MMTE-004
COMPUTER GRAPHICS

Indira Gandhi National Open University
School of Sciences

Curriculum Design Committee

Dr. B.D. Acharya
Dept. of Science & Technology, New Delhi

Prof. Adimurthi
School of Mathematics
TIFR, Bangalore

Prof. Archana Aggarwal
CESP, School of Social Sciences
JNU, New Delhi

Prof. R.B. Bapat
Indian Statistical Institute
New Delhi

Prof. M.C. Bhandari
Dept. of Mathematics
IIT, Kanpur

Prof R. Bhatia
Indian Statistical Institute, New Delhi

Prof. A.D. Dharmadhikari
Dept. of Statistics
University of Pune

Prof. O.P. Gupta
Dept. of Financial Studies
University of Delhi

Prof. S.D. Joshi
Dept. of Electrical Engineering
IIT Delhi

Dr. R.K. Khanna
Scientific Analysis Group, DRDO, Delhi

Prof. Susheel Kumar
Dept. of Management Studies
IIT, Delhi

Prof. Veni Madhavan
Scientific Analysis Group
DRDO, Delhi

Prof. J.C. Misra
Dept. of Mathematics
IIT, Kharagpur

Prof. C. Musili
Dept. of Mathematics and Statistics
University of Hyderabad

Prof. Sankar Pal
ISI, Kolkata

Prof. A.P. Singh
PG Dept. of Mathematics
University of Jammu

Faculty Members
School of Sciences, IGNOU
Prof. Poornima Mital
Dr. Atul Razdan
Prof. Parvin Sinclair
Prof. Sujatha Varma
Dr. S. Venkataraman

Course Design Committee

Prof. Sujatha Varma
Director (SOS), IGNOU

Prof. Chandan Singh(Retd.)
Department of Computer Science,
Punjabi University, Patiala

Prof. S. Balasundaram (Retd.)
School of Computer and Systems Sciences,
JNU, New Delhi

Faculty Members
School of Sciences, IGNOU

Prof. Parvin Sinclair
Dr. S. Venkataraman
Dr. Deepika
Dr. Pawan Kumar

*The syllabus was designed and unitised as per the decision of the Course Expert Committee which met on 17th March, 2021.

Block Preparation Team

Prof. B.S. Panda (Editor)
Dept. of Mathematics
IIT Delhi

Dr. Pawan Kumar
School of Sciences
IGNOU

Course Coordinator: Dr. Pawan Kumar

Acknowledgements: To Mr.Surender Singh Chauhan for typesetting the manuscript and preparing the CRC.

……., 2023

ISBN-8]-

COURSE INTRODUCTION

Computer Graphics is a fast developing field of computer science which primarily deals with modelling, representation and synthesis of real life objects. This is done using geometric objects and shapes and certain basic transformations. Due to tremendous advancements in speed and quality of hardware and software, it has rapidly gained popularity with applications in areas such as industrial design, computer games, virtual reality, synthetic content, media and entertainment, interactive TV, graphical user interfaces, information visualization, interactive art, education and the internet. Computer graphics plays an increasingly important role in popularizing applications of computers in a variety of other areas also. Fundamentals of computer graphics are based on mathematical formulations of curves, surfaces and various transformations for object representation, digital imaging, colouring, rendering and animation based on basic principles of optics and kinematics. Thus one of the important aims of studies in computer graphics is to achieve as much realism as possible in representation and rendering of real life objects, their images and animation.

This course on "Computer Graphics" is aimed to provide you an introduction to mathematical tools and algorithms for generating graphics primitives for applications in 2D and 3D graphics. In the Self Learning Material, we have used C language to implement graphics algorithms with OpenGL (open graphics library). OpenGL is a powerful low-level graphics rendering and imaging library available on all major platforms. **Appendix A** given at the end of the SLM will make you acquainted with basic structure of a graphics program using OpenGL. We advise you to go through the appendix carefully before starting any programming. The SLM contains three blocks, a brief description of them is given below.

Block 1 illustrates the diversity of applications of computer graphics and introduces you to input and hardcopy devices. Basic algorithms for plotting straight line segments, circles, ellipses and few more important output primitive curves are also discussed. It also describes basic methods to fill polygonal or arbitrary shaped closed regions.

In **Block 2**, we have discussed some primitive two-dimensional geometric transformations and their compositions, along with some clipping algorithms.

Eventually, **Block 3** introduces you to three-dimensional transformations. You will see here how all the two-dimensional transformations, except for rotation, are generalized, in a natural way to three dimensions. Here you will also learn what are projection transformations and how they are used to view 3D scenes to 2D display devices.

This course also has a practical component. The set of practicals to be carried out at your Programme/Study Centre are given in **Appendix B**. All these practicals have to be done using **C-programming** with **OpenGL utility toolkit**. You may refer to the programme guide for more details regarding the evaluation and other aspects of the practical component.

Instructions regarding the practical work

For each of the problems given in the assignment you are expected to:

- Compile and run the program and show it to the counsellor at your centre.
- Get a flow chart and the source code of the program signed by the counsellor.

- You are required to maintain a file of these signed programs along with the output of your programs. This file will be a part of your internal assessment and you will have to produce it at the time of the final practical exam. So please keep it neat, systematic, updated and handy.

You may like to refer to some books on the subject and try to solve some exercises given there to get a better grasp of the techniques discussed in this course. Below given are some list of books and online sources for further reading.

1. James D. Foley, Andries van Dam, Steven K. Feiner and John F. Hughes, Computer Graphics, Principles and Practice, Addison Wesley, 1997.
2. F. S. Hill, Jr., Computer Graphics using OpenGL, Pearson Education (second edition), 2006.
3. David F. Rogers, Procedural Elements of Computer Grapgics, Tata McGraw Hill (second edition), 2006.
4. Edward Angel, Interactive Computer Graphics: A Top Down Approach Using OpenGL (fifth edition), Pearson Education, 2009.
5. https://nptel.ac.in/courses

We hope you enjoy the course!

Indira Gandhi National Open University
School of Sciences

Block

1

FUNDAMENTALS OF COMPUTER GRAPHICS

Curriculum Design Committee

Dr. B.D. Acharya
Dept. of Science & Technology, New Delhi

Prof. Adimurthi
School of Mathematics
TIFR, Bangalore

Prof. Archana Aggarwal
CESP, School of Social Sciences
JNU, New Delhi

Prof. R.B. Bapat
Indian Statistical Institute
New Delhi

Prof. M.C. Bhandari
Dept. of Mathematics
IIT, Kanpur

Prof R. Bhatia
Indian Statistical Institute, New Delhi

Prof. A.D. Dharmadhikari
Dept. of Statistics
University of Pune

Prof. O.P. Gupta
Dept. of Financial Studies
University of Delhi

Prof. S.D. Joshi
Dept. of Electrical Engineering
IIT Delhi

Dr. R.K. Khanna
Scientific Analysis Group, DRDO, Delhi

Prof. Susheel Kumar
Dept. of Management Studies
IIT, Delhi

Prof. Veni Madhavan
Scientific Analysis Group
DRDO, Delhi

Prof. J.C. Misra
Dept. of Mathematics
IIT, Kharagpur

Prof. C. Musili
Dept. of Mathematics and Statistics
University of Hyderabad

Prof. Sankar Pal
ISI, Kolkata

Prof. A.P. Singh
PG Dept. of Mathematics
University of Jammu

Faculty Members
School of Sciences, IGNOU
Prof. Poornima Mital
Dr. Atul Razdan
Prof. Parvin Sinclair
Prof. Sujatha Varma
Dr. S. Venkataraman

Course Design Committee

Prof. Sujatha Varma
Director (SOS), IGNOU

Prof. Chandan Singh(Retd.)
Department of Computer Science,
Punjabi University, Patiala

Prof. S. Balasundaram (Retd.)
School of Computer and Systems Sciences,
JNU, New Delhi

Faculty Members
School of Sciences, IGNOU

Prof. Parvin Sinclair
Dr. S. Venkataraman
Dr. Deepika
Dr. Pawan Kumar

*The syllabus was designed and unitised as per the decision of the Course Expert Committee which met on 17th March, 2021.

Block Preparation Team

Prof. B.S. Panda (Editor)
Dept. of Mathematics
IIT Delhi

Dr. Pawan Kumar
School of Sciences
IGNOU

Course Coordinator: Dr. Pawan Kumar

Acknowledgements: To Mr.Surender Singh Chauhan for typesetting the manuscript and preparing the CRC.

……., 2023

ISBN-8]-

BLOCK INTRODUCTION

This is the first block of the course Computer Graphics. In this block which, is divided into 3 units we discuss certain basic concepts of computer graphics.

In Unit 1 we start with a brief background about the applications of computer graphics. Then we discuss what are video display devices and describe the working procedure of a cathode ray tube. We explain what are raster and random scan displays. Finally, we talk about input and hard copy devices.

Unit 2 mainly focuses on drawing of basic output primitives starting from a single point through straight lines and curves. We have discussed here some well algorithms for line, circle and ellipse drawing, and ways to generate other curves as well.

In **Unit 3**, we discuss what are filled area primitives, and describe some of the methods involved in filling regions such as polygons. We also talk about character generation methods.

Lastly, we remind you once again to carefully go through the solved examples, and to attempt all the exercises in each unit. This will help you grasp the theory better.

NOTATIONS AND SYMBOLS

$T(a, b)$	2D translation by (a, b)
$R(\theta)$	2D rotation about the origin by θ
$S(\alpha, \beta)$	2D scaling by α, β
$R(a, b; \theta)$	2D rotation by θ about pivot (a, b)
M_x	reflection about x-axis
M_y	reflection about y-axis
$M_{x=y}$	reflection about the line $y = x$
$M_{y=-x}$	reflection about the line $y = -x$
$S_x^{shear}(\alpha)$	x-direction shear by α
$S_y^{shear}(\beta)$	y-direction shear by β
$S^{shear}(\alpha, \beta)$	simultaneous shear by α, β
$T(a, b, c)$	3D translation by (a,b, c)
$R_x(\theta)$	3D rotation matrix about x-axis by θ
$R_y(\theta)$	3D rotation matrix about y-axis by θ
$R_z(\theta)$	3D rotation matrix about z-axis by θ
$S(\alpha, \beta, \gamma)$	3D scaling by α, β, γ

UNIT 1

BASICS OF INPUT/OUPUT DEVICES

Structure **Page No.**

1.1 INTRODUCTION

With the advancement of computer technology, computer graphics has become a practical tool used in real life applications in diverse areas such as science, engineering, medicine, business, industry, art, entertainment, education, etc. Due to such widespread applications and utility of computer graphics, nowadays a broad range of graphics hardware and software systems are available. You must be using a wide variety of input devices and graphics software packages while doing your day-to-day work on your personal computers. In this unit, we shall familiarise you with the basic features of input and output devices used in computer graphics systems.

We shall start the unit by illustrating in Section 1.2, the diversity of applications of computer graphics in designing of buildings, automobiles, aircraft, space crafts etc., to generate illustrations and slides for use with projector, for making motion pictures, music videos, television shows, models used for educational aid and many more. Section 1.3 introduces you to various video display devices and their components. Technologies available for displaying and printing graphics objects, including input devices, are discussed in Section 1.4.

Objectives

After studying this unit, you should be able to:

- describe the widespread applications of computer graphics in different fields;
- describe the functioning of various display devices;
- distinguish between raster scan and random scan display techniques;
- understand and use the terms such as refresh rate, resolution, aspect ratio, horizontal and vertical retrace;
- understand the use of various input and hardcopy devices.

1.2 APPLICATIONS OF COMPUTER GRAPHICS

When you open an application on a computer, you simply click on a graphics object, called icon. When you watch TV, several times you find that one entity changes its shape to become another entity in many advertisements. For example, a *running cheetah turning to a car tyre*. This is called morphing in computer graphics. You must have watched Hollywood films Jurassic Park and Avatar. Such science fiction films use techniques that are heavily dependent on computer graphics. For planning, designing and construction of buildings, nowadays many softwares show the entire plan, front elevation, top elevation and a walkthrough inside and around the building. All these constitute only a glimpse of what we mean by applications of computer graphics. Below given are some common areas where computer graphics is extensively used.

Movies and Entertainment

Computer graphics is widely used in the production of movies, TV programs and commercials. You must have seen interaction of humans with machines or animals in science-fiction (sci-fi) movies. Graphics and special effects have become an integral part of today's advertisements. Computer games are heavily dependent on animations and simulations.

Computer Art and Computer-Aided Design

Modern art often uses the computer graphics tools for creating beautiful pictures and scenes. Artists use a variety of methods such as **paint brush** program to design images and patterns. Computer-Aided Design (CAD) is a computer graphics tool used for designing of buildings, automobiles and aeroplanes. For instance, look at the wire-frame picture of a car model given in Fig. 1, below.

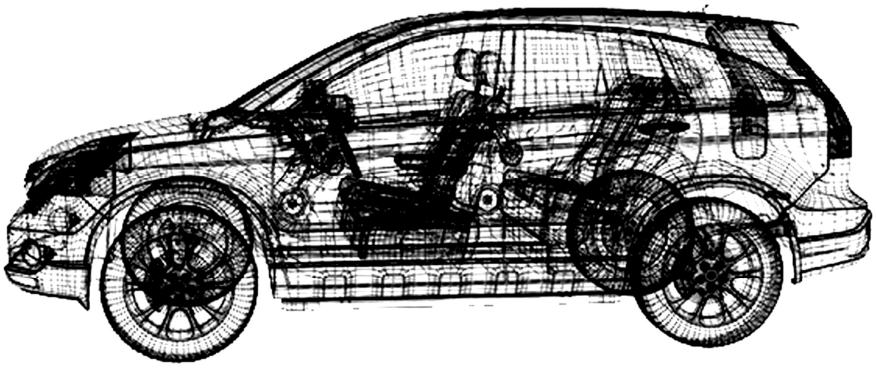


Fig.1: A wireframe model of car

Such a model is usually analysed for creating a particular shape, visualising the interior or estimating the weight before producing the final design.

Presentation graphics

Another major area of application is **presentation graphics** which are used for creating reports, or slides with projectors. Presentation graphics is normally used for summarising mathematical and scientific data for report, consumer information brochures, geographical maps, etc. The data are presented in a way that is easy to understand, and reveals the key features of the presentation.

Education and Training

Nowadays educational materials, especially at the primary level, are equipped with lot of graphics and animations. Computer graphics is also used to educate or train personals in different fields such as air traffic control. Flight simulation and ship navigation are classic examples. Most simulators provide graphics screens for visual operation. For instance look at the automated driving simulators in Fig. 2.

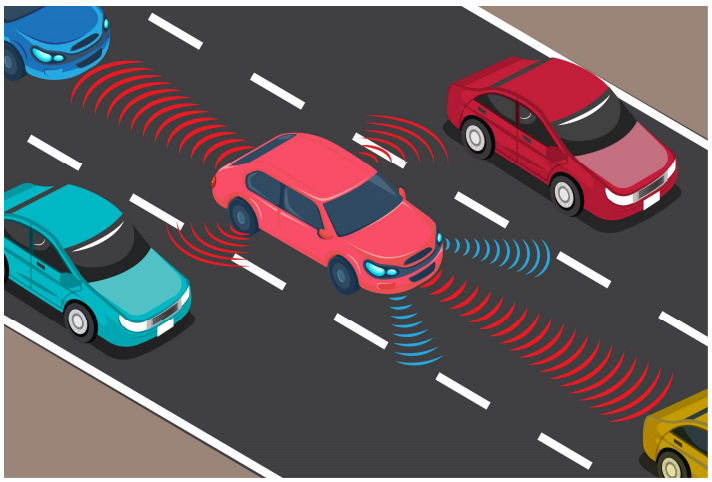


Fig.2: Automated Car Simulator

Graphical User Interfaces

Earlier Softwares were provided in Command line interfaces (CLIs), for instance LATEX. Now-a-days, the software packages are available in a graphical user interface (GUI). A GUI comes usually in the form of a window, which consists of several small icons, and sub-windows. An icon or a sub-window can be activated by a click of mouse. For instance, the desktop of your computer is itself a GUI. Another example is the CodeBlocks GUI which looks like the one given in Fig.3.

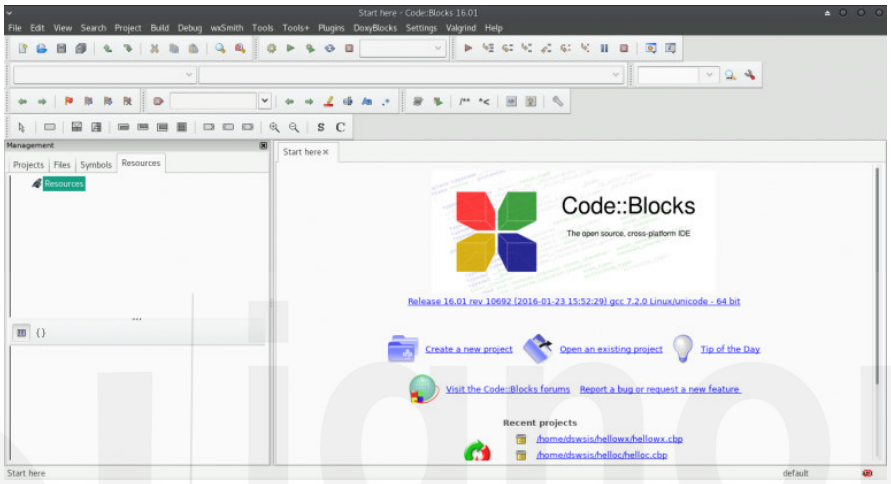


Fig.3: The CodeBlocks GUI

After reading this section you must have been motivated to learn computer graphics. Before we proceed further, you may try the following exercises.

E1) List five different areas of applications of computer graphics.

E2) Explain how computer graphics would play a useful role in education and training.

Let us now discuss some of the output devices used in a graphic system.

1.3 VIDEO DISPLAY DEVICES

This section gives you an introduction to graphics display devices. Details of architecture and functioning of some of the important display technologies have been discussed here.

Usually the primary output device in a graphics system is a video monitor. The operation of a video monitor is based the standard cathode ray tube (CRT) design. Let us look at the general functioning of a CRT.

Refresh Cathode Ray Tube

The basic design of a cathode ray tube is shown in Fig.4 below, which is similar to the display of a television set.

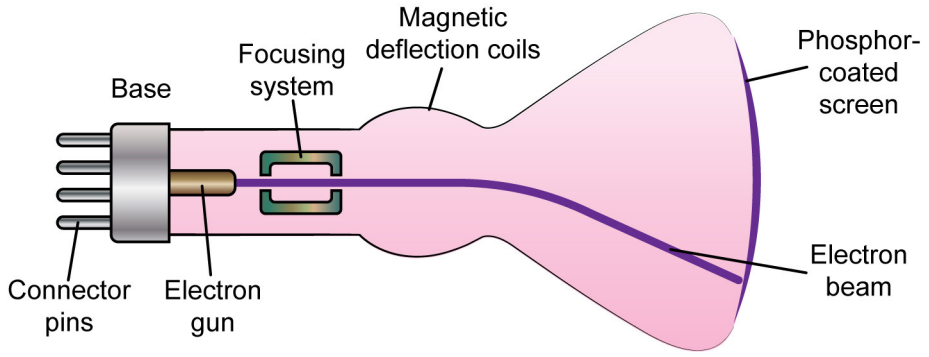


Fig.4: Basic design of a CRT

It has an electron gun, a focusing system, and a magnetic field, and a phosphor–coated screen. An electron beam emitted from the electron gun passes through the focusing system and the magnetic field. The magnetic field deflects the electron beam to hit the screen at the desired locations. The point where an electron beam strikes the phosphor-coated screen starts glowing and the maximum brightness is observed on the focus (centre) of the point. The brightness tends to reduce radically as per the Gaussian distribution. Actually at the centre of the strike point the energy level of phosphor atoms is highest, and it gradually spreads over the nearby phosphor atoms.

In a typical CRT the picture displayed does not remain on the screen for long. In order to keep the picture visible for longer time, the electron beam is directed back over the same points again and again. The display devices using this technique are called **refresh CRTs**.

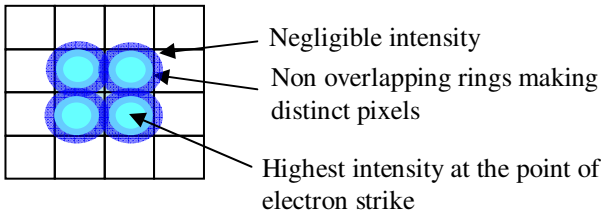
Persistence (of phosphor): Persistence is defined as the time taken by the light emitted from the screen to decay to $\frac{1}{10}$ th of its maximum intensity.

In order to keep the pixel glowing, one requires to re-strike the electron gun on the same point before the screen point fades away, that is, before the energy level of the phosphor atom reduces to a certain limit as defined in the term persistence. This is known as **refreshing** of the pixel. In fact the rate of refreshing depends on the type of phosphor used and also on the human perception limits. Before the human eye could catch that the pixel is beginning to reduce its intensity, the pixel must be refreshed. In refresh CRT monitors, the entire screen frame is refreshed normally 30-60 times in a second. This is called **refresh rate**.

Lower persistence phosphors require faster refresh rate and higher persistence phosphors require lower refresh rates. The lower persistence phosphors are used in monitors that are specially designed for animations,

whereas higher persistence phosphors are mostly used in complex static representation of objects such as in CAD/CAM (Computer Aided Design/Manufacturing).

Resolution: Consider the following four spots shown on the screen of a display device.



The outermost ring of each spot has almost negligible intensity, the second outer circle specifies the diameter which becomes visible to human eyes making a non-overlapping range of spots. The innermost spot is showing maximum intensity. So the non-overlapping points are actually those points which can be clearly distinguished. Usually two adjacent spots are distinct if the distance between their centres is greater than the diameter at which each spot has intensity of about 60% of the maximum intensity.

Each display device can show a fixed number of points in a row, without overlapping. In a layman's term if a CRT monitor can display 1024 points in a row and the total rows that can be displayed simultaneously are 768, then the resolution of the monitor is 1024×768. Each single point of display, which forms one unit of display, is called a **pixel** (short form of picture-cell). So a graphics display device is treated as a graph paper with pixel as the smallest unit of display. In general, if a display device has resolution $p \times n$, then it means that the device has p pixels on a row and it contains n rows. Larger the resolution smaller is the size of the pixel.

Aspect ratio: The ratio of the number of vertical points to the number of horizontal points necessary to produce equal length lines in both directions on the screen is called the aspect ratio. If we are given the height and the width of the screen, then the aspect ratio is defined as **aspect ratio** $= \frac{\text{height}}{\text{width}}$.

Horizontal and vertical retrace: In a refresh CRT monitor the contents of a picture are displayed from top to bottom, line by line. Each line is plotted from left to right, and it is called scanline. The time taken by an electron beam to

return to the left most point on the next horizontal line after refreshing one horizontal line is called **horizontal retrace**. The beam is shut off (blanked) during the period of retrace. Generally, about 83% of the total horizontal retrace time is spent tracing the line. The remaining 17% is spent bringing the beam back to the left side, before starting the next scanline.

In a refresh CRT monitor, the time taken by an electron beam to return to the top left point on the monitor from the end of the last scanline is called **vertical retrace**. The set of all scanlines that are displayed simultaneously is called a **frame**.

Let us now consider an example to illustrate the use of all these terms.

Example 1: Consider a non-interlaced raster monitor with a resolution of 1024×768, a refresh rate of 30 frames per second, a horizontal retrace time of 3 microseconds and a vertical retrace time of 300 microseconds. What is the fraction of total retrace time per frame spent in retrace of the electron beam?

Solution: Refresh rate $= 30$ frames per second. Therefore, 1 frame will be refreshed in $1/30$ seconds. This contains horizontal retrace time of 3 microseconds per horizontal retrace. That means total horizontal retraces will take 3×768 microseconds $= 0.002304$ seconds. Then a vertical retrace will take 300 microseconds $= 0.0003$ seconds. Therefore, total refresh time per frame $0.002304 + 0.0003 = 0.002604$ seconds. So the fraction of total refresh time per frame spent in electron beam retrace is given by

$$\frac{0.002604}{0.033333} \cong 0.07812 \text{ seconds.}$$

$1 \text{ micro second} = 10^{-6} \text{ sec.}$

***

Raster Scan Display

The most commonly used technology for displaying graphics on a CRT-based monitor is the **raster scan display**. For instance, TV sets and printers use this technology. In a raster scan system, an electron beam moves across the screen – from left-to-right in each scanline, and from top-to-bottom from one scanline to another. When the beam passes through a scanline, its intensity is turned on at the points which are to be shown on the screen and turned off at other points. The beam intensity values for each screen point (pixel) is stored in the memory called **refresh buffer** or **frame buffer**. The range of intensity values depends on the capability of the system. For instance, a black-and-

white system requires only one bit (for storing 0 or 1) for each pixel position, where 0 corresponds to off and 1 corresponds to on.

In some raster systems each frame is displayed in two passes – displaying odd numbered scan lines in one pass, and the even numbered scan lines in the other pass. Such system are also called **interlaced** system (eg. a TV set). A **non-interlaced** system is one where each frame is displayed in one go.

Random Scan Display

Random scan display is another display technology for CRT-based monitors. In a random scan display the electron beam is directed towards the screen positions where the picture is drawn. This technology significantly differs from the raster scan technology in the sense that it draws one line at a time. Therefore, we also call it **vector display**. It is best suitable for line drawing applications, and is not useful for displaying realistic pictures or scenes. The picture definition is stored in the memory as a set of line drawing instructions, rather than the set of intensity values for each pixel. Picture resolution in such a system is very high because the electron beam directly follows the line path. A raster scan system, in contrast, produces jagged lines plotted at discreate points.

Colour CRT Monitors

Colour CRT Monitors, as the name suggests, display coloured pictures using a combination of phosphors each emitting a distinct light. Two basic techniques for producing colour displays using a colour CRT monitor are beam-penetration method and shadow-mark method, as described below.

Beam Penetration Method: It uses two layers of phosphors, usually red and green, coated on the inner side of the screen. The displayed colour depends on how far the electron beam penetrates into the phosphor layers. A slow beam only excites the outer red layer and hence produces only red, whereas a fast-moving beam penetrates into both the layers and produces green colour. Beams at medium speeds penetrate at intermediate positions on the layers, producing orange and yellow colours. Thus beam-penetration method shows a picture in only four colours.

Shadow-Mask Method: A shadow – mask method uses three phosphor colour dots – red, green and blue – at each pixel position. A CRT with such a system has three electron guns – one for each colour dot, and a shadow – mask grid behind the screen. Fig. 5 shows the operation of a shadow – mask CRT.

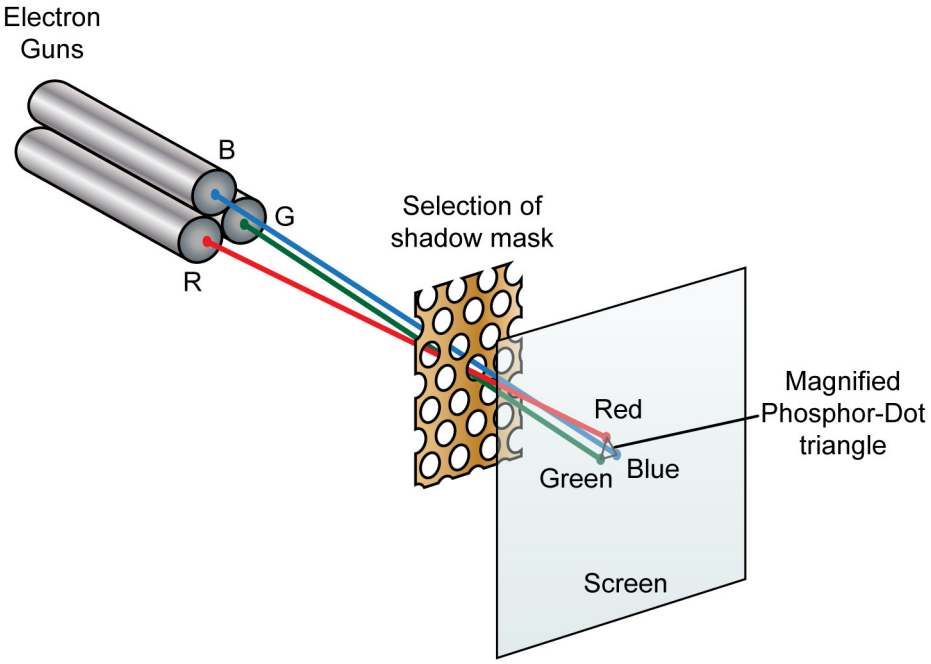


Fig. 5: A Shadow-Mask CRT

The shadow-mask grid contains a series of holes aligned with the phosphor dot patterns. When the three beams pass through a hole in the shadow mask, they activate a dot triangle. Since a human eye cannot distinguish between the three dots, a single dot appears on the screen.

You may now test your understanding of what you have learnt by solving the following exercises.

E3) Compute the following:

a) Size of 420×300 image at 240 pixels per inch.

b) Resolution (per square inch) of 3×2 inch image that has 768×512 pixels.

c) Height of the resized image 1024×768 to one that is 640 pixels wide with the same aspect ratio.

d) Width of an image having height of 6 inches and an aspect ratio 1.5.

E4) Consider two raster systems with resolutions of 640 by 480, and 1280 by 1024. What size frame buffer (in bytes) is needed for each of these systems to store 12 bits per pixel?

E5) Suppose we have a video monitor with a display area that measures 12 inches across and 9.6 inches high. If the resolution is 1280 by 1024 what is the diameter of each pixel (in cm)?

E6) Consider a noninterlaced raster monitor with resolution $p \times n$, a refresh rate of r frames per second, a horizontal retrace time of t_{horiz}, and a vertical retrace time of t_{vert}. What is the fraction of the total refresh time per frame spent in retrace of the electron beam?

E7) What is the fraction of the total refresh time per frame spent in retrace of the electron beam for a noninterlaced raster system with a resolution of 1280 by 1024, a refresh rate of 60 Hz, a horizontal retrace time of 5 microseconds, and a vertical retrace time of 500 microseconds?

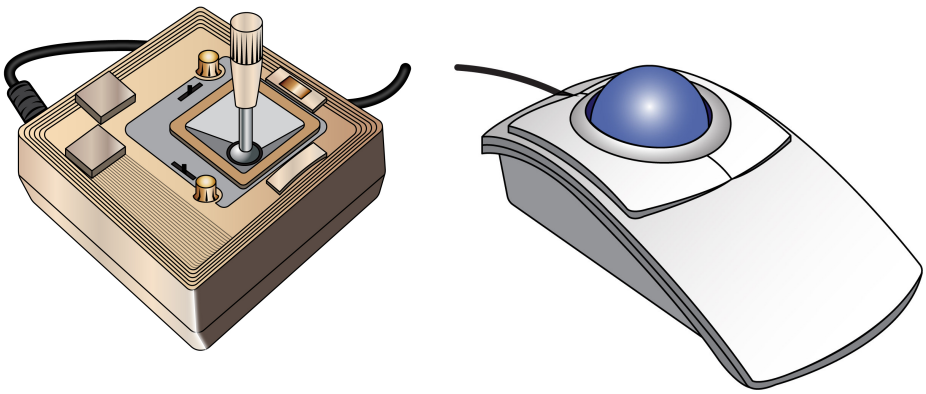
The next section introduces you to some of the technologies used in graphics systems.

1.4 INPUT DEVICES AND HARD COPY DEVICES

In this section we shall give you a brief introduction of the input devices and hard copy devices used at the computer graphics workstations.

Input Devices

The most commonly used input devices, in everyday use, are **keyboard** and **mouse**. A keyboard is normally, used to input character and numeric data, whereas a mouse is used for selecting and moving the curser positions. Another type of input devices are **joystick** and **trackball**. A joystick consists of a small vertical lever (called the stick) mounted on a base which can be moved left, right, forward and backward. The movement of the stick corresponds to the movement of the screen cursor. A trackball, on the other hand, has a ball that can be rotated in any direction with the palm of the hand to move the cursor. A potentiometer is attached in a joystick as well as in a trackball for measuring the magnitude and the direction of movement. Fig. 6 shows a joystick and a trackball.



(a) A joystick **(b)** A trackball

Fig. 6

Digitisers and **image scanners** are also important input devices. A digitiser is commonly used for drawing, painting or interactively selecting coordinate positions on a two-dimensional or a three-dimensional object. A common example of a digitiser is a graphics tablet, which is used to input the coordinates of a two-dimensional object. Three-dimensional digitisers use sonic or electromagnetic transmissions to record positions. The task of an image scanner is to store and process the drawings, graphs or photos. It uses an optical scanning mechanism to store the information about the picture. This information can then be used to rotate, scale, crop, or resize the picture.
The list of input devices also include data gloves, touch panels, light pens and speech recognition systems.
Now we come to the hard copy devices.

Hard Copy Devices

We are often interested in taking a picture on a paper or a photographic film. The most commonly used hard copy devices are printers and plotters. Pictures are drawn on a hard copy by transferring the information from the frame buffer to the picture medium dot by dot. The quality of a picture produced through a printer depends on the dot size and the number of dots per inch.
A **dot matrix printer**, which is a classical model, places dots at a density of roughly 70 dots per inch (dpi). Nowadays, such printers are less common, and can be seen on railway ticket counters, for instance. **Laser printers**, on the other hand, can produce densities of 600 or more dots per inch, and hence print high quality images. A laser printer throws a laser beam on a rotating drum that is coated with some photoelectric material. The toner is then applied to the drum, which transfers to the paper. It is the fast moving laser beam that gives precision to a laser printer.

The following exercises will also be useful to you.

E8) List two advantages of laser printers over dot matrix printers.

We now end the unit by giving a summary of what we have covered here.

1.5 SUMMARY

In this unit, we have covered the following points:

1. Compute graphics has wide spread applications such as Computer-Aided Design, Computer Art, Presentation Graphics, Entertainment and Education and Training.

2. We have described the functioning of a refresh cathode ray tube.

3. The terms such as persistance, refresh rate, resolution, aspect ratio, horizontal and vertical retrace are explained.

4. We have seen how random scan display technique differs from the raster scan display.

5. The difference between the beam penetration method and the shadow mask method is also explained.

6. Finally, we have seen functioning of the input devices (keyboard, mouse, trackball, joystick etc.) and hard copy devices (laser printers, dot-matrix printers).

1.6 SOLUTIONS/ANSWERS

E1) Five major areas of applications of computer graphics are:

i) Study of molecular structures.

ii) Modelling and simulation of automobile items.

iii) Building design.

iv) Visual effects in movies.

v) Making professional presentations more effective by inclusion of graphics and animation.

E2) Computer graphics has occupied an important role in education and training. One of the important areas where computer graphics plays a crucial role is computer based learning. For example, in order to teach a course on engineering drawing, it is better to make use of softwares like **AutoCAD** or **SolidWorks** that give very convenient and efficient ways of generating 2D and 3D geometric objects, used in composition of various objects. For study of molecular structures, visualization of these structures would help understand better the concept of bonds etc. If one wants to study electronic circuits and component design, it is always better to first make a circuit in a supporting software and test its validity, then make a final electronic circuit. There are a variety of different areas in education and training where use of graphics enhances the effectiveness of lectures/training sessions. For example, presentation of lectures with the help of graphics and animation leads to more effective styles of teaching and is adapted worldwide in higher education and also

in elementary education in a number of areas. Especially for making the learning a fun for kids, graphics has been extensively used in many such learning programmes.

E3) a) Size of 420×300 image at 240 pixels per inch $=1.75\times1.25$ sq. inches.

b) Resolution of 3×2 inch image that has 768×512 pixels $=256$ pixels per inch.

c) If h is the height of the resized image, then $\frac{768}{1024}=\frac{h}{640}$, which implies $h=640$ pixels.

d) Width of an image having height of 6 inches and an aspect ratio 1.5 is

$$\text{width}=\frac{\text{height}}{\text{aspect ratio}}=\frac{6}{1.5}=4 \text{ inches.}$$

E4) The first system has $640\times480=307200$ pixels. Since one pixel needs 12 bits to store the information, number of bytes in the frame buffer of each of this systems is:

$$\frac{307200\times12}{8}=460800\ bytes=450\ Kbytes.$$

The second system has $1280\times1024=1{,}310{,}720$ pixels. In this system the number of bytes in the frame buffer is

$$\frac{1310720\times12}{8}=1966080\ bytes=1920\ Kbytes.$$

Remember
1 bytes = 8 bits
1 kbytes = 1024 bytes

E5) The monitor is 12 inches wide and accommodates 1280 pixels in a scan line. Similarly, it is 9.6 inches high and accommodates 1024 pixels. Therefore the area of the rectangle bounding one pixel should be

$$\frac{12}{1280}\times\frac{9.6}{1024}=0.0094\times0.0094.$$

Therefore the pixel diameter is 0.0094 inch, which is 0.024 cm.

E6) Refresh rate is r frames per second. Therefore one frame is refreshed in $1/r$ seconds. Once horizontal refresh takes t_{horiz} and one vertical retrace takes t_{vert} time. Since there are n scan lines, total retrace time

for one frame will be $n \times t_{\text{horiz}} + t_{\text{vert}}$. So the fraction of retrace time $1/\text{r}$ for one frame will be

$$\frac{n \times t_{\text{horiz}} + t_{\text{vert}}}{1/r} = r(n \times t_{\text{horiz}} + t_{\text{vert}}).$$

E7) Substitute $n = 1024$, $t_{\text{horiz}} = 5$ microseconds, $t_{\text{vert}} = 500$ microseconds, $r = 60\ Hz$ in the solution to E6. The required answer is 0.34 sec.

E8) Two advantages of laser printers over dot matrix printers are:

- laser printers produce high quality pictures
- laser printers can print texts and pictures very rapidly than the dot matrix printers.

UNIT 2

GENERATING GRAPHICS PRIMITIVES

Structure **Page No.**

2.1 INTRODUCTION

In Unit 1, you were introduced to the applications of computer graphics and the basics of input/output devices. In this unit, we shall study some of the basic algorithms that help create graphics output primitives. By graphics output primitives, we mean the basic building blocks required to represent and model a real life object. In this sense, output primitives are simple geometric objects that can be easily constructed such as points, lines, polygons, or conic sections.

Section 2.2 introduces you the basics of writing computer graphics programs. We shall study an algorithm to plot points on a straight line segment in Section 2.3. Algorithms for plotting circles and ellipses are discussed in Sections 2.4 and 2.5 respectively. We shall also discuss a few more important output primitive curves in Section 2.6, which have been extensively used in graphics. The aim of this unit is to make you acquainted with the algorithms and their implementation in C language using OpenGL. OpenGL is an industry standard graphics library that gives enormous facilities for object modelling and

rendering. In order to know more about OpenGL, you are advised to refer to the **Appendix A** given at the end of Block 3.

Objectives

After reading this unit, you should be able to:

- generate a line segment using one of the two important line drawing algorithms – DDA and Bresenham algorithm;
- draw a circle with specified radius and centre using Midpoint Circle Drawing Algorithm;
- draw an ellipse if the centre, major and minor axes are specified;
- draw other curves including Bézier curves.

Let us start with understanding the functioning of various input and output devices.

2.2 OUTPUT PREMITIVES

Every picture generated by a computer is composed of basic geometric shapes such as points, lines, polylines, text or filled regions, and these basic shapes are called **output primitives**. Thus before learning the advanced computer graphics concepts, we need to understand how these basic shapes are created. You may refer the OpenGL guide (Appendix A) for basic OpenGL functions and some simple programs such as opening a window.

Drawing Points

Let us discuss how to draw points in a window. The basic command for drawing a point is `glVertex2i(x,y)` Here `gl` refers to the OpenGL library, Vertex is the root command, the suffix `2i` indicate that the arguments are two-dimensional points having integer coordinates. Of course, the command cannot be used directly. It requires the following environment.

```
glBegin(GL_POINTS);
  ..........
  ..........
glEnd();
```

Here, `GL_POINTS` is a built-in constant. Suppose, you wish to draw two points $P(0,0)$ and $Q(200,210)$. This can be done using the following code.

```
glBegin(GL_POINTS);
      glVertex2i(0,0);
      glVertex2i(200,210);
glEnd();
```

We shall put this piece of code inside the `display()` function, between the commands `glClear()` and `glFlush()`. There are two more things we need

to specify. These are the point size and point colour. The command for setting the point size is `glPointSize()`, and for point colour is `glColor3f(R, G, B)`. Of course, to get their effect both these commands must appear before the `glVertex2i()` command. Now, the program for drawing two points is given below in Listing 2.

```
// Drawing two points using OpenGL
#include<stdio.h>
#include<GL/glut.h>

void myInit (void)
{
    gluOrtho2D(-500, 500, -500, 500);  // window dimension
    glClearColor(0.0, 0.0, 1, 0.0);    //window colour
    glColor3f(1, 0, 0);                // point colour
    glPointSize(10);                   //point width
}

void display (void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POINTS);
        glVertex2i(0,0);
        glVertex2i(200, 210);
    glEnd();
    glFlush();
}

int main (int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

    glutInitWindowSize(640, 480);          // window size
    glutInitWindowPosition(0, 0);          // window position
    glutCreateWindow("Drawing points");  // window title

    myInit();
    glutDisplayFunc(display);
    glutMainLoop();
}
```

Listing 1: A program for drawing points

Just run this program and see the output. You can draw as many points as you wish.

Drawing Lines and Polylines

To draw a line joining two points `(x1, y1)` and `(x2, y2)` we use the following piece of code.

```
glBegin(GL_LINES);
  glVertex2i(x1, y1);
  glVertex2i(x2, y2);
glEnd();
```

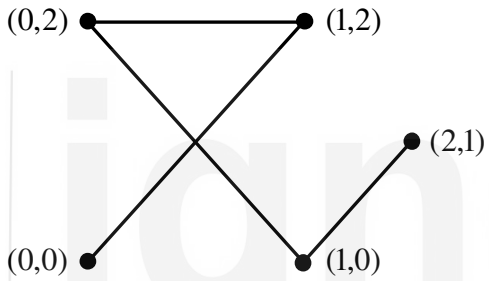
Note that here the argument of `glBegin()` is `GL_LINES`. One can draw any number of lines by specifying an even number of points inside `glBegin()`... `glEnd()`. The points are first paired up as they occur, and then the

corresponding lines are drawn. For instance, the following code draws two intersecting lines.

```
glBegin(GL_LINES);
   glVertex2i(0,0);
   glVertex2i(400,400);.
   glVertex2i(0,400);
   glVertex2i(400,0);
 glEnd();
```

Just like a point, one can set the colour of a line using the command `glColor3f()`. Moreover, a line also has its width, which can be set using the command `glLineWidth()`.

A **polyline** is a shape made of several lines joined end to end. For instance, given below is a polyline.



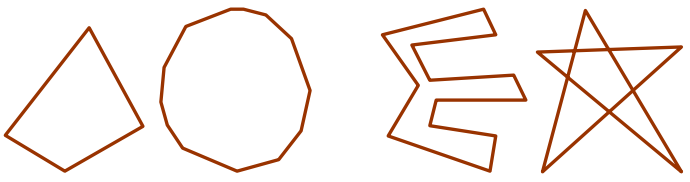
To draw such a polyline the piece of code is:

```
glBegin(GL_LINE_STRIP);
   glVertex2i(0,0);
   glVertex2i(1,2);
   glVertex2i(0,2);
   glVertex2i(1,0);
   glVertex2i(2,1);
glEnd();
```

Here the argument of `glBegin()` is `GL_LINE_STRIP`, which tells the compiler to join the consecutive points rather than joining in pairs.

Drawing Convex Polygons

You can recall that a convex polygon is a region which has the property that for every two points in it the line segment joining them is also inside it. Some examples of convex and nonconvex polygons are given below.



Convex Polygons **Nonconvex Polygons**

OpenGL can quickly render a convex polygon, by specifying its vertices between the following commands.

```
glBegin(GL_POLYGON);
.............
.............
glEnd( );
```

For instance, the following command generates a hexagon, which is not regular.

```
glBegin(GL_POLYGON);
      glVertex2i(100,100);
      glVertex2i(150,100);
      glVertex2i(200,150);
      glVertex2i(200,200);
      glVertex2i(150,250);
      glVertex2i(100,250);
glEnd( );
```

Note that by default a polygon is filled in a solid colour, determined by the current colour settings. One can also use a pattern to fill a polygon. Each polygon has two faces: a back face and a front face. You can set the fill colour and other attributes for each face separately. Vertices are specified in a counter-clockwise order in OpenGL for the front face of the polygon.

You may know that many real-world surfaces consist of polygons, which may be nonconvex, self intersecting or even having holes. Since all such polygons can be constructed from union of simple convex polygons or more precisely triangles, some routines to build more complex objects are provided in the GLU library. These routines take complex objects and tessellate them, or decompose them into simpler OpenGL polygons that can then be rendered.

Note that a rectangle can be generated using the above commands by just specifying the coordinates of its four corners within the environment `glBegin(GL_POLYGON) .... glEnd()`. But OpenGL also has a special command to produce a rectangle, and it is given below.

```
glRecti(x1, y1, x2, y2);
```

Here `(x1, y1)` is the lower left corner and `(x2, y2)` is the diagonally opposite corner of the rectangle. The rectangle is displayed with edges parallel to the coordinate axes. Further, if you want to express coordinates as array elements then you can also use the suffix code **v** for vector. Following piece of code helps you understand how to use array elements as vertices of rectangle.

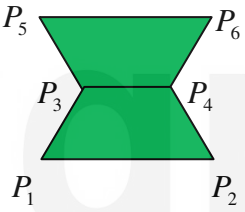
```
int P[] = {200, 100}, Q[] = {350, 250};
glRect2iv(P, Q);
```

There are many other arguments to the `glBegin()` function as given in the following table.

These primitives can also be used to construct nonconvex polygons by appropriately dividing them into triangles or quadrilaterals. Let us consider an example.

Example 1: Draw the following shape using OpenGL primitives.

Solution: We can divide the above shape into two quadrilaterals as follows.



Let the coordinates of the vertices be as follows:

$P_1 = (0, 0)$, $P_2 = (100, 0)$, $P_3 = (80, 50)$, $P_4 = (20, 50)$,
$P_5 = (100, 100)$, $P_6 = (0, 100)$.

Then the code to draw the shape is given below.

```
glBegin(GL_QUAD_STRIP);
  glVertex2f(0,0);
  glVertex2f(100,0);
  glVertex2f(20,50);
  glVertex2f(80,50);
  glVertex2f(0,100);
  glVertex2f(100,100);
glEnd();
```

***

Now try the following exercises.

E1) Modify the program given in Listing 1 to draw a plus sign made of points.

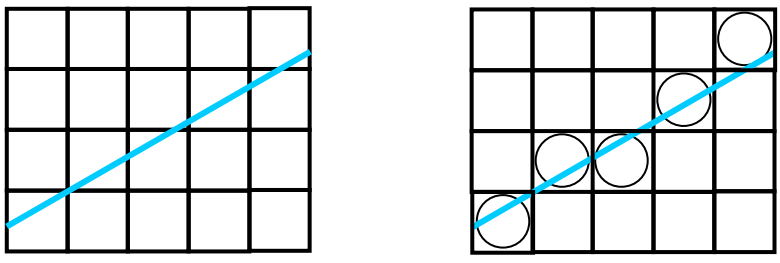
E2) Write a program to draw a cube using the OpenGL commands.

E3) Use an appropriate OpenGL primitive to draw the shape given below.

Thus far we have been using the OpenGL commands to draw points, lines or polylines. Next we shall go deeper into the methods of line segment generation.

2.3 2D LINE SEGMENT GENERATION

A digitally plotted line is basically an approximation of infinite number of points of an abstract line segment by only a finite number of points on a computer display. This is needed because the display devices can plot only a finite number of points, however large the resolution of the device may be. So, the key concept of any line drawing algorithm is to provide an efficient way of mapping a continuous abstract line into a discrete plane of computer display. This process is called **rasterisation** or **scan conversion**. These algorithms basically approximate a real valued line by calculating pixel coordinates to provide an illusion of a line segment. Since the pixels are sufficiently small, the illusion appears to be realistic to the human eyes. To understand what is meant by rasterisation, we plot a line segment on a pixel grid as shown in Fig. 1(a). The segment points are scan converted and approximated by a single shaded pixel as shown in Fig. 1(b). Here we have shown a pixel by a square, but you know that a pixel actually has a disc shape with the boundary marked as the visible portion of the dot formed by the striking electron gun. The pixel shown here is the bounding rectangle of that dot.



(a) (b)

Fig. 1

We shall discuss here two line drawing algorithms.

2.3.1 DDA Algorithm

The Digital Differential Analyser Algorithm, in short knows as DDA Algorithm,

is a scan conversion line drawing algorithm. Recall that the equation of a line with slope m and intercept b (on the y-axis) is given by

$$y = mx + b.$$

If it passes through the points $A(x_A, y_A)$ and $B(x_B, y_B)$ then the equation can be written as

$$y - y_A = \left(\frac{y_B - y_A}{x_B - x_A}\right)(x - x_A).$$

Clearly, we have $m = \frac{y_B - y_A}{x_B - x_A}$ and $b = y_A - m x_A$.

Writing $\Delta x = x_B - x_A$ and $\Delta y = y_B - y_A$, we get $m = \frac{\Delta y}{\Delta x}$. Drawing a line parallel to the coordinate axes is easy. So, let us assume that the line is not parallel to the y-axis for instance. We also assume that the line is drawn from its left endpoint to the right endpoint. To rasterise a line, we sample the values of one variable at integer pixel locations and compute the corresponding values of the other variable using the above equations. First, let us assume that $-1 \le m \le 1$. We take $\Delta x = 1$, so that $x_{k+1} = x_k + 1 \forall k = 0,1,2,\ldots$. The corresponding y values are computed as follows

$$y_{k+1} = y_k + m, k = 0,1,2,\ldots.$$

Here $(x_0, y_0) = (x_A, y_A)$, and k varies until x_k reaches x_B.

When $m > 1$, we have $\Delta y > \Delta x$. So, in this case, we sample along the y-axis and compute the corresponding x values using the equation $\Delta x = \frac{\Delta y}{m}$. It can be rewritten as

$$x_{k+1} - x_k = \frac{y_{k+1} - y_k}{m}, \; k = 0,1,2,\ldots.$$

So, taking $y_{k+1} - y_k = 1$, we get

$$x_{k+1} = x_k + \frac{1}{m}, \; k = 0,1,2,\ldots.$$

Finally assume that $m < -1$. This means $\Delta y < -\Delta x < 0$ or $-\Delta y > \Delta x > 0$. That is, the change along the y-axis is greater than the change along the x-axis. So, we take $y_{k+1} = y_k - 1$ and compute the corresponding x values using

$$x_{k+1} = x_k - \frac{1}{m}, \; k = 0,1,2,\ldots.$$

In either case $m > 1$ or $m < -1$, the variable k increases until y_k reaches y_B. We can summarise the three cases in the following table.

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

Note that in the actual implementation of the DDA Algorithm, we can combine the three cases into a single case as follows. Define $dx = x_B - x_A, dy = y_B - y_A$ and

$$D = \begin{cases} |dx|, \text{ if } |dx| > |dy| \\ |\text{dy}|, \text{ else} \end{cases}$$

Then starting with $(x_0, y_0) = (x_A, y_A)$, we iteratively compute

$$x_{k+1} = x_k + \frac{dx}{D},$$

$$y_{k+1} = y_k + \frac{dy}{D}, \; k = 0,1,2,\ldots,D.$$

Note that, by definition, D is always positive. So, these equations can be used to rasterise any line, joining (x_A, y_A) and (x_B, y_B). At each iteration, the points are rounded off to the nearest integers.

The following example illustrates how the pixel positions are computed.

Example 2: Compute the pixel positions along the line path of the line joining the points $A(5,12)$ and $B(10,8)$ using the DDA Algorithm.

Solution: Here the endpoints of the line are $A(5,12)$ and $B(10,8)$. So, $dx = 10-5 = 5$ and $dy = 8-12 = -4$. Therefore, $D = 5$.

Now the successive pixel positions are computed as follows:

k	(x_k, y_k)
0	(5,12)
1	$\left(5+1, 12-\frac{4}{5}\right) \simeq (6,11)$
2	$\left(6+1, 11-\frac{4}{5}\right) \simeq (7,10)$
3	$\left(7+1, 10-\frac{4}{5}\right) \simeq (8,9)$
4	$\left(8+1, 9-\frac{4}{5}\right) \simeq (9,8)$
5	$\left(10, 8-\frac{4}{5}\right) \simeq (10,7)$

***

The full implementation of DDA Algorithm along with the main program is given below.

```
// DDA Algorithm implementation
#include<stdio.h>
#include<GL/glut.h>

void myInit (void)
{
    gluOrtho2D(-500, 500, -500, 500);
    glClearColor(0.0, 0.0, 1.0, 0.0);
    glColor3f(1, 0, 0.0);
    glPointSize(2);
}

void dda (int xA, int yA, int xB, int yB)
{
    int dx, dy, k;
    float D, x=xA, y=yA;
    dx = xB-xA; dy = yB - yA;
    D = (abs(dx) >= abs(dy)) ? abs(dx): abs(dy);
    glBegin(GL_POINTS);
        glVertex2f(x, y);
        for ( k = 1; k <= D; k++)
        {   x = x + (float)dx/D;
            y = y + (float)dy/D;
            glVertex2f(x, y);
        }
    glEnd();
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    int xA, yA, xB, yB;
    printf("Enter the first point: ");
    scanf("%d%d", &xA, &yA);
    printf("Enter the second point: ");
    scanf("%d%d", &xB, &yB);
    dda(xA, yA, xB, yB);
    glFlush();
}

int main (int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

    glutInitWindowSize(800, 800);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Line drawing using DDA");

    myInit();
    glutDisplayFunc(display);
    glutMainLoop();
}
```

Listing 2: DDA Algorithm implementation

Remark: Although DDA Algorithm is a fast algorithm, it has certain limitations. As you draw line segments with large lengths, you will also notice that the pixels are deviating significantly from the original line segment in case of DDA

algorithm. The reason behind that is (i) round off errors in floating point operations (ii) rounding function used in selecting next pixel values.

2.3.2 Bresenham's Line Drawing Algorithm

Now we shall discuss Bresenham's Line Drawing Algorithm,which is an improvement of the DDA Algorithm. The main advantage of using Bresenham's Algorithm is that the algorithm not only efficiently picks up approximating pixels, it does so with integer calculations only. This results in speed improvement to a great extent.

Assume that we have to draw a line between the integer points $A(x_A, y_A)$ and $B(x_B, y_B)$. Also assume that $x_A \le x_B$ and A is the starting point. Thus beginning with $(x_0, y_0) = (x_A, y_A)$ we have to compute the pixel positions (x_k, y_k) along the line path until x_k reaches x_B. Let the equation of the line be

$$y = mx + b,$$

where m is the slope, and b is the intercept. Let us also assume that $0 < m < 1$. We shall look at the general case later. Thus, we have $y_0 = mx_0 + b$. We take $x_{k+1} = x_k + 1$ for $k = 0,1,2,\ldots$. Then y_{k+1} can be either y_k or $y_k + 1,$ depending upon which of them is nearest to the ideal y value. See Fig. 2 below.

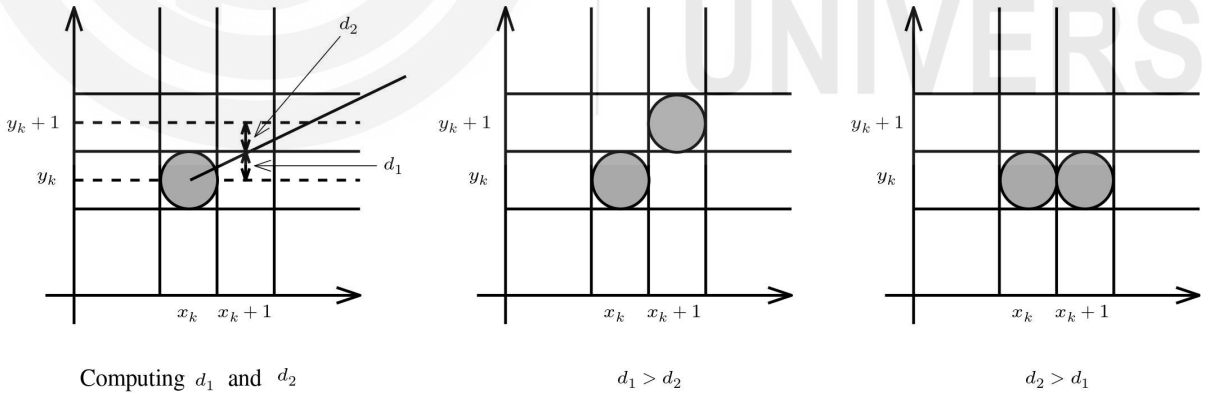


Fig. 2: y_{k+1} depends on whether $d_1 > d_2$ or $d_2 > d_1$.

To determine y_{k+1}, we compute the distances d_1 and d_2 of the ideal y value from the pixel positions y_k and $y_k + 1,$ respectively as follows.

$$d_1 = y - y_k = m(x_k + 1) + b - y_k$$

$$d_2 = y_k + 1 - y = y_k + 1 - m(x_k + 1) - b$$

So $$d_1 - d_2 = 2m(x_k + 1) - 2y_k + 2b - 1$$

$$= 2\frac{\Delta y}{\Delta x}(x_k + 1) - 2y_k + 2b - 1$$

as $m = \frac{\Delta y}{\Delta x}$, where $\Delta x = x_B - x_A$, and $\Delta y = y_B - y_A$. Now, define a **decision parameter** p_k as

$$p_k = \Delta x(d_1 - d_2) = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c,$$

where $c = 2\Delta y + \Delta x(2b - 1)$ is a constant. Now

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

$$= 2\Delta y(x_k + 1) - 2\Delta x \cdot y_{k+1} + c$$

Therefore,

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

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

Using $y_0 = mx_0 + b$, we compute the value p_0 as

$$p_0 = 2\Delta y\, x_0 - 2\Delta xy_0 + 2\Delta y + \Delta x(2b - 1) = 2\Delta y - \Delta x$$

Now, note that the signs of p_k and $d_1 - d_2$ are the same because $\Delta x \geq 0$. So, if $p_k > 0$, then $d_1 - d_2 > 0$, which means $y_k + 1$ is closer to the actual point on the line path rather than y_k. Therefore, in this case $y_{k+1} = y_k + 1$. On the other hand, when $p_k < 0$ then $y_{k+1} = y_k$. Thus the sign of the decision parameter p_k decides whether y_{k+1} would be y_k or $y_k + 1$.

For the special case the algorithm is described below.

Procedure (Bresenham's Line Drawing Algorithm):

Step 1: Input the two endpoints, and take the left endpoint as (x_A, y_A) and the right endpoint as (x_B, y_B).

Step 2: Set $(x, y) = (x_A, y_A)$ and plot (x, y).

Step 3: Compute the constants $\Delta x = |x_B - x_A|$, $\Delta y = |y_B - y_A|$, $2\Delta y$ and $2(\Delta y - \Delta x)$ once for all further computations. Set $k = 0$, and $p_0 = 2\Delta y - \Delta x$.

Step 4: If $p_k < 0$, plot $(x_k + 1, y_k)$ and set $p_{k+1} = p_k + 2\Delta y$. Otherwise, plot $(x_k + 1, y_k + 1)$ and set $p_{k+1} = p_k + 2(\Delta y - \Delta x)$.

Step 5: Increase k by 1. If $x_k = x_B$ stop. Otherwise, set $x_{k+1} = x_k + 1$ and go to Step 4.

Remember that the above procedure plots a line having slope $0 < m < 1$ only. Further, the plotting takes place from the left endpoint to the right endpoint. In general, the algorithm deals with two cases $\Delta x > \Delta y$ and $\Delta y \geq \Delta x$. Both these cases are summarised in the following table.

Table 2: Steps of Bresenham's Line-Drawing Algorithm

$\Delta x > \Delta y$	$\Delta y \geq \Delta x$
$p_0 = 2\Delta y - \Delta x$	$p_0 = 2\Delta x - \Delta y$
$(x_0, y_0) = \begin{cases} (x_A, y_A), \text{ if } x_A < x_B \\ (x_B, y_B), \text{ else} \end{cases}$	$(x_0, y_0) = \begin{cases} (x_A, y_A), \text{ if } y_A < y_B \\ (x_B, y_B), \text{ else} \end{cases}$
$h = \begin{cases} 1, \text{ if } y_A \leq y_B \\ -1, \text{ else} \end{cases}$	$h = \begin{cases} 1, \text{ if } x_B \leq x_A \\ -1, \text{ else} \end{cases}$
$x_{k+1} = x_k + 1$	$y_{k+1} = y_k + 1$
$y_{k+1} = \begin{cases} y_k, \text{ if } p_k < 0 \\ y_k + h, \text{ else} \end{cases}$	$x_{k+1} = \begin{cases} x_k, \text{ if } p_k < 0 \\ x_k + h, \text{ else} \end{cases}$
$p_{k+1} = \begin{cases} p_k + 2\Delta y, \text{ if } p_k < 0 \\ p_k + 2(\Delta y - \Delta x), \text{ else} \end{cases}$	$p_{k+1} = \begin{cases} p_k + 2\Delta x, \text{ if } p_k < 0 \\ p_k + 2(\Delta x - \Delta y), \text{ else} \end{cases}$

The algorithm is implemented in Listing 4. Let us take an example to understand how the pixel positions are computed.

Example 3: Compute the pixel positions along the line path of the line joining the points $A(2,4)$ and $B(10, 9)$, using the Bresenham's Line-Drawing Algorithm.

Solution: Here $(x_A, y_A) = (2, 4)$, and $(x_B, y_B) = (10, 9)$.

So, $\Delta x = |10 - 2| = 8, \Delta y = |9 - 4| = 5$.

Thus we have $\Delta x > \Delta y$. We compute $p_0 = 2\Delta y - \Delta x = 2$. The starting point is $(x_0, y_0) = (2, 4)$. Since $y_A \leq y_B$, we take $h = 1$. Now the successive values of p_k, x_k and y_k are given below.

k	p_k	(x_{k+1}, y_{k+1})	p_{k+1}
0	2	$(2+1, 4+1) = (3,5)$	$2 + 2(5-8) = -4$
1	-4	$(3+1, 5) = (4,5)$	$-4 + 2(5) = 6$
2	6	$(4+1, 5+1) = (5,6)$	$6 + 2(5-8) = 0$
3	0	$(5+1, 6+1) = (6,7)$	$0 + 2(5-8) = -6$
4	-6	$(6+1, 7) = (7,7)$	$-6 + 2(5) = 4$

5	4	$(7+1,7+1)=(8,8)$	$4+2(5-8)=-2$
6	-2	$(8+1,8)=(9,8)$	$-2+2(5)=8$
7	8	$(9+1,8+1)=(10,9)$	

***

The function implementing the Bresenham's Line Drawing Algorithm is given below in Listing 4.

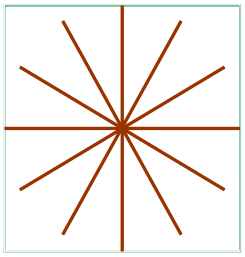
```
void bresenham(int xA, int yA, int xB, int yB)
{
    int x, y, dx, dy, h, last, p;
    dx = abs(xB-xA); dy = abs(yB-yA);
    int two_dx = 2*dx, two_dy = 2*dy;
    int two_dx_dy = 2*(dx - dy), two_dy_dx = 2*(dy - dx);
    if(dx > dy)
    {   p = two_dy - dx;
        if(xA < xB)
        {   x = xA; y = yA; last = xB;
            h = (yA <= yB) ? 1 : -1;
        }
        else
        {   x = xB; y = yB; last = xA;
            h = (yB <= yA) ? 1 : -1;
        }
        glBegin(GL_POINTS);
            while(x < last)
            {   glVertex2i(x, y);
                if(p < 0)
                    p = p + two_dy;
                else
                {   y = y+h;
                    p = p+two_dy_dx;
                }
                x++;
            }
        glEnd();
    }
    else
    {   p = two_dx - dy;
        if(yA < yB)
        {   y = yA; x = xA; last = yB;
            h = (xA <= xB) ? 1 : -1;
        }
        else
        {   y = yB; x = xB; last = yA;
            h = (xB <= xA) ? 1 : -1;
        }
        glBegin(GL_POINTS);
            while(y < last)
            {   glVertex2i(x, y);
                if(p < 0)
                    p = p+ two_dx;
                else
                {   x = x + h;
                    p = p + two_dx_dy;
                }
                y++;
            }
        glEnd();
    }
}
```

Listing 3: Bresenham's Line Drawing Algorithm Implementation

For plotting line segments using the above function, you need to call it in the main function of your program. (Just replace the dda() function in Listing 2 by bresenham() function, and run it.)

Now try the following exercises.

E4) Use DDA algorithm to produce the following output.



E5) Draw a long line segment using

(i) DDA line drawing algorithm

(ii) Bresenham line drawing algorithm

(iii) OpenGL function using GL_LINES.

Observe if DDA line segment deflects from Bresenham line segment or the graphics library line segment.

E6) Modify the dda( ) and bresenham( ) function codes to draw line segments which are (i) dotted (ii) dashed.

You can use the line drawing algorithms to draw many interesting shapes. For instance, write a function that can generate a regular polygon of n sides. When you take n large enough the shape would look like a circle. However, there is an efficient method to generate a circle, which we shall discuss next.

2.4 MIDPOINT CIRCLE DRAWING ALGORITHM

Just as a line drawing algorithm approximates a line segment by only a finite number of pixels, a circle drawing algorithm approximates a circle by a discrete set of points close to the circular arc. One of the well known algorithms for circle generation is Midpoint Circle Drawing Algorithm. The algorithm is a variant of Bresenham's Line Drawing Algorithm, and uses a decision parameter to select the pixel closest to the actual circle path, at each incremental step. Implicit equation of the circle given by

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

is used to define the decision parameter. Here r is the radius and (x_c, y_c) is the centre of the circle. Symmetry of the circle is fully exploited for its scan

conversion. Actually, we need to approximate the circle only in one octant, points in other seven octants similarly get approximated due to symmetry, as shown in Fig. 3. The decision parameter helps in deciding which pixels are closest to the circle boundary. The algorithm is iterative in nature. That is, the next pixel position and the decision parameter are computed iteratively. Just as in Bresenham's Line Drawing Algorithm, one only needs to check the sign of the decision parameter to find out the next pixel position. This makes the implementation of the algorithm very easy.

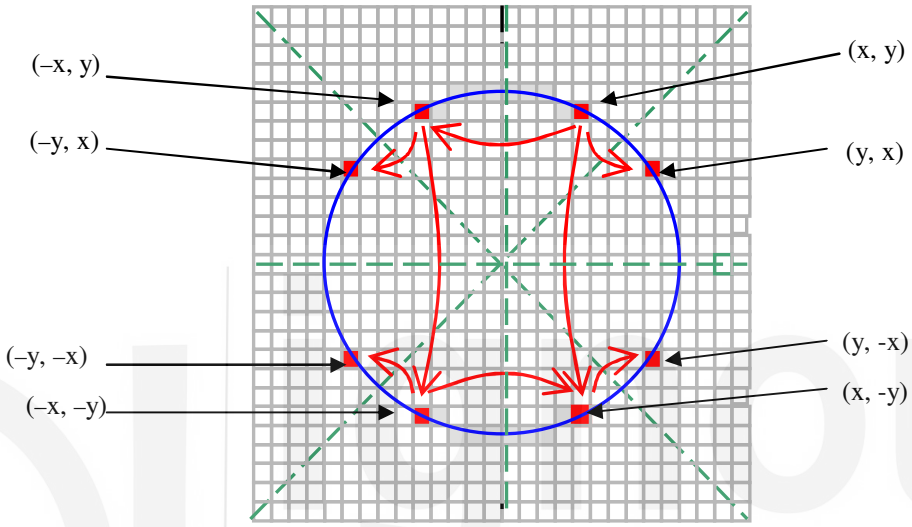


Fig. 3

We first define a circle function $f_{\text{circle}}(x,y) = x^2 + y^2 - r^2$ corresponding to the circle centered at the origin and having radius r. Then for any point (x,y), if $f_{\text{circle}}(x,y) = 0$, then (x,y) lies on the circle, and if $f_{\text{circle}}(x,y) < 0$, then (x,y) lies inside the circle, otherwise (x,y) lies outside the circle. Since we need to rasterise the circle in one octant, we choose the second octant. In this octant x varies from 0 till it reaches the condition $x = y$. Thus the first point to be plotted is $(x_0, y_0) = (0, r)$. The other 3 symmetric positions are plotted as $(0,-r),(r,0),(-r,0)$. Now assuming that the point (x_k, y_k) is plotted, the next point to be plotted is (x_{k+1}, y_{k+1}). Here $x_{k+1} = x_k + 1$, but y_{k+1} can be y_k or $y_k - 1$. The value of y_{k+1} depends on the sign of the **decision parameter** defined as

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

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

Note that p_k is evaluated at $\left(x_k+1, y_k-\frac{1}{2}\right)$ which is the midpoint of the pixel positions (x_k+1, y_k) and (x_k+1, y_k-1) near the circle path. (See Fig. 4.)

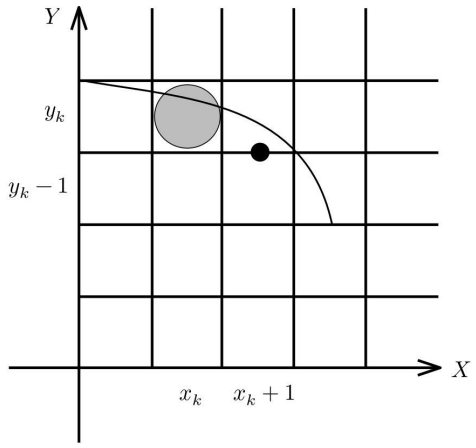


Fig. 4

Now $p_k<0$ means that $\left(x_k+1, y_k-\frac{1}{2}\right)$ lies inside the circle, and hence (x_k+1, y_k) is closer to the circle path than (x_k+1, y_k-1). Therefore, in this case, the next point to be plotted is (x_k+1, y_k). If $p_k \geq 0,$ then the next point to be plotted is (x_k+1, y_k-1). The other 7 symmetric points are computed as shown in Fig. 4. Thus, the main task is the computation of the decision parametar $p_k,$ and we do it recursively. Note that

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

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

Now it can be checked that

$$p_{k+1} - p_k = 2x_k + 3 + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k)$$

$$= 2x_{k+1} + 1 + (y_{k+1} - y_k)(y_{k+1} + y_k - 1)$$

$$= \begin{cases} 2x_{k+1}+1, \text{ if } p_k < 0 \\ 2x_{k+1}+1-2y_{k+1}, \text{ else.} \end{cases}$$

The intial value of the decision parameter is

$$p_0 = (x_0+1)^2 + \left(y_0-\frac{1}{2}\right)^2 - r^2 = \frac{5}{4} - r\,.$$

Now the procedure of the Midpoint Circle Drawing Algorithm is given below.

Procedure (Midpoint Circle Drawing Algorithm):

Step 1: Input the radius r and the centre (x_c, y_c) of the circle, and set the intial point as $(x_0, y_0) = (0, r)$.

Step 2: Set $k=0,$ and $p_0=\frac{5}{4}-r$.

Step 3: Plot the points $(x_c,y_c)+(0,r),(x_c,y_c)+(0,-r),(x_c,y_c)+(r,0)$ and $(x_c,y_c)+(-r,0)$.

Step 4: Set $x_{k+1}=x_k+1$. If $p_k<0,$ set $y_{k+1}=y_k$ and $p_{k+1}=p_k+2x_{k+1}+1$. Otherwise, set $y_{k+1}=y_k-1$ and $p_{k+1}=p_k+2x_{k+1}+1-2y_{k+1}$.

Step 5: Plot the 8 points:

$(x_c,y_c)+(\pm x_{k+1},\pm y_{k+1})$, $(x_c,y_c)+(\pm y_{k+1},\pm x_{k+1})$

Step 6: If $x_k \geq y_k$ stop. Otherwise increase k by 1 and go to Step 4.

The algorithm is easy to implement and we leave it for you as an exercise.

It is always a good **practice** to experiment the algorithm by first rasterising a circle manually.

Example 4: Rasterise the circle with centre at $(-5,4)$ and radius 8, using the Midpoint Circle Drawing Algorithm.

Solution: The centre of the circle is $(x_c,y_c)=(-5,4)$ and the radius is $r=8.$

So, we take $(x_0,y_0)=(0,8)$ and $p_0=\frac{5}{4}-8$. Since the radius is an integer, we round p_0 to the nearest integer, that is $p_0=-7$. Now the successive points are computed as shown below.

k	(x_k,y_k)	p_k	Points to be plotted
0	(0, 8)	-7	$(-5,4)+(0,\pm 8),(-5,4)+(\pm 8,0)$
1	(1, 8)	-4	$(-5,4)+(\pm 1,\pm 8),(-5,4)+(\pm 8,\pm 1)$
2	(2, 8)	1	$(-5,4)+(\pm 2,\pm 8),(-5,4)+(\pm 8,\pm 2)$
3	(3, 7)	-6	$(-5,4)+(\pm 3,\pm 7),(-5,4)+(\pm 7,\pm 3)$
4	(4, 7)	3	$(-5,4)+(\pm 4,\pm 7),(-5,4)+(\pm 7,\pm 4)$
5	(5, 6)	2	$(-5,4)+(\pm 5,\pm 6),(-5,4)+(\pm 6,\pm 5)$
6	(6, 5)		

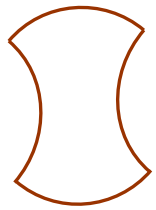
At $x_6=6, y_6=5,$ we find that $x_6>y_6,$ and so we stop the computation. The rasterised pixel positions are shown in the last column of the table.

***

You may now try the following exercises.

E7) Implement the Midpoint Circle Drawing Algorithm in C language.

E8) Draw the following object using the Midpoint Circle Drawing Algorithm. All the four boundary curves are circular arcs.



E9) Write a C code for generating concentric circles.

E10) Use the Midpoint Circle Drawing Algorithm to get the output as shown below.

An ellipse is an elongated circle. Therefore, elliptical curves can be generated by modifying circle-drawing procedures to take into account the different dimensions of an ellipse along the major and minor axes. Let us now see how this can be done.

2.5 MIDPOINT ELLIPSE DRAWING ALGORITHM

You know that a circle is symmetric in all the 8 octants, while an ellipse is symmetric with respect to four quadrants only. Therefore, to draw an ellipse, we need to determine approximating pixels in one quadrant only, namely the first quadrant. We proceed exactly the same way as we did in the case of a circle. Define a decision parameter and choose the next pixel position depending on the sign of the decision parameter. The decision parameter is defined in terms of the equation of an ellipse.

Let us consider an ellipse with its centre at the origin and its major axis along the x-axis. So its equation can be written as $\frac{x^2}{a^2}+\frac{y^2}{b^2}=1$. Consider its segment in the first quadrant. (See Fig. 5) Identify the point on this segment where the tangent line has slope -1. At this point

$$\frac{dy}{dx}=-1 \Rightarrow \frac{-2b^2x}{2a^2y}=-1 \Rightarrow 2b^2x=2a^2y.$$

Then divide the first quadrant in two regions- **Region I** where slope

$dy/dx \geq -1$ and **Region II** where slope $\frac{dy}{dx} < -1$. You may **notice** that for Region I, y values of points of the perimeter are decreasing slowly with respect to x values, than in Region II. This means for determining the approximating pixels, we have to do the following. For each increment in x value, y will either remain same or be decremented depending on the sign of a decision parameter. But in Region II, for each decremented y value, x will either be incremented or will remain the same (compare with two cases of Bresenham line drawing algorithm for slopes $|m| < 1$ and $|m| > 1$).

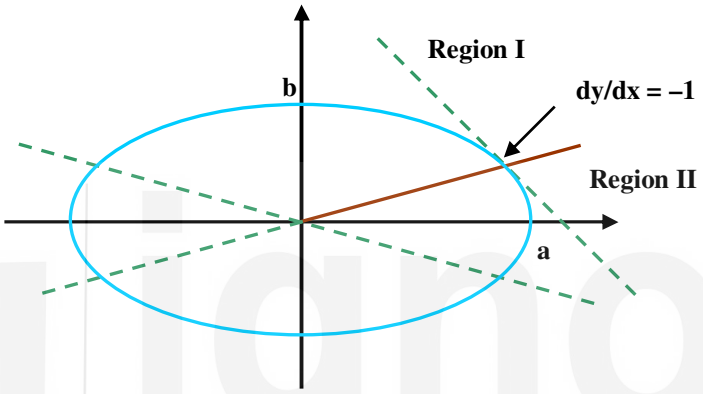


Fig. 5

To rasterise the ellipse in regions I and II, we need two distinct decision parameters.

First we consider Region I. We define an ellipse function as follows:

$$f_{\text{ellipse}}(x, y) = b^2 x^2 + a^2 y^2 - a^2 b^2.$$

For any point (x, y), it is easy to see that if (x, y) lies on the ellipse, then $f_{\text{ellipse}}(x, y) = 0$, if (x, y) lies inside the ellipse, then $f_{\text{ellipse}}(x, y) < 0$. Otherwise, $f_{\text{ellipse}}(x, y) > 0$.

In this region x varies from 0 till it reaches the condition $x = \frac{a^2}{b^2} y$. The first point to be plotted is $(x_0, y_0) = (0, b)$. The other symmetric point that is plotted simultaneously is $(0, -b)$. Now assuming that the point (x_k, y_k) is plotted (see Fig. 6), we determine the next point to be (x_{k+1}, y_{k+1}). Here $x_{k+1} = x_k + 1$, but y_{k+1} can be y_k or $y_k - 1$ depending upon the sign of the **decision parameter** p_k defined as

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

$$= b^2 (x_k + 1)^2 + a^2 \left(y_k - \frac{1}{2} \right)^2 - a^2 b^2$$

$$= b^2 x_k^2 + b^2 + 2b^2 x_k + a^2 y_k^2 + \frac{a^2}{4} - a^2 y_k - a^2 b^2$$

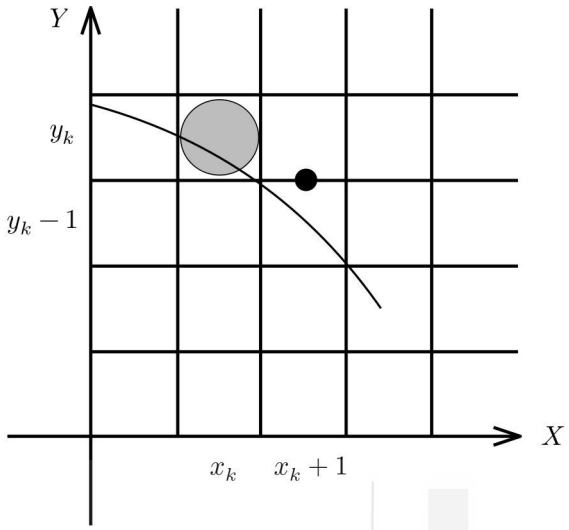


Fig. 6

Observe that $p_k < 0$ means $\left(x_k + 1, y_k - \frac{1}{2} \right)$ lies inside the ellipse, and hence $(x_k + 1, y_k)$ is closer to the ellipse path than the point $(x_k + 1, y_k - 1)$.

Therefore, in this case, we take $y_{k+1} = y_k$. And, when $p_k \geq 0$ we take $y_{k+1} = y_k - 1$. Next, we compute

$$p_{k+1} = b^2 (x_k + 2)^2 + a^2 \left(y_{k+1} - \frac{1}{2} \right)^2 - a^2 b^2$$

$$= b^2 x_k^2 + 4b^2 + 4b^2 x_k + a^2 y_{k+1} + \frac{a^2}{4} - a^2 y_{k+1} - a^2 b^2$$

Now, you can verify that

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

$$= \begin{cases} 2b^2 x_{k+1} + b^2, \text{ if } p_k < 0 \\ 2b^2 x_{k+1} + b^2 - 2a^2 y_{k+1}, \text{ else} \end{cases}$$

The initial value of p_k is

$$p_0 = f_{\text{ellipse}} \left(x_0 + 1, y_0 - \frac{1}{2} \right) = f_{\text{ellipse}} \left(1, b - \frac{1}{2} \right) = b^2 - a^2 b + \frac{a^2}{4}.$$

After rasterising the ellipse path in Region I, we reach in Region II. So, the first point of Region II is the la st point of Region I. In Region II, we decrease the successive y values by 1, and the corresponding x values remain the same or increase depending on another **decision parameter** q_k defined as follows:

$$q_k = f_{\text{ellipse}} \left(x_k + \frac{1}{2}, y_k - 1 \right)$$

$$= b^2\left(x_k + \frac{1}{2}\right)^2 + a^2(y_k - 1)^2 - a^2 b^2$$

If (x_k, y_k) is any point plotted in Region II (see Fig. 7), then the next point is (x_{k+1}, y_{k+1}), where $y_{k+1} = y_k - 1$ and

$$x_{k+1} = \begin{cases} x_k + 1, \text{ if } q_k < 0 \\ x_k, \text{ else.} \end{cases}$$

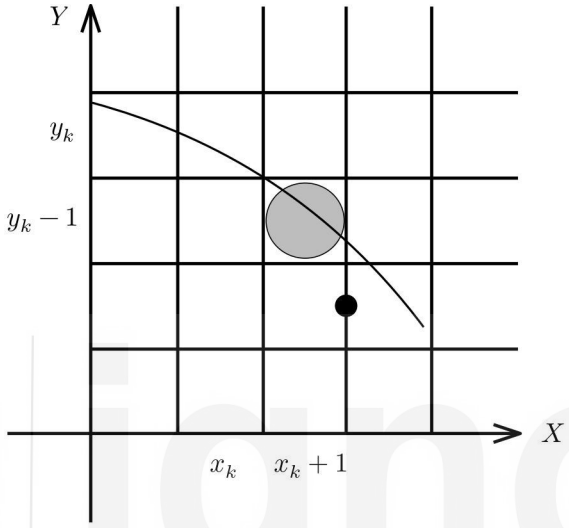


Fig. 7

Now just verify that

$$q_{k+1} - q_k = \begin{cases} 2b^2 x_{k+1} - 2a^2 y_{k+1} + a^2, \text{ if } q_k < 0 \\ -2a^2 y_{k+1} + a^2, \text{ else.} \end{cases}$$

The initial value of q_k is computed at $\left(x_\ell + \frac{1}{2}, y_\ell - 1\right)$, where (x_ℓ, y_ℓ) is the last point plotted in Region I. That is,

$$q_0 = f_{\text{ellipse}}\left(x_\ell + \frac{1}{2}, y_\ell - 1\right)$$

$$= b^2\left(x_\ell + \frac{1}{2}\right)^2 + a^2(y_\ell - 1)^2 - a^2 b^2 .$$

The computation in Region II continues as long as $y_k > 0$. For the above computations we have assumed that the ellipse is in its standard position, i.e., its centre is at the origin and the axes are the coordinate axes. If its centre is (x_c, y_c) and its axes are parallel to the coordinate axes, then we apply the transformation

$$(x, y) \to (x + x_c, y + y_c)$$

for each computed pixel position (x, y), and then plot the point $(x + x_c, y + y_c)$.

The procedure of the algorithm is described below.

Procedure (Midpoint Ellipse Drawing Algorithm):

Step 1: Input the centre (x_c, y_c), the semi-major axis a and the semi-minor axis b of the ellipse.

Step 2: Set $k = 0, (x_0, y_0) = (0, b)$ and $p_0 = b^2 - a^2 b + \frac{a^2}{4}$.

Step 3: Plot the points $(x_c, y_c) + (0, b)$ and $(x_c, y_c) + (0, -b)$.

Step 4: Set $x_{k+1} = x_k + 1$. If $p_k < 0$, set $y_{k+1} = y_k$ and

$p_{k+1} = p_k + 2b^2 x_{k+1} + b^2$. Otherwise, set $y_{k+1} = y_k - 1$ and

$p_{k+1} = p_k + 2b^2 x_{k+1} + b^2 - 2a^2 y_{k+1}$.

Step 5: Plot the four points: $(x_c, y_c) + (\pm x_{k+1}, \pm y_{k+1})$

Step 6: Increase k by 1. If $b^2 x_k \geq a^2 y_k$ go to Step 7, otherwise, go to Step 4.

Step 7: Set $q_0 = b^2\left(x_k + \frac{1}{2}\right)^2 + a^2 (y_k - 1)^2 - a^2 b^2$.

Step 8: Set $y_{k+1} = y_k - 1$. If $q_k < 0$, set $x_{k+1} = x_k + 1$ and

$q_{k+1} = q_k + 2b^2 x_{k+1} - 2a^2 y_{k+1} + a^2$. Otherwise, set $x_{k+1} = x_k$ and

$q_{k+1} = q_k - 2a^2 y_{k+1} + a^2$.

Step 9: Plot the four points: $(x_c, y_c) + (\pm x_{k+1}, \pm y_{k+1})$

Step 10: Increase k by 1. If $y_k = 0$ stop, otherwise go to Step 8.

The algorithm generates an ellipse with major and minor axes parallel to the coordinate axes. In order to draw an ellipse with axes not parallel to the coordinate axes, all we need to do is shift and rotate the ellipse thus constructed to get the desired ellipse. We shall discuss rotation and some more transformations in detail in the next unit.

E11) Implement the Midpoint Ellipse Drawing Algorithm.

E12) Write a C program to plot the sine function from 0 to 2π, taking symmetry into consideration.

E13) Using the midpoint method and symmetry into account develop an efficient method for scan converting the curve $y = \frac{x^3}{12}$ in $[-10, 10]$.

E14) Develop an efficient algorithm for scan converting the parabola $y^2 = 4ax$ in the interval $[0, c]$, where $c > 0$. Also implement your algorithm in C.

Geometric modelling requires curves which are flexible as well as smooth enough to represent a variety of different real life objects. For this we need to have curves that are easy to handle in design problems. In the next section we describe the parametric form of curves that are used in object modelling, namely the Bezier.

2.6 PARAMETRIC CURVES

As you know conic sections are implicitly defined as the curves obtained through the intersection of a plane with an infinite cone. In many applications, however, you require to use parametrically defined curves. In general, it has been observed that parametric curves are more suitable for geometric modelling and are extensively used in graphics. Spline curves are one such parametric curves that have been the first choice for many design applications. Recall that a curve given in the form $F(x, y) = 0$ is said to be in **implicit form**, since neither y nor x could be explicitly expressed in terms of other variable, in general. For example, the circle given in the form $x^2 + y^2 = r^2$ is an implicit curve. A curve is said to be in **parametric form** if it is expressed as a continuous function from a closed interval of real line to the three (or two) dimensional space. For example, the parametric form of an ellipse is given as follows:

$$r(\theta) = (a\cos\theta, b\sin\theta), \quad 0 \le \theta < 2\pi,$$

where a and b are the lengths of the semi-major and semi-minor axes. Such a curve can be easily plotted by incrementing the parameter θ and successively obtaining the new points on the curve. You only need to scan convert the points to the corresponding pixel values rounded to the nearest integers. Following simple function code will help you understand how such curves are plotted on the screen.

```
void param_ellipse( float a, float b)
{
    float x, pi = 22.0/7;
    glColor3f(0, 1, 1); //Set drawing colour
    glBegin(GL_POINTS);
      for(x = 0; x <=2*pi; x += 0.01)
         glVertex2f(a*cos(x), b*sin(x));
    glEnd();
}
```

Bézier curves are one of the most fascinating and extensively used curves in geometric modelling and graphics applications. These are simple polynomial curves with the additional properties that their shape can be controlled by a finite set of points called **control points**. These curves were introduced by two people independently, deCasteljau and Bézier, as solutions to certain geometric modelling and design problems in mechanical engineering. Bézier curves are defined as follows.

Let n be a positive integer and $\boldsymbol{P_0}, \boldsymbol{P_1}, \ldots, \boldsymbol{P_n}$ be $n+1$ points in the plane (or in the space). A **Bézier curve** P of degree n is defined as follows.

$$\boldsymbol{P}(t) = \sum_{i=0}^{n} \boldsymbol{P_i} B_i^n(t), \quad t \in [0,1],$$

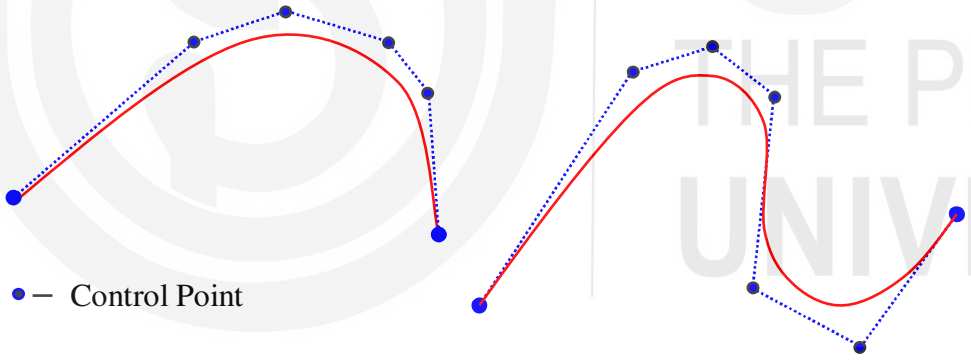
where $B_i^n : [0,t] \to \mathbb{R}$ is the i[th] **Bernstein function** define by

$$B_i^n(t) = \binom{n}{i} (1-t)^{n-i} t^i$$

For example, a linear Bézier curve is simply a straight line segment $(1-t)P_0 + t\ P_1$ and a quadratic Bézier curve is a parabola given by

$$(1-t)^2 P_0 + 2(1-t)\, t\, P_1 + t^2 P_2 .$$

Given below are two examples of Bézier curves.



Note that the i[th] Bernstein function $B_i^n(t)$ can be expressed as

$$B_i^n(t) = \frac{n(n-1)\ldots(n-i+1)}{1.2.\ldots\ldots i}(1-t)^{n-i} t^i .$$

This translates into the following C-code.

```
double B(int n, int i, double t)  //Bernstein function
{
    double prod = pow(1-t, n-i);
    for(int j=0; j < i; j++)
        prod = prod*t*(n-j)/(j+1);
    return prod;
}
```

Since the control points are two-dimensional, we need a data structure to store them. It is defined as below.

```
typedef struct point
{    int x;
     int y;
}Point;
```

A complete C-code for plotting a Bézier curve of degree n is given below in Listing 4. You need to input the control points using mouse clicks. The program opens a window with background colour as black and draws a Bezier curve for every set of $n+1$ control points selected using mouse clicks. The control points are drawn as red.

```
/* Create multiple Bezier curves by selecting control points
with mouse clicks*/
#include<stdio.h>
#include <math.h>
#include <GL/glut.h>

int n = 5;                          //Degree of the Bezier curve
int width=640, height = 480; //Window Size
int mouse_click = 0;

typedef struct point{ int x, y; }Point;

Point points[100];  //to store points obtained by mouse clicks

void myInit()
{
    glClearColor(0.2, 0.2, 0.2, 0);
    glColor3f(0, 1, 0);
    glPointSize(4.0);
    gluOrtho2D(0, 640, 0, 480);
}

double B(int n, int i, double t)  //Bernstein function
{
    double prod = pow(1-t, n-i);
    for(int j=0; j < i; j++)
        prod = prod*t*(n-j)/(j+1);
    return prod;
}

void drawBezier(Point *points, int n)
{
    double x, y, t;
    glColor3f(1, 1, 1); //Set drawing colour for curve
    glBegin(GL_POINTS);
        for(t = 0; t <= 1; t += 0.01)
        {   x = y = 0;
            for(int i = 0; i <= n; i++)
            {   x += points[i].x*B(n, i, t);
                y += points[i].y*B(n, i, t);
            }
            glVertex2f(x, height-y);
        }
    glEnd();
    glFlush();
    mouse_click = 0;
}

void myMouse(int button, int state, int x, int y)
{
    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {   points[mouse_click].x = x;
```

```
        points[mouse_click].y = y;
        glColor3f(1, 0, 0); //  colour of the control points
        glBegin(GL_POINTS);    //drawing the control point
            glVertex2i(x, height - y);
        glEnd();
        glFlush();
        mouse_click++;
    }
    if(mouse_click == n+1)
       drawBezier(points, n);
}

void myDisplay()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);

    glutInitWindowSize(width,height);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Bezier Curves");
    glutDisplayFunc(myDisplay);
    glutMouseFunc(myMouse);
    myInit();
    glutMainLoop();
    return 0;
}
```

Listing 4: A complete code for Bezier Curves

Rational extensions of Bézier curves have also been defined. These curves are being used extensively in computer aided design and geometric modelling. Here we shall not be discussing about these curves.

Spline curves are simple extensions of Bézier curves. They are piecewise polynomial curves with one or more polynomial pieces joined in such a manner that the resulting curve is continuous and may satisfy certain degree of smoothness. By this we mean that if we join two polynomial segments, at the joints the curve should be satisfying the requirements of continuity of derivatives upto certain order, depending upon the requirement of the problem. If higher order derivatives are matched, the composite curve becomes smoother. For example, we might be interested in designing a path of an object which has a continuous velocity along the path. In this case the path should be designed with a spline having continuous derivatives at all points, since the velocity is precisely the derivative vector at every point on the curve. This means, if we are having more than one polynomial segment, then at the joints one should ensure that the derivatives of the two meeting polynomial segments are matching. This is because at all other points the curve will of course be continuously derivable, being a polynomial curve.

We shall discuss spline curves in detail later in Unit 5.

You may now try the following exercises.

E15) Suggest a method to draw an object similar to the one given in E8) using four Bézier curves of suitable degree.

E16) Draw the letters S, P, R or U of English alphabet using multiple Bézier curves.

We now conclude this unit by giving a summary of what we have covered in it.

2.7 SUMMARY

In this unit, we have covered the following points.

1. We have seen how to draw basic output privatives using OpenGL functions such as points, lines or polylines.
2. We have described two important algorithms for line segment plotting– DDA algorithm and Bresenham's Line Drawing Algorithm.
3. It can be noted that the Bresenham's Line Drawing Algorithm is a very fast algorithm because its computations are all based on integer arithmatic.
4. Next, we have seen the Midpoint Circle Drawing Algorithm and the Midpoint Ellipse Drawing Algorithm.
5. We have also described how to draw curves given in parametric form, and also the Bezier curves.

2.8 SOLUTIONS/ANSWERS

E1) The `glBegin()… glEnd()` environment can be modified as follows:

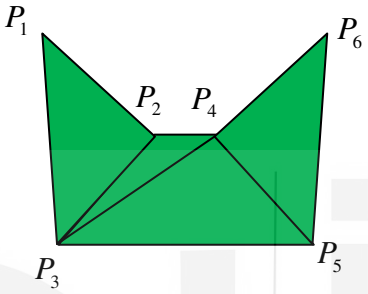
```
glBegin(GL_POINTS);
    for(i= - 200; i<=200; i++)
    {    glVertex2i(0,i);
         glVertex2i(i,0);
    }
glEnd();
```

E2) The required piece of code is given below.

```
glBegin(GL_LINE_LOOP);
  glVertex2i(0, 0);
  glVertex2i(200, 0);
  glVertex2i(200, 200);
  glVertex2i(0, 200);
  glVertex2i(-100, 300);
```

```
    glVertex2i(100, 300);
    glVertex2i(100, 100);
    glVertex2i(-100, 100);
glEnd();
glBegin(GL_LINES);
    glVertex2i(100, 100);
    glVertex2i(200, 0);
    glVertex2i(0, 0);
    glVertex2i(0, 200);
    glVertex2i(100, 300);
    glVertex2i(200, 200);
    glVertex2i(-100, 100);
    glVertex2i(-100, 300);
glEnd();
```

E3) We can triangulate the picture as follows:



Now let the coordinates of the vertices be

$P_1 = (0, 100), P_2 = (50,50), P_3 = (10, 0), P_4 = (100, 50),$
$P_5 = (140, 0), P_6 = (150, 100).$

Then, the required code is given below:

```
glBegin(GL_TRIANGLE_STRIP);
  glVertex2i(0,100);
  glVertex2i(50,50);
  glVertex2i(10,0);
  glVertex2i(100,50);
  glVertex2i(140,0);
  glVertex2i(150,100);
glEnd();
```

E4) Just replace the `display()` function given in Listing 3 by the following function.

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   float i, pi = 3.14, r = 100;
   for(i = 0; i < pi; i = i+pi/6)
      dda(r*cos(i), r*sin(i), -r*cos(i), -r*sin(i));
   glFlush();
}
```

Also include the library file `math.h` in the program in order to be able to use the `sine` and `cosine` functions.

E5) On present day systems with very high accuracy on floating point calculations all the three line segments are plotting almost the same line

segment. However, if you run the same program on an older system, you would find that the pixels generated by DDA algorithm are slightly away from the two line segments that have been generated using Bresenham algorithm and OpenGL function. The OpenGL function and Bresenham algorithm produce almost the same line segments.

E6) i) For the dotted line segment modify the code so that it plots pixels at specified positions. For instance the following code plots every 10th pixel position.

```
void dda_dotted(int xA, int yA, int xB, int yB)
{
    int dx, dy, k;
    float D, x=xA, y=yA;
    dx = xB-xA; dy = yB - yA;
    D = (abs(dx) >= abs(dy)) ? abs(dx) : abs(dy);
    glBegin(GL_LINES);
        glVertex2f(x, y);
        for ( k = 1; k <= D; k++)
        {   x = x + (float)dx/D;
            y = y + (float)dy/D;
            if(k%10 == 0)
                glVertex2i(floor(x), floor(y));
        }
    glEnd();
}
```

ii) For dashed line segments your code must skip a set of consecutive pixels at regular intervals. For instance, just replace the command `if(k%10 == 0)` by `if(k%10 > 3)` in the above code, and see the output.

E7) The required function `midpoint_circle()` is given below.

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

The function `plot_circle_points()` is defined as below.

```
void plot_circle_points(int xc, int yc, int x, int y)
{
    glBegin(GL_POINTS);
       glVertex2i(xc+x, yc+y);
       glVertex2i(xc+x, yc-y);
       glVertex2i(xc-x, yc+y);
```

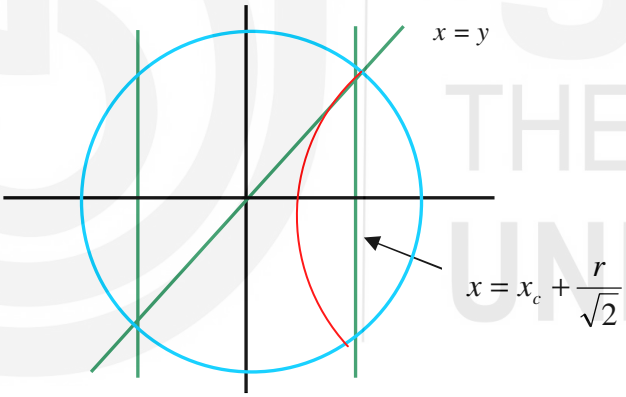
```
        glVertex2i(xc-x, yc-y);
        glVertex2i(xc+y, yc+x);
        glVertex2i(xc+y, yc-x);
        glVertex2i(xc-y, yc+x);
        glVertex2i(xc-y, yc-x);
    glEnd();
}
```

Then the function `display()` can be defined as follows.

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    int xc, yc, r;
    printf("Enter the centre of the circle:  ");
    scanf("%d%d", &xc, &yc);
    printf("Enter the radius:  ");
    scanf("%d", &r);
    midpoint_circle(xc, yc, r);
    glFlush();
}
```

E8) Let (x_c, y_c) be the centre and r the radius of the circle. We have to modify the last four pixels in the function plot_circle_points() as follows.

For the point (x, y) with $x = y$, we get $x = y = x_c + \frac{r}{\sqrt{2}}$. See the picture given below.



The points that are plotted on vertical arcs are simply those points which are symmetrically opposite to the points $(x_c \pm y,\ y_c \pm x)$, where symmetry is taken about the lines $x = x_c + \frac{r}{\sqrt{2}}$ and $x = x_c - \frac{r}{\sqrt{2}}$.

These points are computed to be $(x_c + \sqrt{2}r - y, y_c \pm x)$ and $(x_c - \sqrt{2}r + y, y_c \pm x)$. Accordingly, the last four pixels in the function `plot_circle_points()` given in the previous exercise are to be replaced by the following.

```
(xc + sqrt(2)*r - y, yc + x);
(xc - sqrt(2)*r + y, yc + x);
(xc + sqrt(2)*r - y, yc - x);
(xc - sqrt(2)*r + y, yc - x);
```

E9) Put the circle function `midpoint_circle()` in a for loop as follows:

```
for(r=rmin; r<=rmax; r=r+d)
    midpoint_circle(xc,yc,r);
```

where `rmin` and `rmax` are the radii of the innermost and the outermost circles, respectively, and `d` is the difference of their radii.

E10) Again this is a concentric circular arc drawing problem. Draw a circle first with smallest radius and then for the remaining three circles plot only the pixels $(x_c \pm x,\ y_c + y)$ until $y = x$. Implement dashed arcs as done in the solution to E5). Modify your circle drawing code `midpoint_circle()` accordingly.

E11) The required functions are as follows.

```
void plot_ellipse_points(int xc, int yc, int x, int y)
{
    glBegin(GL_POINTS);
        glVertex2i(xc+x, yc+y);
        glVertex2i(xc+x, yc-y);
        glVertex2i(xc-x, yc+y);
        glVertex2i(xc-x, yc-y);
    glEnd();
}
void midpoint_ellipse(int xc, int yc, int a, int b)
{
    int rx = 0, ry = 2*a*a*b, x = 0, y = b;
    int asq = a*a, bsq = b*b;
    int two_asq = 2*asq, two_bsq = 2*bsq;
    int p = bsq -asq*b + asq/4;
    int q;
    plot_ellipse_points(xc, yc, x, y);
    while(rx < ry)
    {   x++;
        rx = rx + two_bsq;
        if(p < 0)
            p = p + rx + bsq;
        else
        {   y--;
            ry = ry - two_asq;
            p = p + rx - ry + bsq;
        }
        plot_ellipse_points(xc, yc, x, y);
    }
    q = bsq*(x+0.5)*(x+0.5)+asq*(y -1)*(y-1)- asq*bsq;
    while(y > 0)
    {   y--;
        ry = ry - two_asq;
        if(q < 0)
        {   x++;
            rx = rx + two_bsq;
            q = q + asq + rx - ry;
        }
        else
            q = q + asq - ry;
        plot_ellipse_points(xc, yc, x, y);
    }
}
```

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    int xc, yc, a, b;
    printf("Enter the centre of the ellipse: ");
    scanf("%d%d", &xc, &yc);
    printf("Enter the semi-major and semi-minor axes: ");
    scanf("%d%d", &a, &b);
    midpoint_ellipse(xc, yc, a, b);
    glFlush();
}
```

E12) We shall compute the pixel positions only in the interval $\left[0, \frac{\pi}{2}\right]$. Pixel positions in interval $\left[\frac{\pi}{2}, 2\pi\right]$ can be found by exploiting the periodicity and symmetry property of the sine curve. For example, suppose you have computed a point (x, y) in the interval $[0, \pi/2]$, you can also identify three more points $(x, y), (\pi - x, y), (x + \pi, -y)$ and $(2\pi - x, -y)$. The following code implements the algorithm.

```
void plot_sine(int a)
{
    float x, y, pi = 3.1416;
    int c = 50;
    glColor3f(0, 0, 0);
    glBegin(GL_LINES);
        glVertex2i(-400, 0);
        glVertex2i(400, 0);
        glVertex2i(0, -400);
        glVertex2i(0, 400);
    glEnd();
    glColor3f(1, 0, 0);
    glBegin(GL_POINTS);
        for(x = 0; x <= pi/2; x = x+0.01)
        {   y = a*sin(x);
            glVertex2f(c*x, y);
            glVertex2f(c*(pi-x), y);
            glVertex2f(c*(pi+x), -y);
            glVertex2f(c*(2*pi-x), -y);
        }
    glEnd();
}
```

Note the importance of the variables a and c here, which serve as scaling factors in y and x directions, respectively. Small values of a and c would produce tiny output.

E13) Define the curve function $f(x, y) = \frac{1}{12}x^3 - y$. For this curve $\frac{dy}{dx} = \frac{1}{4}x^2$. The slope is greater than 1 for $x > 2$. Divide the first quadrant in two regions, Region 1, where slope ≤ 1 and Region 2, where slope > 1. In Region 1 initial point $(0, 0)$ is clearly a point on the curve. In this region

keep incrementing x values by 1 and determine the corresponding y value for the pixels to be plotted. Initialize the point $(0, 0)$ and send it to the frame buffer for plotting. Suppose the k-th pixel (x_k, y_k) has been identified and sent to the frame buffer. Determine the next pixel as follows:

Set $x_{k+1} = x_k + 1$. The curve is monotonically increasing. Therefore next y value can be either y_k or $y_k + 1$. Find out the value of the curve function at the midpoint of y_k and $y_k + 1$.

$$p_k = f\left(x_k + 1, y_k + \frac{1}{2}\right) = \frac{1}{12}(x_k + 1)^3 - y_k - \frac{1}{2}.$$

If $p_k < 0$, then (x_{k+1}, y_k) is closer to the curve than the pixel $(x_{k+1}, y_k + 1)$. Therefore, set $y_{k+1} = y_k$ if $p_k < 0$, else $y_{k+1} = y_k + 1$. The decision parameter can also be computed iteratively as follows:

$$p_{k+1} = \begin{cases} p_k + \frac{1}{12}\left(3x_k^2 + 9x_k + 7\right), \text{ if } p_k < 0 \\ p_k + \frac{1}{12}\left(3x_k^2 + 9x_k - 5\right), \text{ else} \end{cases}$$

On Region 2, y values will be more rapidly increasing and increment in x values will depend on the decision parameter's sign. Therefore, for each k, $y_{k+1} = y_k + 1$ define

$$q_k = f\left(x_k + \frac{1}{2}, y_k + 1\right) = \frac{1}{12}\left(x_k + \frac{1}{2}\right)^3 - y_k - 1.$$

If $q_k < 0$, then $x_{k+1} = x_k$, else $x_{k+1} = x_k + 1$. For iterative computation of q_k, we have

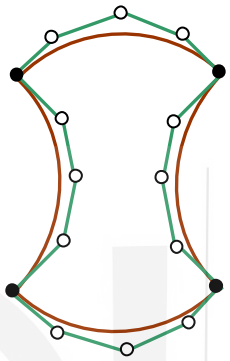
$$p_{k+1} = \begin{cases} q_k - 1, \text{ if } q_k < 0 \\ \frac{x_k^2}{4} + \frac{x_k}{2} - \frac{35}{48}, \text{else} \end{cases}.$$

E14) The main steps for plotting the parabola $y^2 = 4ax$ in the interval $[0, c]$ are given below.

Region I	**Region II**
$\frac{dy}{dx} > 1,\ 0 \le y < 2a$	$\frac{dy}{d} <,\ y > a, \le c$
$p_0 = 1 - 2a$ $(x_0, y_0) = (0, 0)$	$q_0 = f(x_\ell, y_\ell)$, where (x_ℓ, y_ℓ) is the last point plotted in Region I

$y_{k+1} = y_k + 1$ $x_{k+1} = \begin{cases} x_k, \text{if } p_k < 0 \\ x_k + 1, \text{ else} \end{cases}$	$x_{k+1} = x_k + 1$ $y_{k+1} = \begin{cases} y_k + 1, \text{ if } q_k < 0 \\ y_k, \text{ else} \end{cases}$
$p_{k+1} = \begin{cases} p_k + 2y_k + 3, \text{ if } p_k < 0 \\ p_k + 2y_k + 3 - 4a, \text{ else} \end{cases}$	$q_{k+1} = \begin{cases} q_k + 2y_{k+1} - 4a, \text{ if } q_k < 0 \\ q_k - 4a, \text{ else} \end{cases}$

E15) Employ four quartic Bézier curves with the endpoint of one as the initial point of the next Bézier curve and the endpoint of fourth as the initial point of the first Bézier curve as shown below.



Coordinates of control points can be approximated by drawing the picture on a graph paper.

E16) A complete code for plotting Bezier curves is given in Listing 4 of this unit. There in the code, control points for the Bézier curves are taken using mouse input. Plot Bézier curves by first identifying the control points of the curve and then storing them in an array. Always employ curves of same degree for plotting different parts of the alphabet letter. In the following code, character P is generated using only quadratic Bézier curves. Straight line segments are also generated using a quadratic Bézier curve by choosing the control points on a straight line.

```
/* Create the character P using multiple Bezier curves*/

#include <windows.h>
#include <math.h>
#include <gl/glut.h>

int n = 2;
int width=640,height = 480; //Window Size
int outer_last = 12; //Counter for curves in outer boundary
int inner_last=8; // Counter for curves in inner boundary

typedef struct point{    int x, y; }Point;

Point outer_points[] = { {0, 0}, {20, 0}, {40, 0}, {40,
30}, {40, 50}, {40, 100}, {80, 100}, {150, 100}, {150,
150}, {150, 200}, {80, 200}, {40, 200}, {0, 200}, {0, 100},
{0, 0} };
```

```
Point inner_points[] = { {50, 120}, {60, 120}, {70, 120},
{80, 120}, {80, 150}, {80, 180}, {70, 180}, {60, 180}, {50,
180}, {50, 150}, {50, 120} };

void myInit()
{
    glClearColor(0, 0, 0, 0);
    glColor3f(0, 1, 0);
    glPointSize(4.0);
    gluOrtho2D(-500, 500, -500, 500);
}
```

Insert the definitions of B() and drawBezier()

```
void Bezier()
{
    Point control[3];
    int i;
    for (i = 0; i <= outer_last; i += 2) //outer boundary
    {   control[0] = outer_points[i];
        control[1] = outer_points[i+1];
        control[2] = outer_points[i+2];
        drawBezier(control, 2);     // 3 points at a time
    }
    for (i = 0; i <= inner_last; i += 2)   //inner boundary
    {   control[0] = inner_points[i];
        control[1] = inner_points[i+1];
        control[2] = inner_points[i+2];
        drawBezier(control, 2);   // 3 points at a time
    }
}
//Draw character P on a mouse click
void myMouse(int button, int state, int x, int y)
{
    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        Bezier();
}

void myDisplay()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(width,height);
    glutInitWindowPosition(200,200);
    glutCreateWindow("Bezier curves");
    glutMouseFunc(myMouse);
    glutDisplayFunc(myDisplay);
    myInit();
    glutMainLoop();
    return 0;
}
```

Other characters R, S, U can be similarly plotted using appropriate coordinates of the control points.

UNIT 3

FILLING ALGORITHMS

Structure **Page No.**

3.1 INTRODUCTION

In Unit 2, you learnt how some basic geometric objects such as line segments, circles, ellipses and other curves are processed for plotting on a display unit. In most of the graphics packages you will find that polygons are also used as standard output primitives. Filled polygons with single solid color (or other alternatives for filling the interior) are provided as shape primitives.

In this unit, we shall discuss what we mean by filled area primitives and list two basic methods to fill polygonal or arbitrary shaped closed regions in Section 3.2. We consider here methods for solid fill of specified areas. One of these methods, called the Scanline Polygon Fill Algorithm, is discussed with full C-implementation in Section 3.3. The second type of methods are generally called seed fill algorithms, and we shall present two such algorithms in Section 3.4 with full implementation of Boundary Fill Algorithm. While creating or editing a document in a word processing software, you might have used letters, numbers and other characters in various font types and styles. We shall discuss in Section 3.3 how different fonts are generated using 2D drawing primitives.

Objectives

After reading this unit, you should be able to

- describe, implement and use the Scanline Polygon Fill Algorithm;
- apply the Odd-Even rule and the Nonzero Winding Number rule;
- differentiate between a 4-connected region and an 8-connected region;
- implement the Boundary Fill Algorithm for any type of closed region;
- generate a character or a font from the alphabet of any language.

3.2 FILLED-AREA PRIMITIVES

Filled-area primitives are one of the most important types of methods meant to fill a given closed region with a single colour/multiple colours or with some fill pattern. You will find later that almost all types of curves are approximated by polylines surfaces for the purposes of scan conversion. Therefore, polygon fill algorithms are extensively used in applications involved in 2D geometry processing. This section deals with some of the important and most common algorithms being used for defining and implementing filled area primitives.

Two basic approaches are followed in area filling on raster systems. In the first approach overlap intervals for scanlines that cross the area are determined per scanline (see Fig. 1 (a)). Second approach begins with an interior point and fills the area moving outward from this point until the boundary condition is reached (see Fig. 1(b)). An algorithm following the first approach is classified as **scan line algorithm** and that falling under second class is called a **seed fill algorithm**. Simple objects such as polygons, circles etc. are efficiently filled with scan line fill algorithms and more complex regions use the seed fill method. The scan line algorithm is mostly used in general graphics packages.

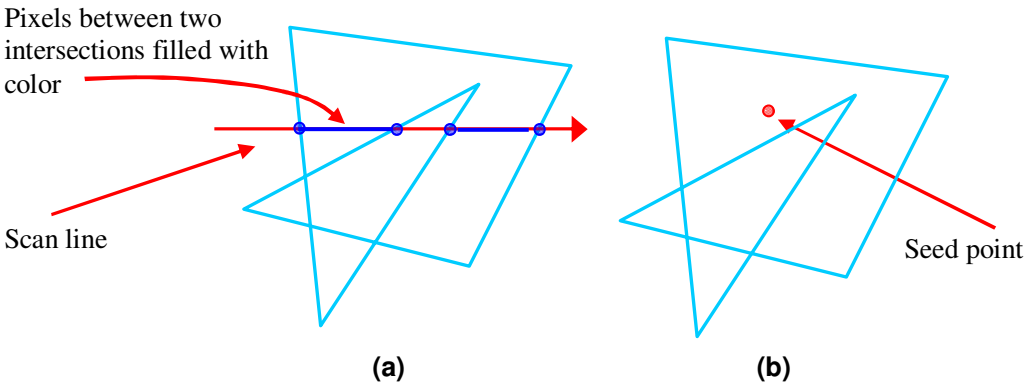


Fig. 1

Let us begin with scan line polygon fill algorithm. Notice that polygons can be as simple as a triangle and could be as complicated as the one shown in Fig. 2 below.

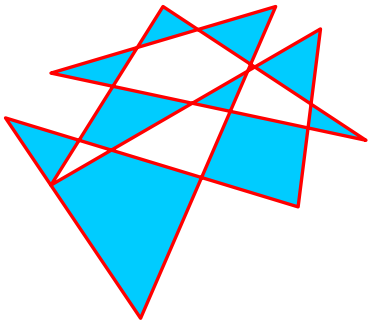


Fig. 2: A self intersecting polygon

We broadly keep the polygons in one of the three categories (i) convex (ii) concave (iii) self intersecting. Mathematically, a self intersecting polygon is concave. You will deal with such polygons in greater details for the purpose of area filling.

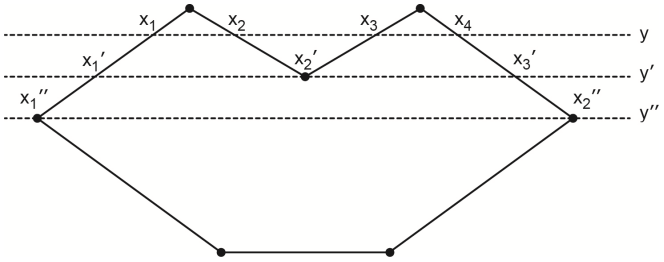
Before we go ahead with area filling algorithms, a word about pixel addressing and object geometry. You know that line segments are discretized into finite number of pixels and each pixel has its height and width depending upon the aspect ratio of the display unit. Therefore, if you plot a single pixel it will occupy some area on display unit. This contributes to the object geometry resulting in aberrations from the actual size. For example, if you have to plot a line segment starting from $(10, 10)$ to $(15, 10)$, then length of the segment is 5 whereas, in plotting this line segment, six pixel areas will be contributing to this segment, resulting in increased length of line segment by one unit. In this situation one can plot the line from $(10, 10)$ to $(14, 10)$. For closed regions we can plot pixels that are interior to the boundary.

Or one can also map the world coordinates (object coordinates) to the screen coordinates between pixels.

3.3 SCANLINE POLYGON FILL ALGORITHM

As the name suggests, this algorithm fills the region inside a polygon scanline by scanline. The basic idea is to compute the intersections of the edges of the polygon with each scanline, and sort them from the smallest to the highest value of x-coordinates. Next, the intersection points are paired and pixels are drawn between every pair of x-coordinates with a specified colour. A major point to be noted is that a scanline may not always intersect the edges in an even number of points, for instance when it passes through a vertex.

Such scanlines need to be processed carefully. For instance, look at the following polygon.



The scanline y intersects the polygon edges in an even number of points which can be paired as $(x_1, x_2), (x_3, x_4)$. So one can fill the scanline y from x_1 to x_2, and from x_3 to x_4 with the given colour. The scanline y' passes through the vertex x_2'. Note that the edges meeting x_2' are both lying above x_2'. In this case, we can duplicate x_2' to get the pairs (x_1', x_2') and (x_2', x_3'). The same approach can be adopted in the cases where a scanline passes through a vertex and both the edges incident to the vertex lie below it.

Now let us look at the scanline y'', which passes through the polygon vertices x_1'' and x_2''. The edges incident on x_1'' lie on the opposite side of the scanline y'', and the same is the case with x_2''. In such cases, each intersection point is counted once. This can be done easily if we **slightly shorten** the upper end of the **lower edge** at an intersection point.

Observe that for any scanline y we need to fill the area between pairs of x-coordinates where y intersects the polygon edges. So, we need to process only the nonhorizontal edges. Now let us look at the detailed procedure.

Procedure (Scanline Polynomial Fill Algorithm):

Let P be a polygon with n vertices given by $v_i = (x_i, y_i), i = 0, 1, \ldots, n-1$ in the counterclockwise orientation. For the sake of convenience, it will be assumed that vertices have non-negative integer coordinates.

Step 1: Input polygon vertices $v_i = (x_i, y_i), i = 0, 1, \ldots, n-1$ with $v_n = v_0$.
Further, define the edges $e_i = v_i v_{i+1}, i = 0, 1, \ldots, n-1$.

Step 2: Find out $y_{min} = \min_i \{y_i\}, y_{max} = \max_i \{y_i\}$.

Step 3: Create a sorted edge table to store the nonhorizontal edges. The table consists of as many rows as there are scanlines. Each edge has three members:

(a) upper y-coordinate,

(b) lower x-coordinate,

(c) inverse of the edge slope.

While moving along a particular direction, say clockwise, record the edges that cross a scanline y, put these edges in a sorted list indexed at the location y in the edge table, where the sorting is done on the lower x-coordinates of the edges. Moreover, if the successor or the predecessor of an edge e (along the moving direction) is lying above e, then the upper y-coordinate of e is reduced by 1.

Step 4: Set $y = y_{\min}$. While $y < y_{\max}$ do the following:

i) Create an active edge list (AEL) that consists of all those edges whose lower y-coordinate is less than or equal to y. The edges are stored in AEL in the ascending order of their lower x-coordinate values.

ii) Make pairs of the lower x-coordinate values of the edges in AEL, and fill each portion of the scanline enclosed by a pair of x-coordinate values with the specified colour. For a particular pair (x_i, x_j) we fill the pixels starting at x_i till the pixel preceding x_j.

iii) Update AEL as follows. For each edge e in AEL if the upper y-coordinate of e is smaller than or equal to y, remove e from AEL, otherwise increase its lower x-coordinate value by slope inverse.

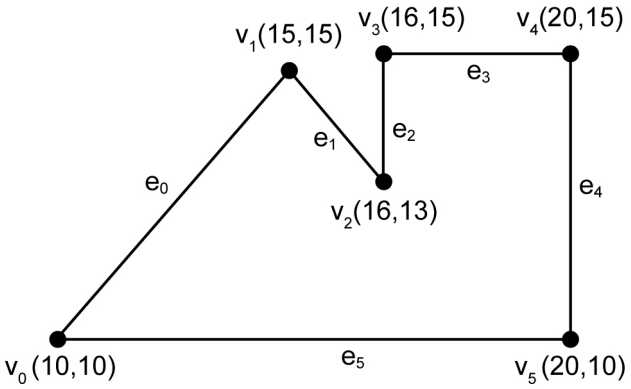
iv) Sort the edges in AEL in the ascending order of lower x-coordinate values.

v) Set $y = y + 1$.

Let us look at an example.

Example 1: Consider a polygon with vertices $v_0(10, 10), v_1 = (15, 15)$, $v_2 = (16, 13)$, $v_3 = (16,15)$, $v_4 = (20, 15)$, $v_5 = (20, 10)$, $v_6 = v_0$ and edges $e_j = v_j v_{j+1}$, $j = 0, 1, 2, 3, 4, 5$. Use the Scanline Polygon Fill Algorithm to fill this polygon.

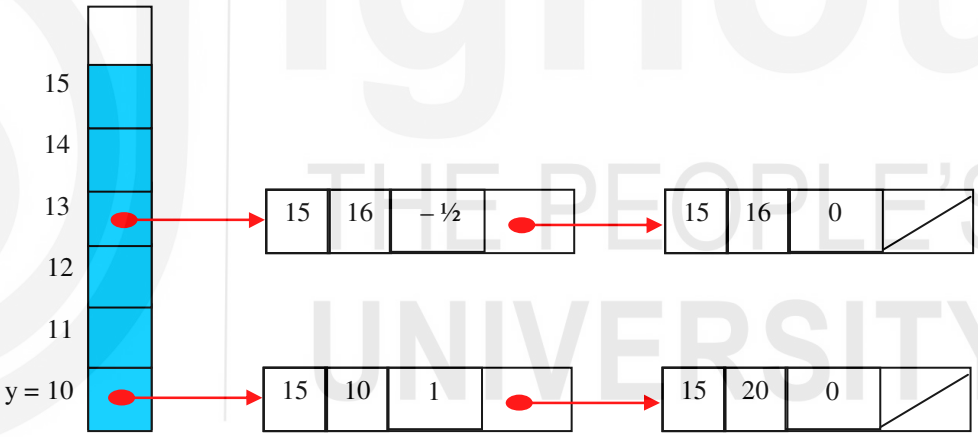
Solution: The polygon can be drawn as below.



Step 1: Store the vertices and edges in the order given.

Step 2: $y_{\min} = \min_i \{y_i\} = 10$ and $y_{\max} = \max_i \{y_i\} = 15$.

Step 3: Sorted edge table is created as follows. Start with $y = 10$ and continue till $y = 15$. Clearly for $y = 10$ there are only two edges that begin with this y-value namely, e_0 and e_4. Slope of e_0 is 1 and that of e_4 is ∞. As described in Step 3, the sorted edge table is given below.



Notice that the edges e_3 and e_5 do not appear in the edge table, being horizontal line segments.

Step 4: Active edge list and x-coordinate pairs for all scanlines

$y = 10,\ AEL = \{e_0, e_4\},\ x\text{-pairs} = (10, 20)$

$y = 11,\ AEL = \{e_0, e_4\},\ x\text{-pairs} = (11, 20)$

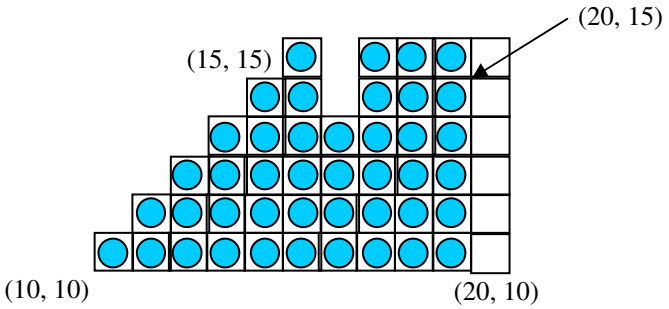
$y = 12,\ AEL = \{e_0, e_4\},\ x\text{-pairs} = (12, 20)$

$y = 13,\ AEL = \{e_0, e_1, e_2, e_4\},\ x\text{-pairs} = (13, 16)\ (16, 20)$

$y = 14,\ AEL = \{e_0, e_1, e_2, e_4\},\ x\text{-pairs} = (14, 15.5)\ (16, 20)$

$y = 15,\ AEL = \phi$

The resulting filled polygon is shown below.



Note that the polygon filling scheme will not fill the pixels on the horizontal edge e_4 joining (16,15) and (20,15). But the boundary of the polygon will display the edge. Similarly the vertex (15, 15) is plotted by virtue of it being a boundary point.

***

To implement the Scanline Polygon Fill Algorithm, we shall define two struct variables namely `point` and `Edge` as follows:

```
typedef struct point { int x, y; }Point;
typedef struct edge
{     int yupper;
      float x_intersect, slope_inverse;
      struct edge *next;
}Edge;
```

So, to store the vertices of the input polygon we can create an array of type `point`. Note that `Edge` contains four members–`yupper` to store the y-coordinate of the upper endpoint of the edge, `x_intersect` to store the x-coordinate of the lower endpoint of the edge, `slop_inverse` to store the inverse of the slope of the edge, and `next` which is a pointer to a variable of type `Edge`. To store the edgelist we create an array of pointers of type `Edge`. This can be done using the following command:

```
Edge* edges[GLUT_INNIT_WINDOW_HEIGHT];
```

Here `GLUT_INNIT_WINDOW_HEIGHT` is an OpenGL constant indicating the height of the window initialised in terms of the number of scanlines. The full C-program that implements the Scanline Polygon Fill Algorithm is given below, in Listing 1.

```
// Scanline Polygon Fill Algorithm
#include<stdio.h>
#include<GL/glut.h>

//insert the definitions of Point and Edge.

void insert_edge(Edge *list, Edge *e)
{
```

```
    Edge *prev = list, *curr;
    for(curr = list->next; curr != NULL; curr = curr->next)
    {   if(e->x_intersect < curr->x_intersect)
            break;
        prev = curr;
    }
    e->next = curr;
    prev->next = e;
}

int y_next(int i, int n, Point *points)
{
    int j = (i+1)%n;          // j becomes 0 if it crosses n-1
    while(points[j].y == points[i].y)
        j = (j+1)%n;
    return (points[j].y);
}

void create_edgelist(int n, Point *points, Edge *edges[])
{
    Edge *e;
    Point u = points[n-1], v;
    int i, j, yprev = points[n-2].y, ynext;
    for(i = 0; i < n; i++)
    {   v = points[i];
        if(u.y != v.y)
        {   e = (Edge *)malloc(sizeof(Edge));
            e->slope_inverse = (float)(u.x-v.x)/(u.y - v.y);
            if(u.y < v.y)
            {   e->x_intersect = u.x;
                e->yupper = v.y;
                ynext = y_next(i, n, points);
                if(v.y < ynext)
                    e->yupper--; //shorten edge upper end by 1
                insert_edge(edges[u.y], e);
            }
            else
            {   e->x_intersect = v.x;
                e->yupper = u.y;
                if(u.y < yprev)
                    e->yupper--;  //shorten edge upper end by 1
                insert_edge(edges[v.y], e);
            }
        }
        yprev = u.y;
        u = v;
    }
}

void create_active_list(int line, Edge *active, Edge *edges[])
{
    Edge *prev, *current;
    for(prev = edges[line]->next; prev != NULL; prev = current)
    {   current = prev->next;
        insert_edge(active, prev);
    }
}

void fillscan(int line, Edge *active)
{
    Edge *curr, *prev = active->next;
    int x;
    glColor3f(0, 1.0, 0.0);
    while(prev != NULL)
    {   curr = prev->next;      //prev->next is not NULL. (Why?)
        glBegin(GL_LINES);
```

```
        for(x = prev->x_intersect; x < curr->x_intersect; x++)
            glVertex2i(x, line);
        glEnd();
        prev = curr->next;
    }
}

void update_active_list(int line, Edge *active)
{
    Edge *prev = active, *curr = active->next;
    while(curr != NULL)
        if(line >= curr->yupper)
        {   prev->next = curr->next;
            free(curr);
            curr = prev->next;
        }
        else
        {   curr->x_intersect += curr->slope_inverse;
            prev = curr;
            curr = curr->next;
        }
}

void sort_active_list(Edge *active)
{
    Edge *curr, *prev = active->next;
    active->next = NULL;
    for(  ; prev != NULL; prev = curr)
    {   curr = prev->next;
        insert_edge(active, prev);
    }
}

void scanline_polygon_fill(int n, Point *points)
{
    Edge *edges[GLUT_INIT_WINDOW_HEIGHT], *active;
    int i, line, ymax = points[0].y, ymin = points[0].y;
    for(line = 0; line < GLUT_INIT_WINDOW_HEIGHT; line++)
    {   edges[line] = (Edge *)malloc(sizeof(Edge));
        edges[line]->next = NULL;
    }
    create_edgelist(n, points, edges);
    active = (Edge *) malloc(sizeof(Edge));
    active->next = NULL;

    for(i = 1; i < n; i++)
    {   if(points[i].y > ymax)
            ymax = points[i].y;
        if(points[i].y < ymin)
            ymin = points[i].y;
    }
    for(line = ymin; line < ymax; line++)
    {   create_active_list(line, active, edges);
        if(active->next)
        {   fillscan(line, active);
            update_active_list(line, active);
            sort_active_list(active);
        }
    }
    glFlush();
}
// function to initialize
void myInit (void)
{
    gluOrtho2D(0, 1000, 0, 1000);
    glClearColor(0.5, 0.5, 0.5, 0.0);
```

```
    glColor3f(1, 0, 0);
    glPointSize(2);
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    int n, i;
    Point points[50];
    printf("How many polygon vertices? ");
    scanf("%d", &n);
    printf("Enter vertices line by line.\n");
    for(i = 0; i < n; i++)
        scanf("%d%d", &points[i].x, &points[i].y);
    glBegin(GL_LINE_LOOP);
        for(i = 0; i < n; i++)
            glVertex2i(points[i].x, points[i].y);
    glEnd();
    glFlush();

    scanline_polygon_fill(n, points);

}

int main (int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

    // giving window size in X- and Y- direction
    glutInitWindowSize(1000, 1000);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Scanline Polygon Fill Algorithm");

    myInit();
    glutDisplayFunc(display);
    glutMainLoop();
}
```

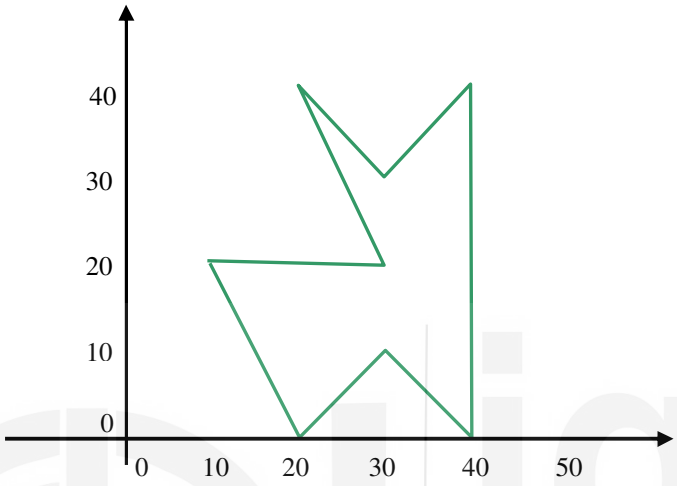
Listing 1: Scanline Polygon Fill Algorithm Implementation

Let us look at the function `scanline polygon_fill ( )`.

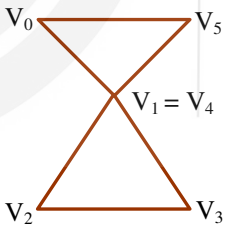
Note that the first `for` loop prepares the array edges to reserve blocks of memory in order to store the edglist for each scanline. Next, the function call `create_edgelist()` actually creates an edgelist. Then the pointer `active` is initialised. The next `for` loop computes the minimum and maximum values of the y-coordinates of the polygon vertices. The last `for` loop fills the polygon line by line starting from `ymin` to `ymax-1`. For each scanline in the range, an active edglist is created using the function call `create_active_edglist()`. If the active list is nonempty, the current scanline is filled through the function `fillscan()`, the active edgelist is updated through `update_active_list()` and then sorted using `sort_active_list()`.

You may now try the following exercises.

E1) For the following polygon, prepare an array of sorted edge lists and then make the active edge list for scanlines $y = 5,\ 20,\ 30,\ 35$. Coordinates of the vertices are $v_0 = (10,\ 20),\ v_1 = (20,\ 0),\ v_2 = (30,\ 10),\ v_3 = (40,\ 0),$ $v_4 = (40,\ 40),\ v_5 = (30,\ 30),\ v_6 = (20,\ 40),\ v_7 = (30,\ 20).$ Also find the pairs of x-coordinate values for $y = 5.$



E2) For the polygon shown below how many times will the vertex V_1 appear in the set of intersection points for the scan line passing through that point? How many times will you count it when you form the pairs of intersection points? Justify your answer.



In the next section, we shall look at a different approach to area filling.

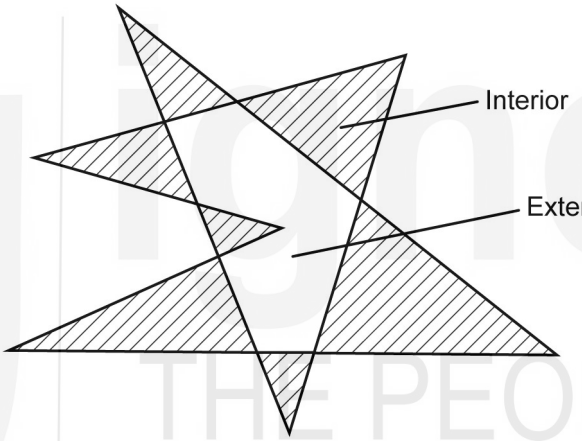
3.4 BOUNDARY FILL ALGORITHM

As you saw the Scan Line Polygon Fill Algorithm works for polygons only. The seed fill algorithms do not require any such constraints. You only need to know an interior point of the closed boundary object to fill it. This interior point is called a **seed point**. However, determination of interior point for complex polygons such as the one shown in Fig. 2 itself is a challenging task. The following methods are used to determine an interior point of the area to be filled.

1. Odd-Even rule,
2. Winding Number Rule.

Once an interior point of the object is determined, the **boundary-fill algorithm** or, **flood-fill algorithm** may be applied to fill the given area.

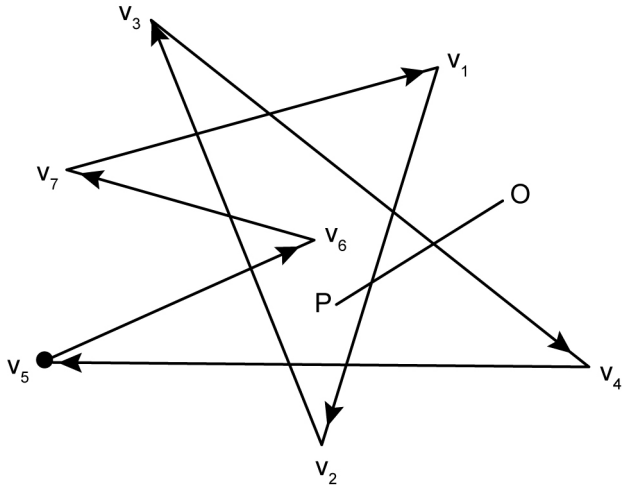
Odd-Even Rule: Consider any polygon A and a point P in the plane of the polygon. To check whether P lies inside or outside A, we draw a line from P to a reference point O chosen far enough from A. Note that we choose O in such a way that the line OP does not pass through any vertex of A. Then P lies inside A if OP cuts an odd number of the edges of A, otherwise P lies outside A. Applying the Odd-Even rule to the following polygon you can easily see that its interior region.



Winding Number Rule: Again consider a polygon A and a point P. In this rule, the edges of A are first oriented in a particular direction (clockwise or anti-clockwise). Then a reference point O is chosen far enough from A in such a way that the line OP does not pass through any vertices of A. Denote the **Winding Number** of P by $\omega(P)$. To compute $\omega(P)$, we move along the line OP, starting at P, and count how many edges of A cross OP from left to right and how many from right to left. Then $\omega(P)$ is essentially the difference of the two numbers. That is,

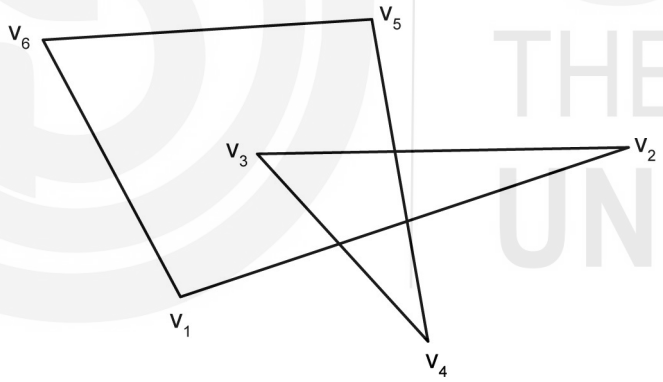
$$\omega(P) = \text{edges crossing } OP \text{ from left}$$
$$- \text{ edges crossing } OP \text{ from right}$$

If $\omega(P) \neq 0$, then P lies inside A, otherwise P lies outside A. For instance, the above polygon edges can be oriented as follows.

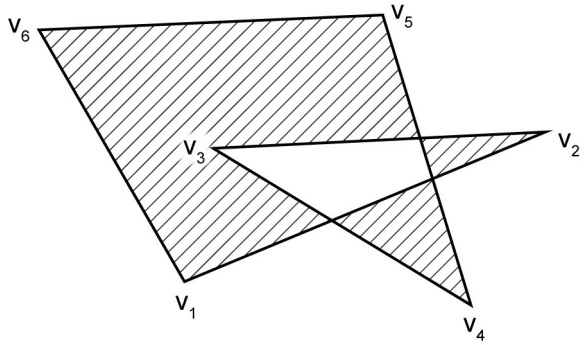


Consider a point P and the reference point O as shown above. As we move from P to O, the edge v_1v_2 crosses OP from left, and the edge v_3v_4 also from left. Thus, $\omega(P) = 2 \neq 0$. Therefore, P lies inside the polygon. Note that both the rules do not give the same interior and exterior regions. Let us look at one more example.

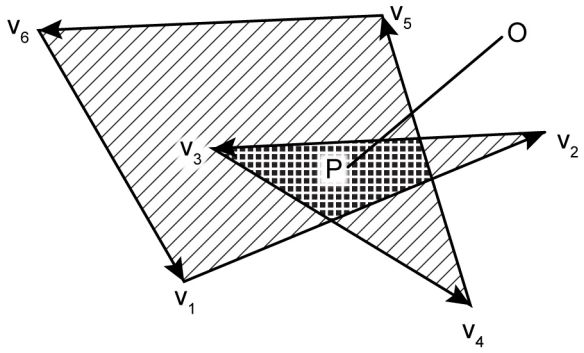
Example 2: Apply the Odd-Even Rule and the Winding Number Rule to find the interior and exterior regions of the following polygon. Do both the rules produce the same exterior regions?



Solution: Using the Odd-Even Rule, it is easy to observe that the shaded region as shown below is the interior region, and the rest is the exterior.



In order the apply the Winding Number Rule, we orient the edges of the polygon as follows.



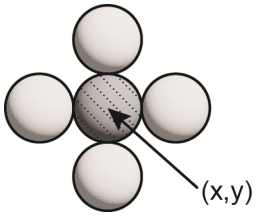
You can now see that all the interior regions that were obtained by the Odd-Even Rule are also the interior regions by the Nonzero Winding Number Rule. Now consider a point P in the region shaded dark above, and a reference point O. When we move from P to O, the edges v_2v_3 and v_4v_5 cross the line PO from right. Therefore, $\omega(P) = 2 \neq 0$. This means P is also an interior point, and hence the whole shaded region (dark and light) is an interior region.

Thus the exterior regions produced by the two rules are not the same.

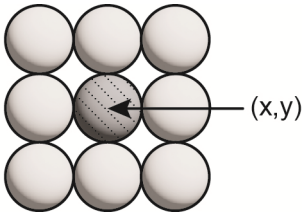
***

After understanding how to select an interior point of a closed region we turn to the procedure of Seed Fill Algorithm. The algorithm starts with an interior point as the seed point and recursively checks whether the neighbouring pixels have the same colour as the boundary colour or not. If yes, the call terminates, otherwise the neighbouring pixel is coloured with given fill colour and the next recursive call is made from the current pixel.

Given a pixel position $(x,\ y)$, the question arises: What are the neighbours of $(x,\ y)$? There are two ways to determine this. (See picture below.)



4-connected



8-connected

If we consider only the top, down, left or right pixels as the neighbours of $(x,\ y)$, then the region defined this way is called a **4-connected region**. Thus, in a 4-connected region the neighbouring pixels of $(x,\ y)$ are $(x,\ y+1)$, $(x,\ y-1)$, $(x-1,\ y)$, $(x+1,\ y)$. On the other hand, if we include the four diagonal pixels as well in addition to the top, down, left and right, we get a region that is called **8-connected**. Depending on the type of region, there are

two versions of Boundary Fill Algorithm (4-connected and 8-connected). Here we have implemented the 4-connected version.

```
// Boundary Fill Algorithm (4 connected regions)
#include<stdio.h>
#include<GL/glut.h>
#define width 1000
#define height 1000
typedef struct point { int x, y; } Point;
int equal(float *colour1, float *colour2)
{
    for(int i = 0; i < 3; i++)
        if(colour1[i] != colour2[i])
            return 0;
    return 1;
}
void draw_point(int x, int y, float *colour)
{
    glColor3fv(colour);
    glBegin(GL_POINTS);
       glVertex2i(x, y);
    glEnd();
    glFlush();
}

void boundary_fill4(int x, int y, float* fill, float* boundary)
{
    float colour[3];
    glReadPixels(x, y, 1, 1, GL_RGB, GL_FLOAT, colour);
    if(!equal(colour, fill) && !equal(colour, boundary))
    {   draw_point(x, y, fill);
        boundary_fill4(x+1, y, fill, boundary);
        boundary_fill4(x-1, y, fill, boundary);
        boundary_fill4(x, y+1, fill, boundary);
        boundary_fill4(x, y-1, fill, boundary);
    }
}

void mouse(int button, int state, int x, int y)
{
    y = height-y;
    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {   float boundary[] = {1, 0, 0};
        float fill[] = {0, 0, 1};
        boundary_fill4(x, y, fill, boundary);
    }
}

void myInit (void)
{
    gluOrtho2D(0, 1000, 0, 1000);
    glClearColor(0.5, 0.5, 0.5, 0.0);
    glColor3f(1, 0, 0);
    glPointSize(2);
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    int n, i;
    Point points[50];
    printf("How many polygon vertices? ");
    scanf("%d", &n);
    printf("Enter (nonnegative) coordinates of vertices:\n");
```

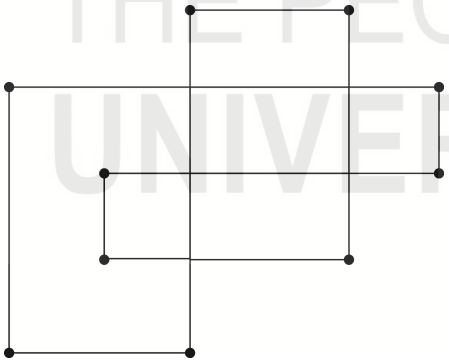
The function glReadPixels ( ) reads the colour of the pixel (x, y) and stores it in the array colour.

```
    for(i = 0; i < n; i++)
        scanf("%d%d", &points[i].x, &points[i].y);
    glBegin(GL_LINE_LOOP);
        for(i = 0; i < n; i++)
            glVertex2i(points[i].x, points[i].y);
    glEnd();
    glFlush();

}

int main (int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

    glutInitWindowSize(width, height);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Boundary Fill Algorithm (4-connected)");

    myInit();
    glutDisplayFunc(display);
    glutMouseFunc(mouse);
    glutMainLoop();
}
```

Listing 2: Boundary Fill Algorithm (4-connected)

You may now try the following exercises.

E3) Shade the interior of the following polygon using Odd-Even Rule and Winding Number Rule.



E4) Devise an algorithm for determining the interior regions for any input set of vertices using the nonzero winding number rule and dot-product calculations to identify the direction of edge-crossings.

E5) Explain how an ellipse displayed with the midpoint method could be properly filled with a boundary fill algorithm.

E6) Develop and implement the Flood Fill Algorithm.

Next, we shall discuss the character generation techniques.

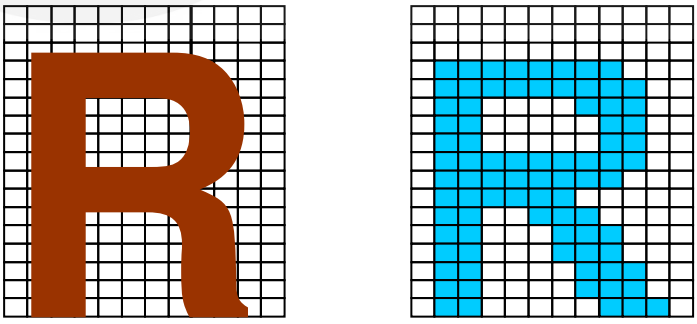
3.5 CHARACTER GENERATION

You know that graphics displays often contain components which are text based such as graph labels, annotations, descriptions on data flow diagrams, fill patterns containing text strings and information for simulation and visual applications. Routines for generating characters are available in most of the graphics systems and graphics application softwares. Here you will learn some simple techniques used in creating texts.

There are two main approaches followed in character or font generation (i) Bitmap font method (ii) Outline font method. In the bitmap method, a matrix of bits (0 or 1) is formed which approximates the shape of the font. Every entry in the matrix corresponds to a pixel and those pixels are plotted for which the matrix entry is 1.

In the outline method the font boundary shape is approximated by spline curves. You have already done some practice on character generation using multiple Bezier curve segments (see E15) of Unit 2).

In order to understand how fonts are converted to bitmap, just think of a letter drawn on a graph paper. Measure its span in terms of number of maximum x and y units it covers. Then map the letter grid to a frame buffer position. For example, in the picture given below, the letter R spans in an area corresponding to the matrix of order 14×10. On the right is shown the corresponding frame buffer positions with 1 indicating the pixel to be drawn, and 0 indicating the pixel not to be drawn.



(a) Letter R drawn on a graph paper **(b) Corresponding frame buffer positions**

In the outline method we simply identify a few shape control points and draw the corresponding Bezier curves for different segments of the outline.

Some predefined character sets are available in the OpenGL Utility Toolkit (GLUT). So you need not create fonts as bitmap shapes unless you want to display a font that is not available in GLUT. For example, you may want to

generate a font of your own mother tongue. The GLUT library contains routines for displaying both bitmapped and outline fonts. Bitmapped GLUT fonts are rendered using the **glutBitmapCharacter**( ) function, as shown below

```
glutBitmapCharacter (font, character);
```

where parameter **font** is assigned a symbolic GLUT constant identifying a particular set of type faces, and parameter **character** is assigned either the ASCII code or the specific character we wish to display. Thus, to display the upper-case letter "A", we can either use the ASCII value 65 or the designation **'A'**. Similarly, a code 97 corresponds to the lower-case letter **'a'** and so on. You can also use fixed-width fonts and proportionally spaced fonts. For example, to select a fixed-width font of 8×13, you need to assign either **GLUT_BITMAP_8_BY_13** or **GLUT_BITMAP_9_BY_15.** For a proportionally spaced font of Times Roman type with size 10 point, you need to select **GLUT_BITMAP_TIMES_ROMAN_10** to parameter **font**. The choice **GLUT_BITMAP_TIMES_ROMAN_12** is also available. For example, to draw the bitmap font Times Roman of size 10 at the raster position (20, 20), you need to use the following statements.

```
glRasterPos2i(20, 20);
glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_10,'B');
```

An outline character is displayed with the following function call.

```
glutStrokeCharacter(font, character);
```

You can assign parameter 'font' either the value GLUT_STROKE_ROMAN, which displays a proportionally spaced font, or the value GLUT_STROKE_MONO_ROMAN, which displays a font with constant spacing. You can specify and control the size and position of these characters by using certain geometric transformations that you will study in the next block. You may refer to OpenGL reference manual for more options.

The following code demonstrates displaying the string 'OpenGL' using the function glutStrokeCharacter( ).

```
char text[] = {'O', 'p', 'e', 'n','G', 'L'};
for(int k = 0; k < 6 ; k++)
      glutStrokeCharacter(GLUT_STROKE_ROMAN, text[k]);
```

You may now test your understanding by trying out the following exercises.

E7) Use the outline method to plot the following font boundaries (style should remain the same). Implement your method in C language using OpenGL.

E8) Design a bitmap for the English vowels A, E, I, O, U for two different sizes and then implement the bitmaps to plot these vowels on the display. Keep in mind that baseline of all the bitmaps should remain the same when plotting, just as it is printed in English language.

We now end this unit by giving a summary of what we have covered in it.

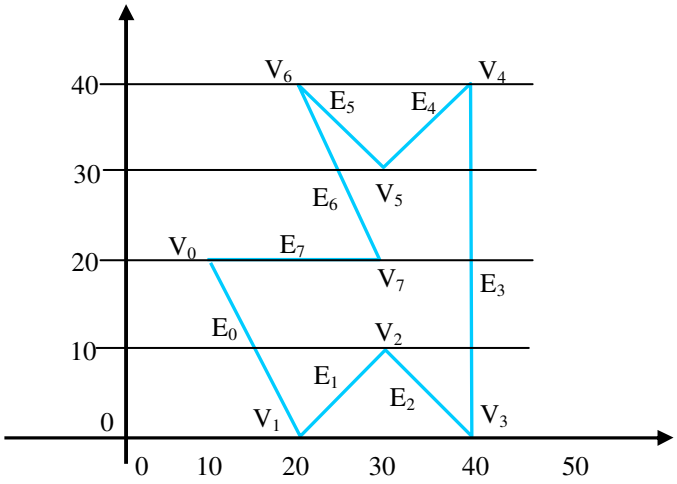
3.6 SUMMARY

In this unit, we have covered the following points.

1. We have described the Scanline Polygon Fill Algorithm and two seed fill algorithms for **filled-area primitives**.
2. **Boundary fill algorithm** is suitable when the boundary has single colour while **flood fill algorithm** is more suitable for filling regions which are defined with boundary having more than one color, for example, a map of a country surrounded with other countries.
3. We have also discussed two basic approaches used in character generation namely – **Bitmap method** and **outline method**.

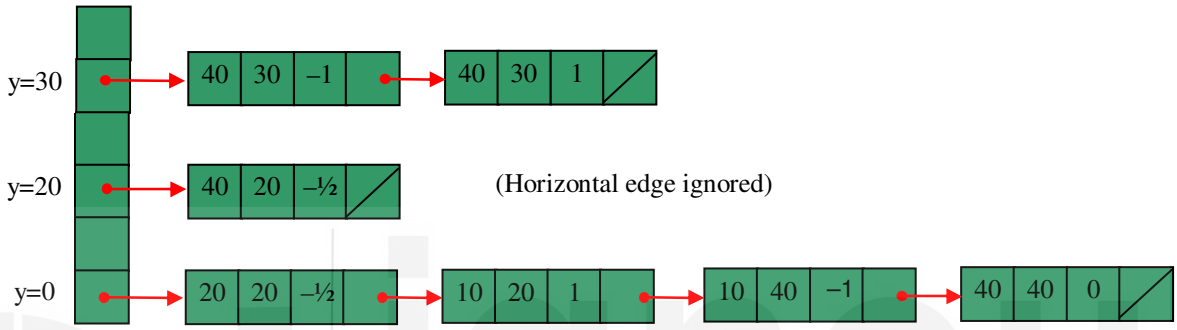
3.7 SOLUTIONS/ANSWERS

E1) First label the vertices and edges as shown below.



Let $e_i = v_i v_{i+1}$, for $i = 0, 1, \ldots, 6$. If m_i is the slope of edge e_i, then we find that $m_0 = -2$, $m_1 = 2$, $m_2 = -1$, $m_3 = \infty$, $m_4 = 1$, $m_5 = -1$, $m_6 = -2$, $m_7 = 0$.

Start with $y = 0$ and continue till $y = 40$. For $y = 0$ there are four edges, namely e_0, e_1, e_2 and e_3. Then for $y = 20$, we get the edge e_6 only, and for $y = 30$ the edges are e_5, e_4. The sorted edge table is created as follows:



Active edge lists (AEL) for scan lines $y = 5, 20, 30, 35$ are as follows.

$y = 5,\ AEL = \{e_0, e_1, e_2, e_3\}$

$y = 20,\ AEL = \{e_6, e_3\}$

$y = 30,\ AEL = \{e_6, e_5, e_4, e_3\}$

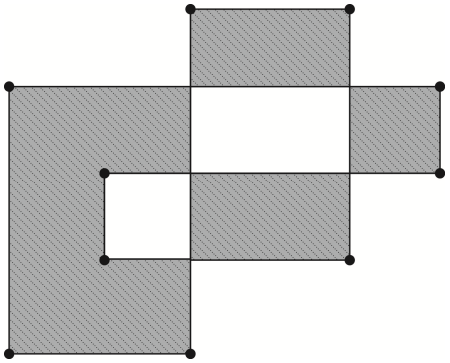
$y = 35,\ AEL = \{e_6, e_5, e_4, e_3\}$

We start with $y = 0$ and iteratively compute the x-coordinate values for each edge till we reach the scanline $y = 5$, as shown in the following table.

Scanline	x-coordinate values			
	e_0	e_1	e_2	e_3
$y = 0$	20	20	40	40
$y = 1$	19.5	21	39	40
$y = 2$	19	22	38	40
$y = 3$	18.5	23	37	40
$y = 4$	18	24	36	40
$y = 5$	17.5	25	35	40

E2) Note that as we move from v_0v_1 to v_1v_2 the lower y-values of these edges are monotonically decreasing. Therefore, the upper endpoint of v_1v_2 is shortened. As a result there is only one intersection at v_1. Likewise, when we move from v_3v_4 to v_4v_5 there is one intersection at v_4. Since $v_1 = v_4$, the intersection is counted once.

E3) Both the rules produce the following shaded region as the interior.



E4) Let P be a given polygon with n vertices given by $V_i = (x_i, y_i)$, $i = 0, 1, \ldots, n-1$ in the counter clockwise orientation.

Find out $x_{\min} = \min_i \{x_i\}$, $x_{\max} = \max_i \{x_i\}$ and $y_{\min} = \min_i \{y_i\}$, $y_{\max} = \max_i \{y_i\}$

Let $Q = (x, y)$ be the point to be tested. Initialize the winding number $\omega_n = 0$. If any of the following conditions is satisfied Q cannot be inside the polygon.

(i) $x < x_{\min}$ (ii) $x > x_{\max}$ (iii) $y < y_{\min}$ (ii) $y > y_{\max}$

In case none of the conditions is satisfied, you are required to have a ray starting from Q and extending to a distant point from the polygon and then construct a vector in the direction of this ray. For simplifying the computations, you can choose a vector **u** from Q in the direction $(1, 0)$. Basically, this vector is on the scan line on which point Q lies. If no vertex is on this scan line, your choice of vector is correct. Else choose a vector with a slightly modified direction so that no vertex falls on the ray from Q in the chosen direction. Create an active edge list (AEL) for this ray. Each edge whose one of the x-extents is greater than x is included in the AEL. Next make a vector **v** perpendicular to **u**. If $\mathbf{u} = (u_x, u_y)$, **v** can be defined as $\mathbf{v} = (-u_y, u_x)$. For each edge $V_i V_{i+1}$ in the AEL, take the dot product of the vector $V_{i+1} - V_i$ with **v**. Let $d_i = (V_{i+1} - V_i).\mathbf{v}$ (dot product). If $d_i > 0$, the edge $V_i V_{i+1}$ crosses the ray from right to left. Update the $\omega_n = \omega_n + 1$, else the edge crosses the ray from left to right and $\omega_n = \omega_n - 1$. When all the edges are processed and $\omega_n \neq 0$, the point Q is inside the polygon else it is outside.

E5) Modify the function plot_ellipse_points() given in E12 of Unit 2 as follows.

```
void plot_ellipse_points(int xc, int yc, int x, int y)
{
   glBegin(GL_POINTS);
   for(int i = xc - x ; i < xc + x ; i++)
   {   glVertex2i(i, yc + y);
       glVertex2i(i, yc - y);
   }
   glEnd();
}
```

E6) A code for flood fill algorithm with 4-connected cells is given below.

```
void flood_fill4(int x, int y, float* fill, float* old)
{
     float colour[3];
     glReadPixels(x, y, 1, 1, GL_RGB, GL_FLOAT, colour);
     if(equal(colour, old))
      { draw_point(x, y, fill);
        boundary_fill4(x+1, y, fill, old);
        boundary_fill4(x-1, y, fill, old);
        boundary_fill4(x, y+1, fill, old);
        boundary_fill4(x, y-1, fill, old);
      }
}
```

E7) Here letter G is produced using linear segments for outline method. You may also use Bezier curves of higher degree to approximate the outline of the character. The outline will be drawn using the code given here. For filling the interior, you need to employ one of the fill area methods. Vertices of the line segments are given as follows.

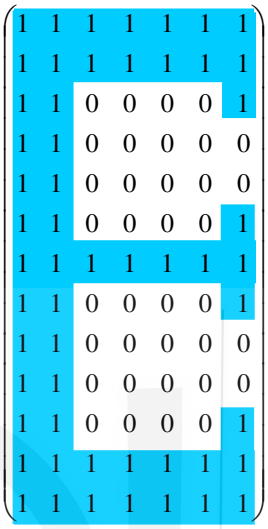
```
GLint vt[][2]={{315,245}, { 315,195}, {316,190},{333,183},
{ 251,183}, {263,186}, { 270, 190},{272,195}, {272,225}, {
253, 238}, {242,236}, {235,229}, {231,219}, {227,198}, {
227,169}, { 228,152}, {233,133}, {242,124}, {248,120}, {
253,120}, {269,119}, { 273,121}, { 280,126}, {292,134}, {
299,141}, {305,149}, {312,162}, {312,111}, {307,110}, {307,
120}, {304, 130}, {301, 133}, {298, 131}, { 289, 125},
{279, 120}, {266, 113}, {251, 109}, {239, 110},{229,110},
{219,114}, {208,120}, {196,129}, {190,138}, {185,148},
{181, 158}, {180,172},{180,184}, {180,196}, {184, 203},
{189, 214}, {198, 226}, {211, 235}, {220,242}, {230,245},
{245,247}, {260, 247}, {271, 241}, {284, 236}, {293, 231},
{299, 231}, {306, 235}, {315,245}};
```

The code uses simple line loop to produce the character.

```
void lineG(void)
{glBegin(GL_LINE_LOOP);
for(int i = 0;i<62;i++) {
      vt[i][1]=480-vt[i][1];
      glVertex2iv(vt[i]);}
      glEnd();
}
```

Call the function lineG in your display function. Similar is the code for letter '**t**'.

E8) Here we discuss the bitmap for letter 'E'. You need to place the character 'E' on a square grid of sufficiently large size. If the character's area is overlapping more than 50% of a cell of the square grid, assign that cell a bit 1, otherwise assign the cell bit 0. This way you will have a rectangular grid having cells either assigned 0 or 1. Map this onto a rectangular matrix and plot the character 'E' using the bitmap method as shown below.



Other characters can also be modelled using the same technique. Repeat your process for a larger size font also.

Notes