Computer Graphics

Introduction:

Computer Graphics is a field related to the generation of graphics using computers. It includes the creation, storage, and manipulation of images of objects. These objects come from diverse fields such as physical, mathematical, engineering, architectural, abstract structures and natural phenomenon. Computer graphics today is largely interactive, that is , the user controls the contents, structure, and appearance of images of the objects by using input devices, such as keyboard, mouse, or touch-sensitive panel on the screen.

Until the early 1980's computer graphics was a small, specialized field, largely because the hardware was expensive and graphics-based application programs that were easy to use and cost-effective were few. Then personal computers with built-in raster graphics displays-such as the Xerox Star, Apple Macintosh and the IBM PC-popularized the use of bitmap graphics for user-computer interaction. A bitmap is a ones and zeros representation of the rectangular array points on the screen. Each point is called a pixel, short for "Picture Elements" . Once bitmap graphics became affordable, and explosion of easy-to-use and inexpensive graphics-based applications soon followed. Graphics-based user interfaces allowed millions of new users to control simple, low-cost application programs, such as word-processors, spreadsheets, and drawing programs.

The concepts of a "desktop" now became a popular for organizing screen space. By means of a window manager, the user could create, position and resize rectangular screen areas called windows. This allowed user to switch among multiple activities just by pointing and clicking at the desired window, typically with a mouse. Besides windows, icons which represent data files, application program, file cabinets, mailboxes, printers, recycle bin, and so on, made the user-computer interaction more effective. By pointing and clicking the icons, users could activate the corresponding programs or objects, which replaced much of the typing of the commands used in earlier operating systems and computer applications.

Today, almost all interactive application programs, even those for manipulating text(e.g.. word processor) or numerical data (e.g. spreadsheet programs), use graphics extensively in the user interface and for visualizing and manipulating the application-specific objects.

Even people who do not use computers encounter computer graphics in TV commercials and as cinematic special effects. Thus computer graphics is and integral part of all computer user interfaces, and is indispensable for visualizing 2D, 3D objects in all most all areas such as education, science, engineering, medicine, commerce, the military, advertising, and entertainment. The theme is that learning how to program and use computers now includes learning how to use simple 2D graphics.

Early History of Computer Graphics

We need to take a brief look at the historical development of computer graphics to place today's system in context. Crude plotting of hardcopy devices such as teletypes and line printers dates from the early days of computing. The Whirlwind Computer developed in 1950 at the Massachusetts Institute of Technology(MIT) had computer-driven CRT displays for output. The SAGE air-defense system developed in the middle 1950s was the first to use command and control CRT display consoles on which operators identified targets with light pens(hand-held pointing devices that sense light emitted by objects on the screen). Later on Sketchpad system by Ivan Sutherland came in light. That was the beginning of modern interactive graphics. In this system, keyboard and light pen were used for pointing, making choices and drawing.

At the same time, it was becoming clear to computer, automobile, and aerospace manufacturers that CAD and computer-aided manufacturing(CAM) activities had enormous potential for automating drafting and other drawing-intensive activities. The General Motors CAD system for automobile design , and the Itek Digitek system for lens design, were pioneering efforts that showed the utility of graphical interaction in the iterative design cycles common in engineering. By the mid-60s , a number of commercial products using these systems had appeared.

At that time only the most technology-intensive organizations could use the interactive computer graphics whereas other used punch cards, a non-interactive system. Among the reasons for this were these:

- The high cost of the graphics hardware-at a time when automobiles cost a few thousand dollors, computers cost several millions of dollors, and the first computer displays cost more that a hundred thousand dollors.

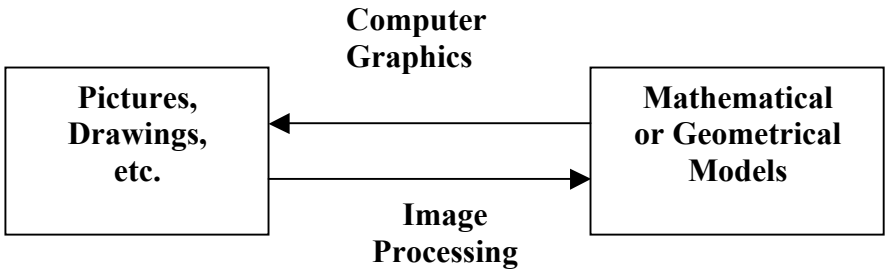
- The need for large-scale, expensive computing resources to support massive design database.

- The difficulty of writhing large, interactive programs using batch-oriented FORTRAN programming.

- One of a kind , non portable software, typically written for a particular manufacturer's display devices. When software is non-portable, moving to new display devices necessitates expensive and time-consuming rewriting of working programs.

This interactive computer graphics had a limited use when it started in the early sixties. But became very common once the Apple Macintosh and IBM PC appeared in the market with affordable cost.

The Difference between Computer Graphics and Image Processing:



- ***Computer Graphics*: Synthesize pictures from mathematical or geometrical models.**
- ***Image Processing*: analyze pictures to derive descriptions (often in mathematical or geometrical forms) of objects appeared in the pictures.**

Representative uses of Computer Graphics

Computer graphics is used today in many different areas of science, engineering, industry, business, education, entertainment, medicine, art and training, All of these are included in the following categories.

1. User interfaces

Most applications have user interfaces that rely on desktop windows systems to manage multiple simultaneous activities , and on point-and click facilities to allow users to select menu items, icons and objects on the screen. These activities fall under computer graphics. Typing is necessary only to input text to be stored and manipulated. For example, Word processing, spreadsheet, and desktop-publishing programs are the typical examples where user-interface techniques are implemented.

2. Plotting

Plotting 2D and 3D graphs of mathematical, physical, and economic functions use computer graphics extensively. The histograms, bar, and pie charts; the task-scheduling charts are the most commonly used plotting. These all are used to present meaningfully and concisely the trends and patterns of complex data.

3. Office automation and electronic publishing

Computer graphics has facilitated the office automation and electronic publishing which is also popularly known as desktop publishing, giving more power to the organizations to print the meaningful materials in-house. Office automation and electronic publishing can produce both traditional printed (Hardcopy) documents

and electronic(softcopy) documents that contain text, tables, graphs, and other forms of drawn or scanned-in graphics.

4. Computer Aided Drafting and Design

One of the major uses of computer graphics is to design components and systems of mechanical, electrical, electrochemical, and electronic devices, including structures such as buildings, automobile bodies, airplane and ship hulls, very large scale integrated (VLSI) chips, optical systems,. and telephone and computer networks. These designs are more frequently used to test the structural, electrical, and thermal properties of the systems.

5. Scientific and business Visualization

Generating computer graphics for scientific, engineering, and medical data sets is termed as scientific visualization whereas business visualization is related with the non scientific data sets such as those obtained in economics. Visualization makes easier to understand the trends and patterns inherent in the huge amount of data sets. It would , otherwise , be almost impossible to analyze those data numerically.

6. Simulation and modeling

Simulation is the imitation of the conditions like those , which is encountered in real life. Simulation thus helps to learn or to feel the conditions one might have to face in near future without being in danger at the beginning of the course. For example, astronauts can exercise the feeling of weightlessness in a simulator; similarly a pilot training can be conducted in flight simulator. The military tank simulator, the naval simulator, driving simulator, air traffic control simulator, heavy-duty vehicle simulator, and so on are some of the mostly used simulator in practice. Simulators are also used to optimize the system, for example the vehicle, observing the reactions of the driver during the operation of the simulator.

7. Entertainment

Disney movies such as Lion Kings and The Beauty of Beast, and other scientific movies like Jurassic Park, The lost world etc are the best example of the application of computer graphics in the field of entertainment. Instead of drawing all necessary frames with slightly changing scenes for the production of cartoon-film, only the key frames are sufficient for such cartoon-film where the in between frames are interpolated by the graphics system dramatically decreasing the cost of production while maintaining the quality. Computer and video games such FIFA, Doom ,Pools are few to name where graphics is used extensively.

8. Art and commerce

Here computer graphics is used to produce pictures that express a message and attract attention such as a new model of a car moving along the ring of the Saturn . These pictures are frequently seen at transportation terminals supermarkets , hotels etc. The slide production for commercial , scientific, or educational presentations is another cost effective use of computer graphics. One of such graphics packages is a PowerPoint.

9. Cartography

Cartography is a subject , which deals with the making of maps and charts. Computer graphics is used to produce both accurate and schematic representations of geographical and other natural phenomena from measurement data. Examples include geographic maps , oceanographic charts, weather maps, contour maps and population-density maps. Surfer is one of such graphics packages , which is extensively used for cartography.

Main Subjects in Computer Graphics Research

- **Mathematical and geometrical modeling**
- **Rendering algorithms**
- **Animation techniques**
- **Input and output technologies**
- **Graphics architecture**

Overview Of Data Visualization

In every field ,we need large number of information to be analyzed and study the behaviors of certain processes. Numerical simulations in supercomputers ,satellite cameras and other sources are amassing large amount of data files faster they can be interpreted. Scanning these huge amount of data files to determine their nature and relationships is a tedious job. But if the data are converted to a visual form it is very easier to infer various conclusions immediately. Producing graphical presentations for scientific, engineering and medical data sets and processes is generally referred to as scientific visualization. If the data sets are concerned with commerce , industry and other non scientific areas , it is called business visualization.

Definitions

- Visualization is the graphical presentation of information, with the goal of providing the viewer with a qualitative understanding of the information contents.
- Information may be data, processes, relations, or concepts.
- Graphical presentation may include manipulation of graphical entities (points, lines, shapes, images, text) and attributes (color, size, position, shape).
- Understanding may involve detection, measurement, and comparison, and is enhanced via interactive techniques and providing the information from multiple views and with multiple techniques.

Characteristics of Data

- Numeric, symbolic (or mix)
- Scalar, vector, or complex structure
- Various units
- Discrete or continuous
- Spatial, quantity, category, chronological , relational, structural
- Accurate or approximate
- Dense or sparse
- Ordered or non-ordered
- Disjoint or overlapping
- Binary, enumerated, multilevel
- Independent or dependent
- One dimensional or Multidimensional
- Single or multiple sets
- May have similarity or distance metric
- May have intuitive graphical representation (e.g. temperature with color)
- Has semantics which may be crucial in graphical consideration

Graphical entities and attributes

- ***Entity***: point, line, polyline, glyph, surface, solid, image, text etc.
- ***Attribute***: color/intensity, location, style, size, relative position/motion
    - ***Example***: Attribute of line may be color, style(dotted, solid, dashed) , thickness etc.

What do we see and how well do we see it?

- Different viewers perceive different graphical/spatial/color in different degrees
- Context varies our sensitivity
- According to one researcher (Cleveland), in increasing inaccuracy
    1. Position along a common scale
    2. Position along identical, non-aligned scales

3. Length
4. Angle/slope
5. Area
6. Volume
7. Shade/Saturation/intensity (informally derived)

- detection is proportional to percent change, not scale
- Stevens' law - perceived scale is proportional to a power of the actual scale.

What makes a good visualization?

- Effective: the viewer gets it (ease of interpretation)
- Accurate: sufficient for correct quantitative evaluation. Lie factor = size of visual effect/size of data effect
- Efficient: minimize data-ink ratio and chart-junk, show data, erase redundant data-ink
- Aesthetics: must not offend(hurt) viewer's senses (e.g. moire patterns)
- Adaptable: can adjust to serve multiple needs

Mapping data to graphics

- Examine cardinality of dimension with detectible variations in graphics
- Use scaling and offset to fit in range
- Use derived values (residuals, logs) to emphasize changes
- Use projections, other combinations, to compress information, get statistics
- Use random jiggling to separate overlaps
- Use multiple views to handle hidden relations, high dimensions
- Use effective grids, keys and labels to aid understanding

Interacting with the data

- Dynamically adjust mapping
- Tour data by varying views
- Labeling to get original data
- Deleting to eliminate clutter (disorder)
- Brushing/Highlighting to see correspondence in multiple views
- Zooming to focus attention
- Panning to explore neighborhoods

Common Techniques

- Charts: bar or pie
- Graphs: good for structure, relationships
- Plots: 1- to n-dimensional
- Maps: one of most effective
- Images: use color/intensity instead of distance (surfaces)
- 3-D surfaces and solids
- isosurfaces/slices

- transparency
- stereopsis : Taking picture in different viewpoint and combine.
- animation

Hardware Concepts

Input Devices

1. Tablet:

A tablet is digitizer. In general a digitizer is a device which is used to scan over an object, and to input a set of discrete coordinate positions. These positions can then be joined with straight-line segments to approximate the shape of the original object. A tablet digitizes an object detecting the position of a movable stylus (pencil-shaped device) or puck (link mouse with cross hairs for sighting positions) held in the user's hand. A tablet is flat surface, and its size of the tablet varies from about 6 by 6 inches up to 48 by 72 inches or more. The accuracy of the tablets usually falls below 0.2 mm. There are mainly three types of tablets.

a. Electrical tablet:

A grid of wires on ¼ to ½ inch centers is embedded in the tablet surface and electromagnetic signals generated by electrical pulses applied in sequence to the wires in the grid induce an electrical signal in a wire coil in the stylus (or puck). The strength of the signal induced by each pulse is used to determine the position of the stylus. The signal strength is also used to determine roughly how far the stylus is from the tablet. When the stylus is within ½ inch from the tablet, it is taken as "near" otherwise it is either "far" or "touching". When the stylus is "near" or "touching", a cursor is usually shown on the display to provide visual feedback to the user. A signal is sent to the computer when the tip of the stylus is pressed against the tablet, or when any button on the puck is pressed. The information provided by the tablet repeats 30 to 60 time per second.

b. Sonic tablet:

The sonic tablet uses sound waves to couple the stylus to microphones positioned on the periphery of the digitizing area. An electrical spark at the tip of the stylus creates sound bursts. The position of the stylus or the coordinate values is calculated using the delay between when the spark occurs and when its sound arrives at each microphone. the main advantage of sonic tablet is that it does not require a dedicated working area for the microphones can be placed on any surface to form the "tablet" work area. This facilitates digitizing drawing on thick books. Because in an electrical tablet this is not convenient for the stylus can not get closer to the tablet surface.

c. Resistive tablet:

The tablet is just a piece of glass coated with a thin layer of conducting material. When a buttery-powered stylus is activated at certain position, it emits high-frequency radio signals, which induces the radio signals on the conducting layer. The strength of

he signal received at the edges of the tablet is used to calculate the position of the stylus.

Several types of tablets are transparent, and thus can be backlit for digitizing x-rays films and photographic negatives. The resistive tablet can be used to digitize the objects on CRT because it can be curved to the shape of the CRT. The mechanism used in the electrical or sonic tablets can also be used to digitize the 3D objects.

2. Touch panel

The touch panel allows the users to point at the screen directly with a finger to move the cursor around the screen, or to select the icons. Following are the mostly used touch panels.

a. Optical touch panel

It uses a series of infra-red light emitting diodes (LED) along one vertical edge and along one horizontal edge of the panel. The opposite vertical and horizontal edges contain photo-detectors to form a grid of invisible infrared light beams over the display area. Touching the screen breaks one or two vertical and horizontal light beams, thereby indicating the finger's position. The cursor is then moved to this position, or the icon at this position is selected. It two parallel beams are broken, the finger is presumed to be centered between them; if one is broken, the finger is presumed to be on the beam. There is a low-resolution panel, which offers 10 to 50 positions in each direction.

b. Sonic panel:

Bursts of high-frequency sound waves traveling alternately horizontally and vertically are generated at the edge of the panel. Touching the screen causes part of each wave to be reflected back to its source. The screen position at the point of contact is then calculated using the time elapsed between when the wave is emitted and when it arrives back at the source. This is a high-resolution touch panel having about 500 positions in each direction.

c. Electrical touch panel:

It consists of slightly separated two transparent plates one coated with a thin layer of conducting material and the other with resistive material. When the panel is touched with a finger, the two plates are forced to touch at the point of contact thereby creating the touched position. The resolution of this touch panel is similar to that of sonic touch panel.

3. Light pen

It is a pencil-shaped device to determine the coordinates of a point on the screen where it is activated such as pressing the button. In raster display, Y is set at Y_{max} and X changes from 0 to X_{max} for the first scanning line. For second line, Y decreases by one and X again changes from 0 to $X_{max,}$ and so on. When the activated light pen "sees" a burst of light at certain position as the electron beam hits the phosphor

coating at that position, it generates a electric pulse, which is used to save the video controller's X and Y registers and interrupt the computer. By reading the saved values, the graphics package can determine the coordinates of the position seen by the light pen. Because of the following drawbacks the light pens are not popular now a days.

- Light pen obscures the screen image as it is pointed to the required spot
- Prolong use of it can cause arm fatigue
- It can not report the coordinates of a point that is completely black. As a remedy one can display a dark blue field in place of the regular image for a single frame time
- It gives sometimes false reading due to background lighting in a room

4. Keyboard

A keyboard creates a code such as ASCII uniquely corresponding to a pressed key. It usually consists of alphanumeric keys, function keys, cursor-control keys, and separate numeric pad. It is used to move the cursor, to select he menu item, pre-defined functions. In computer graphics keyboard is mainly used for entering screen coordinates and text, to invoke certain functions. Now-a-days ergonomically designed keyboard (Ergonomic keyboard) with removable palm rests is available. The slope of each half of the keyboard can be adjusted separately.

5. Mouse

A mouse is a small hand-held device used to position the cursor on the screen. Mice are relative devices, that is, they can be picked up, moved in space, and then put down gain without any change in the reported position. For this, the computer maintains the current mouse position, which is incremented or decremented by the mouse movements. Following are the mice, which are mostly used in computer graphics.

a. Mechanical mouse

When a roller in the base of this mechanical mouse is moved, a pair of orthogonally arranged toothed wheels, each placed in between a LED and a photo detector, interrupts the light path. the number of interrupts so generated are used to report the mouse movements to the computer.

b. Optical mouse

The optical mouse is used on a special pad having a grid of alternating light and dark lines. A LED on the bottom of the mouse directs a beam of light down onto the pad, from which it is reflected and sensed by the detectors on the bottom of the mouse. As the mouse is moved, the reflected light beam is broken each time a dark line is crossed. The number of pulses so generated, which is equal to the number of lines crossed, are used to report mouse movements to the computer.

Display devices

The display devices used in graphics system is video monitor. The most common video monitor is based on CRT technology.

Cathode Ray Tube (CRT)

- CRT are the most common display devices on computer today. A CRT is an evacuated glass tube, with a heating element on one end and a phosphor-coated screen on the other end.
- When a current flows through this heating element (filament) the conductivity of metal is reduced due to high temperature. These cause electrons to pile up on the filament.
- These electrons are attracted to a strong positive charge from the outer surface of the focusing anode cylinder.
- Due to the weaker negative charge inside the cylinder, the electrons head towards the anode forced into a beam and accelerated by the inner cylinder walls in just the way that water is speeds up when its flow though a small diameter pipe.
- The forwarding fast electron beam is called Cathode Ray. A cathode ray tube is shown in figure below.

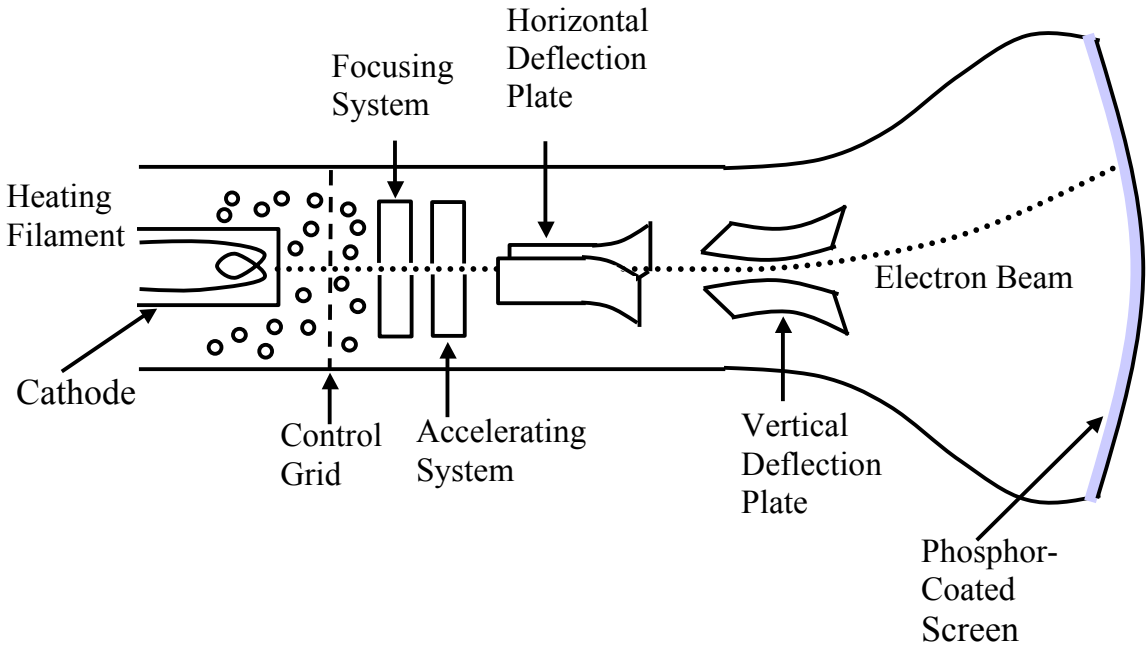


Figure :Cathode Ray Tube

- There are two sets of weakly charged deflection plates with oppositely charged, one positive and another negative. The first set displaces the beam up and down and the second displaces the beam left and right.
- The electrons are sent flying out of the neck of bottle (tube) until the smash into the phosphor coating on the other end.

- When electrons strike on phosphor coating, the phosphor then emits a small spot of light at each position contacted by electron beam. The glowing positions are used to represent the picture in the screen.
- The amount of light emitted by the phosphor coating depends on the no of electrons striking the screen. The brightness of the display is controlled by varying the voltage on the control grid.

Persistence:
- How long a phosphor continues to emit light after the electron beam is removed
- Persistence of phosphor is defined as **the time** it takes for emitted light to decay to **1/10 (10%)** of its original intensity. Range of persistence of different phosphors can react many seconds.
- Phosphors for graphical display have persistence of 10 to 60 microseconds. Phosphors with low persistence are useful for animation whereas high persistence phosphor is useful for highly complex, static pictures.

Refresh Rate:
- Light emitted by phosphor fades very rapidly, so to keep the drawn picture glowing constantly, it is required to redraw the picture repeatedly and quickly directing the electron beam back over the some point. The no of times/sec the image is redrawn to give a feeling of non-flickering pictures is called refresh-rate.
- If Refresh rate decreases, flicker develops.
- For refresh displays, it depends on picture complexity
- Refresh rate above which flickering stops and steady it may be called as critical fusion frequency(CFF).

Resolution:
Maximum number of points displayed horizontally and vertically without overlap on a display screen is called resolution. In other ways , resolution is referred as the no of points per inch(dpi/pixel per inch).

Raster-Scan Display

- Raster Scan Display is based on television technology. In raster-scan the electron beam is swept across the screen, one row at a time from top to bottom. No of scan line per second is called horizontal scan rate.
- As electron beam moves across each row, the beam intensity is turned on and off to create a pattern of illuminated spots. Picture definition is stored in a memory called frame buffer or refresh buffer. Frame buffer holds all the intensity value for screen points.

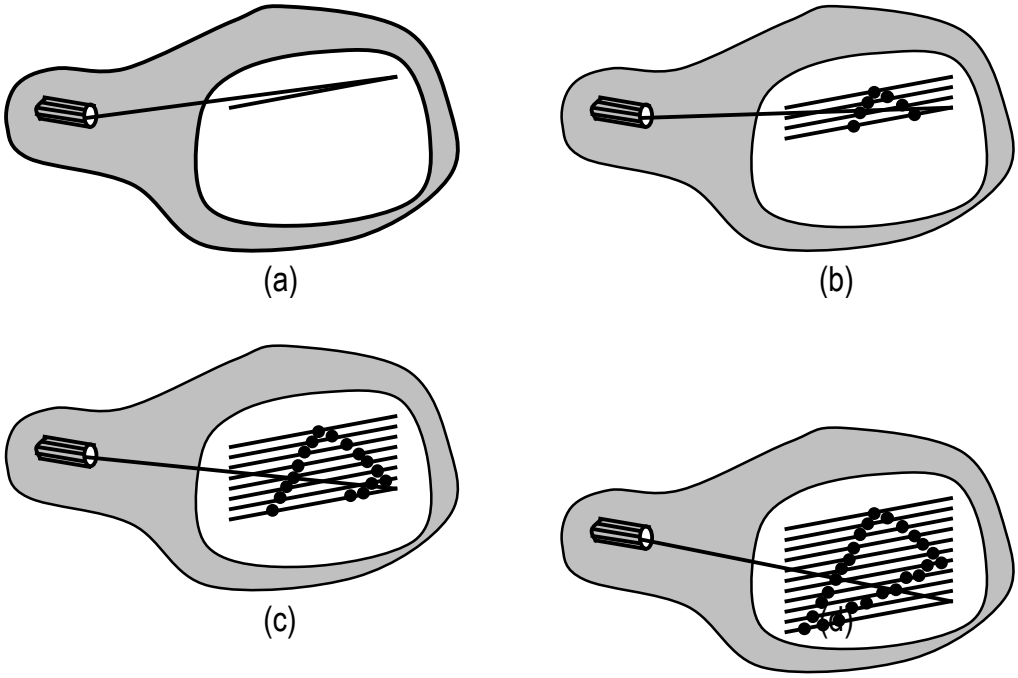


Figure: A raster-scan system displays an object as a set of points across each screen scan line

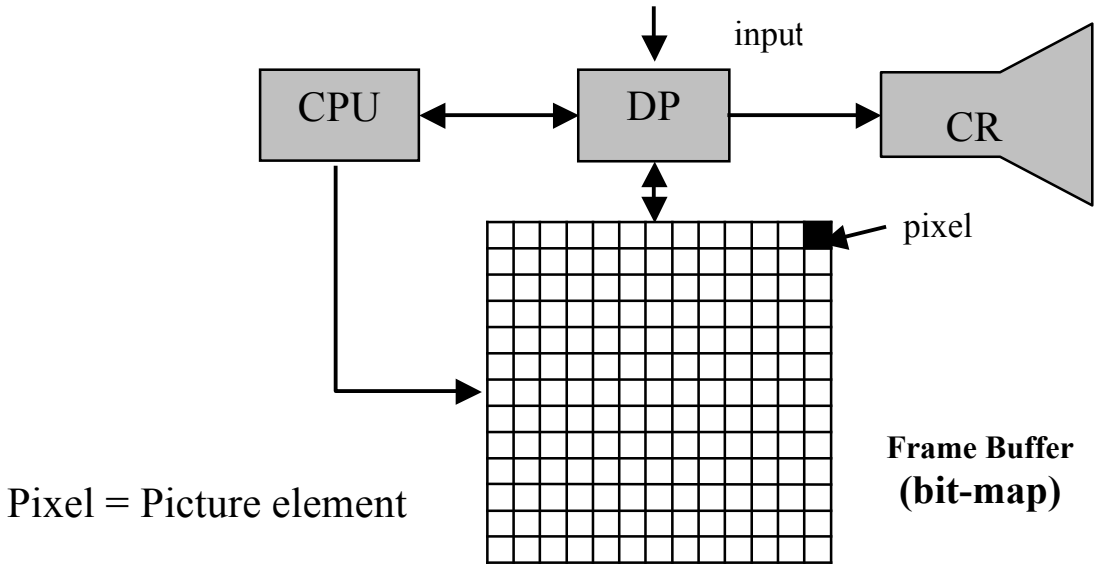


Figure: Raster Scan display system

- The stored intensity value is retrieved from frame buffer and painted on the scan line at a time. Home television are common examples using raster display
- Intensity range for pixel position depends on capability of raster system. For B/W system each point on screen are either on or off, so only one bit per pixel is needed to control the pixel intensity. To display color with varying intensity level, additional bits are needed. Up to 24 to 32 bit per pixel are included in high quality systems, which require more space of storage for the frame buffer, depending upon the resolution of the system.
- A system with 24 bit pixel and screen resolution 1024 × 1024 require 3 megabyte of storage in frame buffer.
  `1024*1024 pixel = 1024*1024*24 bits = 3 MB`

- The frame butter in B/W system stores a pixel with one bit per pixel so it is termed as bitmap. The frame buffer in multi bit per pixel storage, is called pixmap.
- Refreshing on Raster-Scan display is carried out at the rate of 60 or higher frames per second. 60 frames per second is also termed as 60 cycle per second usually used unit Hertz (HZ)
- Returning of electron beam from right end to deft end after refreshing each scan line is **horizontal retrace** . At the end of each frame, the electron beam returns to the top left corner to begin next frame called **vertical retrace**.

Interlaced: Display in two pass with interlacing.

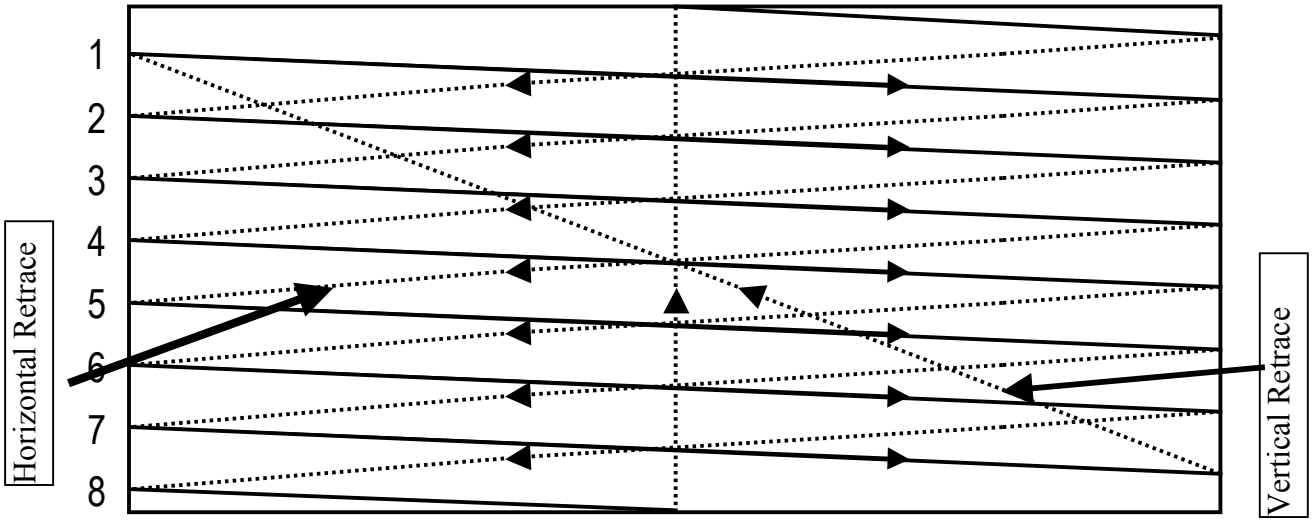


Figure: Horizontal retrace and Vertical retrace

Question*: Consider a RGB raster system is to be designed using 8 inch by 10 inch screen with a resolution of 100 pixels per inch in each direction. If we want to store 6 bits per pixel in the frame buffer, How much storage(in bytes) do we need for the frame buffer?*

Solution: Size of screen = 8 inch × 10 inch.
Pixel per inch(Resolution) = 100.
Then, Total no of pixels = 8×100 × 10 × 100 pixels
Bit per pixel storage = 8

Therefore Total storage required in frame buffer = (800 ×1000×8) bits
= (800 ×1000×8)/8 Bytes
= 800000 Bytes.

Frame Buffer Architecture of Raster Display

1. Indexed-color frame buffer.

In indexed –color frame buffer,

- Each pixel uses one byte in frame buffer.
- Each byte is an index into a color map.
- Each pixel may be one of 3^{24} colors, but only 256 color can be displayed at a time.
- There is a look-up table which has as many entries as there are pixel values.
- The table entry value is used to control the intensity or color of the CRT.

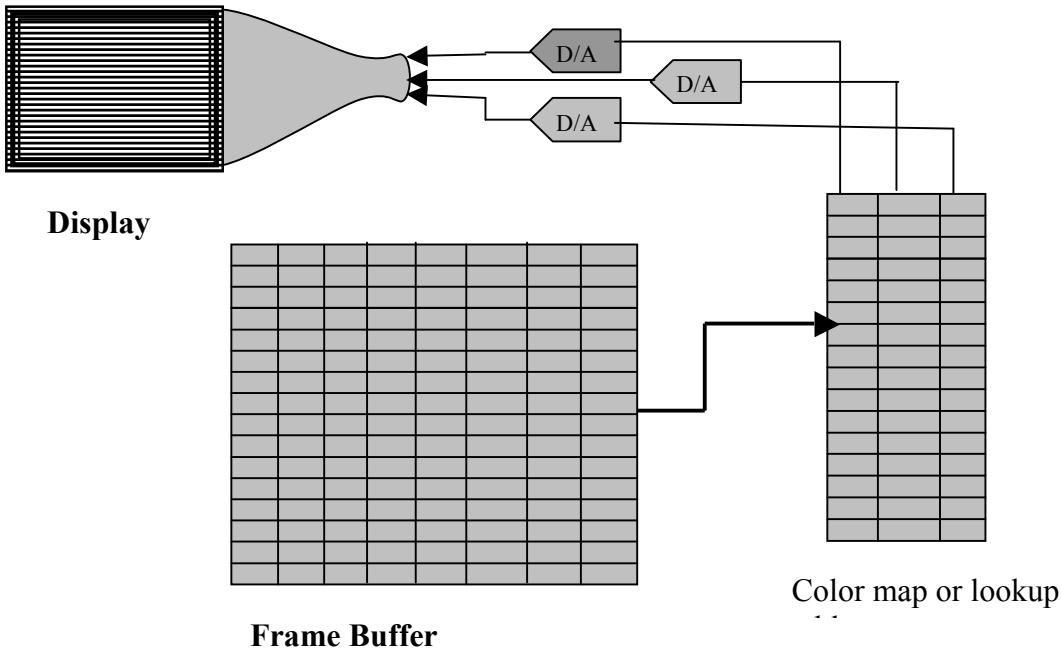


Figure: Indexed color frame buffer

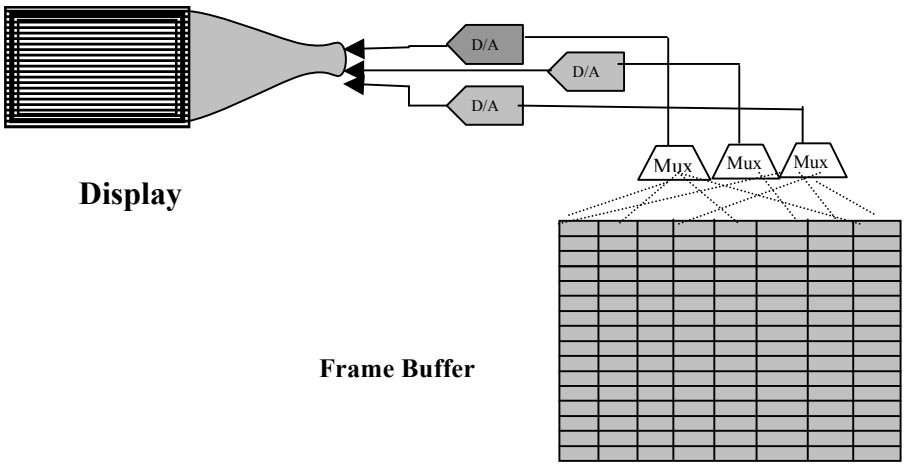


Figure : True Color Frame Buffer

2. **True-color frame buffer**: (24 bit or above): In true color frame buffer,
- Each pixel requires at least 3-bytes, one for each primary color (R,G,B)
- Sometimes combined with a look-up table per primary.
- Each pixel can be one of 2^{24} colors.

3 High-color frame buffer
- Popular PC/SVGA standard
- Pixels are packed in a short i.e. each primary color use 5 bit.
- Each pixel can be one of 2^{15} colors

Random scan display: (Vector display)

In random scan system, the CRT has the electron beam that is directed only to the parts of the screen where the picture is to be drawn. It draws a picture one line at a time, so it is also called **vector display** (or stroke writing or calligraphic display). The component lines of a picture are drawn and refreshed by random scan system in any specified order.

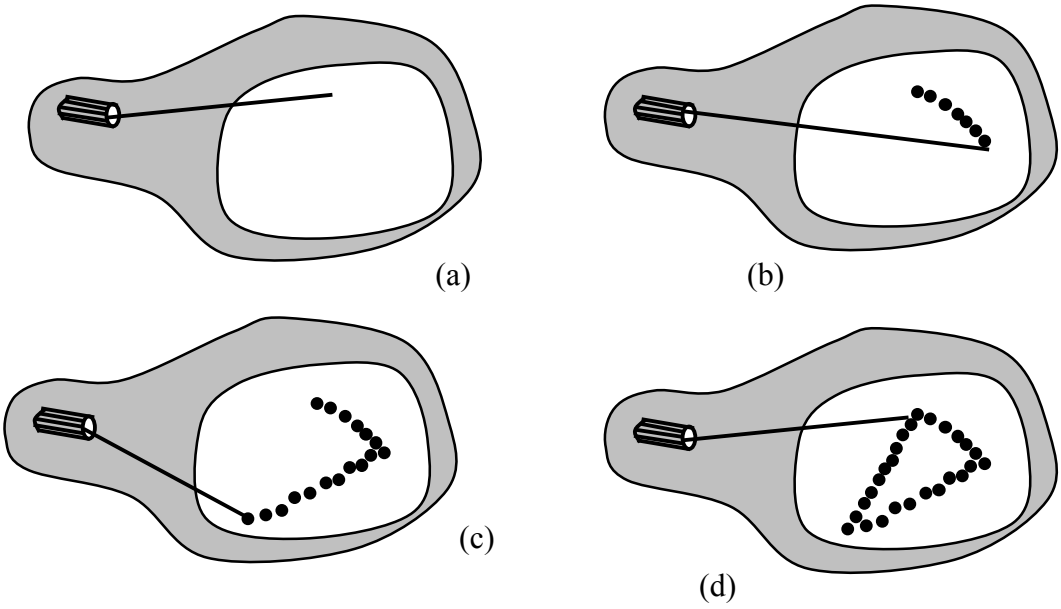


Figure: Random Scan Display

- The refresh rate of vector display depends upon the no of lines to be displayed for any image. Picture definition is stored as a set of line drawing instructions in an area of memory called the refresh display file (Display list or display file)
- To display a picture, the system cycles through the set of commands (line drawing) in the display file. After all commands have been processed, the system cycles back to the first line command in the list.
- Random scan systems are designed for drawing all component lines 30 to60 times per second. Such systems are designed for line-drawing applications and can not display realistic shaded scenes. Since CRT beam directly follows the line path, the vector display system produce smooth line.

Color CRT

In color CRT, the phosphor on the face of CRT screen are laid into different fashion. Depending on the technology of CRT there are two methods for displaying the color pictures into the screen.

1. Beam penetration method 2. Shadow mask method

Beam Penetration method:

This method is commonly used for random scan display or vector display. In random scan display CRT, the two layers of phosphor usually red and green are coated on CRT screen. Display color depends upon how far electrons beam penetrate the phosphor layers.

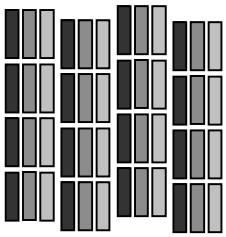
Slow electron excite only red layer so that we can see red color displayed on the screen pixel where the beam strikes. Fast electron beam excite green layer penetrating the red layer and we can see the green color displayed at the corresponding position. Intermediate is combination of red and green so two additional colors are possible – orange and yellow.

So only four colors are possible so no good quality picture in this type of display method.

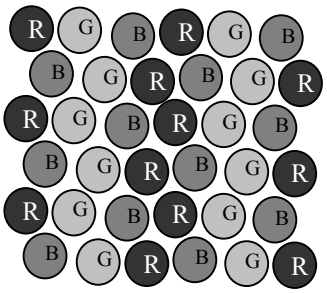
Shadow Mask Method:

Shadow mask method is used for raster scan system so they can produce wide range of colors. In shadow mask color CRT, the phosphor on the face of the screen are laid out in a precise geometric pattern. There are two primary variations.

1. The stripe pattern of inline tube
2. The delta pattern of delta tube



Stripe pattern



Delta Pattern

- In color CRT, the neck of tube, there are three electron guns, one for each red, green and blue colors. In phosphor coating there may be either strips one for each primary color, for a single pixel or there may be three dots one for each pixel in delta fashion.
- Special metal plate called a shadow mask is placed just behind the phosphor coating to cover front face.
- The mask is aligned so that it simultaneously allow each electron beam to see only the phosphor of its assigned color and block the phosphor of other two color.

Depending on the pattern of coating of phosphor, two types of raster scan color CRT are commonly used using shadow mask method.

1. Delta-Delta CRT:

- In delta-delta CRT, three electron beams one for each R,G,B colors are deflected and focused as a group onto shadow mask, which contains a series of holes aligned with the phosphor dots.
-

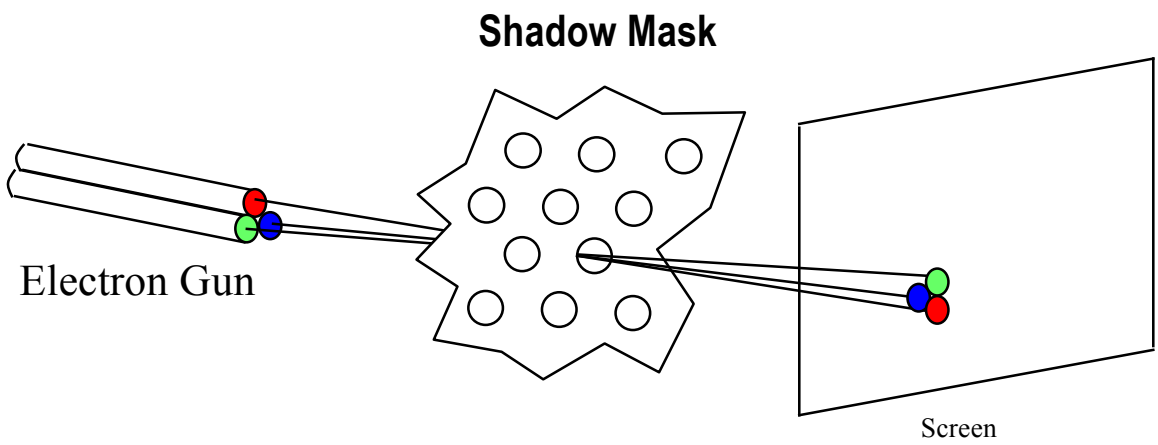


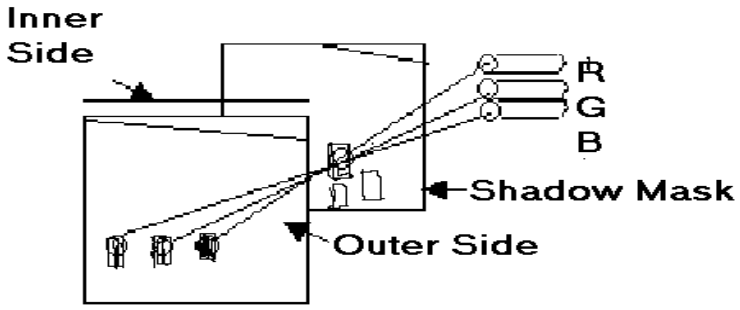
Figure: Shadow mask in Delta-Delta CRT

- Inner side of viewing has several groups of closely spaced red ,green and blue phosphor dot called triad in delta fashion.
- Thin metal plate adjusted with many holes near to inner surface called shadow mask which is mounted in such a way that each hole aligned with respective triad.
- Triad are so small that is perceived as a mixture of colors. When three beams pass through a hole in shadow mask, they activate the dot triangle to illuminate an small spot colored on the screen.
- The color variation in shadow mask CRT can be obtained by varying the intensity level of the three electron guns.

The main draw back of this CRT is due to difficulty for the alignment of shadow mask hole and respective triads.

A precision inline CRT:

This CRT uses strips pattern instead of delta pattern. Three strips one for each R, G, B color are used for a single pixel along a scan line so called inline. This eliminates the drawbacks of delta-delta CRT at the cost of slight reduction of image sharpness at the edge of the tube.



- Normally 1000 scan lines are displayed in this method. Three beams simultaneously expose three inline phosphor dots along scan line.

Architecture of Raster Scan System:

The raster graphics systems typically consists of several processing units. CPU is the main processing unit of computer systems. Besides CPU, graphics system consists of a special purpose processor called video controller or display processor. The display processor controls the operation of the display device.

The organization of raster system is as shown below

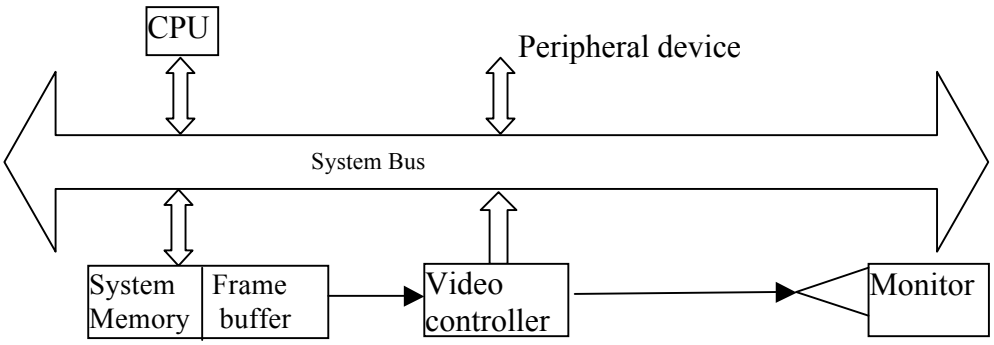


Figure: A simple Raster System.

- A fixed area of system memory is reserved for the frame buffer. The video controller has the direct access to the frame buffer for refreshing the screen.
- The video controller cycles through the frame buffer, one scan line at a time, typically at 60 times per second or higher. The contents of frame buffer are used to control the CRT beam's intensity or color.

The video controller:

The video controller is organized as in figure below. The raster-scan generator produces deflection signals that generate the raster scan and also controls the X and Y address registers, which in turn defines memory location to be accessed next. Assume

that the frame buffer is addressed in X from 0 to X_{max} and in Y from 0 to Y_{max} then, at the start of each refresh cycle, X address register is set to 0 and Y register is set to 0 (top scan line).

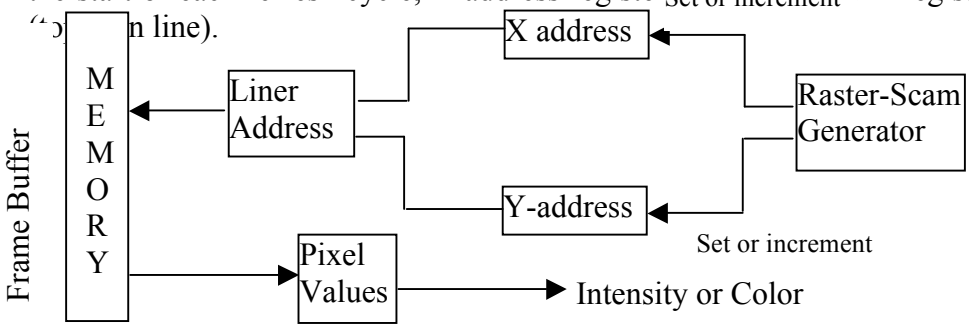


Figure: Organization of Video Controller.

As first scan line is generated, the X address is incremented up to X_{max}. Each pixel value is fetched and used to control the intensity of CRT beam. After first scan line X address is reset to 0 and Y address is incremented by 1. The process is continued until the last scan line ($Y=Y_{max}$) is generated.

Raster-Scan Display Processor:

The raster scan with a peripheral display processor is a common architecture that avoids the disadvantage of simple raster scan system. It includes a separate graphics processor to perform graphics functions such as scan conversion and raster operation and a separate frame buffer for image refresh.

The display processor has its own separate memory called display processor memory.

- System memory holds data and those programs that execute on the CPU, and the application program, graphics packages and OS.
- The display processor memory holds data plus the program that perform scan conversion and raster operations.
- The frame buffer stores displayable image created by scan conversion and raster operations.

The organization is given below in figure

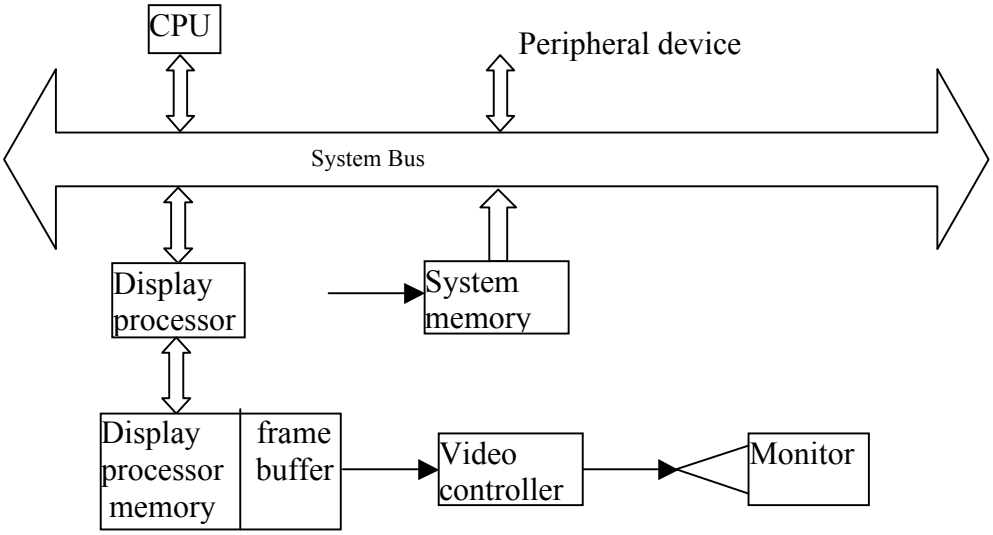


Figure: Architecture Raster scan system with display processor

2. **Vector Display System.**

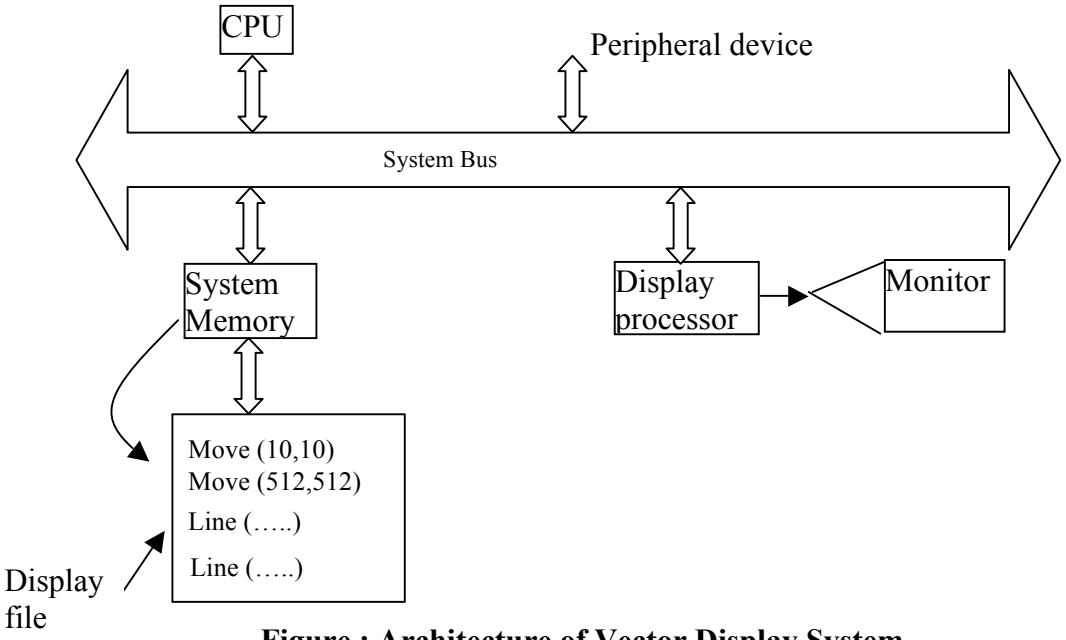


Figure : Architecture of Vector Display System

- Vector display system consists of several units along with peripheral devices. The display processor is also called as graphics controller.
- Graphics package creates a display list and stores in systems memory (consists of points and line drawing commands) called display list or display file.
- Refresh time around so cycle per second.
- Vector display technology is used in monochromatic or beam penetration color CRT.
- Graphics are drawn on a vector display system by directing the electron beam along component line.

Advantages:

- Can produce output with high resolutions.
- Better for animation than raster system since only end point information is needed.

Disadvantages:

- Cannot fill area with pattern and manipulate bits.
- Refreshing image depends upon its complexity.

Line Drawing Algorithms.

The slope-intercept equation of a straight line is:

$y = mx + b$ where m = slope of line and , b = y-intercept.

for any two given points (x_1, y_1) and (x_2, y_2)

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

$\therefore b = y - \frac{y_2 - y_1}{x_2 - x_1} x$ from above equation i.e. $y = mx + b$

At any point (x_k, y_k)

$y_k = mx_k + b$ ……. 1

At (x_{k+1}, y_{k+1}),

$y_{k+1} = mx_{k+1} + b$ ……2.

subtracting 1 from 2 we get-

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

Here $(y_{k+1} - y_k)$ is increment in y as corresponding increment in x.

$\therefore \Delta y = m.\Delta x$

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

For incremental algorithm in line drawing ,

- Increment *x* by 1
- Computer corresponding *y* and display pixel at position *(x_i, round (y_i))*

Problem: Floating point multiplication & addition

- The round function.

DDA line Algorithm:

The digital differential analyzer (DDA) is a scan conversion line drawing algorithm based on calculating either Δx or Δy form the equation,

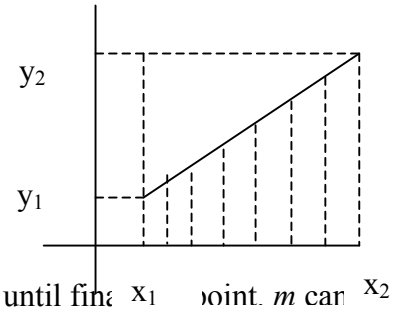
$$\Delta y = m.\Delta x.$$

We sample the line at unit intervals in one co-ordinate and determine the corresponding integer values nearest to the line path for the other co-ordinates.

Consider a line with positive slope.

If m<=1, we sample x co-ordinate. So $\Delta x = 1$ and compute each successive *y* value as:

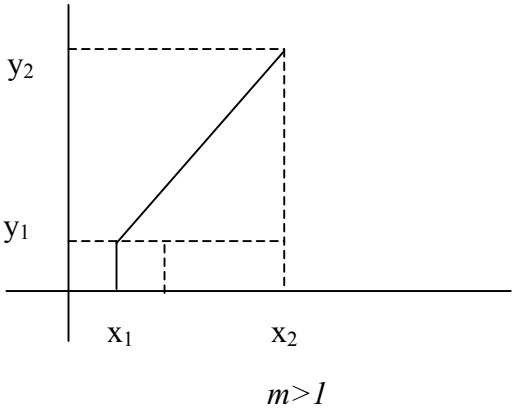
$$y_{k+1} = y_k + m \qquad \because m = \frac{\Delta y}{\Delta x}, \Delta x = 1$$



Here *k* takes value from starting point and increase by 1 until fina point. *m* can any real value between 0 and 1. *m<=1*

For line with positive slope greater than 1, we sample $\Delta y = 1$ and calculate corresponding x values as

$$x_{k+1} = x_k + \frac{1}{m} \quad \because m = \frac{\Delta y}{\Delta x}, \Delta y = 1$$



$m>1$

The above equations are under the assumption that the lines are processed from left to right. i.e. left end point is starting. If the processing is from right to left, we can sample $\Delta y = -1$ for line $|m|<1$

$$\therefore y_{k+1} = y_k - m,$$

If |m|>1, $\Delta y = -1$ and calculate

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

The complete C function for DDA algorithm is.

```
void lineDDA (in x1, int y1, int x2, int y2)
{
      int dx, dy, steps, k;
      float incrx; incry; x,y;
      dx=x2-x1;
      dy=y2-y1;
       if (abs(dx)>abs(dy))
            steps=abs(dx);
      else
            steps=abs(dy);
      incrx=dx/steps;
      incry=dy/steps;
      x=x1;     /* first point to plot */
      y=y1;
      putpixel(round(x),  round(y),1);  //1  is  used  for
color
      for (k=1;k<=steps;k++)
      {
            x = x + incrx;
            y = y + incry;
            putpixel(round(x),round(y),1);
      }
}
```

The DDA algorithm is faster method for calculating pixel position but it has problems:

- *m* is stored in floating point number.
- round of error
- error accumulates when we proceed line.

- so line will move away from actual line path for long line

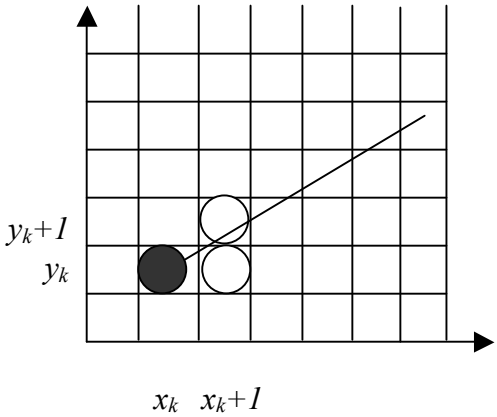
Bresenham's Line algorithm:

An accurate and efficient line generating algorithm, developed by Bresenham that scan converts lines only using integer calculation to find the next *(x,y)* position to plot. It avoids incremental error accumulation.

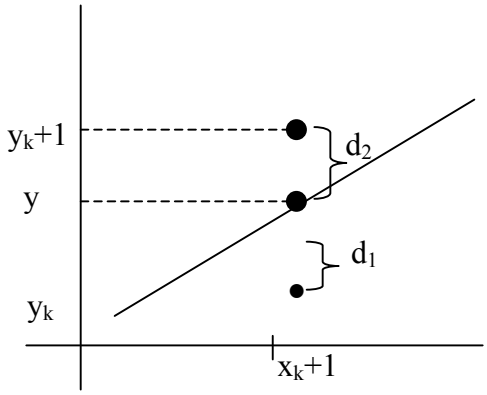
Line with positive slope less than 1 (0<m<1)

Pixel position along the line path are determined by sampling at unit x intervals. Starting from left end point, we step to each successive column and plot the pixel closest to line path.

Assume that (x_k,y_k) is pixel at k^{th} step then next point to plot may be either (x_k+1, y_k) or (x_k+1, y_k+1)



At sampling position x_k+1, we label vertical pixel separation from line path as d_1 & d_2 as in figure .
The y-coordinate on the mathematical line path at pixel column x_k+1 is $y=m(x_k+1)+b$



Then $d_1 = y - y_k$
$= m(x_k + 1)+b - y_k$
$d_2 = (y_k + 1) - y$
$= (y_k + 1)-m(x_k + 1)-b$
Now $d_1-d_2 = 2m(x_k + 1) - (y_k + 1)-y_k+2b$
$= 2m(x_k + 1) - 2y_k + 2b - 1$

A decision parameter p_k for the k^{th} step in the line algorithm can be obtained by substituting $m = \frac{\Delta y}{\Delta x}$ in above eq^n and defining

$$p_k = \Delta x(d_1 - d_2)$$
$$= \Delta x[2\frac{\Delta y}{\Delta x}(x_k + 1) - 2y_k + 2b - 1]$$
$$= 2\Delta y.x_k - 2\Delta x.y_k + 2\Delta y + \Delta x(2b - 1)$$
$$= 2\Delta y.x_k - 2\Delta x.y_k + c$$

Where the constant $c = 2\Delta y + \Delta x(2b - 1)$ which is independent.

If decision parameter p_k is negative i.e. $d_1<d_2$, pixel at y_k is closer to the line path than pixel at y_k+1 . In this case we plot lower pixel. *(x_k+1 , y_k). other wise plot upper pixel (x_k+1,y_k+1).*

Co-ordinate change along the line occur in unit steps in either x, or y direction. Therefore we can obtain the values of successive decision parameters using incremental integer calculations.

At step $k+1$, p_{k+1} is evaluated as.

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

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

Since $x_{k+1} = x_k + 1$

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

The term $y_{k+1} - y_k$ is either 0 or 1 depending upon the sign of p_k.

The first decision parameter p_0 is evaluated as.

$$p_o = 2\Delta y - \Delta x$$

and successively we can calculate decision parameter as

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

so if p_k is negative $y_{k+1} = y_k$ so $p_{k+1} = p_k + 2\Delta y$

otherwise $y_{k+1} = y_k+1$, *then* $p_{k+1} = p_k + 2\Delta y - 2\Delta x$

Algorithm:

1. Input the two line endpoint and store the left endpoint at (x_o, y_o)
2. Load (x_o, y_o) in to frame buffer, i.e. Plot the first point.
3. Calculate constants $2\Delta x, 2\Delta y$ calculating $\Delta x, \Delta y$ and obtain first decision parameter value as
   $$p_o = 2\Delta y - \Delta x$$
4. At each x_k along the line, starting at k=0, perform the following test,
   if $p_k < 0$, next point is $(x_k + 1, y_k)$
   $$p_{k+1} = p_k + 2\Delta y$$
   otherwise
   next point to plot is $(x_k + 1, y_k + 1)$
   $$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$
5. Repeat step 4 Δx times.

Function implementation in C

```
void lineBresenham (int x1, int y1, int x2, int y2)
{
   int x, y, dx, dy, pk, k xEnd;
   dx=abs(x2-x1);
   dy=abs(y2-y1);
   if(x1>x2)
   {
        x = x2;
        y = y2;
```

```
            xEnd = x1;
      }
     else
     {
            x = x1;
            y = y1;
            xEnd = x2;
   }
   putixel (x,y,1);
  pk=2*dy-dx;
   while (x<=xEnd)
   {
      if(pk<0)
      {
            x=x+1;
            y=y;
            pk=pk+2*dy;
      }
      else
      {
            x=x+1;
            y=y+1;
            pk= pk+2*dy-2*dx
      }
     putpixel (x,y,1);
 }
}
```

Brasenham[s] algorithm is generalized to lines with arbitrary slope by considering the symmetry between the various octants & quadrants of xy plane.

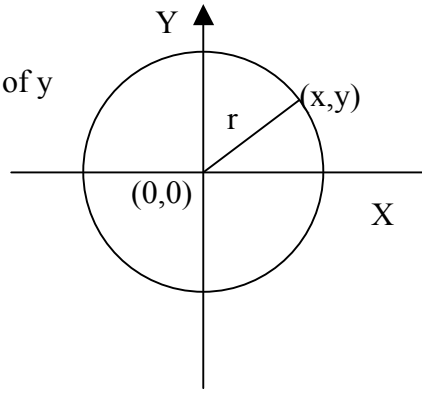
For a line positive slope greater than 1, we simply interchange the role of x & y.

Algorithm for Circle

Simple Algorithm:

The equation of circle centered at origin and radius r is given by $x^2 + y^2 = r^2$

$\Rightarrow y = \pm\sqrt{r^2 - x^2}$



- Increment x in unit steps and determine corresponding value of y from the equation above. Then set pixel at position *(x,y).*
- The steps are taken from *–r* to *+r* .
- In computer graphics, we take origin at upper left corner point on the display screen i.e. first pixel of the screen so any visible circle drawn would be centered at point other than (0,0). If center of circle is *(xc,yc)* then the calculated point from origin center should be moved to pixel position by *(x+xc, y+yc)*.

In general the equation of circle centered at *(xc,yc)* and radius r is

$$(x - xc)^2 + (y - yc)^2 = r^2$$
$$\Rightarrow y = yc \pm \sqrt{r^2 - (x - xc^2} \quad \text{.....(1)}$$

Use this equation to calculate the position of points on the circle. Take unit step from xc-r to *xc+r* for x value and calculate the corresponding value of y- position for pixel position (x,y). This algorithm is simple but,

- Time consuming – square root and squares computations
- Non – uniform spacing , due to changing slope of curve. If non-uniform spacing is avoided by interchanging x and y for slope |m|>y, this leads to more computation.

Following program demonstrates the simple computation of circle using the above equation (1)

```
//program for circle
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>
#include<graphics.h>
#define SQUARE(x) ((x)*(x))
void drawcircle(int ,int,int);
void main()
{
      int gd,gm,err;
      int xc,yc,r;
      gd=DETECT;
      initgraph(&gd,&gm,"\\tc\\bgi");
      err=graphresult();
      if(err!=0)
      {
            printf("ERROR:%s",grapherrormsg(err));
            printf("\nPress a key..");
            getch();
            exit(1);
      }
      xc=getmaxx()/2;
      yc=getmaxy()/2;
      r=50;
      drawcircle(xc,yc,r);
      getch();
      closegraph();
}//end main
void drawcircle(int xc,int yc,int r)
{
      int i,x,y,y1;
      for(i=xc-r;i<=xc+r;i++)
      {
            x=i;
            y=yc+sqrt(SQUARE(r)-SQUARE(x-xc));
            y1=yc-sqrt(SQUARE(r)-SQUARE(x-xc));
            putpixel(x,y,1);
            putpixel(x,y1,1);
      }
```

Drawing circle using polar equations

If (x,y) be any point on the circle boundary with center (0,0) and radius r, then

$x = r\cos\theta$

$x = r\sin\theta$

i.e. $(x, y) = (r\cos\theta, r\sin\theta)$

To draw circle using these co-ordinates approach, just increment angle starting from 0 to 360. Compute (x,y) position corresponding to increment angle. Which draws circle centered at origin, but the circle centered at origin is not visible completely on the screen since (0,0) is the starting pixel of the screen. If center of circle is given by (xc,yc) then the pixel position (x,y) on the circle path will be computed as

$x = xc + r\cos\theta$

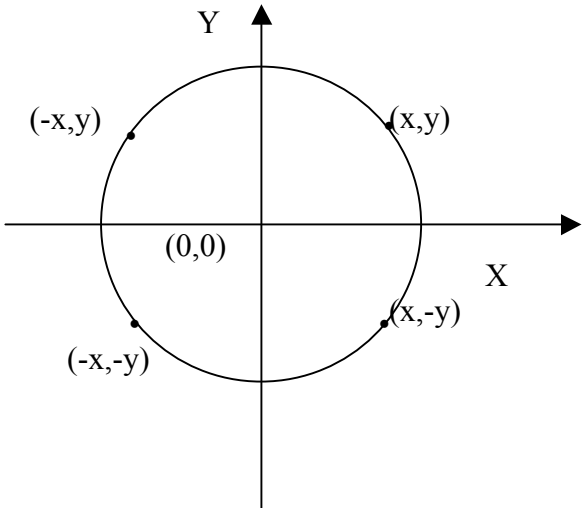
$y = yc + r\sin\theta$

Circle Function to draw circle using the polar transformation:

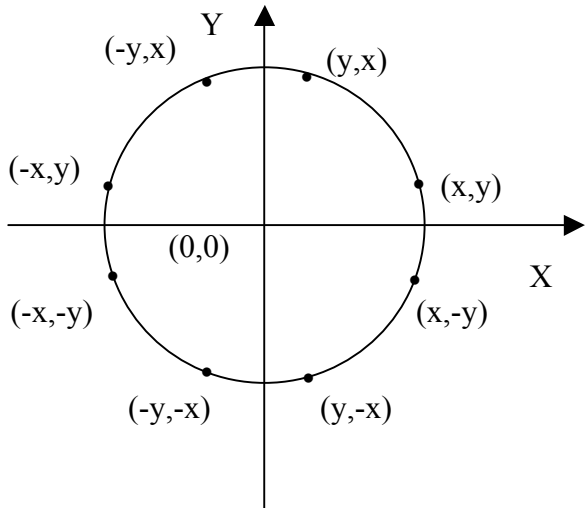
```
void drawcircle(int xc,int yc,int r)
{
      int x,y;
      float theta;
      const float PI=3.14;

      for(theta=0.0;theta<=360;theta+=1)
      {
            x= xc+r*cos(theta*PI/180.0);
            y= yc+r*sin(theta*PI/180.0);
            putpixel(x,y,1);

      }
}
```

Symmetry in circle scan conversion:

We can reduce the time required for circle generation by using the symmetries in a circle e.g. 4-way or 8-way symmetry. So we only need to generate the points for one quadrant or octants and then use the symmetry to determine all the other points.



4-way symmetry



8-way symmetry

Problem of computation still persists using symmetry since there are square roots, trigonometric functions are still not eliminated in above algorithms.

Mid point circle Algorithm:

In mid point circle algorithm, we sample at unit intervals and determine the closest pixel position to the specified circle path at each step.

For a given radius r, and screen center position (xc, yc), we can set up our algorithm to calculate pixel positions around a circle path centered at (0,0) and then each calculated pixel position (x, y) is moved to its proper position by adding xc to x and yc to y.

i.e. $x = x + xc$, $y = y + yc$.

To apply the mid point method, we define a circle function as:

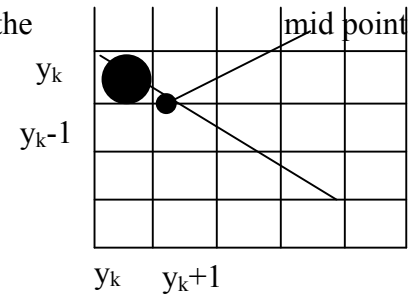
$$f_{circle} = x^2 + y^2 - r^2$$

To summarize the relative position of point (x, y) by checking sign of f_{circle} function,

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

The circle function tests are performed for the mid positions between pixels near the circle path at each sampling step. Thus the circle function is decision parameter in mid point algorithm.

The figure, shows the mid point between the two candidate pixel at sampling position $x_k + 1$, Assuming we have just plotted the pixel (x_k, y_k), we next need to determine whether the pixel at position $(x_k + 1, y_k)$ *or* $(x_k + 1, y_k - 1)$ is closer to the circle.



Our decision parameter is circle function evaluated at the mid point

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

$$= (x_k + 1)^2 + (y_k - 1/2)^2 - r^2 = (x_k + 1)^2 + y_k{}^2 - y_k + \frac{1}{4} - r^2$$

If $p_k < 0$, then mid-point lies inside the circle, so point at y_k is closer to boundary otherwise, $y_k - 1$ closer to choose next pixel position.

Successive decision parameters are obtained by incremental calculation. The decision parameter for next position is calculated by evaluating circle function at sampling position $x_{k+1} + 1$ i.e. $x_k + 2$ as

$$p_{k+1} = f_{circle}(x_{k+1}+1, y_{k+1}-\frac{1}{2})$$
$$= \{(x_{k+1}+1)\}^2 + (y_{k+1}-\frac{1}{2})^2 - r^2$$
$$= (x_{k+1})^2 + 2x_{k+1} + 1 + (y_{k+1})^2 - (y_{k+1}) + \frac{1}{4} - r^2$$

Now, $p_{k+1} - p_k = 2x_{k+1} + (y^2{}_{k+1} - y_k{}^2) - (y_{k+1} - y_k) + 1$

i.e. $p_{k+1} = p_k + 2x_{k+1} + (y^2{}_{k+1} - y_k{}^2) - (y_{k+1} - y_k) + 1$

Where y_{k+1} is either y_k or $y_k - 1$ depending upon sign of p_k. and $x_{k+1} = x_k + 1$

If p_k is negative, $y_{k+1} = y_k$ so we get,

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

If p_k is positive, $y_{k+1} = y_k - 1$ so we get,

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

Where $2x_{k+1} = 2x_k + 2$

$$2y_{k+1} = 2y_k - 2$$

At the start position, (0,r), these two terms have the values 0 an 2r, respectively. Each successive values are obtained by adding 2 to the previous value of 2x and subtracting 2 from previous value of 2y.

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

$$p_0 = f_{circle}(1, r - 1/2)$$
$$= 1 + (r - 1/2)^2 - r^2$$
$$= 1 + r^2 - r + \frac{1}{4} - r^2$$
$$= \frac{5}{4} - r$$

If p_0 is specified in integer,

$$p_0 = 1 - r.$$

The Algorithm:

1. Input radius r and circle centre (x_c, y_c), and obtain the first point on circle centered at origin as.
   $(x_0, y_0) = (0, r)$.
2. Calculate initial decision parameter
   $p_0 = 5/4 - r$
3. At each x_k position, starting at $k = 0$, perform the tests:
   If $p_k < 0$ next point along the circle centre at $(0,0)$ is $(x_k + 1, y_k)$

$$p_{k+1} = p_k + 2x_{k+1} + 1)$$

Otherwise, the next point along circle is $(x_k + 1, y_k - 1)$

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

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

4. Determine symmetry point on the other seven octants.
5. Move each calculated pixels positions (x, y) in to circle path centered at (x_c, y_c) as

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

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

Program for mid point circle algorithm in C

//mid point circle algorithm

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
void drawpoints(int,int,int,int);
void drawcircle(int,int,int);

void main(void)
{
   /* request auto detection */
   int gdriver = DETECT, gmode, errorcode;
   int xc,yc,r;
  /* initialize graphics and local
     variables */
   initgraph(&gdriver, &gmode, "\\tc\\bgi");

   /* read result of initialization */
   errorcode = graphresult();
   if (errorcode != grOk)  /* an error
       occurred */
   {
      printf("Graphics error: %s\n", grapherrormsg(errorcode));
      printf("Press any key to halt:");
      getch();
      exit(1); /* terminate with an error
              code */
   }
  printf("Enter the center co-ordinates:");
  scanf("%d%d",&xc,&yc);
  printf("Enter the radius");
  scanf("%d",&r);
  drawcircle(xc,yc,r);
  getch();
  closegraph();
}
void drawpoints(int x,int y, int xc,int yc)
{
      putpixel(xc+x,yc+y,1);
      putpixel(xc-x,yc+y,1);
      putpixel(xc+x,yc-y,1);
```

```
        putpixel(xc-x,yc-y,1);
        putpixel(xc+y,yc+x,1);
        putpixel(xc-y,yc+x,1);
        putpixel(xc+y,yc-x,1);
        putpixel(xc-y,yc-x,1);
}
void drawcircle(int xc,int yc,int r)
{
        int p,x,y;
        x=0;
        y=r;
        drawpoints(x,y,xc,yc);
        p=1-r;
        while(x<y)
        {
                if(p<0)
                {
                        x=x+1;
                        p=p+2*x+1;
                }
                else
                {
                        x=x+1;
                        y=y-1;
                        p=p+2*(x-y)+1;
                }

                drawpoints(x,y,xc,yc);
        }
}
```

Ellipse Algorithm

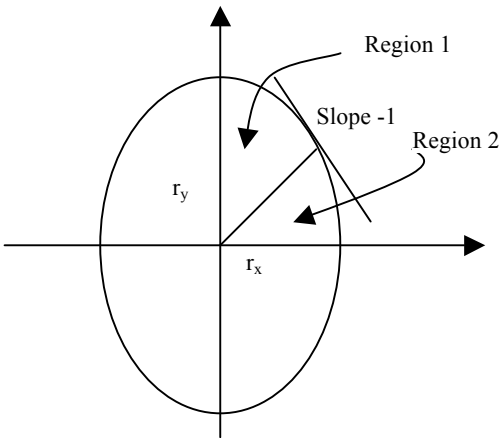
The basic algorithm for drawing ellipse is same as circle computing x and y position at the boundary of the ellipse from the equation of ellipse directly.

We have equation of ellipse centered at origin (0,0) is

$\frac{x^2}{r_x^2} + \frac{y^2}{r_y^2} = 1$ which gives

$y = \pm \frac{r_y}{r_x}\sqrt{(r_x^2 - x^2)}$ .....(1)

Stepping unit interval in x direction from $-r_x$ to r_x we can get corresponding y value at each x position which gives the ellipse boundary co-ordinates. Plotting these computed points we can get the ellipse.



Figure

If center of ellipse is any arbitrary point ***(xc,yc)*** then the equation of ellipse can be written as

$$\frac{(x-xc)^2}{r_x^{\,2}} + \frac{(y-yc)^2}{r_y^{\,2}} = 1$$

i.e. $y = yc \pm \frac{r_y}{r_x}\sqrt{r_x^{\,2} - (x-xc)^2}$ ----------(2)

for any point ***(x,y)***on the boundary of the ellipse If major axis of ellipse with major axis along X-axis the algorithm based on the direct computation of ellipse boundary points can be summarized as

1. Input the center of ellipse ***(xc,yc)*** x-radius ***xr*** and y-radius *yr*.
2. For each x position starting from xc-r and stepping unit interval along x-direction, compute corresponding y positions as

$$y = yc \pm \frac{r_y}{r_x}\sqrt{r_x^{\,2} - (x-xc)^2}$$

3. Plot the point (x , y).

4. Repeat step 2 to 3 until `x >= xc+xr`

Computation of ellipse using polar co-ordinates:

Using the polar co-ordinates for ellipse, we can compute the (x,y) position of the ellipse boundary using the following parametric equations

$x = xc + r\cos\theta$
$y = yc + r\sin\theta$

The algorithm based on these parametric equation on polar co-ordinates can be summarized as below.

1. Input center of ellipse ***(xc,yc)*** and radii xr and yr.
2. Starting θ from angle 0^o step minimum increments and compute boundary point of ellipse as

$$x = xc + r\cos\theta$$
$$y = yc + r\sin\theta$$

3. Plot the point at position (round(x), round(y))
4. Repeat until θ is greater or equal to 360^o.

The *drawellipse(int,int,int,int)* function can be written as

```
void drawellipse(int xc,int yc,int rx,int ry)
```

```
{
      int x,y;
      float theta;
      const float PI=3.14;
      for(theta=0.0;theta<=360;theta+=1)
      {
            x= xc+rx*cos(theta*PI/180.0);
            y= yc+ry*sin(theta*PI/180.0);
            putpixel(x,y,1);

      }
}
```

The methods of drawing ellipses explained above are not efficient. The method based on direct equation of ellipse must perform the square and square root operations due to which there may be floating point number computation which cause rounding off to plot the pixels. Also the square root causes the domain error. Due to the changing slope of curve along the path of ellipse, there may be un-uniform separation of pixel when slope changes. To eliminate this problem , extra computation is needed. Although , the method based on polar co-ordinate parametric equation gives the uniform spacing of pixel due to uniform increment of angle but it also take extra computation to evaluate the trigonometric functions. So these algorithms are not efficient to construct the ellipse. We have another algorithm called mid- point ellipse algorithm similar to mid-point circle algorithm which is efficient algorithm for computing ellipse.

Mid-Point Ellipse Algorithm:

The mid-point ellipse algorithm decides which point near the boundary (i.e. path of the ellipse) is closer to the actual ellipse path described by the ellipse equation. That point is taken as next point .

It is applied to the first quadrant in two parts as in figure. Region 1 and Region 2.We process by taking unit steps in x-coordinates direction and finding the closest value for y for each x-steps in region 1.

In first quadrant at region 1, we start at position *(0,ry)* and incrementing x and calculating y closer to the path along clockwise direction . When slope becomes -1 then shift unit step in x to y and compute corresponding x closest to ellipse path at Region 2 in same direction.

Alternatively , we can start at position *(rx,0)* and select point in counterclockwise order shifting unit steps in y to unit step in x when slope becomes greater than -1.

Here, to implement mid-point ellipse algorithm, we take start position at *(0,ry)* and step along the ellipse path in clockwise position throughout the first quadrant.

We define ellipse function center at origin i.e, *(xc,yc)= (0,0)* as

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

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

So $f_{ellipse}$ function serves as decision parameter in ellipse algorithm at each sampling position. We select the next pixel position according to the sign of decision parameter.

Starting at $(0,r_y)$, we take unit step in x-direction until we reach the boundary between the region 1 and region 2. Then we switch unit steps in y over the remainder of the curve in first quadrant. At each step, we need to test the slope of curve. The slope of curve is calculated as;

$$\frac{dy}{dx} = -\frac{2r_y{}^2 x}{2r_x{}^2 y}$$

At the boundary between region 1 and region 2,

$\frac{dy}{dx} = -1$ and $2r_y{}^2 = 2r_x{}^2$ Therefore , we move out of region 1 when $2r_y{}^2 x \geq 2r_x{}^2$

Assuming the position (x_k, y_k) is filled, we move x_{k+1} to determine next pixel. The corresponding y value for x_{k+1} position will be either y_k or $y_k - 1$ depending upon the sign of decision parameter. So the decision parameter for region 1 is tested at mid point of $(x_k + 1, y_k)$ and $(x_k + 1, y_k - 1)$ i.e.

$$p_{1k} = f_{ellipse}(x_{k+1}, y_k - \frac{1}{2})$$

$$\text{or } p_{1k} = r_y{}^2 (x_{k+1})^2 + r_x{}^2 (y_k - \frac{1}{2})^2 - r_x{}^2 r_y{}^2$$

$$\text{or } p_{1k} = r_y{}^2 (x_{k+1})^2 + r_x{}^2 y_k{}^2 - r_x{}^2 y_k + \frac{r_x^2}{4} - r_x{}^2 r_y{}^2 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

if $p_{1k} < 0$, the mid point lies inside boundary, so next point to plot is $(x_k + 1, y_k)$ otherwise, next point to plot will be $(x_k + 1, y_k - 1)$

The successive decision parameter is computed as

$$p_{1k+1} = f_{ellipse}(x_{k+1} + 1, y_{k+1} - \frac{1}{2})$$

$$= r_y{}^2 (x_{k+1} + 1)^2 + r_x{}^2 (y_{k+1} - \frac{1}{2})^2 - r_x{}^2 r_y{}^2$$

$$\text{or } p_{1k+1} = r_y{}^2 (x_{k+1}^2 + 2x_{k+1} + 1) + r_x{}^2 (y_{k+1}^2 - y_{k+1} + \frac{1}{4}) - r_x{}^2 r_y{}^2$$

$$\text{or } p_{1k+1} = r_y{}^2 x_{k+1}^2 + 2r_y{}^2 x_{k+1} + r_y^2 + r_x{}^2 y_{k+1}^2 - r_x{}^2 y_{k+1} + \frac{r_x{}^2}{4} - r_x{}^2 r_y{}^2 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

Subtracting (2) - (1)

$$p_{1k+1} - p_{1k} = 2r_y{}^2 x_{k+1} + r_y^2 + r_x{}^2 (y_{k+1}^2 - y_k^2) - r_x^2 (y_{k+1} - y_k)$$

if $p_{1k} < 0$, $y_{k+1} = y_k$ then,

$$\therefore p_{1k+1} = p_{1k} + 2r_y{}^2 x_{k+1} + r_y{}^2$$

Otherwise $y_{k+1} = y_k - 1$ then we get,

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

At the initial position, $(0,r_y)$ $2r_y{}^2 x = 0$ and $2r_x{}^2 y = 2r_x^2 r_y$

In region 1, initial decision parameter is obtained by evaluating ellipse function at $(0,r_y)$ as

$$p_{10} = f_{ellipse}(1, r_y - \frac{1}{2})$$

or $$p_{10} = f_{ellipse}(1, r_y - \frac{1}{2})$$

$$= r_y^2 - r_x^2 r_y + \frac{1}{4} r_x{}^2$$

Similarly, over the region 2, the decision parameter is tested at mid point of $(x_k, y_k - 1)$ and $(x_k + 1, y_k - 1)$ i.e.

$$p_{2k} = f_{ellipse}(x_k + \frac{1}{2}, y_k - 1)$$

$$= r_y{}^2 (x_k + \frac{1}{2})^2 + r_x{}^2 (y_k - 1)^2 - r_x{}^2 r_y{}^2$$

$$\therefore p_{2k} = r_y{}^2 x_k{}^2 + r_y^2 x_k + \frac{r_y^2}{4} + r_x{}^2 (y_k - 1)^2 - r_x{}^2 r_y{}^2 \ldots\ldots\ldots\ldots\ldots\ldots(3)$$

if $p_{2k} > 0,$ the mid point lies outside the boundary, so next point to plot is $(x_k, y_k - 1)$ otherwise, next point to plot will be $(x_k + 1, y_k - 1)$

The successive decision parameter is computed as evaluating ellipse function at mid point of

$p_{2k+1} = f_{ellipse}(x_{k+1} + \frac{1}{2}, y_{k+1} - 1)$ with $y_{k+1} = y_k - 1$

$$p_{2k+1} = r_y^2 (x_{k+1} + \frac{1}{2})^2 + r_x^2 [(y_k - 1) - 1]^2 - r_x^2 r_y^2$$

or $$p_{2k+1} = r_y^2 x_{k+1}^2 + r_y^2 x_{k+1} + \frac{r_y^2}{4} + r_x^2 (y_k - 1)^2 - 2r_x^2 (y_k - 1) + r_x^2 - r_x^2 r_y^2 \ldots\ldots\ldots(4)$$

subtracting (4)-(3)

$$p_{2k+1} - p_{2k} = r_y^2 (x_{k+1}^2 - x_k^2) + r_y^2 (x_{k+1} - x_k) - 2r_x^2 (y_k - 1) + r_x^2$$

or $$p_{2k+1} = p_{2k} + r_y^2 (x_{k+1}^2 - x_k^2) + r_y^2 (x_{k+1} - x_k) - 2r_x^2 (y_k - 1) + r_x^2$$

if $p_{2k} > 0,\ x_{k+1} = x_k$ then

$$p_{2k+1} = p_{2k} - 2r_x^2(y_k - 1) + r_x^2$$

otherwise $x_{k+1} = x_k + 1$ then

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

or $p_{2k+1} = p_{2k} + r_y^2(2x_k + 1) + r_y^2 - 2r_x^2(y_k - 1) + r_x^2$

or $p_{2k+1} = p_{2k} + r_y^2(2x_k + 2) - 2r_x^2(y_k - 1) + r_x^2$

or $p_{2k+1} = p_{2k} + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_x^2$ where $x_{k+1} = x_k + 1$ and $y_{k+1} = y_k - 1$

The initial position for region 2 is taken as last position selected in region 1 say which is (x_0, y_0) then initial decision parameter in region 2 is obtained by evaluating ellipse function at mid point of $(x_0, y_0\text{-}1)$ and $(x_0 + 1, y_0\text{-}1)$ as

$$p_{20} = f_{ellipse}(x_0 + \frac{1}{2}, y_0 - 1)$$

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

Now the mid point ellipse algorithm is summarized as;

1. Input center (xc,yc) and r_x and r_y for the ellipse and obtain the first point as

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

2. Calculate initial decision parameter value in Region 1 as

   $$P_{10} = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2$$

3. At each x_k position, in Region 1, starting at k = 0 , compute

   $x_{k+1} = x_k + 1$

   If $p_{1k} < 0,$ then the next point to plot is

   $p_{1k+1} = p_{1k} + 2r_y^2 x_{k+1} + r_y^2$

   $y_{k+1} = y_k$

   Otherwise next point to plot is

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

   $p_{1k+1} = p_{1k} + 2r_y^2 x_{k+1} + r_y^2 - 2r_x^2 y_{k+1}$ with $x_{k+1} = x_k + 1$ and $y_{k+1} = y_k - 1$

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

$$p_{20} = r_y^2(x_0 + \frac{1}{2})^2 + r_x^2(y_0 - 1)^2 - r_x^2 r_y^2$$

5. At each y_k position in Region 2 starting at k = 0, perform computation

$y_{k+1} = y-1;$
if $p_{2k} > 0,$ then
$x_{k+1} = x_k$
$p_{2k+1} = p_{2k} - 2r_x^2(y_k - 1) + r_x^2$
Otherwise
$x_{k+1} = x_k + 1$
$p_{2k+1} = p_{2k} + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_x^2$ where $x_{k+1} = x_k + 1$ and $y_{k+1} = y_k - 1$

6. Determine the symmetry points in other 3 quadrants.

7. Move each calculated point (x_k, y_k) on to the centered (xc,yc) ellipse path as

$x_k = x_k + xc;$

$y_k = y_k + yc$

8. Repeat the process for region 1 until $2r_y^2 x_k \geq 2r_x^2 y_k$ and region until (x_k, y_k) $=(r_x,0)$

Two Dimensional Geometric Transformations

In computer graphics, transformations of 2D objects are essential to many graphics applications. The transformations are used directly by application programs and within many graphics subroutines in application programs. Many applications use the geometric transformations to change the position , orientation, and size or shape of the objects in drawing. Rotation, Translation and scaling are three major transformations that are extensively used by all most all graphical packages or graphical subroutines in applications. Other than these , reflection and shearing transformations are also used by some graphical packages.

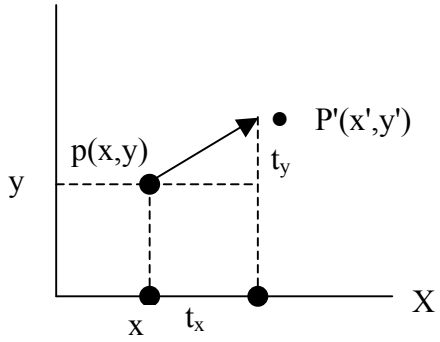
2D Translation

A translation is applied to an object by re-positioning it along a straight line path from one co-ordinate location to another. We translate a two-dimensional point by adding translation distances, t_x, t_y to the respective co-ordinate values of original co-ordinate position (x, y) to move the point to a new position (x', y') as:

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

Y

The translation distance pair (t_x,t_y) is known as translation vector or shift vector. We can express translation equations as matrix representations as



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

$$\therefore P' = P + T$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

Some times matrix transformation are represented by co-ordinate rows vector instead of column vectors as.

$$P = (x, y) \qquad T = (t_x, t_y) \qquad P' = P + T.$$

For translation of any object in Cartesian plane, we transform the distinct co-ordinates by the translation vector and re-draw image at the new transformed locatio

2D Rotation

The 2D rotation is applied to re-position the object along a circular path in XY-plane. To generate rotation, we specify a rotation angle θ , through which the co-ordinates are to be rotated. Rotation can be made by angle θ either clockwise or anticlockwise direction. Besides the angle of rotation θ, there should be a pivot point through which the object is to be rotated. The positive θ rotates object in anti-clockwise direction and the negative value of θ rotates the object in clock-wise direction.

A line perpendicular to rotating plane and passing through pivot point is called axis of rotation.

Let P(x,y) is a point in XY-plane which is to be rotated with angle θ. Also let OP = r (As in figure below) is constant distance from origin. Let r makes angle ϕ with positive X – direction as shown in figure.

When OP is rotated through angle θ Taking origin as pivot point for rotation, then OP' makes angle $\theta+\phi$ with X-axis.

Now ,

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

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

But, $r\cos\phi = x, r\sin\phi = y$ therefore we get

$x' = x\cos\theta - y\sin\theta$----------1

$y' = x\sin\theta + y\cos\theta$----------2 which are equation for rotation of (x,y) with angle θ and taking pivot as origin.

Rotation from any arbitrary pivot point (x_r, y_r)

- Let $Q(x_r,y_r)$ is pivot point for rotation.
- P(x,y) is co-ordinate of point to be rotated by angle θ.
- Let ϕ is the angle made by QP with X-direction. θ.

Then angle made by QP' with X-direction is $\theta+\phi$

Hence,

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

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

or $x'-x_r = r\cos\phi\cos\theta - r\sin\phi\sin\theta$

since $r\cos\phi = x - x_r, rsiin\phi = y - y_r$

$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta \ldots\ldots\ldots\ldots(1)$$

Similarly,

$\sin(\phi+\theta) = (y'-y_r)/r$

or $r\sin(\phi+\theta) = (y'-y_r)$

or $y'-y_r = r\sin\phi\cos\theta + r\cos\phi\sin\theta$

since $r\cos\phi = x - x_r, rsiin\phi = y - y_r$

$$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta \ldots\ldots\ldots\ldots(2)$$

These equations (1) and (2) are the equations for rotation of a point (x,y) with angle θ taking pivot point (x_r,y_r).

The rotation about pivot point (x_r,y_r) can be achieve by sequence of translation, rotation about origin and reverse translation.

- Translate the point (x_r,y_r) and P(x,y) by translation vector $(-x_r,-y_r)$ which translates the pivot to origin and P(x,y) to $(x-x_r,y-y_r)$.
- Now apply the rotation equations when pivot is at origin to rotate the translated point $(x-x_r,y-y_r)$ as:

$$x_1 = (x - x_r)\cos\theta - (y - y_r)\sin\theta$$
$$y_1 = (x - x_r)\sin\theta + (y - y_r)\cos\theta$$

- Re- translate the rotated point (x_1, y_1) with translation vector (x_r,y_r) which is reverse translation to original translation. Finally we get the equation after successive transformation as

$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta \ldots\ldots\ldots\ldots(1)$$

$$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta \ldots\ldots\ldots\ldots(2)$$ Which are actually the equations for rotation of (x,y) from the pivot point (x_r,y_r)

2D Scaling:
A scaling transformation alters the size of the object. This operation can be carried out for polygon by multiplying the co-ordinate values (x,y) of each vertex by scaling factor s_x and s_y to produce transformed co-ordinates (x', y').

i.e. $x' = x.s_x$ and $y' = y.s_y$

Scaling factor s_x scales object in x- direction and s_y scales in y- direction. If the scaling factor is less then 1, the size of object is decreased and if it is greater than 1 the size of object is increased. The scaling factor = 1 for both direction does not change the size of the object. If both scaling factors have same value then the scaling is known as uniform scaling. If the value of s_x and s_y are different, then the scaling is known as differential scaling. The differential scaling is mostly used in the graphical package to change the shape of the object.

The matrix equation for scaling is:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} . \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{i.e. } P' = S.P$$

Homogeneous co-ordinate representation of 2D Transformation

- The homogeneous co-ordinate system provide a uniform frame-work for handling different geometric transformations, simply as multiplication of matrices.
- Its extension to 3D is straight forward which also helps to produce prespective projections by use of matrix multiplication. We simply add a third co-ordinate to 2D point i.e.
  (x,y) =(x_h,y_h,h) where $x = x_h/h$, $y = y_h/h$ where h is 1 usually for 2D case.
- By using this homogeneous co-ordinate system a 2D point would be (x,y,1). The matrix formulation for 2D translation for $T(t_x,t_y)$ is :

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$ By which we can get

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

For Rotation: R(θ) about origin the homogeneous matrix equation will be

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

$x' = x\cos\theta - y\sin\theta$----------1

$y' = x\sin\theta + y\cos\theta$----------2 which are equation for rotation of (x,y) with angle θ and taking pivot as origin.

Rotation about an arbitrary fixed pivot point (x_r,y_r)

For a fixed pivot point rotation, we can apply composite transformation as

1. Translate fixed point (x_r,y_r) to the co-ordinate origin by T(-x_r,-y_r).
2. Rotate with angle θ → R(θ) .
3. Translate back to original position by T(x_r,y_r)

This composite transformation is represented as:

P' = T(x_r,y_r). R(θ). T(-x_r,-y_r).P

Which can be represented in matrix equation using homogeneous co-ordinate system as

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

Expanding this equation we get

$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta$.....(1)

$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta$.....(2) Which are actually the equations for rotation of (x,y) from the pivot point (x_r,y_r)

Scaling with scaling factors (s_x, s_y)

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$ which exactly gives the equations

x' = x.s_x and y' = y.s_y

Fixed point scaling:

Objects transformed with standard scaling equation are scaled as well as re-positioned. Scaling factor with value less then 1 moves object closer to the origin where as value of scaling factor greater than 1 moves object away from origin.

- To control the location of scaled object we can choose the position called fixed point . Let co-ordinates of fixed point = (x_f,y_f). It can be any point on the object.
- A polygon is then scaled relative to (x_f,y_f) by scaling the distance from each vertex to the fixed point.

. For a vertex with co-ordinate (x,y), the scaled co-ordinates (x',y') are calculated as:

$$x' = x_f + (x-x_f)s_x$$
$$y' = y_f + (y-y_f)s_y \quad \text{or equivalently,}$$

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

Where $(1-s_x)x_f$ *and* $(1-s_y)y_f$ are constant for all points in object.

To represent fixed point scaling using matrix equations in homogeneous co-ordinate system, we can use composite transformation as in fixed point rotation.

1. Translate object to the origin so that (x_f, y_f) lies at origin by T(-x_f, -y_f).
2. Scale the object with (s_x, s_y)
3. Re- translate the object back to its original position so that fixed point (x_f, y_f) moves to its original position. In this case translation vector is T(x_f, y_f).

$$\therefore P' = \left[T(x_f, y_f).S(s_x, s_y).T(-x_f, -y_f)\right]P$$

The homogeneous matrix equation for fixed point scaling is

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

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

Directive scaling

Standard and fixed point scaling scales object along x and y axis only. Directive scaling scales the object in any direction.

Let S_1 and S_2 are given directions for scaling at angle Θ from co-ordinate axes as in figure below

1. First perform the rotation so that directions S_1 and S_2 concide with x and y – axes.
2. Then the scaling transformation is applied to scale the object by given scaling factors (s_1,s_2).
3. Re- apply the rotation in opposite direction to return to their original orientation.

For any point P in object, the directive scaling position P' is given by following composite transformation.

$P' = R^{-1}(\theta).S(s_1, s_2).R(\theta).P$ for which the homogeneous matrix equation is

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Reflection:

A reflection is a transformation that produce a mirror image of an object. In 2D-transformation, reflection is generated relative to an axis of reflection. The reflection of an object to an relative axis of reflection , is same as 180^o rotation about the reflection axis.

1. Reflection about X-axis: The line representing x-axis is y = 0. The reflection of a point P(x,y) on x-axis , changes the y-coordinate sign. i.e. Reflection about x-axis , the reflected position of P(x,y) will be P'(x,-y). Hence, reflection in x-axis is accomplished with transformation equation

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

gives the reflection of a point.

To reflect an 2D object, reflect each distinct points of object by above equation, then joining the points with straight line, redraws the image for reflected image.

2. Reflection about Y-axis: The line representing y-axis is x = 0. The reflection of a point P(x,y) on y-axis changes the sign of x-coordinate. i.e. P(x,y) changes to P'(-x,y).

   Hence reflection of a point on y-axis is obtained by following matrix equation

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

   For any 2D object, reflect each point and re-draw image joining the reflected images of all distinct points.

3. Reflection on an arbitrary axis: The reflection on any arbitrary axis of reflection can be achieved by sequence of rotation and co-ordinate axes reflection matrices.

   - First, rotate the arbitrary axis of reflection so that the axis of reflection and one of the co-ordinate axis concide.
   - Reflect the image on the co-ordinate axis to which the axis of reflection concides.
   - Rotate the axis of reflection back to its original position.

For example consider a line y = x for axis of reflection, the possible sequence of transformation for reflection of 2D object are

4. Reflection about origin: The reflection on the line perpendicular to xy-plane and passing through flips x and y co-ordinates both. So sign of x and y co-ordinate value changes. The equivalent matrix equation for the point is:

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

Shearing:

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 shear.

X-direction Shear: An X-direction shear relative to x-axis is produced with transformation matrix equation.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$ which transforms $x' = x + sh_x$ and y' = y

A unit square transformed to a parallelogram using x-direction shear with $sh_x = 2$.

Y-direction shear: An y-direction shear relative to y-axis is produced by following transformation equations.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$ which transforms $x' = x$ and y' = y $y' = x.sh_y + y$

```
/*
      Example : Program for translation of 2D object using
      homogeneous  co-ordinates representation of object for its
      vertices. Which translates the rectangle by given translation
      vector.
*/
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<graphics.h>
void matrixmultiply(int T[3][3],int V[3][5],int r[3][5]);
void main()
{

   int gdriver, gmode, errorcode;
   int i,j,k;
   int vertex[3][5]={{100,100,200,200,100},
            {100,200,200,100,100},
            {1,1,1,1,1},
            };
   int translate[3][3] ={{1,0,100},{0,1,200},{0,0,1}};
   int result[3][5];
   gdriver=DETECT;/* request auto detection */
      /* initialize graphics and local variables */
   initgraph(&gdriver, &gmode, "\\tc\\bgi");
      /* read result of initialization */
   errorcode = graphresult();
      /* an error occurred */
   if (errorcode != grOk)
   {
      printf("Graphics error: %s\n", grapherrormsg(errorcode));
      printf("Press any key ....");
      getch();
            /* terminate with an error code */
      exit(1);
   }

      setbkcolor(2);
      for(i=0;i<4;i++)
      {
            setcolor(BLUE);

      line(vertex[0][i],vertex[1][i],vertex[0][i+1],vertex[1][i+1]);

      }

      printf("Press any key for the translated line.....\n");
      getch();
      matrixmultiply(translate,vertex,result);
      for(i=0;i<4;i++)
      {
            setcolor(YELLOW);

      line(result[0][i],result[1][i],result[0][i+1],result[1][i+1]);

      }
      getch();
      closegraph();
}
void matrixmultiply(int translate[3][3],int vertex[3][5],int
result[3][5])
```

```
{
	for(int i=0;i<=3;i++)
	{
		for(int j=0;j<=5;j++)
		{
			result[i][j]=0;
			for(int k=0;k<=3;k++)
				result[i][j]+=translate[i][k]*vertex[k][j];
		}
	}
}
```

Filled Area primitives

A standard output primitive in general graphics package is solid color or patterned polygon area. Other kinds of area primitives are sometimes available, but polygons are easier to process since they have linear boundaries.

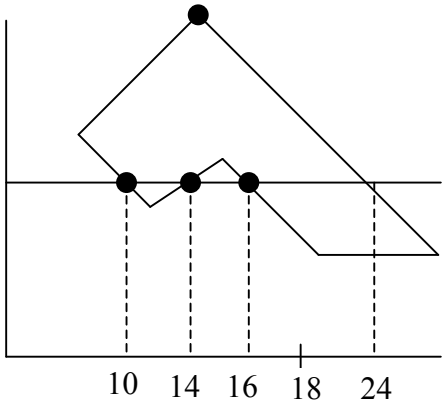
There are two basic approaches to area filling in raster systems. One way to fill an area is to determine the overlap intervals for scan lines that cresses the area. Another method for area filling is to start from a given interior position and point outward from this until a specified boundary is met.

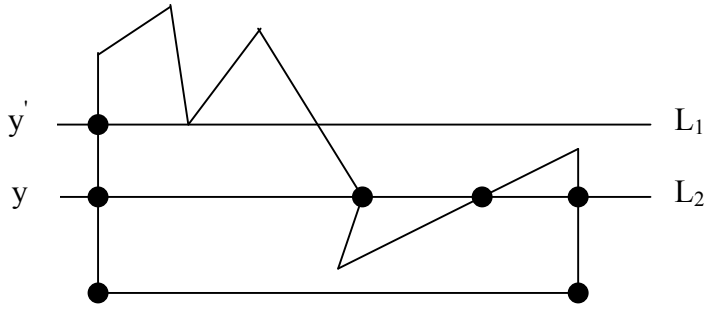
SCAN-LINE Polygon Fill Algorithm:

In scan-line polygon fill algorithm, for each scan-line crossing a polygon, it locates the intersection points of the scan line with the polygon edges. These intersection points are then sorted from left to right, and the corresponding frame-buffer positions between each intersection pair are set to the specified color. In the figure below, the four pixel intersection positions with the polygon boundaries defined two stretches of interior pixel from x=10 to x=14 and from x=16 to x=24.

some scan-line intersections at polygon vertices require extra special handling.

A scan-line passing through a vertex intersect two polygon edges at that position, adding two points to the list of intersection for the scan-line.



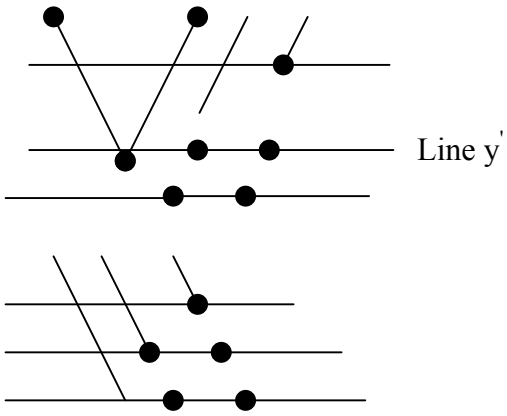


This figure shows two scan lines at position y and y' that intersect the edge points. Scan line at y intersects five polygon edges. Scan line at y' intersects 4 (even numbers) of edges though it passes through vertex.

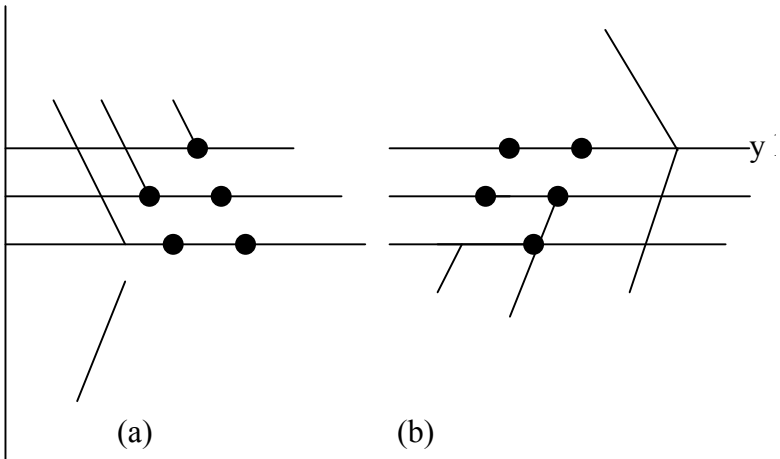
Intersection points along scan line y' correctly identify the interior pixel spans. But with scan line y, we need to do some additional processing to determine the correct interior points.

For scan line y, the two edges sharing the intersecting vertex are on opposite side of the scan-line. But for scan-line y' the two edges sharing intersecting vertex are on the same side (above) the scan line position. So the vertices those are on opposite side of scan line require extra processing.

We can identify these vertices by tracing around the polygon boundary either in clockwise or counter clockwise order and observing the relative changes in vertex y coordinates as we move from one edge to next. If the endpoint y values of two consecutive edges monotonically increases or decrease, we need to count the middle vertex as a single intersection point for any 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.



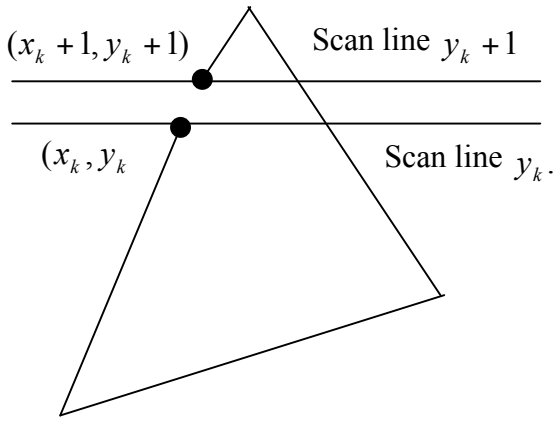
Line y', in which on tracing along edges, the y co-ordinate value is monotonically increasing so the vertex is count as the two intersecting points.



In line y, tracing along edge the y-coordinate of one edge increasing in (a) and the other edge is decreasing in (b) so this vertex is count as a single intersection point for the scan-line fill algorithm.

In successive scan lines crossing a left edge of a polygon, The slope of this polygon boundary line can be expressed in terms of scan-line intersection co-ordinates:

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



Since the change between two scan line in y co-ordinates is 1,

$$y_k + 1 - y_k = 1$$

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

$$x_k + 1 = x_k + {}^{1}\!/_{m}$$

Each successive xx intercept can thus be calculated by x values by the amount of ${}^{1}\!/_{m}$ along an edge can be accomp0lished with integer operation by recalling that the slope m is the ratio to two integers

$$m = {}^{\Delta y}\!/_{\Delta x}$$

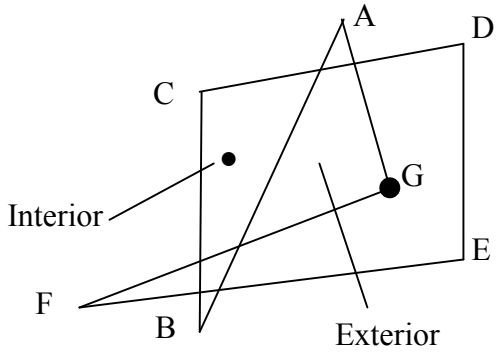
Where $\Delta x \,\&\, \Delta y$ are the differences between the edge endpoint x and y co-ordinate values. Thus incremental calculations of x intercepts along an edge for successive scan lines can be expressed as

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

Inside-Outside Test:

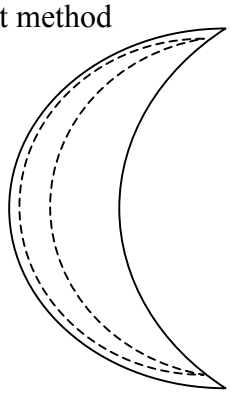
Area filling algorithms and other graphics package often need to identify interior and exterior region for a complex polygon in a plane. For ex. in figure below, it needs to identify interior and exterior region.

We apply add-even rule, also called odd-parity rule. To identify the interior or exterior point, we can draw a line from a point p to a distant point outside the co-ordinate extents of the object and count the number of intersecting edge crossed by this line. If the intersecting edge crossed by this line is odd, P is interior otherwise P is exterior.



Scan-Line Fill of Curved Boundary area

It requires more work then polygon filling, since intersection calculation involves nonlinear boundary for simple curves as circle, eclipses, performing a scan line fill is straight forward process. We only need to calculate the two scan-line intersection on opposite sides of the curve. then simply fill the horizontal spans of pixel between the boundary points on opposite side of curve. Symmetries between quadrants are used to reduce the boundary calculation we can fill generating pixel position along curve boundary using mid point method



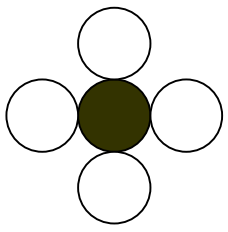
Boundary-fill Algorithm:

In Boundary filling algorithm starts at a point inside a region and paint the interior outward the boundary. If the boundary is specified in a single color, the fill algorithm proceeds outward pixel by until the boundary color is reached.

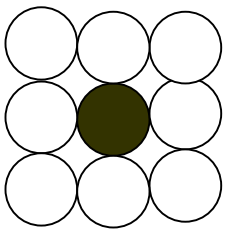
A boundary-fill procedure accepts as input the co-ordinates of an interior point (x,y), a fill color, and a boundary color. Starting from (x,y), the procedure tests neighbouring positions to determine whether they are of boundary color. If not, they are painted with the fill color, and their neighbours are tested. This process continue until all pixel up to the boundary color area have tested.

The neighbouring pixels from current pixel are proceeded by two method: 4- connected if they are adjacent horizontally and vertically.

8- connected if they adjacent horizontally, vertically and diagonally.



4- Connected



8- Connected

- Fill method that applies and tests its 4 neighbouring pixel is called 4- connected.
- Fill method that applies and tests its 8 neighbouring pixel is called 8- connected.

The out line of this algorithm is:

```
void Boundary_fill4(int x,int y,int b_color, int
fill_color)
{
     int value=get pixel (x,y);
```

```
     if (value! =b_color&&value!=fill_color)
     {
          putpixel (x,y,fill_color);
          Boundary_fill 4 (x-1,y, b_color, fill_color);
          Boundary_fill  4  (x+1,y,  b_color,  fill_color);
               Boundary_fill    4    (x,y-1,    b_color,
fill_color);
          Boundary_fill  4  (x,y+1,  b_color,  fill_color);

      }
 }
```

Boundary fill 8- connected:

```
void   Boundary-fill8(int   x,int   y,int   b_color,   int
fill_color)
{
     int current;
     current=getpixel (x,y);
     if (current !=b_color&&current!=fill_color)
     (    putpixel (x,y,fill_color);
          Boundary_fill8(x-1,y,b_color,fill_color);
          Boundary_fill8(x+1,y,b_color,fill_color);
          Boundary_fill8(x,y-1,b_color,fill_color);
          Boundary_fill8(x,y+1,b_color,fill_color);
          Boundary_fill8(x-1,y-1,b_color,fill_color);
          Boundary_fill8(x-1,y+1,b_color,fill_color);
          Boundary_fill8(x+1,y-1,b_color,fill_color);
          Boundary_fill8(x+1,y+1,b_color,fill_color);
     }
}
```

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

Flood-fill Algorithm:

Flood_fill Algorithm is applicable when we want to fill an area that is not defined within a single color boundary. If fill area is bounded with different color, we can paint that area by replacing a specified interior color instead of searching of boundary color value. This approach is called flood fill algorithm.

We start from a specified interior pixel (x,y) and reassign all pixel values that are currently set to a given interior color with desired fill_color.

Using either 4-connected or 8-connected region recursively starting from input position, The algorithm fills the area by desired color.

Algorithm:

```
void    flood_fill4(int    x,int    y,int    fill_color,int
old_color)
{
```

```
      int current;
      current=getpixel (x,y);
      if (current==old_color_
      {
           putpixel (x,y,fill_color);
           flood_fill4(x-1,y, fill_color, old_color);
           flood_fill4(x,y-1, fill_color, old_color);
           flood_fill4(x,y+1, fill_color, old_color);
           flood_fill4(x+1,y, fill_color, old_color);


      }
}
```

Similarly flood fill for 8 connected can be also defined.

We can modify procedure flood_fill4 to reduce the storage requirements of the stack by filling horizontal pixel spans.

Filled Area primitives

A standard output primitive in general graphics package is solid color or patterned polygon area. Other kinds of area primitives are sometimes available, but polygons are easier to process since they have linear boundaries.

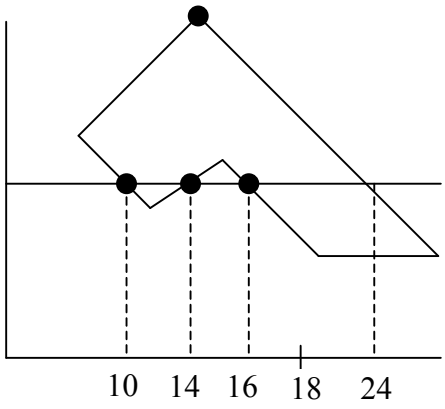
There are two basic approaches to area filling in raster systems. One way to fill an area is to determine the overlap intervals for scan lines that cresses the area. Another method for area filling is to start from a given interior position and point outward from this until a specified boundary is met.

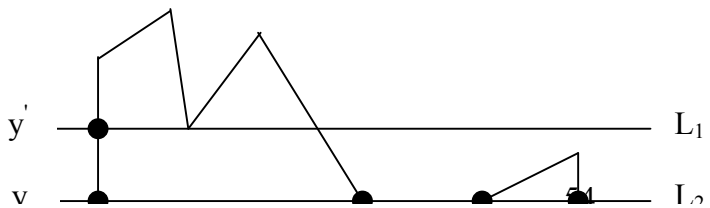
SCAN-LINE Polygon Fill Algorithm:

In scan-line polygon fill algorithm, for each scan-line crossing a polygon, it locates the intersection points of the scan line with the polygon edges. These intersection points are then sorted from left to right, and the corresponding frame-buffer positions between each intersection pair are set to the specified color. In the figure below, the four pixel intersection positions with the polygon boundaries defined two stretches of interior pixel from x=10 to x=14 and from x=16 to x=24.

some scan-line intersections at polygon vertices require extra special handling.

A scan-line passing through a vertex intersect two polygon edges at that position, adding two points to the list of intersection for the scan-line.



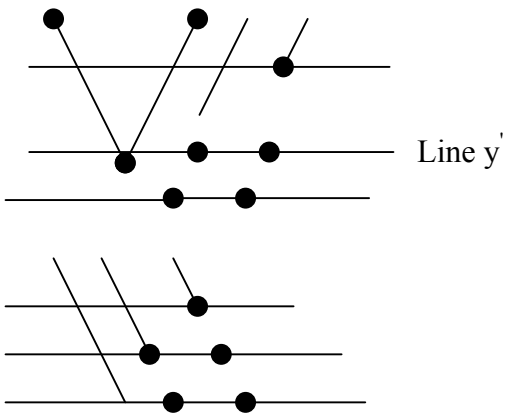


This figure shows two scan lines at position y and y' that intersect the edge points. Scan line at y intersects five polygon edges. Scan line at y' intersects 4 (even numbers) of edges though it passes through vertex.

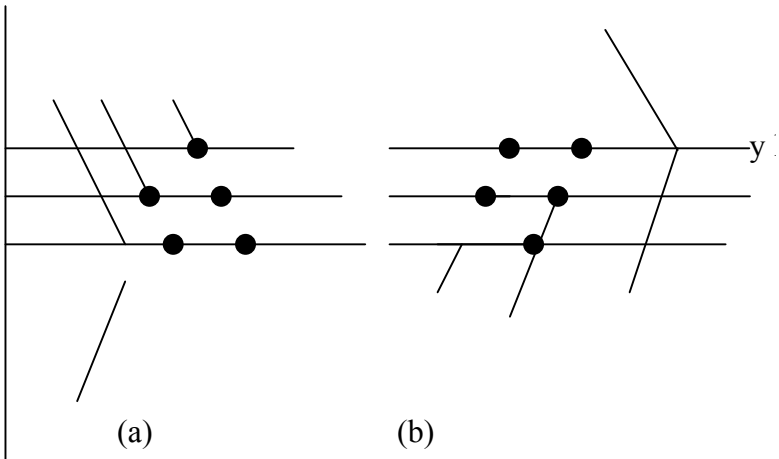
Intersection points along scan line y' correctly identify the interior pixel spans. But with scan line y, we need to do some additional processing to determine the correct interior points.

For scan line y, the two edges sharing the intersecting vertex are on opposite side of the scan-line. But for scan-line y' the two edges sharing intersecting vertex are on the same side (above) the scan line position. So the vertices those are on opposite side of scan line require extra processing.

We can identify these vertices by tracing around the polygon boundary either in clockwise or counter clockwise order and observing the relative changes in vertex y coordinates as we move from one edge to next. If the endpoint y values of two consecutive edges monotonically increases or decrease, we need to count the middle vertex as a single intersection point for any 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.



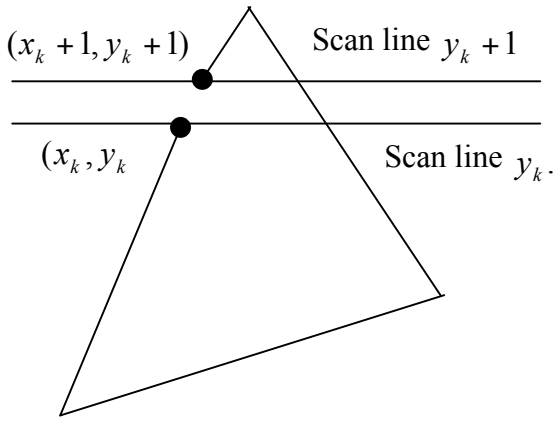
Line y', in which on tracing along edges, the y co-ordinate value is monotonically increasing so the vertex is count as the two intersecting points.



In line y, tracing along edge the y-coordinate of one edge increasing in (a) and the other edge is decreasing in (b) so this vertex is count as a single intersection point for the scan-line fill algorithm.

In successive scan lines crossing a left edge of a polygon, The slope of this polygon boundary line can be expressed in terms of scan-line intersection co-ordinates:

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



Since the change between two scan line in y co-ordinates is 1,

$$y_k + 1 - y_k = 1$$

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

$$x_k + 1 = x_k + {}^{1}\!/_{m}$$

Each successive xx intercept can thus be calculated by x values by the amount of ${}^{1}\!/_{m}$ along an edge can be accomp0lished with integer operation by recalling that the slope m is the ratio to two integers

$$m = {}^{\Delta y}\!/_{\Delta x}$$

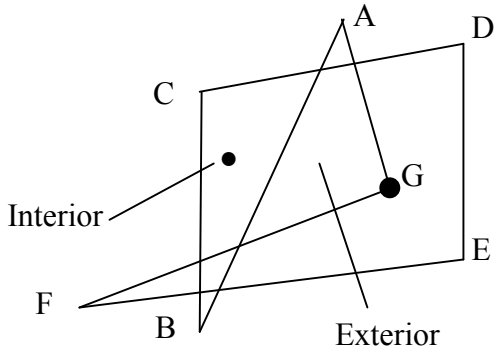
Where $\Delta x \,\&\, \Delta y$ are the differences between the edge endpoint x and y co-ordinate values. Thus incremental calculations of x intercepts along an edge for successive scan lines can be expressed as

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

Inside-Outside Test:

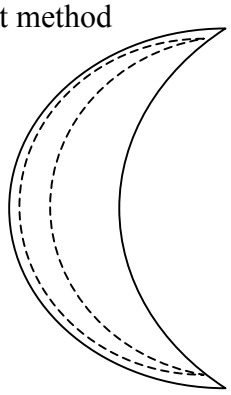
Area filling algorithms and other graphics package often need to identify interior and exterior region for a complex polygon in a plane. For ex. in figure below, it needs to identify interior and exterior region.

We apply add-even rule, also called odd-parity rule. To identify the interior or exterior point, we can draw a line from a point p to a distant point outside the co-ordinate extents of the object and count the number of intersecting edge crossed by this line. If the intersecting edge crossed by this line is odd, P is interior otherwise P is exterior.



Scan-Line Fill of Curved Boundary area

It requires more work then polygon filling, since intersection calculation involves nonlinear boundary for simple curves as circle, eclipses, performing a scan line fill is straight forward process. We only need to calculate the two scan-line intersection on opposite sides of the curve. then simply fill the horizontal spans of pixel between the boundary points on opposite side of curve. Symmetries between quadrants are used to reduce the boundary calculation we can fill generating pixel position along curve boundary using mid point method



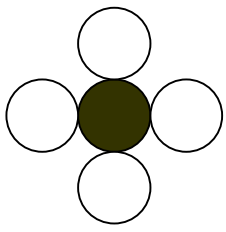
Boundary-fill Algorithm:

In Boundary filling algorithm starts at a point inside a region and paint the interior outward the boundary. If the boundary is specified in a single color, the fill algorithm proceeds outward pixel by until the boundary color is reached.

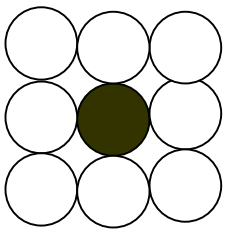
A boundary-fill procedure accepts as input the co-ordinates of an interior point (x,y), a fill color, and a boundary color. Starting from (x,y), the procedure tests neighbouring positions to determine whether they are of boundary color. If not, they are painted with the fill color, and their neighbours are tested. This process continue until all pixel up to the boundary color area have tested.

The neighbouring pixels from current pixel are proceeded by two method: 4- connected if they are adjacent horizontally and vertically.

8- connected if they adjacent horizontally, vertically and diagonally.



4- Connected



8- Connected

- Fill method that applies and tests its 4 neighbouring pixel is called 4- connected.
- Fill method that applies and tests its 8 neighbouring pixel is called 8- connected.

The out line of this algorithm is:

```
void Boundary_fill4(int x,int y,int b_color, int
fill_color)
{
     int value=get pixel (x,y);
```

```
     if (value! =b_color&&value!=fill_color)
     {
          putpixel (x,y,fill_color);
          Boundary_fill 4 (x-1,y, b_color, fill_color);
          Boundary_fill  4  (x+1,y,  b_color,  fill_color);
               Boundary_fill    4    (x,y-1,    b_color,
fill_color);
          Boundary_fill  4  (x,y+1,  b_color,  fill_color);

      }
 }
```

Boundary fill 8- connected:

```
void   Boundary-fill8(int   x,int   y,int   b_color,   int
fill_color)
{
     int current;
     current=getpixel (x,y);
     if (current !=b_color&&current!=fill_color)
     (    putpixel (x,y,fill_color);
          Boundary_fill8(x-1,y,b_color,fill_color);
          Boundary_fill8(x+1,y,b_color,fill_color);
          Boundary_fill8(x,y-1,b_color,fill_color);
          Boundary_fill8(x,y+1,b_color,fill_color);
          Boundary_fill8(x-1,y-1,b_color,fill_color);
          Boundary_fill8(x-1,y+1,b_color,fill_color);
          Boundary_fill8(x+1,y-1,b_color,fill_color);
          Boundary_fill8(x+1,y+1,b_color,fill_color);
     }
}
```

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

Flood-fill Algorithm:

Flood_fill Algorithm is applicable when we want to fill an area that is not defined within a single color boundary. If fill area is bounded with different color, we can paint that area by replacing a specified interior color instead of searching of boundary color value. This approach is called flood fill algorithm.

We start from a specified interior pixel (x,y) and reassign all pixel values that are currently set to a given interior color with desired fill_color.

Using either 4-connected or 8-connected region recursively starting from input position, The algorithm fills the area by desired color.

Algorithm:

```
void    flood_fill4(int    x,int    y,int    fill_color,int
old_color)
{
```

```
      int current;
      current=getpixel (x,y);
      if (current==old_color_
      {
           putpixel (x,y,fill_color);
           flood_fill4(x-1,y, fill_color, old_color);
           flood_fill4(x,y-1, fill_color, old_color);
           flood_fill4(x,y+1, fill_color, old_color);
           flood_fill4(x+1,y, fill_color, old_color);


      }
}
```

Similarly flood fill for 8 connected can be also defined.

We can modify procedure flood_fill4 to reduce the storage requirements of the stack by filling horizontal pixel spans.

3D Object representations

Graphical scenes can contain many different kinds of objects like trees, flowers, rocks, waters...etc. There is no one method that we can use to describe objects that will include all features of those different materials. To produce realistic display of scenes, we need to use representations that accurately model object characteristics.

- Simple Euclidean objects like polyhedrons and ellipsoids can be represented by polygon and quadric surfaces.
- Spline surface are useful for designing aircraft wings, gears and other engineering objects.
- Procedural methods and particle systems allow us to give accurate representation of clouds, clumps of grass, and other natural objects.
- Octree encodings are used to represent internal features of objects. Such as medical CT images.

Representation schemes for solid objects are often divided into two broad categories:

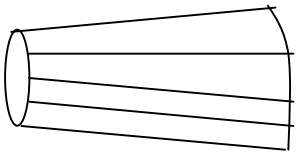
1. **Boundary representations**: describes a 3D object as a set of polygonal surfaces, separate the object interior from environment.

2. **Space-partitioning representation**: used to describe interior properties, by partitioning the spatial region, containing an object into a set of small, non overlapping, contiguous solids. e.g. 3D object as Octree representation.

Boundary Representation: Each 3D object is supposed to be formed its surface by collection of polygon facets and spline patches. Some of the boundary representation methods for 3D surface are:

1. Polygon Surfaces: It is he most common representation for 3D graphics object. In this representations, a 3D object is represented by a set of surfaces that enclose

the object interior. Many graphics system use this method. Set of polygons are stored for object description. This simplifies and speeds up the surface rendering and display of object since all surfaces can be described with linear equations.



A 3D object represented by polygons

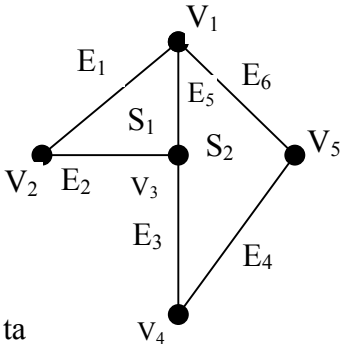
The polygon surface are common in design and solid-modeling applications, since wire frame display can be done quickly to give general indication of surface structure. Then realistic scenes are produced by interpolating shading patterns across polygon surface to illuminate.

Polygon Table: A polygon surface is specified with a set of vertex co-ordinates and associated attribute parameters. A convenient organization for storing geometric data is to create 3 lists:

- A vertex table
- An edge table
- A polygon surface table.

- Vertex table stores co-ordinates of each vertex in the object.
- The edge table stores the Edge information of each edge of polygon facets.
- The polygon surface table stores the surface information for each surface i.e. each surface is represented by edge lists of polygon.

Consider the surface contains polygonal facets as shown in figure (only two polygon are taken here)



→ S_1 and S_2 are two polygon surface that represent the boundary of some 3D object.

Foe storing geometric data, we can use following three ta

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

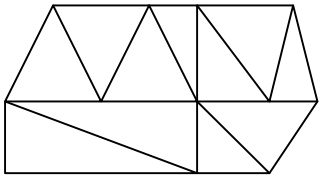
EDGE TABLE
E_1: V_1,V_2
E_2: V_2,V_3
E_3: V_3,V_4
E_4: V_4,V_5
E_5: V_1,V_3
E_6: V_5,V_1

POLYGON SURFACE 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 tables and processing them for surface rendering and visible surface determination.

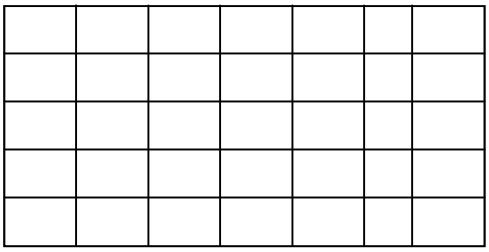
Polygon Meshes: A polygon mesh is collection of edges, vertices and polygons connected such that each edge is shared by at most two polygons. An edge connects two vertices and a polygon is a closed sequence of edges. An edge can be shared by two polygons and a vertex is shared by at least two edges.

When object surface is to be tiled, it is more convenient to specify the surface facets with a mesh function. One type of polygon mesh is triangle strip. This function produce n-2 connected triangles.



Triangular Mesh

Another similar function is the quadrilateral mesh, which generates a mesh of (n-1) by (m-1) quadrilaterals, given the co-ordinates for an $n \times m$ array of vertices.



6 by 8 vertices array , 35 element quadrilateral mesh

- If the surface of 3D object is planer, it is comfortable to represent surface with meshes.

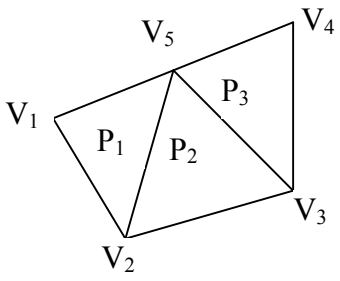
Representing polygon meshes

In explicit representation, each polygon is represented by a list of vertex co-ordinates.

$$P = ((x_1, y_1, z_1), (x_2, y_2, y_2), ..., (x_n, y_n, z_n))$$

The vertices are stored in order traveling around the polygon. There are edge between successive vertices in the list and between the last and first vertices.

- For a single polygon it is efficient but for polygon mesh it is not space efficient since no of vertices may duplicate.
- So another method is to define polygon with pointers to a vertex list. So each vertex is stored just once, in vertex list $V = \{v_1, v_2 ..... v_n\}$ A polygon is defined by list of indices (pointers) into the vertex list e.g. A polygon made up of vertices $3,5,7,10$ in vertex list be represented as $P_1 = \{3,5,7,10\}$



- Representing polygon mesh with each polygon as vertex list.
  - $P_1 = \{v_1, v_2, v_5\}$
  - $P_2 = \{v_2, v_3, v_5\}$
  - $P_3 = \{v_3, v_4, v_5\}$

  Here most of the vertices are duplicated so it is not efficient.
- Representation with indexes into a vertex list

  $V = \{v_1, v_2, v_3, v_4, v_5\} = \{(x_1, y_1, z_1), ...(x_5, y_5, z_5)\}$

  $P_1 = \{1,2,3\}$

  $P_2 = \{2,3,5\}$

  $P_3 = \{3,4,5\}$

- **<u>Defining polygons by pointers to an edge list</u>**

In this method, we have vertex list V, represent the polygon as a list of pointers not to the vertex list but to an edge list. Each edge in edge list points to the two vertices in the vertex list. Also to one or two polygon, the edge belongs.
Hence we describe polygon as

$P = (E_1, E_2, ...E_n)$

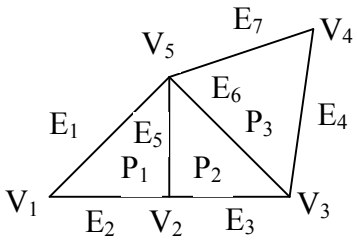
and an edge as

$E = (V_1, V_2, P_1, P_2)$

2Here if edge belongs to only one polygon, either

Then P_1 or P_2 is null.

For the mesh given below,



$V = \{v_1, v_2, v_3, v_4, v_5\} = \{(x_1, y_1, z_1), ...(x_5, y_5, z_5)\}$

$E_1 = (V_1, V_5, P_1, N)$
$E_2 = (V_1, V_2, P_1, N)$
$E_3 = (V_2, V_3, P_2, N)$
$E_4 = (V_3, V_4, P_3, N)$
$E_5 = (V_2, V_5, P_1, P_2)$
$E_6 = (V_3, V_5, P_1, P_3)$
$E_7 = (V_4, V_5, P_3, N)$
$P_1 = (E_1, E_2, E_3)$
$P_2 = (E_3, E_6, E_5)$

Here N represents Null.

$$P_3 = (E_4, E_7, E_6)$$

Polygon mesh defined with edge lists for each polygon.

3D-object representation

1. Polygon Surface: Plane Equation Method

Plane equation method is another method for representation the polygon surface for 3D object. The information about the spatial orientation of object is described by its individual surface, which is obtained by the vertex co-ordinates and the equation of each surface. The equation for a plane surface can be expressed in the form,

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

Where (x,y,z) is any point on the plane, and A,B,C,D are constants describing the spatial properties of the plane. The values of A,B,C,D can be obtained by solving a set of three plane equations using co-ordinate values of 3 non collinear points on the plane.

Let (x_1,y_1,z_1), (x_2,y_3,z_2) and (x_3,y_3,z_3) are three such points on the plane, then-

$$Ax_1 + By_1 + Cz_1 + D = 0$$
$$Ax_2 + By_2 + Cz_2 + D = 0$$
$$Ax_3 + By_3 + Cz_3 + D = 0$$

The solution of these equations can be obtained in determinant from using Cramer's rule as:-

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

For any points (x,y,z)
If $Ax + By + Cz + D \neq 0$, then (x,y,z) is not on the plane.
If $Ax + By + Cz + D < 0$, then (x,y,z) is inside the plane i. e. invisible side
If $Ax + By + Cz + D > 0$, then (x,y,z) is lies out side the surface.

2. Wireframe Representation:

In this method 3D objects is represented as a list of straight lines, each of which is represented by its two end points (x1,y1,z1) and (x2,y2,z2). This method only shows the skeletal structure of the objects.

It is simple and can see through the object and fast method. But independent line data structure is very inefficient i.e don't know what is connected to what. In this method the scenes represented are not realistic.

3. Blobby Objects:

Some objects don't maintain a fixed shape but change their surface characteristics in certain motions or when proximity to another objects e.g molecular structures, water droplets, other liquid effects, melting objects, muscle shaped in human body etc. These objects can be described as exhibiting "blobbiness" and are referred as blobby objects.

Several models have been developed for representing blobby objects as distribution functions over a region of space. One way is to use Gaussian density function or bumps. A surface function is defined as:

$f(x,y,z) = \sum_k b_{k0} e^{-a_k r_k^2} - T = 0$

Where $r_k = \sqrt{x_k^2 + y_k^2 + z_k^2}$, T = Threshold and a and b are used to adjust amount of blobbness.

Other method for generating for generating blobby objects uses quadratic density function as:

$$f(x) = \begin{cases} b(1 - \frac{3r^3}{d^2}) \\ \frac{3}{2} b(1 - \frac{r}{d})^2 \\ 0 \end{cases}$$

Advantages

- Can represent organic, blooby or liquid line structures
- Suitable for modeling natural phenomena like water, human body
- Surface properties cam be easily derived from mathematical equations.

Disadvantages:

- Requires expensive computation
- Requires special rendering engine
- Not supported by most graphics hardware

4. Spline Representation

A Spline is a flexible strips used to produce smooth curve through a designated set of points. A curve drawn with these set of points is spline curve. Spline curves are used to model 3D object surface shape smoothly.

Mathematically, spline are described as pice-wise cubic polynomial functions. In computer graphics, a spline surface can be described with two set of orthogonal spline curves. Spline is used in graphics application to design and digitalize drawings for storage in computer and to specify animation path. Typical CAD application for spline includes the design of automobile bodies, aircraft and spacecraft surface etc.

Interpolation and approximation spline

- Given the set of control points, the curve is said to interpolate the control point if it passes through each points.
- If the curve is fitted from the given control points such that it follows the path of control point without necessarily passing through the set of point, then it is said to approximate the set of control point.

Cubic spline:

It is most often used to set up path for object motions o tot provide a representation for an existing object or drawing. To design surface of 3D object any spline curve can be represented by piece-wise cubic spline.

Cubic polynomial offers a reasonable compromise between flexibility and speed of computation. Cubic spline requires less calculations with comparison to higher order polynomials and require less memory. Compared to lower order polynomial cubic spline are more flexible for modeling arbitrary curve shape.

Given a set of control points, cubic interpolation spines are obtained by fitting the input points with a picewise cubic polynomial curve that passes through every control points.

Suppose we have n+1 control points specified with co-ordinates.

$p_k = (x_k, y_k, z_k), \quad k = 0,1,2,3\ldots\ldots\ldots\ldots\ldots\ldots n$

A cubic interpolation fit of those points is

We can describe the parametric cubic polynomial that is to be fitted between each pair of control points with the following set of parametric equations.

$x(u) = a_x u^3 + b_x u^2 + c_x u + d_x$

$y(u) = a_y u^3 + b_y u^2 + c_y u + d_y \qquad (0 \le u \le 1)$
$z(u) = a_z u^3 + b_z u^2 + c_z u + d_z$

There are three equivalent methods for specifying a particular spline representation.

1. Set of boundary conditions.
   For the parametric cubic polynomial for the x-coordinate along the path of spline section
   $x(u) = a_x U^3 + b_x U^2 + c_x U + d \qquad 0 \le U \le 1$

   Boundary condition for this curve be the set on the end point coordinate x(0) and x(1) and in the first derivatives at end points x'(0) and x'(1). These four boundary condition are sufficient to determine the four coefficient a_x, b_x, c_x, d_x.

2. From the boundary condition, we can obtain the characterizing matrix for spline. Then the parametric equation can be written as-

   $$x(u) = [\ u3\ \ u2\ \ u\ \ 1] \begin{bmatrix} a_x \\ b_x \\ c_x \\ d_x \end{bmatrix}$$

3. Blending function that determines how specified geometric constraints on the curve are combined to calculate position along the curve path.

   $$x(u) = \sum_{k=0}^{3} g_k . BF_k(u)$$

   Where g_k are the geometric constraint parameters such as control points co-ordinate and slope of the curve at control point. $BF_k(u)$ are the polynomial Blending functions.

Bezier curve and surface

This is spline approximation method, developed by the French Engineer Pierre Bezier for use in the design of automobile body. Beizer spline has a number of properties that make them highly useful and convenient for curve and surface design. They are easy to implement. For this reason, Bezier spline is widely available in various CAD systems.

In general Bezier curve can be fitted to any number of control points. The number of control points to be approximated and their relative position determine the degree of Bezier polynomial. The Bezier curve can be specified with boundary condition, with characterizing matrix or blending functions. But for general blending function specification is most convenient.

Suppose we have n+1 control points: $p_k(x_k,y_k,z_k)$, k = 0,1,2,3,4,…….n. These co-ordinate points can be blended to produce the following position vector p(u) which describes path of an approximating Bezier polynomial function p_0 and p_n.

$$p(u) = \sum_{k=0}^{n} p_k . BEZ_{k,n}(u), \qquad 0 \le u \le 1 \text{ ---------- } 1$$

The Bezier belending function $BEZ_{k,n}(u)$ are the Bernstein polynomial as,

$$BEZ_{k,n}(u) = c(n,k)u_k(1-u)^{n-k}$$

The vector equation (1) represents a set of three parametric equation for individual curve conditions.

i.e.
$$x(u) = \sum_{k=0}^{n} x_k . BEZ_{k,n}(u)$$

$$y(u) = \sum_{k=0}^{n} y_k . BEZ_{k,n}(u)$$

$$z(u) = \sum_{k=0}^{n} z_k . BEZ_{k,n}(u)$$

Bezier curve is a polynomial of degree one less than control points i.e. 3 points generate parabola, 4 points a cubic curve and so n.

Properties of Bezier Curve:

1. It always passs through initial and final control points. i.e $p(0) = p_0$ and $p(1)=p_n$.

2. Values of the parametric first derivatives of a Bezier curve at the end points can be calculated from control points as-
   $p'(0) = -np_0 + np_1$
   $p'(1) = -np_{n-1} + np_n$

3. The slope at the beginning of the curve is along the line joining the first two points and slope at the end of curve is along the line joining last two points.

4. Parametric second derivative at a Bezier curve at end points are-
   $p''(0) = n(n-1)[(p_2-p_1) - (p_1-p_0)]$
   $p''(1) = n(n-1)[(p_{n-2}-p_{n-1}) - (p_{n-1}-p_n)]$

Quadric Surface

Quadric Surface is one of the frequently used 3D objects surface representation. The quadric surface can be represented by a second degree polynomial. This includes:

1. Sphere: For the set of surface points (x,y,z) the spherical surface is represented as:
   $x^2+y^2+z^2 = r^2$, with radius r and centered at co-ordinate origin.

2. Ellipsoid: $\frac{x^2}{a^2}+\frac{y^2}{b^2}+\frac{z^2}{c^2}=1$, where (x,y,z) is the surface points and a,b,c are the radii on X,Y and Z directions respectively.

3. Elliptic parboiled: $\frac{x^2}{a^2}+\frac{y^2}{b^2}=z$

4. Hyperbolic parboiled : $\frac{x^2}{a^2}-\frac{y^2}{b^2}=z$

5. Elliptic cone : $\frac{x^2}{a^2}+\frac{y^2}{b^2}-\frac{z^2}{c^2}=0$

6. Hyperboloid of one sheet: $\frac{x^2}{a^2}+\frac{y^2}{b^2}-\frac{z^2}{c^2}=1$

7. Hyperboloid of two sheet: $\frac{x^2}{a^2}-\frac{y^2}{b^2}-\frac{z^2}{c^2}=1$

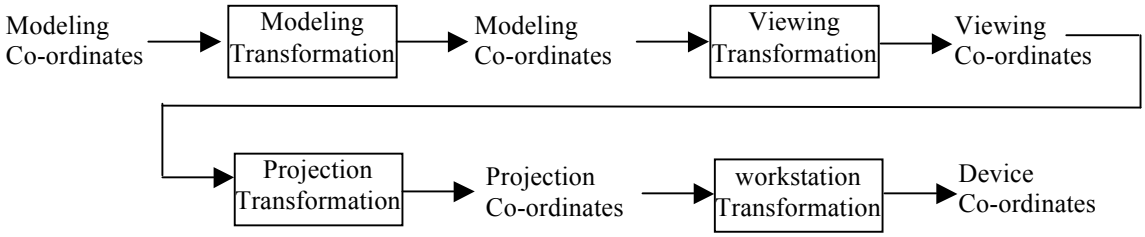
Octree Representation: (Solid-object representation)

This is the space-partitioning method for 3D solid object representation. This is hierarchical tree structures (octree) used to represent solid object in some graphical system. Medical imaging and other applications that require displays of object cross section commonly use this method. E.g.: CT-scan

It provides a convenient representation for storing information about object interiors.

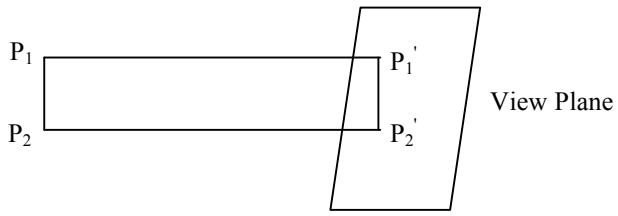
An octree encoding scheme divides region of 3D space into ocatants and stores 8 data elements in each node of the tree. Individual elements are called volume element or vexels. When all voxels in an octant are of same type, this type value is stored in corresponding data elements. Any heterogeneous octants are subdivided into octants again.

3D Viewing pipeline:
The steps for computer generation of a view of 3D scene are analogous to the process of taking photograph by a camera. For a snapshot, we need to position the camera at a particular point in space and then need to decide camera orientation. Finally when we snap the shutter, the seen is cropped to the size of window of the camera and the light from the visible surfaces is projected into the camera film.

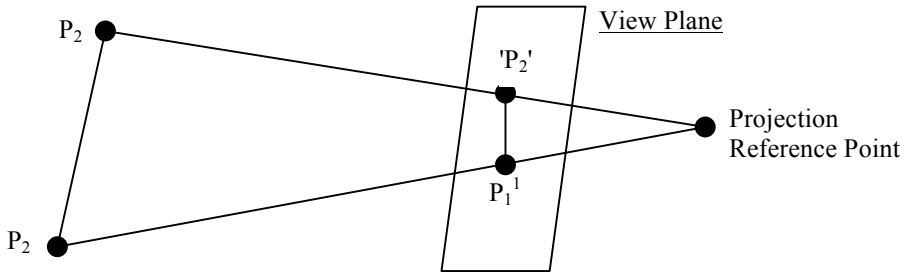


Projections:
Once world co-ordinate description of the objects in a scene are converted to viewing co-ordinates, we can project the three dimensional objects onto the two dimensional view plane. There are two basic projection methods:

Parallel projection: In parallel projection, co-ordinates positions are transformed to the view plane along parallel lines.



Prospective projection: In prospective projection, objects positions are transformed to the view plane along lines that converge to a point called projection reference point (centre of projection). The projected view of an object is determined by calculating the intersection of the projection lines with the view plane.



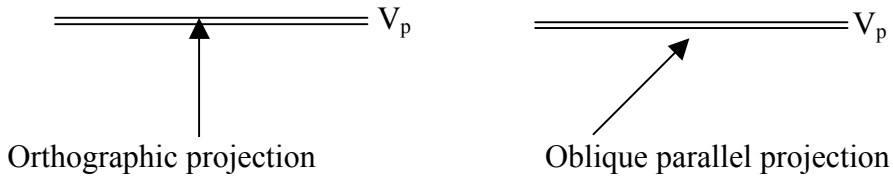
Prospective projection

A parallel projection preserve relative proportions of objects and this is the method used in drafting in drafting to produce scale drawing of three-dimensional objects. Accurate view of various sides of 3D object is obtained with parallel projection. But it does not given a realistic appearance of a 3D-object.

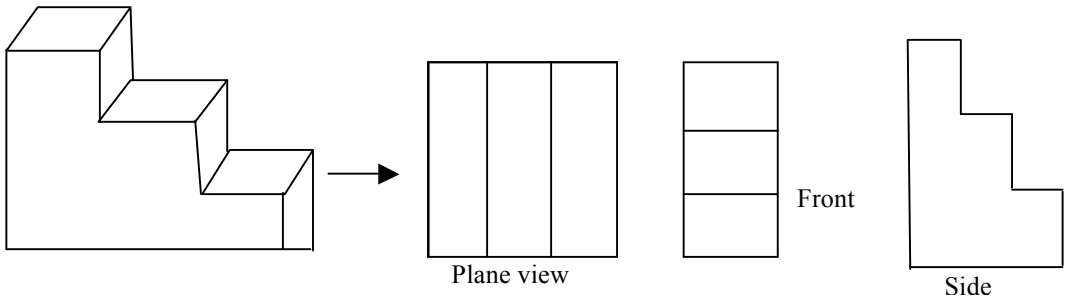
A prospective projection, on the other hand, produces realistic views but does not preserve relative proportions. Projections of distance objects from view plane are smaller than the projections of objects of the same size that are closer to the projection place.

Parallel Projection: We can specify parallel projection withy projection vector that specifies the direction of projection line. When the projection lines are perpendicular to view plane, the projection is orthographic parallel projections.

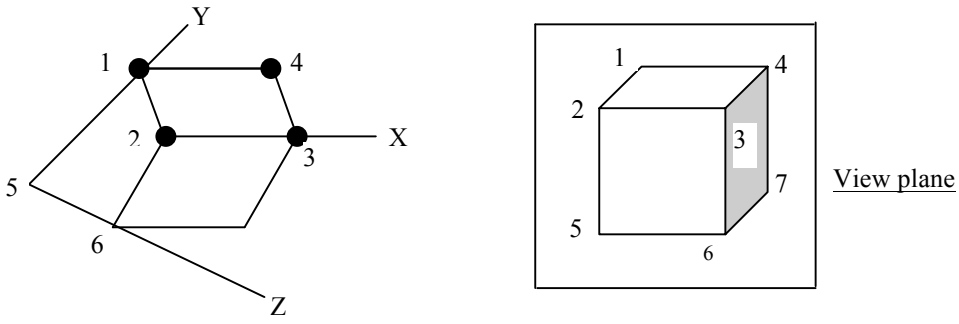
It projection line are not parallel to view plane then it is oblique parallel projection.



- Orthographic projections are most often used to produce the front, side, and top views of an object. Front, side and rear orthographic are called elevations and the top orthographic view of object is known as plan view. Engineering and Architectural drawings commonly employ these orthographic projections.



We also from orthographic projections that display more than one face of an object. Such views are called axonometric orthographic projections. the most commonly used axonometric projection is the isometric projection.



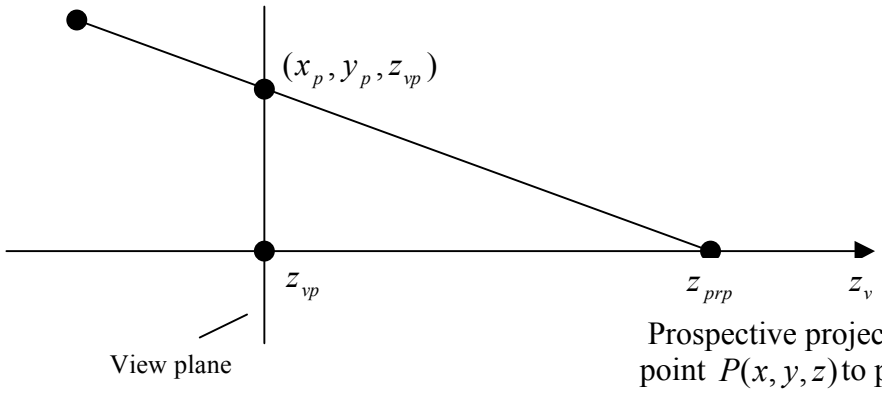
The transformation eq^n for orthographic projection is

$x_p = x,$ $y_p = y,$ z – Coordinate value is preserved for the depth information:

Prospective projections:

To obtain a prospective projection of a three-dimensional object, we transform points along projection lines that meet at a point called projection reference point.

Suppose we set the projection reference point at position Z_{prp} along the Z_v axis, and we place the view plane at Z_{vp} as shown is fig



Prospective projection of a point $P(x, y, z)$ to position (x_p, y_p, z_{vp}) on the view plane.

We can write equations describing co-ordinates positions along this prospective projection line in parametric form as

$$x' = x - xu$$
$$y' = y - yu$$
$$z' = z - (z - z_{prp})u$$

Where u takes value from $0 to 1$. If $u = 0,$ we are at position $P = (x, y, z).$ If $u = 1,$ we have projection reference point $(0,0,z_{prp})$. On the view plane, $z' = z_{vp}$ then

$$u = \frac{z_{vp} - z}{z_{prp} - z}$$

Substituting these value in eq^n for x', y'.

$$x_p = x - x(\frac{z_{vp} - z}{z_{prp} - z}) = x(\frac{z_{prp} - z_{vp}}{z_{prp} - z}) = x(\frac{dp}{z_{prp} - z})$$

Similarly,

$$y_p = y(\frac{z_{prp} - z_{vp}}{z - z_{prp}}) = y(\frac{dp}{z_{prp} - z})$$

Where $dp = z_{prp} - z_{vp}$ is the distance of the view plane from projection reference point.

Using 3-D homogeneous Co-ordinate representation, we can write prospective projection transformation matrix as

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & z_{vp}/dp & \frac{-z_{vp}}{z_{prp}/dp} \\ 0 & 0 & 1/dp & z_{prp}/dp \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

In this representation the homogeneous factor $h = \frac{z - z_{prp}}{dp}$

Projection co-ordinates,

$$xp = xh/h, yp = yh/h$$

There are special case for prospective transformation.

When $z_{prp} = 0$;

$$x_p = x(\frac{z_{prp}}{z_{prp} - z}) = x(\frac{1}{1 - z/z_{prp}})$$

$$y_p = y(\frac{z_{prp}}{z_{prp} - z}) = y(\frac{1}{1 - z/z_{prp}})$$

Some graphics package, the projection point is always taken to be viewing co-ordinate origin. In this ease, $z_{prp} = 0$

$$\therefore x_p = x(\frac{zvp}{z}) = x(\frac{1}{z/zvp})$$

$$yp = y(\frac{zvp}{z}) = y\,(\frac{1}{z/zvp})$$

Visible Surface Detection Methods

(Hidden surface elimination)

Visible surface detection or Hidden surface removal is major concern for realistic graphics for identifying those parts of a scene that are visible from a choosen viewing position. Several algorithms have been developed. Some require more memory, some require more processing time and some apply only to special types of objects.

Visible surface detection methods are broadly classified according to whether they deal with objects or with their projected images.

These two approaches are

- **Object-Space methods:** Compares objects and parts of objects to each other within the scene definition to determine which surface as a whole we should label as visible.

- **Image-Space methods**: Visibility is decided point by point at each pixel position on the projection plane.

Most visible surface detection algorithm use image-space-method but in some cases object space methods are also used for it.

BACK-FACE DETECTION(Plane Equation method)

A fast and simple object space method used to remove hidden surface from a 3D object drawing is known as "Plane equation method" and applied to each side after any rotation of the object takes place. It is commonly known as back-face detection of a polyhedron is based on the "inside-outside" tests.

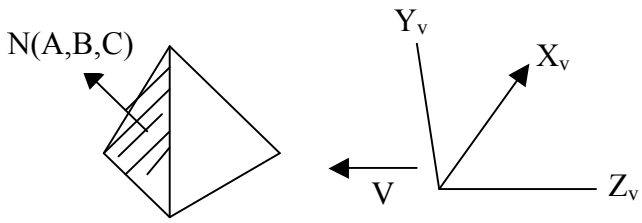
A point (x, y, z) is inside a polygon surface if

$Ax + By + Cz + D < O$

We can simplify this test by considering the normal vector N to a polygon surface which has Cartesian components (A, B, C)

If V is the vector in viewing direction from the eye position then this polygon is a back face if,

V.N>0



In the equation $Ax + By + Cz + D = 0$, if A,B,C remains constant, then varying value of D results in a whole family of parallel planes. One of which (D = 0) contains the origin of the co-ordinates system and ,
If D > 0, plane is behind the origin(Away from observer)
If D < 0, plane is in front of origin(towards the observer)

If we clearly defined our object to have centered at origin, the all those surface that are viewable will have negative D and unviewable surface have positive D.

So , simply our hidden surface removal routine defines the plane corresponding to one of 3D surface from the co-ordinate of 3 points on it and computing D , visible surface are detected.

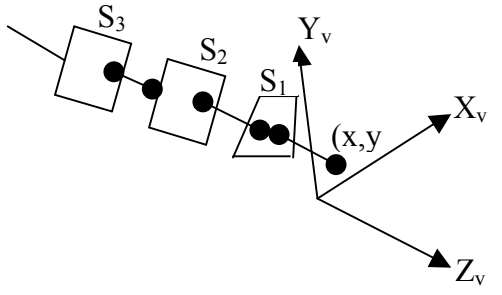
DEPTH-BUFFER-METHOD:
Depth Buffer Method is the commonly used image-space method for detecting visible surface. It is also know as z-buffer method. It compares surface depths at each pixel position on the projection plane. It is called z-buffer method since object depth is usually measured from the view plane along the z-axis of a viewing system.

Each surface of scene is processed separately, one point at a time across the surface. The method is usually applied to scenes containing only polygon surfaces, because depth values can be computed very quickly and method is easy to implement. This method can be apply to non planer surfaces.

With object description converted to projection co-ordinates, each (x, y, z) position on polygon surface corresponds to the orthographic projection point (x, y) on the view plane. Therefore for each pixel position (x, y) on the view plane, object depth is compared by z. values.

With objects description converted to projection co-ordinates, each (X,Y,Z) position on polygon surface correspond to the orthographic projection point (X,Y) on the view plane. The object depth is compared by Z-values.

In figure, three surface at varying distance from view plane X_vY_v, the projection along (x,y) surface S_1 is closest to the view-plane so surface intensity value of S_1 at (x,y) is saved.



In Z-buffer method, two buffers area are required. A depth buffer is used to store the depth value for each (x,y) position or surface are processed, and a refresh buffer stores the intensity value for each position. Initially all the position in depth buffer are set to 0, and refresh buffer is initialize to background color. Each surface listed in polygon table are processed one scan line at a time, calculating the depth (z-val) for each position (x,y). The calculated depth is compared to the value previously stored in depth buffer at that position. If calculated depth is greater than stored depth value in depth buffer, new depth value is stored and the surface intensity at that position is determined and placed in refresh buffer.

Algorithm: Z-buffer

1. Initialize depth buffer and refresh buffer so that for all buffer position (x,y)
   depth (x,y) =0, refresh (x,y) = $I_{background.}$
2. For each position on each polygon surface, compare depth values to previously stored value in depth buffer to determine visibility.
   - Calculate the depth Z for each (x,y) position on polygon
   - If Z> depth (x,y) then
     depth (x,y) =Z
     refresh (x,y) = $I_{surface}$ (x,y)

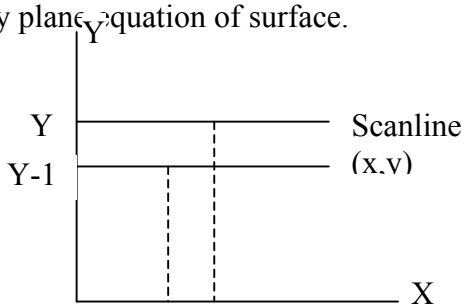
Where $I_{background}$ is the intensity value for background and $I_{surface}$ (x,y) is intensity value for surface at pixel position (x,y) on projected plane. After all surface are processed, the depth buffer contains the depth value of the visible surface and refresh buffer contains the corresponding intensity values for those surface. The depth value of the surface position (x,y) are calculated by plane equation of surface.

$$Z = \frac{-Ax - By - D}{C}$$

Let Depth Z' at position (x+1,y)

$$Z' = \frac{-A(x+1) - By - D}{C}$$

$$\Rightarrow Z' = Z - A/C \quad \text{——} \quad (1)$$

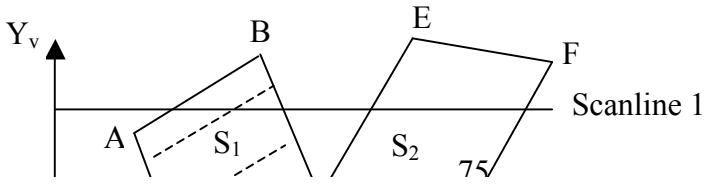


$-A/C$ is constant for each surface so succeeding depth x ax+1 cross a scan line are obtained form preceding values by simple calculation.

SAN LINE METHOD :

This is image-space method for removing hidden surface which is extension of the scan line polygon filling for polygon interiors. Instead of filling one surface we deal with multiple surface here.

As each scan line is processed, all polygon surface intersecting that line are examined to determine which are visible. We assume that polygon table contains the co-efficient of the plane equation for each surface as well as vertex edge, surface information, intensity information for the surface, and possibly pointers to the edge table.



- In figure above, the active edge list for scan line 1 contains information from edge table for edge AB, BC, EH, FG.
- For positions along this scan line between edge AB and BC, only the flag for surface S_1 is on
- Therefore no depth calculation must be made using the plane coefficients for two surface and intensity information for surface S_1 is entered from the polygon table into the refresh buffer.

→ Similarly between EH&FG. Only the flag for S_2 is on. No other positions along scan line 1 intersect surface. So intensity values in the other areas are set to background intensity

→ For Scanline 2 & 3, the active edge list contains edges AD, EH BC, FG. Along scaline 2 from edge AD, to edge EH only surface flag for S_1 is on, but between edges EH& BC, the flags for both surface is on. In this interval, depth calculation is made using the plane coefficients for the two surfaces.

For example, if Z of surface S_1 is less than surface S_2, So the intensity of S_1 is loaded into refresh buffer until boundary BC is encountered. Then the flag for surface S_1 goes off and intensities for surface S_2 are stored until edge FG is passed.

Any no of overlapping surface are processed with this scan line methods.

DEPTH SORTING METHOD:

This method uses both object space and image space method. In this method the surface representation of 3D object are sorted in of decreasing depth from viewer. Then sorted surface are scan converted in order starting with surface of greatest depth for the viewer.

The conceptual steps that performed in depth-sort algorithm are

1. Sort all polygon surface according to the smallest (farthest) Z co-ordinate of each.

2. Resolve any ambiguity this may cause when the polygons Z extents overlap, splitting polygons if necessary.

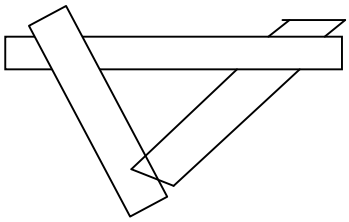
3. Scan convert each polygon in ascending order of smaller Z-co-ordinate i.e. farthest surface first (back to front)

In this method, the newly displayed surface is partly or completely obscure the previously displayed surface. Essentially, we are sorting the surface into priority order

such that surface with lower priority (lower z, far objects) can be obscured by those with higher priority (high z-value).

This algorithm is also called "Painter's Algorithm" as it simulates how a painter typically produces his painting by starting with the background and then progressively adding new (nearer) objects to the canvas.

Problem: One of the major problem in this algorithm is intersecting polygon surfaces. As shown in fig. below.



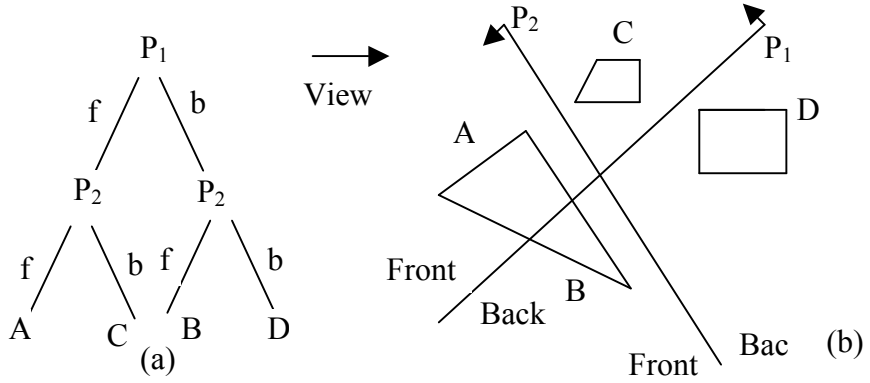
- Different polygons may have same depth.
- The nearest polygon could also be farthest.

We cannot use simple depth-sorting to remove the hidden-surfaces in the images.
Solution: For intersecting polygons, we can split one polygon into two or more polygons which can then be painted from back to front. This needs more t6ime to compute intersection between polygons. So it becomes complex algorithm for such surface existence.

BSP TREE METHOD:

A binary space partitioning (BSP) tree is an efficient method for determining object visibility by painting surfaces onto the screen from back to front as in the painter's algorithm. The BSP tree is particularly useful when the view reference point changes, but object in a scene are at fixed position.

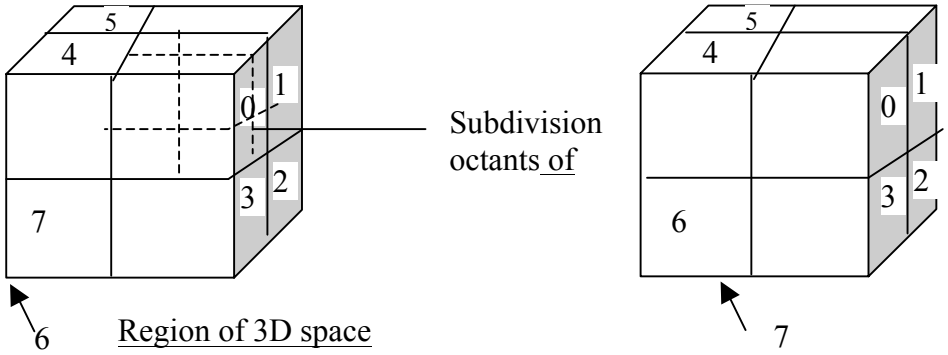
Applying a BSP tree to visibility testing involves identifying surfaces that are "inside" or "outside" the partitioning plane at each step of space subdivision relative to viewing direction. It is useful and efficient for calculating visibility among a static group of 3D polygons as seen from an arbitrary viewpoint.
In the following figure,



Here plane P_1 partitions the space into two sets of objects, one set of object is back and one set is in front of partitioning plane relative to viewing direction. Since one object is intersected by plane P_1, we divide that object into two separate objects labeled A and B. Now object A&C are in front of P_1, B and D are4 back of P_1.

We next partition the space with plane P_2 and construct the binary free as fig (a). In this tree, the objects are represented as terminal nodes, with front object as left branches and behind object as right branches.
When BSP tree is complete, we process the tree by selecting surface for displaying in order back to front. So fore ground object are painted over back ground objects.

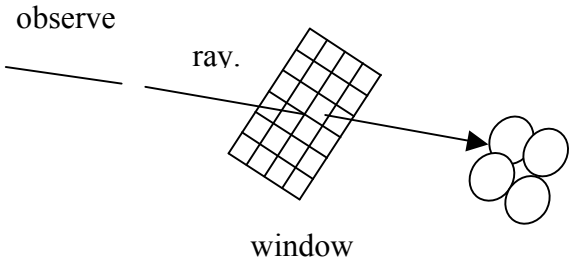
Octree Method: When an octree representation is used for viewing volume, hidden surface elimination is accomplished by projecting octree nodes into viewing surface in a front to back order. Following figure is the front face of a region space is formed with octants 0,1,2,3. Surface in the front of these octants are visible to the viewer. The back octants 4,5,6,7 are not visible. After octant sub-division and construction of octree, entire region is traversed by depth first traversal.



RAY TRACING:

Ray tracing also known as ray casting is efficient method for visibility detection in the objects. It can be used effectively with the object with curved surface. But it also used for polygon surfaces.

- Trace the path of an imaginary ray from the viewing position (eye) through viewing plane t object in the scene.
- Identify the visible surface by determining which surface is intersected first by the ray.
- Can be easily combined with lightning algorithms to generate shadow and reflection.
- It is good for curved surface but too slow for real time application.



Illumination and Surface Rendering:

- Realistic displays of a scene are obtained by perspective projections and applying natural lighting effects to the visible surfaces of object.
- An illumination model is also called lighting model and some times called as a shading model which is used to calculate the intensity of light that we should see at a given point on the surface of a object.
- A surface-rendering algorithm uses the intensity calculations from an illumination model.

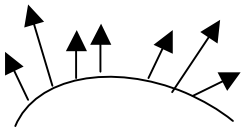
Light Sources:

- Sometimes light sources are referred as light emitting object and light reflectors. Generally light source is used to mean an object that is emitting radiant energy e.g. Sun.

Point Source: Point source is the simplest light emitter e.g. light bulb.

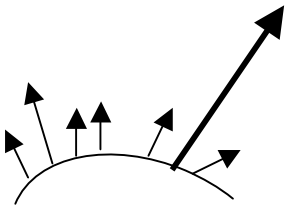
Distributed light source: Fluorescent light

- When light is incident on an opaque surface part of it is reflected and part of it is absorbed.
- Surface that are rough or grainy, tend to scatter the reflected light in all direction which is called diffuse reflection.



Diffuse reflection

- When light sources create highlights, or bright spots, called specular reflection



Specular reflection

Illumination models:

Illumination models are used to calculate light intensities that we should see at a given point on the surface of an object. Lighting calculations are based on the optical properties of surfaces, the background lighting conditions and the light source specifications. All light sources are considered to be point sources, specified with a co-ordinate position and an intensity value (color). Some illumination models are:

1. **Ambient light**:

- This is a simplest illumination model. We can think of this model, which has no external light source-self-luminous objects. A surface that is not exposed directly to light source still will be visible if nearby objects are illuminated.
- The combination of light reflections form various surfaces to produce a uniform illumination is called ambient light or background light.
- Ambient light has no spatial or directional characteristics and amount on each object is a constant for all surfaces and all directions. In this model, illumination can be expressed by an illumination equation in variables associated with the point on the object being shaded. The equation expressing this simple model is

$$I = K_a$$

Where I is the resulting intensity and K_a is the object's intrinsic intensity.

If we assume that ambient light impinges equally on all surface from all direction, then

$$I = I_a K_a$$

Where I_a is intensity of ambient light. The amount of light reflected from an object's surface is determined by K_a, the ambient-reflection coefficient.

K_a ranges from 0 to 1.

2. Diffuse reflection:

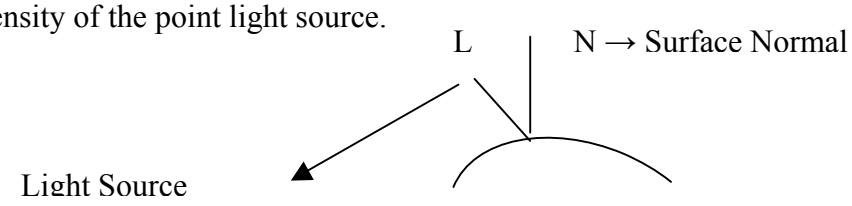
Objects illuminated by ambient light are uniformally illuminated across their surfaces even though light are more or less bright in direct proportion of ambient intensity. Illuminating object by a point light source, whose rays enumerate uniformly in all directions from a single point. The object's brightness varies form one part to another, depending on the direction of and distance to the light source.

- The fractional amount of the incident light that is diffusely reflected can be set for each surface with parameter K_d, the coefficient of diffuse-reflection.
- Value of K_d is in interval 0 to 1. If surface is highly reflected, K_d is set to near 1. The surface that absorbs almost incident light, K_d is set to nearly 0.
- Diffuse reflection intensity at any point on the surface if exposed only to ambient light is

  $$\text{Im}abdiff = I_d K_d$$
- Assuming diffuse reflections from the surface are scattered with equal intensity in all directions, independent of the viewing direction (surface called. "Ideal diffuse reflectors") also called Lambertian reflectors and governed by Lambert's cosine law.

  $$Idiff = K_d I_l \cos\theta$$

Where I_l is the intensity of the point light source.



It N is unit vector normal to the surface & L is unit vector in the direction to the point slight source then

$$I_l, diff = K_d I_l (N.L)$$

In addition, many graphics packages introduce an ambient reflection coefficient K_a to modify the ambient-light intensity I_a

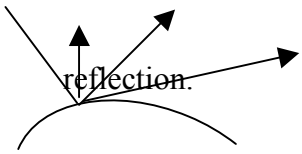
$$Idiff = K_a I_a + K_d I_l (N.L)$$

3. Specular reflection and pong model.

When we look at an illuminated shiny surface, such as polished metal, a person's forehead, we see a highlight or bright spot, at certain viewing direction. Such phenomenon is called specular reflection. It is the result of total or near total

reflection of the incident light in a concentrated region around the " specular reflection angle = angle of incidence".

Let SR angle = angle of incidence as in figure.



N - unit vector normal to surface at incidence point

R - unit vector in the direction of ideal specular reflection.

L - unit vector directed to words point light source.

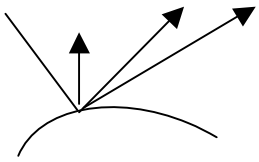
V - unit vector pointing to the viewer from surface.

ϕ – the viewing angle relative to the specular reflection direction.

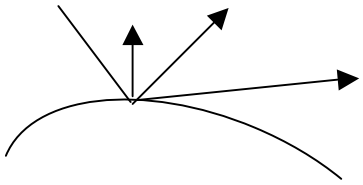
For ideal reflector (perfect mirror), incident light is reflected only in the specular reflection direction.

$\therefore V \,\&\, R$ concide $(\phi = 0)$

- Shiny surface have narrow ϕ and dull surface wider ϕ.
- An empirical model for calculating specular-reflection range developed by Phong Bui Tuong-called "Phong specular reflection model (or simply Phong model), sets the intensity of specular reflection proportional to $\cos^{ns}\phi \to 0 to 90^0$.
- Specular reflection parameter n_s is determined by type of surface
- Very shiny surface with large value n_s (say 100 or more) and dull surface , smaller n_s (down to 1)
- Rough surface such as chalk , $n_s = 1$.



Shhiny surface (large $n_{s)}$



Dull surface (small $n_{s)}$

- Intensity of specular reflection depends upon material properties of the surface and θ. Other factors such as the polarization and color of the incident light.
- For monochromatic specular intersity variations can approximated by SR coefficient $w(\theta)$

- Fresnal's law of reflection describe specular reflection intensity with θ and using $w(\theta)$, Phong specular reflection model as

$I_{spec} = w(\theta) I_l \cos^{n_s} \phi$

Where I_l is intensity of light source. ϕ is viewing angle relative to SR direction R.

For a glass, we can replace $w(\theta)$ with constant K_s specular reflection coefficient.

So , $I_{spec} = K_s I_l \cos^{n_s} \phi$

] $= K_s I_l (V.R)^{n_s}$ Since $\cos\phi$ = V.R

Polygon (surface) Rendering Method

- Application of an illumination model to the rendering of standard graphics objects those formed with polygon surfaces are key technique for polygon rendering algorithm.
- Calculating the surface normal at each visible point and applying the desired illumination model at that point is expensive. We can describe more efficient shading models for surfaces defined by polygons and polygon meshes.
- Scan line algorithms typically apply a lighting model to obtain polygon surface rendering in one of two ways. Each polygon can be rendered with a single intensity, or the intensity can be obtained at each point of the surface using an interpolating scheme.

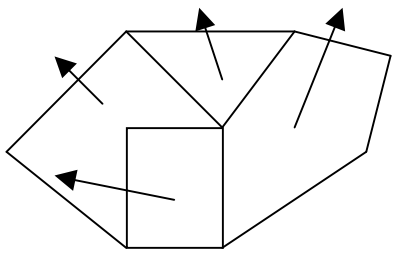
1. **Constant Intensity Shading**: (Flat Shading)

The simplest model for shading for a polygon is constant intensity shading also called as **Faceted Shading** Or flat shading. This approach implies an illumination model once to determine a single intensity value that is then used to render an entire polygon. Constant shading is useful for quickly displaying the general appearance of a curved surface.

This approach is valid if several assumptions are true:

1. The light source is at infinity, so **N.L** is constant across the polygon face.
2. The viewer is at infinity, so **N.V** is constant across the polygon face.
3. The polygon represents the actual surface being modeled and is not an approximation to a curved surface.

Even if all conditions are not true, we can still reasonably approximate surface -lighting effects using small polygon facets with fast shading and calculate the intensity for each facet, at the centre of the polygon of course constant shading does not produce the variations in shade a cross the polygon that should occur.



2. **Interpolated Shading**:

An alternative to evaluating the illumination equation at each point on the polygon, we can use the interpolated shading, in which shading information is linearly interpolated across a triangle from the values determined for its vertices. Gouraud generalized this technique for arbitrary polygons. This is particularly easy for a scan line algorithm that already interpolates the z- value across a span from interpolated z-values computed for the span's endpoints.

Gouraud Shading:

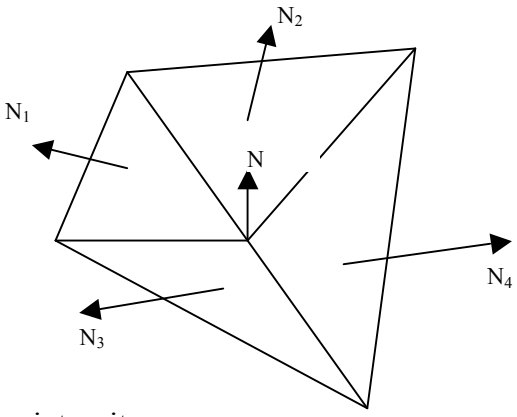
Gouraud shading , also called intensity interpolating shading or color interpolating shading, eliminates intensity discontinuities that occur in flat shading. Each polygon surface is rendered with Gouraud shading by performing following calculations.

1. Determine the average unit normal vector at each vertex. At each polygon vertex, we obtain a normal vertex by averaging the surface normals of all polygons sharing the vertex as:

$$N_v = \frac{\sum_{k=1}^{n} N_k}{|\sum_{k=1}^{n} N_k|}$$

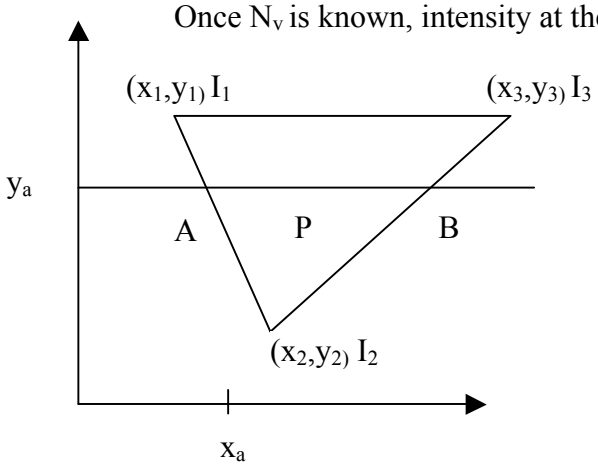
Here $N_v = \frac{N_1 + N_2 + N_3 + N_4}{|N_1 + N_2 + N_3 + N_4|}$

Where N_v is normal vector at a vertex sharing Four surfaces as in figure.



2. Apply illumination model to calculate each vertex intensity.
3. Linearly interpolate the vertex intensity over the surface of the polygon.

Once N_v is known, intensity at the vertices can obtain from lighting model.



-Here in figure, the intensity of vertices 1,2,3 are I_1 , I_2, I_3 . are obtained by averaging normals of each surface sharing the vertices and applying a illumination model.

- For each scan line , intensity at intersection of line with Polygon edge are linearly interpolated from the intensities at the edge end point.

So Intensity at intersection point A, I_a is obtained by linearly interpolating intensities of I_1 and I_2 as`

$$I_a = \frac{y_a - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y_a}{y_1 - y_2} I_2$$

Similarly, the intensity at point B is obtained by linearly interpolating intensities at I_2 and I_3 as

$$I_b = \frac{y_a - y_2}{y_3 - y_2} I_3 + \frac{y_3 - y_a}{y_3 - y_2} I_2$$

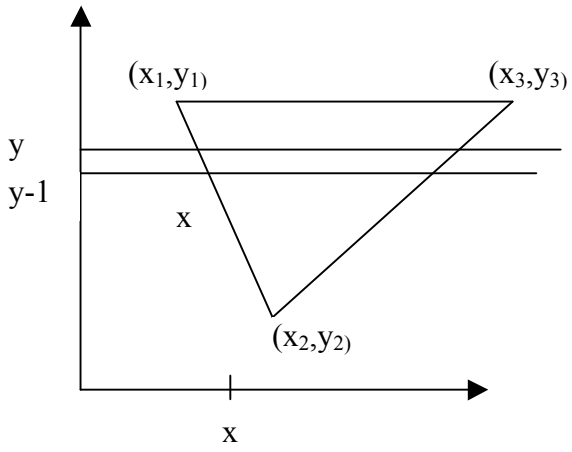
The intensity of a point P in the polygon surface along scan-line is obtained by linearly interpolating intensities at I_a and I_b as,

$$I_p = \frac{x_p - x_a}{x_b - x_a} I_b + \frac{x_b - x_p}{x_b - x_a} I_a$$

Then incremental calculations are used to obtain Successive edge intensity values between scan-lines as :

$$I = \frac{y - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y}{y_1 - y_2} I_2$$

Then we can obtain the intensity along this edge for next scan line at y -1 position as



$$I' = \frac{y - 1 - y_2}{y_1 - y_2} I_1 + \frac{y_1 - (y - 1)}{y_1 - y_2} I_2$$

$$= I + \frac{I_2 - I_1}{y_1 - y_2}$$

Similar calculations are made to obtain intensity successive horizontal pixel.

Advantages: Removes intensity discontinuities at the edge as compared to constant shading.

Disadvantages: Highlights on the surface are sometimes displayed with anomalous shape and linear intensity interpolation can cause bright or dark intensity streak called mach-bands.

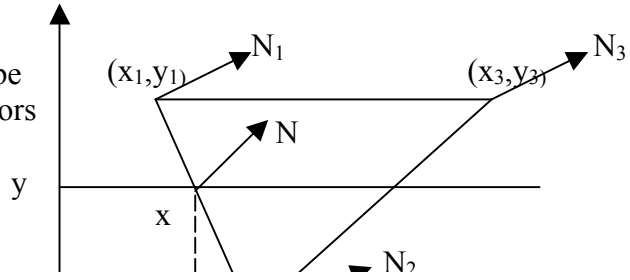
Phong Shading:

A more accurate method for rendering a polygon surface is to interpolate normal vector and then apply illumination model to each surface point. This method is called Phong shading or normal vector interpolation method for shading. It displays more realistic highlights and greatly reduce the mach band effect.

A polygon surface is rendered with Phong shading by carrying out following calculations.

- Determine the average normal unit vectors at each polygon vertex.
- Linearly interpolate vertex normals over the surface of polygon.
-Apply illumination model along each scan line to calculate the pixel intensities for the surface point.

In figure, N_1, N_2, N_3 are the normal unit vectors at each vertex of polygon surface. For scan-line that intersect an edge, the normal vector N can be obtained by vertically interpolating normal vectors of the vertex on that edge as.



$$N = \frac{y - y_2}{y_1 - y_2} N_1 + \frac{y_1 - y}{y_1 - y_2} N_2$$

Incremental calculations are used to evaluate normals between scanlines and along each individual scan line as in Gouraud shading. Phong shading produces accurate results than the direct interpolation but it requires considerably more calculations.

Fast Phong Shading:

Fast Phong shading approximates the intensity calculations using a Taylor series expansion and Triangular surface patches. Since Phong shading interpolates normal vectors from vertex normals , we can express the surface normal N at any point (x,y) over a triangle as

$N = Ax + By + C$

Where A,B,C are determined from the three vertex equations.

$N_k = Ax_k + By_k + C$, $k = 1,2,3$ for (x_k , y_k) vertex.

Omitting the reflectivity and attenuation parameters

$$I_{diff}(x, y) = \frac{L.N}{|L|.|N|} = \frac{L.(Ax + By + C)}{|L|.|Ax + By + C|} = \frac{(L.A)x + (L.B)y + (L.C)}{|L|.|Ax + By + C|}$$

Re writing this

$$I_{diff}(x, y) = \frac{ax + by + c}{(dx^2 + exy + fy^2 + gx + hy + i)^{\frac{1}{2}}} \text{ ----------(1)}$$

Where a,b,c,d….. are used to represent the various dot products as

$a = \frac{L.N}{|L|}$ .....and so on

Finally, denominator of equation (1) can express as Taylor series expansions and relations terms up to second degree in x,y. This yields,

$$I_{diff}(x, y) = T_5 x^2 + T_4 xy + T_3 y^2 + T_2 x + T_1 y + T_0$$

Where each T_k is a function of parameters a,b,c,d,……. And so forth.

This method still takes twice as long as in Gouraud shading. Normal Phong shading takes six to seven times that of Gouraud shading

//// scaling program////

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<graphics.h>
void matrixmulti(int translate[3][3],int vertex[3][5],int result[3][5]);
void main()
{
        int gd=DETECT,gm,errorcode;
        initgraph(&gd,&gm,"\\tc\\bgi");
        errorcode=graphresult();
        if(errorcode!=grOk)
        {
                printf("Graphics ERROR!: %s",grapherrormsg(errorcode));
                printf("\nPress any Key .....");
                getch();
                exit(1);
        }
        setbkcolor(2);
        int vertex[3][5]={{100,100,200,200,100},
        {100,200,200,100,100},
        {1,1,1,1,1}};
        for(int i=0;i<4;i++)
        {
                setcolor(BLUE);
                line(vertex[0][i],vertex[1][i],vertex[0][i+1],vertex[1][i+1]);

        }

        printf("SCALE X=");
        int sx,sy;
        scanf("%d",&sx);
        printf("SCALE Y:=");
        scanf("%d",&sy);
        int translate[3][3]={{2,0,0},{0,1,0},{0,0,1}};
        translate[0][0]=sx;
        translate[1][1]=sy;
        printf("Press any key for the translated line.....\n");
        getch();
        int result[3][5];
        matrixmulti(translate,vertex,result);
        for(i=0;i<4;i++)
        {
                setcolor(YELLOW);
                line(result[0][i],result[1][i],result[0][i+1],result[1][i+1]);

        }
        getch();
        closegraph();
```

```
}
void matrixmulti(int translate[3][3],int vertex[3][5],int result[3][5])
{
        for(int i=0;i<=3;i++)
        {
                for(int j=0;j<=5;j++)
                {
                        result[i][j]=0;
                        for(int k=0;k<=3;k++)
                                result[i][j]+=translate[i][k]*vertex[k][j];
                }
        }
}
```

Translation program

```
#include<stdio.h>
#include<conio.h>85

#include<stdlib.h>
#include<graphics.h>
void matrixmulti(int translate[3][3],int vertex[3][5],int result[3][5]);
void main()
{
        int gd=DETECT,gm,errorcode;
        initgraph(&gd,&gm,"\\tc\\bgi");
        errorcode=graphresult();
        if(errorcode!=grOk)
        {
                printf("Graphics ERROR!: %s",grapherrormsg(errorcode));
                printf("\nPress any Key .....");
                getch();
                exit(1);
        }
        setbkcolor(2);
        int
vertex[3][5]={{100,100,200,200,100},{100,200,200,100,100},{1,1,1,1,1}};
        for(int i=0;i<4;i++)
        {
                setcolor(BLUE);
                line(vertex[0][i],vertex[1][i],vertex[0][i+1],vertex[1][i+1]);

        }

        int translate[3][3]={{1,0,100},{0,1,200},{0,0,1}};
        printf("Press any key for the translated line.....\n");
        getch();
```

```
        int result[3][5];
        matrixmulti(translate,vertex,result);
        for(i=0;i<4;i++)
        {
                setcolor(YELLOW);
                line(result[0][i],result[1][i],result[0][i+1],result[1][i+1]);

        }
        getch();
        closegraph();
}
void matrixmulti(int translate[3][3],int vertex[3][5],int result[3][5])
{
        for(int i=0;i<=3;i++)
        {
                for(int j=0;j<=5;j++)
                {
                        result[i][j]=0;
                        for(int k=0;k<=3;k++)
                                result[i][j]+=translate[i][k]*vertex[k][j];
                }
        }
}
```

//lab 1 Graphics practicle

```
#include<graphics.h>
#include<stdio.h>
#include<process.h>
#include<dos.h>
#include<conio.h>
void main()
{
```

```
        int gd,gm,errorcode;
        int x,y;
        gd=DETECT;
        initgraph(&gd,&gm,"\\BC45\\BGI");
        errorcode=graphresult();
        if(errorcode!=grOk)
        {
                printf("Graphic Error:%s",grapherrormsg(errorcode));
                printf("Press any key....");
                getch();
                exit(1);
        }
        x=getmaxx();
        y=getmaxy();
        for(int i=1;i<=15;i++)
        {
                for(int j=1;j<=11;j++)
                {
                if(kbhit())
                        exit(1);
                setcolor(i);
                setfillstyle(j,i+1);
                circle(x/2,y/2,75);
                floodfill(x/2,y/2,i);
                rectangle(10,10,200,200);
                floodfill(15,15,i);
                //delay(1000);
                fillellipse(400,400,100,75);
                floodfill(400,400,i);
                delay(1000);
                cleardevice();
        }}
}
```

// lab 2 Graphics practicle

```
#include<graphics.h>
#include<stdio.h>
#include<process.h>
#include<dos.h>
#include<conio.h>
void main()
```

```
{
        int gd,gm,errorcode;
        int x,y;
        gd=DETECT;
        initgraph(&gd,&gm,"\\tc\\bgi");
        errorcode=graphresult();
        if(errorcode!=grOk)
        {
                printf("Graphic Error:%s",grapherrormsg(errorcode));
                printf("Press any key....");
                getch();
                exit(1);
        }
        x=getmaxx();
        y=getmaxy();
        for(int i=1;i<=15;i++)
        {
                for(j=1;j<=11;j++)
                setcolor(i);
                setfillstyle(j,i+1);
                circle(x/2,y/2,150);
                floodfill(x/2,y/2);
                rectangle(10,10,200,200);
                floodfill(15,9);
                delay(1000);
                cleardevice();
        }
}
```

//lab 3 Graphics practicle

```
#include<graphics.h>
#include<stdio.h>
#include<process.h>
#include<dos.h>
#include<conio.h>
void main()
{
        int gd,gm,errorcode;
        int x,y;
        gd=DETECT;
        initgraph(&gd,&gm,"\\bc45\\bgi");
        errorcode=graphresult();
        if(errorcode!=grOk)
        {
```

```
            printf("Graphic Error:%s",grapherrormsg(errorcode));
            printf("Press any key....");
            getch();
            exit(1);
        }
        x=getmaxx();
        y=getmaxy();
        while(!kbhit())
        {

            for(int i=x/2;i<x;i+=2)
            for(int j=y/2;j<y;j+=2)
            {
            if(kbhit())
            exit(1);
            setcolor(BLUE);
            setfillstyle(5,RED);
            circle(i,j,50);
            floodfill(i,j,BLUE);
            delay(50);
            cleardevice();
            }}
}
```