

# **Delhi University (DU)**

## **B.Sc. (H) Computer Science**

### **Sem - VI**

## **Computer Graphics**

## **TutorialsDuniya.com**

**Visit <https://www.tutorialsduniya.com> for  
Compiled books, Notes, books, programs,  
question papers with solutions etc.**

**Facebook: <https://www.facebook.com/tutorialsduniya>**

**Youtube: <https://www.youtube.com/user/TutorialsDuniya>**

**LinkedIn: <https://www.linkedin.com/company/tutorialsduniya>**

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

**Computer Graphics Guidelines  
B.Sc(H) Computer Science 6<sup>th</sup> Semester  
(CBCS Guidelines)**

S.No	Topic	Reference	No. Of Lectures
1	<b>Introduction:</b> Basic elements of Computer graphics, Applications of computer graphics.	[2] Sections 1.1-1.8 (Pages 23-54)	3
2	<b>Graphics Hardware:</b> Architecture of Raster and Random scan display devices, input/output devices.	[2] Sections 2.1-2.6 (Pages 57-94)	5
3	<b>Drawing Primitives:</b> Raster scan line, circle and ellipse drawing algorithms, Polygon filling, line clipping and polygon, clipping algorithms	[1] Sections 3.2 -3.2.2 (Pages 72-78), Section 3.3 (Pages 81-85) (before 2 <sup>nd</sup> order differences), Section 3.4 (Pages 88-90), Sections 3.6 (Pages 92-99), Section 3.9 (Pages 104-109) , Section 3.12-3.12.3 (Pages 111-117), Section 3.14 (Pages 124-127), Section 3.17-3.17.3 (Pages 132-137)	12
4	<b>Viewing And Transformations:</b> 2D and 3D Geometric Transformations, 2D and 3D Viewing Transformations , Vanishing points.	[3] Sections 2.1-2.21 (Pages 61-99), Sections 3.1-3.17 (Pages 101-184)	12
5	<b>Geometric Modeling:</b> Representing curves(Hermite and Bezier)	[1] Section 11.2.1-11.2.2 (Pages 483-491)	6
6	<b>Visible Surface determination:</b> Z-buffer algorithm, Depth Sort algorithm and Warnock's algorithm	[1] Section 15.4-15.5.1 (Pages 668-675), Section 15.7.1 Pages(686-689)	6
7	<b>Surface rendering:</b> Color Models, Illumination and shading models, Computer Animation	[2] Sections 14.1-14.2 (Pages 516-531), Sections 14.4-14.5 (Pages 536-545), Sections 15.3-15.7 (Pages 591-597), Sections 16.1-16.6 (Pages 604-616)	6

#### Text Books

1. Computer Graphics: Principles and Practice 2nd Edition in C, James D. Foley , Andries van Dam, Steven K. Feiner , John F. Hughes , Pearson Education Asia, 1999.
2. Computer Graphics C version (2<sup>nd</sup> Edition), D.Hearn, M.P. Baker: Pearson Education, 2006.
3. Mathematical Elements for Computer Graphics 2<sup>nd</sup> Edition, D.F. Rogers, J. A. Adams, Mc Graw Hill 2 nd edition , 2002.

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

**Note: Solutions of all these programs are available on <https://www.tutorialsduniya.com>**

### **PRACTICAL LIST BASED ON COMPUTER GRAPHICS**

- 1) Write a program to implement Bresenham's line drawing algorithm,
- 2) Write a program to implement mid-point circle drawing algorithm
- 3) Write a program to clip a line using Cohen and Sutherland line clipping algorithm.
- 4) Write a program to clip a polygon using Sutherland Hodgeman algorithm.
- 5) Write a program to fill a polygon using Scan line fill algorithm.
- 6) Write a program to apply various 2D transformations on a 2D object (use homogenous coordinates).
- 7) Write a program to apply various 3D transformations on a 3D object and then apply parallel and perspective projection on it.
- 8) Write a program to draw Hermite/Bezier curve.

**Note: Solutions of all these programs are available on <https://www.tutorialsduniya.com>**

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

SECOND EDITION IN C

# Computer Graphics

## PRINCIPLES AND PRACTICE

**James D. Foley**

Georgia Institute of Technology

**Andries van Dam**

Brown University

**Steven K. Feiner**

Columbia University

**John F. Hughes**

Brown University

For any copyright query, contact us at [copyright@tutorialsduniya.com](mailto:copyright@tutorialsduniya.com)



**ADDISON-WESLEY**

Boston • San Francisco • New York • Toronto • Montreal

London • Munich • Paris • Madrid

Cape Town • Sydney • Tokyo • Singapore • Mexico City

# Contents

<b>CHAPTER 1</b>		
<b>INTRODUCTION</b>		<b>25</b>
1.1 Image Processing as Picture Analysis . . . . .	26	
1.2 The Advantages of Interactive Graphics . . . . .	27	
1.3 Representative Uses of Computer Graphics . . . . .	28	
1.4 Classification of Applications . . . . .	30	
1.5 Development of Hardware and Software for Computer Graphics . . . . .	32	
1.6 Conceptual Framework for Interactive Graphics . . . . .	41	
1.7 Summary . . . . .	45	
Exercises . . . . .	46	
<b>CHAPTER 2</b>		
<b>PROGRAMMING IN THE SIMPLE RASTER GRAPHICS PACKAGE (SRGP)</b>		<b>49</b>
2.1 Drawing with SRGP . . . . .	50	
2.2 Basic Interaction Handling . . . . .	64	
2.3 Raster Graphics Features . . . . .	76	
2.4 Limitations of SRGP . . . . .	84	
2.5 Summary . . . . .	87	
Exercises . . . . .	88	
<b>CHAPTER 3</b>		
<b>BASIC RASTER GRAPHICS ALGORITHMS FOR DRAWING 2D PRIMITIVES</b>		<b>91</b>
3.1 Overview . . . . .	91	
3.2 Scan Converting Lines . . . . .	96	
3.3 Scan Converting Circles . . . . .	105	
3.4 Scan Converting Ellipses . . . . .	112	
3.5 Filling Rectangles . . . . .	115	
3.6 Filling Polygons . . . . .	116	
3.7 Filling Ellipse Arcs . . . . .	123	
3.8 Pattern Filling . . . . .	124	
3.9 Thick Primitives . . . . .	128	
3.10 Line Style and Pen Style . . . . .	133	
3.11 Clipping in a Raster World . . . . .	134	
3.12 Clipping Lines . . . . .	135	

**18      Contents**

3.13	Clipping Circles and Ellipses . . . . .	148
3.14	Clipping Polygons . . . . .	148
3.15	Generating Characters . . . . .	151
3.16	SRGP_copyPixel . . . . .	156
3.17	Antialiasing . . . . .	156
3.18	Summary . . . . .	164
	Exercises . . . . .	166

**CHAPTER 11  
REPRESENTING CURVES AND SURFACES**

495

11.1	Polygon Meshes . . . . .	497
11.2	Parametric Cubic Curves . . . . .	502
11.3	Parametric Bicubic Surfaces . . . . .	540
11.4	Quadric Surfaces . . . . .	552
11.5	Summary . . . . .	553
	Exercises . . . . .	554

For any copyright query, contact us at [copyright@tutorialsduniya.com](mailto:copyright@tutorialsduniya.com)

**CHAPTER 15  
VISIBLE-SURFACE DETERMINATION**

673

15.1	Functions of Two Variables . . . . .	675
15.2	Techniques for Efficient Visible-Surface Algorithms . . . . .	680
15.3	Algorithms for Visible-Line Determination . . . . .	689
15.4	The z-Buffer Algorithm . . . . .	692
15.5	List-Priority Algorithms . . . . .	696
15.6	Scan-Line Algorithms . . . . .	704
15.7	Area-Subdivision Algorithms . . . . .	710
15.8	Algorithms for Octrees . . . . .	719
15.9	Algorithms for Curved Surfaces . . . . .	722
15.10	Visible-Surface Ray Tracing . . . . .	725
15.11	Summary . . . . .	739
	Exercises . . . . .	742

# TutorialsDuniya.com

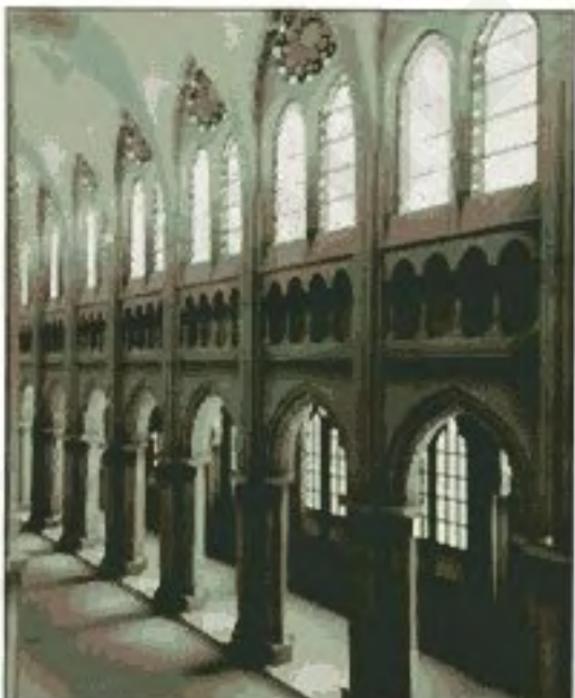
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

# COMPUTER GRAPHICS

C VERSION



DONALD HEARN ■ M. PAULINE BAKER

SECOND EDITION

# Contents

PREFACE	xvii	Stereoscopic and Virtual Reality Systems	50
<b>1 A Survey of Computer Graphics</b>	<b>2</b>	Raster-Scan Systems	53
1-1 Computer-Aided Design	4	Video Controller	53
1-2 Presentation Graphics	11	Raster-Scan Display Processor	55
1-3 Computer Art	13	Random-Scan Systems	56
1-4 Entertainment	18	Graphics Monitors and Workstations	57
1-5 Education and Training	21	Input Devices	60
1-6 Visualization	25	Keyboards	61
1-7 Image Processing	32	Mouse	61
1-8 Graphical User Interfaces	34	Trackball and Spaceball	63
<b>2 Overview of Graphics Systems</b>	<b>35</b>	Joysticks	63
2-1 Video Display Devices	36	Data Glove	64
Refresh Cathode-Ray Tubes	37	Digitizers	64
Raster-Scan Displays	40	Image Scanners	67
Random-Scan Displays	41	Touch Panels	68
Color CRT Monitors	42	Light Pens	70
Direct-View Storage Tubes	45	Voice Systems	70
Flat-Panel Displays	45	Hard-Copy Devices	72
Three-Dimensional Viewing Devices	49	Graphics Software	75
		Coordinate Representations	76
		Graphics Functions	77
		Software Standards	78
		PHIGS Workstations	79
		Summary	79
		References	81
		Exercises	81

For any copyright query, contact us at [copyright@tutorialsduniya.com](mailto:copyright@tutorialsduniya.com)

Contents

12-3	Projections	438	13-12	Wireframe Methods	490
	Parallel Projections	439	13-13	Visibility-Detection Functions	490
	Perspective Projections	443		Summary	491
12-4	View Volumes and General Projection Transformations	447		References	492
	General Parallel-Projection Transformations	452		Exercises	492
	General Perspective-Projection Transformations	454			
12-5	Clipping	456			
	Normalized View Volumes	458			
	Viewport Clipping	460			
	Clipping in Homogeneous Coordinates	461	14-1	Light Sources	496
12-6	Hardware Implementations	463	14-2	Basic Illumination Models	497
12-7	Three-Dimensional Viewing Functions	464		Ambient Light	497
	Summary	467		Diffuse Reflection	497
	References	468		Specular Reflection and the Phong Model	500
	Exercises	468		Combined Diffuse and Specular Reflections with Multiple Light Sources	504
				Warn Model	504
				Intensity Attenuation	505
				Color Considerations	507
				Transparency	508
				Shadows	511
<b>13</b>	<b>Visible-Surface Detection Methods</b>	<b>469</b>			
13-1	Classification of Visible-Surface Detection Algorithms	470	14-3	Displaying Light Intensities	511
13-2	Back-Face Detection	471		Assigning Intensity Levels	512
13-3	Depth-Buffer Method	472		Gamma Correction and Video Lookup Tables	513
13-4	A-Buffer Method	475		Displaying Continuous-Tone Images	515
13-5	Scan-Line Method	476			
13-6	Depth-Sorting Method	478	14-4	Halftone Patterns and Dithering Techniques	516
13-7	BSP-Tree Method	481		Halftone Approximations	516
13-8	Area-Subdivision Method	482		Dithering Techniques	519
13-9	Octree Methods	485			
13-10	Ray-Casting Method	487	14-5	Polygon-Rendering Methods	522
13-11	Curved Surfaces	488		Constant-Intensity Shading	522
	Curved-Surface Representations	488		Gouraud Shading	523
	Surface Contour Plots	489		Phong Shading	525

**14****Illumination Models and Surface-Rendering Methods** **494**

For any copyright query, contact us at [copyright@tutorialsduniya.com](mailto:copyright@tutorialsduniya.com)

Contents

			<b>15</b>	<b>Color Models and Color Applications</b>	<b>564</b>
15-1	Properties of Light	565			
15-2	Standard Primaries and the Chromaticity Diagram	568			
	XYZ Color Model	569			
	CIE Chromaticity Diagram	569			
15-3	Intuitive Color Concepts	571			
15-4	RGB Color Model	572			
15-5	YIQ Color Model	574			
			<b>16</b>	<b>Computer Animation</b>	<b>583</b>
14-6	Fast Phong Shading	526	15-6	CMY Color Model	574
	Ray-Tracing Methods	527	15-7	HSV Color Model	575
	Basic Ray-Tracing Algorithm	528	15-8	Conversion Between HSV and RGB Models	578
	Ray Surface Intersection Calculations	531	15-9	HLS Color Model	579
	Reducing Object Intersection Calculations	535	15-10	Color Selection and Applications	580
	Space-Subdivision Methods	535		Summary	581
	Antialiased Ray Tracing	538		References	581
	Distributed Ray Tracing	540		Exercises	582
14-7	Radiosity Lighting Model	544			
	Basic Radiosity Model	544			
	Progressive Refinement	549			
	Radiosity Method	549			
14-8	Environment Mapping	552			
14-9	Adding Surface Detail	553	16-1	Design of Animation Sequences	584
	Modeling Surface Detail with Polygons	553	16-2	General Computer-Animation Functions	586
	Texture Mapping	554	16-3	Raster Animations	589
	Procedural Texturing Methods	556	16-4	Computer-Animation Languages	587
	Bump Mapping	558	16-5	Key-Frame Systems	588
	Frame Mapping	559		Morphing	588
	Summary	560	16-6	Simulating Accelerations	591
	References	561		Motion Specifications	594
	Exercises	562		Direct Motion Specification	594
				Goal-Directed Systems	595
				Kinematics and Dynamics	595
				Summary	596
				References	597
				Exercises	597
			<b>A</b>	<b>Mathematics for Computer Graphics</b>	<b>599</b>
			A-1	Coordinate-Reference Frames	600
				Two-Dimensional Cartesian Reference Frames	600
				Polar Coordinates in the $xy$ Plane	601

# Introduction

S. No.	Topic	Contents
1.	<b>Basic elements of Computer graphics, Applications of computer graphics</b>	<b>Sections 1.1 - 1.8</b>

## TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

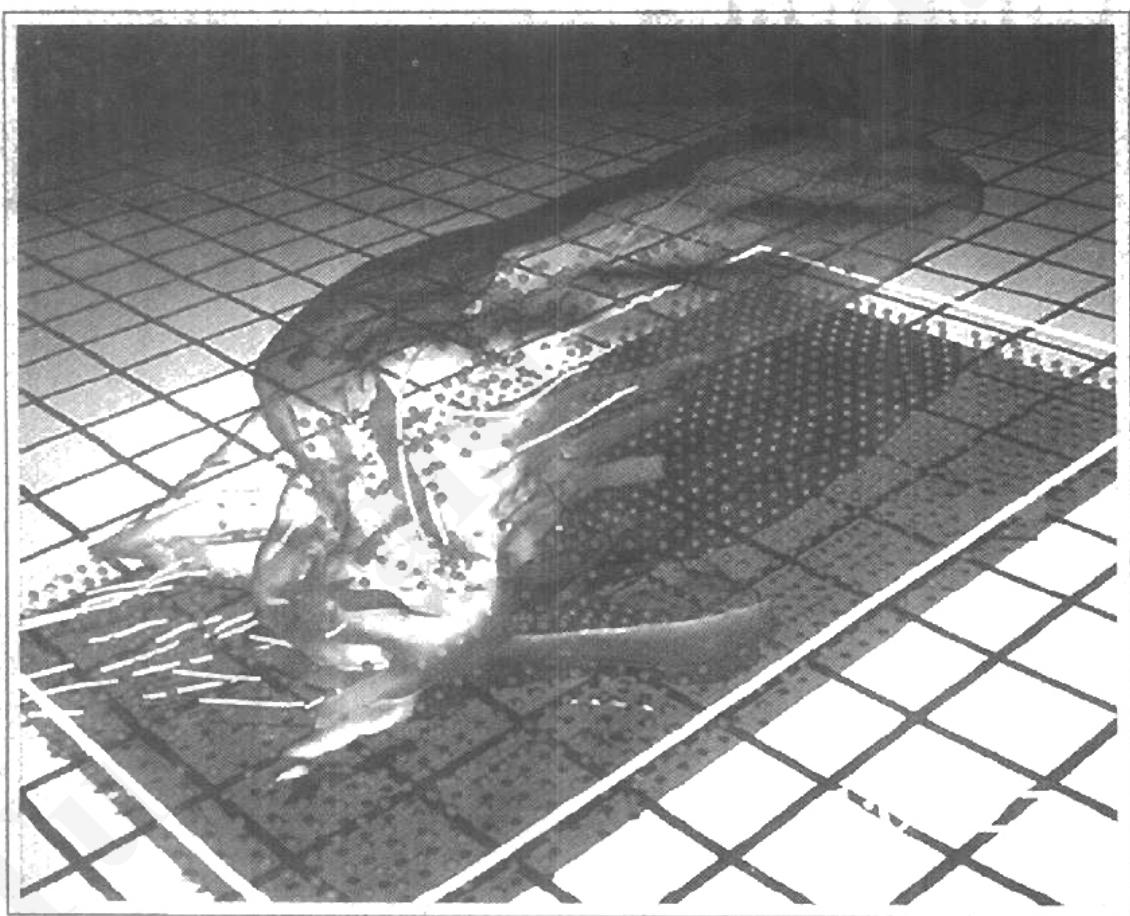
- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

CHAPTER

# 1

## A Survey of Computer Graphics



Computers have become a powerful tool for the rapid and economical production of pictures. There is virtually no area in which graphical displays cannot be used to some advantage, and so it is not surprising to find the use of computer graphics so widespread. Although early applications in engineering and science had to rely on expensive and cumbersome equipment, advances in computer technology have made interactive computer graphics a practical tool. Today, we find computer graphics used routinely in such diverse areas as science, engineering, medicine, business, industry, government, art, entertainment, advertising, education, and training. Figure 1-1 summarizes the many applications of graphics in simulations, education, and graph presentations. Before we get into the details of how to do computer graphics, we first take a short tour through a gallery of graphics applications.

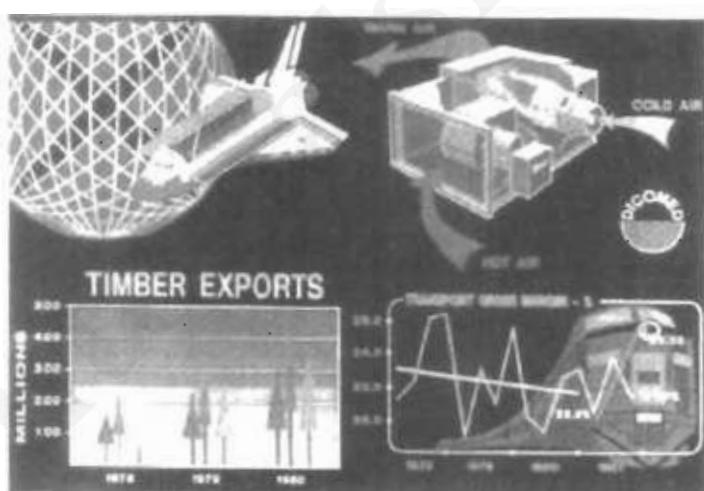


Figure 1-1  
Examples of computer graphics applications. (Courtesy of DICOMED Corporation.)

## COMPUTER-AIDED DESIGN

A major use of computer graphics is in design processes, particularly for engineering and architectural systems, but almost all products are now computer designed. Generally referred to as CAD, computer-aided design methods are now routinely used in the design of buildings, automobiles, aircraft, watercraft, spacecraft, computers, textiles, and many, many other products.

For some design applications, objects are first displayed in a wireframe outline form that shows the overall shape and internal features of objects. Wireframe displays also allow designers to quickly see the effects of interactive adjustments to design shapes. Figures 1-2 and 1-3 give examples of wireframe displays in design applications.

Software packages for CAD applications typically provide the designer with a multi-window environment, as in Figs. 1-4 and 1-5. The various displayed windows can show enlarged sections or different views of objects.

Circuits such as the one shown in Fig. 1-5 and networks for communications, water supply, or other utilities are constructed with repeated placement of a few graphical shapes. The shapes used in a design represent the different network or circuit components. Standard shapes for electrical, electronic, and logic circuits are often supplied by the design package. For other applications, a designer can create personalized symbols that are to be used to construct the network or circuit. The system is then designed by successively placing components into the layout, with the graphics package automatically providing the connections between components. This allows the designer to quickly try out alternate circuit schematics for minimizing the number of components or the space required for the system.

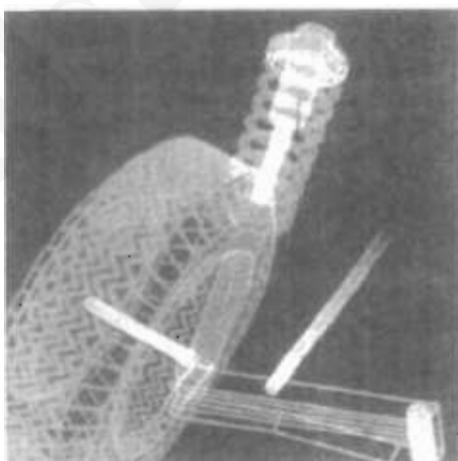


Figure 1-2  
Color-coded wireframe display for  
an automobile wheel assembly.  
(Courtesy of Evans & Sutherland.)

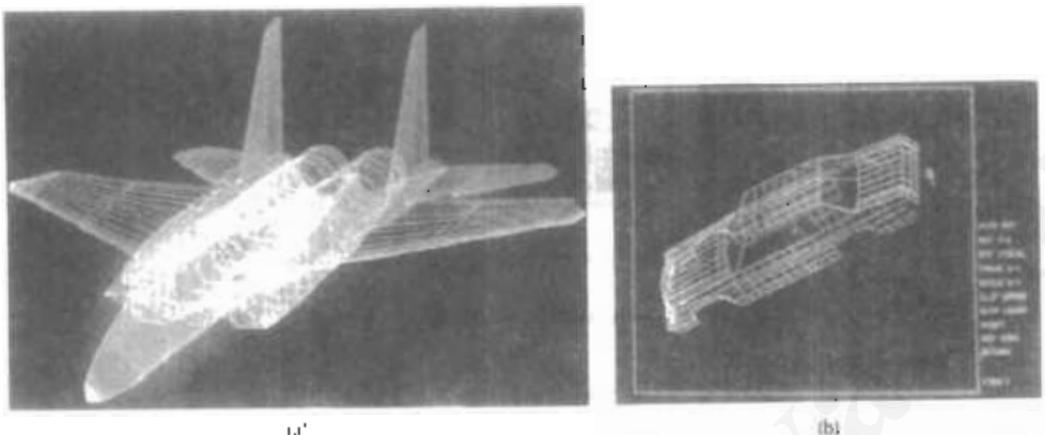


Figure 1-3

Color-coded wireframe displays of body designs for an aircraft and an automobile.  
(Courtesy of (a) Evans & Sutherland and (b) Megatek Corporation.)

Animations are often used in CAD applications. Real-time animations using wireframe displays on a video monitor are useful for testing performance of a vehicle or system, as demonstrated in Fig. 1-6. When we do not display objects with rendered surfaces, the calculations for each segment of the animation can be performed quickly to produce a smooth real-time motion on the screen. Also, wireframe displays allow the designer to see into the interior of the vehicle and to watch the behavior of inner components during motion. Animations in virtual-reality environments are used to determine how vehicle operators are affected by

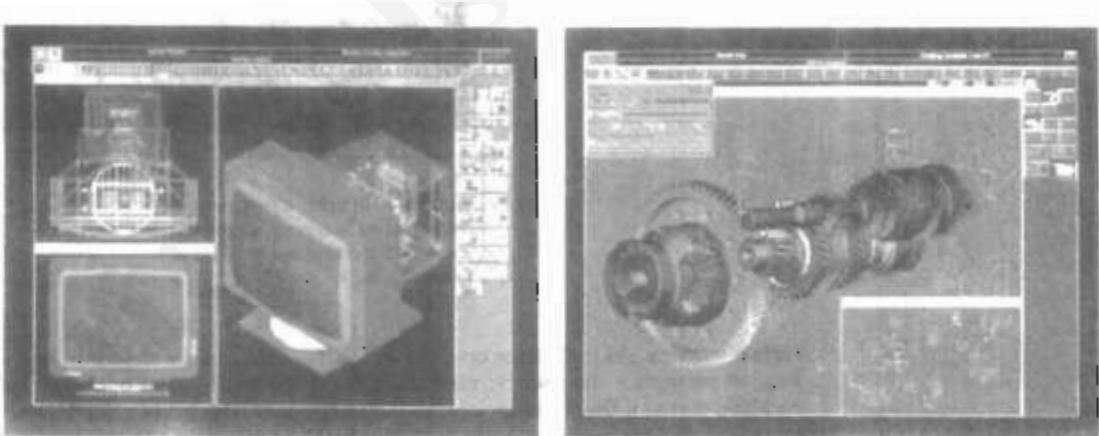
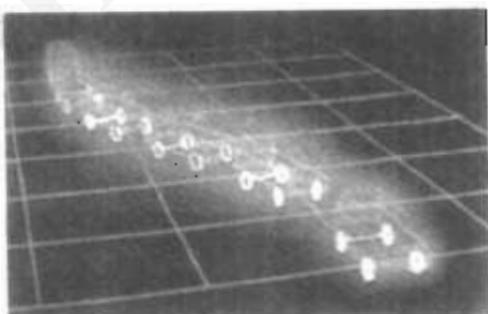


Figure 1-4

Multiple-window, color-coded CAD workstation displays. (Courtesy of Intergraph Corporation.)



*Figure 1-5*  
A circuit-design application, using multiple windows and color-coded logic components, displayed on a Sun workstation with attached speaker and microphone. (Courtesy of Sun Microsystems.)



*Figure 1-6*  
Simulation of vehicle performance during lane changes. (Courtesy of Evans & Sutherland and Mechanical Dynamics, Inc.)

certain motions. As the tractor operator in Fig. 1-7 manipulates the controls, the headset presents a stereoscopic view (Fig. 1-8) of the front-loader bucket or the backhoe, just as if the operator were in the tractor seat. This allows the designer to explore various positions of the bucket or backhoe that might obstruct the operator's view, which can then be taken into account in the overall tractor design. Figure 1-9 shows a composite, wide-angle view from the tractor seat, displayed on a standard video monitor instead of in a virtual three-dimensional scene. And Fig. 1-10 shows a view of the tractor that can be displayed in a separate window or on another monitor.



Figure 1-7

Operating a tractor in a virtual-reality environment. As the controls are moved, the operator views the front loader, backhoe, and surroundings through the headset. (Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, and Caterpillar, Inc.)



Figure 1-8

A headset view of the backhoe presented to the tractor operator. (Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, and Caterpillar, Inc.)

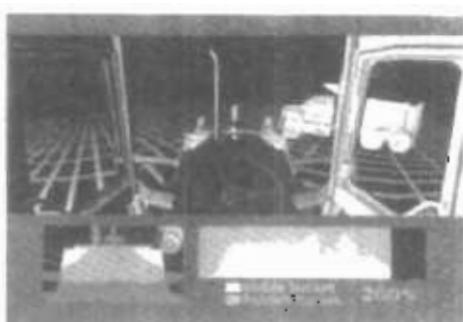


Figure 1-9

Operator's view of the tractor bucket, composed in several sections to form a wide-angle view on a standard monitor. (Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, and Caterpillar, Inc.)



Figure 1-10

View of the tractor displayed on a standard monitor. (Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, and Caterpillar, Inc.)

When object designs are complete, or nearly complete, realistic lighting models and surface rendering are applied to produce displays that will show the appearance of the final product. Examples of this are given in Fig. 1-11. Realistic displays are also generated for advertising of automobiles and other vehicles using special lighting effects and background scenes (Fig. 1-12).

The manufacturing process is also tied in to the computer description of designed objects to automate the construction of the product. A circuit board layout, for example, can be transformed into a description of the individual processes needed to construct the layout. Some mechanical parts are manufactured by describing how the surfaces are to be formed with machine tools. Figure 1-13 shows the path to be taken by machine tools over the surfaces of an object during its construction. Numerically controlled machine tools are then set up to manufacture the part according to these construction layouts.

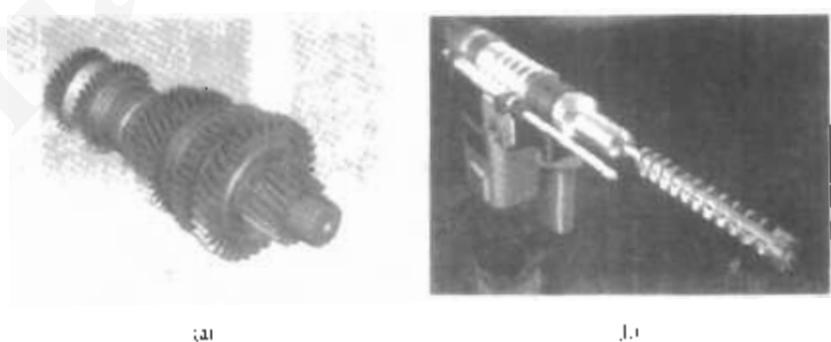
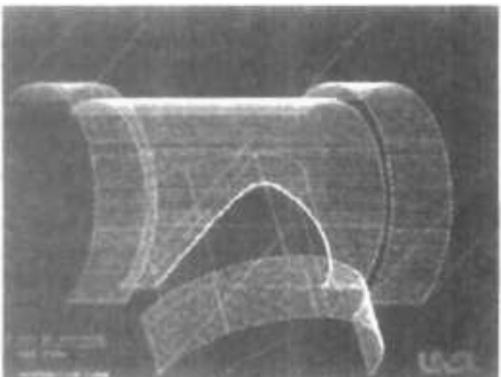


Figure 1-11

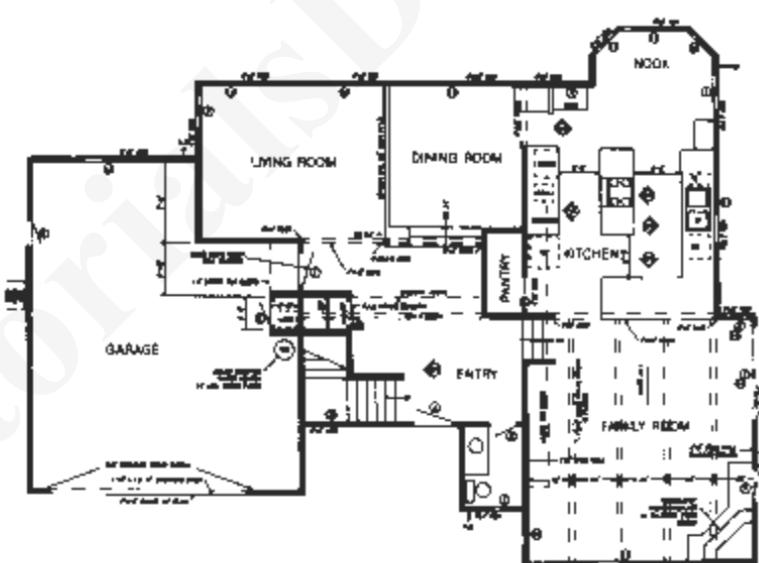
Realistic renderings of design products. (Courtesy of (a) Intergraph Corporation and (b) Evans & Sutherland.)



**Figure 1-12**  
Studio lighting effects and realistic surface-rendering techniques are applied to produce advertising pieces for finished products. The data for this rendering of a Chrysler Laser was supplied by Chrysler Corporation. (Courtesy of Eric Haines, 3D/EYE Inc.)



**Figure 1-13**  
A CAD layout for describing the numerically controlled machining of a part. The part surface is displayed in one color and the tool path in another color. (Courtesy of Los Alamos National Laboratory.)



**Figure 1-14**  
Architectural CAD layout for a building design. (Courtesy of Precision Visuals, Inc., Boulder, Colorado.)

Architects use interactive graphics methods to lay out floor plans, such as Fig. 1-14, that show the positioning of rooms, doors, windows, stairs, shelves, counters, and other building features. Working from the display of a building layout on a video monitor, an electrical designer can try out arrangements for wiring, electrical outlets, and fire warning systems. Also, facility-layout packages can be applied to the layout to determine space utilization in an office or on a manufacturing floor.

Realistic displays of architectural designs, as in Fig. 1-15, permit both architects and their clients to study the appearance of a single building or a group of buildings, such as a campus or industrial complex. With virtual-reality systems, designers can even go for a simulated "walk" through the rooms or around the outsides of buildings to better appreciate the overall effect of a particular design. In addition to realistic exterior building displays, architectural CAD packages also provide facilities for experimenting with three-dimensional interior layouts and lighting (Fig. 1-16).

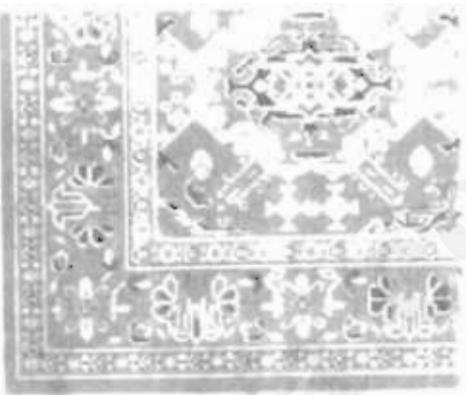
Many other kinds of systems and products are designed using either general CAD packages or specially developed CAD software. Figure 1-17, for example, shows a rug pattern designed with a CAD system.



**Figure 1-15**  
Realistic, three-dimensional renderings of building designs. (a) A street-level perspective for the World Trade Center project. (Courtesy of Skidmore, Owings & Merrill.)  
(b) Architectural visualization of an atrium, created for a computer animation by Marioline Prieur, Lyon, France. (Courtesy of Thomson Digital Image, Inc.)



**Figure 1-16**  
A hotel corridor providing a sense of movement by placing light fixtures along an undulating path and creating a sense of entry by using light towers at each hotel room. (Courtesy of Skidmore, Owings & Merrill.)



**Figure 1-17**  
Oriental rug pattern created with computer graphics design methods.  
(Courtesy of Lexidata Corporation.)

## 1-2

### PRESENTATION GRAPHICS

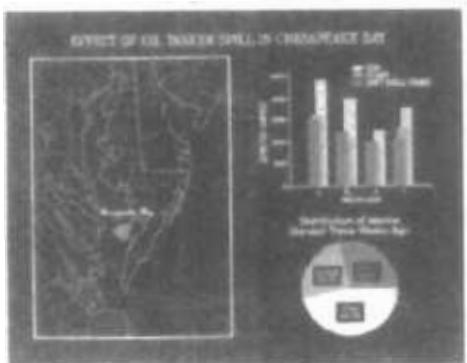
Another major application area is presentation graphics, used to produce illustrations for reports or to generate 35-mm slides or transparencies for use with projectors. Presentation graphics is commonly used to summarize financial, statistical, mathematical, scientific, and economic data for research reports, managerial reports, consumer information bulletins, and other types of reports. Workstation devices and service bureaus exist for converting screen displays into 35-mm slides or overhead transparencies for use in presentations. Typical examples of presentation graphics are bar charts, line graphs, surface graphs, pie charts, and other displays showing relationships between multiple parameters.

Figure 1-18 gives examples of two-dimensional graphics combined with geographical information. This illustration shows three color-coded bar charts combined onto one graph and a pie chart with three sections. Similar graphs and charts can be displayed in three dimensions to provide additional information. Three-dimensional graphs are sometimes used simply for effect; they can provide a more dramatic or more attractive presentation of data relationships. The charts in Fig. 1-19 include a three-dimensional bar graph and an exploded pie chart.

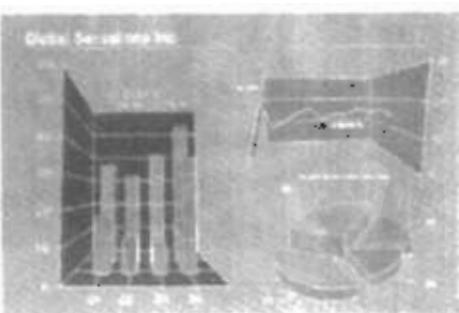
Additional examples of three-dimensional graphs are shown in Figs. 1-20 and 1-21. Figure 1-20 shows one kind of surface plot, and Fig. 1-21 shows a two-dimensional contour plot with a height surface.

Chapter 1

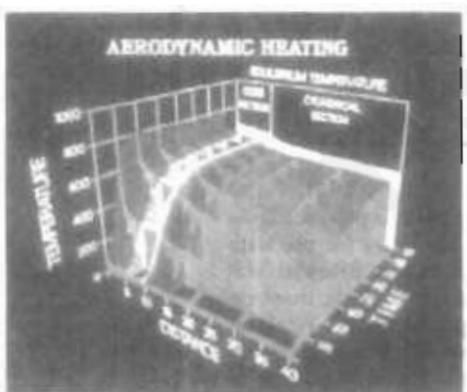
A Survey of Computer Graphics



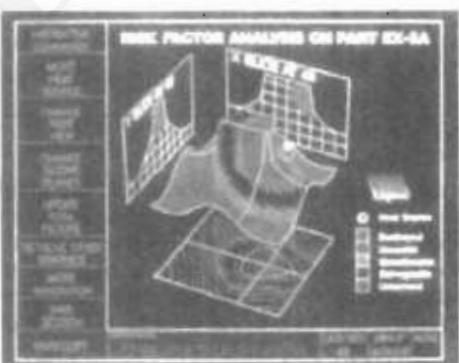
**Figure 1-18**  
Two-dimensional bar chart and pie chart linked to a geographical chart.  
(Courtesy of Computer Associates, copyright © 1992. All rights reserved.)



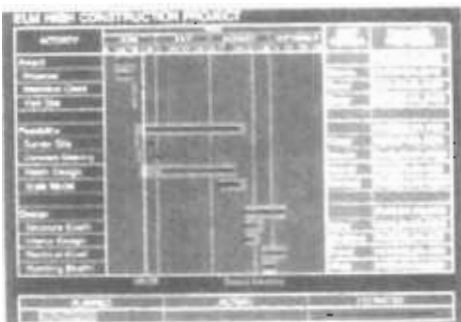
**Figure 1-19**  
Three-dimensional bar chart, exploded pie chart, and line graph.  
(Courtesy of Computer Associates, copyright © 1992. All rights reserved.)



**Figure 1-20**  
Showing relationships with a surface chart.  
(Courtesy of Computer Associates, copyright © 1992. All rights reserved.)



**Figure 1-21**  
Plotting two-dimensional contours in the ground plane, with a height field plotted as a surface above the ground plane.  
(Courtesy of Computer Associates, copyright © 1992. All rights reserved.)



### Section 1-3

---

Computer Art

**Figure 3-22**  
Time chart displaying relevant information about project tasks.  
(Courtesy of Computer Associates, copyright © 1992. All rights reserved.)

Figure 1-22 illustrates a time chart used in task planning. Time charts and task network layouts are used in project management to schedule and monitor the progress of projects.

1-3

---

COMPUTER ART

Computer graphics methods are widely used in both fine art and commercial art applications. Artists use a variety of computer methods, including special-purpose hardware, artist's paintbrush programs (such as Lumenia), other paint packages (such as PixelPaint and SuperPaint), specially developed software, symbolic mathematics packages (such as Mathematica), CAD packages, desktop publishing software, and animation packages that provide facilities for designing object shapes and specifying object motions.

Figure 1-23 illustrates the basic idea behind a paintbrush program that allows artists to "paint" pictures on the screen of a video monitor. Actually, the picture is usually painted electronically on a graphics tablet (digitizer) using a stylus, which can simulate different brush strokes, brush widths, and colors. A paintbrush program was used to create the characters in Fig. 1-24, who seem to be busy on a creation of their own.

A paintbrush system, with a Wacom cordless, pressure-sensitive stylus, was used to produce the electronic painting in Fig. 1-25 that simulates the brush strokes of Van Gogh. The stylus translates changing hand pressure into variable line widths, brush sizes, and color gradations. Figure 1-26 shows a watercolor painting produced with this stylus and with software that allows the artist to create watercolor, pastel, or oil brush effects that simulate different drying out times, wetness, and footprint. Figure 1-27 gives an example of paintbrush methods combined with scanned images.

Fine artists use a variety of other computer technologies to produce images. To create pictures such as the one shown in Fig. 1-28, the artist uses a combination of three-dimensional modeling packages, texture mapping, drawing programs, and CAD software. In Fig. 1-29, we have a painting produced on a pen

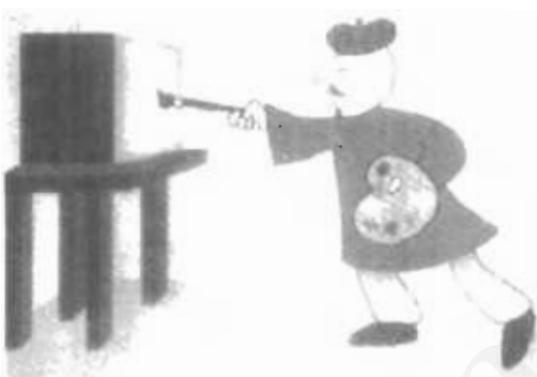


Figure 1-23

Cartoon drawing produced with a paintbrush program, symbolically illustrating an artist at work on a video monitor. (Courtesy of Gould Inc., Imaging & Graphics Division and Aurora Imaging.)

plotter with specially designed software that can create "automatic art" without intervention from the artist.

Figure 1-30 shows an example of "mathematical" art. This artist uses a combination of mathematical functions, fractal procedures, Mathematica software, ink-jet printers, and other systems to create a variety of three-dimensional and two-dimensional shapes and stereoscopic image pairs. Another example of elec-



(a)



(b)

Figure 1-24

Cartoon demonstrations of an "artist" creating a picture with a paintbrush system. The picture, drawn on a graphics tablet, is displayed on the video monitor as the elves look on. In (b), the cartoon is superimposed on the famous Thomas Nast drawing of Saint Nicholas, which was input to the system with a video camera, then scaled and positioned. (Courtesy Gould Inc., Imaging & Graphics Division and Aurora Imaging.)



**Figure 1-25**  
A Van Gogh look-alike created by graphics artist Elizabeth O'Rourke with a cordless, pressure-sensitive stylus. (Courtesy of Wacom Technology Corporation.)



**Figure 1-26**  
An electronic watercolor, painted by John Derry of Time Arts, Inc. using a cordless, pressure-sensitive stylus and Lumena gouache-brush software. (Courtesy of Wacom Technology Corporation.)



**Figure 1-27**  
The artist of this picture, called *Electronic Avalanche*, makes a statement about our entanglement with technology using a personal computer with a graphics tablet and Lumena software to combine renderings of leaves, flower petals, and electronics components with scanned images. (Courtesy of the Williams Gallery. Copyright © 1991 by John Truckenbrod. The School of the Art Institute of Chicago.)



Figure 1-28

From a series called *Spheres of Influence*, this electronic painting (entitled, *Wrigmalaree*) was created with a combination of methods using a graphics tablet, three-dimensional modeling, texture mapping, and a series of transformations. (Courtesy of the Williams Gallery. Copyright © 1992 by Wynne Ragland, Jr.)



Figure 1-29

Electronic art output to a pen plotter from software specially designed by the artist to emulate his style. The pen plotter includes multiple pens and painting instruments, including Chinese brushes. (Courtesy of the Williams Gallery. Copyright © by Roman Verostko, Minneapolis College of Art & Design.)



Figure 1-30

This creation is based on a visualization of Fermat's Last Theorem,  $x^n + y^n = z^n$ , with  $n = 5$ , by Andrew Hanson, Department of Computer Science, Indiana University. The image was rendered using Mathematica and Wavefront software. (Courtesy of the Williams Gallery. Copyright © 1991 by Stewart Dickson.)



Figure 1-31

Using mathematical functions, fractal procedures, and supercomputers, this artist-composer experiments with various designs to synthesize form and color with musical composition. (Courtesy of Brian Evans, Vanderbilt University.)

tronic art created with the aid of mathematical relationships is shown in Fig. 1-31. The artwork of this composer is often designed in relation to frequency variations and other parameters in a musical composition to produce a video that integrates visual and aural patterns.

Although we have spent some time discussing current techniques for generating electronic images in the fine arts, these methods are also applied in commercial art for logos and other designs, page layouts combining text and graphics, TV advertising spots, and other areas. A workstation for producing page layouts that combine text and graphics is illustrated in Fig. 1-32.

For many applications of commercial art (and in motion pictures and other applications), photorealistic techniques are used to render images of a product. Figure 1-33 shows an example of logo design, and Fig. 1-34 gives three computer graphics images for product advertising. Animations are also used frequently in advertising, and television commercials are produced frame by frame, where

#### Section 1-3

##### Computer Art



Figure 1-32  
Page-layout workstation. (Courtesy of Visual Technology.)



Figure 1-33  
Three-dimensional rendering for a logo. (Courtesy of Vertigo Technology, Inc.)



(a)



(b)



Figure 1-34  
Product advertising. (Courtesy of (a) Audrey Fleisher and (b) and (c) SOFTIMAGE, Inc.)

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

each frame of the motion is rendered and saved as an image file. In each successive frame, the motion is simulated by moving object positions slightly from their positions in the previous frame. When all frames in the animation sequence have been rendered, the frames are transferred to film or stored in a video buffer for playback. Film animations require 24 frames for each second in the animation sequence. If the animation is to be played back on a video monitor, 30 frames per second are required.

A common graphics method employed in many commercials is *morphing*, where one object is transformed (metamorphosed) into another. This method has been used in TV commercials to turn an oil can into an automobile engine, an automobile into a tiger, a puddle of water into a tire, and one person's face into another face. An example of morphing is given in Fig. 1-40.

#### 1-4 ENTERTAINMENT

Computer graphics methods are now commonly used in making motion 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 graphics scene generated for the movie *Star Trek—The Wrath of Khan* is shown in Fig. 1-35. The planet and spaceship are drawn in wireframe form and will be shaded with rendering methods to produce solid surfaces. Figure 1-36 shows scenes generated with advanced modeling and surface-rendering methods for two award-winning short films.

Many TV series regularly employ computer graphics methods. Figure 1-37 shows a scene produced for the series *Deep Space Nine*. And Fig. 1-38 shows a wireframe person combined with actors in a live scene for the series *Stay Tuned*.

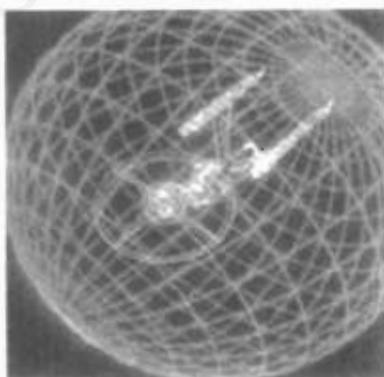


Figure 1-35  
Graphics developed for the Paramount Pictures movie *Star Trek—The Wrath of Khan*. (Courtesy of Evans & Sutherland.)

In Fig. 1-39, we have a highly realistic image taken from a reconstruction of thirteenth-century Dadu (now Beijing) for a Japanese broadcast.

Music videos use graphics in several ways. Graphics objects can be combined with the live action, as in Fig. 1-38, or graphics and image processing techniques can be used to produce a transformation of one person or object into another (morphing). An example of morphing is shown in the sequence of scenes in Fig. 1-40, produced for the David Byrne video *She's Mad*.



(a)



(b)

Figure 1-36

(a) A computer-generated scene from the film *Red's Dream*, copyright © Pixar 1987. (b) A computer-generated scene from the film *Knickknack*, copyright © Pixar 1989. (Courtesy of Pixar.)

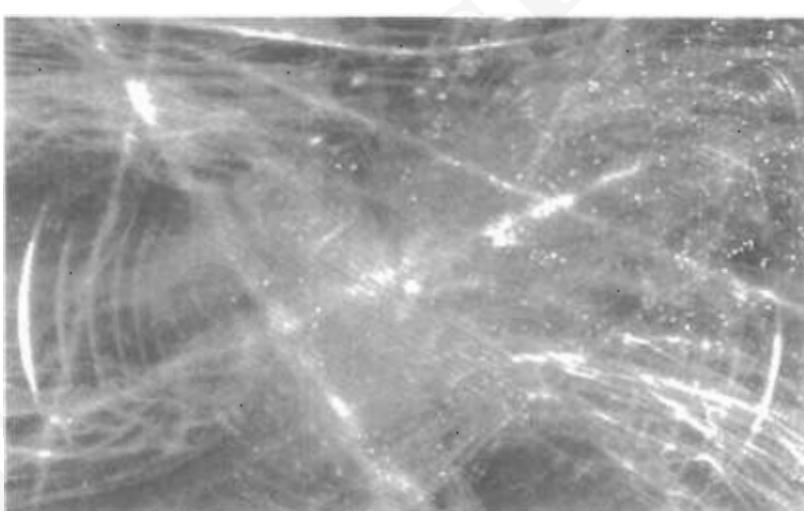
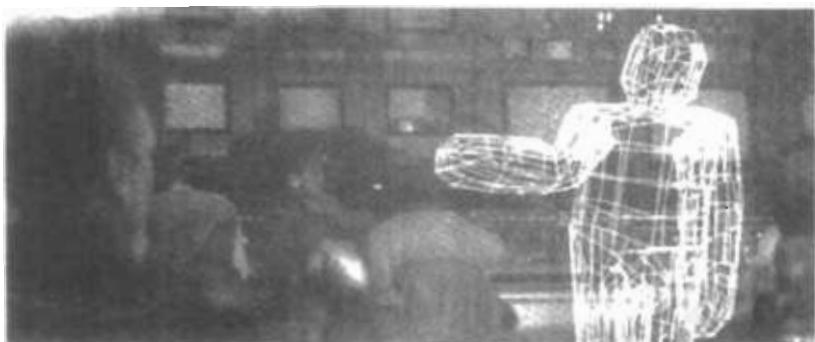


Figure 1-37

A graphics scene in the TV series *Deep Space Nine*. (Courtesy of Rhythm & Hues Studios.)



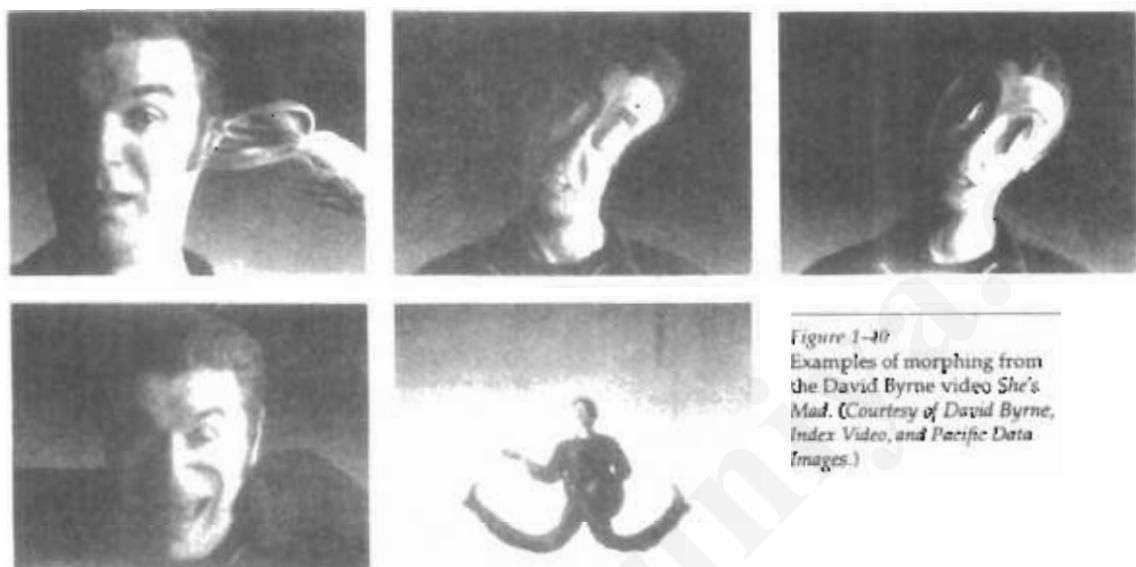
**Figure 1-38**  
Graphics combined with a live scene in the TV series *Stay Tuned*.  
(Courtesy of Rhythm & Hues Studios.)



**Figure 1-39**  
An image from a reconstruction of  
thirteenth-century Dadu (Beijing  
today), created by Taisei  
Corporation (Tokyo) and rendered  
with TDI software. (Courtesy of  
Thompson Digital Image, Inc.)

Section 1-5

Education and Training



**Figure 1-40**  
Examples of morphing from  
the David Byrne video *She's  
Mad*. (Courtesy of David Byrne,  
Index Video, and Pacific Data  
Images.)

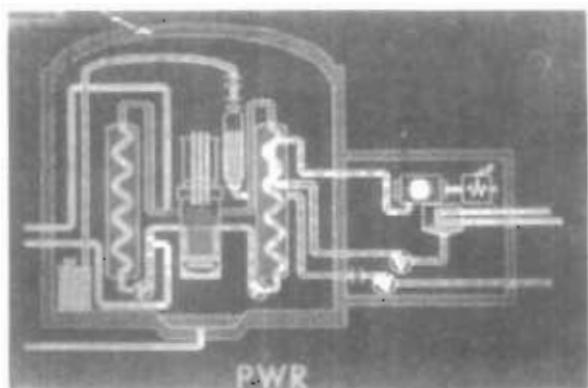
**1-5**

**EDUCATION AND TRAINING**

Computer-generated models of physical, financial, and economic systems are often used as educational aids. Models of physical systems, physiological systems, population trends, or equipment, such as the color-coded diagram in Fig. 1-41, can help trainees to understand the operation of the system.

For some training applications, special systems are designed. Examples of such specialized systems are the simulators for practice sessions or training of ship captains, aircraft pilots, heavy-equipment operators, and air traffic-control personnel. Some simulators have no video screens; for example, a flight simulator with only a control panel for instrument flying. But most simulators provide graphics screens for visual operation. Two examples of large simulators with internal viewing systems are shown in Figs. 1-42 and 1-43. Another type of viewing system is shown in Fig. 1-44. Here a viewing screen with multiple panels is mounted in front of the simulator, and color projectors display the flight scene on the screen panels. Similar viewing systems are used in simulators for training aircraft control-tower personnel. Figure 1-45 gives an example of the instructor's area in a flight simulator. The keyboard is used to input parameters affecting the airplane performance or the environment, and the pen plotter is used to chart the path of the aircraft during a training session.

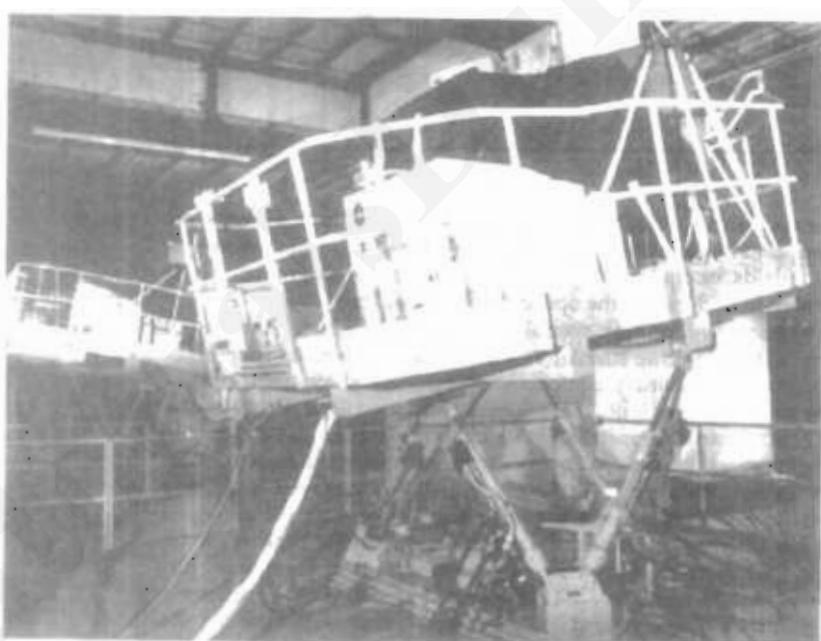
Scenes generated for various simulators are shown in Figs. 1-46 through 1-48. An output from an automobile-driving simulator is given in Fig. 1-49. This simulator is used to investigate the behavior of drivers in critical situations. The drivers' reactions are then used as a basis for optimizing vehicle design to maximize traffic safety.



**Figure 1-41**  
Color-coded diagram used to explain the operation of a nuclear reactor. (Courtesy of Los Alamos National Laboratory.)



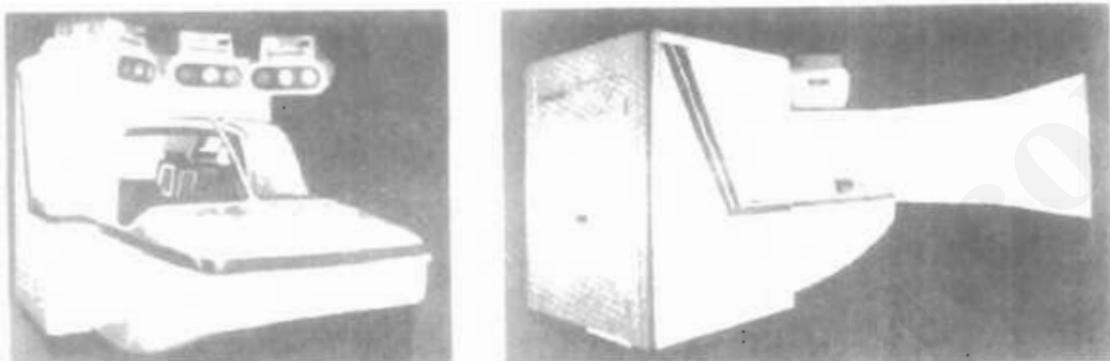
**Figure 1-42**  
A large, enclosed flight simulator with a full-color visual system and six degrees of freedom in its motion. (Courtesy of Frasca International.)



**Figure 1-43**  
A military tank simulator with a visual imagery system. (Courtesy of Mediatech and GE Aerospace.)

Section 1-5

Education and Training



**Figure 1-44**  
A flight simulator with an external full-color viewing system. (Courtesy of Frusco International.)



**Figure 1-45**  
An instructor's area in a flight simulator. The equipment allows the instructor to monitor flight conditions and to set airplane and environment parameters. (Courtesy of Frusco International.)

Chapter I

A Survey of Computer Graphics



**Figure 1-46**  
Flight-simulator imagery. (Courtesy of Evans & Sutherland.)



**Figure 1-47**  
Imagery generated for a naval simulator. (Courtesy of Evans & Sutherland.)



**Figure 1-48**  
Space shuttle imagery. (Courtesy of Mediatech and GE Aerospace.)



Figure 1-49  
Imagery from an automobile simulator used to test driver reaction. (Courtesy of Evans & Sutherland.)

## 1-6

### VISUALIZATION

Scientists, engineers, medical personnel, business analysts, and others often need to analyze large amounts of information or to study the behavior of certain processes. Numerical simulations carried out on supercomputers frequently produce data files containing thousands and even millions of data values. Similarly, satellite cameras and other sources are amassing large data files faster than they can be interpreted. Scanning these large sets of numbers to determine trends and relationships is a tedious and ineffective process. But if the data are converted to a visual form, the trends and patterns are often immediately apparent. Figure 1-50 shows an example of a large data set that has been converted to a color-coded display of relative heights above a ground plane. Once we have plotted the density values in this way, we can see easily the overall pattern of the data. Producing graphical representations for scientific, engineering, and medical data sets and processes is generally referred to as *scientific visualization*. And the term *business visualization* is used in connection with data sets related to commerce, industry, and other nonscientific areas.

There are many different kinds of data sets, and effective visualization schemes depend on the characteristics of the data. A collection of data can contain scalar values, vectors, higher-order tensors, or any combination of these data types. And data sets can be two-dimensional or three-dimensional. Color coding is just one way to visualize a data set. Additional techniques include contour plots, graphs and charts, surface renderings, and visualizations of volume interiors. In addition, image processing techniques are combined with computer graphics to produce many of the data visualizations.

Mathematicians, physical scientists, and others use visual techniques to analyze mathematical functions and processes or simply to produce interesting graphical representations. A color plot of mathematical curve functions is shown in Fig. 1-51, and a surface plot of a function is shown in Fig. 1-52. Fractal proce-

Chapter 1

A Survey of Computer Graphics

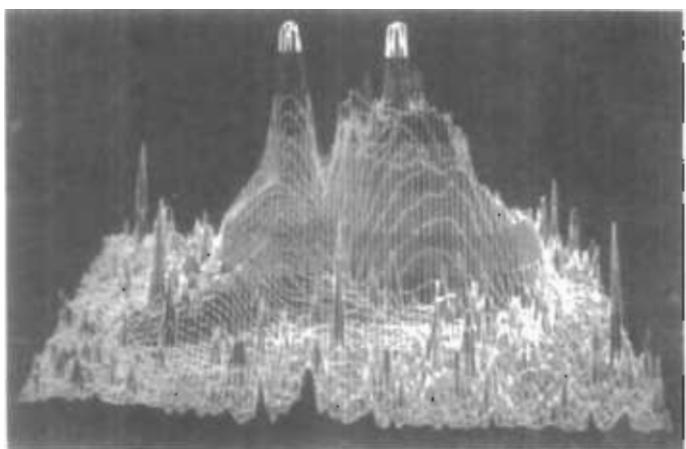


Figure 1-50

A color-coded plot with 16 million density points of relative brightness observed for the Whirlpool Nebula reveals two distinct galaxies.  
(Courtesy of Los Alamos National Laboratory.)

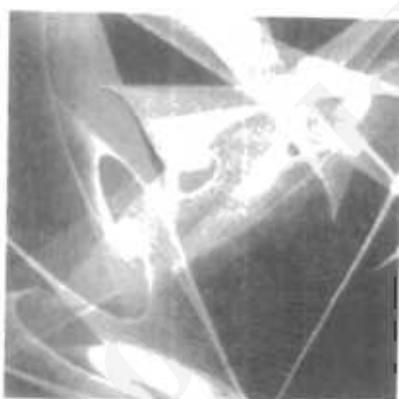


Figure 1-51

Mathematical curve functions plotted in various color combinations. (Courtesy of Melvin L. Prueitt, Los Alamos National Laboratory.)

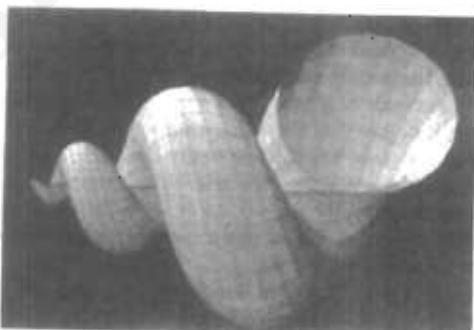


Figure 1-52

Lighting effects and surface-rendering techniques were applied to produce this surface representation for a three-dimensional function. (Courtesy of Wolfram Research, Inc, The Maker of Mathematica.)

dures using quaternions generated the object shown in Fig. 1-53, and a topological structure is displayed in Fig. 1-54. Scientists are also developing methods for visualizing general classes of data. Figure 1-55 shows a general technique for graphing and modeling data distributed over a spherical surface.

A few of the many other visualization applications are shown in Figs. 1-56 through 1-69. These figures show airflow over the surface of a space shuttle; numerical modeling of thunderstorms; study of crack propagation in metals; a color-coded plot of fluid density over an airfoil; a cross-sectional slicer for data sets; protein modeling; stereoscopic viewing of molecular structure; a model of the ocean floor; a Kuwaiti oil-fire simulation; an air-pollution study; a corn-growing study; reconstruction of Arizona's Chaco Canyon ruins; and a graph of automobile accident statistics.

---

**Section 1-6**  
Visualization



**Figure 1-53**  
A four-dimensional object projected into three-dimensional space, then projected to a video monitor, and color coded. The object was generated using quaternions and fractal squaring procedures, with an octant subtracted to show the complex Julia set. (Courtesy of John C. Hart, School of Electrical Engineering and Computer Science, Washington State University.)



**Figure 1-54**

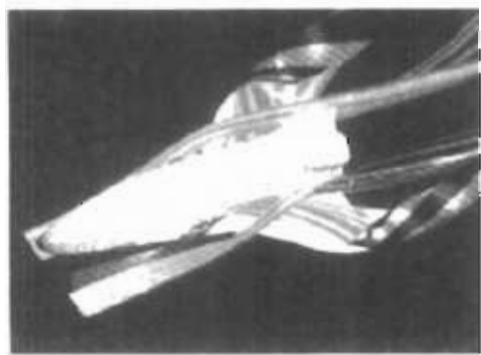
Four views from a real-time, interactive computer-animation study of minimal surfaces ("snails") in the 3-sphere projected to three-dimensional Euclidean space. (Courtesy of George Francis, Department of Mathematics and the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign. Copyright © 1993.)



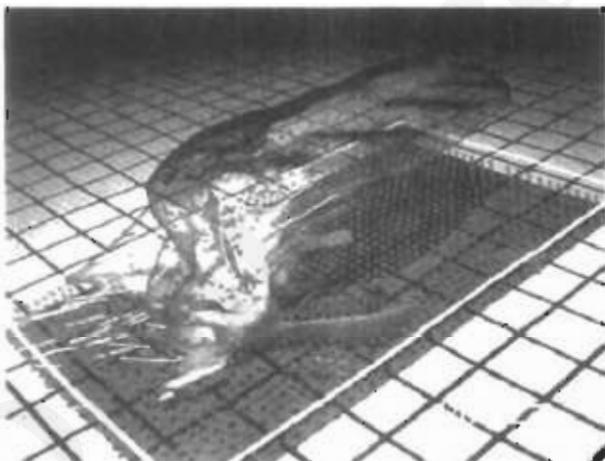
**Figure 1-55**  
A method for graphing and modeling data distributed over a spherical surface. (Courtesy of Greg Nielson, Computer Science Department, Arizona State University.)

Chapter 1

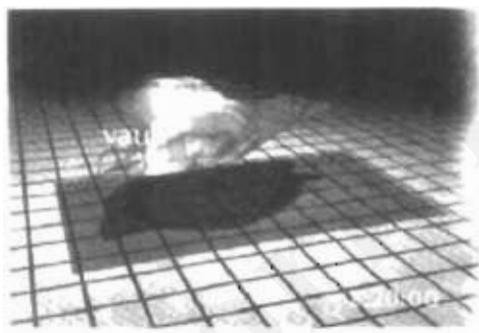
A Survey of Computer Graphics



**Figure 1-56**  
A visualization of stream surfaces flowing past a space shuttle by Jeff Hultquist and Eric Raible, NASA Ames. (Courtesy of Sam Lilesen, NASA Ames Research Center.)

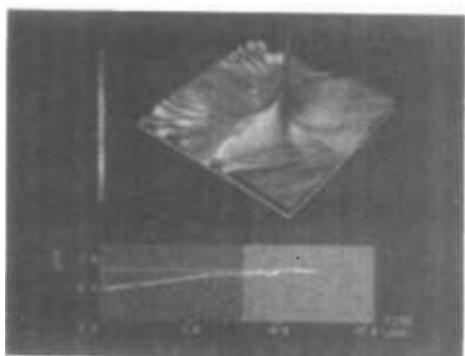


**Figure 1-57**  
Numerical model of airflow inside a thunderstorm. (Courtesy of Bob Wilhelmson, Department of Atmospheric Sciences and the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)

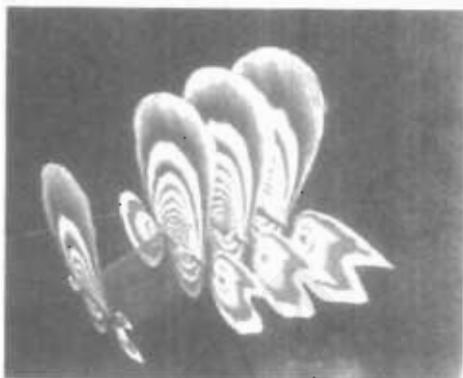


**Figure 1-58**  
Numerical model of the surface of a thunderstorm. (Courtesy of Bob Wilhelmson, Department of Atmospheric Sciences and the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)

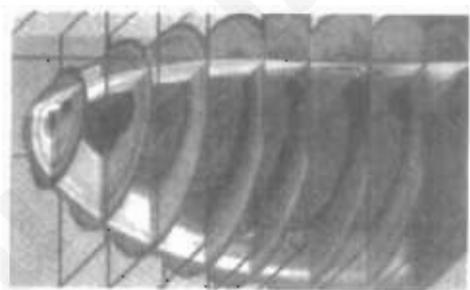
Section 1-6  
Visualization



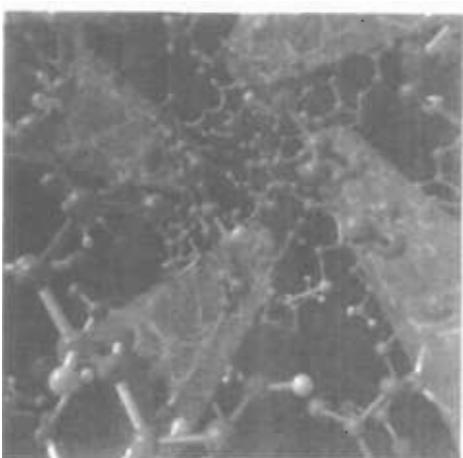
*Figure 1-59*  
Color-coded visualization of stress energy density in a crack-propagation study for metal plates, modeled by Bob Haber. (Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)



*Figure 1-60*  
A fluid dynamic simulation, showing a color-coded plot of fluid density over a span of grid planes around an aircraft wing, developed by Lee-Hian Quek, John Eickemeyer, and Jeffrey Tan. (Courtesy of the Information Technology Institute, Republic of Singapore.)



*Figure 1-61*  
Commercial slicer-dicer software, showing color-coded data values over cross-sectional slices of a data set. (Courtesy of Spyglass, Inc.)



*Figure 1-62*  
Visualization of a protein structure by Jay Siegel and Kim Baldridge, SDSC. (Courtesy of Stephanie Sides, San Diego Supercomputer Center.)

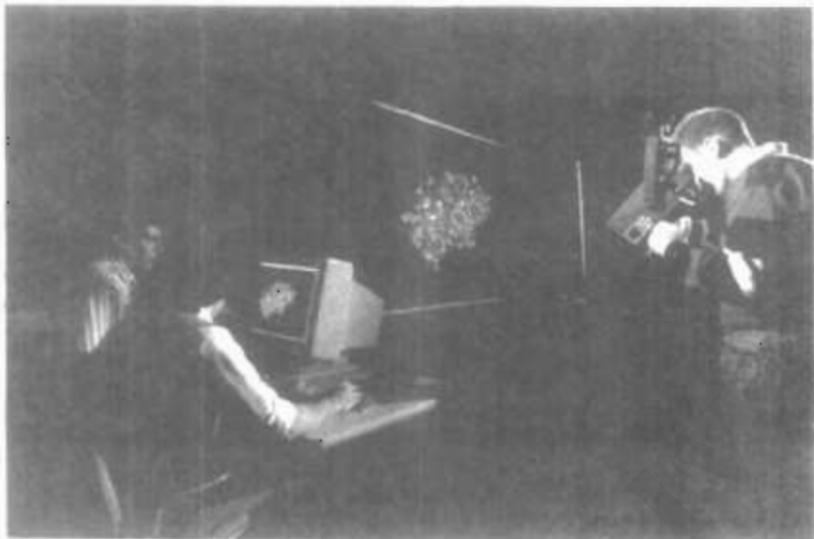


Figure 1-63

Stereoscopic viewing of a molecular structure using a "boom" device.  
(Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)

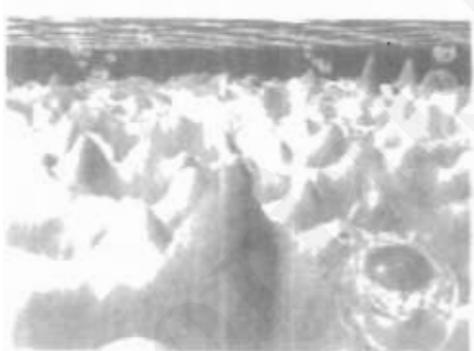


Figure 1-64

One image from a stereoscopic pair, showing a visualization of the ocean floor obtained from satellite data, by David Sandwell and Chris Small, Scripps Institution of Oceanography, and Jim McLeod, SDSC. (Courtesy of Stephanie Sides, San Diego Supercomputer Center.)

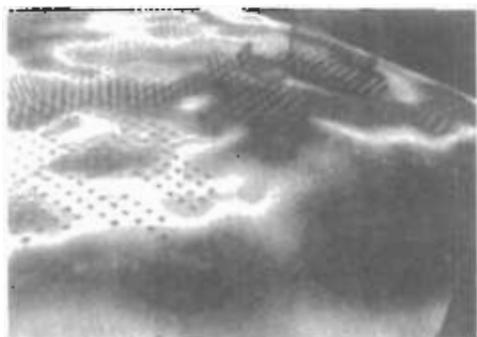


Figure 1-65

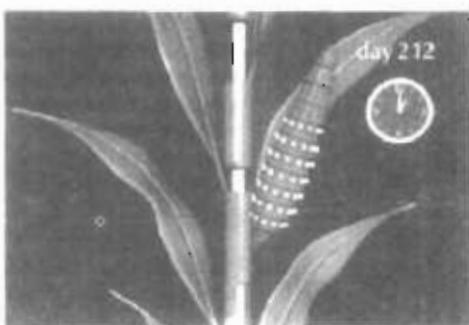
A simulation of the effects of the Kuwaiti oil fire, by Gary Glatzmaier, Chuck Hansen, and Paul Hinken. (Courtesy of Mike Knapp, Advanced Computing Laboratory at Los Alamos National Laboratory.)

Section 1-6

Visualization



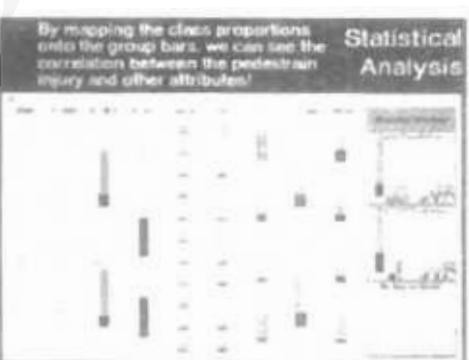
**Figure 1-66**  
A visualization of pollution over the earth's surface by Tom Palmer, Cray Research Inc./NCSC; Chris Landreth, NCSC; and Dave Bock, NCSC. Pollutant SO<sub>2</sub> is plotted as a blue surface, acid-rain deposition is a color plane on the map surface, and rain concentration is shown as clear cylinders. (Courtesy of the North Carolina Supercomputing Center/MCNC.)



**Figure 1-67**  
One frame of an animation sequence showing the development of a corn ear. (Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)



**Figure 1-68**  
A visualization of the reconstruction of the ruins at Chaco Canyon, Arizona. (Courtesy of Melvin L. Pruitt, Los Alamos National Laboratory. Data supplied by Stephen H. Lekson.)



**Figure 1-69**  
A prototype technique, called WinViz, for visualizing tabular multidimensional data is used here to correlate statistical information on pedestrians involved in automobile accidents, developed by a visualization team at ITT. (Courtesy of Lee-Hian Quek, Information Technology Institute, Republic of Singapore.)

## 1-7

### IMAGE PROCESSING

Although methods used in computer graphics and image processing overlap, the two areas are concerned with fundamentally different operations. In computer graphics, a computer is used to create a picture. Image processing, on the other hand, applies techniques to modify or interpret existing pictures, such as photographs and TV scans. Two principal applications of image processing are (1) improving picture quality and (2) machine perception of visual information, as used in robotics.

To apply image-processing methods, we first digitize a photograph or other picture into an image file. Then digital methods can be applied to rearrange picture parts, to enhance color separations, or to improve the quality of shading. An example of the application of image-processing methods to enhance the quality of a picture is shown in Fig. 1-70. These techniques are used extensively in commercial art applications that involve the retouching and rearranging of sections of photographs and other artwork. Similar methods are used to analyze satellite photos of the earth and photos of galaxies.

Medical applications also make extensive use of image-processing techniques for picture enhancements, in tomography and in simulations of operations. Tomography is a technique of X-ray photography that allows cross-sectional views of physiological systems to be displayed. Both *computed X-ray tomography (CT)* and *positron emission tomography (PET)* use projection methods to reconstruct cross sections from digital data. These techniques are also used to

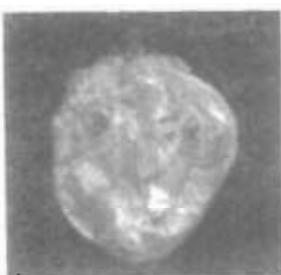


Figure 1-70

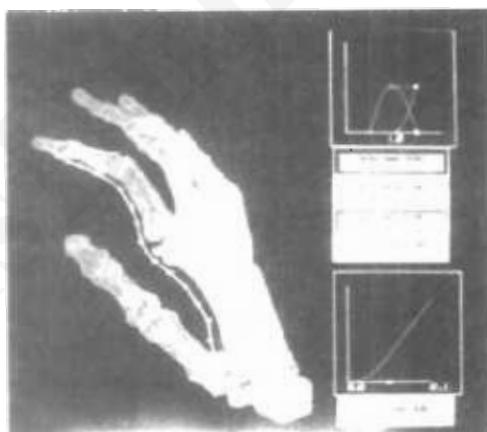
A blurred photograph of a license plate becomes legible after the application of image-processing techniques. (Courtesy of Los Alamos National Laboratory.)

monitor internal functions and show cross sections during surgery. Other medical imaging techniques include ultrasonics and nuclear medicine scanners. With ultrasonics, high-frequency sound waves, instead of X-rays, are used to generate digital data. Nuclear medicine scanners collect digital data from radiation emitted from ingested radionuclides and plot color-coded images.

Image processing and computer graphics are typically combined in many applications. Medicine, for example, uses these techniques to model and study physical functions, to design artificial limbs, and to plan and practice surgery. The last application is generally referred to as *computer-aided surgery*. Two-dimensional cross sections of the body are obtained using imaging techniques. Then the slices are viewed and manipulated using graphics methods to simulate actual surgical procedures and to try out different surgical cuts. Examples of these medical applications are shown in Figs. 1-71 and 1-72.



**Figure 1-71**  
One frame from a computer animation visualizing cardiac activation levels within regions of a semitransparent volume-rendered dog heart. Medical data provided by William Smith, Ed Simpson, and G. Allan Johnson, Duke University. Image-rendering software by Tom Palmer, Cray Research, Inc./NCSC. (Courtesy of Dave Bock, North Carolina Supercomputing Center/MCNC.)



**Figure 1-72**  
One image from a stereoscopic pair showing the bones of a human hand. The images were rendered by Inmo Yoon, D. E. Thompson, and W. N. Waggoner, Jr., LSU, from a data set obtained with CT scans by Rehabilitation Research, GWLNRDC. These images show a possible tendon path for reconstructive surgery. (Courtesy of IMRLAB, Mechanical Engineering, Louisiana State University.)

## GRAPHICAL USER INTERFACES

It is common now for software packages to provide a graphical interface. A major component of a graphical interface is a window manager that allows a user to display multiple-window areas. Each window can contain a different process that can contain graphical or nongraphical displays. To make a particular window active, we simply click in that window using an interactive pointing device.

Interfaces also display menus and icons for fast selection of processing options or parameter values. An icon is a graphical symbol that is designed to look like the processing option it represents. The advantages of icons are that they take up less screen space than corresponding textual descriptions and they can be understood more quickly if well designed. Menus contain lists of textual descriptions and icons.

Figure 1-73 illustrates a typical graphical interface, containing a window manager, menu displays, and icons. In this example, the menus allow selection of processing options, color values, and graphics parameters. The icons represent options for painting, drawing, zooming, typing text strings, and other operations connected with picture construction.



**Figure 1-73**  
A graphical user interface, showing multiple window areas, menus, and icons. (Courtesy of Image-In Corporation.)

# Graphics Hardware

S. No.	Topic	Contents
2.	Architecture of Raster and Random scan display devices, input/output devices	Sections 2.1 - 2.6

## TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

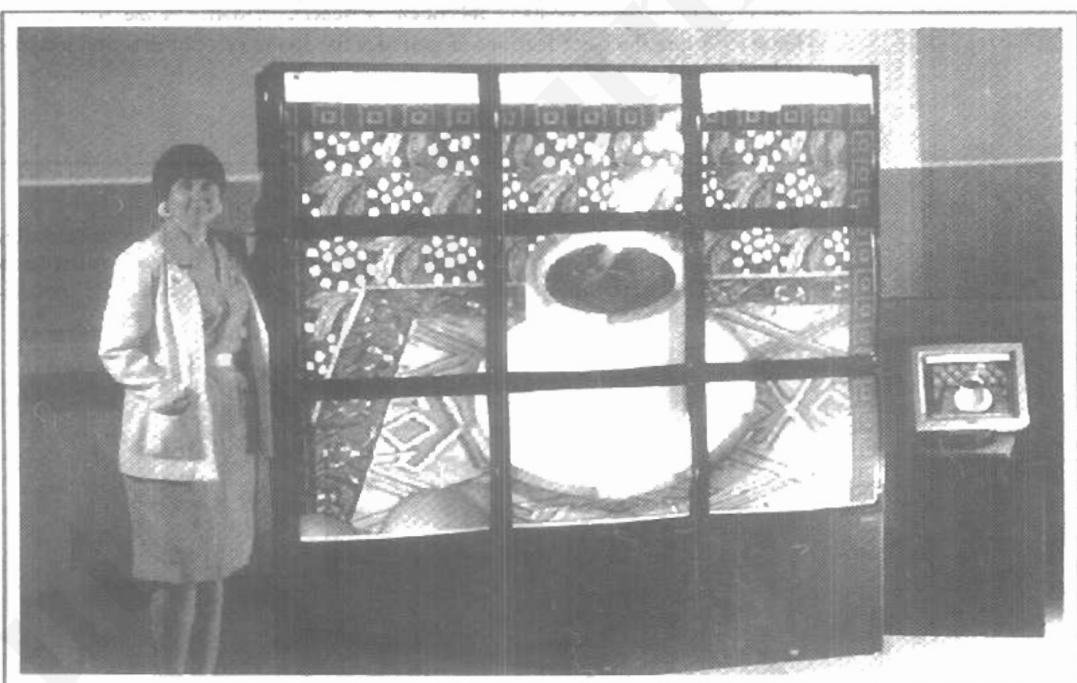
- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

CHAPTER

# 2

## Overview of Graphics Systems



Due to the widespread recognition of the power and utility of computer graphics in virtually all fields, a broad range of graphics hardware and software systems is now available. Graphics capabilities for both two-dimensional and three-dimensional applications are now common on general-purpose computers, including many hand-held calculators. With personal computers, we can use a wide variety of interactive input devices and graphics software packages. For higher-quality applications, we can choose from a number of sophisticated special-purpose graphics hardware systems and technologies. In this chapter, we explore the basic features of graphics hardware components and graphics software packages.

## **2-1** VIDEO DISPLAY DEVICES

Typically, the primary output device in a graphics system is a video monitor (Fig. 2-1). The operation of most video monitors is based on the standard cathode-ray tube (CRT) design, but several other technologies exist and solid-state monitors may eventually predominate.



Figure 2-1  
A computer graphics workstation. (Courtesy of Tektronix, Inc.)

## Refresh Cathode-Ray Tubes

### Section 2-1

#### Video Display Devices

Figure 2-2 illustrates the basic operation of a CRT. A beam of electrons (cathode rays), emitted by an electron gun, passes through focusing and deflection systems that direct the beam toward specified positions on the phosphor-coated screen. The phosphor then emits a small spot of light at each position contacted by the electron beam. Because the light emitted by the phosphor fades very rapidly, some method is needed for maintaining the screen picture. One way to keep the phosphor glowing is to redraw the picture repeatedly by quickly directing the electron beam back over the same points. This type of display is called a refresh CRT.

The primary components of an electron gun in a CRT are the heated metal cathode and a control grid (Fig. 2-3). Heat is supplied to the cathode by directing a current through a coil of wire, called the filament, inside the cylindrical cathode structure. This causes electrons to be "boiled off" the hot cathode surface. In the vacuum inside the CRT envelope, the free, negatively charged electrons are then accelerated toward the phosphor coating by a high positive voltage. The acceler-

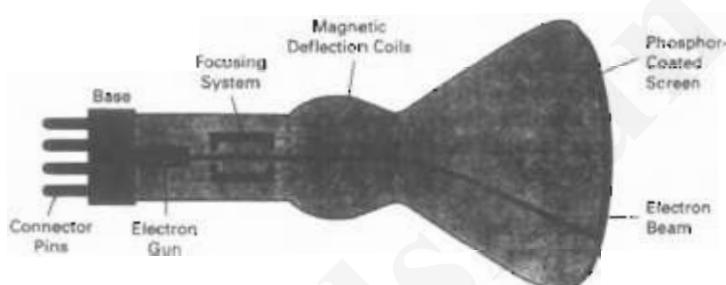


Figure 2-2  
Basic design of a magnetic-deflection CRT.

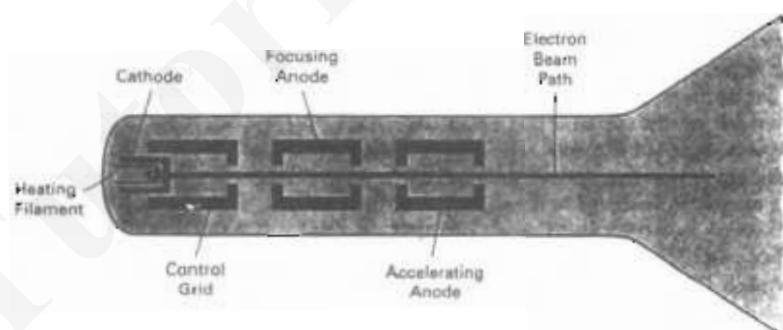


Figure 2-3  
Operation of an electron gun with an accelerating anode.

ating voltage can be generated with a positively charged metal coating on the inside of the CRT envelope near the phosphor screen, or an accelerating anode can be used, as in Fig. 2-3. Sometimes the electron gun is built to contain the accelerating anode and focusing system within the same unit.

Intensity of the electron beam is controlled by setting voltage levels on the control grid, which is a metal cylinder that fits over the cathode. A high negative voltage applied to the control grid will shut off the beam by repelling electrons and stopping them from passing through the small hole at the end of the control grid structure. A smaller negative voltage on the control grid simply decreases the number of electrons passing through. Since the amount of light emitted by the phosphor coating depends on the number of electrons striking the screen, we control the brightness of a display by varying the voltage on the control grid. We specify the intensity level for individual screen positions with graphics software commands, as discussed in Chapter 3.

The focusing system in a CRT is needed to force the electron beam to converge into a small spot as it strikes the phosphor. Otherwise, the electrons would repel each other, and the beam would spread out as it approaches the screen. Focusing is accomplished with either electric or magnetic fields. Electrostatic focusing is commonly used in television and computer graphics monitors. With electrostatic focusing, the electron beam passes through a positively charged metal cylinder that forms an electrostatic lens, as shown in Fig. 2-3. The action of the electrostatic lens focuses the electron beam at the center of the screen, in exactly the same way that an optical lens focuses a beam of light at a particular focal distance. Similar lens focusing effects can be accomplished with a magnetic field set up by a coil mounted around the outside of the CRT envelope. Magnetic lens focusing produces the smallest spot size on the screen and is used in special-purpose devices.

Additional focusing hardware is used in high-precision systems to keep the beam in focus at all screen positions. The distance that the electron beam must travel to different points on the screen varies because the radius of curvature for most CRTs is greater than the distance from the focusing system to the screen center. Therefore, the electron beam will be focused properly only at the center of the screen. As the beam moves to the outer edges of the screen, displayed images become blurred. To compensate for this, the system can adjust the focusing according to the screen position of the beam.

As with focusing, deflection of the electron beam can be controlled either with electric fields or with magnetic fields. Cathode-ray tubes are now commonly constructed with magnetic deflection coils mounted on the outside of the CRT envelope, as illustrated in Fig. 2-2. Two pairs of coils are used, with the coils in each pair mounted on opposite sides of the neck of the CRT envelope. One pair is mounted on the top and bottom of the neck, and the other pair is mounted on opposite sides of the neck. The magnetic field produced by each pair of coils results in a transverse deflection force that is perpendicular both to the direction of the magnetic field and to the direction of travel of the electron beam. Horizontal deflection is accomplished with one pair of coils, and vertical deflection by the other pair. The proper deflection amounts are attained by adjusting the current through the coils. When electrostatic deflection is used, two pairs of parallel plates are mounted inside the CRT envelope. One pair of plates is mounted horizontally to control the vertical deflection, and the other pair is mounted vertically to control horizontal deflection (Fig. 2-4).

Spots of light are produced on the screen by the transfer of the CRT beam energy to the phosphor. When the electrons in the beam collide with the phos-

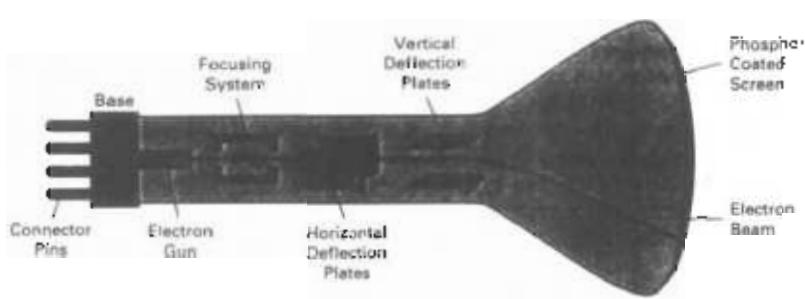


Figure 2-4  
Electrostatic deflection of the electron beam in a CRT.

phosphor coating, they are stopped and their kinetic energy is absorbed by the phosphor. Part of the beam energy is converted by friction into heat energy, and the remainder causes electrons in the phosphor atoms to move up to higher quantum-energy levels. After a short time, the "excited" phosphor electrons begin dropping back to their stable ground state, giving up their extra energy as small quanta of light energy. What we see on the screen is the combined effect of all the electron light emissions: a glowing spot that quickly fades after all the excited phosphor electrons have returned to their ground energy level. The frequency (or color) of the light emitted by the phosphor is proportional to the energy difference between the excited quantum state and the ground state.

Different kinds of phosphors are available for use in a CRT. Besides color, a major difference between phosphors is their persistence: how long they continue to emit light (that is, have excited electrons returning to the ground state) after the CRT 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 without flicker. A phosphor with low persistence is useful for animation; a high-persistence phosphor is useful for displaying highly complex, static pictures. Although some phosphors have a persistence greater than 1 second, graphics monitors are usually constructed with a persistence in the range from 10 to 60 microseconds.

Figure 2-5 shows the intensity distribution of a spot on the screen. The intensity is greatest at the center of the spot, and decreases with a Gaussian distribution out to the edges of the spot. This distribution corresponds to the cross-sectional electron density distribution of the CRT beam.

The maximum number of points that can be displayed without overlap on a CRT is referred to as the resolution. A more precise definition of resolution is the number of points per centimeter that can be plotted horizontally and vertically, although it is often simply stated as the total number of points in each direction. Spot intensity has a Gaussian distribution (Fig. 2-5), so two adjacent spots will appear distinct as long as their separation is greater than the diameter at which each spot has an intensity of about 60 percent of that at the center of the spot. This overlap position is illustrated in Fig. 2-6. Spot size also depends on intensity. As more electrons are accelerated toward the phosphor per second, the CRT beam diameter and the illuminated spot increase. In addition, the increased excitation energy tends to spread to neighboring phosphor atoms not directly in the



Figure 2-5  
Intensity distribution of an illuminated phosphor spot on a CRT screen



**Figure 2-6**  
Two illuminated phosphor spots are distinguishable when their separation is greater than the diameter at which a spot intensity has fallen to 60 percent of maximum.

path of the beam, which further increases the spot diameter. Thus, resolution of a CRT is dependent on the type of phosphor, the intensity to be displayed, and the focusing and deflection systems. Typical resolution on high-quality systems is 1280 by 1024, with higher resolutions available on many systems. High-resolution systems are often referred to as *high-definition systems*. The physical size of a graphics monitor is given as the length of the screen diagonal, with sizes varying from about 12 inches to 27 inches or more. A CRT monitor can be attached to a variety of computer systems, so the number of screen points that can actually be plotted depends on the capabilities of the system to which it is attached.

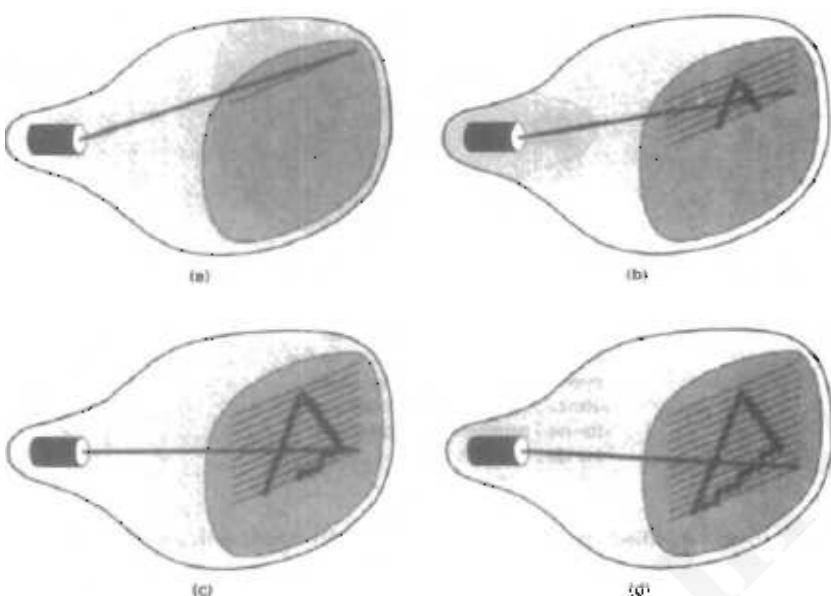
Another property of video monitors is **aspect ratio**. This number gives the ratio of vertical points to horizontal points necessary to produce equal-length lines in both directions on the screen. (Sometimes aspect ratio is stated in terms of the ratio of horizontal to vertical points.) An aspect ratio of 3/4 means that a vertical line plotted with three points has the same length as a horizontal line plotted with four points.

#### Raster-Scan Displays

The most common type of graphics monitor employing a CRT is the raster-scan display, based on television technology. 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 a 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 (Fig. 2-7). Each screen point is referred to as a **pixel** or **pel** (shortened forms of **picture element**). The capability of a raster scan system to store intensity information for each screen point makes it well suited for the realistic display of scenes containing subtle shading and color patterns. Home television sets and printers are examples of other systems using raster-scan methods.

Intensity range for pixel positions depends on the capability of the raster system. In a simple black-and-white system, each screen point is either on or off, so only one bit per pixel is needed to control the intensity of screen positions. For a bilevel system, a bit value of 1 indicates that the electron beam is to be turned on at that position, and a value of 0 indicates that the beam intensity is to be off. Additional bits are needed when color and intensity variations can be displayed. Up to 24 bits per pixel are included in high-quality systems, which can require several megabytes of storage for the frame buffer, depending on the resolution of the system. A system with 24 bits per pixel and a screen resolution of 1024 by 1024 requires 3 megabytes of storage for the frame buffer. On a black-and-white system with one bit per pixel, the frame buffer is commonly called a **bitmap**. For systems with multiple bits per pixel, the frame buffer is often referred to as a  **pixmap**.

Refreshing on raster-scan displays is carried out at the rate of 60 to 80 frames per second, although some systems are designed for higher refresh rates. Sometimes, refresh rates are described in units of cycles per second, or **Hertz (Hz)**, where a cycle corresponds to one frame. Using these units, we would describe a refresh rate of 60 frames per second as simply 60 Hz. 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. The return to the left of the screen, after refreshing each



**Figure 2-7**  
A raster-scan system displays an object as a set of discrete points across each scan line.

scan line, is called the **horizontal retrace** of the electron beam. And at the end of each frame (displayed in 1/80th to 1/60th of a second), the electron beam returns (**vertical retrace**) to the top left corner of the screen to begin the next frame.

On some raster-scan systems (and in TV sets), each frame is displayed in two passes using an *interlaced* refresh procedure. In the first pass, the beam sweeps across every other scan line from top to bottom. Then after the vertical retrace, the beam sweeps out the remaining scan lines (Fig. 2-8). Interlacing of the scan lines in this way allows us to see the entire screen displayed in one-half the time it would have taken to sweep across all the lines at once from top to bottom. Interlacing is primarily used with slower refreshing rates. On an older, 30 frames-per-second, noninterlaced display, for instance, some flicker is noticeable. But with interlacing, each of the two passes can be accomplished in 1/60th of a second, which brings the refresh rate nearer to 60 frames per second. This is an effective technique for avoiding flicker, providing that adjacent scan lines contain similar display information.

#### Random-Scan Displays

When operated as a **random-scan** display unit, a CRT has the electron beam directed only to the parts of the screen where a picture is to be drawn. Random-scan monitors draw a picture one line at a time and for this reason are also referred to as **vector displays** (or **stroke-writing** or **calligraphic** displays). The component lines of a picture can be drawn and refreshed by a random-scan sys-

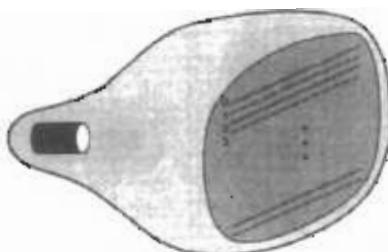


Figure 2-8

Interlacing scan lines on a raster-scan display. First, all points on the even-numbered (solid) scan lines are displayed; then all points along the odd-numbered (dashed) lines are displayed.

tem in any specified order (Fig. 2-9). A pen plotter operates in a similar way and is an example of a random-scan, hard-copy device.

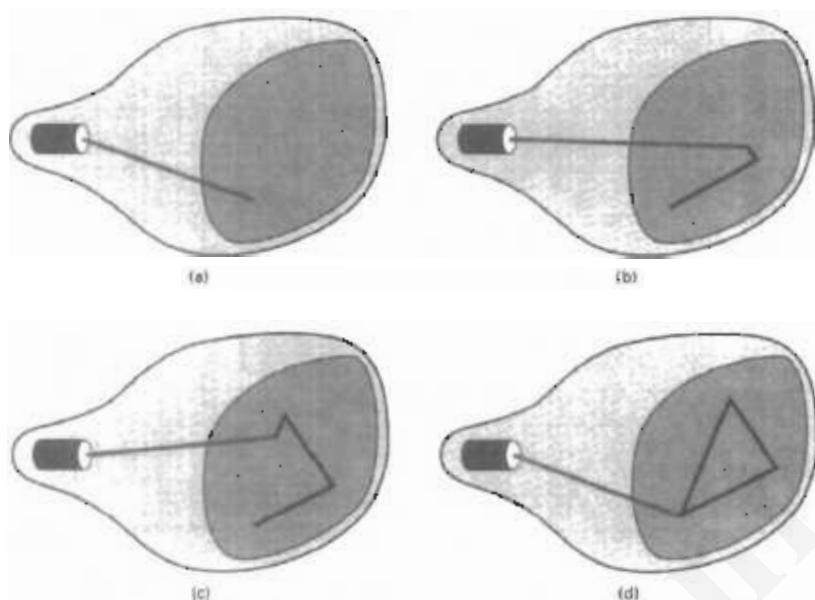
Refresh rate on a random-scan system depends on the number of lines to be displayed. Picture definition is now stored as a set of line-drawing commands in an area of memory referred to as the **refresh display file**. Sometimes the refresh display file is called the **display list**, **display program**, or simply the **refresh buffer**. To display a specified picture, the system cycles through the set of commands in the display file, drawing each component line in turn. After all line-drawing commands have been processed, the system cycles back to the first line command in the list. Random-scan displays are designed to draw all the component lines of a picture 30 to 60 times each second. High-quality vector systems are capable of handling approximately 100,000 "short" lines at this refresh rate. When a small set of lines is to be displayed, each refresh cycle is delayed to avoid refresh rates greater than 60 frames per second. Otherwise, faster refreshing of the set of lines could burn out the phosphor.

Random-scan systems are designed for line-drawing applications and cannot display realistic shaded scenes. Since picture definition is stored as a set of line-drawing instructions and not as a set of intensity values for all screen points, vector displays generally have higher resolution than raster systems. Also, vector displays produce smooth line drawings because the CRT beam directly follows the line path. A raster system, in contrast, produces jagged lines that are plotted as discrete point sets.

### 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. The two basic techniques for producing color displays with a CRT are the **beam-penetration method** and the **shadow-mask method**.

The **beam-penetration method** for displaying color pictures has been used with random-scan monitors. Two layers of phosphor, usually red and green, are



**Figure 2-9**  
A random-scan system draws the component lines of an object in any order specified

coated onto the inside of the CRT screen, and the displayed color depends on how far the electron beam penetrates into the phosphor layers. A beam of slow electrons excites only the outer red layer. A beam of very fast electrons penetrates through the red layer and excites the inner green layer. At intermediate beam speeds, combinations of red and green light are emitted to show two additional colors, orange and yellow. The speed of the electrons, and hence the screen color at any point, is controlled by the beam-acceleration voltage. Beam penetration has been an inexpensive way to produce color in random-scan monitors, but only four colors are possible, and the quality of pictures is not as good as with other methods.

Shadow-mask methods are commonly used in raster-scan systems (including color TV) because they produce a much wider range of colors than the beam-penetration method. A shadow-mask CRT has three phosphor color dots at each pixel position. One phosphor dot emits a red light, another emits a green light, and the third emits a blue light. This type of CRT has three electron guns, one for each color dot, and a shadow-mask grid just behind the phosphor-coated screen. Figure 2-10 illustrates the delta-delta shadow-mask method, commonly used in color CRT systems. The three electron beams are deflected and focused as a group onto the shadow mask, which contains a series of holes aligned with the phosphor-dot patterns. When the three beams pass through a hole in the shadow mask, they activate a dot triangle, which appears as a small color spot on the screen. The phosphor dots in the triangles are arranged so that each electron beam can activate only its corresponding color dot when it passes through the

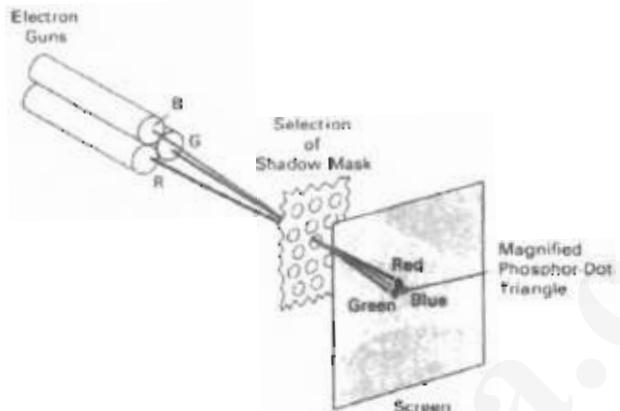


Figure 2-10

Operation of a delta-delta, shadow-mask CRT. Three electron guns, aligned with the triangular color-dot patterns on the screen, are directed to each dot triangle by a shadow mask.

shadow mask. Another configuration for the three electron guns is an *in-line* arrangement in which the three electron guns, and the corresponding red-green-blue color dots on the screen, are aligned along one scan line instead of in a triangular pattern. This in-line arrangement of electron guns is easier to keep in alignment and is commonly used in high-resolution color CRTs.

We obtain color variations in a shadow-mask CRT by varying the intensity levels of the three electron beams. By turning off the red and green guns, we get only the color coming from the blue phosphor. Other combinations of beam intensities produce a small light spot for each pixel position, since our eyes tend to merge the three colors into one composite. The color we see depends on the amount of excitation of the red, green, and blue phosphors. A white (or gray) area is the result of activating all three dots with equal intensity. Yellow is produced with the green and red dots only, magenta is produced with the blue and red dots, and cyan shows up when blue and green are activated equally. In some low-cost systems, the electron beam can only be set to on or off, limiting displays to eight colors. More sophisticated systems can set intermediate intensity levels for the electron beams, allowing several million different colors to be generated.

Color graphics systems can be designed to be used with several types of CRT display devices. Some inexpensive home-computer systems and video games are designed for use with a color TV set and an RF (radio-frequency) modulator. The purpose of the RF modulator is to simulate the signal from a broadcast TV station. This means that the color and intensity information of the picture must be combined and superimposed on the broadcast-frequency carrier signal that the TV needs to have as input. Then the circuitry in the TV takes this signal from the RF modulator, extracts the picture information, and paints it on the screen. As we might expect, this extra handling of the picture information by the RF modulator and TV circuitry decreases the quality of displayed images.

Composite monitors are adaptations of TV sets that allow bypass of the broadcast circuitry. These display devices still require that the picture informa-

tion be combined, but no carrier signal is needed. Picture information is combined into a composite signal and then separated by the monitor, so the resulting picture quality is still not the best attainable.

#### Section 2-1

##### Video Display Devices

Color CRTs in graphics systems are designed as **RGB monitors**. These monitors use shadow-mask methods and take the intensity level for each electron gun (red, green, and blue) directly from the computer system without any intermediate processing. High-quality raster-graphics systems have 24 bits per pixel in the frame buffer, allowing 256 voltage settings for each electron gun and nearly 17 million color choices for each pixel. An RGB color system with 24 bits of storage per pixel is generally referred to as a **full-color system** or a **true-color system**.

#### Direct-View Storage Tubes

An alternative method for maintaining a screen image is to store the picture information inside the CRT instead of refreshing the screen. A **direct-view storage tube (DVST)** stores the picture information as a charge distribution just behind the phosphor-coated screen. Two electron guns are used in a DVST. One, the primary gun, is used to store the picture pattern; the second, the flood gun, maintains the picture display.

A DVST monitor has both disadvantages and advantages compared to the refresh CRT. Because no refreshing is needed, very complex pictures can be displayed at very high resolutions without flicker. Disadvantages of DVST systems are that they ordinarily do not display color and that selected parts of a picture cannot be erased. To eliminate a picture section, the entire screen must be erased and the modified picture redrawn. The erasing and redrawing process can take several seconds for a complex picture. For these reasons, storage displays have been largely replaced by raster systems.

#### Flat-Panel Displays

Although most graphics monitors are still constructed with CRTs, other technologies are emerging that may soon replace CRT monitors. The term **flat-panel display** refers to a class of video devices that have reduced volume, weight, and power requirements compared to a CRT. A significant feature of flat-panel displays is that they are thinner than CRTs, and we can hang them on walls or wear them on our wrists. Since we can even write on some flat-panel displays, they will soon be available as pocket notepads. Current uses for flat-panel displays include small TV monitors, calculators, pocket video games, laptop computers, armrest viewing of movies on airlines, as advertisement boards in elevators, and as graphics displays in applications requiring rugged, portable monitors.

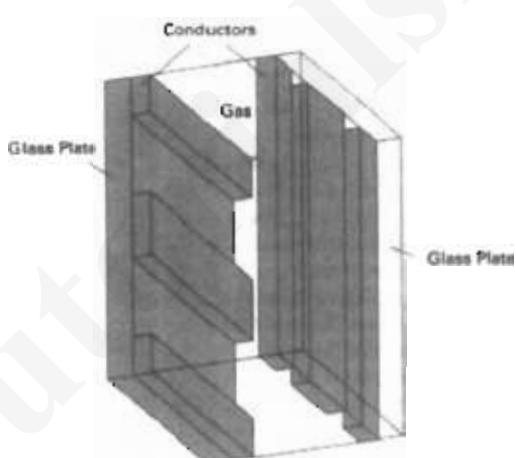
We can separate flat panel displays into two categories: **emissive displays** and **non emissive displays**. The emissive displays (or emitters) are devices that convert electrical energy into light. Plasma panels, thin-film electroluminescent displays, and light-emitting diodes are examples of emissive displays. Flat CRTs have also been devised, in which electron beams are accelerated parallel to the screen, then deflected 90° to the screen. But flat CRTs have not proved to be as successful as other emissive devices. Non emissive displays (or non emitters) use optical effects to convert sunlight or light from some other source into graphics patterns. The most important example of a non emissive flat-panel display is a liquid-crystal device.

Plasma panels, also called gas-discharge displays, are constructed by filling the region between two glass plates with a mixture of gases that usually in

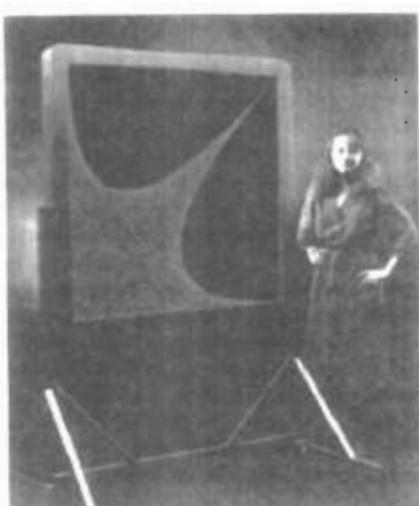
cludes neon. A series of vertical conducting ribbons is placed on one glass panel, and a set of horizontal ribbons is built into the other glass panel (Fig. 2-11). Firing voltages applied to a pair of horizontal and vertical conductors cause the gas at the intersection of the two conductors to break down into a glowing plasma of electrons and ions. Picture definition is stored in a refresh buffer, and the firing voltages are applied to refresh the pixel positions (at the intersections of the conductors) 60 times per second. Alternating-current methods are used to provide faster application of the firing voltages, and thus brighter displays. Separation between pixels is provided by the electric field of the conductors. Figure 2-12 shows a high-definition plasma panel. One disadvantage of plasma panels has been that they were strictly monochromatic devices, but systems have been developed that are now capable of displaying color and grayscale.

**Thin-film electroluminescent displays** are similar in construction to a plasma panel. The difference is that the region between the glass plates is filled with a phosphor, such as zinc sulfide doped with manganese, instead of a gas (Fig. 2-13). When a sufficiently high voltage is applied to a pair of crossing electrodes, the phosphor becomes a conductor in the area of the intersection of the two electrodes. Electrical energy is then absorbed by the manganese atoms, which then release the energy as a spot of light similar to the glowing plasma effect in a plasma panel. Electroluminescent displays require more power than plasma panels, and good color and gray scale displays are hard to achieve.

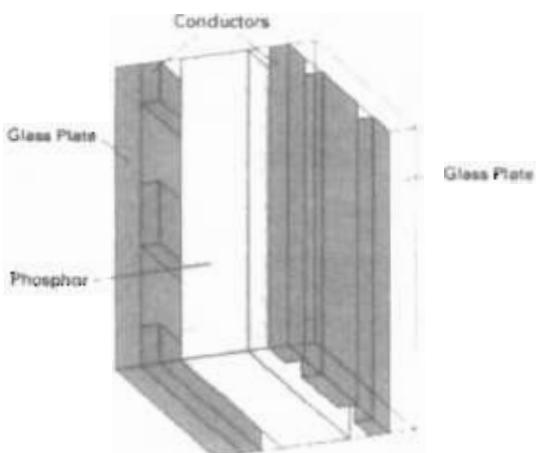
A third type of emissive device is the **light-emitting diode (LED)**. A matrix of diodes is arranged to form the pixel positions in the display, and picture definition is stored in a refresh buffer. As in scan-line refreshing of a CRT, information



**Figure 2-11**  
Basic design of a plasma-panel display device.



**Figure 2-12**  
A plasma-panel display with a resolution of 2048 by 2048 and a screen diagonal of 1.5 meters.  
(Courtesy of Photonics Systems.)



Section 2-1  
Video Display Devices

Figure 2-13  
Basic design of a thin-film electroluminescent display device.

is read from the refresh buffer and converted to voltage levels that are applied to the diodes to produce the light patterns in the display.

Liquid-crystal displays (LCDs) are commonly used in small systems, such as calculators (Fig. 2-14) and portable, laptop computers (Fig. 2-15). These non-emissive devices produce a picture by passing polarized light from the surroundings or from an internal light source through a liquid-crystal material that can be aligned to either block or transmit the light.

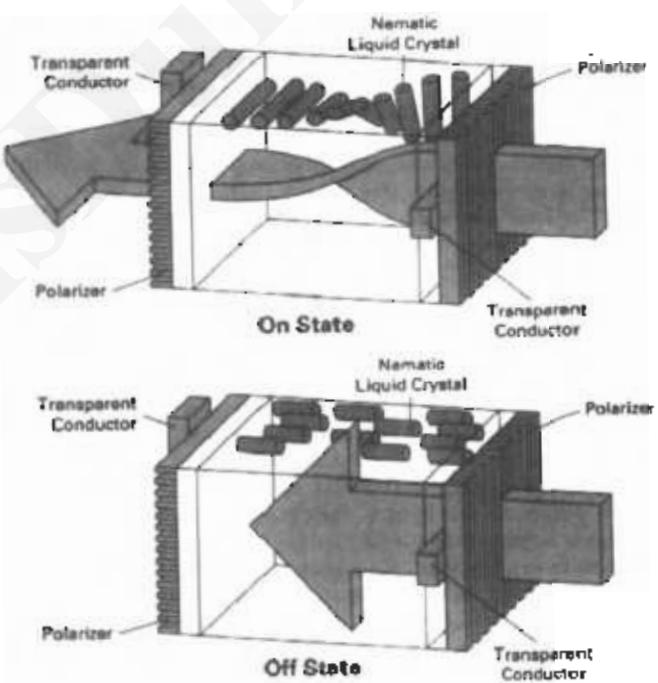
The term *liquid crystal* refers to the fact that these compounds have a crystalline arrangement of molecules, yet they flow like a liquid. Flat-panel displays commonly use nematic (threadlike) liquid-crystal compounds that tend to keep the long axes of the rod-shaped molecules aligned. A flat-panel display can then be constructed with a nematic liquid crystal, as demonstrated in Fig. 2-16. Two glass plates, each containing a light polarizer at right angles to the other plate, sandwich the liquid-crystal material. Rows of horizontal transparent conductors are built into one glass plate, and columns of vertical conductors are put into the other plate. The intersection of two conductors defines a pixel position. Normally, the molecules are aligned as shown in the "on state" of Fig. 2-16. Polarized light passing through the material is twisted so that it will pass through the opposite polarizer. The light is then reflected back to the viewer. To turn off the pixel, we apply a voltage to the two intersecting conductors to align the molecules so that the light is not twisted. This type of flat-panel device is referred to as a **passive-matrix LCD**. Picture definitions are stored in a refresh buffer, and the screen is refreshed at the rate of 60 frames per second, as in the emissive devices. Back lighting is also commonly applied using solid-state electronic devices, so that the system is not completely dependent on outside light sources. Colors can be displayed by using different materials or dyes and by placing a triad of color pixels at each screen location. Another method for constructing LCDs is to place a transistor at each pixel location, using thin-film transistor technology. The transistors are used to control the voltage at pixel locations and to prevent charge from gradually leaking out of the liquid-crystal cells. These devices are called **active-matrix displays**.



Figure 2-14  
A hand calculator with an LCD screen. (Courtesy of Texas Instruments.)



**Figure 2-15**  
A backlit, passive-matrix, liquid-crystal display in a laptop computer, featuring 256 colors, a screen resolution of 640 by 400, and a screen diagonal of 9 inches.  
(Courtesy of Apple Computer, Inc.)



**Figure 2-16**  
The light-twisting, shutter effect used in the design of most liquid-crystal display devices.

### Three-Dimensional Viewing Devices

Graphics monitors for the display of three-dimensional scenes have been devised using a technique that reflects a CRT image from a vibrating, flexible mirror. The operation of such a system is demonstrated in Fig. 2-17. As the varifocal mirror vibrates, it changes focal length. These vibrations are synchronized with the display of an object on a CRT so that each point on the object is reflected from the mirror into a spatial position corresponding to the distance of that point from a specified viewing position. This allows us to walk around an object or scene and view it from different sides.

Figure 2-18 shows the Genisco SpaceGraph system, which uses a vibrating mirror to project three-dimensional objects into a 25-cm by 25-cm by 25-cm volume. This system is also capable of displaying two-dimensional cross-sectional "slices" of objects selected at different depths. Such systems have been used in medical applications to analyze data from ultrasonography and CAT scan devices, in geological applications to analyze topological and seismic data, in design applications involving solid objects, and in three-dimensional simulations of systems, such as molecules and terrain.

### Section 2-1

#### Video Display Devices

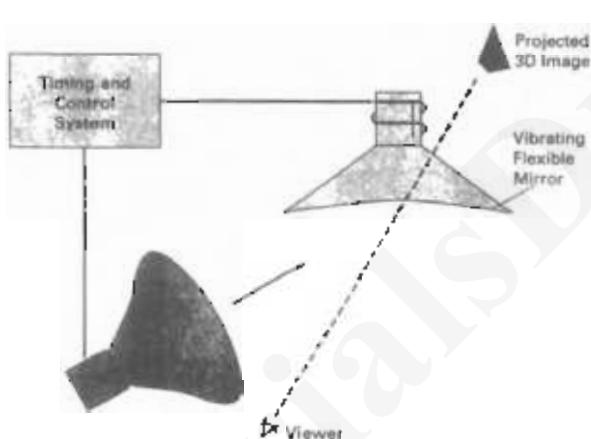


Figure 2-17

Operation of a three-dimensional display system using a vibrating mirror that changes focal length to match the depth of points in a scene.



Figure 2-18

The SpaceGraph interactive graphics system displays objects in three dimensions using a vibrating, flexible mirror. (Courtesy of Genisco Computer Corporation)

### Stereoscopic and Virtual-Reality Systems

Another technique for representing three-dimensional objects is displaying stereoscopic views. This method does not produce true three-dimensional images, but it does provide a three-dimensional effect by presenting a different view to each eye of an observer so that scenes do appear to have depth (Fig. 2-19).

To obtain a stereoscopic projection, we first need to obtain two views of a scene generated from a viewing direction corresponding to each eye (left and right). We can construct the two views as computer-generated scenes with different viewing positions, or we can use a stereo camera pair to photograph some object or scene. When we simultaneous look at the left view with the left eye and the right view with the right eye, the two views merge into a single image and we perceive a scene with depth. Figure 2-20 shows two views of a computer-generated scene for stereographic projection. To increase viewing comfort, the areas at the left and right edges of this scene that are visible to only one eye have been eliminated.



Figure 2-19  
Viewing a stereoscopic projection.  
(Courtesy of StereoGraphics Corporation.)

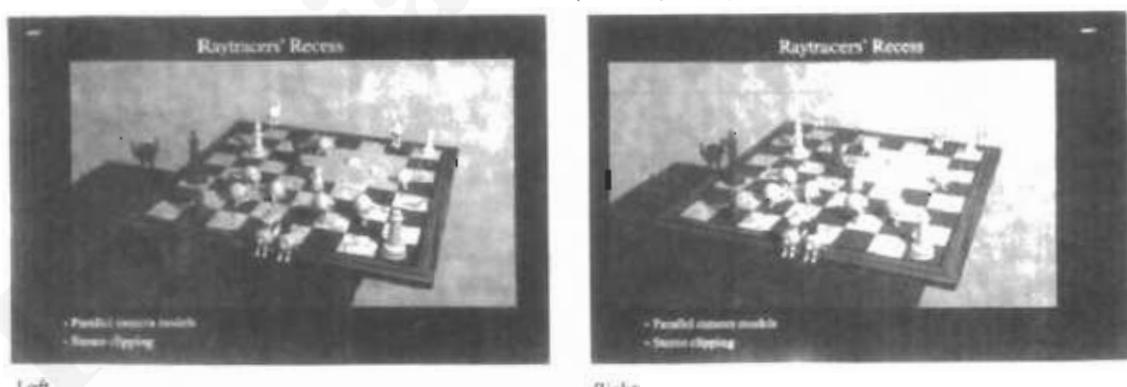


Figure 2-20

A stereoscopic viewing pair. (Courtesy of Jerry Fife.)

One way to produce a stereoscopic effect is to display each of the two views with a raster system on alternate refresh cycles. The screen is viewed through glasses, with each lens designed to act as a rapidly alternating shutter that is synchronized to block out one of the views. Figure 2-21 shows a pair of stereoscopic glasses constructed with liquid-crystal shutters and an infrared emitter that synchronizes the glasses with the views on the screen.

Stereoscopic viewing is also a component in virtual-reality systems, where users can step into a scene and interact with the environment. A headset (Fig. 2-22) containing an optical system to generate the stereoscopic views is commonly used in conjunction with interactive input devices to locate and manipulate objects in the scene. A sensing system in the headset keeps track of the viewer's position, so that the front and back of objects can be seen as the viewer

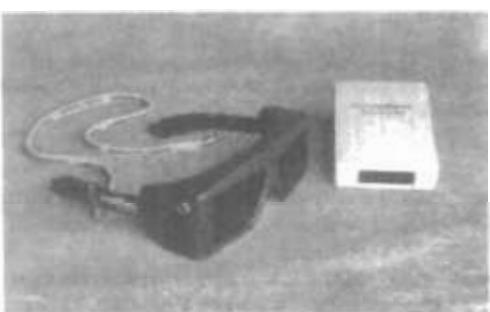


Figure 2-21  
Glasses for viewing a  
stereoscopic scene and an  
infrared synchronizing emitter  
(Courtesy of StereoGraphics Corporation.)



Figure 2-22  
A headset used in virtual-reality systems. (Courtesy of Virtual Research.)



Figure 2-23

Interacting with a virtual-reality environment. (Courtesy of the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.)

"walks through" and interacts with the display. Figure 2-23 illustrates interaction with a virtual scene, using a headset and a data glove worn on the right hand (Section 2-5).

An interactive virtual-reality environment can also be viewed with stereoscopic glasses and a video monitor, instead of a headset. This provides a means for obtaining a lower-cost virtual-reality system. As an example, Fig. 2-24 shows an ultrasound tracking device with six degrees of freedom. The tracking device is placed on top of the video display and is used to monitor head movements so that the viewing position for a scene can be changed as head position changes.

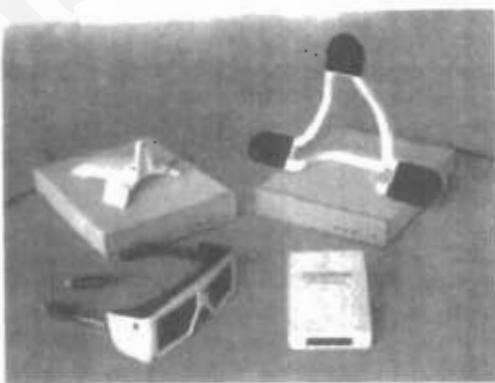


Figure 2-24

An ultrasound tracking device used with stereoscopic glasses to track head position. (Courtesy of Stereographics Corporation.)

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

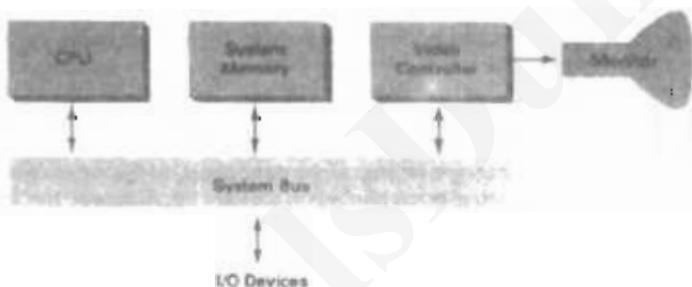
**2-2****RASTER-SCAN SYSTEMS****Section 2-2****Raster-Scan Systems**

Interactive raster graphics systems typically employ several processing units. In addition to the central processing unit, or CPU, a special-purpose processor, called the **video controller** or **display controller**, is used to control the operation of the display device. Organization of a simple raster system is shown in Fig. 2-25. Here, the frame buffer can be anywhere in the system memory, and the video controller accesses the frame buffer to refresh the screen. In addition to the video controller, more sophisticated raster systems employ other processors as co-processors and accelerators to implement various graphics operations.

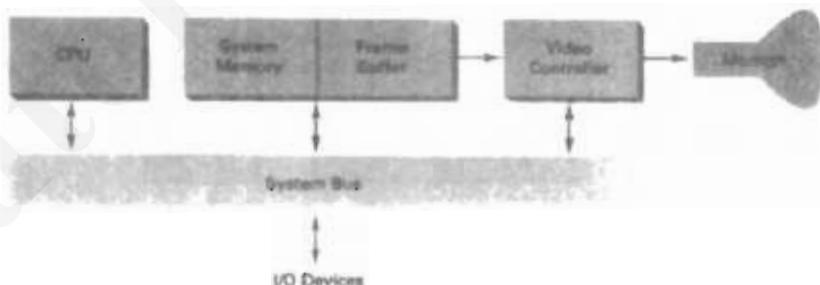
**Video Controller**

Figure 2-26 shows a commonly used organization for raster systems. A fixed area of the system memory is reserved for the frame buffer, and the video controller is given direct access to the frame-buffer memory.

Frame-buffer locations, and the corresponding screen positions, are referenced in Cartesian coordinates. For many graphics monitors, the coordinate ori-



*Figure 2-25*  
Architecture of a simple raster graphics system.



*Figure 2-26*  
Architecture of a raster system with a fixed portion of the system memory reserved for the frame buffer.



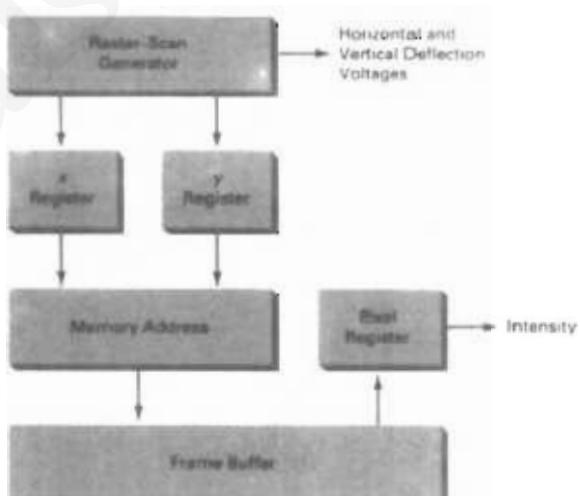
**Figure 2-27**  
The origin of the coordinate system for identifying screen positions is usually specified in the lower-left corner.

gin is defined at the lower left screen corner (Fig. 2-27). The screen surface is then represented as the first quadrant of a two-dimensional system, with positive  $x$  values increasing to the right and positive  $y$  values increasing from bottom to top. (On some personal computers, the coordinate origin is referenced at the upper left corner of the screen, so the  $y$  values are inverted.) Scan lines are then labeled from  $y_{max}$  at the top of the screen to 0 at the bottom. Along each scan line, screen pixel positions are labeled from 0 to  $x_{max}$ .

In Fig. 2-28, the basic refresh operations of the video controller are diagrammed. Two registers are used to store the coordinates of the screen pixels. Initially, the  $x$  register is set to 0 and the  $y$  register is set to  $y_{max}$ . The value stored in the frame buffer for this pixel position is then retrieved and used to set the intensity of the CRT beam. Then the  $x$  register is incremented by 1, and the process repeated for the next pixel on the top scan line. This procedure is repeated for each pixel along the scan line. After the last pixel on the top scan line has been processed, the  $x$  register is reset to 0 and the  $y$  register is decremented by 1. Pixels along this scan line are then processed in turn, and the procedure is repeated for each successive scan line. After cycling through all pixels along the bottom scan line ( $y = 0$ ), the video controller resets the registers to the first pixel position on the top scan line and the refresh process starts over.

Since the screen must be refreshed at the rate of 60 frames per second, the simple procedure illustrated in Fig. 2-28 cannot be accommodated by typical RAM chips. The cycle time is too slow. To speed up pixel processing, video controllers can retrieve multiple pixel values from the refresh buffer on each pass. The multiple pixel intensities are then stored in a separate register and used to control the CRT beam intensity for a group of adjacent pixels. When that group of pixels has been processed, the next block of pixel values is retrieved from the frame buffer.

A number of other operations can be performed by the video controller, besides the basic refreshing operations. For various applications, the video con-



**Figure 2-28**  
Basic video-controller refresh operations.

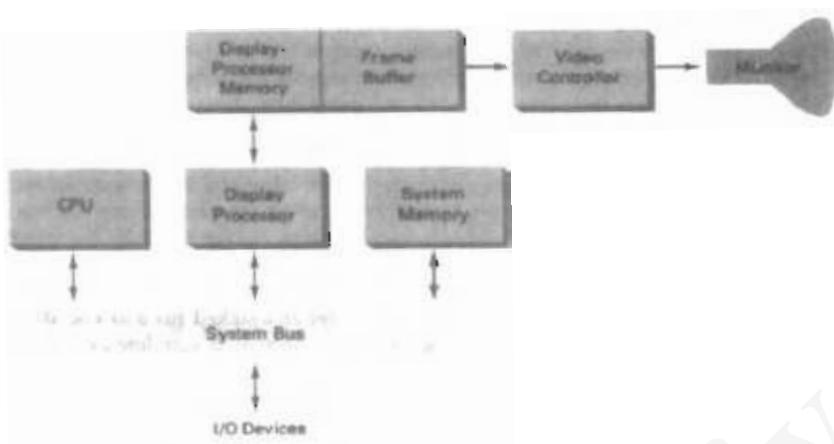


Figure 2-29  
Architecture of a raster-graphics system with a display processor.

troller can retrieve pixel intensities from different memory areas on different refresh cycles. In high-quality systems, for example, two frame buffers are often provided so that one buffer can be used for refreshing, while the other is being filled with intensity values. Then the two buffers can switch roles. This provides a fast mechanism for generating real-time animations, since different views of moving objects can be successively loaded into the refresh buffers. Also, some transformations can be accomplished by the video controller. Areas of the screen can be enlarged, reduced, or moved from one location to another during the refresh cycles. In addition, the video controller often contains a lookup table, so that pixel values in the frame buffer are used to access the lookup table instead of controlling the CRT beam intensity directly. This provides a fast method for changing screen intensity values, and we discuss lookup tables in more detail in Chapter 4. Finally, some systems are designed to allow the video controller to mix the frame-buffer image with an input image from a television camera or other input device.

#### Raster-Scan Display Processor

Figure 2-29 shows one way to set up the organization of a raster system containing a separate display processor, sometimes referred to as a **graphics controller** or a **display coprocessor**. The purpose of the display processor is to free the CPU from the graphics chores. In addition to the system memory, a separate display-processor memory area can also be provided.

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**. Graphics commands specifying straight lines and other geometric objects are scan converted into a set of discrete intensity points. Scan converting a straight-line segment, for example, means that we have to locate the pixel positions closest to the line path and store the intensity for each position in the frame buffer. Similar methods are used for scan converting curved lines and polygon outlines. Characters can be defined with rectangular grids, as in Fig. 2-30, or they can be defined with curved

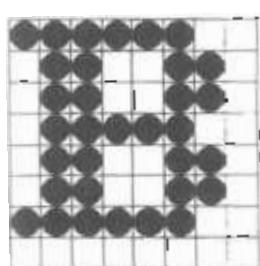


Figure 2-30  
A character defined as a rectangular grid of pixel positions.



Figure 2-31  
A character defined as a curve outline.

outlines, as in Fig. 2-31. The array size for character grids can vary from about 5 by 7 to 9 by 12 or more for higher-quality displays. A character grid is displayed by superimposing the rectangular grid pattern into the frame buffer at a specified coordinate position. With characters that are defined as curve outlines, character shapes are scan converted into the frame buffer.

Display processors are also designed to perform a number of additional operations. These functions include generating various line styles (dashed, dotted, or solid), displaying color areas, and performing certain transformations and manipulations on displayed objects. Also, display processors are typically designed to interface with interactive input devices, such as a mouse.

In an effort to reduce memory requirements in raster systems, methods have been devised for organizing the frame buffer as a linked list and encoding the intensity information. One way to do this is to store each scan line as a set of integer pairs. One number of each pair indicates an intensity value, and the second number specifies the number of adjacent pixels on the scan line that are to have that intensity. This technique, called **run-length encoding**, can result in a considerable saving in storage space if a picture is to be constructed mostly with long runs of a single color each. A similar approach can be taken when pixel intensities change linearly. Another approach is to encode the raster as a set of rectangular areas (cell encoding). The disadvantages of encoding runs are that intensity changes are difficult to make and storage requirements actually increase as the length of the runs decreases. In addition, it is difficult for the display controller to process the raster when many short runs are involved.

## 2-3 RANDOM-SCAN SYSTEMS

The organization of a simple random-scan (vector) system is shown in Fig. 2-32. An application program is input and stored in the system memory along with a graphics package. Graphics commands in the application program are translated by the graphics package into a display file stored in the system memory. This display file is then accessed by the display processor to refresh the screen. The display processor cycles through each command in the display file program once during every refresh cycle. Sometimes the display processor in a random-scan system is referred to as a display processing unit or a graphics controller.

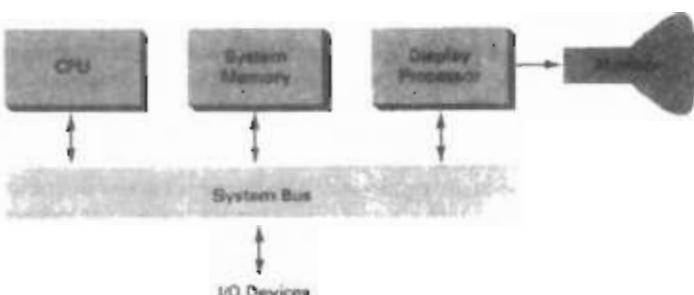


Figure 2-32  
Architecture of a simple random-scan system

Graphics patterns are drawn on a random-scan system by directing the electron beam along the component lines of the picture. Lines are defined by the values for their coordinate endpoints, and these input coordinate values are converted to x and y deflection voltages. A scene is then drawn one line at a time by positioning the beam to fill in the line between specified endpoints.

## 2-4

### GRAPHICS MONITORS AND WORKSTATIONS

Most graphics monitors today operate as raster-scan displays, and here we survey a few of the many graphics hardware configurations available. Graphics systems range from small general-purpose computer systems with graphics capabilities (Fig. 2-33) to sophisticated full-color systems that are designed specifically for graphics applications (Fig. 2-34). A typical screen resolution for personal com-



Figure 2-33  
A desktop general-purpose computer system that can be used for graphics applications. (Courtesy of Apple Computer, Inc.)

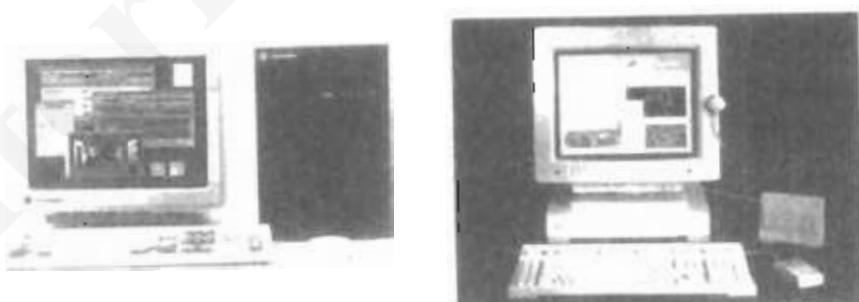


Figure 2-34  
Computer graphics workstations with keyboard and mouse input devices. (a) The Iris Indigo. (Courtesy of Silicon Graphics Corporation.) (b) SPARCstation 10. (Courtesy of Sun Microsystems.)

puter systems, such as the Apple Quadra shown in Fig. 2-33, is 640 by 480, although screen resolution and other system capabilities vary depending on the size and cost of the system. Diagonal screen dimensions for general-purpose personal computer systems can range from 12 to 21 inches, and allowable color selections range from 16 to over 32,000. For workstations specifically designed for graphics applications, such as the systems shown in Fig. 2-34, typical screen resolution is 1280 by 1024, with a screen diagonal of 16 inches or more. Graphics workstations can be configured with from 8 to 24 bits per pixel (full-color systems), with higher screen resolutions, faster processors, and other options available in high-end systems.

Figure 2-35 shows a high-definition graphics monitor used in applications such as air traffic control, simulation, medical imaging, and CAD. This system has a diagonal screen size of 27 inches, resolutions ranging from 2048 by 1536 to 2560 by 2048, with refresh rates of 80 Hz or 60 Hz noninterlaced.

A multiscreen system called the MediaWall, shown in Fig. 2-36, provides a large "wall-sized" display area. This system is designed for applications that require large area displays in brightly lighted environments, such as at trade shows, conventions, retail stores, museums, or passenger terminals. MediaWall operates by splitting images into a number of sections and distributing the sections over an array of monitors or projectors using a graphics adapter and satellite control units. An array of up to 5 by 5 monitors, each with a resolution of 640 by 480, can be used in the MediaWall to provide an overall resolution of 3200 by 2400 for either static scenes or animations. Scenes can be displayed behind mullions, as in Fig. 2-36, or the mullions can be eliminated to display a continuous picture with no breaks between the various sections.

Many graphics workstations, such as some of those shown in Fig. 2-37, are configured with two monitors. One monitor can be used to show all features of an object or scene, while the second monitor displays the detail in some part of the picture. Another use for dual-monitor systems is to view a picture on one monitor and display graphics options (menus) for manipulating the picture components on the other monitor.



**Figure 2-35**  
A very high-resolution (2560 by 2048) color monitor. (Courtesy of BARCO Chromatics.)

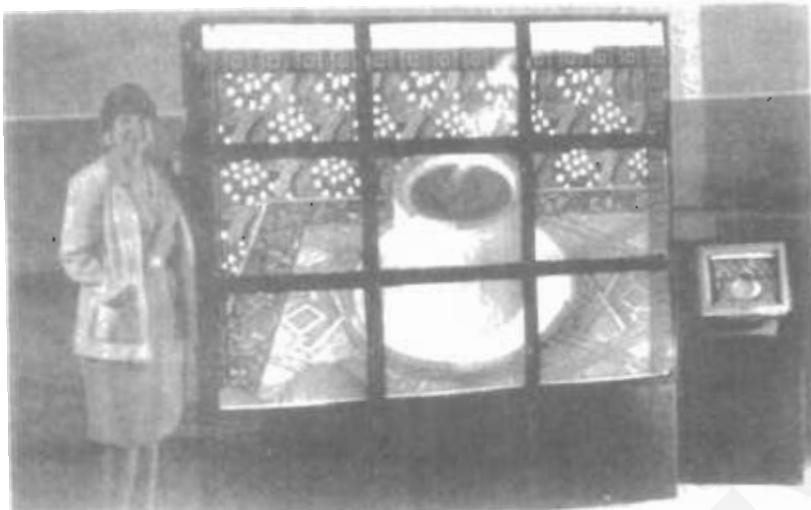


Figure 2-36

The MediaWall: A multiscreen display system. The image displayed on this 3-by-3 array of monitors was created by Deneba Software. (Courtesy of RGB Spectrum.)



Figure 2-37

Single- and dual-monitor graphics workstations. (Courtesy of Intergraph Corporation.)

Figures 2-38 and 2-39 illustrate examples of interactive graphics workstations containing multiple input and other devices. A typical setup for CAD applications is shown in Fig. 2-38. Various keyboards, button boxes, tablets, and mice are attached to the video monitors for use in the design process. Figure 2-39 shows features of some types of artist's workstations.



**Figure 2-38**  
Multiple workstations for a CAD group. (Courtesy of Hewlett-Packard Company.)



**Figure 2-39**  
An artist's workstation, featuring a color raster monitor, keyboard, graphics tablet with hand cursor, and a light table, in addition to data storage and telecommunications devices. (Courtesy of DICOMED Corporation.)

## 2-5 INPUT DEVICES

Various devices are available for data input on graphics workstations. Most systems have a keyboard and one or more additional devices specially designed for interactive input. These include a mouse, trackball, spaceball, joystick, digitizers,

dials, and button boxes. Some other input devices used in particular applications are data gloves, touch panels, image scanners, and voice systems.

### Keyboards

An alphanumeric keyboard on a graphics system is used primarily as a device for entering text strings. The keyboard is an efficient device for inputting such nongraphic data as picture labels associated with a graphics display. Keyboards can also be provided with features to facilitate entry of screen coordinates, menu selections, or graphics functions.

Cursor-control keys and function keys are common features on general-purpose keyboards. Function keys allow users to enter frequently used operations in a single keystroke, and cursor-control keys can be used to select displayed objects or coordinate positions by positioning the screen cursor. Other types of cursor-positioning devices, such as a trackball or joystick, are included on some keyboards. Additionally, a numeric keypad is often included on the keyboard for fast entry of numeric data. Typical examples of general-purpose keyboards are given in Figs. 2-1, 2-33, and 2-34. Fig. 2-40 shows an ergonomic keyboard design.

For specialized applications, input to a graphics application may come from a set of buttons, dials, or switches that select data values or customized graphics operations. Figure 2-41 gives an example of a button box and a set of input dials. Buttons and switches are often used to input predefined functions, and dials are common devices for entering scalar values. Real numbers within some defined range are selected for input with dial rotations. Potentiometers are used to measure dial rotations, which are then converted to deflection voltages for cursor movement.

### Mouse

A mouse is small hand-held box used to position the screen cursor. Wheels or rollers on the bottom of the mouse can be used to record the amount and direc-



Figure 2-40  
Ergonomically designed keyboard with removable palm rests. The slope of each half of the keyboard can be adjusted separately. (Courtesy of Apple Computer, Inc.)

tion of movement. Another method for detecting mouse motion is with an optical sensor. For these systems, the mouse is moved over a special mouse pad that has a grid of horizontal and vertical lines. The optical sensor detects movement across the lines in the grid.

Since a mouse can be picked up and put down at another position without change in cursor movement, it is used for making relative changes in the position of the screen cursor. One, two, or three buttons are usually included on the top of the mouse for signaling the execution of some operation, such as recording cursor position or invoking a function. Most general-purpose graphics systems now include a mouse and a keyboard as the major input devices, as in Figs. 2-1, 2-33, and 2-34.

Additional devices can be included in the basic mouse design to increase the number of allowable input parameters. The Z mouse in Fig. 2-42 includes

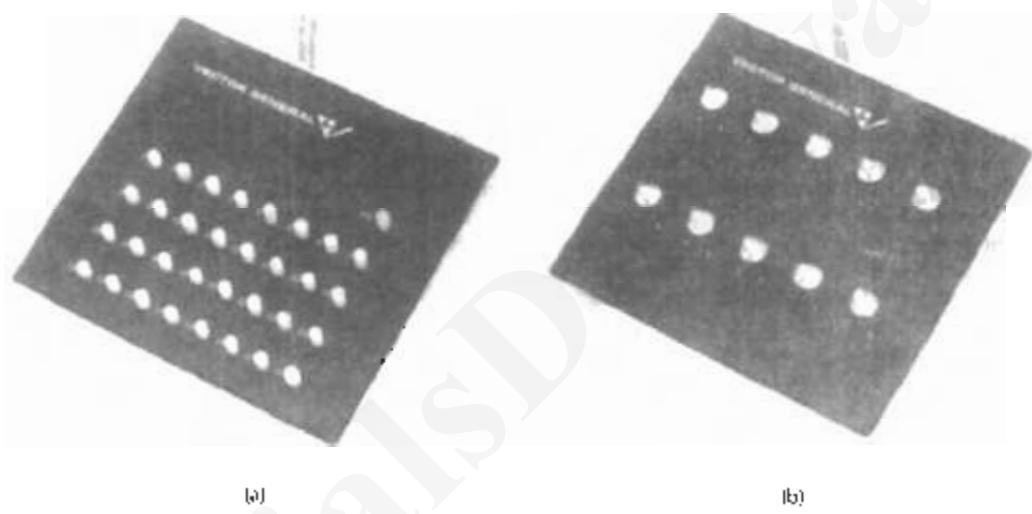


Figure 2-41  
A button box (a) and a set of input dials (b). (Courtesy of Vector Computer.)

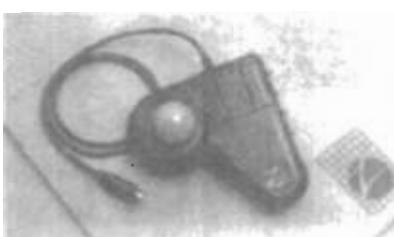


Figure 2-42  
The Z mouse features three buttons, a mouse ball underneath, a thumbwheel on the side, and a trackball on top. (Courtesy of Multipoint Technology Corporation.)

---

## Section 2-5

### Input Devices

three buttons, a thumbwheel on the side, a trackball on the top, and a standard mouse ball underneath. This design provides six degrees of freedom to select spatial positions, rotations, and other parameters. With the Z mouse, we can pick up an object, rotate it, and move it in any direction, or we can navigate our viewing position and orientation through a three-dimensional scene. Applications of the Z mouse include virtual reality, CAD, and animation.

#### Trackball and Spaceball

As the name implies, a trackball is a ball that can be rotated with the fingers or palm of the hand, as in Fig. 2-43, to produce screen-cursor movement. Potentiometers, attached to the ball, measure the amount and direction of rotation. Trackballs are often mounted on keyboards (Fig. 2-15) or other devices such as the Z mouse (Fig. 2-42).

While a trackball is a two-dimensional positioning device, a spaceball (Fig. 2-45) provides six degrees of freedom. Unlike the trackball, a spaceball does not actually move. Strain gauges measure the amount of pressure applied to the spaceball to provide input for spatial positioning and orientation as the ball is pushed or pulled in various directions. Spaceballs are used for three-dimensional positioning and selection operations in virtual-reality systems, modeling, animation, CAD, and other applications.

#### Joysticks

A joystick consists of a small, vertical lever (called the stick) mounted on a base that is used to steer the screen cursor around. Most joysticks select screen positions with actual stick movement; others respond to pressure on the stick. Figure 2-44 shows a movable joystick. Some joysticks are mounted on a keyboard; others function as stand-alone units.

The distance that the stick is moved in any direction from its center position corresponds to screen-cursor movement in that direction. Potentiometers mounted at the base of the joystick measure the amount of movement, and springs return the stick to the center position when it is released. One or more buttons can be programmed to act as input switches to signal certain actions once a screen position has been selected.



Figure 2-43

A three-button track ball. (Courtesy of Measurement Systems Inc., Norwalk, Connecticut.)

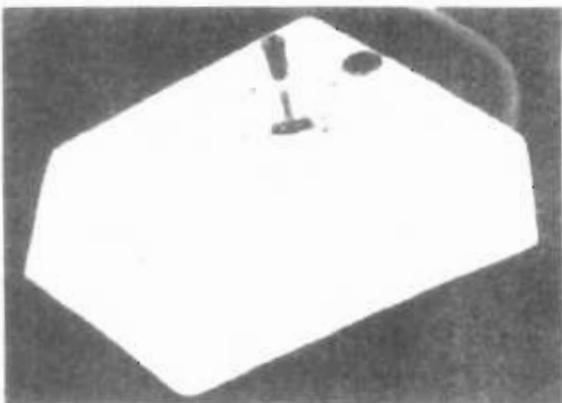


Figure 2-44

A moveable joystick. (Courtesy of CalComp Group, Senders Associates, Inc.)

In another type of movable joystick, the stick is used to activate switches that cause the screen cursor to move at a constant rate in the direction selected. Eight switches, arranged in a circle, are sometimes provided, so that the stick can select any one of eight directions for cursor movement. Pressure-sensitive joysticks, also called isometric joysticks, have a nonmovable stick. Pressure on the stick is measured with strain gauges and converted to movement of the cursor in the direction specified.

#### Data Glove

Figure 2-45 shows a **data glove** that can be used to grasp a "virtual" object. The glove is constructed with a series of sensors that detect hand and finger motions. Electromagnetic coupling between transmitting antennas and receiving antennas is used to provide information about the position and orientation of the hand. The transmitting and receiving antennas can each be structured as a set of three mutually perpendicular coils, forming a three-dimensional Cartesian coordinate system. Input from the glove can be used to position or manipulate objects in a virtual scene. A two-dimensional projection of the scene can be viewed on a video monitor, or a three-dimensional projection can be viewed with a headset.

#### Digitizers

A common device for drawing, painting, or interactively selecting coordinate positions on an object is a **digitizer**. These devices can be used to input coordinate values in either a two-dimensional or a three-dimensional space. Typically, a digitizer is used to scan over a drawing or object and to input a set of discrete coordinate positions, which can be joined with straight-line segments to approximate the curve or surface shapes.

One type of digitizer is the **graphics tablet** (also referred to as a **data tablet**), which is used to input two-dimensional coordinates by activating a hand cursor or stylus at selected positions on a flat surface. A hand cursor contains cross hairs for sighting positions, while a stylus is a pencil-shaped device that is pointed at



**Figure 2-45**  
A virtual-reality scene, displayed on a two-dimensional video monitor, with input from a data glove and a spaceball. (Courtesy of The Computer Graphics Center, Darmstadt, Germany.)

positions on the tablet. Figures 2-46 and 2-47 show examples of desktop and floor-model tablets, using hand cursors that are available with 2, 4, or 16 buttons. Examples of stylus input with a tablet are shown in Figs. 2-48 and 2-49. The artist's digitizing system in Fig. 2-49 uses electromagnetic resonance to detect the three-dimensional position of the stylus. This allows an artist to produce different brush strokes with different pressures on the tablet surface. Tablet size varies from 12 by 12 inches for desktop models to 44 by 60 inches or larger for floor models. Graphics tablets provide a highly accurate method for selecting coordinate positions, with an accuracy that varies from about 0.2 mm on desktop models to about 0.05 mm or less on larger models.

Many graphics tablets are constructed with a rectangular grid of wires embedded in the tablet surface. Electromagnetic pulses are generated in sequence



**Figure 2-46**  
The SummaSketch III desktop tablet with a 16-button hand cursor. (Courtesy of Summagraphics Corporation.)



*Figure 2-47*  
The Micropad III tablet with a 16-button hand cursor, designed for digitizing larger drawings. (Courtesy of Summagraphics Corporation.)



*Figure 2-48*  
The NotePad desktop tablet with stylus. (Courtesy of CalComp Digitizer Division, a part of CalComp, Inc.)

along the wires, and an electric signal is induced in a wire coil in an activated stylus or hand cursor to record a tablet position. Depending on the technology, either signal strength, coded pulses, or phase shifts can be used to determine the position on the tablet.

Acoustic (or sonic) tablets use sound waves to detect a stylus position. Either strip microphones or point microphones can be used to detect the sound emitted by an electrical spark from a stylus tip. The position of the stylus is calcu-



*Figure 2-49*  
An artist's digitizer system, with a pressure-sensitive, cordless stylus. (Courtesy of Wacom Technology Corporation.)

lated by timing the arrival of the generated sound at the different microphone positions. An advantage of two-dimensional acoustic tablets is that the microphones can be placed on any surface to form the "tablet" work area. This can be convenient for various applications, such as digitizing drawings in a book.

Three-dimensional digitizers use sonic or electromagnetic transmissions to record positions. One electromagnetic transmission method is similar to that used in the data glove: A coupling between the transmitter and receiver is used to compute the location of a stylus as it moves over the surface of an object. Figure 2-50 shows a three-dimensional digitizer designed for Apple Macintosh computers. As the points are selected on a nonmetallic object, a wireframe outline of the surface is displayed on the computer screen. Once the surface outline is constructed, it can be shaded with lighting effects to produce a realistic display of the object. Resolution of this system is from 0.8 mm to 0.08 mm, depending on the model.

---

Section 2-5  
Input Devices

#### Image Scanners

Drawings, graphs, color and black-and-white photos, or text can be stored for computer processing with an image scanner by passing an optical scanning mechanism over the information to be stored. The gradations of gray scale or color are then recorded and stored in an array. Once we have the internal representation of a picture, we can apply transformations to rotate, scale, or crop the picture to a particular screen area. We can also apply various image-processing methods to modify the array representation of the picture. For scanned text input, various editing operations can be performed on the stored documents. Some scanners are able to scan either graphical representations or text, and they come in a variety of sizes and capabilities. A small hand-model scanner is shown in Fig. 2-51, while Figs 2-52 and 2-53 show larger models.



Figure 2-50  
A three-dimensional digitizing system for use with Apple Macintosh computers. (Courtesy of Mira Imaging.)

Chapter 2

Overview of Graphics Systems



Figure 2-51  
A hand-held scanner that can be used to input either text or graphics images. (Courtesy of Thunderware, Inc.)



Figure 2-52  
Desktop full-color scanners: (a) Flatbed scanner with a resolution of 600 dots per inch.  
(Courtesy of Sharp Electronics Corporation.) (b) Drum scanner with a selectable resolution from 50 to 4000 dots per inch. (Courtesy of Howtek, Inc.)

### Touch Panels

As the name implies, **touch panels** allow displayed objects or screen positions to be selected with the touch of a finger. A typical application of touch panels is for the selection of processing options that are represented with graphical icons. Some systems, such as the plasma panels shown in Fig. 2-54, are designed with touch screens. Other systems can be adapted for touch input by fitting a transparent device with a touch-sensing mechanism over the video monitor screen. Touch input can be recorded using optical, electrical, or acoustical methods.

Optical touch panels employ a line of infrared light-emitting diodes (LEDs) along one vertical edge and along one horizontal edge of the frame. The opposite vertical and horizontal edges contain light detectors. These detectors are used to record which beams are interrupted when the panel is touched. The two crossing



**Figure 2-53**  
A large floor-model scanner used to scan architectural and engineering drawings up to 40 inches wide and 100 feet long. (Courtesy of Summagraphics Corporation.)

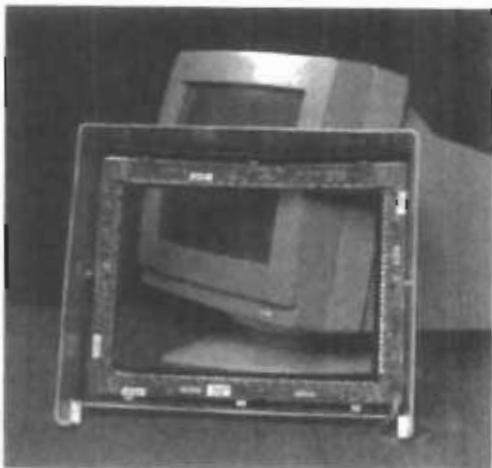
beams that are interrupted identify the horizontal and vertical coordinates of the screen position selected. Positions can be selected with an accuracy of about 1/4 inch. With closely spaced LEDs, it is possible to break two horizontal or two vertical beams simultaneously. In this case, an average position between the two interrupted beams is recorded. The LEDs operate at infrared frequencies, so that the light is not visible to a user. Figure 2-55 illustrates the arrangement of LEDs in an optical touch panel that is designed to match the color and contours of the system to which it is to be fitted.

An electrical touch panel is constructed with two transparent plates separated by a small distance. One of the plates is coated with a conducting material, and the other plate is coated with a resistive material. When the outer plate is touched, it is forced into contact with the inner plate. This contact creates a voltage drop across the resistive plate that is converted to the coordinate values of the selected screen position.

In acoustical touch panels, high-frequency sound waves are generated in the horizontal and vertical directions across a glass plate. Touching the screen causes part of each wave to be reflected from the finger to the emitters. The screen position at the point of contact is calculated from a measurement of the time interval between the transmission of each wave and its reflection to the emitter.



**Figure 2-54**  
Plasma panels with touch screens. (Courtesy of Photonics Systems.)



**Figure 2-55**  
An optical touch panel, showing the arrangement of infrared LED units and detectors around the edges of the frame. (Courtesy of Carroll Touch, Inc.)

### Light Pens

Figure 2-56 shows the design of one type of light pen. Such pencil-shaped devices are used to select screen positions by detecting the light coming from points on the CRT screen. They are sensitive to the short burst of light emitted from the phosphor coating at the instant the electron beam strikes a particular point. Other light sources, such as the background light in the room, are usually not detected by a light pen. An activated light pen, pointed at a spot on the screen as the electron beam lights up that spot, generates an electrical pulse that causes the coordinate position of the electron beam to be recorded. As with cursor-positioning devices, recorded light-pen coordinates can be used to position an object or to select a processing option.

Although light pens are still with us, they are not as popular as they once were since they have several disadvantages compared to other input devices that have been developed. For one, when a light pen is pointed at the screen, part of the screen image is obscured by the hand and pen. And prolonged use of the light pen can cause arm fatigue. Also, light pens require special implementations for some applications because they cannot detect positions within black areas. To be able to select positions in any screen area with a light pen, we must have some nonzero intensity assigned to each screen pixel. In addition, light pens sometimes give false readings due to background lighting in a room.

### Voice Systems

Speech recognizers are used in some graphics workstations as input devices to accept voice commands. The voice-system input can be used to initiate graphics



Figure 2-56  
A light pen activated with a button switch. (Courtesy of Interactive Computer Products.)

operations or to enter data. These systems operate by matching an input against a predefined dictionary of words and phrases.

A dictionary is set up for a particular operator by having the operator speak the command words to be used into the system. Each word is spoken several times, and the system analyzes the word and establishes a frequency pattern for that word in the dictionary along with the corresponding function to be performed. Later, when a voice command is given, the system searches the dictionary for a frequency-pattern match. Voice input is typically spoken into a microphone mounted on a headset, as in Fig. 2-57. The microphone is designed to minimize input of other background sounds. If a different operator is to use the system, the dictionary must be reestablished with that operator's voice patterns. Voice systems have some advantage over other input devices, since the attention of the operator does not have to be switched from one device to another to enter a command.



Figure 2-57  
A speech-recognition system. (Courtesy of Threshold Technology, Inc.)

## HARD-COPY DEVICES

We can obtain hard-copy output for our images in several formats. For presentations or archiving, we can send image files to devices or service bureaus that will produce 35-mm slides or overhead transparencies. To put images on film, we can simply photograph a scene displayed on a video monitor. And we can put our pictures on paper by directing graphics output to a printer or plotter.

The quality of the pictures obtained from a device depends on dot size and the number of dots per inch, or lines per inch, that can be displayed. To produce smooth characters in printed text strings, higher-quality printers shift dot positions so that adjacent dots overlap.

Printers produce output by either impact or nonimpact methods. *Impact* printers press formed character faces against an inked ribbon onto the paper. A line printer is an example of an impact device, with the typefaces mounted on bands, chains, drums, or wheels. *Nonimpact* printers and plotters use laser techniques, ink-jet sprays, xerographic processes (as used in photocopying machines), electrostatic methods, and electrothermal methods to get images onto paper.

Character impact printers often have a *dot-matrix* print head containing a rectangular array of protruding wire pins, with the number of pins depending on the quality of the printer. Individual characters or graphics patterns are obtained by retracting certain pins so that the remaining pins form the pattern to be printed. Figure 2-58 shows a picture printed on a dot-matrix printer.

In a *laser* device, a laser beam creates a charge distribution on a rotating drum coated with a photoelectric material, such as selenium. Toner is applied to the drum and then transferred to paper. Figure 2-59 shows examples of desktop laser printers with a resolution of 360 dots per inch.

*Ink-jet* methods produce output by squirting ink in horizontal rows across a roll of paper wrapped on a drum. The electrically charged ink stream is deflected by an electric field to produce dot-matrix patterns. A desktop ink-jet plotter with



Figure 2-58

A picture generated on a dot-matrix printer showing how the density of the dot patterns can be varied to produce light and dark areas. (Courtesy of Apple Computer, Inc.)

Section 2-6

Hard-Copy Devices



Figure 2-59  
Small-footprint laser printers.  
(Courtesy of Texas Instruments.)

a resolution of 360 dots per inch is shown in Fig. 2-60, and examples of larger high-resolution ink-jet printer/ plotters are shown in Fig. 2-61.

An electrostatic device places a negative charge on the paper, one complete row at a time along the length of the paper. Then the paper is exposed to a toner. The toner is positively charged and so is attracted to the negatively charged areas, where it adheres to produce the specified output. A color electrostatic printer/plotter is shown in Fig. 2-62. *Electrothermal* methods use heat in a dot-matrix print head to output patterns on heat-sensitive paper.

We can get limited color output on an impact printer by using different-colored ribbons. Nonimpact devices use various techniques to combine three color pigments (cyan, magenta, and yellow) to produce a range of color patterns. Laser and xerographic devices deposit the three pigments on separate passes; ink-jet methods shoot the three colors simultaneously on a single pass along each print line on the paper.



Figure 2-60  
A 360-dot-per-inch desktop ink-jet plotter. (Courtesy of Summagraphics Corporation.)

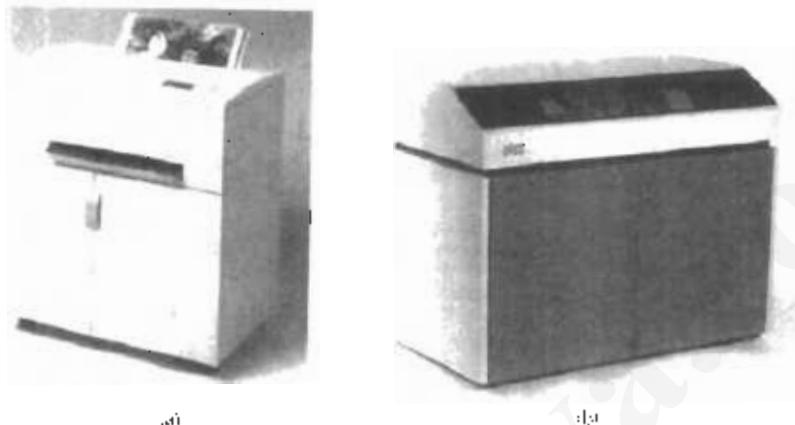


Figure 2-61

Floor-model, ink-jet color printers that use variable dot size to achieve an equivalent resolution of 1500 to 1800 dots per inch. (Courtesy of Iris Graphics Inc., Bedford, Massachusetts.)



Figure 2-62

An electrostatic printer that can display 400 dots per inch. (Courtesy of CalComp Digitizer Division, a part of CalComp, Inc.)

Drafting layouts and other drawings are typically generated with ink-jet or pen plotters. A pen plotter has one or more pens mounted on a carriage, or crossbar, that spans a sheet of paper. Pens with varying colors and widths are used to produce a variety of shadings and line styles. Wet-ink, ball-point, and felt-tip pens are all possible choices for use with a pen plotter. Plotter paper can lie flat or be rolled onto a drum or belt. Crossbars can be either moveable or stationary, while the pen moves back and forth along the bar. Either clamps, a vacuum, or an electrostatic charge hold the paper in position. An example of a table-top flatbed pen plotter is given in Figure 2-63, and a larger, rollfeed pen plotter is shown in Fig. 2-64.

Section 2-7  
Graphics Software



Figure 2-63  
A desktop pen plotter with a resolution of 0.025 mm. (Courtesy of Summagraphics Corporation.)



Figure 2-64  
A large, rollfeed pen plotter with automatic multicolor 8-pen changer and a resolution of 0.0127 mm. (Courtesy of Summagraphics Corporation.)

2-7

GRAPHICS SOFTWARE

There are two general classifications for graphics software: general programming packages and special-purpose applications packages. A general graphics programming package provides an extensive set of graphics functions that can be

---

REFERENCES

---

Exercises

A general treatment of electronic displays, including flat-panel devices, is available in Sher (1993). Flat-panel devices are discussed in Depp and Howard (1993). Janusz (1983) provides a reference for both flat-panel displays and CRTs. Additional information on raster-graphics architecture can be found in Foley et al. (1990). Three-dimensional terminals are discussed in Luchs et al. (1982), Johnson (1982), and Ikeda (1984). Head-mounted displays and virtual-reality environments are discussed in Chung et al. (1989).

For information on PHIGS and PHIGS+, see Hopgood and Duce (1991), Howard et al. (1991), Gaskins (1992), and Blake (1993). Information on the two-dimensional GKS standard and on the evolution of graphics standards is available in Hopgood et al. (1983). An additional reference for GKS is Underle, Kansy, and Pfaff (1984).

---

EXERCISES

---

- 2-1. List the operating characteristics for the following display technologies: raster refresh systems, vector refresh systems, plasma panels, and LCDs.
- 2-2. List some applications appropriate for each of the display technologies in Exercise 2-1.
- 2-3. Determine the resolution (pixels per centimeter) in the *x* and *y* directions for the video monitor in use on your system. Determine the aspect ratio, and explain how relative proportions of objects can be maintained on your system.
- 2-4. Consider three different raster systems with resolutions of 640 by 480, 1280 by 1024, and 2560 by 2048. What size frame buffer (in bytes) is needed for each of these systems to store 12 bits per pixel? How much storage is required for each system if 24 bits per pixel are to be stored?
- 2-5. Suppose an RGB raster system is to be designed using an 8-inch by 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 the frame buffer?
- 2-6. How long would it take to load a 640 by 480 frame buffer with 12 bits per pixel, if 10<sup>9</sup> bits can be transferred per second? How long would it take to load a 24-bit-per-pixel frame buffer with a resolution of 1280 by 1024 using the same transfer rate?
- 2-7. Suppose we have a computer with 32 bits per word and a transfer rate of 1 megabyte (million instructions per second). How long would it take to fill the frame buffer of a 300-dpi (dot per inch) laser printer with a page size of 8 1/2 inches by 11 inches?
- 2-8. Consider two raster systems with resolutions of 640 by 480 and 1280 by 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? What is the access time per pixel in each system?
- 2-9. Suppose we have a video monitor with a display area that measures 12 inches across and 9.6 inches high. If the resolution is 1280 by 1024 and the aspect ratio is 1, what is the diameter of each screen point?
- 2-10. How much time is spent scanning across each row of pixels during screen refresh on a raster system with a resolution of 1280 by 1024 and a refresh rate of 60 frames per second?
- 2-11. Consider a noninterlaced raster monitor with a resolution of  $n$  by  $m$  scan lines and  $n$  pixels per scan line, a refresh rate of  $r$  frames per second, a horizontal retrace time of  $t_{\text{htr}}$ , and a vertical retrace time of  $t_{\text{vtr}}$ . What is the fraction of the total refresh time per frame spent in retrace of the electron beam?
- 2-12. What is the fraction of the total refresh time per frame spent in retrace of the electron beam for a noninterlaced raster system with a resolution of 1280 by 1024, a refresh rate of 60 Hz, a horizontal retrace time of 5 microseconds, and a vertical retrace time of 500 microseconds?

---

**Chapter 2**

**Overview of Graphics Systems**

- 2-13. Assuming that a certain full-color (24-bit per pixel) RGB raster system has a 512-by-512 image buffer, how many distinct color entries (intensity levels) would we have available? How many different colors could we display at any one time?
- 2-14. Compare the advantages and disadvantages of a three-dimensional monitor using a varifocal mirror with a stereoscopic system.
- 2-15. List the different input and output components that are typically used with virtual-reality systems. Also explain how users interact with a virtual scene displayed with different output devices, such as two-dimensional and stereoscopic monitors.
- 2-16. Explain how virtual-reality systems can be used in design applications. What are some other applications for virtual-reality systems?
- 2-17. List some applications for large-screen displays.
- 2-18. Explain the differences between a general graphics system designed for a programmer and one designed for a specific application, such as architectural design!

# Drawing Primitives

S. No.	Topic	Contents
3.	Raster scan line, circle and ellipse drawing algorithms, Polygon filling line clipping and polygon, clipping algorithms	Sections 3.2 -3.2.2, Section 3.3 (before 2 <sup>nd</sup> order difference), Section 3.4, Sections 3.6, Section 3.9, Section 3.12-3.12.3, Section 3.14, Section 3.17-3.17.3

---

## TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

Raster displays invoke clipping and scan-conversion algorithms each time an image is created or modified. Hence, these algorithms not only must create visually satisfactory images, but also must execute as rapidly as possible. As discussed in detail in later sections, scan-conversion algorithms use *incremental methods* to minimize the number of calculations (especially multiplies and divides) performed during each iteration; further, these calculations employ integer rather than floating-point arithmetic. As shown in Chapter 18, speed can be increased even further by using multiple parallel processors to scan convert simultaneously entire output primitives or pieces of them.

### 3.2 SCAN CONVERTING LINES

A scan-conversion algorithm for lines computes the coordinates of the pixels that lie on or near an ideal, infinitely thin straight line imposed on a 2D raster grid. In principle, we would like the sequence of pixels to lie as close to the ideal line as possible and to be as straight as possible. Consider a 1-pixel-thick approximation to an ideal line; what properties should it have? For lines with slopes between  $-1$  and  $1$  inclusive, exactly 1 pixel should be illuminated in each column; for lines with slopes outside this range, exactly 1 pixel should be illuminated in each row. All lines should be drawn with constant brightness, independent of length and orientation, and as rapidly as possible. There should also be provisions for drawing lines that are more than 1 pixel wide, centered on the ideal line, that are affected by line-style and pen-style attributes, and that create other effects needed for high-quality illustrations. For example, the shape of the endpoint regions should be under programmer control to allow beveled, rounded, and mitered corners. We would even like to be able to minimize the jaggies due to the discrete approximation of the ideal line by using antialiasing techniques exploiting the ability to set the intensity of individual pixels on  $n$ -bits-per-pixel displays.

For now, we consider only “optimal,” 1-pixel-thick lines that have exactly 1 bilevel pixel in each column (or row for steep lines). Later in the chapter, we consider thick primitives and deal with styles.

To visualize the geometry, we recall that SRGP represents a pixel as a circular dot centered at that pixel’s  $(x, y)$  location on the integer grid. This representation is a convenient approximation to the more or less circular cross-section of the CRT’s electron beam, but the exact spacing between the beam spots on an actual display can vary greatly among systems. In some systems, adjacent spots overlap; in others, there may be space between adjacent vertical pixels; in most systems, the spacing is tighter in the horizontal than in the vertical direction. Another variation in coordinate-system representation arises in systems, such as the Macintosh, that treat pixels as being centered in the rectangular box between adjacent grid lines instead of on the grid lines themselves. In this scheme, rectangles are defined to be all pixels interior to the mathematical rectangle defined by two corner points. This definition allows zero-width (null) canvases: The rectangle from  $(x, y)$  to  $(x, y)$  contains no pixels, unlike the SRGP canvas, which has a single pixel at that point. For now, we continue to represent pixels as disjoint circles centered on a uniform grid, although we shall make some minor changes when we discuss antialiasing.

Figure 3.4 shows a highly magnified view of a 1-pixel-thick line and of the ideal line that it approximates. The intensified pixels are shown as filled circles and the nonintensified

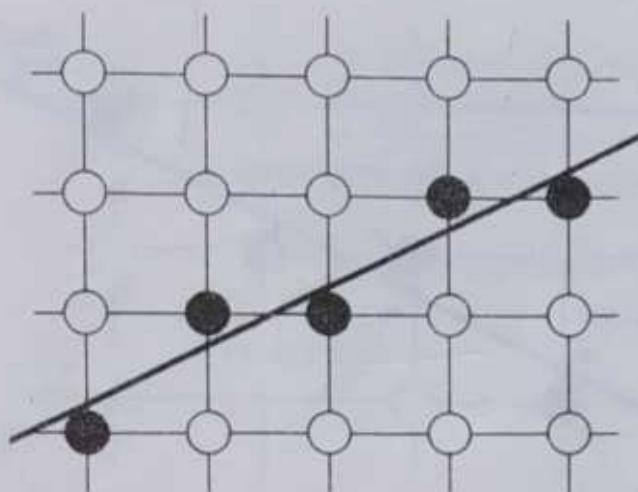


Fig. 3.4 A scan-converted line showing intensified pixels as black circles.

pixels are shown as unfilled circles. On an actual screen, the diameter of the roughly circular pixel is larger than the interpixel spacing, so our symbolic representation exaggerates the discreteness of the pixels.

Since SRGP primitives are defined on an integer grid, the endpoints of a line have integer coordinates. In fact, if we first clip the line to the clip rectangle, a line intersecting a clip edge may actually have an endpoint with a noninteger coordinate value. The same is true when we use a floating-point raster graphics package. (We discuss these noninteger intersections in Section 3.2.3.) Assume that our line has slope  $|m| \leq 1$ ; lines at other slopes can be handled by suitable changes in the development that follows. Also, the most common lines—those that are horizontal, are vertical, or have a slope of  $\pm 1$ —can be handled as trivial special cases because these lines pass through only pixel centers (see Exercise 3.1).

### 3.2.1 The Basic Incremental Algorithm

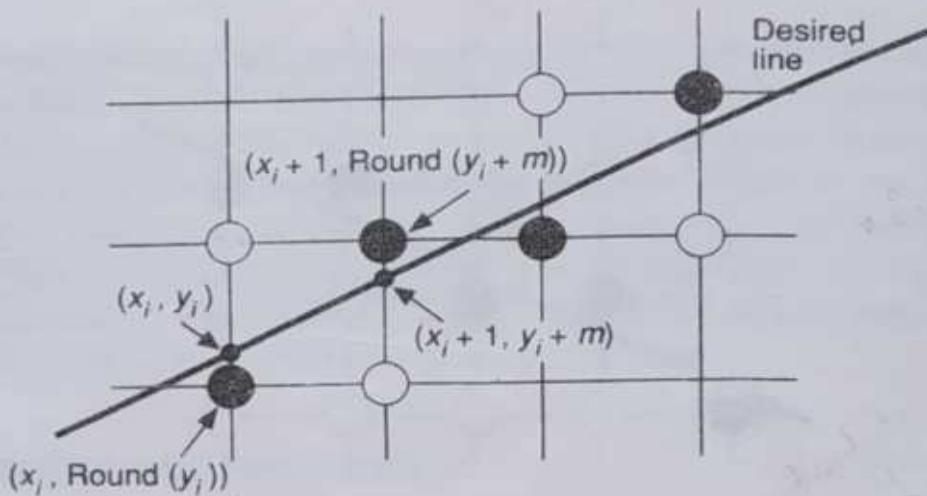
The simplest strategy for scan conversion of lines is to compute the slope  $m$  as  $\Delta y / \Delta x$ , to increment  $x$  by 1 starting with the leftmost point, to calculate  $y_i = mx_i + B$  for each  $x_i$ , and to intensify the pixel at  $(x_i, \text{Round}(y_i))$ , where  $\text{Round}(y_i) = \text{Floor}(0.5 + y_i)$ . This computation selects the closest pixel—that is, the pixel whose distance to the true line is smallest.<sup>1</sup> This brute-force strategy is inefficient, however, because each iteration requires a floating-point (or binary fraction) multiply, addition, and invocation of Floor. We can eliminate the multiplication by noting that

$$y_{i+1} = mx_{i+1} + B = m(x_i + \Delta x) + B = y_i + m\Delta x,$$

and, if  $\Delta x = 1$ , then  $y_{i+1} = y_i + m$ .

Thus, a unit change in  $x$  changes  $y$  by  $m$ , which is the slope of the line. For all points  $(x_i, y_i)$  on the line, we know that, if  $x_{i+1} = x_i + 1$ , then  $y_{i+1} = y_i + m$ ; that is, the values of  $x$  and  $y$  are defined in terms of their previous values (see Fig. 3.5). This is what defines an

<sup>1</sup>In Chapter 19, we discuss various measures of closeness for lines and general curves (also called *error measures*).



**Fig. 3.5** Incremental calculation of  $(x_i, y_i)$ .

incremental algorithm: At each step, we make incremental calculations based on the preceding step.

We initialize the incremental calculation with  $(x_0, y_0)$ , the integer coordinates of an endpoint. Note that this incremental technique avoids the need to deal with the y intercept,  $B$ , explicitly. If  $|m| > 1$ , a step in  $x$  creates a step in  $y$  that is greater than 1. Thus, we must reverse the roles of  $x$  and  $y$  by assigning a unit step to  $y$  and incrementing  $x$  by  $\Delta x = \Delta y/m = 1/m$ . Line, the procedure in Fig. 3.6, implements this technique. The start point must be the left endpoint. Also, it is limited to the case  $-1 \leq m \leq 1$ , but other slopes may be accommodated by symmetry. The checking for the special cases of horizontal, vertical, or diagonal lines is omitted.

WritePixel, used by Line, is a low-level procedure provided by the device-level software; it places a value into a canvas for a pixel whose coordinates are given as the first two arguments.<sup>2</sup> We assume here that we scan convert only in replace mode; for SRGP's other write modes, we must use a low-level ReadPixel procedure to read the pixel at the destination location, logically combine that pixel with the source pixel, and then write the result into the destination pixel with WritePixel.

This algorithm is often referred to as a *digital differential analyzer (DDA)* algorithm. The DDA is a mechanical device that solves differential equations by numerical methods: It traces out successive  $(x, y)$  values by simultaneously incrementing  $x$  and  $y$  by small steps proportional to the first derivative of  $x$  and  $y$ . In our case, the  $x$  increment is 1, and the  $y$  increment is  $dy/dx = m$ . Since real variables have limited precision, summing an inexact  $m$  repetitively introduces cumulative error buildup and eventually a drift away from a true  $\text{Round}(y_i)$ ; for most (short) lines, this will not present a problem.

### 3.2.2 Midpoint Line Algorithm

The drawbacks of procedure Line are that rounding  $y$  to an integer takes time, and that the variables  $y$  and  $m$  must be real or fractional binary because the slope is a fraction. Bresenham developed a classic algorithm [BRES65] that is attractive because it uses only

<sup>2</sup>If such a low-level procedure is not available, the SRGP\_pointCoord procedure may be used, as described in the SRGP reference manual.

```

void Line (
    int  $x_0$ , int  $y_0$ ,
    int  $x_1$ , int  $y_1$ ,
    int value)
{
    int  $x$ ;                                /*  $x$  runs from  $x_0$  to  $x_1$  in unit increments. */

    double  $dy = y_1 - y_0$ ;
    double  $dx = x_1 - x_0$ ;
    double  $m = dy / dx$ ;
    double  $y = y_0$ ;
    for ( $x = x_0; x \leq x_1; x++$ ) {
        WritePixel ( $x$ , Round ( $y$ ), value);    /* Set pixel to value */
         $y += m$ ;                            /* Step  $y$  by slope  $m$  */
    }
} /* Line */

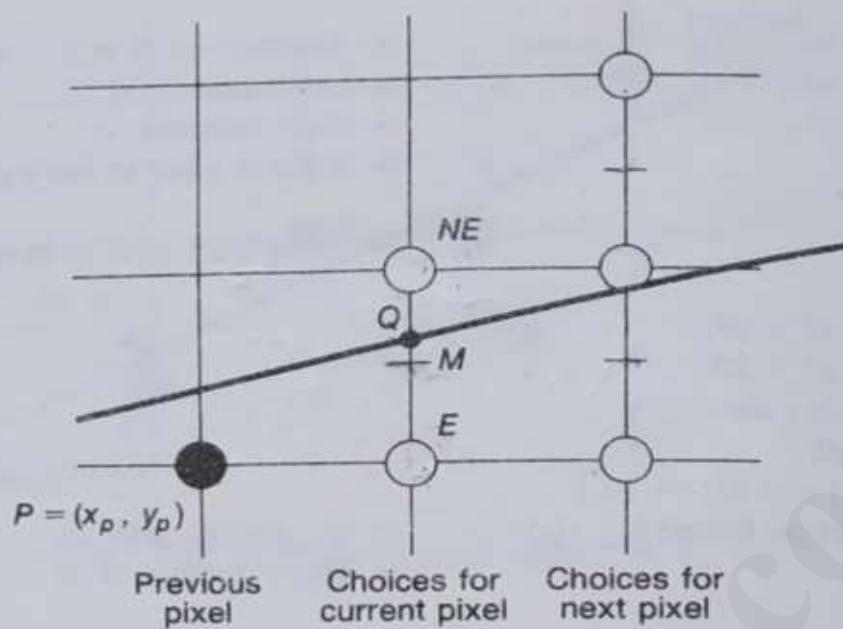
```

**Fig. 3.6** The incremental line scan-conversion algorithm.

integer arithmetic, thus avoiding the Round function, and allows the calculation for  $(x_{i+1}, y_{i+1})$  to be performed incrementally—that is, by using the calculation already done at  $(x_i, y_i)$ . A floating-point version of this algorithm can be applied to lines with arbitrary real-valued endpoint coordinates. Furthermore, Bresenham's incremental technique may be applied to the integer computation of circles as well, although it does not generalize easily to arbitrary conics. We therefore use a slightly different formulation, the *midpoint technique*, first published by Pitteway [PITT67] and adapted by Van Aken [VANA84] and other researchers. For lines and integer circles, the midpoint formulation, as Van Aken shows [VANA85], reduces to the Bresenham formulation and therefore generates the same pixels. Bresenham showed that his line and integer circle algorithms provide the best-fit approximations to true lines and circles by minimizing the error (distance) to the true primitive [BRES77]. Kappel discusses the effects of various error criteria in [KAPP85].

We assume that the line's slope is between 0 and 1. Other slopes can be handled by suitable reflections about the principal axes. We call the lower-left endpoint  $(x_0, y_0)$  and the upper-right endpoint  $(x_1, y_1)$ .

Consider the line in Fig. 3.7, where the previously selected pixel appears as a black circle and the two pixels from which to choose at the next stage are shown as unfilled circles. Assume that we have just selected the pixel  $P$  at  $(x_p, y_p)$  and now must choose between the pixel one increment to the right (called the east pixel,  $E$ ) or the pixel one increment to the right and one increment up (called the northeast pixel,  $NE$ ). Let  $Q$  be the intersection point of the line being scan-converted with the grid line  $x = x_p + 1$ . In Bresenham's formulation, the difference between the vertical distances from  $E$  and  $NE$  to  $Q$  is computed, and the sign of the difference is used to select the pixel whose distance from  $Q$  is smaller as the best approximation to the line. In the midpoint formulation, we observe on which side of the line the midpoint  $M$  lies. It is easy to see that, if the midpoint lies above the line, pixel  $E$  is closer to the line; if the midpoint lies below the line, pixel  $NE$  is closer to the line. The line may pass between  $E$  and  $NE$ , or both pixels may lie on one side, but in any



**Fig. 3.7** The pixel grid for the midpoint line algorithm, showing the midpoint  $M$ , and the  $E$  and  $NE$  pixels to choose between.

case, the midpoint test chooses the closest pixel. Also, the error—that is, the vertical distance between the chosen pixel and the actual line—is always  $\leq 1/2$ .

The algorithm chooses  $NE$  as the next pixel for the line shown in Fig. 3.7. Now all we need is a way to calculate on which side of the line the midpoint lies. Let's represent the line by an implicit function<sup>3</sup> with coefficients  $a$ ,  $b$ , and  $c$ :  $F(x, y) = ax + by + c = 0$ . (The  $b$  coefficient of  $y$  is unrelated to the  $y$  intercept  $B$  in the slope-intercept form.) If  $dy = y_1 - y_0$ , and  $dx = x_1 - x_0$ , the slope-intercept form can be written as

$$y = \frac{dy}{dx}x + B;$$

therefore,

$$F(x, y) = dy \cdot x - dx \cdot y + B \cdot dx = 0.$$

Here  $a = dy$ ,  $b = -dx$ , and  $c = B \cdot dx$  in the implicit form.<sup>4</sup>

It can easily be verified that  $F(x, y)$  is zero on the line, positive for points below the line, and negative for points above the line. To apply the midpoint criterion, we need only to compute  $F(M) = F(x_p + 1, y_p + \frac{1}{2})$  and to test its sign. Because our decision is based on the value of the function at  $(x_p + 1, y_p + \frac{1}{2})$ , we define a *decision variable*  $d = F(x_p + 1, y_p + \frac{1}{2})$ . By definition,  $d = a(x_p + 1) + b(y_p + \frac{1}{2}) + c$ . If  $d > 0$ , we choose pixel  $NE$ ; if  $d < 0$ , we choose  $E$ ; and if  $d = 0$ , we can choose either, so we pick  $E$ .

Next, we ask what happens to the location of  $M$  and therefore to the value of  $d$  for the next grid line; both depend, of course, on whether we chose  $E$  or  $NE$ . If  $E$  is chosen,  $M$  is

<sup>3</sup>This functional form extends nicely to the implicit formulation of both circles and ellipses.

<sup>4</sup>It is important for the proper functioning of the midpoint algorithm to choose  $a$  to be positive; we meet this criterion if  $dy$  is positive, since  $y_1 > y_0$ .

incremented by one step in the  $x$  direction. Then,

$$d_{\text{new}} = F(x_p + 2, y_p + \frac{1}{2}) = a(x_p + 2) + b(y_p + \frac{1}{2}) + c,$$

but

$$d_{\text{old}} = a(x_p + 1) + b(y_p + \frac{1}{2}) + c,$$

Subtracting  $d_{\text{old}}$  from  $d_{\text{new}}$  to get the incremental difference, we write  $d_{\text{new}} = d_{\text{old}} + a$ .

We call the increment to add after  $E$  is chosen  $\Delta_E$ ;  $\Delta_E = a = dy$ . In other words, we can derive the value of the decision variable at the next step incrementally from the value at the current step without having to compute  $F(M)$  directly, by merely adding  $\Delta_E$ .

If  $NE$  is chosen,  $M$  is incremented by one step each in both the  $x$  and  $y$  directions. Then,

$$d_{\text{new}} = F(x_p + 2, y_p + \frac{3}{2}) = a(x_p + 2) + b(y_p + \frac{3}{2}) + c.$$

Subtracting  $d_{\text{old}}$  from  $d_{\text{new}}$  to get the incremental difference, we write

$$d_{\text{new}} = d_{\text{old}} + a + b.$$

We call the increment to add to  $d$  after  $NE$  is chosen  $\Delta_{NE}$ ;  $\Delta_{NE} = a + b = dy - dx$ .

Let's summarize the incremental midpoint technique. At each step, the algorithm chooses between 2 pixels based on the sign of the decision variable calculated in the previous iteration; then, it updates the decision variable by adding either  $\Delta_E$  or  $\Delta_{NE}$  to the old value, depending on the choice of pixel.

Since the first pixel is simply the first endpoint  $(x_0, y_0)$ , we can directly calculate the initial value of  $d$  for choosing between  $E$  and  $NE$ . The first midpoint is at  $(x_0 + 1, y_0 + \frac{1}{2})$ , and

$$\begin{aligned} F(x_0 + 1, y_0 + \frac{1}{2}) &= a(x_0 + 1) + b(y_0 + \frac{1}{2}) + c \\ &= ax_0 + by_0 + c + a + b/2 \\ &= F(x_0, y_0) + a + b/2. \end{aligned}$$

But  $(x_0, y_0)$  is a point on the line and  $F(x_0, y_0)$  is therefore 0; hence,  $d_{\text{start}}$  is just  $a + b/2 = dy - dx/2$ . Using  $d_{\text{start}}$ , we choose the second pixel, and so on. To eliminate the fraction in  $d_{\text{start}}$ , we redefine our original  $F$  by multiplying it by 2;  $F(x, y) = 2(ax + by + c)$ . This multiplies each constant and the decision variable by 2, but does not affect the sign of the decision variable, which is all that matters for the midpoint test.

The arithmetic needed to evaluate  $d_{\text{new}}$  for any step is simple addition. No time-consuming multiplication is involved. Further, the inner loop is quite simple, as seen in the midpoint algorithm of Fig. 3.8. The first statement in the loop, the test of  $d$ , determines the choice of pixel, but we actually increment  $x$  and  $y$  to that pixel location after updating the decision variable (for compatibility with the circle and ellipse algorithms). Note that this version of the algorithm works for only those lines with slope between 0 and 1; generalizing the algorithm is left as Exercise 3.2. In [SPRO82], Sproull gives an elegant derivation of Bresenham's formulation of this algorithm as a series of program transformations from the original brute-force algorithm. No equivalent of that derivation for circles or ellipses has yet appeared, but the midpoint technique does generalize, as we shall see.

## 102 Basic Raster Graphics Algorithms for Drawing 2D Primitives

```

void MidpointLine (int  $x_0$ , int  $y_0$ , int  $x_1$ , int  $y_1$ , int value)
{
    int  $dx = x_1 - x_0$ ;
    int  $dy = y_1 - y_0$ ;
    int  $d = 2 * dy - dx$ ;           /* Initial value of d */
    int  $incrE = 2 * dy$ ;           /* Increment used for move to E */
    int  $incrNE = 2 * (dy - dx)$ ;  /* Increment used for move to NE */
    int  $x = x_0$ ;
    int  $y = y_0$ ;
    WritePixel ( $x, y, value$ );      /* The start pixel */

    while ( $x < x_1$ ) {
        if ( $d \leq 0$ ) {             /* Choose E */
             $d += incrE$ ;
             $x++$ ;
        } else {                   /* Choose NE */
             $d += incrNE$ ;
             $x++$ ;
             $y++$ ;
        }
        WritePixel ( $x, y, value$ );  /* The selected pixel closest to the line */
    } /* while */
} /* MidpointLine */

```

**Fig. 3.8** The midpoint line scan-conversion algorithm.

For a line from point (5, 8) to point (9, 11), the successive values of  $d$  are 2, 0, 6, and 4, resulting in the selection of NE, E, NE, and then NE, respectively, as shown in Fig. 3.9. The line appears abnormally jagged because of the enlarged scale of the drawing and the artificially large interpixel spacing used to make the geometry of the algorithm clear. For the same reason, the drawings in the following sections also make the primitives appear blockier than they look on an actual screen.

### 3.3 SCAN CONVERTING CIRCLES

Although SRGP does not offer a circle primitive, the implementation will benefit from treating the circular ellipse arc as a special case because of its eight-fold symmetry, both for clipping and for scan conversion. The equation of a circle centered at the origin is  $x^2 + y^2 = R^2$ . Circles not centered at the origin may be translated to the origin by integer amounts and then scan converted, with pixels written with the appropriate offset. There are several easy but inefficient ways to scan convert a circle. Solving for  $y$  in the implicit circle equation, we get the explicit  $y = f(x)$  as

$$y = \pm\sqrt{R^2 - x^2}.$$

To draw a quarter circle (the other quarters are drawn by symmetry), we can increment  $x$  from 0 to  $R$  in unit steps, solving for  $+y$  at each step. This approach works, but it is

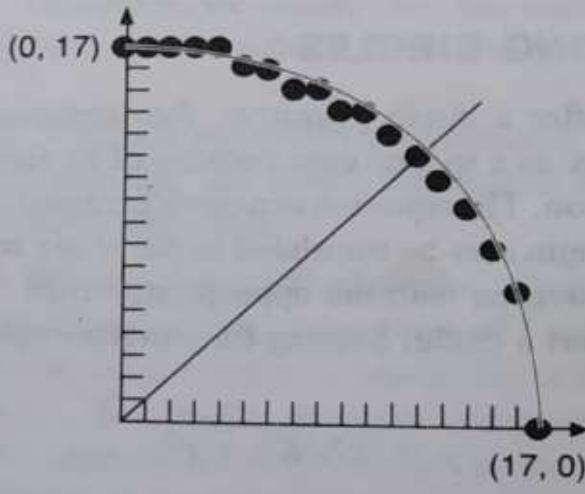
inefficient because of the multiply and square-root operations. Furthermore, the circle will have large gaps for values of  $x$  close to  $R$ , because the slope of the circle becomes infinite there (see Fig. 3.12). A similarly inefficient method, which does, however, avoid the large gaps, is to plot  $(R \cos\theta, R \sin\theta)$  by stepping  $\theta$  from  $0^\circ$  to  $90^\circ$ .

### 3.3.1 Eight-Way Symmetry

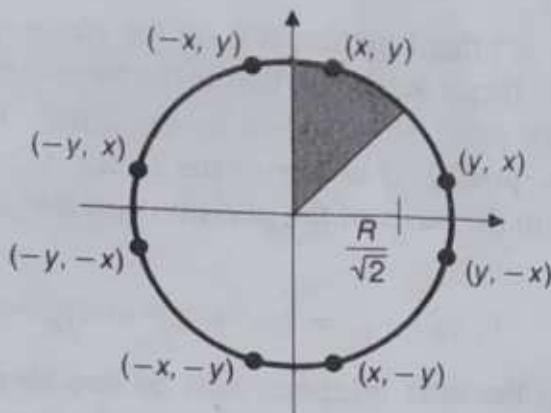
We can improve the drawing process of the previous section by taking greater advantage of the symmetry in a circle. Consider first a circle centered at the origin. If the point  $(x, y)$  is on the circle, then we can trivially compute seven other points on the circle, as shown in Fig. 3.13. Therefore, we need to compute only one  $45^\circ$  segment to determine the circle completely. For a circle centered at the origin, the eight symmetrical points can be displayed with procedure CirclePoints (the procedure is easily generalized to the case of circles with arbitrary origins):

```
void CirclePoints (int x, int y, int value)
{
    WritePixel (x, y, value);
    WritePixel (y, x, value);
    WritePixel (y, -x, value);
    WritePixel (x, -y, value);
    WritePixel (-x, -y, value);
    WritePixel (-y, -x, value);
    WritePixel (-y, x, value);
    WritePixel (-x, y, value);
/* CirclePoints */
```

We do not want to call CirclePoints when  $x = y$ , because each of four pixels would be set twice; the code is easily modified to handle that boundary condition.



**Fig. 3.12** A quarter circle generated with unit steps in  $x$ , and with  $y$  calculated and then rounded. Unique values of  $y$  for each  $x$  produce gaps.

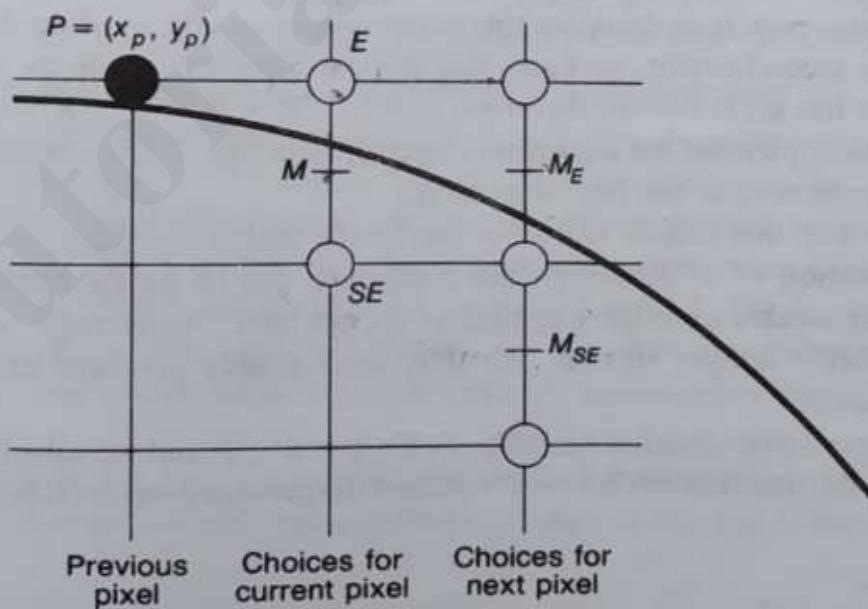


**Fig. 3.13** Eight symmetrical points on a circle.

### 3.3.2 Midpoint Circle Algorithm

Bresenham [BRES77] developed an incremental circle generator that is more efficient than the methods we have discussed. Conceived for use with pen plotters, the algorithm generates all points on a circle centered at the origin by incrementing all the way around the circle. We derive a similar algorithm, again using the midpoint criterion, which, for the case of integer center point and radius, generates the same, optimal set of pixels. Furthermore, the resulting code is essentially the same as that specified in patent 4,371,933 [BRES83].

We consider only  $45^\circ$  of a circle, the second octant from  $x = 0$  to  $x = y = R/\sqrt{2}$ , and use the CirclePoints procedure to display points on the entire circle. As with the midpoint line algorithm, the strategy is to select which of 2 pixels is closer to the circle by evaluating a function at the midpoint between the 2 pixels. In the second octant, if pixel  $P$  at  $(x_p, y_p)$  has been previously chosen as closest to the circle, the choice of the next pixel is between pixel  $E$  and  $SE$  (see Fig. 3.14).



**Fig. 3.14** The pixel grid for the midpoint circle algorithm showing  $M$  and the pixels  $E$  and  $SE$  to choose between.

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

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

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

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

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

and  $d_{\text{new}} = d_{\text{old}} + (2x_p + 3)$ ; therefore, the increment  $\Delta_E = 2x_p + 3$ .

If  $d_{\text{old}} \geq 0$ ,  $SE$  is chosen,<sup>6</sup> and the next midpoint will be one increment over in  $x$  and one increment down in  $y$ . Then

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

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

Recall that, in the linear case,  $\Delta_E$  and  $\Delta_{NE}$  were constants; in the quadratic case, however,  $\Delta_E$  and  $\Delta_{SE}$  vary at each step and are functions of the particular values of  $x_p$  and  $y_p$  at the pixel chosen in the previous iteration. Because these functions are expressed in terms of  $(x_p, y_p)$ , we call  $P$  the *point of evaluation*. The  $\Delta$  functions can be evaluated directly at each step by plugging in the values of  $x$  and  $y$  for the pixel chosen in the previous iteration. This direct evaluation is not expensive computationally, since the functions are only linear.

In summary, we do the same two steps at each iteration of the algorithm as we did for the line: (1) choose the pixel based on the sign of the variable  $d$  computed during the previous iteration, and (2) update the decision variable  $d$  with the  $\Delta$  that corresponds to the choice of pixel. The only difference from the line algorithm is that, in updating  $d$ , we evaluate a linear function of the point of evaluation.

All that remains now is to compute the initial condition. By limiting the algorithm to integer radii in the second octant, we know that the starting pixel lies on the circle at  $(0, R)$ . The next midpoint lies at  $(1, R - \frac{1}{2})$ , therefore, and  $F(1, R - \frac{1}{2}) = 1 + (R^2 - R + \frac{1}{4}) - R^2 = \frac{5}{4} - R$ . Now we can implement the algorithm directly, as in Fig. 3.15. Notice how similar in structure this algorithm is to the line algorithm.

The problem with this version is that we are forced to do real arithmetic because of the fractional initialization of  $d$ . Although the procedure can be easily modified to handle circles that are not located on integer centers or do not have integer radii, we would like a more efficient, purely integer version. We thus do a simple program transformation to eliminate fractions.

First, we define a new decision variable,  $h$ , by  $h = d - \frac{1}{4}$ , and we substitute  $h + \frac{1}{4}$  for  $d$  in the code. Now, the initialization is  $h = 1 - R$ , and the comparison  $d < 0$  becomes  $h < -\frac{1}{4}$ .

<sup>6</sup>Choosing  $SE$  when  $d = 0$  differs from our choice in the line algorithm and is arbitrary. The reader may wish to simulate the algorithm by hand to see that, for  $R = 17$ , 1 pixel is changed by this choice.

```

void MidpointCircle (int radius, int value)
/* Assumes center of circle is at origin */
{
    int x = 0;
    int y = radius;
    double d = 5.0 / 4.0 - radius;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0)          /* Select E */
            d += 2.0 * x + 3.0;
        else {                /* Select SE */
            d += 2.0 * (x - y) + 5.0;
            y--;
        }
        x++;
        CirclePoints (x, y, value);
    } /* while */
} /* MidpointCircle */

```

**Fig. 3.15** The midpoint circle scan-conversion algorithm.

However, since  $h$  starts out with an integer value and is incremented by integer values ( $\Delta_E$  and  $\Delta_{SE}$ ), we can change the comparison to just  $h < 0$ . We now have an integer algorithm in terms of  $h$ ; for consistency with the line algorithm, we will substitute  $d$  for  $h$  throughout. The final, fully integer algorithm is shown in Fig. 3.16.

Figure 3.17 shows the second octant of a circle of radius 17 generated with the algorithm, and the first octant generated by symmetry (compare the results to Fig. 3.12).

### 3.4 SCAN CONVERTING ELLIPSES

Consider the standard ellipse of Fig. 3.19, centered at  $(0, 0)$ . It is described by the equation

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

where  $2a$  is the length of the major axis along the  $x$  axis, and  $2b$  is the length of the minor axis along the  $y$  axis. The midpoint technique discussed for lines and circles can also be applied to the more general conics. In this chapter, we consider the standard ellipse that is supported by SRGP; in Chapter 19, we deal with ellipses at any angle. Again, to simplify the algorithm, we draw only the arc of the ellipse that lies in the first quadrant, since the other three quadrants can be drawn by symmetry. Note also that standard ellipses centered at integer points other than the origin can be drawn using a simple translation. The algorithm presented here is based on Da Silva's algorithm, which combines the techniques used by Pitteway [PITT67], Van Aken [VANA84] and Kappel [KAPP85] with the use of partial differences [DASI89].

We first divide the quadrant into two regions; the boundary between the two regions is the point at which the curve has a slope of  $-1$  (see Fig. 3.20).

Determining this point is more complex than it was for circles, however. The vector that is perpendicular to the tangent to the curve at point  $P$  is called the *gradient*, defined as

$$\text{grad } F(x, y) = \frac{\partial F}{\partial x} \mathbf{i} + \frac{\partial F}{\partial y} \mathbf{j} = 2b^2x \mathbf{i} + 2a^2y \mathbf{j}.$$

The boundary between the two regions is the point at which the slope of the curve is  $-1$ , and that point occurs when the gradient vector has a slope of  $1$ —that is, when the  $\mathbf{i}$  and  $\mathbf{j}$  components of the gradient are of equal magnitude. The  $\mathbf{j}$  component of the gradient is larger than the  $\mathbf{i}$  component in region 1, and vice versa in region 2. Thus, if at the next midpoint,  $a^2(y_p - \frac{1}{2}) \leq b^2(x_p + 1)$ , we switch from region 1 to region 2.

As with any midpoint algorithm, we evaluate the function at the midpoint between two pixels and use the sign to determine whether the midpoint lies inside or outside the ellipse and, hence, which pixel lies closer to the ellipse. Therefore, in region 1, if the current pixel is located at  $(x_p, y_p)$ , then the decision variable for region 1,  $d_1$ , is  $F(x, y)$  evaluated at  $(x_p + 1, y_p - \frac{1}{2})$ , the midpoint between  $E$  and  $SE$ . We now repeat the process we used for deriving the

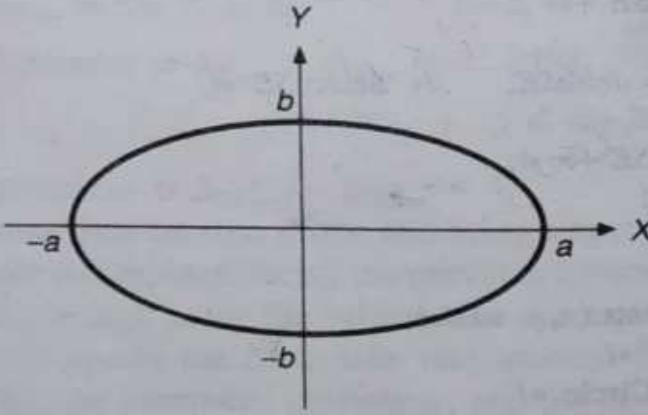
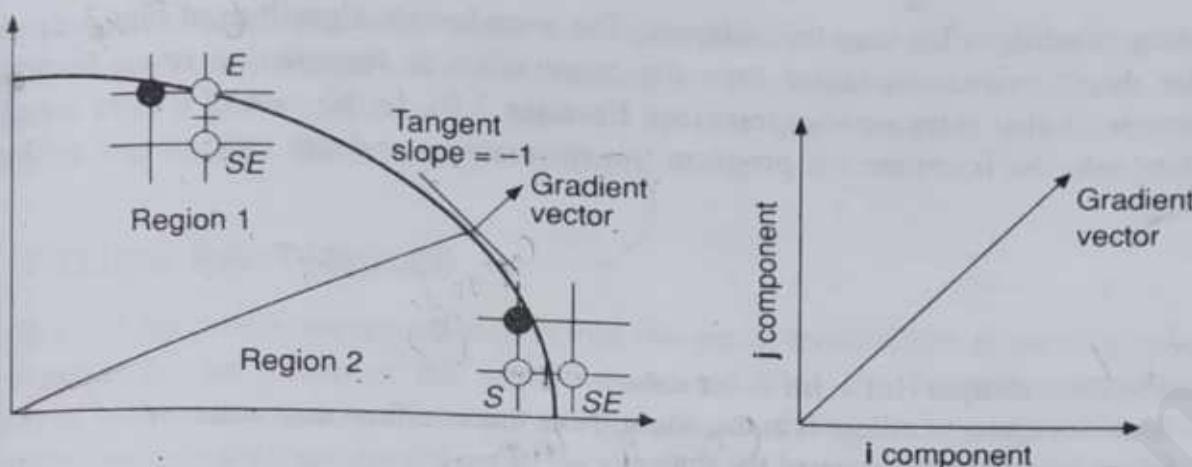


Fig. 3.19 Standard ellipse centered at the origin.



**Fig. 3.20** Two regions of the ellipse defined by the  $45^\circ$  tangent.

two  $\Delta$ s for the circle. For a move to  $E$ , the next midpoint is one increment over in  $x$ . Then,

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

$$d_{\text{new}} = F(x_p + 2, y_p - \frac{1}{2}) = b^2(x_p + 2)^2 + a^2(y_p - \frac{1}{2})^2 - a^2b^2.$$

Since  $d_{\text{new}} = d_{\text{old}} + b^2(2x_p + 3)$ , the increment  $\Delta_E = b^2(2x_p + 3)$ .

For a move to  $SE$ , the next midpoint is one increment over in  $x$  and one increment down in  $y$ . Then,

$$d_{\text{new}} = F(x_p + 2, y_p - \frac{3}{2}) = b^2(x_p + 2)^2 + a^2(y_p - \frac{3}{2})^2 - a^2b^2.$$

Since  $d_{\text{new}} = d_{\text{old}} + b^2(2x_p + 3) + a^2(-2y_p + 2)$ , the increment  $\Delta_{SE} = b^2(2x_p + 3) + a^2(-2y_p + 2)$ .

In region 2, if the current pixel is at  $(x_p, y_p)$ , the decision variable  $d_2$  is  $F(x_p + \frac{1}{2}, y_p - 1)$ , the midpoint between  $S$  and  $SE$ . Computations similar to those given for region 1 may be done for region 2.

We must also compute the initial condition. Assuming integer values  $a$  and  $b$ , the ellipse starts at  $(0, b)$ , and the first midpoint to be calculated is at  $(1, b - \frac{1}{2})$ . Then,

$$F(1, b - \frac{1}{2}) = b^2 + a^2(b - \frac{1}{2})^2 - a^2b^2 = b^2 + a^2(-b + \frac{1}{4}).$$

At every iteration in region 1, we must not only test the decision variable  $d_1$  and update the  $\Delta$  functions, but also see whether we should switch regions by evaluating the gradient at the midpoint between  $E$  and  $SE$ . When the midpoint crosses over into region 2, we change our choice of the 2 pixels to compare from  $E$  and  $SE$  to  $SE$  and  $S$ . At the same time, we have to initialize the decision variable  $d_2$  for region 2 to the midpoint between  $SE$  and  $S$ . That is, if the last pixel chosen in region 1 is located at  $(x_p, y_p)$ , then the decision variable  $d_2$  is initialized at  $(x_p + \frac{1}{2}, y_p - 1)$ . We stop drawing pixels in region 2 when the  $y$  value of the pixel is equal to 0.

As with the circle algorithm, we can either calculate the  $\Delta$  functions directly in each iteration of the loop or compute them with differences. Da Silva shows that computation of second-order partials done for the  $\Delta$ s can, in fact, be used for the gradient as well [DASI89]. He also treats general ellipses that have been rotated and the many tricky

boundary conditions for very thin ellipses. The pseudocode algorithm of Fig. 3.21 uses the simpler direct evaluation rather than the more efficient formulation using second-order differences; it also skips various tests (see Exercise 3.9). In the case of integer  $a$  and  $b$ , we can eliminate the fractions via program transformations and use only integer arithmetic.

```

void MidpointEllipse (int a, int b, int value)
/* Assumes center of ellipse is at the origin. Note that overflow may occur */
/* for 16-bit integers because of the squares. */
{
    double d2;

    int x = 0;
    int y = b;
    double d1 = b2 - (a2b) + (0.25 a2);
    EllipsePoints (x, y, value); /* The 4-way symmetrical WritePixel */

    /* Test gradient if still in region 1 */
    while ( a2(y - 0.5) > b2(x + 1) ) { /* Region 1 */
        if (d1 < 0) /* Select E */
            d1 += b2(2x + 3);
        else { /* Select SE */
            d1 += b2(2x + 3) + a2(-2y + 2);
            y--;
        }
        x++;
        EllipsePoints (x, y, value);
    } /* Region 1 */

    d2 = b2(x + 0.5)2 + a2(y - 1)2 - a2b2;
    while (y > 0) { /* Region 2 */
        if (d2 < 0) /* Select SE */
            d2 += b2(2x + 2) + a2(-2y + 3);
        x++;
        else
            d2 += a2(-2y + 3); /* Select S */
        y--;
        EllipsePoints (x, y, value);
    } /* Region 2 */
} /* MidpointEllipse */

```

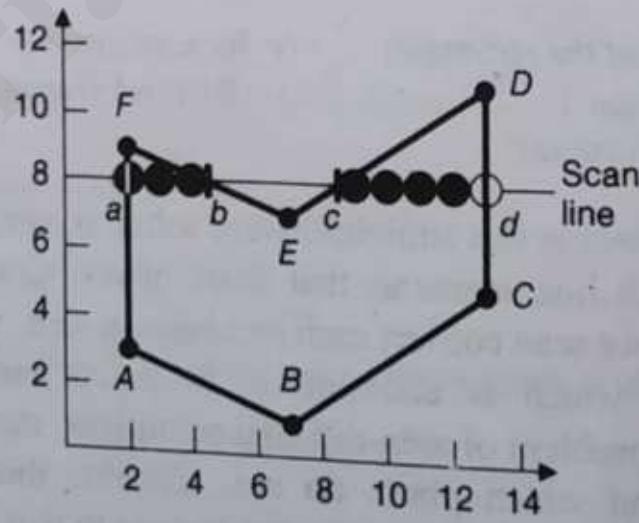
**Fig. 3.21** Pseudocode for midpoint ellipse scan-conversion algorithm.

Now that we have seen how to scan convert lines 1 pixel thick as well as unfilled primitives, we turn our attention to modifications of these algorithms that fill area-defining primitives with a solid color or a pattern, or that draw unfilled primitives with a combination of the line-width and pen-style attributes.

### 3.6 FILLING POLYGONS

The general polygon scan-conversion algorithm described next handles both convex and concave polygons, even those that are self-intersecting or have interior holes. It operates by computing spans that lie between left and right edges of the polygon. The span extrema are calculated by an incremental algorithm that computes a scan line/edge intersection from the intersection with the previous scan line. Figure 3.22, which illustrates the basic polygon scan-conversion process, shows a polygon and one scan line passing through it. The intersections of scan line 8 with edges *FA* and *CD* lie on integer coordinates, whereas those for *EF* and *DE* do not; the intersections are marked in the figure by vertical tick marks labeled *a* through *d*.

We must determine which pixels on each scan line are within the polygon, and we must set the corresponding pixels (in this case, spans from  $x = 2$  through 4 and 9 through 13) to



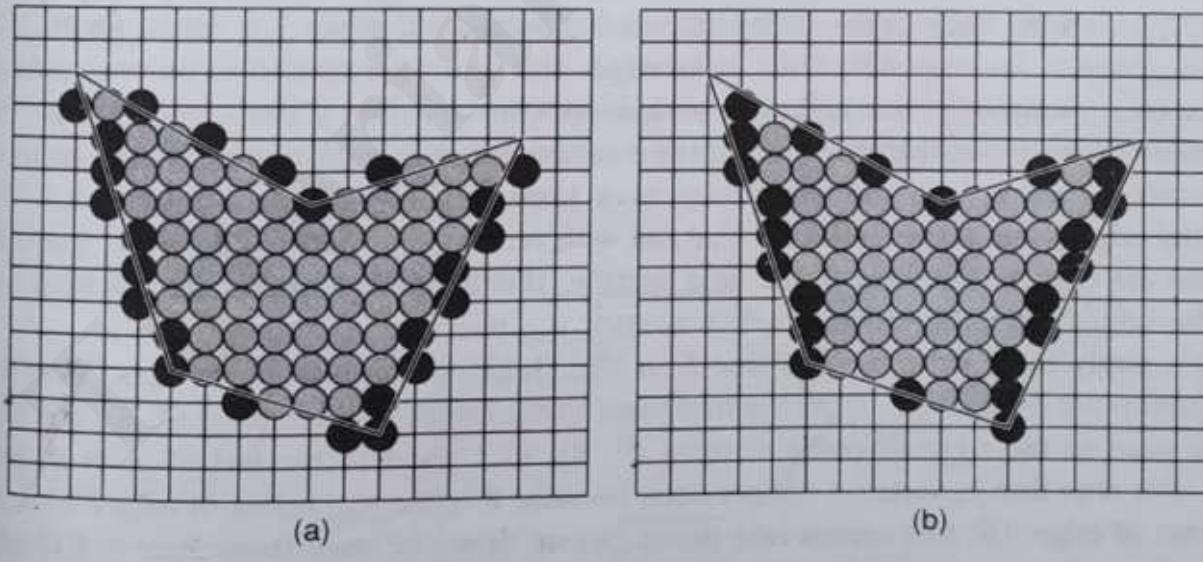
**Fig. 3.22** Polygon and scan line 8.

their appropriate values. By repeating this process for each scan line that intersects the polygon, we can convert the entire polygon, as shown for another polygon in Fig. 3.23.

Figure 3.23(a) shows the pixels defining the extrema of spans in black and the interior pixels on the span in gray. A straightforward way of deriving the extrema is to use the midpoint line scan-conversion algorithm on each edge and to keep a table of span extrema for each scan line, updating an entry if a new pixel is produced for an edge that extends the span. Note that this strategy produces some extrema pixels that lie outside the polygon; they were chosen by the scan-conversion algorithm because they lie closest to an edge, without regard to the side of the edge on which they lie—the line algorithm has no notions of interior and exterior. We do not want to draw such pixels on the outside of a shared edge, however, because they would intrude into the regions of neighboring polygons, and this would look odd if these polygons had different colors. It is obviously preferable to draw only those pixels that are strictly interior to the region, even when an exterior pixel would be closer to the edge. We must therefore adjust the scan-conversion algorithm accordingly; compare Fig. 3.23 (a) with Fig. 3.23 (b), and note that a number of pixels outside the ideal primitive are not drawn in part (b).

With this technique, a polygon does not intrude (even by a single pixel) into the regions defined by other primitives. We can apply the same technique to unfilled polygons for consistency or can choose to scan convert rectangles and polygons a line segment at a time, in which case unfilled and filled polygons do not contain the same boundary pixels!

As with the original midpoint algorithm, we use an incremental algorithm to calculate the span extrema on one scan line from those at the previous scan line without having to compute the intersections of a scan line with each polygon edge analytically. In scan line 8 of Fig. 3.22, for instance, there are two spans of pixels within the polygon. The spans can be filled in by a three-step process:



- Span extrema
- Other pixels in the span

**Fig. 3.23** Spans for a polygon. Extrema shown in black, interior pixels in gray.  
 (a) Extrema computed by midpoint algorithm. (b) Extrema interior to polygon.

1. Find the intersections of the scan line with all edges of the polygon.
2. Sort the intersections by increasing  $x$  coordinate.
3. Fill in all pixels between pairs of intersections that lie interior to the polygon, using the odd-parity rule to determine that a point is inside a region: Parity is initially even, and each intersection encountered thus inverts the parity bit—draw when parity is odd, do not draw when it is even.

The first two steps of the process, finding intersections and sorting them, are treated in the next section. Let's look now at the span-filling strategy. In Fig. 3.22, the sorted list of  $x$  coordinates is (2, 4.5, 8.5, 13). Step 3 requires four elaborations:

- 3.1 Given an intersection with an arbitrary, fractional  $x$  value, how do we determine which pixel on either side of that intersection is interior?
- 3.2 How do we deal with the special case of intersections at integer pixel coordinates?
- 3.3 How do we deal with the special case in 3.2 for shared vertices?
- 3.4 How do we deal with the special case in 3.2 in which the vertices define a horizontal edge?

To handle case 3.1, we say that, if we are approaching a fractional intersection to the right and are inside the polygon, we round down the  $x$  coordinate of the intersection to define the interior pixel; if we are outside the polygon, we round up to be inside. We handle case 3.2 by applying the criterion we used to avoid conflicts at shared edges of rectangles: If the leftmost pixel in a span has integer  $x$  coordinate, we define it to be interior; if the rightmost pixel has integer  $x$  coordinate, we define it to be exterior. For case 3.3, we count the  $y_{\min}$  vertex of an edge in the parity calculation but not the  $y_{\max}$  vertex; therefore, a  $y_{\max}$  vertex is drawn only if it is the  $y_{\min}$  vertex for the adjacent edge. Vertex A in Fig. 3.22, for example, is counted once in the parity calculation because it is the  $y_{\min}$  vertex for edge FA but the  $y_{\max}$  vertex for edge AB. Thus, both edges and spans are treated as intervals that are closed at their minimum value and open at their maximum value. Clearly, the opposite rule would work as well, but this rule seems more natural since it treats the minimum endpoint as an entering point, and the maximum as a leaving point. When we treat case 3.4, horizontal edges, the desired effect is that, as with rectangles, bottom edges are drawn but top edges are not. As we show in the next section, this happens automatically if we do not count the edges' vertices, since they are neither  $y_{\min}$  nor  $y_{\max}$  vertices.

Let's apply these rules to scan line 8 in Fig. 3.22, which hits no vertices. We fill in pixels from point  $a$ , pixel (2, 8), to the first pixel to the left of point  $b$ , pixel (4, 8), and from the first pixel to the right of point  $c$ , pixel (9, 8), to 1 pixel to the left of point  $d$ , pixel (12, 8). For scan line 3, vertex A counts once because it is the  $y_{\min}$  vertex of edge FA but the  $y_{\max}$  vertex of edge AB; this causes odd parity, so we draw the span from there to 1 pixel to the left of the intersection with edge CB, where the parity is set to even and the span is terminated. Scan line 1 hits only vertex B; edges AB and BC both have their  $y_{\min}$  vertices at B, which is therefore counted twice and leaves the parity even. This vertex acts as a null span—enter at the vertex, draw the pixel, exit at the vertex. Although such local minima

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

draw a single pixel, no pixel is drawn at a local maximum, such as the intersection of scan line 9 with the vertex  $F$ , shared by edges  $FA$  and  $EF$ . Both vertices are  $y_{\max}$  vertices and therefore do not affect the parity, which stays even.

### 3.6.1 Horizontal Edges

We deal properly with horizontal edges by not counting their vertices, as we can see by examining various cases in Fig. 3.24. Consider bottom edge  $AB$ . Vertex  $A$  is a  $y_{\min}$  vertex for edge  $JA$ , and  $AB$  does not contribute. Therefore, the parity is odd and the span  $AB$  is drawn. Vertical edge  $BC$  has its  $y_{\min}$  at  $B$ , but again  $AB$  does not contribute. The parity becomes even and the span is terminated. At vertex  $J$ , edge  $IJ$  has a  $y_{\min}$  vertex but edge  $JA$  does not, so the parity becomes odd and the span is drawn to edge  $BC$ . The span that starts at edge  $IJ$  and hits  $C$  sees no change at  $C$  because  $C$  is a  $y_{\max}$  vertex for  $BC$ , so the span continues along bottom edge  $CD$ ; at  $D$ , however, edge  $DE$  has a  $y_{\min}$  vertex, so the parity is reset to even and the span ends. At  $I$ , edge  $IJ$  has its  $y_{\max}$  vertex and edge  $HI$  also does not contribute, so parity stays even and the top edge  $IH$  is not drawn. At  $H$ , however, edge  $GH$  has a  $y_{\min}$  vertex, the parity becomes odd, and the span is drawn from  $H$  to the pixel to the left of the intersection with edge  $EF$ . Finally, there is no  $y_{\min}$  vertex at  $G$ , nor is there one at  $F$ , so top edge  $FG$  is not drawn.

The algorithm above deals with shared vertices in a polygon, with edges shared by two adjacent polygons, and with horizontal edges. It allows self-intersecting polygons. As noted, it does not work perfectly in that it omits pixels. Worse, it cannot totally avoid writing shared pixels multiple times without keeping a history: Consider edges shared by more than two polygons or a  $y_{\min}$  vertex shared by two otherwise disjoint triangles (see Exercise 3.14).

### 3.6.2 Slivers

There is another problem with our scan-conversion algorithm that is not resolved as satisfactorily as is that of horizontal edges: polygons with edges that lie sufficiently close together create a *sliver*—a polygonal area so thin that its interior does not contain a distinct span for each scan line. Consider, for example, the triangle from  $(0, 0)$  to  $(3, 12)$  to  $(5, 12)$  to  $(0, 0)$ , shown in Fig. 3.25. Because of the rule that only pixels that lie interior or on a left

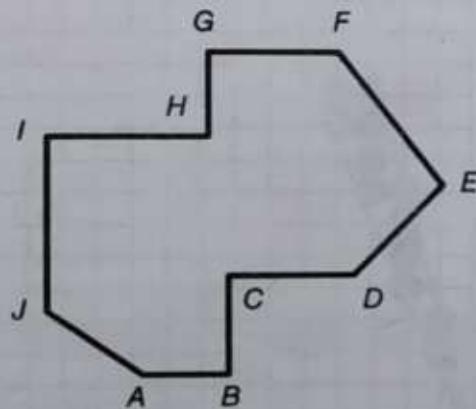


Fig. 3.24 Horizontal edges in a polygon.

or bottom edge are drawn, there will be many scan lines with only a single pixel or no pixels. The problem of having “missing” pixels is yet another example of the aliasing problem; that is, of representing a continuous signal with a discrete approximation. If we had multiple bits per pixel, we could use antialiasing techniques, as introduced for lines in Section 3.17 and for polygons in Section 19.3. Antialiasing would involve softening our rule “draw only pixels that lie interior or on a left or bottom edge” to allow boundary pixels and even exterior pixels to take on intensity values that vary as a function of distance between a pixel’s center and the primitive; multiple primitives can then contribute to a pixel’s value.

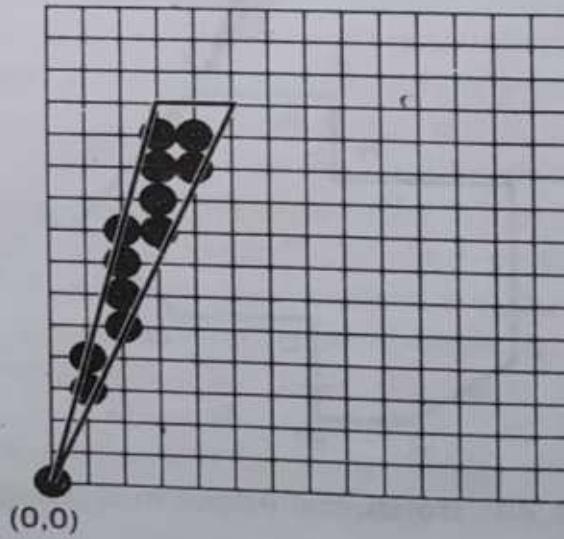
### 3.6.3 Edge Coherence and the Scan-Line Algorithm

Step 1 in our procedure—calculating intersections—must be done cleverly lest it be slow. In particular, we must avoid the brute-force technique of testing each polygon edge for intersection with each new scan line. Very often, only a few of the edges are of interest for a given scan line. Furthermore, we note that many edges intersected by scan line  $i$  are also intersected by scan line  $i + 1$ . This *edge coherence* occurs along an edge for as many scan lines as intersect the edge. As we move from one scan line to the next, we can compute the new  $x$  intersection of the edge on the basis of the old  $x$  intersection, just as we computed the next pixel from the current pixel in midpoint line scan conversion, by using

$$x_{i+1} = x_i + 1/m,$$

where  $m$  is the slope of the edge. In the midpoint algorithm for scan converting lines, we avoided fractional arithmetic by computing an integer decision variable and checking only its sign to choose the pixel closest to the mathematical line; here, we would like to use integer arithmetic to do the required rounding for computing the closest interior pixel.

Consider lines with a slope greater than +1 that are left edges; right edges and other slopes are handled by similar, though somewhat trickier, arguments, and vertical edges are special cases. (Horizontal edges are handled implicitly by the span rules, as we saw.) At the  $(x_{\min}, y_{\min})$  endpoint, we need to draw a pixel. As  $y$  is incremented, the  $x$  coordinate of the point on the ideal line will increase by  $1/m$ , where  $m = (y_{\max} - y_{\min})/(x_{\max} - x_{\min})$  is the



**Fig. 3.25** Scan converting slivers of polygons.

slope of the line. This increase will result in  $x$  having an integer and a fractional part, which can be expressed as a fraction with a denominator of  $y_{\max} - y_{\min}$ . As we iterate this process, the fractional part will overflow and the integer part will have to be incremented. For example, if the slope is  $\frac{5}{2}$ , and  $x_{\min}$  is 3, then the sequence of  $x$  values will be 3,  $3\frac{2}{5}$ ,  $3\frac{4}{5}$ ,  $3\frac{6}{5} = 4\frac{1}{5}$ , and so on. When the fractional part of  $x$  is zero, we can draw the pixel  $(x, y)$  that lies on the line, but when the fractional part of  $x$  is nonzero, we need to round up in order to get a pixel that lies strictly inside the line. When the fractional part of  $x$  becomes greater than 1, we increment  $x$  and subtract 1 from the fractional part; we must also move 1 pixel to the right. If we increment to lie exactly on a pixel, we draw that pixel but must decrement the fraction by 1 to have it be less than 1.

We can avoid the use of fractions by keeping track only of the numerator of the fraction and observing that the fractional part is greater than 1 when the numerator is greater than the denominator. We implement this technique in the algorithm of Fig. 3.26, using the variable *increment* to keep track of successive additions of the numerator until it "overflows" past the denominator, when the numerator is decremented by the denominator and  $x$  is incremented.

We now develop a *scan-line algorithm* that takes advantage of this edge coherence and, for each scan line, keeps track of the set of edges it intersects and the intersection points in a data structure called the *active-edge table* (AET). The edges in the AET are sorted on their  $x$  intersection values so that we can fill the spans defined by pairs of (suitably rounded) intersection values—that is, the span extrema. As we move to the next scan line at  $y + 1$ , the AET is updated. First, edges currently in the AET but not intersected by this next scan line (i.e., those whose  $y_{\max} = y$ ) are deleted. Second, any new edges intersected by this next scan line (i.e., those edges whose  $y_{\min} = y + 1$ ) are added to the AET. Finally, new  $x$  intersections are calculated, using the preceding incremental edge algorithm, for edges that were in the AET but are not yet completed.

```

void LeftEdgeScan (int xmin, int ymin, int xmax, int ymax, int value)
{
    int y;

    int x = xmin;
    int numerator = xmax - xmin;
    int denominator = ymax - ymin;
    int increment = denominator;

    for (y = ymin; y <= ymax; y++) {
        WritePixel (x, y, value);
        increment += numerator;
        if (increment > denominator) {
            /* Overflow, so round up to next pixel and decrement the increment. */
            x++;
            increment -= denominator;
        }
    }
} /* LeftEdgeScan */

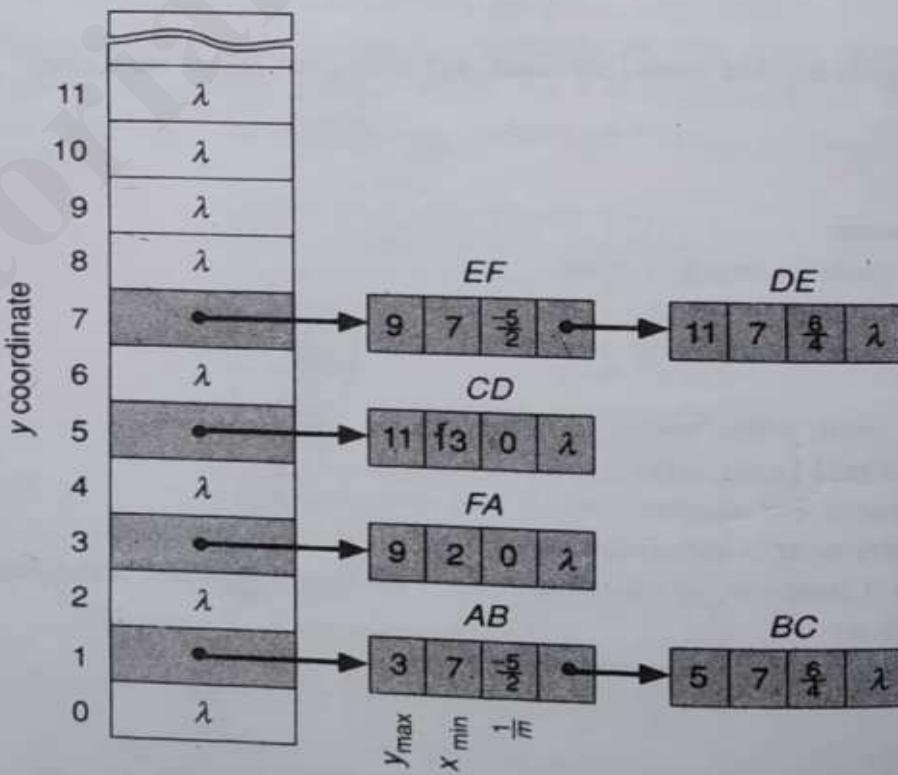
```

**Figure 3.26** Scan converting left edge of a polygon.

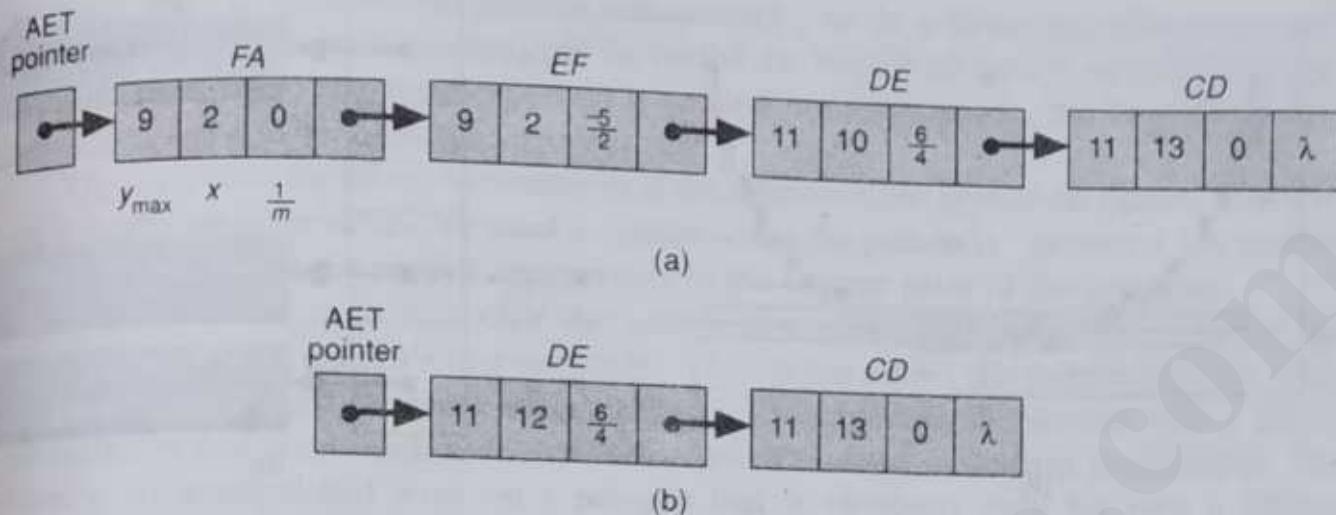
To make the addition of edges to the AET efficient, we initially create a global edge table (ET) containing all edges sorted by their smaller  $y$  coordinate. The ET is typically built by using a bucket sort with as many buckets as there are scan lines. Within each bucket, edges are kept in order of increasing  $x$  coordinate of the lower endpoint. Each entry in the ET contains the  $y_{max}$  coordinate of the edge, the  $x$  coordinate of the bottom endpoint ( $x_{min}$ ), and the  $x$  increment used in stepping from one scan line to the next,  $1/m$ . Figure 3.27 shows how the six edges from the polygon of Fig. 3.22 would be sorted, and Fig. 3.28 shows the AET at scan lines 9 and 10 for that polygon. (In an actual implementation, we would probably add a flag indicating left or right edge.)

Once the ET has been formed, the processing steps for the scan-line algorithm are as follows:

1. Set  $y$  to the smallest  $y$  coordinate that has an entry in the ET; i.e.,  $y$  for the first nonempty bucket
2. Initialize the AET to be empty
3. Repeat until the AET and ET are empty:
  - 3.1 Move from ET bucket  $y$  to the AET those edges whose  $y_{min} = y$  (entering edges).
  - 3.2 Remove from the AET those entries for which  $y = y_{max}$  (edges not involved in the next scan line), then sort the AET on  $x$  (made easier because ET is presorted).
  - 3.3 Fill in desired pixel values on scan line  $y$  by using pairs of  $x$  coordinates from the AET
  - 3.4 Increment  $y$  by 1 (to the coordinate of the next scan line)
  - 3.5 For each nonvertical edge remaining in the AET, update  $x$  for the new  $y$



**Fig. 3.27** Bucket-sorted edge table for polygon of Fig. 3.22.



**Fig. 3.28** Active-edge table for polygon of Fig. 3.22. (a) Scan line 9. (b) Scan line 10  
(Note  $DE$ 's  $x$  coordinate in (b) has been rounded up for that left edge.)

This algorithm uses both edge coherence to calculate  $x$  intersections and scan-line coherence (along with sorting) to calculate spans. Since the sorting works on a small number of edges and since the resorting of step 3.1 is applied to a mostly or completely sorted list, either insertion sort or a simple bubble sort that is  $O(N)$  in this case may be used. In Chapters 15 and 16, we see how to extend this algorithm to handle multiple polygons during visible-surface determination, including the case of handling polygons that are transparent; in Chapter 17, we see how to blend polygons that overlap at a pixel.

For purposes of scan conversion, triangles and trapezoids can be treated as special cases of polygons, since they have only two edges for any scan line (given that horizontal edges are not scan-converted explicitly). Indeed, since an arbitrary polygon can be decomposed into a mesh of triangles sharing vertices and edges (see Exercise 3.17), we could scan convert general polygons by first decomposing them into triangle meshes, and then scan converting the component triangles. Such triangulation is a classic problem in computational geometry [PREP85] and is easy to do for convex polygons; doing it efficiently for nonconvex polygons is difficult.

Note that the calculation of spans is cumulative. That is, when the current iteration of the scan-conversion algorithm in Step 3.5 generates multiple pixels falling on the same scan line, the span extrema must be updated appropriately. (Dealing with span calculations for edges that cross and for slivers takes a bit of special casing.) We can either compute all spans in one pass, then fill the spans in a second pass, or compute a span and fill it when completed. Another benefit of using spans is that clipping can be done at the same time as span arithmetic: The spans may be individually clipped at the left and right coordinates of the clip rectangle. Note that, in Section 15.10.3 we use a slightly different version of span arithmetic to combine 3D solid objects that are rendered using "raytracing."

### 3.9 THICK PRIMITIVES

Conceptually, we produce thick primitives by tracing the scan-converted single-pixel outline primitive. We place the center of a brush of a specified cross-section (or another distinguished point, such as the upper-left corner of a rectangular brush) at each pixel chosen by the scan-conversion algorithm. A single-pixel-wide line can be conceived as being drawn with a brush the size of a single pixel. However, this simple description masks a number of tricky questions. First, what shape is the brush? Typical implementations use circular and rectangular brushes. Second, what is the orientation of a noncircular brush? Does the rectangular pen always stay upright, so that the brush has constant width, or does it turn as the primitive turns, so that the vertical axis of the brush is aligned with the tangent to the primitive? What do the ends of a thick line look like, both ideally and on the integer grid? What happens at the vertex of a thick polygon? How do line style and pen style interact? We shall answer the simpler questions in this section, and the others in Chapter 19.

There are four basic methods for drawing thick primitives, illustrated in Figs. 3.31 through 3.36. We show the ideal primitives for these lines in black-on-white outline; the pixels generated to define the 1-pixel-thick scan-converted primitive in black; and the pixels added to form the thick primitive in gray. The reduced-scale versions show what the thick primitive actually looks like at still rather low resolution, with all pixels set to black. The first method is a crude approximation that uses more than 1 pixel for each column (or row) during scan conversion. The second traces the pen's cross-section along the single-pixel outline of the primitive. The third draws two copies of a primitive a thickness  $t$  apart and fills in the spans between these inner and outer boundaries. The fourth approximates all primitives by polylines and then uses a thick line for each polyline segment.

Let's look briefly at each of these methods and consider its advantages and disadvantages. All the methods produce effects that are satisfactory for many, if not most, purposes, at least for viewing on the screen. For printing, the higher resolution should be used to good advantage, especially since the speed of an algorithm for printing is not as

critical as for online primitive generation. We can then use more complex algorithms to produce better-looking results. A package may even use different techniques for different primitives. For example, QuickDraw traces an upright rectangular pen for lines, but fills spans between confocal ellipse boundaries.

### 3.9.1 Replicating Pixels

A quick extension to the scan-conversion inner loop to write multiple pixels at each computed pixel works reasonably well for lines; here, pixels are duplicated in columns for lines with  $-1 < \text{slope} < 1$  and in rows for all other lines. The effect, however, is that the line ends are always vertical or horizontal, which is not pleasing for rather thick lines, as Fig. 3.31 shows.

The pixel-replication algorithm also produces noticeable gaps in places where line segments meet at an angle, and misses pixels where there is a shift from horizontal to vertical replication as a function of the slope. This latter anomaly shows up as abnormal thinness in ellipse arcs at the boundaries between octants, as in Fig. 3.32.

Furthermore, lines that are horizontal and vertical have a different thickness from lines at an angle, where the *thickness* of the primitive is defined as the distance between the primitive's boundaries perpendicular to its tangent. Thus, if the thickness parameter is  $t$ , a horizontal or vertical line has thickness  $t$ , whereas one drawn at  $45^\circ$  has an average thickness of  $t/\sqrt{2}$ . This is another result of having fewer pixels in the line at an angle, as first noted in Section 3.2.3; it decreases the brightness contrast with horizontal and vertical lines of the same thickness. Still another problem with pixel replication is the generic problem of even-numbered widths: We cannot center the duplicated column or row about the selected pixel, so we must choose a side of the primitive to have an "extra" pixel. Altogether, pixel replication is an efficient but crude approximation that works best for primitives that are not very thick.

### 3.9.2 The Moving Pen

Choosing a rectangular pen whose center or corner travels along the single-pixel outline of the primitive works reasonably well for lines; it produces the line shown in Fig. 3.33. Notice that this line is similar to that produced by pixel replication but is thicker at the endpoints. As with pixel replication, because the pen stays vertically aligned, the perceived thickness of the primitive varies as a function of the primitive's angle, but in the opposite

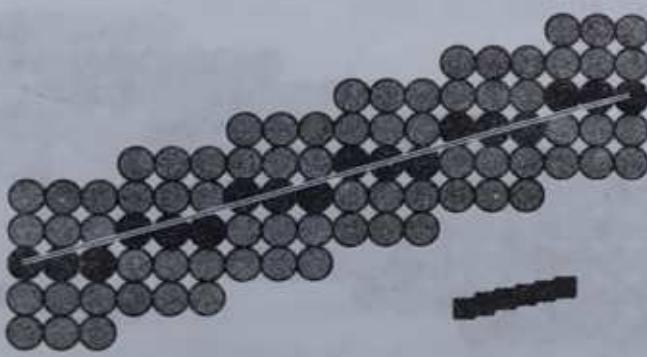
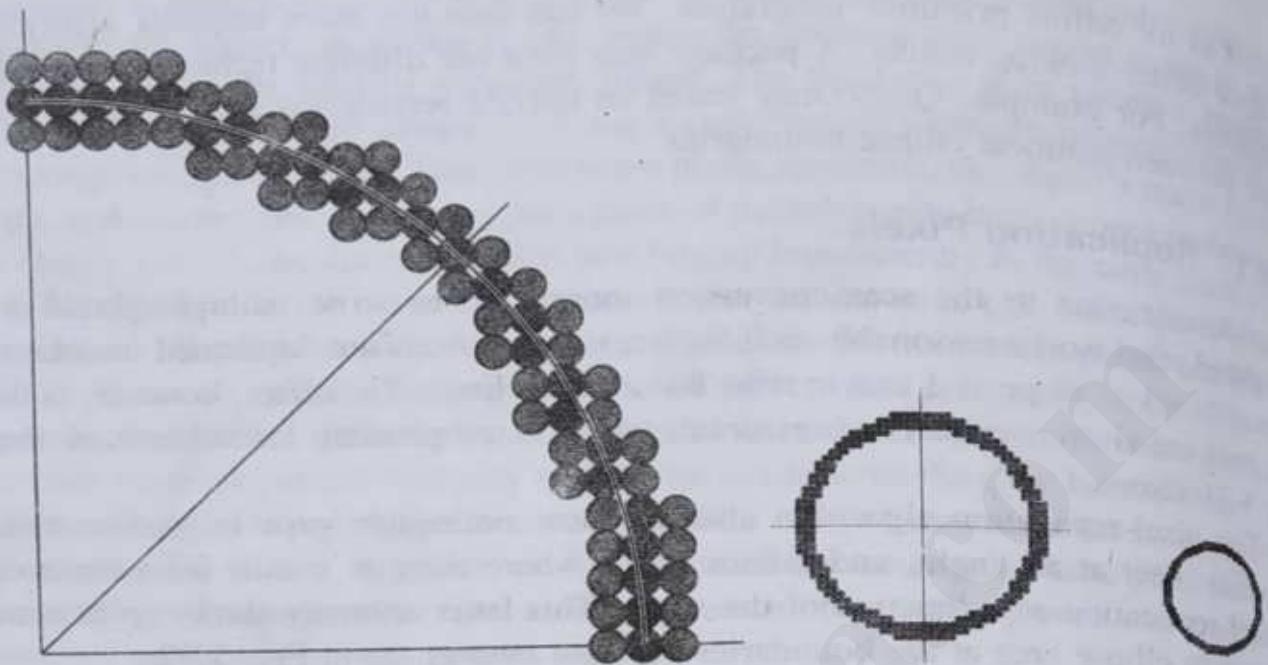


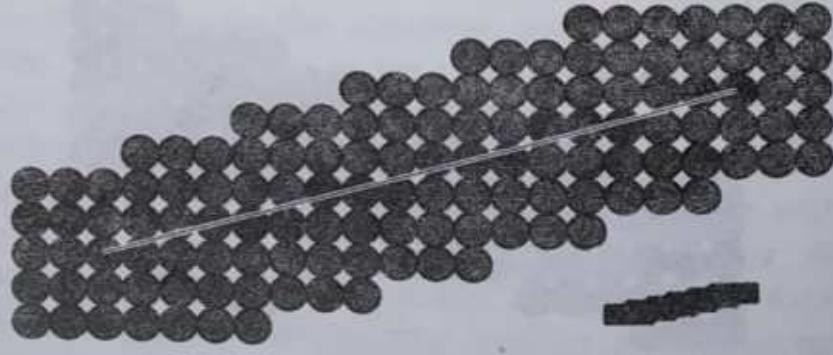
Fig. 3.31 Thick line drawn by column replication.



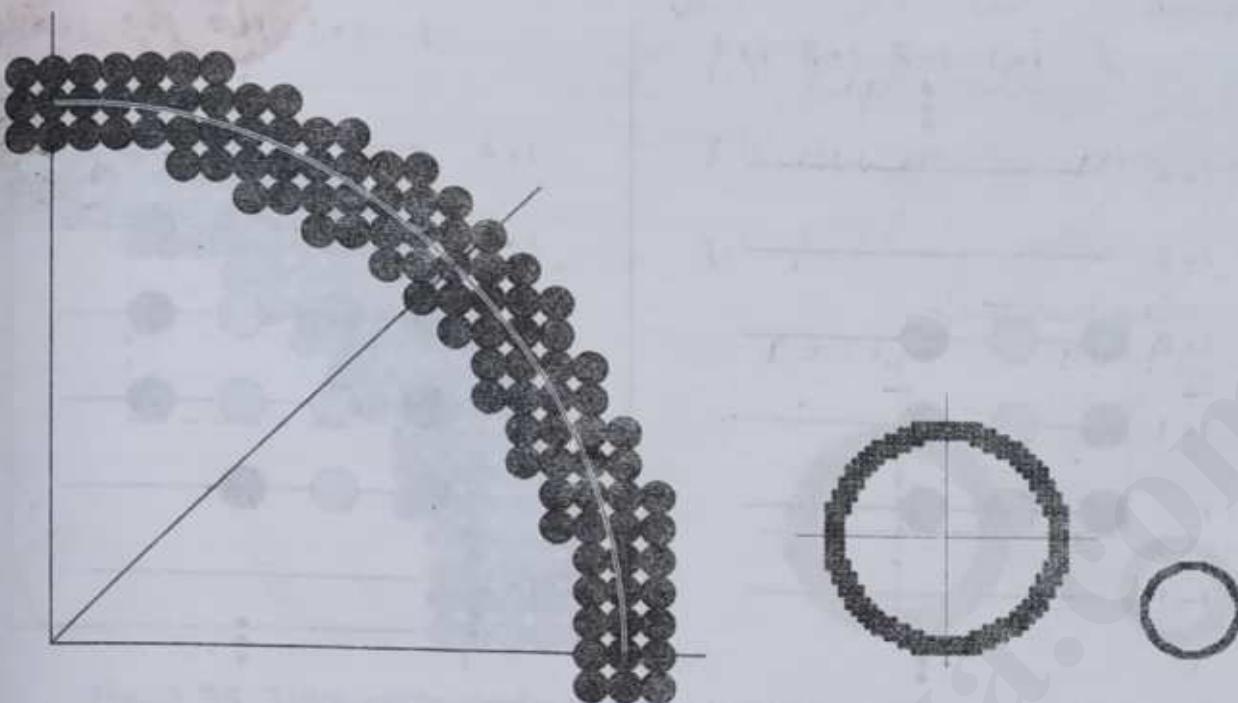
**Fig. 3.32** Thick circle drawn by column replication.

way: The width is thinnest for horizontal segments and thickest for segments with slope of  $\pm 1$ . An ellipse arc, for example, varies in thickness along its entire trajectory, being of the specified thickness when the tangent is nearly horizontal or vertical, and thickened by a factor of  $\sqrt{2}$  around  $\pm 45^\circ$  (see Fig. 3.34). This problem would be eliminated if the square turned to follow the path, but it is much better to use a circular cross-section so that the thickness is angle-independent.

Now let's look at how to implement the moving-pen algorithm for the simple case of an upright rectangular or circular cross-section. The easiest solution is to copyPixel the required solid or patterned cross-section (also called *footprint*) so that its center or corner is at the chosen pixel; for a circular footprint and a pattern drawn in opaque mode, we must in addition mask off the bits outside the circular region, which is not an easy task unless our low-level copyPixel has a write mask for the destination region. The brute-force copyPixel solution writes pixels more than once, since the pen's footprints overlap at adjacent pixels. A better technique that also handles the circular-cross-section problem is to use the spans of the footprint to compute spans for successive footprints at adjacent pixels. As in filling



**Fig. 3.33** Thick line drawn by tracing a rectangular pen.



**Fig. 3.34** Thick circle drawn by tracing a rectangular pen.

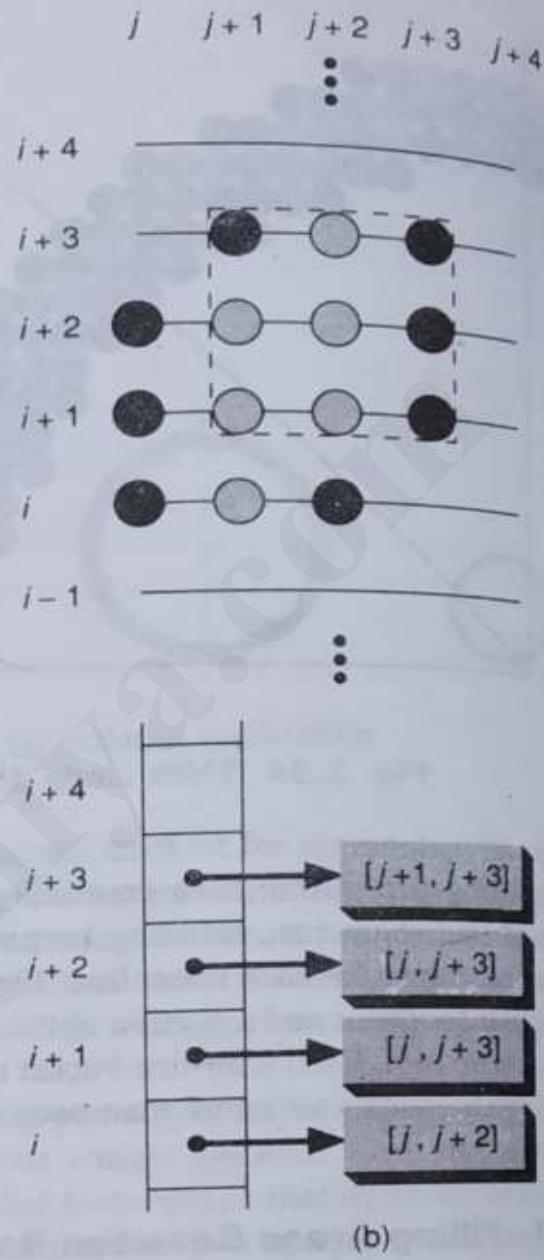
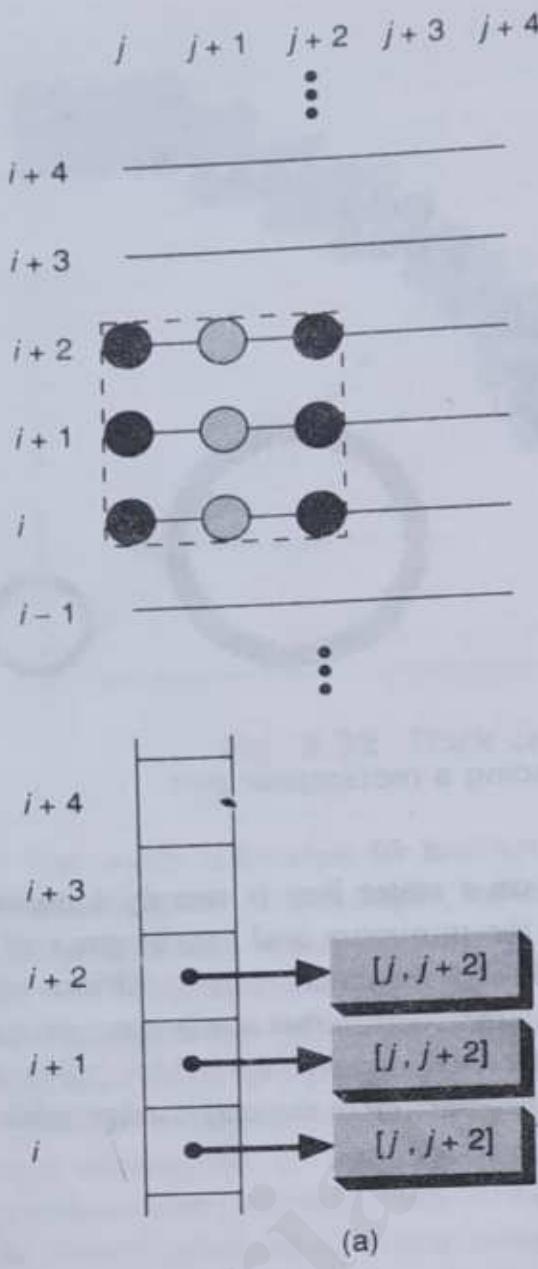
area-defining primitives, such combining of spans on a raster line is merely a union or merge of line segments, entailing keeping track of the minimum and maximum  $x$  of the accumulated span for each raster line. Figure 3.35 shows a sequence of two positions of the rectangular footprint and a portion of the temporary data structure that stores span extremes for each scan line. Each scan-line bucket may contain a list of spans when a thick polygon or ellipse arc is intersected more than once on a scan line, much like the active-edge table for polygons.

### 3.9.3 Filling Areas Between Boundaries

The third method for displaying a thick primitive is to construct the primitive's inner and outer boundary at a distance  $t/2$  on either side of the ideal (single-pixel) primitive trajectory. Alternatively, for area-defining primitives, we can leave the original boundary as the outer boundary, then draw the inner boundary inward. This filling technique has the advantage of handling both odd and even thicknesses; and of not increasing the extent of a primitive when the primitive is thickened. The disadvantage of this technique, however, is that an area-defining primitive effectively "shrinks" a bit, and that its "center line," the original 1-pixel outline, appears to shift.

A thick line is drawn as a rectangle with thickness  $t$  and length of the original line. Thus, the rectangle's thickness is independent of the line's angle, and the rectangle's edges are perpendicular to the line. In general, the rectangle is rotated and its vertices do not lie on the integer grid; thus, they must be rounded to the nearest pixel, and the resulting rectangle must then be scan-converted as a polygon.

To create thick circles, we scan convert two circles, the outer one of radius  $R + t/2$ , the inner one of radius  $R - t/2$ , and fill in the single or double spans between them, as shown in Fig. 3.36.



**Fig. 3.35** Recording spans of the rectangular pen: (a) footprint at  $x = j + 1$ ; (b)  $x = j + 2$ .

For ellipses, the situation is not nearly so simple. It is a classic result in differential geometry that the curves formed by moving a distance  $t/2$  perpendicular to an ellipse are not confocal ellipses, but are described by eighth-order equations [SALM96].<sup>9</sup> These functions are computationally expensive to scan convert; therefore, as usual, we approximate. We scan convert two confocal ellipses, the inner with semidiameters  $a - t/2$  and  $b - t/2$ , the outer with semidiameters  $a + t/2$  and  $b + t/2$ . Again, we calculate spans and fill them in, either after all span arithmetic is done, or on the fly. The standard problems of thin ellipses (treated in Chapter 19) pertain. Also, the problem of generating the inner boundary, noted here for ellipses, also can occur for other primitives supported in raster graphics packages.

<sup>9</sup>The eighth-order curves so generated may have self-intersections or cusps, as may be seen by constructing the normal lines by hand.

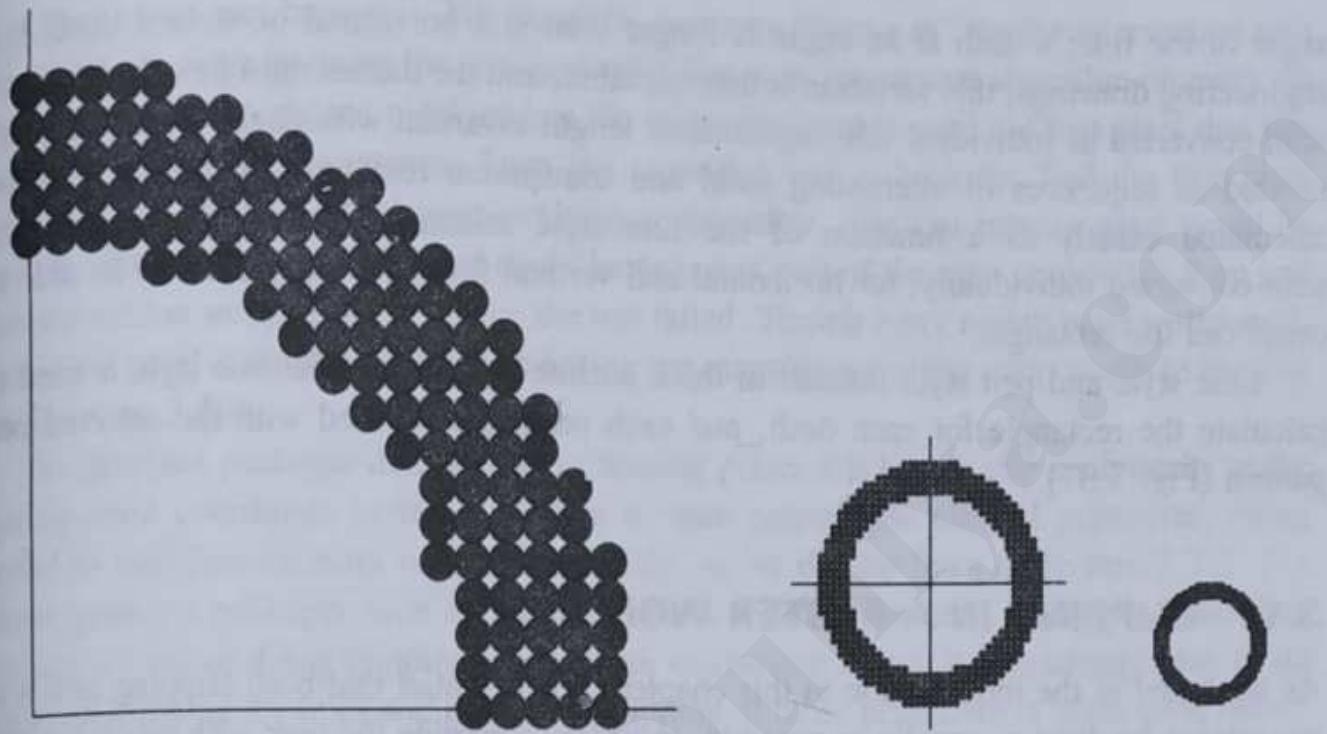


Fig. 3.36 Thick circle drawn by filling between concentric circles.

#### 3.9.4 Approximation by Thick Polylines

We can do piecewise-linear approximation of any primitive by computing points on the boundary (with floating-point coordinates), then connecting these points with line segments to form a polyline. The advantage of this approach is that the algorithms for both line clipping and line scan conversion (for thin primitives), and for polygon clipping and polygon scan conversion (for thick primitives), are efficient. Naturally, the segments must be quite short in places where the primitive changes direction rapidly. Ellipse arcs can be represented as ratios of parametric polynomials, which lend themselves readily to such piecewise-linear approximation (see Chapter 11). The individual line segments are then drawn as rectangles with the specified thickness. To make the thick approximation look nice, however, we must solve the problem of making thick lines join smoothly, as discussed in Chapter 19.

### 3.12 CLIPPING LINES

This section treats analytical clipping of lines against rectangles;<sup>10</sup> algorithms for clipping other primitives are handled in subsequent sections. Although there are specialized algorithms for rectangle and polygon clipping, it is important to note that SRGP primitives built out of lines (i.e., polylines, unfilled rectangles, and polygons) can be clipped by repeated application of the line clipper. Furthermore, circles and ellipses may be piecewise-linearly approximated with a sequence of very short lines, so that boundaries can be treated as a single polyline or polygon for both clipping and scan conversion. Conics are represented in some systems as ratios of parametric polynomials (see Chapter 11), a representation that also lends itself readily to an incremental, piecewise linear approximation suitable for a line-clipping algorithm. Clipping a rectangle against a rectangle results in at most a single rectangle. Clipping a convex polygon against a rectangle results in at most a single convex polygon, but clipping a concave polygon may produce more than one concave polygon. Clipping a circle or ellipse against a rectangle results in as many as four arcs.

Lines intersecting a rectangular clip region (or any convex polygon) are always clipped to a single line segment; lines lying on the clip rectangle's border are considered inside and hence are displayed. Figure 3.38 shows several examples of clipped lines.

---

<sup>10</sup>This chapter does not cover clipping primitives to multiple rectangles (as when windows overlap in a windowing system) or to nonrectangular regions; the latter topic is discussed briefly in Section 19.7.

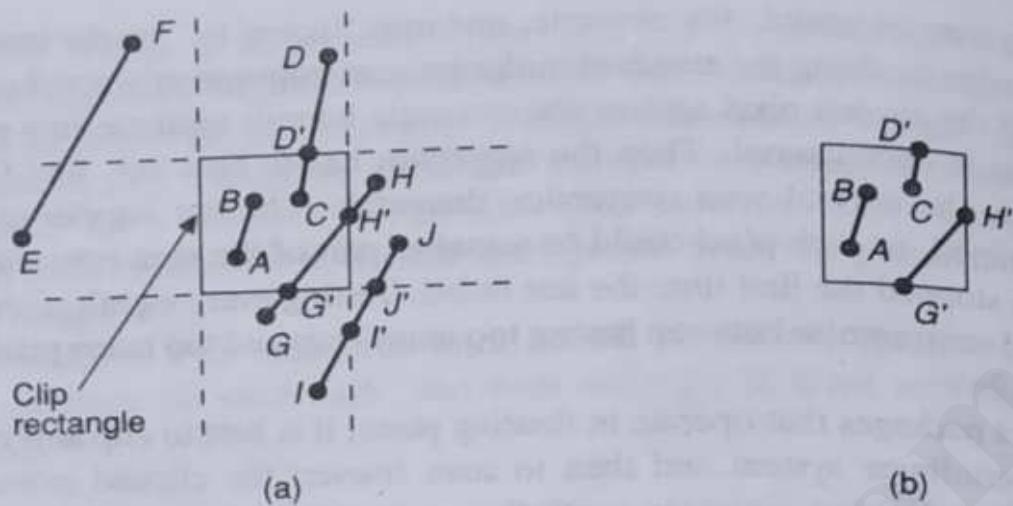


Fig. 3.38 Cases for clipping lines.

### 3.12.1 Clipping Endpoints

Before we discuss clipping lines, let's look at the simpler problem of clipping individual points. If the  $x$  coordinate boundaries of the clip rectangle are at  $x_{\min}$  and  $x_{\max}$ , and the  $y$  coordinate boundaries are at  $y_{\min}$  and  $y_{\max}$ , then four inequalities must be satisfied for a point at  $(x, y)$  to be inside the clip rectangle:

$$x_{\min} \leq x \leq x_{\max}, y_{\min} \leq y \leq y_{\max}.$$

If any of the four inequalities does not hold, the point is outside the clip rectangle.

### 3.12.2 Clipping Lines by Solving Simultaneous Equations

To clip a line, we need to consider only its endpoints, not its infinitely many interior points. If both endpoints of a line lie inside the clip rectangle (e.g.,  $AB$  in Fig. 3.38), the entire line lies inside the clip rectangle and can be *trivially accepted*. If one endpoint lies inside and one outside (e.g.,  $CD$  in the figure), the line 'intersects the clip rectangle and we must compute the intersection point. If both endpoints are outside the clip rectangle, the line may (or may not) intersect with the clip rectangle ( $EF$ ,  $GH$ , and  $IJ$  in the figure), and we need to perform further calculations to determine whether there are any intersections, and if there are, where they occur.

The brute-force approach to clipping a line that cannot be trivially accepted is to intersect that line with each of the four clip-rectangle edges to see whether any intersection points lie on those edges; if so, the line cuts the clip rectangle and is partially inside. For each line and clip-rectangle edge, we therefore take the two mathematically infinite lines that contain them and intersect them. Next, we test whether this intersection point is "interior"—that is, whether it lies within both the clip rectangle edge and the line; if so, there is an intersection with the clip rectangle. In Fig. 3.38, intersection points  $G'$  and  $H'$  are interior, but  $I'$  and  $J'$  are not.

When we use this approach, we must solve two simultaneous equations using multiplication and division for each  $\langle$ edge, line $\rangle$  pair. Although the slope-intercept

formula for lines learned in analytic geometry could be used, it describes infinite lines, whereas in graphics and clipping we deal with finite lines (called *line segments* in mathematics). In addition, the slope-intercept formula does not deal with vertical lines—a serious problem, given our upright clip rectangle. A parametric formulation for line segments solves both problems:

$$x = x_0 + t(x_1 - x_0), \quad y = y_0 + t(y_1 - y_0).$$

These equations describe  $(x, y)$  on the directed line segment from  $(x_0, y_0)$  to  $(x_1, y_1)$  for the parameter  $t$  in the range  $[0, 1]$ , as simple substitution for  $t$  confirms. Two sets of simultaneous equations of this parametric form can be solved for parameters  $t_{\text{edge}}$  for the edge and  $t_{\text{line}}$  for the line segment. The values of  $t_{\text{edge}}$  and  $t_{\text{line}}$  can then be checked to see whether both lie in  $[0, 1]$ ; if they do, the intersection point lies within both segments and is a true clip-rectangle intersection. Furthermore, the special case of a line parallel to a clip-rectangle edge must also be tested before the simultaneous equations can be solved. Altogether, the brute-force approach involves considerable calculation and testing; it is thus inefficient.

### 3.12.3 The Cohen–Sutherland Line-Clipping Algorithm

The more efficient Cohen–Sutherland algorithm performs initial tests on a line to determine whether intersection calculations can be avoided. First, endpoint pairs are checked for trivial acceptance. If the line cannot be trivially accepted, region checks are done. For instance, two simple comparisons on  $x$  show that both endpoints of line  $EF$  in Fig. 3.38 have an  $x$  coordinate less than  $x_{\min}$  and thus lie in the region to the left of the clip rectangle (i.e., in the outside halfplane defined by the left edge); therefore, line segment  $EF$  can be trivially rejected and needs to be neither clipped nor displayed. Similarly, we can trivially reject lines with both endpoints in regions to the right of  $x_{\max}$ , below  $y_{\min}$ , and above  $y_{\max}$ .

If the line segment can be neither trivially accepted nor rejected, it is divided into two segments at a clip edge, so that one segment can be trivially rejected. Thus, a segment is iteratively clipped by testing for trivial acceptance or rejection, and is then subdivided if neither test is successful, until what remains is completely inside the clip rectangle or can be trivially rejected. The algorithm is particularly efficient for two common cases. In the first case of a large clip rectangle enclosing all or most of the display area, most primitives can be trivially accepted. In the second case of a small clip rectangle, almost all primitives can be trivially rejected. This latter case arises in a standard method of doing pick correlation in which a small rectangle surrounding the cursor, called the *pick window*, is used to clip primitives to determine which primitives lie within a small (rectangular) neighborhood of the cursor's *pick point* (see Section 7.12.2).

To perform trivial accept and reject tests, we extend the edges of the clip rectangle to divide the plane of the clip rectangle into nine regions (see Fig. 3.39). Each region is assigned a 4-bit code, determined by where the region lies with respect to the outside halfplanes of the clip-rectangle edges. Each bit in the outcode is set to either 1 (true) or 0 (false); the 4 bits in the code correspond to the following conditions:

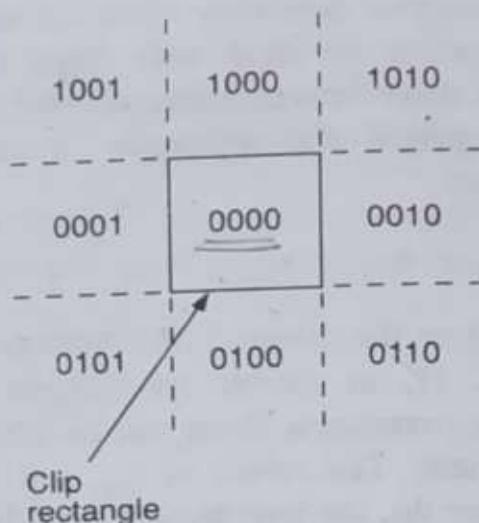


Fig. 3.39 Region outcodes.

First bit	outside halfplane of top edge, above top edge	$y > y_{\max}$
Second bit	outside halfplane of bottom edge, below bottom edge	$y < y_{\min}$
Third bit	outside halfplane of right edge, to the right of right edge	$x > x_{\max}$
Fourth bit	outside halfplane of left edge, to the left of left edge	$x < x_{\min}$

Since the region lying above and to the left of the clip rectangle, for example, lies in the outside halfplane of the top and left edges, it is assigned a code of 1001. A particularly efficient way to calculate the outcode derives from the observation that bit 1 is the sign bit of  $(y_{\max} - y)$ ; bit 2 is that of  $(y - y_{\min})$ ; bit 3 is that of  $(x_{\max} - x)$ ; and bit 4 is that of  $(x - x_{\min})$ . Each endpoint of the line segment is then assigned the code of the region in which it lies. We can now use these endpoint codes to determine whether the line segment lies completely inside the clip rectangle or in the outside halfplane of an edge. If both 4-bit codes of the endpoints are zero, then the line lies completely inside the clip rectangle. However, if both endpoints lie in the outside halfplane of a particular edge, as for *EF* in Fig. 3.38, the codes for both endpoints each have the bit set showing that the point lies in the outside halfplane of that edge. For *EF*, the outcodes are 0001 and 1001, respectively, showing with the fourth bit that the line segment lies in the outside halfplane of the left edge. Therefore, if the logical **and** of the codes of the endpoints is not zero, the line can be trivially rejected.

If a line cannot be trivially accepted or rejected, we must subdivide it into two segments such that one or both segments can be discarded. We accomplish this subdivision by using an edge that the line crosses to cut the line into two segments: The section lying in the outside halfplane of the edge is thrown away. We can choose any order in which to test edges, but we must, of course, use the same order each time in the algorithm; we shall use the top-to-bottom, right-to-left order of the outcode. A key property of the outcode is that bits that are set in a nonzero outcode correspond to edges crossed: If one endpoint lies in the outside halfplane of an edge and the line segment fails the trivial-rejection tests, then the other point must lie on the inside halfplane of that edge and the line segment must cross it. Thus, the algorithm always chooses a point that lies outside and then uses an outcode bit that is set to determine a clip edge; the edge chosen is the first in the top-to-bottom, right-to-left order—that is, it is the leftmost bit that is set in the outcode.

The algorithm works as follows. We compute the outcodes of both endpoints and check

for trivial acceptance and rejection. If neither test is successful, we find an endpoint that lies outside (at least one will), and then test the outcode to find the edge that is crossed and to determine the corresponding intersection point. We can then clip off the line segment from the outside endpoint to the intersection point by replacing the outside endpoint with the intersection point, and compute the outcode of this new endpoint to prepare for the next iteration.

For example, consider the line segment  $AD$  in Fig. 3.40. Point  $A$  has outcode 0000 and point  $D$  has outcode 1001. The line can be neither trivially accepted or rejected. Therefore, the algorithm chooses  $D$  as the outside point, whose outcode shows that the line crosses the top edge and the left edge. By our testing order, we first use the top edge to clip  $AD$  to  $AB$ , and we compute  $B$ 's outcode as 0000. In the next iteration, we apply the trivial acceptance/rejection tests to  $AB$ , and it is trivially accepted and displayed.

Line  $EI$  requires multiple iterations. The first endpoint,  $E$ , has an outcode of 0100, so the algorithm chooses it as the outside point and tests the outcode to find that the first edge against which the line is cut is the bottom edge, where  $EI$  is clipped to  $FI$ . In the second iteration,  $FI$  cannot be trivially accepted or rejected. The outcode of the first endpoint,  $F$ , is 0000, so the algorithm chooses the outside point  $I$  that has outcode 1010. The first edge clipped against is therefore the top edge, yielding  $FH$ .  $H$ 's outcode is determined to be 0010, so the third iteration results in a clip against the right edge to  $FG$ . This is trivially accepted in the fourth and final iteration and displayed. A different sequence of clips would have resulted if we had picked  $I$  as the initial point: On the basis of its outcode, we would have clipped against the top edge first, then the right edge, and finally the bottom edge.

In the code of Fig. 3.41, we use constant integers and bitwise arithmetic to represent the outcodes, because this representation is more natural than an array with an entry for each outcode. We use an internal procedure to calculate the outcode for modularity; to improve performance, we would, of course, put this code in line.

We can improve the efficiency of the algorithm slightly by not recalculating slopes (see Exercise 3.28). Even with this improvement, however, the algorithm is not the most efficient one. Because testing and clipping are done in a fixed order, the algorithm will sometimes perform needless clipping. Such clipping occurs when the intersection with a rectangle edge is an “external intersection”; that is, when it does not lie on the clip-rectangle boundary (e.g., point  $H$  on line  $EI$  in Fig. 3.40). The Nicholl, Lee, and

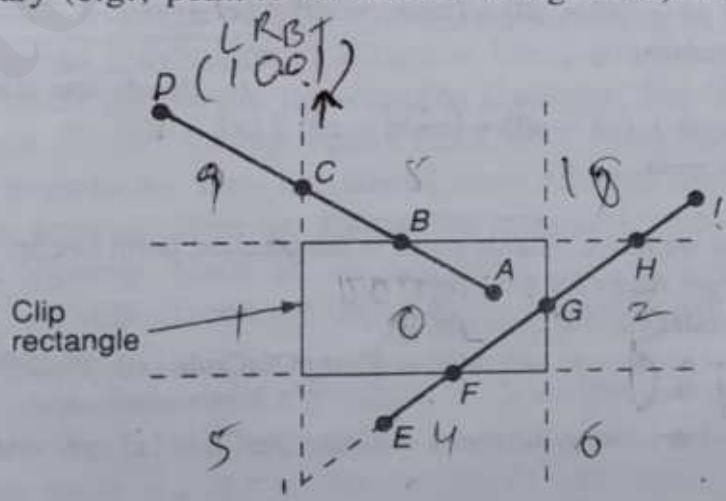


Fig. 3.40 Illustration of Cohen-Sutherland line clipping.

```

typedef unsigned int outcode;
enum {TOP = 0x1, BOTTOM = 0x2, RIGHT = 0x4, LEFT = 0x8};

void CohenSutherlandLineClipAndDraw (
    double x0, double y0, double x1, double y1, double xmin, double xmax,
    double ymin, double ymax, int value)
/* Cohen-Sutherland clipping algorithm for line P0 = (x0, y0) to P1 = (x1, y1) and */
/* clip rectangle with diagonal from (xmin, ymin) to (xmax, ymax) */
{
    /* Outcodes for P0, P1, and whatever point lies outside the clip rectangle */
    outcode outcode0, outcode1, outcodeOut;
    boolean accept = FALSE, done = FALSE;
    outcode0 = CompOutCode (x0, y0, xmin, xmax, ymin, ymax);
    outcode1 = CompOutCode (x1, y1, xmin, xmax, ymin, ymax);
    do {
        if (!(outcode0 | outcode1)) { /* Trivial accept and exit */
            accept = TRUE; done = TRUE;
        } else if (outcode0 & outcode1) /* Logical and is true, so trivial reject and exit */
            done = TRUE;
        else {
            /* Failed both tests, so calculate the line segment to clip: */
            /* from an outside point to an intersection with clip edge. */
            double x, y;
            /* At least one endpoint is outside the clip rectangle; pick it. */
            outcodeOut = outcode0 ? outcode0 : outcode1;
            /* Now find intersection point; */
            /* use formulas y = y0 + slope * (x - x0), x = x0 + (1/slope) * (y - y0). */
            if (outcodeOut & TOP) { /* Divide line at top of clip rect */
                x = x0 + (x1 - x0) * (ymax - y0) / (y1 - y0);
                y = ymax;
            } else if (outcodeOut & BOTTOM) { /* Divide line at bottom edge of clip rect */
                x = x0 + (x1 - x0) * (ymin - y0) / (y1 - y0);
                y = ymin;
            } else if (outcodeOut & RIGHT) { /* Divide line at right edge of clip rect */
                y = y0 + (y1 - y0) * (xmax - x0) / (x1 - x0);
                x = xmax;
            } else { /* Divide line at left edge of clip rect */
                y = y0 + (y1 - y0) * (xmin - x0) / (x1 - x0);
                x = xmin;
            }
            /* Now we move outside point to intersection point to clip, */
            /* and get ready for next pass. */
            if (outcodeOut == outcode0) {
                x0 = x; y0 = y; outcode0 = CompOutCode (x0, y0, xmin, xmax, ymin, ymax);
            } else {
                x1 = x; y1 = y; outcode1 = CompOutCode (x1, y1, xmin, xmax, ymin, ymax);
            }
        } /* Subdivide */
    } while (done == FALSE);
}

```

Fig. 3.41 (Cont.)

```

if (accept)
    MidpointLineReal (x0, y0, x1, y1, value); /* Version for double coordinates */
} /* CohenSutherlandLineClipAndDraw */

outcode CompOutCode (
    double x, double y, double xmin, double xmax, double ymin, double ymax);
{
    outcode code = 0;
    if (y > ymax)
        code |= TOP;
    else if (y < ymin)
        code |= BOTTOM;
    if (x > xmax)
        code |= RIGHT;
    else if (x < xmin)
        code |= LEFT;
    return code;
} /* CompOutCode */

```

**Fig. 3.41** Cohen–Sutherland line-clipping algorithm.

Nicholl [NICH87] algorithm, by contrast, avoids calculating external intersections by subdividing the plane into many more regions; it is discussed in Chapter 19. An advantage of the much simpler Cohen–Sutherland algorithm is that its extension to a 3D orthographic view volume is straightforward, as seen in Section 6.5.3

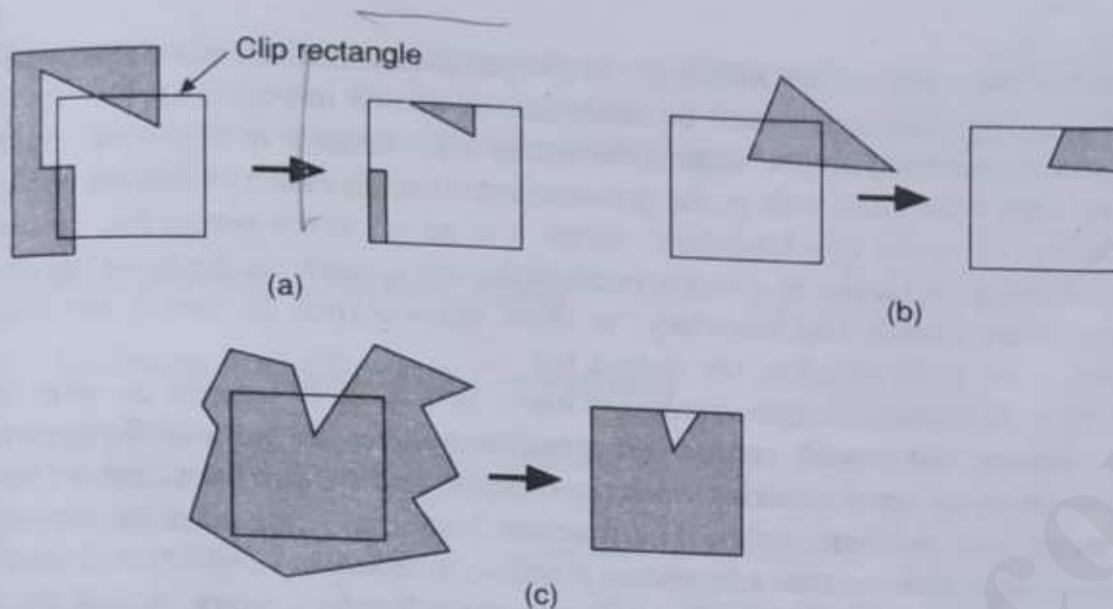
## 3.14 CLIPPING POLYGONS

An algorithm that clips a polygon must deal with many different cases, as shown in Fig. 3.46. The case in part (a) is particularly noteworthy in that the concave polygon is clipped into two separate polygons. All in all, the task of clipping seems rather complex. Each edge of the polygon must be tested against each edge of the clip rectangle; new edges must be added, and existing edges must be discarded, retained, or divided. Multiple polygons may result from clipping a single polygon. We need an organized way to deal with all these cases.

### 3.14.1 The Sutherland–Hodgman Polygon-Clipping Algorithm

Sutherland and Hodgman's polygon-clipping algorithm [SUTH74b] uses a divide-and-conquer strategy: It solves a series of simple and identical problems that, when combined, solve the overall problem. The simple problem is to clip a polygon against a single infinite clip edge. Four clip edges, each defining one boundary of the clip rectangle (see Fig. 3.47), successively clip a polygon against a clip rectangle.

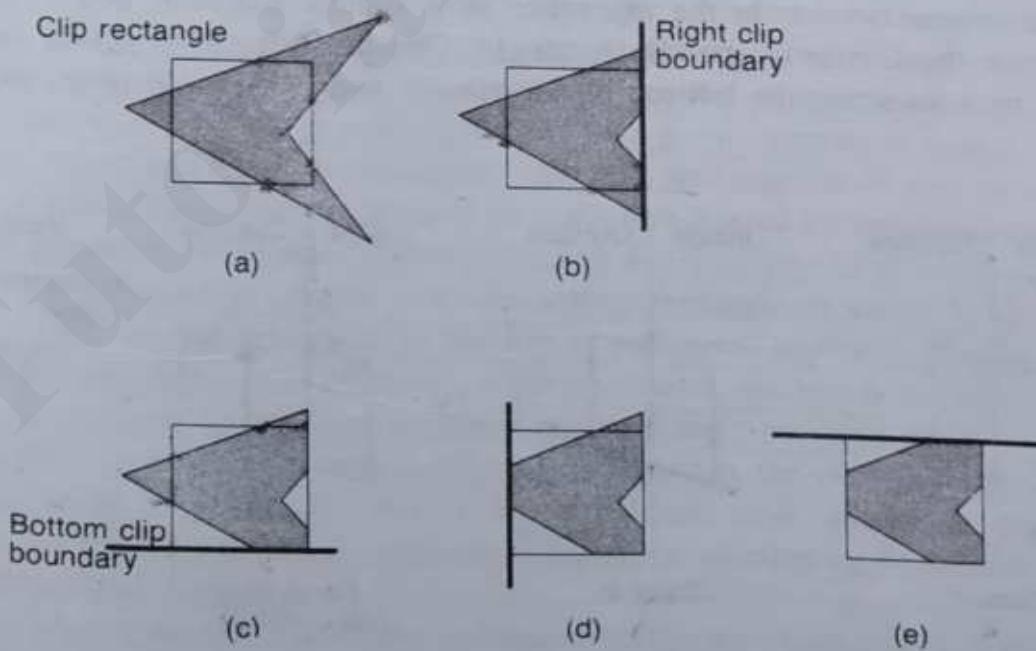
Note the difference between this strategy for a polygon and the Cohen–Sutherland algorithm for clipping a line: The polygon clipper clips against four edges in succession, whereas the line clipper tests the outcode to see which edge is crossed, and clips only when



**Fig. 3.46** Examples of polygon clipping. (a) Multiple components. (b) Simple convex case. (c) Concave case with many exterior edges.

necessary. The actual Sutherland–Hodgman algorithm is in fact more general: A polygon (convex or concave) can be clipped against any convex clipping polygon; in 3D, polygons can be clipped against convex polyhedral volumes defined by planes. The algorithm accepts a series of polygon vertices  $v_1, v_2, \dots, v_n$ . In 2D, the vertices define polygon edges from  $v_i$  to  $v_{i+1}$  and from  $v_n$  to  $v_1$ . The algorithm clips against a single, infinite clip edge and outputs another series of vertices defining the clipped polygon. In a second pass, the partially clipped polygon is then clipped against the second clip edge, and so on.

The algorithm moves around the polygon from  $v_n$  to  $v_1$  and then on back to  $v_n$ , at each step examining the relationship between successive vertices and the clip edge. At each step,



**Fig. 3.47** Polygon clipping, edge by edge. (a) Before clipping. (b) Clip on right. (c) Clip on bottom. (d) Clip on left. (e) Clip on top; polygon is fully clipped.

zero, one, or two vertices are added to the output list of vertices that defines the clipped polygon. Four possible cases must be analyzed, as shown in Fig. 3.48.

Let's consider the polygon edge from vertex  $s$  to vertex  $p$  in Fig. 3.48. Assume that start point  $s$  has been dealt with in the previous iteration. In case 1, when the polygon edge is completely inside the clip boundary, vertex  $p$  is added to the output list. In case 2, the intersection point  $i$  is output as a vertex because the edge intersects the boundary. In case 3, both vertices are outside the boundary, so there is no output. In case 4, the intersection point  $i$  and  $p$  are both added to the output list.

Function SutherlandHodgmanPolygonClip() in Fig. 3.49 accepts an array *inVertexArray* of vertices and creates another array *outVertexArray* of vertices. To keep the code simple, we show no error checking on array bounds, and we use the function *Output()* to place a vertex into *outVertexArray*. The function *Intersect()* calculates the intersection of the polygon edge from vertex  $s$  to vertex  $p$  with *clip Boundary*, which is defined by two vertices on the clip polygon's boundary. The function *Inside()* returns **true** if the vertex is on the inside of the clip boundary, where "inside" is defined as "to the left of the clip boundary when one looks from the first vertex to the second vertex of the clip boundary." This sense corresponds to a counterclockwise enumeration of edges. To calculate whether a point lies outside a clip boundary, we can test the sign of the dot product of the normal to the clip boundary and the polygon edge, as described in Section 3.12.4. (For the simple case of an upright clip rectangle, we need only test the sign of the horizontal or vertical distance to its boundary.)

Sutherland and Hodgman show how to structure the algorithm so that it is reentrant [SUTH74b]. As soon as a vertex is output, the clipper calls itself with that vertex. Clipping is performed against the next clip boundary, so that no intermediate storage is necessary for the partially clipped polygon: In essence, the polygon is passed through a "pipeline" of clippers. Each step can be implemented as special-purpose hardware with no intervening buffer space. This property (and its generality) makes the algorithm suitable for today's hardware implementations. In the algorithm as it stands, however, new edges may be introduced on the border of the clip rectangle. Consider Fig. 3.46 (a)—a new edge is introduced by connecting the left top of the triangle and the left top of the rectangle. A

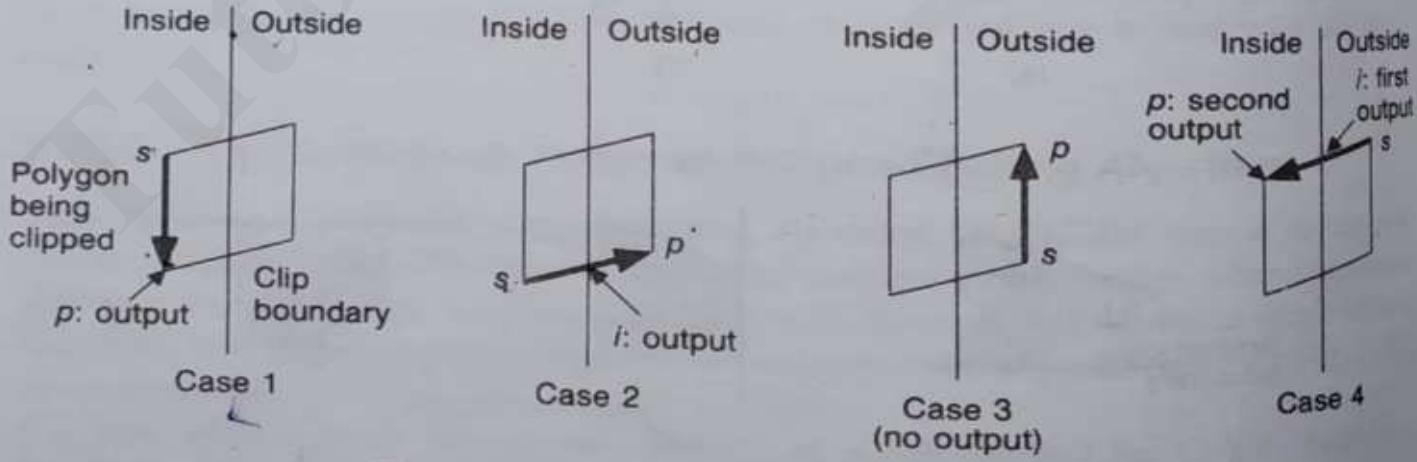


Fig. 3.48 Four cases of polygon clipping.

postprocessing phase can eliminate these edges, as discussed in Chapter 19. A polygon-clipping algorithm based on the parametric-line representation for clipping to upright rectangular clip regions is discussed, along with the Weiler algorithm for clipping polygons to polygons, in Section 19.1.

## 3.17 ANTIALIASING

### 3.17.1 Increasing Resolution

The primitives drawn so far have a common problem: They have jagged edges. This undesirable effect, known as *the jaggies* or *staircasing*, is the result of an all-or-nothing approach to scan conversion in which each pixel either is replaced with the primitive's color or is left unchanged. Jaggies are an instance of a phenomenon known as *aliasing*. The application of techniques that reduce or eliminate aliasing is referred to as *antialiasing*, and primitives or images produced using these techniques are said to be *antialiased*. In Chapter 14, we discuss basic ideas from signal processing that explain how aliasing got its name, why it occurs, and how to reduce or eliminate it when creating pictures. Here, we content ourselves with a more intuitive explanation of why SRGP's primitives exhibit aliasing, and describe how to modify the line scan-conversion algorithm developed in this chapter to generate antialiased lines.

Consider using the midpoint algorithm to draw a 1-pixel-thick black line, with slope between 0 and 1, on a white background. In each column through which the line passes, the algorithm sets the color of the pixel that is closest to the line. Each time the line moves between columns in which the pixels closest to the line are not in the same row, there is a sharp jag in the line drawn into the canvas, as is clear in Fig. 3.54(a). The same is true for other scan-converted primitives that can assign only one of two intensity values to pixels.

Suppose we now use a display device with twice the horizontal and vertical resolution. As shown in Fig. 3.54 (b), the line passes through twice as many columns and therefore has twice as many jags, but each jag is half as large in  $x$  and in  $y$ . Although the resulting picture looks better, the improvement comes at the price of quadrupling the memory cost, memory bandwidth, and scan-conversion time. Increasing resolution is an expensive solution that only diminishes the problem of jaggies—it does not eliminate the problem. In the following sections, we look at antialiasing techniques that are less costly, yet result in significantly better images.

### 3.17.2 Unweighted Area Sampling

The first approach to improving picture quality can be developed by recognizing that, although an ideal primitive such as the line has zero width, the primitive we are drawing has

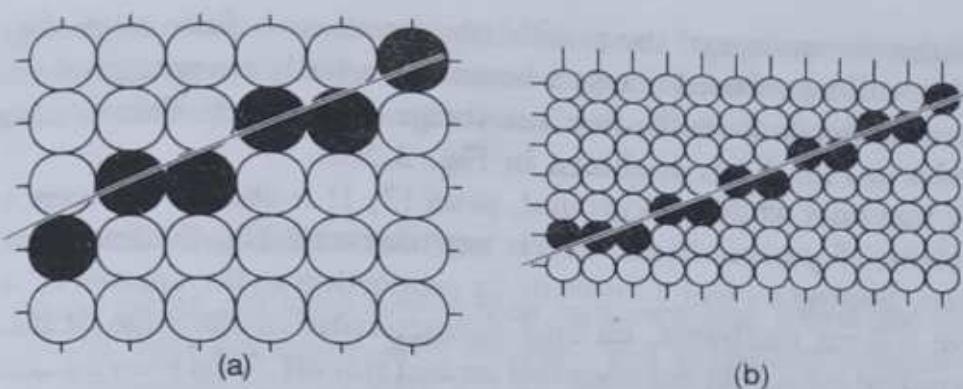


Fig. 3.54 (a) Standard midpoint line on a bilevel display. (b) Same line on a display that has twice the linear resolution.

nonzero width. A scan-converted primitive occupies a finite area on the screen—even the thinnest horizontal or vertical line on a display surface is 1 pixel thick and lines at other angles have width that varies over the primitive. Thus, we think of any line as a rectangle of a desired thickness covering a portion of the grid, as shown in Fig. 3.55. It follows that a line should not set the intensity of only a single pixel in a column to black, but rather should contribute some amount of intensity to each pixel in the columns whose area it intersects. (Such varying intensity can be shown on only those displays with multiple bits per pixel, of course.) Then, for 1-pixel-thick lines, only horizontal and vertical lines would affect exactly 1 pixel in their column or row. For lines at other angles, more than 1 pixel would now be set in a column or row, each to an appropriate intensity.

But what is the geometry of a pixel? How large is it? How much intensity should a line contribute to each pixel it intersects? It is computationally simple to assume that the pixels form an array of nonoverlapping square tiles covering the screen, centered on grid points. (When we refer to a primitive overlapping all or a portion of a pixel, we mean that it covers (part of) the tile; to emphasize this we sometimes refer to the square as the *area represented by the pixel*.) We also assume that a line contributes to each pixel's intensity an amount

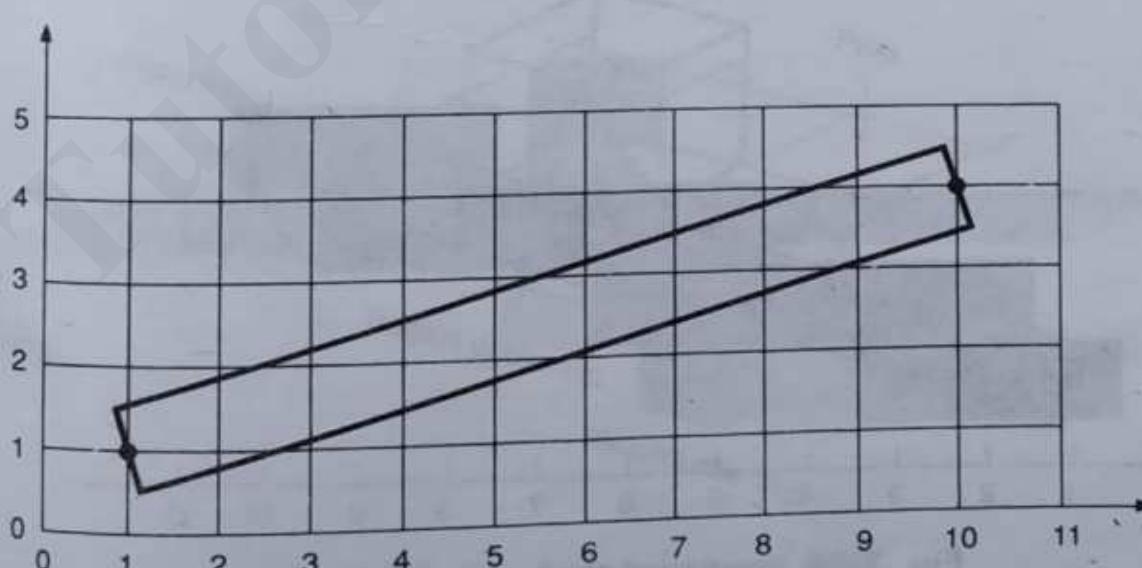


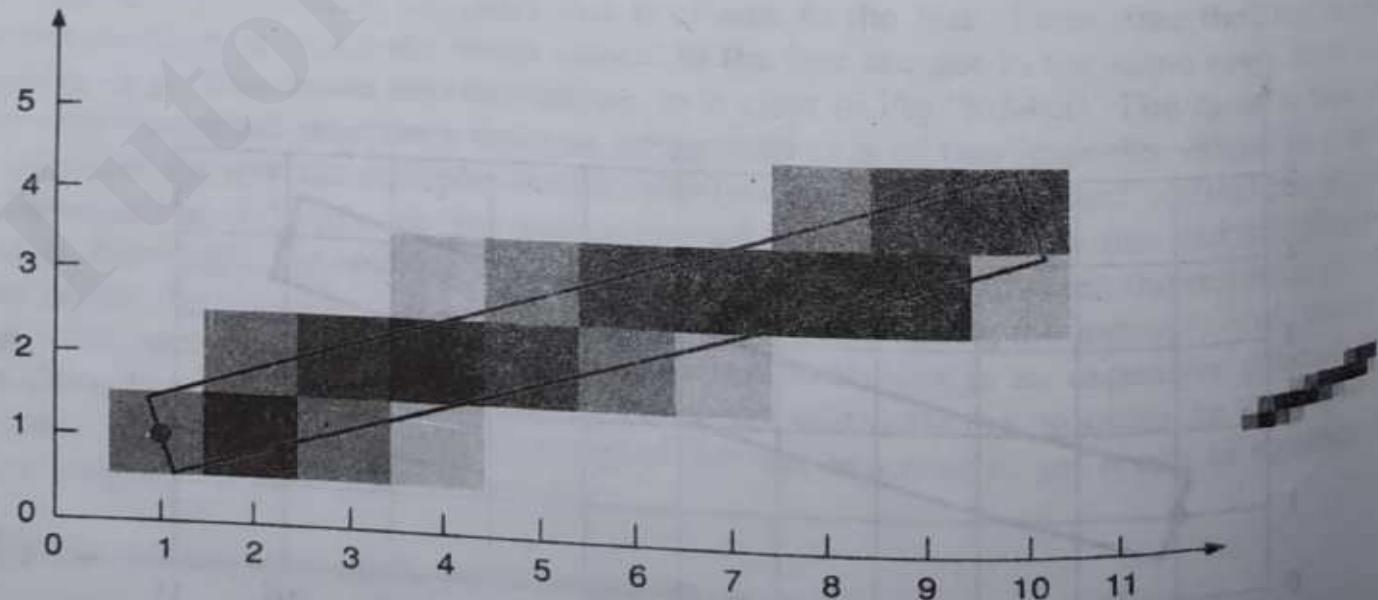
Figure 3.55 Line of nonzero width from point (1,1) to point (10,4).

proportional to the percentage of the pixel's tile it covers. A fully covered pixel on a black and white display will be colored black, whereas a partially covered pixel will be colored a gray whose intensity depends on the line's coverage of the pixel. This technique, as applied to the line shown in Fig. 3.55, is shown in Fig. 3.56.

For a black line on a white background, pixel (2, 1) is about 70 percent black, whereas pixel (2, 2) is about 25 percent black. Pixels not intersected by the line, such as (2, 3), are completely white. Setting a pixel's intensity in proportion to the amount of its area covered by the primitive softens the harsh, on-off characteristic of the edge of the primitive and yields a more gradual transition between full on and full off. This blurring makes a line look better at a distance, despite the fact that it spreads the on-off transition over multiple pixels in a column or row. A rough approximation to the area overlap can be found by dividing the pixel into a finer grid of rectangular subpixels, then counting the number of subpixels inside the line—for example, below the line's top edge or above its bottom edge (see Exercise 3.32).

We call the technique of setting intensity proportional to the amount of area covered *unweighted area sampling*. This technique produces noticeably better results than does setting pixels to full intensity or zero intensity, but there is an even more effective strategy called *weighted area sampling*. To explain the difference between the two forms of area sampling, we note that unweighted area sampling has the following three properties. First, the intensity of a pixel intersected by a line edge decreases as the distance between the pixel center and the edge increases: The farther away a primitive is, the less influence it has on a pixel's intensity. This relation obviously holds because the intensity decreases as the area of overlap decreases, and that area decreases as the line's edge moves away from the pixel's center and toward the boundary of the pixel. When the line covers the pixel completely, the overlap area and therefore the intensity are at a maximum; when the primitive edge is just tangent to the boundary, the area and therefore the intensity are zero.

A second property of unweighted area sampling is that a primitive cannot influence the intensity at a pixel at all if the primitive does not intersect the pixel—that is, if it does not intersect the square tile represented by the pixel. A third property of unweighted area



**Fig. 3.56** Intensity proportional to area covered.

3.17

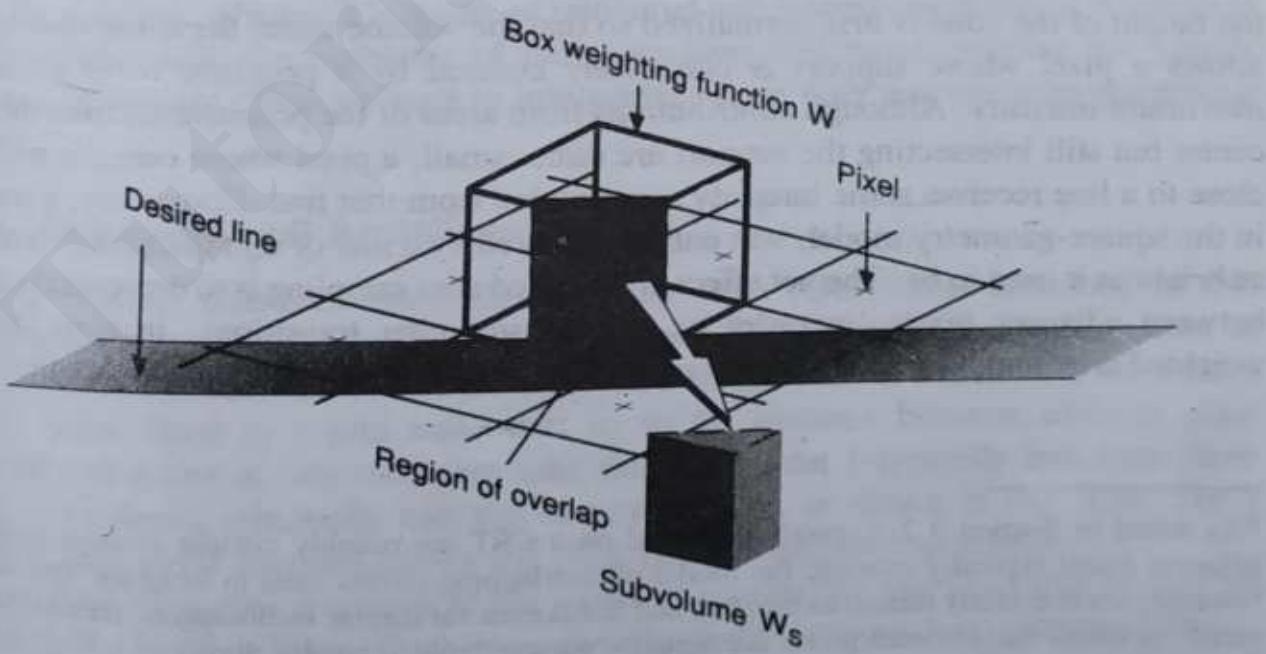
sampling is that equal areas contribute equal intensity, regardless of the distance between the pixel's center and the area; only the total amount of overlapped area matters. Thus, a small area in the corner of the pixel contributes just as much as does an equal-sized area near the pixel's center.

### 3.17.3 Weighted Area Sampling

In weighted area sampling, we keep unweighted area sampling's first and second properties (intensity decreases with decreased area overlap, and primitives contribute only if they overlap the area represented by the pixel), but we alter the third property. We let equal areas contribute unequally: A small area closer to the pixel center has greater influence than does one at a greater distance. A theoretical basis for this change is given in Chapter 14, where we discuss weighted area sampling in the context of filtering theory.

To retain the second property, we must make the following change in the geometry of the pixel. In unweighted area sampling, if an edge of a primitive is quite close to the boundary of the square tile we have used to represent a pixel until now, but does not actually intersect this boundary, it will not contribute to the pixel's intensity. In our new approach, the pixel represents a circular area larger than the square tile; the primitive *will* intersect this larger area; hence, it will contribute to the intensity of the pixel.

To explain the origin of the adjectives *unweighted* and *weighted*, we define a *weighting function* that determines the influence on the intensity of a pixel of a given small area  $dA$  of a primitive, as a function of  $dA$ 's distance from the center of the pixel. This function is constant for unweighted area sampling, and decreases with increasing distance for weighted area sampling. Think of the weighting function as a function,  $W(x, y)$ , on the plane, whose height above the  $(x, y)$  plane gives the weight for the area  $dA$  at  $(x, y)$ . For unweighted area sampling with the pixels represented as square tiles, the graph of  $W$  is a box, as shown in Fig. 3.57.



**Fig. 3.57** Box filter for square pixel.

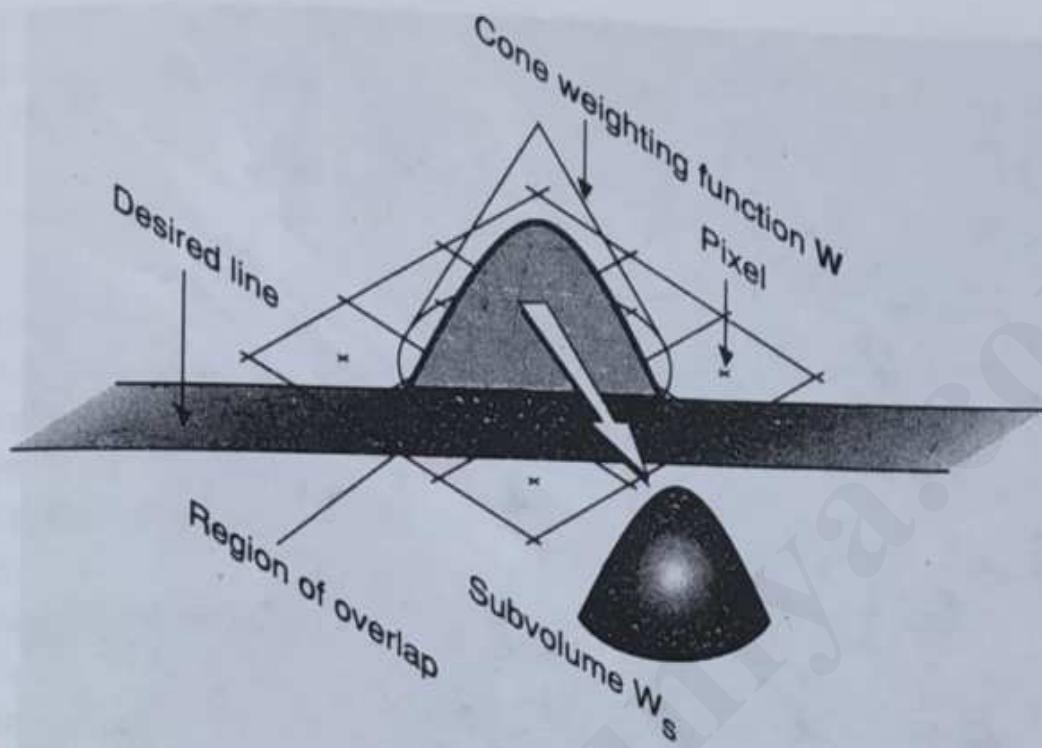
The figure shows square pixels, with centers indicated by crosses at the intersections of grid lines; the weighting function is shown as a box whose base is that of the current pixel. The intensity contributed by the area of the pixel covered by the primitive is the total of intensity contributions from all small areas in the region of overlap between the primitive and the pixel. The intensity contributed by each small area is proportional to the area multiplied by the weight. Therefore, the total intensity is the integral of the weighting function over the area of overlap. The volume represented by this integral,  $W_s$ , is always a fraction between 0 and 1, and the pixel's intensity  $I$  is  $I_{\max} \cdot W_s$ . In Fig. 3.57,  $W_s$  is a wedge of the box. The weighting function is also called a *filter function*, and the box is also called a *box filter*. For unweighted area sampling, the height of the box is normalized to 1, so that the box's volume is 1, which causes a thick line covering the entire pixel to have an intensity  $I = I_{\max} \cdot 1 = I_{\max}$ .

Now let us construct a weighting function for weighted area sampling; it must give less weight to small areas farther away from the pixel center than it does to those closer. Let's pick a weighting function that is the simplest decreasing function of distance; for example, we choose a function that has a maximum at the center of the pixel and decreases linearly with increasing distance from the center. Because of rotational symmetry, the graph of this function forms a circular cone. The circular base of the cone (often called the *support* of the filter) should have a radius larger than you might expect; the filtering theory of Chapter 14 shows that a good choice for the radius is the unit distance of the integer grid. Thus, a primitive fairly far from a pixel's center can still influence that pixel's intensity; also, the supports associated with neighboring pixels overlap, and therefore a single small piece of a primitive may actually contribute to several different pixels (see Fig. 3.58). This overlap also ensures that there are no areas of the grid not covered by some pixel, which would be the case if the circular pixels had a radius of only one-half of a grid unit.<sup>12</sup>

As with the box filter, the sum of all intensity contributions for the cone filter is the volume under the cone and above the intersection of the cone's base and the primitive; this volume  $W_s$  is a vertical section of the cone, as shown in Fig. 3.58. As with the box filter, the height of the cone is first normalized so that the volume under the entire cone is 1; this allows a pixel whose support is completely covered by a primitive to be displayed at maximum intensity. Although contributions from areas of the primitive far from the pixel's center but still intersecting the support are rather small, a pixel whose center is sufficiently close to a line receives some intensity contribution from that line. Conversely, a pixel that, in the square-geometry model, was entirely covered by a line of unit thickness<sup>13</sup> is not quite as bright as it used to be. The net effect of weighted area sampling is to decrease the contrast between adjacent pixels, in order to provide smoother transitions. In particular, with weighted area sampling, a horizontal or vertical line of unit thickness has more than 1 pixel

<sup>12</sup>As noted in Section 3.2.1, pixels displayed on a CRT are roughly circular in cross-section, and adjacent pixels typically overlap; the model of overlapping circles used in weighted area sampling, however, is not directly related to this fact and holds even for display technologies, such as the plasma panel, in which the physical pixels are actually nonoverlapping square tiles.

<sup>13</sup>We now say a "a line of unit thickness" rather than "a line 1 pixel thick" to make it clear that the unit of line width is still that of the SRGP grid, whereas the pixel's support has grown to have a two-unit diameter.



**Fig. 3.58** Cone filter for circular pixel with diameter of two grid units.

intensified in each column or row, which would not be the case for unweighted area sampling.

The conical filter has two useful properties: rotational symmetry and linear decrease of the function with radial distance. We prefer rotational symmetry because it not only makes area calculations independent of the angle of the line, but also is theoretically optimal, as shown in Chapter 14. We also show there, however, that the cone's linear slope (and its radius) are only an approximation to the optimal filter function, although the cone filter is still better than the box filter. Optimal filters are computationally most expensive, box filters least, and therefore cone filters are a very reasonable compromise between cost and quality. The dramatic difference between an unfiltered and filtered line drawing is shown in Fig. 3.59. Notice how the problems of indistinct lines and moiré patterns are greatly ameliorated by filtering. Now we need to integrate the cone filter into our scan-conversion algorithms.

The most important idea of this chapter is that, since speed is essential in interactive raster graphics, incremental scan-conversion algorithms using only integer operations in their inner loops are usually the best. The basic algorithms can be extended to handle thickness, as well as patterns for boundaries or for filling areas. Whereas the basic algorithms that convert single-pixel-wide primitives try to minimize the error between chosen pixels on the Cartesian grid and the ideal primitive defined on the plane, the algorithms for thick primitives can trade off quality and "correctness" for speed. Although much of 2D raster graphics today still operates, even on color displays, with single-bit-per-pixel primitives, we expect that techniques for real-time antialiasing will soon become prevalent.

## EXERCISES

- 3.1 Implement the special-case code for scan converting horizontal and vertical lines, and lines with slopes of  $\pm 1$ .
- 3.2 Modify the midpoint algorithm for scan converting lines (Fig. 3.8) to handle lines at any angle.
- 3.3 Show why the point-to-line error is always  $\leq \frac{1}{2}$  for the midpoint line scan-conversion algorithm.
- 3.4 Modify the midpoint algorithm for scan converting lines of Exercise 3.2 to handle endpoint order and intersections with clip edges, as discussed in Section 3.2.3.
- 3.5 Modify the midpoint algorithm for scan converting lines (Exercise 3.2) to write pixels with varying intensity as a function of line slope.
- 3.6 Modify the midpoint algorithm for scan converting lines (Exercise 3.2) to deal with endpoints that do not have integer coordinates—this is easiest if you use floating point throughout your algorithm. As a more difficult exercise, handle lines of *rational* endpoints using only integers.
- 3.7 Determine whether the midpoint algorithm for scan converting lines (Exercise 3.2) can take advantage of symmetry by using the decision variable  $d$  to draw simultaneously from both ends of the line toward the center. Does your algorithm consistently accommodate the case of equal error on an arbitrary choice that arises when  $dx$  and  $dy$  have a largest common factor  $c$  and  $dx/c$  is even and  $dy/c$  is odd ( $0 < dy < dx$ ), as in the line between  $(0, 0)$  and  $(24, 9)$ ? Does it deal with the subset case in which  $dx$  is an integer multiple of  $2dy$ , such as for the line between  $(0, 0)$  and  $(16, 4)$ ? (Contributed by J. Bresenham.)
- 3.8 Show how polylines may share more than vertex pixels. Develop an algorithm that avoids writing pixels twice. Hint: Consider scan conversion and writing to the canvas in **xor** mode as separate phases.
- 3.9 Expand the pseudocode for midpoint ellipse scan conversion of Fig. 3.21 to code that tests properly for various conditions that may arise.
- 3.10 Apply the technique of forward differencing shown for circles in Section 3.3.2 to develop the second-order forward differences for scan converting standard ellipses. Write the code that implements this technique.
- 3.11 Develop an alternative to the midpoint circle scan-conversion algorithm of Section 3.3.2 based on a piecewise-linear approximation of the circle with a polyline.
- 3.12 Develop an algorithm for scan converting unfilled rounded rectangles with a specified radius for the quarter-circle corners.

- 3.13** Write a scan-conversion procedure for solidly filled upright rectangles at arbitrary screen positions that writes a bilevel frame buffer efficiently, an entire word of pixels at a time.
- 3.14** Construct examples of pixels that are "missing" or written multiple times, using the rules of Section 3.6. Try to develop alternative, possibly more complex, rules that do not draw shared pixels on shared edges twice, yet do not cause pixels to be missing. Are these rules worth the added overhead?
- 3.15** Implement the pseudocode of Section 3.6 for polygon scan conversion, taking into account in the span bookkeeping of potential sliver polygons.
- 3.16** Develop scan-conversion algorithms for triangles and trapezoids that take advantage of the simple nature of these shapes. Such algorithms are common in hardware.
- 3.17** Investigate triangulation algorithms for decomposing an arbitrary, possibly concave or self-intersecting, polygon into a mesh of triangles whose vertices are shared. Does it help to restrict the polygon to being, at worse, concave without self-intersections or interior holes? (See also [PREP85].)
- 3.18** Extend the midpoint algorithm for scan converting circles (Fig. 3.16) to handle filled circles and circular wedges (for pie charts), using span tables.
- 3.19** Extend the midpoint algorithm for scan converting ellipses (Fig. 3.21) to handle filled elliptical wedges, using span tables.
- 3.20** Implement both absolute and relative anchor algorithms for polygon pattern filling, discussed in Section 3.9, and contrast them in terms of visual effect and computational efficiency.
- 3.21** Apply the technique of Fig. 3.30 for writing characters filled with patterns in opaque mode. Show how having a copyPixel with a write mask may be used to good advantage for this class of problems.
- 3.22** Implement a technique for drawing various symbols such as cursor icons represented by small bitmaps so that they can be seen regardless of the background on which they are written. Hint: Define a mask for each symbol that "encloses" the symbol—that is, that covers more pixels than the symbol—and that draws masks and symbols in separate passes.
- 3.23** Implement thick-line algorithms using the techniques listed in Section 3.9. Contrast their efficiency and the quality of the results they produced.
- 3.24** Extend the midpoint algorithm for scan converting circles (Fig. 3.16) to handle thick circles.
- 3.25** Implement a thick-line algorithm that accommodates line style as well as pen style and pattern.
- 3.26** Implement scissoring as part of scan converting lines and unfilled polygons, using the fast-scan-plus-backtracking technique of checking every  $i$ th pixel. Apply the technique to filled and thick lines and to filled polygons. For these primitives, contrast the efficiency of this type of on-the-fly clipping with that of analytical clipping.
- 3.27** Implement scissoring as part of scan converting unfilled and filled circles and ellipses. For these primitives, contrast the feasibility and efficiency of this type of on-the-fly clipping with that of analytical clipping.
- 3.28** Modify the Cohen-Sutherland line-clipping algorithm of Fig. 3.41 to avoid recalculation of slopes during successive passes.
- 3.29** Contrast the efficiency of the Sutherland-Cohen and Cyrus-Beck algorithms for several typical and atypical cases, using instruction counting. Are horizontal and vertical lines handled optimally?
- 3.30** Consider a convex polygon with  $n$  vertices being clipped against a clip rectangle. What is the maximum number of vertices in the resulting clipped polygon? What is the minimum number? Consider the same problem for a concave polygon. How many polygons might result? If a single polygon results, what is the largest number of vertices it might have?

- 3.31 Explain why the Sutherland–Hodgman polygon-clipping algorithm works for only convex clipping regions.
- 3.32 Devise a strategy for subdividing a pixel and counting the number of subpixels covered (at least to a significant degree) by a line, as part of a line-drawing algorithm using unweighted area sampling.
- 3.33 Create tables with various decreasing functions of the distance between pixel center and line center. Use them in the antialiased line algorithm of Fig. 3.62. Contrast the results produced with those produced by a box-filtered line.
- 3.34 Generalize the antialiasing techniques for lines to polygons. How might you handle nonpolygonal boundaries of curved primitives and characters?

# Viewing And Transformations

S. No.	Topic	Contents
4.	<b>2D and 3D Geometric Transformations, 2D and 3D Viewing Transformations , Vanishing points</b>	<b>Sections 2.1 - 2.21 Sections 3.1 - 3.17</b>

---

## TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

---

 CHAPTER  
**TWO**


---

**TWO-DIMENSIONAL TRANSFORMATIONS**
**2-1 INTRODUCTION**

We begin our study of the fundamentals of the mathematics underlying computer graphics by considering the representation and transformation of points and lines. Points and the lines which join them, along with an appropriate drawing algorithm, are used to represent objects or to display information graphically. The ability to transform these points and lines is basic to computer graphics. When visualizing an object, it may be desirable to scale, rotate, translate, distort or develop a perspective view of the object. All of these transformations can be accomplished using the mathematical techniques discussed in this and the next chapter.

**2-2 REPRESENTATION OF POINTS**

A point is represented in two dimensions by its coordinates. These two values are specified as the elements of a 1-row, 2-column matrix:

$$\begin{bmatrix} x & y \end{bmatrix}$$

In three dimensions a  $1 \times 3$  matrix

$$\begin{bmatrix} x & y & z \end{bmatrix}$$

is used. Alternately, a point is represented by a 2-row, 1-column matrix

$$\begin{bmatrix} x \\ y \end{bmatrix}$$

in two dimensions or by

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

in three dimensions. Row matrices like

$$\begin{bmatrix} x & y \end{bmatrix}$$

or column matrices like

$$\begin{bmatrix} x \\ y \end{bmatrix}$$

are frequently called position vectors. In this book a row matrix formulation of the position vectors is used.

A series of points, each of which is a position vector relative to some coordinate system, is stored in a computer as a matrix or array of numbers. The position of these points is controlled by manipulating the matrix which defines the points. Lines are drawn between the points to generate lines, curves or pictures.

### 2-3 TRANSFORMATIONS AND MATRICES

Matrix elements can represent various quantities, such as a number store, a network or the coefficients of a set of equations. The rules of matrix algebra define allowable operations on these matrices (see Appendix B). Many physical problems lead to a matrix formulation. For models of physical systems, the problem is formulated as: given the matrices  $[A]$  and  $[B]$  find the solution matrix  $[T]$ , i.e.,  $[A][T] = [B]$ . In this case the solution is  $[T] = [A]^{-1}[B]$ , where  $[A]^{-1}$  is the inverse of the square matrix  $[A]$  (see Ref. 2-1).

An alternate interpretation is to treat the matrix  $[T]$  as a geometric operator. Here matrix multiplication is used to perform a geometrical transformation on a set of points represented by the position vectors contained in  $[A]$ . The matrices  $[A]$  and  $[T]$  are assumed known. It is required to determine the elements of the matrix  $[B]$ . The interpretation of the matrix  $[T]$  as a geometrical operator is the foundation of mathematical transformations useful in computer graphics.

### 2-4 TRANSFORMATION OF POINTS

Consider the results of the multiplication of a matrix  $\begin{bmatrix} x & y \end{bmatrix}$  containing the coordinates of a point  $P$  and a general  $2 \times 2$  transformation matrix:

$$[X][T] = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} (ax + cy) & (bx + dy) \end{bmatrix} = \begin{bmatrix} x^* & y^* \end{bmatrix} \quad (2-1)$$

This mathematical notation means that the initial coordinates  $x$  and  $y$  are transformed to  $x^*$  and  $y^*$ , where  $x^* = (ax + cy)$  and  $y^* = (bx + dy)$ .<sup>†</sup> We are interested

<sup>†</sup>See Appendix B for the details of matrix multiplication.

in the implications of considering  $x^*$  and  $y^*$  as the transformed coordinates of the point  $P$ . We begin by investigating several special cases.

Consider the case where  $a = d = 1$  and  $c = b = 0$ . The transformation matrix  $[T]$  then reduces to the identity matrix. Thus,

$$[X][T] = [x \ y] \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = [x \ y] = [x^* \ y^*] \quad (2-2)$$

and no change in the coordinates of the point  $P$  occurs. Since in matrix algebra multiplying by the identity matrix is equivalent to multiplying by 1 in ordinary algebra, this result is expected.

Next consider  $d = 1$ ,  $b = c = 0$ , i.e.,

$$[X][T] = [x \ y] \begin{bmatrix} a & 0 \\ 0 & 1 \end{bmatrix} = [ax \ y] = [x^* \ y^*] \quad (2-3)$$

which, since  $x^* = ax$ , produces a scale change in the  $x$  component of the position vector. The effect of this transformation is shown in Fig. 2-1a. Now consider  $b = c = 0$ , i.e.,

$$[X][T] = [x \ y] \begin{bmatrix} a & 0 \\ 0 & d \end{bmatrix} = [ax \ dy] = [x^* \ y^*] \quad (2-4)$$

This yields a scaling of both the  $x$  and  $y$  coordinates of the original position vector  $P$ , as shown in Fig. 2-1b. If  $a \neq d$ , then the scalings are not equal. If  $a = d > 1$ , then a pure enlargement or scaling of the coordinates of  $P$  occurs. If  $0 < a = d < 1$ , then a compression of the coordinates of  $P$  occurs.

If  $a$  and/or  $d$  are negative, reflections through an axis or plane occur. To see this, consider  $b = c = 0$ ,  $d = 1$  and  $a = -1$ . Then

$$[X][T] = [x \ y] \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} = [-x \ y] = [x^* \ y^*] \quad (2-5)$$

and a reflection through the  $y$ -axis results, as shown in Fig. 2-1c. If  $b = c = 0$ ,  $a = 1$ , and  $d = -1$ , then a reflection through the  $x$ -axis occurs. If  $b = c = 0$ ,  $a = d < 0$ , then a reflection through the origin occurs. This is shown in Fig. 2-1d, with  $a = -1$ ,  $d = -1$ . Note that both reflection and scaling of the coordinates involve only the diagonal terms of the transformation matrix.

Now consider the effects of the off-diagonal terms. First consider  $a = d = 1$  and  $c = 0$ . Thus,

$$[X][T] = [x \ y] \begin{bmatrix} 1 & b \\ 0 & 1 \end{bmatrix} = [x \ (bx + y)] = [x^* \ y^*] \quad (2-6)$$

Note that the  $x$  coordinate of the point  $P$  is unchanged, while  $y^*$  depends linearly on the original coordinates. This effect is called shear, as shown in Fig. 2-1e.

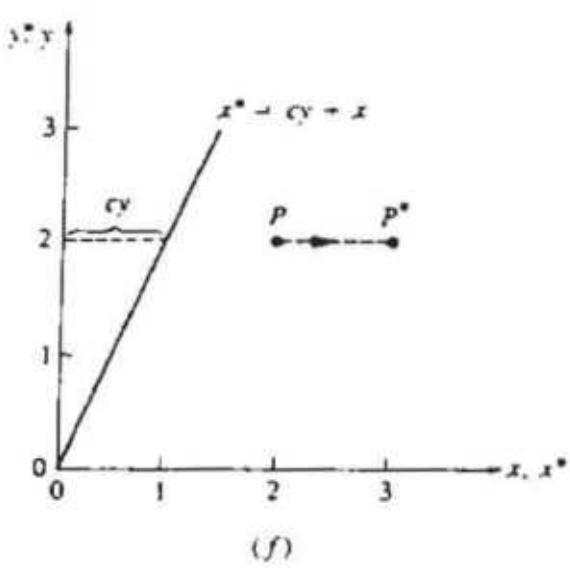
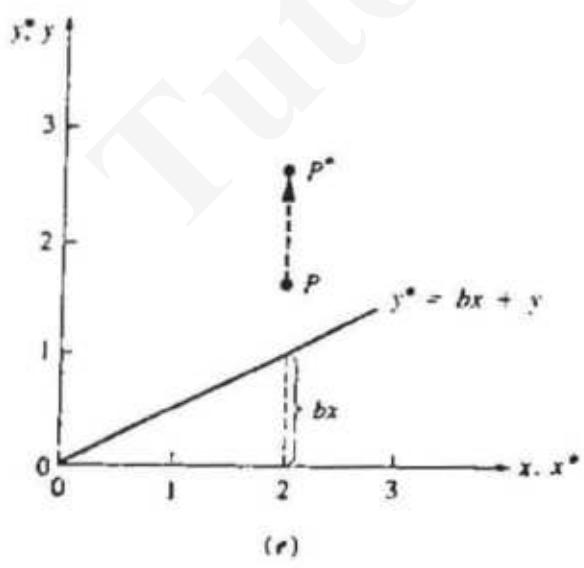
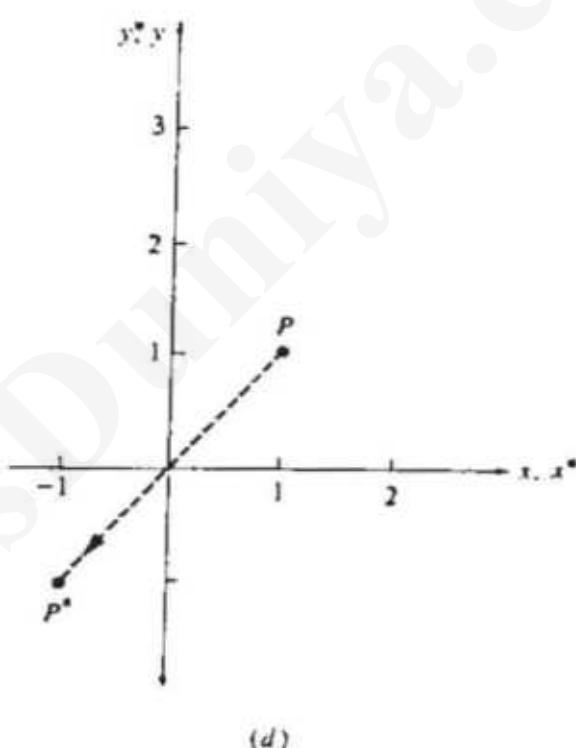
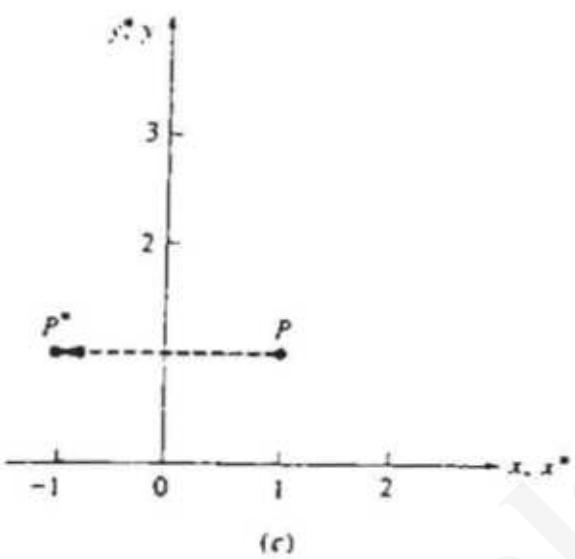
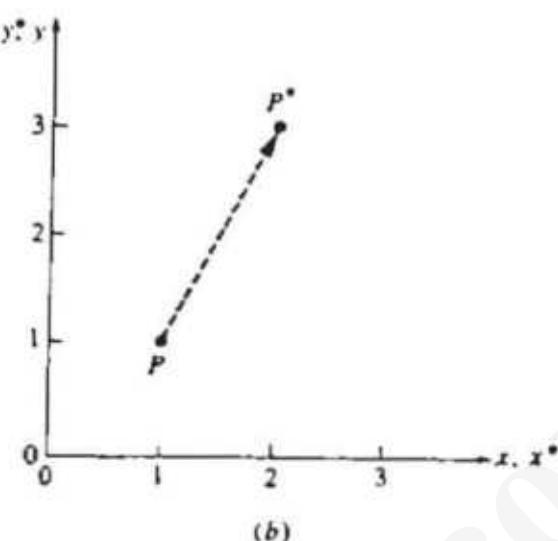
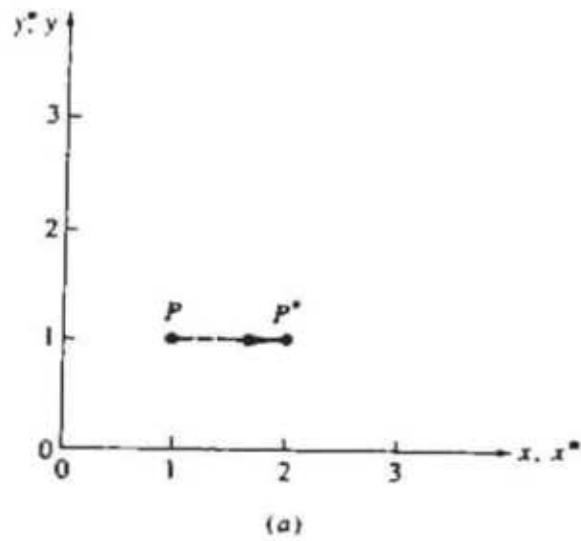


Figure 2-1 Transformation of points.

Similarly, when  $a = d = 1$ ,  $b = 0$ , the transformation produces shear proportional to the  $y$  coordinate, as shown in Fig. 2-1f. Thus, we see that the off-diagonal terms produce a shearing effect on the coordinates of the position vector for  $P$ .

Before completing our discussion of the transformation of points, consider the effect of the general  $2 \times 2$  transformation given by Eq. (2-1) when applied to the origin, i.e.,

$$\begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} (ax + cy) & (bx + dy) \end{bmatrix} = \begin{bmatrix} x^* & y^* \end{bmatrix}$$

or for the origin,

$$\begin{bmatrix} 0 & 0 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 0 & 0 \end{bmatrix} = \begin{bmatrix} x^* & y^* \end{bmatrix}$$

Here we see that the origin is invariant under a general  $2 \times 2$  transformation. This is a limitation which will be overcome by the use of homogeneous coordinates.

## 2-5 TRANSFORMATION OF STRAIGHT LINES

A straight line can be defined by two position vectors which specify the coordinates of its end points. The position and orientation of the line joining these two points can be changed by operating on these two position vectors. The actual operation of drawing a line between two points depends on the display device used. Here, we consider only the mathematical operations on the position vectors of the end points.

A straight line between two points  $A$  and  $B$  in a two-dimensional plane is drawn in Fig. 2-2. The position vectors of points  $A$  and  $B$  are  $[A] = [0 \ 1]$  and  $[B] = [2 \ 3]$ , respectively. Now consider the transformation matrix

$$[T] = \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix} \quad (2-7)$$

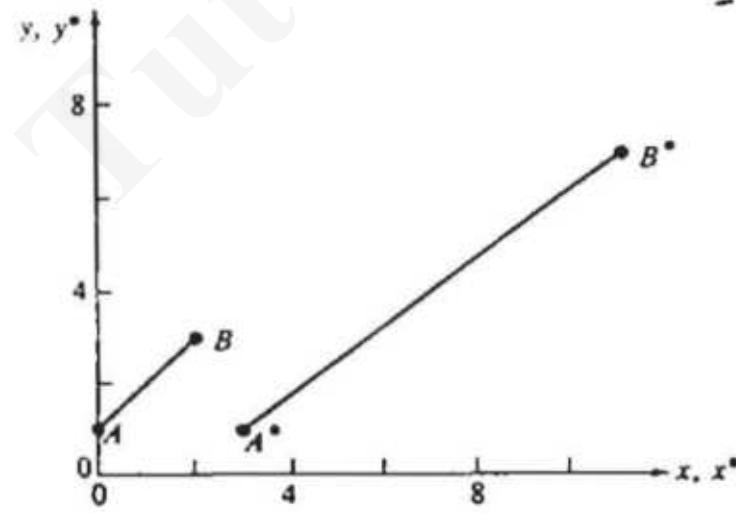


Figure 2-2 Transformation of straight lines.

which we recall from our previous discussion produces a shearing effect. Transforming the position vectors for  $A$  and  $B$  using  $[T]$  produces new transformed position vectors  $A^*$  and  $B^*$  given by

$$[A][T] = [0 \ 1] \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix} = [3 \ 1] = [A^*] \quad (2-8)$$

and

$$[B][T] = [2 \ 3] \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix} = [11 \ 7] = [B^*] \quad (2-9)$$

Thus, the resulting coordinates for  $A^*$  are  $x^* = 3$  and  $y^* = 1$ . Similarly,  $B^*$  is a new point with coordinates  $x^* = 11$  and  $y^* = 7$ . More compactly the line  $AB$  may be represented by the  $2 \times 2$  matrix

$$[L] = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$$

Matrix multiplication by  $[T]$  then yields

$$[L][T] = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 1 \\ 11 & 7 \end{bmatrix} = [L^*] \quad (2-10)$$

where the components of  $[L^*]$  represent the transformed position vectors  $[A^*]$  and  $[B^*]$ . The transformation of  $A$  to  $A^*$  and  $B$  to  $B^*$  is shown in Fig. 2-2. The initial axes are  $x, y$  and the transformed axes are  $x^*, y^*$ . Figure 2-2 shows that the shearing transformation  $[T]$  increased the length of the line and changed its orientation.

## 2-6 MIDPOINT TRANSFORMATION

Figure 2-2 shows that the  $2 \times 2$  transformation matrix (see Eq. 2-7) transforms the straight line  $y = x + 1$ , between points  $A$  and  $B$ , into another straight line  $y = (3/4)x - 5/4$ , between  $A^*$  and  $B^*$ . In fact a  $2 \times 2$  matrix transforms any straight line into a second straight line. Points on the second line have a one-to-one correspondence with points on the first line. We have already shown this to be true for the end points of the line. To further confirm this we consider the transformation of the midpoint of the straight line between  $A$  and  $B$ . Letting

$$[A] = [x_1 \ y_1] \quad [B] = [x_2 \ y_2] \quad \text{and} \quad [T] = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

and transforming both end points simultaneously yields

$$\begin{aligned} \begin{bmatrix} A \\ B \end{bmatrix} [T] &= \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \\ &= \begin{bmatrix} ax_1 + cy_1 & bx_1 + dy_1 \\ ax_2 + cy_2 & bx_2 + dy_2 \end{bmatrix} = \begin{bmatrix} A^* \\ B^* \end{bmatrix} \end{aligned} \quad (2-11)$$

Hence, the end points of the transformed line  $A^*B^*$  are

$$[ A^* ] = [ ax_1 + cy_1 \quad bx_1 + dy_1 ] = [ x_1^* \quad y_1^* ]$$

$$[ B^* ] = [ ax_2 + cy_2 \quad bx_2 + dy_2 ] = [ x_2^* \quad y_2^* ] \quad (2-12)$$

The midpoint of the transformed line  $A^*B^*$  calculated from the transformed end points is

$$\begin{aligned} [ x_m^* \quad y_m^* ] &= \left[ \frac{x_1^* + x_2^*}{2} \quad \frac{y_1^* + y_2^*}{2} \right] \\ &= \left[ \frac{(ax_1 + cy_1) + (ax_2 + cy_2)}{2} \quad \frac{(bx_1 + dy_1) + (bx_2 + dy_2)}{2} \right] \\ &= \left[ a \frac{(x_1 + x_2)}{2} + c \frac{(y_1 + y_2)}{2} \quad b \frac{(x_1 + x_2)}{2} + d \frac{(y_1 + y_2)}{2} \right] \end{aligned} \quad (2-13)$$

Returning to the original line  $AB$  the midpoint is

$$[ x_m \quad y_m ] = \left[ \frac{x_1 + x_2}{2} \quad \frac{y_1 + y_2}{2} \right] \quad (2-14)$$

Using  $[ T ]$  the transformation of the midpoint of  $AB$  is

$$\begin{aligned} [ x_m \quad y_m ] [ T ] &= \left[ \frac{x_1 + x_2}{2} \quad \frac{y_1 + y_2}{2} \right] \begin{bmatrix} a & b \\ c & d \end{bmatrix} \\ &= \left[ a \frac{(x_1 + x_2)}{2} + c \frac{(y_1 + y_2)}{2} \quad b \frac{(x_1 + x_2)}{2} + d \frac{(y_1 + y_2)}{2} \right] \end{aligned} \quad (2-15)$$

Comparing Eqs. (2-13) and (2-15) shows that they are identical. Consequently, the midpoint of the line  $AB$  transforms into the midpoint of the line  $A^*B^*$ . This process can be applied recursively to segments of the divided line. Thus, a one-to-one correspondence between points on the line  $AB$  and  $A^*B^*$  is assured.

### Example 2-1 Midpoint of a Line

Consider the line  $AB$  shown in Fig. 2-2. The position vectors of the end points are

$$[ A ] = [ 0 \quad 1 ] \quad [ B ] = [ 2 \quad 3 ]$$

The transformation

$$[ T ] = \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix}$$

yields the position vectors of the end points of the transformed line  $A^*B^*$  as

$$\begin{bmatrix} A \\ B \end{bmatrix} [ T ] = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 1 \\ 11 & 7 \end{bmatrix} = \begin{bmatrix} A^* \\ B^* \end{bmatrix}$$

The midpoint of  $A^*B^*$  is

$$\begin{bmatrix} x_m^* & y_m^* \end{bmatrix} = \left[ \frac{3+11}{2} \quad \frac{1+7}{2} \right] = [7 \quad 4]$$

The midpoint of the original untransformed line  $AB$  is

$$\begin{bmatrix} x_m & y_m \end{bmatrix} = \left[ \frac{0+2}{2} \quad \frac{1+3}{2} \right] = [1 \quad 2]$$

Transforming this midpoint yields

$$\begin{bmatrix} x_m & y_m \end{bmatrix} [T] = [1 \quad 2] \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix} = [7 \quad 4] = \begin{bmatrix} x_m^* & y_m^* \end{bmatrix}$$

which is the same as our previous result.

For computer graphics applications these results show that any straight line can be transformed into any other straight line in any position by simply transforming its end points and redrawing the line between the end points.

## 2-7 TRANSFORMATION OF PARALLEL LINES

When a  $2 \times 2$  matrix is used to transform a pair of parallel lines, the result is a second pair of parallel lines. To see this, consider a line between  $[A] = [x_1 \quad y_1]$  and  $[B] = [x_2 \quad y_2]$  and a line parallel to  $AB$  between  $E$  and  $F$ . To show that these lines and any transformation of them are parallel, examine the slopes of  $AB$ ,  $EF$ ,  $A^*B^*$  and  $E^*F^*$ . Since they are parallel, the slope of both  $AB$  and  $EF$  is

$$m = \frac{y_2 - y_1}{x_2 - x_1} \quad (2-16)$$

Transforming the end points of  $AB$  using a general  $2 \times 2$  transformation yields the end points of  $A^*B^*$ :

$$\begin{aligned} \begin{bmatrix} A \\ B \end{bmatrix} [T] &= \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \\ &= \begin{bmatrix} ax_1 + cy_1 & bx_1 + dy_1 \\ ax_2 + cy_2 & bx_2 + dy_2 \end{bmatrix} \\ &= \begin{bmatrix} x_1^* & y_1^* \\ x_2^* & y_2^* \end{bmatrix} = \begin{bmatrix} A^* \\ B^* \end{bmatrix} \end{aligned} \quad (2-17)$$

Using the transformed end points, the slope of  $A^*B^*$  is then

$$m^* = \frac{(bx_2 + dy_2) - (bx_1 + dy_1)}{(ax_2 + cy_2) - (ax_1 + cy_1)} = \frac{b(x_2 - x_1) + d(y_2 - y_1)}{a(x_2 - x_1) + c(y_2 - y_1)}$$

or

$$m^* = \frac{b + d \frac{(y_2 - y_1)}{(x_2 - x_1)}}{a + c \frac{(y_2 - y_1)}{(x_2 - x_1)}} = \frac{b + dm}{a + cm} \quad (2-18)$$

Since the slope  $m^*$  is independent of  $x_1, x_2, y_1$  and  $y_2$ , and since  $m$ ,  $a$ ,  $b$ ,  $c$  and  $d$  are the same for  $EF$  and  $AB$ , it follows that  $m^*$  is the same for both  $E^*F^*$  and  $A^*B^*$ . Thus, parallel lines remain parallel after transformation. This means that parallelograms transform into other parallelograms when operated on by a general  $2 \times 2$  transformation matrix. These simple results begin to show the power of using matrix multiplication to produce graphical effects.

## 2-8 TRANSFORMATION OF INTERSECTING LINES

When a general  $2 \times 2$  matrix is used to transform a pair of intersecting straight lines, the result is also a pair of intersecting straight lines. To see this consider a pair of lines, e.g., the dashed lines in Fig. 2-3, represented by

$$\begin{aligned} y &= m_1 x + b_1 \\ y &= m_2 x + b_2 \end{aligned}$$

Reformulating these equations in matrix notation yields

$$\begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} -m_1 & -m_2 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} b_1 & b_2 \end{bmatrix}$$

or

$$\begin{bmatrix} X \end{bmatrix} \begin{bmatrix} M \end{bmatrix} = \begin{bmatrix} B \end{bmatrix} \quad (2-19)$$

If a solution to this pair of equations exists, then the lines intersect. If not, then they are parallel. A solution can be obtained by matrix inversion. Specifically,

$$\begin{bmatrix} X_i \end{bmatrix} = \begin{bmatrix} x_i & y_i \end{bmatrix} = \begin{bmatrix} B \end{bmatrix} \begin{bmatrix} M \end{bmatrix}^{-1} \quad (2-20)$$

The inverse of  $[M]$  is

$$\begin{bmatrix} M \end{bmatrix}^{-1} = \begin{bmatrix} \frac{1}{m_2 - m_1} & \frac{m_2}{m_2 - m_1} \\ \frac{-1}{m_2 - m_1} & \frac{-m_1}{m_2 - m_1} \end{bmatrix} \quad (2-21)$$

since  $[M][M]^{-1} = [I]$ , the identity matrix. Hence, the intersection of the two lines is

$$\begin{aligned} \begin{bmatrix} X_i \end{bmatrix} &= \begin{bmatrix} x_i & y_i \end{bmatrix} = \begin{bmatrix} b_1 & b_2 \end{bmatrix} \begin{bmatrix} \frac{1}{m_2 - m_1} & \frac{m_2}{m_2 - m_1} \\ \frac{-1}{m_2 - m_1} & \frac{-m_1}{m_2 - m_1} \end{bmatrix} \\ \begin{bmatrix} X_i \end{bmatrix} &= \begin{bmatrix} x_i & y_i \end{bmatrix} = \begin{bmatrix} \frac{b_1 - b_2}{m_2 - m_1} & \frac{b_1 m_2 - b_2 m_1}{m_2 - m_1} \end{bmatrix} \quad (2-22) \end{aligned}$$

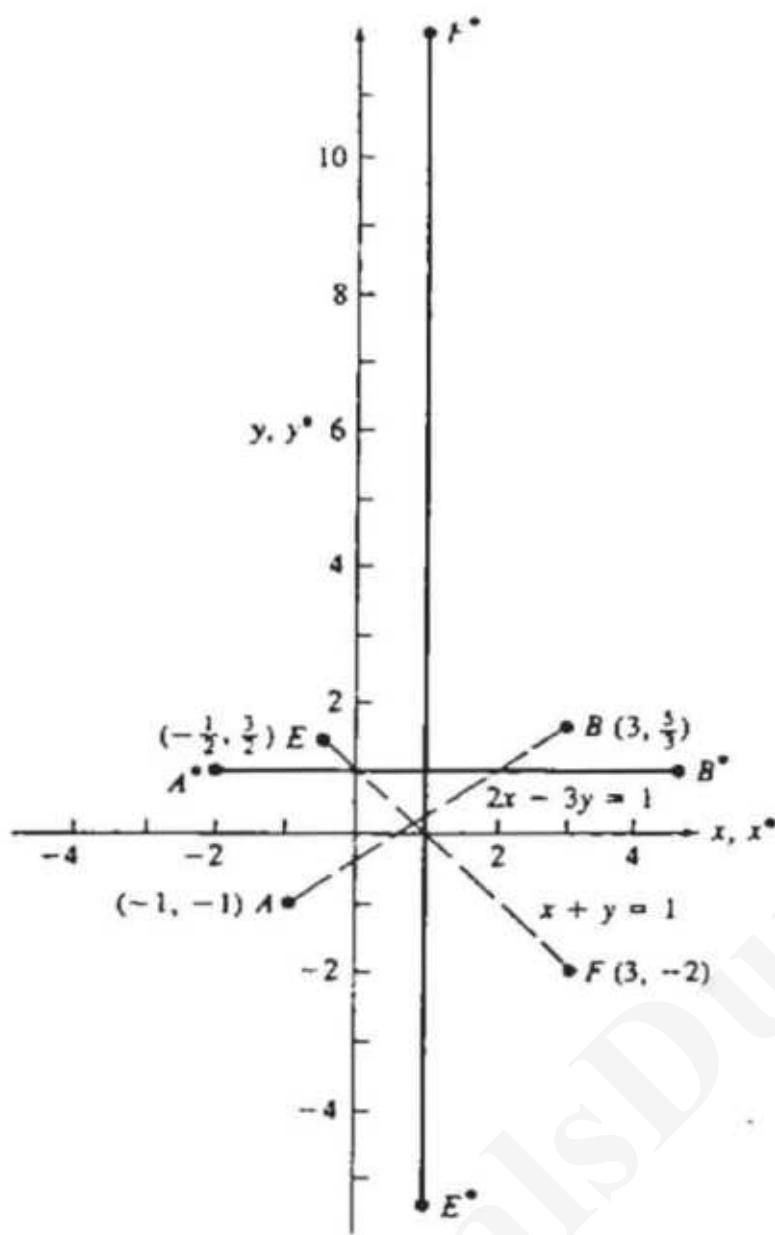


Figure 2-3 Transformation of intersecting lines.

If these two lines are now transformed using a general  $2 \times 2$  transformation matrix given by

$$[T] = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

then they have the form

$$y^* = m_1^* x^* + b_1^*$$

$$y^* = m_2^* x^* + b_2^*$$

It is relatively easy to show that

$$m_i^* = \frac{b + dm_i}{a + cm_i} \quad (2-23)$$

and  $b_i^* = b_i(d - cm_i^*) = b_i \frac{ad - bc}{a + cm_i} \quad i = 1, 2 \quad (2-24)$

The intersection of the transformed lines is obtained in the same manner as that for the untransformed lines. Thus,

$$\begin{aligned}[X_i^*] &= [x_i^* \quad y_i^*] \\ &= \left[ \frac{b_1^* - b_2^*}{m_2^* - m_1^*} \quad \frac{b_1^* m_2^* - b_2^* m_1^*}{m_2^* - m_1^*} \right]\end{aligned}$$

Rewriting the components of the intersection point using Eqs. (2-23) and (2-24) yields

$$\begin{aligned}[X_i^*] &= [x_i^* \quad y_i^*] \\ &= \left[ \frac{a(b_1 - b_2) + c(b_1 m_2 - b_2 m_1)}{m_2 - m_1} \quad \frac{b(b_1 - b_2) + d(b_1 m_2 - b_2 m_1)}{m_2 - m_1} \right] \quad (2-25)\end{aligned}$$

Returning now to the untransformed intersection point  $[x_i \quad y_i]$  and applying the same general  $2 \times 2$  transformation we have

$$\begin{aligned}[x_i^* \quad y_i^*] &= [x_i \quad y_i] [T] \\ &= \left[ \frac{b_1 - b_2}{m_2 - m_1} \quad \frac{b_1 m_2 - b_2 m_1}{m_2 - m_1} \right] \begin{bmatrix} a & b \\ c & d \end{bmatrix} \\ &= \left[ \frac{a(b_1 - b_2) + c(b_1 m_2 - b_2 m_1)}{m_2 - m_1} \quad \frac{b(b_1 - b_2) + d(b_1 m_2 - b_2 m_1)}{m_2 - m_1} \right] \quad (2-26)\end{aligned}$$

Comparing Eqs. (2-25) and (2-26) shows that they are identical. Consequently, the intersection point transforms into the intersection point.

### Example 2-2 Intersecting Lines

Consider the two dashed lines  $AB$  and  $EF$  shown in Fig. 2-3 with end points

$$[A] = [-1 \quad -1] \quad [B] = [3 \quad 5/3]$$

and

$$[E] = [-1/2 \quad 3/2] \quad [F] = [3 \quad -2]$$

The equation of the line  $AB$  is  $-(2/3)x + y = -(1/3)$  and of the line  $EF$ ,  $x + y = 1$ . In matrix notation the pair of lines is represented by

$$[x \quad y] \begin{bmatrix} -2/3 & 1 \\ 1 & 1 \end{bmatrix} = [-1/3 \quad 1]$$

Using matrix inversion (see Eq. 2-21) the intersection of these lines is

$$\begin{aligned}[x_i \quad y_i] &= [-1/3 \quad 1] \begin{bmatrix} -3/5 & -3/5 \\ 3/5 & 2/5 \end{bmatrix} \\ &= [4/5 \quad 1/5]\end{aligned}$$

Now consider the transformation of these lines using

$$[T] = \begin{bmatrix} 1 & 2 \\ 1 & -3 \end{bmatrix}$$

The resulting lines are shown as  $A^*B^*$  and  $E^*F^*$  in Fig. 2-3. In matrix form the equations of the transformed lines are

$$[x^* \ y^*] \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = [1 \ 1]$$

with intersection point at  $[x_i^* \ y_i^*] = [1 \ 1]$ .

Transforming the intersection point of the untransformed lines yields

$$\begin{aligned}[x_i^* \ y_i^*] &= [x_i \ y_i][T] \\ &= [4/5 \ 1/5] \begin{bmatrix} 1 & 2 \\ 1 & -3 \end{bmatrix} = [1 \ 1]\end{aligned}$$

which is identical to the intersection point of the transformed lines.

---

Examination of Fig. 2-3 and Ex. 2-2 shows that the original pair of untransformed dashed lines  $AB$  and  $EF$  are *not* perpendicular. However, the transformed solid lines  $A^*B^*$  and  $E^*F^*$  are perpendicular. Thus, the transformation  $[T]$  changed a pair of intersecting nonperpendicular lines into a pair of intersecting perpendicular lines. By implication,  $[T]^{-1}$  the inverse of the transformation, changes a pair of intersecting perpendicular lines into a pair of intersecting nonperpendicular lines. This effect can have disastrous geometrical consequences. It is thus of considerable interest to determine under what conditions perpendicular lines transform into perpendicular lines. We will return to this question in Sec. 2-14 when a little more background has been presented.

Additional examination of Fig. 2-3 and Ex. 2-2 shows that the transformation  $[T]$  involved a rotation, a reflection and a scaling. Let's consider each of these effects individually.

## 2-9 ROTATION

Consider the plane triangle  $ABC$  shown in Fig. 2-4. The triangle  $ABC$  is rotated through  $90^\circ$  about the origin in a counterclockwise sense by the transformation

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

If we use a  $3 \times 2$  matrix containing the  $x$  and  $y$  coordinates of the triangle's vertices, then

$$\begin{bmatrix} 3 & -1 \\ 4 & 1 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ -1 & 4 \\ -1 & 2 \end{bmatrix}$$

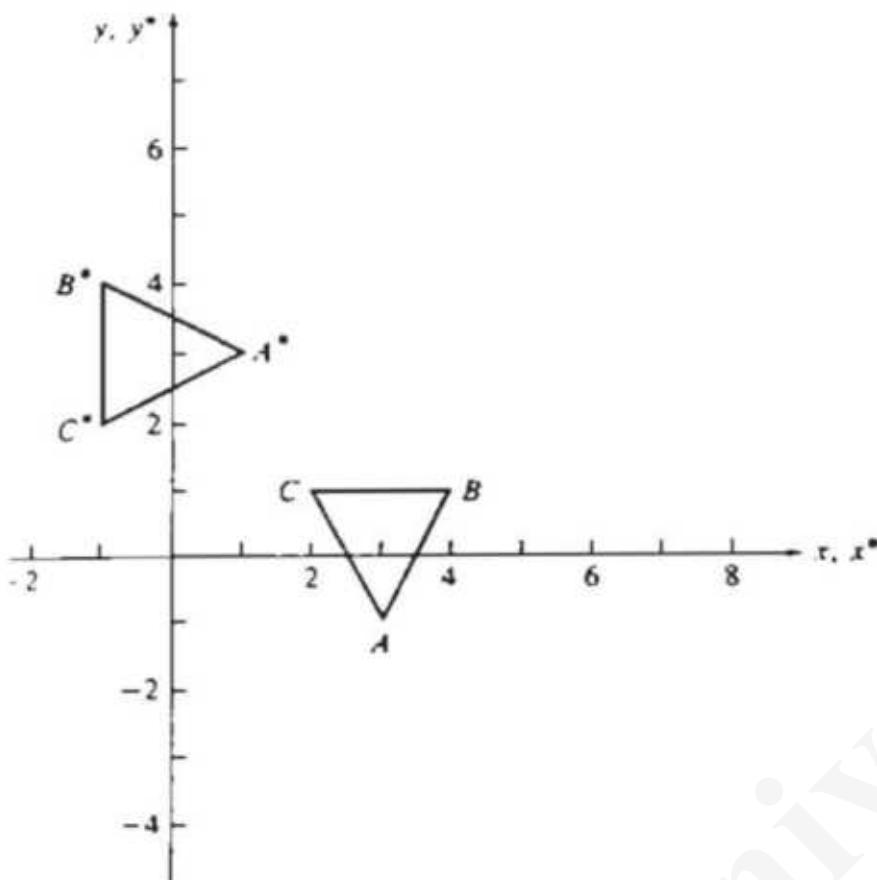


Figure 2-4 Rotation.

which produces the triangle  $A^*B^*C^*$ . A  $180^\circ$  rotation about the origin is obtained by using the transformation

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

and a  $270^\circ$  rotation about the origin by using

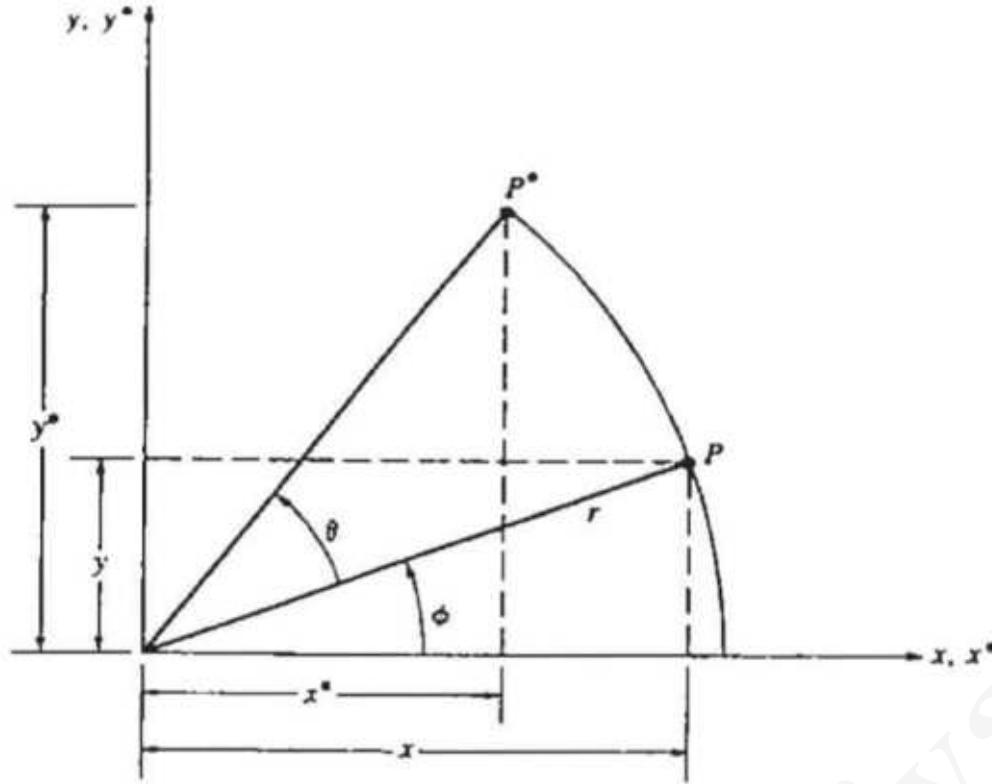
$$[T] = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

Of course, the identity matrix

$$[T] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

corresponds to a rotation about the origin of either  $0^\circ$  or  $360^\circ$ . Note that neither scaling nor reflection has occurred in these examples.

These example transformations produce specific rotations about the origin:  $0^\circ, 90^\circ, 180^\circ, 270^\circ$ . What about rotation about the origin by an arbitrary angle  $\theta$ ? To obtain this result consider the position vector from the origin to the point  $P$  shown in Fig. 2-5. The length of the vector is  $r$  at an angle  $\phi$  to the  $x$ -axis. The position vector  $P$  is rotated about the origin by the angle  $\theta$  to  $P^*$ .



**Figure 2-5** Rotation of a position vector.

Writing the position vectors for  $P$  and  $P^*$  we have

$$P = [x \ y] = [r \cos \phi \ r \sin \phi]$$

and  $P^* = [x^* \ y^*] = [r \cos(\phi + \theta) \ r \sin(\phi + \theta)]$

Using the sum of the angles formulas<sup>†</sup> allows writing  $P^*$  as

$$P^* = [x^* \ y^*] = [r(\cos \phi \cos \theta - \sin \phi \sin \theta) \ r(\cos \phi \sin \theta + \sin \phi \cos \theta)]$$

Using the definitions of  $x$  and  $y$  allows rewriting  $P^*$  as

$$P^* = [x^* \ y^*] = [x \cos \theta - y \sin \theta \ x \sin \theta + y \cos \theta]$$

Thus, the transformed point has components

$$x^* = x \cos \theta - y \sin \theta \quad (2-27a)$$

$$y^* = x \sin \theta + y \cos \theta \quad (2-27b)$$

In matrix form

$$\begin{aligned} [X^*] &= [X][T] = [x^* \ y^*] \\ &= [x \ y] \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \end{aligned} \quad (2-28)$$

<sup>†</sup> $\cos(\phi \pm \theta) = \cos \phi \cos \theta \mp \sin \phi \sin \theta$   
 $\sin(\phi \pm \theta) = \cos \phi \sin \theta \pm \sin \phi \cos \theta$

Thus, the transformation for a general rotation about the origin by an arbitrary angle  $\theta$  is

$$[T] = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \quad (2-29)$$

(rotations are positive counterclockwise about the origin, as shown in Fig. 2-5).

Evaluation of the determinant of the general rotation matrix yields

$$\det [T] = \cos^2 \theta + \sin^2 \theta = 1 \quad (2-30)$$

In general, transformations with a determinant identically equal to +1 yield pure rotations.

Suppose now that we wish to rotate the point  $P^*$  back to  $P$ , i.e., perform the inverse transformation. The required rotation angle is obviously  $-\theta$ . From Eq. (2-29) the required transformation matrix is

$$[T]^{-1} = \begin{bmatrix} \cos(-\theta) & \sin(-\theta) \\ -\sin(-\theta) & \cos(-\theta) \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (2-31)$$

since  $\cos(-\theta) = \cos \theta$  and  $\sin(-\theta) = -\sin \theta$ .  $[T]^{-1}$  is a formal way of writing 'the inverse of'  $[T]$ . We can show that  $[T]^{-1}$  is the inverse of  $[T]$  by recalling that the product of a matrix and its inverse yields the identity matrix. Here,

$$\begin{aligned} [T][T]^{-1} &= \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \\ &= \begin{bmatrix} \cos^2 \theta + \sin^2 \theta & -\cos \theta \sin \theta + \cos \theta \sin \theta \\ -\cos \theta \sin \theta + \cos \theta \sin \theta & \cos^2 \theta + \sin^2 \theta \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = [I] \end{aligned}$$

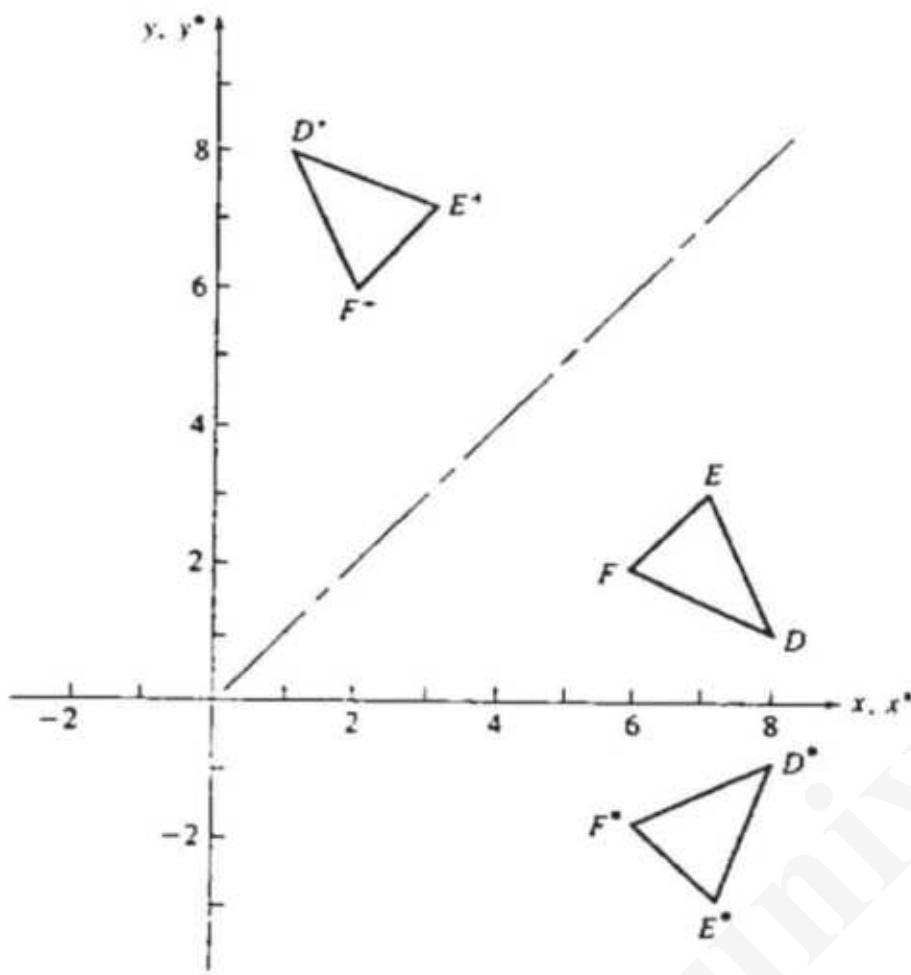
where  $[I]$  is the identity matrix.

Examining Eqs. (2-29) and (2-31) reveals another interesting and useful result. Recall that the transpose of a matrix is obtained by interchanging its rows and columns. Forming the transpose of  $[T]$ , i.e.,  $[T]^T$ , and comparing it with  $[T]^{-1}$  shows that

$$[T]^T = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} = [T]^{-1} \quad (2-32)$$

The inverse of the general rotation matrix  $[T]$  is its transpose. Since formally determining the inverse of a matrix is more computationally expensive than determining its transpose, Eq. (2-32) is an important and useful result. In general, the inverse of any pure rotation matrix, i.e., one with a determinant identically equal to +1, is its transpose.<sup>†</sup>

<sup>†</sup>Such matrices are said to be orthogonal.



**Figure 2-6** Reflection.

## 2-10 REFLECTION

Whereas a pure two-dimensional rotation in the  $xy$  plane occurs entirely in the two-dimensional plane about an axis normal to the  $xy$  plane, a reflection is a  $180^\circ$  rotation out into three space and back into two space about an axis in the  $xy$  plane. Two reflections of the triangle  $DEF$  are shown in Fig. 2-6. A reflection about  $y = 0$ , the  $x$ -axis, is obtained by using

$$[T] = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad (2-33)$$

In this case the new vertices  $D^*E^*F^*$  for the triangle are given by

$$\begin{bmatrix} 8 & 1 \\ 7 & 3 \\ 6 & 2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 8 & -1 \\ 7 & -3 \\ 6 & -2 \end{bmatrix}$$

Similarly reflection about  $x = 0$ , the  $y$ -axis, is given by

$$[T] = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \quad (2-34)$$

A reflection about the line  $y = x$  occurs for

$$[T] = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad (2-35)$$

The transformed, new vertices  $D^+E^+F^+$  are given by

$$\begin{bmatrix} 8 & 1 \\ 7 & 3 \\ 6 & 2 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 8 \\ 3 & 7 \\ 2 & 6 \end{bmatrix}$$

Similarly, a reflection about the line  $y = -x$  is given by

$$[T] = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \quad (2-36)$$

Each of these reflection matrices has a determinant that is identically  $-1$ . In general, if the determinant of a transformation matrix is identically  $-1$ , then the transformation produces a pure reflection.

If two pure reflection transformations about lines passing through the origin are applied successively, the result is a pure rotation about the origin. To see this, consider the following example.

### Example 2-3 Reflection and Rotation

Consider the triangle  $ABC$  shown in Fig. 2-7, first reflected about the  $x$  axis (see Eq. 2-33) and then about the line  $y = -x$  (see Eq. 2-36). Specifically, the result of the reflection about the  $x$ -axis is

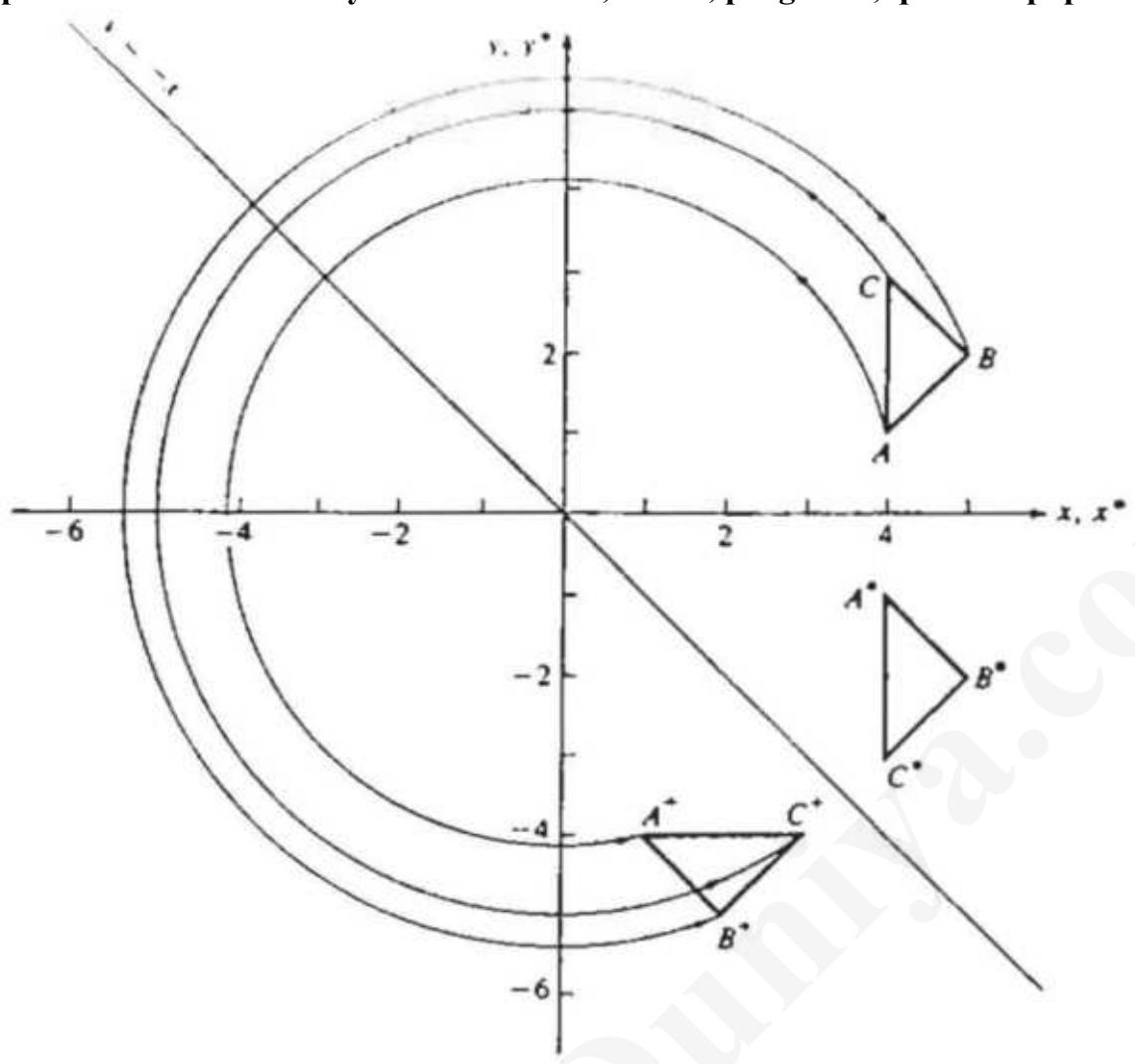
$$[X^*] = [X][T_1] = \begin{bmatrix} 4 & 1 \\ 5 & 2 \\ 4 & 3 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 4 & -1 \\ 5 & -2 \\ 4 & -3 \end{bmatrix}$$

Reflecting the triangle  $A^*B^*C^*$  about the line  $y = -x$  yields

$$[X^+] = [X^*][T_2] = \begin{bmatrix} 4 & -1 \\ 5 & -2 \\ 4 & -3 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & -4 \\ 2 & -5 \\ 3 & -4 \end{bmatrix}$$

Rotation about the origin by an angle  $\theta = 270^\circ$  (see Eq. 2-29) yields the identical result, i.e.,

$$[X^+] = [X][T_3] = \begin{bmatrix} 4 & 1 \\ 5 & 2 \\ 4 & 3 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & -4 \\ 2 & -5 \\ 3 & -4 \end{bmatrix}$$



**Figure 2-7** Combined reflections yield rotations.

Note that the reflection matrices given above in Eqs. (2-33) and (2-36) are orthogonal; i.e., the transpose is also the inverse. For example,

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

## 2-11 SCALING

Recalling our discussion of the transformation of points, we see that scaling is controlled by the magnitude of the two terms on the primary diagonal of the matrix. If the matrix

$$[T] = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

is used as an operator on the vertices of a triangle, a '2-times' enlargement, or uniform scaling, occurs about the origin. If the magnitudes are unequal, a distortion occurs. These effects are shown in Fig. 2-8. Triangle ABC is transformed by

$$[T] = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

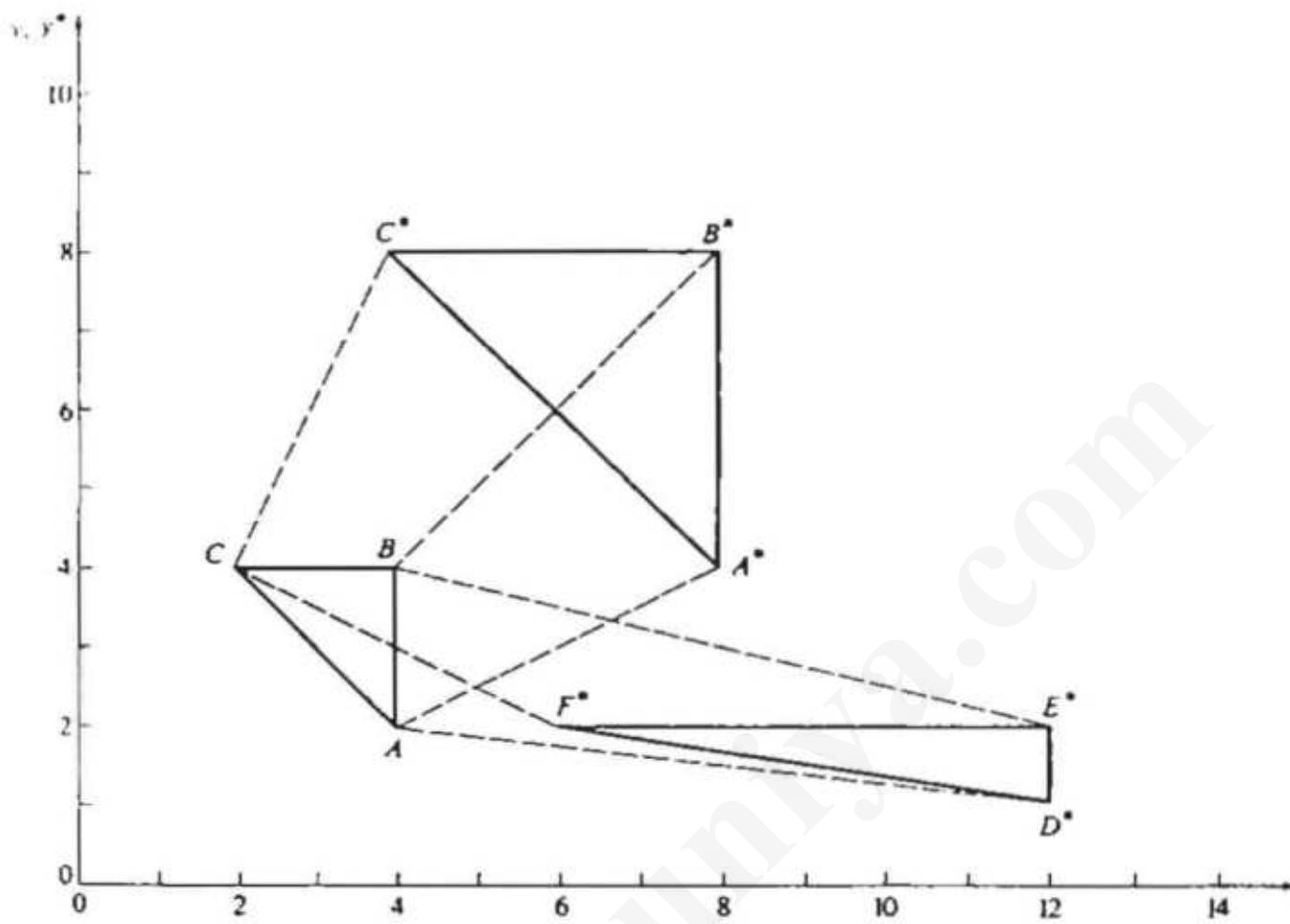


Figure 2-8 Uniform and nonuniform scaling or distortion.

to yield  $A^*B^*C^*$ ; where a uniform scaling occurs. Transforming triangle  $ABC$  by

$$\begin{bmatrix} 1/2 & 0 \\ 0 & 3 \end{bmatrix}$$

to  $D^*E^*F^*$  shows distortion due to the nonuniform scale factors.

In general, if

$$[T] = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad (2-37)$$

with  $a = d$ ,  $b = c = 0$ , a uniform scaling occurs; and if  $a \neq d$ ,  $b = c = 0$ , a nonuniform scaling occurs. For a uniform scaling, if  $a = d > 1$ , a uniform expansion occurs; i.e., the figure gets larger. If  $a = d < 1$ , then a uniform compression occurs; i.e., the figure gets smaller. Nonuniform expansions and compressions occur, depending on whether  $a$  and  $d$  are individually  $> 1$  or  $< 1$ .

Figure 2-8 also reveals what at first glance is an apparent translation of the transformed triangles. This apparent translation is easily understood if we recall that the position vectors, not the points, are scaled with respect to the origin.

To see this more clearly examine the transformation of  $ABC$  to  $D^*E^*F^*$  more closely. Specifically,

$$[X^*] = [X][T] = \begin{bmatrix} 4 & 2 \\ 4 & 4 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} 1/2 & 0 \\ 0 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 2 & 12 \\ 1 & 12 \end{bmatrix}$$

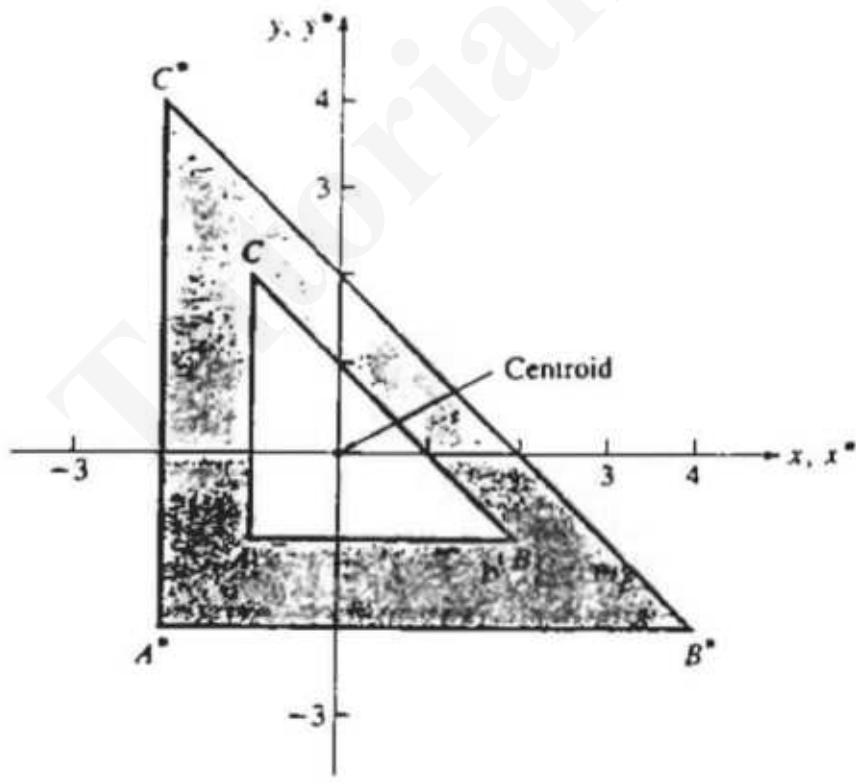
Note that each of the  $x$  components of the *position vectors* of  $DEF$  is increased by a scale factor of 3 and the  $y$  components of the *position vectors* by a scale factor of 2.

To obtain a pure scaling without apparent translation, the centroid of the figure must be at the origin. This effect is shown in Fig. 2-9, where the triangle  $ABC$  with the centroid coordinates ( $1/3$  the base and  $1/3$  the height) at the origin is scaled by a factor of 2. Specifically,

$$[X^*] = [X][T] = \begin{bmatrix} -1 & -1 \\ 2 & -1 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} -2 & -2 \\ 4 & -2 \\ -2 & 4 \end{bmatrix}$$

## 2-12 COMBINED TRANSFORMATIONS

The power of the matrix methods described in the previous sections is clear. By performing matrix operations on the position vectors which define the vertices, the shape and position of the surface can be controlled. However, a desired orientation may require more than one transformation. Since matrix multiplication is noncommutative, the order of application of the transformations is important.



**Figure 2-9** Uniform scaling without apparent translation.

In order to illustrate the effect of noncommutative matrix multiplication, consider the operations of rotation and reflection on the position vector  $[x \ y]$ . If a  $90^\circ$  rotation,  $[T_1]$ , is followed by reflection through the line  $y = -x$ ,  $[T_2]$ , these two consecutive transformations give

$$[X'] = [X][T_1] = [x \ y] \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} = [-y \ x]$$

and then

$$[X^*] = [X'][T_2] = [-y \ x] \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} = [-x \ y]$$

On the other hand, if reflection is followed by rotation, the results given by

$$[X'] = [X][T_2] = [x \ y] \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} = [-y \ -x]$$

and  $[X^*] = [X'][T_1] = [-y \ -x] \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} = [x \ -y]$

are obtained. The results are different, confirming that the order of application of matrix transformations is important.

Another important point is illustrated by the above results and by the example given below. Above, the individual transformation matrices were successively applied to the successively obtained position vectors, e.g.,

$$[x \ y][T_1] \rightarrow [x' \ y']$$

and  $[x' \ y'][T_2] \rightarrow [x^* \ y^*]$

In the example below the individual transformations are first combined or concatenated and then the *concatenated* transformation is applied to the original position vector, e.g.,  $[T_1][T_2] \rightarrow [T_3]$  and  $[x \ y][T_3] \rightarrow [x^* \ y^*]$ .

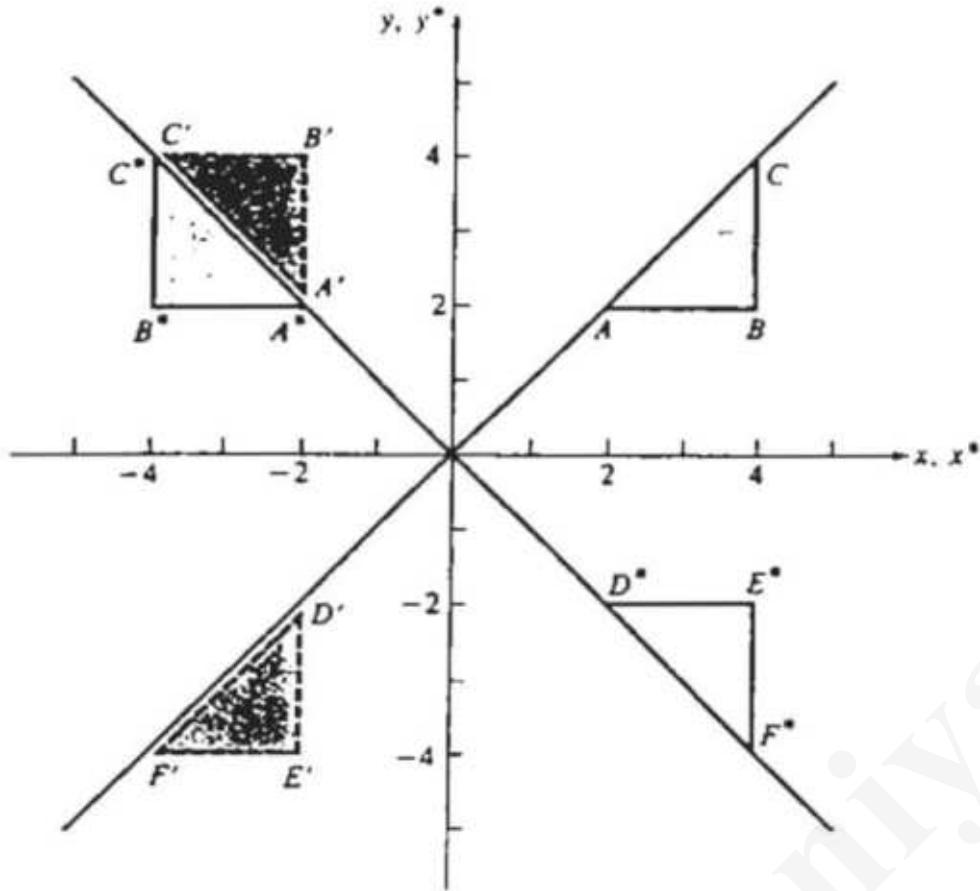
#### Example 2-4 Combined Two-Dimensional Transformations

Consider the triangle  $ABC$  shown in Fig. 2-10. The two transformations are a  $+90^\circ$  rotation about the origin:

$$[T_1] = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

and a reflection through the line  $y = -x$

$$[T_2] = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$$



**Figure 2-10** Combined two-dimensional transformations.

The effect of the combined transformation  $[T_3] = [T_1][T_2]$  on the triangle  $ABC$  is

$$[X^*] = [X][T_1][T_2] = [X][T_3]$$

or

$$\begin{bmatrix} 2 & 2 \\ 4 & 2 \\ 4 & 4 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 4 & 2 \\ 4 & 4 \end{bmatrix} \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} -2 & 2 \\ -4 & 2 \\ -4 & 4 \end{bmatrix}$$

The final result is shown as  $A''B''C''$  and the intermediate result as  $A'B'C'$  in Fig. 2-10.

Reversing the order of application of the transformations yields

$$[X^*] = [X][T_2][T_1] = [X][T_4]$$

or

$$\begin{bmatrix} 2 & 2 \\ 4 & 2 \\ 4 & 4 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 4 & 2 \\ 4 & 4 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 2 & -2 \\ 4 & -2 \\ 4 & -4 \end{bmatrix}$$

The final result is shown as  $D'E'F'$  and the intermediate result as  $D''E''F''$  in Fig. 2-10.

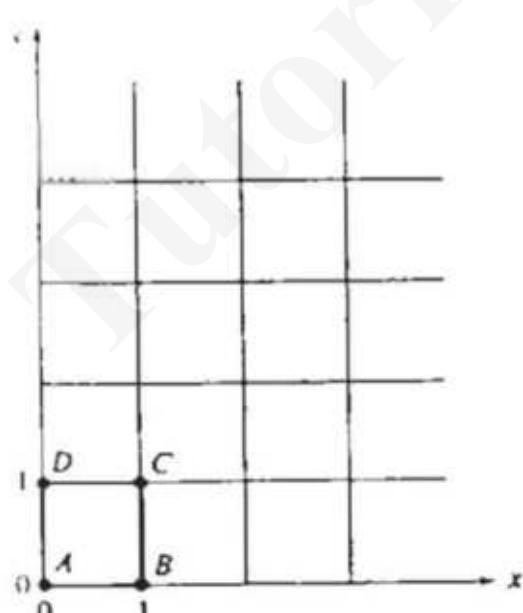
The results are different, again confirming that the order of application of the transformations is important. Note also that  $\det[T_3] = -1$  and  $\det[T_4] = -1$ , indicating that both results can be obtained by a single reflection.  $A^*B^*C^*$  can be obtained from  $ABC$  by reflection through the  $y$ -axis (see  $[T_3]$  and Eq. 2-34).  $D^*E^*F^*$  can be obtained from  $ABC$  by reflection through the  $x$ -axis (see  $[T_4]$  and Eq. 2-33).

## 2.13 TRANSFORMATION OF THE UNIT SQUARE

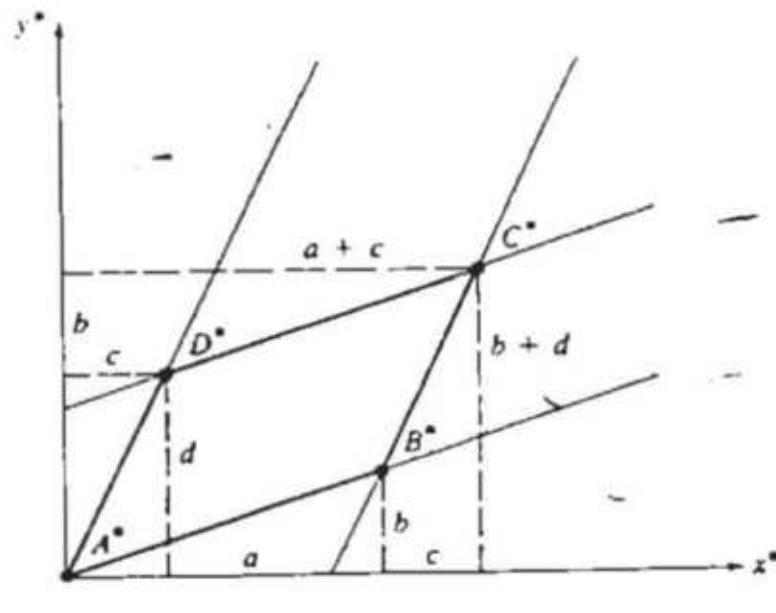
So far we have concentrated on the behavior of points and lines to determine the effect of simple matrix transformations. However, the matrix is correctly considered to operate on *every* point in the plane. As has been shown, the only point that remains invariant under a  $2 \times 2$  matrix transformation is the origin. All other points within the plane are transformed. This transformation may be interpreted as a stretching of the original plane and coordinate system into a new shape. More formally, we say that the transformation causes a mapping from one coordinate space into a second.

Consider a square-grid network consisting of unit squares in the  $xy$  plane as shown in Fig. 2-11. The four position vectors of a unit square with one corner at the origin of the coordinate system are

- $\begin{bmatrix} 0 & 0 \end{bmatrix}$  origin of the coordinates —  $A$
- $\begin{bmatrix} 1 & 0 \end{bmatrix}$  unit point on the  $x$ -axis —  $B$
- $\begin{bmatrix} 1 & 1 \end{bmatrix}$  outer corner —  $C$
- $\begin{bmatrix} 0 & 1 \end{bmatrix}$  unit point on the  $y$ -axis —  $D$



(a)



(b)

Figure 2-11 General transformation of unit square. (a) Before transformation; (b) after transformation.

This unit square is shown in Fig. 2-11a. Application of a general  $2 \times 2$  matrix transformation to the unit square yields

$$\begin{array}{l} A \\ B \\ C \\ D \end{array} \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ a & b \\ a+c & b+d \\ c & d \end{bmatrix} \begin{array}{l} A^* \\ B^* \\ C^* \\ D^* \end{array} \quad (2-38)$$

The results of this transformation are shown in Fig. 2-11b. First notice from Eq. (2-38) that the origin is not affected by the transformation, i.e.,  $[A] = [A^*] = [0 \ 0]$ . Further, notice that the coordinates of  $B^*$  are equal to the first row in the general transformation matrix, and the coordinates of  $D^*$  are equal to the second row in the general transformation matrix. Thus, once the coordinates of  $B^*$  and  $D^*$  (the transformed unit vectors  $[1 \ 0]$  and  $[0 \ 1]$ , respectively) are known, the general transformation matrix is determined. Since the sides of the unit square are originally parallel, and since we have previously shown that parallel lines transform into parallel lines, the transformed figure is a parallelogram.

The effect of the terms  $a, b, c$  and  $d$  in the  $2 \times 2$  matrix can be identified separately. The terms  $b$  and  $c$  cause a shearing (see Sec. 2-4) of the initial square in the  $y$  and  $x$  directions, respectively, as can be seen in Fig. 2-11b. The terms  $a$  and  $d$  act as scale factors, as noted earlier. Thus, the general  $2 \times 2$  matrix produces a combination of shearing and scaling.

It is also possible to easily determine the area of  $A^*B^*C^*D^*$ ; the parallelogram shown in Fig. 2-11b. The area within the parallelogram can be calculated as follows:

$$A_p = (a+c)(b+d) - \frac{1}{2}(ab) - \frac{1}{2}(cd) - \frac{c}{2}(b+b+d) - \frac{b}{2}(c+a+c)$$

which yields

$$A_p = ad - bc = \det \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad (2-39)$$

It can be shown that the area of any parallelogram  $A_p$ , formed by transforming a square, is a function of the transformation matrix determinant and is related to the area of the initial square  $A_s$  by the simple relationship

$$A_p = A_s(ad - bc) = A_s \det [T] \quad (2-40)$$

In fact, since the area of a general figure is the sum of unit squares, the area of any transformed figure  $A_t$  is related to the area of the initial figure  $A_i$  by

$$A_t = A_i(ad - bc) \quad (2-41)$$

This is a useful technique for determining the areas of arbitrary shapes.

**Example 2-5 Area Scaling**

The triangle  $ABC$  with position vectors  $[1 \ 0]$ ,  $[0 \ 1]$  and  $[-1 \ 0]$ , is transformed by

$$[T] = \begin{bmatrix} 3 & 2 \\ -1 & 2 \end{bmatrix}$$

to create a second triangle  $A^*B^*C^*$  as shown in Fig. 2-12.

The area of the triangle  $ABC$  is

$$A_i = \frac{1}{2}(\text{base})(\text{height}) = \frac{1}{2}(2)(1) = 1$$

Using Eq. (2-41) the area of the transformed triangle  $A^*B^*C^*$  is

$$A_t = A_i(ad - bc) = 1(6 + 2) = 8$$

Now the vertices of the transformed triangle  $A^*B^*C^*$  are

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

Calculating the area from the transformed vertices yields

$$A_t = \frac{1}{2}(\text{base})(\text{height}) = \frac{1}{2}(4)(4) = 8$$

which confirms the previous result.

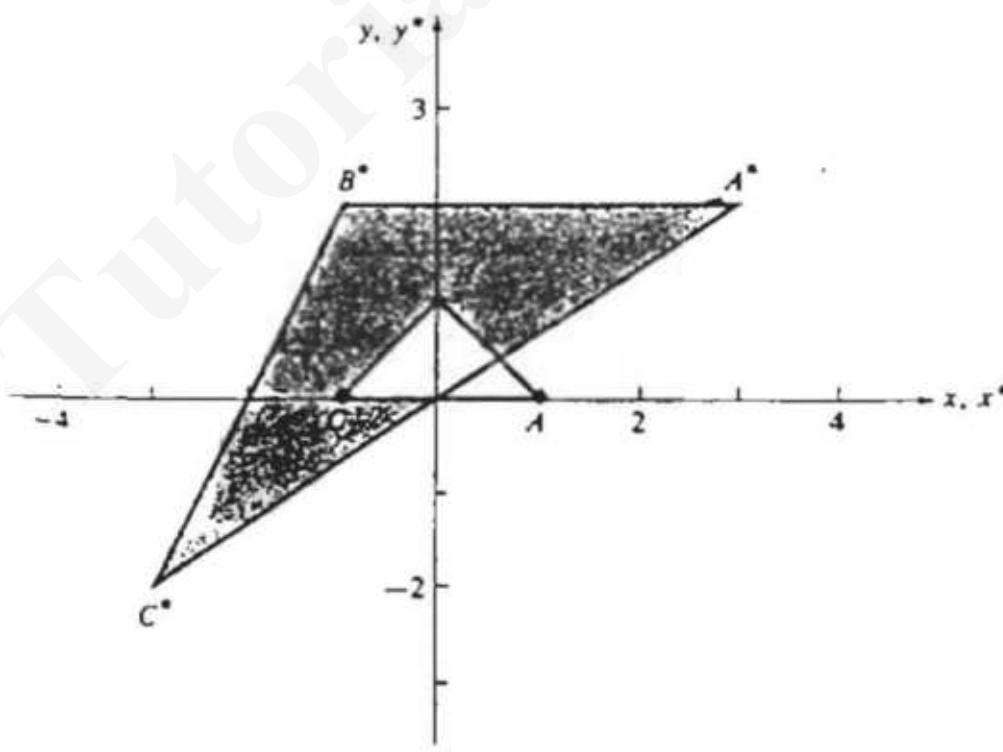


Figure 2-12 Area scaling.

## 2-14 SOLID BODY TRANSFORMATIONS

We now return to the question posed in Sec. 2-8, i.e., when do perpendicular lines transform as perpendicular lines? First consider the somewhat more general question of when is the angle between intersecting lines preserved?

Recall that the dot or scalar product of two vectors is

$$\bar{V}_1 \cdot \bar{V}_2 = V_{1x}V_{2x} + V_{1y}V_{2y} = |\bar{V}_1| |\bar{V}_2| \cos \theta \quad (2-42)$$

and the cross product of two vectors confined to the two-dimensional  $xy$  plane is

$$\bar{V}_1 \times \bar{V}_2 = (V_{1x}V_{2y} - V_{2x}V_{1y})\bar{k} = |\bar{V}_1| |\bar{V}_2| \bar{k} \sin \theta \quad (2-43)$$

where the subscripts  $x, y$  refer to the  $x$  and  $y$  components of the vector,  $\theta$  is the acute angle between the vectors and  $\bar{k}$  is the unit vector perpendicular to the  $xy$  plane.

Transforming  $\bar{V}_1$  and  $\bar{V}_2$  using a general  $2 \times 2$  transformation yields

$$\begin{bmatrix} \bar{V}_1 \\ \bar{V}_2 \end{bmatrix} [T] = \begin{bmatrix} V_{1x} & V_{1y} \\ V_{2x} & V_{2y} \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} aV_{1x} + cV_{1y} & bV_{1x} + dV_{1y} \\ aV_{2x} + cV_{2y} & bV_{2x} + dV_{2y} \end{bmatrix} = \begin{bmatrix} \bar{V}_1^* \\ \bar{V}_2^* \end{bmatrix} \quad (2-44)$$

The cross product of  $\bar{V}_1^*$  and  $\bar{V}_2^*$  is

$$\bar{V}_1^* \times \bar{V}_2^* = (ad - cb)(V_{1x}V_{2y} - V_{2x}V_{1y})\bar{k} = |\bar{V}_1^*| |\bar{V}_2^*| \bar{k} \sin \theta \quad (2-45)$$

Similarly the scalar product is

$$\begin{aligned} \bar{V}_1^* \cdot \bar{V}_2^* &= (a^2 + b^2)V_{1x}V_{2x} + (c^2 + d^2)V_{1y}V_{2y} + (ac + bd)(V_{1x}V_{2y} + V_{1y}V_{2x}) \\ &= |\bar{V}_1^*| |\bar{V}_2^*| \cos \theta \end{aligned} \quad (2-46)$$

Requiring that the magnitude of the vectors, as well as the angle between them, remains unchanged, comparing Eqs. (2-42) and (2-46) and Eqs. (2-43) and (2-45) and equating coefficients of like terms yields

$$a^2 + b^2 = 1 \quad (2-47a)$$

$$c^2 + d^2 = 1 \quad (2-47b)$$

$$ac + bd = 0 \quad (2-47c)$$

$$ad - bc = +1 \quad (2-48)$$

Equations (2-47a,b,c) correspond to the conditions that a matrix be orthogonal, i.e.,

$$[T][T]^{-1} = [T][T]^T = [I]$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} a & c \\ b & d \end{bmatrix} = \begin{bmatrix} a^2 + b^2 & ac + bd \\ ac + bd & c^2 + d^2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Equation (2-48) requires that the determinant of the transformation matrix be +1.

Thus, the angles between intersecting lines are preserved by pure rotation. Since reflective transformations are also orthogonal with a determinant of -1, these results are easily extended. In this case the magnitude of the vectors is preserved, but the angle between the transformed vectors is technically  $2\pi - \theta$ . Hence, the angle is technically not preserved. Still, perpendicular lines transform as perpendicular lines. Since  $\sin(2\pi - \theta) = -\sin\theta$ ,  $ad - bc = -1$ . Pure rotations and reflections are called rigid body transformations. In addition, a few minutes' thought or experimentation reveals that uniform scalings also preserve the angle between intersecting lines but not the magnitudes of the transformed vectors.<sup>t</sup>

## 2.15 TRANSLATIONS AND HOMOGENEOUS COORDINATES

A number of transformations governed by the general  $2 \times 2$  transformation matrix, e.g., rotation, reflection, scaling, shearing etc., were discussed in the previous sections. As noted previously, the origin of the coordinate system is invariant with respect to all of these transformations. However, it is necessary to be able to modify the position of the origin, i.e., to transform every point in the two-dimensional plane. This can be accomplished by translating the origin or any other point in the two-dimensional plane, i.e.,

$$\begin{aligned} x^* &= ax + cy + m \\ y^* &= bx + dy + n \end{aligned}$$

Unfortunately, it is not possible to introduce the constants of translation  $m, n$  into the general  $2 \times 2$  transformation matrix; there is no room!

This difficulty can be overcome by introducing homogeneous coordinates. The homogeneous coordinates of a nonhomogeneous position vector  $[x \ y]$  are  $[x' \ y' \ h]$  where  $x = x'/h$  and  $y = y'/h$  and  $h$  is any real number. Note that  $h = 0$  has special meaning. One set of homogeneous coordinates is always of the form  $[x \ y \ 1]$ . We choose this form to represent the position vector  $[x \ y]$  in the physical  $xy$  plane. All other homogeneous coordinates are of the form  $[hx \ hy \ h]$ . There is no unique homogeneous coordinate representation, e.g.,  $[6 \ 4 \ 2], [12 \ 8 \ 4], [3 \ 2 \ 1]$  all represent the physical point  $(3, 2)$ .

The general transformation matrix is now  $3 \times 3$ . Specifically,

$$[T] = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ m & n & 1 \end{bmatrix} \quad (2-49)$$

<sup>t</sup>Since an orthogonal matrix preserves both the angle between the vectors and their magnitudes, the uniform scaling transformation matrix is not orthogonal.

where the elements  $a, b, c, d$  of the upper left  $2 \times 2$  submatrix have exactly the same effects revealed by our previous discussions.  $m, n$  are the translation factors in the  $x$  and  $y$  directions, respectively. The pure two-dimensional translation matrix is

$$\begin{bmatrix} x^* & y^* & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ m & n & 1 \end{bmatrix} = \begin{bmatrix} x + m & y + n & 1 \end{bmatrix} \quad (2 - 50)$$

Notice that now every point in the two-dimensional plane, even the origin ( $x = y = 0$ ), can be transformed.

## 2-16 ROTATION ABOUT AN ARBITRARY POINT

Previously we have considered rotations as occurring about the origin. Homogeneous coordinates provide a mechanism for accomplishing rotations about points other than the origin. In general, a rotation about an arbitrary point can be accomplished by first translating the point to the origin, performing the required rotation, and then translating the result back to the original center of rotation. Thus, rotation of the position vector  $[x \ y \ 1]$  about the point  $m, n$  through an arbitrary angle can be accomplished by

$$\begin{bmatrix} x^* & y^* & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -m & -n & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ m & n & 1 \end{bmatrix} \quad (2 - 51)$$

By carrying out the two interior matrix products we can write

$$\begin{bmatrix} x^* & y^* & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ \left\{ -m(\cos \theta - 1) \right\} + n \sin \theta & \left\{ -n(\cos \theta - 1) \right\} - m \sin \theta & 1 \end{bmatrix} \quad (2 - 52)$$

An example illustrates this result.

### Example 2-6 Rotation About an Arbitrary Point.

Suppose the center of an object is at  $[4 \ 3]$  and it is desired to rotate the object  $90^\circ$  counterclockwise about its center. Using the matrix

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

causes a rotation about the origin, not the object center. The necessary procedure is to first translate the object so that the desired center of rotation is at the origin by using the translation matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -4 & -3 & 1 \end{bmatrix}$$

Next apply the rotation matrix, and finally translate the results of the rotation back to the original center by means of the inverse translation matrix. The entire operation

$$[x^* \ y^* \ 1] = [x \ y \ 1] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -4 & -3 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 4 & 3 & 1 \end{bmatrix}$$

can be combined into one matrix operation by concatenating the transformation matrices, i.e.,

$$[x^* \ y^* \ 1] = [x \ y \ 1] \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 7 & -1 & 1 \end{bmatrix}$$


---

## 2.17 REFLECTION THROUGH AN ARBITRARY LINE

Previously (see Sec. 2-10) reflection through lines that passed through the origin was discussed. Occasionally reflection of an object through a line that does not pass through the origin is required. This can be accomplished using a procedure similar to that for rotation about an arbitrary point. Specifically,

Translate the line and the object so that the line passes through the origin.  
Rotate the line and the object about the origin until the line is coincident with one of the coordinate axes.

Reflect through the coordinate axis. —

Apply the inverse rotation about the origin. —

Translate back to the original location.

In matrix notation the resulting concatenated matrix is

$$[T] = [T'][R][R'][R]^{-1}[T']^{-1} \quad (2-53)$$

where

$T'$  is the translation matrix

$R$  is the rotation matrix about the origin

$R'$  is the reflection matrix

The translations, rotations and reflections are also applied to the figure to be transformed. An example is given below.

**Example 2-7 Reflection Through an Arbitrary Line**

Consider the line  $L$  and the triangle  $ABC$  shown in Fig. 2-13a. The equation of the line  $L$  is

$$y = \frac{1}{2}(x + 4)$$

The position vectors  $[2 \ 4 \ 1]$ ,  $[4 \ 6 \ 1]$  and  $[2 \ 6 \ 1]$  describe the vertices of the triangle  $ABC$ .

The line  $L$  will pass through the origin by translating it  $-2$  units in the  $y$  direction. The resulting line can be made coincident with the  $x$ -axis by rotating it by  $-\tan^{-1}(\frac{1}{2}) = -26.57^\circ$  about the origin. Equation (2-33) is then used to reflect the triangle through the  $x$ -axis. The transformed position vectors of the triangle are then rotated and translated back to the original orientation. The combined transformation is

$$[T] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{bmatrix} \begin{bmatrix} 2/\sqrt{5} & -1/\sqrt{5} & 0 \\ 1/\sqrt{5} & 2/\sqrt{5} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \\ \begin{bmatrix} 2/\sqrt{5} & 1/\sqrt{5} & 0 \\ -1/\sqrt{5} & 2/\sqrt{5} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix}$$

$$[T] = \begin{bmatrix} 3/5 & 4/5 & 0 \\ 4/5 & -3/5 & 0 \\ -8/5 & 16/5 & 1 \end{bmatrix}$$

and the transformed position vectors for the triangle  $A^*B^*C^*$  are

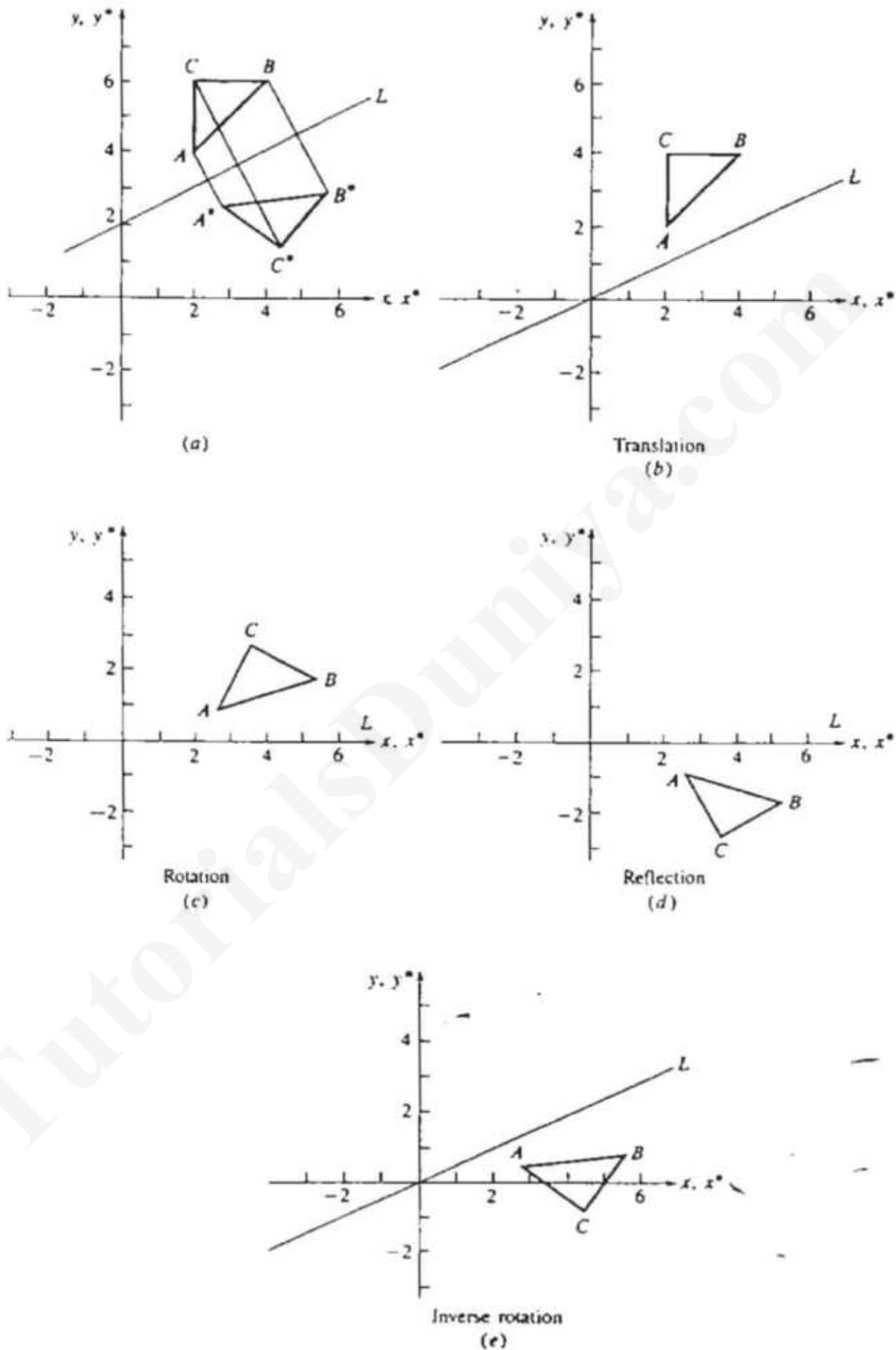
$$\begin{bmatrix} 2 & 4 & 1 \\ 4 & 6 & 1 \\ 2 & 6 & 1 \end{bmatrix} \begin{bmatrix} 3/5 & 4/5 & 0 \\ 4/5 & -3/5 & 0 \\ -8/5 & 16/5 & 1 \end{bmatrix} = \begin{bmatrix} 14/5 & 12/5 & 1 \\ 28/5 & 14/5 & 1 \\ 22/5 & 6/5 & 1 \end{bmatrix}$$

as shown in Fig. 2-13a. Figures 2-13b through 2-13c show the various steps in the transformation.

## 2-18 PROJECTION - A GEOMETRIC INTERPRETATION OF HOMOGENEOUS COORDINATES

The general  $3 \times 3$  transformation matrix for two-dimensional homogeneous coordinates can be subdivided into four parts:

$$[T] = \begin{bmatrix} a & b & : & p \\ c & d & : & q \\ \dots & \dots & \dots & \dots \\ m & n & : & s \end{bmatrix} \quad (2-54)$$



**Figure 2-13** Reflection through an arbitrary line. (a) Original and final position; (b) translate line through origin; (c) rotate line to  $x$ -axis; (d) reflect about  $x$ -axis; (e) undo rotation; (a) undo translation.

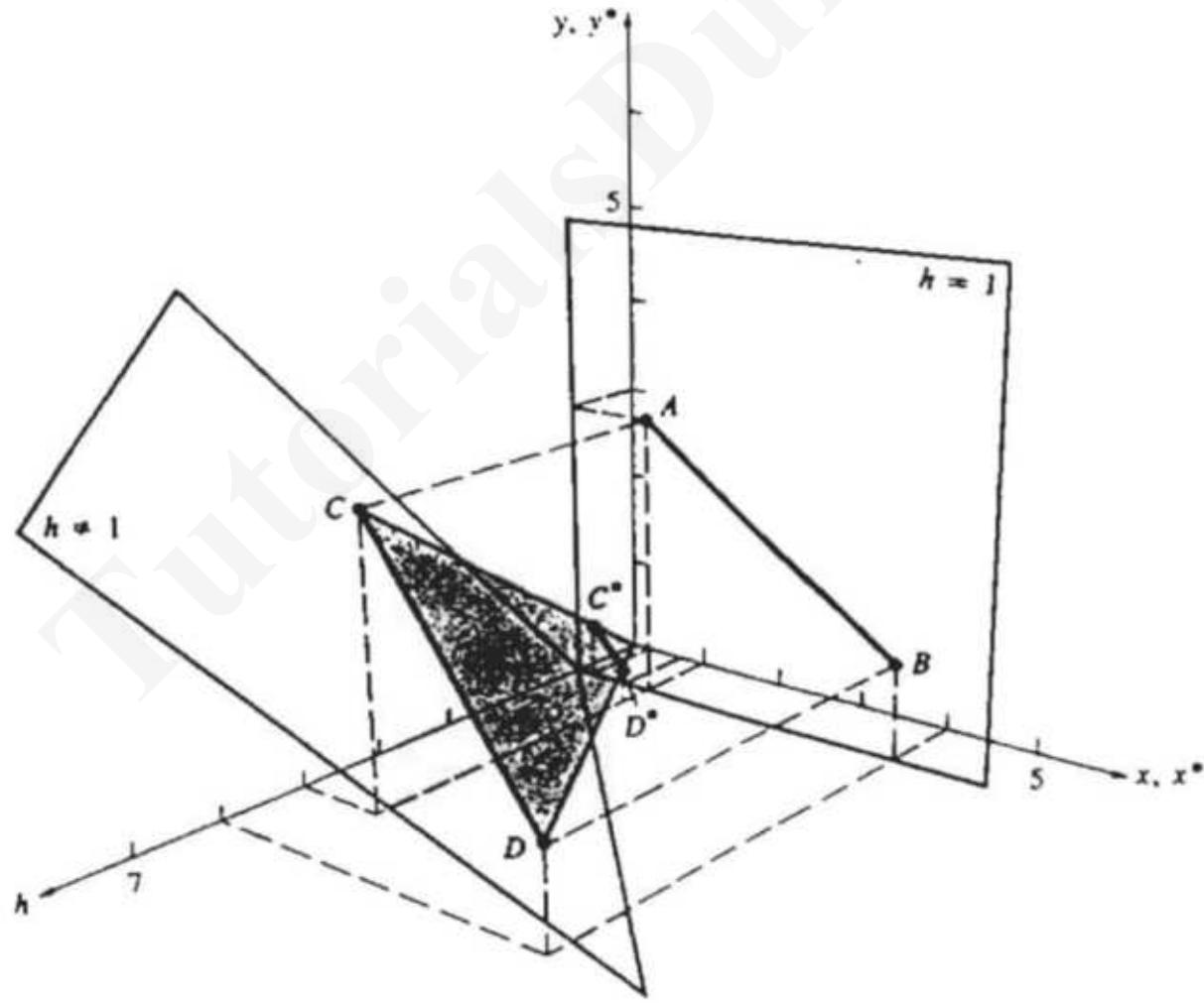
Recall that  $a, b, c$  and  $d$  produce scaling, rotation, reflection and shearing; and  $m$  and  $n$  produce translation. In the previous two sections  $p = q = 0$  and  $s = 1$ . Suppose  $p$  and  $q$  are not zero. What are the effects? A geometric interpretation is useful.

When  $p = q = 0$  and  $s = 1$ , the homogeneous coordinate of the transformed position vectors is always  $h = 1$ . Geometrically this result is interpreted as confining the transformation to the  $h = 1$  physical plane.

To show the effect of  $p \neq 0, q \neq 0$  in the third column in the general  $3 \times 3$  transformation matrix, consider the following:

$$\begin{bmatrix} X & Y & h \end{bmatrix} = \begin{bmatrix} hx & hy & h \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & p \\ 0 & 1 & q \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x & y & (px + qy + 1) \end{bmatrix} \quad (2 - 55)$$

Here  $X = hx$ ,  $Y = hy$  and  $h = px + qy + 1$ . The transformed position vector expressed in homogeneous coordinates now lies in a plane in three-dimensional space defined by  $h = px + qy + 1$ . This transformation is shown in Fig. 2-14, where the line  $AB$  in the physical ( $h = 1$ ) plane is transformed to the line  $CD$  in the  $h \neq 1$  plane, i.e.,  $pX + qY - h + 1 = 0$ .



**Figure 2-14** Transformation from the physical ( $h = 1$ ) plane into the  $h \neq 1$  plane and projection from the  $h \neq 1$  plane back into the physical plane.

However, the results of interest are those in the physical plane corresponding to  $h = 1$ . These results can be obtained by geometrically projecting  $CD$  from the  $h \neq 1$  plane back onto the  $h = 1$  plane using a pencil of rays through the origin. From Fig. 2-14, using similar triangles,

$$x^* = \frac{X}{h} \quad y^* = \frac{Y}{h}$$

or in homogeneous coordinates

$$[x^* \quad y^* \quad 1] = \left[ \frac{X}{h} \quad \frac{Y}{h} \quad 1 \right]$$

Now, normalizing Eq. (2-55) by dividing through by the homogeneous coordinate value  $h$  yields

$$[x^* \quad y^* \quad 1] = \left[ \frac{X}{h} \quad \frac{Y}{h} \quad 1 \right] = \left[ \frac{x}{px + qy + 1} \quad \frac{y}{px + qy + 1} \quad 1 \right] \quad (2-56)$$

or

$$x^* = \frac{X}{h} = \frac{x}{px + qy + 1} \quad (2-57a)$$

$$y^* = \frac{Y}{h} = \frac{y}{px + qy + 1} \quad (2-57b)$$

The details are given in the example below.

### Example 2-8 Projection in Homogeneous Coordinates

For the line  $AB$  in Fig. 2-14 we have, with  $p = q = 1$ ,  $[A] = [1 \quad 3 \quad 1]$  and  $[B] = [4 \quad 1 \quad 1]$ ,

$$\begin{bmatrix} C \\ D \end{bmatrix} = \begin{bmatrix} A \\ B \end{bmatrix} [T] = \begin{bmatrix} 1 & 3 & 1 \\ 4 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 4 & 1 & 6 \end{bmatrix}$$

Thus,  $[C] = [1 \quad 3 \quad 5]$  and  $[D] = [4 \quad 1 \quad 6]$  in the plane  $h = x + y + 1$ . Projecting back onto the  $h = 1$  physical plane by dividing through by the homogeneous coordinate factor yields the two-dimensional transformed points

$$[C^*] = [1 \quad 3 \quad 5] = [1/5 \quad 3/5 \quad 1]$$

$$[D^*] = [4 \quad 1 \quad 6] = [2/3 \quad 1/6 \quad 1]$$

The result is shown in Fig. 2-14.

## 2-19 OVERALL SCALING

The remaining unexplained element in the general  $3 \times 3$  transformation matrix (see Eq. 2-54),  $s$ , produces overall scaling; i.e., all components of the position vector are equally scaled. To show this, consider the transformation

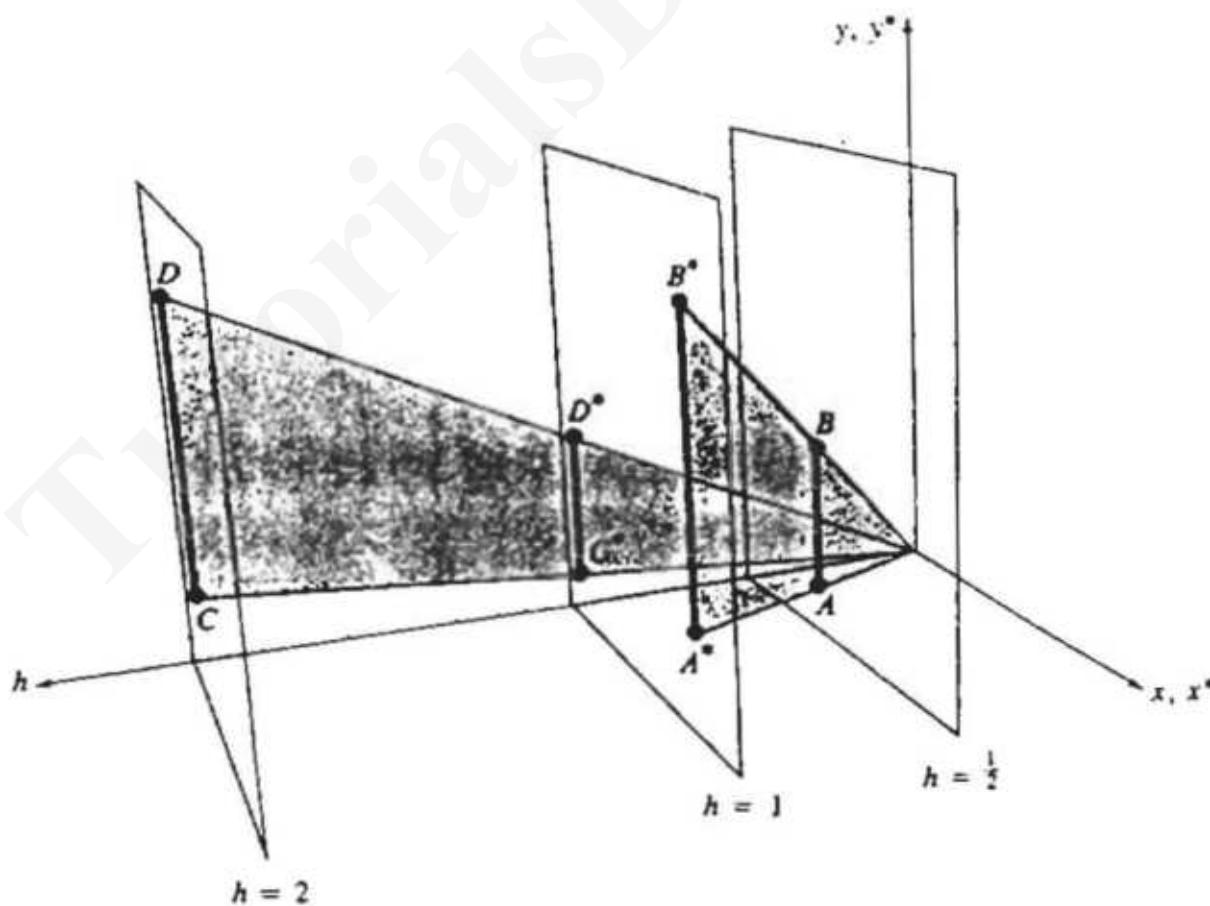
$$[X \ Y \ h] = [x \ y \ 1] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & s \end{bmatrix} = [x \ y \ s] \quad (2-58)$$

Here,  $X = x$ ,  $Y = y$  and  $h = s$ . After normalizing, this yields

$$X^* = \frac{x}{s} \quad \text{and} \quad Y^* = \frac{y}{s}$$

Thus, the transformation is  $[x \ y \ 1][T] = [\frac{x}{s} \ \frac{y}{s} \ 1]$ , a uniform scaling of the position vector. If  $s < 1$ , then an expansion occurs; and if  $s > 1$ , a compression occurs.

Note that this is also a transformation out of the  $h = 1$  plane. Here,  $h = s = \text{constant}$ . Hence, the  $h \neq 1$  plane is parallel to the  $h = 1$  plane. A geometric interpretation of this effect is shown in Fig. 2-15. If  $s < 1$ , then the  $h = \text{constant}$  plane lies between the  $h = 1$  and  $h = 0$  planes. Consequently, when the transformed line  $AB$  is projected back onto the  $h = 1$  plane to  $A^*B^*$ , it becomes larger. Similarly, if  $s > 1$ , then the  $h = \text{constant}$  plane lies beyond the  $h = 1$  plane along the  $h$ -axis. When the transformed line  $CD$  is projected back onto the  $h = 1$  plane to  $C^*D^*$ , it becomes smaller.



**Figure 2-15** A geometric interpretation of overall scaling.

## 2 20 POINTS AT INFINITY

Homogeneous coordinates provide a convenient and efficient technique for mapping a set of points from one coordinate system into a corresponding set in an alternate coordinate system. Frequently, an infinite range in one coordinate system is mapped into a finite range in an alternate coordinate system. Unless the mappings are carefully chosen, parallel lines may not map into parallel lines. However, intersection points map into intersection points. This property is used to determine the homogeneous coordinate representation of a point at infinity.

We begin by considering the pair of intersecting lines given by

$$\begin{aligned}x + y &= 1 \\2x - 3y &= 0\end{aligned}$$

which have an intersection point at  $x = 3/5$ ,  $y = 2/5$ . Writing the equations as  $x + y - 1 = 0$  and  $2x - 3y = 0$  and casting them in matrix form yields

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 1 & -3 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \end{bmatrix}$$

or

$$[X][M'] = [R]$$

If  $[M']$  were square, the intersection could be obtained by matrix inversion. This can be accomplished by slightly rewriting the system of original equations. Specifically,

$$\begin{aligned}x + y - 1 &= 0 \\2x - 3y &= 0 \\1 &= 1\end{aligned}$$

In matrix form this is

$$[X][M] = [R]$$

i.e.,

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 0 \\ 1 & -3 & 0 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

The inverse of this square matrix is †

$$[M]^{-1} = \begin{bmatrix} 3/5 & 2/5 & 0 \\ 1/5 & -1/5 & 0 \\ 3/5 & 2/5 & 1 \end{bmatrix} = \frac{1}{5} \begin{bmatrix} 3 & 2 & 0 \\ 1 & -1 & 0 \\ 3 & 2 & 5 \end{bmatrix}$$

† Matrix inversion techniques are discussed in Ref. 2-1 or any good linear algebra book.

Multiplying both sides of the equation by  $[M]^{-1}$  and noting that  $[M][M]^{-1} = [I]$ , the identity matrix, yields

$$[x \ y \ 1] = \frac{1}{5} [0 \ 0 \ 1] \begin{bmatrix} 3 & 2 & 0 \\ 1 & -1 & 0 \\ 3 & 2 & 5 \end{bmatrix} = [3/5 \ 2/5 \ 1]$$

Thus, the intersection point is again  $x = 3/5$  and  $y = 2/5$ .

Now consider two parallel lines defined by

$$\begin{aligned} x + y &= 1 \\ x + y &= 0 \end{aligned}$$

By definition, in Euclidean (common) geometric space, the intersection point of this pair of parallel lines occurs at infinity. Proceeding, as above, to calculate the intersection point of these lines leads to the matrix formulation

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

However, even though the matrix is square it does not have an inverse, since two rows are identical. The matrix is said to be singular. Another alternate formulation is possible which does have an invertible matrix. This is obtained by rewriting the system of equations as

$$\begin{aligned} x + y - 1 &= 0 \\ x + y &= 0 \\ x &= x \end{aligned}$$

In matrix form this is

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

Here, the matrix is not singular; the inverse exists and is

$$[M]^{-1} = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 1 \\ 1 & -1 & 0 \end{bmatrix}$$

Multiplying both sides of the equation by the inverse yields

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

The resulting homogeneous coordinates  $[1 \ -1 \ 0]$  represent the ‘point of intersection’ for the two parallel lines, i.e., a point at infinity. Specifically it represents the point at infinity in the direction  $[1 \ -1]$  in the two-dimensional plane. In general, the two-dimensional homogeneous vector  $[a \ b \ 0]$  represents the point at infinity on the line  $ay - bx = 0$ . Some examples are:

- $[1 \ 0 \ 0]$  on the positive  $x$ -axis
- $[-1 \ 0 \ 0]$  on the negative  $x$ -axis
- $[0 \ 1 \ 0]$  on the positive  $y$ -axis
- $[0 \ -1 \ 0]$  on the negative  $y$ -axis
- $[1 \ 1 \ 0]$  along the line  $y = x$  in the direction  $[1 \ 1]$

The fact that a vector with the homogeneous component  $h = 0$  does indeed represent a point at infinity can also be illustrated by the limiting process shown in Table 2-1. Consider the line  $y^* = (3/4)x^*$  and the point  $[X \ Y \ h] = [4 \ 3 \ 1]$ . Recalling that a unique representation of a position vector does not exist in homogeneous coordinates, the point  $[4 \ 3 \ 1]$  is represented in homogeneous coordinates in all the ways shown in Table 2-1. Note that in Table 2-1 as  $h \rightarrow 0$ , the ratio of  $y^*/x^*$  remains at  $3/4$ , as is required by the governing equation. Further, note that successive pairs of  $(x^*, y^*)$  all of which fall on the line  $y^* = (3/4)x^*$ , become closer to infinity. Thus, in the limit as  $h \rightarrow 0$ , the point at infinity is given by  $[X \ Y \ h] = [4 \ 3 \ 0]$  in homogeneous coordinates.

By recalling Fig. 2-15, a geometrical interpretation of the limiting process as  $h \rightarrow 0$  is also easily illustrated. Consider a line of unit length from  $x = 0, y = 0$  in the direction  $[1 \ 0]$ , in the plane  $h = s$  ( $s < 1$ ). As  $s \rightarrow 0$  the projection of this line back onto the  $h = 1$  physical plane by a pencil of rays through the origin becomes of infinite length. Consequently, the end point of the line must represent the point at infinity on the  $x$ -axis.

Table 2-1 Homogeneous Coordinates for the Point  $[4 \ 3]$

$h$	$x^*$	$y^*$	$X$	$Y$
1	4	3	4	3
$1/2$	8	6	4	3
$1/3$	12	9	4	3
.	.	.	.	.
$1/10$	40	30	4	3
.	.	.	.	.
$1/100$	400	300	4	3
.	.	.	.	.

## 2-21 TRANSFORMATION CONVENTIONS

Various conventions are used to represent data and to perform transformations with matrix multiplication. Extreme care is necessary in defining the problem and interpreting the results. For example, before performing a rotation the following decisions must be made:

Are the position vectors (vertices) to be rotated defined relative to a right-hand coordinate or a left-hand coordinate system?

Is the object or the coordinate system being rotated?

How are positive and negative rotations defined?

Are the position vectors stored as a row matrix or as a column matrix?

About what line, or axis, is rotation to occur?

In this text a right-hand coordinate system is used, the object is rotated in a fixed coordinate system, positive rotation is defined using the right-hand rule, i.e., clockwise about an axis as seen by an observer at the origin looking outward along the positive axis, and position vectors are represented as row matrices.

Equation (2-29) gives the transformation for positive rotation about the origin or about the  $z$ -axis. Since position vectors are represented as row matrices, the transformation matrix appears *after* the data or position vector matrix. This is a post-multiplication transformation. Using homogeneous coordinates for positive rotation by an angle  $\theta$  of an object about the origin ( $z$ -axis) using a post-multiplication transformation gives

$$\begin{bmatrix} X^* \\ x^* & y^* & 1 \end{bmatrix} = \begin{bmatrix} X \\ x & y & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2-59)$$

If we choose to represent the position vectors in homogeneous coordinates as a column matrix, then the same rotation is performed using

$$\begin{bmatrix} X^* \\ x^* \\ y^* \\ 1 \end{bmatrix} = \begin{bmatrix} R \\ \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (2-60)$$

Equation (2-60) is called a premultiplication transformation because the transformation matrix appears *before* the column position vector or data matrix. Notice that the  $3 \times 3$  matrix in Eq. (2-60) is also the transpose of the  $3 \times 3$  matrix in Eq. (2-59). That is, the rows and columns have been interchanged.

To rotate the coordinate system and keep the position vectors fixed, simply replace  $\theta$  with  $-\theta$  in Eq. (2-59). Recall that  $\sin \theta = -\sin(-\theta)$  and  $\cos \theta = \cos(-\theta)$ . Equation (2-59) is then

$$\begin{bmatrix} x^* & y^* & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2-61)$$

Notice that the  $3 \times 3$  matrix is again the inverse and also the transpose of that in Eq. (2-59).

If the coordinate system is rotated *and* a left-hand coordinate system used, then the replacement of  $\theta$  with  $-\theta$  is made *twice* and Eq. (2-59) is again valid, assuming a post-multiplication transformation is used on a row data matrix.

Note that, as shown in Fig. 2-16, a counterclockwise rotation of the vertices which represent an object is identical to a clockwise rotation of the coordinate axes for a fixed object. Again, no change occurs in the  $3 \times 3$  transformation

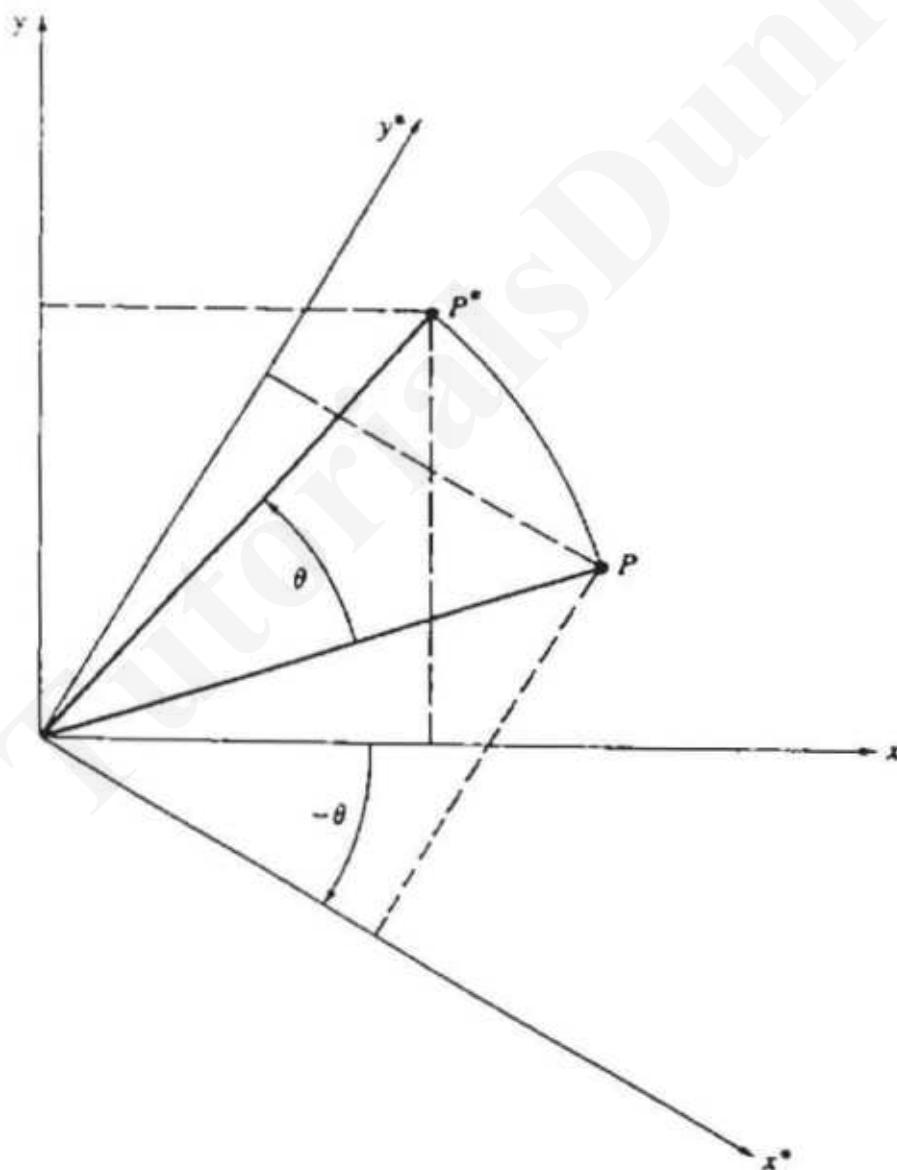


Figure 2-16 Equivalence of position vector and coordinate system rotation.

matrix if no other options are modified. These few examples show how careful we must be when performing matrix transformations.

## 2-22 REFERENCES

- 2-1 Fox, L., *An Introduction to Numerical Linear Algebra*, Oxford University Press, London, 1964.
- 2-2 Forrest, A. R., "Co-ordinates, Transformations, and Visualization Techniques," CAD Group Document No. 23, Cambridge University, June 1969.

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

## CHAPTER

## THREE

THREE-DIMENSIONAL TRANSFORMATIONS  
AND PROJECTIONS

## 3.1 INTRODUCTION

The ability to represent or display a three-dimensional object is fundamental to the understanding of the shape of that object. Furthermore, the ability to rotate, translate, and project views of that object is also, in many cases, fundamental to the understanding of its shape. This is easily demonstrated by picking up a relatively complex unfamiliar object. Immediately it is rotated, held at arm's length, moved up and down, back and forth, etc., in order to obtain an understanding of its shape. To do this with a computer we must extend our previous two-dimensional analysis to three dimensions. Based on our previous experience, we immediately introduce homogeneous coordinates. Hence, a point in three-dimensional space  $[x \ y \ z]$  is represented by a four-dimensional position vector

$$[x' \ y' \ z' \ h] = [x \ y \ z \ 1][T]$$

where  $[T]$  is some transformation matrix. Again, the transformation from homogeneous coordinates to ordinary coordinates is given by

$$[x^* \ y^* \ z^* \ 1] = \left[ \begin{array}{cccc} x' & y' & z' & 1 \\ \hline h & h & h & 1 \end{array} \right] \quad (3-1)$$

The generalized  $4 \times 4$  transformation matrix for three-dimensional homogeneous coordinates is

$$[T] = \begin{bmatrix} a & b & c & p \\ d & e & f & q \\ g & i & j & r \\ l & m & n & s \end{bmatrix} \quad (3-2)$$

The  $4 \times 4$  transformation matrix in Eq. (3-2) can be partitioned into four separate sections:

$$\begin{bmatrix} & & & : & 3 \\ 3 \times 3 & & & : & \times \\ & & & : & 1 \\ \dots & \dots & \dots & : & \dots \\ 1 \times 3 & & & : & 1 \times 1 \end{bmatrix}$$

The upper left  $3 \times 3$  submatrix produces a linear transformation<sup>†</sup> in the form of scaling, shearing, reflection and rotation. The  $1 \times 3$  lower left submatrix produces translation, and the upper right  $3 \times 1$  submatrix produces a perspective transformation. The final lower right-hand  $1 \times 1$  submatrix produces overall scaling. The total transformation obtained after operating on a homogeneous position vector with this  $4 \times 4$  matrix and obtaining the ordinary coordinate is called a bilinear transformation.<sup>‡</sup> In general, this transformation yields a combination of shearing, local scaling, rotation, reflection, translation, perspective and overall scaling.

### 3-2 THREE-DIMENSIONAL SCALING

The diagonal terms of the general  $4 \times 4$  transformation produce local and overall scaling. To illustrate this, consider the transformation

$$[X][T] = [x \ y \ z \ 1] \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & e & 0 & 0 \\ 0 & 0 & j & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [ax \ ey \ jz \ 1] = [x^* \ y^* \ z^* \ 1] \quad (3-3)$$

which shows the local scaling effect. An example follows.

#### Example 3-1 Local Scaling

Consider the rectangular parallelepiped (RPP) shown in Fig. 3-1a with homogeneous position vectors:

$$[X] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 2 & 3 & 1 & 1 \\ 0 & 3 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 1 \\ 2 & 3 & 0 & 1 \\ 0 & 3 & 0 & 1 \end{bmatrix}$$

<sup>†</sup>A linear transformation is one which transforms an initial linear combination of vectors into the same linear combination of transformed vectors.

<sup>‡</sup>A bilinear transformation results from two sequential linear transformations.

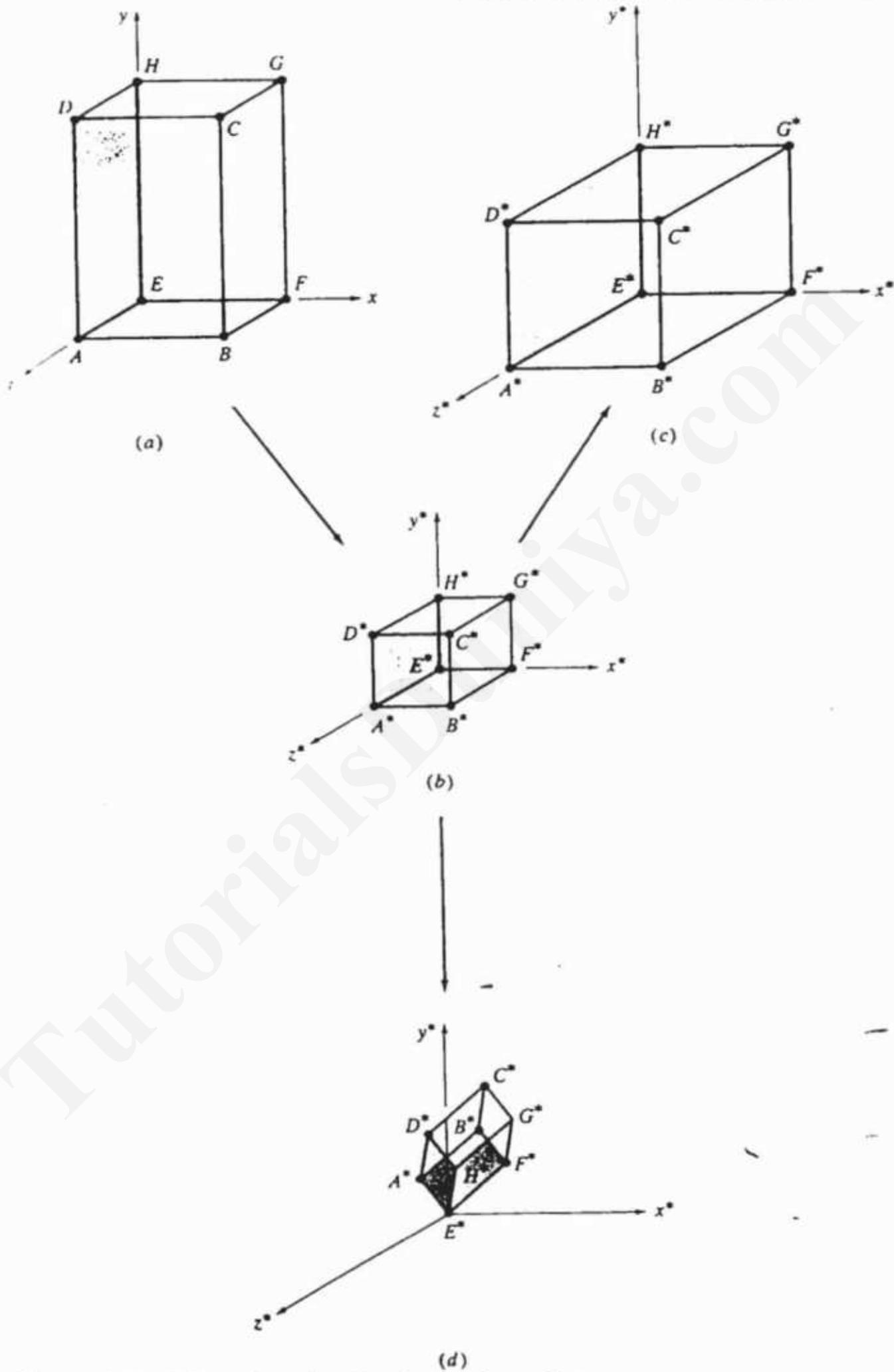


Figure 3-1 Three-dimensional scale transformations.

Locally scaling the *RPP* to yield a unit cube requires scale factors of  $1/2$ ,  $1/3$ ,  $1$  along the  $x, y, z$  axes, respectively. The local scaling transformation is

$$[T] = \begin{bmatrix} 1/2 & 0 & 0 & 0 \\ 0 & 1/3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The resulting cube has homogeneous position vectors

$$\begin{aligned} [X^*] &= [X][T] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 2 & 3 & 1 & 1 \\ 0 & 3 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 1 \\ 2 & 3 & 0 & 1 \\ 0 & 3 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1/2 & 0 & 0 & 0 \\ 0 & 1/3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \end{aligned}$$

Notice that the homogeneous coordinate factor  $h$  is unity for each of the transformed position vectors. The result is shown in Fig. 3-1b.

Overall scaling is obtained by using the fourth diagonal element, i.e.,

$$[X][T] = [x \ y \ z \ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & s \end{bmatrix} = [x' \ y' \ z' \ s] \quad (3-4)$$

The ordinary or physical coordinates are

$$[x^* \ y^* \ z^* \ 1] = \left[ \frac{x'}{s} \ \frac{y'}{s} \ \frac{z'}{s} \ 1 \right]$$

Again, an example illustrates the effect.

### Example 3-2 Overall Scaling

Uniformly scaling the unit cube shown in Fig. 3-1b by a factor of two (doubling the size) requires the transformation (see Eq. 3-4)

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/2 \end{bmatrix}$$

The resulting *RPP* has homogeneous position vectors given by

$$\begin{aligned} [X'] &= [X^*][T] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/2 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 & 1 & 0.5 \\ 1 & 0 & 1 & 0.5 \\ 1 & 1 & 1 & 0.5 \\ 0 & 1 & 1 & 0.5 \\ 0 & 0 & 0 & 0.5 \\ 1 & 0 & 0 & 0.5 \\ 1 & 1 & 0 & 0.5 \\ 0 & 1 & 0 & 0.5 \end{bmatrix} \end{aligned}$$

Notice that the homogeneous coordinate factor for each of the transformed position vectors is  $h = 0.5$ . Thus, to obtain the ordinary or physical coordinates each position vector must be divided by  $h$ . The result, shown in Fig. 3-1c, is

$$[X^*] = \begin{bmatrix} 0 & 0 & 2 & 1 \\ 2 & 0 & 2 & 1 \\ 2 & 2 & 2 & 1 \\ 0 & 2 & 2 & 1 \\ 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 1 \\ 2 & 2 & 0 & 1 \\ 0 & 2 & 0 & 1 \end{bmatrix}$$

Notice here, as in the two-dimensional overall scaling transformation, that the homogeneous coordinate factor is not unity. By analogy with the previous discussion (see Sec. 2-18) this represents transformation out of the physical  $h = 1$  volume into another volume in 4-space. The transformed physical coordinates are obtained by projecting back into the physical  $h = 1$  volume through the center of the 4-space coordinate system. Again, if  $s < 1$ , a uniform expansion of the position vectors occurs. If  $s > 1$ , a uniform compression of the position vectors occurs.

The same effect can be obtained by means of equal local scalings. In this case the transformation matrix is

$$[T] = \begin{bmatrix} 1/s & 0 & 0 & 0 \\ 0 & 1/s & 0 & 0 \\ 0 & 0 & 1/s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice that here the homogeneous coordinate factor is unity, i.e.,  $h = 1$ . Thus, the entire transformation takes place in the  $h = 1$  physical volume.

### 3-3 THREE-DIMENSIONAL SHEARING

The off-diagonal terms in the upper left  $3 \times 3$  submatrix of the generalized  $4 \times 4$  transformation matrix produce shear in three dimensions, i.e.,

$$\begin{aligned} [X][T] &= [x \ y \ z \ 1] \begin{bmatrix} 1 & b & c & 0 \\ d & 1 & f & 0 \\ g & i & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= [x + yd + gz \ bx + y + iz \ cx + fy + z \ 1] \quad (3-5) \end{aligned}$$

An example clarifies these results.

#### Example 3-3 Shearing

Again consider the unit cube shown in Fig. 3-1b. Applying the shearing transformation

$$[T] = \begin{bmatrix} 1 & -0.85 & 0.25 & 0 \\ -0.75 & 1 & 0.7 & 0 \\ 0.5 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

yields

$$\begin{aligned} [X^*] &= [X][T] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & -0.85 & 0.25 & 0 \\ -0.75 & 1 & 0.7 & 0 \\ 0.5 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0.5 & 1 & 1 & 1 \\ 1.5 & 0.15 & 1.25 & 1 \\ 0.75 & 1.15 & 1.95 & 1 \\ -0.25 & 2 & 1.7 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & -0.85 & 0.25 & 1 \\ 0.25 & 0.15 & 0.95 & 1 \\ -0.75 & 1 & 0.7 & 1 \end{bmatrix} \end{aligned}$$

The result is shown in Fig. 3-1d. Notice that in all three examples the origin is unaffected by the transformation.

### 3.4 THREE-DIMENSIONAL ROTATION

Before considering three-dimensional rotation about an arbitrary axis, we examine rotation about each of the coordinate axes. For rotation about the  $x$ -axis, the  $x$  coordinates of the position vectors do not change. In effect, the rotation occurs in planes perpendicular to the  $x$ -axis. Similarly, rotation about the  $y$ - and  $z$ -axes occurs in planes perpendicular to the  $y$ - and  $z$ -axes, respectively. The transformation of the position vectors in each of these planes is governed by the general two-dimensional rotation matrix given in Eq. (2-29). Recalling that matrix, and again noting that for rotation about the  $x$ -axis the  $x$  coordinate of the transformed position vectors does not change, allows writing down the  $4 \times 4$  homogeneous coordinate transformation by the angle  $\theta$  as

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-6)$$

Rotation is assumed positive in a right-hand sense, i.e., clockwise as one looks outward from the origin in the positive direction along the rotation axis.<sup>†</sup> The block shown in Fig. 3-2b is the result of a  $-90^\circ$  rotation about the  $x$ -axis of the block shown in Fig. 3-2a.

In a similar manner, the transformation matrix for rotation by an angle  $\psi$  about the  $z$ -axis is

$$[T] = \begin{bmatrix} \cos \psi & \sin \psi & 0 & 0 \\ -\sin \psi & \cos \psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-7)$$

For rotation by an angle  $\phi$  about the  $y$ -axis, the transformation is

$$[T] = \begin{bmatrix} \cos \phi & 0 & -\sin \phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin \phi & 0 & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-8)$$

Note that in Eq. 3-8 the signs of the sine terms are reversed from those of Eqs. (3-6) and (3-7). This is required to maintain the positive right-hand rule convention.

Examining Eqs. (3-6) to (3-8) shows that the determinant of each transformation matrix is  $+1$  as required for pure rotation. An example more fully illustrates these results.

<sup>†</sup>The right-hand rule for rotation is stated as follows: align the thumb of the right hand with the positive direction of the rotation axis. The natural curl of the fingers gives the positive rotation direction.

**Example 3-4 Rotation**

Consider the rectangular parallelepiped shown in Fig. 3-2a. The matrix of position vectors  $[X]$  is

$$[X] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 3 & 0 & 1 & 1 \\ 3 & 2 & 1 & 1 \\ 0 & 2 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 1 \\ 3 & 2 & 0 & 1 \\ 0 & 2 & 0 & 1 \end{bmatrix} A$$

Here, the row labeled  $A$  in the position matrix  $[X]$  corresponds to the point  $A$  in Fig. 3-2. For rotation by  $\theta = -90^\circ$  about the  $x$ -axis Eq. (3-6) yields

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

Applying the transformation gives the new position vectors

$$\begin{aligned} [X^*] &= [X][T] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 3 & 0 & 1 & 1 \\ 3 & 2 & 1 & 1 \\ 0 & 2 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 1 \\ 3 & 2 & 0 & 1 \\ 0 & 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 & 0 & 1 \\ 3 & 1 & 0 & 1 \\ 3 & 1 & -2 & 1 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 1 \\ 3 & 0 & -2 & 1 \\ 0 & 0 & -2 & 1 \end{bmatrix} A^* \end{aligned}$$

Notice that the  $x$  components of  $[X]$  and  $[X^*]$  are identical as required. The result is shown in Fig. 3-2b.

For rotation by  $\phi = 90^\circ$  about the  $y$ -axis, Eq. (3-7) yields

$$[T'] = \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

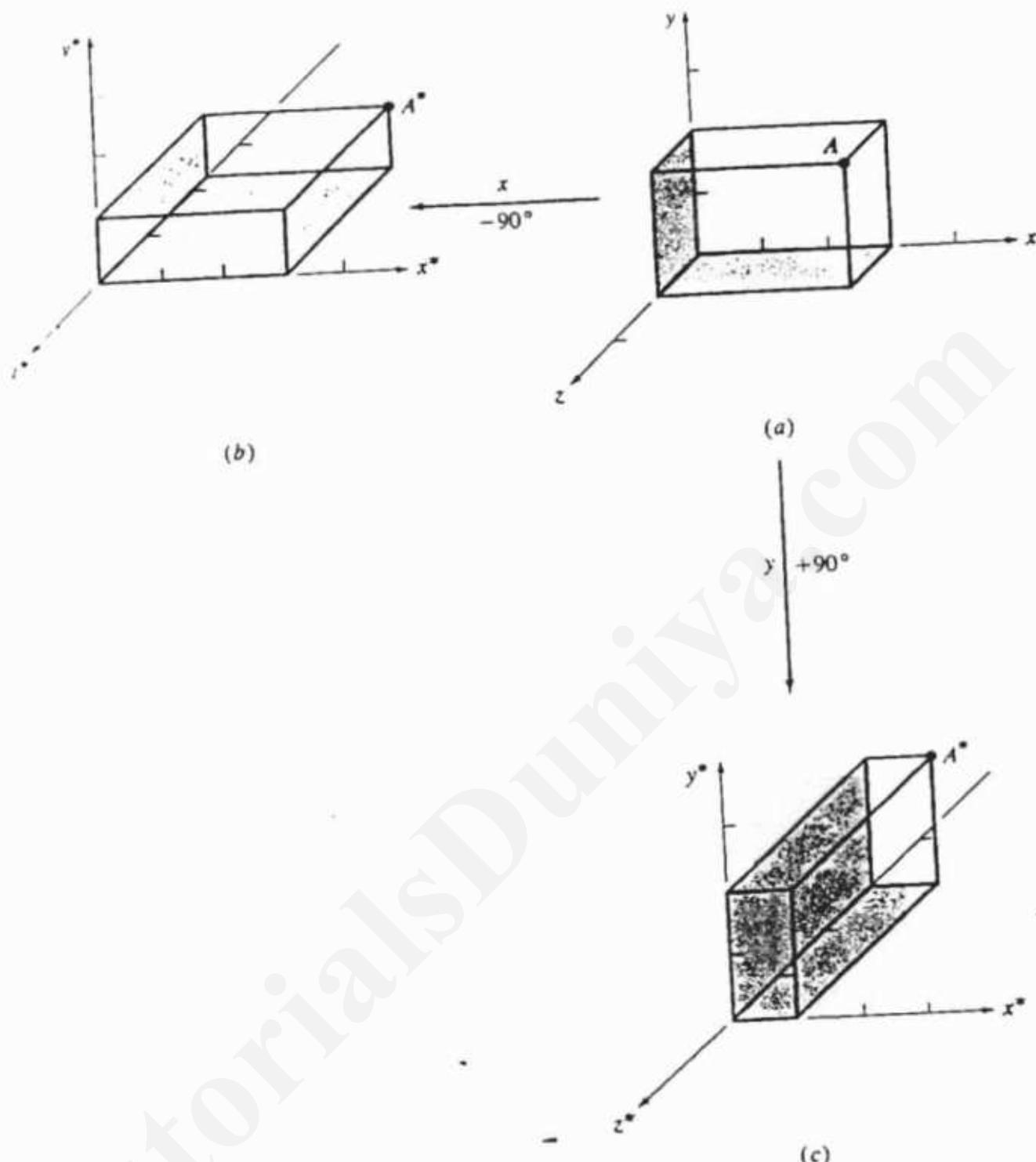


Figure 3-2 Three-dimensional rotations.

Again applying the transformation to the original block yields the new position vectors, i.e.,

$$[X'^*] = [X][T'] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 3 & 0 & 1 & 1 \\ 3 & 2 & 1 & 1 \\ 0 & 2 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 1 \\ 3 & 2 & 0 & 1 \\ 0 & 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & -3 & 1 \\ 1 & 2 & -3 & 1 \\ 1 & 2 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -3 & 1 \\ 0 & 2 & -3 & 1 \\ 0 & 2 & 0 & 1 \end{bmatrix} A^*$$

Notice that in this case the  $y$  components of  $[X]$  and  $[X'']$  are identical. The result is shown in Fig. 3-2c.

Since three-dimensional rotations are obtained using matrix multiplication, they are noncommutative; i.e., the order of multiplication affects the final result (see Sec. 2-12). In order to show this, consider a rotation about the  $x$ -axis followed by an equal rotation about the  $y$ -axis. Using Eqs. (3-6) and (3-8) with  $\theta = \phi$ , we have

$$\begin{aligned} [T] &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ \sin^2 \theta & \cos \theta & \cos \theta \sin \theta & 0 \\ \cos \theta \sin \theta & -\sin \theta & \cos^2 \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-9) \end{aligned}$$

On the other hand, the reverse operation, i.e., a rotation about the  $y$ -axis followed by an equal rotation about the  $x$ -axis with  $\theta = \phi$ , yields

$$\begin{aligned} [T] &= \begin{bmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos \theta & \sin^2 \theta & -\cos \theta \sin \theta & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ \sin \theta & -\cos \theta \sin \theta & \cos^2 \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-10) \end{aligned}$$

Comparison of the right-hand sides of Eqs. (3-9) and (3-10) shows that they are not the same. The fact that three-dimensional rotations are noncommutative must be kept in mind when more than one rotation is to be made.

The result of transformation of the object in Fig. 3-3a consisting of two 90° rotations using the matrix product given in Eq. (3-9) is shown dashed in Figs. 3-3c and 3-3d. When the opposite order of rotation as specified by Eq. (3-10) is used, the solid figures shown in Figs. 3-3b and 3-3d graphically demonstrate that different results are obtained by changing the order of rotation. A numerical example further illustrates this concept.

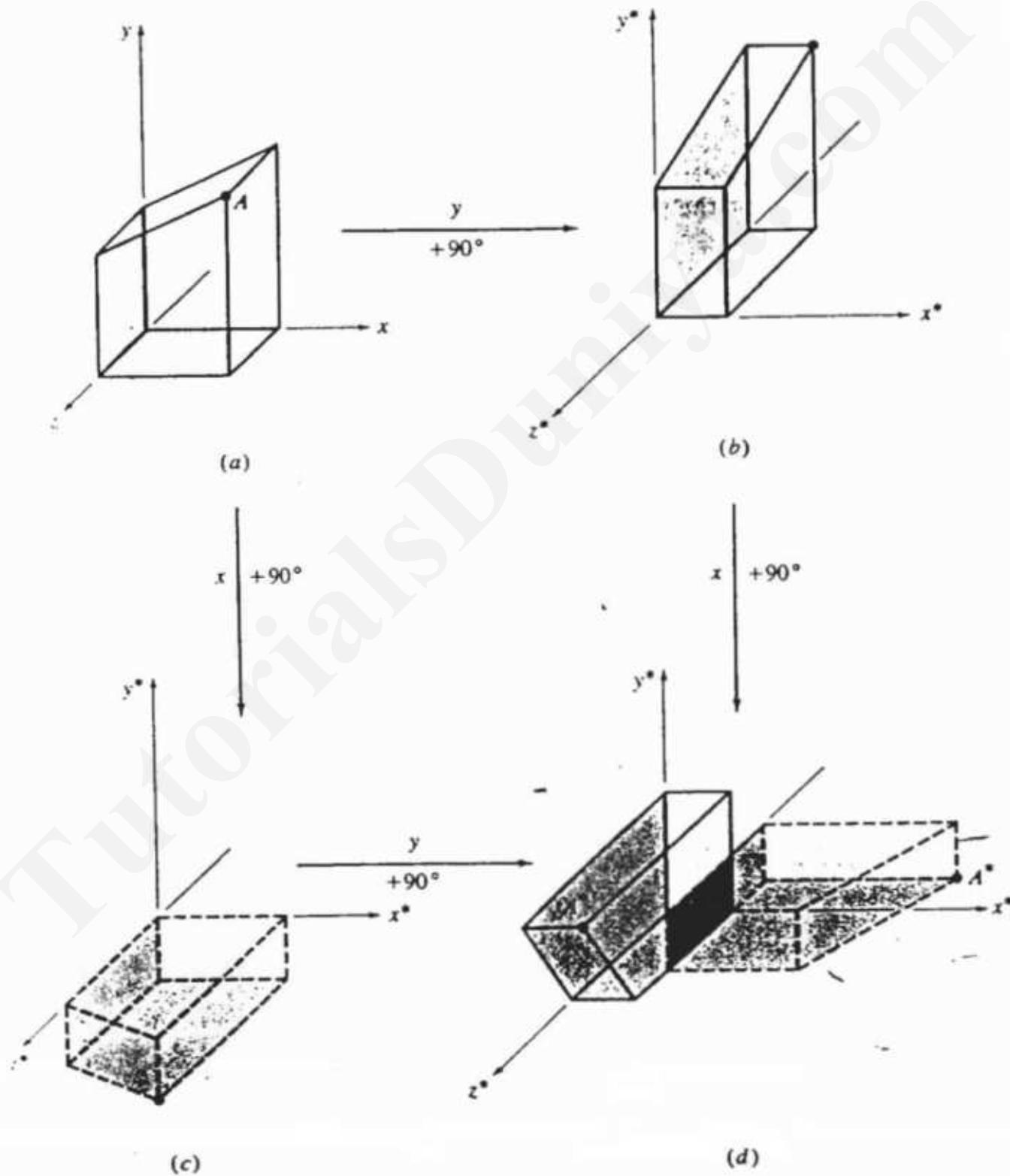


Figure 3-3 Three-dimensional rotations are noncommutative.

**Example 3-5 Combined Rotations**

The object in Fig. 3-3a has position vectors

$$[X] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 2 & 3 & 1 & 1 \\ 0 & 2 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 1 \\ 2 & 3 & 0 & 1 \\ 0 & 2 & 0 & 1 \end{bmatrix} A$$

The concatenated matrix for a rotation about the  $x$ -axis by  $\theta = 90^\circ$  followed by a rotation about the  $y$ -axis by  $\phi = 90^\circ$  is given by Eq. (3-9) as

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

The transformed position vectors are

$$\begin{aligned} [X^*] &= [X][T] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 2 & 3 & 1 & 1 \\ 0 & 2 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 1 \\ 2 & 3 & 0 & 1 \\ 0 & 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & -1 & 0 & 1 \\ 0 & -1 & -2 & 1 \\ 3 & -1 & -2 & 1 \\ 2 & -1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -2 & 1 \\ 3 & 0 & -2 & 1 \\ 2 & 0 & 0 & 1 \end{bmatrix} A^* \end{aligned}$$

The transformed object is shown dashed in Fig. 3-3d.

The concatenated matrix for a rotation about the  $y$ -axis by  $\phi = 90^\circ$  followed by a rotation about the  $x$ -axis by  $\theta = 90^\circ$  is given by Eq. (3-10) as

$$[T'] = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Here, the resulting transformed position vectors are

$$[X''] = [X][T'] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 2 & 3 & 1 & 1 \\ 0 & 2 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 1 \\ 2 & 3 & 0 & 1 \\ 0 & 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 2 & 0 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 1 \\ 0 & 2 & 3 & 1 \\ 0 & 0 & 2 & 1 \end{bmatrix} A''$$

The transformed object is shown by solid lines in Fig. 3-3d.

Comparing the two numerical results also clearly shows that the orientation of the transformed objects is considerably different. Hence, the order of matrix multiplication is important.

### 3.5 THREE DIMENSIONAL REFLECTION

Some orientations of a three-dimensional object cannot be obtained using pure rotations; they require reflections. In three dimensions, reflections occur through a plane. By analogy with the previous discussion of two-dimensional reflection (see Sec. 2-10), three-dimensional reflection through a plane is equivalent to rotation about an axis in three-dimensional space out into four-dimensional space and back into the original three-dimensional space. For a pure reflection the determinant of the reflection matrix is identically  $-1$ .

In a reflection through the  $xy$  plane, only the  $z$  coordinate values of the object's position vectors change. In fact, they are reversed in sign. Thus, the transformation matrix for a reflection through the  $xy$  plane is

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-11)$$

The reflection of a unit cube through the  $xy$  plane is shown in Fig. 3-4. For a reflection through the  $yz$  plane,

$$[T] = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-12)$$

and for a reflection through the  $xz$  plane,

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-13)$$

**Example 3-6 Reflection**

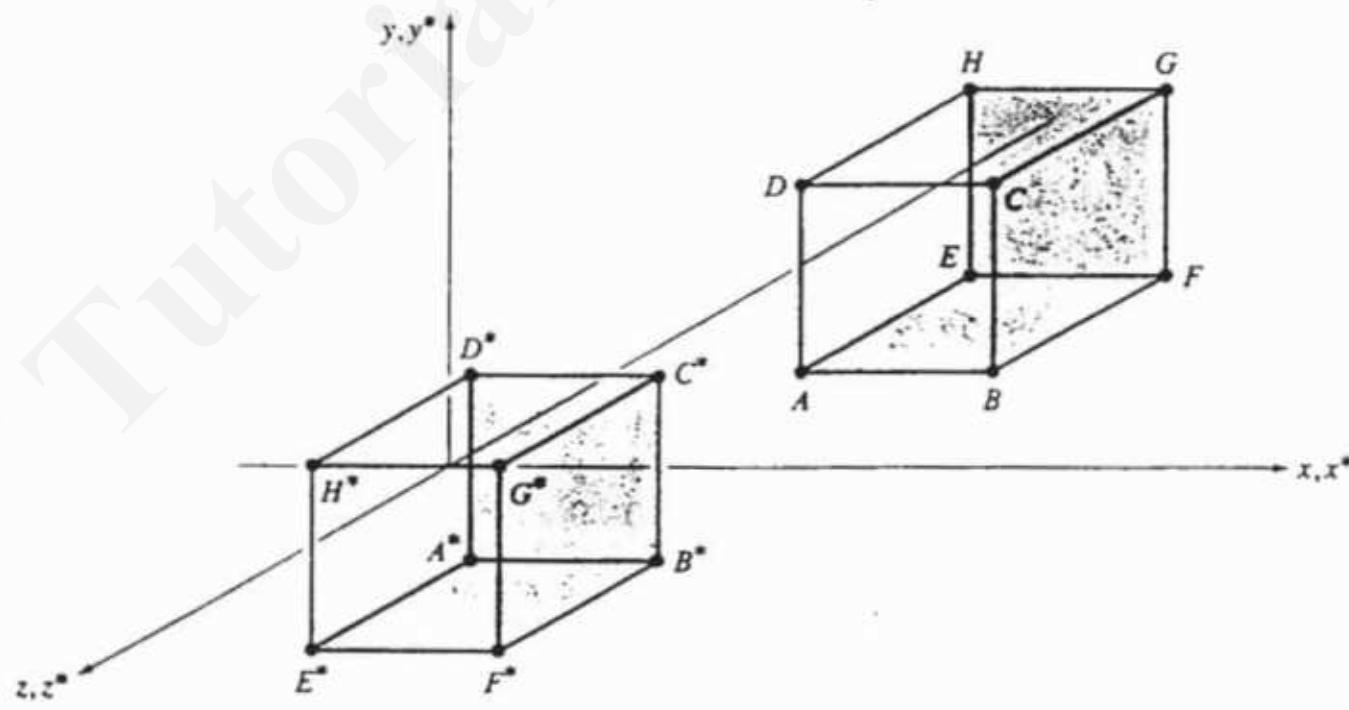
The block  $ABCDEFGH$  shown in Fig. 3-4 has position vectors

$$[X] = \begin{bmatrix} 1 & 0 & -1 & 1 \\ 2 & 0 & -1 & 1 \\ 2 & 1 & -1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 0 & -2 & 1 \\ 2 & 0 & -2 & 1 \\ 2 & 1 & -2 & 1 \\ 1 & 1 & -2 & 1 \end{bmatrix}$$

The transformation matrix for reflection through the  $xy$  plane is given by Eq. (3-11). After reflection the transformed position vectors are

$$[X^*] = [X][T] = \begin{bmatrix} 1 & 0 & -1 & 1 \\ 2 & 0 & -1 & 1 \\ 2 & 1 & -1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 0 & -2 & 1 \\ 2 & 0 & -2 & 1 \\ 2 & 1 & -2 & 1 \\ 1 & 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 2 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 2 & 1 \\ 2 & 0 & 2 & 1 \\ 2 & 1 & 2 & 1 \\ 1 & 1 & 2 & 1 \end{bmatrix}$$

The result  $A^*B^*C^*D^*E^*F^*G^*H^*$  is shown in Fig. 3-4.



**Figure 3-4** Three-dimensional reflection through the  $xy$  plane.

### 3-6 THREE DIMENSIONAL TRANSLATION

The three-dimensional translation matrix is

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ l & m & n & 1 \end{bmatrix} \quad (3-14)$$

The translated homogeneous coordinates are obtained by writing

$$\begin{bmatrix} x' & y' & z' & h \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ l & m & n & 1 \end{bmatrix}$$

When expanded this yields

$$\begin{bmatrix} x' & y' & z' & h \end{bmatrix} = \begin{bmatrix} (x + l) & (y + m) & (z + n) & 1 \end{bmatrix} \quad (3-15)$$

It follows that the transformed physical coordinates are

$$\begin{aligned} x^* &= x + l \\ y^* &= y + m \\ z^* &= z + n \end{aligned}$$

### 3-7 MULTIPLE TRANSFORMATIONS

Successive transformations can be combined or concatenated into a single  $4 \times 4$  transformation that yields the same result. Since matrix multiplication is noncommutative, the order of application is important (in general  $[A][B] \neq [B][A]$ ). The proper order is determined by the position of the individual transformation matrix relative to the position vector matrix. The matrix nearest the position vector matrix generates the first individual transformation, and the farthest, the last individual transformation. Mathematically this is expressed as

$$[X][T] = [X][T_1][T_2][T_3][T_4] \cdots$$

where

$$[T] = [T_1][T_2][T_3][T_4] \cdots$$

and the  $[T_i]$  are any combination of scaling, shearing, rotation, reflection, translation, perspective and projective matrices. Since perspective transformations distort geometric objects (see Sec. 3-15) and projective transformations result in lost information (see Sec. 3-12), if these matrices are included, they must occur next to last and last in the order, respectively.

The example below explicitly illustrates this concept.

**Example 3-7 Multiple Transformations**

Consider the effect of a translation in the  $x, y, z$  directions by  $-1, -1, -1$ , respectively, followed successively by a  $+30^\circ$  rotation about the  $x$ -axis, and a  $+45^\circ$  rotation about the  $y$ -axis on the homogeneous coordinate position vector  $[3 \ 2 \ 1 \ 1]$ .

First derive the concatenated transformation matrix. Using Eqs. (3-14), (3-6) and (3-8) yields

$$\begin{aligned}
 [T] &= [Tr][R_x][R_y] \\
 &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ l & m & n & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\phi & 0 & -\sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ l & m & n & 1 \end{bmatrix} \begin{bmatrix} \cos\phi & 0 & -\sin\phi & 0 \\ \sin\phi\sin\theta & \cos\theta & \cos\phi\sin\theta & 0 \\ \sin\phi\cos\theta & -\sin\theta & \cos\phi\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} \cos\phi & 0 & -\sin\phi & 0 \\ \sin\phi\sin\theta & \cos\theta & \cos\phi\sin\theta & 0 \\ \sin\phi\cos\theta & -\sin\theta & \cos\phi\cos\theta & 0 \\ l\cos\phi & m\cos\theta & -l\sin\phi & 1 \\ +m\sin\phi\sin\theta & -n\sin\theta & +m\cos\phi\sin\theta & 0 \\ +n\sin\phi\cos\theta & +n\cos\phi\cos\theta & +n\cos\phi\cos\theta & 0 \end{bmatrix} \tag{3-16}
 \end{aligned}$$

where  $\theta, \phi$  are the rotation angles about the  $x$ - and  $y$ -axes, respectively; and  $l, m, n$  are the translation factors in the  $x, y, z$  directions, respectively.

For a general position vector we have

$$\begin{aligned}
 [X][T] &= [x \ y \ z \ 1] \begin{bmatrix} \cos\phi & 0 & -\sin\phi & 0 \\ \sin\phi\sin\theta & \cos\theta & \cos\phi\sin\theta & 0 \\ \sin\phi\cos\theta & -\sin\theta & \cos\phi\cos\theta & 0 \\ l\cos\phi & m\cos\theta & -l\sin\phi & 1 \\ +m\sin\phi\sin\theta & -n\sin\theta & +m\cos\phi\sin\theta & 0 \\ +n\sin\phi\cos\theta & +n\cos\phi\cos\theta & +n\cos\phi\cos\theta & 0 \end{bmatrix} \\
 &= \begin{bmatrix} (x+l)\cos\phi & (y+m)\cos\theta & -(x+l)\sin\phi & 1 \\ +(y+m)\sin\phi\sin\theta & -(z+n)\sin\theta & +(y+m)\cos\phi\sin\theta & 0 \\ +(z+n)\sin\phi\cos\theta & +(z+n)\cos\phi\cos\theta & +(z+n)\cos\phi\cos\theta & 0 \end{bmatrix}
 \end{aligned}$$

For specific values of  $\theta = +30^\circ$ ,  $\phi = +45^\circ$ ,  $l = -1$ ,  $m = -1$ ,  $n = -1$  the transformed position vector is  $[3 \ 2 \ 1 \ 1]$ .

$$[X][T] = [3 \ 2 \ 1 \ 1] \begin{bmatrix} 0.707 & 0 & -0.707 & 0 \\ 0.354 & 0.866 & 0.354 & 0 \\ 0.612 & -0.5 & 0.612 & 0 \\ -1.673 & -0.366 & -0.259 & 1 \end{bmatrix}$$

$$[X][T] = [1.768 \ 0.866 \ -1.061 \ 1]$$

To confirm that the concatenated matrix yields the same result as individually applied matrices consider

$$[X'] = [X][Tr]$$

$$= [3 \ 2 \ 1 \ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & -1 & -1 & 1 \end{bmatrix}$$

$$= [2 \ 1 \ 0 \ 1]$$

$$[X''] = [X'][R_z] = [2 \ 1 \ 0 \ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & 0 \\ 0 & -0.5 & 0.866 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= [2 \ 0.866 \ 0.5 \ 1]$$

$$[X'''] = [X''][R_y] = [2 \ 0.866 \ .5 \ 1] \begin{bmatrix} 0.707 & 0 & -0.707 & 0 \\ 0 & 1 & 0 & 0 \\ 0.707 & 0 & 0.707 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

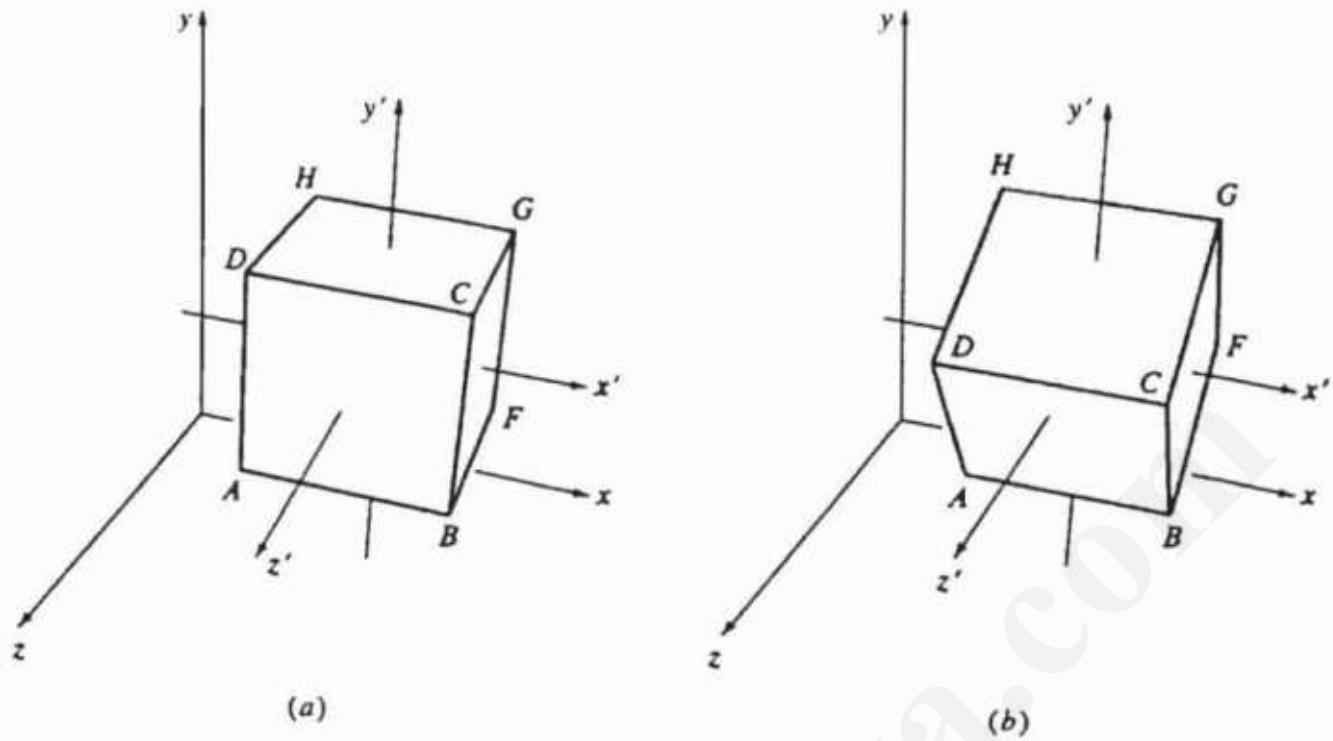
$$[X'''] = [1.768 \ 0.866 \ -1.061 \ 1]$$

which confirms our previous result.

### 3-8 ROTATIONS ABOUT AN AXIS PARALLEL TO A COORDINATE AXIS

The transformations given in Eqs. (3-6) to (3-8) cause rotation about the  $x$ ,  $y$  and  $z$  coordinate axes. Often it is necessary to rotate an object about an axis other than these. Here, the special case of an axis that is parallel to one of the  $x$ ,  $y$  or  $z$  coordinate axes is considered. Figure 3-5 shows a body with a local axis system  $x'y'z'$  parallel to the fixed global axis system  $xyz$ . Rotation of the body about any of the individual  $x'$ ,  $y'$  or  $z'$  local axes is accomplished using the following procedure:

Translate the body until the local axis is coincident with the coordinate axis in the same direction.



**Figure 3-5** Rotation about an axis parallel to one of the coordinate axes.

Rotate about the specified axis.

Translate the transformed body back to its original position.

**Mathematically**

$$[X^*] = [X][Tr][R_x][Tr]^{-1}$$

where

- $[X^*]$  represents the transformed body
  - $[X]$  is the untransformed body
  - $[Tr]$  is the translation matrix
  - $[R_x]$  is the appropriate rotation matrix
  - $[Tr]^{-1}$  is the inverse of the translation matrix

An illustrative example is given below.

### Example 3-8 Single Relative Rotation

Consider the block in Fig. 3-5a defined by the position vectors

$$[X] = \begin{bmatrix} 1 & 1 & 2 & 1 \\ 2 & 1 & 2 & 1 \\ 2 & 2 & 2 & 1 \\ 1 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 1 \\ 2 & 2 & 1 & 1 \\ 1 & 2 & 1 & 1 \end{bmatrix} \begin{matrix} A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \end{matrix}$$

relative to the global  $xyz$ -axis system. Let's rotate the block  $\theta = +30^\circ$  about the local  $x'$ -axis passing through the centroid of the block. The origin of the local axis system is assumed to be the centroid of the block.

The centroid of the block is  $[x_c \ y_c \ z_c \ 1] = [3/2 \ 3/2 \ 3/2 \ 1]$ . The rotation is accomplished by

$$[X^*] = [X][Tr][R][Tr]^{-1}$$

where

$$[Tr] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -y_c & -z_c & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -3/2 & -3/2 & 1 \end{bmatrix}$$

$$[R_x] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & 0 \\ 0 & -0.5 & 0.866 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and

$$[Tr]^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & y_c & z_c & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 3/2 & 3/2 & 1 \end{bmatrix}$$

The first matrix  $[Tr]$  translates the block parallel to the  $x = 0$  plane until the  $x'$ -axis is coincident with the  $x$ -axis. The second matrix  $[R_x]$  performs the required rotation about the  $x$ -axis, and the third matrix  $[Tr]^{-1}$  translates the  $x'$ -axis and hence the rotated block back to its original position. Concatenating these three matrices yields

$$[T] = [Tr][R_x][Tr]^{-1}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & y_c(1 - \cos \theta) + z_c \sin \theta & z_c(1 - \cos \theta) - y_c \sin \theta & 1 \end{bmatrix}$$

After substituting numerical values the transformed coordinates are

$$[X'] = [X][T] = \begin{bmatrix} 1 & 1 & 2 & 1 \\ 2 & 1 & 2 & 1 \\ 2 & 2 & 2 & 1 \\ 1 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 1 \\ 2 & 2 & 1 & 1 \\ 1 & 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & 0 \\ 0 & -0.5 & 0.866 & 0 \\ 0 & 0.951 & -0.549 & 1 \end{bmatrix}$$

$$[X'] = \begin{bmatrix} 1 & 0.817 & 1.683 & 1 \\ 2 & 0.817 & 1.683 & 1 \\ 2 & 1.683 & 2.183 & 1 \\ 1 & 1.683 & 2.183 & 1 \\ 1 & 1.317 & 0.817 & 1 \\ 2 & 1.317 & 0.817 & 1 \\ 2 & 2.183 & 1.317 & 1 \\ 1 & 2.183 & 1.317 & 1 \end{bmatrix} \begin{matrix} A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \end{matrix}$$

The result is shown in Fig. 3-5b.

In the previous example only rotation about a single axis parallel to one of the coordinate axes was required. Thus, it was only necessary to make the rotation axis coincident with the corresponding coordinate axis. If multiple rotations in a local axis system parallel to the global axis system are required, then the origin of the local axis system must be made coincident with that of the global axis system. Specifically, the rotations can be accomplished with the following procedure:

Translate the origin of the local axis system to make it coincident with that of the global coordinate system.

Perform the required rotations.

Translate the local axis system back to its original position.

The example below illustrates this procedure.

### Example 3-9 Multiple Relative Rotations

Again consider the block shown on Fig. 3-5a. To rotate the block  $\phi = -45^\circ$  about the  $y'$ -axis, followed by a rotation of  $\theta = +30^\circ$  about the  $x'$ -axis, requires that the origin of the  $x'y'z'$ -axis system be made coincident with the origin of the  $xyz$ -axis system, the rotations performed and then the result translated back to the original position.

The combined transformation is

$$[X'] = [X][T] = [X][Tr][R_y][R_x][Tr]^{-1}$$

Specifically,

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_c & -y_c & -z_c & 1 \end{bmatrix} \begin{bmatrix} \cos \phi & 0 & -\sin \phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin \phi & 0 & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_c & y_c & z_c & 1 \end{bmatrix}$$

where  $\phi$  and  $\theta$  represent the rotation angle about the  $y'$ - and  $x'$ -axes respectively. Concatenating these matrices yields

$$[T] = \begin{bmatrix} \cos \phi & \sin \phi \sin \theta & -\sin \phi \cos \theta & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ \sin \phi & -\cos \phi \sin \theta & \cos \phi \cos \theta & 0 \\ x_c(1 - \cos \phi) & -x_c \sin \phi \sin \theta & x_c \sin \phi \cos \theta & 1 \\ -z_c \sin \phi & +y_c(1 - \cos \theta) & -y_c \sin \theta & \\ & +z_c \cos \phi \sin \theta & +z_c(1 - \cos \phi \cos \theta) & \end{bmatrix} \quad (3-17)$$

The transformed position vectors are then

$$[X'] = \begin{bmatrix} 1 & 1 & 2 & 1 \\ 2 & 1 & 2 & 1 \\ 2 & 2 & 2 & 1 \\ 1 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 1 \\ 2 & 2 & 1 & 1 \\ 1 & 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0.707 & -0.354 & 0.612 & 0 \\ 0 & 0.866 & 0.5 & 0 \\ -0.707 & -0.354 & 0.612 & 0 \\ 1.5 & 1.262 & -1.087 & 1 \end{bmatrix}$$

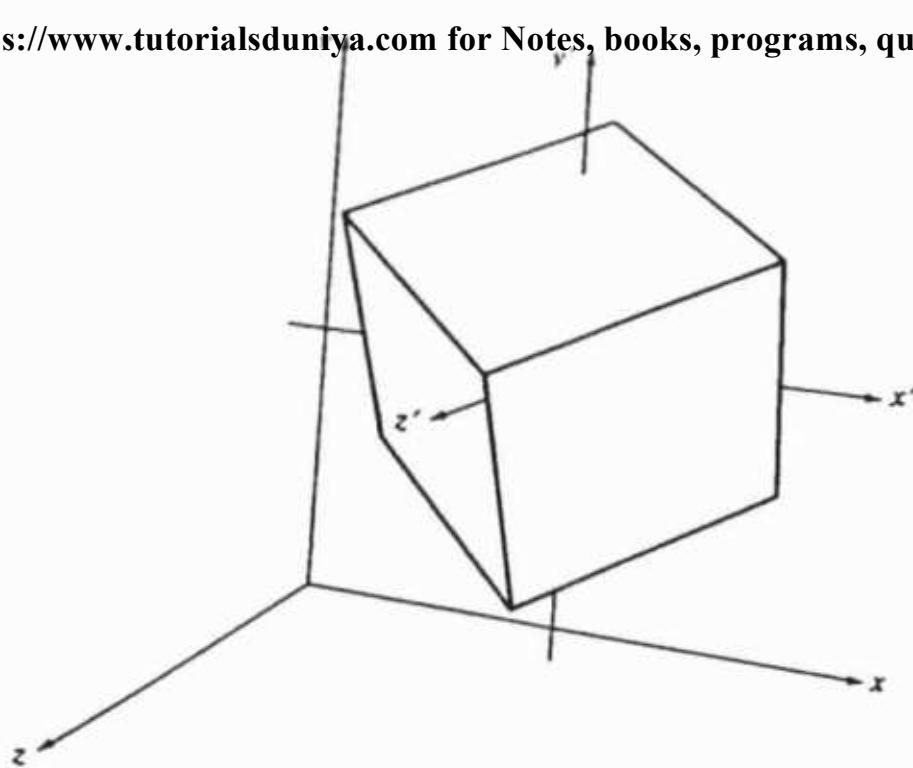
$$[X'] = \begin{bmatrix} 0.793 & 1.067 & 1.25 & 1 \\ 1.5 & 0.713 & 1.862 & 1 \\ 1.5 & 1.579 & 2.362 & 1 \\ 0.793 & 1.933 & 1.75 & 1 \\ 1.5 & 1.421 & 0.638 & 1 \\ 2.207 & 1.067 & 1.25 & 1 \\ 2.207 & 1.933 & 1.75 & 1 \\ 1.5 & 2.287 & 1.138 & 1 \end{bmatrix}$$

The result is shown in Fig. 3-6.

### 3-9 ROTATION ABOUT AN ARBITRARY AXIS IN SPACE

The general case of rotation about an arbitrary axis in space frequently occurs e.g., in robotics, animation, and simulation. Following the previous discussion, rotation about an arbitrary axis in space is accomplished with a procedure using translations and simple rotations about the coordinate axes. Since the technique for rotation about a coordinate axis is known, the underlying procedural idea is to make the arbitrary rotation axis coincident with one of the coordinate axes.

Assume an arbitrary axis in space passing through the point  $(x_0, y_0, z_0)$  with direction cosines  $(c_x, c_y, c_z)$ . Rotation about this axis by some angle  $\delta$  is accomplished using the following procedure:



**Figure 3-6** Multiple rotations about a local axis system.

Translate so that the point  $(x_0, y_0, z_0)$  is at the origin of the coordinate system.

Perform appropriate rotations to make the axis of rotation coincident with the  $z$ -axis.<sup>†</sup>

Rotate about the  $z$ -axis by the angle  $\delta$ .

Perform the inverse of the combined rotation transformation.

Perform the inverse of the translation.

In general, making an arbitrary axis passing through the origin coincident with one of the coordinate axes requires two successive rotations about the other two coordinate axes. To make the arbitrary rotation axis coincident with the  $z$ -axis, first rotate about the  $x$ -axis and then about the  $y$ -axis. To determine the rotation angle,  $\alpha$ , about the  $x$ -axis used to place the arbitrary axis in the  $xz$  plane, first project the unit vector along the axis onto the  $yz$  plane as shown in Fig. 3-7a. The  $y$  and  $z$  components of the projected vector are  $c_y$  and  $c_z$ , the direction cosines of the unit vector along the arbitrary axis. From Fig. 3-7a we have that

$$d = \sqrt{c_y^2 + c_z^2} \quad (3-18)$$

and

$$\cos \alpha = \frac{c_z}{d} \quad \sin \alpha = \frac{c_y}{d} \quad (3-19)$$

---

<sup>†</sup>The choice of the  $z$ -axis is arbitrary.

After rotation about the  $x$ -axis into the  $xz$  plane, the  $z$  component of the unit vector is  $d$ , and the  $x$  component is  $c_x$ , the direction cosine in the  $x$  direction as shown in Fig. 3-7b. The length of the unit vector is, of course, 1. Thus, the rotation angle  $\beta$  about the  $y$ -axis required to make the arbitrary axis coincident with the  $z$ -axis is

$$\cos \beta = d \quad \sin \beta = c_x \quad (3 - 20)$$

The complete transformation is then

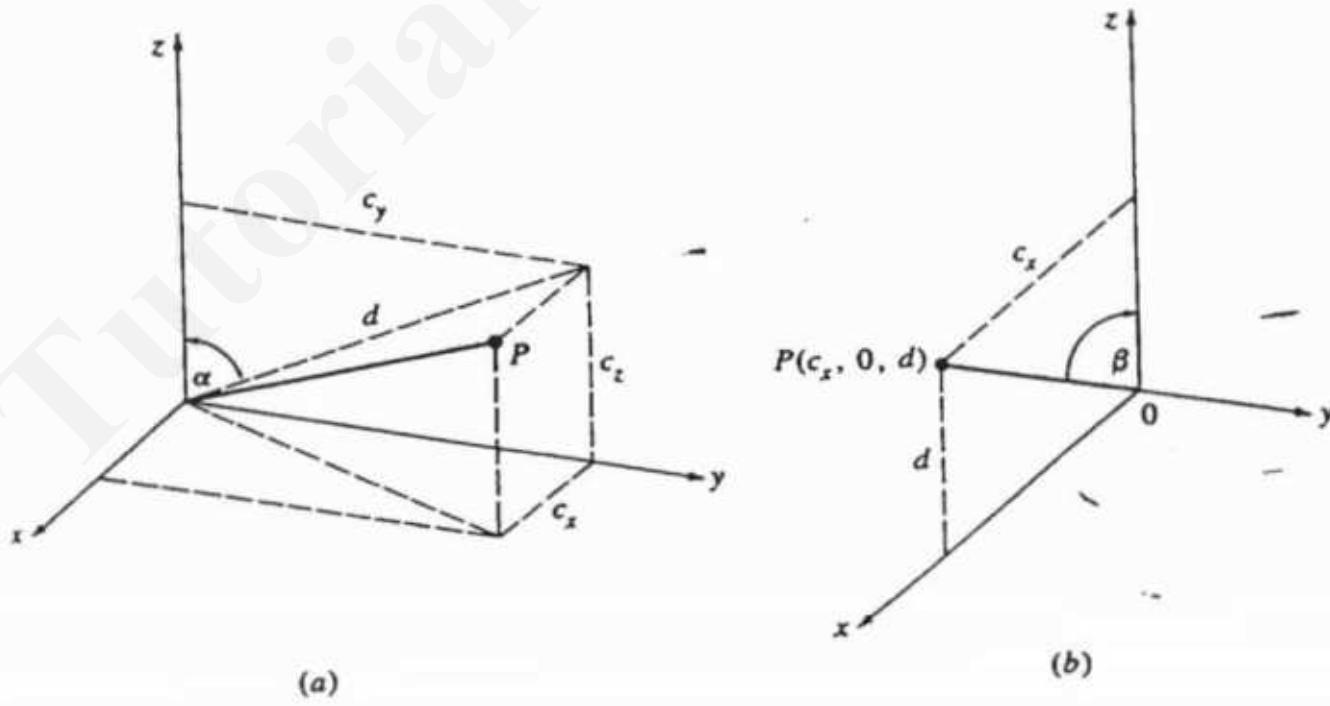
$$[M] = [T][R_x][R_y][R_\delta][R_y]^{-1}[R_x]^{-1}[T]^{-1} \quad (3 - 21)$$

where the required translation matrix is

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_0 & -y_0 & -z_0 & 1 \end{bmatrix} \quad (3 - 22)$$

The transformation matrix for rotation about the  $x$ -axis is

$$[R_x] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & \sin \alpha & 0 \\ 0 & -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c_z/d & c_y/d & 0 \\ 0 & -c_y/d & c_z/d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3 - 23)$$



**Figure 3-7** Rotations required to make the unit vector  $OP$  coincident with the  $z$ -axis.  
 (a) Rotation about  $x$ ; (b) rotation about  $y$ .

and about the  $y$ -axis

$$[ R_y ] = \begin{bmatrix} \cos(-\beta) & 0 & -\sin(-\beta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(-\beta) & 0 & \cos(-\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} d & 0 & c_x & 0 \\ 0 & 1 & 0 & 0 \\ -c_x & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-24)$$

Finally, the rotation about the arbitrary axis is given by a  $z$ -axis rotation matrix

$$[ R_\delta ] = \begin{bmatrix} \cos \delta & \sin \delta & 0 & 0 \\ -\sin \delta & \cos \delta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-25)$$

In practice, the angles  $\alpha$  and  $\beta$  are not explicitly calculated. The elements of the rotation matrices  $[ R_x ]$  and  $[ R_y ]$  in Eq. (3-21) are obtained from Eqs. (3-18) to (3-20) at the expense of two divisions and a square root calculation. Although developed with the arbitrary axis in the first quadrant, these results are applicable in all quadrants.

If the direction cosines of the arbitrary axis are not known, they can be obtained knowing a second point on the axis  $(x_1, y_1, z_1)$  by normalizing the vector from the first to the second point. Specifically, the vector along the axis from  $(x_0, y_0, z_0)$  to  $(x_1, y_1, z_1)$  is

$$[ V ] = [ (x_1 - x_0) \quad (y_1 - y_0) \quad (z_1 - z_0) ]$$

Normalized, it yields the direction cosines

$$[ c_x \quad c_y \quad c_z ] = \frac{[ (x_1 - x_0) \quad (y_1 - y_0) \quad (z_1 - z_0) ]}{[(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2]^{\frac{1}{2}}} \quad (3-26)$$

An example more fully illustrates the procedure.

### Example 3-10 Rotation About an Arbitrary Axis

Consider the cube with one corner removed shown in Fig. 3-8a. Position vectors for the vertices are

$$[ X ] = \left[ \begin{array}{cccc} 2 & 1 & 2 & 1 \\ 3 & 1 & 2 & 1 \\ 3 & 1.5 & 2 & 1 \\ 2.5 & 2 & 2 & 1 \\ 2 & 2 & 2 & 1 \\ 2 & 1 & 1 & 1 \\ 3 & 1 & 1 & 1 \\ 3 & 2 & 1 & 1 \\ 2 & 2 & 1 & 1 \\ 3 & 2 & 1.5 & 1 \end{array} \right] \begin{array}{l} A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \\ I \\ J \end{array}$$

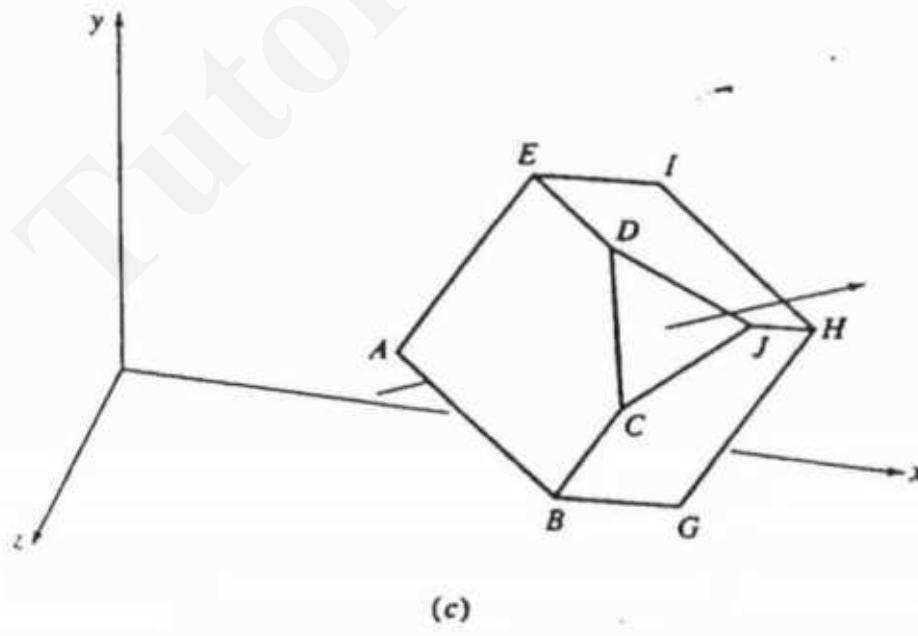
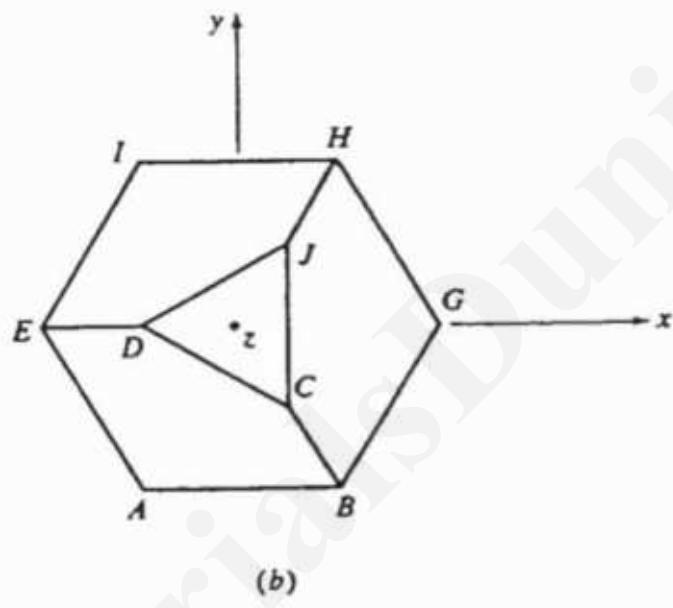
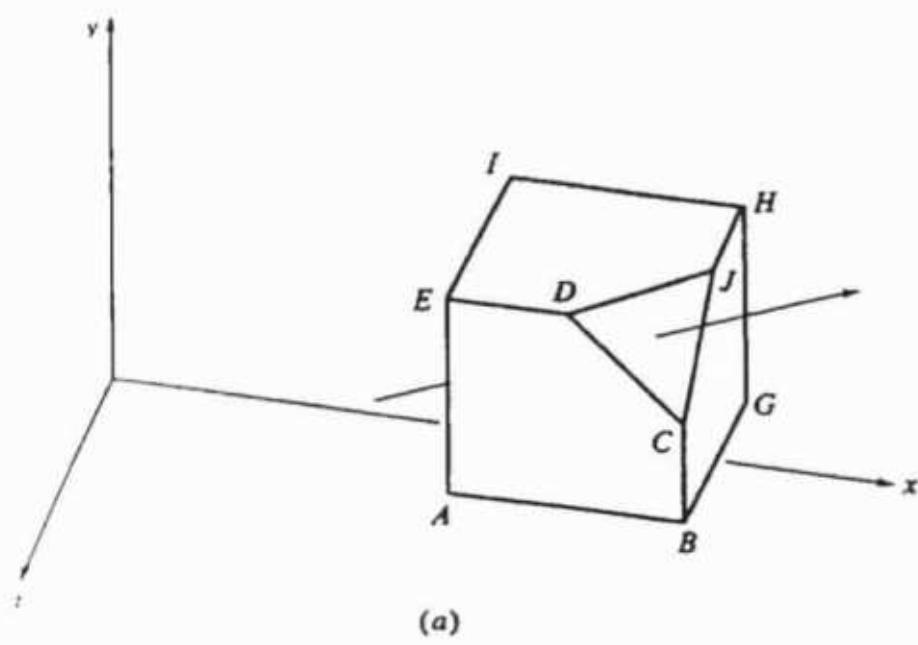


Figure 3–8 Rotation about an arbitrary axis.

Visit <https://www.tutorialsduniya.com> for Notes, books, programs, question papers with solutions etc.

The cube is to be rotated by  $-45^\circ$  about a local axis passing through the point  $F$  and the diagonally opposite corner. The axis is directed from  $F$  to the opposite corner and passes through the center of the corner face.

First, determine the direction cosines of the rotation axis. Observing that the corner cut off by the triangle  $CDJ$  also lies on the axis, Eq. (3-26) yields

$$\begin{bmatrix} c_x & c_y & c_z \end{bmatrix} = \frac{\begin{bmatrix} (3-2) & (2-1) & (2-1) \end{bmatrix}}{((3-2)^2 + (2-1)^2 + (2-1)^2)^{\frac{1}{2}}} \\ = \begin{bmatrix} 1/\sqrt{3} & 1/\sqrt{3} & 1/\sqrt{3} \end{bmatrix}$$

Using Eqs. (3-18) to (3-20) yields

$$d = \sqrt{(1/\sqrt{3})^2 + (1/\sqrt{3})^2} = \sqrt{2/3}$$

and

$$\alpha = \cos^{-1}(1/\sqrt{3}/\sqrt{2/3}) = \cos^{-1}(1/\sqrt{2}) = 45^\circ$$

$$\beta = \cos^{-1}(\sqrt{2/3}) = 35.26^\circ$$

Since the point  $F$  lies on the rotation axis, the translation matrix is

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -2 & -1 & -1 & 1 \end{bmatrix}$$

The rotation matrices to make the arbitrary axis coincident with the  $z$ -axis are then

$$[R_x] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & -1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and

$$[R_y] = \begin{bmatrix} 2/\sqrt{6} & 0 & 1/\sqrt{3} & 0 \\ 0 & 1 & 0 & 0 \\ -1/\sqrt{3} & 0 & 2/\sqrt{6} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$[R_x]^{-1}$ ,  $[R_y]^{-1}$ , and  $[T]^{-1}$  are obtained by substituting  $-\alpha$ ,  $-\beta$  and  $(x_0, y_0, z_0)$  for  $\alpha$ ,  $\beta$  and  $(-x_0, -y_0, -z_0)$ , respectively, in Eqs. (3-22) to (3-24).

Concatenating  $[T][R_x][R_y]$  yields

$$[M] = [T][R_x][R_y] = \begin{bmatrix} 2/\sqrt{6} & 0 & 1/\sqrt{3} & 0 \\ -1/\sqrt{6} & 1/\sqrt{2} & 1/\sqrt{3} & 0 \\ -1/\sqrt{6} & -1/\sqrt{2} & 1/\sqrt{3} & 0 \\ -2/\sqrt{6} & 0 & -4/\sqrt{3} & 1 \end{bmatrix}$$

The transformed intermediate position vectors are

$$[X][M] = \begin{bmatrix} -0.408 & -0.707 & 0.577 & 1 \\ 0.408 & -0.707 & 1.155 & 1 \\ 0.204 & -0.354 & 1.443 & 1 \\ -0.408 & 0 & 1.443 & 1 \\ -0.816 & 0 & 1.155 & 1 \\ 0 & 0 & 0 & 1 \\ 0.816 & 0 & 0.577 & 1 \\ 0.408 & 0.707 & 1.155 & 1 \\ -0.408 & 0.707 & 0.577 & 1 \\ 0.204 & 0.354 & 1.443 & 1 \end{bmatrix}$$

This intermediate result is shown in Fig. 3-8b. Notice that point F is at (0, 0, 0).

The rotation about the arbitrary axis is now given by the equivalent rotation about the z-axis. Hence (see Eq. 3-7)

$$[R_\delta] = \begin{bmatrix} \sqrt{2}/2 & -\sqrt{2}/2 & 0 & 0 \\ \sqrt{2}/2 & \sqrt{2}/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The transformed object is returned to its 'original' position in space, using

$$[M]^{-1} = [R_y]^{-1}[R_x]^{-1}[T]^{-1} = \begin{bmatrix} 2/\sqrt{6} & -1/\sqrt{6} & -1/\sqrt{6} & 0 \\ 0 & 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 1/\sqrt{3} & 1/\sqrt{3} & 1/\sqrt{3} & 0 \\ 2 & 1 & 1 & 1 \end{bmatrix}$$

This result can be obtained either by concatenating the inverses of the individual component matrices of  $[M]$  or by formally taking the inverse of  $[M]$ . Incidentally, notice that  $[R_x][R_y]$  is a pure rotation. The upper left  $3 \times 3$  submatrix of  $[M]^{-1}$  is just the transpose of the upper left  $3 \times 3$  submatrix of  $[M]$ .

The resulting position vectors are

$$[X][M][R_\delta][M]^{-1} = \begin{bmatrix} 1.689 & 1.506 & 1.805 & 1 \\ 2.494 & 1.195 & 2.311 & 1 \\ 2.747 & 1.598 & 2.155 & 1 \\ 2.598 & 2.155 & 1.747 & 1 \\ 2.195 & 2.311 & 1.494 & 1 \\ 2 & 1 & 1 & 1 \\ 2.805 & 0.689 & 1.506 & 1 \\ 3.311 & 1.494 & 1.195 & 1 \\ 2.506 & 1.805 & 0.689 & 1 \\ 3.155 & 1.747 & 1.598 & 1 \end{bmatrix}$$

where

$$[M][R_s][M]^{-1} = \begin{bmatrix} 0.805 & -0.311 & 0.506 & 0 \\ 0.506 & 0.805 & -0.311 & 0 \\ -0.311 & 0.506 & 0.805 & 0 \\ 0.195 & 0.311 & -0.506 & 1 \end{bmatrix}$$

The transformed object is shown in Fig. 3-8c.

---

### 3-10 REFLECTION THROUGH AN ARBITRARY PLANE

The transformations given in Eqs. (3-11) to (3-13) cause reflection through the  $x = 0$ ,  $y = 0$ ,  $z = 0$  coordinate planes, respectively. Often it is necessary to reflect an object through a plane other than one of these. Again, this can be accomplished using a procedure incorporating the previously defined simple transformations. One possible procedure is:

Translate a known point  $P$ , that lies in the reflection plane, to the origin of the coordinate system.

Rotate the normal vector to the reflection plane at the origin until it is coincident with the  $+z$ -axis (see Sec. 3-9, Eqs. 3-23 and 3-24); this makes the reflection plane the  $z = 0$  coordinate plane.

After also applying the above transformations to the object, reflect the object through the  $z = 0$  coordinate plane (see Eq. 3-11).

Perform the inverse transformations to those given above to achieve the desired result.

The general transformation is then

$$[M] = [T][R_x][R_y][R_{fltx}][R_y]^{-1}[R_x]^{-1}[T]^{-1}$$

where the matrices  $[T]$ ,  $[R_x]$ ,  $[R_y]$  are given by Eqs. (3-22) to (3-24), respectively.  $(x_0, y_0, z_0) = (P_x, P_y, P_z)$ , the components of point  $P$  in the reflection plane; and  $(c_x, c_y, c_z)$  are the direction cosines of the normal to the reflection plane.<sup>†</sup>

An example more fully illustrates the procedure.

---

#### Example 3-11 Reflection

Again consider the cube with one corner removed shown in Fig. 3-8a. Reflect the cube through the plane containing the triangle  $CDJ$ .

<sup>†</sup>If the equation of the reflection plane  $ax + by + cz + d = 0$  is known, then the unit normal to the plane is

$$[\hat{n}] = [c_x \quad c_y \quad c_z] = \frac{[a \quad b \quad c]}{\sqrt{a^2 + b^2 + c^2}}$$

See Ref. 3-1 for more details.

Recalling the position vectors for the cube, and choosing to translate the point  $C$  to the origin, yields the translation matrix

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & -3/2 & -2 & 1 \end{bmatrix}$$

The normal to the reflection plane is obtained using the position vectors  $C$ ,  $D$ ,  $J$  (see Ref. 3-1). Specifically, taking the cross product of the vectors  $CJ$  and  $CD$  prior to translation yields

$$\begin{aligned} n &= ([J] - [C]) \times ([D] - [C]) \\ &= [(3-3) \quad (2-1.5) \quad (1.5-2)] \times [(2.5-3) \quad (2-1.5) \quad (2-2)] \\ &= [0 \quad 1/2 \quad -1/2] \times [-1/2 \quad 1/2 \quad 0] \\ &= [1/4 \quad 1/4 \quad 1/4] \end{aligned}$$

Normalizing yields

$$\hat{n} = [1/\sqrt{3} \quad 1/\sqrt{3} \quad 1/\sqrt{3}]$$

Using Eqs. (3-19) and (3-20) gives

$$d = \sqrt{n_y^2 + n_z^2} = \sqrt{(1/\sqrt{3})^2 + (1/\sqrt{3})^2} = \sqrt{2/3}$$

and

$$\alpha = 45^\circ \quad \beta = 35.26^\circ$$

The rotation matrices to make the normal at  $C$  coincide with the  $z$ -axis are (see Eqs. 3-23 and 3-24)

$$[R_x] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & -1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$[R_y] = \begin{bmatrix} 2/\sqrt{6} & 0 & 1/\sqrt{3} & 0 \\ 0 & 1 & 0 & 0 \\ -1/\sqrt{3} & 0 & 2/\sqrt{6} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The matrices  $[R_x]^{-1}$ ,  $[R_y]^{-1}$  and  $[T]^{-1}$  are obtained by substituting  $-\alpha$ ,  $-\beta$ , and  $[x_0 \quad y_0 \quad z_0] = [C]$  into Eqs. (3-22) to (3-24).

Concatenating  $[T]$ ,  $[R_x]$  and  $[R_y]$  yields

$$[M] = [T][R_x][R_y] = \begin{bmatrix} 2/\sqrt{6} & 0 & 1/\sqrt{3} & 0 \\ -1/\sqrt{6} & 1/\sqrt{2} & 1/\sqrt{3} & 0 \\ -1/\sqrt{6} & -1/\sqrt{2} & 1/\sqrt{3} & 0 \\ -5/2\sqrt{6} & 1/2\sqrt{2} & -13/2\sqrt{3} & 1 \end{bmatrix}$$

The transformed intermediate position vectors are

$$[X][M] = \begin{bmatrix} -0.612 & -0.354 & -0.876 & 1 \\ 0.204 & -0.354 & -0.287 & 1 \\ 0 & 0 & 0 & 1 \\ -0.612 & 0.354 & 0 & 1 \\ -1.021 & 0.354 & -0.287 & 1 \\ -0.204 & 0.354 & -1.443 & 1 \\ 0.612 & 0.354 & -0.876 & 1 \\ 0.204 & 1.061 & -0.287 & 1 \\ -0.612 & 1.061 & -0.876 & 1 \\ 0 & 0.707 & 0 & 1 \end{bmatrix}$$

This intermediate result is shown in Fig. 3-9b. Notice that the point C is at the origin and the z-axis points out of the page.

Reflection through the arbitrary plane is now given by reflection through the  $z = 0$  plane. Hence (see Eq. 3-11)

$$[R_{flt}] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Returning the transformed object to its 'original' position in space requires

$$[M]^{-1} = [R_y]^{-1} [R_x]^{-1} [T]^{-1} = \begin{bmatrix} 2/\sqrt{6} & -1/\sqrt{6} & -1/\sqrt{6} & 0 \\ 0 & 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 1/\sqrt{3} & 1/\sqrt{3} & 1/\sqrt{3} & 0 \\ 3 & 3/2 & 2 & 1 \end{bmatrix}$$

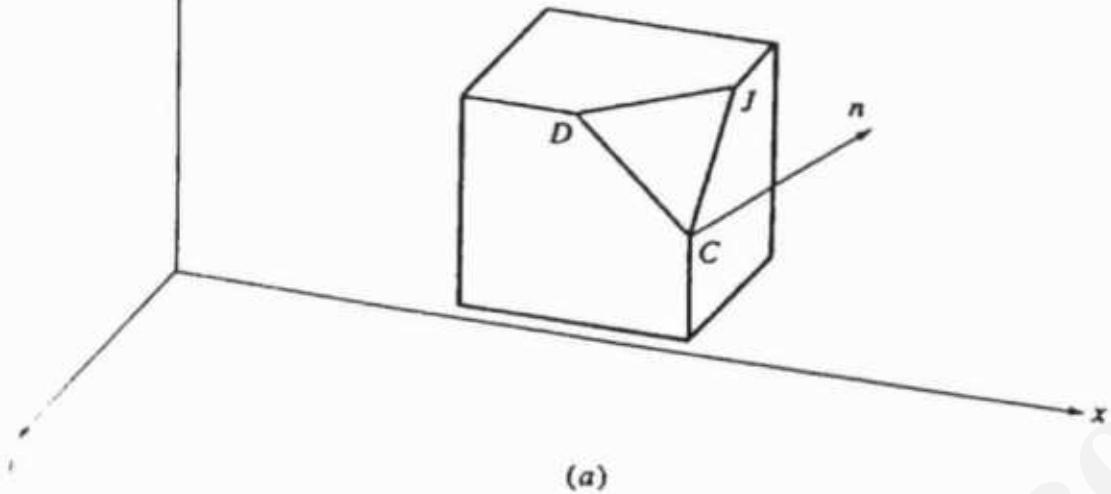
The resulting position vectors are

$$[X][M][R_{flt}][M]^{-1} = \begin{bmatrix} 3 & 2 & 3 & 1 \\ 10/3 & 4/3 & 7/3 & 1 \\ 3 & 3/2 & 2 & 1 \\ 5/2 & 2 & 2 & 1 \\ 7/3 & 7/3 & 7/3 & 1 \\ 11/3 & 8/3 & 8/3 & 1 \\ 4 & 2 & 2 & 1 \\ 10/3 & 7/3 & 4/3 & 1 \\ 3 & 3 & 2 & 1 \\ 3 & 2 & 3/2 & 1 \end{bmatrix}$$

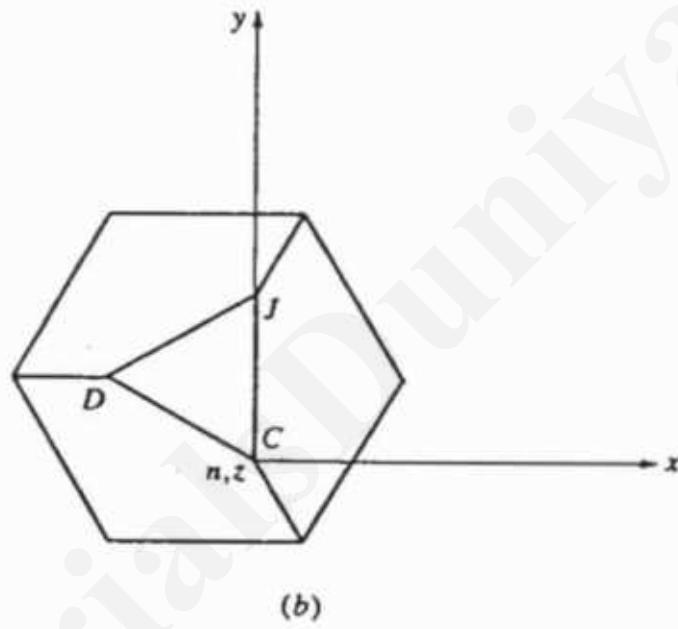
where

$$[M][R_{flt}][M]^{-1} = \begin{bmatrix} 1/3 & -2/3 & -2/3 & 0 \\ -2/3 & 1/3 & -2/3 & 0 \\ -2/3 & -2/3 & 1/3 & 0 \\ 13/3 & 13/3 & 13/3 & 1 \end{bmatrix}$$

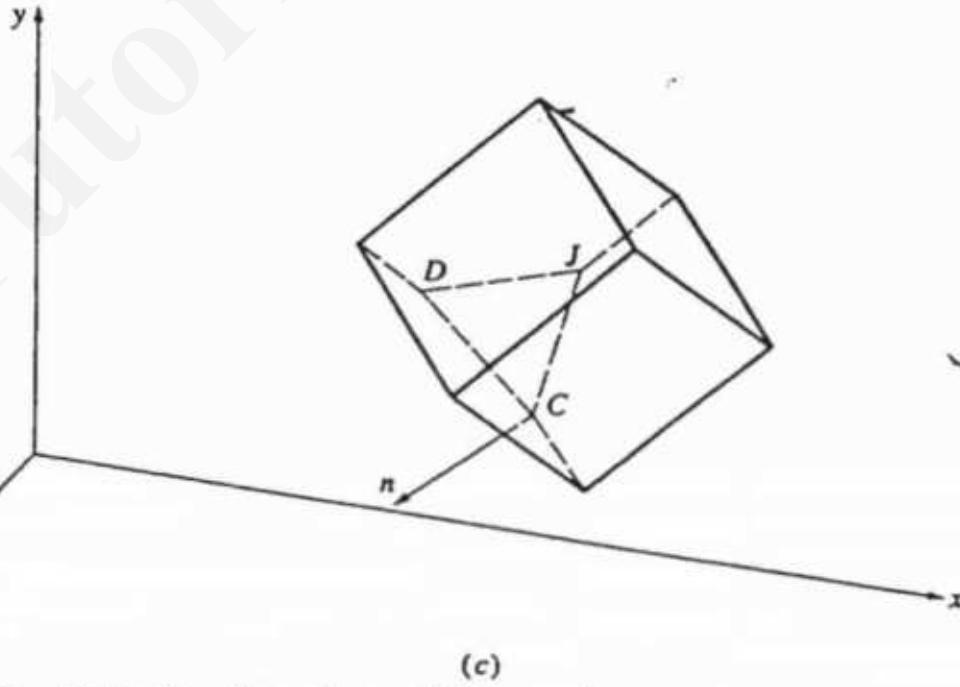
The transformed object is shown in Fig. 3-9c.



(a)



(b)



(c)

Figure 3-9 Reflection through an arbitrary plane.

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

As this and the previous section show, complex manipulative transformations can easily be constructed using procedures involving simple single-action transformations. This is the recommended approach. Generally it is less error prone and is computationally more efficient than a direct mathematical approach.

### 3-11 AFFINE AND PERSPECTIVE GEOMETRY

Geometric theorems have been developed for both perspective and affine geometry. The theorems of affine geometry are identical to those for Euclidean geometry. In both affine and Euclidean geometry parallelism is an important concept. In perspective geometry, lines are generally nonparallel.

An affine transformation is a combination of linear transformations, e.g., rotation followed by translation. For an affine transformation, the last column in the general  $4 \times 4$  transformation matrix is  $[0 \ 0 \ 0 \ 1]^T$ . Otherwise, as shown in Sec. 3-15 below, the transformed homogeneous coordinate  $h$  is not unity; and there is not a one-to-one correspondence between the affine transformation and the  $4 \times 4$  matrix operator. Affine transformations form a useful subset of bilinear transformations, since the product of two affine transformations is also affine. This allows the general transformation of a set of points relative to an arbitrary coordinate system while maintaining a value of unity for the homogeneous coordinate  $h$ .

Since Euclidean geometry has been taught in schools for many years, drawing and sketching techniques based on Euclidean geometry have become standard methods for graphical communication. Although perspective views are often used by artists and architects to yield more realistic pictures, because of the difficulty of manual construction they are seldom used in technical work. However, with the use of homogeneous coordinates to define an object, both affine and perspective transformations are obtained with equal ease.

Both affine and perspective transformations are three-dimensional, i.e., they are transformations from one three space to another three space. However, viewing the results on a two-dimensional surface requires a projection from three space to two space. The result is called a plane geometric projection. Figure 3-10 illustrates the hierarchy of plane geometric projections. The projection matrix from three space to two space always contains a column of zeros. Consequently the determinant of a projective transformation is always zero.

Plane geometric projections of objects are formed by the intersection of lines called projectors with a plane called the projection plane. Projectors are lines from an arbitrary point called the center of projection, through each point in an object. If the center of projection is located at a finite point in three space, the result is a perspective projection. If the center of projection is located at infinity, all the projectors are parallel and the result is a parallel projection. Plane geometric projections provide the basis for descriptive geometry. Nonplanar and nongeometric projections are also useful; e.g., they are used extensively in cartography.

In developing the various transformations shown in Fig. 3-10 two alternate approaches can be used. The first assumes that the center of projection or eye

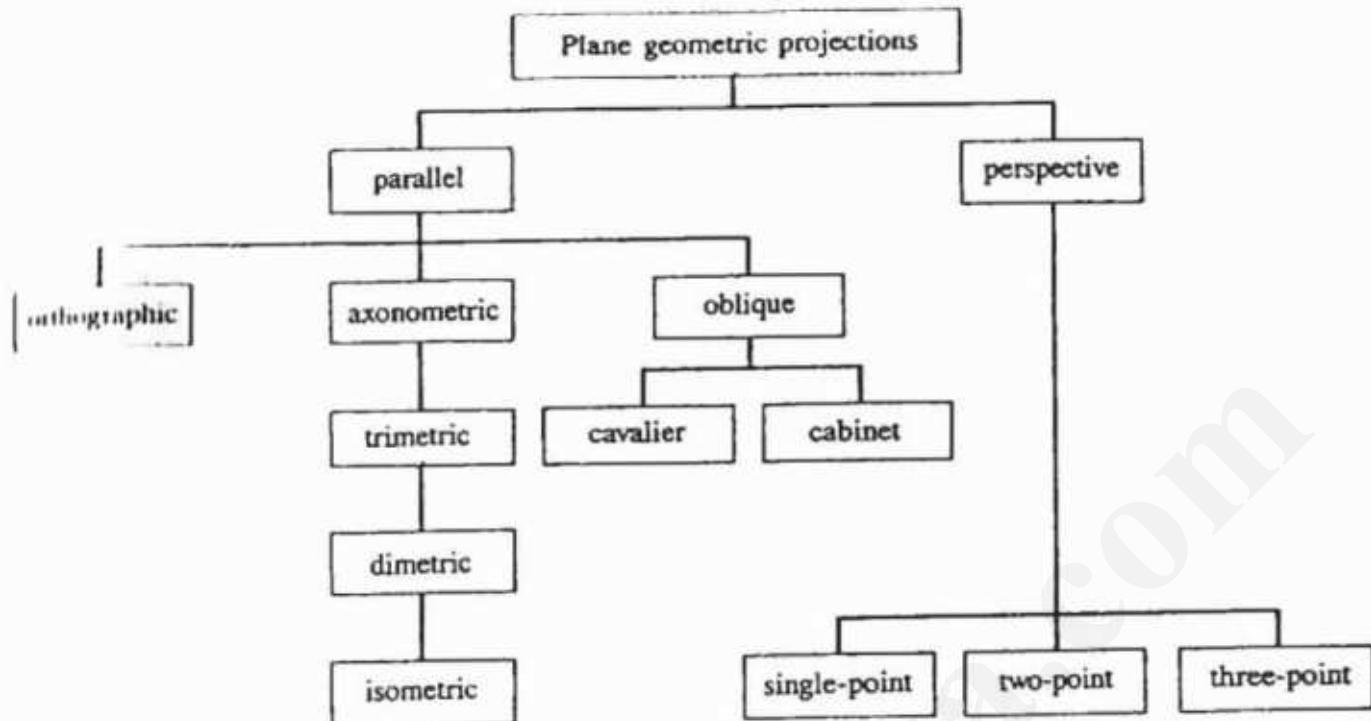


Figure 3–10 Hierarchy of plane geometric projections.

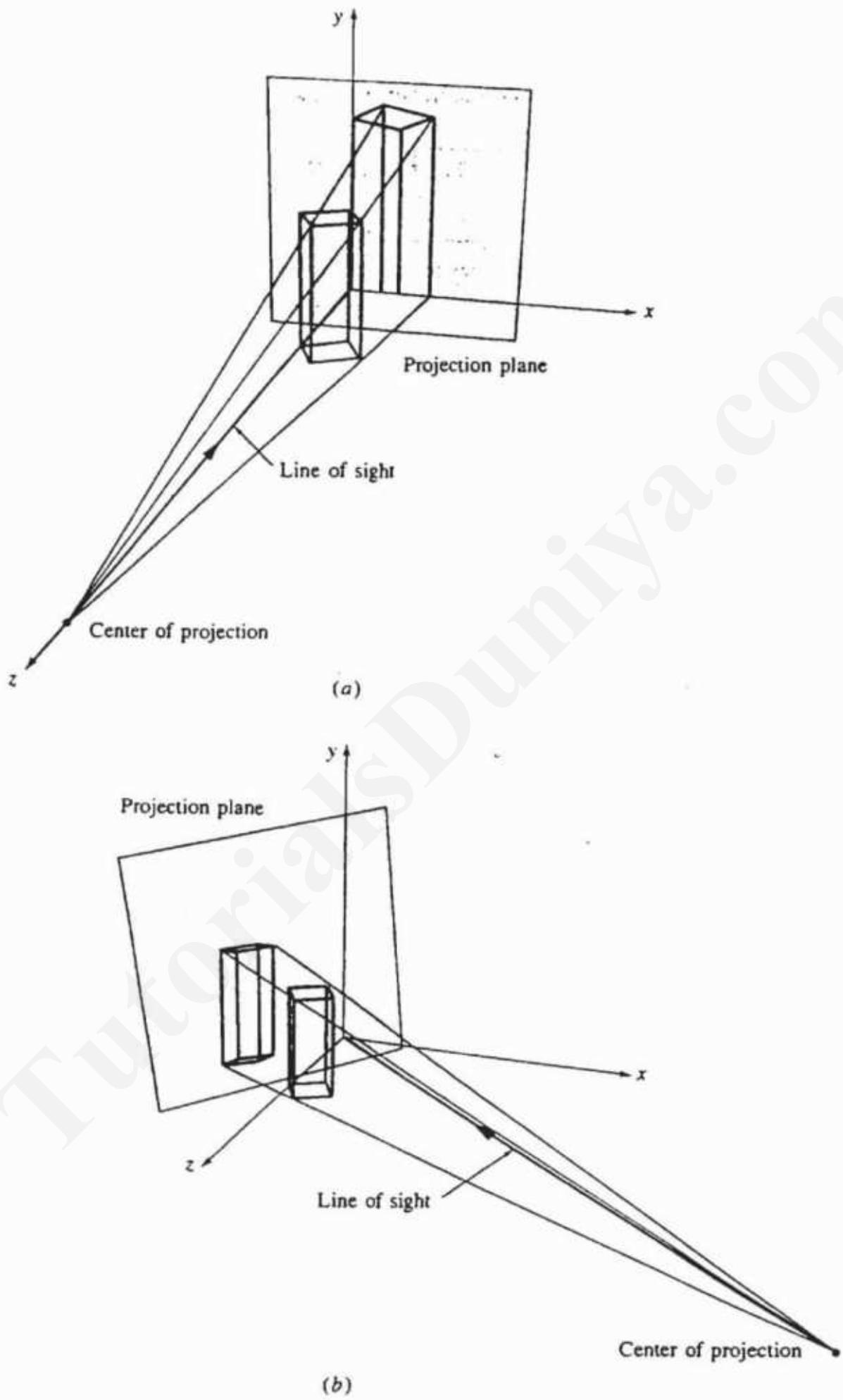
point is fixed and that the plane of projection is perpendicular to each projector as shown in Fig. 3-11a. The object is manipulated to obtain any required view. The second assumes that the object is fixed, that the center of projection is free to move anywhere in three space, and that the plane of projection is not necessarily perpendicular to the viewing direction. An example is shown in Fig. 3-11b. Both approaches are *mathematically equivalent*.

By analogy the first approach is similar to the actions of a human observer when asked to describe a small object, e.g., a book. The object is picked up, rotated and translated in order to view all sides and aspects of the object. The center of projection is fixed and the object is manipulated. The second approach is similar to the actions of the human observer when asked to describe a large object, e.g., an automobile. The observer walks around the object to view the various sides, climbs up on a ladder to view the top, and kneels down to look at the bottom. Here the object is fixed, and the center of projection and eye point are moved.

When designing or viewing an object on a computer graphics display the location of the eye is typically fixed and the plane of projection, i.e., the face of the CRT, is typically perpendicular to the viewing direction. Hence, the first approach is generally more appropriate. However, if the graphics display is used to simulate the motion of a vehicle or of an observer moving through a computer generated model, as is the case for vehicle simulators, or for an observer strolling through an architectured model, then the second approach is more appropriate.

A fixed center of projection, movable object approach is used in this book. The fixed object, movable center of projection approach is nicely developed by Carl bom and Paciorek (Ref. 3-2).

We begin our discussion of plane geometric projections (see Fig. 3-10) by first considering the parallel projections.



**Figure 3-11** Plane projections. (a) Center of projection fixed; (b) object fixed.

## 1.12 ORTHOGRAPHIC PROJECTIONS

The simplest of the parallel projections is the orthographic projection, commonly used for engineering drawings. They accurately show the correct or 'true' size and shape of a single plane face of an object. Orthographic projections are projections onto one of the coordinate planes  $x = 0$ ,  $y = 0$  or  $z = 0$ . The matrix for projection onto the  $z = 0$  plane is

$$[P_z] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-27)$$

Notice that the third column (the  $z$  column) is all zeros. Consequently, the effect of the transformation is to set the  $z$  coordinate of a position vector to zero.

Similarly, the matrices for projection onto the  $x = 0$  and  $y = 0$  planes are

$$[P_x] = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-28)$$

and

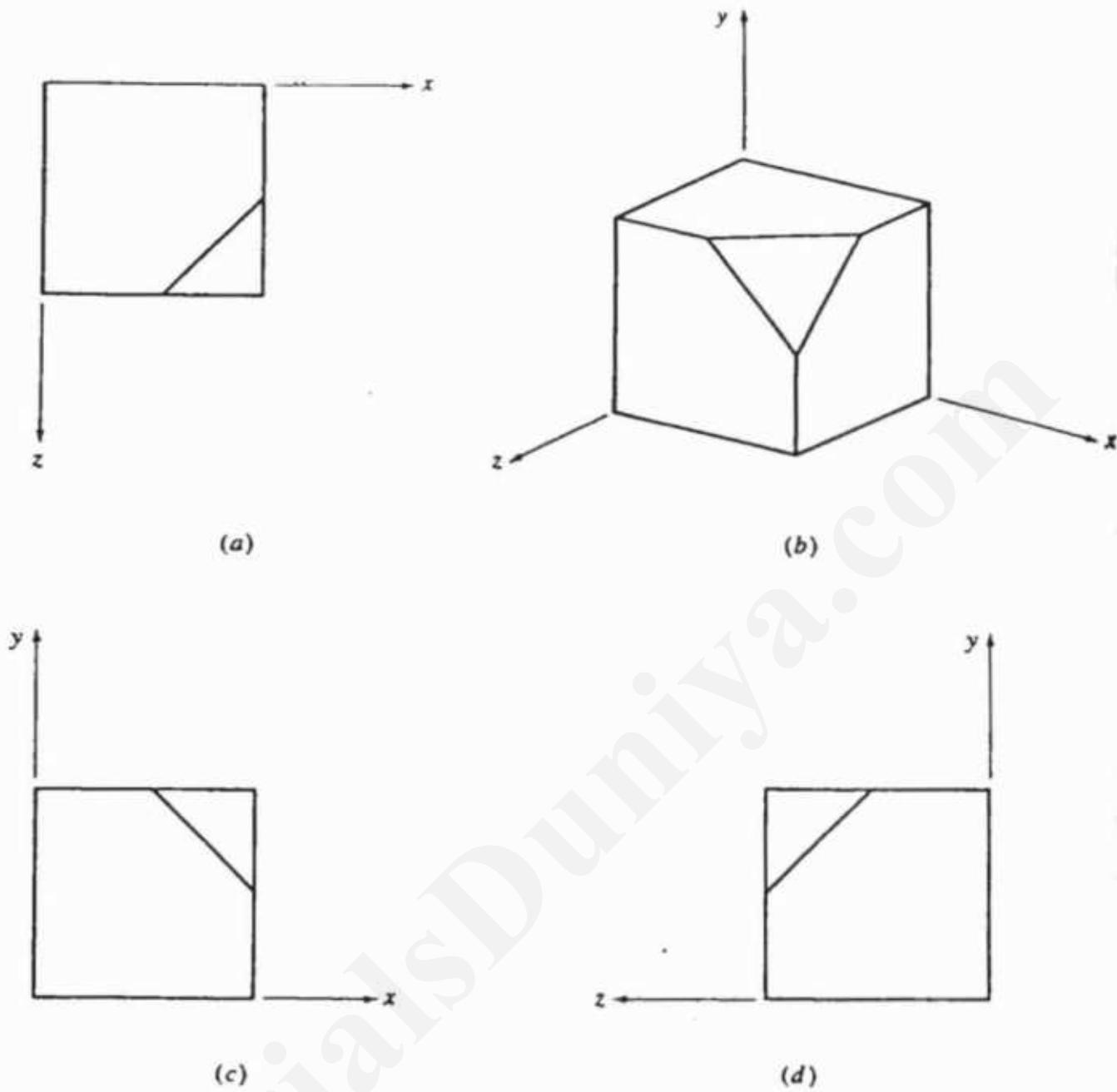
$$[P_y] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-29)$$

Orthographic projections of the object in Fig. 3-12a onto the  $x = 0$ ,  $y = 0$  and  $z = 0$  planes from centers of projection at infinity on the  $+x$ -,  $+y$ - and  $+z$ -axes are shown in Figs. 3-12b, 3-12c and 3-12d respectively.

A single orthographic projection does not provide sufficient information to visually and practically reconstruct the shape of an object. Consequently, multiple orthographic projections are necessary. These multiview orthographic projections are by convention<sup>†</sup> arranged as shown in Fig. 3-13. The front, right side and top views are obtained by projection onto the  $z = 0$ ,  $x = 0$  and  $y = 0$  planes from centers of projection at infinity on the  $+z$ -,  $+x$ - and  $+y$ -axes. The rear, left side and bottom view projections are obtained by projection onto the  $z = 0$ ,  $x = 0$ ,  $y = 0$  planes from centers of projection at infinity on the  $-z$ -,  $-x$ - and  $-y$ -axes. The coordinate axes are not normally shown on the views.

As shown in Fig. 3-13, by convention hidden lines are shown dashed. All six views are normally not required to adequately convey the shape of an object. The front, top and right side views are most frequently used. Even when all six views are not used, the ones that are used appear in the locations shown. The

<sup>†</sup>This is the convention used in the United States.



**Figure 3-12** Orthographic projections onto (b)  $y = 0$ , (c)  $z = 0$  and (d)  $x = 0$  planes.

front and side views are sometimes called the front and side elevations. The top view is sometimes called the plan view.

It is interesting and important to note that all six views can be obtained by combinations of reflection, rotation and translation, followed by projection onto the  $z = 0$  plane from a center of projection at infinity on the  $+z$ -axis. For example, the rear view is obtained by reflection through the  $z = 0$  plane, followed by projection onto the  $z = 0$  plane. Similarly, the left side view is obtained by rotation about the  $y$ -axis by  $+90^\circ$ , followed by projection onto the  $z = 0$  plane.

For objects with planes that are not parallel to one of the coordinate planes, the standard orthographic views do not show the correct or true shape of these planes. Auxiliary views are used for this purpose. An auxiliary view is formed by rotating and translating the object so that the normal to the auxiliary plane

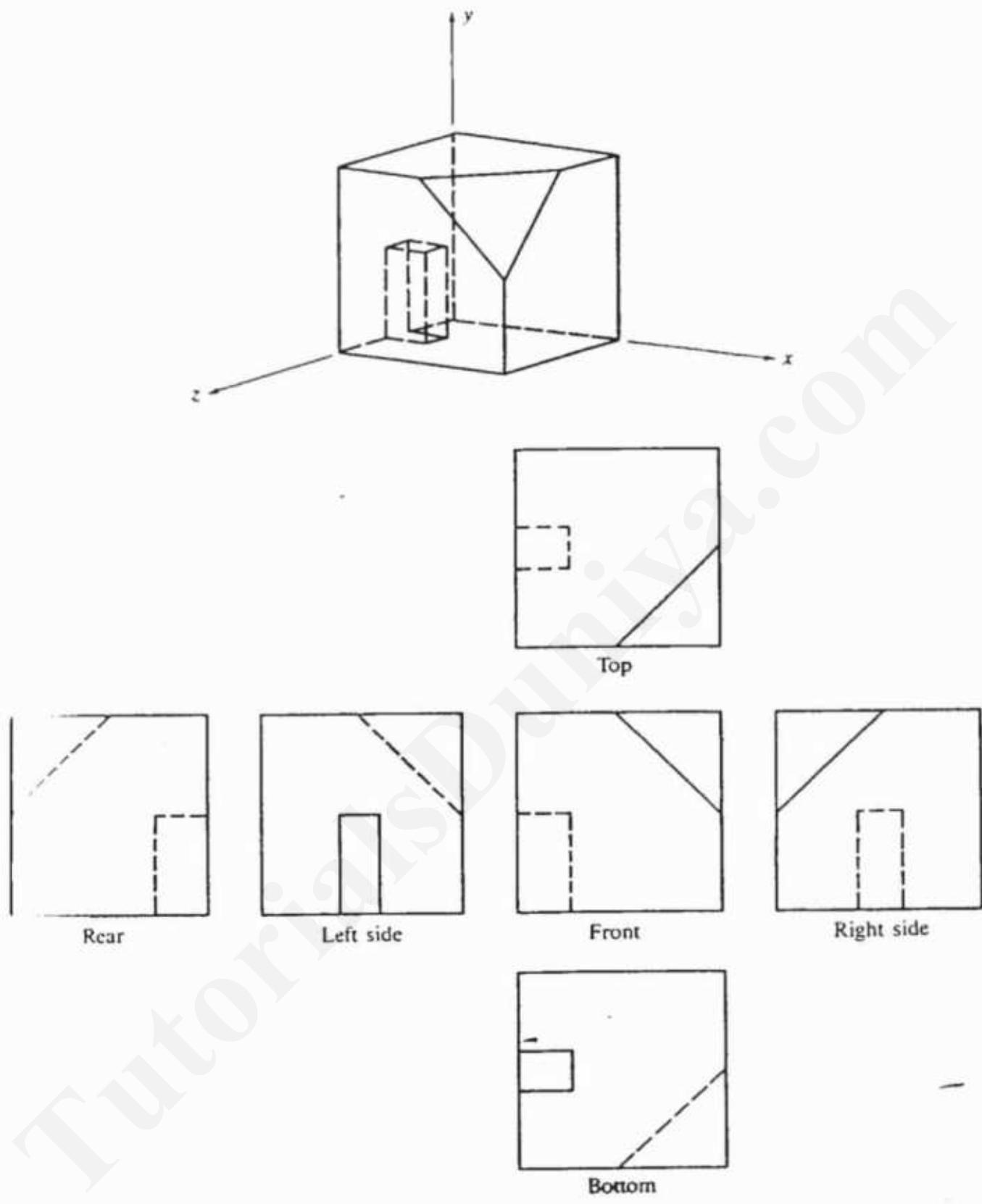


Figure 3-13 Multiview orthographic projection.

is coincident with one of the coordinate axes (see Sec. 3-9). The result is then projected onto the coordinate plane perpendicular to that axis. Figure 3-14c shows an auxiliary plane illustrating the true shape of the triangular corner of the block shown in Fig. 3-13.

An example more fully illustrates these constructions.

### Example 3-12 Auxiliary View

Develop an auxiliary view showing the true shape of the triangular corner for the object shown in Fig. 3-14a. The position vectors for the object are

$$[X] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0.5 & 1 & 1 \\ 0.5 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0.5 & 1 \\ 0 & 0 & 0.6 & 1 \\ 0.25 & 0 & 0.6 & 1 \\ 0.25 & 0.5 & 0.6 & 1 \\ 0 & 0.5 & 0.6 & 1 \\ 0 & 0 & 0.4 & 1 \\ 0.25 & 0 & 0.4 & 1 \\ 0.25 & 0.5 & 0.4 & 1 \\ 0 & 0.5 & 0.4 & 1 \end{bmatrix}$$

The vertex numbers shown in Fig. 3-14 correspond to the rows in the position vector matrix  $[X]$ .

The unit outward normal to the triangular face has direction cosines

$$[c_x \ c_y \ c_z] = [1/\sqrt{3} \ 1/\sqrt{3} \ 1/\sqrt{3}]$$

and passes through the origin and the point  $[0.83333 \ 0.83333 \ 0.83333]$ . Recalling the results of Sec. 3-9 and Ex. 3-10, the normal is made coincident with the  $z$ -axis by rotation about the  $x$ -axis by an angle

$$\alpha = \cos^{-1}(c_z/d) = \cos^{-1}(1/\sqrt{2}) = +45^\circ$$

followed by rotation about the  $y$ -axis by an angle

$$\beta = \cos^{-1}(d) = \cos^{-1}(2/\sqrt{6}) = +35.26^\circ$$

Here, the concatenated transformation matrix is

$$[T] = \begin{bmatrix} 2/\sqrt{6} & 0 & 1/\sqrt{3} & 0 \\ -1/\sqrt{6} & 1/\sqrt{2} & 1/\sqrt{3} & 0 \\ -1/\sqrt{6} & -1/\sqrt{2} & 1/\sqrt{3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The transformed position vectors are

$$[X'] = [X][T] = \begin{bmatrix} -0.408 & -0.707 & 0.577 & 1 \\ 0.408 & -0.707 & 1.155 & 1 \\ 0.204 & -0.354 & 1.443 & 1 \\ -0.408 & 0 & 1.443 & 1 \\ -0.816 & 0 & 1.155 & 1 \\ 0 & 0 & 0 & 1 \\ 0.816 & 0 & 0.577 & 1 \\ 0.408 & 0.707 & 1.155 & 1 \\ -0.408 & 0.707 & 0.577 & 1 \\ 0.204 & 0.354 & 1.443 & 1 \\ -0.245 & -0.424 & 0.354 & 1 \\ -0.041 & -0.424 & 0.491 & 1 \\ -0.245 & -0.071 & 0.779 & 1 \\ -0.449 & -0.071 & 0.635 & 1 \\ -0.163 & -0.283 & 0.231 & 1 \\ 0.041 & -0.283 & 0.375 & 1 \\ -0.163 & 0.071 & 0.664 & 1 \\ -0.367 & 0.071 & 0.52 & 1 \end{bmatrix}$$

The result is shown in Fig. 3-14b. The auxiliary view is created by projecting this intermediate result onto the  $z = 0$  plane using Eq. (3-27), i.e.,

$$[P_z] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The transformation matrices  $[T]$  and  $[P_z]$  are concatenated to yield

$$[T'] = [T][P_z] = \begin{bmatrix} 2/\sqrt{6} & 0 & 0 & 0 \\ -1/\sqrt{6} & 1/\sqrt{2} & 0 & 0 \\ -1/\sqrt{3} & -1/\sqrt{2} & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

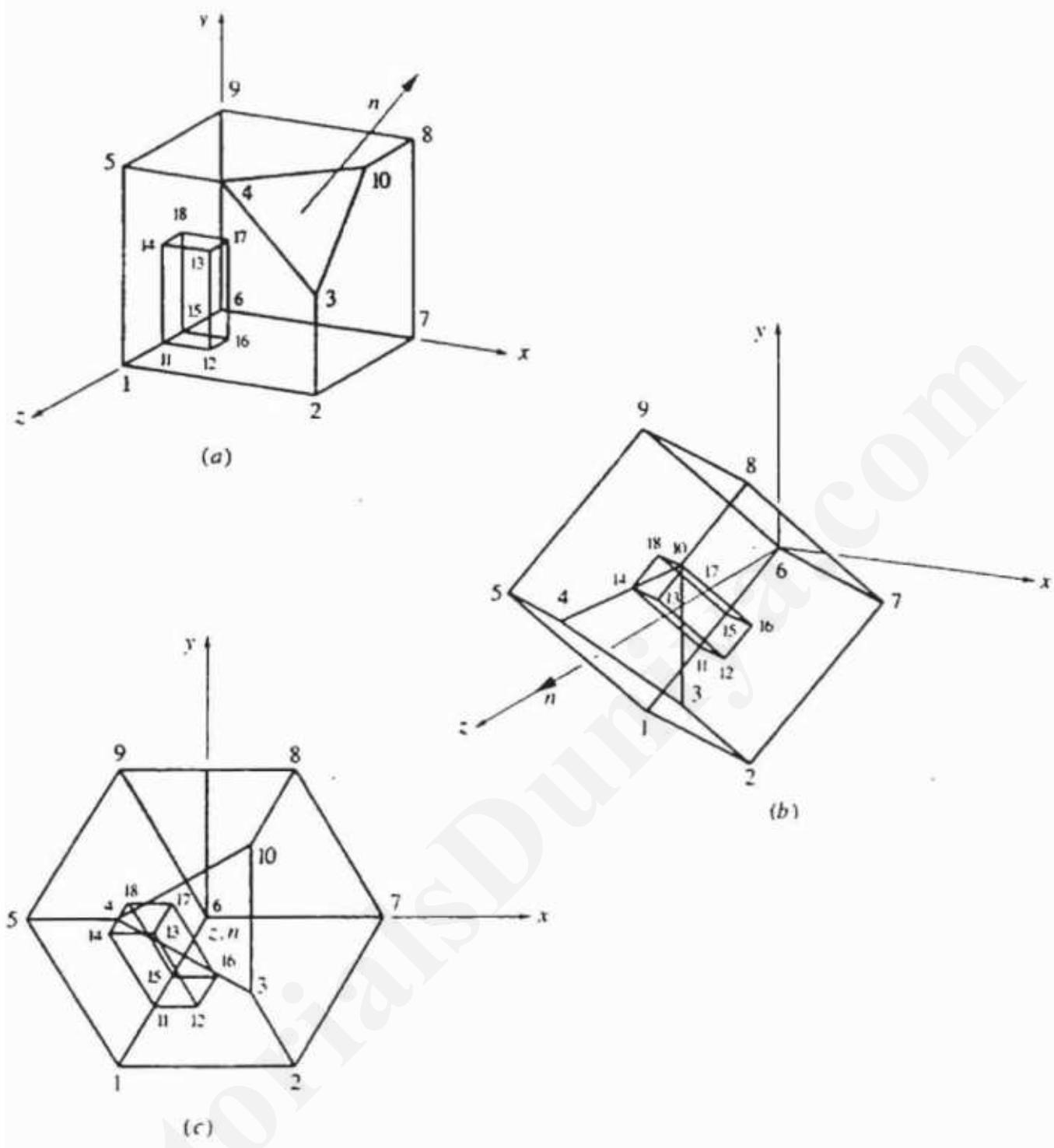
Notice the column of zeros. The auxiliary view is then created by

$$[X''] = [X][T']$$

$[X'']$  is the same as  $[X']$  except that the third column is all zeros, i.e., the effect of the projection is to neglect the  $z$  coordinate. The result is shown in Fig. 3-14c. Hidden lines are shown solid. Notice that the true shape of the triangle, which is equilateral, is shown.

---

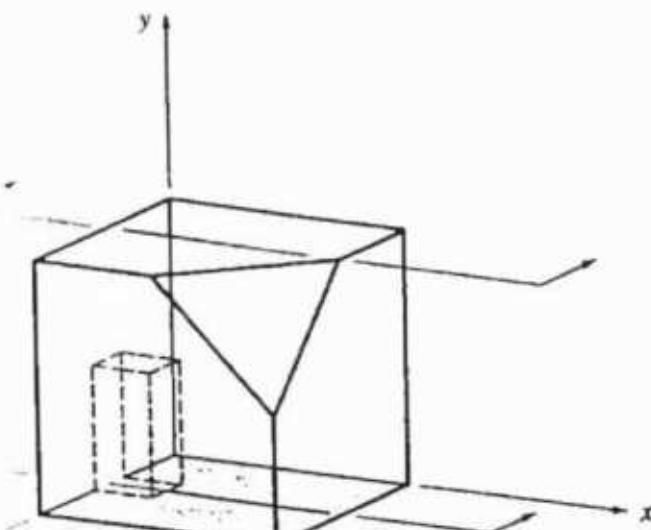
For complex objects it is frequently necessary to show details of the interior. This is accomplished using a sectional view. A sectional view is constructed by passing a plane, called the section or 'cutting' plane, through the object,



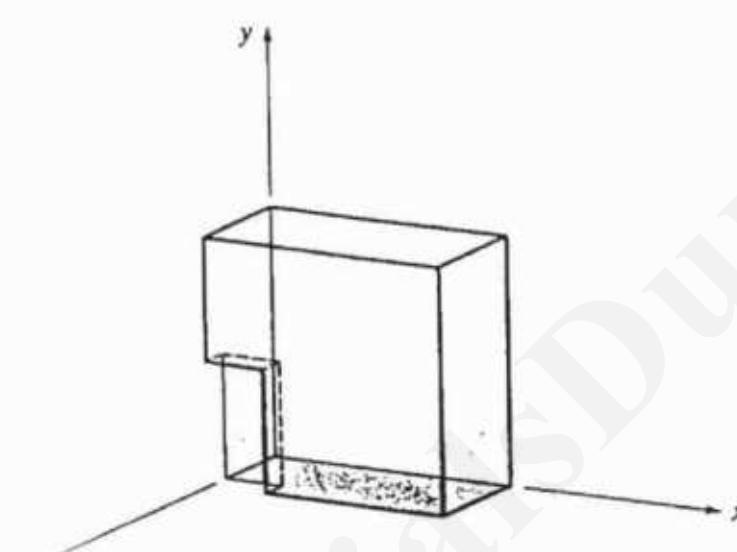
**Figure 3-14** Development of an auxiliary view. (a) Trimetric view; (b) normal coincident with the  $z$ -axis; (c) projected onto the  $z = 0$  plane.

removing the part of the object on one side of the plane and projecting the remainder onto the section plane. Again, a sectional view can be constructed by making the normal to the section plane coincident with one of the coordinate axes (see Sec. 3-9), clipping the object to one side of the section plane (see Ref. 3-1), and finally projecting the result onto the coordinate plane perpendicular to the axis.

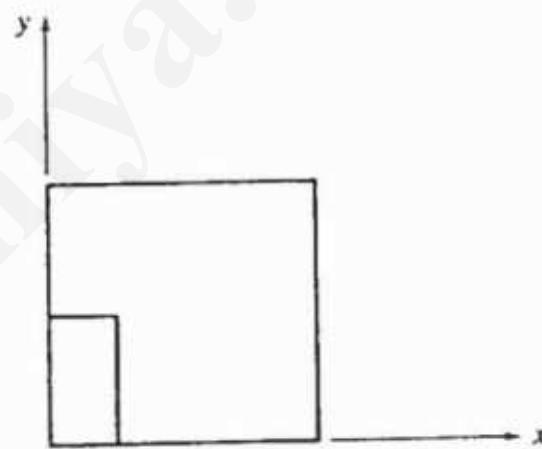
Figure 3-15 shows a section plane passing through the notch on the left side of the object of Fig. 3-13. The arrows are used to show the section plane and the viewing direction.



(a)



(b)



(c)

**Figure 3-15** Development of a sectional view. (a) Complete object; (b) portion between the section plane and the center of projection removed; (c) projected onto the  $z = 0$  plane.

### 3-13 AXONOMETRIC PROJECTIONS

A single orthographic projection fails to illustrate the general three-dimensional shape of an object. Axonometric projections overcome this limitation. An axonometric projection is constructed by manipulating the object, using rotations and translations, such that at least three adjacent faces are shown†. The result is then projected from a center of projection at infinity onto one of the coordinate planes, usually the  $z = 0$  plane. Unless a face is parallel to the plane of projection, an axonometric projection does not show its true shape. However,

†The minimal number of faces occurs for simple cuboidal objects such as are used in most illustrations in this chapter.

the relative lengths of originally parallel lines remain constant, i.e., parallel lines are equally foreshortened. The foreshortening factor is the ratio of the projected length of a line to its true length. There are three axonometric projections of interest: trimetric, dimetric, and isometric, as shown in Fig. 3-10. The trimetric projection is the least restrictive and the isometric projection the most restrictive. In fact, as shown below, an isometric projection is a special case of a dimetric projection, and a dimetric projection is a special case of a trimetric projection.

A trimetric projection is formed by arbitrary rotations, in arbitrary order, about any or all of the coordinate axes, followed by parallel projection onto the  $z = 0$  plane. Most of the illustrations in this book are trimetric projections. Figure 3-16 shows several different trimetric projections. Each projection was formed by first rotating about the  $y$ -axis and then about the  $x$ -axis, followed by parallel projection onto the  $z = 0$  plane.

The foreshortening ratios for each projected principal axis ( $x$ ,  $y$  and  $z$ ) are all different in a general trimetric projection. Here, a principal axis is used in the sense of an axis or edge of the object originally parallel to one of the  $x$ ,  $y$  or  $z$  coordinate axes. The wide variety of trimetric projections precludes giving a general equation for these ratios. However, for any specific trimetric projection, the foreshortening ratios are obtained by applying the concatenated transformation matrix to the unit vectors along the principal axes. Specifically,

$$\begin{aligned} [U][T] &= \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} [T] \\ &= \begin{bmatrix} x_x^* & y_x^* & 0 & 1 \\ x_y^* & y_y^* & 0 & 1 \\ x_z^* & y_z^* & 0 & 1 \end{bmatrix} \quad (3-30) \end{aligned}$$

where  $[U]$  is the matrix of unit vectors along the untransformed  $x$ ,  $y$  and  $z$  axes, respectively, and  $[T]$  is the concatenated trimetric projection matrix. The foreshortening factors along the projected principal axes are then

$$f_x = \sqrt{x_x^{*2} + y_x^{*2}} \quad (3-31a)$$

$$f_y = \sqrt{x_y^{*2} + y_y^{*2}} \quad (3-31b)$$

$$f_z = \sqrt{x_z^{*2} + y_z^{*2}} \quad (3-31c)$$

Example 3-13 provides the details of a trimetric projection.

### Example 3-13 Trimetric Projection

Consider the center illustration of Fig. 3-16 formed by a  $\phi = 30^\circ$  rotation about the  $y$ -axis, followed by a  $\theta = 45^\circ$  rotation about the  $x$ -axis, and then

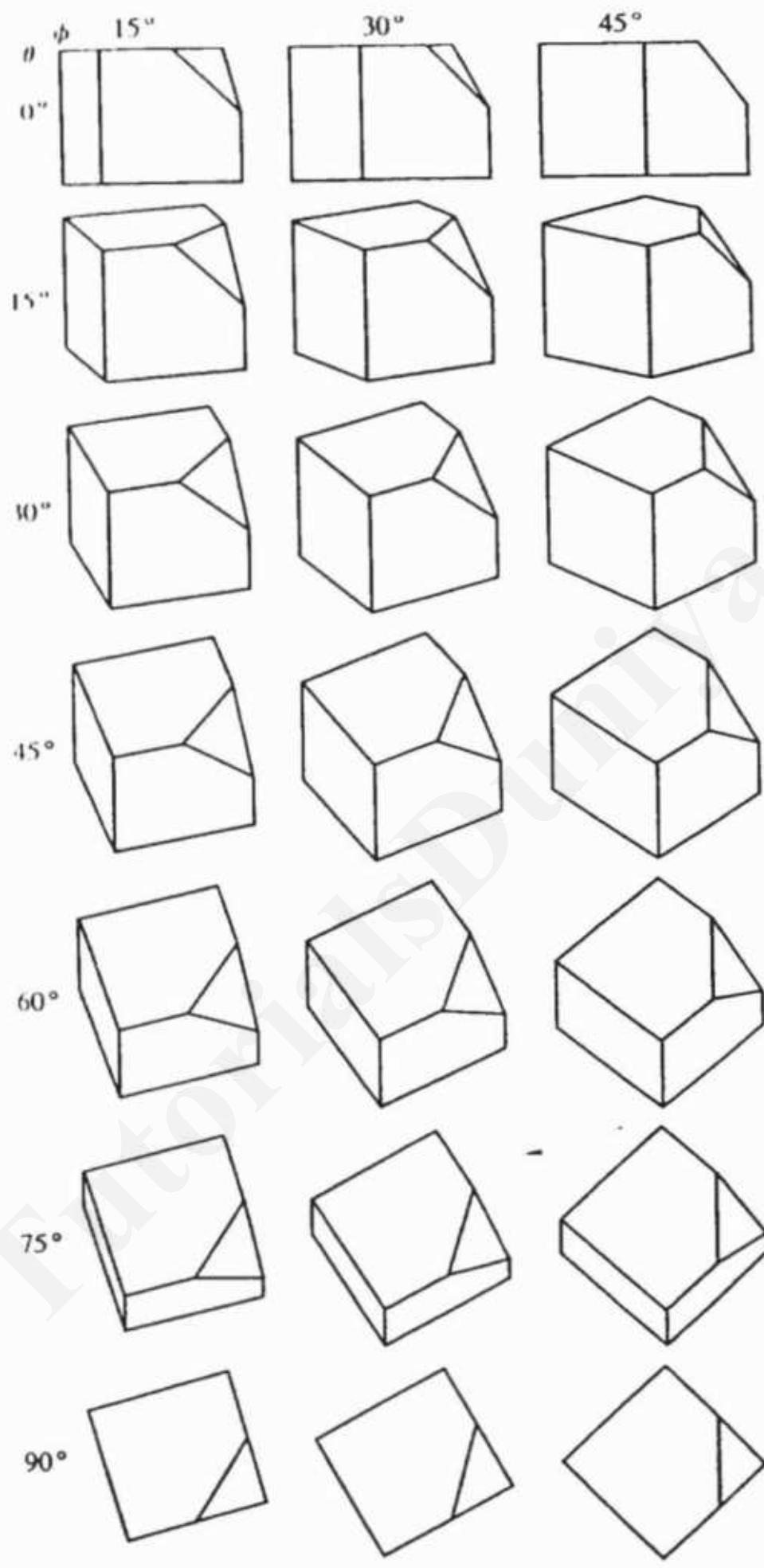


Figure 3-16 Trimetric projections.

parallel projection onto the  $z = 0$  plane. The position vectors for the cube with one corner removed are

$$[X] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0.5 & 1 & 1 \\ 0.5 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0.5 & 1 \end{bmatrix}$$

The concatenated trimetric projection is (see Eqs. 3-8, 3-6, and 3-27)

$$\begin{aligned} [T] &= [R_y][R_x][P_z] \\ &= \begin{bmatrix} \cos\phi & 0 & -\sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos\phi & \sin\phi \sin\theta & 0 & 0 \\ 0 & \cos\theta & 0 & 0 \\ \sin\phi & -\cos\phi \sin\theta & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \sqrt{3}/2 & \sqrt{2}/4 & 0 & 0 \\ 0 & \sqrt{2}/2 & 0 & 0 \\ 1/2 & -\sqrt{6}/4 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Thus, the transformed position vectors are

$$[X^*] = [X][T] = \begin{bmatrix} 0.5 & -0.612 & 0 & 1 \\ 1.366 & -0.259 & 0 & 1 \\ 1.366 & 0.095 & 0 & 1 \\ 0.933 & 0.272 & 0 & 1 \\ 0.5 & 0.095 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0.866 & 0.354 & 0 & 1 \\ 0.866 & 1.061 & 0 & 1 \\ 0 & 0.707 & 0 & 1 \\ 1.116 & 0.754 & 0 & 1 \end{bmatrix}$$

The foreshortening ratios are

$$\begin{aligned} [U][T] &= \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} \sqrt{3}/2 & \sqrt{2}/4 & 0 & 0 \\ 0 & \sqrt{2}/2 & 0 & 0 \\ 1/2 & -\sqrt{6}/4 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \sqrt{3}/2 & \sqrt{2}/4 & 0 & 1 \\ 0 & \sqrt{2}/2 & 0 & 1 \\ 1/2 & -\sqrt{6}/4 & 0 & 1 \end{bmatrix} \end{aligned}$$

and

$$f_z = \sqrt{(\sqrt{3}/2)^2 + (\sqrt{2}/4)^2} = 0.935$$

$$f_y = \sqrt{2}/2 = 0.707$$

$$f_x = \sqrt{(1/2)^2 + (-\sqrt{6}/4)^2} = 0.791$$

A dimetric projection is a trimetric projection with two of the three foreshortening factors equal; the third is arbitrary. A dimetric projection is constructed by a rotation about the  $y$ -axis through an angle  $\phi$  followed by rotation about the  $x$ -axis through an angle  $\theta$  and projection from a center of projection at infinity onto the  $z = 0$  plane. The specific rotation angles are as yet unknown. Using Eqs. (3-8), (3-6) and (3-27), the resulting transformation is

$$[T] = [R_y][R_x][P_z]$$

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

Concatenation yields

$$[T] = \begin{bmatrix} \cos \phi & \sin \phi \sin \theta & 0 & 0 \\ 0 & \cos \theta & 0 & 0 \\ \sin \phi & -\cos \phi \sin \theta & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-32)$$

The unit vectors on the  $x$ ,  $y$ , and  $z$  principal axes transform to

$$[U^*] = [U][T] = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} \cos \phi & \sin \phi \sin \theta & 0 & 0 \\ 0 & \cos \theta & 0 & 0 \\ \sin \phi & -\cos \phi \sin \theta & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} -$$

$$[U^*] = \begin{bmatrix} \cos \phi & \sin \phi \sin \theta & 0 & 1 \\ 0 & \cos \theta & 0 & 1 \\ \sin \phi & -\cos \phi \sin \theta & 0 & 1 \end{bmatrix} \quad (3-33)$$

The square of the length of the original unit vector along the  $x$ -axis, i.e., the square of the foreshortening factor, is now

$$f_x^2 = (x_x^{*2} + y_x^{*2}) = \cos^2 \phi + \sin^2 \phi \sin^2 \theta \quad (3-34)$$

Similarly, the squares of the lengths of the original unit vectors along the  $y$ -

and  $z$ -axes are given by

$$f_y^2 = (x_y^{*2} + y_y^{*2}) = \cos^2 \theta \quad (3 - 35)$$

$$f_z^2 = (x_z^{*2} + y_z^{*2}) = \sin^2 \phi + \cos^2 \phi \sin^2 \theta \quad (3 - 36)$$

Equating the foreshortening factors along the  $x$  and  $y$  principal axes<sup>†</sup> yields one equation in the two unknown rotation angles  $\phi$  and  $\theta$ . Specifically,

$$\cos^2 \phi + \sin^2 \phi \sin^2 \theta = \cos^2 \theta$$

Using the identities  $\cos^2 \phi = 1 - \sin^2 \phi$  and  $\cos^2 \theta = 1 - \sin^2 \theta$  yields

$$\sin^2 \phi = \frac{\sin^2 \theta}{1 - \sin^2 \theta} \quad (3 - 37)$$

A second relation between  $\phi$  and  $\theta$  is obtained by choosing the foreshortening factor along the  $z$  principal axis  $f_z$ . Combining Eqs. (3-36) and (3-37) using  $\cos^2 \phi = 1 - \sin^2 \phi$  yields

$$2 \sin^2 \theta - 2 \sin^4 \theta - (1 - \sin^2 \theta) f_z^2 = 0$$

$$\text{or} \quad 2 \sin^4 \theta - (2 + f_z^2) \sin^2 \theta + f_z^2 = 0 \quad (3 - 38)$$

After letting  $u = \sin^2 \theta$ , solution yields

$$\sin^2 \theta = f_z^2 / 2, 1$$

Since the  $\sin^2 \theta = 1$  solution yields an infinite result when substituted into Eq. (3-37), it is discarded. Hence,

$$\theta = \sin^{-1} \left( \pm f_z / \sqrt{2} \right) \quad (3 - 39)$$

Substituting into Eq. (3-37) yields

$$\phi = \sin^{-1} \left( \pm f_z / \sqrt{2 - f_z^2} \right) \quad (3 - 40)$$

This result shows that the range of foreshortening factors is  $0 \leq f_z \leq 1$ .<sup>‡</sup> Further, note that each foreshortening factor  $f_z$  yields four possible dimetric projections.

Figure 3-17 shows dimetric projections for various foreshortening factors. For each foreshortening factor, the dimetric projection corresponding to a positive rotation about the  $y$ -axis followed by a positive rotation about the  $x$ -axis was chosen.

<sup>†</sup>Any two of the three principal axes could have been used.

<sup>‡</sup>Negative foreshortening factors are not sensible.

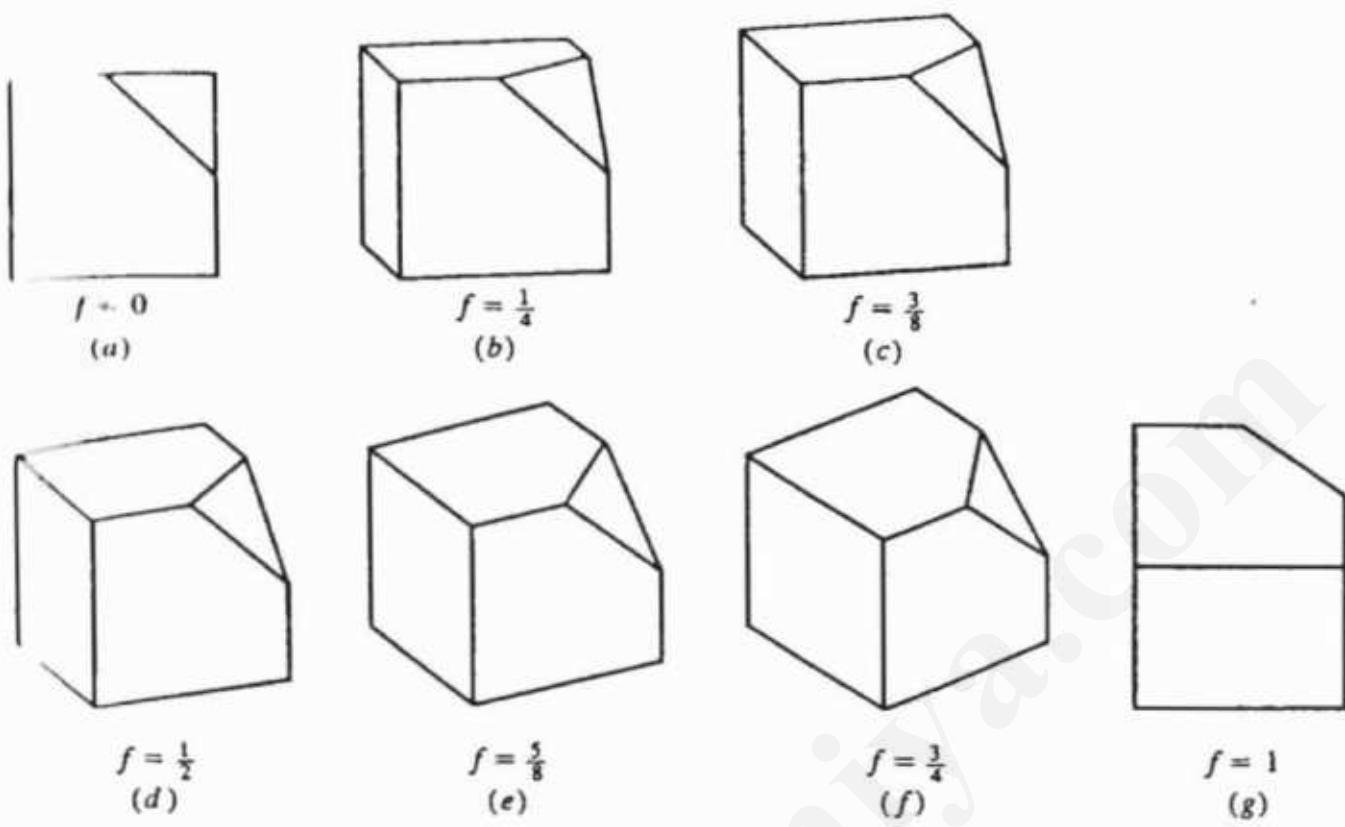


Figure 3-17 Dimetric projections for various foreshortening factors. (a) 0; (b) 1/4; (c) 3/8; (d) 1/2; (e) 5/8; (f) 3/4; (g) 1.

Figure 3-18 shows the four possible dimetric projections for a foreshortening factor of 5/8.

An example illustrates specific results.

#### Example 3-14 Dimetric Projections

For the cube with the corner cut off, determine the dimetric projection for a foreshortening factor along the  $z$ -axis of 1/2.

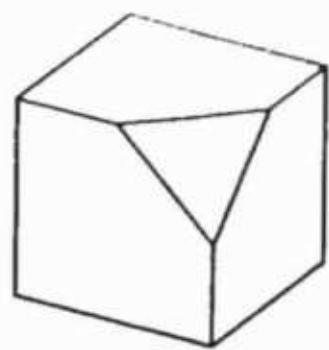
From Eq. (3-39)

$$\begin{aligned}\theta &= \sin^{-1} \left( \pm f_z / \sqrt{2} \right) \\ &= \sin^{-1} \left( \pm 1/2\sqrt{2} \right) \\ &= \sin^{-1}(\pm 0.35355) \\ &= \pm 20.705^\circ\end{aligned}$$

From Eq. (3-40)

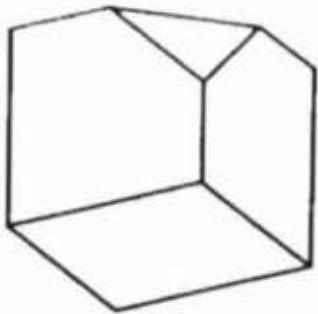
$$\begin{aligned}\phi &= \sin^{-1} \left( \pm f_z / \sqrt{2 - f_z^2} \right) \\ &= \sin^{-1} \left( \pm 1/2 / \sqrt{7/4} \right) \\ &= \sin^{-1}(\pm 0.378) \\ &= \pm 22.208^\circ\end{aligned}$$

Choosing  $\phi = +22.208^\circ$  and  $\theta = +20.705^\circ$ , Eq. (3-32) yields the dimetric



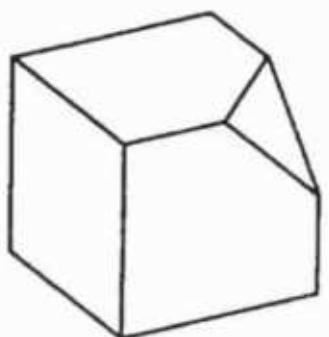
$\phi < 0, \theta > 0$

(a)



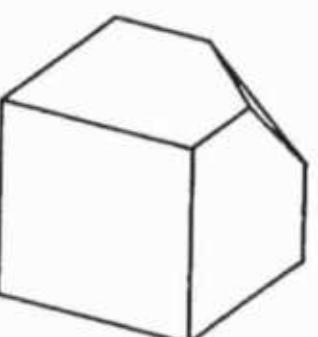
$\phi < 0, \theta > 0$

(b)



$\phi < 0, \theta > 0$

(c)



$\phi < 0, \theta > 0$

(d)

**Figure 3-18** Four possible dimetric projections for a foreshortening factor of  $5/8$  and rotation angles  $\phi = \pm 29.52^\circ, \theta = \pm 26.23^\circ$ . (a)  $\phi = -29.52^\circ, \theta = +26.23^\circ$ ; (b)  $\phi = -29.52^\circ, \theta = -26.23^\circ$ ; (c)  $\phi = +29.52^\circ, \theta = +26.23^\circ$ ; (d)  $\phi = +29.52^\circ, \theta = -26.23^\circ$ .

projection matrix

$$[T] = \begin{bmatrix} 0.926 & 0.134 & 0 & 0 \\ 0 & 0.935 & 0 & 0 \\ 0.378 & -0.327 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Recalling the position vectors for the cube with the corner cut off  $[X]$  (see Ex. 3-13), the transformed position vectors are

$$[X^*] = [X][T] = \begin{bmatrix} 0.378 & -0.327 & 0 & 1 \\ 1.304 & -0.194 & 0 & 1 \\ 1.304 & 0.274 & 0 & 1 \\ 0.841 & 0.675 & 0 & 1 \\ 0.378 & 0.608 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0.926 & 0.134 & 0 & 1 \\ 0.926 & 1.069 & 0 & 1 \\ 0 & 0.935 & 0 & 1 \\ 1.115 & 0.905 & 0 & 1 \end{bmatrix}$$

The result is shown in Fig. 3-17d.

A dimetric projection allows two of the three transformed principal axes to be measured with the same scale factor. Measurements along the third transformed principal axis require a different scale factor. If accurate scaling of the dimensions of the projected object is required, this can lead to both confusion and error. An isometric projection eliminates this problem.

In an isometric projection all three foreshortening factors are equal. Recalling Eqs. (3-34) to (3-36) and equating Eqs. (3-34) and (3-35) again yields Eq. (3-37), i.e.,

$$\sin^2 \phi = \frac{\sin^2 \theta}{1 - \sin^2 \theta} \quad (3-37)$$

Equating Eqs. (3-35) and (3-36) yields

$$\sin^2 \phi = \frac{1 - 2 \sin^2 \theta}{1 - \sin^2 \theta} \quad (3-41)$$

From Eqs. (3-37) and (3-41) it follows that  $\sin^2 \theta = 1/3$  or  $\sin \theta = \pm \sqrt{1/3}$  and  $\theta = 135.26^\circ$ . Then

$$\sin^2 \phi = \frac{1/3}{1 - 1/3} = 1/2$$

and  $\phi = \pm 45^\circ$ . Again note that there are four possible isometric projections. These are shown in Fig. 3-19. The foreshortening factor for an isometric projection is (see Eq. 3-35)

$$f = \sqrt{\cos^2 \theta} = \sqrt{2/3} = 0.8165$$

In fact, an isometric projection is a special case of a dimetric projection with  $f_1 = 0.8165$ .

The angle that the projected  $x$ -axis makes with the horizontal is important in manual construction of isometric projections. Transforming the unit vector along the  $x$ -axis using the isometric projection matrix yields

$$[U_x^*] = [1 \ 0 \ 0 \ 1] \begin{bmatrix} \cos \phi & \sin \phi \sin \theta & 0 & 0 \\ 0 & \cos \theta & 0 & 0 \\ \sin \phi & -\cos \phi \sin \theta & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ = [\cos \phi \ \sin \phi \sin \theta \ 0 \ 1]$$

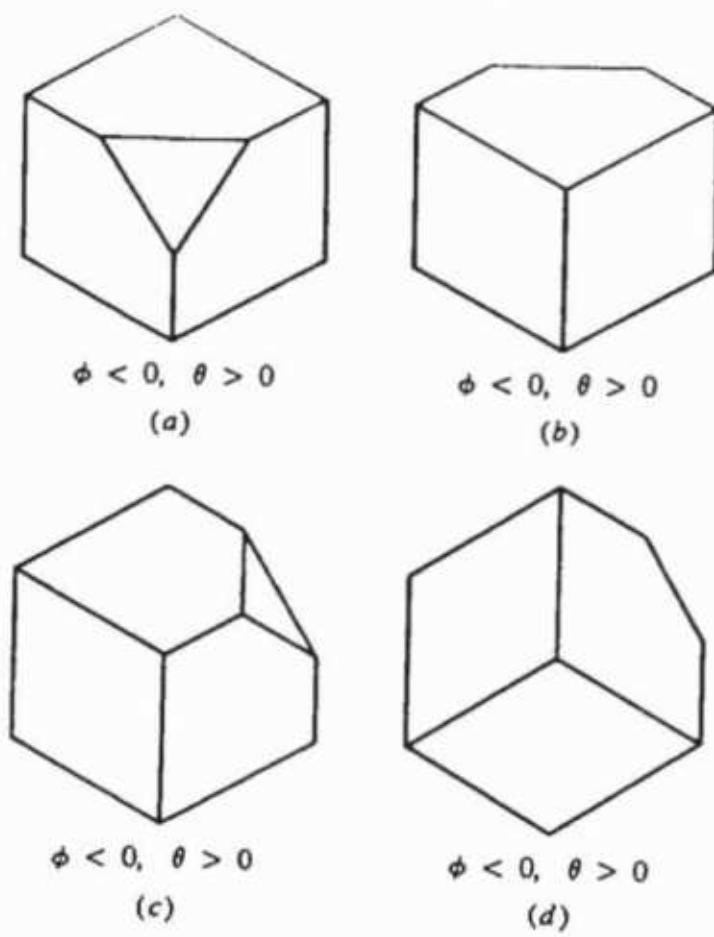
The angle between the projected  $x$ -axis and the horizontal is then

$$\tan \alpha = \frac{y_x^*}{x_x^*} = \frac{\sin \phi \sin \theta}{\cos \phi} = \pm \sin \theta \quad (3-42)$$

Since  $\sin \phi = \cos \phi$  for  $\phi = 45^\circ$ . Alpha is then

$$\alpha = \tan^{-1}(\pm \sin 35.26439^\circ) = \pm 30^\circ$$

A plastic right triangle with included angles of  $30^\circ$  and  $60^\circ$  is a commonly used tool for manually constructing isometric projections. An example illustrates the details.



**Figure 3-19** Four possible isometric projections with rotation angles  $\phi = \pm 45^\circ$ ,  $\theta = \pm 35.26^\circ$ . (a)  $\phi = -45^\circ$ ,  $\theta = +35.26^\circ$ ; (b)  $\phi = -45^\circ$ ,  $\theta = -35.26^\circ$ ; (c)  $\phi = +45^\circ$ ,  $\theta = +35.26^\circ$ ; (d)  $\phi = +45^\circ$ ,  $\theta = -35.26^\circ$ .

### Example 3-15 Isometric Projection

Again considering the cube with the corner cut off (see Ex. 3-13), determine the isometric projection for  $\phi = -45^\circ$  and  $\theta = +35.26439^\circ$ . From Eq. (3-32) the isometric projection transformation is

$$[T] = \begin{bmatrix} 0.707 & -0.408 & 0 & 0 \\ 0 & 0.816 & 0 & 0 \\ -0.707 & -0.408 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Recalling the position vectors  $[X]$ , the transformed position vectors are

$$[X^*] = [X][T] = \begin{bmatrix} -0.707 & -0.408 & 0 & 1 \\ 0 & -0.816 & 0 & 1 \\ 0 & -0.408 & 0 & 1 \\ -0.354 & 0.204 & 0 & 1 \\ -0.707 & 0.408 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0.707 & -0.408 & 0 & 1 \\ 0.707 & 0.408 & 0 & 1 \\ 0 & 0.816 & 0 & 1 \\ 0.354 & 0.204 & 0 & 1 \end{bmatrix}$$

The result is shown in Fig. 3-19a.

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

## 1.14 OBLIQUE PROJECTIONS

In contrast to the orthographic and axonometric projections for which the projectors are perpendicular to the plane of projection, an oblique projection is formed by parallel projectors from a center of projection at infinity that intersect the plane of projection at an oblique angle. The general scheme is shown in Fig. 3-20.

Oblique projections illustrate the general three-dimensional shape of the object. However, only faces of the object parallel to the plane of projection are shown at their true size and shape, i.e., angles and lengths are preserved for these faces only. In fact, the oblique projection of these faces is equivalent to an orthographic front view. Faces not parallel to the plane of projection are distorted.

Two oblique projections, cavalier and cabinet, are of particular interest. A cavalier projection is obtained when the angle between the oblique projectors and the plane of projection is  $45^\circ$ . In a cavalier projection the foreshortening factors for all three principal directions are equal. The resulting figure appears too thick. A cabinet projection is used to 'correct' this deficiency.

An oblique projection for which the foreshortening factor for edges perpendicular to the plane of projection is one-half is called a cabinet projection. As is shown below, for a cabinet projection the angle between the projectors and the plane of projection is  $\cot^{-1}(1/2) = 63.43^\circ$ .

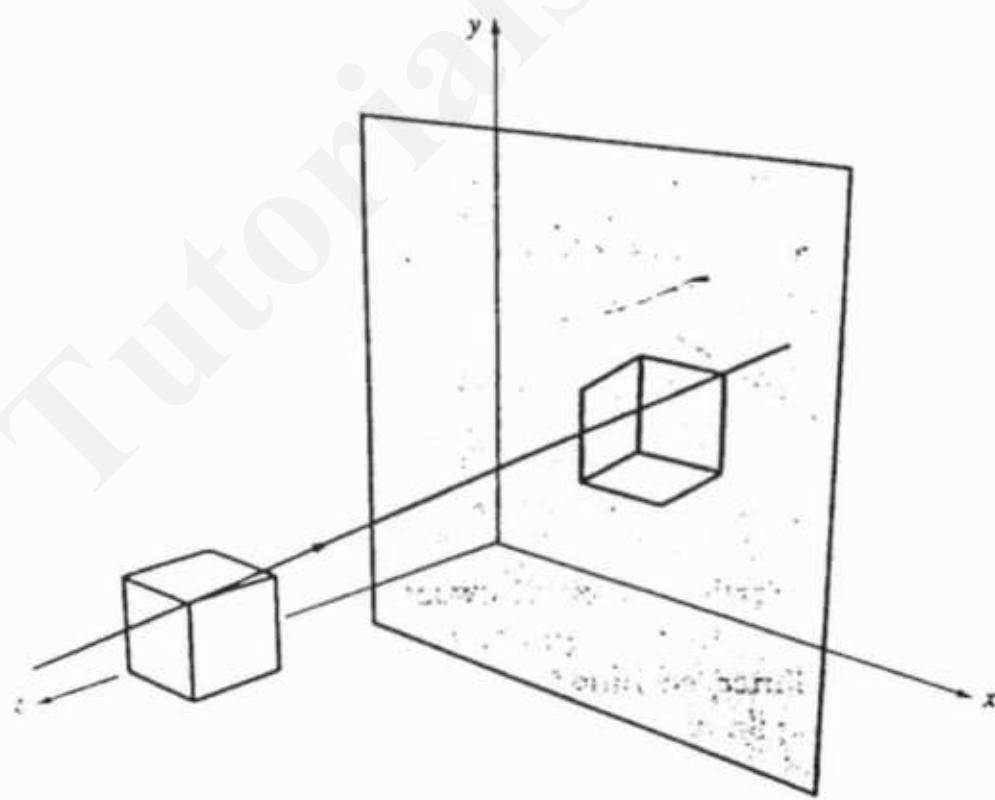


Figure 3-20 Oblique projection.

To develop the transformation matrix for an oblique projection, consider the unit vector  $[0 \ 0 \ 1]$  along the  $z$ -axis shown in Fig. 3-21. For an orthographic or axonometric projection onto the  $z = 0$  plane the vector  $PO$  gives the direction of projection. For an oblique projection, the projectors make an angle with the plane of projection. Typical oblique projectors,  $P_1O$  and  $PP_2$ , are shown in Fig. 3-21.  $P_1O$  and  $PP_2$  make an angle  $\beta$  with the plane of projection  $z = 0$ . Note that all possible projectors through  $P$  or  $O$  making an angle  $\beta$  with the  $z = 0$  plane lie on the surface of a cone with apex at  $P$  or  $O$ . Thus, there are an infinite number of oblique projections for a given angle  $\beta$ .

The projector  $P_1O$  can be obtained from  $PO$  by translating the point  $P$  to the point  $P_1$  at  $[-a \ -b \ 1]$ . In the two-dimensional plane through  $P$  perpendicular to the  $z$ -axis, the  $3 \times 3$  transformation matrix is

$$[T'] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -a & -b & 1 \end{bmatrix}$$

In three dimensions this two-dimensional translation is equivalent to a shearing of the vector  $PO$  in the  $x$  and  $y$  directions. The required transformation to accomplish this is

$$[T''] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -a & -b & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

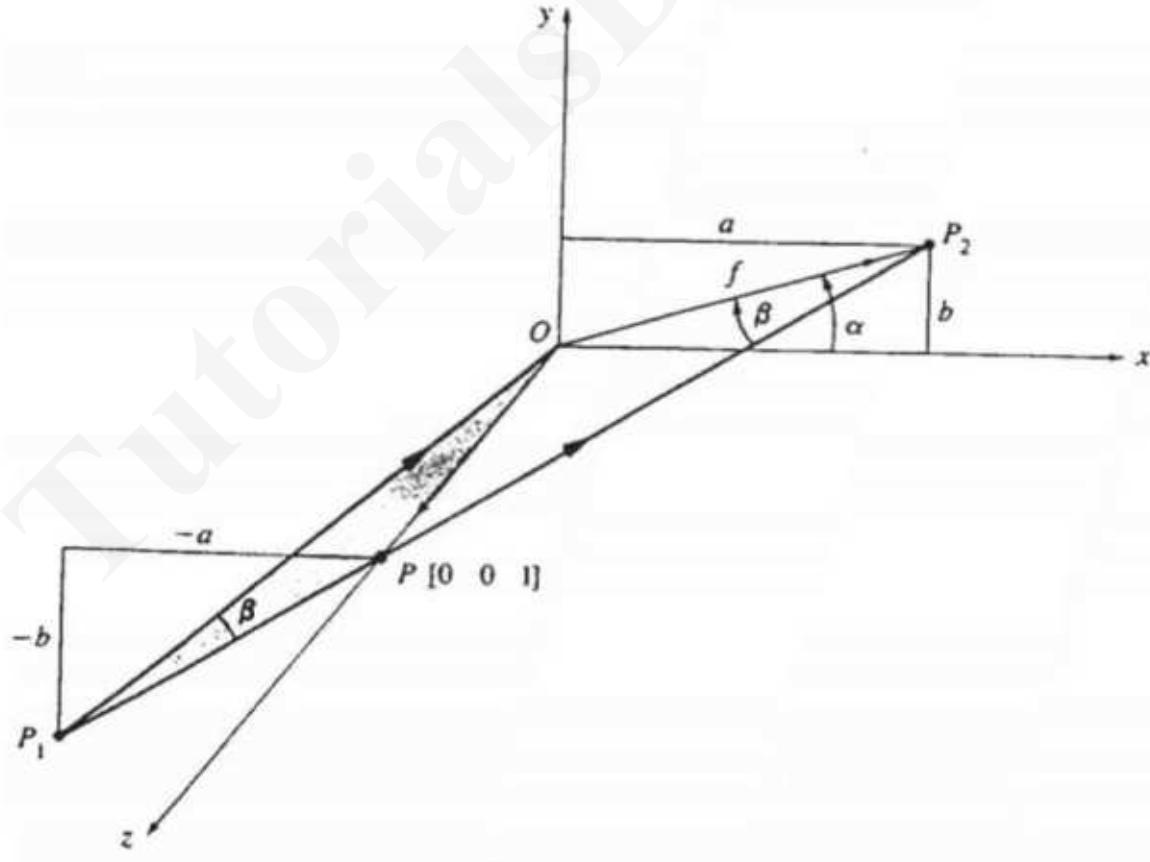


Figure 3-21 Direction of the oblique projection matrix.

projection onto the  $z = 0$  plane yields

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -a & -b & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

From Fig. 3-21

$$a = f \cos \alpha$$

$$b = f \sin \alpha$$

where  $f$  is the projected length of the  $z$ -axis unit vector, i.e., the foreshortening factor, and  $\alpha$  is the angle between the horizontal and the projected  $z$ -axis. Figure 3-21 also shows that  $\beta$ , the angle between the oblique projectors and the plane of projection, is

$$\beta = \cot^{-1}(f). \quad (3-43)$$

Thus, the transformation for an oblique projection is

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -f \cos \alpha & -f \sin \alpha & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-44)$$

If  $f = 0$ ,  $\beta = 90^\circ$ , then an orthographic projection results. If  $f = 1$ , the edges perpendicular to the projection plane are not foreshortened. This is the condition for a cavalier projection. From Eq. (3-43)

$$\beta = \cot^{-1}(1) = 45^\circ$$

For a cavalier projection, notice that  $\alpha$  is still a free parameter. Figure 3-22 shows cavalier projections for several values of  $\alpha$ . Commonly used values of  $\alpha$  are  $30^\circ$  and  $45^\circ$ . Values of  $180^\circ - \alpha$  are also acceptable.

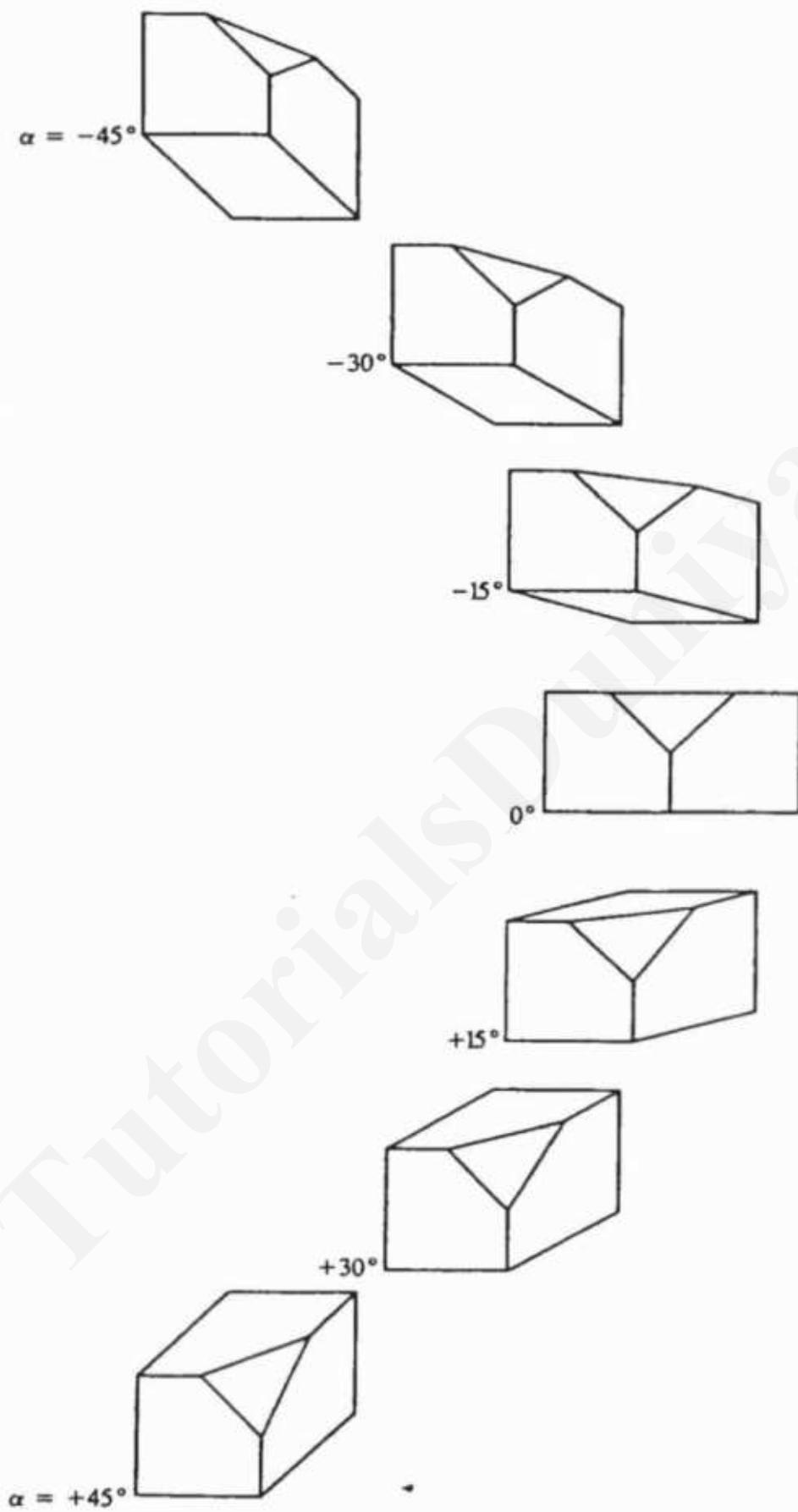
A cabinet projection is obtained when the foreshortening factor  $f = 1/2$ . Here

$$\beta = \cot^{-1}(1/2) = 63.435^\circ$$

Again, as shown in Fig. 3-23,  $\alpha$  is variable. Common values are  $30^\circ$  and  $45^\circ$ . Values of  $180^\circ - \alpha$  are also acceptable. Figure 3-24 shows oblique projections for foreshortening factors  $f = 1, 7/8, 3/4, 5/8, 1/2$ , with  $\alpha = 45^\circ$ .

Because one face is shown in its true shape, oblique projections are particularly suited for illustration of objects with circular or otherwise curved faces. Faces with these characteristics should be parallel to the plane of projection to avoid unwanted distortions. Similarly, as in all parallel projections, objects with one dimension significantly larger than the others suffer significant distortion unless the long dimension is parallel to the projection plane. These effects are illustrated in Fig. 3-25.

A detailed example is given below.



**Figure 3-22** Cavalier projections. Top to bottom,  $\alpha = -45^\circ$  to  $+45^\circ$  at  $15^\circ$  intervals with  $\beta = 45^\circ$ .

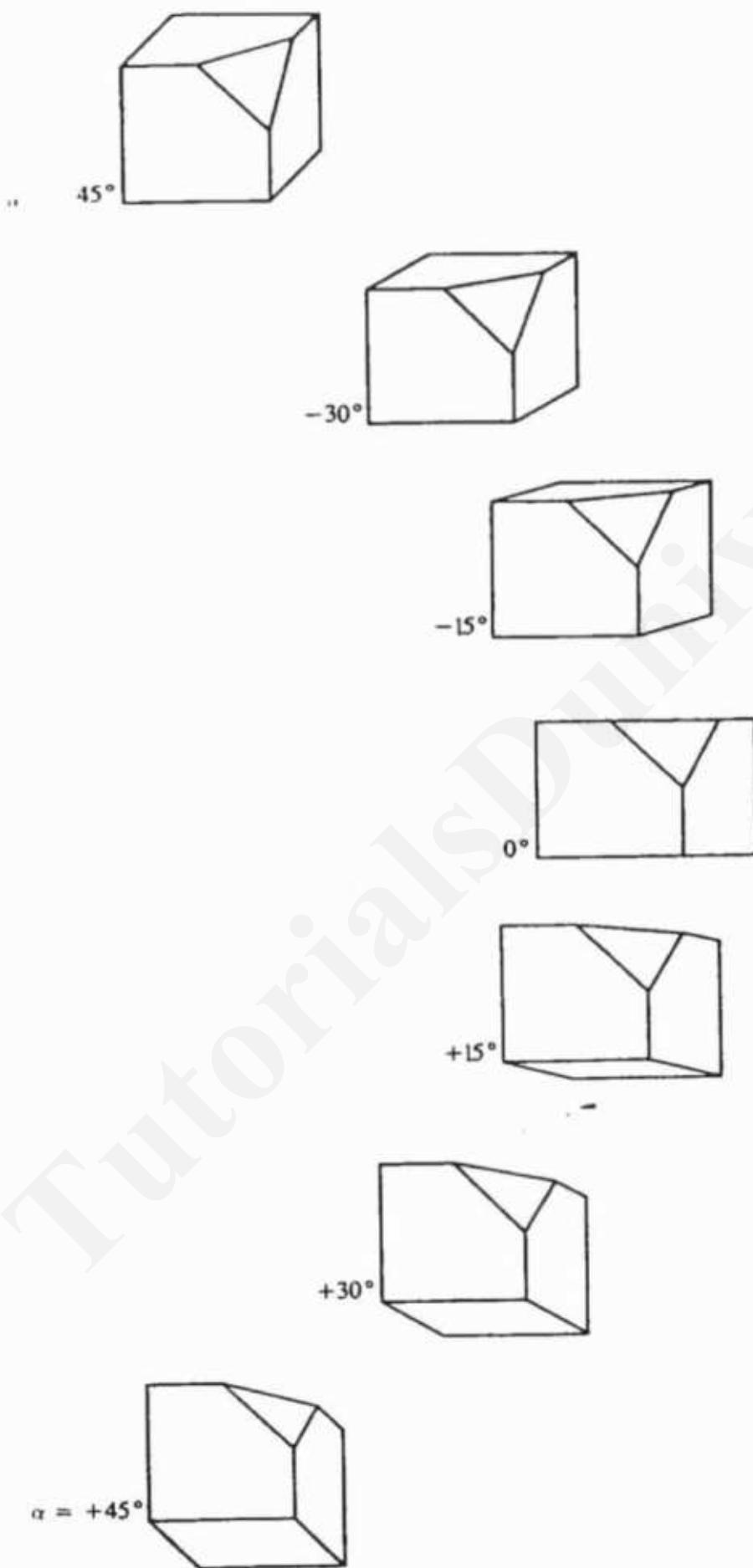
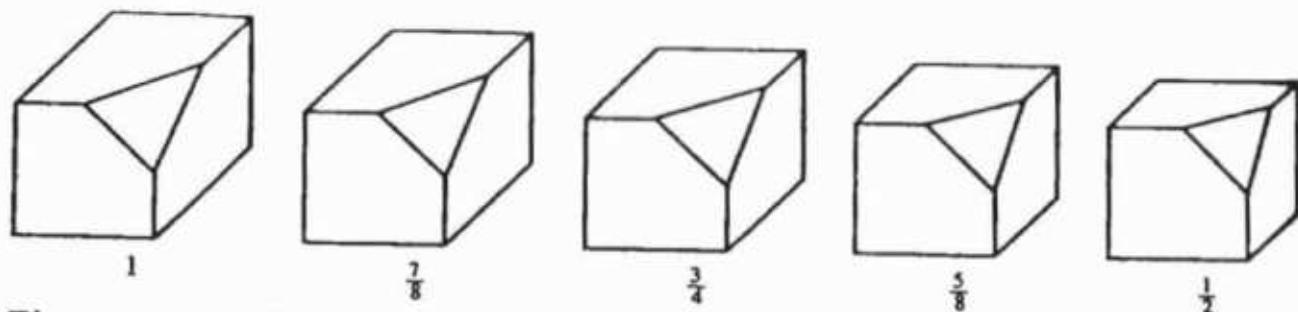


Figure 3-23 Cabinet projections. Top to bottom,  $\alpha = -45^\circ$  to  $+45^\circ$  at  $15^\circ$  intervals with  $f = 0.5$ .



**Figure 3-24** Oblique projections. Left to right,  $f = 1, 7/8, 3/4, 5/8, 1/2$ , with  $\alpha = 45^\circ$ .

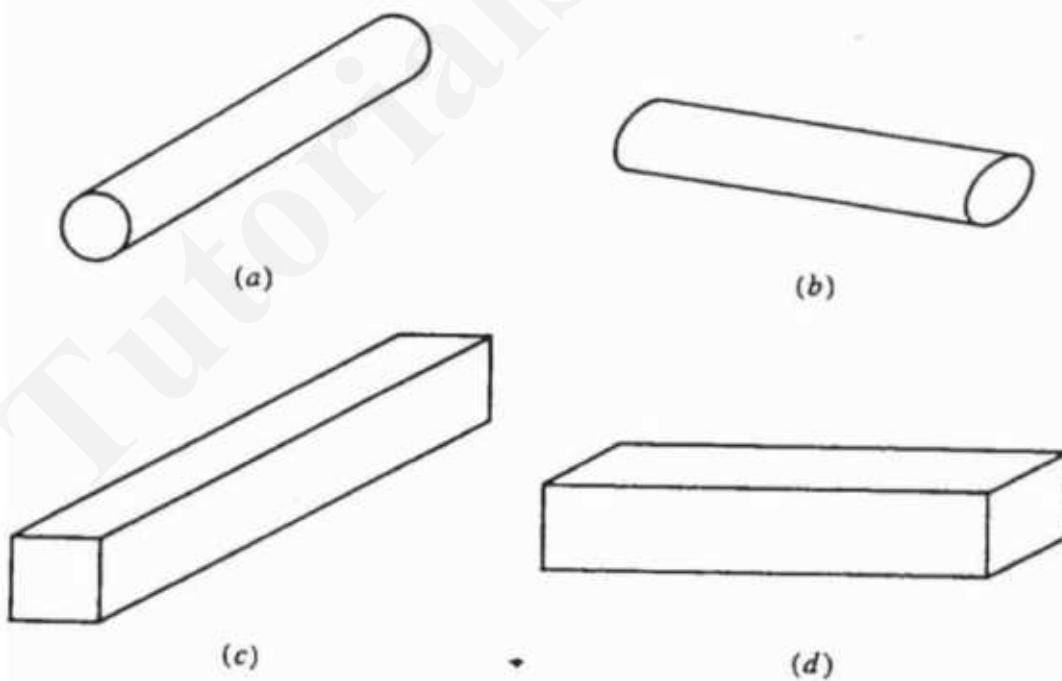
### Example 3-16 Oblique Projections

Develop cavalier and cabinet projections for the cube with one corner cut off (see Ex. 3-13).

Recalling that a cavalier projection is an oblique projection with  $\beta = 45^\circ$ , i.e., a foreshortening factor  $f = 1$ , and choosing a horizontal inclination angle  $\alpha = 30^\circ$ , Eq. (3-44) yields the transformation matrix

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -f \cos \alpha & -f \sin \alpha & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -0.866 & -0.5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Recalling the position vectors for the cube with the corner cut off [X] (see Ex. 3-13), the transformed position vectors are



**Figure 3-25** Distortion in oblique projections,  $f = 5/8, \alpha = 45^\circ$ . (a) Circular face parallel to projection plane; (b) circular face perpendicular to projection plane; (c) long dimension perpendicular to projection plane; (d) long dimension parallel to projection plane.

$$[X^*] = [X][T] = \begin{bmatrix} -0.866 & -0.5 & 0 & 1 \\ 0.134 & -0.5 & 0 & 1 \\ 0.134 & 0 & 0 & 1 \\ -0.366 & 0.5 & 0 & 1 \\ -0.866 & 0.5 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0.567 & 0.75 & 0 & 1 \end{bmatrix}$$

The result is shown in Fig. 3-22.

Turning now to the cabinet projection, and recalling that the foreshortening factor is  $1/2$ , Eq. (3-43) yields

$$\beta = \cot^{-1}(1/2) = \tan^{-1}(2) = 63.435^\circ$$

Again choosing  $\alpha = 30^\circ$  Eq. (3-44) becomes

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -f \cos \alpha & -f \sin \alpha & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -0.433 & -0.25 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The transformed position vectors for the cabinet projection of the cube with the corner cut off are

$$[X^*] = [X][T] = \begin{bmatrix} -0.433 & -0.25 & 0 & 1 \\ 0.567 & -0.25 & 0 & 1 \\ 0.567 & 0.25 & 0 & 1 \\ 0.067 & 0.75 & 0 & 1 \\ -0.433 & 0.75 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0.783 & 0.875 & 0 & 1 \end{bmatrix}$$

The result is shown in Fig. 3-23.

Notice that for both the cavalier and cabinet projections the triangular corner is *not* shown either true size or true shape because it is *not* parallel to the plane of projection ( $z = 0$ ).

### 3 15 PERSPECTIVE TRANSFORMATIONS

When any of the first three elements of the fourth column of the general  $4 \times 4$  homogeneous coordinate transformation matrix is nonzero, a perspective transformation results. As previously mentioned (see Sec. 3-11), a perspective transformation is a transformation from one three space to another three space. In contrast to the parallel transformations previously discussed, in perspective transformations parallel lines converge, object size is reduced with increasing distance.

from the center of projection, and nonuniform foreshortening of lines in the object as a function of orientation and distance of the object from the center of projection occurs. All of these effects aid the depth perception of the human visual system, but the shape of the object is not preserved.

A single-point perspective transformation is given by

$$[x \ y \ z \ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & r \\ 0 & 0 & 0 & 1 \end{bmatrix} = [x \ y \ z \ rz + 1] \quad (3 - 45)$$

Here  $h = rz + 1 \neq 1$ . The ordinary coordinates are obtained by dividing through by  $h$ , to yield

$$[x^* \ y^* \ z^* \ 1] = \left[ \frac{x}{rz + 1} \ \frac{y}{rz + 1} \ \frac{z}{rz + 1} \ 1 \right] \quad (3 - 46)$$

A perspective projection onto some two-dimensional viewing plane is obtained by concatenating an orthographic projection with the perspective transformation. For example, a perspective projection onto the  $z = 0$  plane is given by

$$\begin{aligned} [T] &= [P_r][P_z] \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & r \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & r \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \quad (3 - 47)$$

and

$$[x \ y \ z \ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & r \\ 0 & 0 & 0 & 1 \end{bmatrix} = [x \ y \ 0 \ rz + 1] \quad (3 - 48)$$

The ordinary coordinates are

$$[x^* \ y^* \ z^* \ 1] = \left[ \frac{x}{rz + 1} \ \frac{y}{rz + 1} \ 0 \ 1 \right] \quad (3 - 49)$$

To show that Eq. (3-47) produces a perspective projection onto the  $z = 0$  plane, consider Fig. 3-26, which illustrates the geometry for a perspective projection of the three-dimensional point  $P$  onto a  $z = z^* = 0$  plane at  $P^*$  from a center of projection at  $z_c$  on the  $z$ -axis. The coordinates of the projected point  $P^*$  are obtained using similar triangles. From Fig. 3-26

$$\frac{x^*}{z_c} = \frac{x}{z_c - z}$$

or

$$x^* = \frac{x}{1 - \frac{z}{z_c}}$$

and

$$\frac{y^*}{\sqrt{x^{*2} + z_c^2}} = \frac{y}{\sqrt{x^2 + (z_c - z)^2}}$$

or

$$y^* = \frac{y}{1 - \frac{z}{z_c}}$$

• is, of course, zero.

Letting  $r = -1/z_c$  yields results identical to those obtained using Eq. (3-47). Thus, Eq. (3-47) produces a perspective projection onto the  $z = 0$  plane from a center of projection at  $(-1/r)$  on the  $z$ -axis. Notice that as  $z_c$  approaches infinity,  $r$  approaches zero and an axonometric projection onto the  $z = 0$  plane results. Further, notice that for points in the plane of projection, i.e.,  $z = 0$ , the perspective transformation has no effect. Also note that the origin ( $x = y = z = 0$ ) is unaffected. Consequently, if the plane of projection ( $z = 0$ ) passes through an object, then that section of the object is shown at true size and true shape. All other parts of the object are distorted.

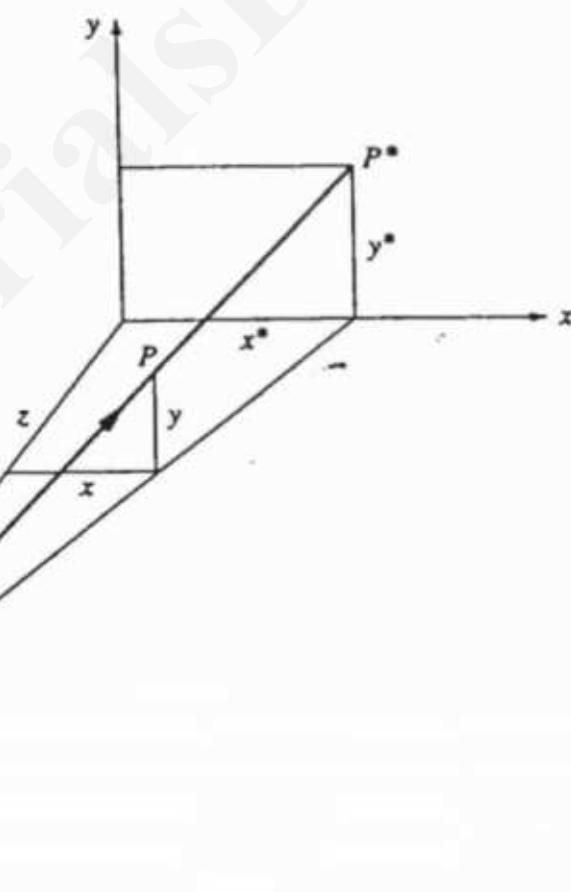


Figure 3-26 Perspective projection of a point.

To help understand the effects of a perspective transformation consider Fig. 3-27. Figure 3-27 shows the perspective projection onto the  $z = 0$  plane of the line  $AB$  originally parallel to the  $z$ -axis, into the line  $A'B'$  in the  $z = 0$  plane, from a center of projection at  $-1/r$  on the  $z$ -axis. The transformation can be considered in two steps (see Eq. 3-47). First, the perspective transformation of the line  $AB$  yields the three-dimensional transformed line  $A'B'$  (see Fig. 3-27 below). Subsequent orthographic projection of the line  $A'B'$  in three-dimensional perspective space onto the  $z = 0$  plane from a center of projection at infinity on the  $z$ -axis yields the line  $A^*B^*$ .

Examination of Fig. 3-27 shows that the line  $A'B'$  intersects the  $z = 0$  plane at the same point as the line  $AB$ . It also intersects the  $z$ -axis at  $z = +1/r$ . Effectively then, the perspective transformation (see Eqs. 3-45 and 3-46) has transformed the intersection point at infinity of the line  $AB$  parallel to the  $z$ -axis and the  $z$ -axis itself into the finite point at  $z = 1/r$  on the  $z$ -axis. This point is called the vanishing point<sup>†</sup>. Notice that the vanishing point lies an equal distance on the opposite side of the plane of projection from the center of projection, e.g., if  $z = 0$  is the projection plane and the center of projection is at  $z = -1/r$ , the vanishing point is at  $z = +1/r$ .

To confirm this observation consider the perspective transformation of the point at infinity on the  $+z$ -axis, i.e.,

$$\begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & r \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & r \end{bmatrix} \quad (3-50)$$

The point  $[x^* \ y^* \ z^* \ 1] = [0 \ 0 \ 1/r \ 1]$ , corresponding to the transformed point at infinity on the positive  $z$ -axis, is now a finite point on the positive  $z$ -axis. This means that the entire semi-infinite positive space ( $0 \leq z \leq \infty$ ) is transformed to the finite positive half space  $0 \leq z^* \leq 1/r$ . Further, all lines originally parallel to the  $z$ -axis now pass through the point  $[0 \ 0 \ 1/r \ 1]$ , the vanishing point.

Before presenting some illustrative examples the single-point perspective transformations with centers of projection and vanishing points on the  $x$ - and  $y$ -axes are given for completeness. The single-point perspective transformation

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & p \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & (px+1) \end{bmatrix} \quad (3-51)$$

with ordinary coordinates

$$\begin{bmatrix} x^* & y^* & z^* & 1 \end{bmatrix} = \left[ \frac{x}{px+1} \quad \frac{y}{px+1} \quad \frac{z}{px+1} \quad 1 \right] \quad (3-52)$$

<sup>†</sup>Intuitively the vanishing point is that point in the 'distance' to which parallel lines 'appear' to converge and 'vanish'. A practical example is a long straight railroad track.

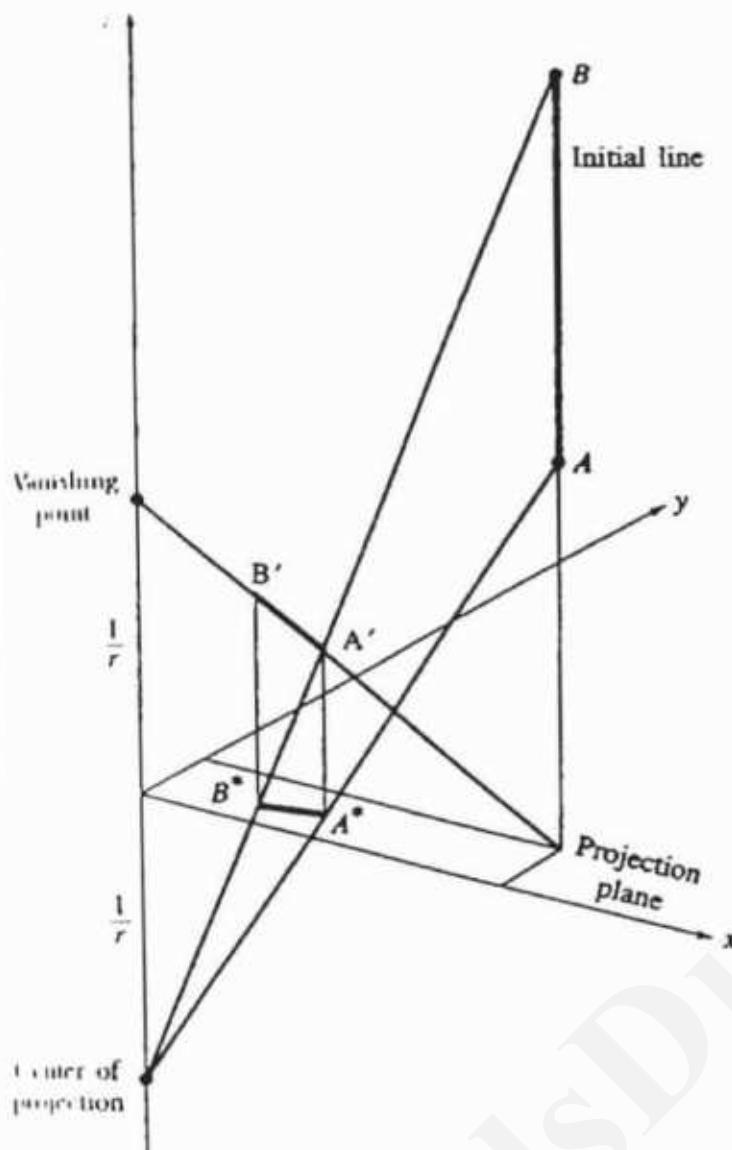


Figure 3-27 Projection of a line parallel to the  $z$ -axis.

has a center of projection at  $[ -1/p \ 0 \ 0 \ 1 ]$  and a vanishing point located on the  $x$ -axis at  $[ 1/p \ 0 \ 0 \ 1 ]$ .

The single-point perspective transformation

$$[x \ y \ z \ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & q \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [x \ y \ z \ (qy + 1)] \quad (3-53)$$

with ordinary coordinates

$$[x^* \ y^* \ z^* \ 1] = \left[ \frac{x}{qy + 1} \quad \frac{y}{qy + 1} \quad \frac{z}{qy + 1} \quad 1 \right] \quad (3-54)$$

has a center of projection at  $[ 0 \ -1/q \ 0 \ 1 ]$  and a vanishing point located on the  $y$ -axis at  $[ 0 \ 1/q \ 0 \ 1 ]$ .

**Example 3-17 Perspective Transformation of a Line Parallel to the  $z$ -Axis**

Consider the line segment  $AB$  in Fig. 3-27 parallel to the  $z$ -axis with end points  $A[3 \ 2 \ 4 \ 1]$  and  $B[3 \ 2 \ 8 \ 1]$ . Perform a perspective projection onto the  $z = 0$  plane from a center of projection at  $z_c = -2$ . The perspective transformation of  $AB$  to  $A'B'$  with  $r = 0.5$  is

$$\begin{aligned} A \begin{bmatrix} 3 & 2 & 4 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} &= \begin{bmatrix} 3 & 2 & 4 & 3 \\ 3 & 2 & 8 & 5 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0.667 & 1.333 & 1 \\ 0.6 & 0.4 & 1.6 & 1 \end{bmatrix} \begin{matrix} A' \\ B' \end{matrix} \end{aligned}$$

The parametric equation of the line segment  $A'B'$  is

$$P(t) = [A'] + [B' - A']t \quad 0 \leq t \leq 1$$

$$\text{or } P(t) = [1 \ 0.667 \ 1.333 \ 1] + [-0.4 \ -0.267 \ 0.267 \ 0]t$$

Intersection of this line with the  $x = 0$ ,  $y = 0$  and  $z = 0$  planes yields

$$\begin{aligned} x(t) = 0 &= 1 - 0.4t \quad \rightarrow \quad t = 2.50 \\ y(t) = 0 &= 0.667 - 0.267t \quad \rightarrow \quad t = 2.50 \\ z(t) = 0 &= 1.333 + 0.267t \quad \rightarrow \quad t = -5.0 \end{aligned}$$

Substituting  $t = 2.5$  into the parametric equation of the line  $A'B'$  yields

$$z(2.5) = 1.333 + (0.267)(2.5) = 2.0$$

which represents the intersection of the line  $A'B'$  with the  $z$ -axis at  $z = +1/r$ , the vanishing point. Now substituting  $t = -5.0$  into the  $x$  and  $y$  component equations yields the intersection with the  $z = 0$  plane, i.e.,

$$\begin{aligned} x(-5.0) &= 1 - (0.4)(-5.0) = 3.0 \\ y(-5.0) &= 0.667 - (0.267)(-5.0) = 2.0 \end{aligned}$$

which is the same as the intersection of the line  $AB$  with the  $z = 0$  plane.

Projection of line  $A'B'$  into the line  $A^*B^*$  in the  $z = 0$  plane is given by

$$\begin{aligned} A' \begin{bmatrix} 1 & 0.667 & 1.333 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} &= \begin{bmatrix} 1 & 0.667 & 0 & 1 \\ 0.6 & 0.4 & 0 & 1 \end{bmatrix} \begin{matrix} A^* \\ B^* \end{matrix} \end{aligned}$$

An example using a simple cube is given below.

**Example 3–18 Single-Point Perspective Transformation of a Cube**

Perform a perspective projection onto the  $z = 0$  plane of the unit cube shown in Fig. 3–28a from a center of projection at  $z_c = 10$  on the  $z$ -axis.

The single-point perspective factor  $r$  is

$$r = -1/z_c = -1/10 = -0.1$$

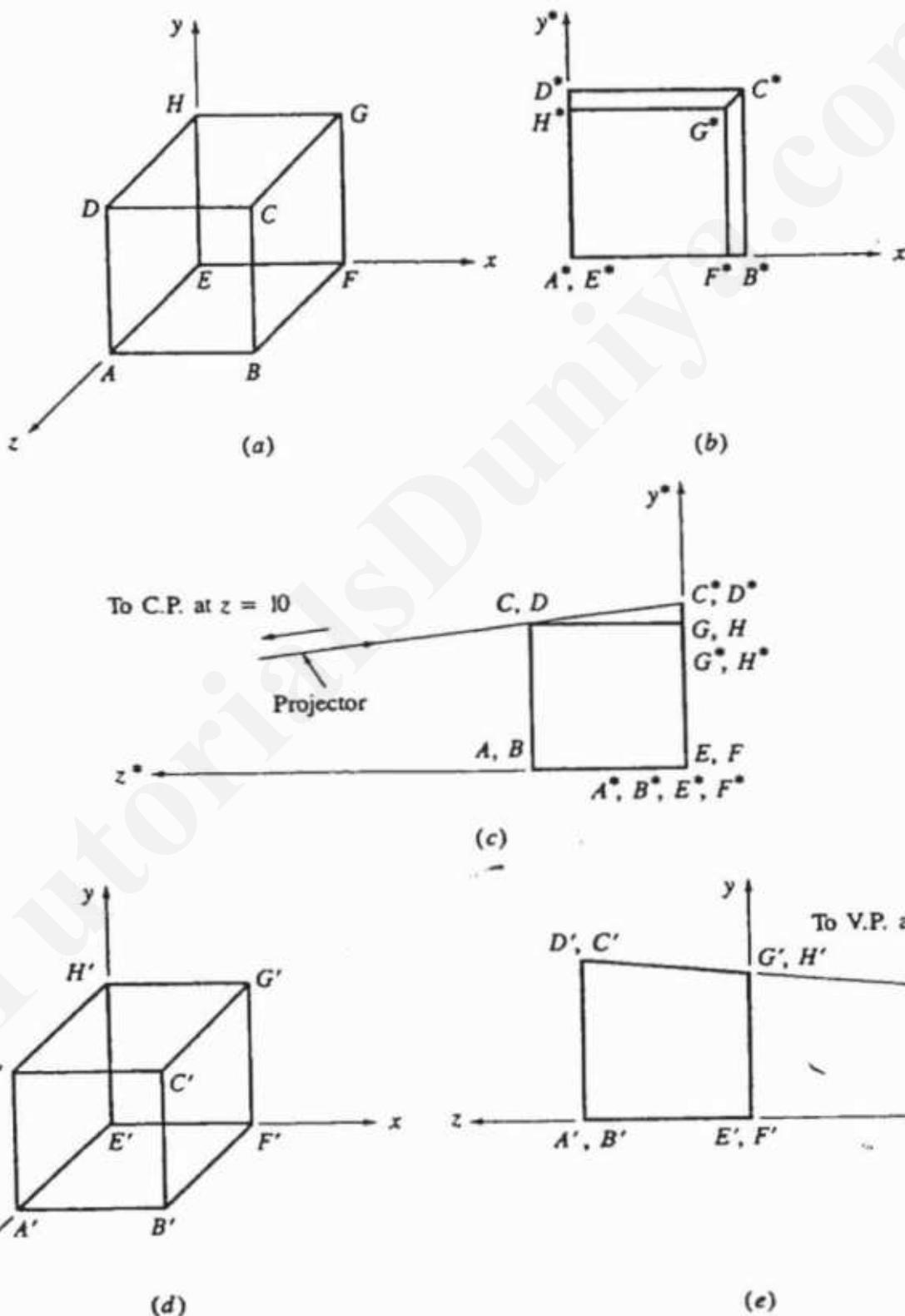


Figure 3–28 Single-point perspective projection of a unit cube.

From Eq. (3-48) the transformation is

$$\begin{aligned}
 [T] &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -0.1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 [X^*] &= [X][T] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -0.1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 0 & 0 & 0 & 0.9 \\ 1 & 0 & 0 & 0.9 \\ 1 & 1 & 0 & 0.9 \\ 0 & 1 & 0 & 0.9 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1.11 & 0 & 0 & 1 \\ 1.11 & 1.11 & 0 & 1 \\ 0 & 1.11 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

The result is shown in Fig. 3-28b. Notice that since the center of projection is on the positive  $z$ -axis the front face of the cube  $ABCD$  projects larger than the back face. Figure 3-28c, which is a parallel projection of the original cube onto the  $x = 0$  plane, shows why.

Notice also that because the vanishing point lies on the  $z$ -axis the line  $C^*G^*$  in Fig. 3-28b passes through the origin.

An alternate and equivalent approach to that above is to first perform the perspective transformation to obtain a distorted object in three space and then to orthographically project the result onto some plane. The distorted object is obtained by

$$\begin{aligned}
 [X'] &= [X][P_r] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -0.1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 0 & 0 & 1 & 0.9 \\ 1 & 0 & 1 & 0.9 \\ 1 & 1 & 1 & 0.9 \\ 0 & 1 & 1 & 0.9 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1.11 & 1 \\ 1.11 & 0 & 1.11 & 1 \\ 1.11 & 1.11 & 1.11 & 1 \\ 0 & 1.11 & 1.11 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

The result is shown using an oblique projection in Fig. 3-28d. Notice that the 'front' face ( $A'B'C'D'$ ) is larger than the 'rear' face ( $E'F'G'H'$ ). Subsequent orthographic projection onto the  $z = 0$  plane yields the same result for  $[X^*]$  as given above and illustrated in Fig. 3-28c.

Figure 3-28e, which is an orthographic projection of the distorted object of Fig. 3-28d onto the  $z = 0$  plane, shows that the edges of the distorted object originally parallel to the  $z$ -axis now converge to the vanishing point at  $r_v = -10$ .

Figure 3-28b does not convey the three-dimensional character of the cube. A more satisfactory result is obtained by centering the cube. This is illustrated in the next example.

### Example 3-19 Single-Point Perspective Transformation of a Centered Cube

The cube shown in Fig. 3-28a can be centered on the  $z$ -axis by translating it  $1/2$  unit in the  $x$  and  $y$  directions. The resulting transformation is

$$\begin{aligned}[T] &= [Tr_{xy}][Pr_z] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -0.5 & -0.5 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -0.1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -0.1 \\ -0.5 & -0.5 & 0 & 1 \end{bmatrix}\end{aligned}$$

The translated cube is shown in Fig. 3-29a.

The transformed ordinary coordinates are

$$\begin{aligned}[X^*] &= [X][T] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -0.1 \\ -0.5 & -0.5 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} -0.5 & -0.5 & 0 & 0.9 \\ 0.5 & -0.5 & 0 & 0.9 \\ 0.5 & 0.5 & 0 & 0.9 \\ -0.5 & 0.5 & 0 & 0.9 \\ -0.5 & -0.5 & 0 & 1 \\ 0.5 & -0.5 & 0 & 1 \\ 0.5 & 0.5 & 0 & 1 \\ -0.5 & 0.5 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -0.56 & -0.56 & 0 & 1 \\ 0.56 & -0.56 & 0 & 1 \\ 0.56 & 0.56 & 0 & 1 \\ -0.56 & 0.56 & 0 & 1 \\ -0.5 & -0.5 & 0 & 1 \\ 0.5 & -0.5 & 0 & 1 \\ 0.5 & 0.5 & 0 & 1 \\ -0.5 & 0.5 & 0 & 1 \end{bmatrix}\end{aligned}$$

The result is shown in Fig. 3-29b. Notice that the lines originally parallel to the  $z$ -axis connecting the corners of the front and rear faces now converge to intersect the  $z$ -axis ( $x = 0, y = 0$ ) in Fig. 3-29b.

Unfortunately, the resulting display still does not provide an adequate perception of the three-dimensional shape of the object. Consequently, we turn our attention to more complex perspective transformations.

If two terms in the first three rows of the fourth column of the  $4 \times 4$  transformation matrix are nonzero, the result is called a two-point perspective transformation. The two-point perspective transformation

$$[x \ y \ z \ 1] \begin{bmatrix} 1 & 0 & 0 & p \\ 0 & 1 & 0 & q \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [x \ y \ z \ (px + qy + 1)] \quad (3-55)$$

with ordinary coordinates,

$$[x^* \ y^* \ z^* \ 1] = \left[ \frac{x}{px + qy + 1} \ \frac{y}{px + qy + 1} \ \frac{z}{px + qy + 1} \ 1 \right] \quad (3-56)$$

has two centers of projection: one on the  $x$ -axis at  $[-1/p \ 0 \ 0 \ 1]$  and one on the  $y$ -axis at  $[0 \ -1/q \ 0 \ 1]$ , and two vanishing points: one on the  $x$ -axis at  $[1/p \ 0 \ 0 \ 1]$  and one on the  $y$ -axis at  $[0 \ 1/q \ 0 \ 1]$ . Note that the two-point perspective transformation given by Eq. (3-55) can be obtained by concatenation of two single-point perspective transformations. Specifically,

$$\begin{aligned} [P_{pq}] &= [P_p][P_q] \\ &= [P_q][P_p] \end{aligned}$$

where  $[P_{pq}]$  is given by Eq. (3-55),  $[P_p]$  by Eq. (3-53) and  $[P_q]$  by Eq. (3-51). The next example shows the details of a two-point perspective projection.

### Example 3-20 Two-Point Perspective Projections

Again consider the cube described in Ex. 3-18 transformed by a two-point perspective transformation with centers of projection at  $x = -10$  and  $y = -10$  projected onto the  $z = 0$  plane. The transformation is obtained by concatenating Eqs. (3-55) and (3-27). Specifically,

$$\begin{aligned} [T] &= [P_{pq}][P_z] = \begin{bmatrix} 1 & 0 & 0 & p \\ 0 & 1 & 0 & q \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 & p \\ 0 & 1 & 0 & q \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

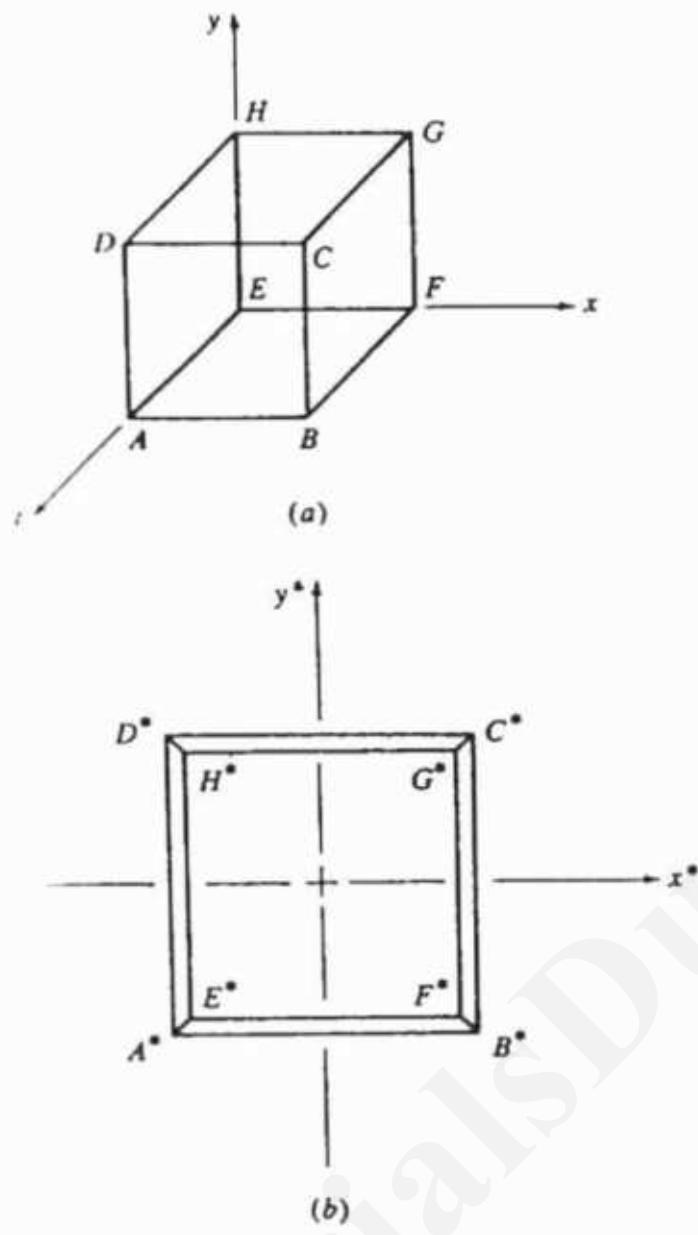


Figure 3-29 Single-point perspective projection of a centered unit cube.

Here  $p$  and  $q$  are

$$p = -1/(-10) = 0.1 \quad q = -1/(-10) = 0.1$$

The transformed coordinates of the cube are

$$[X^*] = [X][T] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0.1 \\ 0 & 1 & 0 & 0.1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1.1 \\ 1 & 1 & 0 & 1.2 \\ 0 & 1 & 0 & 1.1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1.1 \\ 1 & 1 & 0 & 1.2 \\ 0 & 1 & 0 & 1.1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0.909 & 0 & 0 & 1 \\ 0.833 & 0.833 & 0 & 1 \\ 0 & 0.909 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0.909 & 0 & 0 & 1 \\ 0.833 & 0.833 & 0 & 1 \\ 0 & 0.909 & 0 & 1 \end{bmatrix}.$$

The results are shown in Fig. 3-30a. The two vanishing points are at  $x = 10$  and  $y = 10$ .

Centering the cube on the  $z$ -axis by translating  $-0.5$  in  $x$  and  $y$  as was done in Ex. 3-18 yields the concatenated transformation matrix

$$[T] = [T_{xy}][P_{pq}][P_z]$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -0.5 & -0.5 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0.1 \\ 0 & 1 & 0 & 0.1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0.1 \\ 0 & 1 & 0 & 0.1 \\ 0 & 0 & 0 & 0 \\ -0.5 & -0.5 & 0 & 0.9 \end{bmatrix}$$

where projection onto the  $z = 0$  plane has been assumed. Notice that here the overall scaling factor (see Eq. 3-4) is no longer unity, i.e., there is an apparent scaling of the cube caused by translation. The transformed coordinates are

$$[X^*] = [X][T] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0.1 \\ 0 & 1 & 0 & 0.1 \\ 0 & 0 & 0 & 0 \\ -0.5 & -0.5 & 0 & 0.9 \end{bmatrix}$$

$$= \begin{bmatrix} -0.5 & -0.5 & 0 & 0.9 \\ 0.5 & -0.5 & 0 & 1 \\ 0.5 & 0.5 & 0 & 1.1 \\ -0.5 & 0.5 & 0 & 1 \\ -0.5 & -0.5 & 0 & 0.9 \\ 0.5 & -0.5 & 0 & 1 \\ 0.5 & 0.5 & 0 & 1.1 \\ -0.5 & 0.5 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -0.56 & -0.56 & 0 & 1 \\ 0.5 & -0.5 & 0 & 1 \\ 0.46 & 0.46 & 0 & 1 \\ -0.5 & 0.5 & 0 & 1 \\ -0.56 & -0.56 & 0 & 1 \\ 0.5 & -0.5 & 0 & 1 \\ 0.46 & 0.46 & 0 & 1 \\ -0.5 & 0.5 & 0 & 1 \end{bmatrix}$$

The results are shown in Fig. 3-30b.

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

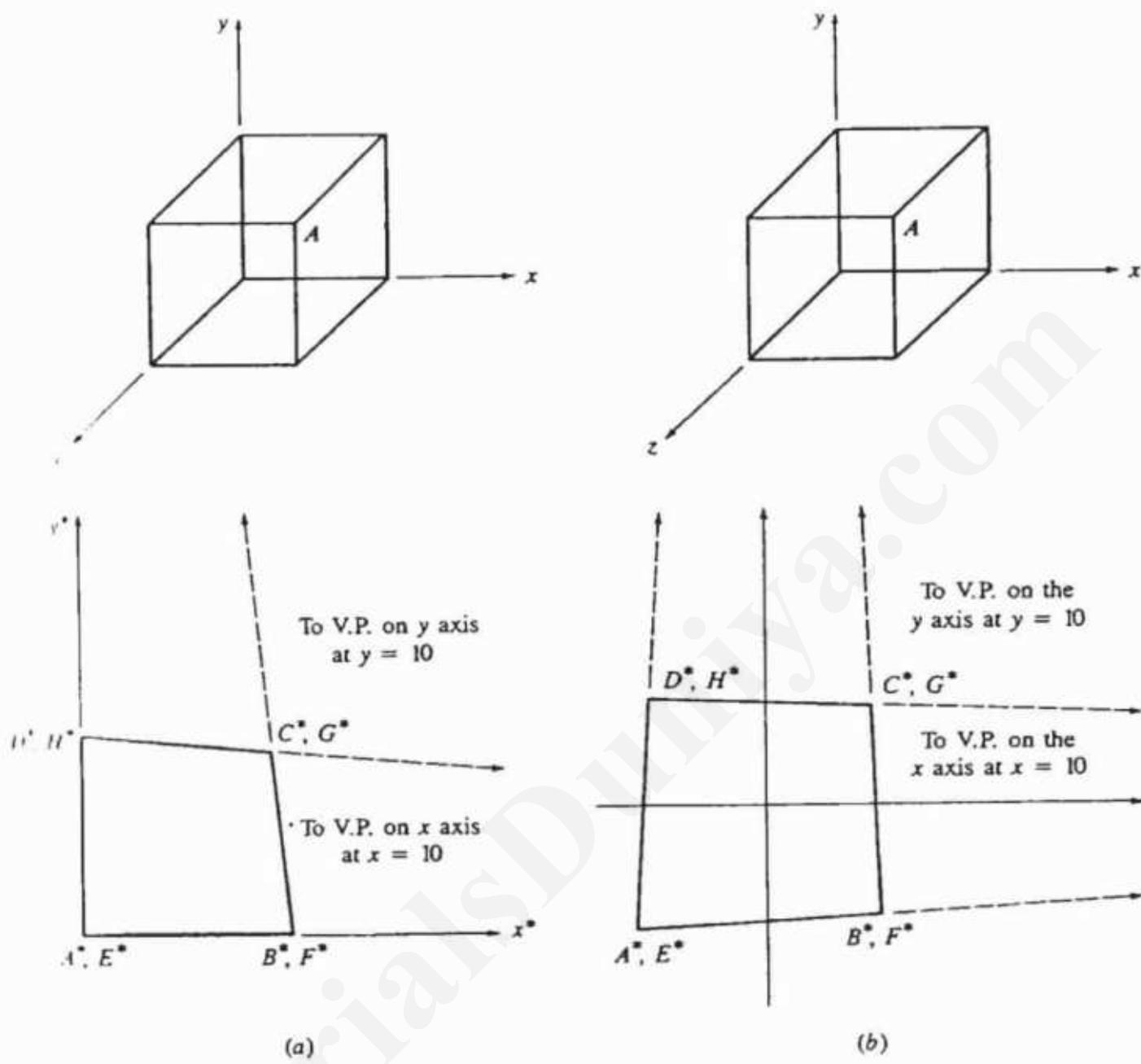


Figure 3-30 Two-point perspective projections. (a) Non-centered; (b) centered.

Again the resulting display does not provide an adequate perception of the three-dimensional shape of the object. Hence we turn our attention to three-point perspective transformations.

If three terms in the first three rows of the fourth column of the  $4 \times 4$  transformation matrix are nonzero, then a three-point perspective is obtained. The three-point perspective transformation

$$[x \ y \ z \ 1] \begin{bmatrix} 1 & 0 & 0 & p \\ 0 & 1 & 0 & q \\ 0 & 0 & 1 & r \\ 0 & 0 & 0 & 1 \end{bmatrix} = [x \ y \ z \ (px + qy + rz + 1)] \quad (3-57)$$

with ordinary coordinates

$$[x^* \ y^* \ z^* \ 1] = \left[ \frac{x}{px + qy + rz + 1} \ \frac{y}{px + qy + rz + 1} \ \frac{z}{px + qy + rz + 1} \ 1 \right] \quad (3-58)$$

has three centers of projection: one on the  $x$ -axis at  $[-1/p \ 0 \ 0 \ 1]$ , one on the  $y$ -axis at  $[0 \ -1/q \ 0 \ 1]$ , and one on the  $z$ -axis at  $[0 \ 0 \ -1/r \ 1]$ , and three vanishing points: one on the  $x$ -axis at  $[1/p \ 0 \ 0 \ 1]$ , one on the  $y$ -axis at  $[0 \ 1/q \ 0 \ 1]$ , and one on the  $z$ -axis at  $[0 \ 0 \ 1/r \ 1]$ .

Again, note that the three-point perspective transformation given by Eq. (3-57) can be obtained by concatenation of three single-point perspective transformations, one for each of the coordinate axes. An example illustrates the generation of a three-point perspective.

### Example 3-21 Three-Point Perspective Transformation

Consider the cube described in Ex. 3-18 transformed by a three-point perspective transformation with centers of projection at  $x = -10$ ,  $y = -10$  and  $z = 10$  projected onto the  $z = 0$  plane. Vanishing points are at  $x = 10$ ,  $y = 10$  and  $z = -10$ . The transformation matrix is

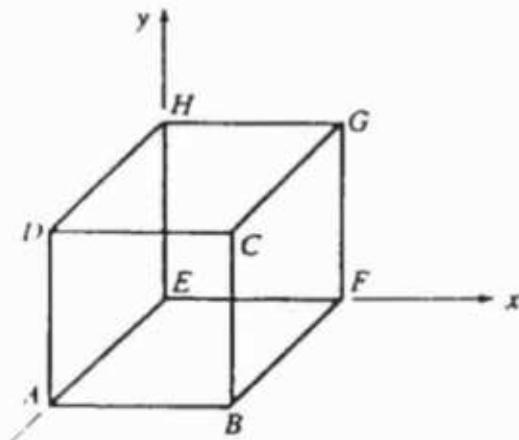
$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0.1 \\ 0 & 1 & 0 & 0.1 \\ 0 & 0 & 0 & -0.1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The transformed coordinates of the cube are

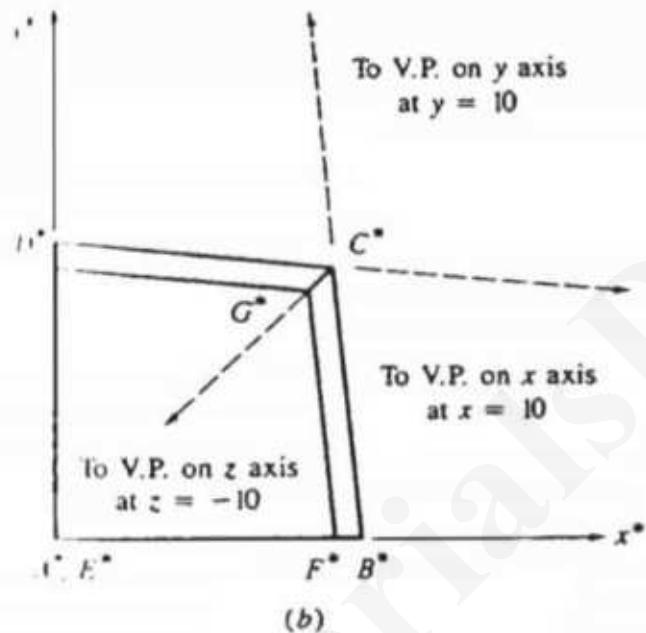
$$\begin{aligned} [X^*] &= [X][T] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0.1 \\ 0 & 1 & 0 & 0.1 \\ 0 & 0 & 0 & -0.1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 & 0 & 0.9 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1.1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1.1 \\ 1 & 1 & 0 & 1.2 \\ 0 & 1 & 0 & 1.1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0.909 & 0.909 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0.909 & 0 & 0 & 1 \\ 0.833 & 0.833 & 0 & 1 \\ 0 & 0.909 & 0 & 1 \end{bmatrix} \end{aligned}$$

The result is shown in Fig. 3-31b. The distorted object, after perspective transformation, is shown in Fig. 3-31c. Note the convergence of the edges.

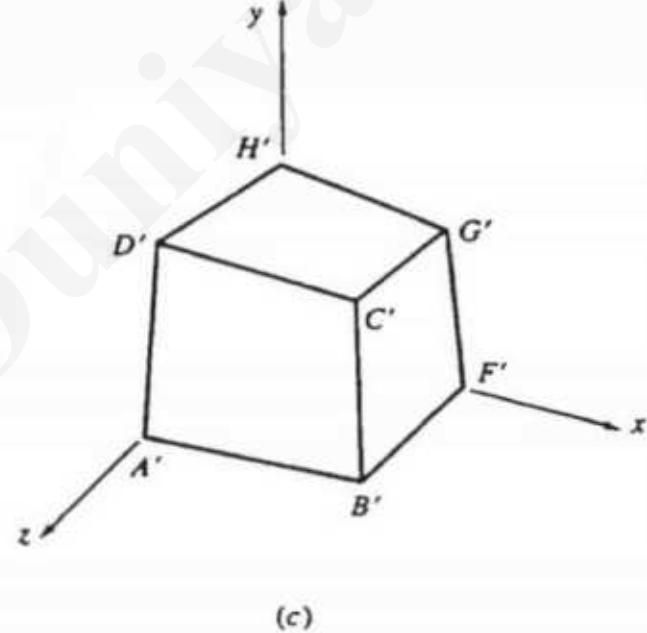
Again, although mathematically correct the resulting view is not informative. Appropriate techniques for generating perspective views are discussed in Sec. 3-16.



(a)



(b)



(c)

**Figure 3-31** Three-point perspective. (a) The original cube; (b) perspective projection onto the  $z = 0$  plane; (c) the distorted cube.

### 3.16 TECHNIQUES FOR GENERATING PERSPECTIVE VIEWS

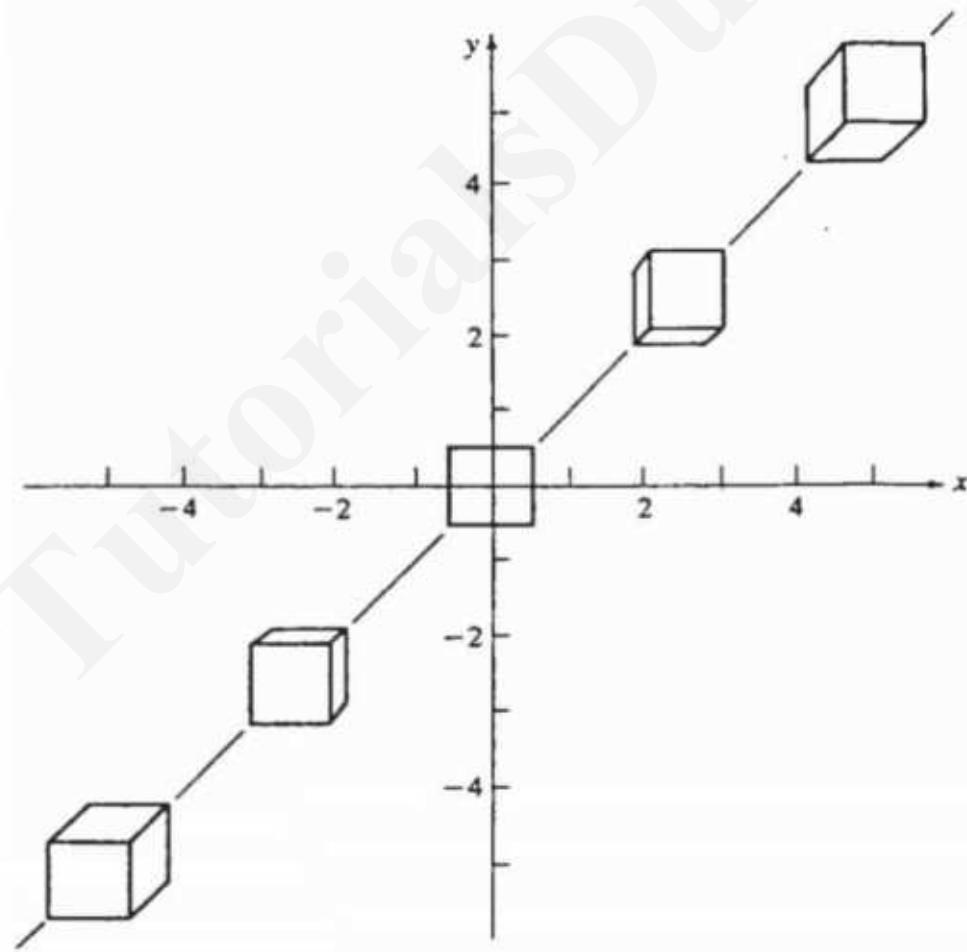
The perspective projection views shown in the previous section were uninformative because in each case only one face of the cube was visible from each center of projection. For an observer to perceive the three-dimensional shape of an object from a single view, it is necessary that multiple faces of the object be visible. For simple cuboidal objects, a minimum of three faces must be visible. For a fixed center of projection with the projection plane perpendicular to the viewing direction, a single-point perspective projection, preceded by translation and/or rotation of the object, provides the required multiple face view. Then, provided the center of projection is not too close to the object, a realistic view is obtained.

First, consider simple translation of the object followed by a single-point perspective projection from a center of projection at  $z = z_c$  onto the  $z = 0$  plane. The required transformation is

$$\begin{aligned}[T] &= [Tr_{xyz}][P_{rz}] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ l & m & n & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & r \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & r \\ l & m & 0 & 1+rn \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/z_c \\ l & m & 0 & 1-n/z_c \end{bmatrix} \quad (3-59)\end{aligned}$$

where  $r = -1/z_c$ .

Equation (3-59), along with Fig. 3-32, shows that translation in the  $x$  and  $y$  directions reveals additional faces of the object. Translation in both  $x$  and  $y$  is required to reveal three faces of a simple cuboidal object. Figure 3-32 shows the results of translating an origin-centered unit cube along the line  $y = x$ , followed by a single-point perspective projection onto the  $z = 0$  plane. Notice that the front face is shown true size and shape.



**Figure 3-32** Single-point perspective projection with  $x$  and  $y$  translations.

Equation (3-59) also shows that translation in the  $z$  direction, i.e., toward or away from the center of projection, results in an apparent scale change (as shown by the term  $1 - n/z_c$ ). This effect corresponds to physical reality, since objects that are farther away from an observer appear smaller. Notice that as the center of projection approaches infinity the scale effect disappears. Figure 3-33 schematically illustrates the effect. As shown in Fig. 3-33, the object can be on either side of the center of projection. If the object and the plane of projection are on the same side of the center of projection, as shown in Fig. 3-33, then an upright image results. However, if the object and the plane of projection are on opposite sides of the center of projection an inverted image results.

Figure 3-34 illustrates the effects of translation in all three directions. Here, a cube is translated along the three-dimensional line from  $-x = -y = -z$  to  $y = z$ . Notice the apparent size increase. Also notice that the true shape but *not* the true size of the front face is shown in all views.

An example more fully illustrates these concepts.

### Example 3-22 Single-Point Perspective Projection with Translation

Consider an origin-centered unit cube with position vectors given by

$$[X] = \begin{bmatrix} -0.5 & -0.5 & 0.5 & 1 \\ 0.5 & -0.5 & 0.5 & 1 \\ 0.5 & 0.5 & 0.5 & 1 \\ -0.5 & 0.5 & 0.5 & 1 \\ -0.5 & -0.5 & -0.5 & 1 \\ 0.5 & -0.5 & -0.5 & 1 \\ 0.5 & 0.5 & -0.5 & 1 \\ -0.5 & 0.5 & -0.5 & 1 \end{bmatrix}$$

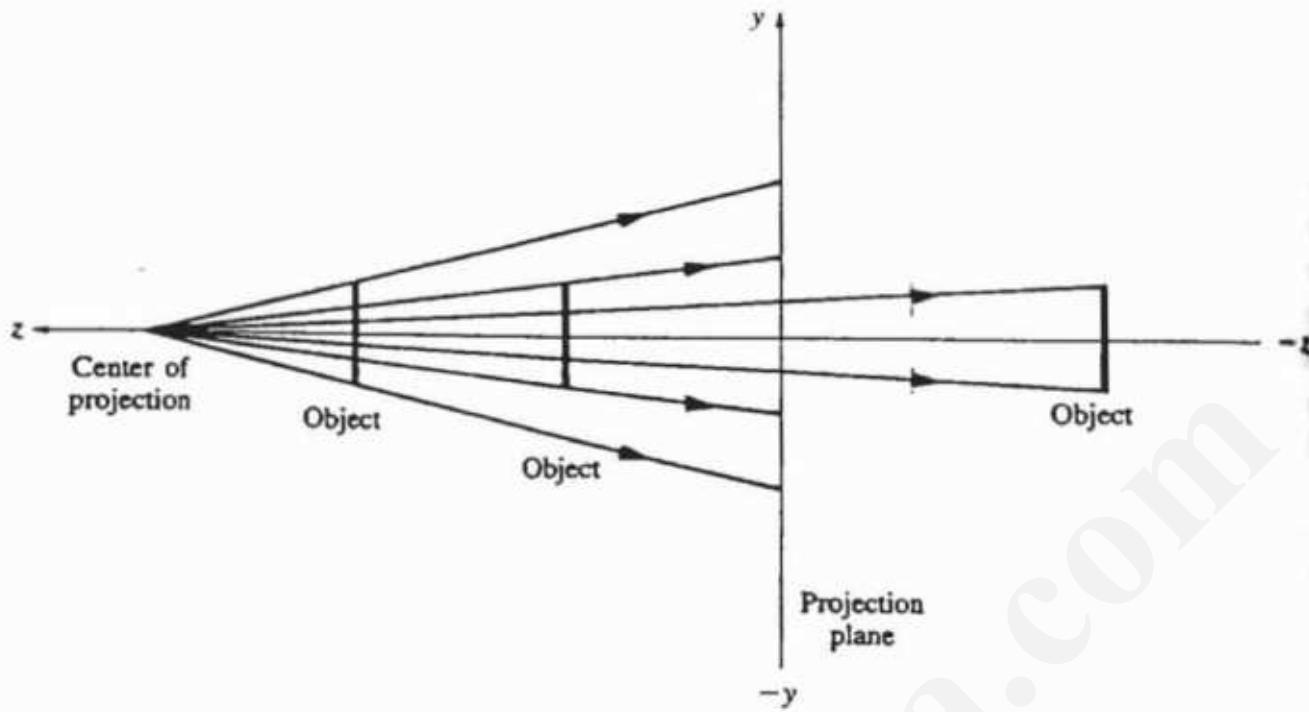
Translate the cube 5 units in the  $x$  and  $y$  directions and perform a single-point perspective projection onto the  $z = 0$  plane from a center of projection at  $z = z_c = 10$ .

From Eq. (3-59) the concatenated transformation matrix is

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -0.1 \\ 5 & 5 & 0 & 1 \end{bmatrix}$$

The resulting transformed position vectors are

$$[X^*] = [X][T] = \begin{bmatrix} 4.5 & 4.5 & 0 & 0.95 \\ 5.5 & 4.5 & 0 & 0.95 \\ 5.5 & 5.5 & 0 & 0.95 \\ 4.5 & 5.5 & 0 & 0.95 \\ 4.5 & 4.5 & 0 & 1.05 \\ 5.5 & 4.5 & 0 & 1.05 \\ 5.5 & 5.5 & 0 & 1.05 \\ 4.5 & 5.5 & 0 & 1.05 \end{bmatrix} = \begin{bmatrix} 4.737 & 4.737 & 0 & 1 \\ 5.789 & 4.737 & 0 & 1 \\ 5.789 & 5.789 & 0 & 1 \\ 4.737 & 5.789 & 0 & 1 \\ 4.286 & 4.286 & 0 & 1 \\ 5.238 & 4.286 & 0 & 1 \\ 5.238 & 5.238 & 0 & 1 \\ 4.286 & 5.238 & 0 & 1 \end{bmatrix}$$



**Figure 3-33** Scale effect of  $z$  translation for a single-point perspective projection.

The result is shown as the upper right hand object in Fig. 3-32.

If the original object is translated by 5 units in the  $x$ ,  $y$  and  $z$  directions and a single-point perspective projection onto the  $z = 0$  plane from a center of projection at  $z = z_c = 20$  is performed, then from Eq. (3-59) the concatenated transformation matrix is

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -0.05 \\ 5 & 5 & 0 & 0.75 \end{bmatrix}$$

Notice the overall scaling indicated by the value of 0.75 in the lower right hand element of the transformation matrix.

The resulting transformed position vectors are

$$[X^*] = [X][T] = \begin{bmatrix} 4.5 & 4.5 & 0 & 0.725 \\ 5.5 & 4.5 & 0 & 0.725 \\ 5.5 & 5.5 & 0 & 0.725 \\ 4.5 & 5.5 & 0 & 0.725 \\ 4.5 & 4.5 & 0 & 0.775 \\ 5.5 & 4.5 & 0 & 0.775 \\ 5.5 & 5.5 & 0 & 0.775 \\ 4.5 & 5.5 & 0 & 0.775 \end{bmatrix} = \begin{bmatrix} 6.207 & 6.207 & 0 & 1 \\ 7.586 & 6.207 & 0 & 1 \\ 7.586 & 7.586 & 0 & 1 \\ 6.207 & 7.586 & 0 & 1 \\ 5.806 & 5.806 & 0 & 1 \\ 7.097 & 5.806 & 0 & 1 \\ 7.097 & 7.097 & 0 & 1 \\ 5.806 & 7.097 & 0 & 1 \end{bmatrix}$$

The result is shown as the upper right hand object in Fig. 3-34.

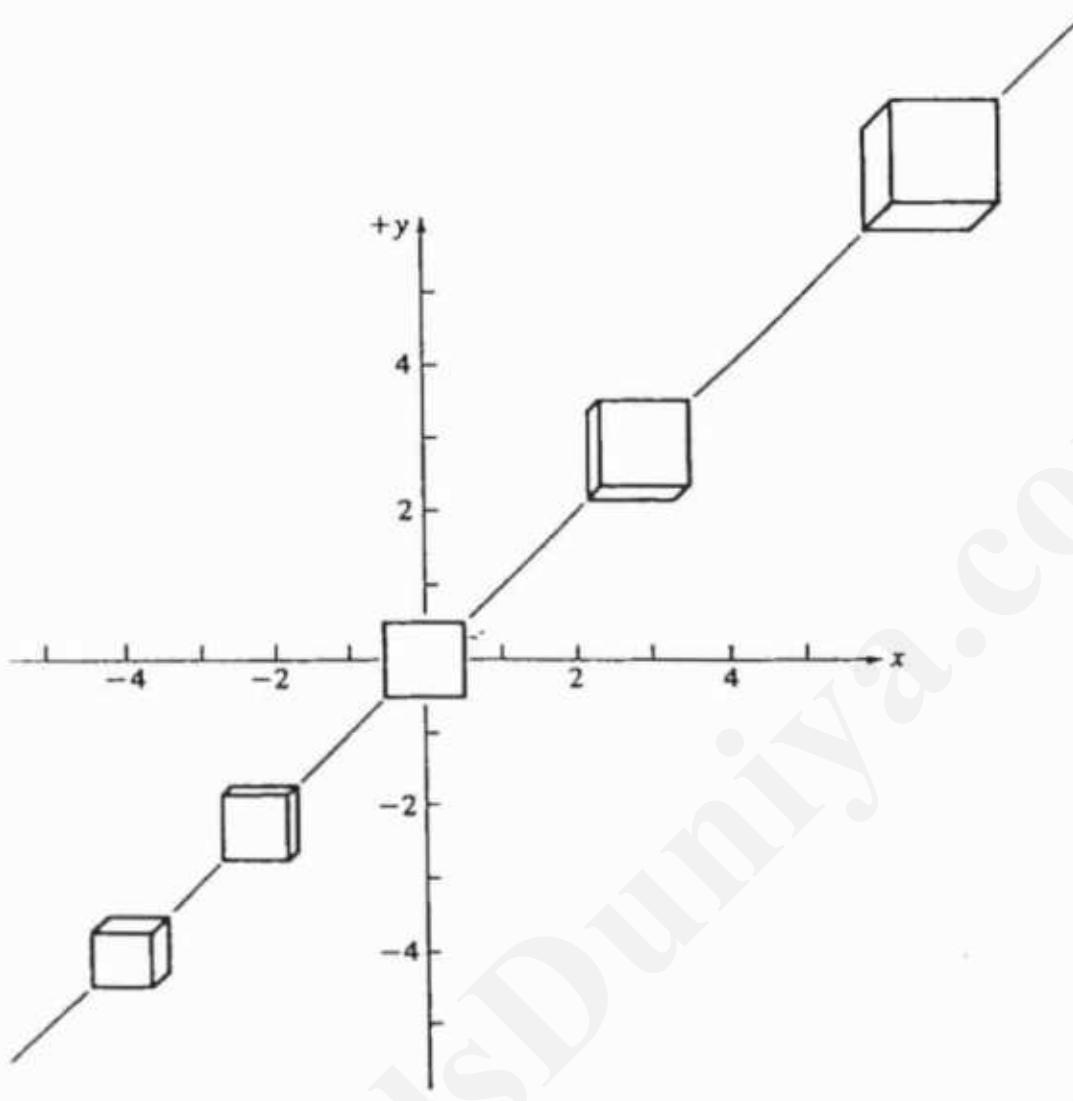


Figure 3-34 Single-point perspective projection with  $x$ ,  $y$  and  $z$  translations.

Multiple faces of an object are also revealed by rotation of the object. A single rotation reveals at least two faces of an object, while two or more rotations about separate axes reveal a minimum of three faces.

The transformation matrix for rotation about the  $y$ -axis by an angle  $\phi$ , followed by a single-point perspective projection onto the  $z = 0$  plane from a center of projection at  $z = z_c$ , is given by

$$\begin{aligned}
 [T] &= [R_y][P_{rz}] = \begin{bmatrix} \cos \phi & 0 & -\sin \phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin \phi & 0 & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} \cos \phi & 0 & 0 & \frac{\sin \phi}{z_c} \\ 0 & 1 & 0 & 0 \\ \sin \phi & 0 & 0 & -\frac{\cos \phi}{z_c} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-60)
 \end{aligned}$$

Similarly, the transformation matrix for rotation about the  $x$ -axis by an angle  $\theta$ , followed by a single-point perspective projection onto the  $z = 0$  plane from a center of projection at  $z = z_c$ , is

$$\begin{aligned} [T] &= [R_x][P_{rz}] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & 0 & -\frac{\sin\theta}{z_c} \\ 0 & -\sin\theta & 0 & -\frac{\cos\theta}{z_c} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-61) \end{aligned}$$

In both Eqs. (3-60) and (3-61) two of the perspective terms in the fourth column of the transformation matrix are nonzero. Thus, a single rotation about a principal axis perpendicular to that on which the center of projection lies is equivalent to a two-point perspective transformation. Rotation about the axis on which the center of projection lies does not have this effect. Notice that for a single rotation the perspective term for the axis of rotation is unchanged, e.g., in Eqs. (3-60) and (3-61)  $q$  and  $p$ , respectively, remain zero.

Rotation about a single principal axis does not in general reveal the minimum three faces for an adequate three-dimensional representation. In general, rotation about a single principal axis must be combined with translation along the axis to obtain an adequate three-dimensional representation. The next example illustrates this.

### Example 3-23 Two-Point Perspective Projection Using Rotation About a Single Principal Axis

Consider the cube shown in Fig. 3-35a rotated about the  $y$ -axis by  $\phi = 60^\circ$  to reveal the left-hand face and translated  $-2$  units in  $y$  to reveal the top face projected onto the  $z = 0$  plane from a center of projection at  $z = z_c = 2.5$ .

Using Eq. (3-38) with  $\phi = 60^\circ$ , Eq. (3-47) with  $z_c = 2.5$  and Eq. (3-14) with  $n = l = 0$ ,  $m = -2$  yields

$$\begin{aligned} [T] &= [R_y][Tr][P_{rz}] \\ &= \begin{bmatrix} 0.5 & 0 & -0.866 & 0 \\ 0 & 1 & 0 & 0 \\ 0.866 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -0.4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0.5 & 0 & 0 & 0.346 \\ 0 & 1 & 0 & 0 \\ 0.866 & 0 & 0 & -0.2 \\ 0 & -2 & 0 & 1 \end{bmatrix} \end{aligned}$$

The transformed position vectors are

$$\begin{aligned}
 [X^*] &= [X][T] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & 0 & 0 & 0.346 \\ 0 & 1 & 0 & 0 \\ 0.866 & 0 & 0 & -0.2 \\ 0 & -2 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 0.866 & -2 & 0 & 0.8 \\ 1.366 & -2 & 0 & 1.146 \\ 1.366 & -1 & 0 & 1.146 \\ 0.866 & -1 & 0 & 0.8 \\ 0 & -2 & 0 & 1 \\ 0.5 & -2 & 0 & 1.346 \\ 0.5 & -1 & 0 & 1.346 \\ 0 & -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1.083 & -2.5 & 0 & 1 \\ 1.192 & -1.745 & 0 & 1 \\ 1.192 & -0.872 & 0 & 1 \\ 1.083 & -1.25 & 0 & 1 \\ 0 & -2 & 0 & 1 \\ 0.371 & -1.485 & 0 & 1 \\ 0.371 & -0.743 & 0 & 1 \\ 0 & -1 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

The result is shown in Fig. 3-35b. The distortion is the result of the center of projection being very close to the cube. Notice the convergence of lines originally parallel to the  $x$  and  $z$  axes to vanishing points that lie on the  $x$ -axis. These vanishing points are determined in Ex. 3-25 in Sec. 3-17.

Similarly, a three-point perspective transformation is obtained by rotating about two or more of the principal axes and then performing a single-point perspective transformation. For example, rotation about the  $y$ -axis followed by rotation about the  $x$ -axis and a perspective projection onto the  $z = 0$  plane from a center of projection at  $z = z_c$  yields the concatenated transformation matrix

$$\begin{aligned}
 [T] &= [R_y][R_x][P_{rz}] \\
 &= \begin{bmatrix} \cos\phi & 0 & -\sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} \cos\phi & \sin\phi\sin\theta & 0 & \frac{\sin\phi\cos\theta}{z_c} \\ 0 & \cos\theta & 0 & -\frac{\sin\theta}{z_c} \\ \sin\phi & -\cos\phi\sin\theta & 0 & -\frac{\cos\phi\cos\theta}{z_c} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-62)
 \end{aligned}$$

Notice the three nonzero perspective terms. The object may also be translated. If translation occurs after rotation, then the resulting concatenated transformation

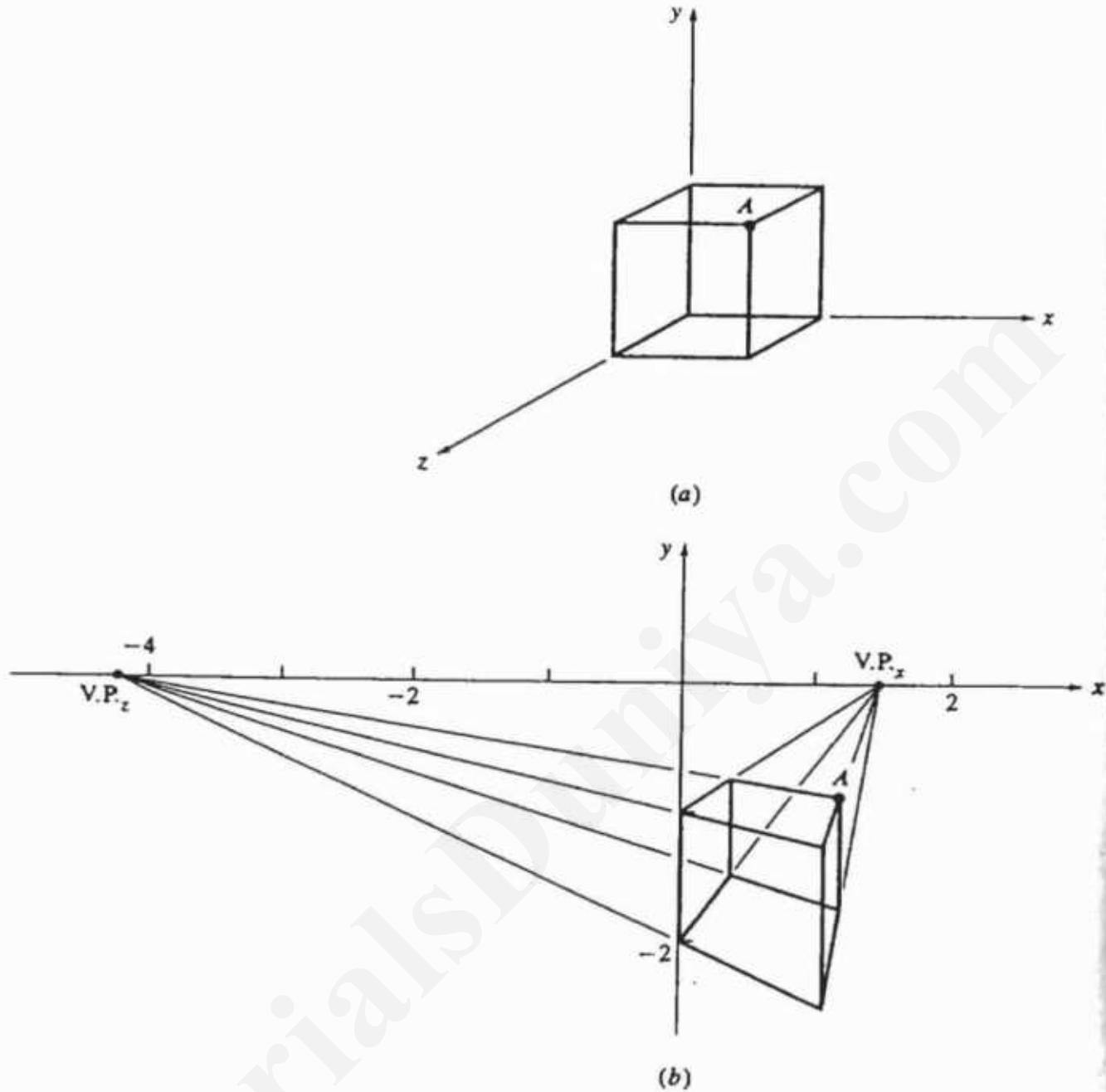


Figure 3-35 Two-point perspective projection with rotation about a single axis.

matrix is

$$\begin{aligned}
 [T] &= [R_y][R_x][Tr][P_{rz}] \\
 &= \begin{bmatrix} \cos\phi & \sin\phi\sin\theta & 0 & \frac{\sin\phi\cos\theta}{z_c} \\ 0 & \cos\theta & 0 & -\frac{\sin\theta}{z_c} \\ \sin\phi & -\cos\phi\sin\theta & 0 & -\frac{\cos\phi\cos\theta}{z_c} \\ l & m & 0 & 1 - \frac{n}{z_c} \end{bmatrix} \quad (3-63)
 \end{aligned}$$

Here, note the apparent scaling effect of translation in  $z$ . If the order of the rotations is reversed or if translation occurs before rotation, the results are different.

**Example 3-24 Three-Point Perspective Projection with Rotation About Two Axes**

Consider the cube shown in Fig. 3-35a rotated about the  $y$ -axis by  $\phi = -30^\circ$ , about the  $x$ -axis by  $\theta = 45^\circ$  and projected onto the  $z = 0$  plane from a center of projection at  $z = z_c = 2.5$ .

Using Eq. (3-62) yields

$$[T] = \begin{bmatrix} 0.866 & -0.354 & 0 & -0.141 \\ 0 & 0.707 & 0 & -0.283 \\ -0.5 & -0.612 & 0 & -0.245 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The transformed position vectors are

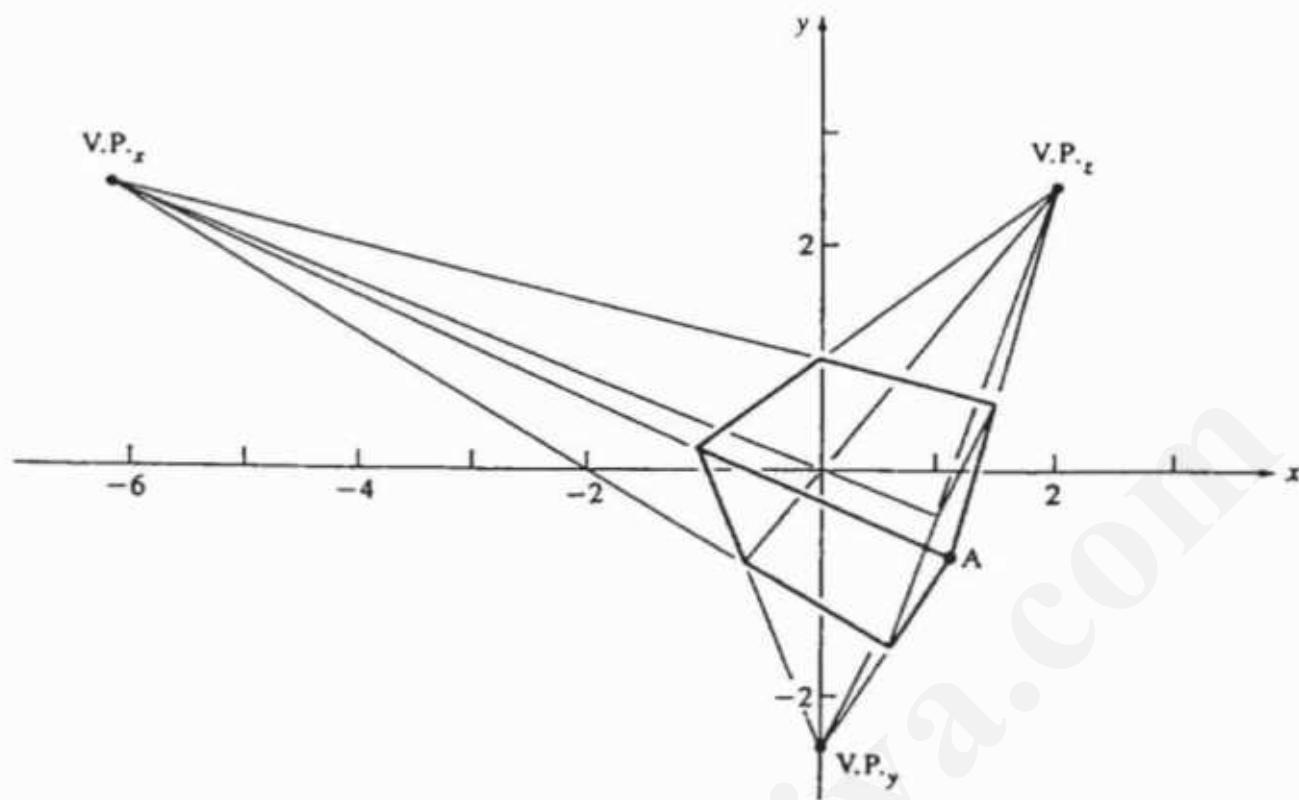
$$\begin{aligned} [X^*] &= [X][T] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.866 & -0.354 & 0 & -0.141 \\ 0 & 0.707 & 0 & -0.283 \\ -0.5 & -0.612 & 0 & -0.245 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} -0.5 & -0.612 & 0 & 0.755 \\ 0.366 & -0.966 & 0 & 0.614 \\ 0.366 & -0.259 & 0 & 0.331 \\ -0.5 & 0.095 & 0 & 0.472 \\ 0 & 0 & 0 & 1 \\ 0.866 & -0.354 & 0 & 0.859 \\ 0.866 & 0.354 & 0 & 0.576 \\ 0 & 0.707 & 0 & 0.717 \end{bmatrix} = \begin{bmatrix} -0.662 & -0.811 & 0 & 1 \\ 0.596 & -1.574 & 0 & 1 \\ 1.107 & -0.782 & 0 & 1 \\ -1.059 & 0.201 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1.009 & -0.412 & 0 & 1 \\ 1.504 & 0.614 & 0 & 1 \\ 0 & 0.986 & 0 & 1 \end{bmatrix} \end{aligned}$$

The result is shown in Fig. 3-36.

From these results it is clear that one-, two- or three-point perspective transformations can be constructed using rotations and translations about and along the principal axes, followed by a single-point perspective transformation from a center of projection on one of the principal axes. These results also follow for rotation about a general axis in space. Consequently, in implementing a graphics system using a fixed center of projection object manipulation paradigm, it is only necessary to provide for a single-point perspective projection onto the  $z = 0$  plane from a center of projection on the  $z$ -axis.

### 3-17 VANISHING POINTS

When a perspective view of an object is created a horizontal reference line, normally at eye level, as shown in Fig. 3-37a is used. Principal vanishing points



**Figure 3-36** Three-point perspective projection with rotation about two axes.

are points on the horizontal reference line at which lines originally parallel to the untransformed principal axes converge. In general, different sets of parallel lines have different principal vanishing points. This is illustrated in Fig. 3-37b. For planes of an object which are tilted relative to the untransformed principal axes, the vanishing points fall above or below the horizontal reference line. These are often called trace points, as shown in Fig. 3-37c.

Two methods for determining vanishing points are of general interest. The first simply calculates the intersection point of a pair of transformed projected parallel lines. The second is more complex but numerically more accurate. Here, an object with sides originally parallel to the principal axes is transformed to the desired position and orientation. A single-point perspective projection is applied. The final concatenated transformation matrix (see Eq. 3-63) is then used to transform the points at infinity on the principal axes. The resulting ordinary coordinates are the principal vanishing points for that object. For trace points resulting from inclined planes, the points at infinity in the directions of the edges of the inclined plane are first found and then transformed.

Several examples illustrate these techniques. The first uses intersection of the transformed lines to find the vanishing points.

#### Example 3-25 Principal Vanishing Points by Line Intersection

Recalling Ex. 3-23, the transformed position vectors for the pair of line segments with one line through the point  $A$  (see Fig. 3-35a) originally parallel to

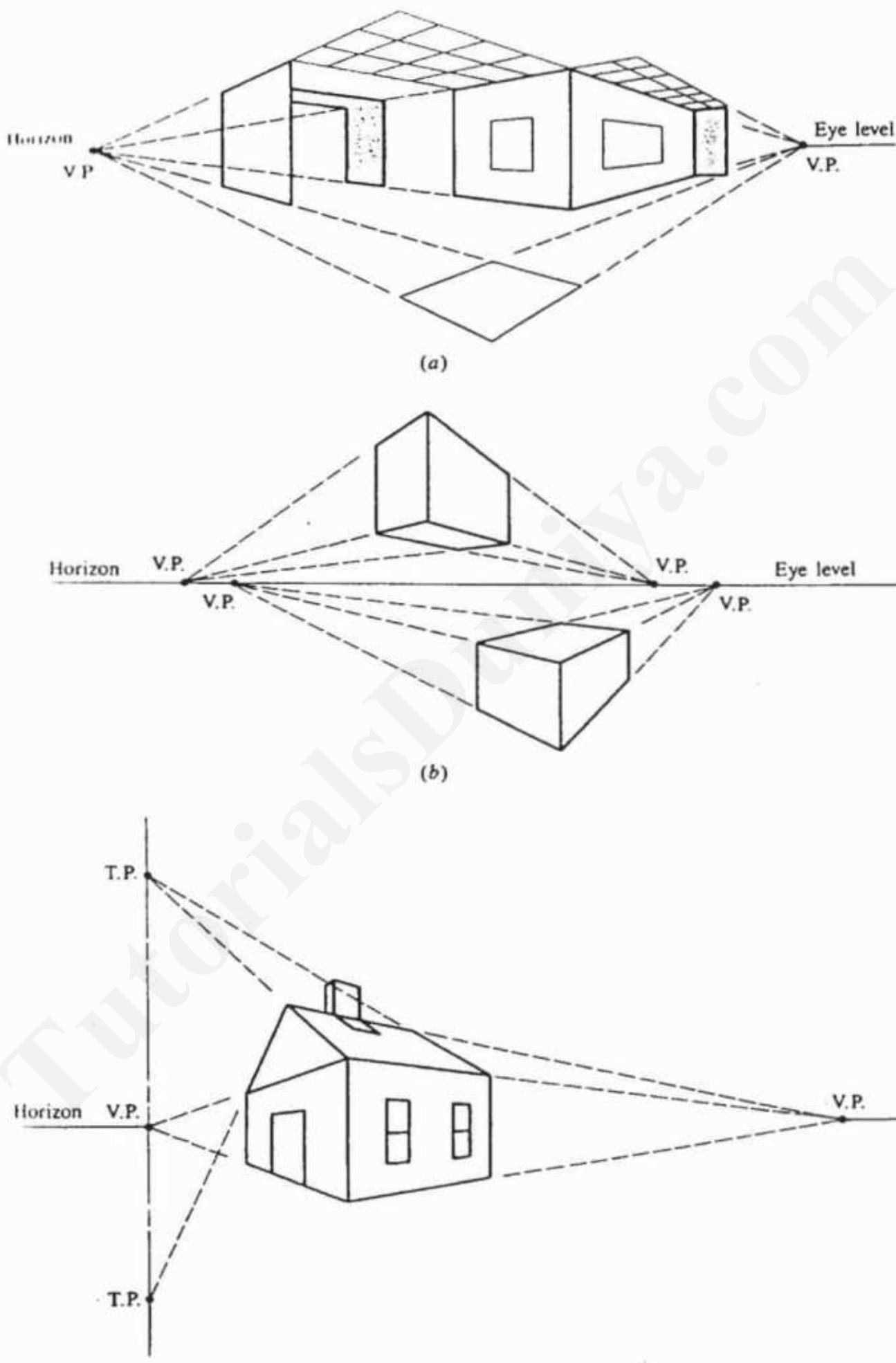


Figure 3-37 Trace points and vanishing points.

the  $x$ - and  $z$ -axes, respectively, are

$$\begin{array}{l} \textcircled{4} \left[ \begin{matrix} 1.083 & -1.25 & 0 & 1 \end{matrix} \right] \\ \textcircled{5} \left[ \begin{matrix} 1.192 & -0.872 & 0 & 1 \end{matrix} \right] \\ \textcircled{6} \left[ \begin{matrix} 0 & -1 & 0 & 1 \end{matrix} \right] \\ \textcircled{7} \left[ \begin{matrix} 0.371 & 0.743 & 0 & 1 \end{matrix} \right] \end{array} \quad \text{and} \quad \begin{array}{l} \left[ \begin{matrix} 1.192 & -0.872 & 0 & 1 \end{matrix} \right] \textcircled{3} \\ \left[ \begin{matrix} 0.371 & -0.743 & 0 & 1 \end{matrix} \right] \textcircled{7} \\ \left[ \begin{matrix} 1.083 & -1.25 & 0 & 1 \end{matrix} \right] \textcircled{4} \\ \left[ \begin{matrix} 0 & -1 & 0 & 1 \end{matrix} \right] \textcircled{8} \end{array}$$

Here, the numbers in circles refer to the rows in the original and transformed data matrices given in Ex. 3-23. The equations of the pair of lines originally parallel to the  $x$ -axis are

$$y = 3.468x - 5.006$$

$$y = 0.693x - 1$$

Solution yields  $[ VP_x ] = [ 1.444 \ 0 ]$ .

The equations of the pair of lines originally parallel to the  $z$ -axis are

$$y = -0.157x - 0.685$$

$$y = -0.231x - 1$$

Solution yields  $[ VP_z ] = [ -4.333 \ 0 ]$ .

These vanishing points are shown in Fig. 3-35b.

The second example uses transformation of the points at infinity on the principal axes to find the vanishing point.

### Example 3-26 Principal Vanishing Points by Transformation

Recalling Ex. 3-24 the concatenated complete transformation was

$$[ T ] = \begin{bmatrix} 0.866 & -0.354 & 0 & -0.141 \\ 0 & 0.707 & 0 & -0.283 \\ -0.5 & -0.612 & 0 & -0.245 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transforming the points at infinity on the  $x$ -,  $y$ - and  $z$ -axes yields

$$[ VP ]_i^T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0.866 & -0.354 & 0 & -0.141 \\ 0 & 0.707 & 0 & -0.283 \\ -0.5 & -0.612 & 0 & -0.245 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} -6.142 & 2.5 & 0 & 1 \\ 0 & -2.5 & 0 & 1 \\ 2.04 & 2.5 & 0 & 1 \end{bmatrix}$$

These vanishing points are shown in Fig. 3-36.

This third example uses transformation of the points at infinity for skew planes to find trace points.

### Example 3-27 Trace Points by Transformation

Consider the simple triangular prism shown in Fig. 3-38a. The position vectors for the prism are

$$[X] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0.5 & 0.5 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0.5 & 0.5 & 0 & 1 \end{bmatrix}$$

Applying the concatenated transformation of Ex. 3-24 yields transformed position vectors

$$[X^*] = [X][T]$$

$$= \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0.5 & 0.5 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0.5 & 0.5 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.866 & -0.354 & 0 & -0.141 \\ 0 & 0.707 & 0 & -0.283 \\ -0.5 & -0.612 & 0 & -0.245 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

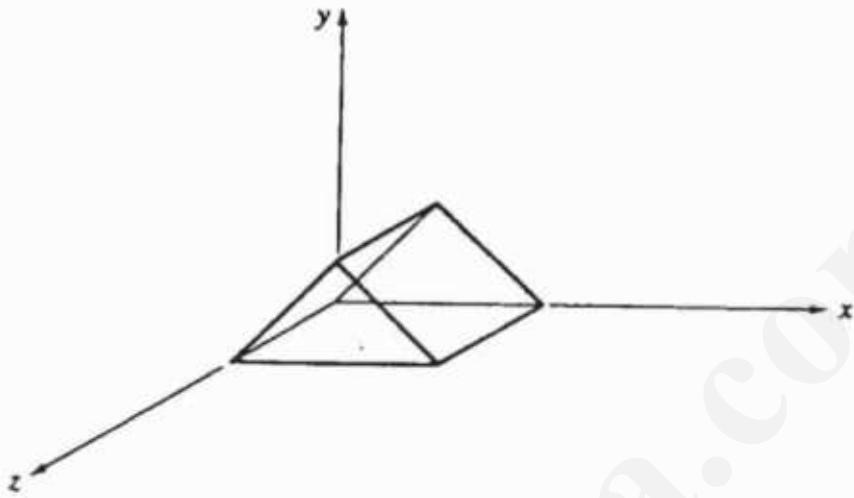
$$= \begin{bmatrix} -0.5 & -0.612 & 0 & 0.755 \\ 0.366 & -0.966 & 0 & 0.614 \\ -0.067 & -0.436 & 0 & 0.543 \\ 0 & 0 & 0 & 1 \\ 0.866 & -0.354 & 0 & 0.859 \\ 0.433 & 0.177 & 0 & 0.788 \end{bmatrix}$$

$$= \begin{bmatrix} -0.662 & -0.811 & 0 & 1 \\ 0.596 & -1.574 & 0 & 1 \\ -0.123 & -0.802 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1.009 & -0.412 & 0 & 1 \\ 0.55 & 0.224 & 0 & 1 \end{bmatrix}$$

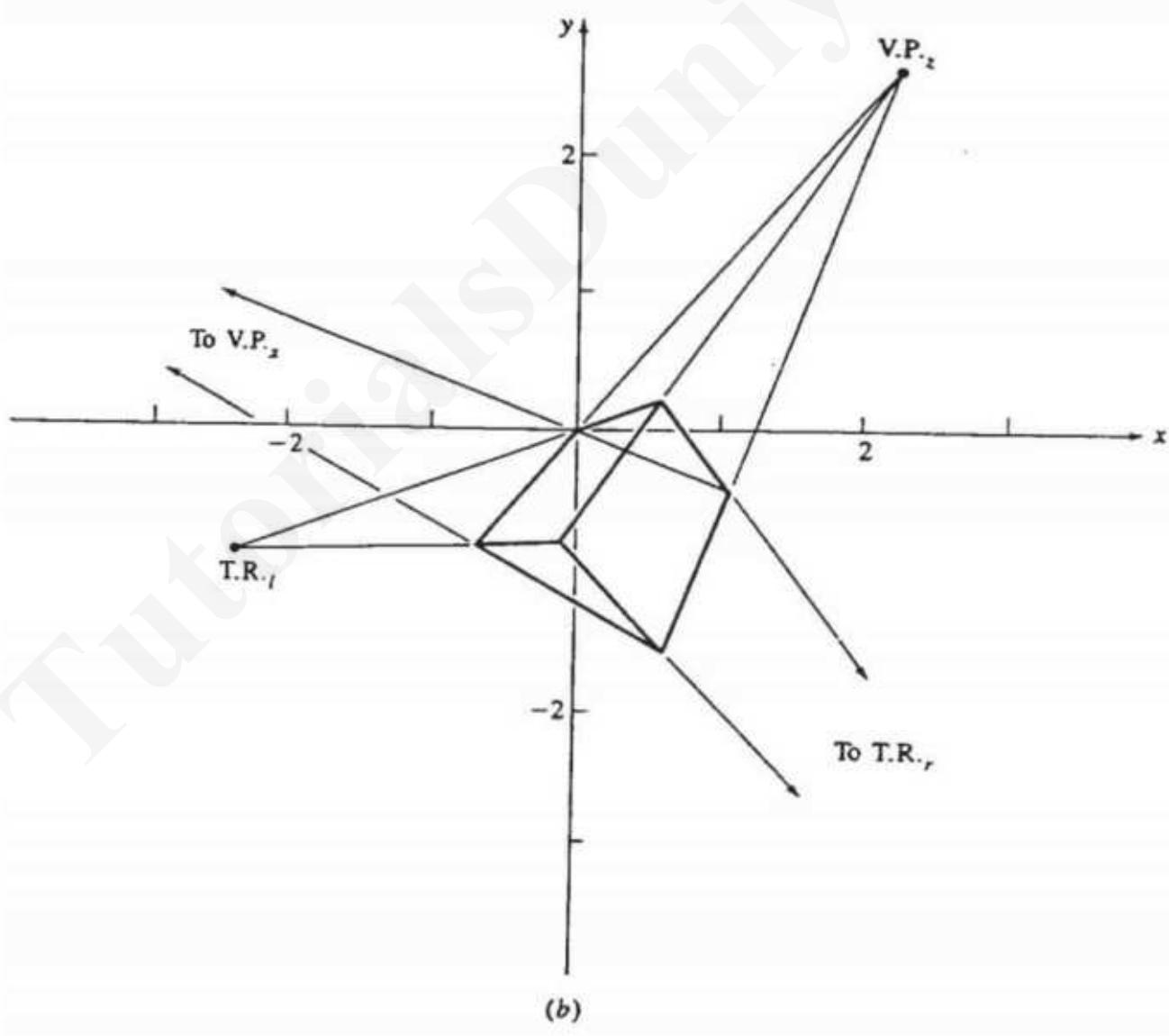
The transformed prism is shown in Fig. 3-38b.

The direction cosines for the inclined edges of the left-hand plane forming the top of the untransformed prism are  $[0.5 \ 0.5 \ 0]$ . Thus, the point at infinity in this direction is  $[1 \ 1 \ 0 \ 0]$ .

Similarly,  $[-0.5 \ 0.5 \ 0]$  are the direction cosines for the inclined edges of the right-hand plane forming the top of the untransformed prism. Thus, the point at infinity in this direction is  $[-1 \ 1 \ 0 \ 0]$ .



(a)



(b)

**Figure 3-38** Trace points.

Transforming these infinite points along with those for the principal axes yields

$$\begin{aligned}
 |VP|[T] &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ -1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0.866 & -0.354 & 0 & -0.141 \\ 0 & 0.707 & 0 & -0.283 \\ -0.5 & -0.612 & 0 & -0.245 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 0.866 & -0.354 & 0 & -0.141 \\ 0 & 0.707 & 0 & -0.283 \\ -0.5 & -0.612 & 0 & -0.245 \\ 0.866 & 0.354 & 0 & -0.424 \\ -0.866 & 1.061 & 0 & -0.141 \end{bmatrix} \\
 &= \begin{bmatrix} -6.142 & 2.5 & 0 & 1 \\ 0 & -2.5 & 0 & 1 \\ 2.041 & 2.5 & 0 & 1 \\ -2.041 & -0.833 & 0 & 1 \\ 6.142 & -7.5 & 0 & 1 \end{bmatrix} \begin{matrix} VP_x \\ VP_y \\ VP_z \\ TP_t \\ TP_r \end{matrix}
 \end{aligned}$$

The vanishing and trace points are also shown in Fig. 3-38b. Notice that as expected  $VP_x$ ,  $VP_y$  and  $VP_z$  are the same as found in Ex. 3-26.

## 1.18 PHOTOGRAPHY AND THE PERSPECTIVE TRANSFORMATION

A photograph is a perspective projection. The general case for a pinhole camera is illustrated in Fig. 3-39. The center of projection is the focal point of the camera lens. It is convenient to consider the creation of the original photographic negative and of a print from that negative as two separate cases.

Figure 3-40a illustrates the geometry for creation of the original negative. Here it is convenient to place the negative at the  $z = 0$  plane with the center of projection and the scene located in the negative half space  $z < 0$ . Perspective

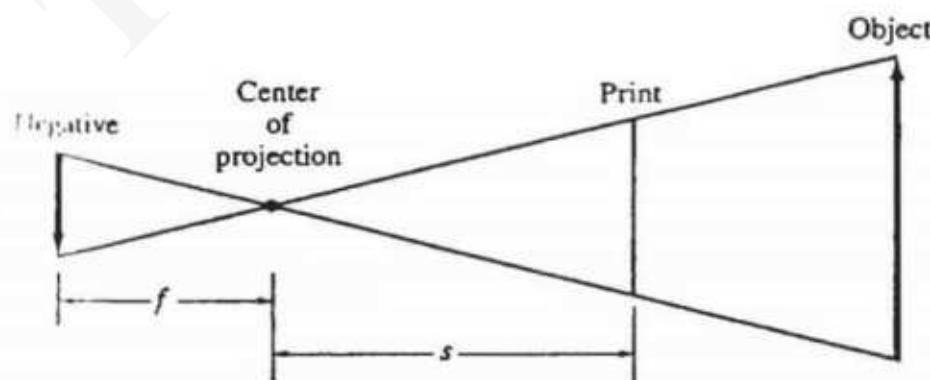
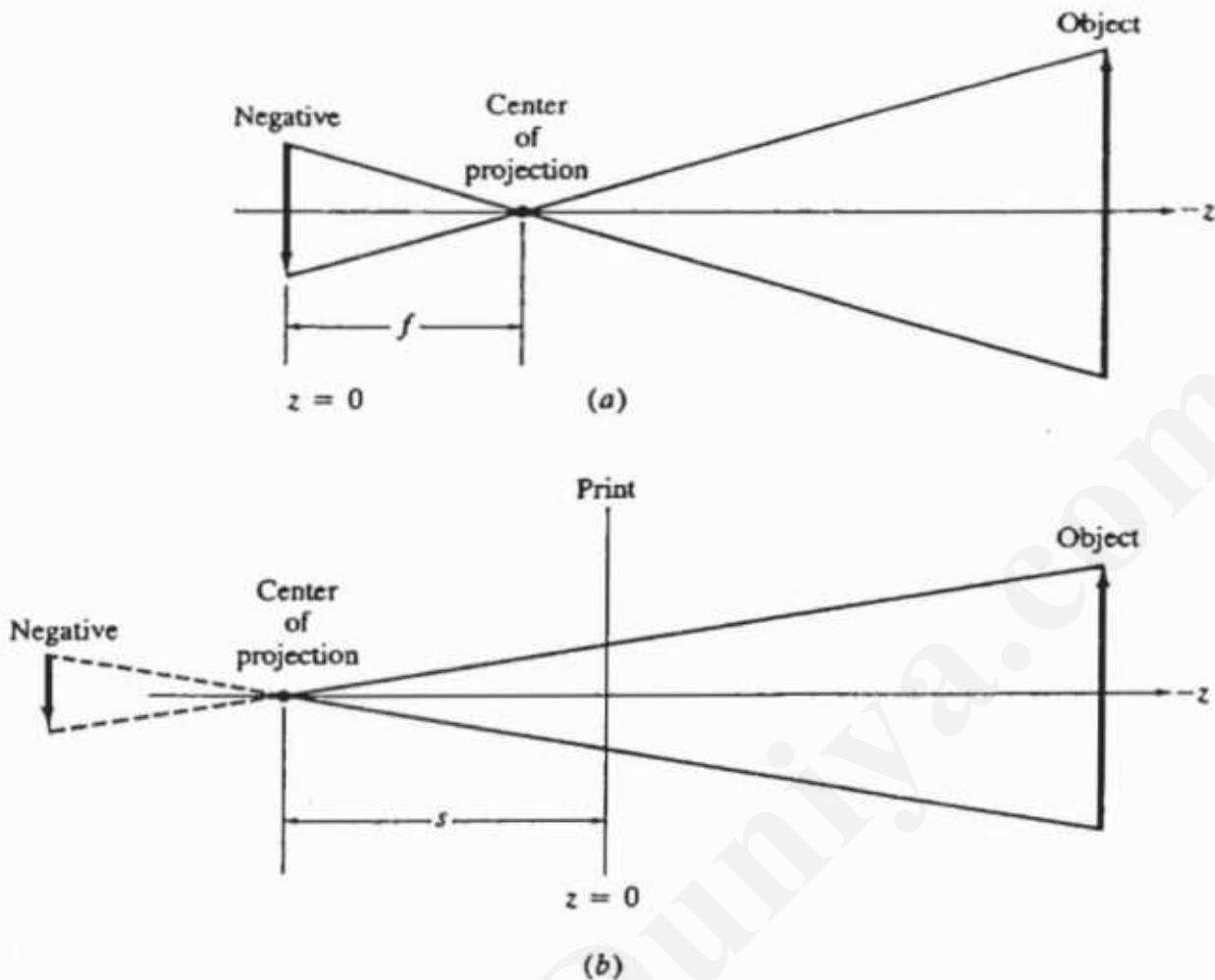


Figure 3-39 A photograph as a perspective projection.



**Figure 3-40** Photographic perspective geometry. (a) Creation of the original negative; (b) creation of a print.

projection onto the  $z = 0$  plane (the negative) yields the transformation

$$[T_n] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/f \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-64)$$

where  $f$  is the focal length of the lens. Note that an inverted image of the object is formed on the negative. Specifically,

$$[x \ y \ z \ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/f \\ 0 & 0 & 0 & 1 \end{bmatrix} = [x \ y \ 0 \ 1 + z/f]$$

and

$$x^* = \frac{xf}{f+z} \quad y^* = \frac{yf}{f+z}$$

Here, for  $f + z < 0$ ,  $x^*$  and  $y^*$  are of opposite sign to  $x$  and  $y$ , and an inverted image is formed on the negative.

Figure 3-40b illustrates the geometry for creation of a print from a photographic negative. Here  $s$  is the distance from the focal point of the enlarger lens to the paper. The paper is assumed to be located at  $z = 0$ . The perspective projection transformation is then

$$[T_p] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/s \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3-65)$$

Note that an upright image of the object is formed on the print. Specifically,

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/s \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x & y & z & 1 - \frac{z}{s} \end{bmatrix}$$

and

$$x^* = \frac{xs}{s-z} \quad y^* = \frac{ys}{s-z}$$

for the object  $z < 0$ ,  $s - z > 0$  and  $x^*y^*$  are of the same sign as  $x, y$ , i.e., an upright image is formed.

## 1.19 STEREOGRAPHIC PROJECTION

Increasing the perception of three-dimensional depth in a scene is important in many applications. There are two basic types of depth perception cues used by the eye-brain system: monocular and binocular, depending on whether they are apparent when one or two eyes are used. The principal monocular cues are:

Perspective — convergence of parallel lines.

Movement parallax — when the head is moved laterally, near objects appear to move more against a projection plane than far objects.

Relative size of known objects.

Overlap — a closer object overlaps and appears in front of a more distant object.

Highlights and shadows.

Atmospheric attenuation of, and the inability of the eye to resolve, fine detail in distant objects.

Focusing accommodation — objects at different distances require different tension in the focusing muscles of the eye.

The principal binocular cues are:

The convergence angles of the optical axes of the eyes.

**Retinal disparity** the different location of objects projected on the eye's retina is interpreted as differences in distance from the eye.

The monocular cues produce only weak perceptions of three-dimensional depth. However, because the eye-brain system fuses the two separate and distinct images produced by each eye into a single image, the binocular cues produce very strong three-dimensional depth perceptions. Stereography attempts to produce an image with characteristics analogous to those for true binocular vision. There are several techniques for generating stereo images (see Refs. 3-4 and 3-5). All depend upon supplying the left and right eyes with separate images.

Two methods, called chromatic anaglyphic and polarized anaglyphic, use filters to insure reception of correct and separate images by the left and right eyes. Briefly, the chromatic anaglyphic technique creates two images in two different colors, one for the left eye and one for the right eye. When viewed through corresponding filters the left eye sees only the left image and the right eye only the right image. The eye-brain system combines both two-dimensional images into a single three-dimensional image with the correct colors. The polarized anaglyphic method uses polarizing filters instead of color filters.

A third technique uses a flicker system to alternately project a left and a right eye view. An associated viewing device is synchronized to block the light to the opposite eye.

A fourth method, autostereoscopy, does not require any special viewing equipment. The method depends on the use of line or lenticular screens. The images are called parallax stereograms, parallax panoramagrams and panoramic parallax stereograms. The details are given in Ref. 3-5.

All of these techniques require projection of an object onto a plane from two different centers of projection, one for the right eye and one for the left eye. Figure 3-41 shows a projection of the point  $P$  onto the  $z = 0$  plane from centers of projection at  $E_L(-e, 0, d_e)$  and  $E_R(e, 0, d_e)$  corresponding to the left and right eye, respectively.

For convenience, the center of projection for the left eye is translated so that it lies on the  $z$ -axis as shown in Fig. 3-41b. Using similar triangles then yields

$$\frac{x_L^{**}}{d_e} = \frac{x'}{d_e - z}$$

and

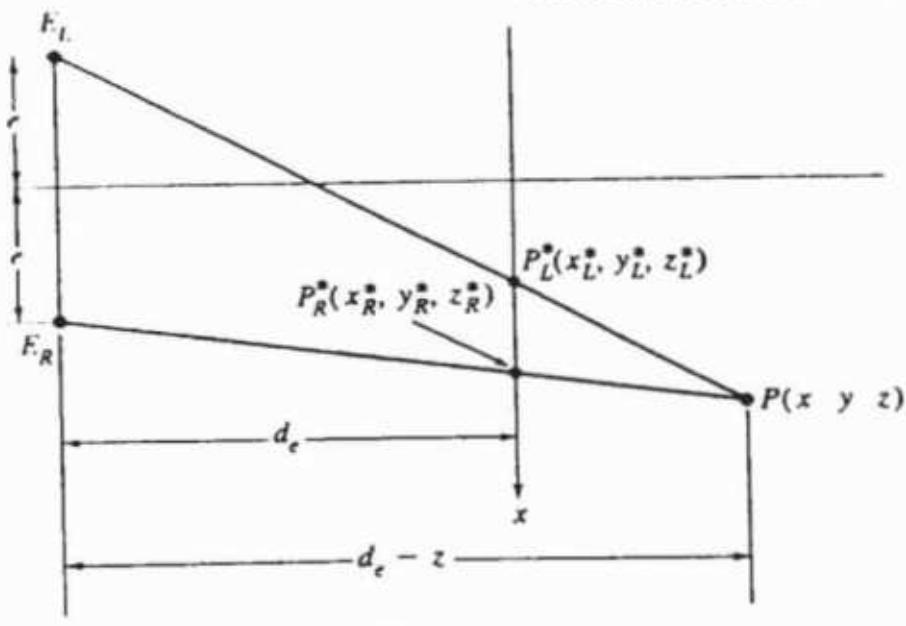
$$x_L^{**} = \frac{x'}{1 - z/d_e} = \frac{x'}{1 + rz}$$

where

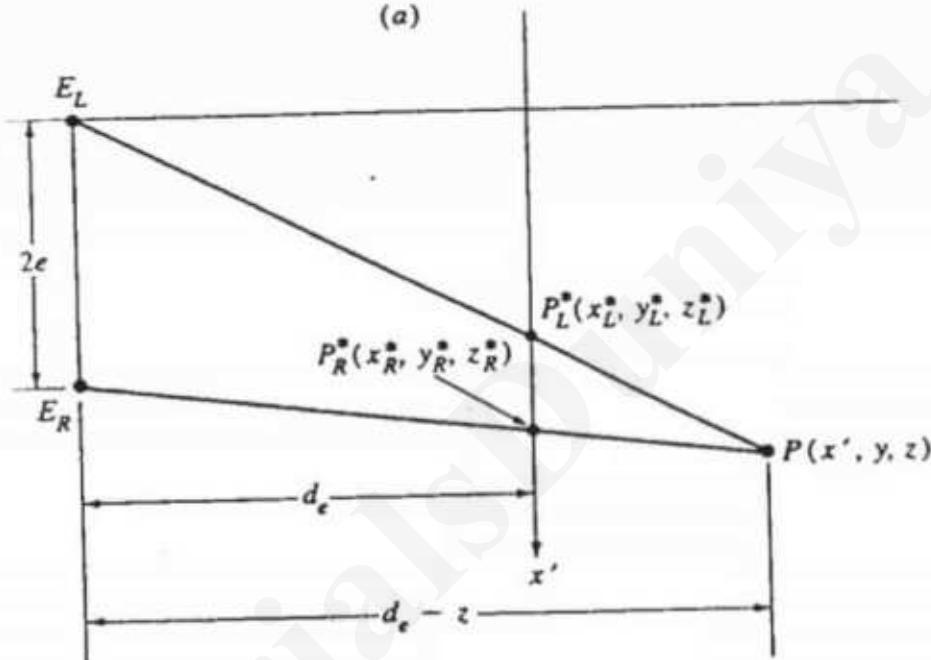
$$r = -1/d_e$$

Similarly, translating the center of projection for the right eye so that it lies on the  $z$ -axis as shown in Fig. 3-41c, and again using similar triangles, yields

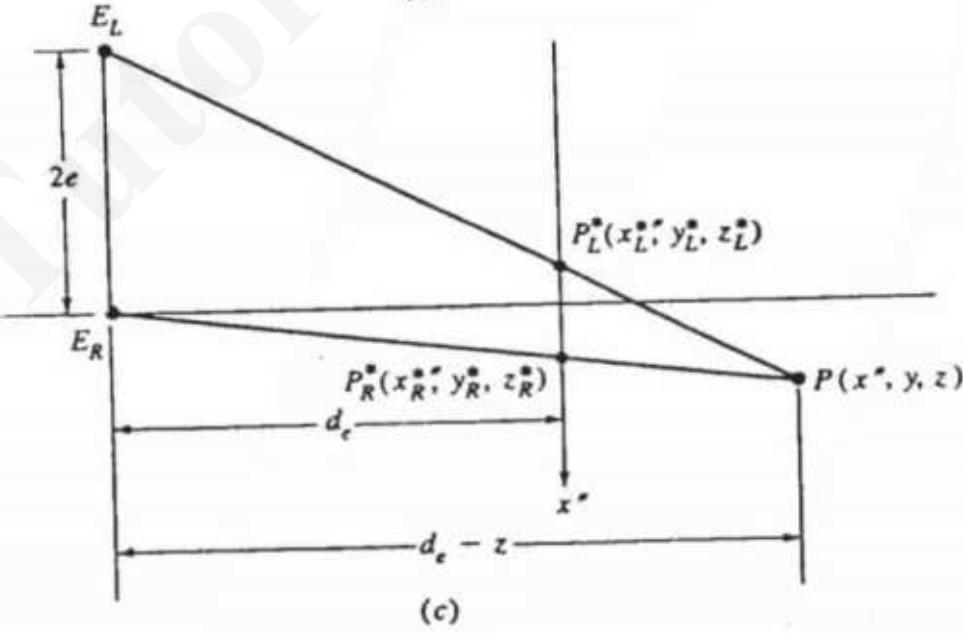
$$\frac{x_R^{**}}{d_e} = \frac{x''}{d_e - z}$$



(a)



(b)



(c)

Figure 3-41 Stereographic projection onto  $z = 0$ .

and

$$x_R'' = \frac{x''}{1 - z/d_e} = \frac{x''}{1 + rz}$$

Since each eye is at  $y = 0$ , the projected values of  $y$  are both

$$y^* = \frac{y}{1 - z/d_e} = \frac{y}{1 + rz}$$

The equivalent  $4 \times 4$  transformation matrices for the left and right eye views are

$$\begin{aligned} [S_L] &= [Tr_z][P_{rz}] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ e & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/d_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/d_e \\ e & 0 & 0 & 1 \end{bmatrix} \end{aligned} \quad (3-66)$$

and

$$[S_R] = [Tr_z][P_{rz}] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/d_e \\ -e & 0 & 0 & 1 \end{bmatrix} \quad (3-67)$$

Consequently, a stereographic projection is obtained by transforming the scene using Eqs. (3-66) and (3-67) and displaying both images.

Stereographic projections are displayed in a number of ways. For many individuals a stereo image can be created without any viewing aid. One technique, which takes a bit of practice, is to first focus the eyes at infinity; then, without changing the focus, gradually move the stereo pairs, held at about arm's length, into view.

Binocular fusion of the stereo pairs is improved by using a small opaque mask, e.g., a strip of black cardboard about an inch wide. As illustrated in Fig. 3-42, the mask is placed between the eyes and the stereo pair and moved back and forth until it is in the position shown. When the mask is in the position shown, the left eye sees only the left image and the right eye only the right image of the stereo pair.

Two more formal devices for viewing stereo pairs are shown in Figs. 3-43a and 3-43b. Figure 3-43a shows a Brewster stereopticon popular in the early part of this century. Figure 3-43b shows a typical modern laboratory stereoscope. Both devices are examples of simple focal plane lens stereoscopes.

Although Eqs. (3-66) and (3-67) provide the basic transformation for generating stereo pairs, it is necessary to modify this basic transformation to accommodate the geometry of various stereo viewing devices. The geometry for a simple focal plane stereoscope is shown in Fig. 3-44. Here the stereo pairs are

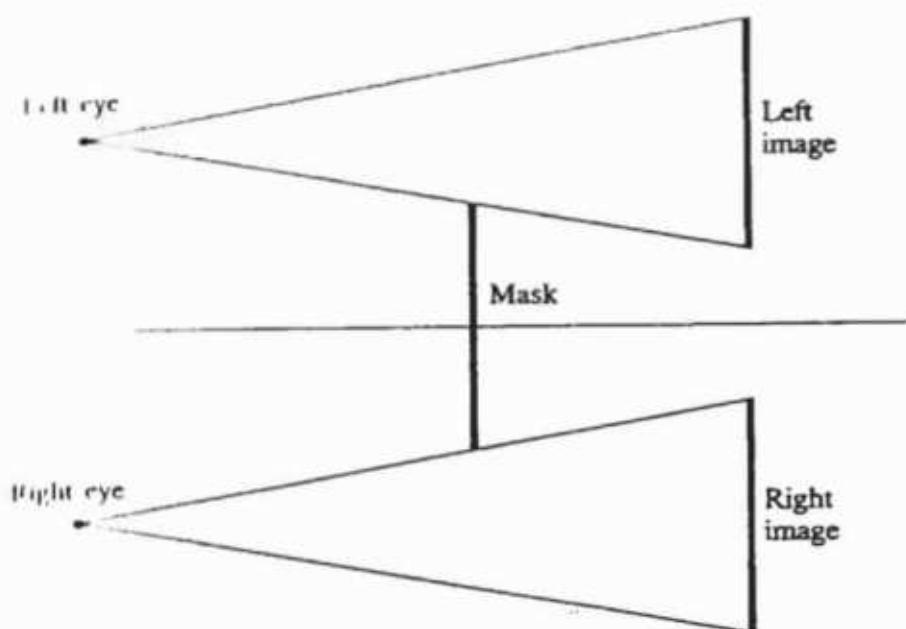
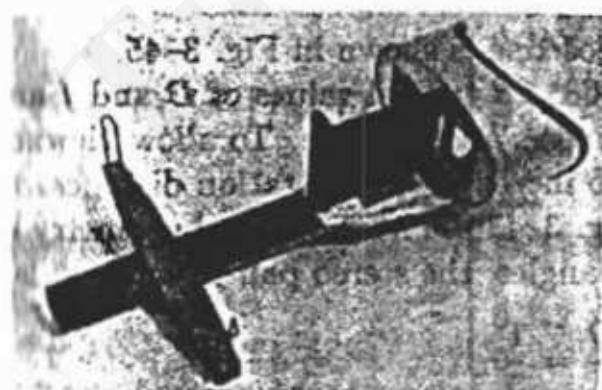


Figure 3-42 Simple stereo pair viewing method.

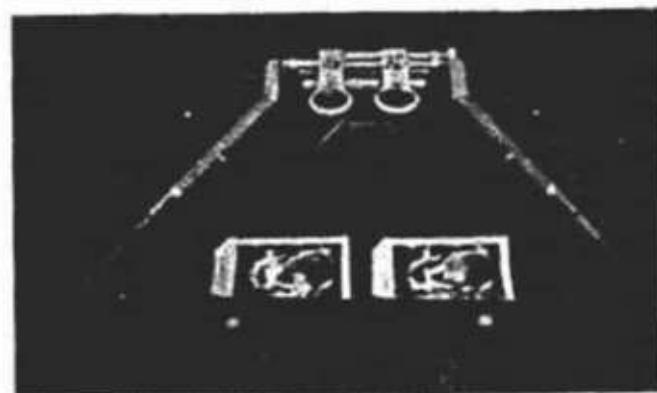
located at the focal distance  $f$  of the lenses. The stereo image is reconstructed at a distance  $I$  from the lenses. As mentioned previously, the stereo convergence angle  $\gamma$ , which is one of the strongest binocular fusion cues, is associated with the convergence of the optical axes of the eyes. For a normal human the inter-pupil distance of the eyes is about 60 millimeters. Experiments (Ref. 3-4) have shown that the strongest stereo effect occurs at a normal viewing distance of about 600 millimeters. Consequently, the strongest stereo effect occurs for  $\tan(\gamma/2) = 1/5$ . If, as shown in Fig. 3-44, the stereo pairs are located at the focal distance  $f$ , they must be separated by an amount  $2w$  to achieve binocular fusion.

From Fig. 3-44 with  $2D$  as the distance between the lenses of the stereoscope, similar triangles show that

$$\frac{D - w}{f} = \frac{D}{I} = \frac{1}{5}$$

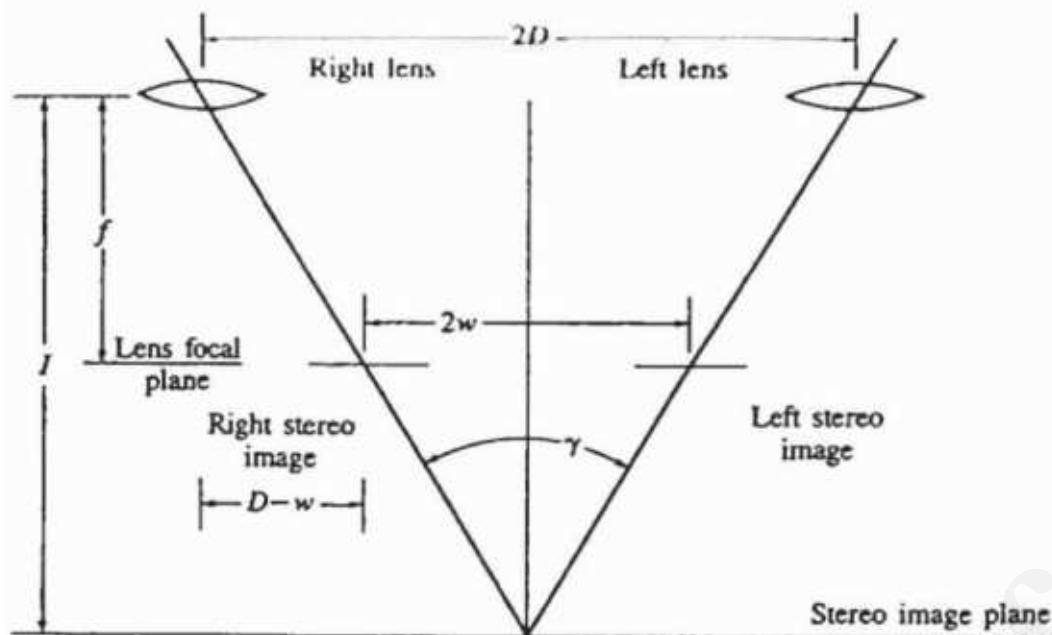


(a)



(b)

Figure 3-43 Stereoscopes. (a) Brewster stereopticon; (b) typical laboratory instrument.



**Figure 3-44** Geometry for a focal plane stereoscope.

Thus, a separation of

$$w = D - \frac{f}{5} \quad (3-68)$$

yields the strongest stereo effect. For a stereo pair to be viewed with a simple lens stereoscope, Eqs. (3-66) and (3-67) become

$$[S_L] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/f \\ w & 0 & 0 & 1 \end{bmatrix} \quad (3-69a)$$

and

$$[S_R] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/f \\ -w & 0 & 0 & 1 \end{bmatrix} \quad (3-69b)$$

A typical stereo pair for a simple lens stereoscope is shown in Fig. 3-45.

The separations obtained from Eq. (3-68) for typical values of  $D$  and  $f$  are quite small. Thus, only small stereo pair images are viewable. To allow viewing larger images, mirrors or prisms are used to increase the separation distance. A typical mirrored stereoscope is shown in Fig. 3-43b. The associated geometry is shown in Fig. 3-46. Again using similar triangles, the stereo pair separation is

$$\frac{w_0 - w}{f} = \frac{D}{I} = \frac{1}{5}$$

and

$$w = \frac{w_0 - f}{5} \quad (3-70)$$

An example illustrates the generation of a typical stereo pair.

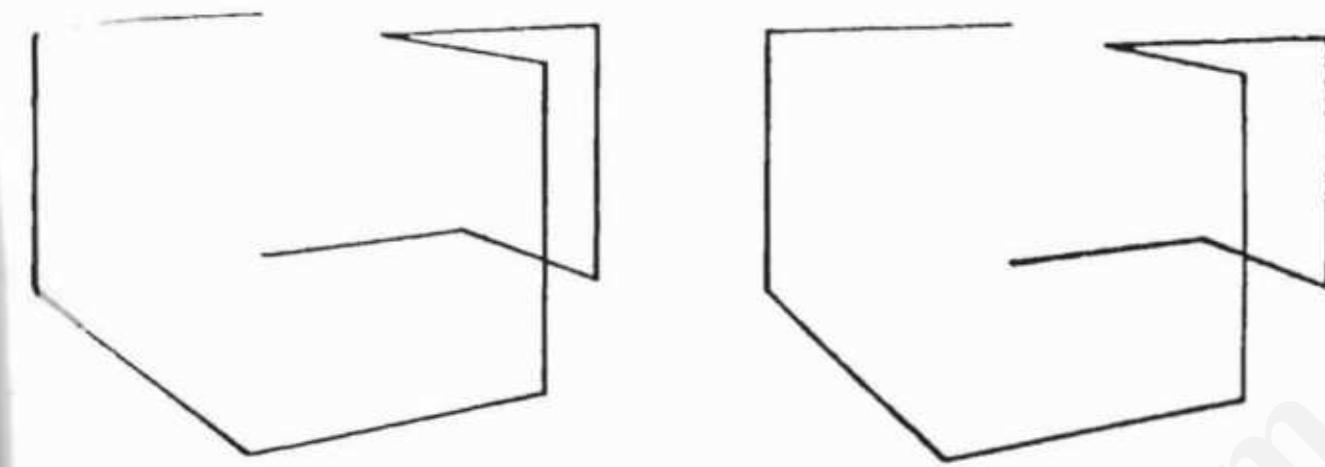


Figure 3-45 Stereo pair.

### Example 3-28 Stereo Pair Generation

Consider a simple three-dimensional wire frame image with position vectors

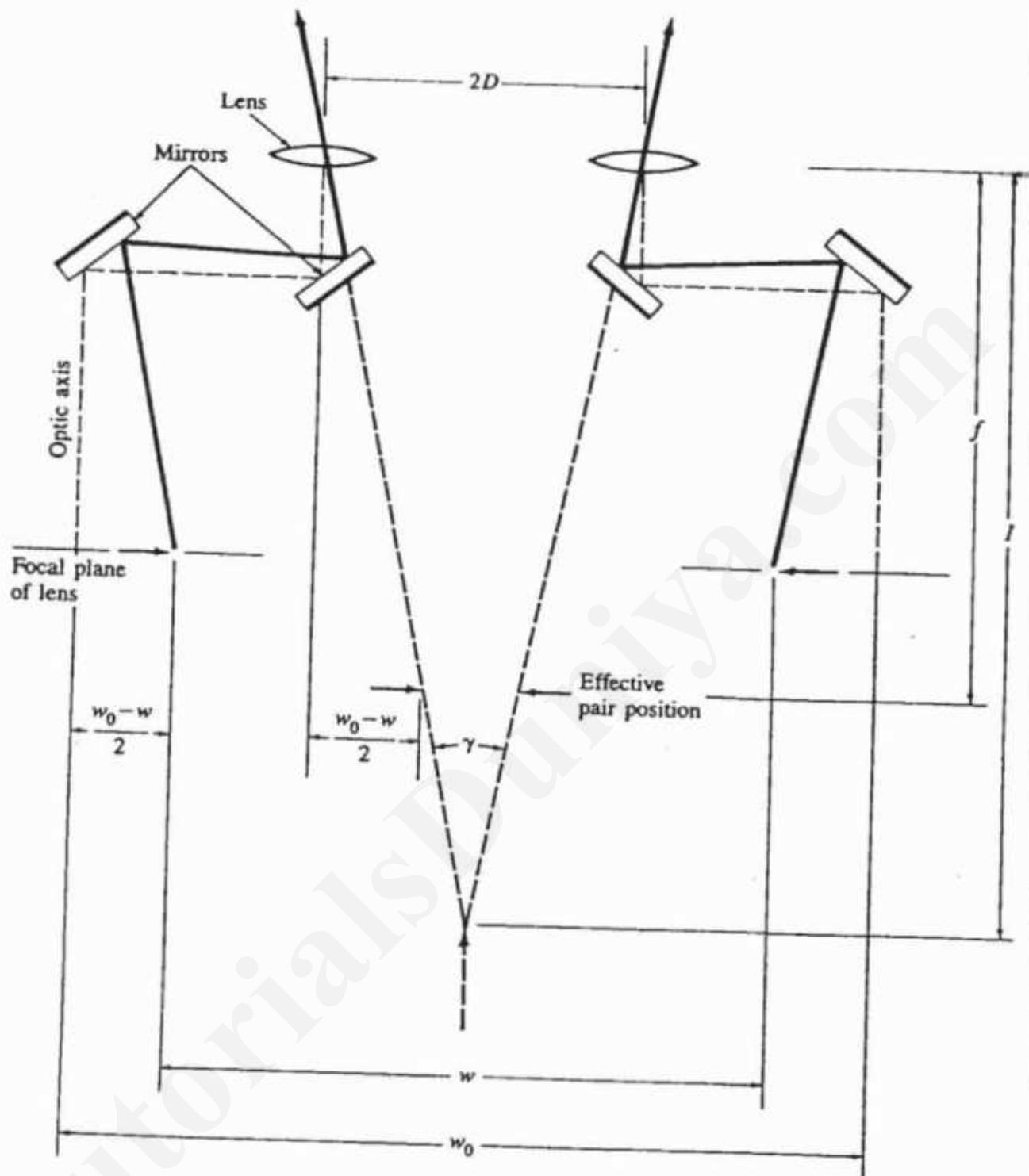
$$[X] = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 2 & 1 \\ 0 & 0 & 2 & 1 \\ -1 & 0 & 2 & 1 \\ -1 & 0 & 0 & 1 \\ -1 & 0 & 0 & 1 \\ -1 & 1 & 0 & 1 \end{bmatrix}$$

as shown in Fig. 3-45. The figure is first rotated about the  $y$ -axis by  $20^\circ$  and then translated  $-1.5$  units in the  $z$  direction for viewing purposes. The resulting transformation is

$$\begin{aligned} [V] &= [R_y][T_z] = \begin{bmatrix} 0.94 & 0 & -0.342 & 0 \\ 0 & 1 & 0 & 0 \\ 0.342 & 0 & 0.94 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1.5 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0.94 & 0 & -0.342 & 0 \\ 0 & 1 & 0 & 0 \\ 0.342 & 0 & 0.94 & 0 \\ 0 & 0 & -1.5 & 1 \end{bmatrix} \end{aligned}$$

A stereo pair is constructed for viewing through a simple lens focal plane stereoscope. The lens separation  $2D = 4$  inches and the focal length is also 4 inches. From Eq. (3-68) the stereo pair separation is

$$w = 2 - 4/5 = 1.2''$$



**Figure 3-46** Geometry for a mirrored stereoscope.

Using Eq. (3-69), the right and left stereo image transformations are

$$[S_L] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -0.25 \\ 1.2 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad [S_R] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -0.25 \\ -1.2 & 0 & 0 & 1 \end{bmatrix}$$

The combined viewing and left and right image transformations are

$$[C_L] = [V][S_L] \quad \text{and} \quad [C_R] = [V][S_R]$$

$$[C_L] = \begin{bmatrix} 0.94 & 0 & 0 & 0.086 \\ 0 & 1 & 0 & 0 \\ 0.342 & 0 & 0 & -0.235 \\ 1.2 & 0 & 0 & 1.38 \end{bmatrix}$$

and

$$[C_R] = \begin{bmatrix} 0.94 & 0 & 0 & 0.086 \\ 0 & 1 & 0 & 0 \\ 0.342 & 0 & 0 & -0.235 \\ -1.2 & 0 & 0 & 1.375 \end{bmatrix}$$

The transformed position vectors for the left and right images are

$$[X_L^*] = [X][C_L] \quad \text{and} \quad [X_R^*] = [X][C_R]$$

$$[X_L^*] = \begin{bmatrix} 0.873 & 0 & 0 & 1 \\ 1.465 & 0 & 0 & 1 \\ 2.025 & 0 & 0 & 1 \\ 2.025 & 0.816 & 0 & 1 \\ 1.353 & 0.877 & 0 & 1 \\ 2.081 & 1.105 & 0 & 1 \\ 2.081 & 0 & 0 & 1 \\ 1.152 & 0 & 0 & 1 \\ 0.202 & 0 & 0 & 1 \\ 0.202 & 0 & 0 & 1 \\ 0.202 & 0.775 & 0 & 1 \end{bmatrix} \quad \text{and} \quad [X_R^*] = \begin{bmatrix} -0.873 & 0 & 0 & 1 \\ -0.178 & 0 & 0 & 1 \\ 0.067 & 0 & 0 & 1 \\ 0.067 & 0.816 & 0 & 1 \\ -0.753 & 0.877 & 0 & 1 \\ -0.57 & 1.105 & 0 & 1 \\ -0.57 & 0 & 0 & 1 \\ -1.776 & 0 & 0 & 1 \\ -1.659 & 0 & 0 & 1 \\ -1.659 & 0 & 0 & 1 \\ -1.659 & 0.775 & 0 & 1 \end{bmatrix}$$

The result is shown in Fig. 3-45. Note that in generating the final image the results must be scaled for each particular output device to yield the correct physical dimensions for  $w$ .

### 3.20 COMPARISON OF OBJECT FIXED AND CENTER OF PROJECTION FIXED PROJECTIONS

An object fixed movable center of projection technique is easily converted to the movable object fixed center of projection technique previously discussed. There are two cases of interest. The first and simpler assumes that the projection plane is perpendicular to the sight vector from the center of projection into the scene. The second eliminates the perpendicular projection plane assumption. Only the first is discussed here.

When the projection plane is perpendicular to the sight vector, the following procedure yields the equivalent movable object fixed center of projection transformation:

Determine the intersection of the sight vector and the projection plane.

Translate the intersection point to the origin.

Rotate the sight vector so that it is coincident with the  $+z$ -axis and pointed towards the origin (see Sec. 3-9).

Apply the concatenated transformations to the scene.

Perform a single-point perspective projection onto the  $z = 0$  plane from the transformed center of projection on the  $z$ -axis.

A relatively simple example serves to illustrate the technique.

**Example 3-29 Object Fixed Perspective Projection onto a Perpendicular Projection Plane**

Consider the cube with one corner removed, previously discussed in Ex. 3-10. Project the cube from a center of projection at  $[10 \ 10 \ 10]$  onto the plane passing through the point  $[-1 \ -1 \ -1]$  and perpendicular to the sight vector as shown in Fig. 3-47.

The equation for the projection plane can be obtained from its normal (see Ref. 3-1 for alternate techniques). Here the normal is in the opposite direction to the sight vector.

The direction of the sight vector is given by

$$[s] = [-1 \ -1 \ -1]$$

The normal to the projection plane through  $[-1 \ -1 \ -1]$  perpendicular to the sight vector is then

$$[n] = [1 \ 1 \ 1]$$

The general form of a plane equation is

$$ax + by + cz + d = 0$$

The normal to the general plane is given by

$$[n] = [a \ b \ c]$$

The value of  $d$  in the plane equation is obtained from any point in the plane. The equation for the projection plane through  $[-1 \ -1 \ -1]$  is then

$$x + y + z + d = 0$$

and

$$d = -x - y - z = 1 + 1 + 1 = 3$$

Hence,

$$x + y + z + 3 = 0$$

is the equation for the projection plane.

The intersection of the sight vector and the projection plane is obtained by writing the parametric equation of the sight vector, substituting into the plane equation, and solving for the parameter value.

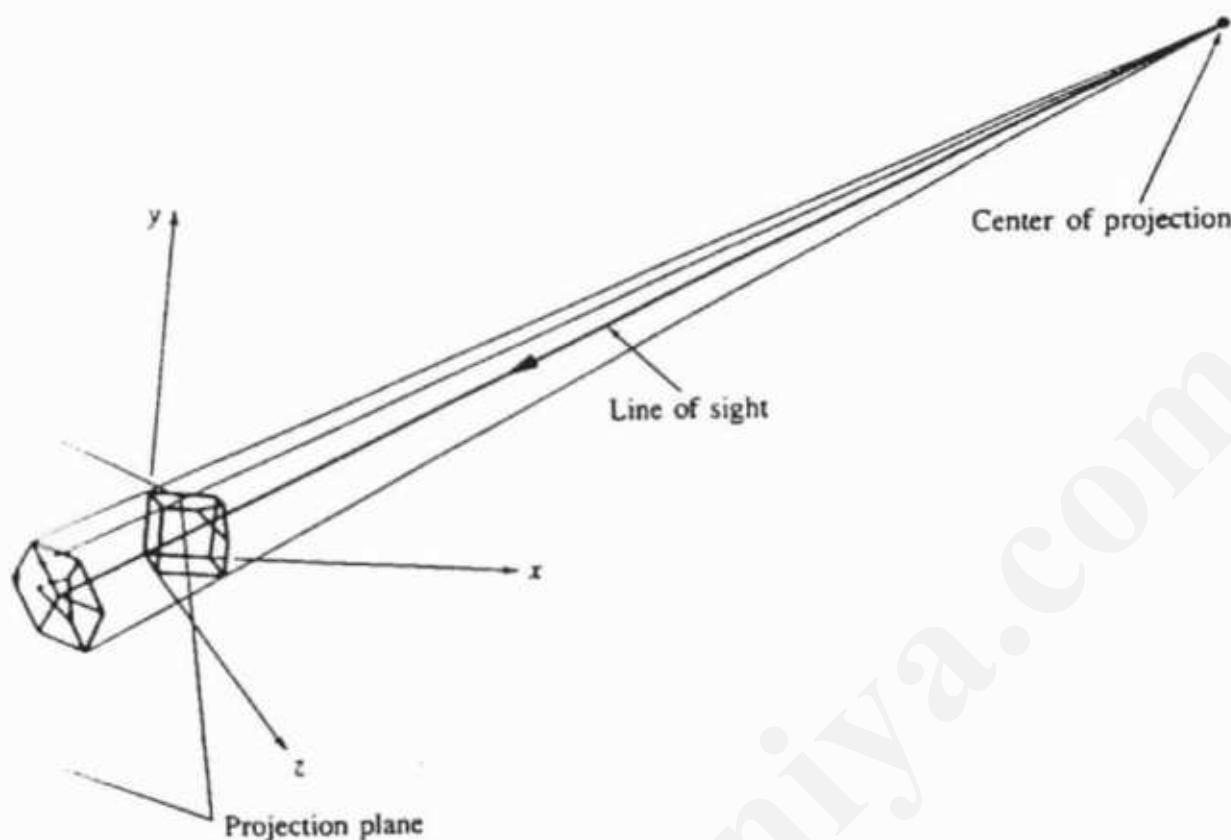


Figure 3.47 Perspective projection from movable center of projection.

The parametric equation of the sight vector is

$$\begin{aligned}[S(t)] &= [x(t) \quad y(t) \quad z(t)] \\ &= [10 \quad 10 \quad 10] + [-11 \quad -11 \quad -11]t \quad 0 \leq t \leq 1\end{aligned}$$

Substituting into the plane equation yields

$$x(t) + y(t) + z(t) + 3 = (10 - 11t) + (10 - 11t) + (10 - 11t) + 3 = 0$$

Solving for  $t$  yields the parameter value for the intersection point, i.e.,

$$-33t + 33 = 0 \rightarrow t = 1.0$$

The intersection is obtained by substituting  $t$  into  $[S(t)]$ . Specifically,

$$\begin{aligned}[I] &= [S(1)] = [10 \quad 10 \quad 10] + [-11 \quad -11 \quad -11](1.0) \\ &= [-1 \quad -1 \quad -1]\end{aligned}$$

The intersection point is at  $x = y = z = -1$  as expected from simple geometric considerations.

The required translation matrix is

$$[Tr] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

After translation the center of projection is at [ 11 11 11 ] and the sight vector passes through the origin.

Using the results of Sec. 3-9, a rotation about the  $x$ -axis of  $\alpha = 45^\circ$  followed by a rotation about the  $y$ -axis by  $\beta = 35.26^\circ$  makes the sight vector coincident with the  $z$ -axis. The rotation matrices are

$$[ R_x ] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & -1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad [ R_y ] = \begin{bmatrix} 2/\sqrt{6} & 0 & 1/\sqrt{3} & 0 \\ 0 & 1 & 0 & 0 \\ -1/\sqrt{3} & 0 & 2/\sqrt{6} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and the concatenated transformation matrix is

$$[ M ] = [ T_r ] [ R_x ] [ R_y ] = \begin{bmatrix} 2/\sqrt{6} & 0 & 1/\sqrt{3} & 0 \\ -1/\sqrt{6} & 1/\sqrt{2} & 1/\sqrt{3} & 0 \\ -1/\sqrt{6} & -1/\sqrt{2} & 1/\sqrt{3} & 0 \\ 0 & 0 & 3/\sqrt{3} & 1 \end{bmatrix}$$

Transforming the center of projection yields

$$\begin{aligned} [ C_p ] [ M ] &= [ 10 \ 10 \ 10 \ 1 ] \begin{bmatrix} 2/\sqrt{6} & 0 & 1/\sqrt{3} & 0 \\ -1/\sqrt{6} & 1/\sqrt{2} & 1/\sqrt{3} & 0 \\ -1/\sqrt{6} & -1/\sqrt{2} & 1/\sqrt{3} & 0 \\ 0 & 0 & 3/\sqrt{3} & 1 \end{bmatrix} \\ &= [ 0 \ 0 \ 33/\sqrt{3} \ 1 ] \end{aligned}$$

The transformation for a single-point perspective projection from a center of projection at  $z = 33/\sqrt{3}$  onto the  $z = 0$  plane is

$$[ P_{rz} ] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -\sqrt{3}/33 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Concatenation with  $[ M ]$  yields

$$\begin{aligned} [ T ] &= [ M ] [ P_{rz} ] = \begin{bmatrix} 2/\sqrt{6} & 0 & 1/\sqrt{3} & 0 \\ -1/\sqrt{6} & 1/\sqrt{2} & 1/\sqrt{3} & 0 \\ -1/\sqrt{6} & -1/\sqrt{2} & 1/\sqrt{3} & 0 \\ 0 & 0 & 3/\sqrt{3} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -\sqrt{3}/33 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 2/\sqrt{6} & 0 & 0 & -1/33 \\ -1/\sqrt{6} & 1/\sqrt{2} & 0 & -1/33 \\ -1/\sqrt{6} & -1/\sqrt{2} & 0 & -1/33 \\ 0 & 0 & 0 & 30/33 \end{bmatrix} \end{aligned}$$

The transformed ordinary coordinates of the projected object are

$$\begin{aligned}
 [X^*] &= [X][T] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0.5 & 1 & 1 \\ 0.5 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0.5 & 1 \end{bmatrix} \begin{bmatrix} 0.816 & 0 & 0 & -0.030 \\ -0.408 & 0.707 & 0 & -0.030 \\ -0.408 & -0.707 & 0 & -0.030 \\ 0 & 0 & 0 & 0.909 \end{bmatrix} \\
 &= \begin{bmatrix} -0.408 & -0.707 & 0 & 0.879 \\ 0.408 & -0.707 & 0 & 0.848 \\ 0.204 & -0.354 & 0 & 0.833 \\ -0.408 & 0 & 0 & 0.833 \\ -0.816 & 0 & 0 & 0.848 \\ 0 & 0 & 0 & 0.909 \\ 0.816 & 0 & 0 & 0.879 \\ 0.408 & 0.707 & 0 & 0.848 \\ -0.408 & 0.707 & 0 & 0.879 \\ 0.204 & 0.354 & 0 & 0.833 \end{bmatrix} \\
 &= \begin{bmatrix} -0.465 & -0.805 & 0 & 1 \\ 0.481 & -0.833 & 0 & 1 \\ 0.245 & -0.424 & 0 & 1 \\ -0.490 & 0 & 0 & 1 \\ -0.962 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0.929 & 0 & 0 & 1 \\ 0.481 & 0.833 & 0 & 1 \\ -0.465 & 0.805 & 0 & 1 \\ 0.245 & 0.424 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

The result is shown in Fig. 3-48.

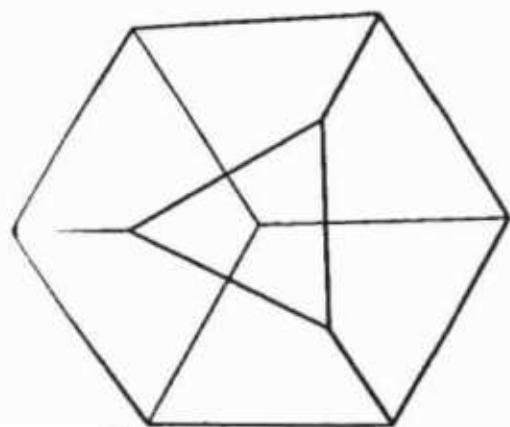


Figure 3-48 Result for Ex. 3-29.

### 3-21 RECONSTRUCTION OF THREE-DIMENSIONAL IMAGES

The reconstruction of a three-dimensional object or position in space is a common problem. For example, it occurs continuously in utilizing mechanical drawings which are orthographic projections. The method of reconstructing a three-dimensional object or position from two or more views (orthographic projections) given on a mechanical drawing is well known. However, the technique of reconstructing a three-dimensional position vector from two perspective projections, for example, two photographs, is not as well known. Of course, if the method is valid for perspective projections, then it is also valid for the simpler orthographic projections, and in fact for all the projections mentioned in previous sections. Further, as is shown below, if certain other information is available, then no direct knowledge about the transformation is required.

Before considering the more general problem we consider the special case of reconstruction of the three-dimensional coordinates of a point from two or more orthographic projections. Front, right-side and top orthographic views (projections) of an object are shown in Fig. 3-49. In determining the three-dimensional coordinates of point *A*, the front view yields values for *x* and *y*, the right-side view for *y* and *z*, and the top view for *x* and *z*, i.e.,

$$\begin{array}{ll} \text{front : } & x_f \quad y_f \\ \text{right side : } & y_r \quad z_r \\ \text{top : } & x_t \quad z_t \end{array}$$

Notice that two values are obtained for each coordinate. In any measurement system, in general  $x_f \neq x_t$ ,  $y_f \neq y_r$ ,  $z_r \neq z_t$ .<sup>†</sup> Since neither value is necessarily correct, the most reasonable solution is to average the values. Mathematically, the problem is said to be overspecified. Here only three independent values need be determined, but six conditions (equations) determining those values are available.

Turning now to reconstruction of three-dimensional coordinates from perspective projections, recall that the general perspective transformation is represented as a  $4 \times 4$  matrix. Thus,

$$[x \quad y \quad z \quad 1][T'] = [x' \quad y' \quad z' \quad h]$$

where

$$[T'] = \begin{bmatrix} T'_{11} & T'_{12} & T'_{13} & T'_{14} \\ T'_{21} & T'_{22} & T'_{23} & T'_{24} \\ T'_{31} & T'_{32} & T'_{33} & T'_{34} \\ T'_{41} & T'_{42} & T'_{43} & T'_{44} \end{bmatrix}$$

<sup>†</sup>For this reason mechanical drawings are explicitly dimensioned.

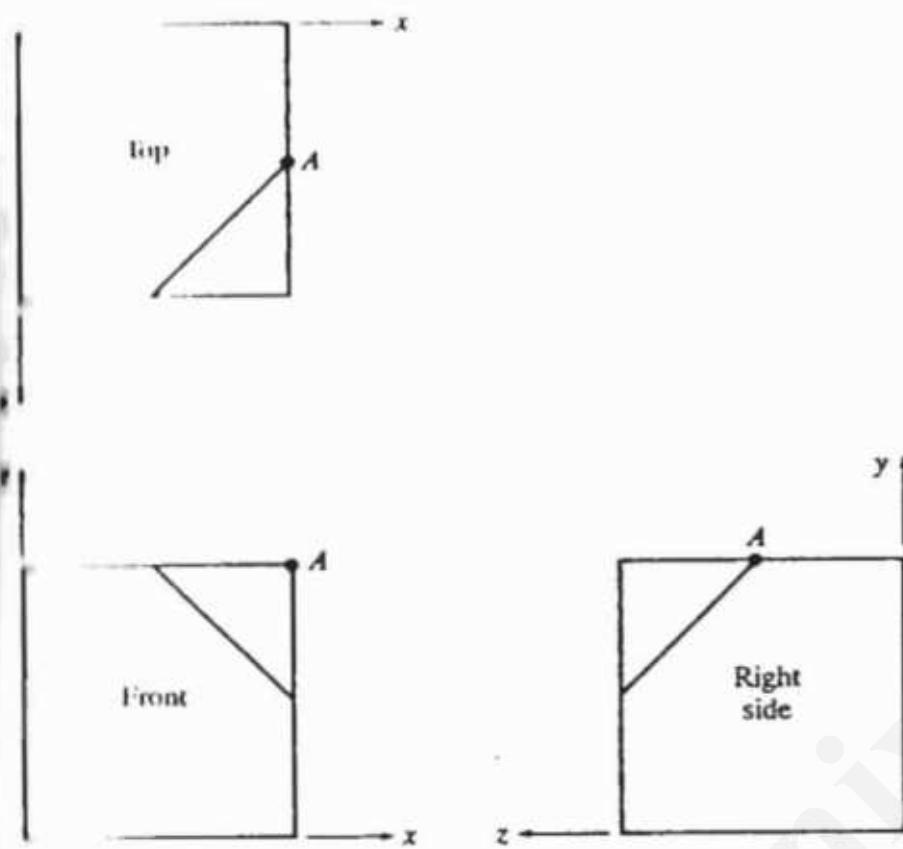


Figure 3-49 Three-dimensional reconstruction from orthographic projections.

The results can be projected onto a two-dimensional plane, say  $z = 0$ , using

$$[T''] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Concatenation of the two matrices yields

$$[T] = [T''][T'] = \begin{bmatrix} T_{11} & T_{12} & 0 & T_{14} \\ T_{21} & T_{22} & 0 & T_{24} \\ T_{31} & T_{32} & 0 & T_{34} \\ T_{41} & T_{42} & 0 & T_{44} \end{bmatrix}$$

It is useful to write the transformation as

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} T_{11} & T_{12} & 0 & T_{14} \\ T_{21} & T_{22} & 0 & T_{24} \\ T_{31} & T_{32} & 0 & T_{34} \\ T_{41} & T_{42} & 0 & T_{44} \end{bmatrix} = \begin{bmatrix} x' & y' & 0 & h \end{bmatrix} = h \begin{bmatrix} x^* & y^* & 0 & 1 \end{bmatrix} \quad (3-71)$$

Note that  $x^*$  and  $y^*$  are the coordinates of the perspective projection onto the  $z = 0$  plane. Projections onto the  $x = 0$  or  $y = 0$  planes could also be used.

Writing out Eq. (3-71) yields

$$T_{11}x + T_{21}y + T_{31}z + T_{41} = hx^* \quad (3-72a)$$

$$T_{12}x + T_{22}y + T_{32}z + T_{42} = hy^* \quad (3-72b)$$

$$T_{14}x + T_{24}y + T_{34}z + T_{44} = h \quad (3-72c)$$

Using  $h$  from Eq. (3-72c) and substituting into Eqs. (3-72a) and (3-72b) yields

$$(T_{11} - T_{14}x^*)x + (T_{21} - T_{24}x^*)y + (T_{31} - T_{34}x^*)z + (T_{41} - T_{44}x^*) = 0 \quad (3-73a)$$

$$(T_{12} - T_{14}y^*)x + (T_{22} - T_{24}y^*)y + (T_{32} - T_{34}y^*)z + (T_{42} - T_{44}y^*) = 0 \quad (3-73b)$$

As suggested by Sutherland (Ref. 3-4), this pair of equations can be considered in three different ways. First assume  $T$  and  $x, y, z$  are known. Then there are two equations in the two unknowns  $x^*$  and  $y^*$ . Thus, they may be used to solve directly for the coordinates of the perspective projection. This is the approach taken in all the previous discussions in this chapter.

Alternately  $T, x^*, y^*$  can be assumed known. In this case two equations in the three unknown space coordinates  $x, y, z$  result. The system of equations cannot be solved. However, if two perspective projections, say two photographs, are available, then Eq. (3-73) can be written for both projections. This yields

$$(T_{11}^1 - T_{14}^1x^{*1})x + (T_{21}^1 - T_{24}^1x^{*1})y + (T_{31}^1 - T_{34}^1x^{*1})z + (T_{41}^1 - T_{44}^1x^{*1}) = 0$$

$$(T_{12}^1 - T_{14}^1y^{*1})x + (T_{22}^1 - T_{24}^1y^{*1})y + (T_{32}^1 - T_{34}^1y^{*1})z + (T_{42}^1 - T_{44}^1y^{*1}) = 0$$

$$(T_{11}^2 - T_{14}^2x^{*2})x + (T_{21}^2 - T_{24}^2x^{*2})y + (T_{31}^2 - T_{34}^2x^{*2})z + (T_{41}^2 - T_{44}^2x^{*2}) = 0$$

$$(T_{12}^2 - T_{14}^2y^{*2})x + (T_{22}^2 - T_{24}^2y^{*2})y + (T_{32}^2 - T_{34}^2y^{*2})z + (T_{42}^2 - T_{44}^2y^{*2}) = 0$$

where the superscripts 1 and 2 indicate the first and second perspective projections. Note that the transformations  $[T^1]$  and  $[T^2]$  need not be the same. These equations can be rewritten in matrix form as

$$[A][X] = [B] \quad (3-74)$$

where

$$[A] = \begin{bmatrix} T_{11}^1 - T_{14}^1x^{*1} & T_{21}^1 - T_{24}^1x^{*1} & T_{31}^1 - T_{34}^1x^{*1} \\ T_{12}^1 - T_{14}^1y^{*1} & T_{22}^1 - T_{24}^1y^{*1} & T_{32}^1 - T_{34}^1y^{*1} \\ T_{11}^2 - T_{14}^2x^{*2} & T_{21}^2 - T_{24}^2x^{*2} & T_{31}^2 - T_{34}^2x^{*2} \\ T_{12}^2 - T_{14}^2y^{*2} & T_{22}^2 - T_{24}^2y^{*2} & T_{32}^2 - T_{34}^2y^{*2} \end{bmatrix}$$

$$[X]^T = [x \ y \ z]$$

$$[B]^T = [T_{44}^1x^{*1} - T_{41}^1 \quad T_{44}^1y^{*1} - T_{42}^1 \quad T_{44}^2x^{*2} - T_{41}^2 \quad T_{44}^2y^{*2} - T_{42}^2]$$

Equation (3-74) represents four equations in the three unknown space coordinates  $x$ ,  $y$ ,  $z$ .  $[A]$  is not a square matrix and consequently cannot be inverted to obtain the solution for  $[X]$ . Again, as in the case of reconstructing three-dimensional coordinates from orthographic projections, the problem is overspecified and thus can be solved only in some mean or best-fit sense.

A mean solution is computed by recalling that a matrix times its transpose is always square. Thus, multiplying both sides of Eq. (3-74) by  $[A]^T$  yields

$$[A]^T [A] [X] = [A]^T [B]$$

Taking the inverse of  $[ [A]^T [A] ]$  yields a mean solution for  $[X]$ , i.e.,

$$[X] = [ [A]^T [A] ]^{-1} [A]^T [B] \quad (3-75)$$

If no solution for  $[X]$  results, then the imposed conditions are redundant and no unique solution which yields a least error condition exists. An example illustrates this technique.

### Example 3-30 Three-Dimensional Reconstruction

Assume that the measured position of a point in one perspective projection is  $[0.836 \quad -1.836 \quad 0 \quad 1]$  and is  $[0.6548 \quad 0 \quad 0.2886 \quad 1]$  in a second perspective projection. The first perspective projection transformation is known to be the result of a  $60^\circ$  rotation about the  $y$ -axis, followed by a translation of 2 units in the negative  $y$  direction. The point of projection is at  $z = -1$ , and the result is projected onto the  $z = 0$  plane. This is effectively a two-point perspective projection. The second perspective projection is the result of a  $30^\circ$  rotation about each of the  $x$ - and  $y$ -axes. The point of projection is at  $y = -1$  and the result is projected onto the  $y = 0$  plane, i.e., effectively a three-point perspective projection.  $[T^1]$  and  $[T^2]$  are thus

$$[T^1] = \begin{bmatrix} 0.5 & 0 & 0 & -0.87 \\ 0 & 1 & 0 & 0 \\ 0.87 & 0 & 0 & 0.5 \\ 0 & -2 & 0 & 1 \end{bmatrix} \text{ and } [T^2] = \begin{bmatrix} 0.87 & 0 & -0.5 & 0 \\ 0.25 & 0 & 0.43 & 0.87 \\ 0.43 & 0 & 0.75 & -0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

First noting that the last two rows of  $[A]$  and  $[B]$  must be rewritten to account for  $[T^2]$  being a projection onto the  $y = 0$  plane, the  $A$ -matrix is

$$[A] = \begin{bmatrix} 1.22 & 0 & 0.45 \\ -1.59 & 1 & 0.92 \\ 0.87 & -0.32 & 0.76 \\ -0.5 & 0.18 & 0.89 \end{bmatrix}$$

and  $[B]^T = [0.84 \quad 0.16 \quad 0.65 \quad 0.29]$

Solution yields  $[X] = [0.5 \quad 0.5 \quad 0.5]$ , i.e., the center of the unit cube.

As a third way of considering Eq. (3-73) note that if the location of several points which appear in the perspective projection are known in object space **and** in the perspective projection, then it is possible to determine the transformation elements, i.e., the  $T_{ij}$ 's. These transformation elements can subsequently be used to determine the location of unknown points using the second technique described above. To see this, rewrite Eq. (3-73) as

$$T_{11}x + T_{21}y + T_{31}z + T_{41} - T_{14}xx^* - T_{24}yx^* - T_{34}zx^* - T_{44}x^* = 0 \quad (3-76a)$$

$$T_{12}x + T_{22}y + T_{32}z + T_{42} - T_{14}xy^* - T_{24}yy^* - T_{34}zy^* - T_{44}y^* = 0 \quad (3-76b)$$

Assuming that  $x^*$  and  $y^*$  as well as  $x, y, z$  are known, Eqs. (3-76a) and (3-76b) represent two equations in the 12 unknown transformation elements  $T_{ij}$ . Applying these equations to 6 noncoplanar known locations in object space and in the perspective projection yields a system of 12 equations in 12 unknowns. These equations can be solved exactly for the  $T_{ij}$ 's. Thus, the transformation that produced the perspective projection, for example, a photograph, is determined. Notice that in this case no prior knowledge of the transformation is required. If, for example, the perspective projections are photographs, neither the location nor the orientation of the camera is required. In matrix form the system of 12 equations is written as

$$\left[ \begin{array}{ccccccccccccc} x_1 & 0 & -x_1x_1^* & y_1 & 0 & -y_1x_1^* & z_1 & 0 & -z_1x_1^* & 1 & 0 & -x_1^* \\ 0 & x_1 & -x_1y_1^* & 0 & y_1 & -y_1y_1^* & 0 & z_1 & -z_1y_1^* & 0 & 1 & -y_1^* \\ x_2 & 0 & -x_2x_2^* & y_2 & 0 & -y_2x_2^* & z_2 & 0 & -z_2x_2^* & 1 & 0 & -x_2^* \\ 0 & x_2 & -x_2y_2^* & 0 & y_2 & -y_2y_2^* & 0 & z_2 & -z_2y_2^* & 0 & 1 & -y_2^* \\ x_3 & 0 & -x_3x_3^* & y_3 & 0 & -y_3x_3^* & z_3 & 0 & -z_3x_3^* & 1 & 0 & -x_3^* \\ 0 & x_3 & -x_3y_3^* & 0 & y_3 & -y_3y_3^* & 0 & z_3 & -z_3y_3^* & 0 & 1 & -y_3^* \\ x_4 & 0 & -x_4x_4^* & y_4 & 0 & -y_4x_4^* & z_4 & 0 & -z_4x_4^* & 1 & 0 & -x_4^* \\ 0 & x_4 & -x_4y_4^* & 0 & y_4 & -y_4y_4^* & 0 & z_4 & -z_4y_4^* & 0 & 1 & -y_4^* \\ x_5 & 0 & -x_5x_5^* & y_5 & 0 & -y_5x_5^* & z_5 & 0 & -z_5x_5^* & 1 & 0 & -x_5^* \\ 0 & x_5 & -x_5y_5^* & 0 & y_5 & -y_5y_5^* & 0 & z_5 & -z_5y_5^* & 0 & 1 & -y_5^* \\ x_6 & 0 & -x_6x_6^* & y_6 & 0 & -y_6x_6^* & z_6 & 0 & -z_6x_6^* & 1 & 0 & -x_6^* \\ 0 & x_6 & -x_6y_6^* & 0 & y_6 & -y_6y_6^* & 0 & z_6 & -z_6y_6^* & 0 & 1 & -y_6^* \end{array} \right] \begin{bmatrix} T_{11} \\ T_{12} \\ T_{14} \\ T_{21} \\ T_{22} \\ T_{24} \\ T_{31} \\ T_{32} \\ T_{34} \\ T_{41} \\ T_{42} \\ T_{44} \end{bmatrix} = 0 \quad (3-77)$$

where the subscripts correspond to points with known locations. Equations (3-77) are written in more compact form as

$$[ A' ] [ T ] = 0$$

Since Eqs. (3-77) are homogeneous, they contain an arbitrary scale factor. Hence  $T_{44}$  may, for example, be defined as unity and the resulting transformation normalized. This reduces the requirement to 11 equations or 5 1/2 points. If the transformation is normalized, then the last column in  $[ A' ]$  is moved to the right-hand side and the nonhomogeneous matrix equation is solved. An example is given below.

**Example 3-31 Elements for Reconstruction**

As a specific example, consider the unit cube with the six known corner points in the physical plane given by

$$[P] = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

The corresponding points in the transformed view are marked by a dot as shown in Fig. 3-50. The corresponding transformed coordinates of these points are

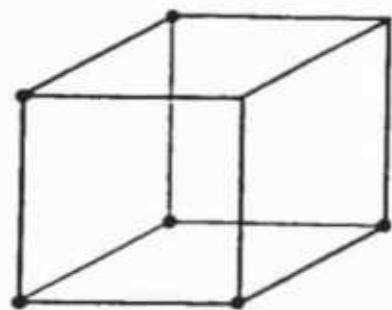
$$\begin{bmatrix} 0 & -1 \\ 0.34 & -0.8 \\ 0.34 & -0.4 \\ 0 & -0.5 \\ 0.44 & -1.75 \\ 0.83 & -1.22 \end{bmatrix}$$

Equation (3-77) then becomes

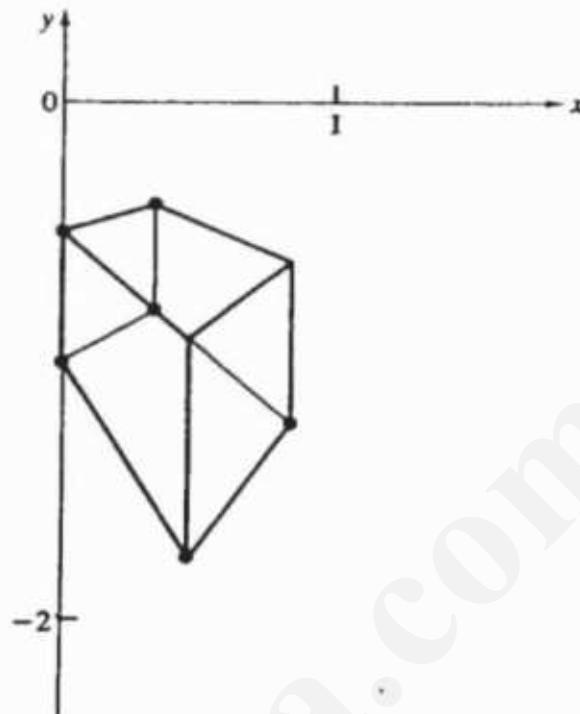
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -0.34 & 1 & 0 & -0.34 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0.8 & 0 & 1 & 0.8 \\ 0 & 0 & 0 & 1 & 0 & -0.34 & 1 & 0 & -0.34 & 1 & 0 & -0.34 \\ 0 & 0 & 0 & 0 & 1 & 0.4 & 0 & 1 & 0.4 & 0 & 1 & 0.4 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0.5 & 0 & 0 & 0 & 0 & 1 & 0.5 \\ 1 & 0 & -0.44 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -0.44 \\ 0 & 1 & 1.75 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1.75 \\ 1 & 0 & -0.83 & 0 & 0 & 0 & 1 & 0 & -0.83 & 1 & 0 & -0.83 \\ 0 & 1 & 1.22 & 0 & 0 & 0 & 0 & 1 & 1.22 & 0 & 1 & 1.22 \end{bmatrix} \begin{bmatrix} T_{11} \\ T_{12} \\ T_{14} \\ T_{21} \\ T_{22} \\ T_{24} \\ T_{31} \\ T_{32} \\ T_{34} \\ T_{41} \\ T_{42} \\ T_{44} \end{bmatrix} = 0$$

Solution for the 12 unknown  $T_{ij}$ 's yields

$$\begin{bmatrix} T_{11} \\ T_{12} \\ T_{14} \\ T_{21} \\ T_{22} \\ T_{24} \\ T_{31} \\ T_{32} \\ T_{34} \\ T_{41} \\ T_{42} \\ T_{44} \end{bmatrix} = \begin{bmatrix} 0.25 \\ 0 \\ -0.43 \\ 0 \\ 0.5 \\ 0 \\ 0.43 \\ 0 \\ 0.25 \\ 0 \\ -1 \\ 1 \end{bmatrix}$$



(a)



(b)

**Figure 3-50** Determining the transformation from a perspective projection.

Substituting these results into the  $4 \times 4$   $[T]$  matrix yields

$$[T] = \begin{bmatrix} 0.25 & 0 & 0 & -0.43 \\ 0 & 0.5 & 0 & 0 \\ 0.43 & 0 & 0 & 0.25 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

### 3-22 REFERENCES

- 3-1 Rogers, David F., *Procedural Elements for Computer Graphics*, McGraw-Hill Book Co., New York, 1985.
- 3-2 Carlbom, Ingrid, and Paciorek, Joseph, "Planar Geometric Projections and Viewing Transformations," *ACM Comp. Surv.*, Vol. 10, No.4, pp. 465-502, 1978.
- 3-3 Sutherland, I.E., "Three Dimensional Data Input by Tablet," *Proc. IEEE*, Vol. 62, No. 2, pp. 453-461, 1974.
- 3-4 Slama, Chester (ed.), *Manual of Photogrammetry*, American Society of Photogrammetry, 1980.
- 3-5 Kingslake, Rudolf (ed.), *Applied Optics and Optical Engineering*, Vol. 2, Academic Press, New York, 1965.

# Geometric Modeling

S. No.	Topic	Contents
5.	Representing curves (Hermite and Bezier)	Sections 11.2.1 - 11.2.2

---

## TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

### 11.2.1 Hermite Curves

The Hermite form (named for the mathematician) of the cubic polynomial curve segment is determined by constraints on the endpoints  $P_1$  and  $P_4$  and tangent vectors at the endpoints  $R_1$  and  $R_4$ . (The indices 1 and 4 are used, rather than 1 and 2, for consistency with later sections, where intermediate points  $P_2$  and  $P_3$  will be used instead of tangent vectors to define the curve.)

To find the *Hermite basis matrix*  $M_H$ , which relates the *Hermite geometry vector*  $G_H$  to the polynomial coefficients, we write four equations, one for each of the constraints, in the four unknown polynomial coefficients, and then solve for the unknowns.

Defining  $G_{H_x}$ , the  $x$  component of the Hermite geometry matrix, as

$$G_{H_x} = \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}_x \quad (11.12)$$

and rewriting  $x(t)$  from Eqs. (11.5) and (11.9) as

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x = T \cdot C_x = T \cdot M_H \cdot G_{H_x} = [t^3 \ t^2 \ t \ 1] M_H \cdot G_{H_x} \quad (11.13)$$

the constraints on  $x(0)$  and  $x(1)$  are found by direct substitution into Eq. (11.13) as

$$x(0) = P_{1_x} = [0 \ 0 \ 0 \ 1] M_H \cdot G_{H_x}, \quad (11.14)$$

$$x(1) = P_{4_x} = [1 \ 1 \ 1 \ 1] M_H \cdot G_{H_x}. \quad (11.15)$$

Just as in the general case we differentiated Eq. (11.7) to find Eq. (11.8), we now differentiate Eq. (11.13) to get  $x'(t) = [3t^2 \ 2t \ 1 \ 0] M_H \cdot G_{H_x}$ . Hence, the tangent vector-constraint equations can be written as

$$x'(0) = R_{1_x} = [0 \ 0 \ 1 \ 0] M_H \cdot G_{H_x}, \quad (11.16)$$

$$x'(1) = R_{4_x} = [3 \ 2 \ 1 \ 0] M_H \cdot G_{H_x}. \quad (11.17)$$

The four constraints of Eqs. (11.14), (11.15), (11.16), and (11.17) can be rewritten in matrix form as

$$\begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}_x = G_{H_x} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix} M_H \cdot G_{H_x} \quad (11.18)$$

For this equation (and the corresponding expressions for  $y$  and  $z$ ) to be satisfied,  $M_H$  must be the inverse of the  $4 \times 4$  matrix in Eq. (11.18)

$$M_H = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}. \quad (11.19)$$

$M_H$ , which is of course unique, can now be used in  $x(t) = T \cdot M_H \cdot G_{H_x}$  to find  $x(t)$  based on the geometry vector  $G_{H_x}$ . Similarly,  $y(t) = T \cdot M_H \cdot G_{H_y}$  and  $z(t) = T \cdot M_H \cdot G_{H_z}$ , so we can write

$$Q(t) = [x(t) \ y(t) \ z(t)] = T \cdot M_H \cdot G_H, \quad (11.20)$$

where  $G_H$  is the column vector

$$\begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}.$$

11.2

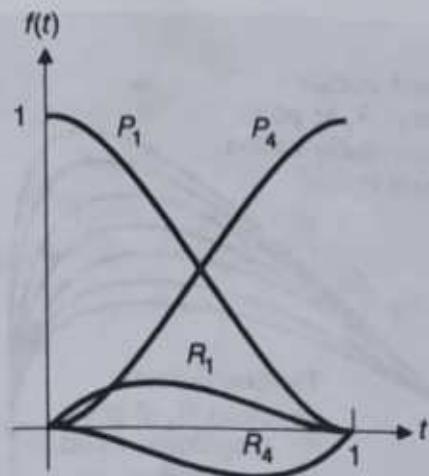


Fig. 11.12 The Hermite blending functions, labeled by the elements of the geometry vector that they weight.

Expanding the product  $T \cdot M_H$  in  $Q(t) = T \cdot M_H \cdot G_H$  gives the *Hermite blending functions*  $B_H$  as the polynomials weighting each element of the geometry vector:

$$\begin{aligned} Q(t) &= T \cdot M_H \cdot G_H = B_H \cdot G_H \\ &= (2t^3 - 3t^2 + 1)P_1 + (-2t^3 + 3t^2)P_4 + (t^3 - 2t^2 + t)R_1 + (t^3 - t^2)R_4. \end{aligned} \quad (11.21)$$

Figure 11.12 shows the four blending functions. Notice that, at  $t = 0$ , only the function labeled  $P_1$  is nonzero: only  $P_1$  affects the curve at  $t = 0$ . As soon as  $t$  becomes greater than zero,  $R_1$ ,  $P_4$ , and  $R_4$  begin to have an influence. Figure 11.13 shows the four functions weighted by the  $y$  components of a geometry vector, their sum  $y(t)$ , and the curve  $Q(t)$ .

Figure 11.14 shows a series of Hermite curves. The only difference among them is the length of the tangent vector  $R_1$ : the directions of the tangent vectors are fixed. The longer the vectors, the greater their effect on the curve. Figure 11.15 is another series of Hermite curves, with constant tangent-vector lengths but with different directions. In an interactive

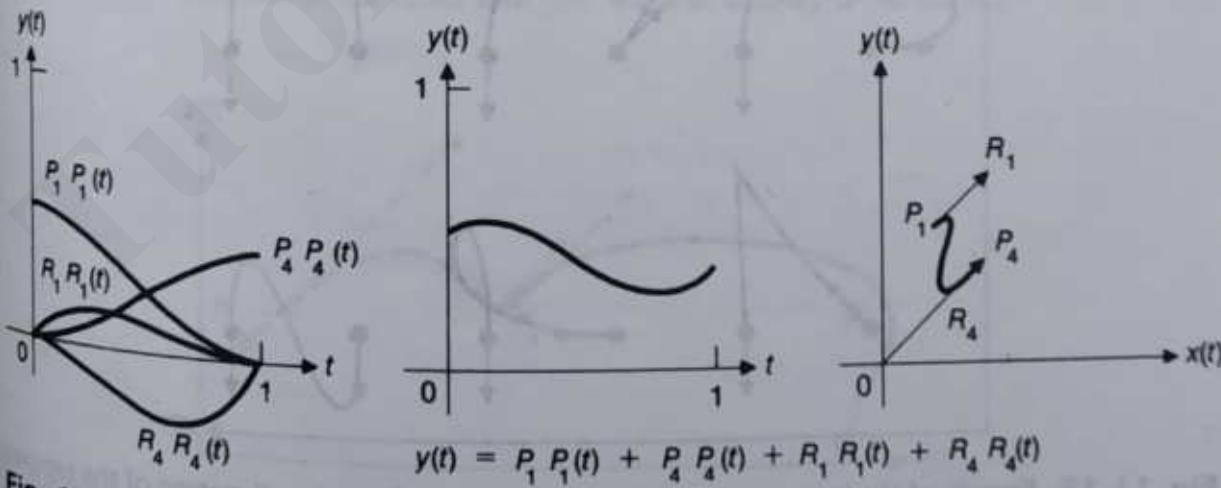
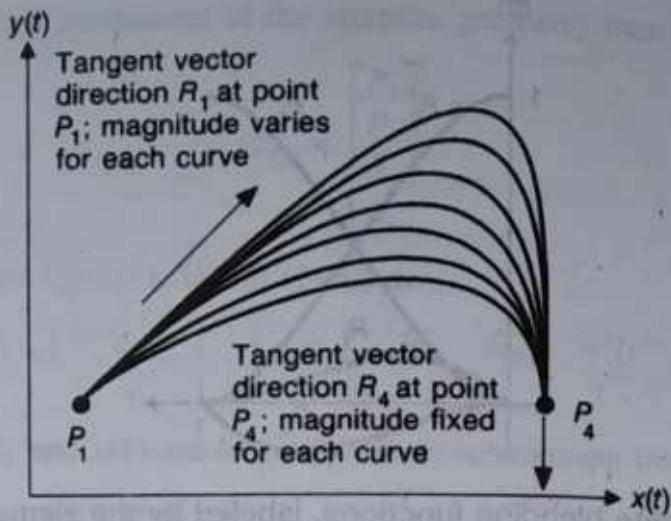


Fig. 11.13 A Hermite curve showing the four elements of the geometry vector weighted by the blending functions (leftmost four curves), their sum  $y(t)$ , and the 2D curve itself (far right).  $x(t)$  is defined by a similar weighted sum.

## 510 Representing Curves and Surfaces

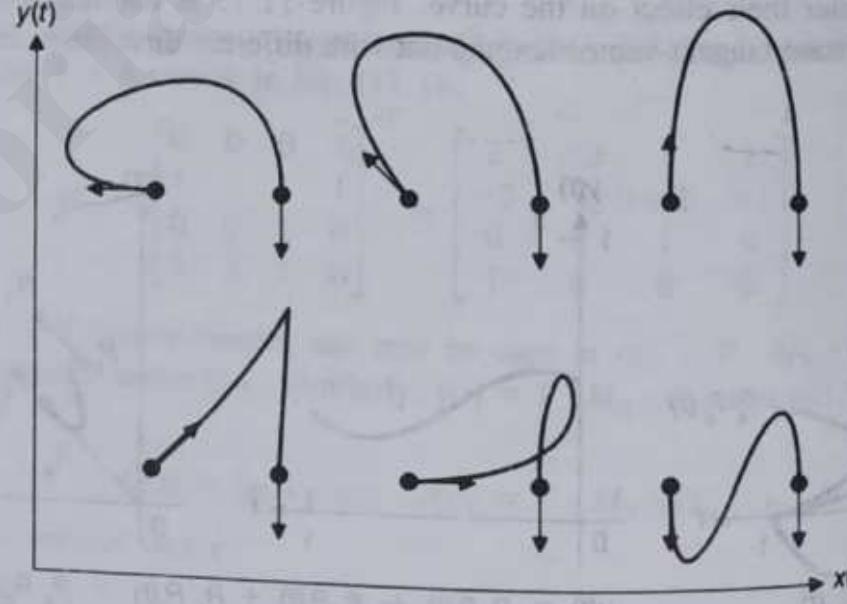


**Fig. 11.14** Family of Hermite parametric cubic curves. Only  $R_1$ , the tangent vector at  $P_1$ , varies for each curve, increasing in magnitude for the higher curves.

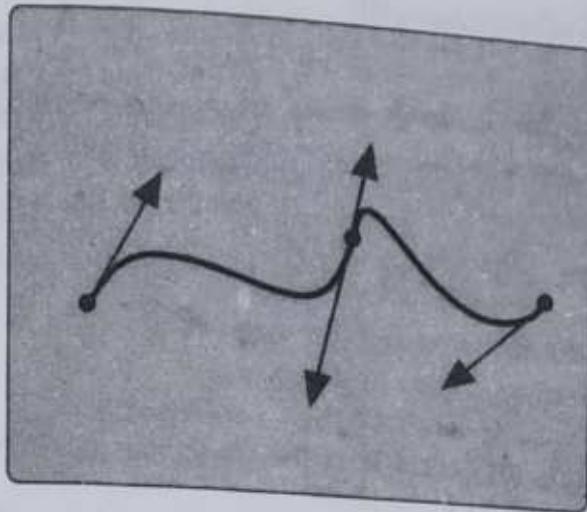
graphics system, the endpoints and tangent vectors of a curve are manipulated interactively by the user to shape the curve. Figure 11.16 shows one way of doing this.

For two Hermite cubics to share a common endpoint with  $G^1$  (geometrical) continuity, as in Fig. 11.17, the geometry vectors must have the form

$$\begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix} \text{ and } \begin{bmatrix} P_4 \\ P_7 \\ kR_4 \\ R_7 \end{bmatrix}, \text{ with } k > 0. \quad (11.22)$$



**Fig. 11.15** Family of Hermite parametric cubic curves. Only the direction of the tangent vector at the left starting point varies; all tangent vectors have the same magnitude. A smaller magnitude would eliminate the loop in the one curve.



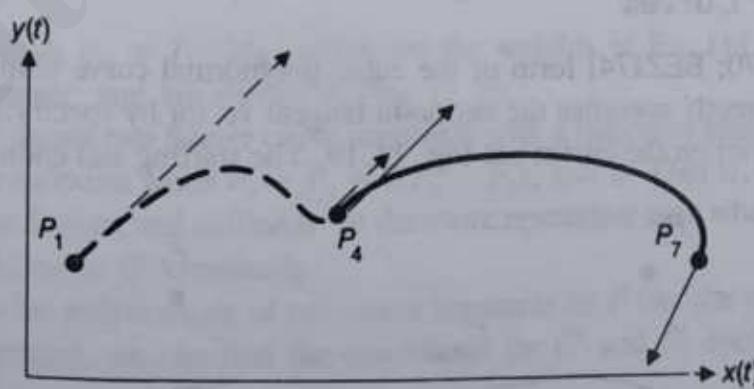
**Fig. 11.16** Two Hermite cubic curve segments displayed with controls to facilitate interactive manipulation. The endpoints can be repositioned by dragging the dots, and the tangent vectors can be changed by dragging the arrowheads. The tangent vectors at the join point are constrained to be collinear (to provide  $C^1$  continuity). The user is usually given a command to enforce  $C^0$ ,  $C^1$ ,  $G^1$ , or no continuity. The tangent vectors at the  $t = 1$  end of each curve are drawn in the reverse of the direction used in the mathematical formulation of the Hermite curve, for clarity and more convenient user interaction.

That is, there must be a shared endpoint ( $P_4$ ) and tangent vectors with at least equal directions. The more restrictive condition of  $C^1$  (parametric) continuity requires that  $k = 1$ , so the tangent vector direction and magnitude must be equal.

Hermite and other similar parametric cubic curves are simple to display: We evaluate Eq. (11.5) at  $n$  successive values of  $t$  separated by a step size  $\delta$ . Figure 11.18 gives the code. The evaluation within the **begin . . . end** takes 11 multiplies and 10 additions per 3D point. Use of Horner's rule for factoring polynomials,

$$f(t) = at^3 + bt^2 + ct + d = ((at + b)t + c)t + d, \quad (11.23)$$

reduces the effort slightly to nine multiplies and 10 additions per 3D point. In Section 11.2.9, we shall examine much more efficient ways to display these curves.



**Fig. 11.17** Two Hermite curves joined at  $P_4$ . The tangent vectors at  $P_4$  have the same direction but different magnitudes, yielding  $G^1$  but not  $C^1$  continuity.

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

```

typedef double CoefficientArray[4];
void DrawCurve (
    CoefficientArray cx,
    CoefficientArray cy,
    CoefficientArray cz,
    int n)
{
    int i;
    double δ = 1.0 / n;
    double t = 0;

    MoveAbs3 (cx[3], cy[3], cz[3]); /* t = 0: start at x(0), y(0), z(0) */
    for (i = 0; i < n; i++) {
        double t2, t3, x, y, z;

        t += δ;
        t2 = t * t;
        t3 = t2 * t;
        x = cx[0] * t3 + cx[1] * t2 + cx[2] * t + cx[3];
        y = cy[0] * t3 + cy[1] * t2 + cy[2] * t + cy[3];
        z = cz[0] * t3 + cz[1] * t2 + cz[2] * t + cz[3];
        DrawAbs3 (x, y, z);
    }
} /* DrawCurve */

```

Fig. 11.18 Program to display a cubic parametric curve.

Because the cubic curves are linear combinations (weighted sums) of the four elements of the geometry vector, as seen in Eq. (11.10), we can transform the curves by transforming the geometry vector and then using it to generate the transformed curve, which is equivalent to saying that the curves are invariant under rotation, scaling, and translation. This strategy is more efficient than is generating the curve as a series of short line segments and then transforming each individual line. The curves are *not* invariant under perspective projection, as will be discussed in Section 11.2.5.

### 11.2.2 Bézier Curves

The Bézier [BEZI70; BEZI74] form of the cubic polynomial curve segment, named after Pierre Bézier, indirectly specifies the endpoint tangent vector by specifying two intermediate points that are not on the curve; see Fig. 11.19. The starting and ending tangent vectors

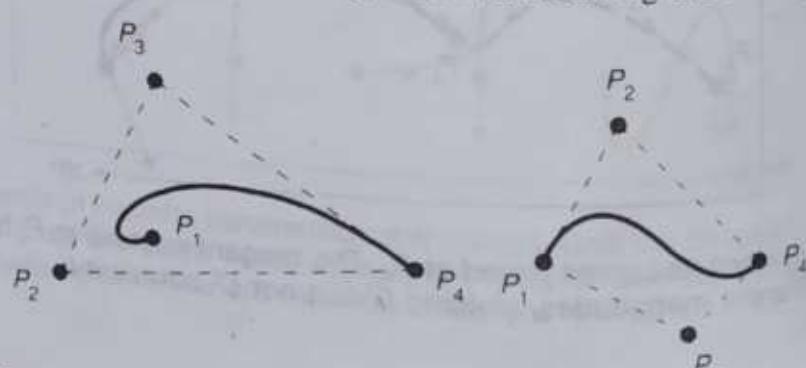


Fig. 11.19 Two Bézier curves and their control points. Notice that the convex hulls of the control points, shown as dashed lines, do not need to touch all four control points

11.2

are determined by the vectors  $P_1P_2$  and  $P_3P_4$  and are related to  $R_1$  and  $R_4$  by

$$R_1 = Q'(0) = 3(P_2 - P_1), R_4 = Q'(1) = 3(P_4 - P_3). \quad (11.24)$$

The Bézier curve interpolates the two end control points and approximates the other two. See Exercise 11.12 to understand why the constant 3 is used in Eq. (11.24).

The Bézier geometry vector  $G_B$ , consisting of four points, is

$$G_B = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}. \quad (11.25)$$

Then the matrix  $M_{HB}$  that defines the relation  $G_H = M_{HB} \cdot G_B$  between the Hermite geometry vector  $G_H$  and the Bézier geometry vector  $G_B$  is just the  $4 \times 4$  matrix in the following equation, which rewrites Eq. (11.24) in matrix form:

$$G_H = \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix} = M_{HB} \cdot G_B. \quad (11.26)$$

To find the Bézier basis matrix  $M_B$ , we use Eq. (11.20) for the Hermite form, substitute  $G_H = M_{HB} \cdot G_B$ , and define  $M_B = M_H \cdot M_{HB}$ :

$$Q(t) = T \cdot M_H \cdot G_H = T \cdot M_H \cdot (M_{HB} \cdot G_B) = T \cdot (M_H \cdot M_{HB}) \cdot G_B = T \cdot M_B \cdot G_B. \quad (11.27)$$

Carrying out the multiplication  $M_B = M_H \cdot M_{HB}$  gives

$$M_B = M_H \cdot M_{HB} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad (11.28)$$

and the product  $Q(t) = T \cdot M_B \cdot G_B$  is

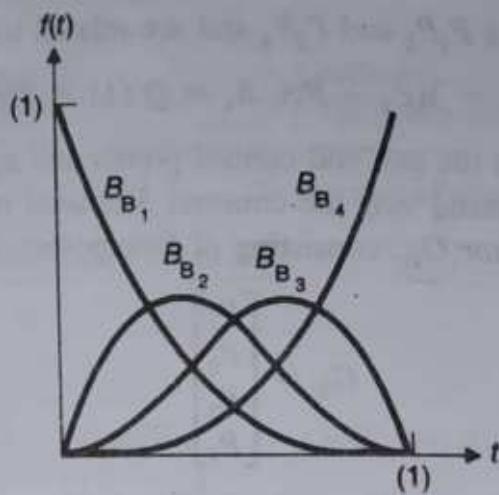
$$Q(t) = (1 - t)^3 P_1 + 3t(1 - t)^2 P_2 + 3t^2(1 - t) P_3 + t^3 P_4. \quad (11.29)$$

The four polynomials  $B_B = T \cdot M_B$ , which are the weights in Eq. (11.29), are called the *Bernstein polynomials*, and are shown in Fig. 11.20.

Figure 11.21 shows two Bézier curve segments with a common endpoint.  $G^1$  continuity is provided at the endpoint when  $P_3 - P_4 = k(P_4 - P_5)$ ,  $k > 0$ . That is, the three points  $P_3$ ,  $P_4$ , and  $P_5$  must be distinct and collinear. In the more restrictive case when  $k = 1$ , there is  $C^1$  continuity in addition to  $G^1$  continuity.

If we refer to the polynomials of two curve segments as  $x^l$  (for the left segment) and  $x^r$  (for the right segment), we can find the conditions for  $C^0$  and  $C^1$  continuity at their join point:

$$x^l(1) = x^r(0), \quad \frac{d}{dt}x^l(1) = \frac{d}{dt}x^r(0). \quad (11.30)$$



**Fig. 11.20** The Bernstein polynomials, which are the weighting functions for Bézier curves. At  $t = 0$ , only  $B_{B_1}$  is nonzero, so the curve interpolates  $P_1$ ; similarly, at  $t = 1$ , only  $B_{B_4}$  is nonzero, and the curve interpolates  $P_4$ .

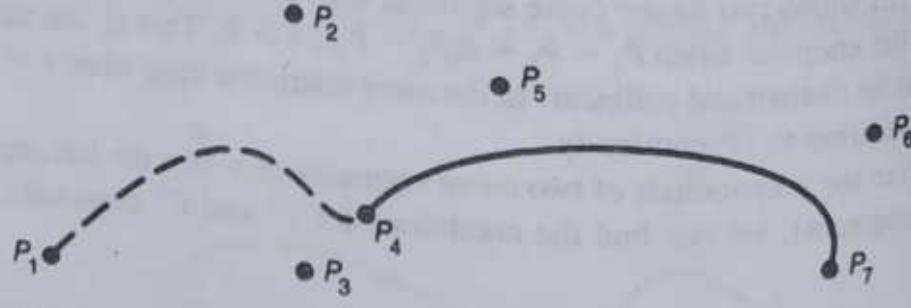
Working with the  $x$  component of Eq. (11.29), we have

$$x^l(1) = x^r(0) = P_{4_x}, \frac{d}{dt}x^l(1) = 3(P_{4_x} - P_{3_x}), \frac{d}{dt}x^r(0) = 3(P_{5_x} - P_{4_x}). \quad (11.31)$$

As always, the same conditions are true of  $y$  and  $z$ . Thus, we have  $C^0$  and  $C^1$  continuity when  $P_4 - P_3 = P_5 - P_4$ , as expected.

Examining the four  $B_B$  polynomials in Eq. (11.29), we note that their sum is everywhere unity and that each polynomial is everywhere nonnegative for  $0 \leq t < 1$ . Thus,  $Q(t)$  is just a weighted average of the four control points. This condition means that each curve segment, which is just the sum of four control points weighted by the polynomials, is completely contained in the *convex hull* of the four control points. The convex hull for 2D curves is the convex polygon formed by the four control points: Think of it as the polygon formed by putting a rubberband around the points (Fig. 11.19). For 3D curves, the convex hull is the convex polyhedron formed by the control points: Think of it as the polyhedron formed by stretching a rubber sheet around the four points.

This convex-hull property holds for all cubics defined by weighted sums of control points if the blending functions are nonnegative and sum to one. In general, the weighted average of  $n$  points falls within the convex hull of the  $n$  points; this can be seen intuitively



**Fig. 11.21** Two Bézier curves joined at  $P_4$ . Points  $P_3$ ,  $P_4$ , and  $P_5$  are collinear. Curves are the same as those used in Fig. 11.17.

for  $n = 2$  and  $n = 3$ , and the generalization follows. Another consequence of the fact that the four polynomials sum to unity is that the value of the fourth polynomial for any value of  $t$  can be found by subtracting the first three from unity.

The convex-hull property is also useful for clipping curve segments: Rather than clip each short line piece of a curve segment to determine its visibility, we first apply a polygonal clip algorithm to clip the convex hull or its extent against the clip region. If the convex hull (extent) is completely within the clip region, so is the entire curve segment. If the convex hull (extent) is completely outside the clip region, so is the curve segment. Only if the convex hull (extent) intersects the clip region does the curve segment itself need to be examined.

**EXERCISES**

- 11.1** Develop the equations, similar to Eq. (11.2), for the coefficients  $A$  and  $B$  of the plane equation. Assume that the polygon vertices are enumerated counterclockwise as viewed toward the plane from the positive side of the plane. The surface normal—given by  $A$ ,  $B$ , and  $C$ —points toward the positive side of the plane (which accounts for the need to negate the area computed for  $B$ , as discussed in Section 11.1.3).
- 11.2** Write a program to calculate the plane equation coefficients, given  $n$  vertices of a polygon that is approximately planar. The vertices are enumerated in a counterclockwise direction, as defined in Exercise 11.1. Test the program with  $n = 3$  for several known planes; then test it for larger  $n$ .
- 11.3** Find the geometry matrix and basis matrix for the parametric representation of a straight line given in Eq. (11.11).
- 11.4** Implement the procedure DrawCurve given in Fig 11.18. Display a number of curves, varying the coefficients  $cx$ ,  $cy$ , and  $cz$ . Try to make the curve correspond to some of the curve segments shown in figures in this chapter. Why is this difficult to do?
- 11.5** Show that, for a 2D curve  $[x(t) \ y(t)]$ ,  $G^1$  continuity means that the geometric slope  $dy/dx$  is equal at the join points between segments.
- 11.6** Let  $\gamma(t) = (t, t^2)$  for  $0 \leq t \leq 1$ , and let  $\eta(t) = (2t + 1, t^3 + 4t + 1)$  for  $0 \leq t \leq 1$ . Notice that  $\gamma(1) = (1, 1) = \eta(0)$ , so  $\gamma$  and  $\eta$  join with  $C^0$  continuity.
- Plot  $\eta(t)$  and  $\gamma(t)$  for  $0 \leq t \leq 1$ .
  - Do  $\eta(t)$  and  $\gamma(t)$  meet with  $C^1$  continuity at the join point? (You will need to compute the vectors  $\frac{d\gamma}{dt}(1)$  and  $\frac{d\eta}{dt}(0)$  to check this.)
  - Do  $\eta(t)$  and  $\gamma(t)$  meet with  $G^1$  continuity at the join point? (You will need to check ratios from part (b) to determine this).
- 11.7** Consider the paths
- $$\gamma(t) = (t^2 - 2t + 1, t^3 - 2t^2 + t) \quad \text{and} \quad \eta(t) = (t^2 + 1, t^3),$$
- both defined on the interval  $0 \leq t \leq 1$ . The curves join, since  $\gamma(1) = (1, 0) = \eta(0)$ . Show that they meet with  $C^1$  continuity, but not with  $G^1$  continuity. Plot both curves as functions of  $t$  to demonstrate exactly why this happens.
- 11.8** Show that the two curves  $\gamma(t) = (t^2 - 2t, t)$  and  $\eta(t) = (t^2 + 1, t + 1)$  are both  $C^1$  and  $G^1$  continuous where they join at  $\gamma(1) = \eta(0)$ .
- 11.9** Analyze the effect on a B-spline of having in sequence four collinear control points.
- 11.10** Write a program to accept an arbitrary geometry matrix, basis matrix, and list of control points, and to draw the corresponding curve.
- 11.11** Find the conditions under which two joined Hermite curves have  $C^1$  continuity.
- 11.12** Suppose the equations relating the Hermite geometry to the Bézier geometry were of the form  $R_1 = \beta(P_2 - P_1)$ ,  $R_4 = \beta(P_4 - P_3)$ . Consider the four equally spaced Bézier control points  $P_1 = (0, 0)$ ,  $P_2 = (1, 0)$ ,  $P_3 = (2, 0)$ ,  $P_4 = (3, 0)$ . Show that, for the parametric curve  $Q(t)$  to have constant velocity from  $P_1$  to  $P_4$ , the coefficient  $\beta$  must be equal to 3.
- 11.13** Write an interactive program that allows the user to create and refine piecewise continuous cubic curves. Represent the curves internally as B-splines. Allow the user to specify how the curve is to be interactively manipulated—as Hermite, Bézier, or B-splines.
- 11.14** Show that duplicate interior control points on a B-spline do not affect the  $C^2$  continuity at the

join point. Do this by writing out the equations for the two curve segments formed by the control points  $P_{i-1}, P_i, P_{i+1}, P_{i+2} = P_{i+1}, P_{i+3}$ . Evaluate the second derivative of the first segment at  $t = 1$ , and that of the second segment at  $t = 0$ . They should be the same.

11.15 Find the blending functions for the Catmull-Rom splines of Eq. (11.47). Do they sum to 1, and are they everyone nonzero? If not, the spline is not contained in the convex hull of the points.

11.16 Using Eqs. (11.49), (11.50), and (11.19), find the basis matrix  $M_{KB}$  for the Kochanek-Bartels spline, using the geometry matrix  $G_{B_{ij}}$  of Eq. (11.32).

11.17 Write an interactive program that allows the user to create, to manipulate interactively, and to refine piecewise continuous  $\beta$ -spline curves. Experiment with the effect of varying  $\beta_1$  and  $\beta_2$ .

11.18 Write an interactive program that allows the user to create, to manipulate interactively, and refine piecewise continuous Kochanek-Bartels curves. Experiment with the effect of varying  $a$ ,  $b$ , and  $c$ .

11.19 Implement both the forward-difference and recursive-subdivision curve-display procedures. Compare execution times for displaying various curves such that the curves are equally smooth to the eye.

11.20 Why is Eq. (11.36) for uniform B-splines written as  $Q_i(t - t_i)$ , whereas Eq. (11.43) for nonuniform B-splines is written as  $Q_i(t)$ ?

11.21 Given a 2D nonuniform B-spline and an  $(x, y)$  value on the curve, write a program to find the corresponding value of  $t$ . Be sure to consider the possibility that, for a given value of  $x$  (or  $y$ ), there may be multiple values of  $y$  (or  $x$ ).

11.22 Given a value  $t$  at which a Bézier curve is to be split, use the de Casteljau construction shown in Fig. 11.35 to find the division matrices  $D_B^L(t)$  and  $D_B^R(t)$ .

11.23 Apply the methodology used to derive Eq. (11.82) for Hermite surfaces to derive Eq. (11.86) for Bézier surfaces.

11.24 Write programs to display parametric cubic curves and surfaces using forward differences and recursive subdivision. Vary the step size  $\delta$  and error measure  $\epsilon$  to determine the effects of these parameters on the appearance of the curve.

11.25 Given four Hermite patches joined at a common corner and along the edges emanating from that corner, show the four geometry matrices and the relations that must hold between elements of the matrices.

11.26 Let  $t_0 = 0, t_1 = 1, t_2 = 3, t_3 = 4, t_4 = 5$ . Using these values, compute  $B_{0,4}$  and each of the functions used in its definition. Then plot these functions on the interval  $-3 \leq t \leq 8$ .

11.27 Expand the recurrence relation of Eq. (11.44) into an explicit expression for  $B_{i,4}(t)$ . Use Fig. 11.26 as a guide.

11.28 Write a program that displays a nonuniform, non-rational B-spline, given as input a knot sequence and control points. Provide a user-controllable option to calculate the  $B_{i,4}(t)$  in two ways: (a) using the recurrence relations of Eq. (11.44), and (b) using the explicit expression found in Exercise 11.27. Measure how much time is taken by each method. Is the faster method necessarily the better one?

11.29 Expand the program from Exercise 11.28 to allow interactive input and modification of the B-splines.

11.30 Write procedures to subdivide a curve recursively in two ways: adaptively, with a flatness test, and fixed, with a uniform level of subdivision. Draw a curve first with the adaptive subdivision, noting the deepest level of subdivision needed. Now draw the same curve with fixed subdivision as deep as needed for the adaptive case. Compare the execution time of the two procedures for a variety of curves.

## Visible Surface determination

S. No.	Topic	Contents
6.	Z – buffer algorithm, Depth Sort algorithm and Warnock's algorithm	Sections 15.4 - 15.5.1 Sections 15.7.1

---

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

## 15.4 THE z-BUFFER ALGORITHM

The *z-buffer* or *depth-buffer* image-precision algorithm, developed by Catmull [CATM74b], is one of the simplest visible-surface algorithms to implement in either software or hardware. It requires that we have available not only a frame buffer  $F$  in which color values are stored, but also a *z-buffer*  $Z$ , with the same number of entries, in which a  $z$ -value is stored for each pixel. The *z-buffer* is initialized to zero, representing the  $z$ -value at the back clipping plane, and the frame buffer is initialized to the background color. The largest value that can be stored in the *z-buffer* represents the  $z$  of the front clipping plane. Polygons are scan-converted into the frame buffer in arbitrary order. During the scan-conversion process, if the polygon point being scan-converted at  $(x, y)$  is no farther from the viewer than is the point whose color and depth are currently in the buffers, then the new point's color and depth replace the old values. The pseudocode for the *z-buffer* algorithm is shown in Fig. 15.21. The WritePixel and ReadPixel procedures introduced in Chapter 3 are supplemented here by WriteZ and ReadZ procedures that write and read the *z-buffer*.

```

void zBuffer(void)
{
    int x, y;

    for (y = 0; y < YMAX; y++) { /* Clear frame buffer and z-buffer */
        for (x = 0; x < XMAX; x++) {
            WritePixel (x, y, BACKGROUND_VALUE);
            WriteZ (x, y, 0);
        }
    }

    for (each polygon) { /* Draw polygons */
        for (each pixel in polygon's projection) {
            double pz = polygon's z-value at pixel coords (x, y);
            if (pz >= ReadZ (x, y)) { /* New point is not farther */
                WriteZ (x, y, pz);
                WritePixel (x, y, polygon's color at pixel coords (x, y));
            }
        }
    }
} /* zBuffer */

```

**Fig. 15.21** Pseudocode for the z-buffer algorithm.

No presorting is necessary and no object-object comparisons are required. The entire process is no more than a search over each set of pairs  $\{Z_i(x, y), F_i(x, y)\}$  for fixed  $x$  and  $y$ , to find the largest  $Z_i$ . The  $z$ -buffer and the frame buffer record the information associated with the largest  $z$  encountered thus far for each  $(x, y)$ . Thus, polygons appear on the screen in the order in which they are processed. Each polygon may be scan-converted one scan line at a time into the buffers, as described in Section 3.6. Figure 15.22 shows the addition of two polygons to an image. Each pixel's shade is shown by its color; its  $z$  is shown as a number.

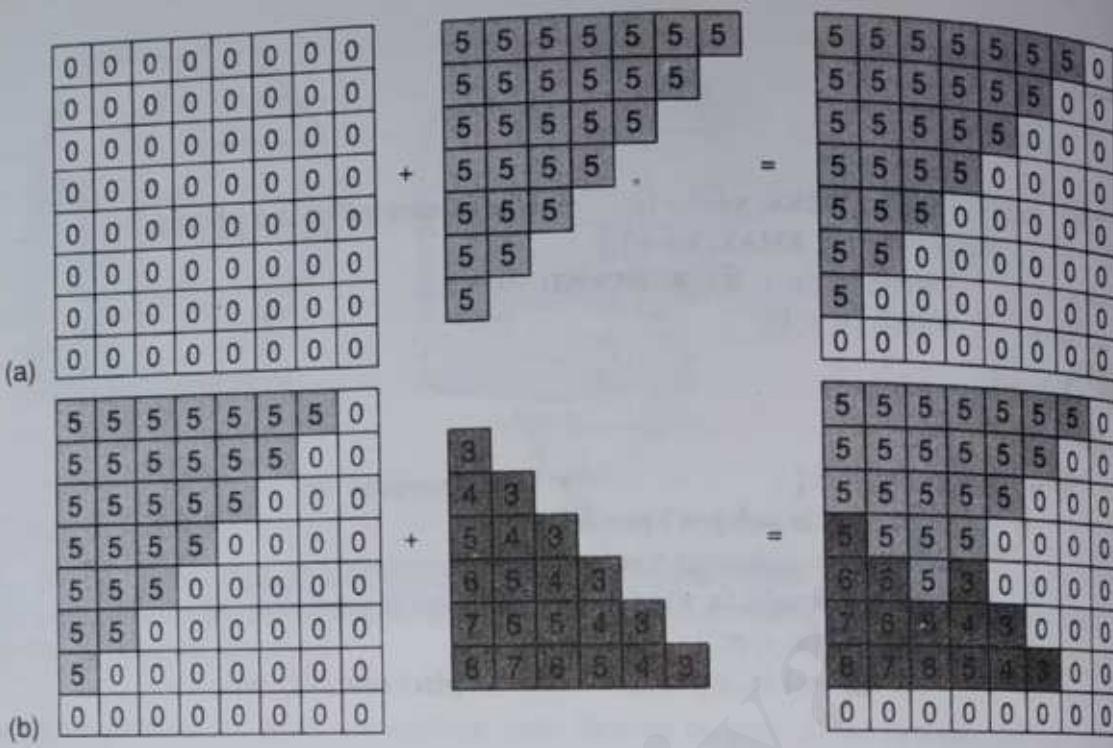
Remembering our discussion of depth coherence, we can simplify the calculation of  $z$  for each point on a scan line by exploiting the fact that a polygon is planar. Normally, to calculate  $z$ , we would solve the plane equation  $Ax + By + Cz + D = 0$  for the variable  $z$ :

$$z = \frac{-D - Ax - By}{C} \quad (15.6)$$

Now, if at  $(x, y)$  Eq. (15.6) evaluates to  $z_1$ , then at  $(x + \Delta x, y)$  the value of  $z$  is

$$z_1 - \frac{A}{C}(\Delta x). \quad (15.7)$$

Only one subtraction is needed to calculate  $z(x + 1, y)$  given  $z(x, y)$ , since the quotient  $A/C$  is constant and  $\Delta x = 1$ . A similar incremental calculation can be performed to determine the first value of  $z$  on the next scan line, decrementing by  $B/C$  for each  $\Delta y$ . Alternatively, if



**Fig. 15.22** The z-buffer. A pixel's shade is shown by its color, its z value is shown as a number. (a) Adding a polygon of constant z to the empty z-buffer. (b) Adding another polygon that intersects the first.

the surface has not been determined or if the polygon is not planar (see Section 11.1.3),  $z(x, y)$  can be determined by interpolating the  $z$  coordinates of the polygon's vertices along pairs of edges, and then across each scan line, as shown in Fig. 15.23. Incremental calculations can be used here as well. Note that the color at a pixel does not need to be computed if the conditional determining the pixel's visibility is not satisfied. Therefore, if the shading computation is time consuming, additional efficiency can be gained by performing a rough front-to-back depth sort of the objects to display the closest objects first.

The z-buffer algorithm does not require that objects be polygons. Indeed, one of its most powerful attractions is that it can be used to render any object if a shade and a z-value

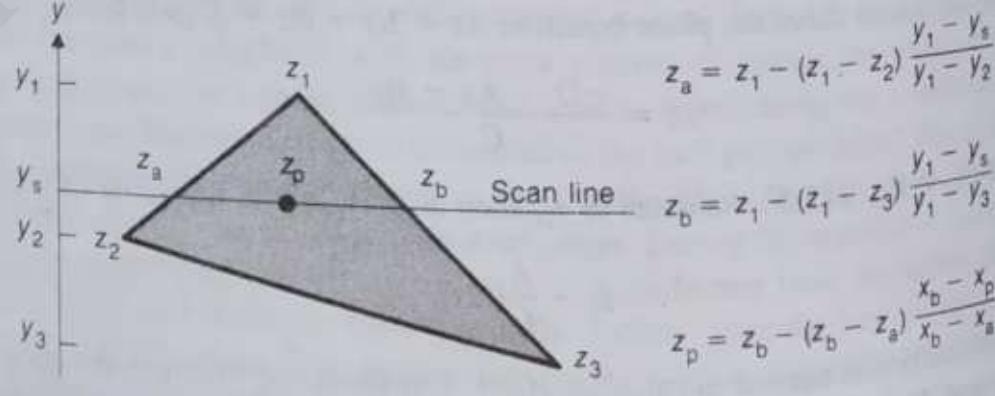


Fig. 15.23 Interpolation of  $z$  values along polygon edges and scan lines.  $z_1$  is interpolated between  $z_1$  and  $z_2$ ;  $z_b$  between  $z_1$  and  $z_3$ ;  $z_p$  between  $z_3$  and  $z_b$ .

can be determined for each point in its projection; no explicit intersection algorithms need to be written.

The z-buffer algorithm performs radix sorts in  $x$  and  $y$ , requiring no comparisons, and its  $z$  sort takes only one comparison per pixel for each polygon containing that pixel. The time taken by the visible-surface calculations tends to be independent of the number of polygons in the objects because, on the average, the number of pixels covered by each polygon decreases as the number of polygons in the view volume increases. Therefore, the average size of each set of pairs being searched tends to remain fixed. Of course, it is also necessary to take into account the scan-conversion overhead imposed by the additional polygons.

Although the z-buffer algorithm requires a large amount of space for the z-buffer, it is easy to implement. If memory is at a premium, the image can be scan-converted in strips, so that only enough z-buffer for the strip being processed is required, at the expense of performing multiple passes through the objects. Because of the z-buffer's simplicity and the lack of additional data structures, decreasing memory costs have inspired a number of hardware and firmware implementations of the z-buffer, examples of which are discussed in Chapter 18. Because the z-buffer algorithm operates in image precision, however, it is subject to aliasing. The A-buffer algorithm [CARP84], described in Section 15.7, addresses this problem by using a discrete approximation to unweighted area sampling.

The z-buffer is often implemented with 16- through 32-bit integer values in hardware, but software (and some hardware) implementations may use floating-point values. Although a 16-bit z-buffer offers an adequate range for many CAD/CAM applications, 16 bits do not have enough precision to represent environments in which objects defined with millimeter detail are positioned a kilometer apart. To make matters worse, if a perspective projection is used, the compression of distant  $z$  values resulting from the perspective divide has a serious effect on the depth ordering and intersections of distant objects. Two points that would transform to different integer  $z$  values if close to the view plane may transform to the same  $z$  value if they are farther back (see Exercise 15.13 and [HUGH89]).

The z-buffer's finite precision is responsible for another aliasing problem. Scan-conversion algorithms typically render two different sets of pixels when drawing the common part of two collinear edges that start at different endpoints. Some of those pixels shared by the rendered edges may also be assigned slightly different  $z$  values because of numerical inaccuracies in performing the  $z$  interpolation. This effect is most noticeable at the shared edges of a polyhedron's faces. Some of the visible pixels along an edge may be part of one polygon, while the rest come from the polygon's neighbor. The problem can be fixed by inserting extra vertices to ensure that vertices occur at the same points along the common part of two collinear edges.

Even after the image has been rendered, the z-buffer can still be used to advantage. Since it is the only data structure used by the visible-surface algorithm proper, it can be saved along with the image and used later to merge in other objects whose  $z$  can be computed. The algorithm can also be coded so as to leave the z-buffer contents unmodified when rendering selected objects. If the z-buffer is masked off this way, then a single object can be written into a separate set of overlay planes with hidden surfaces properly removed (if the object is a single-valued function of  $x$  and  $y$ ) and then erased without affecting the contents of the z-buffer. Thus, a simple object, such as a ruled grid, can be moved about the

image in  $x$ ,  $y$ , and  $z$ , to serve as a "3D cursor" that obscures and is obscured by the objects in the environment. Cutaway views can be created by making the  $z$ -buffer and frame-buffer writes contingent on whether the  $z$  value is behind a cutting plane. If the objects being displayed have a single  $z$  value for each  $(x, y)$ , then the  $z$ -buffer contents can also be used to compute area and volume. Exercise 15.25 explains how to use the  $z$ -buffer for picking.

Rossignac and Requicha [ROSS86] discuss how to adapt the  $z$ -buffer algorithm to handle objects defined by CSG. Each pixel in a surface's projection is written only if it is both closer in  $z$  and on a CSG object constructed from the surface. Instead of storing only the point with closest  $z$  at each pixel, Atherton suggests saving a list of all points, ordered by  $z$  and accompanied by each surface's identity, to form an *object buffer* [ATHE81]. A postprocessing stage determines how the image is displayed. A variety of effects, such as transparency, clipping, and Boolean set operations, can be achieved by processing each pixel's list, without any need to re-scan convert the objects.

## 15.5 LIST-PRIORITY ALGORITHMS

*List-priority algorithms* determine a visibility ordering for objects ensuring that a correct picture results if the objects are rendered in that order. For example, if no object overlaps another in  $z$ , then we need only to sort the objects by increasing  $z$ , and to render them in that order. Farther objects are obscured by closer ones as pixels from the closer polygons overwrite those of the more distant ones. If objects overlap in  $z$ , we may still be able to determine a correct order, as in Fig. 15.24(a). If objects cyclically overlap each other, as Fig. 15.24(b) and (c), or penetrate each other, then there is no correct order. In these cases, it will be necessary to split one or more objects to make a linear order possible.

List-priority algorithms are hybrids that combine both object-precision and image-precision operations. Depth comparisons and object splitting are done with object precision. Only scan conversion, which relies on the ability of the graphics device to overwrite the pixels of previously drawn objects, is done with image precision. Because the list of sorted objects is created with object precision, however, it can be redisplayed correctly at any resolution. As we shall see, list-priority algorithms differ in how they determine the sorted order, as well as in which objects get split, and when the splitting occurs. The sort need not be on  $z$ , some objects may be split that neither cyclically overlap nor penetrate others, and the splitting may even be done independent of the viewer's position.

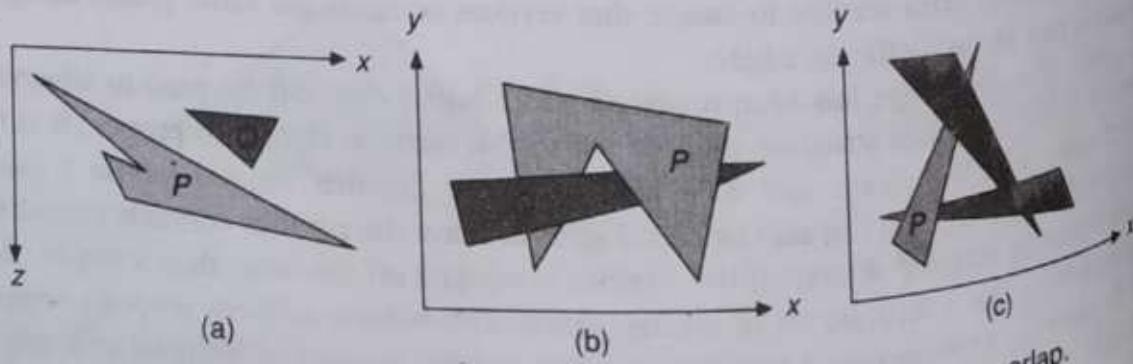


Fig. 15.24 Some cases in which  $z$  extents of polygons overlap.

### 15.5.1 The Depth-Sort Algorithm

The basic idea of the *depth-sort algorithm*, developed by Newell, Newell, and Sancha [NEW72], is to paint the polygons into the frame buffer in order of decreasing distance from the viewpoint. Three conceptual steps are performed:

1. Sort all polygons according to the smallest (farthest)  $z$  coordinate of each
2. Resolve any ambiguities this may cause when the polygons'  $z$  extents overlap, splitting polygons if necessary
3. Scan convert each polygon in ascending order of smallest  $z$  coordinate (i.e., back to front).

Consider the use of explicit priority, such as that associated with views in SPHIGS. The explicit priority takes the place of the minimum  $z$  value, and there can be no depth ambiguities because each priority is thought of as corresponding to a different plane of constant  $z$ . This simplified version of the depth-sort algorithm is often known as the *painter's algorithm*, in reference to how a painter might paint closer objects over more distant ones. Environments whose objects each exist in a plane of constant  $z$ , such as those of VLSI layout, cartography, and window management, are said to be  $2\frac{1}{2}$ D and can be correctly handled with the painter's algorithm. The painter's algorithm may be applied to a scene in which each polygon is not embedded in a plane of constant  $z$  by sorting the polygons by their minimum  $z$  coordinate or by the  $z$  coordinate of their centroid, ignoring step 2. Although scenes can be constructed for which this approach works, it does not in general produce a correct ordering.

Figure 15.24 shows some of the types of ambiguities that must be resolved as part of step 2. How is this done? Let the polygon currently at the far end of the sorted list of polygons be called  $P$ . Before this polygon is scan-converted into the frame buffer, it must be tested against each polygon  $Q$  whose  $z$  extent overlaps the  $z$  extent of  $P$ , to prove that  $P$  cannot obscure  $Q$  and that  $P$  can therefore be written before  $Q$ . Up to five tests are performed, in order of increasing complexity. As soon as one succeeds,  $P$  has been shown not to obscure  $Q$  and the next polygon  $Q$  overlapping  $P$  in  $z$  is tested. If all such polygons pass, then  $P$  is scan-converted and the next polygon on the list becomes the new  $P$ . The five tests are

1. Do the polygons'  $x$  extents not overlap?
2. Do the polygons'  $y$  extents not overlap?
3. Is  $P$  entirely on the opposite side of  $Q$ 's plane from the viewpoint? (This is not the case in Fig. 15.24(a), but is true for Fig. 15.25.)
4. Is  $Q$  entirely on the same side of  $P$ 's plane as the viewpoint? (This is not the case in Fig. 15.24(a), but is true for Fig. 15.26.)
5. Do the projections of the polygons onto the  $(x, y)$  plane not overlap? (This can be determined by comparing the edges of one polygon to the edges of the other.)

Exercise 15.6 suggests a way to implement tests 3 and 4.

If all five tests fail, we assume for the moment that  $P$  actually obscures  $Q$ , and therefore test whether  $Q$  can be scan-converted before  $P$ . Tests 1, 2, and 5 do not need to be repeated,

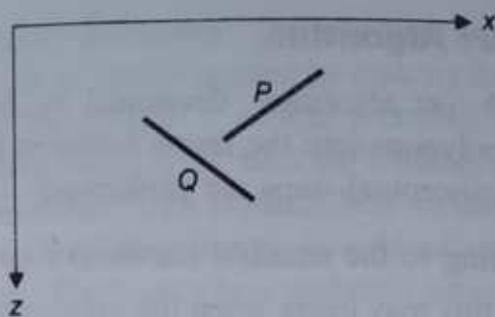


Fig. 15.25 Test 3 is true.

but new versions of tests 3 and 4 are used, with the polygons reversed:

3'. Is  $Q$  entirely on the opposite side of  $P$ 's plane from the viewpoint?

4'. Is  $P$  entirely on the same side of  $Q$ 's plane as the viewpoint?

In the case of Fig. 15.24(a), test 3' succeeds. Therefore, we move  $Q$  to the end of the list and it becomes the new  $P$ . In the case of Fig. 15.24(b), however, the tests are still inconclusive; in fact, there is no order in which  $P$  and  $Q$  can be scan-converted correctly. Instead, either  $P$  or  $Q$  must be split by the plane of the other (see Section 3.14 on polygon clipping, treating the clip edge as a clip plane). The original unsplit polygon is discarded, its pieces are inserted in the list in proper  $z$  order, and the algorithm proceeds as before.

Figure 15.24(c) shows a more subtle case. It is possible for  $P$ ,  $Q$ , and  $R$  to be oriented such that each polygon can always be moved to the end of the list to place it in the correct order relative to one, but not both, of the other polygons. This would result in an infinite loop. To avoid looping, we must modify our approach by marking each polygon that is moved to the end of the list. Then, whenever the first five tests fail and the current polygon  $Q$  is marked, we do not try tests 3' and 4'. Instead, we split either  $P$  or  $Q$  (as if tests 3' and 4' had both failed) and reinsert the pieces.

Can two polygons fail all the tests even when they are already ordered correctly? Consider  $P$  and  $Q$  in Fig. 15.27(a). Only the  $z$  coordinate of each vertex is shown. With  $P$  and  $Q$  in their current position, both the simple painter's algorithm and the full depth-sort algorithm scan convert  $P$  first. Now, rotate  $Q$  clockwise in its plane until it begins to obscure  $P$ , but do not allow  $P$  and  $Q$  themselves to intersect, as shown in Fig. 15.27(b). (You can do this nicely using your hands as  $P$  and  $Q$ , with your palms facing you.)  $P$  and  $Q$

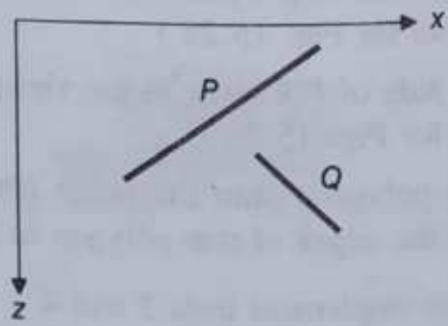
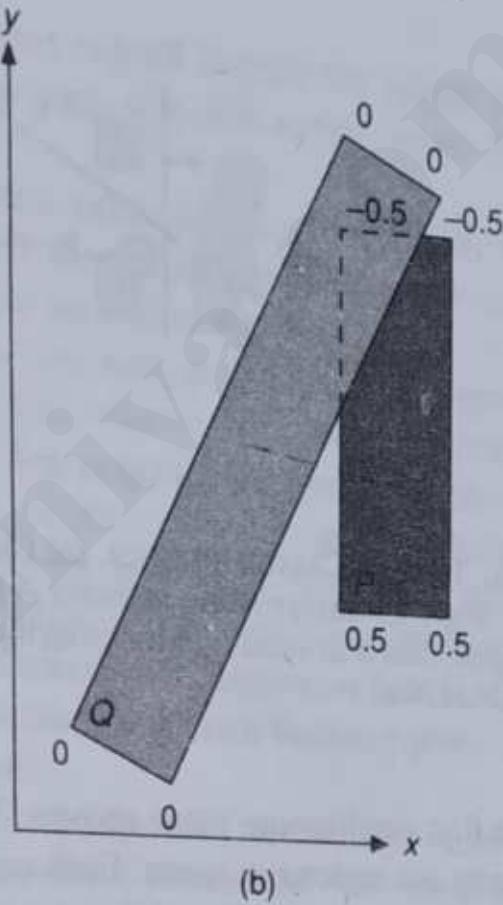
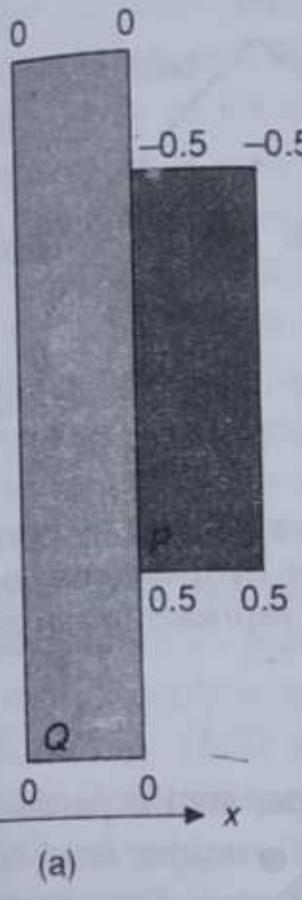


Fig. 15.26 Test 3 is false, but test 4 is true.



**Fig. 15.27** Correctly ordered polygons may be split by the depth-sort algorithm. Polygon vertices are labeled with their  $z$  values. (a) Polygons  $P$  and  $Q$  are scan-converted without splitting. (b) Polygons  $P$  and  $Q$  fail all five tests even though they are correctly ordered.

have overlapping  $z$  extents, so they must be compared. Note that tests 1 and 2 ( $x$  and  $y$  extent) fail, tests 3 and 4 fail because neither is wholly in one half-space of the other, and test 5 fails because the projections overlap. Since tests 3' and 4' also fail, a polygon will be split, even though  $P$  can be scan-converted before  $Q$ . Although the simple painter's algorithm would correctly draw  $P$  first because  $P$  has the smallest minimum  $z$  coordinate, try the example again with  $z = -0.5$  at  $P$ 's bottom and  $z = 0.5$  at  $P$ 's top.

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

## 15.7 AREA-SUBDIVISION ALGORITHMS

*Area-subdivision* algorithms all follow the divide-and-conquer strategy of spatial partitioning in the projection plane. An area of the projected image is examined. If it is easy to decide which polygons are visible in the area, they are displayed. Otherwise, the area is subdivided into smaller areas to which the decision logic is applied recursively. As the areas become smaller, fewer polygons overlap each area, and ultimately a decision becomes possible. This approach exploits *area coherence*, since sufficiently small areas of an image will be contained in at most a single visible polygon.

### 15.7.1 Warnock's Algorithm

The area-subdivision algorithm developed by Warnock [WARN69] subdivides each area into four equal squares. At each stage in the recursive-subdivision process, the projection of each polygon has one of four relationships to the area of interest (see Fig. 15.42):

1. *Surrounding polygons* completely contain the (shaded) area of interest (Fig. 15.42a)
2. *Intersecting polygons* intersect the area (Fig. 15.42b)
3. *Contained polygons* are completely inside the area (Fig. 15.42c)
4. *Disjoint polygons* are completely outside the area (Fig. 15.42d).

Disjoint polygons clearly have no influence on the area of interest. The part of an intersecting polygon that is outside the area is also irrelevant, whereas the part of an intersecting polygon that is interior to the area is the same as a contained polygon and can be treated as such.

In four cases, a decision about an area can be made easily, so the area does not need to be divided further to be conquered:

1. All the polygons are disjoint from the area. The background color can be displayed in the area.

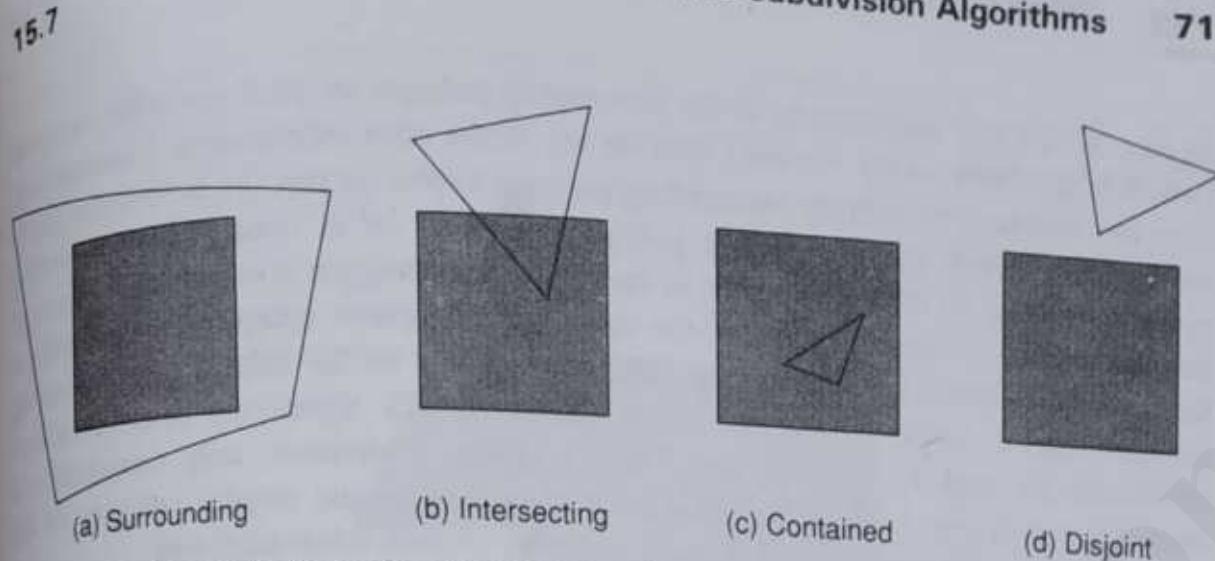
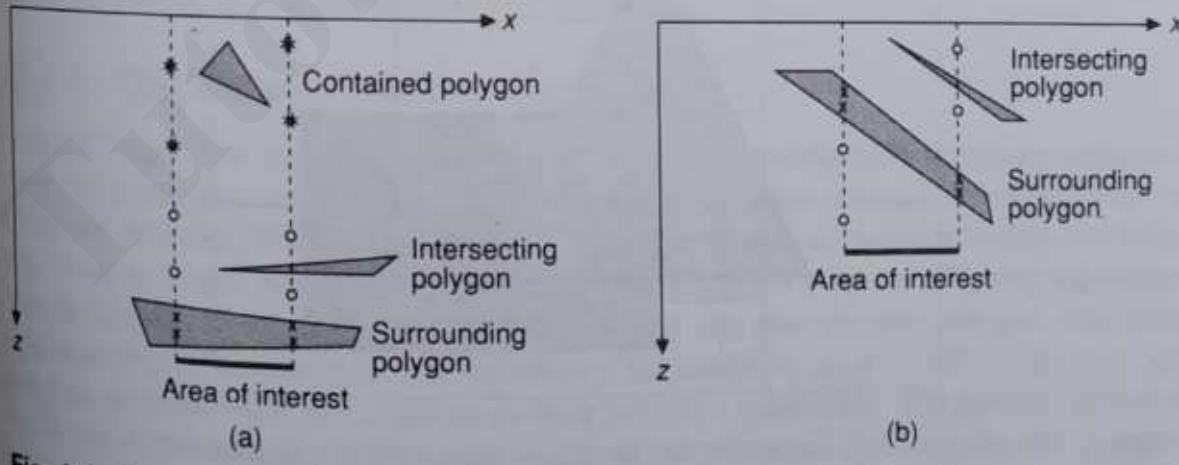


Fig. 15.42 Four relations of polygon projections to an area element: (a) surrounding, (b) intersecting, (c) contained, and (d) disjoint.

2. There is only one intersecting or only one contained polygon. The area is first filled with the background color, and then the part of the polygon contained in the area is scan-converted.
  3. There is a single surrounding polygon, but no intersecting or contained polygons. The area is filled with the color of the surrounding polygon.
  4. More than one polygon is intersecting, contained in, or surrounding the area, but one is a surrounding polygon that is in front of all the other polygons. Determining whether a surrounding polygon is in front is done by computing the  $z$  coordinates of the planes of all surrounding, intersecting, and contained polygons at the four corners of the area; if there is a surrounding polygon whose four corner  $z$  coordinates are larger (closer to the viewpoint) than are those of any of the other polygons, then the entire area can be filled with the color of this surrounding polygon.

Cases 1, 2, and 3 are simple to understand. Case 4 is further illustrated in Fig. 15.43.



**Fig. 15.43** Two examples of case 4 in recursive subdivision. (a) Surrounding polygon is closest at all corners of area of interest. (b) Intersecting polygon plane is closest at left side of area of interest.  $\times$  marks the intersection of surrounding polygon plane;  $\circ$  marks the intersection of intersecting polygon plane;  $*$  marks the intersection of contained polygon plane.

In part (a), the four intersections of the surrounding polygon are all closer to the viewpoint (which is at infinity on the  $+z$  axis) than are any of the other intersections. Consequently, the entire area is filled with the surrounding polygon's color. In part (b), no decision can be made, even though the surrounding polygon seems to be in front of the intersecting polygon, because on the left the plane of the intersecting polygon is in front of the plane of the surrounding polygon. Note that the depth-sort algorithm accepts this case without further subdivision if the intersecting polygon is wholly on the side of the surrounding polygon that is farther from the viewpoint. Warnock's algorithm, however, always subdivides the area to simplify the problem. After subdivision, only contained and intersecting polygons need to be reexamined: Surrounding and disjoint polygons of the original area are surrounding and disjoint polygons of each subdivided area.

Up to this point, the algorithm has operated at object precision, with the exception of the actual scan conversion of the background and clipped polygons in the four cases. These image-precision scan-conversion operations, however, can be replaced by object-precision operations that output a precise representation of the visible surfaces: either a square of the area's size (cases 1, 3, and 4) or a single polygon clipped to the area, along with its Boolean complement relative to the area, representing the visible part of the background (case 2). What about the cases that are not one of these four? One approach is to stop subdividing when the resolution of the display device is reached. Thus, on a 1024 by 1024 raster display, at most 10 levels of subdivision are needed. If, after this maximum number of subdivisions, none of cases 1 to 4 have occurred, then the depth of all relevant polygons is computed at the center of this pixel-sized, indivisible area. The polygon with the closest  $z$  coordinate defines the shading of the area. Alternatively, for antialiasing, several further levels of subdivision can be used to determine a pixel's color by weighting the color of each

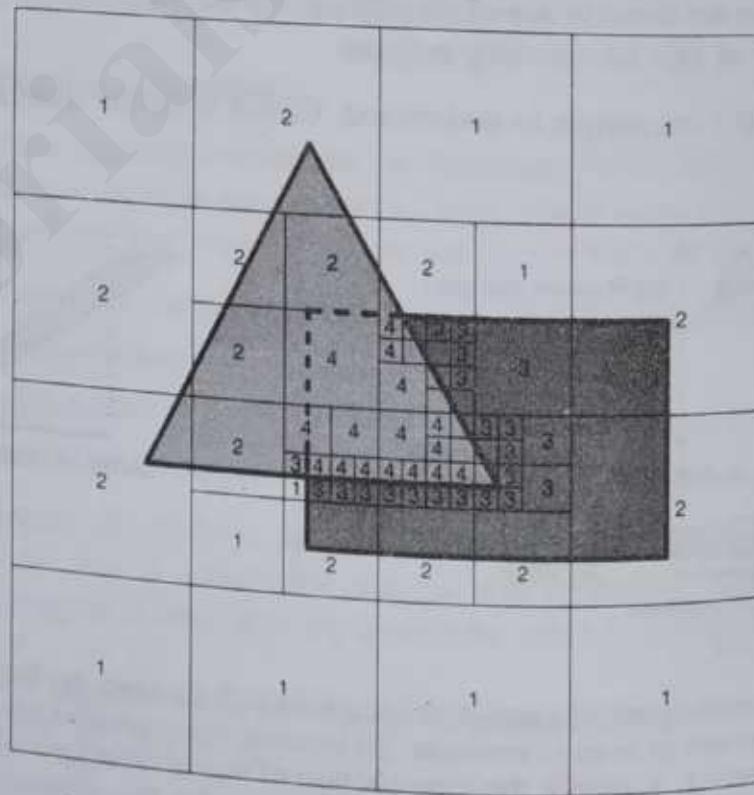
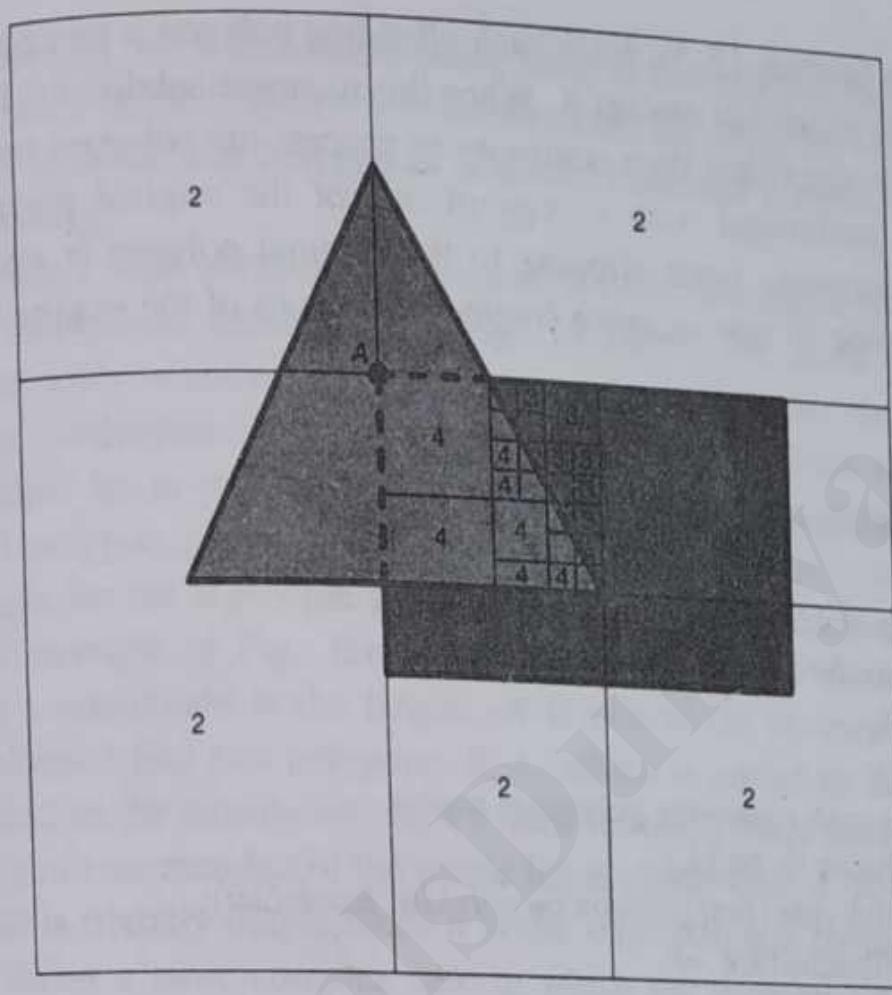


Fig. 15.44 Area subdivision into squares.



**Fig. 15.45** Area subdivision about circled polygon vertices. The first subdivision is at vertex *A*; the second is at vertex *B*.

of its subpixel-sized areas by its size. It is these image-precision operations, performed when an area is not one of the simple cases, that makes this an image-precision approach.

Figure 15.44 shows a simple scene and the subdivisions necessary for that scene's display. The number in each subdivided area corresponds to one of the four cases; in unnumbered areas, none of the four cases are true. Compare this approach to the 2D spatial partitioning performed by quadtrees (Section 12.6.3). An alternative to equal-area subdivision, shown in Fig. 15.45, is to divide about the vertex of a polygon (if there is a vertex in the area) in an attempt to avoid unnecessary subdivisions. Here subdivision is limited to a depth of five for purposes of illustration.

## EXERCISES

- 15.1 Prove that the transformation  $M$  in Section 15.2.2 preserves (a) straight lines, (b) planes, and (c) depth relationships.
- 15.2 Given a plane  $Ax + By + Cz + D = 0$ , apply  $M$  from Section 15.2.2 and find the new coefficients of the plane equation.
- 15.3 How can a scan-line algorithm be extended to deal with polygons with shared edges? Should a shared edge be represented once, as a shared edge, or twice, once for each polygon it borders, with no record kept that it is a shared edge? When the depth of two polygons is evaluated at their common shared edge, the depths will, of course, be equal. Which polygon should be declared visible, given that the scan is entering both?
- 15.4 Warnock's algorithm generates a quadtree. Show the quadtree corresponding to Fig. 15.44. Label all nodes to indicate how the triangle ( $T$ ) and the rectangle ( $R$ ) relate to the node, as (a) surrounding, (b) intersecting, (c) contained, and (d) disjoint.
- 15.5 For each of the visible-surface algorithms discussed, explain how piercing polygons would be handled. Are they a special case that must be treated explicitly, or are they accommodated by the basic algorithm?
- 15.6 Consider tests 3 and 4 of the depth-sort algorithm. How might they be implemented efficiently? Consider examining the sign of the equation of the plane of polygon  $P$  for each vertex of polygon  $Q$  and vice versa. How do you know to which side of the plane a positive value of the equation corresponds?
- 15.7 How can the algorithms discussed be adapted to work with polygons containing holes?
- 15.8 Describe how the visible-line algorithms for functions of two variables, described in Section 15.1, can be modified to work as visible-surface algorithms using the approach taken in the painter's algorithm.
- 15.9 Why does the Roberts visible-line algorithm not eliminate *all* lines that are edges of a back-facing polygon?
- 15.10 One of the advantages of the  $z$ -buffer algorithm is that primitives may be presented to it in any order. Does this mean that two images created by sending primitives in different orders will have identical values in their  $z$ -buffers and in their frame buffers? Explain your answer.
- 15.11 Consider merging two images of identical size, represented by their frame-buffer and  $z$ -buffer contents. If you know the  $z_{\min}$  and  $z_{\max}$  of each image and the values of  $z$  to which they originally corresponded, can you merge the images properly? Is any additional information needed?
- 15.12 Section 15.4 mentions the  $z$ -compression problems caused by rendering a perspective projection using an integer  $z$ -buffer. Choose a perspective viewing specification and a small number of object points. Show how, in the perspective transformation, two points near the center of projection are mapped to different  $z$  values, whereas two points separated from each other by the same distance, but farther from the center of projection, are mapped to a single  $z$  value.
- 15.13 a. Suppose view volume  $V$  has a front clipping plane at distance  $F$  and a back clipping plane at distance  $B$ , and that view volume  $V'$  has clipping planes at  $F'$  and  $B'$ . After transformation of each view volume to the canonical-perspective view volume, the back clipping plane of  $V$  will be at  $z = -1$ , and the front clipping plane at  $z = A$ . For  $V'$ , the front clipping plane will be at  $z = A'$ . Show that, if  $B/F = B'/F'$ , then  $A = A'$ .

- b. Part (a) shows that, in considering the effect of perspective, we need to consider only the ratio of back-plane to front-plane distance. We can therefore simply study the canonical view volume with various values of the front-plane distance. Suppose, then, that we have a canonical-perspective view volume, with front clipping plane  $z = A$  and back clipping plane  $z = -1$ , and we transform it, through the perspective transformation, to the parallel view volume between  $z = 0$  and  $z = -1$ . Write down the formula for the transformed  $z$  coordinate in terms of the original  $z$  coordinate. (Your answer will depend on  $A$ , of course.) Suppose that the transformed  $z$  values in the parallel view volume are multiplied by  $2^n$ , and then are rounded to integers (i.e., they are mapped to an integer  $z$ -buffer). Find two values of  $z$  that are as far apart as possible, but that map, under this transformation, to the same integer. (Your answer will depend on  $n$  and  $A$ .)
- c. Suppose you want to make an image in which the backplane-to-frontplane ratio is  $R$ , and objects that are more than distance  $Q$  apart (in  $z$ ) must map to different values in the  $z$ -buffer. Using your work in part (b), write a formula for the number of bits of  $z$ -buffer needed.

- 15.14 Show that the back-to-front display order determined by traversing a BSP tree is not necessarily the same as the back-to-front order determined by the depth-sort algorithm, even when no polygons are split. (Hint: Only two polygons are needed.)
- 15.15 How might you modify the BSP-tree algorithm to accept objects other than polygons?
- 15.16 How might you modify the BSP-tree algorithm to allow limited motion?
- 15.17 Suppose that you are designing a ray tracer that supports CSG. How would you handle a polygon that is not part of a polyhedron?
- 15.18 Some graphics systems implement hardware transformations and homogeneous-coordinate clipping in  $X$ ,  $Y$ , and  $Z$  using the same mathematics, so that clipping limits are

$$-W \leq X \leq W, \quad -W \leq Y \leq W, \quad -W \leq Z \leq W,$$

instead of

$$-W \leq X \leq W, \quad -W \leq Y \leq W, \quad -W \leq Z \leq 0.$$

How would you change the viewing matrix calculation to take this into account?

- 15.19 When ray tracing is performed, it is typically necessary to compute only whether or not a ray intersects an extent, not what the actual points of intersection are. Complete the ray-sphere intersection equation (Eq. 15.17) using the quadratic formula, and show how it can be simplified to determine only whether or not the ray and sphere intersect.

- 15.20 Ray tracing can also be used to determine the mass properties of objects through numerical integration. The full set of intersections of a ray with an object gives the total portion of the ray that is inside the object. Show how you can estimate an object's volume by firing a regular array of parallel rays through that object.

- 15.21 Derive the intersection of a ray with a quadric surface. Modify the method used to derive the intersection of a ray with a sphere in Eqs. (15.13) through (15.16) to handle the definition of a quadric given in Section 11.4.

- 15.22 In Eq. (15.5),  $O$ , the cost of performing an object intersection test, may be partially overwritten by  $B$ , the cost of performing a bounding-volume intersection test, if the results of the

bounding-volume intersection test can be reused to simplify the object intersection test. Describe an object and bounding volume for which this is possible.

15.23 Implement one of the polygon visible surface algorithms in this chapter, such as a z-buffer algorithm, scan-line algorithm, or BSP tree algorithm.

15.24 Implement a simple ray tracer for spheres and polygons, including adaptive supersampling (Choose one of the illumination models from Section 16.1.) Improve your program's performance through the use of spatial partitioning or hierarchies of bounding volumes.

15.25 If you have implemented the z-buffer algorithm, then add hit detection to it by extending the pick-window approach described in Section 7.12.2 to take visible-surface determination into account. You will need a SetPickMode procedure that is passed a mode flag, indicating whether objects are to be drawn (drawing mode) or instead tested for hits (pick mode). A SetPick Window procedure will let the user set a rectangular pick window. The z-buffer must already have been filled (by drawing all objects) for pick mode to work. When in pick mode, neither the frame-buffer nor the z-buffer is updated, but the z-value of each of the primitive's pixels that falls inside the pick window is compared with the corresponding value in the z-buffer. If the new value would have caused the object to be drawn in drawing mode, then a flag is set. The flag can be inquired by calling InquirePick, which then resets the flag. If InquirePick is called after each primitive's routine is called in pick mode, picking can be done on a per-primitive basis. Show how you can use InquirePick to determine which object is actually visible at a pixel.

## Surface rendering

S. No.	Topic	Contents
7.	<b>Color Models, Illumination and shading models, Computer Animation</b>	<b>Sections 14.1 - 14.2</b> <b>Sections 14.4 - 14.5</b> <b>Sections 15.3 - 15.7</b> <b>Sections 16.1 - 16.6</b>

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>

CHAPTER

# 14 Illumination Models and Surface-Rendering Methods



---

**R**ealistic displays of a scene are obtained by generating perspective projections of objects and by applying natural lighting effects to the visible surfaces. An **illumination model**, also called a **lighting model** and sometimes referred to as a **shading model**, is used to calculate the intensity of light that we should see at a given point on the surface of an object. A **surface-rendering algorithm** uses the intensity calculations from an illumination model to determine the light intensity for all projected pixel positions for the various surfaces 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 surfaces from a small set of illumination-model calculations. Scan-line image-space algorithms typically use interpolation schemes, while ray-tracing algorithms invoke the illumination model at each pixel position. Sometimes, surface-rendering procedures are termed *surface-shading methods*. To avoid confusion, we will refer to the model for calculating light intensity at a single surface point as an **illumination model** or a **lighting model**, and we will use the term **surface rendering** to mean a procedure for applying a lighting model to obtain pixel intensities for all the projected surface positions in a scene.

Photorealism in computer graphics involves two elements: accurate graphical representations of objects and good physical descriptions of the lighting effects in a scene. Lighting effects include light reflections, transparency, surface texture, and shadows.

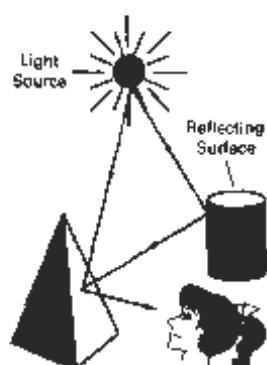
Modeling the colors and lighting effects that we see on an object is a complex process, involving principles of both physics and psychology. Fundamentally, lighting effects are described with models that consider the interaction of electromagnetic energy with object surfaces. Once light reaches our eyes, it triggers perception processes that determine what we actually "see" in a scene. Physical illumination models involve a number of factors, such as object type, object position relative to light sources and other objects, and the light-source conditions that we set for a scene. Objects can be constructed of opaque materials, or they can be more or less transparent. In addition, they can have shiny or dull surfaces, and they can have a variety of surface-texture patterns. Light sources, of varying shapes, colors, and positions, can be used to provide the illumination effects for a scene. Given the parameters for the optical properties of surfaces, the relative positions of the surfaces in a scene, the color and positions of the light sources, and the position and orientation of the viewing plane, illumination models calculate the intensity projected from a particular surface point in a specified viewing direction.

Illumination models in computer graphics are often loosely derived from the physical laws that describe surface light intensities. To minimize intensity cal-

cations, most packages use empirical models based on simplified photometric calculations. More accurate models, such as the radiosity algorithm, calculate light intensities by considering the propagation of radiant energy between the surfaces and light sources in a scene. In the following sections, we first take a look at the basic illumination models often used in graphics packages; then we discuss more accurate, but more time-consuming, methods for calculating surface intensities. And we explore the various surface-rendering algorithms for applying the lighting models to obtain the appropriate shading over visible surfaces in a scene.

## 14-1

### LIGHT SOURCES



**Figure 14-1**  
 Light viewed from an opaque nonluminous surface is in general a combination of reflected light from a light source and reflections of light reflections from other surfaces.

When we view an opaque nonluminous object, we see reflected light from the surfaces of the object. The total reflected light is the sum of the contributions from light sources and other reflecting surfaces in the scene (Fig. 14-1). Thus, a surface that is not directly exposed to a light source may still be visible if nearby objects are illuminated. Sometimes, light sources are referred to as *light-emitting sources*; and reflecting surfaces, such as the walls of a room, are termed *light-reflecting sources*. We will use the term *light source* to mean an object that is emitting radiant energy, such as a light bulb or the sun.

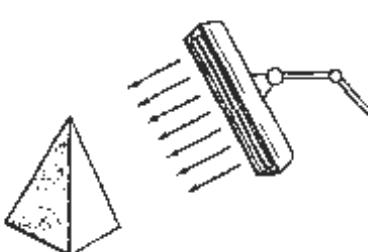
A luminous object, in general, can be both a light source and a light reflector. For example, a plastic globe with a light bulb inside both emits and reflects light from the surface of the globe. Emitted light from the globe may then illuminate other objects in the vicinity.

The simplest model for a light emitter is a *point source*. Rays from the source then follow radially diverging paths from the source position, as shown in Fig. 14-2. This light-source model is a reasonable approximation for sources whose dimensions are small compared to the size of objects in the scene. Sources, such as the sun, that are sufficiently far from the scene can be accurately modeled as point sources. A nearby source, such as the long fluorescent light in Fig. 14-3, is more accurately modeled as a *distributed light source*. In this case, the illumination effects cannot be approximated realistically with a point source, because the area of the source is not small compared to the surfaces in the scene. An accurate model for the distributed source is one that considers the accumulated illumination effects of the points over the surface of the source.

When light is incident on an opaque surface, part of it is reflected and part is absorbed. The amount of incident light reflected by a surface depends on the type of material. Shiny materials reflect more of the incident light, and dull surfaces absorb more of the incident light. Similarly, for an illuminated transparent



**Figure 14-2**  
 Diverging ray paths from a point light source.



**Figure 14-3**  
 An object illuminated with a distributed light source.

surface, some of the incident light will be reflected and some will be transmitted through the material.

Surfaces that are rough, or grainy, tend to scatter the reflected light in all directions. This scattered light is called **diffuse reflection**. A very rough matte surface produces primarily diffuse reflections, so that the surface appears equally bright from all viewing directions. Figure 14-4 illustrates diffuse light scattering from a surface. What we call the color of an object is the color of the diffuse reflection of the incident light. A blue object illuminated by a white light source, for example, reflects the blue component of the white light and totally absorbs all other components. If the blue object is viewed under a red light, it appears black since all of the incident light is absorbed.

In addition to diffuse reflection, light sources create highlights, or bright spots, called **specular reflection**. This highlighting effect is more pronounced on shiny surfaces than on dull surfaces. An illustration of specular reflection is shown in Fig. 14-5.

## Section 14-2

### Basic Illumination Models

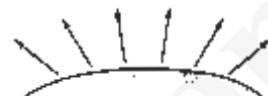


Figure 14-4  
Diffuse reflections from a surface.

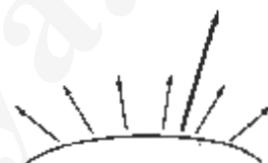


Figure 14-5  
Specular reflection superimposed on diffuse reflection vector.

## 14-2

### BASIC ILLUMINATION MODELS

Here we discuss simplified methods for calculating light intensities. The empirical models described in this section provide simple and fast methods for calculating surface intensity at a given point, and they produce reasonably good results for most scenes. Lighting calculations are based on the optical properties of surfaces, the background lighting conditions, and the light-source specifications. Optical parameters are used to set surface properties, such as glossy, matte, opaque, and transparent. This controls the amount of reflection and absorption of incident light. All light sources are considered to be point sources, specified with a coordinate position and an intensity value (color).

#### Ambient Light

A surface that is not exposed directly to a light source still will be visible if nearby objects are illuminated. In our basic illumination model, we can set a general level of brightness for a scene. This is a simple way to model the combination of light reflections from various surfaces to produce a uniform illumination called the **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.

We can set the level for the ambient light in a scene with parameter  $t_a$ , and each surface is then illuminated with this constant value. The resulting reflected light is a constant for each surface, independent of the viewing direction and the spatial orientation of the surface. But the intensity of the reflected light for each surface depends on the optical properties of the surface; that is, how much of the incident energy is to be reflected and how much absorbed.

#### Diffuse Reflection

Ambient-light reflection is an approximation of global diffuse lighting effects. Diffuse reflections are constant over each surface in a scene, independent of the viewing direction. The fractional amount of the incident light that is diffusely re-

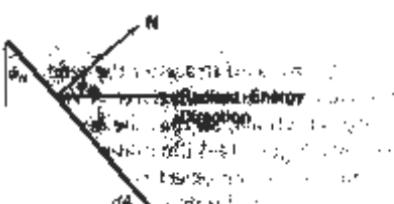


Figure 14-6

Radiant energy from a surface area  $dA$  in direction  $\phi_N$  relative to the surface normal direction.

flected can be set for each surface with parameter  $k_d$ , the **diffuse-reflection coefficient**, or **diffuse reflectivity**. Parameter  $k_d$  is assigned a constant value in the interval 0 to 1, according to the reflecting properties we want the surface to have. If we want a highly reflective surface, we set the value of  $k_d$  near 1. This produces a bright surface with the intensity of the reflected light near that of the incident light. To simulate a surface that absorbs most of the incident light, we set the reflectivity to a value near 0. Actually, parameter  $k_d$  is a function of surface color, but for the time being we will assume  $k_d$  is a constant.

If a surface is exposed only to ambient light, we can express the intensity of the diffuse reflection at any point on the surface as

$$I_{\text{amb}} = k_d I_0 \quad (14-1)$$

Since ambient light produces a flat uninteresting shading for each surface (Fig. 14-19(b)), scenes are rarely rendered with ambient light alone. At least one light source is included in a scene, often as a point source at the viewing position.

We can model the diffuse reflections of illumination from a point source in a similar way. That is, we assume that the diffuse reflections from the surface are scattered with equal intensity in all directions, independent of the viewing direction. Such surfaces are sometimes referred to as *ideal diffuse reflectors*. They are also called *Lambertian reflectors*, since radiated light energy from any point on the surface is governed by *Lambert's cosine law*. This law states that the radiant energy from any small surface area  $dA$  in any direction  $\phi_N$  relative to the surface normal is proportional to  $\cos\phi_N$  (Fig. 14-6). The light intensity, though, depends on the radiant energy per projected area perpendicular to direction  $\phi_N$ , which is  $dA \cos\phi_N$ . Thus, for Lambertian reflection, the intensity of light is the same over all viewing directions. We discuss photometry concepts and terms, such as radiant energy, in greater detail in Section 14-7.

Even though there is equal light scattering in all directions from a perfect diffuse reflector, the brightness of the surface does depend on the orientation of the surface relative to the light source. A surface that is oriented perpendicular to the direction of the incident light appears brighter than if the surface were tilted at an oblique angle to the direction of the incoming light. This is easily seen by holding a white sheet of paper or smooth cardboard parallel to a nearby window and slowly rotating the sheet away from the window direction. As the angle between the surface normal and the incoming light direction increases, less of the incident light falls on the surface, as shown in Fig. 14-7. This figure shows a beam of light rays incident on two equal-area plane surface patches with different spatial orientations relative to the incident light direction (par-

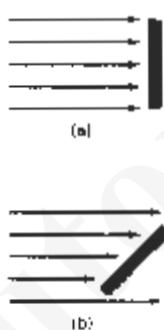


Figure 14-7

A surface perpendicular in the direction of the incident light (a) is more illuminated than an equal-sized surface at an oblique angle (b) to the incoming light direction.

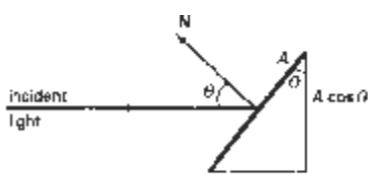


Figure 14-8  
An illuminated area projected perpendicular to the path of the incoming light rays.

allel incoming rays). If we denote the angle of incidence between the incoming light direction and the surface normal as  $\theta$  (Fig. 14-8), then the projected area of a surface patch perpendicular to the light direction is proportional to  $\cos\theta$ . Thus, the amount of illumination (or the “number of incident light rays” cutting across the projected surface patch) depends on  $\cos\theta$ . If the incoming light from the source is perpendicular to the surface at a particular point, that point is fully illuminated. As the angle of illumination moves away from the surface normal, the brightness of the point drops off. If  $I_l$  is the intensity of the point light source, then the diffuse reflection equation for a point on the surface can be written as

$$I_{\text{diff}} = k_d I_l \cos \theta \quad (14-2)$$

A surface is illuminated by a point source only if the angle of incidence is in the range  $0^\circ$  to  $90^\circ$  ( $\cos \theta$  is in the interval from 0 to 1). When  $\cos \theta$  is negative, the light source is “behind” the surface.

If  $N$  is the unit normal vector to a surface and  $L$  is the unit direction vector to the point light source from a position on the surface (Fig. 14-9), then  $\cos \theta = N \cdot L$  and the diffuse reflection equation for single point-source illumination is

$$I_{\text{diff}} = k_d I_l (N \cdot L) \quad (14-3)$$

Reflections for point-source illumination are calculated in world coordinates or viewing coordinates before shearing and perspective transformations are applied. These transformations may transform the orientation of normal vectors so that they are no longer perpendicular to the surfaces they represent. Transformation procedures for maintaining the proper orientation of surface normals are discussed in Chapter 11.

Figure 14-10 illustrates the application of Eq. 14-3 to positions over the surface of a sphere, using various values of parameter  $k_d$  between 0 and 1. Each projected pixel position for the surface was assigned an intensity as calculated by the diffuse reflection equation for a point light source. The renderings in this figure illustrate single point-source lighting with no other lighting effects. This is what we might expect to see if we shined a small light on the object in a completely darkened room. For general scenes, however, we expect some background lighting effects in addition to the illumination effects produced by a direct light source.

We can combine the ambient and point-source intensity calculations to obtain an expression for the total diffuse reflection. In addition, many graphics packages introduce an ambient-reflection coefficient  $k_a$  to modify the ambient-light intensity  $I_a$  for each surface. This simply provides us with an additional parameter to adjust the light conditions in a scene. Using parameter  $k_a$ , we can write the total diffuse reflection equation as

$$I_{\text{diff}} = k_a I_a + k_d I_l (N \cdot L) \quad (14-4)$$

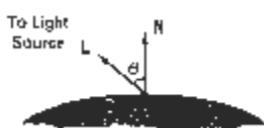
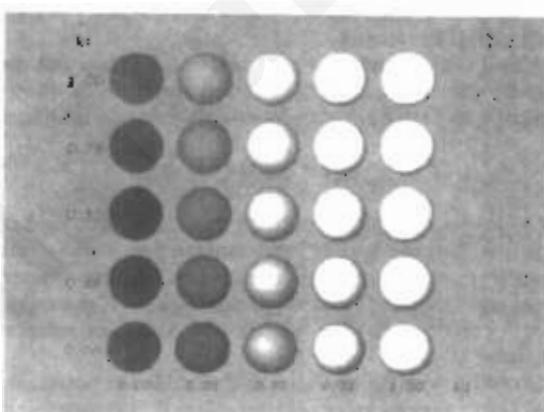


Figure 14-9  
Angle of incidence  $\theta$  between the unit light-source direction vector  $L$  and the unit surface normal  $N$ .



**Figure 14-10**  
Diffuse reflections from a spherical surface illuminated by a point light source for values of the diffuse reflectivity coefficient in the interval  $0 \leq k_d \leq 1$ .



**Figure 14-11**  
Diffuse reflections from a spherical surface illuminated with ambient light and a single point source for values of  $k_e$  and  $k_d$  in the interval  $(0, 1)$ .

where both  $k_e$  and  $k_d$  depend on surface material properties and are assigned values in the range from 0 to 1. Figure 14-11 shows a sphere displayed with surface intensities calculated from Eq. 14-4 for values of parameters  $k_e$  and  $k_d$  between 0 and 1.

#### Specular Reflection and the Phong Model

When we look at an illuminated shiny surface, such as polished metal, an apple, or a person's forehead, we see a highlight, or bright spot, at certain viewing di-

rections. This phenomenon, called **specular reflection**, is the result of total, or near total, reflection of the incident light in a concentrated region around the **specular-reflection angle**. Figure 14-12 shows the specular reflection direction at a point on the illuminated surface. The specular-reflection angle equals the angle of the incident light, with the two angles measured on opposite sides of the unit normal surface vector  $\mathbf{N}$ . In this figure, we use  $\mathbf{R}$  to represent the unit vector in the direction of ideal specular reflection;  $\mathbf{L}$  to represent the unit vector directed toward the point light source; and  $\mathbf{V}$  as the unit vector pointing to the viewer from the surface position. Angle  $\phi$  is the viewing angle relative to the specular-reflection direction  $\mathbf{R}$ . For an ideal reflector (perfect mirror), incident light is reflected only in the specular-reflection direction. In this case, we would only see reflected light when vectors  $\mathbf{V}$  and  $\mathbf{R}$  coincide ( $\phi = 0$ ).

Objects other than ideal reflectors exhibit specular reflections over a finite range of viewing positions around vector  $\mathbf{R}$ . Shiny surfaces have a narrow specular-reflection range, and dull surfaces have a wider reflection range. An empirical model for calculating the specular-reflection range, developed by Phong Bui Tuong, and called the **Phong specular-reflection model**, or simply the **Phong model**, sets the intensity of specular reflection proportional to  $\cos^n \phi$ . Angle  $\phi$  can be assigned values in the range  $0^\circ$  to  $90^\circ$ , so that  $\cos \phi$  varies from 0 to 1. The value assigned to **specular-reflection parameter**  $n_s$  is determined by the type of surface that we want to display. A very shiny surface is modeled with a large value for  $n_s$  (say, 100 or more), and smaller values (down to 1) are used for duller surfaces. For a perfect reflector,  $n_s$  is infinite. For a rough surface, such as chalk or cinderblock,  $n_s$  would be assigned a value near 1. Figures 14-13 and 14-14 show the effect of  $n_s$  on the angular range for which we can expect to see specular reflections.

The intensity of specular reflection depends on the material properties of the surface and the angle of incidence, as well as other factors such as the polarization and color of the incident light. We can approximately model monochromatic specular intensity variations using a **specular-reflection coefficient**,  $W(\theta)$ , for each surface. Figure 14-15 shows the general variation of  $W(\theta)$  over the range  $\theta = 0^\circ$  to  $\theta = 90^\circ$  for a few materials. In general,  $W(\theta)$  tends to increase as the angle of incidence increases. At  $\theta = 90^\circ$ ,  $W(\theta) = 1$  and all of the incident light is reflected. The variation of specular intensity with angle of incidence is described by *Fresnel's Laws of Reflection*. Using the spectral-reflection function  $W(\theta)$ , we can write the Phong specular-reflection model as

$$I_{\text{spec}} = W(\theta) I_i \cos^{n_s} \phi \quad (14.5)$$

where  $I_i$  is the intensity of the light source, and  $\phi$  is the viewing angle relative to the specular-reflection direction  $\mathbf{R}$ .

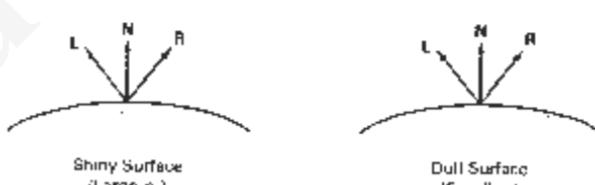


Figure 14-13  
Modeling specular reflections (shaded area) with parameter  $n_s$

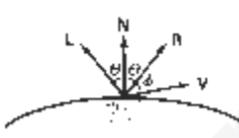


Figure 14-12  
Specular-reflection angle equals angle of incidence  $\theta$ .

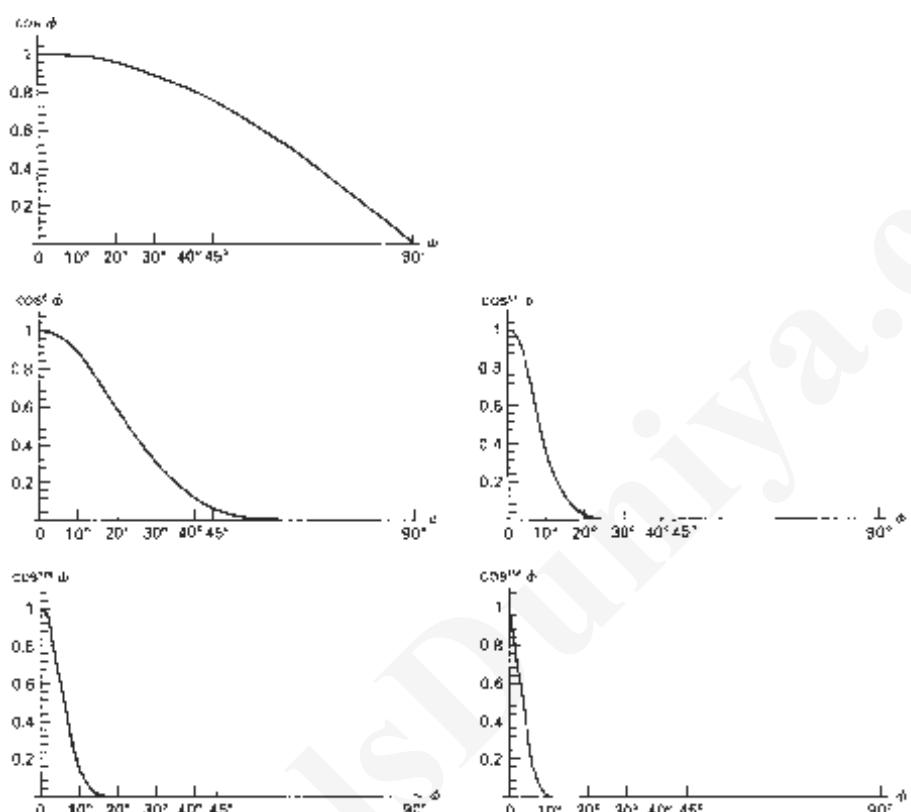


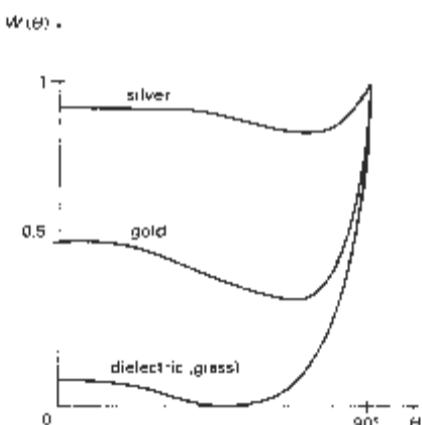
Figure 14-14  
Plots of  $\cos^n \phi$  for several values of specular parameter  $n$ .

As seen in Fig. 14-13, transparent materials, such as glass, only exhibit appreciable specular reflections as  $\theta$  approaches  $90^\circ$ . At  $\theta = 0^\circ$ , about 4 percent of the incident light on a glass surface is reflected. And for most of the range of  $\theta$ , the reflected intensity is less than 10 percent of the incident intensity. But for many opaque materials, specular reflection is nearly constant for all incidence angles. In this case, we can reasonably model the reflected light effects by replacing  $W(\theta)$  with a constant specular-reflection coefficient  $k_s$ . We then simply set  $k_s$  equal to some value in the range 0 to 1 for each surface.

Since  $V$  and  $R$  are unit vectors in the viewing and specular reflection directions, we can calculate the value of  $\cos\phi$  with the dot product  $V \cdot R$ . Assuming the specular-reflection coefficient is a constant, we can determine the intensity of the specular reflection at a surface point with the calculation

$$I_{spec} = k_s I(V \cdot R)^n$$

(14-11)

**Section 14-2****Basic Illumination Models**

**Figure 14-15**  
Approximate variation of the specular-reflection coefficient as a function of angle of incidence for different materials.

Vector  $\mathbf{R}$  in this expression can be calculated in terms of vectors  $\mathbf{L}$  and  $\mathbf{N}$ . As seen in Fig. 14-16, the projection of  $\mathbf{L}$  onto the direction of the normal vector is obtained with the dot product  $\mathbf{N} \cdot \mathbf{L}$ . Therefore, from the diagram, we have

$$\mathbf{R} - \mathbf{L} = (2\mathbf{N} \cdot \mathbf{L})\mathbf{N}$$

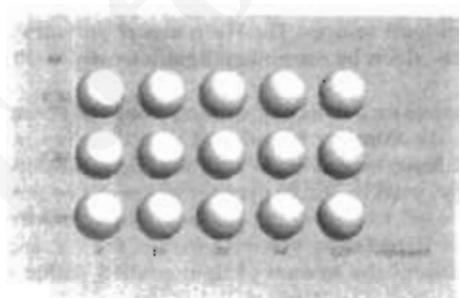
and the specular-reflection vector is obtained as

$$\mathbf{R} = (2\mathbf{N} \cdot \mathbf{L})\mathbf{N} - \mathbf{L}. \quad (14-7)$$

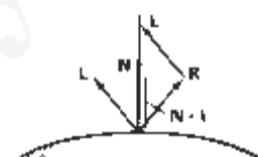
Figure 14-17 illustrates specular reflections for various values of  $k_s$  and  $n_s$  on a sphere illuminated with a single point light source.

A somewhat simplified Phong model is obtained by using the *halfway vector*  $\mathbf{H}$  between  $\mathbf{L}$  and  $\mathbf{V}$  to calculate the range of specular reflections. If we replace  $\mathbf{V} \cdot \mathbf{R}$  in the Phong model with the dot product  $\mathbf{N} \cdot \mathbf{H}$ , this simply replaces the empirical  $\cos \phi$  calculation with the empirical  $\cos \alpha$  calculation (Fig. 14-18). The halfway vector is obtained as

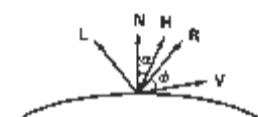
$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{|\mathbf{L} + \mathbf{V}|} \quad (14-8)$$



**Figure 14-17**  
Specular reflections from a spherical surface for varying specular parameter values and a single light source.



**Figure 14-16**  
Calculation of vector  $\mathbf{R}$  by considering projections onto the direction of the normal vector  $\mathbf{N}$ .



**Figure 14-18**  
Halfway vector  $\mathbf{H}$  along the bisector of the angle between  $\mathbf{L}$  and  $\mathbf{V}$ .

If both the viewer and the light source are sufficiently far from the surface, both  $\mathbf{V}$  and  $\mathbf{L}$  are constant over the surface, and thus  $\mathbf{H}$  is also constant for all surface points. For nonplanar surfaces,  $\mathbf{N} \cdot \mathbf{H}$  then requires less computation than  $\mathbf{V} \cdot \mathbf{R}$  since the calculation of  $\mathbf{R}$  at each surface point involves the variable vector  $\mathbf{N}$ .

For given light-source and viewer positions, vector  $\mathbf{H}$  is the orientation direction for the surface that would produce maximum specular reflection in the viewing direction. For this reason,  $\mathbf{H}$  is sometimes referred to as the surface orientation direction for maximum highlights. Also, if vector  $\mathbf{V}$  is coplanar with vectors  $\mathbf{L}$  and  $\mathbf{R}$  (and thus  $\mathbf{N}$ ), angle  $\alpha$  has the value  $\phi/2$ . When  $\mathbf{V}$ ,  $\mathbf{L}$ , and  $\mathbf{N}$  are not coplanar,  $\alpha > \phi/2$ , depending on the spatial relationship of the three vectors.

### Combined Diffuse and Specular Reflections with Multiple Light Sources

For a single point light source, we can model the combined diffuse and specular reflections from a point on an illuminated surface as

$$\begin{aligned} I &= I_{\text{diff}} + I_{\text{spec}} \\ &= k_d I_o + k_s I_o (\mathbf{N} \cdot \mathbf{L}) + k_s I_o (\mathbf{N} \cdot \mathbf{H})^{\gamma} \end{aligned} \quad (14-9)$$

Figure 14-19 illustrates surface lighting effects produced by the various terms in Eq. 14-9. If we place more than one point source in a scene, we obtain the light reflection at any surface point by summing the contributions from the individual sources:

$$I = k_d I_o + \sum_{i=1}^n I_i [k_s(\mathbf{N} \cdot \mathbf{L}_i) + k_s(\mathbf{N} \cdot \mathbf{H}_i)^{\gamma}] \quad (14-10)$$

To ensure that any pixel intensity does not exceed the maximum allowable value, we can apply some type of normalization procedure. A simple approach is to set a maximum magnitude for each term in the intensity equation. If any calculated term exceeds the maximum, we simply set it to the maximum value. Another way to compensate for intensity overflow is to normalize the individual terms by dividing each by the magnitude of the largest term. A more complicated procedure is first to calculate all pixel intensities for the scene, then the calculated intensities are scaled onto the allowable intensity range.

### Warn Model

So far we have considered only point light sources. The **Warn model** provides a method for simulating studio lighting effects by controlling light intensity in different directions.

Light sources are modeled as points on a reflecting surface, using the Phong model for the surface points. Then the intensity in different directions is controlled by selecting values for the Phong exponent. In addition, light controls, such as "bam doors" and spotighting, used by studio photographers can be simulated in the Warn model. Flaps are used to control the amount of light emitted by a source in various directions. Two flaps are provided for each of the  $x$ ,  $y$ , and  $z$  directions. Spotlights are used to control the amount of light emitted within a cone with apex at a point-source position. The Warn model is implemented in

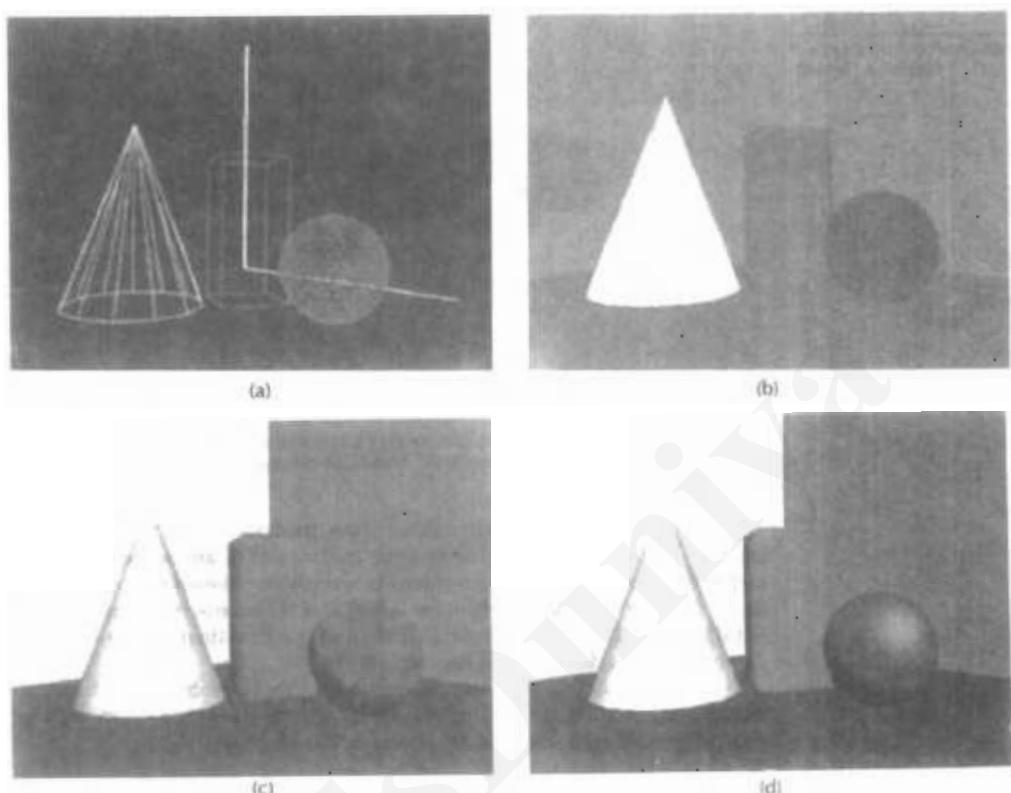


Figure 14-19

A wireframe scene (a) is displayed only with ambient lighting in (b), and the surface of each object is assigned a different color. Using ambient light and diffuse reflections due to a single source with  $k_s = 0$  for all surfaces, we obtain the lighting effects shown in (c). Using ambient light and both diffuse and specular reflections due to a single light source, we obtain the lighting effects shown in (d).

PHIGS+, and Fig. 14-20 illustrates lighting effects that can be produced with this model.

#### Intensity Attenuation

As radiant energy from a point 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. This means that a surface close to the light source (small  $d$ ) receives a higher incident intensity from the source than a distant surface (large  $d$ ). Therefore, 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 displayed as one surface.



*Figure 14-20*  
Studio lighting effects produced with the Warn model, using five light sources to illuminate a Chevrolet Camaro. (Courtesy of David R. Warn, General Motors Research Laboratories.)

Our simple point-source illumination model, however, does not always produce realistic pictures, if we use the factor  $1/d^2$  to attenuate intensities. The factor  $1/d^2$  produces too much intensity variations when  $d$  is small, and it produces very little variation when  $d$  is large. This is because real scenes are usually not illuminated with point light sources, and our illumination model is too simple to accurately describe real lighting effects.

Graphics packages have compensated for these problems by using inverse linear or quadratic functions of  $d$  to attenuate intensities. For example, a general inverse quadratic attenuation function can be set up as

$$f(d) = \frac{1}{a_0 + a_1 d + a_2 d^2} \quad (14-11)$$

A user can then fiddle with the coefficients  $a_0$ ,  $a_1$ , and  $a_2$  to obtain a variety of lighting effects for a scene. The value of the constant term  $a_0$  can be adjusted to prevent  $f(d)$  from becoming too large when  $d$  is very small. Also, the values for the coefficients in the attenuation function, and the optical surface parameters for a scene, can be adjusted to prevent calculations of reflected intensities from exceeding the maximum allowable value. This is an effective method for limiting intensity values when a single light source is used to illuminate a scene. For multiple light-source illumination, the methods described in the preceding section are more effective for limiting the intensity range.

With a given set of attenuation coefficients, we can limit the magnitude of the attenuation function to 1 with the calculation

$$f(d) = \min\left(1, \frac{1}{a_0 + a_1 d + a_2 d^2}\right) \quad (14-12)$$

Using this function, we can then write our basic illumination model as

$$I = k_s f_s - \sum_{i=1}^n f_i(d_i) I_i [k_d(N \cdot L_i) + k_g(N \cdot H_i)^2] \quad (14-13)$$

where  $d_i$  is the distance light has traveled from light source  $i$ .

**Figure 14-21**

Light reflections from the surface of a black nylon cushion, modeled as woven cloth patterns and rendered using Monte Carlo ray-tracing methods. (Courtesy of Stephen H. Winitz, Program of Computer Graphics, Cornell University.)

**Section 14-2****Basic Illumination Models****Color Considerations**

Most graphics displays of realistic scenes are in color. But the illumination model we have described so far considers only monochromatic lighting effects. To incorporate color, we need to write the intensity equation as a function of the color properties of the light sources and object surfaces.

For an RGB description, each color in a scene is expressed in terms of red, green, and blue components. We then specify the RGB components of light-source intensities and surface colors, and the illumination model calculates the RGB components of the reflected light. One way to set surface colors is by specifying the reflectivity coefficients as three-element vectors. The diffuse reflection-coefficient vector, for example, would then have RGB components ( $k_{dr}$ ,  $k_{dg}$ ,  $k_{db}$ ). If we want an object to have a blue surface, we select a nonzero value in the range from 0 to 1 for the blue reflectivity component,  $k_{db}$ , while the red and green reflectivity components are set to zero ( $k_{dr} = k_{dg} = 0$ ). Any nonzero red or green components in the incident light are absorbed, and only the blue component is reflected. The intensity calculation for this example reduces to the single expression

$$I_B = k_{db}I_{dB} + \sum_{i=1}^n f_i(d)I_{Bi}[k_{di}(N \cdot L_i) + k_{gi}(N \cdot H_i)^{\alpha}] \quad (14-14)$$

Surfaces typically are illuminated with white light sources, and in general we can set surface color so that the reflected light has nonzero values for all three RGB components. Calculated intensity levels for each color component can be used to adjust the corresponding electron gun in an RGB monitor.

In his original specular-reflection model, Phong set parameter  $k_s$  to a constant value independent of the surface color. This produces specular reflections that are the same color as the incident light (usually white), which gives the surface a plastic appearance. For a nonplastic material, the color of the specular reflection is a function of the surface properties and may be different from both the color of the incident light and the color of the diffuse reflections. We can approximate specular effects on such surfaces by making the specular-reflection coefficient color-dependent, as in Eq. 14-14. Figure 14-21 illustrates color reflections from a matte surface, and Figs. 14-22 and 14-23 show color reflections from metal

**Figure 14-22**

Light reflections from a teapot with reflectance parameters set to simulate brushed aluminum surfaces and rendered using Monte Carlo ray-tracing methods. (Courtesy of Stephen H. Winitz, Program of Computer Graphics, Cornell University.)



**Figure 14-23**  
Light reflections from trombones  
with reflectance parameters set to  
simulate shiny brass surfaces.  
(Courtesy of SOFTIMAGE, Inc.)

surfaces. Light reflections from object surfaces due to multiple colored light sources is shown in Fig. 14-24.

Another method for setting surface color is to specify the components of diffuse and specular color vectors for each surface, while retaining the reflectivity coefficients as single-valued constants. For an RGB color representation, for instance, the components of these two surface-color vectors can be denoted as  $(S_{dR}, S_{dG}, S_{dB})$  and  $(S_{sR}, S_{sG}, S_{sB})$ . The blue component of the reflected light is then calculated as

$$I_B = k_s S_{dB} I_{sB} + \sum_{i=1}^n f_i(d) I_{hi} [k_d S_{di}(N \cdot L_i) + k_s S_{si}(N \cdot H_i)^{ns}] \quad (14-15)$$

This approach provides somewhat greater flexibility, since surface-color parameters can be set independently from the reflectivity values.

Other color representations besides RGB can be used to describe colors in a scene. And sometimes it is convenient to use a color model with more than three components for a color specification. We discuss color models in detail in the next chapter. For now, we can simply represent any component of a color specification with its spectral wavelength  $\lambda$ . Intensity calculations can then be expressed as

$$I_\lambda = k_s S_{d\lambda} I_{s\lambda} + \sum_{i=1}^n f_i(d) I_{hi} [k_d S_{d\lambda}(N \cdot L_i) + k_s S_{s\lambda}(N \cdot H_i)^{ns}] \quad (14-16)$$

#### Transparency

A transparent surface, in general, produces both reflected and transmitted light. The relative contribution of the transmitted light depends on the degree of trans-



**Figure 14-24**  
Light reflections due to multiple  
light sources of various colors.  
(Courtesy of Sun Microsystems.)

parency of the surface and whether any light sources or illuminated surfaces are behind the transparent surface. Figure 14-25 illustrates the intensity contributions to the surface lighting for a transparent object.

When a transparent surface is to be modeled, the intensity equations must be modified to include contributions from light passing through the surface. In most cases, the transmitted light is generated from reflecting objects in back of the surface, as in Fig. 14-26. Reflected light from these objects passes through the transparent surface and contributes to the total surface intensity.

Both diffuse and specular transmission can take place at the surfaces of a transparent object. Diffuse effects are important when a partially transparent surface, such as frosted glass, is to be modeled. Light passing through such materials is scattered so that a blurred image of background objects is obtained. Diffuse refractions can be generated by decreasing the intensity of the refracted light and spreading intensity contributions at each point on the refracting surface onto a finite area. These manipulations are time-consuming, and most lighting models employ only specular effects.

Realistic transparency effects are modeled by considering light refraction. When light is incident upon a transparent surface, part of it is reflected and part is refracted (Fig. 14-27). Because the speed of light is different in different materials, the path of the refracted light is different from that of the incident light. The direction of the refracted light, specified by the angle of refraction, is a function of the index of refraction of each material and the direction of the incident light. Index of refraction for a material is defined as the ratio of the speed of light in a vacuum to the speed of light in the material. Angle of refraction  $\theta_r$  is calculated from the angle of incidence  $\theta_i$ , the index of refraction  $\eta_i$  of the "incident" material (usually air), and the index of refraction  $\eta_r$  of the refracting material according to Snell's law:

$$\sin \theta_r = \frac{\eta_i}{\eta_r} \sin \theta_i \quad (14-17)$$



Figure 14-26  
A ray traced view of a transparent glass surface, showing both light transmission from objects behind the glass and light reflection from the glass surface.  
(Courtesy of Eric Haines, 3D/YE Inc.)

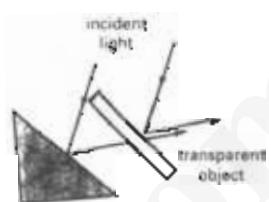


Figure 14-25  
Light emission from a transparent surface is in general a combination of reflected and transmitted light.

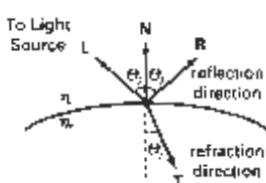


Figure 14-27  
Reflection direction R and refraction direction T for a ray of light incident upon a surface with index of refraction  $\eta_r$ .



Figure 14-28

Refraction of light through a glass object. The emerging refracted ray travels along a path that is parallel to the incident light path (dashed line).

Actually, the index of refraction of a material is a function of the wavelength of the incident light, so that the different color components of a light ray will be refracted at different angles. For most applications, we can use an average index of refraction for the different materials that are modeled in a scene. The index of refraction of air is approximately 1, and that of crown glass is about 1.5. Using these values in Eq. 14-17 with an angle of incidence of 30° yields an angle of refraction of about 19°. Figure 14-28 illustrates the changes in the path direction for a light ray refracted through a glass object. The overall effect of the refraction is to shift the incident light to a parallel path. Since the calculations of the trigonometric functions in Eq. 14-17 are time-consuming, refraction effects could be modeled by simply shifting the path of the incident light a small amount.

From Snell's law and the diagram in Fig. 14-27, we can obtain the unit transmission vector  $\mathbf{T}$  in the refraction direction  $\theta_r$  as

$$\mathbf{T} = \left( \frac{n_i}{n_r} \cos \theta_i - \cos \theta_r \right) \mathbf{N} - \frac{n_r}{n_i} \mathbf{L} \quad (14-18)$$

where  $\mathbf{N}$  is the unit surface normal, and  $\mathbf{L}$  is the unit vector in the direction of the light source. Transmission vector  $\mathbf{T}$  can be used to locate intersections of the refraction path with objects behind the transparent surface. Including refraction effects in a scene can produce highly realistic displays, but the determination of refraction paths and object intersections requires considerable computation. Most scan-line image-space methods model light transmission with approximations that reduce processing time. We return to the topic of refraction in our discussion of ray-tracing algorithms (Section 14-6).

A simpler procedure for modeling transparent objects is to ignore the path shifts altogether. In effect, this approach assumes there is no change in the index of refraction from one material to another, so that the angle of refraction is always the same as the angle of incidence. This method speeds up the calculation of intensities and can produce reasonable transparency effects for thin polygon surfaces.

We can combine the transmitted intensity  $I_{\text{trans}}$  through a surface from a background object with the reflected intensity  $I_{\text{ref}}$  from the transparent surface (Fig. 14-29) using a transparency coefficient  $k_t$ . We assign parameter  $k_t$ , a value between 0 and 1 to specify how much of the background light is to be transmitted. Total surface intensity is then calculated as

$$I = (1 - k_t)I_{\text{ref}} + k_t I_{\text{trans}} \quad (14-19)$$

The term  $(1 - k_t)$  is the opacity factor.

For highly transparent objects, we assign  $k_t$  a value near 1. Nearly opaque objects transmit very little light from background objects, and we can set  $k_t$  to a value near 0 for these materials (opacity near 1). It is also possible to allow  $k_t$  to be a function of position over the surface, so that different parts of an object can transmit more or less background intensity according to the values assigned to  $k_t$ .

Transparency effects are often implemented with modified depth-buffer (z-buffer) algorithms. A simple way to do this is to process opaque objects first to determine depths for the visible opaque surfaces. Then, the depth positions of the transparent objects are compared to the values previously stored in the depth buffer. If any transparent surface is visible, its reflected intensity is calculated and combined with the opaque-surface intensity previously stored in the frame buffer. This method can be modified to produce more accurate displays by using additional storage for the depth and other parameters of the transparent

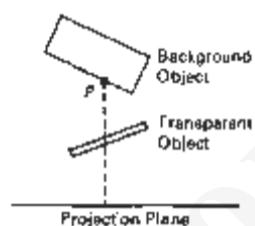
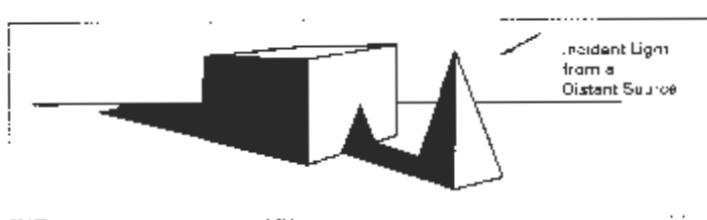


Figure 14-29

The intensity of a background object at point P can be combined with the reflected intensity off the surface of a transparent object along a perpendicular projection line (dashed).



Section 14-3  
Displaying Light Intensities

Figure 14-30  
Objects modeled with shadow regions

surfaces. This allows depth values for the transparent surfaces to be compared to each other, as well as to the depth values of the opaque surfaces. Visible transparent surfaces are then rendered by combining their surface intensities with those of the visible and opaque surfaces behind them.

Accurate displays of transparency and antialiasing can be obtained with the *A*-buffer algorithm. For each pixel position, surface patches for all overlapping surfaces are saved and sorted in depth order. Then, intensities for the transparent and opaque surface patches that overlap in depth are combined in the proper visibility order to produce the final averaged intensity for the pixel, as discussed in Chapter 15.

A depth sorting visibility algorithm can be modified to handle transparency by first sorting surfaces in depth order, then determining whether any visible surface is transparent. If we find a visible transparent surface, its reflected surface intensity is combined with the surface intensity of objects behind it to obtain the pixel intensity at each projected surface point.

### Shadows

Hidden-surface methods can be used to locate areas where light sources produce shadows. By applying a hidden-surface method with a light source at the view position, we can determine which surface sections cannot be "seen" from the light source. These are the shadow areas. Once we have determined the shadow areas for all light sources, the shadows could be treated as surface patterns and stored in pattern arrays. Figure 14-30 illustrates the generation of shading patterns for two objects on a table and a distant light source. All shadow areas in this figure are surfaces that are not visible from the position of the light source. The scene in Fig. 14-26 shows shadow effects produced by multiple light sources.

Shadow patterns generated by a hidden-surface method are valid for any selected viewing position, as long as the light-source positions are not changed. Surfaces that are visible from the view position are shaded according to the lighting model, which can be combined with texture patterns. We can display shadow areas with ambient-light intensity only, or we can combine the ambient light with specified surface textures.

## 14-3 DISPLAYING LIGHT INTENSITIES

Values of intensity calculated by an illumination model must be converted to one of the allowable intensity levels for the particular graphics system in use. Some

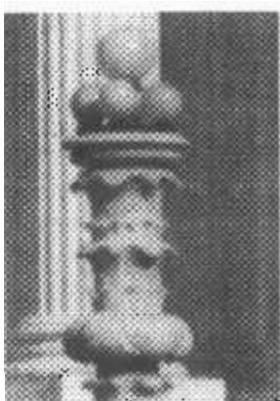


Figure 14-34

An enlarged section of a photograph reproduced with a halftoning method, showing how tones are represented with varying size dots.

## 14-4

### HALFTONE PATTERNS AND DITHERING TECHNIQUES

When an output device has a limited intensity range, we can create an apparent increase in the number of available intensities by incorporating multiple pixel positions into the display of each intensity value. When we view a small region consisting of several pixel positions, our eyes tend to integrate or average the fine detail into an overall intensity. Bilevel monitors and printers, in particular, can take advantage of this visual effect to produce pictures that appear to be displayed with multiple intensity values.

Continuous-tone photographs are reproduced for publication in newspapers, magazines, and books with a printing process called **halftoning**, and the reproduced pictures are called **halftones**. For a black-and-white photograph, each intensity area is reproduced as a series of black circles on a white background. The diameter of each circle is proportional to the darkness required for that intensity region. Darker regions are printed with larger circles, and lighter regions are printed with smaller circles (more white area). Figure 14-34 shows an enlarged section of a gray-scale halftone reproduction. Color halftones are printed using dots of various sizes and colors, as shown in Fig. 14-35. Book and magazine halftones are printed on high-quality paper using approximately 60 to 80 circles of varying diameter per centimeter. Newspapers use lower quality paper and lower resolution (about 25 to 30 dots per centimeter).

#### Halftone Approximations

In computer graphics, halftone reproductions are approximated using rectangular pixel regions, called **halftone patterns** or **pixel patterns**. The number of intensity

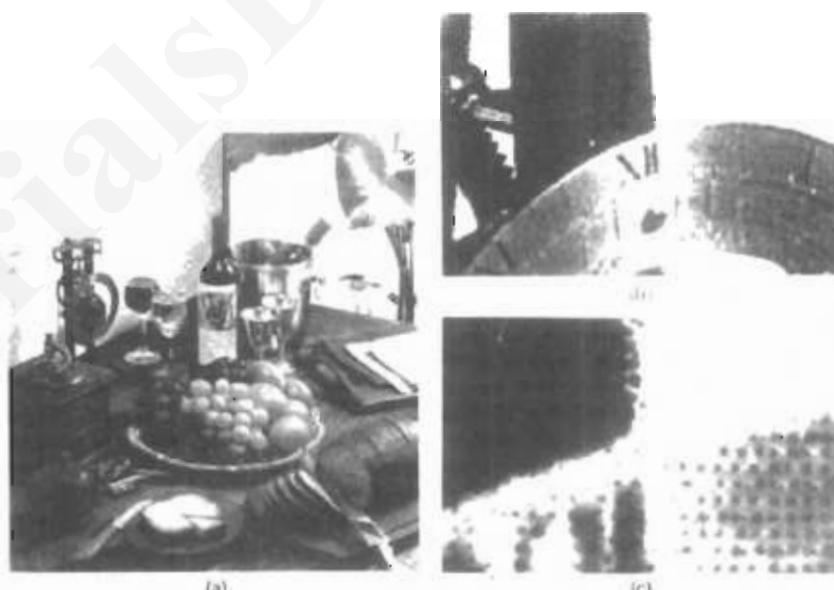
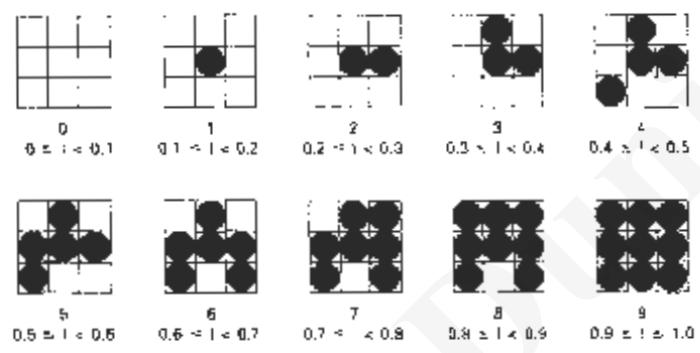


Figure 14-35

Color halftone dot patterns. The top half of the click in the color halftone (a) is enlarged by a factor of 10 in (b) and by a factor of 50 in (c).

**Section 14-4****Halftone Patterns and Dithering Techniques****Figure 14-36**

A 2 by 2 pixel grid used to display five intensity levels on a bilevel system. The intensity values that would be mapped to each grid are listed below each pixel pattern.

**Figure 14-37**

A 3 by 3 pixel grid can be used to display 10 intensities on a bilevel system. The intensity values that would be mapped to each grid are listed below each pixel pattern.

levels that we can display with this method depends on how many pixels we include in the rectangular grids and how many levels a system can display. With  $n$  by  $n$  pixels for each grid on a bilevel system, we can represent  $n^2 + 1$  intensity levels. Figure 14-36 shows one way to set up pixel patterns to represent five intensity levels that could be used with a bilevel system. In pattern 0, all pixels are turned off; in pattern 1, one pixel is turned on, and in pattern 4, all four pixels are turned on. An intensity value  $I$  in a scene is mapped to a particular pattern according to the range listed below each grid shown in the figure. Pattern 0 is used for  $0 \leq I < 0.2$ , pattern 1 for  $0.2 \leq I < 0.4$ , and pattern 4 is used for  $0.8 \leq I \leq 1.0$ .

With 3 by 3 pixel grids on a bilevel system, we can display 10 intensity levels. One way to set up the 10 pixel patterns for these levels is shown in Fig. 14-37. Pixel positions are chosen at each level so that the patterns approximate the increasing circle sizes used in halftone reproductions. That is, the "on" pixel positions are near the center of the grid for lower intensity levels and expand outward as the intensity level increases.

For any pixel-grid size, we can represent the pixel patterns for the various possible intensities with a "mask" of pixel position numbers. As an example, the following mask can be used to generate the nine 3 by 3 grid patterns for intensity levels above 0 shown in Fig. 14-37.

$$\begin{bmatrix} 8 & 3 & 7 \\ 5 & 1 & 2 \\ 4 & 9 & 6 \end{bmatrix}$$

G4.39

To display a particular intensity with level number  $k$ , we turn on each pixel whose position number is less than or equal to  $k$ .

Although the use of  $n$  by  $n$  pixel patterns increases the number of intensities that can be displayed, they reduce the resolution of the displayed picture by a factor of  $1/n$  along each of the  $x$  and  $y$  axes. A 512 by 512 screen area, for instance, is reduced to an area containing 256 by 256 intensity points with 2 by 2 grid patterns. And with 3 by 3 patterns, we would reduce the 512 by 512 area to 128 intensity positions along each side.

Another problem with pixel grids is that subgrid patterns become apparent as the grid size increases. The grid size that can be used without distorting the intensity variations depends on the size of a displayed pixel. Therefore, for systems with lower resolution (fewer pixels per centimeter), we must be satisfied with fewer intensity levels. On the other hand, high-quality displays require at least 64 intensity levels. This means that we need 8 by 8 pixel grids. And to achieve a resolution equivalent to that of halftones in books and magazines, we must display 60 dots per centimeter. Thus, we need to be able to display  $60 \times 8 = 480$  dots per centimeter. Some devices, for example high-quality film recorders, are able to display this resolution.

Pixel-grid patterns for halftone approximations must also be constructed to minimize contouring and other visual effects not present in the original scene. Contouring can be minimized by evolving each successive grid pattern from the previous pattern. That is, we form the pattern at level  $k$  by adding an "on" position to the grid pattern at level  $k - 1$ . Thus, if a pixel position is on for one grid level, it is on for all higher levels (Figs. 14-36 and 14-37). We can minimize the introduction of other visual effects by avoiding symmetrical patterns. With a 3 by 3 pixel grid, for instance, the third intensity level above zero would be better represented by the pattern in Fig. 14-38(a) than by any of the symmetrical arrangements in Fig. 14-38(b). The symmetrical patterns in this figure would produce vertical, horizontal, or diagonal streaks in any large area shaded with intensity level 3. For hard-copy output on devices such as film recorders and some printers, isolated pixels are not effectively reproduced. Therefore, a grid pattern with a single "on" pixel or one with isolated "on" pixels, as in Fig. 14-39, should be avoided.

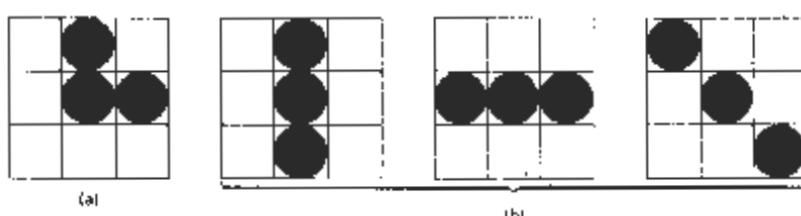
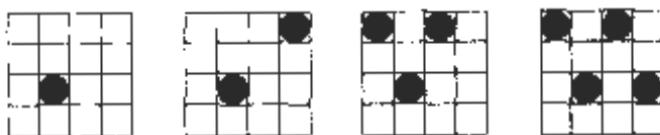


Figure 14-38

For a 3 by 3 pixel grid, pattern (a) is to be preferred to the patterns in (b) for representing the third intensity level above 0.



**Figure 14-39**  
Halftone grid patterns with isolated pixels that cannot be effectively reproduced on some hard-copy devices.

0 0 0 0	0 1 0 0	0 1 1 0	1 1 1 0	1 1 1 1	1 2 1 1
0	1	2	3	4	5
1 2 1 1	2 2 1 1	2 2 2 2	2 3 2 2	2 3 2 1	3 3 2 1
6	7	8	9	10	11
3 3 3 3	3 3 3 3	3 3 3 3	3 3 3 3	3 3 3 3	3 3 3 3
12					

**Figure 14-40**  
Intensity levels 0 through 12 obtained with halftone approximations using 2 by 2 pixel grids on a four-level system.

Halftone approximations also can be used to increase the number of intensity options on systems that are capable of displaying more than two intensities per pixel. For example, on a system that can display four intensity levels per pixel, we can use 2 by 2 pixel grids to extend the available intensity levels from 4 to 16. In Fig. 14-36, the four grid patterns above zero now represent several levels each, since each pixel position can display three intensity values above zero. Figure 14-40 shows one way to assign the pixel intensities to obtain the 16 distinct levels. Intensity levels for individual pixels are labeled 0 through 3, and the overall levels for the system are labeled 0 through 15.

Similarly, we can use pixel-grid patterns to increase the number of intensities that can be displayed on a color system. As an example, suppose we have a three-bit per pixel RGB system. This gives one bit per color gun in the monitor, providing eight colors (including black and white). Using 2 by 2 pixel-grid patterns, we now have 16 phosphor dots that can be used to represent a particular color value, as shown in Fig. 14-41. Each of the three RGB colors has four phosphor dots in the pattern, which allows five possible settings per color. This gives us a total of 125 different color combinations.



**Figure 14-41**  
An RGB 2 by 2 pixel-grid pattern.

### Dithering Techniques

The term **dithering** is used in various contexts. Primarily, it refers to techniques for approximating halftones without reducing resolution, as pixel-grid patterns do. But the term is also applied to halftone-approximation methods using pixel grids, and sometimes it is used to refer to color halftone approximations only.

Random values added to pixel intensities to break up contours are often referred to as **dither noise**. Various algorithms have been used to generate the ran-

dither distributions. The effect is to add noise over an entire picture, which tends to soften intensity boundaries.

*Ordered-dither methods* generate intensity variations with a one-to-one mapping of points in a scene to the display pixels. To obtain  $n^2$  intensity levels, we set up an  $n$  by  $n$  dither matrix  $D_n$ , whose elements are distinct positive integers in the range 0 to  $n^2 - 1$ . For example, we can generate four intensity levels with

$$D_2 = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix} \quad (14-31)$$

and we can generate nine intensity levels with

$$D_3 = \begin{bmatrix} 7 & 2 & 6 \\ 4 & 0 & 1 \\ 3 & 8 & 5 \end{bmatrix} \quad (14-32)$$

The matrix elements for  $D_2$  and  $D_3$  are in the same order as the pixel mask for setting up 2 by 2 and 3 by 3 pixel grids, respectively. For a bilevel system, we then determine display intensity values by comparing input intensities to the matrix elements. Each input intensity is first scaled to the range  $0 \leq I \leq n^2$ . If the intensity  $I$  is to be applied to screen position  $(x, y)$ , we calculate row and column numbers for the dither matrix as

$$i = (x \bmod n) + 1, \quad j = (y \bmod n) + 1 \quad (14-33)$$

If  $I > D_n(i, j)$ , we turn on the pixel at position  $(x, y)$ . Otherwise, the pixel is not turned on.

Elements of the dither matrix are assigned in accordance with the guidelines discussed for pixel grids. That is, we want to minimize added visual effect in a displayed scene. Order dither produces constant-intensity areas identical to those generated with pixel-grid patterns when the values of the matrix elements correspond to the grid mask. Variations from the pixel-grid displays occur at boundaries of the intensity levels.

Typically, the number of intensity levels is taken to be a multiple of 2. Higher-order dither matrices are then obtained from lower-order matrices with the recurrence relation:

$$D_n = \begin{bmatrix} 4D_{n/2} + D_2(1,1)U_{n/2} & 4D_{n/2} + D_2(1,2)U_{n/2} \\ 4D_{n/2} + D_2(2,1)U_{n/2} & 4D_{n/2} + D_2(2,2)U_{n/2} \end{bmatrix} \quad (14-34)$$

assuming  $n \geq 4$ . Parameter  $U_{n/2}$  is the "unity" matrix (all elements are 1). As an example, if  $D_2$  is specified as in Eq. 14-31, then recurrence relation 14-34 yields

$$D_4 = \begin{bmatrix} 15 & 7 & 13 & 5 \\ 3 & 11 & 1 & 9 \\ 12 & 4 & 14 & 6 \\ 0 & 8 & 2 & 10 \end{bmatrix} \quad (14-35)$$

Another method for mapping a picture with  $m$  by  $n$  points to a display area with  $m$  by  $n$  pixels is *error diffusion*. Here, the error between an input intensity

**Section 14-4****Halftone Patterns and Dithering Techniques**

value and the displayed pixel intensity level at a given position is dispersed, or diffused, to pixel positions to the right and below the current pixel position. Starting with a matrix  $M$  of intensity values obtained by scanning a photograph, we want to construct an array  $I$  of pixel intensity values for an area of the screen. We do this by first scanning across the rows of  $M$ , from left to right, top to bottom, and determining the nearest available pixel-intensity level for each element of  $M$ . Then the error between the value stored in matrix  $M$  and the displayed intensity level at each pixel position is distributed to neighboring elements in  $M$ , using the following simplified algorithm.

```

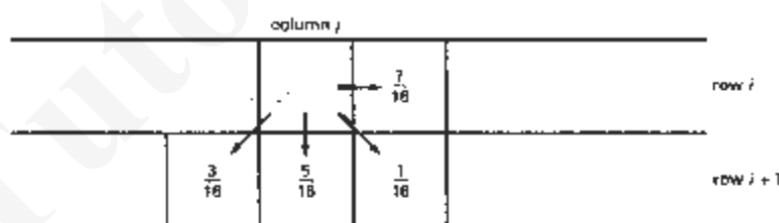
for (i=0; i<m; i++)
    for (j=0; j<n; j++) {
        /* Determine the available intensity level  $I_{ij}$  */
        /* that is closest to the value  $M_{ij}$  */
         $I_{ij} := I_0$ ;
        err :=  $M_{ij} - I_{ij}$ ;
         $M_{ij+1} := M_{ij+1} + \alpha \cdot err$ ;
         $M_{ij+1,j-1} := M_{ij+1,j-1} + \beta \cdot err$ ;
         $M_{ij,j-1} := M_{ij,j-1} + \gamma \cdot err$ ;
         $M_{ij+1,j+1} := M_{ij+1,j+1} + \delta \cdot err$ ;
    }
}

```

Once the elements of matrix  $I$  have been assigned intensity-level values, we then map the matrix to some area of a display device, such as a printer or video monitor. Of course, we cannot disperse the error past the last matrix column ( $j = n$ ) or below the last matrix row ( $i = m$ ). For a bilevel system, the available intensity levels are 0 and 1. Parameters for distributing the error can be chosen to satisfy the following relationship

$$\alpha + \beta + \gamma + \delta \leq 1 \quad (14.36)$$

One choice for the error-diffusion parameters that produces fairly good results is  $(\alpha, \beta, \gamma, \delta) = (7/16, 3/16, 5/16, 1/16)$ . Figure 14-42 illustrates the error distribution using these parameter values. Error diffusion sometimes produces "ghosts" in a picture by repeating, or echoing, certain parts of the picture, particularly with facial features such as hairlines and nose outlines. Ghosting can be re-



**Figure 14-42**  
Fraction of intensity error that can be distributed to neighboring pixel positions using an error-diffusion scheme.

Chapter 14  
Illumination Models and Surface-  
Rendering Methods

34	48	40	32	29	15	23	31
42	68	56	52	21	5	7	10
50	62	61	45	13	1	2	18
38	46	54	37	25	7	9	26
28	14	22	30	35	48	41	33
20	4	6	11	43	59	67	62
12	0	3	19	51	63	50	44
24	16	8	27	29	47	55	36

Figure 14-43  
One possible distribution scheme  
for dividing the intensity array into  
64 dot-diffusion classes, numbered  
from 0 through 63.

duced by choosing values for the error-diffusion parameters that sum to a value less than 1 and by rescaling the matrix values after the dispersion of errors. One way to rescale is to multiply all elements of  $M$  by 0.8 and then add 0.1. Another method for improving picture quality is to alternate the scanning of matrix rows from right to left and left to right.

A variation on the error-diffusion method is *dot diffusion*. In this method, the  $m$  by  $n$  array of intensity values is divided into 64 classes numbered from 0 to 63, as shown in Fig. 14-43. The error between a matrix value and the displayed intensity is then distributed only to those neighboring matrix elements that have a larger class number. Distribution of the 64 class numbers is based on minimizing the number of elements that are completely surrounded by elements with a lower class number, since this would tend to direct all errors of the surrounding elements to that one position.

## 14-5

### POLYGON-RENDERING METHODS

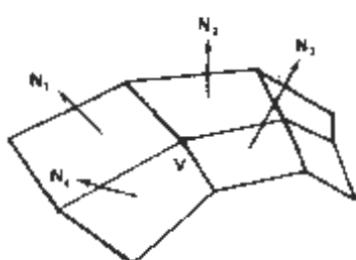
In this section, we consider the application of an illumination model to the rendering of standard graphics objects: those formed with polygon surfaces. The objects are usually polygon-mesh approximations of curved-surface objects, but they may also be polyhedra that are not curved-surface approximations. Scanline 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 interpolation scheme.

#### Constant-Intensity Shading

A fast and simple method for rendering an object with polygon surfaces is **constant-intensity shading**, also called **flat shading**. In this method, a single intensity is calculated for each polygon. All points over the surface of the polygon are then displayed with the same intensity value. Constant shading can be useful for quickly displaying the general appearance of a curved surface, as in Fig. 14-47.

In general, flat shading of polygon facets provides an accurate rendering for an object if all of the following assumptions are valid:

- The object is a polyhedron and is not an approximation of an object with a curved surface.



**Figure 14-44**  
 The normal vector at vertex  $V$  is calculated as the average of the surface normals for each polygon sharing that vertex.

- All light sources illuminating the object are sufficiently far from the surface so that  $\mathbf{N} \cdot \mathbf{L}$  and the attenuation function are constant over the surface.
- The viewing position is sufficiently far from the surface so that  $\mathbf{V} \cdot \mathbf{R}$  is constant over the surface.

Even if all of these conditions are not true, we can still reasonably approximate surface-lighting effects using small polygon facets with flat shading and calculate the intensity for each facet, say, at the center of the polygon.

#### Gouraud Shading

This intensity-interpolation scheme, developed by Gouraud and generally referred to as **Gouraud shading**, renders a polygon surface by linearly interpolating intensity values across the surface. Intensity values for each polygon are matched with the values of adjacent polygons along the common edges, thus eliminating the intensity discontinuities that can occur in flat shading.

Each polygon surface is rendered with Gouraud shading by performing the following calculations:

- Determine the average unit normal vector at each polygon vertex.
- Apply an illumination model to each vertex to calculate the vertex intensity.
- Linearly interpolate the vertex intensities over the surface of the polygon.

At each polygon vertex, we obtain a normal vector by averaging the surface normals of all polygons sharing that vertex, as illustrated in Fig. 14-44. Thus, for any vertex position  $V$ , we obtain the unit vertex normal with the calculation

$$\mathbf{N}_v = \frac{\sum_{k=1}^n \mathbf{N}_k}{\left| \sum_{k=1}^n \mathbf{N}_k \right|} \quad (14-37)$$

Once we have the vertex normals, we can determine the intensity at the vertices from a lighting model.

Figure 14-45 demonstrates the next step: interpolating intensities along the polygon edges. For each scan line, the intensity at the intersection of the scan line with a polygon edge is linearly interpolated from the intensities at the edge endpoints. For the example in Fig. 14-45, the polygon edge with endpoint vertices at positions 1 and 2 is intersected by the scan line at point 4. A fast method for obtaining the intensity at point 4 is to interpolate between intensities  $I_1$  and  $I_2$  using only the vertical displacement of the scan line:

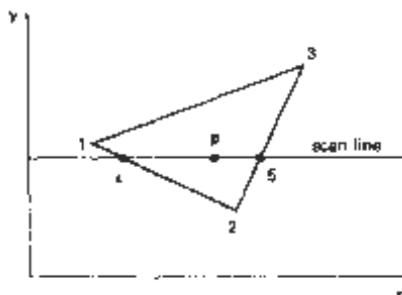


Figure 14-45

For Gouraud shading, the intensity at point 4 is linearly interpolated from the intensities at vertices 3 and 2. The intensity at point 5 is linearly interpolated from intensities at vertices 2 and 3. An interior point  $p$  is then assigned an intensity value that is linearly interpolated from intensities at positions 4 and 5.

$$I_4 = \frac{y_4 - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y_4}{y_1 - y_2} I_2 \quad (14-38)$$

Similarly, intensity at the right intersection of this scan line (point 5) is interpolated from intensity values at vertices 2 and 3. Once these bounding intensities are established for a scan line, an interior point (such as point  $p$  in Fig. 14-45) is interpolated from the bounding intensities at points 4 and 5 as

$$I_p = \frac{x_5 - x_4}{x_5 - x_4} I_4 + \frac{x_4 - x_5}{x_5 - x_4} I_5 \quad (14-39)$$

Incremental calculations are used to obtain successive edge intensity values between scan lines and to obtain successive intensities along a scan line. As shown in Fig. 14-46, if the intensity at edge position  $(x, y)$  is interpolated as

$$I = \frac{y - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y}{y_1 - y_2} I_2 \quad (14-40)$$

then we can obtain the intensity along this edge for the next scan line,  $y = 1$ , as

$$I' = I + \frac{I_2 - I_1}{y_1 - y_2} \quad (14-41)$$

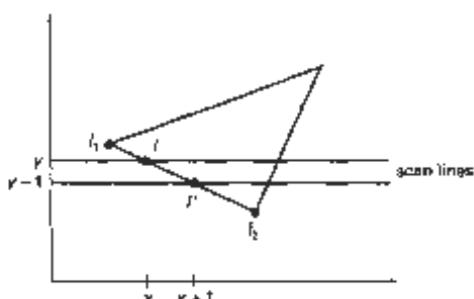


Figure 14-46

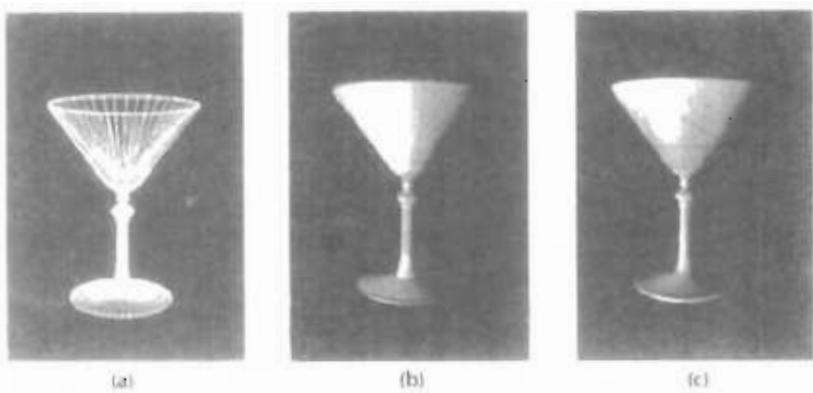
Incremental interpolation of intensity values along a polygon edge for successive scan lines.

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>



**Figure 14-47**  
A polygon mesh approximation of an object (a) is rendered with flat shading (b) and with Gouraud shading (c).

Similar calculations are used to obtain intensities at successive horizontal pixel positions along each scan line.

When surfaces are to be rendered in color, the intensity of each color component is calculated at the vertices. Gouraud shading can be combined with a hidden-surface algorithm to fill in the visible polygons along each scan line. An example of an object shaded with the Gouraud method appears in Fig. 14-47.

Gouraud shading removes the intensity discontinuities associated with the constant-shading model, but it has some other deficiencies. Highlights on the surface are sometimes displayed with anomalous shapes, and the linear intensity interpolation can cause bright or dark intensity streaks, called Mach bands, to appear on the surface. These effects can be reduced by dividing the surface into a greater number of polygon faces or by using other methods, such as Phong shading, that require more calculations.

#### Phong Shading

A more accurate method for rendering a polygon surface is to interpolate normal vectors, and then apply the illumination model to each surface point. This method, developed by Phong Bui Tuong, is called **Phong shading**, or **normal-vector interpolation shading**. It displays more realistic highlights on a surface and greatly reduces the Mach band effect.

A polygon surface is rendered using Phong shading by carrying out the following steps:

- Determine the average unit normal vector at each polygon vertex.
- Linearly interpolate the vertex normals over the surface of the polygon.
- Apply an illumination model along each scan line to calculate projected pixel intensities for the surface points.

Interpolation of surface normals along a polygon edge between two vertices is illustrated in Fig. 14-48. The normal vector  $\mathbf{N}$  for the scan-line intersection point along the edge between vertices 1 and 2 can be obtained by vertically interpolating between edge endpoint normals:

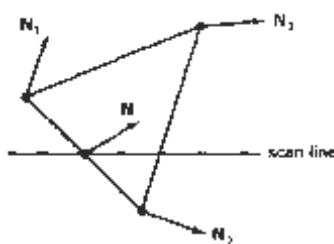


Figure 14-48  
Interpolation of surface normals  
along a polygon edge

$$\mathbf{N} = \frac{y - y_2}{y_1 - y_2} \mathbf{N}_1 + \frac{y_1 - y}{y_1 - y_2} \mathbf{N}_2 \quad (14-42)$$

Incremental methods are used to evaluate normals between scan lines and along each individual scan line. At each pixel position along a scan line, the illumination model is applied to determine the surface intensity at that point.

Intensity calculations using an approximated normal vector at each point along the scan line produce more accurate results than the direct interpolation of intensities, as in Gouraud shading. The trade-off, however, is that Phong shading requires considerably more calculations.

#### Fast Phong Shading

Surface rendering with Phong shading can be speeded up by using approximations in the illumination-model calculations of normal vectors. Fast Phong shading approximates the intensity calculations using a Taylor-series expansion and triangular surface patches.

Since Phong shading interpolates normal vectors from vertex normals, we can express the surface normal  $\mathbf{N}$  at any point  $(x, y)$  over a triangle as

$$\mathbf{N} = \mathbf{Ax} + \mathbf{By} + \mathbf{C} \quad (14-43)$$

where vectors  $\mathbf{A}$ ,  $\mathbf{B}$ , and  $\mathbf{C}$  are determined from the three vertex equations:

$$\mathbf{N}_k = \mathbf{Ax}_k + \mathbf{By}_k + \mathbf{C}, \quad k = 1, 2, 3 \quad (14-44)$$

with  $(x_k, y_k)$  denoting a vertex position.

Omitting the reflectivity and attenuation parameters, we can write the calculation for light-source diffuse reflection from a surface point  $(x, y)$  as

$$\begin{aligned} I_{\text{diff}}(x, y) &= \frac{\mathbf{L} \cdot \mathbf{N}}{|\mathbf{L}| |\mathbf{N}|} \\ &= \frac{\mathbf{L} \cdot (\mathbf{Ax} + \mathbf{By} + \mathbf{C})}{|\mathbf{L}| |\mathbf{Ax} + \mathbf{By} + \mathbf{C}|} \\ &= \frac{(\mathbf{L} \cdot \mathbf{A})x + (\mathbf{L} \cdot \mathbf{B})y + \mathbf{L} \cdot \mathbf{C}}{|\mathbf{L}| |\mathbf{Ax} + \mathbf{By} + \mathbf{C}|} \end{aligned} \quad (14-45)$$

We can rewrite this expression in the form

#### Section 14-6

##### Ray Tracing Methods

$$I_{\text{diff}}(x, y) = \frac{ax + by + c}{(dx^2 + exy + fy^2 + gx + hy + i)^{1/2}} \quad (14-46)$$

where parameters such as  $a$ ,  $b$ ,  $c$ , and  $i$  are used to represent the various dot products. For example,

$$a = \frac{\mathbf{L} \cdot \mathbf{A}}{|\mathbf{L}|} \quad (14-47)$$

Finally, we can express the denominator in Eq. 14-46 as a Taylor-series expansion and retain terms up to second degree in  $x$  and  $y$ . This yields

$$I_{\text{diff}}(x, y) = T_0x^2 + T_1xy + T_2y^2 - T_3x - T_4y + T_5 \quad (14-48)$$

where each  $T_i$  is a function of parameters  $a$ ,  $b$ ,  $c$ , and so forth.

Using forward differences, we can evaluate Eq. 14-48 with only two additions for each pixel position  $(x, y)$  once the initial forward-difference parameters have been evaluated. Although fast Phong shading reduces the Phong-shading calculations, it still takes approximately twice as long to render a surface with fast Phong shading as it does with Gouraud shading. Normal Phong shading using forward differences takes about six to seven times longer than Gouraud shading.

Fast Phong shading for diffuse reflection can be extended to include specular reflections. Calculations similar to those for diffuse reflections are used to evaluate specular terms such as  $(\mathbf{N} \cdot \mathbf{H})^n$  in the basic illumination model. In addition, we can generalize the algorithm to include polygons other than triangles and finite viewing positions.

## 14-6

### RAY-TRACING METHODS

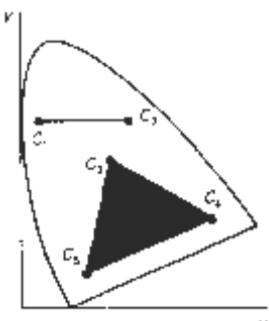
In Section 10-15, we introduced the notion of *ray casting*, where a ray is sent out from each pixel position to iterate surface intersections for object modeling using constructive solid geometry methods. We also discussed the use of ray casting as a method for determining visible surfaces in a scene (Section 13-10). **Ray tracing** is an extension of this basic idea. Instead of merely looking for the visible surface for each pixel, we continue to bounce the ray around the scene, as illustrated in Fig. 14-49, collecting intensity contributions. This provides a simple and powerful rendering technique for obtaining global reflection and transmission effects. The basic ray-tracing algorithm also provides for visible-surface detection, shadow effects, transparency, and multiple light-source illumination. Many extensions to the basic algorithm have been developed to produce photorealistic displays. Ray-traced displays can be highly realistic, particularly for shiny objects, but they require considerable computation time to generate. An example of the global reflection and transmission effects possible with ray tracing is shown in Fig. 14-50.

---

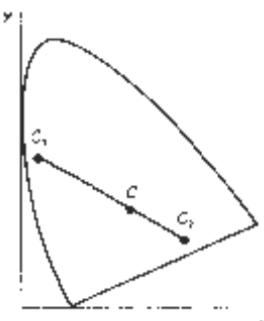
EXERCISES

- 14.1 Write a routine to implement Eq. 14-4 of the basic illumination model using a single point light source and constant surface shading for the faces of a specified polyhedron. The object description is to be given as a set of polygon tables, including surface normals for each of the polygon faces. Additional input parameters include the ambient intensity, light-source intensity, and the surface reflection coefficients. All coordinate information can be specified directly in the viewing reference frame.
- 14.2 Modify the routine in Exercise 14-1 to render a polygon surface mesh using Gouraud shading.
- 14.3 Modify the routine in Exercise 14-1 to render a polygon surface mesh using Phong shading.
- 14.4 Write a routine to implement Eq. 14-9 of the basic illumination model using a single point light source and Gouraud surface shading for the faces of a specified polygon mesh. The object description is to be given as a set of polygon tables, including surface normals for each of the polygon faces. Additional input includes values for the ambient intensity, light-source intensity, surface reflection coefficients, and the specular-reflection parameter. All coordinate information can be specified directly in the viewing reference frame.
- 14.5 Modify the routine in Exercise 14-4 to render the polygon surfaces using Phong shading.
- 14.6 Modify the routine in Exercise 14-4 to include a linear intensity attenuation function.
- 14.7 Modify the routine in Exercise 14-4 to render the polygon surfaces using Phong shading and a linear intensity attenuation function.
- 14.8 Modify the routine in Exercise 14-4 to implement Eq. 14-13 with any specified number of polyhedrons and light sources in the scene.
- 14.9 Modify the routine in Exercise 14-4 to implement Eq. 14-14 with any specified number of polyhedrons and light sources in the scene.
- 14.10 Modify the routine in Exercise 14-4 to implement Eq. 14-15 with any specified number of polyhedrons and light sources in the scene.
- 14.11 Modify the routine in Exercise 14-4 to implement Eqs. 14-17 and 14-19 with any specified number of light sources and polyhedrons (either opaque or transparent) in the scene.
- 14.12 Discuss the differences you might expect to see in the appearance of specular reflections modeled with  $\mathbf{IN} \cdot \mathbf{HP}$  compared to specular reflections modeled with  $(\mathbf{V} \cdot \mathbf{R})^n$ .
- 14.13 Verify that  $2\alpha = \phi$  in Fig. 14-18 when all vectors are coplanar, but that in general  $2\alpha \neq \phi$ .
- 14.14 Discuss how the different visible-surface detection methods can be combined with an intensity model for displaying a set of polyhedrons with opaque surfaces.
- 14.15 Discuss how the various visible surface detection methods can be modified to process transparent objects. Are there any visible surface detection methods that cannot handle transparent surfaces?
- 14.16 Set up an algorithm, based on one of the visible-surface detection methods, that will identify shadow areas in a scene illuminated by a distant point source.
- 14.17 How many intensity levels can be displayed with halftone approximations using a  $n$  by  $n$  pixel grids where each pixel can be displayed with  $m$  different intensities?
- 14.18 How many different color combinations can be generated using halftone approximations on a two-level RGB system with a 3 by 3 pixel grid?
- 14.19 Write a routine to display a given set of surface intensity variations using halftone approximations with 3 by 3 pixel grids and two intensity levels (0 and 1) per pixel.
- 14.20 Write a routine to generate ordered dither matrices using the recurrence relation in Eq. 14-34.

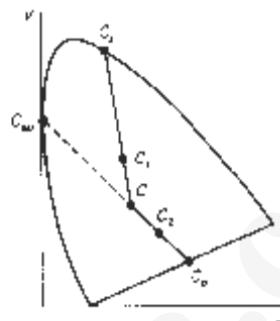
- 14-21. Write a procedure to display a given array of intensity values using the ordered-dither method.
- 14-22. Write a procedure to implement the error-diffusion algorithm for a given  $m$  by  $n$  array of intensity values.
- 14-23. Write a program to implement the basic ray-tracing algorithm for a scene containing a single sphere hovering over a checkerboard ground square. The scene is to be illuminated with a single point light source at the viewing position.
- 14-24. Write a program to implement the basic ray-tracing algorithm for a scene containing any specified arrangement of spheres and polygon surfaces illuminated by a given set of point light sources.
- 14-25. Write a program to implement the basic ray-tracing algorithm using space-subdivision methods for any specified arrangement of spheres and polygon surfaces illuminated by a given set of point light sources.
- 14-26. Write a program to implement the following features of distributed ray tracing: pixel sampling with 16 jittered rays per pixel, distributed reflection directions, distributed refraction directions, and extended light sources.
- 14-27. Set up an algorithm for modeling the motion blur of a moving object using distributed ray tracing.
- 14-28. Implement the basic radiosity algorithm for rendering the inside surfaces of a cube when one inside face of the cube is a light source.
- 14-29. Devise an algorithm for implementing the progressive refinement radiosity method.
- 14-30. Write a routine to transform an environment map to the surface of a sphere.
- 14-31. Write a program to implement texture mapping for (a) spherical surfaces and (b) polyhedrons.
- 14-32. Given a spherical surface, write a bump-mapping procedure to generate the bumpy surface of an orange.
- 14-33. Write a bump-mapping routine to produce surface-normal variations for any specified bump function.



**Figure 15-8**  
Color gamuts defined on the chromaticity diagram for a two-color and a three-color system of primaries.



**Figure 15-9**  
Representing complementary colors on the chromaticity diagram



**Figure 15-10**  
Determining dominant wavelength and purity with the chromaticity diagram.

C<sub>v</sub>. Color C<sub>1</sub> can then be represented as a combination of white light C and the spectral color C<sub>v</sub>. Thus, the dominant wavelength of C<sub>1</sub> is C<sub>v</sub>. This method for determining dominant wavelength will not work for color points that are between C and the purple line. Drawing a line from C through point C<sub>2</sub> in Fig. 15-10 takes us to point C<sub>p</sub> on the purple line, which is not in the visible spectrum. Point C<sub>2</sub> is referred to as a nonspectral color, and its dominant wavelength is taken as the complement of C<sub>v</sub> that lies on the spectral curve (point C<sub>p</sub>). Nonspectral colors are in the purple-magenta range and have spectral distributions with subtractive dominant wavelengths. They are generated by subtracting the spectral dominant wavelength (such as C<sub>p</sub>) from white light.

For any color point, such as C<sub>1</sub> in Fig. 15-10, we determine the purity as the relative distance of C<sub>1</sub> from C along the straight line joining C to C<sub>2</sub>. If d<sub>12</sub> denotes the distance from C to C<sub>1</sub> and d<sub>1v</sub> is the distance from C to C<sub>v</sub>, we calculate purity as the ratio  $d_{12}/d_{1v}$ . Color C<sub>1</sub> in this figure is about 25 percent pure, since it is situated at about one-fourth the total distance from C to C<sub>v</sub>. At position C<sub>p</sub>, the color point would be 100 percent pure.

### 15-3

#### INTUITIVE COLOR CONCEPTS

An artist creates a color painting by mixing color pigments with white and black pigments to form the various shades, tints, and tones in the scene. Starting with the pigment for a "pure color" (or "pure hue"), the artist adds a black pigment to produce different shades of that color. The more black pigment, the darker the shade. Similarly, different tints of the color are obtained by adding a white pigment to the original color, making it lighter as more white is added. Tones of the color are produced by adding both black and white pigments.

To many, these color concepts are more intuitive than describing a color as a set of three numbers that give the relative proportions of the primary colors. It is generally much easier to think of making a color lighter by adding white and making a color darker by adding black. Therefore, graphics packages providing

color palettes to a user often employ two or more color models. One model provides an intuitive color interface for the user, and others describe the color components for the output devices.

## 15.4

### RGB COLOR MODEL

Based on the *tristimulus theory* of vision, our eyes perceive color through the stimulation of three visual pigments in the cones of the retina. These visual pigments have a peak sensitivity at wavelengths of about 630 nm (red), 530 nm (green), and 450 nm (blue). By comparing intensities in a light source, we perceive the color of the light. This theory of vision is the basis for displaying color output on a video monitor using the three color primaries, red, green, and blue, referred to as the RGB color model.

We can represent this model with the unit cube defined on R, G, and B axes, as shown in Fig. 15-11. The origin represents black, and the vertex with coordinates (1, 1, 1) is white. Vertices of the cube on the axes represent the primary colors, and the remaining vertices represent the complementary color for each of the primary colors.

As with the XYZ color system, the RGB color scheme is an additive model. Intensities of the primary colors are added to produce other colors. Each color point within the bounds of the cube can be represented as the triple (R, G, B), where values for R, G, and B are assigned in the range from 0 to 1. Thus, a color  $C_1$  is expressed in RGB components as

$$C_1 = RR + GG + BB \quad (15-5)$$

The magenta vertex is obtained by adding red and blue to produce the triple (1, 0, 1), and white at (1, 1, 1) is the sum of the red, green, and blue vertices. Shades of gray are represented along the main diagonal of the cube from the origin (black) to the white vertex. Each point along this diagonal has an equal contribution from each primary color, so that a gray shade halfway between black and

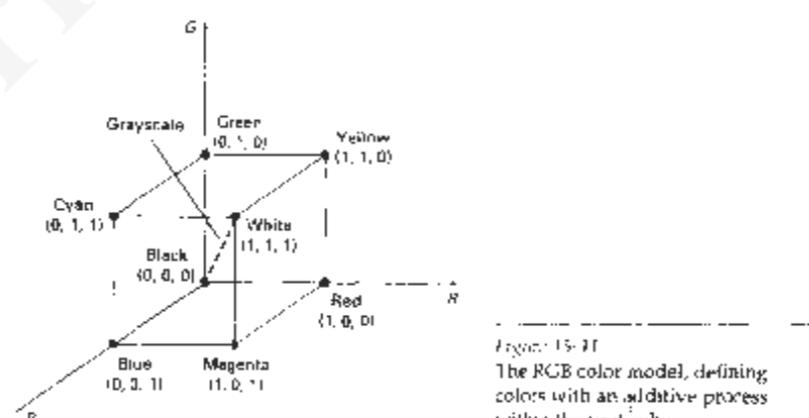
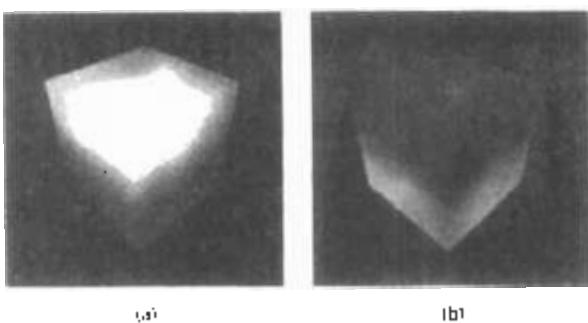


Figure 15-11  
The RGB color model, defining colors with an additive process within the unit cube

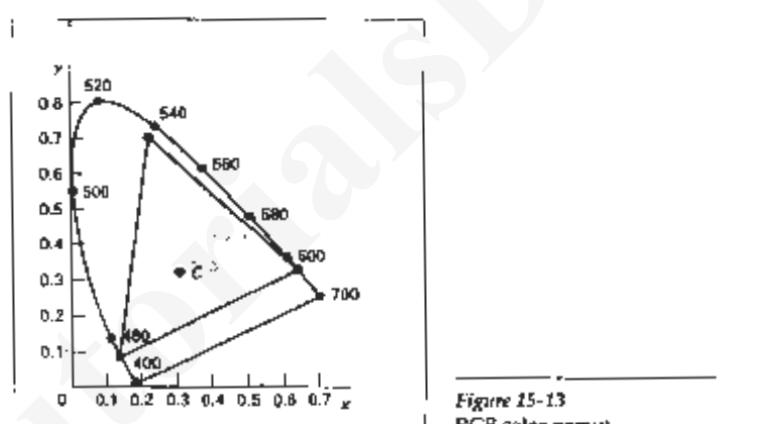


Section 15-4  
RGB Color Model

**Figure 15-12**  
Two views of the RGB color cube: (a) along the grayscale diagonal from white to black and (b) along the grayscale diagonal from black to white.

**TABLE 15-1**  
RGB ( $x$ ,  $y$ ) CHROMATICITY COORDINATES

	NTSC Standard	CIE Model	Approx. Color Monitor Values
R	(0.670, 0.330)	(0.735, 0.265)	(0.628, 0.346)
G	(0.210, 0.710)	(0.274, 0.717)	(0.268, 0.588)
B	(0.140, 0.080)	(0.167, 0.009)	(0.150, 0.070)



**Figure 15-13**  
RGB color gamut.

white is represented as (0.5, 0.5, 0.5). The color gradations along the front and top planes of the RGB cube are illustrated in Fig. 15-12.

Chromaticity coordinates for an NTSC standard RGB phosphor are listed in Table 15-1. Also listed are the RGB chromaticity coordinates for the CIE RGB color model and the approximate values used for phosphors in color monitors. Figure 15-13 shows the color gamut for the NTSC standard RGB primaries.

**15-5****YIQ COLOR MODEL**

Whereas an RGB monitor requires separate signals for the red, green, and blue components of an image, a television monitor uses a single composite signal. The National Television System Committee (NTSC) color model for forming the composite video signal is the YIQ model, which is based on concepts in the CIE XYZ model.

In the YIQ color model, parameter *Y* is the same as in the XYZ model. Luminance (brightness) information is contained in the *Y* parameter, while chromaticity information (hue and purity) is incorporated into the *I* and *Q* parameters. A combination of red, green, and blue intensities are chosen for the *Y* parameter to yield the standard luminosity curve. Since *Y* contains the luminance information, black-and-white television monitors use only the *Y* signal. The largest bandwidth in the NTSC video signal (about 4 MHz) is assigned to the *Y* information. Parameter *I* contains orange-cyan hue information that provides the flesh-tone shading, and occupies a bandwidth of approximately 1.5 MHz. Parameter *Q* carries green-magenta hue information in a bandwidth of about 0.6 MHz.

An RGB signal can be converted to a television signal using an NTSC encoder, which converts RGB values to YIQ values, then modulates and superimposes the *I* and *Q* information on the *Y* signal. The conversion from RGB values to YIQ values is accomplished with the transformation

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.296 & 0.587 & 0.144 \\ 0.596 & 0.275 & -0.321 \\ 0.212 & 0.528 & 0.311 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (15-6)$$

This transformation is based on the NTSC standard RGB phosphor, whose chromaticity coordinates were given in the preceding section. The larger proportions of red and green assigned to parameter *Y* indicate the relative importance of these hues in determining brightness, compared to blue.

An NTSC video signal can be converted to an RGB signal using an NTSC decoder, which separates the video signal into the YIQ components, then converts to RGB values. We convert from YIQ space to RGB space with the inverse matrix transformation from Eq. 15-6:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.956 & 0.620 \\ 1.000 & 0.272 & -0.647 \\ 1.000 & -1.108 & 1.705 \end{bmatrix} \cdot \begin{bmatrix} Y \\ I \\ Q \end{bmatrix} \quad (15-7)$$

**15-6****CMY COLOR MODEL**

A color model defined with the primary colors cyan, magenta, and yellow (CMY) is useful for describing color output to hard-copy devices. Unlike video monitors, which produce a color pattern by combining light from the screen phosphors,

hard-copy devices such as plotters produce a color picture by coating a paper with color pigments. We see the colors by reflected light, a subtractive process.

As we have noted, cyan can be formed by adding green and blue light. Therefore, when white light is reflected from cyan-colored ink, the reflected light must have no red component. That is, red light is absorbed, or subtracted, by the ink. Similarly, magenta ink subtracts the green component from incident light, and yellow subtracts the blue component. A unit cube representation for the CMY model is illustrated in Fig. 15-14.

In the CMY model, point  $(1, 1, 1)$  represents black, because all components of the incident light are subtracted. The origin represents white light. Equal amounts of each of the primary colors produce grays, along the main diagonal of the cube. A combination of cyan and magenta ink produces blue light, because the red and green components of the incident light are absorbed. Other color combinations are obtained by a similar subtractive process.

The printing process often used with the CMY model generates a color point with a collection of four ink dots, somewhat as an RGB monitor uses a collection of three phosphor dots. One dot is used for each of the primary colors (cyan, magenta, and yellow), and one dot is black. A black dot is included because the combination of cyan, magenta, and yellow inks typically produce dark gray instead of black. Some plotters produce different color combinations by spraying the ink for the three primary colors over each other and allowing them to mix before they dry.

We can express the conversion from an RGB representation to a CMY representation with the matrix transformation

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (15-8)$$

where the white is represented in the RGB system as the unit column vector. Similarly, we convert from a CMY color representation to an RGB representation with the matrix transformation

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix} \quad (15-9)$$

where black is represented in the CMY system as the unit column vector.

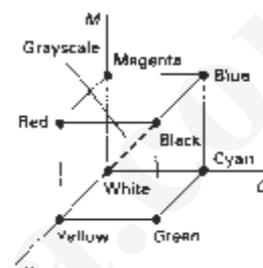
## 15-7

### HSV COLOR MODEL

Instead of a set of color primaries, the HSV model uses color descriptions that have a more intuitive appeal to a user. To give a color specification, a user selects a spectral color and the amounts of white and black that are to be added to obtain different shades, tints, and tones. Color parameters in this model are hue (*H*), saturation (*S*), and value (*V*).

### Section 15-7

#### HSV Color Model



**Figure 15-14**  
The CMY color model, defining colors with a subtractive process inside a unit cube.

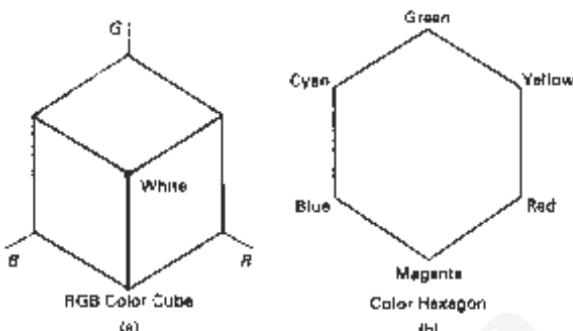


Figure 15-15

When the RGB color cube (a) is viewed along the diagonal from white to black, the color-cube outline is a hexagon (b).

The three-dimensional representation of the HSV model is derived from the RGB cube. If we imagine viewing the cube along the diagonal from the white vertex to the origin (black), we see an outline of the cube that has the hexagon shape shown in Fig. 15-15. The boundary of the hexagon represents the various hues, and it is used as the top of the HSV hexcone (Fig. 15-16). In the hexcone, saturation is measured along a horizontal axis, and value is along a vertical axis through the center of the hexcone.

Hue is represented as an angle about the vertical axis, ranging from  $0^\circ$  at red through  $360^\circ$ . Vertices of the hexagon are separated by  $60^\circ$  intervals. Yellow is at  $60^\circ$ , green at  $120^\circ$ , and cyan opposite red at  $H = 180^\circ$ . Complementary colors are  $180^\circ$  apart.

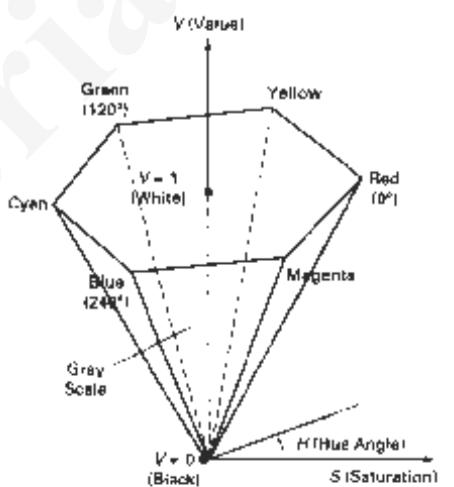


Figure 15-16  
The HSV hexcone.

## Section 15-7

## HSV Color Models

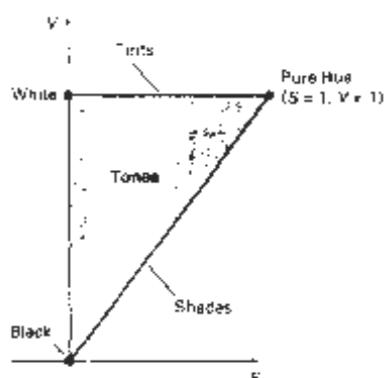


Figure 15-17  
Cross section of the HSV hexcone, showing regions for shades, tints, and tones.

Saturation  $S$  varies from 0 to 1. It is represented in this model as the ratio of the purity of a selected hue to its maximum purity at  $S = 1$ . A selected hue is said to be one-quarter pure at the value  $S = 0.25$ . At  $S = 0$ , we have the gray scale.

Value  $V$  varies from 0 at the apex of the hexcone to 1 at the top. The apex represents black. At the top of the hexcone, colors have their maximum intensity. When  $V = 1$  and  $S = 1$ , we have the "pure" hues. White is the point at  $V = 1$  and  $S = 0$ .

This is a more intuitive model for most users. Starting with a selection for a pure hue, which specifies the hue angle  $H$  and sets  $V = S = 1$ , we describe the color we want in terms of adding either white or black to the pure hue. Adding black decreases the setting for  $V$  while  $S$  is held constant. To get a dark blue,  $V$  could be set to 0.4 with  $S = 1$  and  $H = 240^\circ$ . Similarly, when white is to be added to the hue selected, parameter  $S$  is decreased while keeping  $V$  constant. A light blue could be designated with  $S = 0.7$  while  $V = 1$  and  $H = 240^\circ$ . By adding some black and some white, we decrease both  $V$  and  $S$ . An interface for this model typically presents the HSV parameter choices in a color palette.

Color concepts associated with the terms shades, tints, and tones are represented in a cross-sectional plane of the HSV hexcone (Fig. 15-17). Adding black to a pure hue decreases  $V$  down the side of the hexcone. Thus, various shades are represented with values  $S = 1$  and  $0 \leq V \leq 1$ . Adding white to a pure tone produces different tints across the top plane of the hexcone, where parameter values are  $V = 1$  and  $0 \leq S \leq 1$ . Various tones are specified by adding both black and white, producing color points within the triangular cross-sectional area of the hexcone.

The human eye can distinguish about 128 different hues and about 130 different tints (saturation levels). For each of these, a number of shades (value settings) can be detected, depending on the hue selected. About 23 shades are discernible with yellow colors, and about 16 different shades can be seen at the blue end of the spectrum. This means that we can distinguish about  $128 \times 130 \times 23 = 82,720$  different colors. For most graphics applications, 128 hues, 8 saturation levels, and 15 value settings are sufficient. With this range of parameters in the HSV color model, 16,384 colors would be available to a user, and the system would need 14 bits of color storage per pixel. Color lookup tables could be used to reduce the storage requirements per pixel and to increase the number of available colors.

---

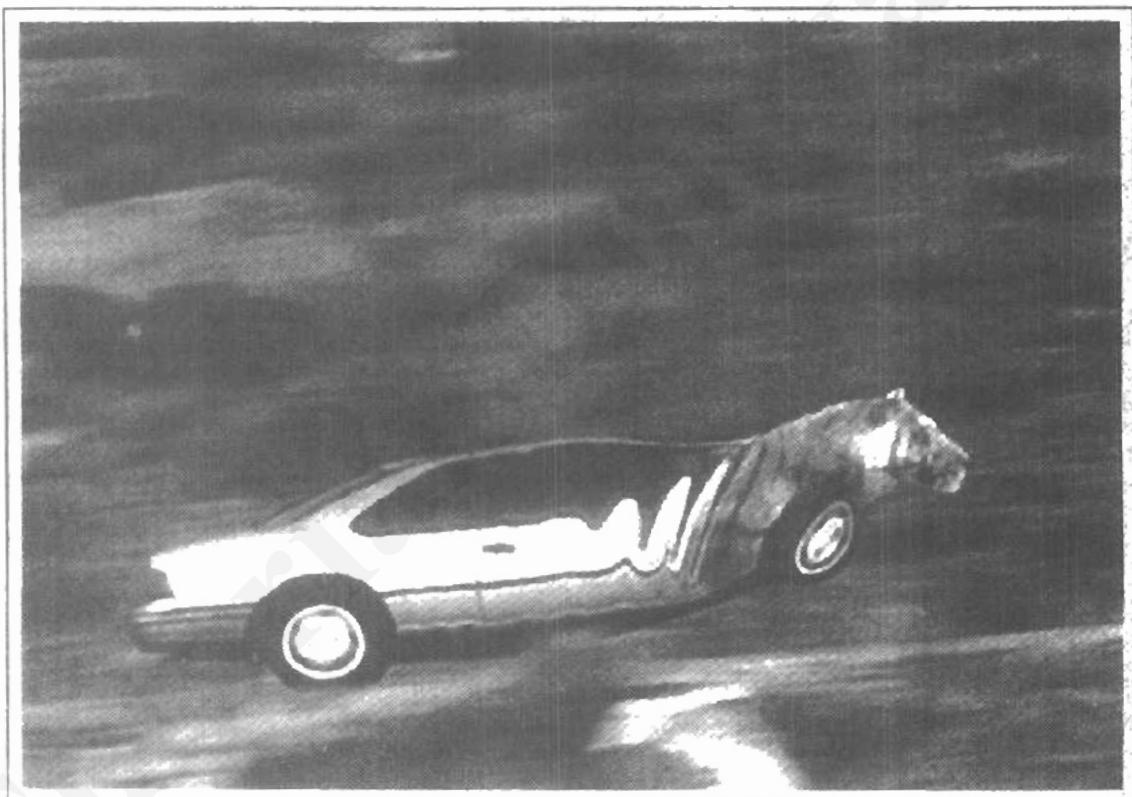
## EXERCISES

---

- 15-1. Derive expressions for converting RGB color parameters to HSV values.
- 15-2. Derive expressions for converting HSV color values to RGB values.
- 15-3. Write an interactive procedure that allows selection of HSV color parameters from a displayed menu, then the HSV values are to be converted to RGB values for storage in a frame buffer.
- 15-4. Derive expressions for converting RGB color values to HLS color parameters.
- 15-5. Derive expressions for converting HLS color values to RGB values.
- 15-6. Write a program that allows interactive selection of HLS values from a color menu then converts these values to corresponding RGB values.
- 15-7. Write a program that will produce a set of colors that are linearly interpolated between any two specified positions in RGB space.
- 15-8. Write an interactive routine for selecting color values from within a specified subspace of RGB space.
- 15-9. Write a program that will produce a set of colors that are linearly interpolated between any two specified positions in HSV space.
- 15-10. Write a program that will produce a set of colors that are linearly interpolated between any two specified positions in HLS space.
- 15-11. Display two RGB color grids, side by side on a video monitor. Fill one grid with a set of randomly selected RGB colors, and fill the other grid with a set of colors that are selected from a small RGB subspace. Experiment with different random selections and different RGB subspaces and compare the two color grids.
- 15-12. Display the two color grids in Exercise 15-11 using color selections from either the HSV or the HLS color space.

CHAPTER —

# 16 Computer Animation



---

---

**S**ome typical applications of computer-generated animation are entertainment (motion pictures and cartoon), advertising, scientific and engineering studies, and training and education. Although we tend to think of animation as implying object motions, the term **computer animation** generally refers to any time sequence of visual changes in a scene. In addition to changing object position with translations or rotations, a computer-generated animation could display time variations in object size, color, transparency, or surface texture. Advertising animations often transition one object shape into another; for example, transforming a can of motor oil into an automobile engine. Computer animations can also be generated by changing camera parameters, such as position, orientation, and focal length. And we can produce computer animations by changing lighting effects or other parameters and procedures associated with illumination and rendering.

Many applications of computer animation require realistic displays. An accurate representation of the shape of a thunderstorm or other natural phenomena described with a numerical model is important for evaluating the reliability of the model. Also, simulators for training aircraft pilots and heavy-equipment operators must produce reasonably accurate representations of the environment. Entertainment and advertising applications, on the other hand, are sometimes more interested in visual effects. Thus, scenes may be displayed with exaggerated shapes and unrealistic motions and transformations. There are many entertainment and advertising applications that do require accurate representations for computer-generated scenes. And in some scientific and engineering studies, realism is not a goal. For example, physical quantities are often displayed with pseudo-colors or abstract shapes that change over time to help the researcher understand the nature of the physical process.

### 16-1

#### DESIGN OF ANIMATION SEQUENCES

---

In general, an animation sequence is designed with the following steps:

- Storyboard layout
- Object definitions
- Key-frame specifications
- Generation of in-between frames

This standard approach for animated cartoons is applied to other animation applications as well, although there are many special applications that do not follow this sequence. Real-time computer animations produced by flight simulators, for instance, display motion sequences in response to settings on the aircraft controls. And visualization applications are generated by the solutions of the numerical models. For *frame-by-frame animation*, each frame of the scene is separately generated and stored. Later, the frames can be recorded on film or they can be consecutively displayed in "real-time playback" mode.

The *storyboard* is an outline of the action. It defines the motion sequence as a set of basic events that are to take place. Depending on the type of animation to be produced, the storyboard could consist of a set of rough sketches or it could be a list of the basic ideas for the motion.

An *object definition* is given for each participant in the action. Objects can be defined in terms of basic shapes, such as polygons or splines. In addition, the associated movements for each object are specified along with the shape.

A *key frame* is a detailed drawing of the scene at a certain time in the animation sequence. Within each key frame, each object is positioned according to the time for that frame. Some key frames are chosen at extreme positions in the action; others are spaced so that the time interval between key frames is not too great. More key frames are specified for intricate motions than for simple, slowly varying motions.

*In-betweens* are the intermediate frames between the key frames. The number of in-betweens needed is determined by the media to be used to display the animation. Film requires 24 frames per second, and graphics terminals are refreshed at the rate of 30 to 60 frames per second. Typically, time intervals for the motion are set up so that there are from three to five in-betweens for each pair of key frames. Depending on the speed specified for the motion, some key frames can be duplicated. For a 1-minute film sequence with no duplication, we would need 1440 frames. With five in-betweens for each pair of key frames, we would need 288 key frames. If the motion is not too complicated, we could space the key frames a little farther apart.

There are several other tasks that may be required, depending on the application. They include motion verification, editing, and production and synchronization of a soundtrack. Many of the functions needed to produce general animations are now computer-generated. Figures 16-1 and 16-2 show examples of computer-generated frames for animation sequences.

#### Section 16-1

##### Design of Animation Sequences

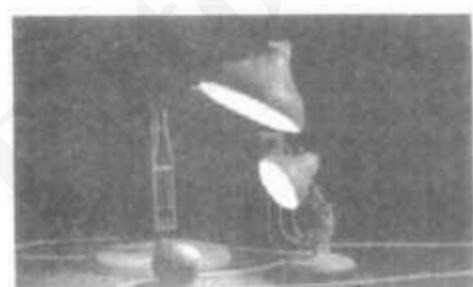


Figure 16-2  
One frame from the award-winning computer-animated short film Luxo Jr. The film was designed using a key-frame animation system and cartoon animation techniques to provide lifelike actions of the lamps. Final images were rendered with multiple light sources and procedural texturing techniques. (Courtesy of Pixar. © 1986 Pixar.)



Figure 16-2  
One frame from the short film *Toy Story*, the first computer-animated film to win an Oscar. Designed using a key-frame animation system, the film also required extensive facial expression modeling. Final images were rendered using procedural shading, self-shadowing techniques, motion blur, and texture mapping. (Courtesy of Pixar. © 1995 Pixar.)

## 16-2

### GENERAL COMPUTER-ANIMATION FUNCTIONS

Some steps in the development of an animation sequence are well-suited to computer solution. These include object manipulations and rendering, camera motions, and the generation of in-betweens. Animation packages, such as WaveFront, for example, provide special functions for designing the animation and processing individual objects.

One function available in animation packages is provided to store and manage the object database. Object shapes and associated parameters are stored and updated in the database. Other object functions include those for motion generation and those for object rendering. Motions can be generated according to specified constraints using two-dimensional or three-dimensional transformations. Standard functions can then be applied to identify visible surfaces and apply the rendering algorithms.

Another typical function simulates camera movements. Standard motions are zooming, panning, and tilting. Finally, given the specification for the key frames, the in-betweens can be automatically generated.

## 16-3

### RASTER ANIMATIONS

On raster systems, we can generate real-time animation in limited applications using *raster operations*. As we have seen in Section 5-8, a simple method for translation in the *xy* plane is to transfer a rectangular block of pixel values from one location to another. Two-dimensional rotations in multiples of 90° are also simple to perform, although we can rotate rectangular blocks of pixels through arbitrary angles using antialiasing procedures. To rotate a block of pixels, we need to determine the percent of area coverage for those pixels that overlap the rotated block. Sequences of raster operations can be executed to produce real-time animation of either two-dimensional or three-dimensional objects, as long as we restrict the animation to motions in the projection plane. Then no viewing or visible-surface algorithms need be invoked.

We can also animate objects along two-dimensional motion paths using the *color-table transformations*. Here we predefine the object at successive positions along the motion path, and set the successive blocks of pixel values to color-table

#### Section 16-4

##### Computer-Animation Languages

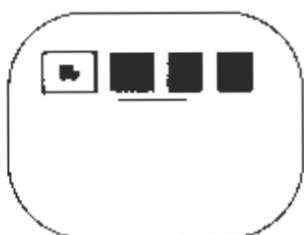


Figure 16-3  
Real-time raster color-table  
animation.

entries. We set the pixels at the first position of the object to "on" values, and we set the pixels at the other object positions to the background color. The animation is then accomplished by changing the color-table values so that the object is "on" at successively positions along the animation path as the preceding position is set to the background intensity (Fig. 16-3).

#### 16-4

##### COMPUTER-ANIMATION LANGUAGES

Design and control of animation sequences are handled with a set of animation routines. A general-purpose language, such as C, Lisp, 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 surfaces, constructive 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 (light-source intensities and surface-illumination properties), and setting the camera parameters (position, orientation, and lens characteristics). Another standard function is *action specification*. This involves the layout of motion paths for the objects 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 specifications, 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. As an example, the single-arm robot in Fig. 16-4 has six degrees of freedom, which are called arm sweep, shoulder swivel, elbow extension, pitch, yaw, and roll. We can extend the number of degrees of freedom for this robot arm to nine by allowing three-dimensional translations for the base (Fig. 16-5). If we also allow base rotations, the robot arm can have a total of 12 degrees of freedom. The human body, in comparison, has over 200 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 degrees of freedom, motion limitations, and allowable shape changes.

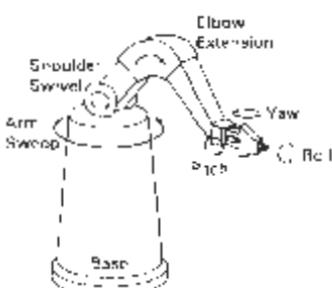


Figure 16-4  
Degrees of freedom for a stationary, single-arm robot



Figure 16-5  
Translational and rotational degrees of freedom for the base of the robot arm

Scripting systems allow object specifications and animation sequences to be defined with a user-input script. From the script, a library of various objects and motions can be constructed.

## 16-5

### KEY-FRAME SYSTEMS

We generate each set of in-betweens from the specification of two (or more) key frames. Motion paths can be given with a *kinematic* description as a set of spline curves, or the motions can be physically based by specifying the forces acting on the objects to be animated.

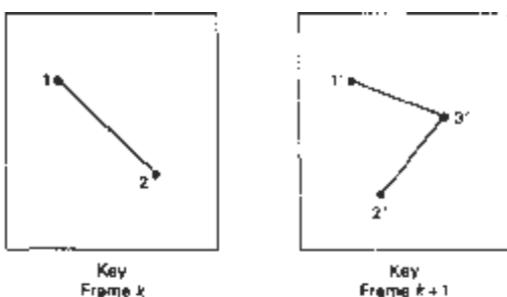
For complex scenes, we can separate the frames into individual components or objects called *cels* (refuloid transparencies), an acronym from cartoon animation. Given the animation paths, we can interpolate the positions of individual objects between any two times.

With complex object transformations, the shapes of objects may change over time. Examples are clothes, facial features, magnified detail, evolving shapes, exploding or disintegrating objects, and transforming one object into another object. If all surfaces are described with polygon meshes, then the number of edges per polygon can change from one frame to the next. Thus, the total number of line segments can be different in different frames.

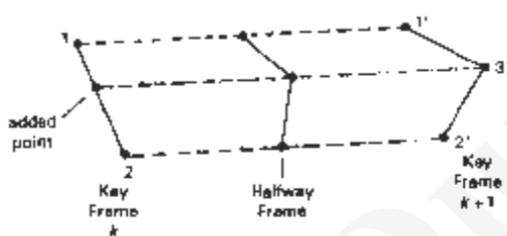
#### Morphing

Transformation of object shapes from one form to another is called **morphing**, which is a shortened form of metamorphosis. Morphing methods can be applied to any motion or transition involving a change in shape.

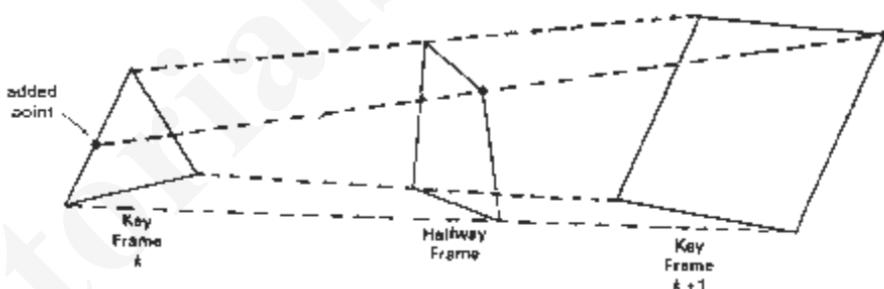
Given two key frames for an object transformation, we first adjust the object specification in one of the frames so that the number of polygon edges (or the number of vertices) is the same for the two frames. This preprocessing step is illustrated in Fig. 16-6. A straight-line segment in key frame  $k$  is transformed into two line segments in key frame  $k+1$ . Since key frame  $k+1$  has an extra vertex, we add a vertex between vertices 1 and 2 in key frame  $k$  to balance the number of vertices (and edges) in the two key frames. Using linear interpolation to generate the in-betweens, we transition the added vertex in key frame  $k$  into vertex 3 along the straight-line path shown in Fig. 16-7. An example of a triangle linearly expanding into a quadrilateral is given in Fig. 16-8. Figures 16-9 and 16-10 show examples of morphing in television advertising.



**Figure 16-6**  
An edge with vertex positions 1 and 2 in key frame  $k$  evolves into two connected edges in key frame  $k + 1$ .



**Figure 16-7**  
Linear interpolation for transforming a line segment in key frame  $k$  into two connected line segments in key frame  $k + 1$ .



**Figure 16-8**  
Linear interpolation for transforming a triangle into a quadrilateral.

We can state general preprocessing rules for equalizing key frames in terms of either the number of edges or the number of vertices to be added to a key frame. Suppose we equalize the edge count, and parameters  $L_k$  and  $L_{k+1}$  denote the number of line segments in two consecutive frames. We then define

$$L_{\max} = \max(L_k, L_{k+1}), \quad L_{\min} = \min(L_k, L_{k+1}) \quad (16-1)$$

and

$$N_e = L_{\max} \bmod L_{\min}$$

$$N_s = \text{int}\left(\frac{L_{\max}}{L_{\min}}\right) \quad (16-2)$$



(a)



(b)



(c)



(d)



(e)

**Figure 16-9**  
Transformation of an STP oil can into an engine block. (Courtesy of Silicon Graphics, Inc.)



(a)



(b)



(c)



(d)

**Figure 16-10**  
Transformation of a moving automobile into a running tiger. (Courtesy of Exxon Company USA and Pacific Data Images.)

Then the preprocessing is accomplished by

---

### Section 16-5

#### Key-Frame Systems

---

1. dividing  $N_e$  edges of  $\text{keyframe}_{\text{use}}$  into  $N_s + 1$  sections
2. dividing the remaining lines of  $\text{keyframe}_{\text{tran}}$  into  $N_t$  sections

As an example, if  $L_i = 15$  and  $L_{i+1} = 11$ , we would divide 4 lines of  $\text{keyframe}_{i+1}$  into 2 sections each. The remaining lines of  $\text{keyframe}_{i+1}$  are left intact.

If we equalize the vertex count, we can use parameters  $V_k$  and  $V_{k+1}$  to denote the number of vertices in the two consecutive frames. In this case, we define

$$V_{\max} = \max(V_k, V_{k+1}), \quad V_{\min} = \min(V_k, V_{k+1}) \quad (16-1)$$

and

$$\begin{aligned} N_b &= (V_{\max} - 1) \bmod (V_{\min} - 1) \\ N_p &= \text{int}\left(\frac{V_{\max} - 1}{V_{\min} - 1}\right) \end{aligned} \quad (16-2)$$

Preprocessing using vertex count is performed by

1. adding  $N_p$  points to  $N_b$  line sections of  $\text{keyframe}_{\text{use}}$ .
2. adding  $N_p - 1$  points to the remaining edges of  $\text{keyframe}_{\text{tran}}$ .

For the triangle-to-quadrilateral example,  $V_k = 3$  and  $V_{k+1} = 4$ . Both  $N_b$  and  $N_p$  are 1, so we would add one point to one edge of  $\text{keyframe}_k$ . No points would be added to the remaining lines of  $\text{keyframe}_{k+1}$ .

#### Simulating Accelerations

Curve-fitting techniques are often used to specify the animation paths between key frames. Given the vertex positions at the key frames, we can fit the positions with linear or nonlinear paths. Figure 16-11 illustrates a nonlinear fit of key-frame positions. This determines the trajectories for the in-betweens. To simulate accelerations, we can adjust the time spacing for the in-betweens.

For constant speed (zero acceleration), we use equal-interval time spacing for the in-betweens. Suppose we want  $n$  in-betweens for key frames at times  $t_1$  and  $t_2$  (Fig. 16-12). The time interval between key frames is then divided into  $n + 1$  subintervals, yielding an in-between spacing of

$$\Delta t = \frac{t_2 - t_1}{n + 1} \quad (16-3)$$

We can calculate the time for any in-between as

$$tR_j = t_1 + j \Delta t, \quad j = 1, 2, \dots, n \quad (16-4)$$

and determine the values for coordinate positions, color, and other physical parameters.

Nonzero accelerations are used to produce realistic displays of speed changes, particularly at the beginning and end of a motion sequence. We can model the start-up and slow-down portions of an animation path with spline or

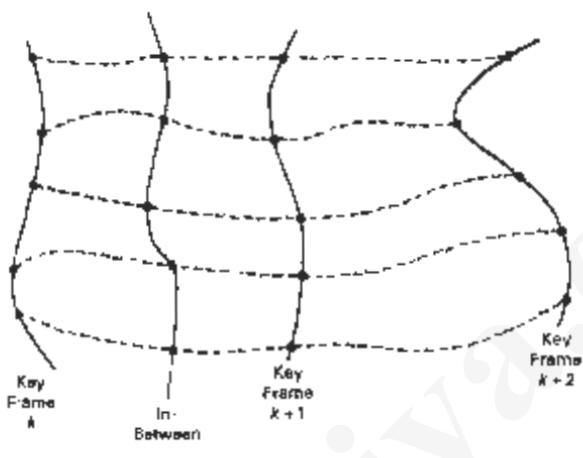


Figure 16-12  
Fitting key-frame vertex positions with nonlinear splines.

trigonometric functions. Parabolic and cubic time functions have been applied to acceleration modeling, but trigonometric functions are more commonly used in animation packages.

To model increasing speed (positive acceleration), we want the time spacing between frames to increase so that greater changes in position occur as the object moves faster. We can obtain an increasing interval size with the function

$$t = \cos \theta, \quad 0 < \theta < \pi/2$$

For  $n$  in-betweens, the time for the  $j$ th in-between would then be calculated as

$$tB_j = t_1 + \Delta t \left[ 1 - \cos \frac{j\pi}{2(n+1)} \right], \quad j = 1, 2, \dots, n \quad (16-7)$$

where  $\Delta t$  is the time difference between the two key frames. Figure 16-13 gives a plot of the trigonometric acceleration function and the in-between spacing for  $n = 5$ .

We can model decreasing speed (deceleration) with  $\sin \theta$  in the range  $0 < \theta < \pi/2$ . The time position of an in-between is now defined as

$$tB_j = t_1 + \Delta t \sin \frac{j\pi}{2(n+1)}, \quad j = 1, 2, \dots, n \quad (16-8)$$

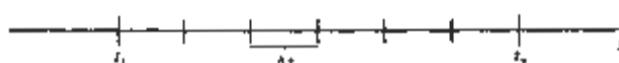


Figure 16-13  
In-between positions for motion at constant speed

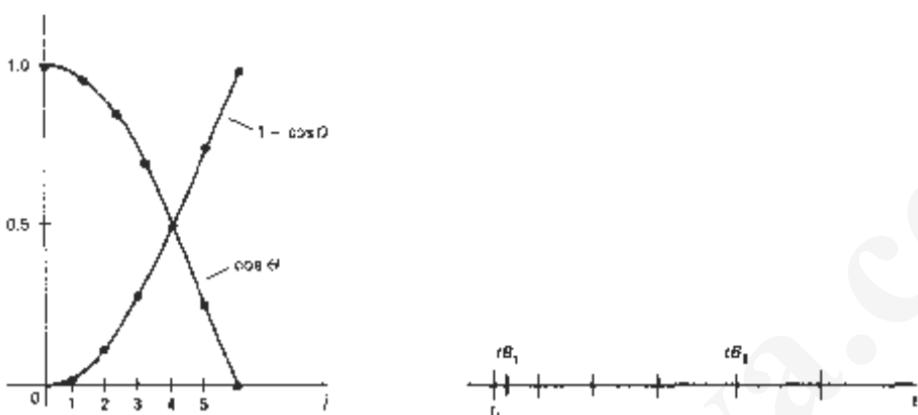


Figure 16-13

A trigonometric acceleration function and the corresponding in-between spacing for  $n = 5$  and  $\theta = j\pi/12$  in Eq. 16-7, producing increased coordinate changes as the object moves through each time interval.

A plot of this function and the decreasing size of the time intervals is shown in Fig. 16-14 for five in-betweens.

Often, motions contain both speed-ups and slow-downs. We can model a combination of increasing-decreasing speed by first increasing the in-between time spacing, then we decrease this spacing. A function to accomplish these time changes is

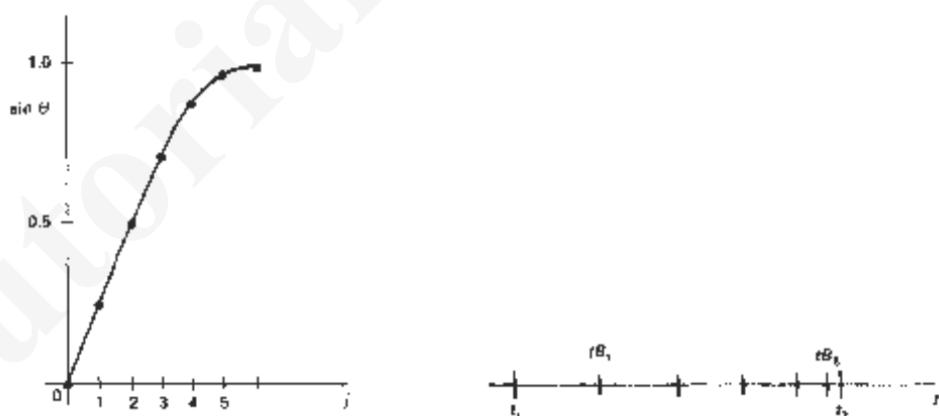
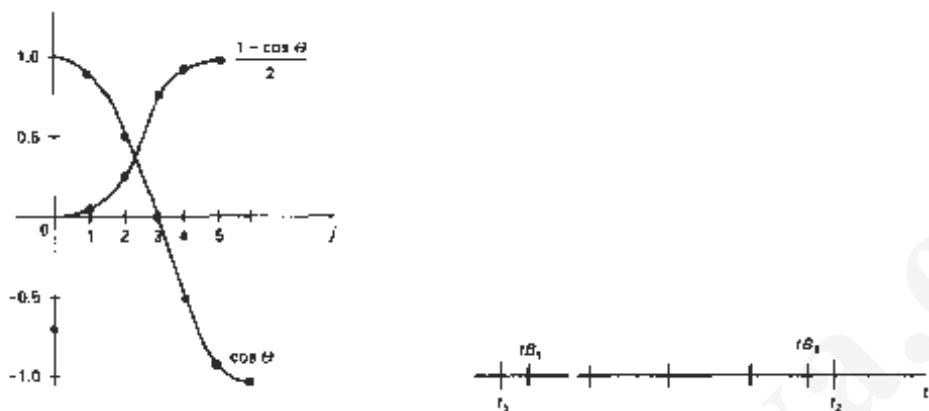


Figure 16-14

A trigonometric deceleration function and the corresponding in-between spacing for  $n = 5$  and  $\theta = j\pi/12$  in Eq. 16-8, producing decreased coordinate changes as the object moves through each time interval.



**Figure 16-15**  
A trigonometric accelerate-decelerate function and the corresponding in-between spacing for  $n = 5$  in Eq. 16-9.

$$\frac{1}{2}(1 - \cos \theta), \quad 0 < \theta < \pi/2$$

The time for the  $j$ th in-between is now calculated as

$$t_B_j = t_1 + \Delta t \left\{ \frac{1 - \cos[j\pi/(n+1)]}{2} \right\}, \quad j = 1, 2, \dots, n \quad (16-9)$$

with  $\Delta t$  denoting the time difference for the two key frames. Time intervals for the moving object first increase, then the time intervals decrease, as shown in Fig. 16-15.

Processing the in-betweens is simplified by initially modeling "skeleton" (wireframe) objects. This allows interactive adjustment of motion sequences. After the animation sequence is completely defined, objects can be fully rendered.

## 16-6 MOTION SPECIFICATIONS

There are several ways in which the motions of objects can be specified in an animation system. We can define motions in very explicit terms, or we can use more abstract or more general approaches.

### Direct Motion Specification

The most straightforward method for defining a motion sequence is direct specification of the motion parameters. Here, we explicitly give the rotation angles and translation vectors. Then the geometric transformation matrices are applied to transform coordinate positions. Alternatively, we could use an approximating

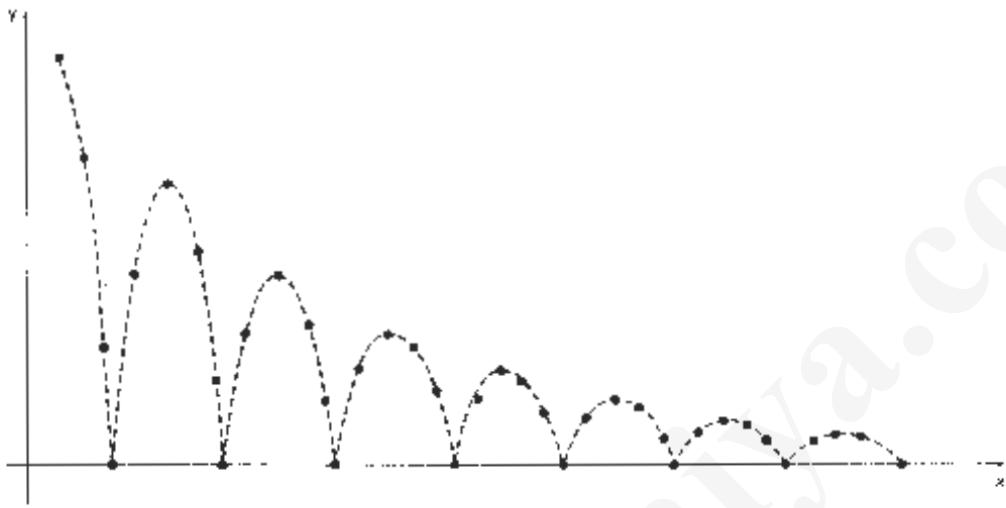


Figure 16-16

Approximating the motion of a bouncing ball with a damped sine function. (Eq. 16-10).

equation to specify certain kinds of motions. We can approximate the path of a bouncing ball, for instance, with a damped, rectified, sine curve (Fig. 16-16):

$$y(x) = A |\sin(\omega x - \theta_0)| e^{-kx} \quad (16-10)$$

where  $A$  is the initial amplitude,  $\omega$  is the angular frequency,  $\theta_0$  is the phase angle, and  $k$  is the damping constant. These methods can be used for simple user-programmed animation sequences.

#### Goal-Directed Systems

At the opposite extreme, we can specify the motions that are to take place in general terms that abstractly describe the actions. These systems are referred to as *goal directed* because they determine specific motion parameters given the goals of the animation. For example, we could specify that we want an object to "walk" or to "run" to a particular destination. Or we could state that we want an object to "pick up" some other specified object. The input directives are then interpreted in terms of component motions that will accomplish the selected task. Human motions, for instance, can be defined as a hierarchical structure of sub-motions for the torso, limbs, and so forth.

#### Kinematics and Dynamics

We can also construct animation sequences using *kinematic* or *dynamical* descriptions. With a kinematic description, we specify the animation by giving motion parameters (position, velocity, and acceleration) without reference to the forces that cause the motion. For constant velocity (zero acceleration), we designate the motions of rigid bodies in a scene by giving an initial position and velocity vector

for each object. As an example, if a velocity is specified as  $(3, 0, -4)$  km/sec, then this vector gives the direction for the straight-line motion path and the speed (magnitude of velocity) is 5 km/sec. If we also specify accelerations (rate of change of velocity), we can generate speed-ups, slow-downs, and curved motion paths. Kinematic specification of a motion can also be given by simply describing the motion path. This is often done using spline curves.

An alternate approach is to use *inverse kinematics*. Here, we specify the initial and final positions of objects at specified times and the motion parameters are computed by the system. For example, assuming zero accelerations, we can determine the constant velocity that will accomplish the movement of an object from the initial position to the final position. This method is often used with complex objects by giving the positions and orientations of an end node of an object, such as a hand or a foot. The system then determines the motion parameters of other nodes to accomplish the desired motion.

Dynamic descriptions on the other hand, require the specification of the forces that produce the velocities and accelerations. Descriptions of object behavior under the influence of forces are generally referred to as a *physically based modeling* (Chapter 10). Examples of forces affecting object motion include electromagnetic, gravitational, friction, and other mechanical forces.

Object motions are obtained from the force equations describing physical laws, such as Newton's laws of motion, for gravitational and friction processes, Euler or Navier-Stokes equations describing fluid flow, and Maxwell's equations for electromagnetic forces. For example, the general form of Newton's second law for a particle of mass  $m$  is

$$\mathbf{F} = \frac{d}{dt}(mv) \quad (7e-13)$$

with  $\mathbf{F}$  as the force vector, and  $\mathbf{v}$  as the velocity vector. If mass is constant, we solve the equation  $\mathbf{F} = m\mathbf{a}$ , where  $\mathbf{a}$  is the acceleration vector. Otherwise, mass is a function of time, as in relativistic motions or the motions of space vehicles that consume measurable amounts of fuel per unit time. We can also use *inverse dynamics* to obtain the forces, given the initial and final positions of objects and the type of motion.

Applications of physically based modeling include complex rigid-body systems and such nonrigid systems as cloth and plastic materials. Typically, numerical methods are used to obtain the motion parameters incrementally from the dynamical equations using initial conditions or boundary values.

## SUMMARY

A computer-animation sequence can be set up by specifying the storyboard, the object definitions, and the key frames. The storyboard is an outline of the action, and the key frames define the details of the object motions for selected positions in the animation. Once the key frames have been established, a sequence of in-betweens can be generated to construct a smooth motion from one key frame to the next. A computer animation can involve motion specifications for the objects in a scene as well as motion paths for a camera that moves through the scene. Computer-animation systems include key-frame systems, parameterized systems, and scripting systems. For motion in two-dimensions, we can use the raster-animation techniques discussed in Chapter 5.

For some applications, key frames are used to define the steps in a morphing sequence that changes one object shape into another. Other in-between methods include generation of variable time intervals to simulate accelerations and decelerations in the motion.

---

#### Exercises

Motion specifications can be given in terms of translation and rotation parameters, or motions can be described with equations or with kinematic or dynamic parameters. Kinematic motion descriptions specify positions, velocities, and accelerations. Dynamic motion descriptions are given in terms of the forces acting on the objects in a scene.

---

## REFERENCES

For additional information on computer animation systems and techniques, see Magnenat Thalmann and Thalmann (1985), Barzel (1992), and Watt and Watt (1992). Algorithms for animation applications are presented in Glassner (1990), Arvo (1991), Kirk (1992), Gascuel (1994), Ngu and Marks (1993), van de Panne and Fiume (1993), and in Snyder et al. (1993). Morphing techniques are discussed in Beer and Neely (1992), Hughes (1992), Kent, Carson, and Laurent (1992), and in Sederberg and Greenwood (1992). A discussion of animation techniques in PLIBGS is given in Cawthon (1992).

---

## EXERCISES

- 16-1. Design a storyboard layout and accompanying key frames for an animation of a single polyhedron.
- 16-2. Write a program to generate the in-betweens for the key frames specified in Exercise 16-1 using linear interpolation.
- 16-3. Expand the animation sequence in Exercise 16-2 to include two or more moving objects.
- 16-4. Write a program to generate the in-betweens for the key frames in Exercise 16-3 using linear interpolation.
- 16-5. Write a morphing program to transform a sphere into a specified polyhedron.
- 16-6. Set up an animation specification involving accelerations and implement Eq. 16-7.
- 16-7. Set up an animation specification involving both accelerations and decelerations and implement the in-between spacing calculations given in Eqs. 16-7 and 16-8.
- 16-8. Set up an animation specification implementing the acceleration/deceleration calculations of Eq. 16-9.
- 16-9. Write a program to simulate the linear, two-dimensional motion of a filled circle inside a given rectangular area. The circle is to be given an initial velocity, and the circle is to rebound from the walls with the angle of reflection equal to the angle of incidence.
- 16-10. Convert the program of Exercise 16-9 into a ball-and-paddle game by replacing one side of the rectangle with a short line segment that can be moved back and forth to intercept the circle path. The game is over when the circle escapes from the interior of the rectangle. Initial input parameters include circle position, direction, and speed. The game score can include the number of times the circle is intercepted by the paddle.
- 16-11. Expand the program of Exercise 16-9 to simulate the three-dimensional motion of a sphere moving inside a parallelepiped. Interactive viewing parameters can be set to view the motion from different directions.
- 16-12. Write a program to implement the simulation of a bouncing ball using Eq. 16-10.
- 16-13. Write a program to implement the motion of a bouncing ball using a downward

gravitational force and a ground-plane friction force. Initially, the ball is to be projected into space with a given velocity vector.

- 16-14. Write a program to implement the two-player pillbox game. The game can be implemented on a flat plane with fixed pillbox positions, or random terrain features and pillbox placements can be generated at the start of the game.
- 16-15. Write a program to implement dynamic motion specifications. Specify a scene with two or more objects initial motion parameters, and specified forces. Then generate the animation from the solution of the force equations. (For example, the objects could be the earth, moon, and sun with attractive gravitational forces that are proportional to mass and inversely proportional to distance squared.)

# TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution\* etc of following subjects from <https://www.tutorialsduniya.com>.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- 
- ❖ Compiled Books: <https://www.tutorialsduniya.com/compiled-books>
  - ❖ Programs: <https://www.tutorialsduniya.com/programs>
  - ❖ Question Papers: <https://www.tutorialsduniya.com/question-papers>
  - ❖ Python Notes: <https://www.tutorialsduniya.com/python>
  - ❖ Java Notes: <https://www.tutorialsduniya.com/java>
  - ❖ JavaScript Notes: <https://www.tutorialsduniya.com/javascript>
  - ❖ JSP Notes: <https://www.tutorialsduniya.com/jsp>
  - ❖ Microprocessor Notes: <https://www.tutorialsduniya.com/microprocessor>
  - ❖ OR Notes: <https://www.tutorialsduniya.com/operational-research>