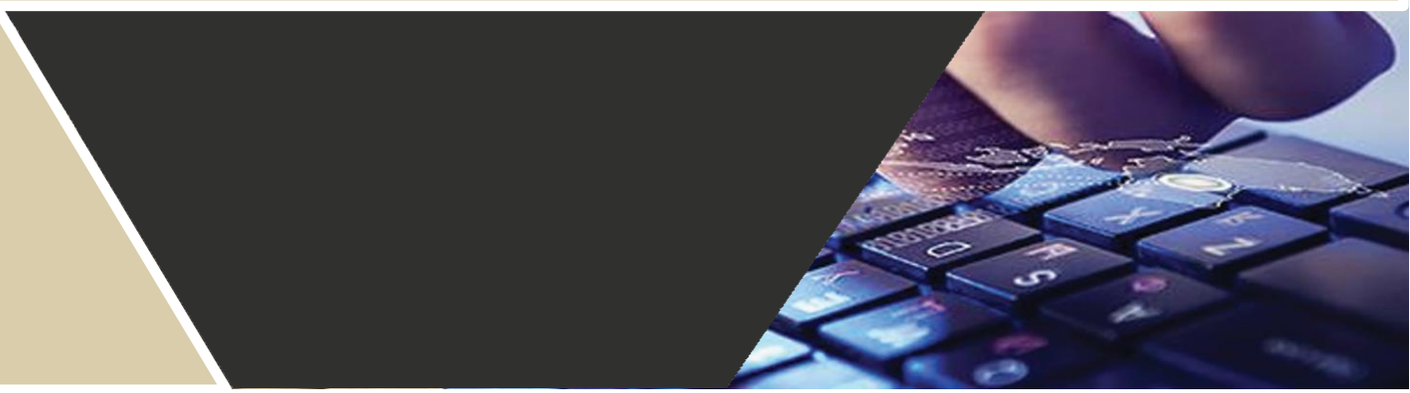
ALAGAPPA UNIVERSITY

[Accredited with 'A+' Grade by NAAC (CGPA:3.64) in the Third Cycle and Graded as Category–I University by MHRD-UGC]

KARAIKUDI – 630 003

DIRECTORATE OF DISTANCE EDUCATION

B.C.A.
10152 / 12752



COMPUTER GRAPHICS

V - Semester

ALAGAPPA UNIVERSITY

[Accredited with 'A+' Grade by NAAC (CGPA:3.64) in the Third Cycle and Graded as Category–I University by MHRD-UGC]

(A State University Established by the Government of Tamil Nadu)

KARAIKUDI – 630 003

Directorate of Distance Education

B.C.A.

V - Semester

10152 / 12752

COMPUTER GRAPHICS

Authors

Rajendra Kumar, *Asst. Professor, Department of CSE, Vidya College of Engineering, Meerut*

Neeraj Pratap, *Senior Lecturer, MCA Department, Vidya College of Engineering, Meerut*

Units (1-4, 5.0-5.3.6, 5.4-5.8, 6-8, 9.0-9.2, 9.4-9.11, 10, 12.0-12.5, 12.8-12.13)

Anirban Mukhopadhyay, *Lecturer RCC Institute of Information Technology, Kolkata*

Arup Chattopadhyay, *Systems Manager and Head Scientific and Technical Application Group DOEACC Centre Jadavpur University Campus Kolkata*

Units (5.3.7, 9.3, 11.2, 12.6-12.7)

Rajendra Kumar, *Asst. Professor, Department of CSE, Vidya College of Engineering, Meerut*

Units (11.0-11.1, 11.3-11.8)

Vikas® Publishing House: Units (13)

Vikas® is the registered trademark of Vikas® Publishing House Pvt. Ltd.

VIKAS® PUBLISHING HOUSE PVT. LTD.
E-28, Sector-8, Noida - 201301 (UP)
Phone: 0120-4078900 • Fax: 0120-4078999
Regd. Office: 7361, Ravindra Mansion, Ram Nagar, New Delhi 110 055
• Website: www.vikaspublishing.com • Email: helpline@vikaspublishing.com

Work Order No. AU/DDE/DE1-291/Preparation and Printing of Course Materials/2018 Dated 19.11.2018 Copies - 500

SYLLABI-BOOK MAPPING TABLE

Computer Graphics

Syllabi	Mapping in Book
BLOCK - I: INTRODUCTION **UNIT 1: Introduction:** Application Areas of Computer Graphics, Overview of Graphics Systems, Video-Display Devices, Raster-Scan Systems, Random Scan Systems, Graphics Monitors and Work Stations and Input Devices. **UNIT 2: Output Primitives:** Points and Lines, Line Drawing Algorithms, Mid-Point Circle and Ellipse Algorithms. **UNIT 3: Filled Area Primitives:** Scan Line Polygon Fill Algorithm, Boundary-Fill and Flood-Fill Algorithms.	**Unit 1:** Introduction to Computer Graphics **(Pages 1-15);** **Unit 2:** Output Primitives **(Pages 16-54);** **Unit 3:** Filled Area Primitives **(Pages 55-65)**
BLOCK - II: 2 D TRANSFORM AND CLIPPING **UNIT 4: 2-D Geometrical Transform:** Translation, Scaling, Rotation, Reflection and Shear Transformations **UNIT 5: 2D Matrix Representations:** Homogeneous Coordinates, Composite Transforms, Transformations between Coordinate Systems. **UNIT 6: 2-D Viewing:** The Viewing Pipeline, Viewing Coordinate Reference Frame, Window to View-Port Coordinate Transformation, Viewing Functions, **UNIT 7: Clipping Algorithms:** Cohen-Sutherland and Cyrus-Beck Line Clipping Algorithms, Sutherland-Hodgeman Polygon Clipping Algorithm.	**Unit 4:** 2D Geometrical Transform **(Pages 66-88);** **Unit 5:** 2D Matrix Representations **(Pages 89-109);** **Unit 6:** 2-D Viewing **(Pages 110-120);** **Unit 7:** Clipping Algorithms **(Pages 121-149)**
BLOCK - III: 3D OBJECT REPRESENTATION **UNIT 8: Introduction:** Polygon Surfaces, Quadric Surfaces, Spline Representation. **UNIT 9: Curve and Surfaces:** Hermite Curve, Bezier Curve and B-Spline Curves, Bezier and B-Spline Surfaces. Basic Illumination Models, Polygon Rendering Methods.	**Unit 8:** Introduction to Surfaces **(Pages 150-161);** **Unit 9:** Curve and Surfaces **(Pages 162-199)**
BLOCK - IV: 3D GEMETRIC TRANSFORMATION **UNIT 10: 3-D Geometric Transformations:** Translation, Rotation, Scaling, Reflection and Shear Transformations, Composite Transformations. **UNIT 11: 3-D Viewing:** Viewing Pipeline, Viewing Coordinates, View Volume and General Projection Transforms and Clipping.	**Unit 10:** 3D Geometric Transformations **(Pages 200-216);** **Unit 11:** 3D Viewing **(Pages 217-228)**
BLOCK - V: VISIBLE SURFACE DETECTION METHODS AND ANIMATION **UNIT 12: Classification:** Back-Face Detection, Depth-Buffer, Scan-Line, Depth Sorting, BSP-Tree Methods, Area Sub-Division and Octree Methods. **UNIT 13: Computer Animation:** Design of Animation Sequence, General Computer Animation Functions, Raster Animation, **UNIT 14: Other Animation Techniques:** Computer Animation Languages, Key Frame Systems, Motion Specifications.	**Unit 12:** Classification **(Pages 229-242);** **Unit 13:** Computer Animation **(Pages 243-250);** **Unit 14:** Other Animation Techniques **(Pages 251-263)**

CONTENTS

INTRODUCTION

NOTES

Computer graphics refers to the use of computational and mathematical foundations that generate and process images using computers to store, create, and manipulate drawings and pictures. Earlier, applications in engineering and science had to rely on expensive and cumbersome equipment. The development and advancement in computers has made interactive computer graphics an effective tool. It can be used to present information in such wide-ranging fields as medicine, entertainment, training, business, engineering, education, government, science, advertising, industry, and art. Computer graphics generates special effects using imaging, geometry, animation and rendering.

Designing is a major application of computer graphics. It is particularly used in engineering and architectural systems. Computer Aided Design (CAD) techniques are now used on a regular basis to design buildings, aircrafts, computer-components, textiles, automobiles and a variety of consumer products. The CAD environment replaces the traditional tools of design with parameterized modelling routines that have interactive graphic capabilities. These capabilities are so versatile and dynamic that a designer can carry out unlimited number of experiments to obtain better designs.

Among the other applications of computer graphics, Image Processing, Animation, Morphing, Simulation, e-Learning Material Designing and Graphic Designing are rapidly gaining demand and usage in education, training, advertisement and entertainment. Computer graphics has highly influenced the film industry with its multimedia applications. Controlled animation, simulation and morphing have increasingly been applied in the study of time-varying physical phenomena, object movement and operating sequences of machinery in scientific and industrial research. Computer-aided image processing and picture analysis are now indispensable tools for remote sensing, aerial survey, space research, pattern recognition, CT scans and research in medical sciences.

This book, *Computer Graphics*, follows the SIM format or the self-instructional mode wherein each Unit begins with an Introduction to the topic followed by an outline of the Objectives. The detailed content is then presented in a simple and an organized manner, interspersed with Check Your Progress questions to test the understanding of the students. A Summary along with a list of Key Words and a set of Self Assessment Questions and Exercises is also provided at the end of each unit for effective recapitulation.

BLOCK - I
INTRODUCTION

UNIT 1 INTRODUCTION TO COMPUTER GRAPHICS

Structure

1.0 INTRODUCTION

Today, computers have become important tools that produce pictures economically. Graphical displays can be used in all areas. Therefore, its extensive usage is not surprising. Although early applications in science and engineering had to rely on cumbersome and expensive equipment, current advances in computer technology have made interactive computer graphics a practical tool. Today, computer graphics are being used in diverse areas like engineering, science, business, medicine, government, industry, entertainment, training, education and art. In computers, the term graphics implies almost everything that is neither text nor sound. These days, almost all computers use some level of graphics, and computer users expect to control their computers through shortcuts/icons and images. They no longer depend on typing alone. One thinks of computer graphics as drawing pictures on computer systems. These can be drawings, photographs, simulations (like java applets), or movies. On the other hand, they may be images from places one cannot see directly, such as medical images (DNA structure, etc.) from inside our body. One would like to use images on computers to not just look more practical, but also to be more practical in terms of their colour and brightness combinations, the way different materials appear, and the way objects are lighted.

NOTES

1.1 OBJECTIVES

After going through this unit, you will be able to:

- Discuss the various applications of computer graphics
- Explain the various types of video display devices, raster-scan and random scan systems
- Describe the various types of input devices

1.2 APPLICATION AREAS OF COMPUTER GRAPHICS

This section will discuss the applications of computer graphics for which an understanding of Computer-aided Design (CAD) is essential.

Computer graphics is mainly used in the process of design, generally for engineering and architectural systems. But almost all products are now computer designed. Generally Computer-aided design (CAD) methods are now used to design automobiles, buildings, watercraft, aircraft, embedded systems, spacecraft, textiles and many other products.

Generally, objects are first displayed for design applications in a wireframe outline form which describes the internal features and the overall shape of the object. Wireframe displays also allow designers to rapidly see the effects of interactive adjustments to design the shape. The software packages for CAD applications generally provide a multi-window environment.

Animations are often used in CAD applications. Real time animations using wireframe displays on a video monitor are used to test the performance of a vehicle or system. When one does not display objects with rendered surfaces, the calculations for each segment of the animation can be performed quickly to produce a smooth real-time motion on the screen. Also, wireframe displays allow the designer to look into the interior of the vehicle and to watch the behaviour of inner components during the motion. Animations in *virtual reality* environments are used to determine how vehicle operations are affected by certain motions.

When object designs are complete, or near completion, realistic lighting models and surface rendering is applied to produce the displays which will show the appearance of the final product. Realistic displays can also be generated to advertise automobiles and other vehicles using special lighting effects and background scenes.

Building Architectures

Architects also use interactive graphics methods to layout floor plans, which show the positioning of the rooms, doors, windows, stairs, shelves, counters, and other

NOTES

building features. Working from the display of a building layout on a video monitor, an electrical designer can try out arrangements for wiring, electrical outlets, and fire warning systems. Also, facility-layout packages can be applied to the layout to determine space utilization in an office or on a manufacturing floor.

Realistic displays of architectural designs, allow both architects and their clients to study the appearance of a single building or a group of buildings, like a campus or an industrial complex. In addition to the realistic building displays, the graphics packages also provide facilities for experimenting with three-dimensional interior layouts and lighting.

Computer Art

Computer graphics methods are widely used in fine art as well as commercial art applications. Artists use a variety of computer methods including special purpose hardware, paintbrush programs, specific purpose software, desktop publishing software, CAD packages, animation packages, etc., to provide facilities for specifying object motions and designing object shapes.

The paintbrush program allows graphics users to paint pictures available on the video monitor. Actually the picture is usually painted electronically on a graphics tablet using a stylus, which can simulate different brush strokes, brush widths and colours.

Fine artists use a variety of other computer technologies to produce images. To create a picture, artists use a combination of three-dimensional modeling packages, texture mapping, drawing programs and CAD software. In the mathematical art, artists use a combination of mathematical functions, fractal procedures, mathematical software, ink-jet printers, and others systems to create a variety of three-dimensional and two-dimensional shapes and stereoscopic image pairs.

A common graphics method employed in many commercials is *morphing*, where one object is transformed into another. This method has been used in TV commercials to turn an oil container into an automobile engine, an automobile into a tiger, a puddle of water into a tire, and one person face into another face.

Presentation Graphics

A major application area of computer graphics is presentation graphics, which is used to produce illustrations for reports or to generate 35-mm slides or transparencies to use with projectors. Presentation graphics is commonly used to summarize statistical, financial, mathematical, economic and scientific data for managerial reports, research reports, and other types of reports. Workstation devices and service bureaus exist for converting screen displays into 35-mm slides or overhead transparencies for use in presentations. The main examples of presentation graphics are bar charts, pie charts, line graphs, surface graphs and other displays.

NOTES

Entertainment

These days, computer graphics methods are widely used to make music videos, motion pictures, television shows, etc. Many TV shows and sports telecast regularly employ computer graphics methods. Music videos use graphics in several ways. Graphics objects can be combined with live action or graphics and image processing techniques can be used to produce a transformation of one person or object into another (i.e., morphing).

Image Processing

Although methods used in computer graphics and image processing overlap, the two areas are concerned with fundamentally different operations. Image processing applies techniques to modify or interpret existing pictures, such as satellite photographs and TV scans. Two principal applications of image processing are improving picture quality and machine perception of visual information, as used in robotics.

To apply image-processing methods, one first digitizes a photograph or any other picture into an image file. Then digital methods can be applied to rearrange picture parts, to increase colour separations, or to improve the quality of shading. These techniques are widely used in commercial art applications that involve retouching and rearranging of sections and of photographs and other art work. Similar methods are used to analyse satellite photos of the earth and photos of galaxies.

Medical science also makes wide use of image-processing techniques to enhance images through preprocessing steps, in tomography and in various simulations. Tomography is a technique of X-ray photography that allows cross-sectional views of physiological systems to be displayed. Both Computed X-ray Tomography (CT) and Position Emission Tomography (PET) use the projection method to reconstruct cross sections from digital data. These techniques are also used to monitor internal functions and show cross sections during surgery. Other medical imaging techniques include ultrasonic and nuclear medicine scanners. Ultrasonic uses high frequency sound waves, instead of X-rays.

Education and Training

Computer-generated methods of financial, physical, and economic systems are often used as educational aids for various purposes. Physiological systems, modeling of physical systems, population trends, equipment, etc., can be used to help trainees understand the operation of a system in an attractive manner. Special systems are designed for some training applications. Examples of such specialized systems are the simulators for practice session or training of aircraft pilots, ship captains, heavy-equipment operators, air traffic control personnel, etc. Some simulators have no video screen, for example, a flight simulator with only a control panel for instrument flying. But most simulators provide graphics screens for visual operation. A

keyboard is used to input parameters affecting the performance of an airplane or an environment, and the pen plotter is used to chart the path of an aircraft during a training session.

NOTES

Visualization

Scientists, engineers, medical personnel, system analysts, and others often need to analyse large amounts of information to study the behaviour of certain processes. Numerical simulations can be carried out on supercomputers for frequently producing data files containing thousands and often millions of data values. Similarly, satellite cameras, radars and other sources are gathering large data files faster than they can be compiled. The process of scanning these large sets of data to determine trends and relationships is a deadly and unproductive process. But if graphics is applied to data, the converted form (visual form) is much more interesting. Producing graphical representations for engineering, scientific, and medical data sets and related processes is generally referred as scientific visualization. The term business visualization is used in connection with data sets related to industry, commerce and other nonscientific areas.

There are many different kinds of data sets, and effective visualization schemes that depend on the characteristics of the data. A collection data can contain scalar values, vectors, higher-order tensors, or any combination of these data types. Additional techniques include contour plots, graphs and charts, surface renderings, and visualizations of volume interiors. In addition, image processing techniques are combined with computer graphics to produce many data visualizations. Mathematicians, scientists, and others use visual techniques to analyse mathematical functions and processes or simply to produce interesting graphical representations.

1.3 OVERVIEW OF GRAPHICS SYSTEMS

Graphics capabilities for both two-dimensional and three-dimensional applications are now common in general-purpose computer applications, including many hand-held tools like calculators. Graphics users can use a wide variety of interactive input devices and graphics software packages with personal computers. They can choose from a number of complicated special-purpose graphics hardware systems (like track ball and plotters) and technologies for higher quality applications.

1.3.1 Video Display Devices

There are various types of video display devices some of them have been discussed in this section.

Refresh cathode-ray tubes (CRTs)

A cathode-ray tube is frequently called a CRT. A CRT is an electronic display device in which an electron beam can be focused on a phosphorescent viewing screen and speedily varied in position and intensity to produce an image. The

NOTES

best-known application of a CRT is as a picture tube in all commercially available televisions. Other applications include radar screens, oscilloscopes, flight simulators, embedded systems, and computer monitors.

A cathode-ray tube consists of three basic parts: the glass wrapper, the phosphor viewing surface, and the electron gun assembly. The electron gun assembly consists of a heated metal cathode enclosed by a metal anode. The anode is given a positive electrical voltage and the cathode is given a negative electrical voltage. There is a flow of electrons from the cathode through a small nozzle in the anode that produces a beam of electrons. The electron gun contains electrical plates or coils which focus, accelerate, and redirect the electron beam to strike the phosphor viewing surface in a quick side-to-side scanning motion starting at the top of the surface and working downwards, repeating the same process. The phosphor viewing surface is a thin layer of material through which visible light is emitted when hit by the electron beam. The colour on the display unit is determined by the chemical composition of the phosphor and can be altered by changing its composition. The glass wrapper consists of a relatively flat face plate (horizontal and vertical), a neck section, and a funnel section. The electron gun assembly is packed into the glass neck at the opposite end. The phosphor viewing surface is deposited on the inside of the glass plate. The function of the funnel is to place the electron gun at an appropriate distance from the plate and to grip the glass envelope together so that a vacuum can be achieved inside the tube.

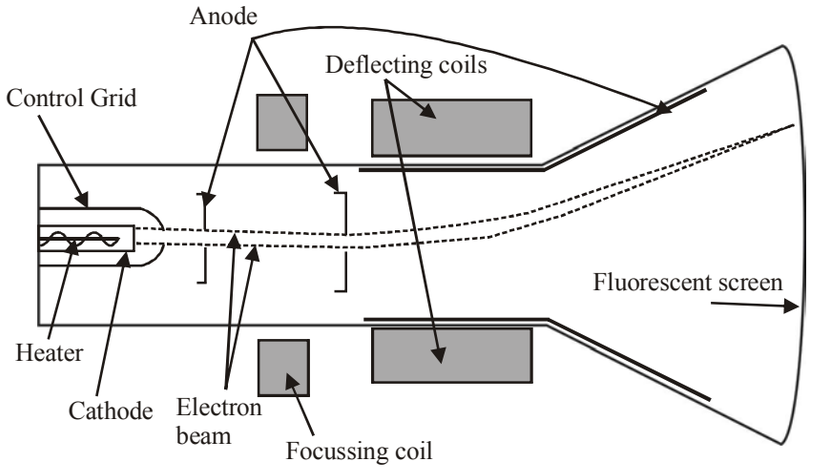


Fig. 1.1 *A Cathode Ray Tube*

A cathode-ray tube, used in a colour computer monitor or colour television, has a few additional parts. There are three electron guns instead of one electron gun in general, one for the red colour signal, one for blue, and one for green. Also three different phosphor materials are used on the display surface, one for each colour. A 63 Centimeters (25-inch) colour TV picture tube may have a shadow mask with 1.5 million individual phosphor dots and 500000 perforations.

NOTES

Design of CRT

As a graphics application is initiated, the electron gun is programmed for each new graphics application. New screen sizes, new image resolution settings, and new overall glass envelope dimensions, all require a new electron gun design. New image resolution necessities may require a new technique of depositing the phosphor dots on the face plate. As a result it may require new material processing techniques. The amount of time the phosphors produce light or shine is controlled by the chemical composition of the phosphor. This phenomenon is known as *persistence*. In a colour television CRT, the electron beam scans the screen 25 times per second (as shown in figure 1.2). If the persistence is longer than one twenty-forth of a second, the image shows two scans at the same time and appears indistinct.

1.3.2 Raster Scan Display

The most common type of graphics monitor, the raster scan, is based on television technology and uses a CRT. The electron beam in a raster scan system is swept across the CRT, one row at a time from top to bottom, as shown in Figure 1.2. The beam intensity is turned on and off as the electron beam moves across each row, to create a pattern of illuminated dots. The picture definition is saved in a memory area (called the *frame buffer*). The frame buffer holds the set of intensity values for all the screen points. The stored intensity values are then fetched from the refresh buffer and displayed on the screen one row at a time. The capability of a raster scan system to save intensity information for each pixel makes it more efficient for the practical display of scenes containing fine shade and colour patterns.

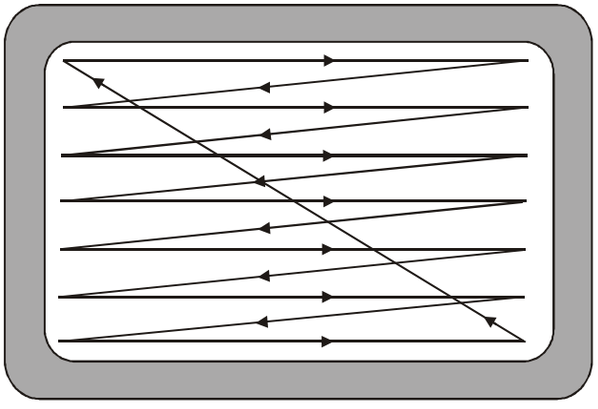


Fig. 1.2 *Raster Scan Display*

In a simple black-and-white system (one kind of monochrome screen), each screen point is either on or off. Thus, only one bit per pixel is needed to control the intensity of screen positions. A bit value of 1 indicates that the electron beam is to be turned on at that position for a bi-level system, and a value of 0 indicates that the beam intensity is off at that pixel position. Up to 24 bits per pixel may be included in high-quality systems, which can require several megabytes of

memory for the frame buffer, depending on the resolution of the system. A system with 24 bits per pixel and a screen resolution of 1024 × 1024 requires 3 MB of storage for the frame buffer.

NOTES

1.3.3 Random Scan Display

Basically there are two types of CRTs: raster scan type and random scan type. The main difference between the two is based on the technique with which the image is generated on the phosphor coated CRT screen. In raster scan, the electron beam sweeps the entire screen in the same way as you would write a full page text in a note book, word by word, character by character, from left to right and top to bottom. On the other hand, in random scan technique, the electron beam is directed at that point of the screen where the image is to be produced. It generates the image by drawing a set of random straight lines much in the same way one might move a pencil over a piece of paper to draw an image, drawing strokes from one point to another, one line at a time. This is why this technique is also referred to as vector drawing or stroke writing or calligraphic display.

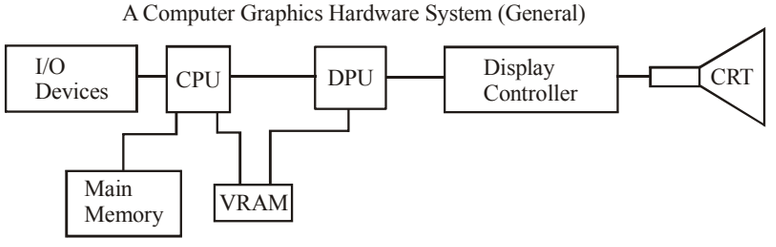


Fig. 1.3 *Block Diagram of Random Scan Display*

There are of course no bit planes containing mapped pixel values in the vector system. Instead the display buffer memory stores a set of line drawing commands along with end point coordinates in a display list or display program created by a graphics package. The display processing unit (DPU) executes each command during every refresh cycle and feeds the vector generator with digital x, y, and x, y values. The vector generator converts the digital signals into equivalent analog deflection voltages. This causes the electron beam to move to the start point or from the start point to the end point of a line or vector. Thus the beam sweep does not follow any fixed pattern; the direction is arbitrary as dictated by the display commands. When the beam focus must be moved from the end of one stroke to the beginning of the other, the beam intensity is set to 0.

Though vector-drawn images lack in depth and colour precision, random displays can work at higher resolution than raster displays. The images of random display are sharp and have smooth edges unlike the jagged edges and lines on raster displays.

NOTES

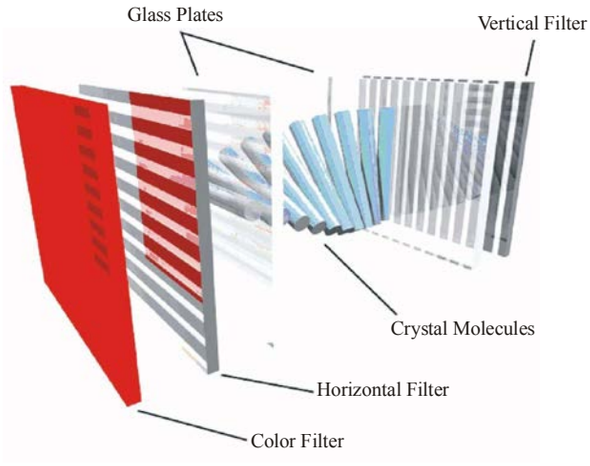


Fig. 1.4 *Functioning of LCD Display*

1.3.4 Liquid Crystal Display (LCD)

LCD is the technology used for displays in notebooks, laptops and other smaller computers. Like gas-plasma technologies and light-emitting diodes (LEDs), LCDs allow displays to be much thinner than CRT technology. The liquid crystal displays use much less power than LED and gas-display displays because they work on the principle of blocking light rather than its release.

A liquid crystal display can be made with either a passive or an active matrix display grid. The active matrix LCD is also called a thin film transistor (TFT) display. The passive matrix liquid crystal display has a grid of conductor materials with pixels located at each intersection in the grid. A current can be applied across two conductors on the grid to control the light for any pixel. Some passive matrix liquid crystal displays have a double scanning mechanism. This means that they scan the grid with current to times in the same time that it takes for one scan in the original technology. However, active matrix technology is still a superior one.

Direct View Storage Tube

Direct View Storage Tube (DVST) is rarely used today as part of a display system. However, DVST marks a significant technological change in the usual refresh type display. Both in the raster scan and random scan system the screen image is maintained flicker free by redrawing or refreshing the screen many times per second. This is done by cycling through the picture data stored in the refresh buffer. In DVST there is no refresh buffer; images are created by drawing vectors or line segments with a relatively slow moving electron beam. The beam is designed not to draw directly on phosphor but on a fine wire mesh (called storage mesh) coated with dielectric and mounted just behind the screen. A pattern of positive charge is deposited on the grid, and this pattern is transferred to the phosphor coated screen by a continuous flood of electrons emanating from a separate flood gun.

NOTES

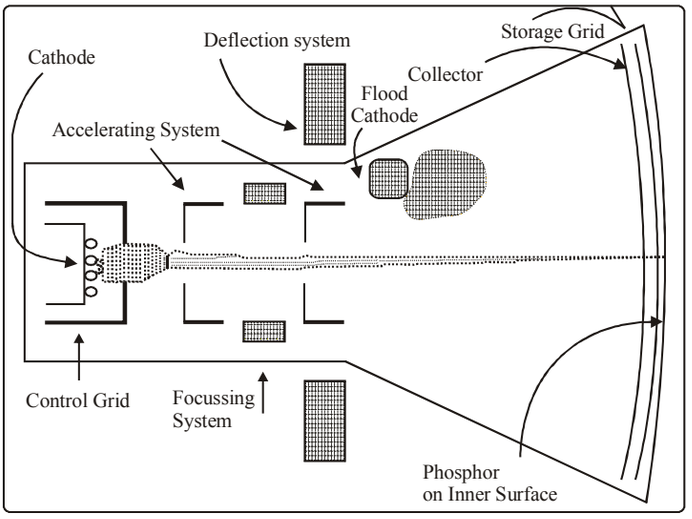


Fig. 1.5 *Direct View Storage Tube (DVST)*

Just behind the storage mesh is a second grid, the collector. Its main purpose is to smooth out the flow of flood electrons. These electrons pass through the collector at a low velocity and are attracted to the positively charged portions of the storage mesh but repelled by the rest. Electrons not repelled by the storage mesh pass right through it and strike the phosphor.

To increase the energy of these slow moving electrons and thus create a bright picture, the screen is maintained at a high positive potential. The storage tube retains the image generated until it is erased. Thus no refreshing is necessary, and the image is absolutely flickering free.

A major disadvantage of DVST in interactive computer graphics is its inability to selectively erase parts of an image from the screen. To erase a line segment from the displayed image, one has to first erase the complete image and then redraw it by omitting that line segment. However, the DVST supports a very high resolution which is good for displaying complex images.

1.4 INPUT DEVICES FOR GRAPHICS

This section will discuss the input devices that are used in a graphics system.

Keyboard

A keyboard is an input device. It is partially modelled after the keyboard of a typewriter and uses a similar arrangement of buttons or keys. These keys act as electronic switches. A computer keyboard typically has characters imprinted on the keys and each press of a key typically corresponds to a single written symbol. However, to produce some special symbols and notations, one is required to press and hold several keys simultaneously or in a particular sequence.

Fig. 1.6 *A Computer Keyboard*

The understanding of key-presses is generally left to the software in a modern computer. A computer keyboard detects distinguishably each physical key from every other and reports all key-presses to the controlling software. Computer keyboards can also be used for computer gaming, either with regular keyboards or by using special gaming keyboards, which can accelerate frequently used key-press combinations. A computer keyboard can also be used to give commands to the operating system of a computer, such as Macintosh Control-Alt-Delete combination.

Types of keyboard

The following are the various types of keyboards:

(*i*) **Standard:** In India, QWERTY keyboard is the standard keyboard, in which the first six alphabet keys are Q, W, E, R, T and Y in that sequence. Standard keyboards, such as the 104-key Windows keyboards include numbers, punctuation symbols, alphabetic characters and a variety of function keys (for special purposes).

(*ii*) **Gaming and multimedia:** Multimedia keyboards are the keyboards with extra keys. They have special keys for music, accessing web, and other often used programs, volume buttons, a mute button and also a sleep (standby) button. Gaming keyboards have extra function keys, which are programmed with keystroke macros as per requirement. For example, 'shift+ctrl+y' can be a keystroke that is frequently used in a certain computer game.

(*iii*) **Thumb-sized:** It is a wireless keyboard. In this keyboard a Windows button and multimedia keys are placed at the top. Smaller keyboards have been introduced for PDAs, laptops, cell phones or users who have limited workspace. The size of a standard keyboard is dictated by the practical consideration that the keys must be large enough to be easily pressed by fingers. As a size manipulation, the numeric keyboard to the right of the alphabetic keyboard can be removed, or the size of the keys can be reduced but it makes harder to enter text. Another way to reduce

NOTES

the size of the keyboard is to reduce the number of keys and use key combinations for certain operations, i.e., pressing several keys simultaneously. For example, the GKOS keyboard has been designed for small wireless devices. Mobile phones allow a single key to type three or four different symbols.

(*iv*) **Numeric:** Numeric keyboards contain only numbers, mathematical symbols for addition, subtraction, multiplication, and division, a decimal point, and several function keys (e.g., End, Delete, etc.). Such types of keyboards are designed specially for data entry.

Mouse

A mouse is an input pointing device. It functions by detecting two-dimensional motion of light or wheel relative to its supporting surface. Physically, a mechanical mouse consists of an object held under one of the user's hands, with one or more buttons. The mouse's motion typically translates into the motion of a pointer on a display. This motion allows for fine control of GUI (Graphical User Interface).

Fig. 1.7 *A Computer Mouse*

The Light Pen

A light pen is a graphics input device. It utilizes a light-sensitive detector to select objects on a display screen. The functioning of a light pen is similar to a mouse, except that a light pen user can move the pointer and select objects on the display screen by directly pointing to the objects with the pen.

Fig. 1.8 *A USB Light Pen*

NOTES

Joystick

A joystick is also a graphics input device consisting of a stick. It plays a major role in game activity controlling. Joysticks usually have one or more push-buttons whose state can also be read by the computer for various operations. Joysticks are often used to control video games. Joysticks have been the principal flight control mechanism in the cockpits of many aircrafts, particularly military jets, where centre stick or side-stick locations are employed. An analog stick is a popular variation of the joystick and is used in modern video game consoles.

Fig. 1.9 *A Joy Stick*

Joysticks often have one or more *fire buttons*. These buttons are used to trigger some kind of action. These are simple on/off switches. Some joysticks have force feedback capability. These are thus active devices, not merely input devices. Most Input/Output interface cards for PCs have a joystick (game control) port.

Check Your Progress

1. Define CRT.
2. What does frame buffer holds?
3. What is a joystick?

1.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. A CRT is an electronic display device in which an electron beam can be focused on a phosphorescent viewing screen and speedily varied in position and intensity to produce an image.
2. The picture definition is saved in a memory area (called theframe buffer). The frame buffer holds the set of intensity values for all the screen points.
3. A joystick is also a graphics input device consisting of a stick. It plays a major role in game activity controlling.

NOTES

1.6 SUMMARY

- Computer graphics is mainly used in the process of design, generally for engineering and architectural systems.
- Computer graphics methods are widely used in fine art as well as commercial art applications.
- Graphics capabilities for both two-dimensional and three-dimensional applications are now common in general-purpose computer applications, including many handheld tools like calculators.
- A CRT is an electronic display device in which an electron beam can be focused on a phosphorescent viewing screen and speedily varied in position and intensity to produce an image.
- LCD is the technology used for displays in notebooks, laptops and other smaller computers. Like gas-plasma technologies and light-emitting diodes (LEDs), LCDs allow displays to be much thinner than CRT technology.
- A keyboard is an input device. It is partially modelled after the keyboard of a typewriter and uses a similar arrangement of buttons or keys. These keys act as electronic switches.
- Presentation graphics is commonly used to summarize statistical, financial, mathematical, economic and scientific data for managerial reports, research reports, and other types of reports.

1.7 KEY WORDS

- **Mouse:** A mouse is an input pointing device. It functions by detecting two-dimensional motion of light or wheel relative to its supporting surface.
- **Light Pen**: A light pen is a graphics input device. It utilizes a light-sensitive detector to select objects on a display screen. The functioning of a light pen is similar to a mouse, except that a light pen user can move the pointer and select objects on the display screen by directly pointing to the objects with the pen.

1.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Discuss the various applications of computer graphics.
2. What are the different types of video display devices?
3. Differentiate between raster scan and random scan systems.

Long Answer Questions

1. Explain the raster scan display systems.
2. What do you understand by the random scan display? Explain.
3. Explain the various types of input devices.

NOTES

1.9 FURTHER READINGS

Hearn, Donal and M. Pauline Baker. 1990. *Computer Graphics*. New Delhi: Prentice-Hall of India.

Rogers, David F. 1985. *Procedural elements for Computer Graphics*. New York: McGraw-Hill.

Foley, D. James, Andries Van Dam, Steven K. Feiner and John F. Hughes. 1997. *Computer Graphics Principles and Practice*. Boston: Addison Wesley.

Mukhopadhyay, A. and A. Chattopadhyay. 2007. *Introduction to Computer Graphics and Multimedia*. New Delhi: Vikas Publishing House.

UNIT 2 OUTPUT PRIMITIVES

NOTES

Structure

2.0 INTRODUCTION

In this unit, you will learn about the line, circle and ellipse drawing algorithms. Basic shapes like lines, circles and curves play an important role in computer graphics. Such shapes are created as a collection of pixels. Various algorithms have been developed to determine the next pixel position to produce a particular shape. Basic arithmetic operations and some complex operations are performed to determine pixel positions.

2.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand point, line and line segment
- Discuss vectors
- Explain line drawing algorithms
- Analyse Bresenham's circle algorithm
- Appreciate ellipse generating algorithm

2.2 POINTS AND LINES

This section will discuss some basic mathematical concepts regarding line, line segments and points.

Point

A position in a plane is known as a point and any point can be represented by any ordered pair of numbers (x, y), where x is the horizontal distance from the origin and y is the vertical distance from the origin.

NOTES

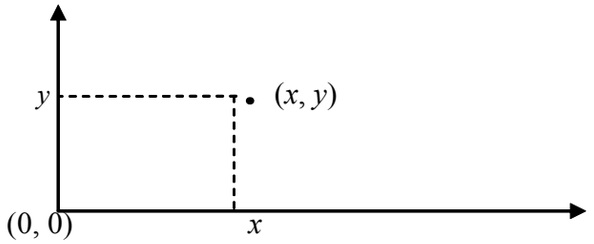


Fig. 2.1 *Representation of a Point at (x, y) Position*

Line

Line can be represented by two points, i.e., both the points will be on the line and lines can also be described by an equation. It can also be defined by any point which satisfies the equation on the line. If two points $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ are specifying a line and another third point $P(x, y)$ also satisfies the equation, the slope between points P_1P and P_1P_2 will be as follows:

$$\frac{y - y_1}{x - x_1} = \frac{y_2 - y_1}{x_2 - x_1} \quad \text{....(2.1)}$$

or $$(x - x_1)(y_2 - y_1) = (x_2 - x_1)(y - y_1)$$

or $$(x_2 - x_1)y = (x - x_1)(y_2 - y_1) + y_1(x_2 - x_1)$$

or $$y = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x - x_1) + y_1 \quad \text{...(2.2)}$$

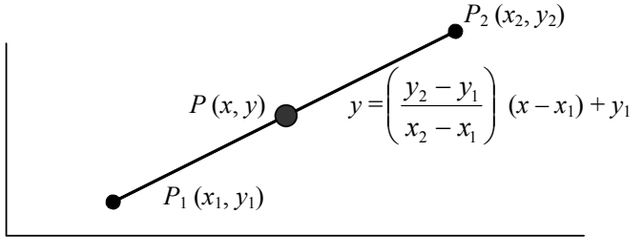


Fig. 2.2 *A Line with Equation*

This is the equation of a line that is passing through the coordinate points (x_1, y_1) and (x_2, y_2). Because terms x_1, y_1, and x_2, y_2 are numerical values therefore the quantity $(y_2 - y_1)/(x_2 - x_1)$ will be a constant. Let us denote it by m.

Putting the value of m in equation (2.2), we have

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

$$y - y_1 = m(x - x_1) \quad \text{...(2.3)}$$

where, $y - y_1 = mx - mx_1$

$y = mx - mx_1 + y_1$

$= mx + (-mx_1 + y_1)$

NOTES

where the value $(-mx_1 + y_1)$ is constant, let us denote it by c

then

$$y = mx + c \quad ...(2.4)$$

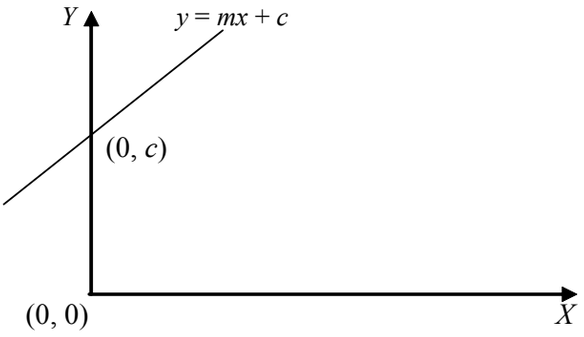


Fig. 2.3 *A Line with Intercept c*

which crosses the y-axis at height c and is known as the slope –intercept form of the line where c is intercept and m is the slope of the line.

If this line passes through the origin (0, 0) then putting (0, 0) in equation 2.4

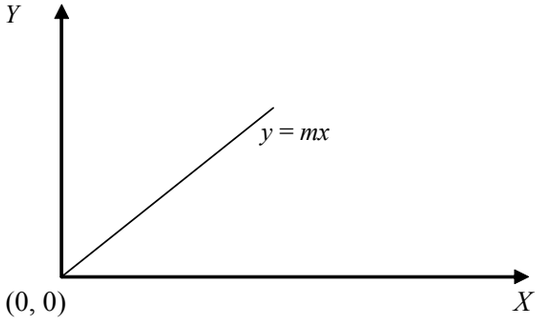


Fig. 2.4 *A Line passing through Origin*

we get, $0 = m.0 + c$

or $c = 0$

or $y = mx$ ....(2.5)

This is the equation of a line passing through the origin as shown in the figure 2.4 The equation 2.2 can be written as follows:

$$(y_2 - y_1)x - (x_2 - x_1)y + x_2y_1 - x_1y_2 = 0$$

Replacing the constant values $(y_2 - y_1)$, $(x_2 - x_1)$ and $(x_2y_1 - x_1y_2)$ by P, $-Q$, R, we get

$$Px + Qy + R = 0 \quad ....(2.6)$$

This is the general form of the line

Comparing this equation with equation 2.5, we have

$$m = -\frac{P}{Q}$$

and

$$c = -\frac{R}{Q}$$

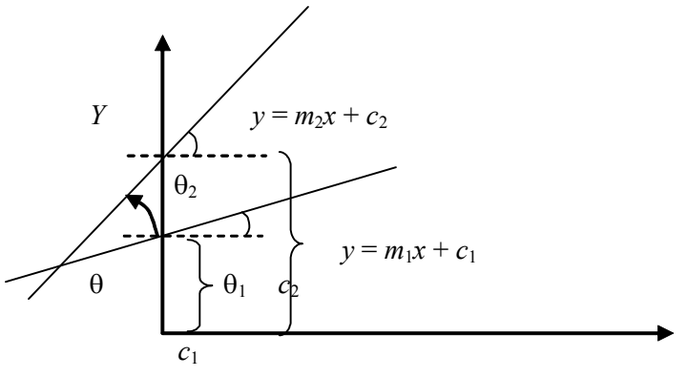


Fig. 2.5 *Two Lines intercepting each Other*

Let us consider two lines $y = m_1x + c_1$, and $y = m_2x + c_2$, as shown in the figure 2.5, which have tangents $m_1 = \tan\theta_1$ and $m_2 = \tan\theta_2$.

Then, $\theta = \theta_1 \sim \theta_2$ (since if $m_1 > m_2$, $\theta_1 - \theta_2$ otherwise $\theta_2 - \theta_1$)

Taking tan of both side

$$\tan\theta = \tan(\theta_1 \sim \theta_2)$$

$$= \frac{\tan\theta_1 \sim \tan\theta_2}{1 + \tan\theta_1 . \tan\theta_2}$$

$$\tan\theta = \frac{m_1 \sim m_2}{1 + m_1m_2}$$

$$\theta = \tan^{-1}\left(\frac{m_1 \sim m_2}{1 + m_1m_2}\right) \qquad ...(2.7)$$

If the line equations are $a_1x + b_1y + c_1 = 0$, and $a_2x + b_2y + c_2 = 0$ then by putting the values of $m_1 = -\frac{a_1}{b_1}$ and $m_2 = -\frac{a_2}{b_2}$, we get

$$\theta = \tan^{-1}\left(\frac{b_1a_2 \sim a_1b_2}{a_1a_2 + b_1b_2}\right) \qquad ....(2.8)$$

NOTES

NOTES

Case 1: If both lines are parallel then angle will be zero

$$\tan^{-1}\left(\frac{b_1a_2 \sim a_1b_2}{a_1a_2 + b_1b_2}\right) = 0$$

or $$\left(\frac{m_1 \sim m_2}{1 + m_1m_2}\right) = 0$$

or $$m_1 \sim m_2 = 0$$

or $$m_1 = m_2 \quad \text{i.e., slopes are equal} \qquad ...(2.9)$$

or $$\tan^{-1}\left(\frac{b_1a_2 \sim a_1b_2}{a_1a_2 + b_1b_2}\right) = 0$$

or $$b_1a_2 \sim a_1b_2 = 0$$

or $$a_1b_2 = b_1a_2 \qquad ...(2.10)$$

Case 2: If both are perpendicular then angle should be 90

i.e., $$\tan^{-1}\left(\frac{m_1 \sim m_2}{1 + m_1m_2}\right) = \pi/2$$

or $$\left(\frac{m_1 \sim m_2}{1 + m_1m_2}\right) = \infty$$

or $$1 + m_1m_2 = 0$$

$$m_1m_2 = -1 \qquad ...(2.11)$$

or $$\tan^{-1}\left(\frac{b_1a_2 \sim a_1b_2}{a_1a_2 + b_1b_2}\right) = \pi/2$$

$$\left(\frac{b_1a_2 \sim a_1b_2}{a_1a_2 + b_1b_2}\right) = \infty$$

$$a_1a_2 + b_1b_2 = 0 \qquad ...(2.12)$$

Line Segments

Any line or piece of line having end points is called a line segment. We can find the equation of any line segment using its end points and it can easily be checked whether any point lies on the line segment or not.

A point will be on the line segment if,

(i) It satisfies the equation of the line segment

(ii) Its x-coordinate lies between the x-coordinates of the end points

(iii) Its y-coordinate lies between the coordinates of the end points

NOTES

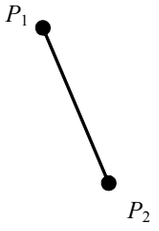


Fig. 2.6 *A Line Segment*

Parametric Form of a Line

The parametric form of a line is expressed in terms of u. This form is very important, specially when we consider any line segment because it gives all the corresponding points of line segments between 0 and 1. When $u = 0$ then it gives x_1 and when u becomes 1 it gives x_2, i.e., x moves uniformly from x_1 to x_2 and the same is true for y values. Therefore

$$0 \le t \le 1$$

An equation can be obtained

$$\left(\frac{y - y_1}{y_2 - y_1}\right) = \frac{x - x_1}{x_2 - x_1} = t$$

equating this equation to u , we get

$$y = y_1 + (y_2 - y_1)t$$
$$x = x_1 + (x_2 - x_1)t$$

these equations are the parametric form of a straight line.

Length of Line Segment

Consider a line segment with end points $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$. We have to find the length of the segment. According to the Pythagorous theorem,

$$(P_1A)^2 + (P_2A)^2 = (P_1P_2)^2$$

or $$(x_2 - x_1)^2 + (y_2 - y_1)^2 = L^2$$

or $$L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Since $$P_1P_2 = L$$

which is the length of line segment P_1P_2.

NOTES

Vector

A vector is defined as the difference between two point positions. Thus for a two dimensional vector, as seen in Figure 2.7, we have

$$\begin{aligned} V &= P_2 - P_1 \\ &= (x_2 - x_1, y_2 - y_1) \\ &= V_x, V_y \end{aligned}$$

where the Cartesian components V_x and V_y are projections of V onto the x and y axis or you can say V_x is the movement along the x-direction and V_y is the movement along the y direction. Given the two point positions, we can obtain vector components in the same way for any co-ordinate frame.

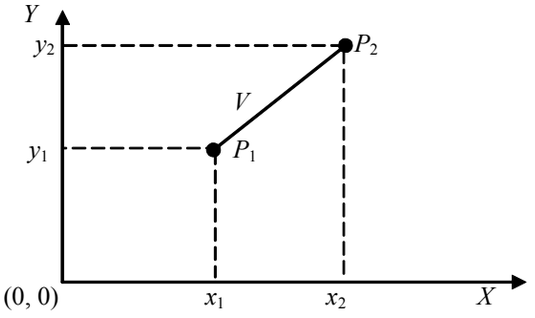


Fig. 2.7 *Vector Representation*

We can describe a vector as a directed line segment that has two main properties: magnitude and direction. For the two dimensional vector in Figure 2.7, we calculate the vector magnitude using the Pythagorous theorem, as follows:

$$|V| = \sqrt{(V_x^2 + V_y^2)}$$

The direction of this two dimensional vector can be given in terms of the angular displacement from the x-axis as follows:

$$\alpha = \tan^{-1}$$

A vector has same properties no matter where we position the vector within the single co-ordinate system, and the vector magnitude is independent of the co-ordinate representation. Of course, if we change the coordinate representation, the values for vector components too change.

Unit Vector

A vector of unit magnitude is called a unit vector. Unit vectors are often used to represent concisely the direction of any vector. The unit vector corresponding to a vector A is written as $\hat{A}$.

Null Vector (Zero Vector)

A vector of zero magnitude which has no direction associated with it is called a zero or null vector and is denoted by **0 (**a thick zero).

(*i*) Vector $\overrightarrow{AB}$ represents the negative of $\overrightarrow{BA}$

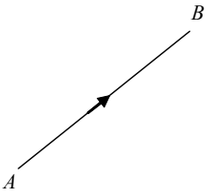


Fig. 2.8 *Vector Representation by a Line*

(*ii*) Two vectors having the same magnitude and same directions are said to be equal and we write $P = Q$

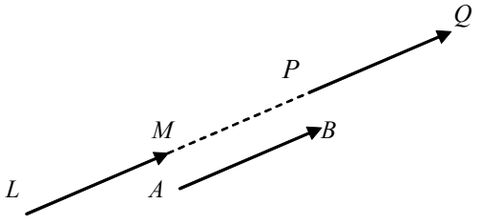


Fig. 2.9 *Two Equal Vectors*

Addition of Vectors

Vectors are added according to the triangle law of addition, which is a matter of common knowledge. Let $\overline{A}$ and $\overline{B}$ be represented by two vectors $\overline{LM}$ and $\overline{MN}$ respectively, then $\overline{LN} = \overline{C}$ is called the sum or resultant of and . Symbolically, we write,

$$\overline{C} = \overline{A} + \overline{B}$$

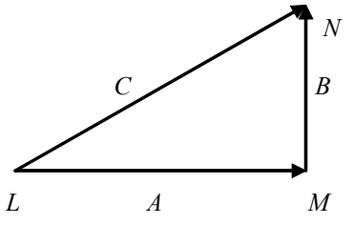


Fig. 2.10 *Addition of Two Vectors*

Subtraction of Two Vectors

The subtraction of a vector $\overline{B}$ from $\overline{A}$ is taken to be the addition of $-\overline{B}$ to $\overline{A}$ and we write it as,

$$\overline{A} + (-\overline{B}) = \overline{A} - \overline{B}$$

NOTES

Multiplication of Vectors by Scalars

We know that $\bar{A} + \bar{A} = 2\bar{A}$

and $(-\bar{A}) + (-\bar{A}) = -2\bar{A}$

where both $2A$ and $-2A$ denote vectors of a magnitude twice of that of A; the former having the same direction as A and the latter the opposite direction.

In general, the product mA of a Vector A and a scalar m is a vector whose magnitude is m times that of A and direction is the same or opposite to A as m is positive or negative.

Thus, $A = a\ \hat{A}$

Resolution of Vectors

Let i, j, k denote a unit vector along OX, OY, OZ respectively. Let $P(x, y, z)$ be a point in space. On OP as diagonal, construct a rectangular parallelepiped with edges OA, OB, OC along axes so that

$$OA = xi$$

$$OB = yi$$

$$OC = zk$$

Then $$R = OP = OC + CP$$

$$= OA + AC' + OC = OA + OB + OC$$

Hence, $R = xi + yj + zk$ is called the position vector of P relative to origin O and

$$r = |R|^2 = \sqrt{(x^2 + y^2 + z^2)}$$

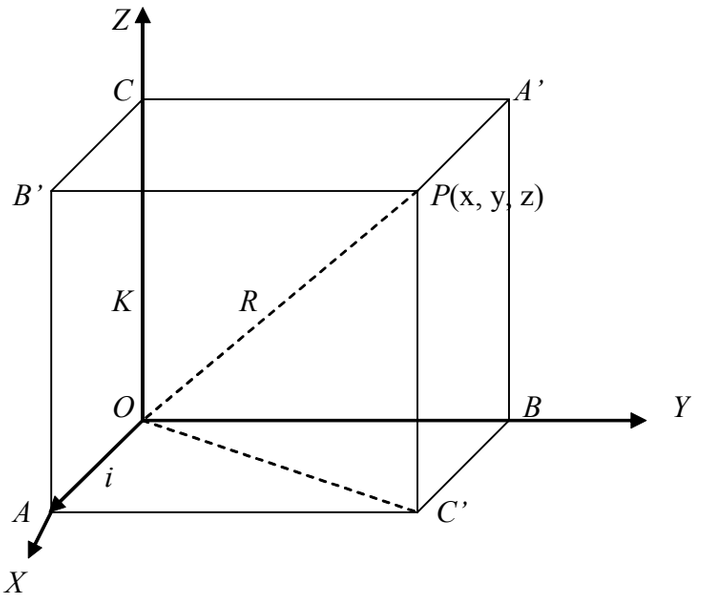


Fig. 2.11 *Resolution of Vectors*

NOTES

Product of Two Vectors

Unlike the product of two scalars or that of a vector by a scalar, the product of two vectors is sometimes seen to result in a scalar quantity and sometimes in a vector. As such, we are lead to define two types of such products: scalar product and vector product respectively. The scalar and vector products of two vectors A and B are usually written as $A.B$ and $A \times B$ respectively and are read as A dot B and A cross B. In view of this notation, the former is sometimes called the dot product and the latter the cross product.

Dot product or scalar product

The dot product of vectors A and B is defined as the product of the length of vector A projected onto B times the length of vector B, or visa versa.

$$\overline{A}.\overline{B} = AB\cos\varphi$$
$$= A_xB_x + A_yB_y + A_zB_z$$

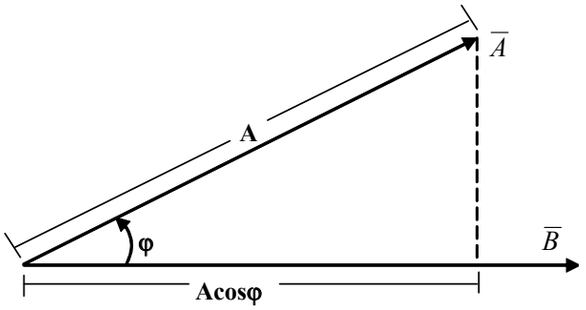


Fig. 2.12 *Scalar Product of Two Vectors*

Length or magnitude of a vector

Make sure that the dot product of a vector with itself is always equal to the square of its magnitude since $\varphi = 0^o$ and $\cos 0^o = 1$.

Parallel vectors

In case $\overline{A}$ and $\overline{B}$ are parallel to each other, then their dot product is similar to the multiplication (ordinary) of their sizes, since $\varphi = 0^o$ and $\cos 0^o = 1$.

$$\overline{A}.\overline{B} = AB$$

Perpendicular vectors

In case $\overline{A}$ and $\overline{B}$ are perpendicular to each other, then their dot product is always Zero, as we know for $\theta = 90^o$ we have $\cos 90^o = 0$.

Components of a vector

The component of vector $\overline{A}$ along a direction d is equal to the dot product of the vector $\overline{A}$ and the unit vector $\hat{d}$ which points along the direction of d.

$$A_d = \overline{A}.\hat{d}$$

NOTES

The Cosines Law:

$$C^2 = A^2 + B^2 - 2AB \cos\Phi$$

Dot Product Proof:

$$\overline{C} = \overline{A} + \overline{B}$$

$$C^2 = \overline{C}.\overline{C} = (\overline{A} + \overline{B}).(\overline{A} + \overline{B}) = \overline{A}.\overline{A} + 2\overline{A}..\overline{B} + \overline{B}.\overline{B}$$

$$= A^2 + 2AB \cos\varphi + B^2$$

$$= A^2 + 2AB \cos(180° - \Phi) + B^2$$

$$= A^2 + B^2 - 2AB \cos\Phi$$

The main property of dot product is that it is distributive.

$$A \bullet (B + C) = A \bullet B + A \bullet C$$

The dot product is commutative. For any two vectors $\overline{A}$ and $\overline{B}$ we have,

$$\overline{A} \bullet \overline{B} = \overline{B} \bullet \overline{A}$$

We know that the projection of a vector on to itself leaves the magnitude of that vector unchanged, the dot product of any vector with itself is the square of that vector's magnitude.

$$\overline{A} \bullet \overline{A} = \overline{A}\,\overline{A} \cos 0° = \overline{A}^2$$

Applying this result to the unit vectors means that the dot product of any unit vector with itself is 1. Additionally, we know that a vector has no projection perpendicular to itself. Therefore the dot product of any unit vector with any other vector is zero.

$$\hat{i}.\hat{i} = \hat{j}.\hat{j} = \hat{k}.\hat{k} = (1)(1)\cos 0 = 1$$

$$\hat{i}.\hat{i} = \hat{j}.\hat{j} = \hat{k}.\hat{k} = (1)(1)\cos 90 = 0$$

By using the above equations, we can originate a general formula for the dot product of any two vectors in a rectangular form. The resulting product appears like it is going to be a dreadful mess, but it consists most of the terms equal to zero.

$$\overline{A} \cdot \overline{B} = (A_x \hat{i} + A_y \hat{j} + A_z \hat{k}).(B_x \hat{i} + B_y \hat{j} + B_z \hat{k})$$

$$= A_x B_x + A_y B_y + A_z B_z$$

Thus, the dot product of two vectors is defined as the sum of the products of their parallel components. By this approach, we can derive the Pythagorous Theorem for three dimensional objects as follows:

$$\overline{A} \cdot \overline{B} = \overline{A}\ \overline{B} \cos 0° = A_x B_x + A_y B_y + A_z B_z$$

$$\overline{A}^2 = A_x^2 + A_y^2 + A_z^2$$

NOTES

Cross product

The cross product of two vectors (represented by) a and b is denoted by $a \times b$. With a right-handed coordinate system in a three-dimensional Euclidean space, $a \times b$ is defined as a vector c that is perpendicular to both a and b.
The cross product of two vectors a and b is defined by the formula

$$a \times b = ab \sin\theta \hat{n}$$

where angle θ is the smaller angle between a and b ($0 \le \theta \le 2\pi$), a and b are the magnitudes of these vectors, and $\hat{n}$ is the unit vector perpendicular to the plane containing vectors a and b. If the vectors a and b are collinear (i.e., the angle between them is either 0 or 2π), then the cross product of a and b is zero.

Computation of cross product

The unit vectors i, j, and k contain an orthogonal coordinate system that satisfies the following equalities:

$i \times j = k$, $j \times k = i$, and $k \times i = j$

Together with the skew-symmetry and bilinear property of the cross product, these identities are sufficient to compute the cross product of any two vectors. Additionally, the following identities also exist:

$j \times i = -k$, $k \times j = -i$, and $i \times k = -j$

and, also

$i \times i = 0 = j \times j = k \times k$.

The coordinates of the cross product of two vectors can be computed easily with the help of the above rules, without the need to determine any angle. Let us consider

$$a = a_1 i + a_2 j + a_3 k$$

which can also be denoted as (a_1, a_2, a_3), and

$$b = b_1 i + b_2 j + b_3 k$$

which can also be denoted as (b_1, b_2, b_3).

The cross product can then be calculated by applying distributive cross-multiplication as follows:

$$a \times b = (a_1 i + a_2 j + a_3 k) \times (b_1 i + b_2 j + b_3 k)$$

$$a \times b = a_1 i \times (b_1 i + b_2 j + b_3 k) + a_2 j \times (b_1 i + b_2 j + b_3 k) + a_3 k \times (b_1 i + b_2 j + b_3 k)$$

$= (a_1 i \times b_1 i) + (a_1 i \times b_2 j) + (a_1 i \times b_3 k) + (a_2 j \times b_1 i) + (a_2 j \times b_2 j) + (a_2 j \times b_3 k) + (a_3 k \times b_1 i) + (a_3 k \times b_2 j) + (a_3 k \times b_3 k)$

NOTES

Since scalar multiplication is commutative with cross multiplication, the right hand side can be regrouped as follows:

$a \times b = a_1 b_1 (i \times i) + a_1 b_2 (i \times j) + a_1 b_3 (i \times k) + a_2 b_1 (j \times i) + a_2 b_2 (j \times j) + a_2 b_3 (j \times k) + a_3 b_1 (k \times i) + a_3 b_2 (k \times j) + a_3 b_3 (k \times k).$

The above equation is the sum of nine terms having cross products. As the multiplication is carried out using the basic cross product relationships between *i*, *j*, and *k* defined above, we have the following:

$a \times b = a_1 b_1 (0) + a_1 b_2 (k) + a_1 b_3 (-j) + a_2 b_1 (-k) + a_2 b_2 (0) + a_2 b_3 (i) + a_3 b_1 (j) + a_3 b_2 (-i) + a_3 b_3 (0).$

This equation can be written in the form given as follows:

$a \times b = (a_2 b_3 - a_3 b_2)\, i + (a_3 b_1 - a_1 b_3)\, j + (a_1 b_2 - a_2 b_1)\, k$

which can be denoted as $(a_2 b_3 - a_3 b_2, a_3 b_1 - a_1 b_3, a_1 b_2 - a_2 b_1)$.

Algebraic Properties

An important property of cross product is that it is anti-commutative, that means

R_1: $a \times b = -b \times a$

The next important property of cross product is that it is distributive over addition,

R_2: $a \times (b + c) = (a \times b) + (a \times c)$,

and it is compatible with scalar multiplication such that

R_3: $(sa) \times b = a \times (sb) = s(a \times b)$.

It is not associative, but satisfies the Jacobi identity:

R_4: $a \times (b \times c) + c \times (a \times b) + b \times (c \times a) = 0$.

It does not obey the cancellation law that means if $a \times b = a \times c$ and also if $a \neq 0$ then we have

$(a \times b) - (a \times c) = 0$

and, by applying the distributive law (given above), we have:

$a \times (b - c) = 0$

Let us consider if vector *a* is parallel to $(b - c)$, then even if $a \neq 0$ it is feasible that $(b - c) \neq$ and therefore $b \neq c$. However, if both $a \bullet b = a \bullet c$ and $a \times b = a \times c$, then it can be accomplished that $b = c$. We can say definitely that

$a \bullet (b - c) = 0$, and $a \times (b - c) = 0$

so that the vector subtraction $(b - c)$ is parallel as well as perpendicular to the non-zero vector *a*. This is only achievable if $(b - c) = 0$. The distributive property, linearity and Jacobi identity show that $(R_2 - R_4)$ together with vector addition and cross product forms a Lie algebra. Additionally, two non-zero vectors *a* and *b* are parallel if and only if they hold

$$a \times b = 0.$$

Check Your Progress

1. What are the conditions for a point to be on a line segment?
2. What is the parametric form of a straight line?
3. What is a zero vector?

NOTES

2.3 LINE-DRAWING ALGORITHMS

This section will discuss the various aspects of line drawing algorithms.

2.3.1 Digital Differential Analyser (DDA)

The DDA algorithm works on the principle of obtaining the successive pixel values based on the differential equation governing that line. For a straight line,

$$\frac{dy}{dx} = \text{constant}$$

i.e., $$\frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

where coordinate positions (x_1, y_1) and (x_2, y_2) are end point coordinates.

We can write the above equation as follows:

$$\Delta y = \frac{y_2 - y_1}{x_2 - x_1} . \Delta x$$

Hence we can obtain the next value of y by using its previous values.

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

or $$y_{k+1} = y_k + \Delta x$$

The DDA algorithm is as follows:

Step 1: Read the end points (x_1, y_1), (x_2, y_2)

Step 2: Approximate the length of the line, i.e.,

if (abs($x_2 - x_1$) > abs($y_2 - y_1$)) then

length = abs($x_2 - x_1$)

otherwise

length = abs($y_2 - y_1$)

Step 3: Select the raster unit, i.e.,

$$x = (x_2 - x_1)/ \text{ length}$$

$$y = (y_2 - y_1)/ \text{ length}$$

***Note*:** either Δx or Δy will be one. Thus the increment value for x or y will be a unit.

Step 4: Round the values by using

$$x = x_1$$
$$y = y_1$$

NOTES

Step 5: Now plot the points

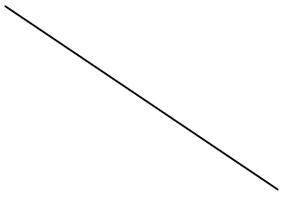
```
 k = 1
 while( k <= length)
{
  plot(int x, int y)
   x = x + Δx
   y = y + Δy
   k = k+1
}
```

Step 6: STOP.

Implementation of Dda Algorithm Using 'C':

```
/*Program to Draw a Line using Digital Differential Analyser
Algorithm*/
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
void main()
{
int gd=DETECT,gm,step,k,dx,dy;
float x,y,x1,x2,y1,y2,xincr,yincr;
clrscr();
initgraph(&gd,&gm,"C:\tc\bgi");
printf("\nEnter the Values of Starting points,(x1,y1):");
scanf("%f %f",&x1,&y1);
printf("\nEnter the Values of Ending points,(x2,y2):");
scanf("%f %f",&x2,&y2);
dx=x2-x1;
dy=y2-y1;
if(abs(dx)>abs(dy))
 step=dx;
else
 step=dy;
xincr=dx/step;
yincr=dy/step;
```

```
x=abs(x1);
y=abs(y1);
putpixel(x,y,RED);
for(k=0;k<step;k++)
 {
 x=abs(x+xincr);
 y=abs(y+yincr);
 putpixel(x,y,BLUE);
 }
getch();
}
The output of this program is as follows:
Enter the Values of Starting points,(x1,y1): 0 0
Enter the Values of Ending points,(x2,y2): 100 160
```



NOTES

2.3.2 Bresenham's Line-Drawing Algorithm

We first consider the scan conversion process for drawing lines with a positive slope that has a value of less than 1. Then we describe Bresenham's line drawing algorithm. Pixel position along a line path can then be determined by sampling at intervals containing unit x. Starting from the left end point (x_0, y_0) of the given line, we step to each succeeding column, i.e., x position and plot the pixel whose scan line y-value is neighboring to the line path.

The following figure demonstrates the n^{th} step in this process. Let us assume that we have determined the pixel to be displayed at coordinate position (x_n, y_n). Now we need to decide which pixel to be plotted in column x_{n+1}. Our options are the pixel at coordinate position (x_{n+1}, y_n) and (x_{n+1}, y_{n+1}).

At sampling position x_{n+1}, we label vertical pixel separations from the mathematical line path as d_1 and d_2. The y co-ordinate on the mathematical line at pixel column position x_{n+1} is calculated as follows:

$$y = m(x_n + 1) + c$$

$$d_1 = y - y_n$$

$$= m(x_n + 1) + c - y_n$$

and

$$d_2 = (y_n + 1) - y$$

$$= y_n + 1 - m(x_n + 1) + c$$

NOTES

The difference between these two separations is

$$d_1 - d_2 = 2m(x_n + 1) - 2y_n + 2c - 1 \quad ...(2.13)$$

The decision parameter P_n for the n^{th} step in the line algorithm is obtained by rearranging the equation (2.13) so that it involves only integer calculations. It can be achieved substituting $m = \frac{\Delta y}{\Delta x}$, where Δy and Δx are the vertical and horizontal separations of the endpoints coordinate positions, and we get,

$$P_n = \Delta x(d_1 - d_2)$$
$$= 2\,\Delta y.x_n - 2\,\Delta x.y_n + e$$

The sign of P_n is the same as the sign of $(d_1 - d_2)$, since $\Delta x > 0$ for this example. Parameter e is constant and has the value $2\,\Delta y + \Delta x(2c - 1)$, which is an independent pixel at y_n which is closer to the line path than to the pixel at y_{n+1}, that means $d_1 < d_2$, therefore the decision parameter P_n is negative. In that case, we can plot the lower pixel, otherwise, we should plot the upper pixel.

The coordinate values that change along the line occur in either the x or y directions depending upon the unit steps. Therefore, we can determine the values of consecutive decision parameters by using incremental integer computations. The decision parameter at step $n + 1$, can be calculated as follows:

$$P_{n+1} = 2\,\Delta y.x_{n+1} - 2\,\Delta x.y_{n+1} + e$$

Therefore,

$$P_{n+1} - P_n = 2\,\Delta y(x_{n+1} - x_n) - 2\,\Delta x(y_{n+1} - y_n)$$

Since $x_{n+1} = x_n + 1$, therefore

$$P_{n+1} = P_n + 2\,\Delta y - 2\,\Delta x(y_{n+1} - y_n)$$

where the term $(y_{n+1} - y_n)$ is equal to either 0 or 1, depending upon the sign of parameter P_n. The recursive calculations of decision parameter can be performed at each integer x-position, starting from the left coordinate end point of the line. The first parameter P_0 can be determined at the starting pixel position (x_0, y_0) and with m calculated as $\frac{\Delta y}{\Delta x}$

$$P_0 = 2y - x$$

The Bresenham's line drawing algorithm is as follows**:**

Step 1: Input the two line end points and store the left end points in (x_0, y_0).

Step 2: Load the point (x_0, y_0) into the frame buffer that has to be plotted as the first point.

Step 3: Compute constants values x, y and obtain the starting coordinate value for the decision parameter as,

$$P_0 = 2\,\Delta y - \Delta x$$

Step 4: At each x_n along the line, starting at $n = 0$, we apply the following test criteria:

If $P_n < 0$, the next point to plot is (x_{n+1}, y_n) and

$P_{0+1} = P_0 + 2 \Delta y$

else

the next point to be plotted is (x_{n+1}, y_{n+1}) and

$P_{0+1} = P_0 + 2\Delta y - 2\Delta x$

Step 5: Step 4 is repeated x times.

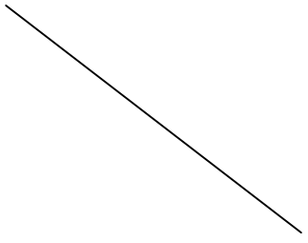
Implementation of Bresenham's line algorithm using 'C':

```
/*Program to draw a Line using Bresenham's Algorithm*/
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
void main()
{
int d,ds,dt,dx,dy, gd=DETECT,gm;
float x,y,x1, y1, x2,y2, xend;
clrscr();
initgraph(&gd,&gm,"C:\tc\bgi");
printf("\nEnter the Values of Starting points,(x1,y1):");
scanf("%f %f",&x1,&y1);
printf("\nEnter the Values of Ending points,(x2,y2):");
scanf("%f %f",&x2,&y2);
dx = x2 - x1;
dy = y2 - y1;
d = 2*dy - dx;
ds = 2*dy;
dt = 2*(dy-dx);
if(x1>x2)
 {
 x = x2;
 y = y2;
 xend = x1;
 }
else
```

NOTES

NOTES

```
 {
 x = x1;
 y = y1;
 xend = x2;
 }
putpixel(x,y,BLUE);
while(x<xend)
 {
 x = x+1;
 if(d<0)
 d = d + ds;
 else
 {
 d = d + dt;
 y = y + 1;
 }
 putpixel(x,y,BLUE);
 }
getch();
}
The output of this program is as follows:
Enter the Values of Starting points,(x1,y1): 10 10
Enter the Values of Ending points,(x2,y2):160 150
```



2.3.3 Bresenham's Circle Algorithm

Bresenham developed an incremental circle generator algorithm, which is more efficient. Conceived for use with pen plotters, the algorithm generates all points on a circle centered at the origin by incrementing all the way around the circle. This algorithm uses the approach of midpoint, which is for the case of integer center point and radius, generating the same, optimal set of pixels.

NOTES

We consider only 45^0 of a circle, the second octant from $x = 0$ to $x = y = R/\sqrt{2}$, and use the circle points procedure to display points on the complete circle. The strategy is to select which of the two pixels is closer to the circle by evaluating a function at the midpoint between the two pixels. In the second octant, if the pixel at (x_p, y_p) has been previously chosen as closest to the circle, the choice of the next pixel is between pixel *E* and *SE*.

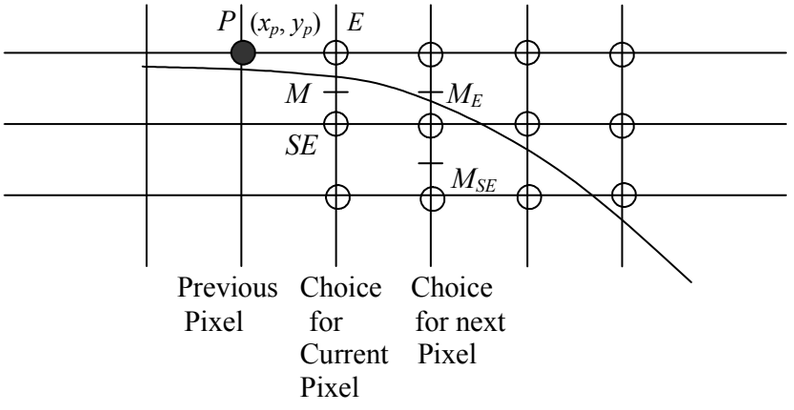


Fig. 2.13 *The Pixel Grid for Midpoint Circle Algorithm*

Let $F(x, y) = x^2 + y^2 - R^2$, this function is 0 on the circle, positive outside the circle, and negative inside the circle. It can be shown that if the midpoint between the pixel *E* and *SE* is outside the circle, then pixel *SE* is closer to the circle. On the other hand, if the midpoint is inside the circle, pixel *E* is closer to the circle.

As for lines, we choose on the basis of the decision variable *d*, that is, the value of the function at the midpoint,

$$d_{old} = F(x_p+1, y_p - \frac{1}{2}) = (x_p+1)^2 + (y_p - \frac{1}{2})^2 - R^2$$

If $d_{old} < 0$, and *E* is chosen, the next midpoint will be one increment over in *x*. Then,

$$d_{new} = F(x_p+2, y_p - \frac{1}{2}) = (x_p+2)^2 + (y_p - \frac{1}{2})^2 - R^2$$

and $$d_{new} = d_{old} + (2\,x_p + 3).$$

Therefore the increment is$_E = 2\,x_p + 3$ if $d_{old} >= 0$, *SE* is selected, and the next midpoint will be one increment over in *x* and one increment down in *y*, then

$$d_{new} = F(x_p+2, y_p - \frac{3}{2}) = (x_p+2)^2 + (y_p - \frac{3}{2})^2 - R^2$$

since $d_{new} = d_{old} + (2\,x_p - 2\,y_p + 5)$, the increment $\Delta_{SE} = 2\,x_p - 2\,y_p + 5$

Note that, in the linear case, Δ_E and Δ_{SE} are constant. However, in the quadratic case, Δ_E and Δ_{SE} vary at each step and are functions of the particular values of x_p

NOTES

and y_p at the pixel chosen in the previous iteration. Because these functions are expressed in terms of (x_p, y_p), we call P the point of evaluation. The Δ function can be evaluated directly at each step by plugging in the values of x and y for the pixel chosen in the previous iteration. This direct evaluation is not expensive computationally, since the functions are only linear.

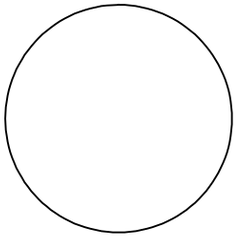
Bresenham's circle drawing algorithm implementation is as follows:

```
/*Program to Draw a Circle using Bresenham Algorithm*/
#include<graphics.h>
#include<stdio.h>
#include<math.h>
#include<conio.h>
void main()
{
int graphd=DETECT, graphm;
float x = 0,y,h,k,r;
float d = 3-(2*r);
clrscr();
initgraph(&graphd, &graphm, "C:\tc\bgi");
printf("\nEnter the Radius of Circle:");
scanf("%f",&r);
y=r;
printf("\nEnter the Centre coordinates, (h,k):");
scanf("%f %f",&h,&k);
while(x<y)
{
 putpixel(x+h,y+k,RED);
 putpixel(y+h,x+k,RED);
 putpixel(-y+h,x+k,RED);
 putpixel(-x+h,y+k,RED);
 putpixel(-x+h,-y+k,RED);
 putpixel(-y+h,-x+k,RED);
 putpixel(y+h,-x+k,RED);
 putpixel(x+h,-y+k,RED);
 if(d<0)
 {
 d=d+(4*x)+6;
 x =x+1;
```

```
 }
 else
 {
 d=d+4*(x-y)+10;
 x =x+1;
 y = y-1;
 }
}
getch();
}
```

Following is the output of this program:

```
Enter the Radius of Circle: 100
Enter the Centre coordinates,(h,k): 120 130
```



Implementation midpoint circle algorithm:

```
#include<graphics.h>
#include<conio.h>
#include<stdio.h>
#include<math.h>
int h,k, r,sa,ea, a;
int getangle(float x,float y)
{
    float q;
    if(x==0)q=0;
    else
    {
       q=atan(y/x);
    }
    return (q*180/3.1415);
}
Putpixel(int x,int y,int val)
{
    putpixel(x,getmaxy()-y,val);
```

NOTES

NOTES

```
}
void CirclePoints(int x,int y,int val)
{
    a=getangle(x,y);
    if(a>=sa && a<ea)
    Putpixel(x+h,y+k,val);
    a=getangle(y,x);
    if(a>=sa && a<ea)
    Putpixel(y+h,x+k,val);
    a=getangle(-x,y)+180;
    if(a>=sa && a<ea)
    Putpixel(-x+h,y+k,val);
    a=getangle(y,-x)+360;
    if(a>=sa && a<ea)
       Putpixel(y+h,-x+k,val);
    a=(getangle(x,y)+180);
    if(a>=sa && a<ea)
       Putpixel(-x+h,-y+k,val);
    a=(getangle(y,x)+180);
    if(a>=sa && a<ea)
    Putpixel(-y+h,-x+k,val);
    a=getangle(-y,x)+180;
    if(a>=sa && a<ea)
     Putpixel(-y+h,x+k,val);
    a=getangle(x,-y)+360;
    if(a>=sa && a<ea)
     Putpixel(x+h,-y+k,val);
}
void midpointcircle(int radius,int val)
{
    int d=1-radius;
    int deltaE=3;
    int deltaSE=-2*radius+5;
    int x=0;
    int y=radius;
    CirclePoints(x,y,val);
    while(y>x)
    {
       if(d<0)
       {
```

```
            d = d + deltaE;
            deltaE = deltaE + 2;
            deltaSE= deltaSE + 2;
         }
         else
         {
            d = d + deltaSE;
            deltaE = deltaE + 2;
            deltaSE = deltaSE + 4;
            y = y-1;
         }
         x = x+1;
      CirclePoints(x,y,val);
      }
}
void main()
{
      int gd = DETECT, gm;
      int r,ch;
      clrscr();
      do
      {
         printf("Enter the center\n");
         scanf("%d%d",&h,&k);
         printf("Enter the radius:\n");
         scanf("%d",&r);
         printf("1.CIRCLE 2.ARC 3.SECTOR 4.QUIT\n\nEnter the
choice: ");
         scanf("%d",&ch);
         if(ch==2 || ch==3)
         {
            printf("\nEnter the START ANGLE: ");
            scanf("%d",&sa);
            printf("\nEnter the END ANGLE: ");
            scanf("%d",&ea);
         }
         else
         {
            sa=0;
            ea=360;
         }
```

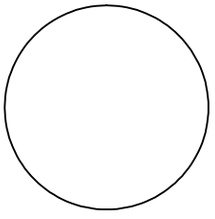
NOTES

NOTES

```
        initgraph(&gd,&gm,"c:\\tc");
        midpointcircle(r,15);
        if(ch==3)
        {
line(h,getmaxy()-k,r*cos(-sa*3.14/180)+h,getmaxy()-
(r*sin(-sa*3.14/180)+k));
        line(h,getmaxy()-k,r*cos(-ea*3.14/180)+h,getmaxy()-
(r*sin(-ea*3.14/180)+k));
        }
        getch();
    }while(ch!=4);
        closegraph();
}
```

The output of this program is as follows:

```
Enter the center 100 100
Enter the radius: 50
1.CIRCLE 2.ARC 3.SECTOR 4.QUIT
Enter the choice: 1
```



The program to draw a circle using polynomial algorithm is as folows:

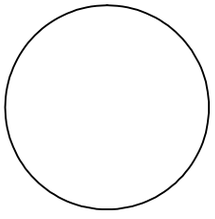
```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
void main()
{
int gd=DETECT,gm;
float x,y;
float h,k,r;
clrscr();
initgraph(&gd,&gm,"C:\tc3\bgi");
printf("\nEnter the Radius of Circle:");
scanf("%f",&r);
printf("\nEnter the Centre coordinates,(h,k):");
scanf("%f %f",&h,&k);
for(x=0;x<=r/sqrt(2);x++)
```

```
{
 y=sqrt((r*r)-(x*x));
 putpixel(x+h,y+k,RED);
 putpixel(y+h,x+k,RED);
 putpixel(-y+h,x+k,RED);
 putpixel(-x+h,y+k,RED);
 putpixel(-x+h,-y+k,RED);
 putpixel(-y+h,-x+k,RED);
 putpixel(y+h,-x+k,RED);
 putpixel(x+h,-y+k,RED);
}
getch();
}
```

NOTES

The output of this program is as follows:

```
Enter the radius of Circle: 50
Enter the Centre coordinates,(h,k): 100 150
```



The circle drawing program using trigonometry function is as follows:

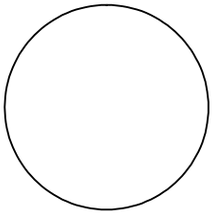
```
/*Program to Draw a Circle using Trigonometric Algorithm*/
#include<graphics.h>
#include<stdio.h>
#include<math.h>
#include<conio.h>
void main()
{
int gd=DETECT,gm;
float x,y;
float h,k,r,angle;
clrscr();
initgraph(&gd,&gm,"C:\tc3\bgi");
printf("\nEnter the Radius of Circle:");
scanf("%f",&r);
printf("\nEnter the Centre coordinates :");
scanf("%f %f",&h,&k);
for(angle=0;angle<=90;angle=angle+0.1)
```

NOTES

```
{
 x=r*cos(angle);
 y=r*sin(angle);
 putpixel(x+h,y+k,RED);
 putpixel(y+h,x+k,RED);
 putpixel(-y+h,x+k,RED);
 putpixel(-x+h,y+k,RED);
 putpixel(-x+h,-y+k,RED);
 putpixel(-y+h,-x+k,RED);
 putpixel(y+h,-x+k,RED);
 putpixel(x+h,-y+k,RED);
}
getch();
}
```

The output of this program is as follows:

```
Enter the Radius of Circle:50
Enter the Centre coordinates :245 245
```



2.4 ELLIPSE GENERATING ALGORITHM

For simplicity, an ellipse having a centre at origin and axes (major and minor) parallel to the coordinate axes is considered. The algebraic expression for ellipse can be written as follows:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

where, $2a$ = length of major axis and $2b$ = length of minor axis.

The above equation can also be written as follows:

$$b^2x^2 + a^2y^2 - a^2b^2 = 0,$$

showing an ellipse can be divided equally into four parts. So if one part (or quadrant) can be generated then the other three parts can easily be simulated by mirroring the original part.

NOTES

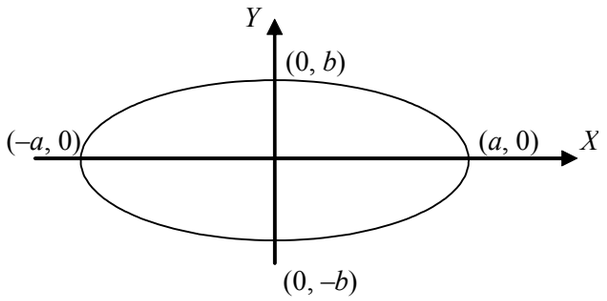


Fig. 2.14 *An Ellipse Centered at the Origin*

Let us generate the first quadrant of the ellipse. For applying the midpoint method the first quadrant is logically divided into two regions as follows:

Region A- closer to the *y*-axis with absolute slope less than 1, and

Region B- closer to the *x*-axis with absolute slope greater than 1.

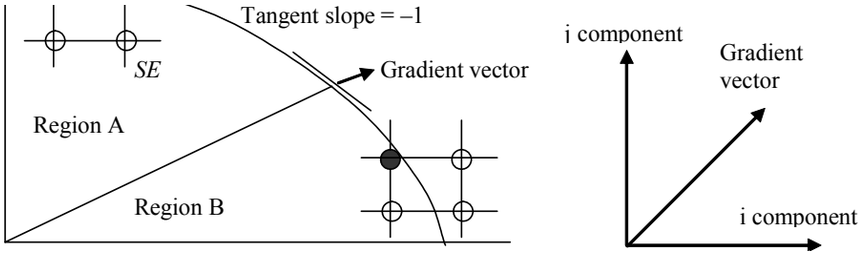


Fig. 2.15 *Two Regions of the Ellipse defined at Tangent of 45°*

Provided we know coordinates (x_i, y_i) of the pixel that lie exactly on the ellipse on the first quadrant we need to find out the next nearest pixel(x_{i+1}, y_{i+1}) using incremental integer value of the decision parameter $f(x, y)$. As in the case of rasterizing a line, we change the unit sampling direction (x or y direction) according to the slope. Here, also, we keep the slope factor in mind. Starting at $(0, b)$ and moving clockwise along the ellipse-path in Region 1 we apply unit steps in x-direction until we reach the boundary between Region 1 and Region 2. Then we change to unit steps in y-direction while sampling Region 2 of the curve.

The partial derivative of function $f(x, y)$ with respect to x and y being $f_x = 2b^2x$ and $f_y = 2a^2y$ respectively, the slope of the ellipse at any point (x, y) is determined by

$$\frac{dy}{dx} = -\frac{f_x}{f_y} = -\frac{b^2x}{a^2y}$$

While deciding whether to plot (x_i+1, y_i) or (x_i+1, y_i-1) as the $(i+1)^{th}$ pixel, the choice is easily made by checking whether the halfway point (the midpoint) between the centers of the two candidate pixels lies inside or outside the theoretical elliptical-path. This is done by calculating $f(x_i + 1, y_i - 1/2)$ and checking the sign. Let,

$$p_1 = f(x_i+1, y_i - \frac{1}{2}) = b^2(x_i+1)^2 + a^2(y_i - \frac{1}{2})^2 - a^2b^2$$

Similarly,

NOTES

$$p_{i+1} = f(x_{i+1}+1, y_{i+1} - \frac{1}{2}) = b^2(x_{i+1}+1)^2 + a^2(y_{i+1} - \frac{1}{2})^2 - a^2b^2$$

$$p_{i+1} - p_1 = b^2(x_{i+1}+1)^2 + a^2(y_{i+1} - \frac{1}{2})^2 - b^2(x_i+1)^2 - a^2(y_i - \frac{1}{2})^2$$

Algorithm for ellipse:

Step 1: Input r_x (the radius of major axis), r_y (the radius of miner axis) and ellipse centre (a, b) and determine the first point on ellipse centered on origin as:

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

Step 2: To calculate the initial value of decision parameter in region 1, the value p_1 is determined as:

$$p_1 = r^2_y + r^2_x + 1/4\ r^2_x$$

Step 3: At each x_k position in Region 1 starting $x = 0$, we perform the following test:

If $p_{1k} < 0$ the next point along the ellipse (x_{k+1}, y_k) and $p_{1x+1} = p_{1k} + 2r^2_y x_{k+1} + r^2_y$

Otherwise $p_{1k} > 0$ the next point along the ellipse (x_{k+1}, y_{k-1})

$$p_{1k+1} = p_{1k} + 2r^2_y x_{k+1} + r^2_y - 2r^2_x y_{k+1}$$

With $2r^2yx_{k+1} = 2r^2yx_k + 2r^2_y$ and $2r^2_x y_{k+1} = 2r^2_x y_k - 2r^2_x$

Step 4: To determine the initial value of the decision parameter in Region 2 with the help of the last point (x_0, y_0) calculated in region 1, we determine p_{20} as:

$$p_{20} = r^2_y(x_0 + 1/2)^2 + r^2_x(y_0 - 1)^2 - r^2_x r^2_y$$

Step 5: For each y_k position in region 2 starting at $k = 0$, we perform the following test:

if $p_{2k} > 0$, the next point along the ellipse (x_k, y_{k-1}) and $p_{2k+1} = p_{2k} - 2r^2_x y_{k+1} + r^2_x$

if $p_{2k} < 0$ the next point along the ellipse is (x_{k+1}, y_{k-1}) and $p_{2k+1} = p_{2k} - 2r^2_x y_{k+1} 2r^2yx_{k+1} + r^2_x$ with the help of the same incremental calculation for x and y as in region 1.

Step 6: The computation of the symmetry points in other three quadrants.

Step 7: Finally we move each calculated pixel position (x, y) on to the elliptical path with center (a, b) and plot the co-ordinates values using

$x = x + a$ and $y = y + b$

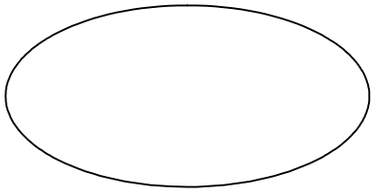
Step 8: Repeat steps for region 1 until $2r^2_y x > 2r^2_x y$.

The program to draw an ellipse using polynomial algorithm is as follows:

```
#include<graphics.h>
#include<conio.h>
#include<stdio.h>
#include<math.h>
void main()
{
int gdriver = DETECT,gmode;
float a,b,h,k;
float x,xend,y;
clrscr();
initgraph(&gdriver,&gmode,"C:\tc\bgi");
printf("\nEnter the Semi Major Axis and Semi Minor Axis:");
scanf("%f %f",&a,&b);
printf("\nEnter the Centre coordinates,(h,k):");
scanf("%f %f",&h,&k);
xend=a;
for(x=0;x<=xend;x = x+0.1)
{
 y=b*sqrt(1-(x*x)/(a));
 putpixel(x+h,y+k,RED);
 putpixel(-x+h,y+k,RED);
 putpixel(-x+h,-y+k,RED);
 putpixel(x+h,-y+k,RED);
}
getch();
}
```

The sample output of this program is as follows:

```
Enter the Semi Major Axis and Semi Minor Axis: 200 150
Enter the Centre coordinates,(h,k): 200 230
```



The following program shows the applications of line and circle/arc functions for representing the motion of a car:

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

NOTES

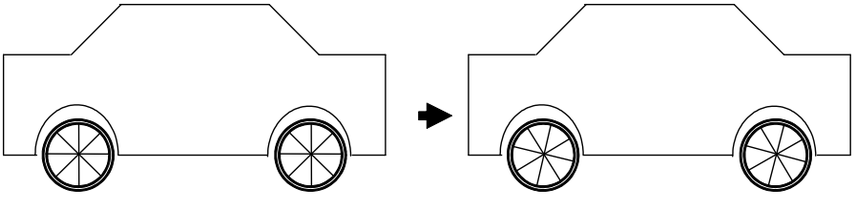
NOTES

```
#include<conio.h>
#include<dos.h>
#include<stdlib.h>
#define PI 3.14159
void draw_wheel(int x,int y,int angle)
{
 int incr=45;
 setcolor(getmaxcolor());
 setfillstyle(EMPTY_FILL,getmaxcolor());
 for(double i = angle;i<angle+360.0;i= i + 2*incr)
 {
 sector(x,y,i,i+incr,20,20);
 arc(x,y,i+incr,i+2*incr,20);
 }
}
void draw_car(int ang)
{
int car_color=BLUE;
draw_wheel(200,200,ang);
draw_wheel(50,200,ang);
setcolor(car_color);
line(0,80,639,80);
line(25,200,0,200);
line(0,300,639,300);
line(0,160,40,160);
line(0,200,0,160);
line(70,130,170,130);
line(40,160,70,130);
line(200,160,260,160);
line(170,130,200,160);
line(260,200,225,200);
line(260,160,260,200);
line(175,200,75,200);
arc(200,200,0,180,25);
arc(50,200,0,180,25);
setfillstyle(SOLID_FILL,car_color);
floodfill(150,170,car_color);
}
void main()
{
```

NOTES

```
int graphd, graphm,i,j;
void *bitmap1,*bitmap2;
detectgraph(&graphd,&graphm);
initgraph(&graphd,&graphm,"c:\\tc ");
draw_car(0);
bitmap1=malloc(imagesize(0,130,270,230));
getimage(0,130,270,230,bitmap1);
putimage(0,130,bitmap1,XOR_PUT);
draw_car(22);
bitmap2 = malloc(imagesize(0,130,270,230));
getimage(0,130,270,230,bitmap2);
putimage(0,130,bitmap2,XOR_PUT);
for(i=0;!kbhit();i= i+10)
{
if(i>500) i=0;
putimage(i,130,bitmap1,OR_PUT);
delay(100);
putimage(i,130,bitmap1,XOR_PUT);
putimage(i+5,130,bitmap2,OR_PUT);
delay(50);
putimage(i+5,130,bitmap2,XOR_PUT);
}
closegraph();
}
```

The output of this program is as follows:



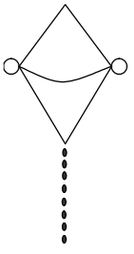
The following program shows the applications of line and circle ellipse functions for representing the motion of a kite:

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

NOTES

```
int graphd=0, graphm=0,i;
clrscr();
initgraph(&graphd,&graphm,"c:\\tc");
do
{
   for(i=500;i>0;i—)
   {
      cleardevice();
      setcolor(WHITE);
      delay(2);
      line(140,60+i,100,100+i); //drawing line1
      line(100,100+i,140,180+i); // drawing line2
      line(140,180+i,180,100+i); // drawing line3
      line(180,100+i,140,60+i); // drawing line4
      setlinestyle(DASHED_LINE,1,3);
      line(140,180+i,140,300+i); // drawing line5
      setlinestyle(SOLID_LINE,1,1);
      circle(95,100+i,6); // drawing circle1
      circle(186,100+i,6); // drawing circle2
      ellipse(140,99+i,180,360,39,16);
   }
}
while(!kbhit());
getch();
closegraph();
return 0;
}
```

The output of this program is as follows:



The following program draws a cube and then rotates it at the desired angle:

```
// cube drawing with the help of line function
#include<graphics.h>
#include<iostream.h>
#include<conio.h>
```

NOTES

```
#include<math.h>
#define PI 3.14
#define offset 15
struct Edge
{
    Vertices v1,v2;
};
struct Vertices
{
    int x,y;
};
struct Cube
{
    Vertices VerticesList[8];
    Edge EdgeList[12];
};
void InitVertices(Cube &aCube,Vertices pivot,int side)
{
    int i;
    aCube.VerticesList[0].x = pivot.x;
    aCube.VerticesList[0].y = pivot.y;
    aCube.VerticesList[1].x = pivot.x+side;
    aCube.VerticesList[1].y = pivot.y;
    aCube.VerticesList[2].x = pivot.x+side;
    aCube.VerticesList[2].y = pivot.y+side;
    aCube.VerticesList[3].x = pivot.x;
    aCube.VerticesList[3].y = pivot.y+side;
    for(i = 4;i < 8;i++)
    {
    aCube.VerticesList[i].x = aCube.VerticesList[i-4].x
+ offset;
    aCube.VerticesList[i].y = aCube.VerticesList[i-4].y
+ offset;
    }
}
void InitEdges(Cube &aCube)
{
    int i,j;
    for(i=0;i<4;i++)
    {
```

NOTES

```
        aCube.EdgeList[i].v1 = aCube.VerticesList[i];
        if(i+1 == 4)
            aCube.EdgeList[i].v2 = aCube.VerticesList[0];
        else
            aCube.EdgeList[i].v2 = aCube.VerticesList[i+1];
    }
    for(i=4;i<8;i++)
    {
        aCube.EdgeList[i].v1 = aCube.VerticesList[i];
        if(i+1 == 8)
            aCube.EdgeList[i].v2 = aCube.VerticesList[4];
        else
            aCube.EdgeList[i].v2 = aCube.VerticesList[i+1];
    }
    for(i=8,j=0;i<12;i++,j++)
    {
        aCube.EdgeList[i].v1 = aCube.VerticesList[j];
        aCube.EdgeList[i].v2 = aCube.VerticesList[j+4];
    }
}
void display(Cube aCube)
{
    for(int i =0;i<12;i++)
    line(aCube.EdgeList[i].v1.x,aCube.EdgeList[i].v1.y,
        aCube.EdgeList[i].v2.x,aCube.EdgeList[i].v2.y);
}
void rotate(Vertices &v,double angle,Vertices pivot)
{
    int oldx = v.x,oldy = v.y;
    oldx = oldx - pivot.x;
    oldy = oldy - pivot.y;
    v.x = (oldx*cos(angle))-(oldy*sin(angle));
    v.y = (oldx*sin(angle))+(oldy*cos(angle));
    v.x = v.x + pivot.x;
    v.y = v.y + pivot.y;
}
void RotateCube(Cube aCube,double angle)
{
    Vertices pivot = aCube.VerticesList[0];
    for(int i = 0;i<8;i++)
```

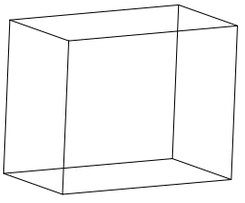
```
        rotate(aCube.VerticesList[i],angle,pivot);
        InitEdges(aCube);
        display(aCube);
}
void main()
{
    int gmode = DETECT,gdriver;
    Cube aCube;
    Vertices pivot;
    int side;
    double angle;
    initgraph(&gmode,&gdriver,"c:\\tc");
    cout << "Enter the reference point to draw the cube:
";
    cin >> pivot.x >> pivot.y;
    cout << "\nEnter the side of the cube: ";
    cin >> side;
    cout << "\nEnter the angle of rotation: ";
    cin >> angle;
    angle = (angle*PI)/180;
    InitVertices(aCube,pivot,side);
    InitEdges(aCube);
    display(aCube);
    outtextxy(50,getmaxx()-50,"Press any key to see
    rotation");
    getch();
    cleardevice();
    RotateCube(aCube,angle);
    getch();
    closegraph();
}
```

The output of this program is as follows:

Enter the reference point to draw the cube: 100 100

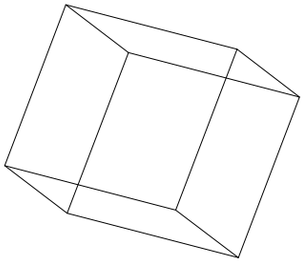
Enter the side of the cube: 40

Enter the angle of rotation:15



NOTES

```
Press any key to see rotation
```



NOTES

Check Your Progress

4. What is the principle on which the DDA algorithm works?
5. What is the algebraic expression of an ellipse?

2.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. A point will be on the line segment if,
   (i) It satisfies the equation of line segment
   (ii) Its x-coordinate lies between x-coordinates of end points
   (iii) Its y-coordinate lies between coordinates of end points
2. The following equations are the parametric form of a straight line:

$$y = y_1 + (y_2 - y_1)t$$

$$x = x_1 + (x_2 - x_1)t$$

3. A vector of zero magnitude which has no direction associated with it is called a zero or null vector and is denoted by 0 (i.e., a thick zero).
4. The DDA algorithm works on the principle of obtaining the successive pixel values based on the differential equation governing that line.
5. The algebraic expression for ellipse can be written as follows:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

where, $2a$ = length of major axis and $2b$ = length of minor axis.

2.6 SUMMARY

- A position in a plane is known as a point and any point can be represented by any ordered pair of numbers (x, y), where x is the horizontal distance from the origin and y is the vertical distance from the origin.

NOTES

- Any line or piece of line having end points is called a line segment.
- A vector of unit magnitude is called a unit vector. Unit vectors are often used to represent concisely the direction of any vector.
- A vector of zero magnitude which has no direction associated with it is called a zero or null vector and is denoted by 0.
- An ellipse having a centre at origin and axes (major and minor) parallel to the coordinate axes is considered.
- The dot product of vectors A and B is defined as the product of the length of vector A projected onto B times the length of vector B, or vice versa.
- The component of vector A along a direction d is equal to the dot product of the vector A and the unit vector ˆd which points along the direction of d.
- The cross product of two vectors (represented by) a and b is denoted by a × b.

2.7 KEY WORDS

- **Point:** A position in a plane is known as a point and any point can be represented by any ordered pair of numbers (x, y).
- **Line Segments:** Any line or piece of line having end points is called a line segment. We can find the equation of any line segment using its end points and it can easily be checked whether any point lies on the line segment or not.

2.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Write short notes on the following:

   (a) Planes (b) Points (c) Vectors
2. Derive the equation for the intercept form of the line.
3. Give the general form of the line passing through each pair of the following points:

   (a) (0, 0) and (2, 4)

   (b) (1, 3) and (2, –3)

   (c) (0, –1) and (–1, –1)

   (d) (1.5, 1) and (3. 5, –1.5)
4. What is the role of DDA in computer graphics?

NOTES

Long Answer Questions

1. Describe the properties of a scalar product.
2. Describe the various properties of a vector product with suitable examples.
3. Differentiate between DDA and Bresenham line drawing algorithm.
4. Find the condition for which two lines are perpendicular.
5. Discuss the ellipse generating algorithm.

2.9 FURTHER READINGS

Hearn, Donal and M. Pauline Baker. 1990. *Computer Graphics*. New Delhi: Prentice-Hall of India.

Rogers, David F. 1985. *Procedural elements for Computer Graphics*. New York: McGraw-Hill.

Foley, D. James, Andries Van Dam, Steven K. Feiner and John F. Hughes. 1997. *Computer Graphics Principles and Practice*. Boston: Addison Wesley.

Mukhopadhyay, A. and A. Chattopadhyay. 2007. *Introduction to Computer Graphics and Multimedia*. New Delhi: Vikas Publishing House.

UNIT 3 FILLED AREA PRIMITIVES

NOTES

Structure

3.0 INTRODUCTION

In this unit, you will learn about morphing, which is a common graphics method which is used in many commercials. Morphing is a special effect in motion pictures and animations that changes (or morphs) one image or shape into another through a seamless transition. You will further learn about polygon representation and polygon filling.

3.1 OBJECTIVES

After going through this unit, you will be able to:

- Define morphing
- Discuss how to create a polygon
- Understand various types of polygon filling algorithm

3.2 CLIPPING AND VIEWPORT

Morphing is a common graphics method which is used in many commercials. It transforms (metamorphoses) one object into another. This method is commonly used in TV commercials: oil can take the shape of a car, a tiger can be morphed into a bike, and one person's face can be morphed into that of another. All these are examples of morphing. Morphing can be applied to any motion or transition involving a change in shape.

Polygon Representation

A many sided figure is termed as a polygon. A polygon can be represented by using various line segments which are connected end to end. It can be defined as a series of points which make line segments and are connected. These line segments are called sides or edges of the polygon and the end points are called vertices.

There are two types of polygon. These are as follows:

(i) Convex polygon

(ii) Concave polygon

NOTES

Convex Polygon

If there is a line connecting two interior points of the polygon, which lies completely inside the polygon, then the polygon is called a convex polygon (Figure 3.1).

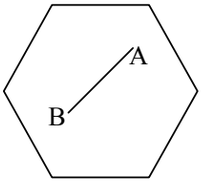


Fig. 3.1 *Illustration of a Convex Polygon*

Concave Polygon

A polygon is called a concave polygon if the line joining any of its two interior points is not completely inside that polygon (Figure 3.2).

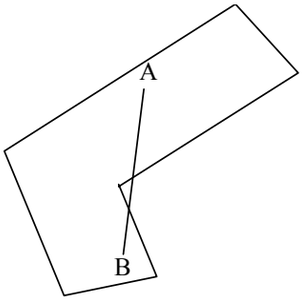
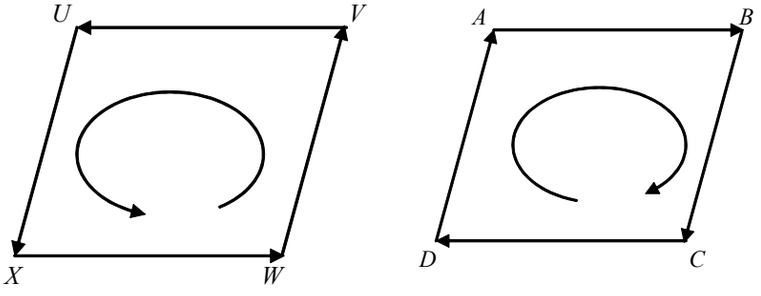


Fig. 3.2 *Illustration of a Concave Polygon*

A polygon having vertices $p_1, p_2, p_3, \ldots\ldots\ldots p_n$ is said to be positively oriented if the visit of the vertices of the polygon in the given order produces an anticlockwise loop (Figure 3.3(a). Similarly, if the visit of the vertices of the polygon in the given order produces a clockwise loop, then it is said to be negatively oriented (Figure 3.3(b).



(a) Positive orientation *(b) Negative orientation*

Fig. 3.3 *Polygon Orientations*

NOTES

There are many methods to represent polygons. Some graphical devices supply a polygon drawing primitive to directly image polygon shape and a polygon can be saved as a unit on these devices. And some devices provide a trapezoid primitive. Trapezoids can be formed by two scan lines and two line segments and can be drawn by stepping down the line segments with two vector generators and, for each scan line, filling in all the pixels between them. Every polygon can be broken up into trapezoids. In other words any polygon can be represented by a series of trapezoids.

How to Create a Polygon

There are two main points needed to make a polygon and to enter it into the display file: the number of sides of polygon and the vertex points. This can be done with the help of two types of coordinates: absolute co-ordinates and relative coordinates.

Algorithm for absolute coordinates

The algorithm for absolute coordinates is as follows:

(i) Input the array containing the vertices of the polygon.
   (a) The number of sides of the polygon
   (b) Coordinates of current pen position and a variable for stepping through the polygon sides.

(ii) Check the number of sides that is 3 or greater than 3, if less than 3, then return invalid polygon.

(iii) Enter the basic instructions for the polygon, i.e., current pen position and number of sides.

(iv) Finally enter the instructions for the sides and return.

Algorithm for relative coordinates

The algorithm for relative coordinates is as follows:

(i) Input the array containing the relative offset the vertices of the polygon
   (a) The number of sides of the polygon
   (b) Coordinates of current pen position and a variable for stepping through the polygon sides

(ii) Check the number of sides that is 3 or greater than 3, if less than 3, then return invalid polygon.

(iii) Increment the current pen position.

(iv) Save the starting point for closing the polygon.

(v) Enter the polygon instructions, i.e., number of sides.

(vi) Enter the instructions for sides and close the polygon then return.

NOTES

Inside and Outside Test

Polygons can be represented in two ways. The first representation is in the form of an outline using move commands. The second representation is in the form of solid objects by setting the pixel high inside the polygon including the pixel on the boundary. The method of finding whether or not a point is inside a polygon is an important issue. There are two approaches to do this. These are as follows:

(i) Even-odd method

(ii) Winding number method

Even-odd method

Those who use this method draw a line segment between the point in question and a point known to be outside the polygon. Just select a point with an *x*-coordinate smaller than the smallest *x*-coordinate of the polygon vertices then draw a line from any position to the distant point chosen outside the coordinate extents of the object and counting the number of edges crossing along the line. If the number of polygon edges crossed by this line is odd, then the point is interior to the polygon, otherwise the point is exterior to the polygon. If we want to find the accurate counting of the edges, then we should select the path which does not intersect the vertices of the polygon.

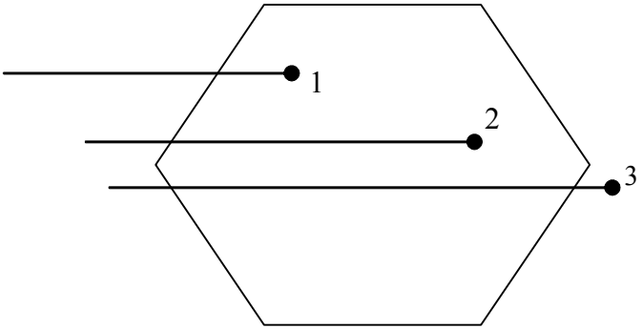


Fig. 3.4 *Illustration of Even-Odd Method*

In Figure 3.4, the line segment from point '1' crosses a single edge, so the point is inside the polygon (because 1 is an odd number). The line from point '3' crosses two edges so the point is outside the polygon. Similarly point '2' is inside the polygon because the line from it crosses a single edge. When a line intersects the vertices of a polygon, then counting is done as follows:

(i) The counting is taken as even if the other end points of the two segments meet at the intersecting vertex.

(ii) The counting is taken as odd if both the endpoints lie on the opposite sides of the constructed line.

NOTES

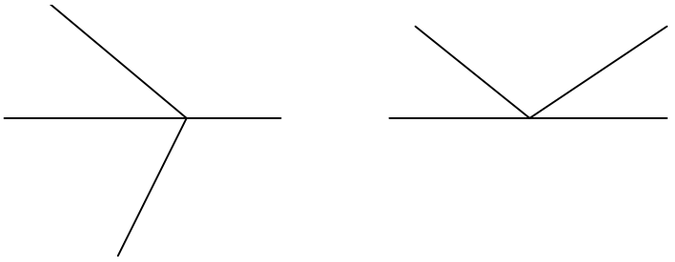


Fig. 3.5 *(a) Odd Count (Total Count = 1), (b) Even Count (Total Count =2)*

Winding number method

Winding number method is another method to define the interior region of a polygon. In this approach we count the number of times the edges of a polygon wind around a particular point in the clockwise direction and we give each boundary line crossed a direction number and we add these direction numbers. The direction number indicates the direction of the polygon edge relative to the line segment we construct for the test. For example to test a given point (x_1, y_1) let us consider a horizontal line segment that sums from the outside of the polygon to (x_1, y_1). Then we find all the sides which cross the given line segment. There are two approaches for a side to cross a given line segment. The side could be drawn starting below the line, then cross it, and finally end above the line. In this case we assign the direction number '–1' to this side. If the edge starts above the line and finishes below it, then we assign the direction number '1' to this side. The sum of direction numbers of the sides which cross the horizontal line gives the winding number for the point. If the winding number is non-zero then the point is inside the polygon, otherwise it is outside the polygon.

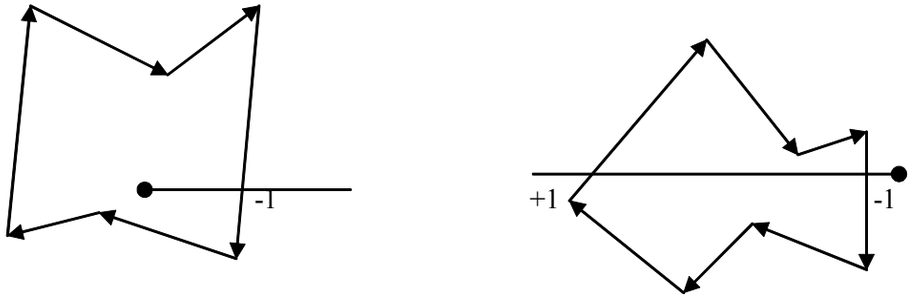


Fig. 3.6 *Computation of Winding Numbers*

From Figure 3.6, you can see that the line segment crosses a single edge and has –1 as the direction number, which is nonzero hence the point is inside the polygon. Similarly we can see from the second diagram that the line crosses two edges and has –1, +1 as the direction number respectively. in which case the winding number = (–1) + (+1) = 0. So the point is outside of the polygon.

NOTES

Polygon Filling

Polygon filling is the process by which an area of a polygon is coloured. Area may be defined as the total number of pixels that outline a polygon. A polygon that is defined by the total number of pixels is called an interior defined polygon. The algorithms used for filling the area of an interior defined polygon are known as flood fill algorithms. A polygon that is defined by the bounding pixels that outline it is called a boundary defined polygon. The algorithms that are used to fill the area of a boundary defined polygon are termed as boundary fill algorithms.

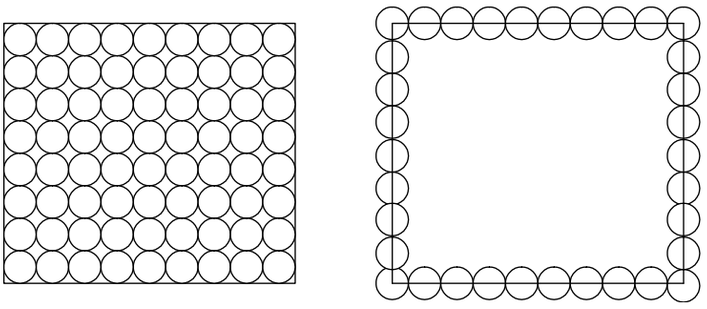


Fig. 3.7 *(a) An Illustration of Flood-Fill, (b) An Illustration of Boundary-Fill*

It is clear that flood fill algorithms and boundary fill algorithms need some starting point inside a polygon, which is called a seed. Therefore, these algorithms are also called seed fill algorithms. These algorithms assume that at least one point interior to the polygon is known to us. Then the algorithm tries to find the all other pixels interior to the polygon and subsequently colours them. There are three types of seed fill algorithms.

(i) Flood Fill Algorithm

(ii) Boundary Fill Algorithm

(iii) Scan-Line Fill Algorithm

3.2.1 Flood Fill Algorithm

In this method we start from the given initial interior pixel, i.e., the seed. From this seed, the algorithm inspects all the surrounding pixels. The surrounding pixels can be seen in the following two ways:

(*i*) ***Four-connected approach*****:** In this approach we check the pixel to the left, right, top, and below from the starting position, i.e., the seed

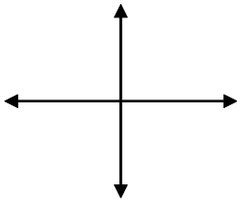


Fig. 3.8 *Four-Connected Approach*

(*ii*) ***Eight-connected Approach*:** In this approach we check the pixel to the left, right, top, below and also check diagonally, which is shown by the following diagram:

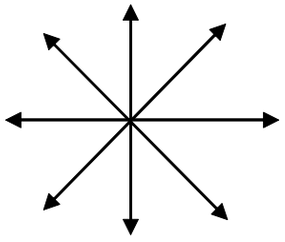


Fig. 3.9 *Eight-Connected Approach*

The following algorithm is shown for the four-connected approach:

```
Procedure Flood fill (x, y, fillcolour, oldcolour: integer)
begin
if getpixel(x, y) = oldcolour then
Setpixel(x, y, fillcolour)
Floodfill(x+1, y, fillcolour, oldcolour);
Floodfill(x-1, y, fillcolour, oldcolour);
Floodfill(x, y+1, fillcolour, oldcolour);
Floodfill(x, y-1, fillcolour, oldcolour);
    end
```

The algorithm will work for the eight-connected approach if we also add the following calls to the four-connected approach

```
Floodfilll(x+1, y-1, fillcolour, oldcolour);
Floodfilll(x+1, y +1,fillcolour, oldcolour);
Floodfilll(x-1, y+1, fillcolour, oldcolour);
Floodfilll(x-1, y-1, fillcolour, oldcolour);
```

3.2.2 Boundary Fill Algorithm

In this algorithm we start at a point inside the polygon and paint with a particular colour. The filling continues until a boundary colour is encountered. There are also two ways to do this. These are as follows:

(i) Four-connected fill where we propagate: left, right, up, and down

(ii) Eight-connected fill where we propagate: left, right, up, down and diagonally also

The algorithm for the four-connected filling approach can be given as follows:

```
Procedure Four Fill (x, y, fillcol, boundcol: integer);
Var
  currcolour: integer;
Begin
```

NOTES

```
  currcolour: = inquirecolour(x, y)
   If(currcolour <> bound colour) and (curr_colour <>
fill_col) then
  Begin
    Setpixel(x, y, fill_col)
    FourFill(x+1, y, fill_col, bound_col);
    FourFill(x-1, y, fill_col, bound_col);
    FourFill(x, y+1, fill_col, bound_col);
    FourFill(x, y-1, fill_col, bound_col);
```

The following problem illustrates four fill:

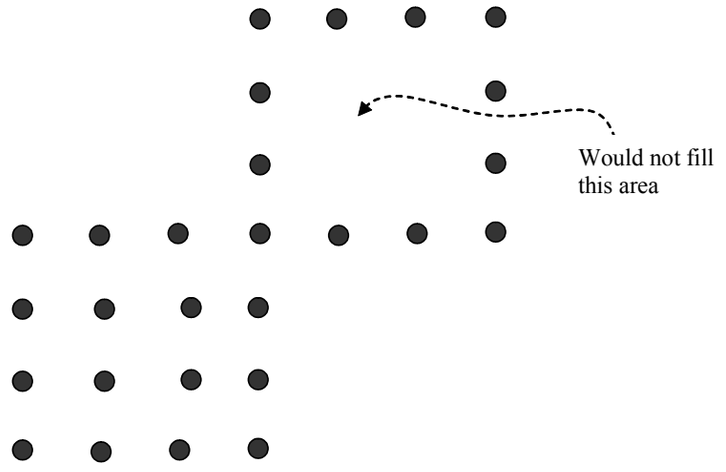


Fig. 3.10 *Boundary Filling*

This leads to the point (ii) wherein eight-connected fill algorithm tests all eight adjacent pixels.

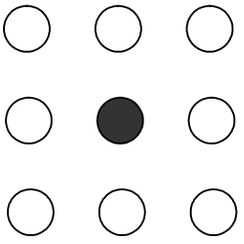


Fig. 3.11 *Eight Adjacent Pixels*

So we add the calls:

```
eightfill (x+1, y-1, fill_col, bound_col);
eightfill (x+1, y+1, fill_col, bound_col);
eightfill (x-1, y-1, fill_col, bound_col);
eightfill (x-1, y+1, fill_col, bound_col);
```

3.2.3 Scan-Line Polygon Fill Algorithm

NOTES

This algorithm tries to find the intersection point of the boundary of a polygon and its scan-lines. These pixels are used to define the pixels inside the polygon. These pixels are set to the required colour. The scan conversion algorithm locates the intersection points of the scan-line with each edge of the polygon. This is done for all the scan-lines starting from left to right. The intersections are grouped and the interior pixel values are set to the colour of the polygon.

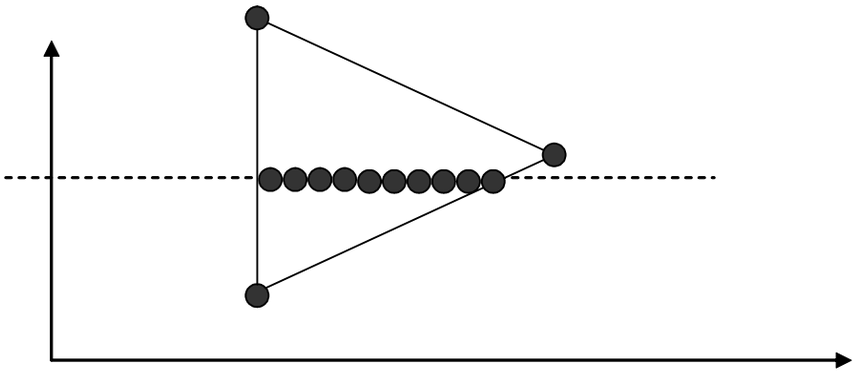


Fig. 3.12 *Scan Line Convex Polygon Filling*

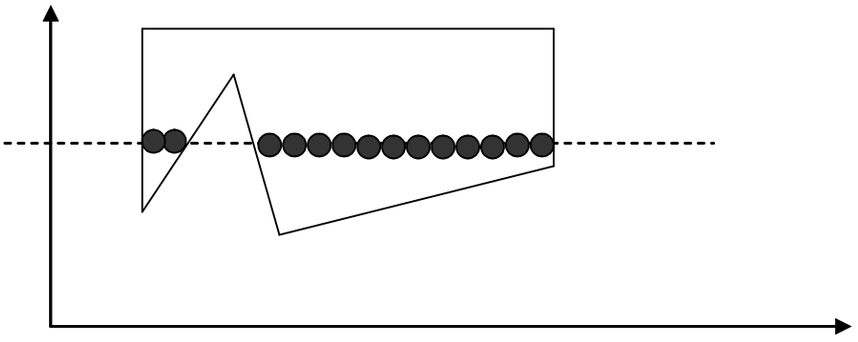


Fig. 3.13 *Scan Line Concave Polygon Filling*

When a scan line intersects a polygon vertex, it may require special handling.

Check Your Progress

1. Name the two types of polygons.
2. List the methods to find whether or not a point is inside a polygon.
3. What is polygon filing?
4. What are flood filled algorithms?

NOTES

3.3 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. There are two types of polygon. These are as follows: (i) Convex polygon (ii) Concave polygon.
2. The method of finding whether or not a point is inside a polygon is an important issue. There are two approaches to do this. These are as follows: (i) Even-odd method (ii) Winding number method
3. Polygon filling is the process by which an area of a polygon is coloured.
4. The algorithms used for filling the area of an interior defined polygon are known as flood filled algorithms.

3.4 SUMMARY

- Morphing is a common graphics method which is used in many commercials. It transforms (metamorphoses) one object into another.
- A polygon can be represented by using various line segments which are connected end to end. It can be defined as a series of points which make line segments and are connected.
- Even-odd method draw a line segment between the point in question and a point known to be outside the polygon.
- Winding number method is another method to define the interior region of a polygon.
- Polygon filling is the process by which an area of a polygon is coloured. Area may be defined as the total number of pixels that outline a polygon. A polygon that is defined by the total number of pixels is called an interior defined polygon.
- This algorithm tries to find the intersection point of the boundary of a polygon and its scan-lines. These pixels are used to define the pixels inside the polygon. These pixels are set to the required colour.
- A polygon can be represented by using various line segments which are connected end to end.

3.5 KEY WORDS

- **Morphing:** It is a common graphics method which is used in many commercials.
- **Interior Defined Polygon:** A polygon that is defined by the total number of pixels is called an interior defined polygon.

- **Boundary Defined Polygon:** A polygon that is defined by the bounding pixels that outline it is called a boundary defined polygon.
- **Boundary Fill Algorithms:** The algorithms that are used to fill the area of a boundary defined polygon are termed as boundary fill algorithms.

NOTES

3.6 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What is morphing?
2. What are the two approaches to find whether or not a point is inside the polygon?

Long Answer Questions

1. Explain the term polygon filling.
2. Explain the following algorithms:
   i. Flood fill algorithm
   ii. Boundary fill algorithm
   iii. Scan line fill algorithm

3.7 FURTHER READINGS

Hearn, Donal and M. Pauline Baker. 1990. *Computer Graphics*. New Delhi: Prentice-Hall of India.

Rogers, David F. 1985. *Procedural elements for Computer Graphics*. New York: McGraw-Hill.

Foley, D. James, Andries Van Dam, Steven K. Feiner and John F. Hughes. 1997. *Computer Graphics Principles and Practice*. Boston: Addison Wesley.

Mukhopadhyay, A. and A. Chattopadhyay. 2007. *Introduction to Computer Graphics and Multimedia*. New Delhi: Vikas Publishing House.

NOTES

BLOCK - II
2D TRANSFORM AND CLIPPING

UNIT 4 2D GEOMETRICAL TRANSFORM

Structure

4.0 INTRODUCTION

You can create a variety of pictures and graphs using the methods that display output primitives and related attributes. Changing or manipulating displays is also required in many applications. Facility layouts and design applications can be created by arranging the orientations and sizes of the component parts of a scene. Animations can be produced by moving the capturing unit or the objects in a scene along animation paths. Changes in shape, size and orientation can be accomplished with geometric transformations that are responsible for altering the coordinate descriptions of objects. As far as the primitive graphic operations are concerned, the basic geometric transformations are translation (shifting the position of an object in the plane), rotation (rotating an object in the plane), and scaling (changing the size of an object in the plane). Other transformations that are often applied to objects include shear and reflection. In this unit, you will learn the general procedures for applying translation, rotation, and scaling parameters to relocate and resize two-dimensional objects.

4.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand translation

- Explain rotation and scaling
- Understand reflection and shear transformations

4.2 TRANSLATION, ROTATION AND SCALING

NOTES

A translation is a transformation that is applied to an object by repositioning it along a straight-line path from one coordinate (location) to another. We can translate a two-dimensional point by adding translation distances, t_x and t_y, to the original position (x, y) to move the point to a new coordinate position (x', y') .

$$x' = x + t_x, \text{ and } y' = y + t_y \quad \ldots.(4.1)$$

The translation distance pair (t_x, t_y) is called a translation vector or shift vector. By this operation the position of a point is shifted t_x in x-direction and t_y in y-direction. We can represent the translation equations 4.1 as a single matrix equation by using column vectors to represent coordinate positions and the translation vector.

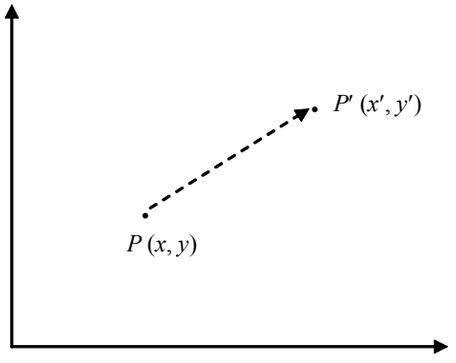


Fig. 4.1 *Translation of a Point from Position P to P′ with Translation Vector T*

$$P = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad P' = \begin{bmatrix} x'_1 \\ x'_2 \end{bmatrix}, \quad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix}, \quad \ldots.(4.2)$$

By using this we can write the two-dimensional translation equations in the following matrix form:

$$P' = P + T \quad \ldots(4.3)$$

Often matrix-transformation equations are represented in terms of coordinate row vectors instead of column vectors. We can write the matrix representations as $P = [x, y]$ and $T = [t_x, t_y]$ since the column-vector representation for a point is a standard mathematical notation. A translation can be treated as a rigid-body transformation which shifts the objects without deformation (twist). That means that every point on the object is shifted by the same amount. We can translate a straight line segment by using the transformation equation 4.3 for each of the line end-points and redrawing the line between the new endpoint positions. Shapes like polygons can be translated by adding the translation vector to the coordinate

NOTES

position of each vertex and regenerating the polygon using the new set of vertex coordinates and the current attribute settings.

Figure 4.2 illustrates the application of a specified translation vector to move an object from one position to another. Similar approaches can be used to translate curved objects. On the other hand, to change the position of an ellipse or circle, we translate the centre coordinates and redraw the figure at a new location. We translate other curves (like splines) by displacing the coordinate positions defining the objects, then the curve paths using the translated coordinate points are reconstructed.

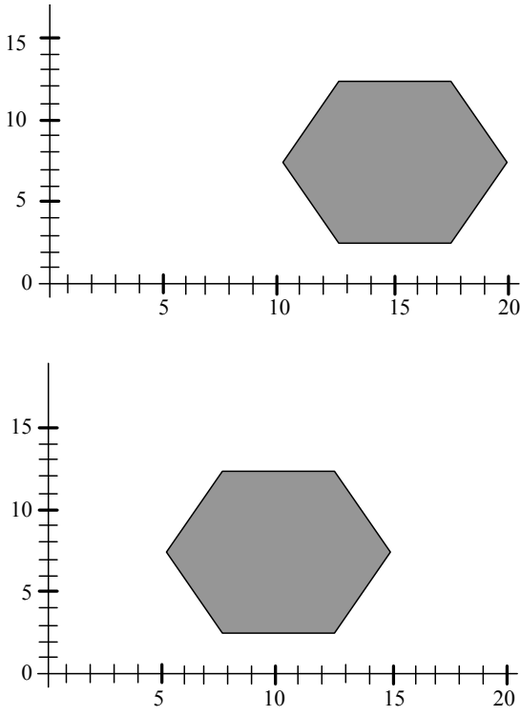


Fig. 4.2 *Moving an Object from one Position to Another*

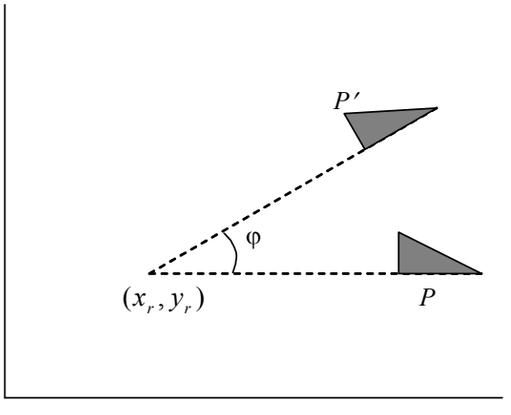


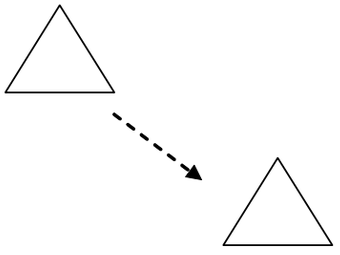
Fig. 4.3 *Rotation of a Rigid Object through an Angle φ about the Point*

NOTES

```
/*Program for Translation of a Triangle*/
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<dos.h>
void main()
{
int gd=DETECT,gm,tx,ty;
initgraph(&gd,&gm,"C:\tc3\bgi");
clrscr();
setbkcolor(RED);
line(100,100,130,150);
line(70,150,130,150);
line(70,150,100,100);
printf("\nEnter the Values ,tx,ty:");
scanf("%d %d",&tx,&ty);
clearviewport();
line(100+tx,100+ty,130+tx,150+ty);
line(70+tx,150+ty,130+tx,150+ty);
line(70+tx,150+ty,100+tx,100+ty);
}
```

The output of this program is as follows

```
Enter the Values ,tx,ty: 100 110
```



Rotation

Two-dimensional rotation can be applied to any kind of object by repositioning it along a circular path in the two-dimensional plane. To initiate a rotation, we have to specify a rotation angle φ and the position of the rotation point (also called pivot point) about which the object is supposed to be rotated. This is illustrated in Figure 4.5. The counterclockwise rotations about the pivot point can be defined by positive values for the rotation angle, as shown in Figure 4.4, and negative values rotate the objects in the clockwise direction. We can also describe transformation as a rotation about a rotation axis which is perpendicular to the *xy* plane and passes through the rotation point. First of all, we determine the

NOTES

transformation equations for rotation of a point position (say *P*) when the rotation (pivot) point is at the coordinate origin.

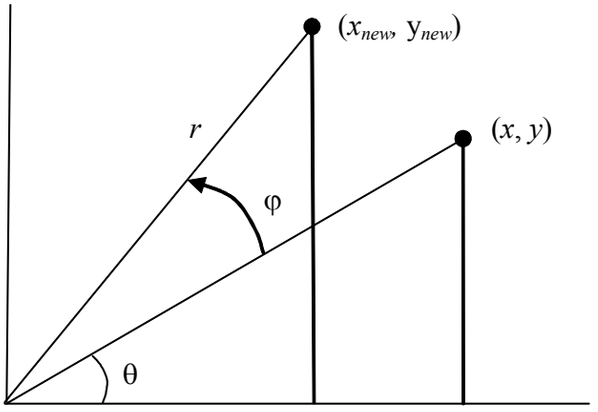


Fig. 4.4 *Illustration of Rotation of a Point from Position (x, y) to Position (x_{new}, y_{new})*

Figure 4.4 illustrates the rotation of a point from initial position (*x, y*) to a new position (x_{new}, y_{new}) through an angle φ relative to the coordinate origin. The angle q is the original angular displacement of the point from the *x* axis. In this figure, *r* is the radial (fixed) distance of the point from the origin, angle θ is the original angular position of the point from the horizontal ($y = 0$), and Φ is the rotation angle. With the help of standard trigonometric identities, we can represent the transformed (final) coordinates in terms of angles Φ and θ by equation 4.4

$$\begin{aligned} x_{new} &= r\cos(\theta + \Phi) = r\cos\theta\cos\Phi - r\sin\Phi\sin\theta \\ y_{new} &= r\sin(\theta + \Phi) = r\sin\theta\cos\Phi + r\cos\theta\sin\Phi \end{aligned} \quad \ldots.(4.4)$$

The original coordinates (*x, y*) of the point in polar coordinates can be represented as

$$x = r\cos\Phi, \qquad \text{and} \qquad y = r\sin\Phi \quad \ldots.(4.5)$$

Substituting expressions of equations 4.5 with 4.4, we can obtain the transformation equations for rotation of a point at position (*x, y*) through an angle Φ about the origin as follows:

$$\begin{aligned} x_{new} &= r\cos\Phi - y\sin\Phi \\ y_{new} &= x\sin\Phi - y\cos\Phi \end{aligned} \quad \ldots.(4.6)$$

With the column-vector representations given by equation 4.2 for coordinate positions, we are able to write the rotation equations in the following matrix form:

$$P' = R.P \quad \ldots(4.7)$$

where the rotation matrix is

$$R = \begin{bmatrix} \cos\Phi & -\sin\Phi \\ \sin\Phi & \cos\Phi \end{bmatrix} \quad \ldots(4.8)$$

The matrix product in rotation as given by equation 4.7, is transposed when coordinate positions are represented as row vectors instead of column vectors, so that the transformed row coordinate vector $[x_{new}, y_{new}]$ is calculated as follows:

$$P'^T = (R.P)^T$$
$$= P^T.R^T$$

where $P^T = [x, y]$ and the transpose R^T of matrix R can be obtained by interchanging the rows and columns. The transpose is obtained by simply changing the sign of the sine terms for a rotation matrix. The rotation of a point about a random pivot position is illustrated in Figure 4.5. Using the trigonometric relationships in this figure, we can generalize equation 4.6 to obtain the transformation equations for rotating a point about some specified rotation position $(x_r y_r)$ as follows:

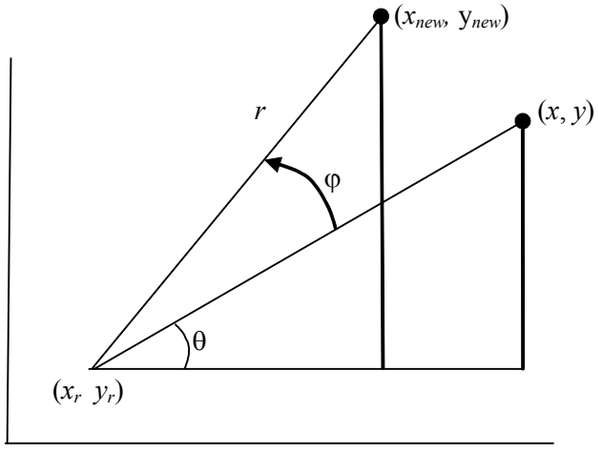


Fig. 4.5 *Rotation of a Point from Position (x, y) to Position (x_{new}, y_{new}) by an Angle φ about Rotation Point $(x_r y_r)$.*

$$x_{new} = x_r + (x - x_r)\cos\Phi - (y - y_r)\sin\Phi$$
$$y_{new} = y_r + (x - x_r)\sin\Phi + (y - y_r)y\cos\Phi \quad ...(4.9)$$

These general rotation equations differ from equation 4.6 by the inclusion of additive terms, as well as the multiplicative factors on the coordinate values. Thus, the matrix expression given by equation 4.7 can be modified to include: pivot coordinates by matrix addition of a column vector whose elements contain the additive (translational) terms in equation 4.9. However, there are better ways, to formulate such matrix equations.

Like translations, rotations are also rigid-body transformations such that objects move without deformation. Every point available on the object is rotated through the same angle. We can rotate a straight line segment by applying the rotation formula given in equation 4.9 to each of the line end points and re-drawing the line between the new end point positions. Polygons can be rotated by displacing each vertex through the specified rotation angle and re-generating the polygon

NOTES

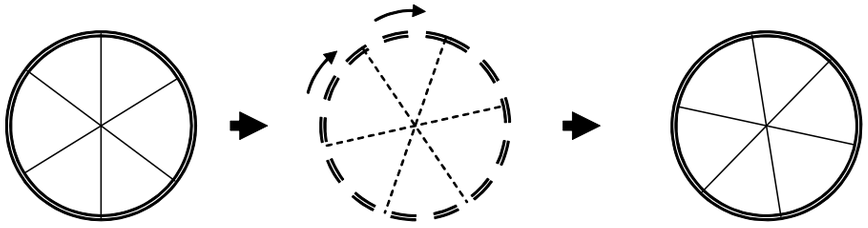
NOTES

with the help of the new vertices. The curves can be rotated by repositioning the defining points and redrawing the curves. For example, an ellipse or a circle can be rotated about any point by moving the centre position through an arc that subtends the particular rotation angle. We can rotate an ellipse about its centre coordinates by rotating the minor and major axes.

The following is the program of wheel rotation and translation parallel to x-axix:

```
#include<graphics.h>
#include<stdio.h>
#include<math.h>
#include<conio.h>
#define PI 3.14159
void rotate_wheel(int xc,int yc,int t)
{
    int x,y;
    for(t=t;t<180;t=t+60)
    {
       x=50*cos(t*PI/180);
       y=50*sin(t*PI/180);
       line(xc+x,yc+y,xc-x,yc-y);
    }
    circle(xc,yc,50);
    circle(xc,yc,52);
}
void main()
{
    int gd=0,gm=0,x;
    initgraph(&gd,&gm,"c:\\tc");
    for(x=0;x<640;x++)
    {
       setcolor(RED);
       rotate_wheel(x,240,x%60);
       delay(5);
       cleardevice();
       rotate_wheel(x,240,x%60);
    }
    getch();
    closegraph();
}
```

The output of this program is as follows:



NOTES

Scaling

Scaling is that transformation which alters the size of an object. This operation is carried out for polygons by multiplying the coordinate values (x, y) of each vertex with scaling factors S_x (in x direction) and S_y (in y direction) to produce the transformed coordinates (x_{new}, y_{new}) as follows:

$$x_{new} = x.\ \ S_x, \qquad y_{new} = y.\ \ S_y \qquad ...(4.10)$$

The scaling factor S_x scales the object in the x-direction, and S_y scales the object in the y-direction. The transformation equations 4.10 can also be written in the following matrix form:

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \qquad ...(4.11)$$

or
$$P' = S \cdot P \qquad ...(4.12)$$

where S is the 2×2 scaling matrix in equation 4.11. Any positive numeric value (integer or float) can be assigned to the scaling factors S_x and S_y. The values of scaling factors less than 1 reduce the size of objects, and the values greater than 1 produce an enlargement. The size of the object remains unchanged for a value of 1 for both S_x and S_y. A uniform scaling is produced that maintains relative object proportions when S_x and S_y are assigned the same value. Different values for S_x and S_y result in a differential scaling which is often used in design applications. The construction of pictures from a few basic shapes can be adjusted by scaling and positioning transformations.

The objects transformed by using equation 4.11 are both scaled and repositioned. The scaling factors with values greater than 1 are responsible to move coordinate positions further from the origin, while values less than 1 are responsible to move objects closer to the coordinate origin. Figure 4.2 represents the scaling of a line by assigning the value 0.50 to both scaling factors S_x and S_y in equation 4.11. The line length as well as the distance from the origin is reduced by a scaling factor of 0.50.

The location of a scaled object can be controlled by choosing a position, called the fixed point, which remains unchanged after the scaling transformation. The coordinates for the fixed point (x_f, y_f) are chosen as one of the vertices, the

NOTES

object centroid, or any other position on the object. A polygon can then be scaled relative to the fixed point by scaling the distance from each vertex to the fixed point. The scaled coordinates (x_{new}, y_{new}) for a vertex with coordinates (x, y), can be calculated as follows:

$$x_{new} = x_f + (x - x_f)S_x \qquad \text{and} \qquad y_{new} = y_f + (y - y_f)S_y \quad ...(4.13)$$

We can write these scaling transformations to separate the multiplicative and additive terms:

$$\begin{aligned} x_{new} &= x.\ \ S_x\ + x_f(1 - S_x) \\ y_{new} &= y.\ \ Sy\ + y_f(1 - S_y) \end{aligned} \qquad ...(4.14)$$

where the additive terms $x_f(1 - S_x)$ and $y_f(1 - S_y)$ are constant terms for all points on the object. Including coordinate for a fixed point in the scaling equations is similar to including coordinates for a pivot point in the rotation equations. A column vector is set up first whose elements are the constant terms in equation 4.14, and then this column vector is added to the product $S.P$ in equation 4.12. A detailed discussion in the next section is given on matrix formulation for transformation equations that involve only matrix multiplications.

Polygons are scaled by applying transformations given by equation 4.14 to each vertex and then regenerating the polygon using the transformed vertices. Other objects are scaled by applying the scaling transformation equations to the parameters defining the objects. An ellipse in standard position is resized by scaling the semi-major and semi-minor axes and redrawing the ellipse about the designated centre coordinates. Uniform scaling of a circle is performed by simply adjusting the radius. Then the circle is redisplayed about the centre coordinates using the transformed radius.

Check Your Progress

1. What is translation?
2. How is scaling carried out for polygons?

4.3 REFLECTION AND SHEAR TRANSFORM

The basic transformations such as translation, rotation, and scaling are incorporated in most graphics softwares. Some softwares provide a few additional transformations which are useful in several applications. Two such kinds of transformations are reflection and shear.

4.3.1 Reflection

Reflection is a transformation that produces a parallel mirror image of an object. The mirror image for a two-dimensional reflection can be generated relative to an

NOTES

axis of reflection by rotating the object by 180° about the reflection axis. An axis of reflection can be chosen in the *XY*-plane or perpendicular to the *XY*-plane. When the reflection axis is across a line in the *XY*-plane, the rotation path about this axis will be in a plane perpendicular to the *XY*-plane. For reflection of the axes that are perpendicular to the *XY*-plane, the rotation path is in the *XY*-plane. Examples of reflection about line $y = 0$ are as follows:

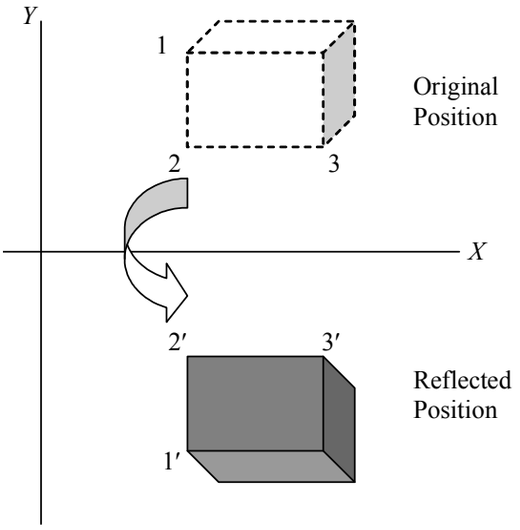


Fig. 4.6 *Reflection of an Object about X-axis*

The reflection of the object about the line $y = 0$ (i.e., the *X*-axis), can be accomplished with the following transformation matrix:

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

....(4.15)

This transformation keeps *x*-values the same, but flips the *y* values of the coordinate positions. The resulting orientation of an object after it has been reflected about the *x*-axis is shown in Figure 4.6. To envision the rotation transformation path for this reflection, we can consider a flat object moving out of the *XY*-plane and rotating 180° through the three-dimensional space about the *x*-axis and back into the *XY*-plane on the other side of the *x*-axis. A reflection about the *y*-axis flips *x*-coordinates while the *y*-coordinates remain the same. The transformation matrix for this is as follows:

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

...(4.16)

NOTES

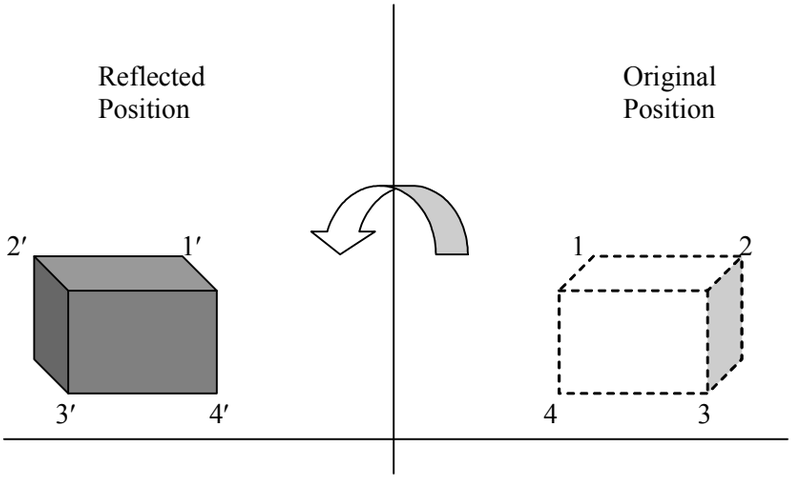


Fig. 4.7 *Reflection of an Object about the Y-axis*

Figure 4.7 illustrates the change in the position of an object that has been reflected about the line $x = 0$. The equivalent rotation in this case is 180° through three dimensional space about the axis. We flip both the x and y coordinates of a point by reflecting relative to an axis that is perpendicular to the XY-plane and that passes through the coordinate origin. This transformation, referred to as a reflection relative to the coordinate origin, has the following matrix representation:

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{....(4.17)}$$

An example of reflection about the origin is shown in Figure 4.8. The reflection matrix 4.17 is the rotation matrix $R(\varphi)$ with $\varphi = 180$. We are simply rotating the object in the xy-plane half a revolution about the origin. Reflection given by equation 4.17 can be generalized to any reflection point in the xy-plane (see Figure 4.9). This reflection is the same as a 180° rotation in the xy-plane using the reflection point as the pivot point. If we chose the reflection axis as the diagonal line $y = x$ (see Figure 4.10), the reflection matrix is

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{...(4.18)}$$

NOTES

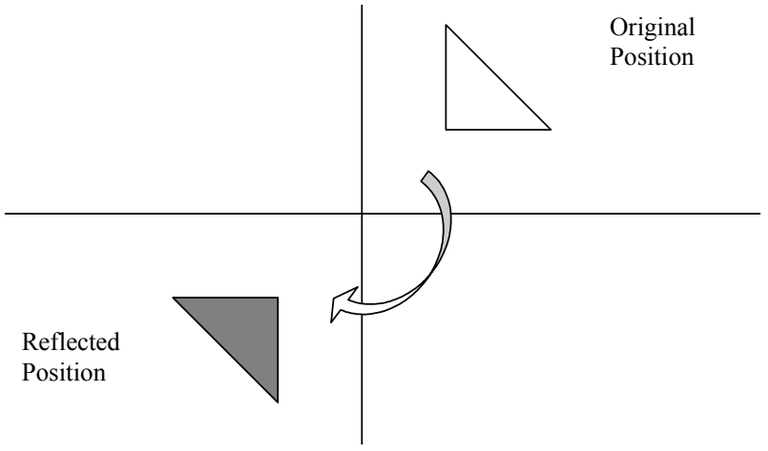


Fig. 4.8 *Reflection of an Object Relative to an Axis Perpendicular to xy-plane*

We can derive this matrix by concatenating a sequence of rotation and coordinate-axis reflection matrices. One possible sequence is shown in figure 4.10. Here, we first perform a clockwise rotation through a 45° angle, which rotates the line $y = x$ onto the x-axis. Next, we perform a reflection with respect to the x axis. The final step is to rotate the line $y = x$ back to its original position with a counterclockwise rotation through 45°. An equivalent sequence of transformations is first to reflect the object about the x-axis, and then to rotate it counterclockwise 90°.

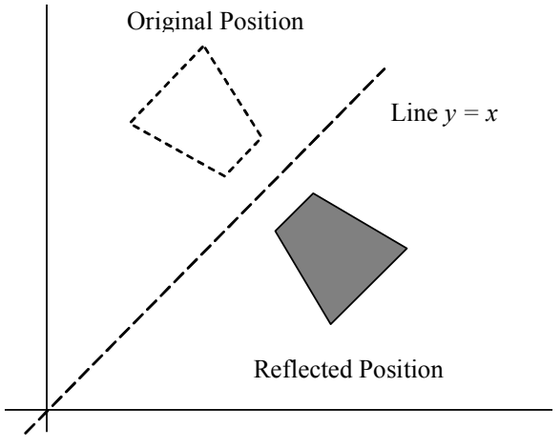


Fig. 4.9 *Reflection of an Object about the Line y = x.*

To obtain a transformation matrix for reflection about the diagonal $y = -x$, we can concatenate matrices for the transformation sequence as follows:

(i) Clockwise rotation by 45°,

(ii) Reflection about the y-axis, and

(iii) Counterclockwise rotation by 45°.

NOTES

The resulting transformation matrix is as follows:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{....(4.19)}$$

Figure 4.9 shows the original and final positions for an object transformed with this reflection matrix. Reflections about any line in the *xy*-plane can be obtained with a combination of translation-rotation-reflection transformations. This way, we first translate the line so that it passes through the origin (0, 0). Then we rotate the line onto one of the coordinate axes and reflect it about that axis. Finally, we bring back the line to its original position with the inverse rotation and translation transformations, respectively. We can implement reflections with respect to the coordinate axes or coordinate origin as scaling transformations with negative scaling factors. Also, elements of the reflection matrix can be set to values other than ±l. Values whose magnitudes are greater than 1 shift the mirror image farther from the reflection (a) axis, and values with magnitudes less than 1 bring the mirror image closer to the reflection axis.

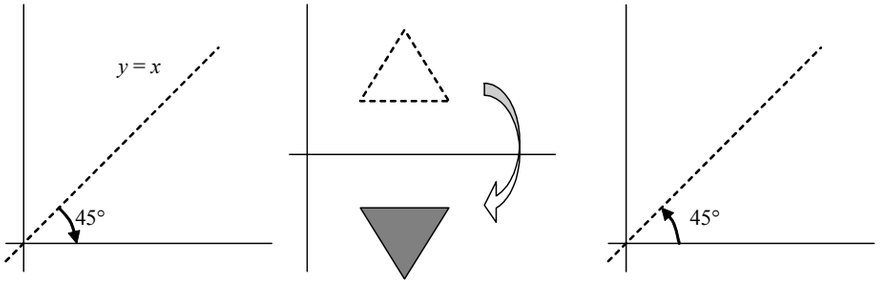


Fig. 4.10 *A Sequence of Transformations to Produce Reflection about the Line y = x*

Figure 4.10 (a) shows a clockwise rotation of 45°. Figure 4.10 (b) shows reflection about the *x*-axis. Figure 4.10 (c) shows a counterclockwise rotation of 45°.

The following is a C++ program for scaling and reflection:

```
// Both Scaling and reflection across y = mx + c of a Home
#include <graphics.h>
#include <iostream.h>
#include <math.h>
#include <conio.h>
#include <stdlib.h>
class Point
{
    public:
       float x,y;
       Point(float tx, float ty)
       {
```

NOTES

```
        x = tx;
        y = ty;
    }
    Point translate(int dx, int dy)
    {
        Point temp(x + dx, y + dy);
        return temp;
    }
    Point scale(float sx, float sy, int tx, int ty)
    {
    Point temp(x, y);
    temp = temp.translate(-tx, -ty).scale(sx,
    sy).translate(tx, ty);
    return temp;
    }
    Point scale(float sx, float sy)
    {
        Point temp(sx * x, sy * y);
        return temp;
    }
    Point reflect_x()
    {
        Point temp(x, -y);
        return temp;
    }
    Point rotate(float theta)
    {
        float tr = theta * M_PI / 180.0;
        Point temp(x*cos(tr) - y*sin(tr), x*sin(tr) +
            y*cos(tr));
        return temp;
    }

};
void draw_home(Point &p1, Point &p2, Point &p3, Point
&p4, Point &p5)
{
    line(p1.x, p1.y, p2.x, p2.y);
    line(p2.x, p2.y, p3.x, p3.y);
    line(p3.x, p3.y, p4.x, p4.y);
    line(p4.x, p4.y, p1.x, p1.y);
    line(p5.x, p5.y, p1.x, p1.y);
    line(p5.x, p5.y, p2.x, p2.y);
}
```

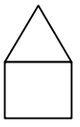
NOTES

```
int main()
{
    int gd, gm, choice;
    detectgraph(&gd, &gm);
    initgraph(&gd, &gm, "c:\\tc\\bgi");
    float sx, sy, x, y, theta, m, c;
    Point p1(60, 20), p2(80, 20), p3(80, 40), p4(60, 40),
    p5(70, 10);
    while(1)
    {
       draw_home(p1, p2, p3, p4, p5);
       cout << "1. Scaling with respect to 0,0," << endl;
       cout << "2. Scaling with respect to arbitary point"
<< endl;
       cout << "3. Reflection about line y = mx + c" <<
endl;
       cout << "4. Exit" << endl;
       cout << "Input the choice : ";
       cin >> choice;
       switch(choice)
       {
          case 1 :    cout << "Input sx and sy: " << endl;
                 cin >> sx >> sy;
draw_home( p1.scale(sx, sy), p2.scale(sx, sy),
p3.scale(sx, sy), p4.scale(sx, sy), p5.scale(sx, sy));
                break;
          case 2 :    cout << "Input sx and sy : ";
                 cin >> sx >> sy;
                 cout << "Input x and y : ";
                 cin >> x >> y;
draw_home( p1.scale(sx, sy, x, y), p2.scale(sx, sy, x,
y), p3.scale(sx, sy, x, y), p4.scale(sx, sy, x, y),
p5.scale(sx, sy, x, y));
                break;
          case 3 :    cout << "Input m and c values : ";
                 cin >> m >> c;
                 draw_home(p1, p2, p3, p4, p5);
                 line(0, c, 640, m*640 + c);
                 theta = atan(m) * 180.0 / M_PI;
                 draw_home(
p1.translate(0,-c).rotate(theta).reflect_x().
rotate(theta).translate(0,c),
p2.translate(0,-c).rotate( theta).reflect_x().
rotate(theta).translate(0,c),
p3.translate(0,-c).rotate(-theta).reflect_x().
rotate(theta).translate(0,c),
```

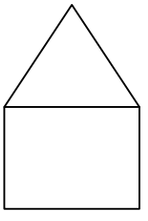
NOTES

```
p4.translate(0,-c).rotate(-theta).reflect_x().
rotate(theta).translate(0,c),
p5.translate(0,-c).rotate(-theta).reflect_x().
rotate(theta).translate(0,c));
                break;
          default :   exit(0);
       }
       getch();
       cleardevice();
    }
    closegraph();
    return 0;
}
```

The output of this program is as follows:



```
1. Scaling with respect to 0,0
    2. Scaling with respect to arbitary point
    3. Reflection about line y = mx + c
    4. Exit
    Input the choice :1
    Input sx and sy: 2 2
1. Scaling with respect to 0,0
    2. Scaling with respect to arbitary point
    3. Reflection about line y = mx + c
    4. Exit
```



4.3.2 Shear

Shear is a transformation that distorts (disturbs) the shape of an object such that the transformed shape of the object appears as if the object were made of several internal layers that had been caused to slide over each other. Two common shearing transformations are those that shift a coordinate in the *x*-direction and those that shift it in the *y*-direction. An *x*-direction shear relative to the *x*-axis can be represented by the following transformation matrix:

NOTES

$$\begin{bmatrix} 1 & SHx & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad ...(4.20)$$

which transforms the coordinate positions as

$$x_{new} = x + SH_x \cdot y, \text{ and } y_{new} = y \quad ...(4.21)$$

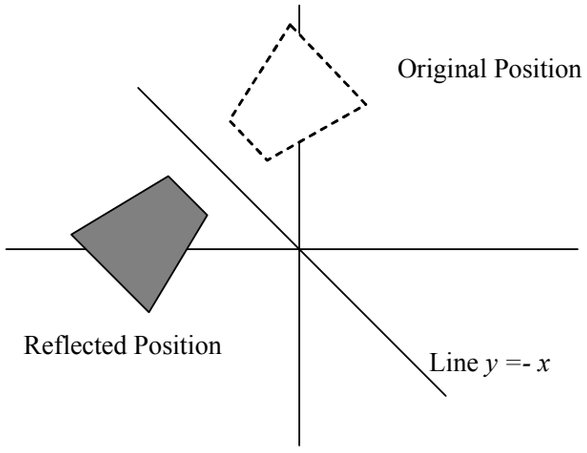


Fig. 4.11 *Reflection with Respect to the Line y = –x*

Any real (float) value can be assigned to the shear parameter SH_x. The coordinate position (x, y) can be shifted horizontally by an amount proportional to its distance (y-value) from the x-axis ($y = 0$). Setting $SH_x = 2$ changes the square into a parallelogram (as shown in Figure 4.12). The negative values for SH_x are responsible to shift coordinate positions to the left. We can generate shears in the x-direction relative to other reference lines with,

$$\begin{bmatrix} 1 & SH_x & -SH_x \cdot y_{ref} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad ....(4.22)$$

with coordinate positions transformed as,

$$x_{new} = x + SHx\ (y - y_{ref}), \text{ and } y_{new} = y \quad ....(4.23)$$

An example of this shearing transformation is given in Figure 4.13 for a shear parameter value of 0.5 relative to the line $y_{ref} = -1$.

NOTES

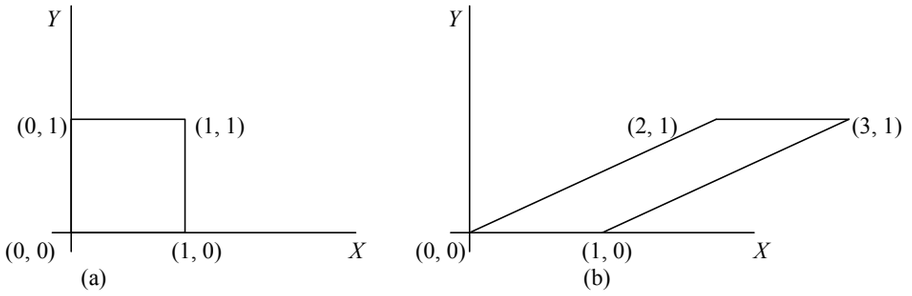


Fig. 4.12 *(a) A Unit Square (b) Conversion into a Parallelogram using the x-direction Shear Matrix 4.20 with* $SH_x = 2$.

A y-direction shear relative to the line $x = x_{ref}$ is generated with the following transformation matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ SH_y & 1 & -SH_y.x_{ref} \\ 0 & 0 & 1 \end{bmatrix} \qquad ...(4.24)$$

which generates transformed coordinate positions

$$x' = x, \qquad \text{and} \qquad y' = SH_x(x - x_{ref}) + y \qquad ...(4.25)$$

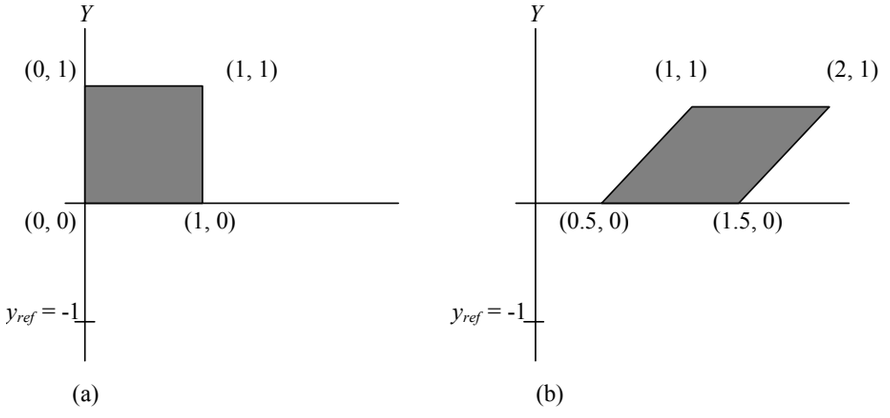


Fig. 4.13 *A Unit Square (a) is Transformed to a Shifted Parallelogram (b) with* $SH_x = 0.5$ *and* $y_{ref} = -1$

This transformation shifts coordinate position vertically by an amount proportional to its distance from the reference line $x = x_{ref}$. Figure 4.14 illustrates the conversion of a square into a parallelogram with $SH_y = 0.5$ and $x_{ref} = -1$. Shearing operations can be expressed as sequences of basic transformation. The x-direction shear matrix 4.20, can be written as a composite transformation involving a series of rotation and scaling matrices that would scale the unit square of figure 4.12 along its diagonal, while maintaining the original lengths and orientations of edges parallel to the x-axis. Shifts in the positions of objects relative to shearing reference lines are equivalent to translations.

NOTES

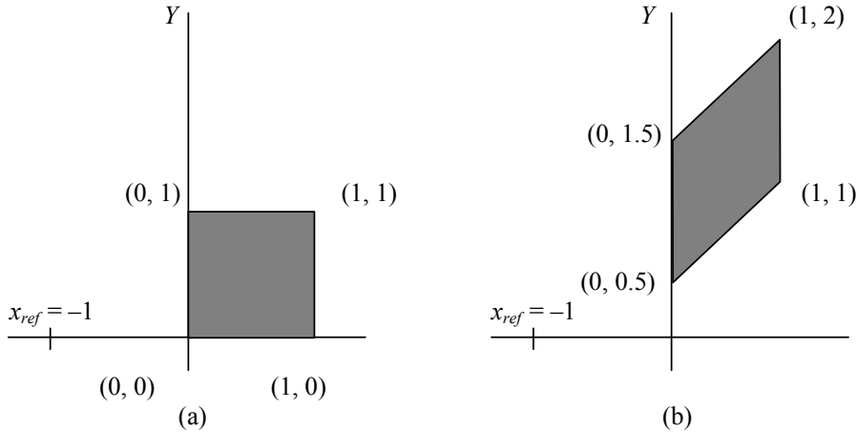


Fig. 4.14 *A Unit Square is transformed into a Shifted Parallelogram with* $SH_y = 1/2$ *and* $x_{ref} = -1$

The following program performs rotation and shearing on a rectangle:

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>
#include <graphics.h>
class Point
{
    public:
       float x, y;
       Point(float tx, float ty)
       {
          x = tx;
          y = ty;
       }
       Point shear(float a, float b)
       {
          Point temp(x+(a*y), y+(b*x));
          return temp;
       }
       Point translate(int dx, int dy)
       {
          Point temp(x + dx, y + dy);
          return temp;
       }
       Point rotate(float theta, int tx, int ty)
       {
```

NOTES

```
            Point temp(x, y);
            temp = temp.translate(-tx, -
            ty).rotate(theta).translate(tx, ty);
            return temp;
        }
        Point rotate(float theta)
        {
            float tr = theta * M_PI / 180.0;
            Point temp(x*cos(tr) - y*sin(tr), x*sin(tr) +
               y*cos(tr));
            return temp;
        }

};
//function fefining for drawing rectangle using line
function
void draw_rectangle(Point &lt, Point &rt, Point &lb, Point
&rb)
{
    line(lt.x, lt.y, rt.x, rt.y);
    line(rt.x, rt.y, rb.x, rb.y);
    line(rb.x, rb.y, lb.x, lb.y);
    line(lb.x, lb.y, lt.x, lt.y);
}
int main()
{
    int graphd, graphm, choice;
    float dx, dy, theta;
//Defining the coordinates of the rectangle
    Point p1(200, 200), p2(350, 200), p3(200, 300), p4(350,
    300);
    detectgraph(&graphd, &graphm);
    initgraph(&graphd, &graphm, "c:\\tc\\tc\\bgi");
    while(1)
    {
    draw_rectangle(p1, p2, p3, p4);
    cout << "1. Rotation about origin followed by
    translation" << endl;
    cout << "2. Rotation about arbitary point" << endl;
    cout << "3. Applying X-shear and Y-shear" << endl;
    cout << "4. Exit" << endl;
    cout << "Input your choice : ";
```

NOTES

```
cin >> choice;
switch(choice)
   {
   case 1 : cout << "Input angle of rotation : ";
         cin >> theta;
         cout << "Input x and y translation values ";
         cin >> dx >> dy;
         draw_rectangle(p1.rotate(theta),
      p2.rotate(theta), p3.rotate(theta),
   p4.rotate(theta));
      draw_rectangle(p1.rotate(theta).translate(dx, dy),
      p2.rotate(theta).translate(dx, dy),
      p3.rotate(theta).translate(dx, dy),
      p4.rotate(theta).translate(dx, dy));
            break;
   case 2 : cout << "Input angle of rotation : ";
         cin >> theta;
         cout << "Input rotation point : ";
         cin >> dx >> dy;
         draw_rectangle(p1.rotate(theta, dx, dy),
                  p2.rotate(theta, dx, dy),
                  p3.rotate(theta, dx, dy),
                  p4.rotate(theta, dx, dy));
         break;
   case 3 : cout << "Input x and y shear values : ";
         cin >> dx >> dy;
         draw_rectangle(p1.shear(0, 0), p2.shear(0,
            dy),
            p3.shear(dx, 0), p4.shear(dx, dy));
            break;
      default :   exit(0);
            break;
   }
   getch();
   cleardevice();
}
```

```
    closegraph();
    return 0;
}
```

NOTES

Check Your Progress

3. What is reflection?
4. What is a shear?

4.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. A translation is a transformation that is applied to an object by repositioning it along a straight-line path from one coordinate (location) to another.
2. Scaling is carried out for polygons by multiplying the coordinate values (x, y) of each vertex with scaling factors Sx (in x direction) and Sy (in y direction) to produce the transformed coordinates.
3. Reflection is a transformation that produces a parallel mirror image of an object.
4. Shear is a transformation that distorts (disturbs) the shape of an object such that the transformed shape of the object appears as if the object were made of several internal layers that had been caused to slide over each other.

4.5 SUMMARY

- A translation is a transformation that is applied to an object by repositioning it along a straight-line path from one coordinate (location) to another.
- Two-dimensional rotation can be applied to any kind of object by repositioning it along a circular path in the two-dimensional plane.
- Scaling is that transformation which alters the size of an object. This operation is carried out for polygons by multiplying the coordinate values (x, y) of each vertex with scaling factors Sx (in x direction) and Sy (in y direction) to produce the transformed coordinates (xnew ,ynew).
- The homogenous parameter h can be chosen to be any nonzero value for two-dimensional geometric transformations. Thus, there are an infinite number of equivalent homogeneous representations to each coordinate point (x, y).

4.6 KEY WORDS

- **Translation:** It involves shifting the position of an object in the plane.

- **Rotation:** It involves rotating an object in the plane.
- **Scaling:** It involves changing the size of an object in the plane.

NOTES

4.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Write a note on the following:
   (i). Translation
   (ii. Rotation
   (iii) Scaling
2. Write a procedure to continuously rotate an object about a pivot point.
3. Show that the composition of two rotations is additive.

Long Answer Questions

1. Write a program for translation of a triangle.
2. Explain the process of translation, rotation and scaling.
3. Explain the reflection and shear transform.

4.8 FURTHER READINGS

Hearn, Donal and M. Pauline Baker. 1990. *Computer Graphics*. New Delhi: Prentice-Hall of India.

Rogers, David F. 1985. *Procedural elements for Computer Graphics*. New York: McGraw-Hill.

Foley, D. James, Andries Van Dam, Steven K. Feiner and John F. Hughes. 1997. *Computer Graphics Principles and Practice*. Boston: Addison Wesley.

Mukhopadhyay, A. and A. Chattopadhyay. 2007. *Introduction to Computer Graphics and Multimedia*. New Delhi: Vikas Publishing House.

UNIT 5 2D MATRIX REPRESENTATIONS

NOTES

Structure

5.0 INTRODUCTION

The most common basic geometric transformations are translation, scaling and rotation. In translations an object is moved in a straight-line path from one position to another. In rotations an object is moved from one position to another in a circular path around a specified pivot point (rotation point). In scaling, the dimensions of an object are changed relative to a specified fixed point. You can express two-dimensional geometric transformations as 3×3 matrix operators, so that sequences of transformations can be concatenated into a single composite matrix. It can be considered as an efficient formulation since it allows you to reduce computations by applying the composite matrix to the initial coordinate positions of an object to obtain the final transformed positions. To perform this, you need to express two-dimensional coordinate positions as three-element column or row matrices. A column-matrix representation can be chosen for coordinate points because this is the standard mathematical convention in many graphics applications.

5.1 OBJECTIVES

After going through this unit, you will be able to:

- Describe matrix representations and homogeneous coordinates

- Explain composite transformations
- Explain the concatenation properties

NOTES

5.2 MATRIX REPRESENTATIONS AND HOMOGENEOUS COORDINATES

There are so many graphics applications that involve sequences of geometric transformations. For example, an animation may require an object to be translated and rotated at each increment of the motion along a linear path. We perform translations, rotations, and scaling, in design and picture construction applications, to fit the picture components into their proper positions. Now, we consider how the matrix representations discussed in the earlier sections can be reformulated so that such transformation sequences can be efficiently processed. Earlier, we have seen that each of the basic transformations are expressed in the following general matrix form

$$P' = M_1.P + M_2 \qquad ...(5.1)$$

where the coordinate positions P and P' are represented as column vectors. Matrix M_1 is a 2×2 array containing multiplicative factors, and matrix M_2 is a two-element column matrix containing translational terms. For translation, matrix M_1 is the identity matrix. For rotation and scaling, matrix M_2 contains the translational terms associated with the rotation point or scaling fixed point. We must calculate the transformed coordinates, one step at a time, to produce a sequence of transformations with these equations, such as scaling followed by rotation and then translation. First, coordinate positions are scaled and then these scaled coordinates are rotated, and finally the rotated coordinates are translated. A more efficient approach can be to combine a sequence of transformations so that the final coordinate positions are obtained directly from the initial coordinates, by eliminating the calculation of intermediate coordinate values. To make it practical, we need to reformulate the equation 5.1 to eliminate the matrix addition associated with the translation terms represented by matrix M_2. Now, we can combine the multiplicative and translational terms for two-dimensional geometric transformations into a single matrix representation by expanding the 2×2 matrix representations into 3×3 matrices. This operation expresses all transformation equations as matrix multiplications, provided that the matrix representations are expended for coordinate positions also. For the representation of any two-dimensional transformation as a matrix multiplication, we need to represent each Cartesian coordinate point (x, y) with the homogeneous coordinate triple (x_h, y_h, h), where

$$x = \frac{x_h}{h}, \quad \text{and} \quad y = \frac{y_h}{h} \qquad ...(5.2)$$

The homogenous parameter h can be chosen to be any nonzero value for two-dimensional geometric transformations. Thus, there are an infinite number of

equivalent homogeneous representations to each coordinate point (x, y). A convenient choice can be simply to set the homogenous parameter $h = 1$. We can represent each two dimensional position having homogeneous coordinates (x, y, 1). Other values for homogeneous parameter h are also desirable in matrix formulations of three-dimensional (3D) viewing transformations. We use the term homogenous coordinates in mathematics to refer to the effect of this demonstration on Cartesian equations. Then we convert a Cartesian point (x, y) to a homogeneous representation (x_h, y_h, h), equations having x and y, such as become homogeneous equation having three attributes x_h, y_h, and h**.** Representing positions in a homogeneous coordinate represents all kinds of geometric transformation equations in the form of matrix multiplications. We can represent the coordinates with three-element column vectors, and transformation operations as 3 × 3 matrices. For translation representation in matrix form, we have,

$$\begin{bmatrix} x_{new} \\ y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad ...(5.3)$$

which can be written in the reduced form as,

$$P' = T(t_x, t_y) \cdot P \quad ...(5.4)$$

with $T(t_x, t_y)$ as the 3×3 translation matrix in equation 5.3. We can obtain the inverse of the translation matrix by replacing the translation parameters t_x and t_y with their negatives values that means $-t_x$ and $-t_y$. In the same way, we can write the rotation transformation equations about the coordinate origin as following:

$$\begin{bmatrix} x_{new} \\ y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\Phi & -\sin\Phi & 0 \\ \sin\Phi & \cos\Phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad ...(5.5)$$

or as

$$P' = R(\mathrm{F}) \cdot P \quad ...(5.6)$$

We can write the rotation transformation operator $R(\Phi)$ as a 3 × 3 matrix as given by equation 5.5 with the rotation parameter Φ. The inverse rotation matrix can be obtained when Φ is replaced with –Φ. Finally, a scaling transformation relative to the coordinate origins can be expressed as the matrix multiplication

$$\begin{bmatrix} x_{new} \\ y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad ...(5.7)$$

or as,

$$P' = S(S_x, S_y) \cdot P \quad ....(5.8)$$

where, $S(S_x, S_y)$ is the 3 × 3 matrix in equation 5.7 with attributes S_x and S_y. By replacing these attributes with their multiplicative inverses (i.e., $1/S_x$ and $1/S_y$) the

NOTES

inverse scaling matrix is produced. Matrix representations can be considered as standard methods for implementation of transformations in graphics. Rotation and scaling functions produce transformations in many systems with respect to the coordinate origin, as given by equations 5.5 and 5.7. We then handle the rotations and scaling relative to other reference positions as a chain of transformation operations. An alternate approach in a graphics package is to provide parameters in the transformation functions for scaling fixed-point coordinates and the pivot-point coordinates General rotation and scaling matrices that include the pivot or fixed point are then set up directly without the need to invoke a succession of transformation functions.

5.3 COMPOSITE TRANSFORMATIONS

With the matrix representations of the previous section, a set-up for a matrix can be made for any series of transformations. This set-up is determined by the product of matrices of the individual transformations. The formation of products of transformation matrices is often referred to as a composition or concatenation of matrices. For column-matrix representation of coordinate positions, we form composite transformations by multiplying matrices in order from right to left. That is, each successive transformation matrix pre-multiplies the product of the preceding transformation matrices.

Translations

If two successive translation vectors $(t_{x1},\ t_{y1})$ and $(t_{x2},\ t_{y2})$ can be applied to a coordinate position P, then the final transformed coordinate position is P', which can be calculated as,

$$P' = T(t_{x2},\ t_{y2})\ .\ (T(t_{x1},\ t_{y1}).P) = (T(t_{x2},\ t_{y2})\ .\ T(t_{x1},\ t_{y1})).P \quad ...(5.9)$$

where P and P' are represented as homogeneous-coordinate column vectors. This result can be verified by calculating the matrix product for the two associative groupings. This way the composite transformation matrix for this succession of translations can be represented as,

$$\begin{bmatrix} 1 & 0 & t_{x2} \\ 0 & 1 & t_{y2} \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & t_{x1} \\ 0 & 1 & t_{y11} \\ 0 & 0 & \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{x1}+t_{x2} \\ 0 & 1 & t_{y1}+t_{y2} \\ 0 & 0 & 1 \end{bmatrix} \quad ...(5.10)$$

It can also be represented as,

$$T(t_{x2},\ t_{y2})\ .\ T(t_{x1},\ t_{y1}) = T(t_{x1}+t_{x2},\ t_{y1}+t_{y2}) \quad ...(5.11)$$

which represents two successive translations that are additive in practice.

NOTES

Rotations

If two successive rotations are applied to a point P, then the transformed position is P' represented as

$$P' = R(\varphi_2) \cdot \{R(\varphi_1) \cdot P\} = \{R(\varphi_2) \cdot R(\varphi_1)\} \cdot P \quad ...(5.12)$$

As we multiply the two rotation matrices, we can easily observe that the two successive rotations are additive,

$$R(\varphi_1) \cdot R(\varphi_2) = R(\varphi_1 + \varphi_2) \quad ...(5.13)$$

As a result we get the composite rotation matrix,

$$P' = R(\varphi_1 + \varphi_2) \cdot P \quad ...(5.14)$$

5.3.1 Consecutive Scaling

The concatenation of transformation matrices for two consecutive scaling operations produces the composite scaling matrix that can be represented as following:

$$\begin{bmatrix} S_{x1} & 0 & 0 \\ 0 & S_{y1} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} S_{x2} & 0 & 0 \\ 0 & S_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} S_{x1} \cdot S_{x2} & 0 & 0 \\ 0 & S_{y1} \cdot S_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

...(5.15)

It can also be written as,

$$S(S_{x1}, S_{y1}) \cdot S(S_{x2}, S_{y2}) = S(S_{x1} \cdot S_{x2}, S_{y1} \cdot S_{y2})$$

....(5.16)

The resulting matrix in this case shows that the successive scaling operations are multiplicative in nature. That is, if we are about to triple the size of an object two times in a chain, the final size would be nine times of the original size.

5.3.2 General Pivot-Point Rotation

With the help of a graphics software that only provides a rotate function for rotating objects about the coordinate origin, we can generate rotations about any selected pivot (rotation) point by performing the following sequence of translate-rotate-translate operations:

(i) Translation of the object such that the rotation -point position is shifted to the coordinate origin.

(ii) Rotation of the object about the coordinate origin.

(iii) Translation of the object such that the rotation point is returned to its original position.

NOTES

This transformation sequence is illustrated in Figure 5.1.

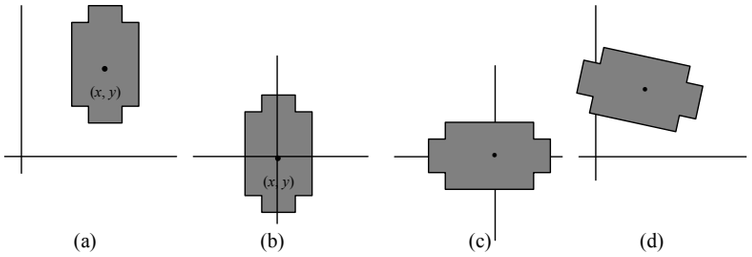


Fig. 5.1 *The Composite Transformation*

Figure 5.1 (a) shows the original position of the object and the rotation point. Figure 5.1 (b) shows the translation of the object such that the rotation point is at the origin. Figure 5.1 (c) shows the rotation of the object. Figure 5.1 (d) shows the translation of the object such that the rotation point is returned to (X_y, Y_y).

A transformation sequence for rotating an object about a specified pivot point using the rotation matrix $R(\varphi)$ of transformation 5.5 is as follows:

$$\begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\varphi & -\sin\varphi & x_r(1-\cos\varphi) + y_r \sin\varphi \\ \sin\varphi & \cos\varphi & y_r(1-\cos\varphi) - x_r \sin\varphi \\ 0 & 0 & 1 \end{bmatrix} \quad ...(5.17)$$

The above equation can also be written as

$$T(x_r, y_r) \;.\; T(-x_r, -y_r) = R(x_r, y_r, \varphi) \quad ...(5.18)$$

where $T(-x_r, -y_r) = T^{-1}$. In general, we can define a rotation function to accept parameters for rotation-point coordinates, as well as the rotation angle (by which the object has to be rotated), and to generate automatically the rotation matrix of equation 5.17.

5.3.3 General Fixed-Point Scaling

Figure 5.2 illustrates a transformation chain to produce scaling with respect to a chosen fixed position (x_t, y_t) using a scaling function that can only scale the object with respect to the coordinate origin:

(i) By translating the object so that the fixed point coincides with the coordinate origin.

(ii) Then by scaling the object with respect to the coordinate origin.

(iii) Finally applying the inverse translation to step (i) to return the object to its original position.

The concatenation of these matrices for these three operations generates the desired scaling matrix as,

$$\begin{bmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & -x_t \\ 0 & 1 & -y_t \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & x_t(1-S_x) \\ 0 & S_y & y_t(1-S_y) \\ 0 & 0 & 1 \end{bmatrix}$$

...(5.19)

This equation can also be written as

$$T(x_t, y_t) \ . \ S(S_x, S_y) \ . \ T(-S_x, -S_y) = S(x_t, y_t, S_x, S_y)$$

...(5.20)

This transformation is automatically generated on graphics systems that produce a scale function accepting coordinates for the fixed point general scaling directions.

The program for scaling is as follows:

```
#include<graphics.h>
#include<stdio.h>
#include<process.h>
#include<conio.h>
void initscreen(float hx[],float hy[])
{   line(0,getmaxy()/2,getmaxx(),getmaxy()/2);
    line(getmaxx()/2,0,getmaxx()/2,getmaxy());
    drawhome(hx,hy);
}
void drawhome(float hx[],float hy[])
{   int i;
    int cx=getmaxx()/2;
    int cy=getmaxy()/2;
    for(i=0;i<4;i++)
       line(cx+hx[i],cy-hy[i],cx+hx[i+1],cy-hy[i+1]);
    line(cx+hx[4],cy-hy[4],cx+hx[0],cy-hy[0]);
}
void main()
{  int graphd=DETECT, graphm,ch,i,px,py,lx1,lx2,ly1,ly2;
    float sx,sy,mx,my,x,y,m,c,thx[5],thy[5];
    float hx[]={40.0,10.0,10.0,70.0,70.0};
    float hy[]={80.0,50.0,10.0,10.0,50.0};
    initgraph(&graphd,& graphm,"c:\\tc");
    initscreen(hx,hy);
    do
```

NOTES

NOTES

```
{  printf("\nEnter the choice from Menu");
   printf("\n1.Scale with respect to 0,0");
   printf("\n2.Scale with respect to arbitrary point");
   printf("\n3.reflect arbitrary point");
   printf("\n4.exit");
   scanf("%d",&ch);
   switch(ch)
   {  case 1: cleardevice();
         initscreen(hx,hy);
         for(i=0;i<=4;i++)
         {  thx[i]=hx[i];
            thy[i]=hy[i];
         }
         printf("\n");
         printf("Enter the scale value of x and y
         axis");
         scanf("%f%f",&sx,&sy);
         for(i=0;i<=4;i++)
         {   thx[i]=thx[i]*sx;
            thy[i]=thy[i]*sy;
         }
         drawhome(thx,thy);
         break;
      case 2: cleardevice();
         initscreen(hx,hy);
         for(i=0;i<=4;i++)
         {  thx[i]=hx[i];
            thy[i]=hy[i];
         }
         printf("\nEnter the reference point");
      scanf("%d%d",&px,&py);
      printf("\n");
      printf("Enter the scale value of x and y axis");
      scanf("%f%f",&sx,&sy);
      for(i=0;i<=4;i++)
      {  thx[i]= thx[i] - px;
         thy[i]= thy[i] - py;
      }
      for(i=0;i<=4;i++)
      {  thx[i]= thx[i]*sx;
```

NOTES

```
      thy[i]= thy[i]*sy;
   }
   for(i=0;i<=4;i++)
   {  thx[i] = thx[i] + px;
      thy[i]= thy[i] + py;
   }
   drawhome(thx,thy);
   break;
case 3: cleardevice();
   initscreen(hx,hy);
   printf("\n");
   printf("Enter the line y = mx + c with m amd c
   value");
   scanf("%f%f",&m,&c);
   if(m>0||m<0)
   {  lx1=0;
      ly1=c;
      lx2=(getmaxy()-c)/m;
      ly2=getmaxy();
   }
   if(m==0)
   {  lx1=0;
      ly1=c;
      lx2=getmaxx();
      ly2=c;
   }
   if(m>=9999)
   {  lx1=lx2=c;
      ly1=0;
      ly2=getmaxy();
   }
   for(i=0;i<=4;i++)
   {  thx[i]= hx[i];
      thy[i]= hy[i];
   }
   for(i=0;i<=4;i++)
   {  x = thx[i];
      y = thy[i];
      if(m==0)
      {  thx[i]= x;
```

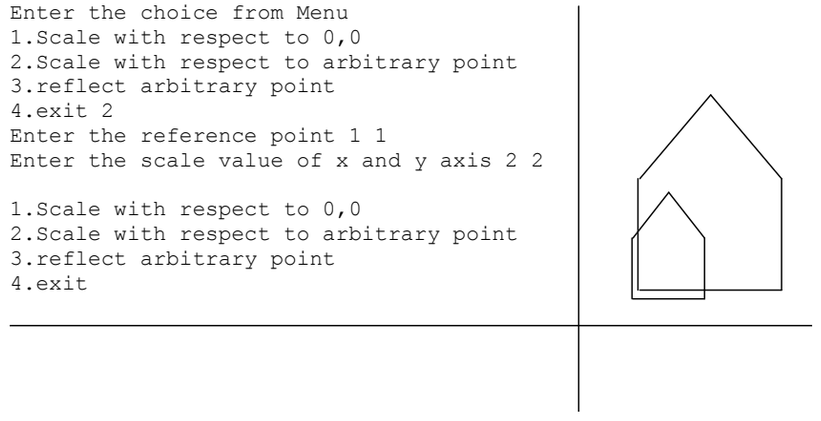
NOTES

```
                    thy[i]= c+(c-y);
                }
                if(m>0||m<0)
                {  thx[i]=2*((m*m*y+x)-c*m*m)/((1+m*m)*m)-
x;
                    thy[i]=2*(c+m*m*y+x)/(1+m*m)-y;
                }
                if(m>=9999)
                {  thy[i]=y;
                    thx[i]=c+(c-x);
                }
            }
                drawhome(thx,thy);
    line(getmaxx()/2+lx1,getmaxy()/2-ly1,getmaxx()/
2+lx2,getmaxy()/2-ly2);
                break;
                case 4: exit(1);
            }
        }while(ch!=4);
}
```

The output of this program is as follows:

```
Enter the choice from Menu
1.Scale with respect to 0,0
2.Scale with respect to arbitrary point
3.reflect arbitrary point
4.exit 2
Enter the reference point 1 1
Enter the scale value of x and y axis 2 2

1.Scale with respect to 0,0
2.Scale with respect to arbitrary point
3.reflect arbitrary point
4.exit
```



5.3.4 Transformations for Scaling

Parameters S_x and S_y scale objects along the x and y directions respectively. We can scale an object in other directions by rotating the object to align the desired scaling directions with the coordinate axes before applying the scaling transformation. Suppose we want to apply scaling factors with values specified by parameters S_1 and S_2 in the directions shown in figure 5.3.

NOTES

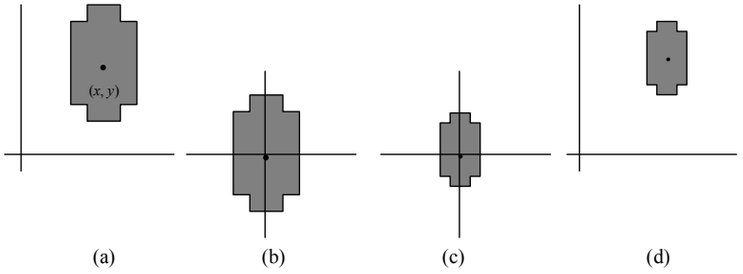


Fig. 5.2 *A Transformation Sequence for scaling an Object with Respect to a Specified Fixed Position using the Scaling Matrix $S(S_x, S_y)$ of Transformation 5.7.*

Figure 5.2(a) shows the original position of the object. Figure 5.2 (b) shows the translation of the object such that the centre of object is origin. Figure 5.2 (c) shows the scaling of the object 1 with respect to the origin. Figure 5.2 (d) shows the translation of the object so that the fixed point is returned to the position.

To accomplish scaling with out changing the orientation of the object, we first carry out a rotation so that the directions for S_1 and S_2 coincide with the x and y-axes, respectively. Then the scaling transformation is applied, followed by an opposite rotation to return the points to their original orientations. The composite matrix resulting from the product of these three transformations is as follows:

$$R^{-1}(\Phi).S(S_1,S_2).R(\Phi) = \begin{bmatrix} S_1\cos^2\Phi + S_2\sin^2\Phi & (S_2 - S_1)\cos\Phi\sin\Phi & 0 \\ (S_2 - S_1)\cos\Phi\sin\Phi & S_1\sin^2\Phi + S_2\cos^2\Phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

...(5.21)

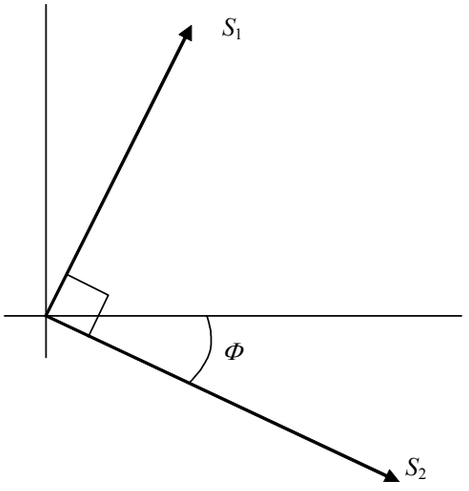


Fig. 5.3 *Scaling Parameter S_1 and S_2 applied in Orthogonal Directions defined by the Angular Displacement Φ*

NOTES

As an example of this scaling transformation, we turn a unit square into a parallelogram (see Figure 5.4) by shifting it along the diagonal from (0, 0) to (1, 1). We rotate the diagonal onto the y-axis and double its length with the transformation parameters $\Phi = 45°$, $S_1 = 1$, and $S_2 = 2$. In equation 5.21, we assumed that scaling was to be performed relative to the origin. We could take this scaling operation one step further and concatenate the matrix with translation operators, so that the composite matrix would include parameters for the specification of a scaling fixed position.

5.3.5 Concatenation Properties

The major property of matrix multiplication is associativeness. For any three matrices, A, B, and C, the matrix product $A.B.C$ can be performed by first multiplying A and B or by B and C and finally multiplying the remaining term with the result of first multiplication:

$$A.B.C = (A.B).C = A.(B.C) \qquad ...(5.22)$$

Therefore, matrix products can be evaluated using either a left-to-right or a right-to-left order. Generally, the products of transformation may not be commutative, that means the matrix product $A.B$ may not be equal to $B.A$.

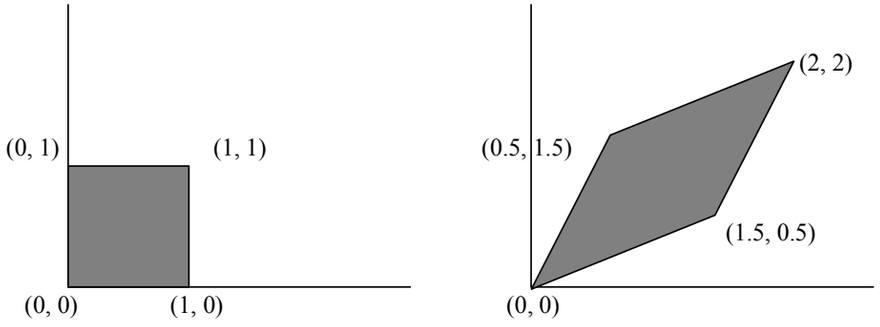


Fig. 5.4 *A Square is Transformed into a Parallelogram using the Composite Transformation Matrix*

This means that if we have to rotate and translate an object, we must be vigilant about the order in which the composite matrix is evaluated (Figure 5.5). For some special cases, like a chain of transformations (all of the same type), multiplication of transformation matrices is commutative in nature. As an example, two successive rotations can be performed in any order and the final position would be the same. Also, this commutative property is applicable for two successive translations or two successive scalings. Another commutative couple of operations is rotation and uniform scaling, that means $S_x = S_y$.

5.3.6 General Composite Transformations and Computational Efficiency

A general two-dimensional transformation, representing a combination of translations, rotations, and scaling, can be expressed as

NOTES

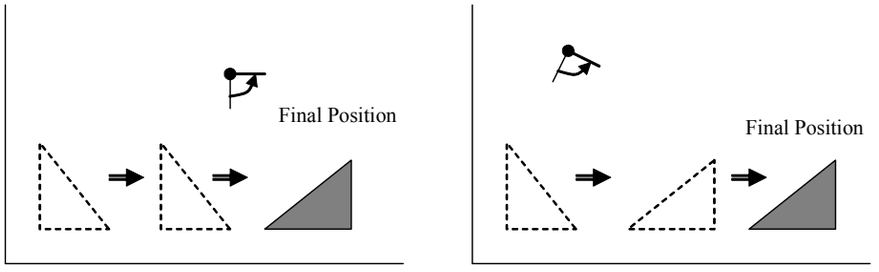
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} rs_{xx} & rs_{xy} & trs_x \\ rs_{yx} & rs_{yy} & trs_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \ldots(5.23)$$

The four attributes rs_{xx}, $rs_{xy,}$ rs_{yx}, and rs_{yy} are the multiplicative rotation-scaling terms in the transformation which are involved in only rotation angles and scaling factors. The elements trs_x and trs_y are the translational terms containing a combination of pivot-point, translation distances, fixed-point coordinates, scaling parameters and rotation angles. For example, if an object has to be scaled and rotated about its centroid coordinates represented by , and then translated, the values for the attributes of the composite transformation matrix are as follows:

$$T(t_x, t_y).R(x_c, y_c, \varphi).S(x_c, y_c, s_x, s_y) =$$

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

...(5.24)



(a) Translation followed by rotation (b) Rotation followed by translation

Fig. 5.5 *Reversing the Order Transformations in which the Sequence of Transformations performed may affect the Final Transformed Position of the Object*

Although matrix equation 5.23 requires six additions and nine multiplications, the open calculations for the transformed coordinates are given by equation 5.25.

$$x' = x.rS_{xx} + y.rS_{xy} + trS_x, \; y' = x.rS_{yx} + y.rS_{yy} + trS_y \quad \ldots(5.25)$$

Thus we only need to perform four additions and four multiplications to transform the coordinate positions. This shows the maximum number of calculations required for any transformations chain, once the individual matrix is concatenated and the elements of the composite matrix are evaluated. Without performing concatenation, individual transformations can be applied one at a time and the number of computations can be drastically increased. Therefore, the efficient completion of the transformation operations is to formulate transformation matrices, concatenation of any transformation sequence, and computation of transformed coordinates using equation 5.25. The direct matrix multiplications with the composite transformation

NOTES

matrix of equation 5.23 can be equally efficient on parallel systems. A general rigid-body transformation matrix, containing only rotations and translation, can easily be represented by the following matrix:

$$\begin{bmatrix} r_{xx} & r_{xy} & tr_x \\ r_{yx} & r_{yy} & tr_y \\ 0 & 0 & 1 \end{bmatrix} \quad \text{....(5.26)}$$

where the four elements r_{xx}, r_{xy}, r_{yx}, and r_{yy} are the multiplicative rotation terms, and elements tr_x and tr_y are the translational terms. The change in rigid-body coordinate position is also sometimes referred to as a rigid-motion transformation. All angles and distances between coordinate positions are unchanged by the transformation. In addition, matrix 5.40 has the property that its upper-left 2×2 sub-matrix is an orthogonal matrix. This means that if we consider each row of the sub-matrix as a vector, then the two vectors (r_{xx}, r_{xy}) and (r_{yx}, r_{yy}) form an orthogonal set of unit vectors (having unit length),

$$r_{xx}^2 + r_{xy}^2 = r_{yx}^2 + r_{yy}^2 = 1 \quad \text{...(5.24)}$$

and the vectors are perpendicular (their dot product is 0 because of $\cos 90° = 0$),

$$r_{xx} . r_{xy} + r_{yx} . r_{yy} = 0 \quad \text{...(5.28)}$$

Therefore, if these unit vectors are transformed by the rotation sub-matrix, (r_{xx}, r_{xy}) is converted to a unit vector along the x-axis and (r_{yx}, r_{yy}) is transformed into a unit vector along the y-axis of the coordinate system. This is given as follows:

$$\begin{bmatrix} r_{xx} & r_{xy} & 0 \\ r_{yx} & r_{yy} & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} r_{xx} \\ r_{xy} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \quad \text{...(5.29)}$$

$$\begin{bmatrix} r_{xx} & r_{xy} & 0 \\ r_{yx} & r_{yy} & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} r_{yx} \\ r_{yy} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \quad \text{...(5.30)}$$

For example, the following rigid-body transformation first rotates an object through an angle φ about a pivot point and then translates:

$$T(t_x, t_y).R(x_r, y_r, \varphi) = \begin{bmatrix} \cos\varphi & -\sin\varphi & x_r(1-\cos\varphi) + y_r \sin\varphi + t_x \\ \sin\varphi & \cos\varphi & y_r(1-\cos\varphi) - x_r \sin\varphi + t_y \\ 0 & 0 & 1 \end{bmatrix} \quad \text{....(5.31)}$$

Here, orthogonal unit vectors in the upper-left 2-by-2 sub-matrix are $(\cos\varphi, -\sin\varphi)$ and $(\sin\varphi, \cos\varphi)$, and

$$\begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\varphi \\ -\sin\varphi \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \quad ...(5.32)$$

NOTES

Similarly, unit vector $(\sin\varphi, \cos\varphi)$ is converted by the transformation matrix in equation 5.46 to the unit vector (0, l) in that direction. The orthogonal property of rotation matrices is useful for constructing a rotation matrix when we know the final orientation of an object rather than the amount of angular rotation necessary to put the object into that position. Directions for the desired orientation of an object can be determined by the alignment of certain objects in a scene or by selected positions in the scene. Figure 5.6 shows an object that is to be aligned with the unit direction vectors $\overline{A}$ and $\overline{B}$. Assuming that the original object orientation, as shown in Figure 5.6(a), is aligned with the coordinate axes, we construct the desired transformation by assigning the elements of $\overline{A}$ to the first row of the rotation matrix and the elements of $\overline{B}$ to the second row. This can be a convenient method to obtain the transformation matrix for rotation within a local coordinate system when we know the final orientation vectors.

Since rotation calculations require trigonometric evaluations and several multiplications for each transformed point, computational efficiency can become an important consideration in rotation transformation. In animations and other applications that involve many repeated transformations and small rotation angles, we can use approximations and iterative calculations to reduce computations in the composite transformation equations. When the rotation angle is small, trigonometric functions can be replaced with approximation values based on the first few terms of their power-series expansions. For small enough angles (less than 5°), $\cos\varphi$ is approximately 1 and sin*j* has a value very close to the value of φ in radians. If we are rotating in small angular steps about the origin, for instance, we can set $\cos\varphi$ to 1 and reduce transformation calculations at each step to two multiplications and two additions for each set of coordinates to be rotated as follows:

$$x_{new} = x - y.\sin\varphi, \quad \text{and } y_{new} = x.\sin\varphi + y \quad ...(5.33)$$

where $\sin\varphi$ is evaluated once for all steps, assuming the rotation angle does not change. The error introduced by this approximation at each step decreases as the rotation angle decreases. But even with small rotation angles, the accumulated error over many steps can become quite large. We can control the accumulated error by estimating the error in x_{new} and y_{new} at each step and resetting object positions when the error accumulation becomes too great. Composite transformations often involve inverse matrix calculations. Reflections and shears are transformation sequences for general scaling directions. As we have noted, inverse matrix representations for basic geometric

NOTES

transformations can be generated with simple procedures. An inverse translation matrix is obtained by changing the signs of the translation distances, and an invert rotation matrix is obtained by performing a matrix transpose (or changing the sign of the sine terms).

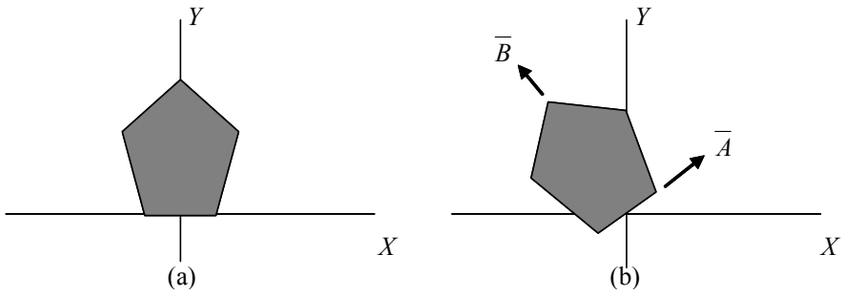


Fig. 5.6 *The Rotation Matrix for Rotating an Object from Original Position (a) to Position (b)*

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

These operations are much more simple than direct inverse matrix calculations. An implementation of composite transformations is given in the following procedure. Matrix *M* can be initialized to the identity matrix. As each individual transformation is specified, it is concatenated with the total transformation matrix *M*. When all transformations have been specified, this composite transformation is applied to a given object. For this example, a polygon is scaled and rotated about a given reference point. Then the object is translated. Figure 5.7 shows the original and final positions of the polygon transformed by this sequence.

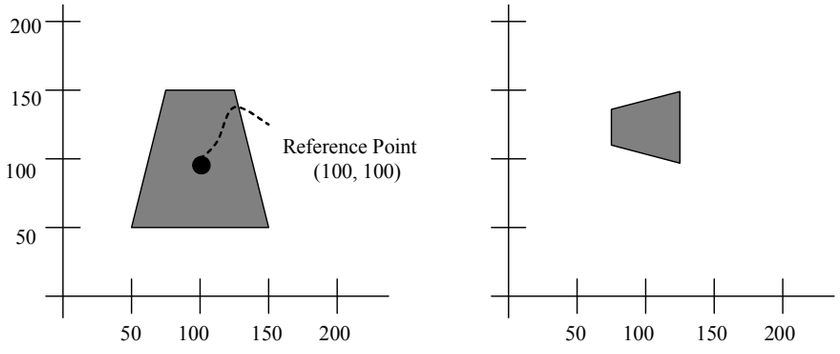


Fig. 5.7 *A Polygon (a) is Transformed into (b) by the Composite Operations by a C Program Given below*

5.3.7 Transformation of Coordinate System

So far we have discussed ***geometric transformation*** of 2D objects which are well defined with respect to a ***global coordinate system*****,** also called the ***world***

coordinate system(WCS) –the principal frame of reference. But it is often found convenient to define quantities (independent of the WCS) with respect to a ***local coordinate system***, also called the ***model coordinate system*** or the ***user coordinate system(UCS).*** While the UCS may vary from entity to entity and as per convenience, the WCS being the master reference system remains fixed for a given display system. Once you define an object "locally" you can place it in the global system simply by specifying the location of the origin and orientation of the axes of the local system within the global system, then mathematically transforming the point coordinates defining the object from local to global system. Such transformations are known as the transformation between the coordinate systems. Here we will briefly discuss only the transformations between two cartesian frames of reference.

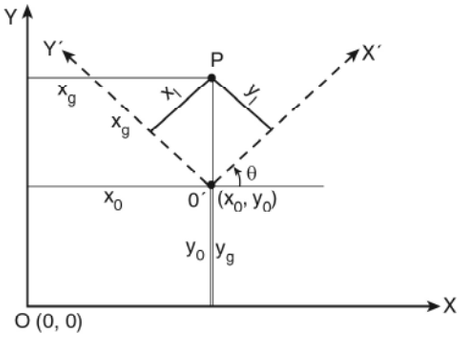


Fig. 5.8 *The coordinates of point P w.r.t local cood.sys X´O´Y´ is, (x_l, y_l) while its coordinates w.r.t. WCS XOY is (x_g, y_g)*

Local to global: Fig. 5.8 shows two cartesian systems global XOY and local X´OY´ with coordinate origins at (0,0) and (x_0, y_0) respectively and with an of angle θ between the X and X´ axes. To transform object descriptions (x_l, y_l) w.r.t. local system to that (x_g, y_g) w.r.t. global system we need to follow the following two steps that superimpose the XOY frame to the X´O´Y´ frame.

Step 1

Translation: So that origin '*O*' of the XY system moves to origin '*O*' sof the X´Y´ system.

Transformation matrix:

$$[T_T]_{x_0,\, y_0} = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

Step 2

Rotation: So that X aligns with X´ and Y aligns with Y´

NOTES

Transformation matrix:

$$[T_T]_\theta = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

So the global coordinates (x_g, y_g) of point P are expressed in terms of its known local coordinate (x_l, y_l) as,

$$\begin{pmatrix} x_g \\ y_g \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_l \\ y_l \\ 1 \end{pmatrix}$$

Global to Local: Unlike the previous case, if the object is originally defined in the global XY system then to transform its description in local X′Y′ system (Fig. 5.8). we have to superimpose local X′O′Y′ frame to the global XOY frame. So the transformations required this time are,

1. $[T_T]_{-x_0, -y_0} = \begin{pmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{pmatrix}$ and

2. $[T_R]_{-\theta} = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$

The description of point P in the local system is given by

$$\begin{pmatrix} x_l \\ y_l \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_g \\ y_g \\ 1 \end{pmatrix}$$

Now will it make any difference if we first rotate and then translate the objects? (Refer problem 15)

Note:

- **Shear** A different category of transformation that distorts the shape of an object such that the transformed shape appears as if the object were composed of internal layers that had been caused to slide over each other.

  When an X direction shear relative to the X axis is produced, the original coordinate position (x, y) is then shifted by an amount proportional to its distance 'y' from the X axis ($y = 0$), i.e.

  $x' = x + sh_x\, y$ where sh_x = shear parameter in the positive X direction.
  $y' = y$

the corresponding transformation matrix is,

$$\begin{pmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

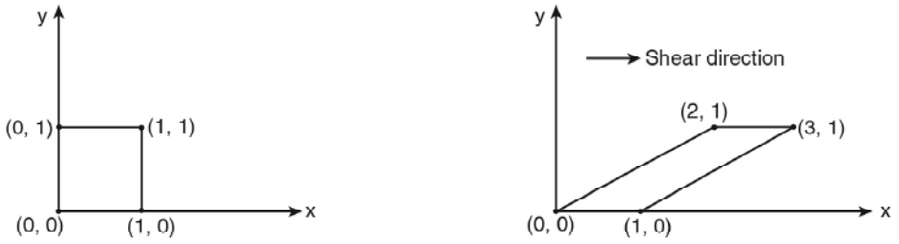


Fig. 5.9 *A unit square is transformed to a parallelogram following a X direction shear with* $sh_x = 2$

Based on the same principle,-shear in the X direction relative to a line parallel to the X axis, ($y = k$) –shear in the Y direction relative to Y axis or parallel to Y axis ($x = h$) can be formulated.

- **Affine Transformation** All the two dimensional transformations where each of the transformed coordinates x' and y' is a linear function of the original coordinates x & y as

$$\left.\begin{aligned} x' &= A_1 x + B_1 y + C_1 \\ y' &= A_2 x + B_2 y + C_2 \end{aligned}\right\}$$ where A_i, B_i, C_i are parameters fixed for a given transformation type.

2D transformation of coordinate systems, translation, rotation, scaling, reflection, shear-all are examples of 2D affine transformations and exhibit the general property, that parallel lines transform to parallel lines & finite points map to finite points. An affine transformation involving only translation, rotation & reflection preserves the length and angle between two lines.

Check Your Progress

1. What is concatenation of matrices?
2. What is the major property of matrix multiplication?
3. How is an inverse translation matrix and an invert rotation matrix obtained?

5.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. The formation of products of transformation matrices is often referred to as a composition or concatenation of matrices.

NOTES

NOTES

2. The major property of matrix multiplication is associativeness.
3. An inverse translation matrix is obtained by changing the signs of the translation distances, and an invert rotation matrix is obtained by performing a matrix transpose (or changing the sign of the sine terms).

5.5 SUMMARY

- An animation may require an object to be translated and rotated at each increment of the motion along a linear path.
- The matrix representations of the previous section, a set-up for a matrix can be made for any series of transformations. This set-up is determined by the product of matrices of the individual transformations.
- The concatenation of transformation matrices for two consecutive scaling operations produces the composite scaling matrix.
- Translation of the object such that the rotation -point position is shifted to the coordinate origin.
- By translating the object so that the fixed point coincides with the coordinate origin.
- we have discussed geometric transformation of 2D objects which are well defined with respect to a global coordinate system, also called the world coordinate system(WCS).
- The formation of products of transformation matrices is often referred to as a composition or concatenation of matrices.

5.6 KEY WORDS

- **Shear transformation:** it is the transformation that slants the shape of an object.
- **Affine transformation:** It is a linear mapping method that preserves points, straight lines, and planes.

5.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What is consecutive scaling?
2. Discuss the concatenation properties for matrix multiplication.
3. Discuss the transformation of coordinate system.

Long Answer Questions

1. Explain the fixed point scaling.
2. Explain the transformation for scaling.
3. What is affine transformation? Explain.

NOTES

5.8 FURTHER READINGS

Hearn, Donal and M. Pauline Baker. 1990. *Computer Graphics*. New Delhi: Prentice-Hall of India.

Rogers, David F. 1985. *Procedural elements for Computer Graphics*. New York: McGraw-Hill.

Foley, D. James, Andries Van Dam, Steven K. Feiner and John F. Hughes. 1997. *Computer Graphics Principles and Practice*. Boston: Addison Wesley.

Mukhopadhyay, A. and A. Chattopadhyay. 2007. *Introduction to Computer Graphics and Multimedia*. New Delhi: Vikas Publishing House.

UNIT 6 2D VIEWING

NOTES

Structure

6.0 INTRODUCTION

In this unit, you will learn about the viewing pipeline and viewing functions. Viewing is the process of drawing a view of a model on a 2-dimensional display. They are used to map from one space to another along the graphics pipeline.

6.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand viewing coordinate reference frame
- Discuss window to viewport coordinate transformation
- Understand 2D viewing functions

6.2 THE VIEWING PIPELINE

The generation of a view of an object on a display device of any computer requires stage wise transformation operations of the object definitions in different coordinate systems.

Different views of an object are possible on the view plane either by moving the object and keeping the eyepoint fixed or by moving the eyepoint and keeping the object fixed. However, the later technique is used in most of the computer graphics application to create a new view of the object.

The two-dimensional Device Coordinate System (DCS) or Screen Coordinate System (SCS) for display monitor locates points on the display/output of a particular output device, such as graphics monitor or plotter. These coordinates are integers in terms of pixels, addressable points, inches, cms, etc.

NOTES

The Normalized Device Coordinates (NDC) are used to map world coordinates in a device independent two-dimensional pseudospace within the range 0 to 1 for each of x and y before final conversion to specific device coordinates.

The modelling and world coordinate positions can be any floating point values in which normalized coordinates (x_{nc}, y_{nc}) satisfy the inequalities: $0 \le x_{nc} \le 1,\ 0 \le y_{nc} \le 1;$ and the device coordinates are integers within the range (0, 0) to (x_{max}, y_{max}), with (x_{max}, y_{max}) depending on the resolution of a particular output device.

You will frequently see the terms object space and image space. Object space corresponds to the world coordinate system and image space to the display screen coordinate system. Object space is an unbounded and infinite collection of continuous points. Image space is a finite 2-D space. In 2-D, we simply specify a window in the object space and a viewport in the display surface. Conceptually, 2-D objects in the object space are clipped against the window and are then transformed to the normalized viewport and finally to the viewport for display using standard window to viewport mapping.

6.2.1 Viewing Coordinate Reference Frame

Before object descriptions, view plane is projected to transfer the viewing coordinates (refer Figure 6.1). Conversion of object description from world to viewing coordinates is equivalent to a transformation which superimposes the viewing reference frame into the world frame using basic geometric translate and rotate operations. Following is the transformation sequence:

- Translate the view reference point to the origin of the world coordinate system.
- Apply rotations to align the x, y and z axes with the world x_u, y_w and z_v axes, respectively.

If the view reference point is specified at world position (x_0, y_0 and z_0) then this point is translated to the world origin with the matrix transformation.

$$T = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The rotation sequence requires up to three coordinate axis rotations depending on the direction which is chosen for **N**. In fact, **N** is not aligned with world coordinate axis. The viewing and world systems are superimposed with transformation sequence $\mathbf{R}_z.\mathbf{R}_y.\mathbf{R}_x$. That is, we first rotate around the world x_w axis to bring z_v into the $x_w z_w$ plane. Then, the world is rotated around y_w axis to align the z_w and z_v axes. The final rotation is about z_w axis to align the y_w and y_v axes. The composite transformation matrix is then applied to world coordinate descriptions to transfer

NOTES

them to viewing coordinates. Then given vectors **N** and **V** are calculated as a set of unit vectors to define the viewing coordinate system which is obtained by the following equations:

$$\mathbf{n} = \frac{\mathbf{N}}{|\mathbf{N}|} = (n_x, n_y, n_z)$$

$$\mathbf{u} = \frac{\mathbf{V} \times \mathbf{n}}{|\mathbf{V}|} = (u_x, u_y, u_z)$$

$$\mathbf{v} = \mathbf{n} \times \mathbf{u} = (v_x, v_y, v_z)$$

The complete world to viewing coordinate transformation matrix is obtained as the matrix product in the following way:

$$\mathbf{M}_{wc.vc} = \mathbf{R.T}$$

This transformation is then applied to coordinate descriptions of objects in which **u** is transformed into the world x_w axis, **v** is transformed onto the y_w axis and n is transformed onto z_w axis. The complete world to viewing coordinate transformation matrix is obtained as the matrix product,

$$\mathbf{M}_{wc.vc} = \mathbf{R.T}$$

This transformation is then applied to coordinate descriptions of objects scene to transfer them with reference to the viewing reference frame.

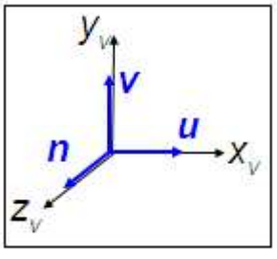


Fig. 6.1 *Viewing Coordinate Reference Frame*

6.2.2 Window to Viewport Coordinate Transformation

Normalization transformation (**N**) maps world coordinates (or viewing coordinates) to normalized device coordinates and workstation transformation (**W**) maps normalized device coordinates to physical device coordinates. In general, the mapping of a part of a world coordinate scene to device coordinates is referred as viewing transformation (**V**), mathematically expressed as follows:

V = W.N

Sometimes, the two-dimensional viewing transformation is simply called window to viewport transformation.

By defining a closed boundary or window, the enclosed portion of a world coordinate scene is clipped against the window boundary and the data of the

clipped portion is extracted for mapping to a separately defined region known as *viewport*. While window selects a part of the scene, viewport displays the selected part at desired location on the display area. When the window is changed we see a different part of the scene at the same portion (viewport) on the display. If we change the viewport only, we see the same part of the scene drawn at different scale or at different place on the display. By successively increasing or decreasing the size of the window around a part of the scene, the viewport remaining fixed, we can get the effect of zoom out or zoom in respectively on the displayed part.

NOTES

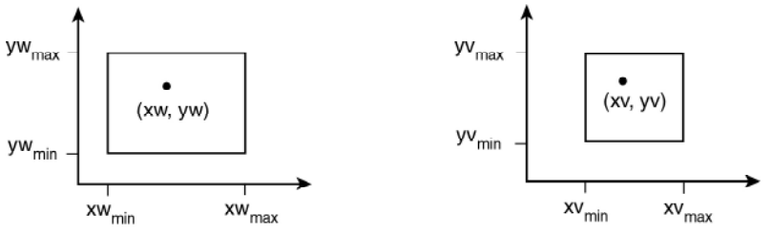
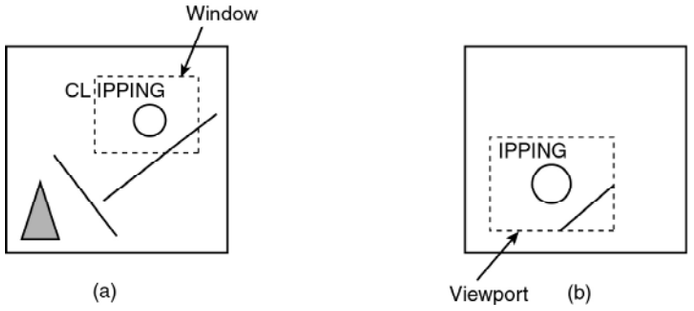
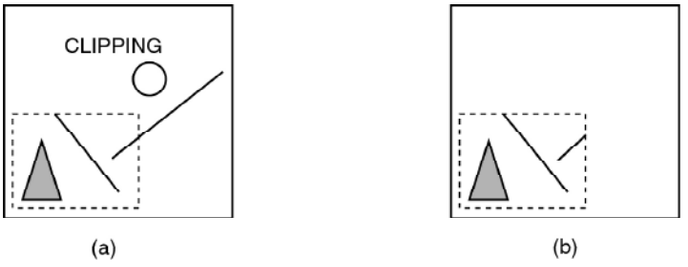


Fig. 6.2 *Object Placement in Viewport*

Window or viewport can be general polygon shaped or circular. For simplicity here, we will consider rectangular window and rectangular viewport, with edges of the rectangles being parallel to the coordinate axes (refer Figure 6.2).

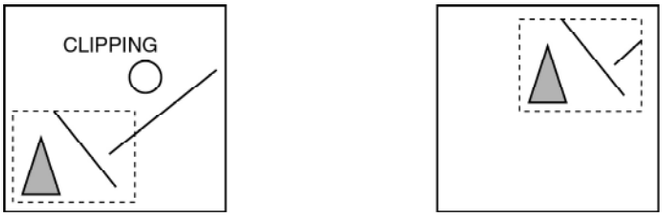


Out of several objects drawn in the object space only those clipped by the window are displayed in the viewport in image space. Note that the viewport objects are larger than the window objects though the object shapes are not distorted because the viewport is a uniformly scaled version of the window as shown in the above screen.



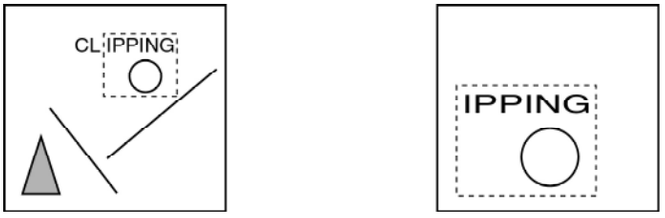
NOTES

When the window is changed, different objects of the scene are displayed in the same viewport as shown in the above screen.



(a) (b)

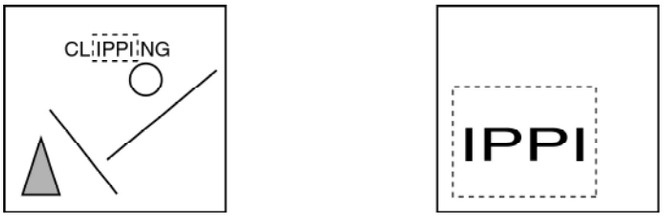
When only the viewport is moved and the window remains same, we see same objects displayed through the viewport at a different position as shown in the above screen. All the screens (a) and (b) above represent screen before effect and after effect, respectively.



(a) (b)

Let us consider a point (*xw*, *yw*) within the window enclosure as shown in Figure 6.2(a). The window is mapped to the viewport such that (*xw*, *yw*) transforms to (*xv*, *yv*) in the device space as shown in Figure 6.2(b).

Screen below shows the final text clipping area 'IPPI' from the given text 'CLIPPING'.



(a) (b)

The viewport remaining fixed, the window size is gradually reduced maintaining the scale factors same; the zooming effect is obtained through the viewports.

To maintain the same relative placement of the point in the viewport as in the window,

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

And also,

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

This implies, $$xv = xv_{\min} + (xw - xw_{\min}) \left(\frac{xv_{\max} - xv_{\min}}{xw_{\max} - xw_{\min}} \right)$$

$$yv = yv_{\min} + (yw - yw_{\min}) \left(\frac{yv_{\max} - yv_{\min}}{yw_{\max} - yw_{\min}} \right)$$

NOTES

If $\left(\frac{xv_{\max} - xv_{\min}}{xw_{\max} - xw_{\min}} \right)$ be termed as s_x (the x scale factor for scaling the window to the size of the viewport) and $\left(\frac{yv_{\max} - yv_{\min}}{yw_{\max} - yw_{\min}} \right)$ be termed as s_y (the y scale factor for window to viewport scaling) then,

and,
$$xv = xv_{\min} + (xw - xw_{\min}) s_x$$
$$yv = yv_{\min} + (yw - yw_{\min}) s_y$$

In terms of two step geometric transformation the above relation can be interpreted as,

Step 1: Scaling the window area to the size of the viewport with scale factors s_x and s_y with reference to a fixed point $(xw_{\min}, yw_{\min})$.

$$\Rightarrow [T_1] = \begin{pmatrix} s_x & 0 & xw_{\min}(1 - s_x) \\ 0 & s_y & yw_{\min}(1 - s_y) \\ 0 & 0 & 1 \end{pmatrix}$$

Step 2: Translating the scaled window to the position of the viewport so that,

$$\Delta x = xv_{\min} - xw_{\min} \quad \text{and}$$
$$\Delta y = yv_{\min} - yw_{\min}$$
$$\Rightarrow [T_2] = \begin{pmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{pmatrix}$$

Concatenating $[T_1]$ and $[T_2]$ we get,

$$[T] = [T_1]\,[T_2] = \begin{pmatrix} s_x & 0 & \Delta x + xw_{\min}(1 - s_x) \\ 0 & s_y & \Delta y + yw_{\min}(1 - s_y) \\ 0 & 0 & 1 \end{pmatrix}$$

Replacing the values of Δx and Δy in the above transformation matrix we finally get,

$$[T] = \begin{pmatrix} s_x & 0 & -s_x xw_{\min} + xv_{\min} \\ 0 & s_y & -s_y yw_{\min} + yv_{\min} \\ 0 & 0 & 1 \end{pmatrix} \qquad ...(6.1)$$

NOTES

The aspect ratio of a rectangular window or viewport is defined by,

$$a = \frac{x_{\max} - x_{\min}}{y_{\max} - y_{\min}}$$

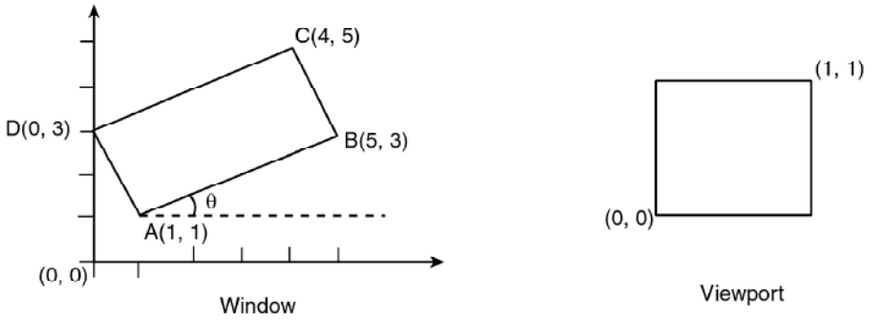
$$\text{If} \quad s_x = s_y \quad \text{then} \quad \frac{xv_{\max} - xv_{\min}}{xw_{\max} - xw_{\min}} = \frac{yv_{\max} - yv_{\min}}{yw_{\max} - yw_{\min}}$$

$$\Rightarrow \frac{xv_{\max} - xv_{\min}}{yv_{\max} - yv_{\min}} = \frac{xw_{\max} - xw_{\min}}{yw_{\max} - yw_{\min}} \Rightarrow a_v = a_{\hat{w}}$$

So, it can be said that if the aspect ratio a_v of the viewport equals the aspect ratio a_w of the window then $s_x = s_y$ and no distortion of displayed scene occurs other than uniform magnification or compression. If $a_v \neq a_w$ then the displayed scene in the viewport gets somewhat distorted with reference to the scene captured by the window.

Example 6.1: Find the normalization transformation **N** which uses the rectangle A (1, 1), B (5, 3), C (4, 5) and D (0, 3) as a window and the normalized device screen as the viewport.

Solution:



Here we see that the window edges are not parallel to the coordinate axes. So we will first rotate the window about A so that it is aligned with the axes.

$$\text{Now,} \quad \tan\theta = \frac{3-1}{5-1} = \frac{1}{2}$$

$$\Rightarrow \sin\theta = \frac{1}{\sqrt{5}}, \cos\theta = \frac{2}{\sqrt{5}}$$

Here we are rotating the rectangle in clockwise direction. So θ is (–)ve, i.e., $-\theta$.

The rotation matrix about $A(1, 1)$ is,

$$[T_{R,\theta}]_A = \begin{pmatrix} \frac{2}{\sqrt{5}} & \frac{1}{\sqrt{5}} & \left(\frac{1-3}{\sqrt{5}}\right) \\ \frac{-1}{\sqrt{5}} & \frac{2}{\sqrt{5}} & \left(\frac{1-1}{\sqrt{5}}\right) \\ 0 & 0 & 1 \end{pmatrix}$$

The x extent of the rotated window is the length of

$\overline{AB}$ which is $\sqrt{(4^2 + 2^2)} = 2\sqrt{5}$.

Similarly, the y extent is the length of $\overline{AD}$ which is $\sqrt{(1^2 + 2^2)} = \sqrt{5}$.

NOTES

For scaling the rotated window to the normalized viewport we calculate s_x and s_y as,

$$s_x = \frac{\text{viewport } x \text{ extent}}{\text{window } x \text{ extent}} = \frac{1}{2\sqrt{5}}$$

$$s_y = \frac{\text{viewport } y \text{ extent}}{\text{window } y \text{ extent}} = \frac{1}{\sqrt{5}}$$

As in Equation (6.1), the general form of transformation matrix representing mapping of a window to a viewport is,

$$[T] = \begin{pmatrix} s_x & 0 & -s_x xw_{\min} + xv_{\min} \\ 0 & s_y & -s_y yw_{\min} + yv_{\min} \\ 0 & 0 & 1 \end{pmatrix}$$

In this problem $[T]$ may be termed as **N** as this is a case of normalization transformation with,

$$xw_{\min} = 1 \qquad xv_{\min} = 0$$
$$yw_{\min} = 1 \qquad yv_{\min} = 0$$
$$s_x = \frac{1}{2\sqrt{5}} \qquad s_y = \frac{1}{\sqrt{5}}$$

By substituting the above values in $[T]$, i.e., **N**,

$$N = \begin{pmatrix} \frac{1}{2\sqrt{5}} & 0 & \left(\frac{-1}{2}\right)\frac{1}{\sqrt{5}} + 0 \\ 0 & \frac{1}{\sqrt{5}} & \left(\frac{-1}{\sqrt{5}}\right)1 + 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Now we compose the rotation and transformation N to find the required viewing transformation $\mathbf{N}_R$,

$$\mathbf{N}_R = \mathbf{N}\,[T_{R,\theta}]_A = \begin{pmatrix} \frac{1}{2\sqrt{5}} & 0 & \frac{-1}{2\sqrt{5}} \\ 0 & \frac{1}{\sqrt{5}} & \frac{-1}{\sqrt{5}} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{2}{\sqrt{5}} & \frac{1}{5} & 1 - \frac{3}{\sqrt{5}} \\ \frac{-1}{\sqrt{5}} & \frac{2}{\sqrt{5}} & 1 - \frac{1}{\sqrt{5}} \\ 0 & 0 & 1 \end{pmatrix}$$

NOTES

6.3 2-D VIEWING FUNCTIONS

Two-dimensional viewing functions include world coordinates to viewing coordinates in which window to viewport process is used. Window is a region of the scene selected for viewing which is also called *clipping window*. Viewport is a region on display device for mapping to window.

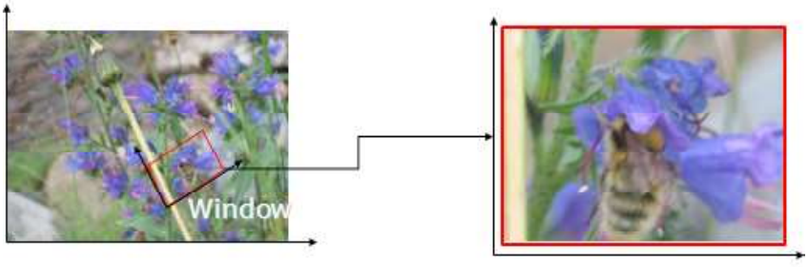


Fig. 6.3 *World Coordinates and Viewing Coordinates*

Figure 6.3 displays the world coordinates and viewing coordinates area in which the clipping window selects what we want to see in our virtual 2-D world. The viewport indicates where it is to be viewed on the output device or within the display window.

Check Your Progress

1. What are the possible views of an object on the view plane?
2. What is viewport?

6.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Different views of an object are possible on the view plane either by moving the object and keeping the eyepoint fixed or by moving the eyepoint and keeping the object fixed.
2. By defining a closed boundary or window, the enclosed portion of a world coordinate scene is clipped against the window boundary and the data of the clipped portion is extracted for mapping to a separately defined region known as viewport.

6.5 SUMMARY

- The two-dimensional Device Coordinate System (DCS) or Screen Coordinate System (SCS) for display monitor locates points on the display/ output of a particular output device, such as graphics monitor or plotter.

- The Normalized Device Coordinates (NDC) are used to map world coordinates in a device independent two-dimensional pseudo space within the range 0 to 1 for each of x and y before final conversion to specific device coordinates.
- Normalization transformation (N) maps world coordinates to normalized device coordinates and workstation transformation (W) maps normalized device coordinates to physical device coordinates.
- Two-dimensional viewing functions include world coordinates to viewing coordinates in which window to viewport process is used. Window is a region of the scene selected for viewing which is also called clipping window.
- The enclosed portion of a world coordinate scene is clipped against the window boundary and the data of the clipped portion is extracted for mapping to a separately defined region known as viewport.

6.6 KEY WORDS

- **Object Space:** An unbounded and infinite collection of continuous points.
- **Viewing Pipeline:** it is a series of transformations, which are passed by geometry data to end up as image data being displayed on a device.
- **Viewport:** it is a polygon viewing region in computer graphics.

6.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. State the difference between DCS and SCS.
2. What is a coordinate reference frame?
3. What is the transformation sequence for viewing coordinate reference frame?
4. Discuss the mathematical expression for viewing transformation.
5. What is viewing function?

Long Answer Questions

1. Explain the window to viewport coordinate transformation.
2. Explain the aspect ratio of a rectangular viewport.
3. What are 2-D viewing functions? Explain with the help of suitable examples.

6.8 FURTHER READINGS

NOTES

Hearn, Donal and M. Pauline Baker. 1990. *Computer Graphics*. New Delhi: Prentice-Hall of India.

Rogers, David F. 1985. *Procedural elements for Computer Graphics*. New York: McGraw-Hill.

Foley, D. James, Andries Van Dam, Steven K. Feiner and John F. Hughes. 1997. *Computer Graphics Principles and Practice*. Boston: Addison Wesley.

Mukhopadhyay, A. and A. Chattopadhyay. 2007. *Introduction to Computer Graphics and Multimedia*. New Delhi: Vikas Publishing House.

UNIT 7 CLIPPING ALGORITHMS

NOTES

Structure

7.0 INTRODUCTION

Clipping refers to the removal of part of a scene. The primary use of clipping in computer graphics is to identify that portion of a picture, object, line or line segment that are outside the viewing region. The viewing transformation is insensitive to the position of points relative to the viewing volume—especially those points behind the viewer—and it is necessary to remove these points before generating the view. We will consider point clipping, line clipping, text clipping and polygon clipping. There are several clipping algorithms. You will learn about Cohen-Sutherland line clipping algorithm, midpoint subdivision algorithm, Cyrus-Beck algorithm and Sutherland-Hodgeman polygon clipping operation.

7.1 OBJECTIVES

After going through this unit, you will be able to:

- Define line clipping
- Understand the explicit line clipping and Sutherland-Cohen line clipping algorithm
- Explain the Sutherland- Hodgeman polygon clipping algorithm

7.2 LINE CLIPPING

The algorithm for clipping is different for different shape of clipping window and for different type of objects (line, curve, etc.) Here we will discuss some of the standard clipping algorithms for rectangular clipping window.

NOTES

Line Clipping — Visibility Test

Fig. 7.1 shows a 2D scene and a rectangular clipping window. It is defined by left (L), right (R), top (T) and bottom (B) edges parallel to the edges or axes of the display surface.

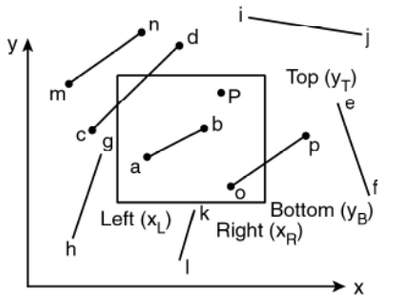


Fig. 7.1 *Two Dimensional Clipping Window*

Now any point (x, y) is interior to the clipping window provided that

$$x_L \le x \le x_R \text{ and } y_B \le y \le y_T$$

where x_L, x_R, y_T, y_B are the x and y coordinates respectively of the left, right, top and bottom edges of the window.

The equal sign indicates that points on the window boundary are included within the window.

To determine visibility of lines w.r.t. the clipping window, we have to deal with three types of line.

(i) lines totally visible

(ii) lines totally invisible

(iii) lines partially visible

Lines are interior to the clipping window and hence totally visible if both end points are interior to the window e.g., line *ab* in Fig. 7.1. However lines are not totally invisible if both end points are exterior to the window e.g., line *cd* in Fig. 7.1. If both end points of a line are to the right of the window (e.g., line *ef*) or to the left (e.g., line *gh*), or above (e.g., line *ij*) or below (e.g., line *kl*) the window the line is completely exterior to the window and hence totally invisible. Lines which fail these two tests are either partially visible (e.g., lines *cd* and *op*) or totally invisible (e.g., line *mn*).

Explicit Line Clipping Algorithm

This technique uses a 4-digit code to indicate which of the nine regions around a clipping window contain the end point of a line. The scheme which assigns this 4 bit code to the line end points is summarised as follows :

We consider x_L & x_R to be the x coordinates of the left and right edges of the window and y_T & y_B to be the y coordinates of the top and bottom edges of the

window respectively. *x,y* are the coordinates of any end point of the test line. Here the rightmost bit is bit 1.

(i) if $x < x_L$ then set bit 1 to 1 else 0
(ii) if $x > x_R$ then set bit 2 to 1 else 0
(iii) if $y < y_B$ then set bit 3 to 1 else 0
(iv) if $y > y_T$ then set bit 4 to 1 else 0

Fig. 7.2 shows the nine region codes w.r.t. a clipping window. You may note that the codes for the diagonal regions of the window are obtained by adding the codes for the two adjacent regions.

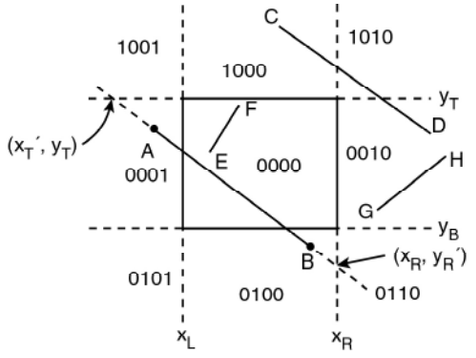


Fig. 7.2

Now the algorithm is as follows,

Input : $x_L, x_R, y_T, y_B, P_1 (x_1, y_1), P_2 (x_2, y_2)$: P_1, P_2 are the endpoints of the test line

Initialise $i = 1$

while $i <= 2$

if $x_i < x_L$ then bit 1 of code $- P_i = 1$ else 0
if $x_i > x_R$ then bit 2 of code $- P_i = 1$ else 0 : The endpoint codes of the line are set

if $y_i < y_B$ then bit 3 of code $- P_i = 1$ else 0
if $y_i > y_T$ then bit 4 of code $- P_i = 1$ else 0
$i = i + 1$

end while

if code $- P_1$ = code $- P_2$ = 0 then (the line is totally visible) draw P_1 P_2 (ref. line *EF* in Fig. 7.2).

if code $- P_1$ AND code $- P_2$ <> 0 then (the line is totally invisible[2]) ignore P_1 P_2 (ref. line *GH* in Fig. 7.2).

NOTES

If both the end points codes are not equal to zero and their logical intersection is also not non-zero (i.e. zero) then the line is checked for partial visibility or total invisibility (ref. line *AB* and *CD* in Fig. 7.2). The intersection of the line $P_1 P_2$ with window edges are found.

For a line having end points at (x_1, y_1) and (x_2, y_2)

$$y_2 = m\ (x_2 - x_1) + y_1$$

where $m = \left(\frac{y_2 - y_1}{x_2 - x_1}\right) = \text{slope}$

The intersection with the window edges are given by:

Left : $x_L,\ y'_L = m(x_L - x_1) + y_1; m \neq \alpha$

Right : $x_R,\ y'_R = m(x_R - x_1) + y_1;\ m \neq \alpha$

Top : $y_T, x'_T = x_1 + \left(\frac{1}{m}\right)(y_T - y_1);\ m \neq 0$

Bottom : $y_B, x'_B = x_1 + \left(\frac{1}{m}\right)(y_T - y_1);\ m \neq 0$

when $m = \alpha$ i.e. the line is vertical and $x_2 - x_1 = 0$

assign very large numbers to x_L, x_R, y'_L, y'_R

when $m = 0$ i.e. the line is horizontal and $y_2 - y_1 = 0$

assign very large numbers to $x'_T,\ x'_B,\ y_T,\ y_B$

To get the two intersection points $(x'_1,\ y'_1)$ and $(x'_2,\ y'_2)$

Set $i = 1$

```
if i <=2 then
   if y'L < yT and y'L > yB then
      x'i = xL, y'i = y'L
      i = i+1
      end if
end if
```

```
if   i <=2 then
     if y'R  < yT and y'R > yB then
     x'i = xR,   y'i = y'R
     i = i+1
     end if
end if
if   i <=2 then
     if   x'B  <  xR and x'B >  xL then
     x'i = x'B,   y'i = yB
     i = i+1
     end if
end if
```

NOTES

if $i <= 2$ then
  if $x'_T < x_R$ and $x'_T > x_L$ then
  $x'_i = x'_T$, $y'_i = y_T$
  $i = i + 1$
  end if
end if

For partly visible lines intersecting two window edges in the visible region (like line AB in Fig. 7.2), any two of these 'if's work here and we get two (x'_i, y'_i) for $i = 1, 2$ which are the end points of the visible portion of the line.

if $i < 2$ then (none of the intersections are found on the visible window edges) ignore P_1P_2 (totally invisible) (ref. line *CD* in Fig. 7.2).

else draw the line between (x'_1, y'_1) and (x'_2, y'_2) where codes of both the end points are non zero and their logical ANDing is zero.

end if

Now we consider the case where one of the end points of the line lies inside the window and the other outside it.

If code – $P_1 = 0$ then (P_1 is inside the window) $P = P_1$ and $P_{out} = P_2$.

else if code – $P_2 = 0$ (P_2 is inside the window) then $P = P_2$ and $P_{out} = P_1$ (Fig. 7.3 (a))

end if

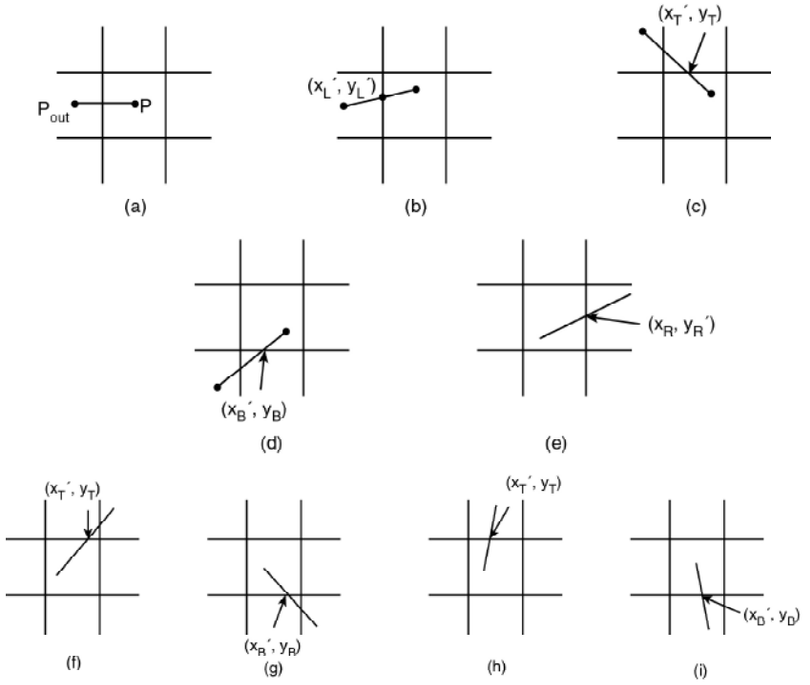


Fig. 7.3

NOTES

if $(P_{out}) \cdot x < x_L$ then

if $y'_L <= y_T$ and $y'_L >= y_B$ then $P_i = (x_L, y'_L)$ (Fig. 7.3 (b))

else if $y'_L > y_T$ then $P_i = (x'_T, y_T)$ (Fig. 7.3 (c))

else (i.e. if $y'_L < y_B$) $P_i = (x'_B, y_B)$ (Fig. 7.3(d))

end if

else if $(P_{out}).x > x_R$ then

if $y'_R <= y_T$ and $y'_R >= y_B$ then $P_i = (x_R, y'_R)$ (Fig. 7.3(e))

else if $y'_R > y_T$ then $P_i = (x'_T, y_T)$ (Fig. 7.3(f))

else (i.e. if $y'_R < y_B$) $P_i = (x'_B, y_B)$ (Fig. 7.3(g))

end if

else if $(P_{out}) \cdot y > y_T$ then $P_i = (x'_T, y_T)$ (Fig. 7.3(h))

else (i.e. if $(P_{out}) \cdot y < y_B$) $P_i = (x'_B, y_B)$ (Fig. 7.3(i))

end if

Draw visible portion PP_i

Example 7.1 Given a window *A* (20, 20), *B* (60, 20), *C* (60, 40), *D* (20, 40) use any clipping algorithm to find the visible portion of the line *P* (30, 50) to *Q* (70, 30) inside the window.

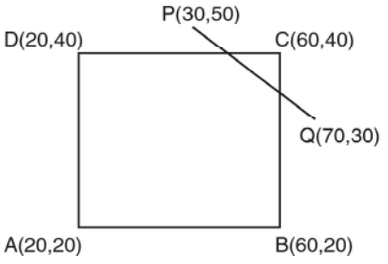


Fig. 7.4

For the line *PQ* the slope is

$$m = \left(\frac{y_2 - y_1}{x_2 - x_1}\right) = \left(\frac{50-30}{30-70}\right) = -\left(\frac{20}{40}\right) = -\frac{1}{2}$$

and the intersections with the window edges are

left : $x = 20, y = m\,(x_L - x_1) + y_1$

or, $y = -1/2\,(20 - 30) + 50 = -\,1/2\,(-10) + 50 = 5 + 50 = 55$

which is greater than y_T and is rejected

right : $x = 60, y = m\,(x_R - x_1) + y_1$

or, $y = -\,1/2\,(60 - 30) + 50 = -15 + 50 = 35\ (< y_T, > y_B)$

$\therefore$ The intersection with the right edge is at point (60, 35).

top : $y = 40, x = x_i + 1/m\ (y_T - y_1)$

or, $x = 30 - 2\ (40 - 50) = 30 + 20 = 50\ (< x_R, > X_L)$

$\therefore$ The intersection with the top edge is at point (50, 40)

bottom : $y = 20, x = x_1 + 1/m\ (y_B - y_1)$

or, $x = 30 - 2\ (20 - 50)$

$= 30 + 60 = 90$

which is greater than x_R and thus rejected.

So the visible part of the line PQ is from P (50,40) to Q (60,35).

7.2.1 Sutherland–Cohen Algorithm

This is one of the most popular line clipping algorithm. The concept of assigning 4-bit region codes to the endpoints of a line and subsequent checking and AND operation of the endpoint codes to determine totally visible lines and totally invisible lines (lying completely at one side of the clip window externally) was originally introduced by Dan Cohen and Ivan Sutherland in this algorithm. For clipping other totally invisible lines and partially visible lines, the algorithm breaks the line segments into smaller subsegments by finding intersection with appropriate window edges. For a pair of a non-zero endpoint and an intersection point, the corresponding subsegment is checked for two primary visibility state as done in the earlier steps. The process is repeated till two visible intersections are found or no intersection with any of the four visible window edge is found. Thus this algorithm cleverly reduces the number of intersection calculation unlike the previous algorithm. The steps of the algorithm are as follows.

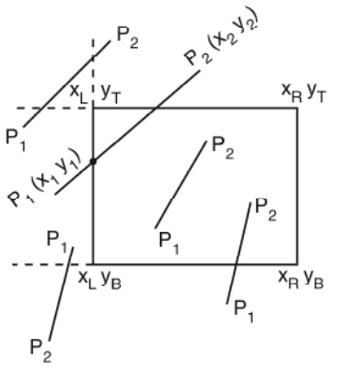


Fig. 7.5

1. Input : $x_L, x_R, y_T, y_B, P_1\ (x_1, y_1), P_2\ (x_2, y_2)$

   Initialise $i = 1$

   while $i <= 2$

NOTES

if $x_i < x_L$ then bit 1 of code $- P_i = 1$ else 0
if $x_i > x_R$ then bit 2 of code $- P_i = 1$ else 0 : The endpoint codes of the line are
if $y_i < y_B$ then bit 3 of code $- P_i = 1$ else 0
if $y_i > y_T$ then bit 4 of code $- P_i = 1$ else 0

$i = i + 1$

end while

$i = 1$

2. Initialise $j = 1$

while $j <= 2$

if $x_j < x_L$ then $C_{j\ \text{left}} = 1$ else $C_{j\ \text{left}} = 0$
if $x_j > x_R$ then $C_{j\ \text{right}} = 1$ else $C_{j\ \text{right}} = 0$: Set flags according to the position of the
if $y_j < y_B$ then $C_{j\ \text{bottom}} = 1$ else $C_{j\ \text{bottom}} = 0$ line endpoints w.r.t. window edges
if $y_j > y_T$ then $C_{j\ \text{top}} = 1$ else $C_{j\ \text{top}} = 0$

end while

3. if codes of P_1 and P_2 are both equal to zero then draw $P_1 P_2$ (totally visible)

4. if logical intersection or AND operation of code $- P_1$ and code $- P_2$ is not equal to zero then ignore $P_1 P_2$ (totally invisible)

5. if code $- P_1 = 0$ then swap P_1 and P_2 along with their flags and set $i = 1$

6. if code $- P_1 <> 0$ then

for $i = 1$,

{if $C_{1\ \text{left}} = 1$ then

find intersection (x_L, y'_L) with left edge vide eqn. (2)

assign code to (x_L, y'_L)

$P_1 = (x_L,\ y'_L)$
end if
$i = i + 1;$

go to 3

}

for $i = 2$,

{if $C_{1\ \text{right}} = 1$ then

find intersection (x_R, y'_R) with right edge vide eqn. (3)

assign code to (x_R, y'_R)

$P_1 = (x_R, y'_R)$

end if

$i = i + 1$

go to 3

}

for $i = 3$

{if $C_{1\ \text{bottom}} = 1$ then

find intersection (x'_B, y_B) with bottom edge vide eqn. (5)

assign code to (x'_B, y_B)

$P_1 = (x'_B, y_B)$

end if

$i = i + 1$

go to 3

}

for $i = 4$

{if $C_{1\ \text{top}} = 1$ then

find intersection (x'_T, y_T) vide eqn (4) with top edge

assign code to (x'_T, y_T)

$P_1 = (x'_T, y_T)$

end if

$i = i + 1$

go to 3

}

end

Example 7.2 A Clipping window ABCD is located as follows:

A (100, 10), B (160, 10), C (160, 40), D (100, 40). Using Sutherland-Cohen clipping algorithm find the visible portion of the line segments EF, GH and $P_1 P_2$. E (50,0), F (70, 80), G (120, 20), H (140, 80), P_1 (120, 5), P_2 (180, 30)

NOTES

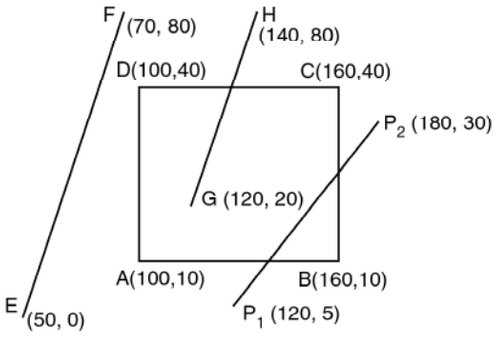


Fig. 7.6

At first considering the line $P_1 P_2$

INPUT: $P_1(120, 5)$, $P_2(180, 30)$
$x_L = 100$, $x_R = 160$, $y_B = 10$, $y_T = 40$

$x_1 > x_L$ then bit 1 of code–$P_1 = 0$ $C_{1\ \text{left}} = 0$
$x_1 < x_R$ then bit 2 of code–$P_1 = 0$ $C_{1\ \text{right}} = 0$
$y_1 < y_B$ then bit 3 of code–$P_1 = 1$ $C_{1\ \text{bottom}} = 1$
$y_1 < y_T$ then bit 4 of code–$P_1 = 0$ $C_{1\ \text{top}} = 0$

code–$P_1 = 0100$,

$x_2 > x_L$ then bit 1 of code–$P_2 = 0$ $C_{2\ \text{left}} = 0$
$x_2 > x_R$ then bit 2 of code–$P_2 = 1$ $C_{2\ \text{right}} = 1$
$y_2 > y_B$ then bit 3 of code–$P_2 = 0$ $C_{2\ \text{bottom}} = 0$
$y_2 < y_T$ then bit 4 of code–$P_2 = 0$ $C_{2\ \text{top}} = 0$

code–$P_2 = 0010$.

code–$P_1 <> 0$ and code–$P_2 <> 0$
then $P_1 P_2$ not totally visible

code–P_1 AND code–$P_2 = 0$
hence the line is not totally invisible

As code–$P_1 <> 0$

for $i = 1$
{
$C_{1\ \text{left}} (= 0) <> 1$ then nothing is done.
$i = i + 1 = 2$
}

code–P_1 < > 0 and code–P_2 < > 0
then P_1P_2 not totally visible.

code–P_1 AND code–P_2 = 0
hence the line is not totally invisible

for $i = 2$
{

$C_{1\ right}$ (=0) < > 1 then nothing is done.

$i = i + 1 = 2 + 1 = 3$
}
code–P_1 < > 0 code–P_2 < > 0
then P_1P_2 not totally visible

code–P_1 AND code–P_2 = 0
hence the line is not totally invisible

for $i = 3$
{

$C_{1\ bottom} = 1$ then find intersection of $P_1\ P_2$ with bottom edge
$y_B = 10$
$x_B = (180 - 120)(10 - 5)/(30 - 5) + 120$
$= 132$
then $P_1 = (132, 10)$

$x_1 > x_L$	then bit 1 of code–P_1 =0	$C_{1\ left} = 0$
$x_1 < x_R$	then bit 2 of code–P_1 =0	$C_{1\ right} = 0$
$y_1 = y_B$	then bit 3 of code–P_1 =0	$C_{1\ bottom} = 0$
$y_1 < y_T$	then bit 4 of code–P_1 =0	$C_{1\ top} = 0$

code–P_1 = 0000
$i = i + 1 = 3 + 1 = 4$
}

code–P_1 = 0 but code–P_2 < > 0
then P_1P_2 not totally visible

code–P_1 AND code–P_2 = 0
hence the line is not totally invisible

As code–P_1 =0
swap P_1 and P_2 along with the respective flags

NOTES

$P_1 = (180,30)$
$P_2 = (132,10)$
code–$P_1 = 0010$
code–$P_2 = 0000$

$C_{1\ left} = 0$		$C_{2\ left} = 0$	
$C_{1\ right}$	$= 1$	$C_{2\ right}$	$= 0$
$C_{1\ bottom}$	$= 0$	$C_{2\ bottom}$	$= 0$
$C_{1\ top} = 0$		$C_{2\ top} = 0$	

Reset $i = 1$
for $i = 1$
{
$C_{1\ left} (= 0) <> 1$ then nothing is done
$i = i + 1 = 1 + 1 = 2$
}
code–$P_1 <> 0$, and code–$P_2 <> 0$
then P_1P_2 not totally visible.

code–P_1 AND code–$P_2 = 0$
hence the line is not totally invisible

for $i = 2$
{
$C_{1\ right} = 1$ then find intersection of $P_1 P_2$ with right edge
$x_R = 160$
$y_R = (30 – 5)(160 – 120)/(180 –120) + 5$
$= 21.6667$
$= 22$
then $P_1 = (160, 22)$

$x_1 > x_L$	then bit 1 of code–$P_1 = 0$	$C_{1\ left} = 0$
$x_1 = x_R$	then bit 2 of code–$P_1 = 0$	$C_{1\ right} = 0$
$y_1 > y_B$	then bit 3 of code–$P_1 = 0$	$C_{1\ bottom} = 0$
$y_1 < y_T$	then bit 4 of code–$P_1 = 0$	$C_{1\ top} = 0$

code–$P_1 = 0000$, $i = i + 1 = 2 + 1 = 3$
}
As both code–$P_1 = 0$ and code–$P_2 = 0$
then the line segment P_1P_2 is totally visible
So the visible portion of input line P_1P_2 is $P_1'P_2'$ where $P_1 = (160,22)$ & $P_2 = (132,10)$.

Considering the line EF

1. The endpoint codes are assigned

$\text{code} - E \rightarrow 0101$
$\text{code} - F \rightarrow 1001$

NOTES

2. Flags are assigned for the two endpoints

   $E_{\text{left}} = 1$ (as x coordinate of E is less than x_L)

   $E_{\text{right}} = 0,\ E_{\text{top}} = 0,\ E_{\text{bottom}} = 1$

   Similarly,

   $F_{\text{left}} = 1,\ F_{\text{right}} = 0,\ F_{\text{top}} = 1,\ F_{\text{bottom}} = 0$

3. Since codes of E and F are both not equal to zero the line is not totally visible
4. Logical intersection of codes of E and F is not equal to zero. So we may ignore EF line and declare it as totally invisible

Considering the line GH

1. The endpoint codes are assigned

   $\text{code} - G \rightarrow 0000$

   $\text{code} - H \rightarrow 1000$

2. Flags are assigned for the two endpoints

   $G_{\text{left}} = 0,\ G_{\text{right}} = 0,\ G_{\text{top}} = 0,\ G_{\text{bottom}} = 0$

   Similarly

   $H_{\text{left}} = 0,\ H_{\text{right}} = 0,\ H_{\text{top}} = 1,\ H_{\text{bottom}} = 0$

3. Since codes of G and H are both not equal to zero so the line is not totally visible
4. Logical intersection of codes of G and H is equal to zero so we cannot declare it as totally invisible
5. Since $\text{code} - G = 0$, Swap G and H along with their flags and set $i = 1$

   implying $G_{\text{left}} = 0,\ G_{\text{right}} = 0,\ G_{\text{top}} = 1,\ G_{\text{bottom}} = 0$

   $H_{\text{left}} = 0,\ H_{\text{right}} = 0,\ H_{\text{top}} = 0,\ H_{\text{bottom}} = 0$

   as $G \rightarrow 1000,\ H \rightarrow 0000$

6. Since $\text{code} - G <> 0$ then

   for $i = 1$, {since $G_{\text{left}} = 0$

   $i = i + 1 = 2$

   go to 3

   }

   The conditions 3 and 4 do not hold and so we cannot declare line GH as totally visible or invisible

   for $i = 2$, {since $G_{\text{right}} = 0$

   $i = i + 1 = 3$

NOTES

go to 3

}

The conditions 3 and 4 do not hold and so we cannot declare line *GH* as totally visible or invisible

for $i = 3$, {since $G_{bottom} = 0$

$i = i + 1 = 4$

go to 3

}

The conditions 3 and 4 do not hold and so we cannot declare line *GH* as totally visible or invisible

for $i = 4$, {since $G_{top} = 1$

Intersection with top edge, say $P(x, y)$ is found as follows

Any line passing through the points G, H and a point $P(x, y)$ is given by

$y - 20 = \{(80 - 20) / (140 - 120)\} (x - 120)$

or, $y - 20 = 3x - 360$

or, $y - 3x = -340$

Since the y coordinate of every point on line *CD* is 40, so we put $y = 40$ for the point of intersection $P(x, y)$ of line *GH* with edge *CD*

$40 - 3x = -340$

or, $-3x = -380$

or, $x = 380/3 = 126.66 \approx 127$

So the point of intersection is $P(127, 40)$

We assign code to it.

Since the point lies on edge of the rectangle so the code assigned to it is 0000.

Now we assign $G = (127, 40)$; $i = 4 + 1 = 5$.

conditions 3 and 4 are again checked.}

Since codes *G* and *H* are both equal to 0, so the line between *H* (120, 20) and *G* (127, 40) is totally visible.

Midpoint Sub-Division Algorithm

Partially visible and totally invisible lines which cannot be identified by checking and operating (ANDing) endpoint codes are subdivided into two equal segments by finding the midpoint. Each half is then separately considered and tested with endpoint codes for immediate identification of totally visible and totally invisible state. Segments which cannot be identified even then are further subdivided at midpoint and each subdivision is subsequently tested. This bisection and testing procedure is continued until the intersection with a window edge is found with some specified accuracy. The sequential steps of the algorithm are given below.

NOTES

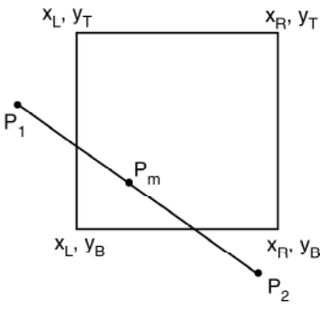


Fig. 7.7

Input : $P_1\ (x_1, y_1)\ P_2, (x_2, y_2), x_L, x_R, y_B, y_T$

code assign (P_1), code assign (P_2)

(Assume the function code assign ( ) assigns 4 bit code to a point with respect to the rectangular clipping window.)

1. if code–P_1, code–P_2 both = 0, then (the line is totally visible) draw P_1P_2
2. if code–P_1 AND code–P_2 <> 0 then (line is totally invisible) ignore $P_1\ P_2$
3. $P_m = \frac{(P_1 + P_2)}{2}$; code assign (P_m)
4. if code–P_m <> 0
   then if code–P_1 AND code–P_m < > 0 then
   $P_1 = P_m$; go to 1
   else if code–P_2 AND code–P_m < > 0 then
   $P_2 = P_m$; go to 1
   end if
   end if
5. if code–P_m = 0 then
   if code–P_1, code–P_m both = 0 then consider $P_2\ P_m$
   else if code–P_2, code–P_m both = 0 then consider $P_1\ P_m$
   end if
   end if
6. Considering $P_1\ P_m$

   do $\{P_{m1} = \frac{(P_1 + P_m)}{2}$; code assign (P_{m1})

   if code–P_{m1} < > 0 then $P_1 = P_{m1}$ else $P_m = P_{m1}$

   } while

   $(P_m \cdot x < > x_L$ and $P_m \cdot x < > x_R$ and $P_m \cdot y < > y_T$ and $P_m \cdot y < > y_B)$

   $P_1 = P_m$

NOTES

7. Considering $P_2 P_m$

do $\{P_{m2} = \frac{(P_2 + P_m)}{2}$; code assign (P_{m2})

if code–$P_{m2} <> 0$ then $P_2 = P_{m2}$ else $P_m = P_{m2}$

} while

$(P_m \cdot x <> x_L$ and $P_m \cdot x <> x_R$ and $P_m \cdot y <> y_B$ and $P_m \cdot y <> y_T)$

$P_2 = P_m$

8. Draw $P_1 P_2$

In the above algorithm a calculated midpoint P_m is considered to be lying on any of the boundary lines of the window say boundary line $x = x_L$ if $P_m \cdot x = x_L \pm$ tolerance, where 'tolerance' is a very small number (say 0.1) prescribed depending on the precision of the display.

The midpoint subdivision algorithm, if implemented in hardware works very fast (even faster than Sutherland-Cohen algorithm) because hardware addition and division by 2 are very fast.

Example 7.3 Using midpoint subdivision algorithm find the visible portion of the line $P_1 P_2$, P_1 (120, 5) P_2 (180, 30) w.r.t. a clipping window *ABCD* where *A* (100, 10), *B* (160, 10), *C* (160, 40), *D* (100, 40).

Refer Fig. 7.6

INPUT: $P_1(120, 5)$, $P_2(180, 30)$

$x_L = 100$, $x_R = 160$, $y_B = 10$, $y_T = 40$

$x_1 > x_L$ then bit 1 of code–$P_1 = 0$
$x_1 < x_R$ then bit 2 of code–$P_1 = 0$
$y_1 < y_B$ then bit 3 of code–$P_1 = 1$
$y_1 < y_T$ then bit 4 of code–$P_1 = 0$

code–$P_1 = 0100$,

$x_2 > x_L$ then bit 1 of code–$P_1 = 0$
$x_2 > x_R$ then bit 2 of code–$P_1 = 1$
$y_2 > y_B$ then bit 3 of code–$P_1 = 0$
$y_2 < y_T$ then bit 4 of code–$P_1 = 0$

code–$P_2 = 0010$.

code–$P_1 <> 0$ and code– $P_2 <> 0$
then $P_1 P_2$ not totally visible.
code–P_1 AND code–$P_2 = 0000$
hence (code–P_1 AND code–$P_2 = 0$)
then line is not totally invisible

$P_m = (P_1 + P_2)/2 = ((120+180)/2, (5+30)/2) = (150, 17.5)$
$x_m > x_L$ then bit 1 of code–$P_m = 0$
$x_m < x_R$ then bit 2 of code–$P_m = 0$

$y_m > y_B$ then bit 3 of code–P_m =0
$y_m < y_T$ then bit 4 of code–P_m =0

P_1	code–P_1	P_2	code–P_2	P_m	code–P_m	Comment
120,5	0100	180,30	0010	150,18	0000	Save P_2P_m, continue with P_1P_m
120,5	0100	150,18	0000	135,12	0000	Continue P_1P_m
120,5	0100	135,12	0000	128,9	0100	Continue P_mP_2
128,9	0100	135,11	0000	132,10	0000	Succeeds

code–P_m = 0000, i.e. code–P_m = 0
(Calculating the point by rounding it)
So one of the intersection points has been found at (132,10)

For the second intersection point:

P_1	code–P_1	P_2	code–P_2	P_m	code–P_m	Comment
120,5	0100	180,30	0010	150,18	0000	Continue with P_mP_2
150,18	0000	180,30	0010	165,24	0010	Continue with P_1P_m
150,18	0000	165,24	0010	158,21	0000	Continue with P_mP_2
158,21	0000	165,24	0010	162,23	0010	Continue with P_1P_m
158,21	0000	162,23	0010	160,22	0000	Succeeds

So the second point of intersection has been found at (160, 22)

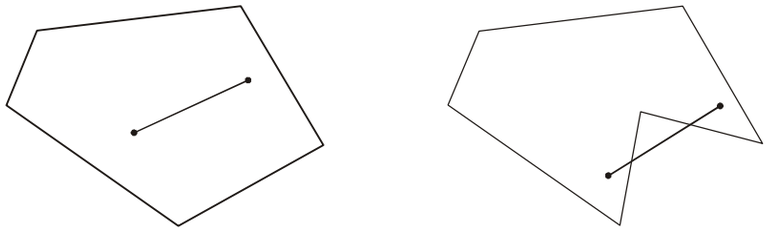
Hence visible portion of line P_1P_2 is $P_1'P_2'$ where P_1 = (132,10) and P_2 = (160, 22).

7.2.2 Clipping Lines against any Convex Polygonal Clipping Window – Cyrus-Beck Algorithm

Before describing the algorithm we need to be acquainted with some relevant concepts first.

- **What does a *Convex Polygon* mean?**

It is a polygon for which the line joining any two interior points lie completely inside the polygon.



(a) *Convex Polygon* ***(b)*** *Non-convex Polygon*

Fig 7.8

- How by using normal vector do we know whether a point on a line lies at the inner side or outer side or on an edge of a convex clipping window?

NOTES

NOTES

Let $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$ be position vectors of three points on $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$ respectively a line that intersects an edge E_i of a convex clipping window R. Let $\mathbf{N_i}$ be the ***inward normal*** vector (pointing to the interior of the clipping window) to the edge E_i:

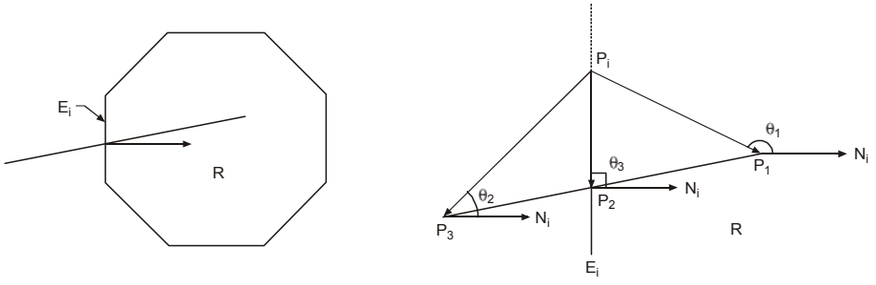


Fig. 7.9

Considering $\mathbf{P_i}$ as position vector of a point on edge E_i, three different relation characterizes position of the three points w.r.t E_i and **R**.

For point $\mathbf{P_1}$ which is at the inner side of E_i $\quad \mathbf{N_i} \cdot (\mathbf{P_1} - \mathbf{P_i}) < 0$

as angle between $\mathbf{N_i}$ and $\mathbf{P_1} - \mathbf{P_i}$ say $\theta_1 > 90°$

For point $\mathbf{P_2}$ which is at the outer side of E_i $\quad \mathbf{N_i} \cdot (\mathbf{P_2} - \mathbf{P_i}) > 0$

as angle between $\mathbf{N_i}$ and $\mathbf{P_2} - \mathbf{P_i}$ say $\theta_2 < 90°$

For point $\mathbf{P_3}$ which is on E_i $\quad \mathbf{N_i} \cdot (\mathbf{P_3} - \mathbf{P_i}) = 0$

as angle between $\mathbf{N_i}$ and $\mathbf{P_3} - \mathbf{P_i}$ say $\theta_3 = 90°$

So it can be inferred that depending on whether $\mathbf{N_i} \cdot (\mathbf{P} - \mathbf{P_i})$ is (–)ve, (+)ve, (+)ve or equal to zero the point **P** lies at the inner side (window side), outer side of or on the window-edge that intersects the line containing **P**.

- **How to represent any straight line parametrically?**

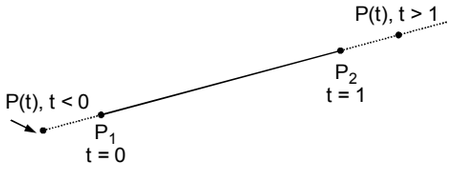
As you have seen in unit 2, using the concept of parallel vectors a line with end point position vectors $\mathbf{P_1}$ and $\mathbf{P_2}$ can be expressed parametrically as,

$\mathbf{P} = \mathbf{P_1} + [\mathbf{P_2} - \mathbf{P_1}]\, t$, t being the parameter and $0 < t < 1$. Here **P**, any point (position vector) on the line, is nothing but a function of t, say, $\mathbf{P}(t)$.

$\mathbf{P}(t = 0)$ implies $\mathbf{P} = \mathbf{P_1}$ and $\mathbf{P}(t = 1)$ implies $\mathbf{P} = \mathbf{P_2}$. $t < 0$ implies points on the line before $\mathbf{P_1}$ whereas $t > 0$ implies points on the line beyond $\mathbf{P_2}$ i.e., in neither case, points lie within the segment $\mathbf{P_1P_2}$.

From the above parametric expression we get,

$$t = \frac{\mathbf{P}(t) - \mathbf{P_1}}{\mathbf{P_2} - \mathbf{P_1}} = \frac{P - P_1}{P_2 - P_1} = \frac{y - y_1}{y_2 - y_1} = \frac{x - x_1}{x_2 - x_1}$$



where $\boldsymbol{P} = [x\ y]$ $\quad \boldsymbol{P_1} = [x_1\ y_1]$; $\boldsymbol{P_2} = [x_2\ y_2]$

Fig. 7.10

The Basic Scheme of Cyrus-Beck Algorithm

For a point $\mathbf{P}(t)$ on the line $\mathbf{P_1P_2}$ to be the point of intersection with an edge E_i (having inward normal vector $\mathbf{N_i}$ and any point $\mathbf{P_i}$ on it) of a convex clipping window,

$$\mathbf{N_i} \cdot [\mathbf{P(t)} - \mathbf{P_i}] = 0$$

$$\Rightarrow \quad \mathbf{N_i} \cdot [\mathbf{P_1} + (\mathbf{P_2} - \mathbf{P_1})t - \mathbf{P_i}] = 0$$

$$\Rightarrow \quad \mathbf{N_i} \cdot [\mathbf{P_1} - \mathbf{P_i}] + \mathbf{N_i} \cdot [\mathbf{P_2} - \mathbf{P_1}]\ t = 0$$

$$\Rightarrow \quad \mathbf{N_i} \cdot [\mathbf{P_1} - \mathbf{P_i}] + \mathbf{N_i} \cdot [\mathbf{P_2} - \mathbf{P_1}]\ t = 0$$

$$\Rightarrow \quad t = -\frac{\mathbf{N_i} \cdot [\mathbf{P_1} - \mathbf{P_i}]}{\mathbf{N_i} \cdot \mathbf{D}} \qquad \text{where } \mathbf{D} = \mathbf{P_2} - \mathbf{P_1} \text{ is the vector from } \mathbf{P_1} \text{ to } \mathbf{P_2} \qquad ...(1)$$

Note that $\mathbf{N_i} \cdot \mathbf{D}$ can be zero if either $\mathbf{D} = 0$ or $\mathbf{D}$ is perpendicular to $\mathbf{N_i}$. $\mathbf{D} = 0$ when $\mathbf{P_1} = \mathbf{P_2}$ implying $\mathbf{P_1}, \mathbf{P_2}$ is a single point $\mathbf{P_1}$. In that case $\mathbf{P_1}$ is at the outer side of E_i if $\mathbf{N_i} \cdot [\mathbf{P_1} - \mathbf{P_i}] < 0$; $\mathbf{P_1}$ **is** at the inner side of E_i if $\mathbf{N_i} \cdot [\mathbf{P_1} - \mathbf{P_i}] > 0$ and $\mathbf{P_1}$ is on E_i if $\mathbf{N_i} \cdot [\mathbf{P_1} - \mathbf{P_i}] = 0$. On the other hand $\mathbf{D}$ is perpendicular to $\mathbf{N_i}$ if $\mathbf{P_1P_2}$ is parallel to edge E_i.

So for a line segment P_1P_2 that is to be clipped against a n-sided convex clipping window (closed polygon), maximum n number of intersections of the line with the window edges can be found. That implies we will get n numbers of 't' or t_i ($i = 0$ to n) from eqn. (1) each corresponding to an intersection. Out of these the visible pair of intersections are to be identified.

In any line clipping algorithm the ability to quickly identify and separate totally visible and totally invisible line is important. But for parametric lines no simple, unique method for distinguishing these two categories of lines is available. Instead the relative position of t_i, s are found w.r.t the clip window by checking the sign of $\mathbf{N_i} \cdot \mathbf{D}$ for every t_i.

$\mathbf{N_i} \cdot \mathbf{D} > 0 \Rightarrow \cos\theta > 0 \Rightarrow \theta < 90° \Rightarrow E_i$ is nearer to $\mathbf{P_1}$ than $\mathbf{P_2} \Rightarrow t_i$ is near the beginning of line i.e. nearer to P_1 than P_2 and hence t_i can be termed as t_E or ***potentially entering*** the clip polygon. The maximum of all such t_E'*s* and 0, say $t_{E\,max}$, is found.

NOTES

NOTES

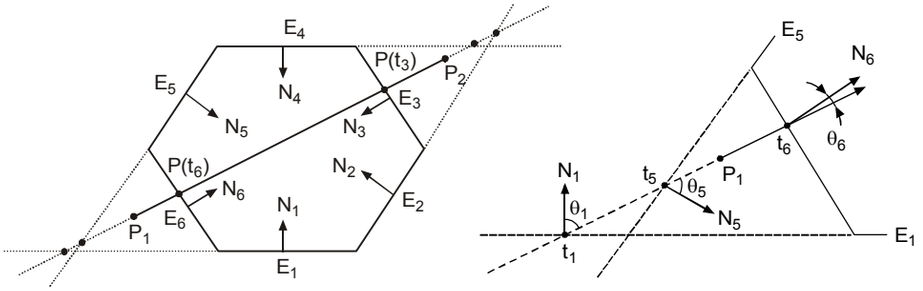
$\mathbf{N_i} \cdot \mathbf{D} < 0 \Rightarrow \cos\theta < 0 \Rightarrow \theta > 90° \Rightarrow E_i$ is nearer to P_2 than $P_1 \Rightarrow t_i$ is near the end of line i.e. nearer to P_2 than P_1 and hence t_i can be termed as t_L or ***potentially leaving*** the clip polygon. The minimum of all such t_L *s* and 1, say $t_{L\,min}$, is found.

The line segment between $\mathbf{P}(t_{E\,max})$ and $\mathbf{P}(t_{L\,min})$ is visible only if $t_{E\,max} \le t_{L\,min}$. [To ensure $P(t_{E\,max})$ and $\mathbf{P}(t_{L\,min})$ lies within $\mathbf{P_1P_2}$, if any $t_E > 1$ it is discarded and if any $t_L < 0$ it is discarded.]

Otherwise there is no visible portion of the line $\mathbf{P_1P_2}$.

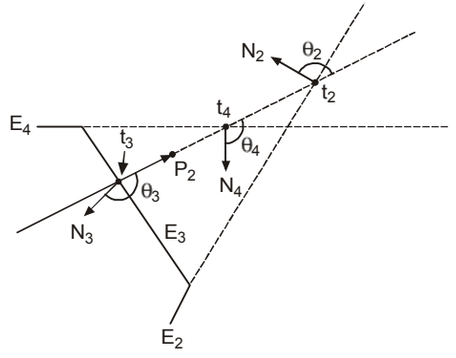
The above simple rule properly displays or ignores all classes of lines namely – Totally invisible, totally visible and partially visible. This is best understood through following illustrations.

Example 1 Partially Visible Line



(a) Hexagonal clipping window clips line P_1P_2. (b) Potentially entering intersections (t_E) near P_1 and E_6.

$\theta_1, \theta_5, \theta_6 < 90°$, t_E : $t_1 < t_5 < 0 < t_6$; $t_{E\,max} = t_6$



(c) Potentially leaving intersections (t_L) near P_2 and E_3.q_3, q_4, q_5 >90°, t_1 t_2 : $t_4 > 1 > t_3$ $t_{L\,min} = t_3$ $t_6 < t_3$ hence visible portion is from $P(t_6)$ to $P(t_3)$

Fig. 7.11

NOTES

Example 2 Totally Invisible Line

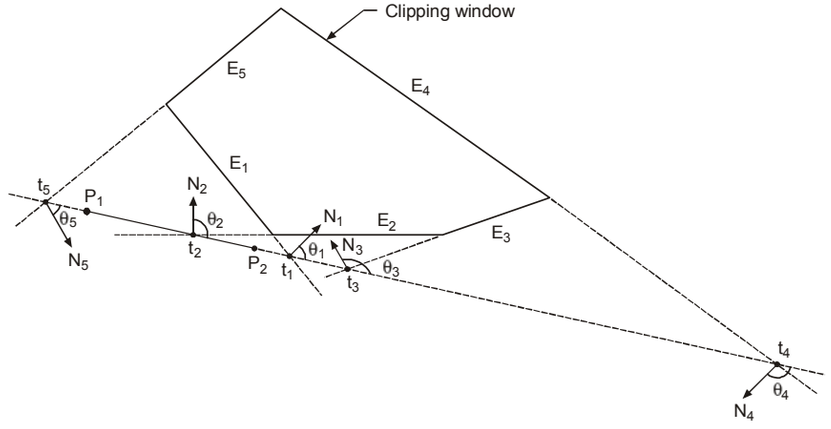


Fig. 7.12

Example 3 Totally Visible Line

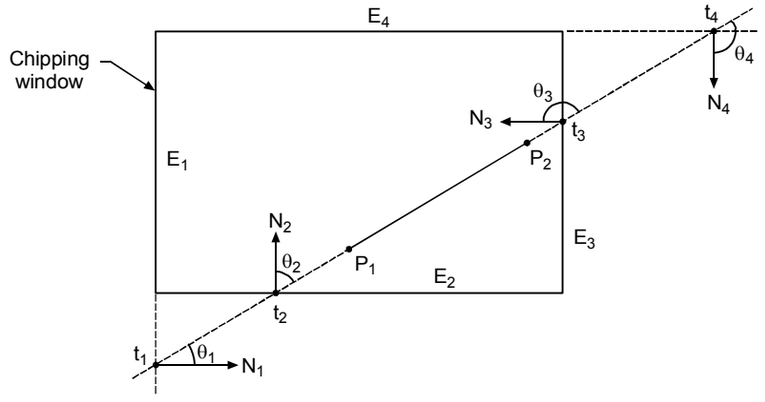


Fig. 7.13

Pseudocode of Cyrus-Beck Algorithm

Input: P_1, P_2 (end points of line to be clipped), n no. vertices V_i, ($i = 0$ to n) for a n sided convex clipping window, n no. points P_i on each window edge E_i (V_i s can be used as P_i s)

Initialize: $t_{E\,\max} = 0$, $t_{L\,\min} = 1$

Calculate: inner normals $\mathbf{N_i}$ for each edge E_i,

$$\mathbf{D = P_2 - P_1}$$

if ($P_1 = P_2$) then clip point P_1

else

NOTES

for (each edge E_i)

if $\mathbf{N_i} \cdot \mathbf{D} = 0$ then ignore E_i as it is parallel to the line

else

{ $t = -\frac{\mathbf{N_i} \cdot [\mathbf{P_1} - \mathbf{P_i}]}{\mathbf{N_i} \cdot \mathbf{D}}$

if $\mathbf{N_i} \cdot \mathbf{D} > 0$ then

if $t > t_{E\max}$ then $t_{E\max} = t$

else if $\mathbf{N_i} \cdot \mathbf{D} < 0$ then

if $t > t_{L\min}$ then $t_{L\min} = t$

end if

if $t_{E\max} \le t_{L\min}$ then draw line between $P(t_{E\max})$ and $P(t_{L\min})$

else no display

endif

}

endif

endfor

endif

7.3 POLYGON CLIPPING

7.3.1 Sutherland–Hodgman Algorithm

So far you have learnt techniques to clip any line by a clipping window. Using any of these techniques we can attempt to clip a polygon because a polygon is simply a set of connected straight lines (edges). But the problem is, each edge clipped separately using a line clipping algorithm will certainly not produce a truncated polygon as one would expect. Rather it would produce a set of unconnected line segments as if the polygon is exploded. Herein lies the need to use a different clipping algorithm to output truncated yet bounded region(s) from input polygon. *Sutherland - Hodgman* algorithm is one such standard method for clipping arbitrary shaped polygons with a rectangular clipping window.

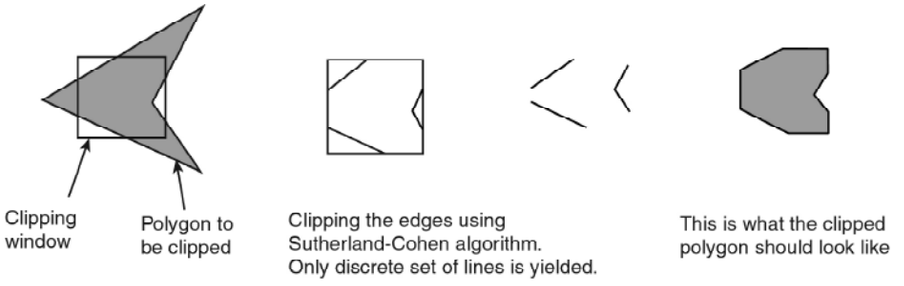


Fig. 7.14

NOTES

To clip a polygon correctly as shown above, the polygon must be tested against each edge of the clip rectangle; new edges must be added, and existing edges must be discarded, retained, or divided. Even multiple polygons may result from clipping a single polygon. This is unlike the case of Sutherland-Cohen line clipping which tests the line-endpoint outcode to see which clip-edge is crossed and clips only when necessary.

Sutherland–Hodgman clipper actually follows a *divide & conquer* strategy. It decomposes the problem of polygon-clipping against a clip window into identical subproblems. A subproblem is to clip all polygon edges (pair of vertices) in succession against a single infinite clip edge. The output is a set of clipped edges or pair of vertices that fall in the visible side w.r.t that clip edge. These set of clipped edges or output vertices are considered as input to the next subproblem of clipping against the second window edge. Thus considering the output of the previous subproblem as the input, each of the subproblems are solved sequentially, finally yielding the vertices that fall on or within the window boundary. These vertices connected in order forms the shape of the clipped polygon that may be optionally scan -filled.

While clipping polygon edges against a window edge we move from one vertex (V_i) to the next vertex (V_{i+1}) and decide the output vertex according to the four simple rules stated below.

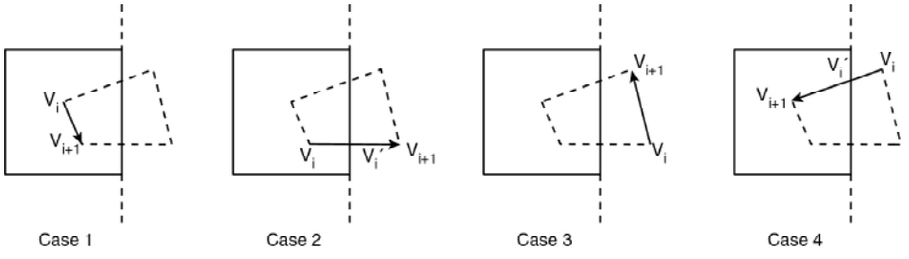


Fig. 7.15

	V_i	→	V_{i+1}	**Output vertex**
Rule 1:	inside (window)		inside	V_{i+1}
Rule 2:	inside		outside	V_i', the intersection with the window edge
Rule 3:	outside		outside	none
Rule 4:	outside		inside	V_i', V_{i+1}

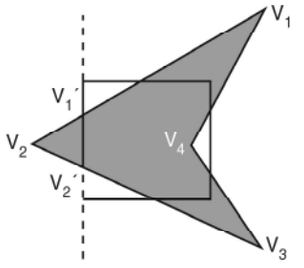
For edge $V_i \rightarrow V_{i+1}$, the starting vertex V_i is assumed to have already been considered as a candidate for output while it was the terminating vertex of the edge $V_{i-1} \rightarrow V_i$.

The intersection point V_i' is easily found as the x or y coordinates of V_i, V_{i+1} and the concerned window edge are already known. By assigning a 4 bit outcode to every vertex we can determine whether the point falls on the visible side (when

NOTES

code = 0) of the window edge or on the other side (when code ≠ 0).

Now let us see in steps what exactly happens while clipping our initial arrow shaped quadrilateral against a rectangular clip window following the above rules. Note the vertex list at each step.



(a)

STEP 1 – Clip against Left edge

Input vertex list [V_1, V_2, V_3, V_4]

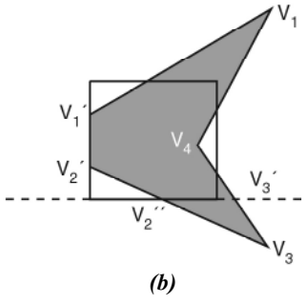
Edge $V_1 \rightarrow V_2$: output V_1'

Edge $V_2 \rightarrow V_3$: output V_2', V_3

Edge $V_3 \rightarrow V_4$: output V_4

Edge $V_4 \rightarrow V_1$: output V_1

Output vertex list [V_1', V_2', V_3, V_4, V_1]



(b)

STEP 2 – Clip against Bottom edge

Input vertex list [V_1', V_2', V_3, V_4 , V_1]

Edge $V_1' \rightarrow V_2'$: output V_2'

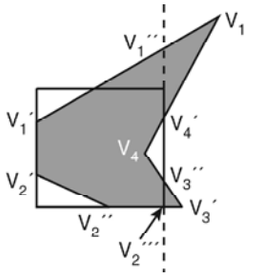
Edge $V_2' \rightarrow V_3$: output V_2''

Edge $V_3 \rightarrow V_4$: output V_3', V_4

Edge $V_4 \rightarrow V_1$: output V_1

Edge $V_1 \rightarrow V_1'$: output V_1'

Output vertex list [V_2', V_2'', V_3', V_4, V_1, V_1']



(c)

Fig. 7.16

STEP 3 – Clip against Right edge

Input vertex list [V_2', V_2'', V_3', V_4, V_1, V_1']

Edge $V_2' \rightarrow V_2''$: output V_2''

Edge $V_2'' \rightarrow V_3'$: output V_2'''

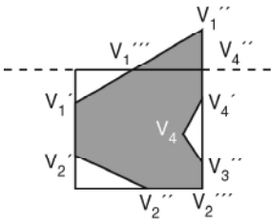
Edge $V_3' \rightarrow V_4$: output V_3'', V_4

Edge $V_4 \rightarrow V_1$: output V_4'

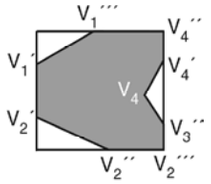
Edge $V_1 \rightarrow V_1'$: output V_1'', V_1'

Edge $V_1' \rightarrow V_2'$: output V_2'

Output vertex list [V_2'', V_2''', V_3'', V_4 , V_4', V_1'' , V_1', V_2']



(d)



(e)

Fig. 7.17

STEP 4 – Clip against Top edge

Input vertex list $[V_2'', V_2''', V_3'', V_4, V_4', V_1'', V_1', V_2']$

Edge $V_2'' \rightarrow V_2'''$: output V_2'''

Edge $V_2''' \rightarrow V_3''$: output V_3''

Edge $V_3'' \rightarrow V_4$: output V_4

Edge $V_4 \rightarrow V_4'$: output V_4'

Edge $V_4' \rightarrow V_1''$: output V_4''

Edge $V_1'' \rightarrow V_1'$: output V_1''', V_1'

Edge $V_1' \rightarrow V_2'$: output V_2'

Edge $V_2' \rightarrow V_2''$: output V_2''

Output vertex list $[V_2''', V_3'', V_4, V_4', V_4'', V_1''', V_1', V_2' V_2'']$

The final clipped polygon,

Output vertex list $[V_2''', V_3'', V_4, V_4', V_4'', V_1''', V_1', V_2', V_2'']$

We started off with 4 vertices and ended up with 9 new vertices except V_4, which survived the 4-step revision of vertices.

Because clipping against one clip-edge is independent of all others, it is possible to arrange the clipping stages in a pipeline. The input polygon is clipped against one edge and any points that are kept are passed on as input to the next stage of the pipeline. This way four polygons can be at different stages of the clipping process simultaneously for a rectangular clipping window. This is often implemented in hardware. The same algorithm, slightly modified, is found more suitable for hardware implementation. The modified approach recursively clips a single polygon edge (vertex) against all the window edges. The portion of the edge (if any) finally lying within the window boundary is saved. The process is repeated for the other polygon edges one after another. Thus this method generates the final clipped polygon without creating & storing any intermediate polygon definitions. If there are *n* polygon edges this method executes *n* cycles of window-edge (4 per cycle) processing whereas the previous approach looped through 4 cycles of polygon-edge ($\geq n$ per cycle) processing.

Though the above discussion assumes only rectangular clip window, the Sutherland –Hodgman algorithm will clip any polygon, convex or concave, against any convex polygonal clipping window. So the usual code-testing method for

NOTES

NOTES

determining the visibility of a polygon vertex w.r.t an arbitrary clip edge will not be applicable. We may use the following simple rule to determine whether a point is inside, on or outside the clip-edge. Remember if the successive edges of the clipping polygon are considered anticlockwise the inside of the polygon is always to the left; if considered clockwise it is to the right.

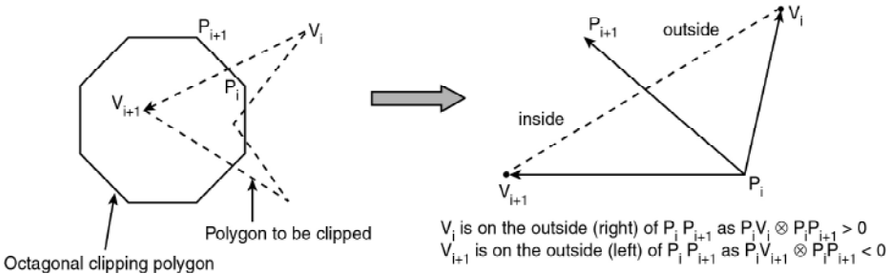


Fig. 7.18

If the position of any polygon vertex V_i is to be determined w.r.t a clipping edge P_iP_{i+1} in general, then the sign of the cross product of the vectors P_iV_i and P_iP_{i+1} is tested.

If sign of $P_iV_i \otimes P_iP_{i+1}$ is +ve, then V_i is to the right of line P_iP_{i+1}

If sign of $P_iV_i \otimes P_iP_{i+1}$ is –ve, then V_i is to the left of line P_iP_{i+1}

If $P_iV_i \otimes P_iP_{i+1} = 0$, then V_i is on the line P_iP_{i+1}

From the basic right hand rule of cross product of vectors, the above interpretation is easily understandable.

Pseudocode for Sutherland–Hodgman Algorithm

Define variables

inVertexArray is the array of input polygon vertices

outVertexArray is the array of output polygon vertices

Nin is the number of entries in inVertexArray

Nout is the number of entries in outVertexArray

n is the number of edges of the clip polygon

ClipEdge [x] is the *x*th edge of clip polygon defined by a pair of vertices

s, *p* are the start and end point respectively of current polygon edge

i is the intersection point with a clip boundary

j is the vertex loop counter

Define Functions

AddNewVertex (newVertex, Nout, outVertexArray)

: Adds newVertex to outVertexArray and then updates Nout

NOTES

```
InsideTest (testVertex, clipEdge[x])
                    :  Checks whether the vertex lies inside the clip edge
                       or not; returns TRUE if inside else returns FALSE
Intersect (first, second, clipEdge[x])
                    :  Clips polygon edge (first, second) against
                       clipEdge[x], outputs the intersection point
{                   :  begin main
x = 1
while (x ≤ n)       :  Loop through all the n clip edges
{
Nout = 0            :  Flush the outVertexArray
s = inVertexArray[Nin]  :  Start with the last vertex in inVertexArray
for j = 1 to Nin do     :  Loop through Nin number of polygon vertices
(edges)
    {
     p = inVertexArray[j]
     if InsideTest (p, clipEdge[x]) = = TRUE then :   Cases 1 and 4
      if InsideTest(s, clipEdge[x]) = = TRUE then
        AddNewVertex (p, Nout, outVertexArray) :   Case 1
         else
          i = Intersect(s, p, clipEdge[x])        :   Case 4
         AddNewVertex (i, Nout, outVertexArray)
         AddNewVertex (p, Nout, outVertexArray)
      end if
      else                          :i.e. if InsideTest (p, clipEdge[x]) = =
FALSE
                                        (Cases 2 and 3)
      if InsideTest(s, clipEdge[x]) = = TRUE then :   Case 2
         {
          Intersect(s, p, clipEdge[x])
         AddNewVertex (i, Nout, outVertexArray)
       end if                       :  No action for case 3
        s = p                       :  Advance to next pair of vertices
        j = j + 1
     end if                         :  end {for}
     }
x = x + 1                           :  Proceed to the next ClipEdge[x + 1]
```

NOTES

```
Nin = Nout
inVertexArray = outVertexArray   :  The ouput vertex array for the current
                                    clip edge becomes the input vertex
                                    array for the next clip edge
}                                :  end while
}                                :  end main
```

Check Your Progress

1. What is the purpose of line clipping?
2. What are the three types of line to determine the visibility of lines?

7.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. The purpose of clipping procedure is to determine which part of a scene or specifically which points, lines (or curves) or portions of the lines (or curves) of a scene lie inside the clipping window.
2. To determine visibility of lines w.r.t. the clipping window, we have to deal with three types of line.

   (i) Lines totally visible (ii) Lines totally invisible (iii) Lines partially visible

7.5 SUMMARY

- Line clipping procedure is to determine which part of a scene or specifically which points, lines (or curves) or portions of the lines (or curves) of a scene lie inside the clipping window.
- Explicit Line Clipping technique uses a 4-digit code to indicate which of the nine regions around a clipping window contain the end point of a line.
- Sutherland–Cohen is one of the most popular line clipping algorithm. The concept of assigning 4- bit region codes to the endpoints of a line and subsequent checking and AND operation of the endpoint codes to determine totally visible lines and totally invisible lines.
- Partially visible and totally invisible lines which cannot be identified by checking and operating (ANDing) endpoint codes are subdivided into two equal segments by finding the midpoint.
- Convex polygon for which the line joining any two interior points lie completely inside the polygon.
- Sutherland–Hodgman is learnt techniques to clip any line by a clipping window. Using any of these techniques we can attempt to clip a polygon because a polygon is simply a set of connected straight lines.

7.6 KEY WORDS

- **Convex Polygon:** It is a polygon for which the line joining any two interior points lie completely inside the polygon.
- **Clipping**: It is a method to selectively enable or disable rendering operations within a defined region of interest.
- **Line Clipping:** The process of removing lines or portions of lines outside of an area of interest.

NOTES

7.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Write a note on line clipping.
2. Write the Sutherland-Cohen algorithm.
3. Write the pseudocode for Sutherland-Hodgeman.

Long Answer Questions

1. Explain the explicit line clipping algorithm.
2. Explain the midpoint sub-division algorithm.
3. Write the pseudocode for Cyrus-Beck algorithm.
4. Describe polygon clipping alongwith Sutherland-Hodgman algorithm.

7.8 FURTHER READINGS

Hearn, Donal and M. Pauline Baker. 1990. *Computer Graphics*. New Delhi: Prentice-Hall of India.

Rogers, David F. 1985. *Procedural elements for Computer Graphics*. New York: McGraw-Hill.

Foley, D. James, Andries Van Dam, Steven K. Feiner and John F. Hughes. 1997. *Computer Graphics Principles and Practice*. Boston: Addison Wesley.

Mukhopadhyay, A. and A. Chattopadhyay. 2007. *Introduction to Computer Graphics and Multimedia*. New Delhi: Vikas Publishing House.

BLOCK - III

3D OBJECT REPRESENTATION

NOTES

UNIT 8 INTRODUCTION TO SURFACES

Structure

8.0 INTRODUCTION

In this unit, you will learn about the polygon surfaces, quadric surfaces and spline representations. *Spline* is a curve that connects two or more specific points, or that is defined by two or more points.

8.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand and define polygon surfaces and quadric surfaces
- Define spline
- Discuss spline specification

8.2 POLYGON SURFACES

The most commonly used boundary presentation for three-dimensional graphics objects, is a set of surface polygons which enclose an object interior. There are several graphics systems that can store object descriptions as sets of surface polygons. All surfaces can be described with linear equations that simplify the surface rendering (interpretation) and display of objects. For this reason, polygon descriptions are often referred to as standard computer graphics objects. In other

words, a polygonal representation is the only available description. Many packages allow objects to be described with other schemes as well (such as spline surfaces), that are then converted to polygonal representations for processing. A polygon representation for a polyhedron in particular defines the surface features of an object. But for other objects, surfaces are tessellated (i.e., tiled) to produce the polygon-mesh approximation. The surface of a cylinder is represented as a polygon mesh in figure 8.1. Since the wire-frame outline can be displayed quickly to give a general indication of the surface structure, such representations are common in graphics design and solid-modelling applications. To eliminate or reduce the presence of polygon edge boundaries, realistic renderings are produced by interpolating shading patterns across the polygon surfaces. The polygon-mesh approximation to a curved surface can be enhanced by dividing the surface into smaller polygon facets.

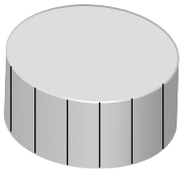


Fig. 8.1 *Wire-Frame Representation of a Cylinder with Back (Hidden) Lines removed*

8.3 QUADRIC SURFACES

Quadric surfaces are the most commonly used class of objects and are described using second-degree equations (called quadratics). They include spheres, ellipsoids, paraboloids, hyperboloids, etc. Quadric surfaces (particularly spheres and ellipsoids), are common elements of graphics scenes and they are often used in graphics packages as primitives components by which more complex objects can be constructed.

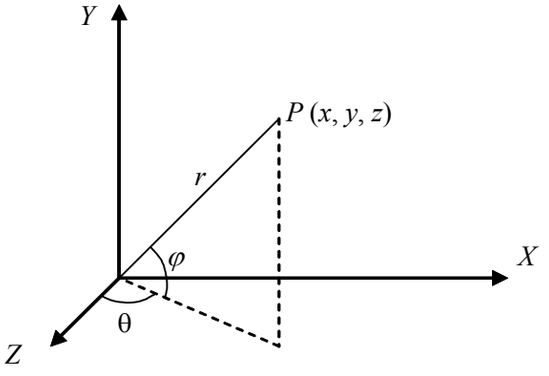


Fig. 8.2 *Parametric Coordinate Position (r, , j) on the Surface of a Sphere with Radius r*

NOTES

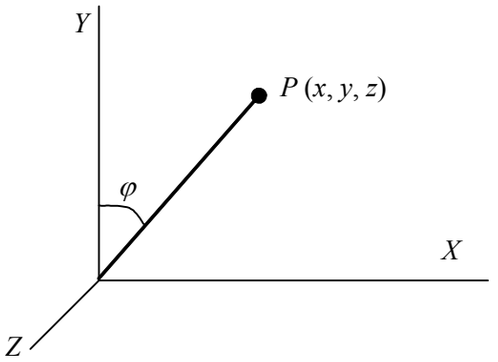


Fig. 8.3 *Spherical Coordinate Parameters (r, θ, φ) using Colatitude for angle φ*

Sphere

A spherical surface with radius *r* centered on the coordinate origin is defined as a set of points (x, y, z) in Cartesian coordinates that satisfies the equation

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

...(8.1)

We can also describe a spherical surface in the parametric form, using latitude and longitude angles as shown in Figure 8.2:

$$x = r\cos\varphi\cos\theta, \qquad \text{where,} \quad -\pi/2 \le \varphi \le \pi/2$$

...(8.2)

$$y = r\cos\varphi\sin\theta, \qquad \text{where,} \quad -\pi \le \theta \le \pi$$

$$z = r\sin\varphi$$

The parametric representation in equation 8.2 provides a symmetric range for the angular parameters θ and *φ*. Alternatively, we can write the parametric equation using standard spherical coordinates, where angle *φ* is specified as the co-latitude (Figure 8.3). Then, *φ* is defined over the range $0 \le \varphi \le \pi$, and θ is often taken in the range $0 \le \theta \le 2\pi$. A set up can also be represented by using parameters *u* and *v,* defined over the range from 0 to 1 by substituting $\varphi = \pi u$ and $\theta = 2\pi v$.

Ellipsoid

The surface of an ellipsoid can be described as an extension of a spherical surface, where the radii in three mutually perpendicular directions can have different values as shown in Figure 8.4. The Cartesian representation of points over the surface of an ellipsoid centered on the origin is expressed as follows:

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1$$

...(8.3)

NOTES

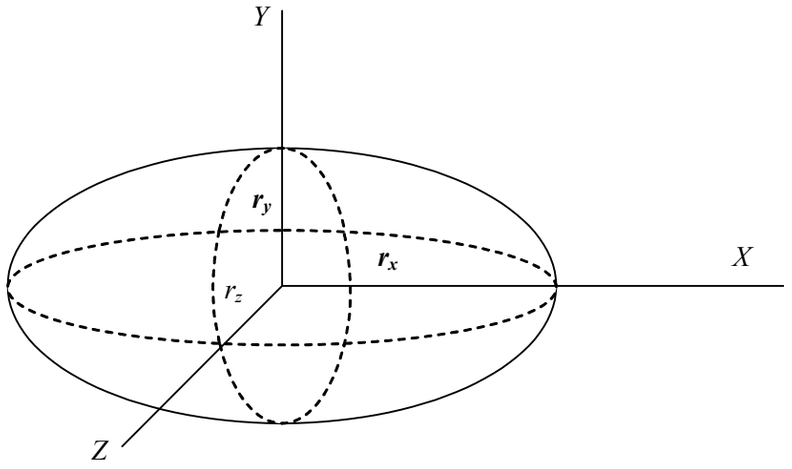


Fig. 8.4 *An Ellipsoid with Radii r_x, r_y and r_z whose Centre is (0, 0, 0)*

And a parametric representation for the ellipsoid in terms of the latitude angle φ in Figure 8.4 and the longitude angle θ in Figure 8.2 is as follows:

$$x = r_x \cos\varphi \cos\theta, \quad \text{where}, -\pi/2 \le \varphi \le \pi/2 \qquad ...(8.4)$$

$$y = r_y \cos\varphi \sin\theta, \quad \text{where}, -\pi \le \varphi \le \pi$$

$$z = r_z \sin\varphi$$

Torus

A torus can be defined as a doughnut-shaped object, as shown in Figure 8.5. It can be generated by rotating a circle or some other conic about a specified axis. We can write the Cartesian representation for points over the surface of a torus in the following form:

$$\left(r - \sqrt{\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2} \right)^2 + \left(\frac{z}{r_z}\right)^2 = 1 \qquad ...(8.5)$$

where *r* is any given offset value. Parametric representations for a torus are similar to those for an ellipse, except that angle φ extends over 360°. Using latitude and longitude angles φ and θ, we can describe the torus surface as the set of points that satisfy the following:

$$x = r_x(r + \cos\varphi)\cos\theta, \quad \text{where}, \quad -\pi \le \varphi \le \pi \qquad ...(8.6)$$

$$y = r_y(r + \cos\varphi)\sin\theta, \quad \text{where}, \quad -\pi \le \theta \le \pi$$

$$z = r_z \sin\varphi$$

NOTES

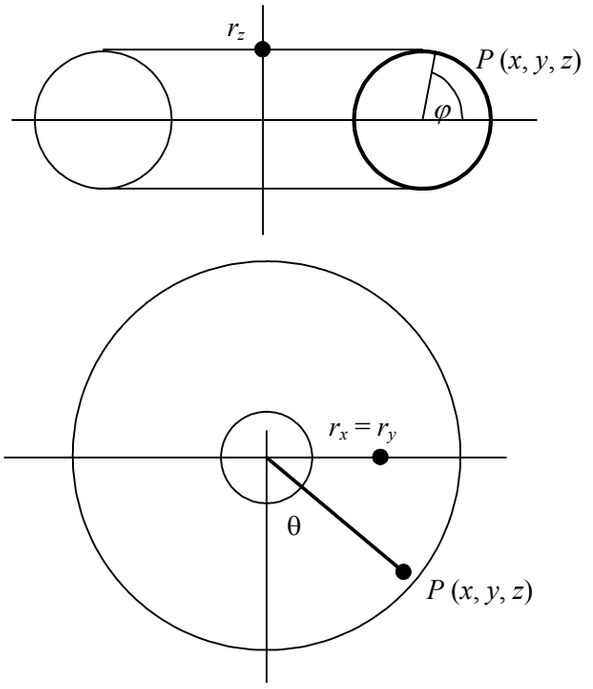


Fig. 8.5 *A Torus with a Circular Cross Section Centered on the Coordinate Origin*

Super Quadrics

Super quadrics is a class of objects that is generalized with the help of quadratic representations. Super quadrics can be formed by incorporating additional parameters into quadratic equations to provide additional flexibility for adjusting object shapes. The dimension of the object is equal to the number of additional parameters used: two parameters for surfaces and one parameter for curves.

8.4 SPLINE REPRESENTATIONS

In drafting terminology, a spline is a flexible strip that is used to produce a smooth curve by using a designated set of points. To hold the spline in position on the drafting table (so as to draw a curve) several small weights are distributed along the length of the strip. The term spline curve originally referred to a curve drawn thus. Mathematically, such a curve can be described with a piecewise cubic polynomial function in which the first and second derivatives are continuous across the various sections of the curve. In graphics terminology, the term spline curve refers to any composite curve formed with polynomial sections which satisfy specified continuity conditions at the boundary of the pieces. A spline surface can be described with two sets of orthogonal spline curves. There are several different kinds of spline specifications that are used in graphics applications. Each individual specification simply refers to a particular type of polynomial with certain specified boundary conditions. Splines are used in graphic applications to design curve and

surface shapes, to digitize drawings for computer storage, and to specify animation paths for objects in a scene. Typical CAD applications for splines include the design of automobile bodies, aircraft and spacecraft surfaces and ship hulls.

NOTES

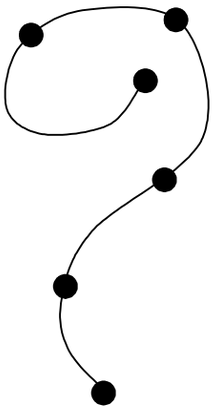


Fig. 8.6 *A Set of Six Control Point Interpolated with Piecewise Continuous Polynomial Sections*

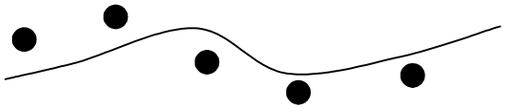


Fig. 8.7 *A Set of Five Control Points Approximated with Piecewise Continuous Polynomial Sections*

Interpolation and Approximation Spline

A spline curve can be specified by giving a set of coordinate positions, also called control points, which indicates the general shape of the curve. These control points are then fitted with piece wise continuously parametric polynomial functions in one of two ways. When polynomial sections are fitted so that the curve passes through each control point, as in Figure 8.6, the resulting curve is assumed to interpolate the set of control points. On the other hand, when the polynomials are set to the general control-point path without necessarily passing through any control point, the resulting curve is said to approximate the set of control points shown in Figure 8.7. Interpolation curves are commonly used to digitize images or to specify the path of an animation.

Approximation curves are primarily used as design tools to arrange object surfaces. A spline curve can be defined, modified, and manipulated by performing operations on the control points. By interactively selecting spatial positions for the control points, a graphics user can set up an initial curve. After the polynomial fit is displayed for a given set of control points, the designer can then reposition some or all of the control points to restructure the shape of the curve. In addition, the curve can be translated, rotated, or scaled with transformations applied to the control points. Computer-aided Design (CAD) packages can also insert extra

NOTES

control points to aid a designer in adjusting the curve shapes. The convex polygon boundary that encloses a set of control points is called the convex hull.

One way to envision the shape of a convex hull is to imagine a rubber band stretched around the positions of the control points so that each control point is either on the perimeter of the hull or inside it as shown in Figure 8.8. A measure for the deviation of a curve or surface from the region bounding the control points is provided by convex hulls. Some splines are bounded by the convex hull, thus ensuring that the polynomials smoothly follow the control points without erratic oscillations. Also, the polygon region inside the convex hull is useful in some algorithms as a clipping region.

A poly line connecting the sequence of control points for an approximation spline is usually displayed to remind a designer of the control-point ordering. This set of connected line segments is often referred to as the control graph of the curve. Other names for the series of straight-line sections connecting the control points in the order specified are control polygon and characteristic polygon. Figure 8.9 shows the shape of the control graph for the control-point sequences given in the Figure 8.8.

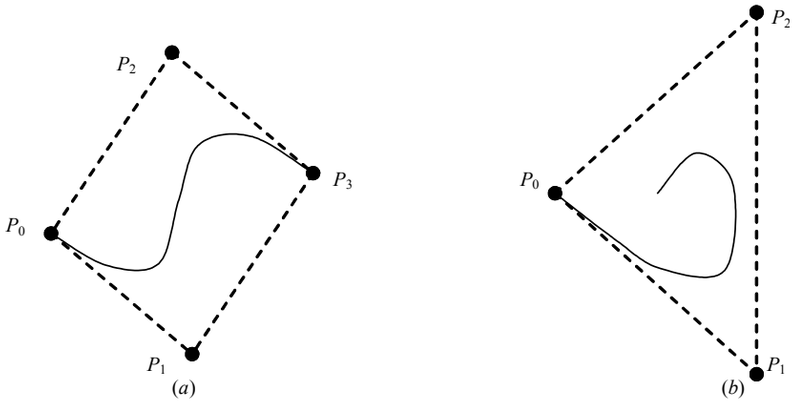


Fig. 8.8 *Convex-Hull Shapes (Dashed Lines) for Two Sets of Control Points*

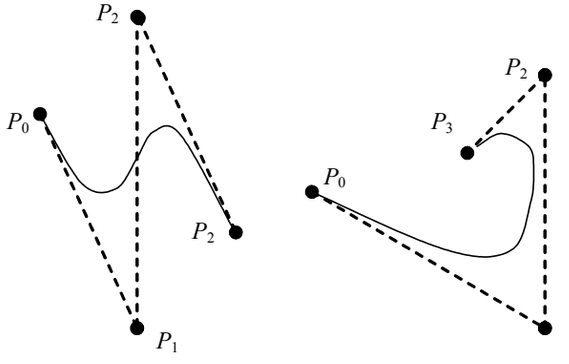


Fig. 8.9 *Control-Graph Shapes (Dashed Lines) for Two different Sets of Control Points*

NOTES

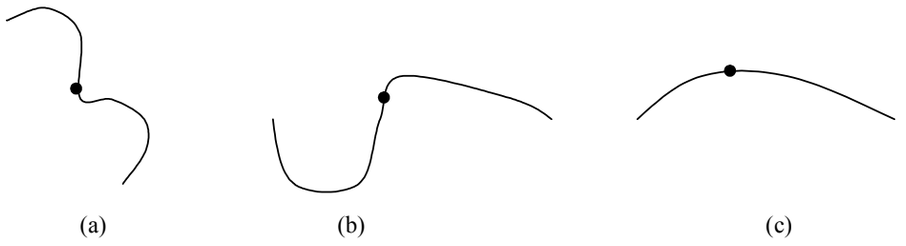


Fig. 8.10 *Piecewise Construction of a Curve by Joining Two Curve Segments using different Orders of Continuity: (a) Zero-order Continuity Only (b) First-order Continuity (c) Second-order Continuity.*

Parametric Continuity Conditions

To ensure a smooth transition from one section of a piecewise parametric curve to the next, we can impose various continuity conditions at the connection points. If each section of a spline is described with a set of parametric coordinate functions of the following form:

$$x = x(u), \qquad y = y(u), \qquad z = z(u), \qquad u_1 \le u \le u_2 \quad ...(8.7)$$

we set parametric continuity by matching the parametric derivatives of adjoining curve sections at their common boundary. Zero-order parametric continuity, described as C^1 continuity, means simply, that the curves meet. That is, the values of *x*, *y* and z evaluated at *u,* for the first curve section are equal, respectively, to the values of *x*, *y* and *z* evaluated at *u,* for the next curve section. First-order parametric continuity, referred to as C^1 continuity, means that the first parametric derivatives (tangent lines) of the coordinate functions in equation 8.7 for two successive curve sections are equal at their joining point. Second-order parametric continuity, or C^2 continuity, means that both the first and second parametric derivatives of the two curve sections are the same at the intersection. Higher-order parametric continuity conditions are defined similarly. Figure 8.10 shows examples of C^0, C^1, and C^2 continuity. In second-order continuity, the rates of change of the tangent vectors for connecting sections are equal at their intersection. Thus, the tangent line transitions smoothly from one section of the curve to the next. See Figure 8.10(c). But in first-order continuity, the rates of change of the tangent vectors for the two sections can be quite different, so that the general shapes of the two adjacent sections can change abruptly. (See Figure 8.10(b)). First-order continuity is often sufficient to digitize drawings and some design applications, while second-order continuity is useful to set-up animation paths for camera motion and for many precision Computer Aided Design requirements. A camera travelling along the curve path given in Figure 8.10(b), with equal steps in parameter *u*, would experience an abrupt change in acceleration at the boundary of the two sections, producing a discontinuity in the motion sequence. But if the camera were travelling along the path in Figure 8.10(c), the frame sequence for the motion would smoothly transition across the boundary.

NOTES

Geometric Continuity Conditions

An alternate method for joining two successive curve sections is to specify conditions for geometric continuity. In this case, we only require parametric derivatives of the two sections to be proportional to each other at their common boundary instead of equal to each other. Zero-order geometric continuity, described as G^0 continuity is the same as zero-order parametric continuity. That is, the two curves sections must have the same coordinate position at the boundary point. First-order geometric continuity, or G_1 continuity, means that the parametric first derivatives are proportional at the intersection of two successive sections. If we denote the parametric position on the curve as $P(u)$, the direction of the tangent vector $P'(u)$, but not necessarily its magnitude, will be the same for two successive curve sections at their joining point under G^1 continuity. Second-order geometric continuity, or G^2 continuity, means that both the first and second parametric derivatives of the two curve sections are proportional at their boundary. Under *GL* continuity, curvatures of two curve sections will match at the joining position. A curve generated with geometric continuity conditions is similar to the one generated with parametric continuity, but with slight differences in the shape of the curve.

Spline Specifications

There are three equivalent methods for specifying a particular spline representation. These are as follows:

(i) We can state the set of boundary conditions that are imposed on the spline.

(ii) We can state the matrix that characterizes the spline.

(iii) We can state the set of blending functions (or basis functions) that determine how specified geometric constraints on the curve are combined to calculate positions along the curve path.

To illustrate these three equivalent specifications, suppose we have the following parametric cubic polynomial representation for the x coordinate along the path of a spline section:

Boundary conditions for this curve might be set, for example, on the endpoint coordinates $x(0)$ and $x(1)$ and on the parametric first derivatives at the endpoints $x'(0)$ and $x'(1)$. These four boundary conditions are sufficient to determine the values of the four coefficients a_x, b_x, c_x, and d_x. From the boundary conditions, we can obtain the matrix that characterizes this spline curve by first rewriting equation 8.7 as the matrix product

$$x(u) = \begin{bmatrix} u^3 & u^2 & u^1 \end{bmatrix} \begin{bmatrix} a_x \\ b_x \\ c_x \\ d_x \end{bmatrix} \qquad ...(8.8)$$

$$= U.C$$

where U is the row matrix of powers of parameter u, and C is the coefficient column matrix. Using the equation 8.8, we can write the boundary conditions in the matrix form and solve for the coefficient matrix C as,

NOTES

$$C = M_{spline}.M_{geom}$$

...(8.9)

where M_{goem} is a four-element column matrix containing the geometric constraint values (boundary conditions) on the spline; and M_{spline} is the 4×4 matrix that transforms the geometric constraint values to the polynomial coefficients and provides a characterization for the spline curve. Matrix M_{geom} contains control point coordinate values and other geometric constraints that have been specified. Thus, we can substitute the matrix representation for C into equation 8.8 to obtain,

$$x(u) = U.M_{spline}.M_{geom}$$

...(8.10)

The matrix, M_{spline} characterizing a spline representation, sometimes called the basis matrix, is particularly useful for transforming from one spline representation to another. Finally, we can expand equation 8.10 to obtain a polynomial representation for x coordinate in terms of the geometric constraint parameters,

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

...(8.11)

where g_k are the constraint parameters, such as the control-point coordinates and slope of the curve at the control points, and $BF_k(u)$ are the polynomial blending functions. In the following sections, we discuss some commonly used splines and their matrix and blending-function specifications.

Check Your Progress

1. What is torus?
2. What are super quadrics?

8.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. A torus can be defined as a doughnut-shaped object.
2. Superquadrics is a class of objects that is generalized with the help of quadratic representations. Super quadrics can be formed by incorporating additional parameters into quadratic equations to provide additional flexibility for adjusting object shapes.

NOTES

8.6 SUMMARY

- The most commonly used boundary presentation for three-dimensional graphics objects, is a set of surface polygons which enclose an object interior. There are several graphics systems that can store object descriptions as sets of surface polygons.
- Quadric surfaces are the most commonly used class of objects and are described using second-degree equations (called quadratics). They include spheres, ellipsoids, paraboloids, hyperboloids, etc.
- The surface of an ellipsoid can be described as an extension of a spherical surface, where the radii in three mutually perpendicular directions can have different values.
- A torus can be defined as a doughnut-shaped object. It can be generated by rotating a circle or some other conic about a specified axis.
- Super quadrics is a class of objects that is generalized with the help of quadratic representations. Super quadrics can be formed by incorporating additional parameters into quadratic equations to provide additional flexibility for adjusting object shapes.
- A spline is a flexible strip that is used to produce a smooth curve by using a designated set of points. To hold the spline in position on the drafting table several small weights are distributed along the length of the strip.

8.7 KEY WORDS

- **Torus:** A torus can be defined as a doughnut-shaped object, it can be generated by rotating a circle or some other conic about a specified axis.
- **Super Quadrics:** Super quadrics is a class of objects that is generalized with the help of quadratic representations.
- **Spline:** *It* is a curve that connects two or more specific points, or that is defined by two or more points.

8.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Write a note on polygon surfaces.
2. What are quadric surfaces?
3. Write the Cartesian representation for points over the surface of a torus.

Long Answer Questions

1. Explain the parametric continuity conditions.
2. What are spline specifications? Explain.

NOTES

8.9 FURTHER READINGS

Hearn, Donal and M. Pauline Baker. 1990. *Computer Graphics*. New Delhi: Prentice-Hall of India.

Rogers, David F. 1985. *Procedural elements for Computer Graphics*. New York: McGraw-Hill.

Foley, D. James, Andries Van Dam, Steven K. Feiner and John F. Hughes. 1997. *Computer Graphics Principles and Practice*. Boston: Addison Wesley.

Mukhopadhyay, A. and A. Chattopadhyay. 2007. *Introduction to Computer Graphics and Multimedia*. New Delhi: Vikas Publishing House.

UNIT 9 CURVE AND SURFACES

NOTES

Structure

9.0 INTRODUCTION

In this unit, you will learn about the Hermite curve, Bezier curve and surfaces. Bezier curves are used in computer graphics to produce curves which appear reasonably smooth at all scales. Bezier surfaces are a species of mathematical spline used in computer graphics, computer-aided design, and finite element modelling. You will also learn about the illumination models and polygon rendering methods. An illumination model may also be called a shading model or a lighting model. It calculates the intensity of light that one can see at a given point on the surface of an object. Surface rendering algorithms use intensity calculations from an illumination model to determine the light intensity for all projected pixel positions for various surfaces in an object.

9.1 OBJECTIVES

After going through this unit, you will be able to:

- Describe about Hermite curve
- Discuss about Bezier curve and surfaces
- Define B- Spline curve and surfaces
- Understand basic Illumination models
- Explain polygon rendering methods

NOTES

9.2 HERMITE CURVE

Hermite spline is a spline curve where each polynomial of the spline is in Hermite form. A cubic Hermite spline or cubic Hermite interpolator is a spline where each piece is a third-degree polynomial specified in Hermite form i.e., by its values and first derivatives at the end points of the corresponding domain interval. These are typically used for interpolation of numeric data specified at given argument values $x_1, x_2, x_3 \ldots\ldots x_n$ to obtain a smooth continuous function. The data should consist of the desired function value and derivative at each x_k. The Hermite formula is applied to each interval (x_k, x_{k+1}) separately. The resulting spline will be continuous and will have continuous first derivative.

Cubic polynomial splines can be specified in other ways, the Bezier form being the most common. These two methods provide the same set of splines, and data can be easily converted between the Bezier and Hermite forms so that the names are often used as if they were synonymous.

9.3 BEZIER CURVES AND SURFACES

Bezier curve is the spline approximation method that was developed by the French engineer Pierre Bezier to design automobile bodies. The Bezier spline has a number of properties that makes it highly useful and convenient for surface and curve design. Bezier curves and surfaces are easy to implement. For these reasons, Bezier splines are widely available in various graphics packages like CAD systems, and in assorted drawing and painting softwares.

Practically, a section of a Bezier curve can be fitted to any number of control points. The number of control points to be approximated and their relative position determines the degree of the Bezier polynomial. Similar to the interpolation splines, a Bezier curve can be specified with boundary conditions, with blending functions, or with a characterizing matrix. In case of general Bezier curves, the blending-function specification is the most convenient function. Suppose we are given $n + 1$ control-point positions: $p_k = (x_k, y_k, z_k)$, with k varying from 0 to n. These coordinate points can be blended to produce the following position vector $P(u)$, which describes the path of an approximating Bezier polynomial function between P_0 and P_n:

$$P(u) = \sum_{k=0}^{n} P_k . BEZ_{k,n}(u), \qquad 0 \le u \le 1 \qquad \ldots(9.1)$$

NOTES

The Bezier blending functions $BEZ_{k,n}(u)$ are the Bernstein polynomials that are defined as follows:

$$BEZ_{k,n}(u) = C(n,k)u^k(1-u)^{n-k} \quad ...(9.2)$$

where the $C(n, k)$ are the binomial coefficients defined as follows:

$$C(n.k) = \frac{n!}{k!(n-k)!} \quad ...(9.3)$$

Equivalently, we can define Bezier blending functions with the recursive calculation as follows:

$$BEZ_{k,n}(u) = (1-u)BEZ_{k,n-1}(u) + uBEZ_{k-1,n-1}(u), \qquad n > k \geq 1 \quad ...(9.4)$$

with $BEZ_{k,k} = u^k$, and $BEZ_{0,k} = (1-u)^k$. The vector equation 9.1 represents a set of three parametric equations for the individual curve coordinate as follows:

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

$$y(u) = \sum_{k=0}^{n} y_k BEZ_{k,n}(u) \quad ...(9.5)$$

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

As a rule, a Bezier curve is a polynomial of degree one less than the number of control points used: three points generate a parabola, four points a cubic curve, and so forth. Figure 9.1 demonstrates the appearance of Bezier curves for various selections of control points in the *xy*-plane (z = 0). With certain control-point placements, however, we obtain degenerate Bezier polynomials. For example, a Bezier curve generated with three collinear control points is a straight-line segment. And a set of control points that are all at the same coordinate position produces a Bezier 'curve' that is a single point. Bezier curves are commonly applied in painting and drawing packages, as well as CAD systems, since they are easy to implement and they are reasonably powerful in a curve design. Efficient methods to determine coordinate positions along a Bezier curve can be set up using recursive calculations. For example, successive binomial coefficients can be calculated as follows:

$$C(n,k) = \frac{n-k+1}{k} C(n,k-1)$$

NOTES

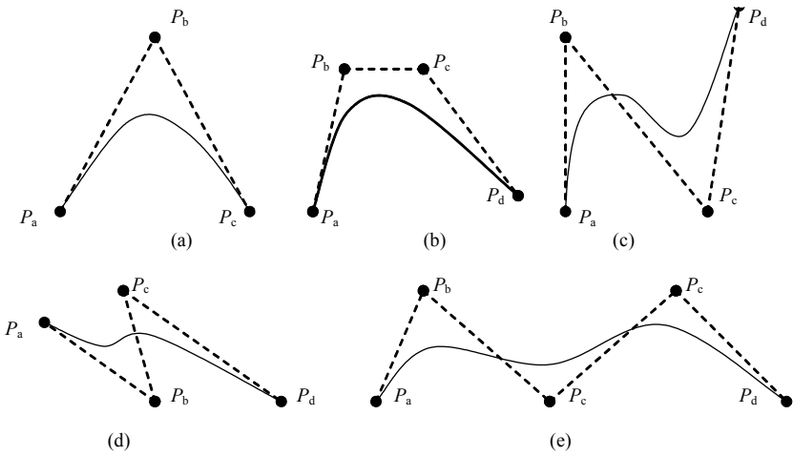


Fig. 9.1 *Examples of Two-Dimensional Bezier Curves Generated from Three, Four and Five Control Points. Dashed Lines Connect the Control-Point Positions*

The following example program illustrates a method for generating Bezier curves:

```
#include<stdio.h>
#include<graphics.h>
#include<bios.h>
#include<stdlib.h>
typedef double CoArr[4];
char buf[20];
//function definition for drawing Bezier curve
void BezierCurve(CoArr x, CoArr y, int n, int value)
{
    int i, a, b;
    double delta = 1.0/n;
    double t = 0, T;
setcolor(8); //defining color for curve
    for(i=0;i<4;i++)
       line(x[i],y[i],x[(i+1)%4],y[(i+1)%4]);
    for(i = 0; i<n; i++)
    {
    t = t + delta;
    T = 1-t;
    a = x[0]*T*T*T+3*t*T*T*x[1]+3*t*t*T*x[2]+t*t*t*x[3];
    b = y[0]*T*T*T+3*t*T*T*y[1]+3*t*t*T*y[2]+t*t*t*y[3];
       putpixel(a,b,value);
```

NOTES

```
    }
}
main()
{
    CoArr x, y;
    int val;
int gd = DETECT, gmode;
    int key;
    int flag = 1;
    double *movey;
double *movex;
char *str;
 x[0]=100; y[0]=100;
 x[3]=400; y[3]=100;
 x[1]=200; y[1]=50;
 x[2]=300; y[2]=50;
    initgraph(&gd,&gmode,"c:\\tc\\bgi");
//function calling for drawing Bezier curve
    BezierCurve(x,y,1000,15);
    movex = &x[1];
    movey = &y[1];
    while((key = bioskey(0))!=283)
    {
       cleardevice();
       switch(key)
       {
        case 3849:
          if(flag == 0)
          {
             flag = 1;
             movex = &x[0]; movey = &y[0];
          }
          else if(flag == 1)
          {
             flag = 2;
```

```
      movex=&x[1]; movey = &y[1];
   }
   else if(flag == 2)
   {
      flag = 3;
      movex = &x[2]; movey = &y[2];
   }
   else if(flag == 3)
   {
      flag = 0;
      movex = &x[3]; movey = &y[3];
   }
   break;
case 18432:
   *movey =*movey - 5; break;
case 20480:
   *movey =*movey + 5; break;
case 19200:
   *movex = *movex - 5; break;
case 19712:
   *movex =*movex + 5; break;
}
if(flag ==0)
{
   setcolor(4);
   circle(x[3],y[3],3);
}
else if(flag == 1)
{
   setcolor(4);
   circle(x[0],y[0],3);
}
else if(flag == 2)
{
   setcolor(GREEN);
```

NOTES

NOTES

```
            circle(x[1],y[1],3);
          }
          else if(flag == 3)
          {
            setcolor(GREEN);
            circle(x[2],y[2],3);
          }
          value = (int)x[1];
          itoa(val,str,10);
          setcolor(4);
          outtextxy(50,50,str);
          value = (int)y[1];
          itoa(val,str,10);
          setcolor(4);
          outtextxy(80,50,str);
          value = (int)x[2];
          itoa(val,str,10);
          setcolor(4);
          outtextxy(50,80,str);
          value = (int)y[2];
          itoa(val,str,10);
          setcolor(4);
          outtextxy(80,80,str);
          BezierCurve(x,y,1000,15);
      }
  return 0;
  }
```

Properties of Bezier Curves

A very useful property of a Bezier curve is that it always passes through the first and last control points. That is, the boundary conditions at the two ends of the curve are as follows:

$$P(0) = P_0 \qquad \ldots(9.6)$$

$$P(1) = P_n$$

NOTES

These are the values of the parametric first derivatives of a Bezier curve at the endpoints. Three-dimensional points can he calculated from control-point ordinates as follows:

$$P'(0) - np_0 + np_1 \qquad ...(9.7)$$

$$P'(1) - np_{n-1} + np_n$$

Thus, the slope at the starting of the curve is across the line joining the first two control points, and the slope at the end of the curve is across the line joining the last two control points. In the same way, the parametric second derivatives of a Bezier curve at the endpoints are calculated as follows:

$$P''(0) = n(n-1)\{(p_2 - p_1) - (p_1 - p_0) \qquad ...(9.8)$$

$$P''(1) = n(n-1)\{(p_{n-2} - p_{n-1}) - (p_{n-1} - p_n)$$

Another important property of any Bezier curve is that it lies within the convex hull (convex polygon boundary) of the control points. This follows from the properties of Bezier blending functions: they are all positive and their sum is always 1.

$$\sum_{k=0}^{n} BEZ_{k,n}(u) = 1 \qquad ...(9.9)$$

so that any curve position is simply the weighted sum of the control-point positions. The convex-hull property for a Bezier curve ensures that the polynomial smoothly follows the control points without erratic oscillations.

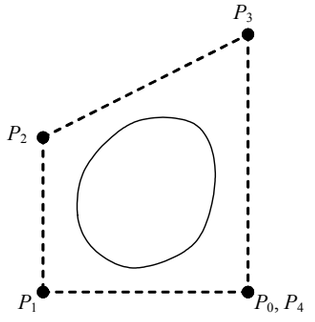


Fig. 9.2 *A Closed Bezier Curve Generated with the Same Starting and End Point*

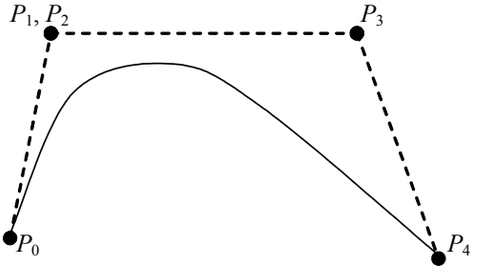


Fig. 9.3 *A Bezier Curve is made to Pass Closer to a Given Coordinate by Assigning Multiple (P_1, P_2) Control Points*

Design Techniques Using Bezier Curves

NOTES

Closed Bezier curves are generated by specifying the first and last control points at the same position, as in the example shown in Figure 9.2. Also, specifying multiple control points at a single coordinate position gives more weight to that position. In Figure 9.3, a single coordinate position is input as two control points, and the resulting curve is pulled nearer to this position. A Bezier curve can fit any number of control points, but this requires the calculation of polynomial functions of a higher degree. When complicated curves are to be generated, they can be formed by piecing several Bezier sections of lower degree together. Piecing together smaller parts also gives us better control over the shape of the curve in small regions. Since Bezier curves pass through endpoints, it is easy to match curve sections (zero order continuity). Additionally, Bezier curves have the important property that the tangent to the curve at an endpoint is across the line joining that control point to the adjacent control point. Therefore, to obtain first-order continuity between curve sections, we can pick control points of a new section that will be along the same straight line as control points of the previous section this is shown in Figure 9.4. We obtain C^1 continuity by choosing the first control point of the new section as the last control point of the previous section and by positioning the second control point of the new section at position P'_0.

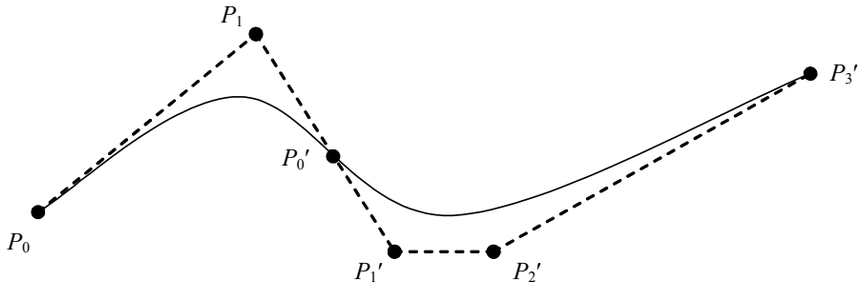


Fig. 9.4 *Piecewise Approximation Curve formed with Two Bezier Sections, Zero-Order and First-Order Continuity are Attained between Curve Sections by setting $P_0' = P_2$ and by making Points P_1, P_2 and P_1' Collinear*

Thus, the three control points are collinear and equally spaced. We obtain continuity between two Bezier sections by calculating the position of the third control point of a new section in terms of the positions of the last three control points of the previous section as. Requiring second-order continuity of Bezier curve sections can be unnecessarily restrictive. This is especially true with cubic curves, which have only four control points per section. In this case, second-order continuity fixes the position of the first three control points and leaves us only one point that we can use to adjust the shape of the curve segment.

The Cubic Bezier Curves

NOTES

Many graphics packages make available only cubic spline functions. This facilitates reasonable design flexibility and avoids the increased calculations required with higher-order polynomials. Cubic Bezier curves can be generated with four control points. The four blending functions for cubic Bezier curves that are obtained by substituting $n = 3$, for $k = 0, 1, 2, 3$ into equation 9.2 are as follows:

$$\begin{aligned} BEZ_{0,3}(u) &= (1-u)^3 \\ BEZ_{1,3}(u) &= 3u(1-u)^2 \\ BEZ_{2,3}(u) &= 3u^2(1-u) \\ BEZ_{3,3}(u) &= u^3 \end{aligned} \qquad ...(9.10)$$

The form of the blending functions determine how the control points influence the shape of the curve for values of parameter u over the range from 0 to 1. At $u = 0$, the only non-zero blending function is $BEZ_{0,3}$ which has the value 1. At $u = 1$, the only non-zero function is $BEZ_{3,3}$ with a value of 1 at that point. Thus, the cubic Bezier curve will always pass through control points p_0 and p_3. The other functions, $BEZ_{1,3}$ and $BEZ_{2,3}$, influence the shape of the curve, at intermediate values of parameter u, so that the resulting curve tends toward the points p_1, and p_2. Blending function $BEZ_{1,3}$ is maximum at $u = \frac{1}{3}$, and $BEZ_{2,3}$ is maximum at $u = \frac{2}{3}$.

Bezier Surfaces

We can use two sets of orthogonal Bezier curves to design an object surface by specifying by an input mesh of control points. The parametric vector function for the Bezier surface is formed as the Cartesian product of Bezier blending functions as follows:

$$P(u,v) = \sum_{j=0}^{m} \sum_{k=0}^{n} p_{j,k} BEZ_{j,m}(v) BEZ_{k,n}(u) \qquad ...(9.11)$$

with $p_{j,k}$ specifying the location of the by control points. Figure 9.5 illustrates two Bezier surface plots in which the dashed lines connect the control points. The control points at the surface are connected by dashed lines, and the solid lines show curves of constant values u and v. Each curve of constant u is plotted by varying v over the interval from 0 to 1, with u fixed at one of the values in this unit interval. Curves of constant v are plotted similarly.

NOTES

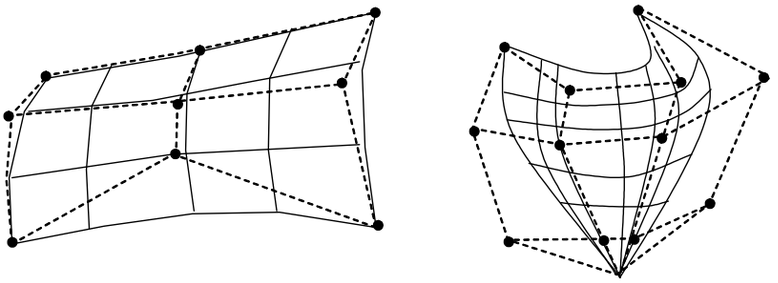


Fig. 9.5 *Bezier Surfaces Constructed for (a) m = 3, n = 3, and (b) m = 4, n = 4*

Bezier surfaces have the same properties as Bezier curves, and they provide a convenient method for interactive design applications. For each surface patch, we can select a mesh of control points in the *xy* 'ground' plane. Then we choose elevations above the ground plane for the z-coordinate values of the control points. Patches can then be pieced together using boundary constraints. Figure 9.6 illustrates a surface formed with two Bezier sections. The dashed lines in this figure connect specify control points. A smooth transition from one section to the other is assured by establishing both zero-order and first-order continuity at the boundary line. Zero-order continuity is obtained by matching control points at the boundary. First-order continuity is obtained by choosing control points along a straight line across the boundary and by maintaining a constant ratio of collinear line segments for each set of specified control points across section boundaries.

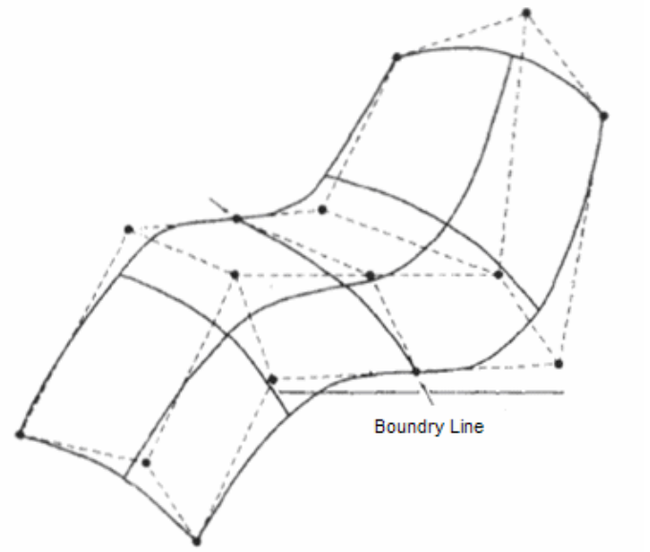


Fig. 9.6 *An Illustration of a Composite Bezier Surface Constructed with Two Bezier Sections, joined at the Indicated Boundary Line*

Example 9.1: A cubic Bezier curve is defined over the control points (1, 1), (2, 3), (4, 4) and (6, 1). Calculate the parametric mid points of this curve and show that its gradient *dy*/*dx* is 1/7.

NOTES

Solution: The blending functions for these control points are as follows:

$$B_{0,3}(u) = (1-u)^3$$

$$B_{1,3}(u) = 3u(1-u)^2$$

$$B_{2,3}(u) = 3u^2(1-u)$$

$$B_{3,3}(u) = u^3$$

The *x*-coordinate of the Bezier curve are as follows:

$$x(u) = x_0 B_{0,3}(u) + x_1 B_{1,3}(u) + x_2 B_{2,3}(u) + x_3 B_{3,3}(u)$$

$$= (1-u)^3 + 6u(1-u)^2 + 12u(1-u) + 6u^3$$

Similarly the, y-coordinates of the Bezier curve are as follows:

$$y(u) = (1-u)^3 + 9u(1-u)^2 + 12u(1-u) + u^3$$

The parameter *u* lies between 0 and 1. Therefore the midpoint of the curve is *u* = ½. By putting this value in *x*(*u*) and *y*(*u*) equations we get parametric midpoint coordinates as (27/8, 23/8).

By differentiating the *x*(*u*) equation with respect to *u* we get,

$$\frac{dx}{du} = -3(1-u)^2 - 12u(1-u) + 6(1-u)^2 - 12u^2 + 24u(1-u) + 18u^2$$

Similarly by differentiating the *y*(*u*) equation with respect to *u* we get,

$$\frac{dy}{du} = -3(1-u)^2 u(1-u) + 9(1-u) + 9(1-u)^2 - 12u^2 + 24(1-u)u + 3u^2$$

The gradient at any point is given by,

$$\left(\frac{dy}{dx}\right)_u = \frac{\left(\frac{dy}{du}\right)_u}{\left(\frac{dx}{du}\right)_u}$$

The gradient at midpoint for *u* = ½ is,

$$\left(\frac{dy}{du}\right)_{u=1/2} = \frac{3}{4}$$

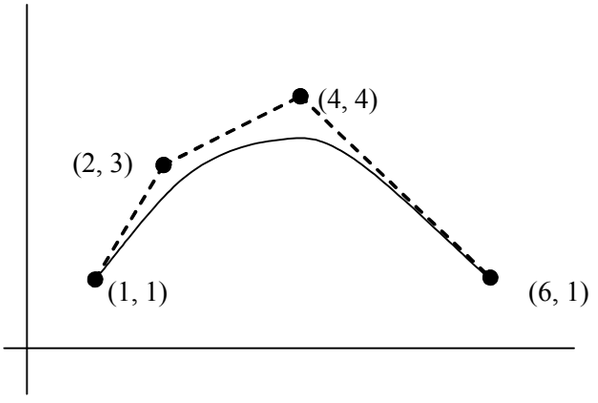
and

$$\left(\frac{dx}{du}\right)_{u=1/2} = \frac{21}{4}$$

NOTES

Then we have $\left(\frac{dy}{dx}\right)_{u=1/2} = \frac{1}{7}$ which is equal to the given gradient.

The cubic Bezier curve is given in Figure given below.



9.4 B-SPLINE CURVE AND SURFACES

Considering again $P(t)$ as the parametric position vector of any point along the curve, the general expression for a B-Spline curve of degree $d-1$ is given by,

$$P(t) = \sum_{i=0}^{n} P_i B_{i,d}(t), \qquad t_{\min} \le t \le t_{\max} \text{ and } 2 \le d \le n+1 \qquad ...(9.12)$$

where P_i are the input set of $n + 1$ control points and $B_{i,d}(t)$ are the $n + 1$ B-Spline blending functions defined by the Cox-deBoor recursion formulae as,

$$B_{i,d}(t) = \left\{\frac{t - t_i}{t_{i+d-1} - t_i}\right\} B_{i,d-1}(t) + \left\{\frac{t_{i+d} - t}{t_{i+d} - t_{i+1}}\right\} B_{i+1,d-1}(t)$$

where $B_{i,1}(t) = 1$, if $t_i \le t < t_{i+1}$ ...(9.13)

else $B_{i,1}(t) = 0$

For a cubic B-Spline, the degree $d-1 = 3$. So $d = 4$ and hence the number of control points $(n + 1)$ required is atleast 4 (from Equation (9.12), $n + 1 \le d$).

Suppose we have four control points P_0, P_1, P_2, P_3; we can obtain four cubic B-Spline blending functions $B_{0,4}(t)$, $B_{1,4}(t)$, $B_{2,4}(t)$ and $B_{3,4}(t)$ from Equation (9.13) such that the cubic B-Spline fitted to those points is given by,

$$P(t) = P_0 B_{0,4}(t) + P_1 B_{1,4}(t) + P_2 B_{2,4}(t) + P_3 B_{3,4}(t) \qquad ...(9.14)$$

NOTES

It follows from Equation (9.12), that unlike the Bezier and Hermite basis functions, the B-Spline functions are not defined over the entire range of parameter *t*. Instead each function is valid only in a limited portion of the total parameter range, $t_{\min} \to t_{\max}$. If this range $(t_{\max} - t_{\min})$ of *t* is divided into $n + d$, i.e., $3 + 4 = 7$ sub-intervals, then each of the four cubic blending functions of Equation (9.14) spans $d = 4$ sub-intervals. Selected eight $(n + d + 1 = 8)$ parameter values $(t_j, 0 \le j \le 7)$, satisfying the relation $t_j \le t_{j+1}$, marks the endpoints of the sub-intervals. The set of t_j, i.e., $[t_0 \ t_1 \ t_2 \ t_3 \ t_4 \ t_5 \ t_6 \ t_7]$ is referred to as a *knot vector*. We can choose uniform integer knot vector like [0 1 2 3 4 5 6 7] with evenly spaced knot values. This fixes the range of parameter *t* as from $t_{\min} = t_0 = 0$ upto $t_{\max} = t_7 = 7$.

Using the recurrence relation Equation (9.13) we obtain the expression for the four blending functions, by putting $d = 4$ and varying *i* from 0 through 3:

$$B_{0,4}(t) = \left(\frac{t - t_0}{t_3 - t_0}\right) B_{0,3}(t) + \left(\frac{t_4 - t}{t_4 - t_1}\right) B_{1,3}(t)$$

$$= \left(\frac{t - 0}{3 - 0}\right) B_{0,3}(t) + \left(\frac{4 - t}{3 - 0}\right) B_{1,3}(t)$$

$$= \left(\frac{t}{3}\right) B_{0,3}(t) + \left(\frac{4 - t}{3}\right) B_{1,3}(t) \qquad ...(9.15)$$

$$B_{1,4}(t) = \left(\frac{t - t_1}{t_4 - t_1}\right) B_{1,3}(t) + \left(\frac{t_5 - t}{t_5 - t_2}\right) B_{2,3}(t)$$

$$= \left(\frac{t - 1}{3}\right) B_{1,3}(t) + \left(\frac{5 - t}{3}\right) B_{2,3}(t) \qquad ...(9.16)$$

Similarly $$B_{2,4}(t) = \left(\frac{t - 2}{3}\right) B_{2,3}(t) + \left(\frac{6 - t}{3}\right) B_{3,3}(t) \qquad ...(9.17)$$

$$B_{3,4}(t) = \left(\frac{t - 3}{3}\right) B_{3,3}(t) + \left(\frac{7 - t}{3}\right) B_{4,3}(t) \qquad ...(9.18)$$

Again putting $d = 3$ and varying *i* from 0 through 4 in Equation (9.13) we get

$$\left.\begin{aligned}
B_{0,3}(t) &= \left(\frac{t}{2}\right) B_{0,2}(t) + \left(\frac{3 - t}{2}\right) B_{1,2}(t) \\
B_{1,3}(t) &= \left(\frac{t - 1}{2}\right) B_{1,2}(t) + \left(\frac{4 - t}{2}\right) B_{2,2}(t) \\
B_{2,3}(t) &= \left(\frac{t - 2}{2}\right) B_{2,2}(t) + \left(\frac{5 - t}{2}\right) B_{3,2}(t) \\
B_{3,3}(t) &= \left(\frac{t - 3}{2}\right) B_{3,2}(t) + \left(\frac{6 - t}{2}\right) B_{4,2}(t) \\
B_{4,3}(t) &= \left(\frac{t - 4}{2}\right) B_{4,2}(t) + \left(\frac{7 - t}{2}\right) B_{5,2}(t)
\end{aligned}\right\} \qquad ...(9.19)$$

NOTES

Finally putting $d = 2$ and varying i from 0 through 5 in Equation (9.13) we get,

$$\left.\begin{aligned}
B_{0,2}(t) &= (t)\ B_{0,1}(t) + (2-t)\ B_{1,1}(t) \\
B_{1,2}(t) &= (t-1)\ B_{1,1}(t) + (3-t)\ B_{2,1}(t) \\
B_{2,2}(t) &= (t-2)\ B_{2,1}(t) + (4-t)\ B_{3,1}(t) \\
B_{3,2}(t) &= (t-3)\ B_{3,1}(t) + (5-t)\ B_{4,1}(t) \\
B_{4,2}(t) &= (t-4)\ B_{4,1}(t) + (6-t)\ B_{5,1}(t) \\
B_{5,2}(t) &= (t-5)\ B_{5,1}(t) + (7-t)\ B_{6,1}(t)
\end{aligned}\right\} \quad \text{...(9.20)}$$

You need not have to work hard to find out the expressions for $B_{i,d}(t)$ if you use the similarity of the successive functions as a shortcut.

As from Equation (9.13) we accept,

$$B_{i,1}(t) = 1 \text{ if } t \le t < t_{i+1}$$
$$\text{else } B_{i,1}(t) = 0$$

Hence by varying i from 0 through 6 we get,

$$\left.\begin{aligned}
B_{0,1}(t) &= 1 \quad \text{if } 0 \le t < 1 \quad \text{else } B_{0,1}(t) = 0 \\
B_{1,1}(t) &= 1 \quad \text{if } 1 \le t < 2 \quad \text{else } B_{1,1}(t) = 0 \\
B_{2,1}(t) &= 1 \quad \text{if } 2 \le t < 3 \quad \text{else } B_{2,1}(t) = 0 \\
B_{3,1}(t) &= 1 \quad \text{if } 3 \le t < 4 \quad \text{else } B_{3,1}(t) = 0 \\
B_{4,1}(t) &= 1 \quad \text{if } 4 \le t < 5 \quad \text{else } B_{4,1}(t) = 0 \\
B_{5,1}(t) &= 1 \quad \text{if } 5 \le t < 6 \quad \text{else } B_{5,1}(t) = 0 \\
B_{6,1}(t) &= 1 \quad \text{if } 6 \le t < 7 \quad \text{else } B_{6,1}(t) = 0
\end{aligned}\right\} \quad \text{...(9.21)}$$

Applying Equations (9.21) in (9.20) and successively evaluating Equations (9.20), (9.19) and (9.15) through (9.18) we get specific expressions for the four blending functions (of Equation (9.14)) in different sub-intervals of t as,

$$B_{0,4}(t) = \begin{cases}
\left(\frac{1}{6}\right)t^3 & \text{for } 0 \le t < 1 \\
\left(\frac{1}{6}\right)t^2(2-t) + \left(\frac{1}{6}\right)t\,(t-1)(3-t) + \left(\frac{1}{6}\right)(t-1)^2(4-t) & \text{for } 1 \le t < 2 \\
\left(\frac{1}{6}\right)t(3-t)^2 + \left(\frac{1}{6}\right)(4-t)(t-1)(3-t) + \left(\frac{1}{6}\right)(t-2)(4-t)^2 & \text{for } 2 \le t < 3 \\
\left(\frac{1}{6}\right)(4-t)^3 & \text{for } 3 \le t < 4
\end{cases}$$

$$B_{1,4}(t) = \begin{cases}
\left(\frac{1}{6}\right)(t-1)^3 & \text{for } 1 \le t < 2 \\
\left(\frac{1}{6}\right)(t-1)^2(3-t) + \left(\frac{1}{6}\right)(t-1)(t-2)(4-t) + \left(\frac{1}{6}\right)(t-2)^2(5-t) & \text{for } 2 \le t < 3 \\
\left(\frac{1}{6}\right)(t-1)(4-t)^2 + \left(\frac{1}{6}\right)(5-t)(t-2)(4-t) + \left(\frac{1}{6}\right)(t-3)(5-t)^2 & \text{for } 3 \le t < 4 \\
\left(\frac{1}{6}\right)(5-t)^3 & \text{for } 4 \le t < 5
\end{cases}$$

NOTES

$$B_{2,4}(t)=\begin{cases}\left(\frac{1}{6}\right)(t-2)^3 & \text{for } 2\le t<3\\ \left(\frac{1}{6}\right)(t-2)^2(4-t)+\left(\frac{1}{6}\right)(t-2)(t-3)(5-t)+\left(\frac{1}{6}\right)(t-3)^2(6-t) & \text{for } 3\le t<4\\ \left(\frac{1}{6}\right)(t-2)(5-t)^2+\left(\frac{1}{6}\right)(6-t)(t-3)(5-t)+\left(\frac{1}{6}\right)(t-4)(6-t)^2 & \text{for } 4\le t<5\\ \left(\frac{1}{6}\right)(6-t)^3 & \text{for } 5\le t<6\end{cases}$$

$$B_{2,4}(t)=\begin{cases}\left(\frac{1}{6}\right)(t-3)^3 & \text{for } 3\le t<4\\ \left(\frac{1}{6}\right)(t-3)^2(5-t)+\left(\frac{1}{6}\right)(t-3)(t-4)(6-t)+\left(\frac{1}{6}\right)(t-4)^2(7-t) & \text{for } 4\le t<5\\ \left(\frac{1}{6}\right)(t-3)(6-t)^2+\left(\frac{1}{6}\right)(7-t)(t-4)(6-t)+\left(\frac{1}{6}\right)(t-5)(7-t)^2 & \text{for } 5\le t<6\\ \left(\frac{1}{6}\right)(7-t)^3 & \text{for } 6\le t<7\end{cases}$$

The hard work involved in deriving the above functions can be easily avoided by exploiting the inherent relations among them. Can you find the relation?

Substitute $t-1$ for t in the expression of $B_{0,4}(t)$ and right shift the parameter domain by 1. What you get are the expressions for $B_{1,4}(t)$. Similarly $B_{2,4}(t)$ and $B_{3,4}(t)$ are obtained by successively shifting $B_{1,4}(t)$ one position (parametrically) to the right.

Now look at Figure 9.7 where we plot the cubic B-Spline blending functions against parameter t.

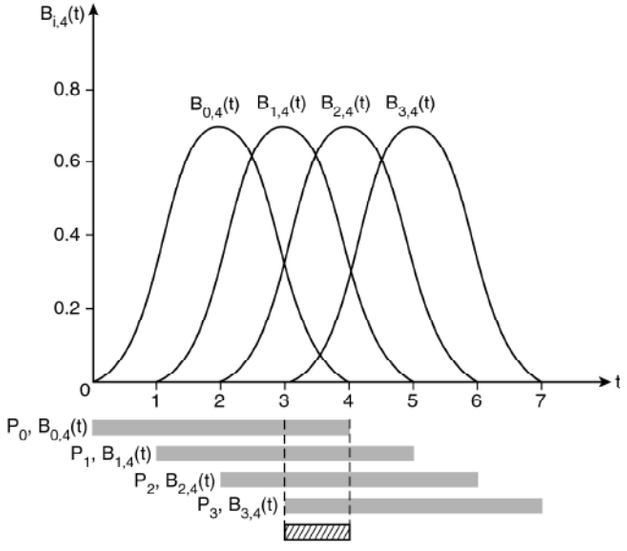


Fig. 9.7 *Periodic blending functions of a cubic B-Spline for n = 3 and a uniform integer knot vector. The ▬ bars show the parameter span of the blending functions as well as the influence range of corresponding control points. The ▨ bar represents the only common span of all the functions from t = 3 to t = 4*

NOTES

Its obvious that all the blending functions have the same shape. Each successive blending function is simply a shifted or translated version of the previous function.[3]

Also note that each of these blending function curves consist of four cubic segments with inherent continuity (C^2 continuity) at the intermediate knot values (refer Figure 9.8).

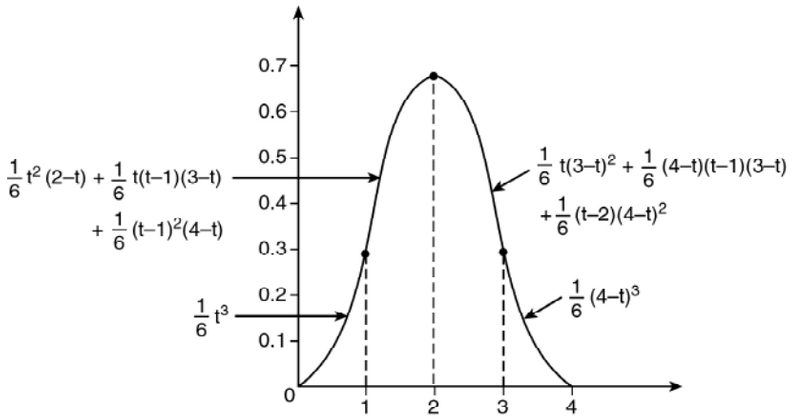


Fig. 9.8 *Plot of $B_{0,4}(t)$; C^2 continuity exists between the adjacent sections at the marked points corresponding to knot values 1, 2 and 3*

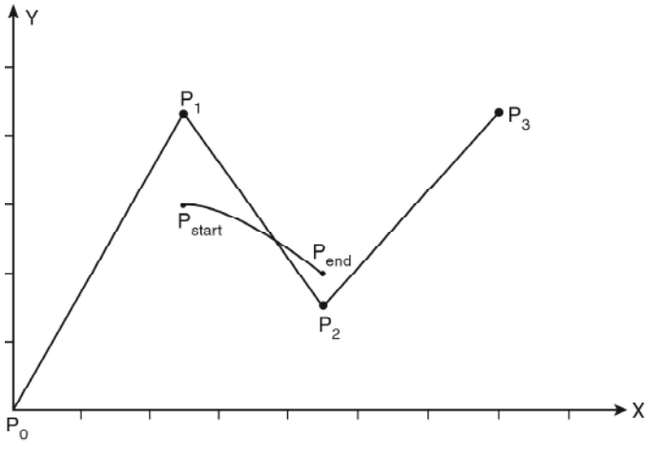


Fig. 9.9 *The defining polygon $P_0 - P_1 - P_2 - P_3$ and the cubic B-spline $P_{start} - P_{end}$ fitted to it. P_{start} corresponds to knot value $t_3 = 3$ while P_{end} corresponds to $t_4 = 4$*

Now it will be interesting to note that though the blending functions $B_{i,4}(t)$ are specifically defined over the entire range of $t(0 \le t \le 7)$, the resulting cubic B-Spline curve doesnot span the full range of t. It is limited to a reduced parameter range, $t_{d-1} = 3$ to $t_{n+1} = 4$, over which the sum of all the four blending functions is equal to unity, everywhere.

$$\text{i.e. } \sum_{i=0}^{3} B_{i,4}(t) = 1 \text{ for } 3 \le t \le 4$$

Thus, the effective expression for a cubic B-Spline defined by four control points P_0, P_1, P_2, P_3 and uniform knot vector [0 1 2 3 4 5 6 7] is

$$P(t) = B_{0,4}(t)P_0 + B_{1,4}(t)\,P_1 + B_{2,4}(t)\,P_2 + B_{3,4}(t)\,P_3 \text{ where } 3 \le t \le 4$$

i.e.,
$$P(t) = \left\{\frac{1}{6}(4-t)^3\right\}P_0 + \left\{\frac{1}{6}(t-1)(4-t)^2 + \frac{1}{6}(5-t)(t-2)(4-t) + \frac{1}{6}(t-3)(5-t)^2\right\}P_1$$
$$+ \left\{\frac{1}{6}(t-2)^2(4-t) + \frac{1}{6}(t-2)(t-3)(5-t) + \frac{1}{6}(t-3)^2(6-t)\right\}P_2$$
$$+ \left\{\frac{1}{6}(t-3)^3\right\}P_3 \text{ where } 3 \le t \le 4$$

...(9.22)

Now from the above expression we can find out the boundary values of the Spline curve as,

$$\left.\begin{aligned} P_{start} &= P(t=3) = \frac{1}{6}(P_0 + 4P_1 + P_2) \\ P_{end} &= P(t=4) = \frac{1}{6}(P_1 + 4P_2 + P_3) \end{aligned}\right\} \text{ end points} \qquad \ldots(9.23)$$

$$\left.\begin{aligned} P'_{start} &= P'(3) = \frac{1}{2}(P_2 - P_0) \\ P'_{end} &= P'(4) = \frac{1}{2}\quad (P_3 - P_1) \end{aligned}\right\} \text{end point tangents (found by evaluating parametric first derivatives } d()/dt \text{ at the end points)}$$

...(9.24)

$$\left.\begin{aligned} P''_{start} &= P''(3) = P_0 - 2P_1 + P_2 \\ P''_{end} &= P''(4) = P_1 - 2P_2 + P_3 \end{aligned}\right\} \text{ rate of change of end point tangents (found by evaluating parametric second derivatives } d^2()/dt^2 \text{ at the end points)}$$

...(9.25)

Unlike the cubic Bezier and Hermite splines, notice that the first and last point on the cubic B-Spline curve do not correspond to the first and last control points of the defining polygon. Nor are the tangent vectors (slope) at the first and last point the same as the first and last spans (slopes) of the defining polygon.

However the curve can be made to pass through the first and last control points by defining the first three and last three consecutive control points identical and coincident at the ends of the defining polygon. Without changing the four control point positions (refer Figure 9.9), we could add four more control points (two at each end) on the defining polygon. Thus the number of control points (n + 1) will be eight but the shape of the defining polygon remains unchanged (refer Figure 9.10).

NOTES

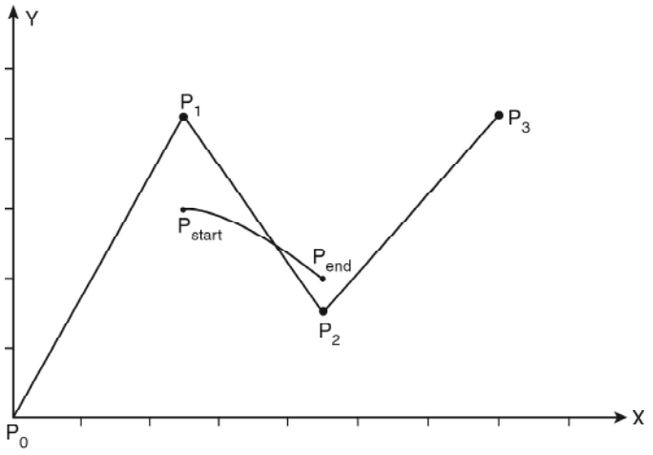


Fig. 9.10 *Effect of triple coincidental vertices at the ends of the defining polygon; for the fitted cubic B-Spline curve,* $P_{start} = P_0 = P_1 = P_2$ *and* $P_{end} = P_5 = P_6 = P_7$

As it can be proved that for a cubic ($d = 4$) periodic B-Spline fitted to $n + 1$ control points $P_0, P_1, P_2, \ldots, P_{n-2}, P_{n-1}, P_n$, the start and end points are

$$\left.\begin{aligned} P_{start} &= \frac{1}{6}(P_0 + 4P_1 + P_2) \\ P_{end} &= \frac{1}{6}(P_{n-2} + 4P_{n-1} + P_n) \end{aligned}\right\}$$

Hence for the above case with triple vertices (control points) at the ends,

$$P_{start} = \frac{1}{6}(P_0 + 4P_0 + P_0) = P_0 \quad [\because\ P_0 = P_1 = P_2]$$

$$P_{end} = \frac{1}{6}(P_5 + 4P_6 + P_7) = \frac{1}{6}(P_7 + 4P_7 + P_7) = P_7 \quad [\because\ P_5 = P_6 = P_7]$$

i.e., the curve starts and ends at the first and last control points.

Note that increasing the number of control points doesnot affect the degree of the curve. Though the shape and length varies we basically get a cubic B-Spline with four as well as eight control points. However if we want we can easily fit a (non-cubic) B-Spline curve of degree two, or, four or maximum upto degree seven (i.e., n) to the same set of eight control points. Thus the *B-Spline concept in general allows the degree of the resulting curve to be changed (with certain limitation as* $2 \le d \le n + 1$*) without changing the number of defining control points.* This is a major plus point compared to the Bezier curve which does not display such flexibility. The degree of a Bezier curve is so intrinsically associated with the number of defining control points (the degree being always one less than the number of control points) that to reduce the degree of a Bezier curve, the only way is to reduce the number of control points. Conversely the only way to increase the degree of a Bezier curve is to increase the number of control points. Thus even if we want we cannot directly fit a cubic Bezier curve to a set of

eight control points or more, we are always restricted to fit only a 7th degree Bezier curve to eight defining control points or a 8th degree Bezier section to nine control points, and so on.

Making use of this advantage we can fit a long single piece of periodic cubic B-Spline approximating several given data points. Considering the proven fact that *if any three consecutive control points are identical, the curve passes through that coordinate position,* we can also force the cubic B-Spline to pass through (or interpolate) a given data point by defining three consecutive control points coincident at the desired data position.

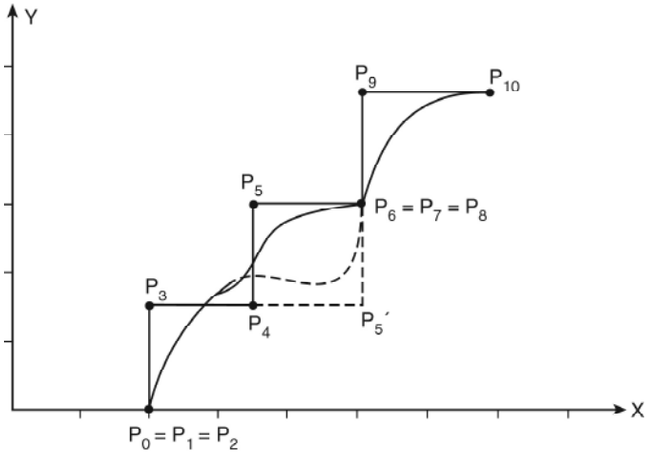


Fig 9.11 *Periodic cubic B-Spline segment fitted to 11 control points. The curve is made to pass through P_0, P_{10} and P_6 by specifying triple control points at those positions. The shape of the curve only around point P_5 changes as it is moved to P_5'*

Another very useful advantage of B-Spline curve in general is its flexibility to allow a local change of shape without altering the entire curve shape. To explain this *'local control'* property let us consider the periodic cubic B-Spline fitted to 11 control points, $P_0, P_1,...,P_{10}$ as shown in Figure 9.11.

So, for this curve ($d = 4$) 11 blending functions. $B_{0,4}(t), B_{1,4}(t),..., B_{10,4}(t)$ are used to blend the corresponding control points, i.e., $P_0, P_1,...,P_{10}$ respectively to yield the curve shape; each control point thus affects the shape of the curve only over a range of parameter values where its associated basis function is non-zero. For a choice of uniform integer knot vector (0 to 15) dividing the entire parameter range in 14 intervals, any change in position of the P_0 control point only effects the shape of the curve upto $t = 4$. Similarly changing the position of the P_5 control point affects the curve shape in the region $t = 5$ to $t = 9$. The dotted curve section in Figure 9.11 represents this localized change as P_5 is moved to P_5', the remaining portion of the curve (i.e., for $3 \leq t \leq 5$ and $9 \leq t \leq 11$) remains unchanged.

This is in contrast to the Bezier Splines, which do allow for local control of the curve shape. If we reposition any one of the control points, the entire curve will be

affected. The reason behind is, all the Bezier blending functions are non-zero for all parameter values over the entire curve, except at the start point and endpoint (i.e. for $0 < t < 1$).

NOTES

Now coming back to the discussion of cubic B-Spline with four defining control points, we can express Equation (9.22) in a matrix form similar to those for Hermite splines and Bezier splines. For this we have to reparameterize the blending functions so that parameter t within interval 3 to 4 is mapped to the interval 0 to 1. If the equivalent parameter in 0 to 1 interval is termed u, then

$$\frac{t-3}{4-3} = \frac{u-0}{1-0} \Rightarrow t-3 = u$$
$$\Rightarrow t = u+3$$

Now replace $u + 3$ for t in Equation (9.22).

On simplifying we get

$$P(t) = \left\{\frac{1}{6}(1-u)^3\right\}P_0 + \left\{\frac{1}{6}(3u^3 - 6u^2 + 4)\right\}P_1$$
$$+ \left\{\frac{1}{6}(-3u^3 + 3u^2 + 3u + 1)\right\}P_2 + \left\{\frac{1}{6}(u)^3\right\}P_3 \text{ where } 0 \le u \le 1$$

In matrix formulation the above expression becomes,

$$P(t) = [u^3 \ u^2 \ u \ 1]\frac{1}{6}\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix}\begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix}$$
$$= [U][M_{BS}][B]$$

Thus the characteristic basis matrix for a periodic cubic B-Spline with 4 control points P_0, P_1, P_2, P_3 is,

$$[M_{BS}] = \frac{1}{6}\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix}$$

9.5 BASIC ILLUMINATION MODELS

When we view an opaque non-luminous object, we observe the light reflected from the surfaces of the object. The total reflected light is equal to the sum of the contributions from light sources and other reflecting surfaces in the scene. This is illustrated in Figure 9.12. Thus, if adjacent models are illuminated, a surface that is not directly exposed to a light source may still be visible. Sometimes, reflecting surfaces, such as the walls of a room, are termed light-reflecting sources. We use the term light source to mean an object that is emitting radiant energy in the form of light waves, such as an electric bulb or the sun.

NOTES

In general, a luminous object can be both a light source and a light reflector. For example, a plastic globe with a light bulb inside both emits and reflects light from the surface of the globe. Emitted light from the globe may then illuminate other objects in the surrounding area.

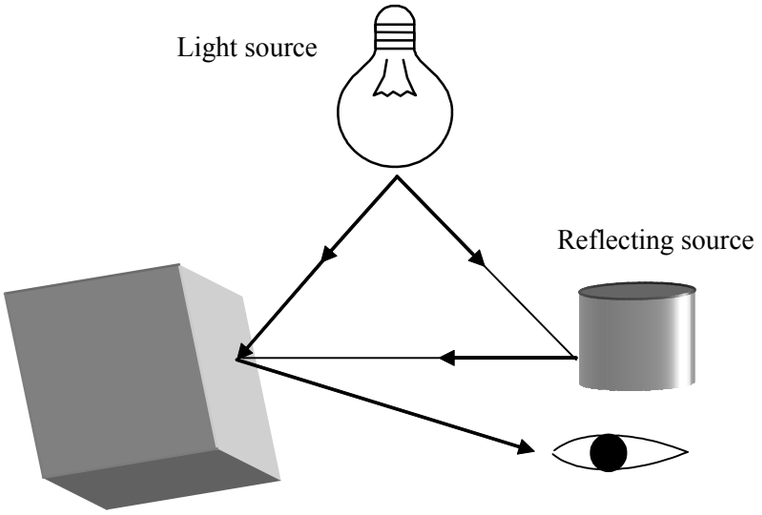


Fig. 9.12 *Light Obtained from an Opaque Non-Luminous Surface is a Combination of Reflected Light From a Light Source and Reflections of Light Reflections from Other Surfaces*

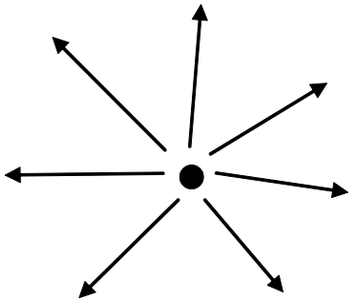


Fig. 9.13 *Diverging Ray Paths from a Point Light Source*

The simplest model for a light emitter is a point source. Rays from the source then follow radial diverging paths from the source position, as shown in Figure 9.13. This light-source model is an approximation for sources whose dimensions are small compared to the size of the objects in the scene. The light sources, such as the sun, that are sufficiently far from the scene can be accurately modelled as point sources.

A nearby source, such as the long fluorescent light in as shown in Figure 9.14, is more accurately modelled as a distributed light source. In this case, the illumination effects cannot be approximated realistically with a point source, because the area of the source is not small compared to the surfaces in the scene. An ideal

NOTES

model for the distributed source is one that considers the accumulated illumination effects of the points over the surface of the source. When light is incident on an opaque surface, a portion of it is reflected and the rest is absorbed.

The amount of incident light reflected by a surface of an object depends upon its material. Glossy materials reflect maximum amount of incident light and rough surfaces absorb maximum amount of incident light. Similarly, for an illuminated transparent surface some of the incident light will be reflected and some will be transmitted through the material. Surfaces, which are rough (or grainy) tend to scatter the reflected light in all directions.

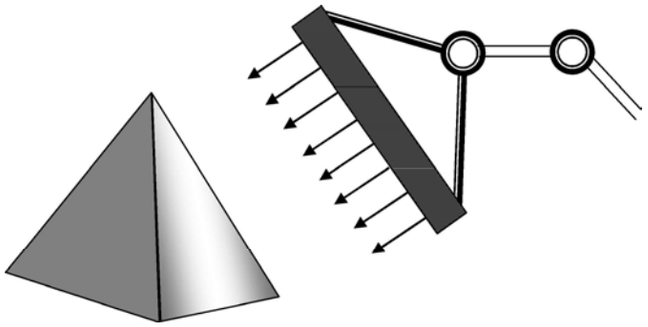


Fig. 9.14 *An Object Illuminated with a Distributed Light Source*

This scattered light is also called diffuse reflection. A very rough surface produces primarily diffuse reflections, so that the surface looks equally bright from all viewing directions. Figure 9.15 illustrates diffuse light scattering from a surface. What we call the colour of an object is the colour of the diffuse reflection of the incident light. For example, a blue object illuminated by a white light source, reflects the blue component of the white light and totally absorbs all other components. If the blue object is viewed under a red light, it appears black since the entire incident light is absorbed by the object. Additionally, to diffuse reflection, light sources create highlights, or bright spots, called specular reflection. This highlighting effect is more evident on shiny surfaces than on dull surfaces. An illustration of specular reflection is shown in Figure 9.16.

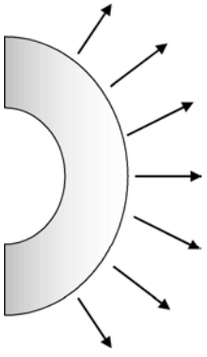


Fig. 9.15 *Illustration of Diffus Reflections from a Surface*

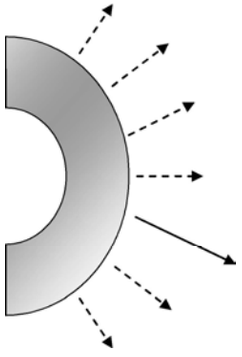


Fig. 9.16 *Specular Reflection Superimposed on Diffuse Reflection Vectors*

NOTES

Basics of Objects Illumination

Empirical models provide simple and fast methods for calculating surface intensity at a given point, and they produce reasonably good results for most scenes. Lighting calculations are based on the following:

- Optical properties of surfaces
- Light-source specifications
- Background lighting conditions

Optical parameters can be used to set surface properties of an object, such as matte, glossy, transparent and opaque. This operation controls the amount of reflection and absorption of incident light on an object. All kinds of light sources are considered to be point sources, specified with a coordinate position and an intensity value (i.e., colour, brightness and contrast).

Ambient Light

An object surface which is not exposed directly to a light source is visible due to the light reflected form the nearby objects that are illuminated. In the basic illumination model, we can set a general level of brightness for a scene. This is a simple way to model the combination of light reflections from various surfaces to produce a uniform illumination called the background light, or ambient light. The background light has no directional or spatial characteristics. The amount of background light incident on each object is a constant for all surfaces and over all directions. We can set the level for the ambient light in a scene with parameter A_L, and each surface is then illuminated with this constant value. The resulting reflected light is a constant for each surface, independent of the spatial orientation and the viewing direction of the surface. But the intensity of the reflected light for each surface depends on the optical properties of the surface, that is, how much of the incident energy is to be reflected and how much is to be absorbed.

Diffuse Reflection

The background-light reflection is an approximation of global diffuse lighting effects. Diffuse reflections are constant over each surface in a scene, independent of the viewing direction. The fractional amount of the incident light diffusely reflected, can be set for each surface with parameter I_d. It is also called diffuse reflectivity, or diffuse-reflection coefficient. Parameter I_d is assigned a constant value in the interval 0 to 1, according to the reflecting properties we want the surface to have. If we want a highly reflective surface, we set the value of I_d near to 1. This produces a bright surface with the intensity of the reflected light about that of the incident light. To simulate a surface that absorbs most of the incident light, we set the reflectivity to a value that is near to 0. The parameter I_d is a function of surface colour, but for

NOTES

the time being we assume I_d as a constant.

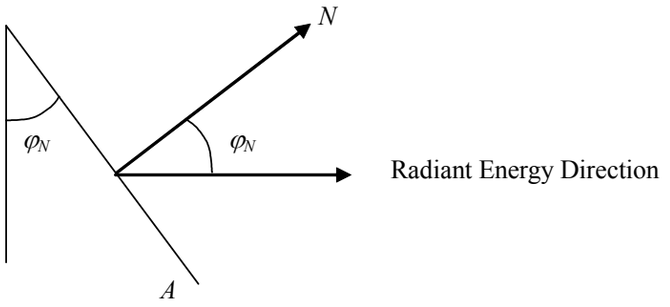


Fig. 9.17 *Radiant Energy from a Surface area A in Direction j_N Relative to the Surface Normal Direction*

If a surface is exposed only to an ambient light, we can express the intensity of the diffuse reflection at any point on the surface as follows:

$$\text{Intensity of diffused reflection} = I_d.A_L \qquad ...(9.23)$$

Since ambient light produces a flat uninteresting shade for each surface, scenes are rarely rendered with ambient light alone. At least one light source is included in a scene, often as a point source at the viewing position of the object. We can model the diffuse reflections of illumination from a point source in the same way. That is, we assume that the diffuse reflections from the surface are scatted with equal intensity in all directions, independent of the viewing direction. Such surfaces are sometimes referred to as ideal diffuse reflectors. They are also called Lambertian reflectors, since radiated light energy from any point on the surface is directed by Lambert's cosine rule. This rule states that the radiant energy from a small surface area '*A*' in a direction φ_N relative to the surface normal is proportional to $\cos\varphi_{N.}$, as shown in Figure 9.17. The light intensity, though, depends on the radiant energy per projected area perpendicular to direction φ_N which is $A \times \cos\varphi_N$. Thus, for Lambertian reflection, the intensity of light is the same over all viewing directions of an object.

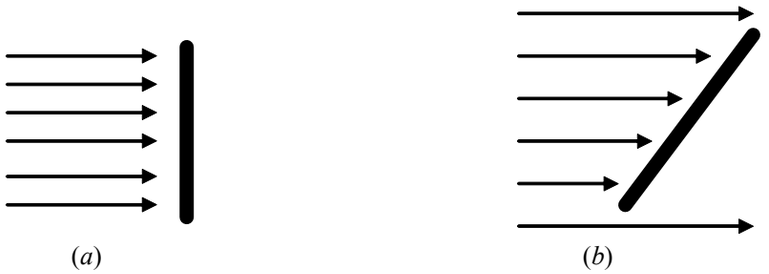


Fig. 9.18 *A Surface Perpendicular to the Direction of the Incident Light in (a) is more Illuminated than an Equal Sized Surface at a Tilted Surface in (b) to the Incoming Light Direction*

Even though a perfect diffuse reflector scatters light equally in all directions, the brightness of the surface depends on the orientation of the surface relative to the light source. A surface which is oriented perpendicular to the direction of the incident light appears brighter than that of the surface tilted at an angle to the direction of the incoming light. This can easily be seen by holding a white sheet of paper or smooth cardboard parallel to a nearby window and slowly rotating the sheet away from the window direction. As the angle between the surface normal and the incoming light direction increases, the incident light falling on the surface decreases. Figure 9.18 illustrates this.

This figure shows a beam of light incident on two equal area plane surface patches with different spatial orientations relative to the incident light direction from a distant source (parallel incoming rays). If we denote the angle of incidence between the incoming light direction and the surface normal as Φ, illustrated in Figure 9.19, then the projected area of a surface patch perpendicular to the light direction is proportional to $\cos\Phi$. Thus, the amount of illumination (or the number of incident light rays cutting across the projected surface patch) depends on $\cos\Phi$. If the incoming light from the source is perpendicular to the surface at a particular point, that point is fully illuminated.

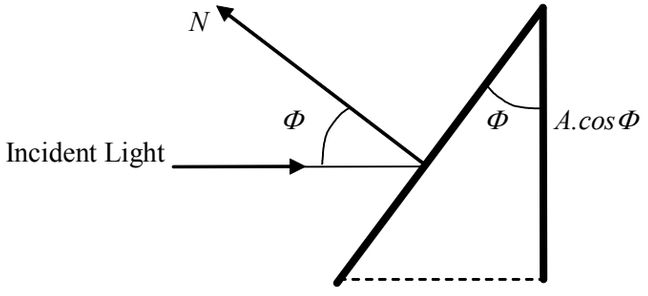


Fig. 9.19 *An Illuminated Area projected Perpendicular to the Path of the Incoming Light*

As the angle of illumination moves away from the surface normal, the brightness of the point decreases continuously. If I_L is the intensity of the point light source, then the diffuse reflection equation for a point on the surface can be written as follows:

$$I_{L(Diff)} = I_d \times I_L \cos\Phi \qquad ...(9.24)$$

A surface is illuminated by a point source only if the angle of incidence is in the range 0°–90° ($\cos\Phi$ is in the interval from 0–1). When $\cos\Phi$ is negative, the light source is 'behind' the surface.

NOTES

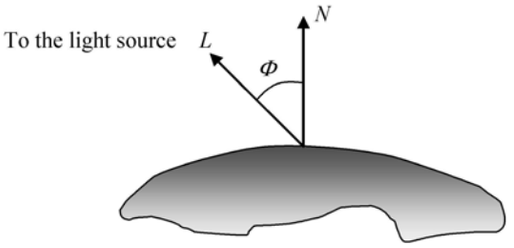


Fig. 9.20 *Angle of Incidence between the Unit Light-Source Direction Vector L and the Unit Surface Normal N*

If N is the unit normal vector to a surface and L is the unit direction vector to the point light source from a position on the surface, then cos $\Phi = N.L$ and the diffuse reflection equation for a single point-source illumination is as follows:

$$I_{I,diff} = k_d I_I (N.L) \qquad ...(9.25)$$

The reflections for point-source illumination may be calculated in world coordinates or viewing coordinates before shearing and perspective transformations are applied. These transformations may change the orientation of normal vectors so that they are no longer perpendicular to the surfaces they represent. Figure 9.21 illustrates the application of equation 9.25 to positions over the surface of a sphere, using various values of parameter k_d between 0 and 1. Each projected pixel position for the surface is assigned intensity as calculated by the diffuse reflection equation for a point light source. The rendering in this figure illustrates single point-source lighting with no other lighting effects. This is what we expect to see if we shine a small light on the object in a completely dark room. In case of general scenes, however, we expect some background lighting effects in addition to the illumination effects produced by a direct light source. We can combine the point and ambient source intensity calculations to obtain an expression for the total diffuse reflection. Additionally, many graphics packages introduce an ambient-reflection coefficient k_a to modify the ambient light intensity I_a for each surface. This simply provides us with an additional parameter to adjust the light conditions in a scene. Using parameter k_a we can write the total diffuse reflection equation as follows:

$$I_{diff} = k_a I_a + k_d I_I (N.L) \qquad ....(9.26)$$

where both k_a and k_d depend on surface material properties and are assigned values in the range from 0 to 1.

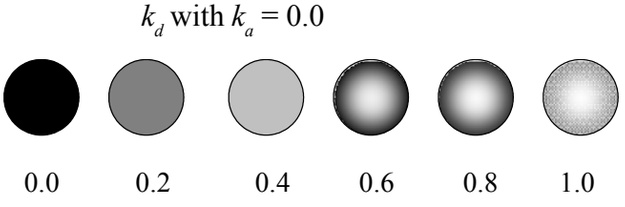


Fig. 9.21 *Diffuse Reflections from a Spherical Surface Illuminated by a Point Light Source for Values of the Diffuse Reflectivity Coefficient in the interval $0 \le k_d \le l$*

Illumination Models

NOTES

Illumination models produce a linear range of intensities. The RGB colour (0.125, 0.125 and 0.125) obtained from a lighting model represents one-half the intensity of the colour (0.25, 0.25 and 0.25). Usually, these calculated intensities are then stored in an image file as integer values, with one byte for each of the three RGB components. This intensity file is also linear, so that a pixel with the value (32, 32 and 32) has half the intensity of a pixel with the value (64, 64 and 64). A video monitor is a nonlinear device. If we set the voltages for the electron gun proportional to the linear pixel values, the displayed intensities are shifted according to the monitor response curve. To correct monitor nonlinearities, graphics system uses a video lookup table that adjusts the linear pixel values. The monitor response curve is described by the exponential function,

$$I = aV^{\gamma}$$

...(9.27)

Parameter I is the displayed intensity, and parameter V is the input voltage. Values for parameters a and g depend up on the characteristics of the monitor used in the graphics system. Thus, if we want to display a particular intensity value I, the correct voltage value to produce this intensity is,

$$V = \frac{(I)^{1/\gamma}}{a}$$

...(9.28)

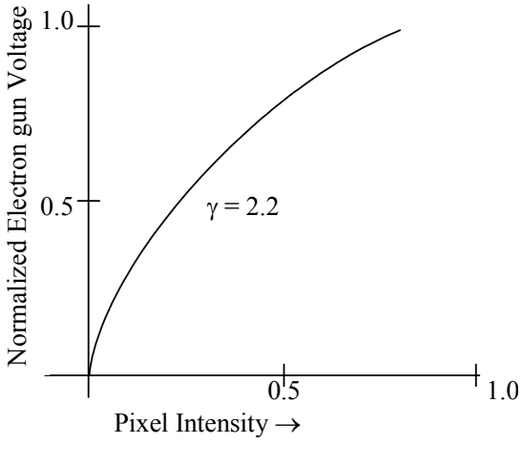


Fig. 9.22 *A Video Lookup Correction Curve for Mapping Pixel Intensities to Electron-Gun Voltages using Gamma Correction with $\gamma = 2.2$*

This calculation is referred to as gamma correction of intensity. Monitor gamma values are typically between 2.0 and 3.0. The National Television System Committee (NTSC) signal standard has $\gamma = 2.2$. Figure 9.22 shows a gamma correction curve using the NTSC gamma value. Equation 9.30 can be used to set

NOTES

up the video lookup table that converts integer pixel values in the image file to values that control the electron-gun voltages. We can combine gamma correction with logarithmic intensity mapping to produce a lookup table that contains both conversions. If I is an input intensity value from an illumination model, we first locate the nearest intensity 4 from a table of values created with equations 9.28 or 9.29. Alternatively, we can determine the level number for this intensity value with the calculation. Then we compute the intensity value at this level using equation 9.29.

$$k = round\left(\log_{\gamma} \frac{I}{I_0}\right) \qquad ...(9.29)$$

Once we have the intensity value I_k, we can calculate the electron-gun voltage,

$$V_k = \left(\frac{I_k}{a}\right)^{1/\gamma} \qquad ...(9.30)$$

The values V_k can then be placed in the lookup tables. Values for k are stored in the frame-buffer pixel positions. If a particular system has no lookup table, computed values for V_k can be stored in the frame buffer, directly. The combined conversion to a logarithmic intensity scale followed by calculation of the V, using equation 9.30 is also sometimes referred to as gamma correction.

We cannot combine two intensity-conversion processes, if the video amplifiers of a monitor are designed to convert the linear input pixel values to electron-gun voltages. Here, gamma correction is built into the hardware, and the logarithmic values 1, must be pre-computed and stored in the frame buffer (or the colour table).

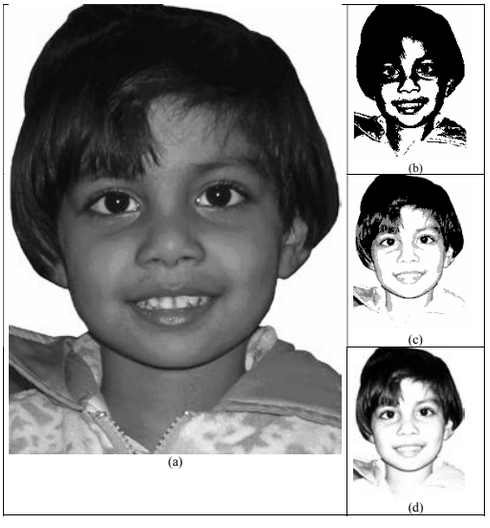


Fig. 9.23 *A Continuous-Tune Photograph (a) printed with (b) Two Intensive Levels (c) Four Intensity Levels (d) Eight Intensity Levels*

NOTES

Displaying Continuous-Tone Images

Generally, high quality computer graphics systems provide 256 intensity levels for each colour component, but acceptable displays can be obtained for many applications with fewer levels as well. A four-level system provides minimum shading capability for continuous-tone images, while photorealistic images can be generated on systems that are capable of 32–256 intensity levels per pixel. Figure 9.23(a) shows a continuous-tone photograph. When a small number of intensity levels are used to reproduce a continuous-tone image, the borders between the different intensity regions are clearly visible. In the two-level reproduction, the features of the photograph are just hardly identifiable. Using four intensity levels, we start to identify the original shading patterns, but the contouring effects are obvious. With eight intensity levels, contouring effects are still obvious, but we begin to have a better indication of the original shading. At 16 or more intensity levels, contouring effects diminish and the reproductions are very close to the original. The reproductions of continuous-tone images using more than 32 intensity levels show only very subtle differences from the original.

9.6 POLYGON RENDERING METHODS

In this section, we discuss the applications of an illumination model to the rendering of graphics objects which are formed with polygon surfaces. Objects are usually polygon-mesh approximations of curved-surface objects, but they may also be polyhedra which are not curved-surface approximations. The scan-line algorithm typically applies a lighting model to obtain polygon surface rendering in one of two possible ways. Each polygon can be rendered by applying single intensity, or the intensity that can be obtained at each point of the surface using an interpolation scheme.

Constant-Intensity Shading

A fast and simple method for rendering an object with polygon surfaces is constant-intensity shading, also known as flat shading. This method calculates a single intensity for each polygon. All specific points over the surface of the polygon are then highlighted with the same intensity value. Constant shading can be useful for quickly displaying the general appearance of a curved surface.

In general, flat shading of polygon facets provides an accurate rendering for an object if all of the following assumptions are valid:

(i) All light sources illuminating the object are sufficiently far from the surface of the object so that $N.L$ and the attenuation function are constant over the surface.

(ii) The viewing position is sufficiently far from the surface so that $V.R$ is constant over the surface.

NOTES

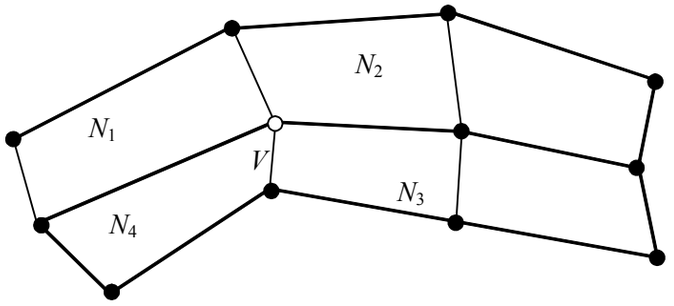


Fig. 9.24 *The Normal Vector at Vertex V is calculated as the Average of the Surface Normals for each Polygon Sharing that Vertex*

The object in Figure 9.24 is a polyhedron and is not an approximation of an object with a curved surface. Even if all of these conditions are not true, we can still reasonably approximate surface-lighting effects using small polygon facets with flat shading and calculate the intensity for each facet, say, at the centre of the polygon.

The Gouraud Shading

This scheme of intensity-interpolation was developed by Gouraud and is generally referred to as Gouraud shading. It can render a polygon surface by linearly interpolating intensity values across the surface. The intensity values for each polygon can be matched with the values of adjacent polygons along the common edges. This eliminates the intensity discontinuities that can occur in flat shading. Each polygon surface can be rendered with Gouraud shading with the help of the following calculations:

$$N_V = \frac{\sum_{k=1}^{n} N_K}{\left|\sum_{k=1}^{n} N_k\right|} \qquad ...(9.31)$$

For this purpose, the average unit normal vector is calculated at each polygon vertex. Then an illumination model is applied to each vertex to calculate the vertex intensity. Now the vertex intensities are linearly interpolated over the surface of the polygon. At each polygon vertex, we determine a normal vector by determining the average of the surface normals of all polygons sharing that vertex, as illustrated in Figure 9.24. Thus, for any vertex position denoted by V, the unit vertex normal can be obtained. Once the vertex normals are determined, the intensity at the vertices can be determined from a lighting model. Figure 9.25 demonstrates the next step that interpolates intensities along the polygon edges. The intensity for each scan line, at the intersection of the scan line with a polygon edge is linearly interpolated from the intensities at the edge endpoints. As seen in Figure 9.25, the

polygon edge with endpoint vertices at positions A and B is intersected by the scan line at point D. A fast method to obtain the intensity at point D is to interpolate between intensities I_A and I_B using only the vertical displacement of the scan line.

$$I_D = \frac{y_D - y_B}{y_A - y_B} I_A + \frac{y_A - y_D}{y_A - y_B} I_B$$

...(9.32)

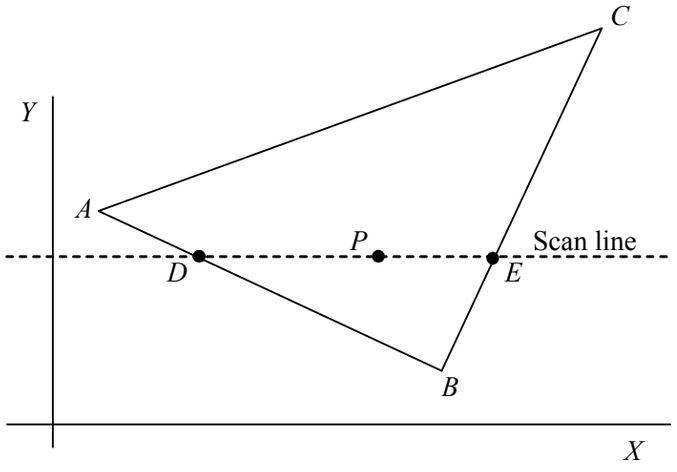


Fig. 9.25 *The Intensity at Point D is Linearly Interpolated from the Intensities at Vertices A and B.*

The intensity at point E is interpolated linearly, from intensities at vertices B and C. An interior point P is then assigned an intensity value that is interpolated linearly from intensities at positions D and E. In the same way, the intensity at the right intersection of this scan line (point E) is interpolated from intensity values at vertices B and C. Once these bounding intensities are established for a scan line, an interior point (such as point P in Figure 9.25) is interpolated from the bounding intensities at points D and E as follows:

$$I_P = \frac{x_E - x_P}{x_E - x_D} I_D + \frac{x_P - x_D}{x_E - x_D} I_E$$

...(9.33)

Incremental calculations are used to obtain successive edge intensity values between scan lines and to obtain successive intensities along a scan line. As shown in Figure 9.26, if the intensity at edge position (x, y) is interpolated as follows:

$$I = \frac{y - y_B}{y_A - y_B} I_A + \frac{y_A - y}{y_A - y_B} I_B$$

...(9.34)

NOTES

then we can obtain the intensity along this edge for the next scan line, $y - 1$, as follows:

$$I' = I + \frac{I_B - I_A}{y_A - y_B} \qquad ...(9.35)$$

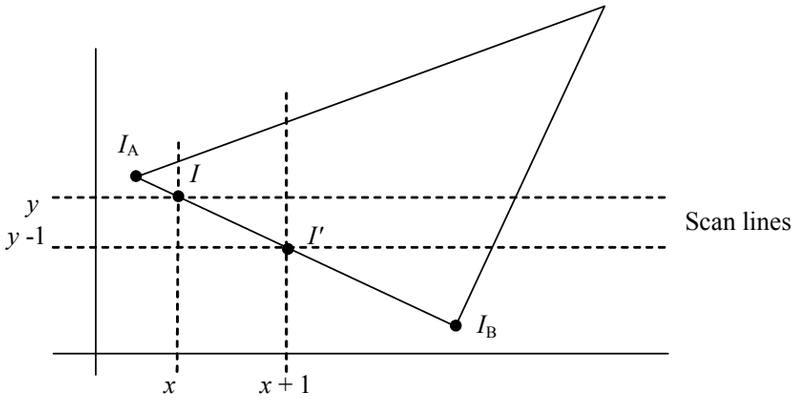


Fig. 9.26 *Incremental Interpolation of Intensity Values along a Polygon Edge for Successive Scan Lines*

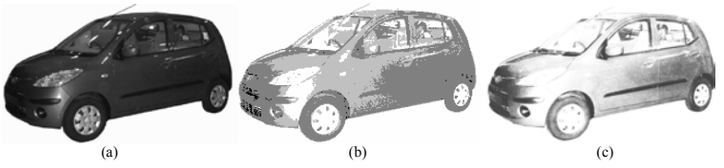


Fig. 9.27 *(a) A Polygon Mesh Approximation of an Object (b) Object is rendered with Flat Shading (c) Object is rendered with Gouraud Shading*

Similar calculations can be used to obtain intensities at successive horizontal pixel positions along each scan line. When surfaces are to be rendered in colour, the intensity of each colour component is calculated at the vertices. Gouraud shading can also be combined with a hidden-surface algorithm to fill in the visible polygons along each scan line. Figure 9.27 illustrates an object that has been shaded using the Gouraud method. Gouraud shading is used to remove the intensity discontinuities associated with the constant-shading model, but it has some other drawbacks. Sometimes highlights on the surface are displayed with anomalous shapes, and the linear intensity interpolation can produce bright or dark intensity streaks (also called Mach bands) to appear on the surface. Such effects can be reduced by dividing the surface into a greater number of polygon faces or by using other methods, such as Phong shading, that require more calculations.

936.1 Phong Shading

A more exact method for rendering a polygon surface interpolates the normal vectors, and then applies an illumination model to each surface point. This method

was developed by Phong Bui Tuong. Therefore it is called Phong shading, or normal vector interpolation shading. It generally displays more realistic highlights on the surface of an object and reduces the Mach-band effect that was present in the earlier shading schemes. A polygon surface rendered using Phong shading carries out the following steps:

(i) Calculation of the average unit normal vector at each polygon vertex

(ii) Interpolation of the vertex normals linearly, over the surface of the polygon

(iii) Application of an illumination model beside each scan line to determine the projected pixel intensities for the surface points

Interpolation of surface normals along a polygon edge between two vertices is illustrated by the Figure 9.28. The normal vector N for the scan-line intersection point along the edge between vertices A and B can be determined by vertically interpolating between edge endpoint normals as follows:

$$N = \frac{y - y_2}{y_1 - y_2} N_1 + \frac{y_1 - y}{y_1 - y_2} N_2 \qquad ...(9.36)$$

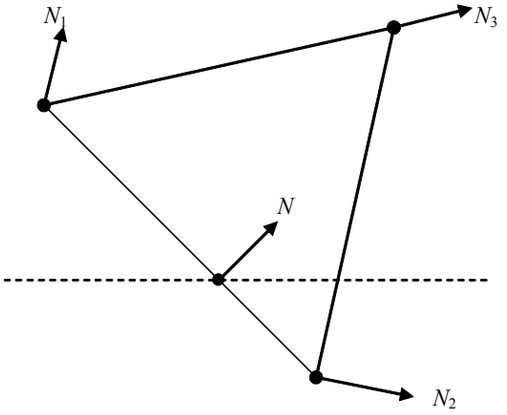


Fig. 9.28 *Interpolation of Surface Normals along a Polygon Edge*

The incremental methods can be used to evaluate normals between scan lines and along each individual scan line. The illumination model is applied at each pixel position along a scan line to determine the surface intensity at that point. Intensity calculations (by using an approximated normal vector at each point of the scan line) produce more accurate results than the direct interpolation of intensities, as we have seen in Gouraud shading. However, the drawback is that Phong shading requires considerably more computations.

9.6.2 Fast Phong Shading

Surface rendering with Phong shading can be accelerated using approximations in the illumination-model calculations of normal vectors. Fast Phong shading

NOTES

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

$$N = Ax + By + C$$

...(9.37)

where vectors A, B, and C are determined from the three vertex equations,

$$N_k = Ax_k + By_k + C, \quad \text{for} \quad k = 1, 2, 3$$

...(9.38)

Omitting the reflectivity and attenuation parameters, the calculation for light-source diffuse reflection can be written from a surface point (x, y) as follows:

$$I_{diff}(x, y) = \frac{L.N}{|L|.|N|}$$

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

...(9.39)

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

We can rewrite this expression in the following form:

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

...(9.40)

where parameters such as a, b, c, and d are used to represent the various dot products. For example,

$$a = \frac{L.A}{|L|}$$

...(9.41)

Finally, we can state the denominator in equation 9.40 as a Taylor-series expansion and retain terms up to the second degree in x and y. This gives the following:

$$I_{diff}(x, y) = T_5x^2 + T_4xy + T_3y^2 + T_2x + T_1y + T_0$$

...(9.42)

NOTES

where each T_i is a function of parameters a, b, c, and so on. Using forward differences, we can evaluate equation 8.20 with only two additions for each pixel position (x, y), once the initial forward-difference parameters have been evaluated. Fast Phong shading reduces the Phong-shading calculations. However, it takes approximately twice the time to render a surface with fast Phong shading as it does with Gouraud shading. As far as efficiency is concerned, normal Phong shading using forward difference methods takes about 6–7 times longer than Gouraud shading. Fast Phong shading for diffuse reflection can be extended to include specular reflections. Calculations similar to those for diffuse reflections can be used to evaluate specular terms such as $(N.H)^{n_s}$ in the basic illumination model. Additionally, we can generalize the algorithm to include polygons other than triangles and to include finite viewing positions.

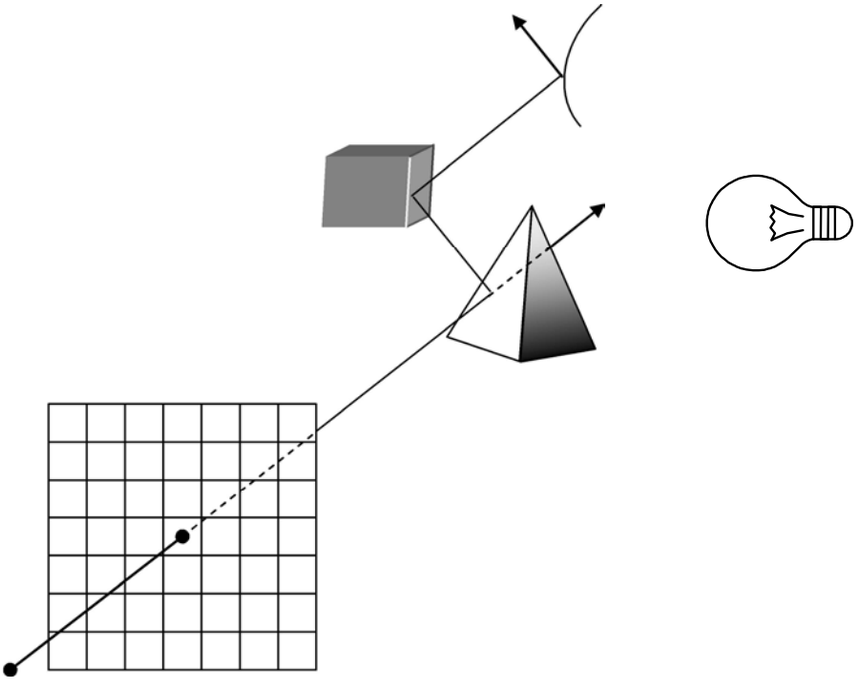


Fig. 9.29 *Tracing a Ray from the Projection Reference Point Through a Pixel Position with Multiple Reflections and Transmissions*

Check Your Progress

1. Write a short note on Bezier curve.
2. What are the factors on which lighting calculations depends?
3. Discuss the constant-intensity shading method.

9.7 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

NOTES

1. Bezier curve is the spline approximation method that was developed by the French engineer Pierre Bezier to design automobile bodies. The Bezier spline has a number of properties that makes it highly useful and convenient for surface and curve design.
2. Lighting calculations are based on the following:
   (i) Optical properties of surfaces
   (ii) Light-source specifications
   (iii) Background lighting conditions
3. A fast and simple method for rendering an object with polygon surfaces is constant- intensity shading, also known as flat shading. This method calculates a single intensity for each polygon. All specific points over the surface of the polygon are then highlighted with the same intensity value.

9.8 SUMMARY

- Bezier curve is the spline approximation method that was developed by the French engineer Pierre Bezier to design automobile bodies. The Bezier spline has a number of properties that makes it highly useful and convenient for surface and curve design.
- Many graphics packages make available only cubic spline functions. This facilitates reasonable design flexibility and avoids the increased calculations required with higher-order polynomials.
- Two sets of orthogonal Bezier curves to design an object surface by specifying by an input mesh of control points.
- An opaque non-luminous object, we observe the light reflected from the surfaces of the object. The total reflected light is equal to the sum of the contributions from light sources and other reflecting surfaces in the scene.
- An object surface which is not exposed directly to a light source is visible due to the light reflected form the nearby objects that are illuminated. In the basic illumination model, we can set a general level of brightness for a scene.
- The background-light reflection is an approximation of global diffuse lighting effects. Diffuse reflections are constant over each surface in a scene, independent of the viewing direction.
- The applications of an illumination model to the rendering of graphics objects which are formed with polygon surfaces. Objects are usually polygon-mesh approximations of curved-surface objects, but they may also be polyhedra which are not curved-surface approximations.

- A more exact method for rendering a polygon surface interpolates the normal vectors, and then applies an illumination model to each surface point. This method was developed by Phong Bui Tuong. Therefore it is called Phong shading, or normal vector interpolation shading.

9.9 KEY WORDS

- **Bezier Curve:** It is the spline approximation method that was developed by the French engineer Pierre Bezier to design automobile bodies.
- **Phong Shading**:- A more exact method for rendering a polygon surface interpolates the normal vectors, and then applies an illumination model to each surface point. This method was developed by Phong Bui Tuong.
- **Illumination Model:** An illumination model, also called a shading model or a lighting model, calculates the intensity of light that one can see at a given point on the surface of an object.

9.10 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Discuss the properties of Bezier curve.
2. What are Bezier surfaces?
3. Discuss the various types of polygon rendering methods.

Long Answer Questions

1. Explain the design techniques using Bezier curve.
2. What do you understand by B-spline curve and surfaces? Explain.
3. Write a detailed note on illumination models.

9.11 FURTHER READINGS

Hearn, Donal and M. Pauline Baker. 1990. *Computer Graphics*. New Delhi: Prentice-Hall of India.

Rogers, David F. 1985. *Procedural elements for Computer Graphics*. New York: McGraw-Hill.

Foley, D. James, Andries Van Dam, Steven K. Feiner and John F. Hughes. 1997. *Computer Graphics Principles and Practice*. Boston: Addison Wesley.

Mukhopadhyay, A. and A. Chattopadhyay. 2007. *Introduction to Computer Graphics and Multimedia*. New Delhi: Vikas Publishing House.

BLOCK - IV

3D GEOMETRIC TRANSFORMATION

NOTES

UNIT 10 3D GEOMETRIC TRANSFORMATIONS

Structure

10.0 INTRODUCTION

In this unit, you will learn about the three basic transformation i.e. translation, rotation and scaling in 3D geometry. Using two-dimension techniques, you can show only some of the graphics applications such as bar charts, pie charts and graphs. However, most natural objects can only be shown in three dimensions. 3D graphics allows you to check the structure in different directions. When you discuss two-dimensional rotations in the *xy*-plane, you need to consider only rotations about axes that are perpendicular to the *xy*-plane. You can select any spatial orientation in the three dimensional space, as a composite of three rotations, one for each of the three Cartesian axes. On the other hand, a user can easily set up a general rotation matrix, given the orientation of the axis and the required rotation angle. Reflection and shear are the two other transformations which are also discussed.

10.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain the translation, rotation and scaling in three dimensional geometry
- Understand reflection and shear transformation in three dimensional geometry

NOTES

10.2 THREE-DIMENSIONAL GEOMETRY

The three-dimensional system has three axes: *x*, *y* and *z* axis. The orientation of coordinate system is determined by two systems: the right hand system and the left hand system. In the right handed system, the thumb of the right hand points in the positive *z* direction as one curls the fingers of the right hand from *x* into *y*, and in the left handed system the thumb points in the negative z direction. In our description we have used the right-handed system.

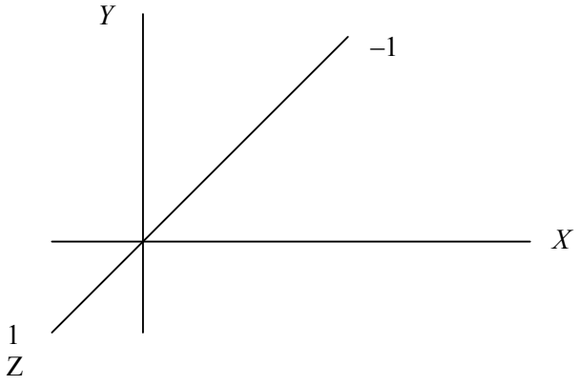


Fig. 10.1 *(a) Right-Handed System*

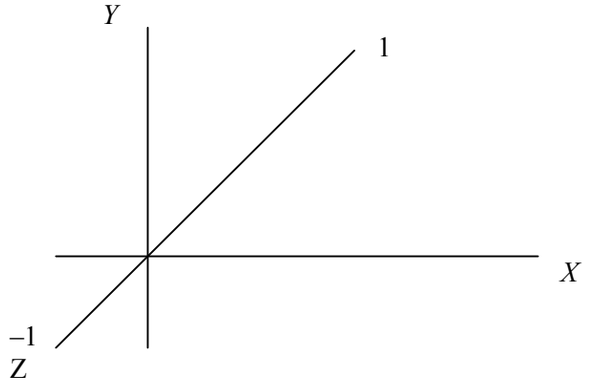


Fig 10.1 *(b) Left- Handed System*

NOTES

As in the case of a two-dimensional system, both Y and Z coordinates change proportionately to X. This is given as follows:

$$(y - y_1)/(x - x_1) = (y_2 - y_1)/(x_2 - x_1)$$

$$(z - z_1)/(x - x_1) = (z_2 - z_1)/(x_2 - x_1)$$

where (x_1, y_1, z_1) and (x_2, y_2, z_2) are the two points. So we can represent the parametric form of a line as,

$$x = (x_2 - x_1)\, u + x_1$$
$$y = (y_2 - y_1)\, u + y_1$$
$$z = (z_2 - z_1)\, u + z_1$$

The equation of the plane is given by,

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

We can show that the constants may be divided out of the equation so that,

$$A_1x + y + C_1z + D_1 = 0$$

where
$$A_1 = A/B$$
$$C_1 = C/B$$
$$D_1 = D/B$$

The coordinates of the three points are, and. Then the equations of the planes are as follows:

$$A_1x_1 + y_1 + C_1z_1 + D_1 = 0$$
$$A_1x_2 + y_2 + C_1z_2 + D_1 = 0$$
$$A_1x_3 + y_3 + C_1z_3 + D_1 = 0$$

If we solve these three equations then we get normalization as follows:

$$A_2 = A/D$$
$$B_2 = B/D$$
$$C_2 = C/D$$
$$D_2 = D/d$$

where $d = \sqrt{(A_2 + B_2 + C_2)}$

The distance between a point (x, y, z) and the plane is given by,

$$L = | A_2x + B_2y + C_2z + D_2 |$$

The sign of this indicates whether a point lies on the front or the back of a plane.

Three-dimensional Transformations

As in two-dimensional transformations, there are three basic transformations for three-dimensional geometry. These are as follows:

(i) Translation (ii) Scaling (iii) Rotation

NOTES

10.2.1 Translation

In 3D homogenous coordinate representation, a transformation matrix for the translation of $P = (x, y, z)$ to the position P' = is given by,

$$[x_1, y_1, z_1] = [x, y, z, 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

where t_x, t_y and t_z are translation factors. So,

$$x_1 = x + t_x$$
$$y_1 = y + t_y$$
$$z_1 = z + t_z$$

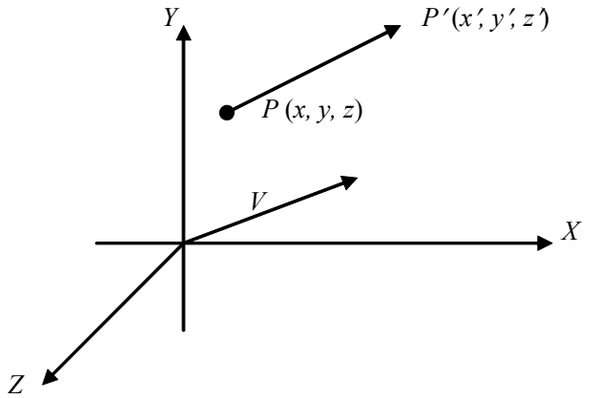


Fig. 10.2 *Three Dimensional Translation*

We can determine the inverse of this by making the values of t_x, t_y and t_z negative. This produces a translation in the opposite direction, and the product of a translation matrix and its inverse produces the identity matrix.

10.2.2 Scaling

Two-dimensional scaling is used to change the size of an object. Scaling transformation matrix will be represented as folows:

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

NOTES

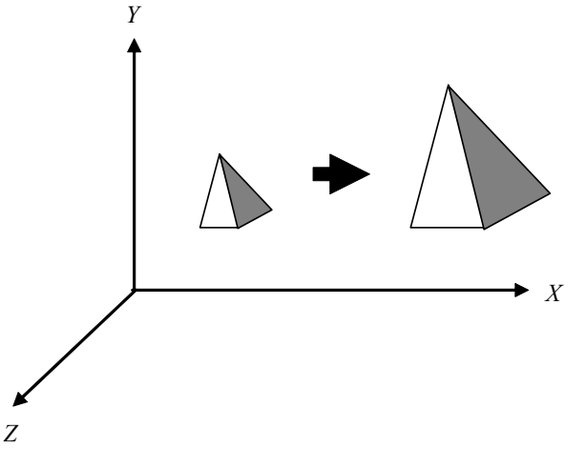


Fig. 10.3 *Three Dimensional Scaling*

Or we can write it in the form of an equation as follows:

$x' = x * s_x$, $y' = y * s_y$ and $z' = z * s_z$

10.2.3 Rotation

Rotation about the origin in two-dimensional geometry considers an angle of rotation and a centre of rotation. But rotation in three-dimensional geometry is more complex than in two-dimensional geometry. Because in this case we consider an angle of rotation and an axis of rotation. Therefore there are three cases from where we can select one of the positive *x*-axis, *y*-axis and *z*-axis as an axis of rotation.

Rotation about *x*-axis

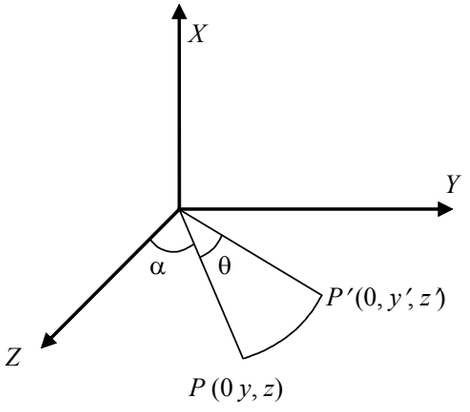


Fig. 10.4 *Rotation about x-axis*

We can see from Figure 10.4 that,

$x' = x$

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

NOTES

Since $r\cos\alpha = y$ and $r\sin\alpha = z$ then the equation becomes,

$$y' = y\cos\theta - z\sin\theta$$

and $z' = r\sin(\alpha + \theta) = r\sin\alpha\cos\theta + r\cos\alpha\sin\theta$

$$z' = z\cos\theta + y\sin\theta$$

So the equation becomes,

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

or the equations can be written in the matrix form as follows:

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

Rotation about *y*-axis

When we rotate the object about the *y*-axis, then the equations becomes,

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

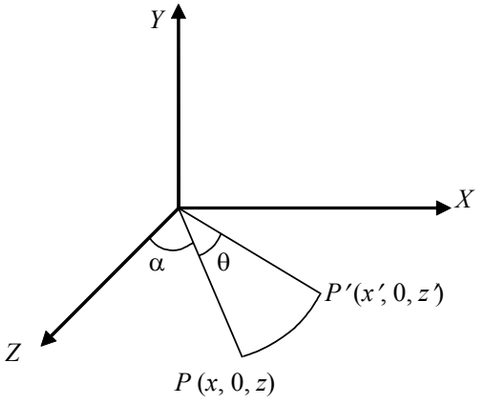


Fig. 10.5 *Rotation about y-axis*

or we can write the above equation in the matrix form as follows:

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

NOTES

Rotation about *z*-axis

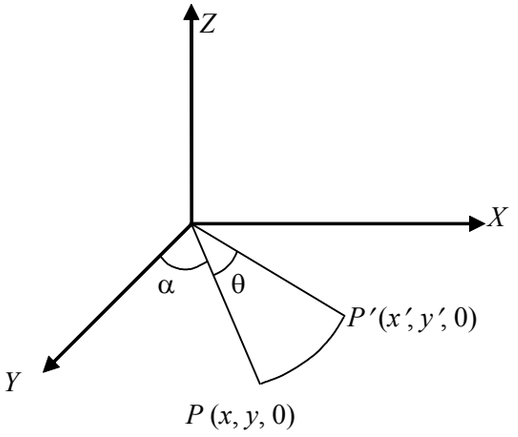


Fig. 10.6 *Rotation about z-axis*

The equations about the z-axis rotation can be written as follows:

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

or the equations can be written in the matrix form as follows:

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

Rotations about an Arbitrary Line

A rotation matrix for any axis that does not overlap with a coordinate axis can be setup as a composite transformation. This involves combinations of the coordinate axis rotations and translations in either sequence. In the special case where an object is to be rotated about an axis (parallel to one of the coordinate axis), we can achieve the desired rotation by following the given transformation sequence:

(i) Translating the object such that the rotating axis coincides with the parallel coordinate axis

(ii) Performing the specified rotation about the axis

(iii) Translating the object by moving the rotation axis back to its original position.

These steps are illustrated in Figure 10.7.

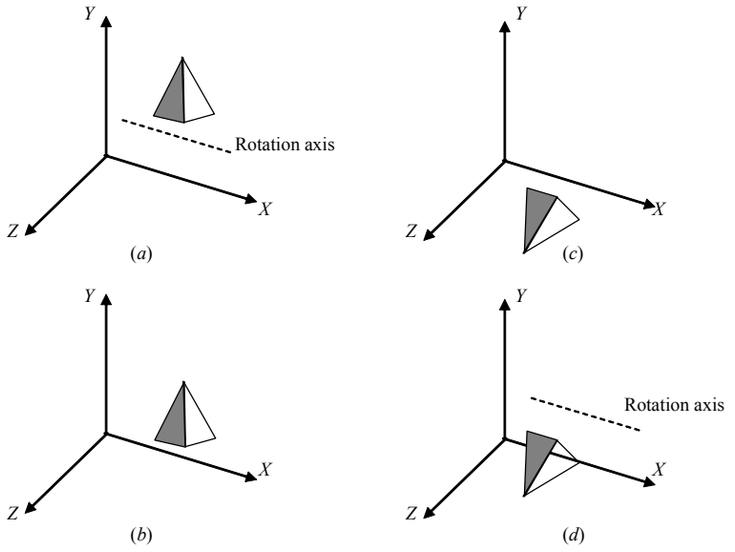


Fig. 10.7 *Sequence of Transformations for Rotation of the Object about Axis Parallel to X-axis*

Any coordinate position P on the object is transformed with the sequence as follows:

$$P' = T^{-1} . R_x (\theta). T. P$$

which is the same as the two-dimensional transformation sequence for rotation about an arbitrary point. When we wish to rotate an object about an axis that is not parallel to one of the coordinate axes, we need to perform a sequence of additional transformations. The sequence also includes rotations to align the axis with a selected coordinate axis and finally, to bring the axis back to its original orientation. We can perform the required rotation in the following steps:

(i) Translation of the object so that the rotation axis passes through the coordinate origin

(ii) Rotation of the object so that the axis of rotation coincides with one of the coordinate axes

(iii) Performing the specified rotation about the axis

(iv) Performing inverse rotations to bring the rotation axis back to its original orientation

(v) Performing the inverse translation to bring the rotation axis back to its original position

NOTES

NOTES

The parametric equation for the line is as follows:

$$x_2 = x_1 + Au$$
$$y_2 = y_1 + Bu$$
$$z_2 = z_1 + Cu$$

is the point on the line and (A, B, C) vector gives the direction.

Step 1: Translation,

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_1 & -y_1 & -z_1 & 1 \end{bmatrix}$$

So the point (x_1, y_1, z_1) moves to the origin.

Now,

$$T_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_1 & y_1 & z_1 & 1 \end{bmatrix}$$

T_1 gives a translation in the opposite direction.

Step 2: In this step we have to align a axis on the Z-axis. For this we follow the given two steps:

(i) Rotation about *X*-axis by an angle *k* such that the shadow lies on *Z*-axis

(ii) Rotate about *Y*-axis by an angle *L* so that an axis will be aligned on the *Z*-axis

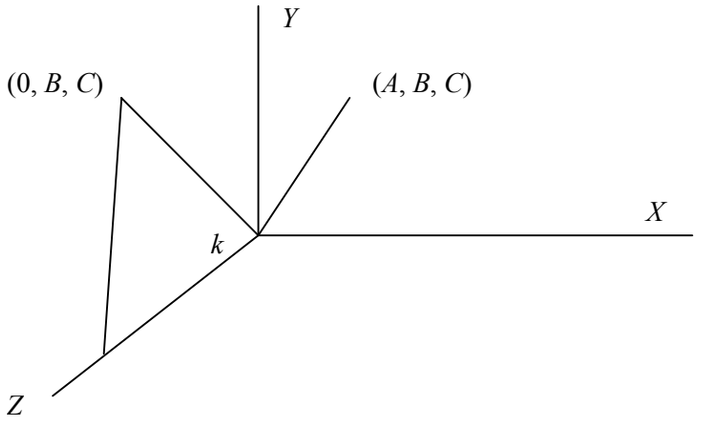


Fig. 10.8 *Rotation about x-axis by an Angle k*

Now we find the transformation matrix for each case.

(*i*) The axis (*A, B, C*) translated to the origin having its shadow in the *yz*-plane. So $A = 0$. The shadow is (0, *B, C*). The length of an axis is,

$$U = (A^2 + B^2 + C^2)^{0.5}$$

And the length of a shadow is,

$$V = (B^2 + C^2)^{0.5}$$

Rotate about *X*-axis by an angle *k* so that the shadow will be on *Z*-axis

From the above diagram we can calculate *sin k* and *cos k* as follows:

$$sin\ k = \text{B/V}$$

$$cos\ k = \text{C/V}$$

The rotation about an axis by an angle *k* is given by a transformation matrix as follows.

$$Rx = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos k & \sin k & 0 \\ 0 & -\sin k & \cos k & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & C/V & B/V & 0 \\ 0 & -B/V & C/V & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And rotation about an *X-axis* in reverses direction Rx_1,

$$Rx_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & C/V & -B/V & 0 \\ 0 & B/V & C/V & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

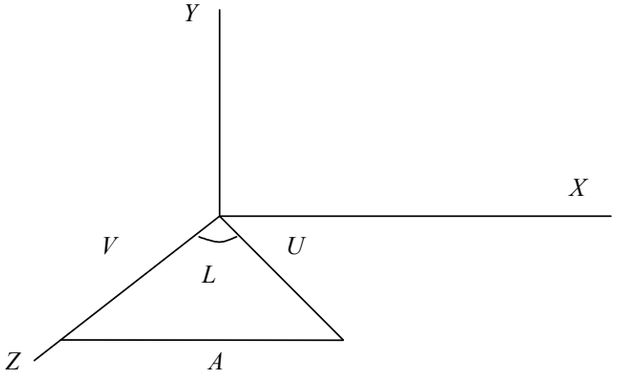


Fig. 10.9 *Rotation about Y-axis by an Angle L*

(*ii*) Now we will rotate about *Y*-axis so that it aligns on *Z*-axis by an angle *L*,

$$sin\ L = \text{A/U}$$

$$cos\ L = \text{V/U}$$

NOTES

NOTES

The rotation matrix will be as follows:

$$Ry = \begin{bmatrix} \cos L & 0 & \sin L & 0 \\ 0 & 1 & 0 & 0 \\ -\sin L & 0 & \cos L & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} V/U & 0 & A/U & 0 \\ 0 & 1 & 0 & 0 \\ -A/U & 0 & V/U & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and
$$Ry_1 = \begin{bmatrix} V/U & 0 & -A/U & 0 \\ 0 & 1 & 0 & 0 \\ A/U & 0 & V/U & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

After this the arbitrary axis aligns on the *z*-axis as shown in Figure 10.10.

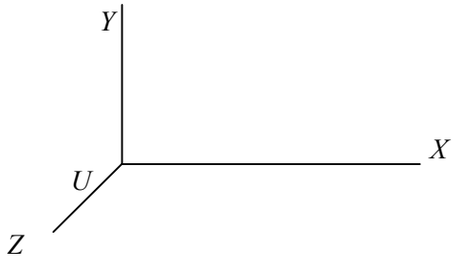


Fig. 10.10 *Alignment of the Arbitrary Axis on Z-axis*

Step 3: Finally we rotate by an angle *A* about the *Z*-axis because we align an arbitrary axis with the *Z*-axis. The rotation about *Z*-axis by an angle *A* is given by,

$$Rz = \begin{bmatrix} \cos A & \sin A & 0 & 0 \\ -\sin A & \cos A & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Step 4: Now inverse rotation about a *y* and *Z*-axis

Step 5: Inverse translation. So the result will be as follows:

$$RA = T.Rx.\ Ry.\ Rz.\ Rx_1.\ Ry_1.\ T_1$$

where Rx_1, Ry_1, T_1 are the inverse of *Rx, Ry, T* matrices and *RA* is the rotation about an arbitrary axis.

10.2.4 Composite Transformations

As with two-dimensional transformations, we make a composite three-dimensional transformation by multiplying the matrix representations for the individual operations

in the transformation sequence. This concatenation is carried out from right to left, where the rightmost matrix is the first transformation which is applied to an object and the leftmost matrix is the last transformation to be applied. A sequence of basic, three-dimensional geometric transformations is combined to produce a single composite transformation, which is then applied to the coordinate definition of an object.

10.3 OTHER TRANSFORMATIONS

There are also some other transformations, which can be applied to a three-dimensional object. These transformations can be given as follows:

10.3.1 Reflection

Reflection in three dimensional (3D) transformations is the reflection of a point/object relative to a plane. The reflection about xy plane changes a right hand side system to left hand side one.

$$M_{xy} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

M_{xy} is the matrix of mirror reflection relative to *xy*-plane. Similarly we can define the matrix relative to the *yz* and *zx*-plane as follows:

$$M_{yz} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_{zx} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

10.3.2 Shear

Shearing can be done with respect to an axis. This transformation keeps the value corresponding to that axis co-ordinate unchanged. The *Z*-axis shear is given as follows:

NOTES

$$SH_Z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ a & b & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Parameters a and b can take any real values. The effect of this transformation matrix is to change x and y co-ordinate values by an amount that is proportional to the z value, while leaving the z co-ordinate unchanged. Similarly,

$$SH_X = \begin{bmatrix} 1 & a & b & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad SH_y =$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ a & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Example 10.1: Find the scaling transformation matrix to scale by s_x, s_y, s_z units with respect a fixed point $P(x, y, z)$.

Solution: Scaling with respect to a fixed point can be performed by the following sequence:

(*i*) Translation

(*ii*) Scaling

(*iii*) Inverse translation

The scaling transformation matrix will be as follows:

$$S = T * S_{XYZ} * T_1$$

where

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

$$T_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{bmatrix} \quad \text{and} \quad S_{XYZ} = \begin{bmatrix} s_X & 0 & 0 & 0 \\ 0 & S_Y & 0 & 0 \\ 0 & 0 & s_Z & 0 \\ x & y & z & 1 \end{bmatrix}$$

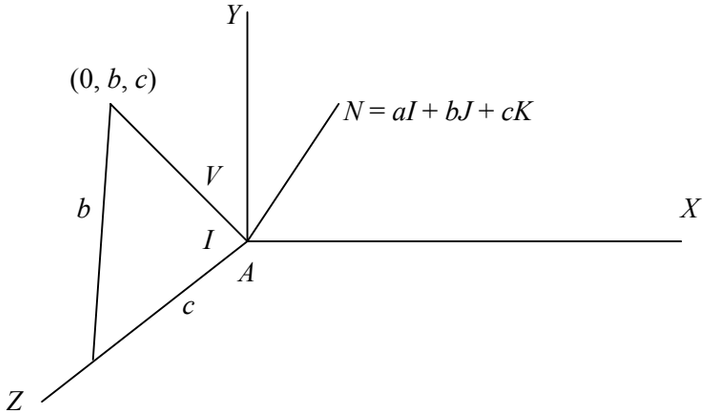
So we get S as,

$$S = \begin{bmatrix} s_X & 0 & 0 & 0 \\ 0 & S_Y & 0 & 0 \\ 0 & 0 & s_Z & 0 \\ (1-s_X)x & (1-S_Y)y & (1-s_Z)z & 1 \end{bmatrix}$$

NOTES

Example 10.2: Find the alignment transformation AN which aligns a given vector $aI + bJ + cK$ to the positive z-axis.

Solution: First rotate about x-axis and then rotate about y-axis.



The distances V and I are determined as follows:

$$V = \sqrt{(b^2 + c^2)}$$

$$I = \sqrt{(a^2 + b^2 + c^2)}$$

$$\cos I = c / V$$

$$\sin I = b / V$$

Now rotate the given vector $aI + bJ + cK$ about x-axis by an angle I

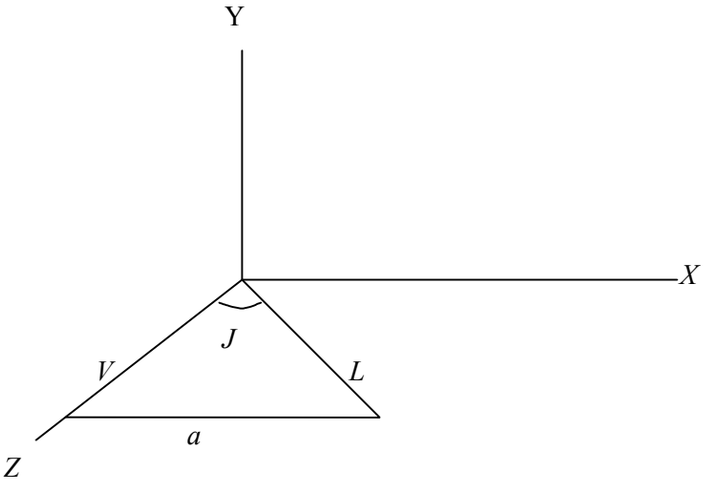
$$Rx = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos I & \sin I & 0 \\ 0 & -\sin I & \cos I & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c/V & b/V & 0 \\ 0 & -b/V & c/V & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotate vector N about y-axis by an angle J so we get vector N align of positive z-axis as follows:

$$\cos J = V/L$$

$$\sin J = a/L$$

NOTES



$$Ry = \begin{bmatrix} \cos J & 0 & \sin J & 0 \\ 0 & 1 & 0 & 0 \\ -\sin J & 0 & \cos J & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} V/L & 0 & a/L & 0 \\ 0 & 1 & 0 & 0 \\ -a/L & 0 & V/L & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We get alignment transformation *AN*, by multiplying *Rx* an *Ry* as following:

$$AN = Rx'Ry = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c/V & b/V & 0 \\ 0 & -b/V & c/V & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} V/L & 0 & a/L & 0 \\ 0 & 1 & 0 & 0 \\ -a/L & 0 & V/L & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} V/L & 0 & a/L & 0 \\ -ba/VL & c/V & b/L & 0 \\ -ca/VL & -b/V & c/L & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Check Your Progress

1. What are three transformations in 3D geometry?
2. What is reflection in 3D transformations?

10.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

NOTES

1. There are three basic transformations for three-dimensional geometry. These are as follows:

   (i) Translation (ii) Scaling (iii) Rotation

2. Reflection in three dimensional (3D) transformations is the reflection of a point/ object relative to a plane.

10.5 SUMMARY

- The three-dimensional system has three axes: x, y and z axis. The orientation of coordinate system is determined by two systems: the right hand system and the left hand system.
- Rotation about the origin in two-dimensional geometry considers an angle of rotation and a centre of rotation. But rotation in three-dimensional geometry is more complex than in two-dimensional geometry.
- A rotation matrix for any axis that does not overlap with a coordinate axis can be setup as a composite transformation. This involves combinations of the coordinate axis rotations and translations in either sequence.
- Two-dimensional transformations, we make a composite three-dimensional transformation by multiplying the matrix representations for the individual operations in the transformation sequence.
- Reflection in three dimensional (3D) transformations is the reflection of a point/ object relative to a plane. The reflection about xy plane changes a right hand side system to left hand side one.
- Shearing can be done with respect to an axis. This transformation keeps the value corresponding to that axis co-ordinate unchanged.

10.6 KEY WORDS

- **Translation:** It is a geometric transformation that moves every point of a figure or a space by the same distance in a given direction.
- **Reflection:** In three dimensional (3D) transformations is the reflection of a point/ object relative to a plane.

NOTES

10.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Write the transformation matrix for translation in 3D geometry.
2. Write the scaling transformation matrix.
3. Discuss the rotation about an arbitrary line.

Long Answer Questions

1. Explain the rotation about x and y axis.
2. Explain the reflection and shear transformation in 3D geometry.

10.8 FURTHER READINGS

Hearn, Donal and M. Pauline Baker. 1990. *Computer Graphics*. New Delhi: Prentice-Hall of India.

Rogers, David F. 1985. *Procedural elements for Computer Graphics*. New York: McGraw-Hill.

Foley, D. James, Andries Van Dam, Steven K. Feiner and John F. Hughes. 1997. *Computer Graphics Principles and Practice*. Boston: Addison Wesley.

Mukhopadhyay, A. and A. Chattopadhyay. 2007. *Introduction to Computer Graphics and Multimedia*. New Delhi: Vikas Publishing House.

UNIT 11 3D VIEWING

NOTES

Structure

11.0 INTRODUCTION

In this unit, you will learn about the three dimensional viewing pipeline and projection transforms. Viewing pipeline in 3D is almost the same as the 2D viewing pipeline. Projection results in representing the 3D objects in 2D plane. Three-dimensional projection can be defined as any method of mapping three dimensional (3D) points on a two-dimensional (2D) plane. The most current methods for displaying graphical data are based on planar two-dimensional media. Therefore the use of this type of projection is widespread, especially in engineering design, computer graphics and drafting. There are basically two methods of projection. One method, called perspective projection, shows the object as it appears. And the other method, called parallel projection, shows the object to its true size and shape.

11.1 OBJECTIVES

After going through this unit, you will be able to:

- Discuss the viewing pipeline and coordinates
- Describe the general projection transform and clipping

11.2 VIEWING PIPELINE AND COORDINATES

Computer generation of a view of an object on a display device requires stage-wise transformation operations of the object definitions in different coordinate systems. We present here a non-mathematical introduction to each of these

NOTES

coordinate systems to help you develop an intuitive notion of their use and relationship to one another.

Global Coordinate System, also called the ***World Coordinate System*** is the principal frame of reference in which all other coordinate systems are defined. This three-dimensional system is the basis for defining and locating in space all objects in a computer graphics scene, including the observers position and line of sight. All geomatric transformations like translation, rotation, scaling, reflection etc. are carried out with reference to this global coordinate system.

A ***Local Coordinate System,*** or a ***Modeling Coordinate System*** is used to define the geometry of an object independently of the global system. This is done for the ease of defining the object-details w.r.t reference point and axes on the object itself. Once you define the object 'locally', you can place it in the global system simply by specifying the location and orientation of the origin and axes of the local system within the global system, then mathematically transforming the point coordinates defining the object from the local to the global system.

The ***Viewing Coordinate System*** locates objects in three-dimensional space relative to the location of the observer. We use it to simplify the mathematics for projecting an image of the object onto the projection plane. To establish the viewing coordinate reference frame first define an *eyepoint* P_E (eye of the observer) or *view reference point*. Next, specify the direction of the observer's line of sight in either of the two ways: as a set of direction angles (or direction cosines) in the global system, or by specifying the location of a *viewpoint* P_V or *look-at point*. This directed line segment from the eye point to the look-at point is also referred to as the *view-plane normal vector* **N** or viewing axis Z_V. A *view plane* or *projection plane* is then set up perpendicular to **N** or Z_V. The (+)ve direction of the Y_V axis of the viewing coordinate system is called the *view-up vector* with the view reference point being the origin of the system.

Different views of an object is obtained on the view plane either by moving the object and keeping the eyepoint fixed or by moving the eyepoint keeping the object fixed. However, the later technique is assumed by most of the computer graphics application to create a new view of the object.

The two-dimensional ***Device Coordinate System*** (or ***Screen Coordinate System*** for display monitor) locates points on the display/output of a particular output device such as graphics monitor or plotter. These coordinates are integers in terms of pixels, addressable points, inches, cms etc.

The ***Normalized Device Coordinates*** are used to map world coordinates in a device independent two-dimensional pseudospace within the range 0 to 1 for each of x and y before final conversion to specific device coordinates.

The modeling and world coordinate positions can be any floating-point values; normalized coordinates (x_{nc}, y_{nc}) satisfy the inequalities: $0 \le x_{nc} \le 1,\ 0 \le y_{nc} \le 1$; and the device coordinates are integers within the range (0, 0) to $(x_{\max}, y_{\max})$, with $(x_{\max}, y_{\max})$ depending on the resolution of a particular output device.

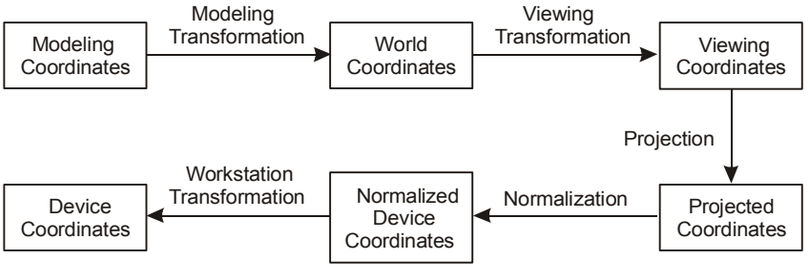


Fig. 11.1 *Graphics coordinate system*

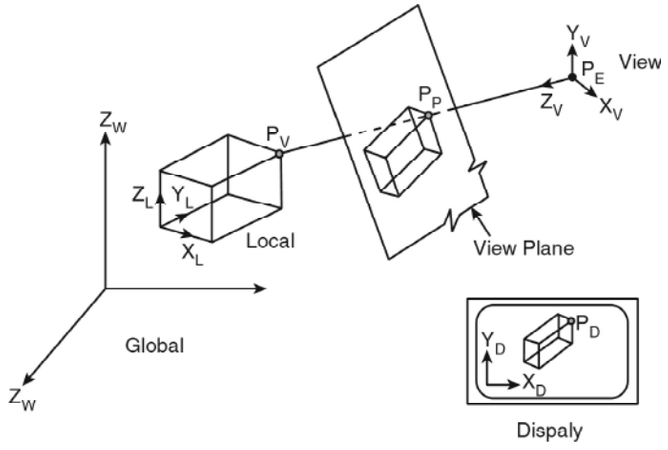


Fig. 11.2 *Three-dimensional viewing pipeline*

You will frequently see the terms *object space* and *image space*. Object space corresponds to the world coordinate system, image space to the display-screen coordinate system. Object space is an unbounded, infinite collection of continuous point. Image space is a finite 2D space. In 2D, we simply specify a window in the object space and a viewport in the display surface. Conceptually, 2D objects in the object space are clipped against the window and are then transformed to the normalized viewport and finally to the viewport for display using standard window-to-viewport mapping.

Unlike 2D, 3D objects are conceptually clipped against a *view volume* and are then projected. The contents of the projection of the view volume onto the projection plane, i.e., the view window or projection window is then mapped to the viewport for display. Only those objects within the view volume will appear in the display viewport; all others are clipped from display. The shape of the view volume varies according to the type of projection though the size is limited by suitably choosing front plane (or near plane) and back plane (or far plane). For a orthographic parallel projection the view volume is a rectangular parallelepiped whereas for perspective projection the view volume is a truncated pyramid or frustum. Refer Fig. 11.2.

NOTES

NOTES

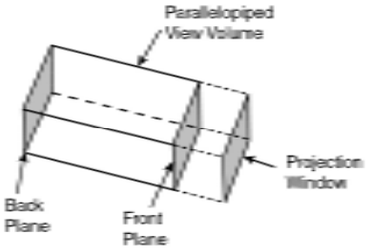


Fig. 11.3 *Parallel projection*

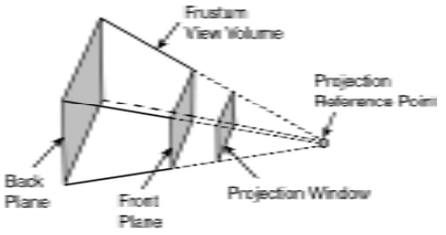


Fig. 11.4 *Perspective projection*

11.3 GENERAL PROJECTION TRANSFORMS AND CLIPPING

With the help of projection, you can take a view of an object from different directions and different distances. In this section, you will learn about the important types of projects.

11.3.1 Parallel Projection

Parallel projection shows the true image, size and shape of an object. The angle made by the direction of projecting lines with the projection plane or view plane is determined. Projection rays (projectors) emanate from a COP (centre of projection) and intersect projection plane (see Figure 11.5). The centre of projection for parallel projectors is at infinity. The length of a line on the projection plane is the same as the 'true length'.

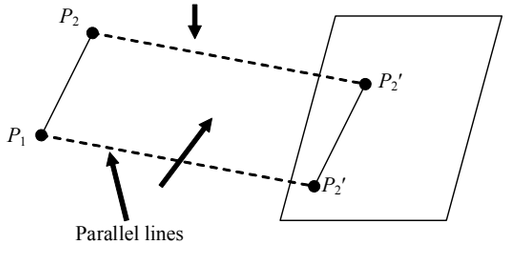


Fig. 11.5 *Illustration of Parallel Projection*

There exist two different types of parallel projections in practice. Let us consider Figure 11.6 that illustrates the parallel projection of a point (*x, y, z*). The projection plane is across the line $z = 0$. The values *x, y* represent the orthographic projection values and the values *xp, yp* are the oblique projection values.

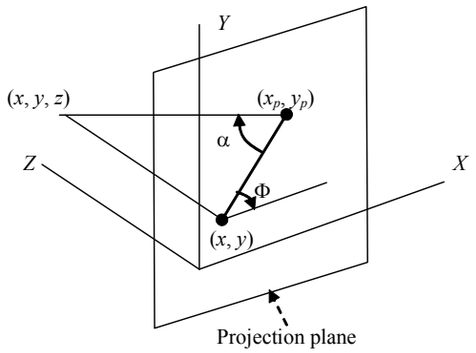


Fig. 11.6 *Illustration of Orthographic Projection*

If you take note of orthographic projection, it just discards the *z* coordinates. Drawings in engineering frequently use top, front and side orthographic views of an object. The following diagram illustrates three orthographic views of an object (see Figure 11.7).

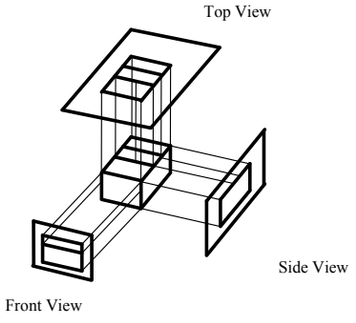


Fig. 11.7 *Orthographic View of an Object*

Orthographic projections that show more than one side of an object are called **axonometric** orthographic projections. The most common axonometric projection is an **isometric** projection where each coordinate axis is intersected by the projection plane in the model coordinate system at an equal distance.

11.3.2 Isometric Projection

This is the projection in which projection plane intersects the *x*-, *y*-and *z*-axes at equal distances and the projection plane normal makes an equal angle with the three axes. To obtain an orthographic projection, let us assume $x_p = x, y_p = y$ and $z_p = 0$. This projection is illustrated by Figure 11.8.

NOTES

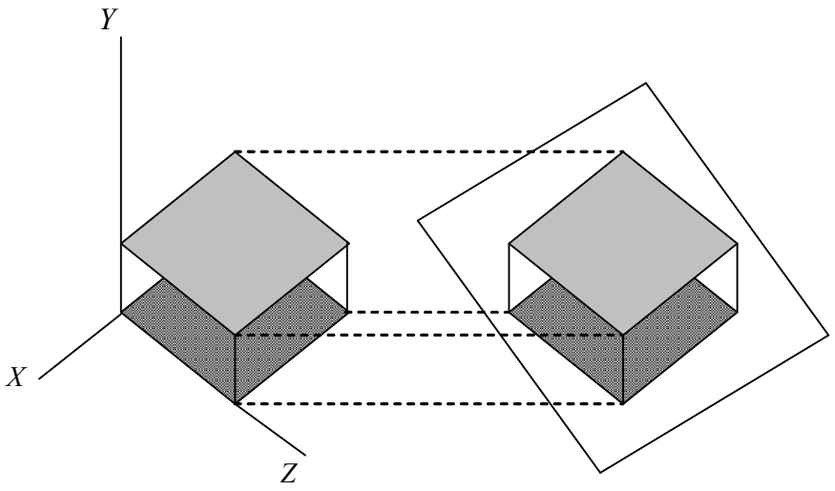


Fig. 11.8 *Illustration of Isometric Projection*

11.3.3 Oblique Projection

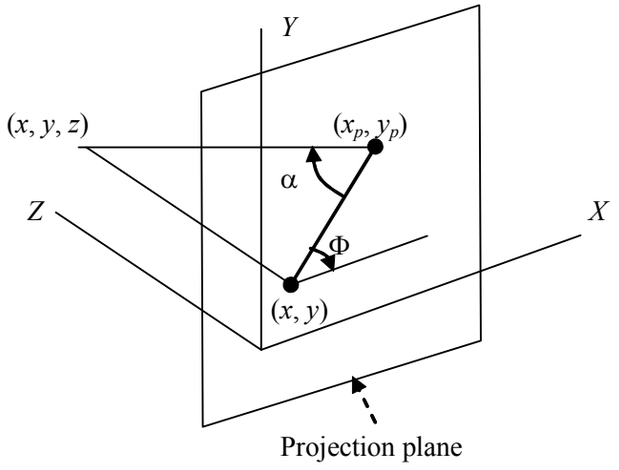


Fig. 11.9 *Illustration of Oblique Projection*

Case (*i*) If $A = 45°$, then $\tan A = 1 \Rightarrow L_1 = 1$. The projectors are defined by two angles A and d where A = angle of line (x, y, x_p, y_p) with projection plane, d = angle of line (x, y, x_p, y_p) with x-axis in projection plane and L = Length of Line (x, y, x_p, y_p), then:

$$\cos d = (x_p - x)/L \Rightarrow x_p = x + L \cos d,$$

$$\sin d = (y_p - y)/L \Rightarrow y_p = y + L \sin d, \quad \tan A = z/L$$

Thus,

$$L_1 = L/z \quad \Rightarrow L = L_1 z,$$

Therefore, $\tan A = z / L = 1 / L_1$

$x_p = x + z(L_1 \cos d)$, and

$$y_p = y + z(L_1 \sin d) \quad P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ L_1 \cos q & L_1 \sin q & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

NOTES

Now, if the angle $A = 90°$ (that is, the projection line is perpendicular to the projection Plane), then $\tan A = \infty \Rightarrow L_1 = 0$, thus giving an orthographic projection. There are two special cases of oblique projection (perpendicular to the projection plane) that are projected with no change in length (refer to Figure 11.10).

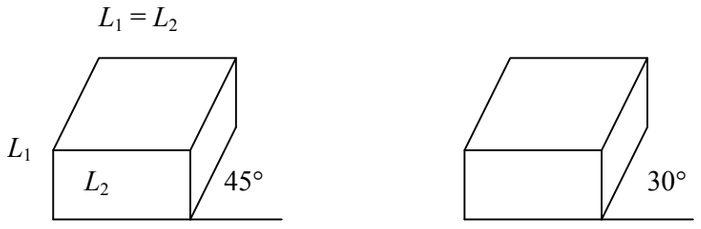


Fig. 11.10 *Two Cubes Cavalier Projection*

Case (ii) If $\tan A = 2$, then $A = 63.40°$ and $L_1 = \frac{1}{2}$. The lines that are perpendicular to the projection plane are projected at $\frac{1}{2}$ length. This is also called a cabinet projection.

$L_1 = ½ L_2$

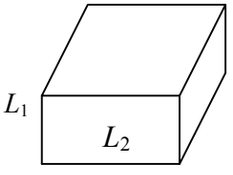


Fig. 1.11 *Illustration of Cabinet Projection*

11.3.4 Perspective Projection

A perspective projection can be viewed as the projection that has a centre of projection at a finite distance from the plane of projection. The perspective projection is shown in Figure 11.12:

NOTES

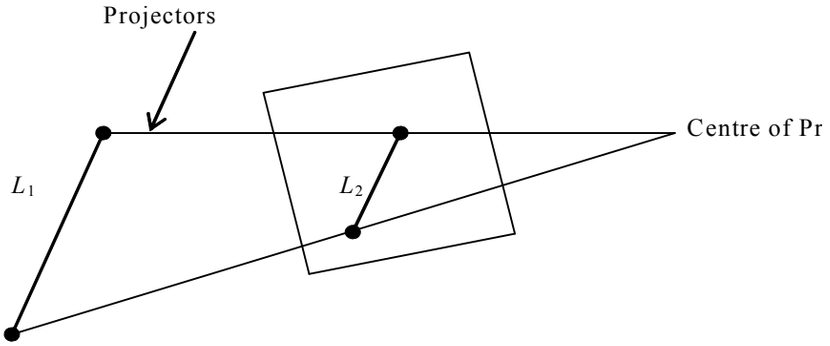


Fig. 11.12 *Illustration of Perspective Projection*

The distance of a line from the projection plane determines its size on the projection plane, which means the further the line from the projection plane, the smaller its image on the projection plane. As seen in Figure 11.12, there is the projection of $L_1 = L_2$ but the actual length of L_1 is not equal to L_2. The perspective projection is more practical because the distant objects appear smaller.

Computing the Perspective Projection

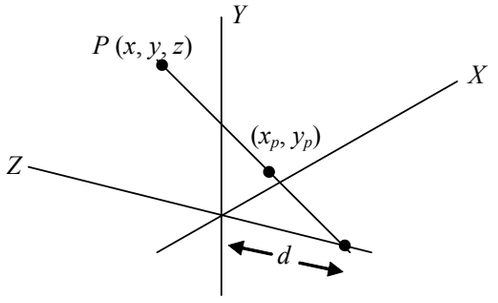


Fig. 11.13 *Perspective Projection from y-axis*

From Figure 11.13, you have,

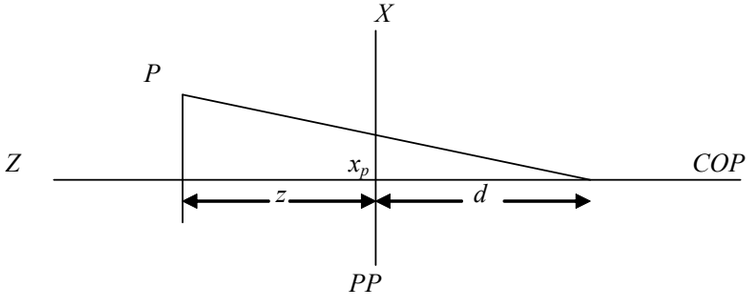


Fig. 11.14 *Perspective Projection*

Now,

$$x/(z+d) = \frac{x_p}{d}$$

$$x_p = x[\frac{d}{z+d}] \text{ and}$$

$$x_p = x/(z/d+1)$$

The same calculation is done for y (look down the x-axis) to get $y_p = y/(z/d + 1), z_p = 0$. You can represent this in matrix form by using homogeneous coordinates as follows:

$$[xh \quad yh \quad zh \quad w] = [x \quad y \quad z \quad 1]\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{d} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where, $xh = x, yh = y, zh = 0, w = (z/d) + 1$

Thus, the points on the projection plane are $[xp \quad yp \quad zp \quad 1] = [\frac{xh}{w} \quad \frac{yh}{w} \quad \frac{zh}{w} \quad 1]$.

This leads to the same x_p, y_p as before. The major problem with perspective transformation is that it does not safeguard straight lines or planes; that is, straight lines cannot be transformed into straight lines. Let us look at an example of a three-dimensional line in object space from:

P_1 ($x_1 = 2, y_1 = 5, z_1 = 6$) to P_2 ($x_2 = 8, y_2 = 7, z_2 = 12$)

In parametric form, this line is represented as:

$x(t) = 2 + 6 \times ty(t) = 5 + 2 \times tz(t) = 6 + 6 \times t$

Let us apply an arbitrary value of t (for example, $t = 0.40$) and compute the x, y, z values:

$x = 2 + 2 \times 0.40 = 4.40y = 5 + 2 \times 0.40 = 5.80$, so $P_i(t = 0.40) = (4.40, 5.80, 8.40)$ $z = 6 + 6 \times 0.40 = 8.40$

Let us now perform the perspective transformation (assume $d = 10.0$) for P_1, P_i, P_2. Thus, you get:

$P_1(x = 1.25, y = 3.125, z = 6.0)$,

$P_i(x = 2.39, y = 3.15, z = 8.4)$ and $P_2(x = 3.64, y = 3.18, z = 12.0)$

If this is still a straight line, then all three coordinates of point P_i must have the same value of the parameter t. So for x you get,

$2.39 = 1.25 + t \times (2.39) \Rightarrow t = 0.48$

NOTES

For *y,* you get $3.15 = 3.12 + t \times (0.57) \Rightarrow t = 0.48$

For *z,* you get $t = 0.40$ which is unchanged, therefore, the points are not collinear. For maintaining linearity, a perspective depth transformation can be done as:

$$Zp = Z / (D + Z)$$

Then, for point 1,

$$Zp = 6 / (10 + 6) = 0.375$$

For point 2,

$$Zp = 12 / (10 + 12) = 0.545\text{For point } i$$

$$Zp = 8.4 / (10 + 8.4) = 0.457$$

Now check with *t* value for point *i*,

$0.457 = 0.375 + t * (0.170) = 0.48$

This is the same value of *t* that you calculated for point *i*, *x* and *y*. Therefore, points 1, 2 and *i* are still collinear after applying perspective depth transformation.

The point to be noted is that the relative *z* depth values remain unchanged, that means if $Z_2 > Z_1$, then there exists a relation $Z_2 / (Z_2 + d) > Z_1 / (Z_1 + d)$ as follows:

$Z_2 > Z_1 Z_2 \times d > Z_1 \times d$ (by multiplying both the sides by *d*)

$(Z_1 \times Z_2 + Z_2 \times d) > (Z_1 \times Z_2 + Z_1 \times d)$ (by adding $Z_1 \times Z_2$ to both sides)

$Z_2 \times (Z_1 + d) > Z_1 \times (Z_2 + d) Z_2 / (Z_2 + d) > Z_1 / (Z_1 + d)$

For $Zp = Z / (Z + d) \Rightarrow 0$ if *d* is greater than *Z* and $\Rightarrow 1.0$ if *Z* is greater than *d*. Therefore,

$0.0 \Leftarrow Zp \Leftarrow 1.0$

Thus, to maintain linearity, you have to transform *Z* as well as *X* and *Y*.

Check Your Progress

1. What is global coordinate system?
2. What is axonometric orthographic projection?

11.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Global Coordinate System, also called the World Coordinate System is the principal frame of reference in which all other coordinate systems are defined.
2. Orthographic projections that show more than one side of an object are called axonometric orthographic projections.

11.5 SUMMARY

NOTES

- The Viewing Coordinate System locates objects in three-dimensional space relative to the location of the observer. To establish the viewing coordinate reference frame first define an eyepoint or view reference point.
- A Local Coordinate System, or a Modeling Coordinate System is used to define the geometry of an object independently of the global system.
- Global Coordinate System, also called the World Coordinate System is the principal frame of reference in which all other coordinate systems are defined.
- The Normalized Device Coordinates are used to map world coordinates in a device independent two-dimensional pseudo space within the range 0 to 1 for each of x and y before final conversion to specific device coordinates.
- Parallel projection shows the true image, size and shape of an object. The angle made by the direction of projecting lines with the projection plane or view plane is determined.
- A perspective projection can be viewed as the projection that has a centre of projection at a finite distance from the plane of projection.
- Orthographic projections that show more than one side of an object are called axonometric orthographic projections.

11.6 KEY WORDS

- **Local Coordinate System:** It is used to define the geometry of an object independently of the global system.
- **Projection:** It means the transformation of a three-dimensional (3D) area into a two-dimensional (2D) area.
- **Isometric Projection:** It refers to the projection in which the projection plane is allowed to intersect the *x*, *y* and *z* axes at equal distances and the plane normal to the projection has equal angles with these three axes.

11.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Discuss the coordinate system in viewing pipeline.
2. What do you understand by viewing pipeline?

Long Answer Questions

1. Explain the parallel and isometric projection.

NOTES

2. What is oblique and perspective projection? Explain.
3. Prove that the perspective projection of a line segment is equal to the line segment between the perspective projection of endpoints.
4. Find the transformation matrix for oblique projection onto *xy*-plane.

11.8 FURTHER READINGS

Hearn, Donal and M. Pauline Baker. 1990. *Computer Graphics*. New Delhi: Prentice-Hall of India.

Rogers, David F. 1985. *Procedural elements for Computer Graphics*. New York: McGraw-Hill.

Foley, D. James, Andries Van Dam, Steven K. Feiner and John F. Hughes. 1997. *Computer Graphics Principles and Practice*. Boston: Addison Wesley.

Mukhopadhyay, A. and A. Chattopadhyay. 2007. *Introduction to Computer Graphics and Multimedia*. New Delhi: Vikas Publishing House.

BLOCK - V
VISIBLE SURFACE DETECTION METHODS AND ANIMATION

NOTES

UNIT 12 CLASSIFICATION

Structure

12.0 INTRODUCTION

A major consideration in the generation of realistic graphics display is identifying those parts of a scene that are visible from a chosen viewing position. There are many approaches that we can take to solve this problem and numerous algorithms have been derived for efficient identification of visible objects of different types of applications. Some methods require memory, some involve more processing time, and some apply only to special types of objects. Deciding upon a method for a particular application can depend on such factors as a complexity of the scene, types of objects to be displayed, available equipment, and whether static or animated displays are to be generated. The various algorithms are referred to as visible-surface detection methods or sometimes called hidden-surface elimination methods; although there can be a subtle difference between identifying visible surfaces and eliminating hidden surfaces.

12.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the general principles of hidden lines and surfaces

- Explain the Z-buffer method and the scan-line method
- Analyse the depth-sorting method
- Familiarize yourself with the Octree Method

NOTES

12.2 BACK-FACE DETECTION

It is an object space method which compares objects and parts of objects to find the visible surfaces. It is also known as **plane equation method**. Consider an example of a triangular surface whose visibility needs to be decided. The idea behind is to check whether the triangle is facing away from the viewer or not. If it is like that then discard it from the current frame and move to the next. The surface has the normal vector. When the normal vector is in the direction of center of projection, it means a front face and viewer can see it and vice versa.

12.3 Z-BUFFER METHOD (DEPTH-BUFFER METHOD)

The z-buffer or depth buffer is the simplest of the visible surface or hidden surface algorithms and is an image space algorithm. The z-buffer is a simple extension of the frame buffer idea. A frame buffer is used to store the attributes of each pixel in the image space. The z-buffer is a separate depth buffer used to store the z-coordinates, or depth of every visible pixel in the image space. In use, the depth or z-value of a new pixel to be written to the frame buffer is compared to the depth of that pixel stored in the z-buffer. If the comparison indicates that the new pixel is in front of the pixel stored in the frame buffer, then the new pixel is written to the frame buffer and the z-buffer updated with new z-value. If not, no action is taken. Conceptually, the algorithm is a search over x, y for the largest value of z(x, y). This algorithm is frequently implemented for polygonally represented scenes and also applicable for any object for which depth and shading characteristics can be calculated. Scenes can contain mixed object types and may be of any complexity.

The algorithm

Step 1: Set the frame buffer to the background intensity or colour.

Step 2: Set the z-buffer to the minimum z-value.

Step 3: Scan converts each polygon in arbitrary order.

Step 4: For each pixel (x, y) in the polygon, calculate the depth $z(x, y)$ at that pixel.

Step 5: Compare the depth $z(x, y)$ with the value stored in the z-buffer at that location

Step 6: if $z(x, y) > z$-buffer(x, y), then write the polygon attributes to the frame buffer and replace z-buffer (x, y) with $z(x, y)$, otherwise no action is taken.

12.4 SCAN-LINE METHOD

NOTES

Scan-line visible surface and visible line algorithms are extensions of scan polygon techniques. Scan-line algorithms reduce the visible line/ visible surface problem from three dimensions to two. A scan plane is defined by the viewpoint at infinity on the positive z-axis and a scan-line, as shown in Figure 12.1.

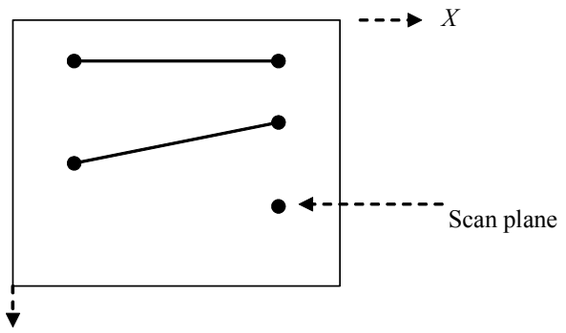


Fig. 12.1 *Illustration of a Scan Plane*

The intersection of the scan plane and the three-dimensional scene defines a one-scan-line-high window. Figure 12.2 shows the intersection of the plane with polygons. The figure points that the visible surface problem is reduced to deciding which line segment is visible for each point on the scan-line. At first glance it might appear that the ordered edge list algorithm is directly applicable. However Figure 12.2 shows that this gives incorrect results. For example, for the scan-line shown in Figure 12.1, there are four active edges on the active edge list. The intersections of these edges with the scan-line are shown by the small dots Figure 12.2. Extracting the intersections in pairs activates the pixels between 1 and 2 and between 3 and 4.

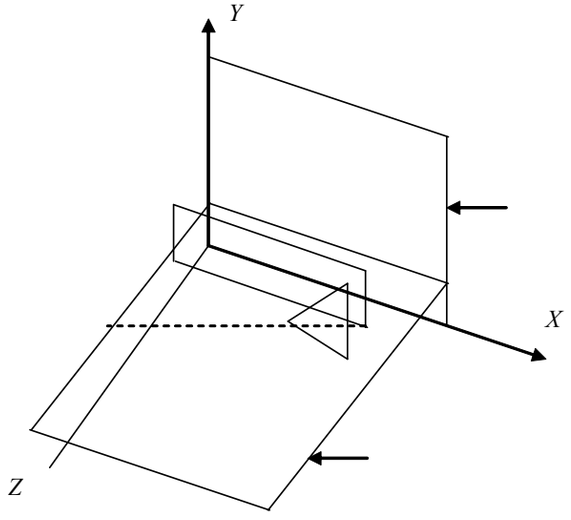


Fig. 12.2 *Another Illustration of Scan Plane*

The pixels between 2 and 3 are not activated. The result is incorrect. A 'hole' is left on the scan-line, where it intersects two polygons.

NOTES

12.5 DEPTH-SORTING METHOD

By using the image-space and object-space operations, the depth-sorting method performs the following basic functions:

(i) Surfaces are sorted in order of decreasing depth

(ii) Surfaces are scan converted in order, from starting with the surface of greatest depth

Sorting operations are taken in both image and object space, and the scan conversion of the polygon surfaces is only performed in the image space. This method for solving the hidden-surface problem is often called the 'painter's algorithm.' In this method first we sort the surfaces according to their distance from the view plane. The intensity values for the outermost surface are then entered into the refresh buffer. Taking each succeeding surface in turn, the surface can be painted with intensities on to the frame buffer over the intensities of the previously processed surfaces.

According to the depth-sorting method, painting polygon surfaces onto the frame buffer is carried out in several steps. Suppose we are viewing along the z direction, surfaces are ordered on the first path according to the largest z value on each surface. Surface S with the greatest depth is then compared to the other surfaces in the list to determine whether there are any overlaps in depth. If no depth overlaps occur, then S is scan converted. Figure 12.3 demonstrates the two surfaces that overlap in the xy plane but they do not have depth overlap.

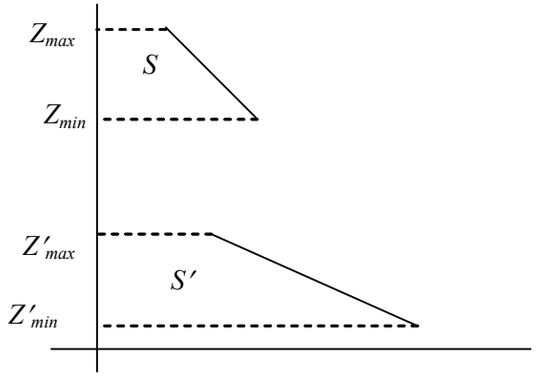


Fig. 12.3 *Two Surfaces having no Depth Overlap*

This process is then repeated for the next surface in the list. As long as no overlap is found, each surface is processed in depth order until all have been scan converted. If a depth overlap is found at any point in the list, then we need to make some

NOTES

additional comparisons to determine whether any of the surfaces should be reordered. We make the following tests for each surface that overlaps with S. If any one of these tests is true, no reordering is necessary for that surface. The tests are listed in order of increasing difficulty.

(i) For two surfaces bounding rectangles in the *xy*-plane do not overlap.

(ii) Relative to the viewing position surface S is completely behind the overlapping surface.

(iii) Relative to the viewing position the overlapping surface is completely in front of S.

(iv) The projections of the two surfaces on to the view plane do not overlap.

We perform these tests in the order listed and proceed to the next overlapping surface as soon as we find one of the tests is true. If all the overlapping surfaces pass at least one of these tests, then none of them is behind *S*. No rendering is then necessary and then *S* is scan converted. The first test is performed in two parts. First we check for overlaps in the *x*-direction, and after this we check for overlaps in the *y*-direction. If either of these directions represents no overlapping, the two planes can not obscure one another. An example of the two surfaces which overlap in *z*-direction but not in *x*-direction is represented in Figure 12.4.

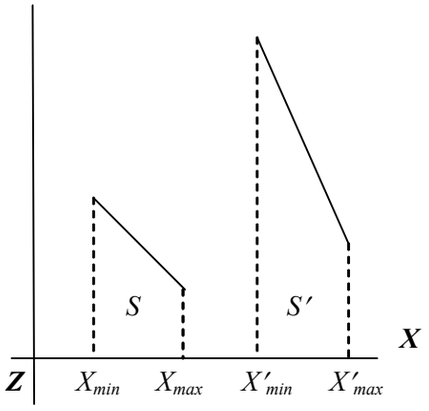


Fig. 12.4 *Two Surfaces having Depth Overlap but no Overlap in x-direction*

With an 'inside-outside' polygon test we can perform tests 2 and 3, i.e., we substitute the coordinates for all vertices of *S* into the plane equation for the overlapping surface and check the sign of the result. If the plane equation are set up so that the outside of the surface is toward the viewing position, then *S* is behind *S'* if all the vertices of *S* are inside *S'*.

NOTES

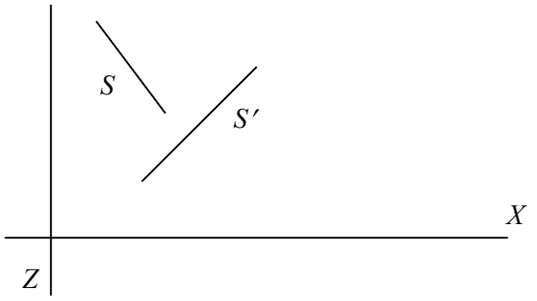


Fig. 12.5 *Surface S is Computed behind the Overlapped Surface S′*

If all vertices of S are outside *S′*, then *S′* is completely in front of S. Figure 12.6 shows an overlapping surface *S′* which is completely in front of S, but surface S is not completely inside *S′* (test 2 is failed).

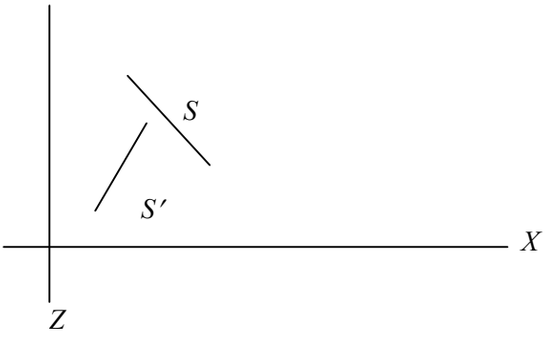


Fig. 12.6 *The Overlapping Surface S′ is Completely Outside of Surface S*

If tests 1 to 3 have all failed, then we try test 4, in which we check for the intersections between the bounding edges of the two surfaces using line equations in the xy plane. As shown in the following figure the two surfaces can or can not intersect although their coordinate extents overlap in the *x*, *y* and *z* directions.

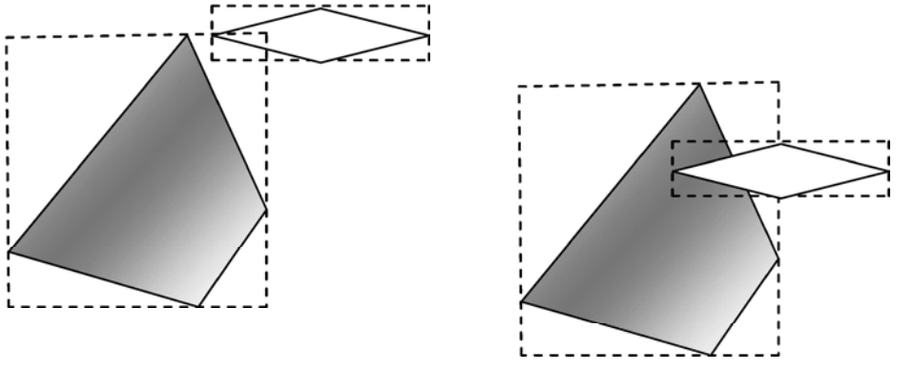


Fig. 12.7 *Two Surfaces having Overlapping Bounding Rectangles in xy-plane*

Should all four tests fail with a particular overlapping surface *S′*, we interchange surfaces S and *S′* in the sorted list. Figure 12.8 shows the example of two surfaces that would be reordered with this procedure.

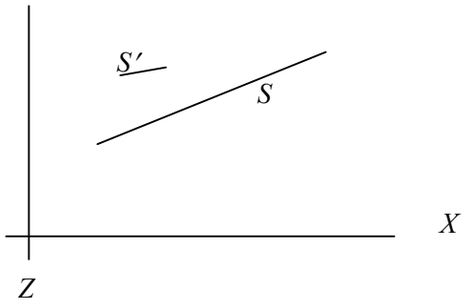


Fig. 12.8 *Surface S with more Depth but Obscure Surface S′*

At this point we still do not know for certain that we have found the farthest surface from the view plane. Figure 12.9 shows a situation in which we would first interchange S and *S′*.

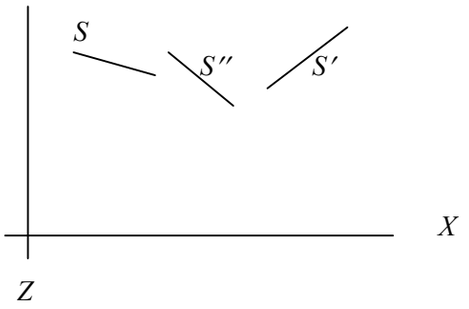


Fig. 12.9 *Three Surfaces Entered into Sorted List (S, S′, S″) of Surfaces*

But since *S″* obscures a part of *S′*, we need to interchange *S″* and *S′* so that we can get the three surfaces into the correct depth order. So we need to repeat the testing process for each surface that is reordered in the list.

12.6 BSP-TREE METHOD

The *Binary Space Partitioning Tree* method is very effective in determining visibility relationship among a static group of 3D polygons as seen from an arbitrary viewpoint.

Following is the procedure to build a BSP Tree in the object space.

I. Select a polygon (arbitrary) as the root of the tree i.e. the 1st partition plane.

II. Partition the object space into two ***half-spaces*** (inside & outside of the partition plane determined by the normal to the plane); some object polygons lie in the rear half while the others in the front half w.r.t the partition plane.

III. If a polygon is intersected by the partition plane, split it into two polygons so that they belong to different half-spaces.

IV. Select one polygon in the root's front as the left node or child and another in the root's back as the right node or child.

V. Recursively subdivide spaces considering the plane of the children's as partition planes until a subspace contains a single polygon.

NOTES

Thus we see that BSP tree's internal nodes are the partitioning planes (polygon objects) with left nodes representing front objects and right nodes the back objects. The leaves represent regions in space.

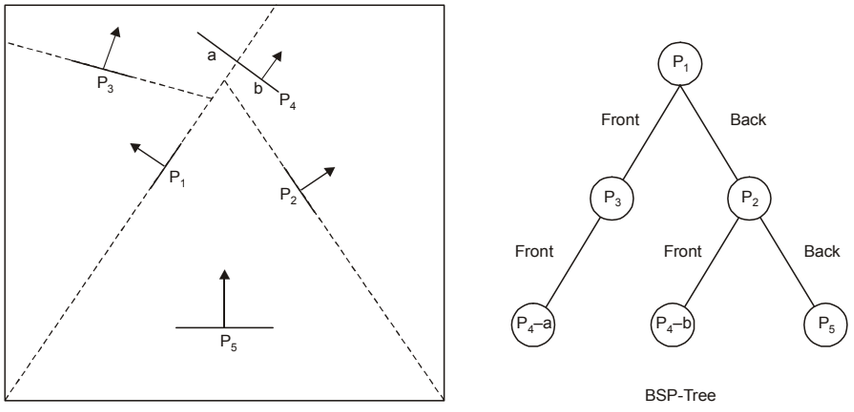


Fig. 12.10 *Binary space partitioning and formation of BSP-Tree*

When the BSP Tree is complete, following the principle of Painter's algorithm the tree is processed by selecting surfaces for display in back-to-front order. If the viewer is in the root polygons front half space, then the algorithm first displays all polygons in the roots rear half space (that too in back-to-front order recursively at each node) then the root itself and finally all polygons in its front half space (in back-to-front order recursively for each node).

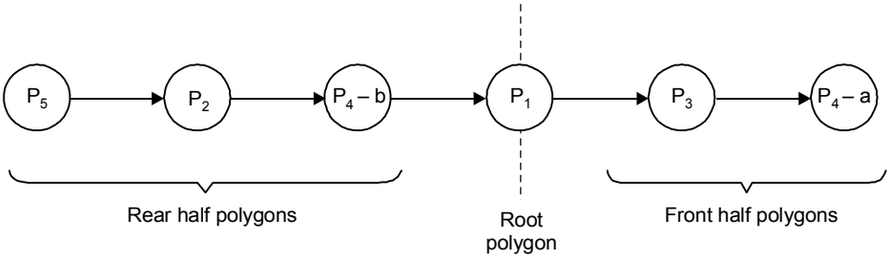


Fig. 12.11 *Order of processing polygons in the BSP Tree as shown in Fig.6.9*

12.7 AREA SUB-DIVISION

This method works in image space as it is concerned with what is displayed on the screen. It considers a window in image space and seeks to determine if the window is empty or if the content of the window is projection of a single visible surface. If not, the window is subdivided into smaller and smaller rectangles or sub-windows

NOTES

until either the content of a sub-window is simple enough to display or the sub-window size is at the limit of the display resolution (i.e., a pixel with a single intensity or color).

Starting with the full display screen as the initial window area, the algorithm divides an area at each stage into four smaller areas (sub-windows), thereby generating a quad tree.

This process basically exploits *area coherence* or the fact that adjacent areas (pixels) in both the x and y directions tend to be similar. As a result sufficiently small areas of an image will be contained in atmost a single visible polygon.

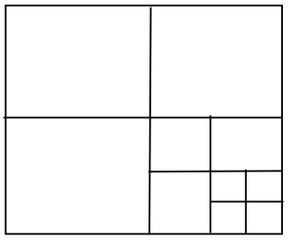


Fig. 12.12

In particular, a polygon P w.r.t a given window area A is

1. ***Disjoint*** if P is totally outside A
2. ***Contained*** if P is totally inside A
3. ***Intersecting*** if P intersects A
4. ***Surrounding*** if P completely contains A

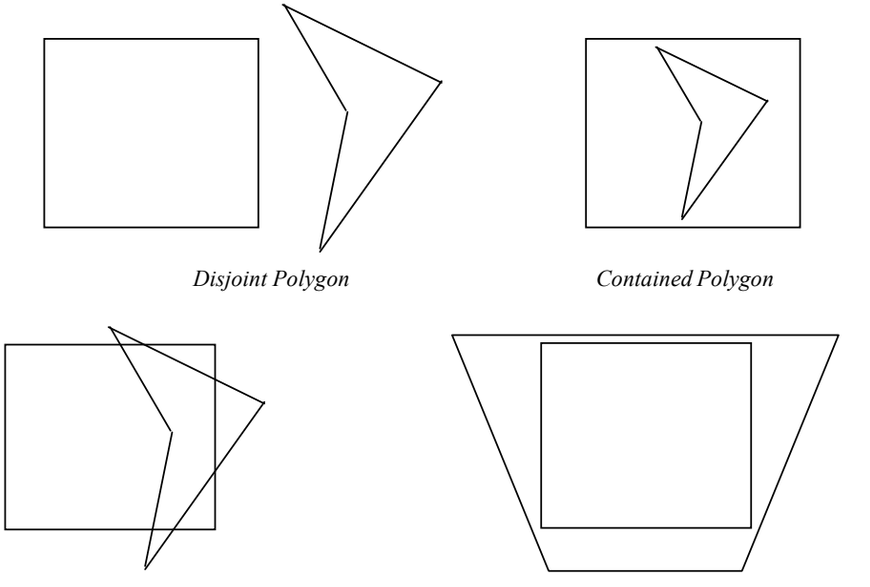


Fig. 12.13

NOTES

While trying to find the potentially visible polygons (**PVP**) w.r.t. a given window area disjoint polygons and the part of an intersecting polygon that is outside the area are ignored as being invisible. Only the contained polygons, portion of the intersecting polygons inside the area and surrounding polygons are potentially visible.

The basic steps of the algorithm can be sequentially listed as following:

Step 1: Initialize the window area to be the whole screen.

Step 2: Create a potentially visible polygons list (**PVPL**) with the PVPs sorted on z_{min}. Place the PVPs in their appropriate categories. Remove polygons hidden by a surrounding polygon and remove disjoint polygons.

Step 3:

(*i*) If all polygons are disjoint to the area, set all the pixels to the background color.

(*ii*) If PVPL has only a single polygon classified as contained, then fill the area of the window outside the polygon with the background color; fill the contained polygon with appropriate color.

(*iii*) If the PVPL has only one polygon which is a surrounding one then fill the area with the appropriate color of the surrounding polygon. (The case is same if there is a surrounding polygon which hides all polygons in the PVPL).

(*iv*) If the PVPL has only one polygon, which is an intersecting polygon, then fill the area of the window outside the polygon with background color, fill the portion of the intersecting polygon within the window area with appropriate color.

(*v*) If the area is a pixel at (x, y) and neither of the above cases applies compute the z co-ordinate z (x, y) at (x, y) of all polygons in the PVPL. Set the pixel at (x, y) to the color of the polygon with the smallest z co-ordinate.

Step 4: If none of the five cases in step 3 applies then subdivide the area into fourths. For each subdivided area go to step 2.

For rectangular windows, ***bounding box*** or ***minimax*** tests can be used to determine whether a polygon is disjoint w.r.t a window. If x_L, x_R, y_B, y_T define the four edges of a window and $x_{min}, x_{max}, y_{min}, y_{max}$, the edges of the bounding box of a polygon, then the polygon is disjoint if

$$x_{min} > x_R \text{ or } x_{max} < x_L \text{ or } y_{min} > y_T \text{ or } y_{max} < y_B$$

The polygon is contained within the window if the bounding box is contained within the window, i.e.

$$x_{min} \geq x_L \text{ and } x_{max} \leq x_R \text{ and } y_{min} \geq y_B \text{ and } y_{max} \leq y_T$$

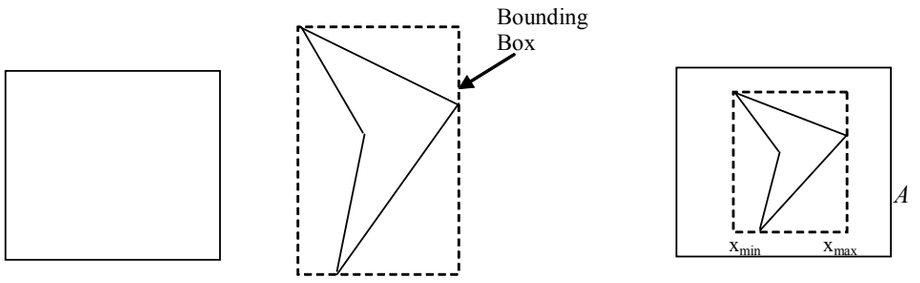


Fig. 12.14

If simple substitution test can be used to identify an intersecting polygon. The coordinates of the window vertices are substituted into a test function say, $f = y - m\,x - c$ where the equation of the line defining a polygon edge is $y = m\,x + c$. If the sign of the test function is the same for each window vertex, then all the vertices lie on the same side of the polygon edge and so there is no intersection. If the signs do not match then the edge intersects the window. If none of the polygon edge intersects the window, the polygon is either disjoint or surrounds the window.

There are cases of typical disjoint and surrounding polygons which are not trapped by the above mentioned tests. More complex tests like *infinite line test* and *angle continuing test* are required. It is, however, not necessary to identify either contained or intersecting polygons. Area subdivision will eventually make contained or intersecting polygons either disjoint or surrounding polygons. Any remaining conflicts are resolved at the pixel level.

12.8 OCTREE METHOD

Octrees are three dimensional analogs of quad-trees. When an octree is used for the viewing volume, hidden-surface elimination is achieved by projecting octree nodes onto the viewing surface in a front-to back order as shown in Figure 12.15. Front face of a region of space is formed with octants 0, 1, 2 and 3. Surfaces in front of these octants are visible to the viewer. Any surface towards the rear of the front octants or in the back octants (4, 5, 6 and 7) may be hidden by the front surface.

Back surfaces are eliminated, for the viewing direction given in Figure 12.14 by processing data elements in octree nodes in the order 0, 1, 2, 3, 4, 5, 6, 7. This results in a depth–first traversal of the octree, so that the nodes representing octants 0, 1, 2, and 3 for the entire region are visited before the nodes representing octants 4, 5, 6 and 7. Similarly the nodes for the front four sub-octants of octants 0 are visited before the nodes for the form back sub-octants. The traversal of the octree continues in this order for each octant subdivision. If a colour value is considered in octree nodes, the pixel area in the frame buffer corresponding to this node is assigned that colour value only if no value has previously been stored in this area.

NOTES

In this way, only the front colours are loaded into the frame buffer. Nothing is loaded if an area is void. Any node that is found to be completely obscured is eliminated from further processing, so that its sub-trees are not accessed. Different views of objects represented as octrees can be obtained by applying transformations to the octree representation that reorient the object according to the view selected. We assume that the octree representation is always setup so that octants 0, 1, 2, and 3 of a region form the front face.

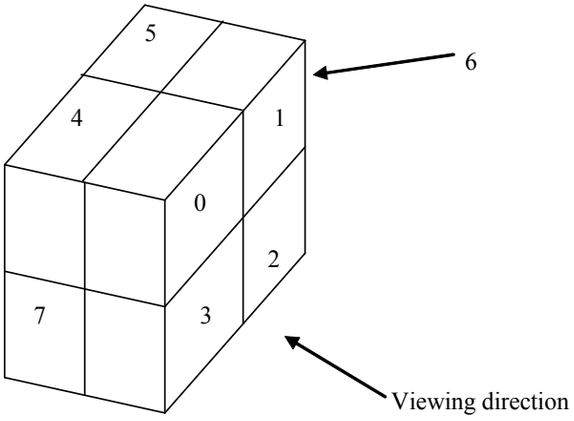


Fig. 12.15 *The Numbered Octants of a Region*

An octree first maps itself onto a quad-tree of visible areas by traversing octree nodes from front to back in a recursive procedure. Then the quad-tree representation for the visible surface is loaded into the frame buffer. Figure 12.16 depicts the octants in a region of space and the corresponding quadrants on the view plane.

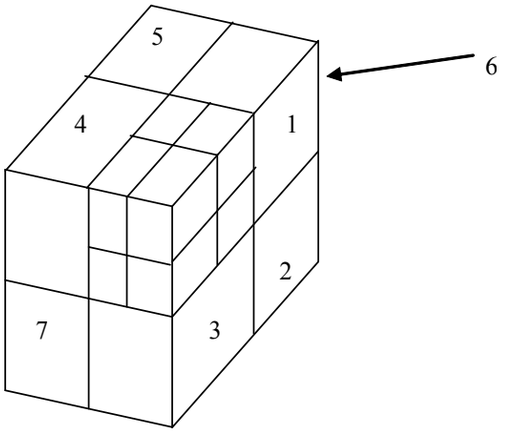


Fig. 12.16 *Octants Division for a Region of Space and the Corresponding Quadrant Planes*

Contribution to quadrant 0 comes from octants 0 and 4. Colour values in quadrant 1are obtained from the surfaces in octants 1 and 5, and the values in each of the

other two quadrants are generated from the pair of octants aligned with each of these quadrants.

Check Your Progress

1. What is the role of scan-line algorithms?
2. What are the basic functions performed by the depth-sorting method?
3. What is the significance of BSP tree method?
4. What are octrees?

NOTES

12.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Scan-line algorithms reduce the visible line/ visible surface problem from three dimensions to two.
2. The depth-sorting method performs the following basic functions:
   (i) Surfaces are sorted in order of decreasing depth
   (ii) Surfaces are scan converted in order, starting with the surface of greatest depth.
3. The Binary Space Partitioning Tree method is very effective in determining visibility relationship among a static group of 3D polygons as seen from an arbitrary viewpoint.
4. Octrees are three dimensional analogs of quad-trees.

12.5 SUMMARY

- **Back-Face detection**, also known as **Plane Equation method,** is an object space method in which objects and parts of objects are compared to find out the visible surfaces.
- The z-buffer or depth buffer is the simplest of the visible surface or hidden surface algorithms and is an image space algorithm. The z-buffer is a simple extension of the frame buffer idea.
- Scan-line visible surface and visible line algorithms are extensions of scan polygon techniques. Scan-line algorithms reduce the visible line/ visible surface problem from three dimensions to two.
- Sorting operations are taken in both image and object space, and the scan conversion of the polygon surfaces is only performed in the image space.
- The Binary Space Partitioning Tree1 method is very effective in determining visibility relationship among a static group of 3D polygons as seen from an arbitrary viewpoint.

- Octrees are three dimensional analogs of quad-trees. When an octree is used for the viewing volume, hidden-surface elimination is achieved by projecting octree nodes onto the viewing surface in a front-to back order.

NOTES

12.6 KEY WORDS

- **Depth Field:** It is a field in the A-Buffer method which it stores a positive or negative real number.
- **Intensity Field:** It is a field in the A-Buffer method which stores surface-intensityinformation or a pointer value.
- **Octrees:** These are three dimensional analogs of quad-trees.

12.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Explain the depth-buffer method for hidden surface detection.
2. Explain the scan-line method for back-face removal.
3. Explain Painter's algorithm with a suitable example.
4. What are the basic disadvantages of the Z-buffer method?

Long Answer Questions

1. Explain the depth-sorting method with a suitable example.
2. What is the difference between Painter's Algorithm and Scan-line Algorithm?
3. Explain the quad tree and the Octree with a suitable diagram.
4. Write the hidden line methods in detail.

12.8 FURTHER READINGS

Hearn, Donal and M. Pauline Baker. 1990. *Computer Graphics*. New Delhi: Prentice-Hall of India.

Rogers, David F. 1985. *Procedural elements for Computer Graphics*. New York: McGraw-Hill.

Foley, D. James, Andries Van Dam, Steven K. Feiner and John F. Hughes. 1997. *Computer Graphics Principles and Practice*. Boston: Addison Wesley.

Mukhopadhyay, A. and A. Chattopadhyay. 2007. *Introduction to Computer Graphics and Multimedia*. New Delhi: Vikas Publishing House.

UNIT 13 COMPUTER ANIMATION

NOTES

Structure

13.0 INTRODUCTION

In this unit, you will learn about computer animation. Computer animation is theprocess used for generating animated images by using computer graphics. The termcomputer generated imagery includes both static scenes and dynamic images whilethe term computer animation is used only for moving images. Nowadays computeranimation typically uses 3-D computer graphics, even though 2-D computer graphicsare still used for specific styles, low bandwidth and faster real time renderings. It canbe used for creating animation using the computer and also for another medium, suchas film. Fundamentally, computer animation is a digital successor to the stop motiontechniques used in traditional animation with 3-D models and frame-by-frame animation of 2-D illustrations. Computer generated animations are easily controllablethan any other physical/manual processes. To create the illusion of movement, animage is displayed on the computer screen and repeatedly replaced by a new imagethat is similar to it but advanced slightly in time generally at a rate of 24 or 30 frames/second and is similar to the technique used in television and motion pictures.

13.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain design of animation sequences
- Describe general animation functions in a computer
- Discuss raster animations

13.2 DESIGN OF ANIMATION SEQUENCES

NOTES

Literally speaking, to *animate* is to bring to life, i.e., to put something into action. Animation makes graphics more realistic by imparting motion and dimension to an inanimate object. Intuitively though we think of animation synonymous with motion, technically speaking, it covers all changes that have a visual effect. Thus it may include time varying position (motion dynamics), shape, size, color, texture (update dynamics) of an object and also changes in lighting, camera position, focus, etc. Animation is referred as the speedy exhibit of a sequence of 2-D or 3-D metaphors of model locations to create an illusion of movement. The consequence is an optical illusion of motion because of the phenomenon of doggedness of vision. It can be produced and demonstrated in numerous ways. Animation adds to graphics the dimensions of time, which tremendously increase the potential of transmitting the desired information. In order to animate an object, the animator should specify directly or indirectly how the object has to move through time and space. The most common method of presenting animation is as a motion picture or video program, although there are other methods also.

With advancement in computer aided techniques, today animation is extensively used in Entertainment (games and movies), Educational and Training presentations, Advertising, Internet and Process simulation. Process simulation through animation is very useful in visualization of functioning and stages of operations of industrial products (like a gear or motor) or gradual transformations in a complex process, such as changing atomic structures in a chemical reaction or distortion of structures under dynamic forces.

Computer animation includes assortment of techniques, the unifying factor being that the 2-D and 3-D animations are created digitally on a computer. This animation takes less time than previous traditional animation methodology. 2-D animation figures are created and/or edited on the computer using 2-D bitmap graphics or created and edited using 2-D vector graphics. 3-D animation is digitally modeled and manipulated by an animator. 2-D animation techniques tend to focus on image manipulation while 3-D techniques usually build virtual worlds in which characters and objects move and interact. Basically, 3-D animation can create images that seem real to the viewer.

Traditional animation also called cel animation or hand-drawn animation was the process used for most animated films of the 20th century. To create the illusion of movement, each drawing differs slightly from the one before it. The animators' drawings were traced or photocopied onto transparent acetate sheets called **cels**, which were filled in with paints in assigned colors or tones on the side opposite the line drawings. The completed character cels were then photographed one-by-one onto motion picture film against a painted background by a rostrum camera. Today, animators' drawings and the backgrounds are either scanned into

or drawn directly into a computer system. Various software programs are used to color the drawings and simulate camera movement and effects.

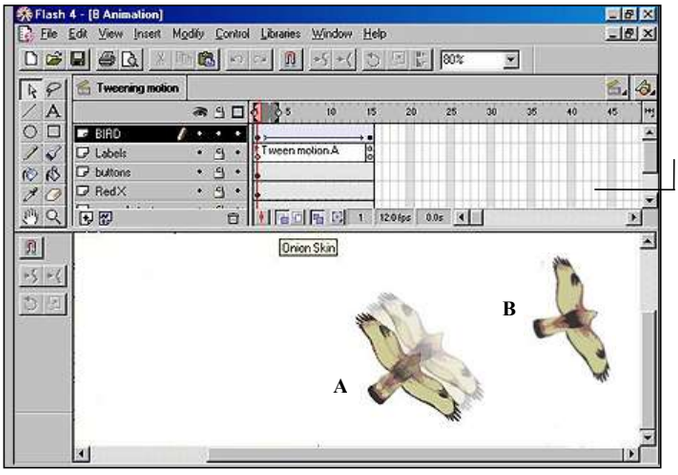
A computer animation sequence can be set by specifying the storyboard, the object definitions and the image frames. The ***storyboard*** is an outline of action. It could consist of rough sketches of motion sequence or it could be a list of basic events that are to take place. ***Object definitions*** are given for each participating object in terms of their shape and movement. The still image frames are either drawn manually or is computer generated to simulate motion sequence of animating objects. The illusion of movement is created by playing 15-20 numbers of such still images frames with small changes made to each one per second. The eyes retain an image for sometime and to allows the brain to connect the frames in an uninterrupted sequence. An object observed by the human eye remains mapped on the eyes retina for a short time say about 1/20th of a second and is known as the ***persistence of viewing.*** Animation basically exploits these biological phenomena where a series of images are changed very slightly but very rapidly, one after the other, to seemingly blend together into a visual illusion of movement. In traditional animation, as many as 30 FPS (Rames Per Second) might be used to give a smoother appearance at high speeds.

Cel Animation

Classically, picture frames depicting animated sequence were drawn manually. For this the ***onionskin*** technique which is popularly known as c*el animation* technique is mostly adopted. By drawing on a onionskin-like translucent paper called 'cel', with a light source beneath the drawing surface, an animator can see the position of an object on one page, while drawing it in a new position on the page above. Only the moving elements on the cel are redrawn for each frame while the fixed part (usually the background) is only made once.

This concept of cel has been implemented in the digital media in the form of ***layer***. Many animation software applications offer translucent drawing layers. In some of the software, such as Macromedia Flash the layers are shown progressively more opaque, to assist in identifying the stacking order of the layers. The image frames of an animated sequence can be made by combining a background layer, which remains static, with one or more animation layers, in which any changes that take place between frames are made. To take a simple example, suppose we wish to animate the sailing of a boat across the river. The first frame could consist of a background layer containing the river field and a foreground layer with an image of the boat. To create the next frame, these two layers are copied in the frame and then using the move tool, displace only the image of the boat up to a small distance. By continuing in this way, sequence depicting the smooth movement of the boat across the background can be produced.

NOTES



Here you see the Macromedia Flash time line. You can see layers of objects stacked up for display as a single image. The numbered columns each represent a frame, with time marching on from left to right.

In this animation, as the bird flies from A to B, two intermediate frames are displayed with onion skin effect. Thus all the in-betweens can be displayed in onion skin mode and can be edited on frame by frame basis if required.

Fig. 13.1 *Cel Animation*

The form of animation based on objects movement only (no change of other properties) is called ***sprite animation***. The moving objects are referred to as *sprites*. Instead of storing changes of sprite position from frame to frame the change values can also be generated by computer programs.

13.3 GENERAL COMPUTER ANIMATION FUNCTIONS

Animation packages provide special functions for designing the animation and processing the individual objects. Several well-suited steps are defined in the animation packages which can be used in the development of animation sequences in a computer system. These include object manipulation, rendering camera motions and the generation of in-betweens. One important function in animation packages is to store and manage the object database. Different object shapes and associated parameters are specifically stored and updated in the database. Additional object functions enhance motion generation and object rendering methods. Motion can be generated as per the specified constraints with the help of 2-D or 3-D transformations. Standard motions are zooming, panning and tilting. Another typical function simulates camera movements. Standard functions are applied to identify visible surfaces using the rendering algorithms. Finally, the given specification for the keyframes and the in-betweens can be automatically generated. A sequence of static images presented in a quick succession appears as continuous flow.

In simple terms, any type of moving image that can be produced through the personal computer is referred as 3-D computer animation, a method to generate the optical illusion of a range of movements. It is typically termed as Computer Generated Imagery or CGI and uses 3-D animation software.

13.4 RASTER ANIMATIONS

NOTES

There are two main types of computer animation, vector based and raster based. Vector animation is based on simple geometric shapes, while raster animation is created using more detailed images similar to photos or paintings. Raster animation is the most common animation technique. The frames are copied very fast from off-screen memory to the frame buffer. Copying can be applied to complete frames or only parts of the frame which contain some movement. In raster animation procedure, a part of the frame in the frame buffer needs to be erased while the static part of the frame is re-projected as a whole and the animated part is over projected.

Raster based animation frames (and all the related raster images) are made up of individual pixels. Each of these pixels contain information about the color and brightness of that particular spot on the image. This is somewhat similar to the concept of pointillism in painting, with the sum of the points making up the totality of the picture or frame. Raster animation is used for depicting realistic representations of people, animals or places, rather than the more stylized, anime-style animation you might get with vector graphics. Raster animation is also used to create animation for logos and banners which are based on photos or drawings.

One of the problems associated with the creation of raster based animations on a computer is the enormous amount of computer power that is consumed while creating them. A major difficulty with working with raster based animations or images is that they are not infinitely enlargeable, for example if you want to create a raster based animation at a certain size, say 400 × 300, it is not possible to enlarge it to any significant extent without any loss of resolution in the image. Raster animation starts with raster images, moving them on the screen as blocks of pixels. If the computer needs to enlarge an image, it blows it up by simple magnification. Figure 13.2 displays the raster views and spatial entities for selected cels used in animation.

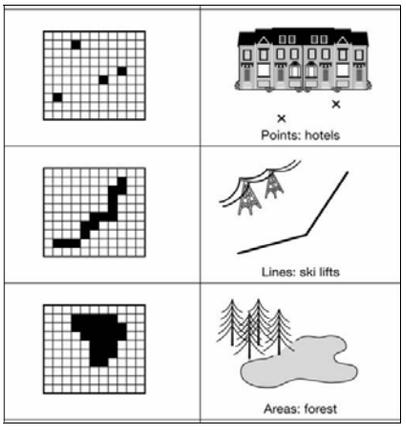


Fig. 13.2 *Raster Views and Spatial Entities*

NOTES

Raster animation is used for depicting realistic representations of people, animals or places. This animation is also used to create animation for logos and banners based on photos or drawings. Raster based animation frames are made up of individual pixels. Each pixel contains information about the color and brightness of that particular spot on the image.

Advantages of Raster Animation

Raster animation provides the animator control over the appearance of an image. Raster animation also uses less system memory. Following are considered as advantages of raster animation:

- **Compression:** Adobe or Macromedia Flash provides an easy way to change the file's compression by right clicking the 'raster file' and selecting specific Properties.
- **Easier on the CPU:** Compared to vector animation, raster animation takes less Central Processing Unit or CPU time.
- **Smooth:** At the expense of CPU time, you can use 'Smoothing' operation on raster animation files that can be resized.
- **Faster Effects:** When filters are applied, a raster graphic will perform faster than a vector graphic.

Check Your Progress

1. What is sprite animation?
2. What are the two main types of computer animation?

13.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. The form of animation based on objects movement only (no change of other properties) is called sprite animation.
2. There are two main types of computer animation, vector based and raster based.

13.6 SUMMARY

- Animation makes graphics more realistic by imparting motion and dimension to an inanimate object. Intuitively though we think of animation synonymous with motion, technically speaking, it covers all changes that have a visual effect.

NOTES

- Classically, picture frames depicting animated sequence were drawn manually. For this the onionskin technique which is popularly known as cel animation technique is mostly adopted.
- The form of animation based on objects movement only (no change of other properties) is called sprite animation.
- Animation packages provide special functions for designing the animation and processing the individual objects. Several well-suited steps are defined in the animation packages which can be used in the development of animation sequences in a computer system.
- Vector animation is based on simple geometric shapes, while raster animation is created using more detailed images similar to photos or paintings.
- Raster based animation frames (and all the related raster images) are made up of individual pixels. Each of these pixels contain information about the color and brightness of that particular spot on the image.
- Raster animation provides the animator control over the appearance of an image. Raster animation also uses less system memory.

13.7 KEY WORDS

- **Storyboard:** It is an outline of action and consists of rough sketches of motion sequence or it could be a list of basic events that are to take place
- **Raster Animation:** It is the most common animation technique in which the frames are copied very fast from off-screen memory to the frame buffer.

13.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What is animation?
2. Define the significance of cel in traditional animation.
3. What are general computer animation functions?
4. What is the significance of raster animation?

Long Answer Questions

1. Explain the steps in the designing of an animation sequence?
2. Explain general computer animation functions with reference to computer graphics.
3. Explain raster animation techniques with reference to computer graphics.

13.9 FURTHER READINGS

NOTES

Hearn, Donal and M. Pauline Baker. 1990. *Computer Graphics*. New Delhi: Prentice-Hall of India.

Rogers, David F. 1985. *Procedural elements for Computer Graphics*. New York: McGraw-Hill.

Foley, D. James, Andries Van Dam, Steven K. Feiner and John F. Hughes. 1997. *Computer Graphics Principles and Practice*. Boston: Addison Wesley.

Mukhopadhyay, A. and A. Chattopadhyay. 2007. *Introduction to Computer Graphics and Multimedia*. New Delhi: Vikas Publishing House.

UNIT 14 OTHER ANIMATION TECHNIQUES

NOTES

Structure

14.0 INTRODUCTION

In this unit, you will learn about the animation languages, key frame systems and motion specifications. Computer animation languages are used to program animation functions which comprises of a graphics editor, key frame generator, in between generator and standard graphics routines. Keyframes are image frames which depict the key positions of the objects being animated.

14.1 OBJECTIVES

After going through this unit, you will be able to:

- Discuss the various types of computer animation languages
- Understand the importance of keyframe systems
- Understand motion specifications

14.2 COMPUTER ANIMATION LANGUAGES

For an animation sequence, the design and control are carried out through a set of animation outlines. General purpose languages like FORTRAN, Pascal, LISP and C, are generally employed for programming the animation functions, but there are various specialized animation languages that have been created. Animation functions comprise a graphics editor, a key frame generator, an in-between generator as well as standard graphics routines. With a graphics editor it becomes possible to design and change object shapes, using spline surface, constructive solid geometry methods, or other representation schemes.

NOTES

Scene description is one of the tasks associated with animation specification. Scene description includes the positioning of objects and light sources, defining the photometric parameters, and setting the camera parameters (position, orientation, and less characteristics). Action specification is also a standard function associated with animation specification. It involves the layout of motion paths for the camera and the object. The usual graphics routines: viewing and perspective transformations, geometric transformations to generate object movements as a function of accelerations or kinematic path specification, visible-surface identification, and the surface rendering operations are associated with animation specification.

There are key-frame systems which are specialized animation languages specifically created for generating in-betweens from the key frames specified by the user. Typically, a scene's every associated object is defined as a set of rigid bodies connected at the joints and with a limited number of degrees of freedom. Use is made of a parameterized system for specifying characteristics of an object motion as part of the object definition. The adjustable parameters control such object characteristics as degree of freedom, motion limitations, and allowable shape changes. With scripting systems one can define object specifications and animation sequences with the accepted user-input script. The script can be used to develop a library of various objects and motions.

The Computer-Animation Process

Half of the process of creating a computer-animated features films has nothing to do with computers. First, the filmmakers write a treatment, which is a roughly sketch of the stories. When they have settled on all of the story beats — major plot points — they are ready to create a storyboard of the films. The storyboard is a 2-D, comics-books-style rendering of each scene in the movie along with some jokes and snippets of important dialogues. During the storyboarding processes, the script is polished and the film makers can start to look how the scenes will work visually. The next steps is to have the voice actors come in and records all of their lines. Using the actors' record dialogue, the filmmaker assembles a video animated only with the storyboard drawing. After further re-writing, editing, and re-recording of dialogues, the real animations are ready to begin. The art departments now design all the characteristics, major set locations, colour and propspalettes for the film. The characteristics and props are model in 3-D or scanned into the computer from clay models. Each character is equipped with hundreds of avars, little hinges that allow the animators to move specific parts of the characters body. Woody from *Toy Story* for example, had over 100 avars on his face alone.

The next steps is to create all of the 3-D set, painstakingly dressed with all of the detail that brings the virtual world to life. Then the characteristics are placed on the sets in a process called blocking. The director and lead animators block the key character position and camera angle for each and every shot of the movies.

Now team of animator is each assigned short snippet of scenes. They take the blocking instruction and create their own more detail key frame. They begin the tweening process. The computer handle a lot of the interpolation — calculating the best way to tween two key frame — but the artist often has to tweak the result so they look even more lifelike.

Now the characteristics and props are given surface texture and colours. They are dressed with clothing that wrinkles and flow naturally with body movement, hair and fur that waves in the virtual breeze, and skin that look real enough to touch. The final steps of the process are called rendering. Using powerful computer, all of the digital informations that the animators has created — character model, key frames, textures, tweens, colours, sets, props, lighting, digitals paintings, is assembled into a single frame of films. Even with the incredible computing power, it take an average of six hours to render one frames of an animated film. That takes over 88 hours of rendering for a 90-minutes film

14.2.1 Key-Frame Systems

Keyframes are image frames that depict the key positions of the objects being animated and marks significant changes in the animation sequence. Usually the extremes of an action or sequence like start, stop and changes of movement direction occur at keyframes. The more intricate and rapidly varying the motions are, the more number of keyframes are required. *In-betweens* are the intermediate frames drawn between the keyframes and are used to smooth the transition from one keyframe to the next.

We are familiar with traditional cartoon animation used in the movies or televisions where artists meticulously draw each frame of a scene and then capture them frame by frame with a movie camera. While the expert artist draws the keyframes, the assistants create the filler in-betweens most mechanically.

In a computer based animation, standard software tools are available to design the keyframes. The software then figures out the in-betweens by applying *interpolation* algorithms. One does not have to design every intermediate frame individually. The process of generating the in-betweens is commonly known as *tweening*. The most basic interpolation technique is *linear interpolation* or *lerp*. Given the values p_1 and p_2 of some attribute (position, color, size) in two keyframes corresponding to times t_1 and t_2 respectively, the value p_t at any intermediate frame corresponding to time t is given by,

$$p_t = \frac{t_2 - 1}{t_2 - t_1} p_1 + \frac{1 - t_1}{t_2 - t_1} p_2 .$$

This implies that $p_t = (1 - n)\, p_1 + n\, p_2$; where $n = \frac{1 - t_1}{t_2 - t_1} \leq 1$ and ≥ 0.

Lerping generates motion that starts and stops instantaneously, with objects attaining their full velocity as soon as they start to move and maintain the movement

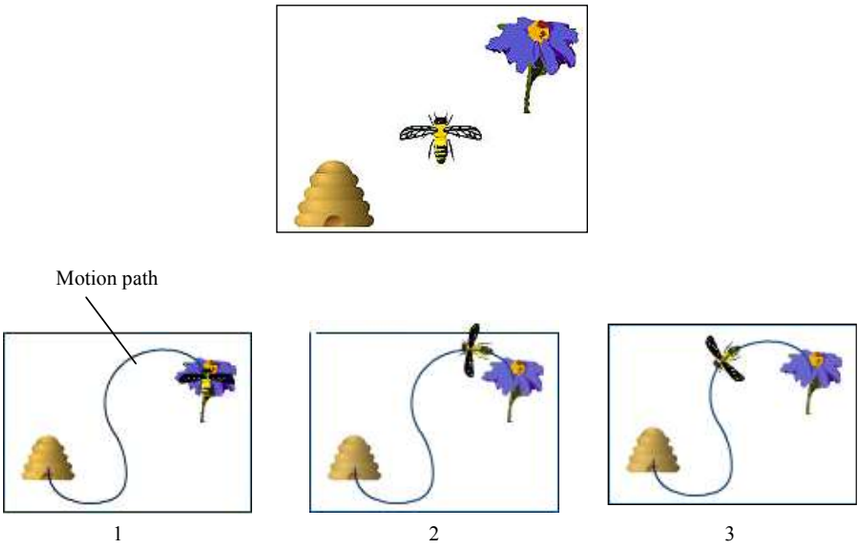
NOTES

speed until they stop. For such constant velocity animation equal interval time spacing is used. Moreover the motion path is linear which cannot simulate the realistic curvilinear trajectory (e.g., projectile path) of moving objects.

To make different parameters vary realistically with time, *spline interpolation* technique is often used. By using Bezier functions instead of linear functions to interpolate between keyframes, smooth motion can be achieved simulating the gradual increase of velocity at the start and conversely gradual decrease of velocity at the end. When a motion begins, the amount of change from one drawing to the next is kept small, but gradually increased. This is called *easing in*. The time spacing between frames is increased so that greater changes in position occur as the object moves faster. When the motion is underway, the changes from frame to frame are held constant. When the motion ends, it is often stopped gradually, by reducing the amount of change from frame to frame of the moving object. This is called *easing out*. The speed control facility of such non-linear animation sequences can be effectively used while it is required to synchronize animation with audio playback.

Note that spline interpolation doesnot mean that objects should follow Bezier shaped paths, but the rate at which their properties change should be interpolated using a Bezier function say $f(t)$. Intermediate parameter p_t can be calculated as $p_t = (1 - f(t))\, p_1 + f(t)\, p_2$.

Thus *spatial interpolation* defines the *motion paths* or change of object position in space. *Temporal interpolation*, on the other hand, affects the rate of change of objects position with time. Given the vertex positions at the keyframes, we can fit the positions with linear or non-linear motion paths. Interpolation can be applied to other properties of a layer. Its angle can be varied, so that it appears to rotate. Zoom in and zoom out effect, i.e., the impression of approaching or receding movement can be introduced by scaling or interpolating size of the object.



NOTES

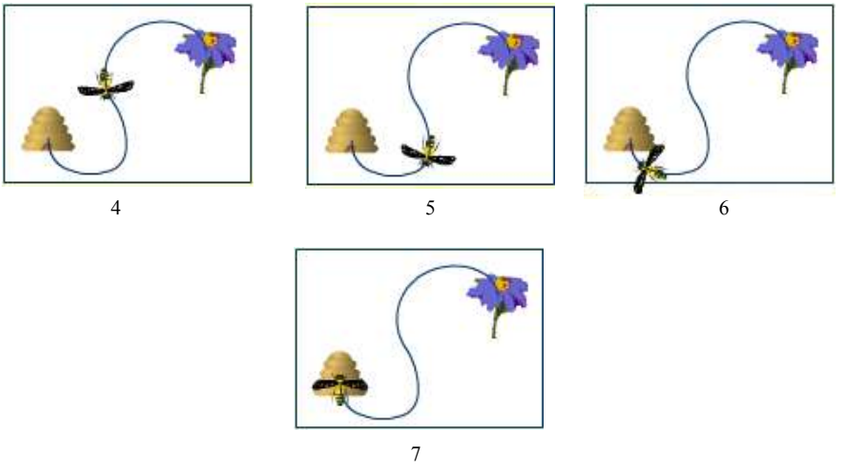


Fig. 14.1 *Keyframe Systems*

In Figure 14.2 the flower, the bee and the honeycomb – three objects are used for animation. The animation can be created which will make the bee to fly from the flower to the honeycomb following a curved path. The curved path is then defined as motion path with the initial and final position of the bee as shown in keyframes 1 and 7. The in-betweens 2-6 showing intermediate position of the bee along the motion path are generated by the software (Macromedia Flash) itself.

Further to geometrical transformations, parameters for different effects (e.g., brightness of *glowing edges*) and filters (e.g., radius of *Gausian blur*) of bitmapped images can be made to vary over time using the standard methods of interpolation between keyframes. Other elements of animation which can be employed for bitmapped images include extensions, tilting, bending, lofting, rendering, fading (in or out) and exploding. These elements change in successive frames as time progesses creating a flowing series of changing imagery called as *motion graphics*. Changes can occur independent or in agreement with other changes. For example, you can make an object rotate and fade while it is moving.

Morphing

Morphing is a special affect which is specifically created in motion pictures and animations to change or morph one image into another through a perfect transition. Most often it is used to depict one person turning into another through technological means or as part of a fantasy or incredible sequence. Traditionally, such a depiction can be attained through cross fading techniques on a film strip. Nowadays, this has been replaced by computer software for creating more realistic transitions. Computer techniques involved in morphing basically distort one image to fade into another through marking corresponding points and vectors on the 'before' and 'after' images used in the morph simultaneously. For example, one would morph one face into another by marking key points on the first face, such as the contour of the nose or location of an eye and then marking on the second face where these

NOTES

same points existed. The computer can then distort the first face to have the shape of the second face at the same time that it faded the two faces.

The contour and color of the image of a cat can be morphed into the image of a tiger through simple shape and color interpolation. Morphing effect can not only be applied in transition between still images but also between moving images. It can produce transformations from one object to a completely different object, like the face of a girl transforming into the face of a Cheetah (refer Figure 14.2). Or it can produce smooth transition from the original object to its distorted version – a smile created in a face or a structure buckled under load or something like that where the basic object remains same, and only the character changes.

Fig. 14.2 *Morphing Application: Face of the Girl Sequentially Transforming into the Face of a Lion*

There are two parts in the morphing algorithm–the warp and the dissolve. Given a source image and a target image, *warping* is the process of distorting the source image so that it matches the target image. Actually enough number of *feature points* or *morph points* are specified on both the source and target image that define the respective profile features and divide the images in non-overlapping triangular and quadrangular meshes. Warping forces morph points of the two different images match up at the end. This in turn produces one-to-one transformation of mesh triangles and quadrangles of the source and target images.

For a morph point A in the source image and a corresponding morph point B in the target image, linear interpolation is used to generate a new morph point C in an intermediate image. It is given as $C = wA + (1 - w)B$, w being a weight applied based on the position of the intermediate image in the timeline. The linear transformation between two triangles, say, ABC and XYZ in the source and target images, respectively are given by,

$$\mathbf{p} = w_1\mathbf{A} + w_2\mathbf{B} + w_3\mathbf{C}$$ [**p** is the position vector of any pixel within ΔABC]

$$\mathbf{p}' = w_1\mathbf{X} + w_2\mathbf{Y} + w_3\mathbf{Z}$$ [**p′** within ΔXYZ is the transformed counterpart of **p**]

$$= w_1 + w_2 + w_3 = 1$$

Thus morphing is a kind of shape tweened animation. There are dedicated morphing software available in the market where the basic keyframing technique applied is same as that adopted in animation *motion tweening* software.

NOTES

At first you should have the image of the original object as the first keyframe (source image) of the morphing sequence. Then you create a second keyframe—the desired number of frames after the first frame. Here you copy the original object or put a completely different object. This would be the last frame (target image) in the sequence depicting the ultimate transformation stage. Set equal number of morph points (keypoints) in the first and last keyframes to specify the changes of different portions of the image separately. As morph points have a one-to-one correspondence in starting and ending shapes, the point you set in the start image will move/change to the corresponding point in the final image.

Thus you can position morph points along the contour of lips, both in the image of a face (in the start keyframe) and in its copy (in the end keyframe). If you want to make the face smiling you have to open up the lips. So reposition the morph points along an imaginary open lip contour in the end frame assuming how it should appear after the original face gets changed to a smiling face. The software will now interpolate the shapes for the frames in between, thereby creating the desired morphing. The more the number of morph points and intermediate frames chosen (with a given frame rate), the smoother the morph will be. Similarly you can morph the image of a girl into the image of a boy. The effect will be as if the girl is gradually transforming to a boy miraculously by some magic touch.

Actually when the morphing routine is run, the image of the start frame is warped into the shape of the end frame image as it gradually fades a the specified image undergoes the fading and warping effect and the target image undergoes the reverse warping and fading. That is, the target image gradually fades in and reveals itself most prominently in the end frame. Hight quality morphing software provides smooth blending of two images so that the source image seems to melt or dissolve into the subsequent image frames and finally to the target image. This part of morphing process is known as *dissolve*. While warping creates transformation between point coordinates, dissolve causes transformation of color vectors (R, G, B) pixel by pixel.

To control the software generated morphing sequence, you can choose any one option for Blend type from the following:

Distributive type creates a sequence in which the intermediate shapes are smoother and more irregular.

Angular type creates a sequence that preserves apparent corners and straight lines in the intermediate shapes. Angular is appropriate only for blending shapes with sharp corners and straight lines.

As in motion tweening, the rate of shape transition can be controlled in a morphing sequence. By default, the rate of shape change through tweened frames is constant. Easing (In and Out) creates a more natural effect of morphing by increasing or decreasing the rate of change toward the beginning or ending of a transition.

NOTES

Figure 14.3 shows how shape change occurs in morphing.

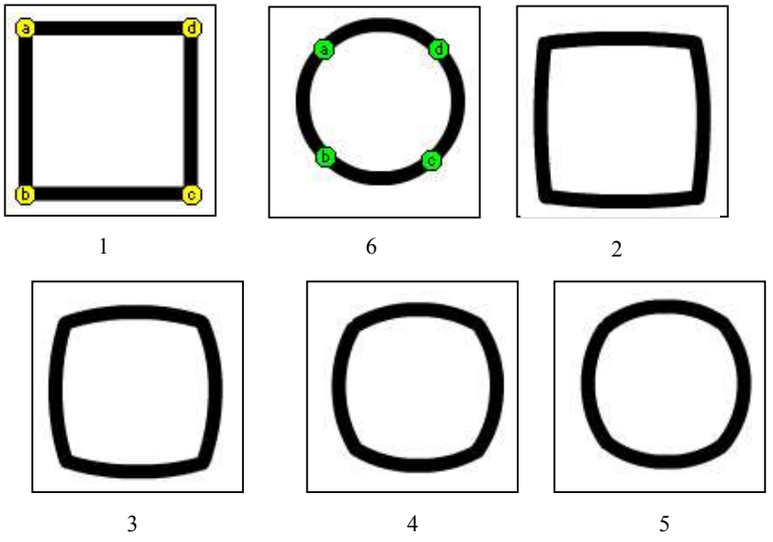


Fig. 14.3 *Morphing*

Frame 1 is the start frame showing morph points a, b, c, d on the square. Frame 6 is the end frame showing the changed position of the same set of morph points on the circle. We want the square to transform to the circle. Frames 2-5 are the intermediate frames generated by the software as the square gradually fades out and the circle takes its shape.

Today some of the most widely used tools for creating multimedia animations are *Macromedia's Director* (dir) and *Flash* (), *AnimatorPro* (fli and flc), 3-D *Studio Max* (max), *Shockwave* (dcr), etc. Within brackets are specified the animation file formats generated by the corresponding software. Besides this, Windows Audio Video Interleaved *format* (avi), mpg and mov formats can also support animations.

Some products that offer morphing features are *Gryphon Softwares Morph*, *Ulead's MorphStudio, ImageWare's MorphWizard, MetaTool's Digital Morph,* etc.

14.2.2 Motion Specifications

There are numerous methods to specify the motions of objects in an animation system. The general methods to describe an animation sequence using explicit specification of paths of motion include the description of the interactions that are produced due to motion. The motion sequence can be specified using motion parameters, such as the rotation angles and translation vectors. The animator can use a technique to specify each and every pixel value at each and every frame. The feature that is appropriate for one kind of animation may not be the appropriate feature for another kind of animation. Hence, the animator can specify the various levels of abstractions.

Besides other special effects, the camera panning effect can also be simulated in animation like in the movies. Typically, *panning* refers to the change of the background across the field of view. A background drawing larger than the object is moved step by step across the animation sequence as the camera exposes frame after frame of film. Objects in the foreground appear to be moving along relative to the scenery behind them. Thus we can make a bird fly while the background (sky) rolls in opposite direction giving the illusion that the bird is covering distance.

Fig. 14.4 *The Baby Crawls from Left to Right and then Holds the Ball with his Hands. The Ball Spins and Rolls from Right to Middle of the Scene*

Fig. 14.5 *The Spotlight Passes Over the Text from Left to Right – such Animation effect is often used in Title Casting of Motion Pictures*

14.2.3 Kinematics and Dynamics

A wide spread method for animating the articulated images is termed as kinematics. It is based on the properties of motion, such as time, position and velocity. The term forward kinematics is used when joint rotations are specified in function of time. The term inverse kinematics refers to the problem of specifying the forward kinematics values when the end-point of the character articulation is known. The most common examples of forward and inverse kinematics are the keyframe editors

NOTES

which interpolate forward kinematics values from a few intermediate values and the possibilities of fixing the extremities of articulated objects while moving the other components or dragging the extremities themselves present in many modelling and animation packages.

3-D Animation

A distinct alternative to the 2-D animations techniques is 3-D animation or stop motion animation.

The classical technique to create 3-D object models out of malleable modelling material are to plasticine and manipulate those objects in 3-D miniature sets between shots to produce natural movement, gesture and otherwise impossible changes. This form of animation is often called *clay animation*.

In the digital realm, 3-D wireframe models are created first and then surface and material properties are assigned using photo realistic rendereing. There are distinct numerical parameters that control object's position (movement) and orientation (rotation) in space, its surface characteristics, its shape, intensity and direction of light sources, camera position and angle. A 3-D animation is achieved by rendering a scene as the first frame, making some changes to the parameters, rendering the next frame, and so on. Motion paths in 3-D, often 3-D Bezier splines, can be used to describe movement.

Realistic shading and rendering based on advanced ray tracing algorithm consumes considerable time to generate a scene. Therefore high processing power and memory is required to cope up with the required frame rate for smooth animation.

At the very highest level of 3-D computer generated animation software, interfaces allow the animator to control different movement parameters to produce smooth movement across the frames. Described below briefly are the different methods of controlling animation.

1. **Full Explicit Control:** It is the simplest type of control where the animator either specifies simple changes like scaling, translation, rotation or provides keyframe information and interpolation methods interactively.
2. **Procedural Control:** It is based on certain kinds of behavior that can be applied to objects and the way they interact. In a *physically based system* the position of one object may influence the motion of another object, for example, spotlight follows a dancer, a sunflower follows the sun, etc. In such systems, objects are modeled with physical attributes, such as mass, moment of inertia, elasticity, velocity, etc., and object behaviour as emulated in animation are based on laws of Newtonian physics against applied external force. Thus moving objects can be made to collide realistically or to bounce of solid surface.

3. **Kinematics:** It is the study of motion of bodies without reference to mass or force. That is, it is only concerned with how things can move, rather than what makes them do so. Animations of linked objects or jointed structures, for example limbs of human or animal figures are controlled by imposing *kinematic constraints* obeyed by real objects or structures. For example, a 3-D model of a door must have the same *degree of freedom* to move/ rotate as a real door has with the movement constraints produced by the hinges.

   Kinematics being a general term, *forward kinematics* and *inverse kinematics* both are used in controlling animation. While the former deals with linked motions from cause to the effect, the inverse kinematics works backward from effect to cause. For example, it is the motion of the upper arm that propels the rest of the arm and hand. Modelling the hand's position from movement and position of the upper arm requires forward kinematics. Whereas, first fixing the position of the hand and then backtracking to find the relevant motion of upper arm is what inverse kinematics is and sometimes it is more useful to the animator.

4. **Tracking Live Action:** This technique produces exceptionally realistic motion. Trajectories of objects to be animated can be generated by tracking of live action. One such method is *rotoscoping*. A film is made in which people or animals act out the parts of the characters in animation. Then the animator draws over the film, changing the background and replacing the human or animal actors with their animation equivalents. In an alternative method some sort of indicators or motion sensors are attached to key points on an actor's body or body suit. By tracking the position of the indicators or sensors, the animator can get locations for corresponding key points in an animated model.

Check Your Progress

1. What are the general purpose languages used for programming the computer animation functions?
2. Define morphing.

14.3 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. General purpose languages like FORTRAN, Pascal, LISP and C, are generally employed for programming the animation functions.
2. Morphing is a special affect which is specifically created in motion pictures and animations to change or morph one image into another through a perfect transition.

NOTES

14.4 SUMMARY

- Animation functions comprise a graphics editor, a key frame generator, an in-between generator as well as standard graphics routines.
- Half of the process of creating a computer-animated features films has nothing to do with computers. First, the filmmakers write a treatment, which is a roughly sketch of the stories.
- Keyframes are image frames that depict the key positions of the objects being animated and marks significant changes in the animation sequence.
- Morphing is a special affect which is specifically created in motion pictures and animations to change or morph one image into another through a perfect transition.
- The motion sequence can be specified using motion parameters, such as the rotation angles and translation vectors. The animator can use a technique to specify each and every pixel value at each and every frame.
- A wide spread method for animating the articulated images is termed as kinematics. It is based on the properties of motion, such as time, position and velocity.
- The classical technique to create 3-D object models out of malleable modelling material are to plasticine and manipulate those objects in 3-D miniature sets between shots to produce natural movement, gesture and otherwise impossible changes. This form of animation is often called clay animation.

14.5 KEY WORDS

- **Animation:** It refers to the technique of photographing successive drawings orpositions of puppets or models to create an illusion of movement when the film isshown as a sequence.
- **Full Explicit Control:** It is the simplest type of control where the animator either specifies simple changes like scaling, translation, rotation or provides key frame information and interpolation methods interactively.
- **Tracking Live Action**: This technique produces exceptionally realistic motion. Trajectories of objects to be animated can be generated by tracking of live action. One such method is rotoscoping.

NOTES

14.6 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What is a key frame?
2. What does motion specifications refer?

Long Answer Questions

1. What are the different types of computer animation languages? Explain.
2. Explain the significance of motion specifications in computer graphics.

14.7 FURTHER READINGS

Hearn, Donal and M. Pauline Baker. 1990. *Computer Graphics*. New Delhi: Prentice-Hall of India.

Rogers, David F. 1985. *Procedural elements for Computer Graphics*. New York: McGraw-Hill.

Foley, D. James, Andries Van Dam, Steven K. Feiner and John F. Hughes. 1997. *Computer Graphics Principles and Practice*. Boston: Addison Wesley.

Mukhopadhyay, A. and A. Chattopadhyay. 2007. *Introduction to Computer Graphics and Multimedia*. New Delhi: Vikas Publishing House.