SYLLABUS

BASICS OF COMPUTER GRAPHICS: -
Introduction, What is computer Graphics?, Area of Computer Graphics, Design and Drawing, Animation Multimedia applications, Simulation, How are pictures actually stored and displayed, Difficulties for displaying pictures.

GRAPHIC DEVICES
Cathode Ray Tube, Quality of Phosphors, CRTs for Color Display, Beam Penetration CRT, The Shadow - Mask CRT, Direct View Storage Tube, Tablets, The light Pen, Three Dimensional Devices

C Graphics Basics
Graphics programming, initializing the graphics, C Graphical functions, simple programs

SIMPLE LINE DRAWING METHODS
Point Plotting Techniques, Qualities of good line drawing algorithms, The Digital Differential Analyzer (DDA), Bresenham's Algorithm, Generation of Circles

TWO DIMENSIONAL TRANSFORMATIONS and CLIPPING AND WINDOWING
What is transformation?, Matrix representation of points, Basic transformation, Need for Clipping and Windowing, Line Clipping Algorithms, The midpoint subdivision Method, Other Clipping Methods, Sutherland - Hodgeman Algorithm, Viewing Transformations

GRAPHICAL INPUT TECHNIQUES
Graphical Input Techniques, Positioning Techniques, Positional Constraints, Rubber band Techniques

EVENT HANDLING AND INPUT FUNCTIONS
Introduction, polling, event queue, functions for handling events, polling task design, Input functions, dragging and fixing, hit detection, OCR.

THREE DIMENSIONAL GRAPHICS
Need for 3-Dimensional Imaging, Techniques for 3-Dimesional displaying, Parallel Projections, Perspective projection, Intensity cues, Stereoscope effect, Kinetic depth effect, Shading

CURVES AND SURFACES

Shape description requirements, parametric functions, Bezier methods, Bezier curves, Bezier surfaces, B-Spline methods

SOLID AREA SCAN CONVERSION and Three Dimensional Transformations

Solid Area Scan Conversion, Scan Conversion of Polygons, Algorithm Singularity, Three Dimensional transformation, Translations, Scaling, Rotation, Viewing Transformation, The Perspective, Algorithms, Three Dimensional Clipping, Perspective view of Cube

HIDDEN SURFACE REMOVAL

Need for hidden surface removal, The Depth - Buffer Algorithm, Properties that help in reducing efforts, Scan Line coherence algorithm, Span - Coherence algorithm, Area-Coherence Algorithms, Warnock's Algorithm, Priority Algorithms

Table of The Contents

UNIT 4
SIMPLE LINE DRAWING METHODS

UNIT 5
TWO DIMENSIONAL TRANSFORMATIONS

UNIT 6
CLIPPING AND WINDOWING

UNIT 7
GRAPHICAL INPUT TECHNIQUES

UNIT 8
EVENT HANDLING AND INPUT FUNCTIONS

8.4 functions for handling events
8.5 polling task design
8.6 Input functions
8.7 dragging and fixing
8.8 hit detection
8.9 OCR.

UNIT 9
THREE DIMENSIONAL GRAPHICS

9.1 INTRODUCTION
9.2 Need for 3-Dimensional Imaging
9.3 Techniques for 3-Dimesional displaying
9.4 Parallel Projections
9.5 Perspective projection
9.6 Intensity cues
9.7 Stereoscope effect
9.8 Kinetic depth effect
9.9 Shading

UNIT 9
CURVES AND SURFACES

9.1 Shape description requirements
9.2 Parametric functions
9.3 Bezier methods
9.4 Bezier curves
9.5 Bezier surfaces
9.6 B-Spline methods

UNIT 9
SOLID AREA SCAN CONVERSION

9.1 Introduction
9.2 Solid Area Scan Conversion
9.3 Scan Conversion of Polygons
9.4 Algorithm Singularity

UNIT 10
Three Dimensional Transformations

10.1 Introduction
10.2 Three-Dimensional transformation
10.3 Translations
10.4 Scaling

UNIT – 1

BASICS OF COMPUTER GRAPHICS

1.1 Introduction

In this unit, you are introduced to the basics of computer graphics. To begin with we should know why one should study computer graphics. Its areas of application include design of objects, animation, simulation etc. Though computer graphics gained importance after the introduction of monitors, these are several other input and output devices that are important for the concept of computer graphics. They include high-resolution color monitors, light pens, joysticks, mouse etc. You will be introduced to the working principles of them.

The concept of computer graphics simply means identifying their areas of the screen that are to be illuminated and those that should not be. Most of the regular figures like straight lines, circles etc, are represented by mathematical equations. Given such equations, the first aspect of computer graphics is to convert them to a sequence of points - picture cells or pixels that are to be illuminated (in case of raster graphic display) or simply covert it to a curve that should be traced on the screen. Since many times these jobs have to be very fast and efficient. You will be introduced to a number of such algorithms and also their limitations.

We also look into the concept of transformations. Whenever as existing is to be moved to a new place or say to be zoomed, the drawing is not done again on the other hand; we only try to transform them. Simple transformation matrices for various operations are also introduced. Further, often we may end up drawing pictures larger than these that can be

represented on the screen. In such cases, we have a mechanism of "clipping" it to the required dimensions. We also have schemes that fit a given picture into a "window" of suitable size and location.

Further, since the computer is an "exact" device, in the sense it cannot approximate operations; sometimes it becomes difficult for the human beings to input exact values, like making the lines join exactly or the ends of a circle meeting perfectly etc. To take care of such cases, certain "constraints" are introduced so that the computer can know what the input is about - or looking at the other way, one cannot "approximate" things he is "constrained" to make them perfect. Similarly there are several other graphical input techniques that allow the user to interactively input the data, mostly drawings, without giving rise to ambiguities. These are also dealt with in this unit.

1.2 What is computer Graphics?

Computer graphics is an art of drawing pictures, lines, charts, etc using computers with the help of programming. Computer graphics is made up of number of pixels. Pixel is the smallest graphical picture or unit represented on the computer screen. Basically there are two types of computer graphics namely

1) **Interactive computer graphics:** It is the computer graphics in which user can interact with the image on the computer screen. Here exist two-way communication between the user and the image. The image is totally under the control of user. Example: Playing the computer game in the computer. Here user controls the image completely. According to the user wish image makes the movements on the screen.

2) **Non-interactive computer graphics:** it is the computer graphics in which user does not have any kind of control over the image. Image is merely the product of static stored program and will work according to the instructions given in the program linearly. The image is totally under the control of program instructions not under the user. Example: screen savers.

1.3 Areas of Computer Graphics

As ancient says " a pixel is worth thousand words", graphics is essential everywhere to understand the things, concepts, etc easily. Computer graphics is useful in almost all part of our life. In the following sections we are discussing some of the popular areas of computer graphics.

1.3.1 Design and Drawing

In almost all areas of engineering, be it civil, mechanical, electronic etc., drawings are of prime importance. In fact, drawing is said to be the language of engineers. The ability of computers to store complex drawings

and display them on demand was one of the major attractions for using computers in graphics mode. Few samples in this area are given below.

a) A mechanical engineer can make use of computer graphics to design nuts, bolts, gears etc.

b) Civil engineer can construct the buildings, bridges, train tracks, roads etc on the computer and can see in different angles and views before actually putting the foundation for them. It helps in finalizing the plans of these structures.

c) A text tile designer designs different varieties of designs through computer graphics

d) Electronics and electrical engineers design their circuits, PCB designs easily through computer graphics.

1.3.2 Animation

Making the pictures to move on the graphical screen is called animation. Animation really makes the use of computers and computer graphics interesting. Animation brought the computers pretty close to the average individuals. It is the well known principle of moving pictures that a succession of related pictures, when flashed with sufficient speed will make the succession of pictures appear to be moving. In movies, a sequence of such pictures is shot and is displayed with sufficient speed to make them appear moving. Computers can do it in another way. The properties of the picture can be modified at a fairly fast rate to make it appear moving. For example, if a hand is to be moved, say, the successive positions of the hand at different periods of time can be computed and pictures showing the position of the hand at these positions can be flashed on the screen. This led to the concept of "animation" or moving pictures. In the initial stages, animation was mainly used in computer games.

However, this led to a host of other possibilities. As we see later on in this course, computers not only allow you to display the figures but also offer you facilities to manipulate them in various ways – you can enlarge, reduce, rotate, twist, morph (make one picture gradually change to another – like an advertisement showing a cheetah change into a motor bike) and do a whole lot of other things. Thus, a whole lot of films made use of computers to generate tricks. In fact, several advertisement films and cartons strips are built with no actors at all – only the computer generated pictures.

Animation also plays very important role in training through computer graphics. If you have been given a bicycle you might have learn to ride it easily with little effort, but if you have been given a flight, automatically it needs the animated images to study the entire scenario of how flight takes

off, on and handling it during flying, contacting with and getting the help from control room etc will be better explained using computers animation technique.

1.3.3 Multimedia applications

The use of sound cards to make computers produce sound effect led to other uses of graphics. The concept of virtual reality, where in one can be taken through an unreal experience, like going through an unbuilt house (to see how it feels inside, once it is built) are possible by the use of computer graphics technology. In fact the ability of computers to convert electronic signals (0 & 1) to data and then on to figures and pictures has made it possible for us to get photographs of distant planets like mars being reproduced here on the earth in almost real time.

1.3.4 Simulation

The other revolutionary change that graphics made was in the area of simulation. Basically simulation is a mockup of an environment elsewhere to study or experience it. The availability of easily interactive devices (mouse is one of them, we are going to see a few other later in the course) made it possible to build simulators. One example is of flight simulators, wherein the trainee, sitting in front of a computer, can operate on the interactive devices as if he were operating on the flight controls and the changes he is expected to see outside his window are made to appear on the screen, so that he can master the skills of flight operations before actually trying his hand on the actual flights.

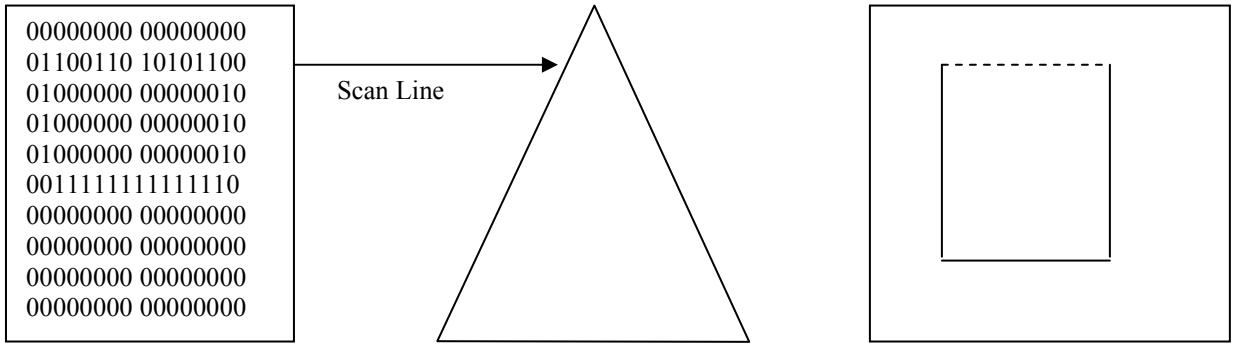
The graphic capabilities of computers are used in a very large variety of areas like criminology (to recreate faces of victims, assailants etc.,) medical fields (recreating pictures of internal cavities, using signals sent by miniature cameras), recreation of satellite pictures etc.

1.4 How are pictures actually stored and displayed?

All operations on computers are in terms of 0's and 1's and hence figures are also to be stored in terms of 0's and 1's. Thus a picture file, when viewed inside the memory, can be no different from other files – a string of Os and 1s. However, their treatment when they are to be displayed makes the difference.

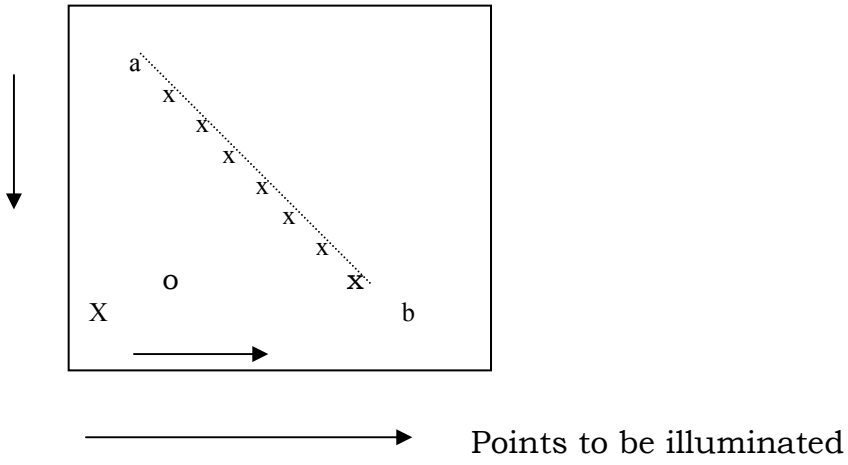
Pictures are actually formed with the help of frame-buffer display as follows

Frame buffer display contains a frame buffer, which is a storage device and stores the image in terms of 0's and 1's. It contains the 0's and 1's in terms of 8's, or multiples of 8's in a row. These 0's and 1's will be read by display controller one line at a time and sent to the screen after converting them from digital to analog. The display controller reads the contents of frame buffer one line at a time or entire digits at time. These digital images after converting into the analog will be displayed on the screen. The following figure illustrates this



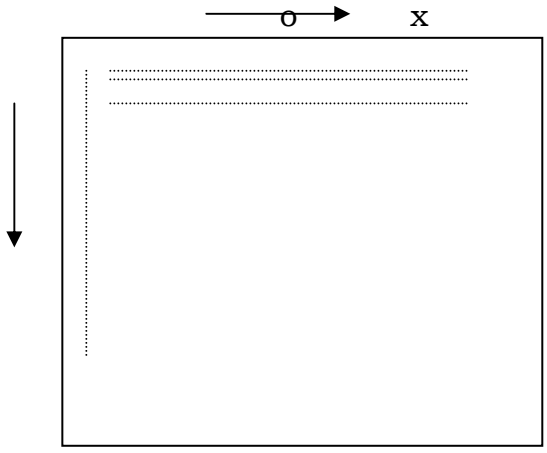
Frame Buffer **Display Controller** **Monitor**

Figures can be stored and drawn in two ways – either by line drawing or by Raster graphic methods. In the line drawing scheme, the figures are represented by equations – for example a straight line can be represented by the equation $y=mx+c$, a circle by $x^2+y^2=r^2$ etc. If (x, y) are representative points, then all these (x,y) value pairs which satisfy the equations form a part of the figure while those that do not, lie outside the figure. Thus, to generate any figure, obviously the equation of the figure is to be known. Then all points that satisfy the equation are evaluated. These are the points to be illuminated on the screen.



A moving electronic beam, as we know illuminates the screen, or the monitor. Whenever the beam is switched on, the electrons illuminate the phosphorescent screen and display a point. In the line drawing schemes, this beam is made to traverse the path of the figure to be traced and we get the figure we need. For example, in the above cited example if the electron beam is made to move from a to be along the points, we get the line.

The raster scan mechanism uses a different technique and is often found more convenient to manipulate and operate with. In this case, a "frame buffer", (a chunk of memory) is made to store the pixel values. (Remember, the screen can be thought of as having beam made up of a number of horizontal rows of pixels (picture cells), each pixel representing a point on the picture. In fact the number of such horizontal and vertical points indicate higher resolutions and therefore better pictures. Typical resolutions are like 640 X 480, 860 X 640, 1024 x 860 etc., where the figures indicate the number of rows and the number of pixels along each row respectively on a computer screen (unlike in standard mathematics) the top left hand point indicates the origin or the point (0,0) and the distances are measured horizontally and vertically as shown).

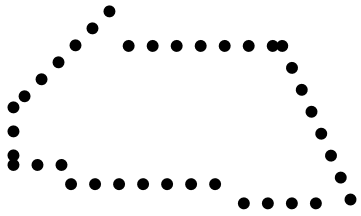


Now, assuming a 1024 x 1024 point screen, any figure that is to be displayed within this space. The "frame buffer" stores "status" of each of these pixels - say 0 indicates the pixel is off and hence is not a part of the picture and 1 indicates it is a part of the picture, and is to be displayed. This data is used to display the pictures.

1.5 Difficulties for displaying pictures

Unfortunately, the concept of graphics of displaying pictures is lot more complicated than what has been described so far - evaluate the points using the equations, store them in a file and use raster graphics methods or use simple line drawing algorithms. We will list a few of them before we close this chapter.

i) **Stair case effects:** Note that the pixel values are always integers (0,0) (0,1) (0,2) - - - - - -- - -, but an algorithm to draw/manipulate pictures need not always return integer values. Suppose the point at which two line meet, say is at (1.4, 2.7). What do we do? Common sense suggests that we round off the values, by using any of the standard algorithms. Excellent. 1.4 gets rounded of to 1 and 2.7 to 3. But another value of 1.6 say gets rounded off to 2 and a value of 3.1 also gets rounded of to 3. So, what do we have? The pointer 1.4 and 1.6, which should be very close to each other, appear to be separated by a distance of 1 and not 0.2 in our figure, i.e. the smoothness of a figure joining these points is lost. Alternately, the points 2.7 and 3.1, instead of appearing to be different, appear to be the same in our picture. A no. of such adjustments makes the figure looks like a jagged one instead of a smooth figure.



(Why this is called a stair case effect and how we can reduce it, we will see in due course)

ii) **Response time:** Especially when talking of animation, the speed at which new calculations are made and the speed at which the screen can interact are extremely important. Imagine a running bus, shown on the screen. Each new position of the bus (and it's surroundings, if needed) is to be calculated and sent to the screen and the screen should delete the earlier position of the bus and display its new position. All this should happen at a speed that convinces the viewer that the vehicle is actually moving at the prescribed speed, otherwise a running vehicle would appear like a "walking" bus or worse a "piecewise movement" bus. For this, most the speed of the algorithm and the speed of the display devices are extremely important. Further, the entire operation should appear smooth and not jerky otherwise, especially in simulation applications, the effects can be dangers.

iii) **What happens when the size of the picture exceeds the size of the screen?:** Obviously, some areas of the picture are to be cut off. But this involves certain considerations and needs to be addressed by software. [Which we will discuss while discussing about clipping and windowing]

iv) **Can the user create pictures directly on the screen?:** Definitely all pictures can not be thought of in terms of regular geometric figures and hence in terms of equations? Now, seeing a particular picture on the screen, the viewer wants to change it slightly, say bend it slightly here, stretch it their etc. This may not suit any regular equation? How should the system handle it?

The subsequent blocks answer these and many other questions.

Review Questions

1. The art of representing moving pictures is called _____-

2. The concept of changing one picture gradually into another is called _____

3. The combination of calculations, sound and pictures in computer is called _____

4. Building a mock up of an environment with the aim of studying the same is called _____

5. The equation of a straight line is given by _____

6. A block of memory to store pixel values is called _____

7. The number of pixels available for display of pictures is indicated by _____

8. The concept of creating pictures directly on the screen is called _____

Answers

1. Animation
2. Morphing
3. Multimedia
4. Simulation
5. y= mx+c
6. Frame buffer
7. Resolution
8. Interactive graphics.

Unit 2
GRAPHIC DEVICES

2.1 Introduction

Due to the widespread reorganization of the power and utility of computer graphics in almost all fields, a broad range of graphics hardware and software systems are available now. Graphics capabilities for both two-dimensional and three-dimensional applications are now common on general-purpose computers, including many hand-held calculators. These need wide variety of interactive devices.

In this unit, we will look into some of the commonly used hardware devices in conjunction with graphics. While the normal concept of a CPU, Memory and I/O devices of a computer still holds good, we will be concentrating more on the I/O devices. The special purpose output devices that allow us to see pictures in color, for example, with different sizes, features etc. Also, once the picture is presented, the user may like to modify it interactively. So one should be able to point to specific portions of the display and change them. Special input devices that allow such operations are also introduced. While ever changing technologies keep producing newer and newer products, what you are being introduced to here are trends of technology.

2.2 The Cathode Ray Tube (CRT/Monitor)

One of the basic and commonly used display devices is Cathode Ray Tube (CRT). A cathode ray tube is based on the simple concept that an electronic beam, when hits a phosphorescent surface, produces a beam of light (momentarily - though we later describe surfaces that produce light intensities lashing over a period of time). Further, the beam of light itself can be focused to any point on the screen by using suitable electronic / magnetic fields. The direction and intensity of the fields will allow one to determine the extent of the

defection of the beam. Further these electronic / magnetic fields can be easily manipulated by using suitable electric fields with this background. In following section we describe the structure and working of the simple CRT.

Simple CRT makes use of a conical glass tube. At the narrow end of the glass tube an electronic gun is kept. This gun generates electrons that will be made to pass through the magnetic system called yoke. This magnetic system is used for making the electronic beam to fall throughout the broad surface of the glass tube. The broad surface of the glass tube contains a single coat of high quality phosphorus. This reflects the electronic beam makes it to fall on the computer screen.

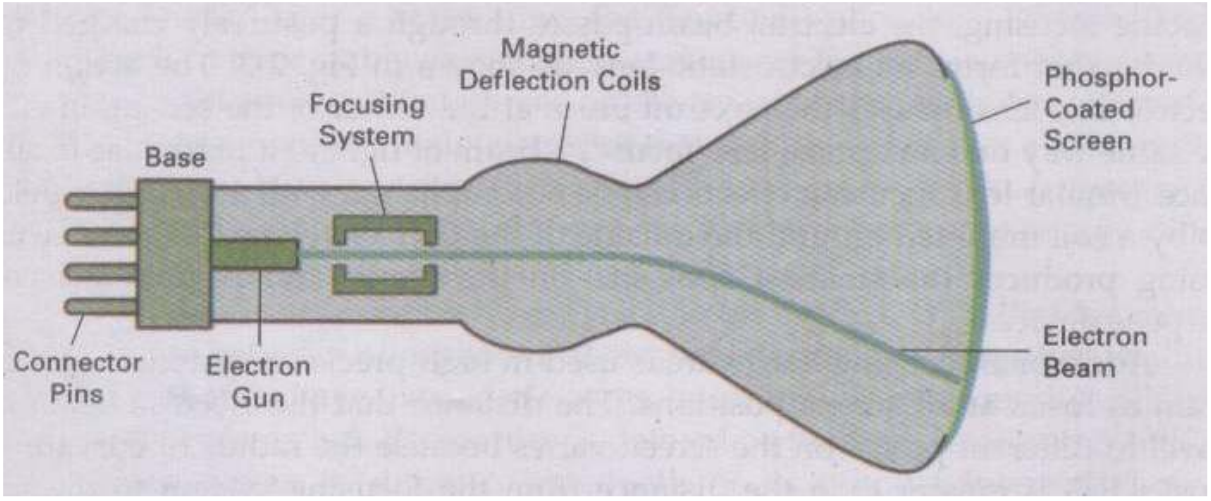
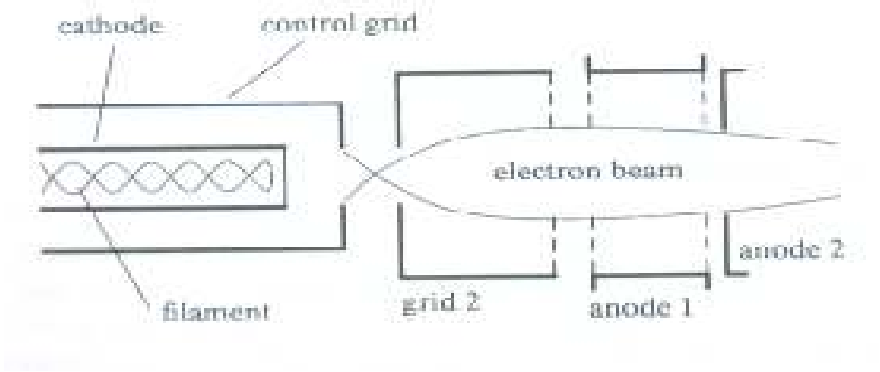


Fig. Basic Design of magnetic Deflection CRT

A pair of focusing grids - one horizontal and another vertical does the actual focusing of the electronic beam on to the screen. Electronic or magnetic fields operate these grids. Depending on the direction (positive or negative) and the intensity of the fields applied to them, the beam is deflected horizontally (or vertically) and thus, by using a suitable combination of these focusing grids; the beam can be focused to any point on the screen.



So, we now have a mechanism wherein any point on the screen can be illuminated (or made dark by simply switching off the beam).

Hence, from a graphics point of view, any picture can be traced on the screen by the electron beam by suitably and continuously manipulating the focusing grids and we get to see the picture on the screen "A basic graphic picture" of course, since the picture produced vanishes once the beam is removed, to give the effect to continuity, we have to keep the beam retracing the picture continuously - (Refreshing).

Quality of Phosphors

The quality of graphic display depends on the quality of phosphors used. The phosphors are usually chosen for their color characteristics and persistence. Persistence is how long the picture will be visible on the screen, after it is first displayed. Most of the standards prescribe that the intensity of the picture should fall to 1/10 of its original intensity is less than 100 milliseconds.

The color of the phosphor is normally chosen as white, also it should be of small grains, so that the resolution of the screen can be high.

However, special types of monitors, to suit special applications have been devised, which may not confirm to the above standards. We will see a few of them in the next sections.

2.3 CRTs for Color Display

This was one the earlier CRTs to produce color displays. Coating phosphors of different compounds can produce different colored pictures. But the basic problem of graphics is not to produce a picture of a predetermined color, but to produce color pictures, with the color characteristics chosen at run time.

The basic principle behind colored displays is that combining the 3 basic colors –Red, Blue and Green, can produce every color. By choosing different ratios of these three colors we can produce different colors - millions of them in-fact. We also have basic phosphors, which can produce these basic colors. So, one should have a technology to combine them in different combinations.

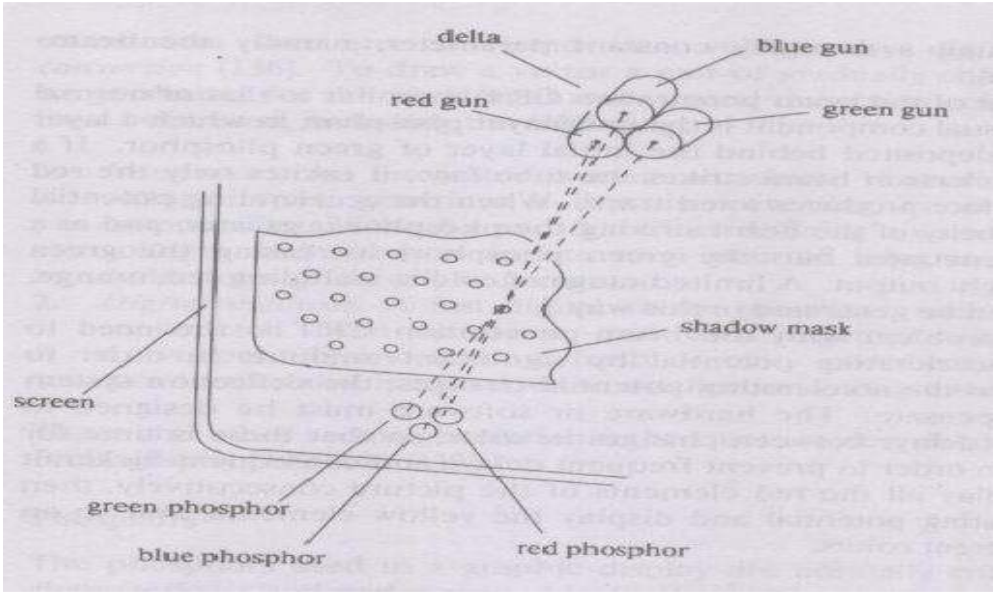
2.4 Beam Penetration CRT

This CRT is similar to the simple CRT, but it makes use of multi coloured phosphorus of number of layers. Each phosphorus layer is responsible for one colour. All other arrangements are similar to simple CRT. It can produce a maximum of 4 to 5 colours

The organization is something like this - The red, green and blue phosphorus are coated in layers - one behind the other. If a low speed beam strikes the CRT, only the red colored phosphorus is activated, a slightly accelerated beam would activate both red and green (because it can penetrate deeper) and a much more activated one would add the blue component also.

But the basic problem is a reliable technology to accelerate the electronic beam to precise levels to get the exact colors - it is easier said than done. However, a limited range of colors can be conveniently produced using the concept.

2.5 The Shadow - Mask CRT



This works, again, on the principle of combining the basic colors - Red, green and Blue - in suitable proportions to get a combination of colors, but it's principle is much more sophisticated and stable.

The shadow mask CRT, instead of using one electron gun, uses 3 different guns placed one by the side of the other to form a triangle or a "Delta"

as shown. Each pixel point on the screen is also made up of 3 types of phosphors to produce red, blue and green colors. Just before the phosphor screen is a metal screen, called a "shadow mask". This plate has holes placed strategically, so that when the beams from the three electron guns are focused on a particular pixel, they get focused on particular color producing pixel only i.e. If for convenience sake we can call the electronic beams as red, blue and green beams (though in practice the colors are produced by the phosphors, and until the beams hit the phosphor dots, they produce no colors), the metal holes focus the red beam onto the red color producing phosphor, blue beam on the blue producing one etc. When focused on to a different pixel, the red beam again focuses on to the red phosphor and so on.

Now, unlike the beam penetration CRTs where the acceleration of the electron beam was being monitored, we now manipulate the intensity of the 3 beams simultaneously. If the red beam is made more intense, we get more of red color in the final combination etc. Since fine-tuning of the beam intensities is comparatively simple, we can get much more combination of colors than the beam penetration case. In fact, one can have a matrix of combinations to produce a wide variety of colors.

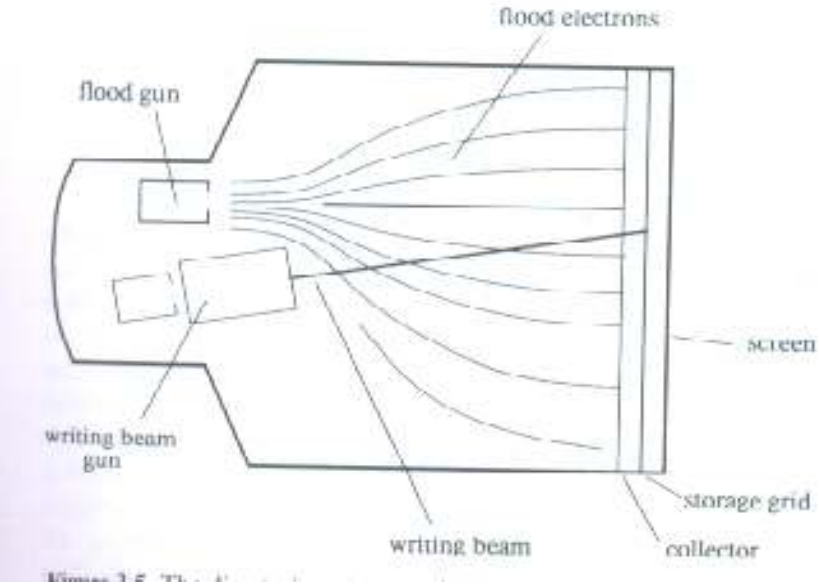
The shadow mask CRT, though better than the beam penetration CRT in performance, is not without it's disadvantages. Since three beams are to be focused, the role of the "Shadow mask" becomes critical. If the focusing is not achieved properly, the results tend to be poor. Also, since instead of one pixel point in a monochrome CRT now each pixel is made up of 3 points (for 3 colors), the resolution of the CRT (no. of pixels) for a given screen size reduces. Another problem is that since the shadow mask blocks a portion of the beams (while focusing them through the holes) their intensities get reduced, thus reducing the overall brightness of the picture. To overcome this effect, the beams will have to be produced at very high intensities to begin with. Also, since the 3 color points, though close to each other, are still not at the same point, the pictures tend to look like 3 colored pictures placed close by, rather than a single picture. Of course, this effect can be reduced by placing the dots as close to one another as possible.

The above displays are called refresh line drawing displays, because the picture vanishes (typically in about 100 Milli seconds) and the pictures have to be continuously refreshed so that the human persistence of vision makes them see as static pictures. They are costly on one hand and also tend to flicker when complex pictures are displayed (Because refreshing because complex). These problems are partly overcome by devices with inherent storage devices - i.e. they continue to display the pictures, till they are

changed or at least for several minutes without the need of being refreshed. We see one such device called the Direct View Storage Tube (DVST) below.

2.6 Direct View Storage Tube

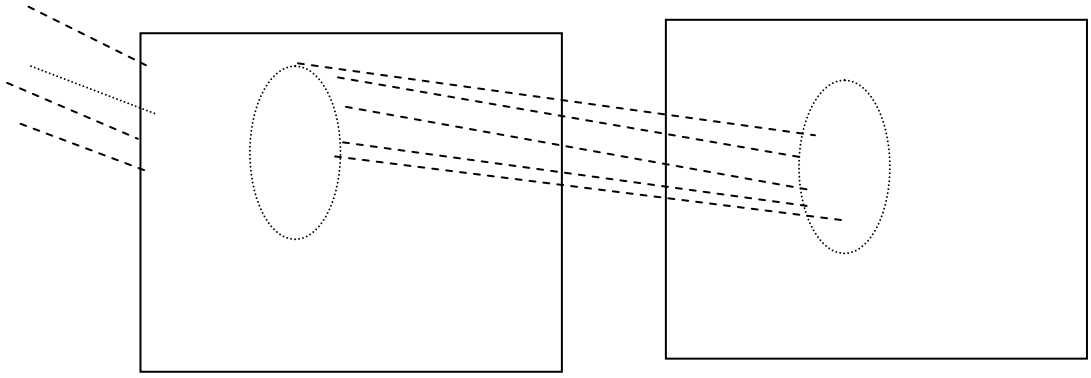
Conceptually the Direct View Storage Tube (DVST) behaves like a CRT with highly persistent phosphor. Pictures drawn on there will be seen for several minutes (40-50 minutes) before fading. It is similar to CRT as far as the electronic gun and phosphor-coated mechanisms are concerned. But instead of the electron beam directly writing the pictures on the phosphor coated CRT screen, the writing is done with the help of a fine-mesh wire grid.



The grid made of very thin, high quality wire, is located with a dielectric and is mounted just before the screen on the path of the electron beam from the gun. A pattern of positive charges is deposited on the grid and this pattern is transferred to the phosphor coated CRT by a continuous flood of electrons. This flood of electrons is produced by a "flood gun" (This is separate frame the electron gun that produces the main electron beam).

Just behind the storage mesh is a second grid called the collector. The function of the collector is to smooth out the flow of flood electrons. Since a large number of electrons are produced at high velocity by the flood gun, the collector grid, which is also negatively charged reduces, the acceleration on these electrons and the resulting low velocity flood pass through the collector and get attracted by the positively charged portions of the storage mesh (Since the electrons are negatively charged), but are repelled by the other portions of

the mesh which are negatively charged (Note that the pattern of positive charges residing on the storage mesh actually defines the picture to be displayed). Thus, the electrons attracted by the positive charges pass through the mesh, travel on to the phosphor coated screen and display the picture. Since the collector has slowed the electrons down, they may not be able to produce sharp and bright images. To over come this problem, the screen itself is maintained at a high positive potential by means of a voltage applied to a thin aluminum coating between the tube face and the phosphor.



Mesh of thin gauge CRT Screen

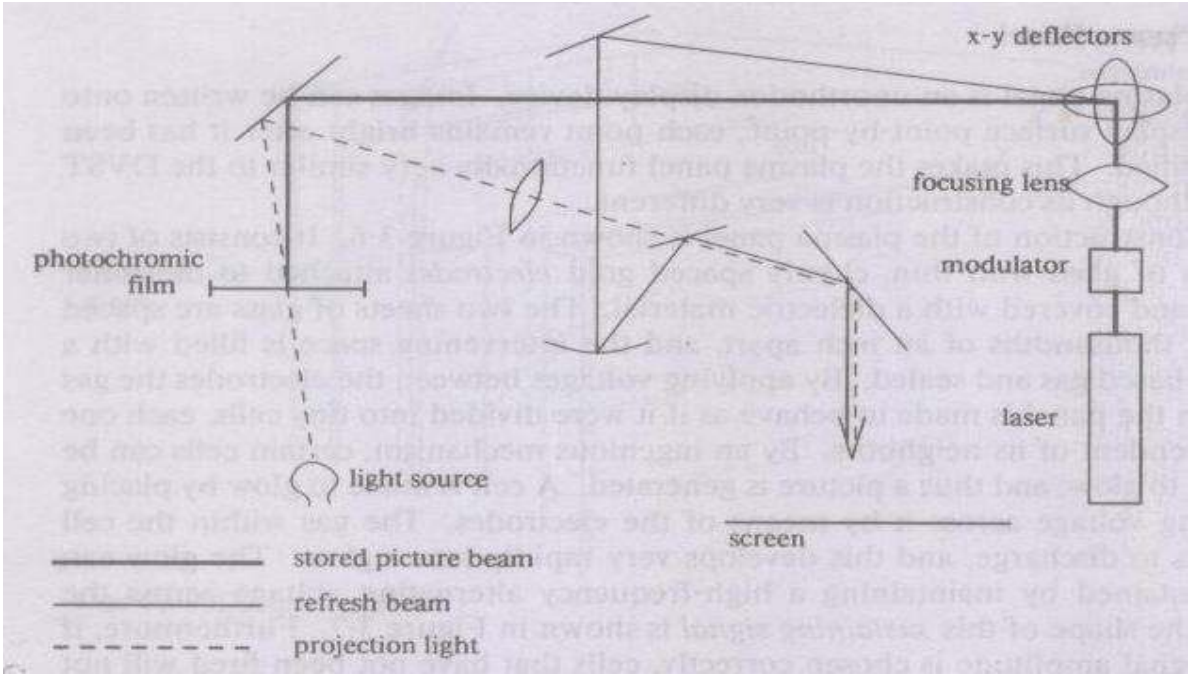
The dotted circle on the mesh is created by positive charges the flood of electrons hit the mesh at all points. But only those electrons that hit the dotted circle pass through and hit the CRT screen. The negatively charged mesh repels others.

Since the phosphor is of a very high persistence quality, the picture created on the CRT screen will be visible for several minutes without the need for being refreshed.

Now the problem arises as to how do we remove the picture, when the time for it's erasure or modification comes up. The simple method is to apply a positive charge to the negatively charged mesh so that it gets neutralized. This removes all charges and clears the screen. But this technique also produces a momentary flash, which may be unpleasant to the viewer. This is mainly so when only portions of the picture are to be modified in an interactive manner. Also, since the electrons hit the CRT screen at very low speeds (though they are slightly accelerated in the last part of their journey to the CRT by a positively charged aluminum coating), the contrasts are not sharp. Also, even though the pictures stay for almost an hour, there will be a gradual degradation because of the accumulation of the background glow. The

other popular display device is the plasma panel device, which is partly similar to the DVST in principle, but over comes some of the undesirable features of the DVST.

2.7. Laser Scan Display



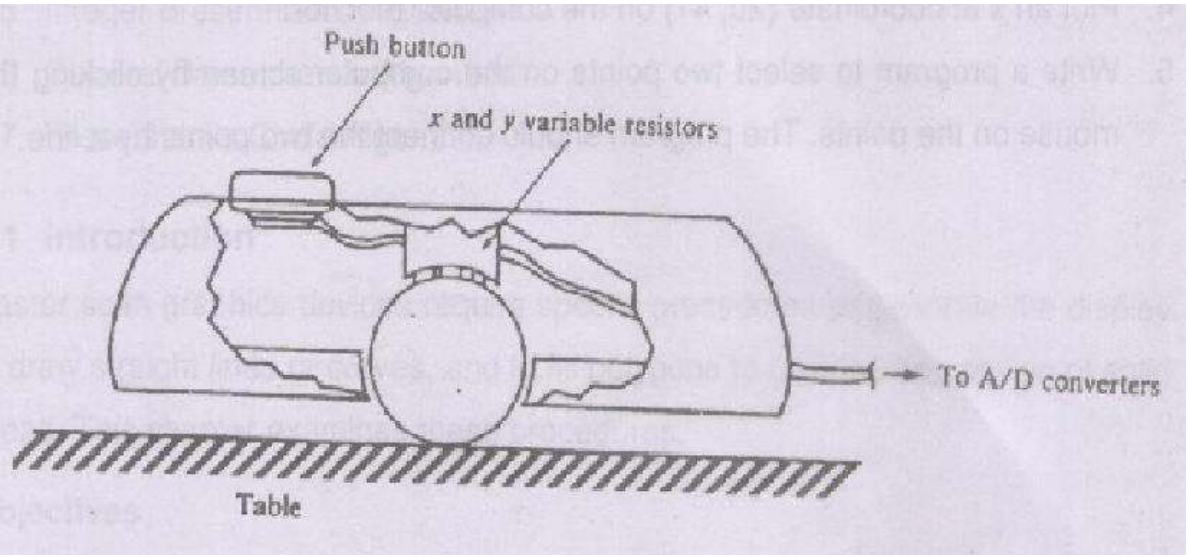
The laser-scan display is one of the high resolutions, large screen display device. It is capable of displaying an image measuring 3 by 4 feet and still has good resolution. The main principle behind working of this display device is light source mixed with laser light and the deflection of laser light according to the natural light source. Modulators, focusing lenses and x-y deflectors make laser light to deflect and fall on the screen where natural light falls.

2.8. Input Devices

We shall now see some of the popularly used input devices for interactive graphics.

i) Mouse:

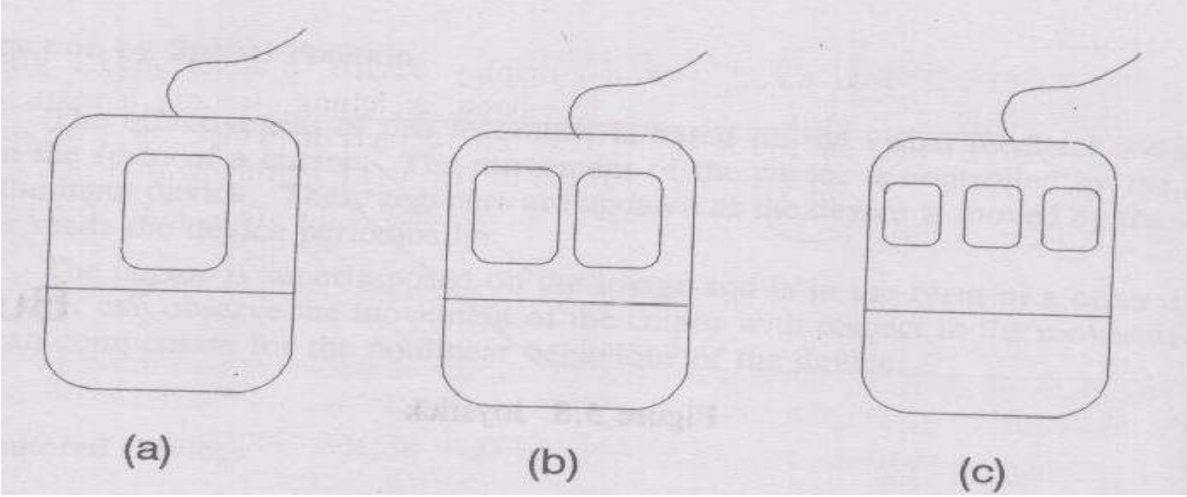
The mouse consists of a small plastic box resting on a metal wheel (see Fig). It was developed originally at Stanford Research Institute. The wheel of the mouse is connected to two variable resistors that deliver analog voltage for every incremental rotation of the wheel. As the mouse is rolled around on a flat surface, its movement in two orthogonal directions is translated into rotation of the wheel. These rotations can be measured by converting the analog voltages to digital values. The converted values may be held in registers accessible to the computer or written directly into the computer's memory; the values are normally sampled 30 or 60 times a second by the computer. Pushbutton is mounted on top of the mouse, and the user can work them with his fingers as he moves the mouse. Ideally the computer should be able to read the position of these buttons whenever it reads the coordinates of the mouse.



In addition to its simplicity and low cost, the mouse has the advantage that the user need not pick it up in order to use it-the mouse simply sits on the table surface until he needs it. This makes the mouse an efficient device for pointing, as experiments have shown. The mouse has some unique properties that are liked by some and disliked by others. For example, if the mouse is picked up and put down somewhere else, the cursor will not move. also, the coordinates delivered by the mouse wrap around when overflow occurs; this effect can be filtered out by software, or can be retained as a means of moving the cursor rapidly from one side of the screen to the other. The mouse has two real disadvantages. It cannot be used for tracing data from paper, since a small rotation of the mouse or a slight loss of contact will cause accumulative error in all the reading, and it is very difficult to handprint

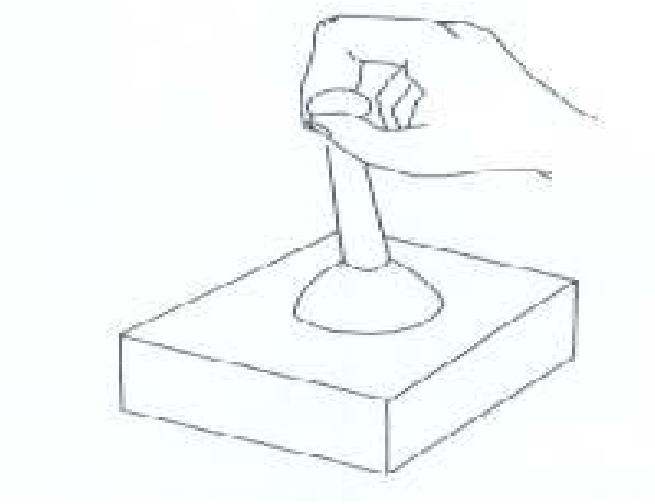
characters for recognition by the computer. For these types of application a tablet is essential.

The following figure shows one button, two buttons and three buttons mouse.



ii) Joystick

In fact, the forerunner of the mouse is a joystick. Here, as the name suggests, we have a stick (or a handle, to be more exact) can be moved in all possible direction.



A joy Stick

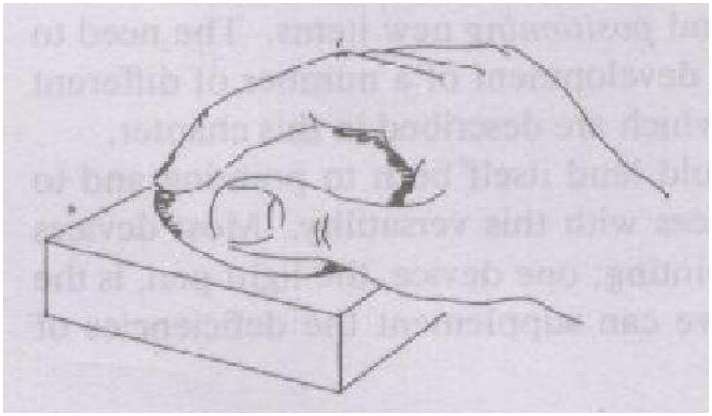
The direction and amount of movement in that direction controls the amount of curser movement. Once the curser arrives at the desired position, clicking the buttons can choose the picture and any modification can be made.

In fact, the joysticks were originally used for video games (hence the name "joy" stick), but later on modified for the more accurate graphics requirements.

However, both the mouse and joysticks may appear a bit cumbersome for the new users. They find some difficulty in aligning the curser to the precisely desired positions.

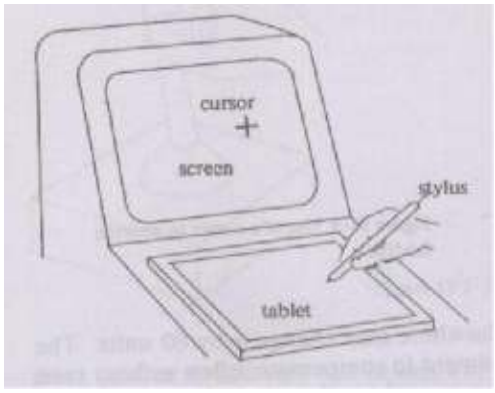
iii) Tracker Ball

The working principle of tracker ball is similar to that of mouse or joystick. In this device instead of holding the device ball inside the device, it will be hold in hand and rotated. According to the ball movement on the roller groove fixed with rollers for X, Y and Z –axis. According to the movements of the ball the rollers will move and they give the position of the pointer on the screen. The following figure shows tracker ball.

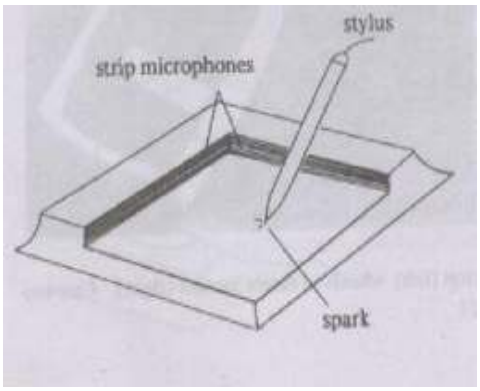


2.9 TABLETS

The Tablets work on the principle of sound and its speed through which the position of the pointer on the screen will be decided. It makes use of flat surface on which we are writing with a stylus. The stylus tip is covered with material called ceramic. It makes sound when writing on the flat surface.



Tablet and stylus

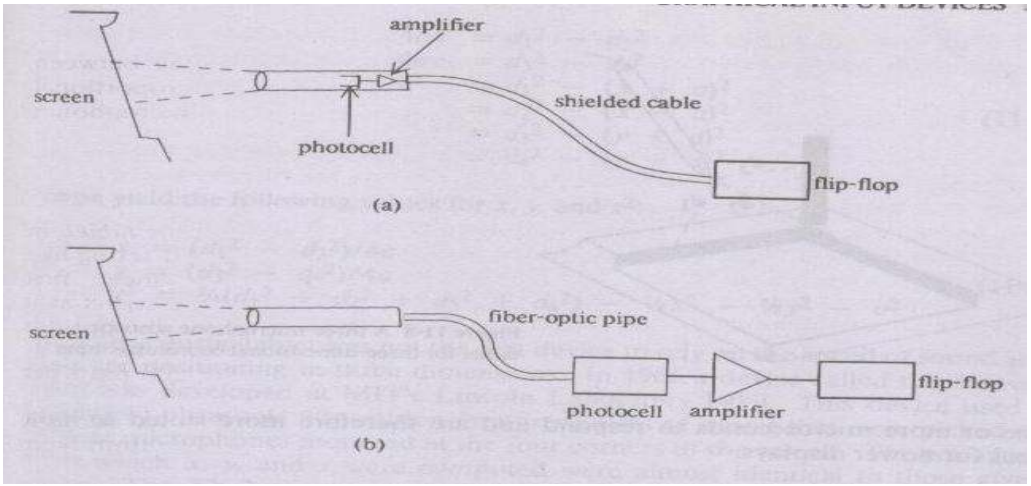


The acoustic tablet

The flat surface is having powerful strip microphones on two sides. These microphones will receive the sound generated by the stylus while writing. Based on the actual time of sound generated and sound received by two microphones will decide the position of the pointer on the screen. In case of 3D acoustic Tablet three strip microphones are used to along the sides of the flat surface and one vertically on their intersecting point.

2.10 The light Pen

The devices discussed so far, the mouse, the tablet, the joystick are called "positioning devices". They are able to position the curser at any point on the screen. (Which in turn means, we can operate at that point or the chain of points)



LIGHT PEN A) USING HAND-HELD PHOTOCELL, B) USING A FIBER OPTIC PIPE

Often, we also need devices that can "point" to a given position on the screen. This becomes essential when a diagram is already there on the

screen, but some changes are to be made. So, instead of trying to know its coordinates, it is advisable to simply "point" to that portion of the picture and ask for changes. The simplest of such devices is the "light pen". Its principle is extremely simple.

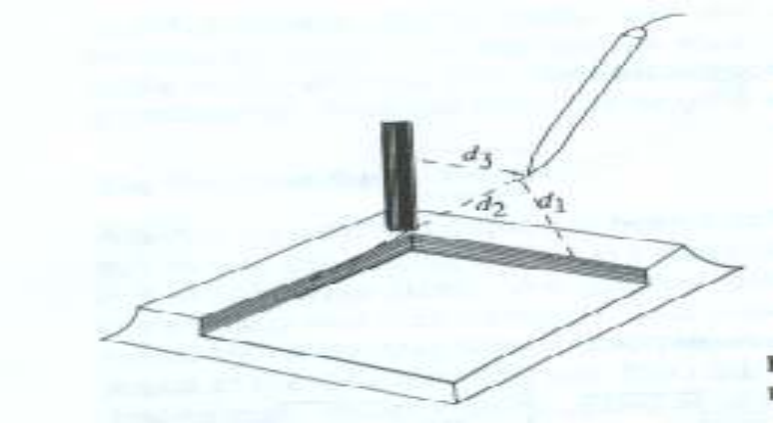
We know that every pixel on the screen that is a part of the picture emits light. In fact they are much brighter than their surrounding pixels. All that the light pen does is to make use of this light signal to indicate the position. A small aperture is held against the portion of the picture to be modified and the light from the pixels, after passing through the operator falls on a photocell. This photocell converts the light signal received from the screen to an electrical pulse - a signal sent to the computer. Since the electrical signal is rather weak, an amplifier amplifies it before being sent to the computer. Since a "tracking software" keeps track of the position of the light pen always (in a manner much similar to the position of the mouse being kept track of by the software), a signal received by the light pen at any point indicates that portion of the picture that needs to be modified (most often that portion gets erased, paving way for any other modifications to be made).

However, when the pen is being moved to it's position - where the modification is required - it will encounter so many other light sources on the way and these should not trigger the computer. So the operator of the light pen is normally kept closed and when the final position is reached, then it can be opened by a switch - in a manner similar to the one used in a photographic camera, though, of course, the period of opening the operator is for much longer periods than in a camera.

2.10 Three Dimensional Devices

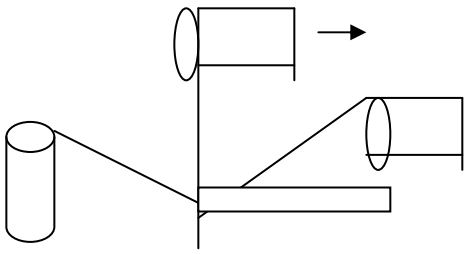
Though the display on the CRT monitor always presents a 2 dimensional picture, it is not necessary that the data stored in the computer about the picture also should be two-dimensional. In particular, when one is taking data from 3-dimensional models it becomes necessary to map input data, which is 3-dimensional in nature into the 2-dimensional pictures. This aspect will be dealt with in a later chapter. However, the input devices should be able to read and transfer data from a 3-dimensional world, in the first place. The devices that we have seen so far, namely the mouse or light pen or joysticks or even tablets work only on two-dimensional data only.

In this section, we see the simplest of input devices, which work only on the extended principle of the 2-dimensional tablet that we have encountered earlier.



The concept is that when two perpendicularly placed microphones can pick up signals and identify them in a 2 dimensional space, 3-perpendiculorly placed microphones can pickup and identifies signals in a 3-dimensional space. The result is the above figure.

But when a 2 -dimensional tablet is made 3-dimensional by adding a third, perpendicular microphone, the tablet becomes more difficult to manage because of the bulk. Hence one more mechanism, wherein a 2-dimensional tablet can be used to affect a 3 dimensional recognition was developed. In this case, all the four sides of the tablet are provided with a microphone each and it can be mathematically shown that any sound made by the stylus tip at a height above the tablet is picket up by the four microphones, the time delays will be proportional not only to the x and y distances of the stylus form the microphones, but also to it's height above the stylus - the z distance. By using very simple mathematics - it is possible to separate the x,y and z values, i.e. the actual position of the stylus.



One more simple method of tracking in 3 dimensions is by the use of wires in 3 dimensions. The trick is to connect the stylus to 3-wires, positioned in x,y and z direction, connected to several length of wires and which are spring loaded. The distance of the stylus from each of these springs

is proportional to the force applied on the springs, which can be used to indicate the position. However, this method is less accurate and is seldom used.

Review Questions

1. In a CRT, a stream of electrons falling on a _____of the screen produces images.

2. The path of the electron beam is focused on to the screen using _____ or _____.

3. The term _____ indicates how long the picture created on the phosphorescent screen remains on it.

4. The three basic colors are _____, _____ and _____

5. Different electron beams are accelerated to different levels in a _____

6. When the picture has to remain on the screen for a long time _____ type of CRT is sued.

7. The first device to allow the user to move the cursor to any point, without actually knowing the coordinates was _____

8. The input device that allows user to write pictures on it an input them directly to the computer is called _____

9. Light pen is a _____ device.

10. Name one device that allows a 3 dimensional input to be given to the computer.

Answers

1. Phosphorescent

2. Magnetic, electrical

3. Persistence

4. Red, Blue, Green

5. Beam Penetration CRT

6. Direct view storage Tube (DVST)

7. Joy stick

8. Tablet

9. Pointing

10. Acoustic Tablet.

UNIT 3
INTRODUCTION TO THE
'GRAPHICS' AND 'C'

3.1 Introduction
3.2 'C' GRAPHICS FUNCTIONS
3.3 C Graphics Programming Examples

3.1 Introduction

'C' is the language of choice for the system programming. It also provides the facility to draw the graphics on the screen. All the graphical related functions are kept in the header file graphics.h. C is a popular programming language. It supports computer graphics and provides number of standard library functions for drawing regular diagrams and figure on the computer screen. One can use these graphical functions to draw the images easily through computer program. For these we need to initialize the graphics mode and detect the related graphics drivers. The standard library functions are kept in the header file called "graphics.h". for using any of the graphical built-in functions "graphics.h" file must be included.

Before starting with the C language syntax for graphics let us discuss some of the important terms that are used in computer graphics.

a) **Pixel :** It is the smallest recognizable picture part on the computer screen. Each dot(.) we can draw on the computer screen id a pixel and any image or picture we draw is the combinations of pixels.

b) **Resolution:** The maximum number of pixels we can put on the computer screen along X-axis is called its resolution. The higher resolution leads to fine quality of an image. Usually it is 640 pixels along X-axis and 480 pixels along Y-axis. But these resolution changes from computer to computer-based on configuration and operating system as well as applications used.

c) **Coordinate system:** usually the coordinate system in C computer graphics considers only positive coordinates with integer values. The left top corner of the screen is origin and X keeps on increasing along X-axis horizontally upto right border of the screen and Y keeps on increasing vertically down until button border o the screen.

d) **Graph mode :** Integer that specifies the initial graphics mode (unless graph driver=DETECT is specified). If graph driver

= DETECT, initgraph sets graph mode to the highest resolution available for the detected driver. We can give graph mode a value using a constant of the 'graphics_modes' enumeration type available in graphics.h header file.

Some of the standard library functions of graphics.h header file used in this project are given below.

3.2 'C' GRAPHICS FUNCTIONS

For doing this project we are using the 'C' graphics. The functions using for our project are given below,

1.initgraph( ); Initializes the graphics system.

Declaration: void far initgraph (int far *graphdriver, int far *graphmode,

char far *path to driver);

Remarks: To start the graphics system, you must first call initgraph. Initgraph Initializes the graphics system by loading a graphics driver from Disk (or validating a registered driver) then putting the system into graphics mode.

2. setbkcolor( ); It sets the current background color using palette.

Declaration: void far setbkcolor (int color);

Remarks: setbkcolor sets the background to the color specified by color.

3. setcolor( ); setcolor sets the current drawing color.

Declaration: void far setcolor (int color);

Remarks: It sets the current drawing color to color, which can range from 0 to getmaxcolor.

4. rectangle( ); Draws a rectangle (graphics mode)

Declaration: void far rectangle (int left, int top, int right, int bottom);

Remarks: rectangle draws a rectangle in the current line style, thickness and (right, bottom) is its lower right corner.

5. settextstyle( ); Sets the current text characteristics.

Declaration:void far settextstyle (int font, int direction, int charsize);

Remarks: It sets the text font, the direction in which text is displayed and the size of the characters.

6. putimage( ); putimage outputs a bit image onto the screen.

Declaration:void far putimage (int left, int top, void far *bitmap, int top);

Remarks: putimage puts the bit image previously saved with getimage back onto the screen, with the upper left corner of the image placed at (left, top)

7. getimage ( ); getimage saves a bit image of the specified region into memory.

Declaration: void far getimage (int left, int top, int right, int bottom, void far * bitmap);

Remarks: getimage copies an image from the screen to memory.

8. malloc( ); It allocates the memory.

Declaration: void *malloc(size_t size) ;

Remarks: It allocates a back of size bytes from the memory heap. It allows a program to allocate memory explicitly as its needed and in the exact amount needed

9. floodfill( ); Flood_fills a bounded region.

Declaration: void far floodfill (int x, int y, int border);

Remarks: floodfill fills an enclosed area on bitmap devices. The areas bounded by the color border are flooded with the current fill pattern and fill color.

10. Closegraph( ); Shut down the graphics system.

Declaration: void far closegraph(void);

Remarks: It reallocates all memory allocated by the graphics system.

11. cleardevice( ); It clears the graphics screen.

Declaration: void far cleardevice(void);

Remarks: It erases the entire graphics screen and moves the current position (CP) to home(0, 0).

12. sleep( ); Suspends execution for interval.

Declaration: void sleep(unsigned seconds);

Remarks: With a call to sleep, the current program is suspended from execution for the number of seconds specified by the argument seconds.

13. exit( ); exit terminates the program.

Declaration: void exit(int status);

Remarks: Exit terminates the calling process.

14. sound( ); sounds turns the PC speaker on at the specified frequency.

Declaration: void sound(unsigned frequency);

Remarks: Sound turns on the PC's speaker at a given frequency.

15. nosound( ); sounds turns the PC speaker off.

Declaration: void sound(void);

Remarks: Sound turns on the PC's speaker off after it has been turned on by a call to sound.

16. textcolor( ); It selects a new character color in text mode.

Declaration: void textcolor(int newcolor);

Remarks: This function works that procedure text-mode output directly to the screen (console output functions), textcolor selects the foreground character color.

17. delay( ); It suspends execution for interval (milliseconds).

Declaration: void delay(unsigned milliseconds);

Remarks: With a call to delay, the current program is suspended from execution for the time specified by the argument milliseconds. It is not necessary to make a calibration call to delay before using it. It is accurate to one milliseconds.

18. imagesize( ); Returns the number of bytes required to store a bit image.

Declaration: unsigned far imagesize(int left, int top, int right, int bottom);

Remarks: determines the size of memory area required storing a bit images.

19. gotoxy( ); Positions cursor in text window.

Declaration: void gotoxy(int x, int y);

Remarks: gotoxy moves the cursor to the given position in the current text window. If the coordinates are invalid, the call to gotoxy is ignored.

20. line( ); line draws a line between two specified points.

Declaration: void far line(int x1, inty1, intx2, inty2);

Remarks: line draws a line from (x1, y1) to (x2,y2) using the current color, line style and thickness. It does not update the current position (CP)

3.3 C Graphics Programming Examples:

Let us consider a small program that illustrates graphics initialization.

```
/* Program to initialize the graph and draw a line * /
#include<graphics.h>
#include<conio.h>
void main()
{
int gd =DETECT:  /*Detect the graph driver dynamically*/
int gm;          /*for graph mode*/
```

```
initgraph (&gd,&gm,"");  /* graph driver, graph mode and
```
path has to be passed as parameters. The empty path is specified means the path will be taken dynamically after searching in the computer. Otherwise we need to spcify the path where bgi directory is stored in the computer */

```
line(10,10,200,200);      /* this function draws a line from
```
starting co-ordinates(10,10) to the target co-ordinates (200,200). These co-ordintes are specified in terms of pixels */

```
getch();
closegraph();        /* close the graph mode */
}
```

The above program draws a line in between the specified co-ordinates. The syntax of the popular graphical functions is given in the 'Appendix A'. in any of the graphical programs it is essential to detect the graph driver and set the graph mode or terminating the program execution. But it is always advisable to close the graphics mode before terminating the program.

Let us consider the program for drawing a rectangle.

```
/*program to initialize the graph and draw a rectangle */
#include<graphics.h>
#include<conio.h>
void main()
{
int gd=DETECT;   /* Detects the graph driver dynamically */
int gm;              *for graph mode*/
initgraph(&gd,&gm,"");
rectangle(10,10,200,200,);      /* This function draws a rectangle
```
taking co-ordinates (10,10) as top left point and target co-ordinates (200,200) as botton right co-ordinates. These co-ordinates are specified in terms of pixels */

```
getch();
colsegraph();        /* closes the graph mode */
}
```

The following program illustrates the combinations of different

Regular shaped graphical objects. Here we will draw a rectangle and lines along its diagonals. A circle is also drawn inside the rectangle.

```
/* program to draw a rectangle , lines as its diagonals and a circle */
#include<graphics.h>
#include<conio.h>
void main()
{
int gd=DETECT;
int gm;
initgraph(&gd,&gm,"");
rectangle(10,10,200,200);       /* draws a rectangle */
line(10,10,200,200);      /* draw a line  on the main diagonal*/
line(10,200,200, 10);     /* draws a line on off diagonal */
circle(105,105,95);             /* draws a circle taking (105,105)
                          as center co-ordinates and 95 as radius
                          all the dimensions are in pixels */

getch();
closegraph();             /*closes the graph mode */
}
```

By making use of the available library functions we can easily draw such graphics. We can also set the writing colors for drawing by using setcolor () function, which takes color code or color name as its parameter. C graphics supports sixteen colors whose codes rang from 0 to 15 (The graphical constants related to colors, styles, patterns etc., are given in appendix B). When the color name is used as the parameter it must be specified in capital alphabets. The following program illustrates the circles and ellipses drawn with different colors.

```
/* program to draw a circle and ellipses */
#include<graphics.h>
#include<conio.h>
void main()
{
int gd=DETECT;
int gm;
Initgraph (&gd,&gm,"");
Circle (getmaxx ()/2,getmaxy ()/2,100);     /*draws a circle taking midpoint of
the screen as center co-ordinates and
100 as radius */
```

```
setcolor(2);  /*sets the drawing color as green */

ellipse(getmaxx()/2,0,360,80,50);
/* draws an ellipse taking center of the screen as its center , 0 as starting angle
and 360 as ending angle and 80 pixel as Y radius */
setcolor(4);  /*sets the drawing color as red */
ellipse(getmaxx()/2, getmaxy()/2,90,270,50, 80);
/*draws half the ellipse starting from 90 angle and ending at 27o angle
with 50 pixels as X-radius and 80 pixels as Y-radius in red color */
getch();
closegraph();                /* closes the graph mode */
}
```

The different combinations of ellipses and arcs are illustrated in the following program.

```
/* program to draw ellipses and arcs */
#include<graphics.h>
#include<conio.h>
void main()
{
int gd=DETECT;
int gm;
initgraph(&gd,&gm,"");

setcolor(1);          /*sets the drawing color as blue*/
ellipse(getmaxx()/2, getmaxy()/2,0,360,80,50);
  /*draws an ellipse taking center of the screen as its center, 0as starting angle
and360 as ending angle and 80 pixels as x-radius, 50 pixels as y radius*/

setcolor(4);      /*sets the drawing color as red*/
ellipse(getmaxx()/2, getmaxy()/2, 90,270,50,80);
  /*draws half the ellipse starting from 90 degree angle and ending at 270
degree with 50 pixels as x-radius and 80 pixels as y-radius in red color*/

setcolor(5);      /*sets the drawing color as pink */
arc(getmaxx()/2, getmaxy()/2, 0, 180, 100);
 /*arc with center of the screen as its center and 100 pixels as radius. It starts
at an angle 0 and ends at an angle 180 degrees, i.e., half circle*/

setcolor(9);     /* sets the drawing color as light blue */
arc(300,200,20,100,70) ;
```

```
  /*arc with (300,200) as its center and 70 pixels as radius. It starts at an angle
20 and ends at an angle 100 degrees*/


getch();
closegraph();              /* closes the graph mode */
}
```

The following program illustrates the putpixel function. It keeps on drawing the pixels throughout the screen until pressing any key from the keyboard. The co-ordinates for drawing the pixel are selected randomly by using rand() library function and taking co-ordinates for x-axis and y-axis randomly within the limits of screen resolution.

```
/* program to demonstrate put pixel */
# include<graphics.h>
# include<conio.h>
# include <stdlib.h>
# include<dos.h>


void main()

{
int gm, gd=DETECT,I;
initgraph(&gd, &gm,"");
while(!kbhit())/* until pressing any key this loop continues */
{
putpixel(rand()%getmaxx(), rand() % getmaxy(), rand()%16);
/*x and y co-ordinates and the color are taken randomly*/
delay(2);          /* just to draw the pixels slowly*/
}
getch ();
closegraph();          /* closes the graph mode */
}
```

The proper combinations of pixels can make any of the graphical objects. The following program shows drawing of lines and rectangles using putpixel () function.

```
/* program to demonstrate rectangles using putpixel and lines*/
#include<graphics.h>
#include<conio.h>
#include<stdlib.h>
#include<dos.h>

void main()
{
int gm,gd=DETECT;
int x1,x2,y1,y2,c,I;

initgraph(&gd,&gm,"");
while(!kbhit())/*until pressing any key this loop continues*/
{
/*for rectangle co-ordinates are taken randomly*/
x1=rand()%getmaxx();
x2=rand()%getmaxx();
y1=rand()%getmaxy();
y2=rand()%getmaxy();


if(x1>x2)
{
c=x1;       /* exchange of x1 and x2 when x1 >x2 */
x1=x2;
x2=c;
}
if(y1>y2)
{
c=y1;          /* exchange of y1 and y2 when y1>y2 */
y1=y2;
y2=c;
}
c=rand()%16;

/*to draw rectangle using putpixel*/
for(I=x1;I<=x2;++i)
```

```
{
putpixel(I,y1,c);
delay(1);
}
for(I=y1;I<=y2;++i)
{
putpixel(x2,I,c);
delay(1);
}
for(I=x2;I>=x1;_- i)
{
putpixel(I,y2,c);
delay(1);
}
for(I=y2;I>y1;-I)
{
putpixel(x1,I,c);
delay(1);
}

delay(200);  /* to draw the pixels slowly */
}
getch();
closegraph();                /* closes the graph mode */
}
```

The closed graphical areas can be filled with different fill effects that can be set using setfillstyle () function. The following program illustrates fill effects for the rectangles, which are drawn randomly using putpixel.

```
/* Program to demonstrate rectangles using putpixel and filling them with
different fill effects */
# Include <graphics.h>
# Include <conio.h>
# include <stdlib.h>
# include <dos.h>


void main()
{
int gm,gd= DETECT;
```

```
int x1,x2,y1,y2,c,I;

initgraph(&gd,&gm,"");
while(!kbhit()) /* until pressing any key this loop continues */
 {
/* To draw rectangle co-ordinatinates are taken randomly */
x1=rand()%getmaxx();
x2=rand()%getmaxx();
y1=rand()%getmaxy();
y2=rand()%getmaxy();

if (x1>x2)
{
c=x1;          /* exchange of x1 and x2 when x1 is >x2 */
x1=x2;
x2=c;
}
if(y1>y2)
{
c=y1;          /* exchange of y1 and y2 when y1 is > y2 */
y1=y2;
y2=c;
}
c=rand()%16;
/* for rectangle using putpixel */
for(I=x1 ;i<=x2;++i)
{
putpixel(I,y1,c);
delay (1);
}
for(i=y1;I<=y2;++i)
{
putpixel(x2,I,c);
delay(1);
}
for(i=x2;i>=x1;            ———— i)
{
putpixel(i,y2,c);
delay(1);
}
for(i=y2;I>=y1;              _____ i)
{
```

```
putpixel(x1,i,c);
 delay(1);
}
setfillsytyle(rand()%12, rand()%8); /* setting the random fill     styles and colors
*
floodfill(x1+1,y1+1,c);
delay(200);                         /* to draw the pixels slowly */
}
getch();
closegraph();                       /* closes the graph mode */
}
```

The lines with different lengths and colors are illustrated in the following program.

```
/* Program to demonstrate lines with different colors and co-ordinates */
#include<graphics.h>
#include<conio.h>
#include<stdlib.h>
#include<dos.h>
void main()
{
int gm, gd=DETECT;
int x1,x2,y1,y2,c,I;


initgraph(&gd,&gm,"");
while(kbhit()) /* until pressing any key this loop continues */
{
/* to draw rectangle co-ordinates are taken randomly */
x1=rand()%getmaxx();
x2=rand()%getmaxx();

y1=rand()%getmaxy();
y2=rand()%getmaxy();

setcolor(rand ()%16);  /*to set the line color */
line(x1,y1,x2,y2);     /* to draw the line */
delay(200);                         /* draw the pixels slowly */
}
getch();
```

```
closegraph();                    /*closes the graph mode */
}
```

The viewport is the portion of the screen within the screen. The entire screen is the default viewport. We can make and choose our own viewports according to our requirements. Once the viewport is set the top left co-ordinates of the viewport becomes (0,0) origin and the maximum number of pixels along x-axis and y-axis change according to the size of the view port. Any graphical setting can be unset using graphdefaults() function.

The following program illustrates setting the viewport and clipping the lines. It also sets the different line styles and colors for the lines. The viewport border co-ordinates are taken from the user as input.

```
/* Program to demonstrate viewport, clipping and lines with different colors, line styles and co- ordinates */

#include<graphics.h>
#include<conio.h>
#include<stdlib.h>
#include<dos.h>
#include<stdio.h>

void main()
{
int gm, gd=DETECT;
int x1,x2,y1,y2,c,i;
clrscr();
printf("enter starting co-ordinates of viewport (x1,y1)/n");
scanf("%d%d",&x1,&y1);
printf("enter ending co-ordinates of viewport(x2,y2)/n");
scanf("%d%d",&x2,&y2);

initgraph(&gd,&gm,"");

rectangle(x1,y1,x2,y2);  /*to show the boundary of viewport */
setviewport(x1,y1,x2,y2,1);  /* view port is set and any drawing now onwards must be drawn within the viewport only */

while(1kbhit()) /*until pressing any key this continues */
{
/* Rectangle coordinates are taken randomly */
```

```
    x1=rand()%getmaxx();
    x2=rand()%getmaxx();
    y1=rand()%getmaxy();
    y2=rand()%getmaxy();
    setlinestle(rand()%10, rand()%20);
    setcolor(rand()%16);  /*to set the line color */
    line(x1,y1,x2,y2);  /*to draw the line */
    delay(200);
    }
    getch();
    closegraph();            /*closes the graph mode */
    }
```

In computer graphics using C language, we can also display the text and set the different styles for the texts. The following two programs illustrate this.

```
/* Program to demonstrate text and its setting */

#include<graphics.h>
#include<conio.h>
#include<stdlib.h>
#include<dos.h>
#include<stdio.h>
void main()
{
int gm, gd=DETECT;
initgraph(&gd,&gm,"");

setcolor(5);
settextstyle(4,0,5); /*sets the text style with font, direction and char size
*/
moveto(100,100);   /*takes the CP to 100,100 */
outtext("Bangalore is");

setcolor(4);
settextstyle(3,0,6);
moveto(200,200);
outtext("silicon");

setcolor(1)
```

```
settextstyle(5,0,6);
moveto(300.300);
outtext("Valley");


setcolor(2);
sertextstyle(1,1,5);
outtextxy(150,50,"Bangalore is");

getch();

}
```

The set of pixels make lines and a set of continuous lines make surfaces. The following program demonstrates the creation of surfaces using lines and different colors.

```
/* Program to demonstrate surfaces using lines and colors */

 #include<graphics.h>
#include<conio.h>
#include<dos.h>
#include<alloc.h>
#include<math.h>
void main()
{
int gm, gd=DETECT;
initgraph(&gd,&gm,"");
setviewport(100,100,300,300,0);
for(j=0;j<200;j=j+20)
{
for(i=0;i<=200;++i)
{
if (i%20==0)
setcolor(rand()%16+1);
line(i,j,i,j+20);
}
delay(100);
}
getch();
}
```

Following is a menu driven program that shows different parts of the car.

/* Program to draw a car. The different graphical functions are used to draw different parts of the car */

```
 #include<stdio.h>
#include<graphics.h>

main()
{
int x,y,i,choice;
unsigned int size;
void*car;

int gd=DETECT,gm;
initgraph(&gd, &gm," ");

do
{
cleardevice();
printf("1:BODY OF THE CAR\n");
printf("2:WHEELS OF THE CAR\n");
printf("3:CAR\n");
printf("4:QUIT");
printf("\nEnter your choice\n");
scanf("%d",&choice);
switch(choice)
{
case 1 :  initgraph (&gd,&gm," ");
line(135,300,265,300);
arc(100,300,0,180,35);
line(65,300,65,270);
line(65,270,110,220);
line(110,220,220,220);
line(140,220,140,215);
line(180,220,180,215);
line(175,300,175,220);
line(120,215,200,250);
line(220,220,260,250);
```

```
line(260,250,85,250);
line(260,250,345,275);
arc(300,300,0,180,35);
line(345,300,345,275);
line(335,300,345,300);
getch();
cleardevice();
break;


case 2:     initgraph(&gd,&gm,"");
             circle(100,300,25);
                circle(100,300,13);
                circle(300,300,25);
                circle(300,300,13);
                getch();
             cleardevice();
             break;


case 3:   initgraph (&gd,&gm," ");
            outtextxy(150,40,"MARUTI 800");
            circle(100,300,25);
           circle(100,300,13);
           line(135,300,265,300);
          arc(100,300,0,180,35);
          line(65,300,65,270);
         line(65,270,110,220);
         line(110,220,220,220);

        line(140,220,140,215);
       line(180,220,180,215);
       line(175,300,175,220);
      line(120,215,200,215);
     line(220,220,260,250);
line(260,250,85,250);
line(260,250,345,275);
arc(300,300,0,180,35);
circle(300,300,25);
circle(300,300,13);
line(345,300,345,275);
line(335,300,345,300);
```

```
getch();
cleardevice();
break;


case 4 :  exit(0);
    }
}
while(choice!=4);
getch();
}
```

Unit 4
SIMPLE LINE DRAWING METHODS

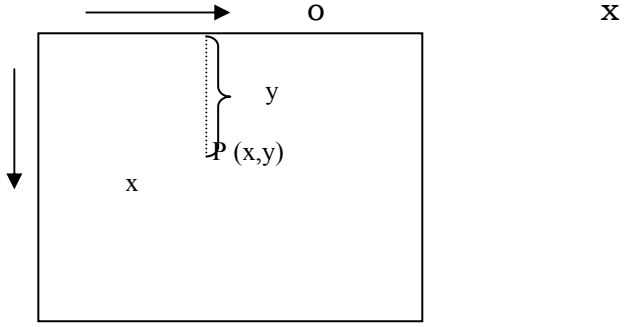
Block Introduction: In this block, you will be introduced the concept of writing pictures on the screen as a set of points. Any picture can be thought of as a combination of points. The idea is to identify the points, which form the part of the picture one is trying to draw and by a suitable technique display these points. To do this, the screen is supposed to be made up of a number of pixels (picture cells), each pixel corresponding to a point. Those pixels, which form a part of the picture being drawn, are made to light up so that the picture is visible on the screen. The trick is to switch on the laser beam (of the CRT) when it is passing over the pixel and switch it off when it is passing over a pixel that does not form a part of the picture.

This block tells us about the techniques of identifying those pixels that should from the part of the picture and the various difficulties that one will have to encounter in the process. Once the pixels are identified, that hardware takes over the question of actually drawing the pictures.

Contents:

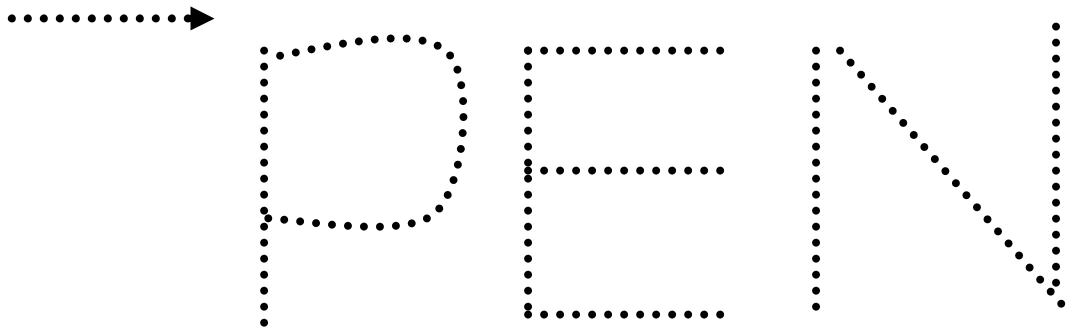
Point Plotting Techniques

The coordinate systems of the monitor: Point plotting techniques are based on the use of Cartesian coordinate system. Each point is addressed by two points (x,y) which indicate the distance of the point with respect to the origin. p (x,y) is pixel at a horizontal distance x and vertical distance y from the origin



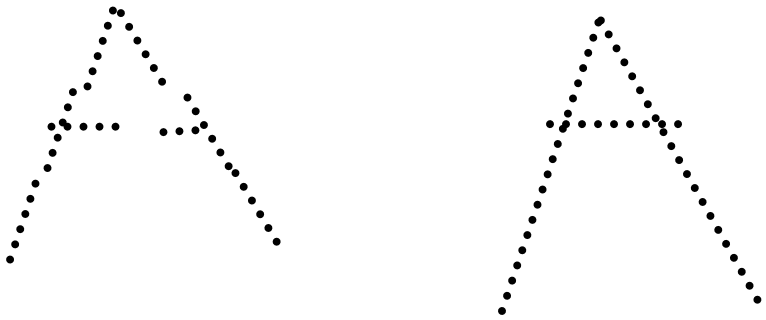
Now any picture to be displayed is to be represented as a combination of points

Examples of point plotted pictures:



Though no continuous lines are drawn but only a sense of points are being made bright, because of the properties of the human eye, we see continuous lines, when the points that are being lighted are fairly close to each other.

In fact, the closer the points to one another, we see better pictures (see the example below)



In the above figure both pictures indicate A, but in the second picture, the points are closer and hence it appears more like A than the first. How many points are there per unit area of the screen indicate what is known as the "resolution" of the monitor. Higher the resolution, we get more number of points and hence better quality pictures can be displayed (As a corollary, such high resolution monitors are costlier)

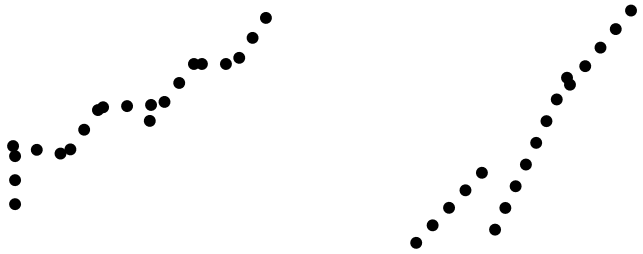
Having surveyed the essentials of hardware of CRT once again we are now in a position to look at the actual process of drawing pictures.

Incremental methods

The concept of incremental methods, as the name suggests, is to draw the picture in stages - incrementally. I.e. from the first point of the picture, we have a method of drawing the second point, from there to the third point etc. They are also sometimes called "iterative methods" because they draw picture in stages - in iterations.

Qualities of good line drawing algorithms: Before we start looking at a few basic line drawing algorithms, we see what are the conditions that they should satisfy. While the same picture can be drawn using several algorithms, some are more desirable than others, because they provide as features that enable us to draw better "quality" pictures. A few of the commonly expected qualities are as follows:

i. **Lines should appear straight:** Often straight lines drawn by the point plotting algorithms do not appear all that straight.



Examples of not so straight lines.

The reason is not far to be searched for. Any point plotting algorithm will give a series of points (x,y) for various values of x and y. In a general case, the values of x and y need not be integers, they can be any real numbers. But on the screen, pixel values are only integers. So, what do we do?. The easy solution is to round off. If two points (successive points) are given say as (6.6, 15.4) and (7.4, 16) when rounded off 6.6 becomes 7 and 15.4 become 15.

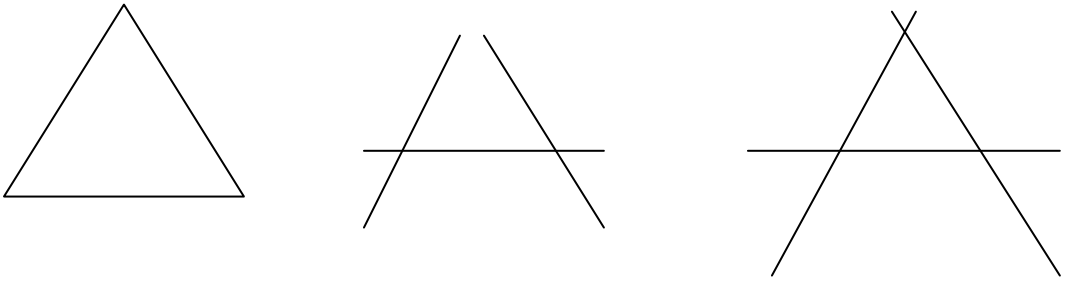
So, the point becomes (7, 15) Similarly the second point will become (7,16). Note that while the difference between 6.6 and 7.4 was 0.8 (almost 1 pixel value) the display shows then as the some point, whereas the

points 15 and 16 are different points. So the segment, instead of appearing as in fig (a) appears as in fig (b)

- (6.6, 15.4)
- (7.4, 16)
- (7,15)
- (7,16)

Note that the slope of the line has changed - A series of such changes between successive points make the lines look as shown in the fig above (Jagged lines)

ii. Lines should terminate accurately: The cause is still the same as in (1). Because of inaccuracies and approximations, the lines do not terminate accurately. Either they stop short of the point at which they should end or extend beyond the points result? Intersections and joints do not form correctly. Look at the examples below



\Required triangle Lines stop short Lines extend beyond

iii. Maintain constant intensity: The pictures are drawn with illumination i.e. a number of point along the line are illuminated. As long as the intensity of these points is uniform, we have a pleasing picture to look at. This can be done if the points to be illuminated are equidistance from one another. However, because of the inaccuracies in the algorithm, we often end up with either dots that are too close or a bit further from each other. Obviously two points, close to each other, when illuminated, make the points look brighter. The result is a line that is brighter in save parts and not so bright in others. The result will be a line that looks jagged and non-uniform.

iv. Lines should be drawn rapidly: This is especially the case in interactive graphics, wherein lines are drawn in real time. While there may not be many problems with straight lines, complex figures may need longer computations before the next pants are identified. Hence the picture is drawn in bits and pieces, which may appear unpleasant or even irritating at times.

Having seen some of the requirements of algorithms, we now see a few practical algorithms to draw simple figures like straight lines or circles.

The Digital Differential Analyzer (DDA)

These include a class of algorithms, which draw lines from their corresponding differential equations. Before we see the actual algorithms, let us see the concept by the example of a simple straight line.

The differential equation of a straight line is given by $\frac{dy}{dx} = \Delta y / \Delta x$

Looked another way, given a point on the straight line, we get the next point by adding x to the x coordinate and y to the y coordinate i.e. given a point p(x,y), we can get the next point as a Q(x+ Δx, y + Δ y) , the next point R as R(x+2* Δx, y+2* Δy)etc. So this is a truly incremental method, where given a starting point we can go on generating points, one after the other each spaced from it's previous points by an additional x and y, until we reach the final point.

Different values of Δx and Δy give us different straight lines.

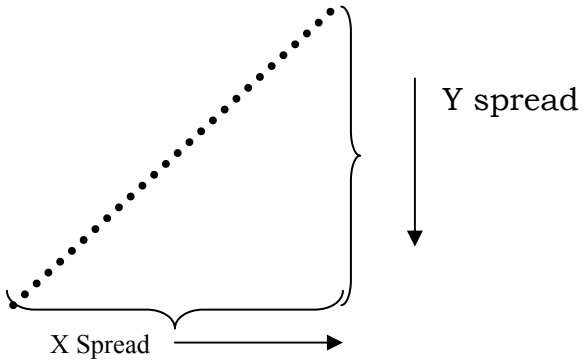
But because of inaccuracies due to rounding off, we seldom get a smooth line, but end up getting lines that are not really perfect.

We now present a simple DDA algorithm in a C like Language.

```
Procedure DDA (x1, y1, x2, y2)
/* The line extends from (x1, y1) to (x2, y2)*/
{
length = abs (x2 - x1);
if length < abs (y2-y1), then length = abs(y2-y1)
x increment = (x2-x1)/length;
y increment = y2-y1)/length;
x=x1+0.5; y=y1+0.5;
for (I=1;I<=length; i++)
{ plot (trun (x), trun (y))
x = x + x increment;
y = y + y increment;
}
}
```

We start from the point (x1,y1) and go up to the point (x2,y2)

The difference (x2-x1) gives the x spread of the line (along the x-axis) and (y2-y1) gives the y spread (along y axis) (x_2,y_2)

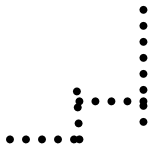


The larger of these is taken as the variable length (not exactly the length of the line)

The variables xincrement and yincrement give the amount of shifts that we should make at a time in each direction. (Note that by dividing by length, we have made one of them equal to unity. If y2-y1 is larger, then yincrement will be unity; otherwise xincrement will be unity. What this means is that in one of the directions, we move by one pixel at a time, while in the other direction, we have to calculate as to whether we have to go to the next valve or should stay in the previous value)

Plot is a function that takes the new values of x and y, truncates then and plots those points. Then we move on to the next valve of x, next value of y, plot it, and so on.

A typical DDA drawn line appears as follows:



Obviously a mean line through these points is the actual line needed.

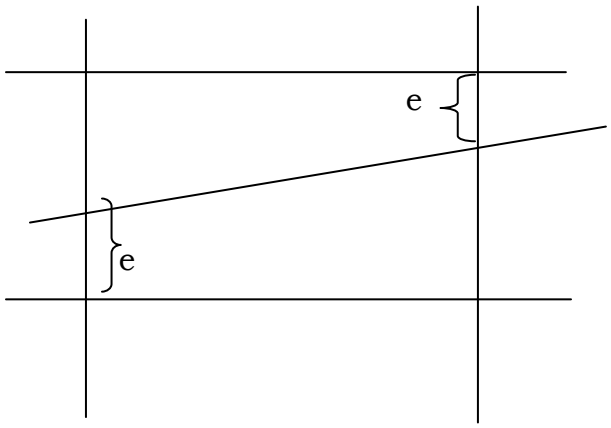
Note that the line looks like a series of steps. This effect is sometimes called a "Stair case" effect.

Now we can explain the program to generate DDA line using C Programming.

```
/*Program to Generate a Line using Digital differential Algorithm(DDA) */
  # include<stdio.h>
# include<conio.h>
# include <graphics.h>
# include<math.h>
/*function for plotting the point and drawing the line */
void ddaline(float x1,float y1,float x2,float y2)
{
int i, color=5;
float x,y, xinc, yinc, steps;
steps=ads(x2-x1);
if (steps<abs(y2-y1);
steps=abs(y2-y1);
xinc=(x2-x1)/steps;
yinc=(y2-y1)/steps;
x=x1;
y=y1;
putpixel((int)x,(int)y,color);
for(i=1;i<=steps; i++)
{
putpixel((int)x,(int)y,color);  /* plots the points with specified color */
x=x+xinc;
y=y=yinc;
}
{
/* The main program that inputs line end point and passes them to ddaline()
function */
void main()
{
 int gd=DETECT,gm,color:
float x1,y1,x2,y2;
printf("\n enter the end points:\n");
scanf("%f %f %f %f",&x1,&y1,&x2,&y2);
clrscr();
initgraph(&gd,&gm, "c:\\tc\\bgi");
ddaline(x1,y1,x2,y2);
getch();
closegraph();
}
```

The main draw back of DDA method is that it generates the line with “stair case” effect. It also needs all parameters as float but C language syntax does not take any floating-point values as co-ordinates in computer graphics.

Bresenham’s algorithm: This algorithm is designed on a very interesting feature of the DDA. At each stage, one of the coordinates changes by a value 1 (That is because we have made either (y2-y1) or x2-x1) equal to length, so that either (x2-x1)/length or (y2-y1)/length will be equal to 1). The other coordinate will either change by 1 or will remain constant. This is because, even though the value should change by a small value, because of the rounding off, the value may or may not be sufficient to take the point to the next level.



Look at the above figure. The left most point should have been at the point indicated by x, but because of rounding off, falls back to the previous pixel. Whereas in the second case, the point still falls below the next level, but because of the rounding off, goes to the next level. In each case, e is the error involved in the process.

So what the Bresenham algorithm does it as follows. In each case adds Δy or Δx as the case may be, to the previous value and finds the difference between the value and the (next) desirable point. This difference is the error e. If the error is >=1, then the point is incremented to the next level and 1 is subtracted from the value. If e is <1, we have not yet reached the point, where we should go to the next point, hence keep the display points unchanged.

We present the algorithm below

```
e=(deeltay/deltax)-0.5;
for(i=1;i=deltax; i++)
{
plot (x ,y);
if e>o then
{
y=y+1;
e=e-1;
}
x=x+1;
e=e+(deltay/ deltax);
}
```

The steps of the algorithm are self-explanatory. After plotting each point, find the error involved, if it is greater than Zero, then in the next step, the next incremental point is to be plotted and error by error-1; else error remains the same and the point will not be incremented. In either case, the other coordinate will be incremented (In this case, it is presented that x - coordinate is uniformly incremented at each stage, while y coordinate is either incremented or retained as such depending on the value of error)

Now we look at the program below that draws the line using Bresneham,s line drawing algorithm.

```
/* Program to generate a line using Bresenham's algorithm */

  # include<stdio.h>
# include<conio.h>
#include<stddlib.h>
# include <graphics.h>

void brline(int,int,int,int);

void main()
{
int gd=DETECT,gm,color:
int x1,y1,x2,y2;
printf("\n enter the starting point x1,y1 :");
scanf("%d%d",&x1,&y1);
```

```
printf("\n enter the starting point x2,y2 :");
scanf("%d%d",&x2,&y2);
clrscr();
initgraph(&gdriver, &gmode,"");
brline(x1,y1,x2,y2);
getch();
closegraph();
}

/* Function for Bresenham's line */

void brline(int x1,int y1,int x2,int y2)
{
int e,l,xend;
int color;
float dx,dy,x,y;
dx=abs(x2-x1);
dy=abs(y2-y1);
if(x1>x2)
{
                x=x2;
y=y2;
                xend=x1;
}
else
{
                x=x1;
                y=y1;
xend=x2;
}
e=2*dy-dx;

while(x<xend)
{
color=random(getmaxcolor());
putpixel((int)x,(int)y,color);
if(e>0)
{
        y++;
e=e+2*(dy-dx);

}
```

```
else
            e=e+2*dy;
            x++;
}
}
```

Generation of Circles

The above algorithms can always be extended to other curves - the only required condition is that we should know the equations of the curve concerned in a differential form. We see a few cases.

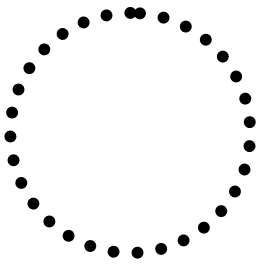
i) A circle generating DDA:

The differential equation of a circle is $\frac{dy}{dx} = -x/y$

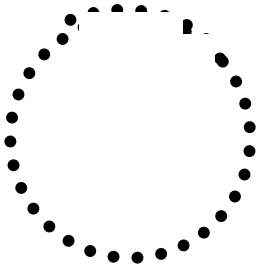
Hence by using the above principle, we can implement the circle plotting DDA by using the following set of equations $x_{n+1} = x_n + \varepsilon y_n$ and $y_{n+1} = y_n - \varepsilon x_n$

Where the subscript n refers to the present value and n+1 to the next value to be computed. Δy and Δx are the increments along the x and y values.

Unfortunately, this method ends up drawing a spiral instead of a circle, because the two ends of a circle do not meet. This is because, at each stage, we move slightly in a direction perpendicular to the radius, instead of strictly along the radius i.e. we keep moving slightly away from the center. So, in the end, we get the closing point a little higher up than where it is required and hence the circle does not close up



Ideal Circle



Drawn by a DDA

However the error can be reduced to a large extent by using the term - x_{n+1} instead of x_n in the second equation.

i.e. $x_{n+1} = x_n + \varepsilon y_n$

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

Another way of drawing circles is by using polar coordinators

$x_{n+1} = x_n \cos\theta + y_n \sin\theta$

$y_{n+1} = y_n \cos\theta - x_n \sin\theta$

Of course, each of them has a few minor disadvantages, which are rectified by special algorithms, discussion of which is beyond the scope of the present course.

Here are the programs for generating circles using DDA and Bresneham's algorithms. Also we have given the program to generate spiral.

```
/* Program to demonstrate circle using DDA algoroyhm */
   # include <graphics.h>
   # include<conio.h>
   # include<dos.h>
   #include<alloc.h>
   #include<math.h>

   void main()
   {
   int gm,gd=DETECT,I,;
   int x,y,x1,y1,j;
   initgraph(&gd,&gm,"");
   x=40;  /*The c0-ordinate values for calculating radius */
   y=40;
   for(i=0;i<=360;i+=10)
   {
            setcolor(i+1);
            x1=x*cos(i*3.142/180)+y*sin(i*3.142/180);
     y1=x*sin(i*3.142/180)-y*cos(I*3.142/180);
     circle(x1+getmaxx()/2,y1+getmaxy()/2,5); /* center of the circle is center
     of the screen*/
        delay(10);
   }
   getch();
   }
```

The following program draws the circle using Bresenham's algorithm.

```
/* program to implement Bresenham's Circle Drawing Algorithm */

  # include<stdio.h>
# include<conio.h>
# include <graphics.h>
# include<math.h>
#include<dos.h>

/* Function for plotting the co-ordinates at four different angles that are placed at egual distences */

void plotpoints(int xcentre, int ycentre,int x,int y)
{
int color=5;
putpixel(xcentre+x,ycevtre+y,color);
putpixel(xcentre+x,ycevtre-y,color);
putpixel(xcentre-x,ycevtre+y,color);
putpixel(xcentre-x,ycevtre-y,color);


putpixel(xcentre+y,ycevtre+x,color);
putpixel(xcentre+y,ycevtre-x,color);
putpixel(xcentre-x,ycevtre+x,color);
putpixel(xcentre-y,ycevtre-x,color);
}

/* Function for calculating the new points for(x,y)co-ordinates. */

void cir(int xcentre, ycentre, int radius)
{
int x,y,p;
x=0; y=radius;
plotpoints(xcentre,ycentre,x,y);
p=1-radius;
while(x<y)
{
if(p<0)
p=p+2*x+1:
else
```

```
{
  y--;
p=p+2*(x-y)+1;
}
x++;
plotpoints xcentre, ycentre,x,y);
delay(100);
}
}
```

/* The main function that takes (x,y) and 'r' the radius from keyboard and activates other functions for drawing the circle */

```
void main()

{
intgd=DETECT,gm,xcentre=200,ycentre=150,redius=5;
printf("\n enter the center points and radius  :\n");
scanf("%d%d%d", &xcentre, &ycentre, &radius);
clrscr();
initgraph(&gd,&gm,"");
putpixel(xcentre,ycentre,5);
cir(xcentre,ycentre,redius);
getch();
closegraph();
}
```

Bresenham specified the algorithm for drawing the ellipse using mid point method. This is illustrated in the following program.

/* BBRESENHAM's MIDPOINT ELLIPSE ALGOTITHM. */

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

# include <graphics.h>
int xcentre, ycentre, rx, ry;
int p,px,py,x,y,rx2,ry2,tworx2,twory2;
void drawelipse();
void main()
{
```

```
int gd=3,gm=1;
clscr();
initgraph(&gd,&gm,"");
printf("n Enter X center value: ");
scanf("%d",&xcentre);
printf("n Enter Y center value: ");
scanf("%d",&ycentre);
printf("n Enter X redius  value: ");
scanf("%d",&rx);
printf("n Enter Y redius  value: ");
scanf("%d",&ry);
cleardevice();
ry2=ry*ry;
rx2=rx*rx;
twory2=2*ry2;
tworx2=2*rx2;

/* REGION first */
x=0;
y=ry;
drawelipse();

p=(ry2-rx2*ry+(0.25*rx2));
px=0;

py=tworx2*y;
while(px<py)
{
x++;
px=px+twory2;
if(p>=0)
{

      y=y-1;
      py=py-tworx2;
  }
   if(p<0)
       p=p+ry2+px;
   else
          {

    p=p+ry2+px-py;
```

```
     drawelipse();
      }
}

/*REGION  second*/
   p=(ry2*(x+0.5)*(x+0.5)+rx2*(y-1)*(y-1)-rx2*ry2);
    while(y>0)
    {
      y=y-1:
       py=py-tworx2;
       if(p<=0)
       {
        x++;
        px =px + twory2;
}
if(p >0)
      p=p+rx2-py;
else
{
     p=p+rx2-py+px;
     drawelipse();
     }
}
  getch();
  closegraph();
}
void drawelipse()
{
Putpixel (xcenter +x, ycenter +y, BROWN);
putpixel (xcenter +x, ycenter +y, BROWN);
putpixel (xcenter +x, ycenter +y, BROWN);
putpixel (xcenter +x, ycenter +y, BROWN);
}
```

The following program demonstrates the generation of spiral.

```
/* Program to demonstrate spiral */
# include <graphics.h>
    # include<conio.h>
# include<dos.h>
#include<alloc.h>
#include<math.h>
```

```
void main()
{
int gm,gd=DETECT;
float x,y,x1,y1,i;
initgraph(&gd,&gm,"");
x=100;
y=100;
for(i=0;i<=360;i+=0.005)
{
        x=x*cos(i*3.142/180)+y*sin(i*3.142/180);
y=x*sin(i*3.142/180)+y*cos(i*3.142/180);
        putpixel((int)x+200,(int)y+200,15);
}
getch();
}
```

Block Summary: In this block, you were introduced to the concept of point plotting. I.e. drawing the curser point by point. The concept of pixels in the monitor helps you to calculate the points that form the line (or curve) and these points can be illuminated giving the picture.

However, because of rounding off errors, certain inaccuracies are introduced in the pictures so drawn, like non uniform slopes, non uniform illumination and in accurate terminations.

The concept of differential analyzer algorithms was introduced - The algorithms which draw the lines based on the Bresenham algorithms were discussed in detail, while the circle generation algorithms were also introduced.

Review Questions:

1. Higher the resolution, better will be the quality of pictures because the _____ will be closer.

2. An algorithm that draws the next point based on the previous paut's location is called _____.

3. The appearance of stair case effect in drawing straight lines is be cause of _____ of mathematical calculations.

4. DDA stands for _____

5. The algorithm that ensures a movement of 1 unit at a time in either x or y direction is the _____ algorithm.

6. The common difficulty in drawing circles using DDA method with it's differential equation is that _____.

7. One method to overcome the above problem is to use _____ equation.

Answers

1. Pixels

2. Incremental method

3. Approximation

4. Digital Differential Analyzer

5. Bresenham's

6. The ends do not meet

7. Parametric equations.

UNIT 5
TWO DIMENSIONAL TRANSFORMATIONS

5.1 Introduction

In this unit, you are introduced to the basics of pictures transformations. Graphics is as much about the concept of creating pictures as also about making modifications in them. Most often, it is not changing the pictures altogether, but about making "transformation" in them. Like shifting the same picture to some other place on the screen, or increasing or decreasing it's size (this can be in one or two directions) or rotating the picture at various angles - The rotation also can be either w.r.t. the original x, y coordinates or with any other axis. All these are essentially mathematical operations. We view points (and hence pictures, which are nothing but the collections of points) as matrices and try to transform them by doing mathematical operations on them. These operations yield the new pixel values, which, when displayed on the CRT give the transformed picture.

5.2 What is transformation?

In the previous unit, we have seen the concept of producing pictures, given their equations. Though we talked of generating only straight lines and circles, needless to say similar procedures can be adopted for the other more complex figures - in many cases a complex picture can always be treated as a combination of straight line, circles, ellipse etc., and if we are able to generate these basic figures, we can also generate combinations of them. Once we have drawn these pictures, the need arises to transform these pictures. We are not essentially modifying the pictures, but a picture in the center of the screen needs to be shifted to the top left hand corner, say, or a

picture needs to be increased to twice it's size or a picture is to be turned through 90^0. In all these cases, it is possible to view the new picture as really a new one and use algorithms to draw them, but a better method is, given their present form, try to get their new counter parts by operating on the existing data. This concept is called transformation.

5.3 Matrix representation of points

Before we start discussing about the actual transformations, we would go through the concept of representation of points. Once we know how to unambiguously represent a pant, we will be able to represent all other possible pictures.

Normally, we represent a point by two values in a rectangular coordinate systems as (x,y). x represents the distance of the point from the origin in the horizontal direction and y in the vertical directions. Negative values are intended to represent movement in the reverse direction (on a CRT screen, however, negative valued pixels can not be represented).

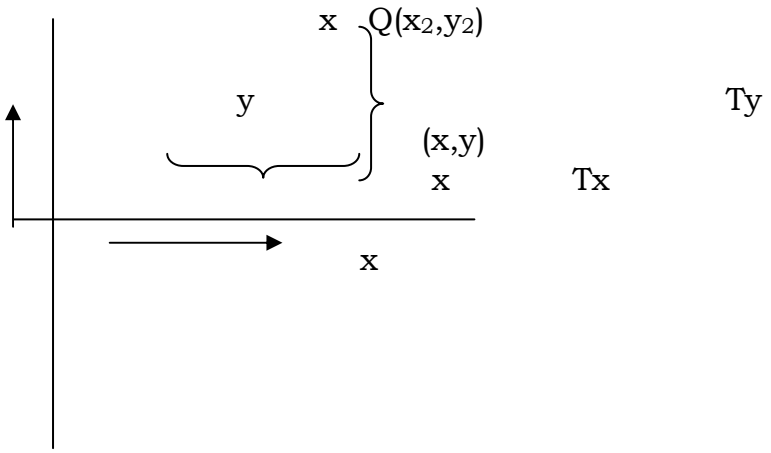
However, in the context of graphics we tend to represent a point as a 3 valued entity [x y 1] where x and y are the coordinates and 1 is just added to the representation. But use of this additional value becomes significant shortly.

5.4 The basic Transformation

Now we are ready to probe into the basics of transformations. As indicated earlier, we talk about transforming points, throughout the discussions, but any complex picture can be transferred using similar techniques in succession.

The three basic transformations are (i) Translation (ii) rotation and (iii) scaling. Translation refers to the shifting of a point to some other place, whose distance with regard to the present point is known. Rotation as the name suggests is to rotate a point about an axis. The axis can be any of the coordinates or simply any other specified line also. Scaling is the concept of increasing (or decreasing) the size of a picture. (in one or in either directions. When it is done in both directions, the increase or decrease in both directions need not be same) To change the size of the picture, we increase or decrease the distance between the end points of the picture and also change the intermediate points are per requirements;

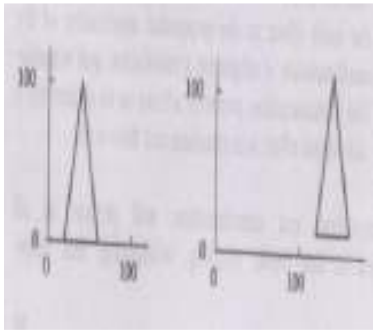
5.5 Translation



Consider a point P(x_1, y_1) to be translated to another point Q(x_2, y_2). If we know the point value (x_2, y_2) we can directly shift to Q by displaying the pixel (x_2, y_2). On the other hand, suppose we only know that we want to shift by a distance of Tx along x axis and Ty along Y axis. Then obviously the coordinates can be derived by $x_2 = x_1 + T_x$ and $Y_2 = y_1 + T_y$.

Suppose we want to shift a triangle with coordinates at A(20,10), B(30,100 and C(40,70). The shifting to be done by 20 units along x axis and 10 units along y axis. Then the new triangle will be at A^1 (20+20, 10+10) B^1 (30+20, 10+10) C^1 (40+20, 70+10)

In the matrix form $[x_2\ y_2\ 1] = [x_1\ y_1\ 1] * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx & Ty & 1 \end{bmatrix}$



5.6Rotation

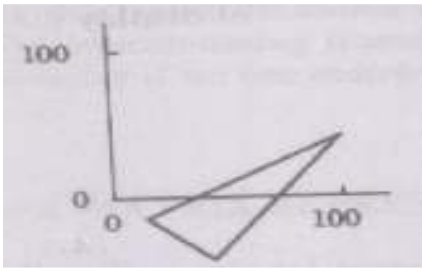
Suppose we want to rotate a point (x_1 y_1) clockwise through an angle/ about the origin of the coordinate system. Then mathematically we can show that

$$x_2 = x_1\cos\theta + y_1\sin\theta \quad \text{and}$$
$$y_2 = x_1\sin\theta - y_1\cos\theta$$

These equations become applicable only if the rotation is about the origin.

In the matrix for $[x_2\ y_2\ 1] = [x_1\ y_1\ 1]$ *

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

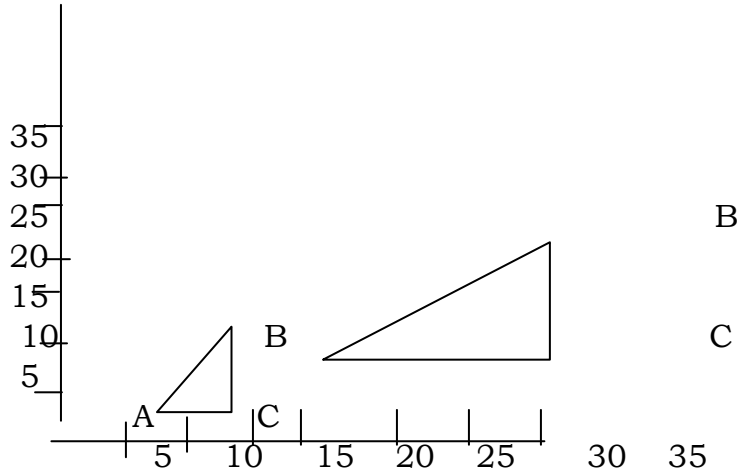


5.7 Scaling

Suppose we want the point (x_1 y_1) to be scaled by a factor s_x and by a factor s_y along y direction.

Then the new coordinates become: $x_2 = x_1 * s_x$ and $y2 = y_1 * s_y$

(Note that scaling a point physically means shifting a point away. It does not magnify the point. But when a picture is scaled, each of the points is scaled differently and hence the dimension of the picture changes.)



For example consider a Triangle formed by the points A (5,5), B(10,10) and C (10,5). Suppose we scale it by a factor of 3 along x-axis and 2 along y-axis.

Then the new points will A(5 * 3, 5 * 2)
B(10*3, 10*2) and
C(10*3,5*2)

In the matrix form we get

$$[x_2 \quad y_1 \quad 1 \ ] = [x_1 \ y_1 \quad 1 \ ] \quad * \quad \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

5.8 Concentration of the operations

Normally, one will not be satisfied with a single scaling or rotation or translation operation, but we will be doing a sequence of such operations. We may translate by a factor, scale by some other factor, translate a second time by other factor, rotate . . . Etc. In such cases, we can simply represent the situation by a sequence of matrix operations. The only constraint is that we should not change the order of operations. Suppose O_1 is the first operation, O_2 is the second operation, O_3 the third etc. Then the final point will be simply

[P2] = [P1] [O1] [O2] [O3] . . . . .

Where [p1] is the original point in matrix form

[P2] is the new point (got after the transformations)

[O1] [O2] … are the respective operations in matrix form.

In fact, we can also undo some of the operations, if need be, by simply taking up the converse operations like inversing. In effect, we will be bringing the computations into the realm of matrix operations, where all the rules of matrix arithmetic become applicable.

5.9 Rotation about an arbitrary point

Note that our rotation formula described previously is about rotating the point w.r.t. the origin. But most often we do want to rotate our pictures about points other than the origin, like say the center of the picture we are talking of, or one of it's vertices or may be a point on a neighboring picture. In this concluding section on transformations, we perform the operation of rotating a point (x1, y1) about another arbitrary point (Rx, Ry).

This is also supposed to provide you an insight about the ease with which matrix representation of operations allows us to Performa a sequence of operations.

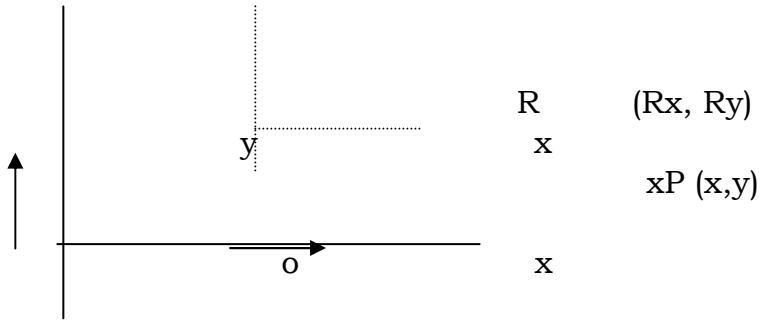
First, how do we sequence the operations in the case?

Since we know how

i) to translate any point to any other point

ii) to rotate it by any angle w.r.t. the origin and

iii) to scale a point, we should be able to combine these operations to do the required transformation.



Now if we look at the figure, we know how to rotate the point w.r.t. O, the origin, whose coordinates are (0,0). But we should rotate it about the point R(Rs, Ry). Looking other way, we could have rotated P about R, if the coordinates of R were (0,0), or if we make the coordinates of R as (0,0). We can make the coordinates of R as (0,0) if we shift the rigin to R, (as shown by dotted lines). If we do that, then we can rotate P about R. But we can shift the origin to R (or shift R to the origin, say) and make corresponding adjustments in the coordinates of P. (In practice, we simply evaluate what would be the value of (x1, y1), if instead o, R were the origin and we start measuring from R. This could be easily done by subtracting the difference of x and y values of R and origin (i.e. Rx & Ry) from the coordinates of (x, y). The new values, x_1^1 and y_1^1 referee to the coordinates w.r.t. R. Now since R is the origin, we know the formula for rotation P w.r.t. R by an angle. We do the operation and get the picture.

But the only hitch is that the whole sequence is about the point R, but it should have been w.r.t. origin. So, to get the desired picture from this, shift the origin back to O. Then we get the desired picture.

Now we can list the sequence of operations as follows.

i) Shift the origin to (Rx, Ry) from (0,0)

Using the matrix $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -Rx & -Ry & 0 \end{bmatrix}$

ii) Rotate the point P(x1, y1) w.r.t. the (new) origin by

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

iii) Shift the origin back to (0,0) by $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Rx & Ry & 1 \end{bmatrix}$

Hence the required point is

$$[x_2, y_2 \ 1] = [x_1 \ y_1 \ 1] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -Rx & -Ry & 0 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Rx & Ry & 1 \end{bmatrix}$$

Review Questions:

1. If a point (x,y) is moved to a point which is at a distance of Tx along x axis what is it's new position?

2. If a point (x,y) is moved to a point which is at a distance Ty along y axis, what is it's new position.

3. If a point (x,y) is rotated anticlockwise through an angle about the origin, what are it's new coordinates.

4. Write the equation for scaling transformations.

5. How many values does the matrix representation of a point (x,y) has ? What are they?

6. Give the matrix formulations for transforming a point (x,y) to (x1, y1) by translation

7. A point (x,y) is to be moved through an angle clockwise about a point (px, py). What is the sequence of operations?

Answers

1. (x+Tx, y)

2. (x, y+Ty)

3. (xcos(- θ) + y sin (-θ), - xsin (-θ) + ycos (θ)

= (xcosθ - ysinθ , xsinθ + ycos (θ)

4. x1=xsx, y1=ysy

5. 3 values (x y 1)

6. $[\ x^1 \quad y^1 \quad 1] = [x \ \ y \ \ 1] \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx & Ty & 1 \end{pmatrix}$

7. Translate (px, py) to origin, effect the rotation Translate the point back to it's original position.

UNIT 6
CLIPPING AND WINDOWING

6.1 Introduction

In this unit, you are introduced to the concepts of handling pictures that are larger than the available display screen size, since any part of the picture that lies beyond the confines of the screen cannot be displayed. We compare the screen to a window, which allows us to view only that portion of the scene outside, as the limits of the window would permit. Any portion beyond that gets simply blocked out. But in graphics, this "blocking out " is to be done by algorithms that decide point beyond which the picture should not be shown. This concept is called clipping. Thus, we are "clipping" a picture so that it becomes viewable on a "window".

The other related concept is the windowing transformation. It is not always necessary that you clip off the larger parts of the picture. You may resolve to zoom it to lower sizes and still present the whole picture. This concept is called windowing. Here you are not cutting off the parts beyond the screen size, but are trying to prepare them to a size where they become displayable on the screen. Again, such a "prepared" picture need not occupy the complete window. In fact, it may be possible for you to divide the screen into 2 or more windows, each showing a different picture. Then, the pictures will be "prepared" to be "fitted" not to the entire screen, but to their respective windows. Since all these operations are done at run time, it is necessary that the algorithms will have to be very fast and efficient.

6.2 Need for Clipping and Windowing

The size of a CRT terminal on which the pictures are displayed is limited – both the physical dimensions and it's resolution. The physical dimensions limit the maximum size of the picture that can be displayed on

the screen and the resolution (no. of pixels/inch) limits the amount of district details that can be shown.

Suppose the size of the picture to be shown is bigger than the size of the screen, then obviously only a portion of the picture can be displayed. The context is similar to that of viewing a scene outside the window. While the scene outside is quite large, a window will allow you to see only that portion of the scene as can be visible from the window – the latter is limited by the size of the window.

Similarly if we presume that the screen, which allows us to see the pictures as a window, then any picture whose parts lie outside the limits of the window cannot be shown and for algorithmic purposes, they have to be "clipped". Note that clipping does not become necessary only when we have a picture larger than the window size. Even if we have a smaller picture, because it is lying in one corner of the window, parts of it may tend to lie outside or a picture within the limits of the screen may go (partly or fully) outside the window limits, because of transformation done on them. And what is normally not appreciated is that as result of transformation, parts, which were previously outside the window limits, may come within limits as well. Hence, in most cases, after each operation an the pictures, it becomes necessary to check whether the picture lies within the limits of the screen and if not, too decide as to where exactly does it reach the limits of the window and clip it at that point. Further, since it is a regular operation in interactive graphics, the algorithms to do this will have to be pretty fast and efficient.

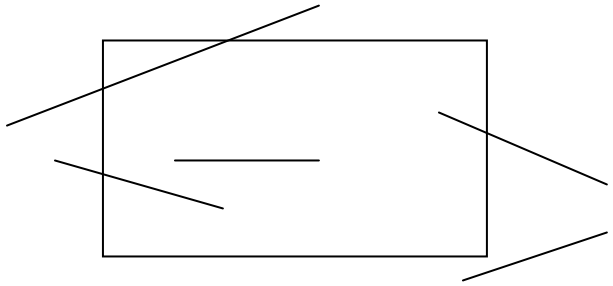
The other related concept is windowing. It is not always that we cut down the invisible parts of the picture to fit it into the window. The alternate option is to scale down the entire picture to fit it into the window size i.e. instead of showing only a part of the picture, it's dimensions can be zoomed down. In fact, the window can be conceptually divided into more than one window and a different picture can be displayed in each window, each of them "prepared" to fit into the window.

In a most general case, one may partly clip a picture and partly transform it by windowing operation. Also, since the clipped out parts cannot be discarded by the algorithm, the system should be able to keep track of every window and the status of every picture in each of them and keep making changes as required all in real time.

Having seen what clipping and windowing is all about; we straightaway introduce you to a few clipping and windowing algorithms.

6.3 A Line Clipping Algorithm

Look at the following set of lines. The rectangle indicates the screen in which they are to be displayed.



How to decide which of the lines, or more precisely which part of every line is to be displayed. The concept is simple. Since we know the coordinates of the screen,

i) Any line whose end points lie within the screen coordinate limits will have to display fully (because we cannot have a straight line whose end points are within the screen and any other middle point in outside).

ii) Any line whose end points lie totally outside the screen coordinates will have to examined to see if any intermediate point is inside the screen boundary.

iii) Any line whose one end point lies inside the boundary will have to be identified.

In case of (ii) and (iii), we should decide up to what point, the line segment can be displayed. Simply finding the intersection of the line with the screen boundary can do this.

Though on paper the concept appears simple, the actual implementation poses sufficient problems. We now see how to find out whether the respective end points lie inside the screen or not. The subsequent sections will inform us about how to go about getting the intersection points.

The Four bit code

001	000	010
001	creen 000	010
101	100	110

Look at the above division of region. Each region is given a code of 4 bits. They are assigned to their values based on the following criterion.

First bit: will be 1 if the point is to the left of the left edge of the screen. (LSB)

Second bit: 1 if the point is to the right of the right edge.

Third bit: is 1 if the point is below the bottom edge and

Fourth bit: is 1 if the point is to the top of the top edge. (MSB)

The conditions can be checked by simply comparing the screen coordinate values with the coordinates of the endpoints of the line.

If for a line, both end points have the bit pattern of 0000, the line can be displayed as it is (trivially).

Otherwise, the pattern of 1's will indicate as to with respect to which particular edge the intersection of the line is to be verified.

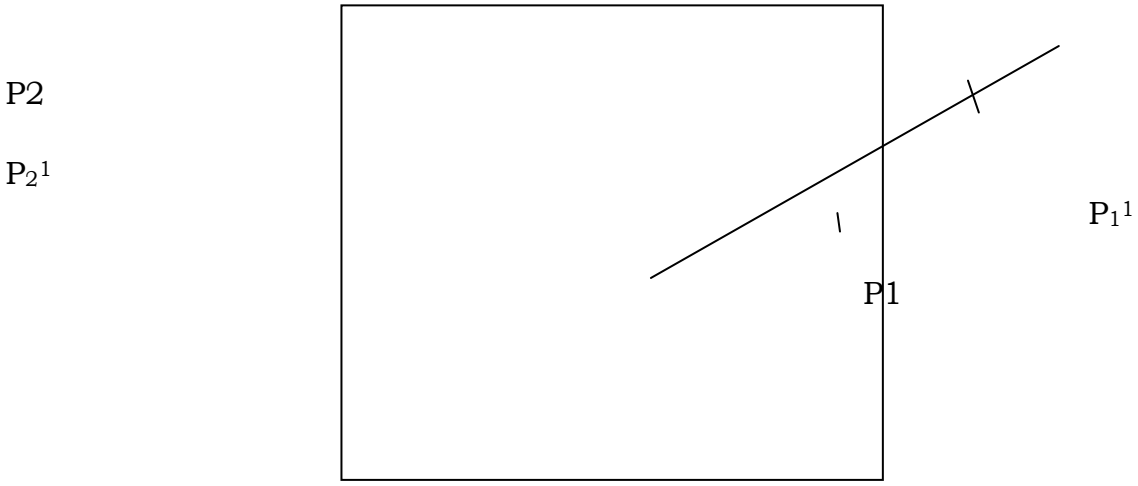
For example if one of the points of a straight line shows 1000, then it's interring section w.r.t. to the top edge needs to be computed (since the point is above the top edge). If for the same line, the other point returns 0010, then since a segment of the line a beyond the right edge, the intersection with the right edge is to be computed.

(The students are encouraged to write a simple algorithm which accepts the end points of a straight line, find out whether it needs any clipping and if so w.r.t. which edges).

Having decided that we need clipping to be done, we look at algorithms that compute the intersections of the line w.r.t. the edges efficiently.

6.4 The Midpoint subdivision method

While mathematical formulae exist to compute the intersection of two straight lines (in this case, the edge under consideration and the straight line to be clipped) it comes computationally intensive and hence another efficient method has been developed. As you know, multiplication and division are most time consuming and hence an algorithm that minimizes on multiplications and divisions is always considered superior. The algorithm in question can be seen to be very efficient in minimizing the divisions that occur, when the intersection of two straight line are computed directly.



Consider a straight line P1 P2. We have decided, (based on the earlier suggested algorithm) that the point $P1^1$ is visible while the point P2 is not. The point is that we should find a point P1 which is the point farthest from P1 and still visible. Any point beyond P^11 becomes invisible. The question is to find P^11.

The algorithm processes by dividing the line P1 P2 at the middle, hence the name mid-point division. Let us say the point is P_1^1 . This point, in this ease is visible. That means the farthest visible point away from P_1^1 · So divide the segment P_1^1 p2 at the middle. In this case, the new mid point P_2^1 is invisible. So the farthest visible point is between P_1^1 P_2^1 . So divide the segment

into two and so on, until you end up at a point that cannot be further divided. The segment P1 to this point is to be visible on the screen.

Now we formally suggest the mid point division algorithm.

6.5 Algorithm mid point subdivision

1. Check whether the line P1 P2 can be trivially included. i.e. when both P1 and P2 are visible. If so exit. Else.

2. Check the point P1, which is visible, and the other point P2, which is invisible.

3. Divide the segment P1 P2 at P_1^1 check if P_1^1 is visible if so, the farthest point is beyond P_1^1, so proceed by dividing P_1^1 P_2 else divide the segment P_1 P_1^1

4. Repeat step (3) until the segment to be divided reduces to a single point. The segment to be displayed is bound by P_1 and this point.

(Note: if in step2, both P_1 and P_2 are invisible, we have to first divide the line, take a visible point and then repeat the algorithm twice for both the segments)

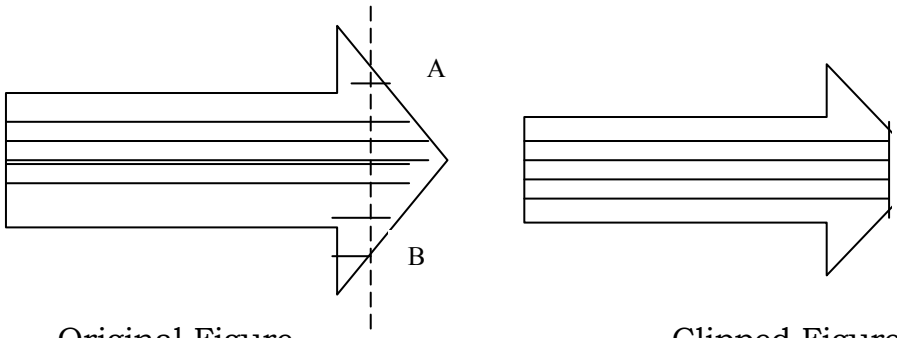
6.1 Other clipping methods

In general, the graphic pictures involve much more straight lines. Curves, if any can be considered as a series of straight lines and each of them can be clipped based on requirements. However, dividing the curve into a series of straight lines may not be very efficient in many cases. One other method is to consider the curves as polygons and use the polygon-clipping algorithm, which will be introduced in the next section.

The other difficulty is about characters. It is normal practice not to clip characters. Either they are shown in full if only a small portion of it is to be clipped; otherwise the entire character is clipped. The normal practice is to divide the character at the middle. If the portion to be clipped lies on the farther side of this middle line, the entire character is deleted otherwise the entire character is seen.

Polygon clipping

A polygon is a closed figure bounded by line segments. While common sense tells us that the figure can be broken into individual lines, each being clipped individually, in certain applications, this method does not work. Look at the following example.



Original Figure Clipped Figure

A

A solid arrow is being displayed. Suppose the screen edge is as shown by dotted lines. After clipping, the polygon becomes opened out at the points A and B. But to ensure that the look of solidly is retained, B uld close the polygon along the line A-

B. This is possible only if we consider the arrow as a polygon – not as several individual lines.

Hence we make use of special polygon clipping algorithms – the most celebrated of them is proposed by Sutherland and Hodgeman.

6.2 Sutherland - Hodgeman algorithm

The basis of the Sutherland Hodgeman algorithms is that it is relatively easy to clip a polygon against one single edge of the screen at a time i.e. given a complete polygon, clip the entire polygon against one edge and take the resultant polygon to clip against a second edge and so on until all the four edges are covered. At first sight, it looks like a rather simplistic and too obvious a solution, but when put in practice this has been found to be extremely efficient.

An algorithm can be represented by a set of vertices v_1, v_2, v_3 ------- ------- v_n which means there is an edge from v_1 to v_2, v_2 to v_3 . . . . v_n to v_1 (we consider only closed polygons and even after clipping would like to have closed polygons, the only difference being that the edges of the screen make for some of the edges of the newly formed, clipped polygon).

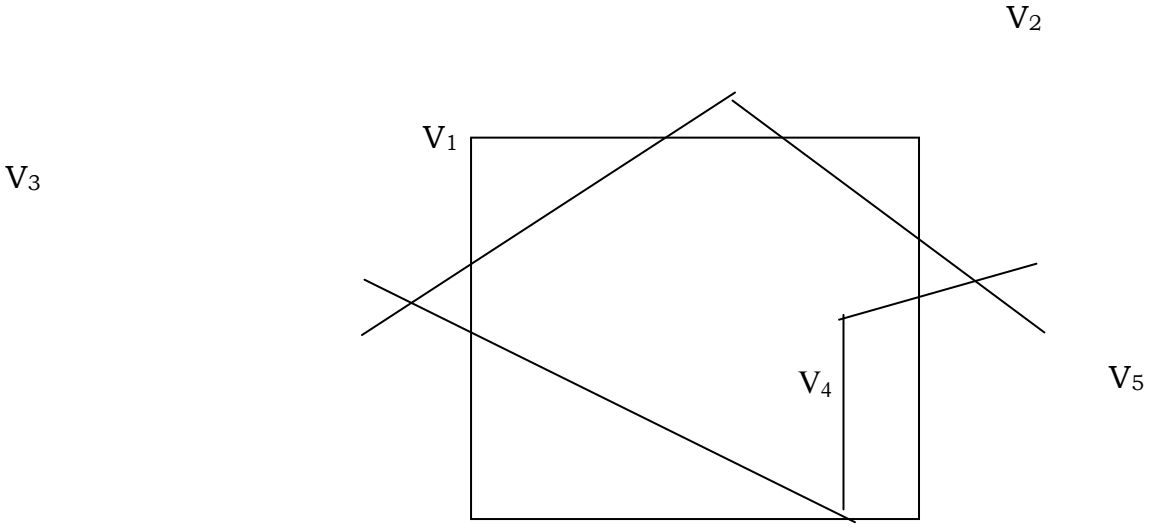
The algorithm tests each vertex v_i (i=1,2 . . . . .n) in succession against a clipping edge e. Now e is an edge of the screen and has two sides. Any vertex lying on one side of the edge will be visible (which we call the visible side). While any other vertex will not be visible if it is on the other side (the invisible side). (For example for the top edge of the screen, any vertex above it is on the invisible side whereas any vertex below it is visible. Similarly for the left edge, any point to it's left is invisible but an edge on it's right is

visible and so on). Now coming back to the algorithm. It tests each vertex of the given polygon in turn against a clipping edge e. Vertices that lie on the visible side of e are included in the output polygon, while those that are on the invisible side are discarded. The next stage is to check whether the vertex v_i (say) lies on the same side of e as it's predecessor v_{i-1}. If it does not, it's intersection with the clipping edge e is to be evaluated and added to the output polygon.

We formally see an algorithm and also the application of the algorithm to a specific example.

Algorithm Sutherland – Hodgeman (v_1, v_2 v_3 . . . . v_n)
For i ⟵ 1 to n do
if (i>1) then begin check whether the line
v[i] V[i-1] intersects the edge e, ifso, compute the intersection
and output the intersection point as the next out put vertex
end;
check always whether vi is on the visible
side of e
if so output vi
if i<n, go to (1)
else
Check whether the line Vn-V1 intersects e
if so, compute the intersection
and output it as the next edge
of the output polygon,
always return.

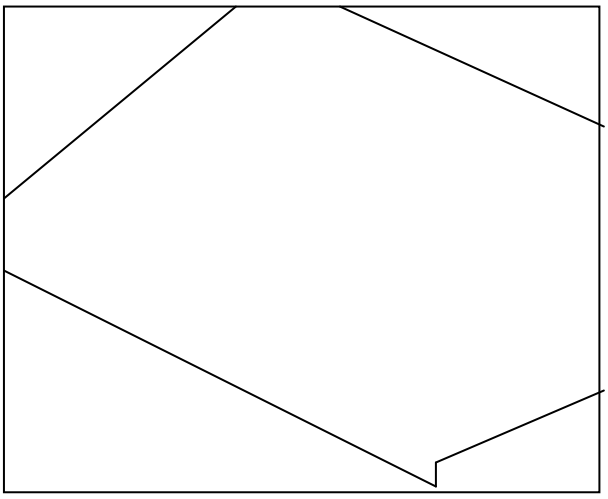
Now to illustrate this algorithm consider the 5 edges polygon below.



Now, let us consider ab as the clipping edge e. Beginning with v_1 the vertex v_1 is an the visible edge of ab so retain it in the output polygon now take the second vertex v_2, the vertices v_1, v_2 are on different side of a. Compute the intersection of V1, V2 let it be i1, add i1 to the output polygon.

Now consider the vertex v_3, v_2 and v_3 are an different sides of ab. Compute the intersection of v_2 and v_3 with ab. Let it be i3. include i3 in the output polygon. Now consider v_3, v_4 and v_5 are all on the same side (visible side) of the polygon, and hence when considered are after the other, they are included in the output polygon straightaway.

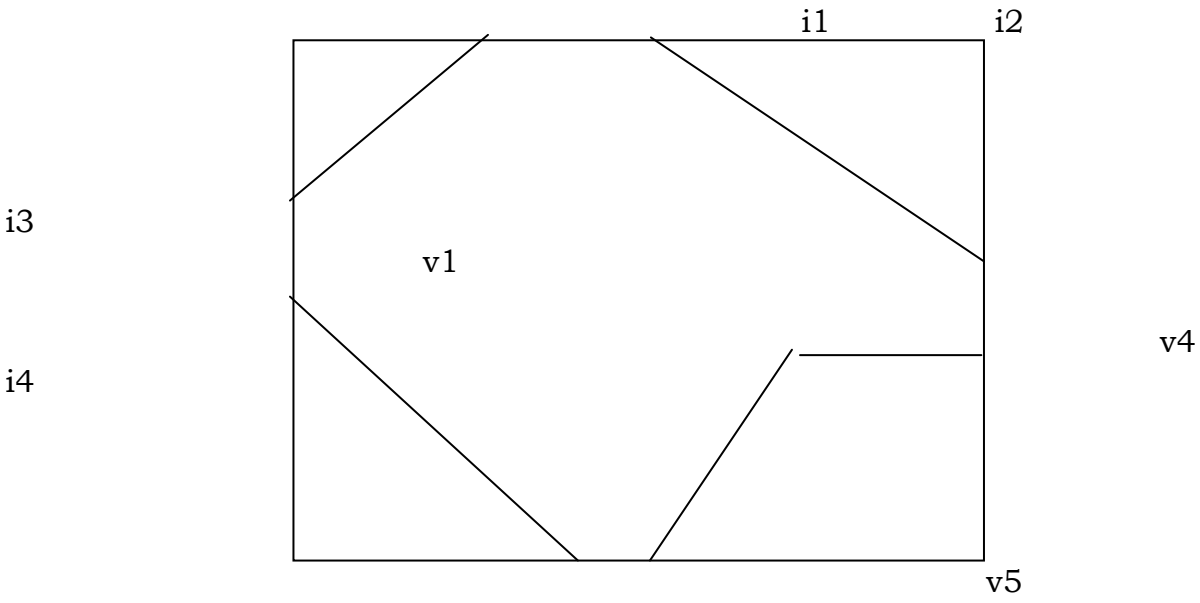
Now the output polygon of stage (1) looks like in the figure below



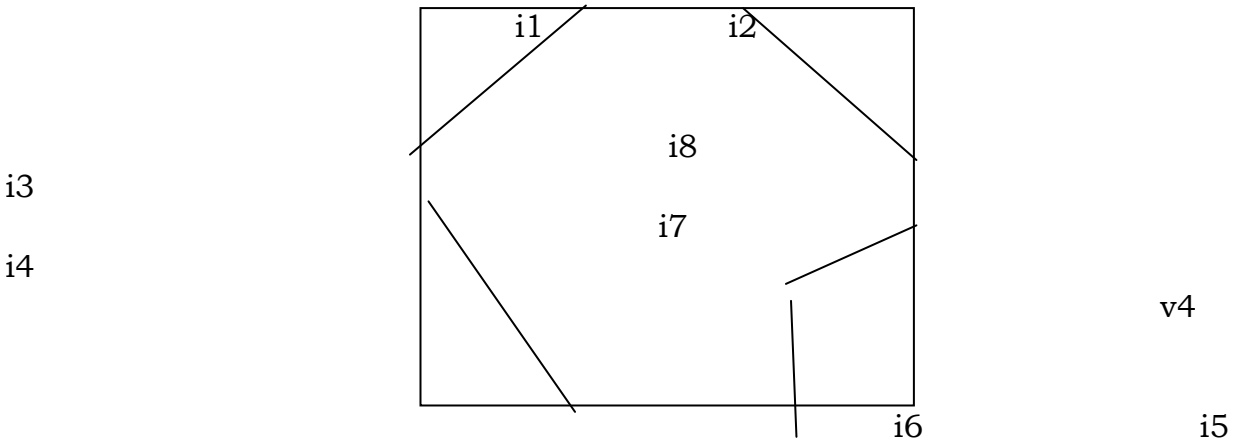
Now repeat the same sequence with respect to the edge b c, for this output polygon of stage (1) v_1, i_1 and iz are on the same side of bc and hence get included in the output polygon of stage (2) since iz and v_3, are the different sides of the line be, the intersection of bc with the line iz is is v_3 computed. Let this point be i_3. Similarly, v_3, v_4 are the different sides of bc, their intersection

with be is computed as i_4, $v_{\backslash 4}$, v_5 are on the same sides of bc and hence pass the test trivially.

Now the output polygon looks like this:



After going through two more clippings against the edges cd and da, the clipped figure looks like the one below



It may be noted that the algorithms works correctly for all sorts of polygons.

6.8 VIEWING TRANSFORMATIONS

Assuming a screen of some size say 1024 x 1200 pixels, this size given the maximum size of the picture that we can represent. But the picture on hand need not always be corresponding to these sizes. Common sense suggests that if the size of the picture to be displayed is larger than the size of the screen, two options are possible (i) clip the picture against the screen edges and display the visible portions. This will need fairly large amount of computations, but in the end, we will be seeing only a portion of the picture. (ii) Scale downs the picture (We have already seen how to enlarge/scale down a point or a set of points by using matrix transformations). This would enable us to see the entire picture, though with a smaller dimensions.

The converse can also be true. If we have a very small picture to be displayed on the screen, we can either display it as it see, thereby seeing only a cramped picture or scale it up so that the entire screen is used to get a better view of the same picture.

However, a picture need not always be presented on the complete screen. More recent applications allow us to see different pictures on the different part of the screen. i.e., the screen is divided into smaller rectangles and each rectangle displays a different picture. Such a situation is encountered when several pictures are being viewed simultaneously either because we want to work on them simultaneously or we want to view several of them for comparison purposes. Now, each of these smaller rectangles that form the space for one such picture is called a “window” and it should be possible for us to open several of these windows at one time and view the pictures. In such a scenario, the problem is still the same: of trying to fit the picture into the rectangle meant for it. i.e. of scaling the picture into it’s window. The only change is that since the window sizes are different for different pictures, we should have a general transformation mechanism that can map a picture of any given size to fit into any window of any given size. Incidentally we call the coordinates system of the picture as the “world coordinate”

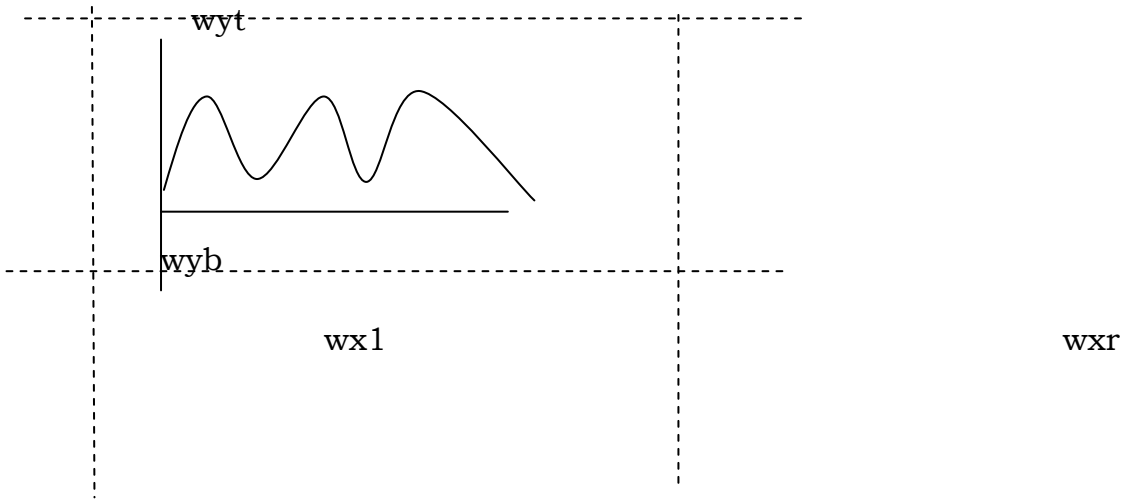
This concept of mapping the points between the two coordinate systems is called the “windowing transformation”

Generally, when the rectangle in which we display the picture on the screen is smaller than the entire screen size, it is called a view port and in such a case the transformation can be called a “View Port Transformation”.

Now we derive a very simple and straightforward method of transforming the world coordinates to the full screen coordinates (or for that matter any window size)

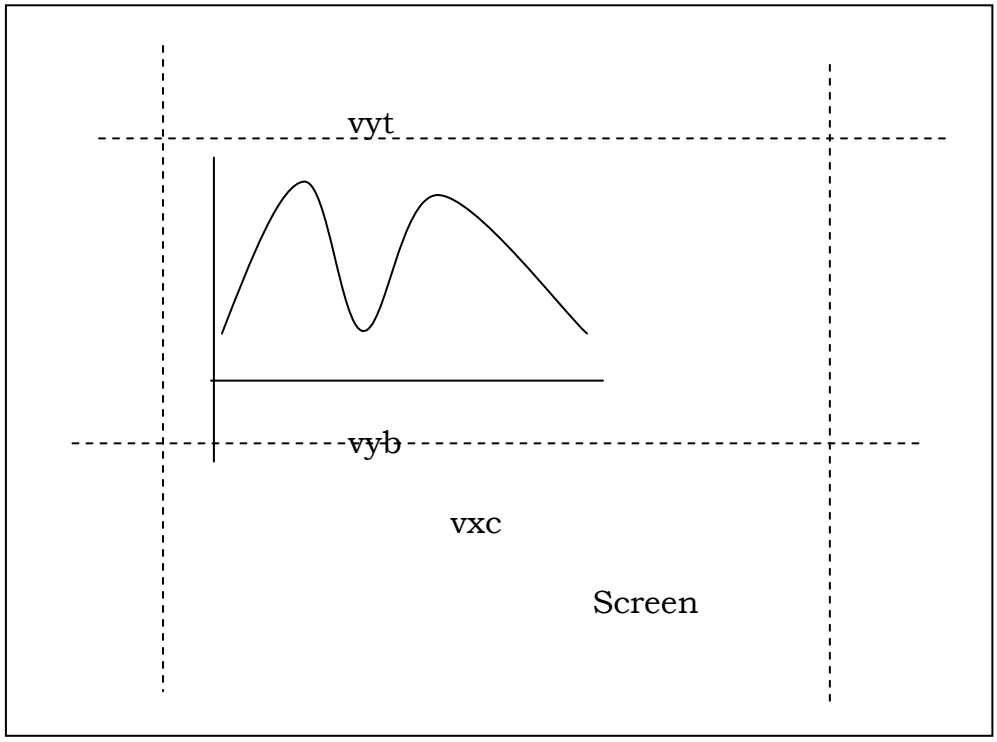
Since different authors use different nomenclatures, in this course, we follow the following conventions. A picture in its normal coordinates system is in the “world Coordinate” system. We are interested only in a part of this picture. That part of picture in which we are interested is called the “window”.

Now we want to transform the portion of the picture that lies within this window to fit into a “viewport”. This view port can be a part of screen or the full screen itself. The following diagrams illustrate the situation and also the various coordinate values that we will be using



The dotted lines indicate the window while the picture is in full lines. The window is bounded by the coordinates w_{x1} and w_{xr} (the x-coordinates on the left side of window and the x – coordinates on the right side of the window) and w_{yt} and w_{yb} (The y- coordinate on the bottom of the window). It is easy to see that these coordinates enclose a window between them (The dotted rectangle of the figure),

We can correspondingly think of a view port rectangle.



The nomenclature is the same as before.

Now consider any point (x_{cw} y_{ws}) on the window. To convert this to the view port coordinates, the following operations are to be done in the sequence.

i) Scale the window coordinates so that the sizes of the window and the view port match. This will ensure that the entire window fits into the view port without leaving blank spaces in the view port. This can be done by simply changing the x and y coordinates in the ratio of the x-size of view port to the x size of window and y – size of view port to y – size of the window respectively

i.e. $\frac{v_{xr} - v_{x1}}{w_{xr} - w_{x1}}$ and $\frac{v_{yt} - v_{yb}}{w_{yt} - w_{yb}}$

It may be noted that in each of the above ratios, the numerator defines the overall space available in the view port and the denominator, the overall space available in the window.

ii) Since the origins of the window and view port need not be coinciding with their world coordinate systems and the screen coordinate system respectively we have to shift them correspondingly. This can be achieved by the following sequence.

a) Before scaling, shift the origin of the window to the origin of the world coordinates.

b) Perform the scaling operation.

c) Shift it back to reflect the difference between the screen origin and view port origin.

Now considering any point (x_w, y_w) to be transformed, we get the following sequence on applying the above sequence of operations.

a) $x_w - w_{x1}$ and $y_w - y_{yb}$

b) $\frac{V_{xy} - V_{x1}}{W_{xr} - W_{xl}} (x_w - w_{x1})$ and $\frac{V_{yt} - V_{yb}}{W_{yt} - W_{yb}} (y_w - w_{yb})$

c) $\frac{V_{xy} - V_{x1}}{W_{xr} - W_{xl}} (x_w - w_{x1}) + V_{x1} \frac{V_{yt} - V_{yb}}{W_{yt} - W_{yb}} (y_w - w_{yb}) + V_{yb}$

The equation in the step c indicates the complete window to view port transformation.

Before closing this section a few observations:

i) It is may not be necessary to transform all the points using the above formula. Regular figures like straight lines or regular curves can be transformed by transforming only their end points.

ii) Since more often than not a transformation from window to view port involves certain portions getting clipped in the process, it is desirable to run a clipping algorithm on the picture w.r.t. the view port so that unnecessary points are not computed only to be thrown off latter.

Review Questions

1. Define Clipping
2. Define Windowing
3. Explain the 4 bit code to define regions used in rejection method.
4. What is the other name of the most popular polygon clipping algorithm?
5. With usual notations, state the equations that transform the window coordinates to screen coordinates.

Answers

1. The process of dividing the picture to it's visible and invisible portions, allowing the invisible portion to be discarded.
2. Specifying an area (or a window) around a picture in world coordinate, so that the contents of the window can be displayed or used otherwise.

3.

1001	1000	1010
0001	Screen 0000	0010
0101	0100	0110

4. Sutherland - Hodgeman algorithm

5. $\frac{V_{xy} - V_{x1}}{W_{xr} - W_{xl}} (x_w - w_{x1}) + v_{x1})$

c) $\frac{V_{yt} - V_{yb}}{W_{yt} - W_{yb}} (y_w - w_{yb}) + V_{yb}$

(x_s, y_s) are the screen coordinates v_{xr} and v_{x1} are the right and left edges of the view port, v_{yt} and v_{yb} are the top and bottom edges of the view port, w_{xr}, w_{x1}, w_{yt} and w_{yb} are the corresponding edges of the window., (xw, yw) are the window coordinates .

UNIT 7
GRAPHICAL INPUT TECHNIQUES

7.1 Introduction

We have familiarized ourselves with many of the interactive input devices. But since the computer expects perfect input values, any errors made in the use of such devices can create problems - like not drawing the lines completely on the tablet or overdrawing it. Similarly, the end points of lines may not appear exactly on a pixel value. One can go on listing such in accuracy's, which would make the computer's understanding of the situation difficult. On the other hand, insisting that one should be able to draw perfectly also is not advisable. Hence, several techniques are available that can cover up for the deficiencies of the input and still make the computer work satisfactorily.

In this block, you will be introduced to various positing techniques using positional constants, concept of modular constraints, ability to draw straight lines interactively using rubber hand techniques selection and the concept of menus. The implementation details, however, are omitted.

7.2 Graphical Input Techniques

We have seen several input devices, which bear resemblance to pens and pencils – example light pens, joysticks etc. To some extent they are intentionally made to resemble the device that the user is familiar with. For example, writing on a pad with a pen like stylus is more convenient for the user. However, there is a basic difference between the targets of such inputs i.e. a material written with a pen are targeted towards the human user, while the graphical input derives are targeted towards the computer. Herein lies the difference. The human can understand variations of input to a large extent. For example the letter A may be written in different ways by different people or for that matter, the same person may write it in different ways at different times. While a human can understand the variations, a computer normally cannot. In other words, the input to human can vary over a range, while the inputs to a computer needs to be precise. Similarly while drawing a circle, if the two ends do not meet properly, a human being can still consider it as a circle, whereas a computer may not. At the same time, training a person to say, precisely write

the letters in the same manner, trial after trial, or to make him draw his graphs to the exact precision would be time consuming. In other words, whereas a common user can be made to be aware of what he wants and would be willing to get it as fast and accurately as possible, making him acquire graphic arts skills would be inexcusable. On the other hand, it is desirable to make the computer understand what he wants to input or alternately, we can make the input devices cover up for the miner lapses of the user and feed a near perfect input to the computer – like making it cover the circle, when the user stops just short of closing it or ends up making the two ends one next to the other. There are several astonishingly simple ways to make the life of the user more comfortable and at the same time improve effectiveness of the input device.

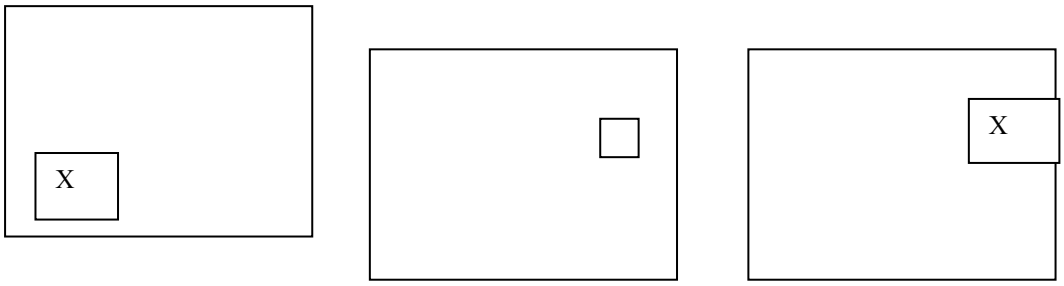
In other words, the graphical input device should not only be influenced by the way it is used, but should also consider other factors like what the user is trying to say or what is the next logical step in the sequence of events and extrapolate or interpolate the same. Of course, some guess work is involved in the process, but most often it should work satisfactorily.

In fact, the very simple concept of cursor is a good example of input technique. It can be thought of as a feedback technique. It indicates the present position of editing / operation. In a more sophisticated case, it can be a "block" of the text / figure selected by blocking. It helps the user to know what he is doing and in fact, ascertains that the function that he is working on is actually working.

However, in this chapter, we look at slightly more sophisticated user-friendly techniques. The algorithms are fairly involved and hence we will only be discussing the details, without going into the implementation details.

7.3 Positioning Techniques

This can be considered the most basic of graphical input operations. In it's simplest form, it involves choosing a symbol / character on the screen and moving it to another location. One way of using it is to choose the symbol or picture involved, moving the cursor to the position required and pressing a (predetermined) key to place in that position.

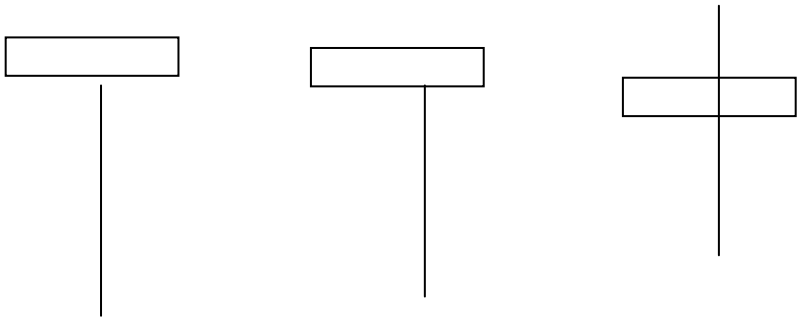


Choose a Symbol Choose a Position End of operation

While in earlier DOS versions, using a combination of pre-selected keys in proper order was doing this operation; the advent of mouse has simplified the matter. The concept of selection, positioning and final movement are all done with the click of buttons.

7.4 Positional Constraints

One of the problems faced by inexperienced users while drawing figures is the concept of positioning. For example, we may want to put an object exactly at the end of a straight-line or a cross at the center of the circle etc. Because of lack of coordination between the eyes and the hand movements, the object may end up either a little away from the line or inside the line as below.



Desired position Away from the end inside the line

Similarly, while locating a center of the circle the cross may get located very near to the center of the circle, but not exactly at the center. In fact, it is easy to appreciate that in the case of putting a rectangle at the end of

the straight lines one may often end up operating between the second and third stages several times before (if at all) successfully reacting the position of (i). One of the methods of helping the user is to put a "construct" on the position of the box. i.e. when the distance between the box and the end of the line is very small, the box automatically aligns itself on the edge of the line. i.e. it is enough if the user brings it to either of the positions (ii) & (iii) and the software automatically aligns it to position (i).

Though we are not considering the implementation aspects of the same, it is easy to note that writing an algorithm for this is fairly straight forward. Assuming each line ends at an integer value of the pixel, if the edge of the base is brought to a value which is a fraction above / below the value, automatically round it off to the pixel value. For example, if the (x,y) values of the end of the lines is say (10,50) and a box is brought to a position say (10.6, 50.7), the values are automatically changed to (10,50), similar being the case if the box position is say (9.7, 49.8). It is easier to see that the first example is the case where the box is slightly above the line and the second where it is inside the line.

This type of putting constraints is often called a "modular constraint". There can be other types of constraints as well. In a certain figure, only horizontal and vertical lines are there, say like in a grid design, any angular lines can be brought into any one of these positions by putting an angular constraint that no straight line can be at any angle other than 0^0 and 90^0. The same can be extended to draw lines at any particular angle.

Now let us go back to the problem of attaching a box to the end of a line. Suppose the end of the line does not terminate always at integer value. Then positional constraints cannot be used. In such cases, we can think of gravity constraints, wherein the box gets attached to the line because of the "gravitational force" of the line i.e. it gets attached to the nearest free point which forms the end of line. Again this relieves the user of the difficulty of exactly putting the box to the end of the line.

7.5 Rubber band techniques

Rubber banding is a very simple, but useful technique for positioning. The user, if he wants to draw a line, say, specifies the end points and as he moves from one point to another, the program displays the line being drawn. The effect is similar to an elastic line being stretched from one point to another and hence the name for the technique. By altering the end points, the position of the line can be modified.

The technique can be extended to draw rectangles, arcs, circles etc. The technique is very useful when figures that pass through several intermediate points are to be drawn. In such cases, just by looking at the end

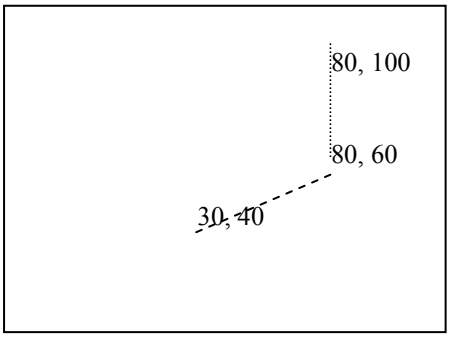
points, it may not be possible to judge the course of the line. Hence, the positioning can be done dynamically, however, rubber band techniques normally demand fairly powerful local processing to ensure that lines are drawn fast enough.

Dragging

As the name suggests, it involves choosing a symbol or a portion of a figure and positioning it at any desired point. It is possible to achieve a accurate and visible results without bothering to know about the actual coordinates involved.

Dimensioning Techniques

It is often desirable to display the coordinate position or the dimensions along with the object. This would be helpful in ascertaining the location of the object, when mere visible accuracy of location may not be enough, but they may have to be positioned w.r.t. the actual coordinate system.



The more difficult problem is that the coordinates need to keep changing as the figure is being dragged around and this demands rapid calculation on the part of the system.

Normally the dimensions are displayed only when the object is being manipulated or moved around and will stay only long enough for the user to take note of them. This ensures that they do not obscure the active parts of the picture, once the completed picture is on display.

Selection of Objects

One of the important points to be addressed is to select parts of the picture for further operations. Once the selection is made properly, tasks like

moving, deletion, copying is whatever can be done. But the actual selection process poses several problems.

The first one is about the choice of coordinates. When a point is randomly chosen at the starting point of the selection process, the system should be able to properly identify its coordinates. The second problem is about how much is being selected. This can be indicated by selecting a number of points around the figure or by enclosing the portion selected by a rectangle. The other method is to use multiple keys i.e. position the cursor at the first point of selection, press certain combination of keys, move the cursor to the final position and again press certain combination of keys, so that the figure lying in between them is selected. The mouse facilitates the same operation by the use of multiple buttons on it. Once the selection is made, normally the system is supposed to display the portion selected so that user can know he has actually selected what he had wanted to. This feed back is done either by changing the color of the screen, modifying the brightness or by blinking.

Menu selection

This is one of the special cases of selection where the user would be able to choose and operate from a set of available commands / figures displayed on the screen. This concept is called the “menu” operation, where you select the item from those available on the menu card. The use of mouse an input technique normally implies menus being provided by the system. The menu concept helps the user to overcome the difficulty of having to draw simple and often used objects by providing them as a part of the system.

Review Question:

Name the type of input facility available to the user in each of the following cases

1. Moving pictures from one place to another.

2. Making a line meet another box accurately.

3. Ending a line exactly a pixel.

4. Drawing a straight line interactively

5. Showing the x, y coordinates of points as the lines are being drawn.

6.. Choosing one out of a number of options.

Answers :

1. Dragging

2.. gravitational constraint

3. Modular constraint

4.. Rubber band technique

5. Dimensioning technique.

6. Menu selection.

UNIT 8
EVENT HANDLING AND INPUT FUNCTIONS

8.1 Introduction
8.2 polling
8.3 event queue
8.4 functions for handling events
8.5 polling task design
8.6 Input functions
8.7 dragging and fixing
8.8 hit detection
8.9 OCR.

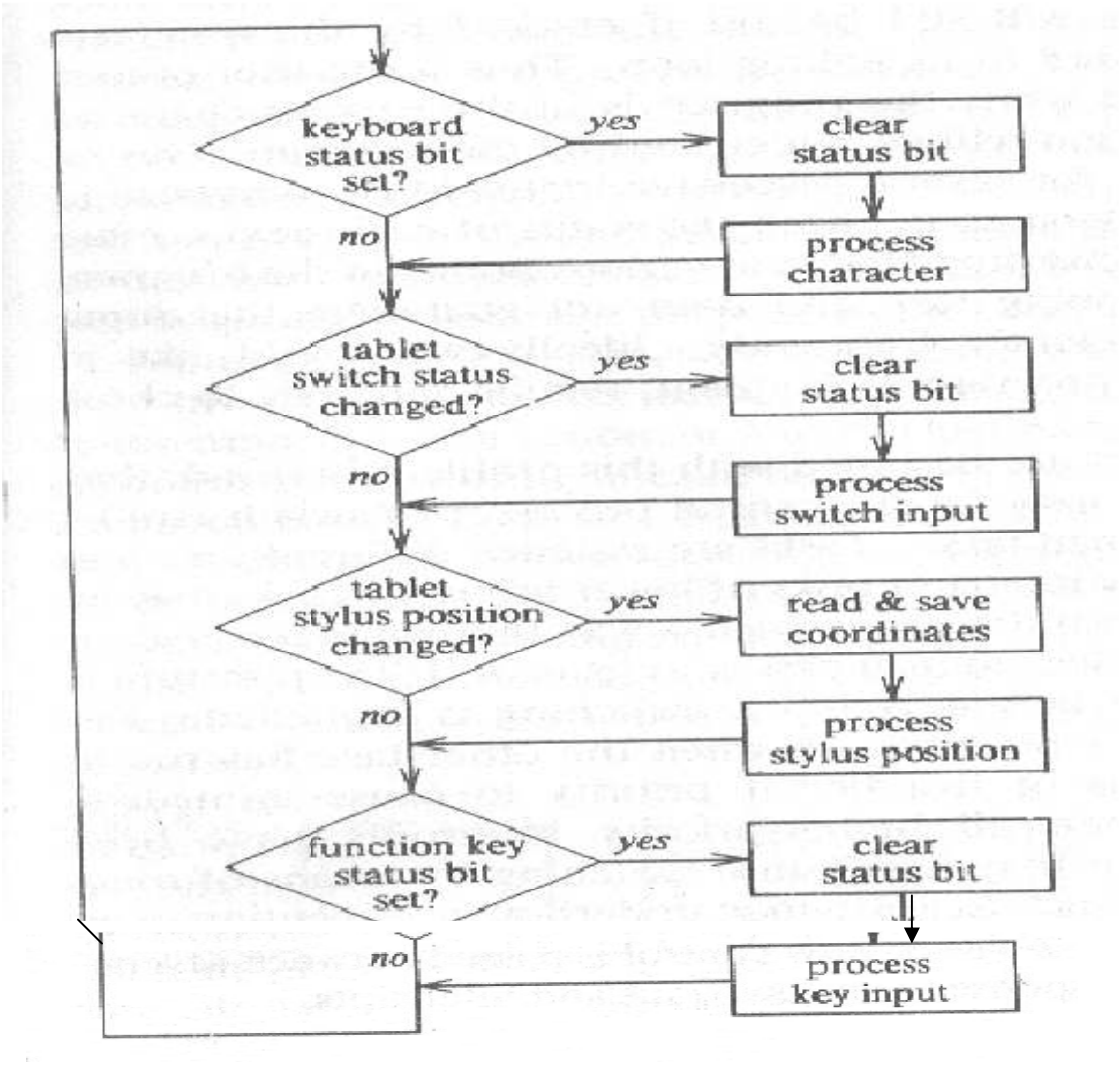
8.1 Introduction

In the last session we have discussed about interactive techniques for inputs. These techniques are varied and many. Some of them are easy to program but some need complicated programming to implement. For this purpose we can provide some of the functions that gives the programmer easy access to the capabilities of input hardware and that protect them from its peculiarities.
For designing the set of input functions we have to think of three factors that makes it difficult. These include

1) The wide variety of techniques the functions must be able to implement.
2) The variations between the input devices.
3) Unpredictable nature of interactive input, designing the input software. Human beings control graphical input devices, not by computers. This sets them apart from conventional input devices like card readers, paper tape reader, magnetic tape drives etc.,.

8.2 Polling

In polling we periodically check the status of each input device connect to the system. A simple polling loop can be used to check the status of each device in a repetitive manner and to pas on to the program the first item of the data received. The following figure shows such an arrangement. Whenever the program receives data, it is processed and generates the fresh display, and then the program returns to the polling loop.



8.3 Event Queue

When the polling task detects a change in the status of the input device, it must be read the input data from the device registers and pass the data to the main program. This may happen at any movement. The main program may be in idle state waiting for input from the user, or it may be in the other computation. In the second situation, the input data must be saved until the computation is complete. For this purpose an 'event queue' must be implemented to store all such unhandled events until main program processes them.

Usually the circular buffer or queue can be used to implement this. This is a vector of memory in which the event blocks are stored contiguously. The polling task maintains a pointer to the tail of the queue, i.e., to the next available position for an event block. The following figure shows an event queue stored in a circular buffer.

8.4 functions for handling events

Mainly two functions namely 'getevent()' and 'permitevent()' . The getevent() function can check the state of the event queue, loops until an event arrives, and then unpacks the contents of the event block. Since the getevent function contains its own idle loop, the main program can call it whenever it requires input from user. The general getevent function for line input can be given as follows.

Type action = (pendown, penmove, penup);
Event=record typ: action: x,y: integer end;

Procedure lineinput:

```
 Var e: event: x0,y0: integer;
Begin
Getevent(e);
If ( e.typ =pendown then begin
X0=e.x;
Y0=e.y;
Getevent(e);
If ( e.typ = pendown ) then begin
Moveto(x0, y0);
Lineto(e.x,e.y)
End
End
End;
```

This program uses a record 'e' of type event to retain the event data and uses a constant 'pendown' to check the event type.

The 'permitevent()' function is used to permit the event to happen. As soon as the main program processes getevent it allows the permitevent function to activate the event. The following program example illustrates the use of this function in programming a rubber-band line;

```
Procedure RubberBand;
Var e: event; x0, y0: integer;
Begin
Repeat ( getevent(e) until e.typ = pendown;
X0 =e.x;
Y0=e.y;
Permitevent(penmove);
Repeat
Getevent(e);
If (e.typ=penmove) then begin
Permitevent(penmove);
Moveto(x0,y0);
Lineto(e.x,e.y);
End
Until e.typ=pendown
End;
```

To implement the permitevent function we must provide the polling task with a set of status bits, one for each event type. When a status bit is set, events of the corresponding type are permitted to occur.

8.5 polling task design

To design the polling task, each input device must be checked for its status. If its status has changed, or if it is capable of generating repetitive events, the appropriate status bit is checked. If the status bit is set, an event block is added to the event queue and status bit will be cleared.

8.6 Input functions

It is difficult for the application programmer to implement everything with just 'getevent' and 'permitevent' functions. So we can have some of the important input functions, which will ease the work of the programmer while implementing the input techniques. These functions are given below.

a) ReadPosition(P)

This functions waits for a pen switch input and then returns the pen's co-ordinates in the two-element record p.

b) Ink (inkarray, length)

This function wait for the user to draw a stroke with the pen and return the stroke trajectory in inkarray (an array of length words), the number of points on the trajectory is returned as the functions value.

c) Recognize()

This function wait for the user to draw a character and then return the code of the character that most closely matches the character drawn.

Three other functions are extremely useful in interactive programming even though they do not operate directly on input devices:

a) HitDetect(x,y)

Returns the name of the segment whose lines pass closest to (x,y) given in screen co-ordinates; if no segment passes within a certain tolerance, the function returns zero.

b) Drag(s)

It causes segment 's' to move with the input-device cursor.

c) Fix(s)

It fixes the event in the particular position and terminates the dragging operation

8.7 ON-LINE CHARACTER RECOGNIZERS (OCR).

These are on of the most important and more widely used devices nowadays. Usually the recognizers contains the following four components:

a) A Tablet-polling routine, which reads the stylus position at regular intervals, applies whatever smoothing is necessary, and builds a list of coordinate pairs representing each stroke of the character.
b) A feature extraction routine, which extracts from each stroke a small number of basic properties or features, that can be used as a basis for recognition.
c) A directory lookup routine, which searches through a dictionary of characters for a set of features matching those of the input strokes.
d) A training routine, used to construct the dictionary.

UNIT 9
CURVES AND SURFACES

9.1 Shape description requirements

Generally shape representations have two different uses, an analytic use and a synthetic use. Representations are used analytically to describe shapes that can be measured; just as a curve can be fitted to a set of data points, a surface can be fitted to the measured properties of some real objects. The requirements of the user and the computer combine to suggest a number of properties that our representations must have. The following are some of the important properties that are used to design the curves. The similar properties can also be used for designing the surfaces as surfaces are made up of curves.

a) **Control Points:** The shape of the curve can be controlled easily with the help of a set of control points, means the points will be marked first and curve will be drawn that intersects each of these points one by one in a particular sequence. The more number of control points makes the curve smoother. The following figure shows the curve with control points.

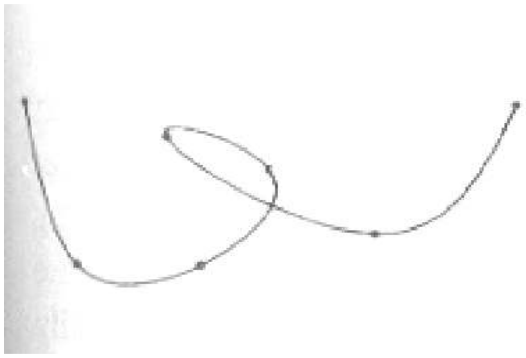


Fig : Control points (Indicated by dots) govern the shape of the curve

b) **Multiple values:** In general any curve is not a graph of single valued function of a coordinate, irrespective the choice of coordinate system. Generally single valued functions of a coordinate make the curves or graphs that are dependent on axis. The following figure shows multi-valued curve with respect to all coordinate systems.

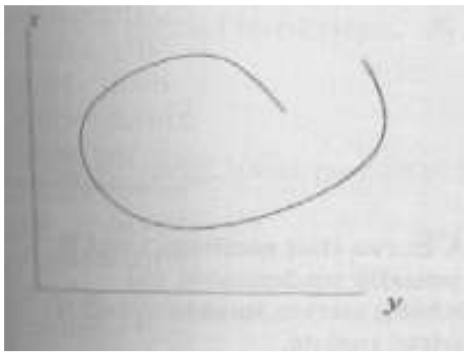


Fig: A curve can be multi-valued with respect to all coordinate systems.

c) **Axis independence:** The shape of an object should not change when the control points are measured in different coordinate systems, that means when an object is rotated to certain angle in any direction (clockwise or anti-clockwise) the shape of the curve should not be affected.

d) **Global or local control:** The control points of a curve must be controlled globally from any function of the same program or it can also be controlled locally by the particular function used to design that curve by calculating the desired control points.

e) **Variation-diminishing property:** Some of the mathematical functions may diminish the curve at particular points and in some other points it may amplify the points. This leads to certain problems for curves appearance at the time of animations, (just as a vehicle looks curved when it is taking turn). This effect must be avoided with the selection of proper mathematical equations with multiple valued functions.

f) **Versatility:** The functions that define the shape of the curve should not be limited to only few varieties of shapes, instead they must provide vide varieties for the designers to make the curves according to their interest.

g) **Order of continuity:** For any complex shapes or curves or surfaces it is essential to maintain continuity in calculating control points. When we are not maintaining the proper continuity of control points it makes a mesh while marking the curve and the complex object.

9.2 Parametric functions:

The dominant form used to model curves and surfaces is the parametric or vector valued function. A point on a curve is represented as a vector:

$$P(u)=[x(u)\ y(u)\ z(u)]$$

For surfaces, two parametric are required:

$$P(u, v)=[(x(u, v)\ y(u, v)\ z(u, v)]$$

As the parametric u and v take on values in a specified range, usually 0 to 1, the parametric functions x, y and z trace out the location of the curve or surface. The parametric functions can themselves take many forms. A single curve be approximated in sever different ways as given below:

$P(u) = [\cos u \sin u]$

$P(u) = [(1 - u^2)/(1 + u^2)\ 2u/(1 + u^2)]$

$P(u) = [u\ (1 - u^2)^{1/2}]$

By using simple parametric functions, we cannot expect the designer to achieve a desirable curve by changing coefficients of parametric polynomial functions or of any other functional form. Instead we must find ways to determine the parametric function from the location of control points that are manipulated by the designer.

9.3 Bezier methods

P. Bezier, a French mathematician designed the equations for design the curves as well as surfaces. This mathematical equations is used in his UNISURF system, has been used in designing several mechanical parts. The following sections describe the curves and surfaces given by Bezier.

9.4 Bezier curves

Bezier defines the curve P(u) in terms of the locations of n + 1 control points pi

$$\mathbf{P(u)=\sum_{i=0}^{n} p_i\ B_{i,n}(u)}$$

Where $\mathbf{B_{i,n}(u)}$ is a blending function

$$\mathbf{B_{i,n}(u)= C(n,i)u^i(1-u)^{n-i}}$$

And C(n,I) is the binomial coefficient. The above equation can be written in terms of x, y and Z axis as follows

$$x(u)=\sum_{i=0}^{n} x_i\, B_{i,n}(u)$$

$$y(u)=\sum_{i=0}^{n} y_i\, B_{i,n}(u)$$

$$z(u)=\sum_{i=0}^{n} z_i\, B_{i,n}(u)$$

Where the three-dimensional location of the control point p_i is $[x_i, y_i, z_i]$.
The Bezier curve methods described above can follow almost all the important properties given in the beginning of this chapter.
The following figure shows the curves drawn using Bezier methods.

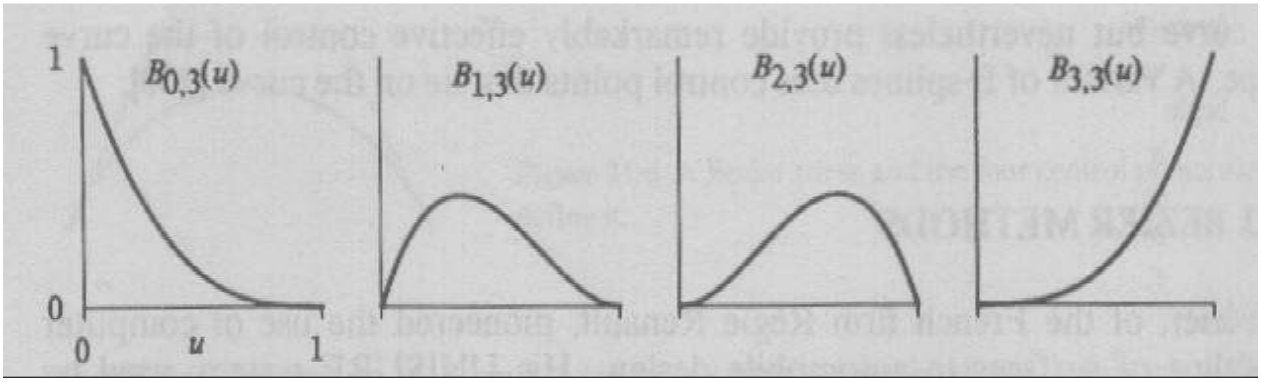


Fig: The curves generated by four Bezier blending functions for n=3

9.5 Bezier surfaces

The formulation of the Bezier curve extends easily to describe three-dimensional surfaces by generating the Cartesian product of two curves. Two similar blending functions are used, one for each parameter:

$$P(u, v)=\sum_{i=0}^{n}\sum_{j=0}^{m} p_{i,j}\, B_{i,n}(u)\, B_{j,m}(v)$$

Figure shows a view of two Bezier surfaces with (n + 1) X (m + 1) control points, arranged in a mesh. Adjacent control points are connected with lines in order to show the mesh. The surface itself is shown by drawing two sets of curves: one set holds the u parameter constant and allows v to range from 0 to 1; the

other set holds v constant and varies u. These curves of constant u and v are in fact Bezier curves.

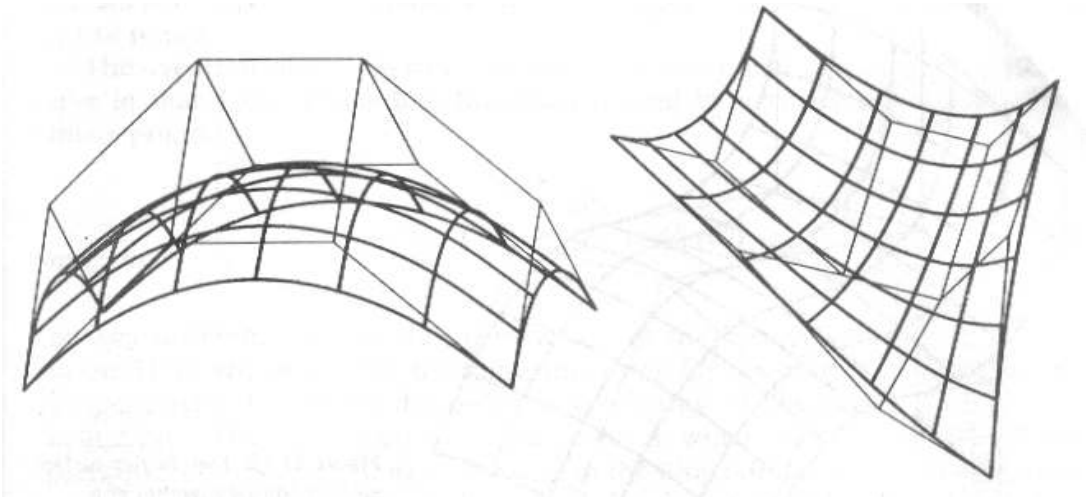


Fig: Two Bezier surface patches. The mesh of control points is shown with thin lines. Both patches have n=2, m=3

9.6 B-Spline methods

The overall formulation of the B-spline curve is much like that of the Bezier curve in that a set of blending functions is used to combine the effects of the control points:

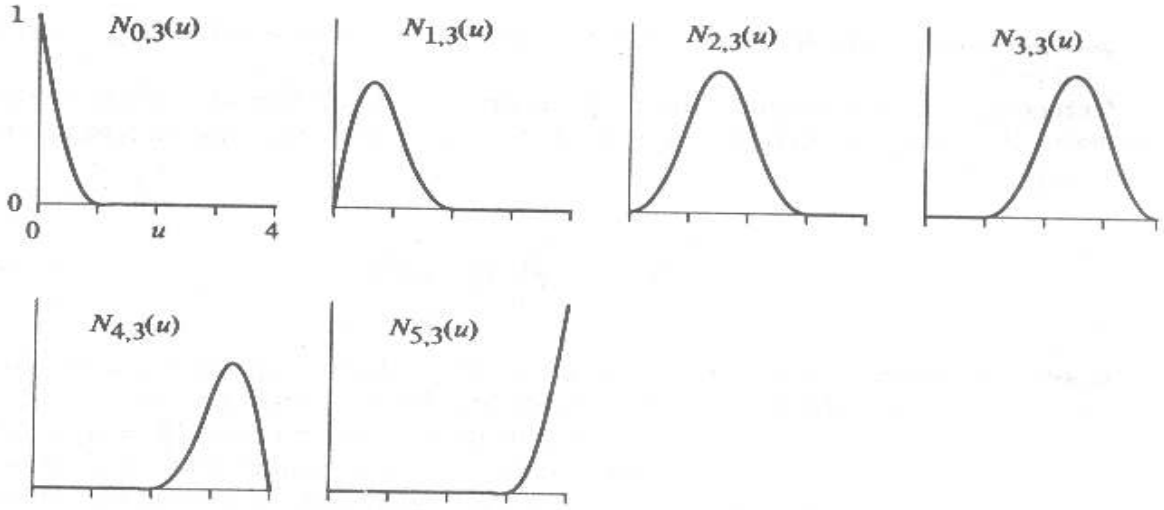
$$P(u)=\sum_{i=0}^{n} p_i \, N_{i,k}(u)$$

The key difference lies in the formulation of the blending functions $N_{i,k}(u)$.
The B-spline blending functions of degree k – 1 (also called the B-spline basis functions) may be defined recursively as follows:

$$N_{i,1}(u)=1 \text{ if } t_i<=u<t_{i+1}$$
$$= 0 \text{ otherwise}$$

$$N_{i,k}(u) = \frac{(u-t_i)N_{i,k-1}(u)}{t_{i+k-1}-t_i} + \frac{(t_{i+k}-u)N_{i+1,k-1}(u)}{t_{i+k}-t_{i+1}}$$

Because the denominators can become zero, this formulation adopts the convention 0/0 = 0. The following figure shows six different curves for six uniform non-periodic B-spline blending functions for n=5 and k=3.

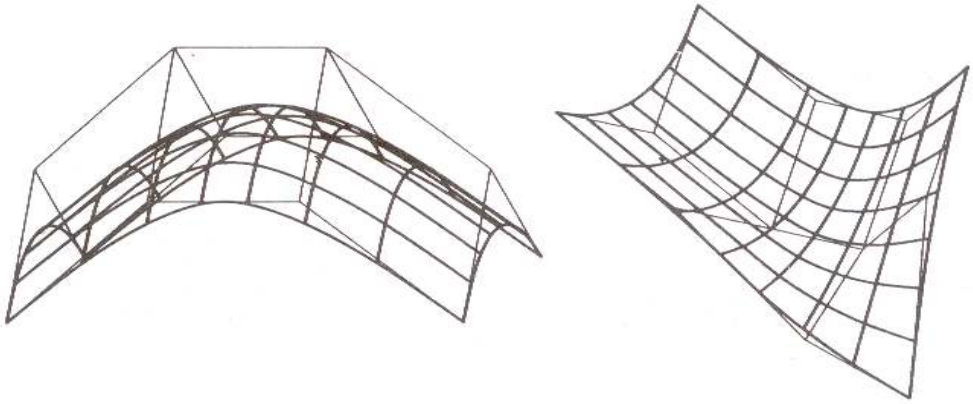


B-Spline Surfaces

B-spline extend to describe surfaces by the same Cartesian product method used with Bezier curves:

$$P(u, v)=\sum_{i=0}^{n}\sum_{j=0}^{m} p_{i,j}\, N_{i,k}(u)\, N_{j,l}(v)$$

Figure shows an example of B-spline surfaces and their mesh of control points.



UNIT 10
THREE DIMENSIONAL GRAPHICS

10.1 INTRODUCTION

In this unit, we get ourselves introduced to the realm of 3-dimensional graphics. Through 2-dimensional pictures help us in a number of areas, there are several applications where it is simply not sufficient to meet the requirements. We look into those areas where 2-D displays fall short of the demands initially. Then, since we have only a 2-dimensional display to represent 3-dimensional objects we briefly look into the various alternatives available for the user in brief. Of course, in the subsequent blocks, we study same of them with greater depth.

10.2 Need for 3-Dimensional Imaging

There are several areas of applications where 2-D imaging is not sufficient we look into some of them in brief.

a. **Computer Aided Design (CAD):** Computer generated images are of atmost use in several design applications like those of automobiles, aircraft's, mechanical parts etc. Since the computers can do fast computations and the displays can draw them for the visual analysis of the designer, CAD has gained immense popularity in recent years. Obviously, a mere 2-dimensional picture seldom tells the complete story. Further, design details like fixtures etc can be studied only in 3-Dimensions. Hence the use of 3-Dimensional pictures is obviously the key in CAD.

b. **Animation:** This is another fast growing area, A sequence of pictures that educate or explain some concept or simply are of entertainment value are presented with motion incorporated. In such

cases, mere 2-dimensional animation is of little interest and the viewer is to be treated to a virtual concept of depth.

c. **Simulation:** There are certain experiments that are either too costly or for certain other reasons cannot be conducted in full-scale reality. In certain other cases, a preliminary sequence of operations are done on the computer before a full fledged experimentation is taken. The examples of flight simulation or nuclear testing illustrate the concepts. In a flight simulation case, the trainee is made to "Experience" real flight even though he is stationery. In such a case, definitely a 2-Dimensional simulation is of very little use and for the trainee to experience fully the various complexities involved, an experience of depth is to be provided. Similarly in the case of a nuclear testing, a realistic study can be made only by having a 3-dimensional view on the screen.

In fact, the list of applications that need **3-**D views can go on endlessly. Instead, we simply underline the fact that using the 2-dimensional screen to provide a 3-dimensional effect is of prime importance and move on to the various ways in which this can be achieved.

10.3 Techniques for 3-Dimensional displaying

At the outset itself, it is to be made clear that since we are using a 2-dimensional screen for a 3-dimensional display, what we can achieve is only an approximation. Even this approximation is achieved at the cost of computational overheads i.e. additional computation is to be done before a picture can be fitted into a 2-D screen. Further there is a limit to the amount of computations that can be done. This limit is not set so much by the hardware / software capacities of the machine as by the available time. Going through some of the applications that need 3-dimensional views, it is clear that the effects are to be achieved within reasonable time. In an animation picture, if time delays prevent a continuous stream of pictures being presented to the viewer, then the whole idea behind animation is lost. In case of simulation, the limitations are more stringent. The views are to be presented to the viewer as they "happen" in the real world. If a plane is moving (or it's motion is being simulated), then the movement of hills, buildings etc should be presented at the same speed as it is experienced in a real case. Otherwise, the entire meaning of simulation is lost.

The aim of this discussion is to highlight the fact that the methods of presentation depend not only on how good is one scheme than the other, but also on how fast can one scheme gets executed than the other. With the rapid changes in hardware technologies, some of the schemes that were unattractive

previously have become useful now and the process will continue in future also.

10.4 Parallel projections

Those familiar with the concepts of engineering drawing will recall that any 3-dimensional object can be represented by it's projections on parallel planes. They constitute the front view, top view (and some times the side view). This is the simplest of the available techniques and can be done quite rapidly and also with reasonable accuracy. But the views will be useful only to trained engineers and architects whereas a common viewer may not be able to make much out of it. For example if a motor car is represented by what it looks like from the front, from the side and from the top, a trained engineer or mechanic can immediately visualize it's form, including the various dimensions. But a common man cannot make any thing out of it. Thus, this method may be useful in applications like CAD, but is useless as far as animation or simulation is concerned.

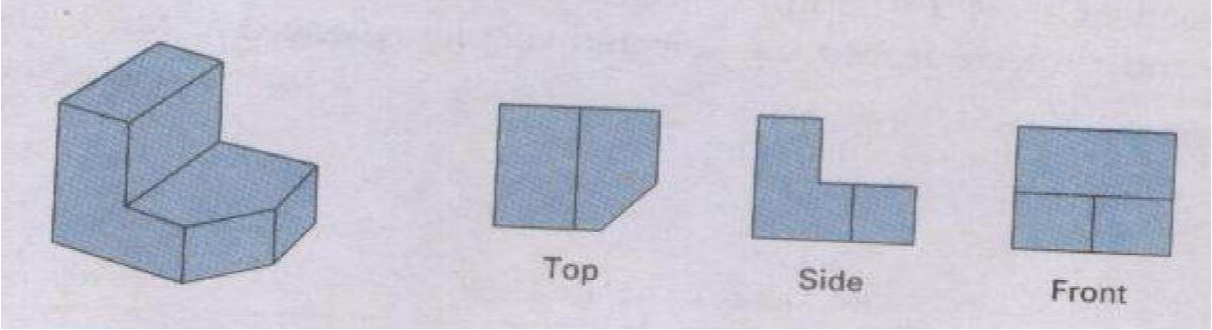


Fig. Three parallel projection views of an object, showing relative proportions from different viewing positions.

10.5 Perspective projection

This is the 'common man's' technique. When we see a number of objects or even a large object, parts that are nearer to the eye appear larger than those that are for away. Thus a matchbox can obscure a building, which is far away. This is the way all humans see and understand things in real life. Thus, the scheme provides very realistic depth information and is best suited for animation and simulation applications. But the draw back is that even though the method provides a feel of depth, it seldom provides the actual information about the depth. (The case of a matchbox obscuring the building clarifies the situation). It also is fairly computation intensive.

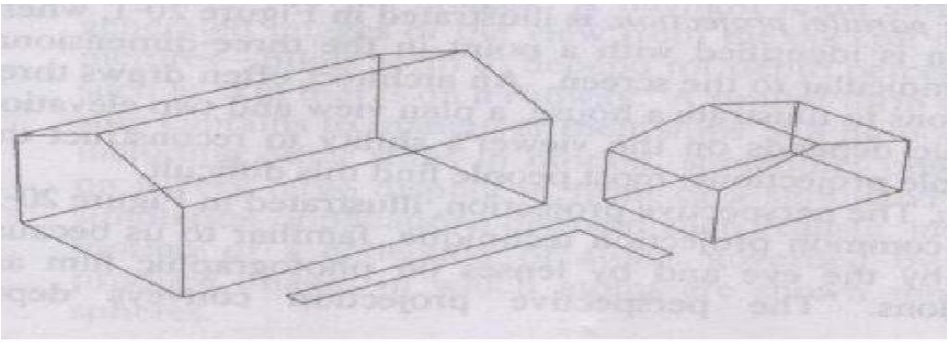


Fig. An object and its prospective view

10.6 Intensity cues

One depth cue that is not computationally intensive is the concept of intensity cues. As an object moves further from the viewer, its intensity decreases. Further, if it is made up of wide lines, the width of the lines decrease with increasing distance.

10.7 Stereoscope effect

The reason why we see depth is because of the stereoscopic effect of the eyes. We get two views of the same object by the two eyes and when these are superimposed, we get the idea about the depth. (In fact a clear idea about the depth coordinates cannot be got if only one eye is functional). The same can be used even in the case of computer displays. How exactly can we show two images differ? Either two different screens showing slightly displayed images of the same object can be shown or the same screen can be used to alternate the two views at more than 20 times per second. The method of polarized glasses is of a recent origin.

10.8 Kinetic depth effect

It is common experience that while in motion; objects that are far away appear to move much slower than those near by. The same can be used in the reverse method to give an indication of depth, especially in motion pictures. The objects that are supposed to be nearer to the viewer can be made

to move faster than those that are to be shown further away. The viewer automatically gets the feeling of difference in depths of the various objects. This could be a very useful technique especially in animation and simulation pictures.

10.9 Shading

Those who have done artistic pictures know that shading is a very powerful method of shading depth. Depending on the direction of incident light and the depth of the point under consideration, shades are generated. If they can be represented graphically, excellent ideas about depth can be created in the viewer. Raster graphics, which allow each pixel to be set to a large number of brightness values, is ideally suited for such shading operations.

Review questions

1. Name the method of sharing fast moving sequence of pictures.
2. State two reasons why simulation is resorted to?
3. What is the need for 3-dimentional representations of pictures?
4. Name the type of projections normally used in engineering drawings.
5. Which projection gives the most realistic view of the object?
6. What is stereoscope technique?
7. How can one produce the stereoscope effect with a computer display?
8. What is kinetic depth effect?

Answers:

1. Animation.
2. Certain experiments may be too costly; certain other experiments need lot of changes to be made, which is easier to incorporate on a computer.
3. Most of the objects we see in real life are 3-dimentional. Also in applications like animation or simulation, where realism is of prime importance, not able to give a concept of depth would make the whole concept useless.
4. Parallel Projection.
5. Perspective Projection.
6. The technique of showing two different pictures which are slightly displaced from each other, so that the user gets the idea of a third dimension is called the stereoscope technique.
7. Either by using two screen displaced slightly from each other or by using a single screen to produce both the views, one after the other at speeds greater than 20 times per second.

8. In moving objects, the following points move slowly compared to the nearby points. If a similar technique is used in moving pictures, the viewer gets a cue about the depth of the object.

UNIT 11
SOLID AREA SCAN CONVERSION

11.1 Introduction

In this unit, we learn about the concept of scan conversion of polygons. We talk about polygons, since any object of any random shape can be though of as a polygon – a figure bounded by a number of sides. Thus if we are able to do certain operations on the polygons, they can be extended to all other bodies.

So for, we have seen the line drawing algorithms. But if only a figure bounded by a number of sides is given, we do not know complex when a large number of polygons is there in the screen. We do not know whether the objects behind the present object are visible or not. So, we would like to make a distinction between objects that are inside the polygon and those that are outside and display them differently. The concept of identifying such pixels is called the “scan conversion”, since we convert the pixels along one scan line at a time.

We make use of the property of coherence- i.e. pixels that are in the same neighborhood share similar properties. Using this, we introduce you to the YX algorithm, which makes use of the intersections of polygons with the scan lines and the concept of coherence to suggest an efficient scan conversion methodology.

11.2 SOLID AREA SCAN CONVERSION

The main reason for the rapid increase in popularity of raster scan displays is their ability to display “solid” images. They are useful in representative, thickness, depth, or objects line up one behind another. Needless to say, the ability to display the third dimension is of prime importance in realistic display of objects, especially in video games and animation.

Generating a display of a solid object means one should be able to

i) Find out pixels that lie within the solid area and find out those that lie outside the solid area. This concept is called the mask of the area. One simple way of representative such pixels is to use a 1 to indicate pixels that lie inside the area and use a 0 to indicate pixels outside. The bit is called the "mask"

ii) To determine the shading rule. The shading rule deals with the pixel intensity of each pixel within the solid area. To give a realistic image for the depth, it is essential that the "shade" of each pixel be indicated separately, so as to give a coherent idea of the concept of depth. Such a mechanism would give the effect of shadows to pictures so that pixels that lie nearer to the observer would caste a shadow on those that are far away. A variable shading technique is of prime importance in presenting realistic 3-dimensional pictures.

iii) To determine the priority. When one speaks of 3-dimensions and a number of objects, the understanding is that some of the objects that are nearer are likely to cover the objects that are far away. Since each pixel can represent only one object, the pixel should become a part of the object that is nearest to the observer i.e. a priority is assigned to each object and if a pixel forms part of more than one object, then it will represent the object with the highest priority amongst them.

Out of these concepts, the problem of identifying those pixels that form a part of the object from those that do not is called "scan conversion" In this block; we see a few algorithms for scan conversion. In the subsequent blocks, we see more about the other aspects.

11.3 Scan conversion of polygons

The simplest algorithm of scan conversion can do something like this

i) Display the boundary of the solid object

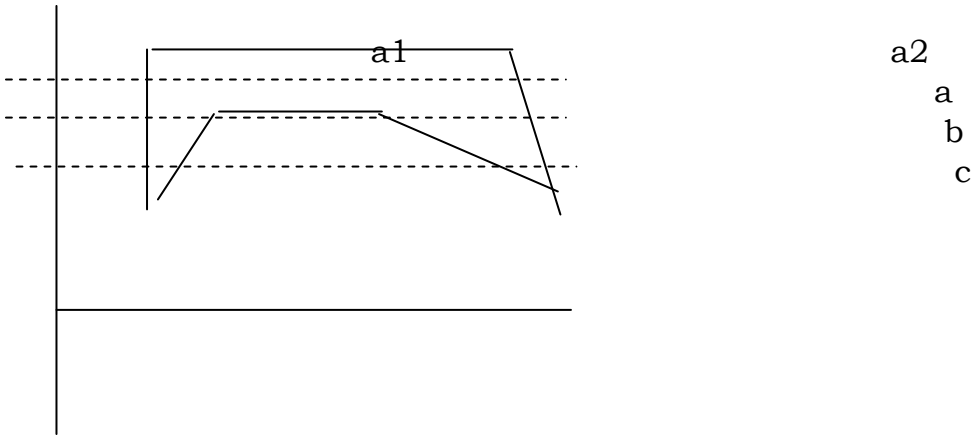
ii) For each pixel on the screen, try to find out whether it lies inside the boundary or on the boundary or outside it. Suitably arrange the mask of each.

Though this method is simple and reliable, it needs enormous amounts of computations to achieve the purpose. Obviously more efficient methods for scan conversion are needed. One may note that the trade off involved is not just the time involved, but such inordinate delays avoid a proper, real time modifications and display of the picture. Hence, several algorithms have been devised so that certain inherent properties of the pictures are utilized to reduce the computations. One such broad technique is to

consider figures as closed polygons and use certain properties of polygons to find out which pixels should lie inside the picture and which are not one such property is coherence.

9.4 Coherence

The performance of a scan conversion algorithm can be substantially improved by taking advantage of the property of coherence i.e. Given a pixel that is inside a polygon, it's immediately adjacent pixels are most likely to be also inside the polygon. Similarity if a pixel is outside a polygon, most of its adjacent ones also will be most probably outside it. A corollary is that the coherence property changes only at the boundaries i.e. we will have to check the status of the pixels only at the boundaries and immediately adjacent to it, to find out whether the pixel lies inside or outside. The property of coherence can be applied to all its neighboring pixels and hence their status need not be checked individually. Consider the following example. Given a polygon, it is to be scan converted.



Suppose we want to identify all those pixels that lie inside the polygon and those that lie outside. This can be done in stages, scan line by scan line. Consider the scan line a. This is made up of a number of pixels. Beginning with left most point of the scan line, compute the intersections of the edges of the polygon with the particular scan line. In this case these are two intersections (91 & a2). Starting at the left most pixels, all pixels lie outside the polygon up to the first intersection. From then on all pixels lie inside the polygon until the next intersection. Then afterwards, all pixels lie outside. Now consider a line b. It has more than two intersections with the polygon. In this case, the first intersection indicates the beginning of the series of pixels inside

the polygon, the next intersection indicates that the following pixels will be inside the polygon and fourth intersection concludes the series.

Now we write this observation as an algorithm. This algorithm is called the (yx) Algorithm (We will see at the end of the algorithm, why this peculiar name).

9.5 (yx) Algorithm

1. For every edge of the polygon, find out it's intersection with all the scan lines (This is a fairly straight forward process, because beginning with one tip of the edge, every incremental value of y gives the next scan line and hence a DDA type algorithm can be written to compute all such intersections very fast and quite efficiency. However, we leave this portion to the student). Build a list of all those (x,y) intersections.

2. Sort the list so that the intersections of each scan line are at one place. Then sort them again with respect to the x coordinate values. (Understanding this concept is central to the algorithm). To simplify the operations, in stage 1, we simply computed the intersections of every edge with every (intersecting) scan line. This gives a fairly large number of (unordered) points. Now sort these points w.r.t. their y-values, i.e. the scan line values. Assuming that the first scan line has a y value of 1, we get the list of it's intersections with every edge. Then of the scan line with value 2 and soon. At this stage, looking at the previous example, we have the intersections of 'a' listed first, then intersection of 'b' and then of 'c' Now sort these intersections separately w.r.t. x points. Then the points a_1 and a_2 appear in the order, similarly of b and c)

3. Remove the intersection points in pairs. The first of these points indicate the beginning of the series of pixels that should lie inside the polygon and the second one ends the series. (a_1 and a_2 in this case) . (In the case of the scan line b, we get two pairs of intersections, since we have two sets of pixels inside the polygon for that scan line, while an intermediate set lies outside). This information can be used to display the pictures.

Incidentally this algorithm is called the yx algorithm, since it sorts the elements first w.r.t. y and then w.r.t. x. We leave it to the student to try and write a xy algorithm and ensure that it does the job equally well.

11.6 Singularities

Note that we have not commented on the scan line c of the picture. The peculiarity of that line is that the intersection lies exactly on the vertex of a polygon. In such a case, it is very easy to see that the algorithm fails. This is because the intersection of the scan line with the vertex not only defines the beginning of a series of pixels that lie inside the polygon, but also the end of the series. Now how do we treat such intersections?

One earlier solution suggested was never to have such intersections at all i,e. instead of sharp vertices, have only blunt vertices. Then every scan line will have two intersections instead of one. But obviously this solution is not a welcome one since it distorts the picture altogether.

The other suggestion is to identify that such a "special" type of intersection has occurred and treat it as two intersections at the same point. This solves the problem elegantly, the only problem being that how do we identify? The answer is to keep track of the direction of the polygon edges. It is easy to note that the polygon changes it's "direction" at it's vertex. So, whenever an intersection is recorded, find out whether the next point on the boundary of polygon lies on a monotonically increasing / decreasing sequence or is on a different direction altogether. Once this is done, if the direction is different, then include two points instead of one into the list of intersections (With the same coordinate values, of course). Now, we write a simple algorithm that treats such singularity problems. This algorithm also takes care of the other imminent problem – that of horizontal edges. A horizontal edge would intersect with every pixel of the scan line and how to deal with such a situation wherein every pixel can be considered to be inside the polygon is also dealt with here.

11.7 Algorithm Singularity:

1. A variable y_{prev} is used to keep track of the previous intersection of the edge. Whenever an intersection is found, not only is a new pair of (x,y) stored as in the yx algorithm, but the y coordinate is stored to indicate the previous intersection by storing it in y_{prev}. Initially its value is set to 0.

2. Go to the next edge of the polygon. If these are no more edges to be processed, exit.

3. Compute it's intersection with the scan lines. If it has no intersections at all (or a very large no. of intersections, the way you look at it) it can be considered horizontal. Go to step 2

4. Compute the difference $dy=y_2-y_1$, where y_2 is the y coordinate of the beginning vertex of the edge and y_1 the y coordinate of the ending vertex. If dy>o go to step5, else go to step6.

5. If dy>0, the first intersection generated must have $y= y_{prev.}+1$ compute all other intersections of the edge. The y coordinate of the last intersection is stored in $y_{prev.}$. Go to step 2 to findout whether any edges are still there.

6. If dy<o, the first intersection generated will have $y= y_{prev.}$ itself generate all intersections for the edge. The y coordinate of the last intersection is preserved in $y_{prev.} = y_{last} -1$. Go to step2.

Note that this algorithm does not generate intersection nor does it produce the scan conversion. The scan conversion algorithm, which does the conversion, will only pass its intersection values to the singularity algorithm to check for the specific cases.

The other aspect to be taken care of while displaying polygons is to decide on the priority. In 3 dimensional graphics, it is obvious that two or more polygons tend to overlap one another. In such cases, only the polygon that it is closest to the observer will visible. This polygon obscures any more polygons behind it. But the problem is that the front polygon may not cover the polygon behind it completely. That means the farther polygon is visible in those places where it is not covered by the front polygon, but will not be visible in those regions where the front polygon covers it. One solution to solve this problem is to find the intersections of the polygons display the front polygon completely and display the back polygon(s) in these areas where the front polygon is not covering it. But, if you consider cases wherein a large number of polygons is covering one another at different regions, this method becomes unwieldy.

A very ingenious method to solve this problem of assigning priorities to the algorithms has been devised. This is called a "painters algorithm". Imagine a painter painting these polygons on his canvas. What does he do? He does not bother himself about intersections or partially obscurities. He begins by painting the furthest polygon, say in a particular color. This obviously has the least priority in display i.e. it will be displayed only when no other polygon is obscuring it. Then he begins painting the next polygon in front

of it. He simply goes about painting this second polygon, without bothering about the previous polygon. This new polygon, let us say polygon 2, has a higher priority than the polygon 1. i.e. when the two polygons appear together, polygon 2 will be visible completely and polygon 1 is visible only if polygon 2 is not obscuring it in that region. Now, once the second polygon is painted, in a different color, it is simple to analyze that the parts of the polygon 1 that are covered by polygon 2 automatically get covered and becomes invisible. Similarly if a polygon 3 is painted, it gets the highest priority in display.

Thus, an extremely simple concept emerges. Do not bother about any mathematical formulations. Start from the farthest polygon and keep displaying them in the order of increasing priorities. The priorities are automatically taken care of.

Expressed in technical terms the algorithm can be expressed as follows. Assign a priority to each polygon, the lowest priority to the polygon that is farthest from the viewer and the highest priority to the one that is nearest. Sort them on the order of priority. Scan covert each of the polygons and start displaying them in the increasing order of priority. The higher order polygons automatically cover the lower order ones and the priority concept is case of.

This algorithm can be called p-(yx) algorithm on the lines similar to yx algorithm. In the yx algorithm we were first ordering on y then on x coordinates. Here, before that, we order the polygons based on priorities. i.e. 3 stages of p, y and x sorting are involved. Hence p(yx) algorithm.

This algorithm, however, has a very minor drawback. It is not visually appealing. Since the algorithms appear in the order of priorities, from the least to the highest, the most important of them appear only in the end. Also, it will be distracting to the user to see the polygons appearing one after the other, one overwriting another. A better scheme will be to make them appear in their final order, even if it is from one end to the other (or top to bottom). A simple way to do this is to first sort the pixels in terms of their y coordinates (scan line by scan line), then on their priorities and finally on there x coordinates. Then the algorithm changes to an ypx algorithm.

The algorithm, in brief, appears as follows:

1. For each polygon, compute the intersections with every scan line. This yields a list of (x,y,p) where x & y are the coordinates of the point of intersection and p is the priority of the polygon.
2. Sort the list first by y, then sort w.r.t. p and finally w.r.t. x

3. Remove pairs of nodes from this sorted list and scan convert as before.

One difficulty of this algorithm is that it takes large amounts of computational efforts for sorting, since a large number of points are involved. The only solution will be to resort to efficient sorting algorithms.

Review questions

1. What is scan conversion?
2. Why are we specific about polygons?
3. What is priority in the concept of a pixel?
4. What is coherence?
5. Why yx algorithm called so?
6. What is singularity? How are they taken care of in yx algorithm?
7. How is a singular point identified?

Answers:

1. The idea of identifying and converting pixels along a scan line that lie inside the polygon so that they can be displayed differently.
2. Because once we are able to do certain operations on polygons, they can be extended to others, since most of the regular and irregular boundaries can be thought of as polygons.
3. In 3-dimensional views, when more than one object stands one behind another, the same pixel on the screen represents more than one object. So the priority for the pixel as to which object it should represent is important.
4. Pixels in the same neighborhood share similar properties – most often. If a pixel is inside a polygon, most probably, it's neighbors also will be inside the same polygon. Hence, the same set of operations need not be repeated on each of them.
5. Since it first sorts the elements with respect to y and then with respect to x.
6. When a vertex coincides with a scan line, it is a singular because the scan line entres and leaves the polygon at the same place. They are counted as z intersections for the algorithm.
7. In a singular point, an edge of the polygon changes it's direction.

UNIT 10
THREE DIMENSIONAL TRANSFORMATIONS

10.1 Introduction

In this unit, we look into the basics of 3-D graphics, beginning with transformations. In fact the ability to transform a 3-dimensional point, i.e. a point represented by 3 Co-ordinates (x,y,z) is of immense importance not only for the various operations on the picture, but also for the ability to display the 3-D picture in a 2-D screen. We briefly see the various transformation operations – they are nearly similar to the 2-D operations. We also see the concepts of clipping and windowing in 3-D.

10.2 Three Dimensional Transformation

Just as in the case of 2D, we represent the transformation operations as a series of matrix operations. With this, we obtain the flexibility of sequencing a series of operations one after the other to get the desired results on one hand and also the ability to undo the operations, by resorting to the reverse sequence. Since in the 2-dimensional case we were representing a point (x,y) as a tuple [x y 1], in the 3-dimensional case, we represent a point (x,y,z) as a [x y z 1]. The dimensions of the matrices grow from 3 x 3 to 4 x 4.

10.3 Translations

Without repeating the earlier methods, we simply write

$$[x_1 \ y_1 \ z_1 \ 1] = [x \ y \ z \ 1] \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{pmatrix}$$

Where the point [x y z 1] gets transformed to [x_1 y_1 z_1 1] after translating by Tx, Ty and Tz along the x,y,z directions respectively.

10.4 Scaling

A given point [x y z 1] gets transformed to [x_1 y_1 z_1 1] after getting scaled by factors Sx, Sy and Sz in the three dimensions to

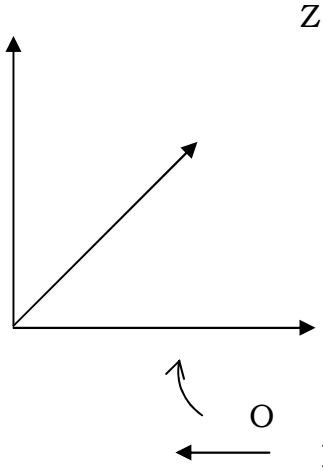
$$[x_1 \ y_1 \ z_1 \ 1] = [x \ y \ z \ 1] \begin{pmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

10.5 Rotation

Rotation in 3-dimensions is a more complex affair. (In fact, even in 2 dimensions, rotation was more involved than scaling or translation because the concept of point of rotation). This is because; the rotation takes place about an axis. The same point, given the same amount of rotation, gets transformed to different points depending on which axis it was rotated.

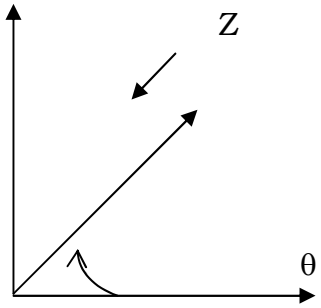
The simplest of the cases is to rotate the point about an axis that passes through the origin, and coincides with one of the axes x, y or z. The next complication arises when the axis passes through the origin, but does not coincide with any of the axes. The most general case would be, of course, when an arbitrary axis that does not pass through the origin becomes the axis of rotation.

Let us begin with simplest cases: The understanding is that a clockwise rotation, when viewed at the origin, standing on the axis is taken as positive and the other direction is negative. If this description looks too complicated, look at the following figures. In each case, we write down the transformation for the rotation through a positive angle of +θ.



$$[x_1 \; y_1 \; z_1 \; 1] = [x \; y \; z \; 1] \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 1 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

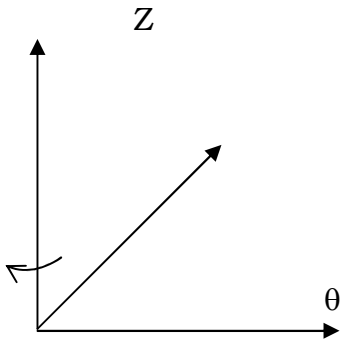
Transformation Matrix



Direction of view of
Origin
Y

$$[x_1 \; y_1 \; z_1 \; 1] = [x \; y \; z \; 1] \begin{pmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Transformation Matrix



Direction of view of origin

Y

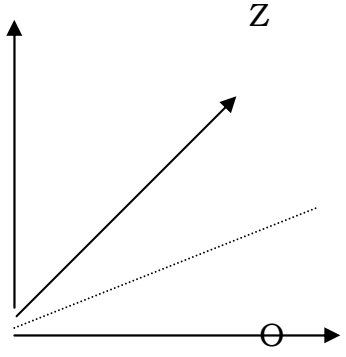
O X

$$[x_1 \ y_1 \ z_1 \ 1] = [x \ y \ z \ 1] \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Transformation Matrix

Now the other cases of rotation can be taken to be concatenations of the various operations.

For example to rotate a point about an axis that passes through the origin, but does not coincide with any of the axes, as in the following figure



Y

X

Suppose the axis passes through the origin, but does not coincide with any of the axes, then the axis itself is to be first aligned to one of the axes before doing the transformations. The sequence of events appears as follows.

(a) Rotate the axis through the desired angle to make it coincide with one of the axes. (Depending on with respect to which axis, it's angle of deviation is available).

(b) Rotate the point (desired to be rotated) about this axis.

(c) Rotate the axis back to its original angle of deviation.

In cases where the axis is an arbitrary axis passing through some points, but not the origin, the sequence lengthens

(a) Shift the point through which the axis points to the origin.

(b) Rotate the axis through an angle necessary to coincide it with one of the primary axes.

(c) Rotate the point to be rotated about this axis.

(d) Rotate the axis back to it's original inclination.

(e) Shift the point from the origin to it's original point.

(Similar operations have been illustrated n the case of 2-dimensional operations in the earlier block. The student is encouraged to attempt to write transformations for above cases on similar lines).

(It is also to be noted that reverse operations can be done fairly easily using matrices. For example if a rotation is made through an angle θ, to undo the operation, one need not go to the extent of finding the inverse of the original transformation matrix, but by simply multiplying the resultant with one more matrix where $\cos(\theta)$ and $\sin(\theta)$ are replaced by $\cos(-\theta)$ and $\sin(-\theta)$. In the case of scaling -Sx and -Sy perform the inverse operations for Sx and Sy).

10.6 Viewing Transformations

Before displaying the 3-D picture one more set of "viewing transformations" is to be done. This is to ensure that the viewer would get the depth perspective that has been displayed on the screen. This is done by transforming the image to another coordinate system with axes xe, ye and ze, the origin being the position of the eye and ze being the axis passing from the eye perpendicularly to the screen.

If this transformation is called V, then it is not enough if we simply display the completed picture as it is, but every point is to be transformed to the eye coordinate system.

Calling the original picture coordinate systems as the world system, if a point in it is represented by $[\ x_w\ y_w\ z_w\ \ 1]$ then it should be

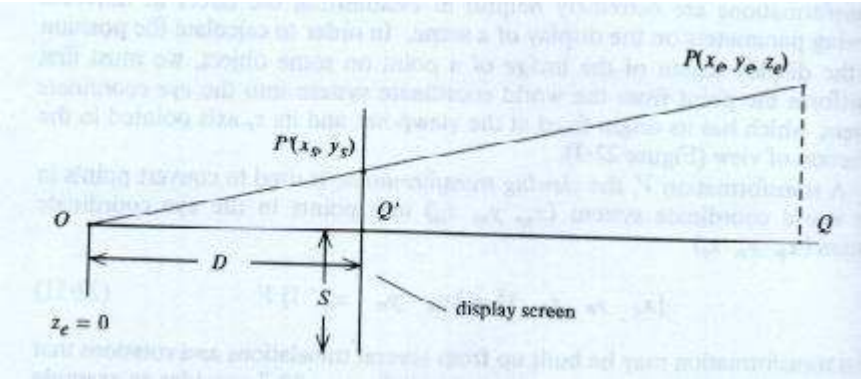
transformed to the eye coordinate system [x_e y_e z_e 1] using the following transformation

$$[x_e\ y_e\ z_e\ 1] = [\ x_w\ y_w\ z_w\ \ 1]\ V$$

V is a series of matrix transformations that can be got by including several translations and rotations that are determined by viewing parameters.

10.7 The perspective Transformation

A perspective display can be generated by simply projecting every point of the object on to the plane of the screen. This section teaches you to get the coordinates (x_s - y_s) in the screen coordinates with respect to the eye coordinates (x_e, y_e, z_e).



Consider the above figure, which indicates the basics of the perspective projection. O is the point behind the screen, which is called the "Perspective Point", the point. The point P measured in eye coordinate is available at P(x_e,y_e,z_e). The effort is to find the coordinates of the same point P(x_s,y_s) on the screen (also called screen coordinates) so that the perspective effect is established. D is the distance of the convergence point (where z_e = 0) behind screen and S is half width of the screen.

The triangles OQ'P' and OQP are similar.
Hence y_s / D = y_e / z_e
Similarly it can be shown that

$$x_s / D = x_e / z_e$$

The numbers x_s and y_s can be converted to fractions by dividing them by the screen size. This operation not only allows as to numbers which are fractions, but it also makes the numbers dimensionless (we are dividing a dimension with another dimension).

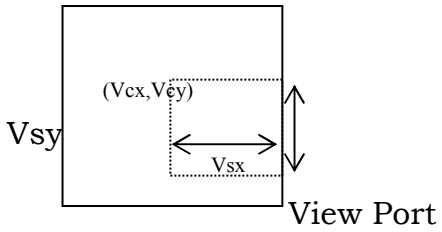
$X_s / D = X_e / sz_e$ and $Y_s / D = Y_e / S z_e$
Or $X_s = D x_e / Sz_e$ and $y_s = D y_e / Sz_e$

Alternatively they can be converted to the screen coordinates by including a specification of the location of view port in which the image is displayed.

$$Xs = (Dx_e / Sz_e) V_{sx} + V_{cx} \text{ and } y_s = (Dye / Sze) V_{sy} + V_{cy}$$

[This can be derived as follows:

The view port is at the centre (Vcx, Vcy) and is 2 Vsx units wide and 2 Vsy unit high



Hence a value xs which is given in the window coordinates as (xs = Dxe / Sze) is scaled by a factor Vsx and is shifted by a value Vcx.

Similarly the coordinate xy given by a value Dye / Sze is scaled by a factor Vsy and is shifted by a value Vcy. You can remember that we used a similar methodology to transform from windows to view port in an earlier unit.

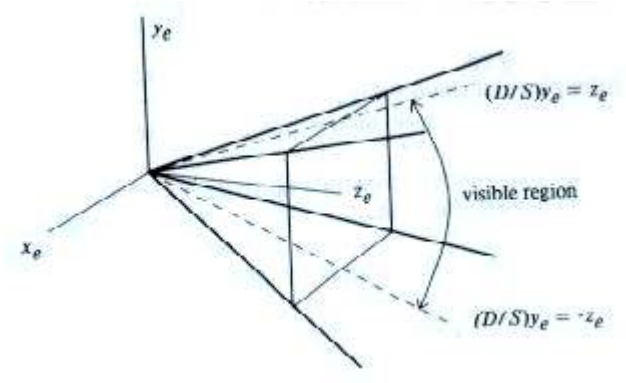
Thus, in order to convert a picture it's perspective equivalent, we should convert every point (xe,ye) using these formulae.

However, this involves a series difficulty. As we can see, divisions are involved in both the conversions. I.e. to convert a single point to it's perspective equivalent, we have a few multiplication's and additions, but more worryingly two divisions as well. Computers are most efficient for additions and multiplications but are thoroughly inefficient with division. Fortunately a picture can be converted into a perspective equivalent by transforming only the corners of the picture.

10.8 Three dimension clipping

The direct application of the perspective conversion may end up mapping the object to size of the window; also points beyond the view port may also get mapped. To circumvent these problems, the object generated needs to

be clipped against a viewing pyramid. The concept is similar to the 2-dimensional case.



10.9 CLIPPING

Every point needs to be checked as to whether it lies within the visible area of the pyramid by comparing the values of (D/S) xe and (D/s) ye to the end values of the visible area

i.e. $-Ze \leq (D/S)\, xe \leq +Ze$ and $-Ze \leq (D/s)\, ye \leq +Ze$.

This will exclude all points beyond the view point ($ze \leq 0$) and all points that go beyond the visible pyramid.

Note that this creates on problem. We indicated before that an object can be transformed to it's perspective by simply mapping it's endpoints. But, if these points go beyond the visible pyramid, then they cannot be displayed. But this does not mean the entire object cannot be rejected. The intersection of the visible pyramid with the profile of the object needs to be computed. This clipping has to work on a 3-dimensional perspective.

A three dimensional extending the 2-D scheme can derive clipping algorithm.

The algorithm determines whether the end point of the line lies in the visible pyramid by assigning a 4 bit code to it.

First bit is 1: if the point is to the left the pyramid, else it will be zero, similarly,

Second bit : If the point is to the right of the pyramid

Third bit: If the point is below the pyramid

Fourth bit : If it is above the pyramid.

As earlier, if the codes for both the end points are 0000, then the line is trivially accepted. If the logical AND of the codes is not zero, both end points lie on the invisible side of one of the planes and can be trivially rejected.

Otherwise the line crosses the side of the pyramid at one/more points. The point of intersection can be computed in the parametric form as

$$((1- t) [x1 \;\; y1 \;\; z1] + t \begin{bmatrix} xz \\ yz \\ Zz \end{bmatrix}) \begin{bmatrix} \alpha \\ \beta \\ 1 \end{bmatrix} = 0$$

Where different values of α and β have different values for different planes. For example if $\alpha = 1$, $\beta = 0$, then it indicates the plane x=z. By substituting the various values of α and β one can find the point of intersection with different planes.

These Intersections then replace the ends of the line in the viewing pyramid.

10.10 Perspective view of a cube

We clarify the above discussions with an example, getting the perspective view of a cube. Consider a cube centered at the origin of the world coordinate system, defined by the following points and lines:

Lines	Points			
		X	Y	Z
AB, BC	A	-1	1	-1
CD, DA	B	1	1	-1
EF, FG	C	1	-1	-1
GH, HE	D	-1	-1	-1
AE, BF	E	-1	1	1
CG, DH	F	1	1	1
	G	1	-1	1
	H	-1	-1	1

We shall observe this cube from a point (6, 8, 7.5) with the viewing axis ze pointed directly at the origin of the world coordinate system. There is still one degree of freedom left, namely an arbitrary rotation about the ze axis: we shall assume that the xe axis lies in the z = 7.5 plane.

The viewing transformation is established by a sequence of changes of coordinate systems. Recall that a transformation that moves a coordinate system is the inverse of the corresponding transformation that moves points.

1. The coordinate system is translated to (6,8,7.5), the point in the original coordinate systems becomes the origin:

$$T_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -6 & -8 & -7.5 & 1 \end{pmatrix}$$

2. Rotate the coordinate system about the x' axis by -90°. Because we require the inverse transformation, we substitute $\theta = 90^{o}$.

$$T_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3. Rotate about the y' axis by an angle θ so that the point (0, 0, 7.5) will lie on the z' axis. We have $\cos -\theta = \cos\theta = -8/10$ and $\sin -\theta = -\sin\theta = 6/10$:

$$T_3 = \begin{pmatrix} -0.8 & 0 & 0.6 & 0 \\ 0 & 1 & 0 & 0 \\ -0.6 & 0 & -0.8 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

4. Rotate about the x' axis by an angle φ so that the origin of the original coordinate system will lie on the z' axis, we have $\cos -\varphi = \cos\varphi = 10/12.5$ and $\sin -\varphi = -\sin\varphi = -7.5/12.5$:

$$T_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.8 & 0.6 & 0 \\ 0 & -0.6 & -0.8 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

4. Finally reverse the sense of the z' axis in order to create a left handed coordinate system that conforms to the conversions of the eye coordinate system, A scaling matrix is used.

$$T_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This completes the five primitive transformations needed to establish the viewing transformation $V = T_1\ T_2\ T_3\ T_4\ T_5$

Suppose that we wish to fill a 30 by 30 centimeter display screen, designed to be viewed from 60 centimeters away, and that the coordinate system of the screen runds from 0 to 1023. Thus D = 60, S = 15, and Vsx = Vcs = Vsy =Vcy = 1023/2

The transformation is therefore

$$N = \begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

xs = 511.5 (xc / zc) + 511.5 ys = 511.5 (yc/zc) + 511.5

All the details of the transformations have now been specified. The matrix VN clipped and converted to screen coordinates using the above equation transforms each vertex of the cube

$$VN = T_1\ T_2\ T_3\ T_4\ T_5 N = \begin{pmatrix} -3.2 & -1.44 & -0.48 & 0 \\ 2.4 & -1.92 & -0.64 & 0 \\ 0 & 3.2 & -0.6 & 0 \\ 0 & 0 & 12.5 & 1 \end{pmatrix}$$

We can now apply this transformation to the eight vertices of the cube:

	Xc	yc	zc
A	5.6	-3.68	12.94
B	-0.8	-6.56	11.98
C	-5.6	-2.72	13.26
D	0.8	0.16	14.22

E	5.6	2.72	11.74
F	-0.8	-0.16	10.78
G	-5.6	3.68	12.06
H	0.8	6.56	13.02

Although the clipping routine must be applied to each line in the cube, it is apparent from the table that all of the vertices lie within the viewing pyramid, and the clipping algorithm will trivially accept each line.

Questions

1. What is the order of matrices in 3-dimensional representation of pictures?
2. What are the sequence of steps involved in rotating a given 3-D point about an axis passing through the origin, but not coinciding with any of the principal axes?
3. What is the name given to the method of projecting drawings in a way similar to that seen by the eye?
5. Write down the formulae for transforming the eye coordinates to screen coordinates.
6. Explain the concept of 4 bit assignment for clipping algorithm.

Answers:

1. 4 X 4
2.

a) Rotate the axis to make it coincide with x,y or z axis
b) Rotate the point suitably over this axis.
c) Bring the axis back to its original position by the sequence of reverse transformations.

3. Perspective projection.
4. Xs = (DXe/SZe) Vsx + Vcx
And Ys = (DYe / SZe) Vsy + Vcy
Where Xs and Ys are screen coordinates,
Xe and Ye are the eye coordinates
S is half screen size
Vsx and Vsy are the dimensions of the viewport
Vcx and Vcy are the shift of the viewport with respect to the screen coordinates.

5. First bit is set to 1 if the point is to the left of viewing pyramid
Second bit is set to 1 if the point is to the right of the viewing pyramid
Third bit is set to 1 if the point is below viewing pyramid
Fourth bit is set to 1 if the point is above the viewing pyramid

UNIT 11
HIDDEN SURFACE REMOVAL

11.1 Introduction

In this unit, you will be introduced to one of the most interesting and involved concept of computer graphics – the concept of hidden surface elimination. When two or more object are represented one behind the other – it is quite clear that some of them either partially or fully obscure the other object in such cases the hidden parts of the objects are to be removed.

Several algorithms for the same are introduced. Almost all of them work on the simple concept of sorting the polygons in the order of their distance, the nearest ones being represented in full, the farther ones in part since, in raster graphics, a given pixel can represent more than one object (each with same x,y) it will represent that object that is nearest to the screen amongst these objects. Though this concept is straight forward and accurate, it suffers from the difficulty that it is computationally intensive. Hence, several efficient algorithms, which perform the same job with more efficiency, are introduced. Most of them make use of some sort of coherence concept – i.e. pixels in the neighborhood of a pixel share the properties of a pixel. i.e. If a pixel forms a part of an object, the neighboring pixel also, most probably, form the part of the same object. You will also be introduced to certain specific instances, wherein these concepts may not yield satisfactory results.

11.2 Need for hidden surface removal

This has been considered as one of the most challenging jobs of computer graphics. Once we start talking of solid objects in 3 dimensional spaces, it is implied that some of the objects that are nearer to the viewer tend to partly or wholly cover other objects. In fact, even if there is only one object, some of its faces are unseen (the back faces) and some are partially seen (the

side faces). The ability to identify the faces and surfaces that are to be covered and the extent of coverage in the case of partially covered surfaces in real time is not only computationally intensive, but also analytically daunting. When only wire frame types of drawings are being displayed, the task gets somewhat simplified to that of "hidden line removal" – identifying those lines that should not be shown. However, when solid objects are being considered, the task becomes more complex because entire surfaces need to be identified for removal.

A large number of algorithms are available for the job –though no single algorithm can be though to be all encompassing capable of being efficient in all possible conditions. However almost all of them share some common feature. The first one is that at some point in the algorithm, they tend to sort the objects in the order of their Z-distance from the viewer and try to eliminate the farthest ones. But the sorting tends to be a difficult task at least in some cases, since often an object may not be identified with a unique distance – Z. When several part of the object has different Z coordinates, simple, direct sorting methods may become inadequate.

The other common feature with these algorithms in the use of coherence. As we have seen in other contexts, the coherence (or similarity with respect to a property) between neighboring pixels is used to reduce the number of computations effectively.

The behavior of the algorithm also depends on which type of images one is talking of. In the case of line drawing algorithms, the problem is solved using the various properties of lines, whereas in the case of raster images, the algorithms tend to look like extensions of 2-dimensional scan conversion algorithms.

The algorithms can also work either with respect to the object space or the image space. One should clearly be able to draw the distinguishing line between them. The object space in the space occupied by the pictures created by the algorithms. However, before these pictures can be displayed, they undergo various operations – like clipping, windowing, perspective transformations etc. This final set of pictures – ready for display on the screen is called the image space. The object space algorithms tend to calculate the values with as a precision as feasible since often these calculations form the basis for the next set of calculations, whereas the image space algorithms calculate with precision that is in line with the precision available with the display devices. This is because any higher precision, achieved with great efforts, will become useless since the display devices cannot anyway handle such precisions. Further, the computational efforts in the case of objects –

since every object tend to rapidly increase with the no. of objects – since every object will have to be tested with other objects, where as in the image apace computations, the increase is much slower, since one tends to look at the number of pixels, irrespective of the no. of objects in the scene. The number of pixels in a given resolution of display device is a constant.

Having noted some of the expected features of the algorithms, we now look into the working of some of the algorithms.

11.3 The Depth – Buffer algorithm

The concept of this algorithm is extremely simple and straightforward. Given a given resolution of the screen, every pixel on the screen can represent a point on one (and only one) object (or it may be set to the back ground if it does not form a part of any object). I.e. irrespective of the number of objects in line with the pixel, it should represent the object nearest to the viewer. The algorithm aims at deciding for every pixel the object whose features it should represent. The depth-buffer algorithm used two arrays, depth and intensity value. The size of the arrays equals the number of pixels. As can be expected, the corresponding elements of the array store the depth and intensity represented by each of the pixels.

The algorithm itself proceeds like this

Algorithm Depth Buffer:

a. For every pixel, set it's depth and intensity pixels to the back ground value ie. At the end of the algorithm, if the pixel does not become a part of any of the objects it represents the background value.

b. For each polygon on the scene, find out the pixels that lie within this polygon (which is nothing but the set of pixels that are chosen if this polygon is to be displayed completely).

For each of the pixels

i) Calculate the depth Z of the polygon at that point (note that a polygon, which is inclined to the plane of the screen will have different depths at different points)

ii) If this Z is less than the previously stored value of depth in this pixel, it means the new polygon is closer than the earlier polygon which the pixel was representing and hence the new value of Z should be stored in it. (i.e from now on it represents the new polygon). The corresponding intensity is stored in intensity vector.

If the new Z is greater than the previously stored vale, the new polygon is at a farther distance than the earlier one and no changes need be made. The polygon continues to represents the previous polygon.

One may note that at the end of the processing of all the polygons, every pixel, will have the intensity value of the object which it should display in its intensity location and this can be displayed.

This simple algorithm, as can be expected, works on the image space. The scene should have properly projected and clipped before the algorithm is used.

The basic limitation of the algorithm is it's computational intensiveness. On a 1024 X 1024 screen it will have to evaluate the status of each of these pixels in a limiting case. In it's present form, it does not use any of the coherence or other geometric properties to reduce the computational efforts.

To reduce the storage, some times the screen is divided into smaller regions like say 50 X 50 or 100 X 100 pixels, computations made for each of this religions, displayed on the screen and then the next region is undertaken. However this can be both advantageous and disadvantageous. It is obvious that such a division of screen would need each of the polygons to be processed for each of the regions – thereby increasing the computational efforts. This is disadvantage. But when smaller regions are being considered, it is possible to make use of various coherence tests, thereby reducing the number of pixels to be handled explicitly.

11.4 Properties that help in reducing the efforts

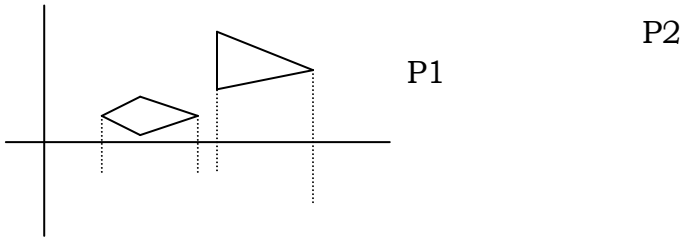
Several features can be made use of to identify the polygons that are totally/partially covered, so that the actual effort of elimination of hidden surfaces can be reduced. A few popularly used tests are as follows.

i) Use of geometric properties: The depth buffer algorithm reduces the objects to a series of polygons and tests them for visibility. The polygon, in geometric terms is a surface, which is represented by the equation ax + by + cz +d =0 for any point (x,y,z) that lies on the surface (or plane in geometric terminology). A point that does not satisfy the equation lies outside the plane. A point that gives a negative value for the equation lies on the back face of the plane (since x,y coordinates cannot be negative, it implied z becomes negative). Such back faces like the back of a cube or a pyramid or some similar shape,

can be totally removed from the calculations then reducing the efforts considerably.

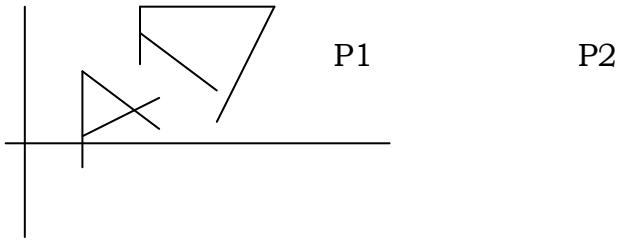
ii) Overlap tests: Common sense gives us one simple idea. An object can obscure another only if (a) one of them is at a farther distance than another – obviously two objects standing side by side cannot obscure each other

iii) Even if they are at unequal distances, they can obscure only if either their x or y coordinates overlap. For example the two polygons in the given figure cannot obscure each other irrespective of how far or near is each of them than the other.



The minimax tests guarantee just this. If the minimum x coordinate of one polygon is larger than the maximum x coordinate of another (P2 and P1 respectively of the figure) and similarly the max is coordinate of one is less than the minimum is coordinate of another (P1 and P2 respectively). The two figures cannot overlap no matter what their z coordinates are. This test allows us to trivially avoid testing a few pairs of polygons for obscuring each other.

However it should be noted that failure to pass the minimax test does not always imply that they obscure consider the following case



Here though P1 and P2 coordinates overlap, they still do not obscure each other. Testing such instances need more elaborate computations.

11.5 Scan Line Coherence Algorithms

Scan line algorithms solve the hidden surface problem, one scanline at a time. They traverse the picture the picture space from top to bottom, one line at a time removing all the hidden surface along that scan line.

The simplest of them can be the depth buffer algorithm itself. Since the algorithm has to consider every pixel, it can take care of pixels along each scan line at a time and then go to the next line and so on.

Scan line coherence algorithm:

For each scan line perform the following steps.

a. For every pixel on a scan line, set depth value to 1.0 and intensity to the back ground value.

b. For each polygon in the view scene, find all pixels of the scan line under consideration that lie within the polygon. For each of them

i. Find the depth z of the polygon at the point.

ii. If Z < depth [x], set depth[x] to z and the intensity to the intensity of the polygon.

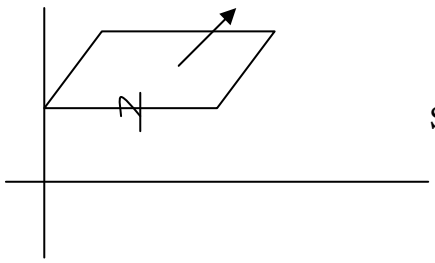
c. Once all polygons have been taken care of, the pixels contain the intensity values that are to be displayed. This algorithm when used in conjunction with Y-X scans conversion forms the simplest of scan line coherence algorithms.

11.6 Span – Coherence algorithm

Another property of coherence that can be made use of in scan conversion is the one-dimensional form of area coherence, called span coherence. If a pixel is inside a polygon, it's neighbors also lie inside the polygon. This holds good upto a "Span" once the Span is detected, all pixels within the span can be set to the intensity value of the polygon and the next comparison can take place at the end of the span. This reduces the computation by a very large amount, especially if the no. of polygons is limited.

The concept of spans can be considered in a simplistic manner by the following example.

In the 3-dimensional space, each scan line produces a plane. I.e. each scan line is for a particular value of y. A plane with this value of y as a constant over different values of x and z form a plane.



If one travels along this scan line, the plane intersects one/more polygons at different points. If these points of intersection are noted and are sorted in the increasing order of x, we get a sort of xz algorithm which gives the list of intersections with different polygons.

Taking them in pairs, just as in the XY algorithm, one can convert the entire plane into several spans.

i. Spans that do not lie within any polygon, the pixels can be set to the background intensity.

ii. Spans that lie within a single polygon. All of them can be set to the intensity of the polygon.

iii. Spans that are intersected by 2 or more polygons. In such spans, the pixel values can be set to the intensity of the nearest polygon.

The algorithm implements the details in the following order.

i. A single active edge list, sorted by x, contains the intersection (and the intensity values) of all polygons that intersect the scan line.

ii. During the actual scan conversion, the process starts from the left to the right. Initially, the pixels are at the back ground intensity. The first node on the list indicates the first polygon being activated i.e. from no on the pixels will be inside the polygon. The next node can be either an entry into a new polygon or the exit from a polygon. (The concept of entry or exit into or out of polygons is stored into the nodes by setting a particular bit to 1 or 0 for entry or exit at the time of creating the list – while taking not of the intersections.

iii. Whenever an entry into a polygon occurs, all polygons that have been currently activated are checked to find the one that is nearest to the viewer and the pixels from then onwards are set to that polygons intensity. While exiting from a polygon, again that polygon is deactivated and the pixel intensity value is recalculated.

This basic algorithm can make use of coherence in many other ways for farther improvisation of efficiency. For example, the intersections need not be calculated for all the scan lines (or scan planes). If a polygon cuts a scan plane at (x,z) say, then the intersection of the same polygon in the next immediate scan plane will be either at (x,z) itself or at one pixel distance from (x,z) in any direction. This can be used to avoid actual calculation of the intersection and in fact, if the direction of the polygon edge is known, it's intersection with next scan plane can be found at accurately by looking at the precise pixel indicated by the direction.

11.7 Area – Coherence Algorithms

The idea of coherence can be extended in both directions. I.e. just as a pixel will have, most probably, the intensity of its left or right neighbor, the coherence can be extended to the other direction as well. This bi-directional or area coherence was made use of by Warnock in his algorithm, known by his name.

11.8 Warnock's Algorithm

This is one of the class of "area" algorithms. It tries to solve the hidden surface problem recursively. The algorithm proceeds on the following lines.

i. Try to solve the problem by taking the entire screen as one window. If no polygons overlap either in x or y or even if they do, overlap so that they do not obscure, then return the screen.

ii. If the problem is not easily solvable in step (i) the algorithm divides the screen into 4 equal parts and tries to apply step (i) each of them. If it is not solvable, again divides into smaller windows and so on.

iii. The recursive process continues till each window is trivially solvable or one endsup with single pixels.

We have still not described how the actual "solution" is done. To do this, in any window, the algorithm classifies the polygons into three groups

i) Disjoint Polygons: Polygons that do not overlap in the window and hence can be trivially passed.

ii) A bigger and a smaller polygon overlapping so that the smaller one will be completely blocked by the bigger one (if the Z of the larger polygon is smaller than Z of the smaller one).

iii) Intersected polygons: Polygons that partly obscure each other.

Polygons that fall into category (i) and (ii) are removed at each level. If the remaining polygons can be easily solved, the recursive process stops at that level, else the process continues (with the polygons of category (i) and (ii) removed).

Since at each recursive level a few polygons are removed, as the windows become smaller and smaller with the advance of recursion, the list of polygons falling into them also reduces and hopefully the problem of hidden surfaces gets solved trivially.

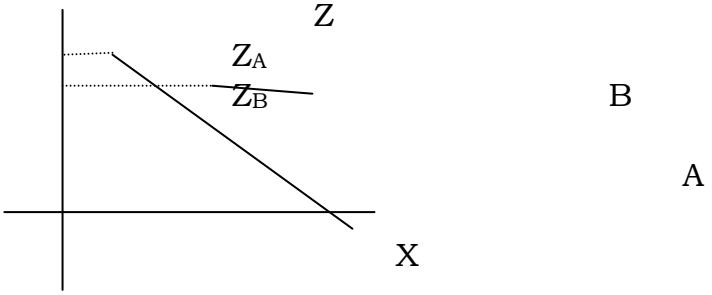
One main draw back of algorithm is that the windows get divided into smaller and smaller rectangles. In many cases it would be efficient if one can divide the window roughly in the shape of the polygons themselves. Such an algorithm, developed by Wieler and Atherton, was found more efficient, though more complex in terms of larger complexities of recursive divisions and clippings.

11.9 Priority Algorithms

In contrast to the scan line and area coherence algorithms, priority algorithms try to discover the depth relations first and then perform the xy calculations only after the visibility has been established. They are similar to the priority algorithms of scan conversion.

Remember the painters algorithm? A painter drawing a painting on a canvas simply keeps painting them beginning from the farthest object. As and when a nearer object gets painted, the hidden areas of the farther objects automatically get covered by the new object.

Similarly if one begins scan converting the polygons beginning with the farthest polygon, the hidden lines and hidden surfaces automatically get eliminated. However, if the polygons in the priority list overlap in depth, the things become more complex. Look at the following figure:



Simply sorting them on the basis of Z max would make computations complicated because for certain scan lines. A is nearer than B

but for certain others B is nearer than A. Hence the priority list, prepared based only on the depths will have to be rearranged as follows.

Consider the last polygon in the A (Say). If it has no overlaps in depths with its predecessors, then it has no overlaps with any other polygons and can remain at the end. Other wise, if it has any depth overlaps with one or more polygons, denoted by the set {b}, then we have to again check if any specific polygon B from this set is obscured by A. If yes, then B has no business to be in that priority since A, which is obscuring B, should have a higher priority than B. Corresponding modifications are to be made to the list.

Based on the considerations, in the above figure A should have a higher priority than B, though the Zmax of B is less than that of A.

The question is how to find out the relation "A obscures B"? Apply the following steps in the same order to ascertain that A does not obscure B.

(a) Depth minimax test should indicate that A and B do not overlap in depth and B is closer to the viewpoint than A. This test is implemented by initially sorting by depth all polygons and by the way A and {b} are selected.

(b) Minimax test in xy should indicate that A and B do not overlap in X or Y.

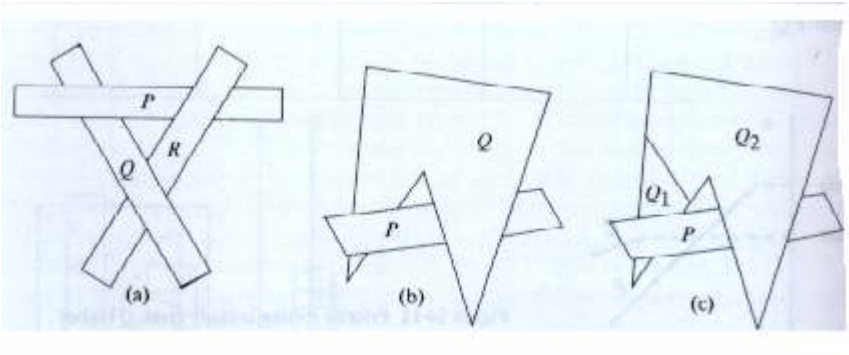
(c) All vertices of A should be farther from the view point than the plane of B. This can be implemented by substituting x,y coordinates of a into the plane equation of B and solving for the depth of B.

(d) All vertices of B should be closer to the viewpoint than the plane of B.

(e) A full overlap test should indicate that A and B do not overlap in x or y.

The order is not very important, except that any one of the tests being true indicates that A does not obscure B. Since the latter tests are more involved, it is desirable that the order is followed so that one can avoid the latter tests if possible.

Is the "A obscures B" relation sufficient condition to sort polygons? Look at the following sequence of figure.



Obviously the algorithm fails to give a clearcut sequence of polygons. In such cases, it is desirable to subdivide one/more polygons so that the "Chain reactions" are avoided.

Review questions:

1. State painters algorithm in 2-3 lines.
2. What is the main difficulty of the scan line algorithm?
3. What is the concept of overlap testing?
4. If tow or more objects fail in overlap testing, does it mean they always obscure at least in some regions.
5. Explain the concept of coherence of pixels.
6. Name the algorithm that work on the concept of area coherence
7. State one method of improving the recursive efficiency of the above algorithm.
8. If a portion of polygon A obscure B, a portion of B obscures C and so on so that they form a cyclic loop, the concept of Zmax fails? How do you apply the scan conversion algorithm in such a case?

Answers:

1. Start painting from the object that is farthest from the viewer. As and when new objects are painted, the earlier objects that are obscured by the nearer objects automatically get removed- either in full or in those regions where they are invisible.
2. It is computationally insensitive.
3. Two objects can obscure each only if Zmax of one is greater than the Zmax of the other. Even then, they overlap only if they overlap in either x or y coordinates. I.e. the maximum y of one is greater than the minimum y of the other or the maximum x of one is greater than the minimum x of the other and this holds for both the objects.
4. No, It depends on their actual shapes and placements

5. The general property of coherence is that neighboring pixels share properties i.e. if a particular pixel belongs to a particular object, most probably it's neighboring pixels also lie in the same object. This applied over certain "Spans".

6. Warnock's Algorithm

7. By dividing the screen recursive not into rectangles but into areas similar to the shape of the polygons.

8. By dividing on/more of these polygons into similar polygons.

Appendix A
Practical Programs Solved

```
/* program to initialize the graph and draw a line*/
#include<graphics.h>
#include<conio.h>
void main()
{
int gd=DETECT;   /* Detects the graph driver dynamically*/
int gm;       /* for graph mode */
initgraph(&gd,&gm,""); /* graph driver, graph mode and path has to be passed
      as parameters. The empty path is specified means the path will be taken
      dynamically after searching in the computer. Otherwise we need to
      specify the path where bgi directory is stored in the computer */
line(10,10,200,200);      /* this function draws a line from starting co-ordinates
      (10,10) to the target co-ordinates (200,200). These co-ordinates are
      specified in terms of pixels */
getch();
closegraph();             /* closes the graph mode */
}

/* Program to draw a rectangle, lines as its diagonals and a circle */
#include<graphics.h>
#include<conio.h>
void main()
{
int gd=DETECT;
int gm;
initgraph(&gd,&gm,"");
rectangle(10,10,200,200);      /* draws a rectangle */
line(10,10,200,200);           /* draws a line on the main diagonal*/
line(10,200,200,10);           /* draws a line on off diagonal */
circle(105,105,95);            /* draws a circle taking (105,105)
                        as centre co-ordinates and 95 as radius
                        all the dimensions are in pixels */
getch();
closegraph();             /* closes the graph mode */
}


/* Program to draw ellipses and arcs */
```

```
#include<graphics.h>
#include<conio.h>
void main()
{
int gd=DETECT;
int gm;
initgraph(&gd,&gm,"");

setcolor(1);  /* sets the drawing color as blue */
ellipse(getmaxx()/2, getmaxy()/2, 0,360,80,50);
            /* draws an ellipse taking centre of the screen as its centre, 0 as
starting angle and 360 as ending angle and 80 pixels as x-radius, 50 pixels as
y radius*/
setcolor(4);  /* sets the drawing color as red */
ellipse(getmaxx()/2, getmaxy()/2, 90,270,50,80);
            /* draws half the ellipse starting from 90 degree angle and
       ending at 270 degree with 50 pixels as x-radius and 80 pixels as y-radius
in red color */
setcolor(5);  /* sets the drawing color as pink */
arc(getmaxx()/2,getmaxy()/2, 0,180,100);
            /* arc with centre of the screen as its center and 100 pixels as
radius. It starts at an angle 0 and ends at an angle 180 degrees, i.e., half circle
*/
setcolor(9);  /* sets the drawing color as light blue */
arc(300,200, 20,100,70);
            /* arc with (300,200) as its center and 70 pixels as radius. It starts
at an angle 20 and ends at an angle 100 degrees */

getch();
closegraph();              /* closes the graph mode */
}


/* Program to demonstrate rectangles using putpixel and filling them with
different fill effects */
#include<graphics.h>
#include<conio.h>
#include<stdlib.h>
#include<dos.h>

void main()
{
```

```
int gm,gd=DETECT;
int x1,x2,y1,y2,c,i;

initgraph(&gd,&gm,"");
while(!kbhit()) /* until pressing any key this loop continues */
{
/* to draw rectangle co-coordinates are taken randomly */
x1=rand()%getmaxx();
x2=rand()%getmaxx();
y1=rand()%getmaxy();
y2=rand()%getmaxy();

if(x1>x2)
{
c=x1;       /* exchange of x1 and x2 when x1 is > x2 */
x1=x2;
x2=c;
}
if(y1>y2)
{
c=y1;       /* exchange of y1 and y2 when y1 is > y2 */
y1=y2;
y2=c;
}
c=rand()%16;
/* for rectangle using putpixel */
for(i=x1;i<=x2;++i)
{
putpixel(i,y1,c);
delay(1);
}
for(i=y1;i<=y2;++i)
{
putpixel(x2,i,c);
delay(1);
}
for(i=x2;i>=x1;--i)
{
putpixel(i,y2,c);
delay(1);
}
for(i=y2;i>=y1;--i)
```

```
{
putpixel(x1,i,c);
delay(1);
}
setfillstyle(rand()%12,rand()%8); /* setting the random fill styles and colors */
floodfill(x1+1,y1+1,c);
delay(200);  /* to draw the pixels slowly */
}
getch();
closegraph();                /* closes the graph mode */
}
```

/* Program to demonstrate viewport, clipping and lines with different colors, line styles and co-ordinates */

```
#include<graphics.h>
#include<conio.h>
#include<stdlib.h>
#include<dos.h>
#include<stdio.h>
void main()
{
int gm, gd=DETECT;
int x1,x2,y1,y2,c,i;
clrscr();
printf("enter starting co-ordinates of viewport  (x1,y1)\n");
scanf("%d%d",&x1,&y1);
printf("enter ending co-ordinates of viewport (x2,y2)\n");
scanf("%d%d",&x2,&y2);

initgraph(&gd, &gm,"");

rectangle(x1,y1,x2,y2);/*to show the boundary of viewport */
setviewport(x1,y1,x2,y2,1); /* view port is set and any  drawing now onwards
            must be drawn within the viewports only */

while(!kbhit()) /* until pressing any key this continues */
{
/*  rectangle coordinates are taken randomly */
x1=rand()%getmaxx();
```

```
x2=rand()%getmaxx();
y1=rand()%getmaxy();
y2=rand()%getmaxy();
setlinestyle(rand()%10,rand()%10,rand()%20);
setcolor(rand()%16); /* to set the line color */
line(x1,y1,x2,y2); /* to draw the line */
delay(200);
}
getch();
closegraph();                    /* closes the graph mode */
}
/* Program to demonstrate text and its settings */
#include<graphics.h>
#include<conio.h>
#include<stdlib.h>
#include<dos.h>
#include<stdio.h>
void main()
{
int gm, =DETECT;
initgraph(&gd, &gm ,"");

setcolor(5);
settextstyle(4,0,5); /* sets the text style with
                        font, direction and char size */
moveto(100,100); /* takes the CP to 100,100 */
outtext("Bangalore is");

setcolor(4);
settextstyle(3,0,6);
moveto(200,200);
outtext("Silicon ");

setcolor(1);
settextstyle(5,0,6);
moveto(300,300);
outtext("Valley");

setcolor(2);
settextstyle(1,1,5);
outtextxy(150,50,"Bangalore is");
```

```
getch();
}
```

/* Program to draw a car. The different graphical functions are used to draw different parts of the car */

```
#include<stdio.h>
#include<graphics.h>

main()
{
  int x,y,i, choice;
  unsigned int size;
  void *car;

  int gd=DETECT, gm;
  initgraph(&gd, &gm," ");

  do
  {
  cleardevice();
   printf("1:BODY OF THE CAR\n");
   printf("2:WHEELS OF THE CAR\n");
   printf("3:CAR\n");
   printf("4:QUIT");
   printf("\nEnter your choice\n");
   scanf("%d",&choice);
   switch(choice)
   {
    case 1:  initgraph(&gd,&gm," ");
             line(135,300,265,300);
             arc(100,300,0,180,35);
             line(65,300,65,270);
             line(65,270,110,220);
             line(110,220,220,220);
             line(140,220,140,215);
             line(180,220,180,215);
             line(175,300,175,220);
             line(120,215,200,215);
             line(220,220,260,250);
             line(260,250,85,250);
             line(260,250,345,275);
             arc(300,300,0,180,35);
             line(345,300,345,275);
```

```
            line(335,300,345,300);
            getch();
            cleardevice();
            break;

 case 2:    initgraph(&gd,&gm,"");
            circle(100,300,25);
            circle(100,300,13);
            circle(300,300,25);
            circle(300,300,13);
            getch();
            cleardevice();
            break;

 case 3:   initgraph(&gd,&gm," ");
           outtextxy(150,40,"MARUTI 800");
            circle(100,300,25);
            circle(100,300,13);
            line(135,300,265,300);
            arc(100,300,0,180,35);
            line(65,300,65,270);
            line(65,270,110,220);
            line(110,220,220,220);

            line(140,220,140,215);
            line(180,220,180,215);
            line(175,300,175,220);
            line(120,215,200,215);
            line(220,220,260,250);
            line(260,250,85,250);
            line(260,250,345,275);
            arc(300,300,0,180,35);
              circle(300,300,25);
            circle(300,300,13);
            line(345,300,345,275);
            line(335,300,345,300);
            getch();
            cleardevice();
            break;

 case 4:   exit(0);
 }
}
while(choice!=4);
closegraph();
```

```
  getch();
}


/* Program to draw a 2D vertical bars */
#include<graphics.h>
#include<stdio.h>
#include<conio.h>
void main()
{
int gm, gd=DETECT, i;
int n, a[20],t;
clrscr();
do
{
t=0;
printf("\n enter n > 0 and <=20 \n");
scanf("%d",&n);
if(n<=0 || n>20)
{
printf(" enter the limit correctly press any key to continue...\n");
getch();
t=1;
}
}while(t==1);
for(i=0;i<n;++i)
{
do
{
t=0;
printf("\n enter value( >0 and <=400) %d : ",i+1);
scanf("%d",&a[i]);
if(a[i]<=0 || a[i]>400)
{
printf(" enter the limit correctly press any key to continue...\n");
getch();
t=1;
}
}while(t==1);
}
initgraph(&gd, &gm,"");
line(10,400,n*25+20,400);
```

```
for(i=0;i<n; ++i)     /* draws n bars with different filling
                      colors and patterns */
        {
        setfillstyle(i+1,i+1);         /* fill pattern and color for each
                                of the bars changes dynamically*/
        rectangle(i*25+25,400-a[i],i*25+40,400);
                                 /* the y-coordinate of each bars changes dynamically
and
                                  between each bars 10 pixels gap is maintained */

        floodfill(i*25+30,399,15);/* fills the rectangular bar with
                                  specified fill styles and colors*/
        }

getch();
closegraph();                 /* closes the graph mode */
}


/* Program to draw 2D horizontal bars with fill effects */
#include<graphics.h>
#include<stdio.h>
#include<conio.h>
void main()
{
int gm, gd=DETECT, i;
int n, a[20],t;
clrscr();
do
{
t=0;
printf("\n enter n > 0 and <=20 \n");
scanf("%d", &n);
if(n<=0 || n>20)
{
printf(" enter the limit correctly press any key to continue...\n");
getch();
t=1;
}
}while(t==1);
for(i=0;i<n;++i)
{
```

```
do
{
t=0;
printf("\n enter value( > 0 and <= 600) %d : ",i+1);
scanf("%d",&a[i]);
if(a[i]<=0 || a[i]>600)
{
printf(" enter the limit correctly press any key to continue...\n");
getch();
t=1;
}
}while(t==1);
}
initgraph(&gd,&gm,"");
line(40,10,40,n*25+35);
for(i=0;i<n;++i)      /* draws n bars with different filling colors and patterns */
        {
        setfillstyle(i+1,i+1);        /* fill pattern and color for each of the bars
changes
                                        dynamically*/
        rectangle(40,i*25+25,a[i]+40,i*25+40);
                                /* the x-coordinate of each bars changes dynamically
and
                                  between each bars 10 pixels gap is maintained */

        floodfill(41,i*25+30,15);/* fills the rectangular bar with specified fill styles
and
                                        colors*/
        }

getch();
closegraph();                   /* closes the graph mode */
}


/* Program to illustrate the pie chart */
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void main()
{
int gd=DETECT, gm;
```

```
int a,b,c,d,s;
clrscr();
printf("\n enter 4 values\n");
scanf("%d%d%d%d",&a,&b,&c,&d);

/* re-calculating the contribution of each value in the circle area */
s=a+b+c+d;
a=a*360/s;
b=b*360/s;
c=c*360/s;
d=d*360/s;
initgraph(&gd, &gm,"");
setfillstyle(1,2);
pieslice(300,250,0,a,100);
setfillstyle(1,4);
pieslice(300,250,a,a+b,100);
setfillstyle(1,6);
pieslice(300,250,a+b,a+b+c,100);
setfillstyle(1,8);
pieslice(300,250,a+b+c,360,100);
getch();
}



/* Program to demonstrate movement of objects */
#include<graphics.h>
#include<conio.h>
#include<dos.h>
#include<alloc.h>

void main()
{
int gm, gd=DETECT, i;
char *z; /* to get the image */
initgraph(&gd, &gm,"");
rectangle(10,10,50,50);
z=malloc(imagesize(10,10,50,50)); /* memory allocation to the
                        variable to store the image */
getimage(10,10,50,50,z); /* gets the image into variable z */
putimage(10,10,z,1);/* erases the image from the original place */
```

```
for(i=0;i<=getmaxy()/2-50;++i)
{
putimage(getmaxx()/2,getmaxy()/2-i,z,2); /* Vertically up */
putimage(getmaxx()/2,getmaxy()/2+i,z,2); /* Vertically down */
putimage(getmaxx()/2-i,getmaxy()/2,z,2); /*  Horizontally left */
putimage(getmaxx()/2+i,getmaxy()/2,z,2); /* Horizontally right */

putimage(getmaxx()/2-i,getmaxy()/2-i,z,2); /* To top-left corner */
putimage(getmaxx()/2-i,getmaxy()/2+i,z,2); /* to top-right corner */
putimage(getmaxx()/2+i,getmaxy()/2-i,z,2); /* To bottom-left corner */
putimage(getmaxx()/2+i,getmaxy()/2+i,z,2); /* To bottom-right corner */

delay(5);                    /* image will be written and    after some delay it will
be erased and re-written in next location based on delay it looks like moving
fast or slow */

putimage(getmaxx()/2,getmaxy()/2-i,z,1);
putimage(getmaxx()/2,getmaxy()/2+i,z,1);
putimage(getmaxx()/2-i,getmaxy()/2,z,1);
putimage(getmaxx()/2+i,getmaxy()/2,z,1);
putimage(getmaxx()/2-i,getmaxy()/2-i,z,1);
putimage(getmaxx()/2-i,getmaxy()/2+i,z,1);
putimage(getmaxx()/2+i,getmaxy()/2-i,z,1);
putimage(getmaxx()/2+i,getmaxy()/2+i,z,1);
}

/* write the image in target position*/

putimage(getmaxx()/2,getmaxy()/2-i,z,2);
putimage(getmaxx()/2,getmaxy()/2+i,z,2);
putimage(getmaxx()/2-i,getmaxy()/2,z,2);
putimage(getmaxx()/2+i,getmaxy()/2,z,2);
putimage(getmaxx()/2-i,getmaxy()/2-i,z,2);
putimage(getmaxx()/2-i,getmaxy()/2+i,z,2);
putimage(getmaxx()/2+i,getmaxy()/2-i,z,2);
putimage(getmaxx()/2+i,getmaxy()/2+i,z,2);

getch();
}
```

```
/*  Program to demonstrate moving  wheel */

#include<stdio.h>
#include<dos.h>
#include<math.h>
#include<conio.h>
#include<graphics.h>
void main()
{
int gd=DETECT,gm,i,x,y,r,j;
float x1,y1;
clrscr();
/* Initialization of center co-ordinates of the circle for wheel */
x1=50;
y1=50;
/* For movement of wheel starting from 500 pixels along x-axis */
j=500;
initgraph(&gd,&gm,""); /* Graphics is initialized */
x=x1;
y=y1;
r=sqrt(x*x+y*y);/* Radius of wheel */
while(1)
{
/* for every circle of the wheel */
for(i=1;i<=360;++i)
{
x=x1*cos(i*3.142/180)+y1*sin(i*3.142/180);/* Calculating the points to draw
line */
y=y1*cos(i*3.142/180)-x1*sin(i*3.142/180);
setcolor(4);
/* Wheels outer circle */
circle(j,200,r);
circle(j,200,r+5);
/* Center of the wheel */
circle(j,200,4);
circle(j+x,y+200,2);
circle(j-x,200-y,2);
line(j,200,x+j,y+200);
line(j,200,j-x,200-y);
delay(30);
/* Erasing the line and circle that makes movement effect */
setcolor(0);
```

```
circle(j,200,r+5);
circle(j,200,r);
circle(j,200,4);
circle(j+x,y+200,2);
circle(j-x,200-y,2);
line(j,200,x+j,y+200);
line(j,200,j-x,200-y);
j--;
if(j<=50)
break;
}
if(j<=50)
break;
}
/* Rewriting the wheel at target position */
setcolor(4);
circle(j,200,r);
circle(j,200,4);
circle(j,200,r+5);
circle(j+x,y+200,2);
circle(j-x,200-y,2);
line(j,200,x+j,y+200);
line(j,200,j-x,200-y);

getch();
closegraph();
}


/* Program to demonstrate the moving car */

#include<stdio.h>
#include<dos.h>
#include<alloc.h>
#include<stdlib.h>
#include<conio.h>
#include<graphics.h>

void main()
{
int gd=DETECT,gm,i,x,y,r,j;
void *z;
```

```
clrscr();
initgraph(&gd,&gm,"");


/* creation of car image */
circle(30,130,8);
circle(30,130,2);
circle(110,130,8);
circle(110,130,2);
arc(30,130,0,180,11);
arc(30,130,60,180,14);
line(16,130,19,130);

arc(110,130,0,180,11);
arc(110,130,0,120,14);
line(124,130,121,130);

arc(70,130,0,180,30);
arc(70,130,20,160,33);
arc(70,130,40,140,24);

line(41,130,99,130);
/* Getting the image of the car into variable 'z' after allocating memory */
z=malloc(imagesize(15,90,145,140));
getimage(15,90,145,140,z);
/* Erasing the car image from its original position */
putimage(15,90,z,1);
while(!kbhit()) /* This loop continues until pressing any key */
{
/* Movement of car from left to right horizontally */
y=rand()%400+50; /* Y co-ordinate is taken randomly within the screen co-
ordinates along y-axis */
for(i=getmaxx()-100;i>=10;--i)
{
putimage(i,y,z,2);
delay(rand()%5);
putimage(i,y,z,1);
}
}
putimage(i,y,z,2);
getch();
closegraph();
```

```
}
```

/* Program to demonstrate the flag flying image through animation. The program makes use of function for three different effects in the flag display */

```
#include <stdio.h>
#include<graphics.h>
#include <conio.h>
#include <dos.h>

void flag1(); /* to display the flag straight */
void flag2(); /*to display the flag little slanting upwards */
void flag3(); /* to display the flag little slanting downwards */
/* displaying flag1,3 and 2 continuously, we get the flag flying image. */
void main()
{
  int gd = DETECT, gm;
  int i, stangle, endangle;
  initgraph(&gd, &gm,"");
  while(!kbhit())
  {
    cleardevice();
    flag1();
    delay(250);
    cleardevice();
    flag3();
    delay(300);
    cleardevice();
    flag2();
    delay(300);
  }
  getch();
  closegraph();
  return;
}


void flag1()
{
  int i;
  setfillstyle(SOLID_FILL,6);
  bar(50,50,300,100);
```

```
  setfillstyle(SOLID_FILL, WHITE);
  bar(50,100,300,150);
  setfillstyle(SOLID_FILL, GREEN);
  bar(50,150,300,200);
  /* the three color bars have been drawn. */
  /*drawing the flag pole */
  setfillstyle(SOLID_FILL,8);
  bar(40,20,50,460);
  for(i=0;i<=360;i+=15)
    pieslice(175,125,i,i+15,25);
}

void flag2()
{
    int i;
  int r1[8] = {50,50,300,75,300,125,50,100};
  int r2[8] = {50,100,300,125,300,175,50,150};
  int r3[8] = {50,150,300,175,300,225,50,200};
  setcolor(0);
  setfillstyle(SOLID_FILL,6);
  fillpoly(4,r1);
  setfillstyle(SOLID_FILL, WHITE);
  fillpoly(4,r2);
    setfillstyle(SOLID_FILL, GREEN);
  fillpoly(4,r3);
  setfillstyle(SOLID_FILL,8);
  bar(40,20,50,460);
  setcolor(9);
  setfillstyle(SOLID_FILL, WHITE);
  for(i=0; i<=360; i+=15)
   pieslice(175,137,i,i+15,25);

}

void flag3()
{
  int i;
  int r1[8] = {50,50,300,25,300,75,50,100};
  int r2[8] = {50,100,300,75,300,125,50,150};
  int r3[8] = {50,150,300,125,300,175,50,200};
  setcolor(0);
  setfillstyle(SOLID_FILL,6);
```

```
  fillpoly(4,r1);
  setfillstyle(SOLID_FILL,WHITE);
  fillpoly(4,r2);
    setfillstyle(SOLID_FILL,GREEN);
  fillpoly(4,r3);
  setfillstyle(SOLID_FILL,8);
  bar(40,20,50,460);
  setcolor(9);
  setfillstyle(SOLID_FILL, WHITE);

  for(i=0; i<=360; i+=15)
   pieslice(175,113,i,i+15,25);

  }
```

```
/* Program to animate a man walking with umbrella. */
#include<stdio.h>
#include<dos.h>
#include<conio.h>
#include<graphics.h>
#include<alloc.h>

void main()
{
int gd=DETECT, gm, i;
void *z;
initgraph(&gd, &gm,"");
/* Construction of man and umbrella in his hand */
circle(50,28,4);
line(50,32,50,56);

line(50,40,42,48);
line(50,40,58,48);
line(50,56,42,64);
line(50,56,58,64);
line(58,48,64,40);
line(64,40,64,11);
arc(64,20,0,180,17);
arc(64,30,30,150,20);
```

```
/* getting the image of the man with umbrella in a variable */
z=malloc(imagesize(5,3,90,90));
getimage(5,3,90,90,z);
putimage(5,3,z,1);

/* Making animation of man walk */
for(i=10;i<=getmaxx()-100;++i)
{
putimage(i,100,z,2);
delay(10);
putimage(i,100,z,1);
}
putimage(i,100,z,2);

getch();
closegraph();
}


/* program to rotate a circle between two ends of a line */
/* It makes use of two functions one for drawing the circle and the other for
erasing the circle */
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>
#include<graphics.h>

#define pi 2*3.1457

void draw_circle(int xa, int ya, int r);
void draw_circle2(int xa, int ya,   int r);

void main()
{
  int gd=DETECT, gm;
  float i, color;
  initgraph(&gd, &gm,"");
  setcolor(YELLOW);
  outtextxy(250,50,"ROTATE A CIRCLE ");
  setcolor(RED);
```

```
    for(i=50;i<585;i+=2)
  {
    line(10,200,640,200);
    draw_circle(i,200,50);
    draw_circle2(i-2,200,50);
  }
  getch();
}

/* FUNCTION FOR DRAWING THE CIRCLE */
void draw_circle(int xa, int ya, int r)
{
   float theta,x,y;
   int color;
   for(theta=0;theta<pi;theta=theta+0.01)
   {
      x = xa + r * cos(theta);
      y = ya + r * sin(theta);
      color = random(getmaxcolor());
         putpixel((int)x,(int)y, color);
   }
}
/* FUNCTION FOR ERASING THE CIRCLE */
void draw_circle2(int xa, int ya, int r)
{
   float theta, x, y;
   int color=0;
   for(theta=0;theta<pi; theta=theta+0.01)
   {
      x = xa + r * cos(theta);
      y = ya + r * sin(theta);
      putpixel((int)x,(int)y, color);
   }
  delay(5);
}
```

/* Program to show the text animation on the screen. The texts are made to move from bottom of the screen to top one by one and they become small at end */

```
# include<graphics.h>
```

```
# include<conio.h>
# include<dos.h>

void main()
{
int gd=0,gm,i,n,j,t,f;
char
a[20][80]={"PROGRAMMING","SKILLS","THROUGH","C","ANMOL","PUBLICATIO
NS","BY","B.M.HAVALDAR"};
 initgraph(&gd,&gm,"");
 settextjustify(1,1);
 setusercharsize(10,5,1,1);
     i=getmaxy();
     n=i;
     j=0,t=-1,f=26;
      while(!kbhit()) /* Animation continues till pressing any key */
  {
   if (i<=-50)
   {
     i=n;
     j++;
     f=26;
   }
   else i-=1;
   if(j==8)  /* The array of strings pointer is reset to starting */
     j=0;
   settextstyle(2,0,f);
   setcolor(f);
   outtextxy(getmaxx()/2,i,a[j]);
   delay(5);

  cleardevice();
  if( (i%20==0))
  f=f+t*1;
  if(f<=1)
   t=1;
  if(f>=26)
  t=-1;
 }
 getch();
 closegraph();
}
```

```
/* Program to Generate a Line using Digital Differential Algorithm (DDA) */
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
#include<math.h>

/* function for plotting the points and drawing the line */
void ddaline(float x1,float y1,float x2,float y2)
 {
   int i, color=5;
   float x, y, xinc, yinc, steps;
   steps = abs(x2-x1);
   if (steps < abs(y2-y1))
     steps= abs(y2-y1);
   xinc=(x2-x1)/steps;
   yinc=(y2-y1)/steps;

   x=x1;
   y=y1;
   putpixel((int)x,(int)y, color);
   for (i=1;i<=steps; i++)
   {
   putpixel((int)x,(int)y,color); /*Plots the points with specified color */
   x=x+xinc;
   y=y+yinc;
   }
   }

/* The main program that inputs line end points and passes them to ddaline()
function */
void main()
{
 int gd = DETECT, gm, color;
 float x1,y1,x2,y2;
 printf("\n enter the end points :\n");
 scanf("%f %f %f %f",&x1,&y1,&x2,&y2);
 clrscr();
 initgraph(&gd, &gm, "c:\\tc\\bgi");
 ddaline(x1,y1,x2,y2);
 getch();
```

```
 closegraph();
 }

/* Program to generate a line using Bresenham's algorithm   */
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<graphics.h>

void brline(int,int,int,int);

void main()
{
int gd=DETECT, gmode;
int x1,y1,x2,y2;
printf("\n enter the starting point x1,y1 :");
scanf("%d%d",&x1,&y1);
printf("\n enter the  ending point x2, y2 :");
scanf("%d%d",&x2,&y2);
clrscr();
initgraph(&gdriver, &gmode,"");
brline(x1,y1,x2,y2);
getch();
closegraph();
}

/* Function for Bresenham's line */

void brline(int x1,int y1,int x2,int y2)
{
int e,l,xend;
int color;
float dx,dy,x,y;
dx=abs(x2-x1);
dy=abs(y2-y1);
if(x1>x2)
{
        x=x2;
        y=y2;
        xend=x1;
}
else
```

```
{
        x=x1;
        y=y1;
        xend=x2;
}
e=2*dy-dx;

while(x<xend)
{
        color=random(getmaxcolor());
        putpixel((int)x,(int)y,color);
        if(e>0)
        {
                y++;
        e=e+2*(dy-dx);
}
else
        e=e+2*dy;
        x++;
}
}
```

```
/* Program to demonstrate circle using DDA algorithm */
#include<graphics.h>
#include<conio.h>
#include<dos.h>
#include<alloc.h>
#include<math.h>

void main()
{
int gm,gd=DETECT,i;
int x,y,x1,y1,j;
initgraph(&gd,&gm,"");
x=40; /* The co-ordinate values for calculating radius */
y=40;
for(i=0;i<=360;i+=10)
{
        setcolor(i+1);
        x1=x*cos(i*3.142/180)+y*sin(i*3.142/180);
        y1=x*sin(i*3.142/180)-y*cos(i*3.142/180);
```

```
        circle(x1+getmaxx()/2,y1+getmaxy()/2,5); /* center of the circle is center
                                                    of the screen */
        delay(10);
}
getch();
}
```

/* Program to Implement Bresenham's Circle Drawing Algorithm */

```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
#include<math.h>
#include<dos.h>
```

/* Function for plotting the co-ordinates at four different angles that are placed at equal distances */

```
void plotpoints(int xcentre, int ycentre, int x, int y)
 {
   int color=5;
   putpixel(xcentre+x, ycentre+y, color);
   putpixel(xcentre+x, ycentre-y, color);
   putpixel(xcentre-x, ycentre+y, color);
   putpixel(xcentre-x, ycentre-y, color);

   putpixel(xcentre+y, ycentre+x, color);
   putpixel(xcentre+y, ycentre-x, color);
   putpixel(xcentre-y, ycentre+x, color);
   putpixel(xcentre-y, ycentre-x, color);
   }
```

/* Function for calculating the new points for (x ,y) co-ordinates.*/

```
void cir(int xcentre, int ycentre, int radius)
{
                int x,y,p;
                x=0; y=radius;
                plotpoints (xcentre, ycentre, x, y);
                p=1-radius;
```

```
                while(x<y)
                {
                        if (p<0)
                          p=p+2*x+1;
                          else
                            {
                               y--;
                               p=p+2*(x-y)+1;
                            }
                            x++;
                            plotpoints xcentre, ycentre, x, y);
                            delay(100);
                 }
}
```

/* The main function that takes (x, y) and 'r' the radius from keyboard and activates other functions for drawing the circle. */

```
void main()
{
 int gd = DETECT, gm, xcentre=200, ycentre=150, radius=50;
 printf("\n enter the centre points and radius :\n");
 scanf("%d%d%d", &xcentre, &ycentre, &radius);
 clrscr();
 initgraph(&gd, &gm,"");
 putpixel(xcentre,ycentre,5);
 cir(xcentre, ycentre, radius);
 getch();
 closegraph();
  }
```

/* BRESENHAM's MIDPOINT ELLIPSE ALGORITHM */

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<graphics.h>
int xcenter, ycenter, rx, ry;
int p,px,py,x,y,rx2,ry2,tworx2,twory2;
void drawelipse();
void main()
{
   int gd=3,gm=1;
```

```
clrscr();
initgraph(&gd, &gm,"");
printf("\n Enter X center value : ");
scanf("%d", &xcenter);
printf("\n Enter Y center value : ");
scanf("%d", &ycenter);
printf("\n Enter X radius value : ");
scanf("%d", &rx);
printf("\n Enter Y radius value : ");
scanf("%d", &ry);
cleardevice();
ry2 = ry * ry;
rx2 = rx * rx;
twory2 = 2 * ry2;
tworx2 = 2 * rx2;

/* REGION first */
x=0;
y=ry;
drawelipse();

p = (ry2 - rx2 * ry + (0.25 * rx2));
px = 0;
py = tworx2 * y;
while( px < py )
{
    x++;
    px = px + twory2;
    if(p >= 0)
    {
        y = y - 1;
        py = py - tworx2;
    }
    if(p < 0)
        p = p + ry2 + px;
    else
       {
         p = p + ry2 + px - py;
         drawelipse();
       }
}
```

```
 /*REGION second */
   p = (ry2 * (x + 0.5) * (x + 0.5) + rx2 * (y - 1) * (y - 1) - rx2 * ry2);
   while( y > 0 )
   {
      y = y - 1;
      py = py - tworx2;
      if( p <= 0 )
      {
          x++;
          px = px + twory2;
      }
      if( p > 0 )
           p = p + rx2 - py;
      else
         {
          p = p + rx2 - py + px;
          drawelipse();
         }
   }
   getch();
   closegraph();
}

void drawelipse()
{
   putpixel(xcenter + x, ycenter + y, BROWN);
   putpixel(xcenter - x, ycenter + y, BROWN);
   putpixel(xcenter + x, ycenter - y, BROWN);
   putpixel(xcenter - x, ycenter - y, BROWN);
}
```

/* Program to draw a rectangle and perform scale with ref to the origin */

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

void rect(int,int,int,int);

void main()
{
```

```
        int x,y,len,bre,gd=DETECT,gm;
        float scx,scy;
        clrscr();
        printf("\n\n\tEnter the X of origin ->> ");
        scanf("%d",&x);
        printf("\n\tEnter the Y of origin ->> ");
        scanf("%d",&y);
        printf("\n\n\tEnter the lenth of rectangle ->> ");
        scanf("%d",&len);
        printf("\n\tEnter the breath of rectangle ->> ");
        scanf("%d",&bre);
        initgraph(&gd,&gm,"");/* Graphics is initialized */
        cleardevice();
        rect(x,y,len,bre);
        printf("\n\tEnter the X scaling factor(2-4) ->> ");
        scanf("%f",&scx);
        printf("\n\tEnter the Y scaling factor(2-4) ->> ");
        scanf("%f",&scy);
        rect(x,y,(int)(len*scx),(int)(bre*scy));/* Draws the rectangle */
        getch();
}

void rect(int x,int y,int len,int bre)
{
        line(x,y,x+len,y);
        line(x,y,x,y-bre);
        line(x+len,y,x+len,y-bre);
        line(x,y-bre,x+len,y-bre);
}

/* program to draw a rectangle and perform scale with ref to     arbitrary point
*/

#include <graphics.h>
#include <stdio.h>
#include <conio.h>
void rect(int,int,int,int);
void main()
{
        int x,y,len,bre,nx,ny,sx,sy,gd=DETECT,gm;
        float scx,scy;
```

```
        clrscr();
        printf("\n\n\tEnter the X of origin ->> ");
        scanf("%d",&x);
        printf("\n\tEnter the Y of origin ->> ");
        scanf("%d",&y);
        printf("\n\n\tEnter the lenth of rectangle ->> ");
        scanf("%d",&len);
        printf("\n\tEnter the breath of rectangle ->> ");
        scanf("%d",&bre);
        initgraph(&gd,&gm,"");
        cleardevice();
        rect(x,y,len,bre);
        printf("\n\n\tEnter the X of arbitary pt ->> ");
        scanf("%d",&nx);
        printf("\n\tEnter the Y of arbitrary pt ->> ");
        scanf("%d",&ny);
        printf("\n\tEnter the X scaling factor(2-4) ->> ");
        scanf("%f",&scx);
        printf("\n\tEnter the Y scaling factor(2-4) ->> ");
        scanf("%f",&scy);
        cleardevice();
        setcolor(8);
        rect(x,y,len,bre);
        setcolor(15);
        sx = abs((int)(x*scx+nx*(1-scx)));
        sy = abs((int)(y*scy+ny*(1-scy)));
        rect(sx,sy,(int)(len*scx),(int)(bre*scy));
        getch();
        closegraph();
}

/* Function to draw rectangle using line() function */
void rect(int x,int y,int len,int bre)
{
        line(x,y,x+len,y);
        line(x,y,x,y-bre);
        line(x+len,y,x+len,y-bre);
        line(x,y-bre,x+len,y-bre);
}
```

/* program to draw a triangle and show its scaling and rotation. This program makes use of functions to get the new points after scaling and rotation.

Reference variables are used to get the newly calculated co-ordinate values from the functions. The triangle co-ordinates are static in nature */

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

/* FUNCTIONS DECLARATIONS */
void scale(int *,int *,float,float,int,int);
void rotate(int *,int *,float,int,int);
void translate(int *,int *,int ,int);

void main()
{
  int gd=DETECT,gm,ch;
  int x1,y1,x2,y2,x3,y3,tx,ty;
  float refx,refy,theta,sx,sy;
  initgraph(&gd,&gm,"");
  do
   {
     x1=150,y1=300,x2=250,y2=300,x3=200,y3=200,refx=200,refy=250;  /* co-ordinate values for triangle to be transformed */
     cleardevice();
     /* Draws the triangle */
     line(x1,y1,x2,y2);
     line(x2,y2,x3,y3);
     line(x1,y1,x3,y3);

     /* Menu creation */
     setcolor(YELLOW);

     outtextxy(140,30,"Main Menu");
     outtextxy(140,50," 1. Scale");
     outtextxy(140,70," 2. Rotate");
     outtextxy(140,90," 3. Translate");
     outtextxy(140,110," 4. Exit");
     outtextxy(100,150," Enter Your choice : ");
     scanf("%d",&ch); /* user choice */
     switch(ch)
     {
```

```
case 1 :
        outtextxy(10,10," Enter sx and sy : ");
        scanf("%f%f",&sx,&sy);    /* scale factors */
        cleardevice();
        /* triangle before scaling */
        line(x1,y1,x2,y2);
        line(x2,y2,x3,y3);
        line(x1,y1,x3,y3);

        /* Scaling to get new points */
        scale(&x1,&y1,sx,sy,refx,refy);
        scale(&x2,&y2,sx,sy,refx,refy);
        scale(&x3,&y3,sx,sy,refx,refy);
        /* Triangle after scaling */

        line(x1,y1,x2,y2);
        line(x2,y2,x3,y3);
        line(x1,y1,x3,y3);

        getch();
        break;

case 2 :
        outtextxy(100,180," Enter Angle of Rotation : ");
        scanf("%f",&theta); /* Angle of rotation */
        cleardevice();

        /* Triangle before rotation */
        line(x1,y1,x2,y2);
        line(x2,y2,x3,y3);
        line(x1,y1,x3,y3);

        /* Getting the new co-ordinates after rotation */
        rotate(&x1,&y1,theta,refx,refy);
        rotate(&x2,&y2,theta,refx,refy);

        rotate(&x3,&y3,theta,refx,refy);

        /* Triangle after rotation */
        line(x1,y1,x2,y2);
        line(x2,y2,x3,y3);
        line(x1,y1,x3,y3);
```

```
                getch();
                break;

        case 3 :
                outtextxy(100,180,"Enter tx and ty : ");
                scanf("%d%d",&tx,&ty);
                cleardevice();

                /* Triangle before translation */
                line(x1,y1,x2,y2);
                line(x2,y2,x3,y3);
                line(x1,y1,x3,y3);

                /* To get the Translated co-ordinates */
                translate(&x1,&y1,tx,ty);
                translate(&x2,&y2,tx,ty);
                translate(&x3,&y3,tx,ty);

                /* Triangle after translation */
                line(x1,y1,x2,y2);
                line(x2,y2,x3,y3);
                line(x1,y1,x3,y3);
                getch();
                break;

        case 4 :
                break;
     }

   }while(ch!=4);
   getch();
   closegraph();
}

/* Function to find the new co-ordinate values for scaling */
void scale( int *xnew,int *ynew,float sx,float sy,int refx,int refy)
{
   translate(xnew,ynew,-refx,-refy);
   *xnew=*xnew*sx;
   *ynew=*ynew*sy;
   translate(xnew,ynew,refx,refy);
```

```
}

/* Function to find the new co-ordinate values for rotaion */
void rotate(int *xnew,int *ynew,float angl,int refx,int refy)
{
   int *xold,*yold;
   translate(xnew,ynew,-refx,-refy);
   *xold=*xnew;
   *yold=*ynew;
   angl=angl*3.14159/180; /* Angle converted to radians */
   *xnew=*xold*cos(angl)-*yold*sin(angl);
   *ynew=*yold*cos(angl)+*xold*sin(angl);
   translate(xnew,ynew,refx,refy);
}

/* Function to find the new co-ordinate values for translation */
void translate(int *xnew,int *ynew,int tx,int ty)
{
   *xnew=*xnew+tx;
   *ynew=*ynew+ty;
}



/* Program to implement the cohen-sutherland line clipping algorithm */

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<graphics.h>

void clip(float,float,float,float);
int outcode(float,float);

float xr,xl,yb,yt,xx[50],yy[50],cc=0;

void main()
{
  int gd=DETECT,gm,n,i,i1,i2,len=0;
  float x[50],y[50];
  initgraph(&gd,&gm,"");
  cleardevice();
```

```
setbkcolor(BLACK);
setcolor(CYAN);
printf("\n\t\t\t LINE CLIPPING ");
printf("\n\t\t\t*****************");
printf("\n\t\tHow Many Lines Are Entering For Clipping : ");
scanf("%d",&n);
cleardevice();
for(i=0;i<2*n;i++)
{
   printf("\n\t\tEnter Starting point of x[%d]y[%d] :",i,i);
   scanf("%f %f",&x[i],&y[i]);
   i++;
   printf("\n\t\tEnter Ending point of x[%d]y[%d]   :",i,i);
   scanf("%f%f",&x[i],&y[i]);
}

cleardevice();

/* Window Creation */
setcolor(MAGENTA);
outtextxy(100,100,"BEFORE CLIPPING");
xl=200,xr=400,yt=150,yb=300;
rectangle(xl,yt,xr,yb);
outtextxy(xl,yt-15,"200,150");
outtextxy(xr-50,yt-15,"400,150");
outtextxy(xl,yb+15,"200,300");
outtextxy(xr-50,yb+15,"400,300");

/* draws the given lines */
for(i=0;i<2*n;)
{
   i1=i;
   i2=i1+1;
   line((int)x[i1],(int)y[i1],(int)x[i2],(int)y[i2]);
   i=i+2;
}

/* function call for clipping each of the lines */
for(i=0;i<2*n;)
{
    i1=i;
    i2=i1+1;
```

```
      clip(x[i1],y[i1],x[i2],y[i2]);
      i=i+2;
  }
  getch();
  cleardevice();

  setcolor(LIGHTBLUE);
  outtextxy(100,100,"AFTER CLIPPING");
  rectangle(xl,yt,xr,yb);
  outtextxy(xl,yt-15,"200,150");
  outtextxy(xr-50,yt-15,"400,150");
  outtextxy(xl,yb+15,"200,300");
  outtextxy(xr-50,yb+15,"400,300");
  setcolor(BROWN);

  for(i=0;i<cc;)
  {
     i1=i;
     i2=i1+1;
     line((int)xx[i1],(int)yy[i1],(int)xx[i2],(int)yy[i2]);
     i=i+2;
  }
  getch();
  closegraph();
}

/* Function for clipping the line */
void clip(float x1,float y1,float x2,float y2)
{
   int code1,code2,chk;
   float x,y,m;
   code1 = outcode(x1,y1);
   code2 = outcode(x2,y2);
   while( code1 != 0 | code2 != 0) /* both coordinates are inside the window */
   {
       m = (float)(y2-y1)/(x2-x1);
       chk = code1 & code2;
       if(chk != 0)
           return;
       chk=code1;
       if(code1 == 0)
           chk = code2;
```

```
        if((chk & 1) != 0)  /*take binary no of 1 when insect with left */
        {
            y = y1 + m * (xl-x1);
            x = xl;
        }
        if((chk & 2) != 0)
        {
            y = y1 + m * (xr-x1);
            x = xr;
        }
        if((chk & 4) != 0)
        {
            x = x1 + (yb-y1) / m;
            y = yb;
        }
        if((chk & 8) != 0)
        {
            x = x1 + (yt-y1) / m;
            y = yt;
        }
        if(chk == code1)
        {
            x1=x;y1=y;
            code1 = outcode(x,y);
        }
        else if(chk == code2)
              {
                x2=x;y2=y;
                code2 = outcode(x,y);
              }
    }
    xx[cc]=x1;
    yy[cc]=y1;cc=cc+1;
    xx[cc]=x2;
    yy[cc]=y2;cc=cc+1;
    return;
}

/* Function for deleting line when it is outside the selected window. */
int outcode(float x, float y)
{
    int temp=0;
```

```
  if(x < xl)
     temp = temp | 1;
  else if (x > xr)
          temp = temp | 2;
       else if(y > yb)
               temp = temp | 4;
            else if(y < yt)
                    temp = temp | 8;
  return temp;
}

/* Program for Cohen Sutherland Hodgeman Polygon Clipping Algorithm */

#include<stdio.h>
#include<conio.h>
#include<graphics.h>

void clip(float, float, float, float);
int outcode(float, float);

float xl, xr, yt, yb;
int poly1[25],cc=0;
void main()
{
  int gdriver = DETECT, gmode;
  int poly[10],i,n,i1,i2,i3,i4;
  initgraph(&gdriver, &gmode, "");
  cleardevice();
  printf("\nEnter number of points : ");
  scanf("%d",&n);
  for(i=0;i<2*n-1;)
  {
    printf("Enter The %d Point : ",i);
    scanf("%d",&poly[i]);
    ++i;
    scanf("%d",&poly[i]);
    ++i;
  }
  /* Specifies the viewport */
  xl=200,xr=400,yt=150,yb=300;
  rectangle(xl,yt,xr,yb);
  outtextxy(xl,yt-15,"200,150");
```

```
   outtextxy(xl-5,yb+15,"200,300");

   outtextxy(xr-50,yt-15,"400,150");
   outtextxy(xr-50,yb+15,"400,300");
   drawpoly(n,poly);
   /* Clips each line of the polygon */
   for(i=0;i<2*n-1;)
   {
      i1=i;i2=i1+1;i3=i2+1;i4=i3+1;
      clip(poly[i1],poly[i2],poly[i3],poly[i4]);
      i=i+4;
   }
   getch();
   cleardevice();
   outtextxy(100,100,"AFTER CLIPPING");
   rectangle(xl,yt,xr,yb);
   outtextxy(xl,yt-15,"200,150");
   outtextxy(xl-5,yb+15,"200,300");
   outtextxy(xr-50,yt-15,"400,150");
   outtextxy(xr-50,yb+15,"400,300");
   setcolor(BROWN);
   drawpoly(n,poly1);

   getch();
   closegraph();
}

/*  CLIPPING  */
void clip(float x1,float y1,float x2,float y2)
{
    int code1,code2,chk;
    float x,y,m;
    code1 = outcode(x1,y1);
    code2 = outcode(x2,y2);
    while( code1 !=0 || code2 !=0) /*do both lies inside window*/
    {
       m = (float)(y2-y1)/(x2-x1);
       chk = code1 & code2;
       if(chk != 0)
           return;
       chk=code1;
       if(code1 == 0)
```

```
        chk = code2;
      if((chk & 1) != 0)
      {
          y = y1 + m*(xl -x1);
          x = xl;
      }
      if((chk & 2) != 0)
      {
          y = y1 + m*(xr - x1);
          x = xr;
      }
      if((chk & 4) != 0)
      {
          x = x1 + (yb - y1)/m;
          y = yb;
      }
      if((chk & 8) != 0)
      {
          x = x1 + (yt - y1)/m;
          y = yt;
      }
           if( chk == code1)
           {
             x1=x;y1=y;
             code1 = outcode(x,y);
           }
           else if( chk == code2)
                {
                  x2=x;y2=y;
                  code2 = outcode(x,y);
                }
   }
   poly1[cc]=x1;cc=cc+1;
   poly1[cc]=y1;cc=cc+1;
   poly1[cc]=x2;cc=cc+1;
   poly1[cc]=y2;cc=cc+1;
   return;
}

/* function for cutting outside window portion of the line*/
int outcode(float x,float y)
{
```

```
   int temp=0;
   if(x < xl)
     temp = temp | 1;
   else if(x > xr)
           temp = temp | 2;
         else if(y > yb)
                temp = temp | 4;
              else if(y < yt)
                     temp = temp | 8;
   return temp;
}
```

References

1) Principles of Interactive Computer Graphics – By Newman & Sproull

2) C Graphics & Projects – By B M Havaldar

3) Computer Graphics – By Hearn & Baker

4) Computer Graphics for Scientists and Engineers – By Asthana and Sinha