

COMPUTER GRAPHICS

UNIT – I

A SURVEY OF COMPUTER GRAPHICS

Nowadays, Computer Graphics are used routinely in various areas as Science, Engineering, Medicine, Business, Industry, Government, Art, Entertainment, Advertising, Education and Training.

Some of the Graphics applications are as follows:

Computer-Aided Design:

A major use of Computer Graphics is in Design Processes, particularly for engineering and architectural systems. **CAD (Computer-Aided Design)** methods are used in the design of buildings, automobiles, aircraft, watercraft, spacecraft, computers, textiles and many other products.

In this,

- ✓ Objects are first displayed in a **Wireframe** outline form that shows the overall shape and internal features of objects. It also allows the designers to quickly see the effects to interactive adjustment to design shapes.
- ✓ Software packages for CAD applications typically provide the designer with a multi-window environment.
- ✓ Animations are often used in CAD applications. Real-time animations using wireframe displays on a video monitor are useful for testing performance of a vehicle or system.
- ✓ When object designs are complete, or nearly complete, realistic lighting models and surface rendering are applied to produce displays that will show the appearance of the final product.
- ✓ Architects use interactive graphics methods to layout floor plans, such as positioning of rooms, doors, windows, stairs, shelves, counters and other building features, electrical outlets, etc.
- ✓ Realistic displays of architectural designs permit both architects and their clients to study the appearance of buildings. Many other kinds of systems and products are designed using either general CAD packages or specially developed CAD software.

Presentation Graphics:

- ✓ Presentation Graphics is used to produce illustrations for reports or to generate 35-mm slides or transparencies for use with projectors.
- ✓ It is commonly used to summarize financial, statistical, mathematical, scientific and economic data for research reports, managerial reports, consumer information bulletins and other types of reports.
- ✓ Typical examples of presentation graphics are bar charts, line graphs, surface graphs, pie charts and other displays showing relationships between multiple parameters.

Computer Art:

Computer Graphics methods are widely used in both fine art and commercial art applications. Artists use a variety of computer methods, including special-purpose hardware, artists' paint brush programs, other paint packages, specially developed software, symbolic mathematics packages, CAD packages, desktop publishing software and animation packages that provide facilities for designing object shapes and specifying object motions.

- ✓ The artist uses a combination of three-dimensional modeling packages, texture mapping, drawing programs and CAD software.
- ✓ There is a Pen Plotter with specially designed software that can create “automatic art” without intervention from the artist.
- ✓ Another art is “mathematical” art, in which the artist uses a combination of mathematical functions, fractal procedures, mathematical software, ink-jet printers and other systems to create a variety of three-dimensional and two-dimensional shapes and stereoscopic image pairs.
- ✓ A common graphics method employed in many commercial is morphing, where one object is transformed into another. This method has been used in TV commercials such as logo design, advertising, etc.

Entertainment:

Computer Graphics methods are now commonly used in making motion pictures, music videos and television shows. Sometimes the graphics scenes are displayed by themselves and sometimes graphics objects are combined with the actors and live scenes.

Music Videos use graphics in several ways. Graphics objects can be combined with the live action, or graphics and image processing techniques can be used to produce morphing effects.

Many TV series regularly employ Computer Graphics methods.

Education and Training:

Computer-generated models of physical, financial and economic systems are often used as educational aids. Models of physical systems, physiological systems, population trends or equipment such as the color-coded diagram can help trainers to understand the operation of the system.

For some training applications, special systems are designed. Examples of such specialized systems are the simulators for practice sessions or training of ship captains, aircraft pilots, heavy-equipment operators and air traffic-control personnel.

Some simulators have no video screens, but most simulators provide graphics screens for visual operation.

For example, in an automobile-driving simulator, it is used to investigate the behavior of drivers in critical situations.

Visualization:

Producing graphical representations for scientific, engineering and medical data sets and processes is generally referred to as **Scientific Visualization**. The term **Business Visualization** is used in connection with data sets related to commerce, industry and other nonscientific areas.

There are many different kinds of data sets and effective visualization schemes depend on the characteristics of the data. Color coding is one way to visualize a data set.

Additional techniques include contour plots, graphs and charts, surface rendering and visualizations of volume interiors. In addition, Image Processing techniques are combined with Computer Graphics to produce many of the data visualization.

Image Processing:

Image Processing is a technique used to modify or interpret existing pictures, such as photographs and TV scans. Two principal applications of Image Processing are,

- i) improving picture quality
- ii) machine perception of visual information as used in Robotics.

To apply image-processing methods, we first digitize a photograph or other picture into an image file. Then digital methods can be applied to rearrange picture parts, to enhance color separations, or to improve the quality of shading.

These techniques are used extensively in commercial art applications that invoke the retouching and rearranging the sections of photographs and other artwork. Similar methods are used to analyze satellite photos of the earth and photos of galaxies.

Image Processing and Computer Graphics are typically combined in many applications. For example, in Medical field this technique is used to model and study physical functions, to design artificial limbs and to plan and practice surgery. This application is generally referred as **Computer-Aided Surgery**.

Graphical User Interfaces:

Nowadays all the software packages provide a **Graphical Interface**. A major component of a graphical interface is a window manager that allows a user to display multiple-window areas. Each window can contain a different process that can contain graphical or non-graphical displays.

Interfaces also display menus and icons for fast selection of processing operations and parameter values.

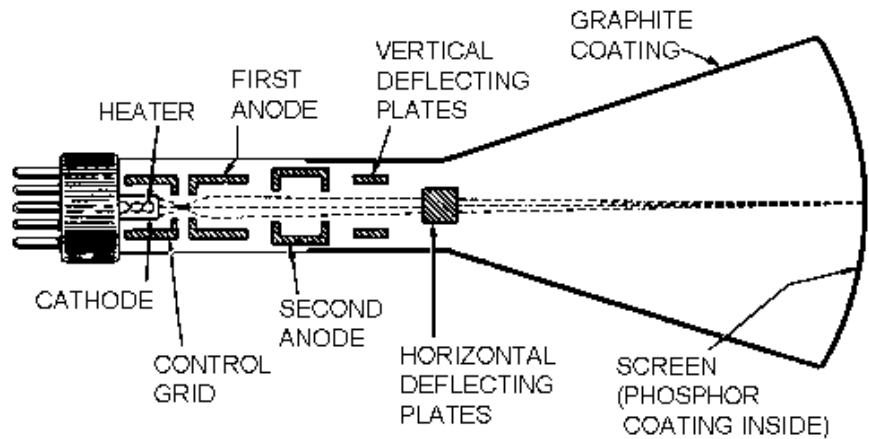
OVERVIEW OF GRAPHICS SYSTEM : VIDEO DISPLAY DEVICES

The primary output device in a graphics system is a **Video Monitor**. The functionality of most video monitors is based on the standard **Cathode-Ray Tube (CRT)** design.

Refresh Cathode Ray Tubes:

A beam of electrons (cathode rays) emitted by an electron gun, passes through focusing and deflection systems that direct the beam toward specified positions on the phosphor-coated screen. The phosphor then emits a small spot of light at each position contacted by the electron beam.

The phosphor glowing is to redraw the picture repeatedly by quickly directing the electron beam back over the same points. This type of display is called a **Refresh CRT**.



- The primary components of an electron gun in a CRT are the heated metal cathode and a control grid. Heat is supplied to the cathode by directing a current through a coil of wire, called the *filament*, inside the cylindrical cathode structure.
- The focusing system in a CRT is needed to force the electron beam to converge into a small spot as it strikes the phosphor.
- As with focusing, deflection of the electron beam can be controlled either with electric fields or with magnetic fields. One pair of plates is mounted horizontally to control the vertical deflection, and the other pair is mounted vertically to control horizontal deflection.
- Different kinds of phosphor are available for use in a CRT. Besides, color a major difference between phosphor is either **persistence**, how long they continue to emit light after the CRT beam is removed.
- The maximum number of points that can be displayed without overlap on a CRT is referred to as the **Resolution**.
- Another property of video monitor is **aspect ratio**, which gives the ratio of vertical points to horizontal points necessary to produce equal-length lines in both directions on the screen.

Raster-Scan Displays:

The most common type of graphics monitor employing a CRT is the **Raster-Scan** Display, based on television technology.

In Raster-Scan systems, the electron beam is swept across the screen, one row at a time from top to bottom. As the electron beam moves across each row, the beam intensity is turned on and off to create a pattern of illuminated spots.

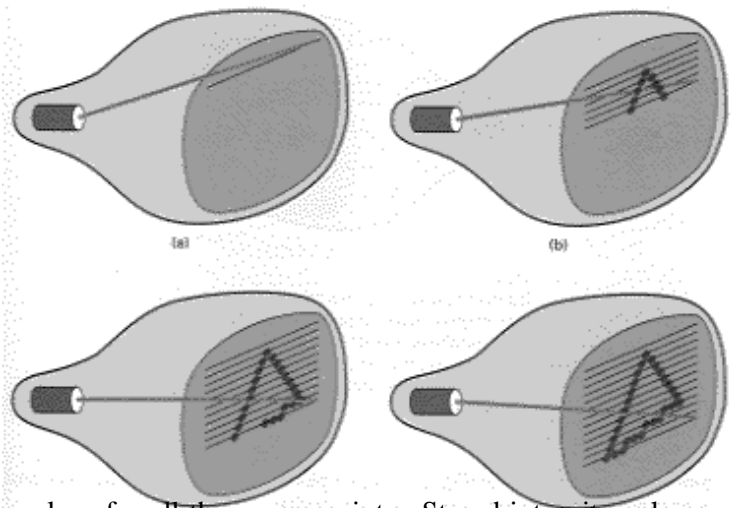
Picture definition is stored in a memory area called the **Refresh Buffer** or **Frame Buffer**.

This memory area holds the set of intensity values for all the screen points. Stored intensity values are then retrieved from the refresh buffer and “painted” on the screen one row (scan line) at a time. Each screen point is referred to as a **Pixel** or **Pel (Picture Element)**.

On a Black-and-White system with one bit per pixel, the frame buffer is commonly called a **Bitmap**. For systems with multiple bits per pixel, the frame buffer is often referred to as a **Pixmap**.

At the end of each scan line, the electron beam returns to the left side of the screen to begin displaying the next scan line.

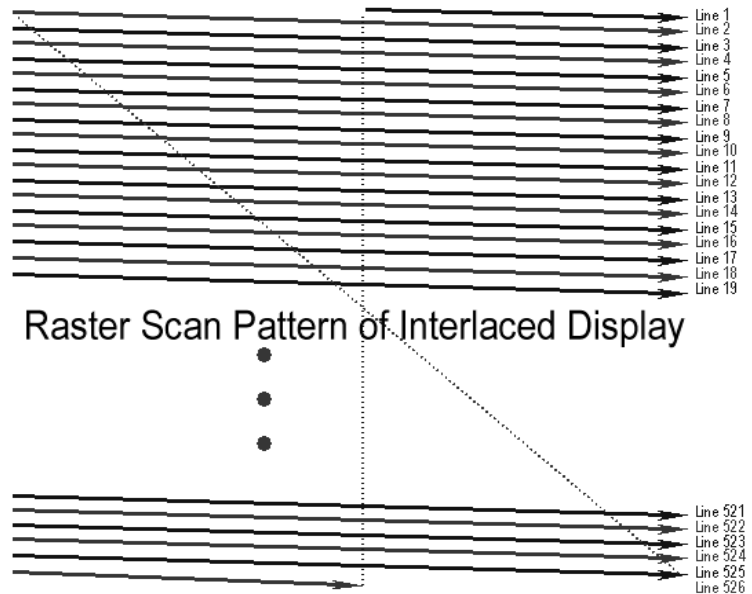
The return to the left of the screen, after refreshing each scan line, is called the **Horizontal Retrace** of the electron beam. And at the end of each frame (displayed in $1/80^{\text{th}}$ to $1/60^{\text{th}}$ of a second), the electron beam returns (vertical retrace) to the top left corner of the screen to begin the next frame.



On some raster-scan systems (and in TV sets), each frame is displayed in two passes using an **INTERLACED** refresh procedure. In the first pass, the beam sweeps across each other scan line from top to bottom. Then after the vertical retrace, the beam sweeps out the remaining scan line.

Interlacing of the scan line in this way allows us to see the entire screen displayed in one-half the time it would have taken to sweep across all the lines at once from top to bottom.

This is an effective technique for avoiding flicker, providing that adjacent scan lines contain similar display information.



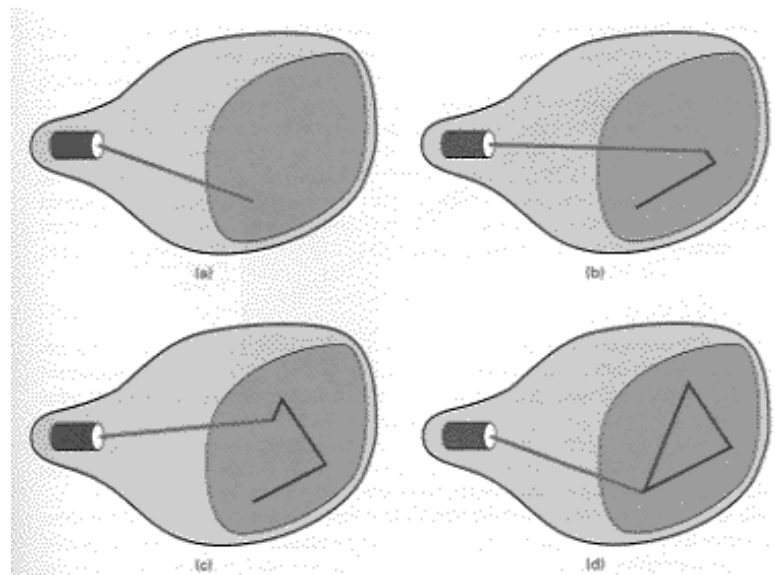
Random-Scan Displays:

When operated as a **Random-Scan** display unit, a CRT has the electron beam directed only to the parts of the screen where a picture is to be drawn.

Random-Scan monitors draw a picture one line at a time and for this reason are also referred to as **Vector Displays** (or stroke-writing or calligraphic displays). The component lien of a picture can be drawn and refreshed by a random-scan system in any specified order. A Pen Plotter operates in a similar way and is an example of a random-scan, hard copy device.

Refresh rate of a random-scan system depends on the number of lines to be displayed. Picture definition is now stored as a set of line-drawing commands in an area of memory referred to as the **Refresh Display File**. It is also called as **Display List**, **Display Program**, **Refresh Buffer**.

Random-Scan systems are designed for line-drawing applications and cannot display realistic shaded scenes.



Color CRT Monitors:

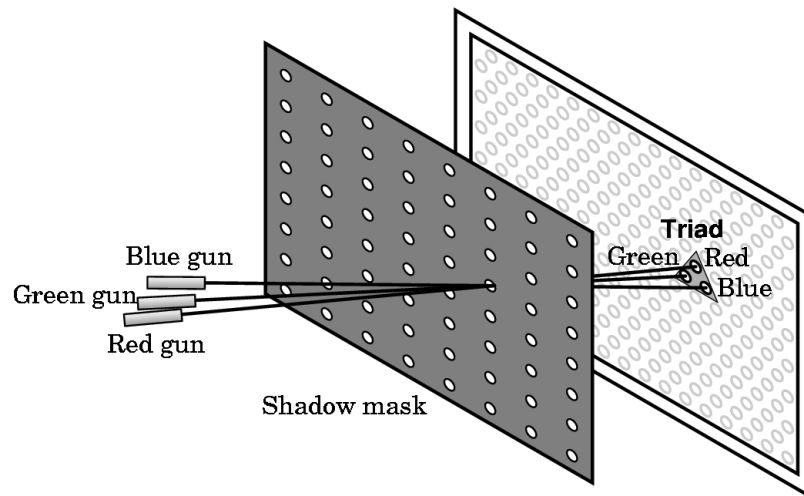
A CRT monitor displays color pictures by using a combination of phosphors that emit different colored light. By combining the emitted light from the different phosphors, a range of colors can be generated.

The two basic techniques for producing color displays with a CRT are the **Beam-Penetration** method and the **Shadow-Mask** method.

✎ The **Beam-Penetration** method for displaying color pictures has been used with *random-scan* monitors.

- ➔ Two layers of phosphor, usually red and green, are coated onto the inside of the CRT screen, and are displayed color depends on how far the electron beam penetrates into the phosphor layers.
- ➔ At intermediate beam speeds, combinations of red and green light are emitted to show two additional colors, orange and yellow.
- ➔ Since only 4 colors are possible, the quality of pictures is not as good as with other methods.

- ↳ *Shadow-Mask* methods are commonly used in *raster-scan* systems because they produce a much wider range of colors than the beam-penetration method.



- ➔ A Shadow-Mask CRT has three phosphor color dots at each pixel position.
- ➔ One phosphor dot emits a red light, another emits a green light and the third emits a blue light.
- ➔ When the 3 beams pass through a hole in the shadow mask, they activate a dot triangle, which appears as a small color spot on the screen.
- ➔ In some low-cost systems, the electron beam can only be set to on or off, limiting displays to 8 colors.
- ➔ More sophisticated systems can set intermediate intensity levels for the electron beams, allowing several million different colors to be generated.

Color CRTs in graphics systems are designed as **RGB Monitors**. An RGB color system with 24 bits of storage per pixel is generally referred to as a **full-color system** or a **true-color system**.

Direct-View Storage Tubes (DVST):

A DVST stores the picture information as a charge distribution just behind the phosphor-coated screen. Two electron guns are used in a DVST. One, the primary gun, is used to store the picture pattern, the second, the flood gun, maintains the picture display.

Advantage: Since no refreshing is needed, very complex pictures can be displayed at very high resolutions without flicker.

Disadvantage: They ordinarily do not display color and that selected parts of a picture cannot be erased.

Flat-Panel Displays:

The term Flat-Panel display refers to a class of video displays that have reduced volume, weight and power requirements compared to a CRT. Example: small TV monitors, laptop, an advertisement board in elevators, etc.

Flat-Panel displays are categorized into **Emissive** and **Nonemissive** displays.

- The **Emissive** displays (or emitters) are devices that convert electrical energy into light. Eg. Plasma panels, light-emitting diode, etc.
- **Nonemissive** displays (or nonemitters) use optical effects to convert sunlight or light from some other source into graphics patterns. The most important example is a liquid-crystal device.

Emissive Devices:

- ↳ **Plasma Panels**, are also called **Gas-Discharge Displays**, are constructed by filling the region between two glass plates with a mixture of gases that usually includes neon. A series of vertical conducting ribbons is placed on one glass panel, and a set of horizontal ribbon is built into the other hand.
- ↳ **Thin-Film Electroluminescent Displays** are similar in construction to a Plasma Panel. The difference is that the region between the glass plates is filled with a phosphor, such as zinc sulfide doped with manganese, instead of a gas.

- ➔ **LED** is a matrix of diodes arranged to form the pixel positions in the display and picture definition is stored in a refresh buffer.

Nonemissive Devices:

- ➔ **Liquid-Crystal Displays (LCDs)** are commonly used in small systems, such as calculators and portable laptop computers. They produce a picture by passing polarized light from the surroundings or from an internal light source through a liquid-crystal material that can be aligned to either block or transmit the light. Two types – Passive Matrix and Active Matrix LCDs.

Three-Dimensional Viewing Devices:

They are devised using a technique that reflects a CRT image from a vibrating, flexible mirror. For example, in Medical applications it is used to analyse data from Ultrasonography, in Geological applications to analyse topological data.

Stereoscopic and Virtual-Reality Systems:

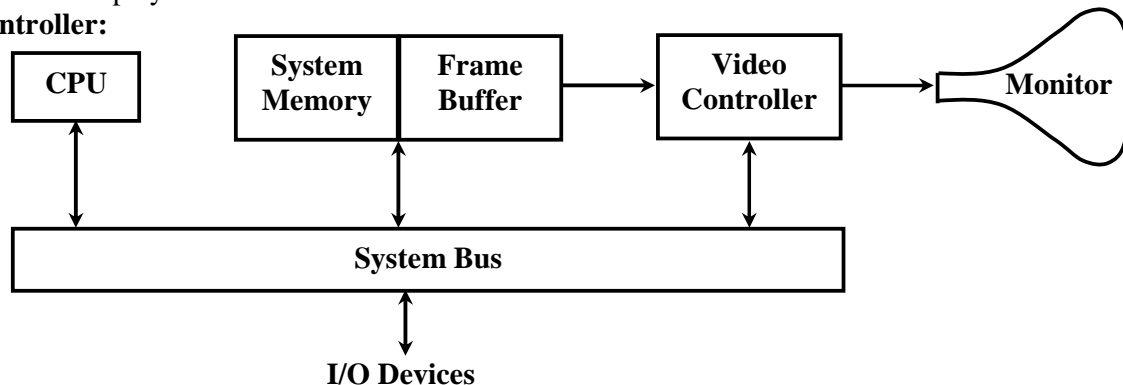
To represent objects in displaying **Stereoscopic Views**. This method does not produce true three-dimensional images, but it does provide a three-dimensional effect by presenting a different view to each eye of an observer so that scenes do appear to have depth.

Stereoscopic viewing is also a component in **Virtual-Reality Systems**, where users can step into a scene and interact with the environment.

RASTER-SCAN SYSTEMS

Interactive Raster graphics systems typically employ several processing units. In addition to the CPU, a special-purpose processor called the **Video Controller** or **Display Controller** is used to control the operation of the display device.

Video Controller:

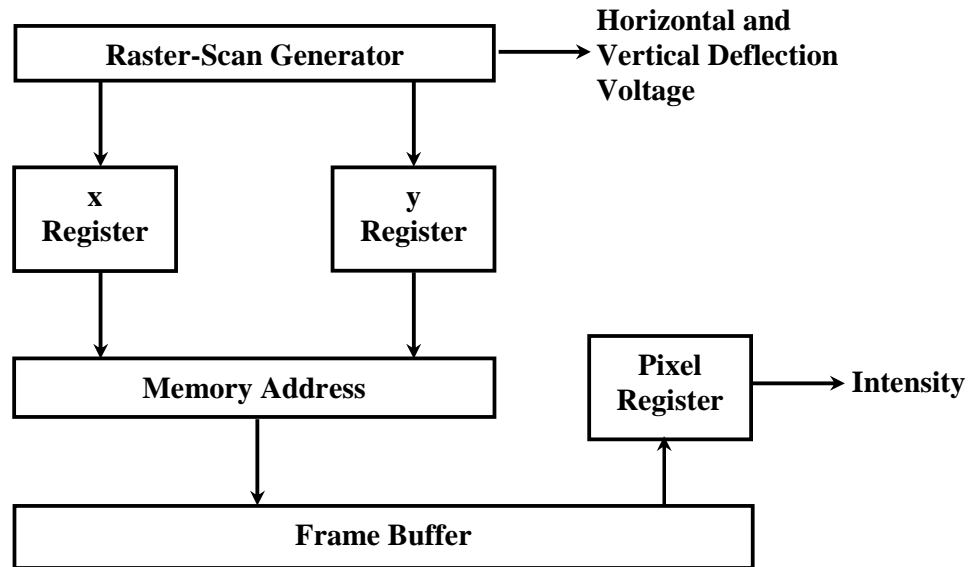


Architecture of a Raster System

A fixed area of the system memory is reserved for the frame buffer, and the Video Controller is given direct access to the Frame-Buffer Memory.

Frame-Buffer locations and the corresponding screen positions are referred in Cartesian Coordinates. For many graphics monitors, the coordinate origin is defined at the lower left screen corner. On some PC, origin is at upper left corner of the screen.

The basic refresh operations of the Video Controller are diagrammed below:



Basic Video-Controller Refresh Operations

- ★ Two registers are used to store the coordinates of the screen pixels. Initially, x register is set to 0 and y is set to y_{max} . The value stored in the frame buffer for this pixel position is then retrieved and used to set the intensity of the CRT beam.
- ★ Then the x register is incremented by 1 and the process repeated for the next pixel on the top scan line. This procedure is repeated for each pixel along the scan line.
- ★ After the last pixel on the top scan line has been processed, the x register is reset to 0 and y register is decremented by 1.
- ★ After cycling through all pixels along the bottom scan line ($y = 0$), the Video Controller resets the registers to the first pixel position on the top scan line and the refresh process starts again.
- ★ Since the screen must be refreshed at the rate of 60 frames per second, the cycle time is too slow, this can't be accommodated by typical RAM chips. To speed up pixel processing, Video Controllers can retrieve multiple pixel values from the refresh buffer on each pass.

When that group of pixels has been processed, the next block of pixel values is retrieved from the frame buffer.

In high-quality systems, 2 frame buffers are often provided so that one buffer can be used for refreshing while the other is being filled with intensity values.

Raster-Scan Display Processor:

Display Processor is also referred as a **Graphics Controller** or a **Display Coprocessor**. The purpose of the display processor is to free the CPU from the graphics chores. In addition to the system memory, a separate display processor memory area can also be provided.

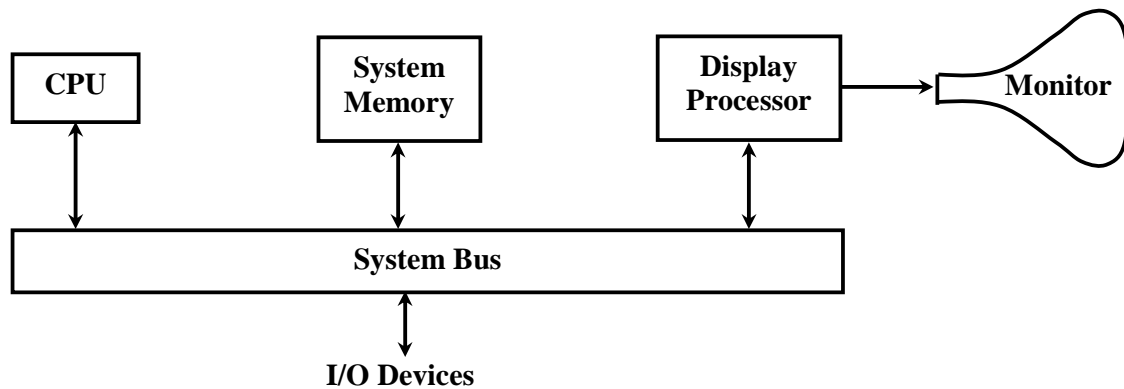
A major task of the display processor is digitizing a picture definition given in an application program into a set of pixel-intensity values for storage in the frame buffer. This digitization process is called **Scan Conversion**.

Graphics commands specifying straight lines and other geometric objects are scan converted into a set of discrete intensity points. Similar methods are used for scan converting curved lines and polygon outlines. Characters can be defined with rectangular grids or they can be defined with curved outlines. The array size for character grids can vary from about 5 by 7 to 9 by 12 or more for higher-quality displays.

Display Processors are also designed to perform a number of additional operations. These functions include generating various line styles (dashed, dotted or solid), displaying color areas and performing certain transformations and manipulations on displayed objects.

Also, display processors are typically designed to interface with interactive input devices, such as a mouse.

RANDOM-SCAN SYSTEMS



An application program is input and stored in the system memory along with a graphics package. Graphics commands in the application program are translated by the graphics package into a display file stored in the system memory. This display file is then accessed by the display processor to refresh the screen. The display processor cycles through each command in the display file program once during every refresh cycle. Sometimes the display processor in a Random-Scan system is referred to as a **Display Processing Unit** or a **Graphics Controller**.

Graphics patterns are drawn on a random-scan system by directing the electron beam along the component lines of the picture. Lines are defined by the values for their coordinate endpoints, and these input coordinate values are converted to x and y deflection voltages.

INPUT DEVICES:

Various devices are available for data input on graphics workstations. Most systems have a keyboard and one or more additional devices include a mouse, trackball, spaceball, joystick, digitizers, dials, and button boxes. Some other input devices used in particular applications are data gloves, touch panels, image scanners, and voice systems.

Keyboards:

The Keyboard is an efficient device for inputting nongraphic data as picture labels, i.e., for entering text strings. Cursor-control keys, Numeric pad and function keys are common features on general-purpose keyboards.

Other types of cursor-positioning devices, such as a trackball or joystick, are included on some keyboards.

For specialized applications, input to a graphics application may come from a set of buttons, dials, or switches that select data values or customized graphics operations.

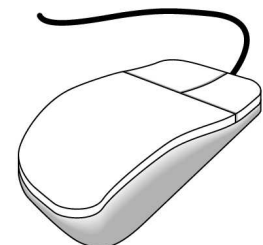
Buttons and switches are often used to input predefined functions, and dials are common devices for entering scalar values.

Real numbers within some defined range are selected for input with dial rotations. Potentiometers are used to measure dial rotations, which are then converted to deflection voltages for cursor movement.

Mouse:

A mouse is small hand-held box used to position the screen cursor. Wheels or rollers on the bottom of the mouse can be used to record the amount and direction of movement.

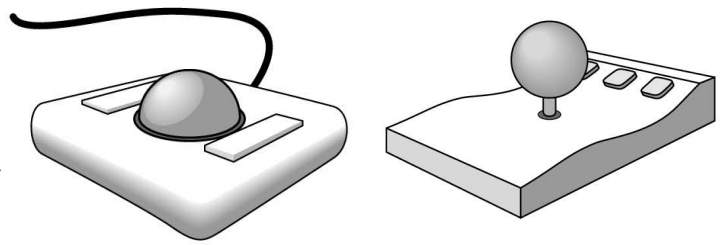
Another method is the optical sensor, which detects movement across the lines in the grid. One, two, or three buttons are usually included on the top of the mouse for signaling the execution of some operation, such as recording cursor position or invoking a function.



Trackball and Spaceball:

A Trackball is a ball that can be rotated with the fingers or palm of the hand, to produce screen-cursor movement. Trackballs are often mounted on keyboards and it is a two-dimensional positioning device.

A Spaceball provides six degrees of freedom. Unlike the trackball, a spaceball does not actually move. Spaceballs are used for three-dimensional positioning and selection operations in virtual-reality systems, modeling, animation, CAD, and other applications.



Trackball

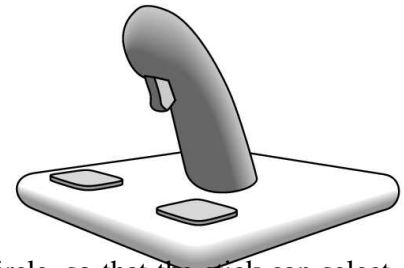
Spaceball

Joysticks:

A Joystick consists of a small, vertical lever (called the stick) mounted on a base that is used to steer the screen cursor around. Most joysticks select screen positions with actual stick movement; others respond to pressure on the stick. Some are mounted on a keyboard; others function as stand-alone units.

In another type of movable joystick, 8 switches are arranged in a circle, so that the stick can select any one of eight directions for cursor movement.

Pressure-sensitive joysticks, also called *Isometric Joysticks*, have a nonmovable stick. Pressure on the stick is measured with strain gauges and converted to movement of the cursor in the direction specified.



Data Glove:

A Data Glove that can be used to grasp a "virtual" object. The glove is constructed with a series of sensors that detect hand and finger motions. Electromagnetic coupling between transmitting antennas and receiving antennas is used to provide information about the position and orientation of the hand. A two-dimensional projection of the scene can be viewed on a video monitor, or a three-dimensional projection can be viewed with a headset.



Digitizers:

A common device for drawing, painting, or interactively selecting coordinate positions on an object is a *Digitizer*. These devices can be used to input coordinate values in either a two-dimensional or a three-dimensional space.

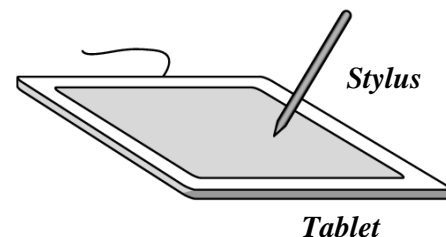
One type of digitizer is the graphics tablet (data tablet), which is used to input two-dimensional coordinates by activating a hand cursor or stylus at selected positions on a flat surface. A hand cursor contains cross hairs for sighting positions, while a stylus is a pencil-shaped device that is pointed at positions on the tablet.

Graphics tablets provide a highly accurate method for selecting coordinate positions, with an accuracy that varies from about 0.2 mm on desktop models to about 0.05 mm or less on larger models.

Many graphics tablets are constructed with a rectangular grid of wires embedded in the tablet surface.

Depending on the technology, either signal strength, coded pulses, or phase shifts can be used to determine the position on the tablet.

Three-dimensional digitizers use sonic or electromagnetic transmissions to record positions. Resolution of this system is from 0.8 mm to 0.08 mm, depending on the model.



Stylus

Tablet

Image Scanners:

Drawings, graphs, color and black-and-white photos, or text can be stored for computer processing with an **Image Scanner** by passing an optical scanning mechanism over the information to be stored. We can also apply various transformations, image-processing methods to modify the array representation of the picture.

Touch Panels:

Touch Panels allow displayed objects or screen positions to be selected with the touch of a finger. A typical application of touch panels is for the selection of processing options that are represented with graphical icons.

Touch input can be recorded using optical, electrical, or acoustical methods.

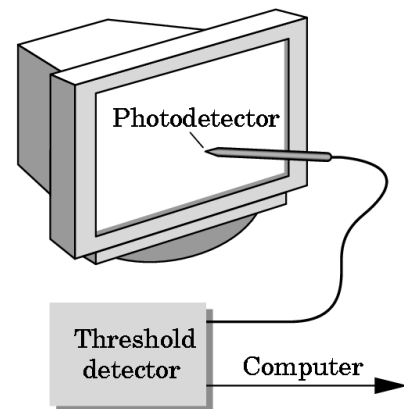
- **Optical Touch Panels** employ a line of infrared light-emitting diodes (LEDs) along one vertical edge and along one horizontal edge of the frame. The opposite vertical and horizontal edges contain light detectors. These detectors are used to record which beams are interrupted when the panel is touched.
- An **Electrical Touch Panel** is constructed with two transparent plates separated by a small distance. One of the plates is coated with a conducting material, and the other plate is coated with a resistive material. When the outer plate is touched, it is forced into contact with the inner plate. This contact creates a voltage drop across the resistive plate that is converted to the coordinate values of the selected screen position.
- In **Acoustical Touch Panels**, high-frequency sound waves are generated in the horizontal and vertical directions across a glass plate.

Light Pens

Light Pen is a pencil-shaped device used to select screen positions by detecting the light coming from points on the CRT screen.

Light Pens sometimes give false readings due to background lighting in a room.

The pen will send a pulse whenever phosphor below it is illuminated. While the image on a refresh display may appear to be stable, it is in fact blinking on and off faster than the eye can detect. This blinking is not too fast, for the light pen. The light pen as easily determines the time at which the phosphor is illuminated. Since there is only one electron beam on the refresh display, only one line segment can be drawn at a time and no two segments are drawn simultaneously.



Voice Systems

Speech recognizers are used in some graphics workstations as input devices to accept voice commands. The **Voice-System** input can be used to initiate graphics operations or to enter data. These systems operate by matching an input against a predefined dictionary of words and phrase.

HARD-COPY DEVICES:

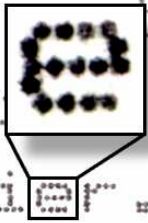
We can obtain hard-copy output for various images in several formats. The most important output device is Printer or Plotter and many types are there.

⇒ Printers produce output by either **Impact** or **Nonimpact** methods.

⇒ **Impact Printers** press formed character faces against an inked ribbon onto the paper. E.g. : Line Printer, with the typefaces mounted on bands, chains, drums, or wheels.

⇒ Character impact printers often have a **Dot-Matrix** print head containing a rectangular array of protruding wire pins, with the number of pins depending on the quality of the printer.

ystem where a
ld allow us t
mercial supplier.



⇒ **Nonimpact Printers** and plotters use laser techniques, ink-jet sprays, xerographic processes (photocopying), electrostatic methods, and electrothermal methods to get images onto Paper.

⇒ In a **Laser Device**, a laser beam mates a charge distribution on a rotating drum coated with a photoelectric material, such as selenium. Toner is applied to the drum and then transferred to paper.

⇒ **Ink-Jet** methods produce output by squirting ink in horizontal rows across a roll of paper wrapped on a drum. The electrically charged ink stream is deflected by an electric field to produce dot-matrix patterns.



Laser Printer



Inkjet Printer

⇒ An **Electrostatic Device** places a negative charge on the paper, one complete row at a time along the length of the paper. Then the paper is exposed to a toner. The toner is positively charged and so is attracted to the negatively charged areas, where it adheres to produce the specified output.

⇒ **Electrothermal** methods use heat in a dot-matrix print head to output patterns on heat sensitive paper. We can get limited color output on an impact printer by using different colored ribbons.

⇒ **Nonimpact** devices use various techniques to combine three color pigments (cyan, magenta, and yellow) to produce a range of color patterns.

⇒ Laser and Xerographic devices deposit the three pigments on separate passes; ink-jet methods shoot the three colors simultaneously on a single pass along each print line on the paper.

Drafting layouts and other drawings are typically generated with ink-jet or pen plotters.

⇒ A **Pen Plotter** has one or more pens mounted on a carriage, or crossbar, that spans a sheet of paper. Pens with varying colors and widths, wet-ink, ball-point, and felt-tip pens are all possible choices for use with a pen plotter. Crossbars can be either moveable or stationary, while the pen moves back and forth along the bar. E.g.: flatbed pen plotter, rollfeed pen plotter.



❧ *End of 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.

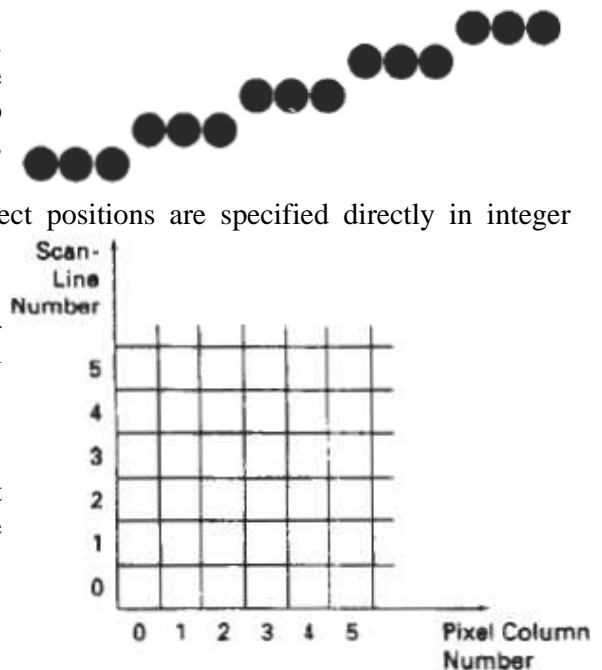
For the raster-graphics device-level algorithms, object positions are specified directly in integer device coordinates.

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 (**x₁**, **y₁**) and (**x₂**, **y₂**).

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

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

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

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 along a line, we can compute the corresponding y interval Δy 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 $|m| < 1$, Δx can be set proportional to a small horizontal deflection voltage and the corresponding vertical deflection is then set proportional to Δy as calculated from Equation 4.
- ⊕ For lines whose slopes have magnitudes $|m| > 1$, Δy can be set proportional to a small vertical deflection voltage with the corresponding horizontal deflection voltage set proportional to Δx , calculated from 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 ($\Delta x=1$) 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 ($\Delta y=1$) 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 $\Delta x = -1$ and

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

or (when the slope is greater than 1) we have $\Delta y = -1$ 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=0; k<steps; k++)
    {
        x += xIncrement;
        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.

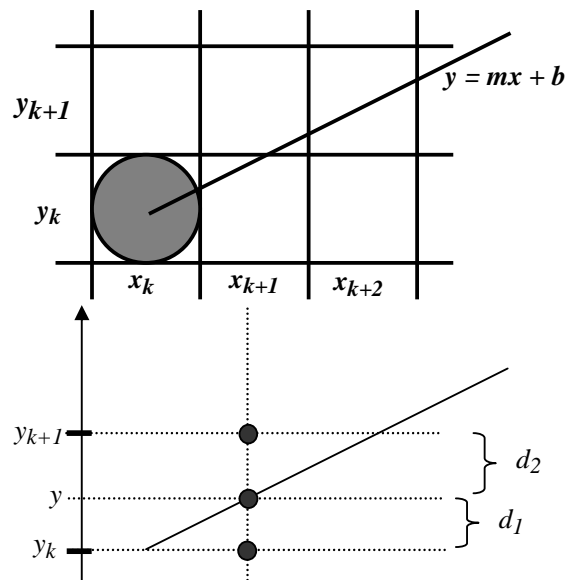
Bresenham's Line Drawing Algorithm.

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

$$\begin{aligned} d_1 &= y - y_k \\ &= m (x_k + 1) + b - y_k \end{aligned}$$

and

$$\begin{aligned} d_2 &= (y_{k+1}) - y \\ &= y_k + 1 - m (x_k + 1) - b \end{aligned}$$

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 k th step in the line algorithm can be obtained by rearranging equation (2) so that it involves only integer calculations. We accomplish this by substituting $m = \Delta y / \Delta x$, where Δy and Δx are the vertical and horizontal separations of the endpoint positions, and defining:

$$\begin{aligned} p_k &= \Delta x(d_1 - d_2) \\ &= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c \end{aligned} \longrightarrow (3)$$

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

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

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 Δx , Δy , $2\Delta y$, $2\Delta y - 2\Delta x$, 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) Δx times.

Parallel Line Algorithms.

With a parallel computer, we can calculate pixel positions along a line path simultaneously by partitioning the computations among the various processors available.

One approach to the partitioning problem is to adapt an existing sequential algorithm to take advantage of multiple processors. Alternatively, we can look for other ways to set up the processing so that pixel positions can be calculated efficiently in parallel.

An important consideration in devising a parallel algorithm is to balance the processing load among the available processors.

Given n_p processors, we can set up a parallel Bresenham line algorithm by subdividing the line path into n_p partitions and simultaneously generating line segments in each of the subintervals.

For a line with slope $0 < m < 1$ and left endpoint coordinate position (x_o, y_o) , we partition the line along the positive x direction. The distance between beginning x positions of adjacent partitions can be calculated as,

$$\Delta x_p = \frac{\Delta x + n_p - 1}{n_p} \longrightarrow (1)$$

where Δx is the width of the line, and the value for partition width Δx_p is computed using integer division. Numbering the partitions, and the as $0, 1, 2$, up to $n-1$, we calculate the starting x coordinate for the k th partition as,

$$x_k = x_o + k\Delta x_p \longrightarrow (2)$$

To apply Bresenham's algorithm over the partitions, we need the initial value for the y coordinate and the initial value for the decision parameter in each partition. The change Δy_p in the y direction over each partition is calculated from the line slope m and partition width Δx_p

$$\Delta y_p = m\Delta x_p \longrightarrow (3)$$

At the k^{th} partition, the starting y coordinate is then

$$y_k = y_o + \text{round}(k\Delta y_p) \longrightarrow (4)$$

The initial decision parameter for Bresenham's algorithm at the start of the k^{th} subinterval is obtained from the equation,

$$\begin{aligned} p_k &= \Delta x(d_1 - d_2) \\ &= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c \end{aligned}$$

we get,

$$p_k = (k\Delta x_p)(2\Delta y) - \text{round}(k\Delta y_p)(2\Delta x) + 2\Delta y - \Delta x$$

Each processor then calculates pixel positions over its assigned subinterval using the starting decision parameter value for that subinterval and the starting coordinates (x_k, y_k) .

The extension of the parallel Bresenham algorithm to a line with slope > 1 is achieved by partitioning the line in the y direction and calculating beginning x values for the partitions. For negative slopes, we increment coordinate values in one direction and decrement in the other.

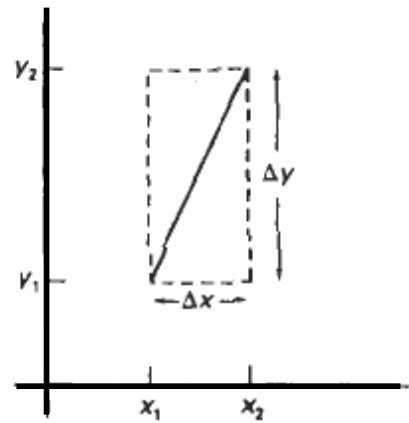
Another way to set up parallel algorithms on raster systems is to assign each processor to a particular group of screen pixels.

Perpendicular distance d from the line in the diagram to a pixel with coordinates (x, y) is obtained with the calculation

$$d = Ax + By + C$$

where,

$$\begin{aligned} A &= \frac{-\Delta y}{\text{linelength}} \\ B &= \frac{\Delta x}{\text{linelength}} \\ C &= \frac{x_o\Delta y - y_o\Delta x}{\text{linelength}} \end{aligned}$$



$$\text{linelength} = \sqrt{\Delta x^2 + \Delta y^2}$$

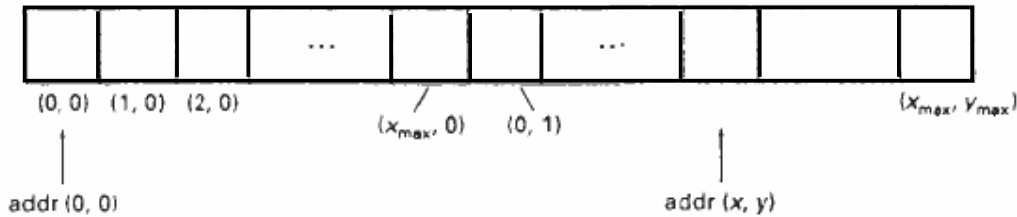
LOADING THE FRAME BUFFER

Scan-conversion algorithms generate pixel positions at successive unit intervals. This allows us to use incremental methods to calculate frame-buffer addresses.

Suppose the frame-buffer array is addressed in row-major order and that pixel positions vary from (0, 0) at the lower left screen corner to (x_{\max} , y_{\max}) at the top right corner).

For a bilevel system (1 bit per pixel), the frame-buffer bit address for pixel position (x , y) is calculated as,

$$\text{addr}(x, y) = \text{addr}(0, 0) + y(x_{\max} + 1) + x$$



Frame Buffer

Moving across a scan line, we can calculate the frame-buffer address for the pixel at ($x + 1$, y) as the following offset from the address for position (x , y):

$$\text{addr}(x + 1, y) = \text{addr}(x, y) + 1$$

Stepping diagonally up to the next scan line from (x , y), we get to the frame-buffer address of ($x + 1$, $y + 1$) with the calculation

$$\text{addr}(x + 1, y + 1) = \text{addr}(x, y) + x_{\max} + 2$$

where the constant $x_{\max} + 2$ is precomputed once for all line segments. Similar incremental calculations can be obtained for unit steps in the negative x and y screen directions. Each of these address calculations involves only a single integer addition.

With systems that can display a range of intensity values for each pixel, frame-buffer address calculations would include pixel width (number of bits), as well as the pixel screen location.

CIRCLE – GENERATING ALGORITHMS:

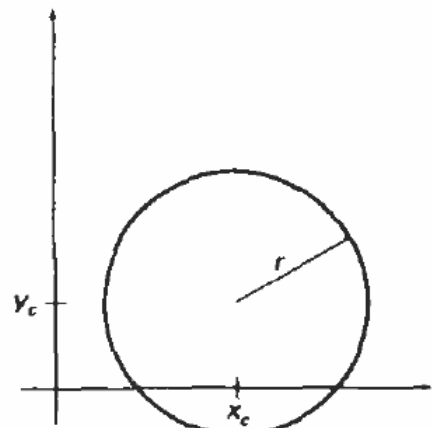
In general, a single procedure can be provided to display either *Circular* or *Elliptical Curves*.

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,

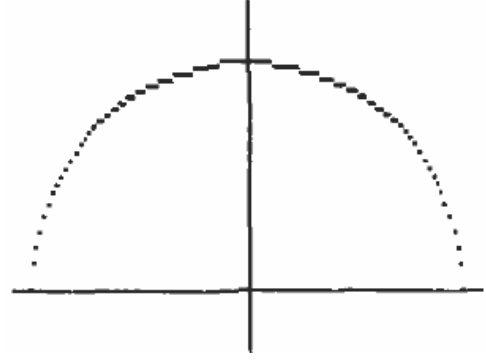
$$(x - x_c)^2 + (y - y_c)^2 = r^2 \longrightarrow (1)$$



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

$$y = y_c \pm \sqrt{r^2 - (x_c - x)^2} \longrightarrow (2)$$

But this is not the best method for generating a circle. One problem with this approach is that it involves considerable computation at each step. Moreover, the spacing between plotted pixel positions is not uniform, as shown in the following figure.



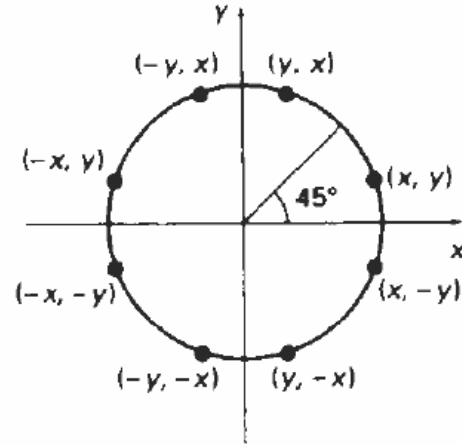
Another way to eliminate the unequal spacing is to calculate points along the circular boundary using polar coordinates r and θ . Expressing the circle equation in parametric polar form yields the pair of equations,

$$\begin{aligned} x &= x_c + r \cos \theta \\ y &= y_c + r \sin \theta \end{aligned} \longrightarrow (3)$$

When a display is generated with these equations using a fixed angular step size, a circle is plotted with equally spaced points along the circumference.

Computation can be reduced by considering the 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 in the 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 $x=y$.

Determining pixel positions along a circle circumference using either equation (1) or equation (3) still requires a good deal of computation time.

Bresenham's line algorithm for raster displays is adapted to circle generation by setting up decision parameters for finding the closest pixel to the circumference at each sampling step.

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:

$$f_{\text{circle}}(x, y) = x^2 + y^2 - r^2 \longrightarrow (4)$$

Any point (x, y) on the boundary of the circle with radius r satisfies the equation $f_{\text{circle}}(x, y) = 0$.

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:

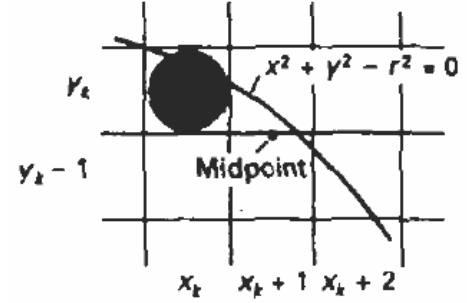
$$f_{\text{circle}}(x, y) \begin{cases} < 0, & \text{if } (x, y) \text{ is inside the circle boundary} \\ = 0, & \text{if } (x, y) \text{ is on the circle boundary} \\ > 0, & \text{if } (x, y) \text{ is outside the circle boundary} \end{cases} \longrightarrow (5)$$

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:

$$\begin{aligned} p_k &= f_{\text{circle}}\left(x_k + 1, y_k - \frac{1}{2}\right) \\ &= (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2 \end{aligned} \longrightarrow (6)$$



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

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

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 circle-drawing 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.

If 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,

$$d_1 + d_2 = \text{constant} \longrightarrow (1)$$

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

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

$$\text{We can rewrite the general ellipse equation in the form,} \\ Ax^2 + By^2 + Cxy + Dx + Ey + F = 0 \longrightarrow (3)$$

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,

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

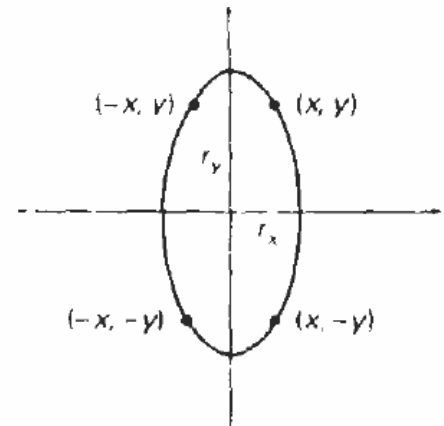
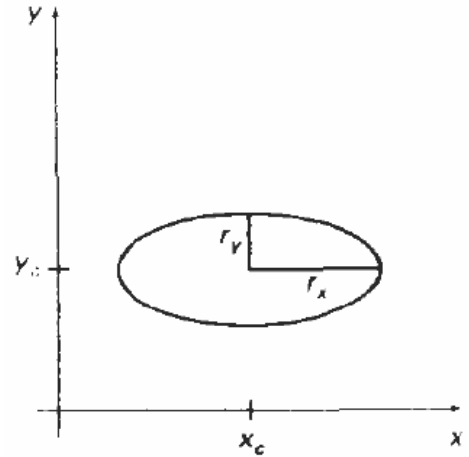
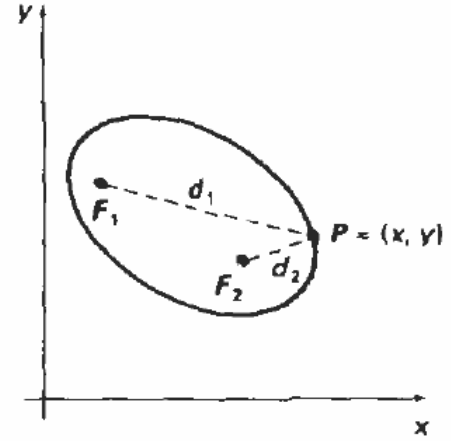
Using polar coordinates r and θ , we can also describe the ellipse in standard position with the parametric equations:

$$\begin{aligned} x &= x_c + r_x \cos \theta \\ y &= y_c + r_y \sin \theta \end{aligned} \longrightarrow (5)$$

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 r_x, r_y , and (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 (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 $(0, r_y)$ 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 .

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

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

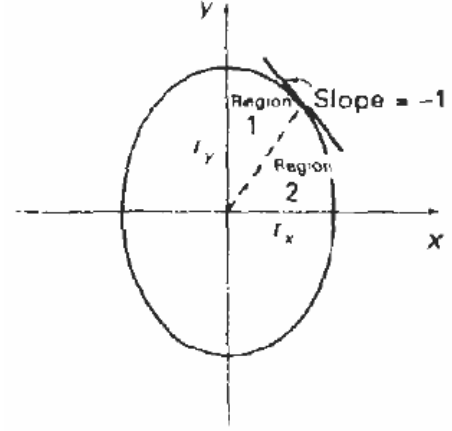
We define an Ellipse Function from equation (4) with $(x_c, y_c) = (0, 0)$ as

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

which has the following properties:

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

Thus the ellipse function serves as the decision parameter in the Midpoint Algorithm.



1. Input \mathbf{r}_x , \mathbf{r}_y and ellipse center $(\mathbf{x}_c, \mathbf{y}_c)$, and obtain the first point on an ellipse centered on the origin as,

$$(\mathbf{x}_0, \mathbf{y}_0) = (0, \mathbf{r}_y)$$

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 \mathbf{x} , position in region 1, starting at $\mathbf{k} = 3$, perform the following test:

If $\mathbf{p1} < 0$, the next point along the ellipse centered on $(0, 0)$ is $(\mathbf{x}_{k+1}, \mathbf{y}_k)$ and

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

Otherwise, the next point along the circle is $(\mathbf{x}_{k+1}, \mathbf{y}_{k-1})$ 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 $(\mathbf{x}_0, \mathbf{y}_0)$ calculated in region 1 as

$$p2_0 = r_y^2 \left(x_0 + \frac{1}{2} \right)^2 + r_x^2 (y_0 - 1)^2 - r_x^2 r_y^2$$

5. At each \mathbf{y}_k position in region 2, starting at $\mathbf{k} = 0$, perform the following test:

If $\mathbf{p_k} > 0$, the next point along the ellipse centered on $(0, 0)$ is $(\mathbf{x}_k, \mathbf{y}_k - 1)$ and

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

Otherwise, the next point along the circle is $(\mathbf{x}_k + 1, \mathbf{y}_k - 1)$ 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 \mathbf{x} and \mathbf{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 $2r_y^2 x \geq 2r_x^2 y$.

FILLED-AREA PRIMITIVES:

A standard output primitive in general graphics packages is a solid-color or patterned polygon area. There are two basic approaches to area filling on raster systems.

- One way to fill an area is to determine the overlap intervals for scan lines that cross the area.
- Another method for area filling is to start from a given interior position and paint outward from this point until we encounter the specified boundary conditions.

The scan-line approach is typically used in general graphics packages to fill polygons, circles, ellipses, and other simple curves. All methods starting from an interior point are useful with more complex boundaries and in interactive painting systems.

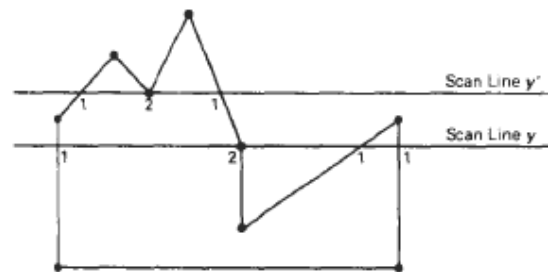
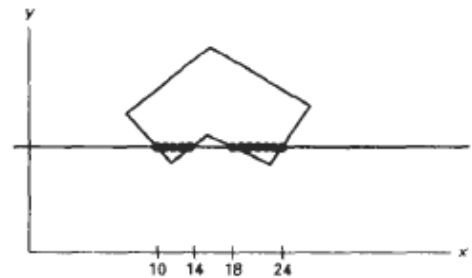
Scan-Line Polygon Fill Algorithm:

For each scan line crossing a polygon, the area-fill algorithm locates the intersection points of the scan line with the polygon edges. These intersection points are then sorted from left to right, and the corresponding frame-buffer positions between each intersection point are set to the specified fill color.

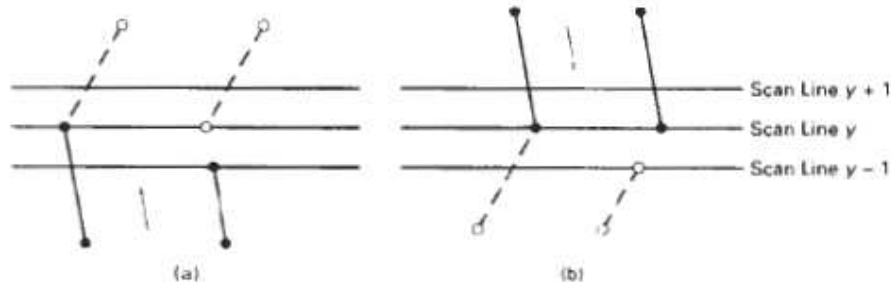
In the diagram, the four pixel intersection positions with the polygon boundaries define two stretches of interior pixels from $x = 10$ to $x = 14$ and from $x = 18$ to $x = 24$.

Some scan-line intersections at polygon vertices require special handling. A scan line passing through a vertex intersects two polygon edges at that position, adding two points to the list of intersections for the scan line.

The following figure shows two scan lines at positions y and y' that intersect edge endpoints. Scan line y intersects five polygon edges. Scan line y' , however, intersects an even number of edges although it also passes through a vertex. Intersection points along scan line y' correctly identify the interior pixel spans. But with scan line y , we need to do some additional processing to determine the correct interior points. The topological difference between scan line y and scan line y' in the diagram is identified by noting the position of the intersecting edges relative to the scan line.



The following figure illustrates shortening of an edge. When the endpoint y coordinates of the two edges are increasing, the y value of the upper endpoint for the current edge is decreased by 1, as in fig. (a). When the endpoint y values are monotonically decreasing, as in fig. (b), we decrease the y coordinate of the upper endpoint of the edge following the current edge.



Calculations performed in scan-conversion and other graphics algorithms typically take advantage of various **coherence** properties of a scene that is to be displayed. Coherence methods often involve

incremental calculations applied along a single scan line or between successive scan lines. In determining edge intersections, we can set up incremental coordinate calculations along any edge by exploiting the fact that the slope of the edge is constant from one scan line to the next.

The following figure shows two successive scan lines crossing a left edge of a polygon. The slope of this polygon boundary line can be expressed in terms of the scan-line intersection coordinates:

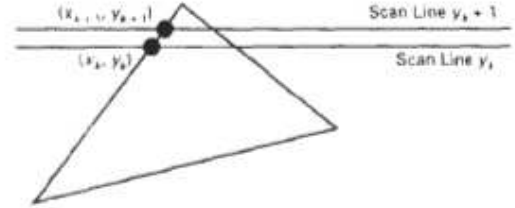
$$m = \frac{y_{k+1} - y_k}{x_{k+1} - x_k}$$

Since the change in y coordinates between the two scan lines is simply

$$y_{k+1} - y_k = 1$$

the x-intersection value x_{k+1} on the upper scan line can be determined from the x-intersection value x_k on the preceding scan line as

$$x_{k+1} = x_k + \frac{1}{m}$$



Each successive x intercept can thus be calculated by adding the inverse of the slope and rounding to the nearest integer.

Along an edge with slope m , the intersection x_k value for scan line k above the initial scan line can be calculated as,

$$x_k = x_0 + \frac{k}{m}$$

In a sequential fill algorithm, the increment of x values by the amount $1/m$ along an edge can be accomplished with integer operations by recalling that the slope m is the ratio of two integers:

$$m = \frac{\Delta y}{\Delta x},$$

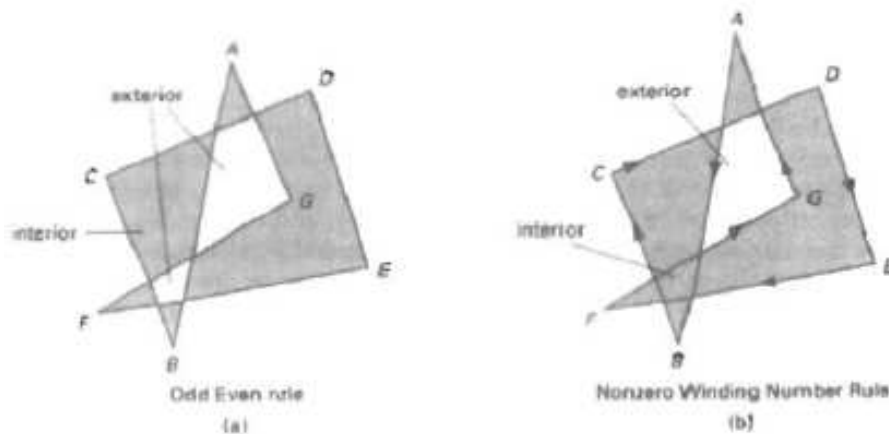
where Δx and Δy are the differences between the edge endpoint x and y coordinate values. Thus, incremental calculations of x intercepts along an edge for successive scan lines can be expressed as

$$x_{k+1} = x_k + \frac{\Delta x}{\Delta y}$$

Using this equation, we can perform integer evaluation of the x intercepts by initializing a counter to 0, then incrementing the counter by the value of Δx each time we move up to a new scan line. Whenever the counter value becomes equal to or greater than Δy , we increment the current x intersection value by 1 and decrease the counter by the value Δy . This procedure is equivalent to maintaining integer and fractional parts for x intercepts and incrementing the fractional part until we reach the next integer value.

Inside-Outside Tests:

Area-filling algorithms and other graphics processes often need to identify interior regions of objects. Examples of standard polygons include triangles, rectangles, octagons, and decagons. The component edges of these objects are joined only at the vertices, and otherwise the edges have no common points in the plane.



For such shapes, it is not always clear which regions of the xy plane we should call "interior" and which regions we should designate as "exterior" to the object. Graphics packages normally use either the odd-even rule or the nonzero winding number rule to identify interior regions of an object.

We apply the **odd-even rule**, also called the **odd parity rule** or **the even-odd rule**, by conceptually drawing a line from any position P to a distant point outside the coordinate extents of the object and counting the number of edge crossings along the line. If the number of polygon edges crossed by this line is odd, then P is an **interior** point. Otherwise, P is an **exterior** point.

To obtain an accurate edge count, we must be sure that the line path we choose does not intersect any polygon vertices. The fig. (a) shows the interior and exterior regions obtained from the odd-even rule for a self-intersecting set of edges.

Another method for defining interior regions is the **nonzero winding number rule**, which counts the number of times the polygon edges wind around a particular point in the counterclockwise direction. This count is called the **winding number**, and the interior points of a two-dimensional object are defined to be those that have a nonzero value for the winding number. We apply the nonzero winding number rule to polygons by initializing the winding number to 0 and again imagining a line drawn from any position P to a distant point beyond the coordinate extents of the object.

Some graphics packages use the nonzero winding number rule to implement area filling, since it is more versatile than the odd-even rule.

Scan-Line Fill of Curved Boundary Areas:

In general, scan-line fill of regions with curved boundaries requires more work than polygon filling, since intersection calculations now involve nonlinear boundaries. For simple curves such as circles or ellipses, performing a scan-line fill is a straightforward process. We only need to calculate the two scan-line Intersections on opposite sides of the curve. This is the same as generating pixel positions along the curve boundary, and we can do that with the midpoint method. Then we simply fill in the horizontal pixel spans between the boundary points on opposite sides of the curve.



Symmetries between quadrants (and between octants for circles) are used to reduce the boundary calculations. Similar methods can be used to generate a fill area for a curve section.

Boundary-Fill Algorithm:

Another approach to area filling is to start at a point inside a region and paint the interior outward toward the boundary. If the boundary is specified in a single color, the fill algorithm proceeds outward pixel by pixel until the boundary color is encountered. This method, called the boundary-fill algorithm, is particularly useful in interactive painting packages, where interior points are easily selected.

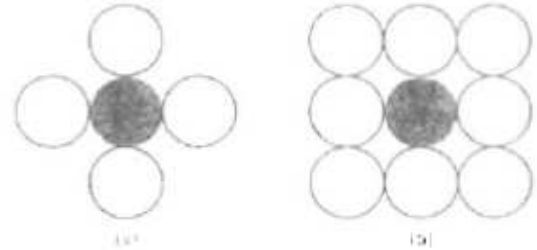
A boundary-fill procedure accepts as input the coordinates of an interior point (x, y) , a fill color, and a boundary color. Starting from (x, y) , the procedure tests neighboring positions to determine whether they are of the boundary color. If not, they are painted with the fill color, and their neighbors are tested.



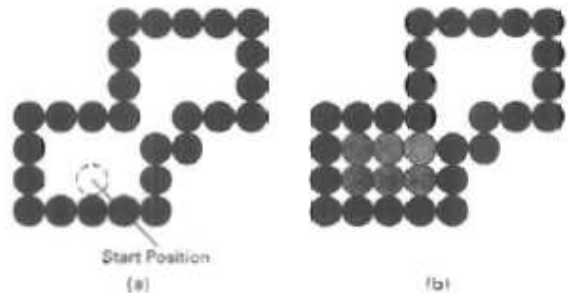
This process continues until all pixels up to the boundary color for the area have been tested. Both inner and outer boundaries can be set up to specify an area, and some examples of defining regions for boundary fill are shown in the figure.

The following figure shows two methods for proceeding to neighboring pixels from the current test position. In this figure, four neighboring points are tested. These are the pixel positions that are right, left, above, and below the current pixel. Areas filled by this method are called **4-connected**.

The second method is used to fill more complex figures. Here the set of neighboring positions to be tested includes the four diagonal pixels. Fill methods using this approach are called **8-connected**.



Recursive boundary-fill algorithms may not fill regions correctly if some interior pixels are already displayed in the fill color. This occurs because the algorithm checks next pixels both for boundary color and for fill color. Encountering a pixel with the fill color can cause a recursive branch to terminate, leaving other interior pixels unfilled. To avoid this, we can first change the color of any interior pixels that are initially set to the fill color before applying the boundary-fill procedure.



Flood-Fill Algorithm:

Sometimes we want to fill in (or recolor) an area that is not defined within a single color boundary. The figure shows an area bordered by several different color regions. We can paint such areas by replacing a specified interior color instead of searching for a boundary color value. This approach is called a **flood-fill algorithm**.

We start from a specified interior point (x, y) and reassign all pixel values that are currently set to a given interior color with the desired fill color. If the area we want to paint has more than one interior color, we can first reassign pixel values so that all interior points have the same color. Using either a 4-connected or 8-connected approach, we then step through pixel positions until all interior points have been repainted.



CHARACTER GENERATION:

Letters, numbers, and other characters can be displayed in a variety of sizes and styles. The overall design style for a set (or family) of characters is called a **type face**. Today, there are hundreds of typefaces available for computer applications. Examples of a few common typefaces are Courier, Helvetica, New York, Palatino, and Zapf Chancery. Originally, the term **font** referred to a set of cast metal character forms in a particular size and format, such as 10-point Courier Italic or 12-point Palatino Bold.

Typefaces (or fonts) can be divided into two broad groups: **serif** and **sans serif**. Serif type has small lines or accents at the ends of the main character strokes, while sans-serif type does not have accents.

Two different representations are used for storing computer fonts. A simple method for representing the character shapes in a particular typeface is to use rectangular grid patterns. The set of characters are then referred to as a **bitmap font (or bitmapped font)**. Another, more flexible, scheme is to describe character shapes using straight-line and curve sections, as in PostScript. In this case, the set of characters is called an **outline font**.

The two methods for character representation are illustrated below. When the pattern in fig.(a) is copied to an area of the frame buffer, the 1 bits designate which pixel positions are to be displayed on the monitor. To display the character shape in Fig. (b), the interior of the character outline must be filled using the scan-line fill procedure.

1	1	1	1	1	1	0	0
0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
0	1	1	1	1	1	0	0
0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0

(a)



(b)

Bitmap fonts are the simplest to define and display: The character grid only needs to be mapped to a frame-buffer position. In general, bitmap fonts require more space, because each variation (size and format) must be stored in a *font cache*.

Outline fonts require less storage since each variation does not require a distinct font cache. We can produce boldface, italic, or different sizes by manipulating the curve definitions for the character outlines. But it does take more time to process the outline fonts, because they must be scan converted into the frame buffer.

A character string is displayed with the following function:

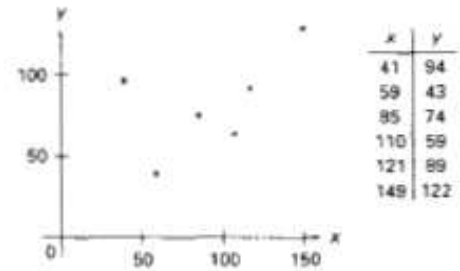
```
text (wcpoint, string)
```

Another convenient character function is one that places a designated character, called a *marker symbol*, at one or more selected positions. This function is defined with the same parameter list as in the line function:

```
polymarker (n, wcpoints)
```

A predefined character is then centered at each of the n coordinate positions in the list **wcpoints**. The default symbol displayed by **polymarker** depends on the particular implementation, but we assume for now that an asterisk is to be used. The diagram illustrates plotting of a data set with the statement

```
polymarker (6, wcpoints)
```



❧ End of Unit – II ❧

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 *lt* 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.

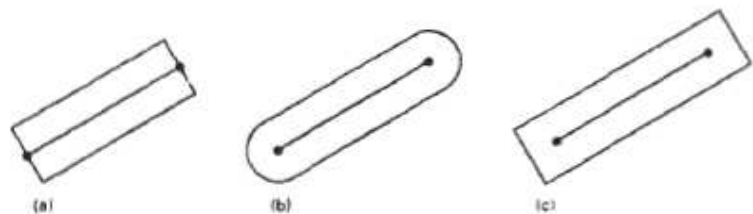
As with other PHIGS attributes, a line-width command is used to set the current line-width value in the attribute list.

We set the line-width attribute with the command:

setLinesidhScaleFactor (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*.

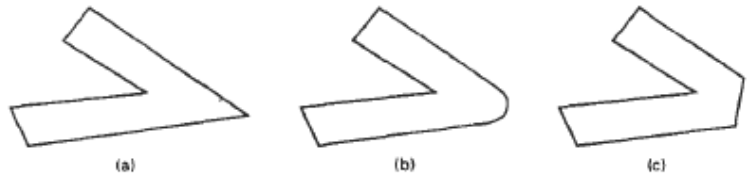
One kind of line cap is the *butt cap* obtained by adjusting the end positions of the component parallel lines so that the thick line is displayed with square ends that are perpendicular to the line path. If the specified line has slope *m*, the square end of the thick line has slope $-1/m$.

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

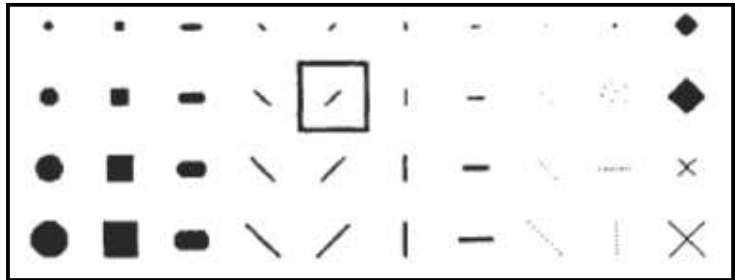
Pen and Brush Options:

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.

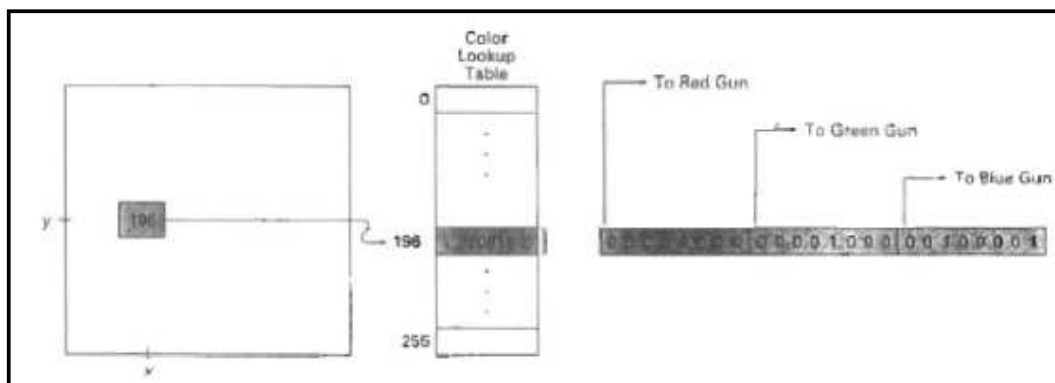
Each of the three bit positions is used to control the intensity level (either on or off) of the corresponding electron gun in an RGB monitor. The leftmost bit controls the red gun, the middle bit controls the green gun, and the rightmost bit controls the blue gun. Adding more bits per pixel to the frame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

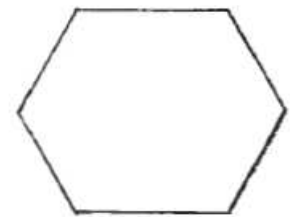
If additional bits per pixel are available in the frame buffer, the value of 0.33 would be mapped to the nearest level. With 3 bits per pixel, we can accommodate 8 gray levels; while 8 bits per pixel would give us 256 shades of gray.

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

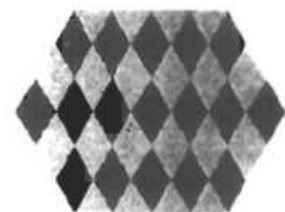
Intensity Codes	Stored Intensity Values In The Frame Buffer (Binary Code)	Displayed Grayscale
0.0	0	
0.33	1	
0.67	2	
1.0	3	



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 *pattern*.

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

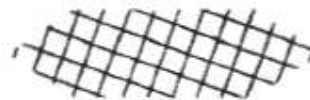
Fill selections for parameter fs are normally applied to polygon areas, but they can also be implemented to fill regions with curved boundaries.

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 (position)

Parameter *position* is a pointer to coordinates (xp, yp) that fix the lower left corner 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 $F \neq B$.

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 :

$$P = tF + (1 - t)B$$

where the “transparency” factor t has a value between 0 and 1 for each pixel.

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

$$P = (P_R, P_G, P_B), F = (F_R, F_G, F_B), B = (B_R, B_G, B_B)$$

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

$$t = \frac{P_k - B_k}{F_k - B_k}$$

where $k = R, G, \text{ or } B$ 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 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

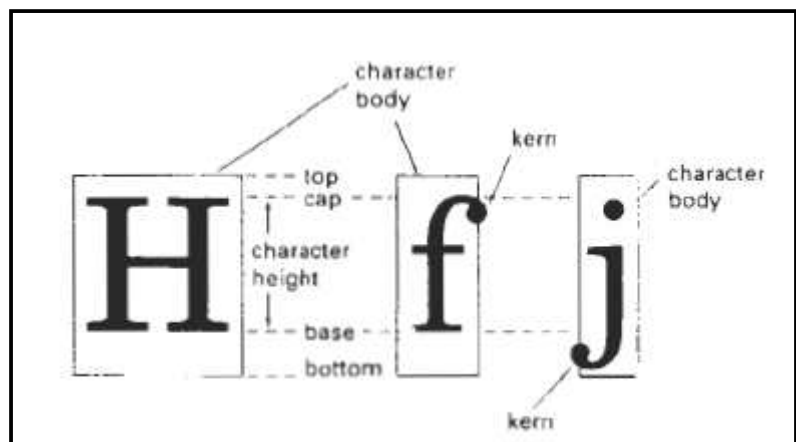
setTextColourIndex (tc)

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, l*, 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 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.

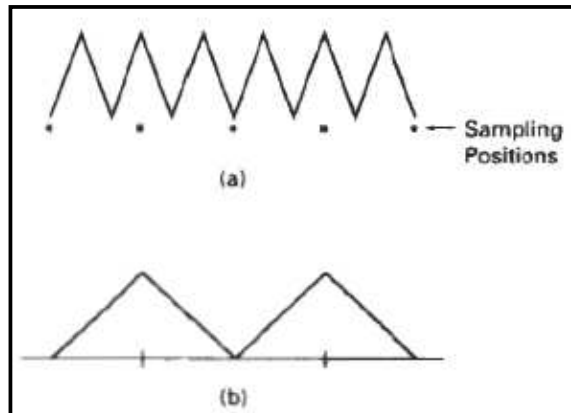
ANTIALIASING:

Displayed primitives generated by the raster algorithms have a jagged, or stairstep, appearance because the sampling process digitizes coordinate points 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 :

$$f_s = 2f_{\max}$$



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 Δx_s , is

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

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

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, at 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.

❧ End of Unit – III ❧

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') .

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

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

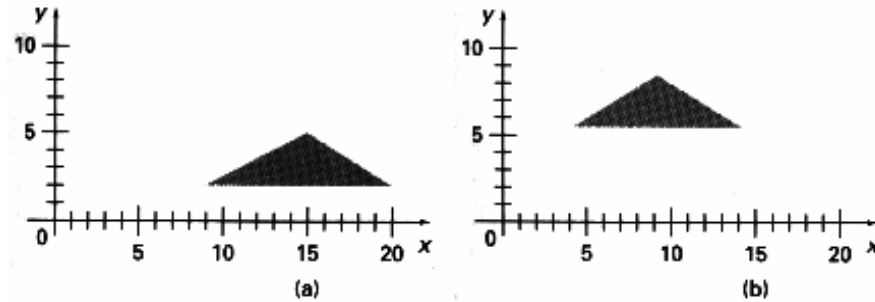
We can express the translation equations as a single matrix equation by using column vectors to represent coordinate positions and the translation vector:

$$P = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad P' = \begin{bmatrix} x'_1 \\ x'_2 \end{bmatrix}, \quad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

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.

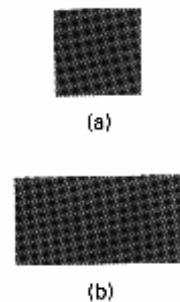
Similar methods are used to translate **curved** objects. To change the position of a circle or ellipse, we translate the center coordinates and redraw the figure in the new location. We translate other curves (splines) by displacing the coordinate positions defining the objects, and then we reconstruct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

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

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 (x_f, y_f) 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,

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

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.

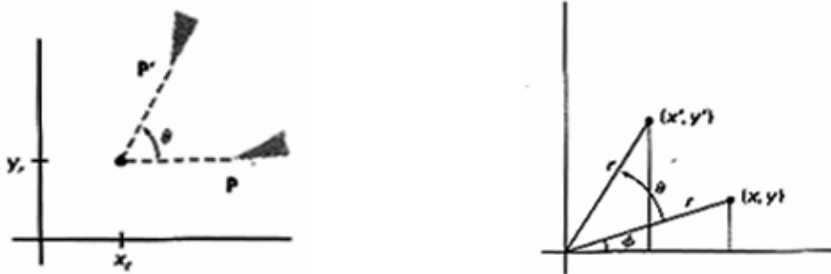
ROTATION

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 θ 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 ϕ is the original angular position of the point from the horizontal, and θ is the rotation angle.

Using standard trigonometric identities, we can express the transformed coordinates in terms of angles θ and ϕ 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

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

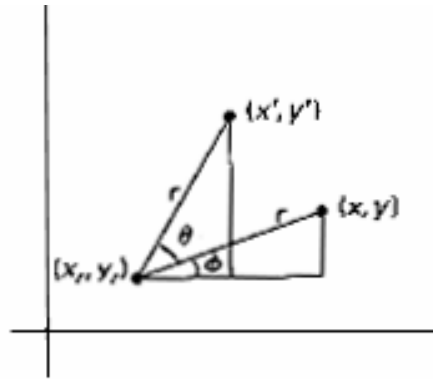
When coordinate positions are represented as row vectors instead of column vectors, the matrix product in rotation equation is transposed so that the transformed row coordinate vector $[x' y']$ calculated as,

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

$$= P^T \cdot R^T$$

where $P'^T = [x' y']$, and the transpose 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 fit the picture components into their proper positions.

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

$$\mathbf{P}' = \mathbf{M}_1 \cdot \mathbf{P} + \mathbf{M}_2$$

with coordinate positions \mathbf{P} and \mathbf{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

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

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

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

The rotation transformation operator $\mathbf{R}(\theta)$ is the 3 by 3 matrix with rotation parameter θ . We get the inverse rotation matrix when θ is replaced with $-\theta$.

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

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

or

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

where $\mathbf{S}(s_x, s_y)$ 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 \mathbf{P} , the final transformed location \mathbf{P}' is calculated as

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

where \mathbf{P} and \mathbf{P}' are represented as homogeneous-coordinate column vectors.

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

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

or

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

which demonstrates that two successive translations are additive.

Rotations:

Two successive rotations applied to point \mathbf{P} produce the transformed position

$$\begin{aligned} \mathbf{P}' &= \mathbf{R}(\theta_2) \cdot \{\mathbf{R}(\theta_1) \cdot \mathbf{P}\} \\ &= \{\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1)\} \cdot \mathbf{P} \end{aligned}$$

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

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

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.

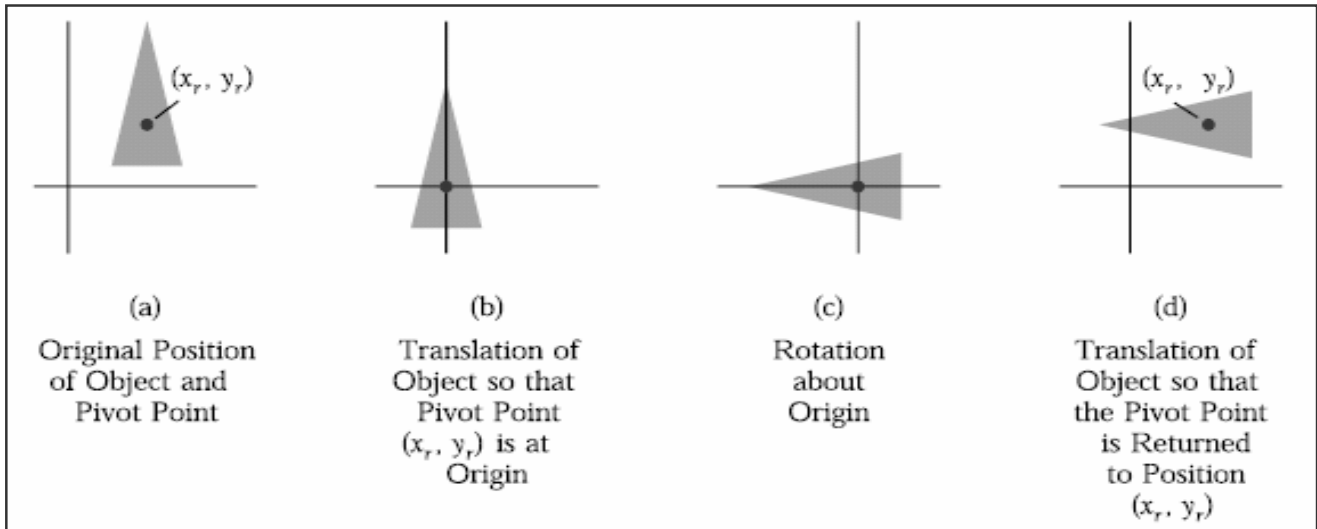
This transformation sequence is illustrated in the following diagram. The composite transformation matrix for this sequence is obtained with the concatenation.

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

which can be expressed in the form

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

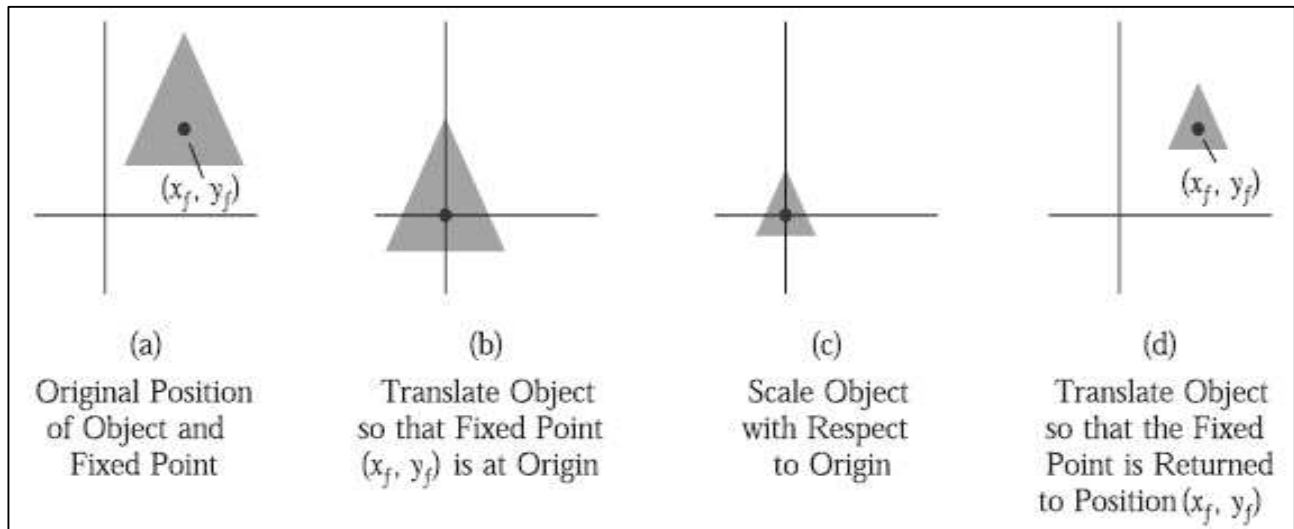
where $\mathbf{T}(-x_r, -y_r) = \mathbf{T}^{-1}(x_r, y_r)$



General Fixed-Point Scaling:

The following diagram illustrates a transformation sequence to produce scaling with respect to a selected fixed position (x_f, y_f) 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

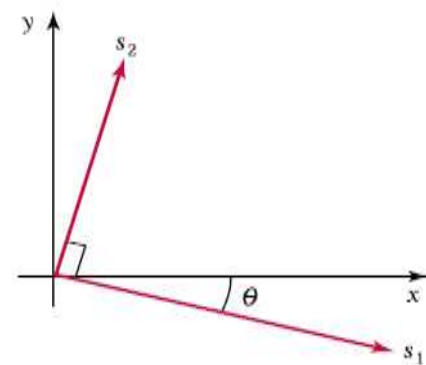
$$\begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_f(1-s_x) \\ 0 & s_y & y_f(1-s_y) \\ 0 & 0 & 1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

or

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

General Scaling Directions:

Parameters s_x and s_y scale objects along the x and y directions. We can scale an object in other directions by rotating



the object to align the desired scaling directions with the coordinate axes before applying the scaling transformation.

If we want to apply scaling factors with values specified by parameters s_1 and s_2 in the directions shown in the diagram, to accomplish the scaling with out changing the orientation of the object, we first perform a rotation so that the directions for s_1 and s_2 coincide with the x and y axes, respectively.

Then the scaling transformation is applied, followed by an opposite rotation to return points to their original orientations. The composite matrix resulting from the product of these three transformations is,

$$\mathbf{R}^{-1}(\theta) \cdot \mathbf{S}(s_1, s_2) \cdot \mathbf{R}(\theta) = \begin{bmatrix} s_1 \cos^2 \theta + s_2 \sin^2 \theta & (s_2 - s_1) \cos \theta \sin \theta & 0 \\ (s_2 - s_1) \cos \theta \sin \theta & s_1 \sin^2 \theta + s_2 \cos^2 \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Concatenation Properties:

Matrix multiplication is associative. For any three matrices, \mathbf{A} , \mathbf{B} and \mathbf{C} , the matrix product $\mathbf{A} \cdot \mathbf{B} \cdot \mathbf{C}$ can be performed by first multiplying \mathbf{A} and \mathbf{B} or by first multiplying \mathbf{B} and \mathbf{C} :

$$\mathbf{A} \cdot \mathbf{B} \cdot \mathbf{C} = (\mathbf{A} \cdot \mathbf{B}) \cdot \mathbf{C} = \mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{C})$$

Therefore, we can evaluate matrix products using either a left-to-right or a right-to-left *associative* grouping.

On the other hand, transformation products may not be *commutative*: The matrix product $\mathbf{A} \cdot \mathbf{B}$ is not equal to $\mathbf{B} \cdot \mathbf{A}$.

This commutative property holds also for two successive translations or two successive scalings. Another commutative pair of operations is rotation and uniform scaling. ($s_x = s_y$)

General Composite Transformations and Computational Efficiency:

A general two-dimensional transformation, representing a combination of translations, rotations, and scalings, can be expressed as

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} rs_{xx} & rs_{xy} & trs_x \\ rs_{yx} & rs_{yy} & trs_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The four elements rs_{ij} are the multiplicative rotation-scaling terms in the transformation that involve only rotation angles and scaling factors.

Elements tr_{sx} and tr_{sy} are the translational terms containing combinations of translation distances, pivot-point and fixed-point coordinates, and rotation angles and scaling parameters.

For example, if an object is to be scaled and rotated about its centroid coordinates (x_c, y_c) and then translated, the values for the elements of the composite transformation matrix are

$$\mathbf{T}(t_x, t_y) \cdot \mathbf{R}(x_c, y_c, \theta) \cdot \mathbf{S}(x_c, y_c, s_x, s_y) = \begin{bmatrix} s_x \cos \theta & -s_y \sin \theta & x_c(1 - s_x \cos \theta) + y_c s_y \sin \theta + t_x \\ s_x \sin \theta & s_y \cos \theta & y_c(1 - s_y \cos \theta) - x_c s_x \sin \theta + t_y \\ 0 & 0 & 1 \end{bmatrix}$$

A general *rigid-body transformation matrix*, involving only translations and rotations, can be expressed in the form

$$\begin{bmatrix} r_{xx} & r_{xy} & tr_x \\ r_{yx} & r_{yy} & tr_y \\ 0 & 0 & 1 \end{bmatrix}$$

where the four elements r_{ij} are the multiplicative rotation terms, and elements t_x and t_y are the translational terms. A rigid-body change in coordinate position is also sometimes referred to as a **rigid-motion** transformation.

The **orthogonal property** of rotation matrices is useful for constructing a rotation matrix when we know the final orientation of an object rather than the amount of angular rotation necessary to put the object into that position.

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:

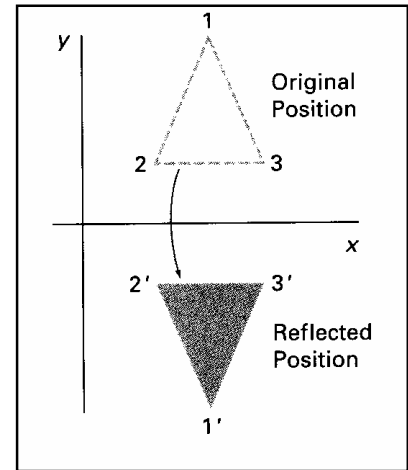
***x*-Reflection:**

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.



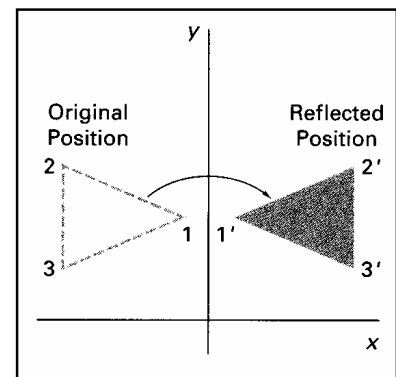
***y*-Reflection:**

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 $x = 0$.

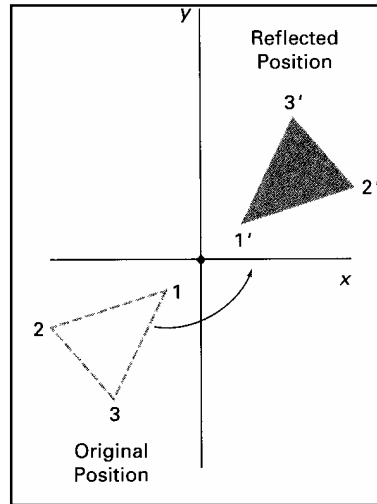


***Origin*-Reflection:**

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.

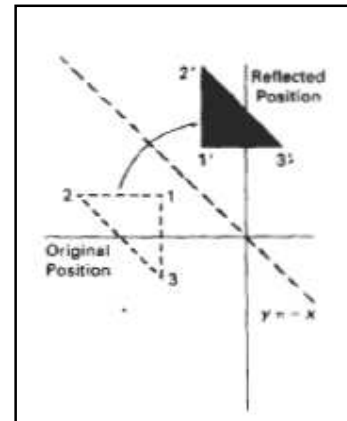
This transformation, referred to as a reflection relative to the coordinate origin, has the matrix representation:

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



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

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



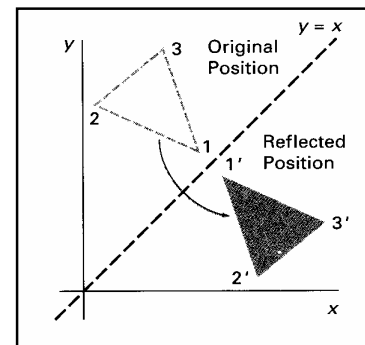
To obtain a transformation matrix for reflection about the diagonal $y = -x$, we could concatenate matrices for the transformation sequence:

- (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:**

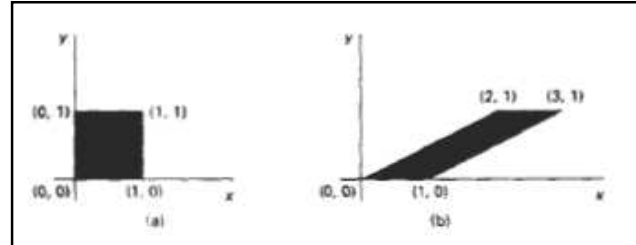
An x -direction shear relative to the x axis is produced with the transformation matrix

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

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.



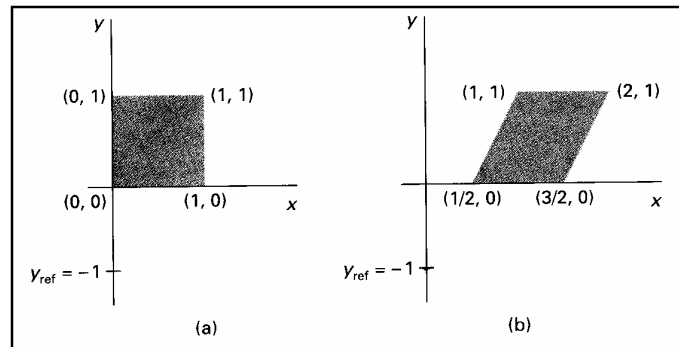
Negative values for sh_x shift coordinate positions to the left. We can generate x -direction shears relative to other reference lines with

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

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 $1/2$ 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

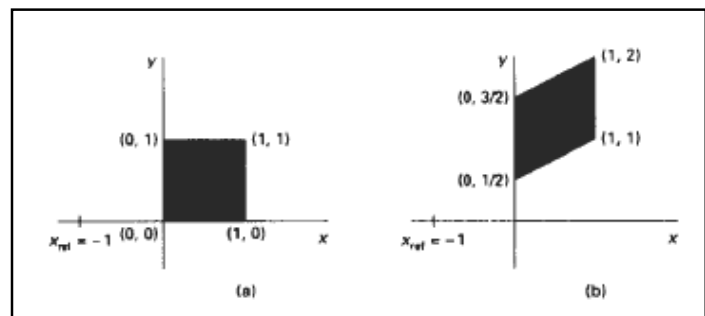
$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & -sh_y \cdot x_{ref} \\ 0 & 0 & 1 \end{bmatrix}$$

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 $sh_y = 1/2$ and $x_{ref} = -1$.



2D VIEWING AND CLIPPING

VIEWING:

THE VIEWING PIPELINE:

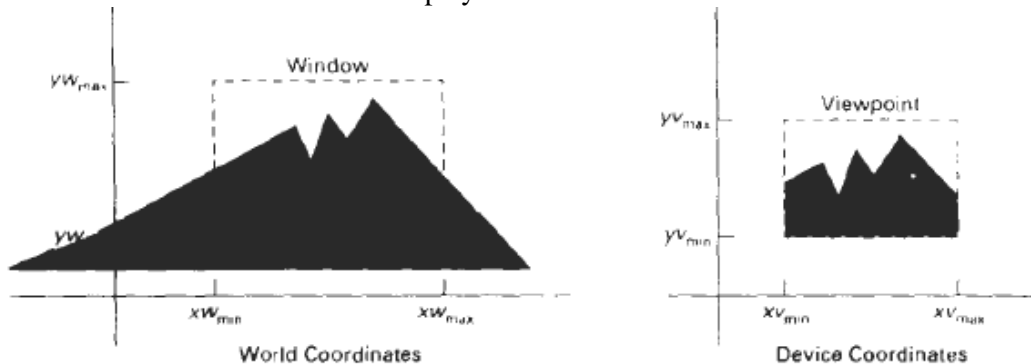
A world-coordinate area selected for display is called a **window**. An area on a display device to which a window is mapped is called a **viewport**.

- ☞ 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.

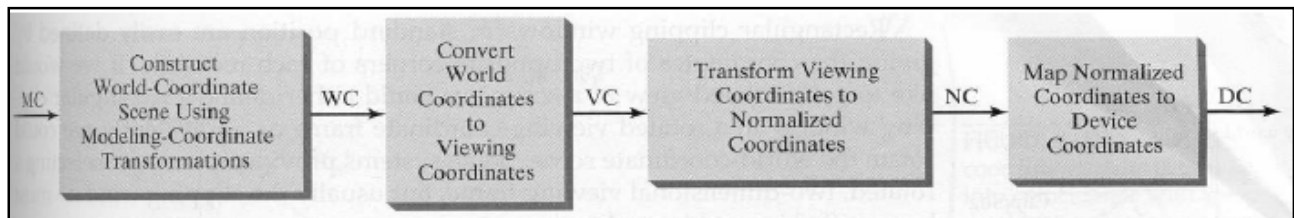
In general, the mapping of a part of a world-coordinate scene to device coordinates is referred to as a **viewing transformation**.

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

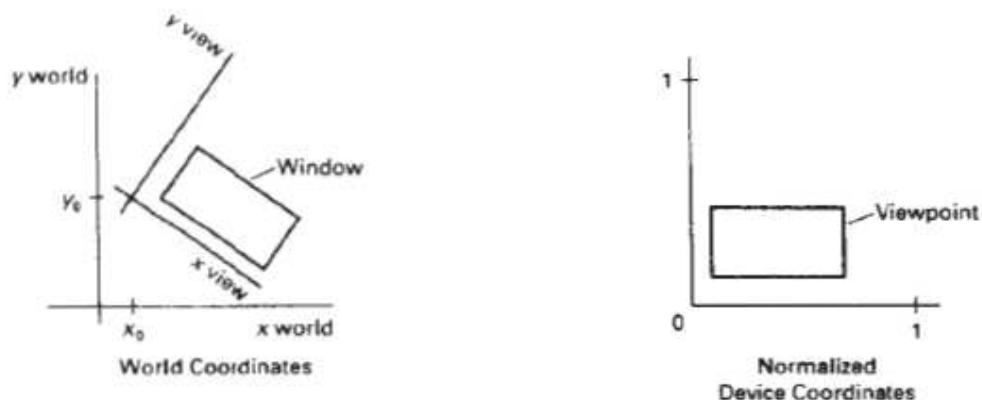


We carry out the viewing transformation in several steps, as indicated below.

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



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: $P_o = (x_o, y_o)$. 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_o , 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 P_o 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 (x_w, y_w) in a designated window is mapped to viewport coordinates (x_v, y_v) so that relative positions in the two areas are the same.

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 (x_w, y_w) in the window is mapped into position (x_v, y_v) in the associated viewport. To maintain the same relative placement in the viewport as in the window, we require that,

$$\frac{x_v - x_{v_{min}}}{x_{v_{max}} - x_{v_{min}}} = \frac{x_w - x_{w_{min}}}{x_{w_{max}} - x_{w_{min}}}$$

$$\frac{y_v - y_{v_{min}}}{y_{v_{max}} - y_{v_{min}}} = \frac{y_w - y_{w_{min}}}{y_{w_{max}} - y_{w_{min}}}$$

Solving these expressions for the viewport position (x_v, y_v) , we have

$$x_v = x_{v_{min}} + (x_w - x_{w_{min}})sx$$

$$y_v = y_{v_{min}} + (y_w - y_{w_{min}})sy$$

where the scaling factors are

$$sx = \frac{x_{v_{max}} - x_{v_{min}}}{x_{w_{max}} - x_{w_{min}}}$$

$$sy = \frac{y_{v_{max}} - y_{v_{min}}}{y_{w_{max}} - y_{w_{min}}}$$

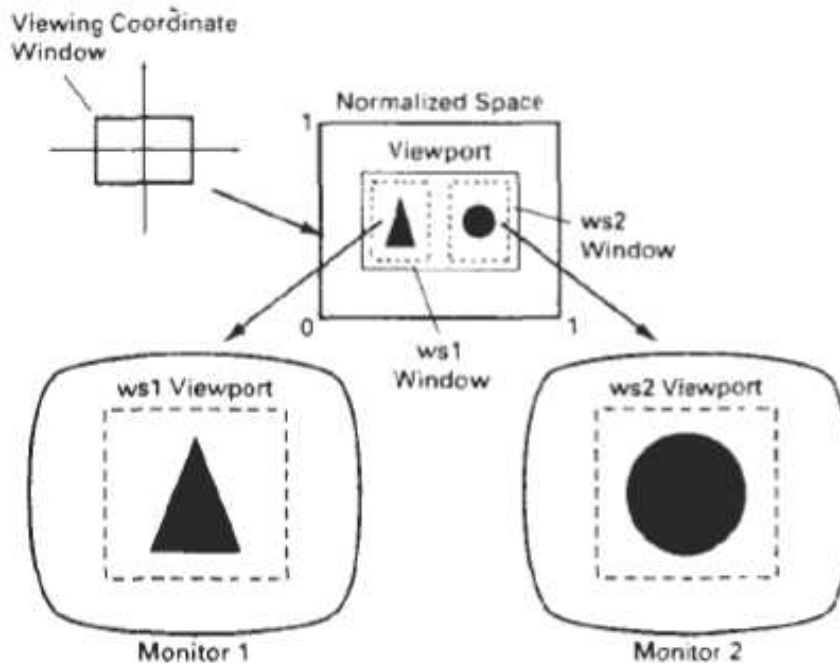
The Equations can also be derived with a set of transformations that converts the window area into the viewport area.

This conversion is performed with the following sequence of transformations:

1. Perform a scaling transformation using a fixed-point position of $(x_{w_{min}}, y_{w_{min}})$ 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-dimensional views; antialiasing line segments 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
- ☞ Line Clipping (straight-line segments)
- ☞ Area Clipping (polygons)
- ☞ Curve Clipping
- ☞ Text Clipping

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:

$$xw_{min} \leq x \leq xw_{max}$$

$$yw_{min} \leq y \leq yw_{max}$$

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:

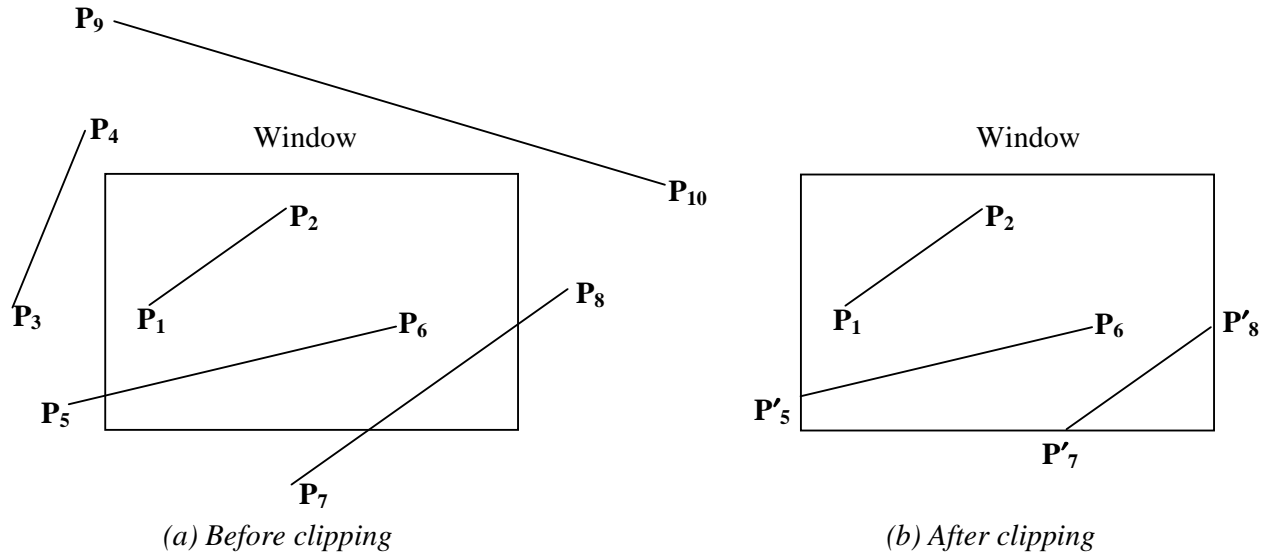
Explain the steps to perform Line Clipping.

(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 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" tests 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

$$\begin{aligned}x &= x_1 + u(x_2 - x_1) \\ y &= y_1 + u(y_2 - y_1), 0 \leq u \leq 1\end{aligned}$$

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:

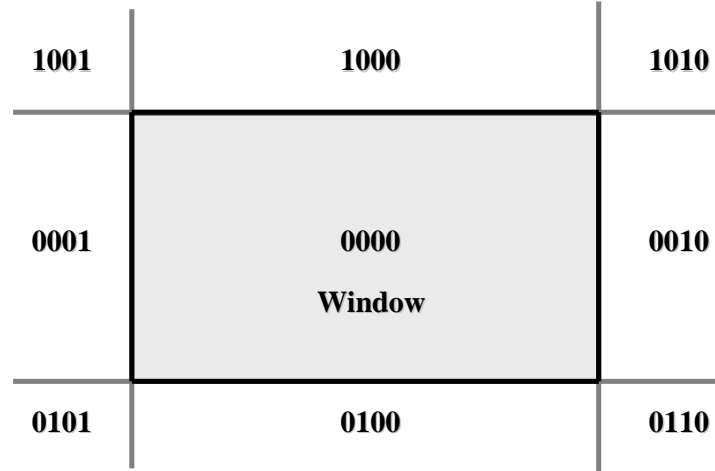
This is one of the oldest and most popular line-clipping procedures. Generally, the method speeds up the processing of line segments by performing initial tests that reduce the number of intersections that must be calculated.

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

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**.

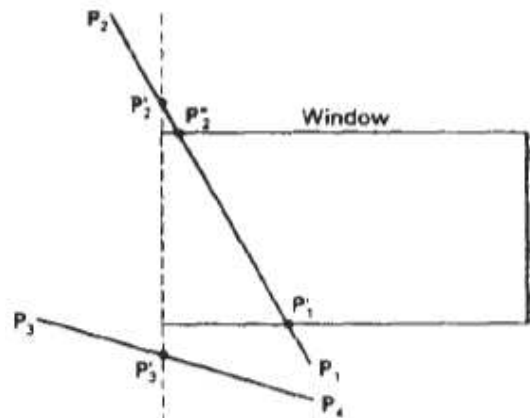


The region-code bit values can be determined with the following two steps:

- (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 **P₁** to **P₂**, we check **P₁** 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 **P₁** to **P'₁**. The line now has been reduced to the section from **P'₁** to **P₂**. Since **P₂** 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

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

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

$$\begin{aligned}x &= x_1 + u \Delta x \\ y &= y_1 + u \Delta y, \quad 0 \leq u \leq 1\end{aligned}$$

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:

$$\begin{aligned}xw_{min} &\leq x_1 + u \Delta x \leq xw_{max} \\ yw_{min} &\leq y_1 + u \Delta y \leq yw_{max}\end{aligned}$$

Each of these four inequalities can be expressed as

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

where parameters p and q are defined as

$$\begin{aligned}p_1 &= -\Delta x, & q_1 &= x_1 - xw_{min} \\ p_2 &= \Delta x, & q_2 &= xw_{max} - x_1 \\ p_3 &= -\Delta y, & q_3 &= y_1 - yw_{min} \\ p_4 &= \Delta y, & q_4 &= yw_{max} - y_1\end{aligned}$$

Any line that is parallel to one of the clipping boundaries has $p_k = 0$ for the value of k corresponding to that boundary ($k = 1, 2, 3, \text{ and } 4$ 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 $q_k \geq 0$, 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 $p_k > 0$, 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

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

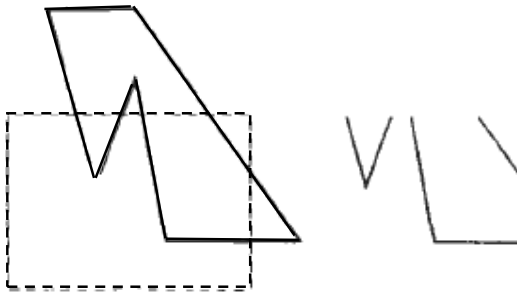
For each line, we can calculate values for parameters u_1 and u_2 that define that part of the line that lies within the clip rectangle. The value of u_1 is determined by looking at the rectangle edges for which the line proceeds from the outside to the inside ($p < 0$). For these edges, we calculate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

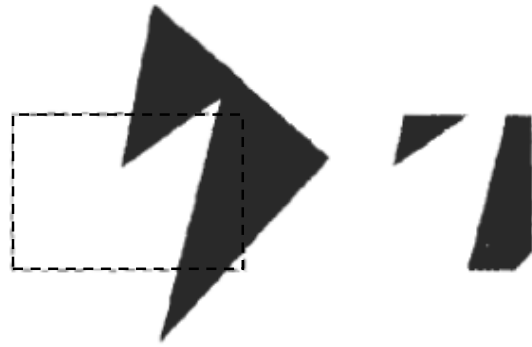
For polygon clipping, we require an algorithm that will generate one or more closed areas that are then scan converted for the appropriate area fill. The output of a polygon clipper should be a sequence of vertices that defines the clipped polygon boundaries.



Before Clipping

Fig. (a)

After Clipping



Before Clipping

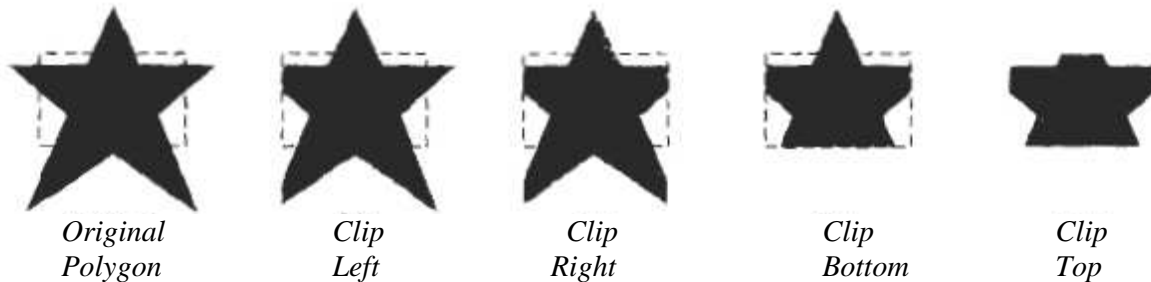
Fig. (b)

After Clipping

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.

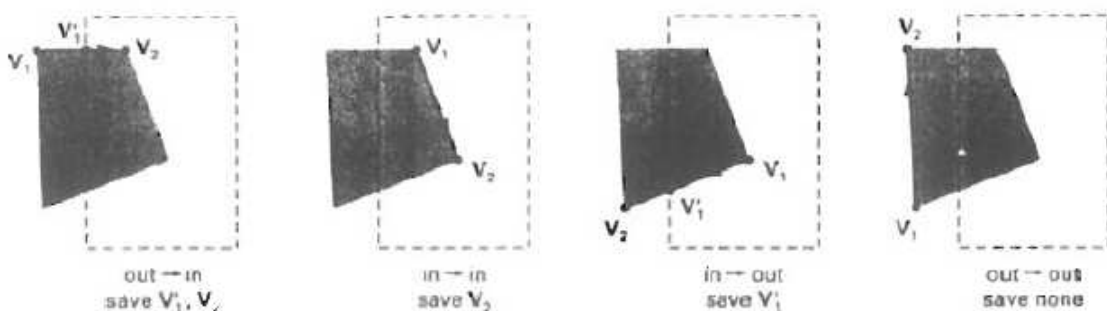
Beginning with the initial set of polygon vertices, we could first clip the polygon against the left rectangle boundary to produce a new sequence of vertices. The new set of vertices could then be successively passed to a right boundary clipper, a bottom boundary clipper, and a top boundary clipper, 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.



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:

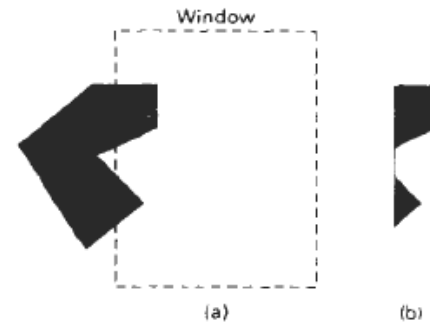
- (1) If the first vertex is outside the window boundary and the second vertex is inside, both the intersection point of the polygon edge with the window boundary and the second vertex are added to the output vertex list.
- (2) If both input vertices are inside the window boundary, only the second vertex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



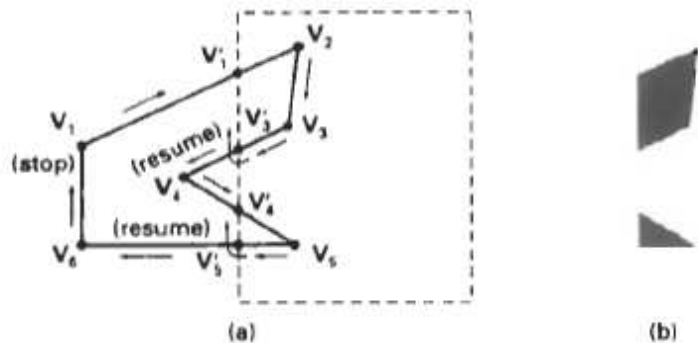
Weiler-Atherton Polygon Clipping:

Here, the vertex-processing procedures for window boundaries are modified so that concave polygons are displayed correctly. This clipping procedure was developed as a method for identifying visible surfaces, and so it can be applied with arbitrary polygon-clipping regions.

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

For clockwise processing of polygon vertices, we use the following rules:

- ⇒ For an outside-to-inside pair of vertices, follow the polygon boundary.
- ⇒ For an inside-to-outside pair of vertices, follow the window boundary in a clockwise direction.



In the figure, the processing direction in the Weiler-Atherton algorithm and the resulting clipped polygon is shown for a rectangular clipping window.

An improvement on the Weiler-Atherton algorithm is the Weiler algorithm, which applies constructive solid geometry ideas to clip an arbitrary polygon against any polygon clipping region. The following diagram illustrates the general idea in this approach.

For the two polygons, the correctly clipped polygon is calculated as the intersection of the clipping polygon and the polygon object.

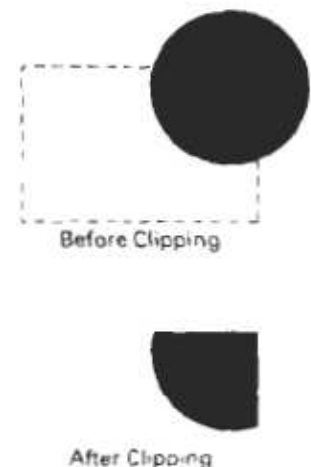


CURVE CLIPPING:

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.

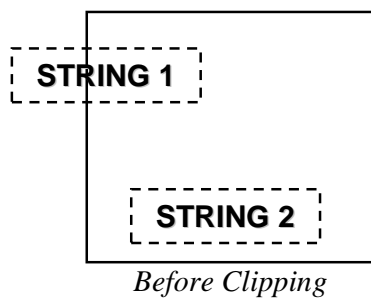
Similar procedures can be applied when clipping a curved object against a general polygon clip region. On the first pass, we can clip the



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.

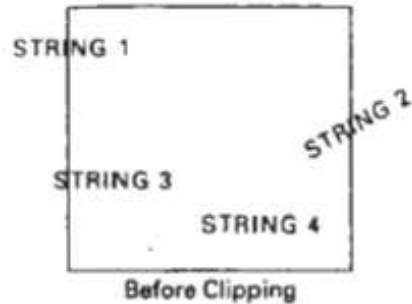
TEXT CLIPPING:

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.



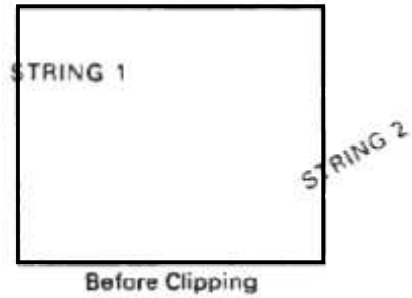
Before Clipping

After Clipping
fig. (1)



Before Clipping

After Clipping
fig. (2)



Before Clipping

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 of 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.

❧ End of Unit – IV ❧

GRAPHICAL USER INTERFACES AND INTERACTIVE INPUT METHODS

USER DIALOGUE:

For a particular application, the *user's model* serves as the basis for the design for the dialogue. The user's model describes what the system is designed to accomplish and what graphics operations are available. It states the type of objects that can be displayed and how the objects can be manipulated.

For example, if the graphics system is to be used as a tool for architectural design, the model describes how the package can be used to construct and display views of buildings by positioning walls, doors, windows, and other building components.

All information in the user dialogue is then presented in the language of the application. In an architectural design package, this means that all interactions are described only in architectural terms, without reference to particular data structures or other concepts that may be unfamiliar to an architect.

Windows and Icons:

- Visual representations are used both for objects to be manipulated in an application and for the actions to be performed on the application objects.
- A window system provides a window-manager interface for the user and functions for handling the display and manipulation of the windows.
- Common functions for the window system are opening and closing windows, repositioning windows, resizing windows, and display routines that provide interior and exterior clipping and other graphics functions.
- Typically, windows are displayed with sliders, buttons, and menu icons for selecting various window options.
- Some general systems, such as X Windows and NeWS, are capable of supporting multiple window managers so that different window styles can be accommodated, each with its own window manager. The window managers can then be designed for particular applications.
- Icons representing objects such as furniture items and circuit elements are often referred to as ***application icons***.
- The icons representing actions, such as rotate, magnify, scale, clip, and paste, are called ***control icons***, or ***command icons***.

Accommodating Multiple Skill Levels:

- Usually, interactive graphical interfaces provide several methods for selecting actions.
 - For example, options could be selected by pointing at an icon and clicking different mouse buttons, or by accessing pull-down or pop-up menus, or by typing keyboard commands. This allows a package to accommodate users that have different skill levels.
- A simplified set of menus and options is easy to learn and remember, and the user can concentrate on the application instead of on the details of the interface.
- Interfaces typically provide a means for masking the complexity of a package, so that beginners can use the system without being overwhelmed with too much detail.
- Experienced users typically want speed. This means fewer prompts more input from the keyboard or with multiple mouse-button clicks.
- Actions are selected with function keys or with simultaneous combinations of keyboard keys, since experienced users will remember these shortcuts for commonly used actions.

Consistency:

- An important design consideration in an Interface is Consistency.
 - For example, a particular icon shape should always have a single meaning, rather than serving to represent different actions or objects depending on the context.
- Generally, a complicated, inconsistent model is difficult for a user to understand and to work with in an effective way.
- The objects and operations provided should be designed to form a minimum and consistent set so that the system is easy to learn, but not oversimplified to the point where it is difficult to apply.

Minimizing Memorization:

- Operations in an interface should also be structured so that they are easy to understand and to remember.

- | Obscure, complicated and inconsistent and abbreviated command formats lead to confusion and reduction in the effectiveness of the use of the package.
- | One key or button used for all delete operations.
 - For example, is easier to remember than a number of different keys for different types of delete operations.
- | Icons and window systems also aid in minimizing memorization.
- | Icons are used to reduce memorizing by displaying easily recognizable shapes for various objects and actions. To select a particular action, we simply select the icon that resembles that action.

Backup and Error Handling:

- | A mechanism for backing up, or aborting, during a sequence of operations is another common feature of an interface. Often an operation can be canceled before execution is completed, with the system restored to the state it was in before the operation was started.
- | With the ability to back up; it any point, we can confidently explore the capabilities of the system, knowing that the effects of a mistake can be erased.
- | Backup can be provided in many forms.
 - A standard undo key or command is used to cancel a single operation.
 - Sometimes a system can be backed up through several operations, allowing us to reset the system to some specified point.
- | In a system with extensive backup capabilities, all inputs could be saved so that we can back up and “replay” any part of a session.
- | Good diagnostics and error messages are designed to help determine the cause of an error.
- | Additionally, interfaces attempt to minimize error possibilities by anticipating certain actions that could lead to an error.

Feedback:

- | Interfaces are designed to carry on a continual interactive dialogue so that we are informed of actions in progress at each step. This is particularly important when the response time is high.
- | Without feedback, we might begin to wonder what the system is doing and whether the input should be given again.
- | As each input is received, the system normally provides some type of response. An object is highlighted, an icon appears, or a message is displayed. This not only informs us that the input has been received, but it also tells us what the system is doing.
- | If processing cannot be completed within a few seconds, several feedback messages might be displayed to keep us informed of the progress of the system.
- | To speed system response, feedback techniques can be chosen to take advantage of the operating characteristics of the type of devices in use.
- | Other feedback methods include highlighting, blinking, and color changes.
- | Special symbols are designed for different types of feedback.
 - For example, a cross, a frowning face, or a thumbs-down symbol is often used to indicate an error; and a blinking “at work” sign is used to indicate that processing is in progress.
- | With some types of input, *echo* feedback is desirable. Typed characters can be displayed on the screen as they are input so that we can detect and correct errors immediately.

INPUT OF GRAPHICAL DATA:

Graphics programs use several kinds of input data. Picture specifications need values for coordinate positions, values for the character-string parameters, scalar values for the transformation parameters, values specifying menu options, and values for identification of picture parts.

To make graphics packages independent of the particular hardware devices used, input functions can be structured according to the data description to be handled by each function. This approach provides a *logical input-device classification* in terms of the kind of data to be input by the device.

Logical Classification of Input Devices:

The various kinds of input data are summarized in the following six logical device classifications used by PHIGS and GKS:

- | **LOCATOR** — a device for specifying a coordinate position (x, y)
- | **STROKE** — a device for specifying a series of coordinate positions

- **STRING** – a device for specifying text input
- **VALUATOR** – a device for specifying scalar values
- **CHOICE** – a device for selecting menu options
- **PICK** – a device for selecting picture components

Locator Devices:

- ⇒ A standard method for interactive selection of a coordinate point is by positioning the screen cursor. We can do this with a mouse, joystick, trackball, spaceball, thumbwheels, dials, a digitizer stylus or hand cursor, or some other cursor-positioning device. When the screen cursor is at the desired location, a button is activated to store the coordinates of that screen point.
- ⇒ Keyboards can be used for locator input in several ways. A general-purpose keyboard usually has four cursor-control keys that move the screen cursor up, down, left, and right.
- ⇒ Alternatively, a joystick, joydisk, trackball, or thumbwheels can be mounted on the keyboard for relative cursor movement.
- ⇒ Light pens have also been used to input coordinate positions, but some special implementation considerations are necessary. Since light pens operate by detecting light emitted from the screen phosphors, some nonzero intensity level must be present at the coordinate position to be selected.

Stroke Devices:

- ⇒ This class of logical devices is used to input a sequence of coordinate positions.
- ⇒ Stroke-device input is equivalent to multiple calls to a locator device. The set of input points is often used to display line sections.
- ⇒ Continuous movement of a mouse, trackball, joystick, or tablet hand cursor is translated into a series of input coordinate values.
- ⇒ The graphics tablet is one of the more common stroke devices. Button activation can be used to place the tablet into “continuous” mode. As the cursor is moved across the tablet surface, a stream of coordinate values is generated. This process is used in paintbrush systems that allow artists to draw scenes on the screen and in engineering systems where layouts can be traced and digitized for storage.

String Devices:

- ⇒ The primary physical device used for string input is the keyboard. Input character strings are typically used for picture or graph labels.
- ⇒ Other physical devices can be used for generating character patterns in a “text-writing” mode. For this input, individual characters are drawn on the screen with a stroke or locator-type device. A pattern-recognition program then interprets the characters using a stored dictionary of predefined patterns.

Valuator Devices:

- ⇒ These devices are employed in graphic systems to input scalar values. Valuers are used for setting various graphics parameters, such as rotation angle and scale factors, and for setting physical parameters associated with a particular application (temperature settings, voltage levels, stress factors, etc.).
- ⇒ A typical physical device used to provide valuator input is a set of control dials. Floating-point numbers within any predefined range are input by rotating the dials.
- ⇒ Dial rotations in one direction increase the numeric input value, and opposite rotations decrease the numeric value. Rotary potentiometers convert dial rotation into a corresponding voltage. This voltage is then translated into a real number within a defined scalar range, such as -10.5 to 25.5. Instead of dials, slide Potentiometers are sometimes used to convert linear movements into scalar values.
- ⇒ Joystick, trackball, tablets, and other interactive devices can be adapted for valuator input by interpreting pressure or movement of the device relative to a scalar range.
- ⇒ Another technique for providing valuator input is to display sliders, buttons, rotating scales, and menus on the video monitor.

Choice Devices:

- ⇒ Graphics packages use menus to select programming options, parameter values, and object shapes to be used in constructing a picture.
- ⇒ A choice device is defined as one that enters a selection from a list (menu) of alternatives. Commonly used choice devices are a set of buttons; a cursor positioning device, such as a mouse, trackball, or keyboard cursor keys; and a touch panel.

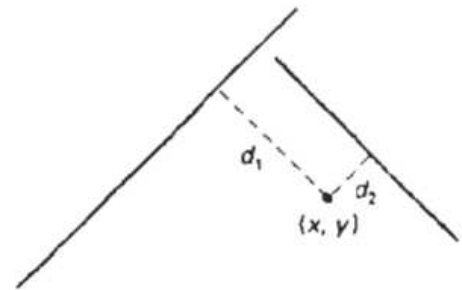
- ⇒ For screen selection of listed menu options, we can use cursor-control devices. When a coordinate position (x, y) is selected, it is compared to the coordinate extents of each listed menu item. A menu item with vertical and horizontal boundaries at the coordinate values x_{min} , x_{max} , y_{min} and y_{max} is selected if the input coordinates (x, y) satisfy the inequalities

$$x_{min} \leq x \leq x_{max}, \quad y_{min} \leq y \leq y_{max}$$

- ⇒ For larger menus with a few options displayed at a time, a touch panel is commonly used.
- ⇒ Alternate methods for choice input include keyboard and voice entry. A standard keyboard can be used to type in commands or menu options.
- ⇒ Similar coding can be used with voice-input systems. Voice input is particularly useful when the number of options is small (20 or less).

Pick Devices:

- ⇒ Pick devices are used to select parts of a scene that are to be transformed or edited in some way.
- ⇒ Typical devices used for object selection are the same as those for menu selection: the cursor-positioning devices.
- ⇒ With a mouse or joystick, we can position the cursor over the primitives in a displayed structure and press the selection button. The position of the cursor is then recorded, and several levels of search may be necessary to locate the particular object (if any) that is to be selected.
- ⇒ First, the cursor position is compared to the coordinate extents of the various structures in the scene. If the bounding rectangle of a structure contains the cursor coordinates, the picked structure has been identified.
- ⇒ One way to find the closest line to the cursor position is to calculate the distance squared from the cursor coordinates (x, y) to each line segment whose bounding rectangle contains the cursor position as shown in the diagram.

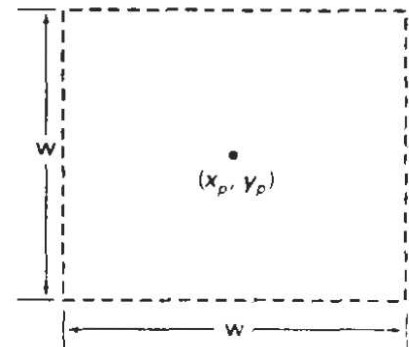


- ⇒ For a line with endpoints (x_1, y_1) and (x_2, y_2) , distance squared from (x, y) to the line is calculated as

$$d^2 = \frac{[\Delta x(y - y_1) - \Delta y(x - x_1)]^2}{\Delta x^2 + \Delta y^2}$$

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

- ⇒ Another method for finding the closest line to the cursor position is to specify the size of a **pick window**.
- ⇒ The cursor coordinates are centered on this window and the candidate lines are clipped to the window, as shown in the diagram.
- ⇒ By making the pick window small enough, we can ensure that a single line will cross the window.
- ⇒ We could use a keyboard to type in structure names. Descriptive names can be used to help the user in the pick process, but the method has several drawbacks.
- ⇒ It is generally slower than interactive picking on the screen, and a user will probably need prompts to remember the various structure names.
- ⇒ In addition, picking structure subparts from the keyboard can be more difficult than picking the subparts on the screen.



INPUT FUNCTIONS

Graphical input functions can be set up to allow users to specify the following options:

- ⇒ Which physical devices are to provide input within a particular logical classification (for example, a tablet used as a stroke device).
- ⇒ How the graphics program and devices are to interact (input mode). Either the program or the devices can initiate data entry, or both can operate simultaneously.
- ⇒ When the data are to be input and which device is to be used at that time to deliver a particular input type to the specified data variables.

Input Modes:

Functions to provide input can be structured to operate in various input modes, which specify how the program and input devices interact. Input could be initiated by the program, or the program and input devices both could be operating simultaneously, or data input could be initiated by the devices. These three input modes are referred to as **request mode**, **sample mode**, and **event mode**.

- ∞ In **request mode**, the application program initiates data entry. Input values are requested and processing is suspended until the required values are received.
- ∞ In **sample mode**, the application program and input devices operate independently. Input devices may be operating at the same time that the program is processing other data.
- ∞ In **event mode**, the input devices initiate data input to the application program. The program and the input devices again operate concurrently, but now the input devices deliver data to an input queue. All input data are saved. When the program requires new data, it goes to the data queue.

An input mode within a logical class for a particular physical device operating on a specified workstation is declared with one of six input-class functions of the form

```
set . . . Mode (ws, deviceCode, inputMode, echoFlag)
```

where devicecode is a positive integer; inputMode is assigned one of the values: *request*, *sample*, or *event*; and parameter echoFlag is assigned either the value *echo* or the value *noecho*.

Request Mode:

Input commands used in this mode correspond to standard input functions in a high-level programming language. When we ask for an input in request mode, other processing is suspended until the input is received. Input requests can be made to that device using one of the six logical-class functions represented by the following:

```
request . . . (ws, devicecode, status, . . . . )
```

Values input with this function are the workstation code and the device code. Returned values are assigned to parameter status and to the data parameters corresponding to the requested logical class.

A value of ok or nonc is returned in parameter status, according to the validity of the input data. A value of none indicates that the input device was activated so as to produce invalid data.

A returned value of none can be used as an end-of-data signal to terminate a programming sequence.

Locator and Stroke Input in Request Mode

The request functions for these two logical input classes are:

```
requestLocator (ws, devcode, status , viewIndex, pt)  
requestStroke (ws, devCode, nMax, status , viewIndex, n , pts)
```

For locator input, pt is the world-coordinate position selected. For stroke input, pts is a list of n coordinate positions, where parameter nMax gives the maximum number of points that can go in the input list. Parameter viewIndex is assigned the two-dimensional view index number.

Determination of a world-coordinate position is a two-step process: (1) The physical device selects a point in device coordinates (usually from the video-display screen) and the inverse of the workstation transformation is performed to obtain the corresponding point in normalized device coordinates. (2) Then, the inverse of the window-to-viewport mapping is carried out to get to viewing coordinates, then to world coordinates.

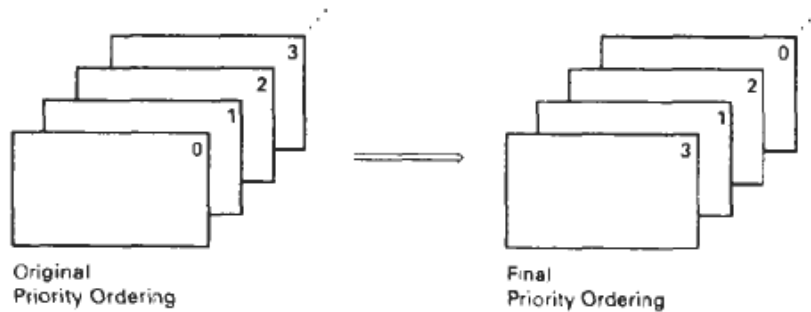
Since two or more views may overlap on a device, the correct viewing transformation is identified according to the view-transformation input priority number. View index 0 has the highest priority. We can change the view priority relative to another (reference) viewing transformation with

```
setViewTransformationInputPriority (ws, viewIndex, refViewIndex, priority)
```

where viewIndex identifies the viewing transformation whose priority is to be changed, refViewIndex identifies the reference viewing transformation, and parameter priority is assigned either the value lower or the value higher.

ASSIGNMENT OF INPUT-DEVICE
CODES

Device Code	Physical Device Type
1	Keyboard
2	Graphics Tablet
3	Mouse
4	Joystick
5	Trackball
6	Button



String Input in Request Mode:

Here, the request input function is

```
requestString (ws, devcode, status, nChars, str)
```

Parameter `str` in this function is assigned an input string. The number of characters in the string is given in parameter `nChars`.

Valuator Input in Request Mode:

A numerical value is input in request mode with

```
requestvaluator (ws, devcode, status, value)
```

Parameter `value` can be assigned any real-number value.

Choice Input in Request Mode

We make a menu selection with the following request function:

```
requestchoice (ws, devCode, status, itemNum)
```

Parameter `itemNum` is assigned a positive integer value corresponding to the menu item selected.

Pick Input in Request Mode

For this mode, we obtain a structure identifier number with the function

```
requestpick (ws, devCode, maxPathDepth, status, pathDepth, pickpath)
```

Parameter `pickpath` is a list of information identifying the primitive selected. This list contains the structure name, pick identifier for the primitive, and the element sequence number. Parameter `pickDepth` is the number of levels returned in `pickpath`, and `maxPathDepth` is the specified maximum path depth that can be included in `pickpath`.

Sample Mode:

Once sample mode has been set for one or more physical devices, data input begins without waiting for program direction. Sampling of the current values from a physical device in this mode begins when a sample command is encountered in the application program. A locator device is sampled with one of the six logical-class functions represented by the following:

```
sample . . . (ws, deviceCode, . . .)
```

A final translation position for the object can be obtained with a locator, and the rotation angle can be supplied by a valuator device, as demonstrated in the following statements.

```
sampleLocator (ws1, dev1, viewIndex, pt)
```

```
sampleValuator (ws2, dev2, angle)
```

Event Mode:

When an input device is placed in event mode, the program and device operate simultaneously. Data input from the device is accumulated in an event queue, or input queue. All input devices active in event mode can enter data (referred to as "events") into this single-event queue, with each device entering data values as they are generated.

An application program can be directed to check the event queue for any input with the function

```
awaitEvent (time, ws, deviceClass, deviceCode)
```

Some additional housekeeping functions can be used in event mode. Functions for clearing the event queue are useful when a process is terminated and a new application is to begin. These functions can be set to clear the entire queue or to clear only data associated with specified input devices and workstations.

Concurrent Use of Input Modes:

An object is dragged around the screen with a mouse. When a final position has been selected, a button is pressed to terminate any further movement of the object. The mouse positions are obtained in sample mode, and the button input is sent to the event queue.

INTERACTIVE PICTURE-CONSTRUCTION TECHNIQUES

There are several techniques that are incorporated into graphics packages to aid the interactive construction of pictures. Input coordinates can establish the position or boundaries for objects to be drawn, or they can be used to rearrange previously displayed objects.

Basic Positioning Methods:

Coordinate values supplied by locator input are often used with positioning methods to specify a location for displaying an object or a character string. We interactively select coordinate positions with a pointing device, usually by positioning the screen cursor.

For lines, straight line segments can be displayed between two selected screen positions. As an aid in positioning objects, numeric values for selected positions can be echoed on the screen. Using the echoed coordinate values as a guide, we can make adjustments in the selected location to obtain accurate positioning.

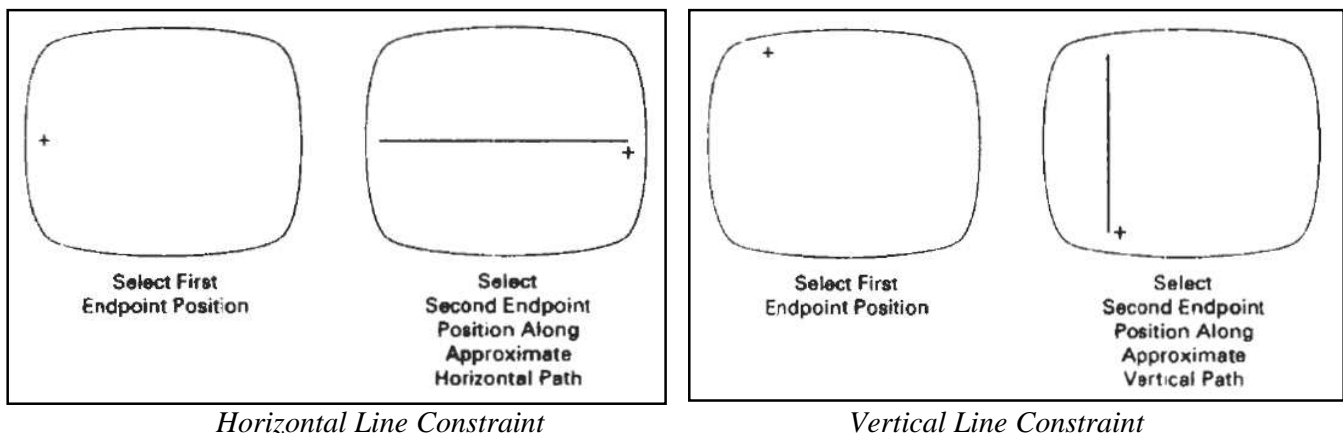
Constraints:

With some applications, certain types of prescribed orientations or object alignments are useful. A constraint is a rule for altering input-coordinate values to produce a specified orientation or alignment of the displayed coordinates. There are many kinds of constraint functions that can be specified, but the most common constraint is a horizontal or vertical alignment of straight lines. This type of constraint, shown in the following diagrams, is useful in forming network layouts.

With this constraint, we can create horizontal and vertical lines without worrying about precise specification of endpoint coordinates.

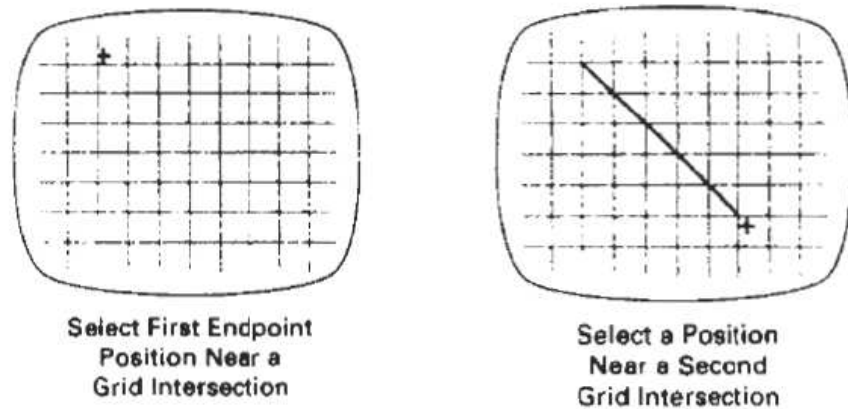
A horizontal or vertical constraint is implemented by determining whether any two input coordinate endpoints are more nearly horizontal or more nearly vertical. If the difference in the y values of the two endpoints is smaller than the difference in x values, a horizontal line is displayed. Otherwise, a vertical line is drawn.

Other kinds of constraints can be applied to input coordinates to produce a variety of alignments. Lines could be constrained to have a particular slant, such as 45", and input coordinates could be constrained to lie along predefined paths, such as circular arcs.



Grids:

Another kind of constraint is a grid of rectangular lines displayed in some part of the screen area. When a grid is used, any input coordinate position is rounded to the nearest intersection of two grid lines.



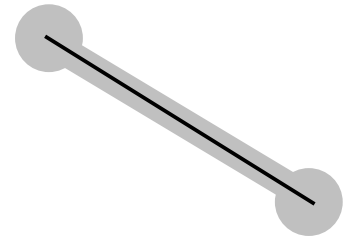
The above diagram illustrates line drawing with a grid. Each of the two cursor positions is shifted to the nearest grid intersection point, and the line is drawn between these grid points. Grids facilitate object constructions, because a new line can be joined easily to a previously drawn line by selecting any position near the endpoint grid intersection of one end of the displayed line.

Gravity Field:

In the construction of figures, we sometimes need to connect lines at positions between endpoints. Since exact positioning of the screen cursor at the connecting point can be difficult, graphics packages can be designed to convert any input position near a line to a position on the line.

This conversion of input position is accomplished by creating a gravity field area around the line. Any selected position within the gravity field of a line is moved ("gravitated") to the nearest position on the line.

A gravity field area around a line is illustrated with the shaded boundary as shown in the diagram. Areas around the endpoints are enlarged to make it easier for us to connect lines at their endpoints. Selected positions in one of the circular areas of the gravity field are attracted to the endpoint in that area. If many lines are displayed, gravity areas can overlap, and it may be difficult to specify points correctly. Normally, the boundary for the gravity field is not displayed.



Rubber-Band Method:

Straight lines can be constructed and positioned using Rubber-Band methods, which stretch out a line from a starting position as the screen cursor is moved. The following figure demonstrates the rubber-band method.

We first select a screen position for one endpoint of the line. Then, as the cursor moves around, the line is displayed from the start position to the current position of the cursor. When we finally select a second screen position, the other line endpoint is set. Rubber-band methods are used to construct and position other objects besides straight lines.

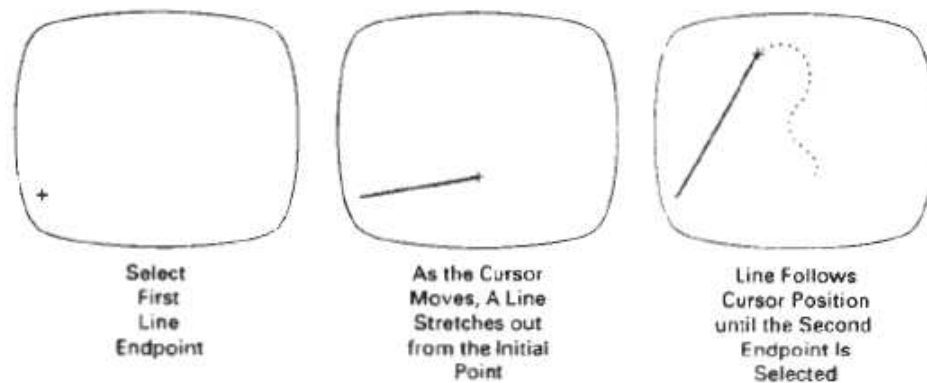


Figure (a) demonstrates rubber-band construction of a rectangle, and Figure (b) shows a rubber-band circle construction.

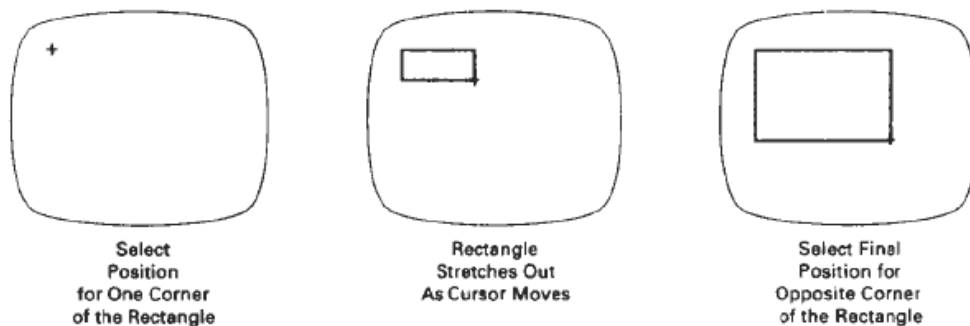


Fig. (a) Rubber-Band method for constructing a Rectangle

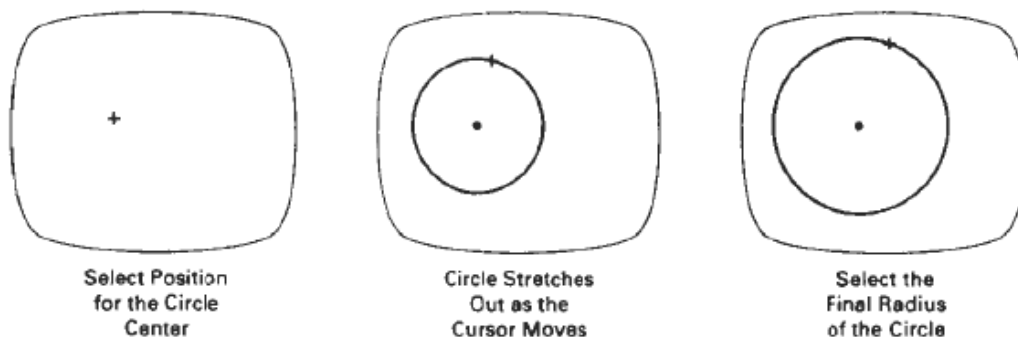


Fig. (b) Rubber-Band method for constructing a Circle

Dragging:

A technique that is often used in interactive picture construction is to move objects into position by dragging them with the screen cursor. We first select an object, and then move the cursor in the direction we want the object to move, and the selected object follows the cursor path.

Painting and Drawing:

Options for sketching, drawing, and painting come in a variety of forms. Curve-drawing options can be provided using standard curve shapes, such as circular arcs and splines, or with freehand sketching procedures. Splines are interactively constructed by specifying a set of discrete screen points that give the general shape of the curve. Then the system fits the set of points with a polynomial curve. In freehand drawing, curves are generated by following the path of a stylus on a graphics tablet or the path of the screen cursor on a video monitor. Once a curve is displayed, the designer can alter the curve shape by adjusting the positions of selected points along the curve path.

End of Unit – V