If you create an act, you create a habit. If you create a habit, you create a character. If you create a character, you create a destiny.

Reputation is what men and women think of us. Character is what God and the angels know of us.

1. Overview: Computer Graphics and OpenGL

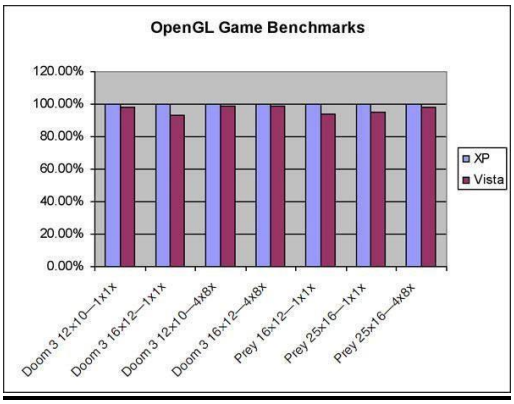
Basics of computer graphics
Application of Computer Graphics,
Video Display Devices
Random Scan and Raster Scan displays,
Color CRT monitors,
1.3.4 Flat panel displays.
Raster-scan systems:
Video controller,
Raster scan Display processor,
Graphics workstations and viewing systems,
Input devices,
Graphics networks,
Graphics on the internet,
Graphics software.
OpenGL:
Introduction to OpenGL ,
Coordinate reference frames,
Specifying two-dimensional world coordinate reference frames in OpenGL,
OpenGL point functions,
OpenGL line functions, point attributes,
Line attributes,
Curve attributes,
OpenGL point attribute functions,
OpenGL line attribute functions,
Line drawing algorithms(DDA, Bresenham's),
Circle generation algorithms (Bresenham's).

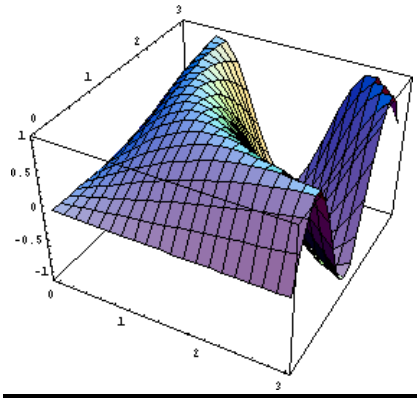
Basics of Computer Graphics

Computer graphics is an art of drawing pictures, lines, charts, etc. using computers with the help of programming. Computer graphics image is made up of number of pixels. Pixel is the smallest addressable graphical unit represented on the computer screen.

Applications of Computer Graphics

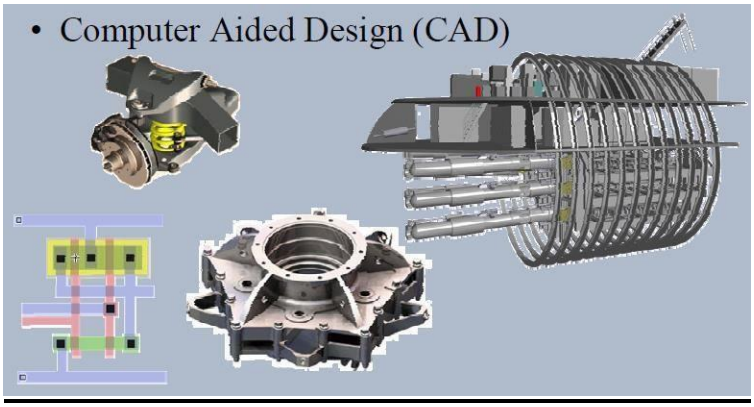
a. Graphs and Charts





- ✓ An early application for computer graphics is the display of simple data graphs usually plotted on a character printer. Data plotting is still one of the most common graphics application.
- ✓ Graphs & charts are commonly used to summarize functional, statistical, mathematical, engineering and economic data for research reports, managerial summaries and other types of publications.
- ✓ Typically examples of data plots are line graphs, bar charts, pie charts, surface graphs, contour plots and other displays showing relationships between multiple parameters in two dimensions, three dimensions, or higher-dimensional spaces

b. Computer-Aided Design



- ✓ A major use of computer graphics is in design processes-particularly for engineering and architectural systems.

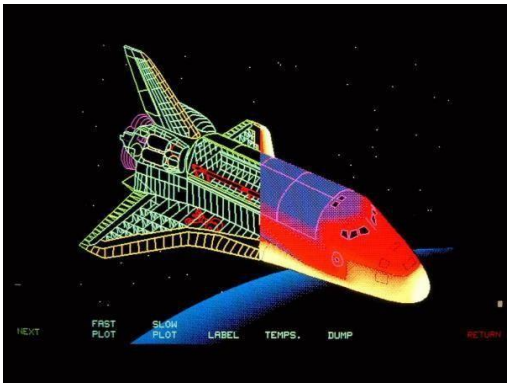
- ✓ CAD, computer-aided design or CADD, computer-aided drafting and design methods are now routinely used in the automobiles, aircraft, spacecraft, computers, home appliances.
- ✓ Circuits and networks for communications, water supply or other utilities are constructed with repeated placement of a few geographical shapes.
- ✓ Animations are often used in CAD applications. Real-time, computer animations using wire-frame shapes are useful for quickly testing the performance of a vehicle or system.

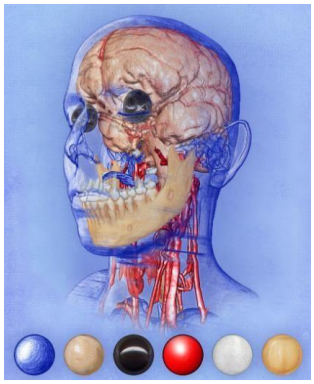
c. Virtual-Reality Environments

- ✓ Animations in virtual-reality environments are often used to train heavy-equipment operators or to analyze the effectiveness of various cabin configurations and control placements.
- ✓ With virtual-reality systems, designers and others can move about and interact with objects in various ways. Architectural designs can be examined by taking simulated "walk" through the rooms or around the outsides of buildings to better appreciate the overall effect of a particular design.
- ✓ With a special glove, we can even "grasp" objects in a scene and turn them over or move them from one place to another.

d. Data Visualizations

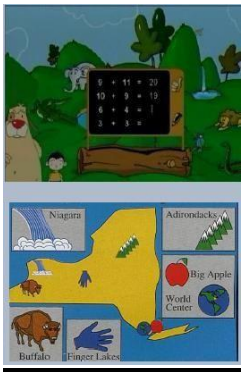
- ✓ Producing graphical representations for scientific, engineering and medical data sets and processes is another fairly new application of computer graphics, which is generally referred to as scientific visualization. And the term business visualization is used in connection with data sets related to commerce, industry and other nonscientific areas.





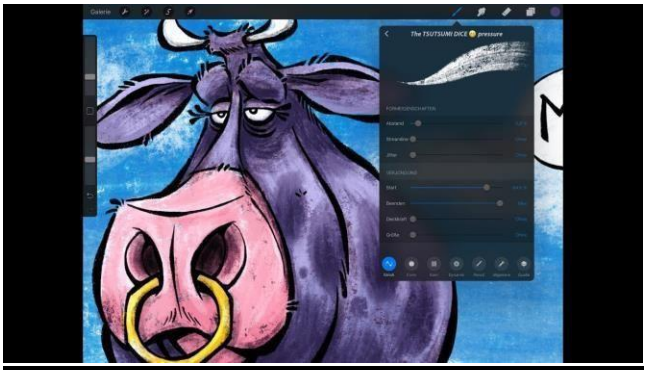
- ✓ There are many different kinds of data sets and effective visualization schemes depend on the characteristics of the data. A collection of data can contain scalar values, vectors or higher-order tensors.

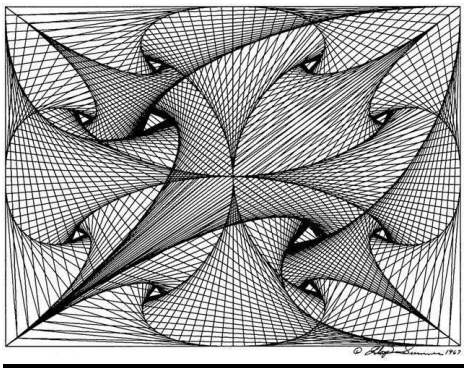
e. Education and Training



- ✓ Computer generated models of physical,financial,political,social,economic & other systems are often used as educational aids.
- ✓ Models of physical processes physiological functions,equipment, such as the color coded diagram as shown in the figure, can help trainees to understand the operation of a system.
- ✓ For some training applications,special hardware systems are designed.Examples of such specialized systems are the simulators for practice sessions ,aircraft pilots,air traffic-control personnel.
- ✓ Some simulators have no video screens,for eg: flight simulator with only a control panel for instrument flying

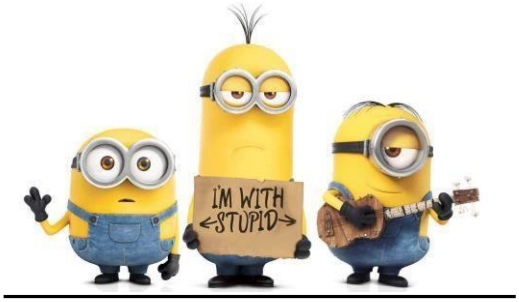
f. Computer Art





- ✓ The picture is usually painted electronically on a graphics tablet using a stylus, which can simulate different brush strokes, brush widths and colors.
- ✓ Fine artists use a variety of other computer technologies to produce images. To create pictures the artist uses a combination of 3D modeling packages, texture mapping, drawing programs and CAD software etc.
- ✓ Commercial art also uses theses "painting" techniques for generating logos & other designs, page layouts combining text & graphics, TV advertising spots & other applications.
- ✓ A common graphics method employed in many television commercials is morphing, where one object is transformed into another.

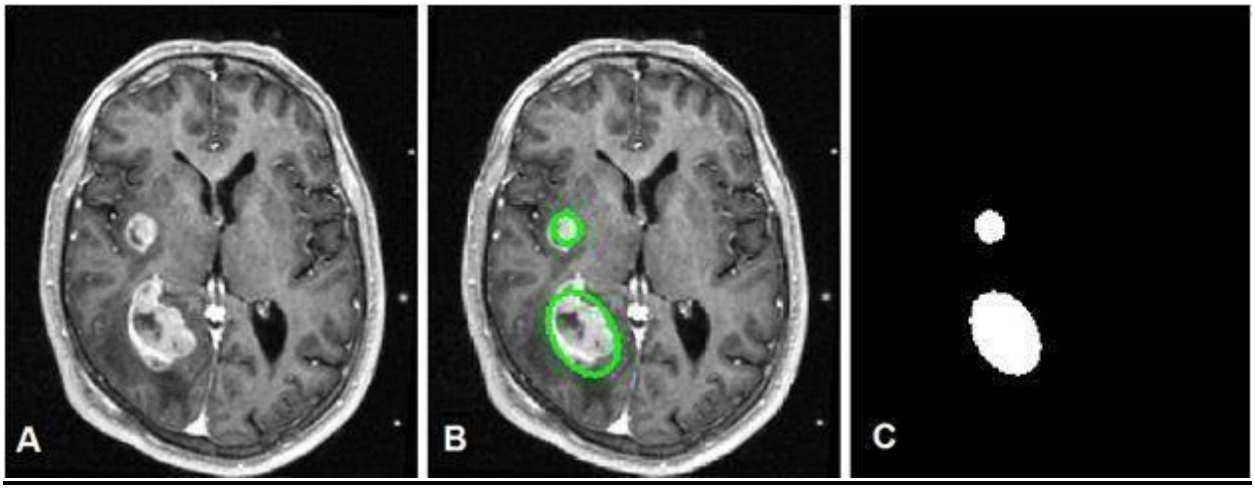
g. Entertainment



- ✓ Television production, motion pictures, and music videos routinely a computer graphics methods.
- ✓ Sometimes graphics images are combined a live actors and scenes and sometimes the films are completely generated a computer rendering and animation techniques.

- ✓ Some television programs also use animation techniques to combine computer generated figures of people, animals, or cartoon characters with the actor in a scene or to transform an actor's face into another shape.

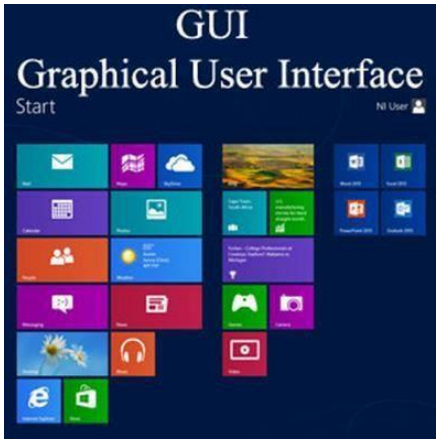
h. Image Processing



- ✓ The modification or interpretation of existing pictures, such as photographs and TV scans is called image processing.
- ✓ Methods used in computer graphics and image processing overlap, the two areas are concerned with fundamentally different operations.
- ✓ Image processing methods are used to improve picture quality, analyze images, or recognize visual patterns for robotics applications.
- ✓ Image processing methods are often used in computer graphics, and computer graphics methods are frequently applied in image processing.
- ✓ Medical applications also make extensive use of image processing techniques for picture enhancements in tomography and in simulations and surgical operations.
- ✓ It is also used in computed X-ray tomography(CT), position emission tomography(PET),and computed axial tomography(CAT).

i. Graphical User Interfaces

- ✓ It is common now for applications software to provide graphical user interface (GUI).
- ✓ A major component of graphical interface is a window manager that allows a user to display multiple, rectangular screen areas called display windows.

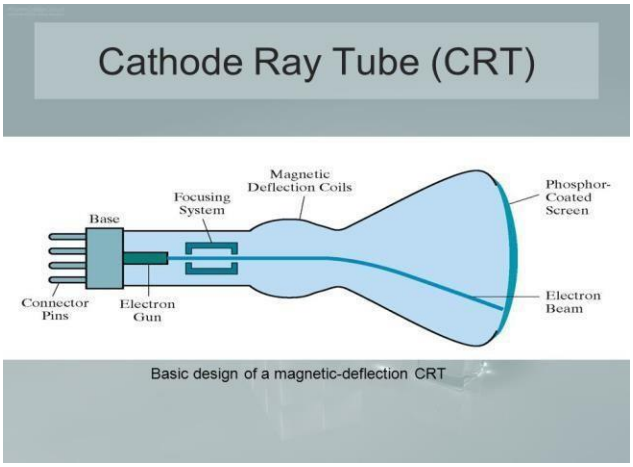


- ✓ Each screen display area can contain a different process, showing graphical or non-graphical information, and various methods can be used to activate a display window.
- ✓ Using an interactive pointing device, such as mouse, we can active a display window on some systems by positioning the screen cursor within the window display area and pressing the left mouse button.

Video Display Devices

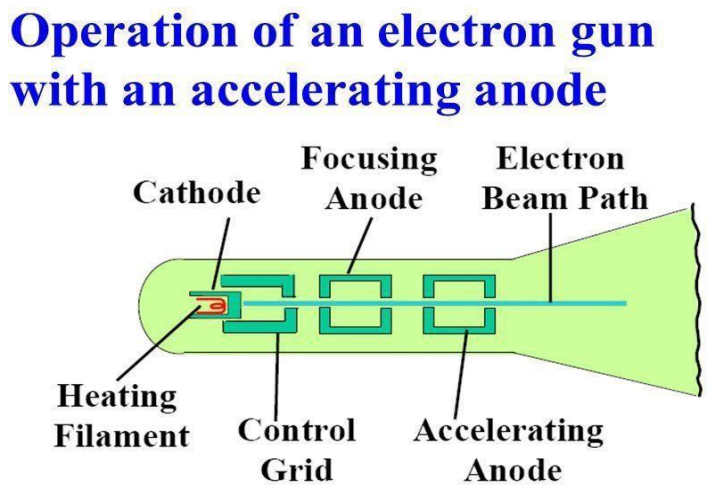
- ✓ The primary output device in a graphics system is a video monitor.
- ✓ Historically, the operation of most video monitors was based on the standard cathoderay tube (CRT) design, but several other technologies exist.
- ✓ In recent years, flat-panel displays have become significantly more popular due to their reduced power consumption and thinner designs.

Refresh Cathode-Ray Tubes



- ✓ A beam of electrons, 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 and the light emitted by the phosphor fades very rapidly.
- ✓ One way to maintain the screen picture is to store the picture information as a charge distribution within the CRT in order to keep the phosphors activated.
- ✓ The most common method now employed for maintaining phosphor glow is to redraw the picture repeatedly by quickly directing the electron beam back over the same screen points. This type of display is called a refresh CRT.
- ✓ The frequency at which a picture is redrawn on the screen is referred to as the refresh rate.

Operation of an electron gun with an accelarating anode



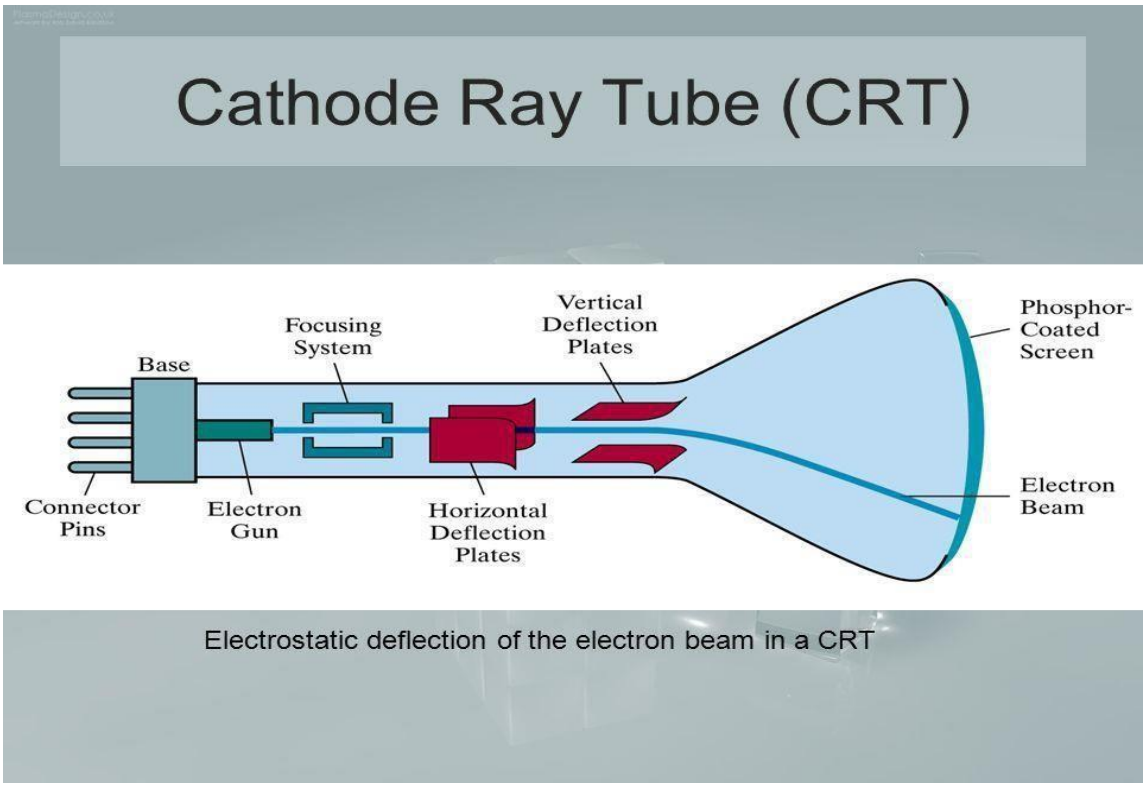
- ✓ The primary components of an electron gun in a CRT are the heated metal cathode and a control grid.
- ✓ The heat is supplied to the cathode by directing a current through a coil of wire, called the filament, inside the cylindrical cathode structure.
- ✓ This causes electrons to be "boiled off" the hot cathode surface.
- ✓ Inside the CRT envelope, the free, negatively charged electrons are then accelerated toward the phosphor coating by a high positive voltage.

- ✓ Intensity of the electron beam is controlled by the voltage at the control grid.
- ✓ Since the amount of light emitted by the phosphor coating depends on the number of electrons striking the screen, the brightness of a display point is controlled by varying the voltage on the control grid.
- ✓ The focusing system in a CRT forces the electron beam to converge to a small cross section as it strikes the phosphor and it is accomplished with either electric or magnetic fields.
- ✓ With electrostatic focusing, the electron beam is passed through a positively charged metal cylinder so that electrons along the center line of the cylinder are in equilibrium position.
- ✓ Deflection of the electron beam can be controlled with either electric or magnetic fields.
- ✓ Cathode-ray tubes are commonly constructed with two pairs of magnetic-deflection coils
- ✓ One pair is mounted on the top and bottom of the CRT neck, and the other pair is mounted on opposite sides of the neck.
- ✓ The magnetic field produced by each pair of coils results in a traverse deflection force that is perpendicular to both the direction of the magnetic field and the direction of travel of the electron beam.
- ✓ Horizontal and vertical deflections are accomplished with these pair of coils

Electrostatic deflection of the electron beam in a CRT

- ✓ When electrostatic deflection is used, two pairs of parallel plates are mounted inside the CRT envelope where, one pair of plates is mounted horizontally to control vertical deflection, and the other pair is mounted vertically to control horizontal deflection.
- ✓ Spots of light are produced on the screen by the transfer of the CRT beam energy to the phosphor.
- ✓ When the electrons in the beam collide with the phosphor coating, they are stopped and their kinetic energy is absorbed by the phosphor.
- ✓ Part of the beam energy is converted by the friction in to the heat energy, and the remainder causes electros in the phosphor atoms to move up to higher quantum-energy levels.

- ✓ After a short time, the "excited" phosphor electrons begin dropping back to their stable ground state, giving up their extra energy as small quantum of light energy called photons.

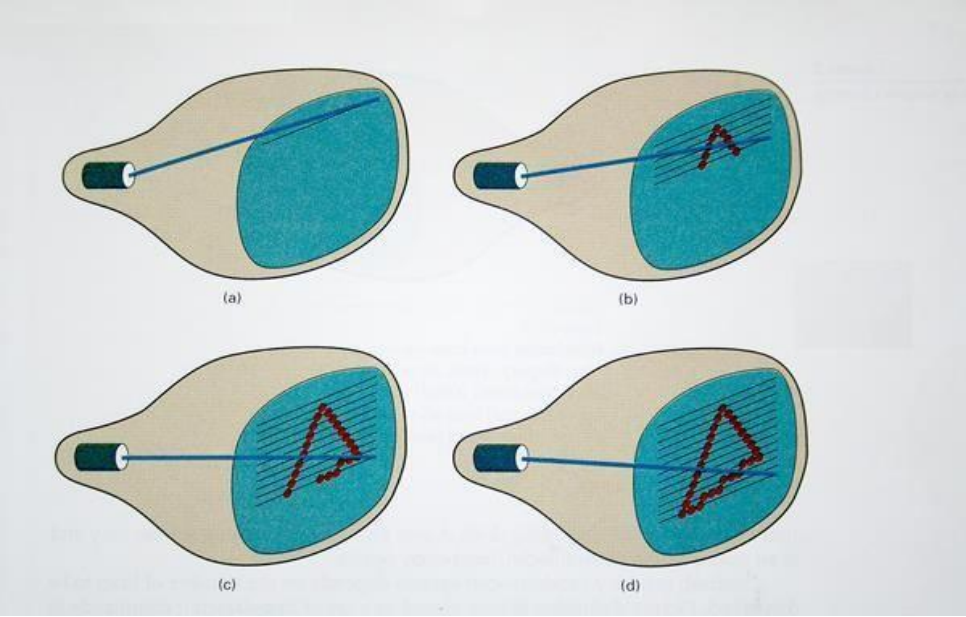


- ✓ What we see on the screen is the combined effect of all the electrons light emissions: a glowing spot that quickly fades after all the excited phosphor electrons have returned to their ground energy level.
- ✓ The frequency of the light emitted by the phosphor is proportional to the energy difference between the excited quantum state and the ground state.
- ✓ Lower persistence phosphors required higher refresh rates to maintain a picture on the screen without flicker.
- ✓ The maximum number of points that can be displayed without overlap on a CRT is referred to as a resolution.
- ✓ Resolution of a CRT is dependent on the type of phosphor, the intensity to be displayed, and the focusing and deflection systems.
- ✓ High-resolution systems are often referred to as high-definition systems.

Raster-Scan Displays and Random Scan Displays

i) Raster-Scan Displays

- The electron beam is swept across the screen one row at a time from top to bottom.
- As it moves across each row, the beam intensity is turned on and off to create a pattern of illuminated spots.
- This scanning process is called refreshing. Each complete scanning of a screen is normally called a frame.
- The refreshing rate, called the frame rate, is normally 60 to 80 frames per second, or described as 60 Hz to 80 Hz.
- Picture definition is stored in a memory area called the frame buffer.
- This frame buffer stores the intensity values for all the screen points. Each screen point is called a pixel (picture element).
- Property of raster scan is Aspect ratio, which defined as number of pixel columns divided by number of scan lines that can be displayed by the system.



Case 1: In case of black and white systems

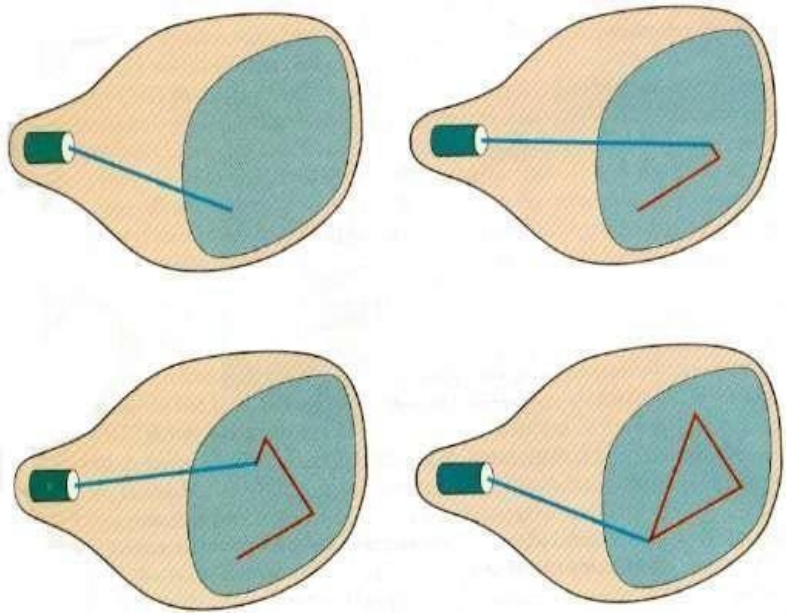
- ✓ On black and white systems, the frame buffer storing the values of the pixels is called a bitmap.
- ✓ Each entry in the bitmap is a 1-bit data which determine the on (1) and off (0) of the intensity of the pixel.

Case 2: In case of color systems

- ❖ On color systems, the frame buffer storing the values of the pixels is called a pixmap (Though now a days many graphics libraries name it as bitmap too).
- ❖ Each entry in the pixmap occupies a number of bits to represent the color of the pixel. For a true color display, the number of bits for each entry is 24 (8 bits per red/green/blue channel, each channel 28=256 levels of intensity value, ie. 256 voltage settings for each of the red/green/blue electron guns).

ii). Random-Scan Displays

- ✓ When operated as a random-scan display unit, a CRT has the electron beam directed only to those parts of the screen where a picture is to be displayed.
- ✓ Pictures are generated as line drawings, with the electron beam tracing out the component lines one after the other.
- ✓ For this reason, random-scan monitors are also referred to as vector displays (or strokewriting displays or calligraphic displays).
- ✓ The component lines 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 on a random-scan system depends on the number of lines to be displayed on that system.
- ✓ Picture definition is now stored as a set of line-drawing commands in an area of memory referred to as the display list, refresh display file, vector file, or display program
- ✓ To display a specified picture, the system cycles through the set of commands in the display file, drawing each component line in turn.
- ✓ After all line-drawing commands have been processed, the system cycles back to the first line command in the list.
- ✓ Random-scan displays are designed to draw all the component lines of a picture 30 to 60 times each second, with up to 100,000 "short" lines in the display list.
- ✓ When a small set of lines is to be displayed, each refresh cycle is delayed to avoid very high refresh rates, which could burn out the phosphor.

Difference between Raster scan system and Random scan system

Base of Difference	Raster Scan System	Random Scan System
Electron Beam	The electron beam is swept across the screen, one row at a time, from top to bottom	The electron beam is directed only to theparts of screen where a picture is to be drawn
Resolution	Its resolution is poor because raster system in contrast produces zigzag lines that are plotted as discrete point sets.	Its resolution is good because this system produces smooth lines drawings because CRT beam directly follows the line path.
Picture Definition	Picture definition is stored as a set of intensity values for all screen points,called pixels in a refresh buffer area.	Picture definition is stored as a set of line drawing instructions in a display file.
Realistic Display	The capability of this system to store intensity values for pixel makes it well suited for the realistic display of scenes	These systems are designed for line-drawing and can't display realistic shaded scenes.

	contain shadow and color pattern.	
Draw an Image	Screen points/pixels are used to draw an image	Mathematical functions are used to draw an image

Color CRT Monitors

- A CRT monitor displays color pictures by using a combination of phosphors that emit different-colored light.
- It produces range of colors by combining the light emitted by different phosphors.
- There are two basic techniques for color display:
    1. Beam-penetration technique
    2. Shadow-mask technique

1) Beam-penetration technique:

- ✓ This technique is used with random scan monitors.
- ✓ In this technique inside of CRT coated with two phosphor layers usually red and green.
- ✓ The outer layer of red and inner layer of green phosphor.
- ✓ The color depends on how far the electron beam penetrates into the phosphor layer.
- ✓ A beam of fast electron penetrates more and excites inner green layer while slow eletron excites outer red layer.
- ✓ At intermediate beam speed we can produce combination of red and green lights which emit additional two colors orange and yellow.
- ✓ The beam acceleration voltage controls the speed of the electrons and hence color of pixel.

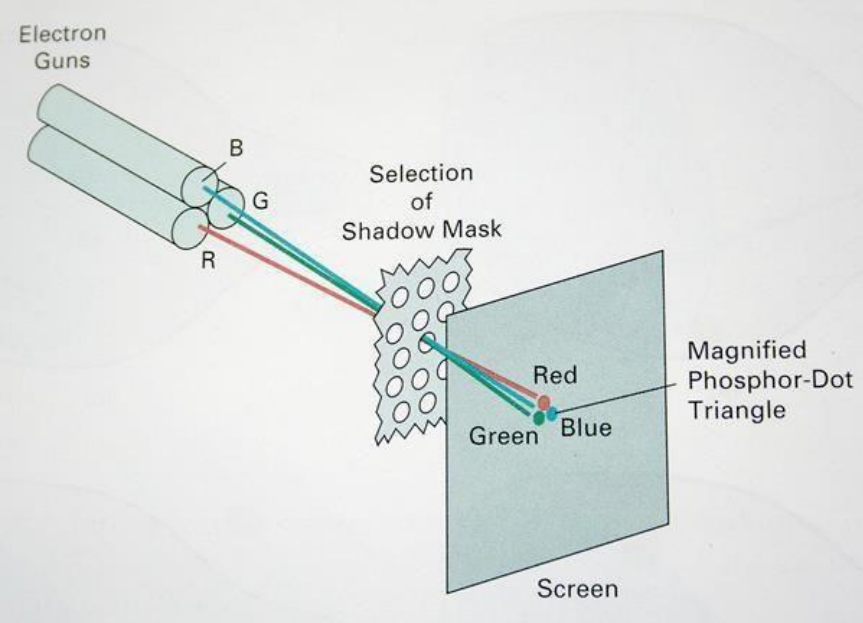
Disadvantages:

- ➢ It is a low cost technique to produce color in random scan monitors.
- ➢ It can display only four colors.
- ➢ Quality of picture is not good compared to other techniques.

2)Shadow-mask technique

- ✓ It produces wide range of colors as compared to beam-penetration technique.
- ✓ This technique is generally used in raster scan displays. Including color TV.

- ✓ In this technique CRT has three phosphor color dots at each pixel position.
- ✓ One dot for red, one for green and one for blue light. This is commonly known as Dot triangle.
- ✓ Here in CRT there are three electron guns present, one for each color dot. And a shadow mask grid just behind the phosphor coated screen.
- ✓ The shadow mask grid consists of series of holes aligned with the phosphor dot pattern.
- ✓ Three electron beams are deflected and focused as a group onto the shadow mask and when they pass through a hole they excite a dot triangle.
- ✓ In dot triangle three phosphor dots are arranged so that each electron beam can activate only its corresponding color dot when it passes through the shadow mask.
- ✓ A dot triangle when activated appears as a small dot on the screen which has color of combination of three small dots in the dot triangle.
- ✓ By changing the intensity of the three electron beams we can obtain different colors in the shadow mask CRT.



Flat Panel Display

- ➔ The term flat panel display refers to a class of video device that have reduced volume, weight & power requirement compared to a CRT.
- ➔ As flat panel display is thinner than CRTs, we can hang them on walls or wear on our wrists.

➔ Since we can even write on some flat panel displays they will soon be available as pocket notepads.

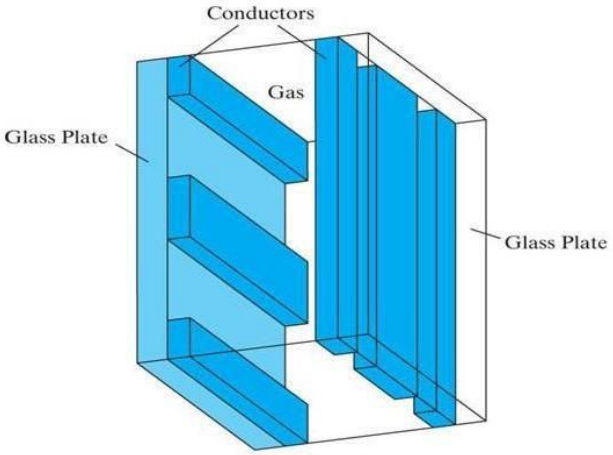
➔ We can separate flat panel display in two categories:

1. Emissive displays: - the emissive display or emitters are devices that convert electrical energy into light. For Ex. Plasma panel, thin film electroluminescent displays and light emitting diodes.

2. Non emissive displays: - non emissive display or non emitters use optical effects to convert sunlight or light from some other source into graphics patterns. For Ex. LCD (Liquid Crystal Display).

a) Plasma Panels displays

- This is also called gas discharge displays.
- It is 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 glass panel.

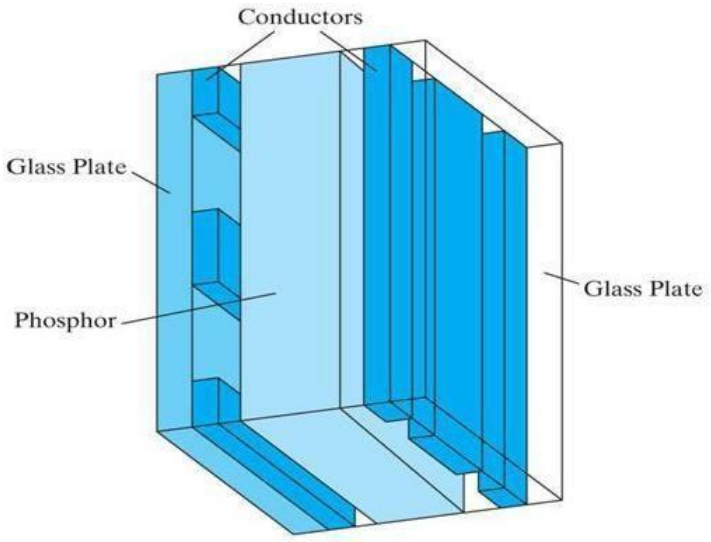


- Firing voltage is applied to a pair of horizontal and vertical conductors cause the gas at the intersection of the two conductors to break down into glowing plasma of electrons and ions.
- Picture definition is stored in a refresh buffer and the firing voltages are applied to refresh the pixel positions, 60 times per second.

- Alternating current methods are used to provide faster application of firing voltages and thus brighter displays.
- Separation between pixels is provided by the electric field of conductor.
- One disadvantage of plasma panels is they were strictly monochromatic device means shows only one color other than black like black and white.

b) Thin Film Electroluminescent Displays

- It is similar to plasma panel display but region between the glass plates is filled with phosphors such as doped with magnesium instead of gas.
- When sufficient voltage is applied the phosphors becomes a conductor in area of intersection of the two electrodes.
- Electrical energy is then absorbed by the manganese atoms which then release the energy as a spot of light similar to the glowing plasma effect in plasma panel.
- It requires more power than plasma panel.
- In this good color and gray scale difficult to achieve.

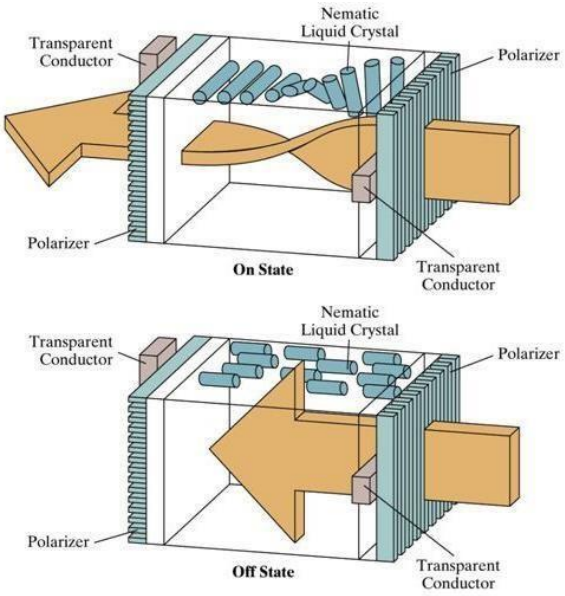


c. Light Emitting Diode (LED)

- In this display a matrix of multi-color light emitting diode is arranged to form the pixel position in the display and the picture definition is stored in refresh buffer.
- Similar to scan line refreshing of CRT information is read from the refresh buffer and converted to voltage levels that are applied to the diodes to produce the light pattern on the display.

d) Liquid Crystal Display (LCD)

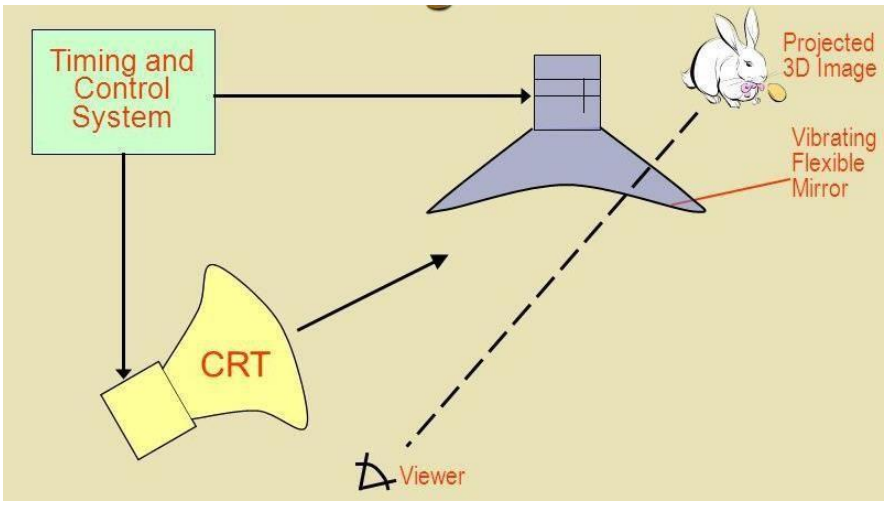
- This non emissive device produce picture by passing polarized light from the surroundings or from an internal light source through liquid crystal material that can be aligned to either block or transmit the light.
- The liquid crystal refreshes to fact that these compounds have crystalline arrangement of molecules then also flows like liquid.
- It consists of two glass plates each with light polarizer at right angles to each other sandwich the liquid crystal material between the plates.
- Rows of horizontal transparent conductors are built into one glass plate, and columns of vertical conductors are put into the other plates.
- The intersection of two conductors defines a pixel position.
- In the ON state polarized light passing through material is twisted so that it will pass through the opposite polarizer.
- In the OFF state it will reflect back towards source.



Three- Dimensional Viewing Devices

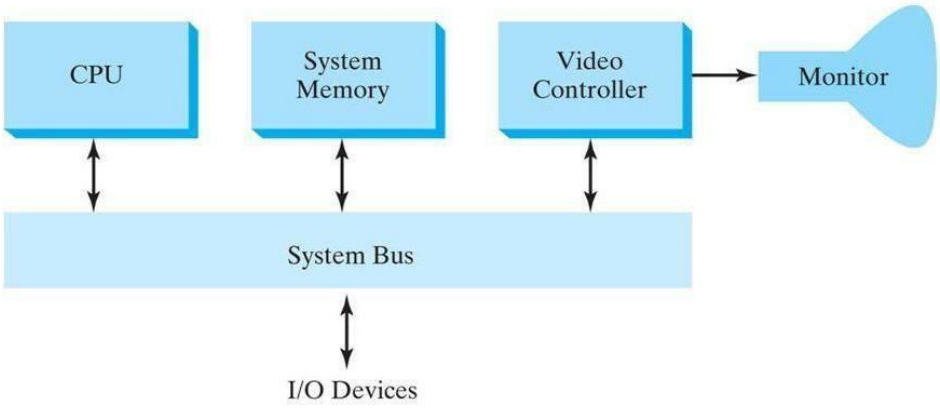
- Graphics monitors for the display of three-dimensional scenes have been devised using a technique that reflects a CRT image from a vibrating, flexible mirror As the varifocal mirror vibrates, it changes focallength.

- These vibrations are synchronized with the display of an object on a CRT so that each point on the object is reflected from the mirror into a spatial position corresponding to the distance of that point from a specified viewing location.
- This allows us to walk around an object or scene and view it from different sides.



Raster-Scan Systems

- Interactive raster-graphics systems typically employ several processing units.
- In addition to the central processing unit (CPU), a special-purpose processor, called the video controller or display controller, is used to control the operation of the display device.
- Organization of a simple raster system is shown in below Figure.

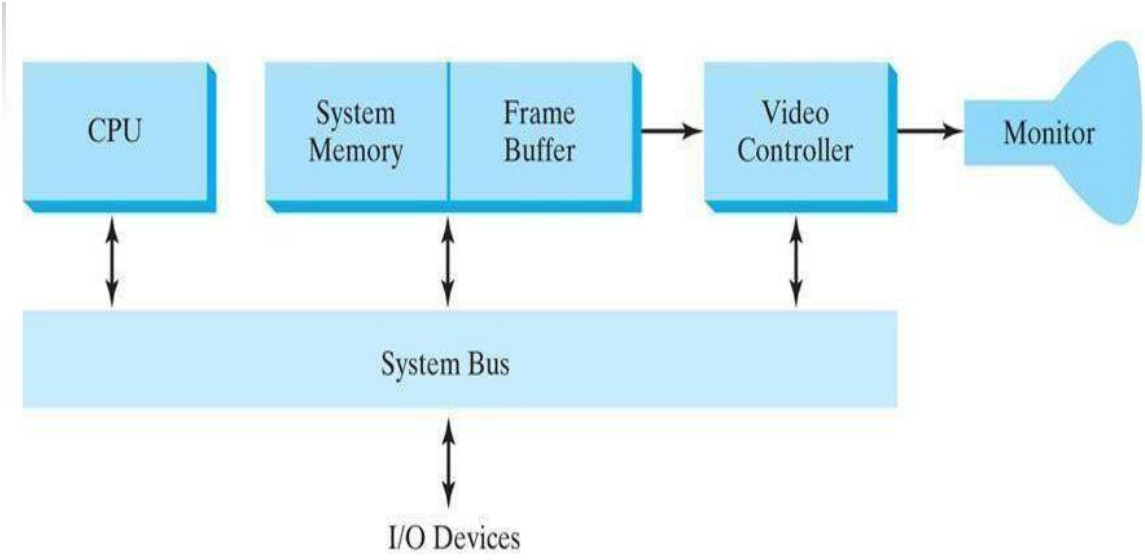


- Here, the frame buffer can be anywhere in the system memory, and the video controller accesses the frame buffer to refresh the screen.

➔ In addition to the video controller, raster systems employ other processors as coprocessors and accelerators to implement various graphics operations.

Video controller:

- ✓ The figure below shows a commonly used organization for raster systems.
- ✓ 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 referenced in the Cartesian coordinates.

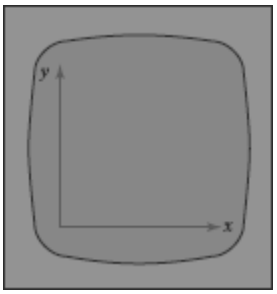


Cartesian reference frame:

- ✓ Frame-buffer locations and the corresponding screen positions, are referenced in Cartesian coordinates.
- ✓ In an application (user) program, we use the commands within a graphics software package to set coordinate positions for displayed objects relative to the origin of the
- ✓ The coordinate origin is referenced at the lower-left corner of a screen display area by the software commands, although we can typically set the origin at any convenient location for a particular application.

Working:

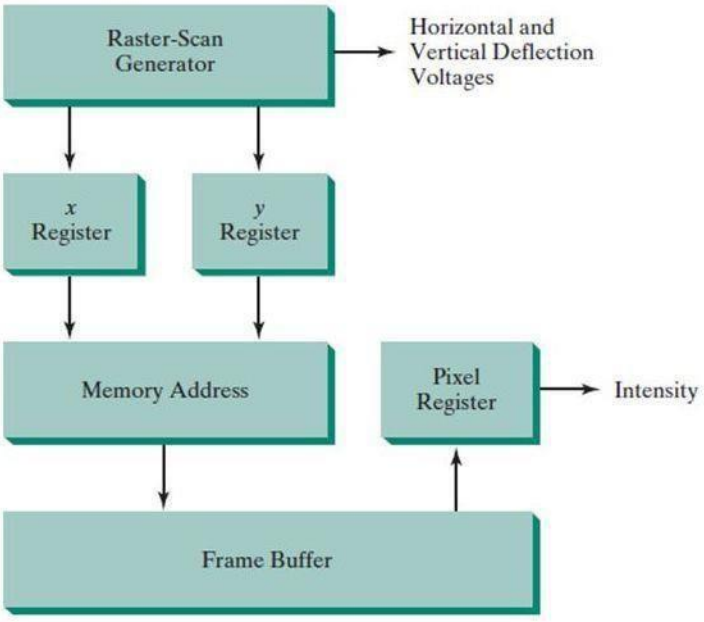
- ✓ Figure shows a two-dimensional Cartesian reference frame with the origin at the lowerleft screen corner.



- ✓ The screen surface is then represented as the first quadrant of a two-dimensional system with positive x and y values increasing from left to right and bottom of the screen to the top respectively.
- ✓ Pixel positions are then assigned integer x values that range from 0 to xmax across the screen, left to right, and integer y values that vary from 0 to ymax, bottom to top.

Basic Video Controller Refresh Operations

- ✓ The basic refresh operations of the video controller are diagrammed



- ✓ Two registers are used to store the coordinate values for the screen pixels.

- ✓ Initially, the x register is set to 0 and the y register is set to the value for the top scan line.
- ✓ The contents of the frame buffer at this pixel position are then retrieved and used to set the intensity of the CRT beam.
- ✓ Then the x register is incremented by 1, and the process is repeated for the next pixel on the top scan line.
- ✓ This procedure continues for each pixel along the top scan line.
- ✓ After the last pixel on the top scan line has been processed, the x register is reset to 0 and the y register is set to the value for the next scan line down from the top of the screen.
- ✓ The procedure is repeated for each successive scan line.
- ✓ After cycling through all pixels along the bottom scan line, the video controller resets the registers to the first pixel position on the top scan line and the refresh process starts over

a.Speed up pixel position processing of video controller:

- ✓ Since the screen must be refreshed at a rate of at least 60 frames per second,the simple procedure illustrated in above figure may not be accommodated by RAM chips if the cycle time is too slow.
- ✓ To speed up pixel processing, video controllers can retrieve multiple pixel values from the refresh buffer on each pass.
- ✓ When group of pixels has been processed, the next block of pixel values is retrieved from the frame buffer.

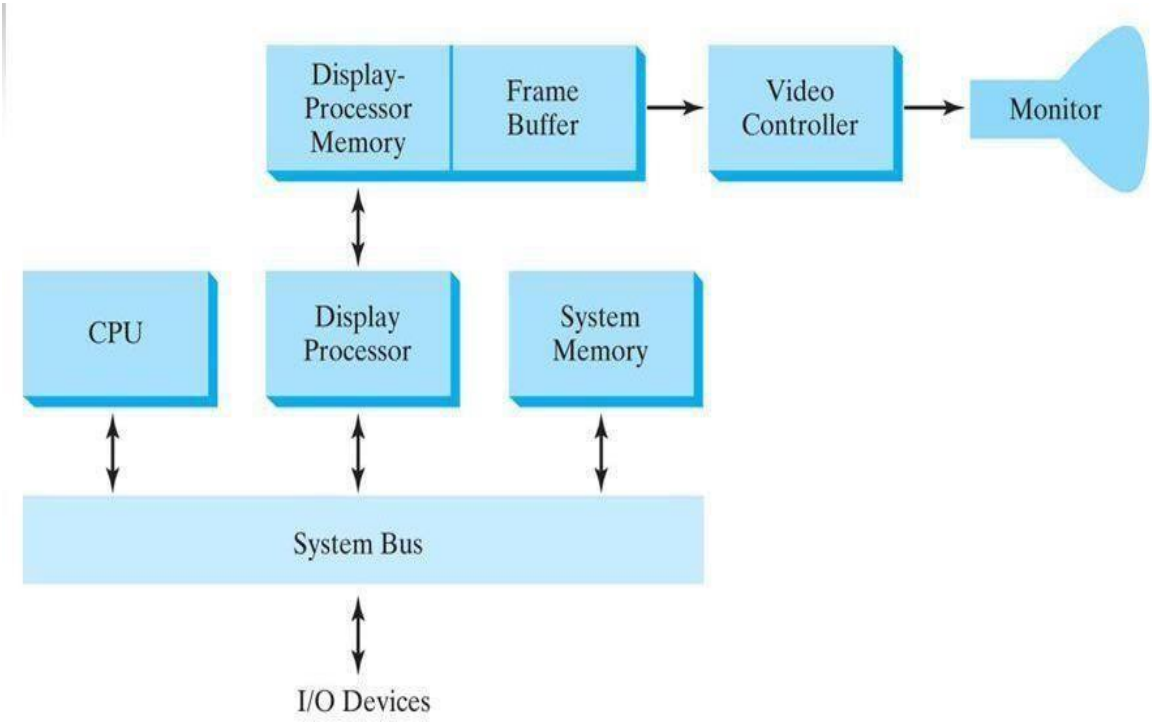
Advantages of video controller:

- ✓ A video controller can be designed to perform a number of other operations.
- ✓ For various applications, the video controller can retrieve pixel values from different memory areas on different refresh cycles.
- ✓ This provides a fast mechanism for generating real-time animations.
- ✓ Another video-controller task is the transformation of blocks of pixels, so that screen areas can be enlarged, reduced, or moved from one location to another during the refresh cycles.
- ✓ In addition, the video controller often contains a lookup table, so that pixel values in the frame buffer are used to access the lookup table. This provides a fast method for changing screen intensity values.

- ✓ Finally, some systems are designed to allow the video controller to mix the framebuffer image with an input image from a television camera or other input device

b) Raster-Scan Display Processor

- ✓ Figure shows one way to organize the components of a raster system that contains a separate display processor, sometimes referred to 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 be provided.

Scan conversion:

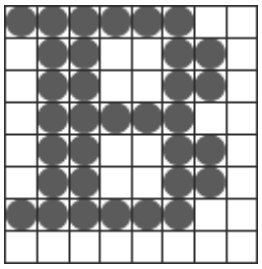
- ✓ A major task of the display processor is digitizing a picture definition given in an application program into a set of pixel values for storage in the frame buffer.
- ✓ This digitization process is called scan conversion.

Example 1: displaying a line

- ➔ Graphics commands specifying straight lines and other geometric objects are scan converted into a set of discrete points, corresponding to screen pixel positions.
- ➔ Scan converting a straight-line segment.

Example 2: displaying a character

- ➔ Characters can be defined with rectangular pixel grids
- ➔ The array size for character grids can vary from about 5 by 7 to 9 by 12 or more for higher-quality displays.
- ➔ A character grid is displayed by superimposing the rectangular grid pattern into the frame buffer at a specified coordinate position.



Using outline:

- ➔ For characters that are defined as outlines, the shapes are scan-converted into the frame buffer by locating the pixel positions closest to the outline.

Additional operations of Display processors:

- ➔ 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 applying transformations to the objects in a scene.
- ➔ Display processors are typically designed to interface with interactive input devices, such as a mouse.

Methods to reduce memory requirements in display processor:

- ➔ In an effort to reduce memory requirements in raster systems, methods have been devised for organizing the frame buffer as a linked list and encoding the color information.
- ➔ One organization scheme is to store each scan line as a set of number pairs.

➔ Encoding methods can be useful in the digital storage and transmission of picture information

i) Run-length encoding:

- The first number in each pair can be a reference to a color value, and the second number can specify the number of adjacent pixels on the scan line that are to be displayed in that color.
- This technique, called run-length encoding, can result in a considerable saving in storage space if a picture is to be constructed mostly with long runs of a single color each.
- A similar approach can be taken when pixel colors change linearly.

ii) Cell encoding:

- Another approach is to encode the raster as a set of rectangular areas (cell encoding).

Disadvantages of encoding:

- ❖ The disadvantages of encoding runs are that color changes are difficult to record and storage requirements increase as the lengths of the runs decrease.
- ❖ In addition, it is difficult for the display controller to process the raster when many short runs are involved.
- ❖ Moreover, the size of the frame buffer is no longer a major concern, because of sharp declines in memory costs

Graphics workstations and viewing systems

- ✓ Most graphics monitors today operate as raster-scan displays, and both CRT and flat panel systems are in common use.
- ✓ Graphics workstation range from small general-purpose computer systems to multi monitor facilities, often with ultra –large viewing screens.
- ✓ High-definition graphics systems, with resolutions up to 2560 by 2048, are commonly used in medical imaging, air-traffic control, simulation, and CAD.
- ✓ Many high-end graphics workstations also include large viewing screens, often with specialized features.

- ✓ Multi-panel display screens are used in a variety of applications that require "wall-sized" viewing areas. These systems are designed for presenting graphics displays at meetings, conferences, conventions, trade shows, retail stores etc.
- ✓ A multi-panel display can be used to show a large view of a single scene or several individual images. Each panel in the system displays one section of the overall picture
- ✓ A large, curved-screen system can be useful for viewing by a group of people studying a particular graphics application.
- ✓ A 360 degree paneled viewing system in the NASA control-tower simulator, which is used for training and for testing ways to solve air-traffic and runway problems at airports.

Input Devices

- ➢ Graphics workstations make use of various devices for data input.Most systems have keyboards and mouses,while some other systems have trackball,spaceball,joystick,button boxes,touch panels,image scanners and voice systems.

Keyboard:

- ➢ Keyboard on graphics system is used for entering text strings,issuing certain commands and selecting menu options.
- ➢ Keyboards can also be provided with features for entry of screen coordinates,menu selections or graphics functions.
- ➢ General purpose keyboard uses function keys and cursor-control keys.
- ➢ Function keys allow user to select frequently accessed operations with a single keystroke.Cursor-control keys are used for selecting a displayed object or a location by positioning the screen cursor.

Button Boxes and Dials:

- ➢ Buttons are often used to input predefined functions .Dials are common devices for entering scalar values.
- ➢ Numerical values within some defined range are selected for input with dial rotations.

Mouse Devices:

- Mouse is a hand-held device,usually moved around on a flat surface to position the screen cursor.wheeler or roolers on the bottom of the mouse used to record the amount and direction of movement.
- Some of the mouses uses optical sensors,which detects movement across the horizontal and vertical grid lines.
- Since a mouse can be picked up and put down,it is used for making relative changes in the position of the screen.
- Most general purpose graphics systems now include a mouse and a keyboard as the primary input devices.

Trackballs and Spaceballs:

- A trackball is a ball device that can be rotated with the fingers or palm of the hand to produce screen cursor movement.
- Laptop keyboards are equipped with a trackball to eliminate the extra space required by a mouse.
- Spaceball is an extension of two-dimensional trackball concept.
- Spaceballs are used for three-dimensional positioning and selection operations in virtual-reality systems,modeling,animation,CAD and other applications.

Joysticks:

- Joystick is used as a positioning device,which uses a small vertical lever(stick) mounded on a base.It is used to steer the screen cursor around and select screen position with the stick movement.
- A push or pull on the stick is measured with strain gauges and converted to movement of the screen cursor in the direction of the applied pressure.

Data Gloves:

- Data glove can be used to grasp a virtual object.The glove is constructed with a series of sensors that detect hand and finger motions.
- Input from the glove is used to position or manipulate objects in a virtual scene.

Digitizers:

- Digitizer is a common device for drawing,painting or selecting positions.
- Graphics tablet is one type of digitizer,which is used to input 2-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 and stylus is a pencil-shaped device that is pointed at positions on the tablet.

Image Scanners:

- Drawings,graphs,photographs or text can be stored for computer processing with an image scanner by passing an optical scanning mechanism over the information to be stored.
- Once we have the representation of the picture, then we can apply various image-processing method to modify the representation of the picture and various editing operations can be performed on the stored documents.

Touch Panels:

- Touch panels allow displayed objects or screen positions to be selected with the touch of a finger.
- Touch panel is used for the selection of processing options that are represented as a menu of graphical icons.
- Optical touch panel-uses LEDs along one vertical and horizontal edge of the frame.
- Acoustical touch panels generates high-frequency sound waves in horizontal and vertical directions across a glass plate.

Light Pens:

- Light pens are pencil-shaped devices used to select positions by detecting the light coming from points on the CRT screen.
- To select positions in any screen area with a light pen,we must have some nonzero light intensity emitted from each pixel within that area.
- Light pens sometimes give false readings due to background lighting in a room.

Voice Systems:

- Speech recognizers are used with some graphics workstations as input devices for voice commands.The voice system input can be used to initiate operations or to enter data.
- A dictionary is set up by speaking command words several times,then the system analyses each word and matches with the voice command to match the pattern

Graphics Networks

➔ So far, we have mainly considered graphics applications on an isolated system with a single user.

➔ Multiuser environments & computer networks are now common elements in many graphics applications.

➔ Various resources, such as processors, printers, plotters and data files can be distributed on a network & shared by multiple users.

➔ A graphics monitor on a network is generally referred to as a graphics server.

➔ The computer on a network that is executing a graphics application is called the client.

➔ A workstation that includes processors, as well as a monitor and input devices can function as both a server and a client.

Graphics on Internet

✓ A great deal of graphics development is now done on the Internet.

✓ Computers on the Internet communicate using TCP/IP.

✓ Resources such as graphics files are identified by URL (Uniform resource locator).

✓ The World Wide Web provides a hypertext system that allows users to loacate and view documents, audio and graphics.

✓ Each URL sometimes also called as universal resource locator.

✓ The URL contains two parts Protocol- for transferring the document, and Server-contains the document.

Graphics Software

✓ There are two broad classifications for computer-graphics software

1. Special-purpose packages: Special-purpose packages are designed for nonprogrammers

   Example: generate pictures, graphs, charts, painting programs or CAD systems in some application area without worrying about the graphics procedure
2. General programming packages: general programming package provides a library of graphics functions that can be used in a programming language such as C, C++, Java, or FORTRAN.

   Example: GL (Graphics Library), OpenGL, VRML (Virtual-Reality Modeling Language), Java 2D And Java 3D

NOTE: A set of graphics functions is often called a computer-graphics application programming interface (CG API)

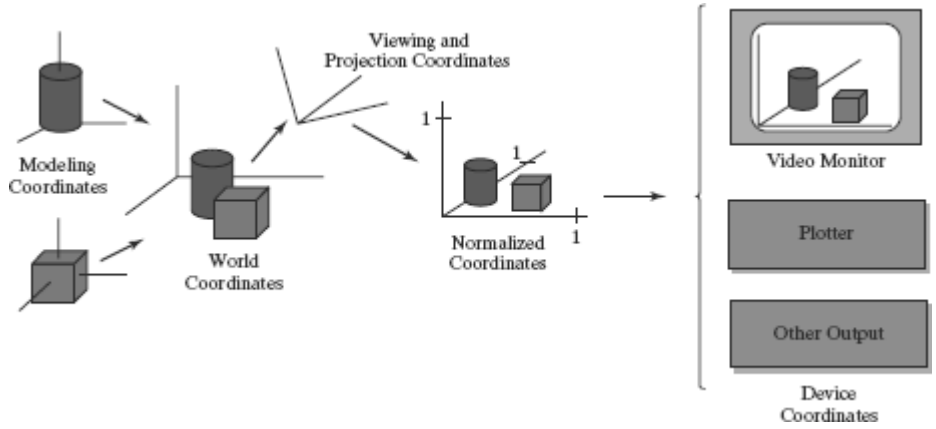
Coordinate Representations

- ✓ To generate a picture using a programming package we first need to give the geometric descriptions of the objects that are to be displayed known as coordinates.
- ✓ If coordinate values for a picture are given in some other reference frame (spherical, hyperbolic, etc.), they must be converted to Cartesian coordinates.
- ✓ Several different Cartesian reference frames are used in the process of constructing and displaying
- ✓ First we define the shapes of individual objects, such as trees or furniture, These reference frames are called modeling coordinates or local coordinates
- ✓ Then we place the objects into appropriate locations within a scene reference frame called world coordinates.
- ✓ After all parts of a scene have been specified, it is processed through various output-device reference frames for display. This process is called the viewing pipeline.
- ✓ The scene is then stored in normalized coordinates. Which range from −1 to 1 or from 0 to 1 Normalized coordinates are also referred to as normalized device coordinates.
- ✓ The coordinate systems for display devices are generally called device coordinates, or screen coordinates.

NOTE: Geometric descriptions in modeling coordinates and world coordinates can be given in

floating-point or integer values.

- ✓ Example: Figure briefly illustrates the sequence of coordinate transformations from modeling coordinates to device coordinates for a display



$$(x_{mc}, y_{mc}, z_{mc}) \rightarrow (x_{wc}, y_{wc}, z_{wc}) \rightarrow (x_{vc}, y_{vc}, z_{vc}) \rightarrow (x_{pc}, y_{pc}, z_{pc})$$
$$\rightarrow (x_{nc}, y_{nc}, z_{nc}) \rightarrow (x_{dc}, y_{dc})$$

Graphics Functions

- ➔ It provides users with a variety of functions for creating and manipulatingpictures
- ➔ The basic building blocks for pictures are referred to as graphics output primitives
- ➔ Attributes are properties of the output primitives
- ➔ We can change the size, position, or orientation of an object using geometric transformations
- ➔ Modeling transformations, which are used to construct a scene.
- ➔ Viewing transformations are used to select a view of the scene, the type of projection to be used and the location where the view is to be displayed.
- ➔ Input functions are used to control and process the data flow from these interactive devices(mouse, tablet and joystick)
- ➔ Graphics package contains a number of tasks .We can lump the functions for carrying out many tasks by under the heading control operations.

Software Standards

- ✓ The primary goal of standardized graphics software is portability.

- ✓ In 1984, Graphical Kernel System (GKS) was adopted as the first graphics software standard by the International Standards Organization (ISO)
- ✓ The second software standard to be developed and approved by the standards organizations was Programmer's Hierarchical Interactive Graphics System (PHIGS).
- ✓ Extension of PHIGS, called PHIGS+, was developed to provide 3-D surface rendering capabilities not available in PHIGS.
- ✓ The graphics workstations from Silicon Graphics, Inc. (SGI), came with a set of routines called GL (Graphics Library)

Other Graphics Packages

- ✓ Many other computer-graphics programming libraries have been developed for

1. general graphics routines
2. Some are aimed at specific applications (animation, virtual reality, etc.)
   Example: Open Inventor Virtual-Reality Modeling Language (VRML).

We can create 2-D scenes with in Java applets (java2D, Java 3D)

Introduction To OpenGL

- ✓ OpenGL basic(core) library :-A basic library of functions is provided in OpenGL for specifying graphics primitives, attributes, geometric transformations, viewing transformations, and many other operations.

Basic OpenGL Syntax

- ➔ Function names in the OpenGL basic library (also called the OpenGL core library) are prefixed with gl. The component word first letter is capitalized.
- ➔ For eg:- glBegin, glClear, glCopyPixels, glPolygonMode
- ➔ Symbolic constants that are used with certain functions as parameters are all in capital letters, preceded by "GL", and component are separated by underscore.
- ➔ For eg:- GL_2D, GL_RGB, GL_CCW, GL_POLYGON, GL_AMBIENT_AND_DIFFUSE.

➔ The OpenGL functions also expect specific data types. For example, an OpenGL function parameter might expect a value that is specified as a 32-bit integer. But the size of an integer specification can be different on different machines.

➔ To indicate a specific data type, OpenGL uses special built-in, data-type names, such as GLbyte, GLshort, GLint, GLfloat, GLdouble, Glboolean

Related Libraries

➔ In addition to OpenGL basic(core) library(prefixed with gl), there are a number of associated libraries for handling special operations:-

1) OpenGL Utility(GLU):- Prefixed with "glu". It provides routines for setting up viewing and projection matrices, describing complex objects with line and polygon approximations, displaying quadrics and B-splines using linear approximations, processing the surface-rendering operations, and other complex tasks.

-Every OpenGL implementation includes the GLU library

2) Open Inventor:- provides routines and predefined object shapes for interactive three-dimensional applications which are written in C++.

3) Window-system libraries:- To create graphics we need display window. We cannot create the display window directly with the basic OpenGL functions since it contains only device-independent graphics functions, and window-management operations are device-dependent. However, there are several window-system libraries that supports OpenGL functions for a variety of machines.

Eg:- Apple GL(AGL), Windows-to-OpenGL(WGL), Presentation Manager to OpenGL(PGL), GLX.

4) OpenGL Utility Toolkit(GLUT):- provides a library of functions which acts as interface for interacting with any device specific screen-windowing system, thus making our program device-independent. The GLUT library functions are prefixed with "glut".

Header Files

✓ In all graphics programs, we will need to include the header file for the OpenGL core library.

- ✓ In windows to include OpenGL core libraries and GLU we can use the following header files:-

  #include <windows.h> //precedes other header files for including Microsoft windows ver of OpenGL libraries

  #include<GL/gl.h>

  #include <GL/glu.h>

- ✓ The above lines can be replaced by using GLUT header file which ensures gl.h and glu.h are included correctly,
- ✓ #include <GL/glut.h> //GL in windows
- ✓ In Apple OS X systems, the header file inclusion statement will be,
- ✓ #include <GLUT/glut.h>

Display-Window Management Using GLUT

- ✓ We can consider a simplified example, minimal number of operations for displaying a picture.

Step 1: initialization of GLUT

- We are using the OpenGL Utility Toolkit, our first step is to initialize GLUT.
- This initialization function could also process any command line arguments, but we will not need to use these parameters for our first example programs.
- We perform the GLUT initialization with the statement

  glutInit (&argc, argv);

Step 2: title

- We can state that a display window is to be created on the screen with a given caption for the title bar. This is accomplished with the function

  glutCreateWindow ("An Example OpenGL Program");

- where the single argument for this function can be any character string that we want to use for the display-window title.

Step 3: Specification of the display window

- Then we need to specify what the display window is to contain.
- For this, we create a picture using OpenGL functions and pass the picture definition to the GLUT routine glutDisplayFunc, which assigns our picture to the display window.

- Example: suppose we have the OpenGL code for describing a line segment in a procedure called lineSegment.
- Then the following function call passes the line-segment description to the display window:

  glutDisplayFunc (lineSegment);

Step 4: one more GLUT function

- But the display window is not yet on the screen.
- We need one more GLUT function to complete the window-processing operations.
- After execution of the following statement, all display windows that we have created, including their graphic content, are now activated:

  glutMainLoop ( );

- This function must be the last one in our program. It displays the initial graphics and puts the program into an infinite loop that checks for input from devices such as a mouse or keyboard.

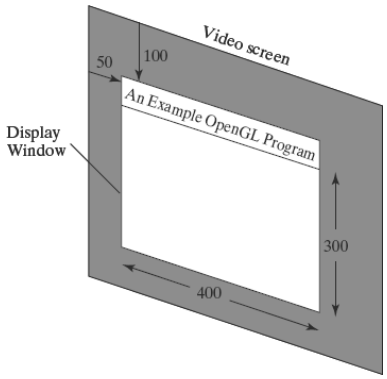
Step 5: these parameters using additional GLUT functions

- Although the display window that we created will be in some default location and size, we can set these parameters using additional GLUT functions.

GLUT Function 1:

➔ We use the glutInitWindowPosition function to give an initial location for the upper left corner of the display window.

➔ This position is specified in integer screen coordinates, whose origin is at the upper-left corner of the screen.



GLUT Function 2:

After the display window is on the screen, we can reposition and resize it.

GLUT Function 3:

- ➔ We can also set a number of other options for the display window, such as buffering and a choice of color modes, with the glutInitDisplayMode function.
- ➔ Arguments for this routine are assigned symbolic GLUT constants.
- ➔ Example: the following command specifies that a single refresh buffer is to be used for the display window and that we want to use the color mode which uses red, green, and blue (RGB) components to select color values:

  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
- ➔ The values of the constants passed to this function are combined using a logical or operation.
- ➔ Actually, single buffering and RGB color mode are the default options.
- ➔ But we will use the function now as a reminder that these are the options that are set for our display.
- ➔ Later, we discuss color modes in more detail, as well as other display options, such as double buffering for animation applications and selecting parameters for viewing threedimensional scenes.

A Complete OpenGL Program

- ➔ There are still a few more tasks to perform before we have all the parts that we need for a complete program.

Step 1: to set background color

- ➔ For the display window, we can choose a background color.
- ➔ Using RGB color values, we set the background color for the display window to be white, with the OpenGL function:

  glClearColor (1.0, 1.0, 1.0, 0.0);
- ➔ The first three arguments in this function set the red, green, and blue component colors to the value 1.0, giving us a white background color for the display window.
- ➔ If, instead of 1.0, we set each of the component colors to 0.0, we would get a black background.

- ➔ The fourth parameter in the glClearColor function is called the alpha value for the specified color. One use for the alpha value is as a "blending" parameter
- ➔ When we activate the OpenGL blending operations, alpha values can be used to determine the resulting color for two overlapping objects.
- ➔ An alpha value of 0.0 indicates a totally transparent object, and an alpha value of 1.0 indicates an opaque object.
- ➔ For now, we will simply set alpha to 0.0.
- ➔ Although the glClearColor command assigns a color to the display window, it does not put the display window on the screen.

Step 2: to set window color

- ➔ To get the assigned window color displayed, we need to invoke the following OpenGL function:

  glClear (GL_COLOR_BUFFER_BIT);

- ➔ The argument GL COLOR BUFFER BIT is an OpenGL symbolic constant specifying that it is the bit values in the color buffer (refresh buffer) that are to be set to the values indicated in the glClearColor function. (OpenGL has several different kinds of buffers that can be manipulated.

Step 3: to set color to object

- ➔ In addition to setting the background color for the display window, we can choose a variety of color schemes for the objects we want to display in a scene.
- ➔ For our initial programming example, we will simply set the object color to be a dark green

  glColor3f (0.0, 0.4, 0.2);

- ➔ The suffix 3f on the glColor function indicates that we are specifying the three RGB color components using floating-point (f) values.
- ➔ This function requires that the values be in the range from 0.0 to 1.0, and we have set red = 0.0, green = 0.4, and blue = 0.2.

Example program

- ➔ For our first program, we simply display a two-dimensional line segment.
- ➔ To do this, we need to tell OpenGL how we want to "project" our picture onto the display window because generating a two-dimensional picture is treated by OpenGL as a special case of three-dimensional viewing.
- ➔ So, although we only want to produce a very simple two-dimensional line, OpenGL processes our picture through the full three-dimensional viewing operations.
- ➔ We can set the projection type (mode) and other viewing parameters that we need with the following two functions:

```
glMatrixMode (GL_PROJECTION);
gluOrtho2D (0.0, 200.0, 0.0, 150.0);
```

- ➔ This specifies that an orthogonal projection is to be used to map the contents of a twodimensional rectangular area of world coordinates to the screen, and that the x-coordinate values within this rectangle range from 0.0 to 200.0 with y-coordinate values ranging from 0.0 to 150.0.
- ➔ Whatever objects we define within this world-coordinate rectangle will be shown within the display window.
- ➔ Anything outside this coordinate range will not be displayed.
- ➔ Therefore, the GLU function gluOrtho2D defines the coordinate reference frame within the display window to be (0.0, 0.0) at the lower-left corner of the display window and (200.0, 150.0) at the upper-right window corner.
- ➔ For now, we will use a world-coordinate rectangle with the same aspect ratio as the display window, so that there is no distortion of our picture.
- ➔ Finally, we need to call the appropriate OpenGL routines to create our line segment.
- ➔ The following code defines a two-dimensional, straight-line segment with integer,
- ➔ Cartesian endpoint coordinates (180, 15) and (10, 145).

```
glBegin (GL_LINES);
glVertex2i (180, 15);
glVertex2i (10, 145);
glEnd ( );
```

- ➔ Now we are ready to put all the pieces together:

The following OpenGL program is organized into three functions.

- ➔ init: We place all initializations and related one-time parameter settings in function init.
- ➔ lineSegment: Our geometric description of the "picture" that we want to display is in function lineSegment, which is the function that will be referenced by the GLUT function glutDisplayFunc.
- ➔ main function main function contains the GLUT functions for setting up the display window and getting our line segment onto the screen.
- ➔ glFlush: This is simply a routine to force execution of our OpenGL functions, which are stored by computer systems in buffers in different locations,depending on how OpenGL is implemented.
- ➔ The procedure lineSegment that we set up to describe our picture is referred to as a display callback function.
- ➔ And this procedure is described as being "registered" by glutDisplayFunc as the routine to invoke whenever the display window might need to be redisplayed.

Example: if the display window is moved.

Following program to display window and line segment generated by this program:

```
#include <GL/glut.h> // (or others, depending on the system in use)
void init (void)
{
        glClearColor (1.0, 1.0, 1.0, 0.0);          // Set display-window color to white.
        glMatrixMode (GL_PROJECTION);    // Set projection parameters.
        gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}
void lineSegment (void)
{
        glClear (GL_COLOR_BUFFER_BIT);      // Clear display window.
        glColor3f (0.0, 0.4, 0.2);                  // Set line segment color to green.
        glBegin (GL_LINES);
        glVertex2i (180, 15);                      // Specify line-segment geometry.
        glVertex2i (10, 145);
        glEnd ( );
```

```
    glFlush ( ); // Process all OpenGL routines as quickly as possible.
}
void main (int argc, char** argv)
{
    glutInit (&argc, argv);                                  // Initialize GLUT.
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB); // Set display mode.
    glutInitWindowPosition (50, 100);     // Set top-left display-window position.
    glutInitWindowSize (400, 300);       // Set display-window width and height.
    glutCreateWindow ("An Example OpenGL Program"); // Create display window.
    init ( );                                  // Execute initialization procedure.
    glutDisplayFunc (lineSegment);       // Send graphics to display window.
    glutMainLoop ( );                    // Display everything and wait.
}
```

Coordinate Reference Frames

To describe a picture, we first decide upon

- A convenient Cartesian coordinate system, called the world-coordinate reference frame, which could be either 2D or 3D.
  - We then describe the objects in our picture by giving their geometric specifications in terms of positions in world coordinates.
  - Example: We define a straight-line segment with two endpoint positions, and a polygon is specified with a set of positions for its vertices.
  - These coordinate positions are stored in the scene description along with other information about the objects, such as their color and their coordinate extents
  - Co-ordinate extents: Co-ordinate extents are the minimum and maximum x, y, and z values for each object.
  - A set of coordinate extents is also described as a bounding box for an object.
- Ex: For a 2D figure, the coordinate extents are sometimes called its bounding rectangle.
  - Objects are then displayed by passing the scene description to the viewing routines which identify visible surfaces and map the objects to the frame buffer positions and then on the video monitor.

➢ The scan-conversion algorithm stores info about the scene, such as color values, in appropriate locations in the frame buffer, and then the scene is displayed on the output device.

Screen co-ordinates:

- ✓ Locations on a video monitor are referenced in integer screen coordinates, which correspond to the integer pixel positions in the frame buffer.
- ✓ Scan-line algorithms for the graphics primitives use the coordinate descriptions to determine the locations of pixels
- ✓ Example: given the endpoint coordinates for a line segment, a display algorithm must calculate the positions for those pixels that lie along the line path between the endpoints.
- ✓ Since a pixel position occupies a finite area of the screen, the finite size of a pixel must be taken into account by the implementation algorithms.
- ✓ For the present, we assume that each integer screen position references the centre of a pixel area.
- ✓ Once pixel positions have been identified the color values must be stored in the frame buffer

Assume we have available a low-level procedure of the form

i) setPixel (x, y);

- stores the current color setting into the frame buffer at integer position(x, y), relative to the position of the screen-coordinate origin

ii) getPixel (x, y, color);

- Retrieves the current frame-buffer setting for a pixel location;
- Parameter color receives an integer value corresponding to the combined RGB bit codes stored for the specified pixel at position (x,y).
- Additional screen-coordinate information is needed for 3D scenes.
- For a two-dimensional scene, all depth values are 0.

Absolute and Relative Coordinate Specifications

Absolute coordinate:

- So far, the coordinate references that we have discussed are stated as absolute coordinate values.
- This means that the values specified are the actual positions within the coordinate system in use.

Relative coordinates:

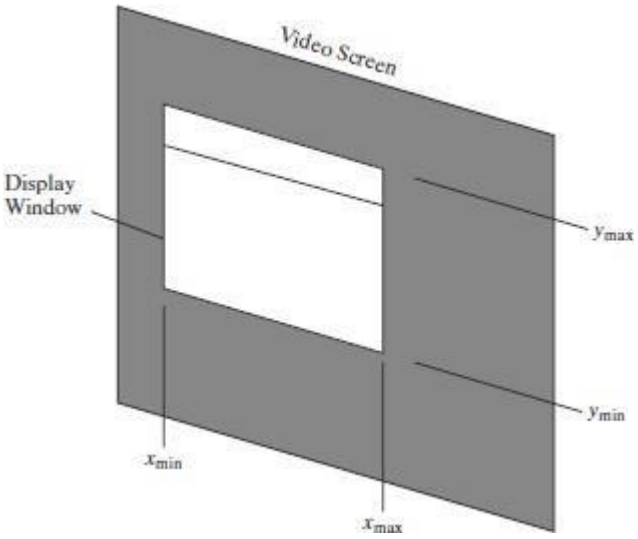
- However, some graphics packages also allow positions to be specified using relative coordinates.
- This method is useful for various graphics applications, such as producing drawings with pen plotters, artist's drawing and painting systems, and graphics packages for publishing and printing applications.
- Taking this approach, we can specify a coordinate position as an offset from the last position that was referenced (called the current position).

Specifying a Two-Dimensional World-Coordinate Reference Frame in OpenGL

- The gluOrtho2D command is a function we can use to set up any 2D Cartesian reference frames.
- The arguments for this function are the four values defining the x and y coordinate limits for the picture we want to display.
- Since the gluOrtho2D function specifies an orthogonal projection, we need also to be sure that the coordinate values are placed in the OpenGL projection matrix.
- In addition, we could assign the identity matrix as the projection matrix before defining the world-coordinate range.
- This would ensure that the coordinate values were not accumulated with any values we may have previously set for the projection matrix.
- Thus, for our initial two-dimensional examples, we can define the coordinate frame for the screen display window with the following statements

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ( );
gluOrtho2D (xmin, xmax, ymin, ymax);
```

- The display window will then be referenced by coordinates (xmin, ymin) at the lower-left corner and by coordinates (xmax, ymax) at the upper-right corner, as shown in Figure below

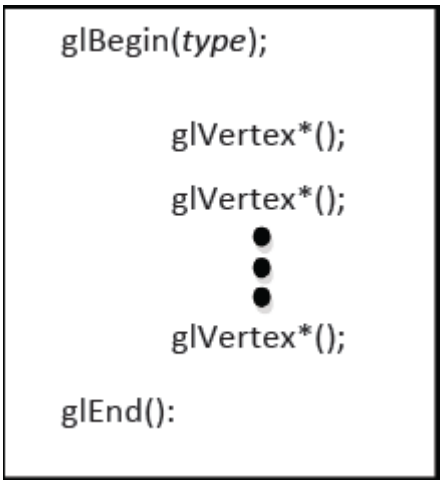


- We can then designate one or more graphics primitives for display using the coordinate reference specified in the gluOrtho2D statement.
- If the coordinate extents of a primitive are within the coordinate range of the display window, all of the primitive will be displayed.
- Otherwise, only those parts of the primitive within the display-window coordinate limits will be shown.
- Also, when we set up the geometry describing a picture, all positions for the OpenGL primitives must be given in absolute coordinates, with respect to the reference frame defined in the gluOrtho2D function.

OpenGL Functions

Geometric Primitives:

- It includes points, line segments, polygon etc.
- These primitives pass through geometric pipeline which decides whether the primitive is visible or not and also how the primitive should be visible on the screen etc.
- The geometric transformations such rotation, scaling etc can be applied on the primitives which are displayed on the screen.The programmer can create geometric primitives as shown below:



where:

glBegin indicates the beginning of the object that has to be displayed

glEnd indicates the end of primitive

OpenGL Point Functions

- The type within glBegin() specifies the type of the object and its value can be as follows:

  GL_POINTS

- Each vertex is displayed as a point.
- The size of the point would be of at least one pixel.
- Then this coordinate position, along with other geometric descriptions we may have in our scene, is passed to the viewing routines.
- Unless we specify other attribute values, OpenGL primitives are displayed with a default size and color.
- The default color for primitives is white, and the default point size is equal to the size of a single screen pixel

Syntax:

Case 1:

glBegin (GL_POINTS);

glVertex2i (50, 100);

glVertex2i (75, 150);

glVertex2i (100, 200);

glEnd ( );

Case 2:

- we could specify the coordinate values for the preceding points in arrays such as

  int point1 [ ] = {50, 100};

  int point2 [ ] = {75, 150};

  int point3 [ ] = {100, 200};

and call the OpenGL functions for plotting the three points as

glBegin (GL_POINTS);

glVertex2iv (point1);

glVertex2iv (point2);

glVertex2iv (point3);

glEnd ( );

Case 3:

- specifying two point positions in a three dimensional world reference frame. In this case, we give the coordinates as explicit floating-point values:

glBegin (GL_POINTS);

glVertex3f (-78.05, 909.72, 14.60);

glVertex3f (261.91, -5200.67, 188.33);

glEnd ( );

OpenGL LINE FUNCTIONS

- Primitive type is GL_LINES
- Successive pairs of vertices are considered as endpoints and they are connected to form an individual line segments.
- Note that successive segments usually are disconnected because the vertices are processed on a pair-wise basis.
- we obtain one line segment between the first and second coordinate positions and another line segment between the third and fourth positions.
- if the number of specified endpoints is odd, so the last coordinate position is ignored.

Case 1: Lines

glBegin (GL_LINES);

glVertex2iv (p1);

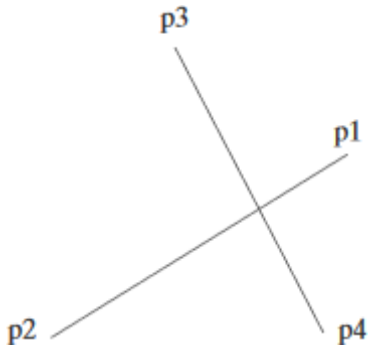
glVertex2iv (p2);

glVertex2iv (p3);

glVertex2iv (p4);

glVertex2iv (p5);

glEnd ( );



Case 2: GL_LINE_STRIP:

Successive vertices are connected using line segments. However, the final vertex is not connected to the initial vertex.

glBegin (GL_LINES_STRIP);

glVertex2iv (p1);

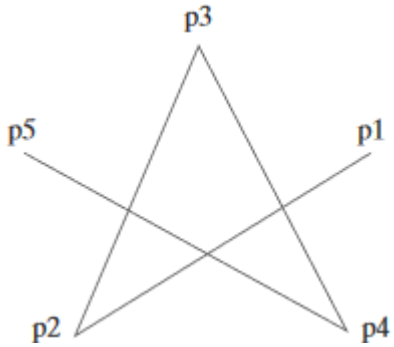
glVertex2iv (p2);

glVertex2iv (p3);

glVertex2iv (p4);

glVertex2iv (p5);

glEnd ( );



Case 3: GL_LINE_LOOP:

Successive vertices are connected using line segments to form a closed path or loop i.e., final vertex is connected to the initial vertex.

glBegin (GL_LINES_LOOP);

glVertex2iv (p1);

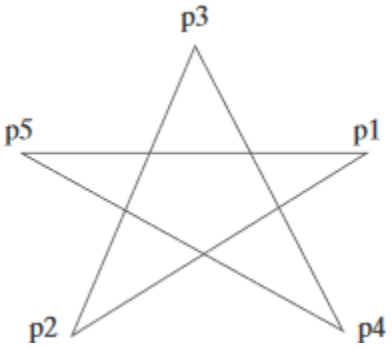
glVertex2iv (p2);

glVertex2iv (p3);

glVertex2iv (p4);

glVertex2iv (p5);

glEnd ( );



Point Attributes

- ➔ Basically, we can set two attributes for points: color and size.
- ➔ In a state system: The displayed color and size of a point is determined by the current values stored in the attribute list.
- ➔ Color components are set with RGB values or an index into a color table.
- ➔ For a raster system: Point size is an integer multiple of the pixel size, so that a large point is displayed as a square block of pixels

Opengl Point-Attribute Functions

Color:

- ➔ The displayed color of a designated point position is controlled by the current color values in the state list.
- ➔ Also, a color is specified with either the glColor function or the glIndex function.

Size:

- ➔ We set the size for an OpenGL point with

  glPointSize (size);

  and the point is then displayed as a square block of pixels.
- ➔ Parameter size is assigned a positive floating-point value, which is rounded to an integer (unless the point is to be antialiased).
- ➔ The number of horizontal and vertical pixels in the display of the point is determined by parameter size.
- ➔ Thus, a point size of 1.0 displays a single pixel, and a point size of 2.0 displays a 2×2 pixel array.
- ➔ If we activate the antialiasing features of OpenGL, the size of a displayed block of pixels will be modified to smooth the edges.
- ➔ The default value for point size is 1.0.

Example program:

- ➔ Attribute functions may be listed inside or outside of a glBegin/glEnd pair.
- ➔ Example: the following code segment plots three points in varying colors and sizes.

➔ The first is a standard-size red point, the second is a double-size green point, and the third is a triple-size blue point:

Ex:

```
glColor3f (1.0, 0.0, 0.0);
        glBegin (GL_POINTS);
        glVertex2i (50, 100);
        glPointSize (2.0);
        glColor3f (0.0, 1.0, 0.0);
        glVertex2i (75, 150);
        glPointSize (3.0);
        glColor3f (0.0, 0.0, 1.0);
        glVertex2i (100, 200);
glEnd ( );
```

Line-Attribute Functions OpenGL

➔ In OpenGL straight-line segment with three attribute settings: line color, line-width, and line style.

➔ OpenGL provides a function for setting the width of a line and another function for specifying a line style, such as a dashed or dotted line.

OpenGL Line-Width Function

➔ Line width is set in OpenGL with the function

Syntax: glLineWidth (width);

➔ We assign a floating-point value to parameter width, and this value is rounded to the nearest nonnegative integer.

➔ If the input value rounds to 0.0, the line is displayed with a standard width of 1.0, which is the default width.

➔ Some implementations of the line-width function might support only a limited number of widths, and some might not support widths other than 1.0.

➔ That is, the magnitude of the horizontal and vertical separations of the line endpoints, deltax and deltay, are compared to determine whether to generate a thick line using vertical pixel spans or horizontal pixel spans.

OpenGL Line-Style Function

➔ By default, a straight-line segment is displayed as a solid line.

➔ But we can also display dashed lines, dotted lines, or a line with a combination of dashes and dots.

➔ We can vary the length of the dashes and the spacing between dashes or dots.

➔ We set a current display style for lines with the OpenGL function:

Syntax: glLineStipple (repeatFactor, pattern);

Pattern:

➔ Parameter pattern is used to reference a 16-bit integer that describes how the line should be displayed.

➔ 1 bit in the pattern denotes an "on" pixel position, and a 0 bit indicates an "off" pixel position.

➔ The pattern is applied to the pixels along the line path starting with the low-order bits in the pattern.

➔ The default pattern is 0xFFFF (each bit position has a value of 1),which produces a solid line.

repeatFactor

➔ Integer parameter repeatFactor specifies how many times each bit in the pattern is to be repeated before the next bit in the pattern is applied.

➔ The default repeat value is 1.

Polyline:

➔ With a polyline, a specified line-style pattern is not restarted at the beginning of each segment.

➔ It is applied continuously across all the segments, starting at the first endpoint of the polyline and ending at the final endpoint for the last segment in the series.

Example:

➔ For line style, suppose parameter pattern is assigned the hexadecimal representation 0x00FF and the repeat factor is 1.

➔ This would display a dashed line with eight pixels in each dash and eight pixel positions that are "off" (an eight-pixel space) between two dashes.

➔ Also, since low order bits are applied first, a line begins with an eight-pixel dash starting at the first endpoint.

➔ This dash is followed by an eight-pixel space, then another eight-pixel dash, and so forth, until the second endpoint position is reached.

Activating line style:

- Before a line can be displayed in the current line-style pattern, we must activate the line-style feature of OpenGL.

  glEnable (GL_LINE_STIPPLE);

- If we forget to include this enable function, solid lines are displayed; that is, the default pattern 0xFFFF is used to display line segments.
- At any time, we can turn off the line-pattern feature with

  glDisable (GL_LINE_STIPPLE);

- This replaces the current line-style pattern with the default pattern (solid lines).

Example Code:

```
typedef struct { float x, y; } wcPt2D;
wcPt2D dataPts [5];
void linePlot (wcPt2D dataPts [5])
{
        int k;
        glBegin (GL_LINE_STRIP);
        for (k = 0; k < 5; k++)
                glVertex2f (dataPts [k].x, dataPts [k].y);
```

```
        glFlush ( );
        glEnd ( );
}
/* Invoke a procedure here to draw coordinate axes. */
glEnable (GL_LINE_STIPPLE); /* Input first set of (x, y) data values. */
glLineStipple (1, 0x1C47); // Plot a dash-dot, standard-width polyline.
linePlot (dataPts);
/* Input second set of (x, y) data values. */
glLineStipple (1, 0x00FF); / / Plot a dashed, double-width polyline.
glLineWidth (2.0);
linePlot (dataPts);
/* Input third set of (x, y) data values. */
glLineStipple (1, 0x0101); // Plot a dotted, triple-width polyline.
glLineWidth (3.0);
linePlot (dataPts);
glDisable (GL_LINE_STIPPLE);
```

Curve Attributes

- ➔ Parameters for curve attributes are the same as those for straight-line segments.
- ➔ We can display curves with varying colors, widths, dot-dash patterns, and available pen or brush options.
- ➔ Methods for adapting curve-drawing algorithms to accommodate attribute selections are similar to those for line drawing.
- ➔ Raster curves of various widths can be displayed using the method of horizontal or vertical pixel spans.

  Case 1: Where the magnitude of the curve slope |m| <= 1.0, we plot vertical spans;

  Case 2: when the slope magnitude |m| > 1.0, we plot horizontal spans.

Different methods to draw a curve:

Method 1: Using circle symmetry property, we generate the circle path with vertical spans in the octant from x = 0 to x = y, and then reflect pixel positions about the line y = x to y=0

Method 2: 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, however, shifts the original curve path either inward or outward, depending on which direction we choose for the second boundary.

Method 3: The pixel masks discussed for implementing line-style options could also be used in raster curve algorithms to generate dashed or dotted patterns

Method 4: Pen (or brush) displays of curves are generated using the same techniques discussed for straight-line segments.

Method 5: Painting and drawing programs allow pictures to be constructed interactively by using a pointing device, such as a stylus and a graphics tablet, to sketch various curve shapes.

Line Drawing Algorithm

- ✓ A straight-line segment in a scene is defined by coordinate positions for the endpoints of the segment.
- ✓ To display the line on a raster monitor, the graphics system must first project the endpoints to integer screen coordinates and determine the nearest pixel positions along the line path between the two endpoints then the line color is loaded into the frame buffer at the corresponding pixel coordinates
- ✓ The Cartesian slope-intercept equation for a straight line is

  y=m * x +b---------->(1)

  with m as the slope of the line and b as the y intercept.
- ✓ Given that the two endpoints of a line segment are specified at positions (x0,y0) and (xend, yend) ,as shown in fig.

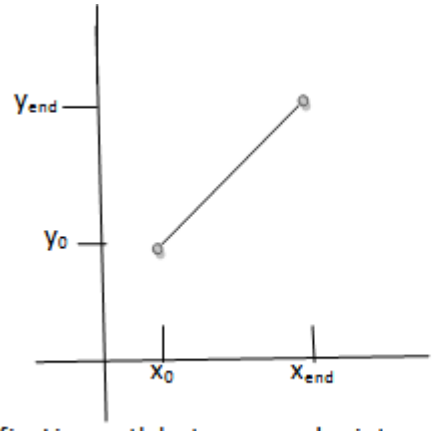


fig. Line path between endpoint positions (x_0, y_0) and (x_{end}, y_{end}).

- ✓ We determine values for the slope m and y intercept b with the following equations:

  m=(yend - y0)/(xend - x0) ----------->(2)

  b=y0 - m.x0---------- >(3)

- ✓ Algorithms for displaying straight line are based on the line equation (1) and calculations given in eq(2) and (3).
- ✓ For given x interval δx along a line, we can compute the corresponding y interval δy from eq.(2) as

  δy=m. δx ---------- >(4)

- ✓ Similarly, we can obtain the x interval δx corresponding to a specified δy as

  δx=δy/m---------- >(5)

- ✓ These equations form the basis for determining deflection voltages in analog displays, such as vector-scan system, where arbitrarily small changes in deflection voltage are possible.
- ✓ For lines with slope magnitudes
  - ➔ |m|<1, δx can be set proportional to a small horizontal deflection voltage with the corresponding vertical deflection voltage set proportional to δy from eq.(4)
  - ➔ |m|>1, δy can be set proportional to a small vertical deflection voltage with the corresponding horizontal deflection voltage set proportional to δx from eq.(5)
  - ➔ |m|=1, δx=δy and the horizontal and vertical deflections voltages are equal

DDA Algorithm (DIGITAL DIFFERENTIAL ANALYZER)

➔ The DDA is a scan-conversion line algorithm based on calculating either δy or δx.

➔ A line is sampled at unit intervals in one coordinate and the corresponding integer values nearest the line path are determined for the other coordinate

➔ DDA Algorithm has three cases so from equation i.e.., $m=(y_{k+1} - y_k)/(x_{k+1} - x_k)$

Case1:

if m<1,x increment in unit intervals

i.e..,$x_{k+1}=x_k+1$

then, $m=(y_{k+1} - y_k)/(x_{k+1} - x_k)$

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

$y_{k+1} = y_k + m$ ---------- >(1)

➔ where k takes integer values starting from 0,for the first point and increases by 1 until final endpoint is reached. Since m can be any real number between 0.0 and 1.0,

Case2:

if m>1, y increment in unit intervals

i.e.., $y_{k+1} = y_k + 1$

then, $m= (y_k + 1- y_k)/(x_{k+1} - x_k)$

$m(x_{k+1} - x_k)=1$

$x_{k+1} =(1/m)+ x_k$ ---------- (2)

Case3:

if m=1,both x and y increment in unit intervals

i.e..,$x_{k+1}=x_k + 1$ and $y_{k+1} = y_k + 1$

Equations (1) and (2) 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 at the right, then either we have δx=-1 and

$y_{k+1} = y_k - m$.....(3)

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

$x_{k+1} = x_k - (1/m)$ ---------- (4)

➔ Similar calculations are carried out using equations (1) through (4) to determine the pixel positions along a line with negative slope. thus, if the absolute value of the slope is less than 1 and the starting endpoint is at left ,we set δx==1 and calculate y values with eq(1).

➔ when starting endpoint is at the right(for the same slope),we set δx=-1 and obtain y positions using eq(3).

➔ This algorithm is summarized in the following procedure, which accepts as input two integer screen positions for the endpoints of a line segment.

➔ if m<1,where x is incrementing by 1

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

➔ So initially x=0,Assuming (x0,y0)as initial point assigning x= x0,y=y0 which is the starting point .

- Illuminate pixel(x, round(y))
- x1= x+ 1 , y1=y + 1
- Illuminate pixel(x1,round(y1))
- x2= x1+ 1 , y2=y1 + 1
- Illuminate pixel(x2,round(y2))
- Till it reaches final point.

➔ if m>1,where y is incrementing by 1

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

➔ So initially y=0,Assuming (x0,y0)as initial point assigning x= x0,y=y0 which is the starting point .

- Illuminate pixel(round(x),y)
- x1= x+(1/m) ,y1=y
- Illuminate pixel(round(x1),y1)
- x2= x1+ (1/m) , y2=y1
- Illuminate pixel(round(x2),y2)
- Till it reaches final point.

➔ The DDA algorithm is faster method for calculating pixel position than one that directly implements .

- ➔ It eliminates the multiplication by making use of raster characteristics, so that appropriate increments are applied in the x or y directions to step from one pixel position to another along the line path.
- ➔ The accumulation of round off error in successive additions of the floating point increment, however can cause the calculated pixel positions to drift away from the true line path for long line segments. Furthermore ,the rounding operations and floating point arithmetic in this procedure are still time consuming.
- ➔ we improve the performance of DDA algorithm by separating the increments m and 1/m into integer and fractional parts so that all calculations are reduced to integer operations.

```
#include <stdlib.h>
#include <math.h>
inline int round (const float a)
{
        return int (a + 0.5);
}
void lineDDA (int x0, int y0, int xEnd, int yEnd)
{
        int dx = xEnd - x0, dy = yEnd - y0, steps, k;
        float xIncrement, yIncrement, x = x0, y = y0;
        if (fabs (dx) > fabs (dy))
                steps = fabs (dx);
        else
                steps = fabs (dy);
        xIncrement = float (dx) / float (steps);
        yIncrement = float (dy) / float (steps);
        setPixel (round (x), round (y));
        for (k = 0; k < steps; k++) {
                x += xIncrement;
                y += yIncrement;
                setPixel (round (x), round (y));
        }
```

}

Bresenham's Algorithm:

- ➔ It is an efficient raster scan generating algorithm that uses incremental integral calculations
- ➔ To illustrate Bresenham's approach, we first consider the scan-conversion process for lines with positive slope less than 1.0.
- ➔ Pixel positions along a line path are then determined by sampling at unit x intervals. Starting from the left endpoint (x0, y0) of a given line, we step to each successive column (x position) and plot the pixel whose scan-line y value is closest to the line path.

- ➔ Consider the equation of a straight line y=mx+c where m=dy/dx

Bresenham's Line-Drawing Algorithm for |m| < 1.0

1. Input the two line endpoints and store the left endpoint in (x0, y0).
2. Set the color for frame-buffer position (x0, y0); i.e., plot the first point.
3. Calculate the constants Δx, Δy, $2\Delta y$, and $2\Delta y - 2\Delta x$, and obtain the starting value for the decision parameter as

$$\mathbf{p0 = 2\Delta y - \Delta x}$$

4. At each xk along the line, starting at k = 0, perform the following test:

If pk < 0, the next point to plot is (xk + 1, yk) and

$$\mathbf{p_{k+1} = p_k + 2\Delta y}$$

Otherwise, the next point to plot is (xk + 1, yk + 1) and

$$\mathbf{p_{k+1} = p_k + 2\Delta y - 2\Delta x}$$

5. Repeat step 4 $\Delta x - 1$ more times.

Note:

If |m|>1.0

Then

$$\mathbf{p0 = 2\Delta x - \Delta y}$$

and

If pk < 0, the next point to plot is (xk , yk +1) and

$$p_{k+1} = p_k + 2\Delta x$$

Otherwise, the next point to plot is (xk + 1, yk + 1) and

$$p_{k+1} = p_k + 2\Delta x - 2\Delta y$$

Code:

```
#include <stdlib.h>
#include <math.h>
/* Bresenham line-drawing procedure for |m| < 1.0. */
void lineBres (int x0, int y0, int xEnd, int yEnd)
{
    int dx = fabs (xEnd - x0), dy = fabs(yEnd - y0);
    int p = 2 * dy - dx;
    int twoDy = 2 * dy, twoDyMinusDx = 2 * (dy - dx);
    int x, y;
    /* Determine which endpoint to use as start position. */
    if (x0 > xEnd) {
        x  =  xEnd;
        y  =  yEnd;
        xEnd = x0;
    }
    else {
        x = x0;
        y = y0;
    }
    setPixel (x, y);
    while (x < xEnd) {
        x++;
        if (p < 0)
        p += twoDy;
```

```
            else {
                y++;
                p += twoDyMinusDx;
            }
            setPixel (x, y);
        }
}
```

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).
- ➔ For any circle point (x, y), this distance relationship is expressed by the Pythagorean theorem in Cartesian coordinates as

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

- ➔ 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

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

- ➔ One problem with this approach is that it involves considerable computation at each step. Moreover, the spacing between plotted pixel positions is not uniform.
- ➔ We could adjust the spacing by interchanging x and y (stepping through y values and calculating x values) whenever the absolute value of the slope of the circle is greater than 1; but this simply increases the computation and processing required by the algorithm.
- ➔ 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

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

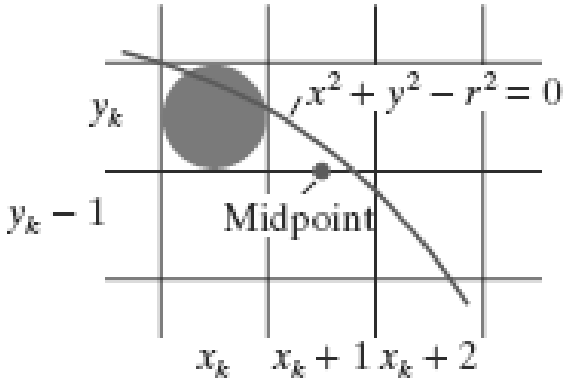
Midpoint Circle Algorithm

- ➔ Midpoint circle algorithm generates all points on a circle centered at the origin by incrementing all the way around circle.
- ➔ The strategy is to select which of 2 pixels is closer to the circle by evaluating a function at the midpoint between the 2 pixels
- ➔ To apply the midpoint method, we define a circle function as

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

- ➔ To summarize, the relative position of any point (x, y) can be determined by checking the sign of the circle function as follows:

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



Eight way symmetry

- ➔ The shape of the circle is similar in each quadrant.
- ➔ Therefore ,if we determine the curve positions in the first quadrant ,we can generate the circle positions in the second quadrant of xy plane.
- ➔ The circle sections in the third and fourth quadrant can be obtained from sections in the first and second quadrant by considering the symmetry along X axis

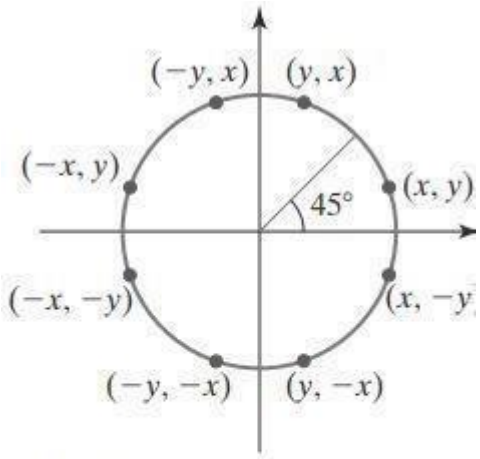


FIGURE 13
Symmetry of a circle. Calculation of a circle point (x, y) in one octant yields the circle points shown for the other seven octants.

➔ Conside the circle centered at the origin,if the point (x, y) is on the circle,then we can compute 7 other points on the circle as shown in the above figure.

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

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

➔ Successive decision parameters are obtained using incremental calculations.

➔ We obtain a recursive expression for the next decision parameter by evaluating the circle function at sampling position $x_{k+1} + 1 = x_k + 2$:

$$p_{k+1} = f_{\text{circ}}\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right)$$
$$= [(x_k + 1) + 1]^2 + \left(y_{k+1} - \frac{1}{2}\right)^2 - r^2$$

or

$$p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$$

where y_{k+1} is either y_k or $y_k - 1$, depending on the sign of p_k.

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

$$p_0 = f_{\text{circ}}\left(1, r - \frac{1}{2}\right)$$

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

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

➔ If the radius r is specified as an integer, we can simply round p0 to

p0 = 1 − r (for r an integer)

because all increments are integers.

Midpoint Circle Algorithm

1. Input radius r and circle center (x_c, y_c), then set the coordinates for 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

$$p0 = 1\text{-}r$$

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, yk) 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

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$

where $2x_{k+1} = 2xk + 2$ and $2y_{k+1} = 2yk - 2$.

4. Determine symmetry points in the other seven octants.

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

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

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

Code:

```
void draw_pixel(GLint cx, GLint cy)
{
        glColor3f(0.5,0.5,0.0);
        glBegin(GL_POINTS);
                glVertex2i(cx, cy);
        glEnd();
}

void plotpixels(GLint h, GLint k, GLint x, GLint y)
{
        draw_pixel(x+h, y+k);
        draw_pixel(-x+h, y+k);
        draw_pixel(x+h, -y+k);
        draw_pixel(-x+h, -y+k);
        draw_pixel(y+h, x+k);
        draw_pixel(-y+h, x+k);
        draw_pixel(y+h, -x+k);
        draw_pixel(-y+h, -x+k);
}

void circle_draw(GLint xc, GLint yc, GLint r)
{
        GLint d=1-r, x=0,y=r;
        while(y>x)
        {
                plotpixels(xc, yc, x, y);
                if(d<0) d+=2*x+3;
                else
                {
```

```
                d+=2*(x-y)+5;
                --y;
            }
            ++x;
        }
        plotpixels(xc, yc, x, y);
}
```

MODULE 2: FILL AREA PRIMITIVES, 2D GEOMETRIC TRANSFORMATIONS AND 3D VIEWING.

Fill area Primitives:

Introduction

Polygon fill-areas,

OpenGL polygon Fill Area Functions,

Fill area attributes,

General scan line polygon fill algorithm,

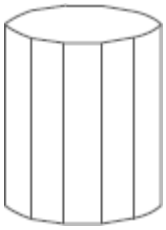
OpenGL fill-area Attribute functions.

Introduction

- An useful construct for describing components of a picture is an area that is filled with some solid color or pattern.
- A picture component of this type is typically referred to as a **fill area or a filled area**.
- Any fill-area shape is possible, graphics libraries generally do not support specifications for arbitrary fill shapes
- Figure below illustrates a few possible fill-area shapes.

- Graphics routines can more efficiently process polygons than other kinds of fill shapes because polygon boundaries are described with linear equations.
- When lighting effects and surface-shading procedures are applied, an approximated curved surface can be displayed quite realistically.
- Approximating a curved surface with polygon facets is sometimes referred to as ***surface tessellation,*** **or fitting the surface with a** ***polygon mesh.***

- Below figure shows the side and top surfaces of a metal cylinder approximated in an outline form as a polygon mesh.



- Displays of such figures can be generated quickly as *wire-frame* views, showing only the polygon edges to give a general indication of the surface structure
- Objects described with a set of polygon surface patches are usually referred to as standard graphics objects, or just graphics objects.

Polygon Fill Areas

- ✓ A polygon is a plane figure specified by a set of three or more coordinate positions, called *vertices*, that are connected in sequence by straight-line segments, called the *edges* or *sides* of the polygon.
- ✓ It is required that the polygon edges have no common point other than their endpoints.
- ✓ Thus, by definition, a polygon must have all its vertices within a single plane and there can be no edge crossings
- ✓ Examples of polygons include triangles, rectangles, octagons, and decagons
- ✓ Any plane figure with a closed-polyline boundary is alluded to as a polygon, and one with no crossing edges is referred to as a *standard polygon* or a *simple polygon*

Problem:

- ➢ For a computer-graphics application, it is possible that a designated set of polygon vertices do not all lie exactly in one plane
- ➢ This is due to roundoff error in the calculation of numerical values, to errors in selecting coordinate positions for the vertices, or, more typically, to approximating a curved surface with a set of polygonal patches

Solution:

- ➢ To divide the specified surface mesh into triangles

Polygon Classifications

- ✓ Polygons are classified into two types
  1. Convex Polygon and
  2. Concave Polygon

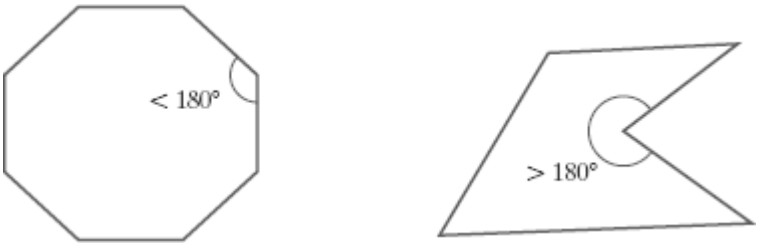
Convex Polygon:

- ✓ The polygon is convex if all interior angles of a polygon are less than or equal to 180◦, where an interior angle of a polygon is an angle inside the polygon boundary that is formed by two adjacent edges
- ✓ An equivalent definition of a convex polygon is that its interior lies completely on one side of the infinite extension line of any one of its edges.
- ✓ Also, if we select any two points in the interior of a convex polygon, the line segment joining the two points is also in the interior.

Concave Polygon:

- ✓ A polygon that is not convex is called a concave polygon.

Te below figure shows convex and concave polygon



- ✓ The term degenerate polygon is often used to describe a set of vertices that are collinear or that have repeated coordinate positions.

Problems in concave polygon:

➔ Implementations of fill algorithms and other graphics routines are more complicated

Solution:

➔ It is generally more efficient to split a concave polygon into a set of convex polygons before processing

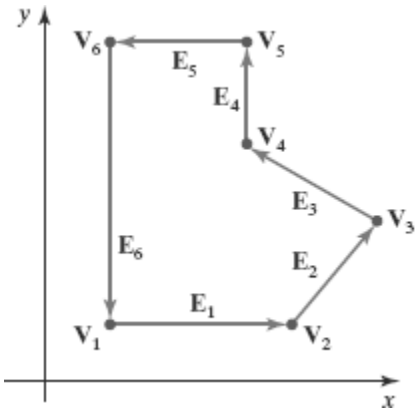
Identifying Concave Polygons

Characteristics:

- A concave polygon has at least one interior angle greater than 180◦.
- The extension of some edges of a concave polygon will intersect other edges, and
- Some pair of interior points will produce a line segment that intersects the polygon boundary

Identification algorithm 1

- Identifying a concave polygon by calculating cross-products of successive pairs of edge vectors.
- If we set up a vector for each polygon edge, then we can use the cross-product of adjacent edges to test for concavity. All such vector products will be of the same sign (positive or negative) for a convex polygon.
- Therefore, if some cross-products yield a positive value and some a negative value, we have a concave polygon



$$(\mathbf{E}_1 \times \mathbf{E}_2)_z > 0$$
$$(\mathbf{E}_2 \times \mathbf{E}_3)_z > 0$$
$$(\mathbf{E}_3 \times \mathbf{E}_4)_z < 0$$
$$(\mathbf{E}_4 \times \mathbf{E}_5)_z > 0$$
$$(\mathbf{E}_5 \times \mathbf{E}_6)_z > 0$$
$$(\mathbf{E}_6 \times \mathbf{E}_1)_z > 0$$

Identification algorithm 2

- Look at the polygon vertex positions relative to the extension line of any edge.
- If some vertices are on one side of the extension line and some vertices are on the other side, the polygon is concave.

Splitting Concave Polygons

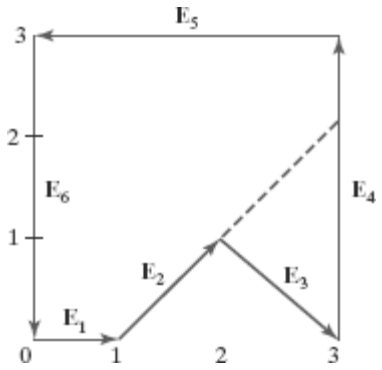
- ✓ Split concave polygon it into a set of convex polygons using edge vectors and edge cross-products; or, we can use vertex positions relative to an edge extension line to determine which vertices are on one side of this line and which are on the other.

Vector method

- ➔ First need to form the edge vectors.
- ➔ Given two consecutive vertex positions, V_k and V_{k+1}, we define the edge vector between them as

$$E_k = V_{k+1} - V_k$$

- ➔ Calculate the cross-products of successive edge vectors in order around the polygon perimeter.
- ➔ If the *z* component of some cross-products is positive while other cross-products have a negative *z* component, the polygon is concave.
- ➔ We can apply the vector method by processing edge vectors in counterclockwise order If any cross-product has a negative *z* component (as in below figure), the polygon is concave and we can split it along the line of the first edge vector in the cross-product pair



E1 = *(*1, 0, 0*)* E2 = *(*1, 1, 0*)*
E3 = *(*1, −1, 0*)* E4 = *(*0, 2, 0*)*
E5 = *(*−3, 0, 0*)* E6 = *(*0, −2, 0*)*

- ➔ Where the *z* component is 0, since all edges are in the *xy* plane.
- ➔ The crossproduct $E_j \times E_k$ for two successive edge vectors is a vector perpendicular to the *xy* plane with *z* component equal to $E_{jx}E_{ky} - E_{kx}E_{jy}$:
- ➔ The values for the above figure is as follows

E1 × E2 = *(*0, 0, 1*)* E2 × E3 = *(*0, 0, −2*)*
E3 × E4 = *(*0, 0, 2*)* E4 × E5 = *(*0, 0, 6*)*

$E5 \times E6 = (0, 0, 6)$ $E6 \times E1 = (0, 0, 2)$

➔ Since the cross-product E2 × E3 has a negative *z* component, we split the polygon along the line of vector E2.

➔ The line equation for this edge has a slope of 1 and a *y* intercept of −1 . No other edge cross-products are negative, so the two new polygons are both convex.

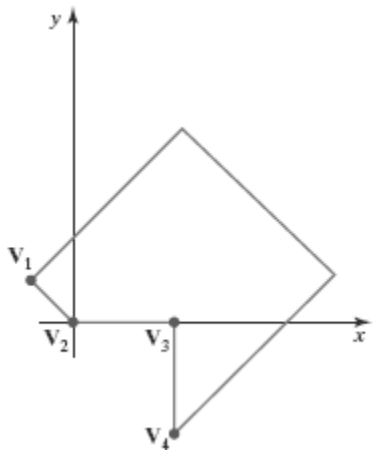
Rotational method

➔ Proceeding counterclockwise around the polygon edges, we shift the position of the polygon so that each vertex V_k in turn is at the coordinate origin.

➔ We rotate the polygon about the origin in a clockwise direction so that the next vertex V_{k+1} is on the *x* axis.

➔ If the following vertex, V_{k+2}, is below the *x* axis, the polygon is concave.

➔ We then split the polygon along the *x* axis to form two new polygons, and we repeat the concave test for each of the two new polygons



Splitting a Convex Polygon into a Set of Triangles

- Once we have a vertex list for a convex polygon, we could transform it into a set of triangles.
- First define any sequence of three consecutive vertices to be a new polygon (a triangle).
- The middle triangle vertex is then deleted from the original vertex list .
- The same procedure is applied to this modified vertex list to strip off another triangle.
- We continue forming triangles in this manner until the original polygon is reduced to just three vertices, which define the last triangle in the set.
- Concave polygon can also be divided into a set of triangles using this approach, although care must be taken that the new diagonal edge formed by joining the first and third selected vertices does not cross the concave portion of the polygon, and that the three selected vertices at each step form an interior angle that is less than 180◦

Identifying interior and exterior region of polygon

- We may want to specify a complex fill region with intersecting edges.
- 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 boundaries.
- Two commonly used algorithms
    1. Odd-Even rule and
    2. The nonzero winding-number rule.

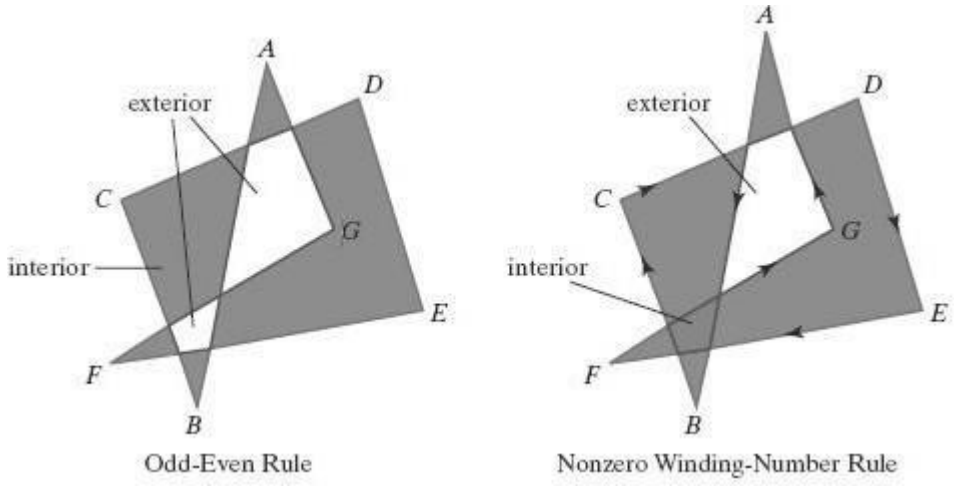
Inside-Outside Tests

- ✓ Also called the *odd-parity rule* or the *even-odd rule.*
- ✓ Draw a line from any position P to a distant point outside the coordinate extents of the closed polyline.
- ✓ Then we count the number of line-segment crossings along this line.
- ✓ If the number of segments crossed by this line is odd, then P is considered to be an *interior* point Otherwise, P is an *exterior* point
- ✓ We can use this procedure, for example,to fill the interior region between two concentric circles or two concentric polygons with a specified color.

Nonzero Winding-Number rule

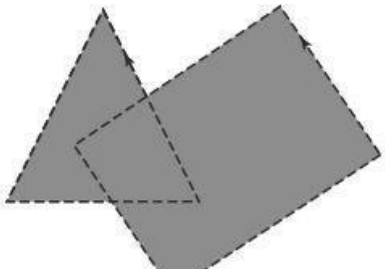
- ✓ This counts the number of times that the boundary of an object "winds" around a particular point in the counterclockwise direction termed as winding number,
- ✓ Initialize the winding number to 0 and again imagining a line drawn from any position P to a distant point beyond the coordinate extents of the object.
- ✓ The line we choose must not pass through any endpoint coordinates.
- ✓ As we move along the line from position P to the distant point, we count the number of object line segments that cross the reference line in each direction
- ✓ We add 1 to the winding number every time we intersect a segment that crosses the line in the direction from right to left, and we subtract 1 very time we intersect a segment that crosses from left to right

- ✓ If the winding number is nonzero, P is considered to be an interior point. Otherwise, P is taken to be an exterior point

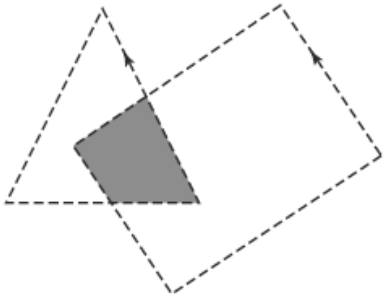


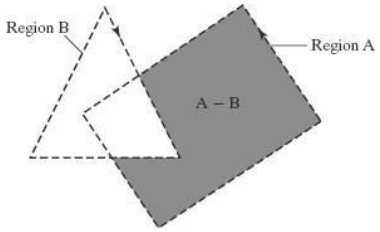
Odd-Even Rule Nonzero Winding-Number Rule

- ✓ The nonzero winding-number rule tends to classify as interior some areas that the odd-even rule deems to be exterior.
- ✓ Variations of the nonzero winding-number rule can be used to define interior regions in other ways define a point to be interior if its winding number is positive or if it is negative; or we could use any other rule to generate a variety of fill shapes
- ✓ Boolean operations are used to specify a fill area as a combination of two regions
- ✓ One way to implement Boolean operations is by using a variation of the basic winding-number rule consider the direction for each boundary to be counterclockwise, the union of two regions would consist of those points whose winding number is positive



- ✓ The intersection of two regions with counterclockwise boundaries would contain those points whose winding number is greater than 1,

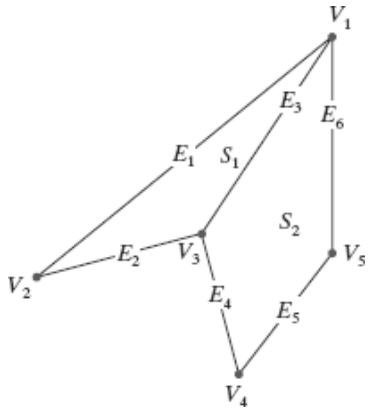




- To set up a fill area that is the difference of two regions (say, A − B), we can enclose region A with a counterclockwise border and B with a clockwise border

Polygon Tables

- ✓ The objects in a scene are described as sets of polygon surface facets
- ✓ The description for each object includes coordinate information specifying the geometry for the polygon facets and other surface parameters such as color, transparency, and light-reflection properties.
- ✓ The data of the polygons are placed into tables that are to be used in the subsequent processing, display, and manipulation of the objects in the scene
- ✓ These polygon data tables can be organized into two groups:
  1. Geometric tables and
  2. Attribute tables
- ✓ Geometric data tables contain vertex coordinates and parameters to identify the spatial orientation of the polygon surfaces.
- ✓ Attribute information for an object includes parameters specifying the degree of transparency of the object and its surface reflectivity and texture characteristics
- ✓ Geometric data for the objects in a scene are arranged conveniently in three lists: a vertex table, an edge table, and a surface-facet table.
- ✓ Coordinate values for each vertex in the object are stored in the vertex table.
- ✓ The edge table contains pointers back into the vertex table to identify the vertices for each polygon edge.
- ✓ And the surface-facet table contains pointers back into the edge table to identify the edges for each polygon



VERTEX TABLE	
V_1:	x_1, y_1, z_1
V_2:	x_2, y_2, z_2
V_3:	x_3, y_3, z_3
V_4:	x_4, y_4, z_4
V_5:	x_5, y_5, z_5

EDGE TABLE	
E_1:	V_1, V_2
E_2:	V_2, V_3
E_3:	V_3, V_1
E_4:	V_3, V_4
E_5:	V_4, V_5
E_6:	V_5, V_1

SURFACE-FACET TABLE	
S_1:	E_1, E_2, E_3
S_2:	E_3, E_4, E_5, E_6

- ✓ The object can be displayed efficiently by using data from the edge table to identify polygon boundaries.
- ✓ An alternative arrangement is to use just two tables: a vertex table and a surface-facet table this scheme is less convenient, and some edges could get drawn twice in a wire-frame display.
- ✓ Another possibility is to use only a surface-facet table, but this duplicates coordinate information, since explicit coordinate values are listed for each vertex in each polygon facet. Also the relationship between edges and facets would have to be reconstructed from the vertex listings in the surface-facet table.
- ✓ We could expand the edge table to include forward pointers into the surface-facet table so that a common edge between polygons could be identifiedmore rapidly the vertex table could be expanded to reference corresponding edges, for faster information retrieval

E_1:	V_1, V_2, S_1
E_2:	V_2, V_3, S_1
E_3:	V_3, V_1, S_1, S_2
E_4:	V_3, V_4, S_2
E_5:	V_4, V_5, S_2
E_6:	V_5, V_1, S_2

- ✓ Because the geometric data tables may contain extensive listings of vertices and edges for complex objects and scenes, it is important that the data be checked for consistency and completeness.
- ✓ Some of the tests that could be performed by a graphics package are

  (1) that every vertex is listed as an endpoint for at least two edges,

(2) that every edge is part of at least one polygon,

(3) that every polygon is closed,

(4) that each polygon has at least one shared edge, and

(5) that if the edge table contains pointers to polygons, every edge referenced by a polygon pointer has a reciprocal pointer back to the polygon.

Plane Equations

- Each polygon in a scene is contained within a plane of infinite extent.
- The general equation of a plane is

$$Ax + B y + C z + D = 0$$

Where,

- ➔ (x, y, z) is any point on the plane, and
- ➔ The coefficients A, B, C, and D (called *plane parameters*) are constants describing the spatial properties of the plane.

- We can obtain the values of A, B, C, and D by solving a set of three plane equations using the coordinate values for three noncollinear points in the plane for the three successive convex-polygon vertices, $(x1, y1, z1)$, $(x2, y2, z2)$, and $(x3, y3, z3)$, in a counterclockwise order and solve the following set of simultaneous linear plane equations for the ratios A/D, B/D, and C/D:

$$(A/D)xk + (B/D)yk + (C/D)zk = -1, \; k = 1, 2, 3$$

- The solution to this set of equations can be obtained in determinant form, using Cramer's rule, as

$$A = \begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{vmatrix} \qquad B = \begin{vmatrix} x_1 & 1 & z_1 \\ x_2 & 1 & z_2 \\ x_3 & 1 & z_3 \end{vmatrix}$$

$$C = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \qquad D = -\begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix}$$

- Expanding the determinants, we can write the calculations for the plane coefficients in the form

$$A = y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2)$$
$$B = z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2)$$
$$C = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$$
$$D = -x_1(y_2z_3 - y_3z_2) - x_2(y_3z_1 - y_1z_3) - x_3(y_1z_2 - y_2z_1)$$

- It is possible that the coordinates defining a polygon facet may not be contained within a single plane.
- We can solve this problem by dividing the facet into a set of triangles; or we could find an approximating plane for the vertex list.
- One method for obtaining an approximating plane is to divide the vertex list into subsets, where each subset contains three vertices, and calculate plane parameters *A, B, C, D*for each subset.

Front and Back Polygon Faces

- The side of a polygon that faces into the object interior is called the back face, and the visible, or outward, side is the front face .
- Every polygon is contained within an infinite plane that partitions space into two regions.
- Any point that is not on the plane and that is visible to the front face of a polygon surface section is said to be *in front of* (or *outside*) the plane, and, thus, outside the object.
- And any point that is visible to the back face of the polygon is *behind* (or *inside*) the plane.
- Plane equations can be used to identify the position of spatial points relative to the polygon facets of an object.
- For any point (*x*, *y*, *z*) not on a plane with parameters *A*, *B*, *C*, *D*, we have

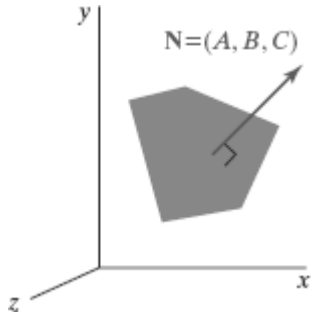
$$Ax + B\,y + C\,z + D \mathrel{!}= 0$$

- Thus, we can identify the point as either behind or in front of a polygon surface contained within that plane according to the sign (negative or positive) of

  $Ax + By + Cz + D$:

  if $Ax + B\,y + C\,z + D < 0$, the point (*x*, *y*, *z*) is behind the plane

  if $Ax + B\,y + C\,z + D > 0$, the point (*x*, *y*, *z*) is in front of the plane
- Orientation of a polygon surface in space can be described with the normal vector for the plane containing that polygon



- The normal vector points in a direction from inside the plane to the outside; that is, from the back face of the polygon to the front face.
- Thus, the normal vector for this plane is N = *(*1, 0, 0*)*, which is in the direction of the positive *x* axis.
- That is, the normal vector is pointing from inside the cube to the outside and is perpendicular to the plane $x = 1$.
- The elements of a normal vector can also be obtained using a vector crossproduct Calculation.
- We have a convex-polygon surface facet and a right-handed Cartesian system, we again select any three vertex positions,V1,V2, and V3, taken in counterclockwise order when viewing from outside the object toward the inside.
- Forming two vectors, one from V1 to V2 and the second from V1 to V3, we calculate N as the vector cross-product:

$$N = (V2 - V1) \times (V3 - V1)$$

- This generates values for the plane parameters *A*, *B*, and *C*.We can then obtain the value for parameter *D* by substituting these values and the coordinates in

$$Ax + By + Cz + D = 0$$

- The plane equation can be expressed in vector form using the normal N and the position P of any point in the plane as

$$N \cdot P = -D$$

OpenGL Polygon Fill-Area Functions

- ✓ A glVertex function is used to input the coordinates for a single polygon vertex, and a complete polygon is described with a list of vertices placed between a glBegin/glEnd pair.
- ✓ By default, a polygon interior is displayed in a solid color, determined by the current color settings we can fill a polygon with a pattern and we can display polygon edges as line borders around the interior fill.
- ✓ There are six different symbolic constants that we can use as the argument in the glBegin function to describe polygon fill areas
- ✓ In some implementations of OpenGL, the following routine can be more efficient than generating a fill rectangle using glVertex specifications:

```
glRect* (x1, y1, x2, y2);
```

- ✓ One corner of this rectangle is at coordinate position ($x1$, $y1$), and the opposite corner of the rectangle is at position ($x2$, $y2$).
- ✓ Suffix codes for glRect specify the coordinate data type and whether coordinates are to be expressed as array elements.
- ✓ These codes are i (for integer), s (for short), f (for float), d (for double), and v (for vector).
- ✓ Example

```
glRecti (200, 100, 50, 250);
```

If we put the coordinate values for this rectangle into arrays, we can generate the same square with the following code:

```
int vertex1 [ ] = {200, 100};
int vertex2 [ ] = {50, 250};
glRectiv (vertex1, vertex2);
```

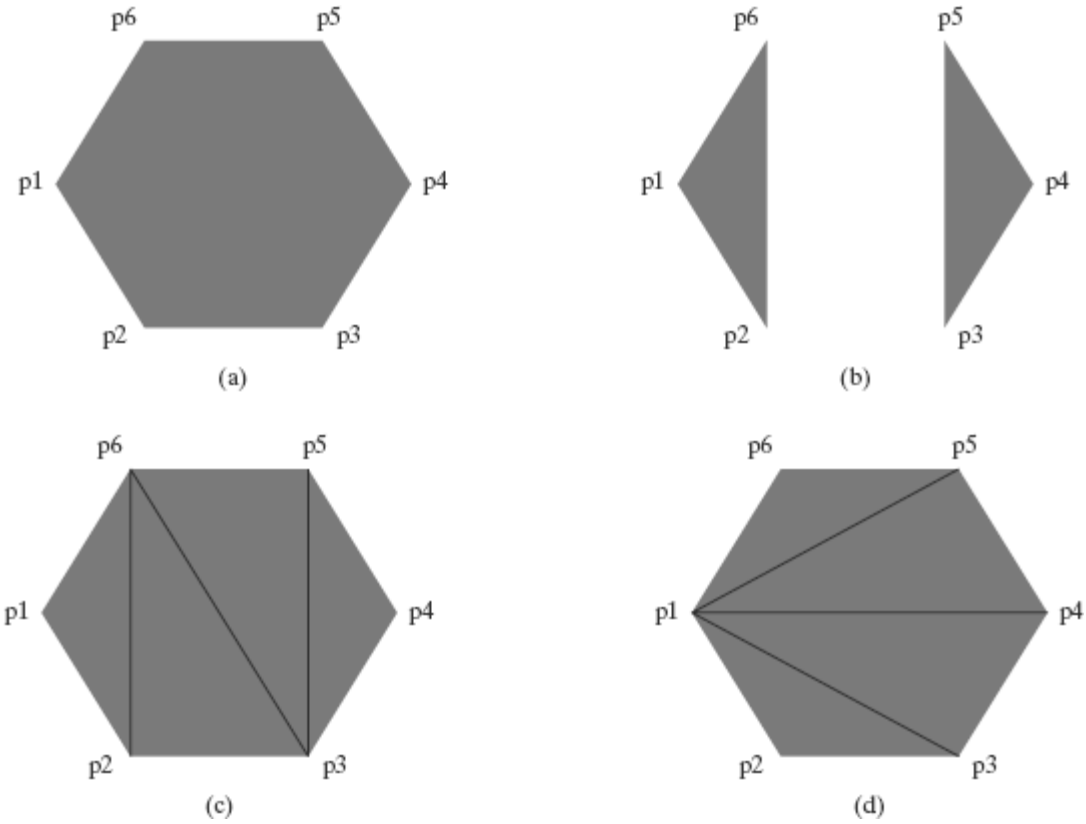
Polygon

- ❖ With the OpenGL primitive constant GL POLYGON, we can display a single polygon fill area.
- ❖ Each of the points is represented as an array of (x, y) coordinate values:

```
glBegin (GL_POLYGON);
    glVertex2iv (p1);
```

```
        glVertex2iv (p2);
        glVertex2iv (p3);
        glVertex2iv (p4);
        glVertex2iv (p5);
        glVertex2iv (p6);
    glEnd ( );
```

- A polygon vertex list must contain at least three vertices. Otherwise, nothing is displayed.



(a) A single convex polygon fill area generated with the primitive constant GL POLYGON. (b) Two unconnected triangles generated with GL TRIANGLES.

(c) Four connected triangles generated with GL TRIANGLE STRIP.

(d) Four connected triangles generated with GL TRIANGLE FAN.

Triangles

- Displays the trianlges.

- Three primitives in triangles, GL_TRIANGLES, GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP

```
glBegin (GL_TRIANGLES);
        glVertex2iv (p1);
        glVertex2iv (p2);
        glVertex2iv (p6);
        glVertex2iv (p3);
        glVertex2iv (p4);
        glVertex2iv (p5);
glEnd ( );
```

- In this case, the first three coordinate points define the vertices for one triangle, the next three points define the next triangle, and so forth.
- For each triangle fill area, we specify the vertex positions in a counterclockwise order triangle strip

```
glBegin (GL_TRIANGLE_STRIP);
        glVertex2iv (p1);
        glVertex2iv (p2);
        glVertex2iv (p6);
        glVertex2iv (p3);
        glVertex2iv (p5);
        glVertex2iv (p4);
glEnd ( );
```

- Assuming that no coordinate positions are repeated in a list of N vertices, we obtain $N - 2$ triangles in the strip. Clearly, we must have $N \geq 3$ or nothing is displayed.
- Each successive triangle shares an edge with the previously defined triangle, so the ordering of the vertex list must be set up to ensure a consistent display.
- Example, our first triangle ($n = 1$) would be listed as having vertices (p1, p2, p6). The second triangle ($n = 2$) would have the vertex ordering (p6, p2, p3). Vertex ordering for the third triangle ($n = 3$) would be (p6, p3, p5). And the fourth triangle ($n = 4$) would be listed in the polygon tables with vertex ordering (p5, p3, p4).

Triangle Fan

- Another way to generate a set of connected triangles is to use the "fan" Approach

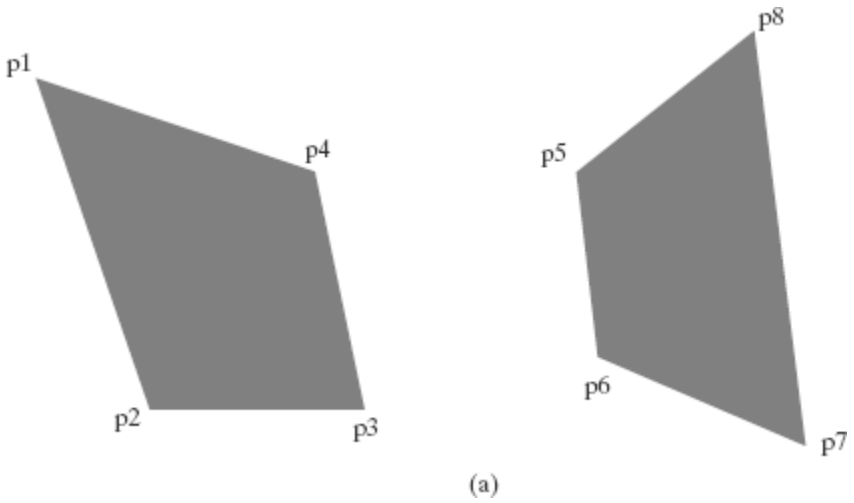
```
glBegin (GL_TRIANGLE_FAN);
        glVertex2iv (p1);
        glVertex2iv (p2);
        glVertex2iv (p3);
        glVertex2iv (p4);
        glVertex2iv (p5);
        glVertex2iv (p6);
glEnd ( );
```

- For *N* vertices, we again obtain *N*−2 triangles, providing no vertex positions are repeated, and we must list at least three vertices be specified in the proper order to define front and back faces for each triangle correctly.
- Therefore, triangle 1 is defined with the vertex list (p1, p2, p3); triangle 2 has the vertex ordering (p1, p3, p4); triangle 3 has its vertices specified in the order (p1, p4, p5); and triangle 4 is listed with vertices (p1, p5, p6).

Quadrilaterals

- ✓ OpenGL provides for the specifications of two types of quadrilaterals.
- ✓ With the GL QUADS primitive constant and the following list of eight vertices, specified as two-dimensional coordinate arrays, we can generate the display shown in Figure (a):

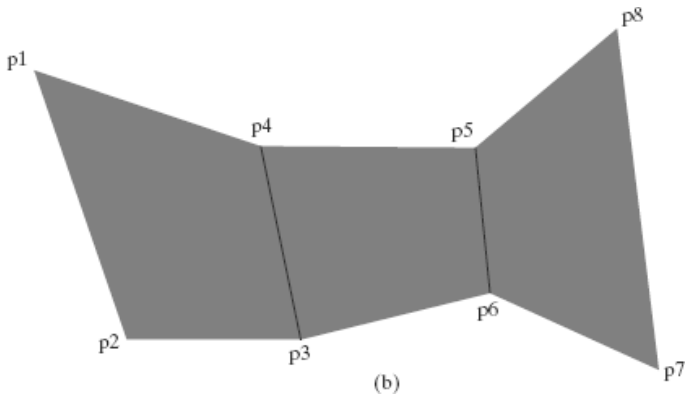
```
glBegin (GL_QUADS);
        glVertex2iv (p1);
        glVertex2iv (p2);
        glVertex2iv (p3);
        glVertex2iv (p4);
        glVertex2iv (p5);
        glVertex2iv (p6);
        glVertex2iv (p7);
        glVertex2iv (p8);
glEnd ( );
```



(a)

- ✓ Rearranging the vertex list in the previous quadrilateral code example and changing the primitive constant to GL QUAD STRIP, we can obtain the set of connected quadrilaterals shown in Figure (b):

```
glBegin (GL_QUAD_STRIP);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p4);
    glVertex2iv (p3);
    glVertex2iv (p5);
    glVertex2iv (p6);
    glVertex2iv (p8);
    glVertex2iv (p7);
glEnd ( );
```



(b)

- ✓ For a list of N vertices, we obtain $N/2-1$ quadrilaterals, providing that $N \geq 4$. Thus, our first quadrilateral ($n = 1$) is listed as having a vertex ordering of (p1, p2, p3, p4). The second quadrilateral (n=2) has the vertex ordering (p4, p3, p6, p5), and the vertex ordering for the third quadrilateral (n=3) is (p5, p6, p7, p8).

Fill-Area Attributes

➔ We can fill any specified regions, including circles, ellipses, and other objects with curved boundaries

Fill Styles

➔ A basic fill-area attribute provided by a general graphics library is the display style of the interior.

➔ We can display a region with a single color, a specified fill pattern, or in a "hollow" style by showing only the boundary of the region

➔ We can also fill selected regions of a scene using various brush styles, color-blending combinations, or textures.

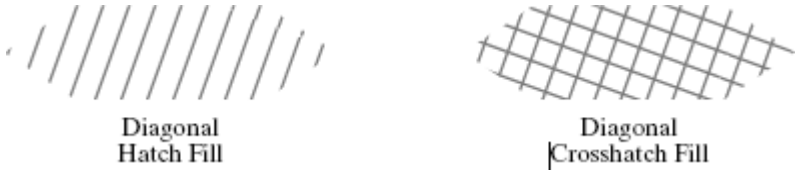
➔ For polygons, we could show the edges in different colors, widths, and styles; and we can select different display attributes for the front and back faces of a region.

➔ Fill patterns can be defined in rectangular color arrays that list different colors for different positions in the array.

➔ An array specifying a fill pattern is a *mask* that is to be applied to the display area.

➔ The mask is replicated in the horizontal and vertical directions until the display area is filled with nonoverlapping copies of the pattern.

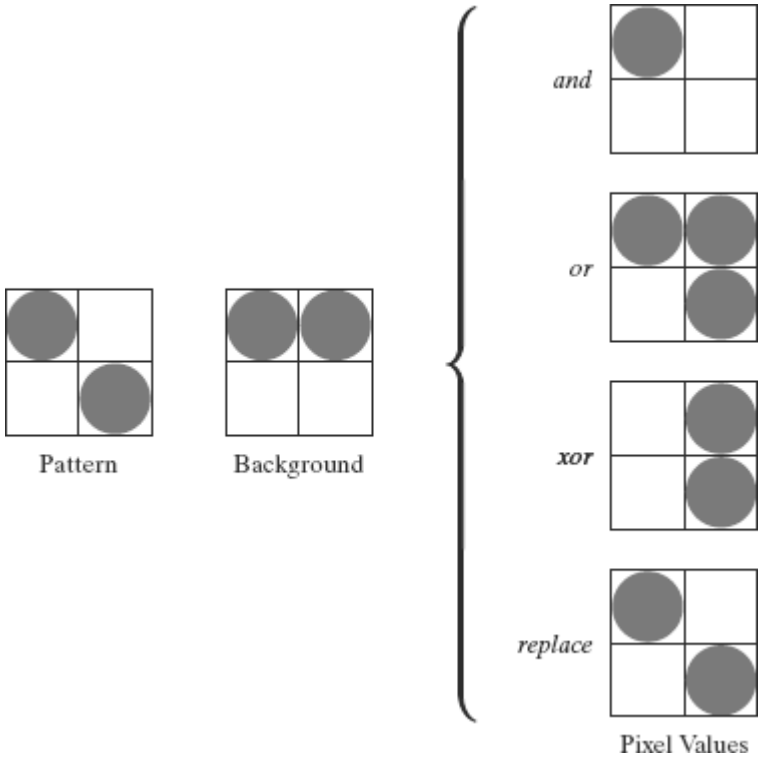
➔ This process of filling an area with a rectangular pattern is called tiling, and a rectangular fill pattern is sometimes referred to as a tiling pattern predefined fill patterns are available in a system, such as the *hatch* fill patterns



➔ Hatch fill could be applied to regions by drawing sets of line segments to display either single hatching or crosshtching

Color-Blended Fill Regions

- Color-blended regions can be implemented using either transparency factors to control the blending of background and object colors, or using simple logical or replace operations as shown in figure



- The *linear soft-fill algorithm* repaints an area that was originally painted by merging a foreground color F with a single background color B, where F != B.
- The current 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.
- For values of *t* less than 0.5, the background color contributes more to the interior color of the region than does the fill color.
- If our color values are represented using separate red, green, and blue (RGB) components, each component of the colors, with

  $P = (P_R, P_G, P_B)$, $F = (F_R, F_G, F_B)$, $B = (B_R, B_G, B_B)$ *is used*
- We can thus calculate the value of parameter *t* using one of the RGB color components as follows:

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

  Where *k* = *R*, *G*, or *B*; and *Fk* != *Bk* .
- When two background colors *B*1 and *B*2 are mixed with foreground color F, the resulting pixel color P is

$$P = t0F + t1B1 + (1 - t0 - t1)B2$$

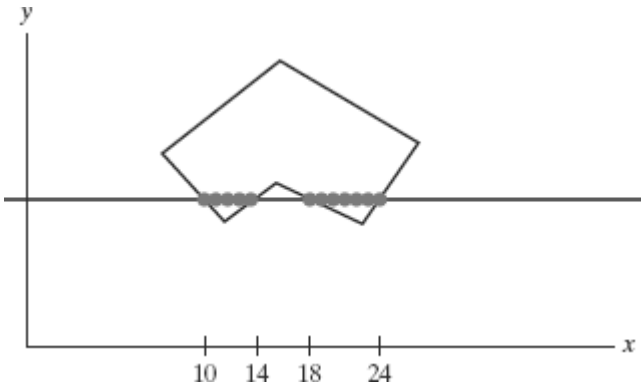
- Where the sum of the color-term coefficients *t*0, *t*1, and $(1 - t0 - t1)$ must equal 1.
- With three background colors and one foreground color, or with two background and two foreground colors, we need all three RGB equations to obtain the relative amounts of the four colors.

General Scan-Line Polygon-Fill Algorithm

➔ A scan-line fill of a region is performed by first determining the intersection positions of the boundaries of the fill region with the screen scan lines.

➔ Then the fill colors are applied to each section of a scan line that lies within the interior of the fill region.

➔ The simplest area to fill is a polygon because each scanline intersection point with a polygon boundary is obtained by solving a pair of simultaneous linear equations, where the equation for the scan line is simply y = constant.

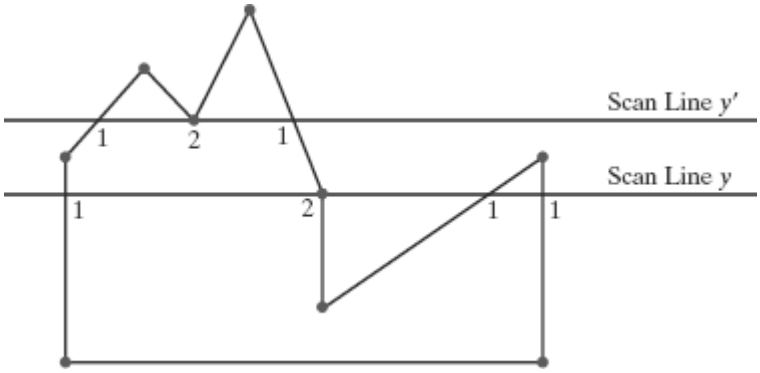


➔ Figure above illustrates the basic scan-line procedure for a solid-color fill of a polygon.

➔ For each scan line that crosses the polygon, the edge intersections are sorted from left to right, and then the pixel positions between, and including, each intersection pair are set to the specified fill color the fill color is applied to the five pixels from $x = 10$ to $x = 14$ and to the seven pixels from $x = 18$ to $x = 24$.

➔ Whenever a scan line passes through a vertex, it intersects two polygon edges at that point.

➔ In some cases, this can result in an odd number of boundary intersections for a scan line.



➔ Scan line *y*' intersects an even number of edges, and the two pairs of intersection points along this scan line correctly identify the interior pixel spans.

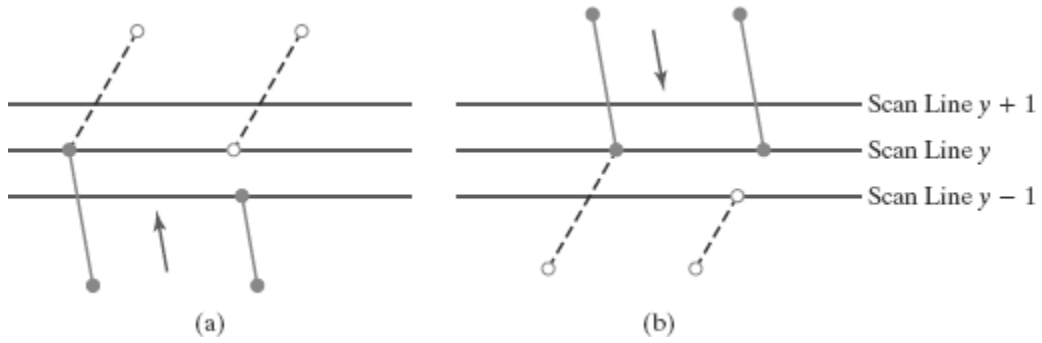
➔ But scan line *y* intersects five polygon edges.

➔ Thus, as we process scan lines, we need to distinguish between these cases.

➔ For scan line *y*, the two edges sharing an intersection vertex are on opposite sides of the scan line.

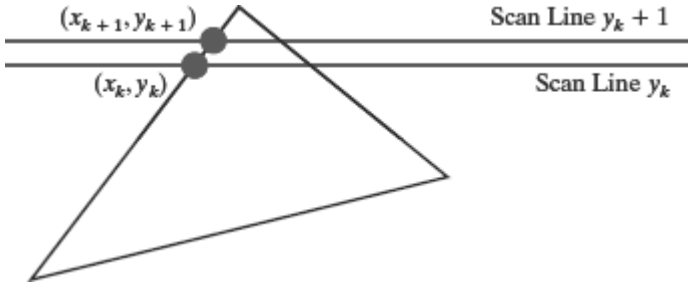
➔ But for scan line *y*', the two intersecting edges are both above the scan line.

- ➔ Thus, a vertex that has adjoining edges on opposite sides of an intersecting scan line should be counted as just one boundary intersection point.
- ➔ If the three endpoint *y* values of two consecutive edges monotonically increase or decrease, we need to count the shared (middle) vertex as a single intersection point for the scan line passing through that vertex.
- ➔ Otherwise, the shared vertex represents a local extremum (minimum or maximum) on the polygon boundary, and the two edge intersections with the scan line passing through that vertex can be added to the intersection list.
- ➔ One method for implementing the adjustment to the vertex-intersection count is to shorten some polygon edges to split those vertices that should be counted as one intersection.
- ➔ We can process nonhorizontal edges around the polygon boundary in the order specified, either clockwise or counterclockwise.
- ➔ Adjusting endpoint y values for a polygon, as we process edges in order around the polygon perimeter. The edge currently being processed is indicated as a solid line



In (a), the y coordinate of the upper endpoint of the current edge is decreased by 1. In (b), the y coordinate of the upper endpoint of the next edge is decreased by 1.

- ➔ Coherence properties can be used in computer-graphics algorithms to reduce processing.
- ➔ Coherence methods often involve incremental calculations applied along a single scan line or between successive scan lines



➔ The slope of this edge can be expressed in terms of the scan-line intersection coordinates:

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

➔ Because 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

$$x_k = x_0 + k/m$$

Where, 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

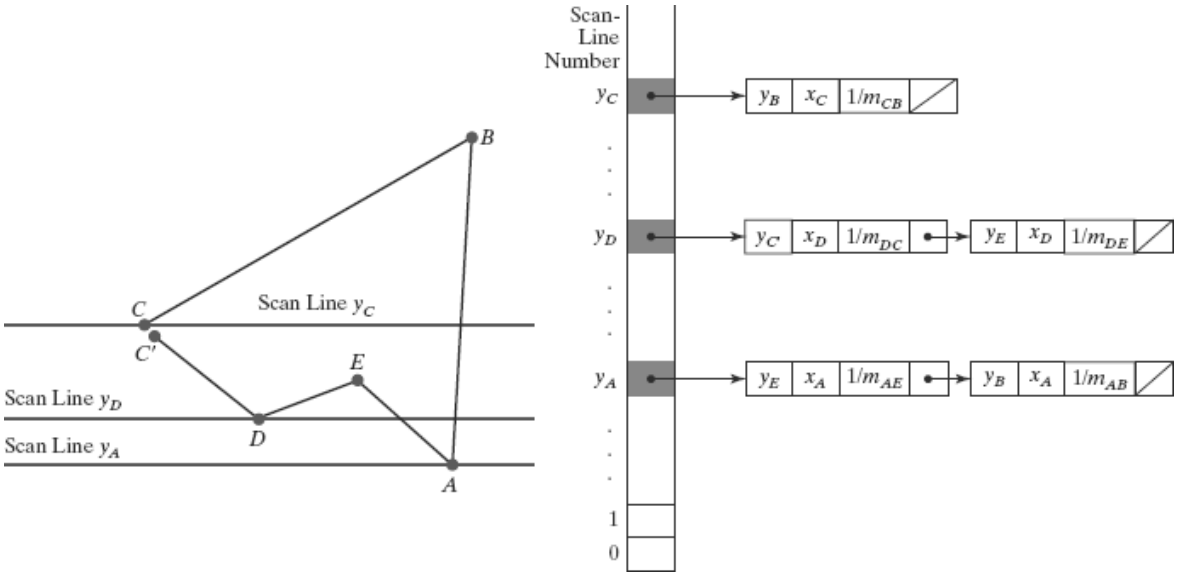
$$x_{k+1} = x_k + \frac{\Delta x}{\Delta y}$$

➔ To perform a polygon fill efficiently, we can first store the polygon boundary in a *sorted edge table* that contains all the information necessary to process the scan lines efficiently.

➔ Proceeding around the edges in either a clockwise or a counterclockwise order, we can use a bucket sort to store the edges, sorted on the smallest *y* value of each edge, in the correct scan-line positions.

➔ Only nonhorizontal edges are entered into the sorted edge table.

➔ Each entry in the table for a particular scan line contains the maximum *y* value for that edge, the *x*-intercept value (at the lower vertex) for the edge, and the inverse slope of the edge. For each scan line, the edges are in sorted order fromleft to right



➔ We process the scan lines from the bottom of the polygon to its top, producing an *active edge list* for each scan line crossing the polygon boundaries.

➔ The active edge list for a scan line contains all edges crossed by that scan line, with iterative coherence calculations used to obtain the edge intersections

➔ Implementation of edge-intersection calculations can be facilitated by storing Δx and Δy values in the sorted edge list

OpenGL Fill-Area Attribute Functions

➔ We generate displays of filled convex polygons in four steps:

1. Define a fill pattern.
2. Invoke the polygon-fill routine.

3. Activate the polygon-fill feature of OpenGL.
4. Describe the polygons to be filled.

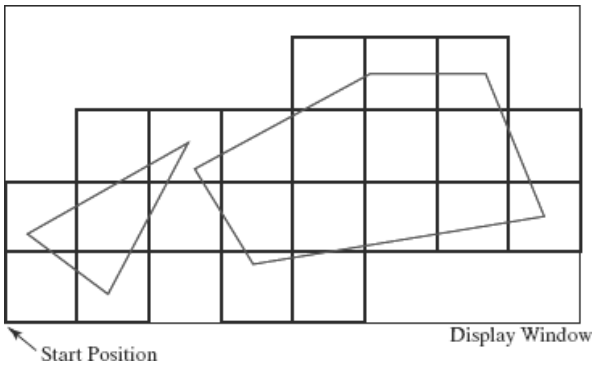
➔ A polygon fill pattern is displayed up to and including the polygon edges. Thus, there are no boundary lines around the fill region unless we specifically add them to the display

OpenGL Fill-Pattern Function

- To fill the polygon with a pattern in OpenGL, we use a 32 × 32 bit mask.
- A value of 1 in the mask indicates that the corresponding pixel is to be set to the current color, and a 0 leaves the value of that frame-buffer position unchanged.
- The fill pattern is specified in unsigned bytes using the OpenGL data type Glubyte

GLubyte fillPattern [ ] = { 0xff, 0x00, 0xff, 0x00, ... };

- The bits must be specified starting with the bottom row of the pattern, and continuing up to the topmost row (32) of the pattern.
- This pattern is replicated across the entire area of the display window, starting at the lower-left window corner, and specified polygons are filled where the pattern overlaps those polygons



- Once we have set a mask, we can establish it as the current fill pattern with the function

glPolygonStipple (fillPattern);

- We need to enable the fill routines before we specify the vertices for the polygons that are to be filled with the current pattern

glEnable (GL_POLYGON_STIPPLE);

- Similarly, we turn off pattern filling with

glDisable (GL_POLYGON_STIPPLE);

OpenGL Texture and Interpolation Patterns

- Another method for filling polygons is to use texture patterns.
- This can produce fill patterns that simulate the surface appearance of wood, brick, brushed steel, or some other material.
- We assign different colors to polygon vertices.
- Interpolation fill of a polygon interior is used to produce realistic displays of shaded surfaces under various lighting conditions.
- The polygon fill is then a linear interpolation of the colors at the vertices:

```
glShadeModel (GL_SMOOTH);
glBegin (GL_TRIANGLES);
        glColor3f (0.0, 0.0, 1.0);
        glVertex2i (50, 50);
        glColor3f (1.0, 0.0, 0.0);
        glVertex2i (150, 50);
        glColor3f (0.0, 1.0, 0.0);
        glVertex2i (75, 150);
glEnd ( );
```

OpenGL Wire-Frame Methods

➔ We can also choose to show only polygon edges. This produces a wire-frame or hollow display of the polygon; or we could display a polygon by plotting a set of points only at the vertex positions.

➔ These options are selected with the function

glPolygonMode (face, displayMode);

➔ We use parameter face to designate which face of the polygon that we want to show as edges only or vertices only.

➔ This parameter is then assigned either

GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK.

➔ If we want only the polygon edges displayed for our selection, we assign the constant GL_LINE to parameter displayMode.

➔ To plot only the polygon vertex points, we assign the constant GL_POINT to parameter displayMode.

➔ Another option is to display a polygon with both an interior fill and a different color or pattern for its edges.

➔ The following code section fills a polygon interior with a green color, and then the edges are assigned a red color:

```
glColor3f (0.0, 1.0, 0.0);
/* Invoke polygon-generating routine. */
glColor3f (1.0, 0.0, 0.0);
glPolygonMode (GL_FRONT, GL_LINE);
/* Invoke polygon-generating routine again. */
```

➔ For a three-dimensional polygon (one that does not have all vertices in the *xy* plane), this method for displaying the edges of a filled polygon may produce gaps along the edges.

➔ This effect, sometimes referred to as **stitching.**

➔ One way to eliminate the gaps along displayed edges of a three-dimensional polygon is to shift the depth values calculated by the fill routine so that they do not overlap with the edge depth values for that polygon.

➔ We do this with the following two OpenGL functions:

```
glEnable (GL_POLYGON_OFFSET_FILL);
glPolygonOffset (factor1, factor2);
```

➔ The first function activates the offset routine for scan-line filling, and the second function is used to set a couple of floating-point values factor1 and factor2 that are used to calculate the amount of depth offset.

➔ The calculation for this depth offset is

$$\textbf{depthOffset} = \textbf{factor1} \cdot \textbf{maxSlope} + \textbf{factor2} \cdot \textbf{const}$$

Where,

maxSlope is the maximum slope of the polygon and

const is an implementation constant

➔ As an example of assigning values to offset factors, we can modify the previous code segment as follows:

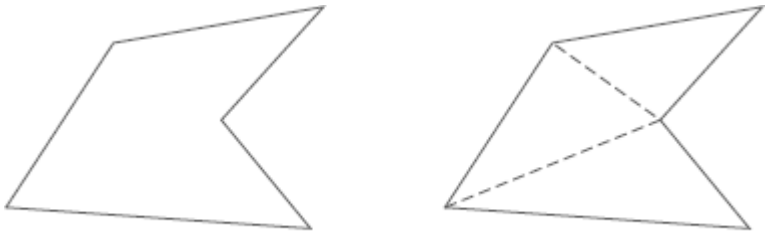
```
glColor3f (0.0, 1.0, 0.0);
```

```
glEnable (GL_POLYGON_OFFSET_FILL);
glPolygonOffset (1.0, 1.0);
/* Invoke polygon-generating routine. */
glDisable (GL_POLYGON_OFFSET_FILL);
glColor3f (1.0, 0.0, 0.0);
glPolygonMode (GL_FRONT, GL_LINE);
/* Invoke polygon-generating routine again. */
```

➔ Another method for eliminating the stitching effect along polygon edges is to use the OpenGL stencil buffer to limit the polygon interior filling so that it does not overlap the edges.

➔ To display a concave polygon using OpenGL routines, we must first split it into a set of convex polygons.

➔ We typically divide a concave polygon into a set of triangles. Then we could display the triangles.



Dividing a concave polygon (a) into a set of triangles (b) produces triangle edges (dashed) that are interior to the original polygon.

➔ Fortunately, OpenGL provides a mechanism that allows us to eliminate selected edges from a wire-frame display.

➔ So all we need do is set that bit flag to "off" and the edge following that vertex will not be displayed.

➔ We set this flag for an edge with the following function:

```
glEdgeFlag (flag)
```

➔ To indicate that a vertex does not precede a boundary edge, we assign the OpenGL constant GL_FALSE to parameter flag.

➔ This applies to all subsequently specified vertices until the next call to glEdgeFlag is made.

➔ The OpenGL constant GL_TRUE turns the edge flag on again, which is the default.

➔ As an illustration of the use of an edge flag, the following code displays only two edges of the defined triangle

```
glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);
glBegin (GL_POLYGON);
    glVertex3fv (v1);
    glEdgeFlag (GL_FALSE);
    glVertex3fv (v2);
    glEdgeFlag (GL_TRUE);
    glVertex3fv (v3);
glEnd ( );
```

➔ Polygon edge flags can also be specified in an array that could be combined or associated with a vertex array.

➔ The statements for creating an array of edge flags are

```
glEnableClientState (GL_EDGE_FLAG_ARRAY);
glEdgeFlagPointer (offset, edgeFlagArray);
```

OpenGL Front-Face Function

- We can label selected surfaces in a scene independently as front or back with the function

```
glFrontFace (vertexOrder);
```

- If we set parameter vertexOrder to the OpenGL constant GL_CW, then a subsequently defined polygon with a clockwise ordering.
- The constant GL_CCW labels a counterclockwise ordering of polygon vertices as front-facing, which is the default ordering.r its vertices is considered to be front-facing

2DGeometric Transformations:

Basic 2D Geometric Transformations,

Matrix representations and homogeneouscoordinates.

Inverse transformations,

2DComposite transformations,

Other 2D transformations,

Raster methods for geometric transformations,

OpenGL raster transformations

OpenGL geometric transformations function,

Two-Dimensional Geometric Transformations

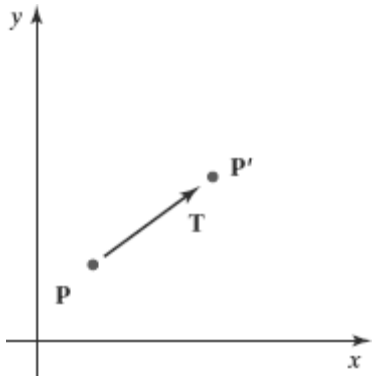
Operations that are applied to the geometric description of an object to change its position, orientation, or size are called **geometric transformations.**

Basic Two-Dimensional Geometric Transformations

The geometric-transformation functions that are available in all graphics packages are those for translation, rotation, and scaling.

Two-Dimensional Translation

- We perform a **translation** on a single coordinate point by adding offsets to its coordinates so as to generate a new coordinate position.
- We are moving the original point position along a straight-line path to its new location.
- To translate a two-dimensional position, we add **translation distances** *tx* and *ty* to the original coordinates (*x*, *y*) to obtain the new coordinate position (*x*', *y*') as shown in Figure



- The translation values of x' and y' is calculated as

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

- The translation distance pair (t_x, t_y) is called a **translation vector** or **shift vector Column vector representation is given as**

$$\mathrm{P} = \begin{bmatrix} x \\ y \end{bmatrix}, \qquad \mathrm{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \qquad \mathrm{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

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

$$\mathrm{P}' = \mathrm{P} + \mathrm{T}$$

- Translation is a *rigid-body transformation* that moves objects without deformation.

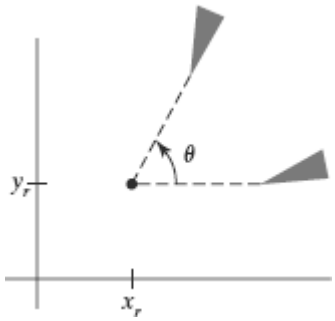
Code:

```
class wcPt2D {
        public:
        GLfloat x, y;
};
void translatePolygon (wcPt2D * verts, GLint nVerts, GLfloat tx, GLfloat ty)
{
    GLint k;
    for (k = 0; k < nVerts; k++) {
        verts [k].x = verts [k].x + tx;
        verts [k].y = verts [k].y + ty;
    }
    glBegin (GL_POLYGON);
    for (k = 0; k < nVerts; k++)
        glVertex2f (verts [k].x, verts [k].y);
    glEnd ( );
}
```

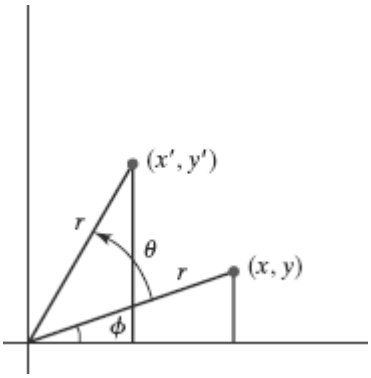
Two-Dimensional Rotation

- ✓ We generate a **rotation** transformation of an object by specifying a **rotation axis** and a **rotation angle.**

- ✓ A two-dimensional rotation of an object is obtained by repositioning the object along a circular path in the *xy* plane.
- ✓ In this case, we are rotating the object about a rotation axis that is perpendicular to the *xy* plane (parallel to the coordinate *z* axis)**.**
- ✓ Parameters for the two-dimensional rotation are the rotation angle θ and a position (*xr*, *yr*), called the **rotation point** (or **pivot point**), about which the object is to be rotated



- ✓ A positive value for the angle θ defines a counterclockwise rotation about the pivot point, as in above Figure , and a negative value rotates objects in the clockwise direction.
- ✓ The angular and coordinate relationships of the original and transformed point positions are shown in Figure



- ✓ 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.
- ✓ 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

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

- ✓ Substituting expressions of x and y in the eaquations of x' and y' we get

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

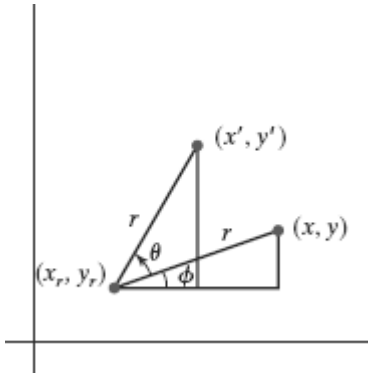
- ✓ We can write the rotation equations in the matrix form

$$\mathbf{P'} = \mathbf{R} \cdot \mathbf{P}$$

Where the rotation matrix is,

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

- ✓ Rotation of a point about an arbitrary pivot position is illustrated in Figure



- ✓ The transformation equations for rotation of a point about any specified rotation position (*xr* , *yr*):

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

Code:

```
class wcPt2D {
        public:
        GLfloat x, y;
};
void rotatePolygon (wcPt2D * verts, GLint nVerts, wcPt2D pivPt, GLdouble theta)
{
    wcPt2D * vertsRot;
    GLint k;
    for (k = 0; k < nVerts; k++) {
```

```
            vertsRot [k].x = pivPt.x + (verts [k].x - pivPt.x) * cos (theta) - (verts [k].y -
            pivPt.y) * sin (theta);
            vertsRot [k].y = pivPt.y + (verts [k].x - pivPt.x) * sin (theta) + (verts [k].y -
            pivPt.y) * cos (theta);
        }
        glBegin (GL_POLYGON);
        for (k = 0; k < nVerts; k++)
            glVertex2f (vertsRot [k].x, vertsRot [k].y);
        glEnd ( );
}
```

Two-Dimensional Scaling

- ✓ To alter the size of an object, we apply a **scaling** transformation.
- ✓ A simple twodimensional scaling operation is performed by multiplying object positions (x, y) by **scaling factors** sx and sy to produce the transformed coordinates (x', y'):

$$x' = x \cdot s_{x'} \qquad y' = y \cdot s_y$$

- ✓ The basic two-dimensional scaling equations can also be written in the following matrix form

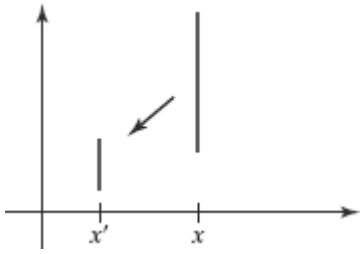
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P}$$

Where **S** is the 2 × 2 scaling matrix

- ✓ Any positive values can be assigned to the scaling factors sx and sy.
- ✓ Values less than 1 reduce the size of objects
- ✓ Values greater than 1 produce enlargements.
- ✓ Specifying a value of 1 for both sx and sy leaves the size of objects unchanged.
- ✓ When sx and sy are assigned the same value, a **uniform scaling** is produced, which maintains relative object proportions.

- ✓ Unequal values for *sx* and *sy* result in a **differential scaling** that is often used in design applications.
- ✓ In some systems, negative values can also be specified for the scaling parameters. This not only resizes an object, it reflects it about one or more of the coordinate axes.
- ✓ Figure below illustrates scaling of a line by assigning the value 0.5 to both *sx* and *sy*



- ✓ 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* , *yf*), are often chosen at some object position, such as its centroid but any other spatial position can be selected.
- ✓ For a coordinate position (*x, y*), the scaled coordinates (*x', y'*) are then calculated from the following relationships:

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

- ✓ We can rewrite Equations to separate the multiplicative and additive terms as

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

- ✓ Where the additive terms *x f (*1 − *sx)* and *yf (*1 − *sy)* are constants for all points in the object.

Code:

```
class wcPt2D {
        public:
        GLfloat x, y;
};
void scalePolygon (wcPt2D * verts, GLint nVerts, wcPt2D fixedPt, GLfloat sx, GLfloat sy)
{
    wcPt2D vertsNew;
```

```
        GLint k;
        for (k = 0; k < nVerts; k++) {
                vertsNew [k].x = verts [k].x * sx + fixedPt.x * (1 - sx);
                vertsNew [k].y = verts [k].y * sy + fixedPt.y * (1 - sy);
        }
        glBegin (GL_POLYGON);
        for (k = 0; k < nVerts; k++)
                glVertex2f (vertsNew [k].x, vertsNew [k].y);
        glEnd ( );
}
```

Matrix Representations and Homogeneous Coordinates

- ✓ Each of the three basic two-dimensional transformations (translation, rotation, and scaling) can be expressed in the general matrix form

$$P' = M_1 \cdot P + M_2$$

- ✓ With coordinate positions **P** and **P**' represented as column vectors.
- ✓ Matrix **M**1 is a 2 × 2 array containing multiplicative factors, and **M**2 is a two-element column matrix containing translational terms.
- ✓ For translation, **M**1 is the identity matrix.
- ✓ For rotation or scaling, **M**2 contains the translational terms associated with the pivot point or scaling fixed point.

Homogeneous Coordinates

- ➢ Multiplicative and translational terms for a two-dimensional geometric transformation can be combined into a single matrix if we expand the representations to 3 × 3 matrices
- ➢ We can use the third column of a transformation matrix for the translation terms, and all transformation equations can be expressed as matrix multiplications.
- ➢ We also need to expand the matrix representation for a two-dimensional coordinate position to a three-element column matrix

- A standard technique for accomplishing this is to expand each twodimensional coordinate-position representation (*x*, *y*) to a three-element representation (*xh*, *yh*, *h*), called **homogeneous coordinates,** where the **homogeneous parameter** *h* is a nonzero value such that

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

- A general two-dimensional homogeneous coordinate representation could also be written as ($h \cdot x$, $h \cdot y$, h).
- 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.

Two-Dimensional Translation Matrix

- ✓ The homogeneous-coordinate for translation is given by

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

- ✓ This translation operation can be written in the abbreviated form

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

with **T**(tx, ty) as the 3 × 3 translation matrix

Two-Dimensional Rotation Matrix

- ✓ Two-dimensional rotation transformation equations about the coordinate origin can be expressed in the matrix form

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

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

- ✓ The rotation transformation operator **R**(θ) is the 3 × 3 matrix with rotation parameter θ.

Two-Dimensional Scaling Matrix

- ✓ A scaling transformation relative to the coordinate origin can now be expressed as the matrix multiplication

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

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

- ✓ The scaling operator S(sx, sy) is the 3 × 3 matrix with parameters sx and sy

Inverse Transformations

- ❖ For translation,we obtain the inverse matrix by negating the translation distances. Thus, if we have two-dimensional translation distances *tx* and *ty*, the inverse translation matrix is

$$\mathbf{T}^{-1} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix}$$

- ❖ An inverse rotation is accomplished by replacing the rotation angle by its negative.
- ❖ A two-dimensional rotation through an angle θ about the coordinate origin has the inverse transformation matrix

$$\mathbf{R}^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- ❖ We form the inverse matrix for any scaling transformation by replacing the scaling parameters with their reciprocals. the inverse transformation matrix is

$$\mathbf{S}^{-1} = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Two-Dimensional Composite Transformations

- ✓ Forming products of transformation matrices is often referred to as a **concatenation,** or **composition,** of matrices if we want to apply two transformations to point position **P**, the transformed location would be calculated as

$$\begin{aligned} \mathbf{P}' &= \mathbf{M}_2 \cdot \mathbf{M}_1 \cdot \mathbf{P} \\ &= \mathbf{M} \cdot \mathbf{P} \end{aligned}$$

- ✓ The coordinate position is transformed using the composite matrix**M**, rather than applying the individual transformations **M**1 and then**M**2.

Composite Two-Dimensional Translations

- ✓ If two successive translation vectors ($t1x$, $t1y$) and ($t2x$, $t2y$) are applied to a twodimensional coordinate position **P**, the final transformed location **P'** is calculated as

$$\begin{aligned} \mathbf{P}' &= \mathbf{T}(t_{2x}, t_{2y}) \cdot \{\mathbf{T}(t_{1x}, t_{1y}) \cdot \mathbf{P}\} \\ &= \{\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y})\} \cdot \mathbf{P} \end{aligned}$$

where **P** and **P'** are represented as three-element, homogeneous-coordinate column vectors

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

$$\begin{bmatrix} 1 & 0 & t_{2x} \\ 0 & 1 & t_{2y} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{1x} \\ 0 & 1 & t_{1y} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{1x} + t_{2x} \\ 0 & 1 & t_{1y} + t_{2y} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y}) = \mathbf{T}(t_{1x} + t_{2x}, t_{1y} + t_{2y})$$

Composite Two-Dimensional Rotations

- ✓ Two successive rotations applied to a point **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 of a point can be calculated with the composite rotation matrix as

$$\mathbf{P'} = \mathbf{R}(\theta 1 + \theta 2) \cdot \mathbf{P}$$

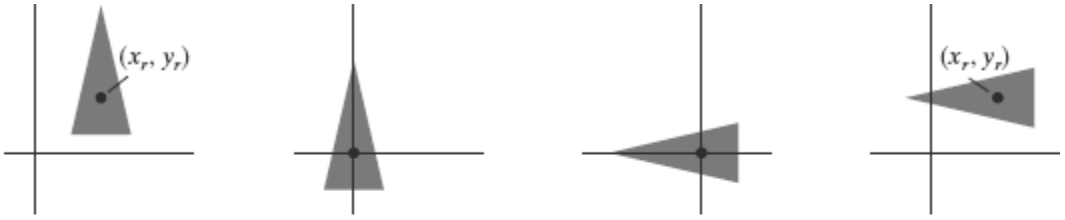
Composite Two-Dimensional Scalings

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

$$\begin{bmatrix} s_{2x} & 0 & 0 \\ 0 & s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{1x} & 0 & 0 \\ 0 & s_{1y} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{1x} \cdot s_{2x} & 0 & 0 \\ 0 & s_{1y} \cdot s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{S}(s_{2x}, s_{2y}) \cdot \mathbf{S}(s_{1x}, s_{1y}) = \mathbf{S}(s_{1x} \cdot s_{2x}, \quad s_{1y} \cdot s_{2y})$$

General Two-Dimensional Pivot-Point Rotation



(a) Original Position of Object and Pivot Point

(b) Translation of Object so that Pivot Point (x_r, y_r) is at Origin

(c) Rotation about Origin

(d) Translation of Object so that the Pivot Point is Returned to Position (x_r, y_r)

- ✓ We can generate a two-dimensional rotation about any other pivot point (*xr , yr*) 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.

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

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

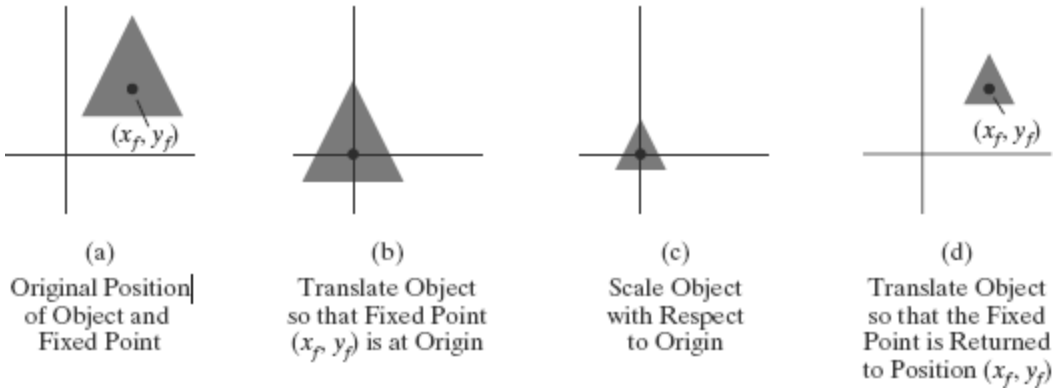
$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r \sin\theta \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r \sin\theta \\ 0 & 0 & 1 \end{bmatrix}$$

which can be expressed in the form

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

where $\mathbf{T}(-xr, -yr) = \mathbf{T}^{-1}(xr, yr)$.

General Two-Dimensional Fixed-Point Scaling



(a) Original Position of Object and Fixed Point

(b) Translate Object so that Fixed Point (x_f, y_f) is at Origin

(c) Scale Object with Respect to Origin

(d) Translate Object so that the Fixed Point is Returned to Position (x_f, y_f)

- ✓ To produce a two-dimensional scaling with respect to a selected fixed position ($x f$, yf), when we have a function that can scale relative to the coordinate origin only. This sequence is

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

  2. Scale the object with respect to the coordinate origin.

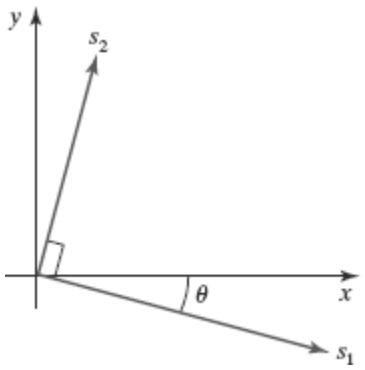
  3. Use the inverse of the translation in step (1) to return the object to its original position.

- ✓ Concatenating the matrices for these three operations produces the required scaling matrix: $\begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_f(1-s_x) \\ 0 & s_y & y_f(1-s_y) \\ 0 & 0 & 1 \end{bmatrix}$

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

General Two-Dimensional Scaling Directions

- ✓ Parameters *sx* and *sy* 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.
- ✓ Suppose we want to apply scaling factors with values specified by parameters *s*1 and *s*2 in the directions shown in Figure



- ✓ The composite matrix resulting from the product of these three transformations is

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

Matrix Concatenation Properties

Property 1:

- ✓ Multiplication of matrices is associative.
- ✓ For any three matrices,**M**1,**M**2, and**M**3, the matrix product **M**3 · **M**2 · **M**1 can be performed by first multiplying **M**3 and **M**2 or by first multiplying**M**2 and **M**1:

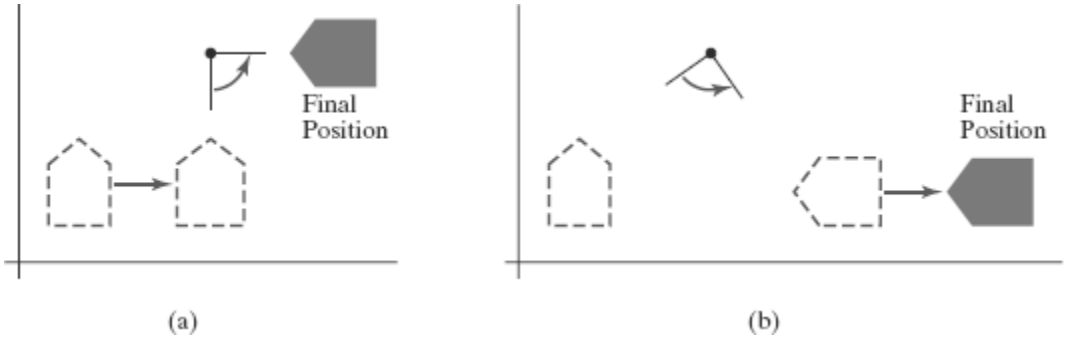
  M3 ·**M**2 · **M**1 = *(***M**3 · **M**2*)* ·**M**1 = **M**3 · *(***M**2 · **M**1*)*

- ✓ We can construct a composite matrix either by multiplying from left to right (premultiplying) or by multiplying from right to left (postmultiplying)

Property 2:

- ✓ Transformation products, on the other hand, may not be commutative. The matrix product**M**2 · **M**1 is not equal to**M**1 · **M**2,in general.

- ✓ This means that if we want to translate and rotate an object, we must be careful about the order in which the composite matrix is evaluated



- ✓ Reversing the order in which a sequence of transformations is performed may affect the transformed position of an object. In (a), an object is first translated in the x direction, then rotated counterclockwise through an angle of 45◦. In (b), the object is first rotated 45◦ counterclockwise, then translated in the x direction.

General Two-Dimensional Composite Transformations and Computational Efficiency

- ✓ A two-dimensional transformation, representing any 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 *rsjk* are the multiplicative rotation-scaling terms in the transformation, which involve only rotation angles and scaling factors if an object is to be scaled and rotated about its centroid coordinates (*xc* , *yc*) and then translated, the values for the elements of the composite transformation matrix are

$$\mathbf{T}(t_x, t_y) \cdot \mathbf{R}(x_c, y_c, \theta) \cdot \mathbf{S}(x_c, y_c, s_x, s_y)$$

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

- ✓ Although the above matrix requires nine multiplications and six additions, the explicit calculations for the transformed coordinates are

$$x' = x \cdot rs_{xx} + y \cdot rs_{xy} + trs_x, \qquad y' = x \cdot rs_{yx} + y \cdot rs_{yy} + trs_y$$

- ✓ We need actually perform only four multiplications and four additions to transform coordinate positions.
- ✓ Because rotation calculations require trigonometric evaluations and several multiplications for each transformed point, computational efficiency can become an important consideration in rotation transformations
- ✓ If we are rotating in small angular steps about the origin, for instance, we can set cos θ to 1.0 and reduce transformation calculations at each step to two multiplications and two additions for each set of coordinates to be rotated.
- ✓ These rotation calculations are

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

Two-Dimensional Rigid-Body Transformation

➔ If a transformation matrix includes only translation and rotation parameters, it is a **rigid-body transformation matrix.**

➔ The general form for a two-dimensional rigid-body transformation matrix is

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

where the four elements *r jk* are the multiplicative rotation terms, and the elements *trx* and *try* are the translational terms

➔ A rigid-body change in coordinate position is also sometimes referred to as a **rigid-motion** transformation.

➔ In addition, the above matrix has the property that its upper-left 2 × 2 submatrix is an *orthogonal matrix*.

➔ If we consider each row (or each column) of the submatrix as a vector, then the two row vectors (*rxx, rxy*) and (*ryx, ryy*) (or the two column vectors) form an orthogonal set of unit vectors.

➔ Such a set of vectors is also referred to as an *orthonormal* vector set. Each vector has unit length as follows

$$r_{xx}^2 + r_{xy}^2 = r_{yx}^2 + r_{yy}^2 = 1$$

and the vectors are perpendicular (their dot product is 0):

$$r_{xx}r_{yx} + r_{xy}r_{yy} = 0$$

➔ Therefore, if these unit vectors are transformed by the rotation submatrix, then the vector (*rxx*, *rxy*) is converted to a unit vector along the *x* axis and the vector (*ryx*, *ryy*) is transformed into a unit vector along the *y* axis of the coordinate system

$$\begin{bmatrix} r_{xx} & r_{xy} & 0 \\ r_{yx} & r_{yy} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{xx} \\ r_{xy} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} r_{xx} & r_{xy} & 0 \\ r_{yx} & r_{yy} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{yx} \\ r_{yy} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

➔ For example, the following rigid-body transformation first rotates an object through an angle θ about a pivot point (*xr* , *yr*) and then translates the object

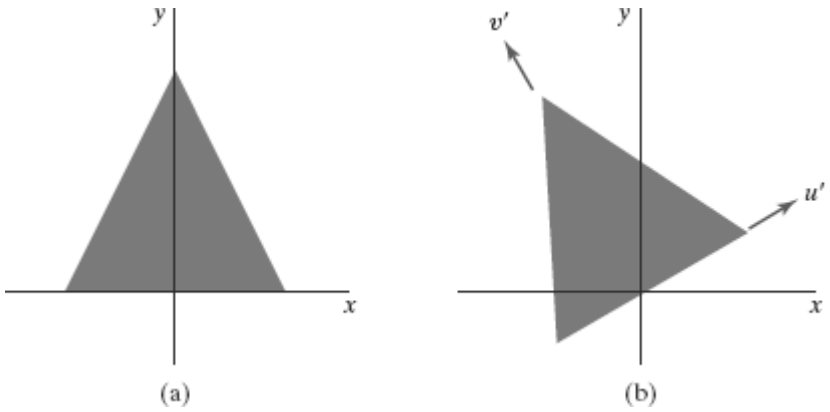
$$\mathbf{T}(t_x, t_y) \cdot \mathbf{R}(x_r, y_r, \theta) = \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r \sin\theta + t_x \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r \sin\theta + t_y \\ 0 & 0 & 1 \end{bmatrix}$$

➔ Here, orthogonal unit vectors in the upper-left 2×2 submatrix are (cos θ, −sin θ) and (sin θ, cos θ).

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

Constructing Two-Dimensional Rotation Matrices

- ✓ The orthogonal property of rotation matrices is useful for constructing the 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.
- ✓ We might want to rotate an object to align its axis of symmetry with the viewing (camera) direction, or we might want to rotate one object so that it is above another object.
- ✓ Figure shows an object that is to be aligned with the unit direction vectors **u**_ and **v**



The rotation matrix for revolving an object from position (a) to position (b) can be constructed with the values of the unit orientation vectors u' and v' relative to the original orientation.

Other Two-Dimensional Transformations

Two such transformations

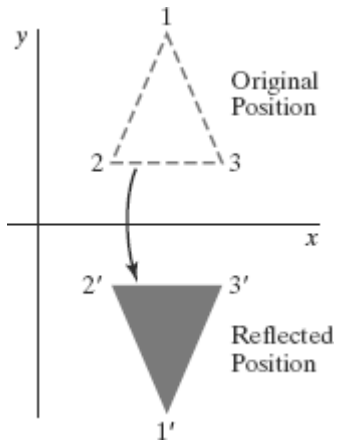
1. Reflection and
2. Shear.

Reflection

- ✓ A transformation that produces a mirror image of an object is called a **reflection.**
- ✓ For a two-dimensional reflection, this image is generated relative to an **axis of reflection** by rotating the object 180◦ about the reflection axis**.**
- ✓ Reflection about the line $y = 0$ (the x axis) is accomplished with the transformation Matrix

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

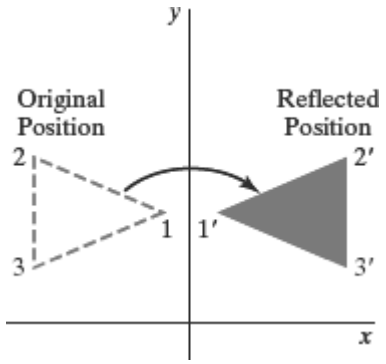
- ✓ This transformation retains x values, but "flips" the y values of coordinate positions.
- ✓ The resulting orientation of an object after it has been reflected about the x axis is shown in Figure



- ✓ A reflection about the line $x = 0$ (the y axis) flips x coordinates while keeping y coordinates the same. The matrix for this transformation is

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

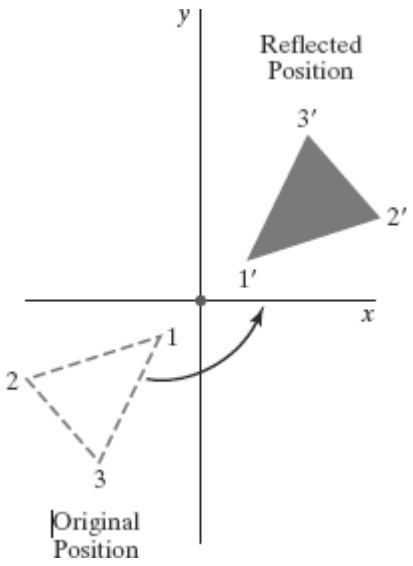
- ✓ Figure below illustrates the change in position of an object that has been reflected about the line $x = 0$.



- ✓ 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 the matrix representation for this reflection is

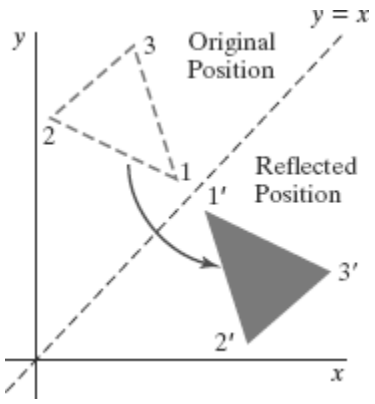
$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- ✓ An example of reflection about the origin is shown in Figure



✓ If we choose the reflection axis as the diagonal line $y = x$ (Figure below), 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

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

Shear

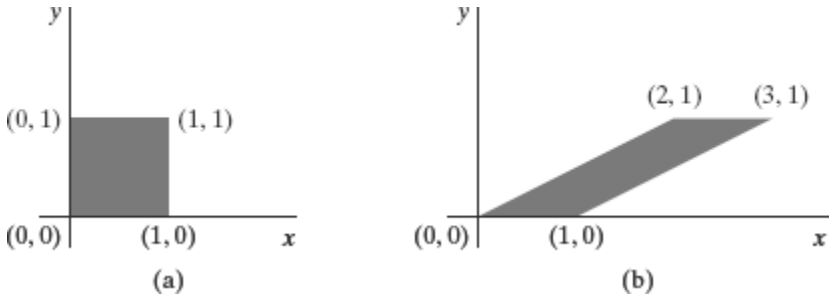
- ✓ A transformation that distorts the shape of an object such that the transformed shape appears as if the object were composed of internal layers that had been caused to slide over each other is called a **shear.**
- ✓ Two common shearing transformations are those that shift coordinate *x* values and those that shift *y* values. 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

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

- ✓ Any real number can be assigned to the shear parameter *shx* Setting parameter *shx* to the value 2, for example, changes the square into a parallelogram is shown below. Negative values for *shx* shift coordinate positions to the left.



A unit square (a) is converted to a parallelogram (b) using the x -direction shear with shx = 2.

- ✓ We can generate *x*-direction shears relative to other reference lines with

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

Now, coordinate positions are transformed as

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

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

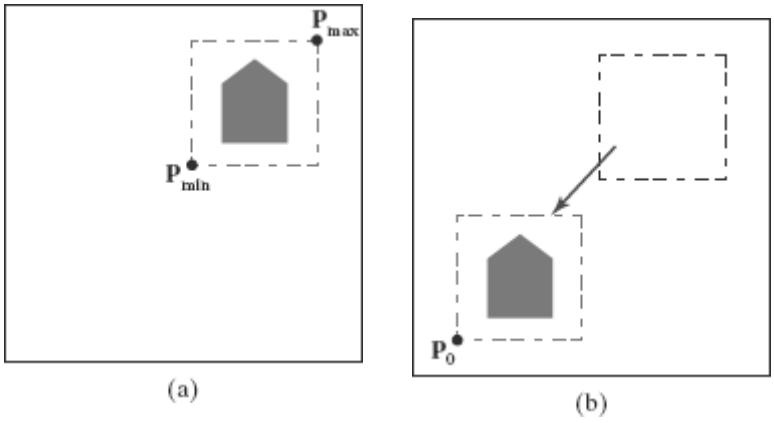
$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & -sh_y \cdot x_{ref} \\ 0 & 0 & 1 \end{bmatrix}$$

which generates the transformed coordinate values

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

Raster Methods for Geometric Transformations

- ✓ Raster systems store picture information as color patterns in the frame buffer.
- ✓ Therefore, some simple object transformations can be carried out rapidly by manipulating an array of pixel values
- ✓ Few arithmetic operations are needed, so the pixel transformations are particularly efficient.
- ✓ Functions that manipulate rectangular pixel arrays are called *raster operations* and moving a block of pixel values from one position to another is termed a *block transfer*, a *bitblt*, or a *pixblt*.
- ✓ Figure below illustrates a two-dimensional translation implemented as a block transfer of a refresh-buffer area



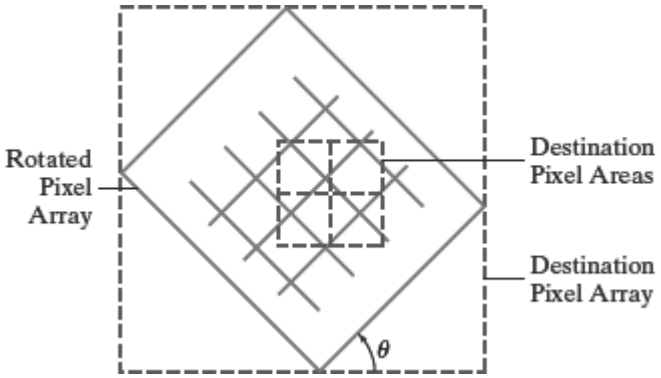
Translating an object from screen position (a) to the destination position shown in (b) by moving a rectangular block of pixel values. Coordinate positions Pmin and Pmax specify the limits of the rectangular block to be moved, and P0 is the destination reference position.

- ✓ Rotations in 90-degree increments are accomplished easily by rearranging the elements of a pixel array.
- ✓ We can rotate a two-dimensional object or pattern 90◦ counterclockwise by reversing the pixel values in each row of the array, then interchanging rows and columns.
- ✓ A 180◦ rotation is obtained by reversing the order of the elements in each row of the array, then reversing the order of the rows.
- ✓ Figure below demonstrates the array manipulations that can be used to rotate a pixel block by 90◦ and by 180◦.

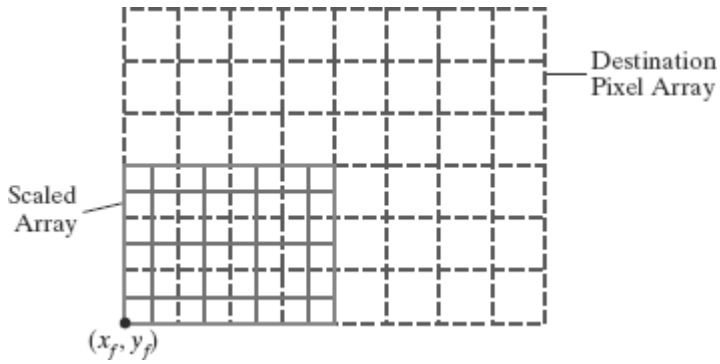
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \quad \begin{bmatrix} 3 & 6 & 9 & 12 \\ 2 & 5 & 8 & 11 \\ 1 & 4 & 7 & 10 \end{bmatrix} \quad \begin{bmatrix} 12 & 11 & 10 \\ 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

(a) (b) (c)

- ✓ For array rotations that are not multiples of 90◦, we need to do some extra processing.
- ✓ The general procedure is illustrated in Figure below.



- ✓ Each destination pixel area is mapped onto the rotated array and the amount of overlap with the rotated pixel areas is calculated.
- ✓ A color for a destination pixel can then be computed by averaging the colors of the overlapped source pixels, weighted by their percentage of area overlap.
- ✓ Pixel areas in the original block are scaled, using specified values for *sx* and *sy*, and then mapped onto a set of destination pixels.
- ✓ The color of each destination pixel is then assigned according to its area of overlap with the scaled pixel areas



OpenGL Raster Transformations

- A translation of a rectangular array of pixel-color values from one buffer area to another can be accomplished in OpenGL as the following copy operation:

  glCopyPixels (xmin, ymin, width, height, GL_COLOR);

- The first four parameters in this function give the location and dimensions of the pixel block; and the OpenGL symbolic constant **GL_COLOR** specifies that it is color values are to be copied.

- A block of RGB color values in a buffer can be saved in an array with the function

glReadPixels (xmin, ymin, width, height, GL_RGB, GL_UNSIGNED_BYTE, colorArray);

- If color-table indices are stored at the pixel positions, we replace the constant GL RGB with GL_COLOR_INDEX.

- To rotate the color values, we rearrange the rows and columns of the color array, as described in the previous section. Then we put the rotated array back in the buffer with **glDrawPixels (width, height, GL_RGB, GL_UNSIGNED_BYTE, colorArray);**

- A two-dimensional scaling transformation can be performed as a raster operation in OpenGL by specifying scaling factors and then invoking either **glCopyPixels** or **glDrawPixels**.
- For the raster operations, we set the scaling factors with

  glPixelZoom (sx, sy);

- We can also combine raster transformations with logical operations to produce various effects with the *exclusive or* operator

OpenGL Functions for Two-Dimensional Geometric Transformations

- ✓ To perform a translation, we invoke the translation routine and set the components for the three-dimensional translation vector.
- ✓ In the rotation function, we specify the angle and the orientation for a rotation axis that intersects the coordinate origin.
- ✓ In addition, a scaling function is used to set the three coordinate scaling factors relative to the coordinate origin. In each case, the transformation routine sets up a 4 × 4 matrix that is applied to the coordinates of objects that are referenced after the transformation call

Basic OpenGL Geometric Transformations

➔ A 4× 4 translation matrix is constructed with the following routine:

glTranslate* (tx, ty, tz);

- ✓ Translation parameters **tx**, **ty**, and **tz** can be assigned any real-number values, and the single suffix code to be affixed to this function is either **f** (float) or **d** (double).
- ✓ For two-dimensional applications, we set **tz** = 0.0; and a two-dimensional position is represented as a four-element column matrix with the *z* component equal to 0.0.
- ✓ example: **glTranslatef (25.0, -10.0, 0.0);**

➔ Similarly, a 4 × 4 rotation matrix is generated with

glRotate* (theta, vx, vy, vz);

- ✓ where the vector **v** = (**vx**, **vy**, **vz**) can have any floating-point values for its components.
- ✓ This vector defines the orientation for a rotation axis that passes through the coordinate origin.
- ✓ If **v** is not specified as a unit vector, then it is normalized automatically before the elements of the rotation matrix are computed.

- ✓ The suffix code can be either **f** or **d**, and parameter **theta** is to be assigned a rotation angle in degree.
- ✓ For example, the statement: **glRotatef (90.0, 0.0, 0.0, 1.0);**

➔ We obtain a 4 × 4 scaling matrix with respect to the coordinate origin with the following routine:

glScale* (sx, sy, sz);

- ✓ The suffix code is again either **f** or **d**, and the scaling parameters can be assigned any real-number values.
- ✓ Scaling in a two-dimensional system involves changes in the *x* and *y* dimensions, so a typical two-dimensional scaling operation has a *z* scaling factor of 1.0
- ✓ **Example: glScalef (2.0, -3.0, 1.0);**

OpenGL Matrix Operations

- ✓ The glMatrixMode routine is used to set the *projection mode which designates the matrix that is to be used for the projection transformation.*
- ✓ We specify the *modelview mode* with the statement

glMatrixMode (GL_MODELVIEW);

- which designates the 4×4 modelview matrix as the **current matrix**
- Two other modes that we can set with the **glMatrixMode** function are the *texture mode* and the *color mode*.
- The texture matrix is used for mapping texture patterns to surfaces, and the color matrix is used to convert from one color model to another.
- The default argument for the **glMatrixMode** function is **GL_MODELVIEW**.

- ✓ With the following function, we assign the identity matrix to the current matrix:

glLoadIdentity ( );

- ✓ Alternatively, we can assign other values to the elements of the current matrix using

glLoadMatrix* (elements16);

- ✓ A single-subscripted, 16-element array of floating-point values is specified with parameter **elements16**, and a suffix code of either **f** or **d** is used to designate the data type
- ✓ The elements in this array must be specified in *column-major* order
- ✓ To illustrate this ordering, we initialize the modelview matrix with the following code:

```
glMatrixMode (GL_MODELVIEW);
GLfloat elems [16];
GLint k;
for (k = 0; k < 16; k++)
        elems [k] = float (k);
glLoadMatrixf (elems);
```

Which produces the matrix

$$\mathbf{M} = \begin{bmatrix} 0.0 & 4.0 & 8.0 & 12.0 \\ 1.0 & 5.0 & 9.0 & 13.0 \\ 2.0 & 6.0 & 10.0 & 14.0 \\ 3.0 & 7.0 & 11.0 & 15.0 \end{bmatrix}$$

- ✓ We can also concatenate a specified matrix with the current matrix as follows:

  glMultMatrix* (otherElements16);

- ✓ Again, the suffix code is either **f** or **d**, and parameter **otherElements16** is a 16-element, single-subscripted array that lists the elements of some other matrix in column-major order.
- ✓ Thus, assuming that the current matrix is the modelview matrix, which we designate as **M**, then the updated modelview matrix is computed as

  M = **M**· **M**'
- ✓ The **glMultMatrix** function can also be used to set up any transformation sequence with individually defined matrices.
- ✓ For example,

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ( ); // Set current matrix to the identity.
glMultMatrixf (elemsM2); // Postmultiply identity with matrix M2.
glMultMatrixf (elemsM1); // Postmultiply M2 with matrix M1.
```

produces the following current modelview matrix:

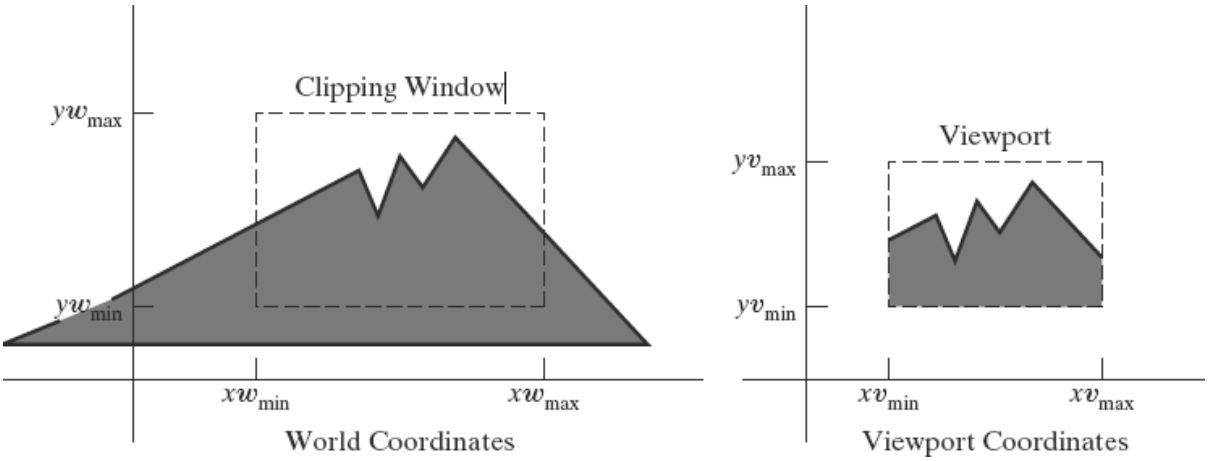
M = **M**2 · **M**1

2.3 Two Dimensional Viewing

2.3.1 2D viewing pipeline
2.3.1 OpenGL 2D viewing functions.

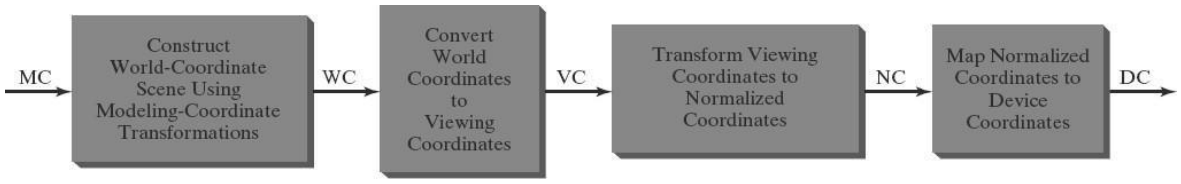
The Two-Dimensional Viewing Pipeline

- A section of a two-dimensional scene that is selected for display is called a clipping Window.
- Sometimes the clipping window is alluded to as the *world window* or the *viewing window*
- Graphics packages allow us also to control the placement within the display window using another "window" called the **viewport**.
- The clipping window selects *what* we want to see; the viewport indicates *where* it is to be viewed on the output device.
- By changing the position of a viewport, we can view objects at different positions on the display area of an output device
- Usually, clipping windows and viewports are rectangles in standard position, with the rectangle edges parallel to the coordinate axes.
- We first consider only rectangular viewports and clipping windows, as illustrated in Figure



Viewing Pipeline

- The mapping of a two-dimensional, world-coordinate scene description to device coordinates is called a **two-dimensional viewing transformation**.
- This transformation is simply referred to as the *window-to-viewport transformation* or the *windowing transformation*
- We can describe the steps for two-dimensional viewing as indicated in Figure



- Once aworld-coordinate scene has been constructed,wecould set up a separate two-dimensional, **viewingcoordinate reference frame** for specifying the clipping window.
- To make the viewing process independent of the requirements of any output device, graphics systems convert object descriptions to normalized coordinates and apply the clipping routines.
- Systems use normalized coordinates in the range from 0 to 1, and others use a normalized range from −1 to 1.
- At the final step of the viewing transformation, the contents of the viewport are transferred to positions within the display window.
- Clipping is usually performed in normalized coordinates.
- This allows us to reduce computations by first concatenating the various transformation matrices

OpenGL Two-Dimensional Viewing Functions

- The GLU library provides a function for specifying a two-dimensional clipping window, and we have GLUT library functions for handling display windows.

OpenGL Projection Mode

- ✓ Before we select a clipping window and a viewport in OpenGL, we need to establish the appropriate mode for constructing the matrix to transform from world coordinates to screen coordinates.

- ✓ We must set the parameters for the clipping window as part of the projection transformation.
- ✓ Function:

  glMatrixMode (GL_PROJECTION);
- ✓ We can also set the initialization as

  glLoadIdentity ( );

  This ensures that each time we enter the projection mode, the matrix will be reset to the identity matrix so that the new viewing parameters are not combined with the previous ones

GLU Clipping-Window Function

- ✓ To define a two-dimensional clipping window, we can use the GLU function:

  gluOrtho2D (xwmin, xwmax, ywmin, ywmax);
- ✓ This function specifies an orthogonal projection for mapping the scene to the screen the orthogonal projection has no effect on our two-dimensional scene other than to convert object positions to normalized coordinates.
- ✓ Normalized coordinates in the range from −1 to 1 are used in the OpenGL clipping routines.
- ✓ Objects outside the normalized square (and outside the clipping window) are eliminated from the scene to be displayed.
- ✓ If we do not specify a clipping window in an application program, the default coordinates are (*xw*min, *yw*min*)* = *(*−1.0, −1.0*)* and (*xw*max, *yw*max*)* = *(*1.0, 1.0*)*.
- ✓ Thus the default clipping window is the normalized square centered on the coordinate origin with a side length of 2.0.

OpenGL Viewport Function

- ✓ We specify the viewport parameters with the OpenGL function

  glViewport (xvmin, yvmin, vpWidth, vpHeight);

  Where,

  ➔ **xvmin** and **yvmin** specify the position of the lowerleft corner of the viewport relative to the lower-left corner of the display window,

➔ **vpWidth** and **vpHeight** are pixel width and height of the viewport

- ✓ Coordinates for the upper-right corner of the viewport are calculated for this transformation matrix in terms of the viewport width and height:

$$xv_{\max} = xv_{\min} + vpWidth, \qquad yv_{\max} = yv_{\min} + vpHeight$$

- ✓ Multiple viewports can be created in OpenGL for a variety of applications.
- ✓ We can obtain the parameters for the currently active viewport using the query function

glGetIntegerv (GL_VIEWPORT, vpArray);

where,

➔ **vpArray** is a single-subscript, four-element array.

Creating a GLUT Display Window

- ✓ The GLUT library interfaces with any window-management system, we use the GLUT routines for creating and manipulating display windows so that our example programs will be independent of any specific machine.
- ✓ We first need to initialize GLUT with the following function:

glutInit (&argc, argv);

- ✓ We have three functions inGLUTfor defining a display window and choosing its dimensions and position:

1. **glutInitWindowPosition (xTopLeft, yTopLeft);**

➔ gives the integer, screen-coordinate position for the top-left corner of the display window, relative to the top-left corner of the screen

2. **glutInitWindowSize (dwWidth, dwHeight);**

➔ we choose a width and height for the display window in positive integer pixel dimensions.

➔ If we do not use these two functions to specify a size and position, the default size is 300 by 300 and the default position is *(*−1, −1*)*, which leaves the positioning of the display window to the window-management system

3. **glutCreateWindow ("Title of Display Window");**

➔ creates the display window, with the specified size and position, and assigns a title, although the use of the title also depends on the windowing system

Setting the GLUT Display-Window Mode and Color

✓ Various display-window parameters are selected with the GLUT function

1. **glutInitDisplayMode (mode);**

➔ We use this function to choose a color mode (RGB or index) and different buffer combinations, and the selected parameters are combined with the logical **or** operation.

2. **glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);**

➔ The color mode specification **GLUT_RGB** is equivalent to **GLUT_RGBA.**

3. **glClearColor (red, green, blue, alpha);**

➔ A background color for the display window is chosen in RGB mode with the OpenGL routine

4. **glClearIndex (index);**

➔ This function sets the display window color using color-index mode,

➔ Where parameter **index** is assigned an integer value corresponding to a position within the color table.

GLUT Display-Window Identifier

✓ Multiple display windows can be created for an application, and each is assigned a positive-integer **display-window identifier,** starting with the value 1 for the first window that is created.

✓ Function:

windowID = glutCreateWindow ("A Display Window");

Deleting a GLUT Display Window

- ✓ If we know the display window's identifier, we can eliminate it with the statement

  glutDestroyWindow (windowID);

Current GLUT Display Window

- ✓ When we specify any display-window operation, it is applied to the **current display window,** which is either the last display window that we created or the one**.**
- ✓ we select with the following command

  glutSetWindow (windowID);
- ✓ We can query the system to determine which window is the current display window:

  currentWindowID = glutGetWindow ( );

  ➔ A value of 0 is returned by this function if there are no display windows or if the current display window was destroyed

Relocating and Resizing a GLUT Display Window

- ✓ We can reset the screen location for the current display window with the function

  glutPositionWindow (xNewTopLeft, yNewTopLeft);
- ✓ Similarly, the following function resets the size of the current display window:

  glutReshapeWindow (dwNewWidth, dwNewHeight);
- ✓ With the following command, we can expand the current display window to fill the screen:

  glutFullScreen ( );
- ✓ Whenever the size of a display window is changed, its aspect ratio may change and objects may be distorted from their original shapes. We can adjust for a change in display-window dimensions using the statement

  glutReshapeFunc (winReshapeFcn);

Managing Multiple GLUT Display Windows

- ✓ The GLUT library also has a number of routines for manipulating a display window in various ways.

- ✓ We use the following routine to convert the current display window to an icon in the form of a small picture or symbol representing the window:

  glutIconifyWindow ( );
- ✓ The label on this icon will be the same name that we assigned to the window, but we can change this with the following command:

  glutSetIconTitle ("Icon Name");
- ✓ We also can change the name of the display window with a similar command:

  glutSetWindowTitle ("New Window Name");
- ✓ We can choose any display window to be in front of all other windows by first designating it as the current window, and then issuing the "pop-window" command:

  glutSetWindow (windowID);

  glutPopWindow ( );
- ✓ In a similar way, we can "push" the current display window to the back so that it is behind all other display windows. This sequence of operations is

  glutSetWindow (windowID);

  glutPushWindow ( );
- ✓ We can also take the current window off the screen with

  glutHideWindow ( );
- ✓ In addition, we can return a "hidden" display window, or one that has been converted to an icon, by designating it as the current display window and then invoking the function

  glutShowWindow ( );

GLUT Subwindows

- ✓ Within a selected display window, we can set up any number of second-level display windows, which are called *subwindows.*
- ✓ We create a subwindow with the following function:

  glutCreateSubWindow (windowID, xBottomLeft, yBottomLeft, width, height);
- ✓ Parameter **windowID** identifies the display window in which we want to set up the subwindow.

- ✓ Subwindows are assigned a positive integer identifier in the same way that first-level display windows are numbered, and we can place a subwindow inside another subwindow.
- ✓ Each subwindow can be assigned an individual display mode and other parameters. We can even reshape, reposition, push, pop, hide, and show subwindows

Selecting a Display-Window Screen-Cursor Shape

- ✓ We can use the following GLUT routine to request a shape for the screen cursor that is to be used with the current window:

  glutSetCursor (shape);

  where, shape can be

  ➔ **GLUT_CURSOR_UP_DOWN :** an up-down arrow.

  ➔ **GLUT_CURSOR_CYCLE:** A rotating arrow is chosen

  ➔ **GLUT_CURSOR_WAIT:** a wristwatch shape.

  ➔ **GLUT_CURSOR_DESTROY:** a skull and crossbones

Viewing Graphics Objects in a GLUT Display Window

- ✓ After we have created a display window and selected its position, size, color, and other characteristics, we indicate what is to be shown in that window
- ✓ Then we invoke the following function to assign something to that window:

  glutDisplayFunc (pictureDescrip);

- ✓ This routine, called **pictureDescrip** for this example, is referred to as a *callback function* because it is the routine that is to be executed whenever GLUT determines that the display-window contents should be renewed.
- ✓ We may need to call **glutDisplayFunc** after the **glutPopWindow** command if the display window has been damaged during the process of redisplaying the windows.
- ✓ In this case, the following function is used to indicate that the contents of the current display window should be renewed:

  glutPostRedisplay ( );

Executing the Application Program

- ✓ When the program setup is complete and the display windows have been created and initialized, we need to issue the final GLUT command that signals execution of the program:

  glutMainLoop ( );

Other GLUT Functions

- ✓ Sometimes it is convenient to designate a function that is to be executed when there are no other events for the system to process. We can do that with

  glutIdleFunc (function);

- ✓ Finally, we can use the following function to query the system about some of the current state parameters:

  glutGet (stateParam);

- ✓ This function returns an integer value corresponding to the symbolic constant we select for its argument.
- ✓ For example, for the stateParam we can have the values
  - ➔ **GLUT_WINDOW_X**: obtains the *x*-coordinate position for the top-left corner of the current display window
  - ➔ **GLUT_WINDOW_WIDTH** or **GLUT_SCREEN_WIDTH** : retrieve the current display-window width or the screen width with**.**

MODULE 3

Clipping,3D Geometric Transformations, Color and Illumination Models

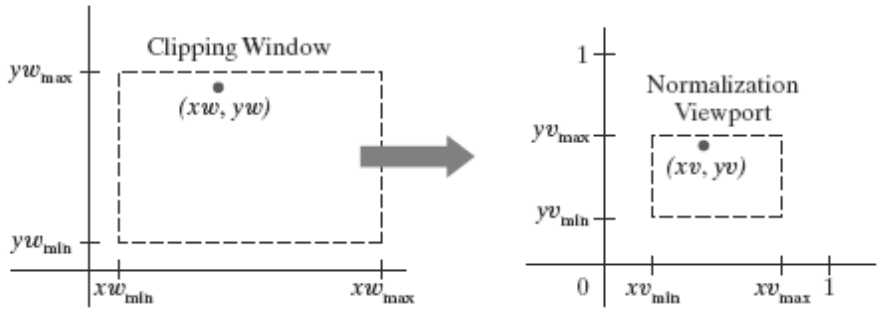
The Clipping Window

Rectangular clipping windows are easily defined by giving the coordinates of two opposite corners of each rectangle. Clipping window with any shape, size, and orientation we can choose.

Normalization and Viewport Transformations

Q.Design a transformation matrix for window to viewport transformation.

- Position (x_w, y_w) in the clipping window is mapped to position (x_v, y_v) in the associated viewport.



- To transform the world-coordinate point into the same relative position within the viewport, we require that

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

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

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

$$x_v = s_x x_w + t_x$$

$$y_v = s_y y_w + t_y$$

Where the scaling factors are

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

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

and the translation factors are

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

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

We could obtain the transformation from world coordinates to viewport coordinates with the following sequence:

1. Scale the clipping window to the size of the viewport using a fixed-point position of (xw_{min}, yw_{min}).

2. Translate (xw_{min}, yw_{min}) to (xv_{min}, yv_{min}).

The scaling transformation in step (1) can be represented with the two dimensional Matrix

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & xw_{\min}(1 - s_x) \\ 0 & s_y & yw_{\min}(1 - s_y) \\ 0 & 0 & 1 \end{bmatrix}$$

The two-dimensional matrix representation for the translation of the lower-left corner of the clipping window to the lower-left viewport corner is

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & xv_{\min} - xw_{\min} \\ 0 & 1 & yv_{\min} - yw_{\min} \\ 0 & 0 & 1 \end{bmatrix}$$

And the composite matrix representation for the transformation to the normalized viewport is

$$\mathbf{M}_{\text{window, normviewp}} = \mathbf{T} \cdot \mathbf{S} = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

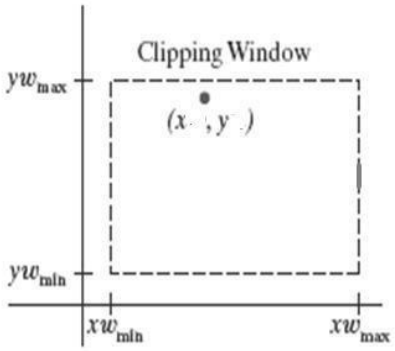
Clipping Algorithms

Any procedure that eliminates those portions of a picture that are either inside or outside a specified region of space is referred to as a **clipping algorithm** or simply **clipping.**

Different objects clipping are

1. Point clipping
2. Line clipping (straight-line segments)
3. Fill-area clipping (polygons)
4. Curve clipping
5. Text clipping

Two-Dimensional Point Clipping



For a clipping rectangle in standard position, we save a two-dimensional point $\mathbf{P} = (x, y)$ for display if the following inequalities are satisfied:

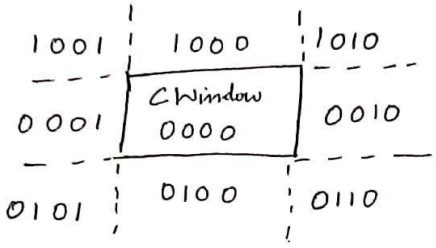
$$xw_{min} \leq x \leq xw_{max} \quad \text{and} \quad yw_{min} \leq y \leq yw_{max}$$

If any of these four inequalities is not satisfied, the point is clipped

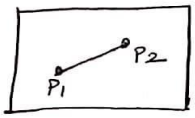
Q.What is Clipping? With the help of a suitable example explain cohen Sutherland line, algorithm.

Cohen Sutherland Algorithm

- Algorithm Works on Region Code
- Region code is 4 Bit Code (A B R L) Above, Below, Right, Left
  (T B R L) Top, Bottom, Right, Left

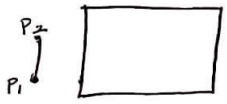


Case 1: If both endpoint Region code is zero, Completely Inside & Visible



P1 0000 (zero)
P2 0000 (zero)
AND 0000 (Zero)

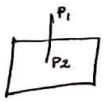
Case 2: If both endpoint Region code is Nonzero, then apply logical AND, and result is Nonzero, Completely Outside & Invisible

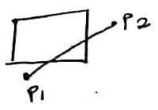


P1 = 0001 (Nonzero)
P2 = 0001 (Nonzero)
AND 0001 (Nonzero)

Case 3: a) If one of P_1 & P_2 is Zero } logical AND is Zero
b) Both Nonzero

Then Find Intersection Point





Finding Intersection Points

1) Find y values of Vertical lines

Consider a line Segment (x_1, y_1) (x_2, y_2) & Intersection Point (x, y)

Find Slope of (x_1, y_1) & (x, y)

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

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

$$\boxed{y = y_1 + m(x - x_1)}$$

$x = x_{wmin}$ (left border)

$x = x_{wmax}$ (right border)

CW

x_{wmin} x_{wmax}

2) Find x values of horizontal line

Find Slope of (x_1, y_1) & (x, y)

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

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

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

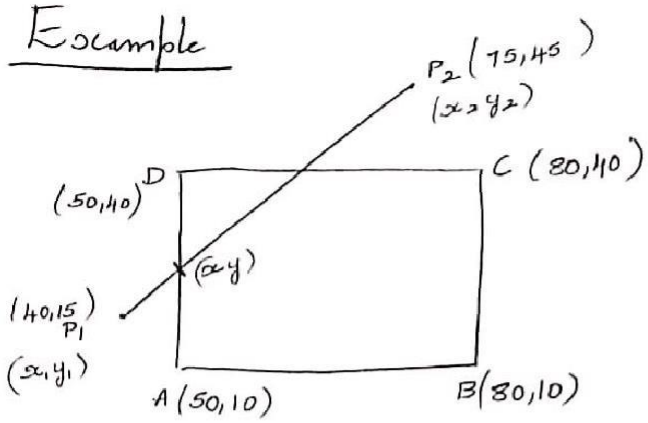
y_{wmax}

CW

$y = y_{wmin}$ (bottom boundary) y_{wmin}

$y = y_{wmax}$ (upper boundary)

Example



$P_1 = 0001$ (Nonzero)

$P_2 = 1000$ (Nonzero)

0000 Zero

Intersection Point (x, y)

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

$= 15 + 0.85(50 - 40)$

$= 15 + 0.85(10)$

$\boxed{y = 23.5}$

$(x, y) = \boxed{(50, 23.5)}$

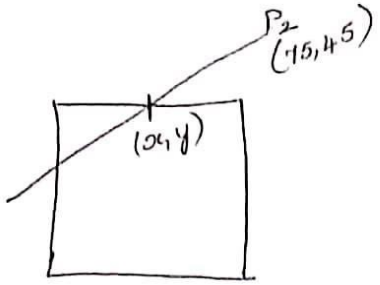
$x_{wmin} = 50$

$x_{wmax} = 80$

P_1 (x_1, y_1) $(40, 15)$ — P_2 (x_2, y_2) $(75, 45)$

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{45 - 15}{75 - 40} = \frac{30}{35}$$

$\boxed{m = 0.85}$



$y_{max} = 40$

$y_{min} = 10$

(x, y)

$$x = x_1 + \frac{1}{m}(y - y_1)$$

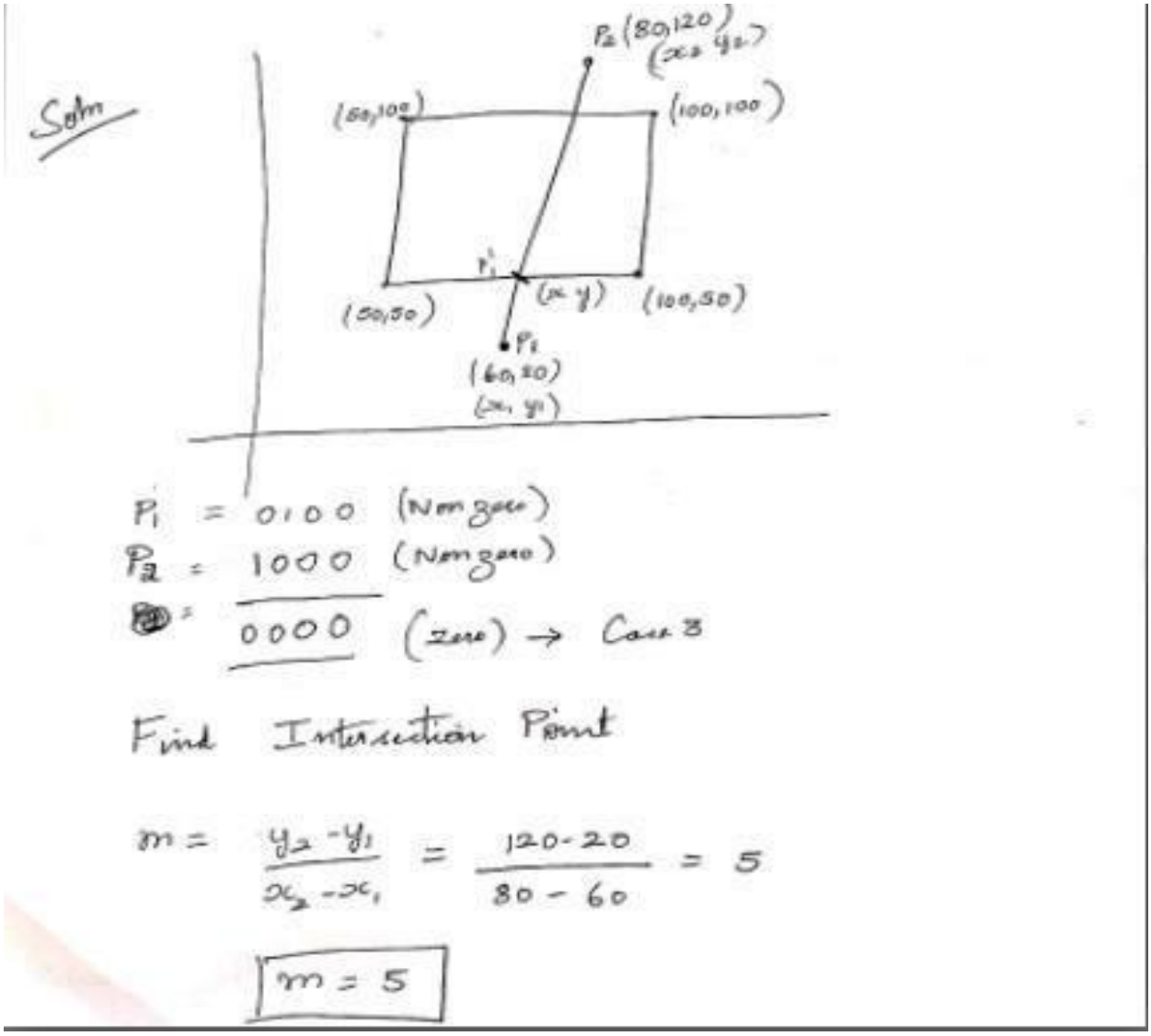
$$= 75 + \frac{1}{0.85}(40 - 45)$$

$$= 75 + (-5.882)$$

$$\boxed{x = 69.2}$$

$$\boxed{x, y = (69.2, 40)}$$

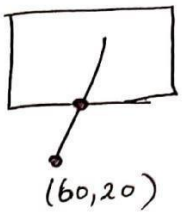
Q.Explain Cohen Sutherland line clipping, clip the lines with coordinates (x0,y0)=(60,20) and (x1,y1) =(80,120) given the window boundaries(xwmin,ywmin)=(50,50) and (xmax, ymax) = (100,100)



Find $P_1'(x, y)$

We know $y = y_{wmin} = 50$

Find x



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

$$= 60 + \frac{(50 - 20)}{5}$$

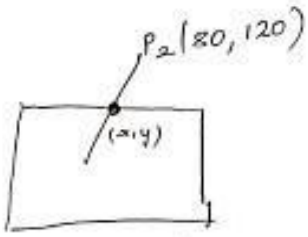
$$x = 66$$

$$P_1'(x, y) = (66, 50)$$

Find $P_2'(x, y)$

We Know $y = y_{max} = 100$

Find x



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

$$= 80 + \frac{(100 - 120)}{5}$$

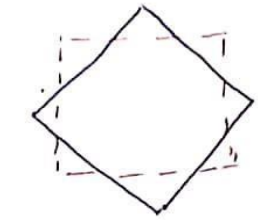
$$= 80 + \frac{(-20)}{5}$$

$$= 80 - 4$$

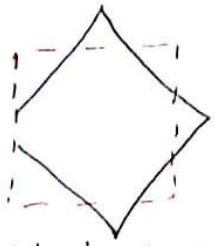
$$x = 76$$

$$P_2'(x, y) = (76, 100)$$

Polygon Clipping

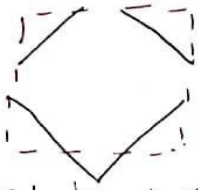


Original polygon

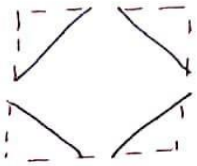


Left boundary clipping

Right Boundary Clipping



Top boundary C



Right boundary C

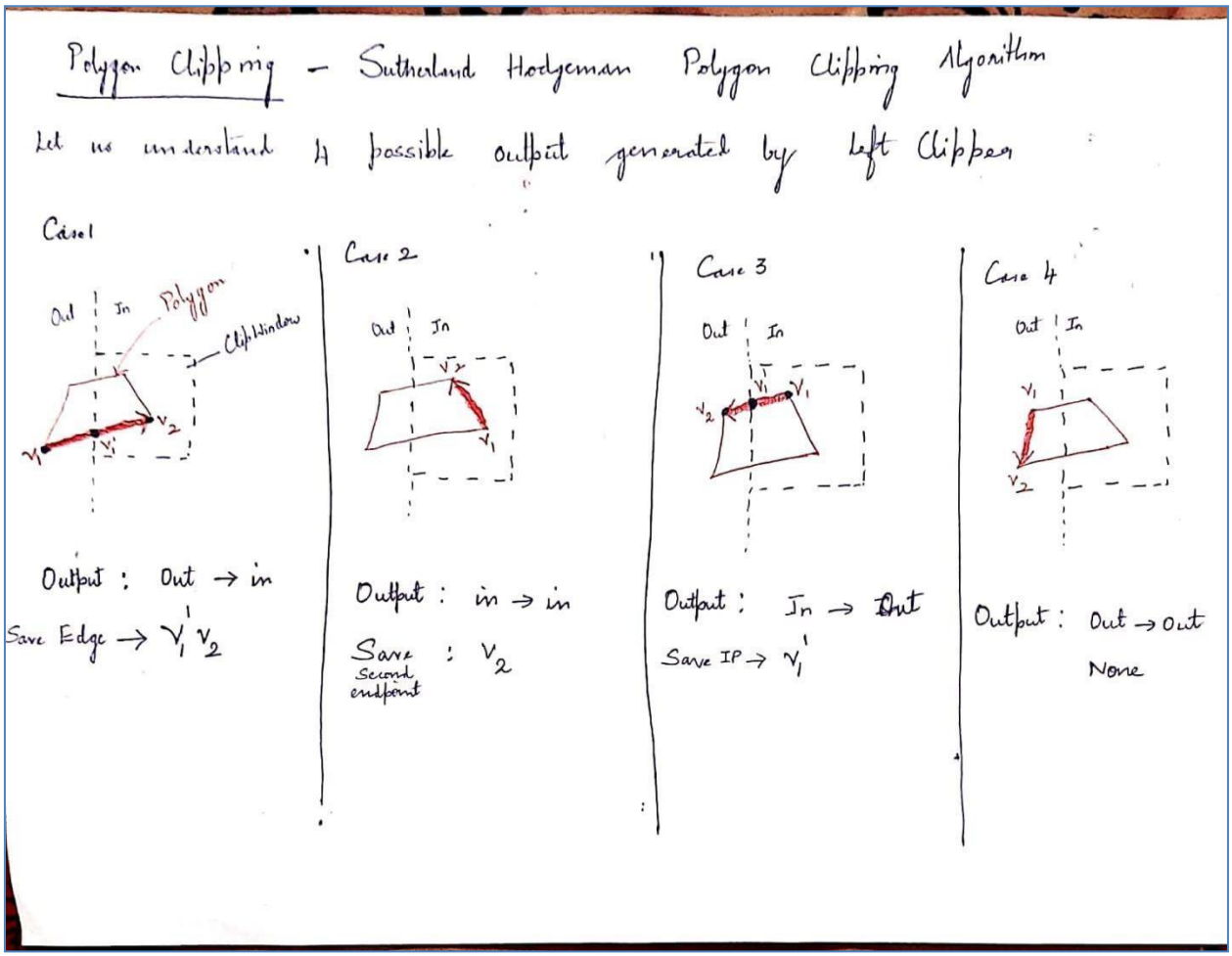
Q.What is Clipping? Explain with example Sutherland Hodgeman Polygon clipping algorithm

Clipping: Any procedure that eliminates those portions of a picture that are either inside or outside a specified region of space

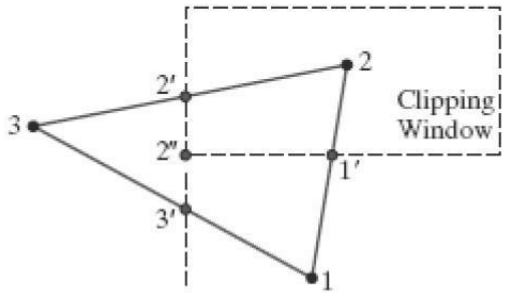
Sutherland Hodgeman Polygon clipping algorithm

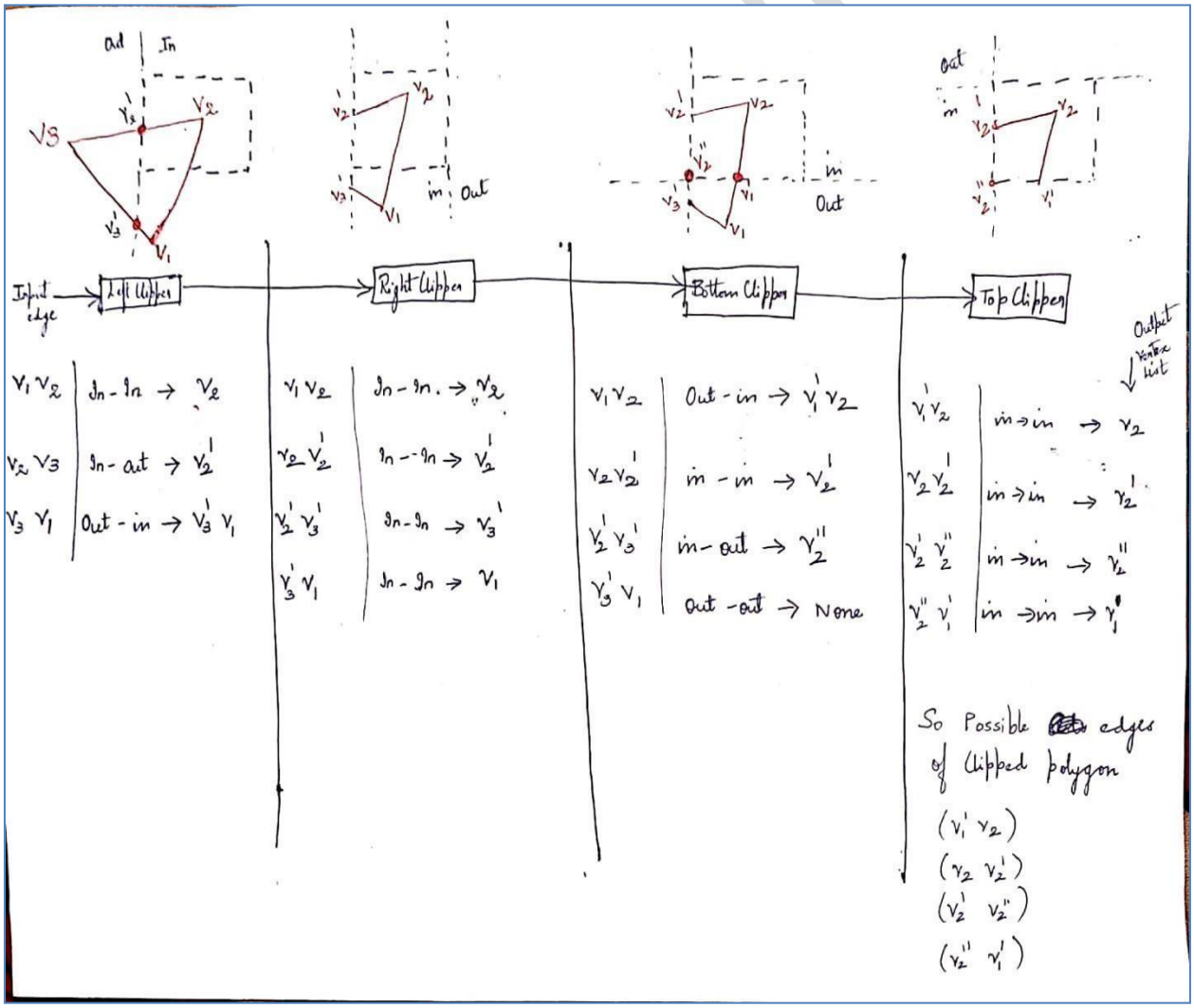
Send pair of endpoints for each successive polygon line segment through the series of clippers.
Four possible cases:

1. If the first input vertex is outside this clipping-window border and the second vertex is inside, both the intersection point of the polygon edge with the window border and the second vertex are sent to the next clipper
2. If both input vertices are inside this clipping-window border, only the second vertex is sent to the next clipper
3. If the first vertex is inside and the second vertex is outside, only the polygon edge intersection position with the clipping-window border is sent to the next clipper
4. If both input vertices are outside this clipping-window border, no vertices are sent to the next clipper



Example for Sutherland Hodgeman Polygon clipping algorithm





3DGeometric Transformations

Three-Dimensional Geometric Transformations

Q.With the help of a suitable diagram explain basic 3D Geometric transformation techniques and give the transformation matrix.

The most common 3D transformations are:

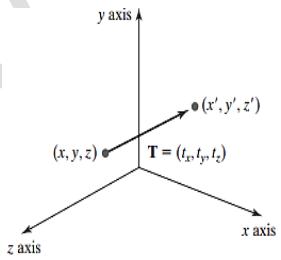
Translation
Rotation
Scaling

Three-Dimensional Translation

- A position P=(x, y, z) in three-dimensional space is translated to a location P'=(x',y',z') by adding translation distances tx, ty, and tz to the Cartesian coordinates of P.

$$x' = x + t_x, \qquad y' = y + t_y, \qquad z' = z + t_z$$

- 3D translation is given in Figure



- 3D translation operations can be represented in matrix format. The coordinate positions, P and P', are represented in homogeneous coordinates with four-element column matrices, and the translation operator T is a 4×4 matrix:

$$\mathbf{P}' = \mathbf{T} \cdot \mathbf{P}$$

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

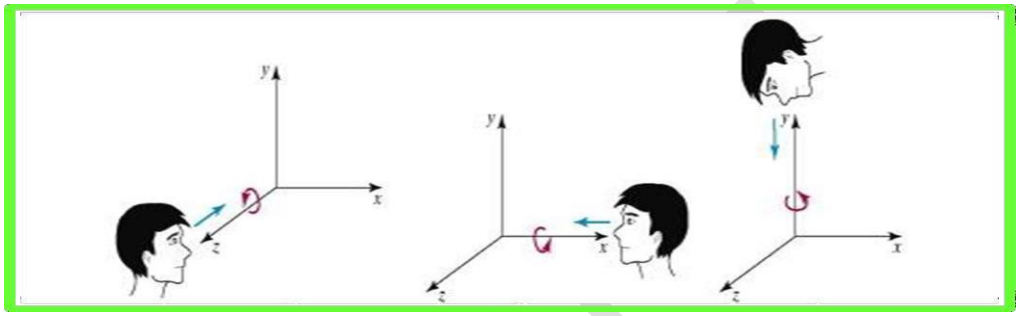
Three-Dimensional Rotation

- Rotate an object about any axis in space.

  - Easiest rotation axes to handle are those that are parallel to the Cartesian-coordinate axes.

  - Also can use combinations of coordinate-axis rotations (along with appropriate translations) to specify a rotation about any other line in space.

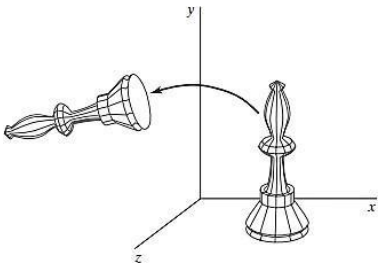
- Positive rotation angles produce counterclockwise rotations about a coordinate axis (for negative direction along that coordinate axis)



- The 3D z-axis rotation equations are easily extended to three dimensions, as follows:

Along z axis:

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



Where, θ the rotation angle about the z axis

z-coordinate values are unchanged by this transformation

- 3D Rotation operations can be represented in matrix format. In homogeneous-coordinate form, the three-dimensional *z*-axis rotation equations are

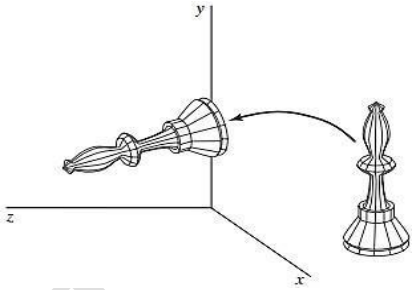
$$\mathbf{P}' = \mathbf{R}_z(\theta) \cdot \mathbf{P}$$

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

- Similarly along x axis and y axis

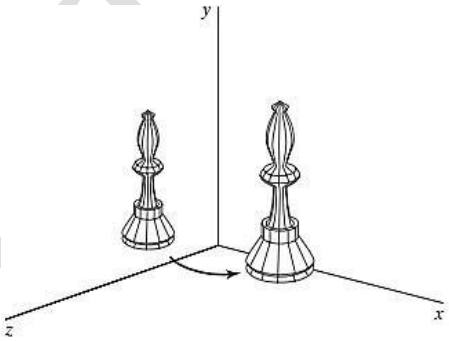
Along x axis

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



Along y axis

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



Three-Dimensional Scaling

- The matrix expression for the three-dimensional scaling transformation of a position P=(x, y, z)relative to the coordinate origin is a simple extension of 3D scaling. Include the parameter for z-coordinate scaling in the transformation matrix:

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

- The three-dimensional scaling transformation for a point position can be represented as

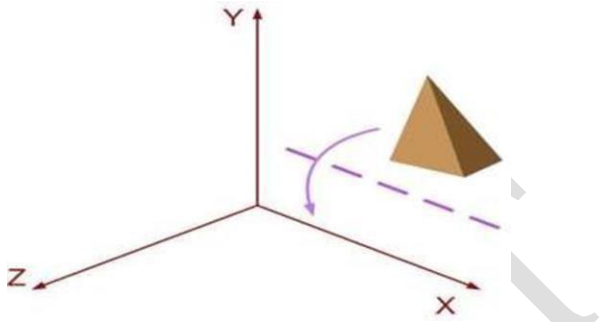
$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P}$$

where scaling parameters *sx*, *sy*, and *sz* are assigned any positive values.

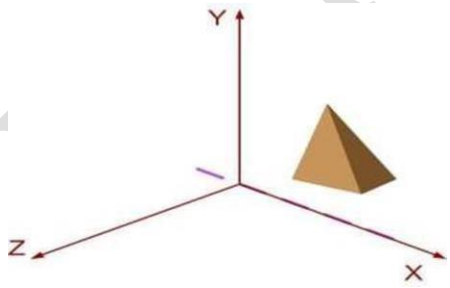
- Explicit expressions for the scaling transformation relative to the origin are

Q.Design transformation matrix to rotate an 3D Object about an axis that is parallel to one of the coordinate axes.

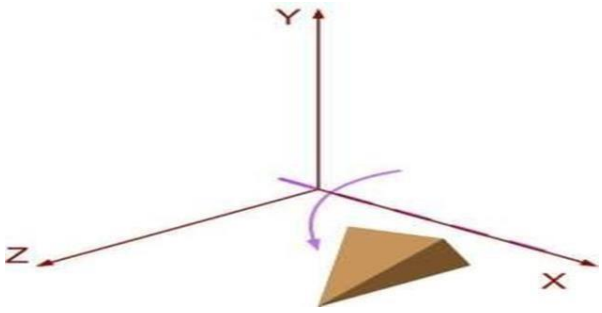
A rotation matrix for any axis that does not coincide with a coordinate axis(as shown in below first figure) can be set up as a composite transformation involving combinations of translations and the coordinate-axis rotations the following transformation sequence is used:



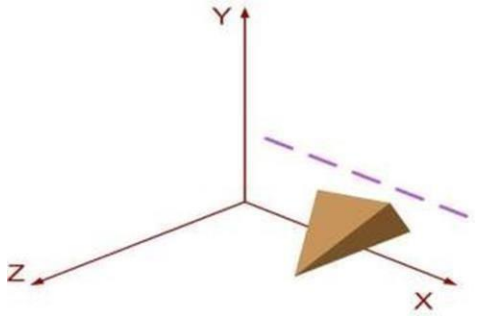
1. **Translate the object so that the rotation axis coincides with the parallel coordinate axis.**



2. **Perform the specified rotation about that axis.**



3. **Translate the object so that the rotation axis is moved back to its original position.**



A coordinate position **P** is transformed with the sequence shown in this figure as

$$P' = T^{-1} \cdot R_x(\theta) \cdot T \cdot P$$

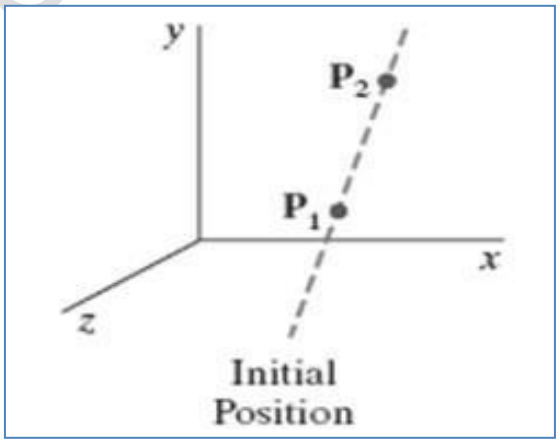
Where the composite rotation matrix for the transformation is

$$R(\theta) = T^{-1} \cdot R_x(\theta) \cdot T$$

Q.Obtain the matrix representation for rotation of a object about an arbitrary axis

Rotation about arbitrary axis that is not parallel to one of the coordinate axes, we can accomplish the required rotation in five steps:

Consider the Initial position of object in the below figure



Components of the rotation-axis vector are then computed as

$$\mathbf{V} = \mathbf{P2} - \mathbf{P1}$$
$$= (x_2 - x_1, y_2 - y_1, z_2 - z_1)$$

The unit rotation-axis vector **u** is

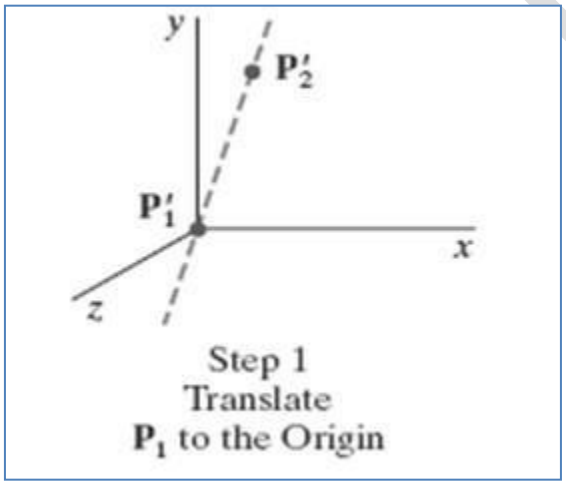
$$\mathbf{u} = \frac{\mathbf{V}}{|\mathbf{V}|} = (a, b, c)$$

Where the components a, b, and c are the direction cosines for the rotation axis

$$a = \frac{x_2 - x_1}{|\mathbf{V}|}, \qquad b = \frac{y_2 - y_1}{|\mathbf{V}|}, \qquad c = \frac{z_2 - z_1}{|\mathbf{V}|}$$

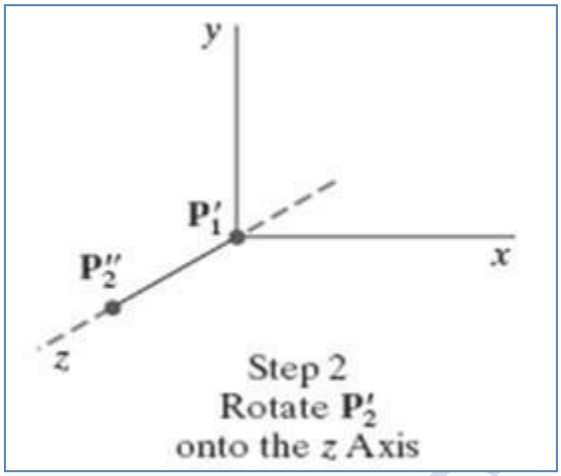
Rotation about arbitrary axis that is not parallel to one of the coordinate axes, we can accomplish the required rotation in five steps:

1. Translate the object so that the rotation axis passes through the coordinate origin.



$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. Rotate the object so that the axis of rotation coincides with one of the coordinate axes.



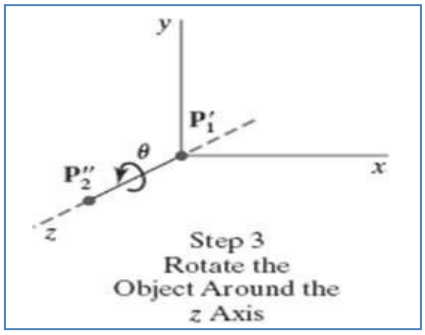
Rotation of u around the x axis into the x z plane is accomplished by rotating u' (the projection of u in the y z plane) through angle α onto the z axis.

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{c}{d} & -\frac{b}{d} & 0 \\ 0 & \frac{b}{d} & \frac{c}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation of unit vector u" (vector u after rotation into the x z plane) about the y axis. Positive rotation angle β aligns u" with vector uz .

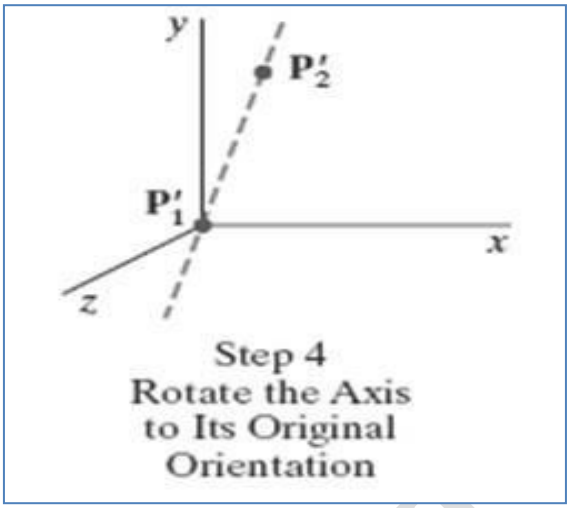
$$\mathbf{R}_y(\beta) = \begin{bmatrix} d & 0 & -a & 0 \\ 0 & 1 & 0 & 0 \\ a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3. Perform the specified rotation about the selected coordinate axis.



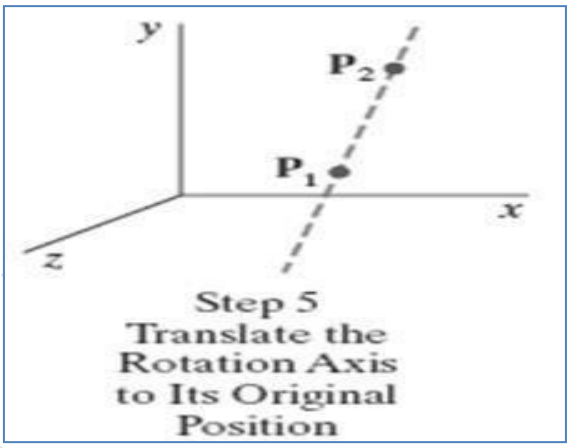
$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4. Apply inverse rotations to bring the rotation axis back to its original orientation.



Step 4
Rotate the Axis
to Its Original
Orientation

5. Apply the inverse translation to bring the rotation axis back to its original spatial position.



Step 5
Translate the
Rotation Axis
to Its Original
Position

The transformation matrix for rotation about an arbitrary axis can then be expressed as the composition of these seven individual transformations:

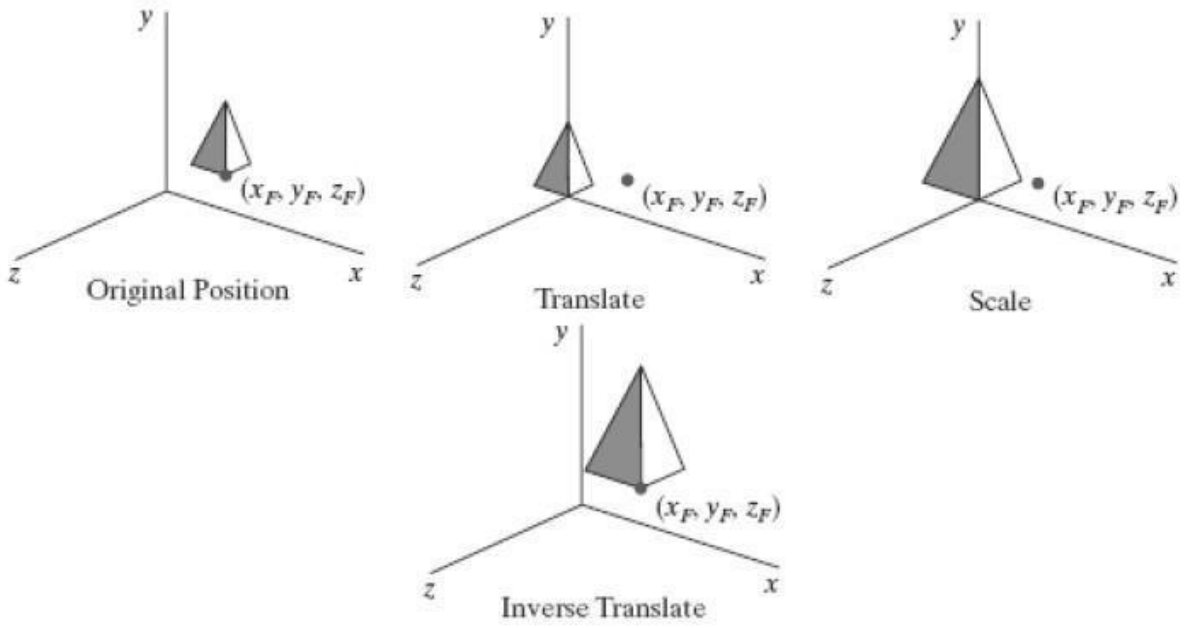
$$R(\theta) = T^{-1} \cdot R_x^{-1}(\alpha) \cdot R_y^{-1}(\beta) \cdot R_z(\theta) \cdot R_y(\beta) \cdot R_x(\alpha) \cdot T$$

Q.Obtain Scaling transformation matrix with respect to any selected fixed position.

Scaling transformation with respect to any selected fixed position (x_f, y_f, z_f) using the following transformation sequence:

1. Translate the fixed point to the origin.
2. Apply the scaling transformation relative to the coordinate origin
3. Translate the fixed point back to its original position.

This sequence of transformations is demonstrated



$$\mathbf{T}(x_f, y_f, z_f) \cdot \mathbf{S}(s_x, s_y, s_z) \cdot \mathbf{T}(-x_f, -y_f, -z_f) = \begin{bmatrix} s_x & 0 & 0 & (1-s_x)x_f \\ 0 & s_y & 0 & (1-s_y)y_f \\ 0 & 0 & s_z & (1-s_z)z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Other Three-Dimensional Transformations

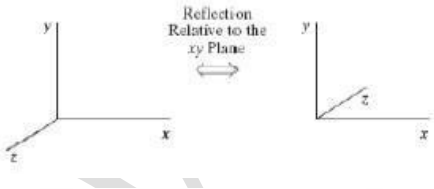
Q.Explain the 3D reflection and shearing.

Three-Dimensional Reflections

A reflection in a three-dimensional space can be performed relative to a selected *reflection axis* or with respect to a *reflection plane*.

Reflections with respect to a plane are similar; when the reflection plane is a coordinate plane (x_y, x_z, or y_z), we can think of the transformation as a 180◦ rotation in four-dimensional space with a conversion between a left-handed frame and a right-handed frame

An example of a reflection that converts coordinate specifications froma right handed system to a left-handed system is shown below



The matrix representation for this reflection relative to the *xy* plane is

$$M_{zreflect} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Three-Dimensional Shears

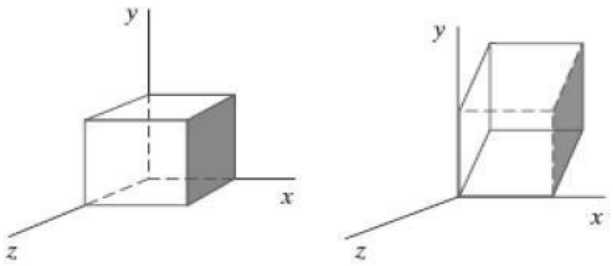
These transformations can be used to modify object shapes.

For three-dimensional we can also generate shears relative to the z axis.

A general z-axis shearing transformation relative to a selected reference position is produced with the following matrix:

$$M_{z\text{shear}} = \begin{bmatrix} 1 & 0 & sh_{zx} & -sh_{zx} \cdot z_{ref} \\ 0 & 1 & sh_{zy} & -sh_{zy} \cdot z_{ref} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The Below figure shows the shear transformation of a cube



Affine Transformations

Q.Explain the meaning of Affine transformation

Affine transformations (in two dimensions, three dimensions, or higher dimensions) have the general properties that parallel lines are transformed into parallel lines, and finite points map to finite points.

Translation, rotation, scaling, reflection,andshear are examples of affine transformations.

Another example of an affine transformation is the conversion of coordinate descriptions for a scene from one reference system to another because this transformation can be described as a combination of translation and rotation

A coordinate transformation of the form

$$\begin{aligned} x' &= a_{xx}x + a_{xy}y + a_{xz}z + b_x \\ y' &= a_{yx}x + a_{yy}y + a_{yz}z + b_y \\ z' &= a_{zx}x + a_{zy}y + a_{zz}z + b_z \end{aligned}$$

is called an **affine transformation**

OpenGL Geometric-Transformation Functions

OpenGL Matrix Stacks

glMatrixMode:

used to select the modelview composite transformation matrix as the target of subsequent OpenGL transformation calls

four modes: modelview, projection, texture, and color

the top matrix on each stack is called the "current matrix".

for that mode. the **modelview matrix stack** is the 4 × 4 composite matrix that combines the viewing transformations and the various geometric transformations that we want to apply to a scene.

OpenGL supports a modelview stack depth of at least 32,

glGetIntegerv (GL_MAX_MODELVIEW_STACK_DEPTH, stackSize);

determine the number of positions available in the modelview stack for a particular implementation of OpenGL.

It returns a single integer value to array **stackSize**

other OpenGL symbolic constants: GL_MAX_PROJECTION_STACK_DEPTH, GL_MAX_TEXTURE_STACK_DEPTH, or GL_MAX_COLOR_STACK_DEPTH.

We can also find out how many matrices are currently in the stack with

glGetIntegerv (GL_MODELVIEW_STACK_DEPTH, numMats);

We have two functions available in OpenGL for processing the matrices in a stack

glPushMatrix ( );

Copy the current matrix at the top of the active stack and store that copy in the second stack position

glPopMatrix ( );

which destroys the matrix at the top of the stack, and the second matrix in the stack becomes the current matrix

Illumination and Color

Illumination Models

Q.Explain the different types of light sources supported by OpenGL

A light source can be defined with a number of properties. We can specify its position, the color of the emitted light, the emission direction, and its shape.

Different Light Sources

1. Point Light Sources
2. Infinitely Distant Light Sources
3. Directional Light Sources and Spotlight Effects
4. Extended Light Sources and the Warn Model

1.Point Light Sources

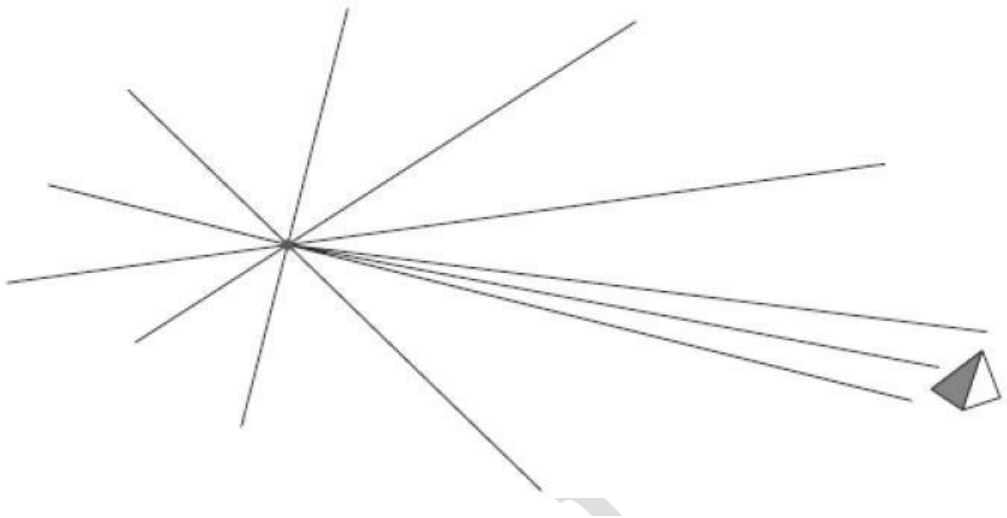
The simplest model for an object that is emitting radiant energy is a **point light source** with a single color, specified with three RGB components

A point source for a scene by giving its position and the color of the emitted light. light rays are generated along radially diverging paths from the single-color source position. This light-source model is a reasonable approximation for sources whose dimensions are small compared to the size of objects in the scene

2. Infinitely Distant Light Sources

A large light source, such as the sun, that is very far from a scene can also be approximated as a point emitter, but there is little variation in its directional effects.

The light path from a distant light source to any position in the scene is nearly constant



We can simulate an infinitely distant light source by assigning it a color value and a fixed direction for the light rays emanating from the source.

The vector for the emission direction and the light-source color are needed in the illumination calculations, but not the position of the source.

3. Directional Light Sources and Spotlight Effects

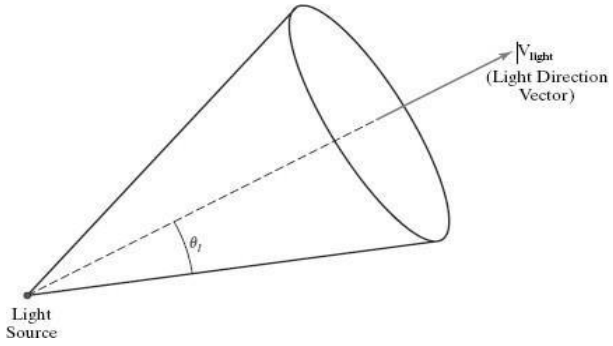
A local light source can be modified easily to produce a directional, or spotlight, beam of light.

If an object is outside the directional limits of the light source, we exclude it from illumination by that source

One way to set up a directional light source is to assign it a vector direction and an angular limit θ_l measured from that vector direction, in addition to its position and color

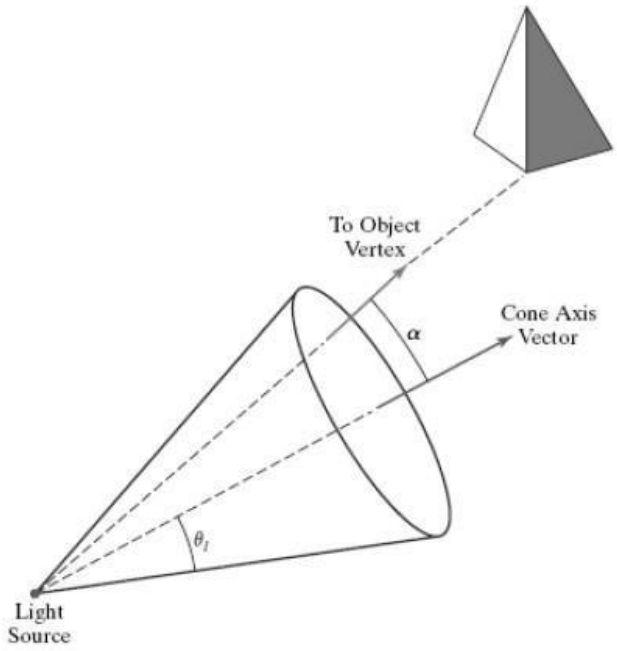
We can denote $\mathbf{V}_{light}$ as the unit vector in the light-source direction and $\mathbf{V}_{obj}$ as the unit vector in the direction from the light position to an object position.

Then $\mathbf{V}_{obj} \cdot \mathbf{V}_{light} = \cos \alpha$



where angle α is the angular distance of the object from the light direction vector.

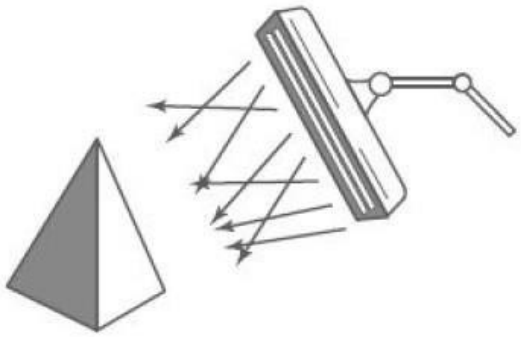
If we restrict the angular extent of any light cone so that $0° < \theta_l \leq 90°$, then the object is within the spotlight if $\cos \alpha \geq \cos \theta_l$, as shown



. If $\mathbf{V}_{obj} \cdot \mathbf{V}_{light} < \cos \theta_l$, however, the object is outside the light cone.

4. Extended Light Sources and the Warn Model

When we want to include a large light source at a position close to the objects in a scene, such as the long neon lamp, we can approximate it as a lightemitting surface



One way to do this is to model the light surface as a grid of directional point emitters.

We can set the direction for the point sources so that objects behind the light-emitting surface are not illuminated.

We could also include other controls to restrict the direction of the emitted light near the edges of the source

The **Warn model** provides a method for producing studio lighting effects using sets of point emitters with various parameters to simulate the barn doors, flaps, and spotlighting controls employed by photographers.

Spotlighting is achieved with the cone of light discussed earlier, and the flaps and barn doors provide additional directional control

Q. Explain basic illumination models

These below models are used to calculate the intensity of light that is reflected at a given point on a surface

- Ambient lighting
- Diffuse reflection
- Specular reflection

Ambient lighting

- This produces a uniform ambient lighting that is the same for all objects, and it approximates the global diffuse reflections from the various illuminatedsurfaces.
- Reflections produced by ambient-light illumination are simply a form of diffuse reflection, and they are independent of the viewing direction and the spatial orientation of a surface.

- The reflected intensity I_{amb} of any point on the surface is:

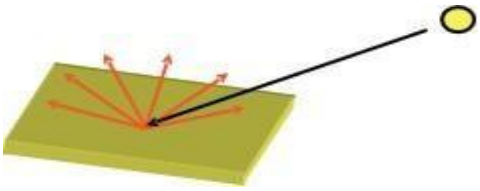
$$I_{amb} = K_a I_a$$

I_a - ambient light intensity
$K_a \in [0,1]$ - surface ambient reflectivity

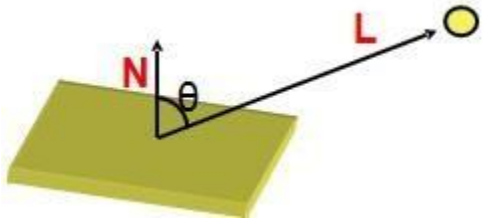
- In principle I_a and K_a are functions of color, so we have I^R_{amb}, I^G_{amb} and I^B_{amb}

Diffuse reflection

- The incident light on the surface is scattered with equal intensity in all directions, independent of the viewing position. Such surfaces are called **ideal diffuse reflectors**
- They are also referred to as **Lambertian reflectors,** because the reflected radiant light energy fromany point on the surface is calculated with **Lambert's cosine law.**
- Diffuse (Lambertian) surfaces are rough or grainy, like clay, soil, fabric
- The surface appears equally bright from all viewing directions



- The brightness at each point is proportional to cos(θ)



- Brightness is proportional to cos(θ) because a surface (a) perpendicular to the light direction is more illuminated than a surface (b) at an oblique angle

- The reflected intensity I_{diff} of a point on the surface is:

$$I_{diff} = K_d I_p \cos(\theta) = K_d I_p (N \cdot L)$$

I_p - the point light intensity. May appear as attenuated source $f_{att}(r) I_P$
$K_d \in [0,1]$ - the surface diffuse reflectivity
N - the surface normal
L - the light direction

NOTE: If N and L have unitary length: $\cos(\theta) = N \cdot L$

- Commonly, there are two types of light sources:
  - A background ambient light
  - A point light source

- The equation that combines the two models is:

$$I = I_{diff} + I_{amb} = K_d I_p N \cdot L + K_a I_a$$

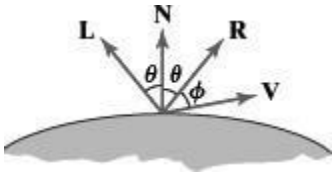
Specular reflection

Q.Explain Specular Reflection and Phong model

Q.Describe phone lighting model

Specular reflection

- The bright spot, or specular reflection, that we can see on a shiny surface is the result of total, or near total, reflection of the incident light in a concentrated region around the specular-reflection angle.
- The below figure shows the specular reflection direction for a position on an illuminated surface



N represents: unit normal surface vector The specular reflection angle equals the angle of the incident light, with the two angles measured on opposite sides of the unit normal surface vector **N**

Rrepresents the unit vector in the direction of ideal specular reflection,

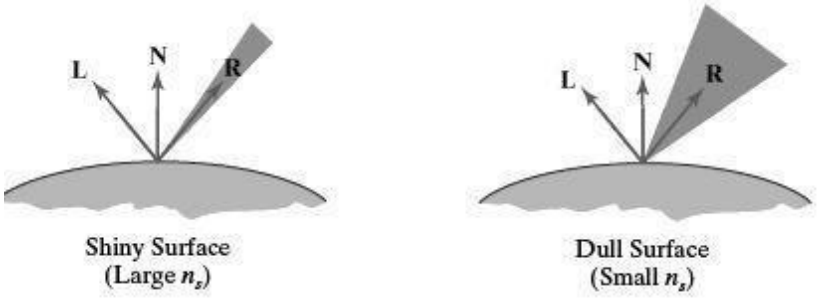
L is the unit vector directed toward the point light source, and

Vis the unit vector pointing to the viewer fromthe selected surface position.

Angle φ is the viewing angle relative to the specular-reflection direction **R**

Phong model

- An empirical model for calculating the specular reflection range, developed by Phong Bui Tuong and called the Phong specular-reflection model or simply the Phon G model, sets the intensity of specular reflection proportional to $\cos^{ns} \varphi$
- Angle φ can be assigned values in the range 0◦ to 90◦, so that cos φ varies from 0 to 1.0.
- The value assigned to the specular-reflection exponent *ns* is determined by the type of surface that we want to display.
    - A very shiny surface is modeled with a large value for *ns* (say, 100 or more)
    - Smaller values (down to 1) are used for duller surfaces.
    - For a perfect reflector, *ns* is infinite.
    - For a rough surface, such as chalk or cinderblock, *ns* is assigned a value near 1.



- We can approximately model monochromatic specular intensity variations using a specular-reflection coefficient, W(θ), for each surface.
- W(θ) tends to increase as the angle of incidence increases.
- At θ = 90◦, all the incident light is reflected (W(θ) = 1)
- Using the spectral-reflection function W(θ), we can write the Phong specular-reflection model as.

$$I_{l,\text{spec}} = W(\theta) I_l \cos^{n_s} \phi$$

where I_l is the intensity of the light source, and φ is the viewing angle relative to the specular-reflection direction **R**.

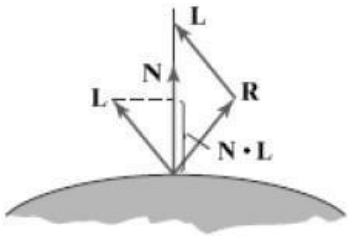
Because**V**and **R** are unit vectors in the viewing and specular-reflection directions, we can calculate the value of cos φ with the dot product $\mathbf{V}\cdot\mathbf{R}$.

In addition, no specular effects are generated for the display of a surface if **V** and **L** are on the same side of the normal vector **N** or if the light source is behind the surface

We can determine the intensity of the specular reflection due to a point light source at a surface position with the calculation

$$I_{l,\text{spec}} = \begin{cases} k_s I_l (\mathbf{V}\cdot\mathbf{R})^{n_s}, & \text{if } \mathbf{V}\cdot\mathbf{R} > 0 \quad \text{and} \quad \mathbf{N}\cdot\mathbf{L} > 0 \\ 0.0, & \text{if } \mathbf{V}\cdot\mathbf{R} \le 0 \quad \text{or} \quad \mathbf{N}\cdot\mathbf{L} \le 0 \end{cases}$$

The direction for **R**, the reflection vector, can be computed from the directions for vectors **L** and **N**.



The projection of **L** onto the direction of the normal vector has a magnitude equal to the dot product$\mathbf{N}\cdot\mathbf{L}$, which is also equal to the magnitude of the projection of unit vector **R** onto the direction of **N**.

Therefore, from this diagram, we see that

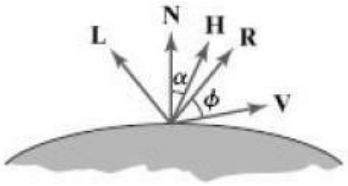
$$\mathbf{R} + \mathbf{L} = (2\mathbf{N}\cdot\mathbf{L})\mathbf{N}$$

and the specular-reflection vector is obtained as

$$\mathbf{R} = (2\mathbf{N}\cdot\mathbf{L})\mathbf{N} - \mathbf{L}$$

A somewhat simplified Phong model is obtained using the **halfway vector H** between **L** and **V** to calculate the range of specular reflections.

If we replace **V·R** in the Phong model with the dot product **N·H**, this simply replaces the empirical cos φ calculation with the empirical cos α calculation



The halfway vector is obtained as

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{|\mathbf{L} + \mathbf{V}|}$$

For nonplanar surfaces, **N·H** requires less computation than **V·R** because the calculation of **R** at each surface point involves the variable vector **N**.

OpenGL Illumination Functions

Q.Explain OpenGL illumination functions

OpenGL Point Light-Source Function

glLight* (lightName, lightProperty, propertyValue);

A suffix code of **i** or **f** is appended to the function name, depending on the data type of the property value

lightName: GL_LIGHT0, GL_LIGHT1, GL_LIGHT2, . . . , GL_LIGHT7

lightProperty: must be assigned one of the OpenGL symbolic property constants

glEnable (lightName);→ turn on that light with the command

glEnable (GL_LIGHTING);→ activate the OpenGL lighting routines

Specifying an OpenGL Light-Source Position and Type

GL_POSITION:

specifies light-source position

this symbolic constant is used to set two light-source properties at the same time: the light-source position and the *light-source type*

Specifying OpenGL Light-Source Colors

Unlike an actual light source, an OpenGL light has three different color properties the symbolic color-property constants **GL_AMBIENT, GL_DIFFUSE,** and **GL_SPECULAR**

Example:

```
GLfloat blackColor [ ] = {0.0, 0.0, 0.0, 1.0};
GLfloat whiteColor [ ] = {1.0, 1.0, 1.0, 1.0};
glLightfv (GL_LIGHT3, GL_AMBIENT, blackColor);
glLightfv (GL_LIGHT3, GL_DIFFUSE, whiteColor);
glLightfv (GL_LIGHT3, GL_SPECULAR, whiteColor);
```

Specifying Radial-Intensity Attenuation Coefficients

For an OpenGL Light Source we could assign the radial-attenuation coefficient values as

```
glLightf (GL_LIGHT6, GL_CONSTANT_ATTENUATION, 1.5);
glLightf (GL_LIGHT6, GL_LINEAR_ATTENUATION, 0.75);
glLightf (GL_LIGHT6, GL_QUADRATIC_ATTENUATION, 0.4);
```

OpenGL Directional Light Sources (Spotlights)

There are three OpenGL property constants for directional effects: **GL_SPOT_DIRECTION, GL_SPOT_CUTOFF**, and **GL_SPOT_EXPONENT**

```
GLfloat dirVector [ ] = {1.0, 0.0, 0.0};
glLightfv (GL_LIGHT3, GL_SPOT_DIRECTION, dirVector);
glLightf (GL_LIGHT3, GL_SPOT_CUTOFF, 30.0);
glLightf (GL_LIGHT3, GL_SPOT_EXPONENT, 2.5);
```

OpenGL Global Lighting Parameters

glLightModel* (paramName, paramValue);

We append a suffix code of i or f, depending on the data type of the parameter value.

In addition, for vector data, we append the suffix code v.

Parameter paramName is assigned an OpenGL symbolic constant that identifies the global property to be set, and parameter paramValue is assigned a single value or set of values.

```
globalAmbient [ ] = {0.0, 0.0, 0.3, 1.0);
glLightModelfv (GL_LIGHT_MODEL_AMBIENT, globalAmbient);
```

glLightModeli (GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);

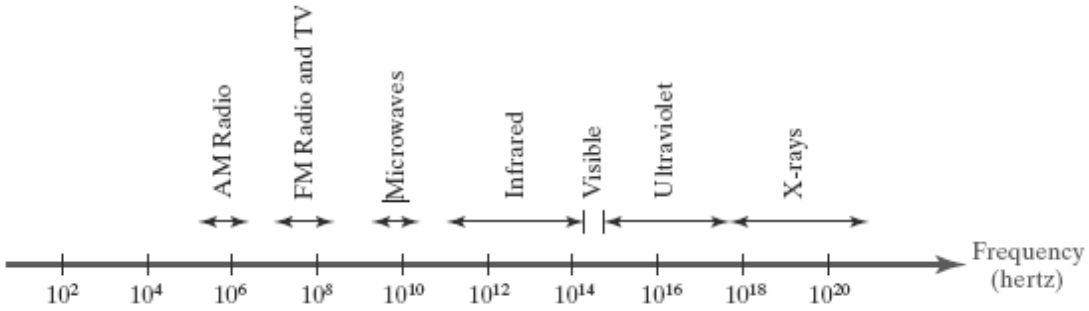
turn off this default and use the actual viewing position (which is the viewing-coordinate origin) to calculate V

Color Models

Properties of Light

The Electromagnetic Spectrum

- Color is electromagnetic radiation within a narrow frequency band.
- Some of the other frequency groups in the electromagnetic spectrum are referred to as radio waves, microwaves, infrared waves, and X-rays. The frequency is shown below



- Each frequency value within the visible region of the electromagnetic spectrum corresponds to a distinct **spectral color.**
- At the low-frequency end (approximately 3.8×1014 hertz) are the red colors, and at the high-frequency end (approximately 7.9 × 1014 hertz) are the violet colors.
- In the wave model of electromagnetic radiation, light can be described as oscillating transverse electric and magnetic fields propagating through space.
- The electric and magnetic fields are oscillating in directions that are perpendicular to each other and to the direction of propagation.
- For one spectral color (a monochromatic wave), the wavelength and frequency are inversely proportional to each other, with the proportionality constant as the speed of light (c):

$$c = \lambda f$$

- When white light is incident upon an opaque object, some frequencies are reflected and some are absorbed.
- If low frequencies are predominant in the reflected light, the object is described as red. In this case, we say that the perceived light has a dominant frequency (or dominant wavelength) at the red end of the spectrum.

- The dominant frequency is also called the **hue,** or simply the **color,** of the light.

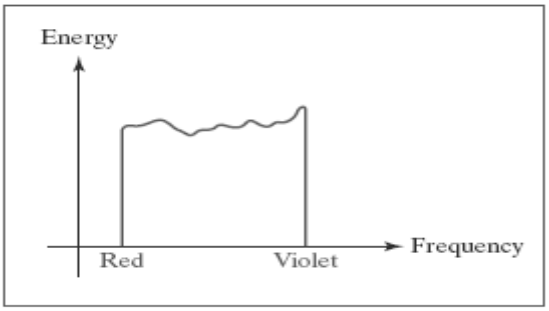
Psychological Characteristics of Color

Other properties besides frequency are needed to characterize our perception of Light

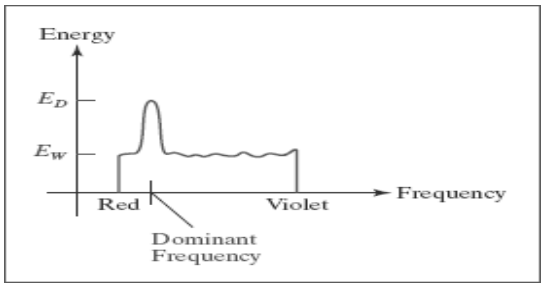
- **Brightness:** which corresponds to the total light energy and can be quantified as the luminance of the light.
- **Purity**, or the **saturation** of the light: Purity describes how close a light appears to be to a pure spectral color, such as red.
- **Chromaticity**, is used to refer collectively to the two properties describing color characteristics: purity and dominant frequency (hue).

We can calculate the brightness of the source as the area under the curve, which gives the total energy density emitted.

Purity (saturation) depends on the difference between ED and EW. Below figure shows Energy distribution for a white light source



- Below figure shows, Energy distribution for a light source with a dominant frequency near the red end of the frequency range.



Color Models

Q.Define color model. With neat diagram explain RGB and CYM color model

Any method for explaining the properties or behavior of color within some particular context is called a **color model.**

Primary Colors

- The hues that we choose for the sources are called the primary colors, and the color gamut for the model is the set of all colors that we can produce from the primary colors.
- Two primaries that produce white are referred to as complementary colors.
- Examples of complementary color pairs are red and cyan, green and magenta, and blue and yellow

Intuitive Color Concepts

- An artist creates a color painting by mixing color pigments with white and black pigments to form the various shades, tints, and tones in the scene.
- Starting with the pigment for a "pure color" ("pure hue"), the artist adds a black pigment to produce different shades of that color.
- Tones of the color are produced by adding both black and white pigments.

Q.With the help of a suitable diagram, explain RGB and CMY color models.

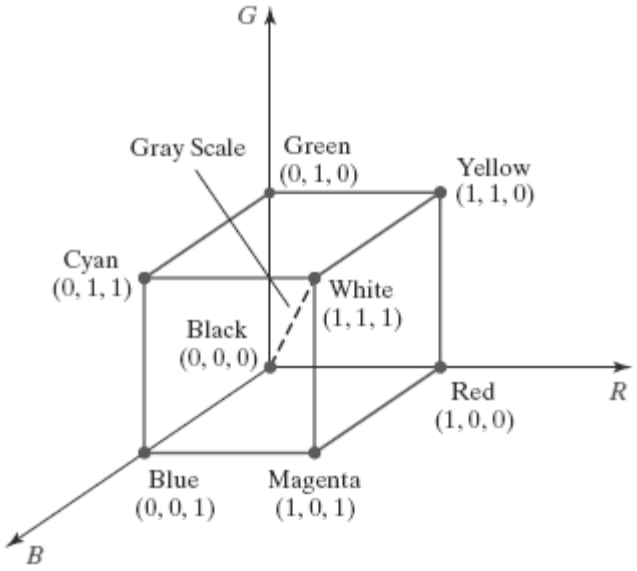
The RGB Color Model

According to the *tristimulus theory* of vision, our eyes perceive color through the stimulation of three visual pigments in the cones of the retina.

One of the pigments is most sensitive to light with a wavelength of about 630 nm (red), another has its peak sensitivity at about 530 nm (green), and the third pigment is most receptive to light with a wavelength of about 450 nm (blue).

The three primaries red, green, and blue, which is referred to as the *RGB color model.*

We can represent this model using the unit cube defined on *R*, *G*, and *B* axes, as shown in Figure



The origin represents black and the diagonally opposite vertex, with coordinates (1, 1, 1), is white the RGB color scheme is an additive model.

Each color point within the unit cube can be represented as a weighted vector sum of the primary colors, using unit vectors **R**, **G**, and **B**:

$$C(\lambda) = (R, G, B) = R\mathbf{R} + G\,\mathbf{G} + B\,\mathbf{B}$$

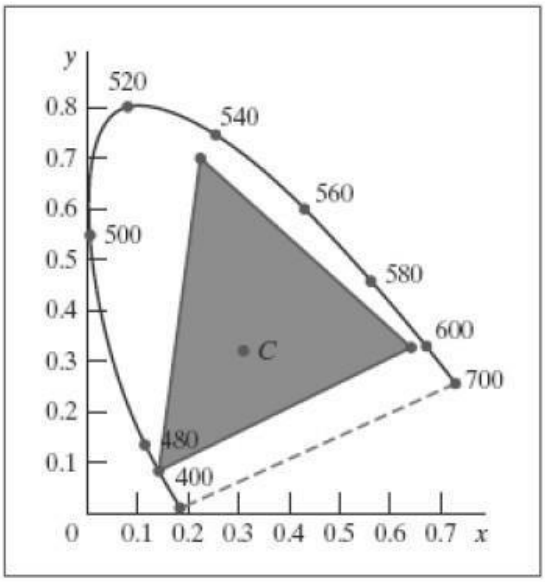
where parameters R, G, and B are assigned values in the range from 0 to 1.0

Chromaticity coordinates for the National Television System Committee (NTSC) standard RGB phosphors are listed in Table

RGB (x, y) Chromaticity Coordinates

	NTSC Standard	CIE Model	Approx. Color Monitor Values
R	(0.670, 0.330)	(0.735, 0.265)	(0.628, 0.346)
G	(0.210, 0.710)	(0.274, 0.717)	(0.268, 0.588)
B	(0.140, 0.080)	(0.167, 0.009)	(0.150, 0.070)

Below figure shows the approximate color gamut for the NTSC standard RGB primaries

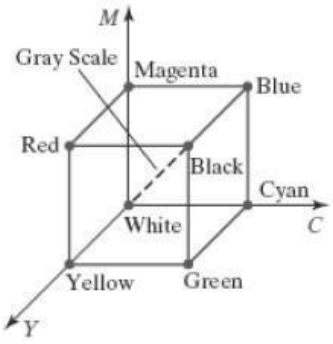


The CMY and CMYK Color Models

The CMY Parameters

A subtractive color model can be formed with the three primary colors cyan, magenta, and yellow

A unit cube representation for the CMY model is illustrated in Figure



In the CMY model, the spatial position (1, 1, 1) represents black, because all components of the incident light are subtracted.

The origin represents white light.

Equal amounts of each of the primary colors produce shades of gray along the main diagonal of the cube.

A combination of cyan and magenta ink produces blue light, because the red and green components of the incident light are absorbed.

Similarly, a combination of cyan and yellow ink produces green light, and a combination of magenta and yellow ink yields red light.

The CMY printing process often uses a collection of four ink dots, which are arranged in a close pattern somewhat as an RGB monitor uses three phosphor dots.

Thus, in practice, the CMY color model is referred to as the CMYK model, where K is the black color parameter.

One ink dot is used for each of the primary colors (cyan, magenta, and yellow), and one ink dot is black

Q.Explain the transformation between CMY and RGB color space

We can express the conversion from an RGB representation to a CMY representation using the following matrix transformation:

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Where the white point in RGB space is represented as the unit column vector.

And we convert from a CMY color representation to an RGB representation using the matrix transformation

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix}$$

MODULE-4

3D Viewing and Visible Surface Detection

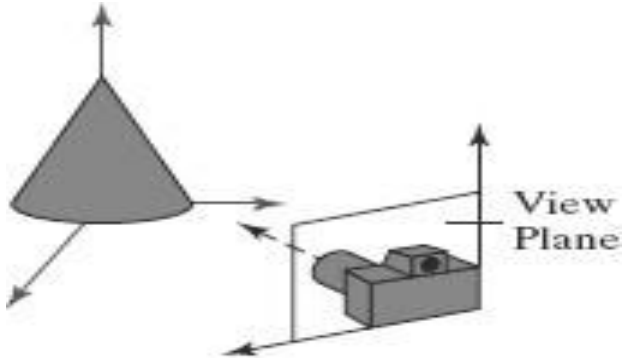
3D Viewing

Q.What is 3D Viewing? With the help of a block diagram explain 3D viewing pipeline architecture

- When we model a three-dimensional scene, each object in the scene is typically defined with a set of surfaces that form a closed boundary around the object interior.
- In addition to procedures that generate views of the surface features of an object, graphics packages sometimes provide routines for displaying internal components or cross sectional views of a solid object.
- Many processes in three-dimensional viewing, such as the clipping routines, are similar to those in the two-dimensional viewing pipeline.
- But three-dimensional viewing involves some tasks that are not present in two dimensional Viewing

Viewing a Three-Dimensional Scene

- To obtain a display of a three-dimensional world-coordinate scene, we first set up a coordinate reference for the viewing, or "camera," parameters.
- This coordinate reference defines the position and orientation for a view plane (or projection plane) that corresponds to a camera film plane



The Three-Dimensional Viewing Pipeline

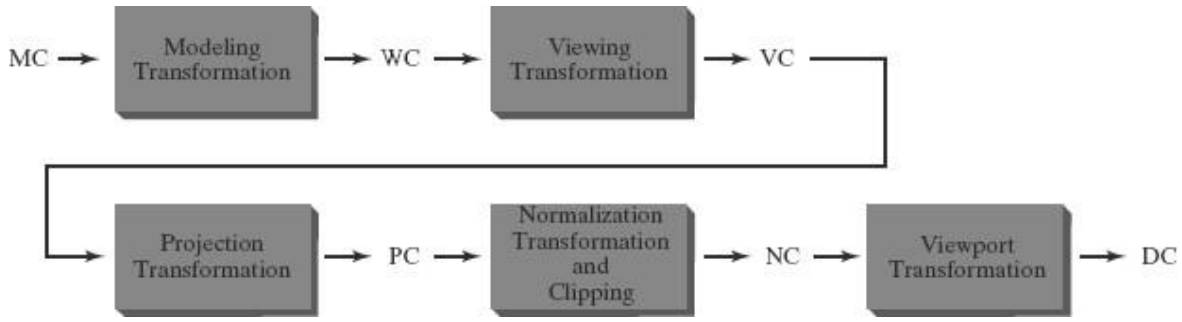
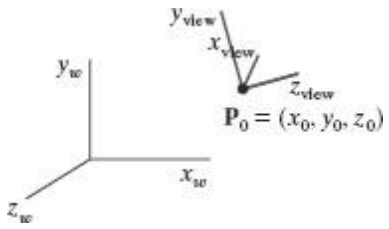


Figure above shows the general processing steps for creating and transforming a three dimensional scene to device coordinates.

- Construct the shape of individual objects in a scene within modeling coordinate, and place the objects into appropriate positions within the scene (world coordinate).
- World coordinate positions are converted to viewing coordinates.
- Convert the viewing coordinate description of the scene to coordinate positions on the projection plane.
- A two-dimensional clipping window, corresponding to a selected camera lens, is defined on the projection plane, and a three-dimensional clipping region is established. This clipping region is called the view volume.
- Objects are mapped to normalized coordinates, and all parts of the scene outside the view volume are clipped off.
- The clipping operations can be applied after all device-independent coordinate transformations. Then viewport is specified in device coordinates and that normalized coordinates are transferred to viewport coordinates, following the clipping operations.
- The final step is to map viewport coordinates to device coordinates within a selected display window

Three-Dimensional Viewing-Coordinate Parameters

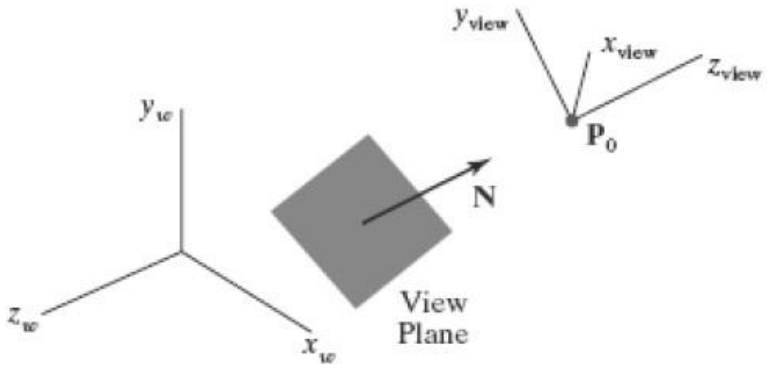
- Select a world-coordinate position $P_0 = (x_0, y_0, z_0)$ for the viewing origin, which is called the view point or viewing position and we specify a view-up vector V, which defines the y_{view} direction.
- Figure below illustrates the positioning of a three-dimensional viewing-coordinate frame within a world system.



The View-Plane Normal Vector

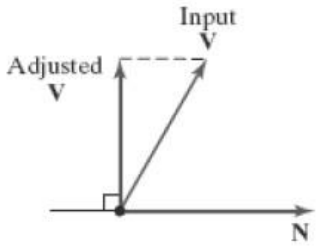
Because the viewing direction is usually along the *z*view axis, the view plane, also called the projection plane, is normally assumed to be perpendicular to this axis.

Thus, the orientation of the view plane, as well as the direction for the positive *z*view axis, can be definedwith a view-plane normal vector N,



View-Up Vector

- Once we have chosen a view-plane normal vector N, we can set the direction for the view-up vector V.
- This vector is used to establish the positive direction for the y_{view} axis.
- Usually, V is defined by selecting a position relative to the world-coordinate origin, so that the direction for the view-up vector is from the world origin to this selected position



- Because the view-plane normal vector N defines the direction for the z_{view} axis, vector V should be perpendicular to N.
- But, in general, it can be difficult to determine a direction for V that is precisely perpendicular to N.

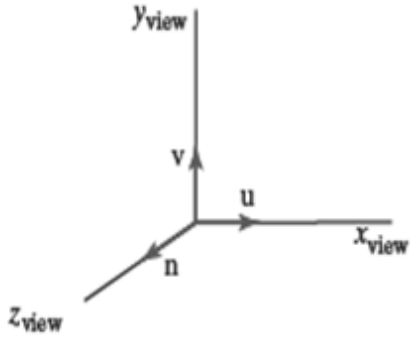
The uvn Viewing-Coordinate Reference Frame

- The coordinate system formed with these unit vectors is often described as a uvn viewing-coordinate reference frame

$$\mathbf{n} = \frac{\mathbf{N}}{|\mathbf{N}|} = (n_x, n_y, n_z)$$

$$\mathbf{u} = \frac{\mathbf{V} \times \mathbf{n}}{|\mathbf{V} \times \mathbf{n}|} = (u_x, u_y, u_z)$$

$$\mathbf{v} = \mathbf{n} \times \mathbf{u} = (v_x, v_y, v_z)$$

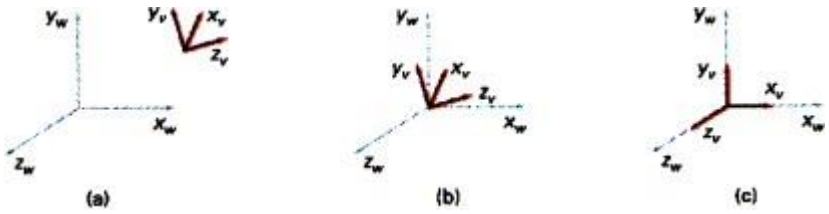


Transformation from World to Viewing Coordinates

Q.Design the transformation matrix from world to viewing coordinate system with matrix representation

In the three-dimensional viewing pipeline, the first step after a scene has been constructed is to transfer object descriptions to the viewing-coordinate reference frame.

This conversion of object descriptions is equivalent to a sequence of transformations that superimposes the viewing reference frame onto the world frame

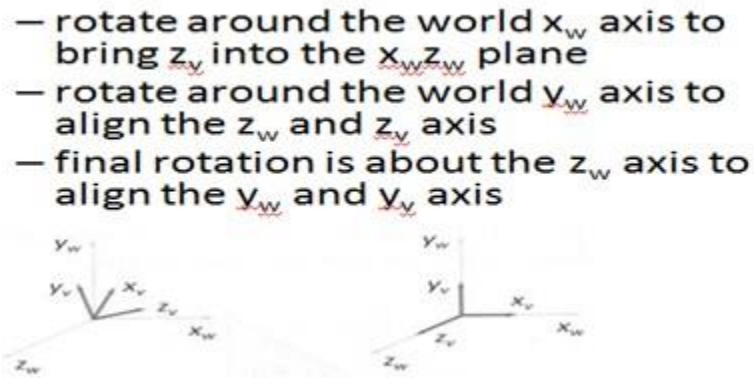


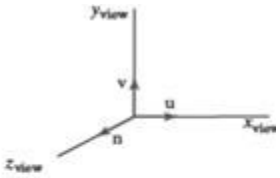
Transformation sequences

1. Translate the view reference point to the origin of the WC system (Figure b)

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. Apply rotations to align the x_v, y_v, and z_v axes with the world x_w, y_w,and z_w axes, respectively.



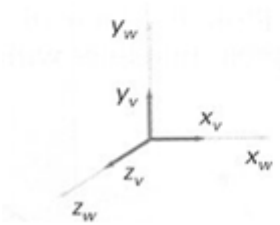


- Rotation by *uvn* system
  - Calculate unit *uvn* vectors
    - N : view-plane normal vector
    - V : view-up vector
    - U : perpendicular to both N and V

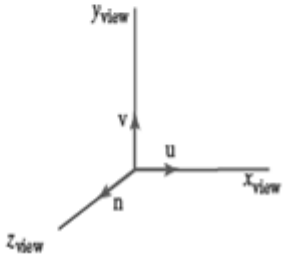
$$\mathbf{n} = \frac{\mathbf{N}}{|\mathbf{N}|} = (n_1, n_2, n_3)$$

$$\mathbf{u} = \frac{\mathbf{V} \times \mathbf{N}}{|\mathbf{V} \times \mathbf{N}|} = (u_1, u_2, u_3)$$

$$\mathbf{v} = \mathbf{n} \times \mathbf{u} = (v_1, v_2, v_3)$$



$$\mathbf{R} = \begin{bmatrix} u_1 & u_2 & u_3 & 0 \\ v_1 & v_2 & v_3 & 0 \\ n_1 & n_2 & n_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



The coordinate transformation matrix is then obtained as the product of the preceding translation and rotation matrices:

$$\mathbf{R} = \begin{bmatrix} u_1 & u_2 & u_3 & 0 \\ v_1 & v_2 & v_3 & 0 \\ n_1 & n_2 & n_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{M}_{WC,\,VC} = \mathbf{R} \cdot \mathbf{T} = \begin{bmatrix} u_x & u_y & u_z & -\mathbf{u} \cdot \mathbf{P}_0 \\ v_x & v_y & v_z & -\mathbf{v} \cdot \mathbf{P}_0 \\ n_x & n_y & n_z & -\mathbf{n} \cdot \mathbf{P}_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

These matrix elements are evaluated as

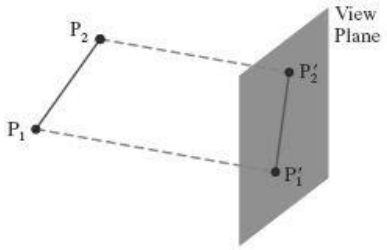
$$-\mathbf{u} \cdot \mathbf{P}_0 = -x_0 u_x - y_0 u_y - z_0 u_z$$
$$-\mathbf{v} \cdot \mathbf{P}_0 = -x_0 v_x - y_0 v_y - z_0 v_z$$
$$-\mathbf{n} \cdot \mathbf{P}_0 = -x_0 n_x - y_0 n_y - z_0 n_z$$

Projection Transformations

Graphics packages generally support both parallel and perspective projections.

Parallel Projection

- Parallel Projection transforms object positions to the view plane along parallel lines.
- A parallel projection preserves relative proportions of objects, and this is the method used in computer aided drafting and design to produce scale drawings of three-dimensional objects.
- All parallel lines in a scene are displayed as parallel when viewed with a parallel projection.



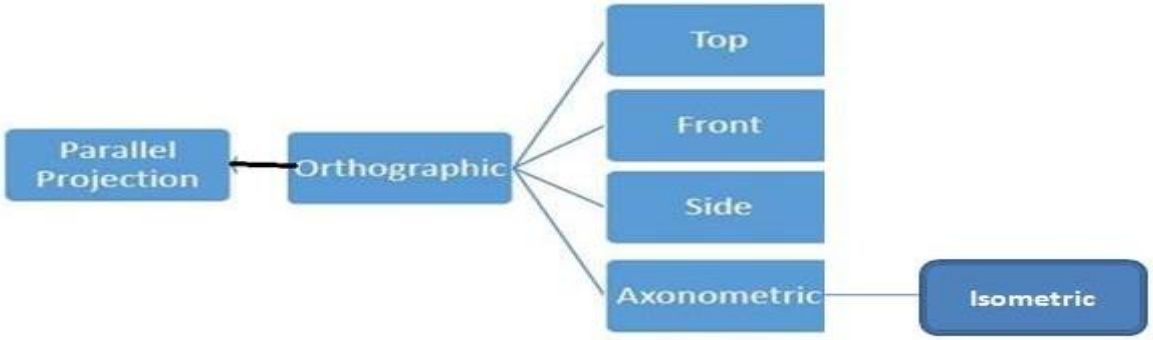
- Parallel Projection Classification: Orthographic Parallel Projection and Oblique Projection
  - Orthographic parallel projections are done by projecting points along parallel lines that are perpendicular to the projection plane.
  - Oblique projections are obtained by projecting along parallel lines that are NOT perpendicular to the projection plane.

Orthogonal Projections

Q. Explain Orthogonal Projection in details

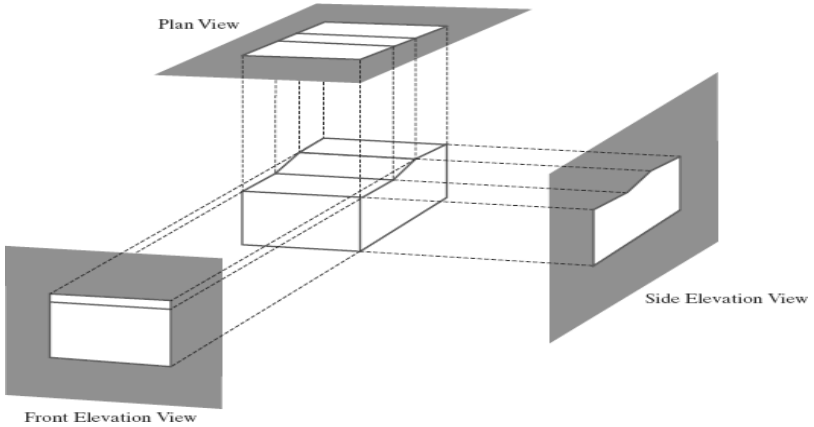
A transformation of object descriptions to a view plane along lines that are all parallel to the view-plane normal vector N is called an orthogonal projection also termed as orthographic projection

Classification of Orthographic Projection



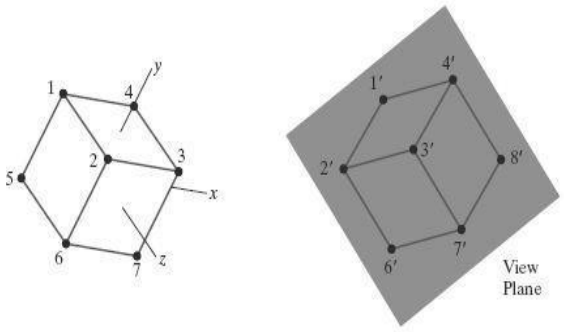
- Different types of orthographic projections
  - Front Projection
  - Top Projection
  - Side Projection
  - Axonometric Projection – Isometric

- Front, side, and rear orthogonal projections of an object are called elevations
- Top orthogonal projection is called a plan view



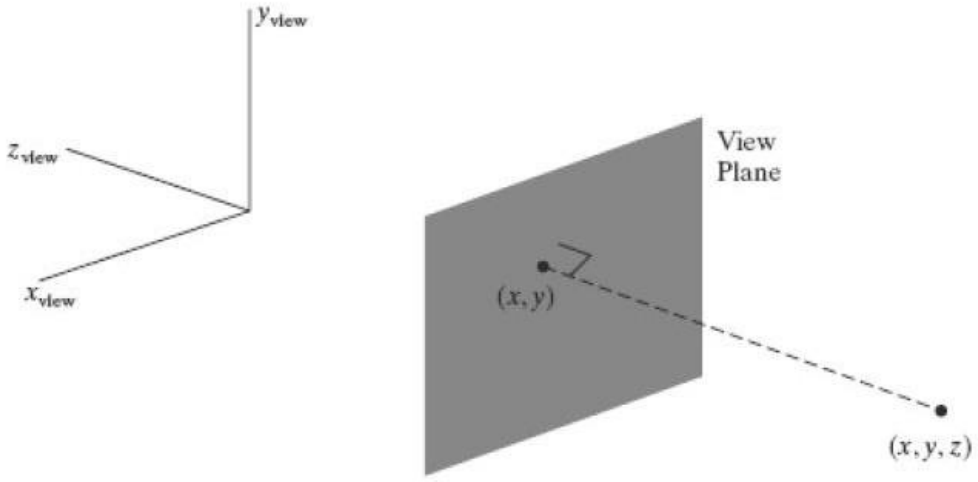
- Axonometric orthogonal projections
  - We can also form orthogonal projections that display more than one face of an object, such views are called Axonometric projection

  - The most commonly used axonometric projection is the isometric projection, which is generated by aligning the projection plane (or the object) so that the plane intersects each coordinate axis in which the object is defined, called the principal axes, at the same distance from the origin



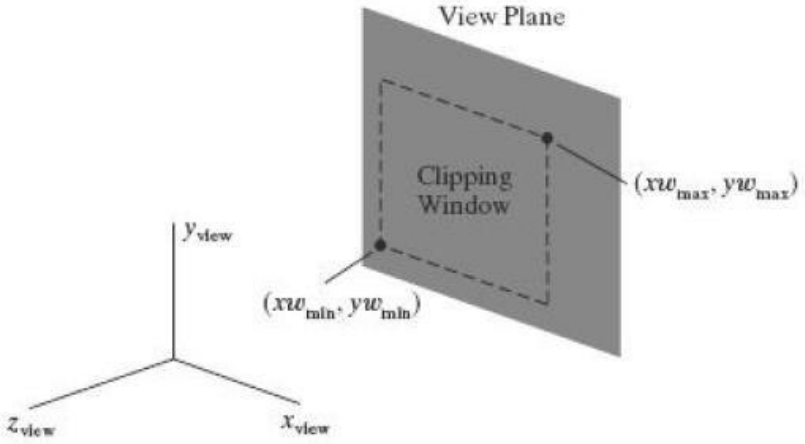
Orthogonal Projection Coordinates

With the projection direction parallel to the z_{view} axis, the transformation equations for an orthogonal projection are trivial. For any position (x, y, z) in viewing coordinates, as in Figure below, the projection coordinates are $x_p = x$, $y_p = y$

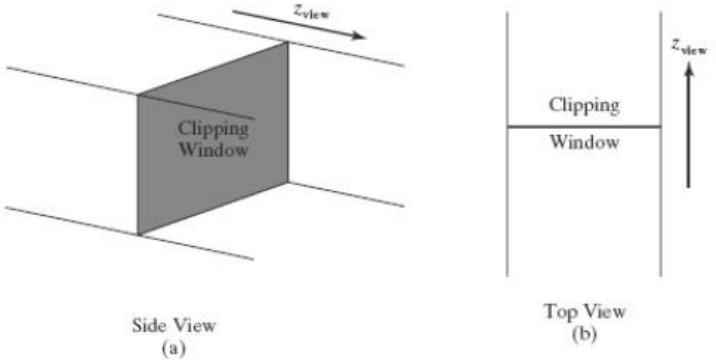


Clipping Window and Orthogonal-Projection View Volume

For three-dimensional viewing, the clipping window is positioned on the view plane with its edges parallel to the x_{view} and y_{view} axes, as shown in Figure below . If we want to use some other shape or orientation for the clipping window, we must develop our own viewing procedures



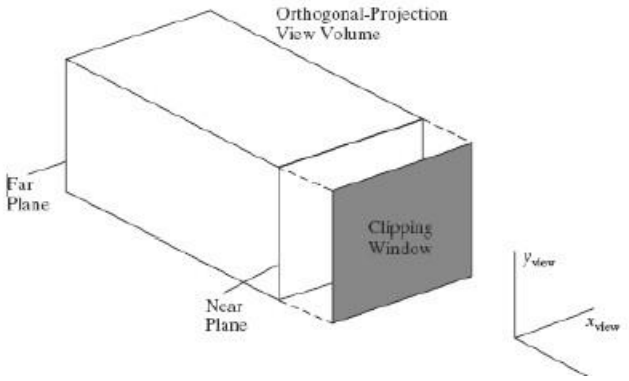
Because projection lines are perpendicular to the view plane, these four boundaries are planes that are also perpendicular to the view plane and that pass through the edges of the clipping window to form an infinite clipping region, as in Figure below.



These two planes are called the near-far clipping planes, or the front-back clipping planes.

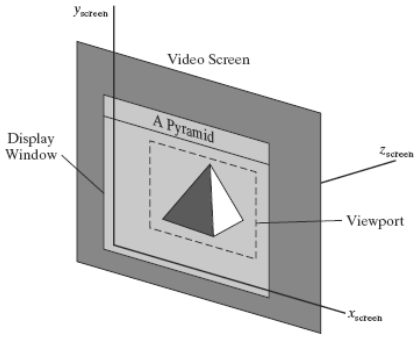
The near and far planes allow us to exclude objects that are in front of or behind the part of the scene that we want to display.

When the near and far planes are specified, we obtain a finite orthogonal view volume that is a *rectangular parallelepiped*, as shown in Figure below along with one possible placement for the view plane



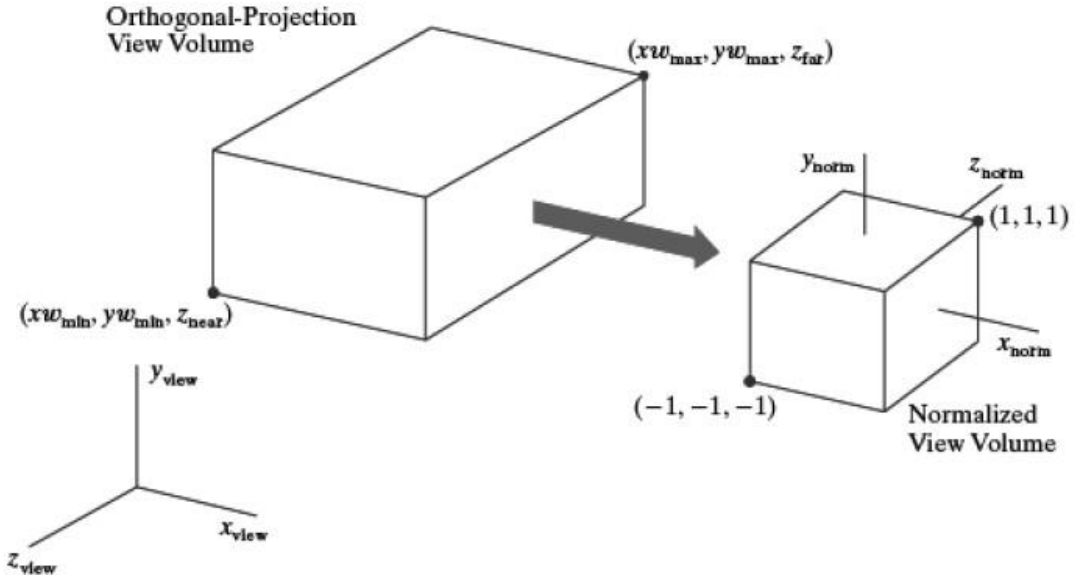
Normalization Transformation for an Orthogonal Projection

- Once we have established the limits for the view volume, coordinate descriptions inside this rectangular parallelepiped are the projection coordinates, and they can be mapped into a normalized view volume without any further projection processing.
- Some graphics packages use a unit cube for this normalized view volume, with each of the *x*, *y*, and *z* coordinates normalized in the range from 0 to 1.
- Another normalization-transformation approach is to use a symmetric cube, with coordinates in the range from −1 to 1



To illustrate the normalization transformation, we assume that the orthogonal-projection view volume is to be mapped into the symmetric normalization cube within a left-handed reference frame.

Also, z-coordinate positions for the near and far planes are denoted as $z$near and $z$far, respectively. Figure below illustrates this normalization transformation

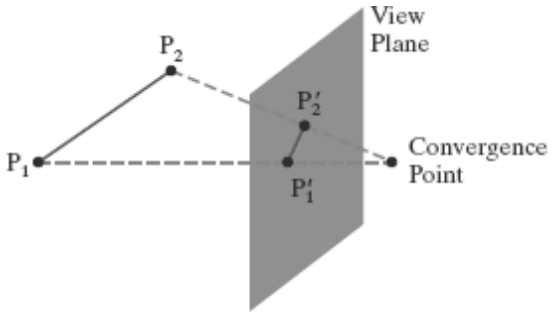


The normalization transformation for the orthogonal view volume is

$$\mathbf{M}_{ortho,norm} = \begin{bmatrix} \frac{2}{xw_{max} - xw_{min}} & 0 & 0 & -\frac{xw_{max} + xw_{min}}{xw_{max} - xw_{min}} \\ 0 & \frac{2}{yw_{max} - yw_{min}} & 0 & -\frac{yw_{max} + yw_{min}}{yw_{max} - yw_{min}} \\ 0 & 0 & \frac{-2}{z_{near} - z_{far}} & \frac{z_{near} + z_{far}}{z_{near} - z_{far}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective Projection

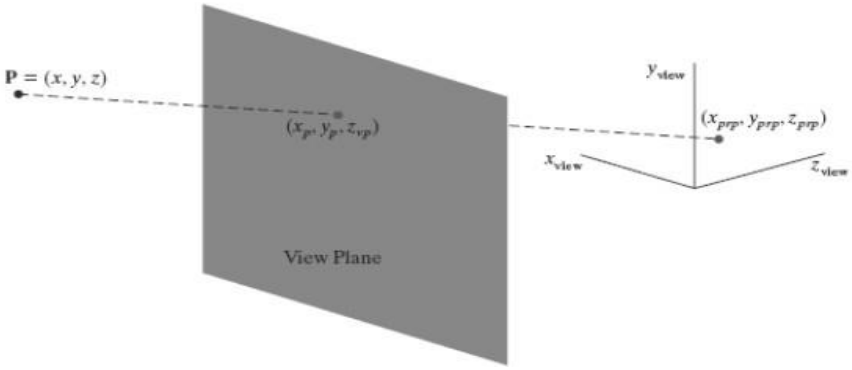
- Perspective Projection transforms object positions to the view plane while converging to a center point of projection.
- Perspective projection produces realistic views but does not preserve relative proportions.
- Projections of distant objects are smaller than the projections of objects of the same size that are closer to the projection plane.



Perspective-Projection Transformation Coordinates

Q. Explain in detail perspective projection transformation coordinates

Figure below shows the projection path of a spatial position (x, y, z) to a general projection reference point at $(x_{prp}, y_{prp}, z_{prp})$.



The projection line intersects the view plane at the coordinate position (x_p, y_p, z_{vp}), where z_{vp} is some selected position for the view plane on the z_{view} axis.

We can write equations describing coordinate positions along this perspective-projection line in parametric form as

$$x' = x - (x - x_{prp})u$$
$$y' = y - (y - y_{prp})u \qquad 0 \le u \le 1$$
$$z' = z - (z - z_{prp})u$$

On the view plane, $z' = z_{vp}$ and we can solve the z' equation for parameter u at this position along the projection line:

$$u = \frac{z_{vp} - z}{z_{prp} - z}$$

Substituting this value of u into the equations for x' and y', we obtain the general perspective-transformation equations

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) + x_{prp}\left(\frac{z_{vp} - z}{z_{prp} - z}\right)$$
$$y_p = y\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) + y_{prp}\left(\frac{z_{vp} - z}{z_{prp} - z}\right)$$

Perspective-Projection Equations: Special Cases

Case 1:

To simplify the perspective calculations, the projection reference point could be limited to positions along the zview axis, then

$x_{prp} = y_{prp} = 0$:

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right), \qquad y_p = y\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right)$$

Case 2:

Sometimes the projection reference point is fixed at the coordinate origin, and

$(x_{prp}, y_{prp}, z_{prp}) = (0, 0, 0)$:

$$x_p = x\left(\frac{z_{vp}}{z}\right), \qquad y_p = y\left(\frac{z_{vp}}{z}\right)$$

Case 3:

If the view plane is the *uv* plane and there are no restrictions on the placement of the projection reference point, then we have

$z_{vp} = 0$:

$$x_p = x\left(\frac{z_{prp}}{z_{prp} - z}\right) - x_{prp}\left(\frac{z}{z_{prp} - z}\right)$$

$$y_p = y\left(\frac{z_{prp}}{z_{prp} - z}\right) - y_{prp}\left(\frac{z}{z_{prp} - z}\right)$$

Case 4:

With the *uv* plane as the view plane and the projection reference point on the z_{view} axis, the perspective equations are

$x_{prp} = y_{prp} = z_{vp} = 0$:

$$x_p = x\left(\frac{z_{prp}}{z_{prp} - z}\right), \qquad y_p = y\left(\frac{z_{prp}}{z_{prp} - z}\right)$$

Vanishing Points for Perspective Projections

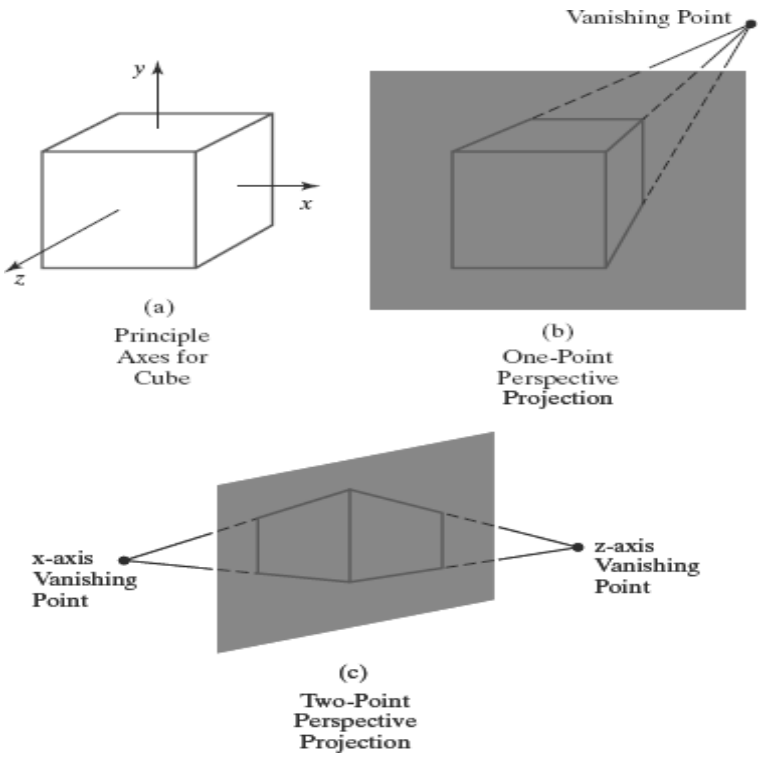
Q.Explain the perspective projection with reference and vanishing point with neat diagram

- The point at which a set of projected parallel lines appears to converge is called a vanishing point.
- Each set of projected parallel lines has a separate vanishing point.

- Based on the number of vanishing points, the perspective projection is of three types
  - One point perspective projection(Figure b)
  - Two point perspective projection(Figure c)

- One point - When the cube is projected to a view plane that intersects only the z axis, a single vanishing point in the z direction

- Two point -When the cube is projected to a view plane that intersects both the z and x axes, two vanishing points are produced.



Perspective-Projection Transformation Matrix

Q.Design a Transformation matrix for perspective projection

- Three-dimensional, homogeneous-coordinate representation to express the perspective-projection equations in the form

$$x_p = \frac{x_h}{h}, \qquad y_p = \frac{y_h}{h}$$

where the homogeneous parameter has the value

$$h = z_{prp} - z$$

$$x_h = x(z_{prp} - z_{vp}) + x_{prp}(z_{vp} - z)$$

$$y_h = y(z_{prp} - z_{vp}) + y_{prp}(z_{vp} - z)$$

- The perspective-projection transformation of a viewing-coordinate position is then accomplished in two steps.

  First, calculate the homogeneous coordinates using the perspective-transformation matrix:

$$\mathbf{P}_h = \mathbf{M}_{\text{pers}} \cdot \mathbf{P}$$

  Where , P_h is the column-matrix representation of the homogeneous point (x_h, y_h, z_h, h)

  P is the column-matrix representation of the coordinate position (x, y, z, 1).

$$\mathbf{M}_{\text{pers}} = \begin{bmatrix} z_{prp} - z_{vp} & 0 & -x_{prp} & x_{prp}z_{prp} \\ 0 & z_{prp} - z_{vp} & -y_{prp} & y_{prp}z_{prp} \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & z_{prp} \end{bmatrix}$$

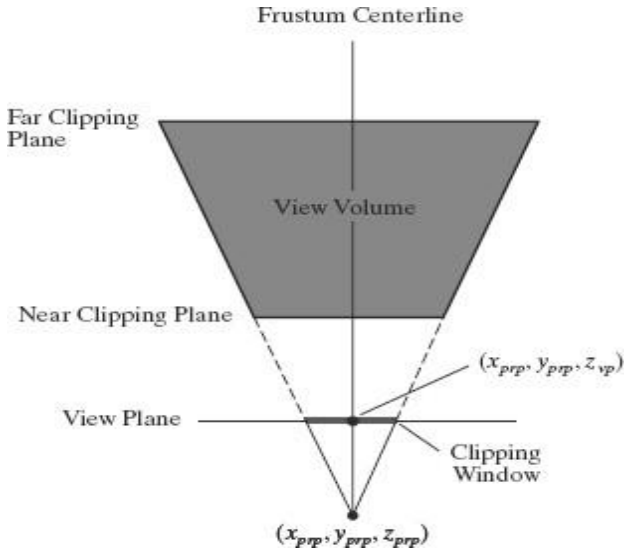
  Second after other processes have been applied, such as the normalization transformation and clipping routines, homogeneous coordinates are divided by parameter *h* to obtain the true transformation-coordinate positions.

Symmetric Perspective-Projection Frustum

Q.Explain in detail symmetric perspective projection Frustum

- The line from the projection reference point through the center of the clipping window and on through the view volume is the centerline for a perspective projection frustum.

- If this centerline is perpendicular to the view plane, we have a symmetric frustum (with respect to its centerline)



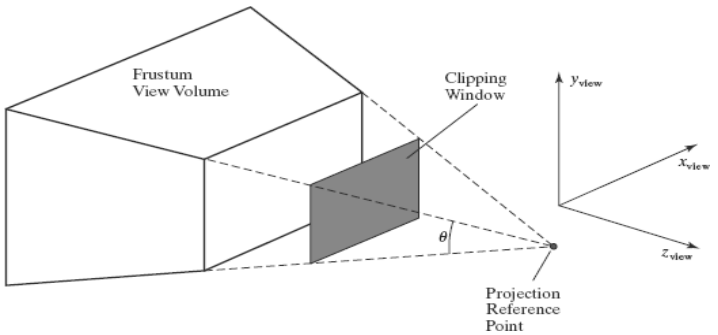
Because the frustum centerline intersects the view plane at the coordinate location $(x_{prp}, y_{prp}, z_{vp})$, we can express the corner positions for the clipping window in terms of the window dimensions:

$$xw_{\min} = x_{prp} - \frac{\text{width}}{2}, \qquad xw_{\max} = x_{prp} + \frac{\text{width}}{2}$$

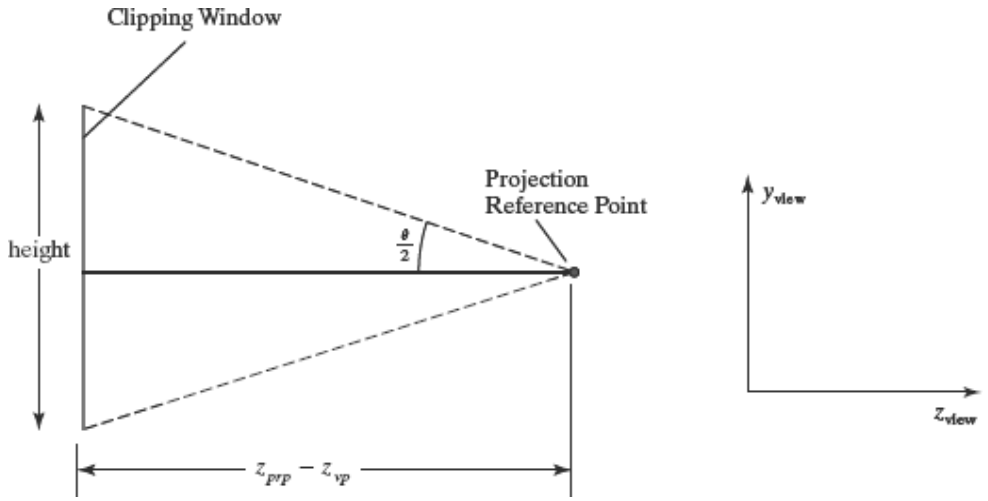
$$yw_{\min} = y_{prp} - \frac{\text{height}}{2}, \qquad yw_{\max} = y_{prp} + \frac{\text{height}}{2}$$

- Another way to specify a symmetric perspective projection is to use parameters that approximate the properties of a camera lens.

  In computer graphics, the cone of vision is approximated with a symmetric frustum, and we can use a field-of-view angle to specify an angular size for the frustum.



For a given projection reference point and view-plane position, the field-of view angle determines the height of the clipping window from the right triangles in the diagram of Figure below,



clipping-window height can be calculated as

$$\text{height} = 2(z_{prp} - z_{vp}) \tan\left(\frac{\theta}{2}\right)$$

Therefore, the diagonal elements with the value $z_{prp} - z_{vp}$ could be replaced by either of the following two expressions

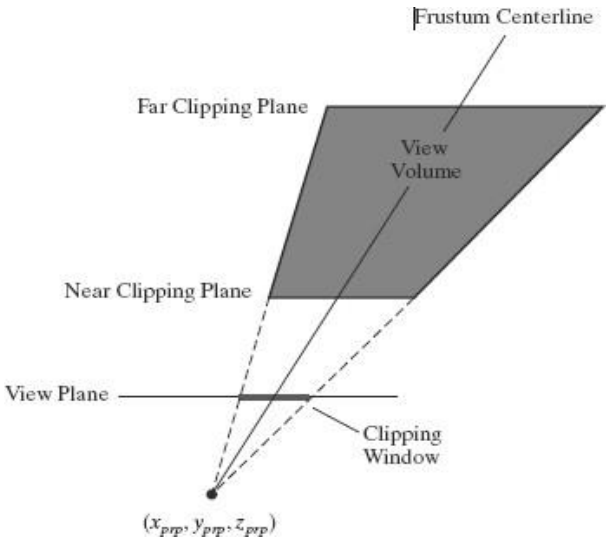
$$z_{prp} - z_{vp} = \frac{\text{height}}{2} \cot\left(\frac{\theta}{2}\right)$$

$$= \frac{\text{width} \cdot \cot(\theta/2)}{2 \cdot \text{aspect}}$$

Oblique Perspective-Projection Frustum

Q.Explain in detail oblique perspective projection Frustum

- If the centerline of a perspective-projection view volume is not perpendicular to the view plane, we have an oblique frustum



In this case, first transform the view volume to a symmetric frustum and then to a normalized view volume.

- An oblique perspective-projection view volume can be converted to a symmetric frustum by applying a *z*-axis shearing-transformation matrix.
- Taking the projection reference point as $(x_{prp}, y_{prp}, z_{prp}) = (0, 0, 0)$, we obtain the elements of the required shearing matrix as

$$\mathbf{M}_{z\,\text{shear}} = \begin{bmatrix} 1 & 0 & \text{sh}_{zx} & 0 \\ 0 & 1 & \text{sh}_{zy} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{sh}_{zx} = -\frac{xw_{\min} + xw_{\max}}{2\,z_{\text{near}}}$$

$$\text{sh}_{zy} = -\frac{yw_{\min} + yw_{\max}}{2\,z_{\text{near}}}$$

- Similarly, with the projection reference point at the viewing-coordinate origin and with the near clipping plane as the view plane, the perspective-projection matrix is simplified to

$$\mathbf{M}_{\text{pers}} = \begin{bmatrix} -z_{\text{near}} & 0 & 0 & 0 \\ 0 & -z_{\text{near}} & 0 & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

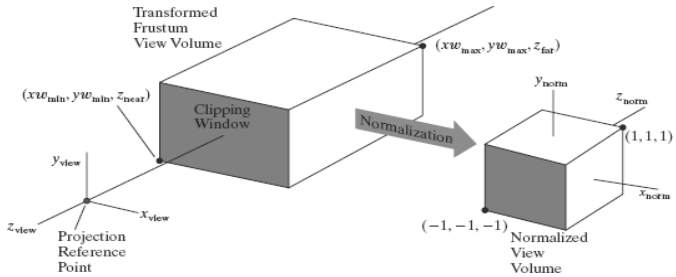
- Concatenating the simplified perspective-projection matrix with the shear matrix we have

$$\mathbf{M}_{\text{obliquepers}} = \mathbf{M}_{\text{pers}} \cdot \mathbf{M}_{z\,\text{shear}}$$

$$= \begin{bmatrix} -z_{\text{near}} & 0 & \dfrac{xw_{\min} + xw_{\max}}{2} & 0 \\ 0 & -z_{\text{near}} & \dfrac{yw_{\min} + yw_{\max}}{2} & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Normalized Perspective-Projection Transformation Coordinates

- The final step in the perspective transformation process is to map this parallelepiped to a normalized view volume.
- The transformed frustum view volume, which is a rectangular parallelepiped, is mapped to a symmetric normalized cube within a left-handed reference frame



Because the centerline of the rectangular parallelepiped view volume is now the z_{view} axis, no translation is needed in the x and y normalization transformations: We require only the x and y scaling parameters relative to the coordinate origin.

The scaling matrix for accomplishing the xy normalization is

$$\mathbf{M}_{xy\,\text{scale}} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Concatenating the xy-scalingmatrix produces the following normalization matrix for a perspective-projection transformation.

$$\mathbf{M}_{\text{normpers}} = \mathbf{M}_{xy\,\text{scale}} \cdot \mathbf{M}_{\text{obliquepers}}$$

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -z_{\text{near}} & 0 & \frac{xw_{\min} + xw_{\max}}{2} & 0 \\ 0 & -z_{\text{near}} & \frac{yw_{\min} + yw_{\max}}{2} & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$\mathbf{M}_{\text{normpers}} = \begin{bmatrix} -z_{\text{near}} s_x & 0 & s_x \frac{xw_{\min} + xw_{\max}}{2} & 0 \\ 0 & -z_{\text{near}} s_y & s_y \frac{yw_{\min} + yw_{\max}}{2} & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

The Viewport Transformation and Three-Dimensional ScreenCoordinates

- Once we have completed the transformation to normalized projection coordinates, clipping can be applied efficiently to the symmetric cube then the contents of the normalized view volume can be transferred to screen coordinates.
- Positions throughout the three-dimensional view volume also have a depth (z coordinate), and we need to retain this depth information for the visibility testing and surface rendering algorithms

If we include this z renormalization, the transformation from the normalized view volume to three dimensional screen coordinates is

$$\mathbf{M}_{\text{normviewvol,3D screen}} = \begin{bmatrix} \frac{xv_{\max} - xv_{\min}}{2} & 0 & 0 & \frac{xv_{\max} + xv_{\min}}{2} \\ 0 & \frac{yv_{\max} - yv_{\min}}{2} & 0 & \frac{yv_{\max} + yv_{\min}}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In normalized coordinates, the $z_{\text{norm}} = -1$ face of the symmetric cube corresponds to the clipping-window area. And this face of the normalized cube is mapped to the rectangular viewport, which is now referenced at $z\text{screen} = 0$.

Thus, the lower-left corner of the viewport screen area is at position $(xv_{\min}, yv_{\min}, 0)$ and the upper-right corner is at position $(xv_{\max}, yv_{\max}, 0)$.

OpenGL Three-Dimensional Viewing Functions

Q. Explain OpenGL 3D Viewing functions

- **OpenGL Viewing-Transformation Function**
- **OpenGL Orthogonal-Projection Function**
- **OpenGL General Perspective-Projection Function**
- **OpenGL Viewports and Display Windows**

OpenGL Viewing-Transformation Function

glMatrixMode (GL_MODELVIEW);

➔ a matrix is formed and concatenated with the current modelview matrix, We set the modelview mode with the statement above

gluLookAt (x0, y0, z0, xref, yref, zref, Vx, Vy, Vz);

Viewing parameters are specified with the above GLU function.

This function designates the origin of the viewing reference frame as the world-coordinate position $P_0 = (x_0, y_0, z_0)$, the reference position as $P_{ref} = (x_{ref}, y_{ref}, z_{ref})$, and the view-up vector as $V = (V_x, V_y, V_z)$.

If we do not invoke the gluLookAt function, the default OpenGL viewing parameters are

$P_0 = (0, 0, 0)$

$P_{ref} = (0, 0, -1)$

$V = (0, 1, 0)$

OpenGL Orthogonal-Projection Function

glMatrixMode (GL_PROJECTION);

set up a projection-transformation matrix.

Then, when we issue any transformation command, the resulting matrix will be concatenated with the current projection matrix.

glOrtho (xwmin, xwmax, ywmin, ywmax, dnear, dfar);

Orthogonal-projection parameters are chosen with the function

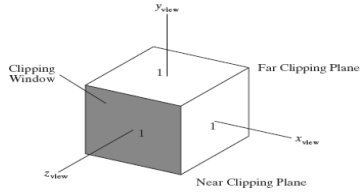
All parameter values in this function are to be assigned double-precision, floating point Numbers

Function glOrtho generates a parallel projection that is perpendicular to the view plane

Parameters d_{near} and d_{far} denote distances in the negative z_{view} direction from the viewing-coordinate origin

We can assign any values (positive, negative, or zero) to these parameters, so long as $d_{near} < d_{far}$.

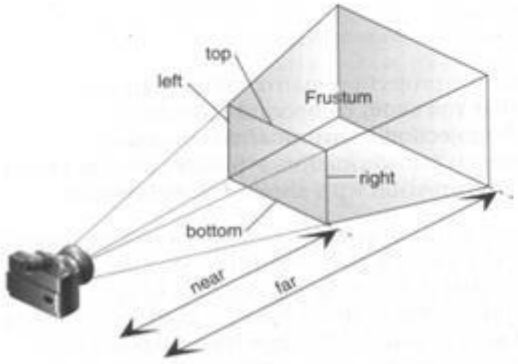
Exa: glOrtho (-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);

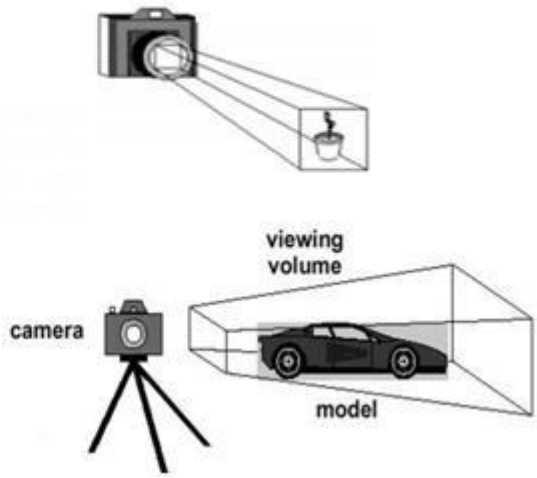


OpenGL General Perspective-Projection Function

glFrustum (xwmin, xwmax, ywmin, ywmax, dnear, dfar);

glFrustum describes a perspective matrix that produces a perspective projection.





OpenGL Viewports and Display Windows

glViewport (xvmin, yvmin, vpWidth, vpHeight);

A rectangular viewport is defined.

The first two parameters in this function specify the integer screen position of the lower-left corner of the viewport relative to the lower-left corner of the display window.

And the last two parameters give the integer width and height of the viewport.

To maintain the proportions of objects in a scene, we set the aspect ratio of the viewport equal to the aspect ratio of the clipping window.

Display windows are created and managed with GLUT routines. The default viewport in OpenGL is the size and position of the current display window

Visible Surface Detection Methods

- Classification of visible surface Detection algorithms,
- back face detection
- Depth buffer method
- OpenGL visibility detection functions.

Classification of Visible Surface Detection Algorithm

Q Give the general Classification of visible detection algorithm and explain any one algorithm in detail (6M)

Visible Surface Detection algorithm are classified into

- Object Space Method
- Image Space Method

Object Space Method

- It compares objects and parts of object to each other within the scene definition to determine which surface is visible.
- Line display algorithms use this method to identify visible line in wire frame display.

Image Space Method

- Here visibilty is decided point by Point at each pixel position on the Projection Plane.
- Visible surface algorithm use this method for Visible line detection.

Back-Face Detection

- This method is used to remove back face of any object.
- This method is like a preprocessor that removes back face in all methods
- A fast and simple object-space method for identifying the back faces of a object is based on the "inside-outside" tests.

First Approach INSIDE and OUTSIDE TEST

- A point (x, y, z) is "inside" a polygon surface with plane parameters A, B, C, and D, if When an inside point is along the line of sight to the surface, the polygon must be a back face

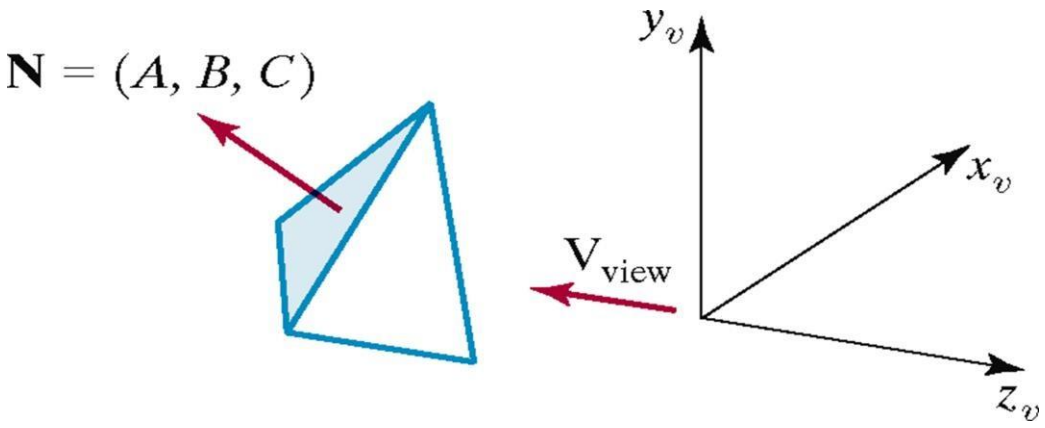
- If the polygon surface equation:

$$Ax + By + Cz + D < 0$$

- Then the point is inside polygon surface, So it is back of the front face, can be eliminated

Second approach

- Consider the normal vector N to a polygon surface, which has Cartesian components (A, B, C).



- $\mathbf{V}_{view}$ is a vector in the viewing direction from the eye ("camera") position, a polygon surface is a back face if:

$$\mathbf{V}_{view} \cdot \mathbf{N} > 0$$

$$\mathbf{V}_{view} = (0,0,V_z) \text{ and } \mathbf{N} = Ax + By + Cz$$

$$\mathbf{V}_{view} \cdot \mathbf{N} = (0,0,V_z)(Ax + By + Cz)$$

$$= V_z C \quad (\text{If } V_z = 1)$$

$$\mathbf{V}_{view} \cdot \mathbf{N} = C$$

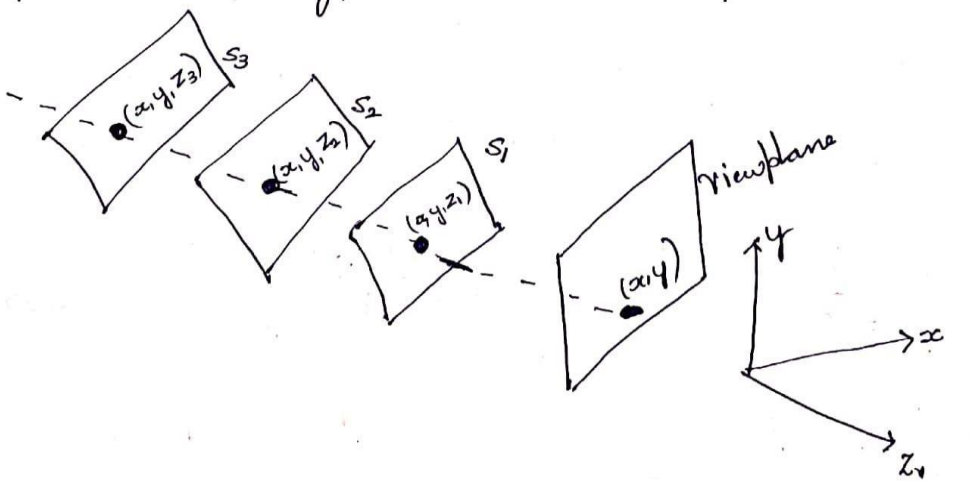
$$\text{Sign}(C) \geq 0 \rightarrow \text{Surface is back(invisible)}$$

$$\text{Sign}(C) < 0 \rightarrow \text{Surface if front (visible)}$$

Depth Buffer Algorithm (Z Buffer Method)

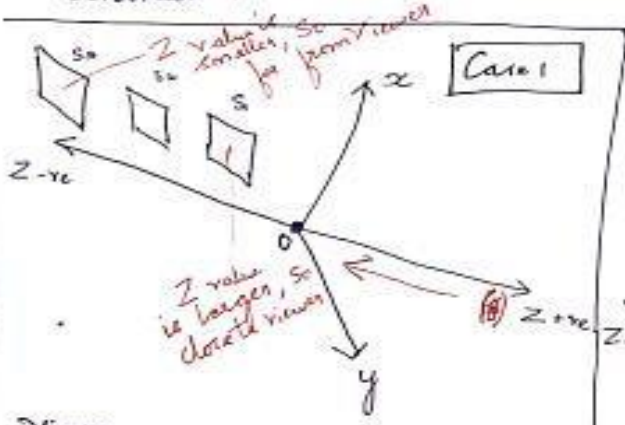
Q. Write and explain Depth Buffer algorithm (8M)

- Depth buffer method uses image-space approach for detecting visible surfaces.
- It compares surface depth values throughout a scene for each pixel position on the projection plane.
- Each surface of a scene is processed separately, one pixel position at a time, across the surface.
- Consider three surfaces at varying distances along the orthographic projection line from position (x, y) on a view plane.



- Basically 2 buffers are used
  1) Depth Buffer store Z values (Depth values)
  2) Refresh Buffer (Frame Buffer) store Surface intensity value

- Consider 2 cases

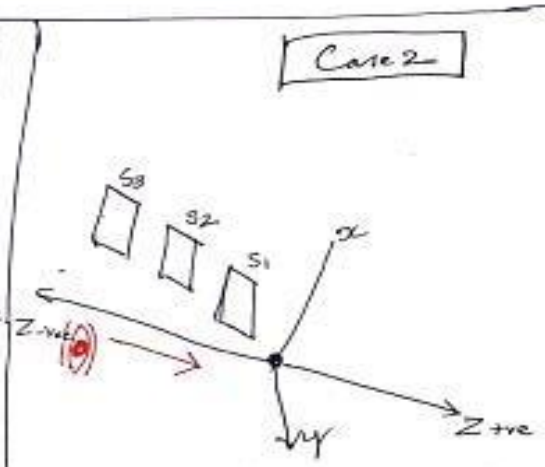


Viewer Sitting at Z+ve & looking at Origin (ie viewing along Z-ve)

- Surface with Larger Z value is closer to viewer
- Smaller Z value is farther to viewer

Depth buffer will be initialised with $d(x,y) = 0$

$Z > depthBuff(x,y)$



Viewer Sitting at Z-ve & Looking at Origin (viewing Z+ve)

- Surface with Smaller Z value is closer
- Larger Z value is far

Depth buffer will be initialized with $d(x,y) = Z_{max}$

$Z < depthBuff(x,y)$

Algorithm steps

1) Initialize the Depth buffer and frame buffer so that for all buffer position (x, y)

$$\text{depthBuff}(x, y) = 1.0$$
$$\text{frameBuff}(x, y) = I_{background}$$

2) Process each polygon in a scene, one at a time, as follows:

- For each projected (x, y) pixel position of a polygon, calculate the depth z
- If $z <$ depthBuff (x, y)
  Compute the surface color at that position and set

$$\text{depthBuff}(x, y) = z$$
$$\text{frameBuff}(x, y) = \text{surfColor}(x, y)$$

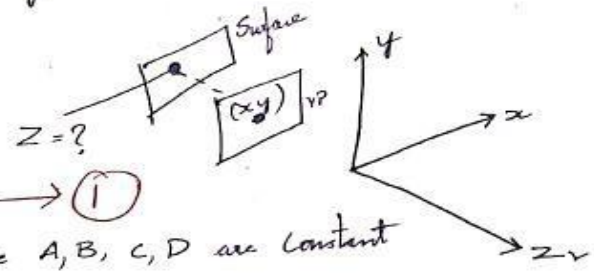
After all surfaces have been processed, the depth buffer contains depth value for visible surface & Frame buffer contains color value for those surface.

Calculate Z (depth at any point on plane containing Polygon)

- Plane Equation is $Ax + By + Cz + D = 0$
- At surface position (x,y), the depth is calculated by

$$Z = \frac{-Ax - By - D}{C} \rightarrow (1)$$

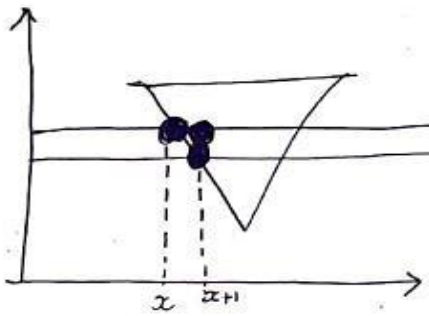
where A, B, C, D are constant



Pixel (x,y)

For Every Pixel (x,y) if we apply this method to calculate Z, it is Time Consuming

To Solve this problem, another approach is Scanline



If Z value at (x,y) is found then we can find Z' value at $(x+1, y)$
Z' value at $(x+1, y-1)$

To Find z' value at $(x+1, y)$

We Know $Z = \dfrac{-Ax - By - D}{C}$

$$z' = \frac{-A(x+1) - By - D}{C}$$

$$= \frac{-Ax - By - D - A}{C}$$

$$\boxed{z' = z - \frac{A}{C}} \rightarrow (2)$$

where $-A/C$ is constant

So succeeding depth value across a scanline are obtained from preceding values with a single addition

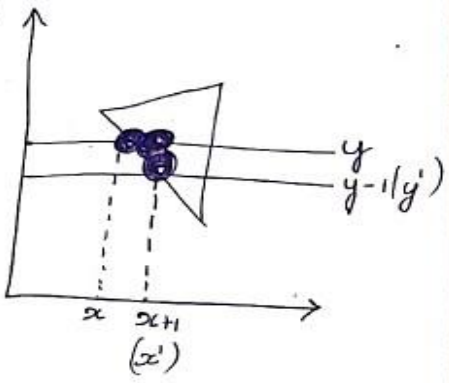
Now to Find z' value at $(x+1, y-1)$

$$m = \frac{y - y'}{x - x'}$$

$$= \frac{y - (y-1)}{x - x'} \Rightarrow \frac{y - y + 1}{x - x'}$$

$$m = \frac{1}{x - x'} \Rightarrow x - x' = \frac{1}{m}$$

$$\boxed{x - \frac{1}{m} = x'}$$



We know $z = \frac{-Ax - By - D}{C} \Rightarrow \frac{-Ax' - By' - D}{C}$

$$z' = \frac{-A\left(x - \frac{1}{m}\right) - B(y-1) - D}{C}$$

$$= \frac{-Ax + \frac{A}{m} - By + B - D}{C}$$

$$= \frac{-Ax - By - D}{C} + \frac{A/m + B}{C}$$

$$\boxed{z' = z + \frac{A/m + B}{C}}$$

If we are processing down a vertical edge, Slope m is infinite, So

$$\boxed{z' = z + \frac{B}{C}} \rightarrow (3)$$

A/m is infinite

Equation (1), (2) & (3) can be used to Calculate Z

Drawback:
- Its time consuming process
- 2 Buffers are need to be maintained, So costly
- Requires large Memory

Advantage
- It is efficient
- Easy to Implement

OpenGL Visibility Detection Function

Q Explain OpenGL Visibility Detection Function (8M)

OpenGL Polygon - Culling Functions

```
glCullFace(mode);
```

- This function is used to remove Back face, Front face or both front and back faces of object.
- Parameter mode → assigned value OpenGL symbolic Constant { GL_BACK, GL_FRONT, GL_FRONT_AND_BACK

```
glEnable(GL_CULL_FACE);   → Activate Culling function
glDisable(GL_CULL_FACE);  → Deactivate Culling function
```

OpenGL Depth - Buffer Functions

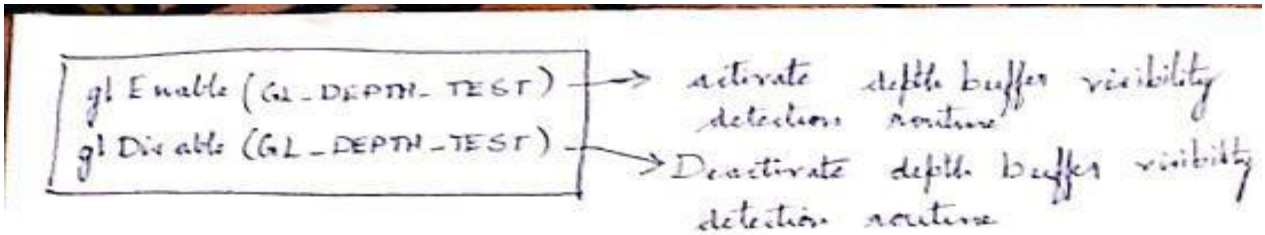
```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)
```

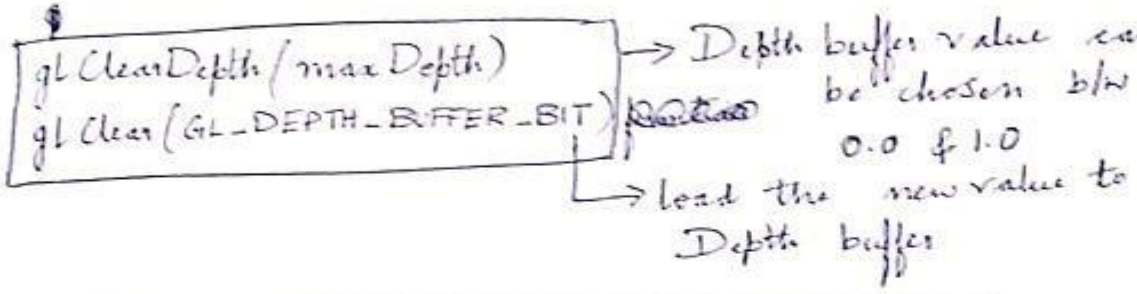
- This initialization function will request for depth buffer and refresh buffer.

```
glClear(GL_DEPTH_BUFFER_BIT);
```

→ Depth buffer values are initialized to max value 1.0 by default

→ It can be Normalized in range 0.0 to 1.0





glDepthRange (nearNormDepth , farNormDepth)

This function will Normalize depth buffer with parameter nearNormDepth = 0.0 & farNormDepth = 1.0

OpenGL Wire Frame Surface – Visibility Method

glPolygonMode (GL_FRONT_AND_BACK , GL_LINE);

This function is used for wire frame display of object, But display both visible and hidden edges

OpenGL Depth - Cueing Function

```
glFogi (GL_FOG_MODE, GL_LINEAR)
```
→ Start or End mode

- This function is used to vary the brightness of an object.
- It applies linear depth function to object colors using $d_{min} = 0.0$ & $d_{max} = 1.0$ by default

```
glFogf (GL_FOG_START, minDepth)
glFogf (GL_FOG_END, maxDepth)
```
→ d_{min} value is set
→ d_{max} value is set

```
glEnable (GL_FOG)
```
→ Activate Fog function

Module 5

Input and Interaction

Interaction refers to the manner in which the application program communicates with input and output devices of the system.

OpenGL doesn't directly support interaction in order to maintain portability. However, OpenGL provides the GLUT library. This library supports interaction with the keyboard, mouse etc

Input devices are the devices which provide input to the computer graphics application program.

Input devices can be categorized in two ways:
1. Physical input devices
2. Logical input devices

Input Devices

There are two different ways to look at the devices:

Physical devices: keyboard or mouse, and discuss how they work
Logical devices: the way application programs look at the devices.

A logical device is characterized by its high-level interface with the user program, rather than by its physical characteristics.

In C input and output are done using, *printf, scanf, getchar, and putchar*.

In computer graphics, the use of logical devices is slightly more complex. This is because the forms that input can take are more varied than the strings of bits or characters. In a nongraphical application, we restricted to bits or characters.

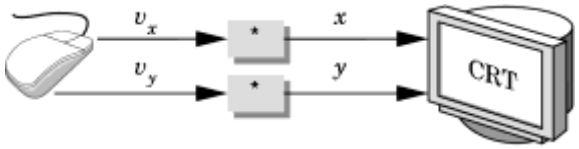
Each device has properties that make it more suitable for certain tasks than for others. There are two primary types of physical devices:

1) pointing devices
2) keyboard devices

Input Devices – cont.

We can view the output of the mouse or trackball as two independent values provided by the device.

Relative Positioning Devices



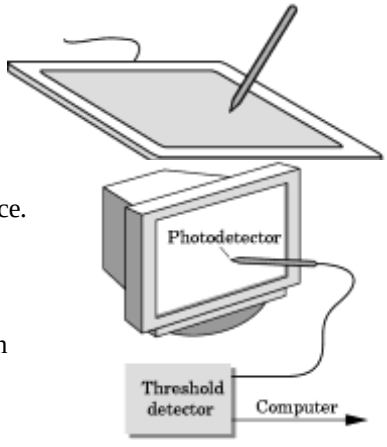
It is not necessary that the output of the mouse or trackball encoders be interpreted as a position. Instead, either the device driver or a user program can interpret the information from the encoder as two independent velocities.

The computer can then integrate these values to obtain a two-dimensional position.

By integrating the distance traveled by the ball as a velocity, we can use the device as a variable-sensitivity input device in which large deviations from the rest causes rapid large changes and small deviations cause slow or small changes.

Input Devices – cont.

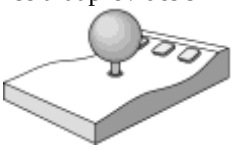
Data tablets provide absolute positioning unlike the relative positioning devices. A typical data tablet has rows and columns of wires embedded under its surface.



The lightpen is another device used in computer graphics. The lightpen contains a light-sensing device, such as a photocell. This device provide a direct positioning device.

Joystick is another physical device. In this device, the stick will control the two dimensional orthogonal motions. This two motions are integrated to identify the location on the screen.

Spaceball is another device that provides six degrees of freedom.



Logical Devices

The main characteristics that describes the logical behavior of an input device:

1) what measurements the device returns to the user program, and
2) when the device returns those measurements.

In general, there are six classes of logical input devices:

1.String – provides ASCII strings to the user program (logical implemented via keyboard)
2.Locator – provides a position in world coordinates to the user program (pointing devices and conversions may be needed)
3.Pick – returns the identifier of an object to the user program. (pointing devices and conversions may be needed)
4.Choice – allows users to select one of the distinct number of options (widgets – menus, scrollbars, and graphical buttons)
5.Dial – provides analog input to the user program (widgets – slidebars,…)
6.Stroke – it returns an array of locations (similar to multiple use of a locator, continuous)

Measure and Trigger

The manner by which physical and logical input devices provide input to an application program can be described in terms of two entities:

1) A measure process, and 2) A device trigger.

The **measure** of a device is what the device returns to the user program.
The **trigger** of a device is a physical input on the device with which the user can signal the computer.

Example: The measure of a keyboard is a string,
The trigger could be the "return" or "enter" key.
Example: For a locator the measure includes the location and
The trigger can be the button on the pointing device.

When we develop an application program, we have to account for the user triggering the device while she is not pointing an object.

We can include, as part of the measure, a status variable that indicates that the user is not pointing to an object.

Input Modes

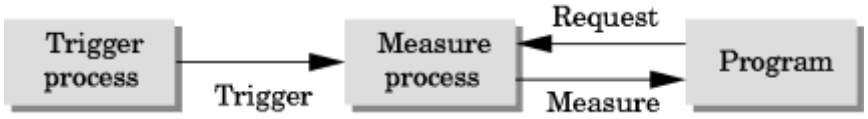
In addition to multiple types of logical input devices, we can obtain the measure of a device in three distinct modes:

1) Request mode, 2) Sample mode, and 3) Event mode.

It defined by the relationship between the measure process and the trigger. Normally, the initialization of an input device starts a measure process.

1) Request mode: In this mode the measure of the device is not returned to the program until the device is triggered.

A locator can be moved to different point of the screen. The Windows system continuously follows the location of the pointer, but until the button is depressed, the location will not be returned.



Input Modes – cont.

2) Sample mode: Input is immediate. As soon as the function call in the user program is encountered, the measure is returned, hence no trigger is needed.

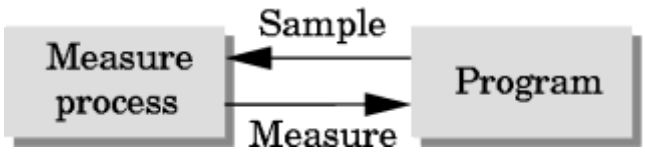
For both of the above modes, the user must identify which devices is to provide the input.

request_locator(device_id, &measure);

sample_locator(device_id, &measure);

identifier *location*

Think of a flight simulator with many input devices



Input Modes – cont.

3) Event mode: The previous two modes are not sufficient for handling the variety of possible human-computer interactions that arise in a modern computing environment. The can be done in three steps:

1) Show how event mode can be described as another mode within the measure-trigger paradigm. 2) Learn the basics of client-servers when event mode is preferred, and 3) Learn how OpenGL uses GLUT to do this.

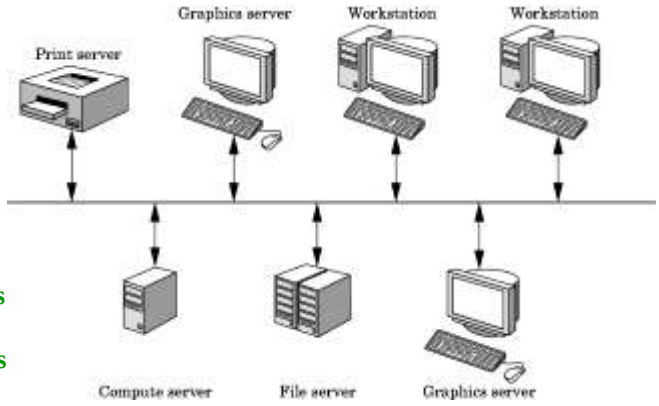
In an environment with multiple input devices, each with its own trigger and each running a measure process. Each time that a device is triggered, an **event** is generated. The device measure, with the identifier for the device, is placed in an **event queue.** The user program executes the events from the queue. When the queue is empty, it will wait until an event appears there to execute it.

Another approach is to associate a function called a **callback** with a specific type of event. This is the approach we are taking.

Client and Servers

If a computer graphics is to be useful for a variety of real applications, it must function well in a world of distributed computing and networks.

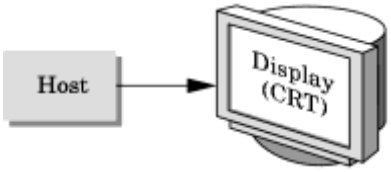
In this world, the building blocks are entities called **servers** that can perform tasks for **clients**.



- **Print servers**
- **Compute servers**
- **File servers**
- **Terminal servers**

Display Lists

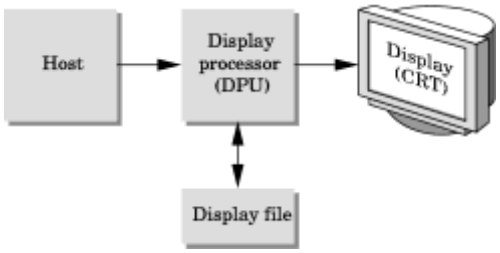
Display lists illustrate how we can use clients and servers on a network to improve graphics performance.



The computer would send out the necessary information to redraw the display at a rate sufficient to avoid noticeable flicker. In the past, since computers were slow and expensive, the cost of keeping even a simple display refreshed was prohibitive for all but a few applications.

The solution is to this problem was to build a special-purpose computer, called a **display processor**.

Today, the display processor is a a graphics server, and the user program on the host computer is a client.



Display Lists – cont.

We can send graphical entities to a display in one of the two ways:

1)Send the complete description of our objects to the graphics server. For a typical geometric primitives, this transfer consists of; sending vertices, attributes, and primitive types, in addition to viewing information.

2)Define the object once, then put its description in a display list. The display list is stored in the server and redisplayed by a simple function call issued from the client to the server. This method is called **retained mode** graphics.

Disadvantages associated with the use of display list:

1) Display lists require memory on the server,

2) There is an overhead for creating a display list.

Definition and Execution of Display Lists

Display lists have many things in common with ordinary files. OpenGL has a small set of functions to manipulate display lists, and places only a few restrictions on display-list contents.

Display lists are defined similarly to geometric primitives. There is a *glNewList* at he beginning and a *glEndList* at the end, and the contents in between.

Each display list must have a unique identifier, usually an integer defined using *#define* directive. Example:

```
#define BOX 1 // or some other unused integer
glNewList(BOX, GL_COMPILE); //send the list to the server, but not to display its contents
        glBegin(GL_POLYGON);
                glColor3f(1.0, 0.0, 0.0);
                glVertex2f(-1.0, -1.0);
                glVertex2f(1.0, -1.0);
                glVertex2f(1.0, 1.0);
                glVertex2f(-1.0, 1.0);
        glEnd( );
glEndList;
```

Definition and Execution of Display Lists – cont.

Each time we wish to draw the box on the server, we execute the function: *glCallList(BOX);*

Note that the present state of the system determines which transformations are applied to the primitives in the display list. Thus, if we change the model-view or projection matrices between executions of the display list, the box will appear in different places or even will no longer appear. Example:

```
glMatrixMode(GL_PROJECTION)
for(i = 1; i < 5; i++)
{
        glLoadIdentity( );
        gluOrtho2D( -2.0*i , 2.0*i , -2.0*i , 2.0*i );
        glCallList(BOX);
}
```

Every time that the *glCallList* is executed, the box is redrawn with a different clipping rectangle.

Definition and Execution of Display Lists – cont.

The OpenGL stack data structure can be used to keep the matrix and its attribute. At the beginning of a display list, place:

glPushAttrib(GL_ALL_ATTRIB_BITS);
glPushMatrix( );

At the end place:
glPopAttrib( );
glPopMatrix( );

If you are not sure about which number to use for a list, use *glGenLists(number).* This returns the first integer (or base) of number consecutive integers that are unused labels.

The function *glCallLists* allows us to execute multiple display lists with a single function call.

Text and Display Lists

Both raster and stroke text require a reasonable amount of code to describe set of characters.
Suppose we have used a 12x10 pattern of bits to store each character. 12x10 = 120 bits, 15 bytes to store each character.
To define a stroke font we need many more bytes.

Filled string (a) Input Output

Magnified outlines (b) 

It will take a significant amount of resources to display a string of character.
One way to fix this problem is to use a display list for each character, and then to store the font on the server via these display lists. Similar technique is used to display bitmap fonts.
Bit-map fonts are stored in ROM and each character is selected and displayed based on single byte; its ASCII code.

Text and Display Lists – cont.

We can define either the standard 96 printable ASCII characters or we can define patterns for a 256-character extended ASCII character set.

```
void OurFont( char c)
{
   switch( c )
   {
         case 'a':
         ...
         break;
         case 'A':
         ...
         break;
         ...
   }
}
```

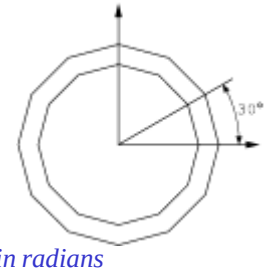
We have to be careful about spacing. Each character in the string must be displayed to the right of the previous character.

We can use the translate function *glTranslate* to get the desired spacing.

Text and Display Lists – cont.

Suppose we want to display letter O and we wish to fit it in a unit square:

```
   switch( c )
   {
      case 'O';
            glTranslatef(0.5, 0.5, 0.0); // move to center
            glBegin(GL_QUAD_STRIP);
             for( I = 0; I <= 12; I++) // 12 vertices
             {
                     angle = 3.14159/6.0 * I; // 30 degrees in radians
                     glVertex2f(0.4*cos(angle), 0.4*sin(angle));
                     glVertex2f(0.5*cos(angle), 0.5*sin(angle));
            }
            glEnd( );
            glTranslatef(0.5, -0.5, 0.0) ; //move to lower right
      break;
      ...
   }
```



Text and Display Lists – cont.

To generate 256 characters using this method, we can use a code like this one:

```
Base = glGenLists(256); //return the index of first of 256 consecutive available
                        //ids
for( I = 0 ; I < 256; I++)
{
        glNewList(base+I , GL_COMPILE);
        OurFont( I );
        glEndList( );
}
```

When we use these display lists to draw individual characters, rather than offsetting the identifier of the display lists by base each time, we can set an offset with:
glListBase(base);
Then a string defined as *char *text_string;*
Can be drawn as:
glCallLists((Glint) strlen(text_string), GL_BYTE, text_string);

Fonts in GLUT

Previous method requires us to create all letters. We prefer to use an existing font, rather than to define our own. GLUT provides a few raster and stroke fonts.

We can access a single character from a **monotype** (evenly spaced) font by the function call:
glutStrokeCharacter(GLUT_STROKE_MONO_ROMAN, int character)

GLUT_STROKE_ROMAN provides proportionally spaced characters with a size of approximately 120 units maximum. Note that this may not be an appropriate size for your program, thus resizing may be needed.

We control the location of the first character using a translation. In addition, once each character is printed there will be a translation to the bottom right of that character.
Note that translation and scaling may affect the OpenGL state. It is recommended that we use the *glPushMatrix* and *glPopMatrix* as necessary, to prevent undesirable results.

Fonts in GLUT – cont.

Raster or bitmap characters are produced in a similar manner. For example:
glutBitmapCharacter(GLUT_BITMAP_8_BY_13, int character)

will produce an 8x13 character.
The position is defined directly in the frame buffer and are not subject to geometric transformations. A **raster position** will keep the position of the next raster primitive. This position can be defined using *Raster-Pos*( )*.

If a character have a different width, we can use the function
glutBitmapWidth(font, char)
To return the width of a particular character.

```
glRasterPos2i(rx, ry);
glutBitmapCharacter(GLUT_BITMAP_8_BY_13, k);
rx += glutBitmapWidth(GLUT_BITMAP_8_BY_13, k);
```


Programming Event-Driven Input
Using the Pointing Device

Two types of events are associated with the pointing device.

move event: is generated when the mouse is moved with one of the buttons depressed, for a mouse the **mouse event** happens when one of the buttons is depressed or released.

passive move event: is generated when the mouse is moved without a button being hold down.

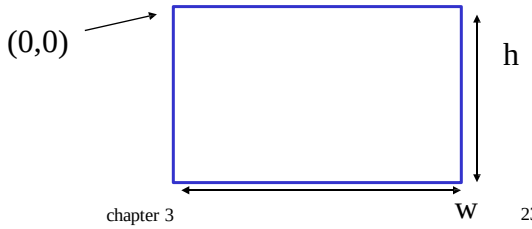
The mouse callback function looks like this:
glutMouseFunc(mouse_callback_func)

void mouse_callback_func(int button, int state, int x, int y)

Within the callback function, we define what action we want to take place if the specified event occurs. There may be multiple actions defined in the mouse callback function corresponding to the many possible button and state combinations.

Positioning

- The position in the screen window is usually measured in pixels with the origin at the top-left corner
  - Consequence of refresh done from top to bottom
- OpenGL uses a world coordinate system with origin at the bottom left
  - Must invert *y* coordinate returned by callback by height of window
  - $y = h - y;$



Using the Pointing Device

Suppose we want the program to terminate when the left button is depressed.

```
void mouse_callback_function(int button, int state, int x, int y)
{
    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        exit( 1);
}
```

It is obvious that depression or release of the other button will result in any action.

Window Events

Window events are relocation and the resizing of the window. In the window size changes, we have to consider three questions:

1) Do we redraw all the objects that were in the window before it was resized?
2) What do we do if the aspect ratio of the new window is different from that of the old window?
3) Do we change the size or attributes of new primitives if the size of the new window is different from that of the old?

Square Example.

The Square Program

The program is to draw an square by pressing the left button and to terminate the program by pressing the right button.

```
int main(int argc, char** argv){
  glutInit(&argc,argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
  glutCreateWindow("square");
   myinit ();
  glutReshapeFunc (myReshape);
  glutMouseFunc (mouse);
  glutMotionFunc(drawSquare);
  glutDisplayFunc(display);
  glutMainLoop();
  return 0;
}
```


The Square Program

```
void myinit(void){
 /* Pick 2D clipping window,match size of screen window This choice avoids having to scale object
   coordinates each time window is resized */
 glViewport(0,0,ww,wh);
 glMatrixMode(GL_PROJECTION);
 glLoadIdentity();
 /* set clear color to black and clear window */
 glOrtho(0.0, (GLdouble) ww , 0.0, (GLdouble) wh , -1.0, 1.0)
 glClearColor (1.0, 0.0, 0.0, 1.0);
 glClear(GL_COLOR_BUFFER_BIT);
 glFlush();/* callback routine for reshape event */
 glutReshapeFunc(myReshape);
}

void mouse(int btn, int state, int x, int y){
 if(btn==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
 exit(1);
}/* display callback required by GLUT 3.0 */

void display(void){ }
```

The Square Program

```
#include <GL/glut.h>
#include<math.h>#include<stdlib.h>
GLsizei wh = 500, ww = 500; /* initial window size */
GLfloat size = 3.0; /* half side length of square */

void drawSquare(int x, int y){
  y=wh-y;
  glColor3ub( (char) rand()%256, (char) rand()%256, (char) rand()%256);
  glBegin(GL_POLYGON);
  glVertex2f(x+size, y+size);
  glVertex2f(x-size, y+size);
  glVertex2f(x-size, y-size);
  glVertex2f(x+size, y-size);
  glEnd();
  glFlush();
}/* rehaping routine called whenever window is resizedor moved */

void myReshape( GLsizei w, GLsizei h){/* adjust clipping box */
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(0.0, (GLdouble)w, 0.0, (GLdouble)h, -1.0, 1.0);
  GlMatrixMode(GL_MODELVIEW);
  GlLoadIdentity(); /* adjust viewport and clear */
  GlViewport(0,0,w,h);
  GlClearColor (1.0, 1.0, 1.0, 1.0);

  glClear(GL_COLOR_BUFFER_BIT);
  glFlush();/* set global size for use by drawing routine */
  ww = w;
  wh = h;
}
```


Keyboard Events

We can use the keyboard event as an input device. Keyboard events are generated when the mouse is in the window and one of the keys is depressed.

In GLUT, there is no callback for the release of a key. The release does not generate a second event. Only one call back function for the keyboard:

glutKeyboardFunc(keyboard);

To use the keyboard to exit a program:

```
void keyboard(unsigned char key, int x, int y)
{
        if(key == 'q' || key == 'Q')
                exit(1);
}
```

The Display and idle Callback

We have seen the display callback: *glutDisplayFunc(display);*

It is invoked when GLUT determines that the window should be redisplayed. One such situation is when the window is open initially.

Since the display event will be generated when the window is first opened, the display callback is a good place to put the code that generates most non-interactive output. GLUT requires all programs to have a display function, even if it is empty.

Some of the things we can do with the display callback:

1) Animation – various values defined in the programs may change

2) Opening multiple windows

3)iconifying a window – replacing a window with a small symbol or picture.

glutPostRedisplay( );

The **idle callback** is invoked when there are no other events. Its default is the null function.

Window Management

GLUT supports both multiple windows and subwindows of a given window.

id = glutCreateWindow("Second Window");

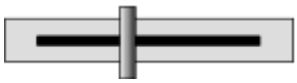
The returned integer value allows us to select this window as the current window:

glutSetWindow(id);

Menus

We can use our graphics primitives and our mouse callback to construct various graphical input devices.

How do you think we can create this one?



GLUT provides **pop-up menus**.

Using menus involves:

1) Must define the entries in the menu,
2) must link the menu to a particular mouse button, and
3) must define a callback function corresponding to each menu entry.

Example – Pop-up menu

```
glutCreateMenu(demo_menu);
glutAddMenuEntry("quit", 1);
glutAddMenuEntry("increase square size", 2);
glutAddMenuEntry("decrease square size", 3);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

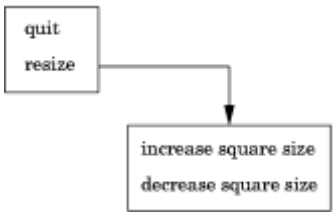
The callback function looks like this:

```
void demo_menu(int id)
{
        if(id == 1)  exit(1);
        else if (id == 2) size = 2*size;
        else size = size / 2;
        glutPostRedisplay( );
}
```


Example – Hierarchical menu

Suppose we want the main menu that we create to have two entries:

1) the first one to terminate the program
2) the second to pop-up a submenu.




```
Sub_menu = glutCreateMenu(size_menu);
glutAddMenuEntry("increase square size", 2);
glutAddMenuEntry("decrease square size", 3);
glutCreateMenu(top_menu);
glutAddMenuEntry("quit", 1);
glutAddSunMenu("Resize", sub_menu);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

Now we have to write the call back functions, *size_menu* and *top_menu*.

Picking

Picking is an input operation that allows the user to identify an object on the display. Although, the picking is done by a pointing device, the information returned to the application program is not a position.

A pick device is more difficult to implement than the locator device. There are two ways to do this:

1)selection, involves adjusting the clipping region and viewport such that we can keep track of which primitives in a small clipping region are rendered into a region near the cursor. Creates a **hit list**.

2)bounding rectangles or **extents,** this is the smallest rectangle, aligned with the coordinates axes, that contains the object.

A Simple Paint Program

This example illustrates the use of callbacks, display lists, and interactive program design by developing a simple paint program.

A paint program should demonstrate:

•Ability to work with geometric objects, such as line segments and polygons. It should allow us to enter the vertices interactively.

• Ability to manipulate pixels and to draw directly into the frame buffer.

• Should allow control of attributes such as color, line type, and fill pattern.

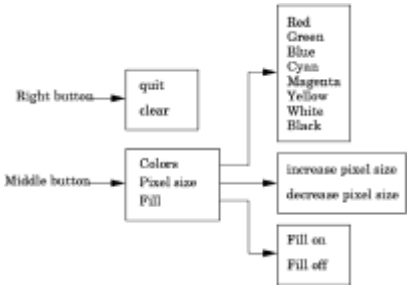
• It should include menus for controlling the application

• It should behave correctly when the window is moved or resized.

Initial Window

The most difficult part is to decide what functionality to place in each of the functions.

Structure

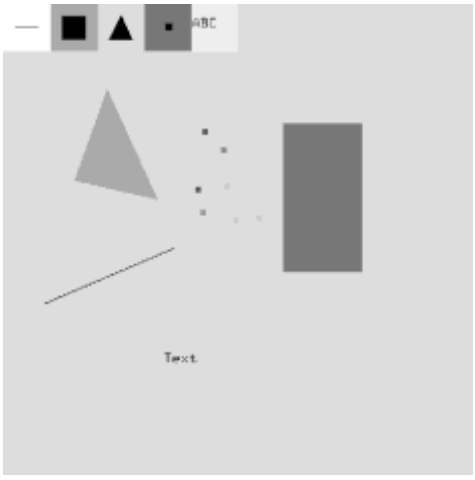


Controlled by the middle and right mouse buttons

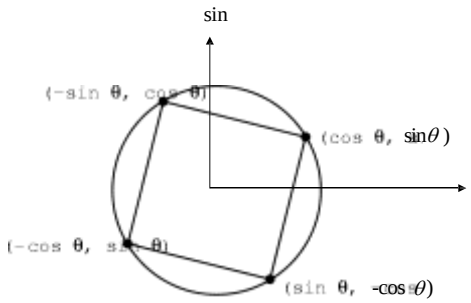
- Right button to clear the screen or to terminate the program
- Middle button allows us to change drawing color, to select between fill and no fill for the triangle and rectangle, and to change the size of the pixel rectangle

Animating Interactive Programs

The simple paint program - demo



The Rotating Square




```
void display( void )
{
  float pi = 3.14159;
  double t = 45.0*pi/180.0; //30 degree

   for(int i = 0; i < 200; i++){
      t = t + i* (2*pi*0.1);
      glClear(GL_COLOR_BUFFER_BIT); /*clear the window */
         glBegin(GL_POLYGON);
         glVertex2f(20*cos(t), 20*sin(t));
         glVertex2f(-20*cos(t), 20*sin(t));
         glVertex2f(-20*cos(t), -20*sin(t));
         glVertex2f(20*cos(t), -20*sin(t));
      glEnd();

      glFlush(); /* clear buffers */
   }
}
```


The idle function

The idle callback will allow us to put the program as if nothing is happening.

glutIdleFunc(idle);

We can use this in our rotate square program to rotate the square by an angle when nothing else is happening:

```
void idle()
{
   t += 20;
   if( t >= 360)
        t -= 360;
   glutPostRedisplay( );
}
```

Combine this with mouse callback *glutMouseFunc(mouse)* operation:

```
void mouse(int button, int state, int x, int y)
{
   if( button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
       glutIdleFunc(idle);
   if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
   glutIdleFunc(NULL);
}
```


Don't for get to include glutMouseFunc (mouse); in the main.

Double Buffering

When we redisplay our CRT, we want to do so at a rate sufficiently high (50-80 times/sec) that we cannot notice the clearing and redrawing of the screen.

That means the contents of the frame buffer must be drawn at this rate.

One problem may occur when a complex shape is drawn. In that case the display may not be done in one refresh cycle. A moving object will be distorted.

Double buffering can provide a solution to these problems. The **front buffer** is displayed when we draw in the **back buffer**. We can swap the back and front buffer from the application program.

With each swap, the display callback is invoked.

The double buffering is set using the *GLUT_DOUBLE*, instead of *GLUT_SINGLE* in *glutInitDisplayMode*. The buffer swap function using GLUT will be: *glutSwapBuffer( );*

Double buffering does not speed of the process of displaying a complex display. It only ensures that we never see a partial display.

Design of Interactive Programs

A good interactive program includes features such as:

1)A smooth display, showing neither flicker nor any artifacts of the refresh process.
2) A variety of interactive devices on the display.
3) A variety of methods for entering and displaying information.
4) An easy-to-use interface that does not require substantial effort to learn.
5) Feedback to the user.
6) Tolerance for user errors.
7)A design that incorporates consideration of both the visual and motor properties of the human.

Curve

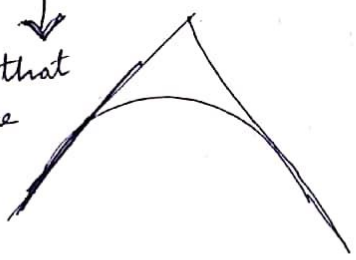
Curve is a large collection of points, where for every point consist of neighbours, excluding end points.

Types of Curve — 1) Implicit Curve
2) Explicit Curve
3) Parametric Curve

Bezier Curve

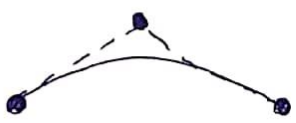
- Curve is drawn by considering Control points
- Approximate tangent is drawn by using Control point

A plane that touches the Curve

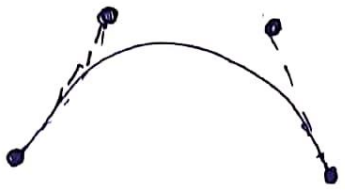


- Simplest form of Bezier Curve is connection 2 endpoints

- Quadric Curve is drawn using 3 control points



- Cubic Curve drawn using 4 Control points



- Bezier Curve Properties
  - ↳ depends on number of Control points
  - ↳ Curve will pass through end point, but not all control point
  - ↳ Polynomial equation of Curve depends on no. of Control points

    n → Control point (4)

    $n-1$ → Degree of polynomial equation (3)

Bezier Spline Curve

- It is developed by French engineer Pierre Bezier.
- Bezier curve can be fitted to any number of control points, although some graphics package limit to four control points

Bezier Curve Equation

$$Q(u) = \sum_{k=0}^{n} P_k \cdot BEZ_{k,n}(u) \qquad 0 \leq u \leq 1$$

Where P_k = Position Vector Coordinate, K varying from 0 to n

$BEZ_{k,n}(u)$ = Bernstein Polynomial
Bezier Blending Function

- $$BEZ_{k,n}(u) = C(n,k) \cdot u^k \cdot (1-u)^{n-k}$$

Where $C(n,k) = \dfrac{n!}{k!\,(n-k)!}$ Binomial Coefficients

- A set of 3 parametric equations for the individual curve coordinates can be represented as

$$x(u) = \sum_{k=0}^{n} x_k \cdot BEZ_{k,n}(u)$$

$$y(u) = \sum_{k=0}^{n} y_k \cdot BEZ_{k,n}(u)$$

$$z(u) = \sum_{k=0}^{n} z_k \cdot BEZ_{k,n}(u)$$

Design a Bezier Curve controlled by 4 Points

A(1,1) B(2,3) C(4,3) D(6,4)

Sol

Control points $K = 0, 1, 2, 3$

$n = 4 - 1 \Rightarrow 3$

P_0 (1,1)

P_1 (2,3)

P_3 (4,3)

P_4 (6,4)

$$Q(u) = \sum_{k=0}^{n} P_k \cdot BEZ_{k,n}(u)$$

$$Q(u) = P_0 B_{0,3}(u) + P_1 B_{1,3}(u) + P_2 B_{2,3}(u) + P_3 B_{3,3}(u)$$

$$Q(u) = P_0 (1-u)^3 + P_1 3u(1-u)^2 + P_2 3u^2(1-u) +$$

$$P_3 u^3$$

$$BEZ_{k,n}(u) = \frac{n!}{k!(n-k)!} \cdot u^k \cdot (1-u)^{n-k}$$

$$BEZ_{0,3}(u) = \frac{3!}{0!\,3!} \cdot u^0 \cdot (1-u)^{3-0}$$

$$= 1 \cdot 1 \cdot (1-u)^3$$

$$\boxed{BEZ_{0,3}(u) = (1-u)^3}$$

$$BEZ_{1,3}(u) = \frac{3!}{1!(2!)} \cdot u^1 \cdot (1-u)^{3-1}$$

$$\boxed{BEZ_{1,3}(u) = 3 \cdot u \cdot (1-u)^2}$$

$$BEZ_{2,3}(u) = \frac{3!}{2!(1!)} \cdot u^2 \cdot (1-u)^{3-2}$$

$$\boxed{BEZ_{2,3}(u) = 3 \cdot u^2 \cdot (1-u)}$$

$$BEZ_{3,3}(u) = \frac{3!}{3!\,0!} \cdot u^3 \cdot (1-u)^{3-3}$$

$$\boxed{BEZ_{3,3}(u) = u^3}$$

$$x(u) = \sum_{k=0}^{n} x_k \cdot BEZ_{k,n}(u)$$

$$= x_0 BEZ_{0,3}(u) + x_1 BEZ_{1,3}(u) + x_3 BEZ_{2,3}(u) + x_4 BEZ_{3,3}(u)$$

$$= 1\cdot(1-u)^3 + 2\{3u\cdot(1-u)^2\} + 4\{3u^2(1-u)\} + 6u^3$$

$$x(u) = (1-u)^3 + 6u(1-u)^2 + 12u^2(1-u) + 6u^3$$

$P_0 = (x_0, y_0) = (1, 1)$
$P_1 = (x_1, y_1) = (2, 3)$
$P_2 = (x_2, y_2) = (4, 3)$
$P_3 = (x_3, y_3) = (6, 4)$

$$y(u) = 1(1-u)^3 + 3(3)u(1-u)^2 + 3(3)u^2(1-u) + 4u^3$$

$$y(u) = (1-u)^3 + 9u(1-u)^2 + 9u^2(1-u) + 4u^3$$

$0 \leq u \leq 1$

u	$x(u)$	$y(u)$
0	1	1
0.2	1.712	1.984
0.4	2.616	2.632
0.6	3.664	3.088
0.8	4.808	3.496
1	6	4