

# COMPUTER GRAPHICS

## UNIT-II

**R.MANIMEGALAI**

DEPARTMENT OF COMPUTER SCIENCE

PERIYAR GOVT ARTS COLLEGE

CUDDALORE.

# Attributes of Output Primitives

Structure :

- ❖ Definition
- ❖ Line Attribute
- ❖ Curve Attribute
- ❖ Color and Grayscale Level
- ❖ Area Filled Attribute
- ❖ Text and Characters

# Introduction

The way a primitive is to be displayed is referred to as an *Attribute Parameter*.

Some attribute parameters include color ,size etc.

Different ways to incorporate attribute changes :

- ❑ Extend the parameter list associated with each primitive
- ❑ Maintain a system list of current attribute values and use separate functions to set attributes.

# LINE ATTRIBUTES

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



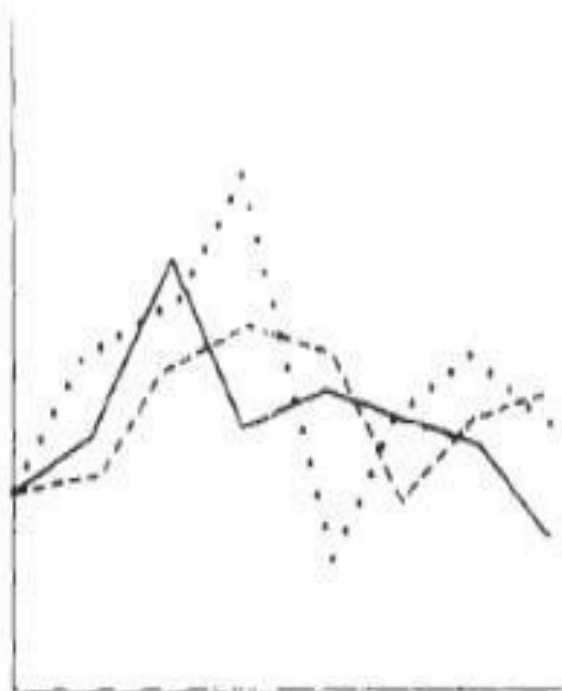
# Line Type

- ▶ 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 inter dash spacing that is equal to the length of the solid sections. Both the length of the dashes and the inter dash spacing are often specified as user options.



- ▶ A dotted line can be displayed by generating very short dashes with the spacing equal to or greater than the dash size.
- ▶ To set line type attributes in a **PHICS application program**, a user invokes the function  
**setLinetype (It)**
- ▶ where parameter **It** is assigned a positive integer value of **1,2,3, or 4 to generate.**
- ▶ lines that are, respectively, solid, dashed, dotted, or dash-dotted.





---

*Figure 4-1*  
Plotting three data sets with three  
different line types, as output by the  
`chartData` procedure.



- ▶ 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.





- ▶ Pixel counts for the span length and interspan spacing can be specified in a pixel mask, which is a string containing the digits 1 and 0 to indicate which positions to plot along the line path.
- ▶ The mask 1111000, for instance, could be used to display a dashed line with a dash length of four pixels and an interdash spacing of three pixels.



Figure 4-2  
Unequal-length dashes  
displayed with the same  
number of pixels.

# Line Width

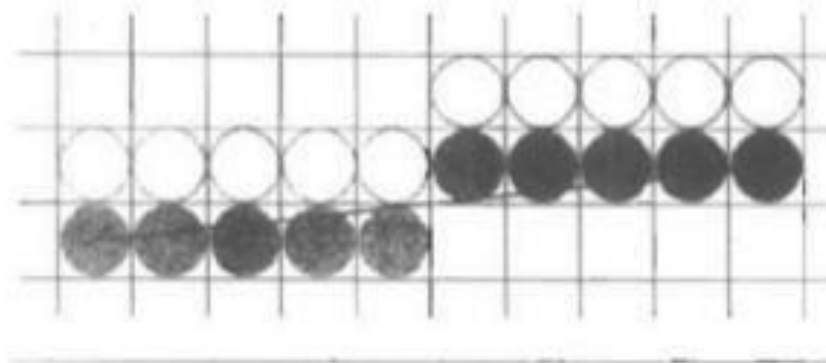
- ▶ Line-width options depends on the capabilities of the output device. A heavy line on video monitor could be displayed as adjacent parallel lines, while a pen plotter might require pen changes.
- ▶ We set the line-width attribute with the command:  
using  
**setLineWidthScaleFactor(lw);**
- ▶ Line-width parameter lw is assigned a positive number to indicate the relative width of the line to be displayed..



- ▶ A value of  $l$  specifies a standard-width line. on pen plotter, for instance, 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.
- ▶ For raster implementation, a standard-width line is generated with single pixels at each sample position.
- ▶ Other-width link **line Attributes** are displayed as positive integer multiples of the standard line by plotting additional pixels along adjacent parallel line paths.

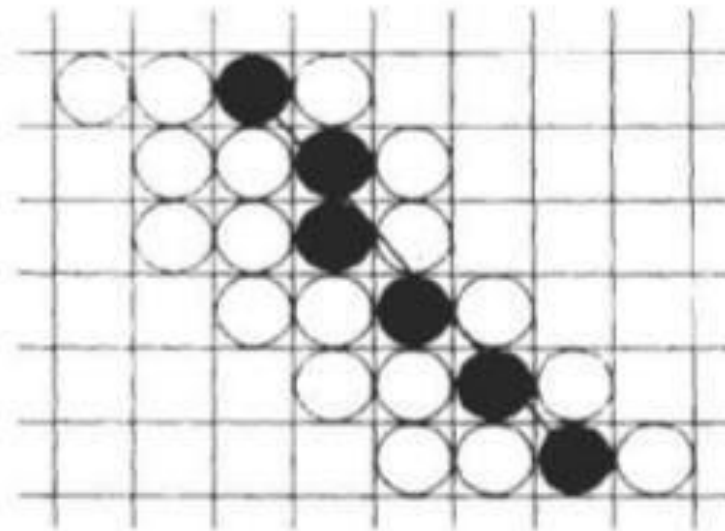


- ▶ For lines with slope magnitude less than 1, we can modify a line-drawing routine to display thick lines by plotting a vertical span of pixels at each  $x$  position along the line. The number of pixels in each span is set equal to the integer magnitude of parameter  $lw$ .



*Figure 4-3*  
Double-wide raster line with slope  $|m| < 1$  generated with vertical pixel spans.

- ▶ For lines with slope magnitude greater than 1, we can plot thick lines with horizontal spans, alternately picking up pixels to the right and left of the line path.



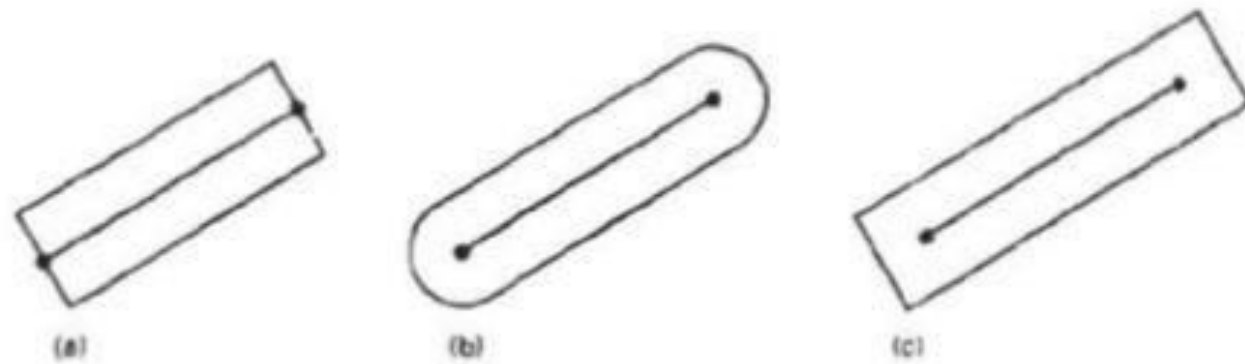
*Figure 4-4*  
Raster line with slope  $|m| > 1$   
and line-width parameter  $2w = 4$   
plotted with horizontal pixel spans.

- ▶ 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  $-1 / m$ .



- ▶ 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.





*Figure 4-5*  
Thick lines drawn with (a) butt caps, (b) round caps, and (c) projecting square caps.



- ▶ We can generate thick polylines that are smoothly joined at the cost of additional processing at the segment endpoints.
- ▶ There three possible methods for smoothly joining two line segments.
- ▶ **miter join**
- ▶ **round join**
- ▶ **bevel join.**



- ▶ 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.
- ▶ a **bevel join** is generated by displaying the line segments with butt caps and filling in the triangular gap where the segments meet.



Figure 4-6

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

# Pen and Brush Options

- ▶ With some packages, lines can be displayed with pen or brush selections.
- ▶ **Options** in this category include shape, size, and pattern.

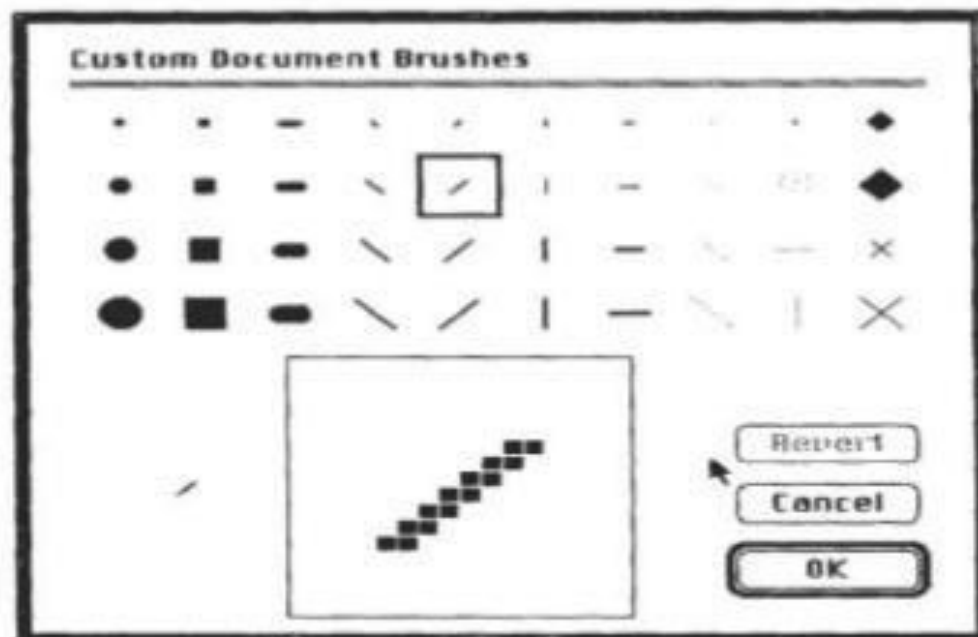
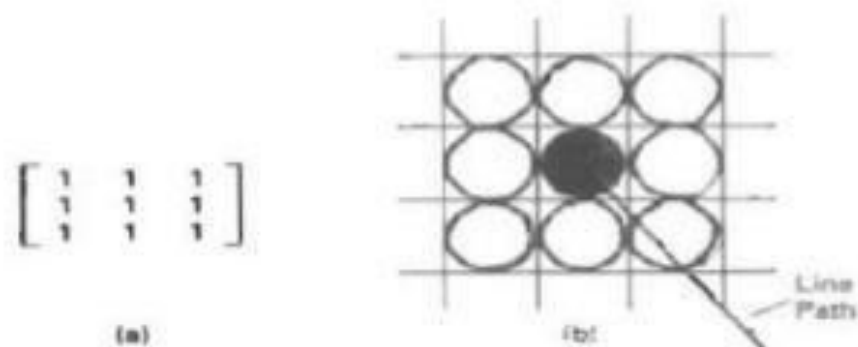
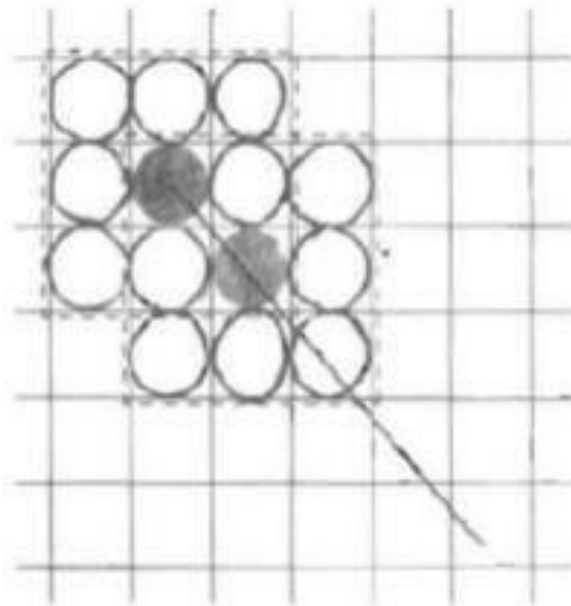


Figure 4-7  
Pen and brush shapes for line display

- ▶ 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.



**Figure 4-8**  
(a) A pixel mask for a rectangular pen, and (b) the associated array of pixels displayed by centering the mask over a specified pixel position.



*Figure 4-9*  
Generating a line with the pen  
shape of Fig. 4-8.



*Figure 4-10*  
Curved lines drawn with a paint program using various shapes and  
patterns. From left to right, the brush shapes are square, round,  
diagonal line, dot pattern, and faded airbrush.

# Line Color

- ▶ 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 **PHICS 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.
- ▶ An example of the use of the various line attribute commands in an applications program is given by the following sequence of statements:



- ▶ `setLineType ( 2 ) ;`
- ▶ `setLinewidthScaleFactor ( 2 : ;`
- ▶ `setPolylinesColourIndex ( 5 ) ;`
- ▶ `polyline ( n l , wcpoints1 ) :`
- ▶ `setPolyline:clourIndex ( 61 ;`
- ▶ `polyline ( nl , wcpoints2 ) :`





# CURVE ATTRIBUTES

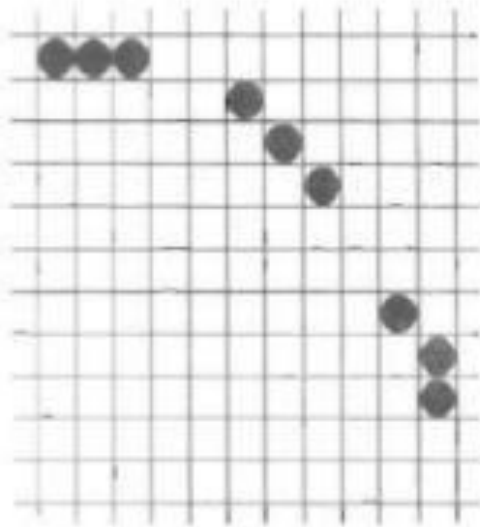
- ▶ Parameters for curve attributes are the same as those for line segments. We can display curves with varying colors, widths, dotdash patterns, and available pen or brush options. Methods for adapting curve-drawing algorithms to accommodate attribute selections are similar to those for line drawing.



# CURVE ATTRIBUTES

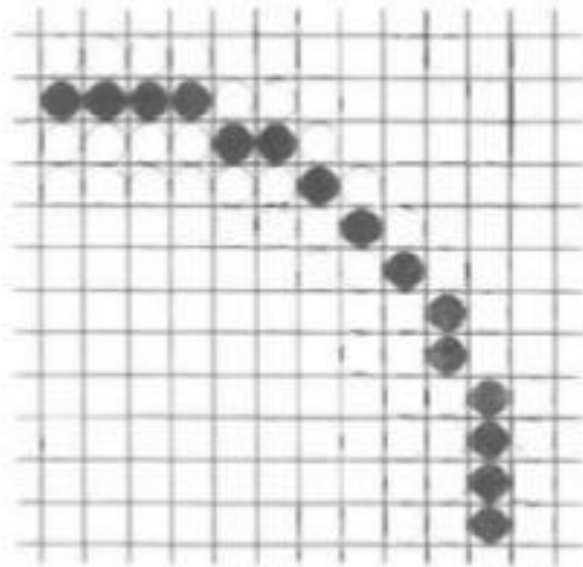
- ▶ Raster curves of various widths can be displayed using the method of horizontal or vertical pixel spans. Where the magnitude of the curve slope is less than 1, we plot vertical spans; where the slope magnitude is greater than 1, we plot horizontal spans.



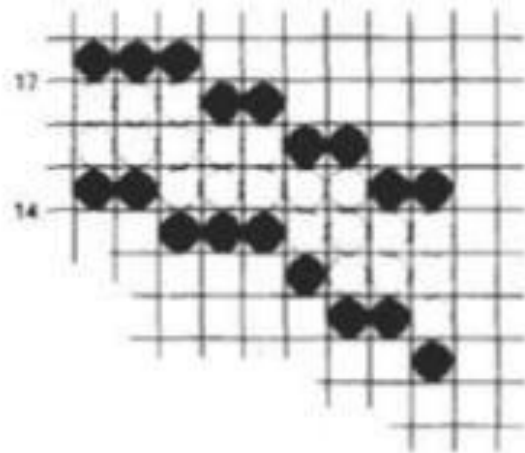


*Figure 4-12*  
A dashed circular arc displayed  
with a dash span of 3 pixels and an  
interdash spacing of 2 pixels.

14. 2D Plotting: Arcs

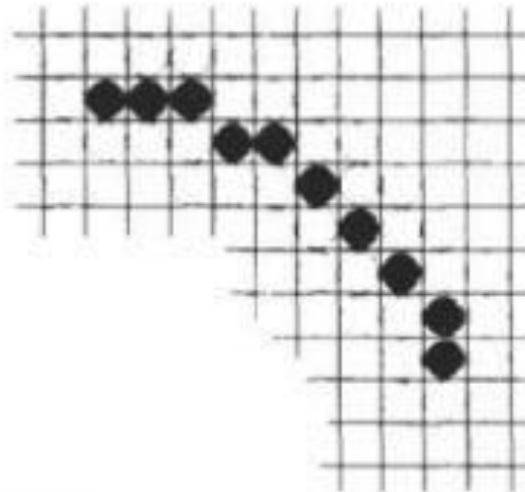


*Figure 4-13*  
Circular arc of width 4 plotted with  
pixel spans.




---

*Figure 4-14*  
A circular arc of width 4 and radius 16 displayed by filling the region between two concentric arcs.




---

*Figure 4-15*  
Circular arc displayed with a rectangular pen.



# COLOR AND GRAYSCALE LEVELS

- ▶ General purpose raster-scan systems, usually provide a wide range of colors, while random-scan monitors typically offer only a few color choices, if any.
- ▶ 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. With color plotters, the codes could control ink-jet deposits or pen selections.



- ▶ 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 Table.

**TABLE 4-1**  
THE EIGHT COLOR CODES FOR A THREE-BIT  
PER PIXEL FRAME BUFFER

<i>Color</i>	<i>Stored Color Values in Frame Buffer</i>			<i>Displayed Color</i>
<i>Code</i>	<i>RED</i>	<i>GREEN</i>	<i>BLUE</i>	
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



- ▶ 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.
- ▶ Adding more bits per pixel to the frame buffer increases the number of color choices. With 6 bits per pixel, 2 bits can be used for each gun.
- ▶ This allows four different intensity settings for each of the three color guns, and a total of 64 color values are available for each screen pixel.





- ▶ With a resolution of 1024 by 1024, a full-color (24bit per pixel) 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.
- ▶ Use of color tables to reduce frame-buffer storage requirements.

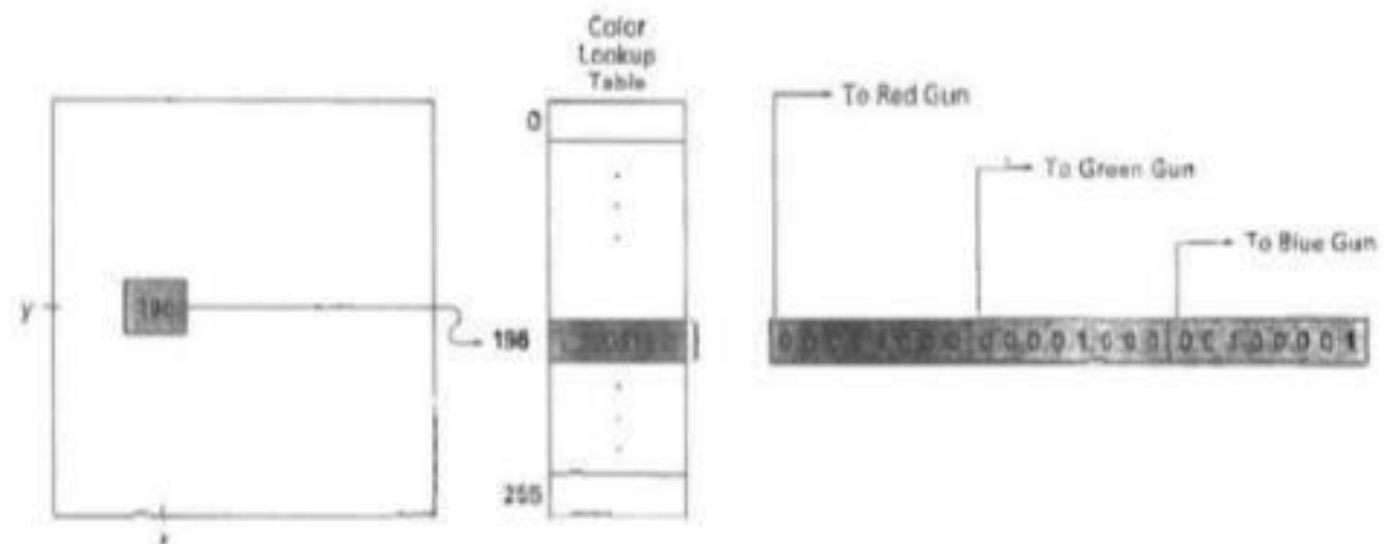


# Color Tables

- ▶ A possible scheme for storing color values in a color lookup table (or video lookup table), where frame-buffer values are now used as indices into the color table.
- ▶ A user can set color-table entries in a PHIGS applications program with the function

`setColourRepresentation (ws, ci, colorptrl)`





**Figure 4-16**

A color lookup table with 24 bits per entry accessed from a frame buffer with 8 bits per pixel. A value of 196 stored at pixel position (x, y) references the location in this table containing the value 2081. Each 8-bit segment of this entry controls the intensity level of one of the three electron guns in an RGB monitor.

- ▶ Parameter **ws** identifies the workstation output device; parameter **ci** specifies the color index, and parameter **colorptr** points to a hio of RGB color values (**r,g, b**) each specified in the range from 0 to 1.
- ▶ An example of possible table entries for color monitors is given

WS = 1		WS = 2	
ci	Color	ci	Color
0	(0, 0, 0)	0	(1, 1, 1)
1	(0, 0, 0.2)	1	(0.9, 1, 1)
.	.	2	(0.8, 1, 1)
.	.	.	.
192	(0, 0.03, 0.13)	.	.
.	.	.	.
.	.	.	.
.	.	.	.

Figure 4-17  
Workstation color tables

- ▶ There are several advantages in storing color codes in a lookup table.

1. "reasonable" number of simultaneous colors without requiring large frame buffers.
2. For most applications, 256 or 512 different colors are sufficient for a single picture.
3. Table entries can be changed at any time.
4. 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.



# 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.



- ▶ When multiple output devices are available at an installation, the same color-table interface may be used for all monitors.
- ▶ with the display intensity corresponding to a given color index  $c_i$  calculated as

$$\text{intensity} = 0.5[\min(r, g, b) + \max(r, g, b)]$$



# AREA-FILL ATTRIBUTES

- ▶ Options for **filling a defined region include a choice between a solid color or a 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.



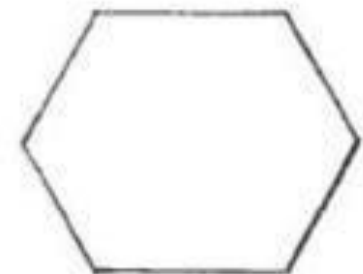


# Fill Styles

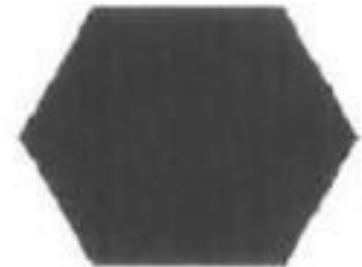
- ▶ Areas are displayed with three basic fill styles: hollow with a color border, filled with a solid color, or filled with a specified pattern or design.
- ▶ A basic fill style is selected in a **PHIGS program with the function**

```
setInteriorStyle (fs)
```

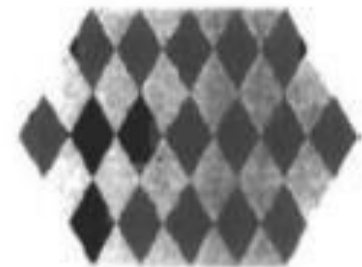
- ▶ Values for the fill-style parameter fs include hollow, solid, and pattern.



Hollow  
(a)



Solid  
(b)



Patterned  
(c)

Figure 4-18  
Polygon fill styles.

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



*Figure 4-19*  
Polygon fill using hatch patterns.

- ▶ 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

```
setInteriorColourIndex (fc)
```

- ▶ where fill color parameter *fc* is set to the desired color code.
- ▶ A polygon hollow fill is generated with a line drawing routine as a closed polyline.



- ▶ Solid fill of a region can be accomplished with the scan-line procedures.
- ▶ Other fill options include specifications for the edge type, edge width, and edge color of a region.
- ▶ These attributes are set independently of the fill style or fill color, and they provide for the same options as the line-attribute parameters



# Pattern Fill

- ▶ We select fill patterns with
- ▶ where pattern index parameter  $pi$  specifies a table position. For example, the following set of statements would fill the area defined in the fill Area command with the second pattern type stored in the pattern table:



- ▶ `setInteriorStyle ( p a t t e r n ) ;`
- ▶ `setinteriorStyleIndex ( 2 ) ;`
- ▶ `fillArea (n, points);`
  
- ▶ For fill style `pattcm`, table entries can be created on individual output devices with
  
- ▶ **SetPatternRepresentation (w s , p., 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.



- ▶ Color array  $cp$  in this example specifies a pattern that produces alternate red and black diagonal pixel lines on an eight-color system.
- ▶ 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.

**TABLE 4-3**  
A WORKSTATION  
PATTERN TABLE WITH  
TWO ENTRIES, USING  
THE COLOR CODES OF  
TABLE 4-1

<i>Index</i> <i>(p<sub>i</sub>)</i>	<i>Pattern</i> <i>(cp)</i>
1	$\begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix}$
2	$\begin{bmatrix} 2 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 1 & 2 \end{bmatrix}$

- ▶ We do this by setting the rectangular coordinate extents of the pattern:

`setpatternsize (dx, dy)`

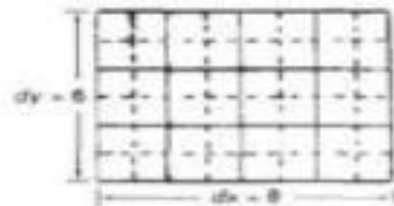


Figure 4-20  
A pattern array with 4 columns and 3 rows mapped to an 8 by 6 coordinate rectangle.

- ▶ 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 (position)`

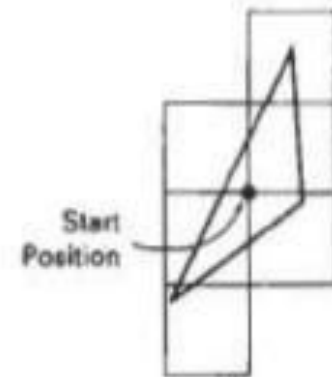




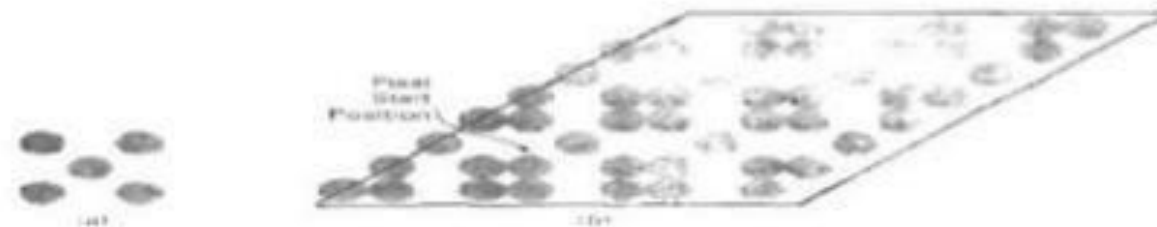
- ▶ Parameter position is a pointer to coordinates  $(x_p, y_p)$  that fix the lower left corner of the rectangular pattern.
- ▶ From this starting position, the pattern is replicated in the  $x$  and  $y$  directions until the defined area is covered by nonoverlapping copies of the pattern array.
- ▶ The process of filling an area with a rectangular pattern is called tiling and



- ▶ Rectangular fill patterns are sometimes referred to as tiling patterns.
- ▶ To illustrate the use of the pattern commands, the following program example displays a black-and-white pattern in the interior of a parallelogram fill area.



*Figure 4-21*  
Tiling an area from a designated start position. Nonoverlapping adjacent patterns are laid out to cover all scan lines passing through the defined area.



*Figure 4-22*  
A pattern array (a) superimposed on a parallelogram fill area to produce the display (b).

```
#define WS 1

void patternFill ()
{
    wcPt2 pts[4];
    int bwPattern[3][3] = { 1, 0, 0, 0, 1, 1, 1, 0, 0 };

    pSetPatternRepresentation (WS, 8, 3, 3, bwPattern);

    pts[0].x = 10; pts[0].y = 10;
    pts[1].x = 20; pts[1].y = 10;
    pts[2].x = 28; pts[2].y = 18;
    pts[3].x = 18; pts[3].y = 18;

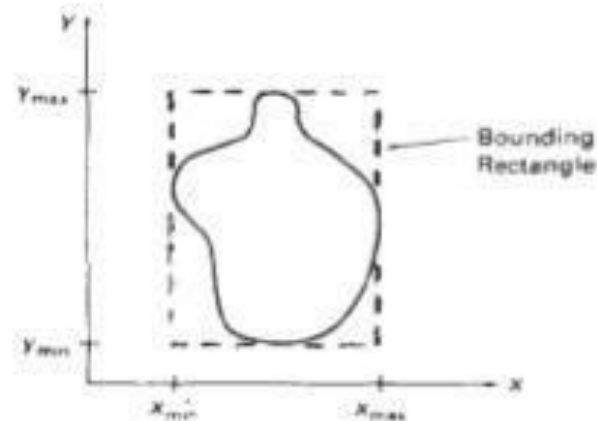
    pSetFillAreaInteriorStyle (PATTERN);
    pSetFillAreaPatternIndex (8);
    pSetPatternReferencePoint (14, 11);

    pFillArea (4, pts);
}
```

- ▶ Hatch fill is applied to regions by displaying sets of parallel lines. The fill procedures are implemented to draw either single hatching or cross hatching.
- ▶ Spacing and slope for the hatch lines can be set as parameters in the hatch table.
- ▶ On raster systems, a hatch fill can be specified as a pattern array that sets color values for groups of diagonal pixels.



- ▶ In many systems, the pattern reference point  $(x_p, y_p)$  is assigned by the system.
- ▶ For any fill region, the reference point can be chosen as the lower left corner of the bounding rectangle (or bounding box) determined by the coordinate extents of the region.



*Figure 4-23*  
Bounding rectangle for a region with coordinate extents  $x_{min}$ ,  $x_{max}$ ,  $y_{min}$ , and  $y_{max}$  in the  $x$  and  $y$  directions.

- ▶ If the row positions in the pattern array are referenced in reverse (that is, from bottom to top starting at 1), a pattern value is then assigned to pixel position  $(1, y)$  in screen or window coordinates as
- ▶ **`setpixel (x, y, cp(y mod ny + 1, x mod nx + 1))`**
- ▶ Where  $ny$  and  $nx$  specify the number of rows and number of columns in the pattern array



- ▶ The pattern and background color can be combined using Boolean operations, or the pattern color can simply replace the background colors.
- ▶ How the Boolean and replace operations for a 2 by 2 fill pattern would set pixel values on a binary (black and white) system against a particular background pattern.

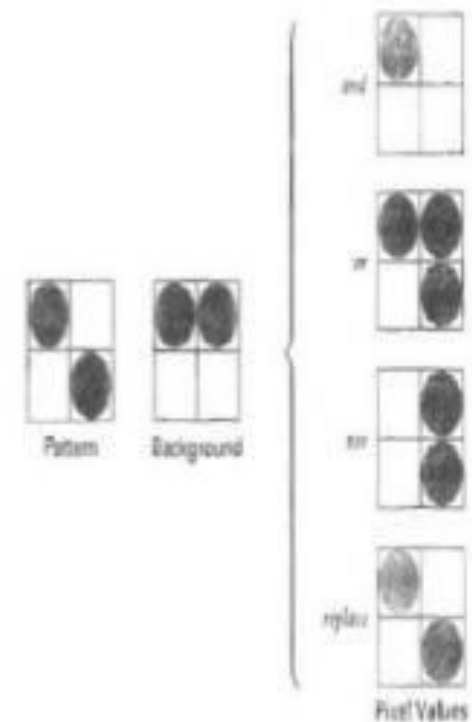


Figure 4-24  
Combining a fill pattern with a background pattern using Boolean operations, and, or, and xor (exclusive or), and using simple replacement.



# 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 .
- ▶ 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.





# 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

- ▶ 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 in 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

**setTextColorIndex(tc)**

- ▶ where text color parameter tc specifies an allowable color code.
- ▶ Character size is specified by printers and compositors in points, where 1 point is 0.013837 inch.
- ▶ Point measurements specify the size of the body of a character but different fonts with the same points specifications can have different character size depending on the design of the typeface.



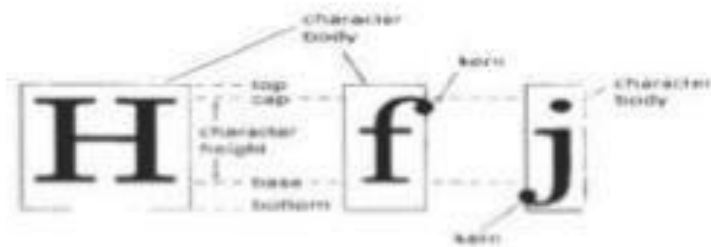


Figure 4-21  
Character body.

- ▶ The distance between the bottom line and the top line of the character body is the same for all characters in a particular size and typeface, but the body width may vary.
- ▶ Character height is defined as the distance between the baseline and the cap line of characters.

- ▶ Text size can be adjusted without changing the width-to-height ratio of characters with **setCharacterHeight(ch)**
- ▶ Parameter `ch` is assigned a real value greater than 0 to set the coordinate height of capital letters: the distance between baseline and capline in user coordinates.

Height 1  
Height 2  
Height 3

---

*Figure 4-26*  
The effect of different character-height settings on displayed text.



- ▶ The width only of text can be set with the function  
**setCharacterExpansionFactor(cw)**
- ▶ where the character-width parameter cw is set to a positive real value that scales the body width of characters.

width 0.5

width 1.0

**width 2.0**

---

*Figure 4-27*  
The effect of different  
character-width settings on  
displayed text.



- ▶ Spacing between characters is controlled separately with **setCharacterSpacing(cs)**
- ▶ where the character-spacing parameter **cs** can be assigned any real value. The value assigned to **cs** determines the spacing between character bodies along print lines.
- ▶ Negative values for **cs** overlap character bodies; positive values insert space to spread out the displayed characters.

Spacing 0.0

S p a c i n g 0.5

S p a c i n g 1.0

---

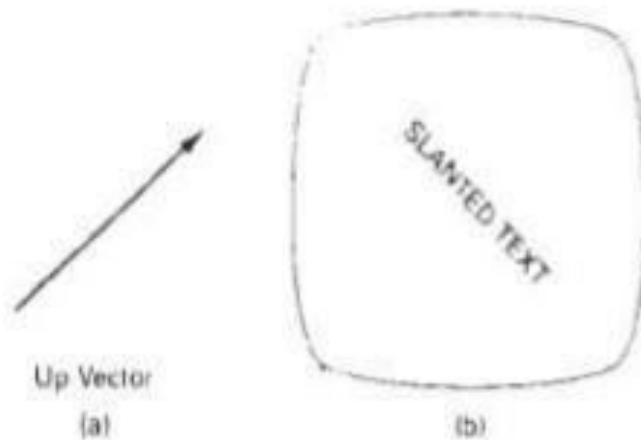
*Figure 4-28*  
The effect of different character spacings on displayed text.



- ▶ The orientation for a displayed character string is set according to the direction of the character up vector:
- ▶ **setCharacterUpVector(upvect)**
- ▶ Parameter upvect in this function is assigned two values that specify the x and y vector components.
- ▶ Text is then displayed so that the orientation of characters from baseline to capline is in the direction of the up vector.



- ▶ For example, with  $\text{upvect} = (1, 1)$ , the direction of the up vector is 45° and text would be displayed



---

*Figure 4-29*  
Direction of the up vector (a)  
controls the orientation of  
displayed text (b).



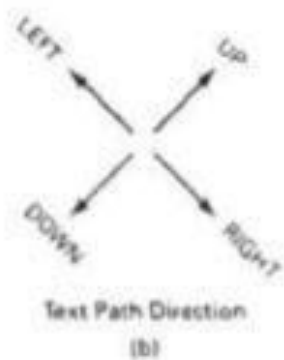



Figure 4-32  
An up-vector specification (a) controls the direction of the text path (b).

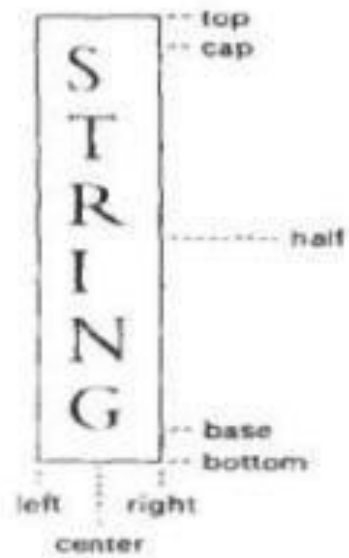


Figure 4-33  
The 45° up vector in Fig. 4-32 produces the display (a) for a *circum* path and the display (b) for a *right* path.

- ▶ It is useful in many applications to be able to arrange character strings vertically or horizontally. An attribute parameter for this option is set with the statement
  - ▶ **setTextPath(tp)**
  - ▶ where the text-path parameter tp can be assigned the value: right, left, up, or down.
  - ▶ A procedure for implementing this option must transform the character patterns into the specified orientation before transferring them to the frame buffer

Figure 4-31  
Text displayed with the four  
text-path options.

- ▶ Another handy attribute for character strings is alignment. This attribute specifies how text is to be positioned with respect to the start coordinates. Alignment attributes are set with **setAlignment(h,v)**
  - ▶ where parameters h and v control horizontal and vertical alignment.
  - ▶ Horizontal alignment is set by assigning h a value of left, centre, or right.
  - ▶ Vertical alignment is set by assigning v a value of top, cap, half, base, or bottom
- 



---

*Figure 4-34*  
Alignment attribute values for horizontal and vertical strings.


- ▶ A precision specification for text display is given with

**setTextPrecision(tpr)**

- ▶ where text precision parameter tpr is assigned one of the values: string, char, or stroke.
- ▶ The highest-quality text is displayed when the precision parameter is set to the value *stroke*



# Marker Attribute:

- ▶ A marker symbol is a single character that can be 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.
- ▶ Values greater than 1 produce character enlargement; values less than 1 reduce the marker size.



## **Inquiry functions**

Introduction: Current settings for attributes and other parameters, such as workstation types and status, in the system lists can be retrieved with inquiry functions. These functions allow current values to be copied into specified parameters, which can then be saved for later reuse or used to check the current state of the system if an error occurs.

We check current attribute values by stating the name of the attribute in the inquiry function. For example, the functions

`inquirePolyLineIndex (last li)`and

`inquireInteriorColourIndex (last fc)`

Copy the current values for line index and fill color into parameters `lastli` and `lastfc`. The following program segment illustrates reusing the current line type value after a set of lines are drawn with a new line type.

# 2D TRANSFORMATIONS COMPUTER GRAPHICS



## 2D Transformations

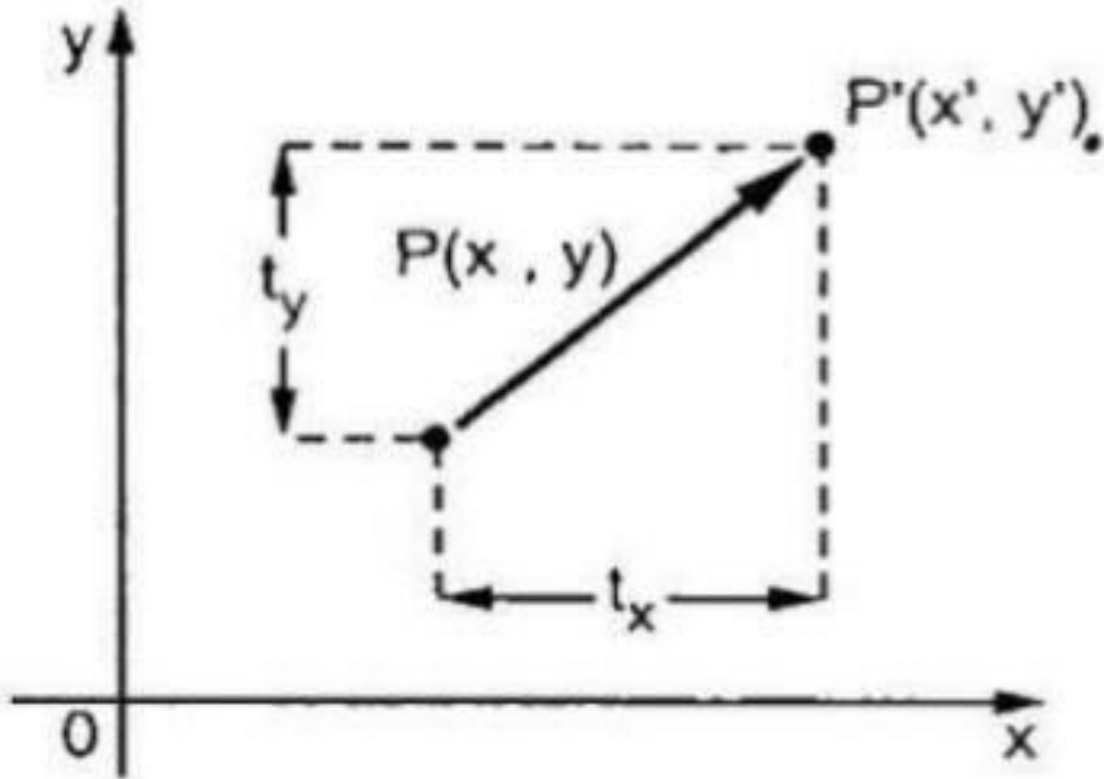
“Transformations are the operations applied to geometrical description of an object to change its position, orientation, or size are called geometric transformations”.



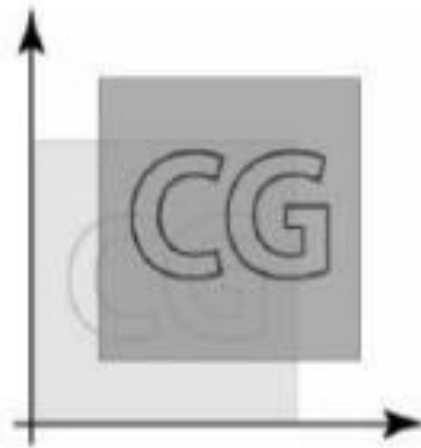
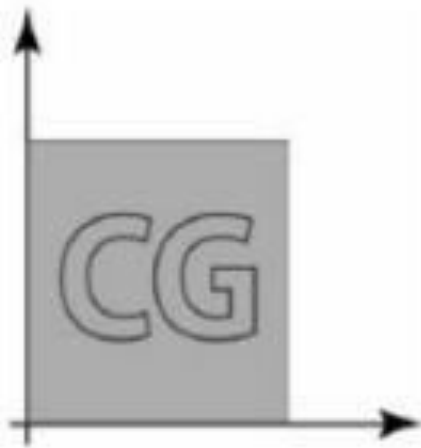
# Translation

- ❖ Translation is a process of changing the position of an object in a straight-line path from one coordinate location to another.
- ❖ We can translate a two dimensional point by adding translation distances,  $t_x$  and  $t_y$ .
- ❖ Suppose the original position is  $(x, y)$  then new position is  $(x', y')$ .
- ❖ Here  $x' = x + t_x$  and  $y' = y + t_y$ .

Click icon to add picture



# Translation



❖ Matrix form of the equations:

$X' = X + tx$  and  $Y' = Y + ty$  is

$$P = \begin{pmatrix} x \\ y \end{pmatrix} \quad P' = \begin{pmatrix} x' \\ y' \end{pmatrix} \quad T = \begin{pmatrix} tx \\ ty \end{pmatrix}$$

❖ we can write it,

$$P' = P + T$$



❖ Translate a polygon with co-ordinates A(2,5) B(7,10) and C(10,2) by 3 units in X direction and 4 units in Y direction.

$$\begin{aligned} \text{❖ } A' &= A + T \\ &= \begin{bmatrix} 2 \\ 5 \end{bmatrix} + \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 5 \\ 9 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} \text{❖ } B' &= B + T \\ &= \begin{bmatrix} 7 \\ 10 \end{bmatrix} + \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 10 \\ 14 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} \text{❖ } C' &= C + T \\ &= \begin{bmatrix} 10 \\ 2 \end{bmatrix} + \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 13 \\ 7 \end{bmatrix} \end{aligned}$$



# Rotation

- ❖ A two dimensional rotation is applied to an object by repositioning it along a circular path in the xy plane.
- ❖ Using standard trigonometric equations , we can express the transformed co-ordinates in terms of  $\theta$  and  $\phi$  as

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

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

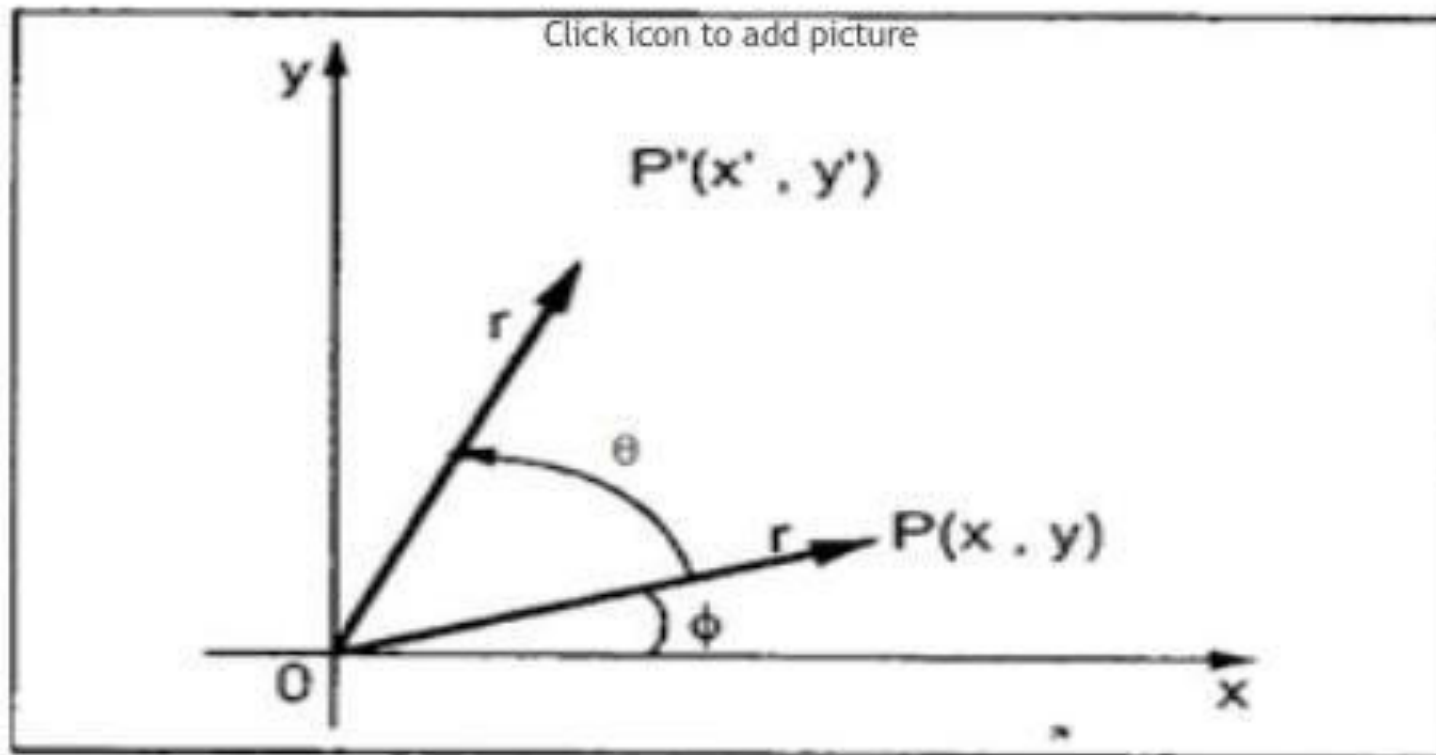
- ❖ The original co-ordinates of the point is

$$x = r \cos \phi$$

$$y = r \sin \phi$$



Click icon to add picture

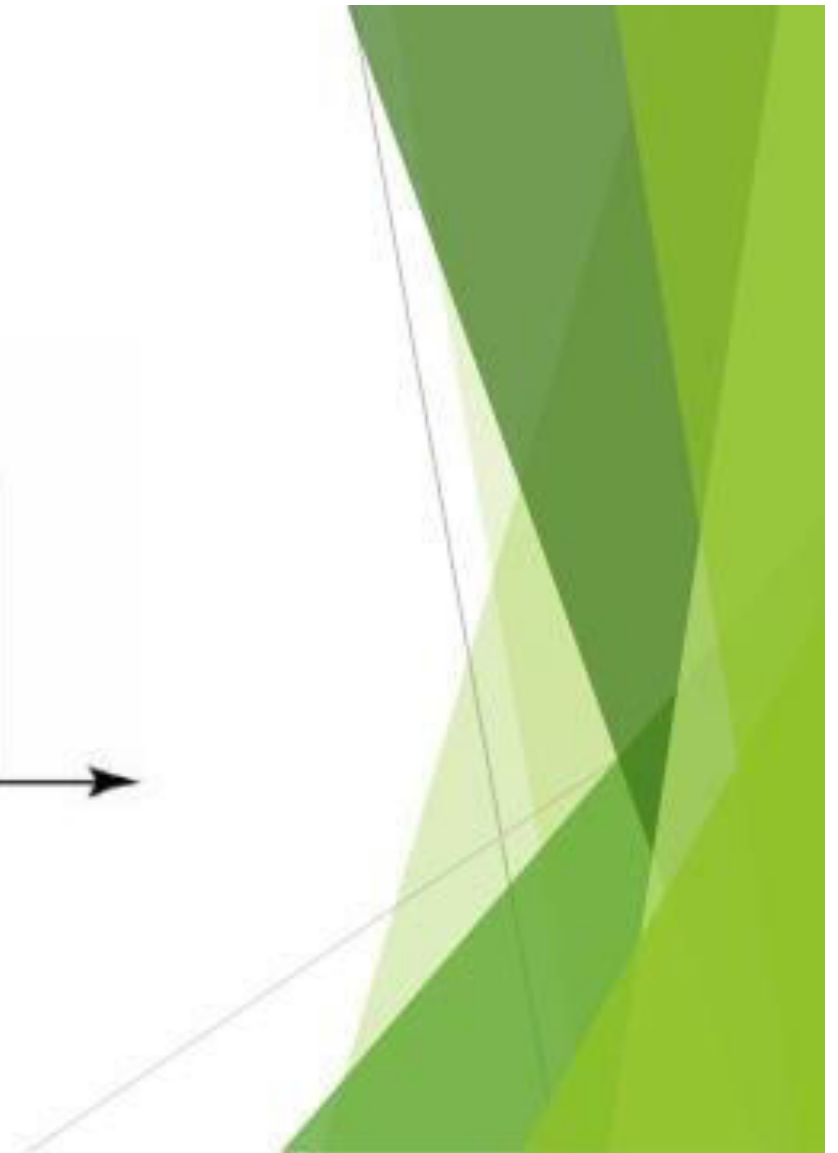
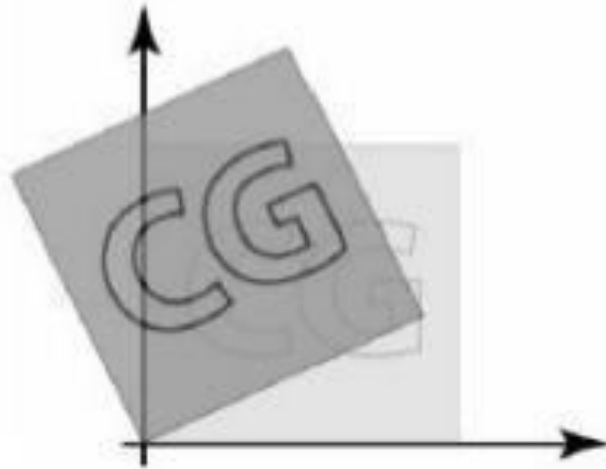
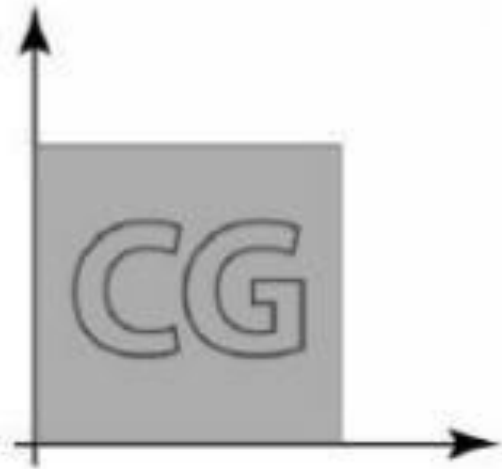


After substituting equation 2 in equation 1 we get

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

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

# Rotation



- ❖ That equation can be represented in matrix form

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

- ❖ we can write this equation as,

$$P' = P \cdot R$$

- ❖ Where R is a rotation matrix and it is given as

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



- ❖ A point (4,3) is rotated counterclockwise by angle of 45.  
find the rotation matrix and the resultant point.

$$\text{❖ } R = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} = \begin{pmatrix} \cos 45 & \sin 45 \\ -\sin 45 & \cos 45 \end{pmatrix}$$

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

$$\begin{aligned} P' &= \begin{bmatrix} 4 & 3 \end{bmatrix} \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & 1/\sqrt{2} \end{pmatrix} \\ &= \begin{bmatrix} 4/\sqrt{2} - 3/\sqrt{2} & 4/\sqrt{2} + 3/\sqrt{2} \end{bmatrix} \\ &= \begin{bmatrix} 1/\sqrt{2} & 7/\sqrt{2} \end{bmatrix} \end{aligned}$$



# Scaling

- ❖ A scaling transformation changes the size of an object.
- ❖ This operation can be carried out for polygons by multiplying the co-ordinates values  $(x, y)$  of each vertex by scaling factors  $S_x$  and  $S_y$  to produce the transformed co-ordinates  $(x', y')$ .

$$x' = x \cdot S_x$$

$$y' = y \cdot S_y$$

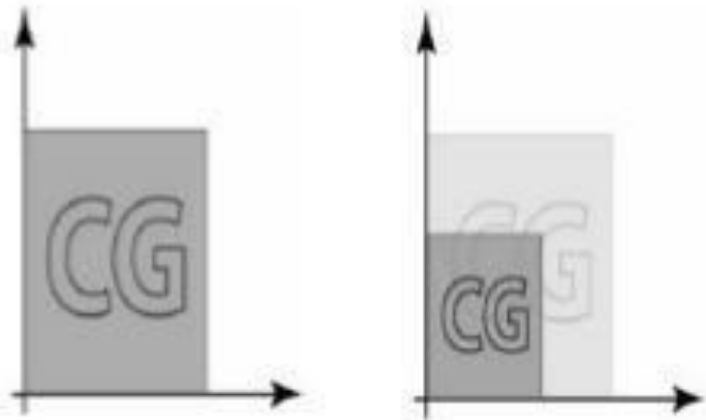
- ❖ In the matrix form

$$\begin{aligned} [x' \ y'] &= [x \ y] \begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix} \\ &= P \cdot S \end{aligned}$$

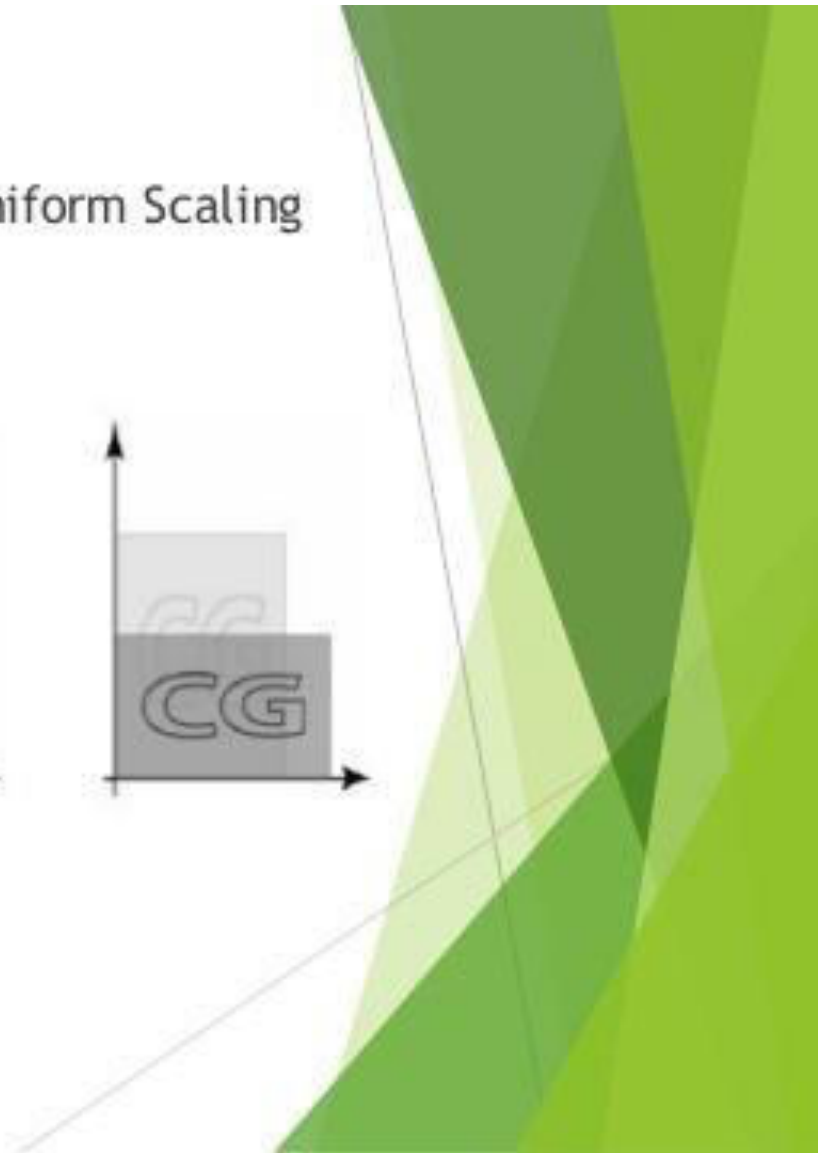
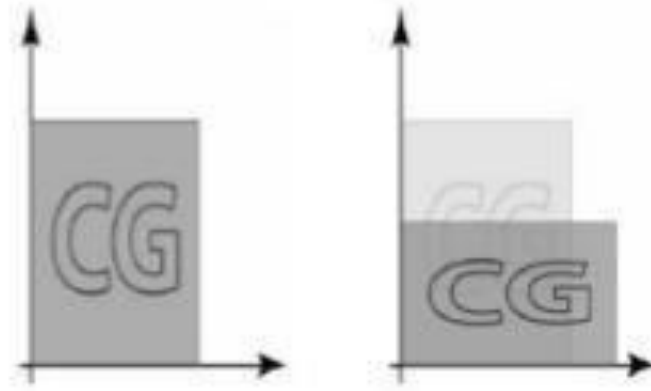


# Scaling

- Uniform Scaling



- Un-uniform Scaling





## Homogeneous co-ordinates for Translation

- ❖ The homogeneous co-ordinates for translation are given as

$$T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ tx & ty & 1 \end{pmatrix}$$

- ❖ Therefore, we have

$$\begin{aligned} [x' \ y' \ 1] &= [x \ y \ 1] \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ tx & ty & 1 \end{pmatrix} \\ &= [x + tx \ y + ty \ 1] \end{aligned}$$

## Homogeneous co-ordinates for rotation

- ❖ The homogeneous co-ordinates for rotation are given as

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

- ❖ Therefore, we have

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

## Homogeneous co-ordinates for scaling

- ❖ The homogeneous co-ordinate for scaling are given as

$$S = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- ❖ Therefore , we have

$$\begin{aligned} [x' \ y' \ 1] &= [x \ y \ 1] \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ &= [x \cdot S_x \ y \cdot S_y \ 1] \end{aligned}$$



# Composite Transformations

## (A) 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: -

$$\begin{aligned} P' &= T(t_{x2}, t_{y2}) \cdot \{T(t_{x1}, t_{y1}) \cdot P\} \\ &= \{T(t_{x2}, t_{y2}) \cdot T(t_{x1}, t_{y1})\} \cdot P \end{aligned}$$

Where  $P$  and  $P'$  are represented as homogeneous-coordinate column vectors. We can verify this result by calculating the matrix product for the two associative groupings. Also, the composite transformation matrix for this sequence of transformations is: -

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

Or,  $T(t_{x2}, t_{y2}) \cdot T(t_{x1}, t_{y1}) = T(t_{x1}+t_{x2}, t_{y1}+t_{y2})$

Which demonstrate that two successive translations are additive.



## (B) Rotations

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

$$\begin{aligned} P' &= R(\Theta_2) \cdot \{R(\Theta_1) \cdot P\} \\ &= \{R(\Theta_2) \cdot R(\Theta_1)\} \cdot P \end{aligned}$$

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

$$R(\Theta_2) \cdot R(\Theta_1) = R(\Theta_1 + \Theta_2)$$

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

$$P' = R(\Theta_1 + \Theta_2) \cdot P$$



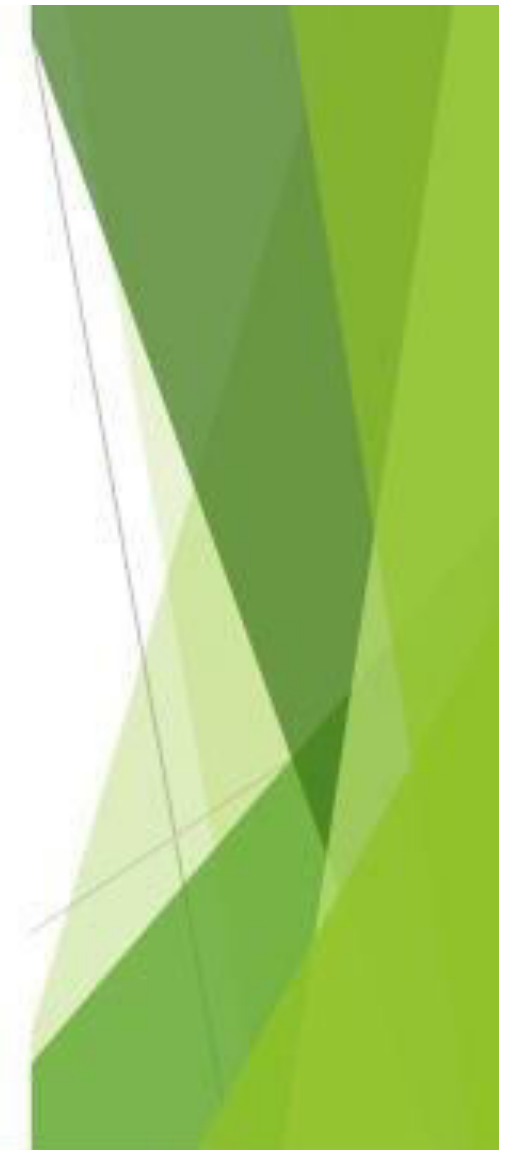
## (C) Scaling

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

$$\begin{vmatrix} S_{x2} & 0 & 0 \\ 0 & S_{y2} & 0 \\ 0 & 0 & 1 \end{vmatrix} \cdot \begin{vmatrix} S_{x1} & 0 & 0 \\ 0 & S_{y1} & 0 \\ 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} S_{x1} \cdot S_{x2} & 0 & 0 \\ 0 & S_{y1} \cdot S_{y2} & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

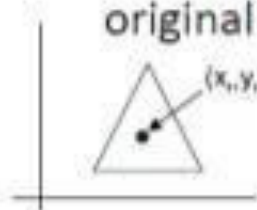
Or,  $S(S_{x2}, S_{y2}) \cdot S(S_{x1}, S_{y1}) = S(S_{x1} \cdot S_{x2}, S_{y1} \cdot S_{y2})$

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



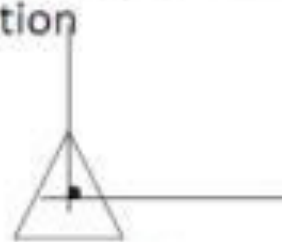
# General pivot point rotation

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



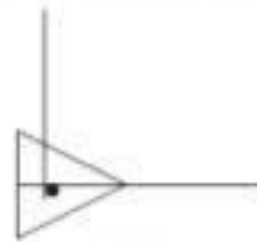
(a)

Original Position  
of Object and  
pivot point



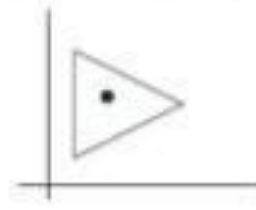
(b)

Translation of  
object so that  
pivot point  $(x_r, y_r)$   
is at origin



(c)

Rotation was  
about origin



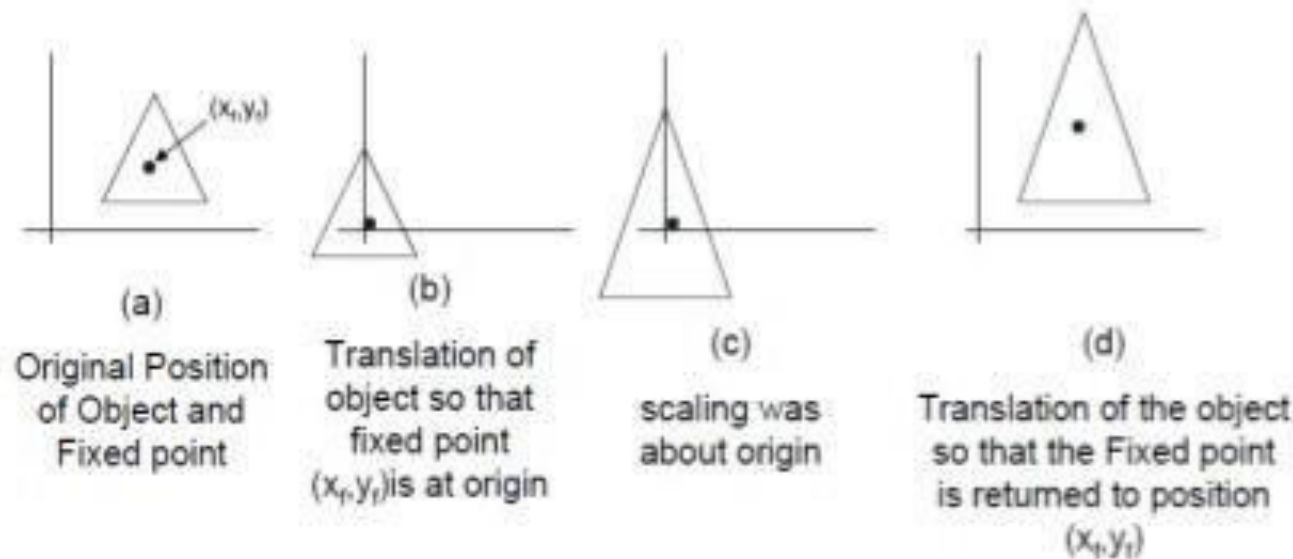
(d)

Translation of the object  
so that the pivot point is  
returned to position  
 $(x_r, y_r)$



# General fixed point scaling

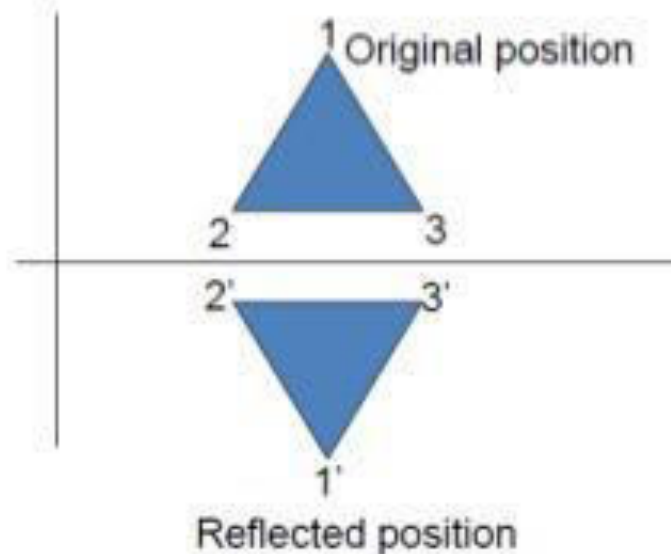
- Translate object so that the fixed point coincides with the coordinate origin
- Scale the object with respect to the coordinate origin
- Use the inverse translation of step 1 to return the object to its original position





## Other transformations

- **Reflection** is a transformation that produces a mirror image of an object. It is obtained by rotating the object by 180 deg about the reflection axis

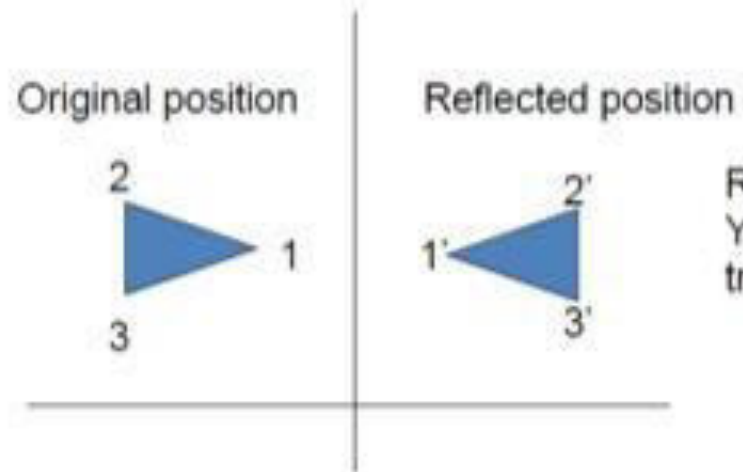


Reflection about the line  $y=0$ , the X- axis , is accomplished with the transformation matrix

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



# Reflection

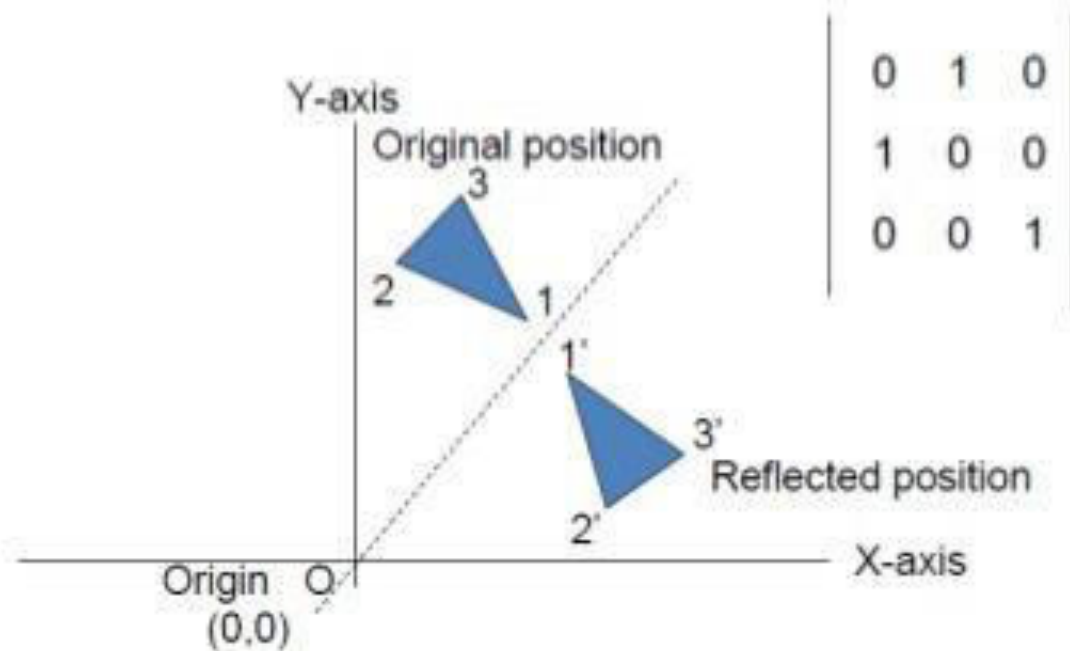


Reflection about the line  $x=0$ , the Y- axis , is accomplished with the transformation matrix

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



## Reflection of an object w.r.t the straight line $y=x$

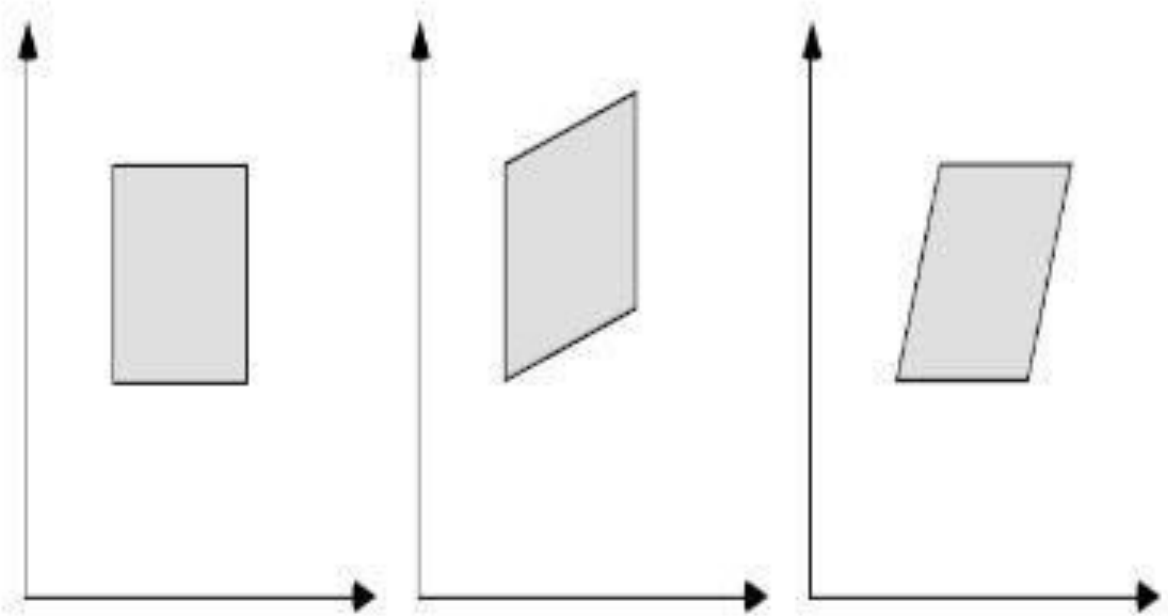


# Shear Transformations

- Shear is a transformation that distorts 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
- Two common shearing transformations are those that shift coordinate x values and those that shift y values



# Shears



Original Data

y Shear

x Shear

