SUBJECT CODE : 210244

Strictly as per Revised Syllabus of

SAVITRIBAI PHULE PUNE UNIVERSITY

Choice Based Credit System (CBCS)

S.E. (Computer) Semester - I

COMPUTER GRAPHICS

(For IN SEM Exam)

Atul P. Godse

M.S. Software Systems (BITS Pilani)
B.E. Industrial Electronics
Formerly Lecturer in Department of Electronics Engg.
Vishwakarma Institute of Technology
Pune

Dr. Deepali A. Godse

M.E., Ph.D. (Computer Engg.)
Head of Information Technology Department,
Bharati Vidyapeeth's College of Engineering for Women,
Pune

Dr. Rajesh Prasad

Ph.D. (Computer Science & Engg.)
Professor (Department of Computer Engineering),
Sinhgad Institute of Technology & Science,
Narhe, Pune.

TECHNICAL PUBLICATIONS
SINCE 1993
An Up-Thrust for Knowledge

COMPUTER GRAPHICS

(For IN SEM Exam)

Subject Code : 210244

S.E. (Computer) Semester - I

First Edition : July 2020

© Copyright with A. P. Godse and Dr. D. A. Godse

All publishing rights (printed and ebook version) reserved with Technical Publications. No part of this book should be reproduced in any form, Electronic, Mechanical, Photocopy or any information storage and retrieval system without prior permission in writing, from Technical Publications, Pune.

Published by :

TECHNICAL PUBLICATIONS
SINCE 1993
An Up-Thrust for Knowledge

Amit Residency, Office No.1, 412, Shaniwar Peth,
Pune - 411030, M.S. INDIA, Ph.: +91-020-24495496/97
Email : sales@technicalpublications.org Website : www.technicalpublications.org

Printer :
Yogiraj Printers & Binders
Sr.No. 10/1A,
Ghule Industrial Estate, Nanded Village Road,
Tal. - Haveli, Dist. - Pune - 411041.

ISBN 978-81-946628-8-4

9 788194 662884

SPPU 19

PREFACE

The importance of **Computer Graphics** is well known in various engineering fields. Overwhelming response to our books on various subjects inspired us to write this book. The book is structured to cover the key aspects of the subject **Computer Graphics**.

The book uses plain, lucid language to explain fundamentals of this subject. The book provides logical method of explaining various complicated concepts and stepwise methods to explain the important topics. Each chapter is well supported with necessary illustrations, practical examples and solved problems. All the chapters in the book are arranged in a proper sequence that permits each topic to build upon earlier studies. All care has been taken to make students comfortable in understanding the basic concepts of the subject.

Representative questions have been added at the end of each section to help the students in picking important points from that section.

The book not only covers the entire scope of the subject but explains the philosophy of the subject. This makes the understanding of this subject more clear and makes it more interesting. The book will be very useful not only to the students but also to the subject teachers. The students have to omit nothing and possibly have to cover nothing more.

We wish to express our profound thanks to all those who helped in making this book a reality. Much needed moral support and encouragement is provided on numerous occasions by our whole family. We wish to thank the **Publisher** and the entire team of **Technical Publications** who have taken immense pain to get this book in time with quality printing.

Any suggestion for the improvement of the book will be acknowledged and well appreciated.

Authors
A.P. Godse
Dr. D. A. Godse
Dr. Rajesh Prasad

Dedicated to God

SYLLABUS

Computer Graphics - 210244

Credit Scheme	Examination Scheme and Marks :
03	**Mid-Semester (TH) :** 30 Marks

Unit I Graphics Primitives and Scan Conversion Algorithms

Introduction, graphics primitives - pixel, resolution, aspect ratio, frame buffer. Display devices, applications of computer graphics.

Introduction to OpenGL - OpenGL architecture, primitives and attributes, simple modelling and rendering of two- and three-dimensional geometric objects, GLUT, interaction, events and call-backs picking. **(Simple Interaction with the Mouse and Keyboard)**

Scan conversion : Line drawing algorithms : Digital Differential Analyzer (DDA), Bresenham. Circle drawing algorithms : DDA, Bresenham, and Midpoint. **(Chapters - 1,2,3)**

Unit II Polygon, Windowing and Clipping

Polygons : Introduction to polygon, types : convex, concave and complex. Inside test.

Polygon Filling : flood fill, seed fill, scan line fill.

Windowing and dipping : viewing transformations, 2-D clipping: Cohen- Sutherland algorithm line Clipping algorithm, Sutherland Hodgeman Polygon clipping algorithm, Weiler Atherton Polygon Clipping algorithm. **(Chapters - 4,5)**

Table of Contents

Unit - I

Unit - II

UNIT - I

1 Introduction to Computer Graphics

Syllabus

Introduction, graphics primitives - pixel, resolution, aspect ratio, frame buffer. Display devices, applications of computer graphics.

Contents

1.1 Basic Concepts

SPPU : Dec.-09, 11, May-13

- The computer graphics is one of the most effective and commonly used way to communicate the processed information to the user.
- It displays the information in the form of **graphics objects** such as pictures, charts, graphs and diagrams instead of simple text. Thus we can say that computer graphics makes it possible to **express data in pictorial form.**
- The picture or graphics object may be an engineering drawing, business graphs, architectural structures, a single frame from an animated movie or a machine parts illustrated for a service manual.

1.1.1 Pixel

- In computer graphics, pictures or graphics objects are presented as a collection of discrete picture elements called **pixels**.
- The pixel is the smallest addressable screen element.
- It is the smallest piece of the display screen which we can control.
- The control is achieved by **setting the intensity** and **colour** of the pixel which compose the screen. This is illustrated in Fig. 1.1.1.

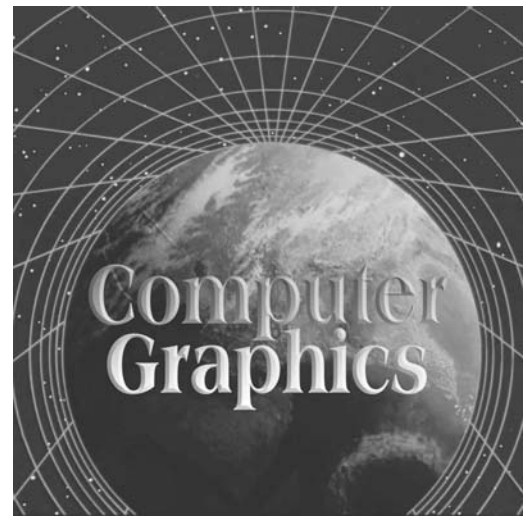


Fig. 1.1.1 Representation of picture

- Each pixel on the graphics display does not represent mathematical point. Rather, it represents a region which theoretically can contain an infinite number of points.
- For example, if we want to display point P_1 whose co-ordinates are (4.2, 3.8) and point P_2 whose co-ordinates are (4.8, 3.1) then P_1 and P_2 are represented by only one pixel (4, 3), as shown in the Fig. 1.1.2.

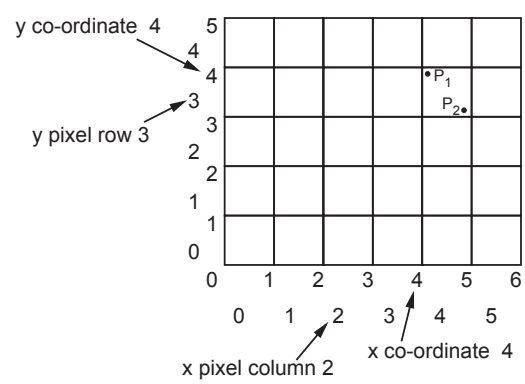


Fig. 1.1.2 Pixel display area of 6 × 5

- In general, a point is represented by the integer part of x and integer part of y, i.e., pixel (int (x), int (y)).

1.1.2 Rasterization and Scan Conversion

- The special procedures determine which pixel will provide the best approximation to the desired picture or graphics object.
- The process of determining the appropriate pixels for representing picture or graphics object is known as **rasterization**, and the process of representing continuous picture or graphics object as a collection of discrete pixels is called **scan conversion**.

1.1.3 Other Elements of Computer Graphics

- The computer graphics allows rotation, translation, scaling and performing various **projections** on the picture before displaying it.
- It also allows to add effects such as **hidden surface removal**, **shading** or **transparency** to the picture before final representation.
- It provides user the control to modify contents, structure, and appearance of pictures or graphics objects using input devices such as a keyboard, mouse, or touch-sensitive panel on the screen. There is a close relationship between the input devices and display devices. Therefore, graphics devices includes both input devices and display devices.

Review Questions

1. *What is pixel ?* **SPPU : Dec.-09,11, May-13, Marks 2**
2. *Define rasterization.*
3. *Define scan conversion.*

1.2 Display Devices

SPPU : Dec.-06, 10, 11, 12, 15, May-05, 09, 13, 14, 15

The display devices are also known as **output devices**. The most commonly used output device in a graphics system is a video monitor. The operation of most video monitors is based on the standard **Cathode-Ray-Tube** (CRT) design. Let us see the basics of the CRT.

1.2.1 Cathode-Ray-Tubes

A CRT is an evacuated glass tube. An **electron gun** at the rear of the tube produces a beam of electrons which is directed towards the front of the tube (screen) by a high voltage typically 15000 to 20,000 volts. The negatively charged electrons are then accelerated toward the phosphor coating at the inner side of the screen by a high positive voltage or by using **accelerating anode**. The phosphor substance gives off light when it is stroked by electrons.

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

The **control grid** voltage determines the velocity achieved by the electrons before they hit the phosphor. The control grid voltage determines how many electrons are actually in the electron beam.

A more negative voltage applied to the control grid will shut off the beam. A smaller negative voltage on the control grid simply decreases the number of electrons passing through. Since the amount of light emitted by the phosphor coating depends on the number of electrons striking the screen, we control the brightness of a display by varying the voltage on the control grid.

The **focusing system** concentrates the electron beam so that the beam converges to a small point when it hits the phosphor coating. It is possible to control the point at which the electron beam strikes the screen, and therefore the position of the dot upon the screen, by deflecting the electron beam. Fig. 1.2.1 shows the electrostatic deflection of the electron beam in a CRT.

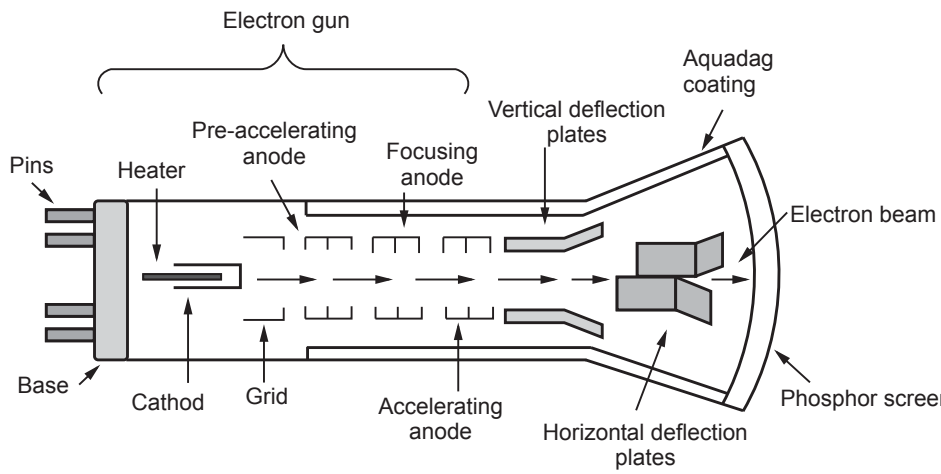


Fig. 1.2.1 Cathode ray tube

The **deflection system** of the cathode-ray-tube consists of two pairs of parallel plates, referred to as the **vertical** and **horizontal deflection** plates. The voltage applied to vertical plates controls the vertical deflection of the electron beam and voltage applied to the horizontal deflection plates controls the horizontal deflection of the electron beam.

The electrons that strike the screen, release secondary emission electrons. Aqueous graphite solution known as **Aquadag** collects these secondary electrons. To maintain the CRT in electrical equilibrium, it is necessary to collect the secondary electrons.

1.2.2 Vector Scan/Random Scan Display

As shown in Fig. 1.2.2, vector scan CRT display directly traces out only the desired lines on CRT i.e. If we want a line connecting point A with point B on the vector graphics display, we simply drive the beam deflection circuitry, which will cause beam to go directly from point A to B. If we want to move the beam from point A to point B without showing a line between points, we can blank the beam as we move it. To move the beam across the CRT, the information about both, magnitude and direction is required. This information is generated with the help of vector graphics generator.

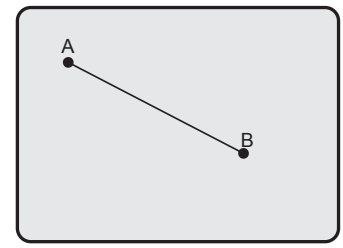


Fig. 1.2.2 Vector scan CRT

The Fig. 1.2.3 shows the typical vector display architecture. It consists of display controller, Central Processing Unit (CPU), display buffer memory and a CRT. A display controller is connected as an I/O peripheral to the central processing unit (CPU). The display buffer memory stores the computer produced display list or display program. The program contains point and line plotting commands with (x, y) or (x, y, z) end point co-ordinates, as well as character plotting commands. The display controller interprets commands for plotting points, lines and characters and sends digital and point co-ordinates to a vector generator. The vector generator then converts the digital co-ordinate values to analog voltages for beam-deflection circuits that displace an electron beam writing on the CRT's phosphor coating.

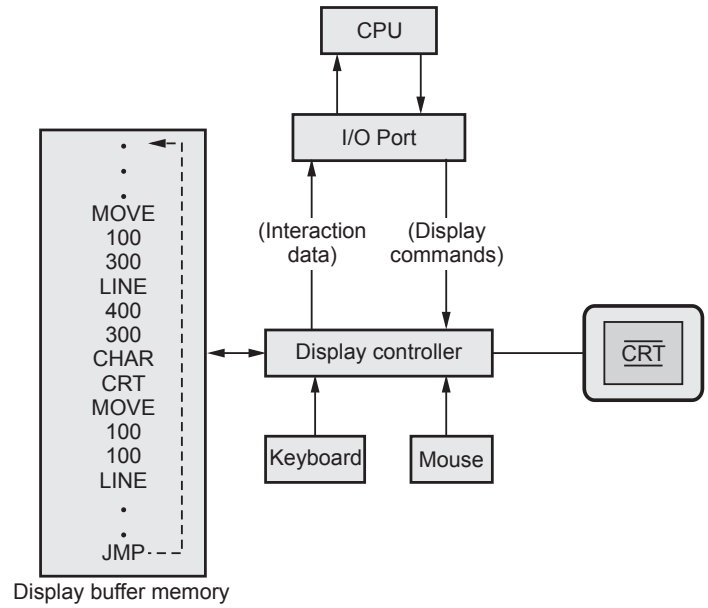


Fig. 1.2.3 Architecture of a vector display

In vector displays beam is deflected from end point to end point, hence this technique is also called **random scan**. We know as beam, strikes phosphor it emits light. But phosphor light decays after few milliseconds and therefore it is necessary to repeat through the display list to refresh the phosphor at least 30 times per second to avoid flicker. As display buffer is used to store display list and it is used for refreshing, the display buffer memory is also called **refresh buffer**.

1.2.3 Raster Scan Display and Frame Buffer

The Fig. 1.2.4 shows the architecture of a raster display. It consists of display controller, central processing unit (CPU), video controller, refresh buffer, keyboard, mouse and the CRT.

As shown in the Fig. 1.2.4, the display image is stored in the form of 1s and 0s in the refresh buffer. The video controller reads this refresh buffer and produces the actual image on the screen. It does this by scanning one scan line at a time, from top to bottom and then back to the top, as shown in the Fig. 1.2.4.

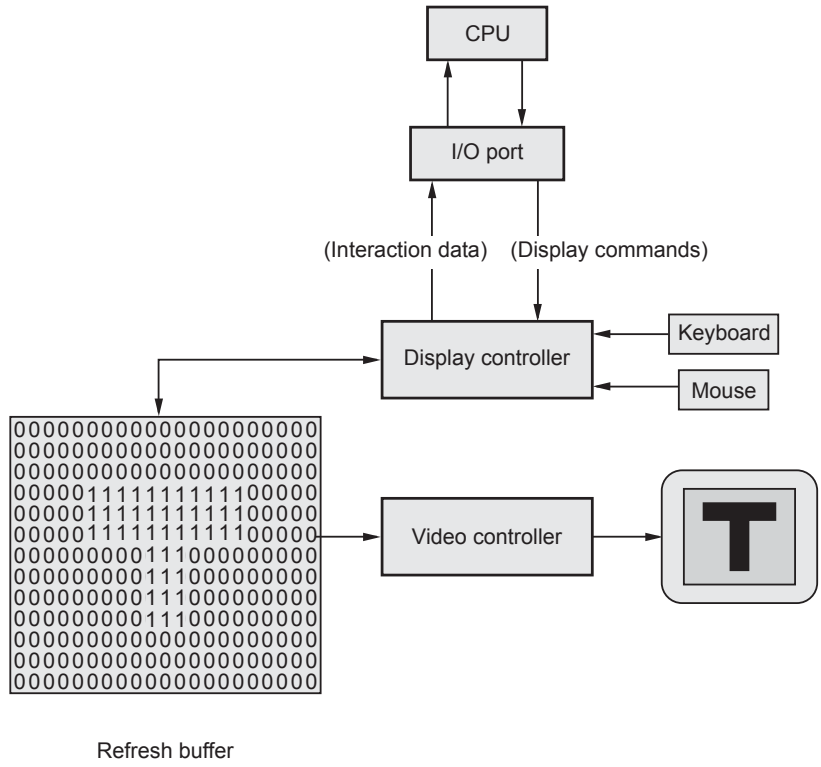


Fig. 1.2.4 Architecture of a raster display

Raster scan is the most common method of displaying images on the CRT screen. In this method, the horizontal and vertical deflection signals are generated to move the beam all over the screen in a pattern shown in the Fig. 1.2.5.

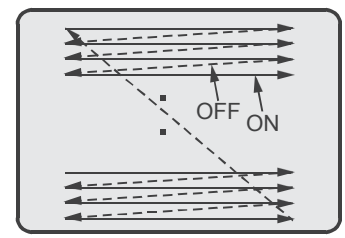


Fig. 1.2.5 Raster scan CRT

Here, the beam is swept back and forth from the left to the right across the screen. When the beam is moved from the left to the right, it is ON. The beam is OFF, when it is moved from the right to the left as shown by dotted line in Fig. 1.2.5.

When the beam reaches the bottom of the screen, it is made OFF and rapidly retraced back to the top left to start again. A display produced in this way is called **raster scan display**. Raster scanning process is similar to reading different lines on the page of a book. After completion of scanning of one line, the electron beam flies back to the start of next line and process repeats. In the raster scan display, the screen image is maintained by repeatedly scanning the same image. This process is known as **refreshing of** screen.

Frame Buffer

In raster scan displays a special area of memory is dedicated to graphics only. This memory area is called **frame buffer**. It holds the set of intensity values for all the screen points. The stored intensity values are retrieved from frame buffer and displayed on the screen one row (scan line) at a time. Each screen point is referred to as a **pixel** or **pel** (shortened forms of picture element). Each pixel on the screen can be specified by its row and column number. Thus by specifying row and column number we can specify the **pixel position** on the screen.

Intensity range for pixel positions depends on the capability of the raster system. It can be a simple black and white system or colour system. In a simple black and white system, each pixel position is either on or off, so only one bit per pixel is needed to control the intensity of the pixel positions. Additional bits are required when colour and intensity variations can be displayed. Upto 24 bits per pixel are included in high quality display systems, which can require several megabytes of storage space for the frame buffer. On a black and white system with one bit per pixel, the frame buffer is commonly called a **bitmap**. For systems with multiple bits per pixel, the frame buffer is often referred to as a **pixmap**.

Sr. No.	Vector (Random) Scan Display	Raster Scan Display
1.	In vector scan display the beam is moved between the end points of the graphics primitives.	In raster scan display the beam is moved all over the screen one scan line at a time, from top to bottom and then back to top.
2.	Vector display flickers when the number of primitives in the buffer becomes too large.	In raster display, the refresh process is independent of the complexity of the image.
3.	Scan conversion is not required.	Graphics primitives are specified in terms of their endpoints and must be scan converted into their corresponding pixels in the frame buffer.
4.	Scan conversion hardware is not required.	Because each primitive must be scan-converted, real time dynamics is far more computational and requires separate scan conversion hardware.

5.	Vector display draws a continuous and smooth lines.	Raster display can display mathematically smooth lines, polygons, and boundaries of curved primitives only by approximating them with pixels on the raster grid.
6.	Cost is more.	Cost is low.
7.	Vector display only draws lines and characters.	Raster display has ability to display areas filled with solid colours or patterns.

Table 1.2.1

Example 1.2.1 *How much memory is needed for the frame butter to store a 640 × 400 display 16 gray levels ?*

Solution : $2^4 = 16$. Therefore, 4-bits are required to store 16 gray levels.

$\therefore$ Frame buffer memory = $640 \times 400 \times 4$

= 1024000 bits = 128 kbytes

Example 1.2.2 *Find the refresh rate of a 512 × 512 frame buffer, if the access time for each pixel is 200 nanoseconds.* **SPPU : Dec.-10, Marks 4**

Solution :

$$\text{Refresh rate} = \frac{1}{\text{Access time for each pixel} \times \text{Number of pixels information in frame buffer}}$$

$$= \frac{1}{200 \times 10^{-9} \times 512 \times 512} = 19.0734 \text{ frames /sec.}$$

Example 1.2.3 *Find the amount of memory required by an 8 plane frame buffer each of red, green, blue having resolution of 1024 × 768.* **SPPU : Dec.-12, Marks 4**

Solution : For 8 plane frame buffer each of red, green and blue requires 24-bits to represent per pixel.

$\therefore$ Amount of memory required = $24 \times 1024 \times 768$

= 18874368 bits

= 2359296 bytes

Example 1.2.4 *How long would it take to load a 1280 by 1024 frame buffer with 12 bits per pixel if transfer rate is 1 Mbps ?*

Solution :

$$\text{Time required to load frame buffer} = \frac{1280 \times 1024 \times 12}{10^6} = \mathbf{15.72\ 64\ sec}$$

1.2.4 Colour CRT Monitors

A CRT monitor displays colour pictures by using a combination of phosphors that emit different-coloured light. It generates a range of colours by combining the emitted light from the different phosphors. There are two basic techniques used for producing colour displays :

- Beam-penetration technique and
- Shadow-mask technique

Beam-penetration Technique

This technique is used with random-scan monitors. In this technique, the inside of CRT screen is coated with two layers of phosphor, usually red and green. The displayed colour depends on how far the electron beam penetrates into the phosphor layers. The outer layer is of red phosphor and inner layer is of green phosphor. A beam of slow electrons excites only the outer red layer. A beam of very fast electrons penetrates through the red layer and excites the inner green layer. At intermediate beam speeds, combinations of red and green light are emitted and two additional colours, orange and yellow displayed. The beam acceleration voltage controls the speed of the electrons and hence the screen colour at any point on the screen.

Merits and Demerits

- It is an inexpensive technique to produce colour in random scan monitors.
- It can display only four colours.
- The quality of picture produced by this technique is not as good as compared to other techniques.

Shadow Mask Technique

The shadow mask technique produces a much wider range of colours than the beam penetration technique. Hence this technique is commonly used in raster-scan displays including colour TV. In a shadow mask technique, CRT has three phosphor colour dots at each pixel position. One phosphor dot emits a red light, another emits a green light, and the third emits a blue light. The Fig. 1.2.6 shows the shadow mask CRT. It has three

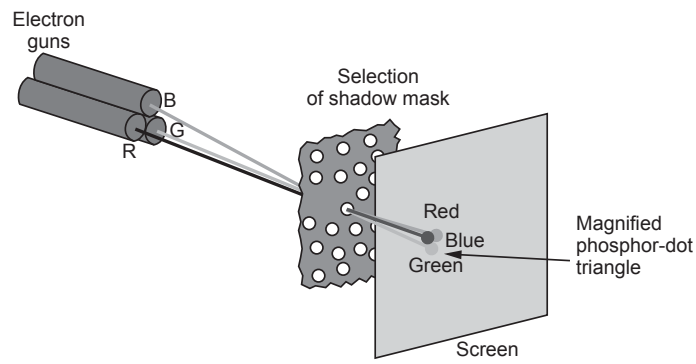


Fig. 1.2.6

electron guns, one for each colour dot, and a shadow mask grid just behind the phosphor coated screen.

The shadow mask grid consists of series of holes aligned with the phosphor dot pattern. As shown in the Fig. 1.2.6, three electron beams are deflected and focused as a group onto the shadow mask and when they pass through a hole in the shadow mask, they excite a dot triangle. A dot triangle consists of three small phosphor dots of red, green and blue colour. These phosphor dots are arranged so that each electron beam can activate only its corresponding colour dot when it passes through the shadow mask. A dot triangle when activated appears as a small dot on the screen which has colour of combination of three small dots in the dot triangle. By varying the intensity of the three electron beams we can obtain different colours in the shadow mask CRT.

1.2.5 Direct-View Storage Tubes

We know that, in raster scan display we do refreshing of the screen to maintain a screen image. The direct-view storage tubes give the alternative method of maintaining the screen image. A Direct-View Storage Tube (DVST) uses the storage grid which stores the picture information as a charge distribution just behind the phosphor-coated screen.

The Fig. 1.2.7 shows the general arrangement of the DVST. It consists of two electron guns : a primary gun and a flood gun. A primary gun stores the picture pattern and the flood gun maintains the picture display. A primary gun produces high speed electrons which strike on the storage grid to draw the picture pattern. As electron beam strikes on the storage grid with high speed, it knocks out electrons from the storage grid keeping the net positive charge. The knocked out electrons are attracted towards the collector. The net positive charge on the storage grid is nothing but the picture pattern. The continuous low speed electrons from flood gun pass through the control grid and are attracted to the positive charged areas of the storage grid. The low speed electrons then penetrate the storage grid and strike the phosphor coating without affecting the positive charge pattern on the storage grid. During this process the collector just behind the storage grid smooths out the flow of flood electrons.

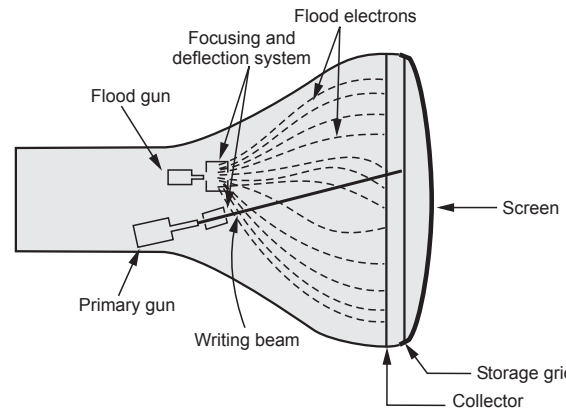


Fig. 1.2.7 Arrangement of the DVST

Advantages of DVST

1. Refreshing of CRT is not required.
2. Because no refreshing is required, very complex pictures can be displayed at very high resolution without flicker.
3. It has flat screen.

Disadvantages of DVST

1. They do not display colours and are available with single level of line intensity.
2. Erasing requires removal of charge on the storage grid. Thus erasing and redrawing process takes several seconds.
3. Selective or part erasing of screen is not possible.
4. Erasing of screen produces unpleasant flash over the entire screen surface which prevents its use of dynamic graphics applications.
5. It has poor contrast as a result of the comparatively low accelerating potential applied to the flood electrons.
6. The performance of DVST is some what inferior to the refresh CRT.

1.2.6 Flat Panel Displays

The term flat-panel display refers to a class of video devices that have reduced volume, weight, and power requirements compared to a CRT. The important feature of flat-panel display is that they are thinner than CRTs. There are two types of flat panel displays : emissive displays and nonemissive displays.

Emissive displays : They convert electrical energy into light energy. Plasma panels, thin-film electro luminescent displays, and light emitting diodes are examples of emissive displays.

Nonemissive displays : They use optical effects to convert sunlight or light from some other source into graphics patterns. Liquid crystal display is an example of nonemissive flat panel display.

1.2.7 Plasma Panel Display

Plasma panel display writes images on the display surface point by point, each point remains bright after it has been intensified. This makes the plasma panel functionally very similar to the DVST even though its construction is markedly different.

The Fig. 1.2.8 shows the construction of plasma panel display. It consists of two plates of glass with thin, closely spaced gold electrodes. The gold electrodes are attached to the inner faces and covered with a dielectric material. These are attached as a series of vertical conducting ribbons on one glass plate, and a set of horizontal ribbons to the other glass plate. The space between two glass plates is filled with neon-based gas and

sealed. By applying voltages between the electrodes the gas within the panel is made to behave as if it were divided into tiny cells, each one independent of its neighbours. These independent cells are made to glow by placing a firing voltage of about 120 volts across it by means of the electrodes. The glow can be sustained by maintaining a high frequency alternating voltage of about 90 volts across the cell. Due to this refreshing is not required.

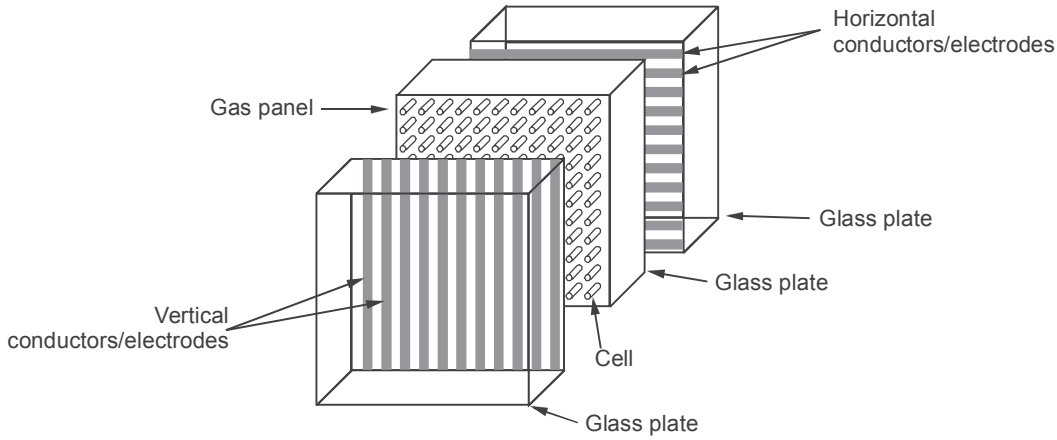


Fig. 1.2.8 Construction of plasma panel display

Advantages

1. Refreshing is not required.
2. Produces a very steady image, totally free of flicker.
3. Less bulky than a CRT.
4. Allows selective writing and selective erasing, at speed of about 20 μsec per cell.
5. It has the flat screen and is transparent, so the displayed image can be superimposed with pictures from slides or other media projected through the rear panel.

Disadvantages

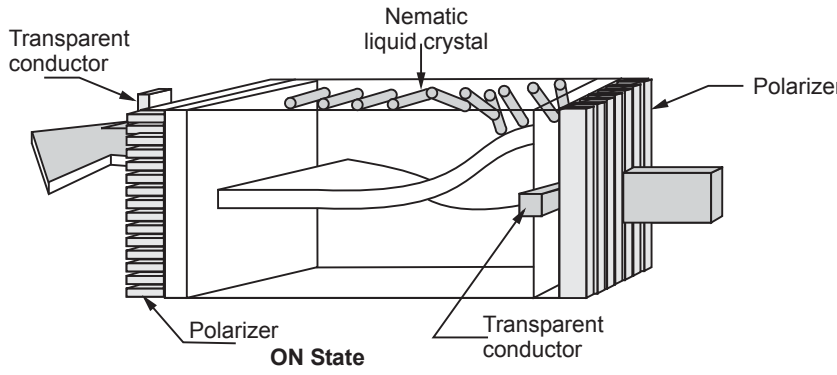
1. Relatively poor resolution of about 60 dots per inch.
2. It requires complex addressing and wiring.
3. Costlier than the CRTs.

1.2.8 Liquid Crystal Monitors

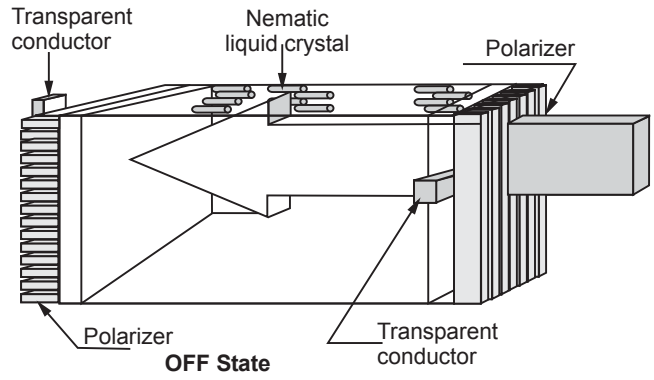
The term liquid crystal refers to the fact that these compounds have a crystalline arrangement of molecules, yet they flow like a liquid. Flat panel displays commonly use nematic (thread like) liquid-crystal compounds that tend to keep the long axes of the rod- shaped molecules aligned.

Two glass plates, each containing a light polarizer at right angles to the other plate sandwich the liquid-crystal material. Rows of horizontal transparent conductors are built into one glass plate, and columns of vertical conductors are put into the other plate. The intersection of two conductors defines a pixel position. In the ON state, polarized light passing through material is twisted so that it will pass through the opposite polarizer. It is then reflected back to the viewer. To turn OFF the pixel, we apply a voltage to the two intersecting conductors to align the molecules so that the light is not twisted as shown in the Fig. 1.2.9. This type of flat panel device is referred to as a passive matrix LCD.

In **active matrix LCD**, transistors are used at each (x, y) grid point. Use of transistors cause the crystal to change their state quickly and also to control the degree to which the state has been changed. These two properties allow LCDs to be used in miniature television sets with continuous-tone images.



(a) Field effect display 'ON State'



(b) Field effect display 'OFF State'

Fig. 1.2.9

The transistor can also serve as a memory for the state of a cell and can hold the cell in that state until it is changed. Thus use of transistor make cell on for all the time giving brighter display than it would be if it had to be refreshed periodically.

Advantages of LCD Displays

- Low cost
- Low weight
- Small size
- Low power consumption

1.2.9 Three-Dimensional Viewing Devices

Graphics monitors for the display of three-dimensional scenes employ the varying focal properties of a vibrating mirror to cause sequentially generated image components to appear 3-D in space. The Fig. 1.2.10 shows such three-dimensional display system. The mirror is constructed as a circular reflective plate having substantial stiffness and resilience. As the mirror vibrates, it changes focal length such that changes in the focal length are proportional to the depth of points in a scene. These vibrations are synchronized with the display of the object on a CRT. Due to this each point on the object is reflected from the mirror into a spatial position corresponding to the distance of that point from the specified viewing position. As a result we can view the object or scene from different sides.

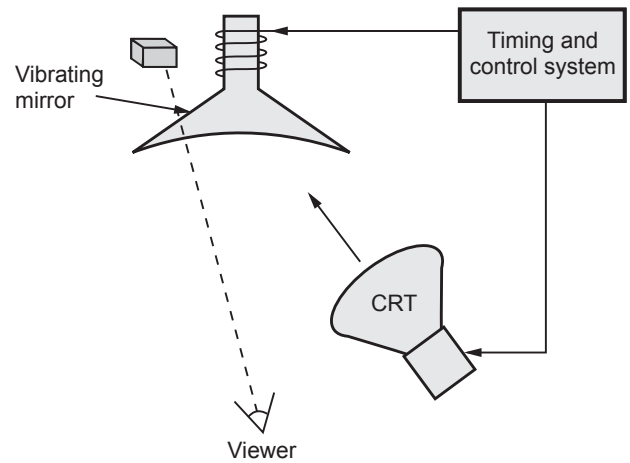


Fig. 1.2.10 Three-dimensional display system

Such a three-dimensional systems are very useful in medical applications such as ultrasonography and CAT scan to analyze data. They are also useful in three dimensional simulation systems and in geological applications to analyze topological and seismic data.

Stereoscopic and Virtual Reality Systems

Creating stereoscopic views is an another technique of viewing three-dimensional objects. The view obtained by such technique is not a true 3D view, but it does provide a 3D effect by presenting a different view to each eye of an observer so that scenes do appear to have depth.

To obtain a stereoscopic projection we have to obtain two views of scene generated from a viewing direction corresponding to each eye (left and right). Now looking

simultaneously at the left view with the left eye and the right view with right eye, two views are merged into a single image providing necessary depth to appear as an 3D image. Two views of a scene can be obtained using computer or we can use a stereo camera pair to photograph scene.

We can produce stereoscopic effect in raster system by :

- Displaying each of the two views on alternate refresh cycles
- And then viewing screen through glasses with each lens designed to act as a rapidly alternating shutter that is synchronized to block out one of the views.

The stereoscopic viewing plays important role in the virtual reality systems. In such systems, the viewer can step into stereoscopic scenes and interact with the environment. Usually, a headset containing an optical system is used to generate the stereoscopic views.

1.2.10 Persistence, Resolution and Aspect Ratio

Persistence : The major difference between phosphors is their persistence. It decides how long they continue to emit light after the electron beam is removed. Persistence is defined as the time it takes the emitted light from the screen to decay to one-tenth of its original intensity. Lower persistence phosphors require higher refreshing rates to maintain a picture on the screen without flicker. However it is useful for displaying animations. On the other hand higher persistence phosphors are useful for displaying static and highly complex pictures.

Resolution : Resolution indicates the maximum number of points that can be displayed without overlap on the CRT. It is defined as the number of points per centimeter that can be plotted horizontally and vertically.

Resolution depends on the type of phosphor, the intensity to be displayed and the focusing and deflection systems used in the CRT.

Aspect Ratio : It is the ratio of vertical points to horizontal points to produce equal length lines in both directions on the screen. An aspect ratio of 4/5 means that a vertical line plotted with four points has the same length as a horizontal line plotted with five points.

Example 1.2.1 *If image of size 1024 × 800 needs to resize to one that has 640 pixels width with the same aspect ratio, what would be the height of the resized image ?*

Solution : Height of the resized image in pixels = $\frac{800}{1024} \times 640 =$ **500**

1.2.11 Applications of Large Screen Displays

1. **Education :** Many institutes use large screen displays to give effective audio-visual training in the class-room.
2. **Entertainment :** Large screen displays are used to watch live matches in the hotels or in the community halls.
3. **Sports ground :** Large screen displays are used to show current scores in the stadiums.
4. **Advertisement :** Large screen displays are used for advertising purpose in many public places.
5. **Railway, airway and roadway transport :** Large screen displays are used at the railway stations, airports and bus stands to provide the schedule information to the travelers.
6. **Seminars, presentations and video-conferencing :** Large screen displays are used during presentations, seminars and video-conferencing.
7. **Space stations :** Large screen displays are used in space stations to display the satellite images of the planets. They are used as one of the visual communication medium for communicating with the astronauts.

Plasma-panel and flat panel displays support large screen displays.

Review Questions

1. *Explain the working of cathode ray tube with a diagram.*
2. *Define random scan and raster scan displays.* **SPPU : May-14, Marks 5**
3. *Compare raster scan and vector scan displays.*
4. *Write a note on raster scan.*
5. *Define the following term : Frame buffer.* **SPPU : Dec.-06,10,11,12,15, May-09,13, Marks 2**
6. *Explain shadow mask technique and explain how does it differ from beam penetration technique ?*
7. *Write short notes on beam-penetration technique and shadow mask technique.*
8. *Explain Direct-View Storage Tubes (DVST). How persistence characteristics of phosphor affect on refresh rate of system ?*
9. *List merit and demerit of DVST.*
10. *Write a short note on a) Flat panel display b) Plasma panel display.*
11. *List merit and demerit of plasma panel display.*
12. *List the important characteristics of video display devices.*
13. *Write a note on liquid crystal displays.*
14. *Write a short note on three-dimensional viewing devices.*
15. *Explain important characteristics of video display derives.*
16. *List the applications of large screen displays.*
17. *Explain following terms*
    1. Persistence 2. Resolution 3. Aspect ratio. **SPPU : May-05, 14, 15, Dec.-15, Marks 4**
18. *Define the following terms : 1) Persistence 2) Frame buffer 3) Resolution 4) Aspect ratio.*

1.3 Applications of Computer Graphics

SPPU : Dec.-14, May-16

The use of computer graphics is wide spread. It is used in various areas such as industry, business, government organizations, education, entertainment and most recently the home. Let us discuss the representative uses of computer graphics in brief.

- **User Interfaces :** User friendliness is one of the main factors underlying the success and popularity of any system. It is now a well established fact that graphical interfaces provide an attractive and easy interaction between users and computers. The built-in graphics provided with user interfaces use visual control items such as buttons, menus, icons, scroll bar etc, which allows user to interact with computer only by mouse-click. Typing is necessary only to input text to be stored and manipulated.
- **Plotting of Graphics and Chart :** In industry, business, government and educational organizations, computer graphics is most commonly used to create 2D and 3D graphs of mathematical, physical and economic functions in form of histograms, bars and pie-charts. These graphs and charts are very useful for decision making.
- **Office Automation and Desktop Publishing :** The desktop publishing on personal computers allow the use of graphics for the creation and dissemination of information. Many organizations do the in-house creation and printing of documents. The desktop publishing allows user to create documents which contain text, tables, graphs and other forms of drawn or scanned images or pictures. This is one approach towards the office automation.
- **Computer-aided Drafting and Design :** The computer-aided drafting uses graphics to design components and systems electrical, mechanical, electro-mechanical and electronic devices such as automobile bodies, structures of building, airplane, ships, very large-scale integrated (VLSI) chips, optical systems and computer networks.
- **Simulation and Animation :** Use of graphics in simulation makes mathematic models and mechanical systems more realistic and easy to study. The interactive graphics supported by animation software proved their use in production of animated movies and cartoons films.
- **Art and Commerce :** There is a lot of development in the tools provided by computer graphics. This allows user to create artistic pictures which express messages and attract attentions. Such pictures are very useful in advertising.
- **Process Control :** By the use of computer now it is possible to control various processes in the industry from a remote control room. In such cases, process systems and processing parameters are shown on the computer with graphic

symbols and identifications. This makes it easy for operator to monitor and control various processing parameters at a time.

- **Cartography :** Computer graphics is also used to represent geographic maps, weather maps, oceanographic charts, contour maps, population density maps and so on.
- **Education and Training :** Computer graphics can be used to generate models of physical, financial and economic systems. These models can be used as educational aids. Models of physical systems, physiological systems, population trends, or equipment, such as colour-coded diagram can help trainees to understand the operation of the system.
- **Image Processing :** In computer graphics, a computer is used to create pictures. Image processing, on the other hands, applies techniques to modify or interpret existing pictures such as photographs and scanned images. Image processing and computer graphics are typically combined in many applications such as to model and study physical functions, to design artificial limbs, and to plan and practice surgery. Image processing techniques are most commonly used for picture enhancements to analyze satellite photos, X-ray photography and so on.

Review Questions

1. *What is computer graphics ? State the applications of computer graphics.*

SPPU : Dec.-14, Marks 6

2. *Write and explain any four state of the applications of computer graphics.*

SPPU : May-16, Marks 4

❑❑❑

UNIT - I

2 Introduction to OpenGL

Syllabus

OpenGL architecture, primitives and attributes, simple modelling and rendering of two- and three-dimensional geometric objects, GLUT, interaction, events and call-backs picking. ***(Simple Interaction with the Mouse and Keyboard)***

Contents

2.1 OpenGL Architecture

- OpenGL (Open Graphics Library) is a cross-language, multi-platform application programming interface (API) for rendering 2D and 3D computer graphics.
- The API is typically used to interact with a Graphics Processing Unit (GPU), to achieve hardware-accelerated rendering.
- A basic library of functions is provided in OpenGL for specifying graphics primitives, attributes, geometric transformations, viewing transformations and many other operations.
- OpenGL is designed to be hardware independent, therefore many operations, such as input and output routines, are not included in the basic library. However, input and output routines and many additional functions are available in auxiliary libraries that have been developed for OpenGL programs.

2.1.1 Basic OpenGL Syntax

- Function names in the **OpenGL basic library** (also called the **OpenGL core library**) are prefixed with **gl** and each component word within a function name has its first letter capitalized. The following examples illustrate this naming convention.
  `glBegin, glClear, glCopyPixels, glPolygonMode`
- Some functions in OpenGl require that one (or more) of their arguments be assigned a symbolic constant specifying, for instance, a parameter name, a value for a parameter or a particular mode. All such constants begin with the uppercase letters GL. In addition, component words within a constant name are written in capital letters and the underscore ( ) is used as a separator between all component words in the name. Following are a few examples of the several hundred symbolic constants available for use with OpenGL functions.
  `GL_2D, GL_RGB, GL_CCW, GL_POLYGON, GL_AMBIENT_AND_DIFFUSE`
- The OpenGL functions **uses specific data types**. For example, an OpenGL function parameter might use a value that is specified as a 32-bit integer. But the size of an integer specification can be different on different machines.
- Special built-in, data-type supported by OpenGL are : **GLbyte**, **GLshort**, **GLint**, **GLfloat**, **GLdouble**, **GLboolean**. Each data-type name begins with the capital letters GL and the remainder of the name is a standard data-type designation, written in lower-case letters.
- It is also possible to assign values using an array that lists a set of data values for some arguments of OpenGL functions. These values are specified as a pointer to an array, rather than specifying each element of the list explicitly as a parameter argument.

2.1.2 Related Libraries

- In addition to the OpenGL basic (core) library, there are a number of associated libraries for handling special operations.

Libraries	Supported Operations
OpenGL Utility (GLU)	Provides routines for setting upviewing and projection matrices, describing complex objects with line and polygon approximations, displaying quadrics and B-splines using linear approximations, processing the surface-rendering operations and other complex tasks. GLU function names start with the prefix **glu**.
Object Oriented Toolkit (Open Inventor)	Provides routines and predefined object shapes for interactive three-dimensional applications. This toolkit is written in C++.

- To create a graphics display using OpenGL, we first need to set up a display window on our video screen. This is simply the rectangular area of the screen in which our picture will be displayed.
- We cannot create the display window directly with the basic OpenGL functions, because of following two reasons :
  - The basic OpenGL functions contains only device independent graphics functions and
  - Window-management operations are machine depend.
- However, there are several window-system libraries that support OpenGL functions for a variety of machines.

Extension/Interface	Supported Operations
(GLX)	OpenGL Extension to the X Window System. Provides a set of routines that are prefixed with the letters **glX**.
Apple GL (AGL)	Apple systems can use the **Apple GL** (AGL) interface for window-management operations. Function names for this library are prefixed with **agl**.
WGL	Provide a Windows-to-OpenGL interface for Microsoft Windows systems. These routines are prefixed with the letters **wgl**.
Presentation Manager to OpenGL (PGL)	Provides interface for the IBM OS/2, which uses the prefix **pgl** for the library routines.
OpenGL Utility Toolkit (GLUT)	Provides a library of functions for interacting with any screen-windowing system. The GLUT library functions are prefixed with **glut**, and this library also contains methods for describing and rendering quadric curves and surfaces.

2.1.3 Header Files

- For graphics programs we need to include the header file for the OpenGL core library.
- For most applications we will also need GLU. For instance, with Microsoft Windows, we need to include the header file (windows.h) for the window system.
- This header file must be listed before the OpenGL and GLU header files because it contains macros needed by the Microsoft Windows version of the OpenGL libraries. So the source file in this case would begin with

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
```

- However, if we use GLUT to handle the window-managing operations, we do not need to include gl.h and glu.h because GLUT ensures that these will be included correctly. Thus, we can replace the header files for OpenGL and GLU with

```
#include <GL/glut.h>
```

- We could include gl.h and glu.h as well, but doing so would be redundant and could affect program portability.
- In addition, we will often need to include header files that are required by the C++ code. For example,

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

2.1.4 Display-Window Management using GLUT

- To get started, we can consider a simplified, minimal number of operations for displaying a picture. Since we are using the OpenGL Utility Toolkit, our first step is to initialize GLUT.
- We perform the GLUT initialization with the statement

```
glutInit (&argc, argv);
```

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

```
glutCreateWindow ("An Example OpenGL Program");
```

- Then we need to specify what the display window is to contain. For this, we create a picture using OpenGL functions and pass the picture definition to the GLUT routine **glutDisplayFunc**, which assigns our picture to the display window.
- As an example, suppose we have the OpenGL code for describing a line segment in a procedure called lineSegment. Then the following function call passes the line-segment description to the display window.

```
glutDisplayFunc (lineSegment);
```

- But the display window is not yet on the screen. We need one more GLUT function to complete the window-processing operations. After execution of the following statement, all display windows that we have created, including their graphic content, are now activated.
  glutMainLoop ( );
- This function must be the last one in our program. It displays the initial graphics and puts the program into an infinite loop that checks for input from devices such as a mouse or keyboard.
- In our first program, the display window that we created will be in some default location and size, we can set these parameters using additional GLUT functions.

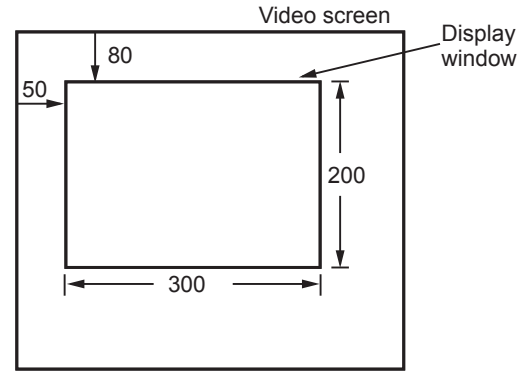


Fig. 2.1.1 A 300 by 200 display window at position (50, 80) relative to the top-left corner of the video display

- We use the **glutInitWindowPosition** function to give an initial location for the top left corner of the display window.
  glutInitWindowPosition (50, 80);
- Similarly, the glutInitWindowSize function is used to set the initial pixel width and height of the display window.
  glutInitWindowSize (300, 200);
- After the display window is on the screen, we can reposition and resize it.
- We can also set a number of other options for the display window, such as buffering and a choice of color modes, with the **glutInitDisplayMode** function. Arguments for this routine are assigned symbolic GLUT constants.
- For example,
  glutInitDisplay**Mode** (GLUT_**SINGLE** | GLUT_RGB);
- The above command specifies that a single refresh buffer is to be used for the display window and that the RGB (red, green, blue) color mode is to be used for selecting color values.
- The values of the constants passed to this function are combined using a logical **or** operation. Actually, single buffering and RGB color mode are the default options.

2.1.5 Format of OpenGL Command

The Fig. 2.1.2 shows the format of OpenGL command.

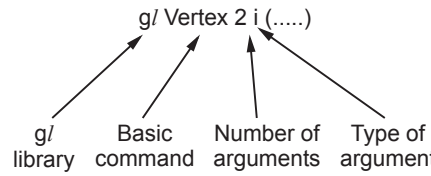


Fig. 2.1.2 Format of OpenGL command

2.1.6 Vertex Function

- We can represent a point in the plane z = 0 as p = (x, y, 0) in the three dimensional world, or as p = (x, y) in the two dimensional plane.
- OpenGL allows us to use both the representations.
- A vertex is a position in space. Single vertex defines a point, two vertices define a line segment, three vertices define a triangle or a circle, four vertices define a quadrilateral and so on.
- In OpenGL, the general syntax for vertex function is

```
glvertex * ( )
```

where the * can be interpreted as either two or three characters of the form nt or ntv, where

  - n : The number of dimensions (2, 3 or 4)
  - t : Denotes data type
    - i : integer
    - f : float
    - d : double
    - v : indicates that the variables are specified through a pointer to an array, rather than through an argument list.

2.1.6.1 Multiple Forms of Vertex Functions

glvertex2i (GLint xi, GLint yi) : Function works in two dimension with integers.

glvertex3f (GLfloat x, GLfloat y, GLfloat z) : Function works in three dimension with floating point numbers.

glvertex3fv (vertex) : Function works in three dimension with floating point numbers.

Here variables are specified through a pointer to predefined array. For example, GLfloat vertex [3].

2.1.7 A Simple OpenGL Program

Fig. 2.1.3 shows a simple OpenGL program. Although this program does nothing useful, it defines a pattern that is used by most OpenGL programs. The line numbers are not part of the program, but are used in the explanation below.

```
1   #include <GL/glut.h>
2
3   void display (void)
4   {
5   glClear(GL_COLOR_BUFFER_BIT);
6   }
7
8   void init (void)
9   {
10  }
11
12  int main (int argc, char *argv[])
13  {
14  glutInit(&argc, argv);
15  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
16  glutInitWindowSize(800, 600);
17  glutInitWindowPosition(100, 50);
18  glutCreateWindow("My first openGL program");
19  init();
20  glutDisplayFunc(display);
21  glutMainLoop();
22  return 0;
23  }
```

Fig. 2.1.3 The basic OpenGL program

Line 1. Every OpenGL program should include GL/glut.h. The file glut.h includes glu.h, gl.h and other header files required by GLUT.

Line 3. You must write a function that displays the graphical objects in your model. The function shown here clears the screen but does not display anything. Your program does not call this function explicitly, but it will be called at appropriate times by OpenGL.

Line 5. It is usually a good idea to clear various buffers before starting to draw new objects. The colour buffer, cleared by this call, is where the image of your model is actually stored.

Line 8. It is a good idea to put "standard initialization" (the glut... calls) in the main program and application dependent initialization in a function with a name like init() or myInit(). We follow this convention in these text because it is convenient to give different bodies for init without having to explain where to put the initialization statements. In this program, the function init() is defined in lines 8 through 10, does nothing, and is invoked at line 19.

Line 14. The function **glutInit()** initializes the OpenGL library. It is conventional to pass the command line arguments to this function because it may use some of them. You will probably not need this feature.

Line 15. The function **glutInitDisplayMode()** initializes the display. Its argument is a bit string. Deleting this line would not affect the execution of the program, because the arguments shown are the default values.

Line 16. This call requests a graphics window 800 pixels wide and 600 pixels high. If this line was omitted, GLUT would use the window manager's default values for the window's size.

Line 17. This call says that the left side of the graphics window should be 100 pixels from the left of the screen and the top of the graphics window should be 50 pixels below the top of the screen. Note that the Y value is measured from the **top** of the screen. If this line was omitted, GLUT would use the window manager's default values for the window's position.

Line 18. This call creates the window using the settings given previously. The text in the argument becomes the title of the window. The window does not actually appear on the screen until **glutMainLoop()** has been called.

Line 19. This is a good place to perform any additional initialization that is needed (see above). Some initialization, such as calls to glEnable(), must come **after** the call to **glutCreateWindow().**

Line 20. This call **registers** the callback function display(). After executing the call, OpenGL knows what function to call to display your model in the window. Section 2.4 describes callback functions.

Line 21. The last statement of an OpenGL program calls **glutMainLoop().** This function processes events until the program is terminated. Well-written programs should provide the user with an easy way of stopping the program, for example by selecting Quit from a menu or by pressing the esc key.

Drawing Objects

All that remains is to describe additions to Fig. 2.1.3. The first one that we will consider is an improved display() function. Fig. 2.1.4 shows a function that displays a line. The call glColor3f(1.0, 0.0, 0.0) specifies the colour of the line as "maximum red, no green, no blue".

```
void display (void)
{
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0, 0.0, 0.0);
glBegin(GL_LINES);
        glVertex2f(-1.0, 0.0);
        glVertex2f(1.0, 0.0);
glEnd();
glFlush();
}
```

Fig. 2.1.4 Displaying a bright red line

The construction glBegin(mode); ...; glEnd(); is used to display groups of primitive objects. The value of mode determines the kind of object. In this case, GL_LINES tells OpenGL to expect one or more lines given as pairs of vertices.

In this case, the line goes from (–1, 0) to (1, 0), as specified by the two calls to **glVertex2f().** If we use the default viewing region, this line runs horizontally across the centre of the window. The call **glFlush()** forces previously executed OpenGL commands to begin execution.

The suffix "3f" indicates that **glColor3f()** requires three arguments of type **GLfloat**. Similarly, **glVertex2f()** requires two arguments of type GLfloat. The code between glBegin() and glEnd() is not restricted to gl calls. Fig. 2.1.5 shows a display function that uses a loop to draw 51 vertical yellow (red + green) lines across the window. Section 2.3 describes some of the other primitive objects available in OpenGL.

```
void display (void)
{
int i;
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0, 1.0, 0.0);
glBegin(GL_LINES);
for (i = -25; i <= 25; i++)
```

```
{
float x = i / 25.0;
glVertex2f(x, -1.0);
glVertex2f(x, 1.0);
}
glEnd();
}
```

Fig. 2.1.5 Drawing vertical yellow lines

The call glClear(GL_COLOR_BUFFER_BIT) clears the colour buffer to the background colour. You can set the background colour by executing

```
glClearColor(r, g, b, a);
```

with appropriate values of r, g, b and a during initialization. Each argument is a floating point number in the range [0, 1] specifying the amount of a colour (red, green, blue) or blending (alpha). The default values are all zero, giving a black background. To make the background blue, you could call

```
glClearColor(0.0, 0.0, 1.0, 0.0);
```

Blending is an advanced feature; unless you know how to use it, set the fourth argument of glClearColor to zero.

2.1.8 A Complete OpenGL Program

- For complete program we have to perform following additional task.
  - For the display window, we can choose a background color.
  - We need to construct a procedure that contains the appropriate OpenGL functions for the picture that we want to display.
- Using RGB color values, we set the background color for the display window to be white with the OpenGL function
  glClearColor (1.0, 1.0, 1.0, 0.0);
- The first three arguments in this function set each of the red, green and blue component colors to the value 1.0. Thus we get a white color for the display window. We can set each of the component colors to 0.0, to get a black background.
- The fourth parameter in the glClearColor function is called the **alpha value** for the specified color. One use for the alpha value is as a **"blending" parameter**. When we activate the OpenGL blending operations, alpha values can be used to determine the resulting color for two overlapping objects. An alpha value of 0.0 indicates a totally transparent object and an alpha value of 1.0 indicates an opaque object.

- At this point we are not using Blending operations, so the value of alpha is irrelevant. For now, we simply set alpha to 0.0.
- Although the glClearColor command assigns a color to the display window, it does not put the display window on the screen. To get the assigned window color displayed, we need to invoke the following OpenGL function.
  glClear (GL_COLOR_BUFFER_BIT);
- The argument **GL_COLOR_BUFFER_BIT** is an OpenGL symbolic constant specifying that it is the bit values in the color buffer (refresh buffer) that are to be set to the values indicated in the **glClearColor function**.
- In addition to setting the background color for the display window, we can choose a variety of color schemes for the objects we want to display in a scene.
- For example, the following function sets object color to be blue.

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

- The suffix 3f on the glColor function indicates that we are specifying the three RGB color components using floating-point (f) values. These values must be in the range from 0.0 to 1.0 and we have set red and green = 0.0 and blue = 1.0.
- In our first program, we simply display a two-dimensional line segment. To do this, we have to set the projection type (mode) and other viewing parameters to tell OpenGL how we want to "project" our picture onto the display window.
- We can set the projection type (mode) and other viewing parameters with following two functions.

```
glMatrixMode (GL_PROJECTION);
gluOrtho2D (0.0, 150.0, 0.0, 120.0);
```

- The above functions specify that an orthogonal projection is to be used to map the contents of a two-dimensional (2D) rectangular area of world coordinates to the screen and that the x-coordinate values within this rectangle range from 0.0 to 150.0 with y-coordinate values ranging from 0.0 to 120.0.
- Whatever objects we define within this world-coordinate rectangle will be shown within the display window. Anything outside this coordinate range will not be displayed. Therefore, the GLU function gluOrtho2D defines the coordinate reference frame with in the display window to be (0.0, 0.0) at the lower-left corner of the display window and (150.0, 120.0) at the upper-right window corner.
- Finally, we need to call the appropriate OpenGL routines to create the line segment. The following code defines a two-dimensional, straight-line segment with integer, Cartesian endpoint coordinates (110, 25) and (20, 100).

```
glBegin (GL_LINES);
glVertex2i (110, 25);
```

```
glVertex2i (20, 100);
glEnd ( );
```

- Let us see the entire program organized into three procedures. We place all initializations and related one-time parameter settings in procedure in it.
- Our geometric description of the "picture" we want to display is in procedure lineSegment, which is the procedure that will be referenced by the GLUT function **glutDisplayFunc**.
- The main procedure contains the GLUT functions for setting up the display window and getting the line segment onto the screen.

```
#include <GL/glut.h>// Includes the header file
                    // depending on the system in use, this may change)
void init (void)
{
glClearColor (1.0, 1.0, 1.0, 0.0);          // Set display-window color to white
glMatrixMode (GL_PROJECTION);               // Set projection parameters
gluOrtho2D (0.0, 150.0, 0.0, 120.0);
}
void lineSegment (void)
{
glClear (GL_COLOR_BUFFER_BIT);              // Clear display window
glColor3f (0.0, 0.0, 1.0);                  // Set line segment color to blue
glBegin (GL_LINES);
glVertex2i (110, 25);                       // Specify line-segment geometry.
glVertex2i (20, 100);
glEnd ( );
glFlush ( );                                // Process all OpenGL routines as quickly as
possible.
}
void main (int argc, char** argv)
{
glutInit (&argc, argv);                                    //Initialize GLUT
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);   // Set display mode
glutInitWindowPosition (50, 80);            // Set top-left position of display-window
glutInitWindowSize (300, 200);              // Set width and height of display-window
glutCreateWindow ("Draw Line");             // Create display window
init ( );                                   // Execute initialization procedure.
glutDisplayFunc (lineSegment);              // Send graphics to display window.
glutMainLoop ( );                           // Display everything and wait.
}
```

2.1.9 OpenGL Data Types

Table 2.1.1 shows the data types supported by OpenGL.

Suffix	Data type	C Type	OpenGL Type
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating point	float	GLfloat, GLclampf
d	64-bit floating point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int	GLuint, GLenum, GLbitfield
	Nothing	void	GLvoid

Table 2.1.1 OpenGL data types

2.1.10 OpenGL Function Types

- An API for interfacing with graphics system can contain hundreds of individual functions. These functions are divided into seven major groups :

1. Primitive functions
2. Attribute functions
3. Viewing functions
4. Transformation functions
5. Input functions
6. Control functions
7. Query functions

- **Primitive functions** : These functions define the low-level objects or atomic entities that our system can display. Depending on the API, the primitives can include points, line segments, polygons, pixels, text, and various types of curves and surfaces.
- **Attribute functions** : These functions allow us to perform operations ranging from choosing the color with which we display a line segment, to picking a pattern with which to fill the inside of a polygon, to selecting a typeface for the titles on a graph.

- **Viewing functions** : These functions allow us to specify various views, although APIs differ in the degree of flexibility they provide in choosing a view.
- **Transformation functions** : These functions allow us to carry out transformations of objects, such as rotation, translation, and scaling. Providing the user with a set of transformation functions is one of the characteristics of a good API.
- **Input functions** : These functions allow us to deal with the diverse forms of input that characterize modern graphics systems. These functions include the functions to deal with devices such as keyboards, mice, and data tablets.
- **Control functions** : These functions enable us to communicate with the window system, to initialize our programs, and to deal with any errors that take place during the execution of our programs.
- **Query function**s : Within our applications we can often use other information within the API, including camera parameters or values in the frame buffer. A good API provides this information through a set of query functions.

Review Questions

1. *What is openGL ?*
2. *List the datatypes supported by openGL.*
3. *List the libraries supported by openGL and their supported operations.*
4. *Explain the display window management using GLUT.*
5. *Draw the format of OpenGL command.*
6. *Explain the vertex function with examples.*
7. *Explain the various types of function supported by OpenGL.*

2.2 Primitives and Attributes

- There are two types of primitives :
  - **Geometric primitives** and
  - **Image or raster primitives**

Geometric primitives

- Geometric primitives are specified in the problem domain and include points, line segments, polygons, curves, and surfaces.
- These primitives pass through a geometric pipeline, as shown in Figure 2.2.1, where they are subject to a series of geometric operations.

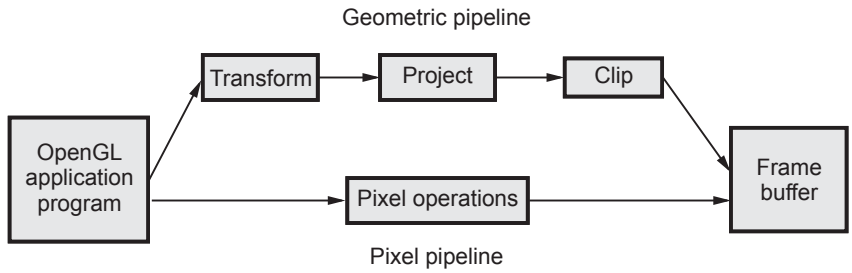


Fig. 2.2.1 Simplified OpenGL pipeline

- Series of geometric operations determine :
  - whether a primitive is visible,
  - where on the display it appears if it is visible, and
  - the rasterization of the primitive into pixels in the frame buffer.
- Because geometric primitives exist in a two- or three-dimensional space, they can be manipulated by operations such as rotation and translation.
- Geometric operations can be used as building blocks for other geometric objects.

Raster primitives

- Raster primitives, such as arrays of pixels, lack geometric properties and cannot be manipulated in space in the same way as geometric primitives.
- They pass through a separate parallel pipeline on their way to the frame buffer.

2.2.1 Basic OpenGL Primitives

- OpenGL supports several basic primitive types: points, lines, quadrilaterals, and general polygons. All of these primitives are specified using a sequence of vertices.
- Fig. 2.2.2 shows the basic primitive types, where the numbers indicate the order in which the vertices have been specified. Note that for the GL_LINES primitive only every second vertex causes a line segment to be drawn. Similarly, for the GL_TRIANGLES primitive, every third vertex causes a triangle to be drawn. Note that for the GL_TRIANGLE_STRIP and GL_TRIANGLE_FAN primitives, a new triangle is produced for every additional vertex. All of the closed primitives shown below are solid-filled, with the exception of GL_LINE_LOOP, which only draws lines connecting the vertices.
- OpenGL provides ten different primitive types for drawing points, lines, and polygons, as shown in Fig. 2.2.2.

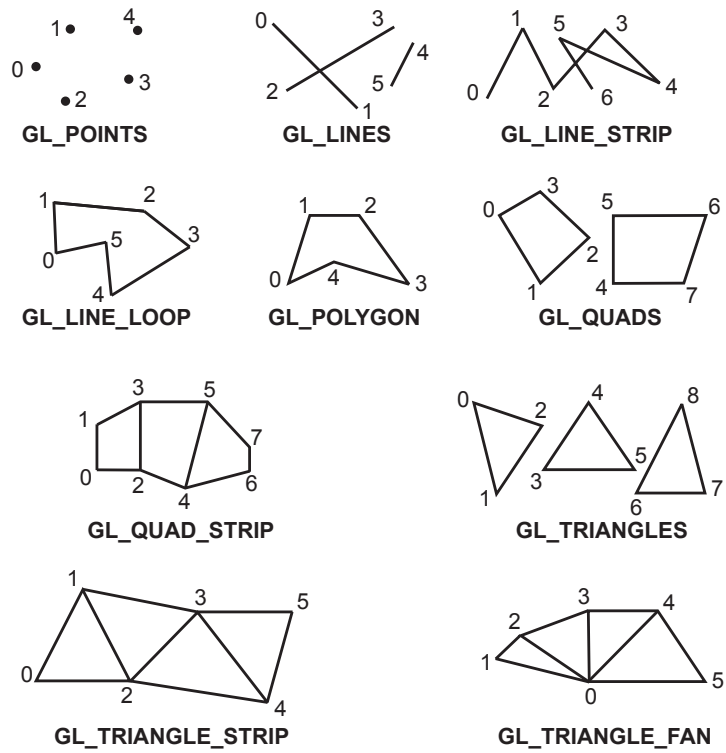


Fig. 2.2.2 OpenGL primitive types

2.2.1.1 GL_POINTS

- We use this primitive type to render mathematical points. OpenGL renders a point for each vertex specified.
  - Treats each vertex as a single point.
  - Vertex n defines a point n.
- **Example :**

```
glBegin(GL_POINTS);
glVertex2f(x1, y1);
glEnd();
```

2.2.1.2 GL_LINES

- We use this primitive to draw unconnected line segments. OpenGL draws a line segment for each group of two vertices. If the application specifies n vertices, OpenGL renders n/2 line segments. If n is odd, OpenGL ignores the final vertex.
  - Treats each pair of vertices as an independent line segment.
  - Vertices 2n–1 and 2n define a line n.
  - n/2 lines are drawn.

- **Example :**

```
glBegin(GL_LINES);
glVertex2f(x1, y1);
glVertex2f(x2, y2);
glEnd();
```

2.2.1.3 GL_LINE_STRIP

- We use this primitive to draw a sequence of connected line segments. OpenGL renders a line segment between the first and second vertices, between the second and third, between the third and fourth, and so on. If the application specifies n vertices, OpenGL renders n–1 line segments.
  - Draws a connected group of line segments from the first vertex to the last.
  - Vertices n and n+1 define line n.
  - N–1 lines are drawn.
- **Example :**

```
glBegin(GL_LINE_STRIP);
glVertex2f(x1, y1);
glVertex2f(x2, y2);
glVertex2f(x3, y3);
glEnd();
```

2.2.1.4 GL_LINE_LOOP

- We use this primitive to close a line strip. OpenGL renders this primitive like a GL_LINE_STRIP with the addition of a closing line segment between the final and first vertices.
  - Draws a connected group of line segments from the first vertex to the last, then back to the first.
  - Vertices n and n+1 define line n.
  - n lines are drawn.
- **Example :**

```
glBegin(GL_LINE_LOOP);
glVertex2f(x1, y1);
glVertex2f(x2, y2);
glVertex2f(x3,y3);
glEnd();
```

2.2.1.5 GL_TRIANGLES

- We use this primitive to draw individual triangles. OpenGL renders a triangle for each group of three vertices. If your application specifies n vertices, OpenGL renders n/3 triangles. If n isn't a multiple of 3, OpenGL ignores the excess vertices.
  - Treats each triplet of vertices as an independent triangle.
  - Vertices 3n–2, 3n–1, and 3n define triangle n.
  - n/3 triangles are drawn.
- **Example :**

```
glBegin(GL_TRIANGLES);
glVertex2f(x1, y1);
glVertex2f(x2, y2);
glVertex2f(x3, y3);
glEnd();
```

2.2.1.6 GL_TRIANGLE_STRIP

- We use this primitive to draw a sequence of triangles that share edges. OpenGL renders a triangle using the first, second, and third vertices, and then another using the second, third, and fourth vertices, and so on. If the application specifies n vertices, OpenGL renders n–2 connected triangles. If n is less than 3, OpenGL renders nothing.
  - Draws a connected group of triangles.
  - One triangle is defined for each vertex presented after the first two vertices.
  - For odd n, vertices n, n+1, and n+2 define triangle n.
  - For even n, vertices n+1, n, and n+2 define triangle n.
  - n–2 triangles are drawn.
- **Example :**

```
glBegin(GL_TRIANGLE_STRIP);
glVertex2f(x1, y1);
glVertex2f(x2, y2);
glVertex2f(x3, y3);
glVertex2f(x4, y4);
glEnd();
```

2.2.1.7 GL_TRIANGLE_FAN

- We use this primitive to draw a fan of triangles that share edges and also share a vertex. Each triangle shares the first vertex specified. If the application specifies a sequence of vertices v, OpenGL renders a triangle using v0 , v1 and v2 ; another triangle using v0 , v2 and v3 ; another triangle using v0 , v3 and v4 and so on. If the application specifies n vertices, OpenGL renders n–2 connected triangles. If n is less than 3, OpenGL renders nothing.
  - Draws a connected group of triangles that fan around a central point.
  - One triangle is defined for each vertex presented after the first two vertices.
  - Vertices 1, n+1, and n+2 define triangle n.
  - n–2 triangles are drawn.
- **Example :**

```
glBegin(GL_TRIANGLE_FAN);
glVertex2f(x1, y1);
glVertex2f(x2, y2);
glVertex2f(x3, y3);
glVertex2f(x4, y4);
glEnd();
```

2.2.1.8 GL_QUADS

- We use this primitive to draw individual convex quadrilaterals. OpenGL renders a quadrilateral for each group of four vertices. If the application specifies n vertices, OpenGL renders n/4 quadrilaterals. If n isn't a multiple of 4, OpenGL ignores the excess vertices.
  - Treats each group of four vertices as an independent quadrilateral.
  - Vertices 4n–3, 4n–2, 4n–1, and 4n define quadrilateral n.
  - n/4 quadrilaterals are drawn.
- **Example :**

```
glBegin(GL_QUADS);
glVertex2f(x1, y1);
glVertex2f(x2, y2);
glVertex2f(x3, y3);
glVertex2f(x4, y4);
glEnd();
```

2.2.1.9 GL_QUAD_STRIP

- We use this primitive to draw a sequence of quadrilaterals that share edges. If the application specifies a sequence of vertices v, OpenGL renders a quadrilateral using v0, v1, v3 and v2 ; another quadrilateral using v2 ,v3 ,v5 and v4 ; and so on. If the application specifies n vertices, OpenGL renders (n–2)/2 quadrilaterals. If n is less than 4, OpenGL renders nothing.
  - Draws a connected group of quadrilaterals.
  - Treats each group of four vertices as an independent quadrilateral.
  - Vertices 4n–3, 4n–2, 4n–1, and 4n define quadrilateral n.
  - n/4 quadrilaterals are drawn.
- **Example :**

```
glBegin(GL_QUADS);
glVertex2f(x1, y1);
glVertex2f(x2, y2);
glVertex2f(x3,y3);
glVertex2f(x4,y4);
glEnd();
```

2.2.1.10 GL_POLYGON

- We use GL_POLYGON to draw a single filled convex n-gon primitive. OpenGL renders an n-sided polygon, where n is the number of vertices specified by the application. If n is less than 3, OpenGL renders nothing.
  - Draws a single, convex polygon.
  - Vertices 1 through N define this polygon.
  - A polygon is convex if all points on the line segment between any two points in the polygon or at the boundary of the polygon lie inside the polygon.
- **Example :**

```
glBegin(GL_POLYGON);
glVertex2f(x1, y1);
glVertex2f(x2, y2);
glVertex2f(x3, y3);
glVertex2f(x4, y4);
glVertex2f(x5, y5);
glEnd();
```

2.2.2 Attributes

- Properties that describe how an object should be rendered are called **attributes**.
- Available attributes depend on the type of object. For example, a line could be black or green. It could be solid or dashed. A polygon could be filled with a single color or with a pattern. We could display it as filled or only by its edges.
- Fig. 2.2.3 shows various attributes for lines and polygons.
- Some of the text attributes include the direction of the text string, the path followed by successive characters in the string, the height and width of the characters, the font, and the style (bold, italic, underlined).

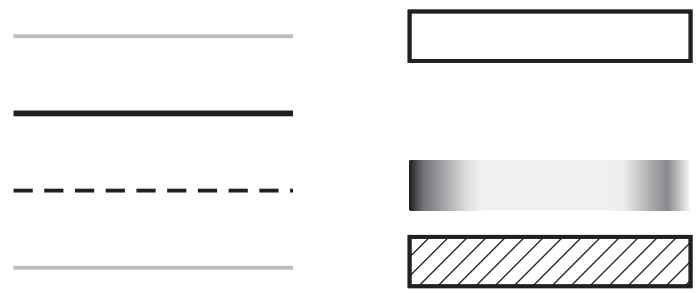


Fig. 2.2.3 Attributes for (a) Lines and (b) Polygons

2.2.2.1 Color Attribute

Colour can be viewed as a point in a colour solid, as shown in Fig. 2.2.4. As shown in Fig. 2.2.4, the solid is drawn using a coordinate system corresponding to the three primaries. The distance along a coordinate axis represents the amount of the corresponding primary in the colour.

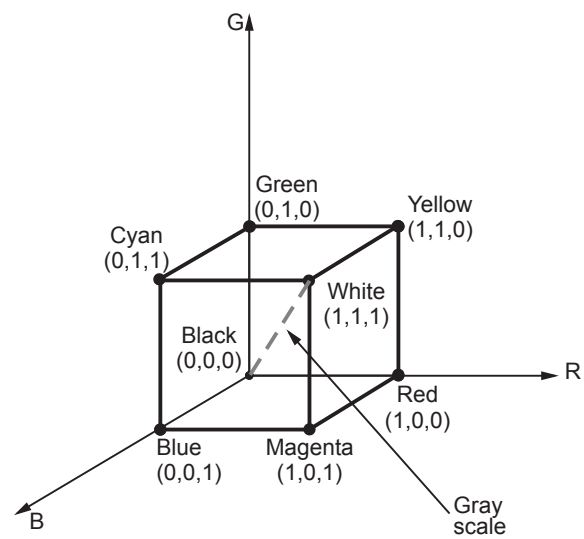


Fig. 2.2.4 Colour solid

The maximum value of each primary is normalized to 1. Any colour can be represented with this set of primaries as a point in a unit cube. The vertices of the cube correspond to black (no primaries on); red, green, and blue (one primary fully on); the pairs of primaries, cyan (green and blue fully on), magenta (red and blue fully on), and yellow (red and green fully on); and white (all primaries fully on). The principal diagonal of the cube connects the origin (black) with white. All colours along this line have equal tristimulus values and appear as shades of gray.

RGB Colour

- The red, green and blue (RGB) colour model used in colour CRT monitors and colour raster graphics employs a Cartesian co-ordinate system. In this model, the individual contribution of red, green and blue are added together to get the resultant colour.
- We can represent this colour model with the unit cube defined on R, G and B axes, as shown in the Fig. 2.2.4.
- The vertex of the cube on the axes represent the primary colours and the remaining vertices represent the complementary colour for each of the primary colours. The main diagonal of the cube, with equal amounts of each primary represents the gray levels. The end at the origin of the diagonal represents black (0, 0, 0) and other end represents white (1, 1, 1).
- Each colour point within the bounds of the cube is represented as the triple (R, G, B), where value for R, G, B are assigned in the range from 0 to 1. As mentioned earlier, it is an additive model. Intensities of the primary colours are added to get the resultant colour. Thus, the resultant colour C_λ is expressed in RGB component as,

  $$C_\lambda = R\mathbf{R} + G\mathbf{G} + B\mathbf{B}$$

- The RGB colour model is used in image capture devices, televisions and colour monitors. This model is additive in which different intensities of red, green and blue are added to generate various colours. However, this model is not suitable for image processing.
- In OpenGL, we use the colour cube as follows. To set a colour for drawing in blue, we have to issue following function call.

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

- In four-dimensional colour scheme to clear an area of the screen (a drawing window) we use following function call.

```
glClearColor(1.0, 1.0, 1.0, 1.0);
```

- Since RGB options are 1.0, 1.0, 1.0 clear colour is white.
- The fourth component in the function call is alpha. It specifies the opacity value. When alpha = 1.0, it is opaque and alpha = 0.0 it is transparent.

Indexed Colour

- It is a fact that any picture has finite number of colours. Thus by storing limited number colour information in the **colour look-up table** it is possible to manage digital images. This techniques is known as **indexed colouring**.

- This technique is used in order to save computer memory and file storage, while speeding up display refresh and file transfers. It is a form of **vector quantization compression**.
- When an image is encoded in this way, colour information is not directly carried by the image pixel data, but is stored in a separate piece of data called a **palette**: an array of colour elements, in which every element, a colour, is indexed by its position within the array. The image pixels do not contain the full specification of its colour, but only its index in the palette. Thus this technique is also referred as **pseudocolour** or **indirect colour**, as colours are addressed indirectly.
- Suppose that our frame buffer has k bits per pixel. Each pixel value or index is an integer between 0 and $2^k - 1$. Suppose that we can display each color component with a precision of n bits; that is, we can choose from 2^n reds, 2^n greens and 2^n blues. Hence, we can produce any of 2^{3n} colours on the display, but the frame buffer can specify only 2^k of them.
- In OpenGL, GLUT allows us to set the entries in a colour table for each window through the following function call.

```
glutsetColor (int color, GLfloat red, GLfloat green, GLfloat blue);
```

- If we are in colour-index mode, the present colour is selected by following function call.

```
glIndexi (element) ;
```

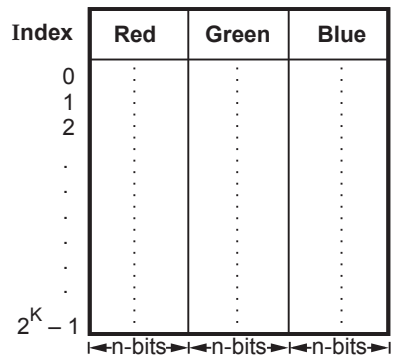


Table 2.2.1 Colour look-up table

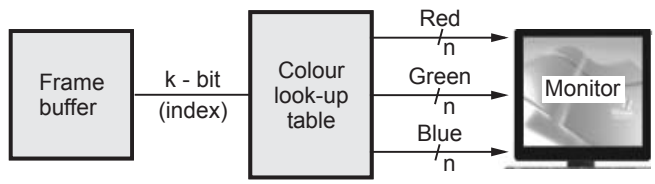


Fig. 2.2.5 Indexed colour

2.2.2.2 OpenGL Point Attributes

- Besides changing the color of a point, we can also specify the point size. We set the size for an OpenGL point with:

```
glPointSize (size);
```

- Parameter size is assigned a positive floating point value, which is rounded to an integer (unless the point is to be anti-aliased). A point size of 1.0 (the default value) displays a single pixel, and a point size of 2.0 displays a 2 by 2 pixel array.

2.2.2.3 OpenGL Line Attributes

- Similar to the point size, the line width can be changed using the following OpenGL function:

```
glLineWidth (width);
```

- We assign a floating-point value to parameter width, and this value is rounded to the nearest nonnegative integer. If the input value rounds to 0.0, the line is displayed with the standard (default) width of 1.0. When using anti-aliasing, fractional widths are possible as well.
- By default, a straight-line segment is displayed as a solid line. But we can also display dashed lines, dotted lines, or a line with a combination of dashed and dots.
- We set a current display style for lines with this OpenGL function:

```
glLineStipple (repeatFactor, pattern);
```

- Parameter pattern is used to reference a 16-bit integer that describes how the line should be displayed. A 1 in the bit-pattern denotes an "on" pixel position. The pattern is applied to the pixels along the line path starting with the low-order bits in the pattern. The default pattern is 0xFFFF which produces a solid line.
- As an example of specifying a line style, the following function call results in dashed lines :

```
glLineStipple (1, 0x00FF);
```

- The first half of this pattern (eight pixels) switches those pixels off, while the second half results in visible pixels. Also, since low-order bits are applied first, a line begins with an eight-pixel dash starting at the first endpoint. This dash is followed by an eight-pixel space, then another eight-pixel dash, and so forth, until the second endpoint position is reached.
- Integer parameter **repeatFactor** specifies how many times each bit in the pattern is to be repeated before the next bit in the pattern is applied. The default repeat value is 1.
- Before a line can be displayed in the current line-style pattern, we must activate the line-style feature of OpenGL. We accomplish this with the following function:

```
glEnable (GL_LINE_STIPPLE);
```

- Without enabling this feature, lines would still appear as solid lines, even though a pattern was provided.
- To disable the use of a pattern we can issue the following function call:

```
glDisable (GL_LINE_STIPPLE);
```

2.2.2.4 OpenGL Fill Attributes

- By default, a polygon is displayed as a solid-color region, using the current color setting.
- To fill the polygon with a pattern in OpenGL, a 32-bit by 32-bit bit mask has to be specified similar to defining a line-style:

```
GLubyte pattern [] = { 0xff, 0x00, 0xff, 0x00, …};
```

- Once we have set the mask, we can establish it as the current fill pattern:

```
glPolygonStipple (pattern);
```

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

```
glEnable (GL_POLYGON_STIPPLE);
```

- Similarly, we turn off the pattern filling with :

```
glDisable (GL_POLYGON_STIPPLE);
```

2.2.2.5 OpenGL Charater Attributes

- We have two methods for displaying characters with the OpenGL. Either we can design a font set using the bitmap functions in the core library, or we can invoke the GLUT character-generation routines.
- The GLUT library contains functions for displaying predefined bitmap and stroke character sets. Therefore, the character attributes we can set are those that apply to either bitmaps or line segments.
- For either bitmap or outline fonts, the display color is determined by the current color state.
- In general, the spacing and size of characters is determined by the font designation, such as GLUT BITMAP 9 BY 15 and GLUT STROKE MONO ROMAN. However, we can also set the line width and line type for the outline fonts.
- We specify the width for a line with the glLineWidth function, and we select a line type with the glLineStipple function. The GLUT stroke fonts will then be displayed using the current values we specified for the OpenGL line-width and line-type attributes.

Review Questions

1. *Explain the different polygons in OpenGL.*
2. *List out different OpenGL primitives, giving examples for each.*
3. *What is an attribute with respect to graphics system ? List attributes for lines and polygons.*
4. *With a neat diagram, discuss the colour formation. Explain the additive and subtractive colours, indexed colour and colour solid concept.*
5. *Write explanatory notes on : i) RGB colour model ii) Indexed colour model.*
6. *List the character attributes.*

2.3 Simple Modelling and Rendering of 2D and 3D Geometric Objects

- Two principal tasks are required to create an image of a two or three-dimensional scene: **modeling** and **rendering.**
- The modeling task generates a model, which is the description of an object that is going to be used by the graphics system. Models must be created for every object in a scene; they should accurately capture the geometric shape and appearance of the object.
- The second task, rendering, takes models as input and generates pixel values for the final image.
- We have already seen that OpenGL supports a handful of primitive types for modeling two-dimensional (2D) and three-dimensional (3D) objects: points, lines, triangles, quadrilaterals, and (convex) polygons.
- In addition, OpenGL includes support for rendering higher-order surface patches using evaluators. A simple object, such as a box, can be represented using a polygon for each face in the object.
- Part of the modeling task consists of determining the 3D coordinates of each vertex in each polygon that makes up a model.
- To provide accurate rendering of a model's appearance or surface shading, the modeler may also have to determine color values, shading normals, and texture coordinates for the model's vertices and faces.
- Complex objects with curved surfaces can also be modeled using polygons. A curved surface is represented by a gridwork or mesh of polygons in which each polygon vertex is placed on a location on the surface.

2.3.1 Primitive Functions Examples

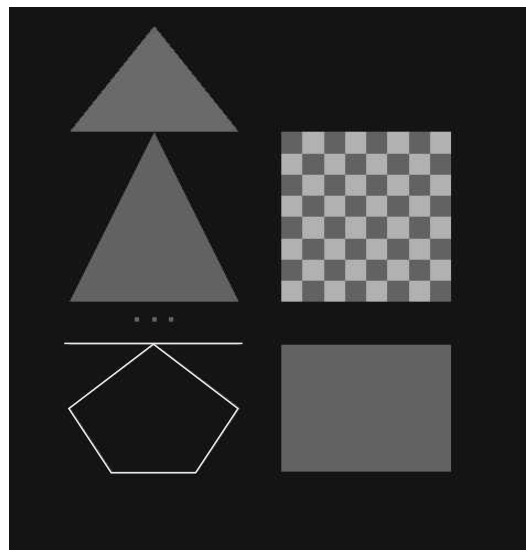
Program 2.3.1 : Draw the basic primitives using OpenGL.

```
#include <GL/glut.h>
void drawChessBoard(int left, int top, int size)
{
int length;
GLfloat col = 0.0;
length = size/8.0;
for(int i=0; i < 8; i++)
{
    for(int j=0; j < 8; j++)
    {
            glColor3f(1.0, col, 0.0);
            glRecti(left + length * j, top - length *i, left + length * j+length,
                    top - length *i-length);
            if(col == 0.0)
            {
                    col = 1.0;
            }
            else
            {
                    col = 0.0;
            }
            }
    if(col == 0.0)
    {
            col = 1.0;
    }
    else
    {
            col = 0.0;
    }
}
}
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
```

```
    glBegin(GL_POINTS);                    // Draw Points
        glVertex2i(180,280);
        glVertex2i(200,280);
        glVertex2i(220,280);
    glEnd();
    glBegin(GL_LINES);                     // Draw Line
        glVertex2i(100, 250);
        glVertex2i(300, 250);
    glEnd();
    glRecti(350, 250, 550, 100);           // Draw Rectangle
    glBegin(GL_TRIANGLES);                 // Draw Triangle
        glVertex2i(100, 300);
        glVertex2i(300, 300);
        glVertex2i(200, 500);
    glEnd();
    glBegin(GL_LINE_LOOP);                 // Draw Polygon
        glVertex2i(100, 175);
        glVertex2i(150, 100);
        glVertex2i(250, 100);
        glVertex2i(300, 175);
        glVertex2i(200, 250);
    glEnd();
    drawChessBoard(350,500,200);
    glBegin(GL_POLYGON);
        glColor3f(1, 0, 0); glVertex3i(100, 500, 0.5);
        glColor3f(0, 1, 0); glVertex3i(300, 500, 0);
        glColor3f(0, 0, 1); glVertex3i(200,625, 0);
    glEnd();
    glFlush();  // Send all output to display
}
void myinit()
{
    // Set a deep black background and draw in a red.
        glClearColor(0.0, 0.0, 0.0, 1.0);      // Set background as black
        glColor3f(1.0, 0.0, 0.0);              // Draw in Red
        glPointSize(4.0);                      // Set point size = 4 pixels
        glMatrixMode(GL_PROJECTION);           // Establish the coordinate system
        glLoadIdentity();
```

```
            gluOrtho2D(0.0, 650.0, 0.0, 650.0);
}
void myKeyboard(unsigned char key, int mouseX, int mouseY )
{
    switch (key) {
       case 27:
            exit(0);
    }
}
int main(int argc, char **argv)
{
    glutInit(&argc, argv);                                  // Initialize the toolkit
            glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);  // Set display mode
            glutInitWindowSize(500, 500);                   // Set window size
            glutInitWindowPosition(100,100);    // Set window position on the screen
            // Open the screen window
            glutCreateWindow("Draw Graphics Primitives on Keypress");
            glutDisplayFunc(display);                       // Register redraw function
            glutKeyboardFunc(myKeyboard);               // Register keyboard function
            myinit();
            glutMainLoop();                                 // go into a perpetual loop
            return 0;
}
```

Output



Program 2.3.2 : Write a program in OpenGL to display the following Fig. 2.3.1 on a raster display system. Assume suitable coordinates for the vertices.

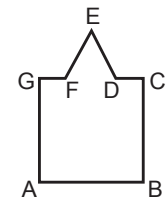
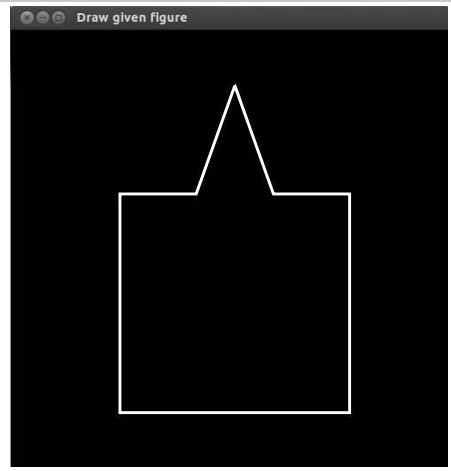


Fig. 2.3.1

```
#include <GL/glut.h>
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_LINE_LOOP);                  // Draw Polygon
        glVertex2i(100, 50);                // Point A
        glVertex2i(310, 50);                // Point B
        glVertex2i(310, 250);               // Point C
        glVertex2i(240, 250);               // Point D
        glVertex2i(205, 350);               // Point E
        glVertex2i(170, 250);               // Point F
        glVertex2i(100, 250);               // Point G
    glEnd();
    glFlush();  // send all output to display
}
void myinit()
{
    // Set a deep black background and draw in a White.
        glClearColor(0.0, 0.0, 0.0, 1.0);   // Set background as black
        glColor3f(1.0, 1.0, 1.0);           // Draw in White
        glMatrixMode(GL_PROJECTION);        // Establish the coordinate system
        glLoadIdentity();
        gluOrtho2D(0.0, 400.0, 0.0, 400.0);
}
int main(int argc, char **argv)
{
    glutInit(&argc, argv);                                  // Initialize the toolkit
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);  // Set display mode
    glutInitWindowSize(500, 500);                           // Set window size
```

```
    glutInitWindowPosition(100,100);          // Set window position on the screen
    // Open the screen window
    glutCreateWindow("Draw given figure");
    glutDisplayFunc(display);                 // Register redraw function
    myinit();
    glutMainLoop();                           // go into a perpetual loop
    return 0;
}
```

Output



2.3.2 Modelling a Colored Cube

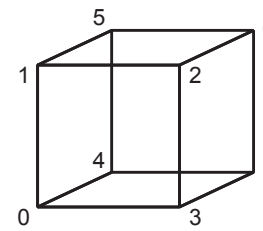


Fig. 2.3.2

- A cube is a simple 3D object.
- Here, we use surface based model. It represents cube either as the intersection of six planes or as the six polygons, called **facets**. Facets define the faces of cube.
- The following code statement initializes an array with the vertices of a cube.

```
GLFloat vertices [8] [3] = {      { – 1.0, – 1.0, – 1.0},
                                  { 1.0, – 1.0, – 1.0},
                                  {1.0, 1.0, – 1.0},
                                  { – 1.0, 1.0, – 1.0},
                                  { – 1.0, – 1.0, 1.0},
                                  { 1.0, – 1.0, 1.0},
                                  { 1.0, 1.0, 1.0},
                                  { – 1.0, 1.0, 1.0 }
                           } ;
```

- Now we can specify the faces of the cube. An example, code to specify one face is as given below :

```
glBegin (GL_POLYGON) ;
    glVertex3fv (vertices [0]) ;
    glVertex3fv (vertices [3]) ;
    glVertex3fv (vertices [2]) ;
    glVertex3fv (vertices [1]) ;
glEnd ( ) ;
```

- Similarly the other five faces can be defined. The other five faces are - (2, 3, 7, 6), (0, 4, 7, 3), (1, 2, 6, 5), (4, 5, 6, 7) and (0, 1, 5, 4). Here, the order of traversal of the edges of cube faces is determined by **right hand rule.** This is illustrated in Fig. 2.3.3.

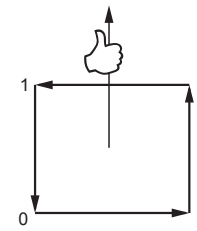


Fig. 2.3.3 Traversal of the edges of a polygon

Review Questions

1. *What is modeling ?*
2. *What is rendering ?*
3. *Write a short note on modeling and rendering.*

2.4 Interaction

- **Definition :** Interaction in the field of computer graphics refers to the process of enabling the users to interact with computer displays. The user can observe the image change in response to the input from the user and act accordingly.
- Support by OpenGL for Interaction OpenGL does not support interaction directly. The main reason for this is to make OpenGL portable (i.e., work on all types of systems irrespective of the hardware).
- However, OpenGL provides the GLUT tool kit which supports minimum functionality such as opening of windows, use of keyboard, mouse and creation of pop-up menus, etc.

2.4.1 Events

- The OpenGL supports basic window events : the window has been resized, the window needs to draw its contents, a key on the keyboard has been pressed or released, a mouse has moved or mouse button was pressed or released. All such event processing is based on callbacks. It is necessary to write functions to process

specific events and set them as callbacks for a specific window. When event occurs, a corresponding callback function is called.

- GLUT handles events with **callback functions**. If you want to handle an event, such as a keystroke or mouse movement, you write a function that performs the desired action. Then you **register** your function by passing its name as an argument to a function with a name of the form glut...Func(), in which "..." indicates which callback you are registering.

2.4.2 Keyboard Callback Functions

- If the program has registered a keyboard callback function, the keyboard callback function is called whenever the user presses a key.

1. Register a keyboard callback function.

```
glutKeyboardFunc(keyboard);
```

2. A keyboard callback function. It receives three arguments : The key that was pressed, and the current X and Y coordinates of the mouse.

- The callback function registered by **glutKeyboardFunc**() recognizes only keys corresponding to ASCII graphic characters and esc, backspace and delete. The keyboard callback function in Fig. 2.4.1 is a useful default keyboard function : It allows the user to quit the program by pressing the escape key.

```
#define ESCAPE 27
void keyboard (unsigned char key, int x, int y)
{
    if (key == ESCAPE)
     exit(0);
}
```

Fig. 2.4.1 Quitting with the escape key

- To make your program respond to other keys, such as the arrow keys and the function ("F") keys :

1. Register a special key callback function.

```
glutSpecialFunc(special);
```

2. Declare the special key function as follows :

GLUT_KEY_F1	GLUT_KEY_F8	GLUT_KEY_UP
GLUT_KEY_F2	GLUT_KEY_F9	GLUT_KEY_DOWN
GLUT_KEY_F3	GLUT_KEY_F10	GLUT_KEY_PAGE_UP

GLUT_KEY_F4	GLUT_KEY_F11	GLUT_KEY_PAGE_DOWN
GLUT_KEY_F5	GLUT_KEY_F12	GLUT_KEY_HOME
GLUT_KEY_F6	GLUT_KEY_LEFT	GLUT_KEY_END
GLUT_KEY_F7	GLUT_KEY_RIGHT	GLUT_KEY_INSERT

Table 2.4.1 Constants for special keys

```
void special (int key, int x, int y)
{
switch (key)
{
case GLUT_KEY_F1:
     // code to handle F1 key
     break;
     ....
}
}
```

- The arguments that OpenGL passes to special() are : the key code, as defined in Table 2.4.1, the X co-ordinate of the mouse and the Y co-ordinate of the mouse.

Program 2.4.1 : This program uses keyboard callback function. The keyboard callback function checks the keypress and accordingly display graphics primitive.

```
#include <GL/glut.h>
int Height=650, Width= 650;
void display(void)
{
     glClear(GL_COLOR_BUFFER_BIT);
     glFlush();  // Send all output to display
}
void myinit()
{
     // Set a deep black background and draw in a red.
          glClearColor(0.0, 0.0, 0.0, 1.0);        // Set background as black
          glColor3f(1.0, 0.0, 0.0);                // Draw in Red
          glMatrixMode(GL_PROJECTION);   // Establish the coordinate system
          glLoadIdentity();
```

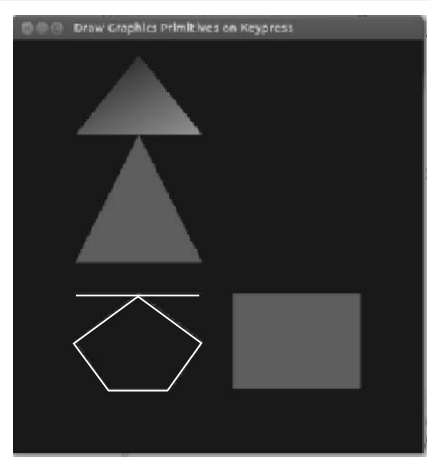
```
            gluOrtho2D(0.0, 650.0, 0.0, 650.0);
}
void myKeyboard(unsigned char key, int mouseX, int mouseY )
{
    switch(key) {
    case 'l':
                        glBegin(GL_LINES);                  // Draw Line
                            glVertex2i(100, 250);
                            glVertex2i(300, 250);
                        glEnd();
                        break;
    case 'r':
                        glRecti(350, 250, 550, 100);        // Draw rectangle
                        break;
    case 't':
                        glBegin(GL_TRIANGLES);              // Draw Triangle
                            glVertex2i(100, 300);
                            glVertex2i(300, 300);
                            glVertex2i(200, 500);
                        glEnd();
                        break;
    case 'p':
                        glBegin(GL_LINE_LOOP);     // Draw Polygon without fill
                            glVertex2i(100, 175);
                            glVertex2i(150, 100);
                            glVertex2i(250, 100);
                            glVertex2i(300, 175);
                            glVertex2i(200, 250);
                        glEnd();
                        break;

    case 'f':
                        glBegin(GL_POLYGON);       // Draw Polygon with fill
                        glColor3f(1, 0, 0); glVertex3i(100, 500, 0.5);
                        glColor3f(0, 1, 0); glVertex3i(300, 500, 0);
                        glColor3f(0, 0, 1); glVertex3i(200, 625, 0);
                    glEnd();
                    break;
```

```
        case 27:
                                exit(0);        // Terminate the program
        }
        glFlush();
}
int main(int argc, char **argv)
{
        glutInit(&argc, argv);                                  // Initialize the toolkit
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);              // Set display mode
        glutInitWindowSize(500, 500);                           // Set window size
        glutInitWindowPosition(100,100);            // Set window position on the screen
        // Open the screen window
        glutCreateWindow("Draw Graphics Primitives on Keypress");
        glutDisplayFunc(display);                   // Register redraw function
        glutKeyboardFunc(myKeyboard);               // Register keyboard function
        myinit();
        glutMainLoop();                             // go into a perpetual loop
        return 0;
}
```

Output

Keypressed

l
r
t
p
f



2.4.3 Mouse Event Callback Functions

- There are several ways of using the mouse and two ways of responding to mouse activity. The first callback function responds to pressing or releasing one of the mouse buttons.

1. Register a mouse callback function.

```
glutMouseFunc(mouse_button);
```

2. The mouse callback function is passed four arguments.

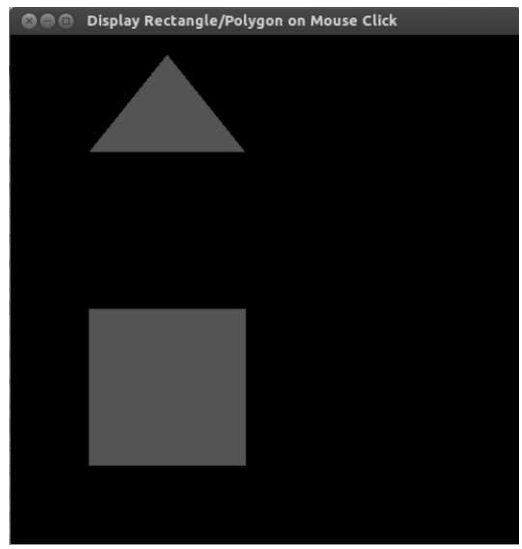
- The first argument specifies the button. Its value is one of
  - GLUT_LEFT_BUTTON,
  - GLUT_MIDDLE_BUTTON, or
  - GLUT_RIGHT_BUTTON
- The second argument specifies the button state. Its value is one of
  - GLUT_DOWN (the button has been pressed) or
  - GLUT_UP (the button has been released).
- The remaining two arguments are the X and Y co-ordinates of the mouse.

Program 2.4.2 : This program displays rectangle on left mouse click and polygon on right mouse click.

```
#include <GL/glut.h>
int Height=650, Width= 650;
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();  // Send all output to display
}
void myinit()
{
    // Set a deep black background and draw in a red.
    glClearColor(0.0, 0.0, 0.0, 1.0);        // Set background as black
    glColor3f(1.0, 0.0, 0.0);                // Draw in Red
    glMatrixMode(GL_PROJECTION);   // Establish the coordinate system
    glLoadIdentity();
    gluOrtho2D(0.0, 650.0, 0.0, 650.0);
}
void myMouse(int button, int state, int x, int y )
{
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {
            glRecti(100, 100, 300, 300);    // Draw rectangle
    }
    if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
```

```
    {
        glBegin(GL_POLYGON);
            glVertex3i(100, 500, 0.5);
            glVertex3i(300, 500, 0);
            glVertex3i(200, 625, 0);
        glEnd();
    }
    glFlush();                                          // Send output to display
}
int main(int argc, char **argv)
{
    glutInit(&argc, argv);                                      // Initialize the toolkit
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);          // Set display mode
    glutInitWindowSize(500, 500);                               // Set window size
    glutInitWindowPosition(100,100);    // Set window position on the screen
    // Open the screen window
    glutCreateWindow("Display Rectangle/Polygon on Mouse Click");
    glutDisplayFunc(display);                   // Register redraw function
    glutMouseFunc(myMouse);                     // Register mouse function
    myinit();
    glutMainLoop();                             // go into a perpetual loop
    return 0;
}
```

Output

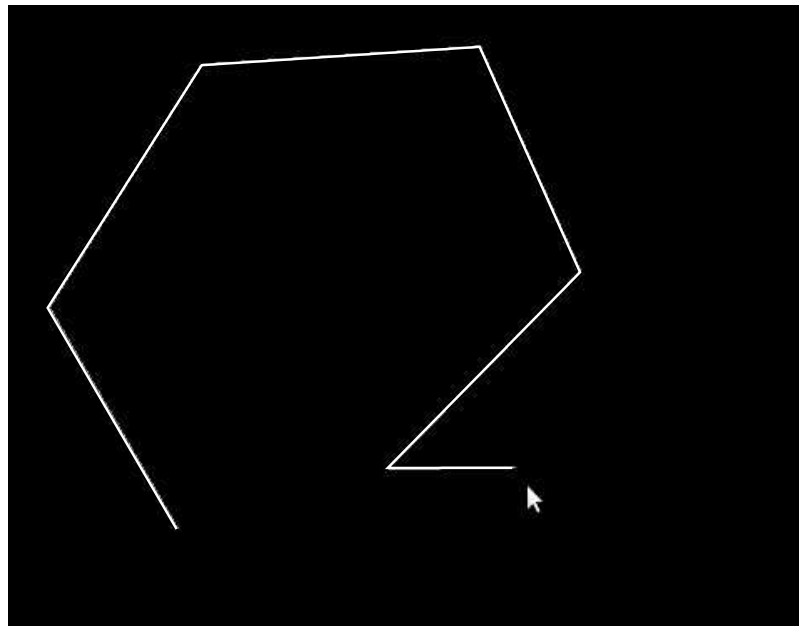


Program 2.4.3 : Create a polyline using mouse interaction using OpenGL

```
#include <GL/glut.h>
struct GLintPoint
{ GLint x,y;
};
int Height=650, Width= 650;
void myMouse(int button, int state, int x, int y );

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();  // Send all output to display
}

void myinit() {
            glClearColor(0.0, 0.0, 0.0, 1.0);        // Set background as black
            glColor3f(1.0, 1.0, 0.0);                // Draw in Yellow
            glMatrixMode(GL_PROJECTION);   // Establish the coordinate system
            glLoadIdentity();
            gluOrtho2D(0.0, 650.0, 0.0, 650.0);
}
void myKeyboard(unsigned char key, int mouseX, int mouseY )
{
    switch (key) {
       case 27:
            exit(0);
    }
}
void myMouse(int button, int state, int x, int y )
{
            static GLintPoint vertex [1];
            static int pt = 0;
            if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
            {
                        if (pt == 0)
                        {
                        vertex [pt].x = x;
```

```
                        vertex [pt].y = Height- y;
                        pt++;
                        }
                        else
                        {
                                glBegin(GL_LINE_STRIP);
                                glVertex2i(vertex [0].x, vertex[0].y);
                                glVertex2i(x, Height- y);
                                glEnd();
                                vertex [0].x = x;
                                vertex [0].y = Height- y;
                        }
                }
        glFlush();
}
int main(int argc, char **argv)
{
    glutInit(&argc, argv);                                  // Initialize the toolkit
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);              // Set display mode
    glutInitWindowSize(500, 500);                           // Set window size
    glutInitWindowPosition(100,100);    // Set window position on the screen
    // Open the screen window
    glutCreateWindow("Draw Polyline using Interaction using OpenGL");
    glutDisplayFunc(display);                   // Register redraw function
    glutKeyboardFunc(myKeyboard);               // Register keyboard function
    glutMouseFunc(myMouse);                     // Register mouse function
    myinit();
    glutMainLoop();                             // go into a perpetual loop
    return 0;
}
```

Output



2.4.4 Window Event

- Most window systems allow a user to resize the window interactively, usually by using the mouse to drag a corner of the window to a new location.
- Whenever the user moves or resizes the graphics window, OpenGL informs your program, provided that you have registered the appropriate **callback** function. The main problem with reshaping is that the user is likely to change the shape of the window as well as its size; you do not want the change to distort your image. Register a callback function that will respond to window reshaping :

```
glutReshapeFunc(reshape);
```

- The above function sets the reshape callback for the current window.
- The reshape callback is triggered when a window is reshaped. A reshape callback is also triggered immediately before a window's first display callback after a window is created.

Reshape Callback Function

```
    void reshape (GLsize w, GLsize h)
{
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity( );
```

```
    gluOrtho2D (0.0, (GLdouble)w, 0.0, (GLdouble)h);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentify ( );
    glViewport (0, 0, w, h);
}
```

- The width and height parameters of the callback specify the new window size in pixels.
- We use these values to create a new OpenGL clipping window using gluOrtho2D, as well as a new viewport with the same aspect ratio.
- Another event such as a window movement without resizing are handled using following function call.

```
glutMotionFunc( );
```

This function sets the motion callback for the current window.

Review Questions

1. *What is interaction?*
2. *What is the necessity of programming event-driven input ? Describe window events and keyboard events.*
3. *Explain how an event driven input can be programmed for a keyboard device.*
4. *Explain how an event driven input can be performed for window events.*
5. *Explain how an event driven input can be programmed for a mouse device.*

2.5 Picking

- In science and engineering 3D visualization computer applications, it is useful for the user to point to something and have the program figure out what is being pointed to. This is called **picking**.
- A pick device is considerably more difficult to implement than the locator device. The reason for this is that the pick device has to identify the object on the display instead to identifying only position on the display.
- We know that, primitives defined in the application program have to go through sequence of geometric operation, rasterization and fragment operation on their way to the frame buffer. Although much of this process is reversible in mathematical sense; however, the hardware is not reversible. Hence, converting from a location on the display to the corresponding primitive is not a simple task.

- There are three ways to deal with this task -
  - Selection
  - Bounding boxes or extends.
  - Back buffer and an extra rendering.
- **Selection** process involves adjusting the clipping region and viewport such that we can keep track of which primitives in a small clipping region are rendered into a region near the cursor. These primitives go into a **hit list** that can be examined by the user program. The OpenGL uses this approach.
- The **bounding box** or extent of an object is the smallest rectangle, aligned with the coordinate axes that contains the object.
- Using bounding boxes it is possible determine the rectangle in screen co-ordinates that corresponds to a rectangle point in object or world co-ordinate.
- This is easy in 2D applications, but difficult in 3D applications.
- In case double buffering, two colour buffers are used : Front buffer and a back buffer. Since **back buffer** is not displayed, we can use it to store rendered objects each in a distinct colour. Thus it is possible to determine an object's contents by simply changing colours wherever a new object appears in the program.

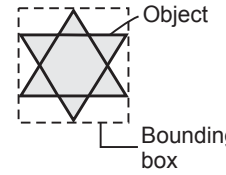


Fig. 2.5.1

- Steps involved in such picking process are as follows :
  - Draw the objects into the back buffer with different colours.
  - Get the position of mouse using callback function.
  - Find the colour at the position in the frame buffer corresponding to the mouse position using **glReadPixel( )** function.
  - Identify the object using pixel colour.

2.5.1 Picking and Selection Mode

- OpenGL uses selection mode approach to implement picking.
- When we plan to use OpenGL's selection mode, we have to first draw our scene into the frame buffer, and then we enter selection mode and redraw the scene. However, once we are in selection mode, the contents of the frame buffer don't change until we exit selection mode.
- When we exit selection mode, OpenGL returns a list of the primitives that intersect the viewing volume (remember that the viewing volume is defined by the current model view and projection matrices and any additional clipping planes).

- Each primitive that intersects the viewing volume causes a selection **hit**. The list of primitives is actually returned as an array of integer-valued **names** and related data - the **hit records** - that correspond to the current contents of the **name stack**.
- We construct the name stack by loading names onto it as we issue primitive drawing commands while in selection mode. Thus, when the list of names is returned, we can use it to determine which primitives might have been selected on the screen by the user.
- In addition to this selection mode, OpenGL provides a utility routine designed to simplify selection in some cases by restricting drawing to a small region of the viewport. Typically, we use this routine to determine which objects are drawn near the cursor, so that we can identify which object the user is picking.

The Basic Steps

- To use the selection mode, we need to perform the following steps :
  1. Specify the array to be used for the returned hit records with **glSelectBuffer()**.
  2. Enter selection mode by specifying GL_SELECT with glRenderMode().
  3. Initialize the name stack using **glInitNames()** and **glPushName()**.
  4. Define the viewing volume we want to use for selection. Usually this is different from the viewing volume we originally used to draw the scene, so we probably want to save and then restore the current transformation state with **glPushMatrix()** and **glPopMatrix()**.
  5. Alternately issue primitive drawing commands and commands to manipulate the name stack so that each primitive of interest has an appropriate name assigned.
  6. Exit selection mode and process the returned selection data (the hit records).

Using Selection Mode for Picking

- We can use selection mode to determine if objects are picked. To do this, we use a special picking matrix in conjunction with the projection matrix to restrict drawing to a small region of the viewport, typically near the cursor.
- Then we allow some form of input, such as clicking a mouse button, to initiate selection mode.
- With selection mode established and with the special picking matrix used, objects that are drawn near the cursor cause selection hits. Thus, during picking we are typically determining which objects are drawn near the cursor.
- Picking is set up almost exactly like regular selection mode is, with the following major differences.
  - Picking is usually triggered by an input device.

- We use the utility routine **gluPickMatrix()** to multiply a special picking matrix onto the current projection matrix. This routine should be called prior to multiplying a standard projection matrix (such as **gluPerspective()** or **glOrtho()**). We will probably want to save the contents of the projection matrix first, so the sequence of operations may look like this :

```
glMatrixMode (GL_PROJECTION);
glPushMatrix ();
glLoadIdentity ();
gluPickMatrix (...);
gluPerspective, glOrtho, gluOrtho2D, or glFrustum
  /* ... draw scene for picking ; perform picking ... */
glPopMatrix();
```

Review Questions

1. *What is picking ?*
2. *Explain the steps to be performed in the selection mode.*
3. *Describe logical input operation of picking in selection mode.*
4. *List down the steps involved in the picking process.*

❑❑❑

Notes

UNIT - I

3 Scan Conversion

Syllabus

Line drawing algorithms : *Digital Differential Analyzer (DDA), Bresenham.*
Circle drawing algorithms : *DDA, Bresenham, and Midpoint.*

Contents

3.1 Introduction

SPPU : Dec.-07,11

- In a raster graphics display system, a picture is completely specified by the set of intensities for the pixel positions in the display.
- We can also describe a picture as a set of complex objects such as trees and furniture and walls, positioned at specified co-ordinate locations within the scene.
- Shapes and colours of the objects can be described internally with pixel arrays or with set of basic geometric structures, such as straight line segments and polygon colour areas.
- Typically, graphics programming packages provide functions to describe a scene in-terms of these basic geometric structures referred to as **output primitives**.
- Point and line segments are the basic output primitives, whereas circles, conic sections, quadric surfaces, polygons, spline curves are the other output primitives.

3.1.1 Lines

- Point is the fundamental element of the picture representation. It is nothing but the position in a plane defined as either pairs or triplets of numbers depending on whether the data are two or three dimensional. Thus, (x_1, y_1) or (x_1, y_1, z_1) would represent a point in either two or three dimensional space.
- Two points would represent a line or edge, and a collection of three or more points a polygon.
- The representation of curved lines is usually accomplished by approximating them by short straight line segments.
- If the two points used to specify a line are (x_1, y_1) and (x_2, y_2), then an equation for the line is given as

$$\frac{y - y_1}{x - x_1} = \frac{y_2 - y_1}{x_2 - x_1} \quad \ldots (3.1.1)$$

$$\therefore \quad y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1 \quad \ldots(3.1.2)$$

$$\text{or} \quad y = mx + b \quad \ldots(3.1.3)$$

$$\text{where} \quad m = \frac{y_2 - y_1}{x_2 - x_1} \quad \ldots(3.1.4)$$

$$\text{and} \quad b = y_1 - mx_1 \quad \ldots(3.1.5)$$

The above equation is called the **slope intercept form** of the line.

- The **slope** m is the change in height $(y_2 - y_1)$ divided by the change in width $(x_2 - x_1)$ for two points on the line.
- The **intercept** b is the height at which the line crosses the y-axis.
- Another form of the line equation, called the general form, may be found by multiplying out the factors and rearranging the equation (3.1.1).

$$(y_2 - y_1)\, x - (x_2 - x_1)\, y + x_2 y_1 - x_1\, y_2 = 0 \quad \ldots(3.1.6)$$

or $$rx + sy + t = 0 \quad \ldots (3.1.7)$$

where $$r = (y_2 - y_1), \quad s = -(x_2 - x_1) \text{ and } t = (x_2\, y_1 - x_1\, y_2)$$

- The values given for r, s and t are called possible values of them because we can see that multiplying r, s and t by any common factor will produce a new set of r′, s′ and t′ values which will also satisfy equation (3.1.7) and, therefore, also describe the same line. Usually values for r, s and t are taken so that

$$r^2 + s^2 = 1 \quad \ldots (3.1.8)$$

Writing equation (3.1.7) in the form of equation (3.1.2) we have,

$$y = \frac{-rx}{s} - \frac{t}{s}$$

$\therefore$ We have, $m = \frac{-r}{s}$ and $b = -\frac{t}{s}$

- We can determine whether two lines are crossing or parallel.
- When two lines cross, they share some point in common. Of course, that point satisfies equations for the two lines. Let us see how to obtain this point.
- Consider two lines having slopes m_1 and m_2 and y-intercepts b_1 and b_2 respectively. Then we can write the equations for these two lines as,

Line 1 : $$y = m_1 x + b_1 \quad \ldots (3.1.9)$$

Line 2 : $$y = m_2 x + b_2 \quad \ldots (3.1.10)$$

- If point (x_p, y_p) is shared by both lines, then it should satisfy equations (3.1.9) and (3.1.10).

$$\therefore \quad y_p = m_1 x_p + b_1 \quad \ldots (3.1.11)$$

and $$y_p = m_2 x_p + b_2 \quad \ldots (3.1.12)$$

Equating the above equations gives,

$$m_1 x_p + b_1 = m_2 x_p + b_2$$

$$\therefore \quad x_p (m_1 - m_2) = b_2 - b_1$$

$$\therefore \quad x_p = \frac{b_2 - b_1}{m_1 - m_2} \quad \ldots (3.1.13)$$

Substituting this into equation for line1, (3.1.11) (or for line2, (3.1.12)) gives,

$$y_p = m_1\left(\frac{b_2 - b_1}{m_1 - m_2}\right) + b_1$$

$$\therefore \quad y_p = \frac{m_1(b_2 - b_1) + b_1(m_1 - m_2)}{m_1 - m_2}$$

$$\therefore \quad y_p = \frac{b_2 m_1 - b_1 m_2}{m_1 - m_2} \quad \ldots (3.1.14)$$

Therefore, the point $\left(\frac{b_2 - b_1}{m_1 - m_2}, \frac{b_2 m_1 - b_1 m_2}{m_1 - m_2}\right)$ … (3.1.15)

is the point shared by both the lines, i.e. intersection point.

- If the two lines are parallel, they have the same slope. In the expression (3.1.15) for the point of intersection, $m_1 = m_2$. So the denominator becomes zero. So the expression results in a divide by zero. This means, there is no point of intersection for the parallel lines.
- If we consider the expression for line given by equation (3.1.7), then by the similar mathematical calculations, we will get the intersecting point,

$$\left(\frac{s_1 t_2 - s_2 t_1}{s_2 r_1 - s_1 r_2}, \frac{t_1 r_2 - t_2 r_1}{s_2 r_1 - s_1 r_2}\right) \quad \ldots (3.1.16)$$

3.1.2 Lines Segments

- When we say the line, it extends forever both forward and backward.
- Any point in a given direction satisfies equation of the line.
- In graphics, we need to display only pieces of lines. When we consider the piece of line, i.e. only those points which lie between two endpoints, then this is called a line segment. Fig. 3.1.1 shows an example of a line segment, where P_1 and P_2 are the endpoints.

Fig. 3.1.1 A line segment

- From the endpoints of a line segment the equation of the line can be obtained. Let the endpoints be P_1 (x_1, y_1) and P_2 (x_2, y_2), and equation of the line, $y = mx + b$. Whether any point lies on the line segment or not, can be determined from the equation of the line and the end-points.
- If any another point $P_i(x_i, y_i)$, is considered, then it lies on the segment if the following conditions are satisfied.
  1. $y_i = mx_i + b$
  2. $\min(x_1, x_2) \leq x_i \leq \max(x_1, x_2)$
  3. $\min(y_1, y_2) \leq y_i \leq \max(y_1, y_2)$

Note *Here min (x_1, x_2) means the smallest of x_1 and x_2, and max (x_1, x_2) means the largest of x_1 and x_2.*

Parametric form

- The equations which give the x and y values on the line in terms of a parameter, u are called 'parametric form' of the line equation.
- This form is useful for constructing line segments.
- For example, we want line segment between (x_1, y_1) and (x_2, y_2). If the parameter is u and if we want the x co-ordinate to go uniformly from x_1 to x_2, then the expression for x can be written as,

$$x = x_1 + (x_2 - x_1)u \quad \ldots (3.1.17)$$

- The line segment points correspond to value of the parameter between 0 and 1. So when u is 0, x is x_1. As u approaches to 1, x changes uniformly to x_2. For a line segment, when x changes from x_1 to x_2, at the same time, y must change from y_1 to y_2 uniformly.

i.e. $$y = y_1 + (y_2 - y_1)u \quad \ldots (3.1.18)$$

- The two equations (3.1.17) and (3.1.18) together describe a straight line. This can be shown by equating over u. If the parameter value for a point on the line is given, we can easily test if the point lies on the line segment.

Length of a line segment

- If the two endpoints of a line segment are known, we can obtain its length.
- Consider the two endpoints of a line segment as P_1 (x_1, y_1) and P_2 (x_2, y_2) and L is the length of a line segment. Construct a right triangle P_1 P_2 A as shown in Fig. 3.1.2.
- The Pythagorean theorem states that the square of the length of the hypotenuse is equal to the sum of the squares of the lengths of the two adjacent sides. Here, hypotenuse is P_1 P_2 and two adjacent sides are P_1A and P_2A.

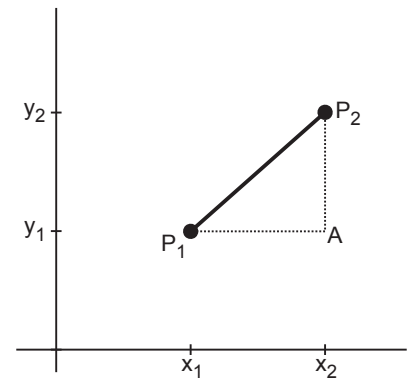


Fig. 3.1.2 Right triangle P_1P_2A

- The co-ordinates of P_1, P_2 and A are (x_1, y_1), (x_2, y_2) and (x_2, y_1) respectively. Then the length of the segment P_1P_2, L is given by,

$$L^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2 \quad \ldots (3.1.9\ (a))$$

$$\therefore \quad L = [(x_2 - x_1)^2 + (y_2 - y_1)^2]^{1/2} \quad \ldots (3.1.19\ (b))$$

Midpoint of a line segment

To obtain the mid-point of a line segment, take the x co-ordinate halfway between the x co-ordinates of the endpoints and y co-ordinate halfway between the y co-ordinates of the endpoints. Thus, the mid-point is,

$$(x_m, y_m) = \left[\frac{(x_1 + x_2)}{2}, \frac{(y_1 + y_2)}{2}\right] \quad \ldots (3.1.20)$$

The mid-point of a line segment can be calculated easily. It is oftenly useful.

3.1.3 Vectors

- A vector has a single direction and a length.
- A vector may be indicated by $[D_x, D_y]$ where D_x denotes the distance on x-axis direction and D_y denote the distance on y-axis direction, as shown in Fig. 3.1.3.
- When we consider a line segment, it has a fixed position in space. But vector does not have a fixed position in space.
- The vector does not tell us the starting point. It gives how far to move and in which direction.

Vector notation

- The vectors may be denoted by a shorthand way. The operations are performed on all co-ordinates.
- For example, the vector form of parametric equations for a line can be written as,

$$V = V_1 + u(V_2 - V_1) \quad \ldots (3.1.21)$$

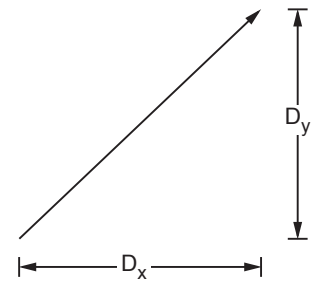


Fig. 3.1.3 A vector

Addition of vectors

- The addition of vectors is nothing but the addition of their respective components. If the two vectors are V_1 $[D_{x1}, D_{y1}]$ and V_2 $[D_{x2}, D_{y2}]$, then the addition of V_1 and V_2 is given by,

$$V_1 + V_2 = [D_{x1}, D_{y1}] + [D_{x2}, D_{y2}]$$
$$= [D_{x1} + D_{x2}, D_{y1} + D_{y2}] \quad \ldots (3.1.22)$$

- We can visualize the vector addition by considering a point A. The first vector V_1 moves from point A to point B, the second vector V_2 moves from point B to point C. Then the addition of V_1 and V_2 is nothing but the vector from point A to point C.

Multiplication of vectors

- The vector can be multiplied by a number. The multiplication is performed by multiplying each component of a vector by a number.

  Let $V(D_x, D_y]$ be a vector which is to be multiplied by a number n. Then,

$$nV = n [D_x, D_y] = [n D_x, nD_y] \quad \ldots (3.1.23)$$

- The magnitude of the resultant vector is given by,

$$|V| = (D_x^2 + D_y^2)^{1/2} \quad \ldots (3.1.24)$$

- The result of the multiplication changes the magnitude of a vector but preserves its direction.

Unit vectors

- From equation (3.1.22) and (3.1.23) we can say that the multiplication of a vector and the reciprocal of its length is equal to 1. Such vectors are called **unit vectors**. They conveniently capture the direction information.

Review Question

1. *What do you mean by output primitives ?*
2. *Define line.* **SPPU : Dec.-07,11, Marks 2**
3. *What is a line segment.*
4. *Define vectors.* **SPPU : Dec.-07,11, Marks 2**

3.2 Line Drawing Algorithms

SPPU : June-12, May-05,10,11,13,14,15,16,17,18,19, Dec.-06,07,08,10,11,12,14, 15,18,19

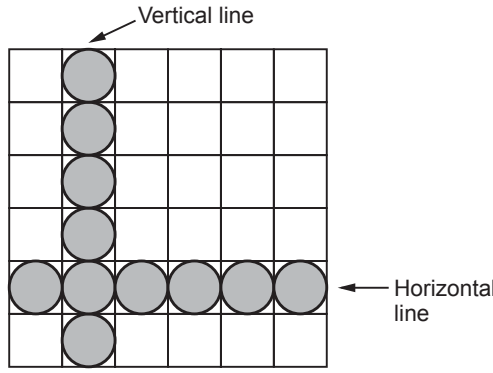
- The process of 'turning on' the pixels for a line segment is called **vector generation** or **line generation** and the algorithms for them are known as **vector generation algorithms** or **line drawing algorithms**.

3.2.1 Qualities of Good Line Drawing Algorithm

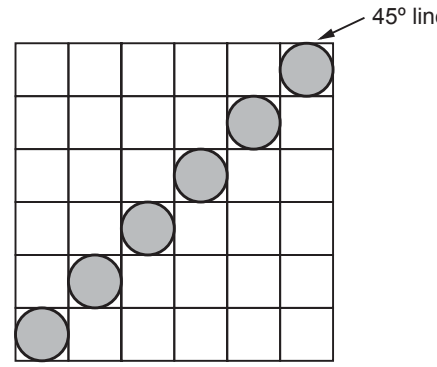
- Before discussing specific line drawing algorithms it is useful to note the general requirements for such algorithms. These requirements specify the desired characteristics of line.
  - The line should appear as a straight line and it should start and end accurately.
  - The line should be displayed with constant brightness along its length independent of its length and orientation.
  - The line should be drawn rapidly.

Let us see the different lines drawn in Fig. 3.2.1.

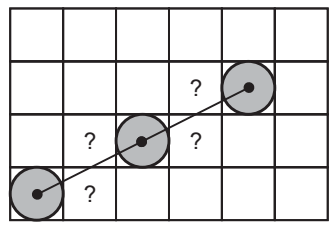
- As shown in Fig. 3.2.1 (a), horizontal and vertical lines are straight and have same width.
- The 45° line is straight but its width is not constant.
- The line with any other orientation is neither straight nor has same width. Such cases are due to the finite resolution of display and we have to accept approximate pixels in such situations, shown in Fig. 3.2.1 (c).

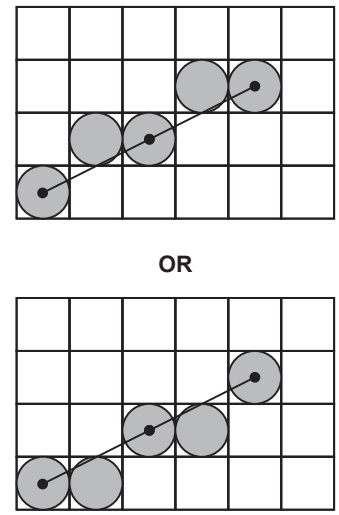


(a) Vertical and Horizontal lines



(b) 45° line





(c) Line with other orientation

Fig. 3.2.1

- The brightness of the line is dependent on the orientation of the line.
- We can observe that the effective spacing between pixels for the 45° line is greater than for the vertical and horizontal lines. This will make the vertical and horizontal lines appear brighter than the 45° line.

- Complex calculations are required to provide equal brightness along lines of varying length and orientation. Therefore, to draw line rapidly some compromises are made such as
  - Calculate only an approximate line length
  - Reduce the calculations using simple integer arithmetic
  - Implement the result in hardware or firmware

Considering the assumptions made most line drawing algorithms use incremental methods. In these methods line starts with the starting point. Then a fix increment is added to the current point to get the next point on the line. This is continued till the end of line. Let us see the incremental algorithm.

Incremental Algorithm

1. CurrPosition = Start

   Step = Increment
2. if (|CurrPosition – End| < Accuracy) then go to step 5

   [This checks whether the current position is reached upto approximate end point. If yes, line drawing is completed.]

   if (CurrPosition < End) then go to step 3

   [Here start < End]

   if (CurrPosition > End) then go to step 4

   [Here start > End]
3. CurrPosition = CurrPosition + Step

   go to step 2
4. CurrPosition = CurrPosition – Step

   go to step 2
5. Stop.

In the following sections we discuss the line rasterizing algorithms based on the incremental algorithm.

3.2.2 Vector Generation/Digital Differential Analyzer (DDA) Algorithm

- The vector generation algorithms which step along the line to determine the pixel which should be turned on are sometimes called **digital differential analyzer (DDA)**.
- The slope of a straight line is given as

$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} \quad \ldots (3.2.1)$$

The above differential equation can be used to obtain a rasterized straight line. For any given x interval Δx along a line, we can compute the corresponding y interval Δy from equation (3.2.1) as

$$\Delta y = \frac{y_2 - y_1}{x_2 - x_1} \Delta x \quad \ldots (3.2.2)$$

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

$$\Delta x = \frac{x_2 - x_1}{y_2 - y_1} \Delta y \quad \ldots (3.2.3)$$

- Once the intervals are known the values for next x and next y on the straight line can be obtained as follows

$$x_{i+1} = x_i + \Delta x$$

$$= x_i + \frac{x_2 - x_1}{y_2 - y_1} \Delta y \quad \ldots (3.2.4)$$

and

$$y_{i+1} = y_i + \Delta y$$

$$= y_i + \frac{y_2 - y_1}{x_2 - x_1} \Delta x \quad \ldots (3.2.5)$$

- The equations (3.2.4) and (3.2.5) represent a recursion relation for successive values of x and y along the required line. Such a way of rasterizing a line is called a **digital differential analyzer** (DDA). For simple DDA either Δx or Δy, whichever is larger, is chosen as one raster unit, i.e.

if $|\Delta x| \geq |\Delta y|$ then

$$\Delta x = 1$$

else

$$\Delta y = 1$$

With this simplification, if $\Delta x = 1$ then

we have $y_{i+1} = y_i + \frac{y_2 - y_1}{x_2 - x_1}$ and

$$x_{i+1} = x_i + 1$$

If $\Delta y = 1$ then

we have $y_{i+1} = y_i + 1$ and

$$x_{i+1} = x_i + \frac{x_2 - x_1}{y_2 - y_1}$$

Let us see the vector generation/digital differential analyzer (DDA) routine for rasterizing a line.

Vector Generation/DDA Line Algorithm

1. Read the line end points (x_1, y_1) and (x_2, y_2) such that they are not equal.
   [if equal then plot that point and exit]
2. $\Delta x = |x_2 - x_1|$ and $\Delta y = |y_2 - y_1|$
3. if $(\Delta x \geq \Delta y)$ then
   length = Δx
   else
   length = Δy
   end if
4. $\Delta x = (x_2 - x_1) /$ length
   $\Delta y = (y_2 - y_1) /$ length
   [This makes either Δx or Δy equal to 1 because length is either $|x_2 - x_1|$ or $|y_2 - y_1|$. Therefore, the incremental value for either x or y is one.]
5. $x = x_1 + 0.5 *$ Sign (Δx)
   $y = y_1 + 0.5 *$ Sign (Δy)

Here, Sign function makes the algorithm work in all quadrant. It returns – 1, 0, 1 depending on whether its argument is < 0, = 0, > 0 respectively. The factor 0.5 makes it possible to round the values in the integer function rather than truncating them.

   plot (Integer (x), Integer (y))

6. i = 1
   [Begins the loop, in this loop points are plotted]
   While (i ≤ length)
   {
   x = x + Δx
   y = y + Δy
   plot (Integer (x), Integer (y))
   i = i + 1
   }
7. Stop

Let us see few examples to illustrate this algorithm.

Example 3.2.1 *Consider the line from (0, 0) to (4, 6). Use the simple DDA algorithm to rasterize this line.*

Solution : Evaluating steps 1 to 5 in the DDA algorithm we have

$x_1 = 0 \quad y_1 = 0 \quad x_2 = 4 \quad y_2 = 6$

$\therefore$ Length $= |y_2 - y_1| = 6$

$\therefore \quad \Delta x = |x_2 - x_1| / \text{length} = \frac{4}{6}$

and $\quad \Delta y = |y_2 - y_1| / \text{length} = 6 / 6 = 1$

Initial value for

$$x = 0 + 0.5 * \text{Sign}\left(\frac{4}{6}\right) = 0.5$$

$$y = 0 + 0.5 * \text{Sign}(1) = 0.5$$

Tabulating the results of each iteration in the step 6 we get,

i	x	y	Plot
1	0.50	0.50	(0, 0)
2	1.17	1.50	(1, 1)
3	1.83	2.50	(1, 2)
4	2.50	3.50	(2, 3)
5	3.17	4.50	(3, 4)
6	3.83	5.50	(3, 5)
7	4.50	6.50	(4, 6)

Table 3.2.1

The results are plotted as shown in the Fig. 3.2.2. It shows that the rasterized line lies to both sides of the actual line, i.e. the algorithm is **orientation dependent**.

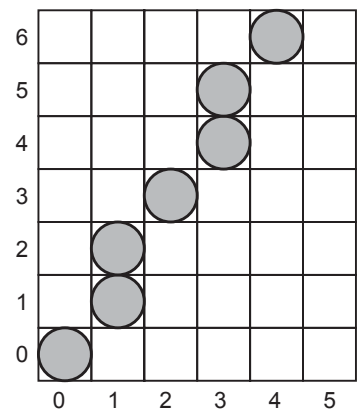


Fig. 3.2.2 Result for a simple DDA

Example 3.2.2 *Consider the line from (0, 0) to (– 6, – 6). Use the simple DDA algorithm to rasterize this line.* **SPPU : June-12, Marks 10**

Solution : Evaluating steps 1 to 5 in the DDA algorithm we have

$$x_1 = 0$$

$$y_1 = 0$$

$$x_2 = -6$$

$$y_2 = -6$$

$$\therefore \quad \text{Length} = |x_2 - x_1| = |y_2 - y_1| = 6$$

$$\therefore \quad \Delta x = \Delta y = -1$$

Initial values for

$$x = 0 + 0.5 * \text{Sign}(-1)$$

$$= -0.5$$

$$y = 0 + 0.5 * \text{Sign}(-1)$$

$$= -0.5$$

Tabulating the results of each iteration in the step 7 we get,

i	x	y	Plot
1	– 0.50	– 0.50	(0, 0)
2	– 1.50	– 1.50	(– 1, – 1)
3	– 2.50	– 2.50	(– 2, – 2)
4	– 3.50	– 3.50	(– 3, – 3)
5	– 4.50	– 4.50	(– 4, – 4)
6	– 5.50	– 5.50	(– 5, – 5)
7	– 6.50	– 6.50	(– 6, – 6)

Table 3.2.2

The results are plotted as shown in the Fig. 3.2.3. It shows that the rasterized line lies on the actual line and it is 45° line.

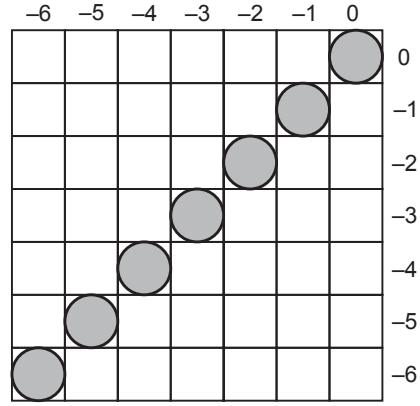


Fig. 3.2.3 Result for a simple DDA

Example 3.2.3 *Using DDA algorithm find out which pixels would be turned on for the line with end points (1, 1) to (5, 3).* **SPPU : May-15, Marks 4**

Solution. : $x_1 = 1,\ y_1 = 1,\ x_2 = 5,\ y_2 = 3$

$\therefore$ Length $= |x_2 - x_1| = 5 - 1 = 4$

$\therefore$ $\Delta x = |x_2 - x_1| / \text{length} = \frac{4}{4} = 1$

$\therefore$ $\Delta y = |y_2 - y_1| / \text{length} = \frac{2}{4} = 0.5$

Initial value for

$x = 1 + 0.5 * \text{sign}(1) = 1.5$

$y = 1 + 0.5 * \text{sign}(0.5) = 1.5$

i	x	y	Plot
1	1.5	1.5	(1, 1)
2	2.5	2.0	(2, 2)
3	3.5	2.5	(3, 2)
4	4.5	3.0	(4, 3)
5	5.5	3.5	(5, 3)

Example 3.2.4 *Scan convert a line with end points (10, 5) and (16, 10) using DDA line drawing algorithm.* **SPPU : May-18, Marks 4**

Solution : $x_1 = 10$, $x_2 = 16$, $y_1 = 5$ and $y_2 = 10$

$\therefore$ Length $= |x_2 - x_1| = 16 - 10 = 6$

$$\Delta y = |y_2 - y_1| / \text{length} = \frac{10-5}{6} = \frac{5}{6}$$

$$\Delta x = |x_2 - x_1| / \text{length} = \frac{6}{6} = 1$$

Initial values for,

$$x = 10 + 0.5 * \text{sign}(1) = 10.5$$

$$y = 5 + 0.5 * \text{sign}(5/6) = 5.5$$

i	x	y	Plot
1	10.50	5.50	(10, 5)
2	11.50	6.33	(11, 6)
3	12.50	7.16	(12, 7)
4	13.50	8.99	(13, 8)
5	14.50	8.83	(14, 8)
6	15.50	9.66	(15, 9)
7	16.50	10.49	(16, 10)

Example 3.2.5 *Consider line segment from (A (– 2), – 1) to B(6, 3) user DDA line drawing algorithm to rasterize this line.* **SPPU : May-19, Marks 4**

Solution : Evaluating steps 1 to 5 in the DDA algorithm we have

$x_1 = -2$, $y_1 = -1$, $x_2 = 6$ and $y_2 = 3$

$\Delta x = |x_2 - x_1| = |6 - (-2)| = 8$

$\Delta y = |y_2 - y_1| = |3 - (-1)| = 4$

$\therefore$ Length $= 8$

$\Delta x = (x_2 - x_1)/\text{Length} = 8/8 = 1$

$\Delta y = (y_2 - y_1)/\text{Length} = 4/8 = 0.5$

i	x	y	Plot
1	– 2	– 1	(– 2, – 1)
2	– 1	– 0.5	(– 1, – 1)
3	0	0	(0, 0)
4	1	0.5	(1, 0)

5	2	1	(2, 1)
6	3	1.5	(3, 1)
7	4	2	(4, 2)
8	5	2.5	(5, 2)
9	6	3	(6, 3)

Advantages of DDA Algorithm

1. It is the simplest algorithm and it does not require special skills for implementation.
2. It is a faster method for calculating pixel positions than the direct use of equation $y = mx + b$. It eliminates the multiplication in the equation by making use of raster characteristics, so that appropriate increments are applied in the x or y direction to find the pixel positions along the line path.

Disadvantages of DDA Algorithm

1. Floating point arithmetic in DDA algorithm is still time-consuming.
2. The algorithm is orientation dependent. Hence end point accuracy is poor.

Examples for Practice

Example 3.2.6 : *Digitize the line with end-points (10, 12) and (20, 18) using DDA algorithm.*

Example 3.2.7 : *Find out points for line segment having end points (0, 0) (– 8, – 4) using DDA line drawing algorithm.* **SPPU : Dec.-15, Marks 6**

3.2.3 Bresenham's Line Algorithm

- Bresenham's line algorithm uses only integer addition and subtraction and multiplication by 2, and we know that the computer can perform the operations of integer addition and subtraction very rapidly. The computer is also time-efficient when performing integer multiplication by powers of 2. Therefore, it is an efficient method for scan-converting straight lines.
- **The basic principle of Bresenham's line algorithm is to select the optimum raster locations to represent a straight line.** To accomplish this the algorithm always increments either x or y by one unit depending on the slope of line. The increment in the other variable is determined by examining the distance between the actual line location and the nearest pixel. This distance is called **decision variable** or the **error**. This is illustrated in the Fig. 3.2.4.

- As shown in the Fig. 3.2.4, the line does not pass through all raster points (pixels). It passes through raster point (0, 0) and subsequently crosses three pixels. It is seen that the intercept of line with the line x = 1 is closer to the line y = 0, i.e. pixel (1, 0) than to the line y = 1 i.e. pixel (1, 1). Hence, in this case, the raster point at (1, 0) better represents the path of the line than that at (1, 1). The intercept of the line with the line x = 2 is close to the line y = 1, i.e. pixel (2, 1) than to the line y = 0, i.e. pixel (2, 0). Hence, the raster point at (2, 1) better represents the path of the line, as shown in the Fig. 3.2.4.

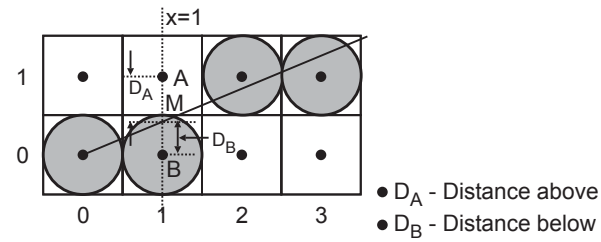


Fig. 3.2.4

- In mathematical terms error or decision variable is defined as

$$e = D_B - D_A \quad \text{or} \quad D_A - D_B$$

- Let us define $e = D_B - D_A$. Now if $e > 0$, then it implies that $D_B > D_A$, i.e., the pixel above the line is closer to the true line. If $D_B < D_A$ (i.e. $e < 0$) then we can say that the pixel below the line is closer to the true line. Thus by checking only the sign of error term it is possible to determine the better pixel to represent the line path.
- The error term is initially set as

$$e = 2\,\Delta y - \Delta x$$

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

Then according to value of e following actions are taken.

```
while ( e ≥ 0)
      {
            y = y + 1
            e = e – 2 * Δx
      }
x = x + 1
e = e + 2 * Δy
```

- When $e \geq 0$, error is initialized with $e = e - 2\,\Delta x$. This is continued till error is negative. In each iteration y is incremented by 1. When $e < 0$, error is initialized to $e = e + 2\,\Delta y$. In both the cases x is incremented by 1. Let us see the Bresenham's line drawing algorithm.

Bresenham's Line Algorithm for $|(m = \Delta y / \Delta x)| < 1$

1. Read the line end points (x_1, y_1) and (x_2, y_2) such that they are not equal.
   [if equal then plot that point and exit]
2. $\Delta x = |x_2 - x_1|$ and $\Delta y = |y_2 - y_1|$
3. [Initialize starting point]
   $x = x_1$
   $y = y_1$
   Plot (x, y) ;
4. $e = 2 * \Delta y - \Delta x$
   [Initialize value of decision variable or error to compensate for nonzero intercepts]
5. $i = 1$
   [Initialize counter]
6. while ($e \geq 0$)
   {
   $y = y + 1$
   $e = e - 2 * \Delta x$
   }
   $x = x + 1$
   $e = e + 2 * \Delta y$
7. Plot (x, y)
8. $i = i + 1$
9. if ($i \leq \Delta x$) then go to step 6.
10. Stop

Example 3.2.8 *Consider the line from (5, 5) to (13, 9). Use the Bresenham's algorithm to rasterize the line.*

Solution : Evaluating steps 1 through 4 in the Bresenham's algorithm we have,

$$\Delta x = | 13 - 5 | = 8$$

$$\Delta y = | 9 - 5 | = 4$$

$$x = 5 \qquad y = 5$$

$$e = 2 * \Delta y - \Delta x = 2 * 4 - 8 = 0$$

Tabulating the results of each iteration in the step 5 through 10.

i	e	x	y	Plot
0	0	5	5	(5, 5)
1	– 8	6	6	(6, 6)
2	0	7	6	(7, 6)
3	– 8	8	7	(8, 7)
4	0	9	7	(9, 7)

5	– 8	10	8	(10, 8)
6	0	11	8	(11, 8)
7	– 8	12	9	(12, 9)
8	0	13	9	(13, 9)
9	– 8	14	10	(14, 10)

Table 3.2.3

The results are plotted as shown in Fig. 3.2.5.

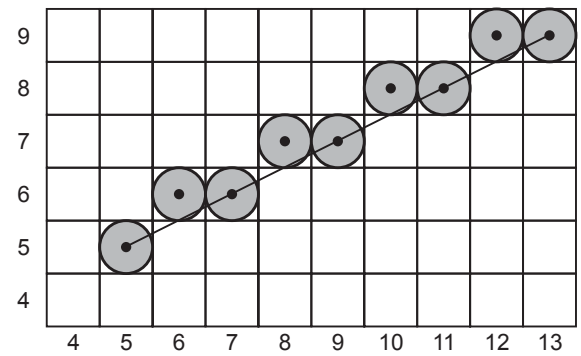


Fig. 3.2.5

3.2.4 Generalized Bresenham's Line Drawing Algorithm

- The Bresenham's algorithm only works for the first octant.
- The generalized Bresenham's algorithm requires modification for lines lying in the other octants. Such algorithm can be easily developed by considering the quadrant in which the line lies and it slope.
- When the absolute magnitude of the slope of the line is greater than 1, y is incremented by one and Bresenham's decision variable

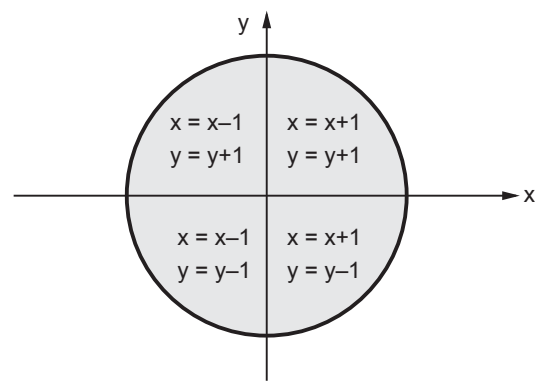


Fig. 3.2.6 Conditions for generalized Bresenham's algorithm

or error is used to determine when to increment x. The x and y incremental values depend on the quadrant in which the line exists. This is illustrated in Fig. 3.2.6.

Generalized Bresenham's Algorithm

1. Read the line end point (x_1, y_1) and (x_2, y_2) such that they are not equal.
2. $\Delta x = |x_2 - x_1|$ and $\Delta y = |y_2 - y_1|$
3. Initialize starting point
   $x = x_1$
   $y = y_1$
   Plot (x, y)
4. s_1 = Sign $(x_2 - x_1)$
   s_2 = Sign $(y_2 - y_1)$
   [Sign function returns – 1, 0, 1 depending on whether its argument is <0, $= 0$, > 0 respectively]
5. if $\Delta y > \Delta x$ then
   Exchange Δx and Δy
   Ex_change = 1
   else
   Ex_change = 0
   end if
   [Interchange Δx and Δy depending on the slope of the line and set Ex_change flag accordingly]
6. $e = 2 * \Delta y - \Delta x$
   [Initialize value of decision variable or error to compensate for nonzero intercept].
7. i = 1
   [Initialize counter]
8. while ($e \geq 0$)
   {
   if (Ex_change = 1) then
   $x = x + s_1$
   else
   $y = y + s_2$
   end if
   $e = e - 2 * \Delta x$
   }
9. if Ex_change = 1 then
   $y = y + s_2$
   else
   $x = x + s_1$
   end if
   $e = e + 2 * \Delta y$
10. plot (x, y)
11. i = i + 1
12. if ($i \leq \Delta x$) then go to step 8
13. Stop

Example 3.2.9 *Generate all raster points on the line segment, if the two end points are given as (10, 20) and (18, 30) using the Bresenham's algorithm.*

Solution : $\Delta x = |18 - 10| = 8$ $\Delta y = |30 - 20| = 10$

$x = 10$ $y = 20$

$S_1 = 1$ $S_2 = 1$

Since $\Delta y > \Delta x$

Exchange Δx and Δy

$\therefore$ $\Delta x = 10$ and $\Delta y = 8$ and Ex_change = 1

Tabulating the result of each iteration we have,

i	e	x	y	Plot
1	6	10	20	(10, 20)
2	2	11	21	(11, 21)
3	– 2	12	22	(12, 22)
4	14	12	23	(12, 23)
5	10	13	24	(13, 24)
6	6	14	25	(14, 25)
7	2	15	26	(15, 26)
8	– 2	16	27	(16, 27)
9	14	16	28	(16, 28)
10	10	17	29	(17, 29)
11	6	18	30	(18, 30)

Example 3.2.10 *Using Bresenham's line algorithm, find out which pixel would be turned on for the line with end points (4, 4) to (12, 9)* **SPPU : Dec.-10, Marks 8**

Solution : Evaluating step 1 through 4 in the Bresenham's algorithm we have,

$\Delta x = |12 - 4| = 8, \quad \Delta y = |9 - 4| = 5$

$x = 4, \quad y = 4$

$e = 2 * \Delta y - \Delta x = 2 * 5 - 8 = 2$

Tabulating the results of each iteration in the step 5 through 10.

i	e	x	y	Plot
1	2	4	4	(4, 4)
2	– 4	5	5	(5, 5)
3	6	6	5	(6, 5)
4	0	7	6	(7, 6)
5	– 6	8	7	(8, 7)
6	4	9	7	(9, 7)
7	– 2	10	8	(10, 8)
8	8	11	8	(11, 8)
9	2	12	9	(12, 9)

Example 3.2.11 *Consider the line from (1, 1) to (6, 4). Use Bresenham's line drawing algorithm to rasterize this line and give output pixels.* **SPPU : Dec.-11, Marks 10**

Solution : Evaluating step 1 through 4 in Bresenham's algorithm we have,

$$\Delta x = |6 - 1| = 5, \quad \Delta y = |4 - 1| = 3$$

$$x = 1, \quad y = 1$$

$$e = 2 * \Delta y - \Delta x = 2 * 3 - 5 = 1$$

Tabulating the results of each iteration in the step 5 through 10.

i	e	x	y	Plot
1	1	1	1	(1, 1)
2	– 3	2	2	(2, 2)
3	3	3	2	(3, 2)
4	– 1	4	3	(4, 3)
5	5	5	3	(5, 3)
6	1	6	4	(6, 4)

Example 3.2.12 *Consider a line from (4, 9) to (7, 7). Use Bresenham's line drawing algorithm to rasterize this line.* **SPPU : June-12, Marks 10**

Solution : Evaluating step 1 through 4 in Bresenham's algorithm we have,

$$\Delta x = |7 - 4| = 3, \quad \Delta y = |7 - 9| = 2$$

$$x = 4, \quad y = 9$$

$$e = 2 * \Delta y - \Delta x = 2 * 2 - 3 = 1$$

Tabulating the results of each iteration in the step 5 through 10.

i	e	x	y	Plot
1	1	4	9	(4, 9)
2	– 1	5	8	(5, 8)
3	3	6	8	(6, 8)
4	1	7	7	(7, 7)

Example 3.2.13 *Consider the line from (0, 0) to (6, 6). Use Bresenham's algorithm to rasterize this line.* **SPPU : May-13, 16, Marks 6**

Solution : Evaluating step 1 through 4 in the Bresenham's algorithm we have,

$\Delta x = |6 - 0| = 6, \quad \Delta y = |6 - 0| = 6$

$x = 0, \quad y = 0$

$e = 2 * \Delta y - \Delta x = 2 * 6 - 6 = 6$

Tabulating the results of each iteration in the step 5 through 10.

i	e	x	y	Plot
1	6	0	0	(0, 0)
2	6	1	1	(1, 1)
3	6	2	2	(2, 2)
4	6	3	3	(3, 3)
5	6	4	4	(4, 4)
6	6	5	5	(5, 5)
7	6	6	6	(6, 6)

Example 3.2.14 *Find out which pixel would be turned on for the line with end points (2, 2) to (6, 5) using the Bresenham's algorithm.* **SPPU : May-14, Marks 3**

Solution : Evaluating step 1 through 4 in the Bresenham's algorithm.

$\Delta x = |6 - 2| = 4, \quad \Delta y = |5 - 2| = 3$

$x = 2, \quad y = 2$

$e = (2 \times \Delta y) - \Delta x = (2 \times 3) - 4 = 2$

Tabulating the result of each iteration in the step 5 through 10.

i	e	x	y	Plot
1	2	2	2	(2, 2)
2	0	3	3	(3, 3)
3	– 2	4	4	(4, 4)
4	4	5	4	(5, 4)
5	2	6	5	(6, 5)

Example 3.2.15 *Find out which pixels would be turned on for the line with end points (5, 2) to (8, 4) using the Bresenham's algorithm.*

SPPU : Dec.-14, Marks 6

Solution : $\Delta x = |8 - 5| = 3$, $\Delta y = |4 - 2| = 2$, $x = 5$, $y = 2$

$e = 2 * \Delta y - \Delta x = 2 * 2 - 3 = 1$

i	e	x	y	Plot
1	1	5	2	(5, 2)
2	– 1	6	3	(6, 3)
3	3	7	3	(7, 3)
4	1	8	4	(8, 4)

Pixels would be turned on for the line : (5, 2), (6, 3), (7, 3), (8, 4).

Example 3.2.16 *Find out which pixel would be turned on for the line with end points (3, 2) to (7, 4) using the Bresenham's algorithm.* **SPPU : May-17, Marks 4**

Solution :

$$\Delta x = |7 - 3| = 4, \quad \Delta y = [4 - 2] = 2, \quad x = 3, \quad y = 2$$

$$e = 2 \times \Delta y - \Delta x = 2 \times 2 - 4 = 0$$

i	e	x	y	Plot
1	0	3	2	(3, 2)
2	– 4	4	3	(4, 3)
3	0	5	3	(5, 3)
4	– 4	6	4	(6, 4)
5	0	7	4	(7, 4)

Pixels would be turned on for the line : (3, 2), (4, 3), (5, 3), (6, 4), (7, 4)

Example 3.2.17 *Consider a line from (2, 5) to (8, 8). Use Bresenham's line drawing algorithm rasterize this line.* **SPPU : Dec.-19, Marks 6**

Solution : Evaluating step 1 through step 4 in the Bresenham's algorithm we have,

$$\Delta x = |8 - 2| = 6, \Delta y = |8 - 5| = 3$$

$$x = 2, y = 5$$

$$e = 2 * \Delta y - \Delta x = 2 * 3 - 6 = 0$$

Tabulating the result of each iteration in the step 5 through 10

i	e	x	y	Plot
1	0	2	5	(2, 5)
2	– 6	3	6	(3, 6)
3	0	4	6	(4, 6)
4	– 6	5	7	(5, 7)
5	0	6	7	(6, 7)
6	– 6	7	8	(7, 8)
7	0	8	8	(8, 8)

Example 3.2.18 *Differentiate Bresenham and Vector generation algorithm for line.*

Solution :

Sr. No.	Vector generation algorithm	Bresenham's line algorithm
1.	It uses floating point arithmetic.	It uses only integer addition, subtraction and multiplication by 2.
2.	Due to floating point arithmetic it takes more time.	It is quicker than vector generation algorithm.
3.	Less efficient.	More efficient.
4.	Where speed is important this algorithm needs to be implemented in hardware.	Hardware implementation is not required.

Examples for Practice

Example 3.2.19 : *Digitize the line with end-points (– 2, 5) to (5, 12) using Bresenham's line algorithm.*

Example 3.2.20 : *Digitize the line with end-points (3, 15) to (– 4, 18) using Bresenham's line algorithm.*

Review Questions

1. *What is vector generation ?* **SPPU : May-05, Marks 2**
2. *What is vector generation ? Explain the problem of vector generation.* **SPPU : Dec.-12, Marks 6**

3. *Explain DDA algorithm for line. Discuss its advantages and disadvantages.*
SPPU : May-05, 10, 11, 17, 19 Dec.-06,07,08, Marks 10
4. *Describe Bresenham's line drawing algorithm.*
SPPU : May-14, 15, 16, 17, 19 Dec 07, 12, 18 Marks 4
5. *Explain any two advantages of Bresenham's line drawing algorithm over other line drawing algorithm.*
SPPU : Dec 18 Marks 2
6. *Give differences between Bresenhams and DDA line drawing algorithm.*
SPPU : May-11, Marks 8

3.3 Antialiasing and Antialiasing Techniques

SPPU : Dec.-10,14, May-13

- In the line drawing algorithms, we have seen that all rasterized locations do not match with the true line and we have to select the optimum raster locations to represent a straight line. This problem is severe in low resolution screens. In such screens line appears like a stair-step, as shown in the Fig. 3.3.1. This effect is known as **aliasing**.

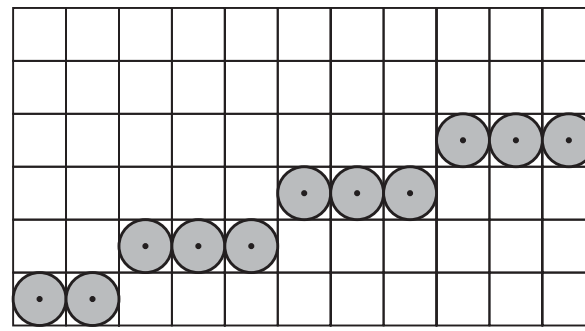


Fig. 3.3.1 Aliasing effect

- It is dominant for lines having gentle and sharp slopes.
- The aliasing effect can be reduced by adjusting intensities of the pixels along the line. The process of adjusting intensities of the pixels along the line to minimize the effect of aliasing is called **antialiasing**.
- The aliasing effect can be minimized by increasing resolution of the raster display. By increasing resolution and making it twice the original one, the line passes through twice as many column of pixels and therefore has twice as many jags, but each jag is half as large in x and in y direction.

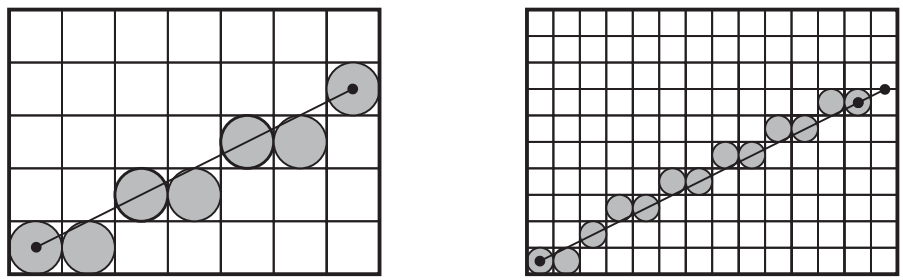


Fig. 3.3.2 Effect on aliasing with increase in resolution

- As shown in the Fig. 3.3.2, line looks better in twice resolution, but this improvement comes at the price of quadrupling the cost of memory, bandwidth of memory and scan-conversion time. Thus increasing resolution is an expensive method for reducing aliasing effect.
- With raster systems that are capable of displaying more than two intensity levels (colour or gray scale), we can apply antialiasing methods to modify pixel intensities. By appropriately varying the intensities of pixels along the line or object boundaries, we can smooth the edges to lessen the stair-step or the jagged appearance. Anti-aliasing methods are basically classified as
  - **Supersampling or Postfiltering :** In this antialiasing method the sampling rate is increased by treating the screen as if it were covered with a finer grid than is actually available. We can then use multiple sample points across this finer grid to determine an appropriate intensity level for each screen pixel. This technique of sampling object characteristics at a high resolution and displaying the results at a lower resolution is called **supersampling** or **postfiltering**.
  - **Area sampling or Prefiltering :** In this antialiasing method pixel intensity is determined by calculating the areas of overlap of each pixel with the objects to be displayed. Antialiasing by computing overlap area is referred to as area sampling or prefiltering.

Antialiasing methods are further classified as shown in the Fig. 3.3.3. Let us discuss them one by one.

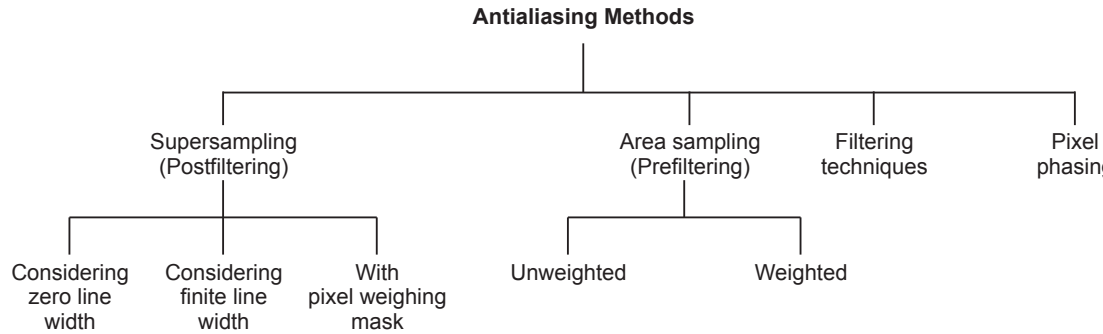


Fig. 3.3.3 Classification of antialiasing methods

3.3.1 Supersampling Considering Zero Line Width

Supersampling of straight lines with zero line width can be performed in several ways.

- For the gray scale display of a straight-line segment, we can divide each pixel into a number of subpixels and count the number of subpixels that are along the line

path. The intensity level for each pixel is then set to a value that is proportional to this subpixel count.

- This is illustrated in Fig. 3.3.4. Here, each pixel area is divided into nine equal-sized square subpixels. The black regions show the subpixels that would be selected by Bresenham's algorithm. In this example, the pixel at position (20, 30) has 3 subpixels along the line path, the pixel at position (21, 31) has 2 subpixels along the line path and the pixel at position (22, 32) has 1 subpixel along the line path. Since pixel at position (20, 30) has 3 (maximum) subpixels its intensity level is kept highest (level 3). The intensity level at pixel position (21, 31) is kept to the medium level (level 2) because pixel has 2 subpixels along the line path, and the intensity at pixel position (22, 32) is kept to the lowest level above zero intensity because it has only 1 subpixel along the line path. Thus the line intensity is spread out over a greater number of pixels, and the stair-step effect is smoothed by displaying a somewhat blurred line path in the vicinity of the stair-steps.
- We can use more intensity levels to antialias the line. For this we have to increase the number of sampling positions across each pixel.
- For example, sixteen subpixels gives us four intensity levels above zero and twenty five subpixels gives us five levels; and so on.

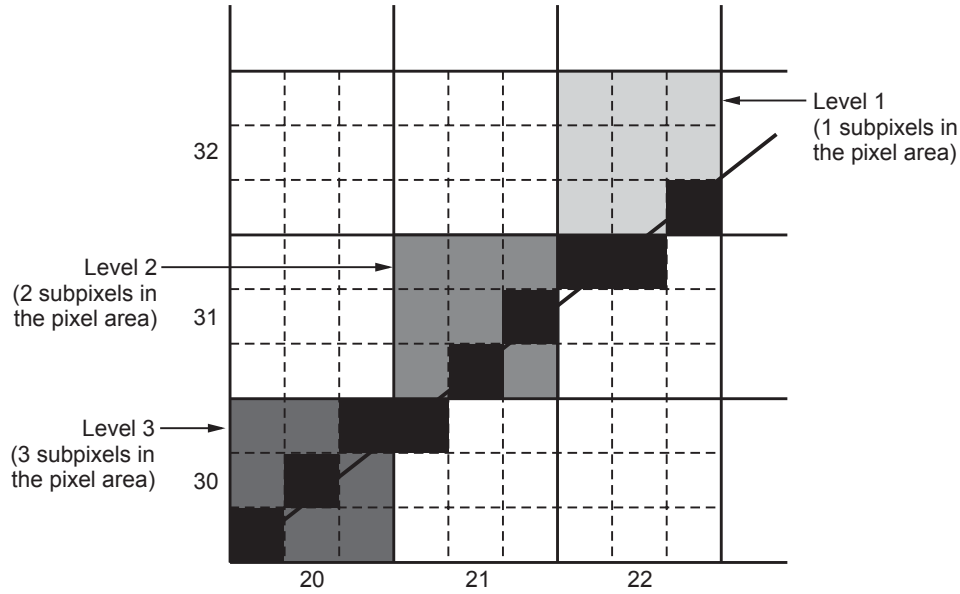


Fig. 3.3.4 Supersampling of subpixel positions

3.3.2 Supersampling Considering Finite Line Width

- Actually most of the times line width is equal to the size of pixel instead of zero width.

- If we take this finite width of the line into account, we can perform supersampling by setting each pixel intensity proportional to the number of subpixels inside the polygon representing the line area as shown in the Fig. 3.3.5.
- A subpixel can be considered to be inside the line if its lower left corner is inside the polygon boundaries. This techniques has following advantages :
  - The number of possible intensity levels for each pixel is equal to the total number of subpixels within the pixel area.
  - The total line intensity is distributed over more pixels.
  - In colour displays, we can extend the method to take background colours into account.

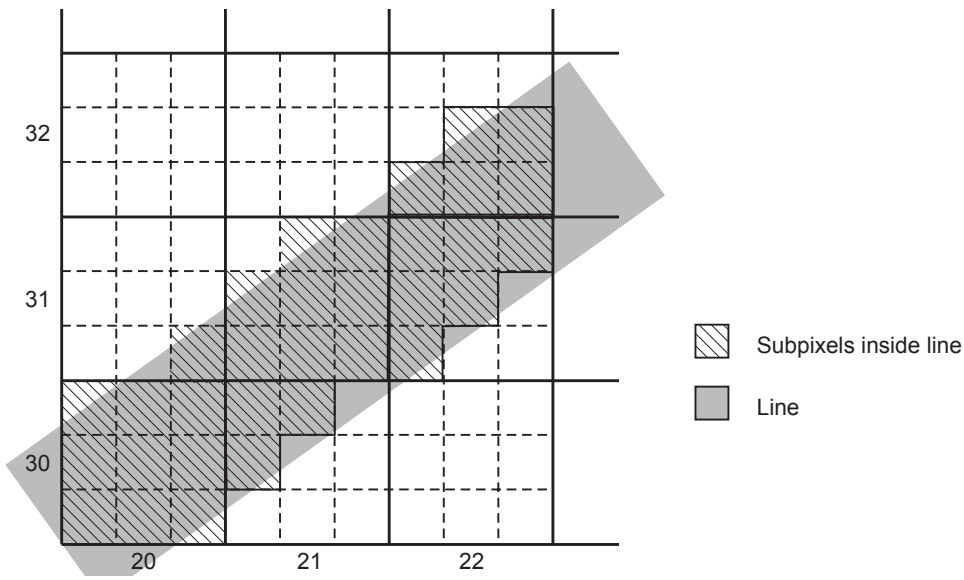


Fig. 3.3.5 Super sampling with finite line width

3.3.3 Supersampling with Pixel-Weighting Mask

- Supersampling method often implemented by giving more weight to subpixels near the center of a pixel area, since we consider these subpixels to be more important in determining the overall intensity of a pixel.
- Fig. 3.3.6 shows the weighting scheme for 3 by 3 pixel subdivisions. Here, center subpixel has four time weight that of the corner subpixels and twice that of the remaining pixels.
- An array of values specifying the relative weights of subpixels is sometimes referred to as a **mask** of subpixel weights.

1	2	1
2	4	2
1	2	1

Fig. 3.3.6 Weighing mask for a grid of 3 by 3 subpixels

- Such pixel-weighting mask can be set up for larger subpixel grids. These masks are often extended to include contributions from subpixels belonging to neighbouring pixels, so that intensities can be averaged over adjacent pixels.

3.3.4 Unweighted Area Sampling

- We have seen that for sloped lines, many a times the line passes between two pixels. In these cases, line drawing algorithm selects the pixel which is closer to the true line. This step in line drawing algorithms can be modified to perform antialiasing.
- In antialiasing, instead of picking closest pixel, both pixels are highlighted. However, their intensity values may differ.

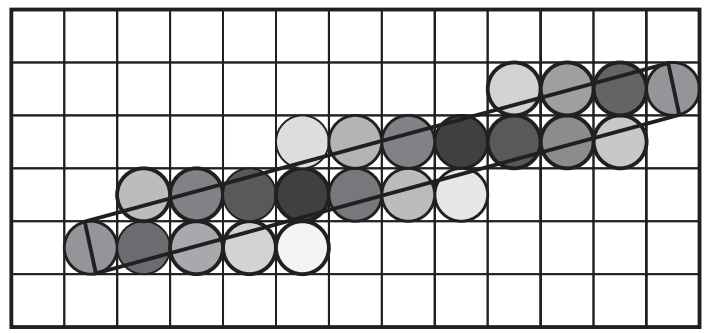


Fig. 3.3.7 Unweighted area sampling

- In unweighted area sampling, the intensity of pixel is proportional to the amount of line area occupied by the pixel.
- This technique produces noticeably better results than does setting pixels either to full intensity or to zero intensity.

3.3.5 Weighted Area Sampling

- We have seen that in unweighted area sampling equal areas contribute equal intensity, regardless of the distance between the pixel's center and the area; only the total amount of occupied area matters. Thus, a small area in the corner of the pixel contributes just as much as does an equal-sized area near the pixel's

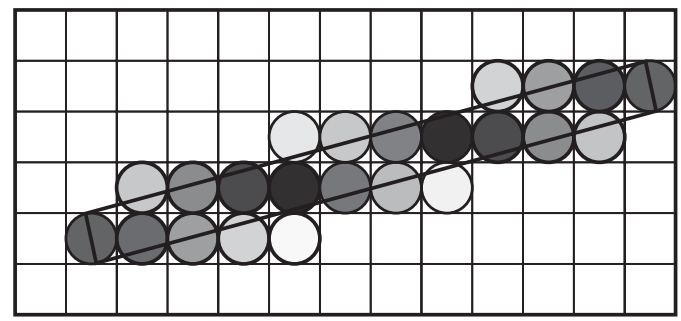


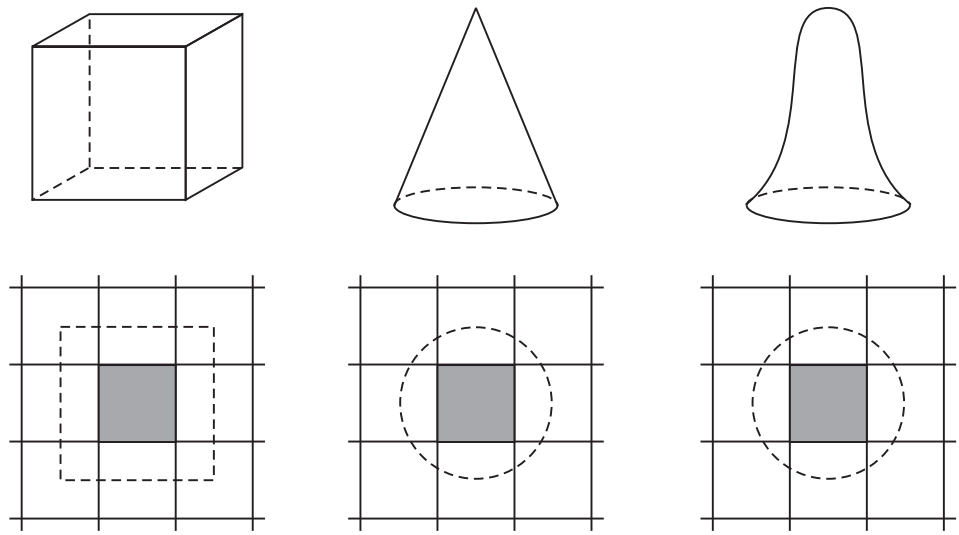
Fig. 3.3.8 Weighted area sampling

center. To avoid this problem even better strategy is used in the weighted area sampling.

- In weighted area sampling equal areas contribute unequally i.e. a small area closer to the pixel center has greater intensity than does one at a greater distance. Thus, in weighted area sampling the intensity of the pixel is dependent on the line area occupied and the distance of area from the pixel's center. This is illustrated in Fig. 3.3.8.

3.3.6 Filtering Techniques

- Filtering technique is more accurate method for antialiasing lines. This method is similar to applying a weighted pixel mask, but here we consider a continuous weighting surface (or filter function) covering the pixel.
- Fig. 3.3.9 shows examples of box, conical and Gaussian filter functions. In this method, we integrate over the pixel surface to obtain the weighted average intensity.



(a) Box filter (b) Cone filter (c) Gaussian filter

Fig. 3.3.9 Common filter functions used to antialias line paths

3.3.7 Pixel Phasing

- Another way to antialias raster objects is to shift the display location of pixel areas. This technique is called **pixel phasing.** It is applied by "micropositioning" the electron beam in relation to object geometry.
- Such technique can be used on raster systems that can address subpixel positions within the screen grid. In this technique stairsteps along a linepath or object

boundary are smoothed out by moving (micropositioning) the electron beam to more nearly approximate positions specified by the object geometry. The electron beam is typically shifted by 1/4, 1/2, or 3/4 of a pixel diameter to plot points closer to the true path of a line or object edge.

Review Questions

1. *List and explain any two antialiasing methods.* **SPPU : Dec.-10, May-13, Dec.-14, Marks 4**
2. *What is aliasing ?* **SPPU : Dec.-14, Marks 2**
3. *What is pixel phasing ?*

3.4 Basic Concepts in Circle Drawing

- A circle is a symmetrical figure. It has eight-way symmetry as shown in the Fig. 3.4.1.
- Any circle generating algorithm can take advantage of the circle symmetry to plot eight points by calculating the co-ordinates of any one point.
- For example, if point A in the Fig. 3.4.1 is calculated with a circle algorithm, seven more points could be found just by reflection.

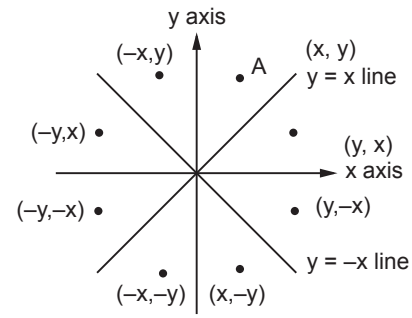


Fig. 3.4.1 Eight-way symmetry of a circle

- There are two standard methods of mathematically representing a circle centered at the origin.

3.4.1 Polynomial Method

- In this method circle is represented by a polynomial equation.

$$x^2 + y^2 = r^2$$

where x : the x co-ordinate

y : the y co-ordinate

r : radius of the circle

- Here, polynomial equation can be used to find y co-ordinate for the known x co-ordinate. Therefore, the scan converting circle using polynomial method is achieved by stepping x from 0 to $r\sqrt{2}$, and each y

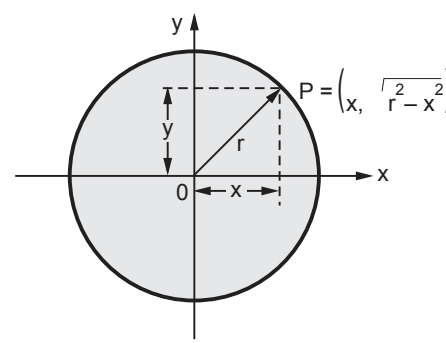


Fig. 3.4.2 Scan converting circle using polynomial method

co-ordinate is found by evaluating $\sqrt{r^2 - x^2}$ for each step of x. This generates the 1/8 portion (90° to 45°) of the circle. Remaining part of the circle can be generated just by reflection.

- The polynomial method of circle generation is an inefficient method. In this method for each point both x and r must be squared and x^2 must be subtracted from r^2; then the square root of the result must be found out.

3.4.2 Trigonometric Method

- In this method, the circle is represented by use of trigonometric functions
  $x = r \cos \theta$ and $y = r \sin \theta$

  where θ : current angle
  r : radius of the circle
  x : the x co-ordinate
  y : the y co-ordinate
- The scan converting circle using trigonometric method is achieved by stepping θ from 0 to $\pi/4$ radians and each value of x and y is calculated.
- This method is more inefficient than the polynomial method because the computation of the values of sin θ and cos θ is even more time-consuming than the calculations required in the polynomial method.

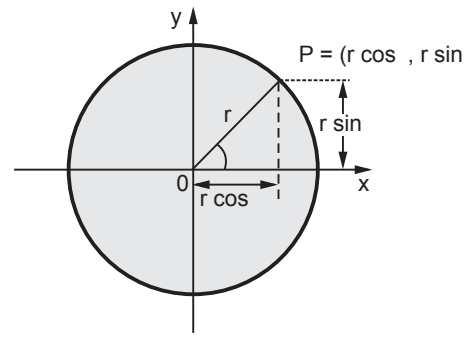


Fig. 3.4.3 Scan converting circle using trigonometric method

Review Questions

1. *Explain the basic concepts in circle drawing.*
2. *Give different methods of representing a circle.*

3.5 Circle Drawing Algorithms

SPPU : May-05,06,07,13,14,16,18,Dec.-07,12,15,17

In this section we are going to discuss three efficient circle drawing algorithms :

- Vector generation/DDA algorithm and
- Bresenham's algorithm
- Midpoint algorithm

3.5.1 Vector Generation/DDA Circle Drawing Algorithm

- We know that, the equation of circle, with origin as the center of the circle is given as
  $x^2 + y^2 = r^2$

- The digital differential analyzer algorithm can be used to draw the circle by defining circle as a differential equation. It is as given below.

$$2x\,dx + 2y\,dy = 0 \qquad \text{where r is constant}$$

$$\therefore \quad x\,dx + y\,dy = 0$$

$$\therefore \quad y\,dy = -x\,dx$$

$$\therefore \quad \frac{dy}{dx} = \frac{-x}{y}$$

- From above equation, we can construct the circle by using incremental x value, $\Delta x = \varepsilon y$ and incremental y value, $\Delta y = -\varepsilon x$, where ε is calculated from the radius of the circle as given below

$$2^{n-1} \leq r < 2^{n} \qquad r : \text{radius of the circle}$$

$$\varepsilon = 2^{-n}$$

- For example, if r = 50 then n = 6 so that $32 \leq 50 < 64$

$$\therefore \quad \varepsilon = 2^{-6} = 0.0156$$

- Applying these incremental steps we have,

$$x_{n+1} = x_n + \varepsilon\, y_n$$

$$y_{n+1} = y_n - \varepsilon\, x_n$$

- The points plotted using above equations give the spiral instead of the circle. To get the circle we have to make one correction in the equation; we have to replace x_n by x_{n+1} in the equation of y_{n+1}.

Therefore, now we have $x_{n+1} = x_n + \varepsilon\, y_n$

$$y_{n+1} = y_n - \varepsilon\, x_{n+1}$$

Algorithm

1. Read the radius (r), of the circle and calculate value of ε
2. start_x = 0
   start_y = r
3. x_1 = start_x
   y_1 = start_y
4. do
   { $x_2 = x_1 + \varepsilon\, y_1$
      $y_2 = y_1 - \varepsilon\, x_2$
   [x_2 represents x_{n+1} and x_1 represents x_n]
      Plot (int (x_2) , int (y_2))
      $x_1 = x_2$;
      $y_1 = y_2$;
   [Reinitialize the current point]

} while (y_1 – start_y) < ε or (start_x – x_1) > ε

[check if the current point is the starting point or not. If current point is not starting point repeat step 4 ; otherwise stop]

5. Stop.

Example 3.5.1 *Calculate the pixel position along the circle path with radius r = 6 centered on the origin using DDA circle algorithm from point (0, 6) to point x = y.*

Solution : Let us find the value of ε. The value of ε is given as, $\varepsilon = 2^{-n}$ where value of 'n' should satisfy the following expression

$2^{n-1} \leq r < 2^n$

Here, r = 6 ∴ Value of n = 3

∴ $\varepsilon = 2^{-3} = 0.125$

Once we know the value of ε we can determine incremental steps as follows :

$$x_{n+1} = x_n + \varepsilon\, y_n$$

$$y_{n+1} = y_n - \varepsilon\, x_{n+1}$$

Tabulating the results of step 4 we get,

x	y	Plot
0.000	6.000	(0, 6)
0.750	5.906	(1, 6)
1.488	5.720	(1, 6)
2.203	5.445	(2, 5)
2.884	5.084	(3, 5)
3.519	4.644	(4, 5)
4.100	4.132	(4, 4)

Example for Practice

Example 3.5.2 : *Calculate the pixel position along the circle path with radius r = 8 centered on the origin using-DDA circle algorithm from point (0, 8) to point x = y.*

3.5.2 Bresenham's Circle Drawing Algorithm

- If the circle is to be plotted efficiently, the use of trignometric and power functions must be avoided.

- It is desirable to perform the calculations necessary to find the scan-converted points with only integer addition, subtraction, and multiplication by powers of 2.
- Bresenham's circle drawing algorithm allows these goals to be met.
- The Bresenham's circle drawing algorithm considers the eight-way symmetry of the circle to generate it. It plots $1/8^{th}$ part of the circle, i.e. from 90° to 45°, as shown in the Fig. 3.5.1. As circle is drawn from 90° to 45°, the x moves in positive direction and the y moves in negative direction.

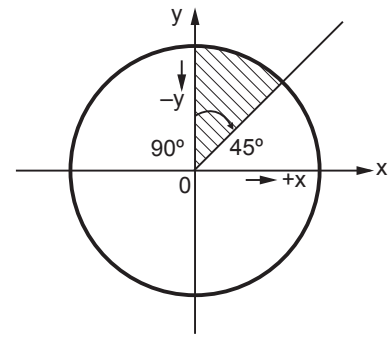


Fig. 3.5.1 1/8 part of circle

- To achieve best approximation to the true circle, we have to select those pixels in the raster that fall the least distance from the true circle. Refer Fig. 3.5.2. Let us observe the 90° to 45° portion of the circle. It can be noticed that if points are generated from 90° to 45°, each new point closest to the true circle can be found by applying either of the two options :

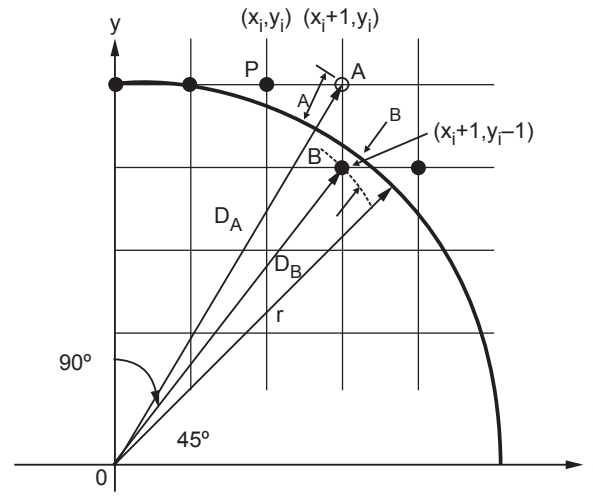


Fig. 3.5.2 Scan conversion with Bresenham's algorithm

  - Increment in positive x direction by one unit or
  - Increment in positive x direction and negative y direction both by one unit
- Therefore, a method of selecting between these two choices is all that is necessary to find the points closest to the true circle.
- Let us assume point $P(x_i, y_i)$ in Fig. 3.5.2 as a last scan converted pixel. Now we have two options either to choose pixel A or pixel B. The closer pixel amongst these two can be determined as follows
- The distances of pixels A and B from the origin are given as

$$D_A = \sqrt{(x_{i+1})^2 + (y_i)^2} \quad \text{and}$$

$$D_B = \sqrt{(x_{i+1})^2 + (y_i - 1)^2}$$

Now, the distances of pixels A and B from the true circle are given as

$$\delta_A = D_A - r \text{ and } \delta_B = D_B - r$$

However, to avoid square root term in derivation of decision variable, i.e. to simplify the computation and to make algorithm more efficient the δ_A and δ_B are defined as

$$\delta_A = D_A^2 - r^2 \quad \text{and}$$

$$\delta_B = D_B^2 - r^2$$

That is

$$\delta_A = (x_i + 1)^2 + y_i^2 - r^2 \quad \ldots (3.5.1)$$

$$\delta_B = (x_i + 1)^2 + (y_i - 1)^2 - r^2 \quad \ldots (3.5.2)$$

This function D provides a relative measurement of the distance from the centre of a pixel to the true circle. Since D_A will always be positive (A is outside of the true circle) and D_B will always be negative (B is inside of the true circle), a **decision variable** d_i can be defined as

$$d_i = \delta_A + \delta_B \quad \ldots (2.10.3)$$

From equations (3.5.1), (3.5.2) and (3.5.3) we have

$$d_i = (x_i + 1)^2 + y_i^2 - r^2 + (x_i + 1)^2 + (y_i - 1)^2 - r^2$$

$$= 2(x_i + 1)^2 + y_i^2 + (y_i - 1)^2 - 2r^2 \quad \ldots (3.5.4)$$

When $d_i < 0$, we have $|\delta_A| < |\delta_B|$ and pixel A is chosen. When $d_i \geq 0$, we have $|\delta_A| \geq |\delta_B|$ and pixel B is selected. In other words we can write,

For $d_i < 0$, $x_{i+1} = x_i + 1$ and

For $d_i \geq 0$, $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i - 1$

Derivation of Decision Variable ($d_i + 1$)

$$d_{i+1} = 2(x_{i+1} + 1)^2 + y_{i+1}^2 + (y_{i+1} - 1)^2 - 2r^2$$

$$\therefore \quad d_{i+1} - d_i = 2(x_{i+1} + 1)^2 + y_{i+1}^2 + (y_{i+1} - 1)^2 - 2(x_i + 1)^2 - y_i^2 - (y_i - 1)^2$$

Since $x_{i+1} = x_i + 1$ We have,

$$d_{i+1} = d_i + 2(x_i + 1 + 1)^2 + y_{i+1}^2 + (y_{i+1} - 1)^2 - 2(x_i + 1)^2 - y_i^2 - (y_i - 1)^2$$

$$= d_i + 2(x_i^2 + 4x_i + 4) + y_{i+1}^2 + y_{i+1}^2 - 2y_{i+1} + 1 - 2(x_i^2 + 2x_i + 1) - y_i^2 - (y_i^2 - 2y_i + 1)$$

$$= d_i + 2x_i^2 + 8x_i + 8 + 2y_{i+1}^2 - 2y_{i+1} + 1 - 2x_i^2 - 4x_i - 2$$

$$- y_i^2 - y_i^2 + 2y_i - 1$$

$$= d_i + 4x_i + 2(y_{i+1}^2 - y_i^2) - 2(y_{i+1} - y_i) + 6$$

If A is chosen pixel (meaning that the $d_i < 0$) then $y_{i+1} = y_i$ and so $d_{i+1} = d_i + 4x_i + 6$. On the other hand, if B is the chosen pixel (meaning that the $d_i \geq 0$) then $y_{i+1} = y_i - 1$ and so

$$d_{i+1} = d_i + 4x_i + 2\left[(y_i - 1)^2 - y_i^2\right] - 2\left[y_i - 1 - y_i\right] + 6$$

$$= d_i + 4x_i + 2\left[y_i^2 - 2y_i + 1 - y_i^2\right] - 2(-1) + 6$$

$$= d_i + 4x_i - 4y_i + 2 + 2 + 6 = d_i + 4(x_i - y_i) + 10$$

Finally, the equation for d_i at starting point, i.e. at $x = 0$ and $y = r$ can be simplified as follows

$$d_i = \delta_A + \delta_B$$

$$= (x_i + 1)^2 + (y_i)^2 - r^2 + (x_i + 1)^2 + (y_i - 1)^2 - r^2$$

$$= (0 + 1)^2 + (r)^2 - r^2 + (0 + 1)^2 + (r - 1)^2 - r^2$$

$$= 1 + r^2 - r^2 + 1 + r^2 - 2r + 1 - r^2 = \mathbf{3 - 2r}$$

We can now summarize the algorithm for generating all the pixel coordinates in the 90° to 45° octant that are needed when scan-converting a circle of radius r.

Algorithm to plot 1/8 of the circle

```
1.  Read the radius (r) of the circle.
2.  d = 3 – 2 r
    [Initialize the decision variable]
3.  x = 0, y = r
    [Initialize starting point]
4.  do
    {
        plot (x, y)
        if (d < 0) then
        {
            d = d + 4x + 6
    }
    else
        { d = d + 4 (x – y) + 10
          y = y – 1
```

```
            }
            x = x + 1
        } while (x < y)
5.      Stop
```

- The remaining part of circle can be drawn by reflecting point about y axis, x axis and about origin as shown in Fig. 3.5.3.
- By adding seven more plot commands after the plot command in the step 4 of the algorithm, the circle can be plotted. The remaining seven plot commands are :

  plot (y, x)

  plot (y, – x)

  plot (x, – y)

  plot (– x, – y)

  plot (– y, – x)

  plot (– y, x) and

  plot (– x, y)

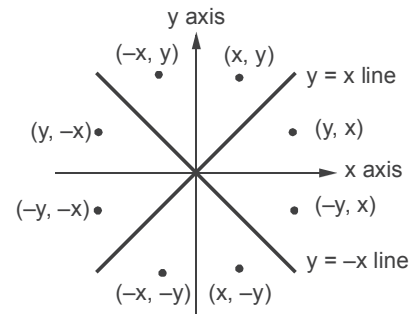


Fig. 3.5.3 Eight-way symmetry of the circle

Example 3.5.3 *Calculate the pixel position along the circle path with radius r = 10 centered on the origin using Bresenham's circle drawing algorithm from point (0,10) to point x = y.*

Solution : The value of d is given as,

$$d = 3 - 2r = 3 - 2 * 10 = -17$$

Tabulating the results of step 4 we get,

x	y	d
0	10	– 17
1	10	– 11
2	10	– 1
3	10	13
4	9	– 5
5	9	17
6	8	11
7	7	13

Example for Practice

Example 3.5.4 : *Calculate the pixel position along the circle path with radius r = 14 centered on the origin using Bresenham's circle drawing algorithm from point (0, 14) to point x = y.*

3.5.3 Midpoint Circle Algorithm

The midpoint circle drawing algorithm also uses the eight-way symmetry of the circle to generate it. It plots 1/8 part of the circle, i.e. from 90° to 45°, as shown in the Fig. 3.5.4. As circle is drawn from 90° to 45°, the x moves in the positive direction and y moves in the negative direction. To draw a 1/8 part of a circle we take unit steps in the positive x direction and make use of decision parameter to determine which of the two possible y positions is closer to the circle path at each step. The Fig. 2.6.4 shows the two possible y positions (y_i and $y_i + 1$) at sampling position $x_i + 1$. Therefore, we have to determine whether the pixel at position $(x_i + 1, y_i)$ or at position $(x_i + 1, y_i - 1)$ is closer to the circle. For this purpose decision parameter is used. It uses the circle function ($f_{circle}(x, y) = x^2 + y^2 - r^2$) evaluated at the midpoint between these two pixels.

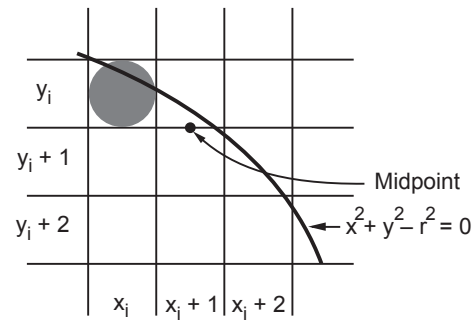


Fig. 3.5.4 Decision parameter to select correct pixel in circle generation algorithm

$$d_i = f_{circle}(x_i + 1, y_i - \frac{1}{2}) = (x_i + 1)^2 + \left(y_i - \frac{1}{2}\right)^2 - r^2$$

$$= (x_i + 1)^2 + y_i^2 - y_i + \frac{1}{4} - r^2 \quad \ldots (3.5.5)$$

If $d_i < 0$, this midpoint is inside the circle and the pixel on the scan line y_i is closer to the circle boundary. If $d_i \geq 0$, the midposition is outside or on the circle boundary, and $y_i - 1$ is closer to the circle boundary. The incremental calculation can be determined to obtain the successive decision parameters. We can obtain a recursive expression for the next decision parameter by evaluating the circle function at sampling position $x_{i+1} + 1 = x_i + 2$.

$$d_{i+1} = f_{circle}(x_{i+1} + 1, y_{i+1} - \frac{1}{2}) = \left[(x_i + 1) + 1\right]^2 + \left(y_{i+1} - \frac{1}{2}\right)^2 - r^2$$

$$= (x_i + 1)^2 + 2(x_i + 1) + 1 + y_{i+1}^2 - \left(y_{i+1}\right) + \frac{1}{4} - r^2 \quad \ldots (3.5.6)$$

Looking at equations (2.6.5) and (2.6.6) we can write

$$d_{i+1} = d_i + 2(x_i + 1) + \left(y_{i+1}^2 - y_i^2\right) - \left(y_{i+1} - y_i\right) + 1$$

where y_{i+1} is either y_i or y_{i-1}, depending on the sign of d_i.

If d_i is negative, $y_{i+1} = y_i$

$\therefore$ $d_{i+1} = d_i + 2(x_i + 1) + 1$

$= d_i + 2x_{i+1} + 1$ … (3.5.7)

If d_i is positive, $y_{i+1} = y_{i-1}$

$\therefore$ $d_{i+1} = d_i + 2(x_i + 1) + 1 - 2y_{i+1}$ … (3.5.8)

The terms $2x_{i+1}$ and $-2y_{i+1}$ in equations (3.5.3) and (3.5.4) can be incrementally calculated as

$$2x_{i+1} = 2x_i + 2$$

$$2y_{i+1} = 2y_i - 2$$

The initial value of decision parameter can be obtained by evaluating circle function at the start position $(x_0, y_0) = (0, r)$.

$$d_0 = f_{circle}\left((0+1)^2 + \left(r - \frac{1}{2}\right)^2 - r^2\right) = 1 + \left(r - \frac{1}{2}\right)^2 - r^2 = 1.25 - r$$

Algorithm

1. Read the radius (r) of the circle
2. Initialize starting position as
   x = 0
   y = r
3. Calculate initial value of decision parameter as
   d = 1.25 – r
4. do
```
{   plot (x, y)
    if (d < 0)
    {       x = x + 1
            y = y
            d = d + 2x + 1
    }
    else
{   x = x + 1
    y = y – 1
    d = d + 2x + 2y + 1
}
while (x < y)
```

5. Determine symmetry points
6. Stop.

Example 3.5.5 *Calculate the pixel position along the circle path with radius r = 10 centered on the origin using midpoint circle drawing algorithm from point (0,10) to point x = y.*

Solution : Initial value of decision parameter

$$d = 1.25 - r = -8.75$$

i	d	x	y
1	– 8.75	0	10
2	– 5.75	1	10
3	– 0.75	2	10
4	6.25	3	10
5	– 2.75	4	9
6	8.25	5	9
7	5.25	6	8
8	6.25	7	7

Review Questions

1. *Explain vector generation algorithm for circle.* **SPPU : May-05, Marks 6**
2. *Derive the expression for decision parameter used in Bresenham's circle algorithm . Also explain the Bresenham's circle algorithm.* **SPPU : May-06, 07, Dec.-07, 12, Marks 10**
3. *Explain Bresenham's circle drawing algorithm with mathematical derivation.* **SPPU : May-14, May-16, Marks 6**
4. *Explain Bresenham's circle drawing algorithm in detail. Also explain error factor with derivations.* **SPPU : May-13, Dec.-15, Marks 12**
5. *Explain vector/DDA generation algorithm for circle.* **SPPU : May-05, Marks 6**
6. *Derive the expression for decision parameter used in Bresenham's circle algorithm . Also explain the Bresenham's circle algorithm.* **SPPU : May-06, 07, Dec.-07, 12, Marks 10**
7. *Explain Bresenham's circle drawing algorithm with mathematical derivation.* **SPPU : May-14, 16, 18 Dec 17 Marks 6**
8. *Explain Bresenham's circle drawing algorithm in detail. Also explain error factor with derivations.* **SPPU : May-13, Dec.-15, Marks 12**
9. *Write and explain the midpoint circle generating algorithm.*

❑❑❑

UNIT - II

4 Polygons and Polygon Filling

Syllabus

Polygons : *Introduction to polygon, types : convex, concave and complex. Inside test.*

Polygon Filling : *flood fill, seed fill, scan line fill.*

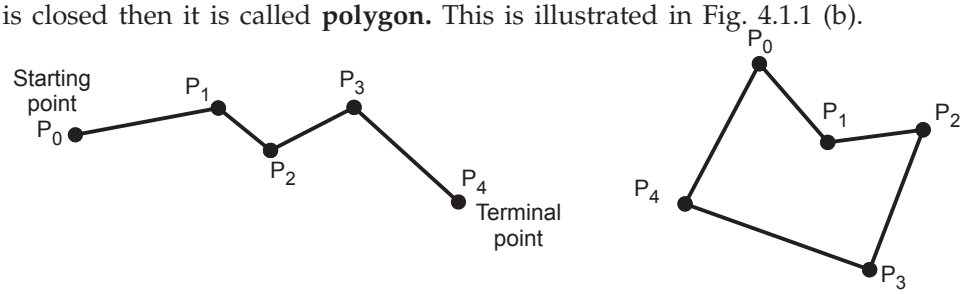
Contents

4.1 Introduction to Polygon

SPPU : Dec.-06, May-11,12

In this chapter we are going to study different types of polygons, their representation and filling algorithms for them.

- A polyline is a chain of connected line segments.
- It is specified by giving the vertices (nodes) P_0, P_1, P_2 … and so on.
- The first vertex is called the initial or starting point and the last vertex is called the final or **terminal point,** as shown in the Fig. 4.1.1 (a).
- When starting point and terminal point of any polyline is same, i.e. when polyline is closed then it is called **polygon.** This is illustrated in Fig. 4.1.1 (b).



(a) Polyline **(b) Polygon**

Fig. 4.1.1

Review Questions

1. *What is polygon ?* **SPPU : Dec.-06, May-12, Marks 2**
2. *Is circle a polygon ? Justify* **SPPU : May-11, Marks 4**

4.2 Types : Convex, Concave and Complex

SPPU : May-05,10,11,12, Dec.-06,09,18

- The classification of polygons is based on where the line segment joining any two points within the polygon is going to lie. There are three types of polygons :
  - Convex
  - Concave and
  - Complex
- A convex polygon is a polygon in which the line segment joining any two points within the polygon lies completely inside the polygon. Fig. 4.2.1 shows the examples of convex polygons.

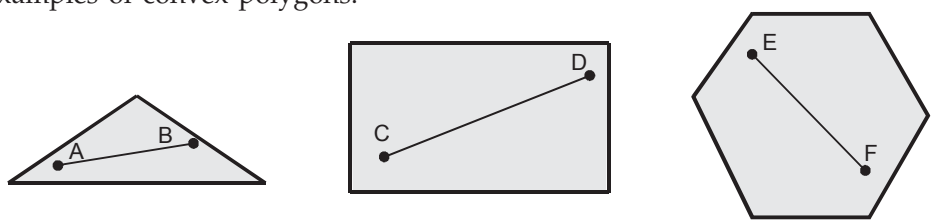


Fig. 4.2.1 Convex polygons

- A concave polygon is a polygon in which the line segment joining any two points within the polygon may not lie completely inside the polygon. Fig. 4.2.2 shows the examples of concave polygons.

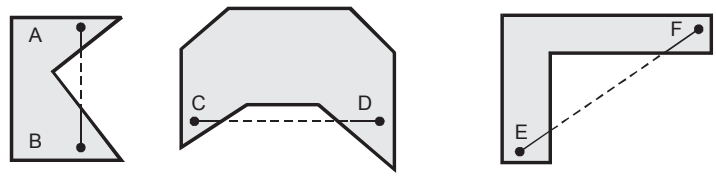


Fig. 4.2.2 Concave polygons

- A polygon is simple if it is described by a single, non-intersecting boundary; otherwise it is said to be **complex.**

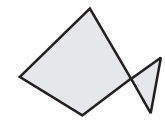


Fig. 4.2.3 Complex polygon

Review Questions

1. *With suitable diagram explain concave and convex polygons ?*

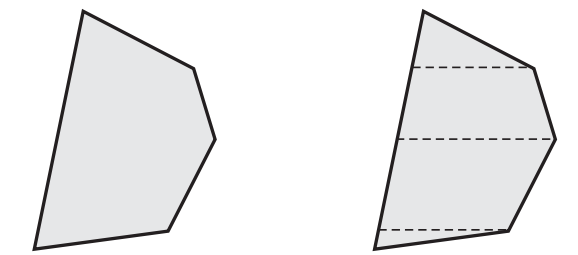
SPPU : May-05, Dec.-06,18, Marks 2

2. *What are different types of polygons ? Explain with example*

SPPU : Dec.-09, May-10, 11, 12, Marks 4

4.3 Representation of Polygon

- There are three approaches to represent polygons according to the graphics systems :
  - Polygon drawing primitive approach
  - Trapezoid primitive approach
  - Line and point approach
- Some graphics devices supports polygon drawing primitive approach. They candirectly draw the polygon shapes. On such devices polygons are saved as a unit.
- Some graphics devices support trapezoid primitive. In such devices, trapezoids are formed from two scan lines and two line segments as shown in the Fig. 4.3.1. Here, trapezoids are drawn by stepping down the line segments with two vector generators and, for each scan line, filling in all the pixels between them. Therefore every polygon is broken up into trapezoids and it is represented as a series of trapezoids.



(a) Polygon (b) Polygon as a series of trapezoids

Fig. 4.3.1 Representation of polygon

- Most of the other graphics devices do not provide any polygon support at all. In such cases polygons are represented using lines and points. A polygon is represented as a unit and it is stored in the display file. In a display file polygon can not be stored only with series of line commands because they do not specify how many of the following line commands are the part of the polygon. Therefore, new command is used in the display file to represent polygons. The opcode for new command itself specify the number of line segments in the polygon. The Fig. 4.3.2 shows the polygon and its representation using display file.

DF_OP	DF_ x	DF_ y
6	0	2
2	0	4
2	4	6
2	6	4
2	6	2
2	4	0
2	0	2

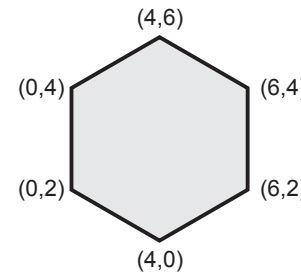


Fig. 4.3.2 Polygon and its representation using display file

Review Question

1. *Explain various approaches used to represent polygon.*

4.4 Inside Test

SPPU : May-05,07,10,13,16,19 Dec.-06,09,10,12,14,15,18

- Once the polygon is entered in the display file, we can draw the outline of the polygon.
- To show polygon as a solid object we have to set the pixels inside the polygon as well as pixels on the boundary of it.

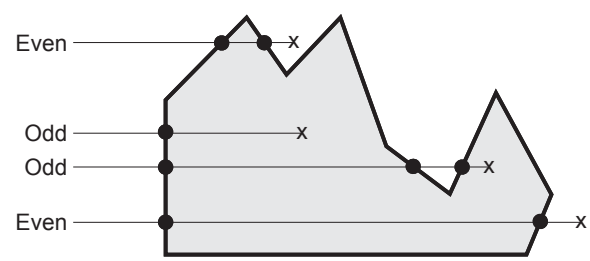


Fig. 4.4.1

- Now the question is how to determine whether or not a point is inside of a polygon. One simple method of doing this is to construct a line segment between the point in question and a point known to be outside the polygon, as shown in the Fig. 4.4.1. Now count how many intersections of the line segment with the polygon boundary occur. If there are an odd number of intersections, then the point in question is inside; otherwise it is outside. This method is called the **even-odd method** of determining polygon inside points.

- If the intersection point is vertex of the polygon then we have to look at the other endpoints of the two segments which meet at this vertex. If these points lie on the same side of the constructed line, then the point in question counts as an even number of intersections. If they lie on opposite sides of the constructed line, then the point is counted as a single intersection. This is illustrated in Fig. 4.4.2.

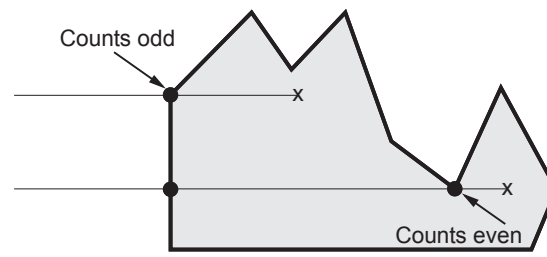


Fig. 4.4.2

- Another approach to do the inside test is the **winding-number method**. Let us consider a point in question and a point on the polygon boundary. Conceptually, we can stretch a piece of elastic between these points and slide the end point on the polygon boundary for one complete rotation. We can then examine the point in question to see how many times the elastic has wound around it. If it is wound at least once, then the point is inside. If there is no net winding, then the point is outside.
- Like even-odd method, in winding number method we have to picturise a line segment running from outside the polygon to the point in question and consider the polygon sides which it crosses. Here, instead of just counting the intersections, we have to give a **direction number** to each boundary line crossed, and we have to sum these direction numbers. The direction number indicates the direction the polygon edge was drawn relative to the line segment we have constructed for the test. This is illustrated in Fig. 4.4.3.

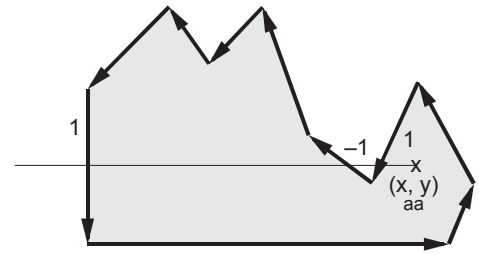


Fig. 4.4.3

- As shown in the Fig. 4.4.3, point (x_a, y_a) is a test point and line $y = y_a$ is a horizontal line runs from outside the polygon to point (x_a, y_a). The polygon edges crossed by this line could be drawn in two ways. The edge could be drawn starting below the line, cross it, and end above the line or starting above the line, cross it, and end below the line. In first case we have to give direction number as –1 and in the second case we have give direction number as 1. After giving the direction numbers we have to take sum of these direction numbers which indicates whether the point in inside the polygon or not. The sum of the direction numbers for the sides that cross the constructed horizontal line segment is called the **winding number** for the point in question. For polygons or two dimensional objects, the point is said to be inside when the value of winding number is nonzero. It is imporant to note that the line we choose must not pass through any vertices.

Review Questions

1. *Explain even-odd method to determine polygon interior points.*
   SPPU : Dec.-12,15, May-13,16, Marks 4
2. *Explain the different methods for testing a pixel inside of polygon.*
   SPPU : May-05,07,10,13, Dec.-09,10,12,14, Marks 6
3. *Explain the winding number method. Does the method supports intersection polygons?*
   SPPU : Dec.-06, Marks 4
4. *Write and explain any one inside test algorithm.* **SPPU : Dec.-18, May-19, Marks 4**
5. *Explain any one inside test algorithm.* **SPPU : May-19, Marks 2**

4.5 Polygon Filling Algorithms

SPPU : May-05,06,07,08,09,10,11,12,13,14,15,18,19, Dec.-05,06,07,08,09,10,11,12,15,19

- Filling the polygon means highlighting all the pixels which lie inside the polygon with any colour other than background colour. Polygons are easier to fill since they have linear boundaries.
- There are two basic approaches used to fill the polygon.
- One way to fill a polygon is to start from a given "seed", point known to be inside the polygon and highlight outward from this point i.e. neighbouring pixels until we encounter the boundary pixels. This approach is called **seed fill** because colour flows from the seed pixel until reaching the polygon boundary, like water flooding on the surface of the container.
- Another approach to fill the polygon is to apply the inside test i.e. to check whether the pixel is inside the polygon or outside the polygon and then highlight pixels which lie inside the polygon. This approach is known as **scan-line algorithm**. It avoids the need for a seed pixel but it requires some computation. Let us see these two methods in detail.

4.5.1 Seed Fill

- The seed fill algorithm is further classified as flood fill algorithm and boundary fill algorithm.
- Algorithms that fill interior-defined regions are called **flood-fill algorithms**; those that fill boundary-defined regions are called **boundary-fill algorithms** or **edge-fill algorithms.**

4.5.1.1 Boundary Fill Algorithm / Edge Fill Algorithm

- In this method, edges of the polygons are drawn. Then starting with some seed, any point inside the polygon we examine the neighbouring pixels to check

whether the boundary pixel is reached. If boundary pixels are not reached, pixels are highlighted and the process is continued until boundary pixels are reached.

- Boundary defined regions may be either 4-connected or 8-connected as shown in the Fig. 4.5.1.
- If a region is 4-connected, then every pixel in the region may be reached by a combination of moves in only four directions : left, right, up and down.
- For an 8-connected region every pixel in the region may be reached by a combination of moves in the two horizontal, two vertical and four diagonal directions.

(a) Four connected region (b) Eight connected region

Fig. 4.5.1

- In some cases, an 8-connected algorithm is more accurate than the 4-connected algorithm. This is illustrated in Fig. 4.5.2. Here, a 4-connected algorithm produces the partial fill.

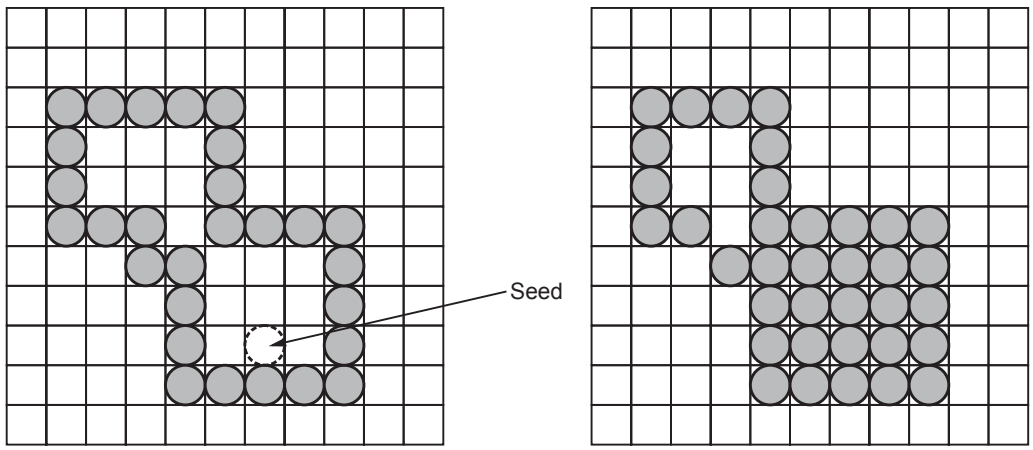


Fig. 4.5.2 Partial filling resulted using 4-connected algorithm

- The following procedure illustrates the recursive method for filling a 4-connected region with colour specified in parameter **fill colour** (f-colour) up to a boundary colour specified with parameter **boundary colour** (b-colour)

Procedure :

```
boundary_fill (x, y, f_colour, b_colour)
{
  if (getpixel (x,y) ! = b_colour && getpixel (x, y) ! = f_colour)
```

```
        {       putpixel (x, y, f_colour)
                boundary_fill (x + 1, y, f_colour, b_colour);
                boundary_fill (x, y + 1, f_colour, b_colour);
                boundary_fill (x – 1, y, f_colour, b_colour);
                boundary_fill (x, y – 1, f_colour, b_colour);
        }
}
```

Note *'getpixel' function gives the colour of specified pixel and 'putpixel' function draws the pixel with specified colour.*

- Same procedure can be modified according to 8 connected region algorithm by including four additional statements to test diagonal positions, such as (x + 1, y + 1).

4.5.1.2 Flood Fill Algorithm

- Sometimes it is required to fill in an area that is not defined within a single colour boundary. In such cases we can fill areas by replacing a specified interior colour instead of searching for a boundary colour. This approach is called a flood-fill algorithm.
- Like boundary fill algorithm, here we start with some seed and examine the neighbouring pixels. However, here pixels are checked for a specified interior colour instead of boundary colour and they are replaced by new colour.
- Using either a 4-connected or 8-connected approach, we can step through pixel positions until all interior point have been filled. The following procedure illustrates the recursive method for filling 8-connected region using flood-fill algorithm.

Procedure : flood_fill (x, y, old_colour, new_colour).

```
{
    if ( getpixel (x, y) = old_colour)
        {   putpixel (x, y, new_colour);
            flood_fill (x + 1, y, old_colour, new_colour);
            flood_fill (x – 1, y, old_colour, new_colour);
            flood_fill (x, y + 1, old_colour, new_colour);
            flood_fill (x, y – 1, old_colour, new_colour);
            flood_fill (x + 1, y + 1, old_colour, new_colour);
            flood_fill (x – 1, y – 1, old_colour, new_colour);
            flood_fill (x + 1, y – 1, old_colour, new_colour);
```

```
            flood_fill (x – 1, y + 1, old_colour, new_colour);
        }
    }
```

Note *'getpixel' function gives the colour of specified pixel and 'putpixel' function draws the pixel with specified colour.*

4.5.2 Scan Line Algorithm

- Recursive algorithm for seed fill methods have got two difficulties :
  - The first difficulty is that if some inside pixels are already displayed in fill colour then recursive branch terminates, leaving further internal pixels unfilled. To avoid this difficulty, we have to first change the colour of any internal pixels that are initially set to the fill colour before applying the seed fill procedures.
  - Another difficulty with recursive seed fill methods is that it cannot be used for large polygons. This is because recursive seed fill procedures require stacking of neighbouring points and in case of large polygons stack space may be insufficient for stacking of neighbouring points.
- To avoid this problem more efficient method can be used. Such method fills horizontal pixel spans across scan lines, instead of proceeding to 4-connected or 8-connected neighbouring points. This is achieved by identifying the rightmost and leftmost pixels of the seed pixel and then drawing a horizontal line between these two boundary pixels. This procedure is repeated with changing the seed pixel above and below the line just drawn until complete polygon is filled. With this efficient method we have to stack only a beginning position for each horizontal pixel span, instead of stacking all unprocessed neighbouring positions around the current position.
- Fig. 4.5.3 illustrates the scan line algorithm for filling of polygon.
- For each scan line crossing a polygon, this algorithm locates the intersection points of the scan line with the polygon edges. These intersection points are then sorted from left to right and the corresponding positions between each intersection pair are set to the specified fill colour.

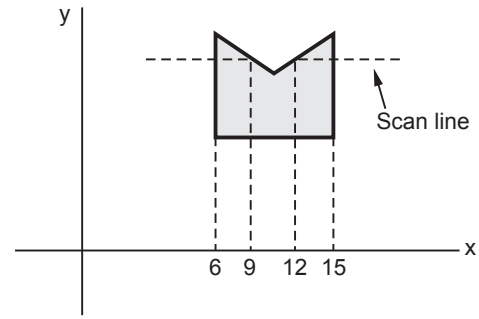


Fig. 4.5.3

- In Fig. 4.5.3, we can see that there are two stretches of interior pixels from x = 6 to x = 9 and x = 12 to x = 15.
- The scan line algorithm first finds the largest and smallest y values of the polygon. It then starts with the largest y value and works its way down, scanning from left to right, in the manner of a raster display.
- The important task in the scan line algorithm is to find the intersection points of the scan line with the polygon boundary.
- When intersection points are even, they are sorted from left to right, paired and pixels between paired points are set to the fill colour.
- In some cases intersection point is a vertex. When scan line intersects polygon vertex a special handling is required to find the exact intersection points. To handle such cases, we must look at the other endpoints of the two line segments of the polygon which meet at this vertex. If these points lie on the same (up or down) side of the scan line, then the point in question counts as an even number of intersections. If they lie on opposite sides of the scan line, then the point is counted as single intersection. This is illustrated in Fig. 4.5.4.

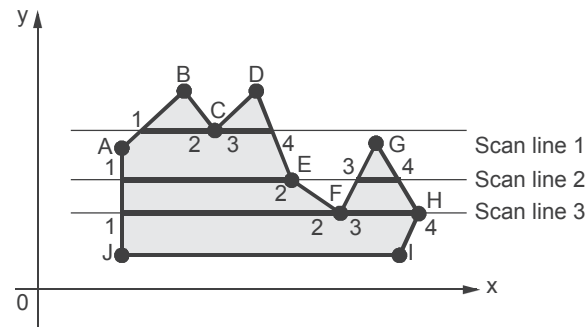


Fig. 4.5.4 Intersection points along the scan line that intersect polygon vertices

- As shown in Fig. 4.5.4, each scan line intersects the vertex or vertices of the polygon.
- For scan line 1, the other end points (B and D) of the two line segments of the polygon lie on the same side of the scan line, hence there are two intersections resulting two pairs : 1 - 2 and 3 - 4. Intersections points 2 and 3 are actually same points.
- For scan line 2 the other endpoints (D and F) of the two line segments of the polygon lie on the opposite sides of the scan line, hence there is a single intersection resulting two pairs : 1 - 2 and 3 - 4.
- For scan line 3, two vertices are the intersection points. For vertex F the other end points E and G of the two line segments of the polygon lie on the same side of the scan line whereas for vertex H, the other endpoints G and I of the two line segments of the polygon lie on the opposite side of the scan line. Therefore, at vertex F there are two intersections and at vertex H there is only one intersection.

- This results two pairs : 1 - 2 and 3 - 4 and points 2 and 3 are actually same points.
- It is necessary to calculate x intersection points for scan line with every polygon side.
- We can simplify these calculations by using **coherence properties**.
- A coherence property of a scene is a property of a scene by which we can relate one part of a scene with the other parts of a scene. Here, we can use a slope of an edge as a coherence property. By using this property we can determine the x intersection value on the lower scan line if the x intersection value for current scan line is known. This is given as

$$x_{i+1} = x_i - \frac{1}{m}$$

where m is the slope of the edge

- As we scan from top to bottom value of y coordinates between the two scan line changes by 1.

$$y_{i+1} = y_i - 1$$

- Many times it is not necessary to compute the x intersections for scan line with every polygon side.
- We need to consider only the polygon sides with endpoints straddling the current scan line. See Fig. 4.5.5.
- It will be easier to identify which polygon sides should be tested for x-intersection, if we first sort the sides in order of their maximum y value.

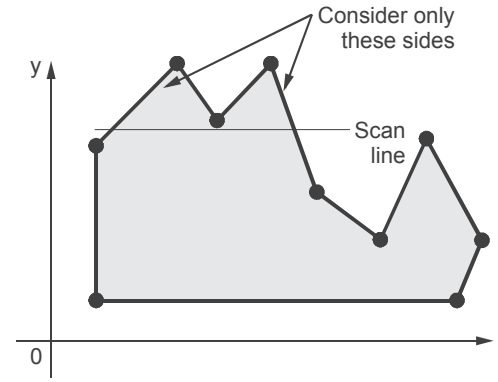


Fig. 4.5.5 Consider only the sides which intersect the scan line

- Once the sides are sorted we can process the scan lines from the top of the polygon to its bottom producing an active edge list for each scan line crossing the polygon boundaries.
- The active edge list for a scan line contains all edges crossed by that scan line.
- Fig. 4.5.6 shows sorted edges of the polygon with active edges.
- A scan line algorithm for filling a polygon begins by ordering the polygon sides on the largest y value.
- It begins with the largest y value and scans down the polygon.

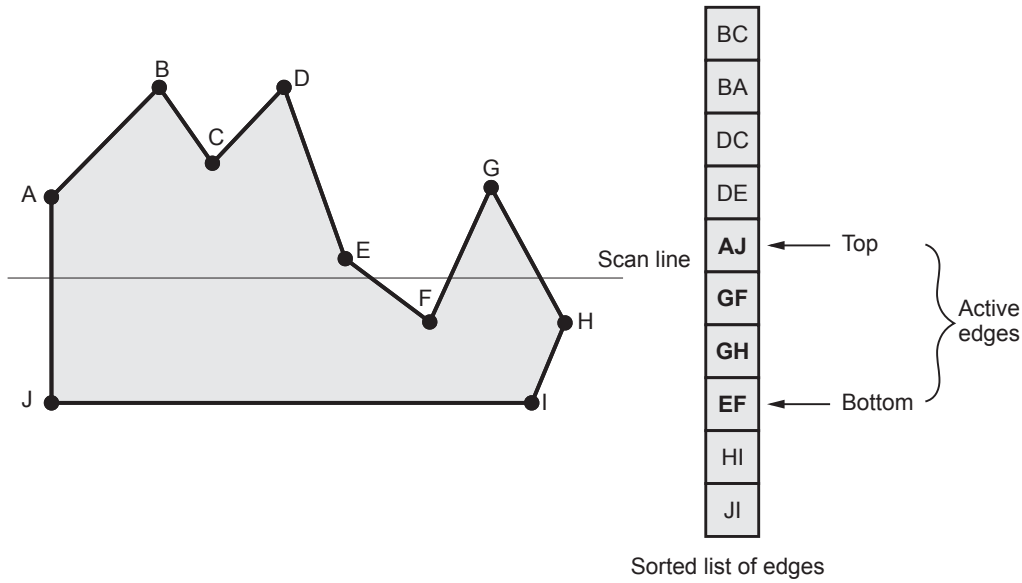


Fig. 4.5.6 Sorted edges of the polygon with active edges

- For each y, it determines which sides can be intersected and finds the x values of these intersection points.
- It then sorts, pairs and passes these x values to a line drawing routine.

Scan Line Conversion Algorithm for Polygon Filling :

1. Read n, the number of vertices of polygon
2. Read x and y coordinates of all vertices in array x[n] and y[n].
3. Find y_{min} and y_{max}.
4. Store the initial x value (x_1) y values y_1 and y_2 for two endpoints and x increment Δx from scan line to scan line for each edge in the array edges [n] [4].

   While doing this check that $y_1 > y_2$, if not interchange y_1 and y_2 and corresponding x_1 and x_2 so that for each edge, y_1 represents its maximum y coordinate and y_2 represents its minimum y coordinate.
5. Sort the rows of array, edges [n] [4] in descending order of y_1, descending order of y_2 and ascending order of x_2.
6. Set $y = y_{max}$
7. Find the active edges and update active edge list :

   if ($y > y_2$ and $y \leq y_1$)

   { edge is active }

   else

   { edge is not active }

8. Compute the x intersects for all active edges for current y value [initially x-intersect is x_1 and x intersects for successive y values can be given as

$$x_{i+1} \leftarrow x_i + \Delta x$$

where $\Delta x = -\frac{1}{m}$ and $m = \frac{y_2 - y_1}{x_2 - x_1}$ i.e. slope of a line segment

9. If x intersect is vertex i.e. x-intersect = x_1 and $y = y_1$ then apply vertex test to check whether to consider one intersect or two intersects. Store all x intersects in the x-intersect [ ] array.
10. Sort x-intersect [ ] array in the ascending order,
11. Extract pairs of intersects from the sorted x-intersect [ ] array.
12. Pass pairs of x values to line drawing routine to draw corresponding line segments
13. Set $y = y - 1$
14. Repeat steps 7 through 13 until $y \geq y_{min}$.
15. Stop

In step 7, we have checked for $y \leq y_1$ and not simply $y < y_1$. Hence step 9 a becomes redundant. Following program takes care of that.

Features of scan line algorithm

1. Scan line algorithm is used for filling of polygons.
2. This algorithm solves the problem of hidden surfaces while generating display scan line.
3. It is used in orthogonal projection.
4. It is non recursive algorithm.
5. In scan line algorithm we have to stack only a beginning position for each horizontal pixel scan, instead of stacking all unprocessed neighbouring positions around the current position. Therefore, it is efficient algorithm.

Review Questions

1. *What is polygon filling.* **SPPU : May-18, Dec.-19, Marks 2**
2. *Explain boundary-fill/edge-fill algorithm for polygon.* **SPPU : May-05,09, Dec.-19, Marks 6**
3. *Explain polygon filling by seed-fill algorithm.* **SPPU : Dec.-05,08, May-08,18, Marks 6**
4. *What are the steps involved in filling polygon in scan line method?* **SPPU : Dec.-05,06,08, May-06,07,08,18,19, Marks 8**
5. *Explain with example and compare seed-fill and edge-fill algorithm for polygon.* **SPPU : May-06,10, Dec.-07, 09, Marks 8**
6. *Enlist any three polygon filling algorithms.* **SPPU : May-13, Marks 4**

7. *Explain how a polygon is filled with pattern.* **SPPU : Dec.-12, May-13, Marks 4**
8. *List various polygon filling algorithms. Explain scan line algorithm with mathematical formulation* **SPPU : May-11, Marks 16**
9. *Compare different polygon filling algorithm.* **SPPU : Dec.-12, Marks 4**
10. *State the characteristics of scan line polygon fill algorithm and compare it with boundary fill algorithm.* **SPPU : May-09, 10, Marks 8**
11. *Explain Scanline algorithm for polygon filling and explain how it can be extended for hidden removal.* **SPPU : Dec.-10, Marks 10**
12. *Explain scan line algorithm with example.* **SPPU : May-14,19, Marks 6**
13. *What is scan line polygon filling algorithm ? Explain with an example.* **SPPU : May-12, Marks 10**
14. *Describe scan line algorithm to generate solid area on the screen.* **SPPU : Dec.-11, Marks 8**
15. *Write algorithm to fill the polygon area using flood fill method.* **SPPU : May-15, Marks 4**
16. *Write flood fill algorithm.* **SPPU : Dec.-15, Marks 3**

❑❑❑

UNIT - II

5 Windowing and Clipping

Syllabus

Viewing transformations, 2-D clipping: Cohen- Sutherland algorithm line Clipping algorithm, Sutherland Hodgeman Polygon clipping algorithm, Weiler Atherton Polygon Clipping algorithm.

Contents

5.1 Introduction

SPPU : May-05,13, Dec.-05,12

- We have to identify the visible part of the picture for inclusion in the display image. This selection process is not straight forward. Certain lines may lie partly inside the visible portion of the picture and partly outside. These lines cannot be omitted entirely from the display image because the image would become inaccurate. This is illustrated in Fig. 5.1.1.
- The process of selecting and viewing the picture with different views is called **windowing**, and a process which divides each element of the picture into its visible and invisible portions, allowing the invisible portion to be discarded is called **clipping**.

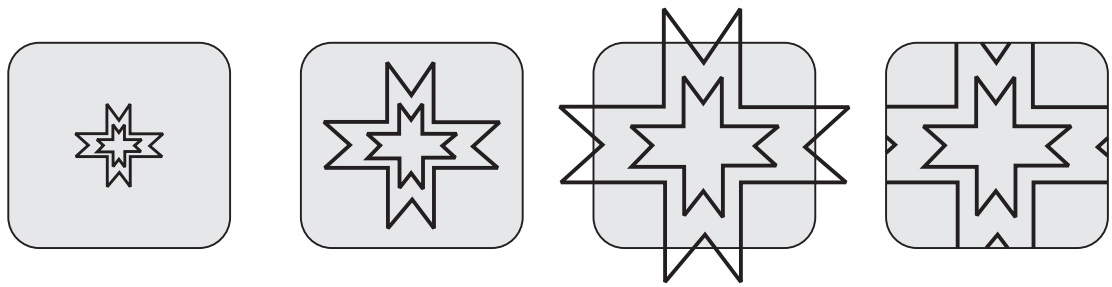


Fig. 5.1.1 Concept of windowing and clipping

Review Questions

1. *What is window ?* **SPPU : May-05, Marks 2**
2. *What is windowing and clipping ?* **SPPU : May-13, Dec.-05, 12, Marks 4**

5.2 Viewing Transformations

SPPU : May-05,12, Dec.-10,11,19

- The picture is stored in the computer memory using any convenient Cartesian co-ordinate system, referred to as **World Co-ordinate System** (WCS).
- When picture is displayed on the display device it is measured in **Physical Device Co-ordinate System** (PDCS) corresponding to the display device. Therefore, displaying an image of a picture involves mapping the co-ordinates of the points and lines that form the picture into the appropriate physical device co-ordinate where the image is to be displayed. This mapping of co-ordinates is achieved with the use of co-ordinate transformation known as **viewing transformation**.
- Sometimes the two dimensional viewing transformation is simply referred to as the **window to view port transformation** or the **windowing transformation**.
- The viewing transformation which maps picture co-ordinates in the WCS to display co-ordinates in PDCS is performed by the following transformations.

- Converting world co-ordinates to viewing co-ordinates.
- Normalizing viewing co-ordinates.
- Converting normalized viewing coordinates to device co-ordinates

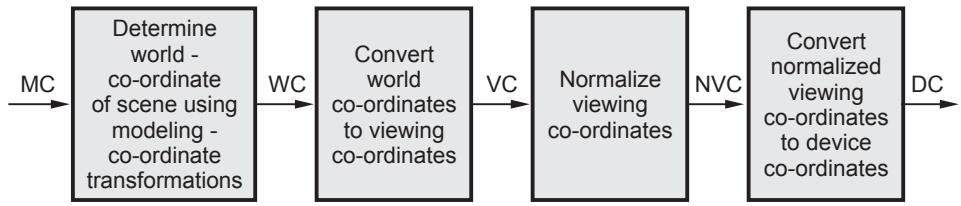
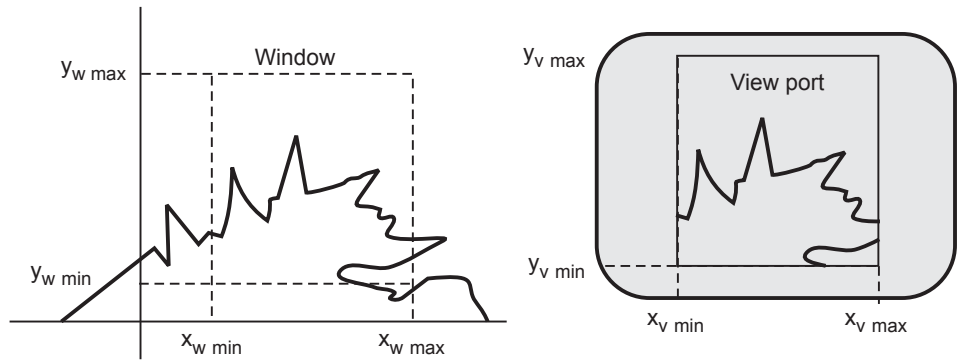


Fig. 5.2.1 (a) Two-dimensional viewing transformation pipeline

- World Co-ordinate System (WCS) is infinite in extent and the device display area is finite.
- To perform a viewing transformation we select a finite world co-ordinate area for display called a **window**.
- An area on a device to which a window is mapped is called a **viewport**.
- The window defines what is to be viewed; the viewport defines where it is to be displayed, as shown in the Fig. 5.2.1 (b).



World co-ordinates **Device co-ordinates**

Fig. 5.2.1 (b) Window and viewport

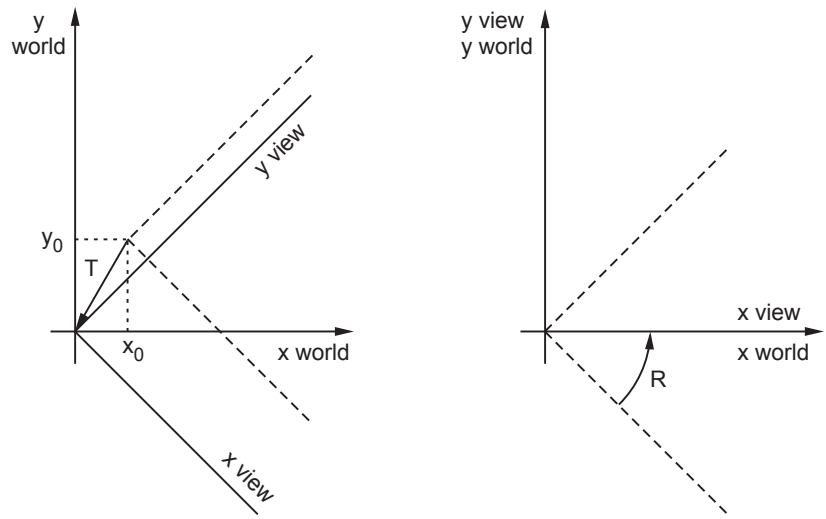
The steps involved in viewing transformations :

1. Construct the scene in world co-ordinates using the output primitives and attributes.
2. Obtain a particular orientation for the window by setting a two-dimensional viewing co-ordinate system in the world-co-ordinate plane and define a window in the viewing co-ordinate system.
3. Use viewing co-ordinates reference frame to provide a method for setting up arbitrary orientations for rectangular windows.
4. Once the viewing reference frame is established, transform descriptions in world co-ordinates to viewing co-ordinates.

5. Define a view port in normalized co-ordinates and map the viewing co-ordinate description of the scene to normalized co-ordinates.
6. Clip all the parts of the picture which lie outside the viewport.

5.2.1 Viewing Co-ordinate Reference Frame

- Fig. 5.2.1 (b) shows the two dimensional viewing pipeline.
- The window defined in world co-ordinate is first transformed into the viewport co-ordinate. For this, viewing co-ordinate reference frame is used. It provides a method for setting up arbitrary orientations for rectangular windows.
- To obtain the matrix for converting world co-ordinate positions to viewing co-ordinates, we have to translate the viewing origin to the world origin and then align the two co-ordinate reference frames by rotation. This is illustrated in Fig. 5.2.2.
- As shown in the Fig. 5.2.2, the composite transformation (translation and rotation) converts world co-ordinates to viewing co-ordinates.



a) Translation of the viewing origin to the world origin.

b) Rotation to align the axes of the two systems.

Fig. 5.2.2

- The matrix used for this composite transformation is given by

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

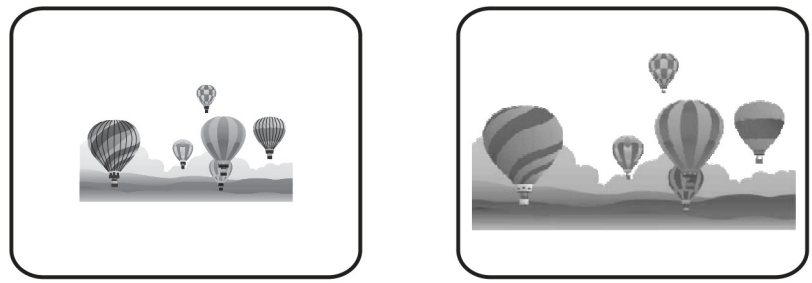
where T is the translation matrix that takes the viewing origin point P_0 to the world origin, and R is the rotation matrix that aligns the axes of the two reference frames.

5.2.2 Transformation to Normalized Co-ordinates

- Once the viewing reference frame is established, we can transform, descriptions in word co-ordinate to viewing co-ordinates. We then define a viewport in normalized co-ordinate.

Normalized Co-ordinates

- The different display devices may have different screen sizes as measured in pixels.
- Size of the screen in pixels increases as resolution of the screen increases.
- When picture is defined in the pixel values then it is displayed large in size on the low resolution screen while small in size on the high resolution screen as shown in the Fig. 5.2.3.



(a) More resolution (b) Less resolution

Fig. 5.2.3 Picture definition in pixels

- To avoid this and to make our programs to be device independent, we have to define the picture co-ordinates in some units other than pixels and use the interpreter to convert these co-ordinates to appropriate pixel values for the particular display device. The device independent units are called the **normalized device co-ordinates**. In these units, the screen measures 1 unit wide and 1 unit length as shown in the Fig. 5.2.4.
- The lower left corner of the screen is the origin, and the upper-right corner is the point (1, 1).
- The point (0.5, 0.5) is the center of the screen no matter what the physical dimensions or resolution of the actual display device may be.
- The interpreter uses a simple linear formula to convert the normalized device co-ordinates to the actual device co-ordinates.

$$x = x_n \times X_W \quad \ldots (5.2.1)$$

$$y = y_n \times Y_H \quad \ldots (5.2.2)$$

where

x : Actual device x co-ordinate.

y : Actual device y co-ordinate.
x_n : Normalized x co-ordinate.
y_n : Normalized y co-ordinate.
X_W : Width of actual screen in pixels.
Y_H : Height of actual screen in pixels.

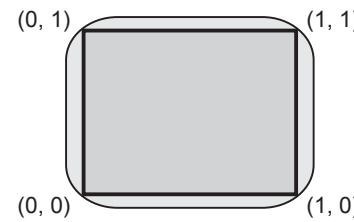


Fig. 5.2.4 Picture definition in normalized device co-ordinates

- The transformation which map the viewing co-ordinate to normalized device co-ordinate is called **normalization transformation**. It involves scaling of x and y, thus it is also referred to as **scaling transformation**.

5.2.3 Window to Viewport Co-ordinate Transformation

- Once the window co-ordinates are transferred to viewing co-ordinates we choose the window extents in viewing co-ordinates and select the viewport limits in normalized co-ordinates. Object descriptions are then transferred to normalize device co-ordinates. In this transformation, the relative placement of the object in the normalized co-ordinates is same as in the viewing co-ordinates.
- Fig. 5.2.5 shows the mapping of object from window to viewport.
- A point at position (x_w, y_w) in the window is mapped into position (x_v, y_v) in the associated viewport.

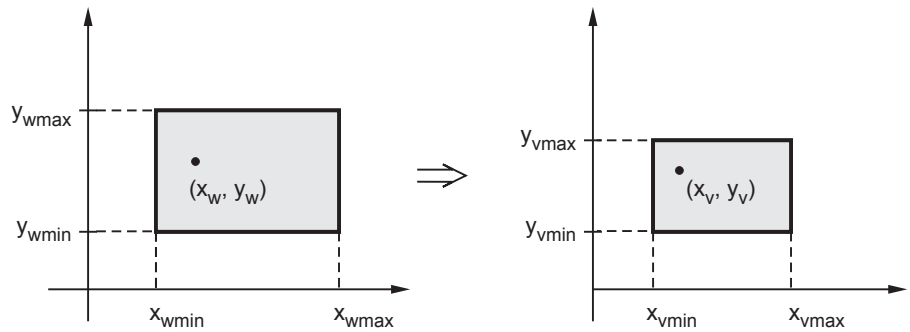


Fig. 5.2.5 Window to viewport mapping

- To maintain the same relative placement in the viewport as in the window, we require that

$$\frac{x_v - x_{vmin}}{x_{vmax} - x_{vmin}} = \frac{x_w - x_{wmin}}{x_{wmax} - x_{wmin}}$$

$$\frac{y_v - y_{vmin}}{y_{vmax} - y_{vmin}} = \frac{y_w - y_{wmin}}{y_{wmax} - y_{wmin}}$$

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

$$x_v = x_{vmin} + (x_w - x_{wmin})\, s_x$$

$$y_v = y_{vmin} + (y_w - y_{wmin})\, s_y$$

where scaling factors are

$$s_x = \frac{x_{vmax} - x_{vmin}}{x_{wmax} - x_{wmin}}$$

$$s_y = \frac{y_{vmax} - y_{vmin}}{y_{wmax} - y_{wmin}}$$

- The mapping of window to viewport can also be achieved by performing following transformations :
  1. By performing scaling transformation using a fixed-point position of (x_{wmin}, y_{wmin}) that scales the window area to the size of the viewport area.
  2. By translating the scaled window area to the position of the viewport area.
- While performing these transformations scaling factors s_x and s_y are kept same to maintain the relative proportions of objects.
- The character strings are transformed in two ways; they are either transformed without any change or if they are formed with line segments, they are transformed as a sequence of line transformations.

Workstation Transformation

- The transformation of object description from normalized co-ordinates to device co-ordinates is called **workstation transformation**.
- The workstation transformation is accomplished by selecting a window area in normalized space and a viewport area in the co-ordinates of the display device.
- By using workstation transformation we can partition a view so that different parts of normalized space can be displayed on different output devices, as shown in the Fig. 5.2.6. (See Fig. 5.2.6 on next page).
- The workstation transformation is achieved by performing following steps :
  1. The object together with its window is translated until the lower left corner of the window is at the origin.
  2. Object and window are scaled until the window has the dimensions of the viewport.
  3. Translate the viewport to its correct position on the screen.

This is illustrated in Fig. 5.2.7. (See Fig. 5.2.7 on page 5 - 9)

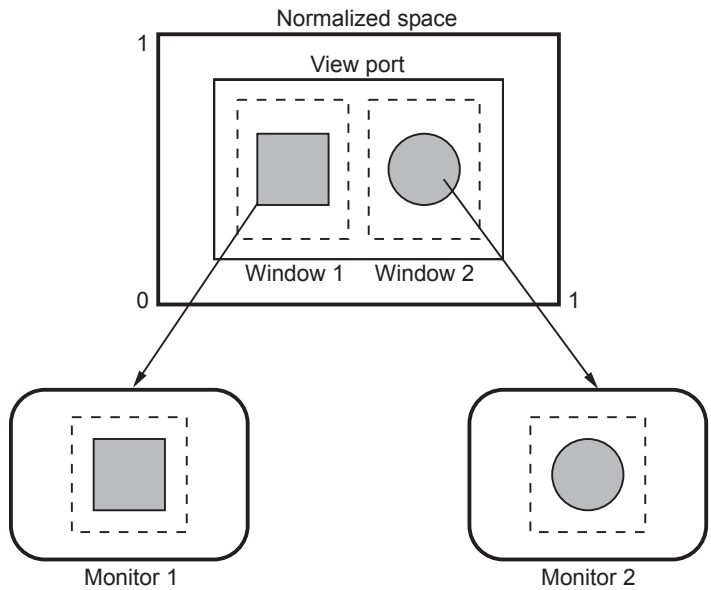


Fig. 5.2.6 Mapping selected portion of the scene in normalized co-ordinates to different monitors with workstation transformations

- The workstation transformation is given as

$$W = T \cdot S \cdot T^{-1} \quad \ldots (5.2.3)$$

- The transformation matrices for individual transformation are as given below :

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -X_{w\,min} & -y_{w\,min} & 1 \end{bmatrix}$$

$$S = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{where} \quad S_x = \frac{x_{v\,max} - x_{v\,min}}{x_{w\,max} - x_{w\,min}}$$

$$S_y = \frac{y_{vmax} - y_{vmin}}{y_{wmax} - y_{wmin}}$$

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_{v\,min} & y_{vmin} & 1 \end{bmatrix}$$

- The overall transformation matrix for W is given as

$$W = T \cdot S \cdot T^{-1}$$

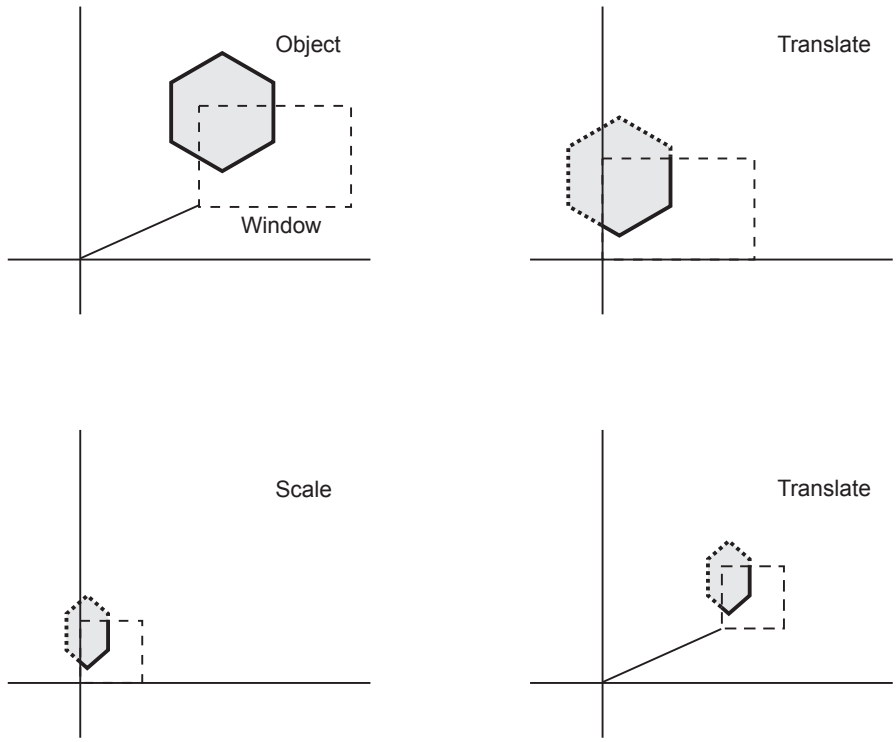


Fig. 5.2.7 Steps in workstation transformation

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -X_{w\,min} & -y_{w\,min} & 1 \end{bmatrix} \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_{v\,min} & y_{v\,min} & 1 \end{bmatrix}$$

$$= \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ x_{vmin} - x_{w\,min}.S_x & y_{v\,min} - y_{w\,min}.S_y & 1 \end{bmatrix}$$

- Fig. 5.2.8 shows the complete viewing transformation.

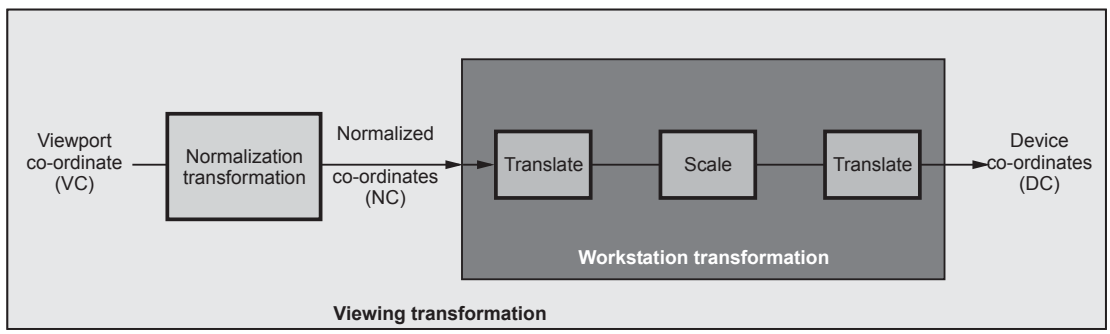


Fig. 5.2.8 Viewing transformation

Applications

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

Example 5.2.1 *Find the normalization transformation window to viewpoint, with window, lower left corner at (1, 1) and upper right corner at (3, 5) onto a viewpoint with lower left corner at (0, 0) and upper right corner at (1/2, 1/2).*

Solution : Given : Co-ordinates for window

$$x_{w\,min} = 1 \qquad y_{w\,min} = 1$$

$$x_{w\,max} = 3 \qquad y_{w\,max} = 5$$

Co-ordinates for view port

$$x_{v\,min} = 0 \qquad y_{v\,min} = 0$$

$$x_{v\,max} = 1/2 = 0.5 \qquad y_{v\,max} = 1/2 = 0.5$$

We know that,

$$S_x = \frac{x_{v\,max} - x_{v\,min}}{x_{w\,max} - x_{w\,min}} = \frac{0.5-0}{3-1} = 0.25$$

and

$$S_y = \frac{y_{v\,max} - y_{v\,min}}{y_{w\,max} - y_{w\,min}} = \frac{0.5-0}{5-1} = 0.125$$

We know that transformation matrix is given as,

$$T \cdot S \cdot T^{-1} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ x_{v\,min} - x_{w\,min} S_x & y_{v\,min} - y_{w\,min} S_y & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0.25 & 0 & 0 \\ 0 & 0.125 & 0 \\ 0-(1\times 0.25) & 0-(1\times 0.125) & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0.25 & 0 & 0 \\ 0 & 0.125 & 0 \\ -0.25 & -0.125 & 1 \end{bmatrix}$$

Example 5.2.2 *Find the normalization transformation N that uses the rectangle A (1, 1), B (5, 3), C (4, 5), D (0, 3) as a window and the normalized device screen as a viewport.*

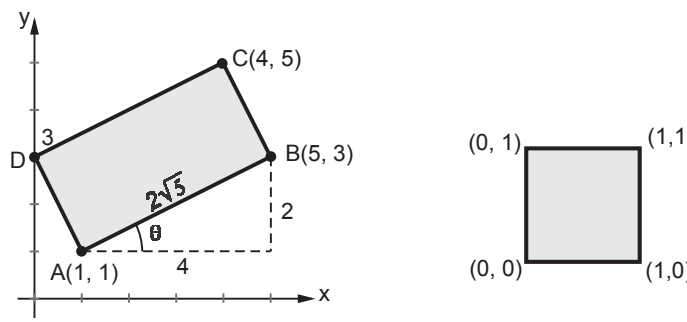
Solution : To align rectangle about A we have to rotate it with the co-ordinate axes. Then we can calculate, S_x and S_y and finally we compose the rotation and the transformation N to find the required normalization transformation N_R.

The slope of the line segment $\overline{AB}$ is,

$$m = \frac{3-1}{5-1} = \frac{1}{2}$$

The Fig. 5.2.9 (a) shows the specified window and viewport. As shown in the Fig. 5.2.9 (a), the angle of rotation is $-\theta$ and it is given by,

$$\tan\theta = \frac{2}{4} = \frac{1}{2}$$



(a) Window **(b) Viewport**

Fig. 5.2.9

Thus,

$$\sin\theta = \frac{1}{\sqrt{5}} \text{ and so } \sin(-\theta) = -\frac{1}{\sqrt{5}}, \quad \cos\theta = \frac{2}{\sqrt{5}}, \quad \cos(-\theta) = \frac{2}{\sqrt{5}}$$

Referring section 3.7.1.

The rotation matrix about A (1, 1) is given as,

$$R_{-\theta, A} = \begin{pmatrix} \frac{2}{\sqrt{5}} & \frac{-1}{\sqrt{5}} & 0 \\ \frac{1}{\sqrt{5}} & \frac{2}{\sqrt{5}} & 0 \\ \left(1-\frac{3}{\sqrt{5}}\right) & \left(1-\frac{1}{\sqrt{5}}\right) & 1 \end{pmatrix}$$

The x extent of the rotated window is the length of $\overline{AB}$. Similarly, the y extent is the length of $\overline{AD}$. Using the distance formula we can calculate these length as,

$$d\ (A, B) = \sqrt{2^2 + 4^2} = \sqrt{20} = 2\sqrt{5}$$

$$d\ (A, D) = \sqrt{1^2 + 2^2} = \sqrt{5}$$

Also, the x extent of the normalized device screen is 1, as is the y extent. Calculating s_x and s_y,

$$s_x = \frac{\text{Viewport x extent}}{\text{Window x extent}} = \frac{1}{2\sqrt{5}}$$

$$s_y = \frac{\text{Viewport y extent}}{\text{Window y extent}} = \frac{1}{\sqrt{5}}$$

We have, $$N = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ -s_x\ x\,W_{min} + x\,V_{min} & -s_y\ y\,W_{min} + y\,V_{min} & 1 \end{bmatrix}$$

where $x\,W_{min} = 1$, $x\,V_{min} = 0$, $y\,W_{min} = 1$ and $y\,V_{min} = 0$

$$= \begin{bmatrix} \frac{1}{2\sqrt{5}} & 0 & 0 \\ 0 & \frac{1}{\sqrt{5}} & 0 \\ \frac{-1}{2\sqrt{5}} \times 1 + 0 & \frac{-1}{\sqrt{5}} \times 1 + 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2\sqrt{5}} & 0 & 0 \\ 0 & \frac{1}{\sqrt{5}} & 0 \\ \frac{-1}{2\sqrt{5}} & \frac{-1}{\sqrt{5}} & 1 \end{bmatrix}$$

The normalization transformation is then,

$$N_R = R_{-\theta A \cdot N} = \begin{bmatrix} \frac{2}{\sqrt{5}} & -\frac{1}{\sqrt{5}} & 0 \\ \frac{1}{\sqrt{5}} & \frac{2}{\sqrt{5}} & 0 \\ \left(1 - \frac{3}{\sqrt{5}}\right) & \left(1 - \frac{1}{\sqrt{5}}\right) & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{2\sqrt{5}} & 0 & 0 \\ 0 & \frac{1}{\sqrt{5}} & 0 \\ \frac{-1}{2\sqrt{5}} & \frac{-1}{\sqrt{5}} & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{1}{5} & -\frac{1}{5} & 0 \\ \frac{1}{10} & \frac{2}{5} & 0 \\ \frac{-3}{10} & \frac{-1}{5} & 1 \end{bmatrix}$$

Example 5.2.3 *Find the complete viewing transformation that maps a window in world coordinates with x extent 1 to 10 and y extent 1 to 10 onto a viewport with x extent 1/4 to 3/4 and y extent 0 to 1/2 in normalized device space and then maps a workstation window with x extent 1/4 to 1/2 and y extent 1/4 to 1/2 in the normalized device space into a workstation viewport with x extent 1 to 10 and y extent 1 to 10 on the physical display device.*

Solution : For normalization transformation

$$T = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ -s_x\, x_{wmin} + x_{vmin} & -s_y\, y_{wmin} + y_{vmin} & 1 \end{bmatrix}$$

The given parameters are $x_{wmin} = 1$, $x_{wmax} = 10$, $y_{wmin} = 1$, $y_{wmax} = 10$, $x_{vmin} = 1/4$, $x_{vmax} = 3/4$, $y_{vmin} = 0$, and $y_{vmax} = \frac{1}{2}$

$$\therefore \quad s_x = \frac{\text{Viewport x extent}}{\text{Window x extent}} = \frac{\frac{3}{4} - \frac{1}{4}}{10 - 1} = \frac{\frac{1}{2}}{9} = \frac{1}{18}$$

$$s_y = \frac{\text{Viewport y extent}}{\text{Window y extent}} = \frac{\frac{1}{2} - 0}{10 - 1} = \frac{\frac{1}{2}}{9} = \frac{1}{18}$$

Substituting values of S_x and S_y we have

$$T_1 = \begin{bmatrix} \frac{1}{18} & 0 & 0 \\ 0 & \frac{1}{18} & 0 \\ -\frac{1}{18} + \frac{1}{4} & -\frac{1}{18} + 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{18} & 0 & 0 \\ 0 & \frac{1}{18} & 0 \\ \frac{7}{36} & -\frac{1}{18} & 1 \end{bmatrix}$$

The given parameters for workstation transformation are $x_{wmin} = \frac{1}{4}$, $x_{wmax} = \frac{1}{2}$, $y_{wmin} = \frac{1}{4}$, $y_{wmax} = \frac{1}{2}$, $x_{vmin} = 1$, $x_{vmax} = 10$, $y_{vmin} = 1$ and $y_{vmax} = 10$.

$$s_x = \frac{10 - 1}{\frac{1}{2} - \frac{1}{4}} = \frac{9}{\frac{1}{4}} = 36$$

$$s_y = \frac{10 - 1}{\frac{1}{2} - \frac{1}{4}} = \frac{9}{\frac{1}{4}} = 36$$

$$\therefore \quad T_2 = \begin{bmatrix} 36 & 0 & 0 \\ 0 & 36 & 0 \\ -36 \cdot \frac{1}{4} + 1 & -36 \cdot \frac{1}{4} + 1 & 1 \end{bmatrix} \begin{bmatrix} 36 & 0 & 0 \\ 0 & 36 & 0 \\ -8 & -8 & 1 \end{bmatrix}$$

The complete viewing transformation will be $T = T_1 \times T_2$

$$T = \begin{bmatrix} \frac{1}{18} & 0 & 0 \\ 0 & \frac{1}{18} & 0 \\ \frac{7}{36} & -\frac{1}{18} & 1 \end{bmatrix} \begin{bmatrix} 36 & 0 & 0 \\ 0 & 36 & 0 \\ -8 & -8 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ -1 & -10 & 1 \end{bmatrix}$$

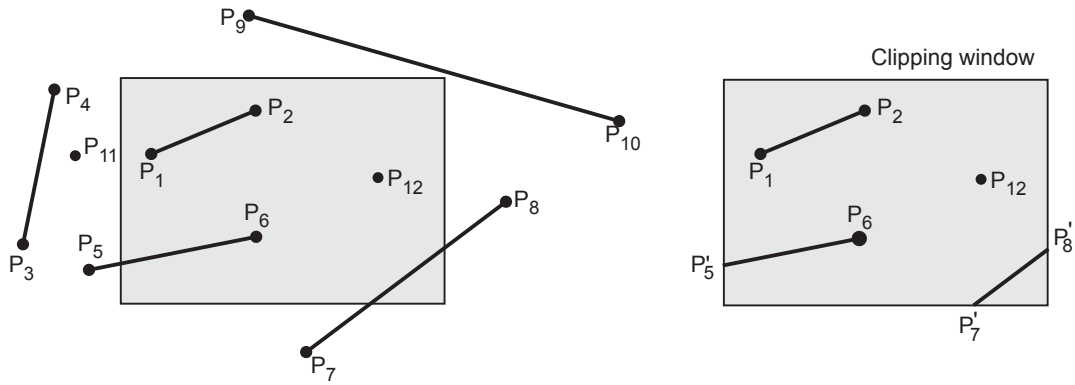
Review Questions

1. *What is window and viewport ?* **SPPU : May-05, Marks 4**
2. *State the application of viewing transformation.* **SPPU : May-05, Marks 4**
3. *Describe viewing transformation.* **SPPU : Dec.-10, 11,19, May-12, Marks 8**

5.3 2 D Clipping

SPPU : May-07, 12

- The procedure that identifies the portions of a picture that are either inside or outside of a specified region of space is referred to as clipping.
- The region against which an object is to be clipped is called a **clip window** or **clipping window.** It usually is in a rectangular shape, as shown in the Fig. 5.3.1.
- The clipping algorithm determines which points, lines or portions of lines lie within the clipping window. These points, lines or portions of lines are retained for display. All others are discarded.



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

Fig. 5.3.1

5.3.1 Point Clipping

- The points are said to be interior to the clipping window if

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

The equal sign indicates that points on the window boundary are included within the window.

5.3.2 Line Clipping

- The lines are said to be interior to the clipping window and hence visible if both end points are interior to the window, e.g. line P_1 P_2 in Fig. 5.3.1.
- If both end points of a line are exterior to the window, the line is not necessarily completely exterior to the window, e.g. line P_7 P_8 in Fig. 5.3.1.
- If both end points of a line are completely to the right of, completely to the left of, completely above, or completely below the window, then the line is completely exterior to the window and hence invisible. For example, line P_3 P_4 in Fig. 5.3.1.
- The lines which across one or more clipping boundaries require calculation of multiple intersection points to decide the visible portion of them.
- To minimize the intersection calculations and to increase the efficiency of the clipping algorithm, initially, completely visible and invisible lines are identified and then the intersection points are calculated for remaining lines.
- There are many line clipping algorithms. Let us discuss a few of them.

Review Questions

1. *What is point clipping*
2. *What is line clipping ?* **SPPU : May-12, Marks 2**

5.4 Cohen-Sutherland Line Clipping Algorithm

SPPU : May-06,07,08,10,11,12,13,15,16,17,19, Dec.-10,15,18

- This is one of the oldest and most popular line clipping algorithm developed by Dan Cohen and Ivan Sutherland.
- To speed up the processing this algorithm performs initial tests that reduce the number of intersections that must be calculated.
- This algorithm uses a four digit (bit) code to indicate which of nine regions contain the end point of line.
- The four bit codes are called **region codes** or **outcodes**. These codes identify the location of the point relative to the boundaries of the clipping rectangle as shown in the Fig. 5.4.1.
- Each bit position in the region code is used to indicate one of the four relative co-ordinate positions of the point with respect to the clipping window : to the left, right, top or

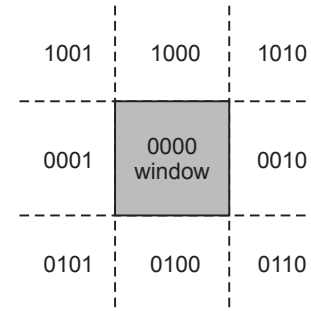


Fig. 5.4.1 Four-bit codes for nine regions

bottom. The rightmost bit is the first bit and the bits are set to 1 based on the following scheme :

- Set Bit 1 - if the end point is to the **left** of the window
- Set Bit 2 - if the end point is to the **right** of the window
- Set Bit 3 - if the end point is **below** the window
- Set Bit 4 - if the end point is **above** the window

Otherwise, the bit is set to zero.

- Once we have established region codes for all the line endpoints, we can determine which lines are completely inside the clipping window and which are clearly outside.
- Any lines that are completely inside the window boundaries have a region code of 0000 for both endpoints and we trivially accept these lines.
- Any lines that have a 1 in the same bit position in the region codes for each endpoint are completely outside the clipping rectangle, and we trivially reject these lines.
- A method used to test lines for total clipping is equivalent to the logical **AND** operator.
- If the result of the logical AND operation with two end point codes is not 0000, the line is completely outside the clipping region.
- The lines that cannot be identified as completely inside or completely outside a clipping window by these tests are checked for intersection with the window boundaries.

Example 5.4.1 *Consider the clipping window and the lines shown in Fig. 5.4.2. Find the region codes for each end point and identify whether the line is completely visible, partially visible or completely invisible.*

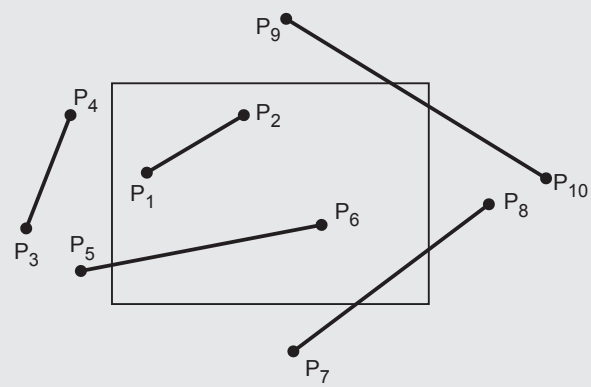


Fig. 5.4.2

Solution : The Fig. 5.4.3 shows the clipping window and lines with region codes. These codes are tabulated and end point codes are logically ANDed to identify the visibility of the line in Table 5.4.1.

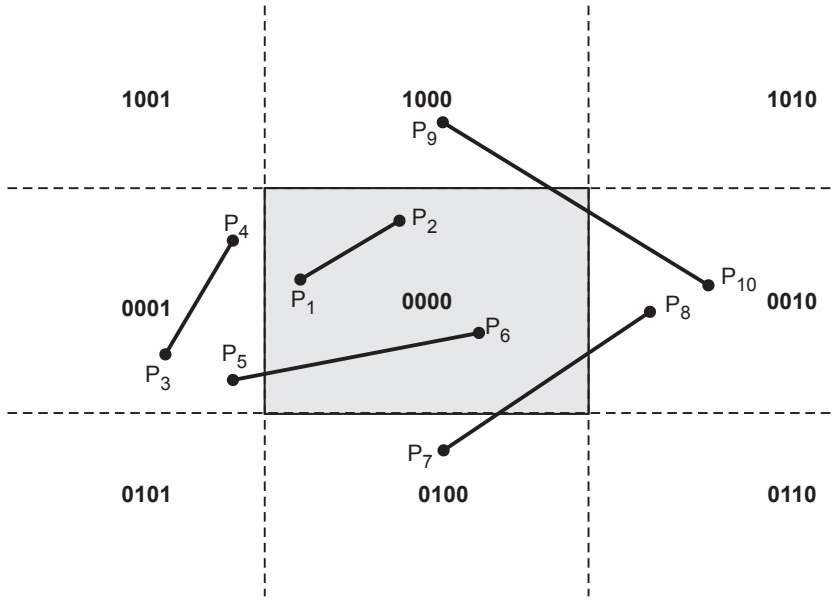


Fig. 5.4.3

Line	End point codes		Logical ANDing	Result
P_1 P_2	0000	0000	0000	Completely visible
P_3 P_4	0001	0001	0001	Completely invisible
P_5 P_6	0001	0000	0000	Partially visible
P_7 P_8	0100	0010	0000	Partially visible
P_9 P_{10}	1000	0010	0000	Partially visible

Table 5.4.1

- The Cohen-Sutherland algorithm begins the clipping process for a partially visible line by comparing an outside endpoint to a clipping boundary to determine how much of the line can be discarded. Then the remaining part of the line is checked against the other boundaries, and the process is continued until either the line is totally discarded or a section is found inside the window.
  This is illustrated in Fig. 5.4.4.

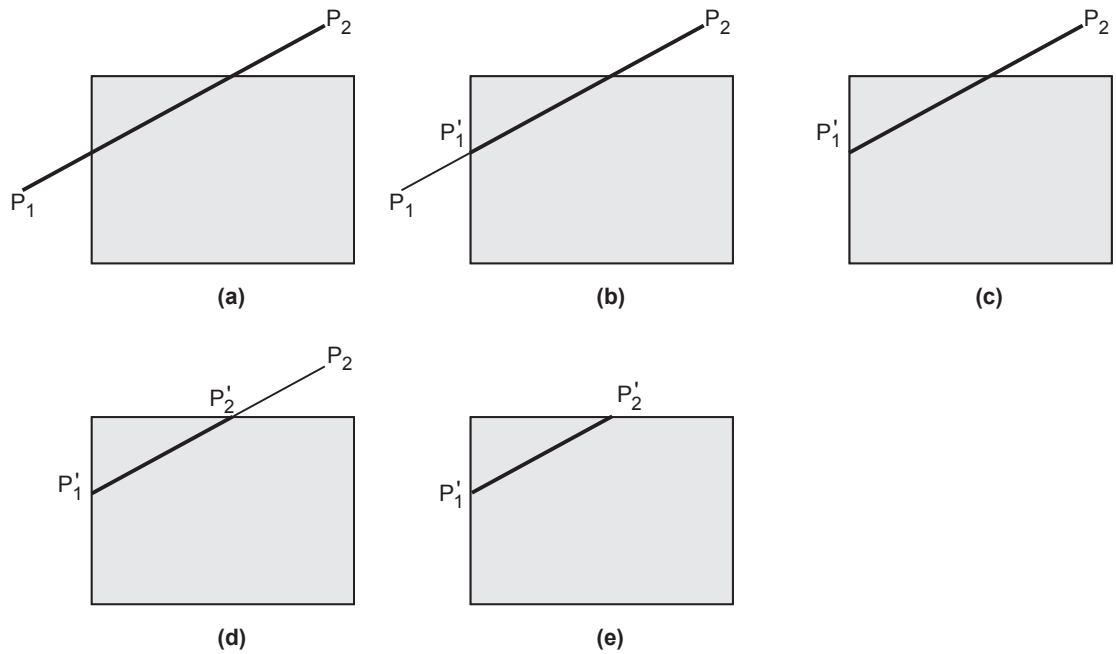


Fig. 5.4.4 Sutherland-Cohen subdivision line clipping

- As shown in the Fig. 5.4.4, line $P_1 P_2$ is a partially visible and point P_1 is outside the window. Starting with point P_1, the intersection point P_1' is found and we get two line segments $P_1 - P_1'$ and $P_1' - P_2$.
- We know that, for $P_1 - P_1'$ one end point i.e. P_1 is outside the window and thus the line segment $P_1 - P_1'$ is discarded.
- The line is now reduced to the section from P_1' to P_2. Since P_2 is outside the clip window, it is checked against the boundaries and intersection point P_2' is found. Again the line segment is divided into two segments giving $P_1' - P_2'$ and $P_2' - P_2$. We know that, for $P_2' - P_2$ one end point i.e. P_2 is outside the window and thus the line segment $P_2' - P_2$ is discarded.
- The remaining line segment $P_1' - P_2'$ is completely inside the clipping window and hence made visible.
- The intersection points with a clipping boundary can be calculated using the slope-intercept form of the line equation.
- The equation for line passing through points $P_1 (x_1, y_1)$ and $P_2 (x_2, y_2)$ is

$$y = m(x - x_1) + y_1 \quad \text{or} \quad y = m(x - x_2) + y_2 \qquad \ldots (5.4.1)$$

where $m = \dfrac{y_2 - y_1}{x_2 - x_1}$ (slope of the line)

- Therefore, the intersections with the clipping boundaries of the window are given as :
  - Left : x_L, $y = m(x_L - x_1) + y_1$; $m \neq \infty$
  - Right : x_R, $y = m(x_R - x_1) + y_1$; $m \neq \infty$
  - Top : y_T, $x = x_1 + \left(\frac{1}{m}\right)(y_T - y_1)$; $m \neq 0$
  - Bottom : y_B, $x = x_1 + \left(\frac{1}{m}\right)(y_B - y_1)$; $m \neq 0$

Sutherland and Cohen subdivision line clipping algorithm :

1. Read two end points of the line say P_1 (x_1, y_1) and P_2 (x_2, y_2).
2. Read two corners (left-top and right-bottom) of the window, say (Wx_1, Wy_1 and Wx_2, Wy_2).
3. Assign the region codes for two endpoints P_1 and P_2 using following steps :

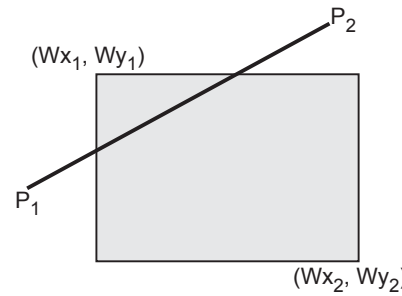


Fig. 5.4.5

Initialize code with bits 0000

Set Bit 1 - if $(x < Wx_1)$

Set Bit 2 - if $(x > Wx_2)$

Set Bit 3 - if $(y < Wy_2)$

Set Bit 4 - if $(y > Wy_1)$

4. Check for visibility of line P_1 P_2
   a) If region codes for both endpoints P_1 and P_2 are zero then the line is completely visible. Hence draw the line and go to step 9.
   b) If region codes for endpoints are not zero and the logical ANDing of them is also non-zero then the line is completely invisible, so reject the line and go to step 9.
   c) If region codes for two endpoints do not satisfy the conditions in 4a) and 4b)the line is partially visible.
5. Determine the intersecting edge of the clipping window by inspecting the region codes of two endpoints.
   a) If region codes for both the end points are non-zero, find intersection points P_1' and P_2' with boundary edges of clipping window with respect to point P_1 and point P_2, respectively
   b) If region code for any one end point is non-zero then find intersection point P_1' or P_2' with the boundary edge of the clipping window with respect to it.

6. Divide the line segments considering intersection points.
7. Reject the line segment if any one end point of it appears outsides the clipping window.
8. Draw the remaining line segments.
9. Stop.

Example 5.4.2 *Use the Cohen-Sutherland outcode algorithm to clip two lines P_1 (40, 15) – P_2 (75, 45) and P_3 (70, 20) – P_4 (100, 10) against a window A (50, 10), B (80, 10), C (80, 40), D(50,40).*

Solution : Line 1 : P_1 (40, 15) P_2 (75, 45) $x_L = 50$ $y_B = 10$ $x_R = 80$ $y_T = 40$

Point	Endcode	ANDing	Position
P_1	0 0 0 1	0 0 0 0	Partially visible
P_2	0 0 0 0		

$$x_L, y = m(x_L - x) + y_1 = \frac{6}{7}(50 - 40) + 15 \qquad m = \frac{45-15}{75-40} = \frac{6}{7}$$

$$= \mathbf{23.57}$$

$$y_T, x = x_1 + \left(\frac{1}{m}\right)(y_T - y_1) = 40 + \left(\frac{7}{6}\right)(40 - 15)$$

$$= \mathbf{69.16}$$

$I_1 = (50, 23.57)$ $I_2 = (69.06, 40)$

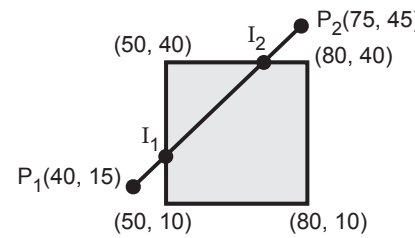


Fig. 5.4.6

Line 2 : P_3 (70, 20) P_4 (100, 10)

Point	Endcode	ANDing	Position
P_3	0 0 0 0	0 0 0 0	Partially visible
P_4	0 0 1 0		

$$\text{Slope } m' = \frac{10-20}{100-70} = \frac{-10}{30} = \frac{-\mathbf{1}}{\mathbf{3}}$$

$$x_R, y = m(x_R - x_1) + y_1 = \frac{-1}{3}(80 - 70) + 20 = \mathbf{16.66}$$

$I = (80, 16.66)$

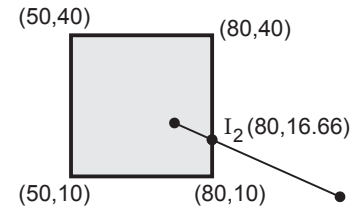


Fig. 5.4.6 (a)

Example 5.4.3 *Use outcode based line clipping method to clip a line starting from (– 13, 5) and ending at (17, 11) against the window having its lower left corner at (– 8, – 4) and upper right corner at (12, 8).*

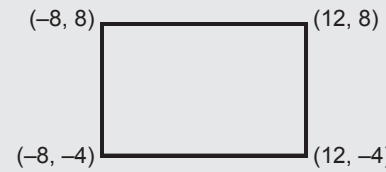


Fig. 5.4.7

Solution : $x_L = -8 \quad y_B = -4$

$x_R = 12 \quad y_T = 8$

Given line : P_1 (–13, 5), P_2 (17, 11)

Point	Endcode	ANDing	Position
P_1	0 0 0 1	0 0 0 0	Partly visible
P_2	1 0 1 0		

Slope of line : $m = \frac{6}{30} = \frac{1}{5}$

$$x_L, y = m(x_L - x) + y_1 = \frac{1}{5}(-8 - (-13)) + 5$$

$$= \frac{1}{5}(-8 + 13) + 5 = 1 + 5 = \mathbf{6}$$

$$y_T, x = x_1 + \frac{1}{m}(y_T - y) + x = -13 + 5\,(8 - 5)$$

$$= -13 + 15 = \mathbf{2}$$

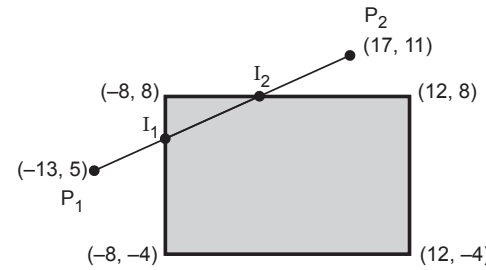


Fig. 5.4.7 (a)

$I_1 = (x_L, x_L, y) = (-8, 6) \quad I_2 = \left(Y_{T,x}, Y_T\right) = (2, 8)$

Example 5.4.4 *Let R be the rectangular window whose lower left-hand corner is at L (– 3, 1) and upper right-hand corner is at* R (2, 6)*. If the line segment is defined with two and points with A (– 4, 2) and B(–1, 7),*

a) The region codes of the two end points.

b) Its clipping category and

c) Stages in the clipping operations using Cohen-sutherland algorithm.

Solution :

a) Region of code for point A = 0001

Region of code for point B = 1000

b) Since (0001) AND (1000) = 0000 line segment AB is partially visible.

c) Stages in the clipping line segment AB are :

1. To push the 1 to 0 in code for A (0001), we clip against the window boundary line $x_{min} = -3$.

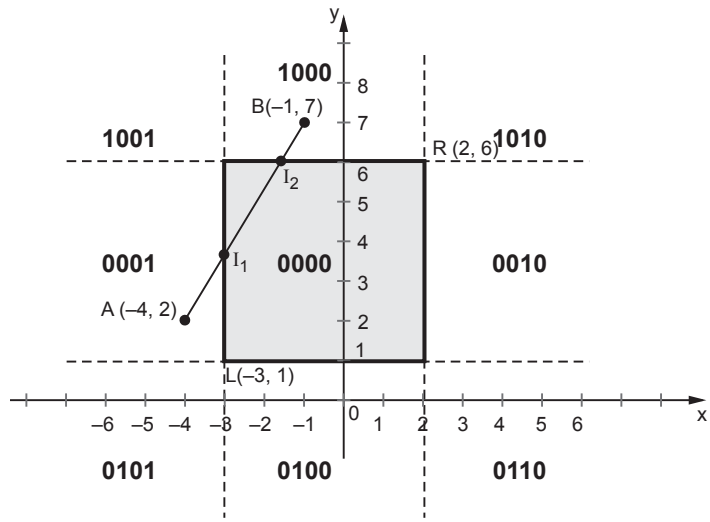


Fig. 5.4.8

2. $m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{7-2}{-1-(-4)} = \frac{5}{3}$

3. We obtain the intersection point as,

$$I_1 = x_{min},\ m(x_{min} - x_1) + y_1 = -3,\ \frac{5}{3}[-3-(-4)]+2$$

$$= -3, \frac{11}{3}$$

$$I_2 = x_1 + \left(\frac{1}{m}\right)(y_{max} - y_1),\ y_{max} = -4 + \frac{3}{5}(6-2),\ 6$$

$$= \frac{-8}{5}, 6$$

4. Clip (do not display) segments AI_1 AI_2.
5. Display segment I_1I_2 since both end points lie in the window.

Example 5.4.5 *Clip the line PQ having coordinates P(4, 1) and Q(6, 4) against the clip window having vertices A(3, 2), B(7, 2), C(7, 6) and D(3, 6) using Cohen Sutherland line clipping algorithm.*

Solution : Line : A (4, 1), B (6, 4), $x_L = 3$, $y_B = 2$, $x_R = 7$, $y_T = 6$

Point	Endcode	ANDing	Position
P	0 1 0 0	0 0 0 0	Partially visible
Q	0 0 0 0		

$$\text{Line slope } m = \frac{4-1}{6-4} = \frac{3}{2}$$

$$y_B, x = x_1 + \left(\frac{1}{m}\right)(y_B - y_1)$$

$$= x_P + \left(\frac{1}{m}\right)(2 - y_p)$$

$$= 4 + \frac{2}{3}(2-1) = 4 + \frac{2}{3} = \frac{14}{3}$$

$$I = (y_B, x, y_B) = \left(\frac{14}{3}, 2\right)$$

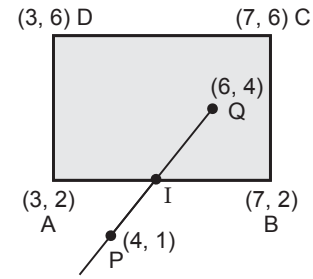


Fig. 5.4.9

Example for Practice

Example 5.4.6 : *Find the clipping co-ordinates to clip the line segment $P_1 P_2$ against the window ABCD using Cohen Sutherland line clipping algorithm $P_1 = (10, 30)$ and $P_2 = (80, 90)$ window ABCD - A (20, 20), B (90, 20), C (90, 70) and D (20, 70).*

Review Question

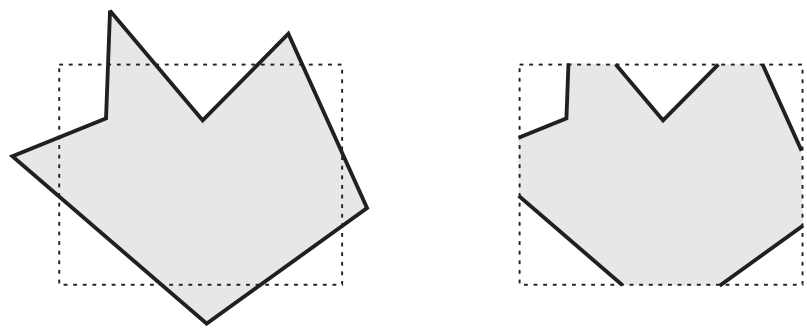
1. *Explain Cohen-Sutherland algorithm with the help of suitable example.*

SPPU : May-06, 07, 08, 10, 11, 12, 13, 15, 16,17,19, Dec.-10, 15,18, Marks 8

5.5 Polygon Clipping

SPPU : Dec.-05,06,07,08,11,14,17,18, May-07,10,14

- A polygon is nothing but the collection of lines. Therefore, we might think that line clipping algorithm can be used directly for polygon clipping. However, when a closed polygon is clipped as a collection of lines with line clipping algorithm, the original closed polygon becomes one or more open polygon or discrete lines as shown in the Fig. 5.5.1. Thus, we need to modify the line clipping algorithm to clip polygons.
- We consider a polygon as a closed solid area. Hence after clipping it should remain closed. To achieve this we require an algorithm that

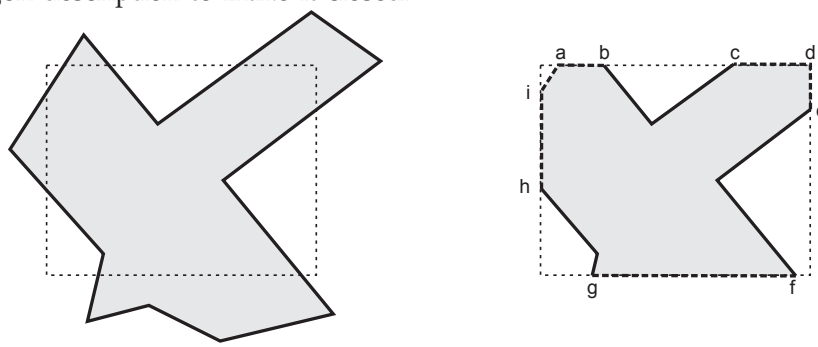


(a) Before clipping (b) After clipping

Fig. 5.5.1 Polygon clipping done by line clipping algorithm

will generate additional line segment which make the polygon as a closed area.

- For example, in Fig. 5.5.2 the lines a - b, c - d, d - e, f - g, and h - i are added to polygon description to make it closed.



(a) (b)

Fig. 5.5.2 Modifying the line clipping algorithm for polygon

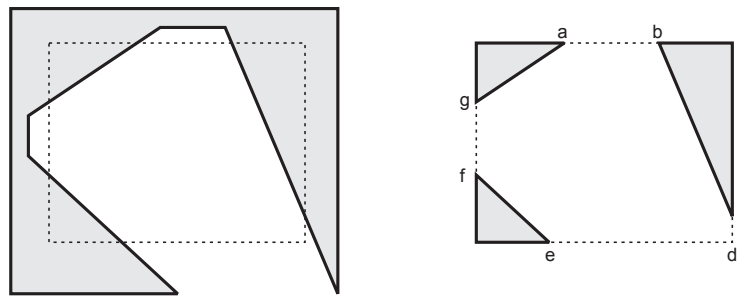


Fig. 5.5.3 Disjoint polygons in polygon clipping

- Adding lines c - d and d - e is particularly difficult. Considerable difficulty also occurs when clipping a polygon results in several disjoint smaller polygons as shown in the Fig. 5.5.3. For example, the lines a - b, c - d, d - e and g - f are frequently included in the clipped polygon description which is not desired.

5.5.1 Sutherland - Hodgeman Polygon Clipping

- A polygon can be clipped by processing its boundary as a whole against each window edge. This is achieved by processing all polygon vertices against each clip rectangle boundary in turn. Beginning with the original set of polygon vertices, we could first clip the polygon against the left rectangle boundary to produce a new sequence of vertices.
- The new set of vertices could then be successively passed to a right boundary clipper, a top boundary clipper and a bottom boundary clipper, as shown in Fig. 5.5.4. At each step a new set of polygon vertices is generated and passed to the next window boundary clipper. This is the fundamental idea used in the Sutherland - Hodgeman algorithm.

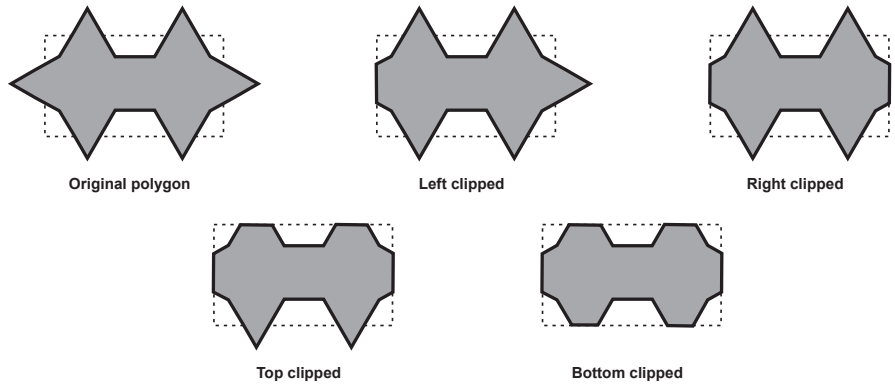


Fig. 5.5.4 Clipping a polygon against successive window boundaries

- The output of the algorithm is a list of polygon vertices all of which are on the visible side of a clipping plane. Such each edge of the polygon is individually compared with the clipping plane.

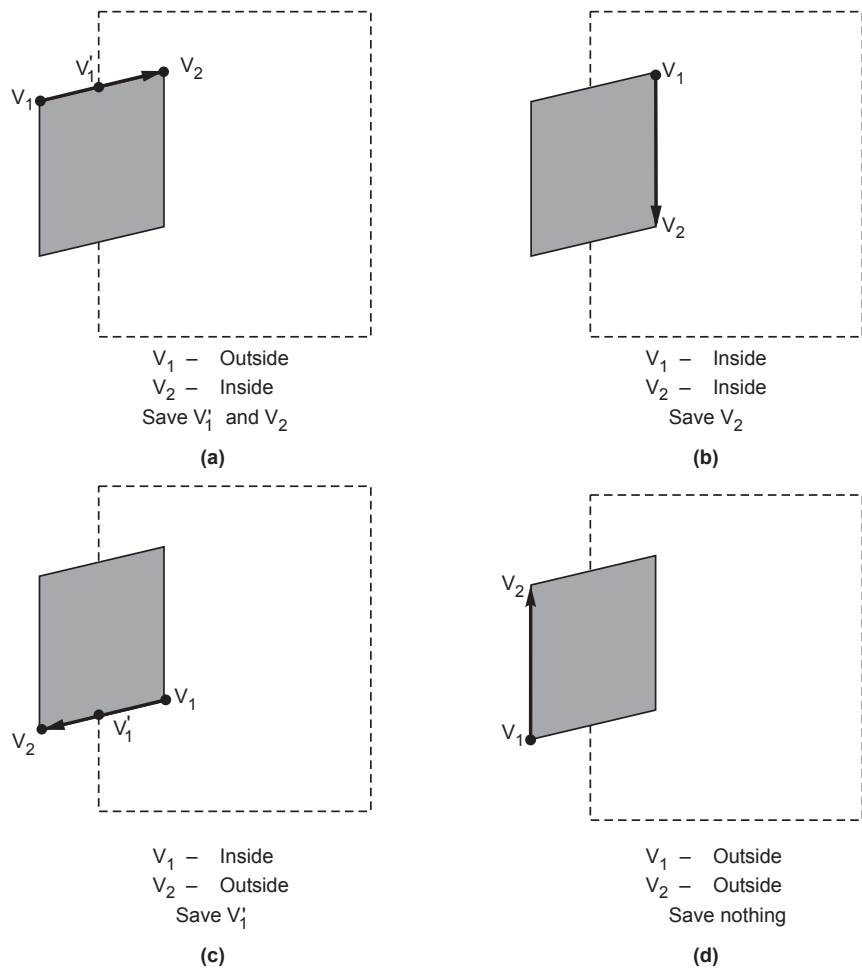


Fig. 5.5.5 Processing of edges of the polygon against the left window boundary

This is achieved by processing two vertices of each edge of the polygon around the clipping boundary or plane. This results in four possible relationships between the edge and the clipping boundary or plane. (See Fig. 5.5.5).

1. If the first vertex of the edge is outside the window boundary and the second vertex of the edge is inside then the intersection point of the polygon edge with the window boundary and the second vertex are added to the output vertex list (See Fig. 5.5.5 (a)).
2. If both vertices of the edge are inside the window boundary, only the second vertex is added to the output vertex list. (See Fig. 5.5.5 (b)).
3. If the first vertex of the edge is inside the window boundary and the second vertex of the edge is outside, only the edge intersection with the window boundary is added to the output vertex list. (See Fig. 5.5.5 (c)).
4. If both vertices of the edge are outside the window boundary, nothing is added to the output list. (See Fig. 5.5.5 (d)).

- Once all vertices are processed for one clip window boundary, the output list of vertices is clipped against the next window boundary.
- Going through above four cases we can realize that there are two key processes in this algorithm.
  1. Determining the visibility of a point or vertex (Inside - Outside test) and
  2. Determining the intersection of the polygon edge and the clipping plane.
- One way of determining the visibility of a point or vertex is described here.
- Consider that two points A and B define the window boundary and point under consideration is V, then these three points define a plane.
- Two vectors which lie in that plane are AB and AV. If this plane is considered in the xy plane, then the vector cross product AV $\times$ AB has only a z component given by $(x_V - x_A)(y_B - y_A) - (y_V - y_A)(x_B - x_A)$.
- The sign of the z component decides the position of point V with respect to window boundary.

If z is : Positive – Point is on the **right side** of the window boundary
Zero – Point is **on** the window boundary
Negative – Point is on the **left side** of the window boundary

Example 5.5.1 *Consider the clipping boundary as shown in the Fig. 5.5.6 and determine the positions of points V_1 and V_2.*

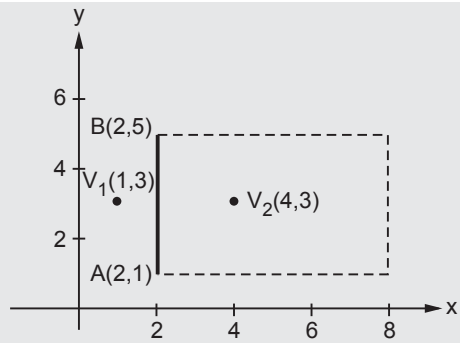


Fig. 5.5.6

Solution : Using the cross product for V_1 we get,

$$(x_V - x_A)\ (y_B - y_A) - (y_V - y_A)\ (x_B - x_A)$$

$$= (1 - 2)\ (5 - 1) - (3 - 1)\ (2 - 2)$$

$$= (-1)\ (4) - 0$$

$$= -4$$

The result of the cross product for V_1 is negative hence V_1 is on the left side of the window boundary.

Using the cross product for V_2 we get, $(4 - 2)\ (5 - 1) - (3 - 1)\ (2 - 2)$

$$= (2)\ (4) - 0$$

$$= 8$$

The result of the cross product for V_2 is positive hence V_1 is on the right side of the window boundary.

The second key process in Sutherland - Hodgeman polygon clipping algorithm is to determine the intersection of the polygon edge and the clipping plane. Any of the line intersection (clipping) techniques discussed in the previous sections such as Cyrus-Beck or mid point subdivision can be used for this purpose.

Sutherland-Hodgeman Polygon Clipping Algorithm

1. Read coordinates of all vertices of the polygon.
2. Read coordinates of the clipping window
3. Consider the left edge of the window
4. Compare the vertices of each edge of the polygon, individually with the clipping plane

5. Save the resulting intersections and vertices in the new list of vertices according to four possible relationships between the edge and the clipping boundary discussed earlier.
6. Repeat the steps 4 and 5 for remaining edges of the clipping window. Each time the resultant list of vertices is successively passed to process the next edge of the clipping window.
7. Stop.

Example 5.5.2 *For a polygon and clipping window shown in Fig. 5.5.7 give the list of vertices after each boundary clipping.*

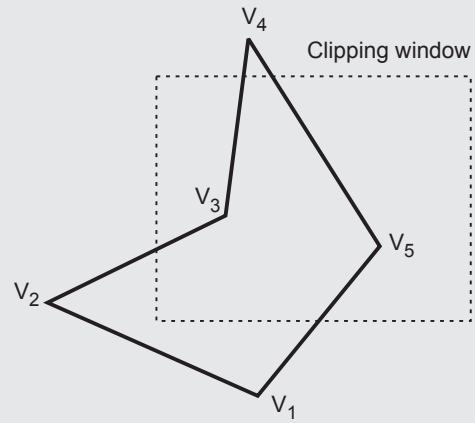


Fig. 5.5.7

Solution : Original polygon vertices are V_1, V_2, V_3, V_4, V_5. After clipping each boundary the new vertices are given in Fig. 5.5.7 (a).

After left clipping : V_1, V_1', V_2', V_3, V_4, V_5

After right clipping : V_1, V_1', V_2', V_3, V_4, V_5

After top clipping : V_1, V_1', V_2', V_3, V_3', V_4', V_5

After bottom clipping : V_2'', V_2', V_3, V_3', V_4', V_5, V_5'

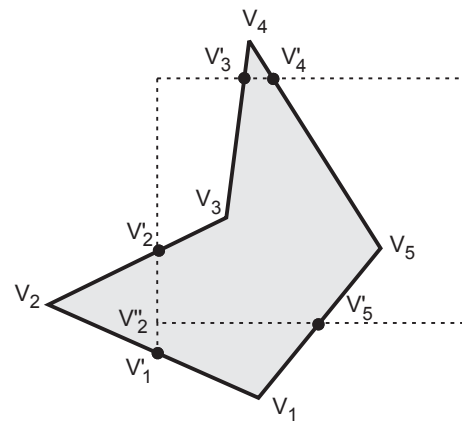


Fig. 5.5.7(a)

- The Sutherland-Hodgeman polygon clipping algorithm clips convex polygons correctly, but in case of concave polygons, clipped polygon may be displayed with extraneous lines, as shown in Fig. 5.5.8.

- The problem of extraneous lines for concave polygons in Sutherland-Hodgeman polygon clipping algorithm can be solved by separating concave polygon into two or more convex polygons and processing each convex polygon separately.

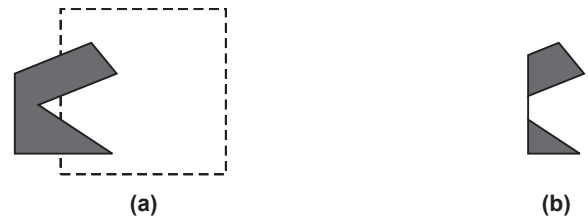


Fig. 5.5.8 Clipping the concave polygon in (a) with the Sutherland-Hodgeman algorithm produces the two connected areas in (b)

5.5.2 Weiler-Atherton Algorithm

The clipping algorithms previously discussed require a convex polygon. In context of many applications, e.g. hidden surface removal, the ability to clip to concave polygon is required. A powerful but somewhat more complex clipping algorithm developed by Weiler and Atherton meets this requirement. This algorithm defines the polygon to be clipped as a **subject polygon** and the clipping region is the clip polygon.

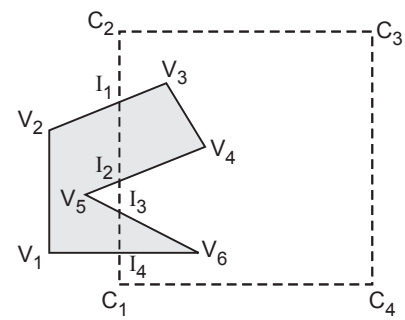


Fig. 5.5.9

The algorithm describes both the subject and the clip polygon by a circular list of vertices. The boundaries of the subject polygon and the clip polygon may or may not intersect. If they intersect, then the intersections occur in pairs. One of the intersections occurs when a subject polygon edge enters the inside of the clip polygon and one when it leaves. As shown in the Fig. 5.5.9, there are four intersection vertices I_1, I_2 , I_3 and I_4. In these intersections I_1 and I_3 are entering intersections, and I_2 and I_3 are leaving intersections. The clip polygon vertices are marked as C_1, C_2, C_3 and C_4.

In this algorithm two separate vertices lists are made one for clip polygon and one for subject polygon including intersection points. The Table 5.5.1 shows these two lists for polygons shown in Fig. 5.5.10.

The algorithm starts at an entering intersection (I_1) and follows the subject polygon vertex list in the downward

For subject polygon		For clip polygon	
	V_1	C_1	
	V_2	I_4	
Start	I_1	I_3	Finish
	V_3	I_2	
	V_4	I_1	Finish
	I_2	C_2	
	V_5	C_3	
Start	I_3	C_4	
	V_6		
	I_4		
	V_1		

Table 5.5.1 List of polygon vertices

direction (i.e. I_1, V_3, V_4, I_2). At the occurrence of leaving intersection the algorithm follows the clip polygon vertex list from the leaving intersection vertex in the downward direction (i.e. I_2, I_1). At the occurrence of the entering intersection the algorithm follows the subject polygon vertex list from the entering intersection vertex. This process is repeated until we get the starting vertex. This process we have to repeat for all remaining entering intersections which are not included in the previous traversing of vertex list. In our example, entering vertex I_3 was not included in the first traversing of vertex list,. Therefore, we have to go for another vertex traversal from vertex I_3.

The above two vertex traversals gives two clipped inside polygons. There are :

I_1, V_3, V_4, I_2, I_1 and I_3, V_6, I_4, I_3

5.5.3 Liang-Barsky Polygon Clipping

This section gives a brief outline of the Liang-Barsky polygon clipping algorithm. This algorithm is optimized for rectangular clipping window but is extensible to arbitrary convex windows. The algorithm is based on concepts from their two and three dimensional line clipping algorithm. The algorithm is claimed to be twice as fast as the Sutherland-Hodgeman clipping algorithm.

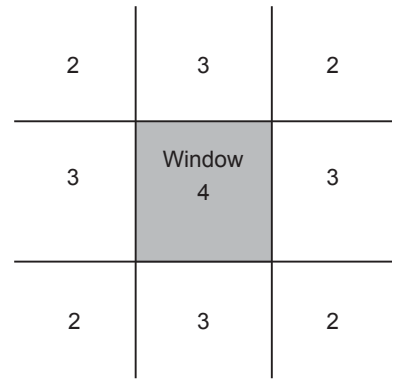


Fig. 5.5.10 Nine regions of clipping plane

The Liang-Barsky algorithm assumes that the clipping region and polygon are coplanar. Each edge of the clipping window divides the plane into two half planes. The side of the half plane containing the window is called the **inside** or the **visible half**. The other half plane is the **outside** or **invisible half** plane. The four window edges divide the clipping plane into nine regions, as shown in the Fig. 5.5.10.

Entering and Leaving Vertices

Assume that $[P_i, P_{i+1}]$ is an edge of a polygon. If it is not horizontal or vertical then the infinite line, L_i, containing the polygon edge intersects each of the four window edges. In fact, as shown in the Fig. 5.5.11, such an infinite line always starts in a corner region, labelled 2, and ends in the diagonally opposite corner region, whether it intersects the window or not. For example, line A and C intersect the window while line B does not. As shown in the Fig. 5.5.11, each line has four window edge intersections. The first intersection of the infinite line, L_i, with a window edge represents a transition from an invisible (outside) to a visible (inside) side of the window edge. Such an intersection is called an **entering intersection**.

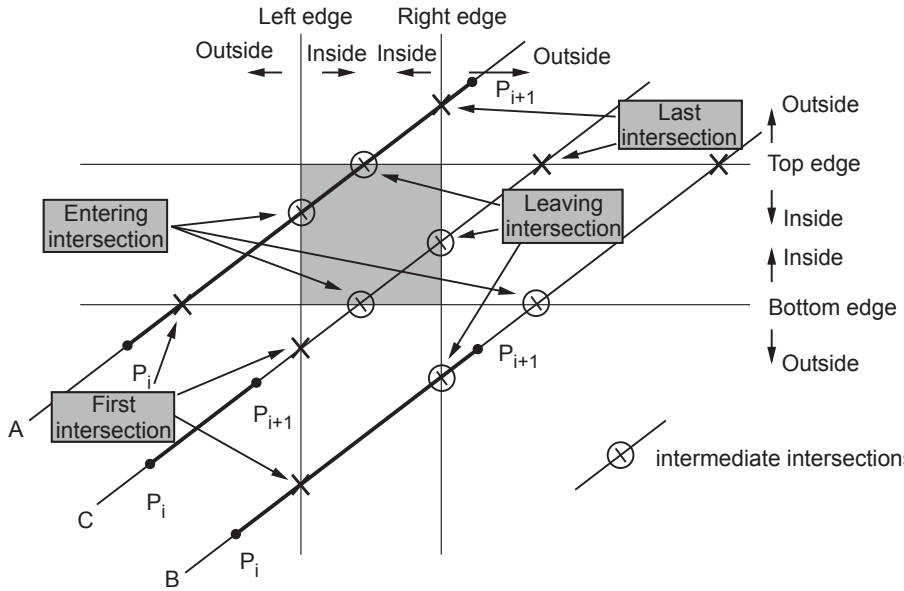


Fig. 5.5.11 Entering and leaving intersections

The last intersection of the infinite line, L_i, with a window edge represents a transition from a visible to an invisible side of the window edge and is called a **leaving intersection**. The two intermediate intersections can occur in either entering-leaving order as illustrated by the lines A and C, or in leaving-entering order, as illustrated by the line B in the Fig. 5.5.11.

If the intermediate intersections occur in entering leaving order, then the infinite line, L_i, intersects the window. However, the visibility of any part or all of the polygon edge, $P_i\ P_{i+1}$, is depend on the location of $P_i\ P_{i+1}$ along L_i, as shown by the polygon edges on lines A and C in Fig. 5.5.11.

If the intermediate intersections occur in leaving-entering order (as in case of line B), no part of the infinite line, L_i, is visible in the window.

Turning Vertices

If a subsequent polygon edge reenters the window through a different window edge, then it is necessary to include one or more of the window corners in the output

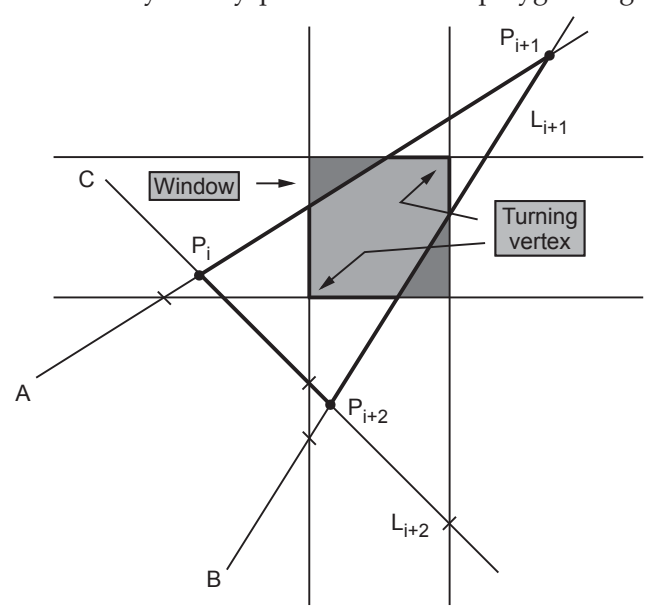


Fig. 5.5.12 Turning vertices

polygon. This is illustrated in Fig. 5.5.12. Liang and Barsky call such a window corner a **turning vertex.**

A necessary, but not sufficient, condition that a turning vertex exist if an entering intersection occurs on $P_{i+1}\ P_{i+2}$ and outside the window as shown in the Fig. 5.5.12. When this condition occurs, the Liang-Barsky polygon clipping algorithm adds the turning vertex closest to the entering vertex to the output polygon. This is a necessary but not sufficient condition, because, the entire polygon could lie outside the window. Thus, extraneous turning vertices can be added to the output polygon. This is the main drawback of the Liang-Barsky algorithm.

However, it is important to note that unless the first entering intersection is a window corner and hence coincident with the second entering intersection, and thus that the first entering intersection cannot be in the window, yields a sufficient condition for including a turning vertex in the output polygon.

It is possible to calculate the actual intersections of the Line L_i and the window edges, as well as the ordering of the intersections by using the parametric equation of the polygon edge.

$$P(t) = P_i + (P_{i+1} - P_i)\,t$$

where $0 < t \leq 1$ represents the polygon edge except for the initial point, P_i, and

$-\infty < t < \infty$ represents the infinite line containing the polygon edge

Development of the Algorithm

Liang-Barsky algorithm assumes a regular clipping region and hence it can be developed by using the components of the parametric line, i.e.

$$x = x_i + (x_{i+1} - x_i)\,t = x_i + \Delta x t$$

$$y = y_i + (y_{i+1} - y_i)\,t = y_i + \Delta y t$$

For a regular clipping region, the edges are given by,

$$x_{left} \leq x \leq x_{right}$$

$$y_{bottom} \leq y \leq y_{top}$$

Using subscripts e and *l* to denote entering and leaving intersections, the parametric values for the four intersections with the window edges are

$$t_{xe} = (x_e - x_i)/\Delta x_i \qquad \Delta x_i \neq 0$$

$$t_{xl} = (x_l - x_i)/\Delta x_i \qquad \Delta x_i \neq 0$$

$$t_{ye} = (y_e - y_i)/\Delta y_i \qquad \Delta y_i \neq 0$$

$$t_{yl} = (y_l - y_i)/\Delta y_i \qquad \Delta y_i \neq 0$$

where

$$x_e = \begin{cases} x_{left} & \Delta x_i > 0 \\ x_{right} & \Delta x_i \leq 0 \end{cases}$$

$$x_l = \begin{cases} x_{right} & \Delta x_i > 0 \\ x_{left} & \Delta x_i \leq 0 \end{cases}$$

$$y_e = \begin{cases} y_{bottom} & \Delta y_i > 0 \\ y_{top} & \Delta y_i \leq 0 \end{cases}$$

$$y_l = \begin{cases} y_{top} & \Delta y_i > 0 \\ y_{bottom} & \Delta y_i \leq 0 \end{cases}$$

The first and second entering and leaving intersections are given by

$$t_{e1} = \min (t_{xe}, t_{ye})$$

$$t_{e2} = \max (t_{xe}, t_{ye})$$

$$t_{l1} = \min (t_{xl}, t_{yl})$$

$$t_{l2} = \max (t_{xl}, t_{yl})$$

Let us observe the conditions that decides whether to contribute, not to contribute, partly contribute, or to add turning vertex to the output polygon.

Conditions for no contribution to the output polygon

1. Termination of edge before the first entering intersection ($1 < t_{e1}$), see line A in Fig. 5.5.13.
2. Starting of edge after the first entering intersection ($0 \geq t_{e1}$) and ending of edge before the second entering intersection ($1 < t_{e2}$), see line B in Fig. 5.5.13.
3. Starting of edge after the second entering intersection ($0 \geq t_{e2}$) and after the first leaving intersection ($0 \geq t_{l1}$), see line C in Fig. 5.5.13.

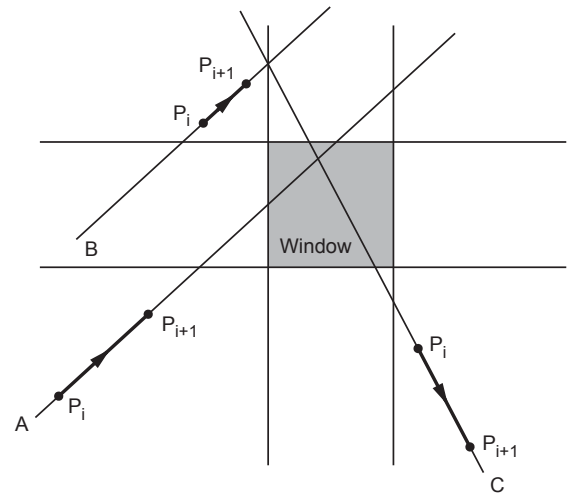


Fig. 5.5.13 Conditions for no contribution of the output polygon

Conditions for contribution to the output polygon

4. If the second entering intersection occurs before the first leaving intersection ($t_{e2} \leq t_{l1}$), and P_i lies on L_i before the first leaving intersection ($0 < t_{l1}$), and P_{i+1} lies on L_i after the second entering intersection ($1 \geq t_{e2}$), the edge is either partially or wholly contained within the window and hence partially or wholly visible. See lines D, E and F in Fig. 5.5.14.

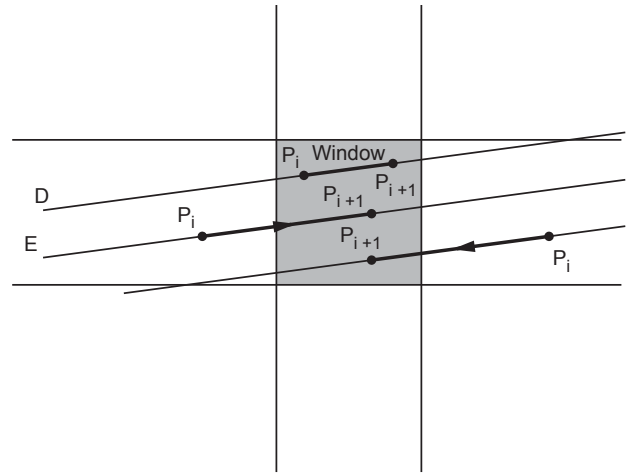


Fig. 5.5.14 Condition for contribution to the output polygon

If the edge is partially visible, we can obtain the intersection point and contribution to the output polygon using parametric equation.

Conditions for addition of a turning vertex to the output polygon

5. If the first entering intersection lies on $P_i\ P_{i+1}$ ($0 < t_{e1} \leq 1$), then the turning vertex is (x_e, y_e), e.g. line G in Fig. 5.5.15.

6. If the second entering intersection lies on $P_i\ P_{i+1}$ ($0 < t_{e2} \leq 1$), and is outside the window ($t_{l1} < t_{e2}$), then the turning vertex is either (x_e, y_l) when the second entering intersection lies on a vertical edge of the window ($t_{xe} > t_{ye}$), e.g. line H in the Fig. 5.5.15, or (x_l, y_e) when the second entering intersection lies on a horizontal edge of the window ($t_{ye} > t_{xe}$), e.g. line I in Fig. 5.5.15.

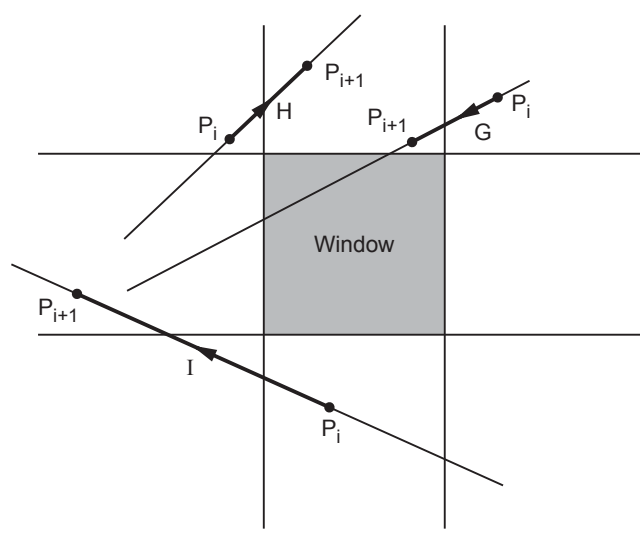


Fig. 5.5.15 Conditions for addition of a turning vertex

The Table 5.5.2 summarizes the six output conditions for polygon edges.

Sr.No.	Condition	Contribution
1.	$1 < t_{e1}$	No
2.	$0 \geq t_{e1}$ and $1 < t_{e2}$	No
3.	$0 \geq t_{e2}$ and $0 \geq t_{l1}$	No
4.	$t_{e2} \leq t_{l1}$ and $0 \leq t_{l1}$ and $1 \geq t_{e2}$	Visible segment
5.	$0 \leq t_{e1} \leq 1$	Turning vertex (x_e, y_e)
6.	$0 < t_{e2} \leq 1$ and $t_{l1} < t_{e2}$	Turning vertex (x_e, y_l) for $t_{xe} > t_{ye}$ (x_l, y_e) for $t_{ye} > t_{xe}$

Table 5.5.2 Output conditions for polygon edges

The conditions listed in the Table 5.5.2 are independent of the others except the condition 5. If condition 5 generates turning vertex, condition 4 may generate a visible segment or condition 6 may generate an another turning vertex. If the polygon edge penetrates the window, a visible segment may be generated by condition 4; or if the polygon edge is completely outside the window, another turning vertex may be generated by condition 6. Therefore, in the algorithm we must evaluate the condition 5 prior to evolution of conditions 4 and 6.

Horizontal and vertical edges

Upto this point, we have ignored the horizontal and vertical polygon edges. These polygon edges are easily detected by looking for Δy or $\Delta x = 0$ respectively. For the horizontal line, $\Delta y = 0$ and $t_{ye} \rightarrow -\infty$ and $t_{yl} \rightarrow +\infty$. Thus, the horizontal line is characterized by

$$-\infty \leq t_{xe} \leq t_{xl} \leq +\infty$$

Similarly the vertical line is characterized by

$$-\infty \leq t_{ye} \leq t_{yl} \leq +\infty$$

Algorithm

For each polygon edge P(i) P(i + 1)

- Determine the direction of the edge and find whether it is diagonal, vertical or horizontal.
- Determine the order of the edge-window intersections.
- Determine the t-values of the entering edge-window intersections and the t-values for the first leaving intersection.

- Analyze the edge.

```
if 1 >= t_enter_1 then [condition 2 or 3 or 4 or 6 and not 1]
    if 0 >= t_enter_1 then [condition 5]
            call output (turning vertex);
    end if
    if 1 >= t_enter_2 then [condition 3 or 4 or 6 and not 1 or 2]
    [determine second leaving t-value there is output]
    if (0 < t_enter_2) or (0 < t_leave_1) then
                    [condition 4 or 6, not 3]
            if t_enter_2 <= t_leave_1 then [condition 4 - visible segment]
            call output (appropriate edge intersection)
    else [end condition 4 and begin condition 6-turning vertex]
            call output (appropriate turning vertex)
    end if [end condition 6]
  end if  [end condition 4 or 6]
end if [end condition 3 or 4 or 6]
end if [end condition 2 or 3 or 4 or 6]
next i      end edge P[i]P[i + 1]
```

Example 5.5.3 *Find clipping co-ordinates for a line $p_1 p_2$ using Liang-Barsky algorithm where p_1 = (50, 25) and p_2 = (80, 50), against window with $(x_{w\min}, y_{w\min})$ = (20, 10) and $(x_{w\max}, y_{w\max})$ = (70, 60).*

Solution : $x_1 = 50,\ y_1 = 25,\ x_2 = 80,\ y_2 = 50$

$x_{min} = 20,\ y_{min} = 10,\ x_{max} = 70,\ y_{max} = 60$

$$p_1 = -(x_2 - x_1) = -(80 - 50) = -30$$

$$p_2 = (x_2 - x_1) = (80 - 50) = 30$$

$$p_3 = -(y_2 - y_1) = -(50 - 25) = -25$$

$$p_4 = (y_2 - y_1) = (50 - 25) = 25$$

$$q_1 = (x_1 - x_{min}) = (50 - 20) = 30$$

$$q_2 = (x_{max} - x_1) = (70 - 50) = 20$$

$$q_3 = (y_1 - y_{min}) = (25 - 10) = 15$$

$$q_4 = (y_{max} - y_1) = (60 - 25) = 35$$

$q_1/p_1 = 30/-30 = -1.000$

$q_2/p_2 = 20/30 = 0.667$

$q_3/p_3 = 15/-25 = -0.600$

$q_4/p_4 = 35/25 = 1.400$

$t_1 = \max(-1.000, -0.600) = 0.00$ Since for these values $p < 0$

$t_2 = \min(0.667, 1.400) = 0.667$ Since for these values $p > 0$

Note : Minimum value of $t_1 = 0$

$xx_1 = x_1 + t_1 * dx = 50 + 0.00 \times 30.00 = 50.00$

$yy_1 = y_1 + t_1 * dy = 25 + 0.00 \times 25.00 = 25.00$

$xx_2 = x_1 + t_2 * dx = 50 + 0.67 \times 30.00 = 70.00$

$yy_2 = y_1 + t_2 * dy = 25 + 0.67 \times 25.00 = 41.67$

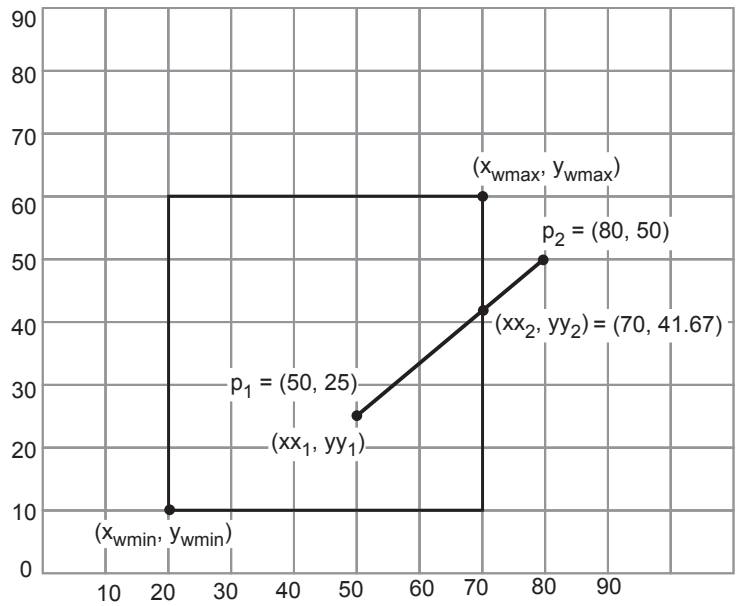


Fig. 5.5.16

Example 5.5.4 *Find clipping co-ordinates for a line AB, where A = (5, 20) and B = (30, 30). Co-ordinates of window are :* $(x_{w\min}, y_{w\min}) = (15, 15)$ *and* $(x_{w\max}, y_{w\max}) = (45,45)$. *Solve using Liang-Barsky line clipping algorithm.*

Solution : $x_1 = 5$, $y_1 = 20$, $x_2 = 30$, $y_2 = 30$

$x_{min} = 15$, $y_{min} = 15$, $x_{max} = 45$, $y_{max} = 45$

$$p_1 = -(x_2 - x_1) = -(30 - 5) = -25$$
$$p_2 = (x_2 - x_1) = (30 - 5) = 25$$
$$p_3 = -(y_2 - y_1) = -(30 - 20) = -10$$
$$p_4 = (y_2 - y_1) = (30 - 20) = 10$$

$$q_1 = (x_1 - x_{min}) = (5 - 15) = -10$$
$$q_2 = (x_{max} - x_1) = (45 - 5) = 40$$
$$q_3 = (y_1 - y_{min}) = (20 - 15) = 5$$
$$q_4 = (y_{max} - y_1) = (45 - 20) = 25$$

$$q_1/p_1 = -10/-25 = 0.400$$
$$q_2/p_2 = 40/25 = 1.600$$
$$q_3/p_3 = 5/-10 = -0.500$$
$$q_4/p_4 = 25/10 = 2.500$$

t_1 = max (0.400, – 0.500) = 0.400 Since for these values $p < 0$

t_2 = min (1.600, 2.500) = 1.600 Since for these values $p > 0$

Note : Maximum value of $t_2 = 1$

$$xx_1 = x_1 + t_1 * dx = 5 + 0.40 \times 25.00 = 15.00$$
$$yy_1 = y_1 + t_1 * dy = 20 + 0.40 \times 10.00 = 24.00$$
$$xx_2 = x_1 + t_2 * dx = 5 + 1.00 \times 25.00 = 30.00$$
$$yy_2 = y_1 + t_2 * dy = 20 + 1.00 \times 10.00 = 30.00$$

(See Fig. 5.5.17 on next page)

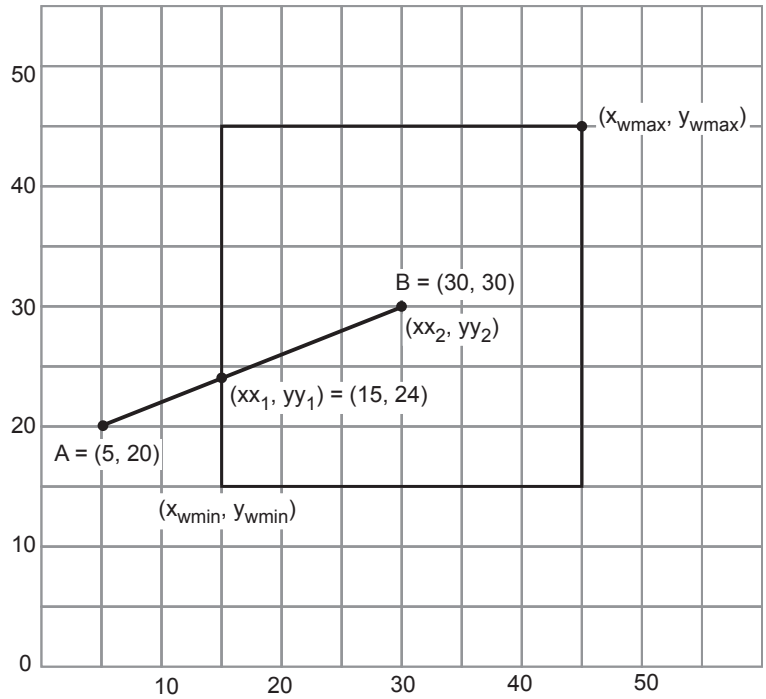


Fig. 5.5.17

Examples for Practice

Example 5.5.5 : *Find clipping co-ordinates for a line AB, where A = (1, 3) and B = (5, 12). Co-ordinates of window are :* $(x_{w\min}, y_{w\min})$ *= (2, 1) and* $(x_{w\max}, y_{w\max})$ *= (9, 10). Solve using Liang-Barsky line clipping algorithm.*

Example 5.5.6 : *Find clipping co-ordinates for a line* $p_1 p_2$ *using Liang-Barsky algorithm where* p_1 *= (90, 20) and* p_2 *= (20, 90), against window with* $(x_{w\min}, y_{w\min})$ *= (10, 10) and* $(x_{w\max}, y_{w\max})$ *= (60, 60).*

Example 5.5.7 : *Find clipping co-ordinates for a line* $p_1 p_2$ *using Liang-Barsky algorithm where* p_1 *= (60, 20) and* p_2 *= (110, 40), against window with* $(x_{w\min}, y_{w\min})$ *= (30, 10) and* $(x_{w\max}, y_{w\max})$ *= (90, 50).*

Example 5.5.8 : *Find clipping co-ordinates for a line* $p_1 p_2$ *using Liang-Barsky algorithm where* p_1 *= (5, –3) and* p_2 *= (3, 4), against window with* $(x_{w\min}, y_{w\min})$ *= (–1, 2) and* $(x_{w\max}, y_{w\max})$ *= (6, 5).*

Review Questions

1. *Can line clipping algorithm be used for polygon clipping ? Justify ?* **SPPU : Dec.-05, Marks 8**
2. *Describe Sutherland - Hodgeman polygon clipping algorithm with example.*
   SPPU : Dec.-06, 07, 08, 11, 14,17, 18, May-10, 14, Marks 8
3. *What is polygon clipping ?* **SPPU : May-07, Marks 2**

5.6 Generalized Clipping

SPPU : Dec.-07,11,12, May-09

- We have seen that in Sutherland - Hodgeman polygon clipping algorithm we need separate clipping routines, one for each boundary of the clipping window. But these routines are almost identical. They differ only in their test for determining whether a point is inside or outside the boundary.
- It is possible to generalize these routines so that they will be exactly identical and information about the boundary is passed to the routines through their parameters.
- Using recursive technique the generalized routine can be 'called' for each boundary of the clipping window with a different boundary specified by its parameters.
- This form of algorithm allows us to have any number of boundaries to the clipping window, thus the generalized algorithm with recursive technique can be used to clip a polygon along an arbitrary convex clipping window.

Review Question

1. *Explain the concept of generalized clipping with the help of suitable example.*
   SPPU : Dec.-07, 11, 12, May-09, Marks 8

5.7 Interior and Exterior Clipping

SPPU : May-05,13, Dec.-05,12

- So far we have discussed only algorithms for clipping point, line and polygon to the interior of a clipping region by eliminating every thing outside the clipping region. However, it is also possible to clip a point, line or polygon to the exterior of a clipping region, i.e., the point, portion of line and polygon which lie outside the clipping region. This is referred to as **exterior clipping**.
- Exterior clipping is important in a multiwindow display environment, as shown in Fig. 5.7.1.
- Fig. 5.7.1 shows the overlapping windows with window 1 and window 3 having priority over window 2. The objects within the window are clipped to the interior of that window.

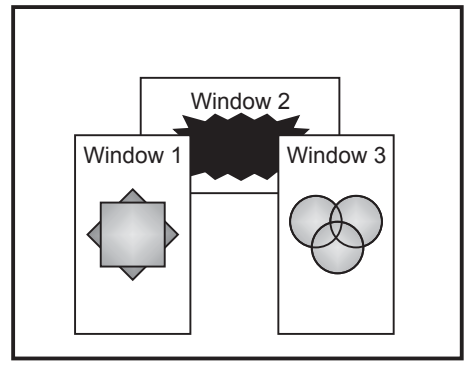


Fig. 5.7.1 Clipping in multiwindow environment

- When other higher-priority windows such as window 1 and/or window 3 overlap these objects, the objects are also clipped to the exterior of the overlapping windows.

Review Question

1. *What is interior and exterior clipping.* **SPPU : May-05, 13, Dec.-05, 12, Marks 4**

❑❑❑

Notes