New Edition

COMPUTER GRAPHICS

BSc. CSIT / BIM

Arjun Singh Saud
Bhupendra Singh Saud
Indra Chaudhary

Syllabus (BIM)

Course Objective: This module aims to develop students' skill in computer graphics. This module should be supported by laboratory experiments to augment the concepts taught in the class.

Course Description: Introduction of Computer Graphics, Hardware and Software Concept, Two Dimensional Algorithm, Three Dimensional Graph, Visible surface detection method, Illumination models and surface rendering methods, and Trends in Computer Graphics, Areas, Text and Colors.

Course Details

Unit 1: Introduction of Computer Graphics ---------- LH 2

1.1 Early History

1.2 Application of Computer Graphics

Unit 2: Hardware and Software Concept ---------- LH 9

1.3 Input Devices Mouse, Touch Screen, Light Pen, Data Glove, Tablet (Digitizer), Bar Code Reader

1.4 Output Devices

1.4.1 Monochromatic & Color CRT

1.4.2 Raster and Random Scan Display

1.4.3 Flat Panel Display

1.4.3.1 LED

1.4.3.2 LCD

1.4.4 Simple Architecture on Raster, Random Scan System

1.4.5 Concept of Three Dimension viewing devices

1.5 Graphics Software

1.5.1 Software standards

1.5.2 Need of machine independent graphics language

1.6 Color Models

1.6.1 RGB

1.6.2 HSV

1.6.3 Conversion between HSV and RGB color models

Unit 3: Two Dimensional Algorithm ---------- LH 16

1.7 Line Drawing Algorithm
- 1.7.1 DDA (With positive and negative slope)
- 1.7.2 Bresenham's Line Algorithm
- 1.7.2.1 for positive slope (m<=1, m>1)
- 1.7.2.2 for negative slope (m<=1, m>1)

1.8 Different Line Styles with Java 2D

1.9 Drawing Thick Lines with Java 2D

1.10 Circle Algorithm
- 1.10.1 General circle equation
- 1.10.2 Mid-Point circle equation

1.11 2D Geometric Transformation
- 1.11.1 Basic Transformation
- 1.11.1.1Translation
- 1.11.1.2Rotation
- 1.11.1.3Scaling
- 1.11.2 Homogenous Coordinate
- 1.11.3 Composite Transformation
- 1.11.3.1Successive Translation
- 1.11.3.2Successive Rotation
- 1.11.3.3Successive Scaling
- 1.11.3.4Pivot point rotation
- 1.11.3.5Fixed point scaling
- 1.11.4 Other Transformation
- 1.11.4.1Reflection
- 1.11.4.2Shear

1.12 Two Dimensional Viewing
- 1.12.1 Viewing pipeline
- 1.12.2 Window to viewport coordinate transformation
- 1.12.3 Clipping
- 1.12.3.1Introduction and application of clipping
- 1.12.3.2Line Clipping
- 1.12.3.2.1 Cohen Sutherland line clipping
- 1.12.3.3Polygon clipping
- 1.12.3.3.1 Sutherland Hodgeman polygon clipping

Unit 4: Three Dimensional Graph ---------- LH 8

1.13 3D object representation

1.13.1 Polygon surface

1.13.2 Polygon tables

1.13.3 Plane equations

1.13.4 Polygon meshes

1.14 Projections

1.14.1 Parallel Projections

1.14.1.1Isometric

1.14.1.2Oblique

1.14.2 Perspectives Projections

1.14.3 Derivation of projections

1.15 3D Transformations

1.15.1 Translation

1.15.2 Rotation

1.15.2.1General 3D rotation

1.15.2.2Geometric transformations in Java 3D

1.15.3 Scaling

1.15.3.1Fixed point scaling

1.15.4 Reflection

1.15.5 Shear

Unit 5: Visible surface detection method ---------- LH 5

1.16 classification of algorithm

1.17 different types of algorithm

1.17.1 depth buffer (z-buffer) method

1.17.2 A-Buffer method

1.17.3 Scan line method

1.17.4 Depth sorting method (Painter's Algorithm)

1.17.5 Clipping in Java 3D

Unit 6: Illumination models and surface rendering methods ---------- LH 6

1.18 light source

1.19 basic illumination models

1.19.1 Ambient light
1.19.2 Diffuse reflection
1.19.3 Specular reflection and Phong model
1.19.4 Intensity attenuation
1.19.5 Color consideration
1.19.6 Transparency
1.19.7 Transparency in Java 3D
1.19.8 Shadows
1.20 Polygon rendering methods
1.20.1 Constant intensity shading
1.20.2 Light sources in java 3D
1.20.3 Gouraud shading
1.20.4 Phong Shading
1.20.5 Shading in Java 3D
1.20.6 Constant and Gouraud Shading in Java 3D

Unit 7: Trends in Computer Graphics ---------- LH 2

1.21 Concept of Virtual reality & simulation
1.22 Computer animation
1.22.1 Design of animation sequences
1.22.2 Computer animation languages
1.22.3 Morphing and simulating accelerations.
1.22.4 Animation in Java 3D
Projections in Java 3D

Unit 8: Areas, Text and Colors

1.23 Filling Areas
1.24 Buffered Images in Java 2D
1.24.1 Double Buffering in Java 2d
1.24.2 Loading and Saving of Images with Java 2D
1.24.3 Textures in Java 2D
1.25 Displaying Text
1.26 Text in Java 2D
1.27 Grey Images and Intensities
1.28 Color models

1.28.1 Colors in Java 2D
1.28.2 Color interpolation
1.28.3 Color interpolation with Java 2D

Laboratory Work

Lab1: DDA
Lab2: Bresenhams line drawing algorithm
Lab3: Circle
Lab4: Basic transformation on 2D
- Translation
- Rotation
  - About origin
    - About pivot point
  - Scaling
  - About origin
  - About fixed point

Lab5: Simple 3D Object
Lab6: Basic Transformation on 3D object
- Translation
- Rotation
- Scaling

Laboratory work will be carried out using visual or non visual high level languages.

References

- Computer Graphics, C Versions (Prentice Hall) : Hearne and Baker
- Computer Graphics – Principles and Practices: J.D. Foley, S.K. Feiner and J.F. Hughes
- Computer Graphics: Principles and Practice, 2nd Edition (3rd would be released around mid 2013) - the book you've mentioned is also called The Bible of CG
- Fundamentals of Computer Graphics, 3rd Edition
- Computer Graphics using OpenGL, 2nd or 3rd Edition*
- Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL, 6th Edition*
- 3D Computer Graphics: A Mathematical Introduction with OpenGL*
- Introduction of Computer Graphics: Using Java 2D and 3D, Frank Klawonn, Second Edition, Springer
- Digital Image Processing: An algorithmic Introduction using Java, Wilhelm Burger, Mark, J. Burge, First edition, Springer
- Fundamentals of Computer Graphics, Peter Shirley, Michael Ashikhmin, Steve Marschner, Third edition, A K Peters/CRC Press

Table of Contents

Chapter 1 INTRODUCTION OF COMPUTER GRAPHICS

Introduction ---------- 2
Types of computer Graphics ---------- 2
Early history of computer graphics ---------- 3
Uses of Computer Graphics ---------- 5
Application of Computer Graphics ---------- 6
Introduction to image processing ---------- 9
Differences between computer graphics and Image processing ---------- 10
Exercise ---------- 10

Chapter 2 HARDWARE AND SOFTWARE CONCEPT

Introduction ---------- 12
Input Devices ---------- 12
Output Devices ---------- 18
Types of Refresh cathode Ray tubes (CRT's) ---------- 24
Graphics Software ---------- 33
Software Standards ---------- 33
Exercise ---------- 34

Chapter 3 SCAN CONVERSION ALGORITHM

Introduction ---------- 38
Line Drawing Algorithm ---------- 38
Circle Algorithm ---------- 52
Methods to draw circle in computer graphics ---------- 53
Ellipse Algorithm ---------- 61
Area filling ---------- 70

Types of Polygons ---------- 70
Filled Area primitives ---------- 71
Exercises ---------- 89

Chapter 4 TWO-DIMENSIONAL GEOMETRIC TRANSFORMATIONS

Introduction ---------- 92
Two-Dimensional translation, Rotation, Scaling, Reflection and Shearing ---------- 94
Two-Dimensional Viewing ---------- 113
Clipping ---------- 118
Exercise ---------- 130

Chapter 5 THREE-DIMENSIONAL GEOMETRIC TRANSFORMATION

Introduction ---------- 134
Three-Dimensional translation, rotation, scaling, reflection and shearing ---------- 135
Three-Dimensional Viewing ---------- 145
Exercise ---------- 154

Chapter 6 3D OBJECTS REPRESENTATION

Introduction ---------- 158
Representing Surface ---------- 158
Surface normal and spatial orientation of surfaces ---------- 160
Plane Equations ---------- 161
Polygon Meshes ---------- 162
Wire frame Representation ---------- 163
Representing Curves ---------- 165
Spline Representation ---------- 166
Quadric Surface ---------- 178
Exercise ---------- 179

Chapter 7 SOLID MODELING

Introduction ---------- 182
Sweep, Boundary and Spatial-Partitioning Representation ---------- 186
Boundary Representation (B – rep) ---------- 187
Binary Space Partition Trees (BSP) ---------- 188
Octree Representation ---------- 189
Exercise ---------- 190

Chapter 8 VISIBLE SURFACE DETECTION

Introduction ---------- 192
Back face Detection (Plane Equation method) ---------- 194
Depth Sorting Method ---------- 200
Binary Space Partitioning (BPS) Tree Method ---------- 201
Ray Tracing method ---------- 204
Exercise ---------- 205

Chapter 9 ILLUMINATION MODELS AND SURFACE RENDERING TECHNIQUES

Introduction ---------- 208
Basic Illumination Models ---------- 212
Intensity attenuation and Color consideration ---------- 214
Transparency ---------- 215
Shadows ---------- 215
Polygon (surface) Rendering Method ---------- 216
Exercise ---------- 222

Chapter 10 INTRODUCTION TO VIRTUAL REALITY AND ANIMATION

Introduction ---------- 224
Application area of virtual reality ---------- 225
Key Elements of a Virtual Reality Experience ---------- 225

Types of Virtual Reality ---------- 226
Key Components in a Virtual Reality System ---------- 227
Key Components Inside of a Virtual Reality Headset ---------- 228
Other Concepts to Understand How Virtual Reality Headsets Work ---------- 229
Design of animation sequences ---------- 230
Computer Animation Languages ---------- 231
Morphing and Simulating Accelerations ---------- 231
Simulating Acceleration ---------- 232
Exercise ---------- 237

Chapter 11 INTRODUCTION TO OPENGL

Introduction ---------- 240
Drawing Primitives ---------- 247
Polygon ---------- 249
Color commands ---------- 252
Basic Lighting ---------- 254
Exercise ---------- 256

Chapter 12 AREAS, TEXT AND COLORS

Introduction ---------- 258
Color Model ---------- 258
RGB Color model ---------- 261
Conversion between HSV and RGB Color Models ---------- 267
Filling Areas ---------- 268
Displaying Text ---------- 270
Exercise ---------- 272

Lab Assignment in Java ---------- 273
Bibliography ---------- 294
Solution ---------- 295

Chapter

1

Introduction of Computer Graphics

STRUCTURE

1.0 Introduction

1.1 Types of Computer Graphics

1.2 Early history of computer graphics

1.3 Uses of Computer Graphics

1.4 Application of Computer Graphics

1.5 Introduction to image processing

1.6 Differences between computer graphics and Image processing

Introduction

Computer graphics is an art of drawing pictures on computer screens with the help of programming. It involves computations, creation, and manipulation of data. In other words, we can say that computer graphics is a rendering tool for the generation and manipulation of images. Computer Graphics is a field related to the generation of graphics using computers. It includes the creation, storage, and manipulation of images of objects. These objects come from diverse fields such as physical, mathematical, engineering, architectural, abstract structures and natural phenomenon. Computer graphics today is largely interactive, i.e. the user controls the contents, structure, and appearance of images of the objects by using input devices, such as keyboard, mouse, or touch-sensitive panel on the screen. In short, Computer graphics refer different things in different contexts:

- **Pictures**, scenes that are generated by a computer.
- **Tools** used to make such pictures, software and hardware, input/output devices.
- The **whole field of study** that involves these tools and the pictures they produce.

Types of computer Graphics

Basically there are two types of computer graphics namely.

- Interactive Computer Graphics and
- Non-interactive Computer Graphics

Interactive Computer Graphics

Interactive Computer Graphics involves a two way communication between computer and user. Here the observer is given some control over the image by providing him with an input device for example the video game controller of the ping pong game. This helps him to signal his request to the computer. The computer on receiving signals from the input device can modify the displayed picture appropriately. To the user it appears that the picture is changing instantaneously in response to his commands. He can give a series of commands, each one generating a graphical response from the computer. In this way he maintains a conversation, or dialogue, with the computer.

Interactive computer graphics affects our lives in a number of indirect ways. For example, it helps to train the pilots of our airplanes. We can create a flight simulator which may help the pilots to get trained not in a real aircraft but on the grounds at the control of the flight simulator. The flight simulator is a mock up of an aircraft flight deck, containing all the usual controls and surrounded by screens on which we have the projected computer generated views of the terrain visible on takeoff and landing. Flight simulators have many advantages over the real aircrafts for training purposes, including fuel savings, safety, and the ability to familiarize the trainee with a large number of the world's airports.

Non Interactive Computer Graphics

The non-interactive computer graphics is also known as passive computer graphics. It is the computer graphics in which user does not have any kind of control over the image. Image is merely the product of static stored program and will work according to the instructions given

in the program linearly. The image is totally under the control of program instructions not under the user. Example: screen savers.

Early history of computer graphics

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

At the same time, it was becoming clear to computer, automobile, and aerospace manufacturers that (CAD) and computer aided manufacturing (CAM) activities had enormous potential for automating drafting and other drawing intensive activities. The General Motors DAC system for automobile design and the Itek-Digitek system for lens design were pioneering efforts that showed the utility of graphical interaction in the iterative design cycles common in engineering. By the mid 60s, a number of commercial products using these systems had appeared at that time only the most technology intensive organizations could use the interactive computer graphics where as others used punch cards, a non-interactive system.

To understand the many issues in today's modern computer graphics, we need to know how developed computer graphics from its beginnings to this day. In brief the history of computer graphics can be categorized as below:

- **1950's** - the first graphic displays, military applications
- **1962** - the first graphics station (sketchpad) consisting of a monitor, light pen and software for interactive operation constructed by Ivan Sutherland
- **1964** - research team working on the algorithms in computer graphics employed at the University of Utah (including Ivan Sutherland, James Blinn, Edwin Catmull)
- **1965** - the first commercial graphics station: IBM 2250 Display Unit and the IBM PC 1130
- **1969** - beginning of a group SIGGRAPH (Special Interest Group on Graphics) in the organization of ACM (Association for Computing Machinery) gathering of IT professionals
- **1974** - creation of graphics laboratory at the New York Institute of Technology
- **1980** - Turner Whitted published article about creating realistic images, beginning of method of ray tracing
- **1982** - *TRON*, the first film that uses computer graphics. The first completely computer-generated scene in the movie Star Trek II: The Wrath of Khan
- **1983** - Development of fractal techniques and their use in computer graphics. Fractals are used for example in the movie Star Trek II: The Wrath of Khan

- **1984** - work of C. Goral, K. Torrance, D. Greenberg and B. Battaile and proposing a new approach for visualization – the method of radiosity
- **1988** – the first film sequence with morphing in Willow
- **1989** - the first character created using 3D graphics in the studio Industrial Light & Magic (ILM)
- **1993** - dinosaurs in Jurassic Park – the first complete and detailed living organisms generated digital technology
- **1995** - Toy Story implemented complete using computer graphics, the first photo-realistic hair and fur computer generated
- **1999** - The World Wide Web Consortium (W3C) begins development of SVG (Scalable Vector Graphics), a way of using text-based (XML) files to provide higher-quality images on the Web. SVG images can include elements of both conventional vector and raster graphics.
- **2007-** Apple launches its iPhone and iPod Touch products with touch screen graphical user interfaces.
- **2017-** Microsoft announces it will not kill off its basic but very popular Paint program, loved by computer artists for over 30 years.
- The beginnings of computer graphics were related to the military industry, due to the very high cost of equipment. The development of new graphics techniques has forced the film industry requires realistic special effects. Currently, computer graphics is used in many areas of life.

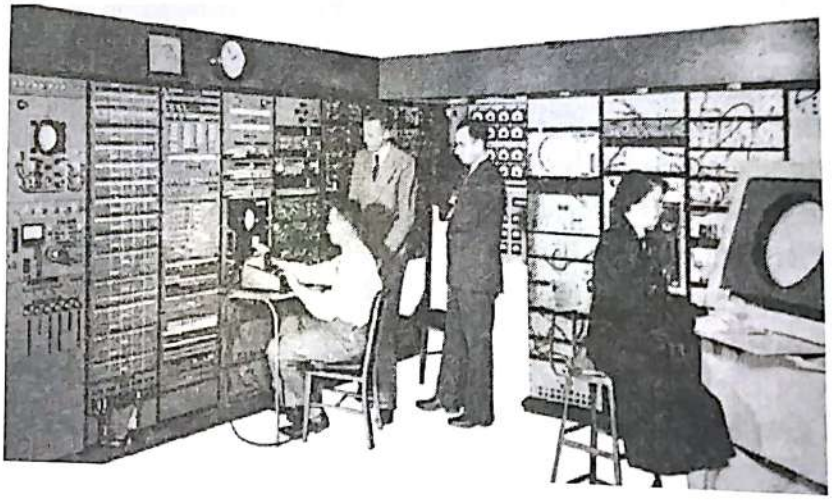


Figure 1.1: Whirlwind: early graphics using Vector Scope (1951)

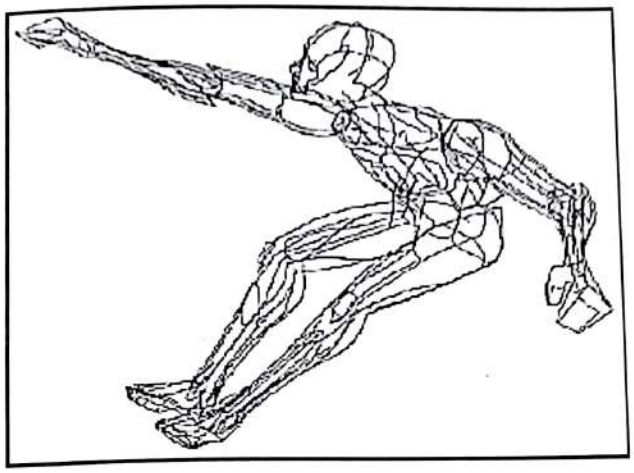


Figure 1.2: Images on a plotter exploring cockpit design using a 3D model of Human body created by William Fetter in 1960.

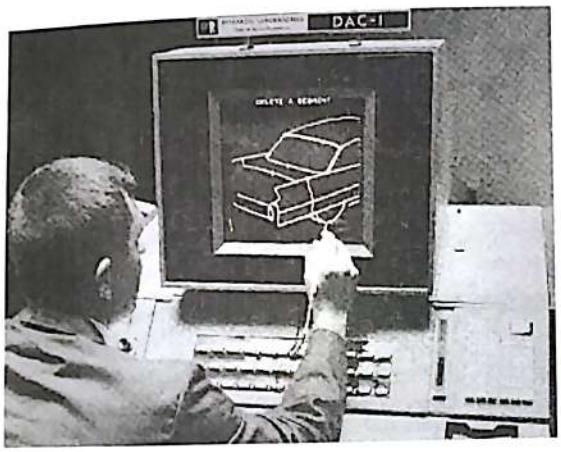


Figure 1.3: DAC-1, first commercial CAD system, developed in 1959 by IBM

Figure 1.4: Sketch Pad: First interactive graphics (1961)

Uses of Computer Graphics

There are several uses of CG which are very useful in the current scenario. Some of its uses include:

- Computer program Development
- Making movies
- Video Games (Ex: Plat formers, Role-playing games, side scrollers, first person shooters)
- Catalogs designing
- Creating Commercial Arts
- Scientific Modeling (Ex: Weather Forecasts, Meteorological data)

Application of Computer Graphics

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

1. User interfaces

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

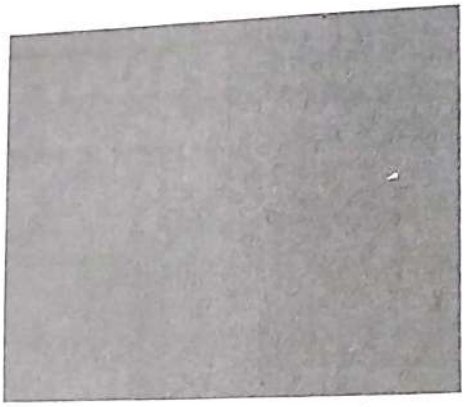


Figure 1.5: B205 Control Console (1960)

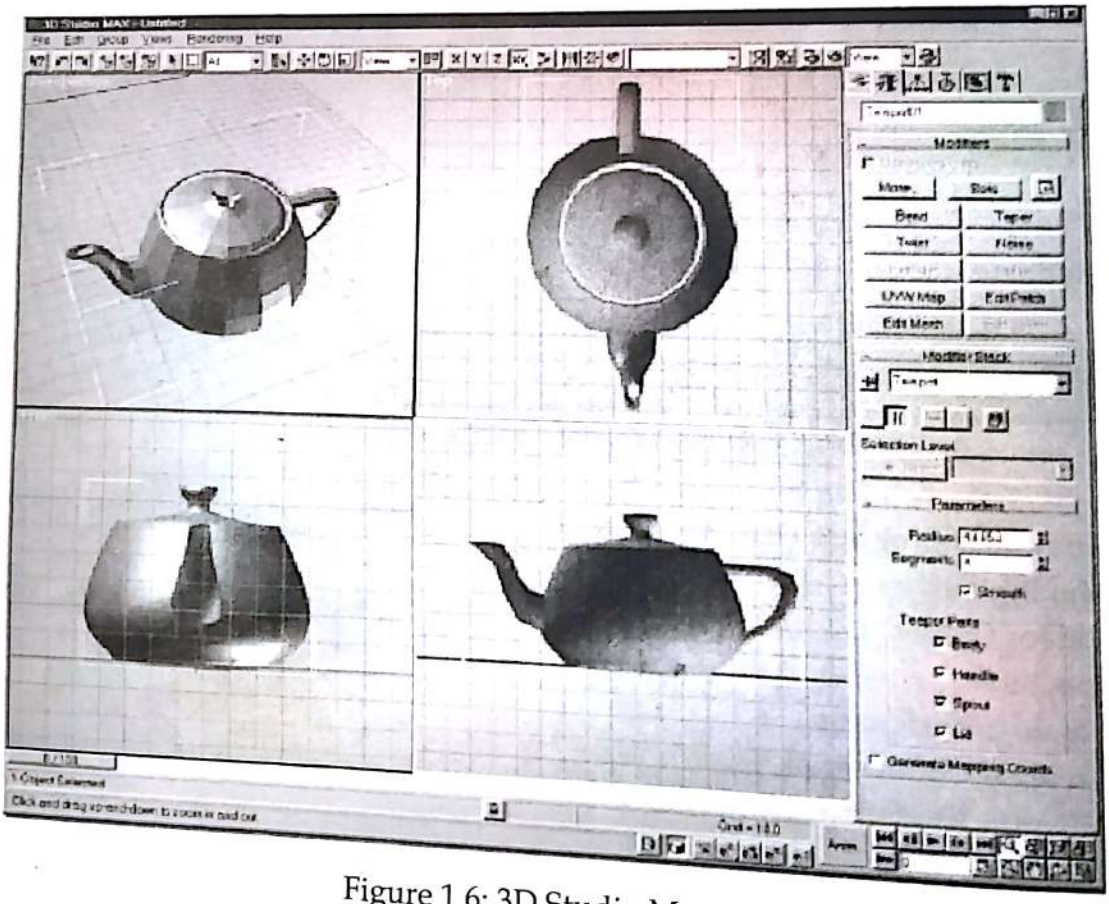


Figure 1.6: 3D Studio Max

2. Plotting

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

Figure 1.7: Cannon iPF607-A1 CAD Plotter

3. Office automation and electronic publishing

Computer graphics has facilitated the office automation and electronic publishing which is also popularly known as desktop publishing, giving more power to the organizations to print the meaningful materials Pictures, Drawings, etc.

4. Computer Aided Drafting and Design

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

5. Scientific and business Visualization

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

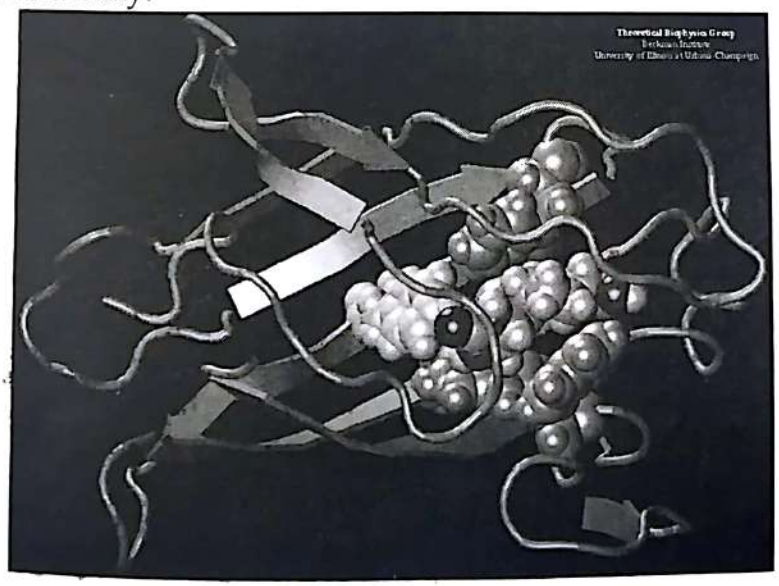


Figure 1.8: Scientific Visualization

6. Simulation and modeling

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

7. Entertainment

Computer graphics methods are now commonly used in making motions pictures, music videos and television shows. Sometimes the graphics scenes are displayed by themselves and sometimes graphics objects are combined with the actors and live scenes. A number of hit movies and shows are made using computer graphics technology. Disney movies such as Lion Kings and The Beauty of Beast, and other scientific movies like Jurassic Park, The lost world etc are the best example of the application of computer graphics in the field of entertainment. Instead of drawing all necessary frames with slightly changing scenes for the production of cartoon-film, only the key frames are sufficient for such cartoon-film where the in between frames are interpolated by the graphics system dramatically decreasing the cost of production while maintaining the quality. Computer and video games such FIFA, Doom, Pools are few to name where graphics is used extensively.

Figure 1.9: The flocking behaviour of the wild beast in Ling King

8. Art and commerce

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

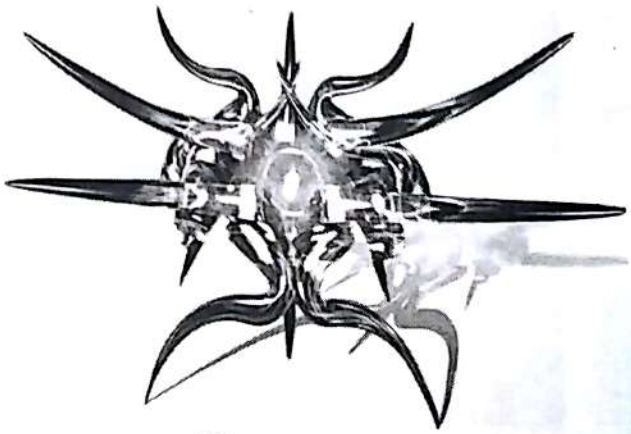


Figure 1.10: Art

9. Cartography

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

10. Medicine and Virtual Surgery

Computer graphics has extensive use in tomography and simulations of operations. Tomography is the technique that allows cross-sectional views of physiological systems in X-rays photography. Moreover, recent advancement is to make model and study physical functions to design artificial limbs and even plan and practice surgery.

11. History and Cultural Heritage

Another important application of computer graphics is in the field of history and cultural heritage. A lot of work is done in this area to preserve history and cultural heritage. The features so for provide are:

- Innovative graphics presentations developed for cultural heritage applications.
- Interactive computer techniques for education in art history and archeology
- New analytical tools designed for art historians
- Computer simulations of different classes of artistic media.

Figure 1.11: Virtual Surgery

Introduction to image processing

Image processing is a method to perform some operations on an image, in order to get an enhanced image or to extract some useful information from it. It is a type of signal processing in which input is an image and output may be image or characteristics associated with that image. Nowadays, image processing is among rapidly growing technologies. It forms core research area within engineering and computer science disciplines too.

Image processing basically includes the following three steps:

- Importing the image via image acquisition tools
- Analyzing and manipulating the image
- Output in which result can be altered image or report that is based on image analysis.

There are two types of methods used for image processing namely, **analogue** and **digital** image processing. Analogue image processing can be used for the hard copies like printouts and photographs. Image analysts use various fundamentals of interpretation while using these visual techniques. Digital image processing techniques help in manipulation of the digital images by using computers. The three general phases that all types of data have to undergo while using digital technique are pre-processing, enhancement, and display, information extraction.

Differences between computer graphics and Image processing

The process of Synthesize pictures from mathematical or geometrical models is called computer graphics. Simply in computer graphics we take pixels as mathematical input parameters and draw different images from these pixels. Whereas the process of analyze pictures to derive descriptions (often in mathematical or geometrical forms) of objects appeared in the pictures is called image processing. Image processing is the reverse process of computer graphics. Here we take any one of the picture as input parameter and return their mathematical form.

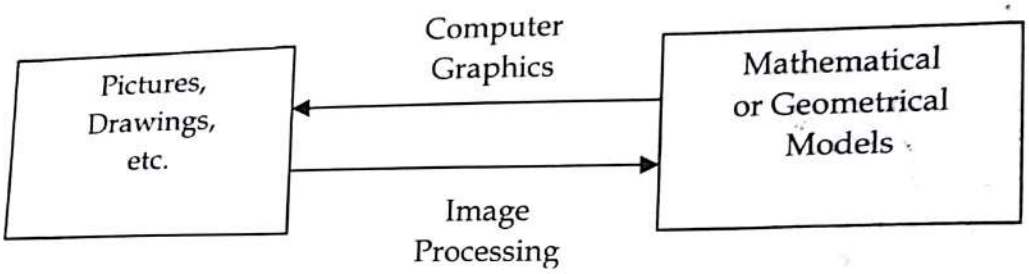


Figure 1.12: Image processing vs. computer graphics

Exercise

1. Define Computer Graphics. Why study this course?
2. What is Point in the Computer Graphics System?
3. What are the applications of computer graphics?
4. What do you mean by interactive computer Graphics?
5. What do you mean by GUI? What is the use of graphics in GUI? Explain.
6. What Is Rasterization?
7. What is meant by refreshing of the screen?
8. Write down brief history of computer graphics.
9. What do you mean by image processing? How it is differ from computer graphics?
10. Describe types of computer graphics with suitable example.
11. Describe uses of computer graphics in computer field.
12. Define pixel. What is the use of pixel in display devices?
13. What is the main objective of computer graphics in your study?
14. Plotting is the application of computer graphics. Justify.
15. What is interactive computer system? Explain.
16. What is resolution? What is the impact of resolution in display devices?
17. What is image processing? What are the applications of image processing in computer science?
18. How computer graphics used in entertainments? Explain.
19. How computer graphics used in education and training? Explain.
20. What is CAD? Describe with suitable example.

❑❑❑

Chapter

2

Hardware and Software Concept

STRUCTURE

2.0 Introduction

2.1 Input Devices

2.2 Output Devices

2.3 Types of Refresh cathode Ray tubes (CRT's)

2.4 Graphics Software

2.5 Software Standards

Introduction

A computer system consists of two major elements: hardware and software. Computer hardware is the collection of all the parts you can physically touch. Hardware refers to the physical elements of a computer. This is also sometime called the machinery or the equipment of the computer. Examples of hardware in a computer are the keyboard, the monitor, the mouse and the central processing unit. However, most of a computer's hardware cannot be seen; in other words, it is not an external element of the computer, but rather an internal one, surrounded by the computer's casing (tower). A computer's hardware is comprised of many different parts, but perhaps the most important of these is the motherboard. The motherboard is made up of even more parts that power and control the computer.

In contrast to software, hardware is a physical entity. Hardware and software are interconnected, without software; the hardware of a computer would have no function. However, without the creation of hardware to perform tasks directed by software via the central processing unit, software would be useless.

Hardware is limited to specifically designed tasks that are, taken independently, very simple. Software implements algorithms (problem solutions) that allow the computer to complete much more complex tasks.

Computer software, on the other hand, is not something you can touch. Software is a set of instructions for a computer to perform specific operations. Software, commonly known as programs or apps, consists of all the instructions that tell the hardware how to perform a task. These instructions come from a software developer in the form that will be accepted by the platform (operating system + CPU) that they are based on. For example, a program that is designed for the Windows operating system will only work for that specific operating system. Compatibility of software will vary as the design of the software and the operating system differ. Software that is designed for Windows XP may experience a compatibility issue when running under Windows 2000 or NT.

Software is capable of performing many tasks, as opposed to hardware which can only perform mechanical tasks that they are designed for. Software provides the means for accomplishing many different tasks with the same basic hardware. Practical computer systems divide software systems into two major classes:

- **System software:** Helps run the computer hardware and computer system itself. System software includes operating systems, device drivers, diagnostic tools and more. System software is almost always pre-installed on your computer.
- **Application software:** Allows users to accomplish one or more tasks. It includes word processing, web browsing and almost any other task for which you might install software. (Some application software is pre-installed on most computer systems.)

Input Devices

The devices that are used to take data from user are called input devices. An **input device** is a hardware or peripheral **device** used to send data to a computer. An **input device** allows users to communicate and feed instructions and data to computers for processing, display, storage

and transmission. Following are some of the important input devices which are used in a computer:

- Keyboard
- Mouse
- Joy Stick
- Light pen
- Track Ball
- Scanner
- Graphic Tablet
- Microphone
- Magnetic Ink Card Reader(MICR)
- Optical Character Reader(OCR)
- Bar Code Reader
- Optical Mark Reader(OMR)

Mouse

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

Figure 2.1: Mouse

Mechanical mouse

A mechanical mouse consists of a heavy rubber ball whose movement makes the cursor move on the screen. Commonly known as the 'rolling rubber ball' mouse, it is considerably heavy, thanks to the rubber ball, a few wheels, and a number of other mechanical parts present inside it. When a roller in the base of this mechanical mouse is moved, a pair of orthogonally arranged toothed wheels, each placed in between a LED and a photo detector, interrupts the light path. The numbers of interrupts so generated are used to report the mouse movements to the computer. Mechanical mice were quite popular in the past decade, but due to their clunkers design and relatively lessened durability, they were quickly replaced by optical mice.

Optical mouse

An optical mouse is technologically much more advanced than a mechanical mouse. Unlike the latter, an optical mouse is completely electronic and therefore has no moving parts. It consists of an LED (that generates the signature red light), a light-detector chip, a switch mechanism and a

few other simple components. Some mice have another LED that lights up a plastic strip installed at the back of the mouse as an indication of the mouse's operation.

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

Trackball

A trackball is a pointing device consisting of a ball housed in a socket containing sensors to detect rotation of the ball about two axes—like an upside-down mouse with an exposed protruding ball. The user rolls the ball with his thumb, fingers, or the palm of his hand to move a cursor. A potentiometer captures the track ball orientation which is calibrated with the translation of the cursor on screen. Tracker balls are common on CAD workstations for ease of use and, before the advent of the touchpad, on portable computers, where there may be no desk space on which to use a mouse.

Figure 2.2: Trackball

Touch Screen

A touch screen is a computer display screen that accepts input directly through the monitor. The screens are sensitive to pressure a user interacts with the computer by touching pictures or words on the screen. They are useful where environmental conditions prohibit the use of a keyboard or mouse. Touch screens are used with automated teller machine (ATM), computer based training devices etc. There are three types of touch screen technology:

- Surface wave
- Capacitive
- Resistive

Figure 2.3: Touch Screen

Tablet

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

- Electrical tablet
- Sonic tablet and
- Resistive tablet

Figure 2.4: Tablet

Electrical tablet

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

Sonic tablet

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

Resistive tablet

The tablet is just a piece of glass coated with a thin layer of conducting material. When a buttery-powered stylus is activated at certain position, it emits high-frequency radio signals which induces the radio signals on the conducting layer. The strength of the signal received at the edges of the tablet is used to calculate the position of the stylus. Several types of tablets are transparent, and thus can be backlit for digitizing x- ray films and photographic negatives. The resistive tablet can be used to digitize the objects on CRT because it can be curved to the shape of the CRT. The mechanism used in the electrical or sonic tablets can also be used to digitize the 3Dobjects.

Light pen

Light pen is an input device that utilizes a light-sensitive detector to select objects on a display screen. A light pen is similar to a mouse, except that with a light pen you can move the pointer and select objects on the display screen by directly pointing to the objects with the pen.

It is a pencil-shaped device to determine the coordinates of a point on the screen where it is activated such as pressing the button. Because of the following drawbacks the light pens are not popular now days. Light pen obscures the screen image as it is pointed to the required spot. Prolong use of it can cause arm fatigue. It cannot report the coordinates of a point that is completely black. As a remedy one can display a dark blue field in place of the regular image for a single frame time. It gives sometimes false reading due to background lighting in a room

Figure 2.5: Light Pen

Data Glove

A data glove is an interactive device, resembling a glove worn on the hand, which facilitates tactile sensing and fine-motion control in robotics and virtual reality . Data gloves are one of several types of electromechanical devices used in haptics applications.

Tactile sensing involves simulation of the sense of human touch and includes the ability to perceive pressure, linear force, torque, temperature, and surface texture. Fine-motion control involves the use of sensors to detect the movements of the user's hand and fingers, and the translation of these motions into signals that can be used by a virtual hand (for example, in gaming) or a robotic hand (for example, in remote-control surgery).

Figure 2.6: Data Glove

Touch panel

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

- Optical touch panel
- Sonic panel and
- Electrical touch panel

Optical touch panel

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

Sonic panel

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

Electrical touch panel

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

Keyboard

A keyboard creates a code such as ASCII uniquely corresponding to a pressed key. It usually consists of alphanumeric keys, function keys, cursor-control keys, and separate numeric pad. It is used to move the cursor, to select the menu item, pre-defined functions. In computer graphics

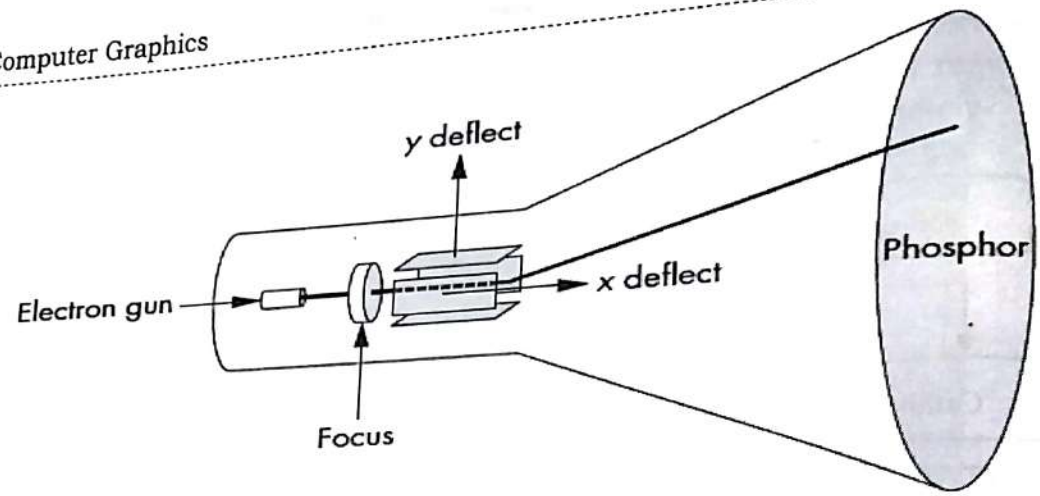


Figure 2.11: Cathode Ray Tube

The amount of light emitted by the phosphor coating depends on the number of electrons striking the screen. The brightness of the display is controlled by varying the voltage on the control grid.

Properties of CRT

Pixel or PEL

It is also called PEL. It comes from the words Picture Element (PEL). Pixel is the smallest addressable screen element. Each pixel has its own intensity, name or address by which we can control. It is a measure of screen resolution. Graphics monitors display pictures by dividing the display screen into thousands (or millions) of pixels, arranged in rows and columns.

The pixels are so close together that they appear connected. The number of bits used to represent each pixel determines how many colors or shades of gray can be displayed. For example, in 8-bit color mode, the color monitor uses 8 bits for each pixel, making it possible to display 2 to the 8th power (256) different colors or shades of gray.

On color monitors, each pixel is actually composed of three dots - a red, a blue, and a green one. Ideally, the three dots should all converge at the same point, but all monitors have some convergence error that can make color pixels appear fuzzy.

Frame buffer

It is the large contiguous piece of memory into which the intensity values for all pixels are placed. It contains the internal representation of the image. A frame buffer (frame buffer, or sometimes frame store) is a portion of RAM containing a bitmap that drives a video display. The frame buffer is the video memory that is used to hold or map the image displayed on the screen. The amount of memory required to hold the image depend primarily on the resolution of the screen image and the color depth. The formula to calculate how much video memory is required at a given resolution and bit depth is given below:

$$\text{Memory MB} = \frac{\text{x} - \text{resolution} \times \text{y} - \text{resolution} \times \text{Bit per pixel}}{8 \times 1024 \times 1024}$$

Pixel density

Number of pixels per unit area is called pixel density. Pixel density is a metric telling us how many pixels there are in a fixed area of a display. It's a very important metric because it lets us know how closely packed the pixels on a display area. This is something that determines the quality, clarity, and readability of the image displayed. It is usually measured in a unit called pixels per inch (ppi). If a careen of size 4 inch by 3 inch and of resolution 800 × 600 then pixel density is,

$$\text{Pixel Density (PD)} = \frac{800}{4} \times \frac{600}{3} = 200 \times 200$$

If we have two displays with the same resolution, the smaller one will have the higher pixel density. And if we have displays with the same size, the one with a higher resolution will have the higher pixel density. So it's clear that the two parameters that we need in order to calculate the pixel density are display size and resolution. We can calculate the pixel density knowing the resolution and size (diagonal) or a display as below,

$$\text{Pixel Density (PD)} = \frac{\sqrt{\text{Width}^2 + \text{Length}^2}}{\text{Screen Size (diagonal)}}$$

Example 2.1: Let's say we have a computer display that is 21.5 inch with a 1920×1080 resolution.

Display size = 21.5 inch

Width = 1920, Length = 1080

Solution:

Diagonal resolution = square root of (19202 + 10802)

Diagonal resolution = square root of (3686400 + 1166400)

Diagonal resolution = square root of 4852800

Diagonal resolution = 2203 (close approximation)

Pixel density = diagonal resolution / display size

Pixel density = 2203 / 21.5

Pixel density = 103 ppi (close approximation)

Persistence

Persistence is the one of the major property of phosphor used in CRT's. It means how long phosphors continue to emit light after the electron beam is removed. Persistence is defined as the time it takes the emitted light from the screen to decay to one-tenth of its original intensity. Lower persistence phosphors require higher refresh rates to maintain a picture on the screen. A phosphor with lower persistence is useful for animation and a higher-persistence phosphor is useful for displaying highly complex static picture. Graphics monitor are usually constructed with the persistence 10 to 60 microseconds.

Resolution

The maximum number of points (pixel) that can be displayed without overlap on a CRT is referred to as there solution. It is also defined as maximum number of points displayed horizontally and vertically without overlap on a display screen. More precise definition of resolution is no of dots per inch (dpi/pixel per inch) that can be plotted horizontally and

vertically. Resolution of 1280 × 720 means that there are 1280 horizontal lines and 720 vertic lines or 1280 pixels horizontally and 720 pixels vertically. We can say that the higher is the pix resolution, the higher is the quality of the image. There are two types of resolutions:

- Image resolution and
- Screen resolution

Image resolution is defined as the distance from one pixel to the next pixel i.e. pixel spacing. Screen resolution is defined as the number of pixels in the horizontal and vertical directions o the screen.

Megapixels

We can calculate mega pixels of a display device using pixel resolution.
Column pixels (width) × row pixels (height) / 1 Million.
The size of an image can be defined by its pixel resolution.
Size = pixel resolution × bpp (bits per pixel)

Example 2.2: Let's say we have an image of dimension: 2500 × 3192.
Its pixel resolution = 2500 × 3192 = 7982350 bytes.
Dividing it by 1 million = 7.9 = 8 mega pixel (approximately).

Refresh Rate

Light emitted by phosphor fades very rapidly, so to keep the drawn picture glowing constantly, it is required to redraw the picture repeatedly and quickly directing the electron beam back over the some point. This process is called refresh operation.

The no of times/sec the image is redrawn to give a feeling of non-flickering pictures is called refresh-rate. Refresh rates are described in units of cycles per second, or Hertz (Hz), where a cycle corresponds to one frame. It is also called a frame rate, horizontal scan rate, and vertical scan rate. A refresh rate of 70 Hz means that the image is redrawn 70 times second. When choosing a monitor, one of the factors that the customer usually considers is the refresh rate. A high refresh rate is important in providing a clear picture and avoiding eye fatigue. An image appears on screen when electron beams strike the surface of the screen in a **zig-zag** pattern. A refresh rate is the number of times a screen is redrawn in one second and is measured in Hertz (Hz). Therefore, a monitor with a refresh rate of 85 Hz is redrawn 85 times per second. A monitor should be "flicker-free meaning that the image is redrawn quickly enough so that the user cannot detect flicker, a source of eye strain. Today, a refresh rate of 75 Hz or above is considered to be flicker-free.

Bit depth (color depth)

Color depth or **color depth** also known as **bit depth**, is either the number of **bits** used to indicate the **color** of a single pixel, in a bitmapped image or video frame buffer, or the number of **bits** used for each **color** component of a single pixel. It is defined as the number of bits assigned to each pixel in the image. It specifies the number of colors that a monitor can display.

Aspect ratio

It is defined as the number of the horizontal points to the vertical points required to produce equal length lines in both directions on the screen. Aspect ratio is the ratio between width of an image and the height of an image. It is commonly explained as two numbers separated by a colon (8:9). This ratio differs in different images, and in different screens. The common aspect ratios are:

1.33:1, 1.37:1, 1.43:1, 1.50:1, 1.56:1, 1.66:1, 1.75:1, 1.78:1, 1.85:1, 2.00:1, etc.

Aspect ratio maintains a balance between the appearances of an image on the screen, means it maintains a ratio between horizontal and vertical pixels. It does not let the image to get distorted when aspect ratio is increased. Aspect ratio of the screen can be calculate by following formula,

$$\text{Aspect ratio (AR)} = \frac{\text{Width of the image}}{\text{Height of the image}}$$

Bit map and pixel map

An image of two colors (generally black and white) is called as a bitmap. An image of more than two colors is called as a pixel map.

Scan line

Time taken to access horizontal pixels is called scan line. A scan line is one line, or row, in a raster scanning pattern, such as a line of video on a cathode ray tube (CRT) display of television set or computer monitor.

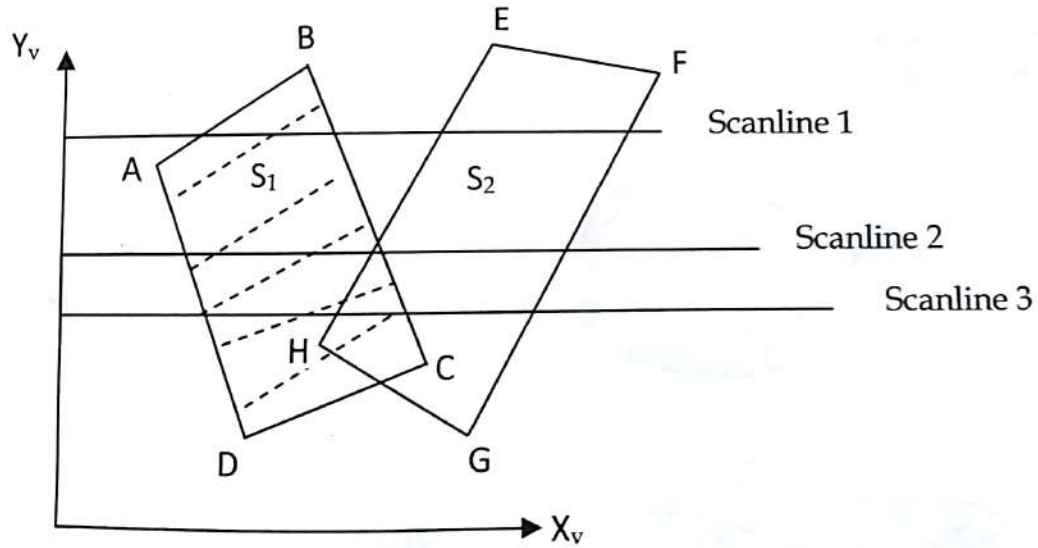


Figure 2.12: Scan Line

Types of Refresh cathode Ray tubes (CRT's)

A **cathode ray tube** (CRT) is a specialized vacuum tube in which images are produced when an electron beam strikes phosphorescent surface. Most desktop computer displays make use of CRTs. There are two types of Refresh CRT's:

- Raster-Scan Displays
- Random-Scan Displays

Raster-Scan Display

In a raster scan system, the electron beam is swept across the screen, one row at a time from top to bottom. As the electron beam moves across each row, the beam intensity is turned on and off to create a pattern of illuminated spots. Picture definition is stored in memory area called the **Refresh Buffer** or **Frame Buffer**. This memory area holds the set of intensity values for all the screen points. Stored intensity values are then retrieved from the refresh buffer and "painted" on the screen one row (scan line) at a time as shown in the following illustration. Each screen point is referred to as a **pixel (picture element)** or **pel**. At the end of each scan line, the electron beam returns to the left side of the screen to begin displaying the next scan line. In comparisons of random scan display method it is more accurate method.

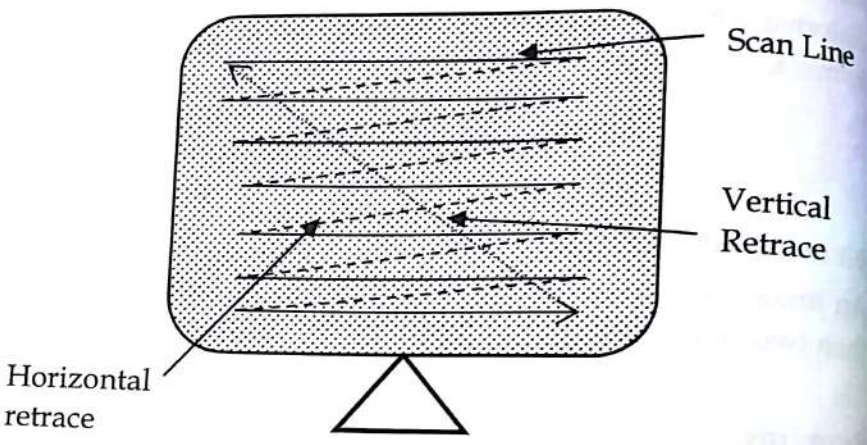


Figure 2.13: Raster Scan Display

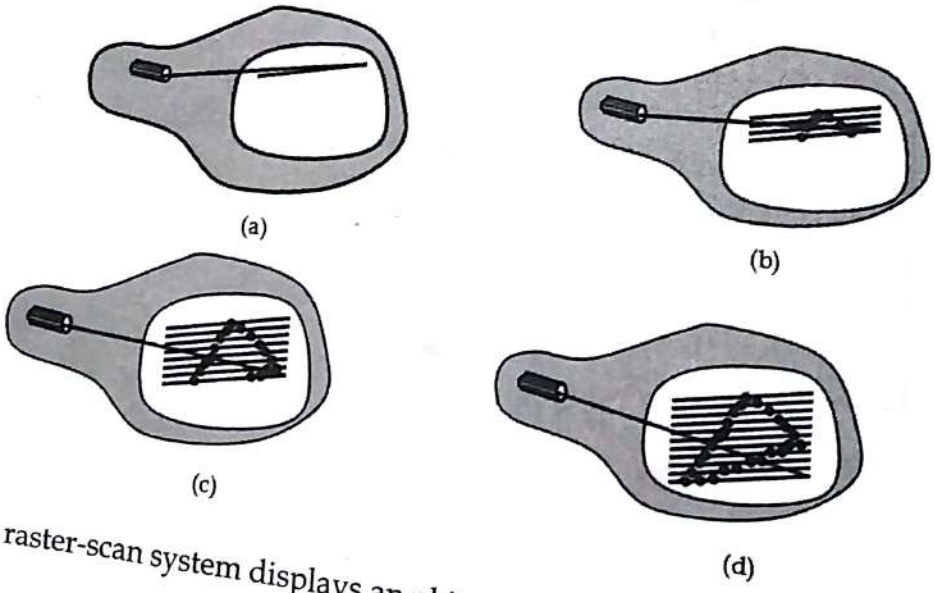


Figure 2.14: A raster-scan system displays an object as a set of points across each screen scan line

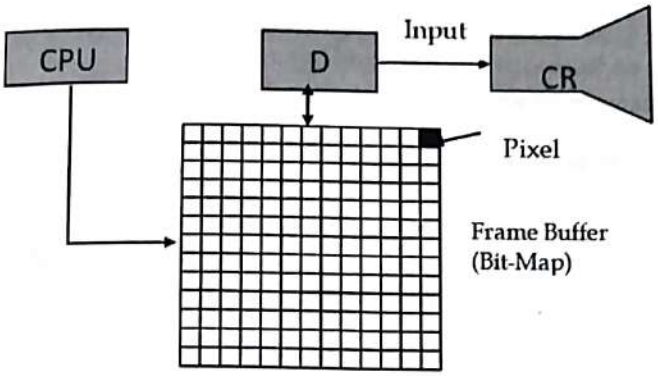


Figure 2.15: Raster Scan display system

The stored intensity value is retrieved from frame buffer and painted on the scan line at a time. The stored intensity value is retrieved from frame buffer and painted on the screen. Refreshing on Raster-Scan display is carried out at the rate of 60 or higher frames per second. 60 frames per second is also termed as 60 cycle per second usually used unit Hertz (HZ). Most of display devices are based on this technology. For example, CRT, color CRT, LCD, and LED etc.

Horizontal retrace/ vertical retrace

Returning of electron beam from right end to left end after refreshing each scan line is horizontal retrace. At the end of each frame, the electron beam returns to the top left corner to begin next frame called vertical retrace.

Architecture of Raster-Scan System

The raster graphics systems typically consist of several processing units. CPU is the main processing unit of computer systems. Besides CPU, graphics system consists of a special buffer processor called video controller or display controller. It is used to control the operation of the display device. In addition, to the video controller, raster scan systems can have other processors as co-processors which are called graphic controller or display processors. A fixed area of system memory is reserved for the frame buffer. The video controller has the direct access to the frame buffer for refreshing the screen. The video controller cycles through the frame buffer, one scan line at a time, typically at 60 times per second or higher. The contents of frame buffer are used to control the CRT beam's intensity or color. The organization of simple raster system is shown in the figure below.

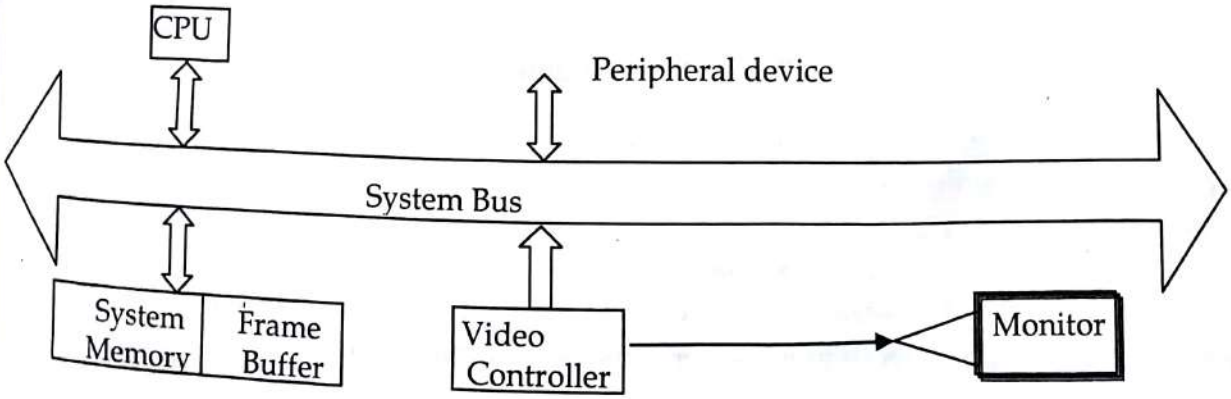


Figure 2.16: Architecture of simple Raster-scan system.

Advantages of Raster scale system

- Show Realistic pictures
- Million Unique hues can be produced
- Shadow scenes are conceivable

Disadvantages of Raster scale system

- Low Resolution
- Electron beam coordinated to whole screen not exclusively to those parts of the scre where picture is to be drawn so tedious when the drawn picture estimate is especia not as much as the whole screen.
- Expensive

Random-Scan (Vector display) system

In this technique, the electron beam is directed only to the part of the screen where the picture to be drawn rather than scanning from left to right and top to bottom as in raster scan. It is al called vector display, stroke-writing display, or calligraphic display. Picture definition is stor as a set of line-drawing commands in an area of memory referred to as the refresh display fi To display a specified picture, the system cycles through the set of commands in the displ file, drawing each component line in turn. After all the line-drawing commands are processe the system cycles back to the first line command in the list. Random-scan displays are designe to draw all the component lines of a picture 30 to 60 times each second.

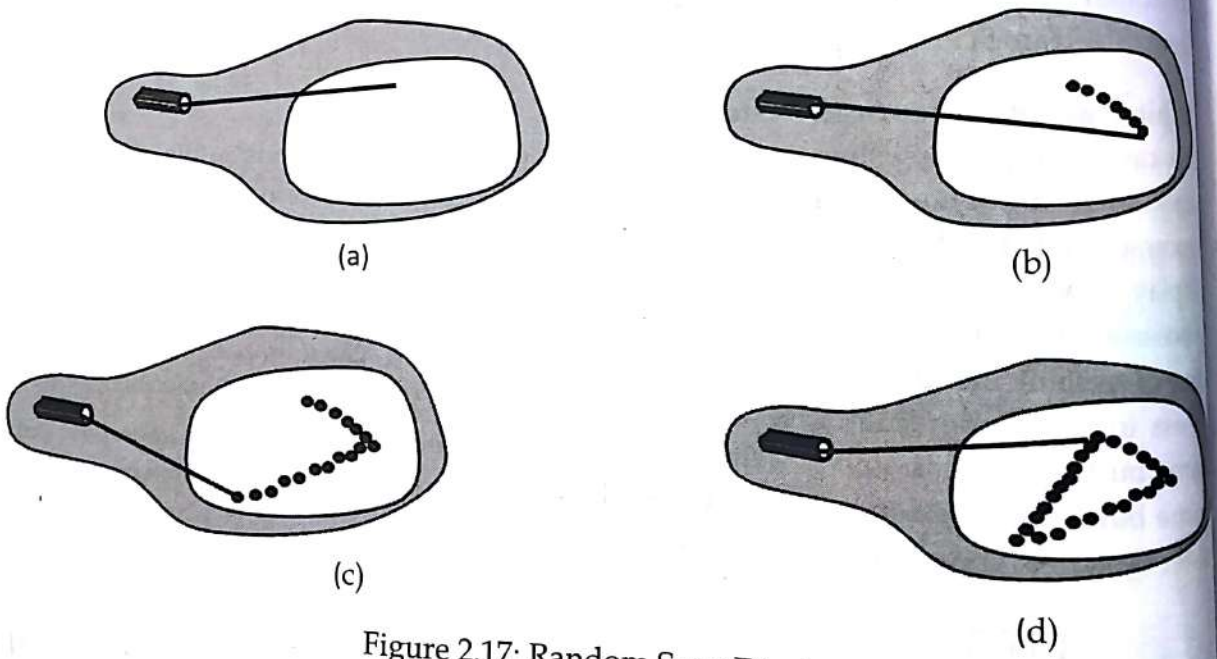


Figure 2.17: Random Scan Display

The refresh rate of vector display depends upon the no of lines to be displayed for any image Picture definition is stored as a set of line drawing instructions in an area of memory called th refresh display file (Display list or display file). It is designed for drawing all component line 30 to 60 times per second. Such systems are designed for line-drawing applications and canno display realistic shaded scenes. Since CRT beam directly follows the line path, the vecto display system produce smooth line. Plotter is the best example of this system.

Architecture of Random Scan Systems

Vector display system consists of additional processing unit along with CPU which is called the display processor or graphics controller. Picture definition is stored as a set of line drawing commands in an area of memory called display list. A display list (or display file) contains a series of graphics commands that define an output image. This display list is then accessed by the display processor to create an image.

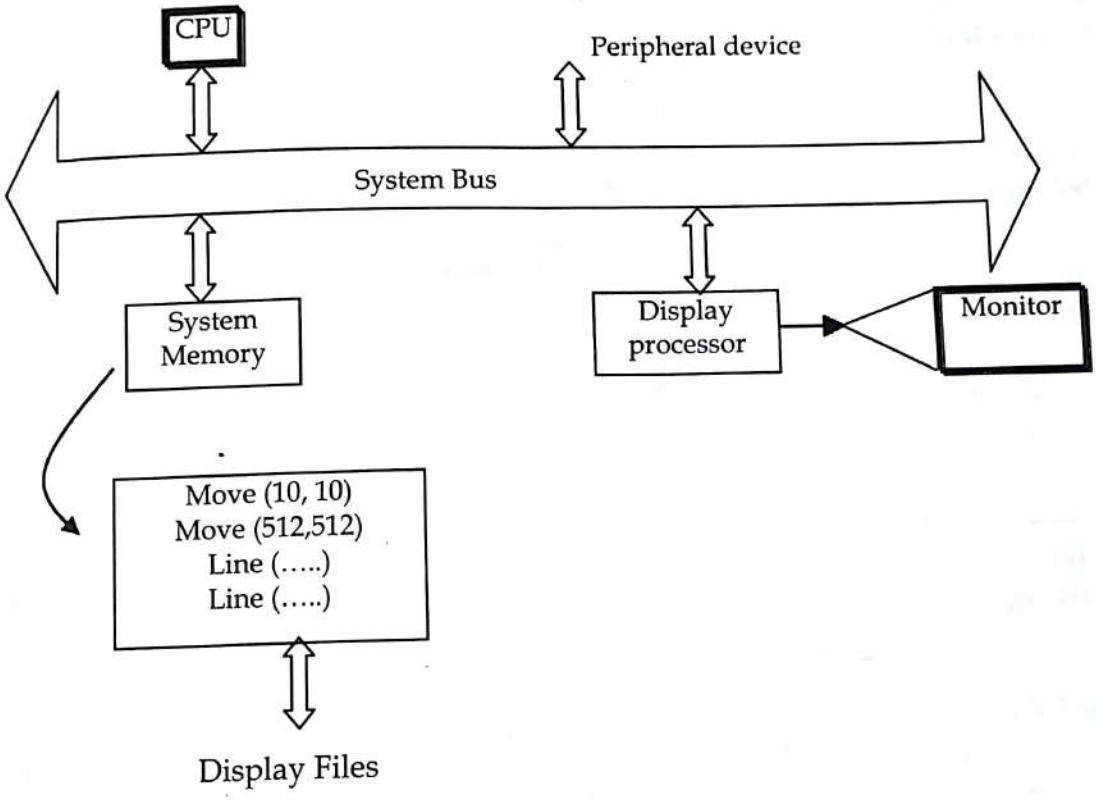


Figure 2.18: Architecture of Vector Display System

Advantages of Random scan display

- high resolution
- Produce smooth line drawings.
- Disadvantages of random scan display
- costlier
- Just does wire frame.
- Complex scene cause visible flicker.

Comparison between Raster and Random (Vector) scan display systems

	Raster Scan System	Random Scan System
Resolution	It has poor or less resolution because picture definition is stored as an intensity value.	It has High Resolution because it stores picture definition as a set of line commands.

Electron-Beam	Electron beam is directed from top to bottom and one row at a time on screen, but electron beam is directed to whole screen.	Electron beam is directed to only that part of screen where picture is required to be drawn, one line at a time so also called vector display.
Cost	It is less expensive than Random Scan System.	It is Costlier than Raster Scan System.
Refresh Rate	The refresh rate is independent of picture complexity. Refresh rate is 60 to 80 frames per second.	Refresh Rate depends on the number of lines to be displayed i.e. 30 to 60 times per second.
Picture Definition	It Stores picture definition in Refresh Buffer also called Frame Buffer.	It Stores picture definition as a set of line commands called Refresh Display File.
Line Drawing	Zig-Zag line is produced because plotted values are discrete.	Smooth line is produced because directly the line path is followed by electron beam.
Realism in display	It contains shadow, advance shading and hidden surface technique so gives the realistic display of scenes.	It does not contain shadow and hidden surface technique so it cannot give realistic display of scenes.
Image Drawing	It uses Pixels along scan lines for drawing an image.	It is designed for line drawing applications and uses various mathematical functions to draw.

Color CRT

Normal CRT generates only a single color at a time but we can also generate multi- colors using multilayer phosphor. In color CRT, the phosphor on the face of CRT screen is laid in to different fashion. The basic principle behind colored displays is that combining the 3 basic colors –Red, Blue and Green can produce every color. By choosing different ratios of these three colors we can produce different colors – millions of them in-fact. We also have basic phosphors, which can produce these basic colors. So, one should have a technology to combine them in different combinations. There are two popular techniques for producing color displays with a CRT are:

- Beam penetration method
- Shadow mask method

Beam Penetration method

This CRT is similar to the simple CRT, but it makes use of multi colored phosphorus of number of layers. Each phosphorus layer is responsible for one color. All other arrangements are similar to simple CRT. It can produce a maximum of 4 to 5 colors. The organization is something like this - The red, green and blue phosphorus are coated in layers - one behind the other. If a low speed beam strikes the CRT, only the red colored phosphorus is activated, a slightly accelerated beam would activate both red and green (because it can penetrate deeper) and a much more activated one would add the blue component also. But the basic problem is a reliable technology to accelerate the electronic beam to precise levels to get the exact colors - it is easier

said than done. However, a limited range of colors can be conveniently produced using the concept.

This method is commonly used for random scan display or vector display. In random scan display CRT, the two layers of phosphor usually red and green are coated on CRT screen. Display color depends upon how far electrons beam penetrate the phosphor layers. Slow electron excite only red layer so that we can see red color displayed on the screen pixel where the beam strikes. Fast electron beam excite green layer penetrating the red layer and we can see the green color displayed at the corresponding position. Intermediate is combination of red and green so two additional colors are possible – orange and yellow so only four colors are possible. In this type of display method no good quality picture occurs.

Advantages of beam penetration CRT

- Half cost as compared to that of shadow mask CRT
- It is an inexpensive way to produce color in random scan monitor
- Its resolution is better

Disadvantages of beam penetration CRT

- A problem arises at the time of switching of colors as it is time consuming.
- It consumes significant amount of accelerating potential in order to switch color.

Shadow Mask Method

In this display, there are three electron guns, one for each primary color i.e. red, green and blue. The electron guns are frequently arranged in a triangular pattern called delta corresponding to a similar triangular pattern of red, green and blue phosphor dots on the face of the CRT. To insure that individual electron guns excite the correct phosphor dots, a perforated metal grid (Shadow mask) is placed between the electron guns and the face of the CRT. Shadow mask method is used for raster scan system so they can produce wide range of colors. In shadow mask color CRT, the phosphor on the face of the screen is laid out in a precise geometric pattern. There are two primary variations.

- The stripe pattern of inline tube
- The delta pattern of delta tube

The color variations in a shadow mask CRT can be obtained by varying the intensity levels of the three beams.

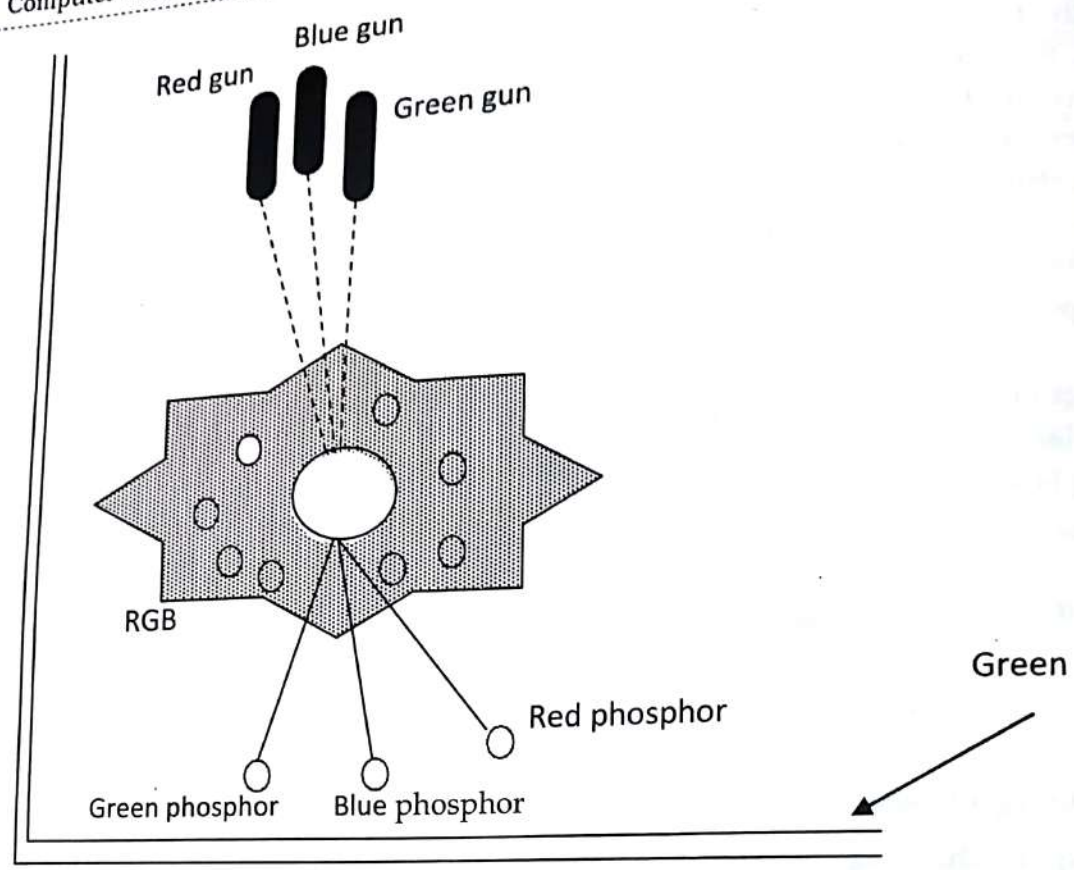


Figure 2.19: Shadow Mask Method

Advantages of shadow mask method

- Produces a much wider range of colors than the beam penetration method.
- An in-line arrangement of RGB color dots on the screen along one scan line is used in higher resolution color CRTs.
- It produces realistic images

Disadvantages of shadow mask method

- It is very difficult to coverage all three beams on same hole in shadow mask. This is because the screen has a limited area.
- They have poor resolution

Differences between Beam Penetration method and Shadow Mask method

	Beam Penetration method	Shadow Mask method
Where Used	It is used with **Random Scan System** to display color.	It is used with **Raster Scan System** to display color.
Colors	It can displays only four colors i.e. Red, Green, Orange and Yellow.	It can display Millions of colors.
Color Dependency	Fewer colors are available because the colors in Beam penetration depend on the speed of the electron beam.	Millions of colors are available because the color in Shadow Mask depends on the type of the ray.

Cost	It is **Less Expensive** as compared to Shadow Mask.	It is **More Expensive** than other methods.
Picture Quality	Quality of picture is not so good i.e. Poor with Beam Penetration Method.	Shadow Mask gives realism in picture with shadow effect and millions of color.
Resolution	It gives High Resolution.	It gives Low Resolution.
Criteria	In Beam Penetration method, Color display depends on how far electron excites outer Red layer and then Green layer.	In Shadow Mask Method, there are no such criteria for producing colors. It is used in computers, in color TV etc.

Flat Panel Display

A flat-panel display is a thin screen display found on all portable computers and is the new standard for desktop computers. Unlike CRT monitors, flat-panel displays use liquid-crystal display (LCD) or light-emitting diode (LED) technology to make them much lighter and thinner compared to a traditional CRT monitor. Flat panel display refers to a class of video device that have reduced volume, weight and power requirement compared to a CRT. These emerging display technologies tend to replace CRT monitors. Current uses of flat-panel displays include TV monitors, calculators, pocket video games, laptops, displays in airlines and ads etc.

Figure 2.20: Flat Panel Display

There are two categories of flat-panel displays:

Emissive displays

The emissive displays (or emitters) are devices that displays and light - emitting diodes are examples of emissive displays. Non-emissive displays (or non-emitters) use optical effects to convert sunlight or light from some other source into graphics patterns. Plasma panels, electroluminescent displays and light-emitting diodes (LED) are examples of emission display devices.

Non-emissive displays

Use optical effects to convert sunlight or light from other sources into graphics patterns. Example: liquid-crystal displays (LCD).

Advantages

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

Disadvantages

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

Differences between CRT and LCD display

Topic	CRT	LCD
Size	Because of the CRT in a CRT monitor the physical size of these displays is much larger than an LCD and usually awkward on small desks.	LCD monitors are much thinner than CRT monitors, being only a few inches in thickness (some can be near 1 inch thick). They can fit into smaller, tighter spaces, whereas a CRT monitor can't in most cases.
Dead pixels	Although a CRT can have display issues there is no such thing as a dead pixel on a CRT monitor. Many issues can also be fixed by degaussing the monitor.	LCD monitors can encounter dead pixels, which causes small black or other colored dots in the display.
Weight	A CRT monitor can weigh 40 pounds or more depending on the size of the monitor.	LCD monitors can be pretty light, weighing as little as 8 to 10 pounds.
Price	Because of the popularity of LCD monitors the price of most CRT monitors is very cheap. You can also usually pick up a used CRT for next to nothing.	LCD monitors are a newer technology and have more demand so will be more expensive than a CRT.
Viewable area	The frame around the glass screen of the monitor causes the viewable area of the screen to be smaller than an LCD.	LCD monitors have a slightly bigger viewable area than a CRT monitor. A 19" LCD monitor has a diagonal screen size of 19" and a 19" CRT monitor has a diagonal screens size of about 18".
Picture	Because of the older technology most CRT monitors will not have as good as quality as picture as most LCD displays.	Depending on the quality of the LCD monitor, the picture quality can be quite superb and amazing, almost like looking out a window.
Viewing angle	Almost every CRT have a better viewing angle than many LCD displays.	Not all LCD monitors can be viewed at every angle, which makes it difficult for anyone who is not in front of the monitor to see the screen.
Glare	Most monitors have a glass screen, which can cause much more glare than an LCD.	An LCD monitor doesn't have a glass screen, virtually eliminating any glare.
Burn	If the same image is left on a CRT for days the image can burn into the display causing a permanent ghost image to appear on the screen.	Unlike a CRT an LCD monitor does is not affected by a burn in or ghosting problem.
Flicker	Some people can see the flicker in a CRT monitor, which is due in part to the refresh rate of the screen refreshing the image.	On an LCD monitor, screen flicker is very minimal if noticeable at all because of the higher refresh rate.

Response	The response on early LCD monitors was not as good as most monitors at the time, making games and movies not as enjoyable to play.	Any LCD made within the last 5 years has no refresh rate issues. Making the display just as enjoyable as CRT monitors.
Power	A 17" CRT monitor will use as much as 80 watts, depending on the age.	LCD monitors are very energy efficient. A 19" LCD monitor only uses about 17 to 31 watts on average.

Graphics Software

Interactive graphics allow users to make change over the displayed objects. Several graphics software packages are now available. There are two general classifications for graphics software:

- General programming packages
- Special-purpose application packages

General programming packages

It contains graphics functions that can be used with high level programming languages such as C, FORTRAN, JAVA etc. **Example**: OpenGL (Graphics library). A general-purpose graphics package provides users with a variety of functions for creating and manipulating pictures. These graphic functions include tools for generating picture components, setting color, selecting views, and applying transformations.

Special-purpose application packages

It is specifically designed for particular applications. Maya, CINEMA 4D are particularly used for animations, different types of CAD applications are designed for medical and business purposes. These are primarily oriented to non-programmers.

Software Standards

Many problems have been encountered due to the plate-form dependency of the graphic software. So the primary goal of standardized graphics software is portability. When graphics packages are designed with standard graphics functions (set of specifications that is independent of any programming languages), software can be easily moved from one H/W system to another. And it can be used in different implementations and applications.

a. **Graphical Kernel System (GKS)**
   GKS was the first graphics software standard adopted by the international standards organization (ISO). It was originally designed as a 2-dimensional graphics package. GKS supports the grouping of logically related primitives such as; lines, polygons, character strings.

b. **Programmer's Hierarchical Interactive Graphics System (PHIGS)**
   It is an extension of GKS. Increased capabilities in object modeling, color specifications, surface rendering, and picture manipulations are provided in PHIGS. PHIGS include all primitives supported by GKS; in addition it also includes geometric transformations (Scaling, Translation, and Rotation).

c. **PHIGS+**
It is extension of earlier PHIGS. 3D surface shading capabilities are added to the PHIG

Exercise

Short Questions

1. Difference between Raster Scan Display and Random Scan Display.
2. Define DVST - Direct View Storage Tube
3. What is Color CRT Display? Explain Beam-penetration and Shadow-mask method.
4. What are frame grabbers? Are frame buffers different from frame grabbers?
5. Draw a neat block diagram, to explain the architecture of a raster display.
6. Give the logical organization of the video controller in a raster display system.
7. What is refresh buffer? Identify the contents and the organization of the refresh buffe for the case of raster display and vector display.
8. Describe the function of an image scanner.
9. What role does CCD play in an image scanner?
10. How are different shades of color generated on the RGB monitors?
11. What is the role of shadow masks in graphics monitors?
12. What do you understand by VGA and SVGA monitors?
13. What is computer graphics? Indicate five practical applications of computer graphics.
14. Discuss in brief different interactive picture construction techniques.
15. How is color depth and resolution of an image related to the video memory requirement?
16. What is the fundamental difference in the method of operation of a monochrome CRT and a color CRT?
17. Compare storage type CRT against refresh type CRT display. List the important properties of phosphor being used in CRTs.
18. Compare and contrast the operating characteristics of raster refresh systems, plasma panels and LCDs.
19. Briefly explain two main classes of hardware device for user interaction.
20. If a monitor screen has 525 scan lines and an aspect ratio of 3:4 and if each pixel contains 8 bits worth of intensity information, how many bits per second are required to show 30 frames each second?

Numerical problems

1. Consider three different raster systems with resolutions of 640 x 480, 1280 x 1024, and 2560 x 2048
2. What size is frame buffer (in bytes) for each of these systems to store 12 bits per pixel?

3. How much storage (in bytes) is required for each system if 24 bits per pixel are to be stored?
4. Suppose RGB raster system is to be designed using on 8 inch X 10 inch screen with a resolution of 100 pixels per inch in each direction. If we want to store 6 bits per pixel in the frame buffer, how much storage (in bytes) do we need for frame buffer?
5. If you are supposed to create an animated movie of 20 minutes and your video is of 30fps (frames/second). Calculate the number of frames should be in this video.
6. Find out the time required to scan one row of screen if the screen resolution is 20MP (mega pixel), aspect ratio 5/4 and refresh frequency is 30Hz.
7. How much time is spent scanning across each row of pixels during screen refresh on a raster system with resolution of 1280 X 1024 and a refresh rate of 60 frames per second?
8. Consider three different raster systems with resolutions of 640 x 480, 1280 x 1024, and 2560 x 2048.
   a) What size is frame buffer (in bytes) for each of these systems to store 12 bits per pixel?
   b) How much storage (in bytes) is required for each system if 24 bits per pixel are to be stored?
9. Consider two raster systems with the resolutions of 640 x 480 and 1280 x 1024.
   a) How many pixels could be accessed per second in each of these systems by a display controller that refreshes the screen at a rate of 60 frames per second?
   b) What is the access time per pixel in each system?
10. Given a 25cm x 20cm display operating in 1024 x 768 x 16 color mode which is refreshed 30 times per second, and for which 10% of the refresh cycle is spent in retrace, calculate
    a. the pixel aspect ratio,
    b. the size of the frame buffer, and
    c. The required data transfer rate in kilobytes per second.
11. What is the size of a pixel on a 21-inch diagonal screen with physical aspect ratio 8:5 operating in 1152 x 800 modes?
12. A raster display system operating in 1280 x 1024 mode displays 60 frames per second, and 8% of its operating time is spent in retrace. In what amount of time must the video controller be able to write a pixel to the screen?
13. Consider a raster system with the resolution of 1024 x 768 pixels and the color palette calls for 65,536 colors. What is the minimum amount of video RAM that the computer must have to support the above-mentioned resolution and number of colors?
14. How Many k bytes does a frame buffer need in a 600 x 400 pixel?
15. How much time is spent scanning across each row of pixels during screen refresh on a raster system with resolution of 1280 X 1024 and a refresh rate of 60 frames per second?
16. Suppose RGB raster system is to be designed using on 8 inch X 10 inch screen with a resolution of 100 pixels per inch in each direction. If we want to store 6 bits per pixel in the frame buffer, how much storage (in bytes) do we need for frame buffer?
17. Consider two raster systems with the resolutions of 640 x 480 and 1280 x 1024. How many pixels could be accessed per second in each of these systems by a display controller that refreshes the screen at a rate of 60 frames per second?

18. How long would it take to load a 24-bit per pixel frame buffer with a resolution of 12
by 1024 using this same transfer rate?
19. Consider a non-interlaced raster monitor of resolution 1280x1024. If horizontal ar
vertical retrace times are 20 microseconds each, then calculate the time available
display a pixel. Assume refresh rate of 60 frames per second.
20. How much time is spent in scanning across each row of pixels during screen refresh o
a raster system with are solution of 1280 by 1024 and a refresh rate of 60 frames pe
second? Assume horizontal and vertical retrace times are negligible.

❑❑❑

tion of 1280

zontal and
vailable to

refresh on
rames per

Chapter

3

Scan Conversion Algorithm

STRUCTURE

3.0 Introduction

3.1 Line Drawing Algorithm

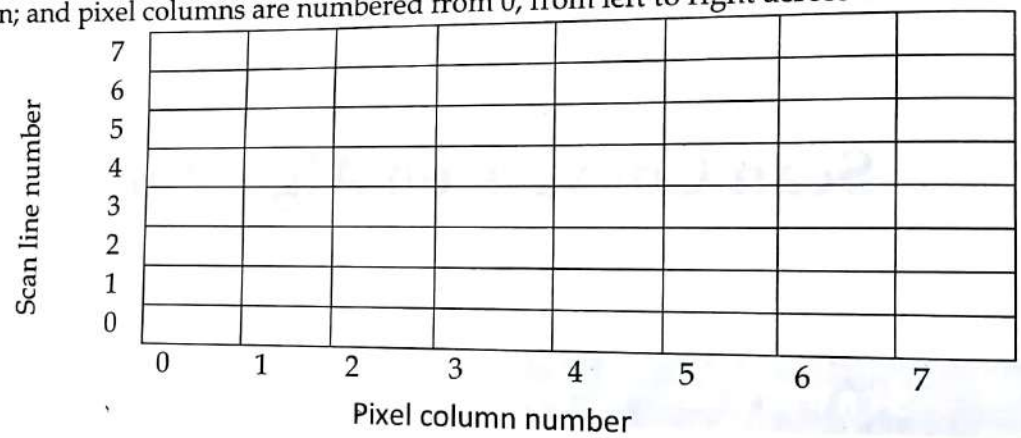
3.2 Circle Algorithm

3.3 Ellipse Algorithm

3.4 Area Filling

3.0 Introduction

This chapter introduces basic concepts that are required for the understanding of two dimensional graphics. Almost all output devices for graphics like computer monitors or printers are pixel-oriented. Therefore, it is crucial to distinguish between the representation of images on these devices and the model of the image itself which is usually not pixel-oriented but defined as scalable vector graphics, i.e., floating point values are used for coordinates. The basic building blocks for pictures are referred to as output primitives. They include character strings, and geometric entities such as points, straight lines, curved lines, polygons, circles etc. Points and straight line segments are the most basic components of a picture. A picture can be displayed in several ways. Assuming we have raster display, a picture is completely specified by the set of intensities for the pixel positions in the display. The process in which the object is represented as the collection of discrete pixels is called scan conversion. At the other extreme, we can describe a picture as a set of geometrical objects, such as lines, triangles, polygons etc. Scan lines are numbered consecutively from 0, starting at the bottom of the screen; and pixel columns are numbered from 0, from left to right across each scan line.



3.1 Line Drawing Algorithm

A line drawing algorithm is a graphical algorithm for approximating a line segment on discrete graphical media. On discrete media, such as pixel-based displays and printers, line drawing requires such an approximation (in nontrivial cases). Basic algorithmsrasterizing lines in one color. There are mainly three most widely used line drawing algorithms:

1. Direct use of Line Equation
2. Digital Differential Analyzer Algorithm (DDA)
3. Bresenham Line Drawing Algorithm (BSA)

Direct use of Line Equation

In this method simply calculate the corresponding y coordinate for each unit x coordinate. The slope-intercept equation of a straight line is:

$$y = mx + b \quad \text{.....(i)}$$

Where, m = slope of line and, b = y-intercept.

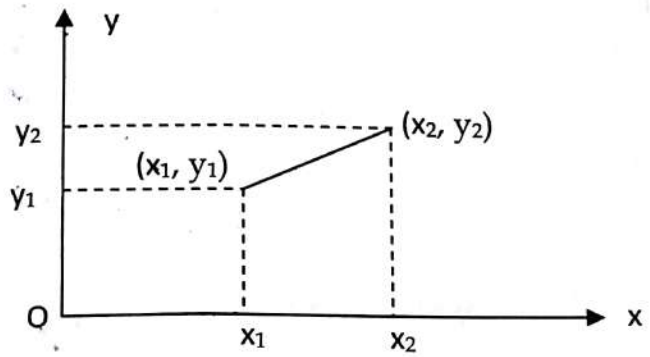


Figure 3.1: Line path between two end points (x_1, y_1), and (x_2, y_2)

Suppose (x_1, y_1) and (x_2, y_2) are the two end points of a line segment as shown in fig above.

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

From above equation (i) becomes,

$\therefore b = y - \dfrac{y_2 - y_1}{x_2 - x_1} x$ ..... (ii)

For any point (x_k, y_k) equation (i) becomes,

$y_k = mx_k + b$ ..... (iii)

For any point (x_{k+1}, y_{k+1}) equation (i) becomes,

$y_{k+1} = mx_{k+1} + b$ ..... (iv)

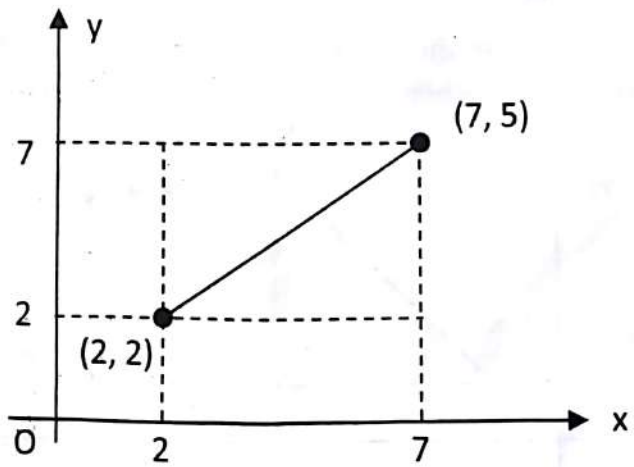
Subtracting equation (iii) from (iv) we get,

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

So, $\Delta y = m.\Delta x$, where Δx and Δy are x and y-increment respectively.

$m = \Delta y / \Delta x$ ..... (vi)

Example 3.1: Let's consider the following example:



First workout m and b as,

$$m = \frac{5-2}{7-2} = \frac{3}{5}$$

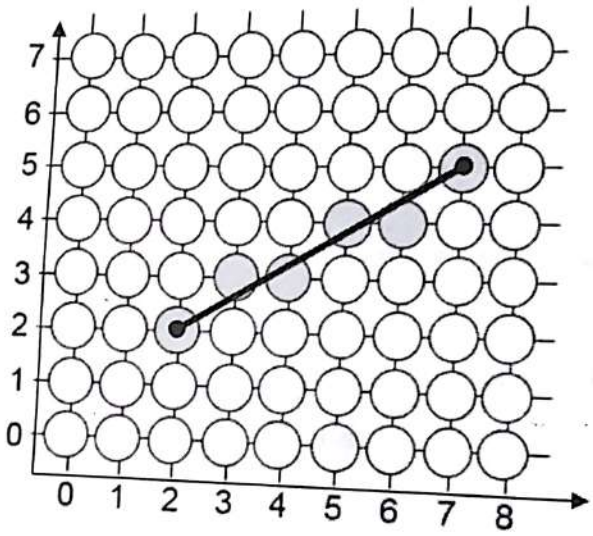
$$b = 2 - \frac{3}{5} \times 2 = \frac{4}{5}$$

Now for each x-values workout the y values,

$$y(3) = \frac{3}{5} \times 3 + \frac{4}{5} = \frac{13}{5} \approx 3 \qquad y(4) = \frac{3}{5} \times 4 + \frac{4}{5} = \frac{16}{5} \approx 3$$

$$y(5) = \frac{3}{5} \times 5 + \frac{4}{5} = \frac{19}{5} \approx 4 \qquad y(6) = \frac{3}{5} \times 6 + \frac{4}{5} = \frac{22}{5} \approx 4$$

Thus we have co-ordinates: (2, 2), (3, 3), (4, 3), (5, 4), (6, 4), (7, 5)

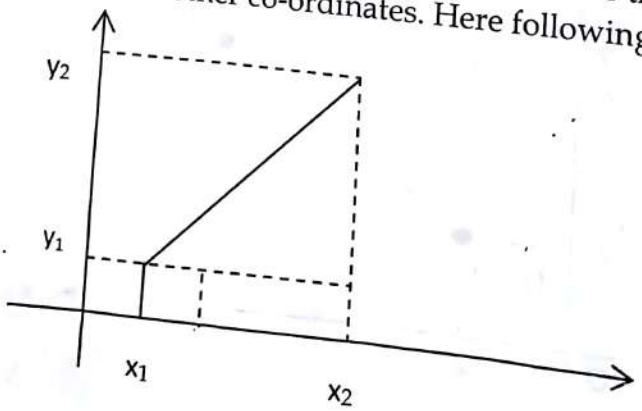


DDA line drawing Algorithm

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

$$m = \Delta y / \Delta x$$

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



$|m| > 1$

Case 1: If m is positive (m>0) and | m | <=1,
Let (x_k, y_k) be the plotted pixel then next pixels going to plot be,

$$x_{k+1} = x_k + \Delta x$$

$$y_{k+1} = y_k + \Delta y$$ Where, Δx and Δy are small incremental distance.

We have $m = \frac{\Delta y}{\Delta x}$

$\Delta y = m\Delta x$.....(1)

Also here | m | <=1, sampling be done along x-axis with Δx=1.
Then,

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

$$y_{k+1} = y_k + m$$

Here *k* takes value from starting point and increase by 1 until final end point. *m* can be any real value between 0 and 1.

Case 2: If m is positive (m>0) and | m | >=1,
Let (x_k, y_k) be the plotted pixel then next pixels going to plot be,

$$x_{k+1} = x_k + \Delta x$$

$$y_{k+1} = y_k + \Delta y$$

Where Δx and Δy are small incremental distance

We have $m = \frac{\Delta y}{\Delta x}$

$\Delta y = m\Delta x$.....(1)

Also here | m | > =1, sampling be done along y-axis with Δy=1.
Then, Δx=1/m

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

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

The above equations are under the assumption that the lines are processed from left to right. I.e. left end point is starting.

If the processing is from right to left
Case 3: if | m | <=1
Let (xk, yk) be the plotted pixel then next pixels going to plot be,

$$x_{k+1} = x_k + \Delta x$$

$$y_{k+1} = y_k + \Delta y$$

Where Δx and Δy are small incremental distance

We have $m = \frac{\Delta y}{\Delta x}$

$\Delta y = m\Delta x$ .....(1)

Also here | m | <=1, sampling be done along x-axis with Δx=-1.
Then,

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

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

Case 4: If $|m|>1$

Let (x_k, y_k) be the plotted pixel then next pixels going to plot be,

$$x_{k+1} = x_k + \Delta x$$

$$y_{k+1} = y_k + \Delta y$$

Where Δx and Δy are small incremental distance

We have $m = \frac{\Delta y}{\Delta x}$

$\Delta y = m\Delta x$ .....(1)

Also here $|m|>=1$, sampling be done along y-axis with $\Delta y=-1$.

Then,

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

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

Therefore, in general,

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

For $|m| < 1$ and

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

For $|m| >1$

Algorithm for DDA line drawing Algorithm

Step 1: Start

Step 2: Input the line endpoints and store the left endpoint in (x_1, y_1) and right end point in (x_2, y_2).

Step 3: Calculate the values of dx and dy,

$dx = x_2 - x_1$

$dy = y_2 - y_1$.

Step 4: if(abs(dx) > abs(dy)) then

steplength = abs(dx)

else

steplength= abs(dy)

Step 5: Calculate the values of x-increment and y-increment as,

xIncrement = dx/ steplength

yIncrment = dy/ steplength

Step 6: set $x=x_1$ and $y=y_1$;

Step 7: draw the pixel (round(x), round(y));

Step 8: for k=0 to steplength do

x = x + xIncrement

y = y + yIncrement

Draw the pixel (round(x), round(y));

Step 9: End

Advantages of DDA line drawing algorithm

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

Disadvantages of DDA line drawing algorithm

- Floating point arithmetic in DDA algorithm is still time consuming.
- The algorithm is orientation dependent. Hence end point accuracy is poor.
- Although DDA is fast, the accumulation of round-off error in successive additions of floating point increment however can cause the calculation pixel position to drift away from the true line path for long line segment.
- Rounding-off in DDA is time consuming.

Example 3.2: Digitize the line with endpoints (1, 5) and (7, 2) using DDA algorithm

Solution:

Here, dx = 7-1=6 and dy = 2-5 = -3

So, steplength = 6 (since abs (dx)>abs (dy))

Therefore, xIncrement = dx/steplength

= 6/6 = 1

yIncrement = dy/steplength

= -3/6 = -1/2 = -0.5

Based on these values the intermediate pixel calculation is shown in the table below.

k	x_{k+1}	y_{k+1}	(x_{k+1}, y_{k+1})	Plot in screen (x_{k+1}, y_{k+1})
1	2	4.5	(2, 4.5)	(2, 5)
2	3	4	(3, 4)	(3, 4)
3	4	3.5	(4, 3.5)	(4,4)
4	5	3	(5,3)	(5,3)
5	6	2.5	(6,2.5)	(6,3)
6	7	2	(7,2)	(7,2)

Lab problem 3.1: First program to draw a line by using build in function line().

```
#include<stdio.h>
#include<graphics.h>//must be included for every graphics program
#include<conio.h>
#include<dos.h> //for including delay function
void main()
{
    int gd=DETECT, gm;
    //gd=detects best available graphics driver, gm =graphics mode
    initgraph(&gd, &gm, "C:\\TurboC3\\BGI");// for initializing graph mode
```

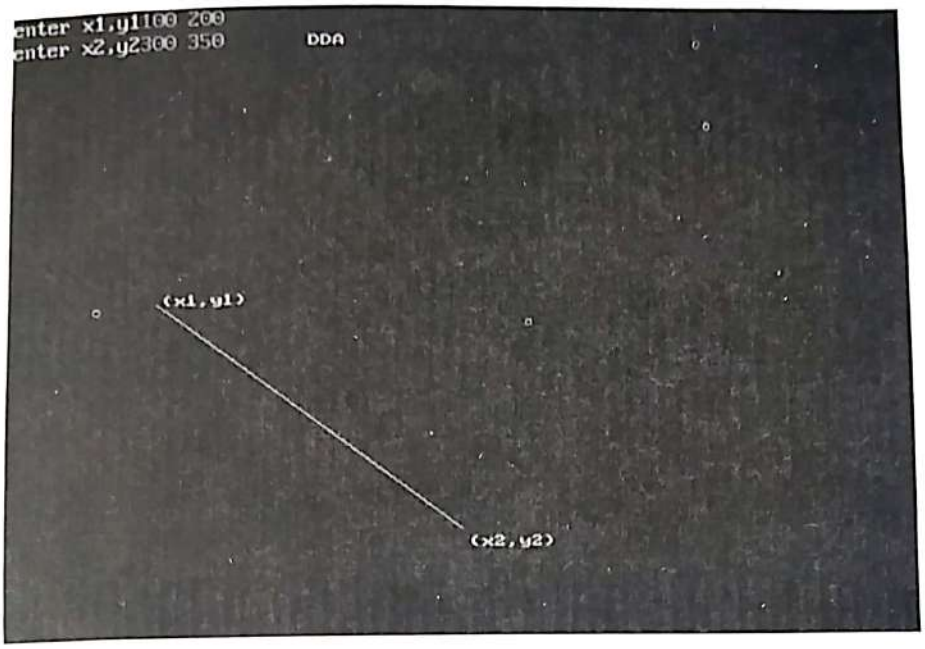
```
        /*above 2 steps are must for every graphics program. Declaration of any variables mu
        be done before calling initgraph() function. */
        line(100,100,200,200);//draws a line segment.
        getch();
}
```

Lab problem 3.2: To implement DDA algorithm for drawing a line segment between two give end points A(x1, x2) and B(x2, y2).

```
#include<stdio.h>
#include<graphics.h>
#include<math.h>
float round(float a);
void main()
{
        int gd=DETECT,gm;
        int x1,y1,x2,y2,steps,k;
        float xincr,yincr,x,y,dx,dy;
        printf("enter x1,y1");
        scanf("%d%d",&x1,&y1);
        printf("enter x2,y2");
        scanf("%d%d",&x2,&y2);
        initgraph(&gd,&gm,"c:\\turboc3\\BGI");
        dy=y2-y1;
        if(abs(dx)>abs(dy))
                steps=abs(dx);
        else
                steps=abs(dy);
        xincr=dx/steps;
        yincr=dy/steps;
        x=x1;
        y=y1;
        for(k=1;k<=steps;k++)
        {
                delay(100);
                y+=yincr;
                putpixel(round(x),round(y),WHITE);
        }
        outtextxy(200,20,"DDA");
        outtextxy(x1+5,y1-5,"(x1,y1)");
        outtextxy(x2+5,y2+5,"(x2,y2)");
        getch();
        closegraph();
}
float round(float a)
{
```

```
    int b=a+0.5;
    return b;
}
```

Output



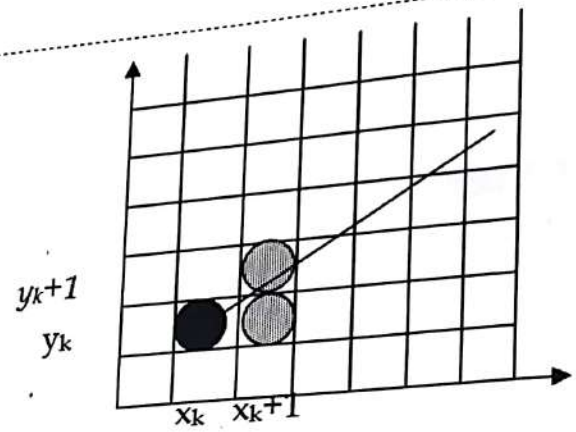
Bresenham Line Drawing Algorithm (BSA)

The Bresenham algorithm is another incremental scan conversion algorithm. The big advantage of this algorithm is that, it uses only integer calculations. Moving across the x axis in unit intervals and at each step choose between two different y coordinates. For example, as shown in the following illustration, from position (2, 3) you need to choose between (3, 3) and (3, 4). You would like the point that is closer to the original line.DDA algorithm follow the floating point calculation such that calculated pixels can be drift from actual line path due to accumulation of floating point error. Hence an integer calculation be made by using a parameter called decision parameter (P-value) that is use to decide the pixel position for next plot and such a method is called Bresenham line drawing algorithm.

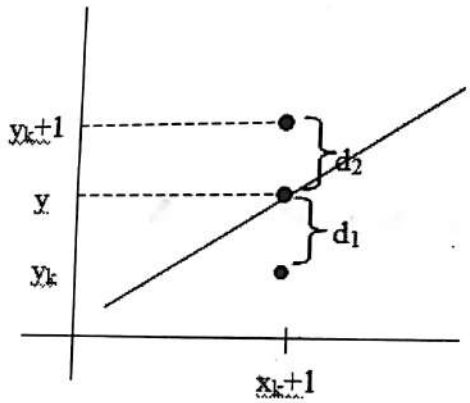
Derivation of P- value for positive slope less than 1 (0<m<1)

Since m<1 hence pixel position along the line path are determined by sampling at unit x intervals. Starting from left end point, we step to each successive column and plot the pixel closest to line path.

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



At sampling position x_k+1, we label vertical pixel separation from line path as d_1& d_2 as in figure below.



The y-coordinate on the mathematical line path at pixel column x_k+1 is

$$y = m(x_k+1)+b$$

Then $d_1 = y - y_k$

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

$$= \frac{\Delta y}{\Delta x}(xk+1)+b-yk$$

Also, $d_2 = (y_k+1) - y$

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

Now, $d_1-d_2 = \frac{\Delta y}{\Delta x}(x_k+1)+b - y_k - [(y_k+1) - \frac{\Delta y}{\Delta x}(x_k+1)-b]$

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

Or, $\Delta x(d_1-d_2) = 2\Delta y x_k + 2\Delta y - \Delta x y_k - \Delta x - \Delta x y_k + 2b\,\Delta x$

$$= 2\Delta y.x_k - 2\Delta x.y_k + 2\Delta y + 2b\Delta x - \Delta x$$

$$= 2\Delta y.x_k - 2\Delta x.y_k + c$$

Where $c = 2\Delta y + 2b\Delta x - \Delta x$ is a constant

Let's say Δx be always positive then we can introduce a parameter for decision parameter p_k for the kth step as,

$$p_k = \Delta x(d_1 - d_2)$$

$$\Rightarrow \quad p_k = 2\Delta y.x_k - 2\Delta x.y_k + c$$

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

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

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

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

Since sapling is done along x-axis

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

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

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

If $p_k \le 0$ then $y_{k+1} = y_k$

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

or, $p_{k+1} = p_k + 2\Delta y$

If $p_k > 0$ then $y_{k+1} = y_k + 1$

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

$$\Rightarrow \quad p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

Derivation for initial decision parameter

We know, $p_k = 2\Delta y.x_k - 2\Delta x.y_k + c$

$$= 2\Delta y.x_k - 2\Delta x.y_k + 2\Delta y + 2b\Delta x - \Delta x$$

When k=0,

$$p_o = 2\Delta y.x_0 - 2\Delta x.y_0 + 2\Delta y + 2b\Delta x - \Delta x$$

Since y=mx+b

$y_0 = mx_0 + b$

Or, $y_0 = \frac{\Delta y}{\Delta x} x_0 + b$

So $$p_o = 2\Delta y.x_0 - 2\Delta x\left(\frac{\Delta y\, x_0 + \Delta x b}{\Delta x}\right) + 2\Delta y + 2b\Delta x - \Delta x$$

⇨ $p_o = 2\Delta y - \Delta x$

Advantages of Bresenham Line Drawing Algorithm

- Involves simple integer calculations, so does not need to perform round off operation.
- Bresenham line algorithm is a highly efficient incremental method over DDA
- It can be used to generate circles and other curves.
- It produces mathematically accurate results using only integer addition, subtraction and multiplication by 2, which can be accomplished by a simple arithmetic shift operation.

Disadvantages of Bresenham Line Drawing Algorithm

- Additional parameter i.e., decision parameter must be calculated at each step

Algorithm of Bresenham Line Drawing Algorithm

Step 1: Start

Step 2: Input the two line endpoint and store the left endpoint at (x_o, y_o) and right endpoint at (x_1, y_1)

Step 3: calculate Δx=abs(x_1-x_0), Δy=abs (y_1-y_0)

Step 4: calculate initial decision parameter as,

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

```
Step 4: if x1 > x0 then
            Set x=x0
            Set y=y0
            Set xend=x1;
        Otherwise
            Set x=x1
            Set y=y1
            Set xend=x0;
Step 5: draw pixel at(x, y)
Step 6: while(x<xend)
        x ++;
        if p<0 then
            p=p+2Δy
        Otherwise
            y++
            p=p+2Δy-2Δx
        Draw pixel at(x, y)
Step 7: Stop
```

Example 1: Digitize a line with end points (10, 15) and (15, 18) using BSA.

Solution: Here, $\triangle y$ = |18-15| = 3

$\triangle x$ = |15-10| = 5

Since, $|m| = \triangle y/\triangle x \leq 1$, so we sample at x direction i.e., at each step we simply increment x-coordinate by 1 and find appropriate y-coordinate.

First pixel to plot is (10, 15), which is the starting endpoint.

Now, initial decision parameter, $p_0 = 2\Delta y - \Delta x = 2 * 3 - 5 = 1$

We know that, If $p_k < 0$, then we need to set $p_{k+1} = p_k + 2\Delta y$ and plot pixel (x_k+1, y_k)

And if $pk \geq 0$, then we need to set $pk+1 = pk + 2\triangle y - 2\triangle x$ then plot pixel (xk+1, yk+1)

Here, $p_0 = 1$, we have i.e., $p_k \geq 0$

So, $p1 = p0 + 2\triangle y - 2\triangle x = 1 + 2 * 3 - 2 \times 5 = -3$

$p2 = p1 + 2\triangle y = -3 + 2 * 3 = 3.$

$p3 = p2 + 2\triangle y - 2\triangle x = 3 + 6 - 10 = -1.$

$p4 = p3 + 2\triangle y = -1 + 6 = 5$

Successive pixel positions and decision parameter calculation is shown in the table below.

k	x	Y	p_k	(x_{k+1}, y_{k+1})
0	10	15	1(>=0)	(11, 16)
1	11	16	-3(<0)	(12, 16)
2	12	16	3	(13, 17)
3	13	17	-1	(14, 17)
4	14	17	5	(15, 18)

Example 2: Go through the steps of the Bresenham line drawing algorithm for a line going from (20,10) to (30,18).

Solution

Here, $\triangle y = |18-10| = 8$

$\triangle x = |30 - 20| = 10$

Since, $|m| = \triangle y/\triangle x \leq 1$, so we sample at x direction i.e., at each step we simply increment x-coordinate by 1 and find appropriate y-coordinate.

First pixel to plot is (20, 10), which is the starting endpoint.

Now, initial decision parameter, $p0 = 2\triangle y - \triangle x = 2 * 8 - 10 = 6$

We know that, If $pk < 0$, then we need to set $pk+1 = pk + 2\triangle y$ and plot pixel (xk+1, yk)

And if $pk \geq 0$, then we need to set $pk+1 = pk + 2\triangle y - 2\triangle x$ then plot pixel (xk+1, yk+1)

Here, $p0 = 6$, we have i.e., $pk \geq 0$

So, $p1 = p0 + 2\triangle y - 2\triangle x = 6 + 2 * 8 - 2 * 10 = 2$

$p2 = p1 + 2\triangle y - 2\triangle x = 2 + 2*8 - 2 * 10 = -2.$

$p3 = p2 + 2\triangle y = -2 + 2*8 = 14.$

$p4 = p3 + 2\triangle y - 2\triangle x = 14 + 2*8 - 2*10 = 10$

$p5 = p4 + 2\triangle y - 2\triangle x = 10 + 2*8 - 2*10 = 6$

$p6 = p5 + 2\triangle y - 2\triangle x = 6 + 2*8 - 2*10 = 2$

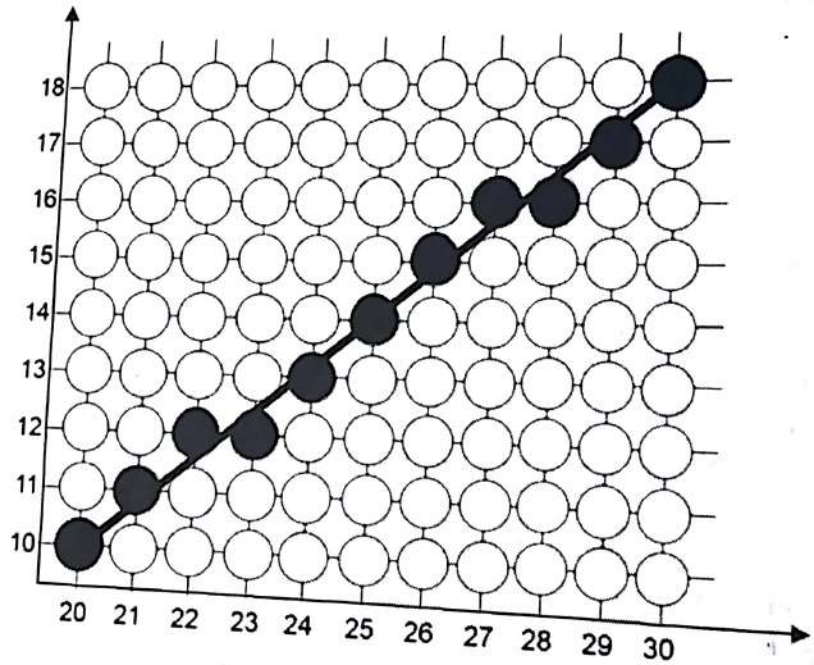
$p7 = p6 + 2\triangle y - 2\triangle x = 2 + 2*8 - 2*10 = -2$

$p8 = p7 + 2\triangle y = -2 + 2*8 = 14$

$p9 = p8 + 2\triangle y - 2\triangle x = 14 + 2*8 - 2*10 = 10$

Successive pixel positions and decision parameter calculation is shown in the table below.

k	x	y	p_k	(x_{k+1}, y_{k+1})
0	20	10	6(>=0)	(21, 11)
1	21	11	2(>=0)	(22, 12)
2	22	12	-2(<0)	(23, 12)
3	23	12	14(>=0)	(24, 13)
4	24	13	10(>=0)	(25, 14)
5	25	14	6(>=0)	(26, 15)
6	26	15	2(>=0)	(27, 16)
7	27	16	-2(<0)	(28, 16)
8	28	16	14(>=0)	(29, 17)
9	29	17	10(>=0)	(30, 18)



Difference between DDA line drawing algorithm and Bresenham line drawing algorithm

	DDA Line Drawing Algorithm	Bresenham Line Drawing Algorithm
Arithmetic	DDA algorithm uses floating points i.e. Real Arithmetic.	Bresenham algorithm uses fixed points i.e. Integer Arithmetic.
Operations	DDA algorithm uses multiplication and division in its operations.	Bresenham algorithm uses only subtraction and addition in its operations.
Speed	DDA algorithm is rather slowly than Bresenham algorithm in line drawing because it uses real arithmetic (floating-point operations).	Bresenham algorithm is faster than DDA algorithm in line drawing because it performs only addition and subtraction in its calculation and uses only integer arithmetic so it runs significantly faster.

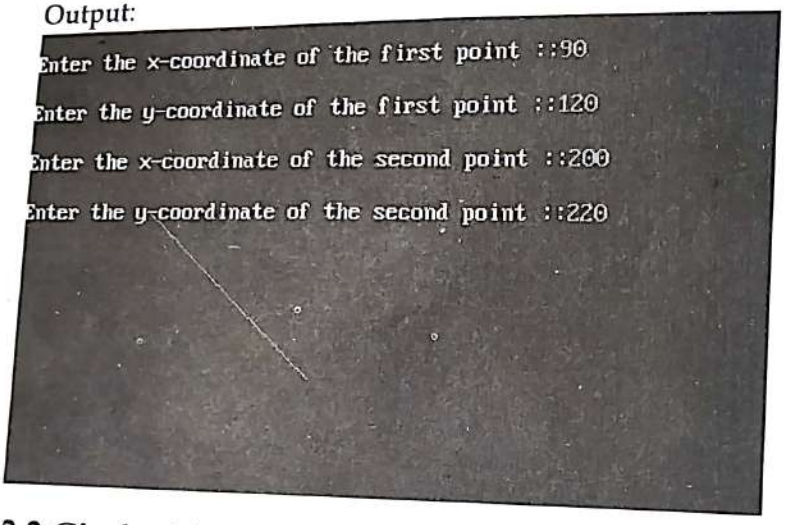
Accuracy & Efficiency	DDA algorithm is not as accurate and efficient as Bresenham algorithm.	Bresenham algorithm is more efficient and much accurate than DDA algorithm.
Drawing	DDA algorithm can draw circles and curves but that are not as accurate as Bresenham algorithm.	Bresenham algorithm can draw circles and curves with much more accuracy than DDA algorithm.
Round Off	DDA algorithm round off the coordinates to integer that is nearest to the line.	Bresenham algorithm does not round off but takes the incremental value in its operation.
Expensive	DDA algorithm uses an enormous number of floating-point multiplications so it is expensive.	Bresenham algorithm is less expensive than DDA algorithm as it uses only addition and subtraction.

Lab problem 3.3: To implement Bresenham's line drawing algorithm for drawing a line segment between two given endpoints A(x1, y2) and B(x2, y2).

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void main()
{
    int x,y,x1,y1,x2,y2,p,dx,dy;
    int gd=DETECT,gm;
    initgraph(&gd, &gm, "C:\\TurboC3\\BGI");
    printf("\nEnter the x-coordinate of the first point ::");
    scanf("%d",&x1);
    printf("\n Enter the y-coordinate of the first point ::");
    scanf("%d",&y1);
    printf("\n Enter the x-coordinate of the second point ::");
    scanf("%d",&x2);
    printf("\n Enter the y-coordinate of the second point ::");
    scanf("%d", &y2);
    x=x1;
    y=y1;
    dx=x2-x1;
    dy=y2-y1;
    putpixel(x,y,2);
    p=(2*dy-dx);
    while(x<=x2)
    {
      if(p<0)
      {
          x=x+1;
```

```
            p=p+2*dy;
        }
        else
        {
                x=x+1;
                y=y+1;
                p=p+(2*dy)-(2*dx);
        }
        putpixel(x,y,7);
    }
    getch();
    closegraph();
}
```

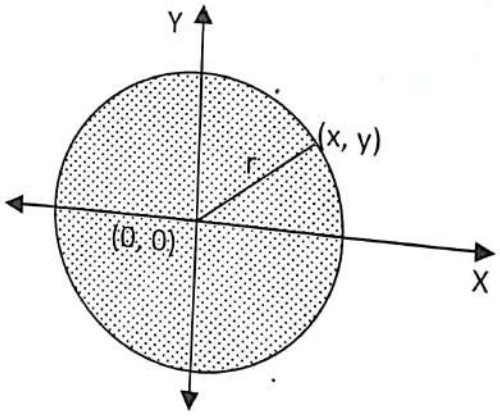
Output:



3.2 Circle Algorithm

A circle is defined as a set of points that are all at a given distance r from the center position (x_c, y_c). The general circle equation can be written as;

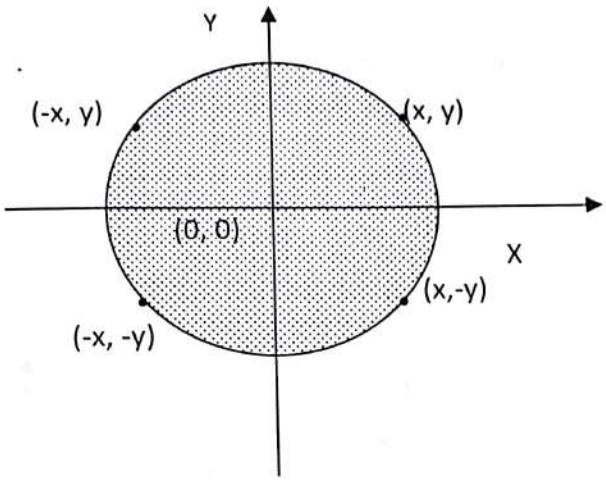
$$(x - x_c)^2 + (y - y_c)^2 = r^2.$$



Properties of circle

Symmetry in quadrants

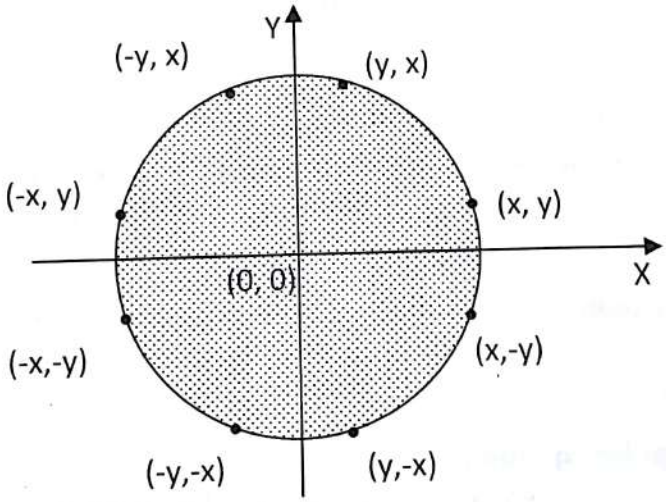
The shape of the circle is similar in each quadrant. Thus by calculating points in one quadrant we can calculate points in other three quadrants.



4-way symmetry

Symmetry in octants

The shape of the circle is similar in each octant. Thus by calculating points in one octant we can calculate points in other seven octants. If the point (x, y) is on the circle, then we can trivially compute seven other points on the circle. Therefore, we need to compute only one 45° segment to determine the circle, as shown in figure below. By taking advantage of circle symmetry in octants, we can generate all pixel positions around a circle by calculating only the points within the sector from x = 0 to y = x.



8-way symmetry

Methods to draw circle in computer graphics

A circle is defined as a set of points that are all the given distance (x_c, y_c). This distance relationship is expressed by the Pythagorean Theorem in Cartesian coordinates as,

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

There are three popular methods for drawing circle in computer graphics,

- Simple Direct method
- Trigonometric method (Polar form)

$r_x = 8$

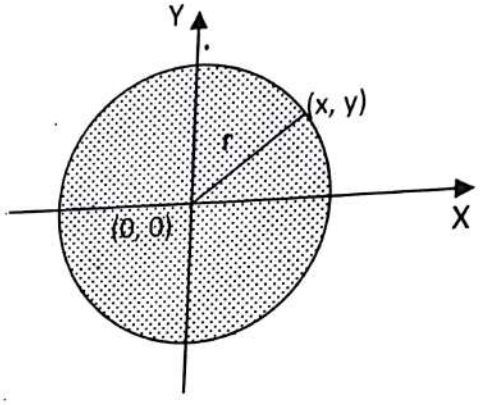
$r_y = 6$

- Mid-point Circle method

Simple Direct method for drawing circle

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

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



Increment x in unit steps and determine corresponding value of y from the equation above. Then set pixel at position *(x, y)*. The steps are taken from *–r* to *+r*.

In computer graphics, we take origin at upper left corner point on the display screen i.e. first pixel of the screen so any visible circle drawn would be centered at point other than (0,0).

If center of circle is (x_c, y_c) then the calculated point from origin center should be moved to pixel position by ($x+x_c$, $y+y_c$).In general the equation of circle centered at (x_c, y_c) and radius r is,

$$(x - xc)^2 + (y - yc)^2 = r^2$$

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

Use this equation to calculate the position of points on the circle. Take unit step fromx_c-r to x_c+r for x value and calculate the corresponding value of y- position for pixel position (x, y).

Drawbacks

- Due to floating point calculation actual pixel position may be away from circular path.
- Due to presence of square root calculation, algorithm requires more computational time that means process becomes slow.

Drawing circle using polar equations

In this method we use general equation of circle in polar co-ordinate as below.If (x, y) be any point on the circle boundary with center (0, 0) and radius r, then

$x = r\cos\theta$

$y = r\sin\theta$

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

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

$x = xc + r\cos\theta$

$y = yc + r \sin \theta$

Drawback

- Follow the floating point calculation.

Midpoint circle Algorithm

In midpoint circle algorithm, we sample at unit intervals and determine the closest pixel position to the specified circle path at each step. For a given radius r, and screen center position (x_c, y_c) we can set up our algorithm to calculate pixel positions around a circle path centered at (0, 0) and then each calculated pixel position (x, y) is moved to its proper position by adding x_c to x and y_c to y.

$x = x + x_c,$

$y = y + y_c.$

It uses the symmetric property of circle to reduce computational time. Pixel positions are only calculated within an octant and transfer them through remaining seven octant to get a complete circle.

Here a decision parameter is used to decide the pixel for next plot.

Derivation of p-value for mid-point circle algorithm

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

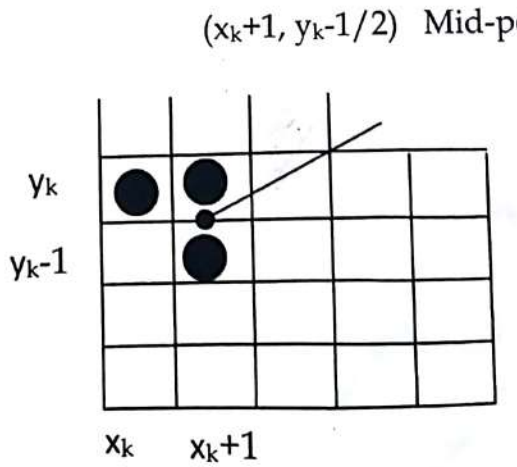
$$f_{circle} = x^2 + y^2 - r^2$$

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

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

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

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



Our decision parameter is circle function evaluated at the mid-point

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Calculating initial decision parameter

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

When k = 0;

$$p_0 = (x_0 + 1)^2 + y_0{}^2 - y_0 + \frac{1}{4} - r^2$$

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

Hence $p_0 = (0+1)^2 + r^2 - r + \frac{1}{4} - r^2$

$$= 1 - r + \frac{1}{4}$$

$$= \frac{5}{4} - r$$

If p_0 is specified in integer,

$$p_0 = 1 - r.$$

Algorithm for Midpoint circle drawing

1. Start
2. Input radius r and circle Centre (h, k)
3. Initialize x=0, y=r and $p_0 = 1 - r$
4. While(x≤y)
   //plot 8 points as
   Plot pixel(x+h, y+k);
   Plot pixel (-x+h, y+k);
   Plot pixel(x+h, -y+k);
   Plot pixel (-x+h, -y+k);
   Plot pixel(y+h, x+k);
   Plot pixel (-y+h, x+k);
   Plot pixel(y+h, -x+k);
   Plot pixel (-y+h, -x+k);
5. If p_0<0 then
   Set, $p_0 = (p_0 + 2x + 1)$
   Set, x=x+1
   Otherwise,
   Set, $p_0 = p_0 + 2x + 1 - 2y$
   Set y=y-1
6. End while loop
7. Stop

Example 1: Calculate the points to draw a circle having radius 10 and center at (0, 0)

Solution: we have r=10
Centre (h, k) = (0, 0)
Set x=0, y=r=10

Now the first pixel to be drawn= (0, 10)

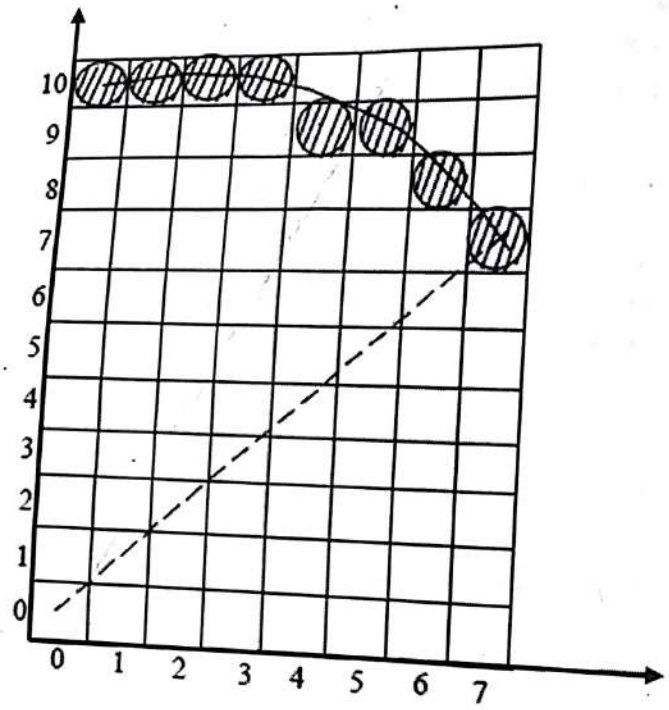
Initial decision parameter (p) = 1-r = 1 – 10 = -9

Now points on the octant are given by,

K	P_k	(x_{k+1}, y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
1	-9	(1, 10)	2	20
2	-6	(2, 10)	4	20
3	-1	(3, 10)	6	20
4	6	(4, 9)	8	18
5	-3	(5, 9)	10	18
6.	8	(6, 8)	12	16
7	5	(7, 7)	14	14

Now using 8 point symmetry we get all the points on the circle as below:

(x, y)	(0, 10)	(1, 10)	(2, 10)	(3,10)	(4, 9)	(5, 9)	(6,8)	(7,7)
(-x, y)	(0, 10)	(-1, 10)	(-2, 10)	(-3,10)	(-4, 9)	(-5, 9)	(-6,8)	(-7,7)
(x, -y)	(0, -10)	(1, -10)	(2,-10)	(3,-10)	(4, -9)	(5, -9)	(6,-8)	(7,-7)
(-x, -y)	(0, -10)	(-1, -10)	(-2, -10)	(-3,-10)	(-4, -9)	(-5,-9)	(-6,-8)	(-7,-7)
(y, x)	(10, 0)	(10, 1)	(10, 2)	(10, 3)	(9, 4)	(9, 5)	(8, 6)	(7,7)
(-y, x)	(10, 0)	(-10, 1)	(-10, 2)	(-10, 3)	(-9, 4)	(-9, 5)	(-8, 6)	(-7,7)
(y, -x)	(10, 0)	(10, -1)	(10, -2)	(10, -3)	(9, -4)	(9,-5)	(8, -6)	(7,-7)
(-y, -x)	(10, 0)	(-10, -1)	(-10, -2)	(-10, -3)	(-9, -4)	(-9,-5)	(-8,-6)	(-7,-7)



Example 2: Digitize a circle $(x-2)^2 + (y-3)^2 = 25$

Solution:

Step 0: We have center C(h, k) = (2, 3)

Radius (r) = 5

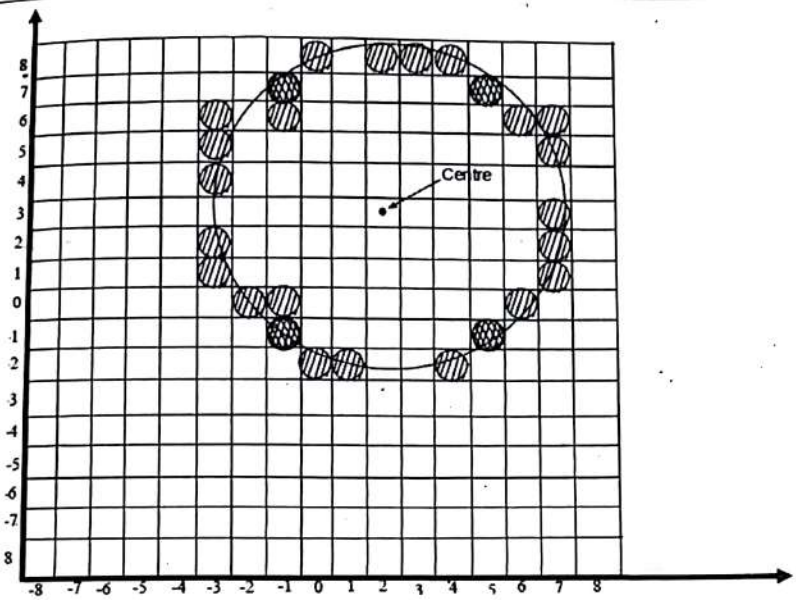
First pixel= (0, 5)

Initial decision parameter (Po) = 1-r= 1-5 = -4

Now points on the octant are given by,

k	P_k	(x_{k+1}, y_{k+1})	$2x_{k+1}$	$2y_{k+1}$	Actual pixel (x_{k+1}, y_{k+1})
1	-4	(1, 5)	2	10	(3, 8)
2	-1	(2, 5)	4	10	(4, 8)
3	4	(3, 4)	6	8	(5, 7)
4	3	(4, 3)	8	6	(6, 6)
5	6	(5, 2)	10	4	(7, 5)
6	13	(6, 1)	12	2	(8, 4)
7	24	(7, 0)	14	0	(9, 3)

8	39	(8, -1)	16	-2	(10, 2)
9	.....				

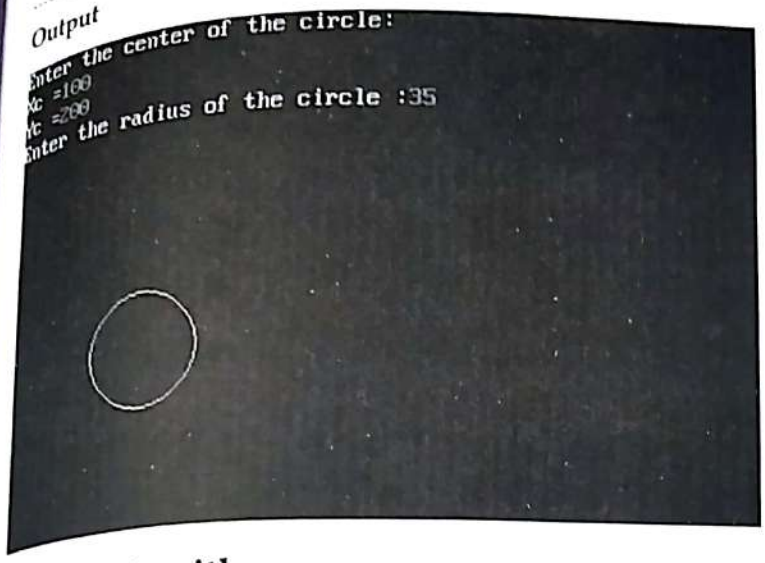


Lab problem 3.4: To implement midpoint circle generation algorithm or bresenham's circle algorithm for drawing a circle of given center (x, y) and radius r.

```
#include<dos.h>
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
voiddraw_circle(int, int, int);
void symmetry(int, int, int, int);
void main()
{
        intxc,yc,R;
        intgd=DETECT,gm;
        initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
        printf("Enter the center of the circle:\n");
        printf("Xc =");
        scanf("%d",&xc);
        printf("Yc =");
        scanf("%d", &yc);
        printf("Enter the radius of the circle :");
        scanf("%d",&R);
        draw_circle(xc,yc,R);
        getch();
        closegraph();
}

voiddraw_circle(int  xc, int yc, int rad)
```

```
{
        int x = 0;
        int y = rad;
        int p = 1-rad;
        symmetry(x,y,xc,yc);
        for(x=0; y>x; x++)
        {
                if(p<0)
                        p += 2*x + 3;
                else
                {
                        p += 2*(x-y) + 5;
                        y--;
                }
                symmetry(x,y,xc,yc);
                delay(50);
        }
}
void symmetry(int x, int y, int xc, int yc)
{
        putpixel(xc+x,yc-y,GREEN);
        delay(50);
        putpixel(xc+y,yc-x, GREEN);
        delay(50);
        putpixel(xc+y,yc+x, GREEN);
        delay(50);
        putpixel(xc+x,yc+y, GREEN);
        delay(50);
        putpixel(xc-x,yc+y, GREEN);
        delay(50);
        putpixel(xc-y,yc+x, GREEN);
        delay(50);
        putpixel(xc-y,yc-x, GREEN);
        delay(50);
        putpixel(xc-x,yc-y, GREEN);
        delay(50);
}
```

Output



Ellipse Algorithm

An ellipse is an elongated circle. Therefore elliptical curves can be generated by modifying circle drawing procedures. The ellipse, like the circle, shows symmetry. An ellipse is symmetric in quadrants. So if one quadrant is generated then other three parts can be easily generated. The basic algorithm for drawing ellipse is same as circle computing x and y position at the boundary of the ellipse from the equation of ellipse directly.

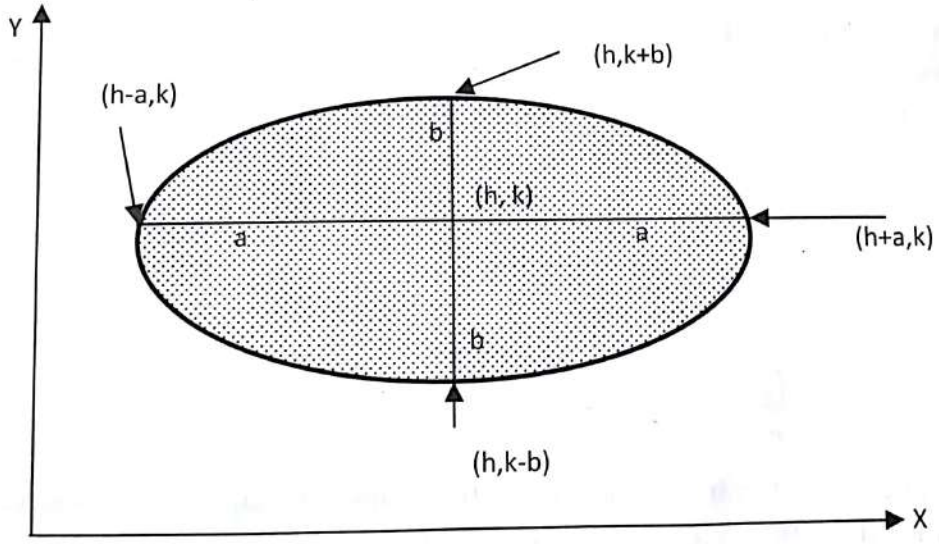


Figure: Ellipse centered at (h, k) with semi-major axis 'a' and semi-minor axis 'b'.

An ellipse can be represented as:

$$\frac{(x-h)^2}{a^2} + \frac{(x-k)^2}{b^2} = 1$$

Where, (h, k) = ellipse center

a = length of semi-major axis.

b = length of semi-minor axis.

An ellipse centered at origin (0, 0) can be represented as:

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

Ellipse generating algorithms

There are three popular methods for drawing ellipse:

- Direct Method
- Trigonometric Method
- Midpoint Ellipse Algorithm

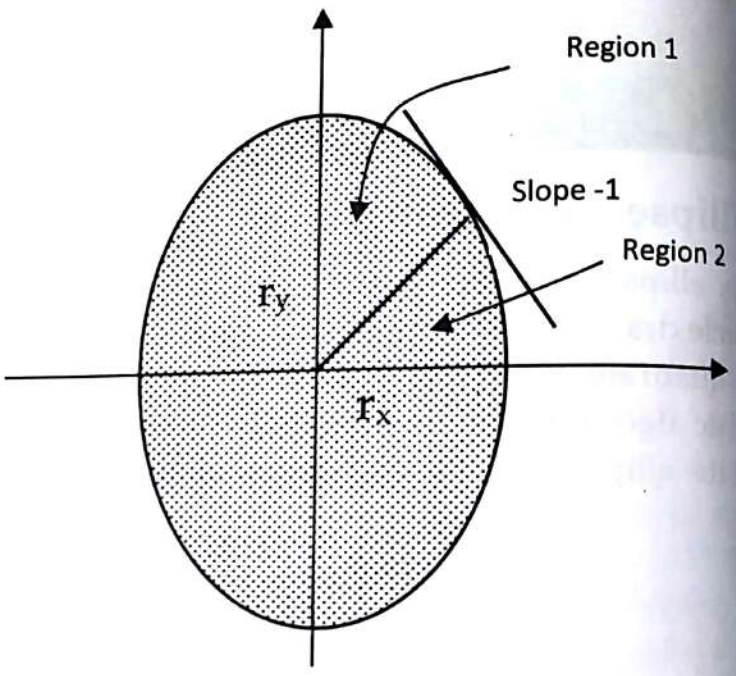
Direct Method for generating ellipse

The basic algorithm for drawing ellipse is same as circle computing x and y position at the boundary of the ellipse from the equation of ellipse directly. We have equation of ellipse centered at origin (0, 0) is,

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

This gives,

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



Stepping unit interval in x direction from $-r_x$ to r_x we can get corresponding y value at each x position which gives the ellipse boundary co-ordinates. Plotting these computed points we can get the ellipse. If center of ellipse is any arbitrary point (x_c, y_c) then the equation of ellipse can be written as,

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

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

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

1. Start
2. Input the center of ellipse (x_c, y_c) x-radius x_r and y-radius y_r.
3. For each x position starting from x_c-r and stepping unit interval along x-direction, compute corresponding y positions as

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

4. Plot the point (x ,y)

5. Repeat step 3 to 4 until $x >= x_c+x_r$
6. Stop

Computation of ellipse using polar co-ordinates

Using polar coordinates an ellipse can be represented as;

$$x = a \cos \theta + h$$
$$y = b \sin \theta + k$$

Where,(x, y) = current coordinate

a = length of semi-major axis

b = length of semi-minor axis

θ = current angle, measured in radians from 0 to 2π.

(h, k) = ellipse center.

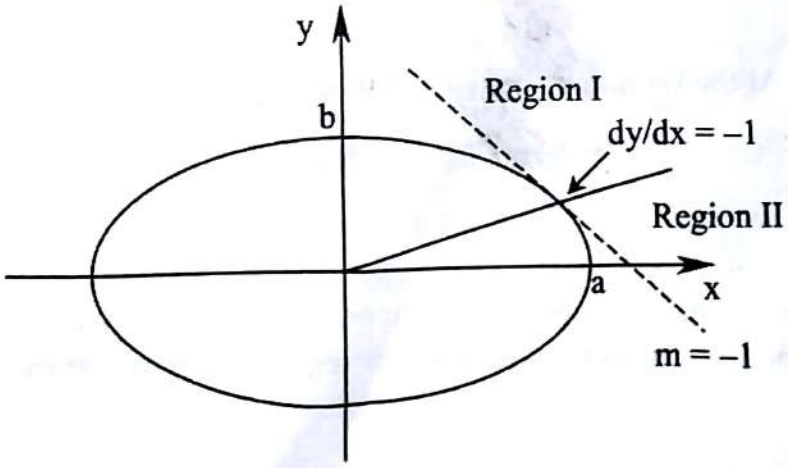
In this method, θ is incremented from 0 to 2π and we compute successive values of x and y. The algorithm based on these parametric equations on polar co-ordinates can be summarized as below,

1. Start
2. Input the center of ellipse (h, k), semi-major and semi-minor axis 'a' and 'b' respectively.
3. Starting from angle $\theta = 0$ compute boundary point of ellipse as
   $$x = a \cos\theta + h$$
   $$y = b \sin\theta + k$$
4. Plot the point at position (round(x), round(y))
5. Repeat until $\theta \geq 2\pi$ i.e. 360^0.
6. Stop

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

Mid-Point Ellipse Drawing Algorithm

Midpoint ellipse algorithm is a method for drawing ellipses in computer graphics. This method is modified from Bresenham algorithm. The advantage of this modified method is that only addition operations are required in the program loops. This leads to simple and fast implementation in all processors.



Let us consider one quarter of an

ellipse. The curve is divided into two regions. In region I, the slope on the curve is greater th
-1 while in region II less than -1.

Consider the general equation of an ellipse,

$b^2x^2 + a^2y^2 - a^2b^2 = 0$ where 'a' is the horizontal radius and b is the vertical radius, w
can define a function f(x, y) by which the error due to a prediction coordinate (x, y) can b
obtained. The appropriate pixels can be selected according to the error so that the require
ellipse is formed. The error can be confined within half a pixel.

Set, $f(x, y) = b^2x^2 + a^2y^2 - a^2b^2$

The mid-point ellipse algorithm decides which point near the boundary (i.e. path of the ellips
is closer to the actual ellipse path described by the ellipse equation. That point is taken as ne
point. It is applied to the first quadrant in two parts as in figure. Region 1 and Region 2.W
process by taking unit steps in x-coordinates direction and finding the closest value for y fo
each x-step in region 1.In first quadrant at region 1, we start at position *(0,ry)* and incrementin
x and calculating y closer to the path along clockwise direction. When slope becomes -1 the
shift unit step in x to y and compute corresponding x closest to ellipse path at Region 2 in sam
direction.

Alternatively, we can start at position *(rx, 0)* and select point in counterclockwise order shiftin
unit steps in y to unit step in x when slope becomes greater than -1. Here, to implement mid-
point ellipse algorithm, we take start position at *(0, ry)* and step along the ellipse path i
clockwise position throughout the first quadrant.

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

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

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

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

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

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

At the boundary between region 1 and region 2,

$$\frac{dy}{dx} = -1 \text{ and } 2r_y^2 = 2r_x^2$$

Therefore, we move out of region 1 when $2r_y^2 \geq 2r_x^2$

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

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

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

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

If $P_{1k} < 0$ the midpoint lies inside boundary, so next point to plot is (x_k+1, y_k) otherwise, next point to plot will be (x_k+1, y_k-1). The successive decision parameter is computed as

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

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

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

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

Subtracting (2) - (1)

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

If $P_{1k} < 0$, $y_{k+1} = y_k$ then,

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

$dy\partial y + dx\partial x = 0$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Subtracting (4)-(3)

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

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

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

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

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

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

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

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

$$\text{Or, } p_{2k+1} = p_{2k} + r_y^2 (2x_k + 1) + r_y^2 - 2r_x^2 (y_k - 1) + r_x^2$$

$$\text{Or, } p_{2k+1} = p_{2k} + r_y^2 (2x_k + 2) - 2r_x^2 (y_k - 1) + r_x^2$$

$$\text{Or, } p_{2k+1} = p_{2k} + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_x^2 \text{ where } x_{k+1} = x_k + 1 \text{ and } y_{k+1} = y_k - 1$$

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

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

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

Now the midpoint ellipse algorithm is summarized as;

1. Input center (x_c, y_c) and r_x and r_y for the ellipse and obtain the first point as
2. $(x_0, y_0) = (0, r_y)$
3. Calculate initial decision parameter value in Region 1 as

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

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

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

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

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

   $y_{k+1} = y_k$

   Otherwise next point to plot is

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

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

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

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

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

   $y_{k+1} = y-1$;

   if $P_{2k} > 0$ then

   $x_{k+1} = x_k$

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

   Otherwise

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

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

8. Determine the symmetry points in other 3 quadrants.
9. Move each calculated point (x_k, y_k) on to the centered (x_c, y_c) ellipse path as

   $x_k = x_k + xc$;

   $y_k = y_k + yc$

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

Example 3.5: Input ellipse parameters $r_x=8$ and $r_y=6$ the midpoint ellipse algorithm by determining raster position along the ellipse path is the first quadrant. Initial values and increments for the decision parameter calculates are:

$2r_y^2 x = 0$ (with increment $2r_y^2 = 72$)

$2r_x^2 y = 2r_x^2 y = 2r_x^2 r_y$ (with increment $-2rx^2 = -128$)

For region 1 the initial point for the ellipse centered on the origin is $(x_0, y_0) = (0, 6)$ and the initial decision parameter value is:

$P1_0 = r_y^2 - r_x^2 r_y^2 + 1/4 r_x^2 = -332$

Successive midpoint decision parameter values and the pixel positions along the ellipse are listed in the following table:

k	$P1_k$	(x_{k+1}, y_{k+1})	$2r_y^2x_{k+1}$	$2r_x^2y_{k+1}$
0	-332	(1, 6)	72	768
1	-224	(2, 6)	144	768
2	-44	(3, 6)	216	768
3	208	(4, 5)	288	640
4	-108	(5, 5)	360	640
5	288	(6, 4)	432	512
6	244	(7, 3)	504	384

Move out of region 1, $2ry^2x>2rx^2y$

For a region 2, the initial point is $(x_0, y_0) = (7, 3)$ and the initial decision parameter is

$$P2_0 = f_{ellipse}(7 + 1/2, 2) = -151$$

The remaining positions along the ellipse path in the first quadrant are then calculated as,

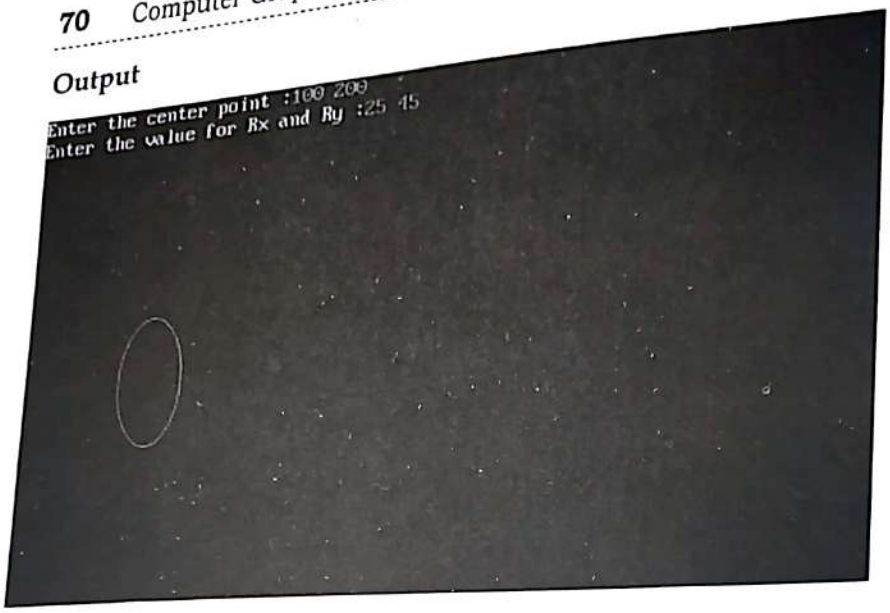
k	$P2_k$	(x_{k+1}, y_{k+1})	$2r_y^2x_{k+1}$	$2r_x^2y_{k+1}$
0	-151	(8, 2)	576	256
1	233	(8, 1)	576	128
2	745	(8, 0)	---	---

Lab problem 3.5: To implement the Ellipse Generation Algorithm for drawing an ellipse of given center(x, y) and radius r_x and r_y.

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
voiddisp();
float x, y;
int xc, yc;
void main()
{
        intgd=DETECT, gm;
        intrx,ry;
        float p1,p2;
        clrscr();
        initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
        printf("Enter the center point :");
        scanf("%d%d",&xc,&yc);
        printf("Enter the value for Rx and Ry :");
        scanf("%d%d", &rx,&ry);
        x=0;
        y=ry;
        disp();
        p1=(ry*ry)-(rx*rx*ry)+(rx*rx)/4;
        while((2.0*ry*ry*x)<=(2.0*rx*rx*y))
```

```
    {
            x++;
            if(p1<=0)
                    p1=p1+(2.0*ry*ry*x)+(ry*ry);
            else
            {
                    y--;
                    p1=p1+(2.0*ry*ry*x)-(2.0*rx*rx*y)+(ry*ry);
            }
            disp();
            x=-x;
            disp();
            x=-x;
    }
    x=rx;
    y=0;
    disp();
    p2=(rx*rx)+2.0*(ry*ry*rx)+(ry*ry)/4;
    while((2.0*ry*ry*x)>(2.0*rx*rx*y))
    {
            y++;
            if(p2>0)
                    p2=p2+(rx*rx)-(2.0*rx*rx*y);
            else
            {
                    x--;
                    p2=p2+(2.0*ry*ry*x)-(2.0*rx*rx*y)+(rx*rx);
            }
            disp();
            y=-y;
            disp();
            y=-y;
    }
    getch();
    closegraph();
}
void disp()
{
    delay(50);
    putpixel(xc+x,yc+y,10);
    putpixel(xc-x,yc+y,10);
    putpixel(xc+x,yc-y,10);
    putpixel(xc-x,yc-y,10);
}
```

Output



Area filling

In this section we are going to study different types of polygons, their representation and filling algorithms for them. A polyline is a chain of connected line segments. It is specified by giving the vertices (nodes) P_0, P_1, P_2 ... and so on. The first vertex is called the initial or starting point and the last vertex is called the final or terminal point, as shown in the Figure 3.1. When starting point and terminal point of any polyline is same, i.e. when polyline is closed then it is called polygon. This is illustrated in Figure 3.1.

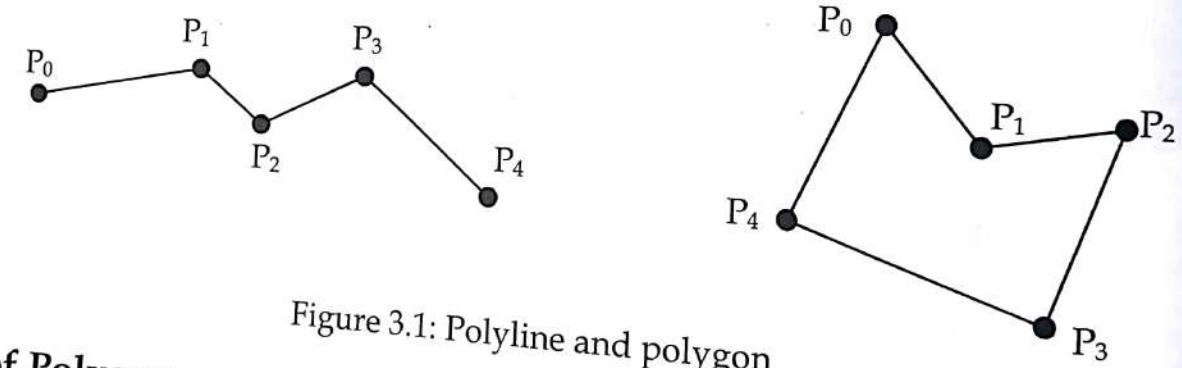


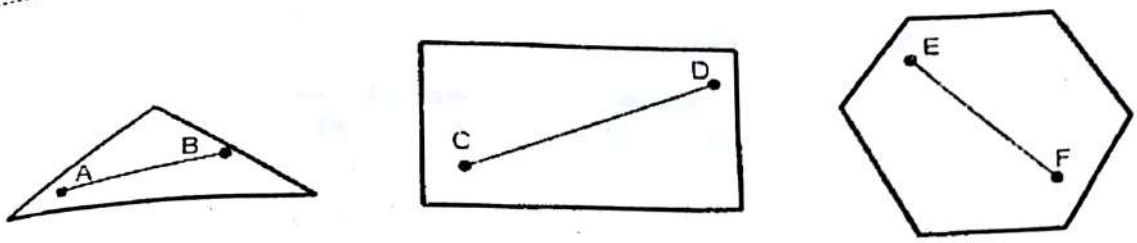
Figure 3.1: Polyline and polygon

Types of Polygons

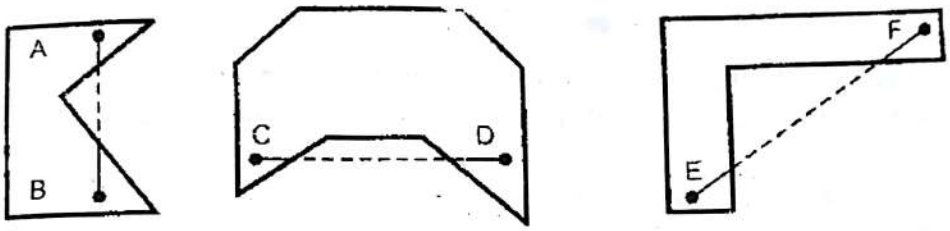
The classification of polygons is based on where the line segment joining any two points within the polygon is going to lie. There are two types of polygons:

- Convex and
- Concave

A convex polygon is a polygon in which the line segment joining any two points within the polygon lies completely inside the polygon. The figure below shows the examples convex polygons.



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



Filled Area primitives

A standard output primitive in general graphics package is solid color or patterned polygon area. Other kinds of area primitives are sometimes available, but polygons are easier to process since they have linear boundaries. There are two basic approaches to area filling in raster systems. One way to fill an area is to determine the overlap intervals for scan lines that cresses the area. Another method for area filling is to start from a given interior position and point outward from this until a specified boundary is met.

There are three popular methods for are fill algorithms:

- Scan line polygon fill algorithms
- Boundary fill algorithms and
- Flood fill algorithms

Scan Line Polygon Fill Algorithm

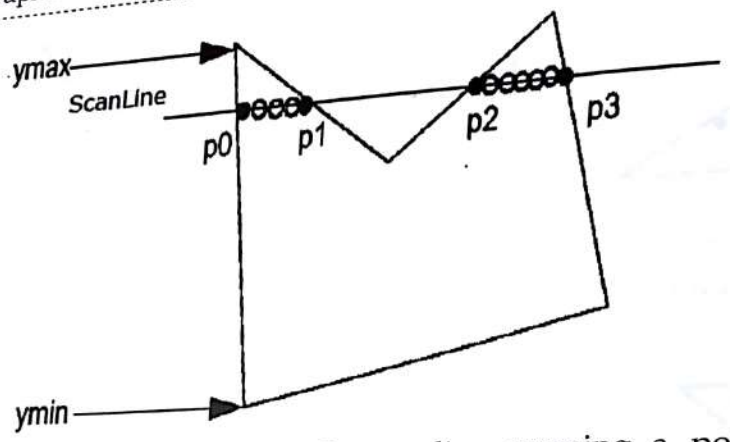
This algorithm works by intersecting scan line with polygon edges and fills the polygon between pairs of intersections. The following steps depict how this algorithm works.

Step 1 – Find out the Y_{min} and Y_{max} from the given polygon.

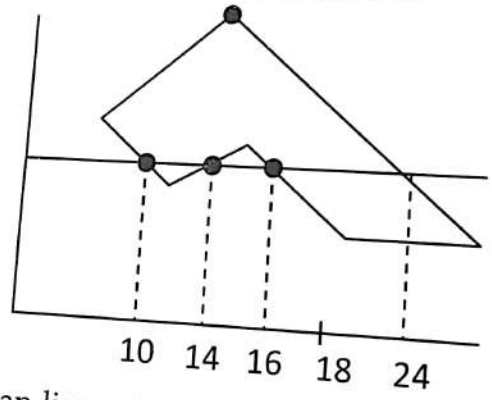
Step 2 – ScanLine intersects with each edge of the polygon from Y_{min} toY_{max}. Name each intersection point of the polygon. As per the figure shown below, they are named as p0, p1, p2, p3.

Step 3 – Sort the intersection point in the increasing order of X coordinate i.e. (p0, p1), (p1, p2), and (p2, p3).

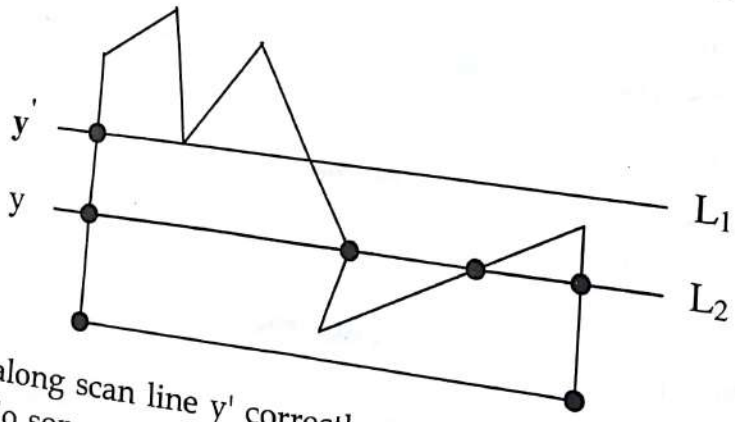
Step 4 – Fill all those pair of coordinates that are inside polygons and ignore the alternate pairs.



In scan-line polygon fill algorithm, for each scan-line crossing a polygon, it locates the intersection points of the scan line with the polygon edges. These intersection points are then sorted from left to right, and the corresponding frame-buffer positions between each intersection pair are set to the specified color. In the figure below, the four pixel intersection positions with the polygon boundaries defined two stretches of interior pixel from x=10 to x=14 and from x=16 to x=24.Some scan-line intersections at polygon vertices require extra special handling. A scan-line passing through a vertex intersects two polygon edges at that position adding two points to the list of intersection for the scan-line.

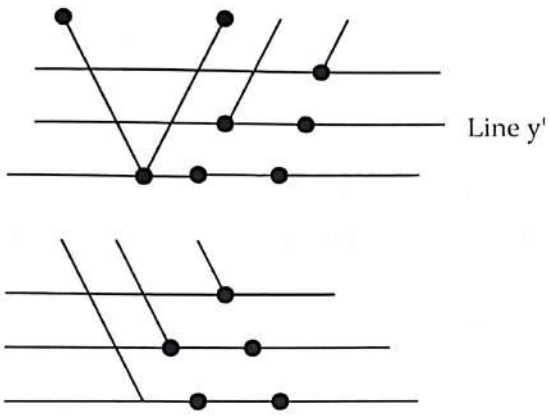


The figure below shows two scan lines at position y and y' that intersect the edge points. Scan line at y intersects five polygon edges. Scan line at y' intersects 4 (even numbers) of edges though it passes through vertex.



Intersection points along scan line y' correctly identify the interior pixel spans. But with scan line y, we need to do some additional processing to determine the correct interior points. For scan line y, the two edges sharing the intersecting vertex are on opposite side of the scan-line. But for scan-line y' the two edges sharing intersecting vertex are on the same side (above) the scan line position. So the vertices those are on opposite side of scan li
processing.

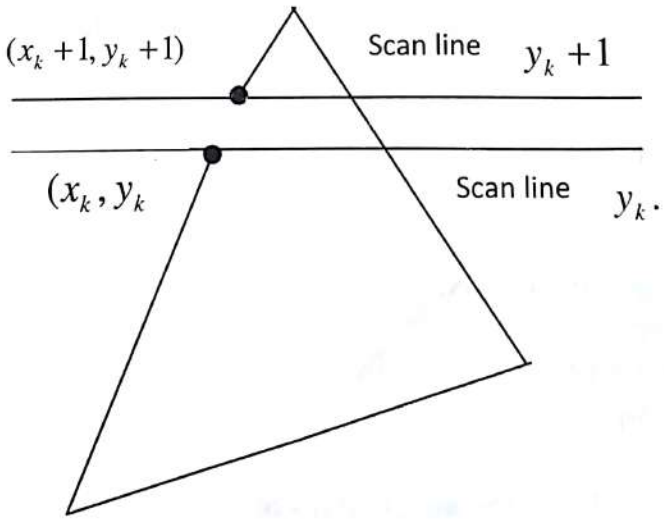
We can identify these vertices by tracing around the polygon boundary either in clockwise or counter clockwise order and observing the relative changes in vertex y coordinates as we move from one edge to next. If the endpoint y values of two consecutive edges monotonically increases or decrease, we need to count the middle vertex as a single intersection point for any scan line passing through that vertex. Otherwise the shared vertex represents a local extremum (minimum or maximum) on the polygon boundary, and the two edge intersections with the scan-line passing through that vertex.



We can identify these vertices by tracing around the polygon boundary either in clockwise or counter clockwise order and observing the relative changes in vertex y coordinates as we move from one edge to next. If the endpoint y values of two consecutive edges monotonically increases or decrease, we need to count the middle vertex as a single intersection point for any scan line passing through that vertex. Otherwise the shared vertex represents a local extremum (minimum or maximum) on the polygon boundary, and the two edge intersections with the scan-line passing through that vertex.

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

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



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

$$y_k + 1 - y_k = 1$$

The x-intersection value $x_k + 1$, on the upper scan line can be determined from the intersection value x_k, on the preceding scan line as

$$x_k + 1 = x_k + 1/m$$

Each successive xx intercept can thus be calculated by x values by the amount of $1/m$ along edge can be accomp0lished with integer operation by recalling that the slope m is the ratio two integers

$$m = \Delta y / \Delta x$$

Where Δx & Δy are the differences between the edge endpoint x and y co-ordinate values Thus incremental calculations of x intercepts along an edge for successive scan lines can expressed as

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

Algorithm Steps

1. the horizontal scanning of the polygon from its lowermost to its topmost vertex
2. identify the edge intersections of scan line with polygon
3. Build the edge table
   - Each entry in the table for a particular scan line contains the maximum y value for that edge, the x-intercept value (at the lower vertex) for the edge, and the inverse slope of the edge.
4. Determine whether any edges need to be splitted or not. If there is need to split, split the edges.
5. Add new edges and build modified edge table.
6. Build Active edge table for each scan line and fill the polygon based on intersection scan line with polygon edges.

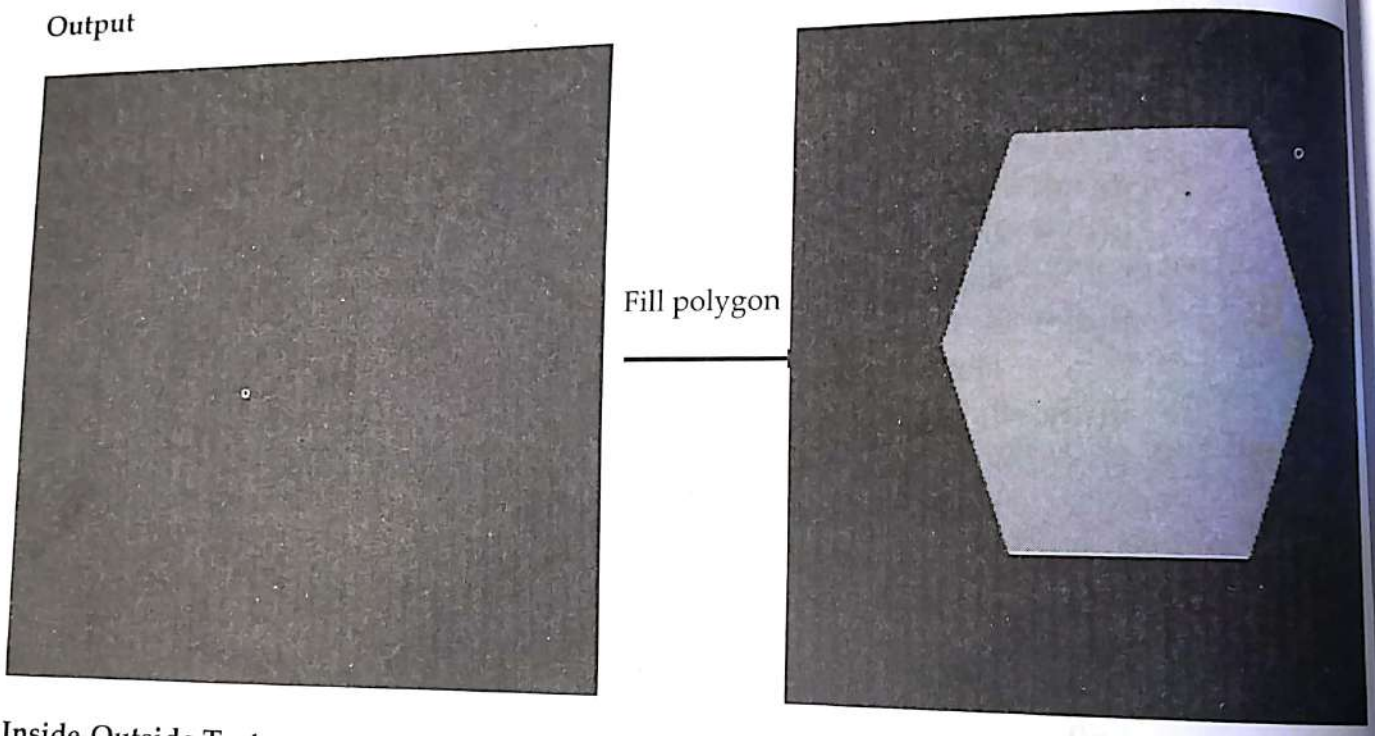
Lab problem 3.6: Program for scan line polygon fill algorithm

```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
void main()
{
        int n, i, j, k, gd, gm, dy, dx;
        int x, y, temp;
        int a[20][2], xi[20];
        float slope[20];
        clrscr();
        printf("\n\n\t Enter the no. of edges of polygon : ");
        scanf("%d", &n);
        printf("\n\n\t Enter the co-ordinates of polygon :\n\n\n ");
```

```
for(i=0;i<n;i++)
{
        printf("\t X%d Y%d : ",i ,i);
        scanf("%d %d", &a[i][0],&a[i][1]);
}
a[n][0]=a[0][0];
a[n][1]=a[0][1];
detectgraph(&gd,&gm);
initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
for(i=0;i<n;i++)
{
        line(a[i][0],a[i][1],a[i+1][0],a[i+1][1]);
}
getch();
for(i=0;i<n;i++)
{
        dy=a[i+1][1]-a[i][1];
        dx=a[i+1][0]-a[i][0];
        if(dy==0)
                slope[i]=1.0;
        if(dx==0)
                slope[i]=0.0;
        if((dy!=0)&&(dx!=0))
        {
                slope[i]=(float) dx/dy;
        }
}
for(y=0;y< 480;y++)
{
        k=0;
        for(i=0;i<n;i++)
        {
                if( ((a[i][1]<=y)&&(a[i+1][1]>y)) | | ((a[i][1]>y)&&(a[i+1][1]<=y)))
                {
                        xi[k]=(int)(a[i][0]+slope[i]*(y-a[i][1]));
                        k++;
                }
        }
        for(j=0;j<k-1;j++) /*- Arrange x-intersections in order -*/
        for(i=0;i<k-1;i++)
        {
                if(xi[i]>xi[i+1])
                {
                        temp=xi[i];
                        xi[i]=xi[i+1];
```

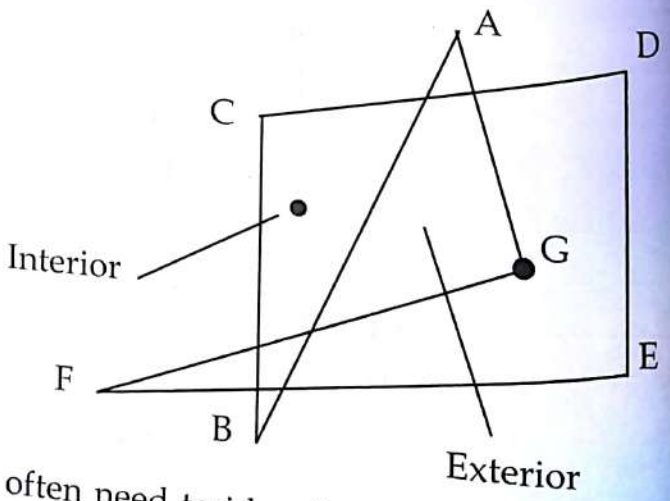
```
                    xi[i+1]=temp;
                }
            }
            setcolor(3);
            for(i=0;i<k;i+=2)
            {
                line(xi[i],y,xi[i+1]+1,y);
                getch();
            }
        }
}
```

Output



Inside-Outside Test

One simple way of finding whether the point is inside or outside a simple polygon is to test how many times a ray, starting from the point and going in any fixed direction, intersects the edges of the polygon. If the point is on the outside of the polygon the ray will intersect its edge an even number of times. If the point is on the inside of the polygon then it will intersect the edge an odd number of times. Unfortunately, this method won't work if the point is on the edge of the polygon.



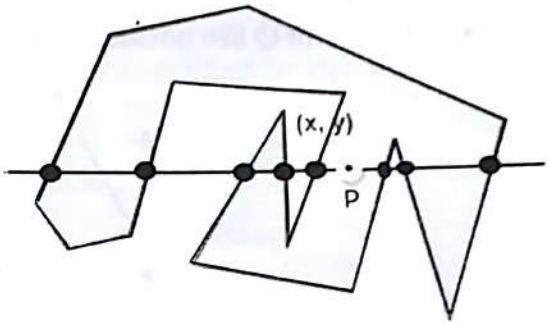
Area filling algorithms and other graphics package often need to identify interior and exterior region for a complex polygon in a plane. For example in figure below, it needs to identify interior and exterior region.

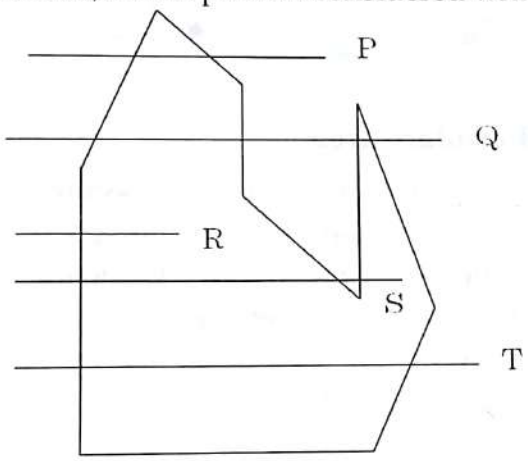
This method is also known as **counting number method**. While filling an object, we often need to identify whether particular point is inside the object or outside it. There are two methods by which we can identify whether particular point is inside an object or outside.

- Odd-Even method
- Nonzero winding number method

Odd-Even method

In this technique, we will count the edge crossing along the line from any point P(x, y) to infinity. If the number of interactions is odd, then the point P(x, y) is an interior point; and if the number of interactions is even, then the point P(x, y) is an exterior point. The following example depicts this concept. From the above figure, we can see that from the point P(x, y), the number of interactions point on the left side is 5 and on the right side is 3. From both ends, the number of interaction points is odd, so the point is considered within the object.





In the above figure the points P, Q, T lays outside of the polygon since the crosses the side of the polygon in even times i.e. 2 times, 4-times and 2-times respectively.
Also the points R and S lies inside of the polygon since they crosses the side of the polygon in odd times i.e. in 1-times and 3-times respectively.

Non-zero Winding Number Rule

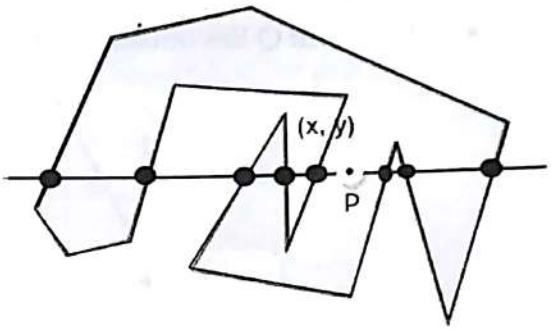
This method is also used with the simple polygons to test the given point is interior or not. It can be simply understood with the help of a pin and a rubber band. Fix up the pin on one of the edge of the polygon and tie-up the rubber band in it and then stretch the rubber band along the edges of the polygon. When all the edges of the polygon are covered by the rubber band, check out the pin which has been fixed up at the point to be test. If we find at least one wind at the point consider it within the polygon, else we can say that the point is not inside the polygon. In non-zero winding number method we need to know the direction of each edge in the polygon, i.e. whether the edge is clockwise or counter-clockwise. The winding number is number of times the polygon edges wind around a particular point in the counterclockwise direction.
Draw a scan line from the point to be test towards the left most of X direction.

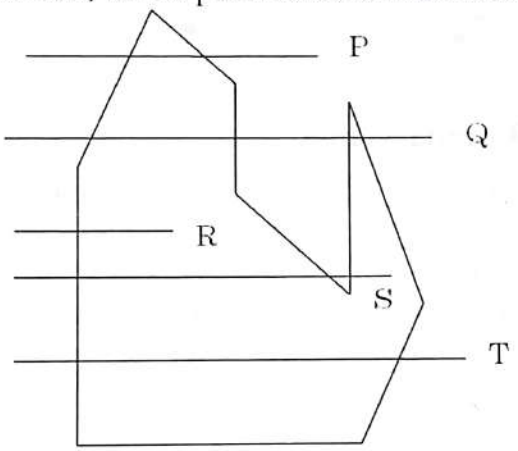
This method is also known as **counting number method**. While filling an object, we often need to identify whether particular point is inside the object or outside it. There are two methods by which we can identify whether particular point is inside an object or outside.

- Odd-Even method
- Nonzero winding number method

Odd-Even method

In this technique, we will count the edge crossing along the line from any point P(x, y) to infinity. If the number of interactions is odd, then the point P(x, y) is an interior point; and if the number of interactions is even, then the point P(x, y) is an exterior point. The following example depicts this concept. From the above figure, we can see that from the point P(x, y), the number of interactions point on the left side is 5 and on the right side is 3. From both ends, the number of interaction points is odd, so the point is considered within the object.



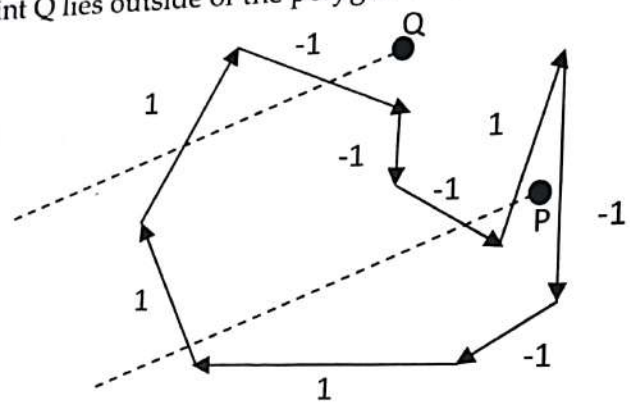


In the above figure the points P, Q, T lays outside of the polygon since the crosses the side of the polygon in even times i.e. 2 times, 4-times and 2-times respectively.
Also the points R and S lies inside of the polygon since they crosses the side of the polygon in odd times i.e. in 1-times and 3-times respectively.

Non-zero Winding Number Rule

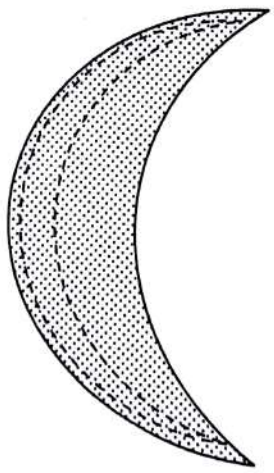
This method is also used with the simple polygons to test the given point is interior or not. It can be simply understood with the help of a pin and a rubber band. Fix up the pin on one of the edge of the polygon and tie-up the rubber band in it and then stretch the rubber band along the edges of the polygon. When all the edges of the polygon are covered by the rubber band, check out the pin which has been fixed up at the point to be test. If we find at least one wind at the point consider it within the polygon, else we can say that the point is not inside the polygon. In non-zero winding number method we need to know the direction of each edge in the polygon, i.e. whether the edge is clockwise or counter-clockwise. The winding number is number of times the polygon edges wind around a particular point in the counterclockwise direction.
Draw a scan line from the point to be test towards the left most of X direction.

- Give the value 1 to all the edges which are going to upward direction and all other -1 direction values.
- Check the edge direction values from which the scan line is passing and sum up them
- If the total sum of this direction value is non-zero, then this point to be tested an interior point, otherwise it is an exterior point.
- In the above figure, we sum up the direction values from which the scan line is passi then the total is 1 – 1 + 1 = 1; which is non-zero. So the point P is said to be an inter point.
- The point Q lies outside of the polygon since 1-1=0



Scan-Line Fill of Curved Boundary area

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



Boundary-fill Algorithm

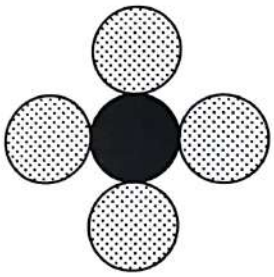
In Boundary filling algorithm starts at a point inside a region and paint the interior outward the boundary. If the boundary is specified in a single color, the fill algorithm proceeds outward pixel by until the boundary color is reached. It takes an interior point(x, y), a fill color, and a boundary color as the input. The algorithm starts by checking the color of (x, y). If its color is not equal to the fill color and the boundary color, then it is painted with the fill color and the

function is called for all the neighbors of (x, y). If a point is found to be of fill color or of boundary color, the function does not call its neighbors and returns. This process continues until all points up to the boundary color for the region have been tested. The boundary fill algorithm can be implemented by following two methods:

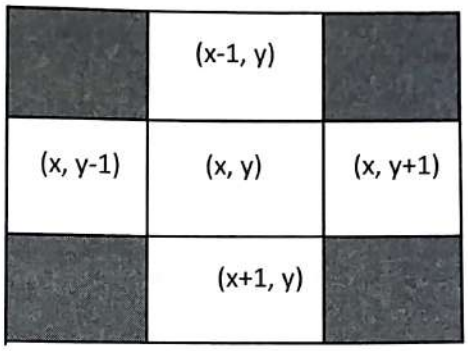
- 4- Connected if they are adjacent horizontally and vertically
- 8- Connected if they adjacent horizontally, vertically and diagonally.

4-connected pixels

After painting a pixel, the function is called for four neighboring points. These are the pixel positions that are right, left, above and below the current pixel. Areas filled by this method are called 4-connected.



4- Connected



The outline of this algorithm is given below

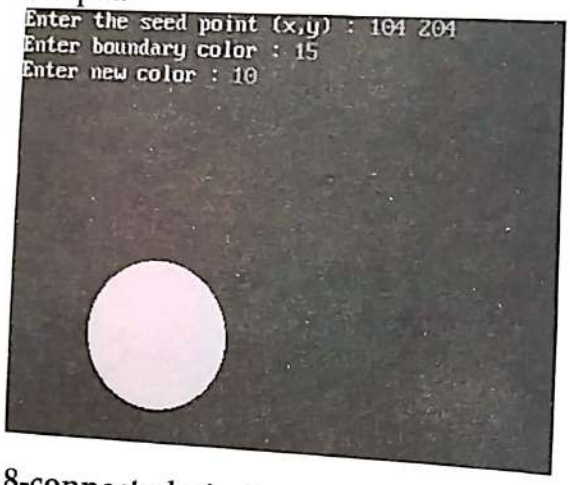
```
void Boundary_fill4(intx,inty,intb_color, intfill_color)
{
        int value=get pixel (x,y);
        if (value! =b_color&&value!=fill_color)
        {
                putpixel (x,y,fill_color);
                Boundary_fill 4 (x-1,y, b_color, fill_color);
                Boundary_fill 4 (x+1,y, b_color, fill_color);
                Boundary_fill 4 (x,y-1, b_color, fill_color);
                Boundary_fill 4 (x,y+1, b_color, fill_color);
        }
}
```

Lab problem 3.7: Program to implement the 4-connected Boundary fill algorithm.

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<dos.h>
void boundary_fill(int x, int y, int fcolor, int bcolor)
{
        if ((getpixel(x, y) != bcolor) && (getpixel(x, y) != fcolor))
        {
                delay(10);
                putpixel(x, y, fcolor);
                boundary_fill(x + 1, y, fcolor, bcolor);
```

```
            boundary_fill(x - 1, y, fcolor, bcolor);
            boundary_fill(x, y + 1, fcolor, bcolor);
            boundary_fill(x, y - 1, fcolor, bcolor);

        }
}
void main()
{
        int x, y, fcolor, bcolor;
        int gd=DETECT,gm;
        initgraph(&gd, &gm, "C:\\TurboC3\\BGI");
        printf("Enter the seed point (x, y) : ");
        scanf("%d%d", &x, &y);
        printf("Enter boundary color : ");
        scanf("%d", &bcolor);
        printf("Enter new color : ");
        scanf("%d", &fcolor);
        circle(100,200,45);
        boundary_fill(x, y, fcolor, bcolor);
        getch();
}
```

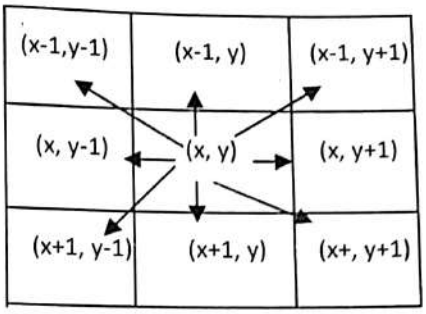
Output:



8-connected pixels

More complex figures are filled using this approach. The pixels to be tested are the neighboring pixels, the pixel on the right, left, above, below and the 4 diagonal pixels. Are filled by this method are called 8-connected. In this technique 8-connected pixels are used shown in the figure below. We are putting pixels above, below, right and left side of the curre pixels as we were doing in 4-connected technique. In addition to this, we are also putting pixe in diagonals so that entire area of the current pixel is covered. This process will continue un we find a boundary with different color.

8- Connected



The outline of this algorithm is given below

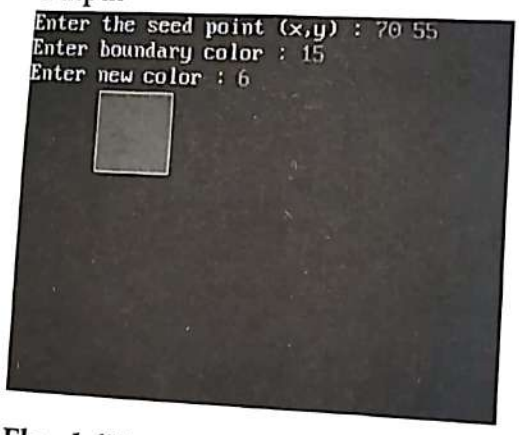
```
void Boundary-fill8(int x, int y, int b_color, int fill_color)
{
        int current;
        current=getpixel (x, y);
        if (current !=b_color&&current!=fill_color)
        (       putpixel (x,y,fill_color);
                Boundary_fill8(x-1, y, b_color,fill_color);
                Boundary_fill8(x+1,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

Lab problem 3.8: Program to implement the 8-connected Boundary fill algorithm.

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<dos.h>
void boundary_fill(int x, int y, int fcolor, int bcolor)
{
  if ((getpixel(x, y) != bcolor) && (getpixel(x, y) != fcolor))
  {
      delay(10);
      putpixel(x, y, fcolor);
      boundary_fill(x + 1, y, fcolor, bcolor);
      boundary_fill(x , y+1, fcolor, bcolor);
      boundary_fill(x+1, y + 1, fcolor, bcolor);
      boundary_fill(x-1, y - 1, fcolor, bcolor);
      boundary_fill(x-1, y, fcolor, bcolor);
      boundary_fill(x , y-1, fcolor, bcolor);
      boundary_fill(x-1, y + 1, fcolor, bcolor);
```

```
        boundary_fill(x+1, y - 1, fcolor, bcolor);

    }
}
void main()
{
    int x, y, fcolor, bcolor;
    int gd=DETECT, gm;
    initgraph(&gd, &gm, "C:\\TurboC3\\BGI");
    printf("Enter the seed point (x, y) : ");
    scanf("%d%d", &x, &y);
    printf("Enter boundary color : ");
    scanf("%d", &bcolor);
    printf("Enter new color : ");
    scanf("%d", &fcolor);
    rectangle(50,50,100,100);
    boundary_fill(x, y, fcolor, bcolor);
    getch();
}
```

Output



Flood-fill Algorithm

Flood fill algorithm is applicable when we want to fill an area that is not defined within a single color boundary. If fill area is bounded with different color, we can paint that area by replacing a specified interior color instead of searching of boundary color value. This approach is called flood fill algorithm. We start from a specified interior pixel (x, y) and reassign all pixel values that are currently set to a given interior color with desired fill color. Once again, this algorithm relies on the Four-connect or Eight-connect method of filling in the pixels. But instead of looking for the boundary color, it is looking for all adjacent pixels that are a part of the interior. There are some cases where the boundary color is different than the fill color. For situations like these Flood fill algorithm is used. Here the process is started in a similar way by examining the colors of neighboring pixels. But instead of matching it with a boundary color a specified color is matched.

The algorithm fills the area by 4- connected region by desired color.

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

Procedure for filling an 8- connected region

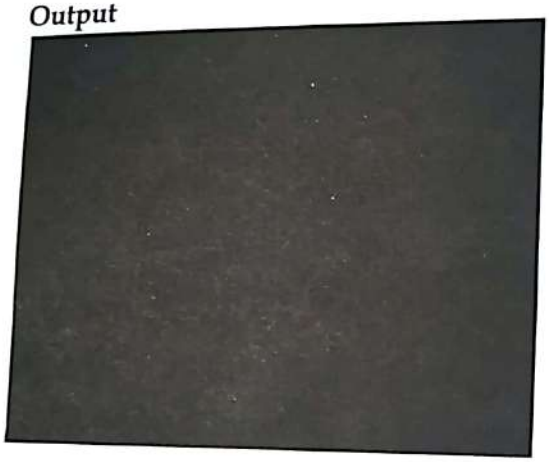
```
flood_ fill (x, y, old_color, new_color)
{
        putpixel(x,y,new_color)
        flood_ fill (x+1, y, old_color, new_color)
        flood_ fill (x-1, y, old_color, new_color)
        flood_ fill (x, y+1, old_color, new_color)
        flood_ fill (x, y-1, old_color, new_color)
        flood_ fill (x+1, y+1, old_color, new_color)
        flood_ fill (x-1, y-1, old_color, new_color)
        flood_ fill (x+1, y-1, old_color, new_color)
        flood_ fill (x-1, y+1, old_color, new_color)
}
```

Lab problem 3.9: Program for 4-connected flood fill

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<dos.h>
void flood(int,int,int,int);
void main()
{
        int gd,gm=DETECT;
        clrscr();
        detectgraph(&gd,&gm);
        initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
        rectangle(50,50,100,100);
        flood(55,55,9,0);
        getch();
}
```

```
    initgraph(&gd,&gm,"C:\\TURBOC3\\BGI");
    setcolor(5);
    rectangle(60,80,150,200);
    rectangle(95,140,120,200);
    line(60,80,100,15);
    line(100,15,150,80);
    circle(100,60,10);
    getch();
    closegraph();
}
```

Output

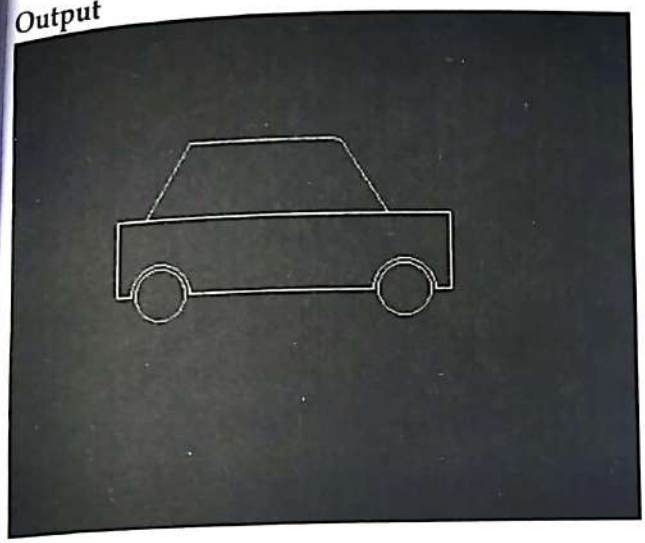


Lab problem 3.12: Program for creating simple car shape:

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<dos.h>
void main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "C:\\TurboC3\\BGI");
    cleardevice();
    line( 150, 100,  242, 100);
    ellipse(242, 105, 0, 90, 10, 5);
    line(150, 100,  120, 150);
    line(252, 105,  280, 150);
    line(100, 150,  320, 150);
    line(100, 150,  100, 200);
    line(320, 150,  320, 200);
    line(100, 200,  110, 200);
    line( 320, 200,  310, 200);
    arc(130, 200, 0, 180, 20);
    arc( 290, 200, 0, 180, 20);
```

```
    line( 270, 200,  150, 200);
    circle(130, 200, 17);
    circle(290, 200, 17);
    getch();
}
```

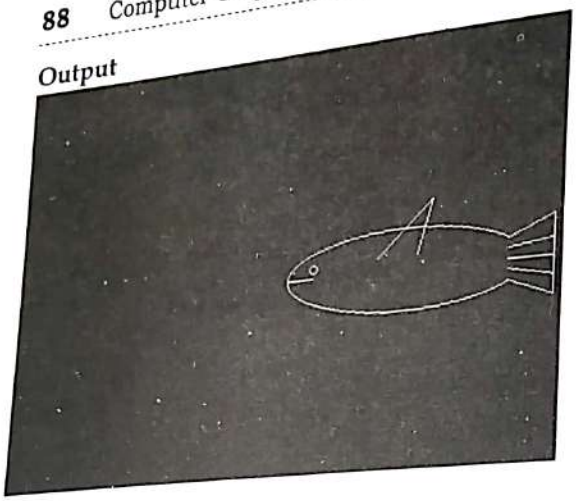
Output



Lab problem 3.13: Program for creating fish:

```
#include<stdlib.h>
#include<conio.h>
#include<dos.h>
#include<graphics.h>
#include<ctype.h>
void main()
{
    int gd=DETECT, gm;
    initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
    cleardevice();
    ellipse(520,200,30,330,90,30);
    circle(450,193,3);
    line(430,200,450,200);
    line(597,185,630,170);
    line(597,215,630,227);
    line(630,170,630,227);
    line(597,200,630,200);
    line(597,192,630,187);
    line(597,207,630,213);
    line(500,190,540,150);
    line(530,190,540,150);
    getch();
}
```

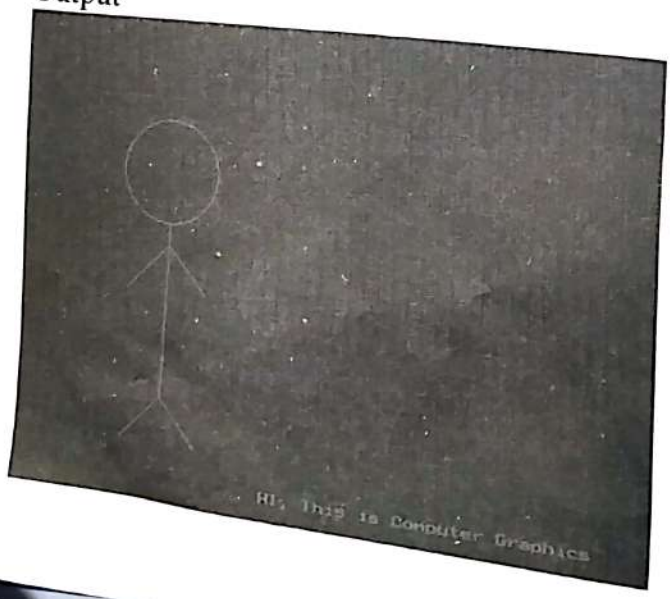
Output



Lab problem 3.14: Program for creating man object

```
#include<stdio.h>
#include<graphics.h>
#include<conio.h>
void main()
{
        int gd=DETECT,gm;
        initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
        setcolor(9);
        circle(150,150,35);
        line(150,185,150,300);
        line(150,200,120,230);
        line(150,200,180,230);
        line(150,300,120,330);
        line(150,300,180,330);
        outtextxy(230,350,"HI, This is Computer Graphics");
        getch();
}
```

Output



Exercises

1. Discuss DDA line drawing algorithm in detail. Also give its advantages and disadvantages.
2. Describe briefly Bresenham's line drawing algorithm. Why we prefer incremental algorithm over DDA?
3. How DDA line drawings differ from Bresenham's line drawing algorithm?
4. Digitize the line with endpoints (1, -6) and (4, 4) using BSA/BLA algorithm.
5. Digitize the line with endpoints (1, 6), (6, 10) using BSA algorithm.
6. Trace BSA for line with endpoints (15, 15), (10, 11).
7. Trace BSA for line with endpoints (20, 10), (30, 18).
8. Trace BSA for endpoints (5, 10), (10, 7).
9. Trace BSA for endpoints (20, 30), (30, 22)
10. Digitize the line with endpoints (1, -6) and (4, 4) using DDA algorithm.
11. Digitize the line with endpoints (1, 6), (6, 10) using DDA algorithm.
12. Trace DDA algorithm for line with endpoints (1, 2), (5, 6).
13. Trace DDA algorithm for endpoints (1, 7), (6, 3).
14. Rasterizing the line from (10,5) to (15,9) using Bresenhams line drawing Algorithm.
15. Rasterizing the circle with radius r=5 and center =(100,100) with midpoint circle generation algorithm.
16. Find the points on a circle one of its octants with the circle centered at (5,5) and has a radius of 8 units.
17. What is scan conversion? Explain Bresenham's incremental circle algorithm for first quadrant assuming all variables as integers.
18. Write an algorithm for scan conversion of circle using Bresenham's method.
19. Differentiate between flood fill and boundary fill algorithms.
20. Describe boundary fill algorithm with suitable example.

❑❑❑

Chapter

4

Two-Dimensional Geometric Transformations

STRUCTURE

4.0 Introduction

4.1 Two-Dimensional translation, Rotation, Scaling, Reflection and Shearing

4.2 2D Composite Transformations, Transformation between Coordinates

4.3 Two Dimensional Viewing:

- 4.3.1 Viewing Pipeline
- 4.3.2 Window to Viewport Coordinate Transformation

4.4 Clipping

- 4.4.1 point
- 4.4.2 Lines and
- 4.4.3 Polygon

Introduction

Mathematically, a geometric transformation is a function f whose domain and range are points. We denote by **P** a general point before any transformation and by **P′** the same point after a transformation. The notation **P′**= f(**P**) implies that the transformed point **P′** is obtained by applying f to **P**. We call our transformations geometric because they have geometric interpretations. Thus, only certain functions f can be used. Years of study and practical experience have shown that in order for it to be meaningful as a geometric transformation, a function must satisfy two conditions: it has to be **onto** and **one-to-one**. The operations that are applied to geometrical description of an object to change its position, orientation, or size are called **geometric transformations**.

Types of transformation are listed below:

1. Rigid body transformation (transformation without deformation in shape)
   - Translation
   - Rotation
   - Reflection
2. Non rigid body transformation (transformation with change in shape)
   - Scaling
   - Shearing

Why Geometric Transformations?

In computer graphics, transformations of 2D objects are essential to many graphics applications (as a viewing aid, as a modeling tool, as an image manipulation tool). Transformations are needed to manipulate the initially created object and to display the modified object without redrawing it.

1. Composite Transformation/Basic Transformations
   - Successive Translation
   - Successive Rotation
   - Pivot point Rotation
   - Successive Scaling
   - Fixed point Scaling
2. Other Transformation
   - Reflection
   - Shear

Rotation, Translation and scaling are three major transformations that are extensively used by all most all graphical packages or graphical subroutines in applications. Other than these reflection and shearing transformations are also used by some graphical packages.

Matrix representation of 2D Transformation

Matrix notation was researched by the British mathematician Arthur Cayley around1858. Cayley formalized matrix algebra, along with the American mathematicians Benjamin and Charles Pierce. Matrices have also come to have important applications in computer graphics where they have been used to represent rotations and other transformations of images. Many graphics applications involve sequences of geometric transformations. An animation might require and object to be translated, rotated and scaling at each increment of the motion

Geometric transformations like rotation, translation, scaling, and projection can be accomplished with matrix multiplication and matrices are a convenient and efficient way to represent a sequence of transformations. Represent a 2D transformation by a 2×2 matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Apply the transformation to a point

$$\begin{aligned} x' &= ax + by \\ y' &= cx + dy \end{aligned} \Longrightarrow \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

Image = T.M. × Object

If object and image are represented in row matrix then

$$\begin{aligned} x' &= ax + by \\ y' &= cx + dy \end{aligned} \Longrightarrow \begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \times \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Image = Object × T.M.

Transformations can be combined by matrix multiplication

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} \begin{bmatrix} i & j \\ k & l \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Image = $T.M_1 \times T.M_2 \times T.M_3 \times$ Object

Homogeneous co-ordinate representation of 2D Transformation

The homogeneous co-ordinate system provides a uniform frame-work for handling different geometric transformations, simply as multiplication of matrices. In homogeneous coordinate representation each 2D point (x, y) is represented as homogeneous coordinate triple (x_h, y_h, h), where $x = x_h/h$, $y = y_h/h$ (h is 1 usually for 2D case). The reason why this coordinate system is called 'homogeneous' is because it is possible to transform functions such as $f(x, y)$ into the form $f(x/h, y/h)$ without disturbing the degree of the curve. In the field of projective geometry it is a very powerful concept.

- (x_h, y_h, h) represents a point at location $(x_h/h, y_h/h)$
- $(x_h, y_h, 0)$ represents a point at infinity
- $(0, 0, 0)$ Is not allowed

Examples 4.1

- 2D point (2, 1) can be represented as (2, 1, 1) or (4, 2, 2) or (6, 3, 3) so on.

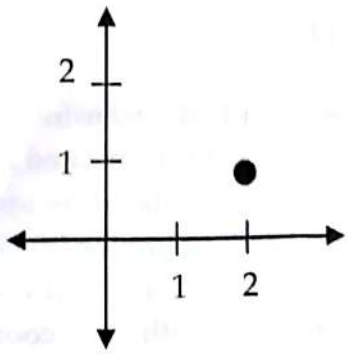


Figure 4.1: Possible Homogenous coordinate of a 2D Point (2, 1)

Why Homogeneous Co-ordinates?

The homogeneous co-ordinate system provides a uniform framework for handling differe geometric transformations, simply as multiplication of matrices.

- To express any two-dimensional transformation as a matrix multiplication.
- To represent all the transformation equations as matrix multiplications.
- To perform more than one transformation at a time.
- To reduce unwanted calculations of intermediate steps, to save time and memory an produce a sequence of transformations.

Two-Dimensional translation, Rotation, Scaling, Reflection and Shearing

2D Translation

A translation is displacement from original place. This displacement happens to be along a straight line; where two distances involves one is along x-axis that is *tx* and second is along y-axis that is *ty*. The same is shown in the figure also we can express it with following equation as well as by matrix. If P(*x*, *y*) is translated to a position P′(*x*′, *y*′) by *tx* units parallel to x-axis and *ty* units parallel to y axis then,

$$x' = x + tx$$
$$y' = y + ty$$

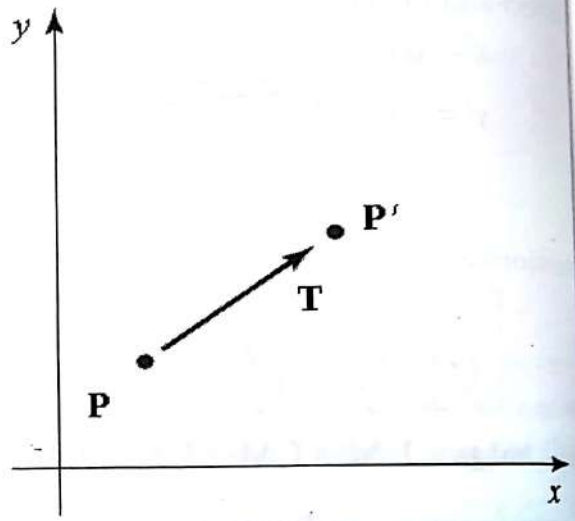


Figure 4.2: Translating a point from position P to position P′ using a translation vector T.

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

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

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

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

Translation is a rigid-body transformation that moves objects without deformation. That is every point on the object is translated by the same amount.

A straight line can be translated by applying the above transformation equation to each of th line endpoints and redrawing the line between the new coordinates. Similarly a polygon can b translated by applying the above transformation equation to each vertices of the polygon an redrawing the polygon with new coordinates. Similarly curved objects can be translated. Fo

example to translate circle or ellipse, we translate the center point and redraw the same using new center point.

By using this homogeneous co-ordinate system a 2D point would be *(x, y, 1)*. The matrix formulation for 2D translation for $T(t_x, t_y)$ is :

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Example 4.2: Translate the given points *(2, 5)* by the translating value *(3, 3)*

Solution: $(x, y) = (2, 5)$

$T_x = 3,$

$T_y = 3$

$x' = x + t_x$

$Y' = y + t_y$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \end{bmatrix} + \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 5 \\ 8 \end{bmatrix}$$

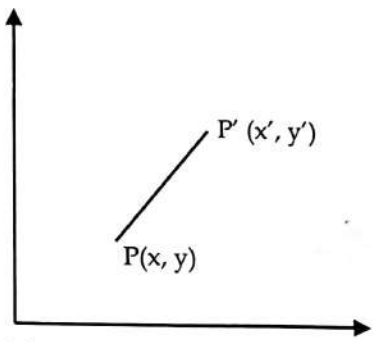


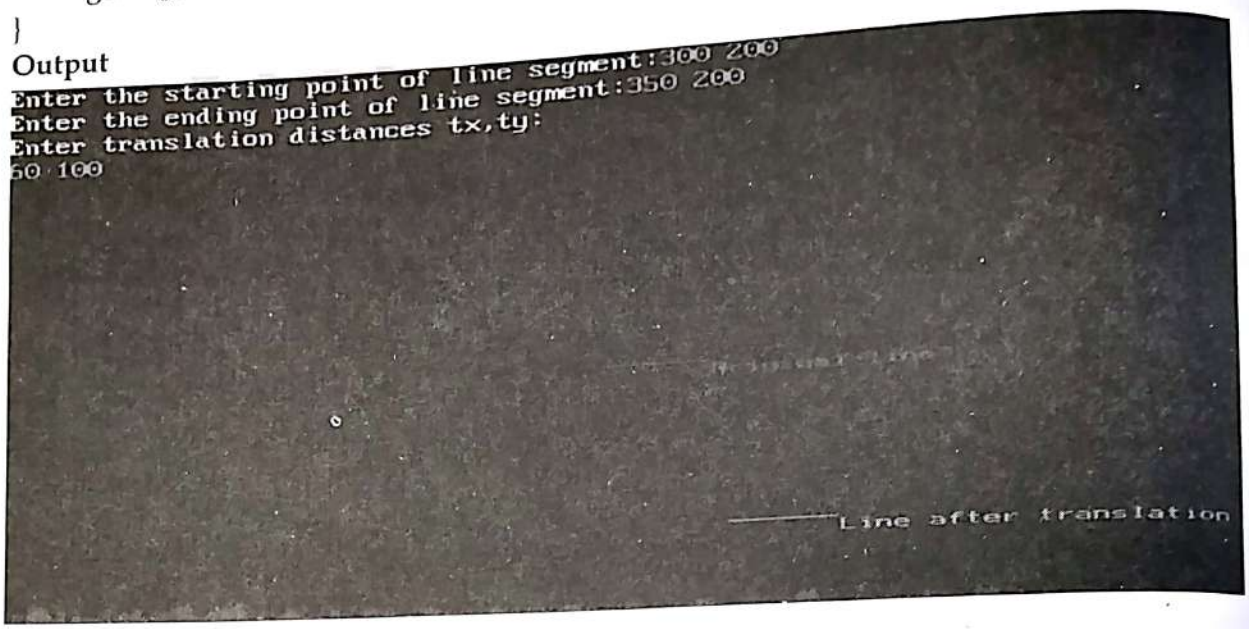
Figure 4.3: Translating a point of example 4.2

Lab problem 4.1: Program for translation

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
void main()
{
    int gd=DETECT, gm;
    int x1,y1,x2,y2,tx,ty,x3,y3,x4,y4;
    initgraph(&gd, &gm,"C:\\TurboC3\\BGI");
    printf("Enter the starting point of line segment:");
    scanf("%d %d",&x1,&y1);
    printf("Enter the ending point of line segment:");
    scanf("%d %d",&x2,&y2);
    printf("Enter translation distances tx, ty:\n");
    scanf("%d%d", &tx,&ty);
    setcolor(5);
    line(x1,y1,x2,y2);
    outtextxy(x2+2,y2+2,"Original line");
    x3=x1+tx;
    y3=y1+ty;
    x4=x2+tx;
    y4=y2+ty;
    setcolor(7);
    line(x3,y3,x4,y4);
```

```
    outtextxy(x4+2,y4+2,"Line after translation");
    getch();
}
```

Output



2D Rotation

A two dimensional rotation is applied to an object by repositioning it along a circular path the *xy* plane. A rotation repositions all points in an object along a circular path. To genera rotation, we specify a rotation angle θ, through which the co-ordinates are to be rotate Rotation can be made by angle θ either clockwise or anticlockwise direction. The positive rotates object in anti-clockwise direction and the negative value of θ rotates the object in cloc wise direction. The point about to which object is to be rotated is called **pivot point.**

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

Rotation about origin

If P(*x, y*) is rotated to a new position P′(*x, y′*) in anti-clockwise direction by an angle θ , then (*y′*) can be calculated as following. Let the angle made by line OP with x-axis is ϕ and the radi of circulation path is r then,

$$x = r\cos\phi$$

$$y = r\sin\phi$$

Also,

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

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

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

$$x' = x\cos\theta - y\sin\theta \ldots\ldots\ldots\ldots\ldots\ldots\ldots (i)$$

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

$$x' = x\sin\theta + y\cos\theta \ldots\ldots\ldots\ldots\ldots\ldots\ldots (ii)$$

In matrix form:

Figure 4.4: Rotation of a point from posit (x, y) to position (x′, y′) through angle θ relative to the coordinate origin.

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

For Rotation about origin the homogeneous matrix equation will be

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

Which gives the

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

$$y' = x\sin\theta + y\cos\theta$$

Which are equation for rotation of *(x, y)* with angle φ and taking pivot as origin.

Rotation from any arbitrary pivot point (x_r, y_r)

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

Hence,

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

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

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

Since $r\cos\phi = x - x_r, r\sin\phi = y - y_r$

$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta \quad \text{.....(i)}$$

Similarly,

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

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

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

Since $r\cos\phi = x - x_r, r\sin\phi = y - y_r$

$$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta \quad \text{.....(ii)}$$

Figure4.5: Rotation of a point from position *(x, y)* to position *(x', y')* through angle θ about rotation point (x_r, y_r).

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

The rotation operation represented in matrix equation using homogeneous co-ordinate system as,

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

Expanding this equation we get

$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta \quad \text{.....(i)}$$

$$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta \quad \text{.....(ii)}$$

Which are actually the equations for rotation of (x, y) from the pivot point (x_r, y_r)

Example 4.3: A point (4, 3) is rotated counterclockwise by an angle of 45°. Find the rotation matrix and the resultant point.

Solution:

$$\text{Composite Transformation Matrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$= \begin{bmatrix} \cos 45° & -\sin 45° \\ \sin 45° & \cos 45° \end{bmatrix}$$

$$= \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

We have,

Image = Transformation Matrix ×Object

$$P' = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 4 \\ 3 \end{bmatrix} = \begin{bmatrix} \frac{4}{\sqrt{2}} - \frac{3}{\sqrt{2}} \\ \frac{4}{\sqrt{2}} + \frac{3}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{7}{\sqrt{2}} \end{bmatrix}$$

Example 4.4: Find the transformation of triangle A(1, 0), B(0, 1) and C(1,1) by rotating 90° about the origin and then translating one unit in x and y direction.

Solution

$$\text{Object} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Composite Transformation matrix = T(1, 1) R(90°)

$$= \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 90° & -\sin 90° & 0 \\ \sin 90° & \cos 90° & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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

We have,

Image = Transformation Matrix ×Object

$$\text{Image} = \begin{bmatrix} 0 & -1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 2 \\ 1 & 1 & 1 \end{bmatrix}$$

The coordinates of the image are A′(1, 2), B′(0, 1) and C′(0, 2).

Example 4.5: Perform a counterclockwise 45° rotation of a triangle A(2, 3), B(5, 5) and C(4, 3) about point (1, 1).

Solution:

$$\text{Object} = \begin{bmatrix} 2 & 5 & 4 \\ 3 & 5 & 3 \\ 1 & 1 & 1 \end{bmatrix}$$

Composite Transformation Matrix $= T(1,1) \times R(45°) \times T(-1,-1)$

$$= \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos 45° & -\sin 45° & 0 \\ \sin 45° & \cos 45° & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

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

We have,

Image = Transformation Matrix ×Object

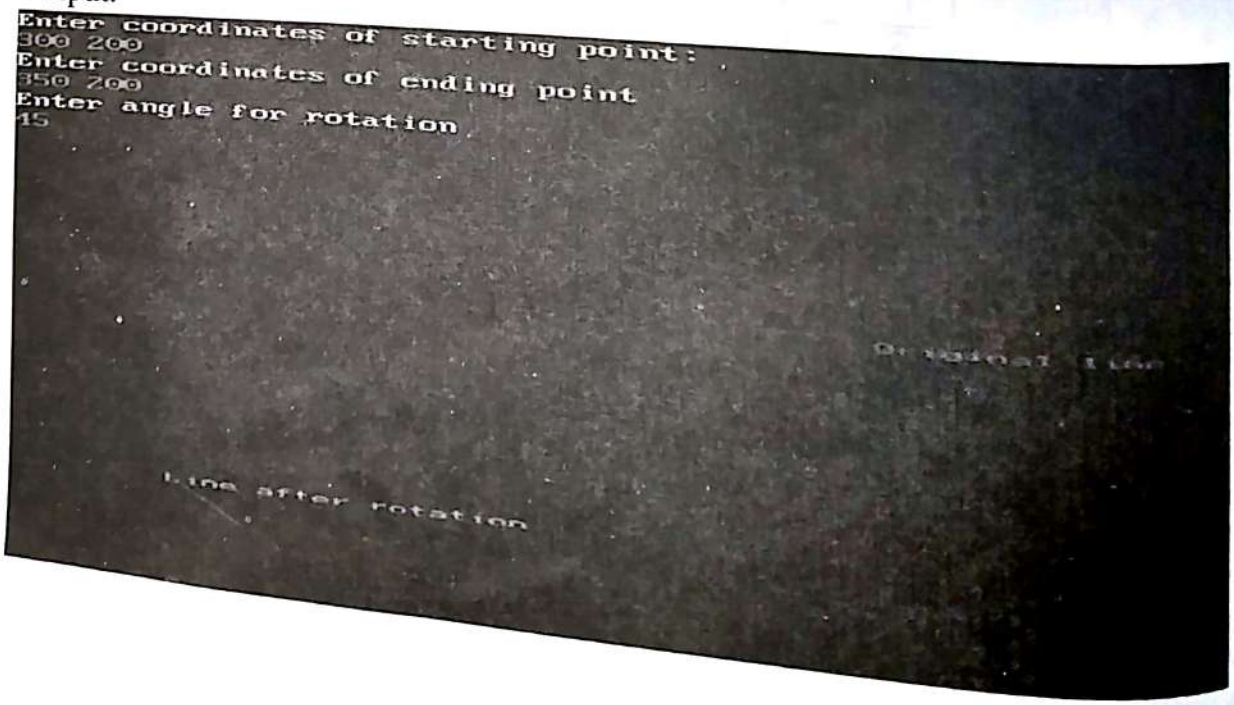
$$\text{Image} = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 1 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & -\sqrt{2}+1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 & 4 \\ 3 & 5 & 3 \\ 1 & 1 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} -\frac{1}{\sqrt{2}}+1 & 1 & \frac{1}{\sqrt{2}}+1 \\ \frac{3}{\sqrt{2}}+1 & \frac{8}{\sqrt{2}}+1 & \frac{5}{\sqrt{2}}+1 \\ 1 & 1 & 1 \end{bmatrix}$$

Lab problem 4.2: Program for Rotation

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
void main()
{
        int gd=DETECT,gm;
        float x1,y1,x2,y2,x3,y3,x4,y4,a,t;
        initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
        printf("Enter coordinates of starting point:\n");
        scanf("%f%f",&x1,&y1);
        printf("Enter coordinates of ending point\n");
        scanf("%f%f",&x2,&y2);
        printf("Enter angle for rotation\n");
        scanf("%f",&a);
        setcolor(5);
        line(x1,y1,x2,y2);
        outtextxy(x2+2,y2+2,"Original line");
        t=a*(3.14/180);
        x3=(x1*cos(t))-(y1*sin(t));
        y3=(x1*sin(t))+(y1*cos(t));
        x4=(x2*cos(t))-(y2*sin(t));
        y4=(x2*sin(t))+(y2*cos(t));
        setcolor(7);
        line(x3,y3,x4,y4);
        outtextxy(x3+2,y3+2,"Line after rotation");
        getch();
}
```

Output:



Scaling

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

i.e. $x' = x.s_x$

and $y' = y.s_y$

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

Figure 4.6: A line scaled with $s_x=s_y=0.5$ is reduced in size and moved closer to the coordinate origin.

The matrix equation for scaling is:

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

The scaling operation represented in matrix equation using homogeneous co-ordinate system as,

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

Which exactly gives the equations $x' = x.s_x$ and $y' = y.s_y$.

Fixed point scaling

To control the location of scaled object we can choose the position called fixed point. Let co-ordinates of fixed point = (x_f, y_f). It can be any point on the object. Scaling factor with value less than 1 moves object closer to the origin whereas value of scaling factor greater than 1 moves object away from origin. For a vertex with co-ordinate (x, y), the scaled co-ordinates (x', y') are calculated as:

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

or equivalently,

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

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

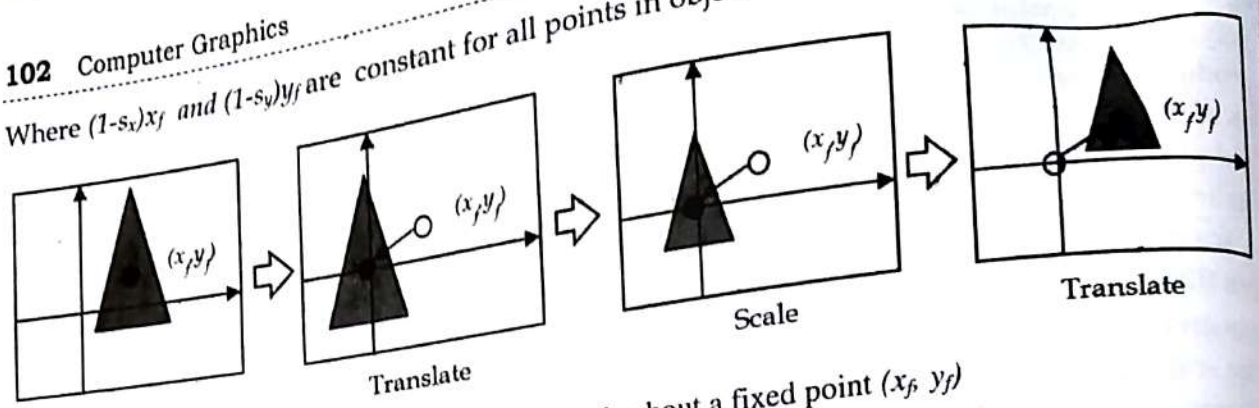


Figure 4.7: Scaling of a triangle about a fixed point (x_f, y_f)

$$T(x_f, y_f)\cdot S(s_x, s_y)\cdot T(-x_f, -y_f) = S(x_f, y_f, s_x, s_y)$$

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

The homogeneous matrix equation for fixed point scaling is

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

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

Example 4.6: Scale the polygon with coordinates A (2, 5), B(7, 10) and C(10, 2) by two units in x direction and two units in y direction.

Solution

Here, Sx=2 and Sy=2. Therefore, transformation matrix is given as

$$S = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

The object matrix is:

$$Object = \begin{bmatrix} 2 & 7 & 10 \\ 5 & 10 & 2 \end{bmatrix}$$

We have,

Image = Transformation Matrix ×Object

$$= \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \times \begin{bmatrix} 2 & 7 & 10 \\ 5 & 10 & 2 \end{bmatrix} = \begin{bmatrix} 4 & 14 & 20 \\ 10 & 20 & 4 \end{bmatrix}$$

The coordinates of the image after scaling are A′(4, 10), B′(14, 20) and C′(20, 4).

Example 4.7: Find the transformation matrix that transforms the given square ABCD to half its size with centre still remaining at the same position. The coordinates of the square are: A(1, 1), B(3, 1), C(3, 3), D(1, 3) and centre at (2, 2). Also find the resultant coordinates of square.

Solution:

This transformation can be carried out in the following steps.

1. Translate the square so that its center coincides with origin.
2. Scale the square with respect to the origin.
3. Translate the square back to the original position.

$$\text{Object} = \begin{bmatrix} 1 & 3 & 3 & 1 \\ 1 & 1 & 3 & 3 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Thus, the composite transformation matrix is formed by the multiplication of three matrices.

$$\text{Composite Transformation Matrix} = T(2,2) \times S(Sx, Sy) \times T(-2,-2)$$

$$= \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.5 & 0 & 1 \\ 0 & 0.5 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

We have,

$$\text{Image} = \text{Transformation Matrix} \times \text{Object}$$

$$\text{Image} = \begin{bmatrix} 0.5 & 0 & 1 \\ 0 & 0.5 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 & 3 & 1 \\ 1 & 1 & 3 & 3 \\ 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1.5 & 2.5 & 2.5 & 1.5 \\ 1.5 & 1.5 & 2.5 & 2.5 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

The resultant coordinates of the resultant square (image) are A′(1.5, 1.5), B′(2.5, 1.5), C(2.5, 2.5) and D′(1.5, 2.5).

Example 4.8: Find out the final coordinates of a figure bonded by the coordinates (1, 1), (3, 4), (5, 7), (10, 3) when rotated about a point (8, 8) by 30° in clock wise direction and scaled by two unit in x-direction and three unit y direction.

Solution

$$\text{Object} = \begin{bmatrix} 1 & 3 & 5 & 10 \\ 1 & 4 & 7 & 3 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\text{Composite Transformation Matrix} = S(2,3)\,T(8,8) \times R(-30°) \times T(-8,-8)$$

$$= \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos 45° & -\sin 45° & 0 \\ \sin 45° & \cos 45° & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0.866 & 0.5 & -2.928 \\ -0.5 & 0.866 & 5.072 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1.73 & 1 & -5.856 \\ -1.5 & 2.598 & 15.216 \\ 0 & 0 & 1 \end{bmatrix}$$

We have,

Image = Transformation Matrix ×Object

$$\text{Image} = \begin{bmatrix} 1.73 & 1 & -5.856 \\ -1.5 & 2.598 & 15.216 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 3 & 5 & 10 \\ 1 & 4 & 7 & 3 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} -3.124 & 3.34 & 9.804 & 14.464 \\ 16.414 & 21.108 & 25.902 & 8.01 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

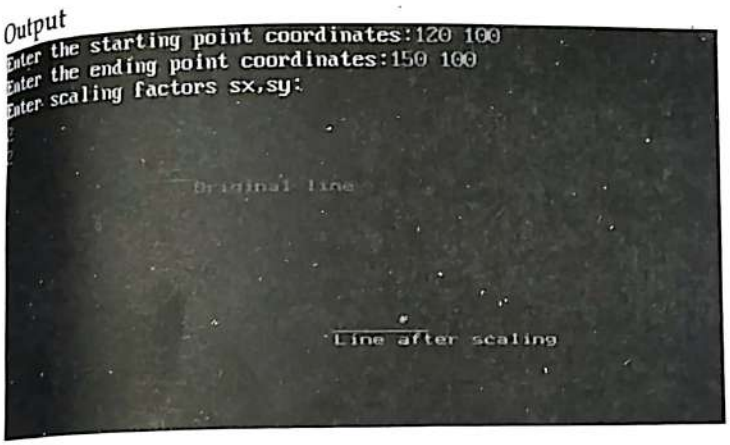
The final coordinates after the transformation are (-3.124, 16.414), (3.34, 21.108), (9.804, 25.902) and (14.464, 8.01).

Lab problem 4.3: Program for scaling

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
void main()
{
    int gd=DETECT,gm;
    float x1,y1,x2,y2,sx,sy,x3,y3,x4,y4;
    initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
    printf("Enter the starting point coordinates:");
    scanf("%f %f",&x1,&y1);
    printf("Enter the ending point coordinates:");
    scanf("%f %f",&x2,&y2);
    printf("Enter scaling factors sx,sy:\n");
    scanf("%f%f",&sx,&sy);
    setcolor(5);
    line(x1,y1,x2,y2);
```

```
outtextxy(x2+2,y2+2,"Original line");
x3=x1*sx;
y3=y1*sy;
x4=x2*sx;
y4=y2*sy;
setcolor(7);
line(x3,y3,x4,y4);
outtextxy(x3+2,y3+2,"Line after scaling");
getch();
```

Output



Reflection

Reflection is a transformation that produces a mirror image of an object. In 2D-transformation, reflection is generated relative to an axis of reflection. The reflection of an object to a relative axis of reflection is same as 180° rotation about the reflection axis.

Following are examples of some common reflections:

- Reflection about the line y=0, the X- axis
- Reflection about the line x=0, the Y- axis
- Reflection about the line passing through the coordinate origin
- Reflection of an object with respect to the straight line y=x
- Reflection of an object with respect to the straight line y= -x

Reflection about X-axis

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

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

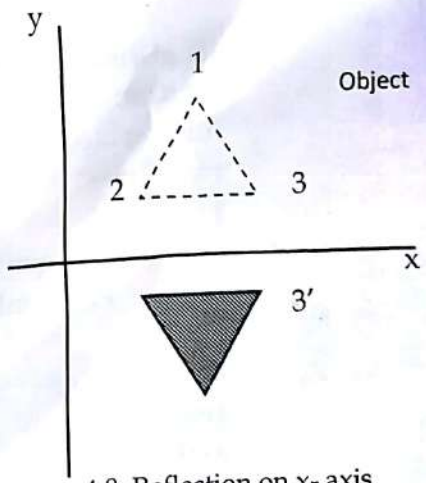


Figure 4.8: Reflection on x- axis

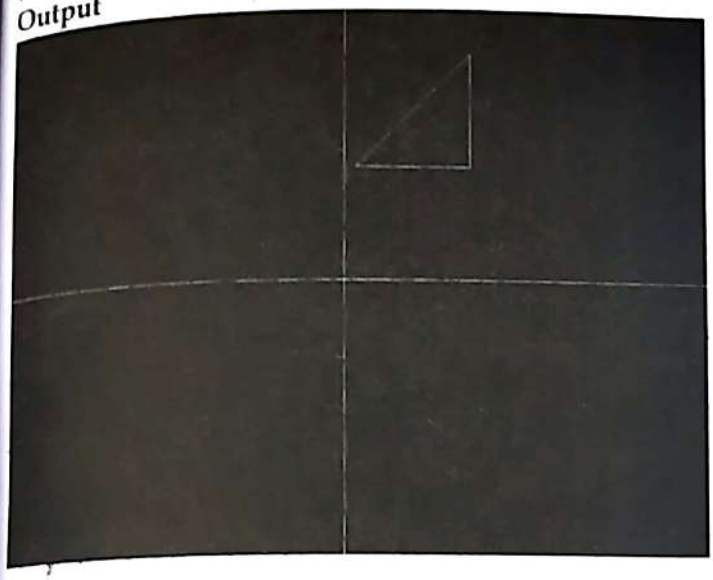
gives the reflection of a point.
To reflect a 2D object, reflect each distinct points of object by above equation, then joining the points with straight line, redraws the image for reflected image.

Lab problem 4.4: Program for reflection along x-axis

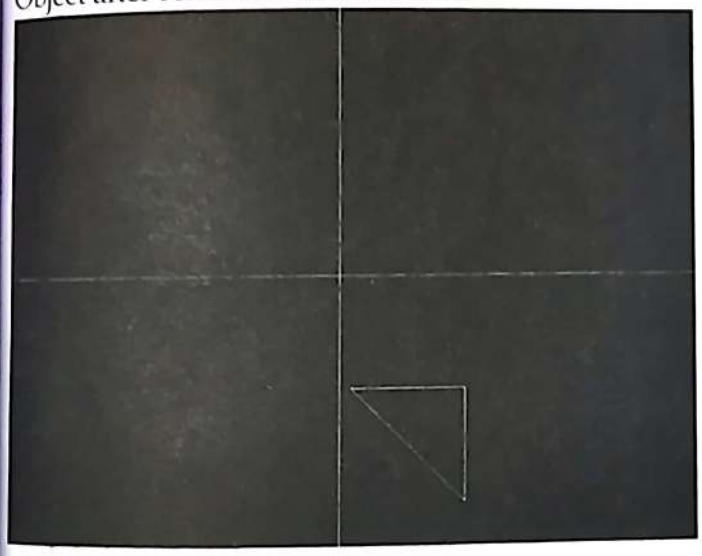
```
# include <stdio.h>
# include <conio.h>
# include <graphics.h>
# include <math.h>
char IncFlag;
int PolygonPoints[3][2] ={{10,100},{110,100},{110,200}};
void PolyLine()
{
        int iCnt;
        cleardevice();
        line(0,240,640,240);
        line(320,0,320,480);
        for (iCnt=0; iCnt<3; iCnt++)
        {
                line(PolygonPoints[iCnt][0],PolygonPoints[iCnt][1],
                PolygonPoints[(iCnt+1)%3][0],PolygonPoints[(iCnt+1)%3][1]);
        }
}
void Reflect()
{
        float Angle;
        int iCnt;
        int Tx,Ty;
        printf("endl");;
        for (iCnt=0; iCnt<3; iCnt++)
                PolygonPoints[iCnt][1] = (480 - PolygonPoints[iCnt][1]);
}
void main()
{
        int gDriver = DETECT, gMode;
        int iCnt;
        initgraph(&gDriver, &gMode, "C:\\TurboC3\\BGI");
        for (iCnt=0; iCnt<3; iCnt++)
        {
                PolygonPoints[iCnt][0] += 320;
                PolygonPoints[iCnt][1] = 240 - PolygonPoints[iCnt][1];
        }
        PolyLine();
        getch();
        Reflect();
        PolyLine();
```

```
    getch();
}
```

Output



Object after reflection about x-axis:



Reflection about Y-axis

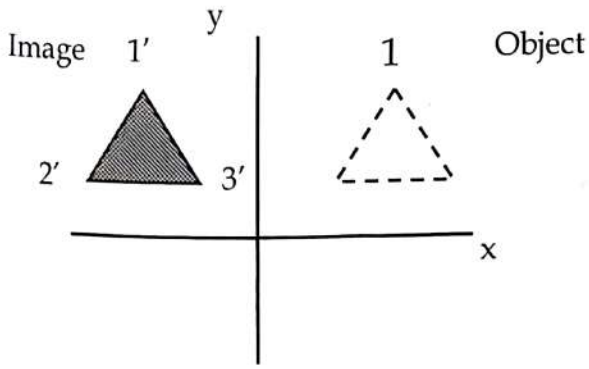


Figure 4.9: Reflection on y- axis

The line representing y-axis is x = 0. The reflection of a point P(x, y) on y-axis changes the sign of x-coordinate. i.e. P(x, y) changes to P'(-x, y). Hence reflection of a point on y-axis is obtained by following matrix equation

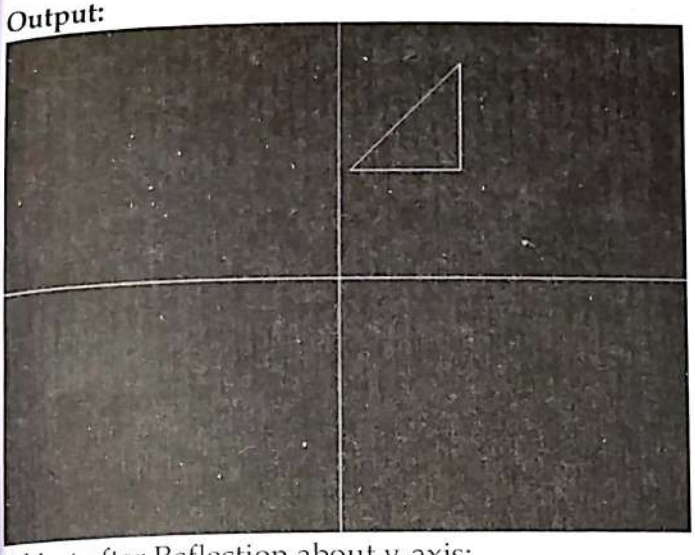
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

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

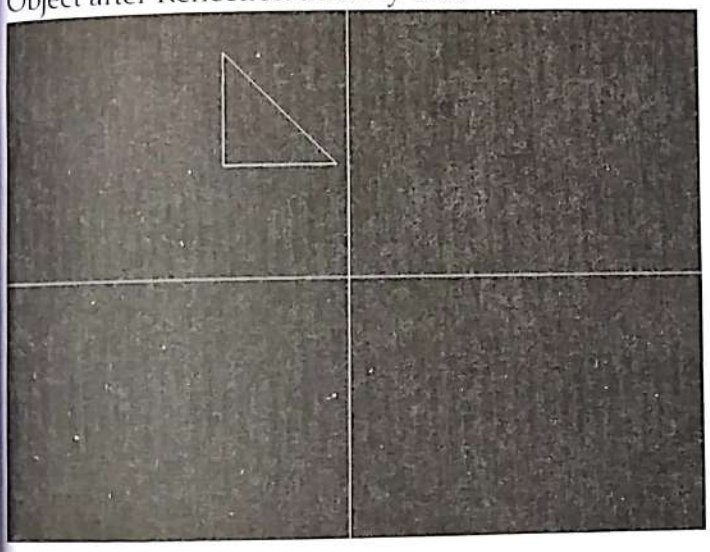
Lab problem 4.5: Program for Reflection about y-axis

```
# include <stdio.h>
# include <conio.h>
# include <graphics.h>
# include <math.h>
char IncFlag;
int PolygonPoints[3][2] = {{10,100},{110,100},{110,200}};
void PolyLine()
{
        int iCnt;
        cleardevice();
        line(0,240,640,240);
        line(320,0,320,480);
        for (iCnt=0; iCnt<3; iCnt++)
        {
            line(PolygonPoints[iCnt][0],PolygonPoints[iCnt][1],
            olygonPoints[(iCnt+1)%3][0],PolygonPoints[(iCnt+1)%3][1]);
        }
}
void Reflect()
{
        float Angle;
        int iCnt;
        int Tx,Ty;
        for (iCnt=0; iCnt<3; iCnt++)
            PolygonPoints[iCnt][0] = (640 - PolygonPoints[iCnt][0]);
}
void main()
{
        int gd = DETECT, gm;
        int iCnt;
        initgraph(&gd, &gm, "C:\\TurboC3\\BGI");
        for (iCnt=0; iCnt<3; iCnt++)
        {
            PolygonPoints[iCnt][0] += 320;
            PolygonPoints[iCnt][1] = 240 - PolygonPoints[iCnt][1];
        }
        PolyLine();
        getch();
        Reflect();
        PolyLine();
        getch();
}
```

Output:



Object after Reflection about y-axis:



Reflection about origin

The reflection on the line perpendicular to xy-plane and passing through flips x and y co-ordinates both. So sign of x and y co-ordinate value changes.

$x' = -x$

$y' = -y$

The equivalent matrix equation for the point is:

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

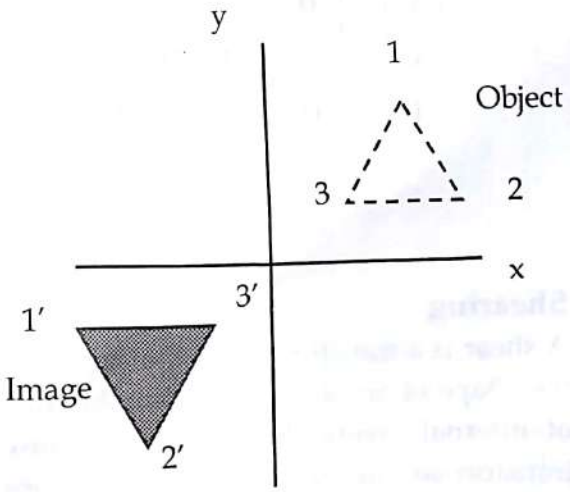


Figure 4.10: Reflection on about Image

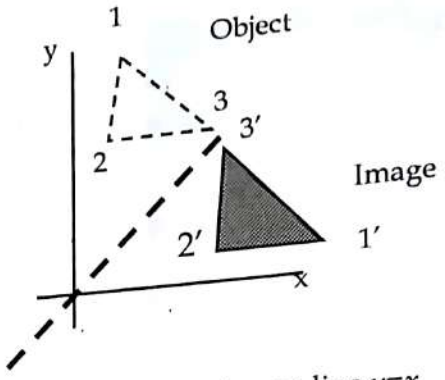


Figure 4.11: Reflection on line $y=x$

Reflection of an object with respect to the straight line $y=x$

x = y
y = x

The equivalent matrix equation for the point is:

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

Reflection of an object with respect to the straight line $y= -x$

$x = -y$
$y = -x$

The equivalent matrix equation for the point is:

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

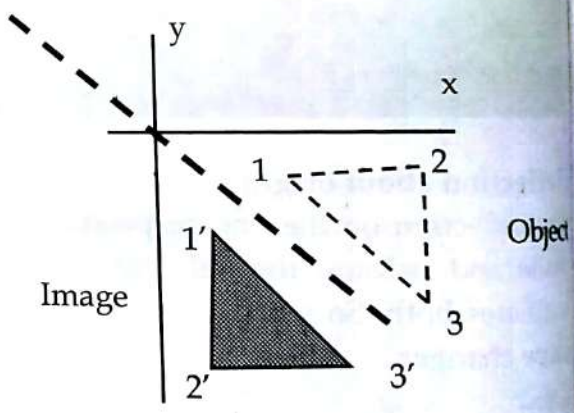


Figure 4.12: Reflection on line $y = -x$.

Shearing

A shear is a transformation that distorts the shape of an object along coordinate axes. It 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. Two common shearing transformations are those that shift coordinate x-values and those that shift y-values.

X-direction Shear

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

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

Which transforms $x' = x + sh_x.y$ and $y' = y$

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

For example: sh_x=2

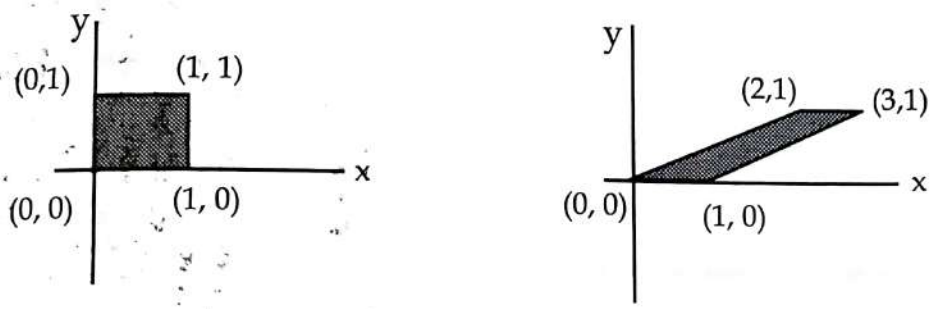
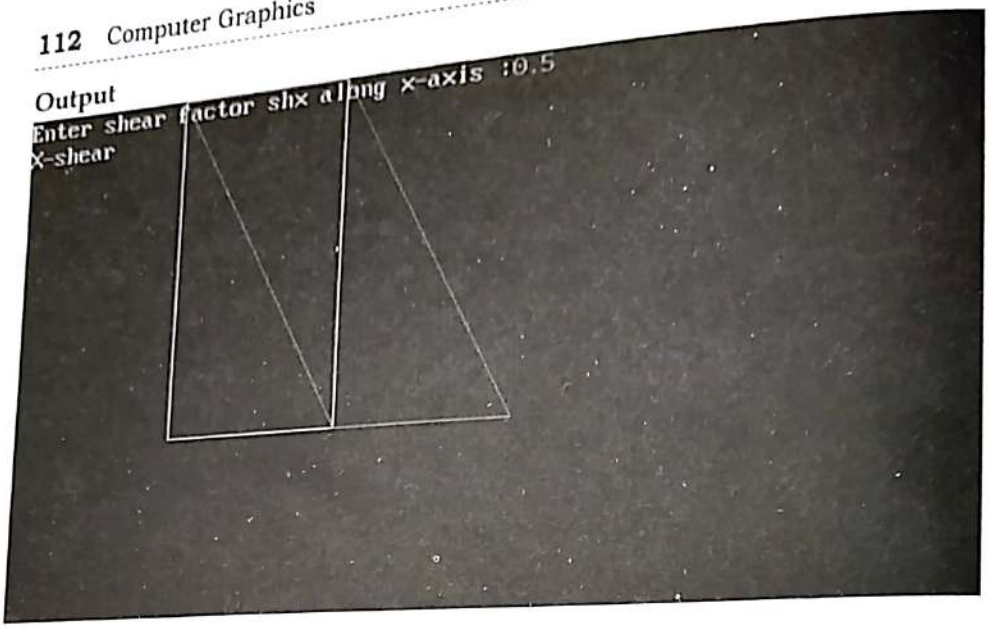


Figure 4.13: share on x direction having sh_x=2

Lab problem 4.6: Program for X-shear

```
#include<stdio.h>
#include<conio.h>
#include<dos.h>
#include<graphics.h>
void main()
{
        int gd=DETECT,gm;
        float shx,shy;
        initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
        printf("Enter shear factor shx along x-axis :");
        scanf("%f", &shx);
        line(100,0,200,0);
        line(200,0,200,200);
        line(200,200,100,200);
        line(100,200,100,0);
        printf("X-shear");
        setcolor(12);
        line((100+(0*shx)),0,(200+(0*shx)),0);
        line((200+(0*shx)),0,(200+(200*shx)),200);
        line((200+(200*shx)),200,(100+(200*shx)),200);
        line((100+(200*shx)),200,(100+(0*shx)),0);
        getch();
}
```

Output



Y-direction Shear

A y-direction shear relative to y-axis is produced by following transformation equations.

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

For example: $sh_y = 2$

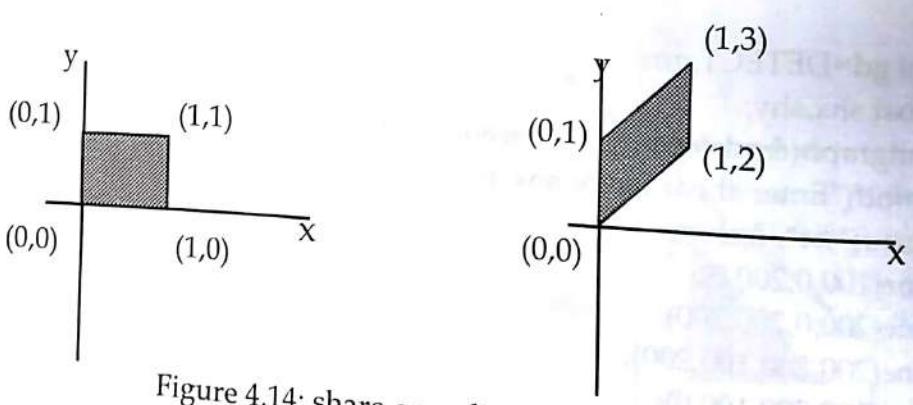


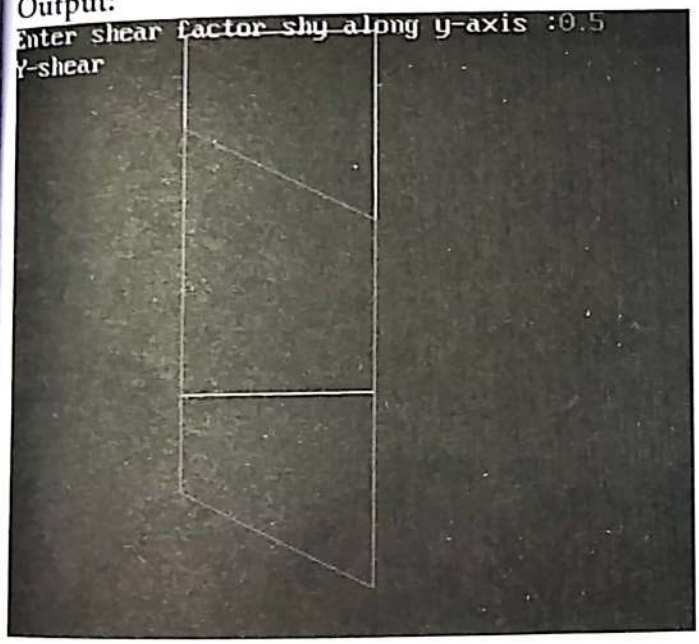
Figure 4.14: share on y direction having sh_y=2

Lab problem 4.7: Program for Y-shear

```
#include<stdio.h>
#include<conio.h>
#include<dos.h>
#include<graphics.h>
void main()
{
        int gd=DETECT,gm;
        float shx,shy;
```

```
initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
printf("Enter shear factor shy along y-axis :");
scanf("%f",&shy);
line(100,10,200,10);
line(200,10,200,200);
line(200,200,100,200);
line(100,200,100,10);
printf("Y-shear");
setcolor(12);
line(100,10+(shy*100),200,10+(shy*200));
line(200,10+(shy*200),200,200+(shy*200));
line(200,200+(shy*200),100,200+(shy*100));
line(100,200+(shy*100),100,10+(shy*100));
getch();
closegraph();
}
```

Output:



Two-Dimensional Viewing

Two-dimensional viewing is the mechanism for displaying views of a picture on an output device. A graphical package allows a user to specify which part of the defined picture is to be displayed and where that part is to be placed on the display device. Any convenient Cartesian coordinate system; referred to as the world coordinate reference frame can be used to define the picture. The picture parts within the selected areas are then mapped onto specified areas of the device co-ordinates. In computer graphics several coordinate systems (reference frame) are used to construct, and display a 2D scene or image.

Modeling coordinate system

- It is used to define coordinates that are used to construct the shape of individual parts (**objects**) of a 2D scene.

World Coordinate system

- Used to organize the individual objects **(points, lines, circles etc)** into a scene.
- A scene is made up of a collection of objects.
- These objects make up the "scene" or "world" that we want to view, and the coordinates that we use to define the scene are called world coordinates.

Viewing coordinate system

- Used to define particular view of a 2D scene. Translation, scaling, and rotation of the window will generate a different view of the scene. For a 2-D picture, a view is selected by specifying a subarea of the total picture area.

Normalized viewing coordinate system

- They are viewing coordinates between 0 and 1. NVC's are used to make the viewing process independent of the output device **(monitor, mobile, hard copy devices).**

Device Coordinate or Screen Coordinate system

- Used to define coordinates in an output device.
- Device coordinates are integers within the range (0, 0) to (x_{max}, y_{max}) for a particular output device.

What is 2D Viewing transformation?

The process of mapping of a part of world coordinate scene to device coordinate is called the 2D viewing transformation. That is, process of mapping the coordinates of the points and lines that form the picture into the appropriate coordinates on the device where the image is to be displayed is called 2D viewing transformation. Sometimes also referred to as the window-to-viewport transformation or the windowing transformation.

Transformations from world to device coordinates involve translation, rotation and scaling operations, as well as procedures for deleting those parts of the picture that are outside the limits of a selected display area(clipping).

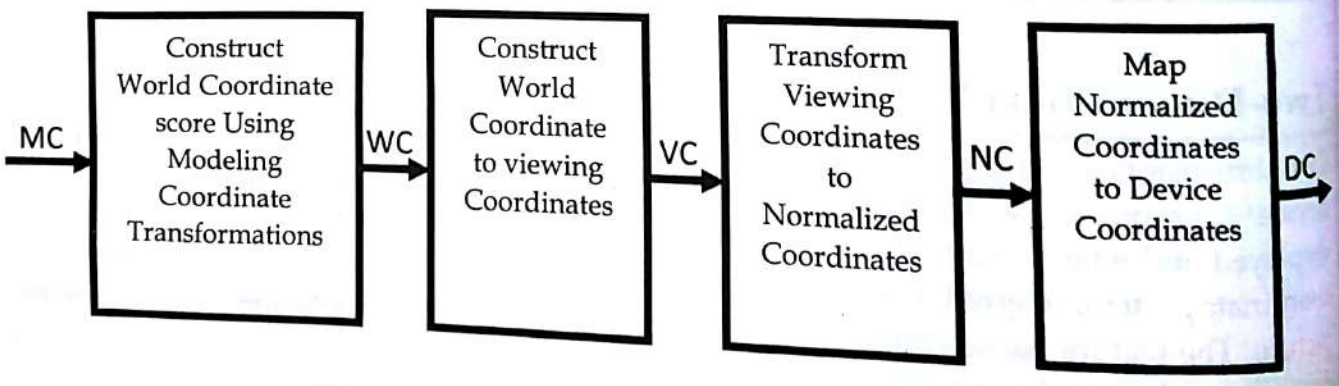


Figure 4.15: 2D viewing transformation pipeline

Window

A world-coordinate area selected for display is called a window. That is, window is the section of the 2D scene that is selected for viewing. The window defines what is to be viewed.

Viewport

An area on a display device to which a window is mapped is called a viewport. The viewport indicates where on an output device selected part will be displayed. Figure given below illustrates the mapping of a 2D picture that falls within a rectangular window onto a designated rectangular viewpoint.

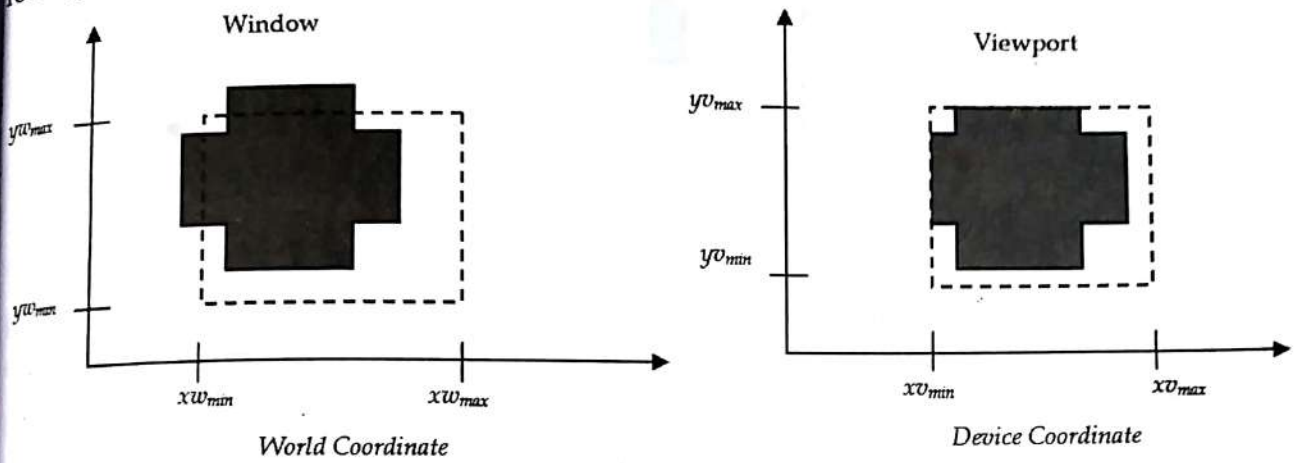


Figure 4.16: World and Viewport Coordinates, specified as rectangles aligned with the co-ordinate axes.

Window and viewport are often rectangular in standard positions, because it simplifies the transformation process and clipping process. Other shapes such as polygons, circles take longer time to process.

Window to viewport coordinate transformation

A window can be specified by four world coordinates: xw_{min}, xw_{max}, $yw_{min}$and yw_{max}. Similarly, a viewport can be described by four device coordinates: xv_{min}, xv_{max}, yv_{min} and y_{Vmax}.

Then, the window-to-viewport transformation can be explained in three steps as below:

1. Translate object along with window such that the lower left corner of the window is at the origin. That is, apply T $(-xw_{min}, -yw_{min})$.

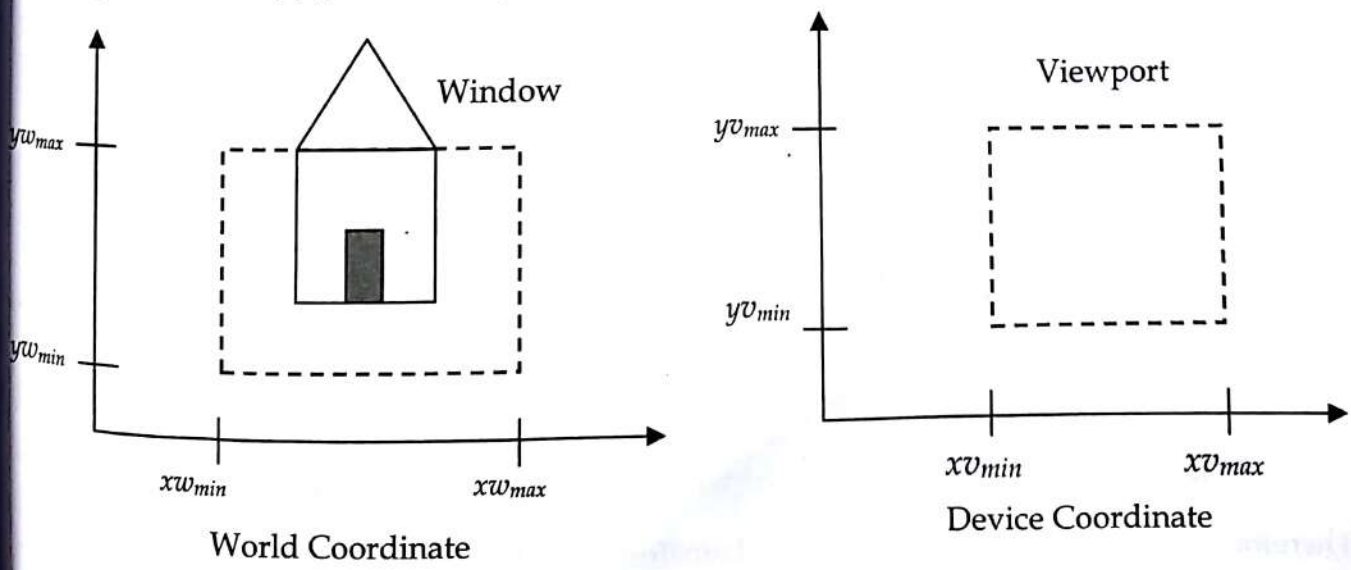


Figure 4.17: Window port and viewport before coordinate transformation

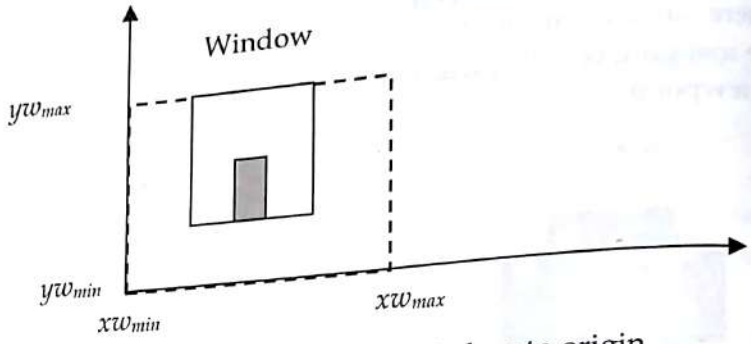


Figure 4.18: Translation of window to origin

2. Scale the object and the window such that window has the same dimension as that of viewport. Simply we can say that we are converting the object into image and window viewport. That is, apply S *(sx, sy)*.

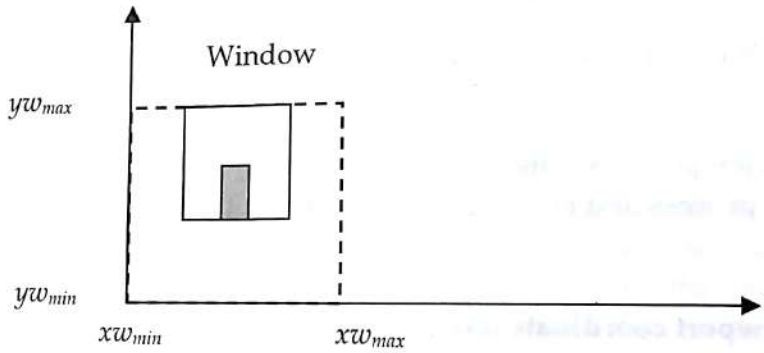


Figure 4.19: Scaling the window to the size of view port

3. Finally, apply another translation to move the viewport to its original position on the screen. That is, apply $T(xv_{min}, yv_{min})$.

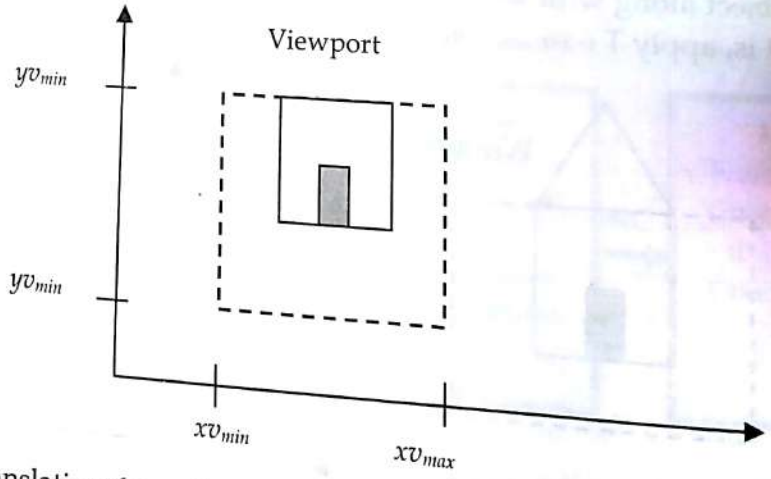


Figure 4.20: Translating the scaled window to the original position of viewport.

Therefore, net transformation (Composite Transformation),

$T_{WV} = T(xv_{min}, yv_{min}).\ S(S_x, S_y).T(-xw_{min}, -yw_{min})$ .....(i)

Here, Sx and Sy are scaling factors, where;

$$Sx = \frac{xv_{max} - xv_{min}}{xw_{max} - xw_{min}}$$

$$Sy = \frac{yv_{max} - yv_{min}}{yw_{max} - yw_{min}}$$

Now, writing in matrix form;

$$Twv = \begin{bmatrix} 1 & 0 & xv_{min} \\ 0 & 1 & yv_{min} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -yw_{min} \\ 0 & 1 & -yw_{min} \\ 0 & 0 & 1 \end{bmatrix}$$

This is required window-to-viewpoint transformation. Let $P(x, y)$ be the world coordinate point that is mapped onto the viewport point $P'(u, v)$, then we must have:

$P' = TWV.P$

Example 4.9: Find the normalization transformation window to viewport, with window, lower left corner at (1, 1) and upper right corner at (3, 5) onto a viewport with lower left corner at (0, 0) and upper right corner at (0.5, 0.5).

Solution:

Coordinates for window:

Xwmin= 1 Ywmin=1

Xwmax= 3 Ywmax=5

Coordinates of view port

Xvmin= 0 Yvmin=0

Xvmax= 0.5 Yvmax=0.5

We know that,

$$S_x = \frac{x_{v\max} - x_{v\min}}{x_{w\max} - x_{w\min}}$$

$$= \frac{0.5-0}{3-1} = 0.25$$

And, $$S_y = \frac{y_{v\max} - y_{v\min}}{y_{w\max} - y_{w\min}}$$

$$= \frac{0.5-0}{5-1} = 0.125$$

We know that, Composite transformation matrix is given as

$T_{WV} = T(x_{vmin}, y_{vmin}).\ S(S_x, S_y).T(-x_{wmin}, -y_{wmin})$

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

$$= \begin{bmatrix} 0.25 & 0 & -0.250 \\ 0 & 0.125 & -0.125 \\ 0 & 0 & 1 \end{bmatrix}$$

Clipping

The process of discarding (cutting off) those parts of a picture which are outside of a specif region (windows) is called **clipping**. Any procedure that identifies those parts of a picture t are either inside or outside of the specified region is called a **clipping algorithm**. The reg against which the clipping operation is performed is called a clip window.

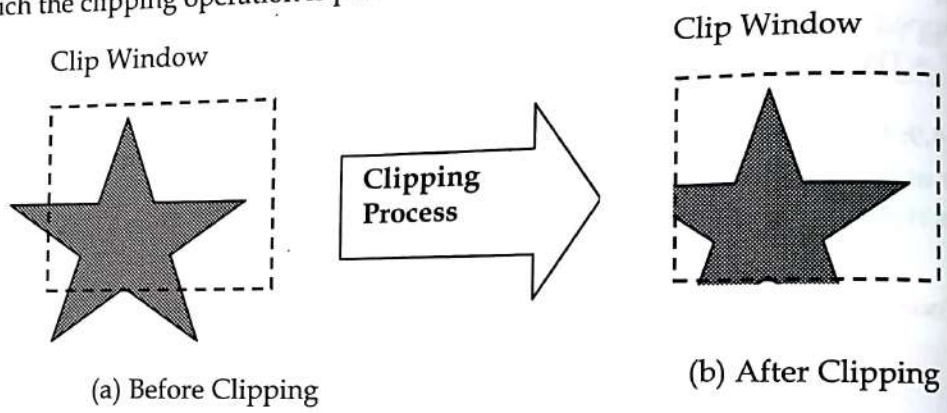


Figure 4.21: Clipping process (a) before clipping (b) after clipping

Why Clipping?

- Excludes unwanted graphics from the screen.
- Improves efficiency, as the computation dedicated to objects that appear off screen c be significantly reduced.

Applications of Clipping

- Extracting part of a defined scene for viewing.
- Identifying visible surfaces in three-dimensional views.
- Anti-aliasing line segments or object boundaries.
- Creating objects using solid-modeling procedures.
- Displaying a multi-window environment.
- Drawing and painting operations that allow parts of a picture to be selected copying, moving, erasing, or duplicating.

Point Clipping

Assuming a rectangular clip window, point clipping is easy. We save point iff:

$$x_{min} \le x \le x_{max}$$

$$y_{min} \le y \le y_{max}$$

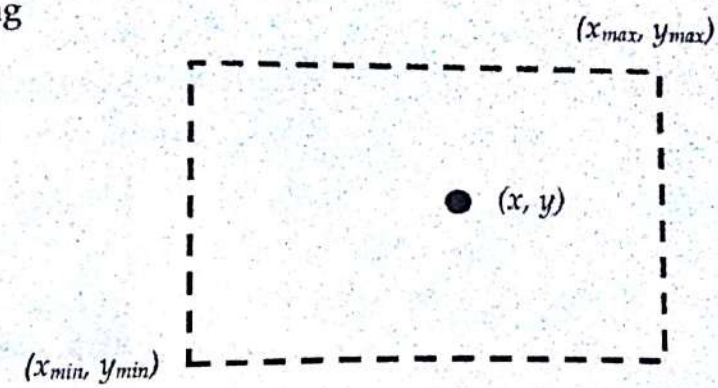


Figure 4.22: Point Clipping

Line Clipping

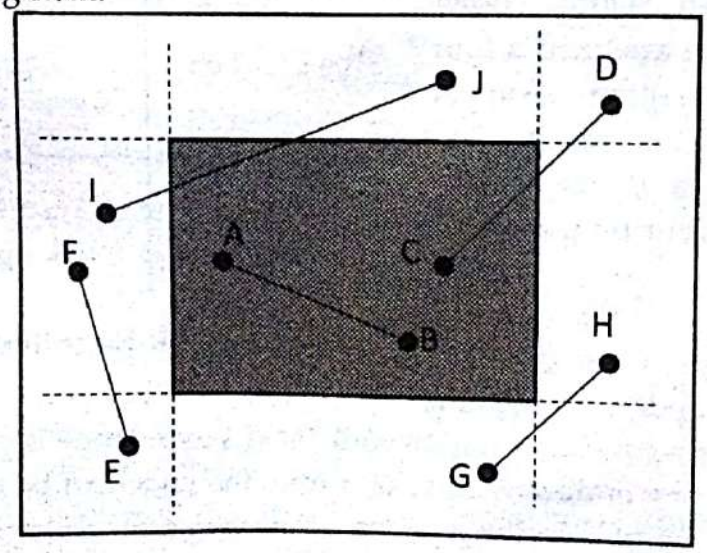
Lines that do not intersect the clipping window and are either completely inside the window or completely outside the window. Any particular line segment falls into one of the following clipping categories:

Visible

a. Both end points of the line segment lie inside the window (line AB).
b. No clipping necessary.

Non-Visible

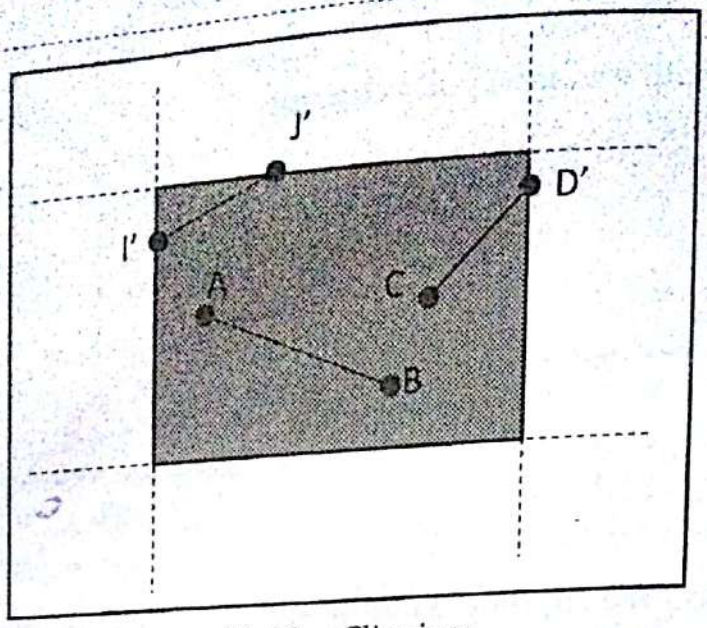
a. Line lies completely outside the clip window (lines EF, GH).
b. Discard the line segment.



(a) Before Clipping

Partially visible (clipping candidate)

a. A line is partially visible when a part of it lies within a window (lines CD, IJ).
b. In this case we need to compute intersection points and clip at the intersection points.



(b) After Clipping

Figure 4.23: Line Clipping

Cohen Sutherland line clipping algorithm

This is one of the oldest and most popular line-clipping algorithms. In this method, coordinate system is divided into nine regions. All regions have their associated region codes. Every line endpoint is assigned a four digit binary code (region code or outcodes). Each bit in the code is set to either a 1(true) or a 0(false). Each region is assigned a four bit pattern as shown in figure.

	Top	
1001	1000	1010
0001 Left	0000 Window	0010 Right
0101	0100 Bottom	0110

Figure 4.24: Region code according to TBRL

The numbers in fig above are called as outcodes or region codes. Any point inside the clipping window has a region code 0000. An outcode is computed for each of the two points of a line. For any endpoint *(x, y)* of a line, the code can be determined that identifies in which region the endpoint lies. The outcode's bits are set according to the following conditions:

- First bit is set to 1 if point lies towards left of window
- Second bit is set to 1 if point lies towards right of window
- Third bit is set to 1 if point lies bottom/below of window
- Fourth bit is set to 1 if point lies top/ above of window

Algorithm

Step 1: Given a line segment with endpoint $P_1=(x_1,y_1)$ and $P_2=(x_2, y_2)$

Step 2: Compute the 4-bit outcodes for the two endpoints of the line segment

Step 3: If outcode of P_1=outcode of P_2= 0000 ($P_1 || P_2$=0000) then

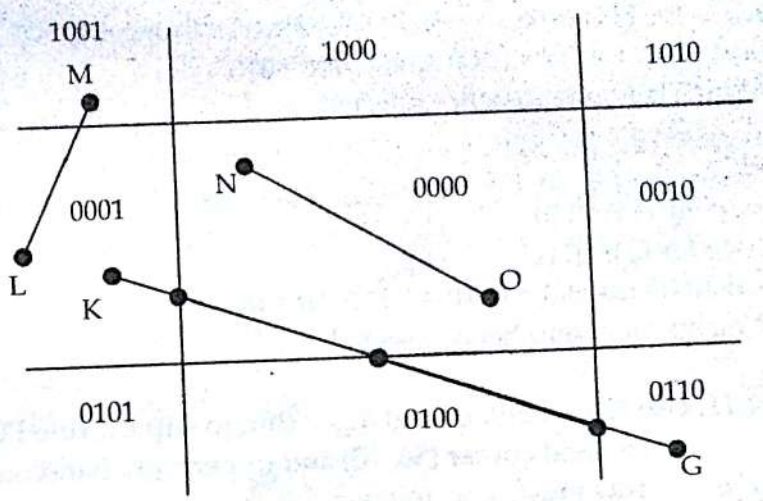
Line lies completely inside the window, so draw the line segment.

Step 4: else if outcode of P_1&& outcode of $P_2 \neq$ 0000 (P_1&&$P_2 \neq$0000) then

Line lies completely outside the window, so discard the line segment.

Step 5: else if outcode of P_1&& outcode of P_2= 0000 (P_1&&P_2=0000) then
Compute the intersection of the line segment with window boundary, and discard the portion of the segment that falls completely outside the window. Assign a new four-bit code to the intersection and repeat until either 3 or 4 above are satisfied.

Example 4.10: Clip the following lines using Cohen Sutherland line clipping algorithm.



Solution

For line LM:
The outcode for L is 0001
And outcode for M is 1001
The Logical AND for LM =0001 AND 1001 = 0001
Which is non zero hence it may be rejected.

Also for line ON:
The outcode for O is 0000
And outcode for N is 0000
The Logical OR for ON = 0000 OR 0000 = 0000
It is inside the window and is accepted.

Again for line KG: The outcode for K is 0001
And outcode for G is 0110

The Logical AND for KG = 0001 AND 0110 = 0000
This is zero. This means that this line crosses more lines which contain the clipping boundary.

Here line segment KG is broken into KJ, JI, JH and HG and again each segment is tested as shown below:

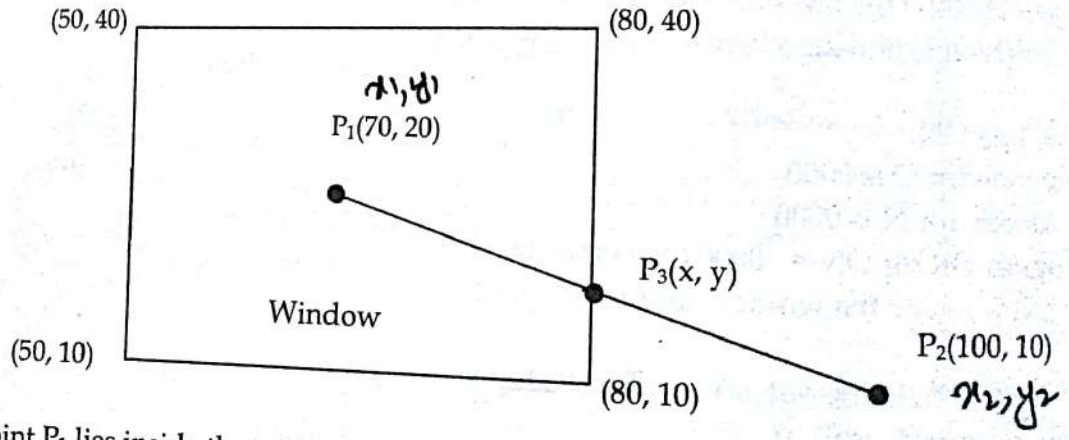
For line segment KJ
The outcode for K is 0001
And outcode for J is 0110
The Logical AND for KG = 0001 AND 0001 =0001.
Which is non-zero hence rejected

For line JI:
The outcode for J is 0000
And outcode for I is 0000
The Logical AND for JI = 0000 AND 0000 = 0000
Which is zero hence both ends points are inside the window. So it is accepted.

For line IH:
The outcode for I is 0100
And outcode for H is 0100
The Logical AND for IH = 0100 AND 0100 =0100.
Which is non-zero hence rejected

For line HG:
The outcode for H is 0110
And outcode for G is 0110
The Logical AND for HG = 0110 AND 0110 = 0110
Which is non-zero hence rejected

Example 4.11: Use Cohen-Sutherland algorithm to clip the line $P_1(70, 20)$ and $P_2(100, 10)$ against a window lower left hand corner (50, 10) and upper right hand corner (80, 40).
Solution: The window must be as follows:



Since the point P_1 lies inside the window hence its outcode is 0000
Similarly the outcode of P_2=0010
The outcode for P_1 is 0000
And outcode for P_2 is 0010
The Logical AND for P_1P_2 = 0000 AND 0010 = 0000.
Which is zero hence line is partially visible.

The slope of line $P_1P_2 = \frac{y_2 - y_1}{x_2 - x1} = \frac{10-20}{100-70} = \frac{-10}{30} = \frac{-1}{3}$

From the above fig, $P_3(x, y) = P_3(80, y)$ here we need to find the value of y

Now slope of the line $P_2P_3 = \frac{y - y_2}{x - x_2} = \frac{y-10}{80-100} = \frac{y-10}{-20}$

Since slope of P_1P_2=slope of P_2P_3

Hence $\frac{-1}{3}=\frac{y-10}{-20}$

Or, $y-10=\frac{20}{3}$

Or, $y=10+\frac{20}{3}=16.667$

Now the parts P_2P_3 of line P_1P_2 is clipped as it is outside the window

Hence $P_3(x, y) = P_3(80, 16.667)$

Example 4.12: Find the clipping coordinates for a line AB where A=(10, 10) and B(60, 30), against window with (x_{wmin}, y_{wmin}) = (15, 15) and (x_{wmax}, y_{wmax})=(25, 25) using Liang-Barsky Line clipping Algorithm.

Solution:

Here,

$x_1=10$	$x_{wmin} = 15$	
$y_1=10$	$y_{wmin} = 15$	
$x_2=60$	$x_{wmax} = 25$	
$y2=30$	$y_{wmax} = 25$	
$p_1= -50$	$q_1 = -5$	$q_1/p_1= 0.1$
$p2= 50$	$q_2 = 15$	$q_2/p_2= 0.3$
$p_3= -20$	$q_3 = -5$	$q_3/p_3= 0.25$
$p_4= 20$	$q_4 = 15$	$q_4/p_4= 0.75$

t_1= max (0.25, 0.1) = 0.25 since for these values p <0.

t_2= min (0.3, 0.75) = 0.3 since for these values p >0.

Here, $t_1 < t_2$ and the endpoints of clipped line are:

$xx_1=x_1+t_1\Delta x$

$= 10 + 0.25 \times 50$

$= 22.5$

$xx_2 =x_1+t_2\Delta x$

$= 10 + 0.3 \times 50$

$= 25$

$yy_1= y_1+t_1\Delta y$

$= 10 + 0.25 \times 20$

$= 15$

$yy_2 = y_1+t_2\Delta y$

$= 10 + 0.3 \times 20$

$= 16$

Liang - Barsky Line Clipping

Based on analysis of parametric equation of a line segment, faster line clippers have been developed, which can be written in the form:

$$x = x_1 + u\,\Delta x$$

$$y = y_1 + u\,\Delta y$$

$$where, 0 \le u \le 1.$$

Where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$. Using these parametric equations, Cryus and Beck developed an algorithm that is generally more efficient than the Cohen-Sutherland algorithm. Later, Liang and Barsky independently devised an even faster parametric line clipping algorithm. Following the Liang-Barsky approach, we first write the point clipping in a parametric way:

$$x_{\min} \le x_1 + u\,\Delta x \le x_{\max}$$

$$y_{\min} \le y_1 + u\,\Delta y \le y_{\max}$$

Each of these four inequalities can be expressed as

$$up_k \le q_k,\ for\ k = 1, 2, 3, 4$$

Where parameters p and q are defined as:

$$p_1 = -\Delta x \qquad q_1 = x_1 - xw_{\min}$$

$$p_2 = \Delta x \qquad q_2 = xw_{\max} - x_1$$

$$p_3 = -\Delta y \qquad q_3 = y_1 - yw_{\min}$$

$$p_4 = \Delta y \qquad q_4 = yw_{\max} - y_1$$

Any line that is parallel to one of the clipping boundaries have $p_k = 0$ for values of k corresponding to boundary k=1, 2, 3, 4 correspond to left, right, bottom and top boundaries. For values of k, find $q_k < 0$, the line is completely outside the boundary. If $q_k \ge 0$, the line is inside the parallel clipping boundary.

When $p_k < 0$ the infinite extension of line proceeds from outside to inside of the infinite extension of this clipping boundary. If $p_k > 0$, the line proceeds from inside to outside, for non-zero value of p_k calculate the value of u that corresponds to the point where the infinitely extended line intersect the extension of boundary k as:

$$u = \frac{q_k}{p_k}$$

For each line, calculate values for parameters u1and u2 that define the part of line that lies within the clip rectangle. The value of u1 is determined by looking at the rectangle edges for which the line proceeds from outside to the inside (p < 0). For these edges we calculate

$$r_k = \frac{q_k}{p_k}$$

The value of u_1 is taken as largest of set consisting of 0 and various values of r. The value of u_2 is determined by examining the boundaries for which lines proceeds from inside to outside (P > 0). A value of r_k is calculated for each of these boundaries and value of u_2 is the minimum of the set consisting of 1 and the calculated r values. If $u_1 > u_2$, the line is completely outside the clip

window and it can be rejected. Otherwise, the end points of the clipped line are calculated from the two values of parameter u.

Line intersection parameters are initialized to values $u_1=0$ and $u_2=1$. For each clipping boundary, the appropriate values for P and q are calculated and used by function Clip Test to determine whether the line can be rejected or whether the intersection parameter can be adjusted.

When $p < 0$, the parameter r is used to update u1. When $p > 0$, the parameter r is used to update u_2. If updating u_1 or u_2 results in $u_1 > u_2$ reject the line, when $p = 0$ and $q < 0$, discard the line, it is parallel to and outside the boundary. If the line has not been rejected after all four value of p and q have been tested, the end points of clipped lines are determined from values of u_1 and u_2.

In general, the Liang-Barsky algorithm is more efficient than the Cohen Sutherland algorithm, since intersection calculations are reduced. Each update of parameters u_1 and u_2 requires only one division; and window intersections of the line are computed only once, when the final values of u_1 and u_2 have computed. In contrast, the Cohen-Sutherland algorithm can repeatedly calculate intersections along a line path, even though the line may be completely outside the clip window, and, each intersection calculation requires both a division and a multiplication. Both the Cohen Sutherland and the Liang Barsky algorithms can be extended to three-dimensional clipping.

Algorithm

1. Set $t_{min}=0$ and $t_{max}=1$
2. Calculate the values of tL, tR, tT, and tB (tvalues).
   - if $t < t_{min}$ or $t > t_{max}$ ignore it and go to the next edge.
   - otherwise classify the tvalue as entering or exiting value (using inner product to classify)
   - if **t** is entering value set $t_{min} = t$; if **t** is exiting value set $t_{max}=t$
3. If $t_{min} < t_{max}$ then draw a line from $(x1 + dx*t_{min}, y1 + dy*t_{min})$ to $(x1 + dx*t_{max}, y1 + dy*t_{max})$
4. If the line crosses over the window, you will see $(x1 + dx*t_{min}, y1 + dy*t_{min})$ and $(x1 + dx*t_{max}, y1 + dy*t_{max})$ are intersection between line and edge.

Polygon clipping

A polygon can be defined as a geometric object "consisting of a number of points (called vertices) and an equal number of line segments (called sides or edges)". Polygon clipping is defined as the process of removing those parts of a polygon that lie outside of a clipping window.

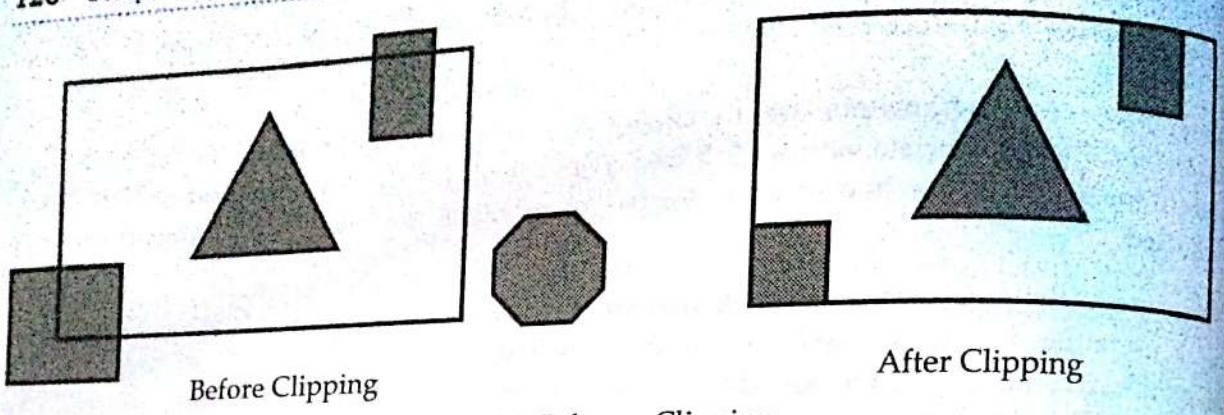


Figure 4.24: Polygon Clipping

Sutherland-Hodgeman Polygon Clipping Algorithm

Sutherland and Hodgeman's polygon-clipping algorithm uses a divide-and-conquer strategy. It clips a given polygon successively against the edges of the given clip window. It starts with the initial set of polygon vertices, first clips the polygon against the left boundary of the window. Then successively against the right boundary, bottom boundary, and finally against the top boundary, as shown in figure

Basic approach

Clip a polygon against each clip window edge i.e., left, right, bottom, and top.

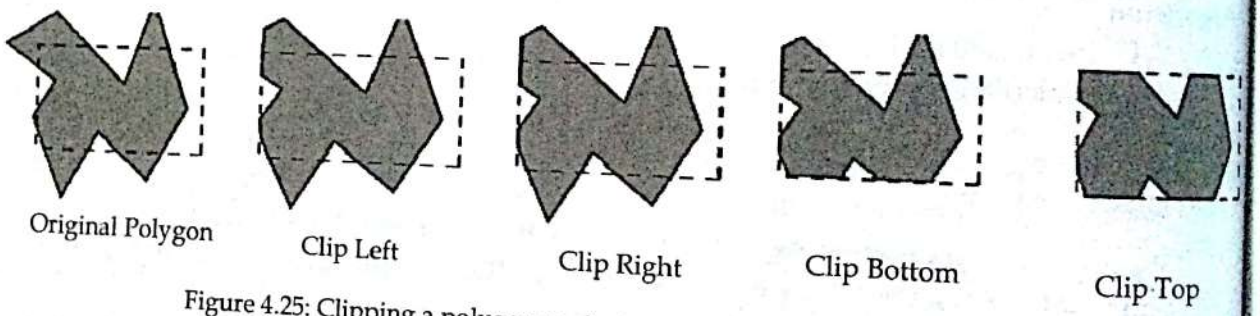


Figure 4.25: Clipping a polygon against successive window boundaries

Algorithm

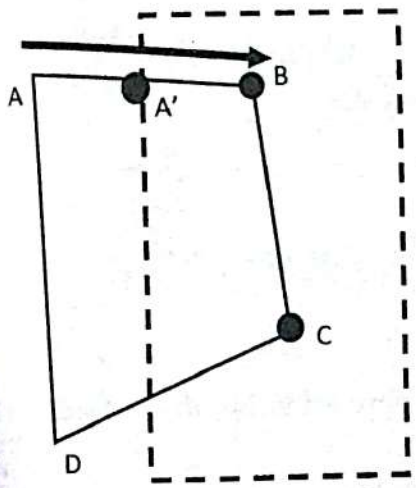
For each clip window boundary perform the following steps:

1. Construct an input vertex list ($v_0, v_1 \ldots v_n$) where $v_0 = v_n$
2. Create empty output vertex list
3. For each pair of adjacent vertices vi and v_i+1 perform the following inside-outside test:
   a. If out-in (v_i is outside the window boundary and v_i+1 is inside),
      i. add intersection point vi′ and vi+1 to the output vertex list.
   b. If in-in (both v_i, v_i+1 are inside the window boundary),
      i. add v_i+1 to output vertex list
   c. If in-out (v_i is inside the window boundary and v_i+1 is outside),
      i. add intersection point v_i' to output vertex list
   d. If out-out (both v_i, v_i+1 are outside the window boundary),
      i. add nothing to output vertex list
4. Stop

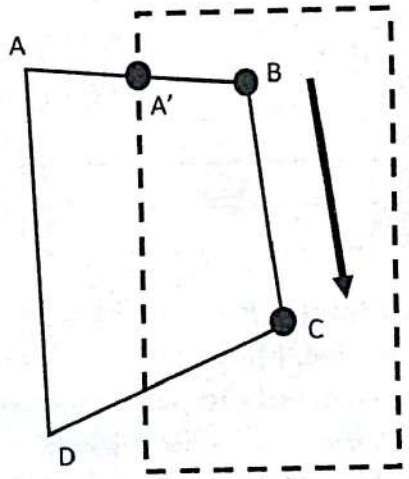
Four Cases of polygon clipping against one edge

S. No.	Case	Output
1	Wholly inside visible region (in – in)	save endpoint
2	Exit visible region (in - out)	save the intersection
3	Wholly outside visible region (out - out)	save nothing
4	Enter visible region (out – in)	save intersection and endpoint

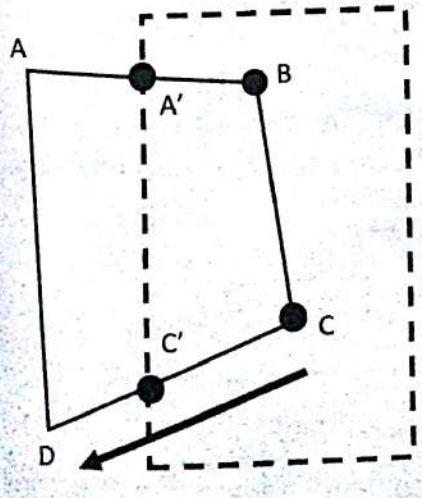
Example 4.13: Clip the given polygon using Sutherland-Hodgeman Polygon clipping Algorithm.



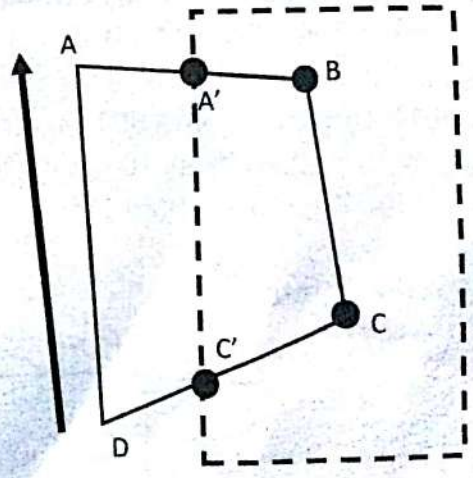
Out → In
Save new clip vertex
and ending vertex



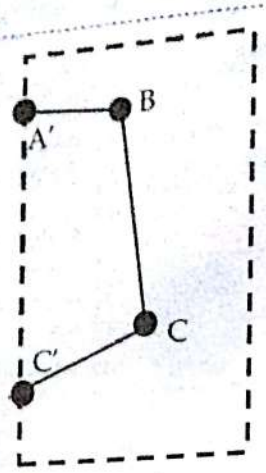
In→ In
Save ending vertex



In→ Out
Save new clip vertex



Out → Out
Save nothing



Clipped Polygon

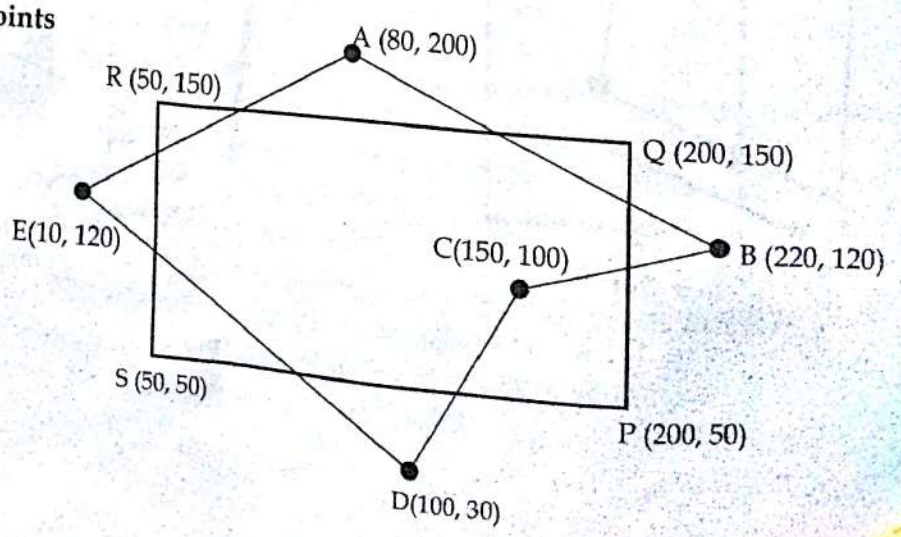
Vertex	Case	Output Vertex
AB	Out → In	A′B
BC	In → In	C
CD	In → Out	C′
DA	Out → Out	

Input vertex list: {A, B, C, D}
Output vertex list: { }
For vertices A, B, **out - in** pair so intersection point A′ and Bare added to the output vertex list
Therefore, output vertex list = {A′, B}
For vertices B, C, **in-in**pair so only v2 is added to the output vertex list.
Now, output vertex list = {A′, B, C}
For vertices C, D, **in-out**pair so only intersection point C′ is added to the output vertex list.
Now, output vertex list = { A′, B, C, C′}
For vertices D, A, **out-out**pair so nothing is added to the output vertex list.
Finally, output vertex list = {A′, B, C, C′}

Example 4.14: Clip polygon ABCD against window PQRS. The Co-ordinates of the polygon are A(80, 200), B(220, 120), C(150, 100), D(100, 30), E(10, 120). Co-ordinates of the window are P(200, 50), Q(200, 150), R (50, 150), S(50, 50).

Solution:

Step 1: Plot the points



Step 2: Clipping against left edge of the window (Left Clipping)

Vertex	Case	Output Vertex	Remarks
AB	In → In	B	
BC	In → In	C	
CD	In → In	D	
DE	In → Out	D′	New Vertex
EA	Out → In	E′A	New Vertex

Step 3: Right Clipping

Vertex	Case	Output Vertex	Remarks
AB	In → Out	A′	New Vertex
BC	Out → In	B′C	New Vertex
CD	In → In	D	
DD′	In → In	D′	
D′E′	In → In	E′	
E′A	In → In	A	

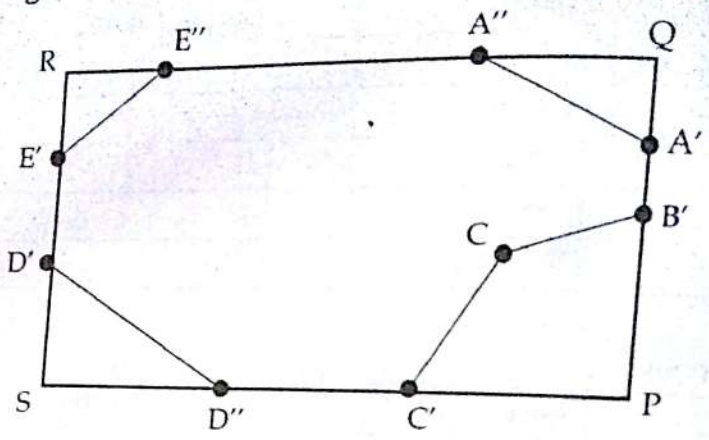
Step 4: Bottom Clipping

Vertex	Case	Output Vertex	Remarks
AA′	In → In	A′	
A′B′	In → In	B′	
B′C	In → In	C	
CD	In → Out	C′	New Vertex
DD′	Out → In	D″D′	New Vertex
D′E′	In → In	E′	
E′A	In → In	A	

Step 5: Top Clipping

Vertex	Case	Output Vertex	Remarks
AA′	Out → In	A″A′	New Vertex
A′B′	In → In	B′	
B′C	In → In	C	
CC′	In → In	C′	
C′D″	In → In	D″	
D″D′	In → In	D′	
D′E′	In → In	E′	
E′A	In → Out	E″	New Vertex

Step 6: After Clipping



Exercise

1. Explain in detail on any two basic two dimensional geometric transformations.
2. Explain 2D Translation, Rotation, Mirror and Shear with practical applications.
3. Prove that two successive rotations are additive. Obtain a transformation matrix for rotating an object about a specified pivot point.
4. Prove that two successive scaling are multiplicative. Also, prove that reflection is equal to rotation by 180°.
5. The reflection along the line y=x is equivalent to the reflection on the x axis followed by counter clockwise rotation by θ degree. Find the value of θ.
6. What do you mean by homogeneous coordinates? Explain it with equation and practical application.
7. Why homogenous coordinate are used for transformation computations in computer graphics? Explain.
8. Derive the matrix representation of composite transformation.
9. What do you mean by viewing? Enumerate the differences between a window and view port.
10. Derive the window to viewport transformation coefficient matrix.Write the applications of window to view port transformation.
11. What are the stages involved in 2D viewing transformation pipeline? Explain briefly about each stage.
12. Illustrate the window to view port transformation with an example.
13. What do you mean by clipping? How will you clip a point? Explain.
14. Write down the principle of two dimensional clipping.
15. Compare interior and exterior clipping.
16. Explain the line clipping algorithm with an example.
17. What do you mean by line clipping? Explain the procedures for line clipping.

How are lines grouped into visible, invisible and partially visible categories in 2D clipping? Explain.

Explain in detail the Cohen-Sutherland line clipping algorithm with an example.

Explain Sutherland Hodgeman polygon clipping algorithm, with suitable example.

Numerical

1. Suppose we have a quadrilateral ABCD with vertices A(3,1), B(-7,-4), C(5,-8) and D(3,0). What is the coordinates of quadrilateral if it is to be doubled in size about origin. Then translate obtained coordinates with (-4, 3).
2. Rotate the triangle having co-ordinates (1, 2), (2, 3) (4, 5) by 60° about origin. Then translate with T(9, 0), and then scale about origin with scale factor S(2, 4).
3. Rotate a triangle [(4, 6), (2, 2), (6, 2)] about the vertex (4, 6) by 180° CCW and find the new vertices.
4. Rotate a triangle A(5, 6), B(6, 2) and C(4, 1) by 45° about an arbitrary pivot point (3, 3).
5. Consider a triangle A(0, 0), B(1, 1), C(5, 2) and the triangle has to be rotated by an angle of 45° about the point P(-1,-1). Calculate the vertices of triangle A'B'C'.
6. Rotate the △ABC by 45° clock wise about origin and scale it by (2, 3) about origin, where A(7,15), B(5,8), and C (10,10).
7. Transform a square with corners A(2, 2), B(4, 2), C(2, 4), D(4, 4) into a rectangle whose breadth is half of the breadth of square and length is double of it.
8. A triangle A(4, 5), B(2, 1), and C(6, 1) is required to enlarge with scaling factors S(2, 2), and S(2, 0.5). Find the new vertices of the triangle.
9. Perform the scaling transformation to the triangle with vertices A(6,9), B(10,5), C(4,3) with scaling factors Sx= 3 and Sy = 2.
10. Rotate the point (2, -4) about the origin 30° in anti-clockwise direction.
11. A triangle A(4, 5), B(2, 1), and C(6, 1) is required to enlarge twice its initial size about the fixed point (4, 2). Find the final coordinates.
12. A Square with vertices (0, 0), (2, 0), (2, 2) and (0, 2) is scaled 3 units in x as well as y direction about the fixed point which is centre of square (2, 2). Find the co-ordinates of vertices of new square.
13. Find the co-ordinate of a image of a quadrilateral ABCD with vertices A(6, 2), B(10, 2), C(11, 4) and D(5, 4) under the combined transformation of the reflection on x-axis then on the y-axis which is then rotated in 60° about origin.
14. Find the co-ordinate of a image of a triangle ABC with vertices A(2,0), B(3,1) and C(1,1) which is rotated about the arbitrary point (1,1) through 30°.
15. Consider a triangle ABC with vertices A(3,4), B(7, 4), and C(5, 8). What is the transformation if the triangle is to be doubled in size keeping A(3, 4) fixed? Obtain new vertices of the triangle.
16. Reflect a triangle ABC with vertices A(1, 9), B(3, 3), C(1, 6) about the line y = x+2.
17. Translate the given triangle A(1, 3), B(-2, -1), C(6, -2) with translation vector T(-2, -6) and then reflect with y= x axis.

18. If the window coordinates are (10, 10), (20, 10), (20, 20) and (10, 20) and the view port (0.5, 0.5), (1,0.5), (1, 1) and (0.5, 1). Calculate the composite transformation matrix.
19. If window is defined as A(10,20), B(20, 20), C(20,10) and D(10,10). Find the visible portion of the line P(15, 15) and Q(15, 5).
20. Clip the line XY with endpoints X(3, 5), Y(9, 3) against clipping boundary MNOP; M(3, 3) N(3, 8), O(8, 8), P(8, 3) using Cohen-Sutherland line clipping algorithm.
21. Use Cohen Sutherland algorithm to clip the line for the following dimensions. Line endpoints (15, 45) and (25,15) Window: Lower left: (10, 20) and Upper right: (30, 40)
22. Clip the polygon ABC against the window PQRS. Coordinates of polygon are A(80,80) B(120,60) and C(75,60) and Coordinates of window are P (50,50), Q(100,50), R(100,100) and S(50,100).
23. A polygon has four vertices located at A(20, 10) B(60, 10) C(60, 30) D(20, 30). Calculate vertices after applying a transformation matrix to double the size of polygon with point located on the same place.
24. Clip a quadrilateral ABCD with coordinates (10, 18) (22, 18) (34, 27) and (10, 37) against window QRST with coordinate (5, 15) (30, 15) (30, 25) and (5, 25) using Cohen Sutherland algorithm.
25. Given a clipping window P(0, 0), Q(30, 0), R(30, 20) and S(0,20) use the Cohen Sutherland algorithm to determine the visible portion of line joint A(10, 30) and B (40, 0).
26. Using Liang Barsky line clipping algorithm find the clipping co-ordinates of line segment with end co-ordinates A(-10, 50) and B(30, 80) against window (xwmin, ywmin) = (-30, 10) and (xwmax, ywmax)=(20, 60).

❑❑❑

Chapter

5

Three-Dimensional Geometric Transformation

STRUCTURE

5.0 Introduction

5.1 Three-Dimensional translation, rotation, scaling, reflection and shearing

5.2 Three-Dimensional Viewing:

- Viewing pipeline
- World to screen viewing transformation.
- Projection Concepts (Orthographic, parallel, perspective projections)

5.0 Introduction

The process of moving points in space is called three-dimensional transformations. Methods for geometric transformations in three dimensions are extended from two-dimensional methods by including considerations for the z coordinate.

In three-dimensional space, we can now select any spatial orientation for the rotation axis. Some graphics packages handle three-dimensional rotation as a composite of three rotations, one for each of the three Cartesian axes.

Matrix representation of 3D Transformation

Represent a 3D transformation by a 3×3 matrix

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

Apply the transformation to a point

$$\begin{aligned} x' &= ax + by + cz \\ y' &= dx + ey + fz \\ z' &= gz + hy + iz \end{aligned} \Longrightarrow \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Image = T.M. × Object

If object and image are represented in row matrix then:

Image = Object × T.M.

Transformations can be combined as in two-dimension.

Homogeneous co-ordinate representation of 3D Transformation

Homogeneous co-ordinate representation of 3D Transformation has same idea as two dimensional transformations. In homogeneous coordinate representation each 3D point (x, y, z) is represented as homogeneous coordinate by four points (x_h, y_h, z_h, h), where $x = x_h/h$, $y = y_h/h$ and $z = z_h/h$. The reason why this coordinate system is called 'homogeneous' is because it is possible to transform functions such as $f(x, y, z)$ into the form $f(x/h, y/h, z/h)$ without disturbing the degree of the curve. In the field of projective geometry it is a very powerful concept.

- (x_h, y_h, z_h, h) represents a point at location $(x_h/h, y_h/h, z/h)$
- $(x_h, y_h, z_h, 0)$ represents a point at infinity
- $(0, 0, 0, 0)$ is not allowed

A three-dimensional position, expressed in homogenous coordinates, is represented as a four-element column vector. Thus, each geometric transformation operator is now a 4×4 matrix, which pre multiplies a coordinate column vector.

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ h \end{bmatrix}$$

Three-Dimensional translation, rotation, scaling, reflection and shearing

Translation

Translation is used to move a point, or a set of points, linearly in space. Since now we are talking about 3D, therefore each point has 3 coordinates i.e. x, y and z. Similarly, the translation distances can also be specified in any of the 3 dimensions.

These Translation Distances are given by t_x, t_y and t_z.

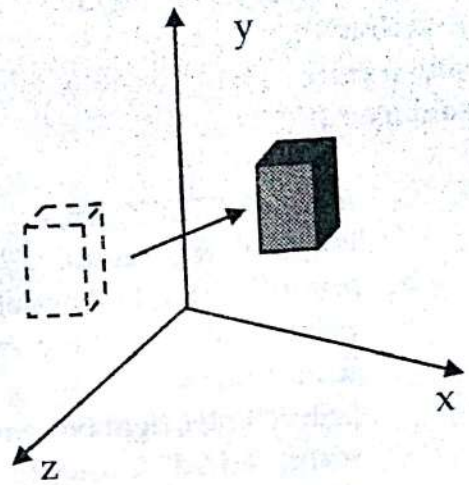


Figure 5.1: 3D Translation

For any point $P(x,y,z)$ after translation we have $P'(x',y',z')$ where

$$x' = x + t_x,$$
$$y' = y + t_y,$$
$$z' = z + t_z$$

And (t_x, t_y, t_z) is Translation vector.

Now this can be expressed as a single matrix equation.

$$P' = P + T$$

Where,

$$P = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \quad P' = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}, \text{ and } T = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

Homogeneous Coordinates

Analogous to their 2D Counterpart, the homogeneous coordinates for 3D translation can be expressed as:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Abbreviated as:

$$P' = T(t_x, t_y, t_z).\ P$$

On solving the RHS of the matrix equation, we get:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

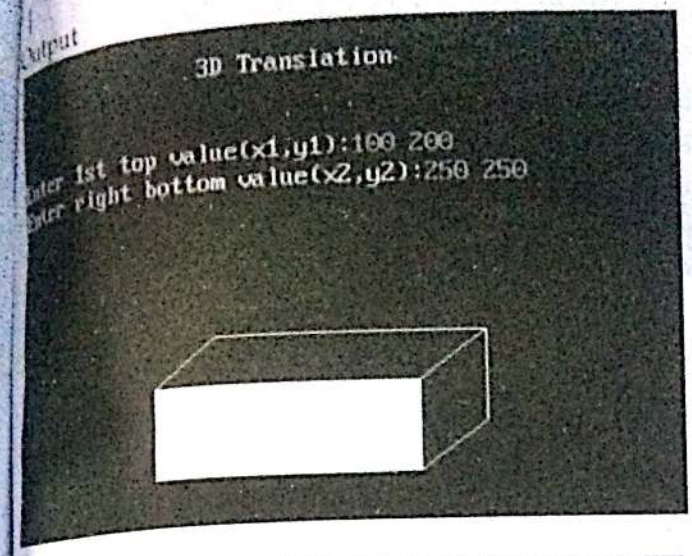
This shows that each of the 3 coordinates gets translated by the corresponding translation distance.

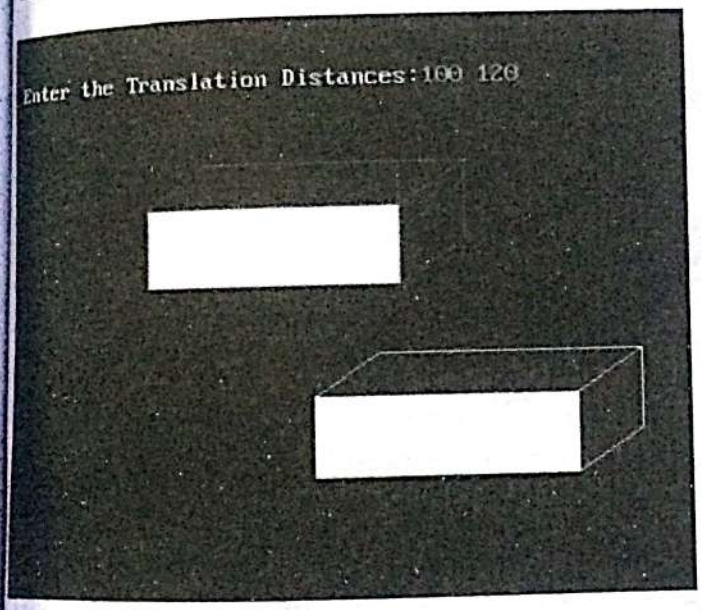
Lab problem 5.1: Program for 3D translation

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<process.h>
#include<graphics.h>
int x1,x2,y1,y2,mx,my,depth;
void draw();
void trans();
void main()
{
        int gd=DETECT, gm, c;
        initgraph(&gd, &gm, "C:\\TurboC3\\BGI");
        printf("\n\t\t3D Translation\n\n");
        printf("\nEnter 1st top value(x1,y1):");
        scanf("%d%d",&x1,&y1);
        printf("Enter right bottom value(x2,y2):");
        scanf("%d%d",&x2,&y2);
        depth=(x2-x1)/4;
        mx=(x1+x2)/2;
        my=(y1+y2)/2;
        draw();
        getch();
        cleardevice();
        trans();
        getch();

}
void draw()
{
  bar3d(x1,y1,x2,y2,depth,1);
}
void trans()
{

        int a1,a2,b1,b2,dep,x,y;
        printf("\n Enter the Translation Distances:");
        scanf("%d%d", &x,&y);
        a1=x1+x;
        a2=x2+x;
        b1=y1+y;
        b2=y2+y;
        dep=(a2-a1)/4;
        bar3d(a1,b1,a2,b2,dep,1);
        setcolor(5);
```

```
draw();
```

Output





Rotation

Rotation is the process of moving a point in space in a non-linear manner. More particularly, it involves moving the point from one position on a sphere whose center is at the origin to another position on the sphere. Why would you want to do something like this? As we will show in later section, allowing the point of view to move around is only an illusion – projection requires that the POV be at the origin. When the user thinks the POV is moving, you are actually translating all your points in the opposite direction; and when the user thinks the POV is looking down a new vector, you are actually rotating all the points in the opposite direction; and when the user thinks the POV is looking down a new vector, you are actually rotating all the points in the opposite direction.

3D rotation is not same as 2D rotation. In 3D rotation, we have to specify the angle of rotation along with the axis of rotation. We can perform 3D coordinate axes rotation as

- Z-axis rotation (Roll)
- Y-axis rotation (Yaw)
- X-axis rotation (Pitch)

Z-axis rotation (Roll)

If you look closely, you should note that when we rotate around the Z axis, the Z element of the point does not change. In fact, we can just ignore the Z – we already know what it will be after the rotation. If we ignore the Z element, then we have the same case as if we were rotating the two-dimensional point <x, y> through the angle θ. Z-axis rotation is same as the origin about the 2D for which we have the derived matrices already.

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

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

$$z' = z$$

Homogeneous representation is:

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

i.e. $P' = R_z(\theta).P$

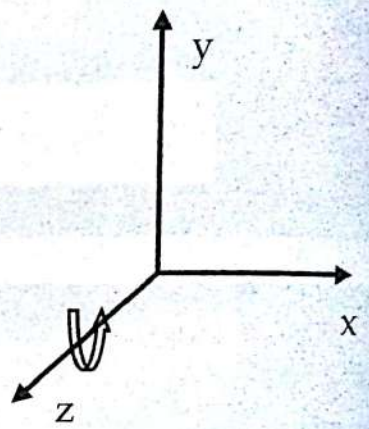


Figure 5.2: Roll

Y-axis rotation (Yaw)

The equations for Y-axis rotation

$$x' = x\cos\theta + z\sin\theta$$

$$y' = y$$

$$z' = z\cos\theta - x\sin\theta$$

Homogeneous representation is:

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

i. e. $P' = R_y(\theta).P$

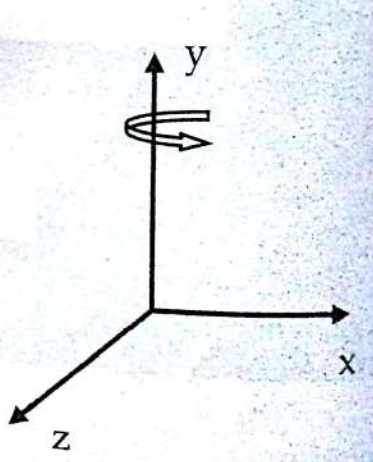


Figure 5.3: Yaw

X-axis rotation (Pitch)

The equations for X-axis rotation

$$x' = x$$

$$y' = y\cos\theta - z\sin\theta$$

$$z' = y\sin\theta + z\cos\theta$$

Homogeneous representation is:

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

i.e. $P' = R_x(\theta).P$

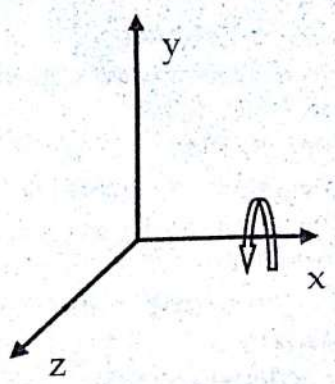


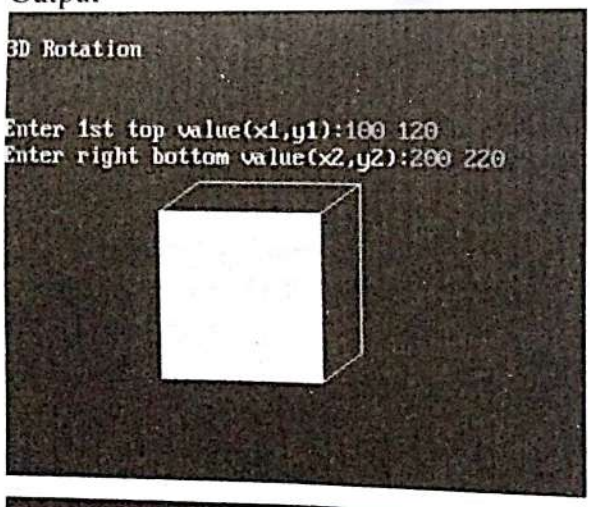
Figure 5.4: Pitch

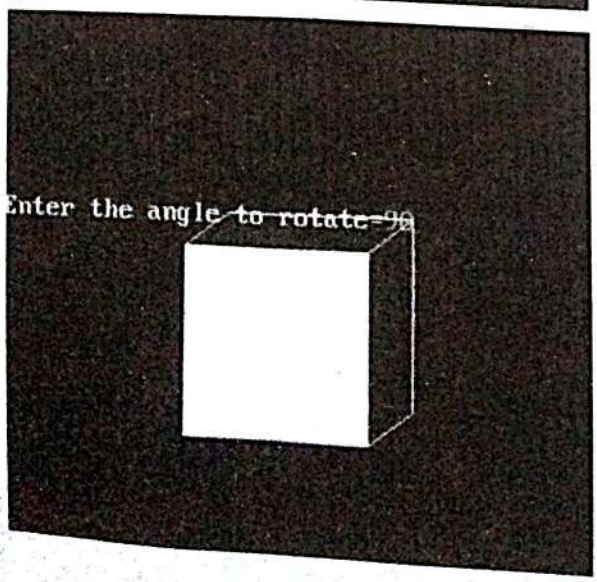
Lab problem 5.2: Program for Rotation

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<graphics.h>
int x1,x2,y1,y2,mx,my,depth;
void draw();
void rotate();
void main()
{
  int gd=DETECT,gm,c;
  initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
  printf("\n3D Transformation Rotating\n\n");
  printf("\nEnter 1st top value(x1,y1):");
  scanf("%d%d",&x1,&y1);
  printf("Enter right bottom value(x2,y2):");
  scanf("%d%d",&x2,&y2);
  depth=(x2-x1)/4;
  mx=(x1+x2)/2;
  my=(y1+y2)/2;
  draw(); getch();
  cleardevice();
  rotate();
  getch();
  }
void draw()
{
  bar3d(x1,y1,x2,y2,depth,1);
}
void rotate()
{
```

```
    float t;
    int a1,b1,a2,b2,dep;
    printf("Enter the angle to rotate=");
    scanf("%f", &t);
    t=t*(3.14/180);
    a1=mx+(x1-mx)*cos(t)-(y1-my)*sin(t);
    a2=mx+(x2-mx)*cos(t)-(y2-my)*sin(t);
    b1=my+(x1-mx)*sin(t)-(y1-my)*cos(t);
    b2=my+(x2-mx)*sin(t)-(y2-my)*cos(t);
    if(a2>a1)
          dep=(a2-a1)/4;
    else
          dep=(a1-a2)/4;
    bar3d(a1,b1,a2,b2,dep,1);
    setcolor(5);
}
```

Output





Scaling

Coordinate transformations for scaling relative to the origin are

$$x' = x.S_x$$

$$y' = y.S_y$$

$$z' = z.S_z$$

Scaling an object with transformation changes the size of the object and repositions the object relative to the coordinate origin. If the transformation parameters are not all equal, relative dimensions in the object are changed.

Uniform Scaling: We preserve the original shape of an object with a uniform scaling (Sx = Sy = Sz).

Differential Scaling: We do not preserve the original shape of an object with a differential scaling ($S_x \neq S_y \neq S_z$).

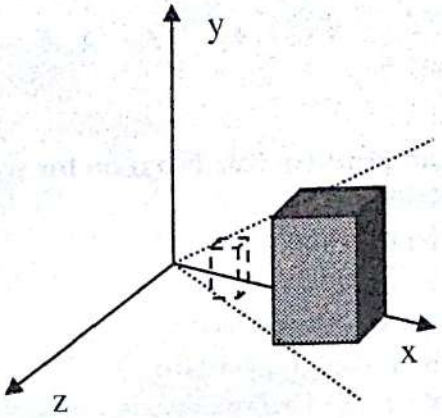


Figure 5.5: 3D Scaling

Scaling relative to the coordinate Origin

Scaling transformation of a position P = *(x, y, z)* relative to the coordinate origin can be written as

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Scaling with respect to a selected fixed position

Scaling with respect to a selected fixed position (xf, yf, zf) can be represented with the following transformation sequence: Translate the fixed point to the origin. Scale the object relative to the coordinate origin Translate the fixed point back to its original position. For these three transformations we can have composite transformation matrix by multiplying three matrices into one

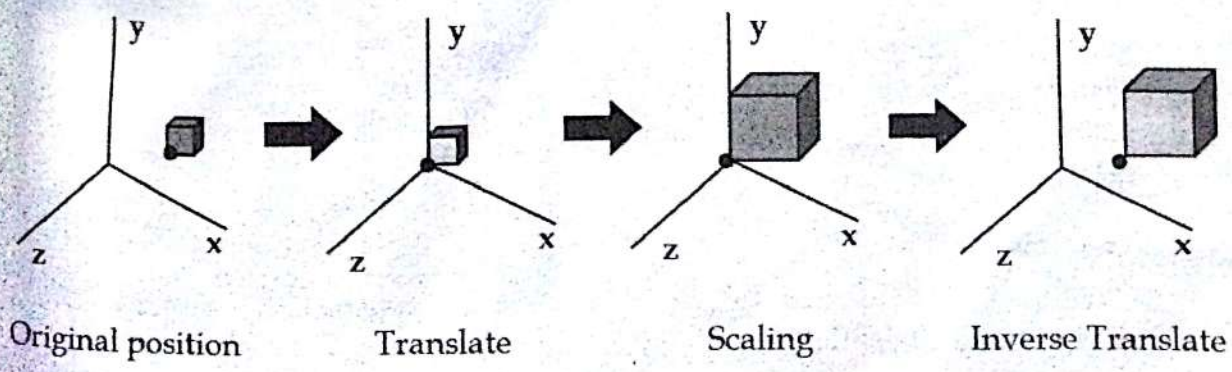


Figure 5.6: 3D Scaling with respect to fixed point

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

Where composite matrix for scaling is

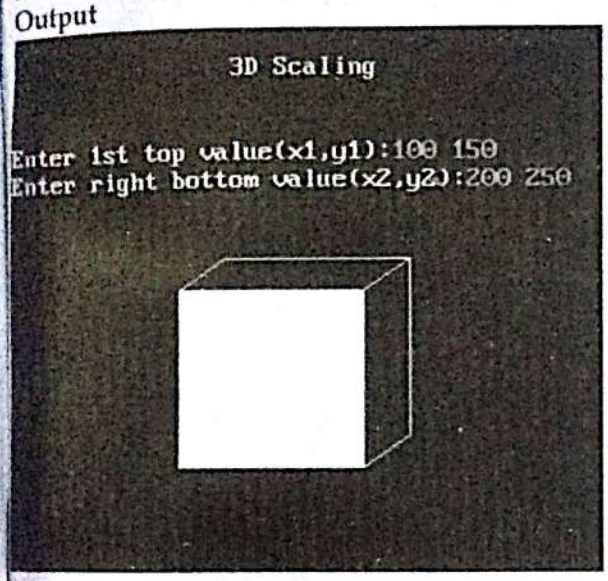
$$T(x_f, y_f, z_f) \cdot S(s_x, s_y, s_z) \cdot T(-x_f, -y_f, -z_f) = \begin{bmatrix} s_x & 0 & 0 & (1-s_x)x_f \\ 0 & s_y & 0 & (1-s_y)y_f \\ 0 & 0 & s_z & (1-s_z)z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

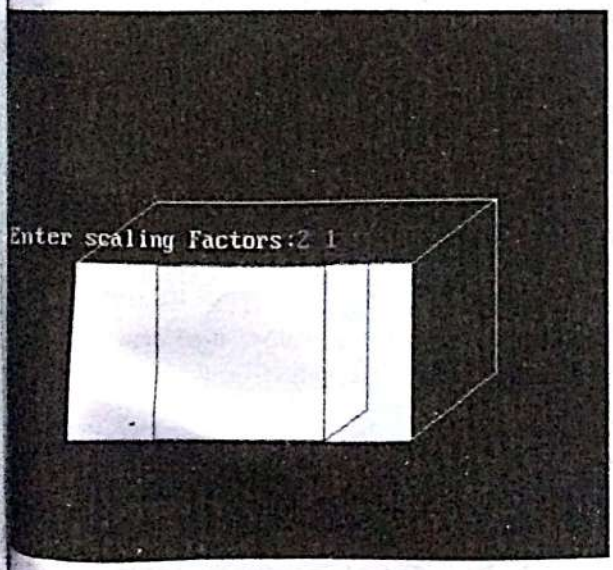
Lab problem 5.3: Program for scaling

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<process.h>
#include<graphics.h>
int x1,x2,y1,y2,mx,my,depth;
void draw();
void scale();
void main()
{
        int gd=DETECT,gm,c;
        initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
        printf("\n\t\t3D Scaling\n\n");
        printf("\nEnter 1st top value(x1,y1):");
        scanf("%d%d",&x1,&y1);
        printf("Enter right bottom value(x2,y2):");
        scanf("%d%d",&x2,&y2);
        depth=(x2-x1)/4;
        mx=(x1+x2)/2;
        my=(y1+y2)/2;
        draw();
        getch();
        cleardevice();
        scale();
        getch();
 }
void draw()
{
        bar3d(x1,y1,x2,y2,depth,1);
}
void scale()
{
        int x,y,a1,a2,b1,b2,dep;
```

```
    printf("\n\n Enter scaling Factors:");
    scanf("%d%d",&x,&y);
    a1=mx+(x1-mx)*x;
    a2=mx+(x2-mx)*x;
    b1=my+(y1-my)*y;
    b2=my+(y2-my)*y;
    dep=(a2-a1)/4;
    bar3d(a1,b1,a2,b2,dep,1);
    setcolor(5);
    draw();
}
```

Output





Reflection

A three-dimensional reflection can be performed relative to a selected reflection axis or with respect to a selected reflection plane. In general, three-dimensional reflection matrices are set up similarly to those for two dimensions. Reflections relative to a given axis are equivalent to 180

degree rotations in four-dimensional space with a conversion between a left-handed frame and a right-handed frame.

The matrix representation for this reflection of points relative to the X axis (Z-X Plane)

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

The matrix representation for this reflection of points relative to the Y axis(Y-Z Plane)

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

The matrix representation for this reflection of points relative to the Z axis(X-Y Plane)

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

Shears

Shearing transformations can be used to modify object shapes. As an example of three dimensional shearing, the following transformation produces az-axis shear:

$$\begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Parameters 'a' and 'b' can be assigned and real values. The effect of this transformation matrix to alter x and y- coordinate values by an amount that is proportional to the z-value, whi leaving the z coordinate unchanged.

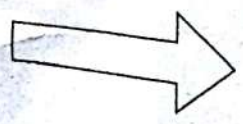


Figure 5.7: A unit cub is sheared relative to the origin by z-axis shear with a = b =1.

Y-Axis Shear

$$\begin{bmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & c & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Parameters 'a' and 'c' can be assigned and real values. The effect of this transformation matrix is to alter x and z- coordinate values by an amount that is proportional to the y value, while leaving the y coordinate unchanged.

X - Axis Shear

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ b & 1 & 0 & 0 \\ c & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Parameters b and c can be assigned and real values. The effect of this transformation matrix is to alter y and z- coordinate values by an amount that is proportional to the x value, while leaving the x coordinate unchanged.

Three-Dimensional Viewing

Three-Dimensional Viewing

For two-dimensional graphics applications, viewing operations transfer positions from the world-coordinate plane to pixel positions in the plane of the output device. Using the rectangular boundaries for the clipping window and the view port, a two-dimensional package clips a scene and maps it to device coordinates. Three-dimensional viewing operations, however, are more involved, because we now have many more choices as to how we can construct a scene and how we can generate views of the scene on an output device. Many processes in three-dimensional viewing, such as the clipping routines, are similar to those in the two-dimensional viewing pipeline. But three-dimensional viewing involves some tasks that are not present in two dimensional viewing. For example, projection routines are needed to transfer the scene to a view on a planar surface, visible parts of a scene must be identified, and, for a realistic display, lighting effects and surface characteristics must be taken into account.

Viewing Pipeline

Modeling Transformation and Viewing Transformation can be done by 3D transformations. The viewing-coordinate system is used in graphics packages as a reference for specifying the observer viewing position and the position of the projection plane. Projection operations convert the viewing-coordinate description (3D) to coordinate positions on the projection plane (2D). (Usually combined with clipping, visual-surface identification, and surface-rendering)Workstation transformation (Normalization transformation and Clipping + Viewport Transformation) maps the coordinate positions on the projection plane to the output device.

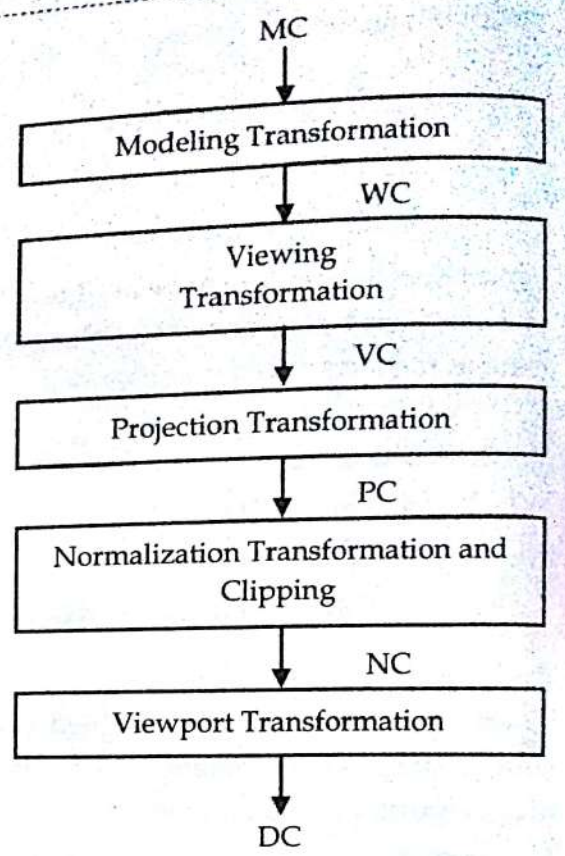


Figure 5.8: 3D viewing pipeline

World to Screen Viewing Transformation

In the three-dimensional viewing pipeline, the first step after a scene has been constructed is to transfer object descriptions to the viewing-coordinate reference frame. Conversion of objection descriptions from world to viewing coordinates is equivalent to a transformation that superimposes the viewing reference frame onto the world frame using the basic geometric translate-rotate operations:

1. Translate the viewing-coordinate origin to the origin of the world coordinate system.
2. Apply rotations to align the x-view, y-view, and z-view axes with the world xw, yw, and zw-axes, respectively.

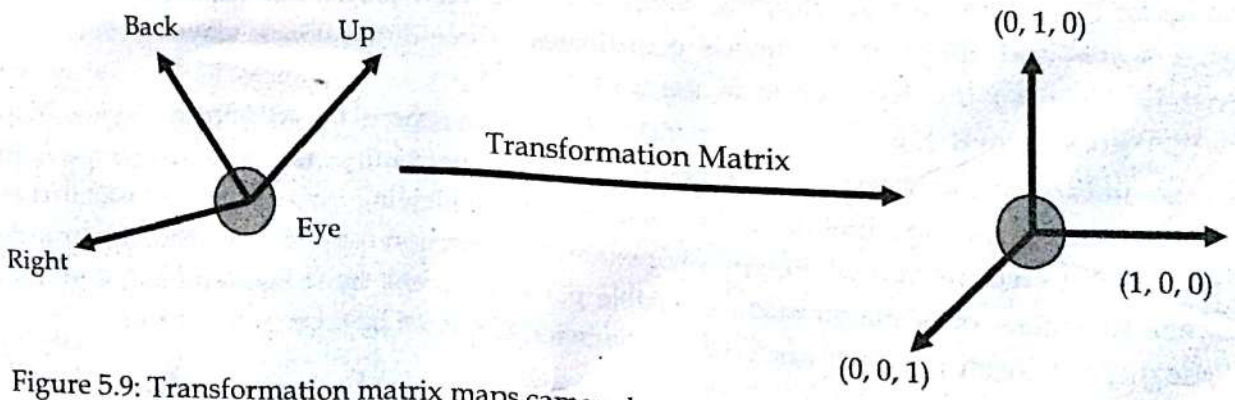


Figure 5.9: Transformation matrix maps camera basis vectors to canonical vectors in viewing coordinate system

Projection Concept

For centuries, artists, engineers, designers, drafters, and architects have been facing difficulties and constraints imposed by the problem of representing a three-dimensional object or scene in a two-dimensional medium -- the problem of projection. The implementers of a computer graphics system face the same challenge.

Projection can be defined as a mapping of point $P(x, y, z)$ onto its image $P'(x', y', z')$ in the projection plane or view plane, which constitutes the display surface. The mapping is determined by a projection line called the projector that passes through P and intersects the view plane. There are two basic methods of projection:

- Parallel Projection
- Perspective Projection

These methods are used to solve the basic problems of pictorial representations. We characterize each method and introduce the mathematical description of the projection process respectively.

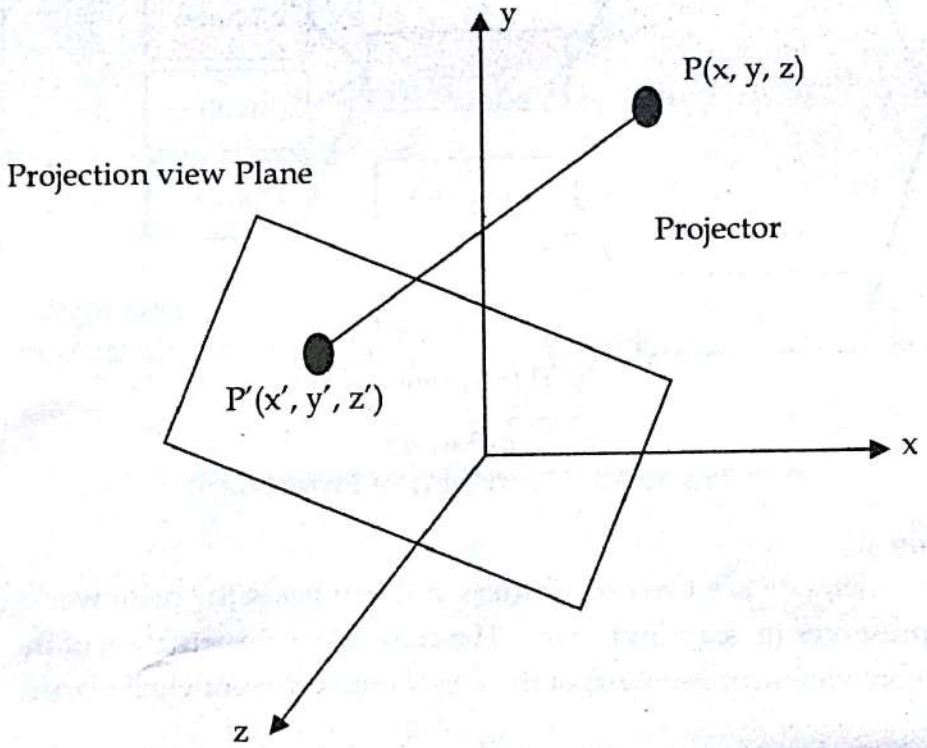


Figure 5.10: The Problem of Projection

Taxonomy of Projection

We can construct different projections according to the view that is desired. Following figure provides taxonomy of the families of perspective and parallel projections. Some projections have names – cavalier, cabinet, isometric, and so on. Other projections qualify the main type of projection – one principal vanishing-point perspective and so forth.

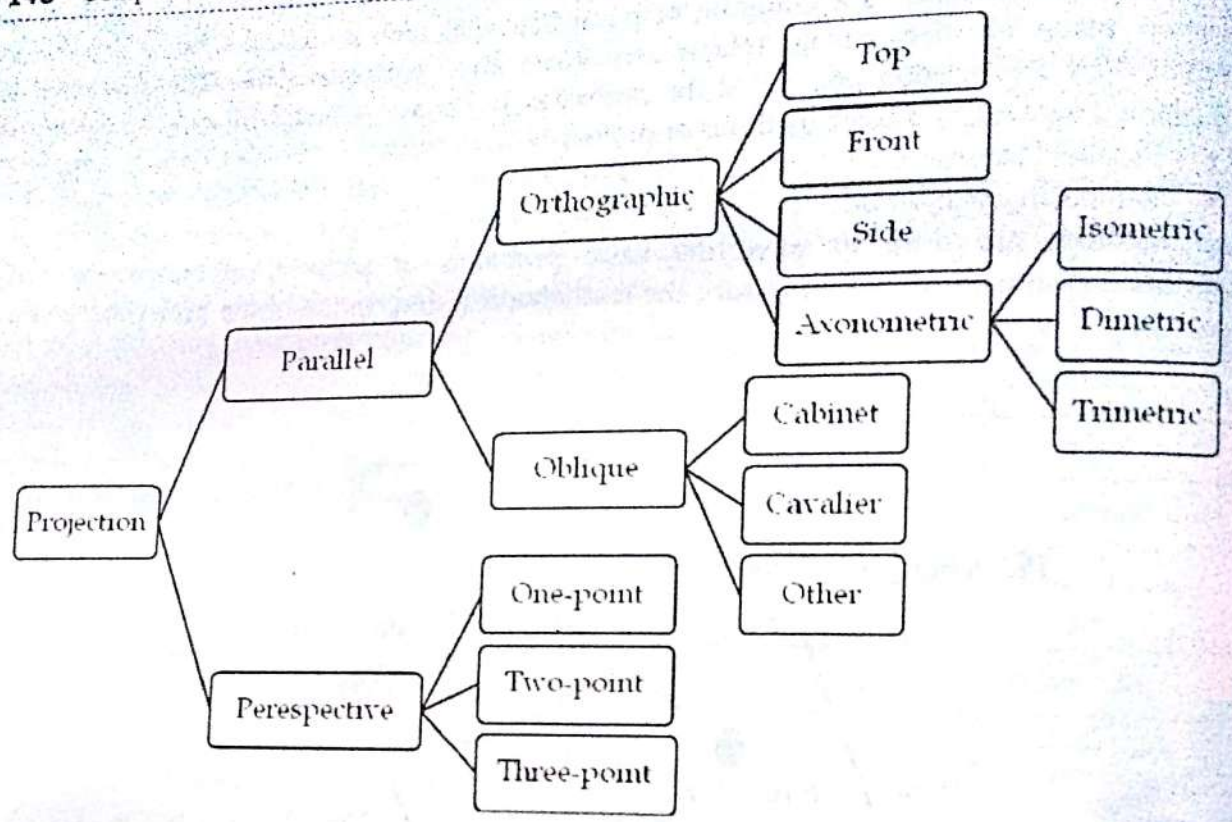


Figure 5.11: Taxonomy of Projection

Parallel Projection

Parallel projection methods are used by drafters and engineers to create working drawings of an object which preserves its scale and shape. The complete representation of these details often requires two or more views (projections) of the object onto different view planes.

In parallel projection, image points are found as the intersection of the view plane with a projector drawn from the object point and having a fixed direction. The direction of projection is the prescribed direction for all projections. **Orthographic projections are characterized by the fact that the direction of projection is perpendicular to the view plane.** When the direction of projection is parallel to any of the principal axes, this produces the front, top, and side views of mechanical drawings (also referred to as multi view drawings).

Axonometric projections are orthographic projections in which the direction of projection is not parallel to any of the three principal axes. **Non orthographic parallel projections are called oblique parallel projection.**

Projection rays (projectors) emanate from a Center of Projection (COP) and intersect Projection Plane (PP). The COP for parallel projectors is at infinity. The length of a line on the projection plane is the same as the "true Length".

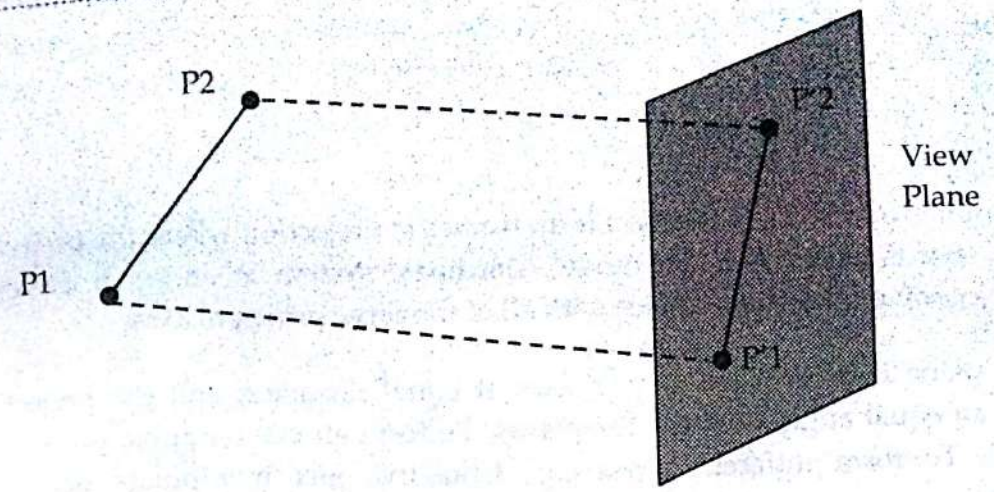


Figure 5.12: Parallel Projection

There are two different types of parallel projections:

- **Orthographic**
- **Oblique**

1) Orthographic Projection

If the direction of projection is perpendicular to the projection plane then it is an **orthographic** projection.

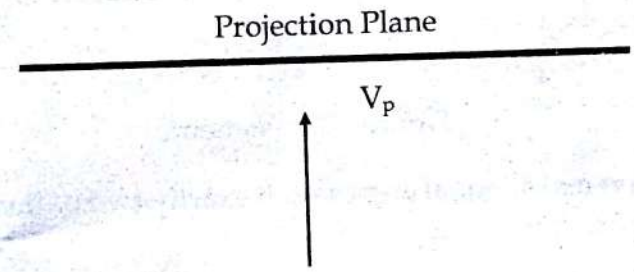


Figure 5.13: Orthographic Projection

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

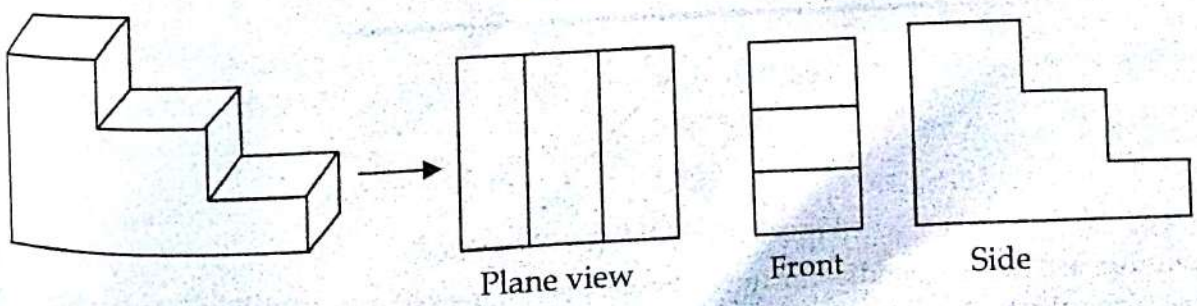


Figure 5.14: orthographic projections

Axonometric orthographic projection

Orthographic projections that show more than one side of an object are called **axonometric** orthographic projections. There are three axonometric projections and they are:

- Isometric
- Dimetric
- Trimetric

Isometric

The most common axonometric projection is an isometric projection where the projection plane intersects each coordinate axis in the model coordinate system at an equal distance or the direction of projection makes equal angles with all of the three principal axes.

The projection plane intersects the x, y, z axes at equal distances and the projection plane Normal makes an equal angle with the three axes. To form an orthographic projection $x_p = x$, $y_p = y$, $z_p = 0$. To form different types e.g., Isometric, just manipulate object with 3D transformations.

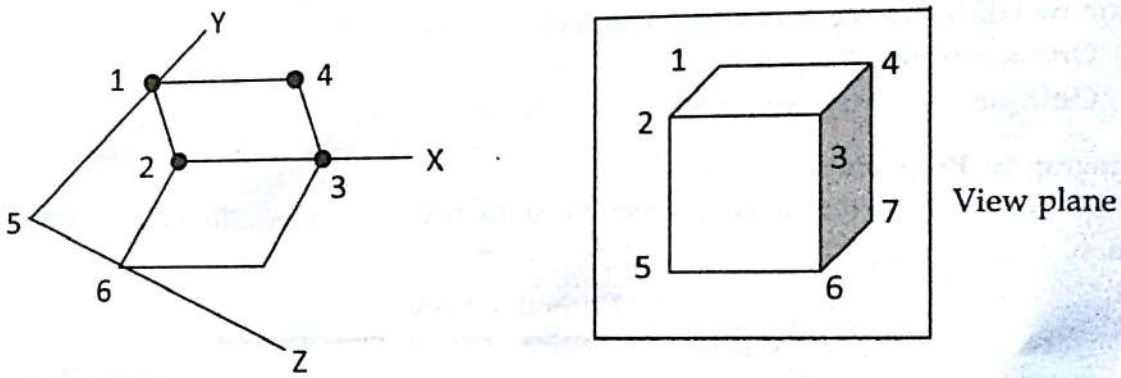


Figure 5.15: Isometric

Dimetric

The direction of projection makes equal angles with exactly two of the principal axes

Trimetric

The direction of projection makes unequal angles with the three principal axes

2) Oblique Projection

If the direction of projection is not perpendicular to the projection plane then it is an **oblique** projection.

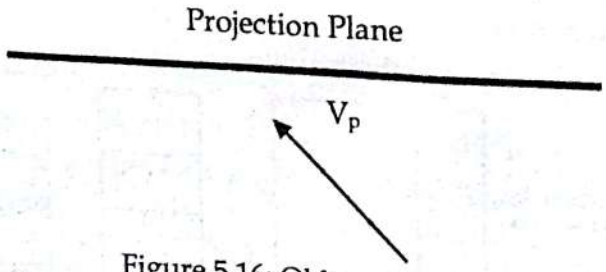


Figure 5.16: Oblique Projection

The projectors are not perpendicular to the projection plane but are parallel from the object to the projection plane. Transformation equations for an orthographic parallel projection are straightforward. If the view plane is placed at position Z_{vp} along the Z axis, then any point (x, y, z) in viewing coordinates is transformed to projection coordinates as:

$x_p = x$

$y_p = y$

Where the original Z-coordinate value is preserved for the depth information needed in depth cueing and visible-surface determination procedures.

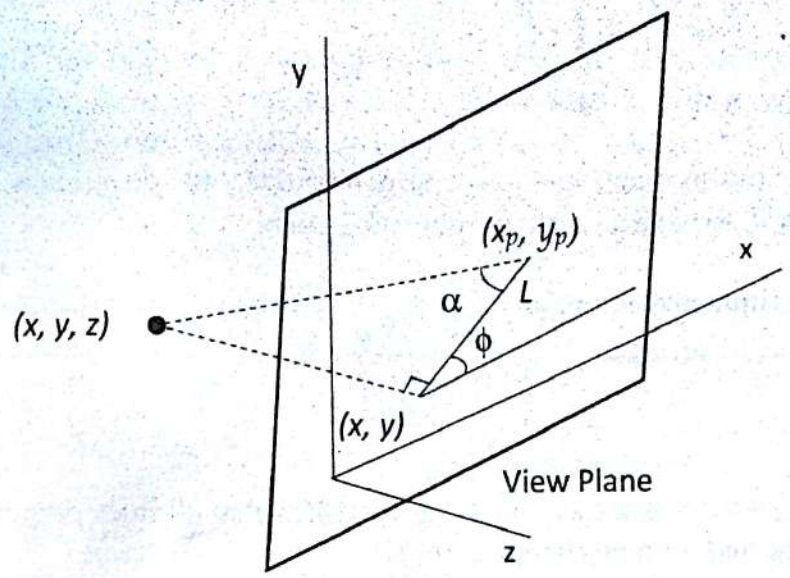


Figure 5.17: Oblique projection of coordinate position (x, y, z) to position (x_p, y_p) on the view plane

An oblique projection is obtained by projecting points along parallel lines that are not perpendicular to the projection plane. In some applications packages, an oblique projection vector is specified with two angles, alpha and phi, as shown in the figure. Point(x, y, z) is projected to position (x_p, y_p) on the view plane. Orthographic projection coordinates on the plane are (x, y). The oblique projection line from (x,y,z) to (x_p, y_p)makes an angle alpha with the line on the projection plane that joins (x_p, y_p) and (x, y).This line, of length L, is at an angle phi with the horizontal direction in the projection plane. We can express the projection coordinates in terms of x, y, L, and phi as

$$x_p = x + L\cos\phi, y_p = y + L\sin\phi$$

$$\tan\alpha = \frac{z}{L},\ L = \frac{z}{\tan\alpha} = zL_1$$

$$x_p = x + z(L_1\cos\phi)$$

$$y_p = y + z(L_1\sin\phi)$$

Where L_1 is the inverse of tan a, which is also the value of L when z=1

The transformation matrix for producing any parallel projection onto the xy-plane can be written as

$$\mathbf{M}_{parallel} = \begin{bmatrix} 1 & 0 & L_1\cos\phi & 0 \\ 0 & 1 & L_1\sin\phi & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

i.e.

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ w_p \end{bmatrix} = \begin{bmatrix} 1 & 0 & L_1 \cos\phi & 0 \\ 0 & 1 & L_1 \sin\phi & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Now if $\alpha = 90°$ (projection line is perpendicular to Projection Plane) then $\tan(\alpha)$ = infinity => $L_1 = 0$, so have an orthographic projection.

Two special cases of oblique projection are:

- Cavalier
- Cabinet

Cavalier

$\alpha = 45°$, $\tan \alpha = 1$ => $L_1 = 1$ this is a Cavalier projection such that all lines perpendicular to the projection plane are projected with no change in length.

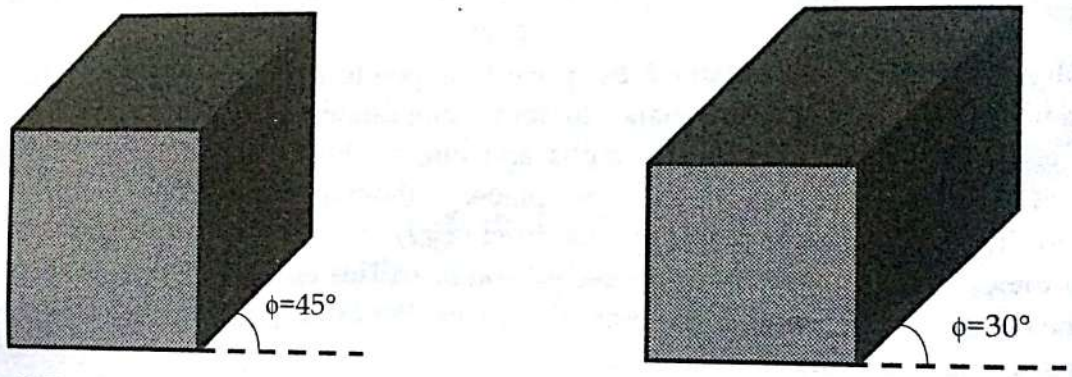


Figure 5.18: Cavalier projections of a cube onto a view plane for two values of angle ϕ. The depth of the cube is projected with a length equal to that of the width and height.

Cabinet

Tan $\alpha = 2$, $\alpha = 63.40°$, $L_1 = 1/2$ Lines which are perpendicular to the projection plane are projected at 1 / 2 length. This is a **Cabinet** projection

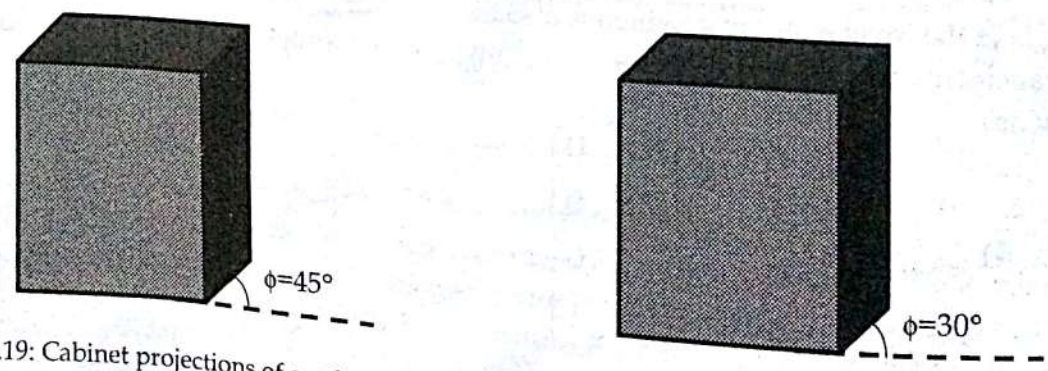


Figure 5.19: Cabinet projections of a cube onto a view plane for two values of angle φ. The depth is projected with a length at is one half that of the width and height of the cube.

Perspective Projection

The Perspective viewing projection has a Center of Projection ("eye") at a finite distance from the projection plane (PP) as in figure 5.20.

The distance of a line from the projection plane determines its size on the projection plane, i.e. the farther the line is from the projection plane, the smaller its image on the projection plane. In the two images above, the projections of L1 = L2 but the actual length of L1 ≠ L2. Perspective projection is more realistic since distant objects appear smaller.

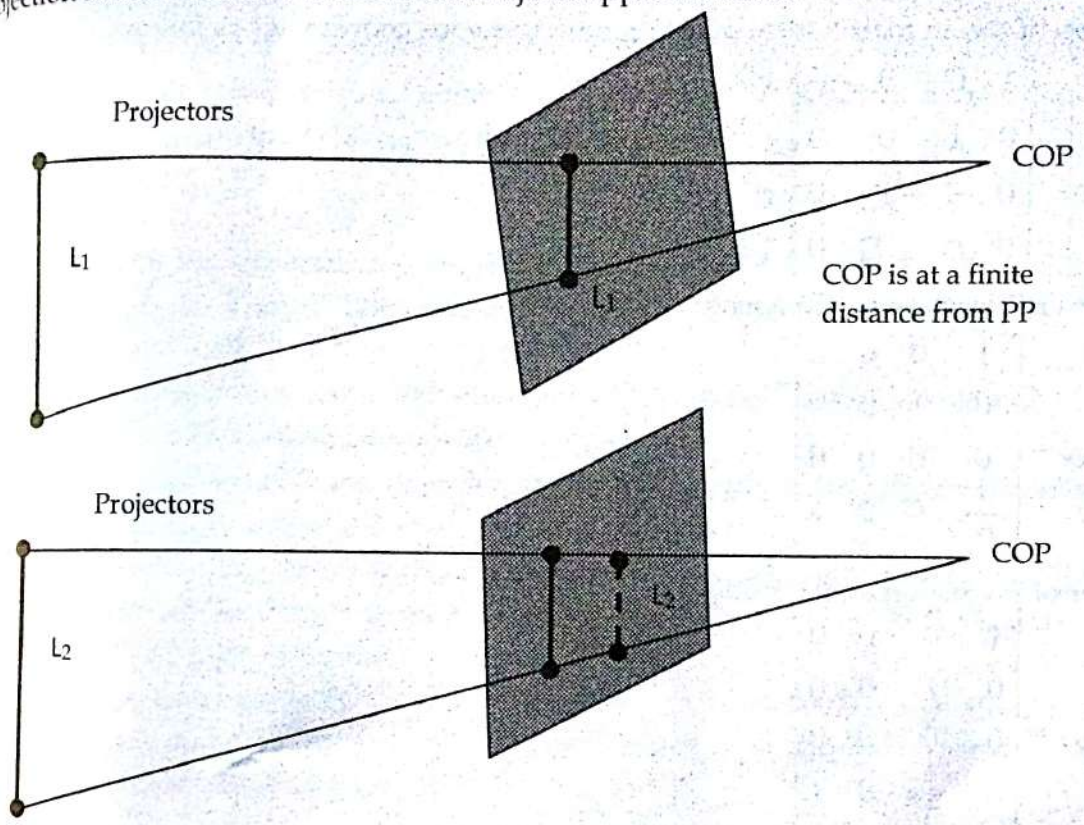
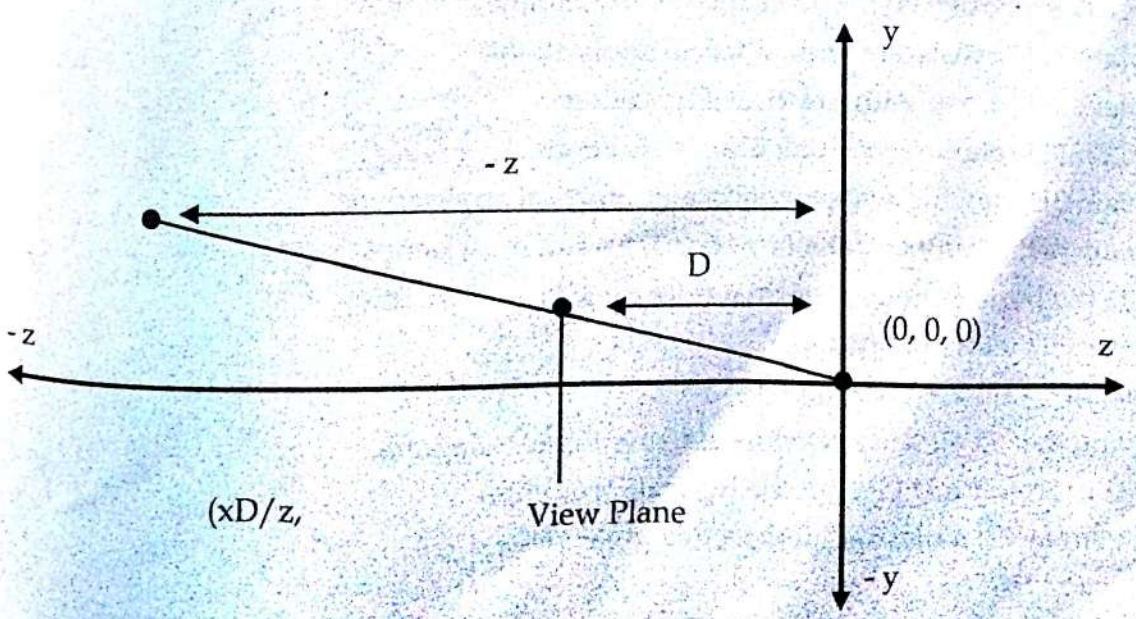


Figure 5.20: Perspective projection

Computing the Perspective Projection

Computing 2D coordinates from 3D coordinates with similar triangles.



Now,

$$x_p = xD/z \qquad x' = x$$
$$y_p = yD/z \qquad y' = y$$
$$z_p = D \qquad z' = z$$
$$w_p = 1 \qquad w_p = z/D$$

We can represent this in matrix form by using homogeneous coordinates as follows:

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ w_p \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/D & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

When Center of Projection on the x axis

$$\mathbf{M}_{PER} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{D_x} & 0 & 0 & 1 \end{bmatrix}$$

When Center of Projection on the y axis

$$\mathbf{M}_{PER} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \frac{1}{D_y} & 0 & 1 \end{bmatrix}$$

Exercise

1. Explain the Basic 3D transformation with example.
2. Explain 3D Translation with practical applications.
3. Explain 3D Mirror with practical applications.
4. Explain 3D rotation with practical applications.
5. Explain 3D viewing with equation and practical applications.
6. Explain 3D rotational transformation with suitable example.
7. Write the homogenous coordinates for all 3D transformation.
8. Explain 3D viewing process.
9. Explain 3D viewing parameters.
10. Explain the 3D viewing pipeline along with neat diagram.
11. What is projection? List out the types of projection.
12. Differentiate parallel and perspective projections.

13. Describe projection transformation in 3D.
14. Differentiate oblique and orthogonal projections.
15. Derive the transformation matrix for general parallel projection.
16. Derive the transformation matrix for perspective projection.
17. What are different perspective anomalies?
18. Give the single-point perspective projection transformation matrix when projectors are placed on the z-axis.
19. Write a procedure to perform a two-point perspective projective projection of an object.
20. Generate a homogeneous matrix representation for oblique projection of co-ordinate position (x, y, z) to position (x_1, y_1) on the view plane.

Numerical

1. Determine the 3D transformation matrices to scale a line PQ in the x direction by 3 by keeping point P fixed. Then rotate this line by 45° anticlockwise about the z axis. Given P (1, 5, 2) and Q (4, 5, 6).
2. Demonstrate local scaling taking scaling vectors along the x, y, z axes as 2, 3, 1 respectively for a cube with homogeneous position vectors.
3. Find the new co-ordinates of a unit cube 90 degree rotated about an axis defined by its end points A(2, 1, 0) and B(3, 3, 1).
4. A tetrahedron of size 10 units is placed on the x-plane with one edge along the x-axis and the origin at the center of the object. Assuming the COP at (5, 0, 0), draw the projected image using perspective projection transformation.
5. A tetrahedron of size 10 units is placed on xy plane with one edge along x-axis (+ve) and one vertex at origin. Assuming the tetrahedron to be opaque evaluates and draw projected image if centre of projection is (10, 0, 0).

❑❑❑

Chapter

6

3D objects Representation

STRUCTURE

- Introduction
- Representing Surface: Boundary and Space partitioning
  - Polygon Surface: Polygon tables, Surface normal and spatial orientation of surfaces, Plane equations, Polygon meshes
  - Wireframe Representation
  - Blobby Objects
- Representing Curves: Parametric Cubic Curves, Spline Representation, Cubic spline interpolation, Hermite Curves, Bezier and B-spline Curve and surface.
- Quadric Surface: Sphere and Ellipsoid

Introduction

Graphics scenes can contain many different kinds of objects and material surfaces: trees, flowers, clouds, rocks, water, bricks, wood paneling, rubber, paper, marble, steel, glass, plastic, and cloth and so on. There is no single method that we can use to describe objects that will include all the characteristics of these different materials. Polygon and quadric surfaces provide precise descriptions for simple Euclidean objects such as polyhedrons and ellipsoids. They are examples of **boundary representations (B-reps)**, which describe a three-dimensional object as a set of surfaces that separate the object interior from the environment. Spline surface are useful for designing aircraft wings, gears and other engineering objects. Procedural methods and particle systems allow us to give accurate representation of clouds, clumps of grass, and other natural objects. Octree encodings are used to represent internal features of objects. Such as medical CT images

Representing Surface

Representation schemes for solid objects are divided into two categories as follows:

1. **Boundary Representation (B-res)**
   It describes a three dimensional object as a set of surfaces that separate the object interior from the environment. Examples are polygon facets and spline patches.
2. **Space Partitioning Representation**
   It describes the interior properties, by partitioning the spatial region containing an object into a set of small, non-overlapping, contiguous solids (usually cubes). One of the examples is Octree Representation

Polygon Surface

Polygon surfaces are boundary representations for a 3D graphics object is a set of polygons that enclose the object interior.

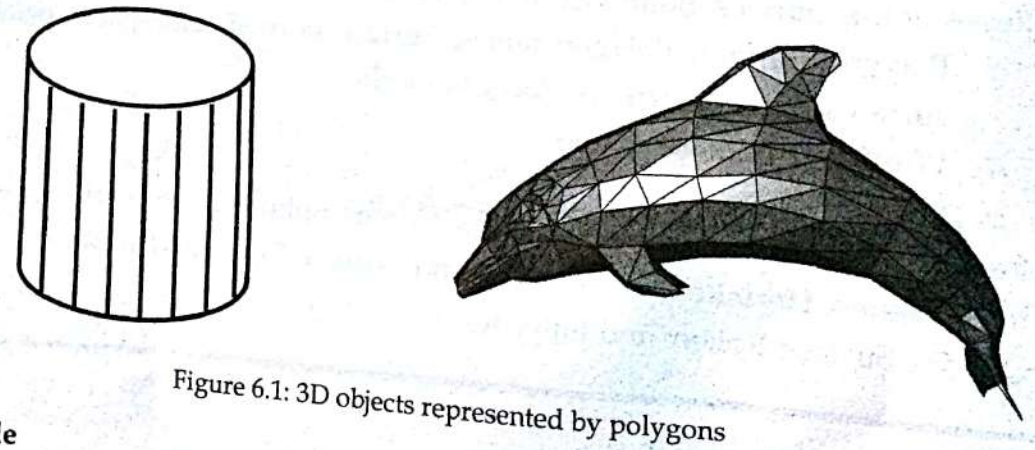


Figure 6.1: 3D objects represented by polygons

Polygon Table

The polygon surface is specified with a set of vertex coordinates and associated attribute parameters. For each polygon input, the data are placed into tables that are to be used in the subsequent processing. Polygon data tables can be organized into two groups: Geometric tables and attribute tables.

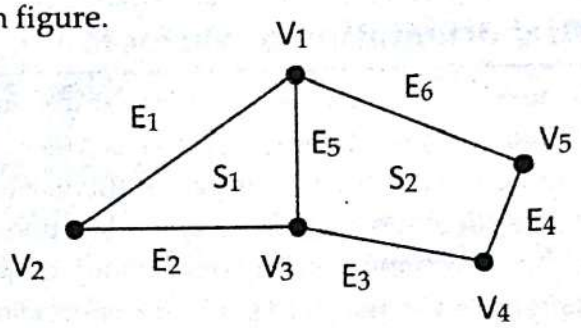
Geometric Tables

Contain vertex coordinates and parameters to identify the spatial orientation of the polygon surfaces.

Attribute tables Contain attribute information for an object such as parameters specifying the degree of transparency of the object and its surface reflectivity and texture characteristics.

A convenient organization for storing geometric data is to create three lists:

- **The Vertex Table**
  Coordinate values for each vertex in the object are stored in this table.
- **The Edge Table**
  It contains pointers back into the vertex table to identify the vertices for each polygon edge.
- **The Polygon Table**
  It contains pointers back into the edge table to identify the edges for each polygon.
  This is shown in figure.



VERTEX TABLE
V_1: x_1, y_1, z_1
V_2: x_2, y_2, z_2
V_3: x_3, y_3, z_3
V_4: x_4, y_4, z_4
V_5: x_5, y_5, z_5

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

POLYGON SURFACE TABLE
S_1: E_1, E_2, E_5
S_2: E_3, E_4, E_5, E_6

Listing the geometric data in three tables provides a convenient reference to the individual components (vertices, edges and polygons) of each object. The object can be displayed efficiently by using data from the edge table to draw the component lines. Extra information can be added to the data tables for faster information extraction. For instance, edge table can be expanded to include forward points into the polygon table so that common edges between polygons can be identified more rapidly.

EDGE TABLE
E_1: V_1, V_2, S_1
E_2: V_2, V_3, S_1
E_3: V_3, V_4, S_2
E_4: V_4, V_5, S_2
E_5: V_1, V_3, S_1, S_2
E_6: V_5, V_1, S_2

This is useful for the rendering procedure that must vary surface shading smoothly across the edges from one polygon to the next. Similarly, the vertex table can be expanded so that vertices

are cross-referenced to corresponding edges. Additional geometric information that is stored in the data tables includes the slope for each edge and the coordinate extends for each polygon. As vertices are input, we can calculate edge slopes and we can scan the coordinate values to identify the minimum and maximum x, y and z values for individual polygons. The more information included in the data tables will be easier to check for errors.

Some of the tests that could be performed by a graphics package are:

- That every vertex is listed as an endpoint for at least two edges.
- That every edge is part of at least one polygon.
- That every polygon is closed.
- That each polygon has at least one shared edge.
- That if the edge table contains pointers to polygons, every edge referenced by a polygon pointer has a reciprocal pointer back to the polygon.

Surface normal and spatial orientation of surfaces

A normal is the technical term used in Computer Graphics (and Geometry) to describe the orientation of a surface of a geometric object at a point on that surface. Technically, the surface normal to a surface at point P can be seen as the vector perpendicular to a plane tangent to the surface at P. At the same time, the direction of this vector determines the orientation of the surface. In the case of polygons, this direction is usually determined by the right hand rule.
In computer graphics, manipulations of the normal vector are often used as a way to simulate geometrical detail on otherwise planar surfaces. In this case, a function will determine small aberrations of the true direction of the normal vector on every point of the surface, in order special create highlight or shadow effects. Normal plays an important role in shading where they are used to compute the brightness of objects. If e.g. the vector is slightly shifted in accordance to a sinus function, then the surface will appear in a rendered image, as if it were made of corrugated material (except for the edges).

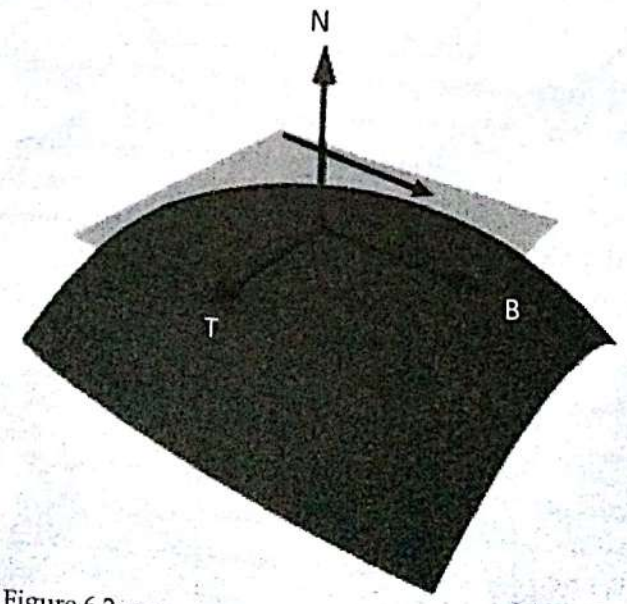


Figure 6.2: a normal is perpendicular to the plane tangent at P.

Spatial Orientation of a polygon face is the vertex coordinates values and the equations of the polygon surfaces. The general equation of a plane containing a polygon is

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

Where (x, y, z) is any point on the plane; A, B, C and Dare plane parameters giving the spatial properties of the plane and A, B, C and D can be calculated by three non-collinear points in the plane, selected in a **strictly counterclockwise order**, viewing the surface along a front-to-back direction.

Plane Equations

To produce a display of a 3D object, we must process the input data representation for the object through several procedures such as, Transformation of the modeling and world coordinate descriptions to viewing coordinates. Then to device coordinates: Identification of visible surfaces, the application of surface-rendering procedures. For these processes, we need information about the spatial orientation of the individual surface components of the object. This information is obtained from the vertex coordinate value and the equations that describe the polygon planes.

The equation for a plane surface is

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

Where (x, y, z) is any point on the plane, and the coefficients A, B, C and D are constants describing the spatial properties of the plane.

We can obtain the values of A, B, C and D by solving a set of three plane equations using the coordinate values for three non collinear points in the plane. For that, we can select three successive polygon vertices (x_1, y_1, z_1), (x_2, y_2, z_2) *and* (x_3, y_3, z_3) and solve the following set of simultaneous linear plane equations for the ratios $\frac{A}{D}, \frac{B}{D}$ *and* $\frac{C}{D}$.

$$\frac{A}{D}x_k + \frac{B}{D}y_k + \frac{C}{D}z_k = -1; \quad for\ k = 1,2,3.$$

The solution for this set of equations can be obtained in determinant form, using Cramer's rule as

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

$$D = -\begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix}$$

Expanding the determinants, we can write the calculations for the plane coefficients in the form:

$$A = y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2)$$
$$B = z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2)$$
$$C = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$$
$$D = -x_1(y_2 z_3 - y_3 z_2) - x_2(y_3 z_1 - y_1 z_3) - x_3(y_1 z_2 - y_2 z_1)$$

As vertex values and other information are entered into the polygon data structure, values for A, B, C and D are computed for each polygon and stored with the other polygon data.

Plane equations are used also to identify the position of spatial points relative to the plane surfaces of an object. For any point (x, y, z) not on a plane with parameters A, B, C, D, we have

$$Ax + By + Cz + D \neq 0$$

We can identify the point as either inside or outside the plane surface according to the sigh (negative or positive) of $Ax + By + Cz + D$:

If $Ax + By + Cz + D < 0$, the point (x, y, z) is inside the surface

If $Ax + By + Cz + D > 0$, the point (x, y, z) is outside the surface

These inequality tests are valid in a right handed Cartesian system, provided the plane parameters A, B, C and D were calculated using vertices selected in a counter clockwise order when viewing the surface in an outside-to-inside direction.

Polygon Meshes

A polygon mesh is collection of edges, vertices and polygons connected such that each edge is shared by at most two polygons. An edge connects two vertices and a polygon is a closed sequence of edges. An edge can be shared by two polygons and a vertex is shared by at least two edges. A single plane surface can be specified with a function such as fillArea. But when object surfaces are to be tiled, it is more convenient to specify the surface facets with a mesh function. One type of polygon mesh is the triangle strip. This function produces n-2 connected triangles.

Another similar function is the quadrilateral mesh, which generates a mesh of (n-1) by (m-1) quadrilaterals, given the co-ordinates for an $n \times m$ array of vertices. Shown figure is a vertex forming a mesh of 12 quadrilaterals.

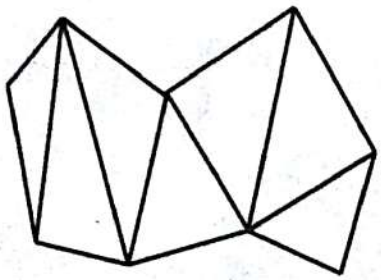


Figure 6.3: Triangle strip

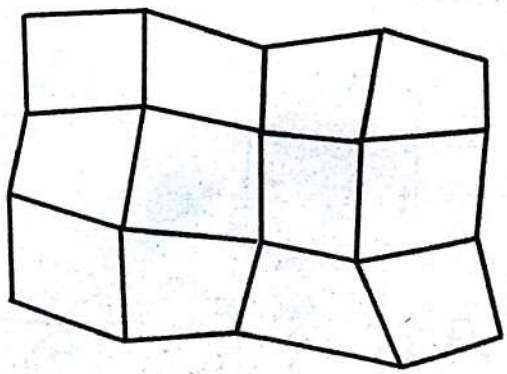


Figure 6.4: Quadrilateral mesh

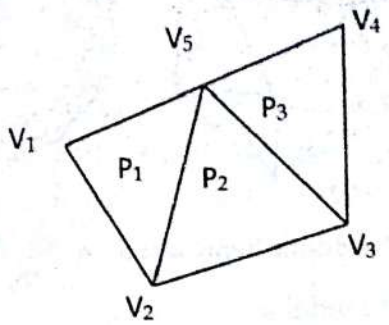
Representing polygon meshes

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

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

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

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



Representing polygon mesh with each polygon as vertex list

$$P_1 = \{v_1, v_2, v_5\}$$
$$P_2 = \{v_2, v_3, v_5\}$$
$$P_3 = \{v_3, v_4, v_5\}$$

Here most of the vertices are duplicated so it is not efficient.

Representation with indexes into a vertex list

$$V = \{v_1, v_2, v_3, v_4, v_5\} = \{(x_1, y_1, z_1), ...(x_5, y_5, z_5)\}$$
$$P_1 = \{1, 2, 5\}$$
$$P_2 = \{2, 3, 5\}$$
$$P_3 = \{3, 4, 5\}$$

Pros and Cons of Polygon Mesh

Pros

- It can be used to model almost any object.
- They are easy to represent as a collection of vertices.
- They are easy to transform.
- They are easy to draw on computer screen.

Cons

- Curved surfaces can only be approximately described.
- It is difficult to simulate some type of objects like hair or liquid.

Wire frame Representation

If the object is defined only by a set of nodes, and a set of lines connecting the nodes, then the resulting object representation is called a wireframe model. In this method, a 3D object is represented as a list of straight lines, each of which is represented by its two end points (x_1, y_1, z_1) and (x_2, y_2, z_2). This method only shows the skeletal structure of the objects. It is

simple and can see through the object and fast method. But independent line data structure is very inefficient. In this method, the scenes represented are not very realistic.

A wireframe model consists of edges, vertices, and polygons. Here vertices are connected by edges, and polygons are sequences of vertices or edges. The edges may be curved or straight line segments. In the latter case, the wireframe model is called a polygonal net or polygonal mesh.

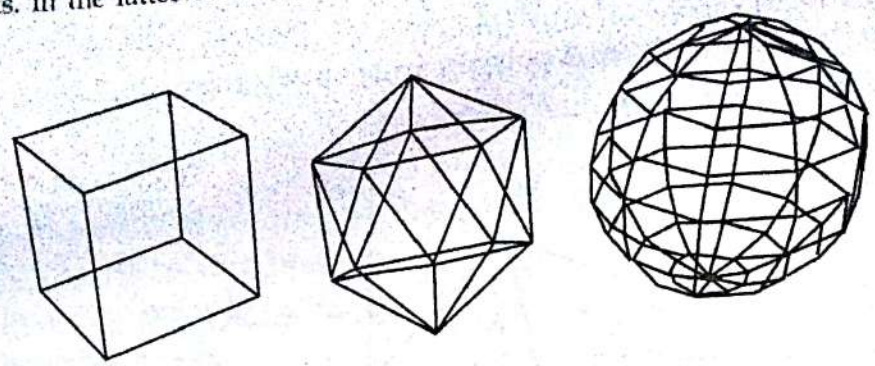


Figure 6.5: Wireframe Representation

Polygon definitions are used in an object model to:

a) Render the object structure, with shades/colors for different polygonal faces constituting the object.
b) Eliminate hidden lines after projection, by comparing the line segments with the polygonal elements.
c) Eliminate hidden surfaces using z-buffer, surface normal methods.

Polyhedron

A polyhedron is a closed polygonal net (i.e. one which encloses a define volume) in which each polygon is planar. The polygons are called the faces of the polyhedron. In modeling, polyhedrons are quite often treated as solid (i.e. block) objects as opposed to wireframes or two -dimensional surfaces.

Advantages and Disadvantages of Wireframe Models

Wireframe models are used in engineering applications. They are easy to construct and, if they are compound of straight lines, easy to clip and manipulate through the use of geometric and coordinate transformations. However, for building realistic models, especially of highly curved objects, we must use a very large number of polygons to achieve the illusions of roundness and smoothness.

Blobby objects: Some objects do not maintain a fixed shape but change their surface characteristics in certain motions or when proximity to other objects. e.g. molecular structures, water droplets, other liquid effects, melting objects, muscle shape in human body etc. these objects can be described as exhibiting "blobbiness" and referred as blobby objects. Several models have been developed for representing blobby objects as distribution functions over a region of space. One way is to use **Gaussian density function.**

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

Where, $r = \sqrt{(x_k^2 + y_k^2 + 2_k^2)}$, T=Threshold; 'a' and 'b' are used to adjust amount of blobby ness. Other methods for generating blobby objects use **quadratic density function** as

$$f(r) = \begin{cases} b(1-\frac{3r^2}{d^2}) \text{ if } 0 \le r \le \frac{d}{3} \\ \frac{3}{2}b(1-r/d)^2 \text{ if } d/3 < r \le d \\ 0 \text{ if } r > d. \end{cases}$$

Representing Curves

In computer graphics, we often need to draw different types of objects onto the screen. Objects are not flat all the time and we need to draw curves many times to draw an object.

Types of Curves

A curve is an infinitely large set of points. Each point has two neighbors except endpoints. Curves can be broadly classified into three categories and they are **explicit, implicit,** and **parametric curves.**

Implicit Curves

Implicit curve representations define the set of points on a curve by employing a procedure that can test to see if a point in on the curve. Usually, an implicit curve is defined by an implicit function of the form:

$f(x, y) = 0$ In two dimensions and

$f(x, y, z) = 0$ In three dimensions

It can represent multi valued curves (multiple y values for an x value). Curves in three dimensions are not as easily represented in implicit form. A common example is the circle, whose implicit representation is

$$x^2 + y^2 - r^2 = 0$$

Explicit Curves

A mathematical function $y = f(x)$ can be plotted as a curve. Such a function is the explicit representation of the curve. The explicit representation is not general, since it cannot represent vertical lines and is also single-valued. For each value of x, only a single value of y is normally computed by the function.

Parametric Curves

Curves having parametric form are called parametric curves. The explicit and implicit curve representations can be used only when the function is known. In practice the parametric curves are used. A two-dimensional parametric curve has the following form:

$$P(t) = f(t), g(t) \; or P(t) = x(t), y(t)$$

The functions f and g become the (x, y) coordinates of any point on the curve, and the points are obtained when the parameter t is varied over a certain interval [a, b], normally [0, 1].

Parametric cubic curves

Once we have decided to use parametric polynomial curves, we must choose the degree of the curve. If we choose a high degree, we will have many parameters that we can set to form the desired shape, but evaluation of points on the curve will be costly. In addition, as the degree of

a polynomial curve becomes higher, there is more danger that the curve will become rougher. On the other hand, if we pick too low a degree, we may not have enough parameters with which to work. However, if we design each curve segment over a short interval, we can achieve many of our purposes with low-degree curves. Although there may be only a few degrees of freedom these few may be sufficient to allow us to produce the desired shape in a small region. For this reason, most designers, at least initially, work with cubic polynomial curves

Algebraic Representation of parametric curve

All of these curves are just parametric algebraic polynomials expressed in different bases

Parametric linear curve

$$\mathbf{p}(u) = \mathbf{a}u + \mathbf{b}$$

$$x = a_x u + b_x$$

$$y = a_y u + b_y$$

$$z = a_z u + b_z$$

Parametric cubic curve

$$\mathbf{p}(u) = \mathbf{a}u^3 + \mathbf{b}u^2 + \mathbf{c}u + \mathbf{d}$$

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

$$y = a_y u^3 + b_y u^2 + c_y u + d_y$$

$$z = a_z u^3 + b_z u^2 + c_z u + d_z$$

Advantages of parametric curves

- More degrees of freedom
- Directly transformable
- Dimension independent
- No infinite slope problems
- Separates dependent and independent variables
- Inherently bounded
- Easy to express in vector and matrix form
- Common form for many curves and surfaces
- Fewer parameters makes it faster to create a curve, and easier to edit an existing curve
- Normal fields can be properly defined everywhere
- Parametric curves are easier to animate than polygon meshes

Spline Representation

A Spline is a flexible strip used to produce a smooth curve through a designated set of points. Several small weights are distributed along the length of the strip to hold it in position on the drafting table as the curve is drawn. The **Spline curve** refers to any sections curve formed with polynomial sections satisfying specified continuity conditions at the boundary of the pieces. Mathematically, spline curves are described as piece-wise cubic polynomial function. A **Spline surface** can be described with two sets of orthogonal spline curves.

Splines are used in graphics applications to design curve and surface shapes, to digitize drawings for computer storage, and to specify animation paths for the objects or the camera in the scene. CAD applications for splines include the design of automobiles bodies, aircraft and spacecraft surfaces, ship hulls, and so on.

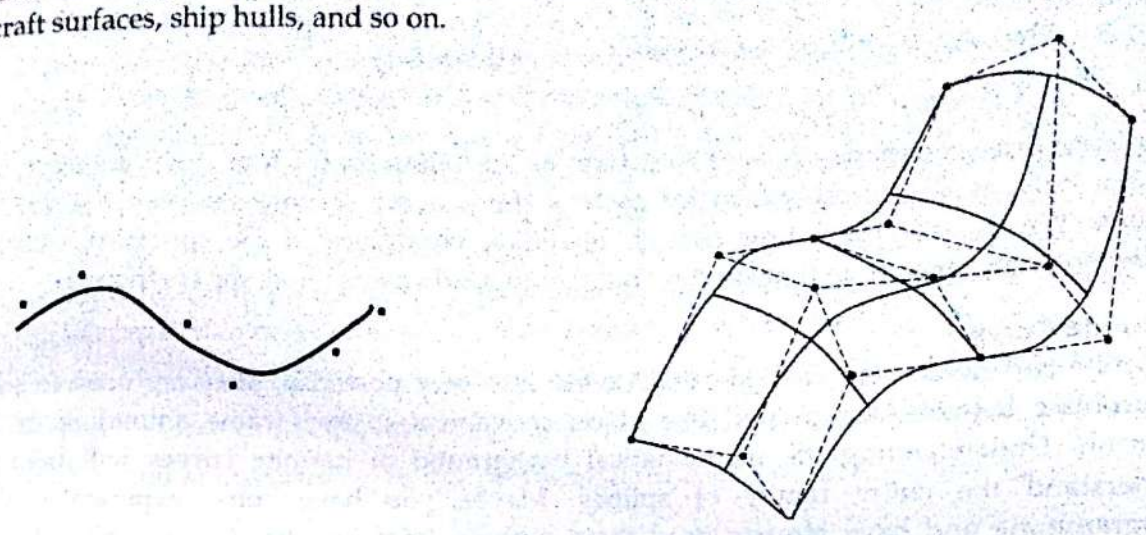


Figure 6.6: designing curve and surface shape with spline

Cubic spline interpolation

The cubic spline was originally introduced by James Ferguson. *Spline* was a term for elastic rulers that were bent to pass through a number of predefined points ("knots"). These were used to make technical drawings for shipbuilding and construction by hand. Cubic spline interpolation is a special case for Spline interpolation that is used very often to avoid the problem of Runge's phenomenon. This method gives an interpolating polynomial that is smoother and has smaller error than some other interpolating polynomials such as Lagrange polynomial and Newton polynomial. Cubic polynomials offer a reasonable compromise between flexibility and speed of computation. Cubic spline requires less calculations compares to higher order polynomials and less memory. Compared to lower polynomials cubic spline are more flexible for modeling arbitrary curve shape.

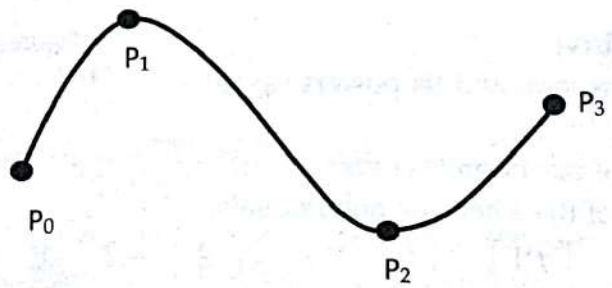


Figure 6.7: Interpolation with cubic splines between 4 control points

Given a set of control points, cubic interpolation splines are obtained by fitting the input points with a piecewise cubic polynomial curve that passes through every control points.
Suppose we have n+1 control points specified with co-ordinates

$$p_k = (x_k, y_k, z_k),\ k = 0,1,2.....n$$

The parametric cubic polynomial that is to be filled between each pair of control points with the following set of equations.

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

$$y(u) = a_y u^3 + b_y u^2 + c_y u + dy \qquad (0 \le u \le 1)$$

$$z_u(u) = a_z u^3 + b_z u^2 + c_z u + dz$$

For these three equations, we need to determine the values for the four coefficients *a*, *b*, *c*, and *d* in the polynomial representation for each of the *n* curve sections between the *n*+1 control points. We do this by setting enough boundary conditions at the control-point positions between curve sections so that we can obtain numerical values for all the coefficients.

Hermite Curve

Hermite curves are very easy to calculate but also very powerful. They are used to smoothly interpolate between key-points (like object movement in key frame animation or camera control). Understanding the mathematical background of hermite curves will help you to understand the entire family of splines. Maybe you have some experience with 3D programming and have already used them without knowing that (the so called kb-splines, curves with control over tension, continuity and bias are just a special form of the hermite curves).

To keep it simple we first with 2-dimensional curves here. Hermite curves work in in any number of dimensions. To calculate a hermite curve you need the following vectors:

- P1: the start point of the curve
- T1: the tangent (e.g. direction and speed) to how the curve leaves the start point
- P2: the endpoint of the curve
- T2: the tangent (e.g. direction and speed) to how the curves meets the endpoint

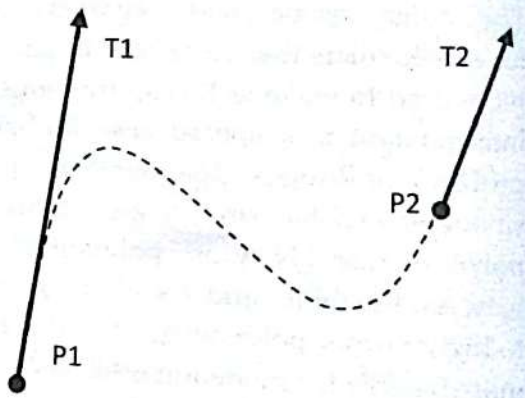


Figure 6.8: Hermite Curve

Matrix form of Hermite Curve

Vector S: The interpolation-point and its powers up to 3:

Vector C: The parameters of our hermite curve:

Matrix h: The matrix form of the 4 hermite polynomials:

$$S = \begin{bmatrix} s^3 \\ s^2 \\ s^1 \\ 1 \end{bmatrix} \qquad C = \begin{bmatrix} P1 \\ P2 \\ T1 \\ T2 \end{bmatrix} \qquad h = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

To calculate a point on the curve you build the Vector S, multiply it with the matrix h and then multiply with C.

$$P = S \times h \times C.$$

Bézier Curve

Bezier curve is discovered by the French engineer **Pierre Bézier.** The **Bézier** curve is a parametric curve **P**(*t*) that is a polynomial function of the parameter *t*. The degree of the polynomial depends on the number of points used to define the curve. The method employs **control points** and produces an approximating curve. The curve does not pass through the interior points but is attracted by them. It is as if the points exert a pull on the curve. Each point influences the direction of the curve by pulling it toward itself, and that influence is strongest when the curve gets nearest the point. Bézier curves are widely used in computer graphics to model smooth curves. As the curve is completely contained in the convex hull of its control points, the points can be graphically displayed and used to manipulate the curve intuitively. Affine transformations such as translation and rotation can be applied on the curve by applying the respective transform on the control points of the curve.

Consider a case of n+1 control points positions denoted by $p_k = (x_k, y_k, z_k), k = 0, 1, 2, ..n$. These co-ordinate points can be blended to produce the following position vector p(u) which describes path of and approximating Bezier polynomial function between p_o and p_n.

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

The Bezier blending functions $BEZ_{k,n}(u)$ the *Bernstein polynomial.*

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

Where parameters $C(n,k)$ are binomial coefficients

$$C(n,k) = \frac{n!}{k!(n-k)!}$$

The equation $p(u) = \sum_{k=0}^{n} p_k BEZ_{k,n}(u)$, *where* $0 \le u \le 1$ represents a set of three parametric equations for individual curve condition

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

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

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

In most cases, a Bézier curve is a polynomial of a degree that is one less than the designated number of control points: Three points generate a parabola, four points a cubic curve, and so forth. i.e. *(n-1)* degree of polynomial equation for the *n* control points.

Figure 6.9: Simple Bezier Curve

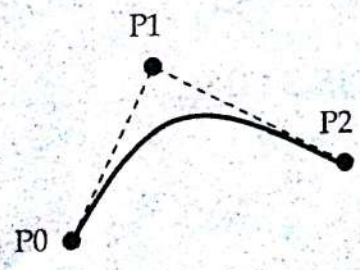


Figure 6.10: Quadratic Bezier Curve

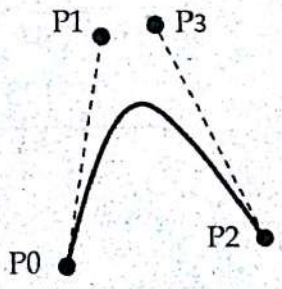


Figure 6.11: Cubic Bezier Curve

Properties of Bézier curve

Bezier curves have the following properties:

- They generally follow the shape of the control polygon, which consists of the segments joining the control points.
- They always pass through the first and last control points.
- They are contained in the convex hull of their defining control points.
- The degree of the polynomial defining the curve segment is one less that the number of defining polygon point. Therefore, for 4 control points, the degree of the polynomial is 3, i.e. cubic polynomial.
- A Bezier curve generally follows the shape of the defining polygon.
- The direction of the tangent vector at the end points is same as that of the vector determined by first and last segments.
- The convex hull property for a Bezier curve ensures that the polynomial smoothly follows the control points.
- No straight line intersects a Bezier curve more times than it intersects its control polygon.
- They are invariant under an affine transformation.
- Bezier curves exhibit global control means moving a control point alters the shape of the whole curve.
- A given Bezier curve can be subdivided at a point $t=t_0$ into two Bezier segments which join together at the point corresponding to the parameter value $t=t_0$.

Example 6.1: Construct the Bezier curve of order 3 and with 4 polygon vertices A(1, 1), B(2, 3), C(4, 3) and D(6, 4).

Solution: The equation for the Bezier Curve is given as

$$P(u) = (1-u)^3 P_1 + 3u(1-u)^2 P_2 + 3u^2(1-u)P_3 + u^3 P_4, \quad for\, 0 \le u \le 1$$

Where $P(u)$ is the point on the curve P_1, P_2, P_3, P_4

Let us take, $u = 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}$

$$\therefore P(0) = P_1 = (1,1)$$

$$\therefore P\left(\frac{1}{4}\right) = \left(1-\frac{1}{4}\right)^3 P_1 + 3\times\frac{1}{4}\left(1-\frac{1}{4}\right)^2 p_2 + 3\left(\frac{1}{4}\right)^2\left(1-\frac{1}{4}\right)p_3 + \left(\frac{1}{4}\right)^3 p_4$$

$$= \frac{27}{64}(1,1) + \frac{27}{64}(2,3) + \frac{9}{64}(4,3) + \frac{1}{64}(6,4)$$

$$= \left(\frac{27}{64}\times 1, + \frac{27}{64}\times 2 + \frac{9}{64}\times 4 + \frac{1}{64}\times 6, \frac{27}{64}\times 1, + \frac{27}{64}\times 3 + \frac{9}{64}\times 3 + \frac{1}{64}\times 4\right)$$

$$= \left(\frac{123}{64}, \frac{139}{64}\right) = (1.9218, 2.1718)$$

$$\therefore P\left(\frac{1}{2}\right)=\left(1-\frac{1}{2}\right)^3 P_1+3\times\frac{1}{2}\left(1-\frac{1}{2}\right)^2 p_2+3\left(\frac{1}{2}\right)^2\left(1-\frac{1}{2}\right)p_3+\left(\frac{1}{2}\right)^3 p_4$$

$$=\frac{1}{8}(1,1)+\frac{3}{8}(2,3)+\frac{3}{8}(4,3)+\frac{1}{8}(6,4)$$

$$=\left(\frac{1}{8}\times1,+\frac{3}{8}\times2+\frac{3}{8}\times4+\frac{1}{8}\times6,\ \frac{1}{8}\times1,+\frac{3}{8}\times3+\frac{3}{8}\times3+\frac{1}{8}\times4\right)$$

$$=\left(\frac{25}{8},\frac{23}{8}\right)=(3.125,\ 2.875)$$

$$\therefore P\left(\frac{3}{4}\right)=\left(1-\frac{3}{4}\right)^3 P_1+3\times\frac{3}{4}\left(1-\frac{3}{4}\right)^2 p_2+3\left(\frac{3}{4}\right)^2\left(1-\frac{3}{4}\right)p_3+\left(\frac{3}{4}\right)^3 p_4$$

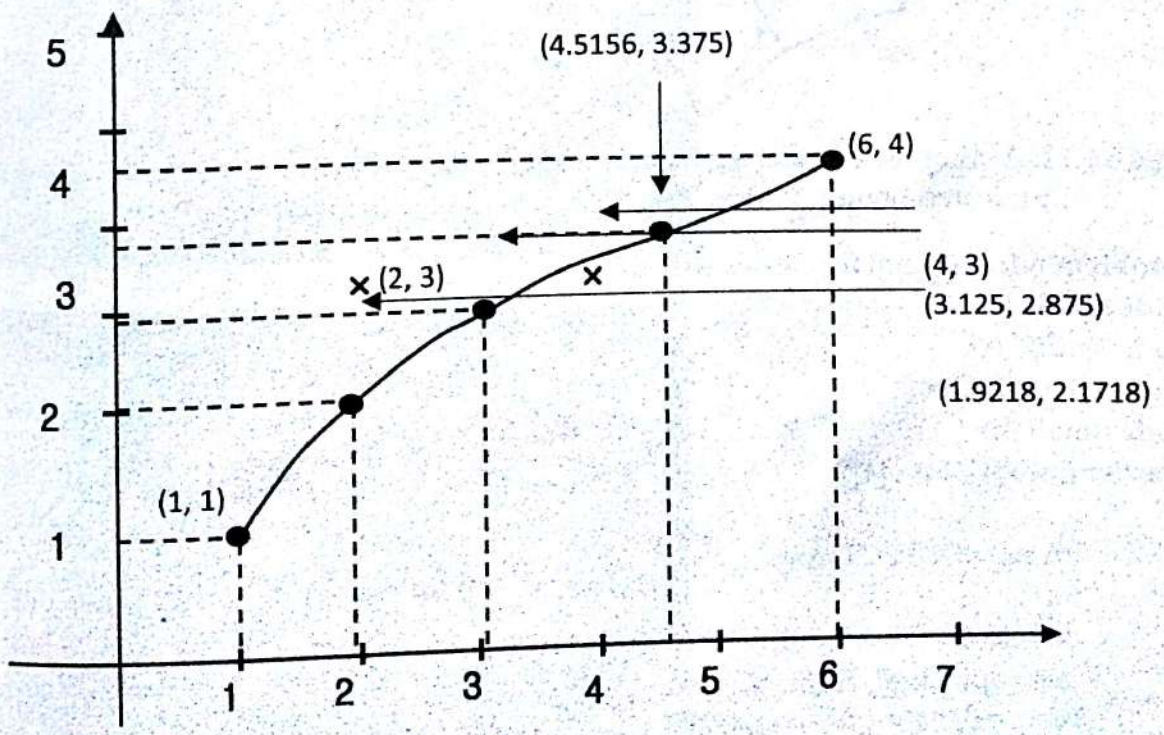
$$=\frac{1}{64}(1,1)+\frac{9}{64}(2,3)+\frac{27}{64}(4,3)+\frac{27}{64}(6,4)$$

$$=\left(\frac{1}{64}\times1,+\frac{9}{64}\times2+\frac{27}{64}\times4+\frac{27}{64}\times6,\ \frac{1}{64}\times1,+\frac{9}{64}\times3+\frac{27}{64}\times3+\frac{27}{64}\times4\right)$$

$$=\left(\frac{289}{64},\frac{217}{64}\right)=(4.5156,\ 3.375)$$

$$\therefore P(1)=P_3=(6,4)$$

The graph below shows that calculated points of the Bezier curve and curve passing through it.



Bézier Surface

Two sets of orthogonal Bézier curves can be used to design an object surface. The parametric vector function for the Bézier surface is formed as the tensor product of Bézier blending functions:

$$P(u,v) = \sum_{j=0}^{m} \sum_{k=0}^{n} P_{j,k} BEZ_{j,m}(v) BEZ_{k,n}(u)$$

With $P_{j,k}$ specifying the location of the $(m + 1)$ by $(n + 1)$ control points. Figure 6.12 illustrates a surface formed with two Bezier sections. As with curves, 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.

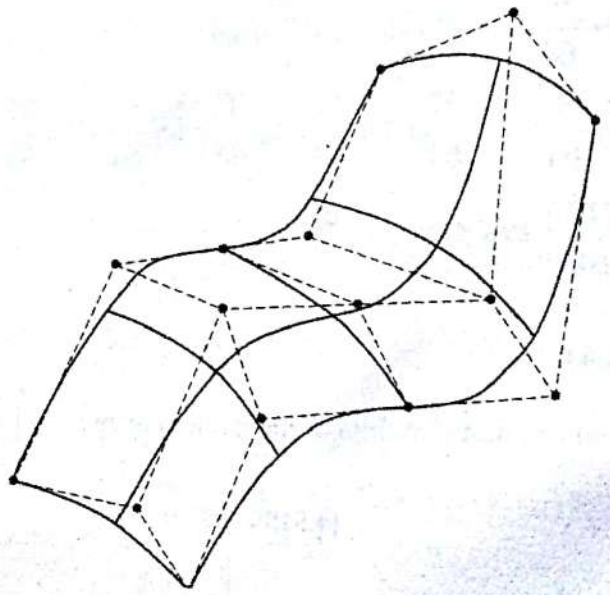


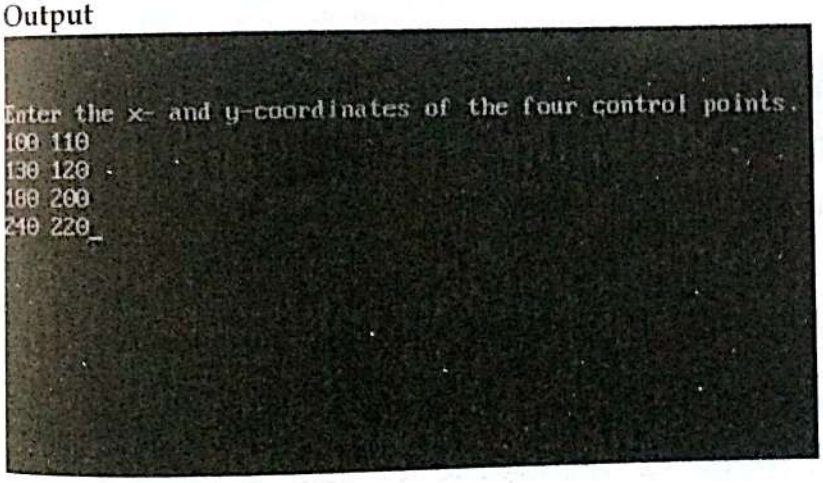
Figure 6.12: A composite Bézier surface constructed with two Bézier sections, joined at the indicated boundary line. The dashed lines connect the control points.

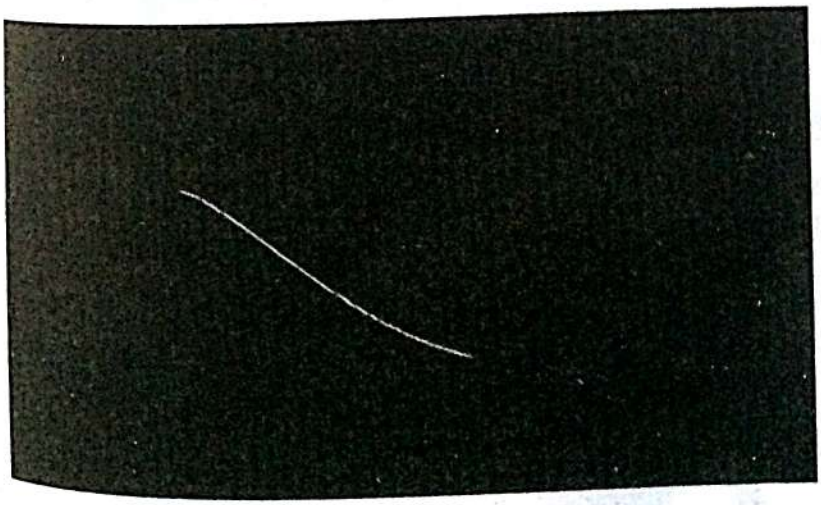
Lab problem 6.1: Program for Bezier curve:

```
#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>
#include <math.h>
void bezier (int x[4], int y[4])
{
        int gd = DETECT, gm;
        int i;
        double t;
        initgraph (&gd, &gm, "C:\\TurboC3\\BGI");
        for (t = 0.0; t < 1.0; t += 0.0005)
        {
```

```
        double xt = pow (1-t, 3) * x[0] + 3 * t * pow (1-t, 2) * x[1] + 3 * pow (t, 2) * (1-t)
                    * x[2] + pow (t, 3) * x[3];
        double yt = pow (1-t, 3) * y[0] + 3 * t * pow (1-t, 2) * y[1] + 3 * pow (t, 2) * (1-
                    t) * y[2] + pow (t, 3) * y[3];
        putpixel (xt, yt, WHITE);
    }
    for (i=0; i<4; i++)
        putpixel (x[i], y[i], YELLOW);
    getch();
    closegraph();
    return;
}
void main()
{
    int x[4], y[4];
    int i;
    printf ("Enter the x- and y-coordinates of the four control points.\n");
    for (i=0; i<4; i++)
        scanf ("%d%d", &x[i], &y[i]);
    bezier (x, y);
}
```

Output





B-Spline Curves and Surface

B-spline methods for curves and surfaces were first proposed in the 1940s but were seriously developed only in the 1970s, by several researchers, most notably R. Riesenfeld. They have been studied extensively, have been considerably extended since the 1970s,and much is currently known about them. The designation "B" stands for Basis, so the full name of this approach to curve and surface design is the basis spline which contains the Bernstein basis as a special case. The B-spline basis is non-global. It is non-global because each B_i vertex is associated with a unique basis function. Thus each vertex affects the shape of the curve only over a range of parameter values where it associated basis function is nonzero. The B-spline basis also allows the order of the basis function and hence the degree of the resulting curve is independent on the number of vertices. It is possible to change the degree of the resulting curve without changing the number of vertices of the defining polygon.

There is most widely used class of approximating splines. B-spline has a general expression for the calculation of Co-ordinate positions along a curve in a blending function as:

$$P(u) = \sum_{k=0}^{n} B_i B_{i,k}(u), \quad u_{\min} \le u \le u_{\max}, 2 \le k \le n+1;$$

Where $\mathbf{B_i}$ are the position vectors of the **n+1** defining polygon vertices and the $\mathbf{N_{i,\,k}}$ are the normalized B-spline basis functions. For the ith normalized B-spline basis function of order K, the basis function $N_{i,\,k}$ (u) are defined as

$$N_{i,1}(u) = \begin{cases} 1 \ if \ x_i \le u < x_{i+1} \\ 0 \ Otherwise \end{cases}$$

And $$N_{i,1}(u) = \frac{(u - x_1)N_{i,k-1}(u)}{x_{i+k-1} - x_i} + \frac{(x_{i+k} - u)N_{i+1,k-1}(u)}{x_{i+k} - x_{i+1}}$$

The values of x_i are the elements of a knot vector satisfying the relation $x_i \le x_{i+1}$. The parameter u varies from u_{min} to u_{max} along the curve P(u). The choice of knot vector has a significant influence on the B-spline basis functions $N_{i,k}$ (u) and hence on the resulting B-spline curve. There are three types of knot vector: uniform open uniform and on uniform.

In a uniform knot vector, individual knot values are evenly spaced. For example,

[0 1 2 3 4]

For a given order k, uniform knot vectors give periodic uniform basis functions for which

$$N_{i,1}(u) = N_{i-1,k}(u-1) = N_{i+1,k}(u+1)$$

An open uniform knot vector has multiplicity of knot values at the ends equal to the order k of the B-spline basis function. Internal knot values are evenly spaced. Examples are,

k=2[0 0 1 2 3 3]

k=3[0 0 0 1 2 3 3 3]

k=4[0 0 0 0 1 2 2 2 2]

Generally, an open uniform knot vector is given by,

$x_i=0 \qquad 1 \le i \le k$

$x_i = i-k$ $\quad k+1 \le i \le n+1$

$x_i = n-k+2$ $\quad n+2 \le i \le n+k+1$

The curve resulted by the use of open uniform basis function are nearly like Bezier curve. In fact, when the number of defining polygon vertices is equal to the order of the B-spline basis and an open uniform knot vector is used, the B-spline basis reduces to the Bernstein basis. Hence the resulting B-spline curve is a Bezier curve.

B-spline has two advantages.

1. Can be any degree, independent of no. of. Control points.
2. Allow local control over the shape of surface.

Disadvantage: (Solid-object Representation)

Hierarchical tree structures, octrees are used to represent solid objects in some graphics system, medical imaging and other applications that require displays of objects cross sections commonly use this method.

Properties of B-spline Curve

- B-spline curves have the following properties:
- The sum of the B-spline basis functions for any parameter value is 1.
- Each basis function is positive or zero for all parameter values.
- Each basis function has precisely one maximum value, except for k=1.
- The maximum order of the curve is equal to the number of vertices of defining polygon.
- The degree of B-spline polynomial is independent on the number of vertices of defining polygon.
- B-spline allows the local control over the curve surface because each vertex affects the shape of a curve only over a range of parameter values where its associated basis function is nonzero.
- The curve exhibits the variation diminishing property.
- The curve generally follows the shape of defining polygon.
- Any affine transformation can be applied to the curve by applying it to the vertices of defining polygon.
- The curve line within the convex hull of its defining polygon.
- Any affine transformation can be applied to the curve by applying it to the vertices of defining polygon.
- The curve line within the convex hull of its defining polygon.

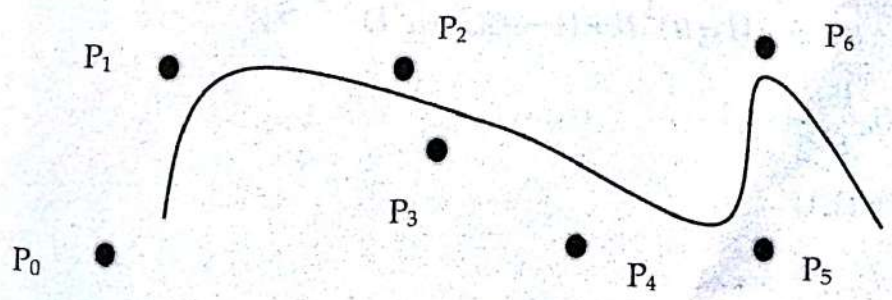


Figure 6.13: B-spline Curve

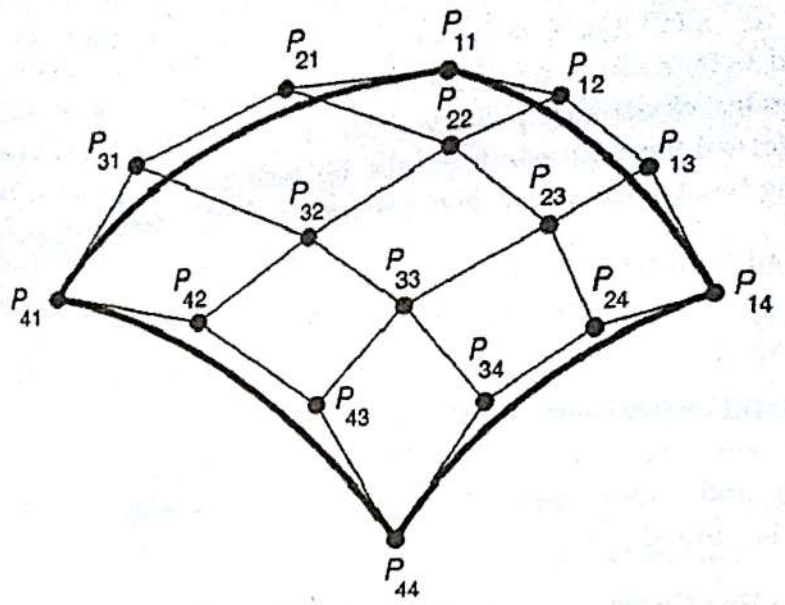


Figure 6.14: B-spline Surface

Example 6.2: Construct the B-spline curve of order 4 and with 4 polygon vertices A(1,1), B(2,3), C(4,3) and D(6,2).

Solution:

Here n=3 and k=4, we have open uniform knot vector as X=[000 0 1 1 1 1] and we have basis functions for various parameters are as follows:

$0 \le u < 1$

$N_{4,1}(u) = 1;$ $\quad N_{i,1}(u) = 0, i \neq 4$

$N_{3,2}(u) = (1-u);$ $\quad N_{4,2}(u) = u, \quad N_{i,2}(u) = 0, i \neq 3,4$

$N_{2,3}(u) = (1-u)^2;$ $\quad N_{3,3}(u) = 24(1-u);$

$N_{4,3}(u) = u^2;$ $\quad N_{i,3}(u) = 0, i \neq 2,3,4$

$N_{1,4}(u) = (1-t)^3;$ $\quad N_{2,4}(u) = u(1-u)^2 + 2u(1-u)^2 = 3u(1-u)^2;$

$N_{3,4}(u) = 2u^2(1-u) + (1-u)u^2 = 3u^2(1-u);\ N_{4,4}(u) = u^3$

The parametric B-Spline is

$$P(u) = AN_{1,4}(u) + BN_{2,4}(u) + CN_{3,4}(u) + DN_{4,4}(u)$$

$$\therefore P(u) = (1-u)^3 A + (1-u)^2 B + (1-u)C + u^3 D$$

Let us take, $\quad u = 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1$

$$\therefore P(0) = A = (1,1)$$

$$\therefore P\left(\frac{1}{4}\right)=\left(1-\frac{1}{4}\right)^3 A+3\times\frac{1}{4}\left(1-\frac{1}{4}\right)^2 B+3\left(\frac{1}{4}\right)^2\left(1-\frac{1}{4}\right)C+\left(\frac{1}{4}\right)^3 D$$

$$=\frac{27}{64}(1,1)+\frac{27}{64}(2,3)+\frac{9}{64}(4,3)+\frac{1}{64}(6,2)$$

$$=\left(\frac{27}{64}\times 1,+\frac{27}{64}\times 2+\frac{9}{64}\times 4+\frac{1}{64}\times 6,\ \frac{27}{64}\times 1,+\frac{27}{64}\times 3+\frac{9}{64}\times 3+\frac{1}{64}\times 2\right)$$

$$=\left(\frac{123}{64},\frac{137}{64}\right)=(1.9218,\ 2.14)$$

$$\therefore P\left(\frac{1}{2}\right)=\left(1-\frac{1}{2}\right)^3 A+3\times\frac{1}{2}\left(1-\frac{1}{2}\right)^2 B+3\left(\frac{1}{2}\right)^2\left(1-\frac{1}{2}\right)C+\left(\frac{1}{2}\right)^3 D$$

$$=\frac{1}{8}(1,1)+\frac{3}{8}(2,3)+\frac{3}{8}(4,3)+\frac{1}{8}(6,2)$$

$$=\left(\frac{1}{8}\times 1,+\frac{3}{8}\times 2+\frac{3}{8}\times 4+\frac{1}{8}\times 6,\ \frac{1}{8}\times 1,+\frac{3}{8}\times 3+\frac{3}{8}\times 3+\frac{1}{8}\times 2\right)$$

$$=\left(\frac{25}{8},\frac{21}{8}\right)=(3.125,\ 2.625)$$

$$\therefore P\left(\frac{3}{4}\right)=\left(1-\frac{3}{4}\right)^3 A+3\times\frac{3}{4}\left(1-\frac{3}{4}\right)^2 B+3\left(\frac{3}{4}\right)^2\left(1-\frac{3}{4}\right)C+\left(\frac{3}{4}\right)^3 D$$

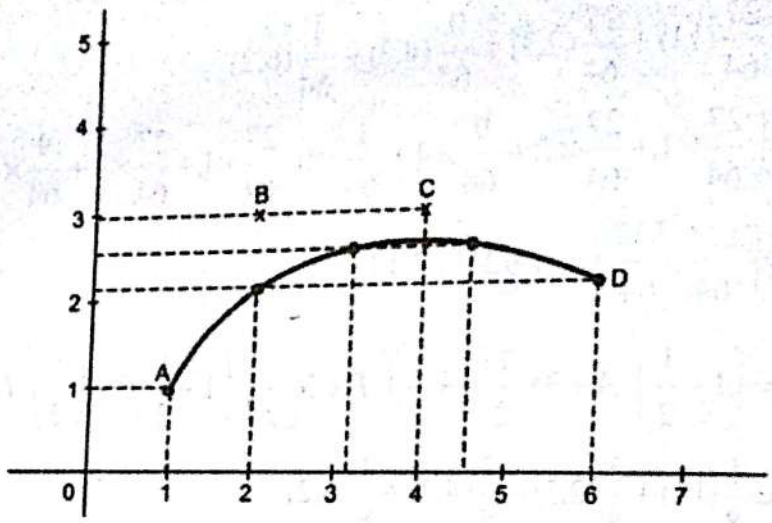
$$=\frac{1}{64}(1,1)+\frac{9}{64}(2,3)+\frac{27}{64}(4,3)+\frac{27}{64}(6,2)$$

$$=\left(\frac{1}{64}\times 1,+\frac{9}{64}\times 2+\frac{27}{64}\times 4+\frac{27}{64}\times 6,\ \frac{1}{64}\times 1,+\frac{9}{64}\times 3+\frac{27}{64}\times 3+\frac{27}{64}\times 2\right)$$

$$=\left(\frac{289}{64},\frac{163}{64}\right)=(4.5156,\ 2.5468)$$

$$\therefore P(1)=D=(6,2)$$

The figure below shows the calculated points of the B-spline curve and curve passing through it.



Quadric Surface

The quadric surfaces are described with second degree equations (quadratics). They include spheres, ellipsoids, tori, parabolids, and hyperboloids.

Sphere

In Cartesian coordinates, a spherical surface with radius r centered on the coordinate's origin is defined as the set of points (x, y, z) that satisfy the equation.

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

The spherical surface can be represented in parametric form by using latitude and longitude angles

$$x = r\cos\phi\cos\theta, \quad \frac{-\pi}{2} \le \phi \le \frac{\pi}{2}$$

$$y = r\cos\phi\sin\theta, \quad -\pi \le \phi \le \pi$$

$$z = r\sin\phi$$

The parameter representation in above equation provides a symmetric range for the angular parameter θ and ϕ.

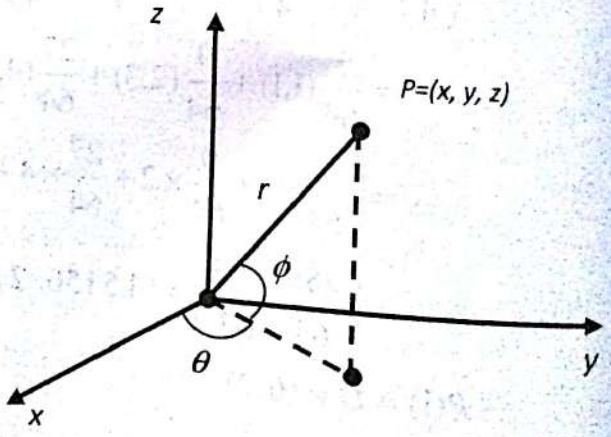


Figure 6.15: Parametric coordinate position (r, θ, ϕ) on the surface of a sphere with radius r.

Ellipsoid

Ellipsoid surface is an extension of a spherical surface where the radius in three mutually perpendicular directions can have different values. The Cartesian representation for points over the surface of an ellipsoid centered on the origin is

$$\left(\frac{x}{r_x}\right)^2+\left(\frac{y}{r_y}\right)^2+\left(\frac{z}{r_z}\right)^2=1$$

The parametric representation for the ellipsoid in terms of the latitude angle ϕ and the longitude angle θ is

$$x = r_x \cos\phi\cos\theta,\ \frac{-\pi}{2} \le \phi \le \frac{\pi}{2}$$

$$y = r_y \cos\phi\sin\theta,\ -\pi \le \phi \le \pi$$

$$z = r_z \sin\phi$$

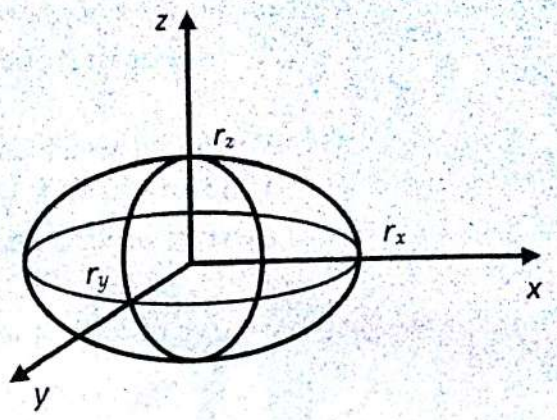


Figure 6.16: An ellipsoid with radii r_x, r_y and r_z centered on the coordinate origin.

Exercise

1. Discuss the 3D object representation in detail.
2. List out the various representation schemes used in three dimensional objects.
3. Explain the different 3D representations in detail.
4. Define polygon. What are the different types of polygons? Explain with examples.
5. Explain in detail about polygon table along with its importance.
6. Explain in detail about plain equation method.
7. Set up a procedure for establishing polygon tables for any input set of data points defining an object.
8. What is polygon mesh? Explain the application of polygon mesh with example.
9. Why polygon description is consider as standard graphics objects?
10. How curve be generated? Explain it with suitable algorithm.
11. What is curve? Explain the types of curve briefly.
12. Explain about parametric cubic curve.
13. Explain Hermite curve with suitable example.
14. Describe the properties of Bezier Curve.
15. Explain the advantages and disadvantages of B spline surface over Bezier surfaces.
16. What are the advantages of B spline over Bezier curve? List the properties of B-spline curve.
17. Model the Bezier curve. Explain the importance of Bezier curve in graphical modeling.
18. Show that the Bezier curve always touches the starting point (for u = 0) and the ending point (for u = 1)
19. Differentiate between periodic B-spline curves and non-periodic B-spline curves.
20. What do you mean by quadratic surface? Explain Sphere and Ellipsoid.

Numerical

1. Construct the polygon table for a object with six vertex, eight edge and three surface.
2. Construct the Bezier curve of order 3 and 4 polygon vertices A(1, 1) B(2, 3) C(4, 3) and D(6, 4).
3. Find the points on the Bezier curve which has starting and ending points $P_0(2, 3)$ and $P_3(4, 3)$ and is controlled by $P_1(5, 6)$ and $P_2(7, 1)$ for u=0.9.
4. Construct the Bezier curve of order 3 and with 4 polygon vertices vertices A(1, 1), B(2, 3), C(4, 3) and D(6, 3).
5. Construct the B-spline curve of order 4 and with 4 polygon vertices A(1, 1), B(2, 3), C(4, 3) and D(6, 5).

❑❑❑

Chapter

7

Solid Modeling

STRUCTURE

7.0 Introduction

7.1 Sweep, Boundary and Spatial-Partitioning Representation

7.2 Binary Space Partition Trees (BSP)

7.3 Octree Representation

Introduction

Solid modeling techniques are based on informationally complete, valid and unambiguous representation of objects. Simply stated, a complete geometric data representation of an object is one that enables points in space to be classified relative to the object, if it is inside, outside, or on the object. **Solid modeling** (or **modeling**) is a consistent set of principles for mathematical and computer modeling of three-dimensional solids. Solid modeling is distinguished from related areas of geometric modeling and computer graphics by its emphasis on physical fidelity. Together, the principles of geometric and solid modeling form the foundation of 3D-computer-aided design and in general support the creation, exchange, visualization, animation, interrogation, and annotation of digital models of physical objects.

Figure 7.1: Solid Modeling

The use of solid modeling techniques allows for the automation of several difficult engineering calculations that are carried out as a part of the design process. Simulation, planning, and verification of processes such as machining and assembly were one of the main catalysts for the development of solid modeling. More recently, the range of supported manufacturing applications has been greatly expanded to include sheet metal manufacturing, injection molding, welding, pipe routing etc. Solid modeling research and development has effectively addressed many of these issues, and continues to be a central focus of computer-aided engineering.

User input required to create solid models on existing CAD/CAM systems depends on both the internal representation scheme used by each system as well as the user interface. It is crucial to distinguish between the user interface and the internal data representation of a given CAD/CAM system. The two are quite separate aspects of the systems and can be linked together by software that is transparent to the user. For example, a system that has a B-rep (boundary representation) internal data representation may use a CSG (constructive solid geometry)-oriented user interface; that is, input a solid model by its primitives. Most systems use the building-block approach (CSG oriented) and sweep operations as the basis for user interface.

The notion of solid modeling as practiced today relies on the specific need for informational completeness in mechanical geometric modeling systems, in the sense that any computer model should support all geometric queries that may be asked of its corresponding physical object. The requirement implicitly recognizes the possibility of several computer representations of the same physical object as long as any two such representations are consistent. It is impossible to computationally verify informational completeness of a representation unless the notion of a physical object is defined in terms of computable mathematical properties and independent of any particular representation. Such reasoning led to the development of the modeling paradigm that has shaped the field of solid modeling as we know it today.

All manufactured components have finite size and well behaved boundaries, so initially the focus was on mathematically modeling rigid parts made of homogeneous isotropic material that could be added or removed. These postulated properties can be translated into properties of subsets of three-dimensional Euclidean space. The two common approaches to define solidity rely on point-set topology and algebraic topology respectively. Both models specify how solids can be built from simple pieces or cells.

Solid representation schemes

Based on assumed mathematical properties, any scheme of representing solids is a method for capturing information about the class of semi-analytic subsets of Euclidean space. This means all representations are different ways of organizing the same geometric and topological data in the form of a data structure. All representation schemes are organized in terms of a finite number of operations on a set of primitives. Therefore, the modeling space of any particular representation is finite, and any single representation scheme may not completely suffice to represent all types of solids. For example, solids defined via combinations of regularized Boolean operations cannot necessarily be represented as the sweep of a primitive moving according to a space trajectory, except in very simple cases. This forces modern geometric modeling systems to maintain several representation schemes of solids and also facilitate efficient conversion between representation schemes.

The lists of common techniques used to create or represent solid models are listed below:

- Parameterized primitive instancing
- Spatial occupancy enumeration
- Cell decomposition
- Boundary representation
- Surface mesh modeling
- Constructive solid geometry
- Sweeping
- Implicit representation
- Parametric and feature-based modeling

Parameterized primitive instancing

This scheme is based on motion of families of object, each member of a family distinguishable from the other by a few parameters. Each object family is called **a generic primitive**, and individual objects within a family are called **primitive instances**. For example, a family of bolts is a generic primitive, and a single bolt specified by a particular set of parameters is a primitive instance.

Spatial occupancy enumeration

This scheme is essentially a list of spatial cells occupied by the solid. The cells, also called voxels are cubes of a fixed size and are arranged in a fixed spatial grid (other polyhedral arrangements are also possible but cubes are the simplest). Each cell may be represented by the coordinates of a single point, such as the cell's centroid.

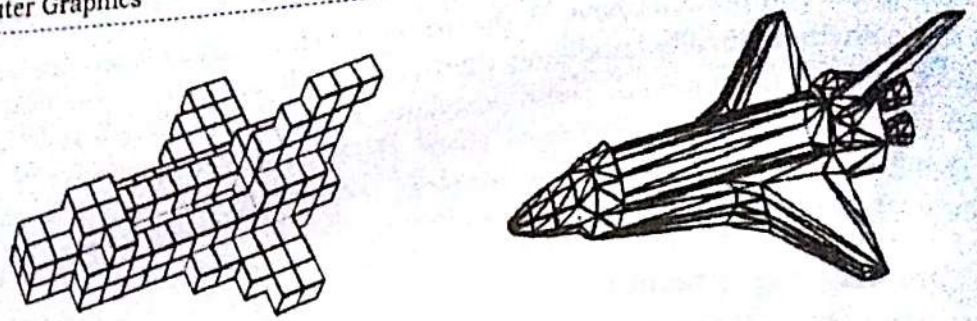


Figure 7.2: Spatial occupancy enumeration

Cell decomposition

A solid can be represented by its decomposition into several cells. Spatial occupancy enumeration schemes are a particular case of cell decompositions where all the cells are cubical and lie in a regular grid. Cell decompositions provide convenient ways for computing certain topological properties of solids such as its connectedness (number of pieces) and genus (number of holes). Cell decompositions in the form of triangulations are the representations used in 3d finite elements for the numerical solution of partial differential equations.

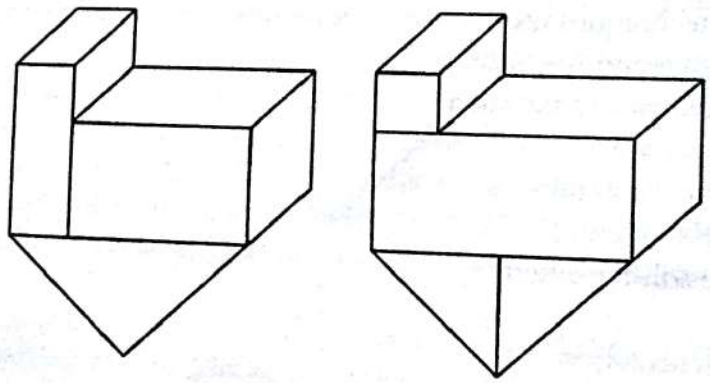


Figure 7.3: Cell decomposition

Surface mesh modeling

Similar to boundary representation, the surface of the object is represented. However, rather than complex data structures, a simple surface mesh of vertices and edges is used. Surface meshes can be structured or unstructured meshes with randomly grouped triangles and higher level polygons.

Constructive solid geometry

Constructive solid geometry (CSG) connotes a family of schemes for representing rigid solids as Boolean constructions or combinations of primitives via the regularized set operations. CSG and boundary representations are currently the most important representation schemes for solids. CSG representations take the form of ordered binary trees where non-terminal nodes represent either rigid transformations (orientation preserving isometrics) or regularized set operations.

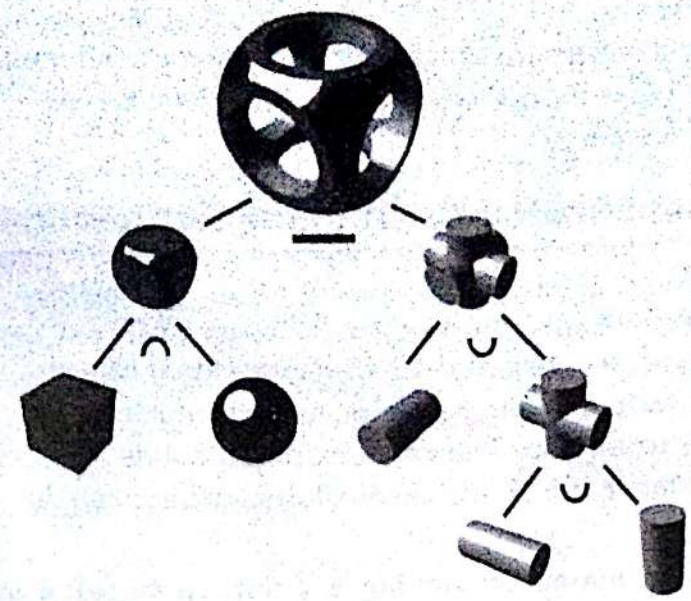


Figure 7.4: Constructive solid geometry (CSG)

Advantages of Solid Modeling

Solid modeling is one of the most important applications of the CAD software and it has been becoming increasingly popular of late. The solid modeling CAD software helps the designer to see the designed object as if it were the real manufactured product. It can be seen from various directions and in various views. This helps the designer to be sure that the object looks exactly as they wanted it to be. It also gives additional vision to the designer as to what more changes can be done in the object.

Process of Making the Solid Models

To make the solid models you have to first make the wire frame model of the object and convert it into 3D view. Thereafter the surfaces are added to the 3D wire model to convert it into 3D solid model. For creating the solid models you need to have special CAD software that can create solid models. One of the most popular CAD software for solid modeling is Solid Works. Its latest version is Solid Works 2009. A number of other CAD software like AutoCAD and others also have features of creating the solid models.

Applications of Solid Modeling

Solid modeling is used not only for creating solid models of machine parts, but also the buildings, electric circuits and even of the human beings. The solid modeling software are being used for a large variety of applications, here are some of them:

- **Engineering:**
  The engineering design professionals use solid modeling to see how the designed product will actually look like. The architects and civil engineers use it to use the layout of the designed building.
- **Entertainment industry:**
  The animation industry has been using solid modeling to create various characters and the movies out of them.

- **Medical industry:**
  Modern imaging scanners are being used to create the solid models of the internal parts of the body. This helps the doctors to visualize specific tissues of the body, designing various medical devices etc.

Sweep, Boundary and Spatial-Partitioning Representation

Sweep Representation

Solid models can also be built up from **sweeps**. Schemes based on sweep representation are useful in creating solid models of **two-and-a-half-dimensional objects**. The class of two-and-a-half-dimensional objects includes both solids of uniform thickness in a given direction and axisymmetric solids. The former are known as extruded solids and are created via linear or translational sweep; the latter are solids of revolution which can be created via rotational sweep.

Sweeping is based on the notion of moving a point, curve, or a surface along a given path. There are three types of sweep: **linear, nonlinear, and hybrid sweeps**. In linear sweep, the path is a linear or circular vector described by a linear, most often parametric, equation while in nonlinear sweep, the path is a curve described by a higher-order equation (quadratic, cubic, or higher). Hybrid sweep combines linear and/or nonlinear sweep via set operations and is, therefore, a means of increasing the modeling domain of sweep representations.

Linear sweep can be divided further into **translational and rotational sweep**. In translational sweep, a planar two-dimensional point set described by its boundary (or contour) can be moved a given distance in space in a perpendicular direction (called the directrix) to the plane of the set. Nonlinear sweep is similar to linear sweep but with the directrix being a curve instead of a vector. Hybrid sweep tends to utilize some form of set operations.

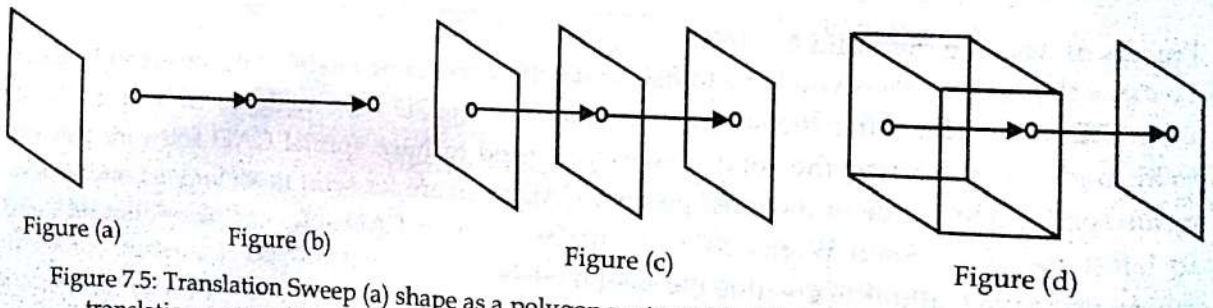


Figure 7.5: Translation Sweep (a) shape as a polygon vertex table (b) sweep path as a sequence of translation vectors (c) translate the shape continue building a vertex table (d) surface table

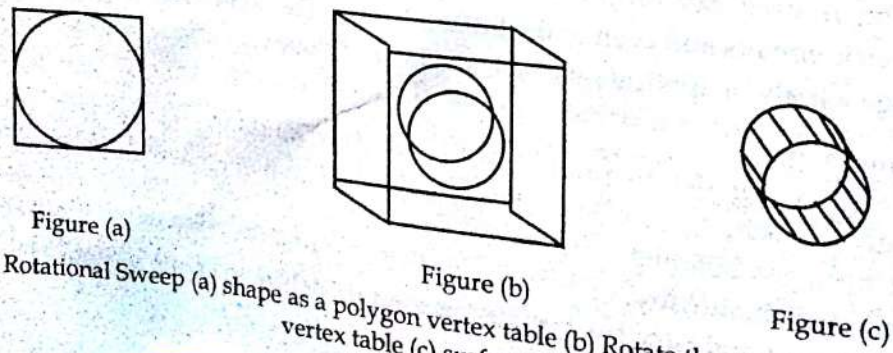


Figure 7.6: Rotational Sweep (a) shape as a polygon vertex table (b) Rotate the shape continue building a vertex table (c) surface table

Boundary Representation (B – rep)

Boundary representation is one of the two most popular and widely used schemes to create solid models of physical objects. A B-rep model or boundary model is based on the topological notion that a physical object is bounded by a set of faces. These faces are regions or subsets of closed and orientable surface. A closed surface is one that is continuous without breaks. An orientable surface is one in which it is possible to distinguish two sides by using the direction of the surface normal to point to the inside or outside of the solid model under construction. Each face is bounded by edges and each edge is bounded by vertices. Thus, topologically, a boundary model of an object is comprised of faces, edges, and vertices of the object linked together in such a way as to ensure the topological consistency of the model.

Advantages and disadvantages of B-rep

The B-rep scheme is very popular and has a strong history in computer graphics because it is closely related to traditional drafting. Its main advantage is that it is very appropriate to construct solid models of unusual shapes that are difficult to build using primitives.

Another major advantage is that it is relatively simple to convert a B-rep model into a wireframe model because the model's boundary definition is similar to the wireframe definition.

One of the major disadvantages of the boundary model is that it requires large amounts of storage because it stores the explicit definitions of the model boundaries. It is also a verbose scheme- more verbose than CSG. The model is defined by its faces, edges, and vertices which tend to grow fairly fast for complex models. If B-rep systems do not have a CSG-compatible user interface, then it becomes slow and inconvenient to use Euler operations in a design and production environment.

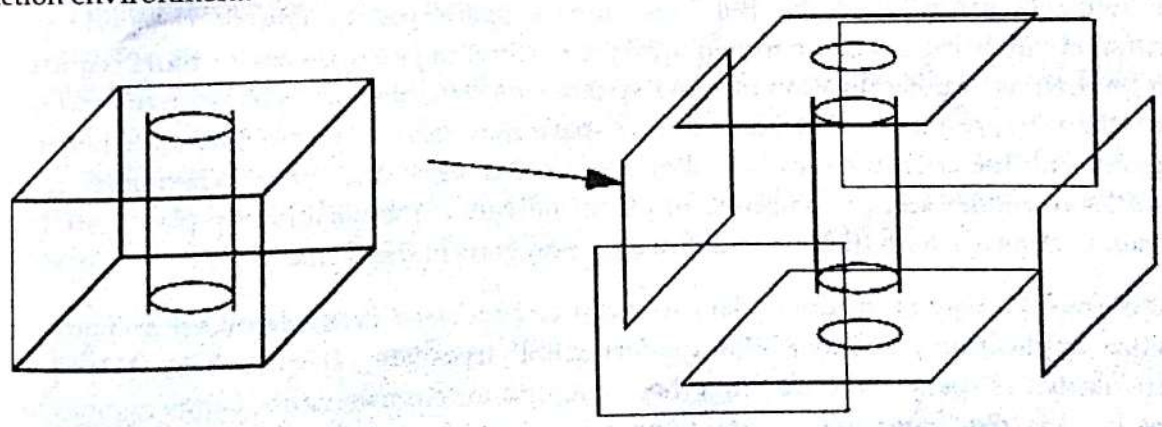


Figure 7.7: Solids Boundary Representation (B-rep) having 7 faces, 8 vertices and 14 Edges

Spatial-Partitioning Representation

Spatial-partitioning representations describe objects as collections of adjoining nonintersecting solids. Spatial-partitioning creates collections of solids that may or may not be the same type as the original object. Spatial-partitioning creates collections of solids that are like building blocks and can vary in type, size, position, parameterization, and orientation. Solid objects can be formed with spatial-partitioning using cell decomposition, spatial-occupancy enumeration, octrees and quadtrees, etc.

Spatial partitioning is particularly important in computer graphics, especially heavily used in ray tracing, where it is frequently used to organize the objects in a virtual scene. A typical scene may contain millions of polygons. Performing a ray/polygon intersection test with each would be a very computationally expensive task. Recursively partitioning space using planes in this way produces a BSP tree, one of the most common forms of space partitioning.

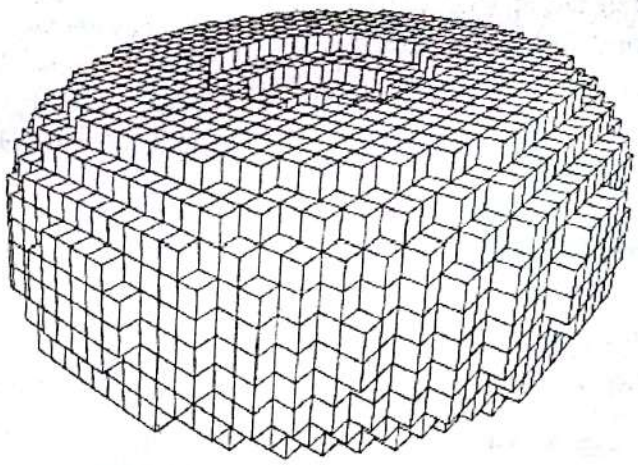


Figure 7.8: Spatial-partitioning Representation Voxels

Binary Space Partition Trees (BSP)

Binary Space Partition Trees (or BSP trees for short) where introduced by **Fuchs, Kedem,** and **Naylor** around 1980.Many 3D modeling and rendering programs utilize a Binary Space Partition tree (BSP tree) to make rendering go faster. Sometimes these programs will even have customizable settings for the BSP tree. Binary space partitioning is a generic process of recursively dividing a scene into two until the partitioning satisfies one or more requirements. It can be seen as a generalization of other spatial tree structures such as *k*-d trees and quadtrees, one where hyper planes that partition the space may have any orientation, rather than being aligned with the coordinate axes as they are in *k*-d trees or quadtrees. When used in computer graphics to render scenes composed of planar polygons, the partitioning planes are frequently chosen to coincide with the planes defined by polygons in the scene.

A BSP tree is a way of grouping data so it can be processed faster. It's used in many computer science applications, not just VFX rendering.BSP trees are often used by 3D video games, particularly first-person shooters and those with indoor environments. Game engines using BSP trees include the Doom, Quake, and Source engines. The grouping process starts with data of some kind, and divides it into two parts. Then the two parts are divided into two parts, and so on until the parts at the end contain the smallest piece of working data. As a very simple example, let's take the word "graphics" and partition it with a BSP tree. For this example, a single letter of the alphabet is the smallest possible element.

BSP trees are very useful for real time interaction with displays of static images. Before the static scene is rendered, the BSP tree is calculated. BSP trees can be traversed very quickly (linear time) for hidden surface removal and shadow casting. With some work, BSP trees can be modified to handle dynamic events in a scene.

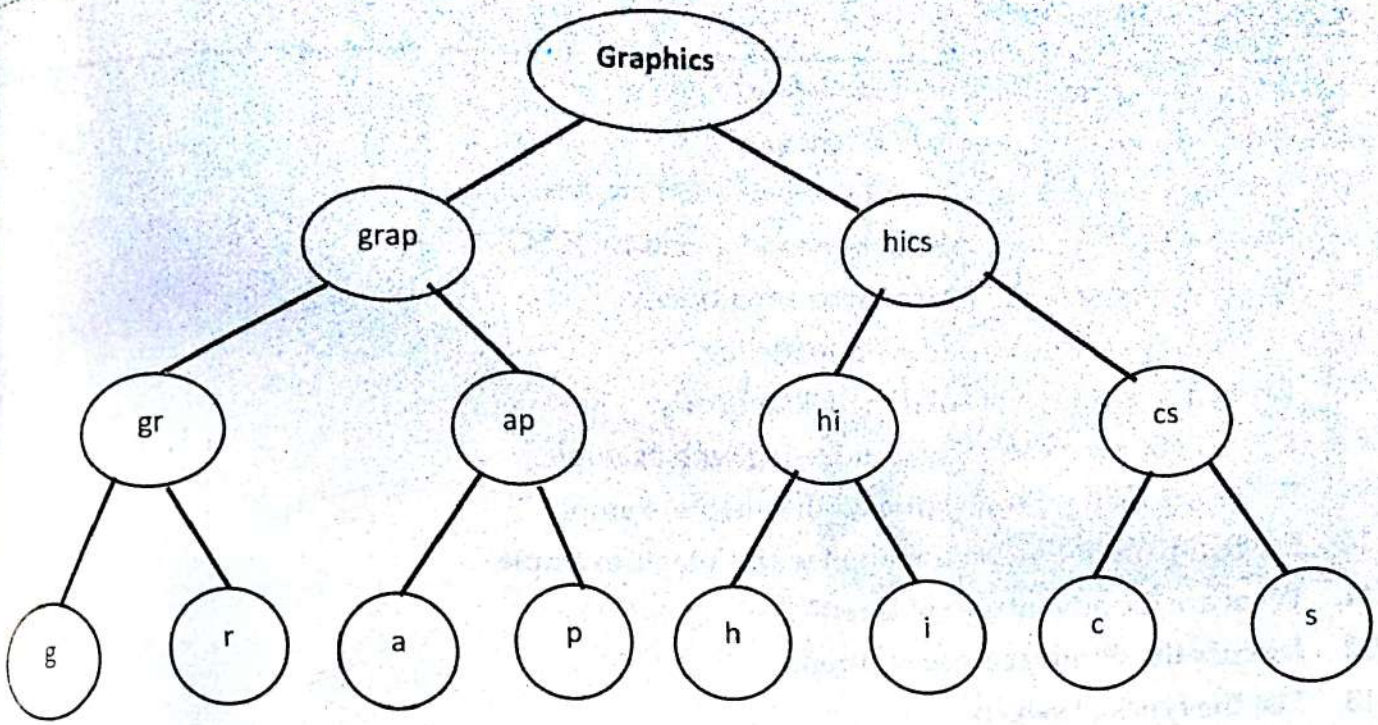


Figure 7.9: Simple BSP tree

Octree Representation

Octree are three dimensional quadtrees, its three dimensions are recursively subdivided into octants and have quadrants. The number of nodes in a quadtree or octree is proportional to the object's perimeter or surface, respectively because subdivision occurs only from the need to represent an object's boundary. Therefore, subdivision only occurs in those quadrants where a boundary passes. Boolean set operators can also apply to both quadtrees and octrees by traversing the two trees in parallel. Individual partitions of 3D space are called Voxels (Volume Elements). Octrees are hierarchical tree structures used to represent solid objects Octrees are particularly useful in applications that require cross sectional views – for example medical applications. Octrees are typically used when the interior of objects is important. Octrees are usually applied in raycasting and shadow casting.

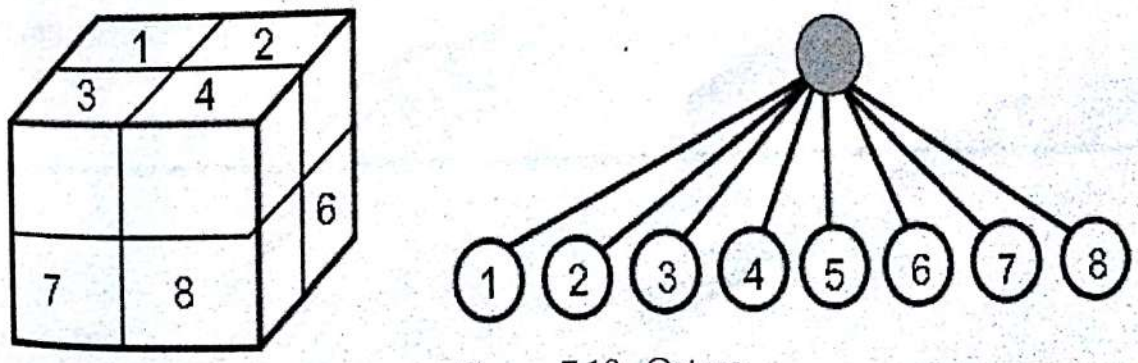


Figure 7.10: Octree

Exercise

1. What do you mean by solid modeling?
2. What are solid representation schemes?
3. List common techniques used to create solid models.
4. Write short notes on constructive solid geometry (CSG).
5. Explain spatial occupancy enumeration briefly.
6. Discuss the advantages of solid modeling.
7. Explain application of solid modeling briefly.
8. Explain the process for solid modeling with example.
9. Explain Sweep representation with suitable example.
10. Explain Boundary representation with suitable example.
11. What are the advantages of B-rep?
12. Discuss the disadvantages of B-rep.
13. List the types of sweep.
14. Discuss the linear sweep along with its types.
15. Differentiate between translational and rotational sweep.
16. Explain Spatial-Partitioning representation with suitable example.
17. Explain Binary Space Partitioning Trees (BSP) with suitable example.
18. Explain Octree representation with suitable example.
19. Differentiate between Binary Space Partitioning Tree and Octree representation.
20. What are the differences between Boundary representation and spatial-partitioning representation?

❑❑❑

Chapter

8

Visible Surface Detection

STRUCTURE

7.0 Introduction
7.1 Input Devices
7.2 Output Devices
7.3 Properties of CRT

Introduction

It is the process of identifying those parts of a scene that are visible from a chosen viewing position. There are numerous algorithms for efficient identification of visible objects for different types of applications. These various algorithms are referred to as visible-surface detection methods. Sometimes these methods are also referred to as hidden-surface elimination methods.

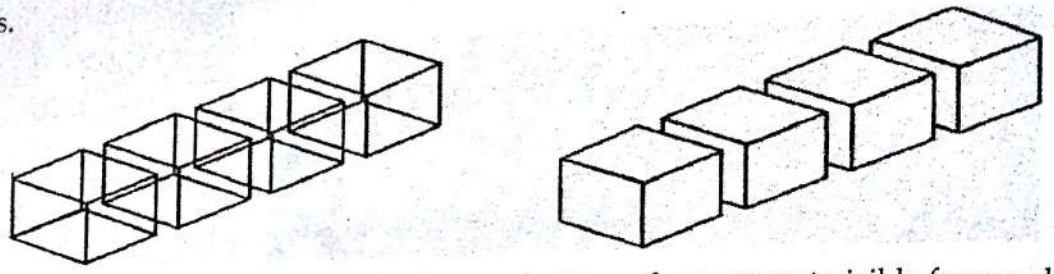


Figure 8.1: Identification and Removal of parts of picture those are not visible from a chosen viewing position

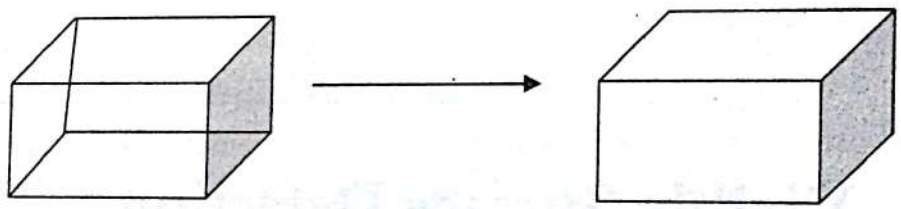


Figure 8.2: Hidden surface removed

When we view a picture containing non-transparent objects and surfaces, then we cannot see those objects from views which are behind from objects closer to eye. We must remove these hidden surfaces to get a realistic screen image. The identification and removal of these surfaces is called Hidden-surface problem. There are two approaches for removing hidden surface problems.

- Object-Space method and
- Image-space method

Object-space Methods

An object-space method compares objects and parts of objects to each other within the scene definition to determine which surfaces are visible. It is suitable for scene with small number of objects and objects with simple geometrical descriptions. It is easy to implement but slow than other methods. Mainly object-space method is implemented in physical coordinate system.

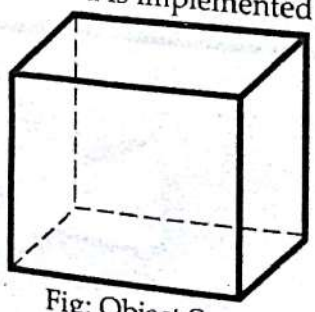


Fig: Object Space

Basic Procedure

For each object in the scene do

1. Begin
   - Determine those parts of the object whose view is unobstructed by other parts of it or any other object with respect to the viewing specification.
   - Draw those parts in the object color.
2. End.

Characteristics of object-space method

- Compare each object with all other objects to determine the visibility of the object parts.
- If there are n objects in the scene, complexity = $O(n^2)$
- Calculations are performed at the resolution in which the objects are defined (only limited by the computation hardware).
- Process is unrelated to display resolution or the individual pixel in the image and the result of the process is applicable to different display resolutions.
- Display is more accurate but computationally more expensive as compared to image space methods because due to the possibility of intersection between surfaces.
- Suitable for scene with small number of objects and objects with simple relationship with each other.

The most popular methods for object-space method are listed below:

1. Back-Face Removal
2. Depth Sorting (Painter Algorithm)
3. Area Subdivision (Quadtrees and Octrees)
4. Binary Space Partition Trees

Image-space method

Image space method is implemented in the screen coordinate system in which the objects are viewed. In an image-space algorithm, visibility is decided point by point at each pixel Position on the view plane. Most hidden line/surface algorithms use image-space methods.Color calculation in object space algorithms is done only one time and is retained by it but in image space algorithm the calculation once done is over written later.

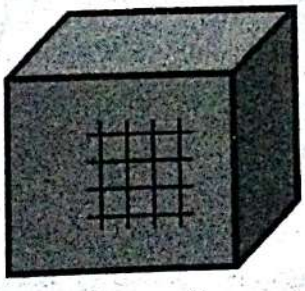


Fig: Image Space

Basic Procedure

For each pixel in the image do

1. Begin
   - Determine the object closest to that pixel,
   - Draw the pixel in the object color.
2. End.

Characteristics of image space method

- For each pixel, examine all n objects to determine the one closest to the viewer.
- If there are p pixels in the image, complexity depends on n and p (O(np))
- Accuracy of the calculation is bounded by the display resolution.
- A change of display resolution requires re-calculation.

The most popular methods for image-space method are listed below:

1. Depth Buffer (Z-buffer)
2. Scan-line
3. Depth sorting (Painter Algorithm)

Back face Detection (Plane Equation method)

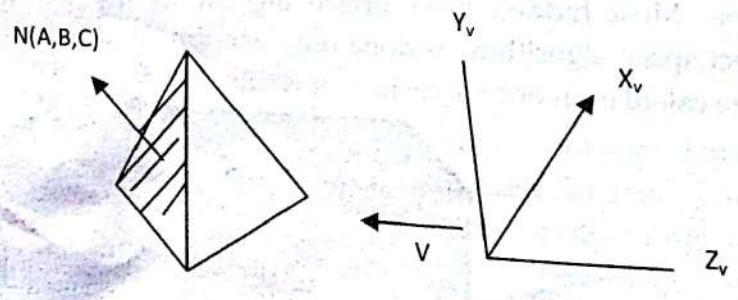
Back-Face detection, also known as **Plane Equation method,** is an object space method in which objects and parts of objects are compared to find out the visible surfaces. Let us consider a triangular surface that whose visibility needs to decide. The idea is to check if the triangle will be facing away from the viewer or not. If it does so, discard it for the current frame and move onto the next one. Each surface has a normal vector. If this normal vector is pointing in the direction of the center of projection, then it is a front face and can be seen by the viewer. If this normal vector is pointing away from the center of projection, then it is a back face and cannot be seen by the viewer. It is commonly known as back-face detection of a polyhedron is based on the "inside-outside" tests.

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

$$Ax + By + Cz + D < O$$

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

If V is the vector in viewing direction from the eye position then this polygon is a back face if, V.N>0



In the equation $Ax + By + Cz + D = 0$, if A, B, C remains constant, then varying value of D results in a whole family of parallel planes. One of which (D = 0) contains the origin of the co-ordinates system and,

- If D > 0, plane is behind the origin (Away from observer)
- If D < 0, plane is in front of origin (towards the observer)

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

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

Limitations

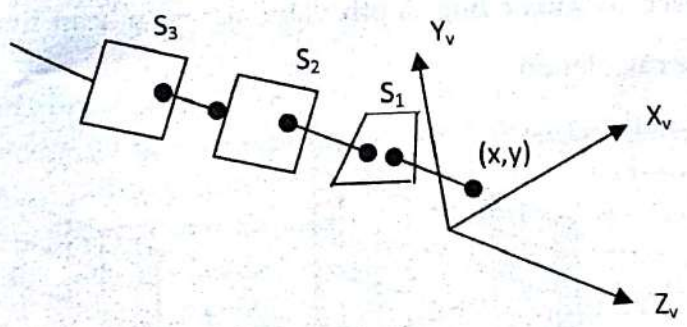
- This method works fine for convex polyhedra, but not necessarily for concave polyhedra.
- This method can only be used on solid objects modeled as a polygon mesh.

Depth Buffer (Z-Buffer) Method

Depth Buffer Method is the commonly used image-space method for detecting visible surface. It is also known as z-buffer method. This method is developed by Cut mull. It is an image-space approach. The basic idea is to test the Z-depth of each surface to determine the closest (visible) surface. In this method each surface is processed separately one pixel position at a time across the surface. The depth values for a pixel are compared and the closest (smallest z) surface determines the color to be displayed in the frame buffer. It is applied very efficiently on surfaces of polygon. Surfaces can be processed in any order. To override the closer polygons from the far ones, two buffers named **frame buffer** and **depth buffer,** are used.

- **Depth buffer** is used to store depth values for (x, y) position, as surfaces are processed ($0 \leq \text{depth} \leq 1$).
- The **frame buffer** is used to store the intensity value of color value at each position (x, y).

The z-coordinates are usually normalized to the range [0, 1]. The 0 value for z-coordinate indicates back clipping pane and 1 value for z-coordinates indicates front clipping pane.



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

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

Algorithm

1. Start
2. Set the buffer values
   - Depthbuffer (x, y) = 0
   - Framebuffer (x, y) = background color
3. Process each polygon (One at a time)
   3.1. For each projected (x, y) pixel position of a polygon, calculate depth z.
      3.1.1. If Z >depthbuffer (x, y)
         - Compute surface color,
         - set depthbuffer (x, y) = z,
         - framebuffer (x, y) = surfacecolor (x, y)
4. Stop

After all surfaces are processed, the depth buffer contains the depth value of the visible surface and refresh buffer contains the corresponding intensity values for that surface. The depth value of the surface position (x, y) is calculated by plane equation of surface.

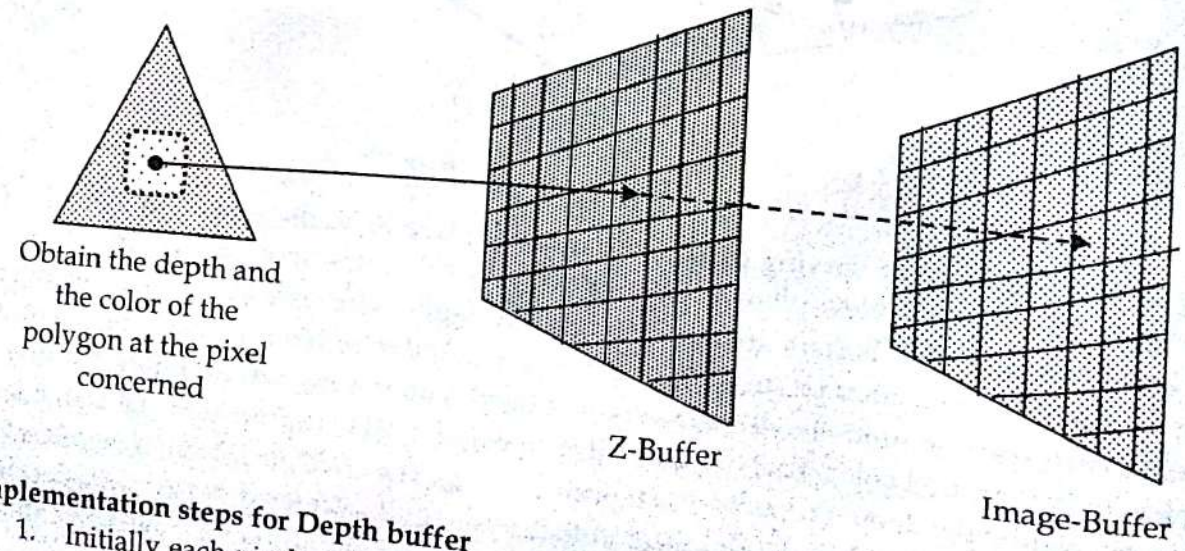
$$Z = \frac{-Ax - By - D}{C}$$

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

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

$$\Rightarrow Z' = Z - A/C \quad \ldots\ldots\ldots\ldots\ldots \quad (1)$$

$-A/C$ is constant for each surface so succeeding depth value across a scan line are obtained from preceding values by simple calculation.



Implementation steps for Depth buffer

1. Initially each pixel of the z-buffer is set to the maximum depth value (the depth of the back clipping plane).
2. The image buffer is set to the background color.
3. Surfaces are rendered one at a time.

4. For the first surface, the depth value of each pixel is calculated.
5. If this depth value is smaller than the corresponding depth value in the z-buffer (i.e. it is closer to the view point), both the depth value in the z-buffer and the color value in the image buffer are replaced by the depth value and the color value of this surface calculated at the pixel position.
6. Repeat step 4 and 5 for the remaining surfaces.
7. After all the surfaces have been processed, each pixel of the image buffer represents the color of a visible surface at that pixel.

Advantages

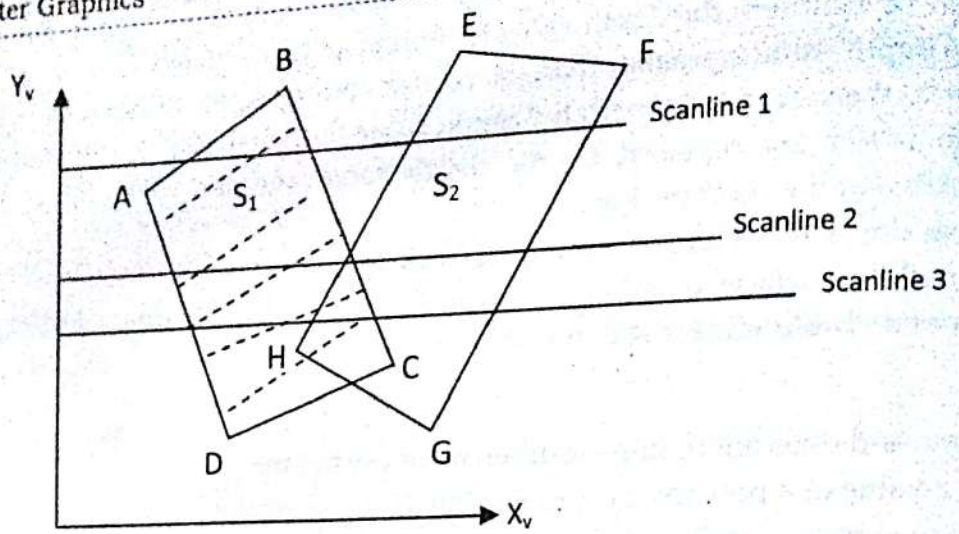
- Simple and does not require additional data structures.
- The z-value of a polygon can be calculated incrementally.
- No pre-sorting of polygons is needed.
- No object-object comparison is required.
- Can be applied to non-polygonal objects
- For large images, the algorithm could be applied to, eg., the 4 quadrants of the image separately, so as to reduce the requirement of a large additional buffer.
- It reduces the speed problem if implemented in hardware.
- It processes one object at a time.

Disadvantages

- This method requires an additional buffer (if compared with the Depth-Sort Method) and the overheads involved in updating the buffer. So this method is less attractive in the cases where only a few objects in the scene are to be rendered.
- It requires large memory.
- It is time consuming process.

Scan Line Method

This is image-space method for removing hidden surface which is extension of the scan line polygon filling for polygon interiors. In this method, as each scan line is processed, all polygon surfaces intersecting that line are examined to determine which are visible. Across each scan line, depth calculations are made for each overlapping surface to determine which is nearest to the view plane. When the visible surface has been determined, the intensity value for that position is entered into the image buffer. Instead of filling one surface we deal with multiple surfaces here. As each scan line is processed, all polygon surfaces intersecting that line are examined to determine which are visible. We assume that polygon table contains the co-efficient of the plane equation for each surface as well as vertex edge, surface information, intensity information for the surface, and possibly pointers to the edge table.



Working Procedure

Above figure illustrate the scan line method for locating visible portion of surfaces for pixel position along the line. The active list for scan line one contains information from the edge table for edges AB, BC, EH and FG.

For scan line 1, between edges AB and BC, flag (S1) = ON and flag (S2) = OFF. Therefore no depth calculations are necessary and intensity information for surface S1 is entered from the polygon table into the refresh buffer. Similarly, between edge EH and FG flag (S1) = OFF and flag (S2) = ON.

For scan line 2 and 3, the active edge list contains edges AD, EH, BC and FG. Along the scan line 2, from edge AD to edge EH, flag (S1) =ON and flag (S2) = OFF.

Between edges EH and BC, Flag (S1) = ON and flag (S2) = ON. Therefore depth calculation must be made using the plane coefficient for the two surfaces. For this example the depth of surface S1 is assumed to be less than that of S2, So intensity for surface S1 are loaded into the refresh buffer until boundary BC is encountered. Then the flag for the surface S1 goes off. And intensity for surface S2 is stored until edge FG is passed.

Characteristics

- It is an image-space method for identifying visible surfaces. It deals with multiple polygons.
- The scan line method solves the hidden surface problem one scan line at a time. Processing of the scan line start from the top to the bottom of the display.
- This method calculates the z-value for only the overlapping surface which is tested by scan line.
- Each scan line is processed; all polygon surfaces intersecting that line are examined to determine which are visible.
- Across each scan line, depth calculations are made for each overlapping surface to determine which surface is nearest to the view plane.
- When the visible surface has been determined, the intensity value for that position is entered into the refresh buffer.
- It requires an edge table, polygon table, active edge list and flag.
- **Edge table contains:** Coordinate end points for each scan line, pointers into the polygon table to identify the surfaces bounded by each line.

- **Polygon table contains:** Coefficients of plane equations for each surfaces, pointers into the edge table, intensity information of the surfaces.
- **Active edge list contains:** Edges that cross the current scan line, shorted in order of increasing x.
- **Flag** is defined for each surface that is set 'ON' or 'OFF' to indicate whether a scan line is inside or outside of the surface. At the left most boundary surface flag is 'ON' and at right most boundary flag is 'OFF'.

Algorithm

1. Start
2. Established data structure.
   a. Edge table with line endpoints.
   b. Polygon table with plane coefficients, color and pointer to the edge table.
   c. Active edge list sorted in increasing order of x.
   d. A flag for each surface.
3. Repeat for each scan line.
   a. Update AEL for each scan line y.
   b. When flag is ON, enter intensity of that surface to refresh buffer.
   c. When two or more flags are ON, calculate depth and store the intensity of the surface nearest to the view plane.
4. Stop

Pseudo code for Scan line algorithm

1. Start
2. For each scan line do
   Begin
3. For each pixel (x,y) along the scan line
   a. Begin z_buffer(x,y) = 0
   b. Image_buffer(x,y) = background_color
4. End for loop
5. For each polygon in the scene do
6. Begin for each pixel (x,y) along the scan line that is covered by the polygon do
   Begin
7. Compute the depth or z of the polygon at pixel location (x,y).
8. If z <z_buffer(x,y) then
   a. Set z_buffer(x,y) = z
   b. Set Image_buffer(x,y) = polygon's color
9. End for loop
10. Stop

Advantages

- Any number of overlapping polygon surfaces can be processed with this method.
- Deals with transparent, translucent, and opaque surfaces.
- Can be applied to non-polygonal objects.

Disadvantages

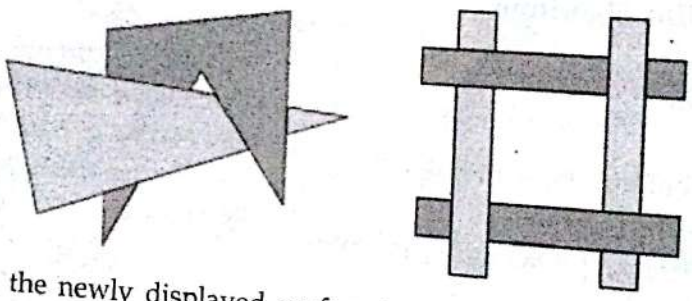
- Depth calculations are performed only when there are overlapping polygons.
- Additional memory buffer is required
- Complex.

Depth Sorting Method

The basic idea of this method is simple. When there are only a few objects in the scene, this method can be very fast. However, as the number of objects increases, the sorting process can become very complex and time consuming. This method uses both object space and image space method. The depth-sorting method performs two basic functions:

- First, the surfaces are sorted in order of decreasing depth.
- Second, the surfaces are scan-converted in order, starting with the surface of greatest depth.

The intensity values for farthest surface are then entered into the refresh buffer. That is farthest polygon is displayed first, then the second farthest polygon, so on, and finally, the closest polygon surface. After all surfaces have been processed, the refresh buffer stores the final intensity values for all visible surfaces. When there are only a few objects in the scene, this method can be very fast. However, as the number of objects increases, the sorting process can become very complex and time consuming. The scan conversion of the polygon surfaces is performed in image space. This method for solving the hidden-surface problem is often referred to as the **painter's algorithm**. The following figure shows the effect of depth sorting:



In this method, the newly displayed surface is partly or completely obscures the previously displayed surface. Essentially, we are sorting the surface into priority order such that surface with lower priority (lower z, far objects) can be obscured by those with higher priority (high z-value).

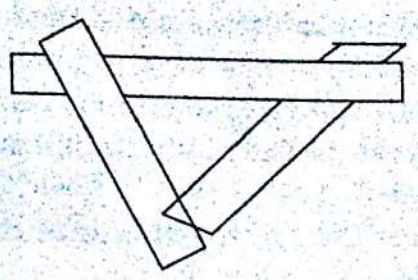
Advantages

- The sorting computation is very fast, so a quick calculation of the image display is possible.
- The finished image data remains object based (sorted polygons), and can be edited in terms of line weight, exploded views, etc.
- It will also print at the highest resolution available on devices such as postscript laser printers.

Limitations

- Certain conditions, such as intersecting polygons or cyclic overlap (A covers part of B, which covers part of C, which covers part of A), are not processed correctly, since parts

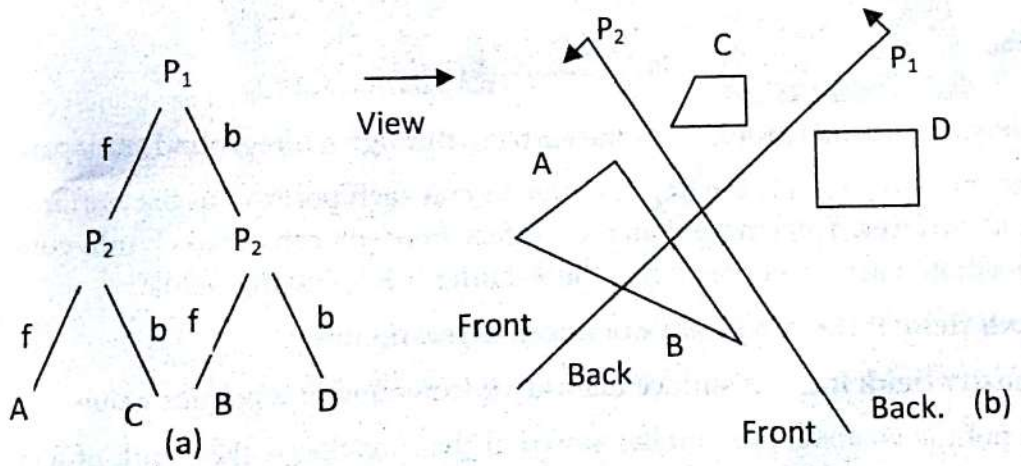
of each should be visible, but the algorithm will always yield a sorted list. The problem described as shown in figure below.



- Different polygons may have same depth.
- The nearest polygon could also be farthest.
- We cannot use simple depth-sorting to remove the hidden-surfaces in the images.

Binary Space Partitioning (BPS) Tree Method

A binary space partitioning (BSP) tree is an efficient method for determining object visibility by painting surfaces onto the screen from back to front as in the painter's algorithm. The BSP tree is particularly useful when the view reference point changes, but object in a scene are at fixed position. Applying a BSP tree to visibility testing involves identifying surfaces that are "inside" or "outside" the partitioning plane at each step of space subdivision relative to viewing direction. It is useful and efficient for calculating visibility among a static group of 3D polygons as seen from an arbitrary viewpoint.



Here plane P_1 partitions the space into two sets of objects, one set of object is back and one set is in front of partitioning plane relative to viewing direction. Since one object is intersected by plane P_1, we divide that object into two separate objects labeled A and B. Now object A&C are in front of P_1, B and D are4 back of P_1. We next partition the space with plane P_2 and construct the binary free as fig (a). In this tree, the objects are represented as terminal nodes, with front object as left branches and behind object as right branches. When BSP tree is complete, we process the tree by selecting surface for displaying in order back to front. So for ground object are painted over back ground objects.

Procedure to build a BSP tree

- Select a polygon as the root of the tree i.e. the first partition plane.
- Partition the object space into two half-spaces (inside and outside of the partition plane determined by the normal to the plane); some object polygon lie in the rear half while the others in the front half with respect to the partition plane.
- If a polygon is intersected by the partition plane, split it into two polygons so that they belong to different half-spaces
- 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.
- Recursively subdivide spaces considering the plane of the children's as partition planes until a subspace contains a single polygon.

A-Buffer method

A-Buffer method is an extension of depth buffer method. A drawback of the depth buffer method is that it can only find one visible surface at each pixel position. That means, it deals only with opaque surface and cannot accumulate intensity values for more than one surface, as is necessary if transparent surfaces are to be displayed.

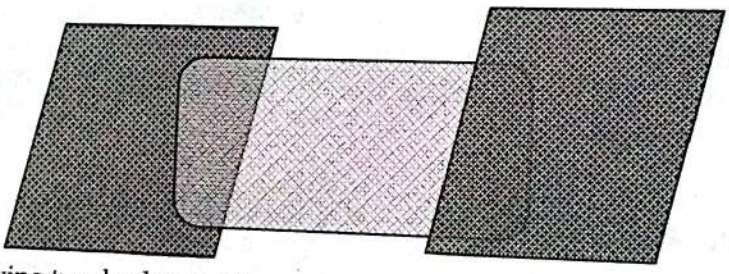


Figure: Viewing two background opaque surfaces through a foreground transparent surface

The A-buffer method expands the depth buffer so that each position in the buffer can reference a linked list of surfaces. Thus more than one surface intensity can be taken into consideration at each pixel position. Each pixel position in the A-buffer has following fields:

- **Depth field:** It stores a positive or negative real numbers
- **Intensity field:** It stores surface intensity information or a pointer value.

If the depth field is positive, the number stored at that position is the depth of a single surface overlapping the corresponding pixel area. If the depth field is negative, this indicates multiple surface contributions to the pixel intensity.

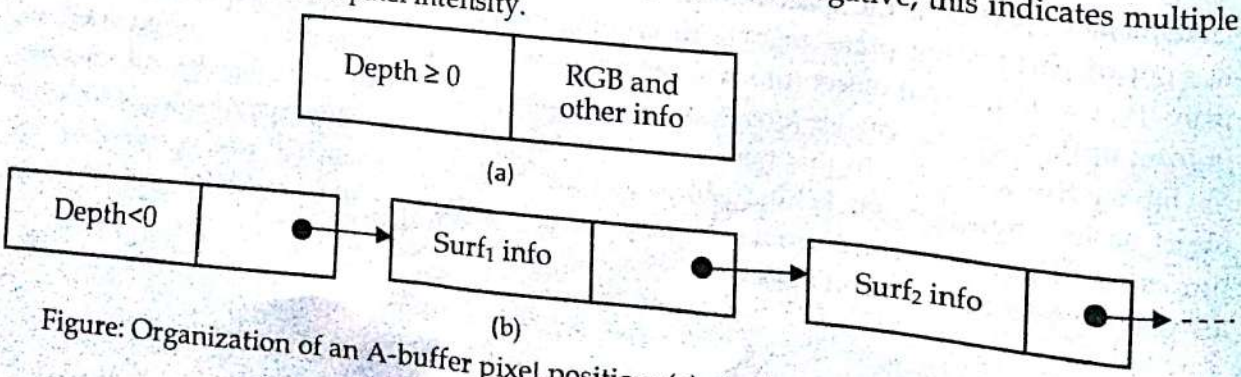


Figure: Organization of an A-buffer pixel position: (a) single-surface (b) multiple surfaces overlap

As shown in the above figure (a), if the value of depth is >= 0, the number stored at that position is the **depth of single surface** overlapping the corresponding pixel area. The 2nd field, i.e., the intensity field then stores the RGB components of the surface color at that point and the percent of pixel coverage.

As shown in the above figure (b), **multiple-surface contributions** to the pixel intensity are indicated by depth < 0. The 2nd field, i.e., the intensity field then stores a pointer to a linked list of surface data.

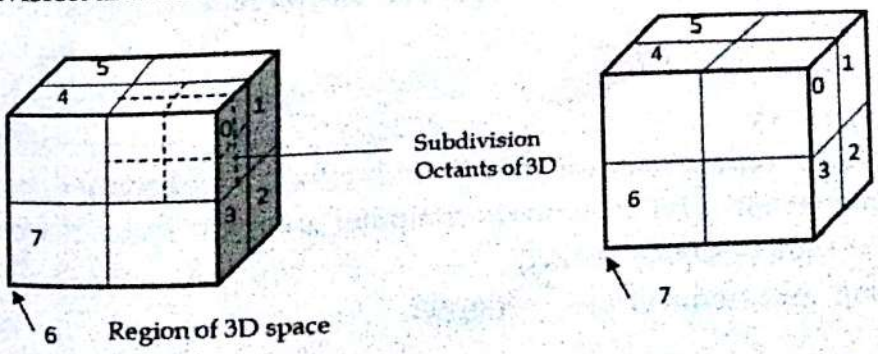
Comparisons between 'Z' buffer and 'A' buffer methods

Z buffer and 'A' buffer are two of the most popular visible surface detection techniques. In fact, 'A' buffer is an extension to Z buffer, which adds anti-aliasing. Typically, 'A' buffer has a better image resolution than Z buffer, because it uses an easily computable Fourier window. However, 'A' buffer is slightly costly than Z buffer.

Octree Method

An Octree is a tree data structure in which each internal node has exactly eight children. Octrees are most often used to partition a three-dimensional space by recursively subdividing it into eight octants. Octrees are the three-dimensional analog of quadtrees. The name is formed from **oct** + tree, but note that it is normally written "octree" with only one "t". Octrees are often used in 3D graphics and 3D game engines.

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



Ray Tracing method

In computer graphics, **ray tracing** is a rendering technique for generating an image by tracing the path of light as pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scanline rendering methods, but at a greater computational cost. This makes ray tracing best suited for applications where the image can be rendered slowly ahead of time, such as in still images and film and television visual effects, and more poorly suited for real-time applications like video games where speed is critical. Ray tracing is capable of simulating a wide variety of optical effects, such as reflection and refraction, scattering, and dispersion phenomena.

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

Characteristics of ray tracing method

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

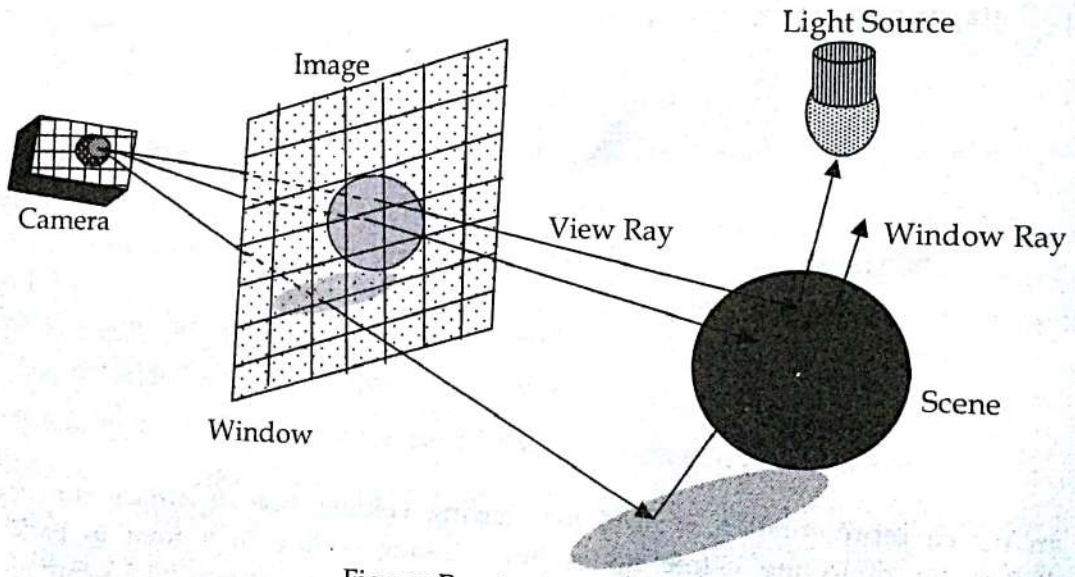


Figure: Ray tracing method

Advantages of ray tracing

1. Conceptually simple. Same basic ray-object intersection code handlers
   - shadows
   - reflections
   - transparency
2. Automatic hidden surface removal (always finds closest intersection)
3. Ray-object intersection is fundamental in computer graphics:
   - line of sight (visibility testing)
   - tracing light - indirect energy transfer

Exercise

1. Explain scan line algorithm for Hidden surface removal.
2. Distinguish between Image Space Method and object space method for visible surface determination.
3. Explain Depth buffer algorithm for Hidden surface removal.
4. What do you mean by hidden surface removal? Describe any hidden surface removal algorithm with suitable example.
5. Discuss the problems of interpolated shading models. How A-buffer method removes the drawback of z-buffer algorithm? Explain.
6. Outline the Z-buffer method. List the advantages and disadvantages of the Z-buffer algorithm.
7. What is the mini-max test used in Z-buffer algorithm? When the mini-max test fails?
8. Explain depth buffer and scan line algorithm for back face detection.
9. Explain painter's algorithm. Also explain depth sorting algorithm of hidden surface removal.
10. Explain BSP method with a suitable example.
11. What difficulties are encountered in implementing the painter's algorithm?
12. Show how the calculations of the intersection of an edge with a scan line can be made incremental as opposed to absolute.
13. Explain BSP tree and octree methods for hidden surface removal.
14. Compare different hidden surface removal methods. How will you eliminate a hidden line?
15. What is edge coherence? Discuss its applications.
16. Why we require hidden surface removal algorithm? Write and explain the scan line method for hidden surface removal. How the amount of computation can be reduced in the method?
17. Assume that one allows 256 depth value levels to be used. Approximately how much memory would a 512 x 512 pixel display require to stove the Z-buffer?
18. What do you mean by edge coherence? What is its significance in depth buffer algorithm?
19. Given the points p1(3, 6, 20), p2(2, 4, 6) and p3(2, 4, 6) and a view point C(0, 0, -10). Determine which point obscure the other when viewed from C?
20. Explain the following algorithms for visible surface detection:
    a. Octree method
    b. Ray casting method
    c. Area sub division method
    d. Z-buffer method
    e. Painter's algorithm
    f. Warnock's algorithm

□□□

Chapter

9

Illumination Models and Surface Rendering Techniques

STRUCTURE

9.0 Introduction

9.1 Basic Illumination Models

9.2 Intensity attenuation and Color consideration

9.3 Transparency

9.4 Shadows

9.5 Polygon (surface) Rendering Method

9.0 Introduction

Illumination model or alighting model is the model for calculating light intensity at a single surface point. Surface rendering is a procedure for applying a lighting model to obtain pixel intensities for all the projected surface positions in a scene. Surface rendering can be performed by applying the illumination model to every visible surface point, or the rendering can be accomplished by interpolating intensities across the surface from a small set of illumination-model calculations.

Light source

Light source means an object that is emitting radian energy. For example, a light bulb, the sun etc. When we view an opaque non-luminous object, we see reflected light from the surface of the object. The total reflected light is the sum of the contribution from the light source and other reflective surfaces as shown in the figure:

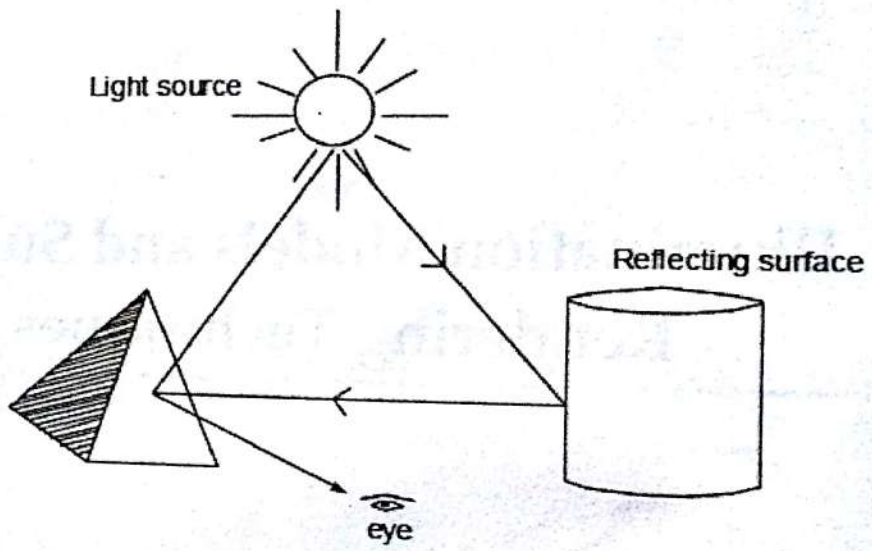


Figure 9.1: Light viewed from an opaque non luminous surface

Light viewed from an opaque non luminous surface is in general a combination of reflected light from a light source and reflection of light reflection from other surfaces. Thus a surface that is not directly exposed to light sources may still be visible if nearby objects are eliminated. Light sources that illuminate on object are of two types:

- Light emitting sources and
- Light reflecting sources

Point Source

When the surface of the object that we want to illuminate is bigger than the surface of light emitting source then we call the light emitting source as a point source. Point source is the simplest light emitter that emits light in equal intensity in all directions e.g. light bulb, sun etc.

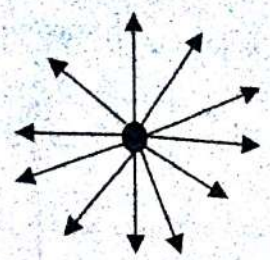


Figure 9.2: A point source

Distributed light source

When the surface of light emitting source is greater than the surface of object then it is called as a distributed light source. All of the rays from a directional/distributed light source have the same direction, and no point of origin. All light rays are parallel. For example, a Fluorescent tube light.

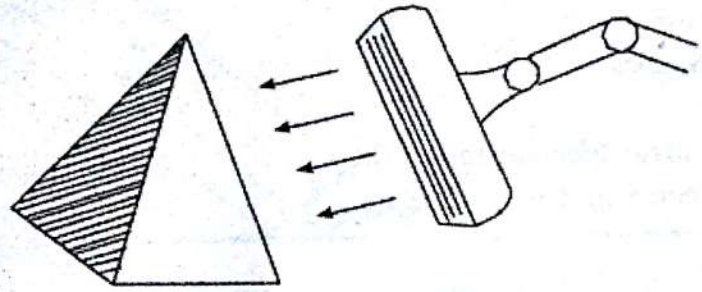


Figure 9.3: Distributed light source

Reflection of light

Reflection is the change in direction of a wave front at an interface between two different media so that the wave front returns into the medium from which it originated. Common examples include the reflection of light, sound and water waves. The law of reflection says that for specular reflection the angle at which the wave is incident on the surface equals the angle at which it is reflected. When light is incident on opaque surface part of it is reflected and part of it is absorbed.

$\therefore \quad I = A+R$

The amount of incident light reflected by a surface depends on the type of material. Shining material reflects more incidents light and dull surface absorbs more of the incident light. For transparent surfaces, some of the incident light will be reflected and some will be transmitted through the material. Rough surface tends to scatter the reflected light in all direction. The scattered light is called **diffuse reflection**. So surface appears equally bright from all viewing directions.

Reflection of light is either **specular** (mirror-like) or **diffuse** (retaining the energy, but losing the image) depending on the nature of the interface.

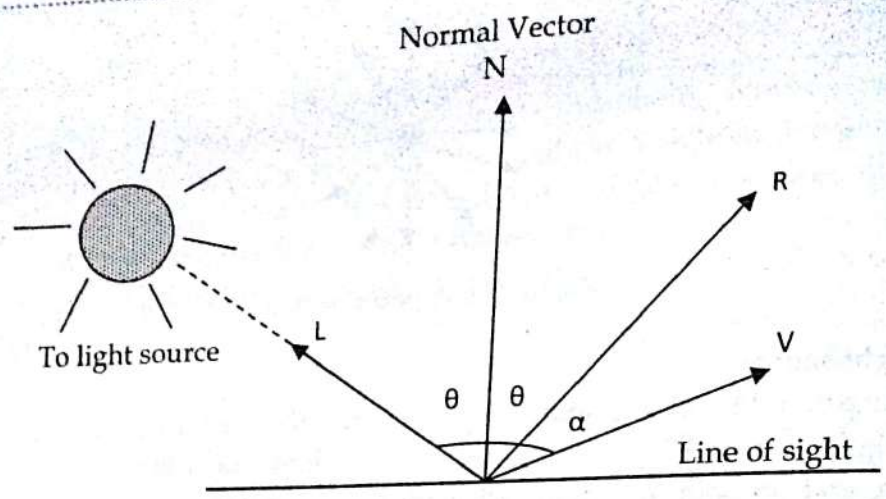


Figure 9.4: Reflection of light

Where, L=Light source direction
N=Normal vector
θ=Angle of incidence
V=line of sight
R=Direction of ideal specular reflection
α =Angle between R and V

Diffuse reflection

When light strikes the surface of a (non-metallic) material it bounces off in all directions due to multiple reflections by the microscopic irregularities inside the material (e.g. the grain boundaries of a polycrystalline material, or the cell or fiber boundaries of an organic material) and by its surface, if it is rough. Thus, an image is not formed. This is called **diffuse reflection.** The exact form of the reflection depends on the structure of the material. One common model for diffuse reflection is Lambertian reflectance, in which the light is reflected with equal luminance (in photometry) or radiance in all directions

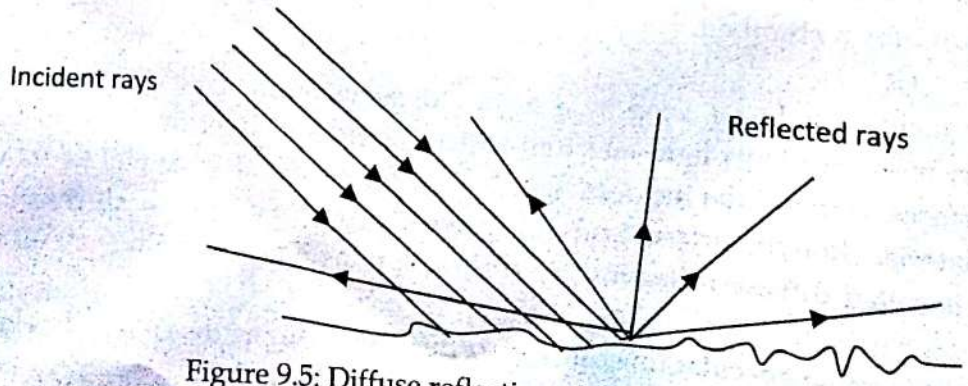


Figure 9.5: Diffuse reflections from a rough surface

Specular reflection

The name specular is derived from the Latin word speculum, meaning mirror. **Specular reflection**, also known as **regular reflection**, is the mirror like reflection of waves, such as light, from a surface. In this process, each incident ray is reflected at the same angle to the surface

normal as the incident ray, but on the opposing side of the surface normal in the plane formed by incident and reflected rays. The result is that an image reflected by the surface is reproduced in mirror-like (specular) fashion. In addition to diffuse refection, light source creates high lights or bright spots called **specular reflection.** However, this effect is seen more on shiny surface then dull surfaces.

Specular reflection obeys the Laws of Reflection and the light is reflected from a smooth surface at a definite angle. Specular reflection occurs for all polished and smooth surfaces, like mirrors. The angle of reflection in specular reflection is equal to the angle of incidence. Rough or unpolished surfaces are not good specular reflectors as they do not reflect light at the same angle as the angle of incidence.

Example: Persons forehead.

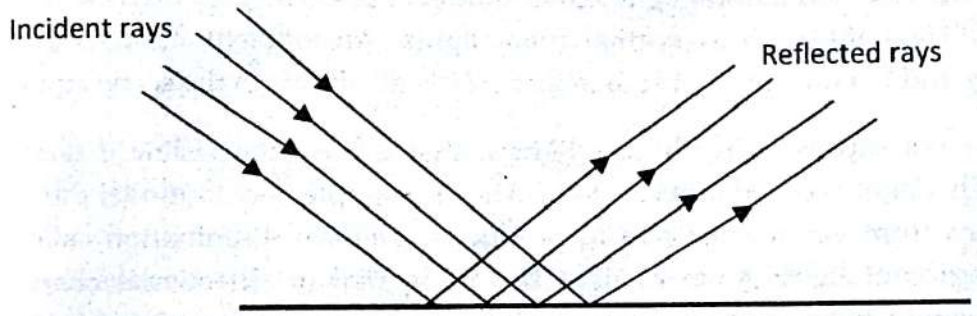


Figure 9.6: Specular reflection

Applications of Specular and Diffuse Reflection

1. There are several interesting applications of this distinction between specular and diffuse reflection. One application pertains to the relative difficulty of night driving on a wet asphalt roadway compared to a dry asphalt roadway. Most drivers are aware of the fact that driving at night on a wet roadway results in an annoying glare from oncoming headlights. The glare is the result of the specular reflection of the beam of light from an oncoming car. Normally a roadway would cause diffuse reflection due to its rough surface. But if the surface is wet, water can fill in the crevices and smooth out the surface. Rays of light from the beam of an oncoming car hit this smooth surface, undergo specular reflection and remain concentrated in a beam. The driver perceives an annoying glare caused by this concentrated beam of reflected light.
2. A second application of the distinction between diffuse and specular reflection pertains to the field of photography. Many people have witnessed in person or have seen a photograph of a beautiful nature scene captured by a photographer who set up the shot with a calm body of water in the foreground. The water provides for the specular reflection of light from the subject of the photograph. Light from the subject can reach the camera lens directly or it can take a longer path in which it reflects off the water before traveling to the lens. Since the light reflecting off the water undergoes specular reflection, the incident rays remain concentrated. The light is thus able to travel together to the lens of the camera and produce an image of the subject which is strong enough to perceive in the photograph.

Basic Illumination Models

A model for the interaction of light with a surface is called an illumination model. Illumination models are used to calculate light intensities that we should see at a given point on the surface of an object. Lighting calculations are based on the optical properties of surfaces, the background lighting conditions and the light source specifications. There are three illumination models listed below:

1. Ambient light
2. Diffuse reflection
3. Specular reflection

1. Ambient light

Ambient light means the light that is already present in a scene, before any additional lighting is added. It usually refers to natural light, either outdoors or coming through windows etc. It can also mean artificial lights such as normal room lights. An ambient light source represents a fixed-intensity and fixed-color light source that affects all objects in the scene equally.

Surface that is not exposed directly to a light source still will be visible if nearby objects are illuminated. This light is called ambient light. This is a simple way to model the combination of light reflections from various surfaces to produce a uniform illumination called the ambient light, or background light. Ambient light has no spatial or directional characteristics. The amount of ambient light incident on each object is a constant for all surfaces and over all directions.

If a surface is exposed only to ambient light, then the intensity of the diffuse reflection at any point on the surface is;

$$I = k_a I_a.$$

Where, I_a is the intensity of the ambient light, and k_a is the ambient reflection coefficient.

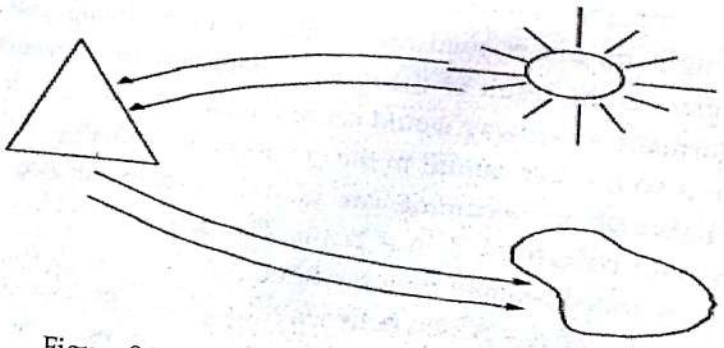


Figure 9.7: Object illuminated due to ambient light

Ambient light does not come from a specific light source and has no direction. It represents the light that is more or less everywhere in the scene, originating from multiple reflections of light at various surfaces. In a room with a lamp on a table, it will not be completely dark under the table although the lamp cannot shed its light directly under the table. The light is reflected by the surface of the table, the walls, the ceiling and the floor. Of course, the light under the table will have a lower intensity, but it will still be there with approximately the same intensity everywhere, not coming from a specific direction.

2. Diffuse reflection

Objects illuminated by ambient light are uniformally illuminated across their surfaces then the illumination is called diffuse illumination. It is the background light reflected from walls, floor and ceilings. We assume that the reflection is constant over each surface of the object and are independent of the viewing direction. This type of reflection on dull surfaces is called diffuse reflection.

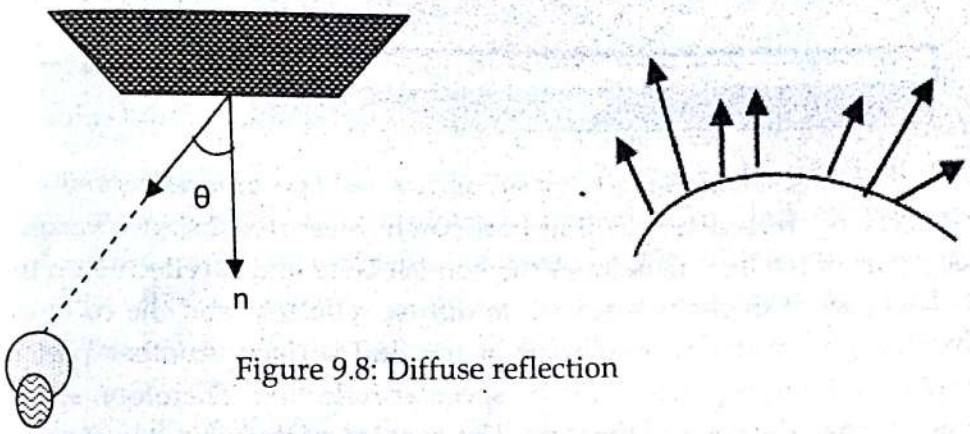


Figure 9.8: Diffuse reflection

This effect of light reflection for purely dull surfaces can be computed according to Lambert's cosine law by the illumination equation:

$$I = I_L . K_d . \cos\theta \quad \text{.....(1)}$$

Where, I_L is the intensity of the light hitting the surface, $0 \le k_d \le 1$ is the reflection coefficient of the surface or the material, and θ is the angle between the normal vector **n** to the surface in the considered point and the vector **I** pointing in the direction where the light comes from. The illumination equation (1) for diffuse reflection is only valid for angles θ between 0° and 90°. Otherwise, the light ray hits the surface from the backside so that no reflection occurs. In the case of directional light coming from a light source in infinite distance, the variable I_L in (1) has the same value everywhere. In the case of a point light source, I_L is the intensity of the light source multiplied by the attenuation factor, which depends on the distance of the point on the surface to the light source.

3. Specular reflection

Specular light is the white highlight reflection seen on smooth, shinny objects. Specular light is dependent on the direction of the light, the surface normal and the viewer location. When we look at an illuminated shiny surface, such as polished metal, a person's forehead, we see a highlight or bright spot, at certain viewing direction. Such phenomenon is called specular reflection. This phenomenon occurs as a result of total internal reflection of the incident light in a concentrated region around the specular reflection angle. Due to this phenomenon, the surface appears to be not in its original color but of white color.

Diffuse reflection on dull surfaces reflects the light into all directions. **Specular reflection** occurs on shiny surfaces. Such shiny surfaces reflect at least a portion of the light in a similar way as a

mirror. In contrast to diffuse reflection, ideal specular reflection takes place only in one direction. The difference between diffuse and specular reflection is illustrated in fig below.

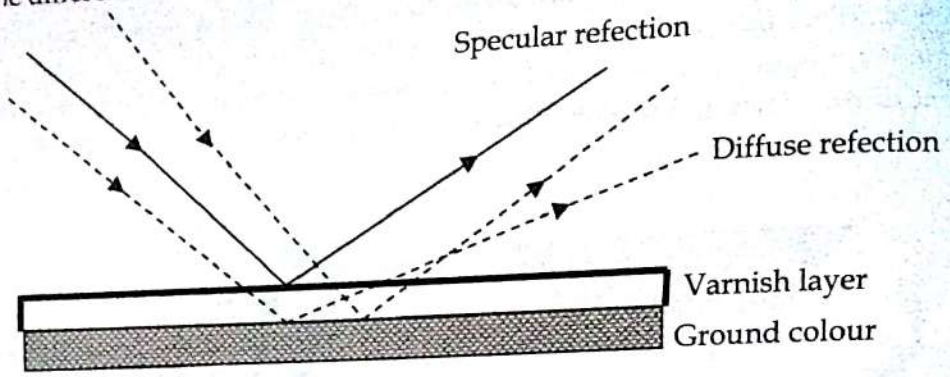


Figure 9.9: Difference between diffuse and specular reflections

Shiny surfaces very often have a very thin transparent layer, for instance varnish. When light hits the surface, part of the light penetrates the varnish layer and is reflected on the dull surface of the object. This part of the light is subject to diffuse reflection and the color of the reflected light depends strongly on the ground color of the dull surface. Another part of the light is directly reflected on the transparent layer by specular reflection. Therefore, specular reflection does usually not change the color of the light. The position of the viewer is of no importance for calculating effects coming from diffuse reflection. Whether or where the viewer can see specular reflection depends on his position.

The Phong specular reflection model described by the relation,

$$I = I_L . w(\theta) . \cos(\alpha)^n \ldots\ldots\ldots\ldots\ldots\ldots (1)$$

Where, I_L is the intensity of the light. The value $0 \leq w(\theta) \leq 1$ is the fraction of the light which is directly reflected at the shiny surface. n is the specular reflection exponent of the surface. For a perfect mirror, $n=\infty$ would hold. A smaller n leads to a less focused specular reflection.

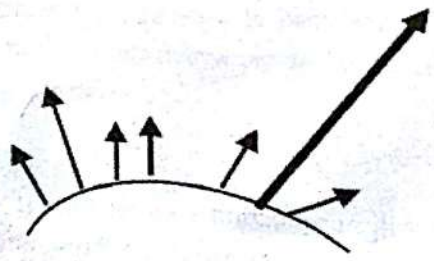


Figure 9.10: specular reflections

Intensity attenuation and Color consideration

As radiant energy from a pint light source travels through space, its amplitude is attenuated by the factor $1/d^2$, where d is the distance that the light has traveled i.e. surface close to the light source receives a higher incident intensity from the source than a distance surface.

To produce realistic lighting effects, our illumination model should take this intensity attenuation into account. Otherwise, we are illuminating all surfaces with the same intensity, no

matter how far they might be from the light source. If two parallel surfaces with the same optical parameters overlap, they would be indistinguishable from each other. The two surfaces would be indistinguishable from each other. The two surfaces would be displayed as one surface.

Most graphics displays of realistic scenes are in colour. But the illumination model discussed so far considers only monochromatic lighting effects. To incorporate colour, we need to write the intensity equation as a function of the colour properties of the light sources and object surfaces.

Transparency

Transparent surfaces reflect a part of the light, but objects behind them can also be seen. A typical transparent object is a colored glass pane. A transparent surface, in general, produces both reflected and transmitted light. The relative contribution of the transmitted light depends on the degree of transparency of the surface and whether any light sources or illuminated surfaces are behind the transparent surface. Transparency means that only a fraction of the light of the objects behind the transparent surface can pass through the transparent surface, but no distortion as with frosted glass happens. Such objects like milk glass are called translucent.

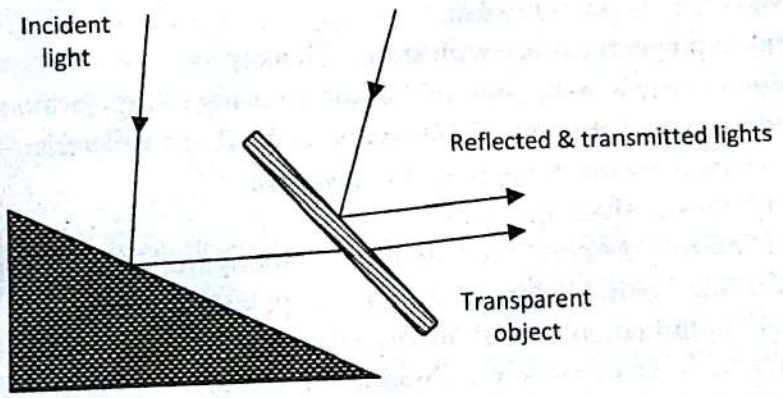


Figure 9.11: Light emission from a transparent surface

Shadows

Shadow can help to create realism. Without it, a cup, e.g., on a table may look as if the cup is floating in the air above the table. By applying hidden-surface methods with pretending that the position of a light source is the viewing position, we can find which surface sections cannot be seen from the light source shadow areas. We usually display shadow areas with ambient-light intensity only.

"Casting a shadow" is not an active matter, but simply the lack of light from a light source that does not reach the object's surface with the shadow on it.

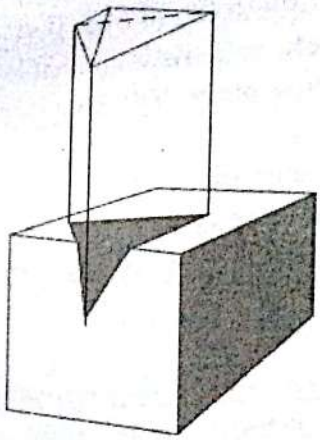


Figure 9.12: Shadow on an object

Polygon (surface) Rendering Method

Polygon rendering is the process of calculating intensity and color considerations for a polygon surface.

Scene Description + Illumination Model + Rendering Technique = Image

There are two ways of polygon rendering:

- Render each polygon surface with single intensity, or
- Calculate intensity at each point of the surface using interpolation scheme.

In order to produce a realistic image with various kinds of reflection, there are 3 common shading methods which are mainly applied to polygons:

1. Constant Intensity shading Method
2. Gouraud Shading method (Intensity Interpolation) and
3. Phong Shading Method (Normal Vector Interpolation)

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

1. Constant Intensity Shading(Flat Shading)

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

- Polygon surface must be one face of a polyhedron and is not a section of a curved-surface.
- All light sources illuminating the polygon are sufficiently far from surface. **(diffuse reflection)**

- The viewing position is sufficiently far away from the surface. **(specular reflection)**

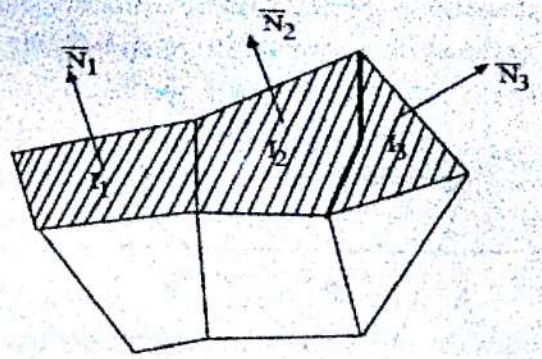


Figure 9.13: Flat shading

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

Interpolated Shading

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

2. Gouraud Shading

It is a method for linearly interpolating a color or shades across the polygon. It was developed by Henri **Gouraud.** Gouraud shading, also called intensity interpolating shading or color interpolating shading eliminates intensity discontinuities that occur in flat shading. Each polygon surface is rendered with Gouraud shading by performing following calculations.

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

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

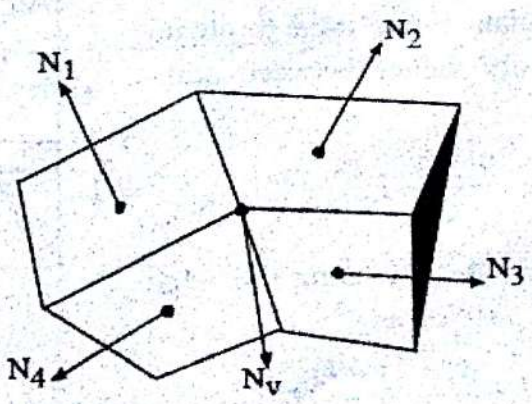
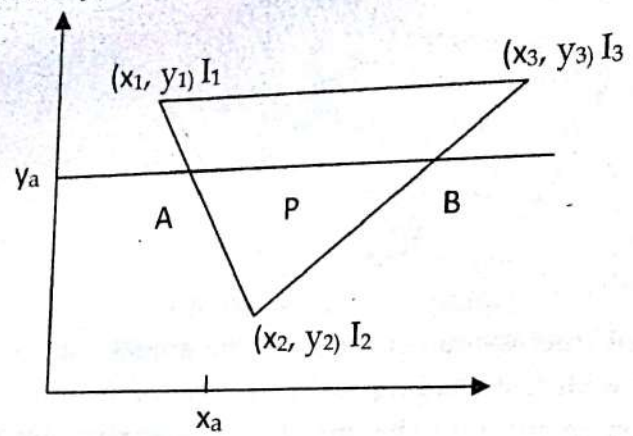


Figure 9.14: Gouraud Shading

Here $N_v = \frac{N_1 + N_2 + N_3 + N_4}{|N_1 + N_2 + N_3 + N_4|}$ Where N_v is normal vector at a vertex sharing four surfaces as in figure.

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

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



Here in figure, the intensity of vertices 1, 2, 3 are I_1, I_2, $I_3$are obtained by averaging normal of each surface sharing the vertices and applying an illumination model. For each scan line, intensity at intersection of line with Polygon edges is linearly interpolated from the intensities at the edge end point. So Intensity at intersection point A, I_a is obtained by linearly interpolating intensities of I_1 and I_2 as

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

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

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

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

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

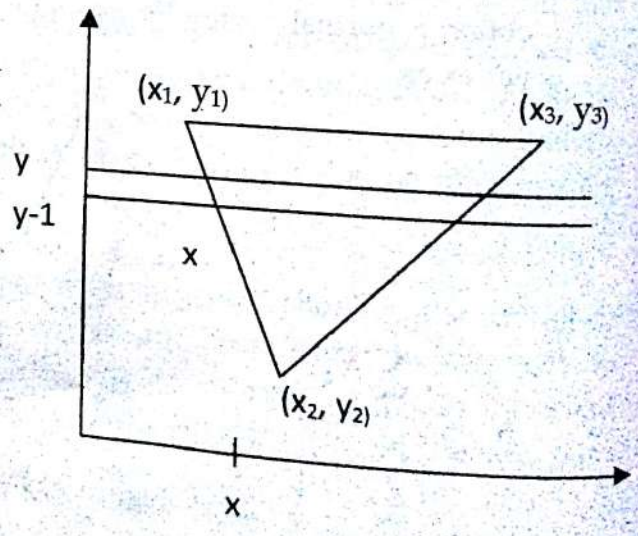
Then incremental calculations are used to obtain successive edge intensity values between scan-lines as:

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

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

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

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



Similar calculations are made to obtain intensity successive horizontal pixel.

Advantages

- Remove intensity discontinuities at the edge as compared to constant shading.
- In this model we calculate the intensity for each pixel; it ends up with the neighboring pixel across the border having nearly the same intensity. So the rendered model appears continuous.
- Gouraud shading can be combined with a hidden surface algorithm to fill in the visible polygons along each scan line.

Disadvantages

- Highlights on the surface are sometimes displayed with anomalous shape and linear intensity interpolation can cause bright or dark intensity streak called mach-bands.
- Sharp drop of intensity values on the polygon surface cannot be displayed.

3. Phong Shading

Best known shading algorithm, developed by Phong Bui Tuong, is called Phong shading or normal vector interpolation shading. Idea here is to interpolate the normal vectors instead of the light intensity. Then apply the illumination model at each surface point. It provides more accurate calculation of light intensity values and more realistic surface highlights. But it is computationally inefficient.

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

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

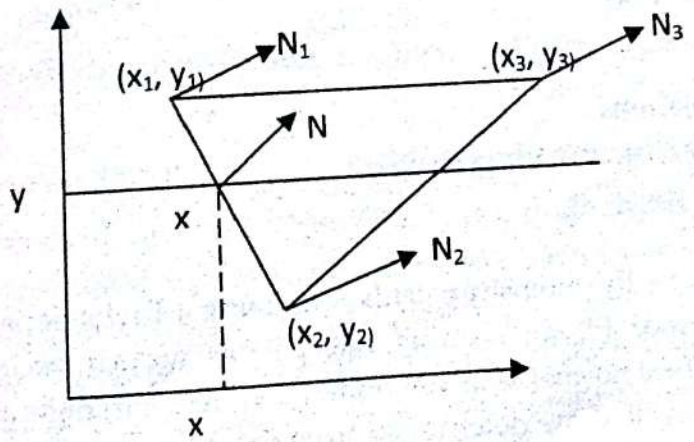


Figure 9.15: Phong Shading

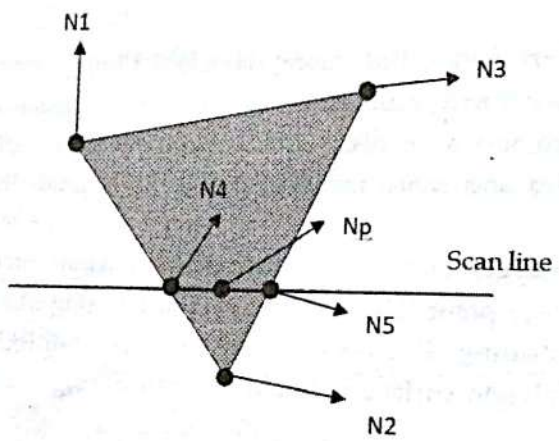
In figure, N_1, N_2, N_3 are the normal unit vectors at each vertex of polygon surface. For scan-line that intersects an edge, the normal vector N can be obtained by vertically interpolating normal vectors of the vertex on that edge as,

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

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

Algorithm

- Determine the average unit normal vector at each polygon vertex.
- Linearly interpolate the vertex normal over the surface of the polygon.
- Apply an illumination model along each scan line to calculate pixel intensities for the surface points. The intensity value is calculated using the interpolated normal vector.



$$\begin{cases} N_4 = \dfrac{y_4 - y_2}{y_1 - y_2} N_1 + \dfrac{y_1 - y_4}{y_1 - y_2} N_2 \\[2ex] N_5 = \dfrac{y_5 - y_2}{y_3 - y_2} N_3 + \dfrac{y_3 - y_5}{y_3 - y_2} N_2 \\[2ex] N_p = \dfrac{y_p - y_5}{y_4 - y_5} N_4 + \dfrac{y_4 - y_p}{y_4 - y_5} N_5 \end{cases}$$

Figure 9.16: Phong shading

Advantages

- It displays more realistic highlights on a surface
- It reduces the mach band effect
- It gives more accurate result

Disadvantages

- It requires more calculations
- It greatly increases the cost of shading steeply.

Fast Phong Shading

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

$N = Ax + By + C$ Where A, B, C are determined from the three vertex equations.

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

Omitting the reflectivity and attenuation parameters

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

Re writing this

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

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

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

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

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

Where, each T_k is a function of parameters a, b, c, d... And so forth. This method still takes twice as long as in Gouraud shading. Normal Phong shading takes six to seven times that of Gouraud shading.

Difference between Gouraud Shading and Phong Shading

S. No	Gouraud Shading	Phong Shading
1.	Gouraud shading, named after Henri Gouraud.	Phong Shading model named after Bui TuongPhong.
2.	Gouraud first published the technique in 1971.	Phong Shading published the technique in 1973.
3.	Computes illumination at border vertices and interpolates	Illumination at every point of polygon surface.
4.	Interpolates colors along edges and scan line.	Interpolates normal instead of colors
5.	Not so expensive	More expensive than Gouraud Shading
6.	Requires moderate processing and time.	Required complex processing and it is slower. It's produce good quality.
7.	Lighting equation used at each vertex.	Lighting Equation used at each pixel.
8.	Gouraud shading is in between the two: like Phong shading, each polygon has one normal vector per vertex, but instead of interpolating the vectors, the color of each vertex is computed and then interpolated across the surface of the polygon	Each rendered polygon has one normal vector per vertex; shading is performed by interpolating the vectors across the surface and computing the color for each point of interest
9.	Smooth surfaces	Smooth and shining surfaces

Exercise

1. What is attenuation factor? Calculate the reflection vector in the specular reflection.
2. Explain the principle of Phong shading algorithm. And also write their advantages and disadvantages over Gouraud shading.
3. Briefly explain "flat shading", "Gouraud shading" and "Phong shading", and describe the differences between them. Which of these modes does OpenGL implement or not implement and why?
4. Explain different polygon rendering methods. Also write down their uses in computer graphics.
5. What is the rendering equation? Derive one version of it. Explain how ray tracing and radiosity approximate the rendering equation. What is the radiosity equation? Make the appropriate approximations to derive it from the full rendering equation.
6. Differentiate between diffuse reflection and specular reflection? Also explain Phong illumination model.
7. Explain how to intersect a ray with a sphere in ray tracing. Show how to do this for a general implicit surface.
8. Explain basic rendering and illumination models. Also describe their advantages and disadvantages.
9. Explain how shading is done, assuming diffuse illumination. Give emphasis on Phong Shading and Gourand Shading.
10. Explain the concept of Rendering. Explain Gourand shading with their advantages and disadvantages.
11. Discuss relative advantages and disadvantages of Gourand Shading and Phong Shading.
12. Write short note on rendering and illumination. Also describe their applications in real life.
13. Describe Phong illumination model. Explain the parameters used in Phong model.
14. What is Mach Band effect?
15. What is concept behind Mach Band effect? Discuss its effect in Phong Shading, Gourand Shading and constant Shading.
16. What is the difference between a local illumination model and a global illumination model?
17. The color of an object is largely determined by its diffuse reflection coefficient. If k_d= (0.8, 0.4, 0) and the light is blue, what is the color of the object?
18. State and explain the law of Lambert's.
19. What do you mean by transparency? Explain its properties.
20. What is the main advantage of using fast Phong shading over shading? Write down advantages and disadvantages of Phong shading.

❑❑❑

Chapter

10

Introduction to Virtual Reality and Animation

STRUCTURE

10.1 Introduction
10.2 Key Elements of a Virtual Reality Experience
10.3 Types of Virtual Reality
10.4 Key Components in a Virtual Reality System
10.5 Other Concepts to Understand How Virtual Reality Headsets Work
10.6 Design of animation sequences
10.7 Computer animation languages
10.8 Morphing and simulating accelerations.

10.1 Introduction

Virtual reality is an artificial environment that is created with software and presented to the user in such a way that the user suspends belief and accepts it as a real environment. On a computer, virtual reality is primarily experienced through two of the five senses: sight and sound. Virtual Reality (VR) is the use of computer technology to create a simulated environment. Unlike traditional user interfaces, VR places the user inside an experience. Instead of viewing a screen in front of them, users are immersed and able to interact with 3D worlds. By simulating as many senses as possible, such as vision, hearing, touch, even smell, the computer is transformed into a gatekeeper to this artificial world. The only limits to near-real VR experiences are the availability of content and cheap computing power. The simulation environment can be similar to the real world, for example, simulations for pilot or combat training, or it can differ significantly from reality, as in VR games.

Virtual reality can be divided into:

1. The simulation of a real environment for training and education
2. The development of an imagined environment for a game or interactive story

VR has five main components which are variable according per the instructional context requirements:

- Dimensionality
- Motion or animation
- Interaction
- Viewpoint or frame of reference, and
- Immersion, or embodiment, through enhanced multisensory experiences

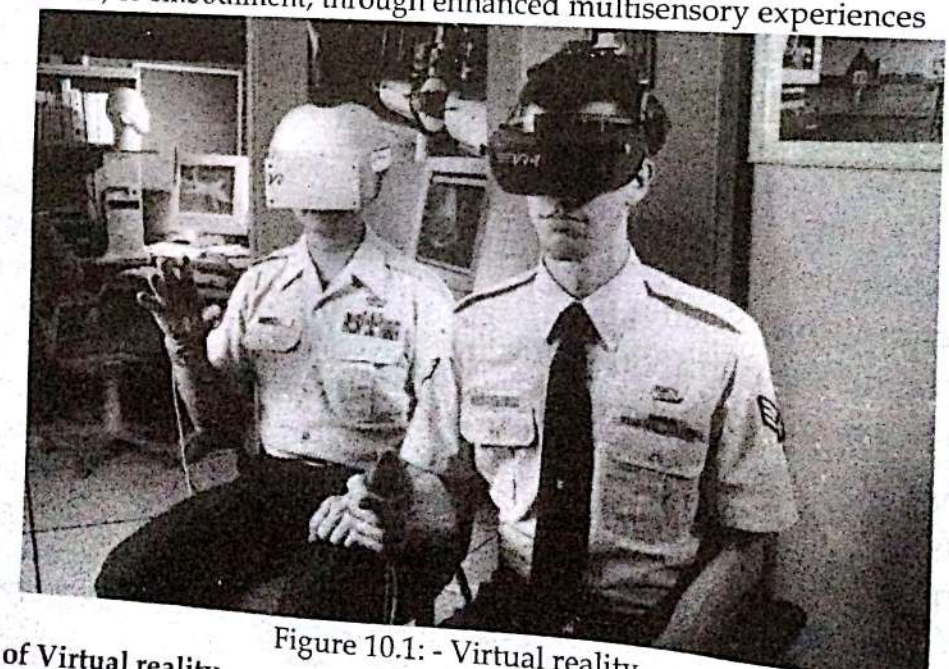


Figure 10.1: - Virtual reality

Advantages of Virtual reality

- VR is imaginably more personal than electronic mail or instant messaging, or even a letter or a telephone call.
- VR is a great social leveler; it may find a common ground across differences in age, culture and linguistic orientation.

- People will be drawn together by similar interests instead of purely by geographic location
- Communication will be challenging and rewarding, more effective and productive, and thus more enjoyable.
- A tremendous opportunity for every connected person to find his or her field and discipline.

Disadvantages of Virtual reality

- An inescapable aspect of social life is the formation and maintenance of interpersonal relationships.
- Interaction ought not to be substituted for community.
- Separates the 'haves' from the 'have-nots', a technology of information age industrialized nations
- Could lead to low self esteem, feelings of worthlessness and insignificance, even self destructive acts

Application area of virtual reality

Here is a list of the many **applications of virtual reality**:

- In the Military
- In Education
- in Healthcare
- in Entertainment
- Fashion
- Heritage
- in Business
- in Engineering
- in Sport
- in Media
- Scientific Visualizations
- in Telecommunications
- in Construction
- in Film
- Programming languages

Key Elements of a Virtual Reality Experience

1. Virtual World

A virtual world is a three-dimensional environment that is often, but not necessarily, realized through a medium (i.e. rendering, display, etc.) where one can interact with others and create objects as part of that interaction. In a virtual world, visual perspectives are responsive to changes in movement and interactions mimic those experienced in the real world.

2. Immersion

Virtual reality immersion is the perception of being physically present in a non-physical world. It encompasses the sense of presence, which is the point where the human brain believes that is

somewhere it is really not, and is accomplished through purely mental and/or physical means. The state of total immersion exists when enough senses are activated to create the perception of being present in a non-physical world. Two common types of immersion include:

- Mental Immersion – A deep mental state of engagement, with suspension of disbelief that one is in a virtual environment.
- Physical Immersion – Exhibited physical engagement in a virtual environment, with suspension of disbelief that one is in a virtual environment.

3. Sensory Feedback

Virtual reality requires as many of our senses as possible to be simulated. These senses include vision (visual), hearing (aural), touch (haptic), and more. Properly stimulating these senses requires sensory feedback, which is achieved through integrated hardware and software (also known as inputs). Examples of this hardware and inputs are discussed below as key components to a virtual reality system, which include head mounted displays (HMD), special gloves or hand accessories, and hand controls.

4. Interactivity

The element of interaction is crucial for virtual reality experiences to provide users with enough comfort to naturally engage with the virtual environment. If the virtual environment responds to a user's action in a natural manner, excitement and senses of immersion will remain. If the virtual environment cannot respond quickly enough, the human brain will quickly notice and the sense of immersion will diminish. Virtual environment responses to interaction can include the way a participant moves around or changes in their viewpoint; generally through movements of their head.

Types of Virtual Reality

Several categories of virtual reality technologies exist, with more likely to emerge as this technology progresses. The various types of virtual reality differ in their levels of immersion and also virtual reality applications and use cases. Below, we explore a few of the different categories of virtual reality:

1. Non-Immersive
2. Semi-Immersive
3. Fully Immersive

1. Non-Immersive

Non-immersive simulations are the least immersive implementation of virtual reality technology. In a non-immersive simulation, only subsets of the user's senses are stimulated, allowing for peripheral awareness of the reality outside the virtual reality simulation. Users enter into these three-dimensional virtual environments through a portal or window by utilizing standard high resolution monitors powered by processing power typically found on conventional desktop workstations.

2. Semi-Immersive

Semi-immersive simulations provide a more immersive experience, in which the user is partly but not fully immersed in a virtual environment. Semi-immersive simulations closely resemble and utilize many of the same technologies found in flight simulation. Semi-immersive

simulations are powered by high performance graphical computing systems, which are often then coupled with large screen projector systems or multiple television projection systems to properly stimulate the user's visuals.

3. Fully Immersive

Fully-immersive simulations provide the most immersive implementation of virtual reality technology. In a fully-immersive simulation, hardware such as head-mounted displays and motion detecting devices are used to stimulate all of a user's senses. Fully immersive simulations are able to provide very realistic user experiences by delivering a wide field of view, high resolutions, increased update rates (also called refresh rate), and high levels of contrast into a user's head-mounted display (HMD).

Key Components in a Virtual Reality System

1. PC (Personal Computer)/Console/Smart phone

Virtual reality content, which is the users view inside of a virtual reality headset, is equally important as the headset itself. In order to power these interactive three-dimensional environments, significant computing power is required. This is where PC (Personal Computer), consoles, and smart phones come in. They act as the engine to power the content being produced.

2. Head-Mounted Display

A head-mounted display is a type of device that contains a display mounted in front of a user's eyes. This display usually covers the user's full field of view and displays virtual reality content. Some virtual reality head mounted displays utilize Smartphone displays, including the Google Cardboard and Samsung Gear VR. Head-mounted displays are often also accompanied with a headset to provide for audio stimulation.

3. Input Devices

Input devices are one of the two categories of components that provide users with a sense of immersion (i.e. convincing the human brain to accept an artificial environment as real). They provide users with a more natural way to navigate and interact within a virtual reality environment. Some of the more common forms of virtual reality input devices include:

- Joysticks
- Force Balls/Tracking Balls
- Controller Wands
- Data Gloves
- Track pads
- On-Device Control Buttons
- Motion Trackers/Bodysuits
- Treadmills
- Motion Platforms etc.

Key Components Inside of a Virtual Reality Headset

1. Sensors

The three most common sensors in a virtual reality headset are magnetometers, accelerometers and gyroscopes. These sensors work together by measuring the user's motions and direction in space. Their ultimate goal is to achieve true six-degrees-of-freedom (6DoF), which covers all the degrees of motion for an object in space.

- **Magnetometer** – The magnetometer tells your device which direction it is facing on the surface of the earth. The magnetometer acts as a sort of compass for the device. As such, it is able to do this by measuring magnetic fields.
- **Accelerometers** – The accelerometer tells your device which way up it is. To do this, your device will have several accelerometers to work together measuring things like gravitational pull in relation the accelerometer measuring the device's orientation.
- **Gyroscopes** – A gyroscope calculates the orientation of your device. It does this to either help your device maintain a particular orientation or make sure it properly changes orientation when it should.

2. Lenses

Lenses lie between your eyes and pixels on the display screen. They focus and reshape the picture for each eye by angling two 2D images to mimic how each of our eyes takes in views of the world. This creates an impression of depth and solidity, which we perceive to be a three-dimensional image. Lenses on each virtual reality device are not one-size-fits all and have to be adjusted for initial use as all devices have different lens properties.

3. Display Screens

Display screens show the images that user view through the lenses. They are typically LCD and receive video feed from the computer or Smartphone. Depending on the headset, the video feed is either sent to one display or two displays (one per eye). This happens via wireless connection, Smartphone connection, or HDMI. The most common types of virtual reality display technology is a Liquid Crystal Display (LCD) screen, similar to the kinds used in smart phones and computer monitors. An alternative display technology is an Organic Light-Emitting Diode (OLED) screen.

4. Processing

Virtual reality systems demand a substantial amount of power, even in comparison to notoriously power hungry gaming systems. The processing power required by virtual reality systems can be broken down into several categories:

- **Input Processor** – Controls the devices used to input information to the computer. They retrieve and distribute data to the rest of the system with minimal lag time. Examples include keyboards, mouse, 3D position trackers, and voice recognition systems.
- **Simulation Processor** – Takes the user inputs along with any other tasks that are programmed from the natural world and determines the actions that will take place in the virtual world. This is a core component of the VR system.
- **Rendering Processor** – Creates the sensations that are output to the user. These include visual, auditory, haptic and other sensory systems. Separate rendering processes are used for each sensory system.

Other Concepts to Understand How Virtual Reality Headsets Work

1. **Field of View**

Field of view (also called Field of Vision or FOV) is an important component used in virtual reality to provide users with a realistic perception of their environmental landscape. Simply put, field of view refers to how wide the picture is. Field of view is measured based on the degree of display (e.g. 360°). Most high-end headsets make do with 100° or 110° field of view which is sufficient for most virtual reality content.

2. **Frame Rate**

Frame rate refers to the frequency at which the display screen shows consecutive images, which are also called frames. Television shows run at 30 frames per second (fps) and some game consoles run at 60 frames per second (fps). In virtual reality, a minimum frame rate of approximately 60 frames per second is needed to avoid content stuttering or cause of simulation sickness. The Oculus Rift runs at 90 fps, providing Oculus Rift users with a very lifelike experience. Future Frame rates for virtual reality headsets are set to inevitably continue getting faster, providing for a more realistic experience.

3. **Latency**

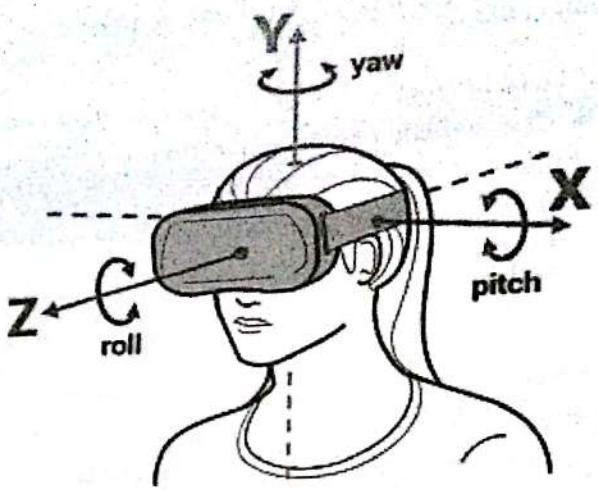
Latency refers to the amount of time it takes for an image displayed in a user's headset to catch up to their changing head position. Latency can also the thought of as a delay, and is measured in milliseconds (ms). In order for an experience to feel real, latency usually needs to be in the range of 20 milliseconds (ms) or less. Low latency, or very little delay, is needed to make the human brain accept the virtual environment as real. The higher the latency, a noticeable and unnatural lag may set in, consequently causing simulation sickness for the user.

4. **Audio**

Virtual reality audio may not be as technically-complex as the visual components, however, it is an equally important component to stimulate a user's senses and achieve immersion. Most virtual reality headsets provide users with the option to use their own headphones in conjunction with a headset. Other headsets may include their own integrated headphones. Virtual reality audio works via positional, multi-speaker audio that gives the illusion of a 3-dimensional world. Positional audio is a way of seeing with your ears and is used in virtual reality. This technology is already quite common and often found in home theater surround sound systems.

5. **Tracking**

Tracking handles the vital task of understanding a user's movements and then acting upon them accordingly to maintain full immersion in virtual reality. Below, we explore the three of the main types of virtual reality tracking: Head Tracking



Head tracking refers to the way in which the view in front of you will shift as you look up, down and side-to-side. A system called six degrees of freedom (6DoF) plots your head in terms of your x, y and z axis to measure head movements forward and backwards, side-to-side and shoulder to shoulder, otherwise known as pitch, yaw and roll. Head tracking utilizes a series of sensors, vital to any virtual reality headset, which includes a gyroscope, accelerometer, and magnetometer. Head-tracking technology must be low latency in order to be effective. Anything above 50ms will cause a lag between the headset movement and virtual reality environment changes.

Design of animation sequences

Common Steps of designing the animation sequence are as given:

1. **Layout of Storyboard:** Storyboard layout is the action outline utilized to illustrate the motion sequence as a set of basic events which are to acquire place. This is the kind of animation to be produced that selects the storyboard layout. So, the storyboard comprises a set of rough sketches or a list of basic concepts for the motion.
2. **Definition of Object:** The object definition is specified for all participant objects in action. The objects can be explained in terms of fundamental shapes, related movements or movement with shapes.
3. **Specification of Key Frame:** this is the detailed drawing of the scene at an exact time in the animation sequence. Inside each key frame, all objects are positioned as per to time for that frame. Several key frames are selected at the extreme positions in the action; others are spaced hence the time interval among key frames is not as great. More key frames are given for intricate motion than for easy, slowly varying motions.
4. **In-between frames Generation:** In-among frames are the middle frames among the key frames. The number of among frames is based on the media to be utilized to display the animation. In common, film needs twenty-four frames per second, and graphic terminals are refreshed on the rate of 30 to 60 frames per second. Classically the time interval for the motion is set up hence there are 3 to 5 among for each pair of key frames. Based upon the speed identified for the motion, several key frames can be duplicated.

Computer Animation Languages

Design and control of animation sequences are handled with a set of animation outlines. A general purpose language, such as c, lips, Pascal, or FORTRAN, is often used to program the animation functions, but several specialized animation languages have been developed. Animation functions include a graphics editor, a key frame generator, an in-between generator, and standard graphics routines. The graphics editor allows us to design and modify object shapes, using spline surface, constrictive solid geometry methods, or other representation schemes.

A typical task in an animation specification is scene description. This includes the positioning of objects and light sources, defining the photometric parameters, and setting the camera parameters (position, orientation, and fewer characteristics). Another standard function is action specification. This involves the layout of motion paths for the object and camera. And we need 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. Key-frame systems are specialized animation languages designed simply to generate the in betweens from the user-specified key frames. Usually, each object in the scene is defined as a set of rigid bodies connected at the joints and with a limited number of degrees of freedom. Parameterized systems allow object motion characteristics to be specified as part of the object definitions. The adjustable parameters control such object characteristics as degree of freedom, motion limitations, and allowable shape changes. Scripting systems allows object specifications and animation sequences to be define with a user-input script from the script, a library of various objects and motions can be constructed.

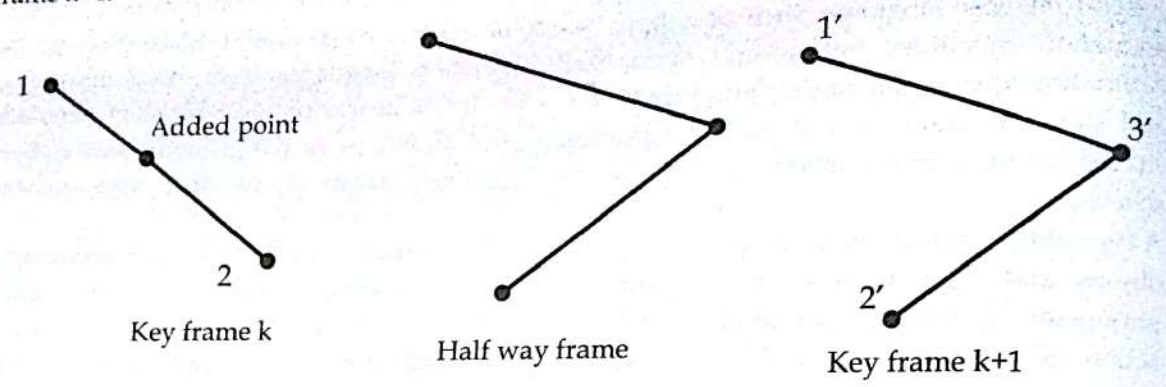
Morphing and Simulating Accelerations

Morphing

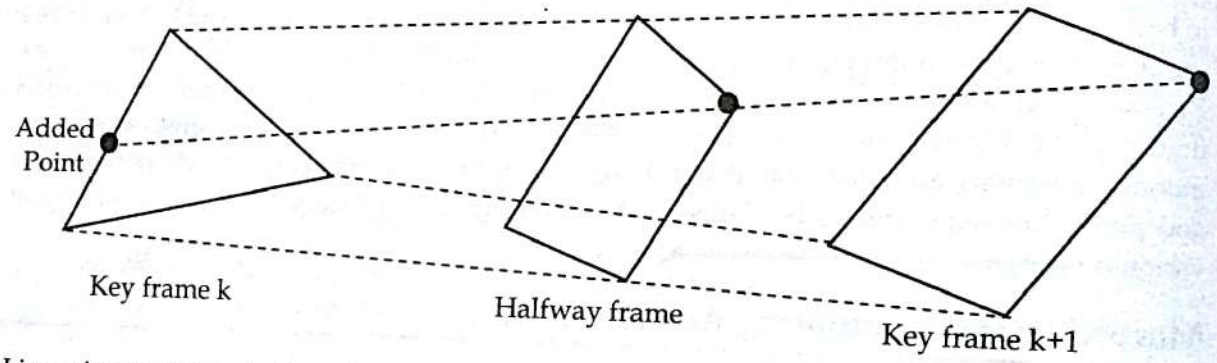
The transformation of object shapes from one form to another form is called morphing. It is one of the most complicated transformations. A morph looks as if two images melt into each other with a very fluid motion. In technical terms, two images are distorted and a fade occurs between them. Morphing method can be applied to any motion or transition involving charge in shape. Given the key frames for an object transformation, we first adjust the object specification in one of the frames so that the number of polygon edges (vertices) is the same for two frames.

Examples of morphing are in television advertising is described below:

An edge with vertex position 1 and 2 in key frame k evolves into two connected edges in key frame k+1.



Linear interpolation for transforming a line segment in key frame k into two connected line segments in key frame k+1



Linear interpolation for transforming a triangle into a quadrilateral

We can state general preprocessing rule for equalizing key frames in terms of either the number of edges or the number of vertices to be added to a key frame.

Simulating Acceleration

Curve fitting techniques are often used to specify the animation paths between key frames. Given the vertex positions at the key frames, we can hit the positions with linear or non-linear paths. To simulate accelerations, we can adjust time spacing for the in-betweens. For constant speed (zero acceleration), we use equal interval time spacing for the in-between for key frames at time t1 and t2.

The time interval between key frames is then divided into (n+1) subintervals, yielding an in-between spacing of

$$\Delta t = (t2-t1)/(n+1)$$

We can calculate the time for any in-between as

$$tBj = t1 + j\Delta t,\ j=1,2,\ldots\ldots\ldots\ldots,n$$

And determine the values for the coordinate positions color, and other physical parameters. None zero accelerations are used to produce realistic displays of speed charges, particularly at the beginning and end of the motion sequence. We can model the starting and slow down

portion of an animation path with spline or trigonometric functions. Parabolic and cubic time functions have been applied to acceleration modeling, but trigonometric functions are mare commonly used in animation packages.

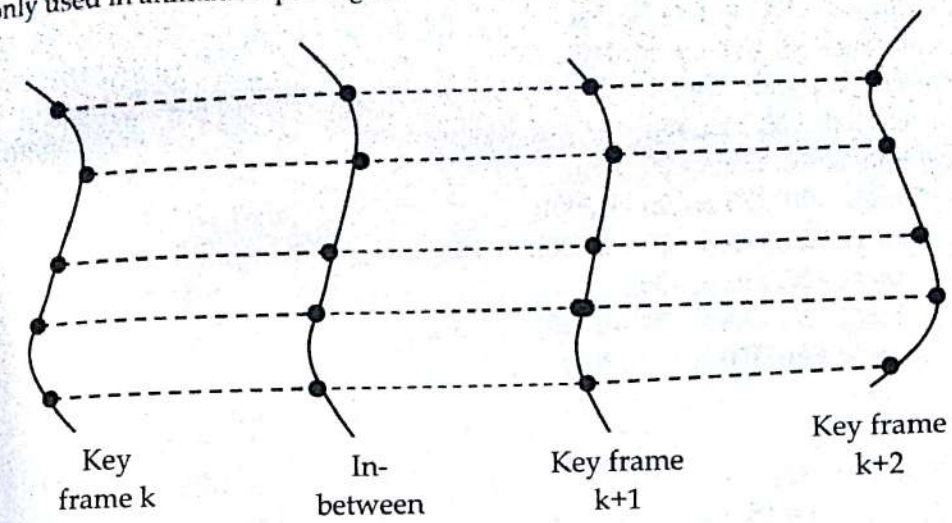


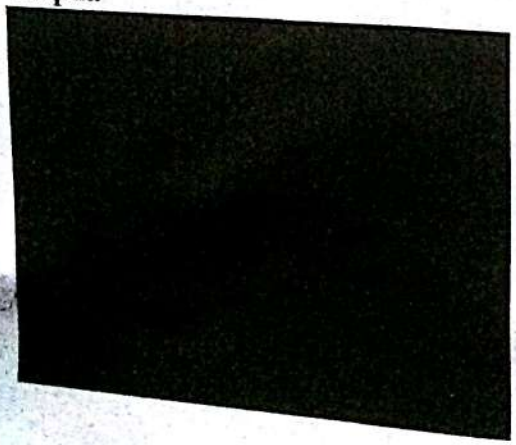
Figure 10.2: Fitting key grame vertex positions with non-linear splines

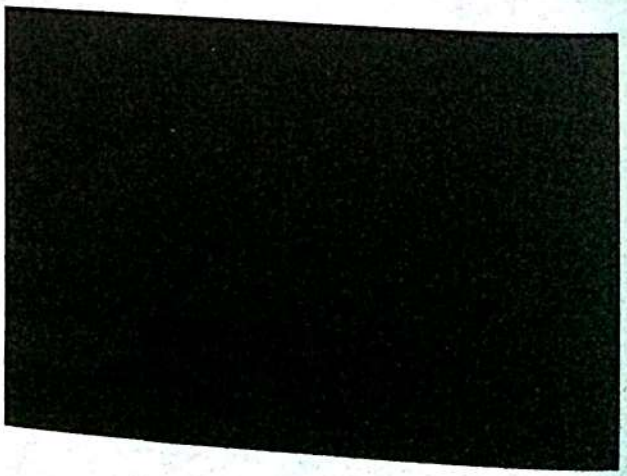
Lab problem 10.1: Program for man walking

```
#include<stdio.h>
#include<dos.h>
#include<conio.h>
#include<graphics.h>
#include<stdlib.h>
void main()
{
        int gd = DETECT, gm = DETECT, c = -200, i = 0, x = 40, l = 15, h = 15, ht = 0;
        initgraph(&gd, &gm, "C:\\TurboC3\\BGI");
        cleardevice();
        setcolor(BROWN);
        line(0, 201, 600, 201);
        cont:
        while (!kbhit())
        {
                setcolor(4);
                ellipse(x, 100, 0, 180, 50, 30);
                line(x - 50, 100, x + 50, 100);
                line(x, 100, x, 150);
                circle(x - 20, 115, 15);
                line(x - 20, 130, x - 20, 175);
                line(x - 20, 175, x - 20 - l, 200);
                line(x - 20, 175, x - 20 + l, 200);
                line(x - 20, 140, x, 150);
```

```
            line(x - 20, 140, x - 20 - h, 160);
            setcolor(0);
            delay(50);
            ellipse(x, 100, 0, 180, 50, 30);
             line(x - 50, 100, x + 50, 100);
             line(x, 100, x, 150);
             circle(x - 20, 115, 15);
             line(x - 20, 130, x - 20, 175);
             line(x - 20, 175, x - 20 - l, 200);
             line(x - 20, 175, x - 20 + l, 200);
             line(x - 20, 140, x, 150);
             line(x - 20, 140, x - 20 - h, 160);
             line(x + 50, 100, x + 50, 200);
             x++;
             l--;
             if (l == -15)
                    l = 15;
             if (ht == 1)
                    h++;
             else
                    h--;
             if (h == 15)
                    ht = 0;
             else if (h == -15)
                    ht = 1;
        }
        if (getch() == ' ')
        {
                while (!kbhit());
                getch();
                goto cont;
        }
}
```

Output:



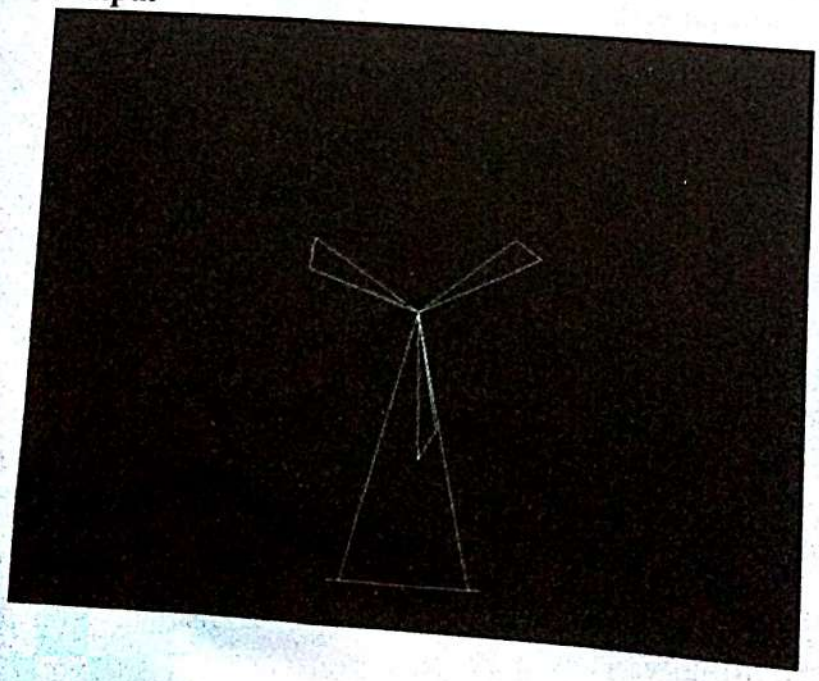


Lab problem 10.2: Program for windmill rotation

```
#include<stdio.h>
#include<graphics.h>
#include<math.h>
void wind(float x[7],float y[7]);
void main()
{
        int gd=DETECT,gm;
        float x[7],y[7],maxx,maxy,xw1,yw1,xw2,yw2;
        float theta=30;
        initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
        maxx=getmaxx();
        maxy=getmaxy();
        x[0]=maxx/2;
        y[0]=maxy/2;
        x[1]=y[4]=x[2]=-90;
        y[6]=y[5]=y[1]=60;
        y[2]=35;
        y[3]=-100;
        x[4]=20;
        x[3]=0;
        x[5]=90;
        x[6]=65;
        theta=theta*22/7/180;
        while(kbhit()==0)
        {
                wind(x, y);
                xw1=cos(theta)*x[1]+sin(theta)*y[1];
                yw1=-sin(theta)*x[1]+cos(theta)*y[1];
                xw2=cos(theta)*x[2]+sin(theta)*y[2];
                yw2=-sin(theta)*x[2]+cos(theta)*y[2];
                x[1]=xw1;
                y[1]=yw1;
                x[2]=xw2;
                y[2]=yw2;
                xw1=cos(theta)*x[3]+sin(theta)*y[3];
                yw1=-sin(theta)*x[3]+cos(theta)*y[3];
                xw2=cos(theta)*x[4]+sin(theta)*y[4];
                yw2=-sin(theta)*x[4]+cos(theta)*y[4];
                x[3]=xw1;
                y[3]=yw1;
                x[4]=xw2;
                y[4]=yw2;
                xw1=cos(theta)*x[5]+sin(theta)*y[5];
```

```
            yw1=-sin(theta)*x[5]+cos(theta)*y[5];
            xw2=cos(theta)*x[6]+sin(theta)*y[6];
            yw2=-sin(theta)*x[6]+cos(theta)*y[6];
            x[5]=xw1;
            y[5]=yw1;
            x[6]=xw2;
            y[6]=yw2;
            delay(50);
            cleardevice();
        }
        closegraph();
}
void wind(float x[7],float y[7])
{
        cleardevice();
        line(x[0],y[0],x[0]-50,y[0]+200);
        line(x[0],y[0],x[0]+50,y[0]+200);
        line(x[0]-60,y[0]+200,x[0]+60,y[0]+200);
        line(x[0],y[0],x[0]+x[1],y[0]-y[1]);
        line(x[0],y[0],x[0]+x[2],y[0]-y[2]);
        line(x[0]+x[1],y[0]-y[1],x[0]+x[2],y[0]-y[2]);
        line(x[0],y[0],x[0]+x[3],y[0]-y[3]);
        line(x[0],y[0],x[0]+x[4],y[0]-y[4]);
        line(x[0]+x[3],y[0]-y[3],x[0]+x[4],y[0]-y[4]);
        line(x[0],y[0],x[0]+x[5],y[0]-y[5]);
        line(x[0],y[0],x[0]+x[6],y[0]-y[6]);
        line(x[0]+x[5],y[0]-y[5],x[0]+x[6],y[0]-y[6]);
}
```

Output



Exercise

1. What are the different types of VR devices?
2. What are common mistakes advertisers make with VR?
3. How do you measure the reach of VR?
4. Who are the best VR producers?
5. What is the most important technical challenge in Virtual Reality?
6. What is the most important technical challenge in Virtual Reality?
7. What is Mixed Reality?
8. What are the best options for measuring eye movements in VR?
9. How do natural and programming languages work on the Web Virtual Reality?
10. What is Virtual Reality and how does it affects human life?
11. What are some leading areas for research in the field of Virtual Reality, Interactive Systems and Virtual Environment Development?
12. How to implement virtual reality scene with multiple collaborators using Unity?
13. How we can apply Virtual Reality for education?
14. Describe key components of virtual reality with suitable examples.
15. Write down advantages and disadvantages of virtual reality.
16. Describe application areas of virtual reality.
17. Describe use of virtual reality in education sector.
18. Differentiate between simple reality and mixed reality. Also write down applications of mixed reality with suitable example.
19. Describe various hardware and software components of virtual reality.
20. How do you measure quality of virtual reality in daily life? Explain.

❑❑❑

Chapter

11

Introduction to OpenGL

STRUCTURE

11.1 Introduction
11.2 Access to Libraries
11.3 Data Types
11.4 Callback functions
11.5 The OpenGL Utility Toolkit–GLUT
11.6 Our First Program
11.7 Drawing Primitives
11.8 Drawings pixels, lines, polygons using OpenGL

Introduction

OpenGL is the industry's most widely used environment for the development of portable, interactive 2D and 3D graphics applications. The OpenGL API is open source and is platform independent. OpenGL is a low-level graphics library specification. It makes available to the programmer a small set of geometric primitives - points, lines, polygons, images, and bitmaps. OpenGL provides a set of commands that allow the specification of geometric objects in two or three dimensions, using the provided primitives, together with commands that control how these objects are rendered (drawn).

Since OpenGL drawing commands are limited to those that generate simple geometric primitives (points, lines, and polygons), the OpenGL Utility Toolkit (GLUT) has been created to aid in the development of more complicated three-dimensional objects such as a sphere, a torus, and even a teapot. GLUT may not be satisfactory for full-featured OpenGL applications, but it is a useful starting point for learning OpenGL.

GLUT is designed to fill the need for a window system independent programming interface for OpenGL programs. GLUT simplifies the implementation of programs using OpenGL rendering. The GLUT application programming interface (API) requires very few routines to display a graphics scene rendered using OpenGL. The GLUT routines also take relatively few parameters.

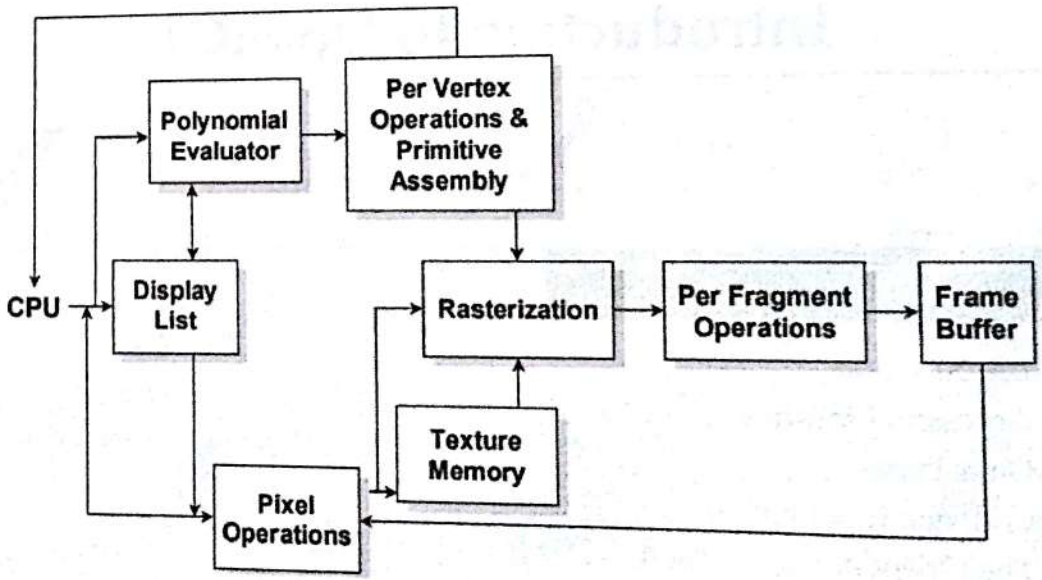


Figure 11.1: OpenGL Architecture

Like all software libraries, OpenGL consists of a series of function calls that can be invoked from your own programs. OpenGL has been ported to a number of platforms and languages. In typical use cases OpenGL is invoked via C or C++; most graphics applications demand efficient and fast-executing code - and this requirement is most often satisfied by C or C++. However it is possible to use OpenGL with other languages such as Java (via the JOGL API). OpenGL was developed by Silicon Graphics (SGI); a company that pioneered many early movie special effects and graphics techniques. OpenGL was derived from SGI's proprietary graphics running on their IRIX operating system. The 'Open' in OpenGL indicates that SGI opened the library up

to other operating systems - it does not indicate Open Source. Indeed, there is often little code in an OpenGL library - the library often acts simply as an interface to the graphics hardware in a machine e.g. a PC. However, in cases where a machine has no specialized graphics hardware, an OpenGL library will often emulate its presence. This is the case in Microsoft Windows - if your PC has no specialized graphics hardware, OpenGL will imitate the hardware using software only.

Why OpenGL?

- The first vendor-independent API for development of graphics applications
- There is no need to license it to use it.
- It was designed to use the graphics card where possible to improve performance.
- Originally based on a state machine, procedural model; thus it can be used with a wide variety of programming languages.

Access to Libraries

Although the OpenGL interface provides platform independence, the set-up of libraries and header files varies across systems and compilers and we cannot hope to cover all such variations.

Typically the initial header inclusions for a Windows system will include

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

Plus any other library access (maths.h etc) that we may require. This will be slightly different on Linux where if we are using X windows then a likely include will be

```
#include <X11/Xlib.h>
#include <GL/glut.h>
```

Data Types

As various versions of compilers are released along with different platforms the sizes of data types often vary. OpenGl provides a prefix (GL) to many data types to ease portability and a list of these are given in Table below. Most of the types are self-explanatory. When a **'u'** occurs, it is to indicate all bitsare used for storing numerical values of say a colour level.

C/C++ type	Data Type	OpenGL name & Suffix
signed char	8 bit integer	GLbyte b
short	16 bit integer	GLshort s
int or long	32 bit integer	GLint, Glsizei i
float	32 bit floating point	GLfloat, GLclampf f
double	64 bit floating point	GLdouble, Glclampd d
unsigned char	8 bit unsigned number	GLubyte, GLboolean ub
unsigned short	16 bit unsigned number	GLushort us
unsigned int or long	32 bit unsigned number	GLuint, GLenum, GLbitfield ui

Callback functions

Callbacks simplify event processing for the application developer. Callbacks simplify the process by defining what actions are supported, and automatically handling the user events. A callback function is a function which the library (GLUT) calls when it needs to know how to

process something. E.g. when GLUT gets a key down event it uses the glutKeybourdFunc callback routine to find out what to do with a key press.

The GLU callback functions, **gluBeginPolygon, gluTessVertex, gluNextContour,** and **gluEndPolygon**, are similar to the OpenGL polygon functions. They typically save the data for the triangles, triangle meshes, and triangle fans in user-defined data structures or in OpenGL display lists. To render the polygons, other code traverses the data structures or calls the display lists. Although the callback functions could call OpenGL functions to display polygons directly, this is usually not done, as tessellation can be computationally resource-intensive. It's a good idea to save the data if there is any chance that you want to display it again. The GLU tessellation functions are guaranteed never to return any new vertices, so interpolation of vertices, texture coordinates, or colors is never required.

The following are valid callback functions:

Callback	Description
GLU_TESS_BEGIN	The GLU_TESS_BEGIN callback is invoked like **glBegin** to indicate the start of a (triangle) primitive. The function takes a single argument of type **GLenum**. If you set the GLU_TESS_BOUNDARY_ONLY property to GL_FALSE, the argument is set to either GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP, or GL_TRIANGLES. If you set the GLU_TESS_BOUNDARY_ONLY property to GL_TRUE, the argument is set to GL_LINE_LOOP. The function prototype for this callback is as follows: **void begin(GLenum** type);
GLU_TESS_BEGIN_DATA	GLU_TESS_BEGIN_DATA is the same as the GLU_TESS_BEGIN callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when you call **gluTessBeginPolygon**. The function prototype for this callback is: **void beginData (GLenum** type, **void** * polygon_data);
GLU_TESS_EDGE_FLAG	The GLU_TESS_EDGE_FLAG callback is similar to **glEdgeFlag**. The function takes a single Boolean flag that indicates which edges lay on the polygon boundary. If the flag is GL_TRUE, then each vertex that follows begins an edge that lies on the polygon boundary; that is, an edge which separates an interior region from an exterior one. If the flag is GL_FALSE, then each vertex that follows begins an edge that lies in the polygon interior. The GLU_TESS_EDGE_FLAG callback (if defined) is invoked before the first vertex callback is made. Because triangle fans and triangle strips do not support edge flags, the begin callback is not called with GL_TRIANGLE_FAN or GL_TRIANGLE_STRIP if an edge flag callback is provided. Instead, the fans and strips are converted to independent triangles. The function prototype for this callback is: **void edgeFlag (GLboolean** flag);
GLU_TESS_EDGE_FLAG_DATA	The GLU_TESS_EDGE_FLAG_DATA callback is the same as the GLU_TESS_EDGE_FLAG callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when you call **gluTessBeginPolygon**. The function prototype for this callback is: **void edgeFlagData (GLboolean** flag, **void** * polygon_data);

GLU_TESS_VERTEX	The GLU_TESS_VERTEX callback is invoked between the begin and end callbacks. It is similar to glVertex, and it defines the vertexes of the triangles created by the tessellation process. The function takes a pointer as its only argument. This pointer is identical to the opaque pointer that you provided when you defined the vertex (see **gluTessVertex**). The function prototype for this callback is: **void vertex (void** * vertex_data);
GLU_TESS_VERTEX_DATA	The GLU_TESS_VERTEX_DATA is the same as the GLU_TESS_VERTEX callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when you call **gluTessBeginPolygon**. The function prototype for this callback is: **void vertexData** (void * vertex_data, **void** * polygon_data);
GLU_TESS_END	The GLU_TESS_END callback serves the same purpose as **glEnd**. It indicates the end of a primitive, and it takes no arguments. The function prototype for this callback is: **void end (void)**;
GLU_TESS_END_DATA	The GLU_TESS_END_DATA callback is the same as the GLU_TESS_END callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when you call **gluTessBeginPolygon**. The function prototype for this callback is: **void endData (void** * polygon_data);

GLU_TESS_COMBINE	Call the GLU_TESS_COMBINE callback to create a new vertex when the tessellation detects an intersection, or to merge features. The function takes four arguments: An array of three elements, each of type Gldouble. An array of four pointers. An array of four elements, each of type GLfloat. A pointer to a pointer. The function prototype for this callback is:
	void combine(GLdouble coords[3], **void** * vertex_data[4], **GLfloat** weight[4], **void****outData);
	The vertex is defined as a linear combination of up to four existing vertexes, stored in vertex_data. The coefficients of the linear combination are given by weight; these weights always sum to 1.0. All vertex pointers are valid even when some of the weights are zero. The coords parameter gives the location of the new vertex.
	Allocate another vertex, interpolate parameters using vertex_data and weight, and return the new vertex pointer in outData. This handle is supplied during rendering callbacks. Free the memory sometime after calling **gluTessEndPolygon**.
	For example, if the polygon lies in an arbitrary plane in three-dimensional space, and you associate a color with each vertex, the GLU_TESS_COMBINE callback might look like the following:
	Copy
	void myCombine(GLdouble coords[3], VERTEX *d[4], GLfloat w[4], VERTEX **dataOut)
	{
	VERTEX *newVertex = new_vertex();
	newVertex->x = coords[0];

	newVertex->y = coords[1];
	newVertex->z = coords[2];
	newVertex->r = w[0]*d[0]->r + w[1]*d[1]->r + w[2]*d[2]->r +
	w[3]*d[3]->r;
	newVertex->g = w[0]*d[0]->g + w[1]*d[1]->g + w[2]*d[2]->g +
	w[3]*d[3]->g;
	newVertex->b = w[0]*d[0]->b + w[1]*d[1]->b + w[2]*d[2]->b +
	w[3]*d[3]->b;
	newVertex->a = w[0]*d[0]->a + w[1]*d[1]->a + w[2]*d[2]->a +
	w[3]*d[3]->a;
	*dataOut = newVertex;
	}
	When the tessellation detects an intersection, the GLU_TESS_COMBINE or GLU_TESS_COMBINE_DATA callback (see below) must be defined, and must write a non-**NULL** pointer into data Out. Otherwise the GLU_TESS_NEED_COMBINE_CALLBACK error occurs, and no output is generated. (This is the only error that can occur during tessellation and rendering.)

GLU_TESS_ COMBINE_ DATA	The GLU_TESS_COMBINE_DATA callback is the same as the GLU_TESS_COMBINE callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when you call **gluTessBeginPolygon**. The function prototype for this callback is: **void combineData** (**GLdouble** coords[3], **void***vertex_data[4], **GLfloat** we ight[4], **void** **outData, **void** * polygon_data);
GLU_TESS_ ERROR	The GLU_TESS_ERROR callback is called when an error is encountered. The one argument is of type **GLenum**; it indicates the specific error that occurred and is set to one of the following: GLU_TESS_MISSING_BEGIN_POLYGON GLU_TESS_MISSING_END_POLYGON GLU_TESS_MISSING_BEGIN_CONTOUR GLU_TESS_MISSING_END_CONTOUR GLU_TESS_COORD_TOO_LARGE GLU_TESS_NEED_COMBINE_CALLBACK Call gluErrorString to retrieve character strings describing these errors. The function prototype for this callback is as follows: **void error** (**GLenum** errno); The GLU library recovers from the first four errors by inserting the missing call or calls. GLU_TESS_COORD_TOO_LARGE indicates that some vertex coordinate exceeded the predefined constant GLU_TESS_MAX_COORD in absolute value, and that the value has been clamped. (Coordinate values must be small enough that two can be multiplied together without overflow.) GLU_TESS_NEED_COMBINE_CALLBACK indicates that the tessellation detected an intersection between two edges in the input data, and the GLU_TESS_COMBINE or GLU_TESS_COMBINE_DATA callback was not

	provided. No output will be generated.
GLU_TESS_ERROR_DATA	The GLU_TESS_ERROR_DATA callback is the same as the GLU_TESS_ERROR callback, except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when you call **gluTessBeginPolygon**. The function prototype for this callback is: **void errorData (GLenum** errno, **void** * polygon_data);

The OpenGL Utility Toolkit–GLUT

The OpenGL library only contains rendering commands that are independent of operating system. It is not capable of opening windows or responding to interrupts from a mouse or keyboard. The GLUT libraries (Kilgard, 1996) provide such a facility. The commands to set up a window in which wemay render an object are as follows.

```
glutInit (&argc, argv ); //initialise GLUTlibrary
glutInitDisplayMode ( GLUT_SINGLE | GLUT_RGB ); //Single buffer with RGB colour model
glutInitWindowSize ( width, height ); //window size in pixels
glutInitWindowPosition( x-position, y-position ); // & position onscreen
glutCreateWindow ( "Window Title" ); // create an OpenGL window
glutDisplayFunc ( Rendering Function ); // Display image from rendering Function
glutMainLoop ( ); // get GLUT functions above executing
```

These are usually found in the main function of most OpenGL programs. The GLUT library is particularly important in that it provides a window creation that is independent of both Microsoft and UNIX and thus platform independent.

The toolkit used above can be broadly divided into two components: the Window creation and initialization and the registering and processing of the callback functions associated with different devices on the computer. glutInit() initializes theGLUT library and should be the first function call. glutInitDisplayMode() defines the display modes such as the type of pixel storage and whether single buffered usually used for static displays or double buffered when using animation. glutInitWindowSize(width, height) specify the window size and glutInitWindowPosition ()specifies the position of the GLUT window in (x, y) pixels on the screen. It is agood idea to position this away from the console window if input/output takes place via the latter. glutCreateWindow () creates a window of the size defined with a title bar and name.

After window initialization there are a number of functions associated with calling functions used to draw, update and interact with the display. glutDisplayFunc()is used to specify the function used to render or draw the window contents or redraw them after some interaction with an existing display. The callback functions registered with the program for different tasks by the GLUT kit will becalled when events such as mouse movement, pressing a key or rendering require them. This is akin to real time programming where we wait for an event or interrupt to occur and then execute the appropriate action or function. Program termination occurs if we close the glut window or use a 'break' as a result of a key input or a mouse click.

Our First Program

The height and width in pixels of our window for displaying results in this example is 250 × 250 and the top left hand corner of the window is positioned at (x, y)location (180,90) on the screen of the computer. The window title is written in the top bar as 'Display a Dot'. Comments indicate the role of the OpenGL functions in the program.

```
#include <iostream>
#include <windows.h>
#include <GL/glut.h>
using std::cout;
GLsizei wh = 250 ; // initial height of window
GLsizei ww = 250 ; // initial width of window
void Displaydot ( void );
void MyInit ( );
void Displaydot ( void )
{
        glClear ( GL_COLOR_BUFFER_BIT ); //clear pixel buffer
        glBegin(GL_POINTS); // render with points
        glVertex2i(40,210); //display a point
        glEnd();
        glFlush();
}
void MyInit ( void )
{
        cout << glGetString(GL_VERSION) <<"\n";
        cout << glGetString(GL_VENDOR) <<"\n";
        glClearColor ( 1.0, 1.0, 1.0, 0.0 ); //white background
        glColor3f(0.0f, 1.0f,0.0f); // green drawing color
        glPointSize(10.0); // 10 pixel dot!
        glMatrixMode ( GL_PROJECTION );
        glLoadIdentity ( );
        gluOrtho2D ( 0.0, (GLdouble)ww, 0.0, (GLdouble)wh ); //Display area
}
void main(int argc, char **argv)
{
        glutInit ( &argc, argv );
        glutInitDisplayMode ( GLUT_SINGLE | GLUT_RGB );
        glutInitWindowSize ( ww, wh ); // window size
        glutInitWindowPosition ( 180, 90 ); // & position on screen
        glutCreateWindow ( "Display a Dot" );
        MyInit ( );
        glutDisplayFunc ( Displaydot );
        glutMainLoop ( );
}
```

The function MyInit () is used for setting up the rendering environment which in this case has a white background in the GLUT window using glClearColor (1.0, 1.0, 1.0, 0.0); to set the color; the object to be drawn is green glColor3f (0.0f, 1.0f,0.0f); and setting the size of the dot to be drawn as 10 pixels glPointSize (10.0). Thecolors red, green and blue (RGB) are set to be 1.0 when fully on or 0 when offusing glClearColor () and glColor3f (). The glMatrixMode (GL_PROJECTION) and glLoadIdentity ( ) functions set up the projection identity matrix used by the routine gluOrtho2D (0.0, (GLdouble) width, 0.0, (GLdouble) height) to project the object coordinates onto the screen from (left, bottom) position to the (right, top) position of the screen. The centre of the window is by convention (0, 0) with the top right defined as (1, 1) and the bottom left as (-1, -1) with all points on an object being scaled to these limits. In this example we have a window of250 × 250 pixels superimposed in the main computer screen window of whatever operating system we are using. For the first example this means that the largest object we can completely display has vertices that must lie between values of (-125, -125) and (125, 125). The size of the glut window measured in centimeters is whatever fraction 250 is of the screen display being used (800 × 600 in the author's case).

Drawing Primitives

We have used a limited set of drawing primitives and we will now begin to generalize capabilities. Drawing is accomplished using points, lines, polylines and polygons all of which are defined by vertices. OpenGL provides beyond the primitives of points (GL_POINTS) to include a segmented lines (GL_LINES) facility, to have joined lines (GL_LINE_STRIP) and closed loops (GL_LINE_LOOP). Further surfaces may be patched using polygons, triangles and quadrilaterals. A set of primitives required to render the image occur between the lines glBegin() to glEnd(). glBegin()takes an argument Glvalue that governs how the vertex list is to be processed.

glBegin(Glvalue):start of vertex list where argument Glvalue specifies the geometric primitive to be constructed at vertices.

Value	Meaning
GL_POINTS	individual points at vertices
GL_LINES	individual line segments
GL_LINE_STRIP	connected line segments
GL_LINE_LOOP	connected line segments including first and last
GL_TRIANGLES	vertex triples interpreted as triangles
GL_TRIANGLES_STRIP	linked strip of triangles
GL_TRIANGLES_FAN	linked fan of triangles
GL_QUADS	vertex quadruples interpreted as 4 sided polygons
GL_QUADS_STRIP	linked strip of quadrilaterals
GL_POLYGON	boundary of simple, convex polygon
glEnd()	

Drawings pixels, lines, polygons using OpenGL

All geometric primitives are eventually described in terms of their vertices - coordinates that define the points themselves, the endpoints of line segments, or the corners of polygons. In any

OpenGL implementation, floating-point calculations are of finite precision, and they have round-off errors. Consequently, the coordinates of OpenGL points, lines, and polygons suffer from the same problems. Another more important difference arises from the limitations of a raster graphics display. On such a display, the smallest displayable unit is a pixel, and although pixels might be less than 1/100 of an inch wide, they are still much larger than the mathematician's concepts of infinitely small (for points) or infinitely thin (for lines). When OpenGL performs calculations, it assumes points are represented as vectors of floating-point numbers. However, a point is typically drawn as a single pixel, and many different points with slightly different coordinates could be drawn by OpenGL on the same pixel.

Points

A point is represented by a set of floating-point numbers called a vertex. All internal calculations are done as if vertices are three-dimensional. Vertices specified by the user as two-dimensional (that is, with only x and y coordinates) are assigned a z coordinate equal to zero by OpenGL.

OpenGL works in the homogeneous coordinates of three-dimensional projective geometry, so for internal calculations, all vertices are represented with four floating-point coordinates (x, y, z, w). If w is different from zero, these coordinates correspond to the Euclidean three-dimensional point (x/w, y/w, z/w).

To control the size of a rendered point, use **glPointSize()** and supply the desired size in pixels as the argument.

void **glPointSize**(GLfloat size);

Sets the width in pixels for rendered points; size must be greater than 0.0 and by default is 1.0.

Lines

In OpenGL, the term line refers to a line segment, not the mathematician's version that extends to infinity in both directions. There are easy ways to specify a connected series of line segments, or even a closed, connected series of segments. In all cases, though, the lines constituting the connected series are specified in terms of the vertices at their endpoints.

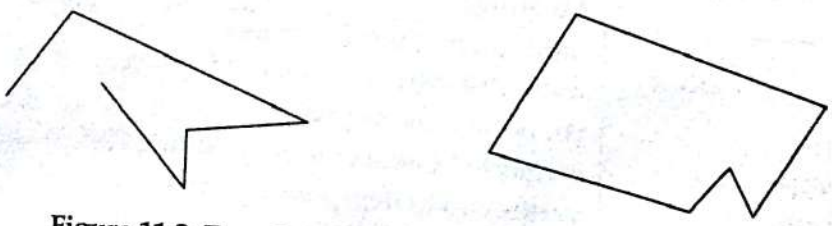


Figure 11.2: Two Connected Series of Line Segments

With OpenGL, we can specify lines with different widths and lines that are stippled in various ways - dotted, dashed, drawn with alternating dots and dashes, and so on.

Drawing a line with OpenGL is apparently quite simple using the following code

```
glBegin(GL_LINES);
        glVertex2i(x0, y0);
        glVertex2i(x1, y1);
glEnd();
```

Figure 11.3: Step effect when plotting lines

The sample below shows how to draw 2D points and lines in OpenGL. We can consider it as the 'Hello World' program of OpenGL. This is the program of OpenGL in C/C++.

```
#include <GL/glut.h>
void init2D(float r, float g, float b)
{
        glClearColor(r,g,b,0.0);
        glMatrixMode (GL_PROJECTION);
        gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}
void display(void)
{
        glClear(GL_COLOR_BUFFER_BIT);
        glColor3f(1.0, 0.0, 0.0);
        glBegin(GL_POINTS);//draw two points
        for(int i = 0; i < 10; i++)
        {
                glVertex2i(10+5*i,110);
        }
        glEnd();
        glBegin(GL_LINES);//draw a line
                glVertex2i(10,10);
                glVertex2i(100,100);
        glEnd();
        glFlush();
}
void main(int argc,char *argv[])
{
        glutInit(&argc,argv);
        glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize (500, 500);
        glutInitWindowPosition (100, 100);
        glutCreateWindow ("points and lines");
        init2D(0.0,0.0,0.0);
        glutDisplayFunc(display);
        glutMainLoop();
}
```

Polygon

Polygons are typically drawn by filling in all the pixels enclosed within the boundary, but you can also draw them as outlined polygons or simply as points at the vertices. A filled polygon might be solidly filled or stippled with a certain pattern. Although the exact details are omitted here, filled polygons are drawn in such a way that if adjacent polygons share an edge or vertex, the pixels making up the edge or vertex are drawn exactly once—they're included in only one of the polygons. This is done so that partially transparent polygons don't have their edges

drawn twice, which would make those edges appear darker (or brighter, depending on what color you're drawing with).

A polygon has two sides—front and back—and might be rendered differently depending on which side is facing the viewer. This allows you to have cutaway views of solid objects in which there is an obvious distinction between the parts that are inside and those that are outside. By default, both front and back faces are drawn in the same way. To change this, or to draw only outlines or vertices, use

glPolygonMode().

We can draw polygon by following code of segment;

```
glBegin(GL_POLYGON); // Draw A Quad
glVertex3f(-0.5f, 1.0f, 0.0f); // Top Left
glVertex3f(-1.0f, 0.0f, 0.0f); // Left
glVertex3f(-0.5f, 1.0f, 0.0f); // Bottom Left
glVertex3f(0.5f, 1.0f, 0.0f); // Top Right
glVertex3f(1.0f, 0.0f, 0.0f); // Right
glVertex3f(0.5f, 1.0f, 0.0f); // Bottom Right
glEnd();
```

Complete example of drawing a polygon in OpenGL by using c/c++ as below:

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
#include <glut.h>
void myInit(void)
{
        glClearColor(1.0,1.0,1.0,0.0);
        glColor3f(1.0f,0.0f,0.0f);
        glPointSize(4.0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0,800.0,0.0,600.0);
}
void myDisplay(void)
{
        glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_POLYGON);
        glVertex2i(100,100);
        glVertex2i(100,300);
        glVertex2i(400,300);
        glVertex2i(600,150);
        glVertex2i(400,100);
```

```
    glEnd(); glFlush();
}
int main(int argc,char * argv[])
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(800,600);
    glutInitWindowPosition(100,100);
    glutCreateWindow("opengl Window");
    glutDisplayFunc(myDisplay);
    myInit(); glutMainLoop();
    getch();
    return 0;
}
```

Creating a Window

Now that we have our library, we can begin to design our interface by creating our window inside our int main() function:

```
#ifndef WIN32 //if using windows then do windows specific stuff.
#define WIN32_LEAN_AND_MEAN //remove MFC overhead from windows.h which can cause slowness
#define WIN32_EXTRA_LEAN
#include <windows.h>
#endif
#include <GL/gl.h>
#include <GL/glut.h>
#include <GL/glu.h>
void display()
{

}
int main(int argc, char *argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
    glutInitWindowSize(800,600);
    glutCreateWindow("Hello World");
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

We first call glutInit(), which starts up GLUT for our use. Next, we set up a display mode. For now, just use the settings we've provided here for glutInitDisplayMode. Then we tell GLUT how big we want our window to be; 800 by 600 is OK for now. Finally, we actually create the

window with glutCreateWindow(), passing the window title as an argument, and give control of our program to GLUT through glutMainLoop().

Color commands

OpenGL has two color modes: the RGBA mode and color index mode. In RGBA mode, a color is specified by three intensities (for the Red, Green, and Blue components of the color) and optionally a fourth value, Alpha, which controls transparency. The function

glColor4f(red, green, blue, alpha)

Maps available red, green, and blue intensities onto (0.0, 1.0) where 0.0 means that the color component is absent ((0.0, 0.0, 0.0) is black) and 1.0 is a saturated color ((1.0, 1.0, 1.0) is white.) The number of bits used to represent each color depends upon the graphics card. Current graphics cards have several Mbytes of memory, and use 24 or 32 bits for color (24-bit: 8 bits per color; 32-bit: 8 bits per color + 8 bits padding or transparency). The term bit plane refers to an image of single-bit values. Thus, a system with 24 bits of color has 24 bit planes.

For a system with 8 bit planes, 28 = 256 different colors could be displayed simultaneously. These 256 colors could be selected from a set of 2m colors. For m = 24 this gives 22 4 ≈ 16 million colors. The color table could be altered to give different palettes of colors. OpenGL supports this mode, called color index mode.

Using glColor3f

glColor3f() takes 3 arguments: the red, green and blue components of the color. After we use glColor3f, everything draw will be in that color. For example, consider this display function:

```
void display()
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glColor3f(0.5f, 0.0f, 1.0f); // (0.5, 0, 1) is half red and full blue, giving purple.
        glBegin(GL_QUADS);
        glVertex2f(-0.75, 0.75);
        glVertex2f(-0.75, -0.75);
        glVertex2f(0.75, -0.75);
        glVertex2f(0.75, 0.75);
        glEnd();
        glutSwapBuffers();
}
```

Giving Individual Vertices Different Colors

glColor3f can be called in between glBegin and glEnd. When it is used this way, it can be used to give each vertex its own color. The resulting rectangle is then shaded with an attractive color gradient.

```
void display()
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
glBegin(GL_QUADS);
glColor3f(1.0f, 1.0f, 1.0f); // make this vertex purple
glVertex2f(-0.75, 0.75);
glColor3f(1.0f, 1.0f, 1.0f); // make this vertex red
glVertex2f(-0.75, -0.75);
glColor3f(1.0f, 1.0f, 1.0f); // make this vertex green
glVertex2f(0.75, -0.75);
glColor3f(1.0f, 1.0f, 1.0f); // make this vertex yellow
glVertex2f(0.75, 0.75);
glEnd();
glutSwapBuffers();
}
```

OpenGl Color Codes

For Background color:

gl.glClearColor(float red, float green, float blue, float alpha)

```
glClearColor(0.0,0.0,1.0,0.0);//dark blue
```

Fill color:

```
glColor4f(1.0f, 0.0f, 0.0f, 0.0f);//red
glColor4f(1.0f, 1.0f, 1.0f, 0.0f);//white
glColor4f(1.0f, 1.0f, 0.0f, 0.0f);//yellow
glColor4f(1.0f, 0.0f, 1.0f, 0.0f);//purple
glColor4f(0.0f, 1.0f, 1.0f, 1.0f);//light blue
glColor4f(1.0f, 0.5f, 0.0f, 0.0f);//orange/brown
glColor3f(0.0f, 1.0f, 0.0f);//Green
glColor3f(0.0f, 0.0f, 1.0f);//Blue
glColor3f(0.5f, 1.0f, 1.0f);//cyan
glColor3f(0.0f, 0.0f, 0.0f);//Black
glColor3f(1.0f, 0.0f, 1.0f);//Purple
glColor3f(1.0f, 0.5f, 0.0f);//Orange
glColor3f(0.5f, 0.5f, 0.5f);//Violet
glColor3f(0.0f, 0.5f, 0.5f);//Blue-Green
glColor3f(0.0f, 0.5f, 1.0f);//baby Blue
glColor3f(2.0f, 0.5f, 1.0f);//Lilac
glColor3f(0.1f, 0.1f, 0.1f);//Dark grey
glColor3f(0.1f, 0.0f, 0.1f);//Dark Purple
glColor3f(0.1f, 0.1f, 0.0f);//Bronze
glColor3f(0.0f, 0.1f, 0.1f);//Dark blue
glColor3f(0.0f, 0.1f, 0.0f);//Forest Green
glColor3f(0.1f, 0.0f, 0.0f);//Brown
```

Basic Lighting

Lighting in the real world is extremely complicated and depends on way too many factors, something we can't afford to calculate on the limited processing power we have. Lighting in OpenGL is therefore based on approximations of reality using simplified models that are much easier to process and look relatively similar. These lighting models are based on the physics of light as we understand it. One of those models is called the Phong lighting model. The major building blocks of the Phong model consist of 3 components: ambient, diffuse and specular lighting.

- **Ambient lighting:** even when it is dark there is usually still some light somewhere in the world (the moon, a distant light) so objects are almost never completely dark. To simulate this we use an ambient lighting constant that always gives the object some color. **Ambient** illumination is light that's been scattered so much by the environment that its direction is impossible to determine - it seems to come from all directions. Backlighting in a room has a large ambient component, since most of the light that reaches your eye has first bounced off many surfaces. A spotlight outdoors has a tiny ambient component; most of the light travels in the same direction, and since you're outdoors, very little of the light reaches your eye after bouncing off other objects. When ambient light strikes a surface, it's scattered equally in all directions.

- **Diffuse lighting:** simulates the directional impact a light object has on an object. This is the most visually significant component of the lighting model. The more a part of an object faces the light source, the brighter it becomes. The diffuse component is the light that comes from one direction, so it's brighter if it comes squarely down on a surface than if it barely glances off the surface. Once it hits a surface, however, it's scattered equally in all directions, so it appears equally bright, no matter where the eye is located. Any light coming from a particular position or direction probably has a diffuse component.

- **Specular lighting:** it simulates the bright spot of a light that appears on shiny objects. Specular highlights are often more inclined to the color of the light than the color of the object. Specular light comes from a particular direction, and it tends to bounce off the surface in a preferred direction. A well-collimated laser beam bouncing off a high-quality mirror produces almost 100 percent specular reflection. Shiny metal or plastic has a high specular component, and chalk or carpet has almost none. You can think of secularity as shininess.

Example: Drawing a Lit Sphere

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
void init(void)
{
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_shininess[] = { 50.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    glClearColor (0.0, 0.0, 0.0, 0.0);
```

```
    glShadeModel (GL_SMOOTH);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);

}
void display(void)
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glutSolidSphere (1.0, 20, 16);
    glFlush ();

}
void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
      glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
    else
      glOrtho (-1.5*(GLfloat)w/(GLfloat)h,1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

Exercise

1. Why use SDL and OpenGL instead of just OpenGL?
2. What would be a good way to abstract OpenGL-Concepts in a game engine?
3. Why are things bound instead of being passed directly as parameters?
4. What are the advantages of matrix multiplication over individual variables (graphics)?
5. How to better isolate JOGL or LWJGL3 dependency from game clients?
6. What design pattern best suits managing handles to objects, without passing handles or Manager around?
7. Original GLUT library was open source, but license prevented releasing any changes to it. What could be the rationale behind such license?
8. Why have hardware-accelerated vector graphics not taken off?
9. OpenGL programming vs. Blender Software, which is better for custom video creation?
10. What is OpenGL (or Open Graphics Library)? Give Brief introduction about it.
11. Name major competitors of OpenGL. Also give main advantages & disadvantages OpenGL have over other graphics libraries in the market.
12. Give the main advantages that OpenGL have over Microsoft's proprietary Direct3D.
13. OpenGL is written in which language? Is it possible to implement (or use) same library in programming languages other than that?
14. Is OpenGL API platform independent? Is it possible to port the library to embedded systems such as mobile phones etc?
15. Name few OpenGL related libraries that simplify the programming task by providing a layer of abstraction over OpenGL.
16. How OpenGl can be considered as a state machine?
17. What is term named Rasterization means? How is it different from vector graphics?
18. How do we clear a window in OpenGL? Also write a code snippet for the same.
19. How to apply color to a geometrical object? Give the syntax of glColor3f() method.
20. What is difference between glColor3f() & glClearColor()?

❑❑❑

Chapter

12

Areas, Text and Colors

STRUCTURE

12.1 Introduction

12.2 What is Color?

12.3 Color models

- RGB Color model
- HSV color model

12.4 Conversion between RGB and HSV Color Models

12.5 Conversion between HSV and RGB Color Models

12.6 Filling Areas

12.7 Displaying Text

12.8 Grey Images and Intensities

12.9 Color interpolation

Introduction

Filling areas is applied in the context of drawing thick lines where thick lines are considered as long rectangles. But filling areas and polygons is also a general technique of computer graphics that is also needed as a drawing mode. This topic contains in addition to techniques for filling areas also basic models for colors and a short introduction to handling text in images.

Text mode is a computer display mode in which content is internally represented on a computer screen in terms of characters rather than individual pixels. Typically, the screen consists of a uniform rectangular grid of character cells, each of which contains one of the characters of a character set. Text mode is contrasted to all points addressable (APA) mode or other kinds of computer graphics modes.

A color model is an orderly system for creating a whole range of colors from a small set of primary colors. There are two types of color models, those that are subtractive and those that are additive

What is Color?

Color is a sensation produced by the human eye and nervous system. It is related to light. Color is a significant component of computer aided visualization of information, concepts and ideas. The use of color in all applications of computer graphics enhances the image, clarifies the information presented, and helps distinguish features that are obscure in black and white pictures. Color is used to differentiate elements in the diagrams so that the comparative information is read and understood rapidly and accurately. Color visualization techniques increase the amount of information that can be integrated into the visual message or picture, and thus create layers of information. It is clear that research and discovery can be supported and enhanced by color application, or inhibited by ineffectual use of color. In order to effectively utilize color in the visualization of ideas, information, or concepts, the role of color in various applications must be examined, and the perceptual behavior of color must be delineated. The basic principles of color theory have been discussed to provide a fundamental understanding of the characteristics of color and the manner in which colors interact with one another. Based on these principles, the user will be better able to utilize color in an effective manner relative to specific applications in color graphics.

Color Model

A color space or color model is a method by which we can specify, create and visualize color. A color model is a method for explaining the properties or behavior of color within particular context. It is a mathematical way of representing a set of colors. A color model is an orderly system for creating a whole range of colors from a small set of primary colors. There is several established color models used in computer graphics, **RGB, CMY, CMYK, YIQ** etc. But the two most common are the RGB model (Red-Green-Blue) for computer display and the CMYK model (Cyan-Magenta-Yellow-black) for printing. Several Color models, but fall into two broad categories:

- Additive Color Model and
- Subtractive Color Model

Additive Color Model

Adaptive synthesis is based on adding or adding individual wavelengths. This method is simpler to describe by adding three light beams, red, green, and blue. It is possible to achieve a wide range of different colors with different beam intensity ratios. This is possible because the maximum sensitivity of individual lumps to the human eye mesh partially coincides with the red, green, and blue part of the visible spectrum, and depending on the level of excitement of individual lumps the observer will experience a color. (For example, if only the solids sensitive to the blue part of the spectrum are excited, the brain of the observer interprets it as a blue color. To display colors on monitors, the RGB system (is, red-green-blue) is based on the principle of additive color mixing. This is about mixing the light of different wavelengths, not the mixing of pigments that reflects some parts of the spectrum, while others absorb. These colors are the basic colors for the additive system; each pixel is defined with 3 data that represent the value of these 3 basic colors. The monitors emit light and the colors on the monitor are obtained by additive color pixel combinations.

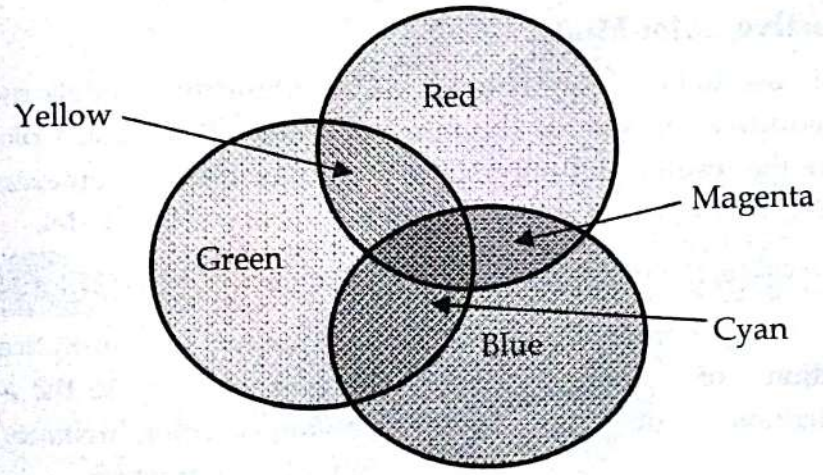


Figure: 12.1: Additive color model

Subtractive Color Model

The subtractive synthesis is based on the absorption or subtraction of certain wavelengths from white light. Selective wavelength removal is performed using a filter. With the green filter we subtract the red, purple, subtract the green, and the yellow subtract the blue part of the spectrum. A wide degree of absorption of the individual wavelengths is possible to achieve a wide range of different colors. Of course, in this case, the color the observer feels depends on what has happened to his eye. If we use a filter that absorbs the green part of a visible spectrum of white light, the blue and red parts remain, which means that the eye of the observer will be excited by the susceptible particles on the blue and the reddish-sensitive red particles that will interpret the brain as a purple color. The pigment system is what we learned at school, at the hours of the artistic – that the blending of red and yellow gives orange, blue and red purple, and yellow and blue green, while blending these colors in even order gives black. Pigments do not emit light, but reflect, they accurately reflect the light of some wavelength. This is a

subtractive system, i.e. a system of colors that arises by subtraction, and not by the addition of color. In fact, the basic colors in this system are magenta, yellow and cyan and their combination is black, but the print uses the CMYK system (cyan-magenta-yellow-key, where the "key" is black, how CMY would not need unnecessary consumption to create a black color).

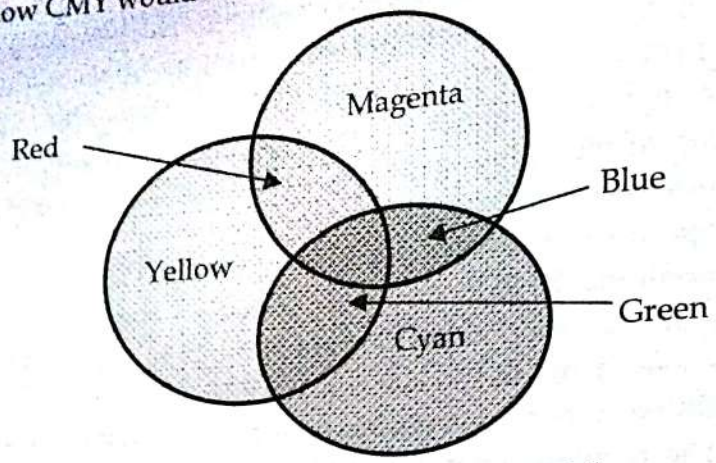


Figure: 12.2: Subtractive color model

Additive vs. Subtractive Color Model

Additive color models use light to display color while subtractive models use printing inks. Colors perceived in additive models are the result of transmitted light. Colors perceived in subtractive models are the result of reflected light. RGB color model is an example of additive color model and CMYK color model is an example of subtractive color model.

Additive Color Model	Subtractive Color Model
1. Additive color mixing occurs with the simultaneous action of various color sensations - 'irritants' on the eye.	1. subtractive (multiplicative) color mixing, contrary to the additive, is not mixing of color 'irritants', but creating color by subtraction.
2. Additive color models use light to display color	2. Subtractive models use printing inks.
3. Additive color synthesis occurs when three light zones are mixed optically (red, green, and blue).	3. Subtractive synthesis occurs by mixing the basic material colors (cyan, magenta and yellow).
4. Additive color synthesis: green + red = yellow, blue + red = magenta, blue + green = cyan.	4. Subtractive mixing occurs: yellow + magenta = red, yellow + cyan = green, magenta + cyan = blue.
5. RGB color model is an example of additive color model	5. CMYK color model is an example of subtractive color model.
6. Additive colors are colors which are "pure", i.e. colors add up to form white light. A RED light looks RED because it emits RED light.	6. Subtractive colors are "impure". You perceive RED pigment to be RED because it reflects RED light

Colors perceived in additive models are the result of transmitted light	and absorbs everything except RED light falling on it. 7. Colors perceived in subtractive models are the result of reflected light.

RGB Color model

The RGB color model used in color CRT monitors. In this model, Red, Green and Blue are added together to get the resultant color white. Each color point within the bounds of the cube is represented as the triple (R, G, B).Where value for R, G, B are assigned in the range from 0 to 1. Here RGB color place together at 120 degree. RED + GREEN + BLUE = WHITE (contribute) all other colors are generated from these three primary colors. It is a type of Additive color model.

Magenta = red + blue

Cyan = green +blue

Yellow = red + green

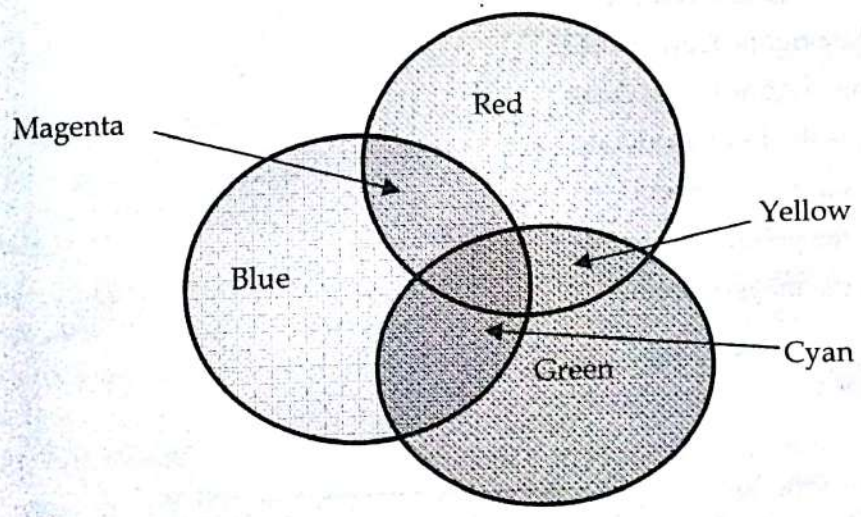


Figure: 12.3: RGB color model

In the RGB color model, colors are represented by varying intensities of red, green and blue light. The intensity of each of the red, green and blue components is represented on a scale from 0 to 255 with 0 being the least intensity (no light emitted) to 255 (maximum intensity). It is used in hardware applications like PC monitors, cameras and scanners.

The main purpose of the RGB color model is for the sensing, representation and display of images in electronic systems, such as televisions and computers, though it has also been used in conventional photography.

RGB Color Model: Geometrical Structure

RGB color model geometrically represented as a unit cube defined on R, G, and B primary axes. Each point on this cube represents a color. The origin represents black (0, 0, 0) and the vertex with coordinates (1, 1, 1) is white. Vertices of the cube on the axes represent primary colors, and

remaining vertices represent the complementary color. The line joining black and white represents gray scale and has equal components of R, G, B.

RGB is a basic color model for TV, on computers and for web graphics but it cannot be used for print production. Also in RGB color model, the higher values of R, G and B, the brighter is the color. If R=G=B the color will be a shade of gray. Our system will generally display RGB using 24-bit color. In 24-RGB color model there are 256 variations for each of these additive colors of RGB.

Thus, RGB color model has=256 reds * 256 greens * 256 blue

(24-bit)= 16,777,216 possible colors.

The intensity of each R, G and B components are represented on a scale from 0 to 255, where,

0- Least intensity(no light emitted)

255- Maximum intensity &

127- is the half intensity

Thus, (0, 0, 0) is the darkest black color

(255, 0, 0) is the brightest, red color

(0, 255, 0) is the brightest, green color

(0, 0, 255) is the brightest, blue color

(255, 255, 255) is the brightest white color

(127,127,127) is the gray color

(255, 255, 0) is the yellow color and

(255, 0, 255) is the magenta color.

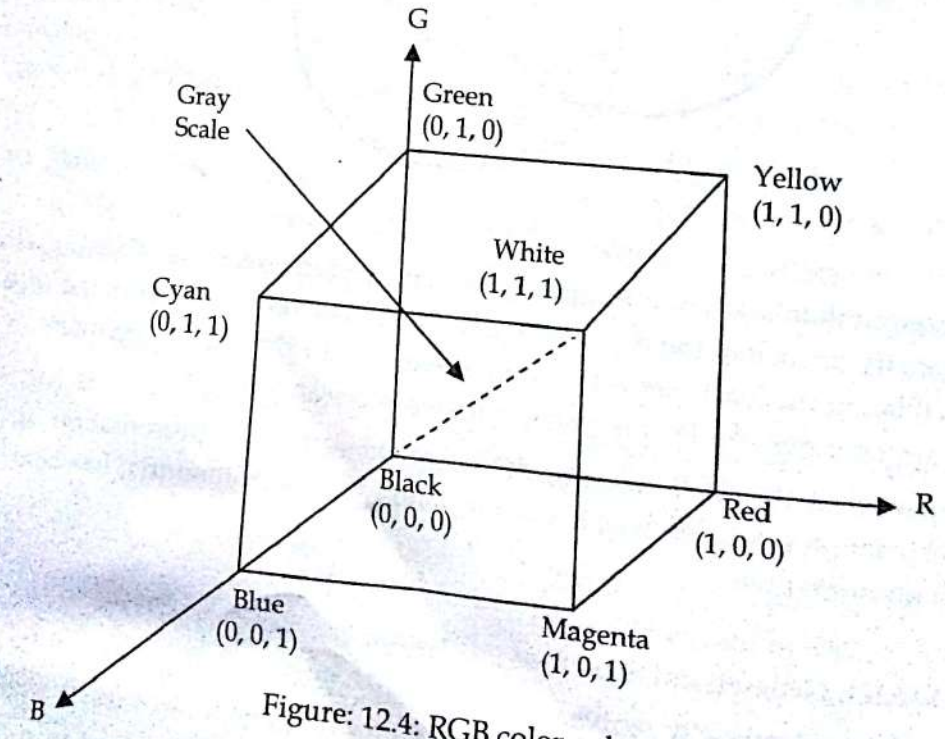


Figure: 12.4: RGB color cube

HSV Color Model

The HSV color model attempts to characterize colors according to their hue, saturation, and value (brightness). **Hue-** The hue of a color identifies what is commonly called color. For example, all reds have a similar hue value whether they are light, dark, and intense. The hue is determined by the dominant wavelength. The **saturation** of a color identifies how pure or intense the color is. A fully saturated color is deep and brilliant as the saturation decreases; the color gets paler and more washed out until it eventually fades to neutral. The HSV color wheel is sometimes depicted as a cone or cylinder, but always with these three components:

Hue

The **Hue** is the color portion of the color model, and is expressed as a number from 0 to 360 degrees:

Color	Angle
Red	0-60
Yellow	60-120
Green	120-180
Cyan	180-240
Blue	240-300
Magenta	300-360

Saturation

Saturation is the amount of gray in the color, from 0 to 100 percent. A faded effect can be had from reducing the saturation toward zero to introduce more gray. However, saturation is sometimes viewed on a range from just 0-1, where 0 is gray and 1 is a primary color.

Value (or Brightness)

Value works in conjunction with saturation and describes the brightness or intensity of the color, from 0-100 percent, where 0 is completely black and 100 is the brightest and reveals the most color.

HSV Color Model: Geometrical Structure

HSV is derived from the RGB model. It is the polar coordinate representation of RGB color model. It has Hex-cone like geometrical structure. Hex cone origin (apex) represents black. Vertical axis represents Value V, varies from 0 at the apex to 1 at the top. Saturation S is measured along horizontal axis. It varies from 0 to 1. Hex cone boundary represents various hues. The **Hue** is represented as an angle, ranging from 0 degree at red through 360 degree. Vertices of hex cone are separated by 60 degree.

The Hue-Saturation-Value model is oriented towards the user. The allowed coordinates fill a six sided pyramid the 3 top faces of the color cube as base. Note that at the same height colors of different perceived brightness are positioned. Value is given by the height; saturation is coded in the distance from the axes and hue by the position on the boundary. The HSV color space is

used when selecting colors for paint or ink because HSV better represents how people relate to colors than does the RGB color space.

The HSV color wheel is also used to generate high-quality graphics. Although less well known than its RGB and CMYK cousins, the HSV approach is available in many high-end image editing software programs. Selecting an HSV color begins with picking one of the available hues, which is how most humans relate to color, and then adjusting the shade and brightness value

The HSV color space is widely used to generate high quality computer graphics or is used to select various different colors needed for a particular picture. An HSV color wheel is used to select the desired color. A user can select the particular color needed for the picture from the color wheel. It is used in color pickers, image editing software, image analysis, computer vision etc.

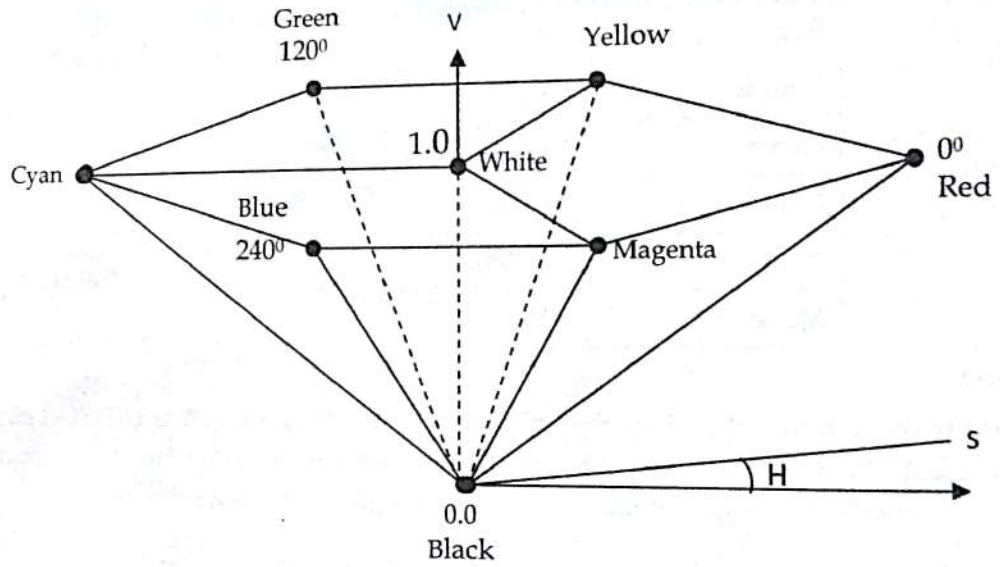


Figure: 12.5: HSV color model

- The outer edge of the top of the cone is the color wheel, with all the pure colors. The H parameter describes the angle around the wheel.
- The S (saturation) is zero for any color on the axis of the cone; the center of the top circle is white. An increase in the value of S corresponds to a movement away from the axis.
- The V (value or lightness) is zero for black. An increase in the value of V corresponds to a movement away from black and toward the top of the cone.

Conversion between RGB and HSV Color Models

Converting from RGB to HSV Color vision can be processed using RGB color space or HSV color space. RGB color space describes colors in terms of the amount of red, green, and blue present. HSV color space describes colors in terms of the Hue, Saturation, and Value. In situations where color description plays an integral role, the HSV color model is often preferred over the RGB model. The HSV model describes colors similarly to how the human eye tends to perceive color. RGB defines color in terms of a combination of primary colors, whereas, HSV

describes color using more familiar comparisons such as color, vibrancy and brightness. The basketball robot uses HSV color space to process color vision.

The color camera, on the robot, uses the RGB model to determine color. Once the camera has read these values, they are converted to HSV values. The HSV values are then used in the code to determine the location of a specific object/color for which the robot is searching. The pixels are individually checked to determine if they match a predetermined color threshold.

Color name	Hex	(R, G, B)	(H, S,V)
Black	#000000	(0,0,0)	(0°, 0%, 0%)
White	#FFFFFF	(255,255,255)	(0°, 0%, 100%)
Red	#FF0000	(255,0,0)	(0°, 100%, 100%)
Lime	#00FF00	(0,255,0)	(120°, 100%, 100%)
Blue	#0000FF	(0,0,255)	(240°, 100%, 100%)
Yellow	#FFFF00	(255,255,0)	(60°, 100%, 100%)
Cyan	#00FFFF	(0,255,255)	(180°, 100%, 100%)
Magenta	#FF00FF	(255,0,255)	(300°, 100%, 100%)
Silver	#C0C0C0	(192,192,192)	(0°, 0%, 75%)
Gray	#808080	(128,128,128)	(0°, 0%, 50%)
Maroon	#800000	(128,0,0)	(0°, 100%, 50%)
Olive	#808000	(128,128,0)	(60°, 100%, 50%)
Green	#008000	(0,128,0)	(120°, 100%, 50%)
Purple	#800080	(128,0,128)	(300°, 100%, 50%)
Teal	#008080	(0,128,128)	(180°, 100%, 50%)
Navy	#000080	(0,0,128)	(240°, 100%, 50%)

Table: RGB to HSV color table

RGB to HSV conversion formula

Input: RGB

Output: HSV

Step1: normalize the RGB values to be in the range [0, 1]

r = R/ 255

g = G/ 255

b = B/ 255

Step2: Find the difference between max and min values

M = max(r, g, b), m = min(r, g, b).

And, difference Δ = M-m.

Step3: Calculate value V

V = max(r, g, b)

Step4: Calculate saturation S

If Δ = 0, then S = 0, otherwise S = Δ / M

Step5: Calculate hue H

If M=r then H = 60*((g-b / Δ) * mod 6)

If M=g then H = 60*(2 + (b-r)/Δ)

If M=b then H = 60(4 + (r-g)/ Δ)

Step6: output HSV

The calculated H, S, and V are the output of the algorithm.

The generic C-code for converting RGB color to HSV color is given below.

```
/* r, g, b values are from 0 to 1
        h = [0, 360], s = [0, 1], v = [0, 1]
        if s == 0, then h = -1 (undefined) */
void RGBtoHSV( float r, float g, float b, float *h, float *s, float *v )
{
        Float min, max, delta;
        Min = MIN( r, g, b );
        Max = MAX( r, g, b );
        *v = max;
        Delta = max - min;
        if( max != 0 )
                *s = delta / max;
        else
        {
                // r = g = b = 0
                // s = 0, v is undefined
                *s = 0;
                *h = -1;
                return;
        }
        if( r == max )
                *h = ( g - b ) / delta;
        else if( g == max )
                *h = 2 + ( b - r ) / delta;  // between cyan & yellow
```

```
else
        *h = 4 + ( r - g ) / delta;  // between magenta & cyan
*h *= 60;          // degrees
if( *h < 0 )
        *h += 360;

}
```

Conversion between HSV and RGB Color Models

If HSV colors parameters are made available to a user of graphics package, these parameters are transformed to RGB setting needed in this transformation, we first consider how the HSV hex cone corresponds to the projection seen by looking along the principal diagonal of the RGB color cube from white toward black, as shown in figure below;

Each sub cube, when viewed along its main diagonal, is like the hexagon in figure except smaller. Each plane of constant V in HSV space corresponds to such a view of a sub tube in RGB space. The diagonal of RGB space becomes the V exist of HSV space. Thus, we can see intuitively the corresponding between RGB and HSV

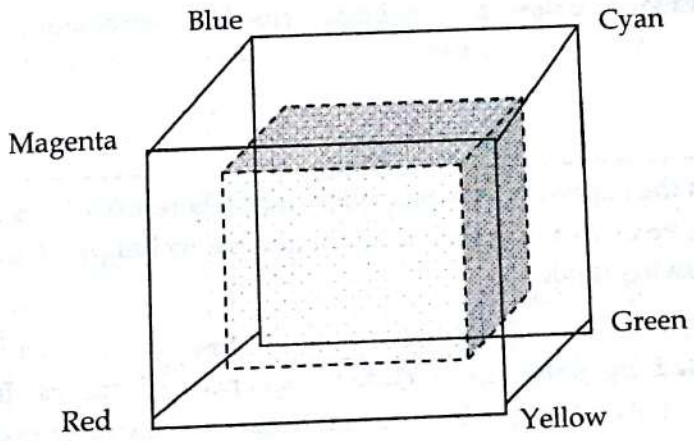


Figure 12.6: RGB cube and its sub cube

HSV to RGB conversion formula

Input: HSV

Output: RGB

1. When $0 \le H < 360$, $0 \le S \le 1$ and $0 \le V \le 1$.
   - $C = V \times S$
   - $X = C \times (1 - |(H / 60^\circ) \bmod 2 - 1|)$
   - $m = V - C$

$$(R', G', B') = \begin{cases} (C, X, 0), & 0^0 \le H < 60^0 \\ (X, C, 0), & 60^0 \le H < 120^0 \\ (0, C, X), & 120^0 \le H < 180^0 \\ (0, X, C), & 180^0 \le H < 240^0 \\ (X, 0, C), & 240^0 \le H < 300^0 \\ (C, 0, X), & 300^0 \le H < 360^0 \end{cases}$$

2. $(R, G, B) = (R'+ m, G'+ m, B'+ m)$.

Applications

- RGB model is used in hardware devices such as computer monitors, cameras, and scanners.
- CMYK model is used in hard copy devices such as printer, plotter.
- And HSV color model is used in real life applications to describe perception of viewer.

Note: Color Gamut: The range of colors that can be described by combinations of other colors is called a color gamut.

Filling Areas

Filling areas is applied in the context of drawing thick lines where thick lines are considered as long rectangles. But filling areas and polygons is also a general technique of computer graphics that is also needed as a drawing mode.

Areas are usually bounded by polygons or closed sequences of curves. In order to fill a polygon, points inside the polygon must be distinguished from exterior ones. For polygons whose edges do not intersect, it is obvious which points should be considered as inner and outer points. For polygons with intersecting edges it is not immediately clear what its inner parts are. The odd parity rule provides a definition of inner and outer points based on the following considerations.

If one starts to move along a line from an inner point of a polygon in one direction, then a bounding edge of the polygon must be reached at some point on the line. If the polygon is not convex, then it might happen that other edges are met, when the movement along the line is continued. Since the movement along the line was started at an inner point, one changes from the inner to the outer part of the polygon when the first polygon edge is met. After the second intersection point along the line with an edge, the polygon is entered again. So each time, when the line meets an edge, a change from the inside to the outside of the polygon or vice versa takes place. Since the polygon is bounded, the last edge that is met along the line must correspond to a change from the inside to the outside of the polygon. Because the starting point

was assumed to be an inner point of the polygon, the intersection point with the first edge must also represent a change from inside to outside.

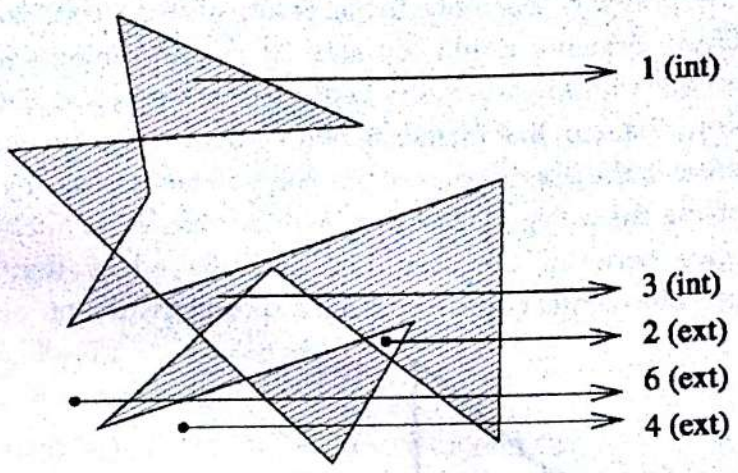
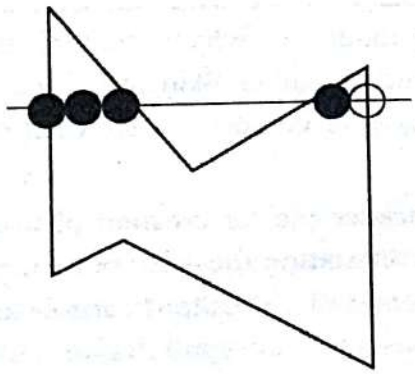


Figure 12.7: Odd parity rule

Figure illustrates the application of the odd parity rule to selected points inside and outside a polygon. For each point a line is drawn and the number of intersection points of the line with edges of the polygon is given in the figure. If the number is odd, then the point is an interior ("int") point of the polygon, otherwise it is an exterior ("ext") point.



The odd parity rule is a useful mathematical definition of interior and exterior points of a polygon, but it is not suited for implementation, since the computational costs would be unacceptable if the rule is applied separately to each pixel. Instead, a scan line technique is applied. Scan line techniques are very common in computer graphics. They carry out computations along a line, the scan line, usually along lines parallel to one of the coordinate axes. For each pixel row, the corresponding line is considered, and the intersection points of the line with polygon edges are determined. These intersection points are sorted in ascending order with respect to their x-coordinates. This means the scan line enters the polygon at xi, leaves the polygon at x2, and enters it again at x3 until it finally leaves the polygon at xn. Therefore, exactly the pixels between x1 and x2, between x3 and x4, etc. and between xn-1 and xn must be drawn for filling the polygon. The number n of intersection points must be even including the possible value zero.

For the implementation of the scan line technique, some specific problems need extra attention. One of these problems is clipping. It is necessary to compute all intersection points of the scan line with the polygon. But only those pixels need to be drawn that lie within the clipping area. Therefore, drawing might not start at the first intersection point x1. Another problem occurs when the scan line intersects vertices of the polygon. These cases need special treatment. In Fig. below, the scan line intersects two vertices of the polygon. At the first vertex a change from the exterior to the interior part of the polygon takes place. At the second vertex, the scan line remains outside the polygon. The treatment of vertices as intersection points requires considering the angles between the scan line and the edges that are attached to the corresponding vertices. Horizontal edges also need a special treatment.

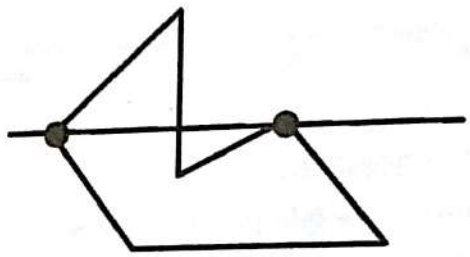


Figure 12.8: A scan line intersecting two vertices of a polygon

Displaying Text

Text mode is a **computer display** mode in which content is internally represented on a **computer** screen in terms of characters rather than individual pixels. Typically, the screen consists of a uniform rectang

...(horizontal/ vertical) and size of font.

...xy(). It will print message passed to it at some certain coordinate (x, y).

...nctions:

TextHeight()

textheight()

TextWidth()

textwidth()

SetUserCharSize()

setusercharsize(x_1, y_1, x_2, y_2); etc.

In com
examp
(vertice
determi
resoluti
created

In comp
implicit
colour s

More fu

need extra
on points of

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

sented on
the screen
one of the

; graphical
n different
re we will
n different

screen will

of font.

ere are just two intensities, full
ntensities in range like [1, 0.9,
owever, these total number of
suitable for digital systems.

between known values. For
ages from their corner points
nterpolation algorithm, which
when a video image in a low
olution, the missing lines are

nd the geometry of this space
colour spaces, and different
our spaces in this exercise, the


```
                    ...mputer Graphics ");
                delay(500);
        }
        delay(2000);
}
int main()
{
        printMsg();
        return 0;
}
```

Grey Images and Intensities

A black and white image is one in which at each pixel position the intensity i.e. 1 and no intensity i.e.0. If we allow pixels to have i 0.8...0.1, 0] such an image will be called a gray scale image. Ho steps in this range are in power of 2 (8, 16, 256 etc.) to make them s

Color interpolation

In computer graphics, it is the creation of new values that lie example, when objects are rasterized into two-dimensional ima (vertices), all the pixels between those points are filled in by an i determines their color and other attributes. Another example is resolution is upscale to display on a monitor with a higher res created by interpolation.

In computer graphics, colors are represented in a colour space, a implicitly answers this question. However, there exist a lot of colour spaces can yield different results. We will look at two colo

RGB and HSV colour spaces. The most commonly encountered colour space in computer graphics is the RGB colour space. Here, we specify colors as a mix of Red, Green, and Blue. This is a natural choice since this is how colors are formed on computer screens. In addition, it is closely related to the physiology of the human eye (which decomposes light into red, green and blue). However, this colour space is not particularly intuitive seen from a psychological point of view; humans do not describe colors in this way.

Exercise

1. Describe RGB color model and HSV color mode.
2. Discuss the problems of interpolated model
3. Find the CMY coordinate of a color at (0.2, 1, 0.5) in the RGB space
4. Compare HSV and HSL color model
5. What is a color model?
6. With 3 bits per pixel, we can accommodate 8 gray levels. If we use 8 bits per pixel then what is the value of gray levels?
7. What is the range that specifies the gray or grayscale levels?
8. Why Work in RGB and Convert to CMYK?
9. What is saturation?
10. How is color specified?
11. Should I use a color specification system for image data?
12. What weighting of red, green and blue corresponds to brightness?
13. Can blue be assigned fewer bits than red or green?
14. What is color temperature?
15. How can I characterize red, green and blue?
16. Is RGB always device-dependent?
17. How do I transform data from one set of RGB primaries to another?
18. Should I use RGB or XYZ for image synthesis?
19. What is subtractive color?
20. Why did my grade three teacher tell me that the primaries are red, yellow and blue?

❑❑❑

Lab Assignment in Java

Drawing Methods of the Graphics Class

S.No	Method	Description
1	clearRect()	Erase a rectangular area of the canvas.
2	copyAre()	Copies a rectangular area of the canvas to another area
3	drawArc()	Draws a hollow arc
4	drawLine()	Draws a straight line
5	drawOval()	Draws a hollow oval
6	drawPolygon()	Draws a hollow polygon
7	drawRect()	Draws a hollow rectangle
8	drawRoundRect()	Draws a hollow rectangle with rounded corners
9	drawString()	Display a text string
10	FillArc()	Draws a filled arc
11	fillOval()	Draws a filled Oval
12	fillPolygon()	Draws a filled Polygon
13	fillRect()	Draws a filled rectangle
14	fillRoundRect()	Draws a filled rectangle with rounded corners
15	getColor()	Retrieves the current drawing color
16	getFont()	Retrieves the currently used font
17	getFontMetrics()	Retrieves the information about the current font
18	setColor()	Sets the drawing color
19	setFont()	Sets the font

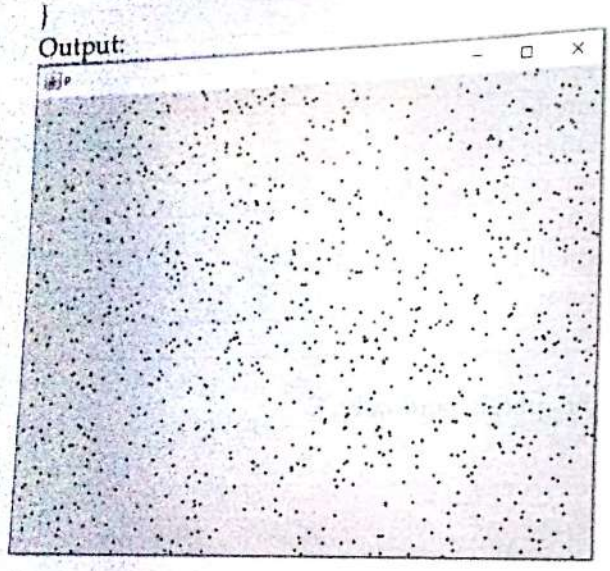
Program no 1: Write a java program to display various pixels randomly.

Source code:

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.util.Random;
import javax.swing.JFrame;
import javax.swing.JPanel;
public class point extends JPanel
{
    @Override
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        Graphics2D g2d = (Graphics2D) g;
        g2d.setColor(Color.black);
        for (int i = 0; i <= 1000; i++)
        {
            Dimension size = getSize();
            int w = size.width ;
            int h = size.height;
            Random r = new Random();
            int x = Math.abs(r.nextInt()) % w;
            int y = Math.abs(r.nextInt()) % h;
```

```
                g2d.drawOval(x, y, 4, 4);
            }
        }
        public static void main(String[] args)
        {
            point p = new point();
            JFrame frame = new JFrame("P");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.add(p);
            frame.setSize(850, 700);
            frame.setLocationRelativeTo(null);
            frame.setVisible(true);
        }
}
```

Output:



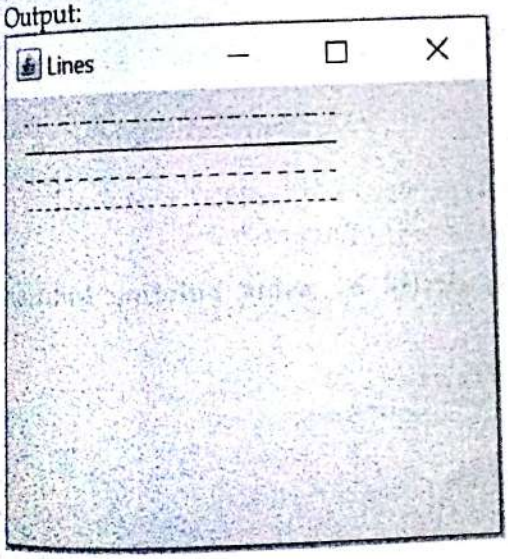
Program no 2: Write a java program to display lines in different style.

Source code:

```
package computer_graphics;
import java.awt.BasicStroke;
import java.awt.Graphics;
import java.awt.Graphics2D;
import javax.swing.JFrame;
import javax.swing.JPanel;
public class Line_Style extends JPanel
{
 @Override
 public void paintComponent(Graphics g)
 {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D) g;
  g2d.drawLine(20, 40, 250, 40);
  float[] dash1 = { 2f, 0f, 2f };
  BasicStroke bs1 = new BasicStroke(1, BasicStroke.CAP_BUTT,
  BasicStroke.JOIN_ROUND, 3.0f, dash1, 6f);
  g2d.setStroke(bs1);
```

```
g2d.drawLine(20, 80, 250, 80);
float[ ] dash3 = { 4f, 0f, 2f };
BasicStroke bs3 = new BasicStroke(1, BasicStroke.CAP_BUTT, BasicStroke.JOIN_ ROUND, 1.0f,
dash3, 2f);
g2d.setStroke(bs3);
g2d.drawLine(20, 60, 250, 60);
float[ ] dash4 = { 4f, 4f, 1f };
BasicStroke bs4 = new BasicStroke(1, BasicStroke.CAP_BUTT, BasicStroke.JOIN_ ROUND, 1.0f,
dash4, 2f);
g2d.setStroke(bs4);
g2d.drawLine(20, 20, 250, 20);
}

public static void main(String[] args)
{
Line_Style l = new Line_Style();
JFrame frame = new JFrame("Lines");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.add(l);
frame.setSize(380, 370);
frame.setLocationRelativeTo(null);
frame.setVisible(true);
}
}
```

Output:



Program no 3: Write a java program to display lines by using building function drawLine().

```
import java.awt.*;
import java.applet.*;
public class Lines extends Applet
{
        public void init()
        {
                setBackground (Color.black);
                setForeground(Color.green);
        }
```

```
        public void paint(Graphics g)
        {
                g.drawLine(0, 0, 100, 100);
                g.drawLine(0, 100, 100, 0);
                g.drawLine(40, 25, 250, 180);
                g.drawLine(75, 90, 400, 400);
                g.drawLine(20, 150, 400, 40); //line
                g.drawLine(5, 290, 80, 19); //line
                g.drawLine (5, 75, 5, 75); // point
                g.drawLine (50, 5, 50, 5); // point
        }
}
```

Program no 4: Write a java program to display rectangles by using building function drawRect().

```
import java.awt.*;
import java.applet.*;
public class Rectangles extends Applet
{
        public void init()
        {
                setBackground(Color.black);
                setForeground(Color.green);
        }
        public void paint(Graphics g)
        {
                g.drawRect(10, 10, 60, 50);
                g.fillRect(100, 10, 60, 50);
                g.drawRoundRect(190, 10, 60, 50, 15, 15);
                g.fillRoundRect(70, 90, 140, 100, 30, 40);
        }
}
```

Program no 5: Write a java program to display ellipse, circles by using building function drawOval().

```
import java.awt.*;
import java.applet.*;
public class Ovals extends Applet
{
        public void init()
        {
                setBackground(Color.black);
                setForeground(Color.green);
        }
        public void paint(Graphics g)
        {
                g.drawOval(10, 10, 50, 50);
                g.fillOval(100, 10, 75, 50);
                g.drawOval(190, 10, 90, 30);
                g.fillOval(70, 90, 140, 100);
        }
}
```

Program no 6: Write a java program to display Arc by using building function drawArc().

```
import java.awt.*;
import java.applet.*;
public class Arcs extends Applet
{
        public void init()
        {
                setBackground(Color.black);
                setForeground(Color.green);
        }

        public void paint(Graphics g)
        {
                g.drawArc(10, 40, 70, 70, 0, 75);
                g.fillArc(100, 40, 70, 70, 0, 75) ;
                g.drawArc(10, 100, 70, 80, 0, 175);
                g.fillArc(100, 100, 70, 90, 0, 270);
                g.drawArc(200, 80, 80, 80, 0, 180);
        }
}
```

Program no 7: Write a java program to display Polygon by using building function drawPolygon().

```
import java.awt.*;
import java.applet.*;
public class HourGlass extends Applet
{
        public void init()
        {
                setBackground(Color.black);
                setForeground(Color.green);
        }
        public void paint(Graphics g)
        {
                int xpoints[ ] = {30, 200, 30, 200, 30};
                int ypoints[ ] = {30, 30, 200, 200, 30};
                int num = 5;
                g.drawPolygon(xpoints, ypoints, num);
        }
}
```

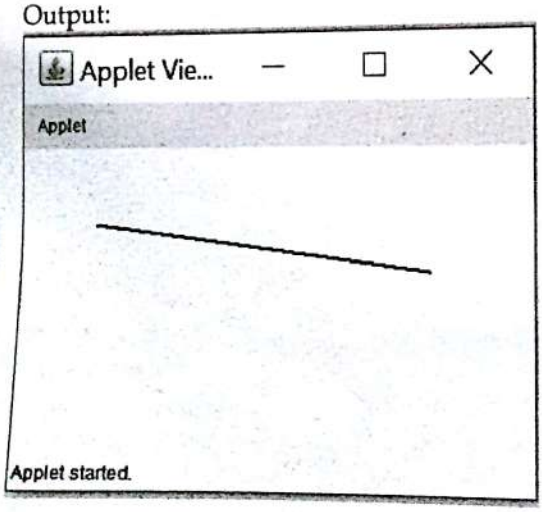
Program no 8: Write a java program to display line by using DDA Line drawing algorithm.

Source code:

```
package computer_graphics;
import java.awt.*;
import java.applet.*;
public class DDA_Line extends Applet
{
 @Override
 public void paint(Graphics g)
```

```
    {
      double dx,dy,steps,x,y,k;
      double xc,yc;
      double x1=50,y1=50,x2=280,y2=80;
      dx=x2-x1;
      dy=y2-y1;
      if(Math.abs(dx)>Math.abs(dy))
        steps=Math.abs(dx);
      else
        steps=Math.abs(dy);
      xc=(dx/steps);
      yc=(dy/steps);
      x=x1;
      y=y1;
      g.fillOval(200,500,2,2);
      for(k=1;k<=steps;k++)
      {
        x=x+xc;
        y=y+yc;
        g.fillOval((int)x,(int)y,2,2);
            }
      }
  }
```

Output:



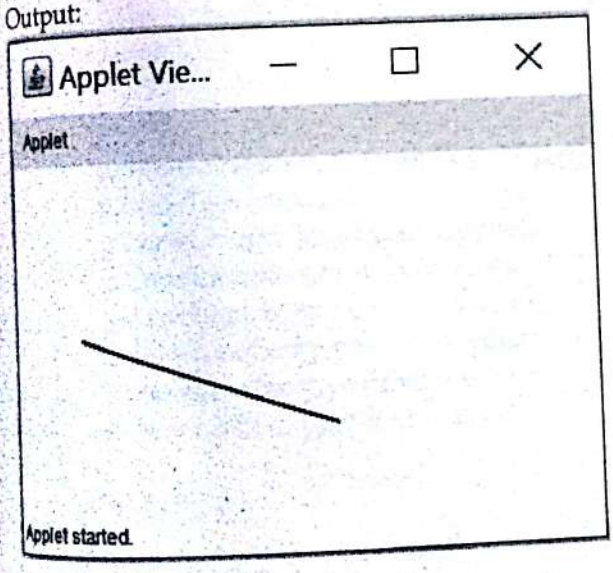
Program no 9: Write a java program to display line by using Bresenham Line drawing algorithm.

Source code:

```
package computer_graphics;
import java.applet.*;
import java.awt.*;
public class Bre1 extends Applet
{
    public void paint(Graphics g)
  {
    int x,y,k;
    double dx,dy,p;
    int x1=50,y1=100,x2=200,y2=150;
```

```
dx=Math.abs(x2-x1);
dy=Math.abs(y2-y1);
x=x1;
y=y1;
p=2*dy-dx;
g.fillOval(200,300,2,2);
for(k=0;k<dx;k++)
{
  if(p<0)
  {
     g.fillOval(x++, y, 2, 2);
     p=p+(2*dy);
  }
  else
  {
     g.fillOval(x++,y++,2,2);
     p=p+(2*(dy-dx));
  }
  }
 }
}
```

Output:

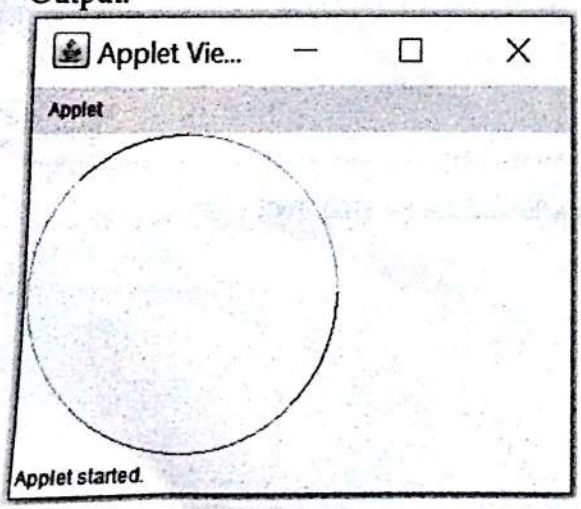


Program no 10: Circle drawing program in Java of radius 90 and center (100, 100)

```
import java.awt.*;
import java.applet.*;
public class Circle extends Applet
{
    public void paint(Graphics g)
    {
        double h, k, p, r, x, y;
        h=100;
        k=100;
        r=90;
```

```
        x=0;
        y=r;
        p=1-r;
        while(x<y)
        {
            g.setColor(Color.red);
            g.fillOval((int)(h+x),(int)(k+y), 1, 2);
            g.fillOval((int)(h+x),(int)(k-y), 1, 2);
            g.fillOval((int)(h-x),(int)(k+y), 1, 2);
            g.fillOval((int)(h-x),(int)(k-y), 1, 2);
            g.fillOval((int)(h+y),(int)(k+x), 1, 2);
            g.fillOval((int)(h+y),(int)(k-x), 1, 2);
            g.fillOval((int)(h-y),(int)(k+x), 1, 2);
            g.fillOval((int)(h-y),(int)(k-x), 1, 2);
            if(p<0)
            {
                p=p+2*x+1;
                x++;
            }
            else
            {
                p=p+2*x-2*y+1;
                y--;
                x++;
            }
        }
    }
}
```

Output:

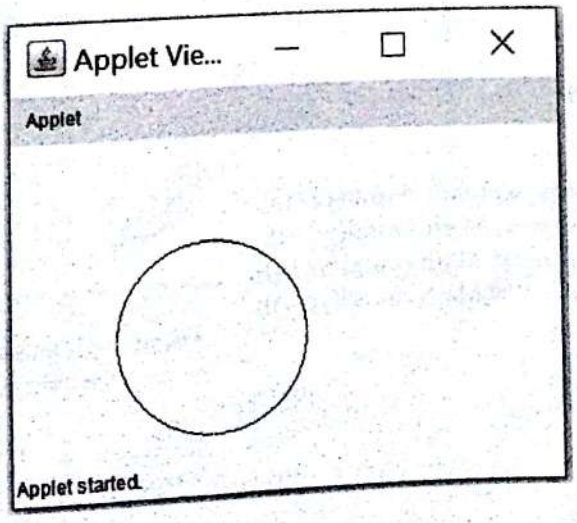


Program no 11: Write a java program to display circle by using Bresenham circle drawing algorithm.

Source code:

```
package computer_graphics;
import java.awt.*;
import java.applet.*;
public class Brescir1 extends Applet
{
    @Override
    public void paint(Graphics g)
    {
      int xc,yc,r;
      xc=125;
      yc=125;
      r=60;
      int x=0;
      int y=r;
      int p=3-2*r;
      do
      {
        if(p<0)
         p=p+4*x+6;
        else
        {
         p=p+4*(x-y)+10;
          y=y-1;
        }
        x=x+1;
        g.drawLine(xc+x,yc+y,xc+x,yc+y);
        g.drawLine(x+xc,yc-y,xc+x,yc-y);
        g.drawLine(xc-x,yc+y,xc-x,yc+y);
        g.drawLine(xc-x,yc-y,xc-x,yc-y);
        g.drawLine(xc+y,yc+x,xc+y,yc+x);
        g.drawLine(xc+y,yc-x,xc+y,yc-x);
        g.drawLine(xc-y,yc+x,xc-y,yc+x);
        g.drawLine(xc-y,yc-x,xc-y,yc-x);
      }
      while(x<y);
    }
}
```

Output:



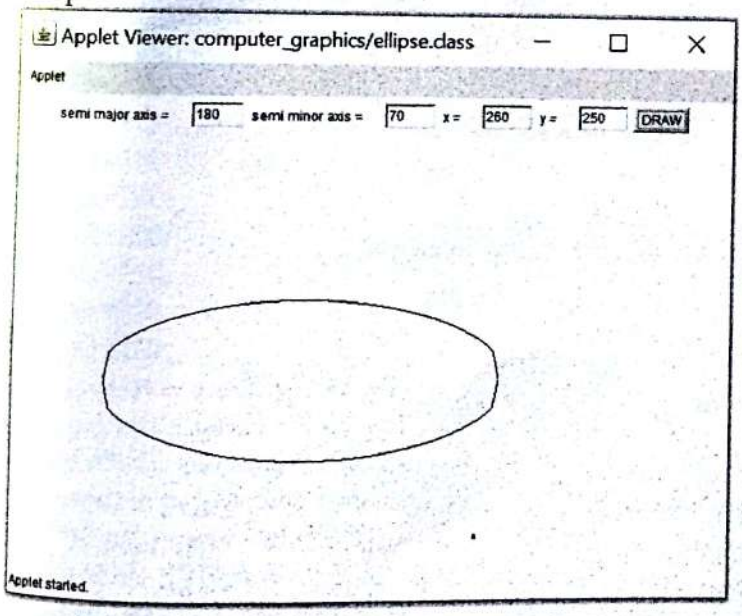
Program no 12: Midpoint Ellipse

Source code:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class ellipse extends Applet implements ActionListener
{
  float x,y,xc,yc,rx,ry,p1,p2;
  Label l1 = new Label("semi major axis = ");
  Label l2 = new Label("semi minor axis = ");
  Label l3 = new Label("x = ");
  Label l4 = new Label("y = ");
  TextField t1 = new TextField(3);
  TextField t2 = new TextField(3);
  TextField t3 = new TextField(3);
  TextField t4 = new TextField(3);
  Button bw = new Button("DRAW");
  public void init()
  {
    add(l1);
    add(t1);
    add(l2);
    add(t2);
    add(l3);
    add(t3);
    add(l4);
    add(t4);
    add(bw);
    bw.addActionListener(this);
  }
  public void  paint(Graphics g)
  {
    rx=Integer.parseInt(t1.getText());
    ry=Integer.parseInt(t2.getText());
    xc=Integer.parseInt(t3.getText());
    yc=Integer.parseInt(t4.getText());
    x=0;
    y=ry;
    p1=(ry*ry)-(rx*rx*y)+(rx*rx/4);
    do
    {
      g.drawString(".",Math.round(xc-x),Math.round(yc+y));
      g.drawString(".",Math.round(xc-x),Math.round(yc-y));
      g.drawString(".",Math.round(xc+x),Math.round(yc+y));
      g.drawString(".",Math.round(xc+x),Math.round(yc-y));
      x++;
      if(p1<0)
        p1=p1+(2*ry*ry*x)+(ry*ry);
      else
      {
```

```
        y-=1;
        p1=p1+(2*ry*ry*x)-(2*rx*rx*y)+(ry*ry);
      }
    }
    while((2*ry*ry*x)<(2*rx*rx*y));
    p2= (ry*ry*(x+1/2)*(x+1/2))+(rx*rx*(y-1)*(y-1))-(ry*ry*rx*rx);
    while(y>0)
    {
      g.drawString(".",Math.round(xc-x),Math.round(yc+y));
      g.drawString(".",Math.round(xc-x),Math.round(yc-y));
      g.drawString(".",Math.round(xc+x),Math.round(yc+y));
      g.drawString(".",Math.round(xc+x),Math.round(yc-y));
      y--;
      if(p2>0)
        p2 = p2-(2*rx*rx*y)+(ry*ry);
      else
      {
        x+=1;
        p2 = p2=(2*ry*ry*x)+(2*rx*rx*y)+(ry*ry);
      }
    }
  }
  public void actionPerformed(ActionEvent aw)
  {
    repaint();
  }
}
```

Output:

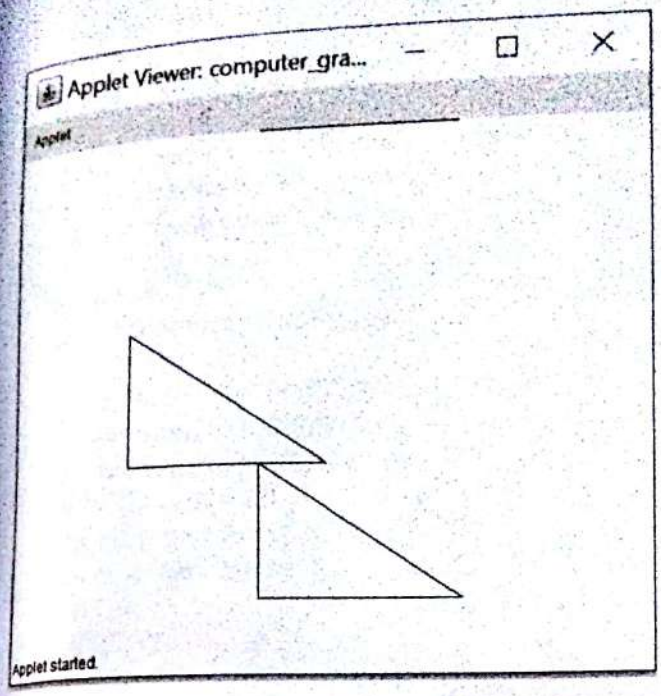


Program no 13: Translation in 2D

```
package computer_graphics;
import java.awt.*;
import java.applet.*;
public class Trans extends Applet
```

```
{
  @Override
  public void paint(Graphics g)
  {
    g.drawLine(100,150,100,250);
    g.drawLine(100,150,250,250);
    g.drawLine(100,250,250,250);
    int a[][]=new int[3][3];
    int b[][]=new int[3][3];
    int c[][]=new int[3][3];
    int i,j,k;
    a[0][0]=1;
    a[0][1]=0;
    a[0][2]=100;
    a[1][0]=0;
    a[1][1]=1;
    a[1][2]=100;
    a[2][0]=0;
    a[2][1]=0;
    a[2][2]=1;

    b[0][0]=100;
    b[0][1]=100;
    b[0][2]=250;
    b[1][0]=150;
    b[1][1]=250;
    b[1][2]=250;
    b[2][0]=1;
    b[2][1]=1;
    b[2][2]=1;
    for(i=0;i<3;i++)
    {
      for(j=0;j<3;j++)
      {
        c[i][j]=0;
        for(k=0;k<3;k++)
        {
          c[i][j]=(c[i][j]+(a[i][k]*b[k][j]));
        }
        System.out.println();
      }
      g.drawLine(c[0][0],c[1][0],c[0][1],c[1][1]);
      g.drawLine(c[0][1],c[1][1],c[0][2],c[1][2]);
      g.drawLine(c[0][0],c[1][0], c[0][2],c[1][2]);
    }
  }
}
```

Output:

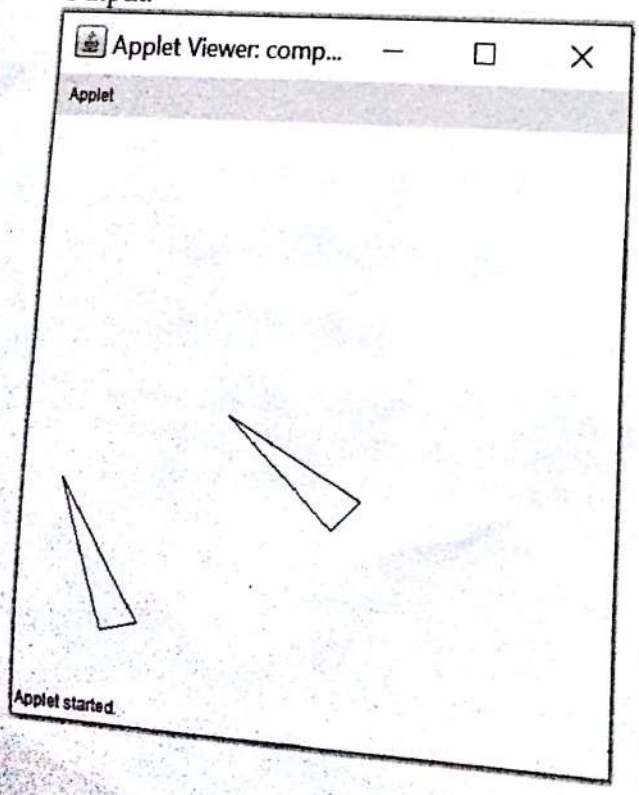


Program no 14: Rotation in 2D

```
package computer_graphics;
import java.awt.*;
import java.applet.*;
public class Rotation extends Applet
{
  @Override
  public void paint(Graphics g)
  {
    int i,j,k;
    double r=30*3.14/180;
    int x1,y1,x2,y2,x3,y3,x11,y11,x22,y22,x33,y33;
    x1=150;
    y1=200;
    x2=250;
    y2=250;
    x3=230;
    y3=270;
    g.drawLine(x1, y1, x2, y2);
    g.drawLine(x2, y2, x3, y3);
    g.drawLine(x1, y1, x3, y3);
    double p1[][]=new double[3][3];
    int p[][]=new int[3][3];
    double t[][]=new double[3][3];
    t[0][0]=t[1][1]=Math.cos(r);
    t[0][1]=-(Math.sin(r));
    t[1][0]=Math.sin(r);
    t[2][2]=1;
    t[0][2]=t[1][2]=t[2][0]=t[2][1]=0;
    p[0][0]=x1;
    p[0][1]=x2;
```

```
        p[0][2]=x3;
        p[1][0]=y1;
        p[1][1]=y2;
        p[1][2]=y3;
        p[2][0]=p[2][1]=p[2][2]=1;
        for(i=0;i<3;i++)
        {
          for(j=0;j<3;j++)
          {
            p1[i][j]=0;
            for(k=0;k<3;k++)
            {
              p1[i][j]=p1[i][j]+t[i][k]*p[k][j];
            }
          }
          System.out.println();
        }
        x11=(int)p1[0][0];
        x22=(int)p1[0][1];
        x33=(int)p1[0][2];
        y11=(int)p1[1][0];
        y22=(int)p1[1][1];
        y33=(int)p1[1][2];
        g.drawLine(x11, y11, x22, y22);
        g.drawLine(x22, y22, x33, y33);
        g.drawLine(x11,y11, x33, y33);
    }
}
```

Output:

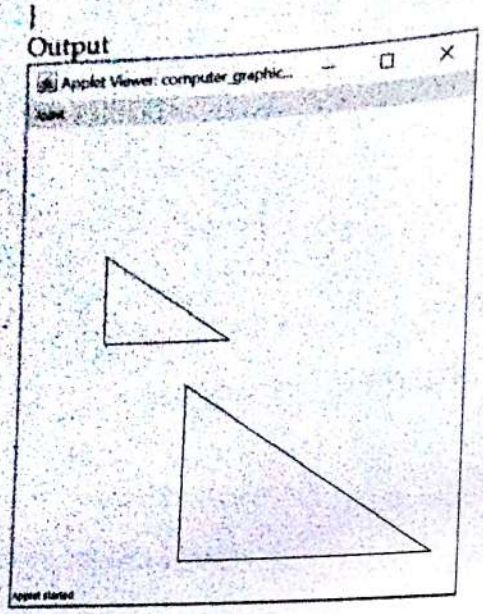


Program no 15: Scaling in 2D

```
package computer_graphics;
import java.awt.*;
import java.applet.*;
public class Scale extends Applet
{
 @Override
 public void paint(Graphics g)
 {
   g.drawLine(100,150,100,250);
   g.drawLine(100,150,250,250);
   g.drawLine(100,250,250,250);
   int a[][]=new int[3][3];
   int b[][]=new int[3][3];
   int c[][]=new int[3][3];
   int i,j,k;
   a[0][0]=2;
   a[0][1]=0;
   a[0][2]=0;
   a[1][0]=0;
   a[1][1]=2;
   a[1][2]=0;
   a[2][0]=0;
   a[2][1]=0;
   a[2][2]=1;

   b[0][0]=100;
   b[0][1]=100;
   b[0][2]=250;
   b[1][0]=150;
   b[1][1]=250;
   b[1][2]=250;
   b[2][0]=1;
   b[2][1]=1;
   b[2][2]=1;
   for(i=0;i<3;i++)
   for(j=0;j<3;j++)
   {
     c[i][j]=0;
     for(k=0;k<3;k++)
     {
       c[i][j]=(c[i][j]+(a[i][k]*b[k][j]));
     }
     System.out.println();
   }
   g.drawLine(c[0][0],c[1][0],c[0][1],c[1][1]);
   g.drawLine(c[0][1],c[1][1],c[0][2],c[1][2]);
   g.drawLine(c[0][0],c[1][0], c[0][2],c[1][2]);
}
```

```
}
```

Output



Program no 16: Cohen Sutherland Line Clipping Algorithm

Code:

```
importjava.applet.*;
importjava.awt.*;
importjava.util.*;
public class CohenSutherland extends Applet
{
        intxmax=90,ymax=80,xmin=40,ymin=40;
        publicint[ ] set(int x, int y)
        {
                int a[]=new int[4];
                if(x<xmin)
                        a[3]=1;
                else
                        a[3]=0;
                if(x>xmax)
                        a[2]=1;
                else
                        a[2]=0;
                if(y<ymin)
                        a[0]=1;
                else
                        a[0]=0;
                if(y>ymax)
                        a[1]=1;
                else
                        a[1]=0;
                return a;
        }
        boolean check(int a[])
        {
```

```
        for(int i=0;i<a.length;i++)
                if(a[i]==1)
                        return false;
        return true;
}
int[ ] produceXY(inti,int x1,int y1,float m)
{
        int a[]=new int [2];
        float x=0,y=0;
        switch(i)
        {
                case 0:
                        x=xmin;
                        y=y1+m*(x-x1);
                        break;

                case 1:
                        x=xmax;
                        y=y1+m*(x-x1);
                        break;
                case 3:
                        y=ymin;
                        x=x1+(y-y1)/m;
                        break;
                case 2:
                        y=ymax;
                        x=x1+(y-y1)/m;
                        break;
        }
        a[0]=(int)x;
        a[1]=(int)y;
        return a;
}
booleandoAnd(int a[],int b[])
{
        for(int i=0;i<a.length;i++)
        {
                int k=a[i]&b[i];
                if(k==1)
                        return false;
        }
        return true;
}
public void paint(Graphics g)
{
        g.drawRect(xmin,ymin,xmax-xmin,ymax-ymin);
        g.drawRect(xmin+100,ymin,xmax-xmin,ymax-ymin);
        int a[][]=new int[2][4];
        int b[][]=new int[2][4];
        int c[]=new int[2];
```

```
int c1=20;
int x1=45,y1=45,x2=20,y2=90;
float m=(y2-y1)*1.0f/(x2-x1);
g.drawString(m+" ",100,100);
g.drawLine(x1, y1, x2, y2);
a[0]=set(x1,y1);
a[1]=set(x2,y2);
g.drawString(Arrays.toString(a[0]),300,300);
g.drawString(Arrays.toString(a[1]),400,400);
if(check(a[0])&&check(a[1]))
{
        g.drawLine(x1, y1, x2, y2);
}
else
{
        if(doAnd())
        {
                for(int i=a[0].length-1;i>=0;i--)
                {
                        if(a[0][i]==1)
                        {
                                c=produceXY(a[0].length-1-i,x1,y1,m);
                                b[0]=set(c[0],c[1]);
                        g.drawString("first "+Arrays.toString(b[0]),200,300+c1);
                                if(check(b[0]))
                                {
                                        x1=c[0];
                                        y1=c[1];
                                        break;
                                }
                                c1+=20;
                        }
                }
                for(int i=a[0].length-1;i>=0;i--)
                {
                        if(a[1][i]==1)
                        {
                                c=produceXY(a[0].length-1-i,x1,y1,m);
                                b[1]=set(c[0],c[1]);
                                        g.drawString("second
                                        "+Arrays.toString(b[1]),200,300+c1);
                                if(check(b[1]))
                                {
                                        x2=c[0];
                                        y2=c[1];
                                        break;
                                }
                        }
                }
                g.drawLine(x1+100, y1, x2+100, y2);
```

```
            }
        }
    }
}
```

Program no 17: Bezier Curve

Code:

```
import java.awt.*;
import java.applet.*;
import java.awt.geom.GeneralPath;
import java.awt.image.BufferedImage;
public class KBezier extends Applet
{
    Point[ ] controlPoints,curvePoints;
    public void init()
    {
        controlPoints=new Point[4];
        curvePoints=new Point[25];
        controlPoints[0]=new Point(20,260);
        controlPoints[1]=new Point(50,10);
        controlPoints[2]=new Point(250,50);
        controlPoints[3]=new Point(450,290);
        for(int i=0;i<curvePoints.length;i++)
            curvePoints[i]=new Point(0,0);
    }
    public void SubDivide(Point p1,Point p2,double t)
    {
        if(p1.x>p2.x)
            p1.x-=Math.abs(p1.x-p2.x)*t;
        else
            p1.x+=Math.abs(p1.x-p2.x)*t;
        if (p1.y>p2.y)
            p1.y-=Math.abs(p1.y-p2.y)*t;
        else
            p1.y+=Math.abs(p1.y-p2.y)*t;
    }
    public void Compute()
    {
        Point[ ] tmp=new Point[controlPoints.length];
        for (int i=0; i<tmp.length; i++)
            tmp[i] = new Point(0,0);
        for (int i=0; i<curvePoints.length; i++)
        {
            double t = ((double) i)/ (curvePoints.length-1);
            for (int j=0; j<controlPoints.length; j++)
                tmp[j]=new Point(controlPoints[j].x, controlPoints[j].y);
            int Depth = tmp.length;
            while (Depth>1)
            {
                for (int j=0; j<Depth-1; j++)
```

```
                    SubDivide(tmp[j], tmp[j+1], t);
                Depth--;
            }
            curvePoints[i]=new Point(tmp[0].x, tmp[0].y);
        }
    }
    public void Draw(Graphics2D g2d)
    {
        g2d.setColor(Color.BLACK);
        for (int i=0; i<controlPoints.length-1; i++)
        g2d.drawLine((int) controlPoints[i].x,(int) controlPoints[i].y,
        (int) controlPoints[i+1].x,(int) controlPoints[i+1].y);
        GeneralPath path = new GeneralPath();
        g2d.setColor(Color.RED);
        path.moveTo(curvePoints[0].x, curvePoints[0].y);
        for (int i=1; i<curvePoints.length; i++)
            path.lineTo(curvePoints[i].x, curvePoints[i].y);
        g2d.draw(path);
    }
    public void paint(Graphics g)
    {
        BufferedImage image = new BufferedImage(getWidth(), getHeight(),
        BufferedImage.TYPE_INT_RGB);
        Graphics2D g2d = image.createGraphics();
        g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON);
        g2d.setColor(Color.WHITE);
        g2d.fillRect(0, 0, getWidth(), getHeight());
        Compute();
        Draw(g2d);
        g.drawImage(image, 0, 0, this);
    }
}
```

Program no 18: Koch Curve

Code:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class KKoch extends Applet implements MouseListener
{
    int iteration;
    int size = 20;
    public void init()
    {
        setBackground(Color.white);
        addMouseListener(this) ;
        iteration=1;
    }
```

```
public void mouseMoved(MouseEvent e){}
public void mouseDragged(MouseEvent e){}
public void mouseReleased(MouseEvent e){}
public void mouseClicked(MouseEvent e){}
public void mouseEntered(MouseEvent e){}
public void mouseExited(MouseEvent e){}

void koch(double x1, double y1, double x2, double y2,int iteration, Graphics g)
{
    if (iteration<=1)
    {
            g.drawLine ((int )x1,(int )y1,(int )x2,(int )y2);
    }
    else
    {
        double x4=x1*2/3 + x2*1/3;
        double y4=y1*2/3 + y2*1/3;
        double x5=x1*1/3 + x2*2/3;
        double y5=y1*1/3 + y2*2/3;
        double x6=(int) (0.5 * (x1+x2) + Math.sqrt(3) * (y1-y2)/6);
        double y6=(int) (0.5 * (y1+y2) + Math.sqrt(3) * (x2-x1)/6);
        koch(x1,y1,x4,y4,iteration-1,g);
        koch(x4,y4,x6,y6,iteration-1,g);
        koch(x6,y6,x5,y5,iteration-1,g);
        koch(x5,y5,x2,y2,iteration-1,g);
    }
}
public void paint(Graphics g)
{
    g.setColor(Color .black);
    Dimension w = getSize();
    koch(size*3, w.height - size*7, w.width - size*3, w.height - size*7, iteration, g);
    koch(w.width/2, size*2, size*3, w.height - size*7, iteration, g);
    koch(w.width - size*3, w.height - size*7, w.width/2, size*2, iteration, g);
}
public void mousePressed(MouseEvent e)
{
    iteration++;
    repaint();
}
}
```

Bibliography

1. Hearn, Donald D., Pauline Baker, and Warren Carithers. *Computer Graphics with Open GL: Pearson New International Edition*. Pearson Higher Ed, 2013.
2. Foley, James D., Andries Van Dam, Steven K. Feiner, John F. Hughes, and Richard L. Phillips. *Introduction to computer graphics*. Vol. 55. Reading: Addison-Wesley, 1994.
3. Marschner, Steve, and Peter Shirley. *Fundamentals of computer graphics*. CRC Press, 2015.
4. Vince, John. *Mathematics for computer graphics*. Springer, 2017.
5. Salomon, David. *The computer graphics manual*. Springer Science & Business Media, 2011.
6. McReynolds, Tom, and David Blythe. *Advanced graphics programming using OpenGL*. Elsevier, 2005.
7. Whitrow, Robert. *OpenGL graphics through applications*. Springer Science & Business Media, 2008.
8. Salomon, David. *Curves and surfaces for computer graphics*. Springer Science & Business Media, 2007.
9. Plemenos, Dimitri. *Intelligent Computer Graphics 2011*. Edited by GeorgiosMiaoulis. Vol. 441. Springer, 2012.
10. Klawonn, Frank. *Introduction to computer graphics: using Java 2D and 3D*. Springer Science & Business Media, 2012.
11. Mortenson, Michael E. *Mathematics for computer graphics applications*. Industrial Press Inc., 1999.
12. Strothotte, Thomas, and Stefan Schlechtweg. *Non-photorealistic computer graphics: modeling, rendering, and animation*. Morgan Kaufmann, 2002.
13. Ammeraal, Leendert, Kang Zhang, and Kang Zhang. *Computer graphics for Java programmers*. Chichester: Wiley, 1998.

SOLUTION

1. Define Computer graphics.

Computer graphics remains one of the most existing and rapidly growing computer fields. Computer graphics may be defined as a pictorial representation or graphical representation of objects in a computer.

2. What is meant by scan code?

When a key is pressed on the keyboard, the keyboard controller places a code carry to the key pressed into a part of the memory called as the keyboard buffer. This code is called as the scan code.

3. What is meant by refreshing of the screen?

Some method is needed for maintaining the picture on the screen. Refreshing of screen is done by keeping the phosphorus glowing to redraw the picture repeatedly i.e. by quickly directing the electronic beam back to the same points.

4. Define Random scan/Raster scan displays?

Random scan is a method in which the display is made by the electronic beam which is directed only to the points or part of the screen where the picture is to be drawn. The Raster scan system is a scanning technique in which the electrons sweep from top to bottom and from left to right. The intensity is turned on or off to light and unlight the pixel.

5. List out the merits and demerits of Penetration techniques?

The merits and demerits of the Penetration techniques are as follows

- It is an inexpensive technique
- It has only four colors
- The quality of the picture is not good when it is compared to other techniques
- It can display color scans in monitors
- Poor limitation etc.

6. List out the merits and demerits of DVST?

The merits and demerits of direct view storage tubes [DVST] are as follows

- It has a flat screen
- Refreshing of screen is not required
- Selective or part erasing of screen is not possible
- It has poor contrast
- Performance is inferior to the refresh CRT.

7. What do you mean by emissive and non-emissive displays?

The emissive display converts electrical energy into light energy. The plasma panels, thin film electro-luminescent displays are the examples. The Non emissive are optical effects to convert the sunlight or light from any other source to graphic form. Liquid crystal display is an example.

8. List out the merits and demerits of Plasma panel display?

Merits

- Refreshing is not required
- Produce a very steady image free of Flicker
- Less bulky than a CRT. Demerits

- Poor resolution of up to 60 d.p.i
- It requires complex addressing and wiring
- It is costlier than CRT.

9. What is persistence?

The time it takes the emitted light from the screen to decay one tenth of its original intensity is called as persistence.

10. What is resolution?

The maximum number of points that can be displayed without an overlap on a CRT is called as resolution.

11. What is Aspect ratio?

The ratio of vertical points to the horizontal points necessary to produce length of lines in both directions of the screen is called the Aspect ratio. Usually the aspect ratio is ¾.

12. What is meant by Addressability?

The Addressability is the number of individual dots per inch (d.p.i) that can be created. If the address of the current dot is (x, y) then the next dot will be (x+y), (x+y+1) etc.

13. What is a dot size?

Dot size may be defined as the diameter of a single dot on the devices output. Dot size is also called as the Spot size.

14. What is inter dot distance?

Interdot distance is the reciprocal of addressability. If the addressability is large, the interdot distance will be less. The interdot distance should be less to get smooth shapes.

15. What is the difference between impact and non-impact printers?

Impact printer press formed character faces against an inked ribbon on to the paper. A line printer and dot-matrix printer are examples. Non-impact printer and plotters use Laser techniques, inkjet sprays, Xerographic process, electrostatic methods and electro thermal methods to get images onto the papers. Examples are: Inkjet/Laser printers.

16. What are the features of Inkjet printers?

- They can print 2 to 4 pages/minutes.
- Resolution is about 360d.p.i. Therefore better print quality is achieved.
- The operating cost is very low. The only part that requires replacement is ink cartridge.
- 4 colors cyane, yellow, majenta, black are available.

17. What are the advantages of laser printer?

- High speed, precision and economy.
- Cheap to maintain.
- Quality printers.
- Lasts for longer time.
- Toner power is very cheap.

18. What are the advantages of electrostatic plotters?

- They are faster than pen plotters and very high quality printers.
- Recent electrostatic plotters include a scan-conversion capability.
- Color electrostatic plotters are available. They make multiple passes over the paper to plot color pictures.

19. Define pixel?

Pixel is shortened forms of picture element. Each screen point is referred to as pixel or pel.

20. What is frame buffer?

Picture definition is stored in a memory area called frame buffer or refresh buffer.

21. What is bitmap and what is pixmap?

The frame buffer used in the black and white system is known as bitmap which take one bit per pixel. For systems with multiple bits per pixel, the frame buffer is often referred to as a pixmap.

22. What is a Vector display or stroke writing or calligraphic display?

Random scan monitors draw a picture one line at a time and for this reason are also referred as vector displays.

23. Where the video controller is used?

A special purpose processor, which is used to control the operation of the display device, is known as video controller or display controller.

24. What do you mean by scan conversion?

A major task of the display processor is digitizing a picture definition given in an application program into a set of pixel-intensity values for storage in the frame buffer. This digitization process is called scan conversion.

25. What is run length encoding?

Run length encoding is a compression technique used to store the intensity values in the frame buffer, which stores each scan line as a set of integer pairs. One number each pair indicates an intensity value, and second number specifies the number of adjacent pixels on the scan line that are to have that intensity value

UNIT – II

1. What is an output primitive?

Graphics programming packages provide function to describe a scene in terms of these basic geometric structures, referred to as output primitives.

2. What do you mean by 'jaggies'?

Line with stair step appearance is known as jaggies.

3. What is point in the computer graphics system?

The point is a most basic graphical element & is completely defined by a pair of user coordinates (x , y).

4. Write short notes on lines?

A line is of infinite extent can be defined by an angle of slope θ and one point on the line P=P(x,y). This can also be defined as Y=mx+C where C is the Y- intercept.

5. Define Circle?

Circle is defined by its center xc, yc and its radius in user coordinate units. The equation of the circle is (x-xc) + (y-yc) = r2.

6. Define Ellipse?

An ellipse can use the same parameters xc, yc, r as a circle, in addition to the eccentricity e. the eqn of an ellipse is:

(x-xc)2/a2 + (y-yc)2/b2 = 1

7. Define polygon?

A polygon is any closed continues sequence of line segments ie, a polyline whose last node point is same as that of its first node point. The line segments form the sides of the polygon and their intersecting points from the vertices of the polygon.

8. Distinguish between convex and concave polygons?

If the line joining any two points in the polygon lies completely inside the polygon then, they are known as convex polygons. If the line joining any two points in the polygon lies outside the polygon then, they are known as concave polygons.

9. What is seed fill?

One way to fill a polygon is to start from a given point (seed) known to be inside the polygon and highlight outward from this point i.e. neighboring pixels until encounter the boundary pixels, this approach is called seed fill.

10. What is scan line algorithm?

One way to fill the polygon is to apply the inside test. i.e to check whether the pixel is inside the polygon or outside the polygon and then highlight the pixel which lie inside the polygon. This approach is known as scan-line algorithm.

11. Define coherence properties?

A coherence property of a scene is a part of a scene by which relate one part of the scene with the other parts of the scene.

12. What is an active edge list in the scan line algorithm?

The active edge list for a scan line contains all edges crossed by that scan line.

13. What is a winding number?

Winding number method is used to check whether a given point is inside or outside the polygon. In this method give a direction number to all the edges which cross the scan line. If the edge starts below the line and ends above scan line give direction as -1. otherwise1. Fr polygons or two dimensional objects, the point is said to be inside when the value of winding number is nonzero.

14. What is cell array?

The cell array is a primitive that allows users to display an arbitrary shape defined as a two dimensional grid pattern.

15. What is type face?

Letters, numbers and other characters can be displayed in a variety of sizes and styles. The overall design style for a set of characters is called a type face.

16. What do you mean by font?

The term font referred to a set of cast metal character forms in a particular size and format, such as 10-point courier italic.

17. What is a bitmapped font?

A simple method for representing the character shapes in a particular typeface is to use rectangular grid patterns. The set of characters are then referred to as a bitmap font.

18. What is an outline font?

A flexible scheme is to describe character shapes using straight-line and curve sections. In this case, the set of character is called an outline font.

19. What is an attribute parameter?

Any parameter that affects the way a primitive is to be displayed is referred to as an attribute parameter.

20. What are the various attributes of a line?

The line type, width and color are the attributes of the line. The line type includes solid line, dashed lines, and dotted lines.

21. What is pixel mask?

Pixel mask is a string containing the digits 1 and 0 to indicate which positions to plot along the line path. The mask 1111000 could be used to display a dashed line with a dash length of 4 and inter dot spacing of three.

22. What is a Line cap?

Line caps can be used to adjust the shape of the line ends to give a better appearance. There are three types of line caps. Butt cap which has a square end, round cap which has a semi circle end, projecting square cap which has one half of the line width beyond the specified end points.

23. List out the methods used for smoothly joining two line segments?

Mitter join- by extending the outer boundaries of each of the two lines until they meet.

Round join – by capping the connection between the two segments with a circular boundary whose diameter is equal to the line width.

Bevel join – by displaying the line segments with butt caps and filling in the triangular gap where the segment meet.

24. What is Color Look up table?

In color displays, 24 bits per pixel are commonly used, where 8 bits represent 256 levels for each color. It is necessary to read 24- bit for each pixel from frame buffer. This is very time consuming. To avoid this video controller uses look up table to store many entries to pixel values in RGB format. This look up table is commonly known as colour table.

25. What is tiling patterns?

The process of filling an area with rectangular pattern is called tiling and rectangular fill patterns are sometimes referred to as tiling patterns.

26. What is soft fill?

Soft fill is a filling method in which fill color is combined with the background colors.

27. What is kerned character?

The characters which extend beyond the character body limits are known as kerned character. Example f and j.

28. What is character up vector?

The orientation for a displayed character string is set according to the direction of the character up vector.

29. Define bundled attributes?

Individual attribute commands provide a simple and direct method for specifying attributes when a single output device is used. When several kinds of output device are available at a graphics installation, it is convenient to set up a table for each output device that lists set of attribute values that are to be used on that device to display each primitive type. Attribute specified in this manner is known as bundled attribute.

30. What is aliasing?

In the line drawing algorithms, all rasterzed locations do not match with the true line and have to represent a straight line. This problem is severe in low resolution screens. In such screens line appears like a stair-step. This effect is known as aliasing.

31. What is antialiasing?

The process of adjusting intensities of the pixels along the line to minimize the effect of aliasing is called antialiasing.

32. What is pixel phasing?

Pixel phasing is an antialiasing technique, stair steps are smoothed out by moving the electron beam to more nearly approximate positions specified by the object geometry.

UNIT - III

1. What is Transformation?

Transformation is the process of introducing changes in the shape size and orientation of the object using scaling rotation reflection shearing & translation etc.

2. Write short notes on active and passive transformations?

In the active transformation the points x and x| represent different coordinates of the same coordinate system. Here all the points are acted upon by the same transformation and hence the shape of the object is not distorted. In a passive transformation the points x and x| represent same points in the space but in a different coordinate system. Here the change in the coordinates is merely due to the change in the type of the user coordinate system.

3. What is translation?

Translation is the process of changing the position of an object in a straight-line path from one coordinate location to another. Every point (x , y) in the object must undergo a displacement to (x|,y|). The transformation is:

x| = x + tx ; y| = y+ty

4. What is rotation?

A 2-D rotation is done by repositioning the coordinates along a circular path, in the x-y plane by making an angle with the axes. The transformation is given by: X| = r cos (θ + φ) and Y| = r sin (θ + φ).

5. What is scaling?

The scaling transformations changes the shape of an object and can be carried out by multiplying each vertex (x, y) by scaling factor Sx, Sy where Sx is the scaling factor of x and Sy is the scaling factor of y.

6. What is shearing?

The shearing transformation actually slants the object along the X direction or the Y direction as required.ie; this transformation slants the shape of an object along a required plane.

7. What is reflection?

The reflection is actually the transformation that produces a mirror image of an object. For this use some angles and lines of reflection.

8. Distinguish between window port & view port?

A portion of a picture that is to be displayed by a window is known as window port. The display area of the part selected or the form in which the selected part is viewed is known as view port.

9. Define clipping?

Clipping is the method of cutting a graphics display to neatly fit a predefined graphics region or the view port.

10. What is covering (exterior clipping)?

This is just opposite to clipping. This removes the lines coming inside the windows and displays the remaining. Covering is mainly used to make labels on the complex pictures.

11. What is the need of homogeneous coordinates?

To perform more than one transformation at a time, use homogeneous coordinates or matrixes. They reduce unwanted calculations intermediate steps saves time and memory and produce a sequence of transformations.

12. Distinguish between uniform scaling and differential scaling?

When the scaling factors sx and sy are assigned to the same value, a uniform scaling is produced that maintains relative object proportions. Unequal values for sx and sy result in a differential scaling that is often used in design application.

13. What is fixed point scaling?

The location of a scaled object can be controlled by a position called the fixed point that is to remain unchanged after the scaling transformation.

14. Define affine transformation?

A coordinate transformation of the form

X= axxx + axyy + bx, y' ayxx + ayy y +by is called a two-dimensional affine transformation. Each of the transformed coordinates x 'and y 'is a linear function of the original coordinates x and y, and parameters aij and bk are constants determined by the transformation type.

15. Distinguish between bitBlt and pixBlt?

Raster functions that manipulate rectangular pixel arrays are generally referred to as raster ops. Moving a block of pixels from one location to another is also called a block transfer of pixel values. On a bilevel system, this operation is called a bitBlt (bit-block transfer), on multilevel system t is called pixBlt.

16. List out the various Text clipping?

All-or-none string clipping - if all of the string is inside a clip window, keep it otherwise discards.

All-or-none character clipping – discard only those characters that are not completely inside the window. Any character that either overlaps or is outside a window boundary is clipped.

Individual characters – if an individual character overlaps a clip window boundary, clip off the parts of the character that are outside the window.

UNIT – IV

1. What is the various representation schemes used in three dimensional objects?

Boundary representation (B-res) – describe the 3 dimensional object as a set of surfaces that separate the object interior from the environment.

Space- portioning representation – describe interior properties, by partitioning the spatial region containing an object into a set of small, no overlapping, contiguous solids.

2. What is Polygon mesh?

Polygon mesh is a method to represent the polygon, when the object surfaces are tiled; it is more convenient to specify the surface facets with a mesh function. The various meshes are

Triangle strip – (n-2) connected triangles

Quadrilateral mesh – generates (n-1)(m-1) Quadrilateral

3. What is Bezier Basis Function?

Bezier Basis functions are a set of polynomials, which can be used instead of the primitive polynomial basis, and have some useful properties for interactive curve design.

4. What is surface patch?

A single surface element can be defined as the surface traced out as two parameters (u, v) take all possible values between 0 and 1 in a two-parameter representation. Such a single surface element is known as a surface patch.

5. Write short notes on rendering bi-cubic surface patches of constant u and v method?

The simple way is to draw the isoperimetric lines of the surface. Discrete approximations to curves on the surface are produced by holding one parameter constant and allowing the other to vary at discrete intervals over its whole range. This produce curves of constant u and constant v.

6. What are the advantages of rendering polygons by scan line method?

- The max and min values of the scan were easily found.
- The intersection of scan lines with edges is easily calculated by a simple incremental method.
- The depth of the polygon at each pixel is easily calculated by an incremental method.

7. What are the advantages of rendering by patch splitting?

- It is fast- especially on workstations with a hardware polygon-rendering pipeline.
- Its speed can be varied by altering the depth of sub-division.

8. Define B-Spline curve?

A B-Spline curve is a set of piecewise (usually cubic) polynomial segments that pass close to a set of control points. However the curve does not pass through these control points, it only passes close to them.

9. What is a spline?

To produce a smooth curve through a designed set of points, a flexible strip called spline is used. Such a spline curve can be mathematically described with a piecewise cubic polynomial function whose first and second derivatives are continuous across various curve section.

10. What is the use of control points?

Spline curve can be specified by giving a set of coordinate positions called control points, which indicates the general shape of the curve, can specify spline curve.

11. What are the different ways of specifying spline curve?

- Using a set of boundary conditions that are imposed on the spline.
- Using the state matrix that characteristics the spline
- Using a set of blending functions that calculate the positions along the curve path by specifying combination of geometric constraints on the curve

12. What are the important properties of Bezier Curve?

- It needs only four control points
- It always passes through the first and last control points
- The curve lies entirely within the convex half formed by four control points.

13. Differentiate between interpolation spline and approximation spline?

When the spline curve passes through all the control points then it is called interpolate. When the curve is not passing through all the control points then that curve is called approximation spline.

14. What do you mean by parabolic splines?

For parabolic splines a parabola is fitted through the first three points p1,p2,p3 of the data array of kot points. Then a second parabolic arc is found to fit the sequence of point's p2, p3, p4. This continues in this way until a parabolic arc is found to fit through point's pn-2, pn-1 and pn. The final plotted curve is a meshing together of all these parabolic arcs.

15. What is cubic spline?

Cubic splines are a straight forward extension of the concepts underlying parabolic spline. The total curve in this case is a sequence of arcs of cubic rather than parabolic curves

Each cubic satisfies: ax3 + bx 2+ cx + d

16. What is a Blobby object?

Some objects do not maintain a fixed shape, but change their surface characteristics in certain motions or when in proximity to other objects. That is known as blobby objects. Example – molecular structures, water droplets.

17. Define Octrees?

Hierarchical tree structures called octrees, are used to represent solid objects in some graphics systems. Medical imaging and other applications that require displays of object cross sections commonly use octree representation.

18. Define Projection?

The process of displaying 3D into a 2D display unit is known as projection. The projection transforms 3D objects into a 2D projection plane.

19. What are the steps involved in 3D transformation?

- Modeling Transformation
- Viewing Transformation
- Projection Transformation
- Workstation Transformation

20. What do you mean by view plane?

A view plane is nothing but the film plane in camera which is positioned and oriented for a particular shot of the scene.

21. What is view-plane normal vector?

This normal vector is the direction perpendicular to the view plane and it is called as [DXN DYN DZN]

22. What is view distance?

The view plane normal vector is a directed line segment from the view plane to the view reference point. The length of this directed line segment is referred to as view distance

23. Define projection?

The process of converting the description of objects from world coordinates to viewing coordinates is known as projection

24. What you mean by parallel projection?

Parallel projection is one in which z coordinates is discarded and parallel lines from each vertex on the object are extended until they intersect the view plane.

25. What do you mean by Perspective projection?

Perspective projection is one in which the lines of projection are not parallel. Instead, they all converge at a single point called the center of projection.

26. What is Projection reference point?

In Perspective projection, the lines of projection are not parallel. Instead, they all converge at a single point called Projection reference point.

27. What is the use of Projection reference point?

In Perspective projection, the object positions are transformed to the view plane along these converged projection line and the projected view of an object is determined by calculating the intersection of the converged projection lines with the view plane.

28. What are the different types of parallel projections?

The parallel projections are basically categorized into two types, depending on the relation between the direction of projection and the normal to the view plane. They are orthographic parallel projection and oblique projection.

29. What is orthographic parallel projection?

When the direction of the projection is normal (perpendicular) to the view plane then the projection is known as orthographic parallel projection

30. What is orthographic oblique projection?

When the direction of the projection is not normal (not perpendicular) to the view plane then the projection is known as oblique projection.

31. What is an axonometric orthographic projection?

The orthographic projection can display more than one face of an object. Such an orthographic projection is called axonometric orthographic projection.

32. What is cavalier projection?

The cavalier projection is one type of oblique projection, in which the direction of projection makes a 45-degree angle with the view plane.

33. What is cabinet projection?

The cabinet projection is one type of oblique projection, in which the direction of projection makes a n angle of arctan (2) =63.4- with the view plane.

34. What is vanishing point?

The perspective projections of any set of parallel lines that are not parallel to the projection plane converge to appoint known as vanishing point.

35. What do you mean by principle vanishing point?

The vanishing point of any set of lines that are parallel to one of the three principle axes of an object is referred to as a principle vanishing point or axis vanishing point.

36. What is view reference point?

The view reference point is the center of the viewing coordinate system. It is often chosen to be close to or on the surface of the some object in the scene.

UNIT – V

1. Define computer graphics animation?

Computer graphics animation is the use of computer graphics equipment where the graphics output presentation dynamically changes in real time. This is often also called real time animation.

2. What is tweening?

It is the process, which is applicable to animation objects defined by a sequence of points, and that change shape from frame to frame.

3. Define frame?

One of the shape photographs that a film or video is made of is known as frame.

4. What is key frame?

One of the shape photographs that a film or video is made of the shape of an object is known initially and for a small no of other frames called key frame.

5. What is pseudo animation?

Pseudo animation is creating a sequence of stills, photographing or video graphing each still as one frame, and then later playing back the frames at a faster speed.

6. What is the normal speed of a visual animation?

Visual animation requires a playback of at least 25 frames per second.

7. What are the different tricks used in computer graphics animation?

- Color looks Up Table manipulation b. Bit plane manipulation
- Use of UDCS

- Special drawing modes e. Sprites
- Bit blitting

8. What is color look up table?

In color display unit it is necessary to read 44-bit for each pixel from buffer. This very time consuming process. To avoid this video controller uses look up table to store many entries of pixel vales in RGB format. This look up table is commonly known as color look up table.

9. What is solid modeling?

The construction of 3 dimensional objects for graphics display is often referred to as solid modeling.

10. What is an intuitive interface?

The intuitive interface is one, which simulates the way a person would perform a corresponding operation on real object rather than have menu command.

11. What is Sprite?

A Sprite is graphics shape in animation and games programs. Each sprite provided in the system has its own memory area similar to but smaller than pixel RAM.

12. What is the UDC technique?

UDC stands for User Defined Character set. It is graphics animation trick, which is used in early microcomputer system.

13. What is the use of hidden line removing algorithm?

The hidden line removal algorithm determines the lines, edges, surfaces or volumes that are visible or invisible to an observer located at a specific point in space.

14. What is computer graphics realism?

The creation of realistic picture in computer graphics is known as realism. It is important in fields such as simulation, design, entertainments, advertising, research, education, command, and control.

15. How realistic pictures are created in computer graphics?

To create a realistic picture, it must be process the scene or picture through viewing-coordinate transformations and projection that transform three-dimensional viewing coordinates onto two-dimensional device coordinates.

16. What is Fractals?

A Fractal is an object whose shape is irregular at all scales.

17. What is a Fractal Dimension?

Fractal has infinite detail and fractal dimension. A fractal imbedded in n- dimensional space could have any fractional dimension between 0 and n. The Fractal Dimension D= LogN / Log S, Where N is the No of Pieces and S is the Scaling Factor.

18. What is random fractal?

The patterns in the random fractals are no longer perfect and the random defects at all scale.

19. What is geometric fractal?

A geometric fractal is a fractal that repeats self-similar patterns over all scales.

20. What is Koch curve?

The Koch curve can be drawn by dividing line into 4 equal segments with scaling factor 1/3. And middle 2 segments are so adjusted that they form adjustment sides of an equilateral triangle.

21. What is turtle graphics program?

The turtle program is a Robert that can move in 2 dimensions and it has a pencil for drawing. The turtle is defined by the following parameters.

- Position of the turtle (x, y)
- Heading of the turtle 0 the angle from the x axis.

22. What is graftals?

Graftals are applicable to represent realistic rendering plants and trees. A tree is represented by a String of symbols 0, 1, [,]

23. What is a Particle system?

A particle system is a method for modeling natural objects, or other irregularly shaped objects, that exhibit "fluid-like" properties. Particle systems are suitable for realistic rendering of fuzzy objects, smoke, sea and grass.

24. Give some examples for computer graphics standards?

i. CORE – The Core graphics standard

j. GKS – The Graphics Kernel system

k. PHIGS – The Programmers Hierarchical Interactive Graphics System. l. GSX – The Graphics system extension

m. NAPLPS – The North American presentation level protocol syntax.

1. **Explain refresh cathode ray tube?**

A beam of electrons, emitted by an electron gun, passes through focusing and deflection systems that direct the beam toward specified positions on the phosphor coated area.

- The phosphor that emits a small spot of light at each position contacted by the electron beam
- One way to keep phosphor glowing is to redraw the picture repeatedly by quickly directing the electron beam back over the same points
- This type of display is called is called ***refresh CRT***
- Primary components of electron gun are:
  - Heated metal cathode

Heat is supplied to the cathode by directing the beam through a coil of wire called the filament inside the cylindrical cathode structure.

  - Control grid

Intensity of the electron beam is controlled by setting the voltage levels on the control grid, which is a metal cylinder that fits to the cathode.

- Different kinds of phosphorus are available
- Besides color the major difference between phosphors is their persistence
- The maximum number of points that can be displayed without overlap on a CRT is referred to **as resolution.**
- Another property is aspect ratio

2. **Explain color CRT monitors?**

A CRT monitor displays color pictures by using a combination of phosphors that emit different colored light.

By combining the emitted light from the different phosphors, a range of colors can be generated

Two techniques

Beam penetration method

Shadow mask method

- **Beam penetration method**

Display color pictures

Two layers of phosphors, usually red and green

- A beam of slow electrons excites the outer red layer
- Fast electron penetrates through the red layer and excites the inner green layer

- An intermediate beam speeds, combinations of red and green light
- **Shadow mask method**
- They produce a wider range of colors
- Has three phosphor color dots at each pixel position
- One emits red light, another emits green light, and the third emits a blue light
- The three beams are deflected and focused as a group on to the shadow mask, which contains a series of holes aligned with a phosphor dot patterns.
- When the three beams passes through a hole in the shadow mask, they activate a dot triangle, which appears as a small color spot on the screen.

3. Explain direct view storage tubes and liquid crystal displays?

Liquid crystal displays

- Refers to the compounds having crystalline arrangement of molecules flow of liquid
- Two plates each glass plate contains a light polarized that are right angles to each other
- Two types

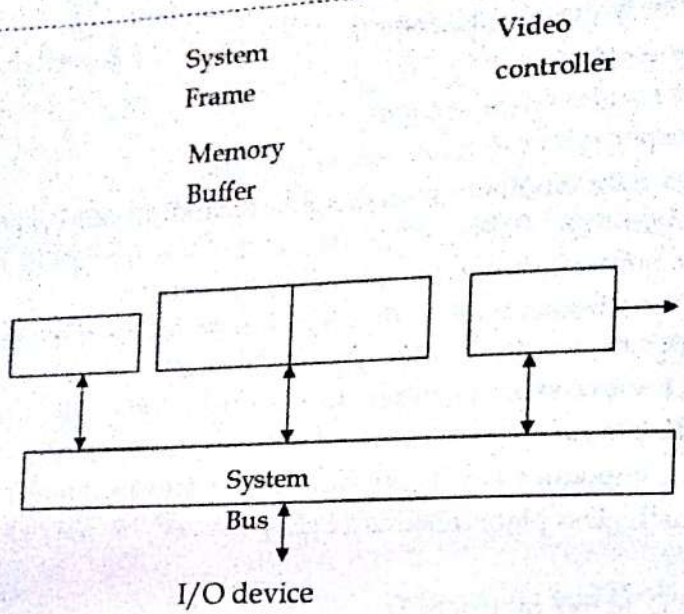
Passive matrix LCD Active matrix LCD

Direct view storage tubes

- Alternative method for maintaining a screen image
- Stores picture information as a charge distribution
- Very complex pictures can be displayed at very high resolutions
- To eliminate the picture section the entire screen must be erased
- The erasing and redrawing process can take several seconds for a picture

4. Explain Raster scan systems?

- Several processing units
- Contains a special purpose processor, called video controller or display controller
- **Video controller**
- A fixed area of the system memory is reserved for the frame buffer, and the video controller is given access to the frame- buffer memory
- Tow registers are used to store the coordinates of the screen pixels
- The value is stored in the frame buffer for this pixel position is then retrieved and used to set the intensity of the CRT beam
- Initially the x register is set to 0 and the y register is set to ymax
- Then the x register is incremented by 1, and the process repeated for the next pixel on the top scan line
- This process is repeated for each pixel along the scan line
- After the last pixels on the top scan line has been processed, the x register is reset to 0 and the yregister is decremented by 1.
- The procedure is repeated fro each successive line
- Frame buffer locations and the corresponding screen positions are referenced as Cartesian coordinates



4. **Explain the following?**

1. Z-mouse
2. Joysticks
3. Touch panels
4. Image scanners
5. Data glove

Z-Mouse:

- Include three buttons
- A thumb wheel on the side, a track ball on the top, and a standard mouse ball underneath.
- This design provides six degrees of freedom to select an object from the spatial position.
- Wit this we can pickup an object, rotate it and we can move it in any direction
- Used in virtual reality and CAD systems

Joysticks

- Consists of small vertical liver mounted on a base
- Used to move the cursor around the screen
- The screen cursor is moved according to the distance
- One or two buttons is usually intended for signaling certain actionsTouch panels
- Three types
- Optical touch panel
- Electrical touch panel
- Acoustical touch panel
- Allow selecting the screen position with the touch of finger. Touch input can be recorded using optical, electrical methods

Image scanners

- Drawings, color and black and white photos or text can be given as an input to the computer with an optical scanning mechanism.
- According to reflected light intensity the gradations of gray scale or color can be stored in an array

Data glove

- Constructed with a series of sensors that can detect hand and finger motions
- The transmitting and receiving antennas can be structured as a set of three mutually perpendicular cols, forming a three dimensional Cartesian coordinates system.
- Electromagnetic coupling between the three pairs of coil is used to provide information about the position and orientation of hand.

UNIT II

1. Explain in detail about the DDA scan conversion algorithm?

The digital differential analyzer is a scan conversion algorithm based on calculation either Δy or Δx using the following euations

$\Delta y = m \Delta x$

$\Delta x = \Delta y / m$

Sample the line at unit intervals in one coordinate and determinecorresponding integer values nearest the line path for the coordinates

Sample at X intervals ($\Delta x = 1$) and compute each successive Y value as

$Y_{k+1} = Y_k + m$

For lines with positive slope greater than 1, reverse the roles of X and Y. Sample at unit Y intervals ($\Delta y = 1$)and calculate each successive X value as

$X_{k+1} = X_k + 1/m$

Algorithm

Step 1: Input the line endpoints and store the left endpoint in (x1, y1) and right endpoint in (x2, y2)

Step 2: Calculate the values of Δx and Δy using $\Delta x = xb - xa$, $\Delta y = yb - ya$

Step 3: if the values of $\Delta x > \Delta y$ assign values of steps as Δx otherwise the values of steps as Δy

Step 4: Calculate the values of X increment and Y increment and assign the value x= xa and y = ya

Step 5: for k=1 to steps do

X = X + X increment Y= Y + Y increment Putpixel(ceil(x), ceil(y),15)

Step 6: End

2. Explain Bresenhams line drawing algorithm?

In Bresenham's approach the pixel position along a line path are determined by sampling unit X intervals. Starting from the left end point(X_0, Y_0)of a given line we step to each successive columns and plot the pixel whose scan line Y-value is closest to the line path.Assuming the Kth step in process, determined that the pixel at (X_k, Y_k)decide which pixel to plot in column X_{k+1}.The choices are (X_{k+1}, Y_k) and (X_{k+1}, Y_{k+1})

Algorithm

Step 1: Input the line endpoints and store the left endpoint in (X_0, Y_0) Step 2: Load (X_0, Y_0) in to the frame buffer

Step 3: Calculate constants Δx, Δy, $2 \Delta y$, $-2 \Delta x$, and obtain the decision parameters as

$P0 = 2 \Delta y - \Delta x$

Step 4 : At each X_k along the line, starting at k = 0, perform the following test

If $P_k < 0$, the next point to plot is (X_{k+1}, Y_k) and

$P_{k+1} = P_k + 2\Delta y$

Otherwise, the next point to plot is (X_{k+1}, Y_{k+1}) and

$P_{k+1} = P_k + 2\Delta y - 2\Delta x$

Step 5: Repeat step 4 Δx times

3. Explain Midpoint Circle algorithm?

Algorithm

Step 1:Input radius r and circle center(Xc, Yc)and obtain the first point on the circumference of a circle centered on the origin as

$(X0, Y0) = (0, r)$

Step 2: Calculate the initial values of the decision parameter as

$P0 = 5/4 - r$

Step 3: At each position starting at k perform the following test: If $P_k < 0$, the next point to plot is (X_{k+1}, Y_k) and

$P_{k+1} = P_k + 2X_{k+1} + 1$

Otherwise the next point is (X_{k+1}, Y_{k+1}) and

$P_{k+1} = P_k + 2X_{k+1} + 1 - 2Y_{k-1}$

Step 4: Determine symmetry points in the other seven octants

Step 5: Move each pixel position(X, Y) onto the circular path centred on

(Xc, Yc) and plot the coordinate values as

$X = X + X_c$ $\quad$ $Y = Y + Y_c$

Step 6: Repeat steps 3 through until $X >= Y$ $P_k + 1 = P_k + 2\Delta Y$

Other wise, the next point is (X_{k+1}, Y_{k+1}) and

$P_k + 1 = P_k + 2\Delta Y - 2\Delta X$ Step 5: Repeat steps 4 ΔX times

4. Explain Ellipse generating Algorithm?

Algorithm

Step 1:Input radius r_x, r_y and ellipse center$(X_c, Y_c)$and obtain the first point on the circumference of a circle centered on the origin as

$(X_0, Y_0) = (0, r_y)$

Step 2: Calculate the initial values of the decision parameter in region 1 as $P1_0 = r^2$

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

Step 3: At each position starting at X_k position in region 1,starting at k = 0, perform the following test:

If $P_k < 0$, the next point to plot is (X_{k+1}, Y_k) and

$P1_{k+1} = P1_k + 2 r_y X_{k+1} + r_y$

Otherwise the next point is (X_{k+1}, Y_{k-1}) and

$P1_{k+1} = P1_k + 2 r_y X_{k+1} - 2r_y \quad Y_{k+1} + r_y$

Step 4: Calculate the initial values of the decision parameter in region 2 as

$P2_0 = r_y \quad (X_0 + 1/2) + r_x \quad (Y_0 - 1) - r_x \; r_y$

Step 5: At each position starting at Y_k position in region 2,starting at k = 0, perform the following test:

If $P_k > 0$, the next point to plot is (X_k, Y_{k-1}) and

$P2k+1 = P2k - 2 r_y Y_{k+1} + r_x$

Otherwise the next point is (X_{k+1}, Y_{k-1}) and

$P2k+1 = P2k - 2 r_y Y_{k+1} - 2r_x Y_{k+1} + r_x$

Step 6: Determine symmetry points in the other three octants

Step 7: Move each pixel position(X, Y) onto the circular path centred on (Xc, Yc) and plot the coordinate values as

$X = X + X_c$ $Y = Y + Y_c$

Step 8: Repeat steps for region 1 until $2 r_y X >= 2 r_y Y$

5.Explain Boundary fill Algorithm?

- If the boundary is specified in a single color, the fill algorithm proceeds outward pixel-by-pixel until the boundary color is encountered. This is called boundary fill algorithm
- The boundary fill procedure accepts as input the coordinates of an interior point (x, y), a fill color, and a boundary color.
- Starting from (x, y), the procedure tests the neighboring positions to determine whether they are boundary color.
- If not, they are painted with the fill color, and the neighbors are tested.
- This process continues until all pixels up to the boundary color for the area have been tested
- Two methods
- 4- connected
- 4 neighbouring points are conected
- 8-connected
- correctly fill the interior of the area defined

program

```
void boundaryfill( int x, int y, int fill, int boundary)
{
int current;
current = getpixel (x, y);
if ((current != boundary) &&(current != nil)
{
setColor ( fill);
setPixel ( x, y);
boundaryfill (x+1, y, fill, boundary); boundaryfill (x-1, y, fill, boundary); boundaryfill (x, y+1, fill,
boundary); boundaryfill (x, y-1, fill, boundary);
}
}
```

UNIT III

1. Explain reflection and shear?

Reflection

- It is a mirror image of an object
- Rotating about 180 degree
- Flat object moving in the xy plane
- Reflection about y axis flips x coordinates

- Reflection point as pivot point is same as above
- To obtain the transformation matrix for reflection diagonal is y = -x
- Sequence
- Clockwise rotation by 45 degree
- Reflection about y axis
- Counter wise rotation by 45 degree

Shear

- Internal layer cause to slide over each other called shear
- Transforms coordinate position as

$X' = x + shx . y'$, $Y' = y$

- Shift in the position of objects relative to shearing reference lines are equivalent to translations

2. Explain Liang Barsky line clipping

- Faster line clipper of the parametric equation of a line segment
- Line parallel to one of the clipping boundaries
- Line intersects the extension of boundary k
- If u1 > u2 line is outside the clipping window
- Else inside the clipping window
- Clipping is done using the reflection in the clip window

3. Explain Sutherland Hodgeman polygon clipping

- Clipping polygon which lies inside the clipping window
- Four possible cases
- If the first vertex is outside the window boundary and the second vertex inside
- If the first vertex is inside the window boundary and the second vertex outside
- If both are outside
- If both are inside
- Repeat the process of algorithm
- Convex polygon are correctly clipped using this clipping
- Concave and convex polygon are also used

4. Explain about clipping operations

- Clip a picture from either outside or inside a region known as clipping
- Also called as clipping algorithm
- The region against the object is known as clip window
- Clipping operations on different types of objects
- Point clipping
- Polygon clipping
- Area clipping
- Line clipping
- Curve clipping
- Text clipping
- Polygon and line clipping are the standard clipping components

UNIT IV

1. Explain the three dimensional display methods?

- Parallel projection
- The production of the 2D display of the 3D scene is called projection
- Project points on the object surface along the parallel lines on to the display plane
- Different 2D views of objects can be produced by projecting the visible points
- Perspective projection
- Done by the projecting points to the display plane along the converging points
- Causes the objects farther from the viewing point should be smaller of the same sized object present here.
- Depth CUEING
- Basic problem for visualization techniques is called depth cueing
- Some 3D objects are without depth information
- Visible line and surface identification
- To highlight the visible lines
- Display visible lines as dashed lines
- Removing the invisible lines
- Surface rendering
- Lightening conditions in the screen
- Assigned characteristics
- Degree of transparency
- How rough or smooth the surfaces are to be
- Exploded and cutaway views
- Three dimensional and stereoscopic views

2. Explain spline representation

- It is referred to a curve drawn in a different manner
- Interpolation and approximation splines
- Set of coordinate points called control points
- Curve can be translated , rotated and scaled
- Enclosing a set of points called convex hull
- Set of connected points is often called control graph
- Parametric continuity condition
- Geometric continuity condition
- Spline specification

3. Explain Bezier curves and surfaces

- Have number of properties
- Can be fitted to any number of control points
- Polynomial functions between p0 and pn n

  $P(u) = \Sigma\ pk\ BEZ_{k,n}(u)\ K = 0$
- Calculated x(u), y(u), z(u)
- Properties of Bezier curves
- Cubic Bezier curves
- Design techniques in Bezier curves

- Bezier surfaces

4. Explain general three dimensional rotations

- Transformation sequences
- $P' = T'.Rx(\theta).T.P$
- Rotation in five steps
- Translate the object that rotates in parallel coordinate axis
- Rotate the object with one coordinate axis
- Apply inverse rotation to its original position
- Apply inverse translation to its original position
- V = p2 – p1
- After rotation to original position

$R(\theta) = T'.M.T$

UNIT V

1. Explain Back face detection method and Depth buffer method

Backface Detection method

- Fast and simple object space method
- For identifying the back faces of a polyhedron
- Based ion inside outside test
- Plane parameters A, B, C, D $Ax + By + Cz + D < 0$
- Inside point is along the line of sight to the surface

$V.N = Vz\ C$

- Polygon is back face if $C < 0$
- Polygon cannot see on any face if $C = 0$
- Z component value is $C <= 0$

Depth Buffer Method

- To compare the surface depths at each pixel position on the plane projection
- Also referred as z- buffer method
- Two buffer areas
- Depth buffer to store depth values
- Refresh buffer to store intensity values
- Depth value for a surface position (x, y) $Z(-Ax - By - D)/C$
- Depth z′ of next position (x + 1, y) $Z'(-A(x + 1) - By - D)/C$

2. Explain area subdivision and A- Buffer method

A Buffer Method

- Extension idea of the depth buffer method
- It represents an antialiased, area averaged, accumulation-buffer method
- Only one visible surface
- Two fields
- Depth field

Stores a positive or negative real number

- Intensity field

stores a surface intensity information or a pointer value

- Data for each surface in the linked list includes
- RGB intensity components
- Opacity parameters
- Depth
- Percent of area coverage
- Surface identifier
- Pointer to next surface

Area Sub Division Method

- Image space method
- The relative characteristic of surface are
- Surrounding surface
- Inside surface
- Outside surface
- Conditions for subdivision
- All surfaces are outside surfaces with respect to the area
- Only one inside
- A surrounding surface with in the area boundaries

3. Explain Depth sorting method

- Both image and object space operations
- Perform the basic function
- Surface are sorted in order of decreasing depth
- Surface are scan converted in order, starting with the surface of greatest depth
- Often referred as painters algorithm
- Test listed in the order of increasing difficulty
- Surface do not overlap
- Projections of the two surfaces on to the view plane do not overlap

4. Explain key frame systems

Key frame systems

- Motion path can be given with a kinematics description
- Physically based on force acting object
- Frame in to individual component or object called cells

Morphing

- Transformation of object shapes from one form to another is called morphing
- Two key frames for an object transformation
- Preprocessing using vertex count

Simulating accelerations

- Time interval between keyframe is divided in to n + 1

$\Delta t = (t2 - t1) / n = 1$

$tBj = t1 + j\Delta t, \quad j = 1,2,3, \ldots\ldots$

- Time for jth in- between is

$tBj = t1 + \Delta t[(1-\cos[j\pi / (n + 1)]) / 2], j= 1, 2, 3, \ldots.$

❑❑❑

KEC's B.Sc. CSIT Text Book Series

First Semester
- Introduction to Informatioin Technology
- C Programming
- Digital Logic
- Mathematics I
- Physics

Second Semester
- Discrete Structure
- Object Oriented Progamming
- Microprocessor
- Mathematics II
- Statistics I

Third Semester
- Data Structure and Algorithms
- Numerical Method
- Computer Architecture
- Computer Graphics
- Statistics II

Fourth Semester
- Theory of Computation
- Computer Networks
- Operating System
- Database Management System
- Artificial Intelligence

Fifth Semester
- Design and Analysis of Algorithms
- System Analysis and Design
- Cryptography
- Simulation and Modeling
- Web Technology
- Elective I

Sixth Semester
- Software Engineering
- Complier Design and Construction
- E-Governance
- NET Centric Computing
- Technical Writing
- Elective II

Seventh Semester
- Advanced Jave Programming
- Data Warehousing and Data Mining
- Principles of Management
- Project Work
- Elective III

Eighth Semester
- Advanced Database
- Internship
- Elective IV
- Elective V

KEC
Publication and Distribution Pvt. Ltd.
Kathmandu, Nepal, Tel.: 01-4168301, 01-4241777
E-mail: kecpublication14@gmail.com