation equation is transposed so that the transformed row coordinate vector [x' y'] calculated as,

$$P'^T = (R \cdot P)^T$$
$$= P^T \cdot R^T$$

where $\mathbf{P'^T}$ = **[x y],** and the transpose $\mathbf{R^T}$ of matrix R is obtained by interchanging rows and columns. For a rotation matrix, the transpose is obtained by simply changing the sign of the sine terms.

Rotation of a point about an arbitrary pivot position is illustrated in the following diagram.




Using the trigonometric relationships in this figure, we can generalize to obtain the transformation equations for rotation of a point about any specified rotation position **($x_r$, $y_r$):**

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

As with translations, rotations are rigid-body transformations that move objects without deformation. Every point on an object is rotated through the same angle.

Polygons are rotated by displacing each vertex through the specified rotation angle and regenerating the polygon using the new vertices. Curved lines are rotated by repositioning the defining points and redrawing the curves.

A circle or an ellipse, for instance, can be rotated about a noncentral axis by moving the center position through the arc that subtends the specified rotation angle.

An ellipse can be rotated about its center coordinates by rotating the major and minor axes.

**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 design and picture construction applications, we perform translations, rotations, and scalings to tit the picture components into their proper positions.

Each of the basic transformations can be expressed in the general *matrix form*

$$\mathbf{P' = M_1 \cdot P + M_2}$$

with coordinate positions P and P' represented as column vectors.

Matrix $\mathbf{M_1}$ is a 2 by 2 array containing multiplicative factors, and $\mathbf{M_2}$ is a two-element column matrix containing translational terms.

For translation, $\mathbf{M_1}$ is the identity matrix.

For rotation, $\mathbf{M_2}$ contains the translational terms associated with the pivot point. For scaling, $\mathbf{M_2}$ contains the translational terms associated with the fixed point.

To produce a sequence of transformations with these equations, such as scaling followed by rotation then translation, we must calculate the transformed coordinate one step at a time.

To express any two-dimensional transformation as a matrix multiplication, we represent each Cartesian coordinate position *(x, y)* with the homogeneous coordinate triple $(x_h, y_h, h)$ where

$$x = \frac{x_h}{h}, \quad y = \frac{y_h}{h}$$

Thus, a general homogeneous coordinate representation can also be written as *(h.x, h.y, h)*.

For two-dimensional geometric transformations, we can choose the homogeneous parameter *h* to be any nonzero value. A convenient choice is simply to set *h = 1.*

Each two-dimensional position is then represented with homogeneous coordinates *(x, y, 1).*

The term ***homogeneous coordinates*** is used in mathematics to refer to the effect of this representation on Cartesian equations.

When a Cartesian point *(x, y)* is converted to a homogeneous representation $(x_h, y_h, h)$ equations containing *x* and *y* such as *f (x, y) = 0*, become homogeneous equations in the three parameters $x_h$, $y_h$ and *h.*

Expressing positions in homogeneous coordinates allows us to represent all geometric transformation equations as matrix multiplications. Coordinates are represented with three-element column vectors, and transformation operations are written as 3 by 3 matrices.

***For Translation***, we have

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

which we can write in the abbreviated form

$$\mathbf{P' = T}\,(t_x, t_y) \cdot \mathbf{P}$$

with $\mathbf{T}\,(t_x, t_y)$ as the 3 by 3 translation matrix.

The inverse of the translation matrix is obtained by replacing the translation parameters $t_x$ and $t_y$ with their negatives $-t_x$ and $-t_y$ .

Similarly, ***Rotation Transformation*** equations about the coordinate origin are written as

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

or as

$$\mathbf{P' = R\,(\theta)\,.\,P}$$

The rotation transformation operator R (θ) is the 3 by 3 matrix with rotation parameter θ. We get the inverse rotation matrix when θ is replaced with -θ.

A ***Scaling Transformation*** relative to the coordinate origin is now expressed as the matrix or

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

$$\mathbf{P' = S\,(}s_x\mathbf{,}\ s_y\mathbf{)\,.\,P}$$

where **S (*s*$_x$, *s*$_y$)** is the 3 by 3 matrix with parameters ***s*$_x$** and ***s*$_y$**.

Replacing these parameters with their multiplicative inverses **(1/*s*$_x$** and ***1/s*$_y$**)** yields the inverse scaling matrix.

Matrix representations are standard methods for implementing transformations in graphics systems. Rotations and Scalings relative to other reference positions are then handled as a succession of transformation operations.

An alternate approach in a graphics package is to provide parameters in the transformation functions for the scaling fixed-point coordinates and the pivot-point coordinates

## COMPOSITE TRANSFORMATIONS

With the matrix representations, 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. Forming products of transformation matrices is often referred to as a ***concatenation*****,** or ***composition*****,** of matrices.

For column-matrix representation of coordinate positions, we form composite transformations by multiplying matrices in order from right to left, i.e., each successive transformation matrix premultiplies the product of the preceding transformation matrices.

**Translations:**

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

$$\mathbf{P'} = \mathbf{T}(t_{x2}, t_{y2}) \cdot \{\mathbf{T}(t_{x1}, t_{y1}) \cdot \mathbf{P}\}$$

$$= \{\mathbf{T}(t_{x2}, t_{y2}) \cdot \mathbf{T}(t_{x1}, t_{y1})\} \cdot \mathbf{P}\}$$

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

Also, the composite transformation matrix for this sequence of translations is

Or

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

$\mathbf{T}(t_{x2}, t_{y2}) \cdot \mathbf{T}(t_{x1}, t_{y1}) = \mathbf{T}(t_{x1} + t_{x2}, t_{y1} + t_{y2})$
which demonstrates that two successive translations are additive.

*x*

**Rotations:**

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

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

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

$$\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1) = \mathbf{R}(\theta_1 + \theta_2)$$

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

$$\mathbf{P'} = \mathbf{R}(\theta_1 + \theta_2) \cdot \mathbf{P}$$

**Scaling:**

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

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

or

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

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

**General Pivot-Point Rotation:**

With a graphics package that only provides a rotate function for revolving objects about the coordinate origin, we can generate rotations about any selected pivot point $(x_r, y_r)$ by performing the following sequence of translate-rotate-translate operations:

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

This transformation sequence is illustrated in the following diagram. The composite transformation matrix for this sequence is obtained with the concatenation.

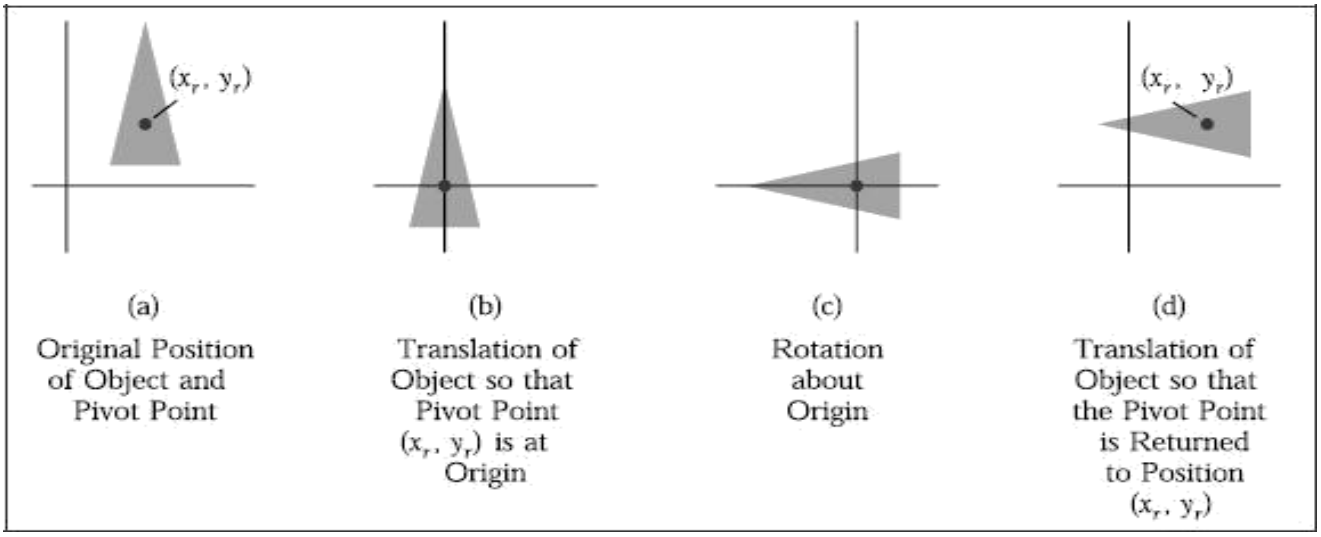
$$\begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix}$$

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

which can be expressed in the form

$$\mathbf{T}(x_r, y_r) \cdot \mathbf{R}(\theta) \cdot \mathbf{T}(-x_r, -y_r) = \mathbf{R}(x_r, y_r, \theta)$$

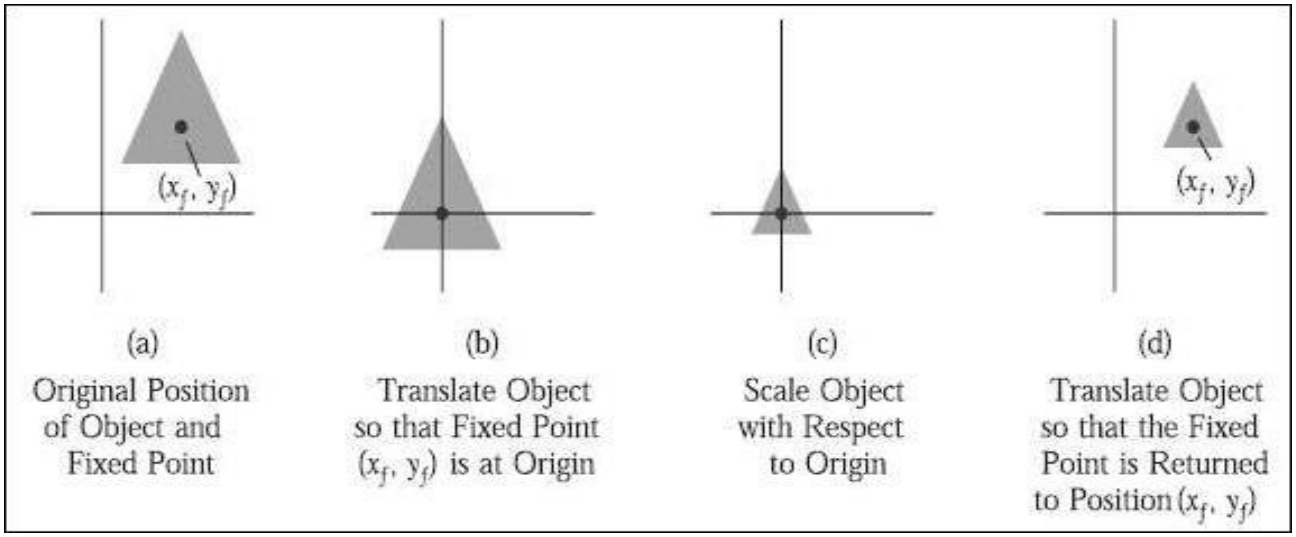
where $\mathbf{T}(-x_r, -y_r) = \mathbf{T}^{-1}(x_r, y_r)$




**General Fixed-Point Scaling:**

The following diagram illustrates a transformation sequence to produce scaling with respect to a selected fixed position ***(xf , yf )*** using a scaling function that can only scale relative to the coordinate origin.

1. Translate object so that the fixed point coincides with the coordinate origin.
2. Scale the object with respect to the coordinate origin.
3. Use the inverse translation of step 1 to return the object to its original position.




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

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

or

$$\mathbf{T}(x_f, y_f) \cdot \mathbf{S}(s_x, s_y) \cdot \mathbf{T}(-x_f, -y_f) = \mathbf{S}(x_f, y_f, s_x, s_y)$$

**OTHER TRANSFORMATIONS:**

Some other additional transformations are ***reflection*** and ***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" about the reflection axis. Some common reflections are as follows:

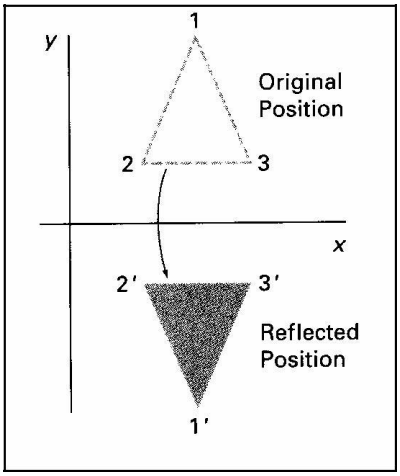
***x-Reflection:***

Reflection about the line $y = 0$, the $x$ axis, is accomplished with the transformation Matrix

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

This transformation keeps $x$ values the same, but "flips" the $y$ values of coordinate positions.

The resulting orientation of an object after it has been reflected about the $x$ axis is shown in the diagram.




***y-Reflection:***

A reflection about the y axis flips $x$ coordinates while keeping $y$ coordinates the same.

The matrix for this transformation is,

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

The diagram illustrates the change in position of an object that has been reflected about the line $x = 0$.

***Origin-Reflection:***

We flip both the $x$ and $y$ coordinates of a point by reflecting relative to an axis that is perpendicular to the $xy$ plane and that passes through the coordinate origin.

This transformation, referred to as a reflection relative to the coordinate origin, has the matrix representation:

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

If we chose the reflection axis as the diagonal line $y = x$, the reflection matrix is

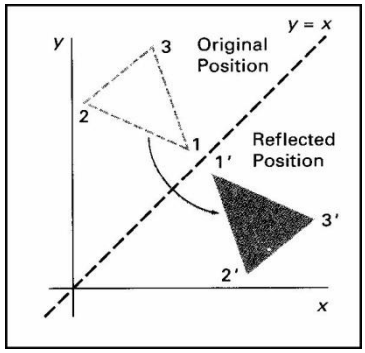
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

To obtain a transformation matrix for reflection about the diagonal $y = -x$, we could concatenate matrices for the transformation sequence:

(1) clockwise rotation by 45°,
(2) reflection about the y axis, and
(3) counterclockwise rotation by 45°.

The resulting transformation matrix is

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




Reflections about any line $y = mx + h$ in the $xy$ plane can be accomplished with a combination of translate-rotate-reflect transformations.

**Shear:**

A transformation that distorts (deform or alter) the shape of an object such that the transformed shape appears as if the object were composed of internal layers that had been caused to slide over each other is called a ***shear***.

Two common shearing transformations are those that shift coordinate *x* values and those that shift *y* values.

***x-Shearing:***

An *x*-direction shear relative to the x axis is produced with the transformation matrix

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

which transforms coordinate positions as

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

In the following diagram, $sh_x = 2$, changes the square into a parallelogram.

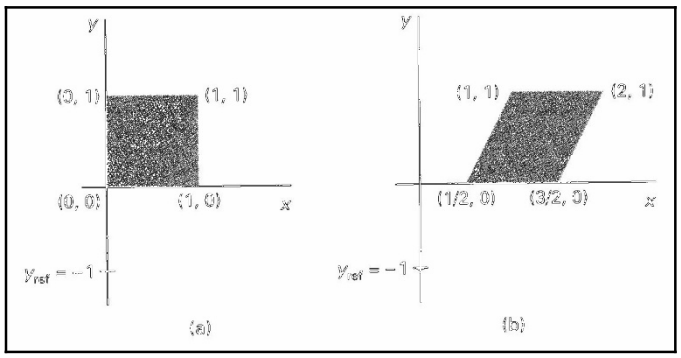
Negative values for $sh_x$ shift coordinate positions to the left. We can generate x-direction shears relative to other reference lines with

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

with coordinate positions transformed as

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

An example of this shearing transformation is given in the following diagram for a shear parameter of ½ relative to the line $y_{ref} = -1$.




***y-Shearing:***

A y-direction shear relative to the line $x = x_{ref}$ is generated with the transformation matrix

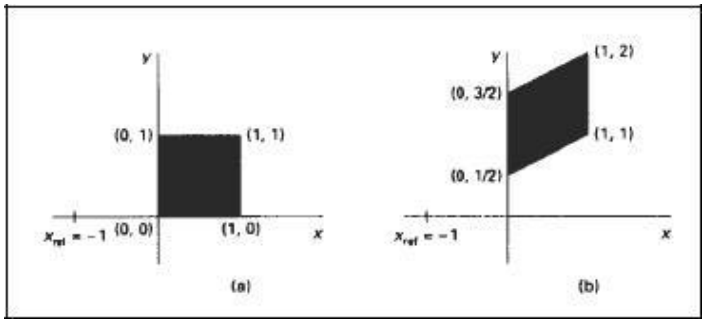
$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & -sh_y \cdot x_{ref} \\ 0 & 0 & 1 \end{bmatrix}$$

which generates transformed coordinate positions

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

This transformation shifts a coordinate position vertically by an amount proportional to its distance from the reference line $x = x_{ref}$.

The diagram shows the conversion of a square into a parallelogram with $sh_y = ½$ and $x_{ref} = -1$.



## VIEWING:

### THE VIEWING PIPELINE:

A world-coordinate area selected for display is called a ***window***. An area on a display device to which a window is mapped is called a ***viewport***.

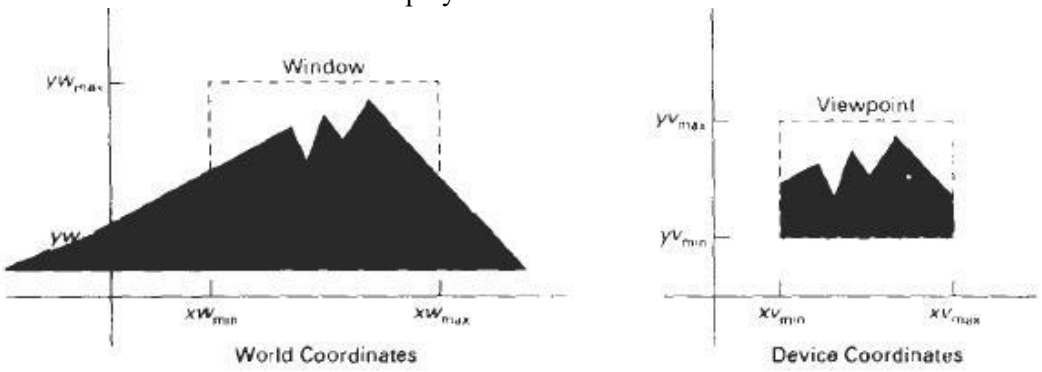
The window defines ***what*** is to be viewed.

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

Often, windows and viewports are rectangles in standard position, with the rectangle edges parallel to the coordinate axes.

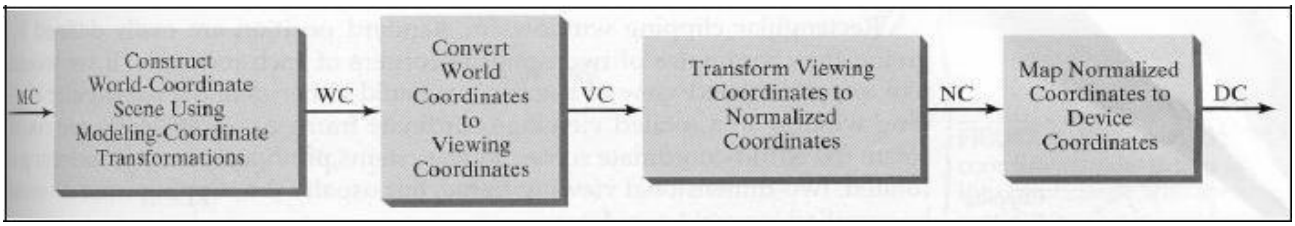
In general, the mapping of a part of a world-coordinate scene to device coordinates is referred to as a ***viewing transformation***.

Sometimes the two-dimensional viewing transformation is simply referred to as the ***window-to-viewport transformation*** or the ***windowing transformation.*** The term window to refer to an area of a world-coordinate scene that has been selected for display

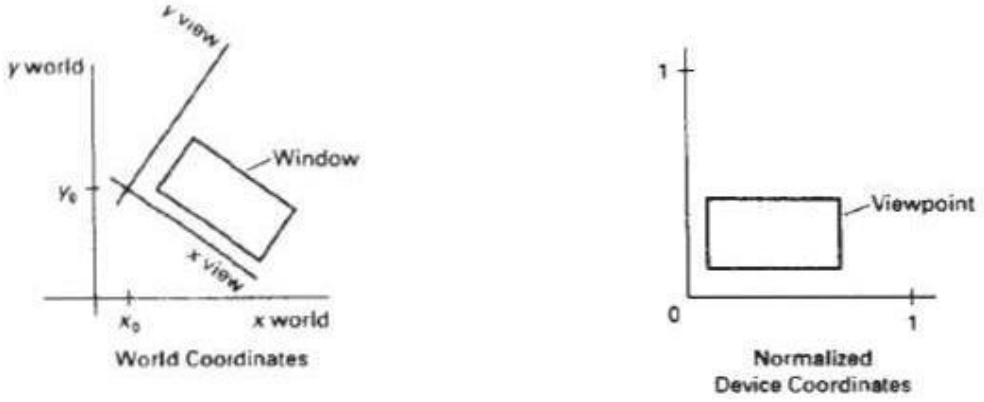



We carry out the viewing transformation in several steps, as indicated below.

1. First, we construct the scene in world coordinates using the output primitives and attributes.
2. 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.
3. 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.
4. 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.
5. 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.




The following diagram illustrates a rotated viewing-coordinate reference frame and the mapping to




www.Jntufastupdates.com

normalized coordinates.

By changing the position of the viewport, we can view objects at different positions on the display area of an output device. Also, by varying the size of viewports, we can change the size and proportions of displayed objects. We achieve zooming effects by successively mapping different-sized windows on a fixed-size viewport.

Panning effects are produced by moving a fixed-size window across the various objects in a scene. When all coordinate transformations are completed, viewport clipping can be performed in normalized coordinates or in device coordinates. This allows us to reduce computations by concatenating the various transformation matrices.

**VIEWING COORDINATE REFERENCE FRAME:**

This coordinate system provides the reference frame for specifying the world coordinate window. First, a viewing-coordinate origin is selected at some world position: $P_0 = (x_0, y_0)$. Then we need to establish the orientation, or rotation, of this reference frame. One way to do this is to specify a world vector V that defines the viewing $y_0$, direction. Vector V is called the ***view up vector***.

Given V, we can calculate the components of unit vectors $v = (v_x, v_y)$ and $u = (u_x, u_y)$ for the viewing $y_v$ and $x_v$ axes, respectively. These unit vectors are used to form the first and second rows of the rotation matrix R that aligns the viewing $x_v y_v$ axes with the world $x_w y_w$ axes.

We obtain the matrix for converting world coordinate positions to viewing coordinates as a two-step composite transformation:

First, we translate the viewing origin to the world origin,

Then we rotate to align the two coordinate reference frames.

The composite 2D transformation to convert world coordinates to viewing coordinate is

$$M_{WC,VC} = R \cdot T$$

where T is the translation matrix that takes the viewing origin point $\mathbf{P_0}$ to the world origin, and R is the rotation matrix that aligns the axes of the two reference frames.




*A viewing-coordinate frame is moved into coincidence with the world frame in two steps: (a) translate the viewing origin to the world origin, then (b) rotate to align the axes of the two systems.*

**WINDOW-TO-VIEWPORT COORDINATE TRANSFORMATION**




*A point at position (xw, yw) in a designated window is mapped to viewport coordinates (xv, yv) so that relative positions in the two areas are the same. -*

Once object descriptions have been transferred to the viewing reference frame, we choose the window extents in viewing coordinates and select the viewport limits in normalized coordinates. Object descriptions are then transferred to normalized device coordinates. We do this using a transformation that maintains the same relative placement of objects in normalized space as they had in viewing coordinates. If a coordinate position is at the center of the viewing window, for instance, it will be displayed at the center of the viewport.

The above diagram illustrates the window-to-viewport mapping. A point at position ***(xw, yw)*** in the window is mapped into position ***(xv, yv)*** in the associated viewport. To maintain the same relative placement in the viewport as in the window, we require that,

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

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

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

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

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

where the scaling factors are

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

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

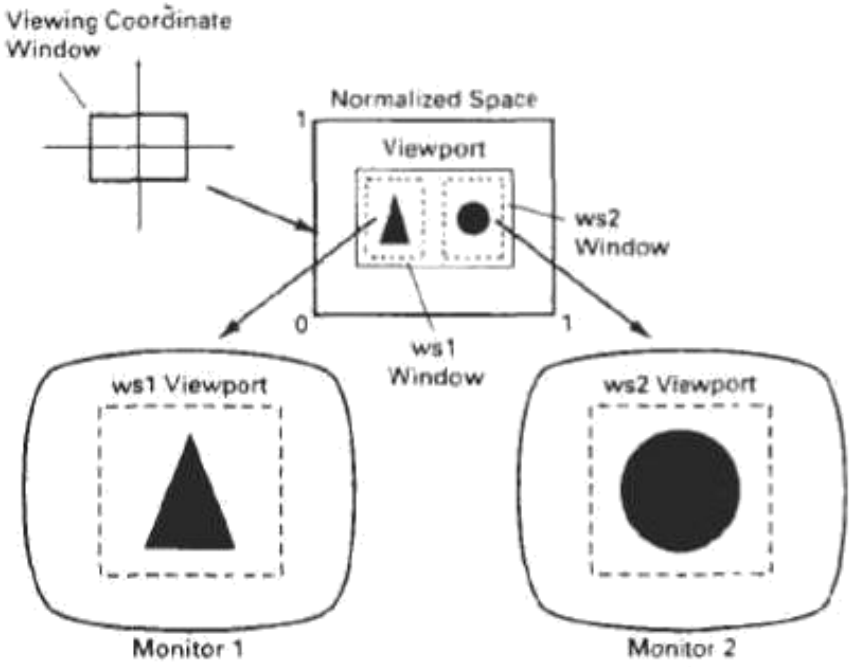
The Equations can also be derived with a set of transformations that converts the window area into the viewport area.

This conversion is performed with the following sequence of transformations:

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

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

The mapping, called the ***workstation transformation***, is accomplished by selecting a window area in normalized space and a viewport area in the coordinates of the display device.




*Mapping selected parts of a scene in normalized coordinates to different video monitors with workstation transformations.*

**2D CLIPPING:**

**CLIPPING OPERATIONS**

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

Applications of clipping include extracting part of a defined scene for viewing; identifying visible surfaces in three-dimensiona1 views; antialiasing line *seg*ments or object boundaries; creating objects using solid-modeling procedures; displaying a multiwindow environment; and drawing and painting operations that allow parts of a picture to be selected for copying, moving, erasing, or duplicating.

Clipping algorithms can be applied in world coordinates, so that only the contents of the window interior are mapped to device coordinates.

World-coordinate clipping removes those primitives outside the window from further consideration, thus eliminating the processing necessary to transform those primitives to device space.

Viewport clipping, can reduce calculations by allowing concatenation of viewing and geometric transformation matrices.

We consider algorithms for clipping the following primitive types

| | |
|---|---|
| Point Clipping | Curve Clipping |
| Line Clipping (straight-line segments) | Text Clipping |
| Area Clipping (polygons) | |

**POINT CLIPPING**

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

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

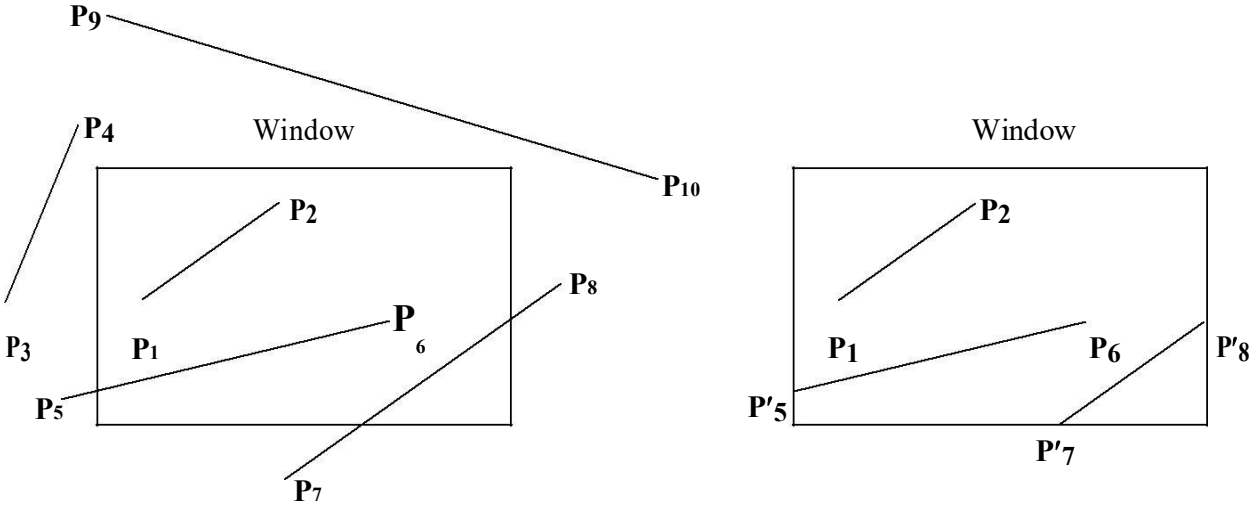
where the edges of the clip window ***($xw_{min}$, $xw_{max}$, $yw_{min}$, $yw_{max}$)*** can be either the world-coordinate window boundaries or viewport boundaries. If any one of these four inequalities is not satisfied, the point is clipped (not saved for display).

**LINE CLIPPING:**

**Explain the steps to perform Line Clipping. (Or) (5, 10 Marks)**
**Describe Cohen-Sutherland Line Clipping algorithm in detail.**

**Explain Liang-Barsky Line Clipping algorithm.**

The following diagram illustrates the possible relationships between line positions and a standard rectangular clipping region.




*(a) Before clipping* *(b) After clipping*

A line clipping procedure involves several parts.

First, we can test a given line segment to determine whether it lies completely inside the clipping window.

If it does not, we try to determine whether it lies completely outside the window.

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

We process lines through the "inside-outside" test by checking the line endpoints.

A line with both endpoints inside all clipping boundaries, such as the line from $P_1$ to $P_2$ is saved.

A line with both endpoints outside any one of the clip boundaries (line $P_3P_4$ in the diagram) is outside the window.

All other lines ***cross*** one or more clipping boundaries, and may require calculation of multiple intersection points.

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

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

could be used to determine values of parameter $u$ for intersections with the clipping boundary coordinates.

If the value of $u$ for an intersection with a rectangle boundary edge is outside the range $0$ to $1$, the line does not enter the interior of the window at 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 turn to determine whether any part of the line segment is to be displayed.

**Cohen-Sutherland Line Clipping:**

This is one of the oldest and most popular line-clipping procedures. Generally, the method speeds up the processing of line segments by performing initial tests that reduce the number of intersections that must he calculated.

Every line end-point 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. Regions are set up in reference to the boundaries as shown below:.

Each bit position in the region code is used to indicate one of the four relative coordinate positions of the point with respect to the 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 can be correlated with the bit positions as

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

| | | |
|---|---|---|
| **1001** | **1000** | **1010** |
| **0001** | **0000**<br>**Window** | **0010** |
| **0101** | **0100** | **0110** |

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.**

The region-code bit values can be determined with the following two steps:

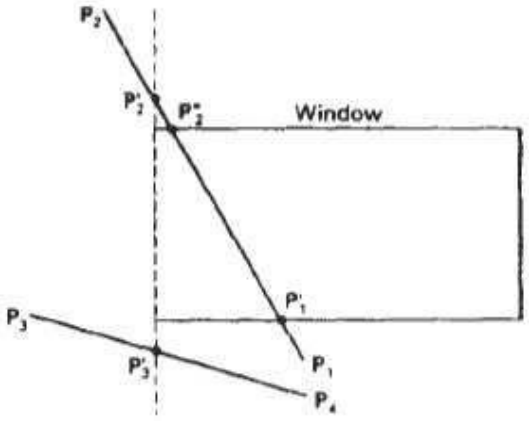
(1) Calculate differences between endpoint coordinates and clipping boundaries.

(2) Use the resultant sign bit of each difference calculation to set the corresponding value in the region code.

- Any lines that are completely contained within the window boundaries have a region code of 0000 for both endpoints, and we trivially 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 trivially reject these lines.
- 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.

.

To illustrate the specific steps in clipping lines against rectangular boundaries using the Cohen-Sutherland algorithm, we show how the lines in the above diagram could be processed.




Starting with the bottom endpoint of the line from $P_1$ to $P_2$, we check $P_1$, against the left, right, and bottom boundaries in turn and find that this point is below the clipping rectangle. We then find the intersection point $P'_1$ with the bottom boundary and discard the line section from $P_1$ to $P'_1$. The line now has been reduced to the section from $P'_1$ to $P_2$. Since $P_2$ is outside the clip window, we check this endpoint against the boundaries and find that it is to the left of the window. Intersection point $P'_2$ is calculated, but this point is above the window. So the final intersection calculation yields $P''_2$, and the line from $P'_1$ to $P''_2$ is saved. This completes processing for this line, so we save this part and go on to the next line.

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 $(x_2, y_2)$, 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 the $x$ value is set either to $xw_{min}$ or to $xw_{max}$ and the slope of the line is calculated as

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

Similarly, if we are looking for the intersection with a horizontal boundary, the $x$ coordinate can be calculated as

$$x = x_1 + \frac{y - y_1}{m}$$

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

**Liang-Barsky Line Clipping:**

Faster line clippers have been developed that are based on analysis of the parametric equation of a line segment, which we can write in the form

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

where $\Delta x = x_2 - x_1$, and $\Delta y = y_2 - y_1$.

Liang and Barsky independently devised an even faster parametric line-clipping algorithm. In this approach, we first write the point-clipping conditions in the parametric form:

$$xw_{min} \le x1 + u\,\Delta x \le xw_{max}$$
$$yw_{min} \le y1 + u\,\Delta y \le yw_{max}$$

Each of these four inequalities can be expressed as

$$u\,p_k \le q_k\,,\ k = 1, 2, 3, 4$$

where parameters p and q are defined as

$$p_1 = -\Delta x, \qquad q_1 = x_1 - xw_{min}$$
$$p_2 = \Delta x, \qquad q_2 = xw_{maz} - x_1$$
$$p_3 = -\Delta y, \qquad q_3 = y_1 - yw_{min}$$
$$p_4 = \Delta y, \qquad q_4 = yw_{maz} - y_1$$

Any line that is parallel to one of the clipping boundaries has $p_k = 0$ for the value of $k$ corresponding to that boundary ($k = 1, 2, 3,$ and $4$ correspond to the left, right, bottom, and top boundaries, respectively).

If, for that value of $k$, we also find $q_k < 0$, then the line is completely outside the boundary and can be eliminated from further consideration. If $q_k \ge 0$, the line is inside the parallel clipping boundary.

- When $p_k < 0$, the infinite extension of the line proceeds from the outside to the inside of the infinite extension of this particular clipping boundary.
- If $p$, > 0, the line proceeds from the inside to the outside.

For a nonzero value of $p_k$, we can calculate the value of **u** that corresponds to the point where the infinitely extended line intersects the extension of boundary k as

$$u = \frac{q_k}{p_k}$$

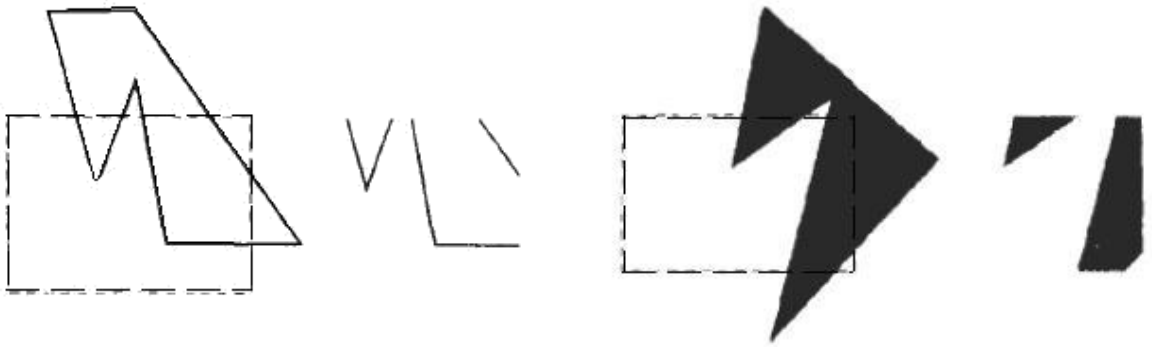
For each line, we can calculate values fur parameters $u_1$ and $u_2$ that define that part of the line that lies within the clip rectangle. The value of $u_1$ is determined by looking at the rectangle edges for which the line proceeds from the outside to the inside ($p < 0$). For these edges, we calculate $r_k = q_k / p_k$. The value of $u_1$ is taken as the largest of the set consisting of $0$ and the various values of $r$. Conversely, the value of $u_2$ is determined by examining the boundaries for which the line proceeds from inside to outside ($p > 0$).

A value of $r_k$ is calculated for each of these boundaries, and the value of $u$, is the minimum of the set consisting of 1 and the calculated $r$ values. If $u_1 > u_2$, the line is completely outside the clip window and it can be rejected. Otherwise, the endpoints of the clipped line are calculated from the two values of parameter $u$.

**POLYGON CLIPPING:**

A polygon boundary processed with a line clipper may be displayed as a series of unconnected line segments (fig. (a)) depending on the orientation of the polygon to the clipping window. But we want to display a bounded area after clipping (fig. (b)).

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.



*Before Clipping* *After Clipping* *Before Clipping* *After Clipping*

*Fig. (a)* *Fig. (b)*

**Sutherland-Hodgeman Polygon Clipping:**

We can correctly clip a polygon 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 in turn.

Beginning with the initial set of polygon vertices, we could first clip the polygon against the left rectangle boundary to produce a new sequence of vertices. The new set of vertices could then k successively passed to a right boundary clipper, a bottom boundary clipper, and a top boundary clipper, as shown in the diagram below. At each step, a new sequence of output vertices is generated and passed to the next window boundary clipper.



*Original Polygon* *Clip Left* *Clip Right* *Clip Bottom* *Clip Top*

There are four possible cases when processing vertices in sequence around the perimeter of a polygon. As each pair of adjacent polygon vertices is passed to a window boundary clipper, we make the following tests:

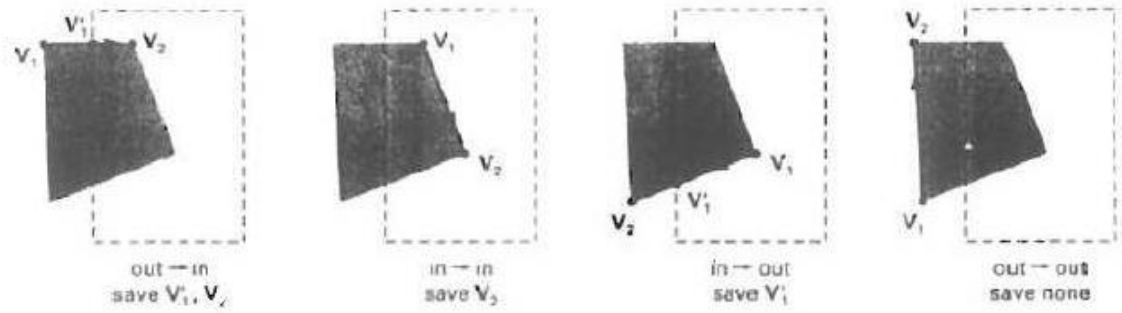
(1) If the first vertex is outside the window boundary and the second vertex is inside, both the intersection point of the polygon edge with the window boundary and the second vertex are added to the output vertex list.

(2) If both input vertices are inside the window boundary, only the second vertex is added to the output vertex list.

(3) If the first vertex is inside the window boundary and the second vertex is outside, only the edge intersection with the window boundary is added to the output vertex list.

(4) If both input vertices are outside the window boundary, nothing is added to the output list.

These four cases are illustrated in the following diagram for successive pairs of polygon vertices.

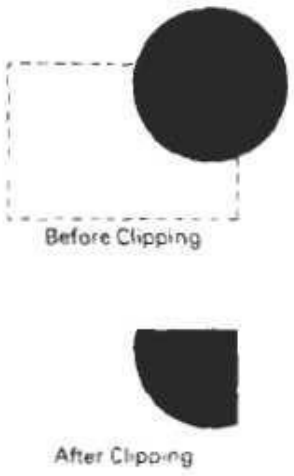


Once all vertices have been processed for one clip window boundary, the output list of vertices is clipped against the next window boundary.

Convex polygons are correctly clipped by this algorithm, but concave polygons may be displayed with extraneous lines as shown below. This occurs when the clipped polygon should have two or more separate sections.

**CURVE CLIPPING:**

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

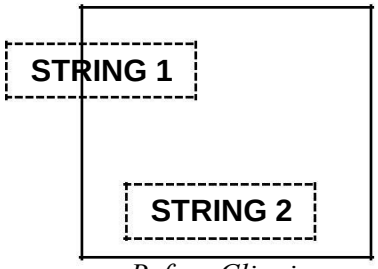



The bounding rectangle for a circle or other curved object can be used first to test for overlap with a rectangular clip window. If the bounding rectangle for the object is completely inside the window, we save the object. If the rectangle is determined to be completely outside the window, we discard 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.

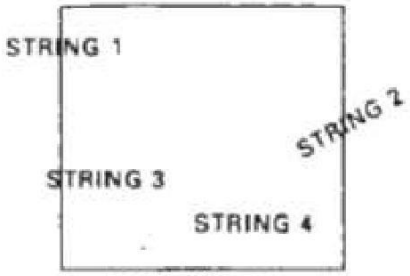
Similar procedures can be applied when clipping a curved object against a general polygon clip region. On the first pass, we can clip the bounding rectangle of the object against the bounding rectangle of the clip region. If the two regions overlap, we will need to solve the simultaneous line-curve equations to obtain the clipping intersection points.

**TEXT CLIPPING:**

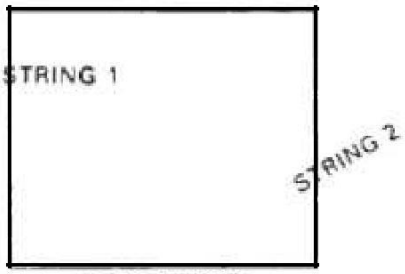
There are several techniques that can be used to provide text clipping in a graphics package. The clipping technique used will depend on the methods used to generate characters and the requirements of a particular application.



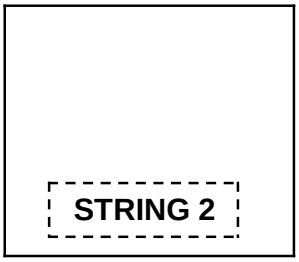

*Before Clipping*



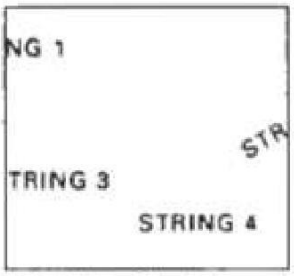

Before Clipping




Before Clipping

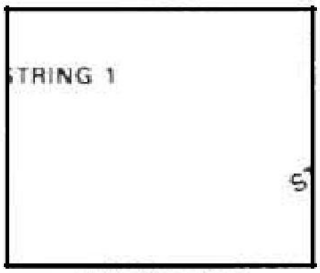



*After Clipping*
*fig. (1)*




After Clipping

*fig. (2)*




After Clipping

*fig. (3)*

The simplest method for processing character strings relative to a window boundary is to use the ***all-or-none string-clipping*** strategy shown in the diagram (fig. (1)). If all of the string is inside a clip window, we keep it. Otherwise, the string is discarded.

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 shown in fig. (2).

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 as in fig. (3).

**EXTERIOR CLIPPING:**

There are procedures to clip ***inside*** the region, to save the parts if the picture that are ***outside*** the region ***exterior clipping***.

A typical example of the application of exterior clipping is in multiple-window systems. To correctly display the screen windows, we often need to apply both internal and external clipping.

Objects within a window are clipped to the interior of that window. When other higher-priority windows overlap these objects, the objects are also clipped to the exterior of the overlapping windows.

Exterior clipping is used also in other applications that require overlapping pictures. Examples here include the design of page layouts in advertising or publishing applications or for adding labels or design patterns to a picture. The technique can also be used for combining graphs, maps, or schematics. For these applications, we can use exterior clipping to provide a space for an insert into a larger picture.

Procedures for clipping objects to the interior of concave polygon windows can also make use of external clipping.